Web Scraping With R

William Marble[*]

August 11, 2016

There is a wealth of valuable information that is publicly available online, but seems to be locked away in web pages that are not amenable to data analysis. While many organizations make their data easily available for researchers, this practice is still the exception rather than the rule – and, oftentimes, the data we seek may not be centralized or the creators may not have intended to create a database. But researchers are increasingly turning to these useful data sources.

For example, Grimmer (2013) analyzes press releases from members of Congress to study how representatives communicate to their constituents. Similarly, Nielsen & Simmons (2015) use European Union press releases to measure whether treaty ratification leads to praise from the international community. In both cases, these documents were available online, but they were not centralized and their authors did not intend to create a database of press releases for researchers to analyze. Nonetheless, they represent valuable data for important social science questions — if we have a way to put them into a more usable format.

Fortunately, there are many tools available for translating unruly HTML into more structured databases. The goal of this tutorial is to provide an introduction to the philosophy and basic implementation of "web scraping" using the open-source statistical programming language `R`.[1] I think the best way to learn webscraping is by doing it, so after a brief overview of the tools, most of this document will be devoted to working through examples.

1 High-Level Overview: the Process of Webscraping

There are essentially six steps to extracting text-based data from a website:

1. Identify information on the internet that you want to use.
2. If this information is stored on more than one web page, figure out how to automatically navigate to the web pages. In the best case scenario, you will have a directory page or the URL will have a consistent pattern that you can recreate — e.g., `www.somewebsite.com/year/month/day.html`.
3. Locate the features on the website that flag the information you want to extract. This means looking at the underlying HTML to find the elements you want and/or identifying some sort of pattern in the website's text that you can exploit.
4. Write a script to extract, format, and save the information you want using the flags you identified.
5. Loop through all the websites from step 2, applying the script to each of them.
6. Do some awesome analysis on your newly unlocked data!

This tutorial will focus on steps 3 and 4, which are the most difficult part of webscraping.

There is also another, simpler way to do webscraping that I'll show an example of: namely, using **A**pplication **P**rogramming **I**nterfaces (APIs) that some websites make available. APIs basically give you a

[*] Prepared for the political science/international relations Summer Research College at Stanford. Vincent Bauer's excellent slides on web scraping were very helpful in preparing this tutorial (and teaching me web scraping in `R` in the first place). The code used in this tutorial can be downloaded at www.stanford.edu/~wpmarble/webscraping_tutorial/code.R. Feel free to email me with questions or comments at wpmarble@stanford.edu.

[1] Python is another programming language that has excellent capabilities for web scraping — particularly with the `BeautifulSoup` package. However, I focus on `R` because more social scientists tend to be familiar with it than with Python.

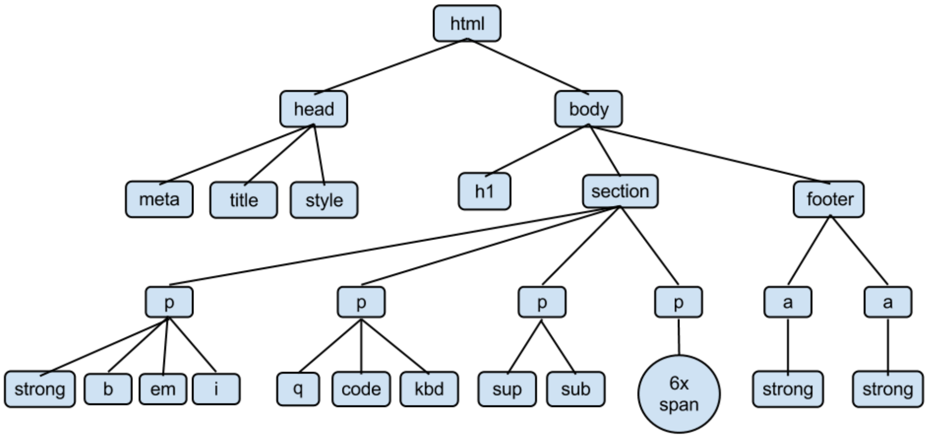


Figure 1: HTML document tree. Source: http://www.openbookproject.net/tutorials/getdown/css/images/lesson4/HTMLDOMTree.png

simple way to query a database and return the data you ask for in a nice format (usually JSON or XML). APIs are great, but aren't usually available, so I don't emphasize them here.

2 Basics of HTML and Identifying the Info We Want

Hypertext Markup Language — or HTML — is a standardized system for writing web pages. Its structure is fairly simple, and understanding its basics is important for successful web scraping.

This is basically what a website looks like under the hood:[2]

```
<!DOCTYPE html>
<html>

  <head>
    <title>This is the title of the webpage</title>
  </head>

  <body>
    <h1>This is a heading</h1>
    <p class="notThisOne">This is a paragraph</p>
    <p class="thisOne">This is another paragraph with a different class!</p>

    <div id="myDivID">
      <p class="divGraf">
        This is a paragraph inside a division, along with a
        <a href="http://stanford.edu">a link</a>.
      </p>
    </div>
  </body>

</html>
```

Figure 1 also provides a visual representation of an HTML tree.

There are several things to note about this structure. First, elements are always surrounded by code that tells web browsers what they are. These tags are opened with triangular brackets `<tag>` and closed with a

[2] To see what this webpage looks like in a browser, go to this link.

slash inside more triangular brackets `</tag>`. Second, these tags often have additional information, such as information about the `class`. Third, these elements are always nested inside other elements. Together, we can use these features to extract the data we want.

It's easy to see the underlying HTML for any webpage: in Chrome, click View → Developer → View Source. This is the first thing you should do when you want to extract data from a webpage. There is also an excellent Chrome add-on called SelectorGadget that allows you to point-and-click the parts of the website that you want to extract. It will automatically tell you what the underlying tags are, and you can copy-paste that into your script.

3 Tools for Webscraping

3.1 `rvest`

How can you select elements of a website in `R`? The `rvest` package is the workhorse toolkit. The workflow typically is as follows:[3]

1. Read a webpage using the function `read_html()`. This function will download the HTML and store it so that `rvest` can navigate it.

2. Select the elements you want using the function `html_nodes()`. This function will take an HTML object (from `read_html`) along with a CSS or Xpath selector (e.g., `p` or `span`) and save all the elements that match the selector. This is where SelectorGadget can be helpful.

3. Extract components of the nodes you've selected using functions like `html_tag()` (the name of the tag), `html_text()` (all text inside the tag), `html_attr()` (contents of a single attribute) and `html_attrs()` (all attributes).

The `rvest` package also has other features that are more advanced — such as the ability to fill out forms on websites and navigate websites as if you were using a browser.

3.2 Regular Expressions

Oftentimes you'll see a pattern in text that you'll want to exploit. For instance, a new variable might always follow a colon that comes after a single word in a new line. Regular expressions (or regex) is a language to precisely define those patterns. They're pretty crucial for webscraping and text analysis. Explaining regex is beyond the scope of this tutorial but I posted a good cheatsheet from AddedBytes.com at this link. In `R`, some regex commands you might need to use:

- `grep(pattern, string)`
  This command takes a string vector and returns a vector of the indices of the string that match the pattern

```
string = c("this is", "a string", "vector", "this")
grep("this", string)

## [1] 1 4
```

- `grepl(pattern, string)`
  This command takes a string vector with length `n` as an input and returns a logical vector of length `n` that says whether the string matches the pattern. Example:

```
grepl("this", string)

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

[3] More information can be found on the GitHub page for rvest.

- `gsub(pattern, replacement, string)`
  This command finds all the instances of `pattern` in `string` and replaces it with `replacement`. Example:

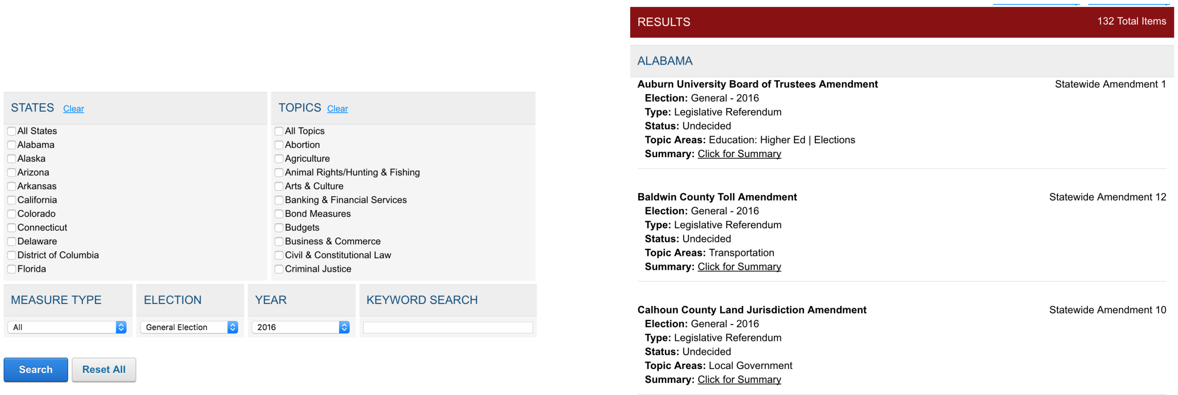
  ```
  gsub(pattern="is", replacement="WTF", string)

  ## [1] "thWTF WTF" "a string"  "vector"    "thWTF"
  ```

4 Simple Example of Webscraping

Let's see what that fake website above looks like in `rvest`. I'll first read in the HTML, then I'll select all paragraphs, then select elements with class "thisOne," then select elements with the ID "myDivID." Finally, I'll extract some text and the link.

```
## First, load required packages (or install if they're not already)
pkgs = c("rvest", "magrittr", "httr", "stringr")
for (pkg in pkgs){
  if (!require(pkg, character.only = T)){
    install.packages(pkg)
    library(pkg)
  }
}
```

```
## Read my example html with read_html()
silly_webpage = read_html("http://stanford.edu/~wpmarble/webscraping_tutorial/html/silly_webpage.html")

# get paragraphs (css selector "p")
my_paragraphs = html_nodes(silly_webpage, "p")
my_paragraphs

## {xml_nodeset (3)}
## [1] <p class="notThisOne">This is a paragraph</p>
## [2] <p class="thisOne">This is another paragraph with a different class! ...
## [3] <p class="divGraf"> \n        This is a paragraph inside a division, ...

# get elements with class "thisOne" -- use a period to denote class
thisOne_elements = html_nodes(silly_webpage, ".thisOne")
thisOne_elements

## {xml_nodeset (1)}
## [1] <p class="thisOne">This is another paragraph with a different class! ...

# get elements with id "myDivID" -- use a hashtag to denote id
myDivID_elements = html_nodes(silly_webpage, "#myDivID")
myDivID_elements

## {xml_nodeset (1)}
## [1] <div id="myDivID"> \n      <p class="divGraf"> \n        This is a p ...

# extract text from myDivID_elements
myDivID_text = html_text(myDivID_elements)
myDivID_text
```



STATES Clear

□ All States
□ Alabama
□ Alaska
□ Arizona
□ Arkansas
□ California
□ Colorado
□ Connecticut
□ Delaware
□ District of Columbia
□ Florida

TOPICS Clear

□ All Topics
□ Abortion
□ Agriculture
□ Animal Rights/Hunting & Fishing
□ Arts & Culture
□ Banking & Financial Services
□ Bond Measures
□ Budgets
□ Business & Commerce
□ Civil & Constitutional Law
□ Criminal Justice

MEASURE TYPE: All
ELECTION: General Election
YEAR: 2016
KEYWORD SEARCH:

(a) Search interface

RESULTS 132 Total Items

ALABAMA

Auburn University Board of Trustees Amendment Statewide Amendment 1
Election: General - 2016
Type: Legislative Referendum
Status: Undecided
Topic Areas: Education: Higher Ed | Elections
Summary: Click for Summary

Baldwin County Toll Amendment Statewide Amendment 12
Election: General - 2016
Type: Legislative Referendum
Status: Undecided
Topic Areas: Transportation
Summary: Click for Summary

Calhoun County Land Jurisdiction Amendment Statewide Amendment 10
Election: General - 2016
Type: Legislative Referendum
Status: Undecided
Topic Areas: Local Government
Summary: Click for Summary

(b) Search results

Figure 2: Screenshots from the NCSL ballot measure database.

```
## [1] " \n       \n        This is a paragraph inside a division, along with a \n        a link.\n

# extract links from myDivID_elements. first i extract all the "a" nodes (as in a href="website.com")
# and then extract the "href" attribute from those nodes
myDivID_link = html_nodes(myDivID_elements, "a") %>% html_attr("href")
myDivID_link

## [1] "http://stanford.edu"
```

Here, I used CSS selectors (class and ID) to extract nodes from the HTML. One thing to note is that to select classes, you put a period before the name of the class — `html_nodes(silly_webpage, ".thisOne")`. To select ID's, put a hashtag in front of the ID you want — `html_nodes(silly_webpage, "#myDivID")`.

5 More Difficult Example

Say we want to know what ballot initiatives will be up for a vote in 2016 in each state. The National Conference of State Legislatures has a nice searchable database of all the initiatives sorted by year, state, and policy topic. It's available at http://www.ncsl.org/research/elections-and-campaigns/ballot-measures-database.aspx and Figure 2 shows screenshots of the database. There's a lot of information here: it has the name of the ballot measure, when it's being voted on, the results, the policy topic areas it covers, and a fairly detailed summary. Unfortunately, it's not easy to download this database and it doesn't return new URL's for each search, meaning it's not easy to loop through searches automatically.

One solution to this is to search for *all* ballot measures in 2016, manually download and save the resulting HTML, then use `R` to extract the info I want. This way, I don't need to figure out how to get `R` to search for me.[4]

5.1 End Product

I want the final result to be a spreadsheet-like object that looks something like this:

[4] This is a reasonable approach when a single search can return all the results I want. If instead we need to perform many searches, we might want to automate it. `rvest` has `html_form()` and related functions to facilitate filling out forms on webpages. Unfortunately this package doesn't work very well in this case; instead, we'd probably need to use other, more complicated tools like the `POST()` command from the `httr` package.

State	Name	Title	Topic Areas	Summary
Alabama	Amendment 1	AU Board of Trustees Amendment	Education	...
Alabama	Amendment 12	Baldwin County Tolls	Transportation	...
⋮	⋮	⋮	⋮	⋮
Wyoming	Amendment A	Non-Permanent Fund Investment	Budgets	...

5.2 Selecting What We Want

With this end goal in mind, how can we get there? First, using SelectorGadget, I can tell there are two types of tags I want to extract:

- The information on ballot measures (except their state) can be extracted using this Xpath selector:
  `//*[contains(concat( " ", @class, " " ), concat( " ", "divRepeaterResults", " " ))]`
- States can be extracted using this Xpath selector:
  `//*[contains(concat( " ", @class, " " ), concat( " ", "h2Headers", " " ))]`

(Don't worry about not understanding that code; I don't either.) With these selectors in hand, I can `read_html()` and then select the parts of the website I want to extract using `html_nodes()`.

```
# STEP 1, OUTSIDE OF R
# Open that webpage on Chrome and search for the relevant set of ballot measures
# (in this case, everything from 2016). Then download the page source.
# I did this and saved it to my website.

# STEP 2
# Use rvest to read the html file
measures = read_html("http://stanford.edu/~wpmarble/webscraping_tutorial/html/ballot_measures_2016.html

# STEP 3
# Select the nodes I want -- I can use the | character to return both types of
# Xpath selectors I want
selector = '//*[contains(concat( " ", @class, " " ), concat( " ", "divRepeaterResults", " " ))]|//*[con
my_nodes = measures %>% html_nodes(xpath=selector)

# let's look at what we got
my_nodes[1:9]

## {xml_nodeset (9)}
## [1] <div class="h2Headers">&#13;\n            States <a id="btnClearAllS ...
## [2] <div class="h2Headers">&#13;\n            Topics<a id="btnClearAllTo ...
## [3] <div class="h2Headers">&#13;\n            Measure Type&#13;\n          ...
## [4] <div class="h2Headers">&#13;\n            Election&#13;\n         </div>
## [5] <div class="h2Headers">&#13;\n            Year&#13;\n         </div>
## [6] <div class="h2Headers">&#13;\n            Keyword Search&#13;\n        ...
## [7] <div class="h2Headers">Alabama</div>
## [8] <div class="divRepeaterResults">&#13;\n                <div class="d ...
## [9] <div class="divRepeaterResults">&#13;\n                <div class="d ...

# the first 6 nodes don't have information I want, so get rid of them
my_nodes = my_nodes[-c(1:6)]
```

5.3 Parsing the Text Information

The text in a node looks like this:

```
\r\n                \r\n                    State-Provided Campaign
Finance System Funded by Imposing a Non-Resident Sales Tax\r\n
\r\n                    Initiative 1464\r\n
\r\n                \r\n                \r\n
\r\n
Election: \r\n                    General\r\n
-\r\n                    2016\r\n                    \r\n
Type: \r\n                    Initiative\r\n
\r\n                    Status: Undecided\r\n
Topic Areas: \r\n                     Ethics/Lobbying/Campaign Finance |
Tax & Revenue\r\n                    \r\n
\r\n                        Summary: Click for Summary\r\n
\r\n                    \r\n
Creates a campaign-finance system; allows residents to direct
state funds to candidates; repeals the
non-resident sales-tax exemption; restricts lobbying employment by
certain former public employees; and
adds enforcement requirements.\r\n                \r\n
\r\n                \r\n
```

Here we see all the information we want, but it's all jumbled up and pasted together into one long string. But there are clearly some clues we can use to split up the string to make it more manageable. Note that the fields are separated by two line breaks, like this: `\r\n \r\n` (with some extra spaces). We can split the string into chunks by using that identifier.[5]

```r
thetext = html_text(my_nodes[[128]])  # randomly chose 128 as an example to work thru

# get rid of all those extra spaces
thetext = gsub(pattern = "[ ]+", replacement = " ", thetext)

# let's split up the string using the "\r\n \r\n" identifier plus the one field that's
# not separated by two line breaks -- topic areas
thetext = strsplit(thetext, split= "\r\n \r\n|\r\n Topic")[[1]]
thetext

##  [1] ""
##  [2] " Corporate Tax Increase"
##  [3] " "
##  [4] " "
##  [5] " \r\n Election: \r\n General\r\n -\r\n 2016"
##  [6] " Type: \r\n Initiative"
##  [7] " Status: Undecided"
##  [8] " Areas: \r\n Business & Commerce | Tax & Revenue"
##  [9] " \r\n Summary: Click for Summary"
## [10] " \r\n Increases annual minimum tax on corporations with Oregon sales of more than $25 million;
## [11] " "
## [12] " "
```

Looking better — now we have each field on its own line, and we can get rid of the extraneous stuff to just leave what we want.

[5] There is one field that isn't separated by this double-line break, "Topic Areas." I handle this below.

```
# get rid of the \r\n,  extra whitespace, and empty entries
thetext = gsub(pattern="\\r|\\n", replacement="", thetext) %>% str_trim
thetext = thetext[thetext != ""]
thetext

## [1] "Corporate Tax Increase"
## [2] "Election:  General - 2016"
## [3] "Type:  Initiative"
## [4] "Status: Undecided"
## [5] "Areas:  Business & Commerce | Tax & Revenue"
## [6] "Summary: Click for Summary"
## [7] "Increases annual minimum tax on corporations with Oregon sales of more than $25 million; impose
```

Home stretch! We can easily figure out what each entry is by using the `grepl()` command.

```
title = thetext[1]
election = thetext[grepl(pattern = "^Election", thetext)] %>%
  gsub("Election:", "", x = .) %>% str_trim
type = thetext[grepl(pattern = "^Type", thetext)] %>%
  gsub("Type:", "", x = .) %>% str_trim
status = thetext[grepl(pattern = "^Status", thetext)] %>%
  gsub("Status:", "", x = .) %>% str_trim
topic_areas = thetext[grepl(pattern = "^Area:|Areas:", thetext)] %>%
  gsub("Area:|Areas:", "", x = .) %>% str_trim

# summary is a little trickier to get because the actual summary comes
# the entry after the one that says "Summary: Click for Summary"
summary_index = grep(pattern="^Summary", thetext) + 1
summary = thetext[summary_index]
```

And we're done! Let's have a look:

```
## title :  Corporate Tax Increase
## election :  General - 2016
## type :  Initiative
## status :  Undecided
## summary :  Increases annual minimum tax on corporations with Oregon sales of more than $25 million;
## topic_areas :  Business & Commerce | Tax & Revenue
```

5.4 Scaling Up

Now that we have the basic infrastructure in place, we can repeat this process for each node using a `for` loop. The only extra thing we need to do is keep track of what state we're in. This is easy to do by first noting which indices of the nodes object contain states (i.e., those with class "h2Headers"). Once we take care of that, we can repeat the exact same code as above and store the results in a data frame.

```
# create state / info indicator vector
state_or_info = my_nodes %>% html_attr("class")
state_or_info = ifelse(state_or_info == "h2Headers", "state", "info")

# set up data frame to store results
results_df = data.frame(state = rep(NA_character_, length(my_nodes)),
                        title = NA_character_,
                        election = NA_character_,
                        type = NA_character_,
                        status = NA_character_,
                        topic_areas = NA,
                        summary = NA_character_,
                        stringsAsFactors = F)


state = NA_character_ # this variable will keep track of what state we're in

# loop through all the nodes
for (i in 1:length(my_nodes)){

  # first see if the node tells us what state we're in; if so, update
  # the state variable
  if (state_or_info[i] == "state") {
    state = html_text(my_nodes[[i]])
  }

  # if it doesn't say what state we're in, apply the parsing code from above
  else {
    results_df$state[i] = state # fill in state

    # parse text like above
    thetext = html_text(my_nodes[[i]])
    thetext = gsub(pattern = "[ ]+", replacement = " ", thetext)
    thetext = strsplit(thetext, split= "\r\n \r\n|\r\n Topic")[[1]]

    thetext = gsub(pattern="\\r|\\n", replacement="", thetext) %>% str_trim
    thetext = thetext[thetext != ""]


    results_df$title[i] = thetext[1]
    results_df$election[i] = thetext[grepl(pattern = "^Election", thetext)] %>%
      gsub("Election:", "", x = .) %>% str_trim
    results_df$type[i] = thetext[grepl(pattern = "^Type", thetext)] %>%
      gsub("Type:", "", x = .) %>% str_trim
    results_df$status[i] = thetext[grepl(pattern = "^Status", thetext)] %>%
      gsub("Status:", "", x = .) %>% str_trim
    results_df$topic_areas[i] = thetext[grepl(pattern = "^Area:|Areas:", thetext)] %>%
      gsub("Area:|Areas:", "", x = .) %>% str_trim

    summary_index = grep(pattern="^Summary", thetext) + 1
    results_df$summary[i] = thetext[summary_index]
  }
}
results_df = results_df[!is.na(results_df$state),]

# let's have a look at a bit of the final product (some variables omitted for space)
print(results_df[1:5, 1:3], row.names = F)

##    state                                         title       election
##  Alabama Auburn University Board of Trustees Amendment General - 2016
##  Alabama                 Baldwin County Toll Amendment General - 2016
##  Alabama    Calhoun County Land Jurisdiction Amendment General - 2016
##  Alabama       County Affairs Administration Amendment General - 2016
##  Alabama                         Impeachment Amendment General - 2016
```

And voilà! We took the data from a mess of HTML and turned it into a nice, spreadsheet-like database that is ready to be used in our analysis.

6 A Very, Very Short Introduction to APIs

A much easier way to download data from websites is through an application programming interface. The idea behind an API is that you send a request to the server using a precisely defined query structure, and the server sends you the data you asked for in a precisely defined format.

With most APIs, you just need to know how to format the URL, and there is documentation telling you what to include. For example, the OMDb API lets you download the IMDb data for any movie or TV show by going to a URL like this: `http://omdbapi.com/?parameter1=value1&parameter2=value2`. The parameter "t" is for title, so if I want to look up information about "Game of Thrones," I can follow this link: http://omdbapi.com/?t=game+of+thrones. (Other possible parameters can be found in the OMDb documentation.)

You'll notice that the information is returned in JSON, which is a data storage format that's often used on websites. We'll have to convert this to an `R` format. So how can we automate webscraping using an API? We'll again use `rvest` and then process the JSON using the package `rjson`.

As a small example, let's look up all the TV shows in the past 10 years that got Emmy nominations for best drama and compare their IMDb ratings.

```
# install rjson
if (!require(rjson)) {install.packages("rjson");library(rjson)}
```

```
list_of_shows = c("breaking bad", "mad men", "game of thrones",
                  "homeland", "house of cards", "true detective",
                  "orange is the new black", "the americans", "mr robot",
                  "boardwalk empire", "the good wife", "dexter",
                  "lost", "true blood", "house", "big love", "downton abbey",
                  "damages", "boston legal", "grey's anatomy", "the sopranos",
                  "heroes", "better call saul")

show_db = data.frame(title = list_of_shows,
                     year = NA, genre = NA, plot = NA, country = NA,
                     awards = NA, metascore = NA, imdbrating = NA,
                     imdbvotes = NA, imdbid = NA, totalseasons = NA)

# construct the url for each show by pasting the name of the show after
# the API base, and encoding using URLencode().
for (show in list_of_shows){
  show_url = paste0("http://omdbapi.com/?&t=", URLencode(show, reserved = T))
  show_info = read_html(show_url) %>% html_text %>% fromJSON

  show_db$year[show_db$title==show] = show_info$Year
  show_db$genre[show_db$title==show] = show_info$Genre
  show_db$plot[show_db$title==show] = show_info$Plot
  show_db$country[show_db$title==show] = show_info$Country
  show_db$awards[show_db$title==show] = show_info$Awards
  show_db$metascore[show_db$title==show] = show_info$Metascore
```

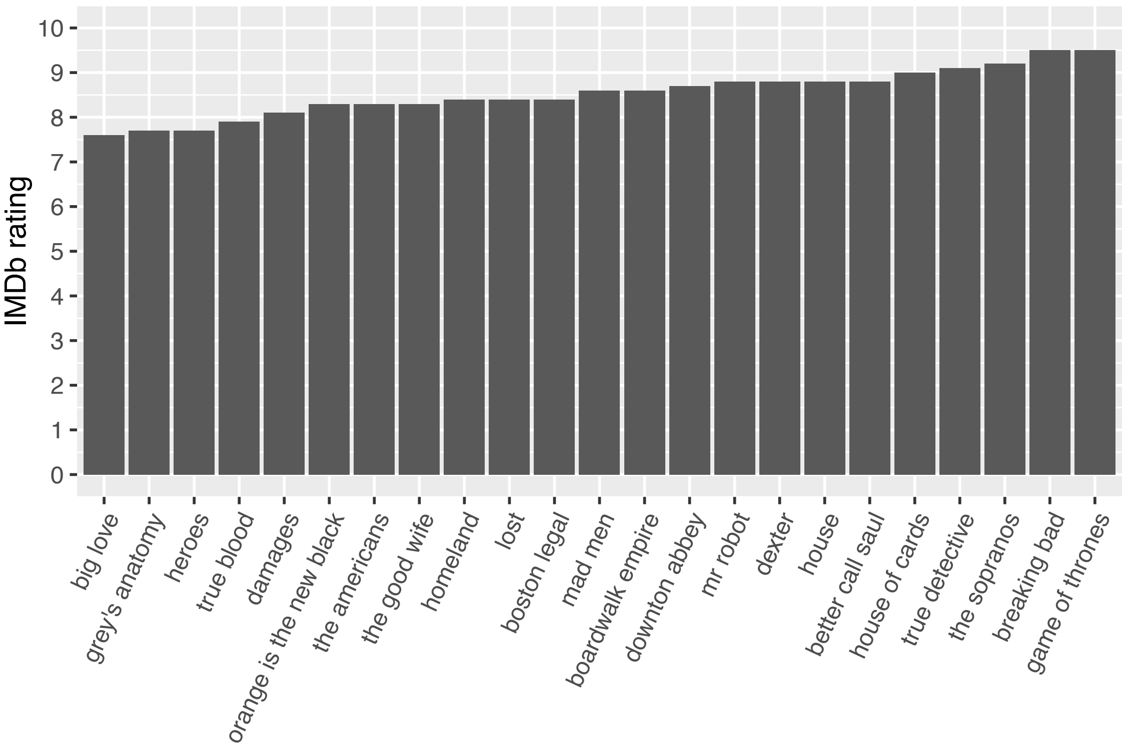


Figure 3: IMDb ratings of Best Drama-nominated TV shows, 2006-2016.

```
  show_db$imdbrating[show_db$title==show] = show_info$imdbRating
  show_db$imdbvotes[show_db$title==show] = show_info$imdbVotes
  show_db$imdbid[show_db$title==show] = show_info$imdbID
  show_db$totalseasons[show_db$title==show] = show_info$totalSeasons
}
show_db[1:5, c(1:3, 8)]

##             title      year                     genre imdbrating
## 1    breaking bad 20082013    Crime, Drama, Thriller        9.5
## 2         mad men 20072015                     Drama        8.6
## 3 game of thrones     2011 Adventure, Drama, Fantasy        9.5
## 4        homeland     2011     Crime, Drama, Mystery        8.4
## 5  house of cards     2013                     Drama        9.0
```

Now again, we have a data frame that's amenable to analysis, like Figure 3. There's a lot more to learn about APIs, so I encourage you to explore more — but many will follow this basic formula: base URL + ? + parameters.

References

Grimmer, Justin. 2013. *Representational Style in Congress: What Legislators Say and Why It Matters.* Cambridge University Press.

Nielsen, Richard A. & Beth A. Simmons. 2015. "Rewards for Ratification: Payoffs for Participating in the International Human Rights Regime?" *International Studies Quarterly* 59(2):197–208.