Scraping and Cleaning Data From the Web
- Goal
- The Data
- Reading the Data from the Web
- Extracting the tables
- Converting the table to a data frame
- Cleaning the data
- Refining the plot: Customizing Legends
- Some Simple Text Manipulation

Code

STAT 209: Lab 11

Scraping and Cleaning Data From the Web

Goal

Use a reproducible workflow to extract data tables from HTML, clean it, and make some visualizations. This will involve the following steps:

1. Pulling the HTML from the site
2. Extracting the tables
3. Converting a table to a data frame
4. Fixing variable formats (quantitative data is numeric, dates are parsed)
5. Visualization

Moreover we want to be able to do all of these steps with R code so that our visualization is fully *reproducible*. That means no manual editing, copy-pasting, or Excel fixes.

The Data

We'll look at the data on the all time biggest grossing opening weekends for movies Box Office Mojo (http://www.boxofficemojo.com/alltime/weekends)

Load the tidyverse:

Code

Reading the Data from the Web

The `rvest` package has a `read_html()` function that reads HTML directly.

Code:

Code

Well, we've got the HTML into R. That's step 1 down. But this isn't in a format that we can use yet.

Extracting the tables

The HTML is structured as a collection of "nodes" (think page elements: text blocks, tables, etc.), which are arranged hierarchically in a tree (nodes are composed of other nodes). We don't need to know the fine details, but the main point is that somewhere within the mess of HTML that we just read in are some tables that can be converted into data frames in R. Fortunately, the good people behind `rvest` have figured out how to do this for us, and so all we need to do is call their nice function `html_nodes()` to get the tables.

Code:

Code

Looks like there are six tables at that URL. By trial and error, we can discover that the table we care about, with the actual box office data, is fifth in this list.

A piece of R arcana (Rcana?): the usual square bracket notation for extracting a subset that we use to get an entry in a particular position in a vector (think `myvariable[3]` to get the third entry in `myvariable`) doesn't quite work as expected when the data structure we start with is a `list` instead of a `vector`.

When we start with a list:

1. Single square brackets (`mylist[3]` or `mylist[1:3]`) return a *sub-list*
2. Double square brackets (`mylist[[3]]`) return a single element
3. This is a syntax error: `mylist[[1:3]]`

Since `tables` is a list of tables, even though it is only length 1, to get the table out (instead of a list of one table), we could say `tables[[1]]`.

However, for syntactic consistency with the rest of our workflow, I like to use the `extract()` and `extract2()` functions from the `magrittr` package, which enable extraction of elements within a pipeline. These functions are equivalent to the `[]` and `[[]]` syntax, but take the object as the first argument and the subscript (or subscript range in the case of `extract()`) as the second. Using a pipe, this becomes

Code:

Code

OK, we have the table we want. Step 2 done.

Converting the table to a data frame

Obviously this is still not a data frame. But getting one is easy (for us; as always there's a lot of complexity wrapped in these simple functions). Just call `html_table()`.

Code

Now we're getting somewhere!

Aside: Finding the right table

In some cases you will get more than one table in your list of tables. How do we know which one is the one we're trying to get?

We could just converting all of the tables in the list and see which one is what we're looking for. We can do this efficiently by calling `lapply()`.

Below is some code to extract the tables from the Wikipedia page listing all songs recorded by the Beatles.

Code:

Code

Exercise 1 Use `lapply()` to take our original list of tables (`tables`), and produce a list of data frames. Then use `lapply()` again to get the dimensions of each table. Based on the dimensions, identify which one is the big table under the "Main Songs" section on the page.

Sample solution:

Code

Cleaning the data

Fixing variable names

Let's look at the variable names in the box office data set.

Code:

Code

Most of these are … not ideal: They're full of spaces and special characters that are making me anxious. Let's fix them with a `rename()` operation.

Code:

Code

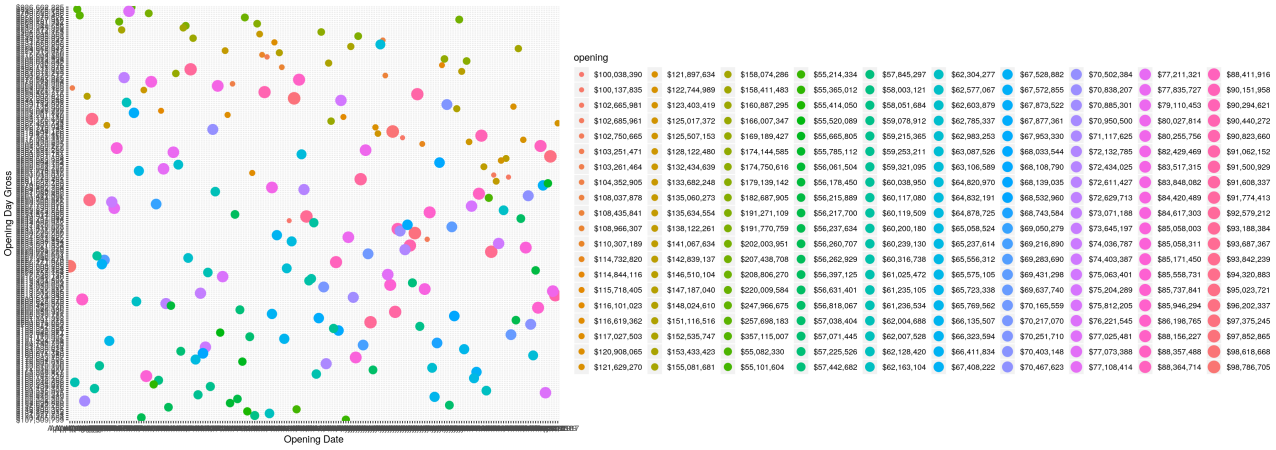
Much better. I can breathe now.

Plotting: Attempt 1

Ok, let's try a plot. Unfortunately the `% of Total` variable is no longer available (it used to be there), so we can't tell how dominant the record movies were in the year they were released due to changes in the total number of movies being released. But we'll go ahead plot the total gross box office receipts for each movie against the date when that movie opened using a `geom_point()`, to look at the trend over time, and we'll map the movie's opening weekend haul to the size and color of the dot. The scatterplot will to get a sense of whether the big blockbusters have become more financially dominant relative to other movies over time.

Code:

Code



Ohhhh noooooooo…. (Note: I had to set `fig.width = 20` and `fig.height = 7` in my chunk options to even get the plot to display)

Fixing variable formats

It looks like many of the variables which we intended to be quantitative, are actually stored as categorical. Take a `glimpse()` at the dataset to confirm this:

Code

Yup: `opening`, `total_gross`, and `date` are stored as type `<chr>`; in other words, as text strings.

No problem. We've seen this issue before. We can use `parse_number()` to fix it.

Exercise 2 Use `parse_number()` together with the appropriate wrangling verb to convert all variables that should be quantitative into numbers. (Skip `date` for now; we'll have to handle that separately)

Sample Solution:

Code

That's better.

Handling the dates

The dates in the `date` variable are written in `MM/DD/YYYY` format. Perfectly understandable by a person, but not by `ggplot` by default. The `lubridate` package provides a function called `mdy()` that converts text in this form to a date format that `ggplot` can understand and treat as a continuous scale.

Code

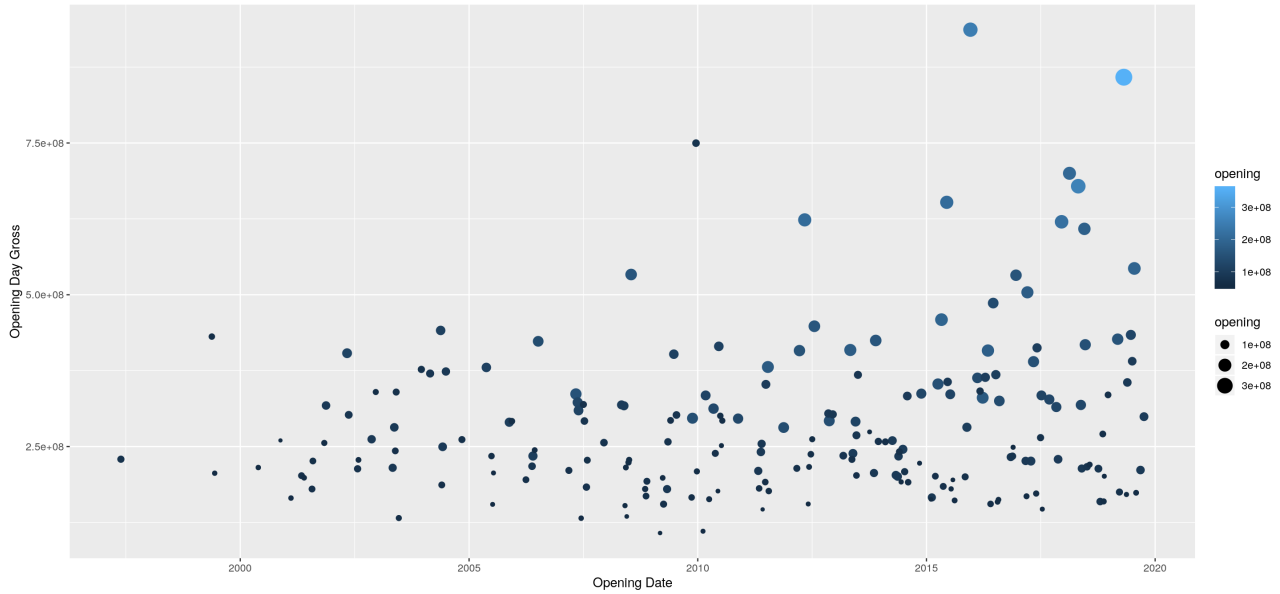
Code

Plotting: Attempt 2

Now the same code we used before will produce a more readable plot.

Code:

Code



Note that since `opening` is now quantitative, `ggplot` uses a sequential color palette, which makes a lot more sense for this variable than the categorical palette it was using before.

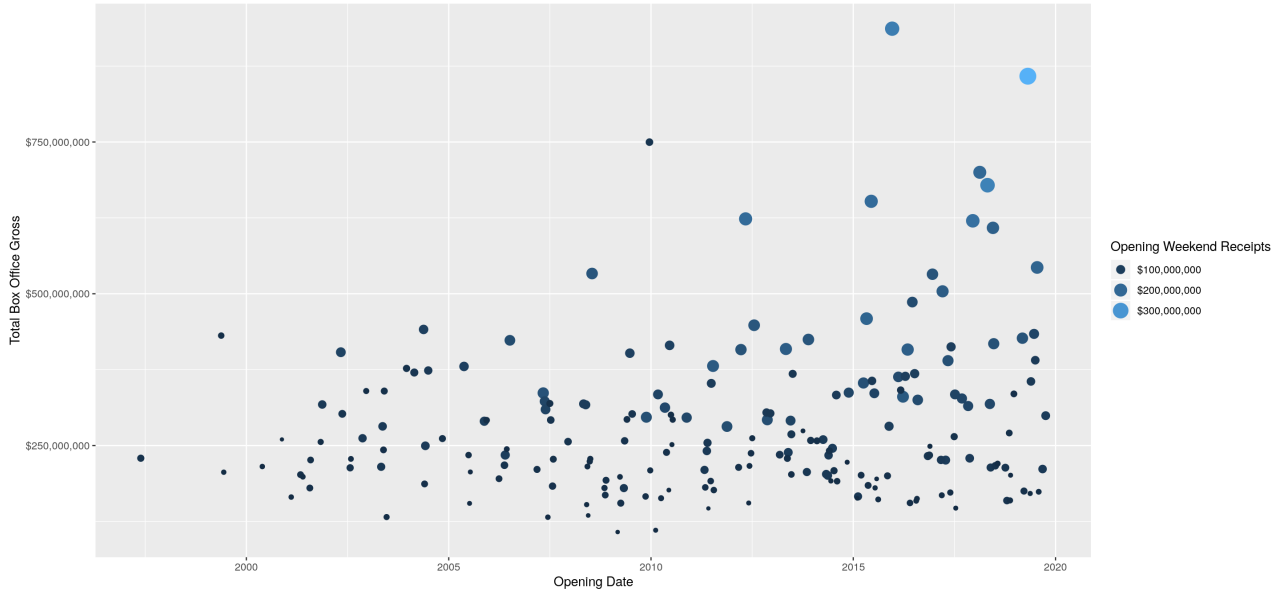
Refining the plot: Customizing Legends

There are still some issues: the dollar amounts are displayed in scientific notation, which is awkward, and the legend shows the variable name that we created because it was nice to work with in code; but it's not so nice on a plot.

We can use the `guides()` element to customize the legend titles, the `scales` library to have the y axis display dollar amounts, and the `scale_x_date()` element to have the x-axis labels format as dates.

To wit: **Code:**

Code



Setting the legend titles this way has the nice effect that the two legends get merged into one, which since color and size correspond to the same variable, is good.

And we're done!

Exercise 3

A shortcoming of this plot is that the meaning of a dollar is not constant over time. Some of the increases in returns are attributable to simple inflation. Fortunately, we can adjust for inflation. The website at `https://inflationdata.com/Inflation/Consumer_Price_Index/HistoricalCPI.aspx` contains a table showing the consumer price index by month since 1913. To find out how many of todays dollars are equivalent to $1 at a given point in the past, we can find the ratio between the CPI now and the CPI in the year when the data comes from. If we multiply a past dollar amount by this ratio, we can express it in terms of today's dollars. Scrape the inflation data from the site and use it together with the relevant join and mutate operations to convert the box office returns to today's dollars (for simplicity you can just use the year and the January column). In order to match up the dates, you'll need to extract the year from the release date column of the box office data. You can use the `year()` function from the `lubridate` package (included already in `tidyverse`) to do this. Send me and Chris I a Slack DM with your code chunk, and post your plot in `#lab11`.

Exercise 4

Here (https://www.the-numbers.com/box-office-records/domestic/all-movies/weekend/opening) is some data about the same thing (movie opening weekends), but from a different website (and so some formats may be different), and restricted to U.S. ticket sales. Create two plots using this data: one that is structured the same way as the one above, and one that shows something else you find interesting. Post the second plot in Slack at #lab11.

Some Simple Text Manipulation

Let's revisit the table of Beatles songs:

Code

Some of the fields in this data have odd formatting. For example, the song titles have quotation marks in them. How could we go about cleaning these values to remove the quotes?

To modify each entry in a variable we can use `mutate()` … but what operation will take a string with quotes and return the same string with no quotes? One option is to use the `gsub()` function. The first argument specifies a pattern we want to find in each string. The second argument specifies what we would like to put in place of any substring that matches the first pattern. And the third argument is a vector of strings.

So, to remove quotes (and while we're at it, remove the parentheses from some of the variable names), we can do

Code

Code

Notice that to refer to a quotation mark in a pattern string, we need to precede it with two backslashes. This tells the parser to treat the following character as a literal instance of that character and not use any special syntactic meaning it would normally have.

Let's take a look at how many songs were written by each Beatle.

Code

Early in the bands career they recorded a lot of covers, so we see one song credited to a lot of different people. Let's try to filter the data to include only Beatles originals.

One option would be to simply list all of the different ways that songs are credited that include members of the band, and use an `%in%` operator to filter. But since the members cowrote songs in a number of different combinations, this is a bit awkward. A more elegant solution would be to find the entries where the string includes one of the members' names (Lennon, McCartney, Harrison, or Starkey – Ringo Starr's real name is Richard Starkey).

We can create a filter that checks whether a string contains a particular pattern using the `grepl()` function. For example, to find all songs that Paul McCartney wrote or co-wrote:

Code

To find instances where any of a set of substrings is contained in a string, we can use the following syntax:

Code

Looks like there's one entry credited to "George Harrison", whereas the rest of the credits use only the songwriter's last name. Let's clean this up with a `gsub()`

Code

We'll come back to text manipulation later on and see how to do more sophisticated things, but if you want to learn more on your own about the way patterns are specified for functions like `gsub()` and `grepl()`, you can read the documentation for those two functions, as well as the `regex` help page.

Exercise 5

Find out how many of the Beatles' songs contain personal pronouns such as "I", "you", "me", "my", "your", "she", "he". Out of these, how many also contain the word "love"? (Hint: Use `grepl()` in a `mutate()` instead of a `filter()` to create a binary variable that is `TRUE` if the song title contains (one of) the strings in question, and `FALSE` otherwise, and use the and (`&`) and/or or (`|`) operators to create binary variables that check for conjunctions or disjunctions of other binary variables). Make a bar graph comparing the proportion of song titles containing the word "love" out of those containing a personal pronoun to the proportion containing "love" out of those not containing a personal pronoun. Post your graph to `#lab11`.