Web Scraping in R

Promothesh Chatterjee*

Overview:

We will try to understand how to scrape text data for subsequent analysis. We will understand concepts such as:

- Scraping Politely

- Dealing with HTML/XML

  - Using Developer Tools (Chrome)
  - Using Selector Gadget

- Using all the above to scrape a website and create dataset

Web Scraping

Let's say we want to understand impact of critic's reviews on movie's success. In order to answer this question, we need details about the movie, critics reviews and a bunch of other variables that we want to control for as they might influence our results. Some of this data could be readily available online in a tabular format such as profit figures for the movies, for other details we may need to do some web scraping.

Web scraping refers to the process of automatically extracting data from web pages. Web scraping can have many potential uses in business such as, brand monitoring, comparing prices of competitors, using scraped reviews of product/services for sentiment analysis and other analysis. The process of web scraping is not very easy given that different websites use diverse technologies (i.e. HTML, XML, JSON). We will familiarize ourselves with some basics here regarding how to access the data stored in these technologies using R packages.

Is it Legal to Scrape Web Pages?

The short answer is it depends! Until a few years ago, web scraping was very common, as legal guidelines regarding this were very vague. Because of ubiquity of bots and malware, websites have now become very careful. Before initiating any web-scraping based data collection activity you **must** ensure whether the website legally allows scraping. Check the the "Terms of Service" (typically can be found at the bottom of a web page). Most decent websites also use a file called "robots.txt" which specifies the policies about web scraping and other issues in details. In case of any doubt, please consult a legal professional for advice regarding web scraping permissions.

Scraping "Politely"

We will use an R package to scrape 'politely' (sort of best practices for friendly web scraping).

Located at the website's root level, usually following the domain name (e.g., .com, .org), the robots.txt file provides directives to web crawlers and scrapers regarding the sections of the site they are permitted to collect data from.

For instance, if you go to https://www.rottentomatoes.com/robots.txt , here is what you see:

```
User-agent: *
Disallow: /search
Disallow: /user/id/
Sitemap: https://www.rottentomatoes.com/sitemaps/sitemap.xml
```

The User-agent: * section tells about any software/bot not listed on the file (especially us) whether to allow or disallow certain operations. The presence of a forward slash (/) at the beginning of (dis)allowed paths indicates that they originate from the root directory. For instance, "/search" refers to rottentomatoes.com/search. These lines imply that we are not permitted to extract information from rottentomatoes.com/search or rottentomatoes.com//user/id/.

I have pasted a portion of robots.txt file for the nytimes website. You can see that different bots have been given different privileges.

```
User-agent: Twitterbot
Allow: /*?*smid=

User-Agent: omgilibot
Disallow: /

User-Agent: omgili
Disallow: /

User-agent: ia_archiver
Disallow: /
```

The `polite` R package automates the procedure for us by checking the robots.txt file and allowing to scrape only that which we are allowed to. Since automated web scraping may send too many requests too fast (which may sometime crash the website, termed a Denial-of-Service (DoS) attack), `polite` R package will change the time delay between attempts. For example, the code below specifies a delay of 10 seconds so `polite` will automatically adjust the scraping speed.

```
‘‘‘
User-agent: *
Crawl-delay: 10
‘‘‘
```

Let's test some basic commands of the library before we go into details of web scraping.

```
library(polite)
bow("https://www.rottentomatoes.com")
```

```
## <polite session> https://www.rottentomatoes.com
##     User-agent: polite R package
```

```
##     robots.txt: 10 rules are defined for 9 bots
##    Crawl delay: 5 sec
##   The path is scrapable for this user-agent
```

Each scraping session starts with `bow()`, which tells us if scraping is permitted on the site or not. Basically, R is going through the robots.txt for us.

Let's scrape a website

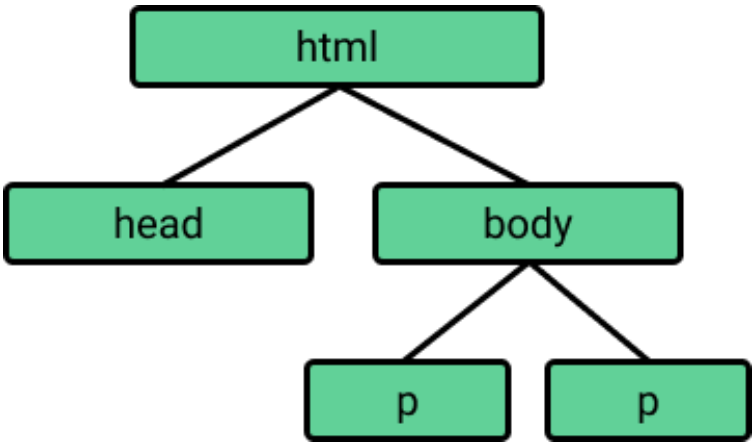
Let's start with a relatively simple Wikipedia page on consumer behavior: https://en.wikipedia.org/wiki/Consumer_behaviour

We will use the `rvest` package, (part of the tidyverse world) as it is probably the best R package for scraping.

```
library(polite)
library(rvest)

wiki_cb <-
  bow("https://en.wikipedia.org/wiki/Consumer_behaviour") %>%
  scrape()
```

While the browser we use displays the content to us, but under the hood there is lot of HTML or XML code which we need to deal with when scraping the websites for content. Typical websites have the HTML or XML code in this tree like structure:



From: https://www.dataquest.io/blog/web-scraping-in-r-rvest/

The easiest way to deal with this is to use *Developer Tools* in the Chrome browser. Let's first look at the source code of the website of interest.

```
wiki_cb # to test if we have properly downloaded the webpage
```

```
## {html_document}
## <html class="client-nojs vector-feature-language-in-header-enabled vector-feature-language-in-main-pa
```

```
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
## [2] <body class="skin-vector skin-vector-search-vue mediawiki ltr sitedir-ltr ...
```

As you can see, the output suggests that wiki_cb is an HTML document. Text content stored within HTML elements is typically indicated by a start tag and an end tag:content. These tags are pre-defined by HTML and thus, useful to work with because of the predictable structure. XML documents are like HTML but more complex (however rvest package has functions to deal with XML as well).

```
<h1>, <h2>,...,<h6>: 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
```

HTML code is organized hierarchically, where the highest level there is a $< html >$ tag. The next level has $< head >$ and $< body >$ tags, and within the $< body >$ of the webpage there are various elements separated by $< div >$ tags. This nested structure works pretty much like lists in R, thus, we can access elements under specific nodes. For instance, we some text content can be stored like this:

```
<p>
Text content within website
is stored like this in HTML
</p>
```

Finding Content from the HTML code

```
wikitext<-wiki_cb %>%
html_nodes("p") %>% #from rvest package
html_text()

wikitext[2] # output from p2
```

```
## [1] "Consumer behaviour is the study of individuals, groups, or organisations and all the activities
```

Similarly we can access say the secondary heading texts under h2

```
wikihead<-wiki_cb %>%
html_nodes("h2") %>% #from rvest package
html_text()

head(wikihead)
```

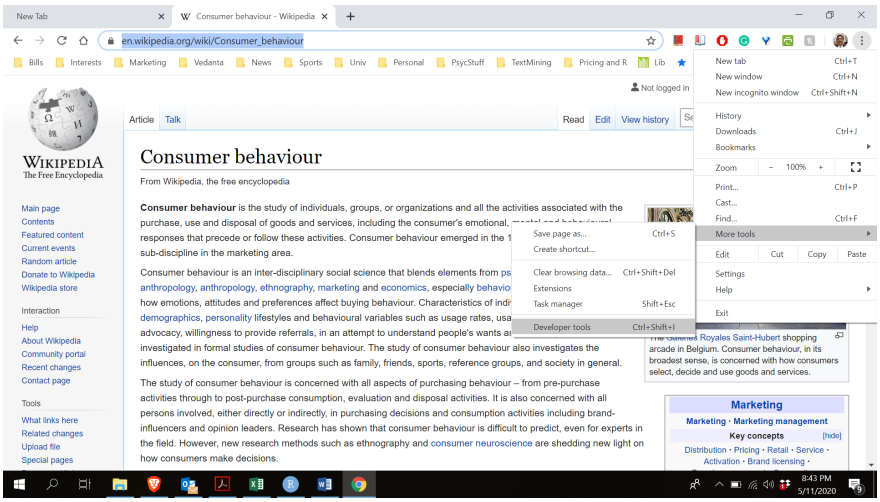
```
## [1] "Contents"
## [2] "Origins of consumer behaviour[edit]"
## [3] "Definition and explanation[edit]"
## [4] "The purchase decision and its context[edit]"
## [5] "Influences on purchase decision[edit]"
## [6] "Consumer decision styles[edit]"
```

To access lists

```
ul_wiki <- wiki_cb %>%
        html_nodes("ul") %>%
        html_text()
ul_wiki[2]
```

```
## [1] "HelpLearn to editCommunity portalRecent changesUpload file\n\t\t"
```

Using Developer Tools from Chrome Typically, the content that we are interested in is hidden under bunch of code in the tree structure. We can use a useful function from Chrome browser's "Developer Tools". You can access the "Developer Tools" like this:



Developer Tool gives us an interface to view the webpage alongside the source code. One can right click at a particular place on the website and choose to "inspect", Developer Tool will point the part of the source code where the wanted content is stored. However, as we have previously seen in the HTML tree, the text is generally stored under the node "p", so we will first select the downloaded page, then use html_nodes function from rvest package to point to location "p" and finally select the text stored there with the function html_text. Once you have extracted the text, then one can do the usual preprocessing. Generally, text is accessible fairly easily from a website like Wikipedia but even here if you want to access some other element, it can get complicated very quick.

Say, I want to access a particular table. There are two approaches to accessing such elements:

1) Using xpath (from Developer Tools)

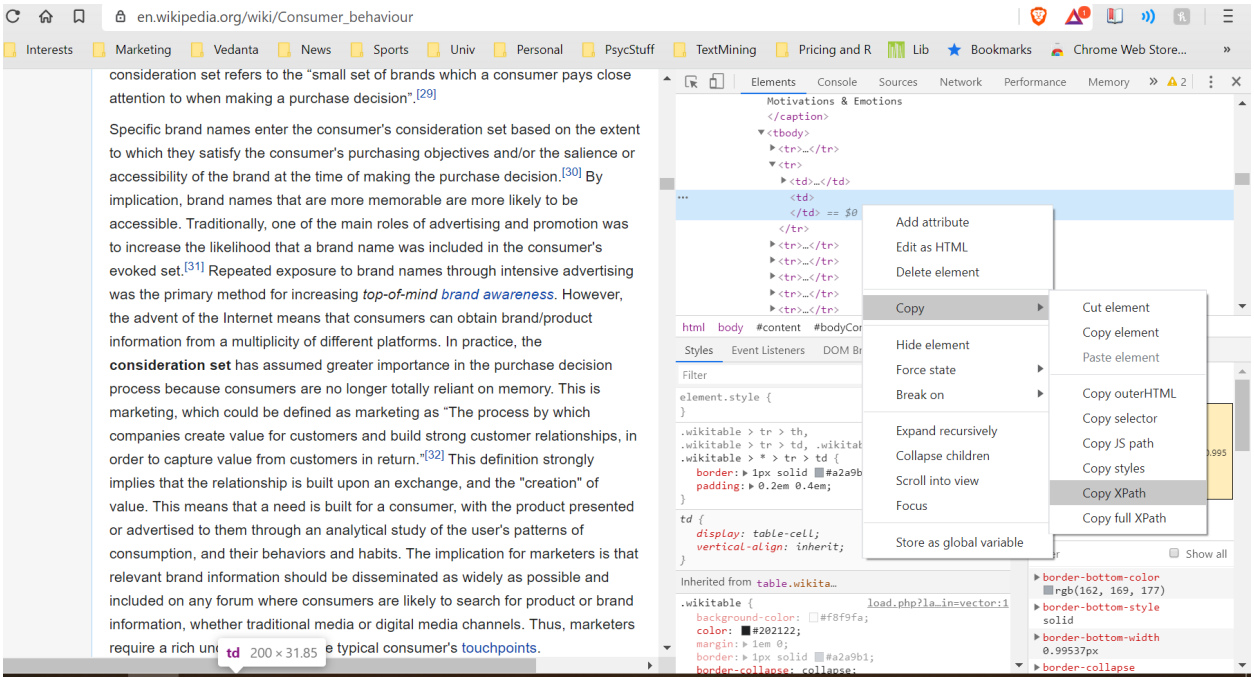
2) Using Selector Gadget

For both the methods, it helps if you have some background (or take some tutorials) on HTML and CSS (check out this link for a particularly lucid explanation of CSS: https://www.w3schools.com/css/css_intro.asp). Otherwise, it needs a little trial and error. For instance, say, I want to access this particular table from the webpage:

Rossiter and Percy's Purchase Motivations & Emotions

Motivation	Emotional Sequence
NEGATIVE	
Problem removal	Annoyance → Relief
Problem avoidance	Fear → Relaxation
Incomplete satisfaction	Disappointment → Optimism
Mixed approach avoidance	Conflict → Peace-of-mind
Normal depletion	Mild annoyance → Convenience
POSITIVE	
Sensory gratification	Dull (or neutral) → Sensory anticipation
Intellectual simulation	Bored (or neutral) → Excited
Social approval/ conformity	Apprehensive (or ashamed) → Flattered/ proud

We can use the Developer tools to select the portion of html code that contains the table. Right clicking on the table and selecting 'inspect', opens up the source code with a particular line of code highlighted blue. Next, you right click on that and select 'copy'->'copy xpath'



However, in this case if you select that, it is just for single line instead of the entire table. After a little trial and error (checking a few lines up and down), you insert the correct html code in the xpath below.

```
section_of_wikipedia<-html_node(wiki_cb, xpath='//*[@id="mw-content-text"]/div/table[3]')
percy_table<-html_table(section_of_wikipedia)
head(percy_table)
```

```
## # A tibble: 6 x 2
##   Motivation               ‘Emotional Sequence‘
##   <chr>                    <chr>
## 1 NEGATIVE                 ""
## 2 Problem removal          "Annoyance → Relief"
## 3 Problem avoidance        "Fear → Relaxation"
## 4 Incomplete satisfaction  "Disappointment → Optimism"
## 5 Mixed approach avoidance "Conflict → Peace-of-mind"
## 6 Normal depletion         "Mild annoyance → Convenience"
```

Let's understand the code: 'section_of_wikipedia<-html_node(wiki_cb,xpath='//*[@id="mw-content-text"]/div/table[3]'): This line of code uses the html_node() function from the rvest package in R to extract a specific HTML node from a webpage. The node being extracted is the third table in the content section of a Wikipedia page, and it is identified by its XPath, which is a way to navigate through the HTML structure of a webpage to find specific elements. The XPath used here selects the table with the ID mw-content-text and then selects the third table within that section. The resulting HTML node is stored in the section_of_wikipedia variable.

percy_table<-html_table(section_of_wikipedia): This line of code uses the html_table() function from the rvest package to convert the HTML node selected in the previous step into a data frame. The resulting data frame, which contains the contents of the table, is stored in the percy_table variable. Alternatively, you can get the same result with a simpler piece of code:

```
results <-
  wiki_cb  %>%
  html_table()
head(results[[3]])
```

```
## # A tibble: 6 x 2
##   Motivation               ‘Emotional Sequence‘
##   <chr>                    <chr>
## 1 NEGATIVE                 ""
## 2 Problem removal          "Annoyance → Relief"
## 3 Problem avoidance        "Fear → Relaxation"
## 4 Incomplete satisfaction  "Disappointment → Optimism"
## 5 Mixed approach avoidance "Conflict → Peace-of-mind"
## 6 Normal depletion         "Mild annoyance → Convenience"
```

If you are interested in drilling down the details of different elements, you can try a few things like these:

```r
body_nodes <- wiki_cb %>%
 html_nodes('body') %>%
 html_children() # for looking at nested elements

body_nodes %>%
 html_children()
```

```
## {xml_nodeset (3)}
## [1] <header class="vector-header mw-header"><div class="vector-header-start"> ...
## [2] <div class="mw-page-container-inner">\n\t\t<div class="vector-sitenotice- ...
## [3] <ul>\n<li>\n\t\t<button class="cdx-button cdx-button--icon-only vector-li ...
```

We know that wiki_cb is a variable containing the HTML content of a webpage, the code selects all the body nodes within the HTML using the html_nodes() function from rvest. The html_children() function is then used to retrieve all the children elements of each body node. The second block of code continues to retrieve the children elements of the body nodes that were previously selected in body_nodes. This will return a list of all the nested HTML elements within the body tag.

Finally, if you want to capture all the text material within the webpage (without worrying about headings, lists etc.), as is often the case in Bag-of-Words model there is an easier approach. All contents can be captured by using $< div >$ as R will extract all text contained in that division or section regardless of if it is contained in a paragraph or list. Note, there will be lot of 'junk' in the output that we will need to data wrangle.

```r
all_text<-wiki_cb %>%
html_nodes("div") %>%
html_text2()
```

Cleaning the Output If you use head(alltext2), you can see the output which needs a lot of cleaning. Let's try to clean some of the irrelevant things.

```r
# Clean up the plain text by removing extra spaces, newlines, etc.
clean_text <- gsub("[[:space:]]+", " ", all_text)
clean_text <- gsub("\n+", "\n", clean_text)
clean_text <- trimws(clean_text)

# Print the cleaned up text using the command below. Did
#cat(clean_text)
```

We cleaned up the text using a combination of regular expressions and string manipulation functions to remove extra spaces, newlines, and other formatting that may not be needed. Finally, it prints the cleaned up text using the cat() function. But even now, lot of cleaning is needed. For example, to remove all URLs from the text, you could use a regular expression like:

```
library(stringr)
clean_text <- gsub("(f|ht)tps?://\\S+", "", clean_text)
```

You may also want to clean lot of CSS output.

```
clean_text <- gsub("\\.mw-parser-output[^{]*\\{[^}]*\\}", "", clean_text)
```

In this example, the regular expression "\.mw-parser-output[^{]*\{[^}]*\}" matches any text that starts with .mw-parser-output and includes all the CSS properties until the closing brace. The gsub() function then replaces this text with an empty string, effectively removing it from the output.

Let's say we also want to clean up the references by putting them in separate lines.

```
clean_text <- gsub("\\^\\s*", "\n", clean_text) # Add newlines after each reference
```

The regular expression "\^\s*" matches each reference marker (^) and any whitespace characters that follow it, and replaces it with a newline character.

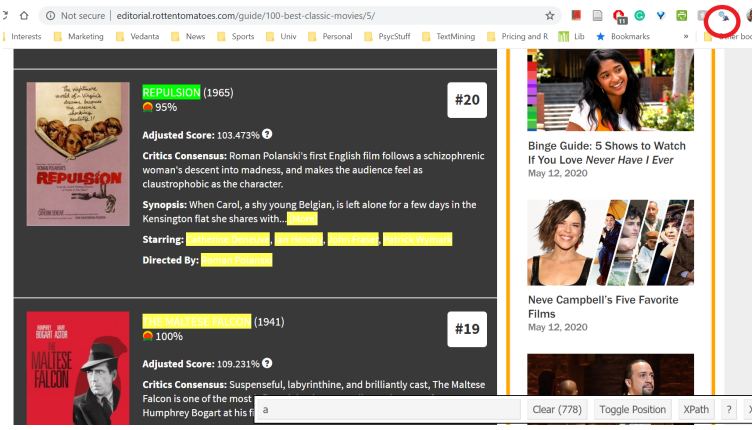
These were some examples of using regex for text cleaning. Regex is complicated and needs fair bit of practice. Here is a useful resource:

https://www.hackerearth.com/practice/machine-learning/advanced-techniques/regular-expressions-string-manipulation-r/tutorial/

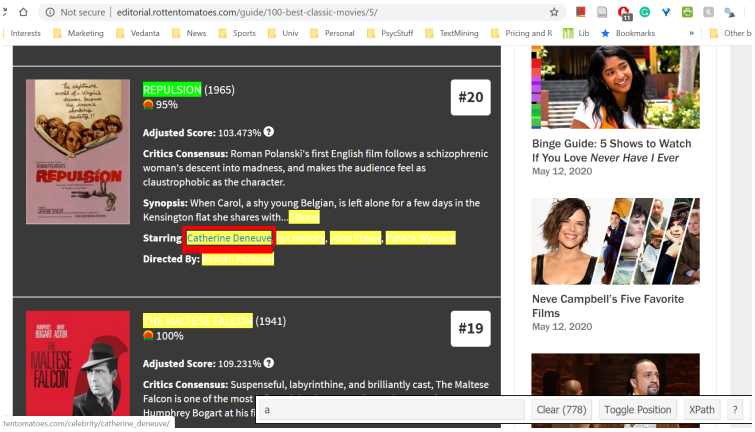
Using Selector Gadget for CSS (Cascading Style Sheets) The second option is to use Selector Gadget, it is fairly intuitive but old and not updated any more, as such it tends to give trouble with some modern websites. However, here is an overview of how to use selector gadget:

http://rvest.tidyverse.org/articles/selectorgadget.html

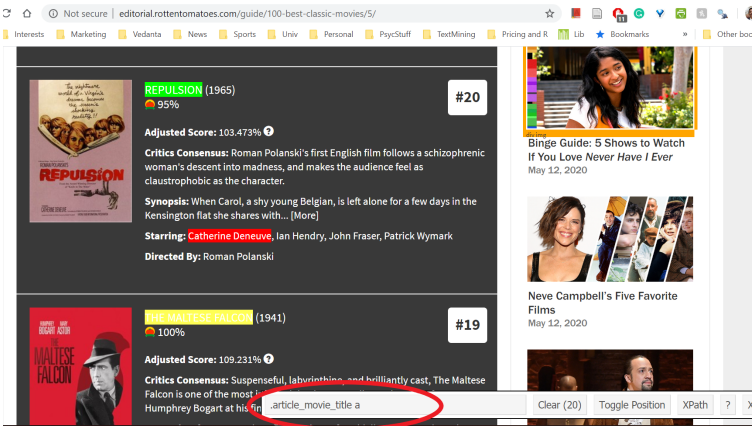
Basically, you just and point and click at the element you are interested in. After installing the Selector Gadget extension, you point at the selector gadget icon to activate it. Click on the element you are interested in. To illustrate how this works, let's scrape the top 20 best classic movies from RottenTomatoes website using Selector Gadget. Once you click on the selector gadget icon (encased in the red circle), selector gadget gets activated. Then let's say, we are interested in scraping the titles of the movies. Once you click on the movie title, it becomes highlighted in green (indicating correct selection). However, it also selects other possibilities in yellow (we are obviously not interested in adding the cast members and directors to the movie title variable)



To eliminate the yellow selection, if you hover the mouse over the yellow selection, it acquires a red outline (indicating an unwanted element). See below:



Once you click on that element, all the yellow selections vanish leaving only the titles of the movies, which was our primary interest. I have highlighted the CSS code with a red circle. Just copy that and paste it in the R code.



Creating R Dataset We will scrape the rottentomatoes website to create a dataset

```
top20movies <-
  bow("https://editorial.rottentomatoes.com/guide/100-best-classic-movies/") %>%
  scrape()
```

We will use Selector Gadget to scrape the movie titles

```
title_html <- html_nodes(top20movies,'.article_movie_title a')

#Converting the ranking data to text
titletext <- html_text(title_html)

#Let's have a look at the titles
length(titletext)
```

```
## [1] 100
```

```
head( titletext)
```

```
## [1] "The Philadelphia Story" "Seven Samurai"          "Meet Me in St. Louis"
## [4] "Singin' in the Rain"    "Laura"                  "M"
```

Now let's select the ranks

```
rank_html <- html_nodes(top20movies,'.countdown-index')

#Converting the ranking data to text
ranktext <- html_text(rank_html)

#Let's have a look at the rankings
head(ranktext)
```

```
## [1] "#1" "#2" "#3" "#4" "#5" "#6"
```

```
library(stringr)
ranktext<-str_replace_all(ranktext, "[^[:alnum:]]", "") # remove special characters
ranktext<-as.numeric(ranktext) # convert to numeric
```

Let's select the year and rankings

```
year_html <- html_nodes(top20movies,'.start-year')

#Converting the ranking data to text
yeartext <- html_text(year_html)
```

```
#Let's have a look at the rankings
head(yeartext)
```

```
## [1] "(1940)" "(1954)" "(1944)" "(1952)" "(1944)" "(1931)"
```

```
yeartext<-str_replace_all(yeartext, "[^[:alnum:]]", "") # remove special characters
yeartext<-as.numeric(yeartext) # convert to numeric
```

Let's also include Critic's consensus

```
critic_html <- html_nodes(top20movies,'.critics-consensus')

#Converting the critic's consensus data to text
critictext <- html_text(critic_html)

#Let's have a look at the critic's
head(critictext)
```

```
## [1] "Critics Consensus: Offering a wonderfully witty script, spotless direction from George Cukor, ar
## [2] "Critics Consensus: Arguably Akira Kurosawa's masterpiece, The Seven Samurai is an epic adventure
## [3] "Critics Consensus: A disarmingly sweet musical led by outstanding performances from Judy Garlan
## [4] "Critics Consensus: Clever, incisive, and funny, Singin' in the Rain is a masterpiece of the clas
## [5] "Critics Consensus: A psychologically complex portrait of obsession, Laura is also a deliciously
## [6] "Critics Consensus: A landmark psychological thriller with arresting images, deep thoughts on mo
```

And the synopsis

```
synopsis_html <- html_nodes(top20movies,'.synopsis')

#Converting the ranking data to text
synopsistext <- html_text(synopsis_html)

#Let's have a look at the rankings
head(synopsistext)
```

```
## [1] "Synopsis: This classic romantic comedy focuses on Tracy Lord (Katharine Hepburn), a Philadelphia
## [2] "Synopsis: A samurai answers a village's request for protection after he falls on hard times. The
## [3] "Synopsis: \"Meet Me in St. Louis\" is a classic MGM romantic musical comedy that focuses on four
## [4] "Synopsis: A spoof of the turmoil that afflicted the movie industry in the late 1920s when movies
## [5] "Synopsis: In one of the most celebrated 1940s film noirs, Manhattan detective Mark McPherson (Da
## [6] "Synopsis: In this classic German thriller, Hans Beckert (Peter Lorre), a serial killer who preys
```

We will also add the cast

```
cast_html <- html_nodes(top20movies,'.cast')

#Converting the cast data to text
casttext <- html_text(cast_html)

#Let's have a look at the casting
head(casttext)
```

```
## [1] "\n                Starring: Cary Grant, Katharine Hepburn, James Stewart, Ruth Hussey"
## [2] "\n                Starring: Toshiro Mifune, Takashi Shimura, Yoshio Inaba, Seiji Miyaguchi"
## [3] "\n                Starring: Judy Garland, Margaret O'Brien, Leon Ames, Lucille Bremer"
## [4] "\n                Starring: Gene Kelly, Debbie Reynolds, Donald O'Connor, Jean Hagen"
## [5] "\n                Starring: Gene Tierney, Dana Andrews, Clifton Webb, Vincent Price"
## [6] "\n                Starring: Peter Lorre, Ellen Widmann, Inge Landgut, Otto Wernicke"
```

Finally combine these into a dataframe

```
library(tidyverse)
movies_df<-tibble(Rank = ranktext, Title = titletext, Year= yeartext, Critic= criticktext,Synopsis= synop
head(movies_df)
```

```
## # A tibble: 6 x 6
##    Rank Title                  Year Critic                       Synopsis Cast
##   <dbl> <chr>                 <dbl> <chr>                        <chr>    <chr>
## 1     1 The Philadelphia Story  1940 Critics Consensus: Offering~ "Synops~ "\n ~
## 2     2 Seven Samurai           1954 Critics Consensus: Arguably~ "Synops~ "\n ~
## 3     3 Meet Me in St. Louis    1944 Critics Consensus: A disarm~ "Synops~ "\n ~
## 4     4 Singin' in the Rain     1952 Critics Consensus: Clever, ~ "Synops~ "\n ~
## 5     5 Laura                   1944 Critics Consensus: A psycho~ "Synops~ "\n ~
## 6     6 M                       1931 Critics Consensus: A landma~ "Synops~ "\n ~
```

When Should I Web-Scrape?

Looking into the previous sections, hopefully, you can understand that web-scraping can be pretty troublesome. After checking legality of web-scraping and ease of doing it, you can take a judgment call. However, another option which is widely used can be the Application Programming Interfaces (APIs), our next topic.

These notes have borrowed liberally from these excellent resources:

i. http://rvest.tidyverse.org/
ii. https://www.dataquest.io/blog/web-scraping-in-r-rvest/
iii. https://www.freecodecamp.org/news/an-introduction-to-web-scraping-using-r-40284110c848/

iv. https://afit-r.github.io/scraping
v. https://towardsdatascience.com/tidy-web-scraping-in-r-tutorial-and-resources-ac9f72b4fe47
vi. https://www.dataquest.io/blog/r-api-tutorial/