R For Econometrics: A Modest Handbook

Ryan Safner (safner@hood.edu)

Last Updated: 2018-12-15

Contents

Chapter 1

Prerequisites

Open Source: The raw (`.Rmd`) code used to produce this guide, along with the guide itself, are available on GitHub, and are updated regularly. GitHub does not automatically render HTML, so download the HTML file and open it, or view it where I host it on my website.

Note to Students: This is a work in progress, check the date at the top for when this was last updated. This compiles all of my instructions, advice, and examples from econometrics class lectures regarding `R`. It also contains some advanced material that I did not or will not cover in class, but will be useful to know for future data analysis and understanding or diagnosing problems. **Note to Everyone Else**: This guide is oriented primarily for my Econometrics class at Hood College, but should be of wider use to anyone interested in learning `R` for data analysis. Lecture slides, handouts, and guides (both PDFs and source code in R Markdown) are openly available on GitHub.

See also my companion guide to using R Markdown to more effectively manage your entire workflow (text, data analysis, tables, graphs, and citations!) in a single plain text file and make your work reproducible and shareable, hosted on my website, with source available on GitHub

This guide is meant to be a somewhat comprehensive resource such that you can come back to different sections when you encounter a specific limitation or problem in your own work. I *do not* recommend reading through this guide from start to finish, or in order.

1.1 How to Read this Guide

As an econometrics student, the core of your data analysis life will be working with `data.frame`s (think "spreadsheets", where each row is an observation and each column is a variable). You will: - import data into a `data.frame` - transform ("wrangle") data into more useful variables or `data.frames` - plot data from `data.frame`s (in histograms, scatterplots, etc.) - run regressions using data from `data.frame`s

This guide attempts to introduce you to `R` from the ground up, which means it starts with simpler types of objects than `data.frames` (namely, `vectors`). I would not necessarily recommend reading from beginning to end. The first two sections describe a lot about `R` as a language and discuss different types of `R` objects, data types, and commands. Starting at the very beginning, reading them will seem overwhelming. They will become more useful to return to for reference later, once you have some practice under your belt.

Chapter 2

Basics

2.1 Operating R Studio

- There are a few ways you can use R Studio:

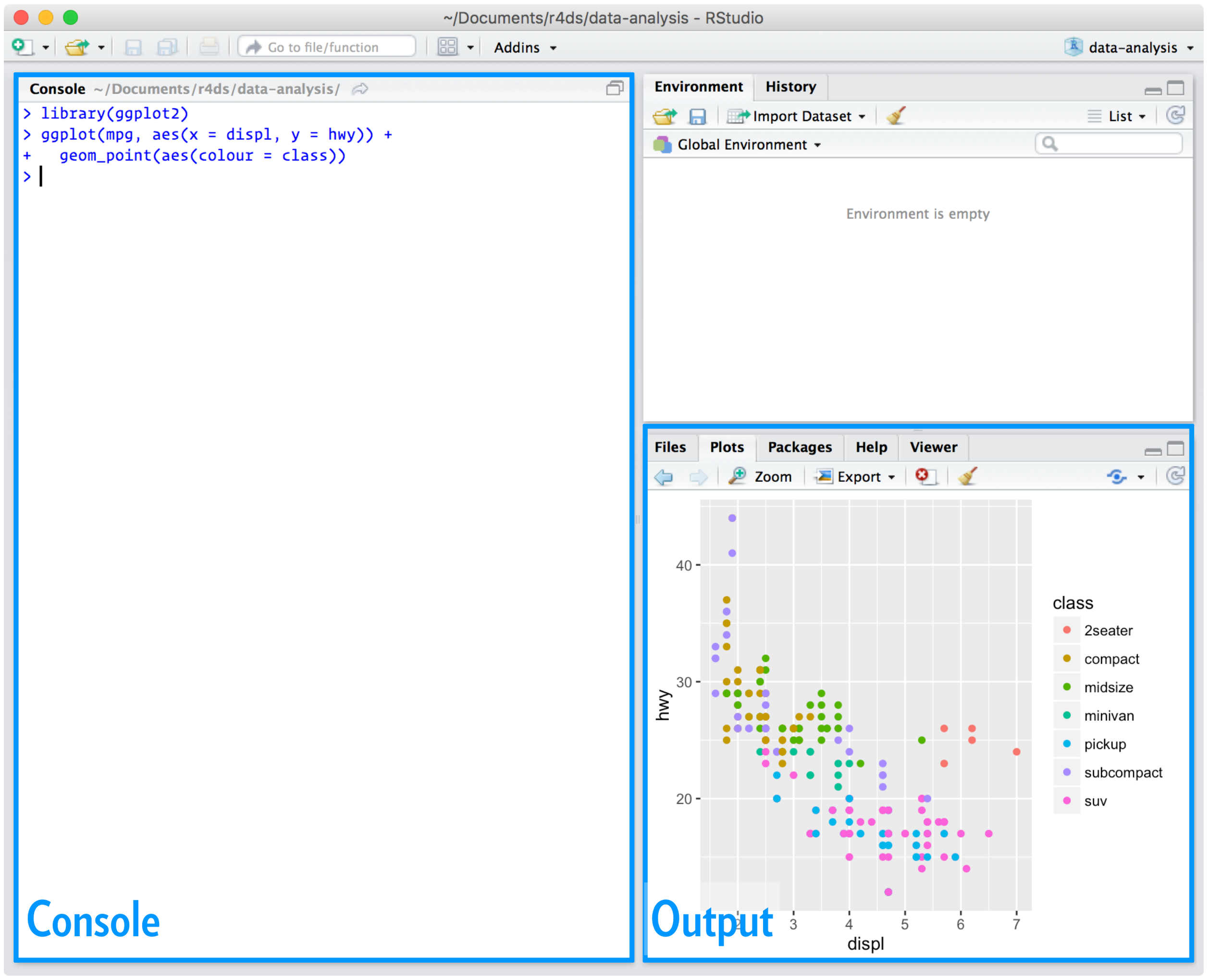
1. **Command line/Console**: writing each command by itself and copying down the result as needed
    - Great for testing individual commands to see what happens
    - Not reproducible! Not saved! NOT RECOMMENDED!
2. **.R files**: A sequence of commands (and hopefully comments) saved as a script, the entire script is run all at once
    - Can test individual commands in command line and then put good commands in *.R* file
    - Equivalent to a *.do* file for Stata
    - Reproducible, saved, commented
3. **R Markdown (*.Rmd*) files**: A plain text document written in *R Markdown* language
    - Allows for individual chunks of *R* code to be run individually (great for testing one command instead of all at once)
    - Reproducible, saved, commented as if a normal document
    - Can write an entire document (text, equations, R commands, figures, tables, etc) with one file!
    - Can export to html, MS Word, Beamer, etc!
    - Markdown is a language that is intuitive, simple, human- and machine-readable

2.1.1 Keyboard Shortcuts

- `Ctrl+2`: move cursor to console
- `Ctrl` (`Cmd` on Mac)`+Enter`: run current line (from editor) in console
- `Ctrl` (`Cmd` on Mac)`+Enter`
- `Uparrow`: retrieve recent commands in console
- `Ctrl` (`Cmd` on Mac)`+Uparrow`: search previous commands
- `Option -`: insert assignment operator (`<-`)
- `Ctrl` (`Cmd` on Mac)+`Shift+M`: insert pipe operator (`%>%`)

2.2 Working Directory (`wd`)

- *R* assumes a default (often inconvenient) working directory on your computer
    - this is where it thinks it will **load** any files you want to load and **save** anything you want to save by default



```
> library(ggplot2)
> ggplot(mpg, aes(x = displ, y = hwy)) +
+   geom_point(aes(colour = class))
> 
```

Figure 2.1: Rstudio Windows

Package name	Use
`ggplot2`	Rendering beautiful graphics (scatterplots, histograms, etc)
`stargazer`	Rendering professioanl looking regression output tables
`dplyr`	Manipulating data much more intuitively
`sandwich`	More tools for regression, particularly robust SE's
`tidyverse`	An epic *meta*package of `ggplot2, dplyr` and other popular packages

- Find out where *R* currently thinks this is with **`getwd()`**
  - this is often Operating System specific, e.g.:
    - Mac: `/Users/yourusername/`
    - Windows: `C:/Users/yourusername/Documents/`
  - you can move everything you want to load into this folder on your computer (and save everything there too), but this may be inconvenient
- *Change* the working directory to wherever you plan on keeping your related data and documents with **`setwd("/path/to/folder")`**
  - you can move to a new `wd` *relative* to the current working directory:
    - move down a folder by typing the folder name with a `/` after i
      - e.g. to move from `/Ryansafner/Documents/` to `/Ryansafner/Documents/Econometrics/`
    - move up one folder in a hierarchy with `..`
    - e.g. to move from `/Ryansafner/Documents/` to `/Ryansafner/Downloads`, use `setwd("../Downloads/")` to move up from the `Documents` folder to `Ryansafner` folder, then down to Downloads

2.3 Packages

- **Packages** are extensions of base `R` designed by users
  - Remember, `R` is open source, packages are usually published first on Github
  - Official packages distributed and documented through CRAN
- To use a (previously-installed) package (note the “”), use the `library()` command:

  `library("packagename")`
- If you do not have a package, they are easy to install with (note the plural “s”)

  `install.packages("packagename")`
- To install or load multiple packages at once, we can use the `c()` function to select multiple packages (**see below**)

```
library(c("gapminder","ggplot2","dplyr"))
```

2.4 Useful Packages

- There are several packages we will use often (and are featured later in this guide)
  - Packages are often very well-documented with explanations and examples
  - Google each package for more information

2.5 Calculations

- `R` can be used as a calculator
  - Basic operations `+`, `-`, `*`, `/`

  - More advanced math operators like exponents, logarithms, trigonometric functions, etc

```
2+2
```

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

```
6^2 # 6 to the second power (i.e. squared)
```

```
## [1] 36
```

```
sqrt(100/4) # square root
```

```
## [1] 5
```

```
log(5) # logarithm
```

```
## [1] 1.609438
```

```
sin(2*pi) # sin
```

```
## [1] -2.449294e-16
```

```
factorial(5) # factorial (e.g. 5!)
```

```
## [1] 120
```

```
choose(2,6) # binomial choose function
```

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

```
# order of operations matters
3*3+4
```

```
## [1] 13
```

```
3*(3+4)
```

```
## [1] 21
```

- **Note on Notation**: `R` often reports very large (or very small) numbers in scientific notation with `e`
  - For positive `e`: the number of zeros (or digits after the decimal point) to the right of a number
    - e.g. $1.25e6 = 1.25 \times 10^6 = 1,250,000$
  - For negative `e`: one less than the number of zeros (or digits after the decimal point) to the left of a number
    - e.g. $1.25e - 6 = 1.25 \times 10^{-6} = 0.00000125$

2.6 Hints for Writing Code

2.6.1 Naming Objects

- Object names cannot start with a digit or contain a space or comma
- FOR THE LOVE OF GOD AVOID SPACES IN GENERAL
  - You've seen webpages intended to be called `"my webpage in html"` turned into `http://my%20webpage%20in%20ht`
  - Consider both your `R` objects and your files and folder names on your computer...(`/School/ECON_480_Econometrics`
- It will be wise to adopt some consistent standard for demarcating names:

```
i.use.snake.case
otherPeopleUseCamelCase
some_people_use_underscores
```

```
And_aFew.People_RENOUNCEconvention
```

2.6.2 Commenting

- Always comment your commands! Describe what you are doing so someone else (or you, 5 years later) can understand what is happening and why!
  - Use the hashtag `#` to start a comment (`R` ignores everything on that line after the hashtag)
  - Can be made its own line or at the end of lines
  - e.g.

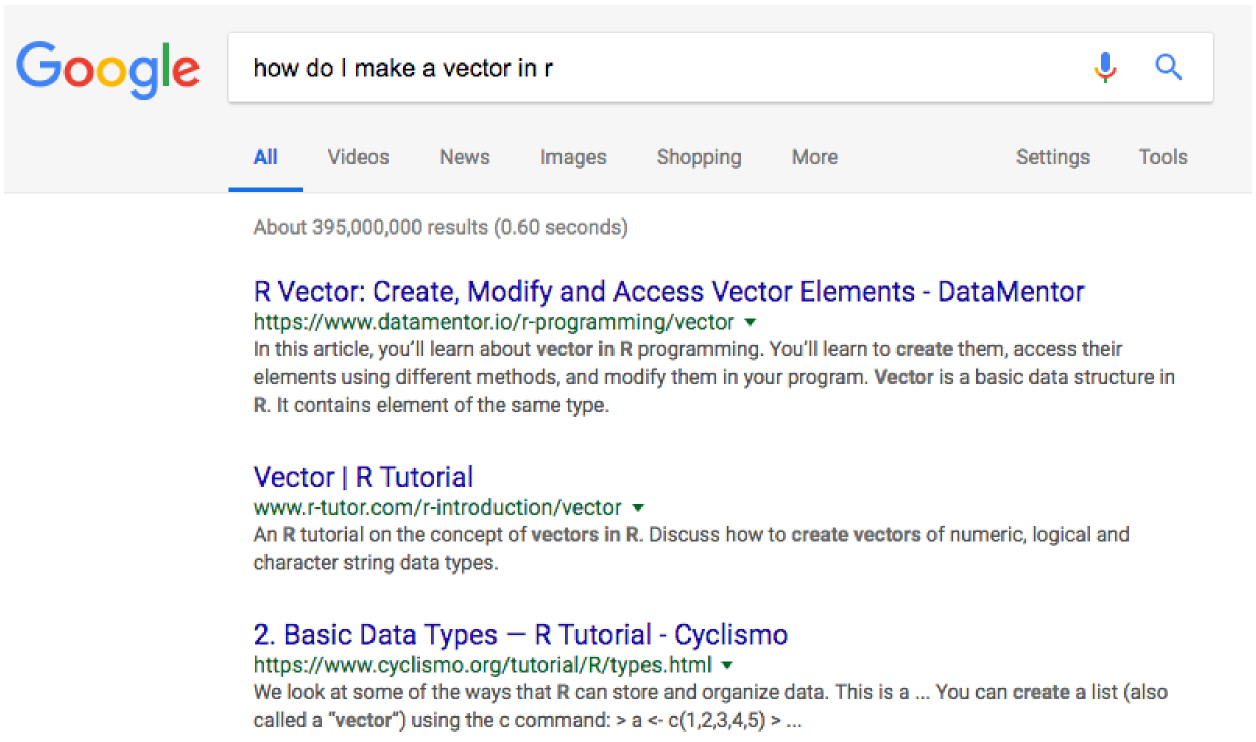
```r
# Run regression of y on x, save as reg1
reg1<-lm(y~x, data=mydata) # runs regression using mydata
summary(reg1$coefficients) # prints coefficients
```

2.6.3 Managing Your Workflow

- **Save often!**
  - Better yet, ask me about version control and GitHub

2.7 Getting Help

- You can get documentation, explanations, and examples of every command in `R`
  - simply type `?commandname` or `help("commandname")`
- Meet your new friend:
- Meet your new *best* friend:
- The **only** way to learn coding is by tweaking existing examples, messing up, and searching the internet for help!



how do I make a vector in r

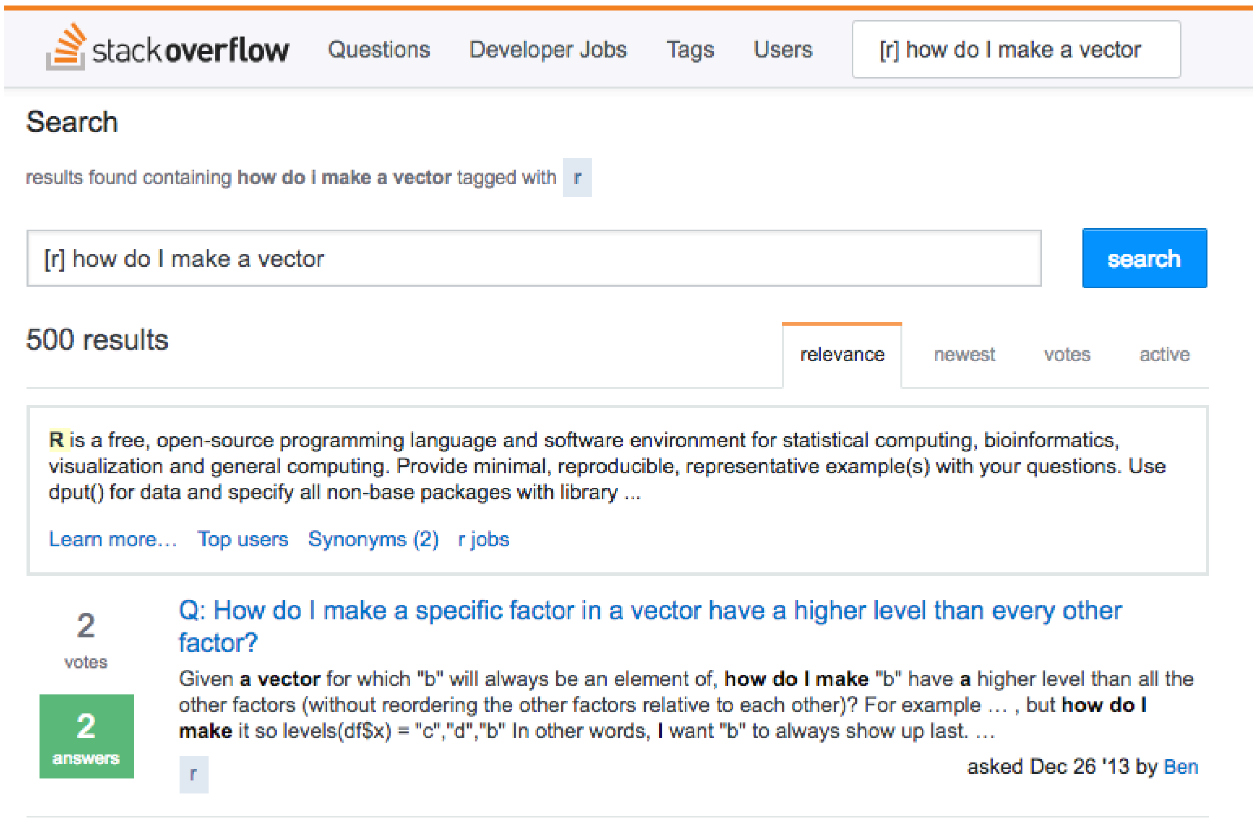
About 395,000,000 results (0.60 seconds)

R Vector: Create, Modify and Access Vector Elements - DataMentor
https://www.datamentor.io/r-programming/vector
In this article, you'll learn about **vector in R** programming. You'll learn to **create** them, access their elements using different methods, and modify them in your program. **Vector** is a basic data structure in **R**. It contains element of the same type.

Vector | R Tutorial
www.r-tutor.com/r-introduction/vector
An **R** tutorial on the concept of **vectors in R**. Discuss how to **create vectors** of numeric, logical and character string data types.

2. Basic Data Types — R Tutorial - Cyclismo
https://www.cyclismo.org/tutorial/R/types.html
We look at some of the ways that **R** can store and organize data. This is a ... You can **create a** list (also called a "**vector**") using the c command: > a <- c(1,2,3,4,5) > ...

Figure 2.2:



Search

results found containing **how do I make a vector** tagged with r

[r] how do I make a vector

500 results

R is a free, open-source programming language and software environment for statistical computing, bioinformatics, visualization and general computing. Provide minimal, reproducible, representative example(s) with your questions. Use dput() for data and specify all non-base packages with library ...

Q: How do I make a specific factor in a vector have a higher level than every other factor?

Given **a vector** for which "b" will always be an element of, **how do I make** "b" have **a** higher level than all the other factors (without reordering the other factors relative to each other)? For example … , but **how do I make** it so levels(df$x) = "c","d","b" In other words, **I** want "b" to always show up last. …

asked Dec 26 '13 by Ben

Figure 2.3:

Objects

`R` is an **object-oriented** programming language, meaning we will almost always store data in **objects** and run **functions** on those objects. We **assign** values to objects using the **assignment operator** (`<-`)[1]

```
myobject<-value
```

The keyboard shortcut for inserting `<-` (inside the Console or an `R` chunk) is `Alt+-` (on Windows) or `Option+-` (on Mac).

Functions take the form:

```
functionname(myobject)
```

Functions can have *other* functions for **arguments** (the object the function is run on), e.g.

```
round(rnorm(5),2)
# rnorm(5) takes 5 random draws from a normal distribution
# then round(, 2) rounds the result to 2 decimal places
```

2.8 Vectors

The simplest data structure in `R` is a **vector**, simply a collection of objects or elements. To construct a vector, use the "combine/concatenate" function "`c()`"

As an example, let's make a vector of the numbers 1 through 5, called `v`.

```
v<-c(1,2,3,4,5)
```

We can also build vectors via generating mathematical **series** with the `:` operator, which lists all integers in a series from `beginning:end`.

```
v<-1:5
```

To inspect an object, we simply "call" it up by typing the name of the object to print its contents.

```
v
```

```
## [1] 1 2 3 4 5
```

2.8.1 Functions

Since a vector is an object, we can run functions on that object. Let's start with some simple mathematical functions, such as taking the sum and taking the mean of our simple vector `v`.

[1]Think of the assignment operator like like an = sign, but we want to avoid using the equals sign. `<-` was originally its own key on early computer keyboards.

```r
sum(v)
```

```
## [1] 15
```

```r
mean(v)
```

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

Functions in `R` are "vectorized," meaning the function is run on every object inside a vector.

We can perform mathematical operations on a vector as a whole:

```r
sum(1:5)
```

```
## [1] 15
```

```r
mean(1:5)
```

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

Chapter 3

Other Object Types

3.1 Lists

- A **list** is a non-atomic vector, meaning you can gather data elements of different classes in one object

```
mylist<-list(5, pi, TRUE, 4.3, "cabbage")
class(mylist)
```

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

- Another great property of lists is that elements of the list can themselves be vectors

```
vectored.list<-list(c(1.82, 1940, 93.20, 192.917),
     c("Orange", "Cyan", "Pink"),
     c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, FALSE, FALSE))

str(vectored.list) # look at structure of the list
```

```
## List of 3
##  $ : num [1:4] 1.82 1940 93.2 192.92
##  $ : chr [1:3] "Orange" "Cyan" "Pink"
##  $ : logi [1:8] TRUE FALSE TRUE TRUE FALSE TRUE ...
```

- We can create a **label** for each element in a list, called a `name`

```
vectored.list<-list(numbers=c(1.82, 1940, 93.20, 192.917), # first element is a vector called 'numbers'
     colors=c("Orange", "Cyan", "Pink"), # second element is a vector called `colors`
     logic=c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, FALSE, FALSE)) # third element is a vector called `l

vectored.list
```

```
## $numbers
## [1]    1.820 1940.000   93.200  192.917
##
## $colors
## [1] "Orange" "Cyan"   "Pink"
##
## $logic
## [1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE
```

- The `names` command prints (or changes) the name of the label of each element in the list

```
names(vectored.list) # print the names of the list elements
```

```
## [1] "numbers" "colors"  "logic"
```

```
names(vectored.list)<-c("name1","name2","name3") # rename the lables to 'name1', 'name2', and 'name3'

names(vectored.list) # print new names
```

```
## [1] "name1" "name2" "name3"
```

```
vectored.list # print list with new names
```

```
## $name1
## [1]    1.820 1940.000   93.200  192.917
## 
## $name2
## [1] "Orange" "Cyan"   "Pink"  
## 
## $name3
## [1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE
```

3.2 Matrix

- Everything thus far has been 1 dimension, but we often work with 2-dimensional data
  - Rows are observations
  - Columns are variables
- A **matrix**
  - `matrix()` command creates a matrix by column,
    - can define number of rows with `nrow=`, `R` will divide the elements into equal number of columns

```
matrix1<-matrix(c(1,2,3,4,5,6),nrow=3) # make a 3-row matrix
matrix1
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

3.3 Data Frame

- The most important object in `R` is a **data frame** (what you call a "spreadsheet"), used for statistics, plots, regressions, etc
  - "Rectangular" data, rows are observations, columns are variables
  - Can hold variables of different classes (e.g. a quantitative variable like income, a character variable like name, etc)
  - In essence, data frames are actually lists (where each list object itself is a vector)
  - All vectors (columns) must have the same length!

```
df<-data.frame(x = 1:3,
           y = c("a", "b", "c"),
           z = c(TRUE, FALSE, TRUE))
df
```

```
##   x y     z
## 1 1 a  TRUE
## 2 2 b FALSE
## 3 3 c  TRUE
```

Chapter 4

Data Classes

- Vectors **must** contain the same type of elements (e.g. numerical or text)
- Technically this refers to **atomic vectors** (nearly all vectors are atomic)
- Vectors with "mixed" types will convert all elements to the lowest-common denominator, e.g. character
- You can always check the type of vector using **class()**

```
mixed<-c(5, pi, TRUE, 4.3, "cabbage")
class(mixed)
```

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

4.0.1 Numeric

- **Numeric** (aka "double"), as it sounds, can perform mathematical operations

```
numeric<-c(1,2,3,4,5)
```

- There are two **types** of numeric objects: **double** and **integer**

4.0.2 Double

- If numeric values contain decimal points, they are technically called **floating point double** or simply **double** class
- `R` may simply call them numeric, but contrast with integer below

```
double<-c(pi,2.34,9.99)

class(double)
```

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

```
typeof(double) # will return the more specific type
```

```
## [1] "double"
```

```
is.double(double) # a logical test to see if object is "double" type
```

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

```
is.integer(double) # a logical test to see if object is "integer" type
```

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

4.0.2.1 Integer

- If numeric values are all whole numbers, they are **integer** class

```
integers<-c(1,2,3,4)
class(integers)
```

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

```
typeof(integers)
```

```
## [1] "double"
```

```
is.double(double)
```

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

```
is.integer(double)
```

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

4.0.3 Logical

- **Logical** is a series of binary elements or statements that can either be TRUE or FALSE

```
logical<-c(TRUE,FALSE,FALSE,TRUE)
```

- We can perform logical tests with common operators:
  - `<` less than
  - `>` greater than
  - `<=` less than or equal to ($\leq$)
  - `>=` greater than or equal to ($\geq$)
  - `==` is equal to (note two equals signs are needed!)
  - `!=` is not equal to
  - `%in%` is a member of a set ($\in$)

```
3==4 #is 3 equal to 4?
```

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

```
3<4 # is three less than 4?
```

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

```
3<=4 # is three less than or equal to 4?
```

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

```
3>4 # is three greater than 4?
```

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

```
3!=4 # is three not equal to four?
```

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

```
3 %in% c(0,1,2) # is three in the following set of numbers?
```

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

```
3 %in% c(0,1,2,3) # is three in the following set of numbers?
```

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

- We are not limited to using numeric data, `R` can also perform logical tests on other classes of variable, like characters (which need quotes):

```
"red"=="blue" # is red the same as blue?
```

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

```
"red"!="blue" # is red not equal to blue?
```

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

```
political.party<-c("Republican","Democrat") # define political party as a set of Republican and Democra
"Libertarian" %in% political.party # check if Libertarian is in the set of political parties we created
```

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

```
"Democrat" %in% political.party # check if Democrat is in our set of political parties
```

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

- We can also perform more than one test at a time with multiple conditions:
  - `&` AND
  - `|` OR

```
2==2 & 2>3 # is 2 equal to 2 AND greater than 3?
```

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

```
2==2 | 2>3 # is 2 equal to 2 OR greater than 3?
```

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

- These commands will become very useful when we want to subset data or look at portions of our data based on some condition

4.0.4 Character

- **Character** is a string of text: letters, numbers, and symbols, cannot perform mathematical operations
  - Character values require quotation marks around each value

```
character<-c("one","two","7","orange")
```

4.0.4.1 Dates

- Dates are a specific type of character class
- Specific dates
  - Can do days, weeks, months, quarters, years

```
today<-Sys.Date() #print today's date
format(today, format="%B %d %Y") # specify how to report date format
```

```
## [1] "December 15 2018"
```

```
months<-seq(as.Date("2010/1/1"), as.Date("2012/1/1"), "months") # generate sequence of months between J
months
```

```
##  [1] "2010-01-01" "2010-02-01" "2010-03-01" "2010-04-01" "2010-05-01"
##  [6] "2010-06-01" "2010-07-01" "2010-08-01" "2010-09-01" "2010-10-01"
## [11] "2010-11-01" "2010-12-01" "2011-01-01" "2011-02-01" "2011-03-01"
## [16] "2011-04-01" "2011-05-01" "2011-06-01" "2011-07-01" "2011-08-01"
## [21] "2011-09-01" "2011-10-01" "2011-11-01" "2011-12-01" "2012-01-01"
```

4.0.5 Factor

- **Factor** is a special type of character variable, often used to indicate membership in one of several possible categories, called **levels** (e.g. for plotting, or conditional statistics and data work)

```
students<-factor(c("freshman", "senior", "senior", "junior", "freshman",
                   "sophomore", "freshman"))
students # note order is arbitrary
```

```
## [1] freshman  senior    senior    junior    freshman  sophomore freshman
## Levels: freshman junior senior sophomore
```

```
levels(students) #extract unique levels
```

```
## [1] "freshman"  "junior"    "senior"    "sophomore"
```

```
nlevels(students) #count the number of levels
```

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

```
table(students) #tabulate number of values for each level
```

```
## students
##  freshman    junior    senior sophomore
##         3         1         2         1
```

4.0.5.1 Ordered Factors

- Factors have ordered `levels()` which control the order on plots and in `table()`

```
students.o<-ordered(students, levels=c("freshman","sophomore","junior","senior"))
students.o
```

```
## [1] freshman  senior    senior    junior    freshman  sophomore freshman
## Levels: freshman < sophomore < junior < senior
```

- **Be advised**: when `R` stores and calls factors, it actually stores them as integers [1..k, for k categories] instead of characters (e.g. "freshman"=1, "sophomore"=2), making this a **nominal** variable. This allows for some mathematical operations.
- An **ordered factor** is where the ordering matters (e.g. "small", "medium", "large" coded as 1, 2, 3 in order)

4.1 Checking or Reclassifying Objects

- We can always check the class of an object with `class()` or `typeof()`.
  - We can perform logical tests `is.numeric()`, `is.factor()`, etc. to see if an object is a specified type
  - We can change the class of an object by redefining it with `as.classname()`, e.g.

```
x<-1:5
is.numeric(x) # check if x is numeric
```

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

```r
is.factor(x) # check if x is a character
```

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

```r
x<-as.character(x) # change vector x to a character
class(x)
```

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

```r
x<-as.numeric(x) # change vector x back to numeric
class(x)
```

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

Data Wrangling

- 90% of data work is “wrangling” raw data files into something we can actually work with

There are perhaps 5 common tasks that most data analysis will require some combination of: 1. Importing data (from an external source) 2. Merging data (from multiple sources) 3. Tidying data (transforming it to a useful structure) 4. Subsetting data (for conditional analysis) 5. Summarizing data (in summary statistics and plots)

4.2 Packages

All of the tasks in this section can be undertaken with base `R` commands. However, several packages make these tasks much more efficient and intuitive to understand and document. `- dplyr - tidyr - readr`

4.3 Importing Data

`getwd() setwd() list.files() list.dirs()`

4.4 Merging Data

- A simple merge

4.5 Using the `%>%` “Pipe” Operator

4.6 Subsetting

4.7

Chapter 5

Plotting

Data visualization is one of the most useful tools and gives you the most "bang for your buck." Base `R` is very powerful and intuitive to plot, but is not very aesthetically pleasing or advanced. We also use the `ggplot2` package, part of the `tidyverse` to suit our advanced plotting needs.

5.1 Plotting in Base `R`

The basic syntax is quite simple, put the variable(s) you wish to plot (which come from a dataframe) inside the argument of a plot function:

```r
plottype(my_df$my_variable1, my_df$my_variable2)
```

If you are using multiple variables, you can avoid having to invoke the same dataframe and `$` multiple times by just including the `names` of the variables in the dataframe, and then add `, data=my_df` as the final argument of the function, e.g.

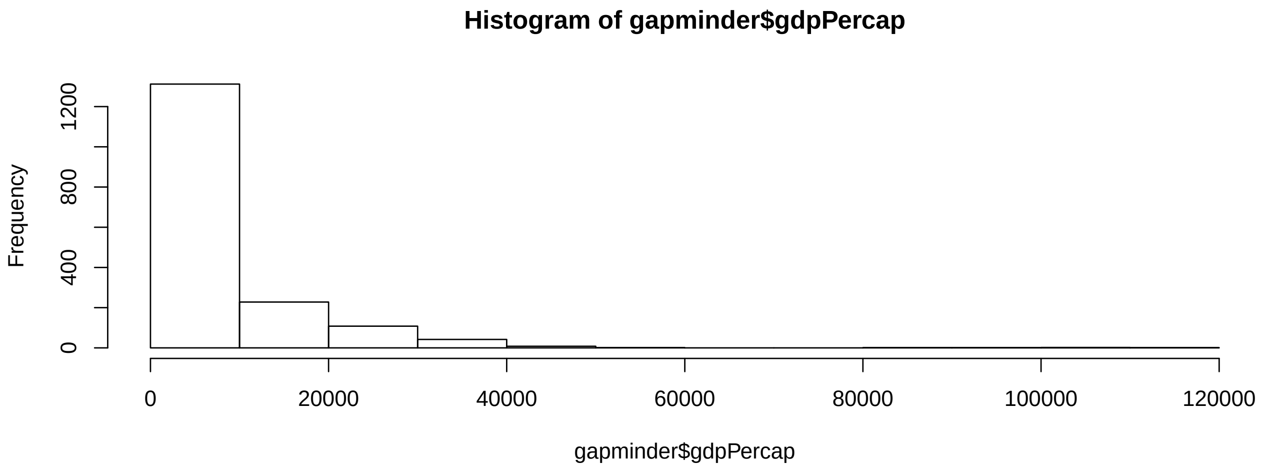
```r
plottype(my_variable1, my_variable_2, data=my_df)
```

The three simple plots that we can look at are

Function	Plot Type	When Used
`hist()`	Histogram	Exploring the distribution of a single variable
`barplot()`	Bar Graph	Exploring the counts of different categories of a variable
`boxplot()`	Boxplot	Exploring the distribution of a single variable
`plot()`	Scatterplot	Exploring the relationship between two variables

5.1.1 Histogram

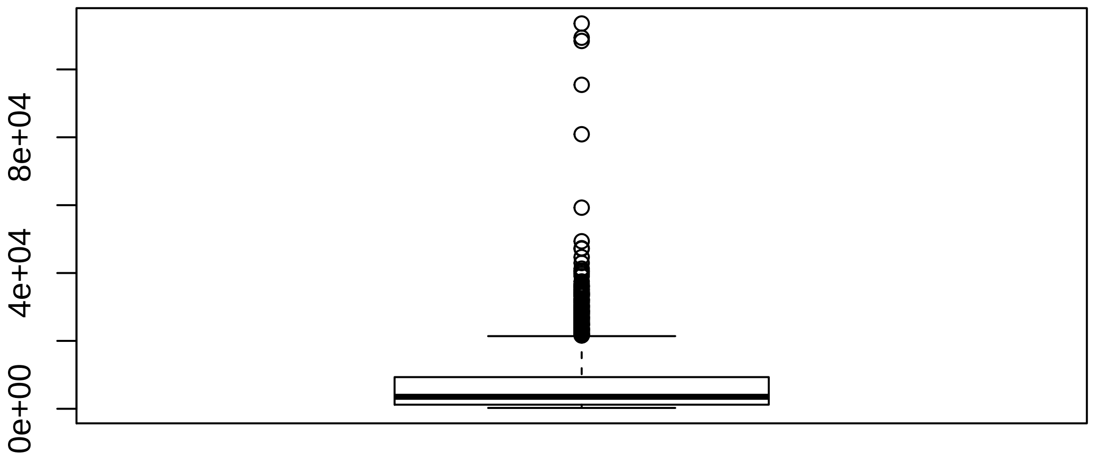
```r
library("gapminder")
hist(gapminder$gdpPercap)
```



5.1.2 Boxplot

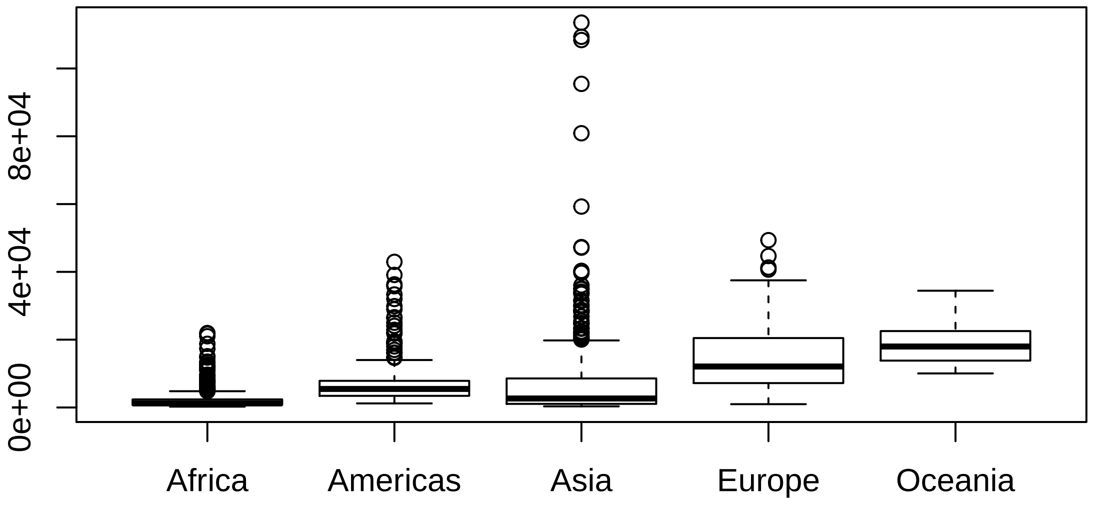
- Boxplots are similar syntax

```
boxplot(gapminder$gdpPercap)
```



- If we want a boxplot for each category, use `variable.name~category.variable.name` to tell `R` to plot a boxplot **by** category

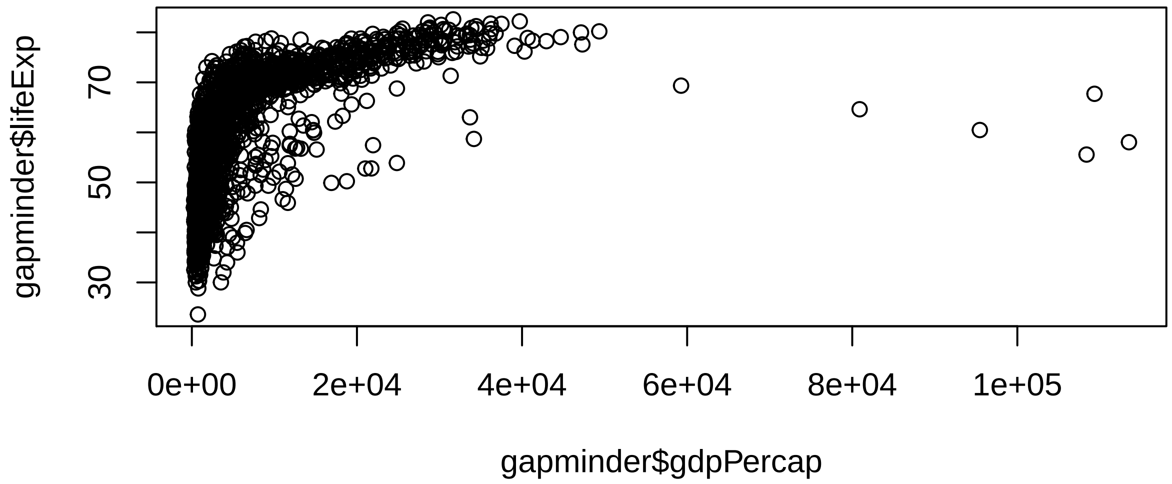
```
boxplot(gdpPercap~continent,data=gapminder)
```



5.1.3 Scatterplot

- Scatterplot syntax for plotting is similar to `hist()` and `boxplot()`: `plot(df$x,df$y)`

```r
plot(gapminder$gdpPercap, gapminder$lifeExp)
```



5.2 With ggplot2

'`ggplot2` is one of the premier packages at the center of the `tidyverse`. It is very powerful and creates beautiful data visualizations, but has a steeper learning curve at first. All of those "cool graphics" you see in media outlets such as the New York Times, fivethirtyeight, Vox, the Economist, etc use something are based off of `ggplot2`. The `gg` stands for a "grammar of graphics"

5.2.1 Two Ways to Plot

1. Just the single `ggplot` command
   - Will view plot right after producing it
   - Does not save as an object
   - Need to rerun or copy/paste full command producing plot in order to modify or view it again

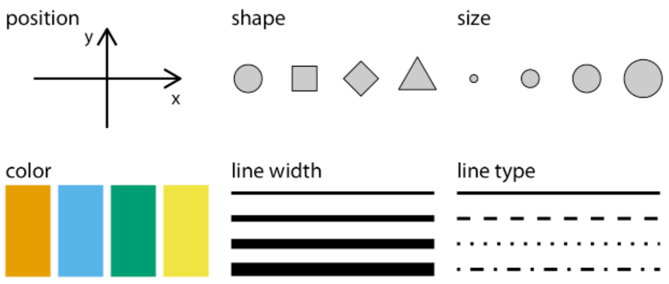


Figure 5.1:

- Can still put it in a document

```
ggplot(...) # make and view plot
ggplot(some.options) # remake plot with new options and view plot
```

5.2.2 Two Ways to Plot II

2. Create an object (as usual in R)
   - This allows you to save the plot for later (re)use
   - Also allows you to modify it
   - Any time you want to view display it (i.e. for putting it in a document), just call up the plot by name

```
plot.name<-ggplot(...) # make plot
plot.name<-plot.name+some.options # add new options to existing plot
plot.name # view plot
```

```
plot.name<-ggplot(data=mydf, mapping=aes(x=xvar,y=yvar))+
  geom_something(options)+
  moreoptions...
```

- gg "grammar of graphics" implies any graphic can be built from the same components/layers:
  1. **Data**: base-layer describes the data used
     - mydf is the dataframe containing xvar and yvar
     - aes() "aesthetics" identifies xvar (and if applicable yvar) from data to be "mapped" to a visual mark
  2. **Geoms**: visual marks that represent data observations or models, common examples:
     - e.g. geom_point, geom_line, geom_bar, geom_histogram, geom_density, geom_boxplot
  3. **Coordinates**: Cartesian coordinates are default
     - change scales, axes, labels, etc; advanced options like maps
- Most important idea to master is aes() **aesthetics** that map data to visual markings
- Aesthetics come in many forms and many options, depending on the context of the data
  - Must identify position (e.g. what is x and y)
  - Determine the marking with various geoms (points, bars, lines, boxes, etc)
  - Can pass additional options into geom (color, size, shape, etc)
    * Particularly important if we want color, size, or shape to depend on a particular variable in dataset

For our example, we'll use the mpg dataset loaded with the ggplot2 package

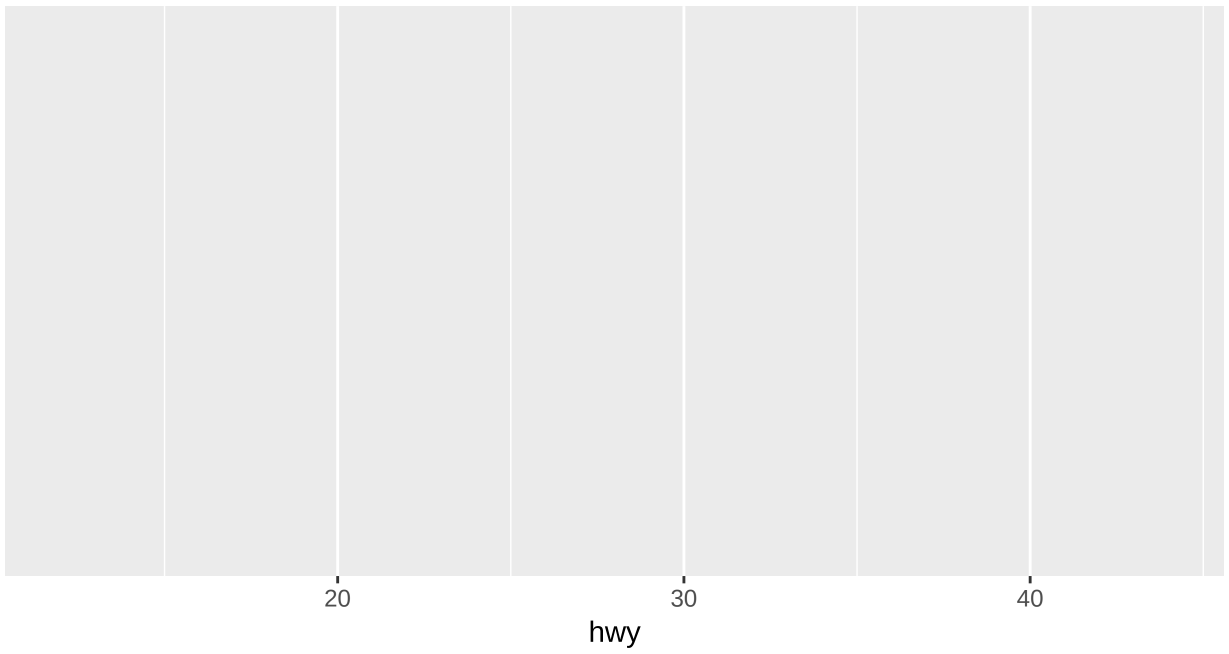
```
library("ggplot2") #load ggplot2
mpg #look at dataset
```

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

5.2.3 gg Histogram: Base Layer

- Start with the base layer: establish the data source, define x variable

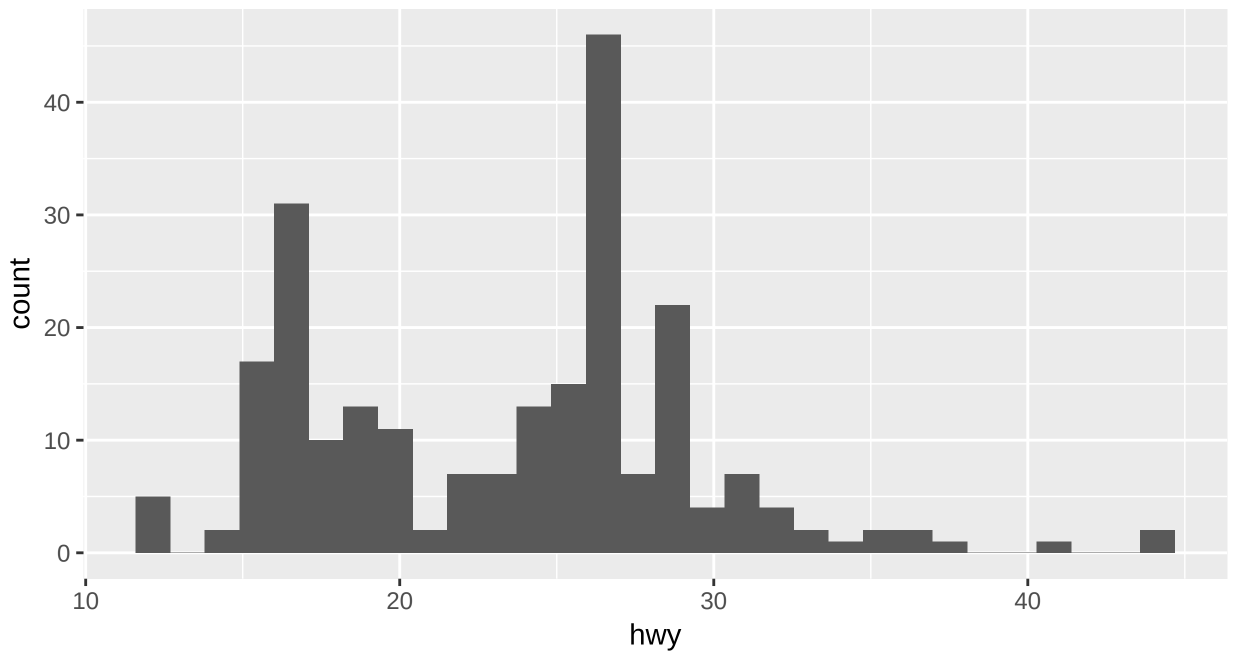
```
mpg.h<-ggplot(data=mpg,mapping=aes(x=hwy))
mpg.h
```



5.2.4 gg Histogram: Adding Geoms

- Add a histogram layer of `hwy`

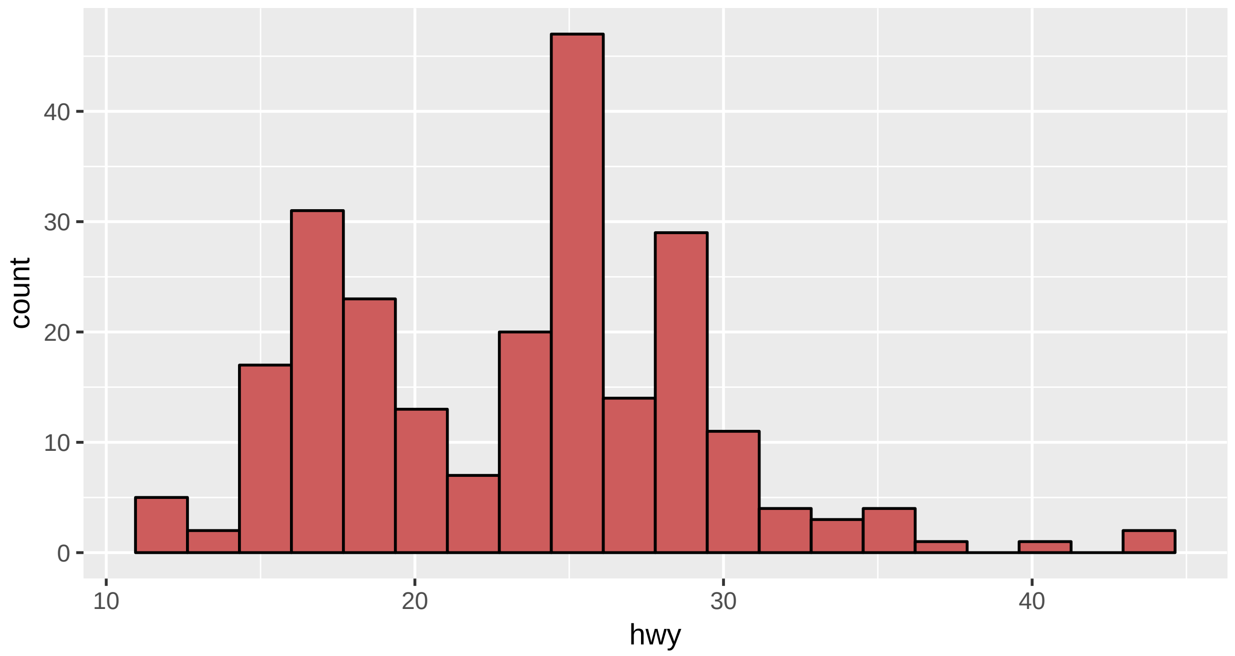
```
mpg.h1<-mpg.h+geom_histogram()
mpg.h1
```



5.2.5 gg Histogram: Customizing Geoms

- Edit the histogram (# of bins, color, etc)

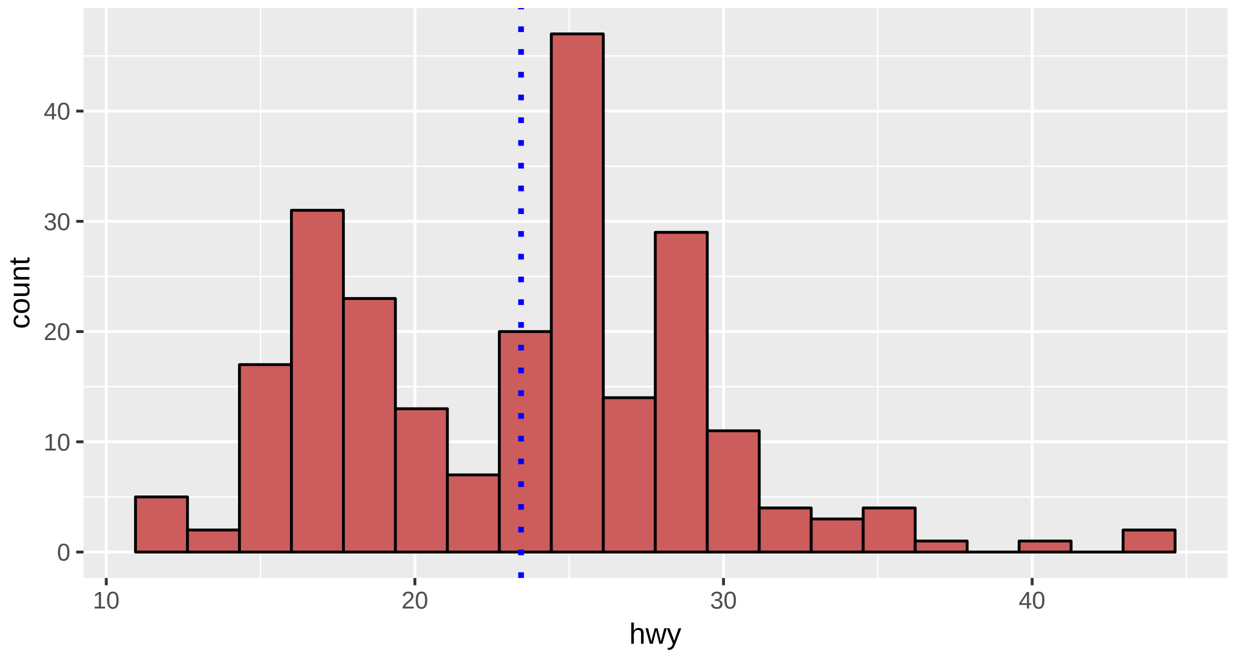
```
mpg.h2<-mpg.h+geom_histogram(bins=20, color="black",fill="indianred")
mpg.h2
```



5.2.6 gg Histogram: Adding Other Layers

- Add a vertical line for the mean with another `geom` called `vline`

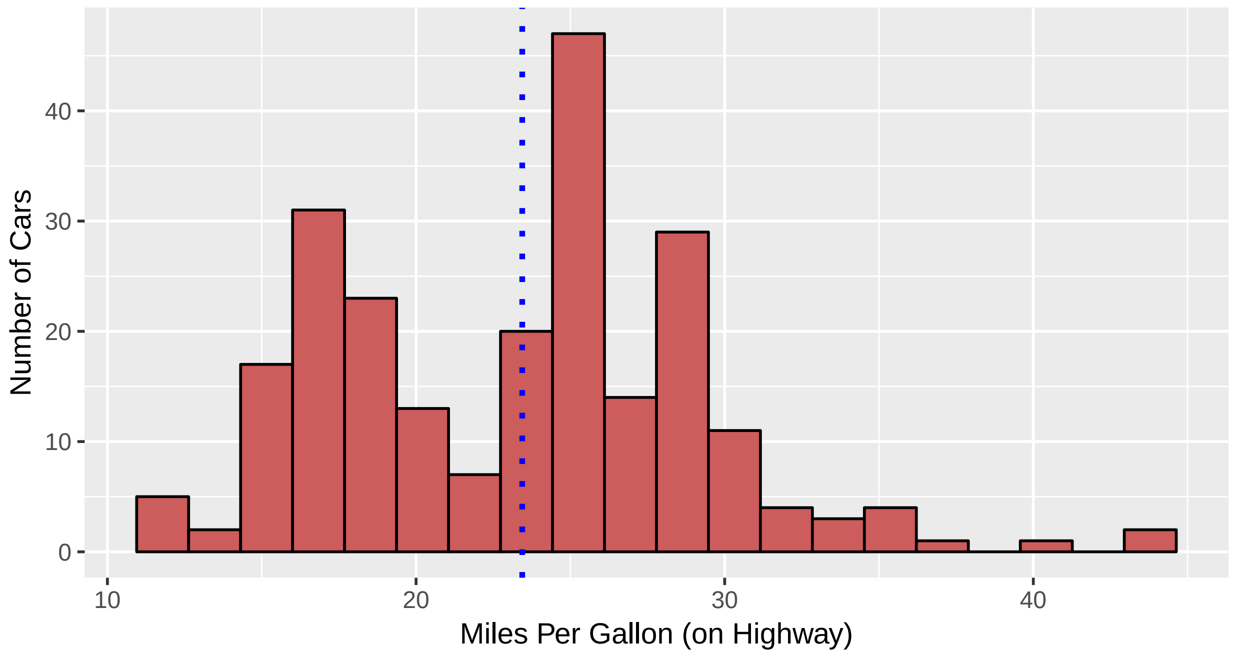
```
mpg.h2<-mpg.h2+
  geom_vline(xintercept=mean(mpg$hwy),linetype="dotted",color="blue",size=1)
mpg.h2
```



5.2.7 gg Histogram: Editing Coordinates (Axes)

- Change the labels on the axes with `xlab()` and `ylab()`

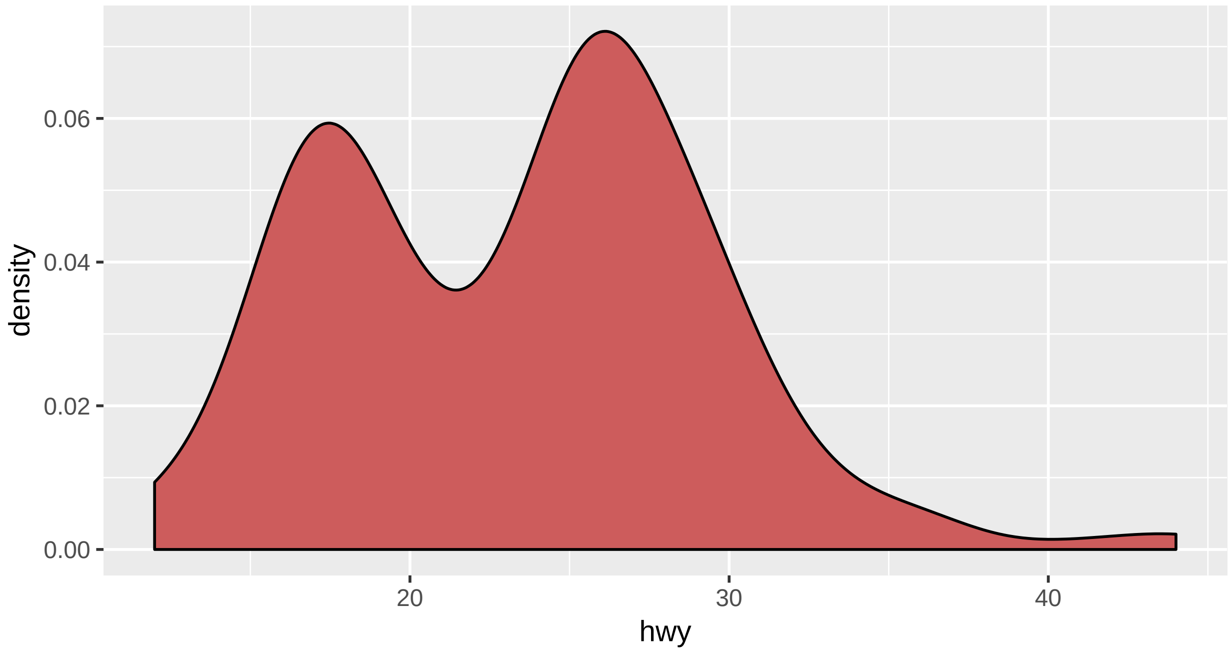
```
mpg.h2<-mpg.h2+xlab("Miles Per Gallon (on Highway)")+ylab("Number of Cars")
mpg.h2
```



5.2.8 gg Other Geoms

- How about a **density plot**: use `geom_density()` instead of `geom_histogram()`

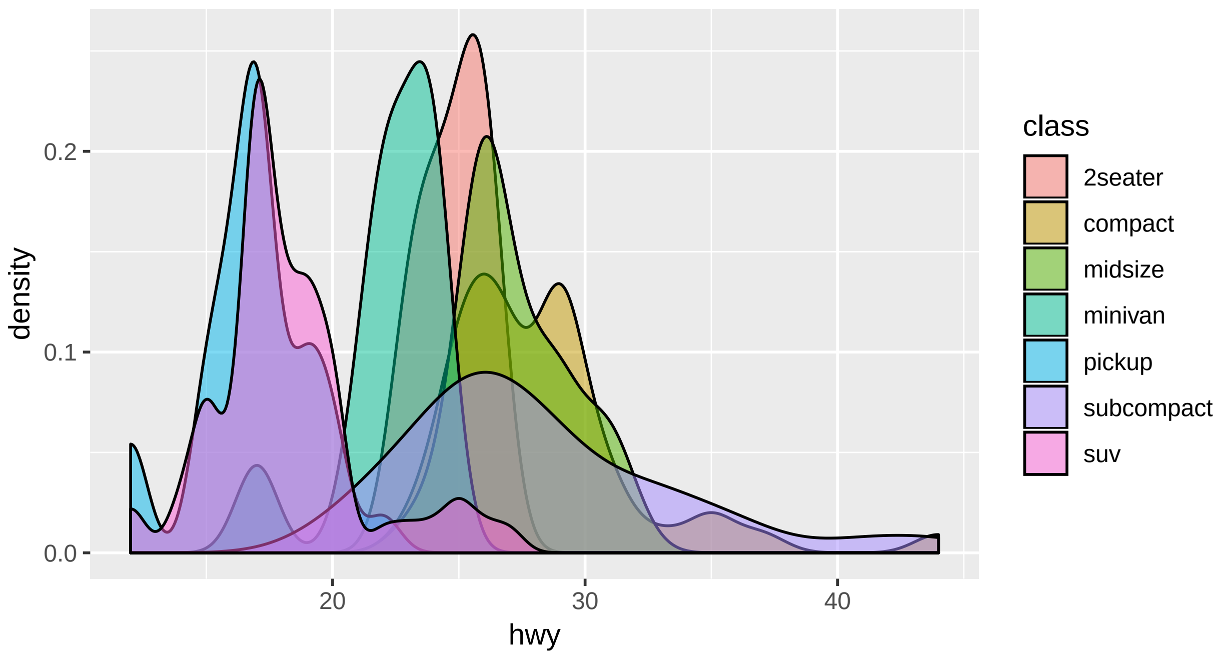
```
mpg.d<-ggplot(data=mpg,aes(x=hwy))+
  geom_density(fill="indianred")
mpg.d
```



5.2.9 gg Other Geoms

- Let's make a separate density plot for each `class`, set `aes` to `fill` by `class`

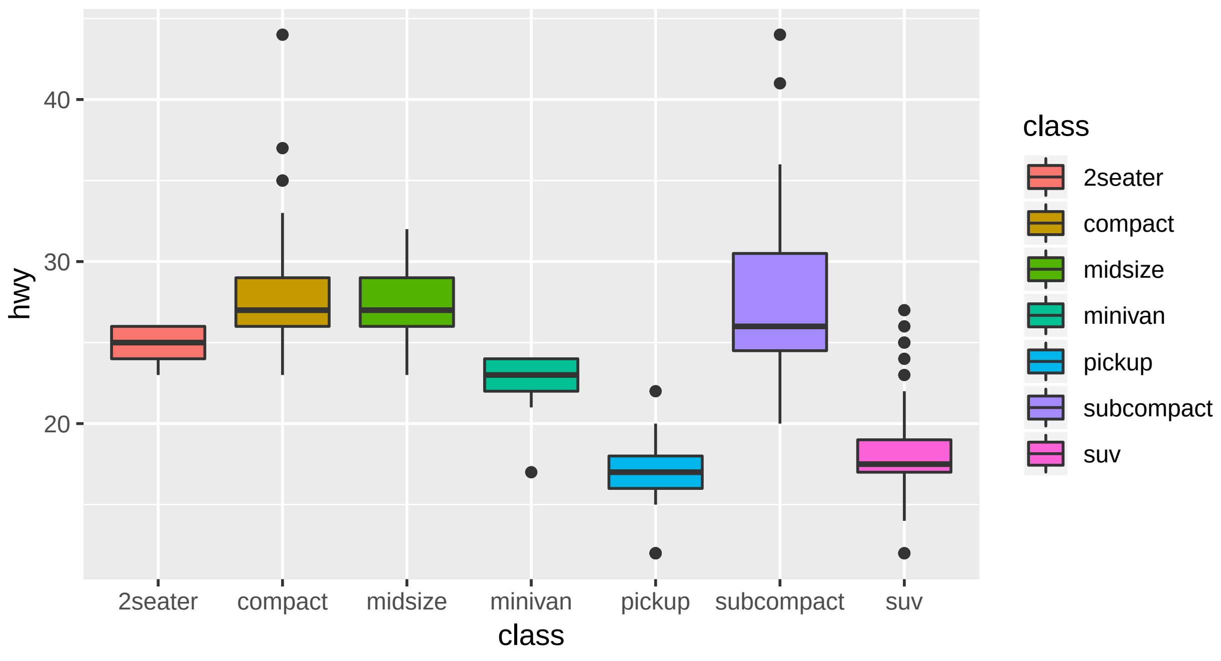
```
mpg.d<-ggplot(data=mpg,aes(x=hwy,fill=class))+
  geom_density(alpha=0.5) # alpha adds transparency
mpg.d
```



5.2.10 gg Boxplot

- Instead of a density plot, a `boxplot` by `class` (note now `x` is `class` and `y` is `hwy`):

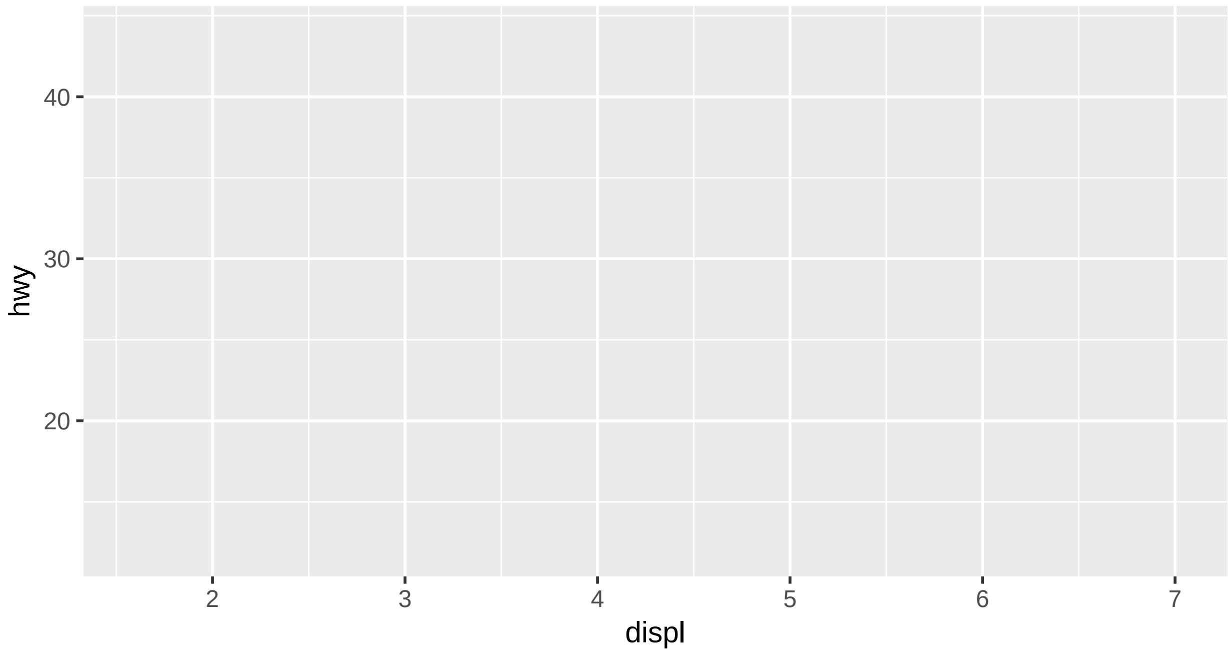
```
mpg.b<-ggplot(data=mpg,aes(x=class,y=hwy,fill=class))+
  geom_boxplot()
mpg.b
```



5.2.11 Scatterplot

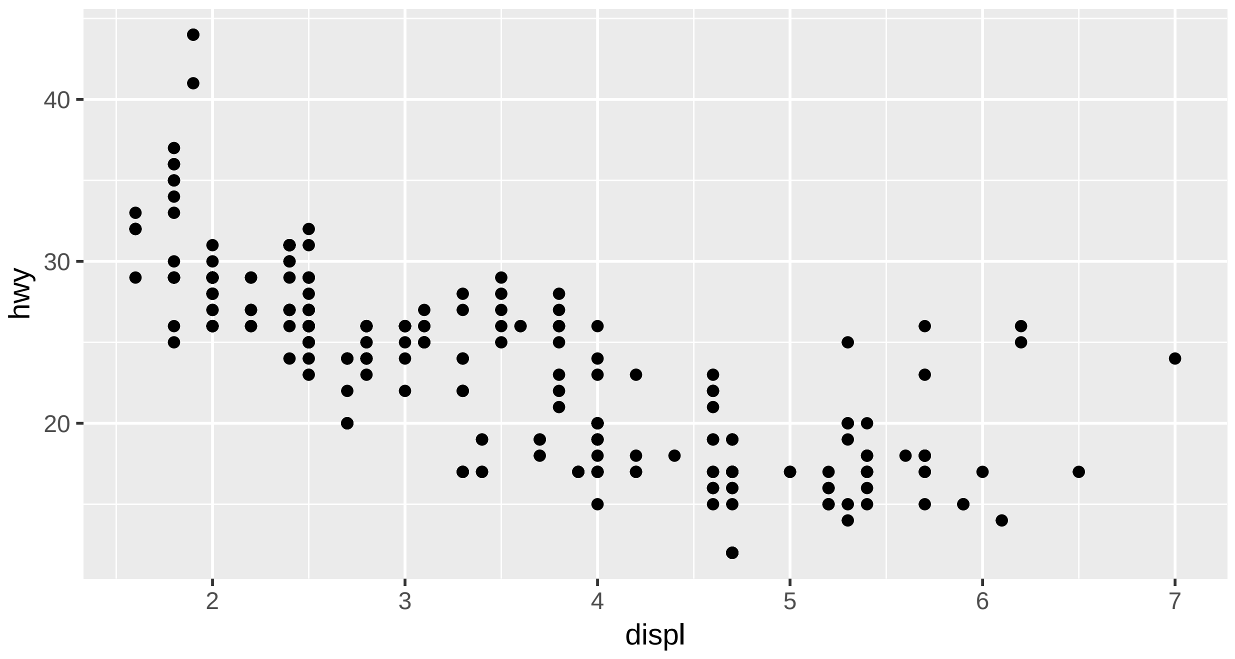
- Start with the base layer: establish data source, define x and y variables

```
mpg.p<-ggplot(data=mpg,aes(x=displ, y=hwy)) #use mtcars df, let x=displ, y=hwy

mpg.p
```



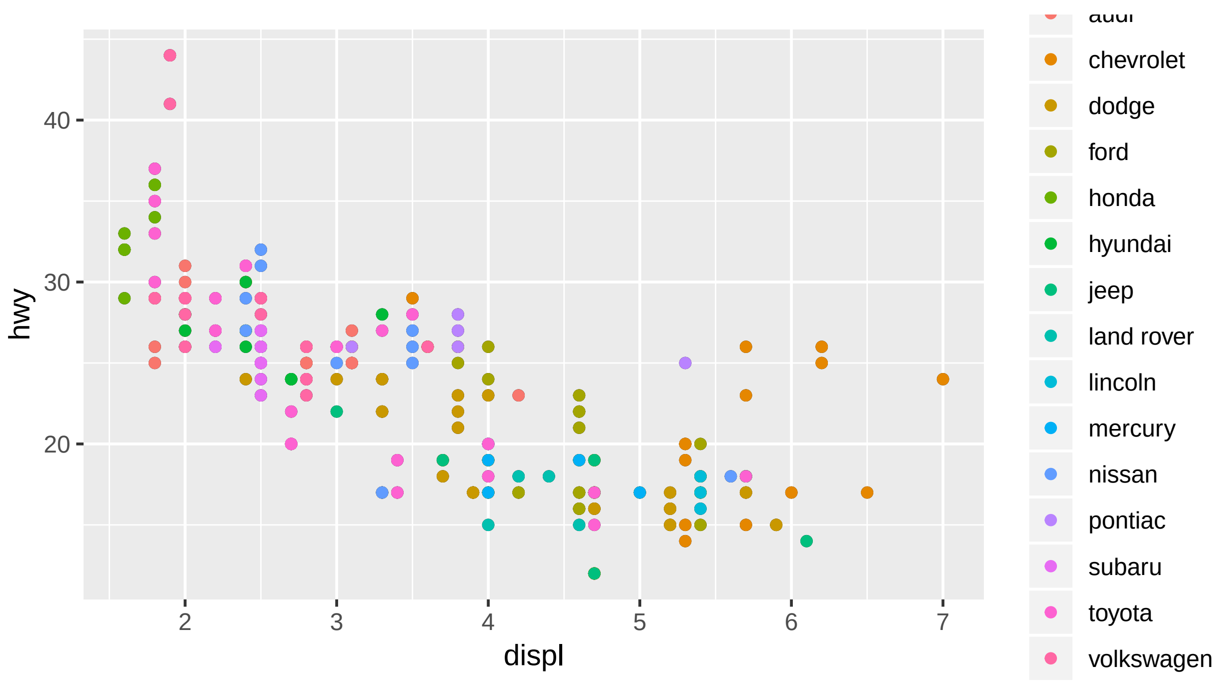
5.2.12 Scatterplot: Geom Layer

```
mpg.p<-mpg.p+geom_point() # specify observations as points on graph

mpg.p
```



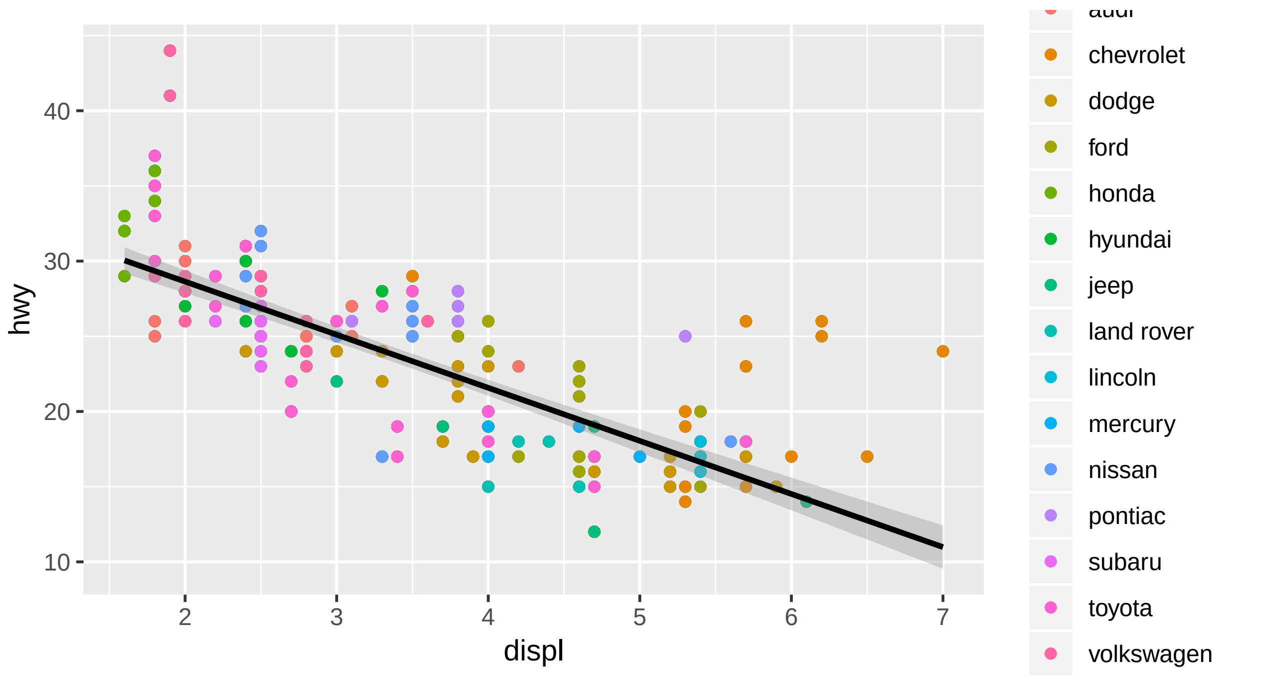
5.2.13 Scatterplot: Geom Layer Options

```r
mpg.p<-mpg.p+geom_point(aes(color=manufacturer)) # color data points by manuf.

mpg.p
```



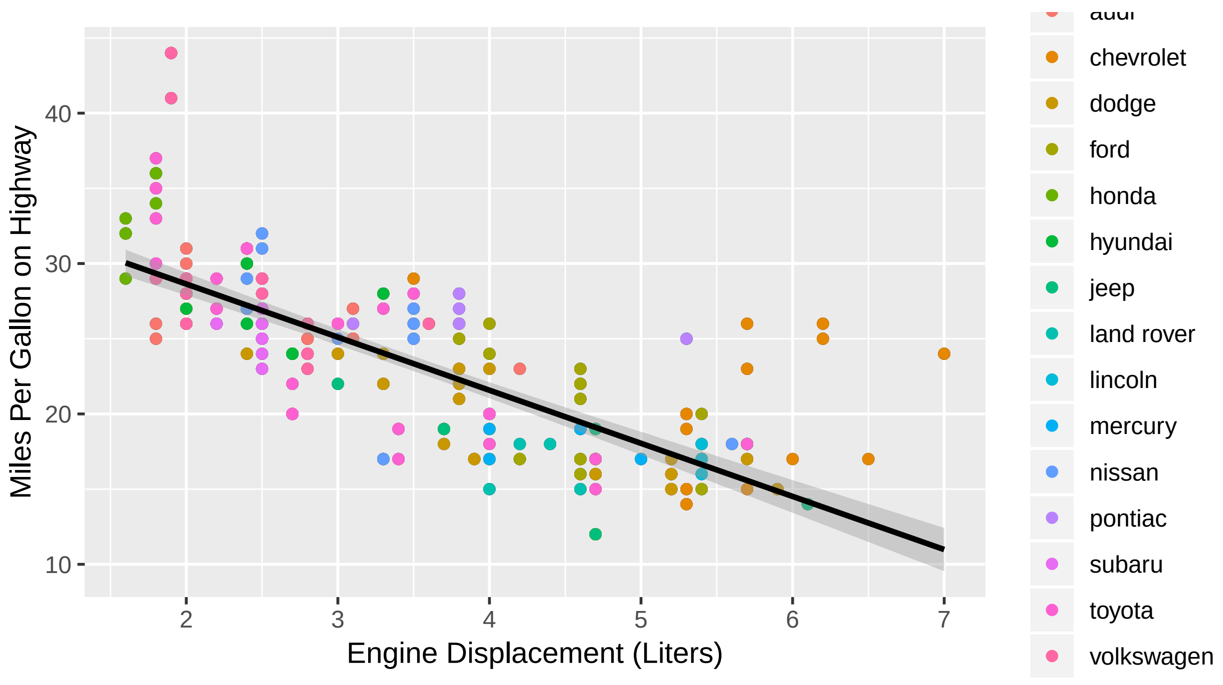
5.2.14 Scatterplot: Geom Layer II

```
mpg.p<-mpg.p+geom_smooth(method="lm", color="black") # add a black OLS line

mpg.p
```



5.2.15 Scatterplot: Coordinate Layer

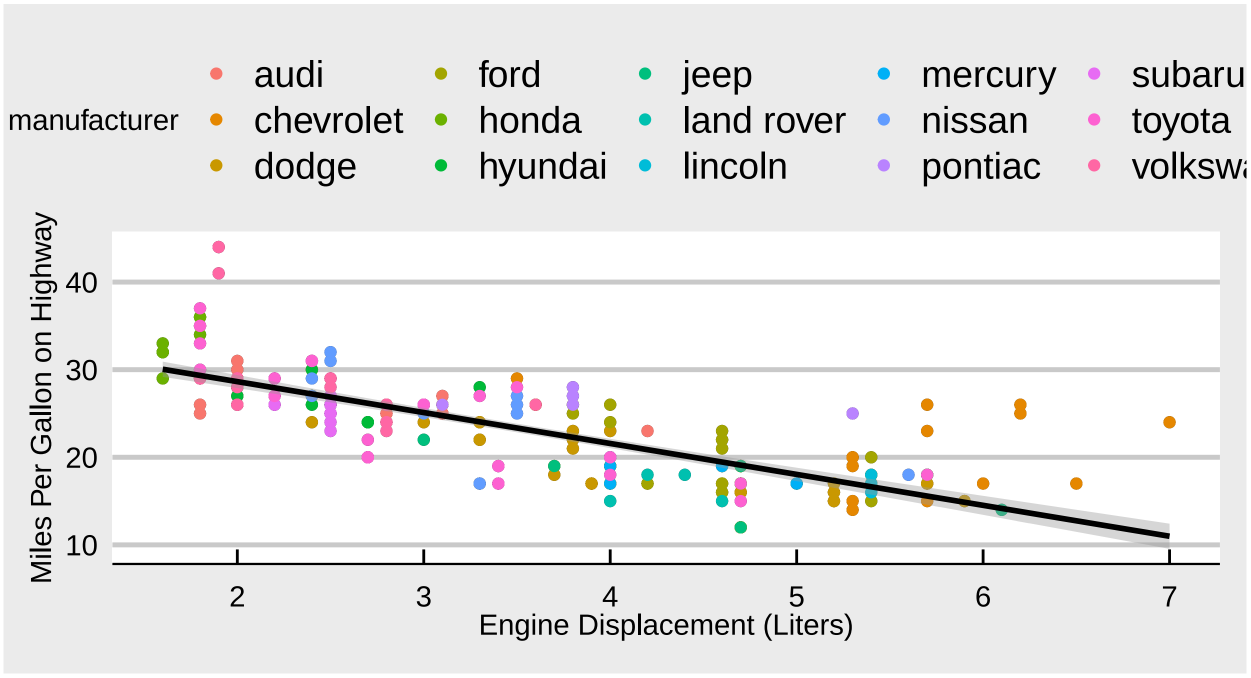
```
mpg.p<-mpg.p+xlab("Engine Displacement (Liters)")+
  ylab("Miles Per Gallon on Highway")
mpg.p
```



5.2.16 Scatterplot: Coordinate Options

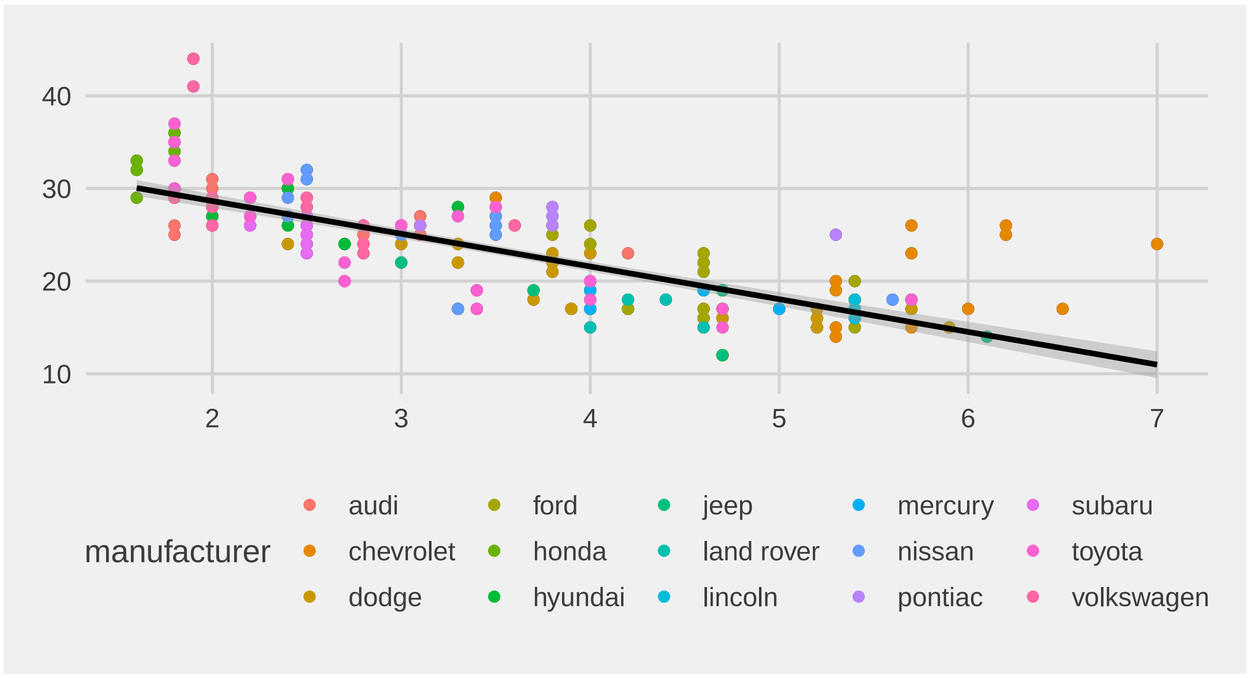
- Let's have some fun changing the theme

```
library("ggthemes") # need ggthemes package (install if first use)
mpg.p<-mpg.p+theme_economist_white() #make it look like The Economist magazine
mpg.p
```



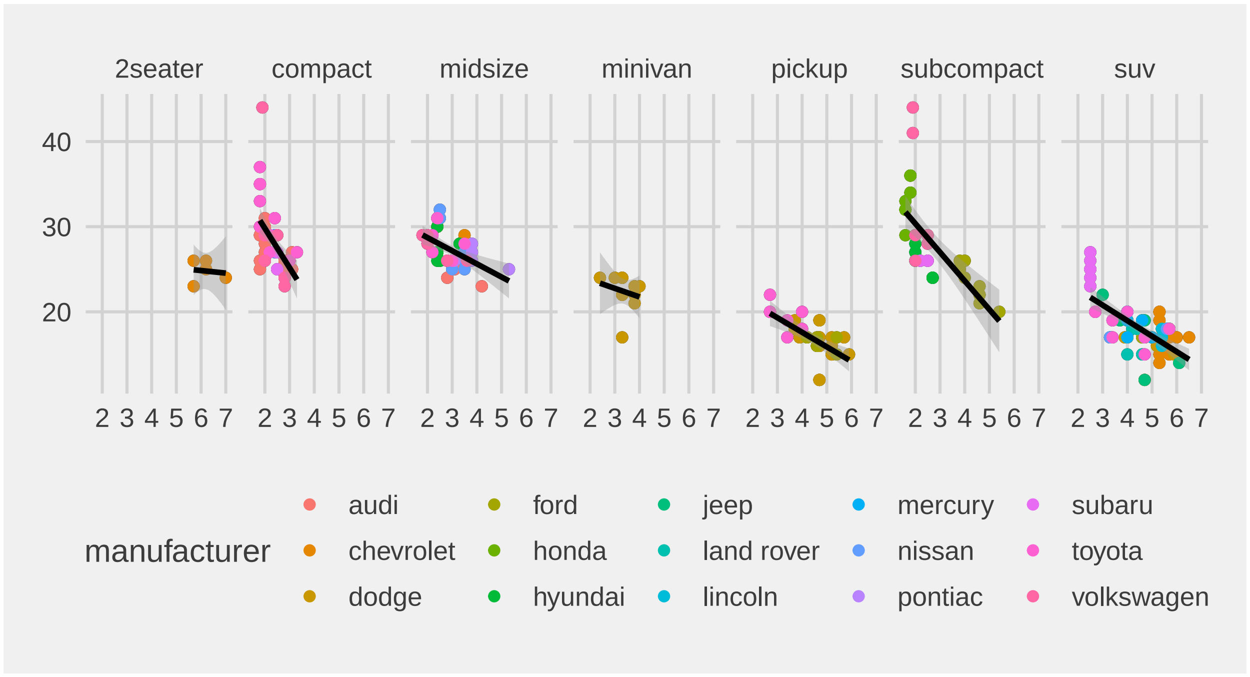
5.2.17 Scatterplot: Coordinate Options II

```
mpg.p<-mpg.p+theme_fivethirtyeight() #make it look like fivethirtyeight
mpg.p
```



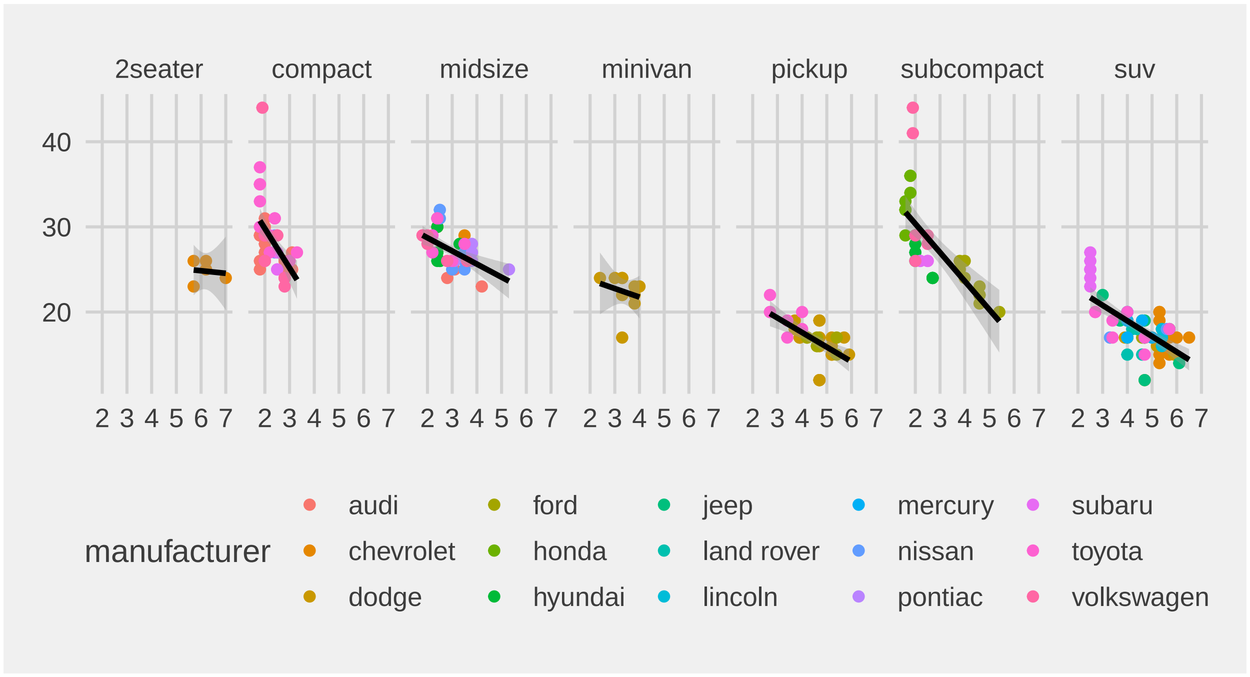
5.2.18 Scatterplot: Coordinate Options: Facetting

```
# make columns of separate 'facet' figures for each class of car
mpg.p<-mpg.p+facet_grid(cols = vars(class)) # make 'columns' by variable 'class'
mpg.p
```

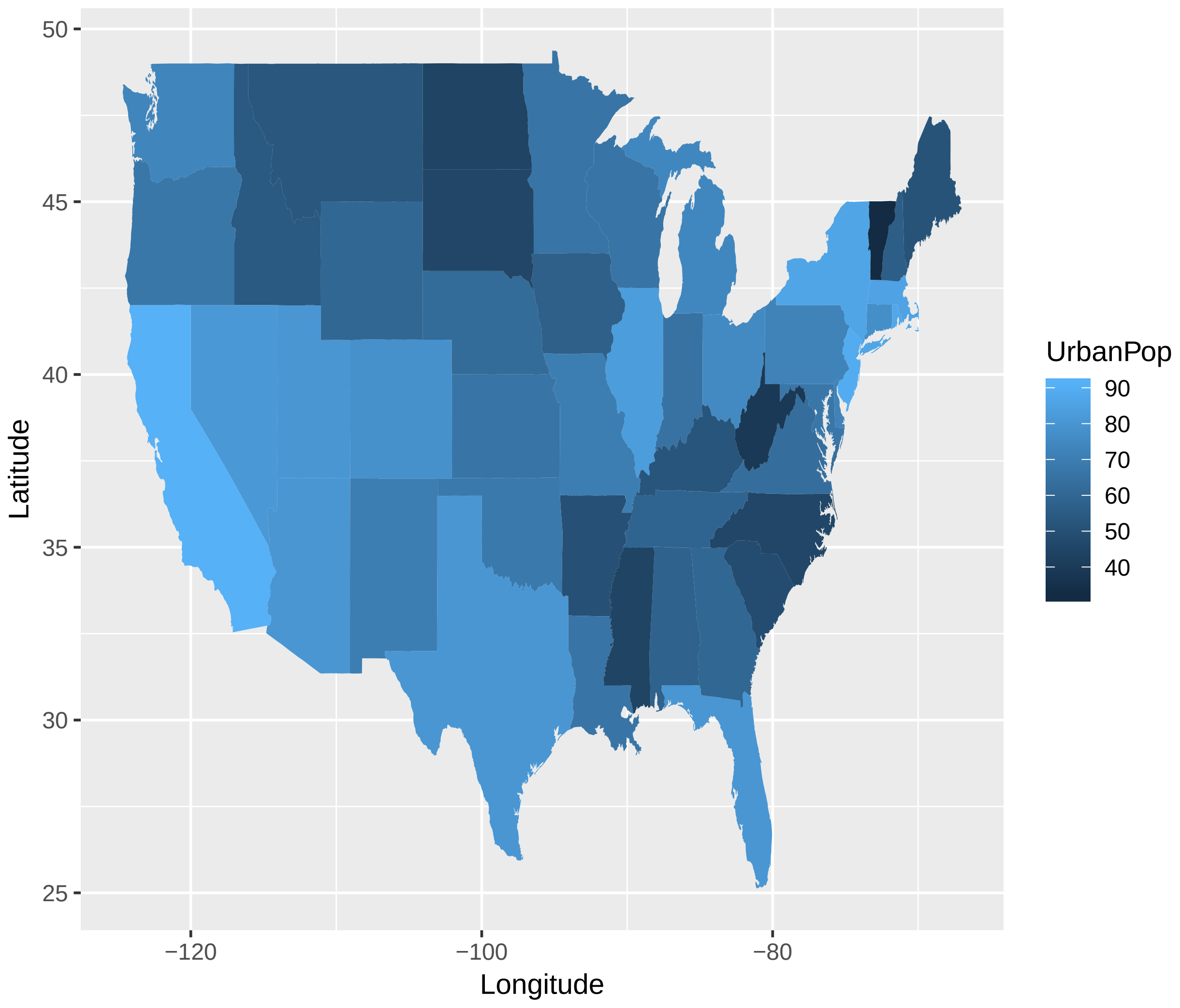


5.2.19 All Together Now

```
ggplot(data=mpg,aes(x=displ, y=hwy))+geom_point(aes(color=manufacturer))+
  geom_smooth(color="black",method="lm")+
  xlab("Engine Displacement (Liters)")+ylab("Miles Per Gallon on Highway")+
  theme_fivethirtyeight()+facet_grid(cols = vars(class))
```



5.2.20 Advanced Uses of `ggplot2`: Maps (See Rmd for Code)



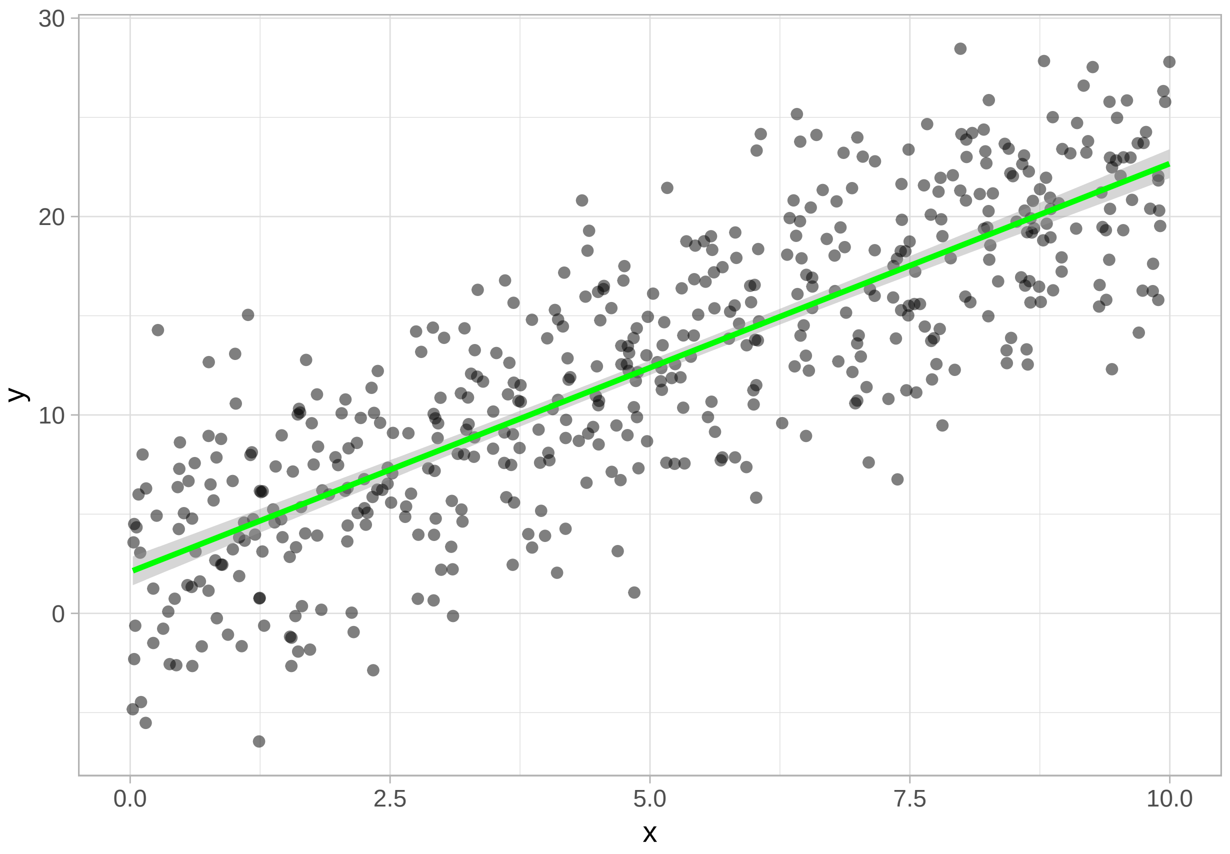
Chapter 6

Regression Basics

6.1 Ordinary Least Squares (OLS) Regression

In `R`, the Ordinary Least Squares (OLS) regression model is simply called the **"linear model"**, abbreviated `lm`. Regressions are run on several variables from a `data.frame` and stored as a `lm` object that we can inspect and modify.

```r
set.seed=1 #makes 'random' draws reproducible
x<-runif(500,min=0,max=10) #500 draws from uniform distr
y<-2*x+rnorm(500,2,4)
my_df<-data.frame(x,y)

ggplot(my_df, aes(x=x,y=y))+
        geom_point(alpha=0.5)+
        geom_smooth(method="lm", color="green")+
        xlim(c(0,10))+theme_light()
```



The syntax for running a regression in R is simple. We store the regression as an `lm()` object (e.g. called "`my_reg`") and regress our dependent (`mydf$y`) variable on (`~`) the independent (`my_df$x`) variable.

```
my_reg<-lm(df$y~df$x)
```

Alternatively, we can simply use the variable `names` from `my_df` and then tell R that the variables are coming from `my_df`:

```
my_reg<-lm(y~x, data = my_df)
```

$$y = \beta_0 + \beta_1 x$$

When we inspect our `lm` object, R simply prints the coefficients ("`Intercept`" for $\hat{\beta}_0$) and ("`x`" for $\hat{\beta}_1$ on x):

```
my_reg
```

```
##
## Call:
## lm(formula = y ~ x, data = my_df)
##
## Coefficients:
## (Intercept)            x
##       2.095        2.057
```

We can get a more detailed summary by running `summary()` on our `lm` object.

```
summary(my_reg)
```

```
##
## Call:
```

```
## lm(formula = y ~ x, data = my_df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -11.1018  -2.7631   0.0865   2.9730  11.6323
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.09494    0.36946    5.67 2.42e-08 ***
## x            2.05739    0.06413   32.08  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.136 on 498 degrees of freedom
## Multiple R-squared:  0.6739, Adjusted R-squared:  0.6733
## F-statistic:  1029 on 1 and 498 DF,  p-value: < 2.2e-16
```

The `summary()` prints:

- The formula for the regression
- A 5 number summary of the distribution of the residuals
- Table of coefficients
  - Column 1: Estimate for each β
  - Column 2: Standard error of each β
  - Column 3: t-statistic for each β with $H_0 : \beta = 0$
  - Column 4: p-value for the t-test
- Regression Diagnostics
  - Standard error of the regression (SER), `R` calls it `Residual standard error (RSE)`
  - `R-squared` and `Adjusted R-squared`
  - "All F-test" where $H_0 :$ all β's $= 0$

Inside the `lm` object `my_reg` is stored a lot of things that may not show up in the `summary`. To get a full inspection, check the structure with `str()`.

```r
str(my_reg)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 2.09 2.06
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "x"
##  $ residuals    : Named num [1:500] -3.62 2.2 3.73 2 -1.89 ...
##   ..- attr(*, "names")= chr [1:500] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:500] -276.27 -132.69 4.06 2.05 -1.89 ...
##   ..- attr(*, "names")= chr [1:500] "(Intercept)" "x" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:500] 20.08 21.02 2.26 19.17 22.28 ...
##   ..- attr(*, "names")= chr [1:500] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:500, 1:2] -22.3607 0.0447 0.0447 0.0447 0.0447 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:500] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "x"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.04 1.06
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 498
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = y ~ x, data = my_df)
##  $ terms        :Classes 'terms', 'formula'  language y ~ x
##   .. ..- attr(*, "variables")= language list(y, x)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "y" "x"
##   .. .. .. ..$ : chr "x"
##   .. ..- attr(*, "term.labels")= chr "x"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(y, x)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "y" "x"
##  $ model        :'data.frame':   500 obs. of  2 variables:
##   ..$ y: num [1:500] 16.46 23.22 5.99 21.17 20.39 ...
##   ..$ x: num [1:500] 8.743 9.1977 0.0807 8.2981 9.812 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language y ~ x
##   .. .. ..- attr(*, "variables")= language list(y, x)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "y" "x"
##   .. .. .. .. ..$ : chr "x"
##   .. .. ..- attr(*, "term.labels")= chr "x"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(y, x)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "y" "x"
##  - attr(*, "class")= chr "lm"
```

Note that `lm` objects are actually `lists`, (`data.frames` are also `lists`), so we can extract elements of the list and subset using `$` or `[[]]`. Some of the important elements of the list:

- `my_reg$coefficients` is a list of coefficients
- `my_reg$residuals` is a list comprised of the residual for each `x` value
- `my_reg$fitted.values` is a list comprised of the predicted/fitted value ($\hat{y}$) for each `x` value

```
my_reg$coefficients # look at coefficients
```

```
## (Intercept)           x
##    2.094943    2.057391
```

```
my_reg$residuals[1:5] # look at first 5 residuals
```

```
##         1         2         3         4         5
## -3.618573  2.204530  3.732511  1.999283 -1.887456
```

```
my_reg$fitted.values[1:5] # look at first 5 fitted.values
```

```
##         1         2         3         4         5
## 20.082632 21.018127  2.260903 19.167355 22.282027
```

These stored values will come in handy. We can run functions on them, for example, to discover things about the residuals:

```
summary(my_reg$residuals) # the same as the first thing printed in the regression output above!
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -11.10182  -2.76309   0.08653   0.00000   2.97297  11.63227
```

```
sd(my_reg$residuals) # the standard deviation of the residuals
```

```
## [1] 4.131991
```

Since these are stored in `lm` as objects, we can also assign them to new columns in our original `data.frame`, `my_df`. This can be helpful for plotting with `x`, `y`, the residuals ϵ, and the predicted values $\hat{y}$.

```
# save predicted values from model as "yhat"
my_df$yhat<-my_reg$fitted.values

# save residuals from model as "res"
my_df$res<-my_reg$residuals

# look at new dataframe
kable(head(my_df))
```

x	y	yhat	res
8.7429587	16.464059	20.082632	-3.618573
9.1976582	23.222657	21.018127	2.204530
0.0806652	5.993414	2.260903	3.732511
8.2980863	21.166638	19.167355	1.999283
9.8119796	20.394571	22.282027	-1.887456
0.0386886	4.502773	2.174541	2.328232

There are also specific functions for assigning the predicted values and the residuals to a `data.frame`, using the `lm` object as the argument. They will produce the same result as above.

```
# save predicted values from model as "yhat"
my_df$yhat<-predict(my_reg)

# save residuals from model as "res"
my_df$res<-residuals(my_reg)

# we get the same result
head(my_df)
```

```
##            x         y      yhat       res
## 1 8.74295870 16.464059 20.082632 -3.618573
## 2 9.19765820 23.222657 21.018127  2.204530
## 3 0.08066519  5.993414  2.260903  3.732511
## 4 8.29808628 21.166638 19.167355  1.999283
## 5 9.81197958 20.394571 22.282027 -1.887456
## 6 0.03868858  4.502773  2.174541  2.328233
```

6.1.1 Diagnostics

Some of the regression diagnostics are stored (idiosyncratically) in the `summary()` object, and can be extracted by name:

```
summary(my_reg)$sigma # extract residual squared error (SER)
```

```
## [1] 4.136138
```

```
summary(my_reg)$r.squared # extract R^2
```

```
## [1] 0.6739212
```

```
summary(my_reg)$adj.r.squared # extract adjusted R^2
```

```
## [1] 0.6732664
```

```
summary(my_reg)$f # extract the F-statistic
```

```
##    value    numdf    dendf
## 1029.238    1.000  498.000
```

These might be useful if we wished to perform manual calculations using these statistics. As an example, if we wanted to calculate the correlation coefficient between X and Y, and we know that R^2 is the correlation coefficient squared:

```
R2<-summary(my_reg)$r.squared
sqrt(R2)
```

```
## [1] 0.820927
```

```
# compare to actual correlation coefficient
cor(my_df$x, my_df$y)
```

```
## [1] 0.820927
```

6.2 Prediction

We can use the model to make pedictions using the estimated regression model.

$$\hat{Y} = 2.090 + 1.974X$$

```
x<-3
prediction<-my_reg$coef[1]+my_reg$coef[2]*x
prediction
```

```
## (Intercept)
##    8.267118
```

```
# multiple predictions
x<-c(1,3,7,10)
prediction<-my_reg$coef[1]+my_reg$coef[2]*x
prediction
```

```
## [1]  4.152335  8.267118 16.496684 22.668858
```

```
# alternatively, we can use the predict() function and insert
# a dataframe of our desired x values to predict y-hat
```

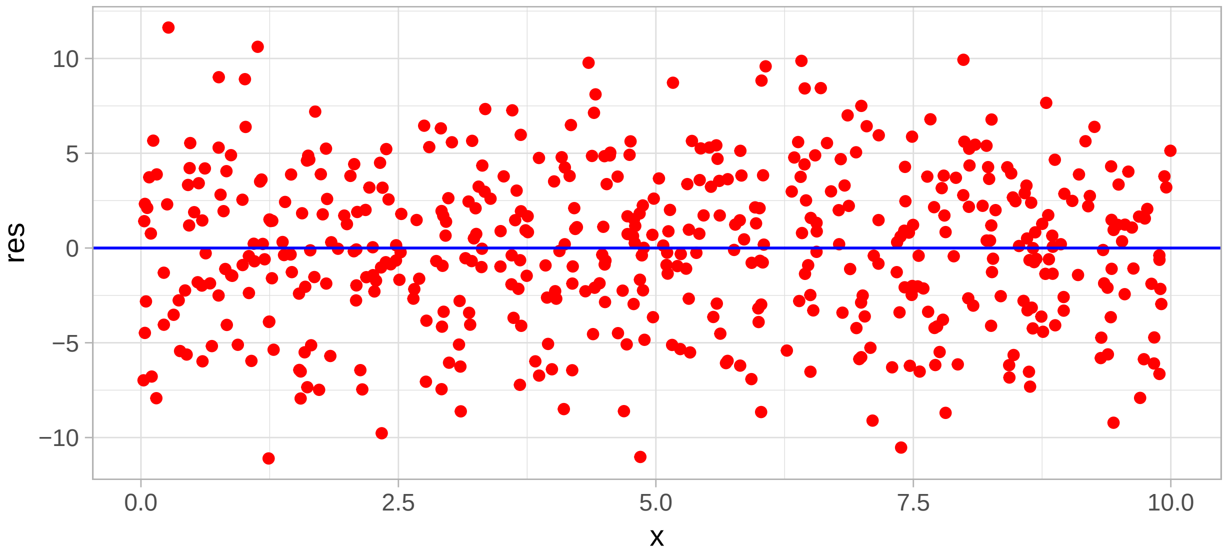
```r
prediction2<-predict(my_reg, data.frame(x=c(1,3,7,10)))
prediction2
```

```
##         1         2         3         4
##  4.152335  8.267118 16.496684 22.668858
```

6.3 Residual Plots

For more, see Plotting{#04-plotting}.

```r
ggplot(data = my_df, aes(x = x, y = res))+
  geom_point(color="red")+
  geom_hline(yintercept=0, color="blue")+ # add horizontal line at y=0
  theme_light()
```



6.4 Regression Output Table

The `broom` package converts `lm` objects into a tidy `data.frame` that can easily be printed in a nice table using `knitr`'s `kable()` function for `html` output.

```r
# install.packages("broom") # install first if you don't have
library(broom)
reg2<-tidy(my_reg)
kable(reg2)
```

term	estimate	std.error	statistic	p.value
(Intercept)	2.094943	0.3694586	5.670306	0
x	2.057392	0.0641297	32.081742	0

```r
stargazer(my_reg, type="html")
```

Dependent variable:

y

x

2.057***

(0.064)

Constant

2.095***

(0.369)

Observations

500

R2

0.674

Adjusted R2

0.673

Residual Std. Error

4.136 (df = 498)

F Statistic

1,029.238*** (df = 1; 498)

Note:

p<0.1; ***p<0.05;*** p<0.01

Chapter 7

Advanced Regression

7.1 Multivariate Regression

```r
set.seed(1)
x<-rnorm(500,10,2)
z<-runif(500,10,20)
y<-rnorm(500,2*x*z,2)

# generate categorical variables

# make a shape variable
shapes<-c("square","circle","triangle","rectangle","trapezoid")
shape<-sample(shapes,500,replace=TRUE) # sample 500 random draws with replacement from shapes

# make a region variable
regions<-c("north","south","east","west")
region<-sample(regions,500,replace=TRUE) # sample 500 random draws with replacement from regions

# make a dummy variable
yes<-sample(c(0,1),500,replace=TRUE)

# combine into dataframe called df

df<-data.frame(x=x,
               y=y,
               z=z,
               shape=factor(shape),
               region=factor(region),
               yes=yes)

# look at new df

head(df)
```

```
##           x        y        z     shape region yes
## 1  8.747092 264.8931 15.30809  triangle   west   1
## 2 10.367287 347.6574 16.84861    square  south   1
## 3  8.328743 227.9193 13.83283 trapezoid  north   1
## 4 13.190562 517.0824 19.54988  triangle  south   1
## 5 10.659016 235.8301 11.18357 trapezoid   west   0
## 6  8.359063 169.6481 10.39100  triangle   west   1
```

It is quite simply to simply add additional covariates to a regression. In the `lm` object, we add variables with +.

```
reg1<-lm(y~x+z, data=df)
summary(reg1)
```

```
##
## Call:
## lm(formula = y ~ x + z, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -47.969  -5.115  -0.013   5.756  42.078
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -295.7258     3.7549  -78.76   <2e-16 ***
## x             29.4336     0.2671  110.19   <2e-16 ***
## z             20.1389     0.1849  108.94   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 12.05 on 497 degrees of freedom
## Multiple R-squared:  0.9809, Adjusted R-squared:  0.9808
## F-statistic: 1.277e+04 on 2 and 497 DF,  p-value: < 2.2e-16
```

7.2 Dummy Variables

```
reg_d<-lm(y~yes, data=df)
summary(reg_d)
```

```
##
## Call:
## lm(formula = y ~ yes, data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -197.442  -66.094   -8.786   64.039  274.874
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  297.172      5.354  55.506   <2e-16 ***
## yes            1.936      7.809   0.248    0.804
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 87.16 on 498 degrees of freedom
## Multiple R-squared:  0.0001234,  Adjusted R-squared:  -0.001884
## F-statistic: 0.06148 on 1 and 498 DF,  p-value: 0.8043
```

The effect on y of going from "No" to "Yes" is 1.94.

If we wanted to make a dummy variable for an existing categorical variable

```
df$north<-ifelse(region=="north",1,0)
df$south<-ifelse(region=="south",1,0)
df$east<-ifelse(region=="east",1,0)
df$west<-ifelse(region=="west",1,0)

head(df)
```

```
##           x        y        z     shape region yes north south east west
## 1  8.747092 264.8931 15.30809  triangle   west   1     0     0    0    1
## 2 10.367287 347.6574 16.84861    square  south   1     0     1    0    0
## 3  8.328743 227.9193 13.83283 trapezoid  north   1     1     0    0    0
## 4 13.190562 517.0824 19.54988  triangle  south   1     0     1    0    0
## 5 10.659016 235.8301 11.18357 trapezoid   west   0     0     0    0    1
## 6  8.359063 169.6481 10.39100  triangle   west   1     0     0    0    1
```

Here is where a `for` loop also can come in handy:

```
for(i in unique(df$region)){
  region[i]<-ifelse(df$region==i,1,0)
}
```

```
## Warning in region[i] <- ifelse(df$region == i, 1, 0): number of items to
## replace is not a multiple of replacement length

## Warning in region[i] <- ifelse(df$region == i, 1, 0): number of items to
## replace is not a multiple of replacement length

## Warning in region[i] <- ifelse(df$region == i, 1, 0): number of items to
## replace is not a multiple of replacement length

## Warning in region[i] <- ifelse(df$region == i, 1, 0): number of items to
## replace is not a multiple of replacement length
```

```
head(df)
```

```
##           x        y        z     shape region yes north south east west
## 1  8.747092 264.8931 15.30809  triangle   west   1     0     0    0    1
## 2 10.367287 347.6574 16.84861    square  south   1     0     1    0    0
## 3  8.328743 227.9193 13.83283 trapezoid  north   1     1     0    0    0
## 4 13.190562 517.0824 19.54988  triangle  south   1     0     1    0    0
## 5 10.659016 235.8301 11.18357 trapezoid   west   0     0     0    0    1
## 6  8.359063 169.6481 10.39100  triangle   west   1     0     0    0    1
```

7.3 Polynomial Regression