Lab 2: The Linear Model in R and the Tidyverse

Isabel Laterzo, adapted from code and text from Simon Hoellerbauer and Chelsea Estancona

1/19/2021

This lab will introduce you to `lm()`, the function we will be using to fit linear models (for now) in `R`. In addition, it will provide a brief introduction to the `tidyverse` which is useful for data manipulation and cleaning in `R`. It will also cover material regarding using `ggplot`. I will walk you through the different parts of this lab, but then we will largely be workin independently. Chime in whenever you have questions!

Don't forget to set your working directory!

```r
setwd()
```

Part 1: LM in R

First, we'll simulate some data to use. This will be a very useful skill in the future, so pay attention to what we do here.

We first create our variables and then stipulate what our effects should be.

Let's set our seed to 123118, representing the date I adopted by dog!

For our $\mathbf{x}_1$ variable, let's take 20 draws from a $\mathcal{N}(1.3, 2.3)$ (using the `R` parameterization, where the second parameter is the standard deviation).

Let's set β_0 to be 17, and β_1 to be 1.3. We also can't forget to simulate $\boldsymbol{\epsilon}$! Let's $\boldsymbol{\epsilon} \sim \mathcal{N}(0, 3.4)$.

```r
set.seed(123118)

##Explanatory variables
x1 <- rnorm(__, __, __)

##Coefficients
b0 <- 17 #Intercept
b1 <- 1.3 #Beta 1

##standard deviation
sigma <- 3.4

error <- rnorm(length(x1), 0, sigma) #Let's simulate some random error, too.
#What would happen in the next step if we didn't simulate any error?
```

What would we do in order to simulate a linear relationship with the variables and effects above? Create the outcome variable `y`.

```r
y <- ______________________
```

What do $\mathbf{x}_1$ and $\mathbf{y}$ look like together? Use the `plot` function to plot a scatter plot of $\mathbf{x}_1$ against $\mathbf{y}$.

```
plot(x1, y)
```

Now that we have simulated explanatory variables and set up a 'true' relationship between our explanatory variable (x1) and outcome variable, we can use R's built-in functions to evaluate our linear model. For a linear model, we will want to use the `lm()` function. The basic syntax for this function is `lm(Y ~ X1 + X2)`. Use `summary()` on the model object (`m1` in this case) in order to see some of the most interesting information about the fitted model.

```
m1 <- lm(___________) #the basic syntax here: lm(Y ~ X1 + X2)
#summarize the 'model object' we've created
summary(m1)
```

Part 2: Combining LM and the tidyverse

Some of you might already use the `tidyverse` and its associated packages (if you do not have it installed yet, I suggest doing `install.packages("tidyverse")`) The tidyverse allows us to use the piping operator `%>%`, which is a useful piece of code. It pipes an object (either by itself or created by another function) into the next function in the sequence, as the first argument. This makes the code much easier to read. For example, we could calculate the standard deviation of the `mpg` variable in the `mtcars` dataset in the following ways:

```
library(tidyverse)
```

```
#normal
sd(mtcars$mpg)
```

```
## [1] 6.026948
```

```
#piping
mtcars$mpg %>% sd()
```

```
## [1] 6.026948
```

That is a very simple example, of course, and using a pipe in this case does not really gain you much. However, pipes and the myriad of tools in the `tidyverse` can make our work much, much easier. The `dplyr` package in the `tidyverse` allows us to manipulate data much more easily than in base R. `dplyr` allows us to manipulate our data in a structured way. It's really cool. The key functions are `mutate()`, which creates new variables, `select()`, which allows us to pick variable by characteristics of their names, `filter()`, which allows us to choose rows according to criteria we define, `summarise()`, which allows us to "reduce multiple values down to a single summary", and `arrange()`, which allows us to order the observations according values of 1 or more variables.

RStudio makes useful cheat sheets for their packages, including `dplyr`.

As we dive into learning linear models in `R`, we're going to use some data from the `mtcars`. But first, lets use the `tidyverse` to clean it up a bit. First, take a look at the mtcars data (its readily available so you don't really need to load it in!).

```
#examine data
summary(mtcars)
```

```
##       mpg             cyl             disp             hp       
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0  
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
##       drat             wt             qsec             vs        
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
##        am              gear            carb      
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
##  Median :0.0000   Median :4.000   Median :2.000  
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812  
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000  
```

Great. Let's say we want to analyze the role that `disp` and `cyl` play in influencing `mpg`. However, we're only interested in observations where `cyl` is greater than 4. Let's use `dplyr` and the `tidyverse` to only keep observations where `cyl` is greater than 4. Then, only keep the three above variables in the new data set. Label this data set `new_data`.

Hint: the `dplyr` functions you are interested in here are `filter()` and `select()` - don't forget to call their help files if you need more information (e.g., `?filter()` in your console).

```
new_data <- mtcars %>%
  filter(cyl > 4) %>%
  select(__, __, __)
```

Now that we have our data ready, we can use `lm()`. Fill in the blank in the chunk below where Y is `mpg`, X1 is `cyl`, and X2 is `disp`. Use `summary()` on the model object (`m2` in this case) in order to see some of the most interesting information about the fitted model.

Remember, our data is `new_data`.

```
#model
m2 <- lm(___________)

#summarize
summary(___)
```

The `summary` function provides lots of valuable information about the model we've created. We can also extract different information if we need, for example, just the coefficients, or just the residuals. Use `View()` on `m2`. What does it look like `m2` is? What do you think we could do if we wanted to access the different objects within `m2`?

```
#Taking a look at m1
____________

#extract coefficients
m2$_____

#extract residuals
m2$______

#extract y hat
m2$_______
```

We can also use indexing to get to parts of these vectors - the output from a model is a list object, which means we can use brackets with $ indexing.

```
m2$coefficients[1] #What does this give us?
m2[2] #What about this?
```

What if we wanted to include an interaction term? Try interacting `disp` and `cyl`.

```
m3 <- lm(mp ~ disp + cyl + _______, data=new_data) #use * for interaction- include both variables

summary(m3)
```

We don't actually have to write all of the variables two times; the `lm` function will include the variables involved in the interaction even if we don't specify them individually in the formula. Try this out below, creating model object `m4` and compare it to `m3`.

```
#m4:
_________
```

How might we present these results in a table? Both `xtable()` and `stargazer()` are functions that produce LaTeX code that could be copy-pasted into a .tex document or included in an Rmarkdown document. Try both of these on one of the model object `m4` we created above and see what happens. Note, these are VERY flexible and can be customized to make very nice tables, the ones we are generating here are just basic.

```
#These produce code for a .pdf that we can copy and paste
xtable(______)
stargazer(_______)
```

Part 3: ggplot

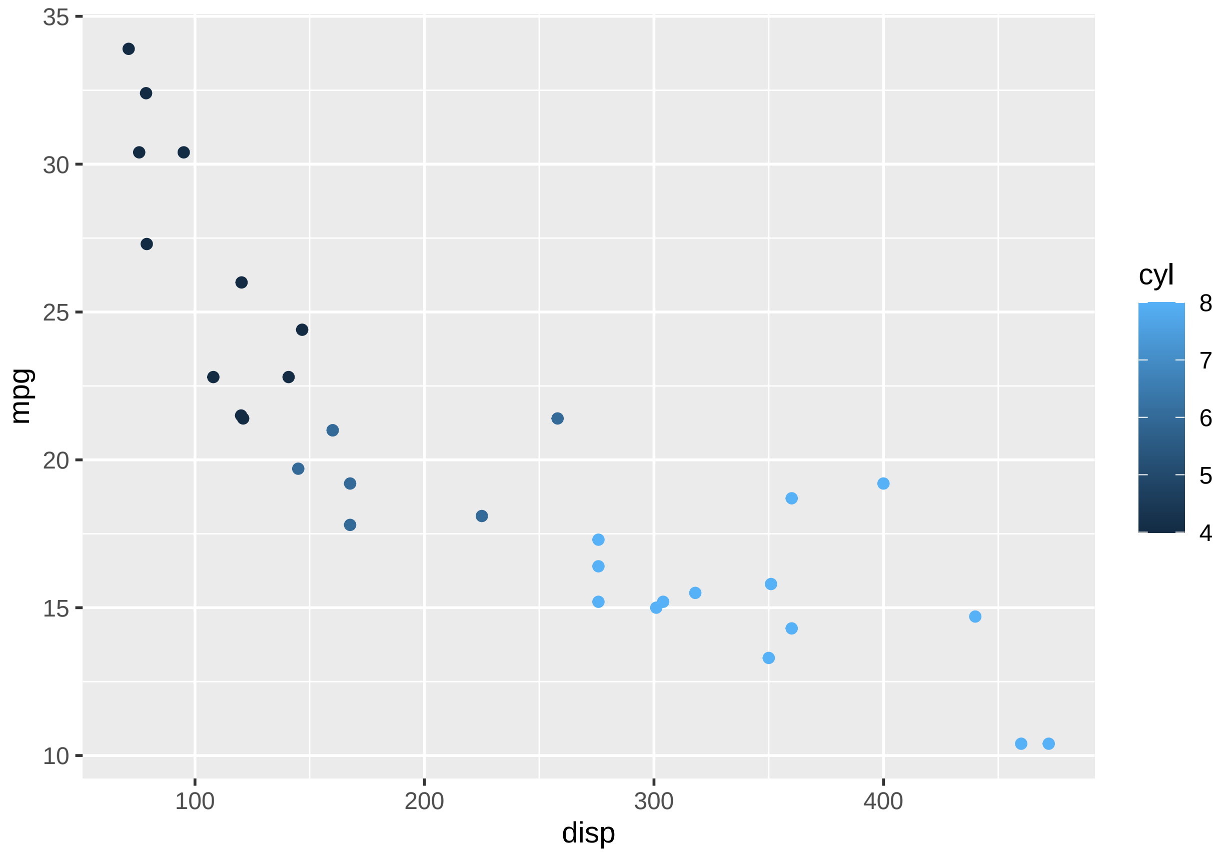
`ggplot` is very important, and requires some time and patience to get to know well. We won't have time to dive fully into it, but we will begin reviewing the following code so you become more familiar. R has its own built-in plot function, but ggplot is far more flexible and creates far more descriptive (and thus useful!) plots.

For a comprehensive 'cheat sheet', see: https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf

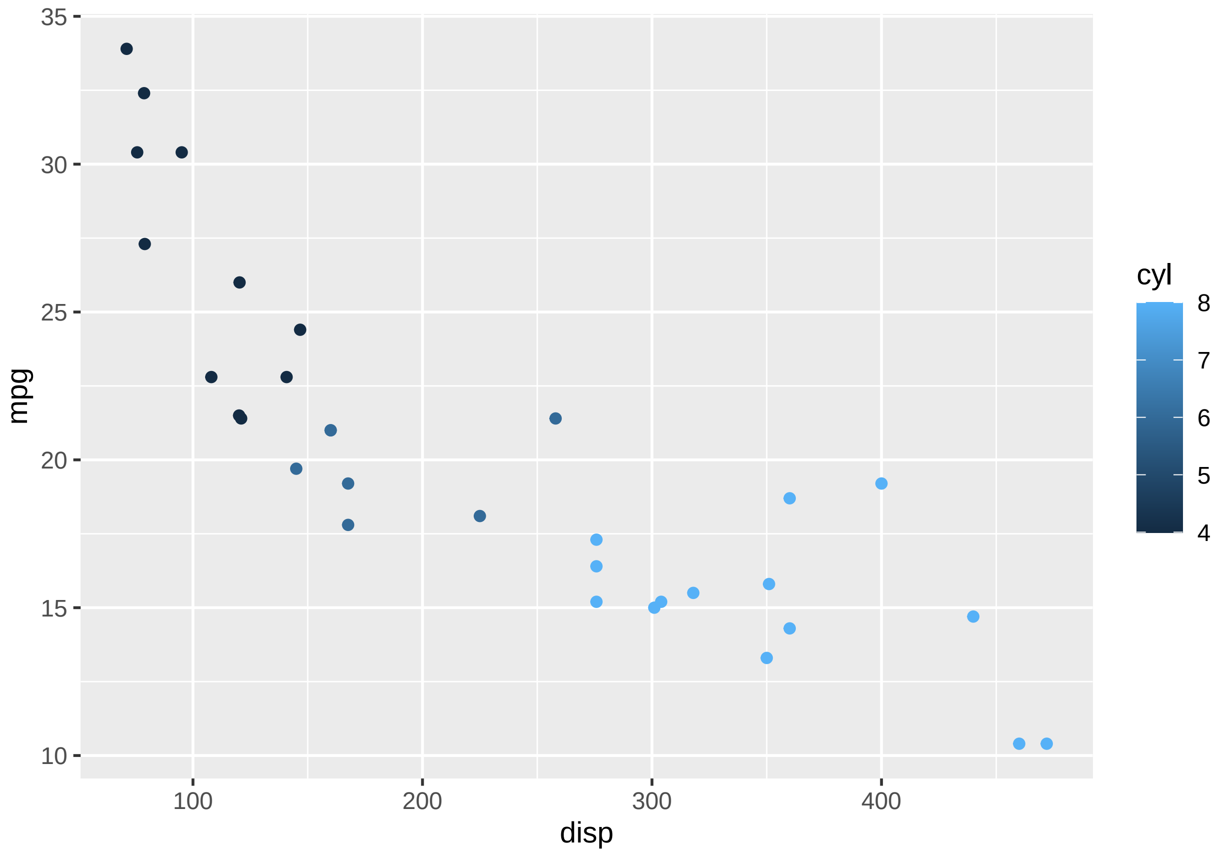
```
#we have to load the package first
library(ggplot2)
```

To begin, we can use the 'qplot' or 'ggplot' commands. I use ggplot. It's better.

```
data(mtcars)

qplot(x = disp, y = mpg, data = mtcars, color = cyl, geom = "point")
```



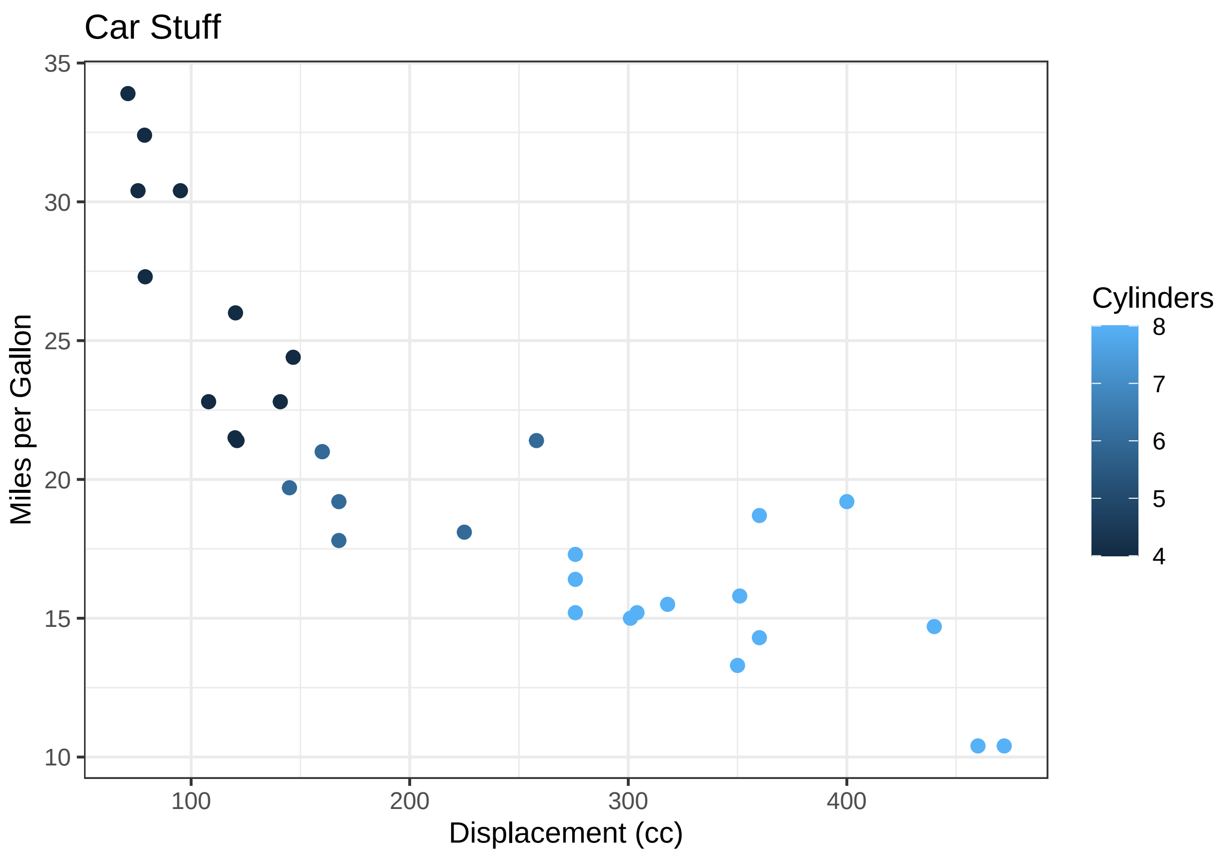
```r
#x and y are independent + dependent variables,
#data can be specified (or use mtcars$var),
#color distinguishes a characteristic over which to map color
#and geom is a plot type

ggplot(data = mtcars, aes(x = disp, y = mpg)) + geom_point(aes(color = cyl))
```



```
#Note the similarities above.
#ggplot is more flexible in that you map to different `aesthetics'-
#hence why we use 'aes.' Plenty of examples below to dig into this more!
```

What are some ways we can make this plot clearer?

```
ggplot(data = mtcars, aes(x = disp, y = mpg)) + #we add 'layers' with +
  geom_point(aes(colour = cyl), size = 2) + #point=scatterplot
  labs(title = "Car Stuff", x = "Displacement (cc)", y = "Miles per Gallon")+
  #label x and y axes, give the plot a main title. Note the quotes!
  scale_colour_continuous(name = "Cylinders")+ #label the scale
  theme_bw() #make the plot background blank w/grid lines
```



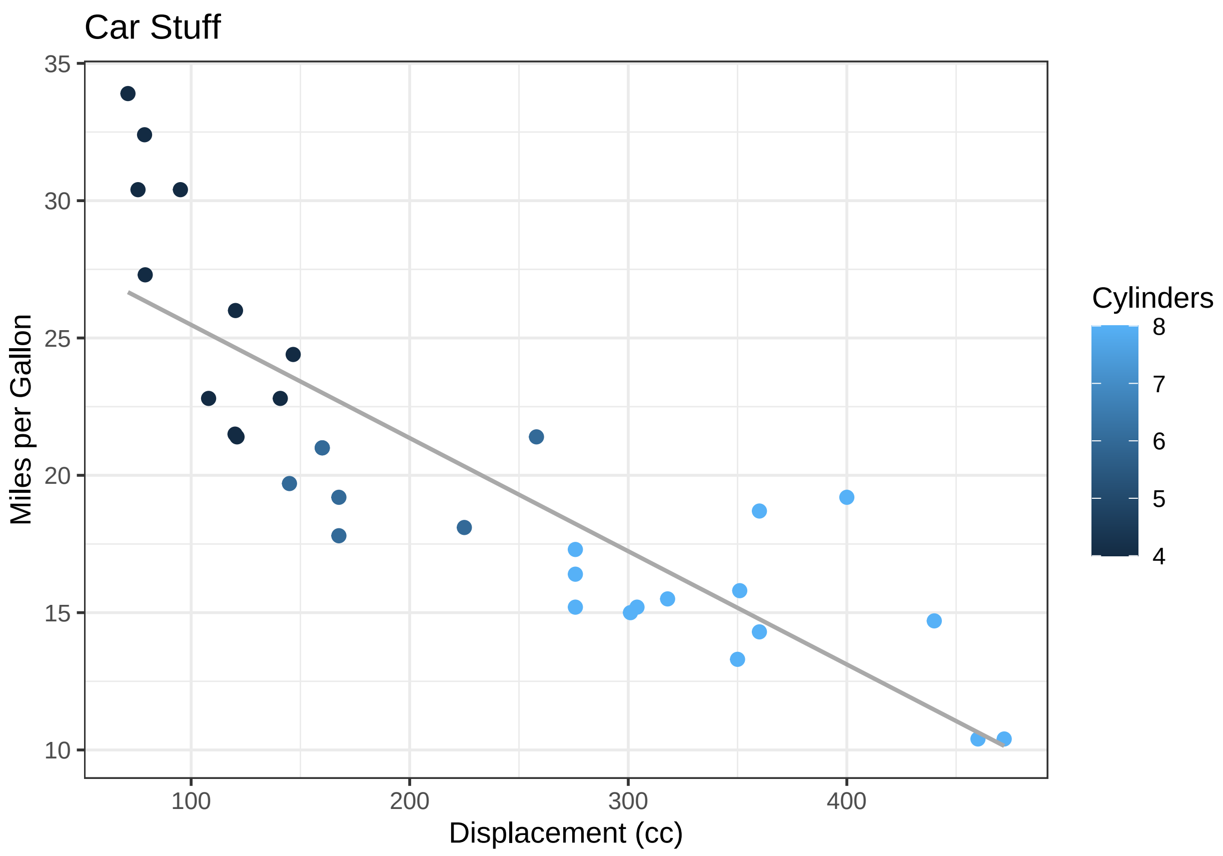
You can also assign this plot to an object and save that object as a .pdf.

```
carsplot <- ggplot(data = mtcars, aes(x = disp, y = mpg)) +
  geom_point(aes(colour = cyl), size = 2) +
  labs(title = "Car Stuff", x = "Displacement (cc)", y = "Miles per Gallon") +
  scale_colour_continuous(name = "Cylinders")+
  theme_bw()

#One method for saving plots:
#pdf("Plot1.pdf")
#carsplot
#dev.off()

#Where do these plots go?!

#Unique to ggplot:
#ggsave("Plot1a.pdf") #can edit image sizes as well
```

What if we wanted to plot the linear relationship between our x and y?

```
carsplot + geom_smooth(method = "lm", se = FALSE,
                       color = "darkgrey", size = 0.75)
```

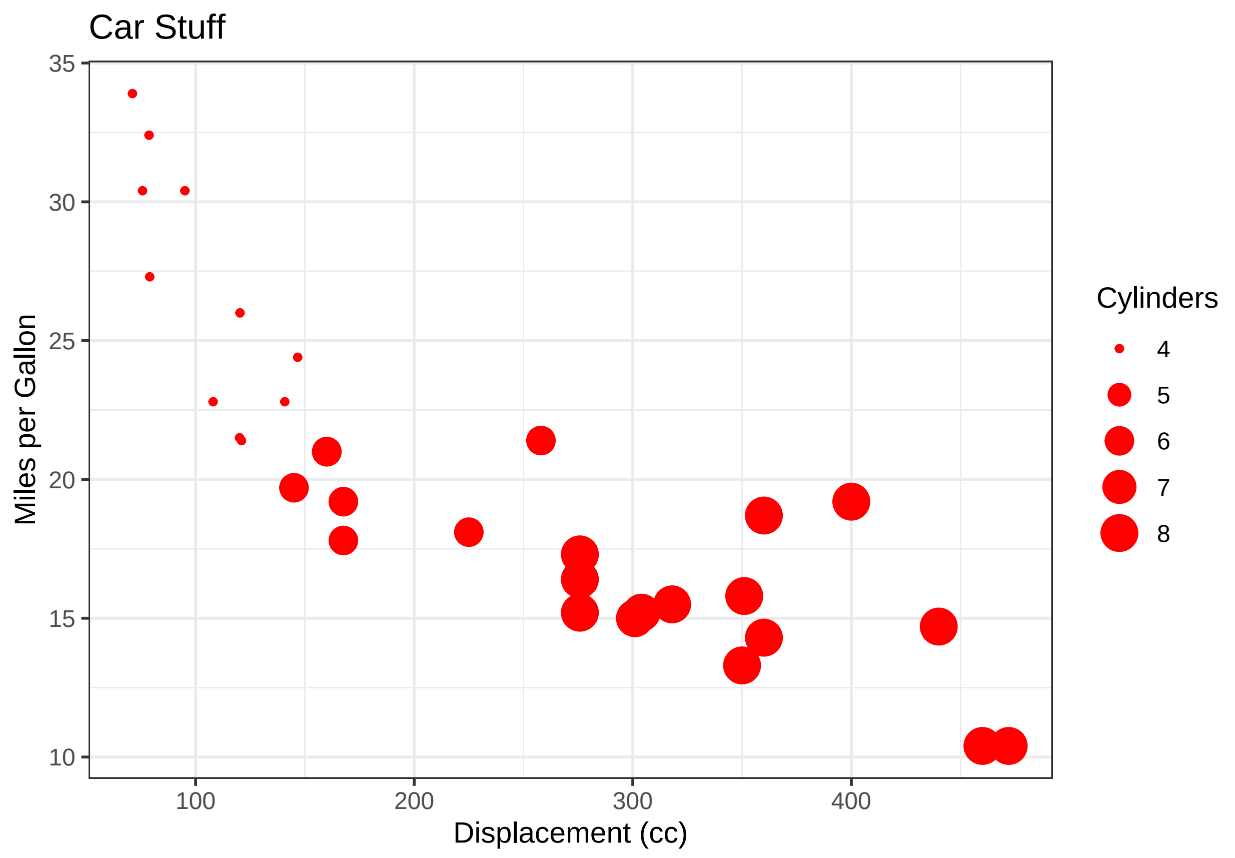
```
## `geom_smooth()` using formula 'y ~ x'
```



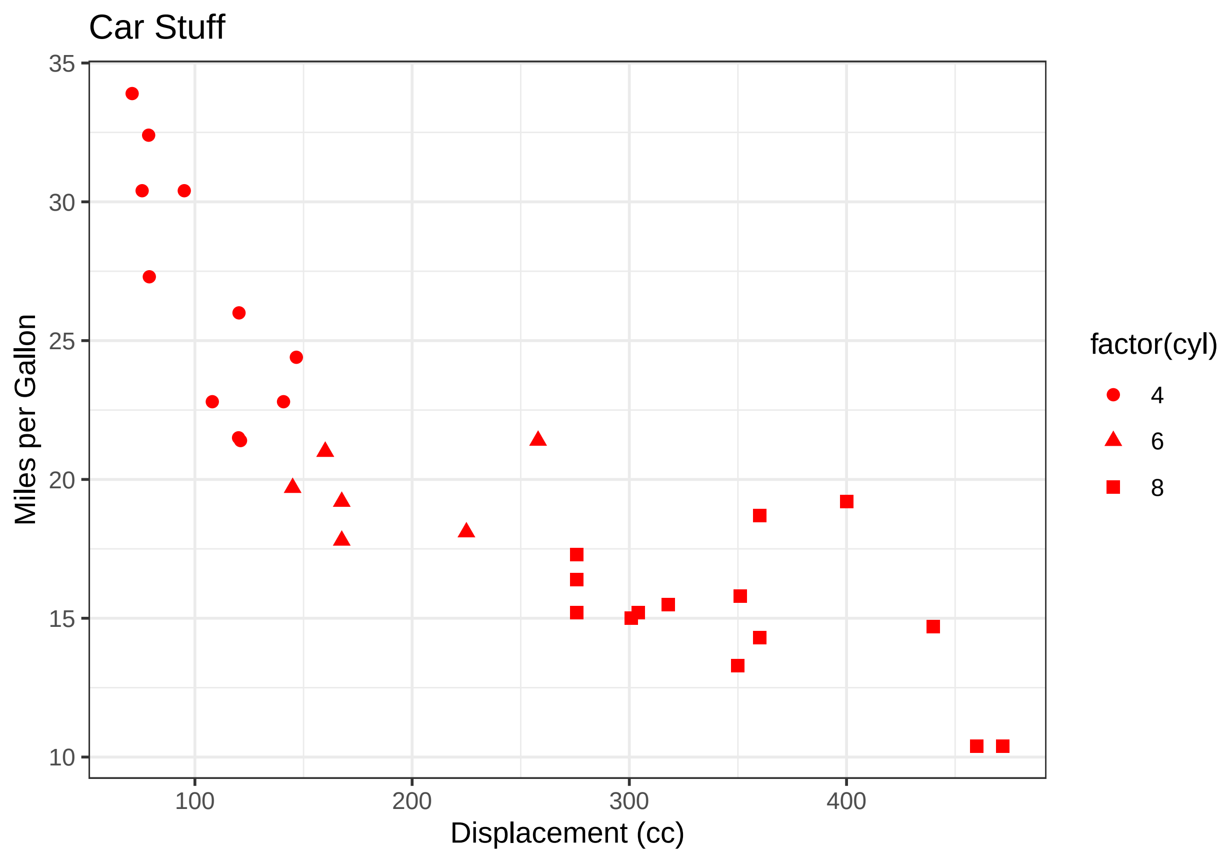
```
#geom_smooth is for fitted lines.
#se=TRUE plots confidence intervals
#We can adjust the color and size of the line plotted.
#Also, note that we just added a layer over carsplot - did we change the carsplot object itself?
```

Right now, we're using color as the aesthetic mapped to a specific explanatory variable (color represents the number of cylinders). We can play around with this:

```
ggplot(data = mtcars, aes(x = disp, y = mpg)) +
  geom_point(aes(size = cyl), colour = "red") +
  labs(title = "Car Stuff", x = "Displacement (cc)", y = "Miles per Gallon") +
  scale_size_continuous(name = "Cylinders") +
  theme_bw()
```



```r
ggplot(data=mtcars, aes(x = disp, y = mpg)) +
  geom_point(aes(shape = factor(cyl)), colour = "red", size = 2) +
  labs(title = "Car Stuff", x = "Displacement (cc)", y = "Miles per Gallon") +
  scale_size_continuous(name = "Cylinders") +
  theme_bw()
```



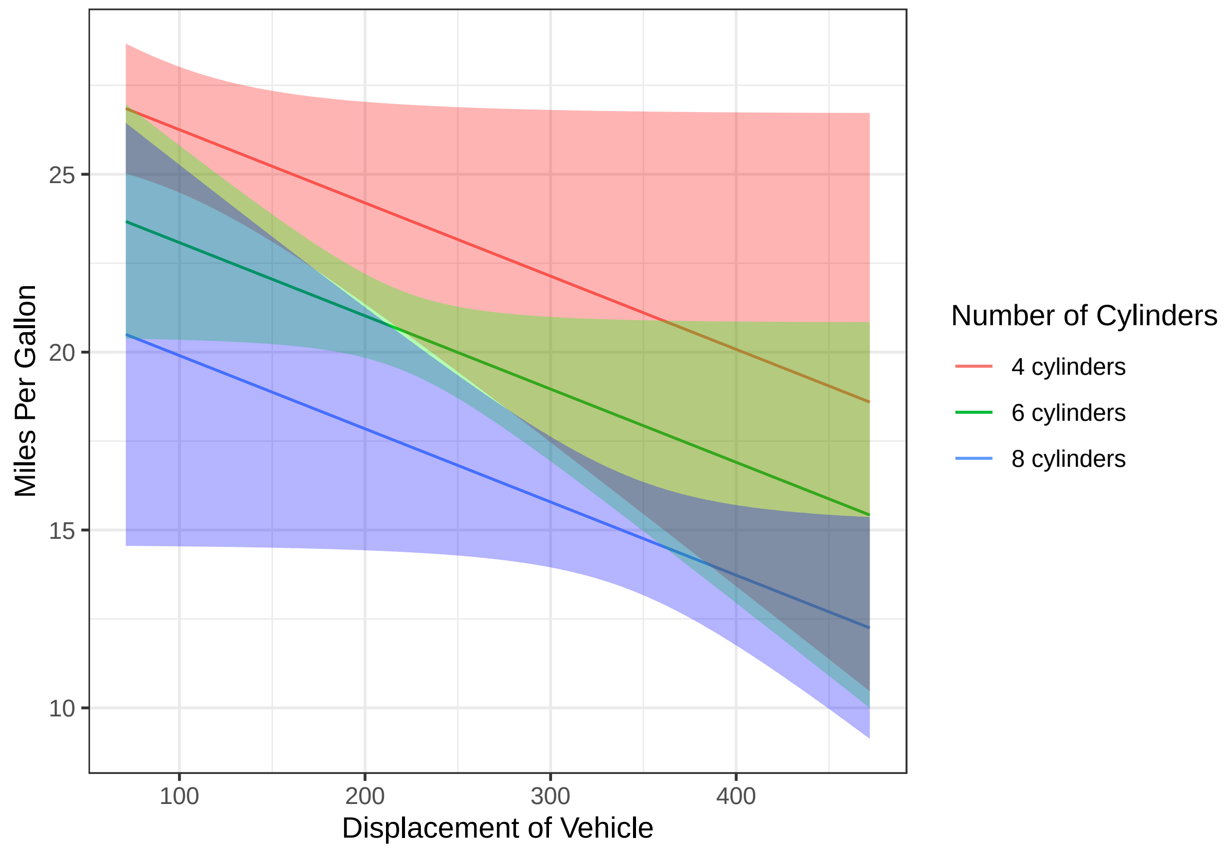
What are the benefits/drawbacks of each of these strategies? Think about how well the plot communicates information about the types of variables plotted as well as the relationship between them.

When presenting our results, we might be largely concerned with prediction: if I have a model and provide it with 'new' data, what should I expect? How confident are we in that expectation? We can use the handy `predict()` function here, which works with almost any type of model object. Check out `predict.lm` using ?, as that is the method that `predict` invokes when we feed it an lm object.

```r
#fitting our model
m5 <- lm(mpg ~ disp + cyl, data = mtcars)

#looking at our model
summary(m5)
```

```
##
## Call:
## lm(formula = mpg ~ disp + cyl, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.4213 -2.1722 -0.6362  1.1899  7.0516
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 34.66099    2.54700  13.609 4.02e-14 ***
## disp        -0.02058    0.01026  -2.007   0.0542 .
## cyl         -1.58728    0.71184  -2.230   0.0337 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.055 on 29 degrees of freedom
## Multiple R-squared:  0.7596, Adjusted R-squared:  0.743
## F-statistic: 45.81 on 2 and 29 DF,  p-value: 1.058e-09
```

```r
#what are we doing here? Also, what does with allow us to do here?
x_vec_n <- with(mtcars, seq(min(disp), max(disp), length.out = 500))

#why am I choosing these values? What are other options?
new_dat_n_1 <- with(mtcars, data.frame(x_vec_n, 4))
new_dat_n_2 <- with(mtcars, data.frame(x_vec_n, 6))
new_dat_n_3 <- with(mtcars, data.frame(x_vec_n, 8))

#what are we doing here?
colnames(new_dat_n_1) <- c("disp", "cyl")
colnames(new_dat_n_2) <- c("disp", "cyl")
colnames(new_dat_n_3) <- c("disp", "cyl")
#this is very important for predict! See note below.


#getting our predicted values
fit_n_1 <- predict(m5, newdata = new_dat_n_1, interval = "confidence")

fit_n_2 <- predict(m5, newdata=new_dat_n_2, interval="confidence")

fit_n_3 <- predict(m5, newdata=new_dat_n_3, interval="confidence")

#ggplot is rather particular about the format of the data you will be using
plot_dat_n <- data.frame(fit_n_1, fit_n_2, fit_n_3, new_dat_n_1)
#note we never call `cyl` on its own below so the iteration of `new_dat_n_x` we include
#here doesn't matter a ton, it just gives us all values of disp

#this looks like a monster plot, but there are just a lot of layers. A lot of
#ggplot plot code will end up looking like this
ggplot(data=plot_dat_n) +
  labs(x = "Displacement of Vehicle", y = "Miles Per Gallon")+
  geom_line(aes(x = disp, y = fit_n_1[, 1], color = "4 cylinders"),linetype = "solid") +
  geom_line(aes(x = disp, y = fit_n_2[, 1], color = "6 cylinders"),linetype = "solid") +
  geom_line(aes(x = disp, y = fit_n_3[, 1], color = "8 cylinders"), linetype = "solid") +
  #what are we doing with the ribbons? What does alpha do? Play around with it.
  geom_ribbon(aes(x = disp, ymin = fit_n_1[, 2], ymax = fit_n_1[, 3]), alpha = .3, fill = "red") +
  geom_ribbon(aes(x = disp, ymin = fit_n_2[, 2], ymax = fit_n_2[, 3]), alpha = .3, fill = "green") +
  geom_ribbon(aes(x = disp, ymin = fit_n_3[, 2], ymax = fit_n_3[, 3]), alpha = .3, fill = "blue") +
  theme(legend.title = element_text(size = 14),
        legend.position = "bottom",
        panel.grid.minor.x = element_blank(), #what does this do?
        panel.grid.minor.y = element_blank()) +
  scale_color_discrete(name="Number of Cylinders") + #what does this seem to do? what happens if you ta
  theme_bw()
```



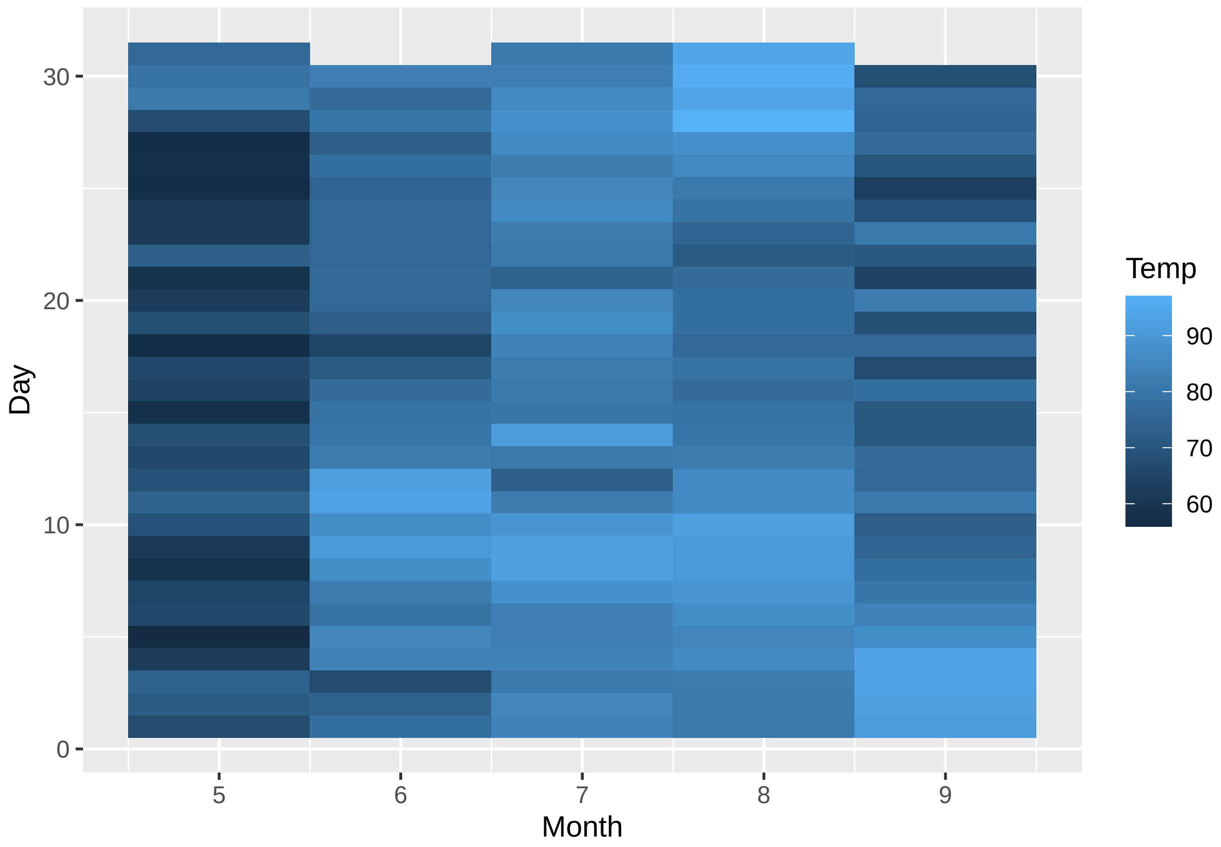
Note that when using predict, it is very important that our new observations have variables that bear the same name as in the formula. Otherwise `predict` won't know what to do! Try out the following:

```
#recreating new_dat_n_1
new_dat_n_1_a <- with(mtcars, data.frame(x_vec_n, 4))

#but this time we don't label the variables
predict(m5, newdata = new_dat_n_1_a, interval = "confidence")

#and this time we label them incorrectly
colnames(new_dat_n_1_a) <- c("cisp", "dyl")
predict(m5, newdata = new_dat_n_1_a, interval = "confidence")

#and this time we flip the variables
new_dat_n_1_a <- with(mtcars, data.frame(4, x_vec_n))
colnames(new_dat_n_1_a) <- c("disp", "cyl")
predict(m5, newdata = new_dat_n_1_a, interval = "confidence")
#this time it works, but our predictions are clearly incorrect (compare them to
#fit_n_1 from above)

#this works!
new_dat_n_1_a <- with(mtcars, data.frame(4, x_vec_n))
colnames(new_dat_n_1_a) <- c("cyl", "disp")
fit_n_1_a <- predict(m5, newdata = new_dat_n_1_a, interval = "confidence")

all(fit_n_1 == fit_n_1_a) #what does all() do here?
```

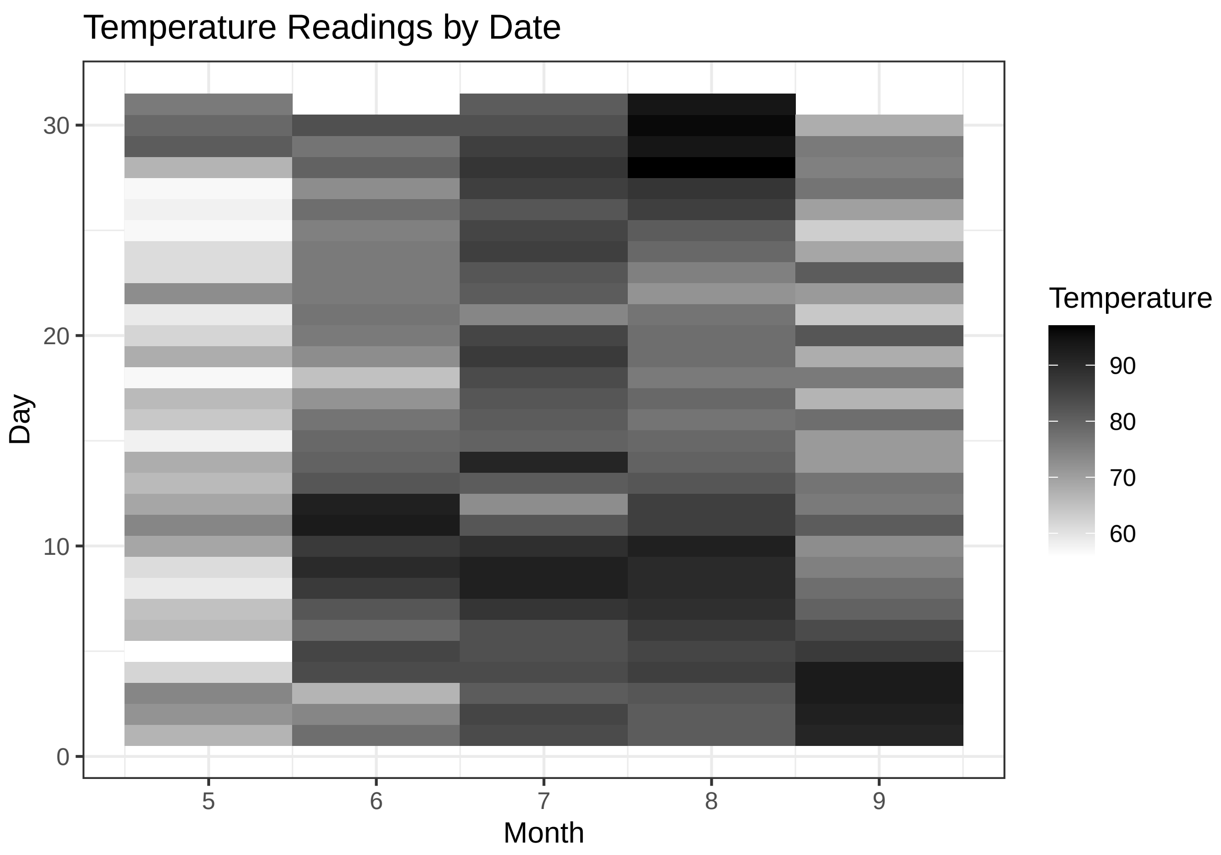
Sometimes its also useful to make "heatmaps" or levelplots (this word should look familiar). Let's play around with this with the `airquality` data set.

Let's say we want to look at how temperature varies, but both month and day. We could do the following:

```
data(airquality)

#basic plot
airqual <- ggplot(airquality, aes(x = Month, y = Day, fill = Temp)) +
  geom_tile()

airqual
```



```
#let's make it less colorful and add some axis labels
new_airqual <- airqual + scale_fill_gradient(name = 'Temperature', low = 'white', high = 'black') +
  theme(legend.position="none")+
  xlab("Month")+ylab("Day")+labs(title="Temperature Readings by Date") +
  theme_bw()

new_airqual
```



What if we want to add a specific point showing a value of interest? `geom_point()` becomes useful. Let's put a point on July 20th. Try it out!

```
even_newer_airqual <- new_airqual + geom_point(x = ___, y = ____, color = "green")

even_newer_airqual
```