Stat 240 - Lab 06

Dr. Lloyd T. Elliott

March 9, 2022

Now we know how to get data from database using SQL and harvest data using API from web services like Twitter. What if data is on a web page but there is no API? This is where web scraping comes in. Web scraping refers to the process of parsing the HTML source code of a web page in order to extract, retrieve, or scrape some specific data. The goal of this lab is to show you different ways to read data files from the web into R. Hand in this assignment as pdfs through crowdmark, and provide all code for your solutions in the pdfs (you may find it helpful to use knitr or R markdown).

1 Scraping Basic Web Page

The most basic form of web scraping involves grabbing the entire web page and extracting the pieces needed. For basic web scraping tasks, the `readLines()` function from base R will usually suffice (note that `readLines()` can download a URL if it's provided as an argument, similar to `download.file()`). The key argument in the `readLines()` function is the connection which specifies the location of the file (here is just the URL of the web page).

For a single web page, a basic procedure for web scraping in R is as follows:

1. View the source code; get familiar with the HTML tags surrounding the data you want
2. Read the web page source code into R using `readLines()`
3. Clean the data in R

Steps 1 and 2 are usually easy. Step 3 can be extremely time consuming. Below is how we use the `readLines()` function and the page URL to import the web page into R. For this example, we'll use the SFU course outline webpage. The main advantage is that it provides a very simple query system. For example, `https://www.sfu.ca/outlines.html?2022/spring/stat/240/d100` returns a web page listing course outline information for our class, STAT240 section d100 in the Spring 2022 This makes it easy for us to loop through the year, term, discipline, course number and section number to obtain a list of URLs.

```
> course_url =
+   "https://www.sfu.ca/outlines.html?2022/spring/stat/240/d100"
> course_page = readLines(course_url)
```

Sometimes you might see a warning message saying that the last line of the web page didn't contain a newline character. Those messages are safe to ignore.

```
> # check the number of elements in the vector
> length(course_page)
```

The output vector will contain as many elements as number of lines in the read file. There 450ish elements in this vector, each of them representing a line in the HTML file. Let's have a look at an arbitrary set of 10 lines in the page: it is basically text information embedded inside HTML elements, with leading white spaces.

```
> # arbitrary set of 10 lines of html
> course_page[240:250]
```

HTML elements are written with a start tag, an end tag, and with the content in between: `<tagname>content</tagname>`.

- `<h1>, <h2>, ...`: Largest heading, second largest heading, etc.
- `<p>`: Paragraph elements
- `<ul>`: Unordered bulleted list
- `<ol>`: Ordered list
- `<li>`: Individual list item
- `<div>`: Division or section
- `<table>`: Table

It is through these tags that we can start to extract textual components of HTML web pages.

Now we have to focus on what we are trying to extract. The first step is to find where it is. The page is structured into different sections by headings. Let's extract the elements of the largest heading, which should be surrounded by the HTML tag `<h1>`. We can locate this line using the `grep()` function

```
> grep('<h3', course_page)
```

Turn on `value = T` can help us to visually see the result. To extract the index for the line, we can just omit this argument.

```
> grep('<h3', course_page, value = T)

> heading_index = grep('<h3', course_page)
> # note this gives us the same result as using grep(, value = T)
> course_page[heading_index]
>
```

The advantage to extracting the location instead of the value is that we can extract information around that location.

```
> heading = course_page[(heading_index-1):(heading_index+1)]
>
```

What we get are two strings with leading/trailing white spaces. The useful information is embedded in the HTML tags. We can recycle the regular expression from the previous lab to remove the white spaces and HTML tags.

For the HTML tags, we can remove each of them separately, or we can remove all the `<...>` tags at once. The first way is somewhat inefficient, while the second way might cause trouble in some cases. My advice here is to try to remove all the `<...>` tags first. If the unexpected results appears, do some fixes.

Question 1

1. Extract all of the h3 headings, and remove all of the html formatting and any excess white space from all h3 headings, using the following code:

   ```
   > # remove html formatting pieces and replace them
   > #   with white space.
   > (heading3 = grep("<h3",course_page,value=TRUE))
   > (heading3a = gsub("<[^<>]+>", " ", heading3))
   > gsub("\\s{2,}", " ", heading3a)
   ```

   Write similar code to extract the following information from the Stat 240 course website (note that you will want to recycle this in the next question:

2. Course number (i.e. Stat 240) (2pts)

3. Course title (i.e. Introduction to Data Science) (2pts)

4. Instructor (2pts)

5. Course Time + Location (Monday... Burnaby) (2pts)

Question 2

Make sure your scraper from Question 1 works for other webpages of this type by testing it on these courses: Spring 2017 EVSC 100 - d100, Fall 2018 Stat 452, and any of these which are offered this term: (STAT100, 201, 203, 270, 330, 350). Build a data frame with 1 row per course and columns for the different data elements collected in the previous questions. Print the data frame. (1 point per course, 8 points total).

2 Scraping tables

Tables are pretty common in web pages as data sources. We begin by extracting a simple HTML table from a website. For the first example, we will use `http://www.imdb.com/chart`, which is a box-office summary of the top 10 movies along with their gross profits for the current weekend, and their total gross profit. We would like to make a data frame with that information. Of course we can use the method introduced in the previous section to read the page into R, and use the anchor to find the part of the data we want. By doing this, we will have to do extensive data cleaning to extract the information before wrapping it into a data frame. Luckily, R has nice packages that can help to scrape tables from web pages. To read an HTML table we need to use the R package `rvest`, which is a convenient wrapper to parse an HTML page and retrieve the table elements.

```
> library(rvest)
> movie_url = "https://www.imdb.com/chart/boxoffice/"
> movie_table = read_html("https://www.imdb.com/chart/boxoffice/")
> length(html_nodes(movie_table, "table"))
> html_table(html_nodes(movie_table, "table")[[1]])
> html_table(html_nodes(movie_table, "table")[[2]])
```

Often more tables are picked up than we intend. In this case there is some Amazon affiliate data at the bottom of the page that gets picked up as a table. You'll need to look at the content to see what's in the table and also to obtain the correct table. As another example, let's read the Canadian National Parks HTML table from Wikipedia (`https://en.wikipedia.org/wiki/List_of_National_Parks_of_Canada`.

```
> park_url =
+ "https://en.wikipedia.org/wiki/List_of_National_Parks_of_Canada"
> parks =  read_html(park_url)
> length(html_nodes(parks, "table"))
> html_table(html_nodes(parks, "table")[[1]])
> park_table = html_table(html_nodes(parks, "table")[[1]])
```

Question 3

1. Scrape the **Box office performance** and the **Critical and public response** tables for all three phases of the Marvel Cinematic Universe films `https://en.wikipedia.org/wiki/List_of_Marvel_Cinematic_Universe_films`. Obtain those tables as a single R data frame. Hint: The R function *merge* will allow you to make a single data frame by specifying your two tables using SQL behaviour (left join). Try something like this:

   ```
   > merge(x, y, by.x = NameOfMovieNameColumnFromX,
   +    by.y=NameOfMovieNameColumnFromY)
   ```

   You'll need to clean up a couple of rows of data before using such a command (5pts).

2. Make a clean table of the movie name, `Worldwide Box office gross` (make sure it is numeric), *budget* (use net budget and make it numeric), release year (no need for month or day) and the (numeric values) Rotten Tomatoes and Metacritic Scores. Print out the first 10 rows of your data frame. (5pts)

3. Let's look at the moving averages of Box Office Gross Worldwide and Budget over time. Put these two lines on a single plot. For clarity report dollar amounts as log10 dollars. (5pts)

4. What is the distribution of revenue for Marvel movies? Make a plot of the log base 10 difference between Box Office Gross Worldwide and budget for these movies. (3pts)

5. What is the relationship between budget and ratings? On a single plot, show the log base 10 budget and the log base 10 Box office gross vs Rotten Tomatoes score. Include a moving average for ratings with respect to budget (not time). (6pts)

6. How have the ratings evolved over time? Make a plot of ratings over time. (3pts)

Action Required: look at the last few slides of this weeks notes.

For next time: sign up for the 'rcg-shiny-users' mailing list here: `https://www.sfu.ca/information-systems/services/sfu-maillist.html`. This will let you run R Shiny Apps (interactive websites backed by R) on SFU's servers.