Spring 2024

Gregory Bruich, Ph.D.
Dept. of Economics, Harvard University

Lab 1: Introductory Statistical Concepts and Statistical Computing

Methods/concepts: histograms, means, quantiles, indicator variables, rank transformation, bin scatters, and random assignment

LAB DESCRIPTION

The goal of this first lab is to acquire some familiarity working with data in statistical software. This lab uses an extract from the 1966 and 1979 National Longitudinal Survey called **nls6679.dta**. For more details on the variables included in these data, see Table 1. A list and description of each of the Stata and R commands needed for this lab are contained in Table 2 and Table 3, respectively. You should have these commands next to you as you work through the lab.

QUESTIONS

1. A *histogram* is a simple way to visualize the distribution of a variable in a data set. Strictly speaking, a histogram is drawn so that area represents probability. The height of the rectangles corresponds to the proportion of the data falling into the range indicated by the x-axis divided by the width of the rectangles. The y-axis is in *density units*. The y-axis is scaled so that the area of the rectangles (base times height) adds up to 1. Plot and save a histogram of *kid_income*. Include the image in your lab solutions.

2. The sample *mean* or arithmetic average is the "balance point" of the histogram: "Think of the rectangles as weights and the x-axis of the histogram as a wooden board. Below the wooden board is a steel rod. Move the rod back and forth. Where the board balances is the mean of the histogram" (Brightman 1986). It is often represented with a bar over the variable, such as $\bar{Y}$. It equals the sum of the values of the variable Y_i for each observation divided by the total number of observations: $\bar{Y} = \frac{1}{N}\sum_{i=1}^{N} Y_i$. Calculate and report the sample mean of *kid_income*.

3. What fraction of observations have *kid_income* below its mean? To answer this question:

   a. Generate a new variable called *below_mean* that equals 1 if *kid_income* is (strictly) less than its mean, and 0 if it is greater than or equal to its mean.

   b. The arithmetic average of the *indicator variable* *below_mean* is the fraction of observations with *kid_income* below its mean. Calculate and report the sample mean of *below_mean*.

   c. What features of the histogram from question 1 explains why this fraction is *not* 50%?

4. The sample *median* is the value for which 50% of the observations have a value above the value and 50% have a value below the value. The median is also called the 50th percentile, or sometimes abbreviated p50. Calculate and report the sample median of *kid_income*.

5. The sample *standard deviation* is a measure of spread, sometimes abbreviated SD. It equals the square root of the sample *variance*, which is the sum of the squared differences between the values of the variable Y_i for each observation and its mean $\bar{Y} = \frac{1}{N}\sum_{i=1}^{N} Y_i$ divided by the total

number of observations: $SD = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(Y_i - \bar{Y})^2}$. Calculate and report the sample standard deviation of *kid_income*. Note that the units will be in dollars.

6. An important rule of thumb is that most of the data is usually within one standard deviation of the mean and almost all of the data is usually within two standard deviations of the mean. This is a statement that we could make more precise mathematically, but here we will simply verify this important rule of thumb in these data by using *indicator variables* to calculate two statistics:

    a. What fraction of observations are within one SD of the mean of *kid_income*?

    b. What fraction of observations are within two SDs of the mean of *kid_income*?

    Hint: Use the same trick as you used in question 3 by defining indicator variables and summarizing them to answer part a and part b.

7. An important data transformation is the *percentile rank transformation*, as we saw in Professor Chetty's lecture and the pre-recorded video for this week. In this question, you will generate a new variable *kid_inc_rank* that equals *kid_income* converted to *percentile ranks*, normalized so that the highest rank is 100. To see how this works, proceed in the following steps:

    a. Use the rank() function to generate a new variable that equals each observation's rank based on *kid_income*. Start with just their rank, without any normalization yet.

    b. Next sort the data by *kid_income*. Browse the data and examine which observations are at the top and bottom of the data frame and how this corresponds to their rank and the variable *kid_income*

    c. Now normalize the rank so that the highest rank is 100, generating the new variable *kid_inc_rank*

    d. Finally, browse the data once again and examine *kid_income* and *kid_inc_rank* for observations that are at the top and bottom of the data frame

8. The percentile rank transformation has several useful properties. To see a couple of them:

    a. Plot a histogram of *kid_inc_rank*. Notice that the histogram is approximately uniformly distributed between 0 and 100. Include your graph as an image in your solutions.

    b. Verify that the sample mean of *kid_inc_rank* approximately equals its sample median.

9. Most of social science research is concerned with the relationship *between* two or more variables. Visualize the relationship between *kid_income* (y-axis) and *parent_income* (x-axis) measured in dollars (not ranks) in two ways, commenting on which is a more useful summary of the data and including your graphs in your solutions:

    a. Scatter plot of the individual level data

    b. Binned scatter plot

10. In your judgement, are *kid_income* and *parent_income* *linearly* related or *non-linearly* related? Explain clearly what you see in your graphical analysis that leads you to your conclusions.

11. In social science research, random assignment is the ideal solution to the "fundamental problem of causal inference," because it ensures comparability between treatment and control groups, and eliminates confounding. To conclude this week's lab, we will explore how random assignment works in these data.

    a. First, set the "seed" using your Harvard University ID number (`set seed 12345` in stata or `set.seed(12345)` in R), which will make your simulation reproducible, but different from your classmates' simulations. Then assign to each observation a random number drawn uniformly between 0 and 1.

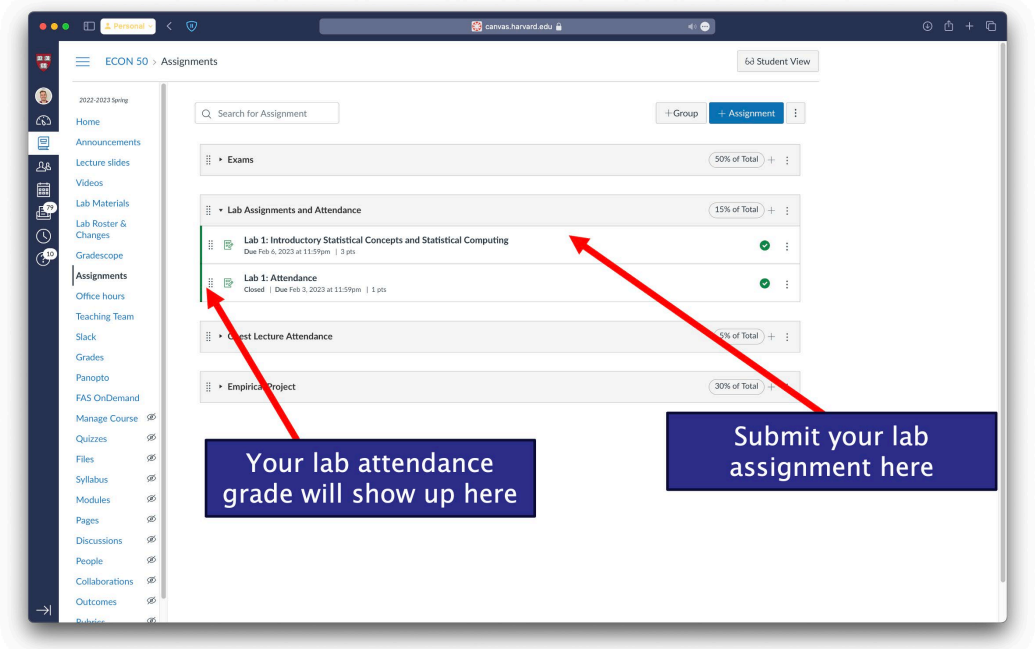
    b. Generate a new variable *treatment_group* that equals 1 ("treatment group") if the number generated in part a is greater than or equal to 0.5, and otherwise equals 0 when the number is less than 0.5 ("control group"). How many observations are in the treatment group? How many are in your control group?

    c. Compute the sample mean and sample standard deviation of all the variables listed in Table 1 separately for observations in your treatment group and your control group.

    d. What is the purpose of random assignment in an experiment? In this simulation, random assignment balances all the variables listed in Table 1. Would you prefer to use random assignment to achieve comparability? Or would you prefer to allocate observations *yourself* to treatment and control groups to ensure that both groups have exactly the same composition of the variables listed in Table 1? Explain why.

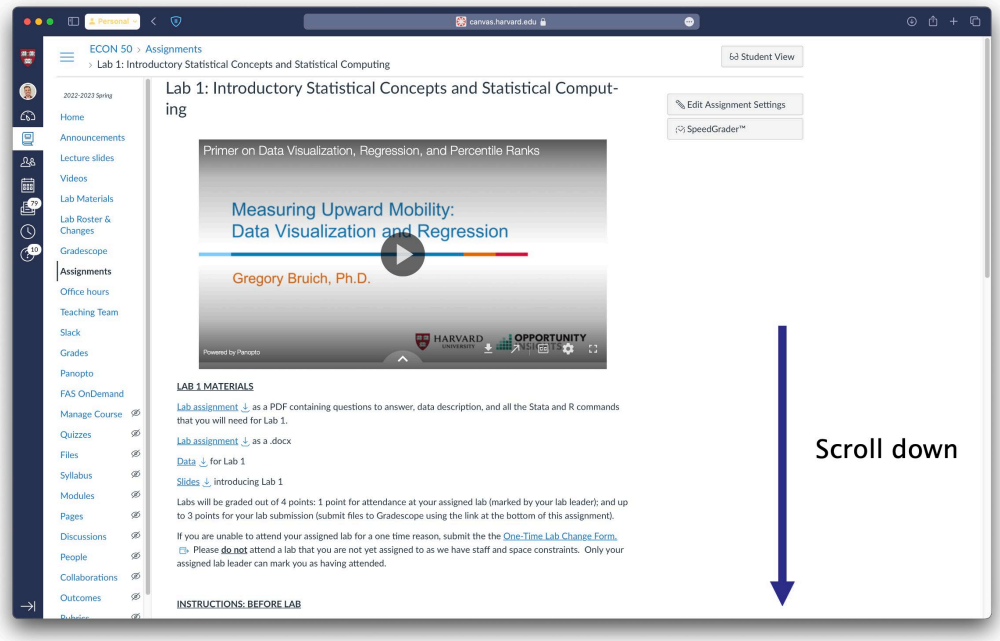
12. Create an annotated/commented do-file, .ipynb Jupyter Notebook, or .R file that can replicate all your analyses above. This will be the final code that you submit on Gradescope. The motivation for using do-files and .R files is described on page 10, which has been adapted from training materials used by Innovations for Poverty Action (IPA) and the Abdul Latif Jameel Poverty Action Lab (J-PAL).

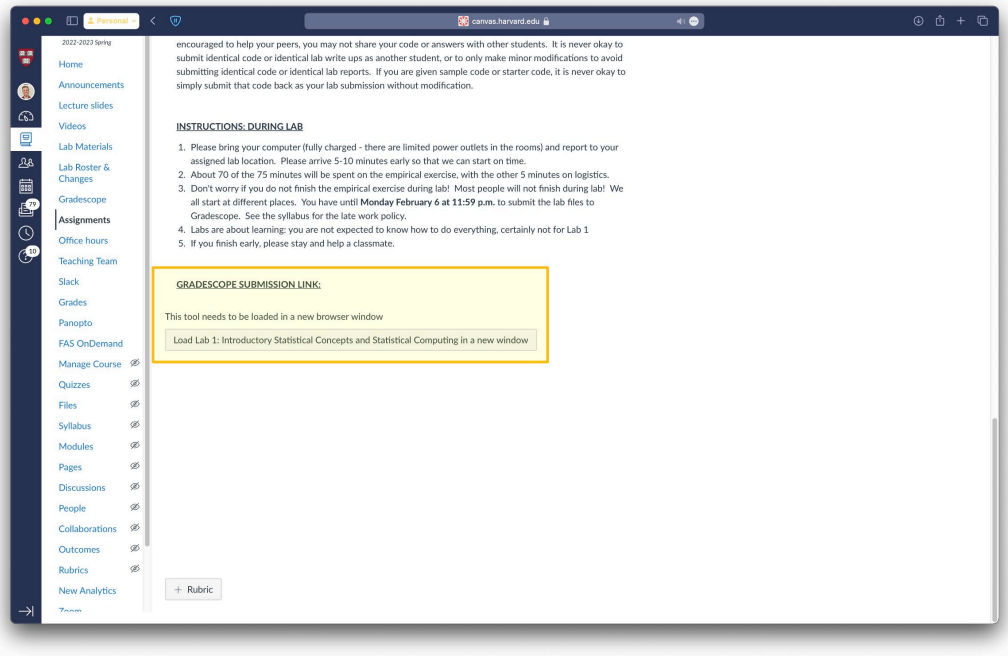
13. Here's how to submit your assignment:

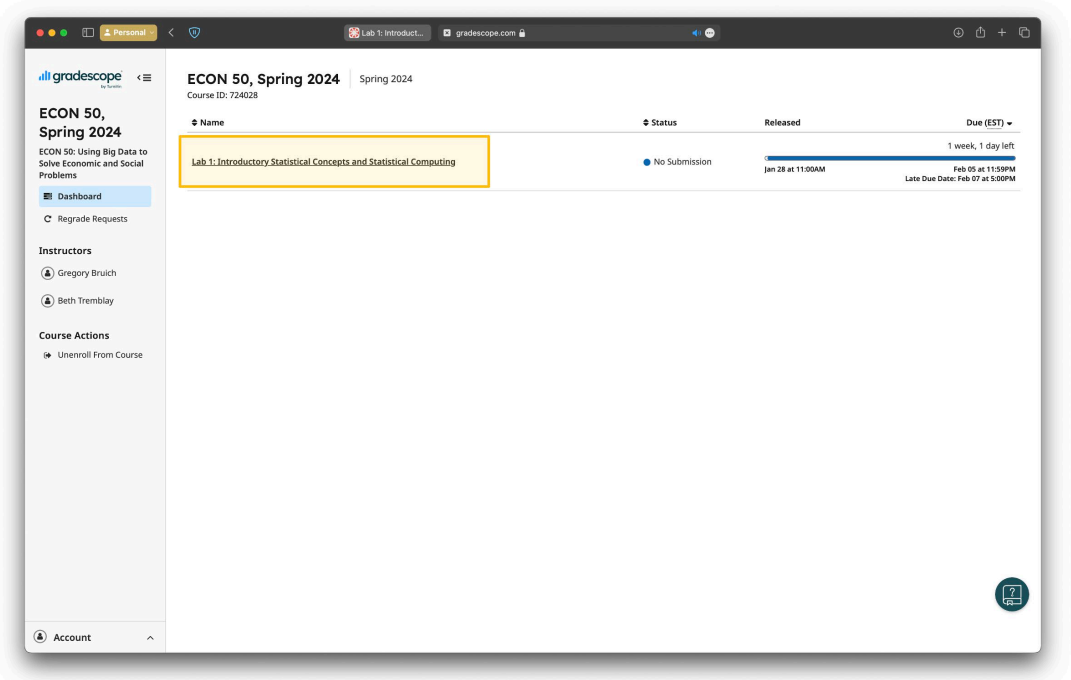
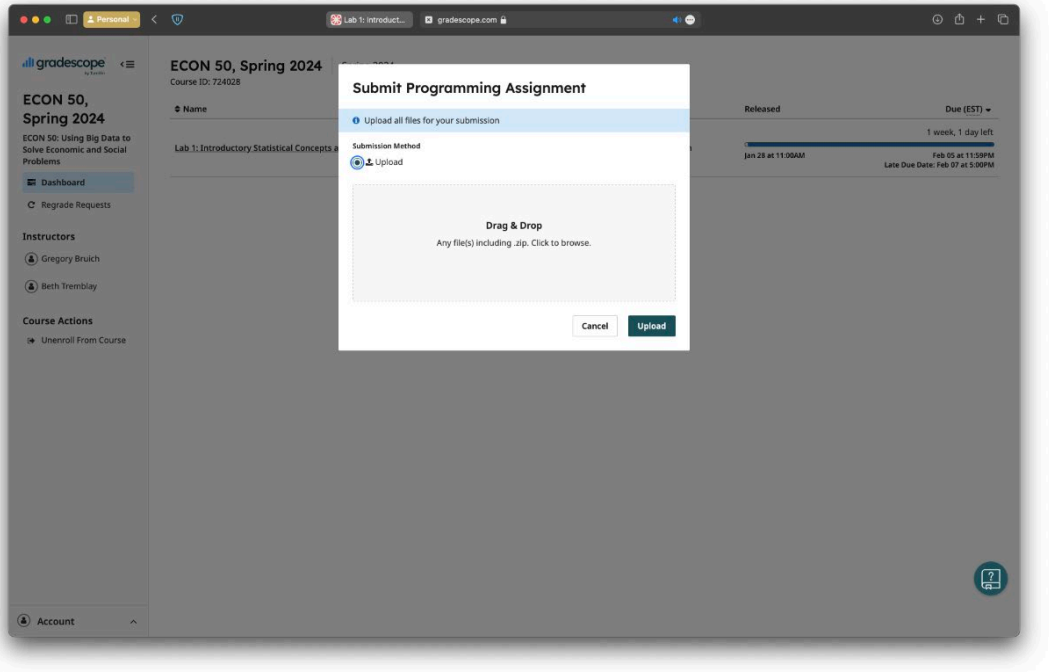
Step 1 Access the lab assignment under the "Assignments" tab on Canvas	

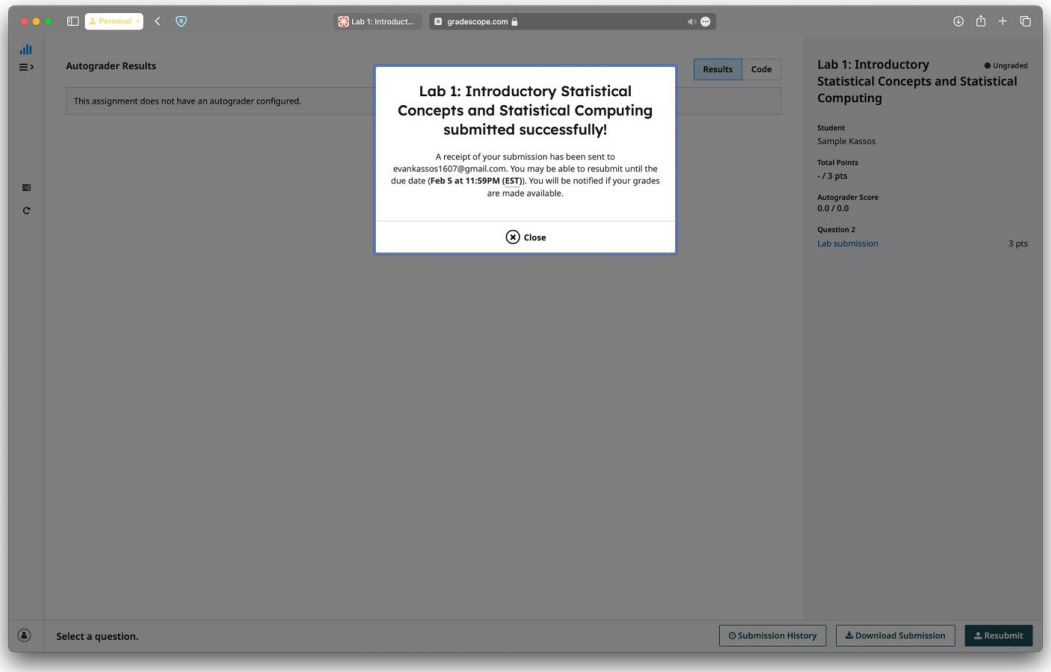
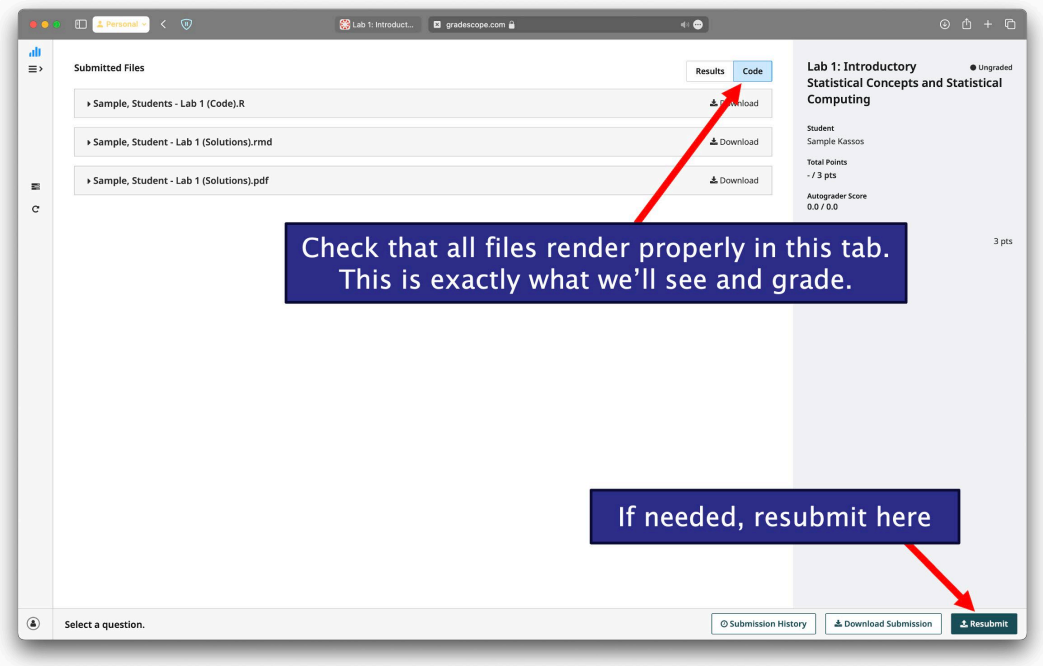
Step 2

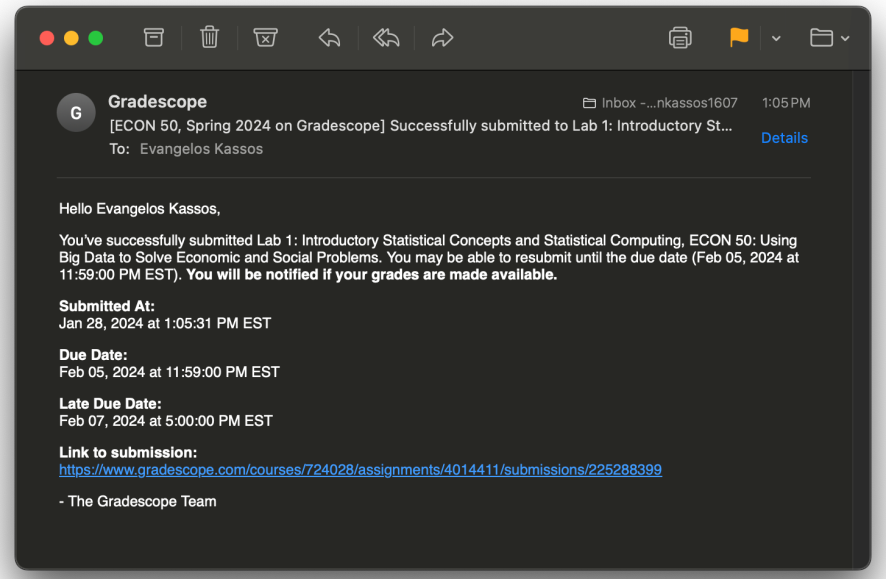
Access Gradescope from Canvas





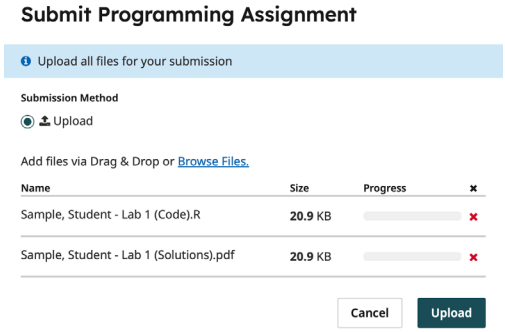
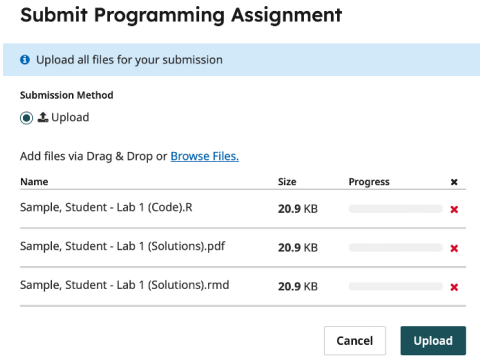
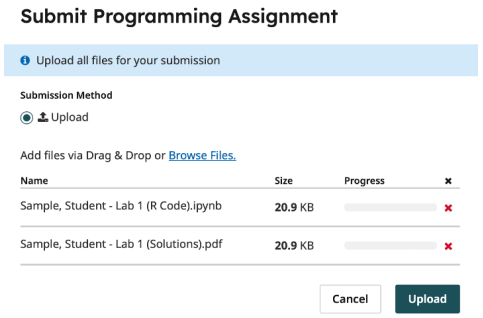
Step 3 Access the lab assignment on Gradescope	
Step 4 Upload your files *Check the table on the next page to confirm what files you need to submit.*	

Step 5 What you'll see after submitting your lab assignment	
Step 6 Check your submitted files	

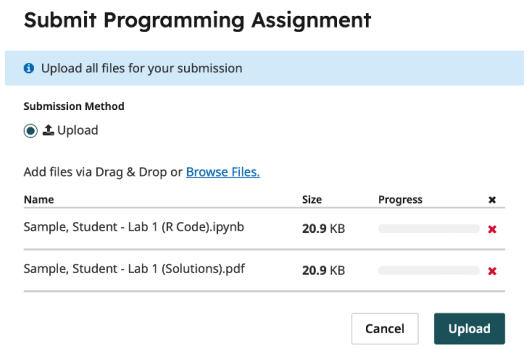
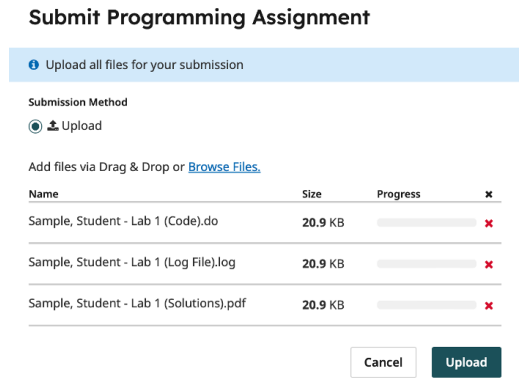
Step 7 You'll receive an email confirmation as well	

Check the table on the next page to confirm what files you need to submit.

WHAT FILES TO SUBMIT

If you're using R to write your code, and a document editor to write your answers 	• Submit your .R code file replicating all your analyses above (with enough comments that a principal investigator on a research project would be able to follow and understand what each step of the code is doing). We need your raw code so that we can run your code ourselves if needed. • Submit your answers as a .pdf file. **Do not submit a .doc/.docx file (Word document)**, as we are unable to read those files on Gradescope. *For graphs, always be sure to save them as .png files and insert them into the answer document even if it was not explicitly asked.* **Important:** If we do not have both your .R solution and .pdf answer files, you will lose 1 out of the 3 lab assignment points.
If you're using R to write your code, and an .rmd file to write your answers 	• Submit your .R code file replicating all your analyses above (with enough comments that a principal investigator on a research project would be able to follow and understand what each step of the code is doing). We need your raw code so that we can run your code ourselves if needed. • Submit your answers as a **KNITTED** .pdf file from the .rmd file[1]. *For graphs, always be sure to save them as .png files and insert them into the answer document even if it was not explicitly asked.* **Important:** If we do not have both your .R solution and .pdf answer files, you will lose 1 out of the 3 lab assignment points.
If you're using Jupyter Notebook to write your R code, and a document editor to write your answers 	• Submit your .ipynb R code file replicating all your analyses above (with enough comments that a principal investigator on a research project would be able to follow and understand what each step of the code is doing). We need your raw code so that we can run your code ourselves if needed. • Submit your answers as a .pdf file. **Do not submit a .doc/.docx file (Word document)**, as we are unable to read those files on Gradescope. *For graphs, always be sure to save them as .png files and insert them into the answer document even if it was not explicitly asked.*

[1] Using R Markdown is never required; but if you have chosen to use it, you have to knit the file to generate the PDF. A better option than R Markdown is to instead work in a Jupyter Notebook from which you can easily create a PDF document. We encourage students wishing to typeset their lab reports to use a Jupyter Notebook (talk to a TF if you need more help).

	Important: If we do not have both your .ipynb code and .pdf answer files, you will lose 1 out of the 3 lab assignment points. **Word of caution: <u>Do not</u>** use Python to perform your analysis in this Lab, unless when explicitly instructed. You will receive no points if you're not using R or Stata for your analysis.
If you're using a Jupyter Notebook to write your R code AND to write your answers 	• Submit your .ipynb R code and answers file replicating all your analyses above (with enough comments that a principal investigator on a research project would be able to follow and understand what each step of the code is doing). We need your raw code so that we can run your code ourselves if needed. • Submit the .pdf version of your .ipynb file. *For graphs, always be sure to save them as .png files and insert them into the answer document even if it was not explicitly asked.* **Important:** If we do not have both your .ipynb code and .pdf answer files, you will lose 1 out of the 3 lab assignment points. **Do not submit ONLY the .ipynb file**, as we might have trouble reading your answers using those files on Gradescope. **Word of caution: <u>Do not</u>** use Python to perform your analysis in this Lab, unless when explicitly instructed. You will receive no points if you're not using R or Stata for your analysis.
If you're using Stata to write your code, and a document editor to write your answers 	• Submit your .do code file replicating all your analyses above (with enough comments that a principal investigator on a research project would be able to follow and understand what each step of the code is doing). We need your raw code so that we can run your code ourselves if needed. • Submit your .log file with the log showing the output generated by your final do-file. **Please do not submit a .smcl file**: we can only read .log files in Gradescope. • Submit your answers as a .pdf file. **Do not submit a .doc/.docx file (Word document)**, as we are unable to read those files on Gradescope. *For graphs, always be sure to save them as .png files and insert them into the answer document even if it was not explicitly asked.* **Important:** If we do not have your .do AND .log solution and .pdf answer files, you will lose 1 out of the 3 lab assignment points.

WHAT ARE DO-FILES AND .R FILES AND WHY DO WE NEED ONE?

Let's imagine the following situation - you just found out you have to present your results to a partner– all the averages you produced and comparisons you made. Suppose you also found out that the data you had used to produce all these results was not completely clean, and have only just fixed it. You now have incorrect numbers and need to re-do everything.

How would you go about it? Would you reproduce everything you did for Lab 1 from scratch? Can you do it? How long would it take you to do? Just re-typing all those commands into Stata or R in order and checking them would take an hour.

An important feature of any good research project is that the results should be reproducible. For Stata and R the easiest way to do this is to create a text file that lists all your commands in order, so anyone can re-run all your Stata or R work on a project anytime. Such text files that are produced within Stata or linked to Stata are called do-files, because they have an extension .do (like intro_exercise.do). Similarly, in R, these files are called .R files because they have an extension of .R. These files feed commands directly into Stata or R without you having to type or copy them into the command window.

An added bonus is that having do-files and .R files makes it very easy to fix your typos, re-order commands, and create more complicated chains of commands that wouldn't work otherwise. You can now quickly reproduce your work, correct it, adjust it, and build on it.

Finally, do-files and .R files make it possible for multiple people to work on a project, which is necessary for collaborating with others or when you hand off a project to someone else.

DATA DESCRIPTION, FILE: nls6679.dta

The data consist of $N = 6{,}042$ children from the 1966 and 1979 National Longitudinal Survey, born between 1948 and 1964. I measure the income of the children, when they have grown up and are between 26 and 43 years old. I measure their parents' income in the first three waves of the survey when their child is 15-17 years old. The sample restrictions and income definitions are adapted from Davis and Mazumder (2023).

TABLE 1
Variable Definitions

	Variable	Description	Obs.	Mean	St. Dev.	Min	Max
	(1)	(2)	(3)	(4)	(5)	(6)	(7)
1	*id_num*	Individual identifier	6,042	n/a	n/a	n/a	n/a
2	*kid_income*	Child's income (2015$) at age 26-43	6,042	70,425	56,437	$544.20	$687,304
3	*parent_income*	Parents' income (2015$) when child is 15-17	6,042	66,549	49,178	$813.50	$1,235,852
4	*child_education*	Child's education (highest grade completed)	6,042	13.19	2.278	8	18
5	*parent_education*	Parents' education (highest grade completed)	6,042	11.15	2.401	8	18
6	*siblings*	Number of siblings	6,042	3.611	2.515	0	16
7	*white*	Child is white	6,042	0.596	0.491	0	1
8	*black*	Child is Black	6,042	0.277	0.448	0	1
9	*female*	Child is female	6,042	0.481	0.500	0	1
10	*age*	Child's age when their own income is measured	6,042	34.32	6.261	26	43
11	*cohort*	Child's year of birth	6,042	1958	5.989	1948	1964

Note: Table reports variable definitions and summary statistics for the data set.

TABLE 2
Stata Commands

STATA command	Description
*clear the workspace clear all version 18 *change working directory and open data cd "C:\Users\gbruich\Ec 50\Lab 1\" use nls6679.dta, clear *Display all variables in the data describe *Report detailed information on all variables codebook	This code shows how to clear the workspace, change the working directory, and open a Stata data file. To change directories on either a mac or windows PC, you can use the drop down menu in Stata. Go to file -> change working directory -> navigate to the folder where your data is located. The command to change directories will appear; it can then be copied and pasted into your .do file. The describe and codebook commands will report information on what is included in the data set loaded into memory.
*Histogram histogram yvar graph export histogram_yvar.png, replace *Histogram, changing number of bins to 50 histogram yvar, bin(50) graph export histogram_yvar.png, replace	These commands create and save histograms of a variable "yvar" which is a placeholder for the name of a variable in your data set. The first line creates a histogram (letting Stata decide how many bins to use). The second line saves the graph as a .png file. The second block of code changes the options by adding ", bin(50)" which will override the default binning and group the data into 50 buckets.
*Summary stats for one variable sum yvar *Summary stats for observations with wvar<=55 sum yvar if wvar <= 55 *Observations with treatment_group equal to 1 sum yvar if treatment_group == 1 *Observations with treatment_group equal to 0 sum yvar if treatment_group == 0 *Summary stats by each value of treatment_group bys treatment_group: sum yvar *Summary statistics for all variables in data sum *Detailed summary statistics sum yvar, d	These commands report means and standard deviations for *yvar*. The first line calculates these statistics across the full sample. The other lines illustrate how to calculate these statistics for observations meeting certain criteria: when another variable in the data is less than or equal to 55, equal to 1, or equal to 0. In these examples, the following are placeholders for variables in your data: yvar, treatment_group, wvar, xvar. To report summary statistics for multiple variables, list them next to each other with no commas: *sum yvar wvar xvar* To report means, standard deviations, medians, and other quantiles and statistics use the ", d" option to report detailed summary statistics
*Create new indicator variables gen dvar= 0 replace dvar = 1 if inrange(xvar, 32.1, 52.1) gen dvar= 0 replace dvar = 1 if xvar >= 0.5	These commands show how to generate a new indicator variable called dvar. In the first example, dvar equals 1 if another variable xvar is within 10 units of the number 42.1. We start by generating a variable that always equals 0. Then we replace it with 1 for observations that meet the criteria that value of xvar is in the range [32.1, 52.1]. The function inrange() takes three arguments: the variable, the lower bound, and the upper bound. The second block of code illustrates another example, where the indicator instead equals 1 when xvar is greater than or equal to 0.5.

*Sort data from lowest to highest sort yvar *Sort data from highest to lowest gsort -yvar	These commands show how to sort the data from lowest to highest based on the value of the variable yvar using the sort command. In the second example, the data are instead sorted from highest to lowest using the gsort command with a – in front of yvar
*Create variable in percentile ranks *Start by rank ordering the data based on yvar egen yvar_rank = rank(yvar) *Get maximum rank, automatically stored as r(max) sum yvar_rank *Store maximum rank as a scalar variable scalar max_rank = r(max) *Normalize rank so that maximum is 100 replace yvar_rank = 100* yvar_rank / max_rank	These commands show how to convert a variable yvar into percentile ranks, normalized so that the highest rank is 100. We start using egen and the rank() function to generate a new variable that rank orders yvar. Then to normalize the variable, we divide it by the maximum rank and multiply by 100. The maximum rank is saved temporarily as r(max) after the sum command. I store it as a "scalar variable" called max_rank, and use that variable in the denominator in the last line.
twoway (scatter yvar xvar) (lfit yvar xvar) graph export figure1.png, replace	This pair of commands first draws a scatter plot of *yvar* against *xvar*. The second line saves the graph as a .png file.
*install bin scatter command ssc install binscatter *Bin scatter plot – connected dots binscatter yvar xvar, linetype(connect) graph export figure2_connected.png, replace *Bin scatter plot – linear best fit line binscatter yvar xvar, linetype(lfit) graph export figure2_linear.png, replace	These commands show how to create binned scatter plots. The first line installs the command from the SSC. The second block of code shows how to create a binned scatter plot where a variable yvar is along the y-axis and a variable xvar is along the x-axis. It will connect the dots with a line. The third block of code shows how to create a binned scatter plot where a variable yvar is along the y-axis and a variable xvar is along the x-axis. It will also plot a linear best fit line. The commands beginning with "graph export" save the graphs as .png files.
*Set seed to make reproducible set seed 505050505 *Generate uniform random number between 0-1 gen random_number = runiform()	These commands show how to randomly assign a number between 0 and 1 to each observation. We start by setting the "seed". Then we generate a new variable called random_number that equals runiform(). runiform() is a function that creates a pseudo random number that has a uniform distribution.
*close any possibly open log-files cap log close *start a log file log using lab1.log, replace *commands go here *close and save log file log close	These commands show how to start and close a log file, which will save a text file of all the commands and output that appears on the command window in stata. The first line is short for "capture log close" which will close any open log files, and otherwise just proceed to the next step. Then the "log using lab1.log replace" starts the log file and changes the default in two ways. First, it changes the file type to have a .log file extension, which creates a plain text log file (which is readable in Gradesope so is important!). Second, it also adds the ", replace" option which will save over any other log file that has the same name. This is usually what you want. The rest of your lab code can go below the "log using lab1.log, replace" line. At the end of your do-file you can include the last line which is "log close" which will close and save the log-file.

TABLE 3
R Commands

R command	Description

#clear the workspace
rm(list=ls()) # removes all objects from the environment
cat('\014') # clears the console
 	
#Install and load haven package
if (!require(haven)) install.packages("haven"); library(haven)
 	
#Change working directory and load stata data set
setwd("C:/Users/gbruich/Ec 50/Lab 1")
nls <- read_dta("nls6679.dta")
	This sequence of commands shows how to open Stata datasets in R. The first block of code clears the work space. The second block of code installs and loads the "haven" package. The third block of code changes the working directory to the location of the data and loads in nls6679.dta. To change the working directory in R Studio, you can also use the drop down menu. Go to session -> set working directory -> choose working directory.
	
	The easiest way to open a Stata data set in R Studio is to use the drop down menu. Go to file, then import data set, and finally browse to locate the file you want to open. This option will be available after you install the haven package.

#Histogram in base R
hist(nls$yvar, probability = T)
 	
#Saving a histogram drawn in base R
png("histogram_yvar.png")
hist(nls$yvar, probability = T)
dev.off()
 	
#Histogram using ggplot
if (!require(tidyverse)) install.packages("tidyverse"); library(tidyverse)
if (!require(ggplot2)) install.packages("ggplot2"); library(ggplot2)
 	
ggplot(nls) + geom_histogram(aes(x=yvar, y=..density..))
ggsave("histogram_yvar.png")
 	
#Use 50 bins, overriding default
ggplot(nls) + geom_histogram(aes(x=yvar, y=..density..), bins = 50)
	These commands create and save histograms of a variable "yvar" which is a placeholder for the name of a variable in your data set. The first line creates a histogram (letting R decide how many bins to use) using base R. The probability = T option produces a histogram on the density scale. The default is a bar graph of frequencies (counts).
	
	The second block of code shows how to save the histogram, by adding one line before and one line after the hist() function. The png() function is used to name the file "histogram_yvar.png" that will be saved in the working directory.
	
	The third block of code shows how to do this using ggplot. First start by installing the tidyverse library. Then use ggplot to draw the graph. The ggsave() line saves the graph as a .png file.
	
	The last line overrides the default to show 50 bins by adding the bins = 50 option.

# summary stats, unweighted 	
summary(nls$yvar)
mean(nls$yvar, na.rm=TRUE)
sd(nls$yvar, na.rm=TRUE)
median((nls$yvar, na.rm=TRUE)
	These commands show how to calculate unweighted summary statistics. The variable yvar is a placeholder for a variable in a data frame called nls. The ", na.rm=TRUE" argument takes care of missing values.

#Create new indicator variable called dvar nls$dvar <- 0 nls$dvar[which(nls$xvar >= 0.5)] <- 1 #Alternatively, use ifelse() function nls$dvar <- ifelse(nls$xvar >= 0.5, 1, 0)	These commands illustrate an example of how to generate an indicator that equals 1 when xvar is greater than or equal to 0.5. The first two lines show how to start a variable that always equals 0 and then replace it equal to 1 if a logical condition is satisfied (i.e., xvar >= 0.5) An alternative way to do it uses the ifelse() function, which takes three arguments: the logical condition, the value if the condition is satisfied, and the value of the condition is not satisfied.

#Indicator for being between two values
 	
#Step 1: define the upper value and the lower value
upper_bound <- 52.1
lower_bound <- 32.1
 	
#Step 2: use ifelse function to define the indicator variable
nls$dvar <- ifelse(nls$xvar <= upper_bound &
nls$xvar >= lower_bound,
1,
0)
	This example shows how to generate a new indicator variable called dvar that equals 1 if another variable xvar is within 10 units of the number 42.1, similar to question 6.
	
	I start by defining two objects (upper_bound and lower_bound) that store the values of these end points.
	
	Then I use the ifelse() function, which takes three arguments: the logical condition, the value if the condition is satisfied, and the value of the condition is not satisfied.
	
	The logical statement in this case involves two things both being true: yvar is less than upper_bound and yvar is greater than lower_bound. In R, this is done using the "&" symbol between the two conditions. (In R, a vertical pipe "|" indicates "or"; and an exclamation mark "!" indicates "not").

#Sort data from lowest to highest
nls <- nls[order(nls$yvar),]
View(nls)
 	
#Sort data in highest to lowest
nls <- nls[order(-nls$yvar),]
View(nls)
	These commands show how to sort the data from lowest to highest based on the value of the variable yvar using the order() function. In the second example, the data are instead sorted from highest to lowest using the order() function with a – in front of yvar. The View() command allows you to browse the data. Note that the V is capitalized.

#Create variable in percentile ranks
#Start by rank ordering the data based on yvar
nls$yvar_rank <- rank(nls$yvar)
 	
#Store the maximum rank
max_rank <- max(nls$yvar_rank)
 	
#Normalize rank so that maximum is 100
nls$yvar_rank <- 100*nls$yvar_rank / max_rank
 	
# Create Function that will Calculate Percentile Ranks with NAs 	
 	
#Define function for percentile ranking
percentile_rank<-function(variable){
 	
#Convert to ranks, taking care of potential missing values
r <- ifelse(is.na(variable), NA, rank(variable, ties.method = "average"))
 	
#Return percentile rank = rank normalized so max is 100
100*r/max(r, na.rm = T)
}
 	
#Example using Function to Define ranks
nls$yvar_rank <-with(nls, percentile_rank(yvar))
	These commands show how to convert a variable yvar into percentile ranks, normalized so that the highest rank is 100. We start using the rank() function to generate a new variable that rank orders yvar. Then to normalize the variable, we divide it by the maximum rank and multiply by 100. The code uses the max() function in R in the denominator to do the normalization.
	
	Unfortunately, the rank() function does not work as desired for data with missing values (NAs). But we can create our own function to do what we want that will work as intended in more complex data sets. This second block of code shows how to define a new function called percentile_rank() that will generate percentile ranks that assign missing values to NAs, and returns the percentile rank normalized to have a maximum rank of 100.
	
	The last line shows how to use the function to create the variable yvar_rank. The with() function in R takes two arguments: a data frame and an expression. The data frame argument is nls and the expression applies the new function we wrote to the variable yvar: percentile_rank(yvar).

# Install and load ggplot2 package 	
if (!require(tidyverse)) install.packages("tidyverse"); library(tidyverse)
if (!require(ggplot2)) install.packages("ggplot2"); library(ggplot2)
 	
# Draw scatter plot with linear fit line 	
ggplot(nls) + geom_point(aes(x = xvar1, y = yvar)) +
  geom_smooth(aes(x = xvar, y = yvar), method = "lm", se = F)
 	
#Save graph as figure1a.png
ggsave("figure1a.png")
	These commands show how to draw a scatter plot of *yvar* against *xvar1*. The *geom_smooth* part of the code adds an OLS regression line. The last line saves the graph as a .png file.

#install ggplot and statar packages
if (!require(tidyverse)) install.packages("tidyverse"); library(tidyverse)
if (!require(ggplot2)) install.packages("ggplot2"); library(ggplot2)
if (!require(statar)) install.packages("statar"); library(statar)
 	
#Bin scatter plot – connected dots
ggplot(nls, aes(x = xvar , y = yvar)) +
  stat_binmean(n = 20, geom = "line") +
  stat_binmean(n = 20, geom = "point")
 	
#Save graph
ggsave("binscatter_connected.png")
 	
#Bin scatter plot – linear best fit line
ggplot(nls, aes(x = xvar , y = yvar)) +
  stat_smooth(method = "lm", se = FALSE) +
  stat_binmean(n = 20, geom = "point")
 	
#Save graph
ggsave("binscatter_bestfitline.png")
	These commands show how to create binned scatter plots. The first lines install packages, including the statar package so that we can use the stat_binmean() function with ggplot.
	
	The second block of code shows how to create a binned scatter plot where a variable yvar is along the y-axis and a variable xvar is along the x-axis. It will connect the dots with a line. It then uses ggsave to save the graph.
	
	The third block of code shows how to create a binned scatter plot where a variable yvar is along the y-axis and a variable xvar is along the x-axis. It will also plot a linear best fit line. It then uses ggsave to save the graph.

#Store HUID
HUID <- 505050505
 	
#Set seed so that simulations are replicable
set.seed(HUID)
 	
#Uniformly distributed random number between 0 and 1
nls$random_number <- runif(length(nls$yvar))
	These commands show how to randomly assign a number between 0 and 1 to each observation. We start by setting the "seed".
	
	Then we generate a new variable called random_number that equals runif(). runif() is a function that R uses to create a pseudo random number that has a uniform distribution.
	
	Inside the argument of runif() is length(nls$yvar), which counts the number of observations. The length() function returns the length of an object in R.

#Report total number of observations in treatment group
sum(nls$treatment_group)
 	
#Report total number of observations in control group
sum(1-nls$treatment_group)
	This code shows how to count how many observations have treatment_group equal to 1 and how many observations have treatment_group equal to 0.
	
	The sum() command adds up the indicator variable across all observations, yielding the total number of observations with treatment_group equal to 1. Summing the values of 1 minus treatment_group gives the total number of observations with treatment_group equal to 0.

#Report summary statistics split by different groups
 	
#Various ways to do this. First tapply()
tapply(nls$yvar, nls$treatment_group, mean)
tapply(nls$yvar, nls$treatment_group, sd)
 	
#Alternatively, by()
by(nls$yvar, list(nls$treatment_group), mean)
by(nls$yvar, list(nls$treatment_group), sd)
 	
#Third - Tidyverse summarise_all()
nls %>% group_by(treatment_group) %>% summarise_all("mean")
nls %>% group_by(treatment_group) %>% summarise_all("sd")
 	
#To report all variables, add this line before running:
options(dplyr.width = Inf)
nls %>% group_by(treatment_group) %>% summarise_all("mean")
nls %>% group_by(treatment_group) %>% summarise_all("sd")
	These commands shows how to report summary statistics separately by groups defined by another variable, enabling for example summary statistics to be computed separately for a treatment group and a control group.
	
	The first example uses the tapply() function to report the mean and standard deviation of yvar grouped by another variable treatment_group.
	
	The second example uses the by() function to do the same thing.
	
	The third example uses a combination of commands from the tidyverse library to report the means and standard deviations for all the variables in the data frame all at once with summarise_all() .
	
	By default, only the first several variables will be displayed. The options(dplyr.width = Inf) line will change the default to show summary statistics for all the variables