Webscraping

Jasmine Kobayashi

09/19/2022

Web Scraping

The data we need to answer a question is not always in a spreadsheet, ready for us to read. For example, the US murders dataset we used in the R Basics module originally comes from this Wikipedia page: https://en.wikipedia.org/wiki/Murder_in_the_United_States_by_state. You can see the data table when you visit the web page.

But, unfortunately, there is no link to a data file. To make the data frame we loaded using `data(murders)`, or reading the csv file made available through `dslabs`, we had to do some *web scraping*.

***Web scraping*, or *web harvesting*, are the terms we use to describe the process of extracting data from a website**. The reason we can do this is because the information used by a browser to render web pages is received as **text** from a server. The text is computer code written in hyper text markup language (HTML). To see the code for a web page you can visit the page on your browser, then you can use the *View Source* tool to see it.

Because this code is accessible, we can download the HTML files, import it into R, and then write programs to extract the information we need from the page. However, once we look at HTML code this might seem like a daunting task. But I will show you some convenient tools to facilitate the process. To get an idea of how it works, here we show a few lines of code from the Wikipedia page that provides the US murders data:

```
p>The 2015 U.S. population total was 320.9 million. The 2015 U.S. overall murder rate per 100,000 inhabi
<h2><span class="mw-headline" id="States">States</span><span class="mw-editsection"><span class="mw-edi
<table class="wikitable sortable">
<tr>
<th>State</th>
<th><a href="/wiki/List_of_U.S._states_and_territories_by_population" title="List of U.S. states and te
<small>(total inhabitants)</small><br />
<small>(2015)</small> <sup id="cite_ref-1" class="reference"><a href="#cite_note-1">[1]</a></sup></th>
<th>Murders and Nonnegligent
<p>Manslaughter<br />
<small>(total deaths)</small><br />
<small>(2015)</small> <sup id="cite_ref-2" class="reference"><a href="#cite_note-2">[2]</a></sup></p>
</th>
<th>Murder and Nonnegligent
<p>Manslaughter Rate<br />
<small>(per 100,000 inhabitants)</small><br />
<small>(2015)</small></p>
</th>
</tr>
<tr>
<td><a href="/wiki/Alabama" title="Alabama">Alabama</a></td>
<td>4,853,875</td>
```

```
<td>348</td>
<td>7.2</td>
</tr>
<tr>
<td><a href="/wiki/Alaska" title="Alaska">Alaska</a></td>
<td>737,709</td>
<td>59</td>
<td>8.0</td>
</tr>
<tr>
```

You can actually see the data! We can also see a pattern of how it is stored. If you know HTML, you can write programs that leverage knowledge of these patterns to extract what we want. We also take advantage of a language widely used to make web pages look "pretty" called Cascading Style Sheets (CSS).

Although we provide tools that make it possible to scrape data without knowing HTML, for data scientists, it is quite useful to learn some HTML and CSS. Not only does this improve your scraping skills but it might come in handy if you are creating a webpage to showcase your work. There are plenty of online courses and tutorials for learning these. Two examples are code academy and WWW3 school

The `rvest` package

The `tidyverse` provides a web harvesting package called `rvest`. The first step in using this package is to import the web page into R. The package makes this quite simple:

```
library(rvest)
url <- "https://en.wikipedia.org/wiki/Murder_in_the_United_States_by_state"

h <- read_html(url)
```

Note that the entire Murders in the US Wikipedia webpage is now contained in `h`. The class of this object is

```
class(h)
```

```
## [1] "xml_document" "xml_node"
```

The `rvest` package is actually more general, it handles XML documents. XML is a general markup language, that's what the ML stands for, that can be used to represent any kind of data. HTML is a specific type of XML specifically developed for representing web pages. Here we focus on HTML documents.

Now, how do we extract the table from the object `h`? if we print `h` we don't really see much:

```
h
```

```
## {html_document}
## <html class="client-nojs" lang="en" dir="ltr">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
## [2] <body class="mediawiki ltr sitedir-ltr mw-hide-empty-elt ns-0 ns-subject  ...
```

When we know that the information is stored in an HTML table, you can see this in this line of the HTML code above `<table class="wikitable sortable">`. For this we can use the following code. The different parts of an HTML document, often defined with a message in between `<` and `>` are referred to as *nodes*. The `rvest` package includes functions to extract nodes of an HTML document: `html_nodes` extracts all nodes of different type and `html_node` extracts the first one. To extract all tables we use:

```
tab <- h %>% html_nodes("table")
```

Now, instead of the entire web page, we just have the html code for the tables:

```
tab
```

```
## {xml_nodeset (5)}
## [1] <table class="wikitable">\n<caption>Legend\n</caption>\n<tbody><tr>\n<th  ...
## [2] <table class="wikitable sortable"><tbody>\n<tr>\n<th data-sort-type="text ...
## [3] <table class="wikitable">\n<caption>Legend\n</caption>\n<tbody><tr>\n<th  ...
## [4] <table class="wikitable sortable"><tbody>\n<tr>\n<th data-sort-type="text ...
## [5] <table class="nowraplinks hlist mw-collapsible mw-collapsed navbox-inner" ...
```

But we want the second table on the page since the first table is the legend that details what the colors mean. Looking at the output above it looks like the table index is [2]. To extract just the second table - the table with the data we are interetsed in - we can type the following:

```
tab <- h %>% html_nodes("table") %>% .[2]
head(tab)
```

```
## {xml_nodeset (1)}
## [1] <table class="wikitable sortable"><tbody>\n<tr>\n<th data-sort-type="text ...
```

```
class(tab)
```

```
## [1] "xml_nodeset"
```

```
# NOW WE NEED TO CONVERT DATA SET TO DATA FRAMES (DONE USING RVEST)
```

We are not quite there yet because this is clearly not a tidy dataset, not even a data frame. In the code above you can definitely see a pattern and writing code to extract just the data is very doable. In fact, `rvest` includes a function just for converting HTML tables into data frames:

```
tab <- tab %>% html_table %>% .[[1]]
class(tab)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

We are now much closer to having a usable data table:

```
tab <- tab %>% setNames(c("state", "population", "murder_manslaughter_total", "murder_total", "gun_murd
head(tab)
```

```
## # A tibble: 6 x 9
##   state      population murder_manslaug~ murder_total gun_murder_total ownership
##   <chr>      <chr>      <chr>            <chr>        <chr>                <dbl>
## 1 Alabama    4,853,875  348              - [a]        - [a]                 48.9
## 2 Alaska     737,709    59               57           39                    61.7
## 3 Arizona    6,817,565  306              278          171                   32.3
## 4 Arkansas   2,977,853  181              164          110                   57.9
## 5 California 38,993,940 1,861            1,861        1,275                 20.1
## 6 Colorado   5,448,819  176              176          115                   34.3
## # ... with 3 more variables: murder_manslaughter_rate <dbl>, murder_rate <chr>,
## #   gun_rate <chr>
```

We still have some wrangling to do. For example, we need to remove the commas and turn characters into numbers. Before continuing with this, we will learn a more general approach to extracting information from web sites.

CSS Selectors

The default look of a webpage made with the most basic HTML is quite unattractive. The aesthetically pleasing pages we see today are made using CSS (Cascading Style Sheets). CSS is used to add style to webpages. The fact that all pages for a company have the same style is usually a result that they all use the same CSS file. The general way these CSS files work is by defining how each of the elements of a webpage will look.

The title, headings, itemized lists, tables, and links for example, each receive their own style including font, color, size, and distance from the margin, among others. To do this CSS leverages patterns used to define these elements, referred to as *selectors*. An example of a pattern we used above is `table` but there are many many more.

So if we want to grab data from a web page and we happen to know a selector that is unique to the part of the page, we can use the `html_nodes` function.

However, knowing which selector can be quite complicated. To demonstrate this we will try to extract the **recipe name**, **total preparation time**, and **list of ingredients** from this guacamole recipe.

Looking at the code for this page, it seems that the task is impossibly complex. However, selector gadgets actually make this possible.

SelectorGadget is piece of software that allows you to interactively determine what css selector you need to extract specific components from the web page.

If you plan on scrapping data other than tables we highly recommend you install it. A Chrome extension is available which permits you to turn on the gadget and then as you click through the page it highlights parts and shows you the selector you need to extract these parts. There are various demos of how to do this.

For the guacamole recipe page we have already done this and determined that we need the following selectors:

```
h           <- read_html("http://www.foodnetwork.com/recipes/alton-brown/guacamole-recipe-1940609")
recipe      <- h %>% html_node(".o-AssetTitle__a-HeadlineText") %>% html_text()
prep_time   <- h %>% html_node(".m-RecipeInfo__a-Description--Total") %>% html_text()
ingredients <- h %>% html_nodes(".o-Ingredients__a-Ingredient+ .o-Ingredients__a-Ingredient .o-Ingredie
```

You can see how complex the selectors are. In any case, we are now ready to extract what we want and create a list:

```
guacamole <- list(recipe,prep_time,ingredients)
guacamole
```

```
## [[1]]
## [1] "Guacamole"
##
## [[2]]
## [1] " 1 hr 20 min"
##
## [[3]]
##  [1] "3 Haas avocados, halved, seeded and peeled"
##  [2] "1 lime, juiced"
##  [3] "1/2 teaspoon kosher salt"
```

```
##  [4] "1/2 teaspoon ground cumin"
##  [5] "1/2 teaspoon cayenne"
##  [6] "1/2 medium onion, diced"
##  [7] "1/2 jalapeno pepper, seeded and minced"
##  [8] "2 Roma tomatoes, seeded and diced"
##  [9] "1 tablespoon chopped cilantro"
## [10] "1 clove garlic, minced"
```

Since recipe pages from this website follow this general layout, we can use this code to create a function that extracts this information:

```
get_recipe <- function(url){
  h           <- read_html(url)
  recipe      <- h %>% html_node(".o-AssetTitle__a-HeadlineText") %>% html_text()
  prep_time   <- h %>% html_node(".m-RecipeInfo__a-Description--Total") %>% html_text()
  ingredients <- h %>% html_nodes(".o-Ingredients__a-Ingredient+ .o-Ingredients__a-Ingredient .o-Ingredi
  return(list(recipe = recipe, prep_time = prep_time, ingredients = ingredients))
}
```

and then use it on any of their webpages:

```
get_recipe("http://www.foodnetwork.com/recipes/food-network-kitchen/pancakes-recipe-1913844")
```

```
## $recipe
## [1] "Pancakes"
##
## $prep_time
## [1] " 22 min"
##
## $ingredients
## [1] "1 1/2 cups all-purpose flour"
## [2] "3 tablespoons sugar"
## [3] "1 tablespoon baking powder"
## [4] "1/4 teaspoon salt"
## [5] "1/8 teaspoon freshly ground nutmeg"
## [6] "2 large eggs, at room temperature"
## [7] "1 1/4 cups milk, at room temperature"
## [8] "1/2 teaspoon pure vanilla extract"
## [9] "3 tablespoons unsalted butter, plus more as needed"
```

There are several other powerful tools provided by `rvest`. For example the functions `html_form`, `set_values`, and `submit_form` permit you to query a web page from R. This is a more advanced topic not covered here.