Optimization and Bootstrapping Methods

Lecture 4

Kevin McAlister

University of Michigan

April 7, 2017

Building Data Sets

Where Does Data Come From?

- All empirical research requires data.
- Relevant to research question.
- Sometimes, the data is already in a format that we can use.
- This generally comes from someone building the data set for us.
- If it exists in a nice format, though, someone has already probably used it to answer questions similar to yours.
- Cutting edge research often requires building original data sets.

Data From The Internet

- We're going to discuss building data sets from data that exists on the web.
- In particular, we're going to talk about building data sets where the amount of data is too large for a cash-strapped graduate student to collect alone.
- Building a data set requires the following steps:
  1. Find a source of data that is relevant to your research questions.
  2. Figure out how to programmatically extract the data from the data source.
  3. Clean the data.
  4. Write the data to a locally accessible disk.
- This process works for data sets that are "moderately inconvenient" in size.
- If the data set is too large to be stored on disk (think all Tweets written over the last week), we have to utilize *streaming* methods.

Data From The Internet

- Let's assume that you've already found a webpage that has all the data you need. It can be a collection of newspaper articles, a set of tables that have the amount spent by school districts in a year, etc.
- How we go about accessing the data depends on how the data is stored and how the data is propagated to the webpage (We'll talk more about this later).
- The first piece of information that we need is the address at which the data is accessible.
- Generally, this is one of two things: a URL or an API endpoint.
- This "address" is the data's unique abode on the vast internet. It uniquely stores a given piece of information.

An Important Distinction

- Alluded to on the last slide, there are two different methods that we're going to discuss for getting data from the internet.
- The first is making a call to an API (Application Programming Interface) endpoint.
- A public-facing API is an interface created by the developers of a website to allow users like you to programmatically pull information from their databases.
- In other words, if there is a public API for a website, the developers are giving you permission to pull information from their webpage.
- An API *endpoint* is often a URL which returns already cleaned data from the website's servers.
- The user tells their computer to make an API request with this URL. In turn, the API accesses the database servers and returns the requested information, typically in a format that is friendly for automated data pulling processes.
- In short, this means that pulling data from the website is easy.

A Quick Aside on Ethics

- Why should we care if we're allowed? If we can see it, we should be able to programmatically pull it as fast as we can, right?
- When we type a URL into a web browser, the web browser makes a request for access to the website's servers. The servers return the code that forms the webpage and the browser parses this and displays what you see on the website.
- Typically, servers are owned by a hosting company and website owners are charged a fee proportional to the number of requests that are made.
- While the per request rate is generally really low, imagine that thousands of people are using a web scraping tool to access the webpage thousands of times. The total can really add up.
- Another problem that arises is that better servers are more expensive. Lower quality servers can handle less requests per second without running into problems. In the same vein, many fast calls to a webpage can cause server outages. We should be cognisant of this limitation.

A Quick Aside on Ethics

- In general, this doesn't matter. As a totally theoretical example, however, imagine that you're interested in helping a wonderful, well-meaning student figure out how to scrape news articles from an online news site in Mozambique. Trying to scrape 50,000 articles simultaneously might take down their servers for a few minutes, which could possibly prevent the site from functioning normally for its thousands of regular consumers. This is probably not all that wonderful ethically.
- In summary, on big-time websites, go hog-wild. Be more aware of your actions when the website may not be as financially stable.

Back to APIs

- Let's learn about APIs by doing stuff in R.
- We're going to start by utilizing the package `library(RCurl)`.
- `curl` is a standard software which allows programmatic pulls of data from a network. `curl` is at the base of almost everything related to gathering data from the web.
- In `RCurl`, there are many functions which allow R to utilize the `curl` protocol that is a UNIX standard. We'll use a few of these functions, but there is much more to the package.

Back to APIs

- The first function of interest is `getURL("url")`. This function tells R to return the source code for this webpage. Let's start by pulling the Wikipedia page on Davey Havok:

  ```
  getURL("https://en.wikipedia.org/wiki/Davey_Havok")
  ```

- This returns nonsense! However, if you scroll through the returned character string, you can see real words about AFI and Blaqaudio (Davey's awful foray into Electronica).
- Simply calling the webpage will return the raw *HTML*. We'll deal with this later.

Back to APIs

- Wikipedia has a public facing API. If we call the API instead of the webpage, we'll get results in a usable data format called *JSON*.
- APIs typically have documentation which details the arguments that can passed to the API. These arguments will dictate what is returned.
- An example Wikipedia API call looks like:

  `https://en.wikipedia.org/w/api.php?`
  `action=opensearch&search=Bee%20%Movie&limit=2&format=json`
- `http://en.wikipedia.org/w/api.php?` is the address for the API. `action=` what kind of query we're making. `search=` what we're searching for. `limit=` how many results to return. `format=` determines the returned format.
- This query searches for Jerry Seinfeld's finest movie and returns the first two search results along with the summary for each result.
- Note that the string %20% is used to indicate a space in API call.

Back to APIs

- APIs generally return data in JSON. JSON is a standardized data format which is akin to a nested list, i.e.:

```
my.list <- list()
my.list[[1]] <- list("a","b")
my.list[[2]] <- list("c","d")
```

- To deal with JSON in R, utilize the package `library(jsonlite)`.
- In `jsonlite`, `fromJSON()` will return JSON results from a URL call and format it into something R can interpret.
- Try running the previous query. This should return a list with 4 entries. Each entry has 2 more entries.

Going Rogue

- What do we do if there is no public facing API?
- Ethical concerns aside, there is another approach.
- We can scrape the Dynamic Web Content that is sent to our browsers when we view a page.
- This information can be pulled programmatically via R and then processed to get the information that we want.
- This, however, takes a lot more time than making a simple API call.
- Similarly, web scraping is only as good as our ability to interpret the returned code.

Going Rogue

- Without an API, we're on our own.
- We need to find the data we want to scrape, figure out how to scrape it, process it, and analyze it all on our own.
- But, there's a lot of rich information that exists on sites that don't have large dev teams.
- Like with an API, web scraping starts with the URL.
- In order to access the information on the webpage, we need to know the link.

An Example

- Let's start with a simple example. Let's scrape the top 200 grossing films of 2016, their earnings, their MPAA ratings, their respective directors, and their respective actors.
- On Canvas in R Code, there's a file with a set of links we'll be using today. Go to the link for the top grossing films of 2016.
- One of the key parts of web scraping is solving the puzzle of how we can programmatically access all the info that we're after.
- How many films are included on the page? Look at the link. How can we change the link to get the other 150 movies?
- This kind of link maneuvering is a key part of web scraping.
- How can I change parts of a URL in a script to pull large amounts of information quickly?

Some Text Functions

- Recall the function `paste()`.
- If we use `paste("cat","dog",sep="")`, R returns `catdog`. `sep=` takes a character argument and uses that as the separator for the inputs of `paste()`. If the input is a vector or list of character strings, use `collapse=` instead.
- The opposite of `paste()` is `strsplit()`. `strsplit("cat dog", " ")` returns a list with a 2 element vector including cat and dog.
- `strsplit()` can split on just about anything. However, some characters have a special meaning in *regular expressions*.
- A common split is on a period. However, we cannot use `"."` as the separator. Instead, we need to use `"[.]"`.

Some Text Functions

- Given a collection character strings, we might want to search over the strings to figure out which ones include a specific word. For this, we can use `grep(pattern,object)`.
- The grep pattern can be a specific character string or a regular expression. The object can be a single string or a vector of strings. grep returns a scalar or vector that says whether or not the object includes the specified pattern.
- Perhaps we want to search for a specific string and replace it with something else. For this, we use `gsub(from, to, obj)`. From is the character string that we're searching for. It can be a string or RegEx. To is what we want to change From to. obj is the object we're searching over.

An Example

- Let's write a for loop that is going to generate the 4 links that we need to collect our info.

```
links <- list()
for(i in 1:4){
    links[[i]] <- paste("http://www.imdb.com/search/title?year=2016,
                        2016&title_type=feature&sort=
                        boxoffice_gross_us,desc&page=",
                        as.character(i),sep="")
}
```

- To do web scraping, we're mostly going to utilize Hadley "Dreamboat" Wickham's package `library(rvest)`.
- Rvest is based on an old web scraping tool "Beautiful Soup" and provides the most intuitive way to work with HTML in R.

An Example

- We'll start by just working with `links[[1]]`.
- `read_html(url)` returns the raw URL on a webpage.

```
first.imdb <- read_html(links[[1]])
```

- Note that `first.imdb` contains two elements and they are compressed. This is for your own good.
- If we want to see what was returned, we can convert the html to html text using `html_text(first.imdb)`.
- This is a mess! That's because HTML is a garbage language that's impossible to read. We need a smart function to parse this nonsense code.

An Example

- HTML, CSS, and XML are all front-end programming languages. They rely on tree structures. Each thing you see on a webpage is part of a tree which is a series of nodes that dictates what is shown on the webpage.
- We can use this tree structure to our advantage.
- We need to figure out which nodes the information we're after are included under.
- Fortunately, there's a really great tool for this called SelectorGadget.
- In Google Chrome, SelectorGadget can be added as an extension. (Add it now so we can use it soon)
- If we use this tool, we can find the node that contains the info that we're after on the page.

An Example

- Let's start by pulling the rankings.
- On the IMDB page, use the SelectorGadget to select the rank next to "Rogue One". Check that only the ranks are selected. Copy the name of this node that is given by the selector gadget.
- We can use the function `html_nodes(html, node)` to subset to HTML included under the specified node:

```
ranks <- html_nodes(first.imdb, ".text-primary")
ranks <- html_text(ranks)
```

- Look at that! We got a vector of numbers!
- We can use this approach to get more complex things from the page.

An Example

- Let's get the titles next.
- Use the Gadget to select the name of the first movie. SelectorGadget selects a lot more than we want. Click on one of the things that it's including that we don't want. It should turn red. Make sure that the titles are yellow. Copy this path.
- Using the same procedure as before, get the titles:

```
titles <- html_nodes(first.imdb, ".lister-item-header a")
titles <- html_text(titles)
```

- Now we've got something more meaningful.
- Use this same process to extract earnings.

An Example

- Let's move on to actors. This is a little more complicated. There are multiple main actors in a movie!
- We don't necessarily want to make an assumption about how many actors are listed for each movie. What if there are less than 4 actors in a film lower on the list?
- Use SelectorGadget to isolate the actors names and the movie name. Use this node and pull the information into R:

```
actors <- html_nodes(first.imdb, ".lister-item-header a ,
                                  .lister-item-content .ghost~ a")
actors <- html_text(actors)
```

- But, now, we have a vector that includes movie names and actors. What do we do?
- Note that the movie title comes first and then the actors before the next movie name.
- We already have the movie titles, so we can use this to map actors to a movie!

An Example

- Let's start by finding which element corresponds to each movie name.

```
title.map.actors <- c()
for(j in 1:length(titles)){
  title.map.actors[j] <- grep(titles[j],actors)
}
```

- Given this mapping, we can find the actors associated with a movie:

```
actors.list <- list()
for(j in 1:length(titles)){
  start <- title.map.actors[j] + 1
  if(j == length(titles)){
    end <- length(actors)
  }else{
    end <- title.map.actors[j+1] - 1
  }
  actors.list[[j]] <- actors[start:end]
}
```

An Example

- This is an extra careful way to ensure that we get associations correct in web scraping. We can repeat a process like this for any other field that we're interested in.
- You can see how to scrape the other fields in the provided R script.
- Because this web site is professionally developed, we know that the HTML nodes are consistent across pages. So, when we want to scrape the other 150 movies, we can use the same HTML tags.
- This means that we can create a function that takes in a page number (1 for 1-50, 2 for 51-100, etc.), and returns an object with all this information.
- This process works for a wide variety of webpages and web objects, but there are exceptions.

JavaScript a.k.a. Your Enemy

- Pages that rely purely on HTML are relatively easy to scrape. We just follow the recipe we created above.
- HTML is not the only way that information is propagated to a page, however.
- A common approach for more dynamic web content is to utilize JS to build an application that makes a call to internal databases and returns the requested information in a widget.
- This makes our life a lot harder, because the information we're after isn't contained in the raw HTML.
- Unfortunately, solving this problem is outside of the scope of this class.

JavaScript a.k.a. Your Enemy

- A promising approach to scraping this kind of material is using PhantomJS, a JS variant that is really good for retrieving this kind of content.
- There are numerous tutorials for this online, but be warned that JavaScript is not an easy language to learn.
- If this is something that you think you're going to rely on in your research, JS is taught in most introductory Software Development courses.