Notes Mining for Tweets
DATA 110
Saidi

Name _____
Date _____

- Ethics of Mining and Web Scraping
- Webscraping tutorial
- Homework Week 9

Ethics of Mining and Web Scraping

We will use two tutorials to learn a bit about mining for data and scraping the web for data. In both cases, the overall goal is to create our own dataset, based on information we can glean directly from sources. The downside is that, if used improperly, we could violate ethics by revealing private information.

Example of Questionable Use of Data Scraping

(from https://www.vox.com/2016/5/12/11666116/70000-okcupid-users-data-release)

In May 2016, the online OpenPsych Forum published a paper by Kirkegaard and Bjerrekær (2016) titled "The OkCupid data set: A very large public data set of dating site users." The resulting data set contained nearly 70,000 users with 2,620 variables scraped from OkCupid dating website. Variables included usernames, gender, dating preferences, religiosity, political preferences, etc. The purpose of the data dump was to provide an open public data set to fellow researchers, and it could possibly be used to answer questions such as this one suggested in the abstract of the paper: whether the zodiac sign of each user was associated with any of the other variables (spoiler alert: it wasn't).

The data scraping did not involve any illicit technology such as breaking passwords.

"But the data set reveals deeply personal information about many of the users. OkCupid uses a series of personal questions — on topics such as sexual habits, politics, fidelity, feelings on homosexuality, etc. — to help match people on the site. The data dump did not reveal anyone's real name. But it's entirely possible to use clues from a user's location, demographics, and OkCupid user name to determine their identity." (Vox)

Nonetheless, the author received many comments on the OpenPsych Forum challenging the work as an ethical breach and accusing him of doxing people by releasing personal data.

AI-GENERATED ART CANNOT BE COPYRIGHTED IN THE US
FROM DSCN (DATA SCIENCE COMMUNITY NEWSLETTER) #275

US District Judge **Beryl Howell** in Washington DC has **ruled** that fully AI-generated art cannot be copyrighted in the US.

This may somewhat miss the heart of the legal matter around generative AI, for three reasons. One there is a set of questions about how the use of copyrighted training data is allowable without permission from (or compensation to) the copyright holder. Most of the big foundational models were trained in the absence of clear legal guidance and the hope that, at least in the US, fair use doctrine would allow training to go forward without much more than a web scraper and a yoink sound effect. Second, within the realm of copyright, many generative AI tools offer advanced editing capabilities, where the final output is a composite partially created by humans. Those composites would likely still receive copyright protections, though apparently writing a prompt is not sufficient human input for the resulting image to be considered worthy of copyright protection. Third, there are unsettled legal questions about the protections afforded to the underlying technology (e.g. the models) and the relationship between the technology, the training data, and the model outputs. This case confirms that fully synthetic outputs cannot be copyrighted, which could leave a legal argument available for the fair use argument to stand when it comes to the training data.

Webscraping Basics by Paul Bauer

5.1 Web scraping: Basics

- Increasing amount of data is available on websites:
  - Speeches, sentences, biographical information…
  - Social media data, newspaper articles, press releases…
  - Geographic information, conflict data…
- Often data comes in unstructured form (e.g. no dataframe with rows/columns)
- Web scraping = process of extracting this information automatically and transforming it into a structured dataset

5.1.1 Scraping data from websites: Why?

- Copy & paste is time-consuming, boring, prone to errors, and impractical for large datasets
- In contrast, automated web scraping:
  1. Scales well for large datasets
  2. Is reproducible
  3. Involved adaptable techniques
  4. Facilitates detecting and fixing errors
- When to scrape?
  1. Trade-off: More time now to built, less time later
  2. Computer time is cheap; human time is expensive
- Example for time saving: legislatorR package vs. manual collection

5.1.2 Scraping the web: two approaches

Two different approaches:

1. Screen scraping: extract data from source code of website, with html parser and/or regular expressions
   - `rvest` package in R
2. Web APIs (application programming interfaces): a set of structured http requests that return JSON or XML data
   - `httr` package to construct API requests
   - API packages: `weatherData`, `WDI`, `Rfacebook` (CRAN Task View on Web Technologies provides an overview)

5.1.3 The rules of the game

1. Respect the hosting site's wishes:
   - Check if an API exists or if data are available for download
   - Keep in mind where data comes from and give credit (and respect copyright if you want to republish the data!)
   - Some websites disallow scrapers on robots.txt file
2. Limit your bandwidth use:
   - Wait one or two seconds after each hit
   - Scrape only what you need, and just once (e.g. store the html file in disk, and then parse it)
3. When using APIs, read documentation
   - Is there a batch download option?
   - Are there any rate limits?
   - Can you share the data?

Excerpts from https://statsandr.com/blog/web-scraping-in-r/

Introduction

When you type any site address in your browser, your browser will download and render the page for you, but for rendering the page it needs some instructions.

There are 3 types of instructions:

- **HTML**: describes a web page's infrastructure;
- **CSS**: defines the appearance of a site;
- **JavaScript**: decides the behavior of the page.

Web scraping is the art of extracting information from the HTML, CSS and Javascript lines of code. The term usually refers to an automated process, which is less error-prone and faster than gathering data by hand.

It is important to note that web scraping can raise **ethical concerns**, as it involves accessing and using data from websites without the explicit permission of the website owner. It is a good practice to respect the terms of use for a website, and to seek written permission before scraping large amounts of data.

This article aims to cover the basics of how to do web scraping in R. We will conclude by creating a database on Formula 1 drivers from Wikipedia.

Note that this article doesn't want to be exhaustive on topic. To learn more, see this section.

HTML and CSS

Before starting it is important to have a basic knowledge of HTML and CSS. This section aims to briefly explain how HTML and CSS work, to learn more we leave you some resources at the bottom of this article.

Feel free to skip this section if you already are knowledgeable in this topic.

Starting from **HTML**, an HTML file looks like the following piece of code.

```
<!DOCTYPE html>
<html lang="en">
<body>

<h1 href="https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss"> Carl Friedrich
Gauss</h1>
<h2> Biography </h2>
<p> Johann Carl Friedrich Gauss was born on 30 April 1777 in Brunswick. </p>
<h2> Profession </h2>
<p> Gauss is considered as one of the greatest mathematician, statistician and
physicist of all time. </p>
```

```
</body>
</html>
```

Those instructions produce the following:

Carl Friedrich Gauss

Biography

Johann Carl Friedrich Gauss was born on 30 April 1777 in Brunswick.

Profession

Gauss is considered as one of the greatest mathematician, statistician and physicist of all time.

As you read above, **HTML** is used to describe the infrastructure of a web page, for example we may want to define the headings, the paragraphs, etc.

This infrastructure is represented by what are called *tags* (for example `<h1>...<\h1>` or `<p>...<\p>` are tags). Tags are the core of an HTML document as they represent the nature of what is inside the tag (for example `h1` stands for heading 1). It is important to observe that there are two types of tags:

- starting tags (e.g. `<h1>`)
- ending tags (e.g. `<\h1>`)

This is what allows to nest different tags.

Tags can also have attributes, for example in `<h1 href="https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss"Carl Friedrich Gauss</h1>`, `href` is an attribute of the tag `h1` that specifies an URL.

As the output of the above HTML code is not super elegant, **CSS** is used to style the final website. For example CSS is used to define the font, the color, the size, the spacing and many more features of a website.

What is important for this article are *CSS selectors*, which are patterns used to select elements. The most important is the `.class` selector, which selects all elements with the same class. For example the `.xyz` selector selects all elements with `class="xyz"`.

Web scraping vs. APIs

Going back to web scraping, you may know that APIs are another way to access data from websites and online services.

In fact, an API is a set of rules and protocols that allows two different software systems to communicate with each other. When a website or online service provides an API, it means that they have made it possible for developers to access their data in a structured and controlled way.

Why does web scraping exist if APIs are so powerful and do exactly the same work?

The main difference between web scraping and using APIs is that APIs are typically provided by the website or service to allow access to their data, while web scraping involves accessing data without the explicit permission of the website owner.

This means that using APIs is generally considered more ethical than web scraping, as it is done with the explicit permission of the website or service.

However, there are also some limitations to using APIs:

- many APIs have rate limits, which means that they will only allow a certain number of requests to be made within a certain time period, i.e. you may not access large amounts of data;
- not all websites or online services provide APIs, which means the only way to access their data is via web scraping.

How to Inspect a Website for HTML Content

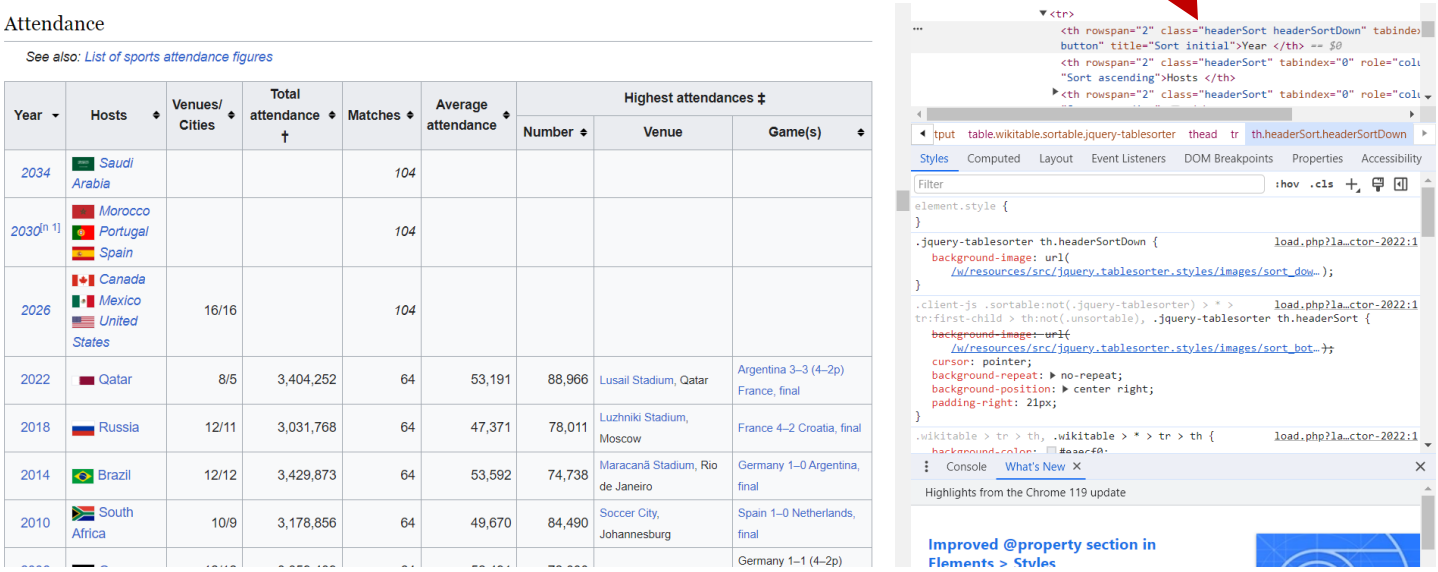
1. Right-click on a website element and click “inspect”

OR

2. Control + Shift + i (for PCs)

OR

3. Command + option + i (for Macs)



Attendance

See also: List of sports attendance figures

Year	Hosts	Venues/ Cities	Total attendance †	Matches	Average attendance	Highest attendances ‡		
						Number	Venue	Game(s)
2034	Saudi Arabia			104				
2030[n 1]	Morocco Portugal Spain			104				
2026	Canada Mexico United States	16/16		104				
2022	Qatar	8/5	3,404,252	64	53,191	88,966	Lusail Stadium, Qatar	Argentina 3–3 (4–2p) France, final
2018	Russia	12/11	3,031,768	64	47,371	78,011	Luzhniki Stadium, Moscow	France 4–2 Croatia, final
2014	Brazil	12/12	3,429,873	64	53,592	74,738	Maracanã Stadium, Rio de Janeiro	Germany 1–0 Argentina, final
2010	South Africa	10/9	3,178,856	64	49,670	84,490	Soccer City, Johannesburg	Spain 1–0 Netherlands, final
								Germany 1–1 (4–2p)

Web scraping in R

There are several packages for web scraping in R, every package has its strengths and limitations. We will cover only the rvest package since it is the most used.

To get started with web scraping in R you will first need R and RStudio installed (if needed, see here). Once you have R and RStudio installed, you need to install the rvest package:

```
install.packages("rvest")
```

rvest

Inspired by beautiful soup and RoboBrowser (two Python libraries for web scraping), rvest has a similar syntax, which makes it the most eligible package for those who come from Python.

rvest provides functions to access a web page and specific elements using CSS selectors and XPath. The library is a part of the Tidyverse collection of packages, i.e. it shares some coding conventions (e.g. the pipes) with other libraries as tybble and ggplot2.

Before the real scraping it is necessary to load the rvest package:

```
library(rvest)
```

Now that everything is settled down, we can start the web scraping operation, which is usually made in 3 steps:

1. **HTTP GET request**
2. **Parsing HTML content**
3. **Getting HTML element attributes**

These steps are detailed in the following sections.

HTTP GET request

The HTTP GET method is a method used to send a server a question to get certain data and information. It is important to notice that this method does not change the state of the server.

To send a GET request we need the link (as a character) to the page we want to scrape:

```
url <- "https://www.nytimes.com/"
```

Sending the request to the page is simple, rvest provides the read_html function, which returns an object of html_document type:

```
NYT_page <- read_html(url)


NYT_page
## {html_document}
## <html lang="en" class=" nytapp-vi-homepage"
xmlns:og="http://opengraphprotocol.org/schema/">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8
...
## [2] <body>\n    <div id="app">\n<a class="css-kgn7zc" href="#site-content">Sk
...
```

Parsing HTML content

As we saw in the last chunk of code, NYT_page contains the raw HTML code, which is not so easily readable.

In order to make it readable from R it has to be parsed, which means generating a Document Object Model (DOM) from the raw HTML. DOM is what connects scripts and web pages by representing the structure of a document in memory. If you retrieve the HTTP request using Node.js, you can give the raw HTML response to R for parsing and further analysis.

rvest provides 2 ways to select HTML elements:

1. **XPath**
2. **CSS selectors**

We will focus on the CSS selector:

```
NYT_page |>
  html_elements(css = "")
```

Web Scraping IMDB 2019 Movies

AUTHOR

Rachel Saidi

PUBLISHED

Install necessary packages for this project

```
#install.packages('rvest')
#Loading the rvest package
library(rvest)
library(tidyverse)
library(plotly)
```

Scrape the IMDB website to create a data frame of information from 2019 top 100 movies

Use the following URL from IMBD movies of 2019

https://www.imdb.com/search/title/?title_type=feature&release_date=2019-01-01,2019-12-31&count=100

```
#Specifying the url for desired website to be scraped
url <- 'https://www.imdb.com/search/title/?title_type=feature&release_date=2019-01-01,2019-12-
31&count=100'

#Reading the HTML code from the website
webpage <- read_html(url)
```

```
# save_url(webpage, filename="webpage.html")
```

Load various elements and clean data using gsub.

Scrape for Movie Rank Information

Use the command, length, to ensure that each list contains 100 elements or NAs for missing data to sum to 100 elements.

```
#Use CSS selectors to scrape the rankings section
rank_title_html <- html_elements(webpage, css='.ipc-title__text')
head(rank_title_html)
{xml_nodeset (6)}
[1] <h1 class="ipc-title__text">Advanced search</h1>
[2] <h3 class="ipc-title__text">1. The Gentlemen</h3>
[3] <h3 class="ipc-title__text">2. Code 8</h3>
[4] <h3 class="ipc-title__text">3. Once Upon a Time... in Hollywood</h3>
[5] <h3 class="ipc-title__text">4. Midsommar</h3>
[6] <h3 class="ipc-title__text">5. Parasite</h3>
#Convert the ranking data to text
rank_title_data <- html_text(rank_title_html)

#Remove the first and last rows - they are not movie titles
rank_title_data <- rank_title_data[-c(1, 102)]

#Let's have a look at the rankings
head(rank_title_data)
[1] "1. The Gentlemen"                    "2. Code 8"
[3] "3. Once Upon a Time... in Hollywood" "4. Midsommar"
[5] "5. Parasite"                         "6. Little Women"
# notice that the format is "rank. title"

length(rank_title_data)
[1] 100
#should be 100
```

Scrape for Rank Information from the rank_title information

```
# remove the title and extract just the number
rank_data <- parse_number(rank_title_data)

summary(rank_data)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00   25.75   50.50   50.50   75.25  100.00
```

Scrape for Title Information

```
# Use the rank_title_data and extract just the characters from the title
title_data <- str_sub(rank_title_data, start = 4L, end = -1L)

head(title_data)      #check first 6 titles
[1] "The Gentlemen"                    "Code 8"
[3] "Once Upon a Time... in Hollywood" "Midsommar"
[5] "Parasite"                         "Little Women"
length(title_data)     # check number of titles - should be 100
[1] 100
```

Scrape for Title Information

```
#Use CSS selectors to scrape the description section
description_data_html <- html_elements(webpage, css='.ipc-html-content-inner-div')

#Convert the description data to text
description_data <- html_text(description_data_html)



#Let's have a look at the description data
head(description_data)
```

```
[1] "An American expat tries to sell off his highly profitable marijuana empire in London, triggering plots, schemes, bribery and blackmail in an attempt to steal his domain out from under him."
[2] "A super-powered construction worker falls in with a group of criminals in order to raise the funds to help his ill mother."
[3] "A faded television actor and his stunt double strive to achieve fame and success in the final years of Hollywood's Golden Age in 1969 Los Angeles."
[4] "A couple travels to Northern Europe to visit a rural hometown's fabled Swedish mid-summer festival. What begins as an idyllic retreat quickly devolves into an increasingly violent and bizarre competition at the hands of a pagan cult."
[5] "Greed and class discrimination threaten the newly formed symbiotic relationship between the wealthy Park family and the destitute Kim clan."
[6] "Jo March reflects back and forth on her life, telling the beloved story of the March sisters - four young women, each determined to live life on her own terms."
```

```
#What is the length of this vector for description data
length(description_data)
```

```
[1] 100
```

```
#It should be 100
```

Scrape for details information

```
#Use CSS selectors to scrape the Movie runtime
details_data_html <- html_elements(webpage, css = 'span.sc-b0691f29-8.ilsLEX.dli-title-metadata-item')

#Convert the description data to text
details_data <- html_text(details_data_html)
head(details_data)
```

```
[1] "2019"      "1h 53m"    "R"         "2019"      "1h 38m"    "Not Rated"
```

Notice details include year, runtime, and rating

Filter just for the runtime

```
# Filter out the movie runtimes in the form "Xh XXm" from details_data
runtime_text <- details_data[grep("\\d+h", details_data)]
head(runtime_text)
```

```
[1] "1h 53m" "1h 38m" "2h 41m" "2h 28m" "2h 12m" "2h 15m"
```

Convert runtime_text from hours and minutes to minutes

```r
# Convert runtime_text from hours and minutes to minutes
converted_runtimes <- sapply(strsplit(runtime_text, "h |m"), function(x) as.numeric(x[1]) * 60 +
as.numeric(x[2]))

# Display the converted movie runtimes
head(converted_runtimes)
```
```
[1] 113  98 161 148 132 135
```
```r
length(converted_runtimes)
```
```
[1] 100
```

Check to make sure movies match with runtimes with a temporary data frame

```r
# Display the titles of movies with missing runtimes and their corresponding runtimes
df1 <- data.frame(Title = title_data, Runtime = converted_runtimes)
df1
```
```
                                        Title Runtime
1                                The Gentlemen     113
2                                       Code 8      98
3                 Once Upon a Time... in Hollywood     161
4                                    Midsommar     148
5                                     Parasite     132
6                                  Little Women     135
7                                        Joker     122
8                             Avengers: Endgame     181
9                                Ford v Ferrari     152
10                                    Yesterday     116
```

Scrape for Voting Information

```r
# Use CSS selectors to scrape the number of votes
votes_labels <- html_nodes(webpage, ".sc-d80c3c78-0.jPoaKS span.sc-d80c3c78-7.gHmMgF")
votes_numbers <- html_nodes(webpage, ".sc-d80c3c78-0.jPoaKS")

votes_text <- html_text(votes_numbers)
votes_values <- gsub("Votes", "", votes_text)
votes_values <- gsub(",", "", votes_values)
votes_values <- as.numeric(votes_values)

# Display the extracted number of votes
head(votes_values)
```
```
[1] 390090  50397 840222 398046 947798 242859
```
```r
length(votes_values)
```
```
[1] 100
```

Combine all the lists to form a data frame

```r
# Display the movies with missing or invalid runtimes
df_movies <- data.frame(rank = rank_data, title = title_data, description = description_data,
runtime = converted_runtimes, votes = votes_values)
head(df_movies)
```
```
  rank                           title
1    1                 The Gentlemen
2    2                        Code 8
```

```
3    3 Once Upon a Time... in Hollywood
4    4                        Midsommar
5    5                         Parasite
6    6                      Little Women

description
1                                                 An American expat tries to sell off his highly
profitable marijuana empire in London, triggering plots, schemes, bribery and blackmail in an
attempt to steal his domain out from under him.
2
A super-powered construction worker falls in with a group of criminals in order to raise the funds
to help his ill mother.
3                                                                                          A faded
television actor and his stunt double strive to achieve fame and success in the final years of
Hollywood's Golden Age in 1969 Los Angeles.
4 A couple travels to Northern Europe to visit a rural hometown's fabled Swedish mid-summer
festival. What begins as an idyllic retreat quickly devolves into an increasingly violent and
bizarre competition at the hands of a pagan cult.
5
Greed and class discrimination threaten the newly formed symbiotic relationship between the
wealthy Park family and the destitute Kim clan.
6                                                                                Jo March reflects back
and forth on her life, telling the beloved story of the March sisters - four young women, each
determined to live life on her own terms.
  runtime  votes
1     113 390090
2      98  50397
3     161 840222
4     148 398046
5     132 947798
6     135 242859
```

Problem 1: Based on the scraped 2019 IMDB movie data frame, create a scatterplot that shows runtime on the x-axis, number of votes on the y-axis. Be sure to provide a title, axis labels, and caption for the data source.

```
## ggplot
```

Problem 2: Use the filter function to answer the following question.

Which movie had the longest runtime AND highest votes from the top 10 highest ranked movies? I must see code that shows your filtering. Be sure to answer what the name of the movie is, what its runtime is and what its number of votes are.

```
## top10
```

Problem 3: In the runtime of 130-160 mins, which movie from the lowest ranked 10 movies (out of 100) had highest votes?

Again, you must use the filter function to get the exact movie which answers this question. Be sure to state the rank and number of votes for this movie.

```
## bottom10
```

Problem 4: Create a categorical variable

1. Create an additional categorical variable, "rank_groups" that groups movies into ranks 1-20, 21-40, 41-60, 61-80, and 81-100.
2. Color your scatterplot from Problem #1 by this new variable. Be sure to use a NON-DEFAULT color palette.
3. Make the size of the point based on the runtime.
4. Be sure your plot includes a legend for the rank_groups.
5. Add interactivity to get the movie title on mouseover for that point.

```
## df_movies2
## ggplot
```

Week 9 Homework Assignment

(Worth up to 15 points)

a. On your own quarto document, follow the Webscraping for 2019 IMDB MoviesTutorial.

b. Try all code, and focus on trying to learn some HTML and CSS.

c. Answer (fully) Problems 1-4 at the end of the tutorial. Be sure I can find your answers to the questions.

d. ONLY render to Word or pdf this week – not HTML. You will get a zero for this assignment if you publish your html content to rpubs.

Submit the completed tutorial by **11:59 pm on Sunday, March 31st. We will go over answers during the next week's class.**