Practical Data Analysis for Political Scientists

Brenton Kenkel

2017-01-24

Contents

Chapter 1

About This Book

This book contains the course notes for Brenton Kenkel's course Statistics for Political Research II (PSCI 8357 at Vanderbilt University). It covers the basics of statistical modeling and programming with linear models, along with applications in R.

This book is written in R Markdown and published via Bookdown on GitHub Pages. You can find the R Markdown source files at https://github.com/brentonk/pdaps.

Chapter 2

Principles of Programming

It may seem strange to begin a statistics class with two weeks on programming. It is strange. Here is why I have made this strange choice.

First, as a working social scientist, most of the time you spend on data analysis won't be on the *analysis* part. It'll be on obtaining and cleaning the data, to get it in a form that makes sense to analyze. Good programming skills will let you spend less time cleaning data and more time publishing papers.

Second, even if you don't want to develop good programming habits, journals are going to force you to. Every reputable political science journal requires that you provide replication scripts, and some of the best (e.g., *American Journal of Political Science*) have begun auditing the replication materials as a condition of publication. Better to learn The Right Way now when you have lots of time than to be forced to when you're writing a dissertation or on the market or teaching your own courses.

Third, while I feel embarrassed to invoke the cliché that is Big Data, that doesn't mean it's not a real thing. Political scientists have access to more data and more computing power than ever before. You can't collect, manage, clean, and analyze large quantities of data without understanding the basic principles of programming.

As Bowers (2011) puts it, "Data analysis is computer programming." By getting a PhD in political science,[1] by necessity you're going to become a computer programmer. The choice before you is whether to be a good one or a bad one.

Wilson et al. (2014) list eight "best practices for scientific computing." The first two encapsulate most of what you need to know:

1. Write programs for people, not computers.
2. Let the computer do the work.

[1] Or whatever other social science field.

2.1 Write Programs for People, Not Computers

The first two words here—*write programs*—are crucial. When you are doing analysis for a research project, you should be writing and running scripts, not typing commands into the R (or Stata) console. The console is ephemeral, but scripts are forever, at least if you save them.

Like the manuscripts you will write to describe your findings, your analysis scripts are a form of scientific communication. You wouldn't write a paper that is disorganized, riddled with grammatical errors, or incomprehensible to anyone besides yourself. Don't write your analysis scripts that way either.

Each script should be self-contained, ideally accomplishing one major task. Using an omnibus script that runs every bit of analysis is like writing a paper without paragraph breaks. A typical breakdown of scripts for a project of mine looks like:

- `0-download.r`: downloads the data
- `1-clean.r`: cleans the data
- `2-run.r`: runs the main analysis
- `3-figs.r`: generates figures

The exact structure varies depending on the nature of the project. Notice that the scripts are numbered in the order they should be run.

Within each script, write the code to make it as easy as possible for your reader to follow what you're doing. You should indent your code according to style conventions such as http://adv-r.had.co.nz/Style.html. Even better, use the `Code -> Reindent Lines` menu option in R Studio to automatically indent according to a sane style.

```
# Bad
my_results <- c(mean(variable),
quantile(variable,
probs = 0.25),
max(variable))

# Better
my_results <- c(mean(variable),
                quantile(variable,
                         probs = 0.25),
                max(variable))
```

Another way to make your code readable—one that, unfortunately, cannot be accomplished quite so algorithmically—is to add explanatory comments. The point of comments is not to document how the language works. The following comment is an extreme example of a useless comment.

```
# Take the square root of the errors and assign them to
# the output variable
output <- sqrt(errors)
```

A better use for the comment would be to explain *why* you're taking the square root of the errors, at least if your purpose in doing so would be unclear to a hypothetical reader of the code.

My basic heuristic for code readability is *If I got hit by a bus tomorrow, could one of my coauthors figure out what the hell I was doing and finish the paper?*

2.2 Let the Computer Do the Work

Computers are really good at structured, repetitive tasks. If you ever find yourself entering the same thing into the computer over and over again, you are Doing It Wrong. Your job as the human directing the computer is to figure out the structure that underlies the repeated task and to program the computer to do the repetition.

For example, imagine you have just run a large experiment and you want to estimate effects by subgroups.[2] Your respondents differ across four variables—party ID (R or D), gender (male or female), race (white or nonwhite), and education (college degree or not)—giving you 16 subgroups. You *could* copy and paste your code to estimate the treatment effect 16 times. But this is a bad idea for a few reasons.

- Copy-paste doesn't scale. Copy-paste is managable (albeit misguided) for 16 iterations, but probably not for 50 and definitely not for more than 100.
- Making changes becomes painful. Suppose you decide to change how you calculate the estimate. Now you have to go back and individually edit 16 chunks of code.
- Copy-paste is error-prone, and insidiously so. If you do the calculation wrong all 16 times, you'll probably notice. But what if you screwed up for just one or two cases? Are you *really* going to go through and check that you did everything right in each individual case?

We're going to look at the most basic ways to get the computer to repeat structured tasks—functions and control flow statements. To illustrate these, we will use a result you discussed in Stat I: the central limit theorem.

The central limit theorem concerns the *sampling distribution* of the sample mean,

[2] There could be statistical problems with this kind of analysis, at least if the subgroups were specified *post hoc*. See https://xkcd.com/882/ ("Significant"). We're going to leave this issue aside for now, but we'll return to it later when we discuss the statistical crisis in science.

$$\bar{X} = \frac{1}{N} \sum_{n=1}^{N} X_n,$$

where each X_n is independent and identically distributed with mean μ and variance σ^2. Loosely speaking, the CLT says that as N grows large, the sampling distribution of $\bar{X}$ becomes approximately normal with mean μ and variance σ^2/N.

Here's what we would need to do to see the CLT in practice. We'd want to take a bunch of samples, each of size N, and calculate the sample mean of each. Then we'd have a sample of sample means, and we could check to verify that they are approximately normally distributed with mean μ and variance σ^2/N. This is a structured, repetitive task—exactly the kind of thing that should be programmed. We'll try it out with a random variable from a Poisson distribution with $\lambda = 3$, which has mean $\mu = 3$ and variance $\sigma^2 = 3$.

First things first. We can use the `rpois` function to draw a random sample of N numbers from the Poisson distribution.

```
samp <- rpois(10, lambda = 3)
samp
```

```
##  [1] 2 3 8 3 5 4 3 4 2 2
```

To calculate the sample mean, we simply use the `mean` function.

```
mean(samp)
```

```
## [1] 3.6
```

We are interested in the distribution of the sample mean across many samples like this one. To begin, we will write a **function** that automates our core task—drawing a sample of N observations from Poisson(3) and calculating the sample mean. A function consists of a set of *arguments* (the inputs) and a *body* of code specifying which calculations to perform on the inputs to produce the output.

```
pois_mean <- function(n_obs) {
  samp <- rpois(n_obs, lambda = 3)
  ans <- mean(samp)
  return(ans)
}
```

This code creates a function called `pois_mean`. It has a single argument, called `n_obs`. It generates a random sample of `n_obs` draws from Poisson(3) and calculates its sample mean. It then `return`s the sample mean as the output.

Let's try calling this function a few times, each with a sample size of $N = 30$. Your output will differ slightly from what's printed here, since the function is generating random numbers.

```
pois_mean(n_obs = 30)
```

```
## [1] 3.0333
```

```
pois_mean(n_obs = 30)
```

```
## [1] 2.4667
```

```
pois_mean(n_obs = 30)
```

```
## [1] 2.9667
```

Remember that what we're interested in is the *sampling distribution* of the sample mean—the distribution of the sample mean across every possible sample of N observations. We can approximate this distribution by running `pois_mean` many times (e.g., 1000 or more). This would be infeasible via copy-paste. Instead, we will use a **for loop**.

```
# Set up a vector to store the output
n_replicates <- 1000
sampling_dist <- rep(NA, n_replicates)

for (i in 1:n_replicates) {
  sampling_dist[i] <- pois_mean(n_obs = 30)
}
```

Here's how the for loop works. We specified `i` as the name of the index variable, with values `1:n_replicates`. The for loop takes each value in the sequence, assigns it to the variable `i`, runs the given expression (in this case, assigning the output of `pois_mean` to the `i`'th element of `sampling_dist`), and then moves on to the next value in sequence, until it reaches the end.

Let's take a look at the results and compare them to our expectations.

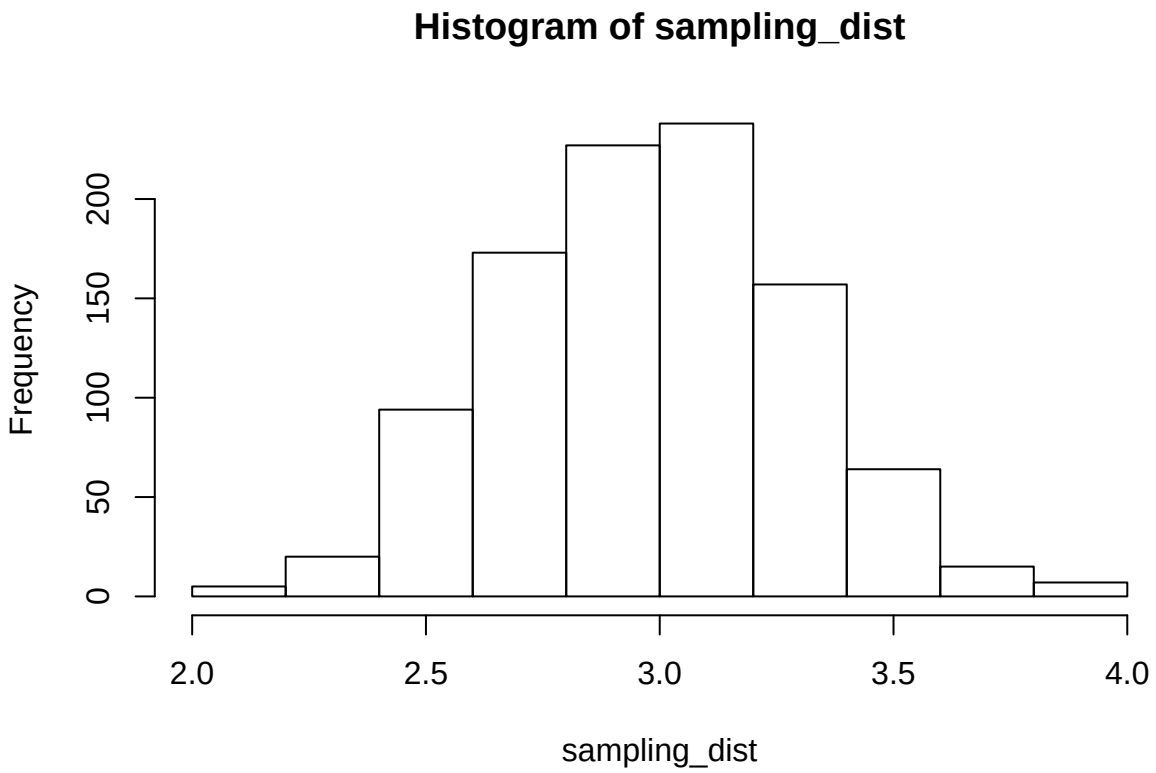
```
mean(sampling_dist)  # Expect 3
```

```
## [1] 2.9952
```

```
var(sampling_dist)  # Expect 1/10
```

```
## [1] 0.096944
```

```
hist(sampling_dist)  # Expect roughly normal
```



For loops are fun, but don't overuse them. Many simple operations are **vectorized** and don't require a loop. For example, suppose you want to take the square of a sequence of numbers. You could use a for loop …

```
input <- c(1, 3, 7, 29)
output <- rep(NA, length(input))

for (i in 1:length(input)) {
  output[i] <- input[i]^2
}

output
```

```
## [1]   1   9  49 841
```

… but it's faster (in terms of computational speed) and easier to just take advantage of vectorization:

```
input^2
```

```
## [1]   1   9  49 841
```

Another useful piece of control flow is **if/else statements**. These check a logical condition—an expression whose value is `TRUE` or `FALSE`—and run different code depending on the value of the expression. (You may want to catch up on the comparison operators: `==`, `>`, `>=`, `<`, `<=`, etc.)

Let's edit the `pois_mean` function to allow us to calculate the median instead of the mean. We'll add a second argument to the function, and implement the option using an if/else statement.

```
pois_mean <- function(n_obs, use_median = FALSE) {
  samp <- rpois(n_obs, lambda = 3)
  if (use_median) {
    ans <- median(samp)
  } else {
    ans <- mean(samp)
  }
  return(ans)
}
```

A couple of things to notice about the structure of the function. We use a comma to separate multiple function arguments. Also, we've specified `FALSE` as the *default* for the `use_median` argument. If we call the function without explicitly specifying a value for `use_median`, the function sets it to `FALSE`.

```
pois_mean(n_obs = 9)
```

```
## [1] 3.7778
```

```
pois_mean(n_obs = 9, use_median = TRUE)
```

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

```
pois_mean(n_obs = 9, use_median = FALSE)
```

```
## [1] 2.6667
```

There is a vectorized version of if/else statements called, naturally, the `ifelse` function. This function takes three arguments, each a vector of the same length: (1) a logical condition, (2) an output value if the condition is `TRUE`, (3) an output value if the condition is `FALSE`.

```
x <- 1:10
big_x <- x * 100
small_x <- x * -100

ifelse(x > 5, big_x, small_x)
```

```
##  [1] -100 -200 -300 -400 -500  600  700  800  900 1000
```

Functions, for loops, and if/else statements are just a few of the useful tools for programming in R.[3] But even these simple tools are enough to allow you to do much more at scale than you could with a copy-paste philosophy.

[3]Others include the `replicate` function, the `apply` family of functions (`sapply`, `lapply`, `tapply`, `mapply`, …), the **foreach** package, the **purrr** package, just to name a few of the most useful off the top of my head.

Chapter 3

Working with Data

Some material in this chapter is adapted from notes Matt DiLorenzo wrote for the Spring 2016 session of PSCI 8357.

Let me repeat something I said last week. In your careers as social scientists, starting with your dissertation research—if not earlier—you will probably spend more time collecting, merging, and cleaning data than you will on statistical analysis. So it's worth taking some time to learn how to do this well.

Best practices for data management can be summarized in a single sentence: *Record and document everything you do to the data.*

The first corollary of this principle is that raw data is sacrosanct. You should never edit raw data "in place". Once you download the raw data file, that file should never change.[1]

In almost any non-trivial analysis, the "final" data—the format you plug into your analysis—will differ significantly from the raw data. It may consist of information merged from multiple sources. The variables may have been transformed, aggregated, or otherwise altered. The unit of observation may even differ from the original source. You must document every one of these changes, so that another researcher working from the exact same raw data will end up with the exact same final data.

The most sensible way to achieve this level of reproducibility is to do all of your data merging and cleaning in a script. In other words, no going into Excel and mucking around manually. Like any other piece of your analysis, your pipeline from raw data to final data should follow the principles of programming that we discussed last week.

Luckily for you,[2] the **tidyverse** suite of R packages (including **dplyr**, **tidyr**, and others) makes it easy to script your "data pipeline". We'll begin by loading the package.

```
library("tidyverse")
```

[1] Even if it's data you collected yourself, that data should still have a "canonical" representation that never gets overwritten. See Leek (2015) for more on distributing your own data.

[2] But not for me, because these tools didn't exist when I was a PhD student. Also, get off my lawn!

3.1 Loading

The first step in working with data is to acquire some data. Depending on the nature of your research, you will be getting some or all of your data from sources available online. When you download data from online repositories, you should keep track of where you got it from. The best way to do so is—you guessed it—to script your data acquisition.

The R function `download.file()` is the easiest way to download files from URLs from within R. Just specify where you're getting the file from and where you want it to go. For the examples today, we'll use an "untidied" version of the World Development Indicators data from the World Bank that I've posted to my website.

```
download.file(url = "http://bkenkel.com/data/untidy-data.csv",
              destfile = "my-untidy-data.csv")
```

Once you've got the file stored locally, use the utilities from the **readr** package (part of **tidyverse**) to read it into R as a data frame.[3] We have a CSV file, so we will use `read_csv`. See `help(package = "readr")` for other possibilities.

```
untidy_data <- read_csv(file = "my-untidy-data.csv")
```

```
## Parsed with column specification:
## cols(
##   country = col_character(),
##   gdp.2005 = col_double(),
##   gdp.2006 = col_double(),
##   gdp.2007 = col_double(),
##   gdp.2008 = col_double(),
##   pop.2005 = col_double(),
##   pop.2006 = col_double(),
##   pop.2007 = col_double(),
##   pop.2008 = col_double(),
##   unemp.2005 = col_double(),
##   unemp.2006 = col_double(),
##   unemp.2007 = col_double(),
##   unemp.2008 = col_double()
## )
```

Remember that each column of a data frame might be a different type, or more formally *class*, of object. `read_csv` and its ilk try to guess the type of data each column contains: character, integer, decimal number ("double" in programming-speak), or something else. The readout

[3] More precisely, the **readr** functions produce output of class `"tbl_df"` (pronounced "tibble diff," I'm told), which are like data frames but better. See `help(package = "tibble")` for what can be done with `tbl_df`s.

above tells you what guesses it made. If it gets something wrong—say, reading a column as numbers that ought to be characters—you can use the `col_types` argument to set it straight.

FYI, you could also run `read_csv()` directly on a URL, as in:

```
read_csv("http://bkenkel.com/data/untidy-data.csv")
```

However, in analyses intended for publication, it's usually preferable to download and save the raw data. What's stored at a URL might change or disappear, and you'll need to have a hard copy of the raw data for replication purposes.

Now let's take a look at the data we've just loaded in.

```
head(untidy_data)
```

```
## # A tibble: 6 × 13
##   country gdp.2005 gdp.2006 gdp.2007 gdp.2008  pop.2005  pop.2006
##     <chr>    <dbl>    <dbl>    <dbl>    <dbl>     <dbl>     <dbl>
## 1      AD   3.8423   4.0184   4.0216   3.6759  0.081223  0.083373
## 2      AE 253.9655 278.9489 287.8318 297.0189  4.481976  5.171255
## 3      AF   9.7630  10.3052  11.7212  12.1445 24.399948 25.183615
## 4      AG   1.1190   1.2687   1.3892   1.3902  0.082565  0.083467
## 5      AL   9.2684   9.7718  10.3483  11.1275  3.011487  2.992547
## 6      AM   7.6678   8.6797   9.8731  10.5544  3.014917  3.002161
## # ... with 6 more variables: pop.2007 <dbl>, pop.2008 <dbl>,
## #   unemp.2005 <dbl>, unemp.2006 <dbl>, unemp.2007 <dbl>, unemp.2008 <dbl>
```

We have a `country` variable giving country abbreviations. The other variables are numerical values: the country's GDP in 2005, 2006, 2007, and 2008; then the same for population and unemployment. Let's get this into a format we could use for analysis.

3.2 Tidying

Wickham (2014) outlines three qualities that make data "tidy":

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

For one thing, this means that whether a dataset is tidy or not depends—at least in part (some data collections are messy from any angle)—on the purpose it's being analyzed for.

Each row of `untidy_data` is a country. In observational studies in comparative politics and international relations, more commonly the unit of observation is the country-year.[4] How can we take `untidy_data` and easily make it into country-year data?

We'll use the **tidyr** package (again, part of **tidyverse**) to clean up this data. The biggest problem right now is that each column, besides the country identifier, really encodes two pieces of information: the year of observation and the variable being observed. To deal with this, we'll have to first transform the data from one untidy format to another. We're going to use the `gather()` function to make each row a country-year-variable.

What `gather()` does is make a row for each entry from a set of columns. It's probably easiest to understand it by seeing it in practice:

```
long_data <- gather(untidy_data,
                    key = variable,
                    value = number,
                    gdp.2005:unemp.2008)
head(long_data)
```

```
## # A tibble: 6 × 3
##   country variable   number
##     <chr>    <chr>    <dbl>
## 1      AD gdp.2005   3.8423
## 2      AE gdp.2005 253.9655
## 3      AF gdp.2005   9.7630
## 4      AG gdp.2005   1.1190
## 5      AL gdp.2005   9.2684
## 6      AM gdp.2005   7.6678
```

With the first argument, we told `gather()` to use the `untidy_data` data frame. With the last argument, we told it the set of columns to "gather" together into a single column. The `key` column specifies the name of the variable to store the "key" (original column name) in, and the `value` column specifies the name of the variable to store the associated value. For example, the second row of `long_data` encodes what we previously saw as the `gdp.2005` column of `untidy_data`.

Now we have a new problem, which is that `variable` encodes two pieces of information: the variable and the year of its observation. **tidyr** provides the `separate()` function to solve that, splitting a single variable into two.

```
long_data <- separate(long_data,
                      col = variable,
                      into = c("var", "year"))
head(long_data)
```

[4] Insert lame joke about how Americanists haven't heard of other countries. But, seriously, if you're confused because you haven't heard of other countries, just think of "state-years".

```
## # A tibble: 6 × 4
##   country   var  year   number
##     <chr> <chr> <chr>    <dbl>
## 1      AD   gdp  2005   3.8423
## 2      AE   gdp  2005 253.9655
## 3      AF   gdp  2005   9.7630
## 4      AG   gdp  2005   1.1190
## 5      AL   gdp  2005   9.2684
## 6      AM   gdp  2005   7.6678
```

So now we have country-year-variable data, with the year and variable conveniently stored in different columns. To turn this into country-year data, we can use the `spread()` function, which is like the inverse of `gather()`. `spread()` takes a key column and a value column, and turns each different key into a column of its own.

```
clean_data <- spread(long_data,
                     key = var,
                     value = number)
head(clean_data)
```

```
## # A tibble: 6 × 5
##   country  year      gdp      pop unemp
##     <chr> <chr>    <dbl>    <dbl> <dbl>
## 1      AD  2005   3.8423 0.081223    NA
## 2      AD  2006   4.0184 0.083373    NA
## 3      AD  2007   4.0216 0.084878    NA
## 4      AD  2008   3.6759 0.085616    NA
## 5      AE  2005 253.9655 4.481976   3.1
## 6      AE  2006 278.9489 5.171255   3.3
```

When using `spread()` on data that you didn't previously `gather()`, be sure to set the `fill` argument to tell it how to fill in empty cells. A simple example:

```
test_data
```

```
## # A tibble: 3 × 3
##        id     k     v
##     <chr> <chr> <dbl>
## 1 brenton     a    10
## 2 brenton     b    20
## 3 patrick     b     5
```

```
spread(test_data, key = k, value = v)
```

```
## # A tibble: 2 × 3
##        id     a     b
## *   <chr> <dbl> <dbl>
## 1 brenton    10    20
## 2 patrick    NA     5
```

```
spread(test_data, key = k, value = v, fill = 100)
```

```
## # A tibble: 2 × 3
##        id     a     b
## *   <chr> <dbl> <dbl>
## 1 brenton    10    20
## 2 patrick   100     5
```

One more important note on **tidyverse** semantics. It includes a fabulous feature called the *pipe*, `%>%`, which makes it easy to string together a truly mind-boggling number of commands.

In pipe syntax, `x %>% f()` is equivalent to `f(x)`. That seems like a wasteful and confusing way to write `f(x)`, and it is. But if you want to string together a bunch of commands, it's much easier to comprehend

```
x %>%
  f() %>%
  g() %>%
  h() %>%
  i()
```

than `i(h(g(f(x))))`.

You can pass function arguments using the pipe too. For example, `f(x, bear = "moose")` is equivalent to `x %>% f(bear = "moose")`.

The key thing about the **tidyverse** functions is that each of them takes a data frame as its first argument, and returns a data frame as its output. This makes them highly amenable to piping. For example, we can combine all three steps of our tidying above with a single command, thanks to the pipe:[5]

[5] If you are reading the PDF copy of these notes (i.e., the ones I hand out in class), the line breaks are eliminated, making the piped commands rather hard to read. I am working on fixing this. For now, you may find the online notes at http://bkenkel.com/pdaps easier to follow.

```
untidy_data %>%
  gather(key = variable,
         value = number,
         gdp.2005:unemp.2008) %>%
  separate(col = variable,
           into = c("var", "year")) %>%
  spread(key = var,
         value = number)
```

```
## # A tibble: 860 × 5
##    country  year      gdp      pop unemp
## *    <chr> <chr>    <dbl>    <dbl> <dbl>
## 1       AD  2005   3.8423 0.081223    NA
## 2       AD  2006   4.0184 0.083373    NA
## 3       AD  2007   4.0216 0.084878    NA
## 4       AD  2008   3.6759 0.085616    NA
## 5       AE  2005 253.9655 4.481976   3.1
## # ... with 855 more rows
```

Without the pipe, if we wanted to run all those commands together, we would have to write:

```
spread(separate(gather(untidy_data,
                       key = variable,
                       value = number,
                       gdp.2005:unemp.2008),
                col = variable,
                into = c("var", "year")),
       key = var,
       value = number)
```

Sad!

3.3 Transforming and Aggregating

Tidying the data usually isn't the end of the process. If you want to perform further calculations on the raw, that's where the tools in **dplyr** (part of, you guessed it, the **tidyverse**) come in.

Perhaps the simplest **dplyr** function (or "verb", as the R hipsters would say) is `rename()`, which lets you rename columns.

```
clean_data %>%
  rename(gross_domestic_product = gdp)
```

```
## # A tibble: 860 × 5
##   country  year gross_domestic_product      pop unemp
## *   <chr> <chr>                  <dbl>    <dbl> <dbl>
## 1      AD  2005                 3.8423 0.081223    NA
## 2      AD  2006                 4.0184 0.083373    NA
## 3      AD  2007                 4.0216 0.084878    NA
## 4      AD  2008                 3.6759 0.085616    NA
## 5      AE  2005               253.9655 4.481976   3.1
## # ... with 855 more rows
```

The **dplyr** functions, like the vast majority of R functions, do not modify their inputs. In other words, running `rename()` on `clean_data` will return a renamed copy of `clean_data`, but won't overwrite the original.

```
clean_data
```

```
## # A tibble: 860 × 5
##   country  year      gdp      pop unemp
## *   <chr> <chr>    <dbl>    <dbl> <dbl>
## 1      AD  2005   3.8423 0.081223    NA
## 2      AD  2006   4.0184 0.083373    NA
## 3      AD  2007   4.0216 0.084878    NA
## 4      AD  2008   3.6759 0.085616    NA
## 5      AE  2005 253.9655 4.481976   3.1
## # ... with 855 more rows
```

If you wanted to make the change stick, you would have to run:

```
clean_data <- clean_data %>%
  rename(gross_domestic_product = gdp)
```

`select()` lets you keep a couple of columns and drop all the others. Or vice versa if you use minus signs.

```
clean_data %>%
  select(country, gdp)
```

```
## # A tibble: 860 × 2
##   country      gdp
## *   <chr>    <dbl>
## 1      AD   3.8423
## 2      AD   4.0184
## 3      AD   4.0216
## 4      AD   3.6759
## 5      AE 253.9655
## # ... with 855 more rows
```

```
clean_data %>%
  select(-pop)
```

```
## # A tibble: 860 × 4
##   country  year      gdp unemp
## *   <chr> <chr>    <dbl> <dbl>
## 1      AD  2005   3.8423    NA
## 2      AD  2006   4.0184    NA
## 3      AD  2007   4.0216    NA
## 4      AD  2008   3.6759    NA
## 5      AE  2005 253.9655   3.1
## # ... with 855 more rows
```

`mutate()` lets you create new variables that are transformations of old ones.

```
clean_data %>%
  mutate(gdppc = gdp / pop,
         log_gdppc = log(gdppc))
```

```
## # A tibble: 860 × 7
##   country  year      gdp      pop unemp  gdppc log_gdppc
##     <chr> <chr>    <dbl>    <dbl> <dbl>  <dbl>     <dbl>
## 1      AD  2005   3.8423 0.081223    NA 47.305    3.8566
## 2      AD  2006   4.0184 0.083373    NA 48.198    3.8753
## 3      AD  2007   4.0216 0.084878    NA 47.381    3.8582
## 4      AD  2008   3.6759 0.085616    NA 42.935    3.7597
## 5      AE  2005 253.9655 4.481976   3.1 56.664    4.0371
## # ... with 855 more rows
```

`filter()` cuts down the data according to the logical condition(s) you specify.

```
clean_data %>%
  filter(year == 2006)
```

```
## # A tibble: 215 × 5
##    country  year      gdp       pop unemp
##      <chr> <chr>    <dbl>     <dbl> <dbl>
## 1       AD  2006   4.0184  0.083373    NA
## 2       AE  2006 278.9489  5.171255   3.3
## 3       AF  2006  10.3052 25.183615   8.8
## 4       AG  2006   1.2687  0.083467    NA
## 5       AL  2006   9.7718  2.992547  12.4
## # ... with 210 more rows
```

`summarise()` calculates summaries of the data. For example, let's find the maximum unemployment rate.

```
clean_data %>%
  summarise(max_unemp = max(unemp, na.rm = TRUE))
```

```
## # A tibble: 1 × 1
##   max_unemp
##       <dbl>
## 1      37.6
```

This seems sort of useless, until you combine it with the `group_by()` function. If you group the data before `summarise`-ing it, you'll calculate a separate summary for each group. For example, let's calculate the maximum unemployment rate for each year in the data.

```
clean_data %>%
  group_by(year) %>%
  summarise(max_unemp = max(unemp, na.rm = TRUE))
```

```
## # A tibble: 4 × 2
##    year max_unemp
##   <chr>     <dbl>
## 1  2005      37.3
## 2  2006      36.0
## 3  2007      34.9
## 4  2008      37.6
```

`summarise()` produces a "smaller" data frame than the input—one row per group. If you want to do something similar, but preserving the structure of the original data, use `mutate` in combination with `group_by`.

```
clean_data %>%
  group_by(year) %>%
  mutate(max_unemp = max(unemp, na.rm = TRUE),
         unemp_over_max = unemp / max_unemp) %>%
  select(country, year, contains("unemp"))
```

```
## Source: local data frame [860 x 5]
## Groups: year [4]
##
##   country  year unemp max_unemp unemp_over_max
##     <chr> <chr> <dbl>     <dbl>          <dbl>
## 1      AD  2005    NA      37.3             NA
## 2      AD  2006    NA      36.0             NA
## 3      AD  2007    NA      34.9             NA
## 4      AD  2008    NA      37.6             NA
## 5      AE  2005   3.1      37.3        0.08311
## # ... with 855 more rows
```

This gives us back the original data, but with a `max_unemp` variable recording the highest unemployment level that year. We can then calculate each individual country's unemployment as a percentage of the maximum. Whether grouped `mutate` or `summarise` is better depends, of course, on the purpose and structure of your analysis.

Notice how I selected all of the unemployment-related columns with `contains("unemp")`. See `?select_helpers` for a full list of helpful functions like this for `select`-ing variables.

3.4 Merging

Only rarely will you be lucky enough to draw all your data from a single source. More often, you'll be merging together data from multiple sources.

The key to merging data from separate tables is to have consistent identifiers across tables. For example, if you run an experiment, you might have demographic data on each subject in one table, and each subject's response to each treatment in another table. Naturally, you'll want to have a subject identifier that "links" the records across tables, as in the following hypothetical example.

```
subject_data
```

```
## # A tibble: 3 × 4
##      id gender loves_bernie does_yoga
##   <dbl>  <chr>        <chr>     <chr>
## 1  1001   male          yes        no
```

```
## 2  1002 female           no       yes
## 3  1003   male           no        no
```

```
subject_response_data
```

```
## # A tibble: 6 × 3
##      id treatment response
##   <dbl>     <chr>    <chr>
## 1  1001 read_book      sad
## 2  1001  watch_tv      sad
## 3  1002 read_book    happy
## 4  1002  watch_tv      sad
## 5  1003 read_book      sad
## 6  1003  watch_tv    happy
```

Let's practice merging data with our cleaned-up country-year data. We'll take two datasets from my website: a country-level dataset with latitudes and longitudes, and a country-year–level dataset with inflation over time.

```
latlong_data <- read_csv("http://bkenkel.com/data/latlong.csv")
latlong_data
```

```
## # A tibble: 245 × 3
##   country latitude longitude
##     <chr>    <dbl>     <dbl>
## 1      AD   42.546    1.6016
## 2      AE   23.424   53.8478
## 3      AF   33.939   67.7100
## 4      AG   17.061  -61.7964
## 5      AI   18.221  -63.0686
## # ... with 240 more rows
```

```
inflation_data <- read_csv("http://bkenkel.com/data/inflation.csv")
inflation_data
```

```
## # A tibble: 1,070 × 3
##   country  year inflation
##     <chr> <int>     <dbl>
## 1      AD  2004        NA
## 2      AD  2005        NA
## 3      AD  2006        NA
## 4      AD  2007        NA
## 5      AD  2008        NA
## # ... with 1,065 more rows
```

For your convenience, both of these datasets use the same two-letter country naming scheme as the original data. Unfortunately, out in the real world, data from different sources often use incommensurate naming schemes. Converting from one naming scheme to another is part of the data cleaning process, and it requires careful attention.

dplyr contains various `_join()` functions for merging. Each of these take as arguments the two data frames to merge, plus the names of the identifier variables to merge them on. The one I use most often is `left_join()`, which keeps every row from the first ("left") data frame and merges in the columns from the second ("right") data frame.

For example, let's merge the latitude and longitude data for each country into `clean_data`.

```
left_join(clean_data,
          latlong_data,
          by = "country")
```

```
## # A tibble: 860 × 7
##   country  year      gdp      pop unemp latitude longitude
##     <chr> <chr>    <dbl>    <dbl> <dbl>    <dbl>     <dbl>
## 1      AD  2005   3.8423 0.081223    NA   42.546    1.6016
## 2      AD  2006   4.0184 0.083373    NA   42.546    1.6016
## 3      AD  2007   4.0216 0.084878    NA   42.546    1.6016
## 4      AD  2008   3.6759 0.085616    NA   42.546    1.6016
## 5      AE  2005 253.9655 4.481976   3.1   23.424   53.8478
## # ... with 855 more rows
```

Since `latlong_data` is country-level, the value is the same for each year. So the merged data contains redundant information. This is one reason to store data observed at different levels in different tables—with redundant observations, it is easier to make errors yet harder to catch them and fix them.

We can also merge data when the identifier is stored across multiple columns, as in the case of our country-year data. But first, a technical note.[6] You might notice that the `year` column of `clean_data` is labeled `<chr>`, as in character data. Yet the `year` column of `inflation_data` is labeled `<int>`, as in integer data. We can check that by running `class()` on each respective column.

```
class(clean_data$year)
```

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

[6] This won't be the case if you got `clean_data` by loading it in directly from `clean-data.csv` on my website, since `read_csv()` will have correctly encoded `year` as an integer.

```
class(inflation_data$year)
```

```
## [1] "integer"
```

From R's perspective, the character string `"1999"` is a very different thing than the integer number `1999`. Therefore, if we try to merge `clean_data` and `inflation_data` on the `year` variable, it will throw an error.

```
left_join(clean_data,
          inflation_data,
          by = c("country", "year"))
```

```
## Error in eval(substitute(expr), envir, enclos): Can't join on 'year' x 'year' because of i
```

To fix this, let's use `mutate()` to convert the `year` column of `clean_data` to an integer. We probably should have done this in the first place—after all, having the year encoded as a character string would have thrown off plotting functions, statistical functions, or anything else where it would be more natural to treat the year like a number.

```
clean_data <- mutate(clean_data,
                     year = as.integer(year))
clean_data
```

```
## # A tibble: 860 × 5
##   country  year      gdp      pop unemp
##     <chr> <int>    <dbl>    <dbl> <dbl>
## 1      AD  2005   3.8423 0.081223    NA
## 2      AD  2006   4.0184 0.083373    NA
## 3      AD  2007   4.0216 0.084878    NA
## 4      AD  2008   3.6759 0.085616    NA
## 5      AE  2005 253.9655 4.481976   3.1
## # ... with 855 more rows
```

Looks the same as before, except with an important difference: `year` is now labeled `<int>`.

Now we can merge the two datasets together without issue. Notice how we use a vector in the `by` argument to specify multiple columns to merge on.

```
left_join(clean_data,
          inflation_data,
          by = c("country", "year"))
```

```
## # A tibble: 860 × 6
##   country  year      gdp      pop unemp inflation
##     <chr> <int>    <dbl>    <dbl> <dbl>     <dbl>
## 1      AD  2005   3.8423 0.081223    NA        NA
## 2      AD  2006   4.0184 0.083373    NA        NA
## 3      AD  2007   4.0216 0.084878    NA        NA
## 4      AD  2008   3.6759 0.085616    NA        NA
## 5      AE  2005 253.9655 4.481976   3.1        NA
## # ... with 855 more rows
```

You might remember that `inflation_data` contained some country-years not included in the original data (namely, observations from 2004). If we want the merged data to use the observations from `inflation_data` rather than `clean_data`, we can use the `right_join()` function.

```
right_join(clean_data,
           inflation_data,
           by = c("country", "year"))
```

```
## # A tibble: 1,070 × 6
##   country  year    gdp      pop unemp inflation
##     <chr> <int>  <dbl>    <dbl> <dbl>     <dbl>
## 1      AD  2004     NA       NA    NA        NA
## 2      AD  2005 3.8423 0.081223    NA        NA
## 3      AD  2006 4.0184 0.083373    NA        NA
## 4      AD  2007 4.0216 0.084878    NA        NA
## 5      AD  2008 3.6759 0.085616    NA        NA
## # ... with 1,065 more rows
```

One last common issue in merging is that the identifier variables have different names in the two datasets. If it's inconvenient or infeasible to correct this by renaming the columns in one or the other, you can specify the `by` argument as in the following example.

```
inflation_data <- rename(inflation_data,
                         the_country = country,
                         the_year = year)
inflation_data
```

```
## # A tibble: 1,070 × 3
##   the_country the_year inflation
##         <chr>    <int>     <dbl>
## 1          AD     2004        NA
## 2          AD     2005        NA
```

```
## 3          AD     2006        NA
## 4          AD     2007        NA
## 5          AD     2008        NA
## # ... with 1,065 more rows
```

```
left_join(clean_data,
          inflation_data,
          by = c("country" = "the_country", "year" = "the_year"))
```

```
## # A tibble: 860 × 6
##   country  year      gdp      pop unemp inflation
##     <chr> <int>    <dbl>    <dbl> <dbl>     <dbl>
## 1      AD  2005   3.8423 0.081223    NA        NA
## 2      AD  2006   4.0184 0.083373    NA        NA
## 3      AD  2007   4.0216 0.084878    NA        NA
## 4      AD  2008   3.6759 0.085616    NA        NA
## 5      AE  2005 253.9655 4.481976   3.1        NA
## # ... with 855 more rows
```

3.5 Appendix: Creating the Example Data

I used the same tools this chapter introduces to create the untidy data. I may as well include the code to do it, in case it helps further illustrate how to use the **tidyverse** tools (and, as a bonus, the **WDI** package for downloading World Development Indicators data).

First I load the necessary packages.

```
library("tidyverse")
library("WDI")
library("countrycode")
library("stringr")
```

Next, I download the relevant WDI data. I used the `WDIsearch()` function to locate the appropriate indicator names.

```
dat_raw <- WDI(country = "all",
               indicator = c("NY.GDP.MKTP.KD",  # GDP in 2000 USD
                             "SP.POP.TOTL",     # Total population
                             "SL.UEM.TOTL.ZS"), # Unemployment rate
               start = 2005,
               end = 2008)

head(dat_raw)
```

```
##   iso2c    country year NY.GDP.MKTP.KD SP.POP.TOTL SL.UEM.TOTL.ZS
## 1    1A Arab World 2005     1.6428e+12   313430911        12.1402
## 2    1A Arab World 2006     1.7629e+12   320906736        11.3296
## 3    1A Arab World 2007     1.8625e+12   328766559        10.8961
## 4    1A Arab World 2008     1.9799e+12   336886468        10.5060
## 5    1W      World 2005     5.7703e+13  6513959904         6.1593
## 6    1W      World 2006     6.0229e+13  6594722462         5.9000
```

I want to get rid of the aggregates, like the "Arab World" and "World" we see here. As a rough tack at that, I'm going to exclude those so-called countries whose ISO codes don't appear in the **countrycode** package data.[7]

```
dat_countries <- dat_raw %>%
  filter(iso2c %in% countrycode_data$iso2c)
```

Let's check on which countries are left. (I cut it down to max six characters per country name for printing purposes.)

```
dat_countries$country %>%
  unique() %>%
  str_sub(start = 1, end = 6)
```

```
##   [1] "Andorr" "United" "Afghan" "Antigu" "Albani" "Armeni" "Angola"
##   [8] "Argent" "Americ" "Austri" "Austra" "Aruba"  "Azerba" "Bosnia"
##  [15] "Barbad" "Bangla" "Belgiu" "Burkin" "Bulgar" "Bahrai" "Burund"
##  [22] "Benin"  "Bermud" "Brunei" "Bolivi" "Brazil" "Bahama" "Bhutan"
##  [29] "Botswa" "Belaru" "Belize" "Canada" "Congo," "Centra" "Congo,"
##  [36] "Switze" "Cote d" "Chile"  "Camero" "China"  "Colomb" "Costa "
##  [43] "Cuba"   "Cabo V" "Curaca" "Cyprus" "Czech " "German" "Djibou"
##  [50] "Denmar" "Domini" "Domini" "Algeri" "Ecuado" "Estoni" "Egypt,"
##  [57] "Eritre" "Spain"  "Ethiop" "Finlan" "Fiji"   "Micron" "Faroe "
##  [64] "France" "Gabon"  "United" "Grenad" "Georgi" "Ghana"  "Gibral"
##  [71] "Greenl" "Gambia" "Guinea" "Equato" "Greece" "Guatem" "Guam"
##  [78] "Guinea" "Guyana" "Hong K" "Hondur" "Croati" "Haiti"  "Hungar"
##  [85] "Indone" "Irelan" "Israel" "Isle o" "India"  "Iraq"   "Iran, "
##  [92] "Icelan" "Italy"  "Jamaic" "Jordan" "Japan"  "Kenya"  "Kyrgyz"
##  [99] "Cambod" "Kiriba" "Comoro" "St. Ki" "Korea," "Korea," "Kuwait"
## [106] "Cayman" "Kazakh" "Lao PD" "Lebano" "St. Lu" "Liecht" "Sri La"
## [113] "Liberi" "Lesoth" "Lithua" "Luxemb" "Latvia" "Libya"  "Morocc"
## [120] "Monaco" "Moldov" "Monten" "St. Ma" "Madaga" "Marsha" "Macedo"
## [127] "Mali"   "Myanma" "Mongol" "Macao " "Northe" "Maurit" "Malta"
```

[7] **countrycode** is a very useful, albeit imperfect, package for converting between different country naming/coding schemes.

```
## [134] "Maurit" "Maldiv" "Malawi" "Mexico" "Malays" "Mozamb" "Namibi"
## [141] "New Ca" "Niger"  "Nigeri" "Nicara" "Nether" "Norway" "Nepal"
## [148] "Nauru"  "New Ze" "Oman"   "Panama" "Peru"   "French" "Papua "
## [155] "Philip" "Pakist" "Poland" "Puerto" "West B" "Portug" "Palau"
## [162] "Paragu" "Qatar"  "Romani" "Serbia" "Russia" "Rwanda" "Saudi "
## [169] "Solomo" "Seyche" "Sudan"  "Sweden" "Singap" "Sloven" "Slovak"
## [176] "Sierra" "San Ma" "Senega" "Somali" "Surina" "South " "Sao To"
## [183] "El Sal" "Sint M" "Syrian" "Swazil" "Turks " "Chad"   "Togo"
## [190] "Thaila" "Tajiki" "Timor-" "Turkme" "Tunisi" "Tonga"  "Turkey"
## [197] "Trinid" "Tuvalu" "Tanzan" "Ukrain" "Uganda" "United" "Urugua"
## [204] "Uzbeki" "St. Vi" "Venezu" "Britis" "Virgin" "Vietna" "Vanuat"
## [211] "Samoa"  "Yemen," "South " "Zambia" "Zimbab"
```

With that out of the way, there's still some cleaning up to do. The magnitudes of GDP and population are too large, and the variable names are impenetrable. Also, the `country` variable, while helpful, is redundant now that we're satisfied with the list of countries remaining.

```
dat_countries <- dat_countries %>%
  select(-country) %>%
  rename(gdp = NY.GDP.MKTP.KD,
         pop = SP.POP.TOTL,
         unemp = SL.UEM.TOTL.ZS,
         country = iso2c) %>%
  mutate(gdp = gdp / 1e9,
         pop = pop / 1e6)

head(dat_countries)
```

```
##   country year      gdp      pop unemp
## 1      AD 2005   3.8423 0.081223    NA
## 2      AD 2006   4.0184 0.083373    NA
## 3      AD 2007   4.0216 0.084878    NA
## 4      AD 2008   3.6759 0.085616    NA
## 5      AE 2005 253.9655 4.481976   3.1
## 6      AE 2006 278.9489 5.171255   3.3
```

Now I convert the data to "long" format.

```
dat_countries_long <- dat_countries %>%
  gather(key = variable,
         value = value,
         gdp:unemp)

head(dat_countries_long)
```

```
##   country year variable    value
## 1      AD 2005      gdp   3.8423
## 2      AD 2006      gdp   4.0184
## 3      AD 2007      gdp   4.0216
## 4      AD 2008      gdp   3.6759
## 5      AE 2005      gdp 253.9655
## 6      AE 2006      gdp 278.9489
```

I then smush `variable` and `year` into a single column, and drop the individual components.

```
dat_countries_long <- dat_countries_long %>%
  mutate(var_year = paste(variable, year, sep = ".")) %>%
  select(-variable, -year)

head(dat_countries_long)
```

```
##   country    value var_year
## 1      AD   3.8423 gdp.2005
## 2      AD   4.0184 gdp.2006
## 3      AD   4.0216 gdp.2007
## 4      AD   3.6759 gdp.2008
## 5      AE 253.9655 gdp.2005
## 6      AE 278.9489 gdp.2006
```

Finally, I "widen" the data, so that each `var_year` is a column of its own.

```
dat_countries_wide <- dat_countries_long %>%
  spread(key = var_year, value = value)

head(dat_countries_wide)
```

```
##   country gdp.2005 gdp.2006 gdp.2007 gdp.2008  pop.2005  pop.2006
## 1      AD   3.8423   4.0184   4.0216   3.6759  0.081223  0.083373
## 2      AE 253.9655 278.9489 287.8318 297.0189  4.481976  5.171255
## 3      AF   9.7630  10.3052  11.7212  12.1445 24.399948 25.183615
## 4      AG   1.1190   1.2687   1.3892   1.3902  0.082565  0.083467
## 5      AL   9.2684   9.7718  10.3483  11.1275  3.011487  2.992547
## 6      AM   7.6678   8.6797   9.8731  10.5544  3.014917  3.002161
##    pop.2007  pop.2008 unemp.2005 unemp.2006 unemp.2007 unemp.2008
## 1  0.084878  0.085616         NA         NA         NA         NA
## 2  6.010100  6.900142        3.1        3.3        3.4        4.0
## 3 25.877544 26.528741        8.5        8.8        8.4        8.9
## 4  0.084397  0.085350         NA         NA         NA         NA
## 5  2.970017  2.947314       12.5       12.4       13.5       13.0
## 6  2.988117  2.975029       27.8       28.6       28.4       16.4
```

Now we have some ugly data. I save the output to upload to my website.

```
write_csv(dat_countries_wide, path = "untidy-data.csv")
```

And here's how I made the second country-year dataset used in the merging section. The country dataset with latitudes and longitudes is from https://developers.google.com/public-data/docs/canonical/countries_csv.

```
dat_2 <-
  WDI(country = "all",
      indicator = "FP.CPI.TOTL.ZG",
      start = 2004,
      end = 2008) %>%
  as_data_frame() %>%
  select(country = iso2c,
         year,
         inflation = FP.CPI.TOTL.ZG) %>%
  mutate(year = as.integer(year)) %>%
  filter(country %in% clean_data$country) %>%
  arrange(country, year)

write_csv(dat_2, path = "inflation.csv")
```

Chapter 4

Data Visualization

Visualization is most important at the very beginning and the very end of the data analysis process. In the beginning, when you've just gotten your data together, visualization is perhaps the easiest tool to explore each variable and learn about the relationships among them. And when your analysis is almost complete, you will (usually) use visualizations to communicate your findings to your audience.

We only have time to scratch the surface of data visualization. This chapter will cover the plotting techniques I find most useful for exploratory and descriptive data analysis. We will talk about graphical techniques for presenting the results of regression analyses later in the class—once we've, you know, learned something about regression.

4.1 Basic Plots

We will use the **ggplot2** package, which is part of—I'm as tired of it as you are—the **tidyverse**.

```
library("tidyverse")
```

For the examples today, we'll be using a dataset with statistics about the fifty U.S. states in 1977,[1] which is posted on my website.

```
state_data <- read_csv("http://bkenkel.com/data/state-data.csv")
state_data
```

```
## # A tibble: 50 × 12
##        State Abbrev Region Population Income Illiteracy LifeExp Murder
##        <chr>  <chr>  <chr>      <dbl>  <dbl>      <dbl>   <dbl>  <dbl>
```

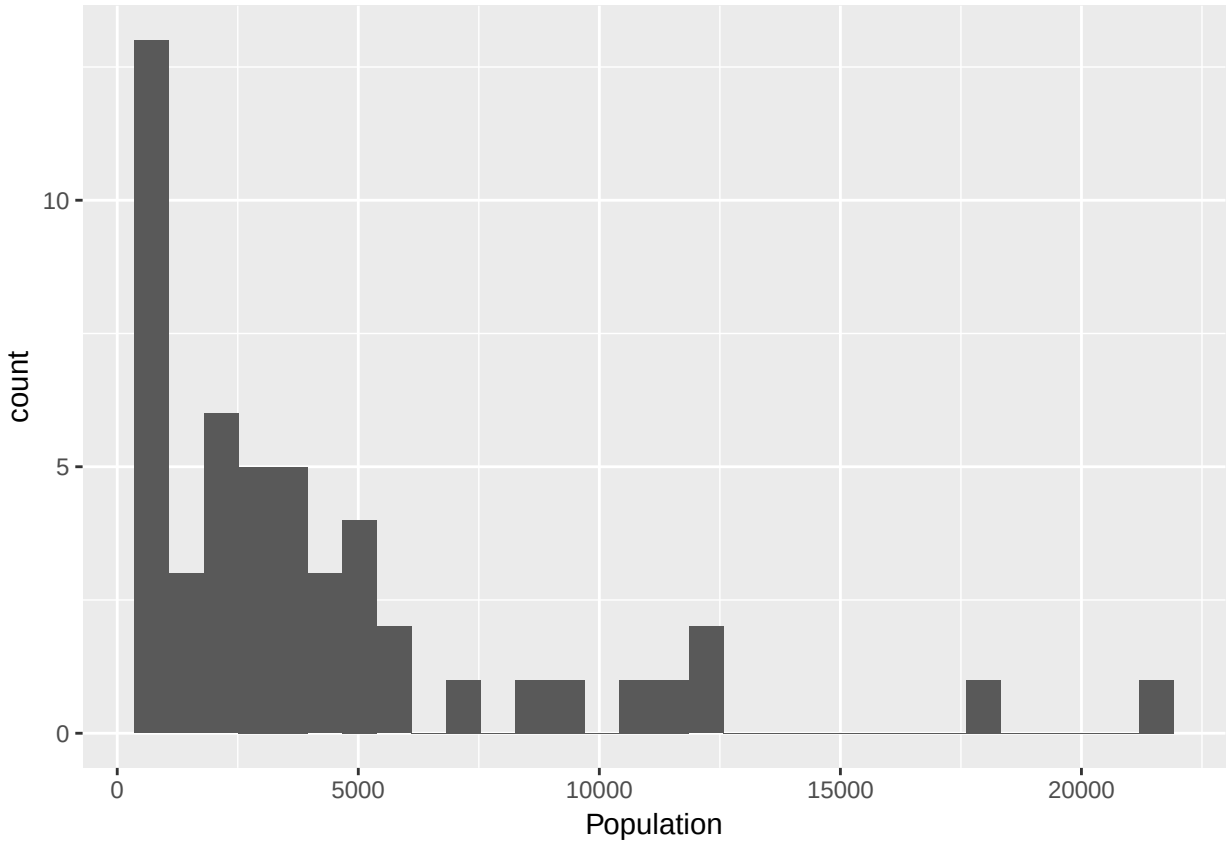
[1] Why 1977? Because it was easily available. See the appendix to this chapter.

```
## 1    Alabama     AL  South       3615   3624        2.1   69.05   15.1
## 2     Alaska     AK   West        365   6315        1.5   69.31   11.3
## 3    Arizona     AZ   West       2212   4530        1.8   70.55    7.8
## 4   Arkansas     AR  South       2110   3378        1.9   70.66   10.1
## 5 California     CA   West      21198   5114        1.1   71.71   10.3
## # ... with 45 more rows, and 4 more variables: HSGrad <dbl>, Frost <dbl>,
## #   Area <dbl>, IncomeGroup <chr>
```

When I obtain data, I start by looking at the univariate distribution of each variable via a histogram. The following code creates a histogram in ggplot.

```
ggplot(state_data, aes(x = Population)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



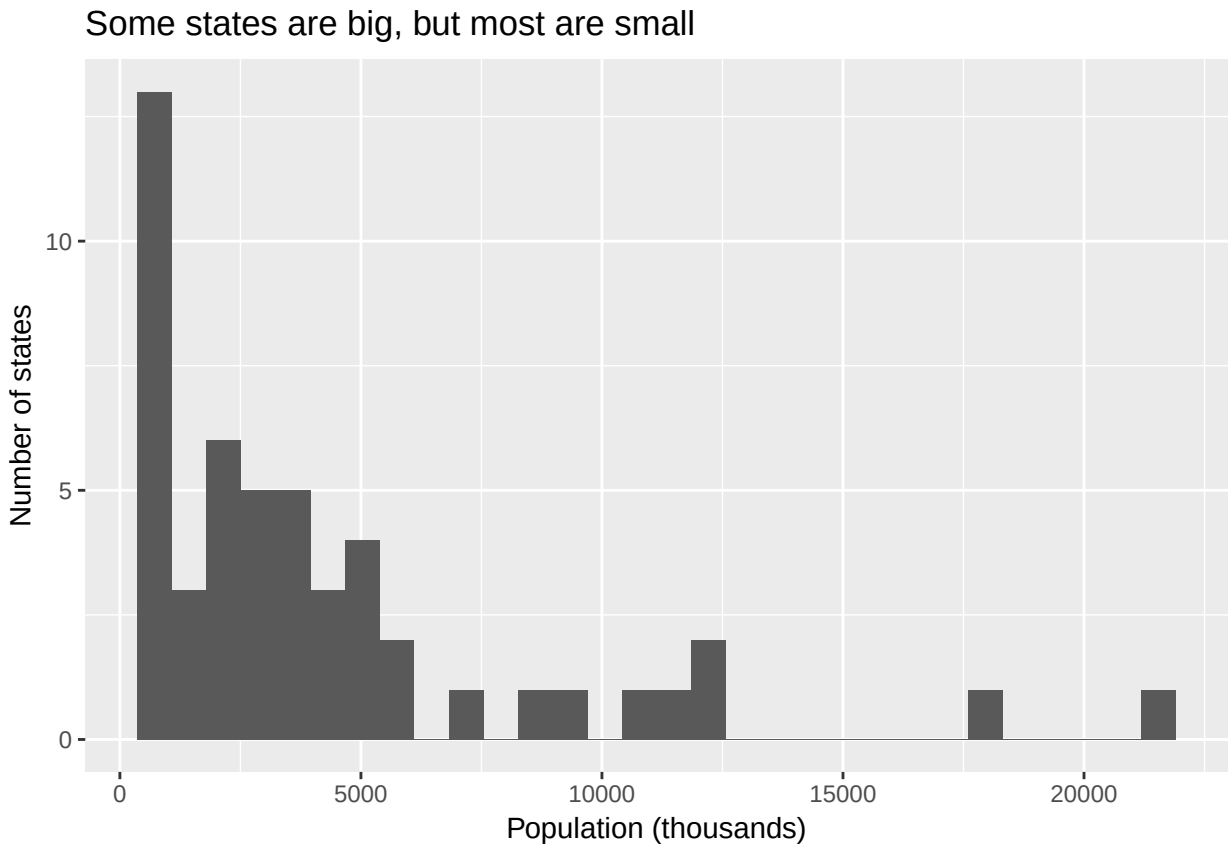
Let's walk through the syntax there. In the first line, we call `ggplot()`, specifying the data frame to draw from, then in the `aes()` command (which stands for "aesthetic") we specify the variable to plot. If this were a bivariate analysis, here we would have also specified a `y` variable to put on the y-axis. If we had just stopped there, we would have a sad, empty plot. The `+` symbol indicates that we'll be adding something to the plot. `geom_histogram()` is the command to overlay a histogram.

We'll only be looking at a few of the ggplot commands today. I recommend taking a look at the online package documentation at http://docs.ggplot2.org to see all of the many features available.

When you're just making graphs for yourself to explore the data, you don't need to worry about things like axis labels as long as you can comprehend what's going on. But when you prepare graphs for others to read (including those of us grading your problem sets!) you need to include an informative title and axis labels. To that end, use the `xlab()`, `ylab()`, and `ggtitle()` commands.

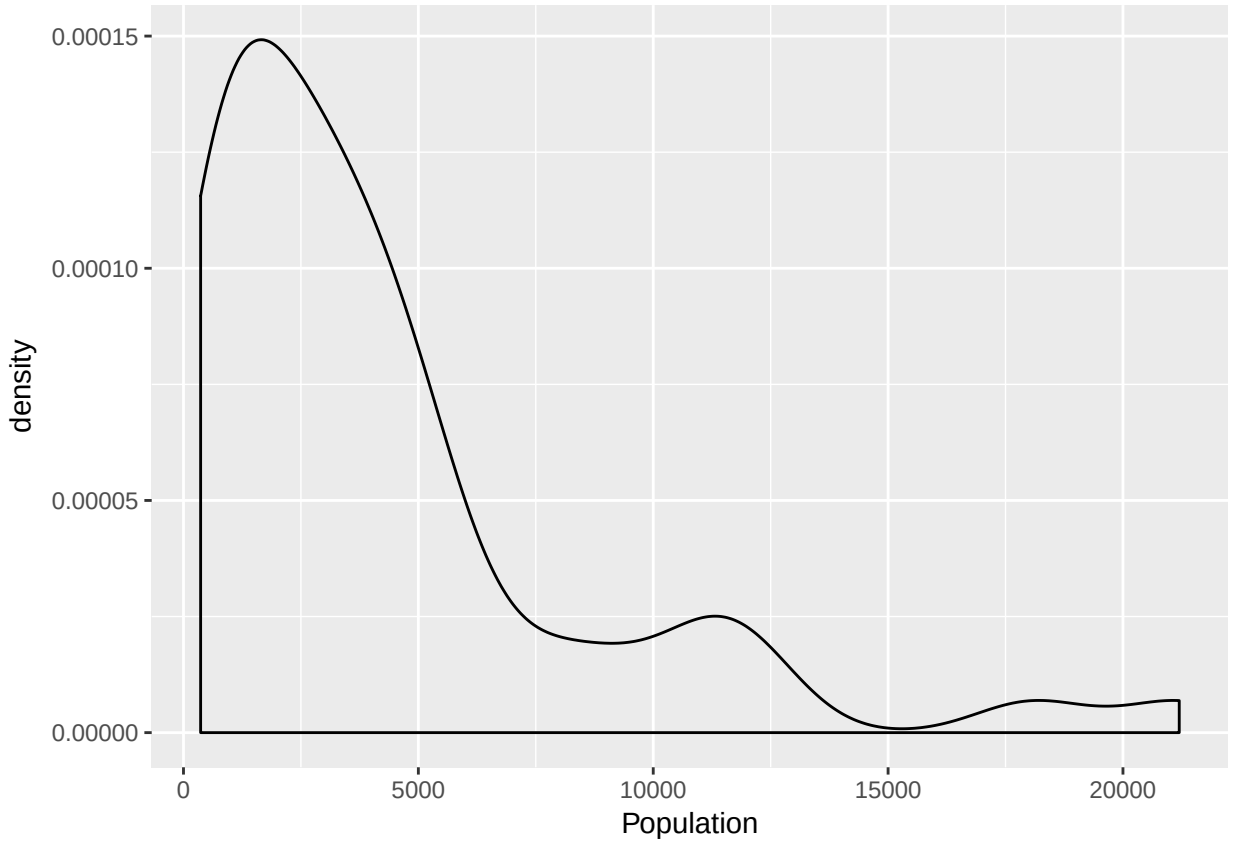
```
ggplot(state_data, aes(x = Population)) +
  geom_histogram() +
  xlab("Population (thousands)") +
  ylab("Number of states") +
  ggtitle("Some states are big, but most are small")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



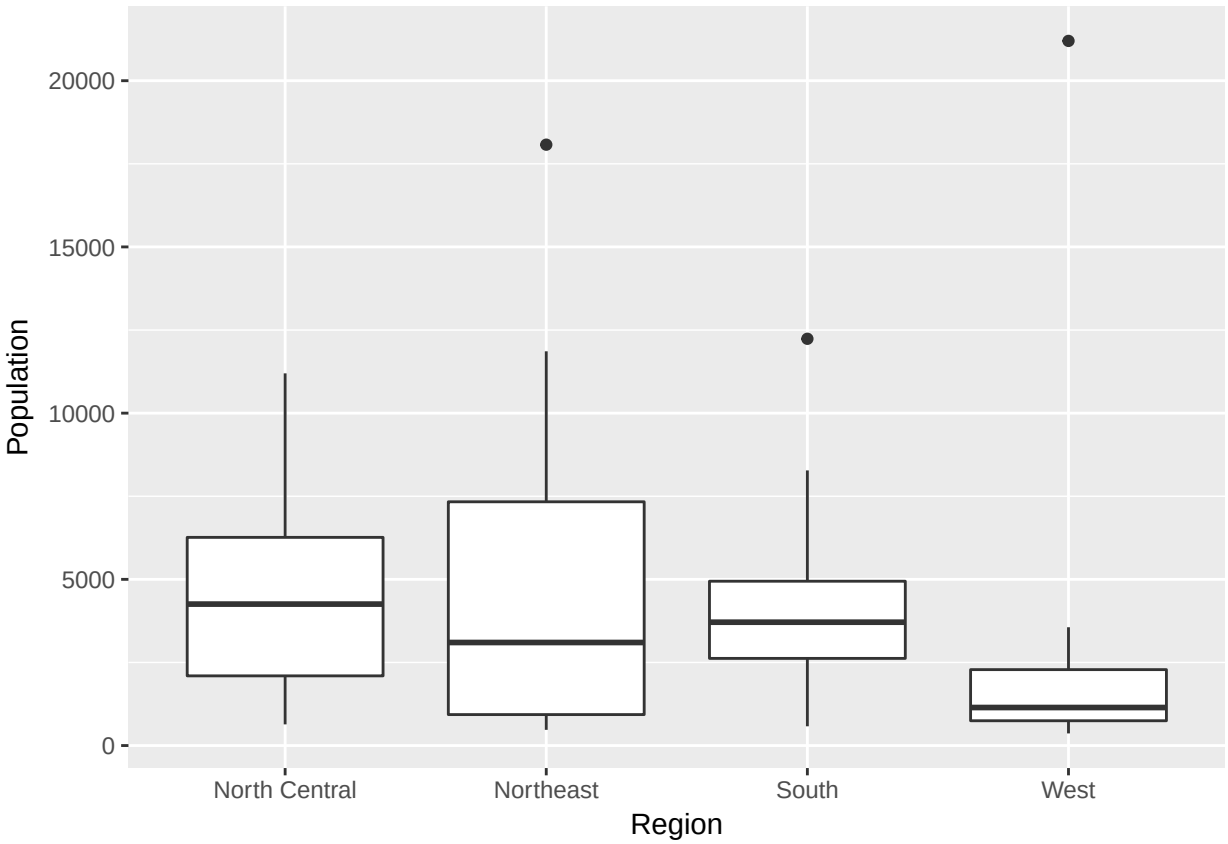
The density plot is a close relative of the histogram. It provides a smooth estimate of the probability density function of the data. Accordingly, the area under the density plot integrates to one. Depending on your purposes, this can make the y-axis of a density plot easier or (usually) harder to interpret than the count given by a histogram.

```
ggplot(state_data, aes(x = Population)) +
  geom_density()
```



The box plot is a common way to look at the distribution of a continuous variable across different levels of a categorical variable.

```
ggplot(state_data, aes(x = Region, y = Population)) +
  geom_boxplot()
```

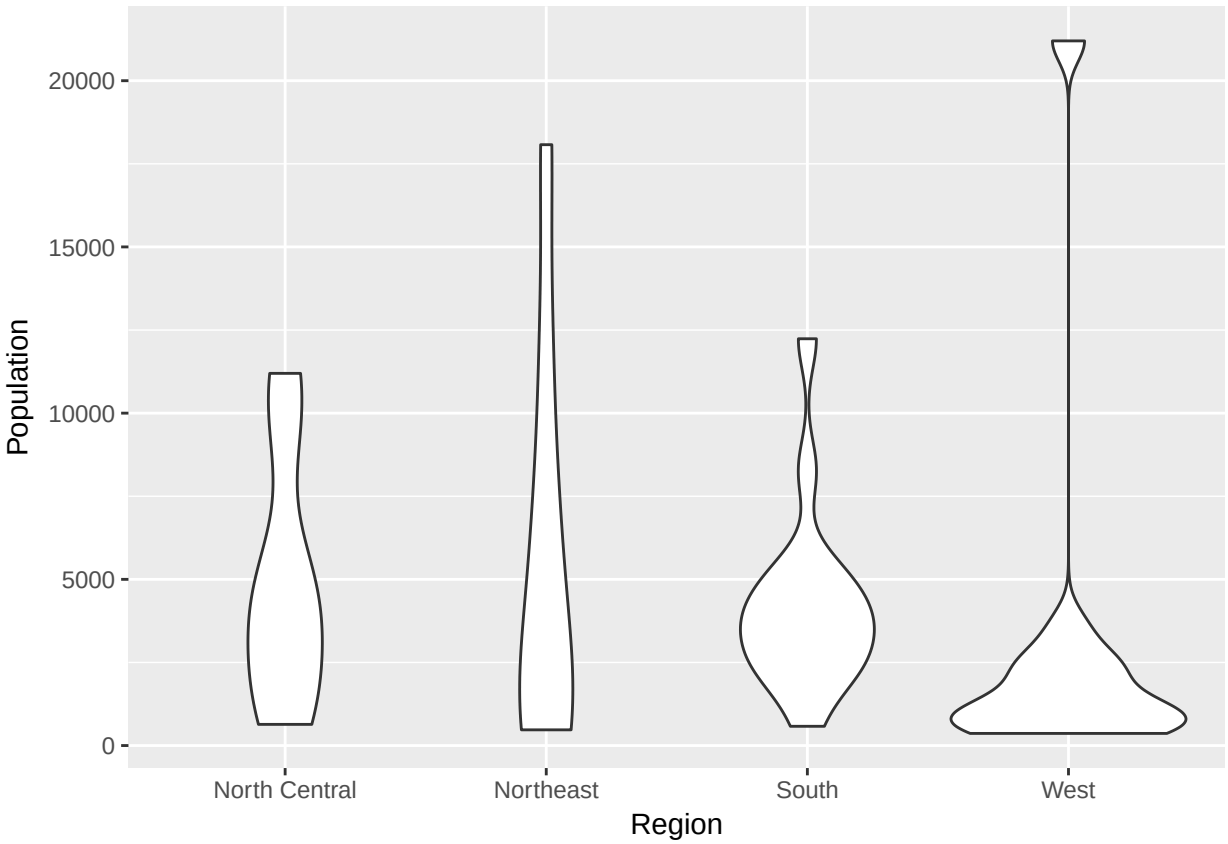


A box plot consists of the following components:

- Center line: median of the data
- Bottom of box: 25th percentile
- Top of box: 75th percentile
- Lower "whisker": range of observations no more than 1.5 IQR (height of box) below the 25th percentile
- Upper "whisker": range of observations no more than 1.5 IQR above the 75th percentile
- Plotted points: any data lying outside the whiskers

If you want to skip the summary and plot the full distribution of a variable across categories, you can use a violin plot.

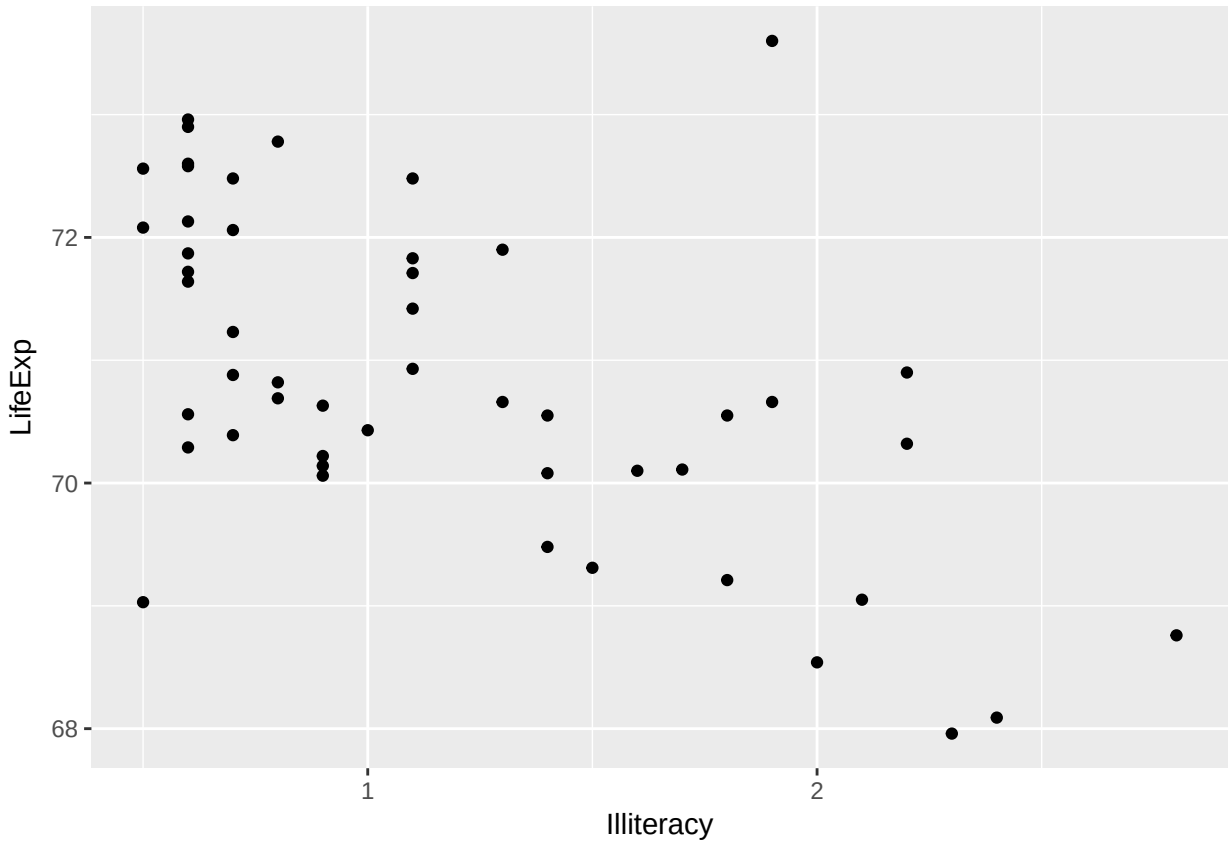
```
ggplot(state_data, aes(x = Region, y = Population)) +
  geom_violin()
```



Technically, violin plots convey more information than box plots since they show the full distribution. However, readers aren't as likely to be familiar with a violin plot. It's harder to spot immediately where the median is (though you could add that to the plot if you wanted). Plus, violin plots look goofy with outliers—see the "West" column above—whereas box plots handle them easily.

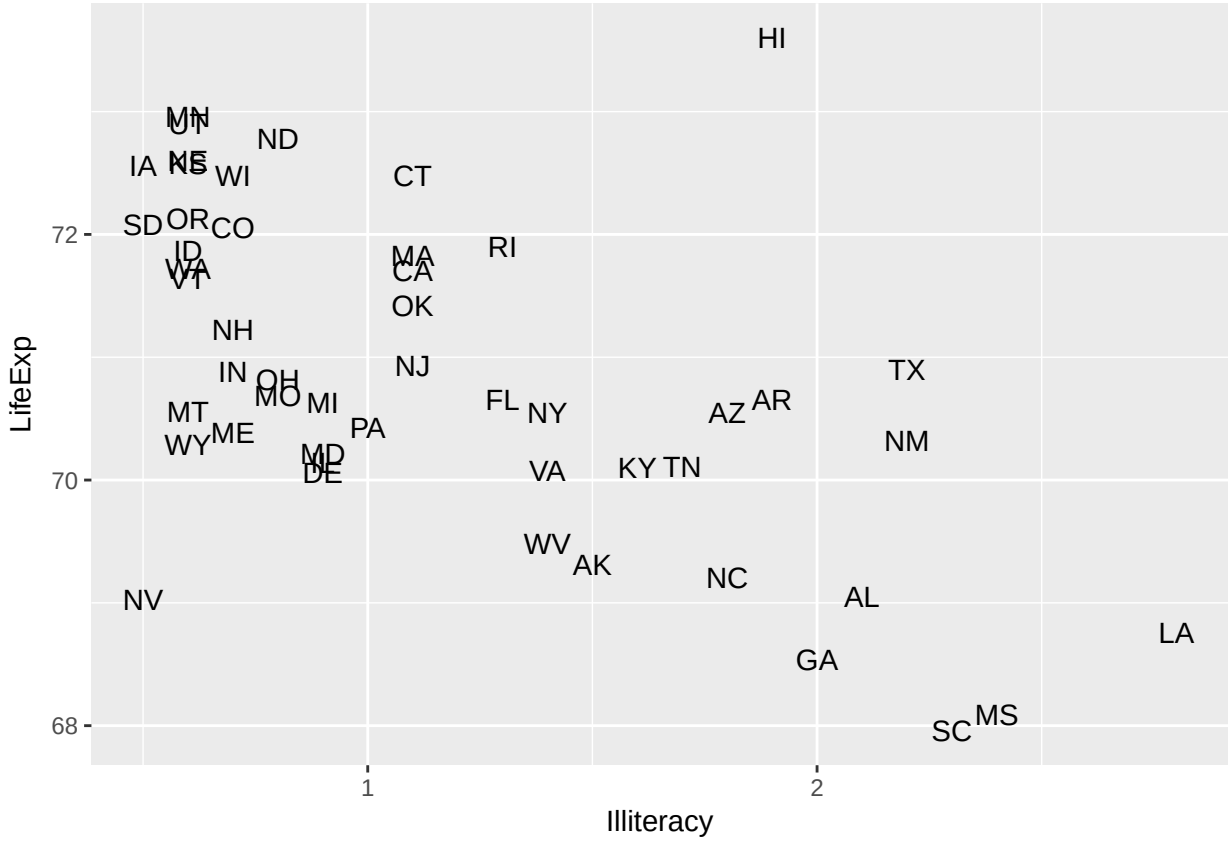
For visualizing relationships between continuous variables, nothing beats the scatterplot.

```
ggplot(state_data, aes(x = Illiteracy, y = LifeExp)) +
  geom_point()
```



When you're plotting states or countries, a hip thing to do is plot abbreviated names instead of points. To do that, you can use `geom_text()` instead of `geom_point()`, supplying an additional aesthetic argument telling ggplot where to draw the labels from.

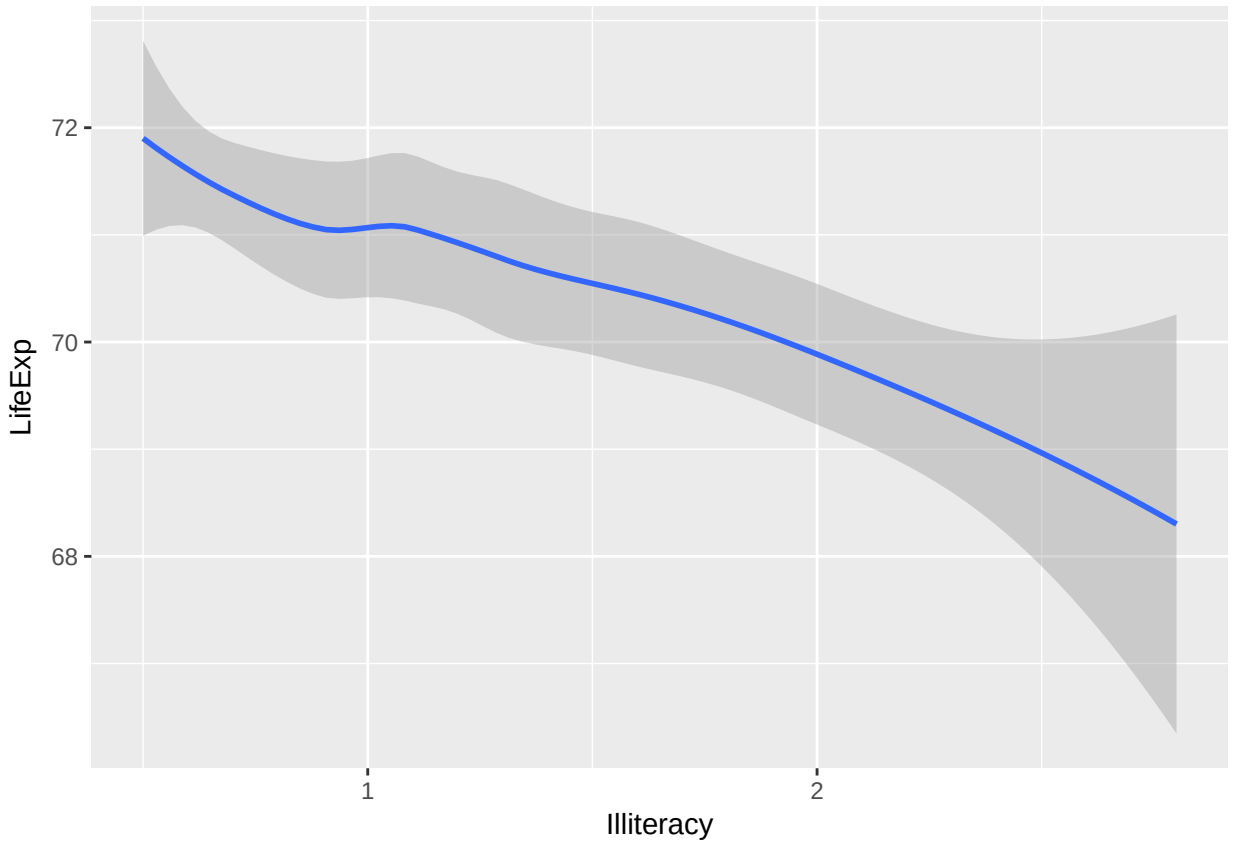
```
ggplot(state_data, aes(x = Illiteracy, y = LifeExp)) +
  geom_text(aes(label = Abbrev))
```



Maybe it's overwhelming to look at all that raw data and you just want a summary. For example, maybe you want an estimate of expected `LifeExp` for each value of `Illiteracy`. This is called the *conditional expectation* and will be the subject of much of the rest of the course. For now, just now that you can calculate a smoothed conditional expectation via `geom_smooth()`.

```
ggplot(state_data, aes(x = Illiteracy, y = LifeExp)) +
  geom_smooth()
```

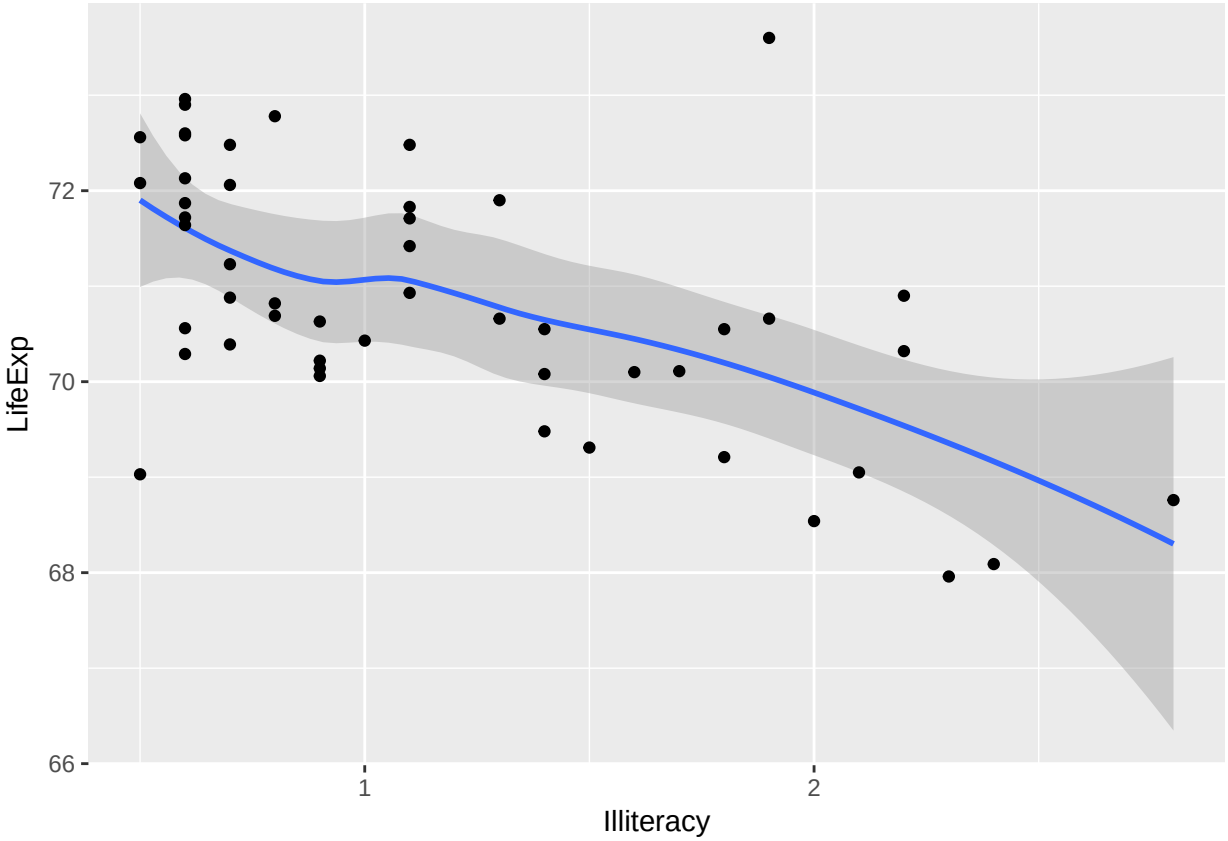
```
## `geom_smooth()` using method = 'loess'
```



And if you're the kind of overachiever who likes to have the raw data *and* the summary, you can do it. Just add them both to the `ggplot()` call.

```
ggplot(state_data, aes(x = Illiteracy, y = LifeExp)) +
  geom_smooth() +
  geom_point()
```

```
## `geom_smooth()` using method = 'loess'
```



4.2 Saving Plots

When you're writing in R Markdown, the plots go straight into your document without much fuss. Odds are, your dissertation will contain plots but won't be written in R Markdown, which means you'll need to learn how to save them.

It's pretty simple:

1. Assign your `ggplot()` call to a variable.
2. Pass that variable to the `ggsave()` function.

```
pop_hist <- ggplot(state_data, aes(x = Population)) +
  geom_histogram()

ggsave(filename = "pop-hist.pdf",
       plot = pop_hist,
       width = 6,
       height = 3)
```

If you want plot types other than PDF, just set a different extension. See `?ggsave` for the possibilities.

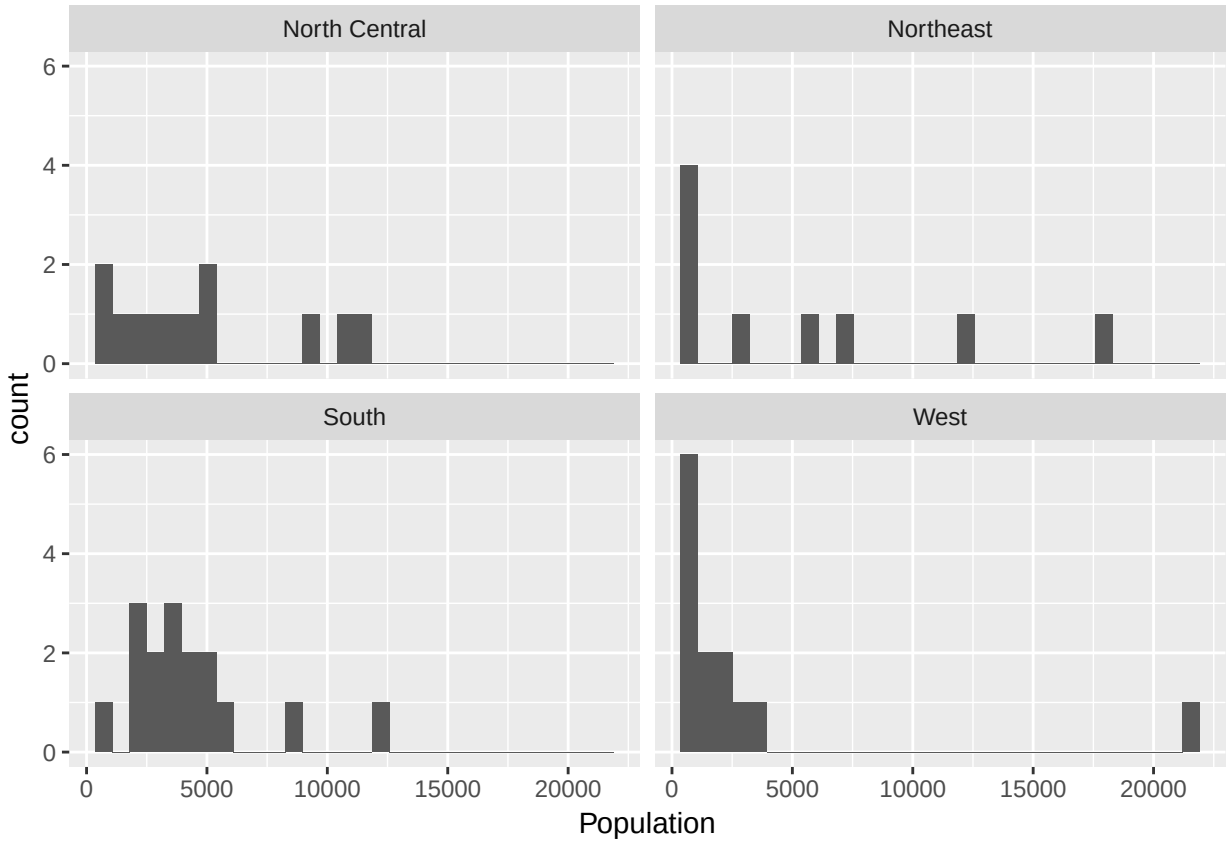
4.3 Faceting

Suppose you want to split the data into subgroups, as defined by some variable in the data (e.g., the region states are in), and make the same plot for each subgroup. ggplot's *faceting* functions, `facet_wrap()` and `facet_grid()`, make this easy.

To split up plots according to a single grouping variable, use `facet_wrap()`. This uses R's *formula* syntax, defined by the tilde `~`, which you'll become well acquainted with once we start running regressions.

```
ggplot(state_data, aes(x = Population)) +
  geom_histogram() +
  facet_wrap(~ Region)
```

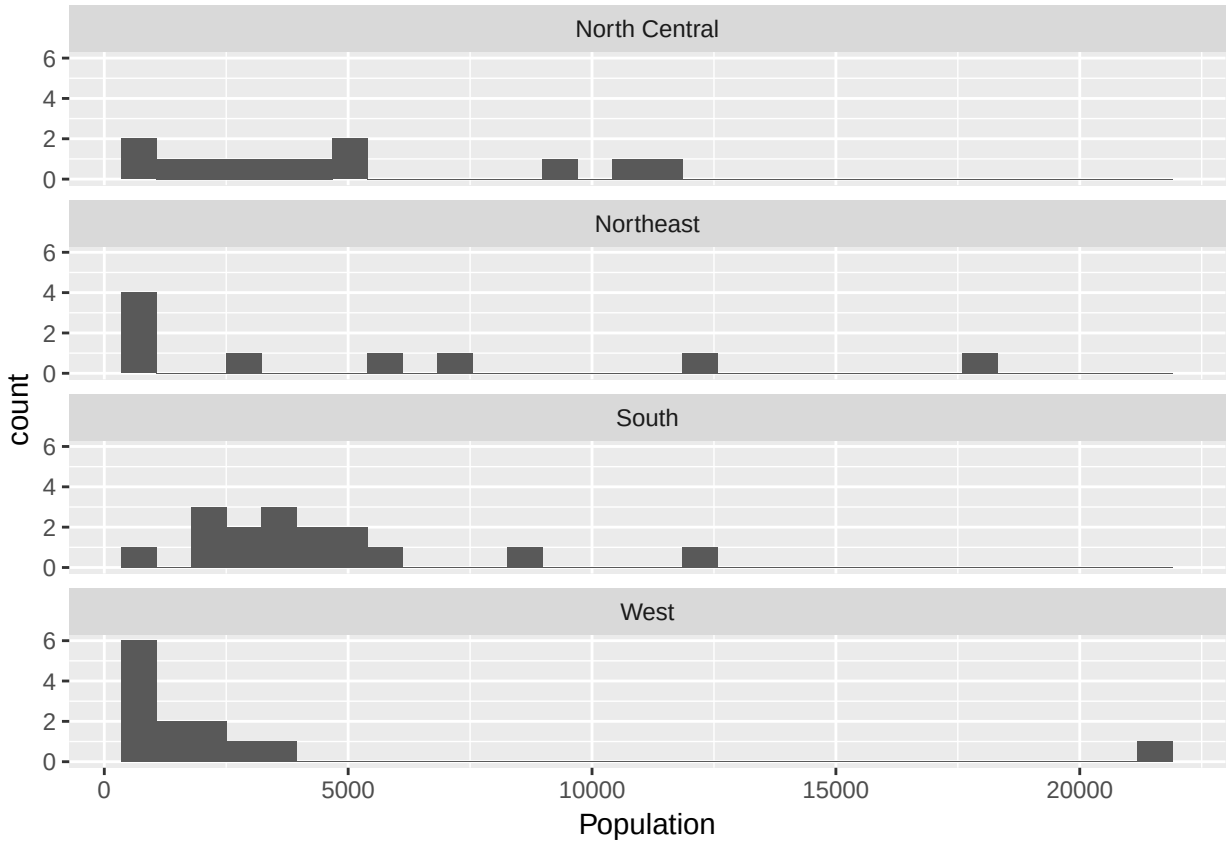
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



If you don't like the default arrangement, use the `ncol` argument.

```
ggplot(state_data, aes(x = Population)) +
  geom_histogram() +
  facet_wrap(~ Region, ncol = 1)
```

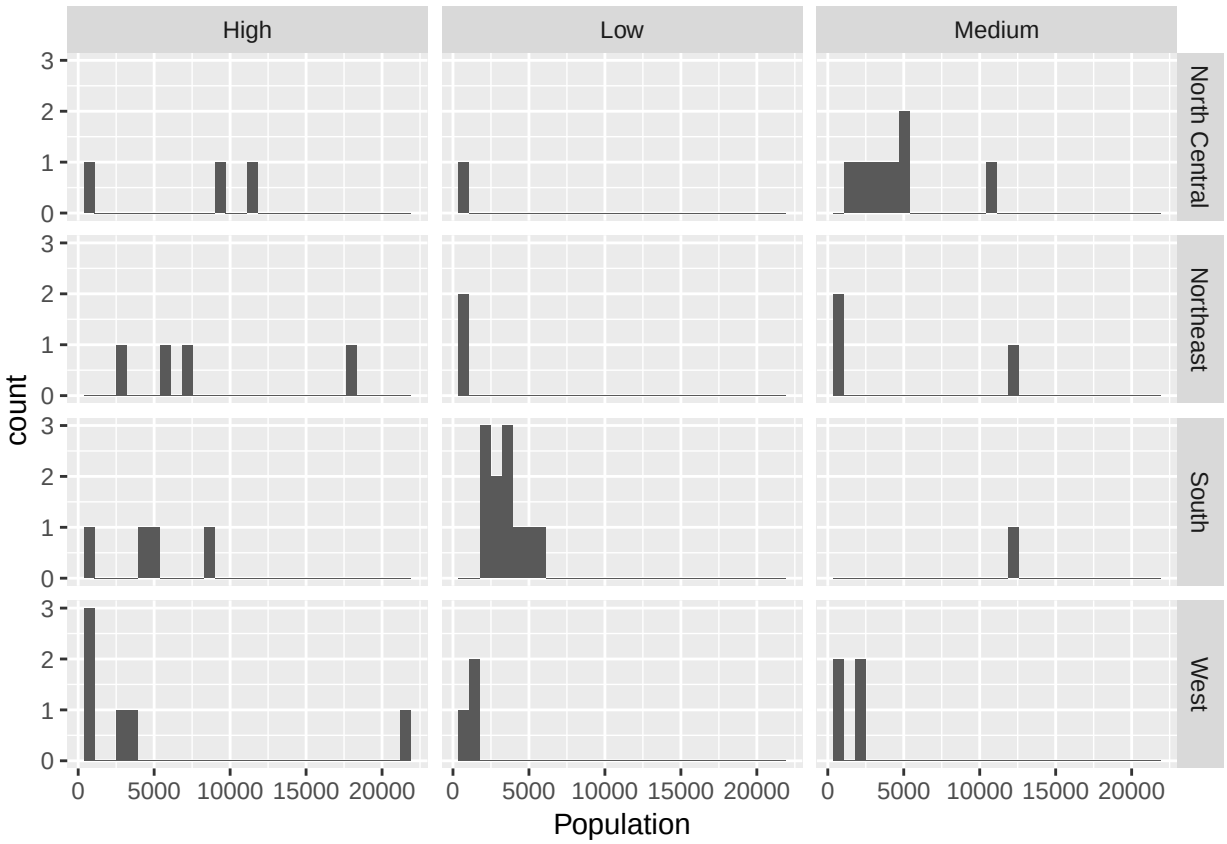
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



For two grouping variables, use `facet_grid()`, putting variables on both sides of the formula.

```
ggplot(state_data, aes(x = Population)) +
  geom_histogram() +
  facet_grid(Region ~ IncomeGroup)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

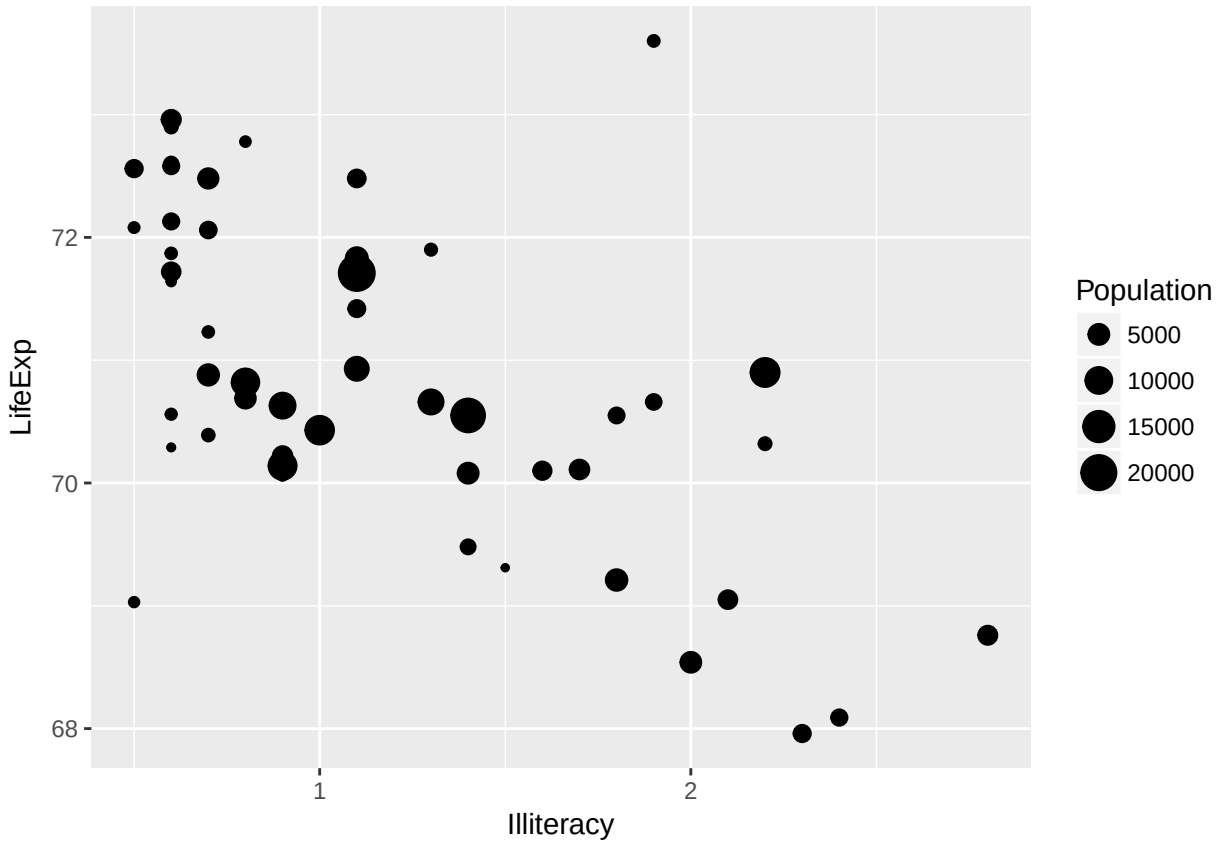


4.4 Aesthetics

Faceting is one way to incorporate information about additional variables into what would otherwise be a plot of just one or two variables. Aesthetics—which alter the appearance of particular plot features depending on the value of a variable—provide another way to do that.

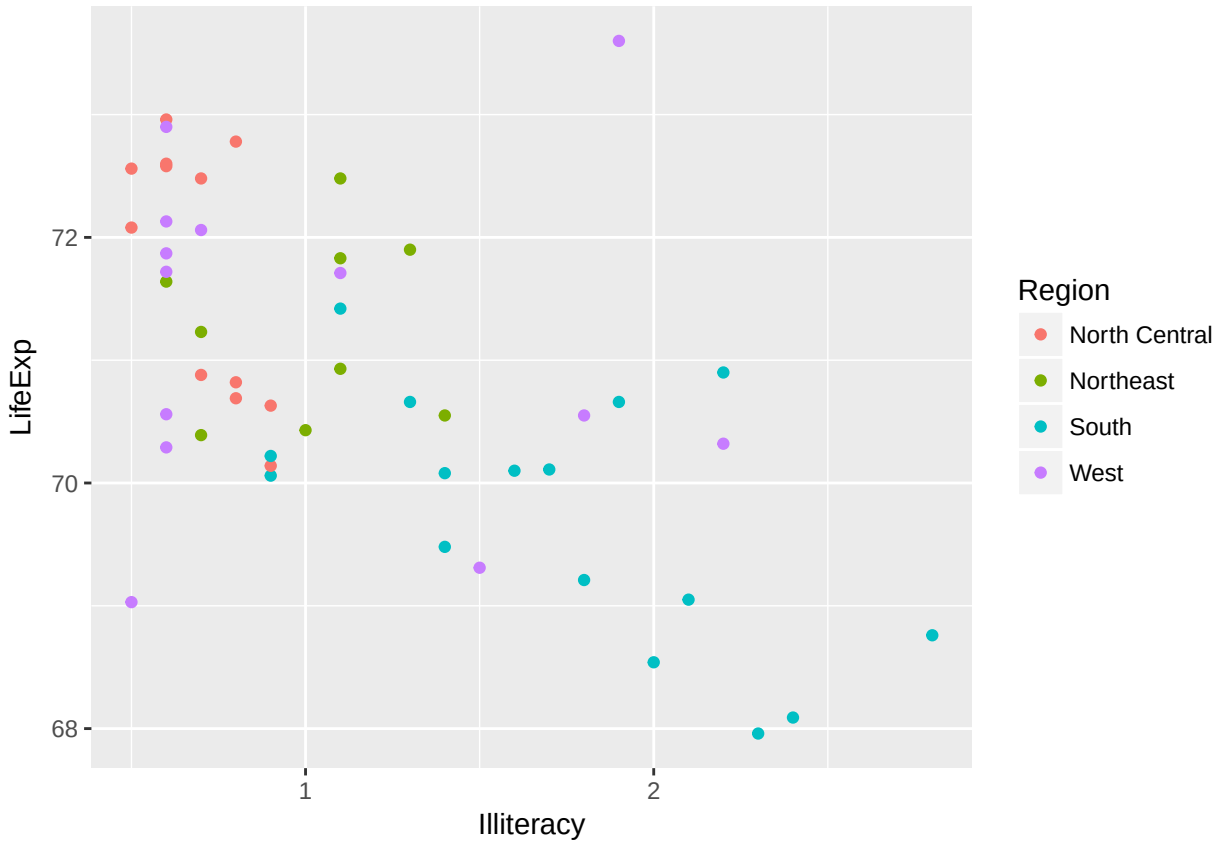
For example, when visualizing the relationship between statewide illiteracy and life expectancy, you might want larger states to get more visual weight. You can set the `size` aesthetic of the `point` geometry to vary according to the state's population.

```
ggplot(state_data, aes(x = Illiteracy, y = LifeExp)) +
  geom_point(aes(size = Population))
```



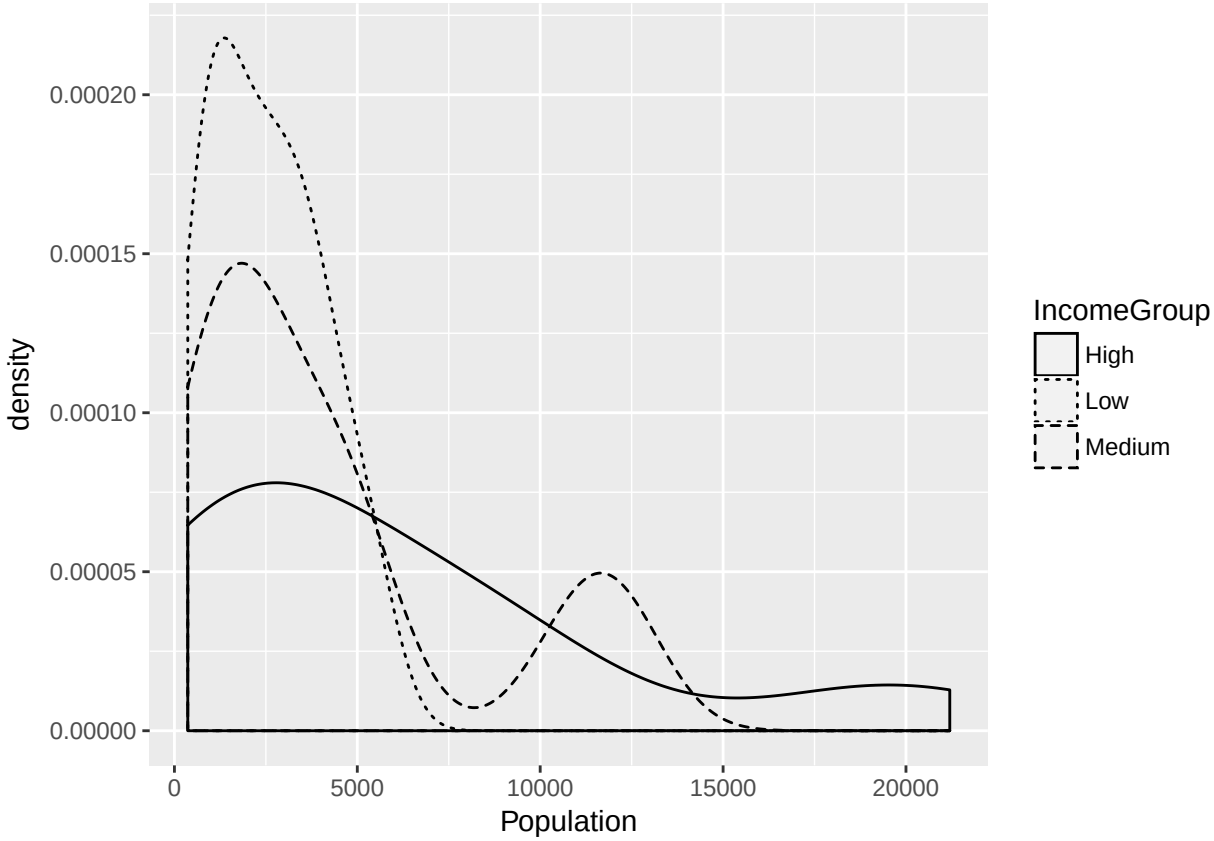
The **ggplot2** documentation lists the available aesthetics for each function. Another popular one is `colour`, which is great for on-screen display but not so much for the printed page. (And terrible for the colorblind!)

```
ggplot(state_data, aes(x = Illiteracy, y = LifeExp)) +
  geom_point(aes(colour = Region))
```



For line graphs or density plots, you can set the `linetype` to vary by category.

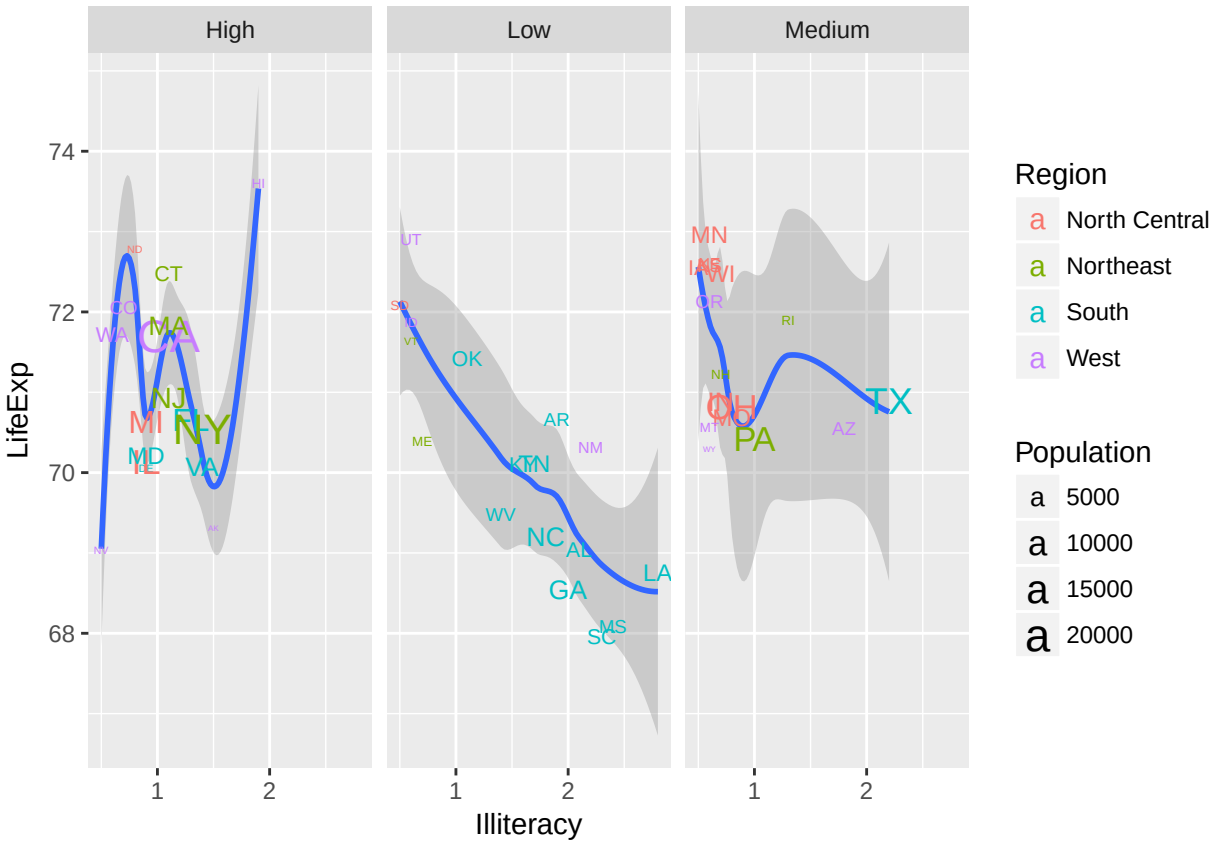
```
ggplot(state_data, aes(x = Population)) +
  geom_density(aes(linetype = IncomeGroup))
```



(I always find these incomprehensible with more than two lines, but maybe that's just me.) You can use multiple aesthetics together, and you can even combine aesthetics with faceting, as in the following example.

```
ggplot(state_data, aes(x = Illiteracy, y = LifeExp)) +
  geom_smooth() +
  geom_text(aes(label = Abbrev, colour = Region, size = Population)) +
  facet_wrap(~ IncomeGroup)
```

```
## `geom_smooth()` using method = 'loess'
```



But the fact that you *can* do something doesn't mean you *should*. That plot is so cluttered that it's hard to extract the relevant information from it. Data visualizations should communicate a clear message to viewers without overwhelming them. To do this well takes practice, patience, and maybe even a bit of taste.

4.5 Appendix: Creating the Example Data

The example data comes from data on U.S. states in 1977 that are included with base R. See `?state`.

```
library("tidyverse")

state_data <- state.x77 %>%
  as_tibble() %>%
  add_column(State = rownames(state.x77),
             Abbrev = state.abb,
             Region = state.region,
             .before = 1) %>%
  rename(LifeExp = `Life Exp`,
         HSGrad = `HS Grad`) %>%
  mutate(IncomeGroup = cut(Income,
```

```
                            breaks = quantile(Income,
                                              probs = seq(0, 1, by = 1/3)),
                            labels = c("Low", "Medium", "High"),
                            include.lowest = TRUE))

write_csv(state_data, path = "state-data.csv")
```

Chapter 5

Bivariate Regression

The goal of empirical social science is usually to learn about the relationships between variables in the social world. Our goals might be descriptive: were college graduates more likely to vote for Clinton in 2016? Or causal: does receiving more education make a person more liberal on average? Or predictive: what kinds of voters should Democrats target in 2020 to have the best chance of victory?

The linear model is one of the simplest ways to model relationships between variables. Ordinary least squares regression is one of the easiest and (often) best ways to estimate the parameters of the linear model. Consequently, a linear model estimated by OLS is the starting point for many analyses. We will start with the simplest case: regression on a single covariate.

5.1 Probability Refresher

Let Y be a random variable that takes values in the finite set $\mathcal{Y}$ according to the probability mass function $f_Y : \mathcal{Y} \to [0, 1]$. The *expected value* (aka *expectation*) of Y is the weighted average of each value in $\mathcal{Y}$, where the weights are the corresponding probabilities:

$$E[Y] = \sum_{y \in \mathcal{Y}} y \, f_Y(y); \tag{5.1}$$

For a continuous random variable Y on $\mathbb{R}$ with probability density function f_Y, the expected value is the analogous integral:

$$E[Y] = \int y \, f_Y(y) \, dy. \tag{5.2}$$

Now suppose (X, Y) is a pair of discrete random variables drawn according to the joint mass function f_{XY} on $\mathcal{X} \times \mathcal{Y}$, with respective marginal mass functions f_X and f_Y.[1] Recall the

[1] The marginal mass function, if you don't recall, is $f_X(x) = \sum_{y \in \mathcal{Y}} f_{XY}(x, y)$. In the continuous case, the marginal density function is $f_X(x) = \int f_{XY}(x, y) \, dy$.

formula for conditional probability,

$$\Pr(Y = y \mid X = x) = \frac{\Pr(X = x, Y = y)}{\Pr(X = x)} = \frac{f_{XY}(x, y)}{f_X(x)}. \tag{5.3}$$

For each $x \in \mathcal{X}$, we have the *conditional mass function*

$$f_{Y|X}(y \mid x) = \frac{f_{XY}(x, y)}{f_X(x)} \tag{5.4}$$

and corresponding *conditional expectation*

$$E[Y|X = x] = \sum_{y \in \mathcal{Y}} y \, f_{Y|X}(y \mid x). \tag{5.5}$$

For continuous random variables, the conditional expectation is

$$E[Y|X = x] = \int y \, f_{Y|X}(y \mid x) \, dy, \tag{5.6}$$

where $f_{Y|X}$ is the conditional density function.

The *variance* of a random variable Y is

$$V[Y] = E[(Y - E[Y])^2]. \tag{5.7}$$

Given a sample $Y_1, \ldots, Y_N$ of observations of Y, we usually estimate $V[Y]$ with the *sample variance*

$$S_Y^2 = \frac{1}{N-1} \sum_n (Y_n - \bar{Y})^2, \tag{5.8}$$

where $\bar{Y}$ is the sample mean and $\sum_n$ denotes summation from $n = 1$ to N.

Similarly (in fact a generalization of the above), the *covariance* between random variables X and Y is

$$\mathrm{Cov}[X, Y] = E[(X - E[X])(Y - E[Y])],$$

which we estimate with the *sample covariance*

$$S_{XY} = \frac{1}{N-1} \sum_n (X_n - \bar{X})(Y_n - \bar{Y}). \tag{5.9}$$

A fun fact about the sample covariance is that

$$S_{XY} = \frac{1}{N-1}\sum_n (X_n - \bar{X})(Y_n - \bar{Y}) \tag{5.10}$$

$$= \frac{1}{N-1}\left[\sum_n X_n(Y_n - \bar{Y}) + \sum_n \bar{X}(Y_n - \bar{Y})\right] \tag{5.11}$$

$$= \frac{1}{N-1}\left[\sum_n X_n(Y_n - \bar{Y}) + \bar{X}\sum_n (Y_n - \bar{Y})\right] \tag{5.12}$$

$$= \frac{1}{N-1}\sum_n X_n(Y_n - \bar{Y}). \tag{5.13}$$

If we had split up the second term instead of the first, we would see that

$$S_{XY} = \frac{1}{N-1}\sum_n Y_n(X_n - \bar{X})$$

as well.

Since the (sample) variance is a special case of the (sample) covariance, by the same token we have

$$S_Y^2 = \frac{1}{N-1}\sum_n Y_n(Y_n - \bar{Y}). \tag{5.14}$$

5.2 The Linear Model

Suppose we observe a sequence of N draws from f_{XY}, denoted $(X_1, Y_1), (X_2, Y_2), \ldots, (X_N, Y_N)$, or $\{(X_n, Y_n)\}_{n=1}^N$ for short. What can we learn about the relationship between X and Y from this sample of data?

If we were really ambitious, we could try to estimate the shape of the full joint distribution, f_{XY}. The joint distribution encodes everything there is to know about the relationship between the two variables, so it would be pretty useful to know. But except in the most trivial cases, it would be infeasible to estimate f_{XY} precisely. If X or Y can take on more than a few values, estimating the joint distribution would require an amount of data that we're unlikely to have.[2]

The first way we simplify our estimation task is to set our sights lower. Let Y be the *response* or the *dependent variable*—i.e., the thing we want to explain. We call X the *covariate* or the *independent variable*. Instead of estimating the full joint distribution, we're just going to try to learn the conditional expectation, $E[Y \mid X]$. In other words, for each potential value of the covariate, what is the expected value of the response? This will allow us to answer questions like whether greater values of X are associated with greater values of Y.

[2]This problem only gets worse as we move from bivariate into multivariate analysis, a phenomenon called the *curse of dimensionality*.

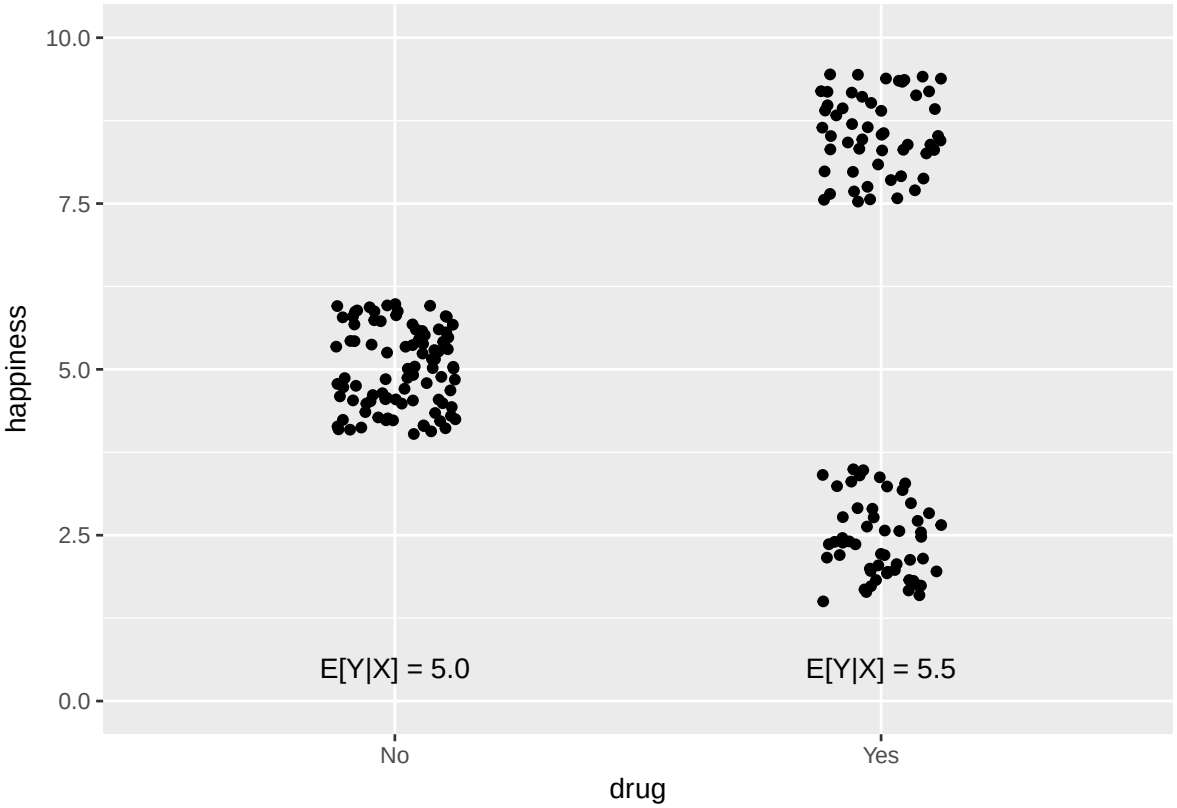
Two important things about the estimation of conditional expectations before we go any further.

1. Statements about conditional expectations are not causal. If Y is rain and X is umbrella sales, we know $E[Y|X]$ increases with X, but that doesn't mean umbrella sales make it rain.

   We will spend some time in the latter part of the course on how to move from conditional expectations to causality. Then, in Stat III, you will learn about causal inference in excruciating detail.

2. The conditional expectation doesn't give you everything you'd want to know about the relationship between variables.

   As a hypothetical example, suppose I told you that taking a particular drug made people happier on average. In other words, $E[\text{Happiness} \mid \text{Drug}] > E[\text{Happiness} \mid \text{No Drug}]$. Sounds great! Then imagine the dose-response graph looked like this:



   The fact that expected happiness rises by half a point doesn't quite tell the whole story.

In spite of these caveats, conditional expectation is a really useful tool for summarizing the relationship between variables.

If X takes on sufficiently few values (and we have enough data), we don't need to model the conditional expectation function. We can just directly estimate $E[Y|X = x]$ for each $x \in \mathcal{X}$. The graph above, where there are just two values of X, is one example.

But if X is continuous, or even if it is discrete with many values, estimating $E[Y|X]$ for each distinct value is infeasible. In this case, we need to *model* the relationship. The very simplest choice—and thus the default for social scientists—is to model the conditional expectation of Y as a linear function of X:

$$E[Y \mid X] = \alpha + \beta X. \tag{5.15}$$

In this formulation, α and β are the parameters to be estimated from sample data. We call α and β "coefficients," with α the "intercept" and β the "slope." Regardless of how many different values X might take on, we only need to estimate two parameters of the linear model.

Exercise your judgment before using a linear model. Ask yourself, is a linear conditional expectation function at least minimally plausible? Not perfect—just a reasonable approximation. If X is years of education and Y is annual income, the answer is probably yes (depending on the population!). But if X is hour of the day (0–24) and Y is the amount of traffic on I-65, probably not.

To obtain the linear conditional expectation, we usually assume the following model of the response variable:

$$Y_n = \alpha + \beta X_n + \epsilon_n, \tag{5.16}$$

where ϵ_n is "white noise" error with the property

$$E[\epsilon_n \mid X_1, \ldots X_N] = 0. \tag{5.17}$$

You can think of ϵ_n as the summation of everything besides the covariate X_n that affects the response Y_n. The assumption that $E[\epsilon_n \mid X_1, \ldots, X_N] = 0$ implies that these external factors are uncorrelated with the covariate. This is not a trivial technical condition that you can ignore—it is a substantive statement about the variables in your model. It requires justification, and it is difficult to justify.

For now we will proceed assuming that our data satisfy the above conditions. Later in the course, we will talk about how to proceed when $E[\epsilon_n \mid X_1, \ldots, X_N] \neq 0$, and you will learn much more about such strategies in Stat III.

5.3 Least Squares

To estimate the parameters of the linear model, we will rely on a mathematically convenient method called *least squares*. We will see that this method not only is convenient, but also has nice statistical properties.

Given a parameter estimate $(\hat{\alpha}, \hat{\beta})$, define the *residual* of the n'th observation as the difference between the true and predicted values:

$$e_n(\hat{\alpha}, \hat{\beta}) = Y_n - \hat{\alpha} - \hat{\beta} X_n. \tag{5.18}$$

The residual is directional. The residual is positive when the regression line falls below the observation, and vice versa when it is negative.

We would like the regression line to lie close to the data—i.e., for the residuals to be small in magnitude. "Close" can mean many things, so we need to be a bit more specific to derive an estimator. The usual one, *ordinary least squares*, is chosen to minimize the sum of squared errors,

$$\text{SSE}(\hat{\alpha}, \hat{\beta}) = \sum_n e_n(\hat{\alpha}, \hat{\beta})^2.$$

(Throughout the rest of this chapter, I write $\sum_n$ as shorthand for $\sum_{n=1}^{N}$.) When we focus on squared error, we penalize a positive residual the same as a negative residual of the same size. Moreover, we penalize one big residual proportionally more than a few small ones.

It is important to keep the linear model and ordinary least squares distinct in your mind. The linear model is a model of the data. Ordinary least squares is one estimator—one among many—of the parameters of the linear model. Assuming a linear model does not commit you to estimate it with OLS if you think another estimator is more appropriate. And using OLS does not necessarily commit you to the linear model, as we will discuss when we get to multiple regression.

To derive the OLS estimator, we will derive the conditions for minimization of the sum of squared errors. The SSE is a quadratic and therefore continuously differentiable function of the estimands, $\hat{\alpha}$ and $\hat{\beta}$. You will remember from calculus that, at any extreme point of a continuous function, all its partial derivatives equal zero. To derive necessary conditions for minimization,[3] we can take the derivatives of the SSE and set them to equal zero.

The derivative with respect to the intercept is

$$\frac{\partial \, \text{SSE}(\hat{\alpha}, \hat{\beta})}{\partial \hat{\alpha}} = -2 \sum_n (Y_n - \hat{\alpha} - \hat{\beta} X_n).$$

Setting this to equal zero gives

$$\hat{\alpha} = \frac{1}{N} \sum_n (Y_n - \hat{\beta} X_n) = \bar{Y} - \hat{\beta} \bar{X}.$$

This gives us one important property of OLS: the regression line estimated by OLS always passes through $(\bar{X}, \bar{Y})$.

The derivative with respect to the slope is

$$\frac{\partial \, \text{SSE}(\hat{\alpha}, \hat{\beta})}{\partial \hat{\beta}} = -2 \sum_n X_n (Y_n - \hat{\alpha} - \hat{\beta} X_n).$$

[3] In fact, since the SSE function is strictly convex, these conditions are sufficient for global minimization.

Setting this equal to zero and substituting in the expression for $\hat{\alpha}$ we derived above gives

$$\sum_n X_n(Y_n - \bar{Y}) = \hat{\beta} \sum_n X_n(X_n - \bar{X}).$$

As long as the sample variance of X is non-zero (i.e., X is not a constant), we can divide to solve for $\hat{\beta}$:

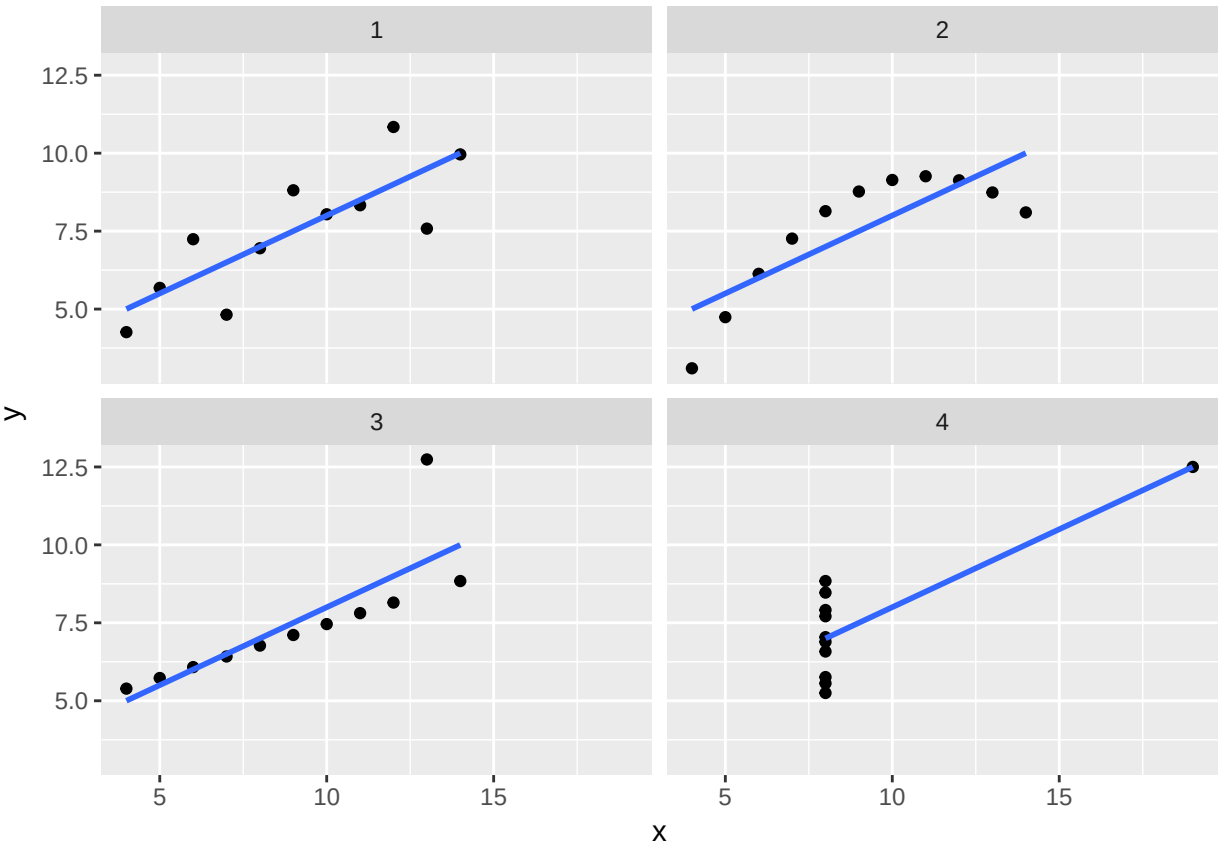
$$\hat{\beta} = \frac{\sum_n X_n(Y_n - \bar{Y})}{\sum_n X_n(X_n - \bar{X})} = \frac{S_{XY}}{S_X^2}.$$

Combining these two results, we have the OLS estimators of the intercept and slope of the bivariate linear model. We write them as functions of $(X_1, \ldots, X_N, Y_1, \ldots, Y_N)$, or (X, Y) for short,[4] to emphasize that an estimator is a statistic, which in turn is a function of sample data. We place the "OLS" subscript on them to emphasize that there are many estimators of these parameters, of which OLS is just one (good!) choice.

$$\hat{\alpha}_{\text{OLS}}(X, Y) = \bar{Y} - \frac{S_{XY}}{S_X^2}\bar{X},$$
$$\hat{\beta}_{\text{OLS}}(X, Y) = \frac{S_{XY}}{S_X^2}.$$

Regression is a convenient way to summarize the relationship between variables, but it is a complement to—not a substitute for—graphical analysis. The statistician Francis Anscombe found that OLS yields nearly identical regression lines for all four of the datasets in the following graph:

[4]This is a bit of an abuse of notation, since previously I used X and Y to refer to the random variables and now I'm using them to refer to vectors of sample data. Sorry.



Unless your data all lie along a line, the regression line estimated by OLS will not predict the data perfectly. Let the *residual sum of squares* be the squared error left over by OLS,

$$\text{RSS} = \text{SSE}(\hat{\alpha}_{\text{OLS}}, \hat{\beta}_{\text{OLS}}),$$

and let the *total sum of squares* be the squared error that would result from a horizontal regression line through the mean of Y,

$$\text{TSS} = \text{SSE}(\bar{Y}, 0).$$

The R^2 statistic is the proportion of "variance explained" by X, calculated as

$$R^2 = 1 - \frac{\text{RSS}}{\text{TSS}}.$$

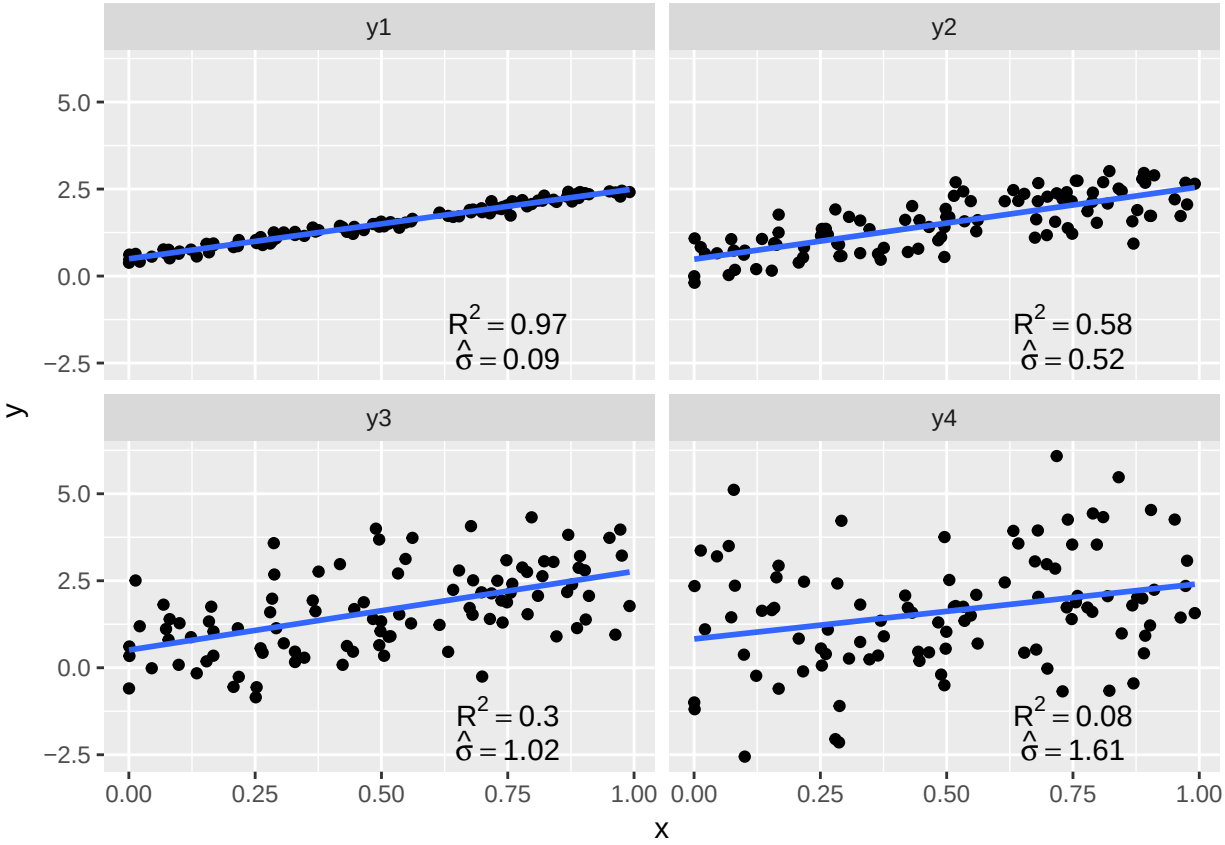
If the regression line is flat, in which case $\hat{\beta}_{\text{OLS}} = 0$ and RSS = TSS, we have $R^2 = 0$. Conversely, if the regression line fits perfectly, in which case RSS = 0, we have $R^2 = 1$.

A statistic that is often more useful than R^2 is the *residual variance*. The residual variance is (almost) the sample variance of the regression residuals, calculated as

$$\hat{\sigma}^2 = \frac{1}{N-2} \sum_n e_n(\hat{\alpha}_{\text{OLS}}, \hat{\beta}_{\text{OLS}})^2 = \frac{\text{RSS}}{N-2}$$

Since bivariate regression uses two degrees of freedom (one for the intercept, one for the slope), we divide by $N-2$ instead of the usual $N-1$. The most useful quantity is $\hat{\sigma}$,

the square root of the residual variance. $\hat{\sigma}$ is measured in the same units as Y, and it is a measure of the spread of points around the regression line. If the residuals are roughly normally distributed, then we would expect roughly 95% of the data to lie within $\pm 2\hat{\sigma}$ of the regression line.



5.4 Properties

We didn't use any fancy statistical theory to derive the OLS estimator. We just found the intercept and slope that minimize the sum of squared residuals. As it turns out, though, OLS indeed has some very nice statistical properties as an estimator of the linear model.

The first desirable property of OLS is that it is *unbiased*. Recall that an estimator $\hat{\theta}$ of the parameter θ is unbiased if $E[\hat{\theta}] = \theta$. This doesn't mean the estimator always gives us the right answer, just that on average it is not systematically biased upward or downward. In other words, if we could take many many samples and apply the estimator to each of them, the average would equal the true parameter.

We will begin by showing that the OLS estimator of the slope is unbiased; i.e., that $E[\hat{\beta}_{\text{OLS}}(X, Y)] = \beta$. At first, we'll take the conditional expectation of the slope estimator,

treating the covariates $(X_1, \ldots, X_N)$ as fixed.

$$\begin{aligned}
E[\hat{\beta}_{\text{OLS}}(X, Y) \mid X] &= E\left[\frac{S_{XY}}{S_X^2} \,\middle|\, X \right] \\
&= E\left[\frac{\sum_n Y_n (X_n - \bar{X})}{\sum_n X_n (X_n - \bar{X})} \,\middle|\, X \right] \\
&= \frac{\sum_n E[Y_n \mid X](X_n - \bar{X})}{\sum_n X_n (X_n - \bar{X})} \\
&= \frac{\sum_n (\alpha + \beta X_n)(X_n - \bar{X})}{\sum_n X_n (X_n - \bar{X})} \\
&= \frac{\alpha \sum_n (X_n - \bar{X}) + \beta \sum_n X_n (X_n - \bar{X})}{\sum_n X_n (X_n - \bar{X})} \\
&= \frac{\beta \sum_n X_n (X_n - \bar{X})}{\sum_n X_n (X_n - \bar{X})} \\
&= \beta.
\end{aligned}$$

It then follows from the *law of iterated expectation*[5] that

$$E[\hat{\beta}_{\text{OLS}}(X, Y)] = \beta.$$

Then, for the intercept, we have

$$\begin{aligned}
E[\hat{\alpha}_{\text{OLS}}(X, Y) \mid X] &= E[\bar{Y} - \hat{\beta}_{\text{OLS}}(X, Y)\bar{X} \mid X] \\
&= E[\bar{Y} \mid X] - E[\hat{\beta}_{\text{OLS}}(X, Y) \mid X]\bar{X} \\
&= E\left[\frac{1}{N} \sum_n Y_n \,\middle|\, X \right] - \beta \bar{X} \\
&= E\left[\frac{1}{N} \sum_n (\alpha + \beta X_n + \epsilon_n) \,\middle|\, X \right] - \beta \bar{X} \\
&= \frac{1}{N} \sum_n E[\alpha + \beta X_n + \epsilon_n \mid X] - \beta \bar{X} \\
&= \frac{1}{N} \sum_n \alpha + \frac{\beta}{N} \sum_n X_n + \frac{1}{N} \sum_n E[\epsilon_n \mid X] - \beta \bar{X} \\
&= \alpha + \beta \bar{X} - \beta \bar{X} \\
&= \alpha.
\end{aligned}$$

As with the slope, this conditional expectation gives us the unconditional expectation we want:

$$E[\hat{\alpha}_{\text{OLS}}(X, Y)] = \alpha.$$

To sum up: as long as the crucial condition $E[\epsilon_n \mid X_1, \ldots, X_N] = 0$ holds, then OLS is an unbiased estimator of the parameters of the linear model.

[5]For random variables A and B, $E[f(A, B)] = E_A[E_B[f(A, B) \mid A]] = E_B[E_A[f(A, B) \mid B]]$.

Another important property of OLS is that it is *consistent*. Informally, this means that in sufficiently large samples, the OLS estimates $(\hat{\alpha}_{\text{OLS}}, \hat{\beta}_{\text{OLS}})$ are very likely to be close to the true parameter values (α, β). Another way to think of consistency is that, as $N \to \infty$, the bias and variance of the OLS estimator both go to zero.[6]

Of course the bias "goes to" zero, since OLS is unbiased. The real trick to proving consistency is to show that the variance goes to zero. If you wanted to do that for the slope estimate, you'd derive an expression for

$$V[\hat{\beta}_{\text{OLS}}] = E[(\hat{\beta}_{\text{OLS}} - E[\hat{\beta}_{\text{OLS}}])^2] = E[(\hat{\beta}_{\text{OLS}} - \beta)^2]$$

and show that

$$\lim_{N \to \infty} V[\hat{\beta}_{\text{OLS}}] = 0.$$

This takes more algebra than we have time for, so I leave it as an exercise for the reader.

5.5 Appendix: Regression in R

We will be using the **tidyverse** package as always, the **car** package for the `Prestige` data, and the **broom** package for its convenient post-analysis functions.

```
library("tidyverse")
library("car")
library("broom")
```

Let's take a look at `Prestige`, which records basic information (including perceived prestige) for a variety of occupations.

```
head(Prestige)
```

```
##                     education income women prestige census type
## gov.administrators      13.11  12351 11.16     68.8   1113 prof
## general.managers        12.26  25879  4.02     69.1   1130 prof
## accountants             12.77   9271 15.70     63.4   1171 prof
## purchasing.officers     11.42   8865  9.11     56.8   1175 prof
## chemists                14.62   8403 11.68     73.5   2111 prof
## physicists              15.64  11030  5.13     77.6   2113 prof
```

Suppose we want to run a regression of prestige on education. We will use the `lm()` function, which stands for *linear model*. This will employ the "formula" syntax that you previously saw when faceting in ggplot. The basic syntax of a formula is `response ~ covariate`,

[6] What I am describing here is *mean square consistency*, which is stronger than the broadest definitions of consistency in statistical theory.

where `response` and `covariate` are the names of the variables in question. In this case, with `prestige` (note that the variable is lowercase, while the dataset is capitalized) as the response and education as the covariate:

```
lm(prestige ~ education, data = Prestige)
```

```
##
## Call:
## lm(formula = prestige ~ education, data = Prestige)
##
## Coefficients:
## (Intercept)    education
##      -10.73         5.36
```

You'll notice that didn't give us very much. If you've previously used statistical programs like Stata, you might expect a ton of output at this point. It's all there in R too, but R has a different philosophy about models. R sees the fitted model as an object in its own right—like a data frame, a function, or anything else you load or create in R. Therefore, to analyze regression results in R, you will typically save the regression results to a variable.

Like any other variable, you'll want to give your regression results meaningful names. I typically call them `fit_` to indicate a fitted model, followed by some memorable description.

```
fit_educ <- lm(prestige ~ education, data = Prestige)
```

When you do this, the output doesn't get printed. To see the default output, just run the variable name, just like you would to see the content of a data frame:

```
fit_educ
```

```
##
## Call:
## lm(formula = prestige ~ education, data = Prestige)
##
## Coefficients:
## (Intercept)    education
##      -10.73         5.36
```

For a more detailed readout, use the `summary()` method:

```
summary(fit_educ)
```

```
##
## Call:
## lm(formula = prestige ~ education, data = Prestige)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -26.040  -6.523   0.661   6.743  18.164
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -10.732      3.677   -2.92   0.0043
## education      5.361      0.332   16.15   <2e-16
##
## Residual standard error: 9.1 on 100 degrees of freedom
## Multiple R-squared:  0.723,  Adjusted R-squared:  0.72
## F-statistic:  261 on 1 and 100 DF,  p-value: <2e-16
```

This prints out a whole boatload of information, including inferential statistics that we're going to wait until later in the course to discuss how to interpret:

- The model you ran
- Basic statistics about the distribution of the residuals
- For each coefficient:
  - Parameter estimate
  - Standard error estimate
  - Test statistic for a hypothesis test of equality with zero
  - p-value associated with the test statistic
- $\hat{\sigma}$ (called the "residual standard error", a term seemingly unique to R)
- R^2 and an "adjusted" variant that accounts for the number of variables in the model
- F statistic, degrees of freedom, and associated p-value for a hypothesis test that every coefficient besides the intercept equals zero

Strangely, `summary()` doesn't give you the sample size. For that you must use `nobs()`:

```
nobs(fit_educ)
```

```
## [1] 102
```

You can use a fitted model object to make predictions for new data. For example, let's make a basic data frame of education levels.

```
my_data <- data_frame(education = 8:16)
my_data
```

```
## # A tibble: 9 × 1
##   education
##       <int>
## 1         8
## 2         9
## 3        10
## 4        11
## 5        12
## 6        13
## 7        14
## 8        15
## 9        16
```

To calculate the predicted level of prestige for each education level, use `predict()`:

```
predict(fit_educ, newdata = my_data)
```

```
##      1      2      3      4      5      6      7      8      9
## 32.155 37.516 42.877 48.238 53.599 58.959 64.320 69.681 75.042
```

When using `predict()`, it is crucial that the `newdata` have the same column names as in the data used to fit the model.

You can also extract a confidence interval for each prediction:

```
predict(fit_educ,
        newdata = my_data,
        interval = "confidence",
        level = 0.95)
```

```
##      fit    lwr    upr
## 1 32.155 29.615 34.695
## 2 37.516 35.393 39.639
## 3 42.877 41.024 44.730
## 4 48.238 46.441 50.034
## 5 53.599 51.627 55.571
## 6 58.959 56.632 61.287
## 7 64.320 61.525 67.116
## 8 69.681 66.353 73.010
## 9 75.042 71.142 78.942
```

One of the problems with `summary()` and `predict()` is that they return inconveniently shaped output. The output of `summary()` is particularly hard to deal with. The **broom** package provides three utilities to help get model output into shape. The first is `tidy()`, which makes a tidy data frame out of the regression coefficients and the associated inferential statistics:

```
tidy(fit_educ)
```

```
##          term estimate std.error statistic    p.value
## 1 (Intercept) -10.7320   3.67709   -2.9186 4.3434e-03
## 2   education   5.3609   0.33199   16.1478 1.2863e-29
```

The second is `glance()`, which provides a one-row data frame containing overall model characteristics (e.g., R^2 and $\hat{\sigma}$):

```
glance(fit_educ)
```

```
##   r.squared adj.r.squared  sigma statistic    p.value df logLik    AIC
## 1    0.7228       0.72003 9.1033    260.75 1.2863e-29  2   -369 744.01
##      BIC deviance df.residual
## 1 751.88     8287         100
```

The third is `augment()`, which "augments" the original data—or new data you supply, as in `predict()`—with information from the model, such as predicted values.

```
# Lots of output, so only printing first 10 rows
head(augment(fit_educ), 10)
```

```
##              .rownames prestige education .fitted .se.fit  .resid     .hat
## 1   gov.administrators     68.8     13.11  59.549 1.19689  9.2509 0.017287
## 2     general.managers     69.1     12.26  54.992 1.03332 14.1076 0.012885
## 3          accountants     63.4     12.77  57.726 1.12584  5.6736 0.015295
## 4  purchasing.officers     56.8     11.42  50.489 0.92936  6.3108 0.010422
## 5            chemists     73.5     14.62  67.644 1.57269  5.8559 0.029846
## 6           physicists     77.6     15.64  73.112 1.86034  4.4879 0.041763
## 7           biologists     72.6     15.09  70.164 1.70291  2.4363 0.034993
## 8           architects     78.1     15.44  72.040 1.80254  6.0600 0.039208
## 9      civil.engineers     73.1     14.52  67.108 1.54561  5.9920 0.028827
## 10    mining.engineers     68.8     14.64  67.751 1.57814  1.0487 0.030053
##    .sigma    .cooksd .std.resid
## 1  9.1010 0.00924267    1.02511
## 2  9.0372 0.01587881    1.55981
```

```
## 3  9.1311 0.00306360    0.62807
## 4  9.1269 0.00255745    0.69688
## 5  9.1296 0.00656112    0.65310
## 6  9.1375 0.00552702    0.50362
## 7  9.1458 0.00134578    0.27244
## 8  9.1280 0.00941104    0.67914
## 9  9.1287 0.00662109    0.66793
## 10 9.1485 0.00021198    0.11697
```

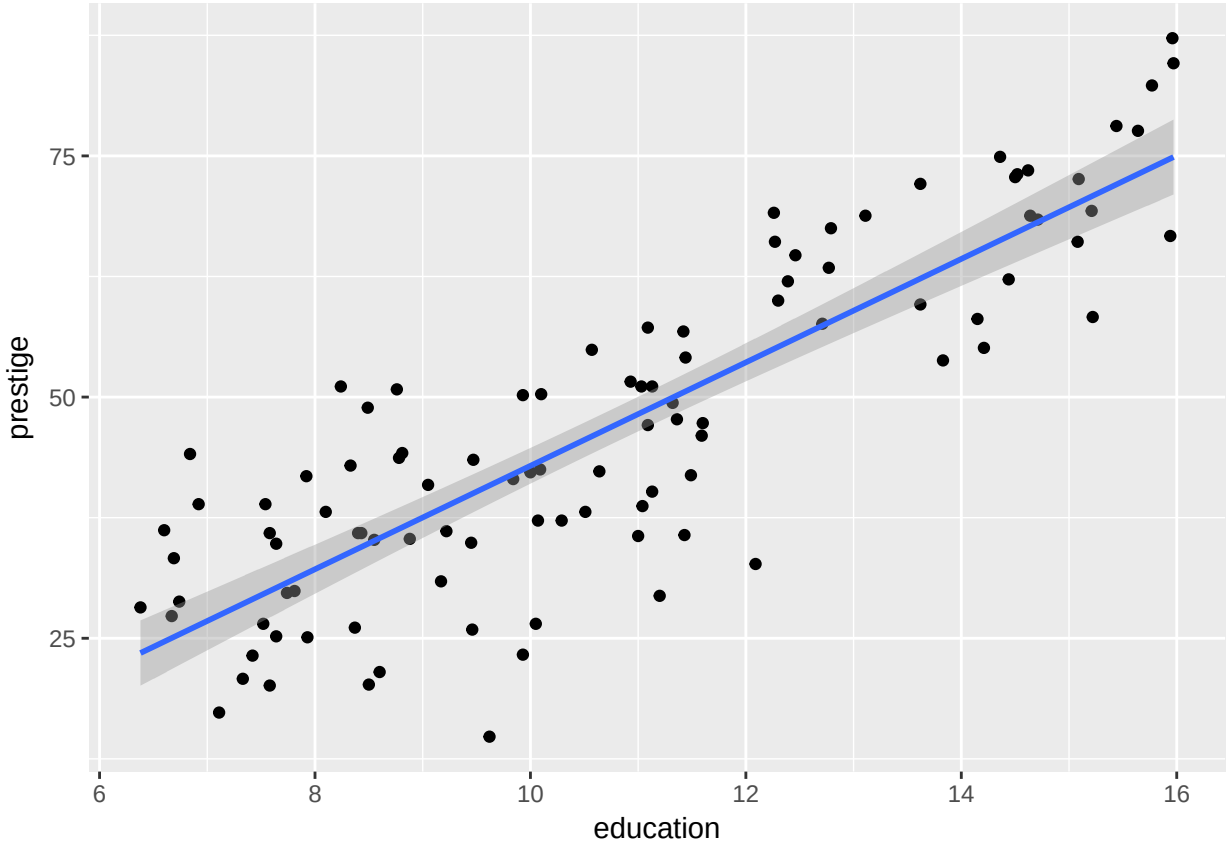
```
augment(fit_educ,
        newdata = my_data)
```

```
##   education .fitted .se.fit
## 1         8  32.155 1.28013
## 2         9  37.516 1.07023
## 3        10  42.877 0.93407
## 4        11  48.238 0.90555
## 5        12  53.599 0.99397
## 6        13  58.959 1.17319
## 7        14  64.320 1.40897
## 8        15  69.681 1.67763
## 9        16  75.042 1.96574
```

Notice that you get back more information for the data used to fit the model than for newly supplied data. The most important is `.fitted`, the predicted value. See `?augment.lm` for what all the various output represents.

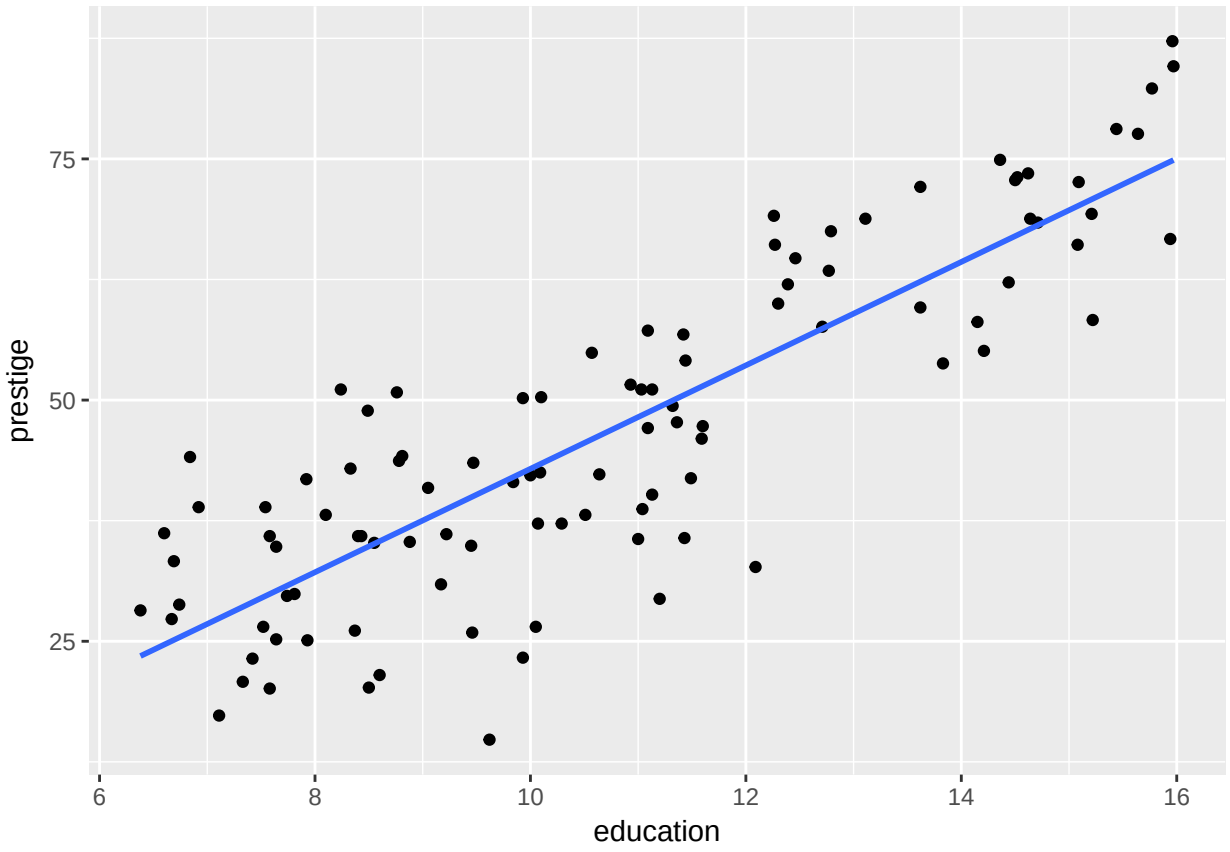
One last note on plotting regression lines with ggplot. Use `geom_smooth(method = "lm")`.

```
ggplot(Prestige, aes(x = education, y = prestige)) +
  geom_point() +
  geom_smooth(method = "lm")
```



To get rid of the confidence interval:

```
ggplot(Prestige, aes(x = education, y = prestige)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```


prestige
75
50
25
6
8
10
12
14
16
education

Bibliography

Bowers, J. (2011). Six Steps to a Better Relationship with Your Future Self. *The Political Methodologist*, 18(2):2–8.

Leek, J. (2015). *The Elements of Data Analytic Style*. Leanpub.

Wickham, H. (2014). Tidy Data. *Journal of Statistical Software*, 59(10).

Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K. D., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P., and Wilson, P. (2014). Best Practices for Scientific Computing. *PLOS Biology*, 12(1):e1001745.