Lab Assignment #1

Math 437 - Modern Data Analysis

Due January 25, 2023

Instructions

This is an R Markdown file. This file allows you to include both your R code and your commentary in a single file. All lab work - code and answers - should be done in this notebook.

When you click *Knit*, R Studio creates (or overwrites) a pdf file containing your answers as well as the code and output. Note that creating a pdf file requires your *LaTeX* distribution to work nicely with R Studio, which some students have reported problems with. You can change the output type to a HTML file by changing the `output: pdf_document` line to `output: html_document`, which should allow you to still produce a solution file while you figure out what the issue is and how to fix it.

R code is written in chunks. Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by pressing *Ctrl/Cmd+Alt+I*. A chunk looks like this:

```
# write your R comments here
```

Note that the text in curly braces must start with the name of the language your code is written in (`r`) followed immediately by a (unique) chunk name, a comma, and options for how R Studio should print the code and output. Some common options are:

- `echo`: by default `echo = FALSE` hides the code in the chunk. You typically want to set this to `echo = TRUE` in the global options so that I can see your code.
- `eval`: by default `eval = TRUE`: R Studio will run the code chunk and display the output. Many of the chunks in the lab and homework files set `eval = FALSE` because the code does not run properly. Once you get a chunk to run the way you want, you should remove that.
- `include`: by default `include = TRUE`: R Studio will display the code (if `echo = TRUE`) and results (if `eval = TRUE`) for the chunk. Use `include = FALSE` if you want R Studio to run the code but display neither the code nor the output.
- `message`: by default `message = TRUE`: R Studio will print in the file any messages from R. Generally you want to keep this, but you can use `message = FALSE` if there is an annoying persistent message that isn't informative.
- `warning`: by default `warning = TRUE`: if the code runs but produces a warning message, R Studio will print the warning message. Generally you want to keep this, but you can use `warning = FALSE` if you don't want, for example, a bunch of reminders about the version of R the most updated versions of the packages were written in.

To run an individual chunk, click the green triangle (Play button) in the top right of the chunk.

This lab assignment is worth a total of **15 points**.

Pair Programming

Pair programming is a collaborative technique used in all disciplines of software development (including data science). In typical pair programming, the "driver" has total control over the keyboard and mouse, but can only type in response to instructions from the "navigator."

I expect that these roles will be fluid throughout the semester, depending on who is present and how comfortable each group member is with R and the content. However, it is in everyone's best interest to work collaboratively on these problems. If you feel confident with R programming, especially early on, please consider giving up keyboard/mouse duties to someone who is less confident and helping them to learn the syntax.

Please do not have one person in the group control *both* the keyboard and the group's thought process if multiple group members are present. Also, please do not split up and tackle problems individually during the lab section (though this may be necessary if your group does not finish during the allotted time).

Problem 1: Introductory Material (Code: 1 pt)

You should always include two chunks at the beginning of your R Markdown file. The first chunk is the default chunk that R Studio puts in when you create a new R Markdown file:

```
knitr::opts_chunk$set(echo = TRUE)
```

This chunk sets the global options for printing code and output. Generally the only global option you need to change is the one already in the default chunk.

The second chunk that you should include at the beginning of the file is a chunk that imports the packages (a package is a set of additional functions and data) and datasets you will be using. For example, let's import the *Auto* dataset from the ISLR2 package.

If you do not have the ISLR2 package installed, in R Studio, click on the *Packages* tab (in the bottom right) and install the package.

```
library(ISLR2)  # load the library
View(Auto) # View the data in a separate window, you can also use the fix() function
```

Generally, to import a dataset from Canvas, download the file and then click *Import Dataset* in the *Environment* tab (top right). Select the right type of file (usually you want `From Text (readr)` for my csv files), click *Browse* and find the file, then copy the code from the *Code Preview* section.

Note that the code to import your dataset includes your entire file structure, which means that I and your partners won't be able to replicate your data import. To fix this, type `setwd("Your Working Directory")` in the *Console* tab (bottom left) and run it, make sure that the dataset(s) you are importing are in that directory, and then fix the data import code to include only the filename.

The first thing you want to do after importing a dataset is do some sanity checks to make sure everything imported okay.

Typically you should check that the number of observations and number of variables in the dataset are what you expected:

```
dim(Auto)  # number of observations and number of variables
```

```
## [1] 392   9
```

It's also good to check the names of your variables. Generally, variable names in R should be one word or multiple words separated by underscores (_) or periods (.).

```
names(Auto)  # names of variables in the dataset
```

```
## [1] "mpg"          "cylinders"    "displacement" "horsepower"   "weight"
## [6] "acceleration" "year"         "origin"       "name"
```

Sometimes it may be useful to see a summary of the variables in the dataset, including missing data.

```
summary(Auto)  # brief summary of each variable in the dataset
```

```
##       mpg          cylinders      displacement     horsepower        weight    
##  Min.   : 9.00   Min.   :3.000   Min.   : 68.0   Min.   : 46.0   Min.   :1613  
##  1st Qu.:17.00   1st Qu.:4.000   1st Qu.:105.0   1st Qu.: 75.0   1st Qu.:2225  
##  Median :22.75   Median :4.000   Median :151.0   Median : 93.5   Median :2804  
##  Mean   :23.45   Mean   :5.472   Mean   :194.4   Mean   :104.5   Mean   :2978  
##  3rd Qu.:29.00   3rd Qu.:8.000   3rd Qu.:275.8   3rd Qu.:126.0   3rd Qu.:3615  
##  Max.   :46.60   Max.   :8.000   Max.   :455.0   Max.   :230.0   Max.   :5140  
##                                                                                
##   acceleration        year           origin                      name    
##  Min.   : 8.00   Min.   :70.00   Min.   :1.000   amc matador       :  5  
##  1st Qu.:13.78   1st Qu.:73.00   1st Qu.:1.000   ford pinto        :  5  
##  Median :15.50   Median :76.00   Median :1.000   toyota corolla    :  5  
##  Mean   :15.54   Mean   :75.98   Mean   :1.577   amc gremlin       :  4  
##  3rd Qu.:17.02   3rd Qu.:79.00   3rd Qu.:2.000   amc hornet        :  4  
##  Max.   :24.80   Max.   :82.00   Max.   :3.000   chevrolet chevette:  4  
##                                                  (Other)           :365  
```

To view information about a dataset or help about a function, you can type `?` followed by the name of the dataset or function, or just search for it in the *Help* tab in R Studio.

Problem 2: More Basic R Commands

Part a (Code: 1 pt)

Work through ISLR Lab 2.3.1. Copy each chunk in the lab to its own chunk in this file.

```
x <- c(1,3,2,5)
x
```

```
## [1] 1 3 2 5
```

```
x = c(1,6,2)
x
```

```
## [1] 1 6 2
```

```
y = c(1,4,3)
```

```
length(x)
```

```
## [1] 3
```

```
length(y)
```

```
## [1] 3
```

```
x + y
```

```
## [1]  2 10  5
```

```
ls()
```

```
## [1] "x" "y"
```

```
rm(x,y)
ls()
```

```
## character(0)
```

```
#remove all objects at once
#rm(list = ls())
```

```r
?matrix
```

```
## starting httpd help server ... done
```

```r
x = matrix(data = c(1,2,3,4), nrow = 2, ncol = 2)
x
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```r
x = matrix(c(1,2,3,4), 2, 2)
x
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```r
matrix(c(1,2,3,4),2,2,byrow=TRUE)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

```r
sqrt(x)
```

```
##          [,1]     [,2]
## [1,] 1.000000 1.732051
## [2,] 1.414214 2.000000
```

```r
x^2
```

```
##      [,1] [,2]
## [1,]    1    9
## [2,]    4   16
```

```r
set.seed(1242)
x <- rnorm(50)
y <- x + rnorm(50, mean = 50, sd = .1)
cor(x,y)
```

```
## [1] 0.9972803
```

```r
set.seed(1303)
rnorm(50)
```

```
##  [1] -1.1439763145  1.3421293656  2.1853904757  0.5363925179  0.0631929665
##  [6]  0.5022344825 -0.0004167247  0.5658198405 -0.5725226890 -1.1102250073
## [11] -0.0486871234 -0.6956562176  0.8289174803  0.2066528551 -0.2356745091
## [16] -0.5563104914 -0.3647543571  0.8623550343 -0.6307715354  0.3136021252
## [21] -0.9314953177  0.8238676185  0.5233707021  0.7069214120  0.4202043256
## [26] -0.2690521547 -1.5103172999 -0.6902124766 -0.1434719524 -1.0135274099
## [31]  1.5732737361  0.0127465055  0.8726470499  0.4220661905 -0.0188157917
## [36]  2.6157489689 -0.6931401748 -0.2663217810 -0.7206364412  1.3677342065
## [41]  0.2640073322  0.6321868074 -1.3306509858  0.0268888182  1.0406363208
## [46]  1.3120237985 -0.0300020767 -0.2500257125  0.0234144857  1.6598706557
```

```r
set.seed(3)
y <- rnorm(100)
mean(y)
```

```
## [1] 0.01103557
```

```
var(y)
```

```
## [1] 0.7328675
```

```
sqrt(var(y))
```

```
## [1] 0.8560768
```

```
sd(y)
```

```
## [1] 0.8560768
```

Part b (Explanation: 1 pt)

Explain in your own words the purpose of the `set.seed()` function. What happens if you *don't* include this line before running the `rnorm` function?

Without `set.seed()`, the `rnorm` function will produce random vectors every time the function is called. With `set.seed()`, the function will output a pre-made/pre-saved vector in R.

This is similar to video games like Pokemon. Each game will use a different seed. No two game files will be the same unless you use a software where you control what seed of the game you play. This is commonly seen on Twitch where two streamers will compete on the same seed to keep fair game play.

Problem 3: Indexing Data

Part a (Code: 1 pt)

Work through ISLR Lab 2.3.3. Copy each chunk in the lab to its own chunk in this file.

```
A <- matrix(1:16,4,4)
A
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

```
A[2,3]
```

```
## [1] 10
```

```
A[c(1,3), c(2,4)]
```

```
##      [,1] [,2]
## [1,]    5   13
## [2,]    7   15
```

```
A[1:3, 2:4]
```

```
##      [,1] [,2] [,3]
## [1,]    5    9   13
## [2,]    6   10   14
## [3,]    7   11   15
```

```
A[1:2, ]
```

```
##      [,1] [,2] [,3] [,4]
```

```
## [1,]    1    5    9   13
## [2,]    2    6   10   14
```

```r
A[1, ]
```

```
## [1]  1  5  9 13
```

```r
A[-c(1,3), ]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    6   10   14
## [2,]    4    8   12   16
```

```r
# A[-c(1,3), -c(1,3,4)]
```

```r
dim(A)
```

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

Part b (Code: 0.5 pts)

When you import data into R, it usually will be stored in a *data frame* object instead of a matrix. There are a variety of ways to extract an individual column from a data frame.

```r
Auto2 <- Auto[1:10,]  # first 10 rows
```

```r
Auto2[, 1]
```

```
##  [1] 18 15 18 16 17 15 14 14 14 15
```

```r
Auto2[[1]]
```

```
##  [1] 18 15 18 16 17 15 14 14 14 15
```

```r
Auto2$mpg
```

```
##  [1] 18 15 18 16 17 15 14 14 14 15
```

Note that each of these methods extracts the column as a vector. There are also ways to extract the column as a one-column data frame:

```r
Auto2[, 1, drop = FALSE]
```

```
##    mpg
## 1   18
## 2   15
## 3   18
## 4   16
## 5   17
## 6   15
## 7   14
## 8   14
## 9   14
## 10  15
```

```r
Auto2["mpg"]
```

```
##    mpg
## 1   18
## 2   15
## 3   18
```

```
## 4   16
## 5   17
## 6   15
## 7   14
## 8   14
## 9   14
## 10  15
```

In the code chunks below, find two ways to subset the `Auto2` data to get a two-column dataset containing `mpg` and `horsepower` (columns 1 and 4). Note that `drop = FALSE` is only necessary when you want a one-column data frame.

```
Auto2[ ,c(1,4)]
```

```
##    mpg horsepower
## 1   18        130
## 2   15        165
## 3   18        150
## 4   16        150
## 5   17        140
## 6   15        198
## 7   14        220
## 8   14        215
## 9   14        225
## 10  15        190
```

```
Auto2[c("mpg","horsepower")]
```

```
##    mpg horsepower
## 1   18        130
## 2   15        165
## 3   18        150
## 4   16        150
## 5   17        140
## 6   15        198
## 7   14        220
## 8   14        215
## 9   14        225
## 10  15        190
```

Problem 4: Hypothesis Testing in R

Part a (Code: 1 pt)

Fix the chunk below to create a vector `x` consisting of 50 random numbers from a normal distribution with mean 100 and standard deviation 15 and a vector `y` consisting of 50 random numbers from a normal distribution with mean 90 and standard deviation 15. (Refer back to the end of Problem 2 if you need help.)

```
set.seed(1220)
x <- rnorm(50, mean=100, sd=15) # replace the ... with appropriate arguments and values
y <- rnorm(50, mean=90, sd=15)
```

Now, run the chunk below to perform a two-sample t-test to determine whether the population means of `x` and `y` are different, and store the output in a variable called `t_test_sim`:

```
t_test_sim <- t.test(x,y, alternative = "t") # "t" for two-sided
```

After you run this chunk, the variable `t_test_sim` should appear in your Environment tab as a "List of 10." In R, the output of a hypothesis test is an *htest* object containing information about the methods and results of the inference. Let's see what information we can get out of the `t_test_sim` object:

```
t_test_sim <- t.test(x,y, alternative = "t") # "t" for two-sided
names(t_test_sim)
```

```
##  [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
##  [6] "null.value"  "stderr"      "alternative" "method"      "data.name"
```

We can extract the p-value for our test using the `$` operator:

```
t_test_sim <- t.test(x,y, alternative = "t") # "t" for two-sided
t_test_sim$p.value
```

```
## [1] 0.5915391
```

Part b (Code: 1 pt)

Sometimes our data is in this "wide" format (where each column represents the values from a group and we can use the individual vectors as the arguments), but more often our data is in "long" format (where each column represents a variable). Let's see what this data would look like in long format:

```
group_values <- rep(c("x", "y")) # 50 x followed by 50 y
numerical_values <- c(x, y)
sim_df_long <- data.frame(group = group_values, value = numerical_values)
```

Write code in the chunk below that finds the number of rows and columns in the `sim_df_long` data frame.

```
group_values <- rep(c("x", "y")) # 50 x followed by 50 y
numerical_values <- c(x, y)
sim_df_long <- data.frame(group = group_values, value = numerical_values)
dim(sim_df_long)
```

```
## [1] 150   2
```

Write code in the chunk below that finds the names of the variables in the data frame.

```
group_values <- rep(c("x", "y")) # 50 x followed by 50 y
numerical_values <- c(x, y)
sim_df_long <- data.frame(group = group_values, value = numerical_values)
names(sim_df_long)
```

```
## [1] "group" "value"
```

To perform a two-sample t-test when the data is in "long" format, we use a *formula* argument of the form `response ~ explanatory`:

```
group_values <- rep(c("x", "y")) # 50 x followed by 50 y
numerical_values <- c(x, y)
sim_df_long <- data.frame(group = group_values, value = numerical_values)
t_test_sim <- t.test(x,y, alternative = "t") # "t" for two-sided
t_test_sim_long = t.test(value ~ group, data = sim_df_long, alternative = "t")
```

Part c (Explanation: 0.5 pts)

Write a chunk to extract the p-value from `t_test_sim_long`. Do you get the same p-value that you got originally?

```r
group_values <- rep(c("x", "y")) # 50 x followed by 50 y
numerical_values <- c(x, y)
sim_df_long <- data.frame(group = group_values, value = numerical_values)
t_test_sim <- t.test(x,y, alternative = "t") # "t" for two-sided
t_test_sim_long = t.test(value ~ group, data = sim_df_long, alternative = "t")
t_test_sim_long$p.value
```

```
## [1] 0.2980448
```

```r
t_test_sim$p.value == t_test_sim_long$p.value
```

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

```r
#if TRUE, then they have the same p.value
```

We get the same p-value in `t_test_sim_long` as we do in `t_test_sim`. ## Part d (Explanation: 1.5 pts)

What information is contained the `statistic`, `parameter`, and `conf.int` objects within `t_test_sim` and `t_test_sim_long`? (You can write and run code chunks to verify your answer.)

```r
group_values <- rep(c("x", "y")) # 50 x followed by 50 y
numerical_values <- c(x, y)
sim_df_long <- data.frame(group = group_values, value = numerical_values)
t_test_sim <- t.test(x,y, alternative = "t") # "t" for two-sided
t_test_sim_long = t.test(value ~ group, data = sim_df_long, alternative = "t")
t_test_sim$statistic
```

```
##         t
## 0.5387569
```

```r
t_test_sim$parameter
```

```
##       df
## 80.62452
```

```r
t_test_sim$conf.int
```

```
## [1] -0.2542485  0.4430446
## attr(,"conf.level")
## [1] 0.95
```

```r
t_test_sim <- t.test(x,y, alternative = "t") # "t" for two-sided
t_test_sim_long$statistic
```

```
##        t
## 1.044356
```

```r
t_test_sim_long$parameter
```

```
##       df
## 146.1541
```

```r
t_test_sim_long$conf.int
```

```
## [1] -0.1421736  0.4608096
## attr(,"conf.level")
## [1] 0.95
```

Both `t_test_sim` and `t_test_sim_long` have the same `statistic` t=3.200753, `parameter` df=97.71287, and `conf.int` attr(,"conf.level")=0.95. # Problem 5: Graphing with Base R

For parts (a) through (c):

- Create the requested plot using the *Auto* dataset. Make sure each plot has appropriate labels for the x-and y-axis. You can find all of the relevant code in ISLR Lab 2.3.5.
- Write a short paragraph describing the graph to someone who does not know anything about the *Auto* dataset. When I describe a graph, I use the TAME strategy to orient the reader/listener:

  Topic: what data (observational units and variables) you are graphing

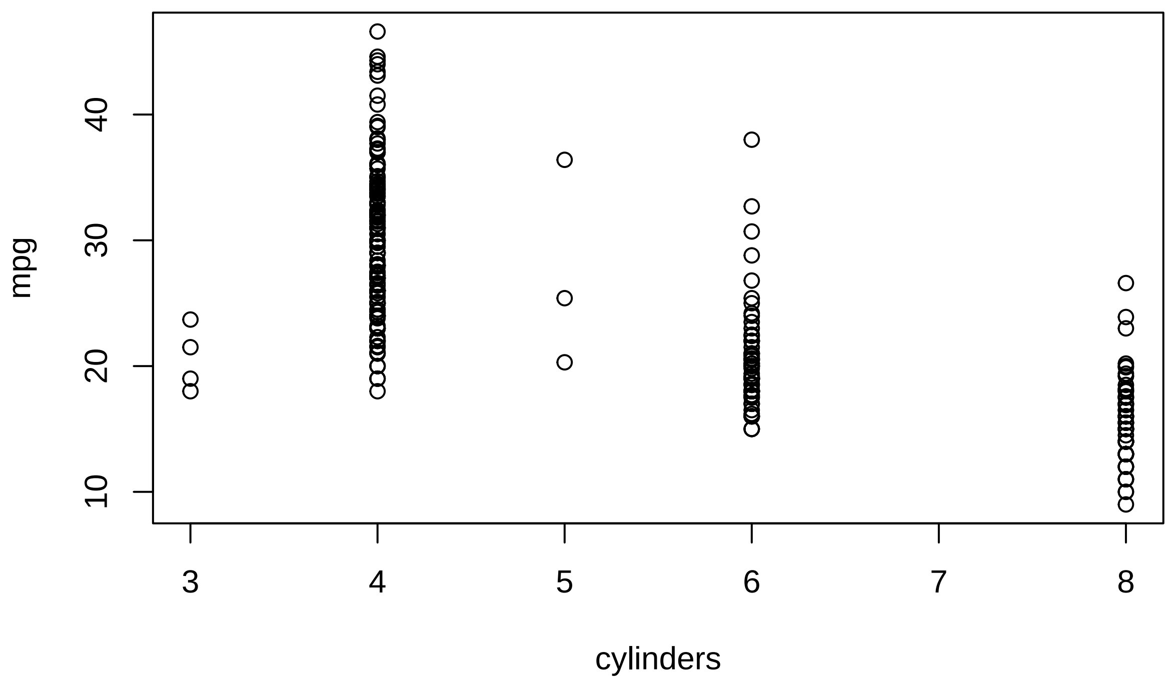
  Axes: what variables are being mapped to the x-axis and y-axis (and other properties, e.g., color)

  Main point: the most important takeaway (for one variable, this usually deals with the mode of the distribution; for two variables, this usually deals with the association) for scatterplot, looking for: direction, form, strength of relationship **E**xtra information (optional): anything else that you find interesting to point out

Part a (Code: 0.5 pts; Explanation: 1 pt)

Create and describe a scatterplot of `mpg` (response) against `cylinders` (numerical predictor). Do not attach the dataset; use the `$` sign to select the variables.

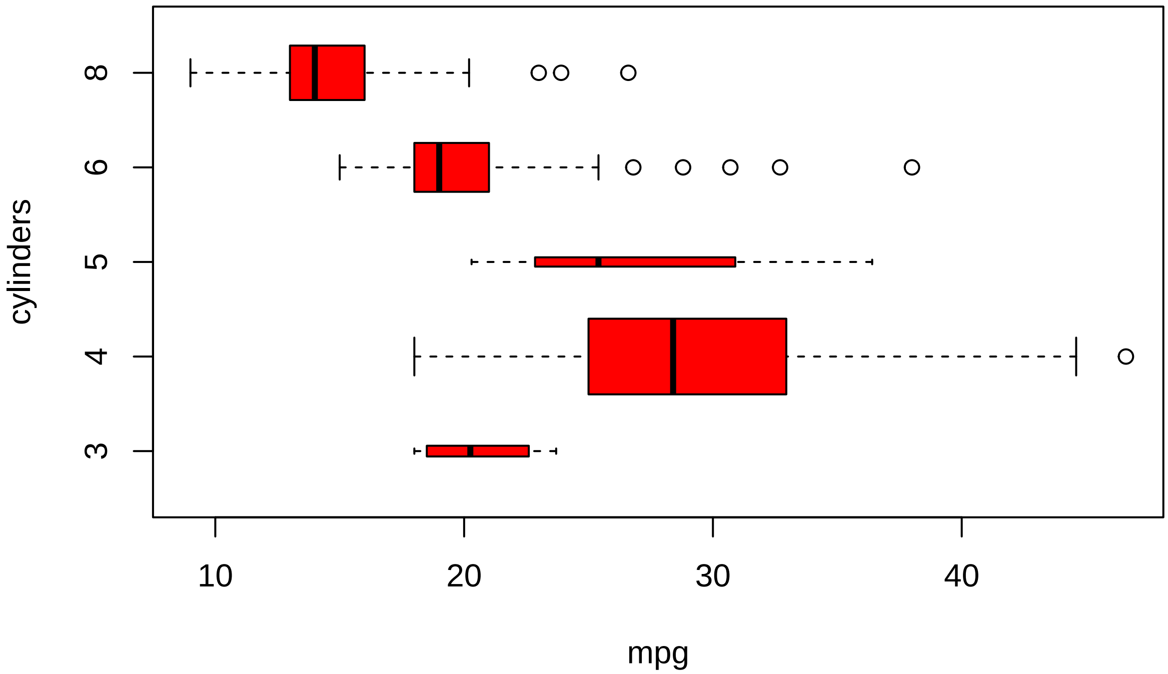
```
plot(Auto$cylinders, Auto$mpg, xlab="cylinders", ylab="mpg")
```



We are plotting Cylinders and MPG for all the vehicles in our datset. Cylinders is on the x-axis, and MPG is on the y-axis. For vehicles with an even number of cylinders, MPG decreases with more cylinders. A majority of vehicles seem to have either have 4, 6, or 8 cylinders. Very few vehicles have an odd number of cylinders. There are no vehicles in this dataset that have 7 cylinders.
Part b (Code: 0.5 pts; Explanation: 1 pt)

Create and describe a boxplot of `mpg` (response) by `cylinders` (categorical predictor). The boxes should be horizontal and red.

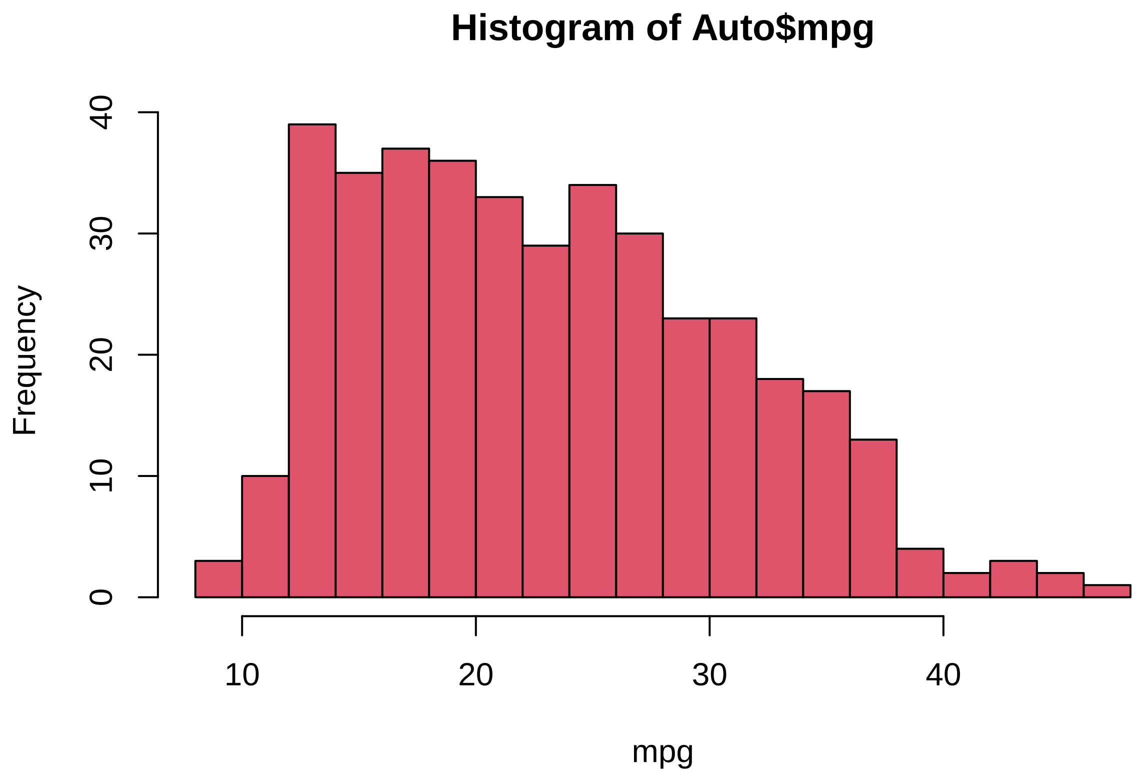
```
Auto$cylinders <- as.factor(Auto$cylinders)
plot(Auto$cylinders, Auto$mpg, col = "red", varwidth = TRUE, horizontal = TRUE, xlab = "mpg", ylab = "cy
```



We are graphing Cylinders and MPG. Cylinders is on the x-axis, and MPG is on the y-axis. The boxplots in red are basically the same as the scatterplot, but something that is different is we can see the medians for vehicles with different cylinders and there are vehicles that get way more mpg than the average vehicle. ## Part c (Code: 0.5 pts; Explanation: 1 pt)

Create and describe a histogram of `mpg`. The bars should be red and there should be roughly 15 bars.

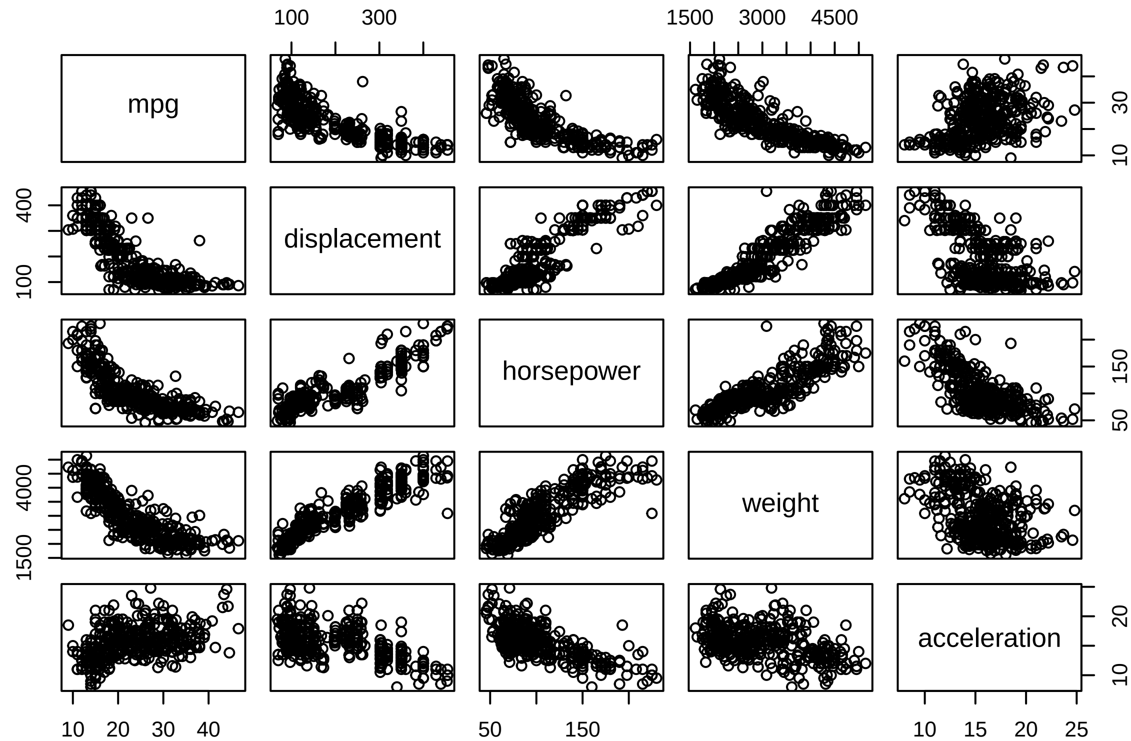
```
hist(Auto$mpg, col=2, breaks=15, xlab="mpg")
```



The histogram shows the range of MPG for all cars in our dataset. MPG is on the x-axis and the number of cars within a certain range of MPG is on the y-axis. We can see most vehicles have MPG between 10 and 40. Vehicles that get less than 10 mpg or more than 40 mpg are rare. ## Part d (Code: 0.5 pts)

Using the `pairs` function, create a scatterplot matrix showing `mpg`, `displacement`, `horsepower`, `weight`, and `acceleration`.

```
pairs(~ mpg + displacement + horsepower + weight + acceleration, data = Auto)
```



Problem 6: Graphing with ggplot2

The plotting commands in the textbook are designed to work with base R. However, most people who use R find the base R graphics system to be cumbersome and use use either the *ggplot2* package or the *lattice* package. We will use the *ggplot2* package. This problem just requires you to run the code chunks and understand the commands.

Install the package if you do not have it installed, then load the package.

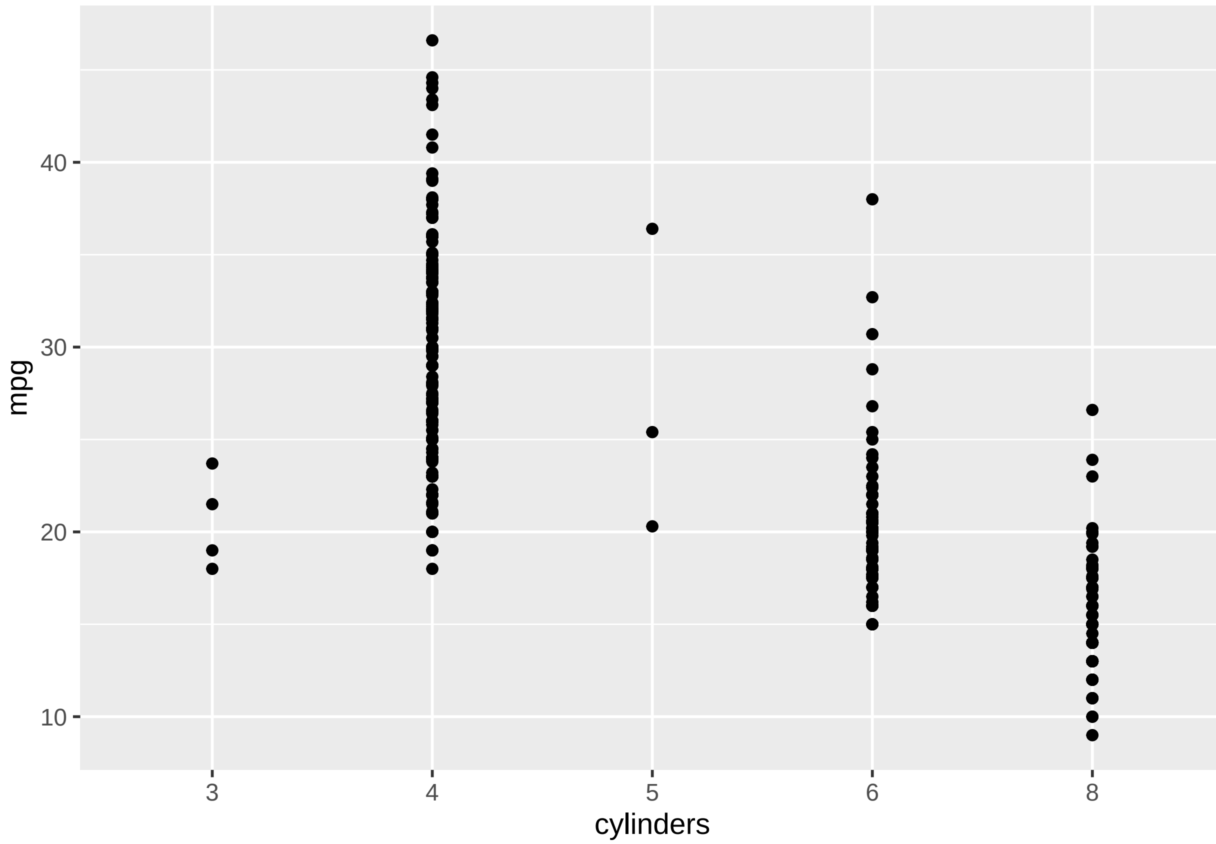
```
library(ggplot2)
```

(This is probably a good chunk to set options `message = FALSE, warning = FALSE` for.)

Part a (Code: 0.5 pts)

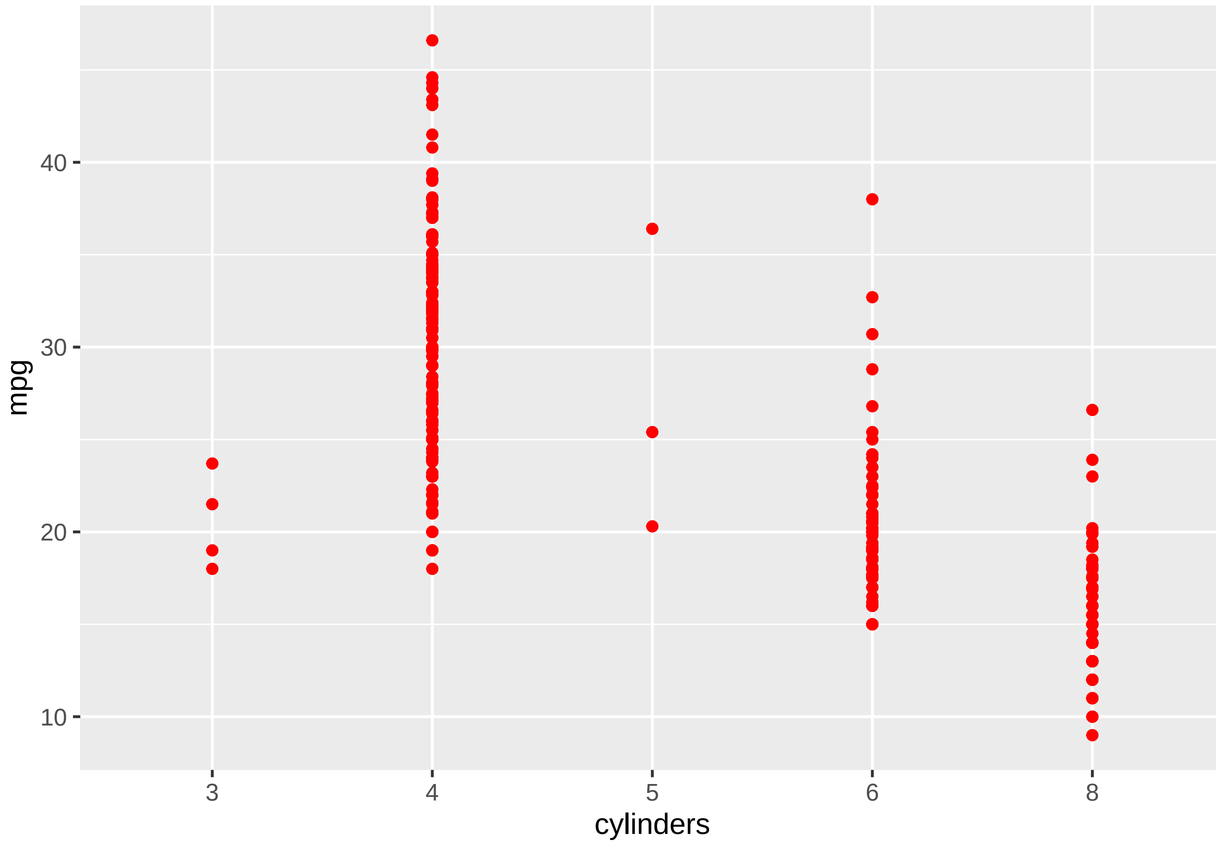
Now let's start by creating the basic scatterplot.

```
plot_setup <- ggplot(data = Auto, mapping = aes(x = cylinders, y = mpg))  # setup the plot
scatterplot <- plot_setup + geom_point()  # add a scatterplot to the setup
print(scatterplot)  # show what's been created
```



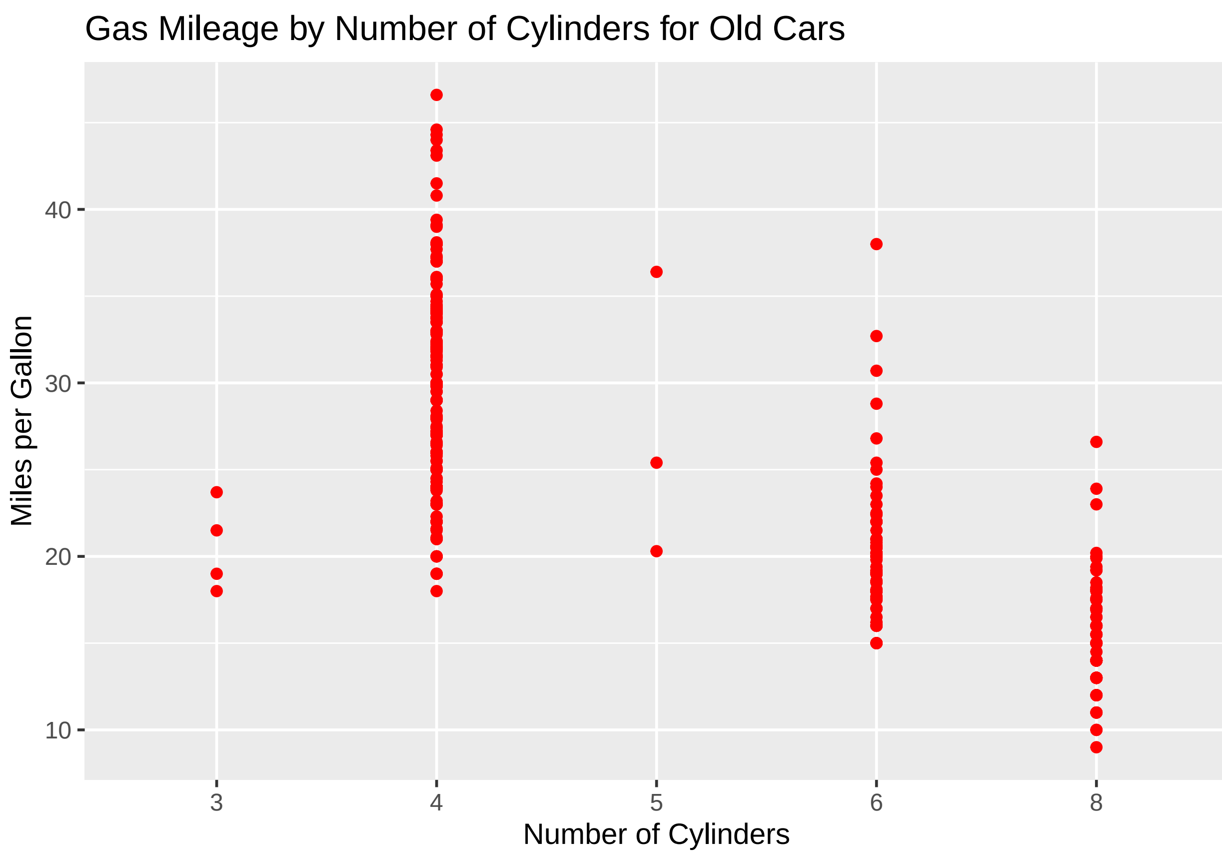
Yes, it took three lines to do what we did in one line in Problem 1. So why do we use ggplot2? Customization is much easier and intuitive in ggplot2. For example, to create the red points:

```
scatterplot_red <- plot_setup + geom_point(color = "red")
print(scatterplot_red)
```



We can also add a title and axis labels using the `labs` function:

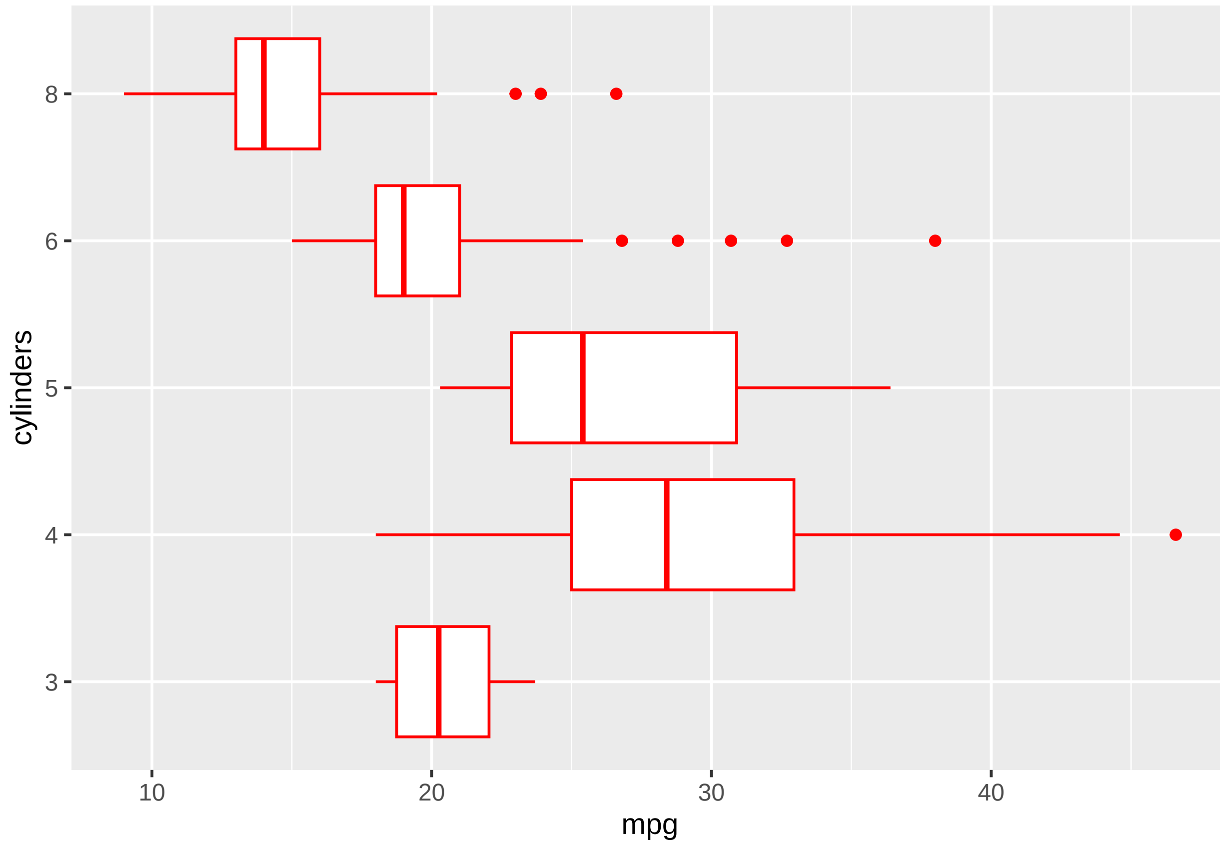
```
scatterplot_labeled <- plot_setup + geom_point(color = "red") +
  labs(title = "Gas Mileage by Number of Cylinders for Old Cars",
       x = "Number of Cylinders",
       y = "Miles per Gallon")
print(scatterplot_labeled)
```



Part b (Code: 0.5 pts)

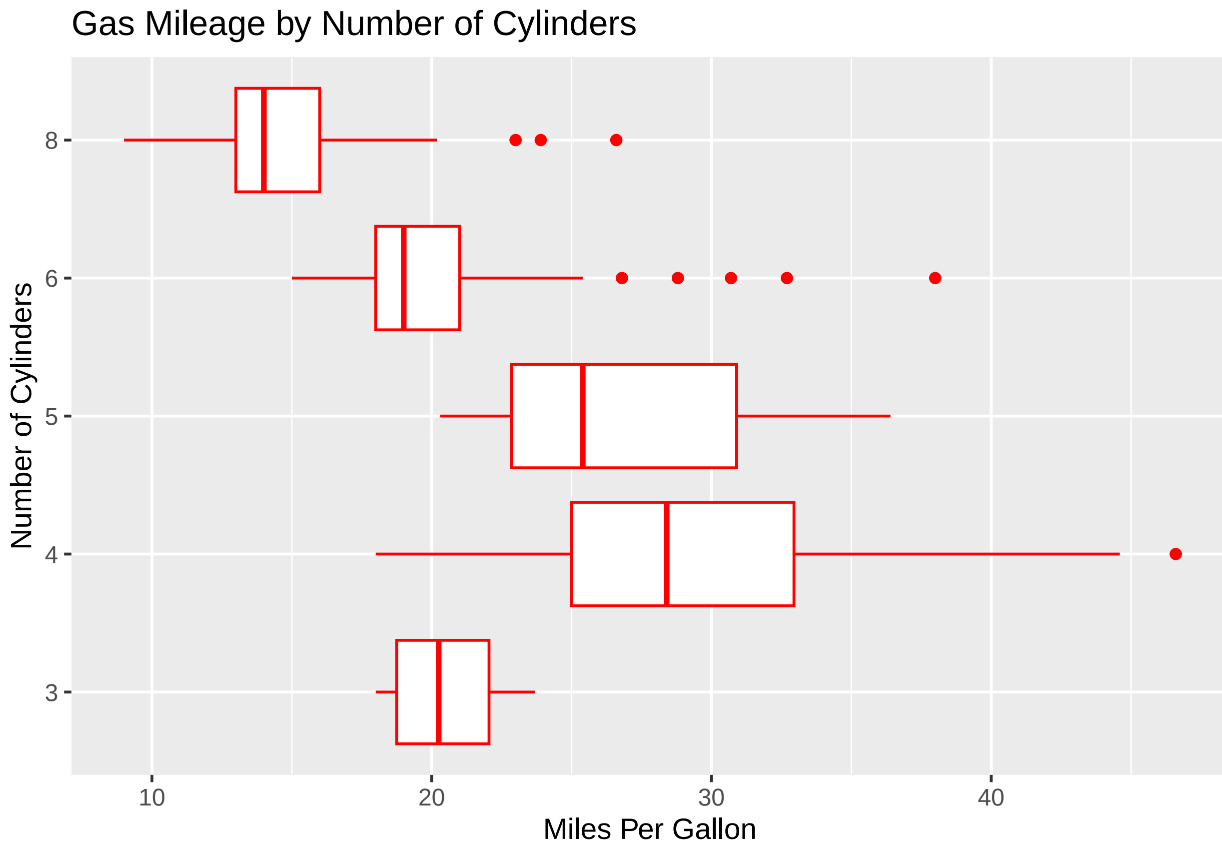
What about making a set of boxplots?

```
Auto.factor <- Auto
Auto.factor$cylinders <- as.factor(Auto.factor$cylinders) # convert to factor variable
boxplot_setup <- ggplot(data = Auto.factor, mapping = aes(x = cylinders, y = mpg))
boxplot_red_horizontal <- boxplot_setup + geom_boxplot(color = "red") + coord_flip() # flips what goes
print(boxplot_red_horizontal)
```



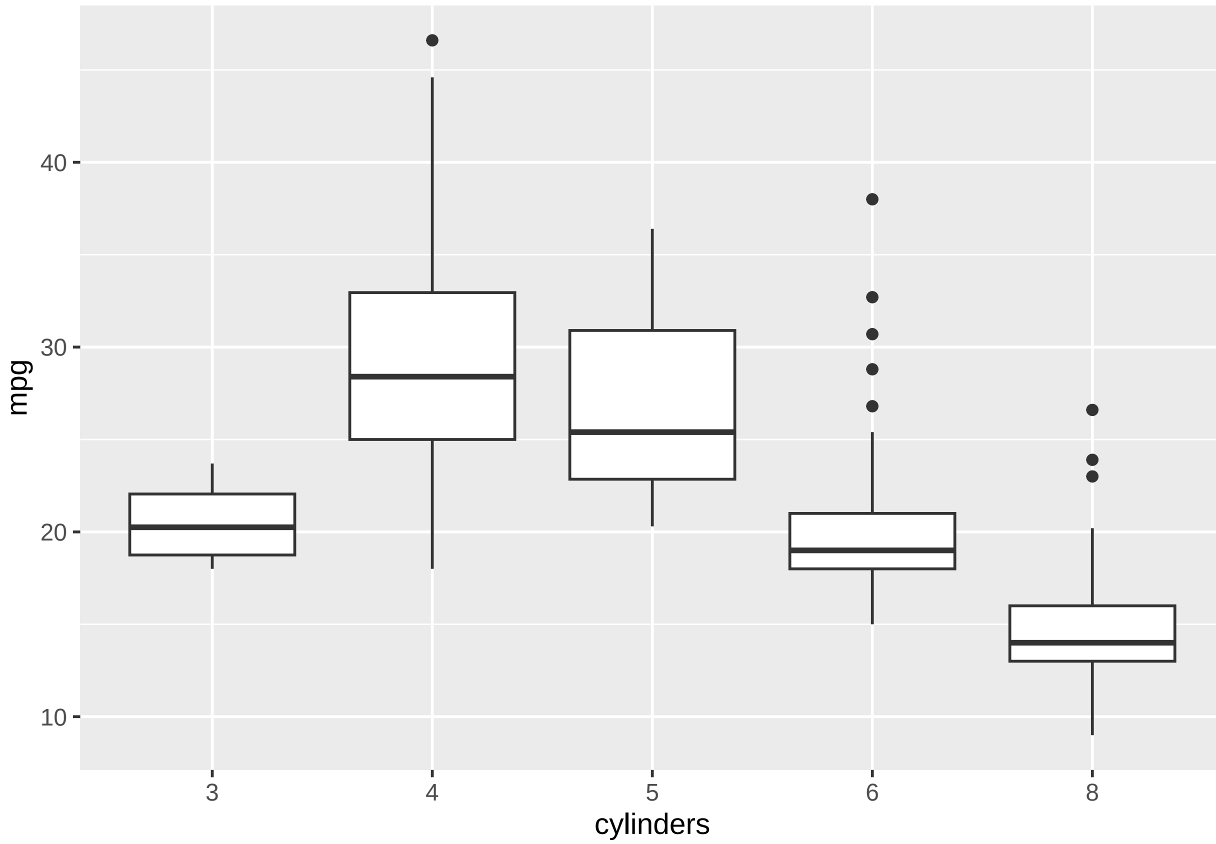
What about adding labels?

```
boxplot_labeled <- boxplot_red_horizontal +
  labs(title = "Gas Mileage by Number of Cylinders",
       x = "Number of Cylinders",
       y = "Miles Per Gallon") # yes, it's weird, you still need to pretend you didn't flip x and y
print(boxplot_labeled)
```



Sometimes, especially if you are trying to plot multiple datasets on the same graph, it is easier to specify the data and mappings in the `geom_` function itself:

```
plot_bare <- ggplot()  # no setup
boxplot2 <- plot_bare + geom_boxplot(data = Auto.factor, mapping = aes(x = cylinders, y = mpg))  # add
print(boxplot2)  # show what's been created
```



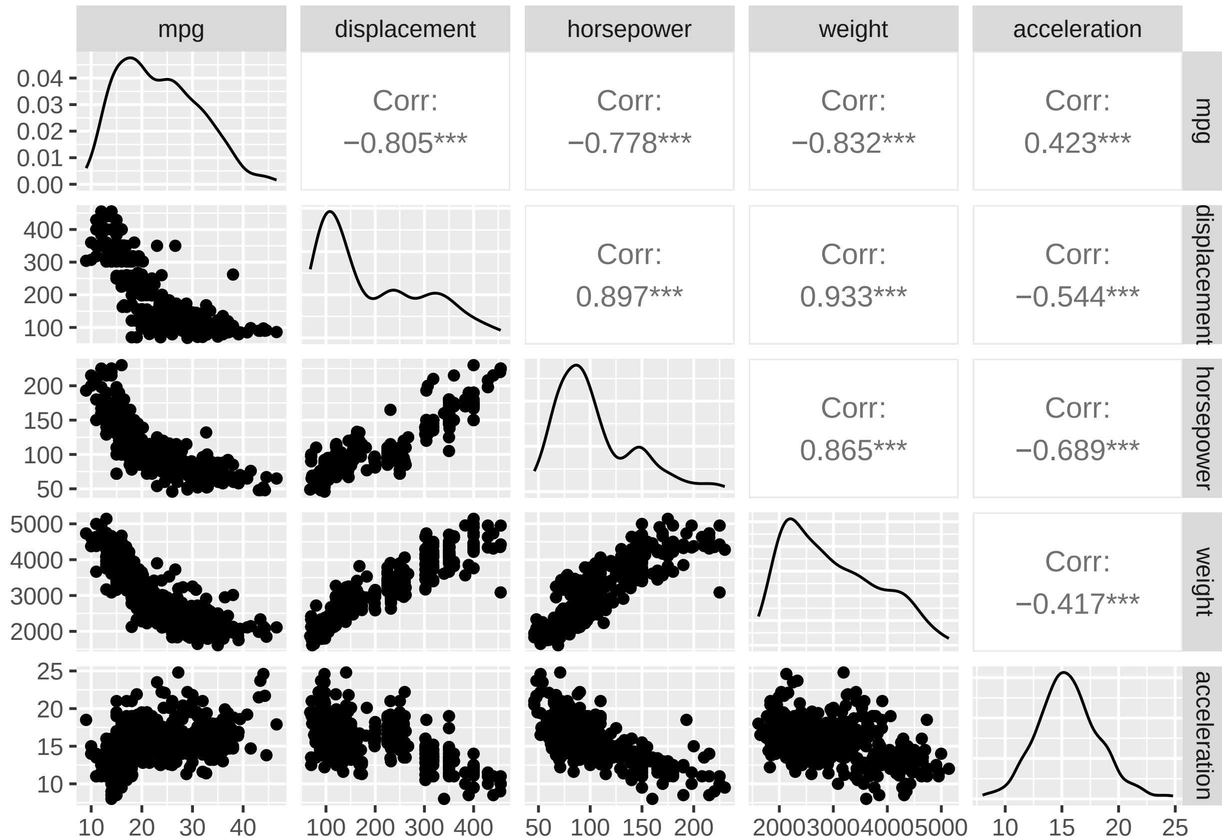
Part c (Code: 0.5 pts)

The *ggplot2* package cannot create a scatterplot matrix by itself, but the *GGally* package can. Install it and load it, then run the chunk below.

```
library(GGally)
```

```
## Registered S3 method overwritten by 'GGally':
##   method from
##   +.gg   ggplot2
```

```
ggpairs(Auto.factor, columns = c("mpg", "displacement", "horsepower", "weight", "acceleration"))
```


mpg
displacement
horsepower
weight
acceleration
Corr:
−0.805***
Corr:
−0.778***
Corr:
−0.832***
Corr:
0.423***
Corr:
0.897***
Corr:
0.933***
Corr:
−0.544***
Corr:
0.865***
Corr:
−0.689***
Corr:
−0.417***