Data Science 101: Introduction to Data Science for the Social Sciences
Introducing Visualization

Welcome to our first substantive topic: visualization! Visualization just means that we'll be making graphs and other displays (visuals) to help us learn about our data. This handout gives the code you'll need, along with comments. It's designed to accompany the lecture online. I'll be walking us through examples of how to use R and the tidyverse to accomplish our goal. I'll include some graphs in this handout, but I've kept them small to keep the document legible. You should do this with R open on your computer so you can run the code yourself and see them more clearly.

I've uploaded my command file to Canvas so that you have it as a reference. But to learn this material, you need to be writing the code yourself. Just use my file as a reference in case you can't figure out why your code isn't working. If you get stuck, remember to pull code from this file, not the handout – extra symbols that print in the handout may get in the way of your code running.

By the end of this lesson, you will understand how to use R to draw one-variable and two-variable graphs in a variety of different contexts. You'll see some of options one can use to change the graph to more effectively present the data. We'll do more with visualization throughout the term, but this is a great way to get started.

As a reminder from our last lecture, data science involves learning by doing. For this to make sense, you should be using these handouts, and working through the lectures with me. Pause the video and make sure the code runs on your end. If it doesn't, carefully compare the code, paying particular attention to spelling, parentheses, and such (again, use my file as a reference). Computers are brilliant, but they're also really dumb: if your variable is called class, and you write clas, the computer has no idea what you mean. Much of the time when I make an error, it's because of something like that.

If you check that and it isn't working, Google is your friend. Lots of people use R, and there are tons of helpful folks online, especially at places like StackOverflow (https://stackoverflow.com). StackOverflow even has a special section where they've pulled together all of the posts they have related to R (https://stackoverflow.com/questions/tagged/r), but I usually find it easier to just go to their site and search. And don't forget that you can also ask the instructor/TAs. The key thing is to make sure you understand. Watching the lectures and reading handouts won't do anything if you are not actively following along!

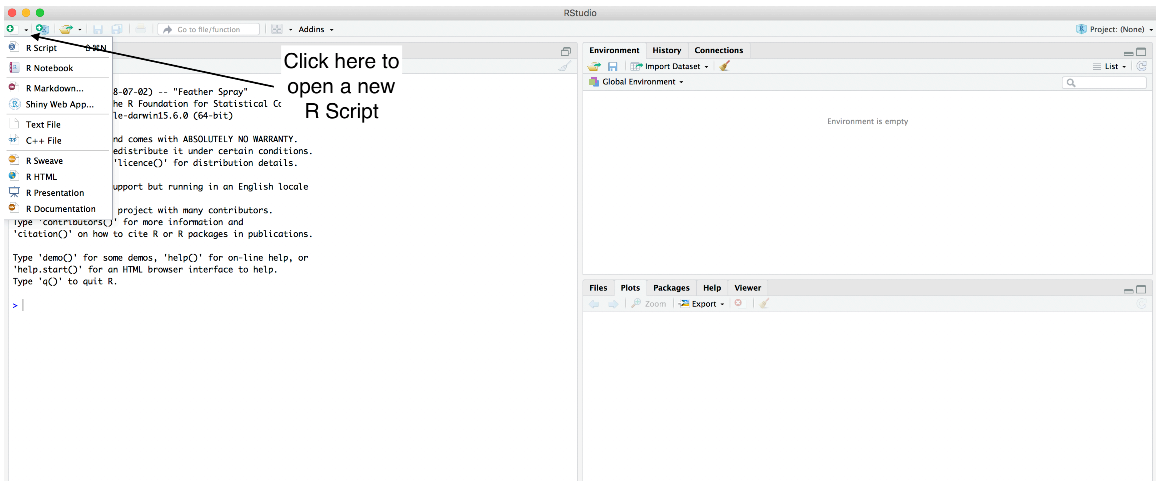
I'll also call out some self-test problems in what appears below. I'll say **TEST YOURSELF**, and then ask a question. Don't just look at my answer, try to answer this on your own! Otherwise, there's no value in them.

In what appears below, I'll put notes and comments in Times New Roman font (the one I'm using now), and I'll use `the Courier New font` for R code and results so you can tell them apart.

We'll work through the same dataset as Chapter 3 of the textbook, the `mpg dataset`, which is data from the EPA on fuel efficiency for 38 popular car models in 1999 and 2008. We'll see how we can use this relatively simple data to highlight how R lets us create graphs to learn about our data!

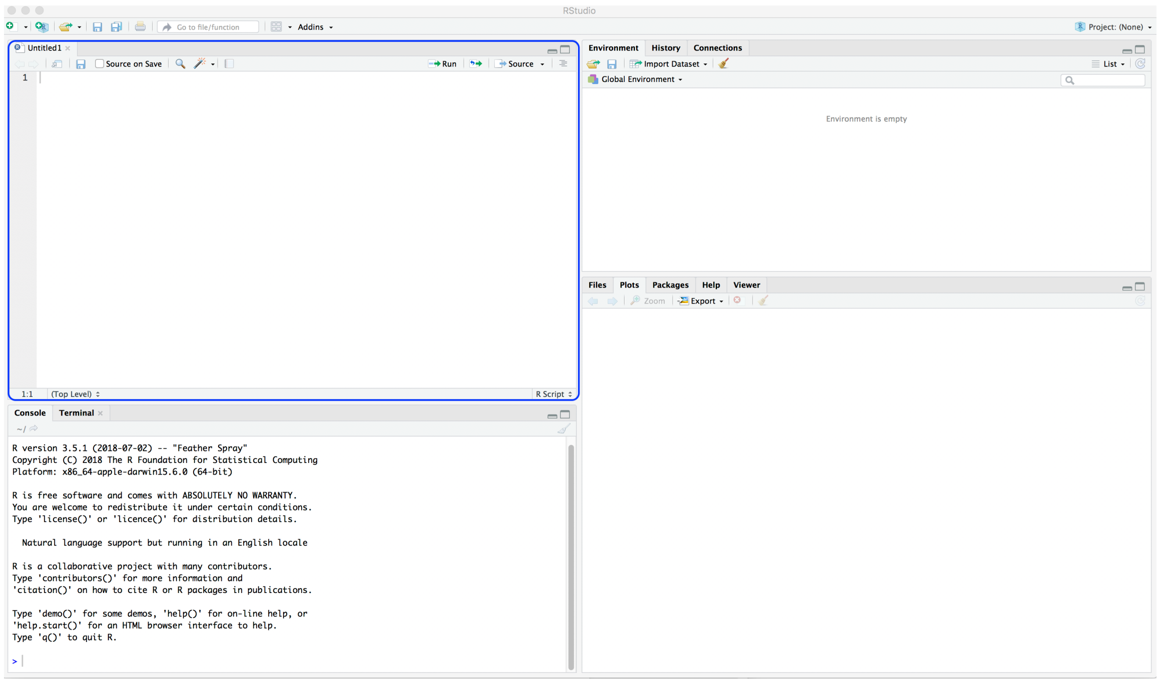
Before we begin, remember that you should be saving your commands so you have a record of what you've done. You can store this in your "code" folder: so create a file in ~/Dropbox/DATA101/Code. Naming this logically helps too, so I might name this file "visualization_code.r", for example.

To create a new code file in RStudio, I would open a new R Script. To do this, click on the green plus in the upper-left hand corner of RStudio where the arrow is pointing in the screenshot below.



```
8-07-02) -- "Feather Spray"
he R Foundation for Statistical Cc
le-darwin15.6.0 (64-bit)

nd comes with ABSOLUTELY NO WARRANTY.
edistribute it under certain conditions.
'licence()' for distribution details.

upport but running in an English locale

 project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

That will open a new R Script, which is where we type our commands. Once I click on the "R Script" option, my RStudio environment will look like this:

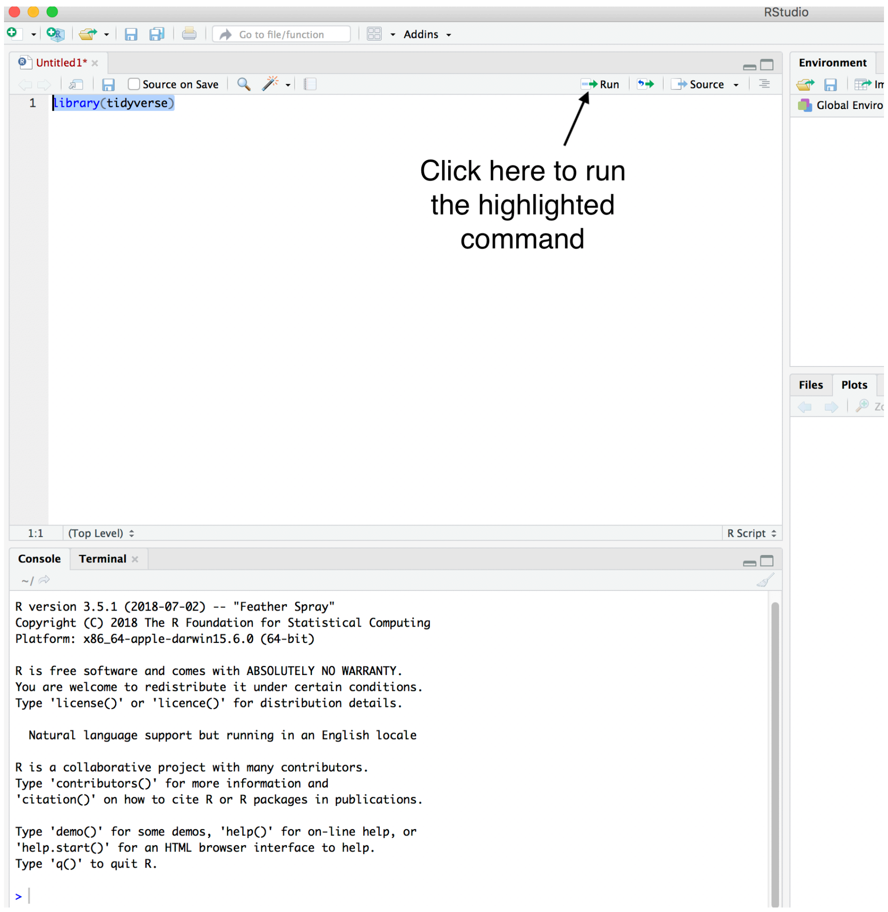


```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

You add the commands to the upper left-hand window (the space highlighted in blue). RStudio makes it very easy to run those commands. When I'm ready to run a command, it becomes very easy to execute it in R, as I'll demonstrate below.

Let's start with a simple example. What is the relationship between engine size and fuel efficiency: how are these two variables related? What would you predict? You might expect that cars with larger engines would use more fuel. But how strong is this relationship? Is it linear (i.e., the relationship follows a straight line)? Nonlinear? Let's visualize it to find out!

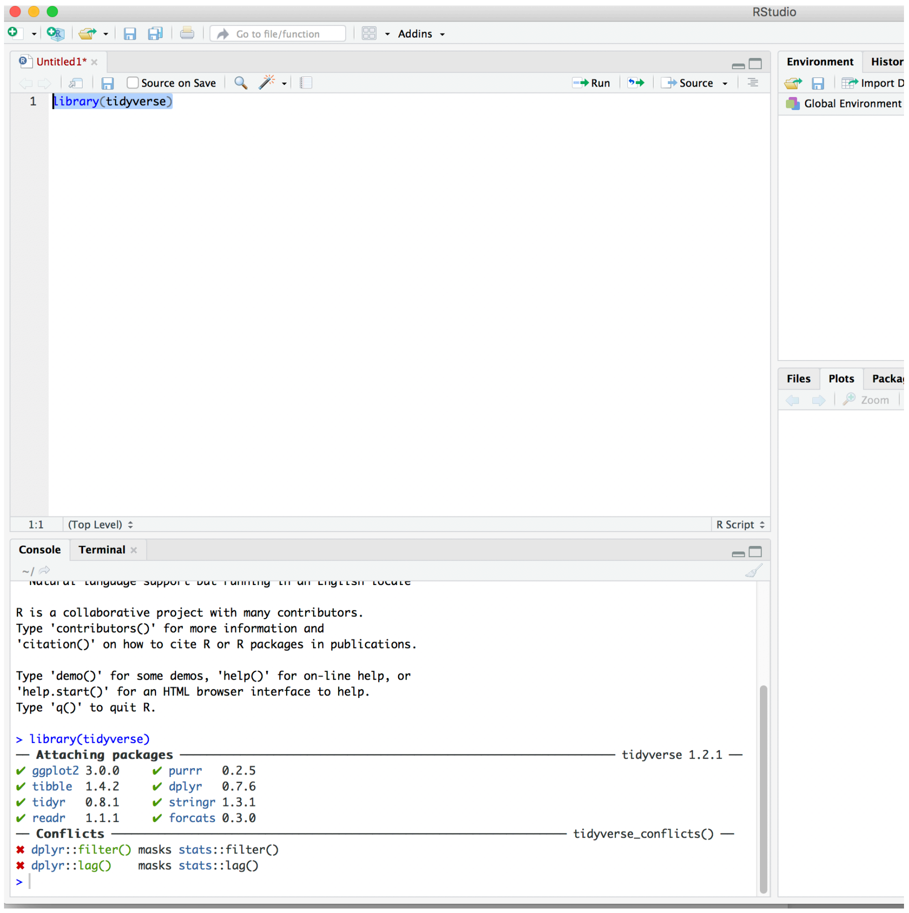
To use this data, we'll need to call R. Recall back to our last lecture where we installed R and installed the tidyverse environment. To begin, we load the tidyverse library. To do this, I'll type the command in my R Script, highlight it, and then click the run button:



```
library(tidyverse)
```

```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

When I run the command, the output transfers to the R console in the lower left-hand panel:



```
library(tidyverse)
```

```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> library(tidyverse)
── Attaching packages ─────────────────────────────────────── tidyverse 1.2.1 ──
✔ ggplot2 3.0.0     ✔ purrr   0.2.5
✔ tibble  1.4.2     ✔ dplyr   0.7.6
✔ tidyr   0.8.1     ✔ stringr 1.3.1
✔ readr   1.1.1     ✔ forcats 0.3.0
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
>
```

So this loads the tidyverse() and it's associated libraries. Today, we'll be making use of just `ggplot2()`, which uses R to visualize data. Later in the term, we'll discuss the other packages and their roles. For now, don't worry about the section labeled "conflicts."

By the way, if you were wondering why `ggplot()` is called ggplot (which seems like an odd name), it's because "ggplot" stands for grammar of graphics plotting, where the grammar of graphics is a way of organizing plots. For more information, see http://vita.had.co.nz/papers/layered-grammar.html.

If you don't see what I have above, and instead see:

```
Error in library(tidyverse) : there is no package called
'tidyverse'
```

That means that you did not install the tidyverse library. As we discussed previously, you need to install a package using the `install.packages()` command before you use it in R. So make sure you run `install.packages("tidyverse")` on your machine. Note that you must be connected to the Internet for `install.packages()` to work (it installs the package from an online repository).

You only need to run `install.packages()` once, but you need to run library() in every session where you use the commands from a library. So for example, if I quit and re-start R, I'll need to run `library(tidyverse)` again to use the commands below.

I'll stop showing these screenshots since they take up a lot of space and just show you the code for now. But remember that you can always watch the video to see the commands running in R in real time.

The mpg data is automatically included as part of the tidyverse, so we don't have to load it separately. Let's begin by exploring the mpg dataset. We can learn about data by typing the name of the dataset:

```
> mpg
# A tibble: 234 x 11
   manufacturer model      displ  year   cyl trans      drv     cty   hwy fl    class  
   <chr>        <chr>      <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>  
 1 audi         a4           1.8  1999     4 auto(l5)   f        18    29 p     compact
 2 audi         a4           1.8  1999     4 manual(m5) f        21    29 p     compact
 3 audi         a4           2    2008     4 manual(m6) f        20    31 p     compact
 4 audi         a4           2    2008     4 auto(av)   f        21    30 p     compact
 5 audi         a4           2.8  1999     6 auto(l5)   f        16    26 p     compact
 6 audi         a4           2.8  1999     6 manual(m5) f        18    26 p     compact
 7 audi         a4           3.1  2008     6 auto(av)   f        18    27 p     compact
 8 audi         a4 quattro   1.8  1999     4 manual(m5) 4        18    26 p     compact
 9 audi         a4 quattro   1.8  1999     4 auto(l5)   4        16    25 p     compact
10 audi         a4 quattro   2    2008     4 manual(m6) 4        20    28 p     compact
# ... with 224 more rows
```

This is actually quite informative. Don't worry about what a tibble is for now (it's R-speak for a data set), but this is telling us that there are 234 rows and 11 columns to our data: we have 234 observations (where each observation is a particular make of car in a particular year), and 11 variables. By default, we only see the first 10 rows for easy viewing. Typically in a data set, the rows are observations, and columns are variables.

Let's look at the first few observations (the first few rows). Our first observation is the 1999 Audi A4 with an automatic transmission, the second observation is the 1999 Audi A4 with a manual transmission, and so forth.

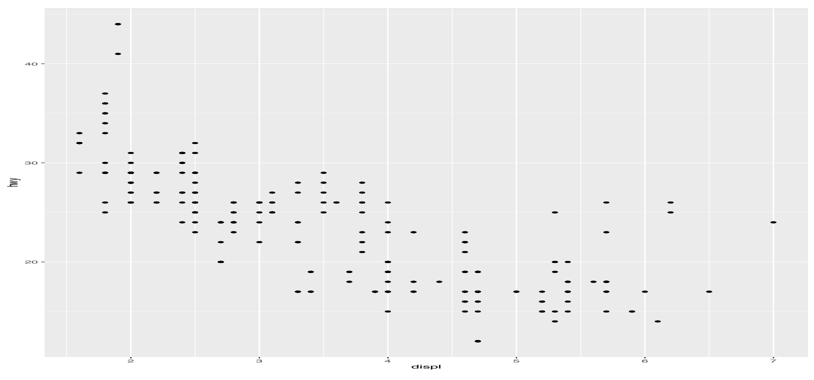
If we look at the first few variables (the first few columns), we see that the first variable is the Manufacturer of the automobile, and that it's a <chr> variable, meaning that its value is stored in characters, rather than numbers. This makes sense: the manufacturer of a car is word, rather than a numeric value. The same is true for the second variable, model (another text variable). The third variable, displ, which measures the engine displacement (or size of the engine), is a <dbl> (double) variable, which is a numeric value (a double is just a way of recording numbers with decimals in them). Note that year and cyl are both <int> or integer variables. Integers are just whole numbers like 6 or 1999. We could continue across in the same fashion looking at the other variables.

To learn more about this datasets, we run the command `?mpg`. Note that in our "help" window in RStudio, we now see the documentation for this dataset, which tells us more about the variables. Anytime we want to know more about a dataset, we can type ?NAME, where NAME is the name of the dataset.

Now that we know a tiny bit about our data, let's go back to the question we raised above: do larger engines use more fuel? Let's look at the data! In particular, let's look at the relationship between engine size (displ, or engine displacement) and fuel efficiency (hwy, for highway gas mileage):

```
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy))
```

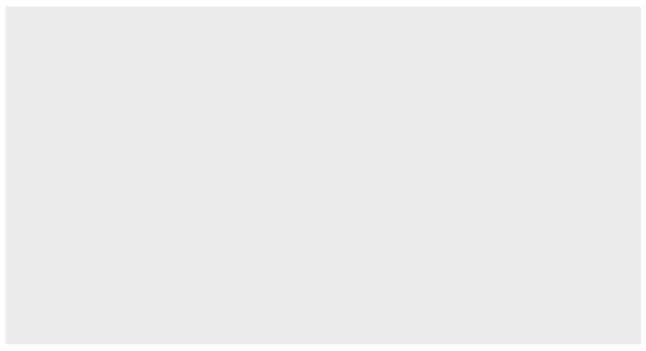
which produces the plot below:



This is a scatterplot with engine size (displ) on the x-axis, and fuel efficiency (hwy) on the y-axis. Note that there is a negative relationship between engine size and fuel efficiency: cars with larger engines use more fuel (perhaps not surprisingly). Interestingly, the relationship is roughly linear.

Let's walk through the code step-by-step:

To begin drawing a graph, you call the command ggplot(). So if I run the command `ggplot()`:



I get a blank plotting window. So this first command establishes the plotting area for us (which we'll now use to add our data!).

To add data, however, I need to tell R which dataset to use. I do that with the data = mpg syntax. Note that I write this as:

```
ggplot(data=mpg)
```

Because this is telling R that I want to use the mpg data to draw this graph. But which parts of the mpg data do I want to use? I have to tell R. To do that, I run the command after the +:

```
geom_point(mapping = aes(x=displ, y=hwy))
```

That's a mouthful. Let's work it step-by-step. `ggplot()` has a syntax that is confusing at first, but then makes sense as you get more familiar with it (remember: R is language no different than Spanish or Italian).

Start by looking at the `geom_point()` outer part. R has a range of different geometries (or geoms) that produce different types of graphs. Here, we plotted a set of points, a type of graph known as a scatterplot, so we called `geom_point()`. Below, we'll see how the graph would look with various different types of geoms.

The arguments to any geom (again, the type of graph, here a set of points) always include 2 parts: a mapping and an aesthetic. The mapping tells R how your variables are put onto the graph. The `aes()` is the aesthetic, which covers how the graph looks. This can take a variety of different forms, but here, we're just telling R which variables to plot on the x- and y-axes (we'll see how to change other aspects of the graph below). Note that displ is on our x-axis, and hwy is on our y-axis.

So note that what our command does is it tells R to produce a scatterplot of engine size (disp) against fuel efficiency (hwy) using the mpg dataset. Pretty neat!

It's also worth commenting on the way the code is written. We write it over 2 lines as:

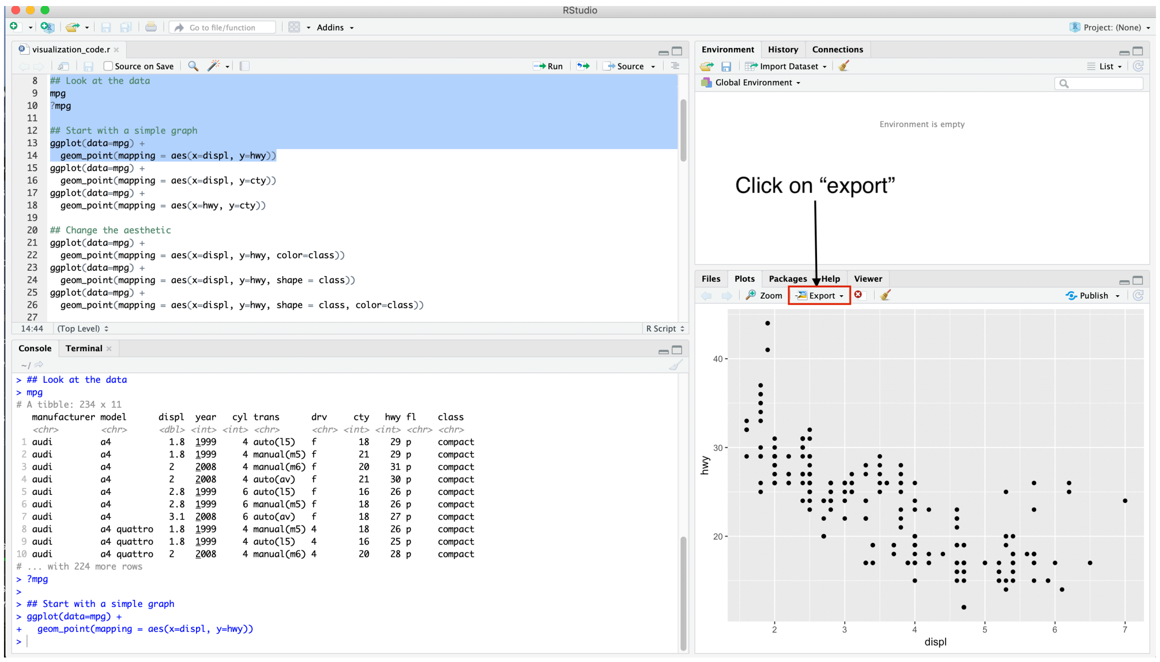
```
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy))
```

One advantage of using RStudio is that it knows that a + is a signal in R that this is all one big command, and so it auto-indents in our screen to tell us that. We write this on 2 lines just so it's a bit easier to read. Note that if you're writing code, the + must come at the end of the line. If you instead write:

```
ggplot(data=mpg)
+ geom_point(mapping = aes(x=displ, y=hwy))
```

You'll get an error. This is because R now reads this as two separate commands (line 1 is one command, line 2 is another), rather than as one joined command. Remember, R is language like French or German, and you have to obey its syntax rules!

Now that you've made this plot, you might want to save it to submit on a homework, or to include in a presentation to a client (for the latter, you'd want to clean it up more, but we'll talk about that in a future lecture). RStudio makes it easy to save plots. In the plot window (the bottom right window in RStudio), there is a button labeled export:



```
## Look at the data
mpg
?mpg

## Start with a simple graph
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy))
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=cty))
ggplot(data=mpg) +
  geom_point(mapping = aes(x=hwy, y=cty))

## Change the aesthetic
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy, color=class))
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy, shape = class))
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy, shape = class, color=class))

```

```
> ## Look at the data
> mpg
# A tibble: 234 x 11
   manufacturer model      displ  year   cyl trans      drv     cty   hwy fl    class  
   <chr>        <chr>      <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>  
 1 audi         a4           1.8  1999     4 auto(l5)   f        18    29 p     compact
 2 audi         a4           1.8  1999     4 manual(m5) f        21    29 p     compact
 3 audi         a4           2    2008     4 manual(m6) f        20    31 p     compact
 4 audi         a4           2    2008     4 auto(av)   f        21    30 p     compact
 5 audi         a4           2.8  1999     6 auto(l5)   f        16    26 p     compact
 6 audi         a4           2.8  1999     6 manual(m5) f        18    26 p     compact
 7 audi         a4           3.1  2008     6 auto(av)   f        18    27 p     compact
 8 audi         a4 quattro   1.8  1999     4 manual(m5) 4        18    26 p     compact
 9 audi         a4 quattro   1.8  1999     4 auto(l5)   4        16    25 p     compact
10 audi         a4 quattro   2    2008     4 manual(m6) 4        20    28 p     compact
# ... with 224 more rows
> ?mpg
>
> ## Start with a simple graph
> ggplot(data=mpg) +
+   geom_point(mapping = aes(x=displ, y=hwy))
> 
```

If you click on export, it gives you the option to save the graph as a PDF, or to copy it to the clipboard so that you can paste it into another document (such as a Word or PowerPoint file). This makes it very easy to take your great work form R and show it to the world!

TEST YOURSELF: Highway fuel efficiency is only one measure of fuel efficiency, we could also use the measure taken during city driving (cty in our dataset). How would I produce a scatterplot of engine size vs. city fuel efficiency?

ANSWER:
```
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=cty))
```

Note that this calls a new variable for the y-axis (though the plot looks very similar, as highway and city fuel efficiency are closely related).

So now we can state a general template for making graphs in R:

```
ggplot(data = <DATA>) +
 <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

Where <DATA> is our dataset, <GEOM_FUNCTION> is the type of graph we want R to make (point, line, etc.), and <MAPPINGS> tells R which variables go on the x- and y-axes. So to draw a graph with ggplot, we need to tell R three things:

1. The dataset with the variables we want to plot
2. The geometry (geom) that we want to use to plot the data (points, line, boxplot, etc.)
3. The mapping, which tells R which variables to use

That's the core of a ggplot graph. But of course, we can do much more with it! Let's see that in action by examining how we can change an aesthetic.

Changing an Aesthetic

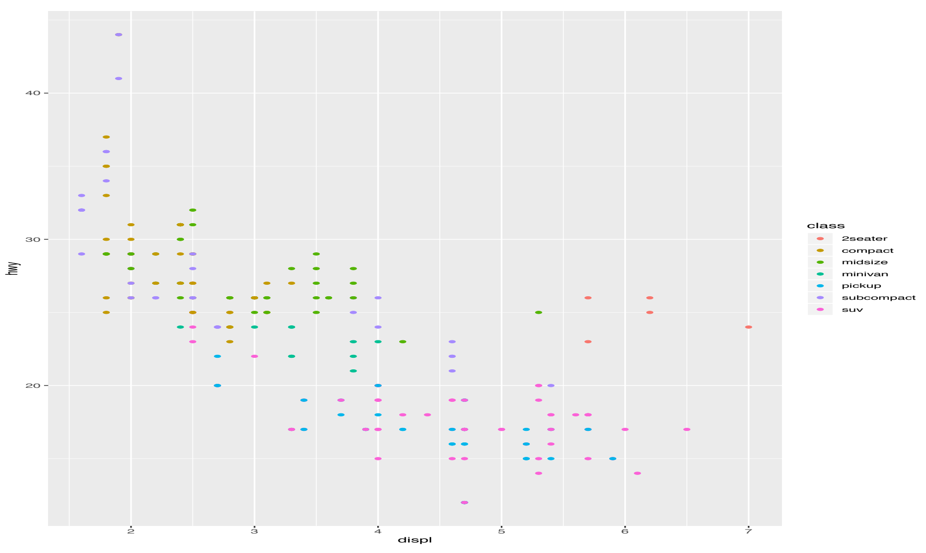
So far, we've used ggplot() to give us a scatterplot of our data. But one of the neat things we can do with R is to dig into our data. For example, we might expect that different classes of cars: subcompacts vs. SUVs, say, might be in quite different parts of the graph. We have the class variable, which tells us each auto's type. So we can tell R to plot each type of car with a different color/symbol so we can tell them apart.

To do that, we have to add another aesthetic to our plot. An aesthetic is a visual property of the graph: things like the size, shape, or color of your plot. So instead of plotting small black circles (as we had above, the default), we could plot large red triangles, for example.

If we want to make each class of car a different size, shape, or color, we change its aesthetic. Let's try having each class of color be represented by a different color. We can do that with the following code:

```
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy, color=class))
```

Which gives us:



Note how my code changed here: I added another argument to aes, which controls the aesthetics of the plot (hence the name). Also note the parentheses: because color is a type of aesthetic, I include it inside the parentheses for aes(). Because I wanted the color of the points to correspond to the class of each car, I used the class variable to set the color.

So what do we learn substantively? We see a few really interesting patterns. First, notice that for SUVs and pickups, the relationship is somewhat weaker. Because these cars are bigger and heavier, they really don't get more than 20 miles to the gallon on the highway (at least if they've got more than 3 cylinders). Second, note the small cluster of coral dots, which correspond to 2-seater cars. Those cars are sports cars with big engines, but they're small and light, so they still get pretty good gas mileage. In the graph above, they're something of an outlier, so if we were going to do a more sophisticated analysis, we might consider them separately.

What different aesthetics can you control? There are 4 basics types you're likely to see:

Size: controls the size of each point
Alpha: controls the transparency (opacity) of each point
Shape: controls the shape of each point (i.e., square, triangle, cross, etc.)
Color: controls the color of each point, as above

Which one should you use? It really depends on the problem at hand. Personally, I think colors work well for monitors, but if you're going to print something in black-and-white, changing the shape will work better. The size feature can be nice for emphasizing differences across units, say by population. The alpha feature is useful when you have a dense graph, as it can make it easier to read. Ultimately, you just have to use a bit of trial-and-error and your best judgment!

TEST YOURSELF: How would you produce the scatterplot of highway gas mileage (hwy) versus engine size (displ), but using different symbols instead of different colors?

ANSWER:
```
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy, shape=class))
```

Note that by default, R only allows 6 symbols, so the 7th level of class (SUV) is not plotted. You could also experiment by setting the same variable to two different aesthetics (what happens then?).

Adding Facets to a Graph

Above, we showed how to use color to differentiate our plot. This is really helpful, and gave us a lot of new information, but the graph is still pretty hard to read. Let's graph each class of car separately.

To do that, we'll need to introduce something called a facet, which is just R-speak for making subplots, each with a different part of the data.

Facets are really helpful when we want to visualize the relationship between two variables across different levels of a third variable. For example, suppose we had a dataset that recorded GDP and mortality rate for each country over time.[1] I might suspect that the relationship between GDP and mortality might look very different in Western Europe than it does in Africa, for example (since there are many differences between those continents). To help make this sort of comparison easier, we could make a facet where we plotted the relationship between GDP and mortality separately by continent. We would then have 6 graphs (one per continent) showing how GDP and mortality relate to one another on each continent.

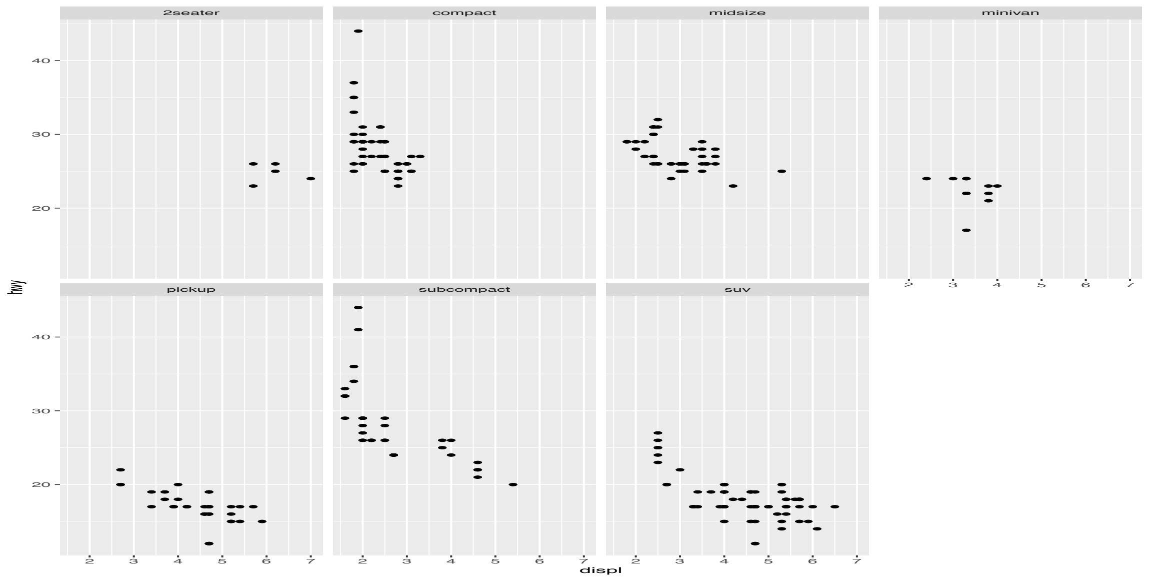
We also saw an example of a faceted graph in our introductory lecture when we looked at the trends in opioid deaths over time by Census region. That also helped us see the patterns in our data more clearly.

Let's use a facet here to look at our data within each class of car:

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```

Let's dig into this code. The first two lines are the same as before, but now the third line is different. The `facet_wrap()` option allows you to split your plot by 1 variable (and note that this variable should be discrete, that is, have a limited set of values). The ~ (tilde) symbol is just part of R's syntax, and the `nrow = 2` tells R to plot this over 2 rows (rather than 1, since that gets a bit crowded). This now gives us:

[1] For an example of such a dataset, you can install the `gapminder` library in R using the `install.packages` command we discussed in our last lecture.



Note that this is actually quite helpful! What we can see is that relationship between engine size and fuel efficiency differs pretty widely by the type of car. For example, for 2-seater sports cars, there's really no relationship at all. For compacts and subcompacts, there's a relationship, but there's some non-linearity for the smallest engines. In short, the car's class is an important variable for understanding the relationship between engine size and fuel efficiency.

TEST YOURSELF: How would you produce a graph of highway mileage vs. engine size where you plotted 1998 and 2008 model cars in two different facets?

ANSWER:
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ year, nrow = 2)
```

Interestingly, notice that the pattern is pretty similar in both years (though the most fuel efficient cars are from 1998, not 2008!).

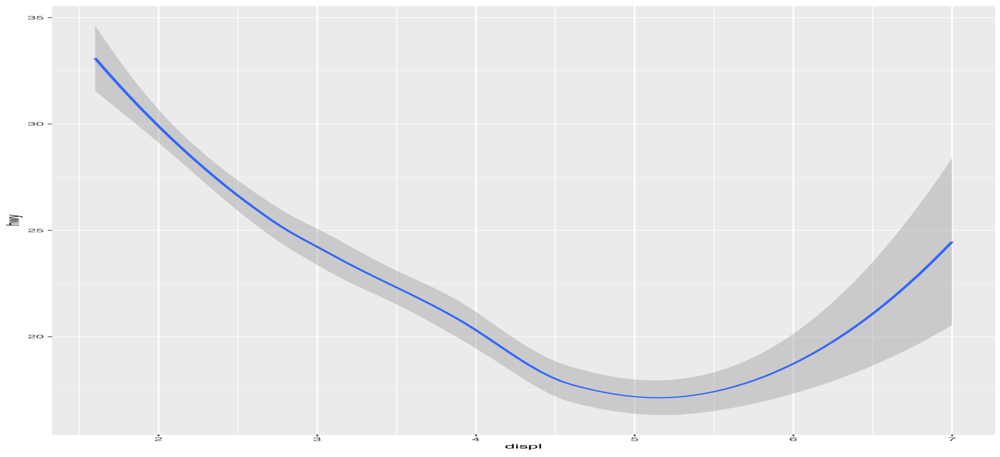
Most of the time, if you're using a plot with facets, you use one variable, and hence `facet_wrap()`. But you can create a plot where you make a facet by 2 different variables. This is done using `facet_grid()` rather than `facet_wrap()`, see the textbook section 3.5 for an example.

Beyond Scatterplots: Different Types of Graphs

So far, we've seen how we can change the aesthetics and the facet of a graph to convey more information. But we've only had scatterplots, that is, showing a collection of x- and y-points on a graph. This is a very useful type of graph, but it's not the only type of graph. Often, we can learn a lot just by changing the type of graph, so let's see what we find when we change from a scatterplot to a line plot. To do that, we change the geom() in our command:

```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

Which gives us:

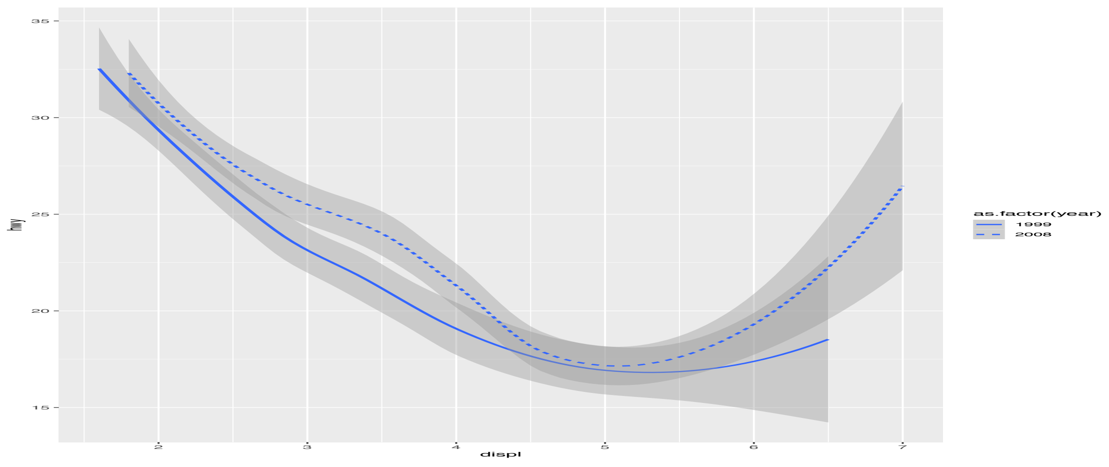


Notice that now instead of `geom_point()`, we have `geom_smooth()`, which tells R to produce a smooth line (rather than set of points). There are more than 30 different geoms you can use, and we'll go over a few more below. But to see all of them, look at the help file within R (`?ggplot`), or dig around on Google.

Substantively, what do we see? We see that as engine size increases, fuel efficiency goes down until around a 5-liter engine, and then it goes up somewhat. This uptick at the end is due to the 2-seater sports cars in our data, which as we noted before, are somewhat different than the rest of our data, so we might want to separate them out in future analyses. But the advantage of the line is that we can see the pattern quite clearly.

Note that we also change the aesthetics of this plot. For example, we could draw separate lines for each of the 2 years in our data:

```
 ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, linetype = 
as.factor(year))
```

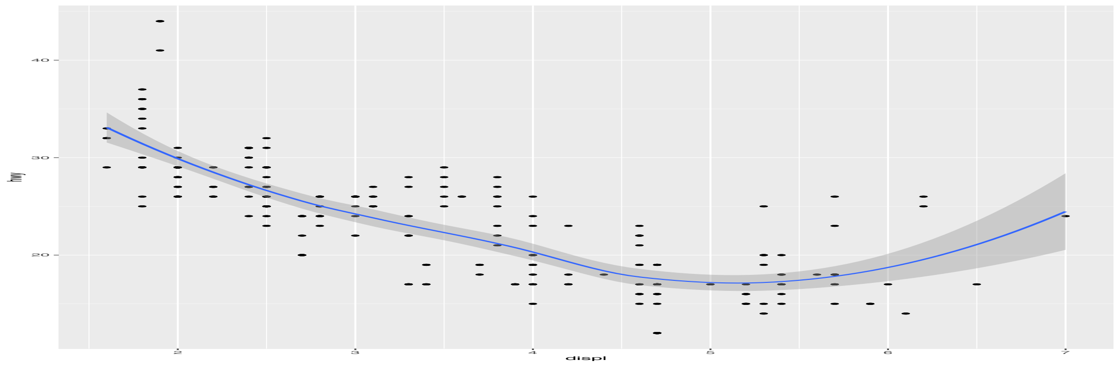


So here linetype draws different types of lines for each model year (a solid line for 1999, and a dashed line for 2008). Because year is a continuous variable, and linetype requires a discrete variable, we have to use the as.factor() code, but don't worry too much about this for now.

Note that we can overlay multiple geoms on top of one another. For example, we could have a graph with both the points and the lines overlaid on top. This can be a nice way of combining the strengths of the separate plots seen above:

```
ggplot(data = mpg) +
  geom_point(mapping=aes(x=displ, y=hwy)) +
  geom_smooth(mapping=aes(x=displ, y=hwy))
```

Note that this is just a combination of our code above, which produces:



This nicely highlights the role of the one car with a 7-liter engine in pulling the line up toward it. This is a good point to note that we'll formalize in future courses: such points are known as outliers, and can exert a large effect on our results.

Notice that in my call above, I include separate calls to `geom_point()` and `geom_smooth()`, both of which contain the same calls to `aes()`. This is fine, but there's a slightly more efficient way to write this command:

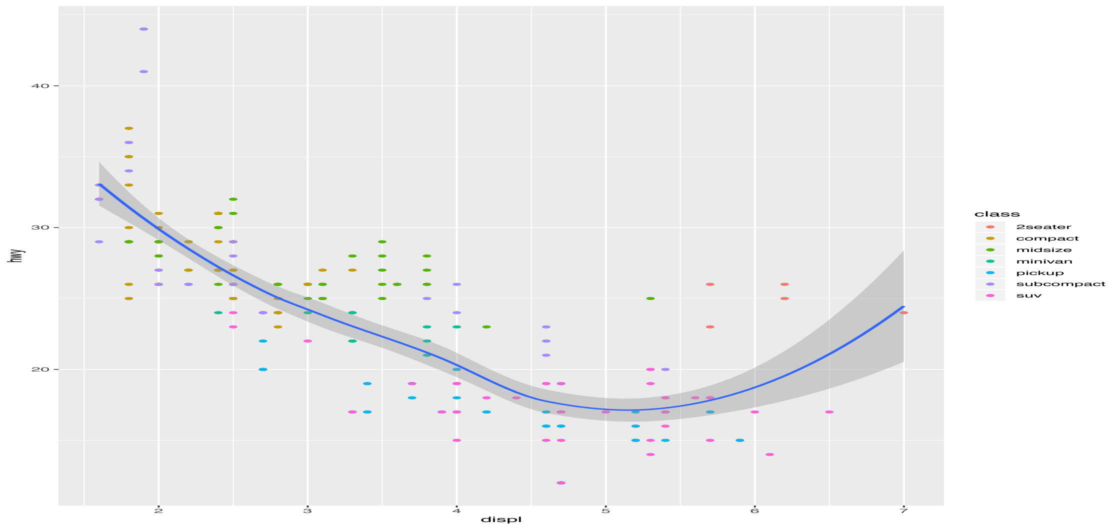
```
ggplot(data = mpg, mapping=aes(x=displ, y=hwy)) +
  geom_smooth() +
  geom_point()
```

Note that this produces the same graph as the above.

So here, I put the `mapping` argument within `ggplot()`, which then passed it to `geom_smooth()` and `geom_point()`. Why would I prefer this way? It is less likely to produce errors, because I'm ensuring that both geoms get the same mapping.

It is also possible, and in some cases desirable, to pass different mappings to different geoms, however. Let's see how that can work with an example:

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth()
```



So notice that while both geoms() plot the relationship between engine size and fuel efficiency, the color aesthetic only applies to the scatterplot (so the points are colored based on the class of each observation, but the line is not).

TEST YOURSELF: How would you reproduce the chart above, except with different symbols for car class, rather than different colors?

ANSWER:

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(shape = class)) +
  geom_smooth()
```

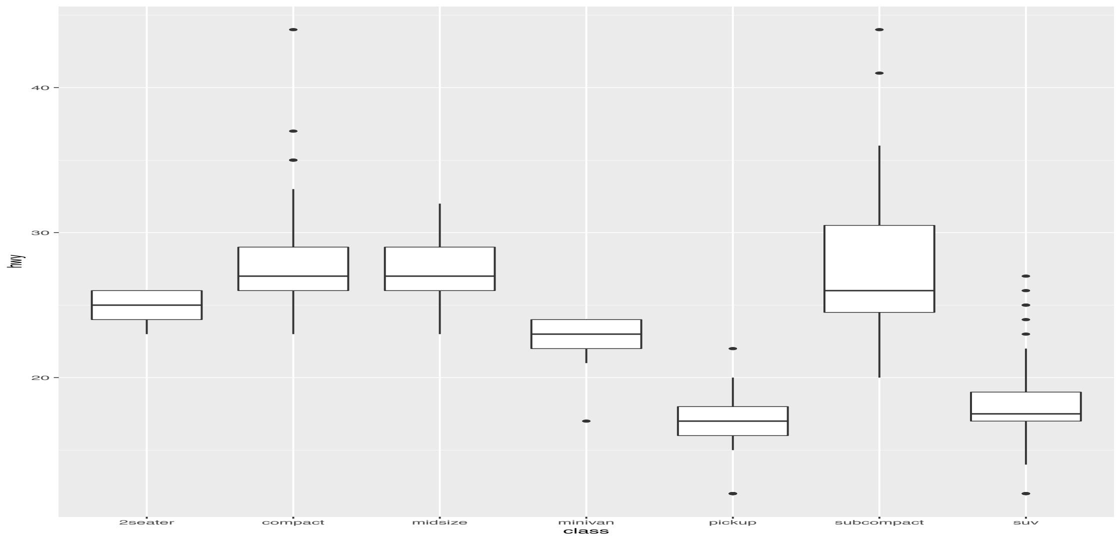
There’s one more very useful type of two-variable plot that we should cover now: a boxplot. A boxplot provides another way of seeing how two variables relate to one another (you’ll see why it’s called a boxplot below).

Whether a scatterplot, a line plot, or a boxplot make the most sense depends on your data. This is in part a function of whether your x and y variables are discrete or continuous. Remember, discrete variables are those that take on a limited number of values. In our dataset above, class is a discrete variable, since it takes on a limited set of values (i.e., 2-seater, subcompact, etc.). In contrast, fuel efficiency is a continuous variable, since it takes on a potentially infinite range of values: with enough precision, we could know that car x gets 19.88893 miles per gallon on the highway, for example. If you have a discrete x and a continuous y, a boxplot is often quite useful plot, as it allows you to see the distribution of your data. But if you have two continuous variables, then a scatterplot or line plot can be more useful. But at the end of the day, trial-and-error is often the best way: try several different graphs, and see which one is the most effective.

Let’s look at the box plot of fuel efficiency for each class of auto in our dataset:

```
ggplot(data=mpg) +
  geom_boxplot(mapping=aes(x=class,y=hwy))
```

Which gives us:

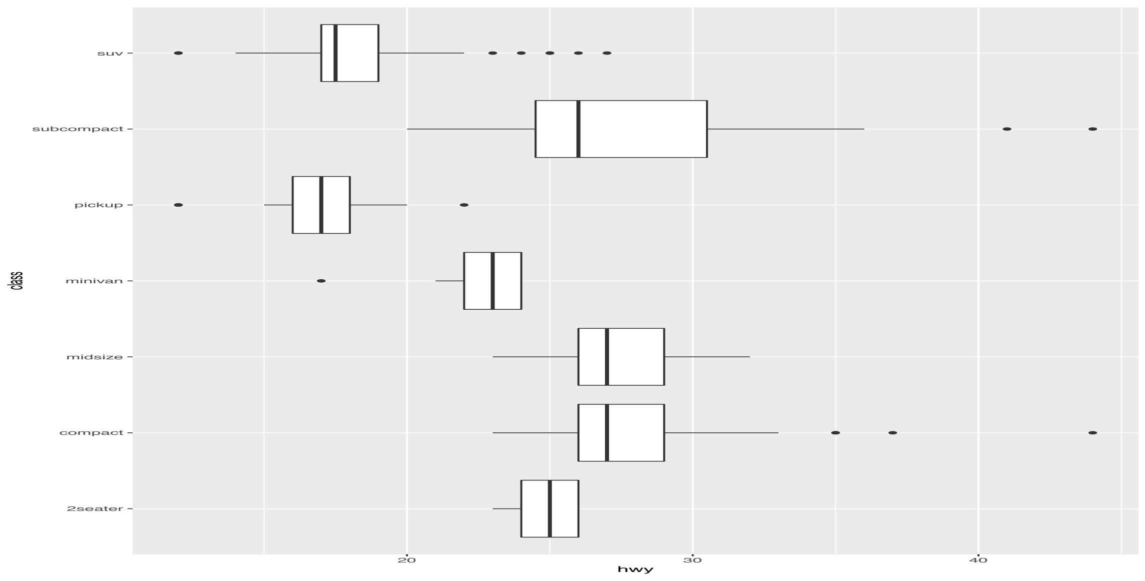


This plot type shows the distribution of fuel efficiency within each class of car (you should understand how I can tell that from the aes() arguments above). It’s called a boxplot because it produces a “box” of the data (the heavy line is the median, the top and bottom of the box are the 25th and 75th percentiles, but don’t worry about that now). The points outside the box are more extreme observations, which as we saw above are known as outliers.

What do we notice? Some classes of cars have very little within-class variation. For example, 2-seaters and minivans all basically get very similar gas mileage. But others have quite a bit more variation, particularly subcompact cars.

Note that we can also re-orient a boxplot for ease of reading:

```
ggplot(data=mpg) +
  geom_boxplot(mapping=aes(x=class,y=hwy)) + coord_flip()
```



The `coord_flip()` option flips the coordinates (x and y) around, so that fuel economy is on the x-axis and the car class is on the y-axis. In some cases, this can make a boxplot easier to read (though this is, in part, a personal preference).

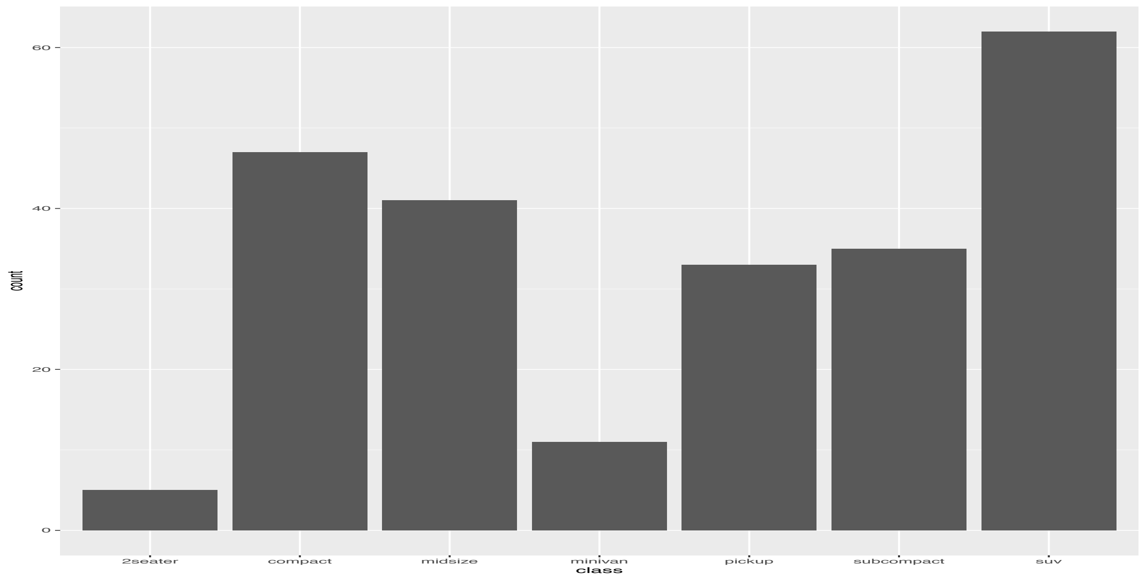
There is obviously quite a bit more one can do with a boxplot. To see a whole set of additional options, please visit: https://owi.usgs.gov/blog/boxplots/.

Single Variable Graphs (Bar Charts)

We can also use R to produce graphs for a single variable as well. For example, sometimes we might just want to see a bar chart showing us how many cars are in each class. We can do that with the following code:

```
ggplot(data=mpg) +
  geom_bar(mapping=aes(x=class))
```

Which produces:

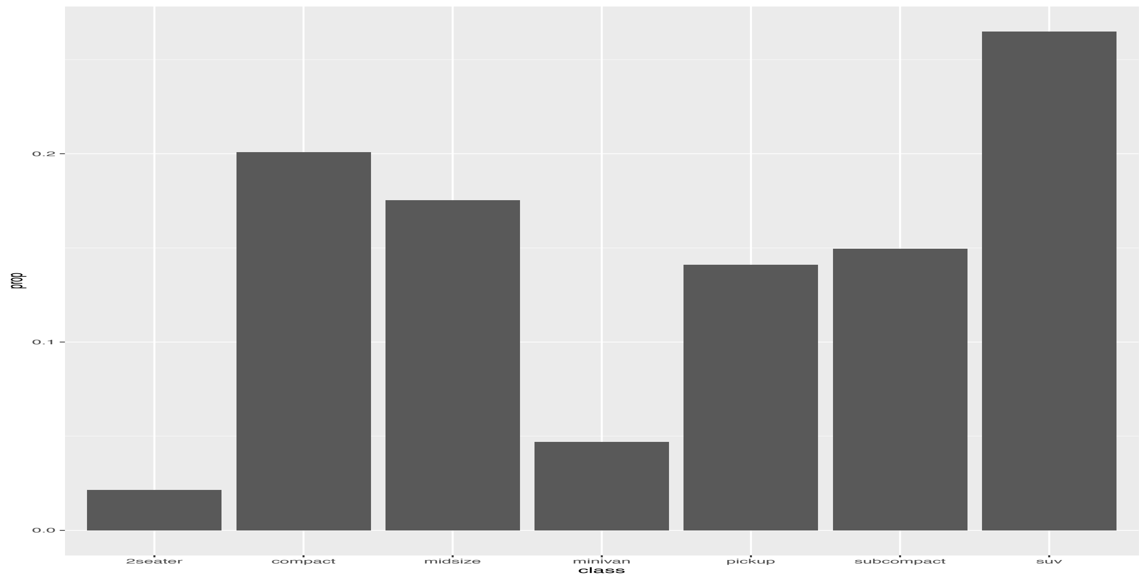


So this shows us that there are the most SUVs in our dataset (more than 60!), but there are only a small number of 2-seater sports cars (5 in total) and 11 minivans. Note that because we are only displaying one variable here, we only include a call to x in our aesthetic (rather than both x and y).

This reflects a somewhat subtle property of how R makes graphs. When you tell R you want a boxplot, it says "aha, you want to know how many observations there are for each level of x." So above, it counted the number of cars within each class (since we set x=class above). To do this, R creates an internal statistic that records these values, and then uses that statistic to complete the graph. R did the same thing above when we called geom_smooth(), which uses an algorithm to draw the smoothed line in the graph (don't worry about the details of this at the moment).

95% of the time, you can just let R work out the right statistic for you behind the scenes. But occasionally, it is useful to know how to over-ride it. For example, sometimes you don't want to know the number (count) within each group, you'd like to know the proportion of all cars in our dataset. For example, what proportion of the cars in our dataset are SUVs? We can have the bar chart tell us this as well:

```
ggplot(data = mpg) +
  geom_bar(mapping = aes(x = class, y = ..prop.., group = 1))
```

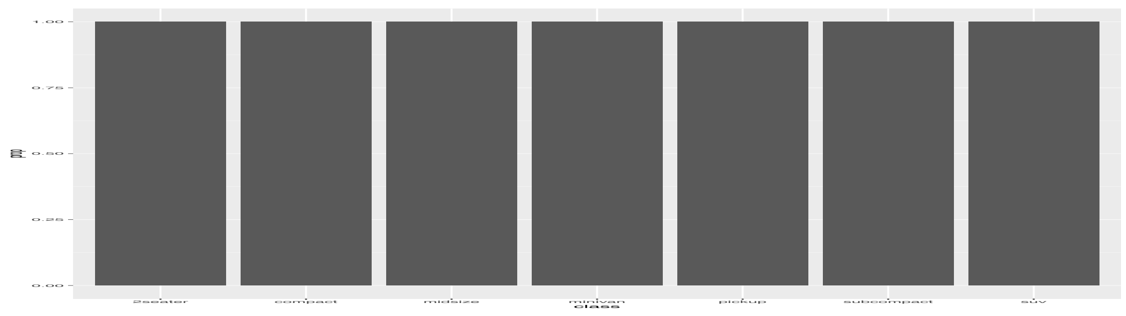


So now we can see that SUVs make up a bit over 25% of the sample, and 20% are compacts, and so forth. Note that the y-axis is the proportion, rather than the percentage, but it's easy enough to just multiply by 100 to get the proportion for now (0.10 is 10%). In a later lecture, we'll see how to fix up these sorts of graphical issues.

Note that now we're giving an explicit call to y here within our aesthetics, and we're calling a different statistic here. The `..prop..` value here tells R to calculate the proportion of total cases within each class. Note the two dots before and after "prop", those are part of the variable name. The dots reference the fact that it is an internal function in R, but don't worry about that. For example, there are 47 total compact cars, and 243 total cars, implying that 47/243 = 19.3% of the sample is a compact car. This is just an easier way of referring to it. So here, instead of having the count be our statistic, our statistic is the proportion.

Note that there's another change we made here: we have the `group = 1` call to the aesthetic as well. Why do we need it? Let's see what happens if we drop it:

```
ggplot(data = mpg) +
  geom_bar(mapping = aes(x = class, y = ..prop..))
```

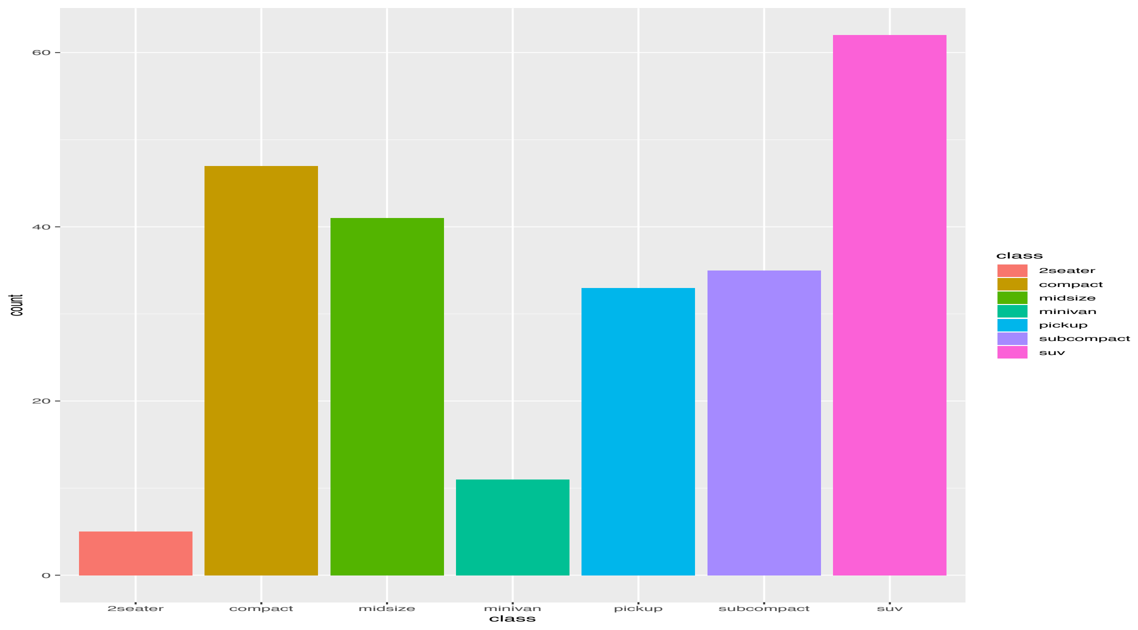


What just happened? Normally, within a bar chart, each category of x is grouped with itself. So, for example, I group 2-seaters together, count the total number of observations, and report that value. I do the same for compact cars, midsize cars, and so forth, producing our original bar chart, which reports the count of each type of car. But if I want to know what proportion of all cars are 2-seaters, I cannot just look within two-seater cars, I need to look at <u>all</u> cars. To get all cars, I need to tell R that I want it to treat all cars as one group, hence the `group = 1` option. When we drop the group = 1 part of the command, what R is telling us is that 100% of 2-seater cars are 2-seaters cars, 100% of compact cars are compact cars, etc. Remember, the default for a bar chart is to group each level separately, because it thinks you just want the counts!

There's a good lesson here: often, making what seems like a trivial change will produce a big change in R's behavior. When this happens, Google is a great resource, as someone else has probably wondered the same question. For example, in the last case, I might try Googling something like "R ggplot bar chart what does group = 1 do", which returns several helpful results (including some pages from StackOverflow).

We can also add color to our bar charts using the fill() option:

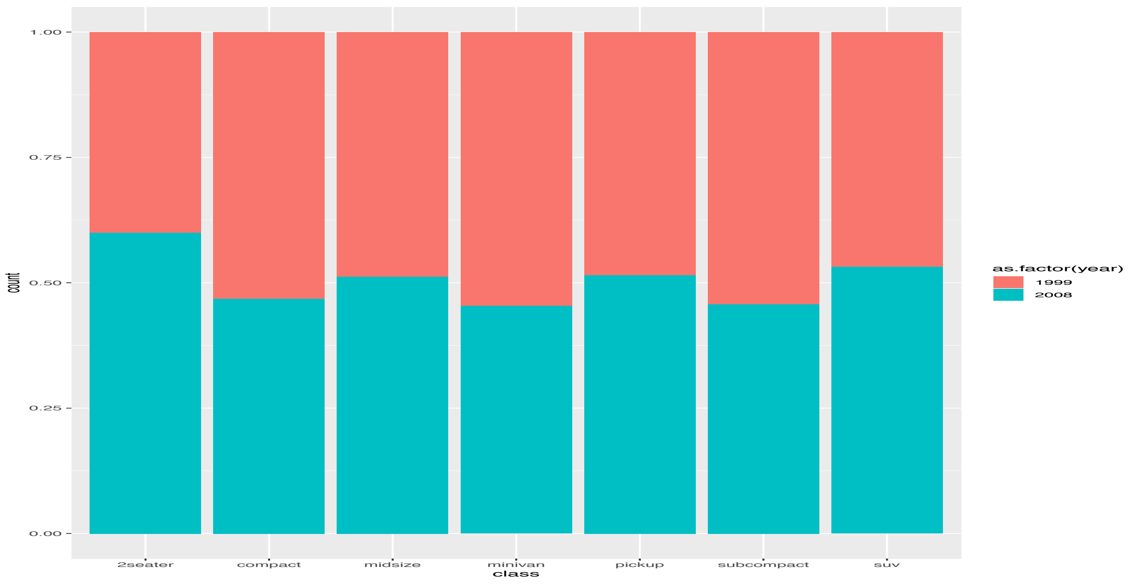
```
ggplot(data=mpg) +
  geom_bar(mapping=aes(x=class, fill=class))
```



That's nice, but not really all that useful to be frank. But color can be a big help when we add in a second variable. Let's see how the number of cars in each class differed by year (1999 vs. 2008). To do that, we edit our code slightly:

```
ggplot(data=mpg) +
  geom_bar(mapping=aes(x=class, fill=as.factor(year)),
position = "fill")
```

Note what this code does: it takes each class of car, and fills it in with color depending on the year (again, don't worry about the as.factor() around year). Note that we also have the additional call to position = fill, which we'll explain below.

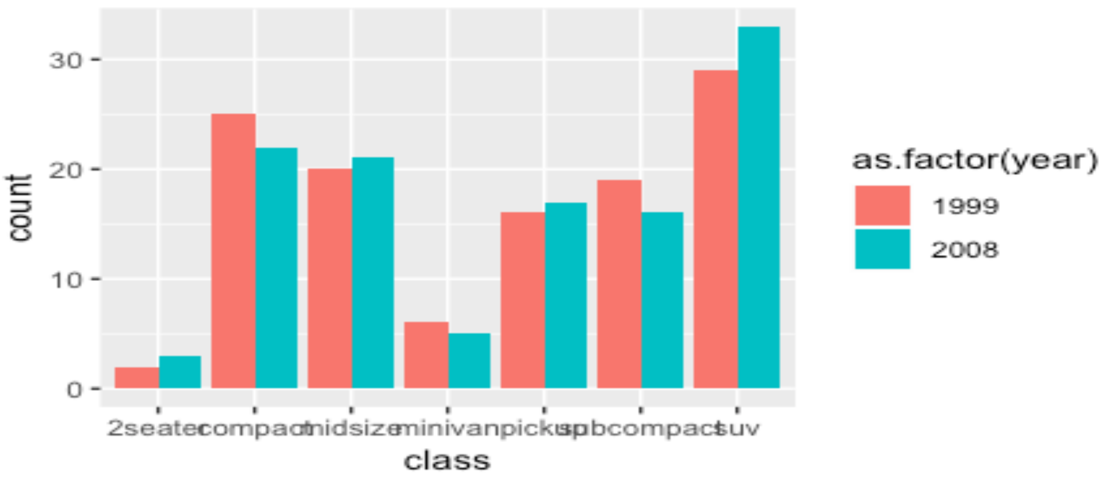


So here, we can see that there were slightly more 2-seater cars in 2008, and slightly more compacts and subcompacts in 1999. But really, the distribution of cars looks quite similar in both years.

But there is another way of presenting this graph: instead of presenting 1999 and 2008 on top of each other, we can present them side-by-side. To do that, we use the following bit of code:

```
ggplot(data=mpg) +
  geom_bar(mapping=aes(x=class, fill=as.factor(year)),
position = "dodge")
```

Which gives us:



Notice that the only difference between those two bits of code is that one uses fill and the other dodge. So fill makes it easier to compare differences across groups (i.e., how does the model year breakdown differ across classes?), and dodge makes it easier to compare within groups (i.e., how does the number of 2-seaters differ between those 2 years?). Especially with multiple categories, dodge can be a powerful tool for learning about your data. Patterns uncovered here can suggest places to dig deeper into the data.

So now let's pull back and think a bit more broadly about everything we learned so far. We've learned not just how to make scatterplots, boxplots, and bar plots, but we've learned a more general grammar for making graphs that you can apply to any type of graph. Pulling back, we can see that the general pattern of R graphics:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
     mapping = aes(<MAPPINGS>))+
  <FACET_FUNCTION>
```

So notice that there are several different aspects you need to supply, each written <LIKE THIS>:

- Data: the dataset you're using
- Geom_Function: The type of graph (the geometry or geom), such as point, line, smoothed line, boxplot, etc.
- Mapping: the aesthetics of the graph (size, color, shape, x/y values, etc.)
- Facet_Function: If you want to display separate plots by a variable

Note that you need not supply a facet for every graph, as that's not useful for every plot. But the first three factors are very important, and allow you to build a tremendous variety of different graphs.