WEB-SCRAPING FOR LINKEDIN

Many websites contain a large amount of invaluable data, like pictures, stock prices, sports statistics, or just plain text. In order to gather all those data, a user could copy and paste the object of interest, but what would happen if it was not just one single piece of text, but multiple entries that are contained in many different sections of a webpage? It would be very tedious and time consuming to gather all the relevant pieces of information by hand. What a user could do instead, is use a web scraping algorithm to achieve his goals.

Web scraping is a technique, which automatically gathers data from the internet, by using the hierarchical structure of a webpage and then exports the results into a format that is more useful to the user. However, since the webpages come in all shapes and sizes, so do the algorithms that perform web scraping. In general, a webpage is comprised of different languages, such as HTML, CSS, JavaScript, PHP, etc. that are interlinked within its source code and have it respond dynamically to whatever a user does (like click, or point, or hover over an object with a mouse, or perhaps type a string of characters using a keyboard, etc.). Therefore, the goal of web scraping, is to utilize this unique structure and get the data that a user desires. There are of course many web scrapers available online, however for the needs of this thesis, what is needed is a custom-made web scraper that will produce the desirable results.

This whole procedure could be done in any of the popular languages, but as R was used since the beginning, so it was used for this part as well. It really helps that it had some of the more user-friendly libraries, which will be discussed below, along with other tools that were used to make this project a reality.

Docker:

This is a tool that is used for application deployment either through the cloud, or locally on a personal computer. What the web scraping program needs is a host (or a server) to run the library called RSelenium on a local machine, by deploying a Docker image of the web browser that is going to be used and then connecting that image with our code and its dependencies. If RSelenium was not chosen as a library, Docker would not even be mentioned, but the alternative was a very slow method.

The rvest library:

This library is commonly used in unison with others (e.g., *polite* library) to perform web scraping. It is used to locate whatever part of the source code a user might need. It can access HTML code hierarchically and use CSS selectors that speed up the whole parsing process.

The pipe operator (`%>%`) is used to access the inner workings of each HTML element. Therefore if anyone wanted to access and read the first paragraph within the `<body>` part of the HTML code, then they would need to write the following code:

```
html %>% html_element("body") %>% html_element("p") %>% html_text()
```

Therefore from the whole html output, this piece of code first accesses the <body> part of the HTML code and then directs its attention to the first element of a paragraph (which is symbolized as <p> in HTML).

Apart from that, as it was mentioned above, this library also uses CSS selectors. Since most sites use inline CSS code along with HTML code, it becomes easier to skip a few levels in order to reach the piece of information that a user needs, as CSS code is more unique and thus can be instantly recognized by a web scraper. Therefore, if we have many paragraphs within a source code (and thus lots of <p>), if these paragraphs have inline CSS code, for instance like:

```
<p id="fourth">
```

Then it is easy to jump to that piece of code by using CSS selectors, as no other paragraph will bear the same id. The four most important ones are the following:

- p to select all paragraphs inside HTML. For example: `html_elements("p")`.

- $.keyword$ to select all $class$ elements that start with a keyword. Thus if we have a paragraph with an inline CSS code, like: , then we could select it by typing: `html_elements(".sign")`.

- $p.special$ to select all <p> with $class$ elements that start with special. Same thing as above, but for paragraphs only.

- $\#keyword$ to select all id elements that start with a keyword. So if we have the following piece of source code <p id="fourth"> surrounding the information we need, then we can select it in the following manner: `html_element("#fourth")`.

The RSelenium library:

This library is using the Selenium Webdriver Application Programming Interface (API), which can help a user automate and perform browser-based actions (like clicking, scrolling, selecting and much more) and is mainly used for testing applications. The only downside in this instance, is that the program must link to the Selenium Server to perform the actions required and that is the reason why Docker is used. But other than that, this Java-based program can perform any action a browser does and thus allowing R users to automate any procedure that they might need to do, which is extremely helpful especially for web scraping applications.

The XML library:

This is another versatile library that can parse and create both XML and HTML source code. It is similar to rvest in its functionality however it goes a bit slower when selecting specific topics from multiple webpages. It uses $htmlParse()$, which generates an R structure representing the XML/ HTML tree, as well as $xpathSApply()$, which searches within a text for the information needed according to the path provided by the user.

Specifically for LinkedIn:

In the case of LinkedIn, one would see that it is a dynamic webpage, meaning that when a user scrolls or clicks on an object, it responds to the action. In the job section, one could clearly think that it is all just one page, but if one explores the source code, then they will see a two-fold hierarchy that uses anchor tags and internal links to each of the job advertisements. Therefore, the main page stores all the links to the jobs, within a certain country, and for the particular choice of words that were used within the search bar and only after we click on the particular job, can we see its description and the particulars of the job on the right-hand side of the page.

However, that is not all. As it was mentioned above, LinkedIn is a dynamic website and that also applies to whenever a user scrolls down the webpage itself. There are parts of source code that remain invisible as long as nothing happens, but as the user scrolls down the webpage, parts of code get loaded and appear within the source code. This too must be done automatically through R in order to get adequate results, as many jobs remain invisible, as long as the source code remains in its original form when a user only opens a webpage without scrolling down and loading most of the available links. In this part of code, this can go up to 150 jobs in total, per country, per term searched, as after this, there is a button prompt to load even more job positions, however it was deemed adequate to stop there so as to reduce the bias in the data.

How it works:

Before performing any data scraping, we first install Docker locally on our computer, because it is required for the library RSelenium to work. After we map the container port, to the host port, we will need to pull and load the image of the browser that we will be using (for example Firefox) through the command prompt of our operating system. Once that is done, we can start the web scraping procedure.

- First of all, we construct our url, based on the country and the term to be searched. For instance:

  https://www.linkedin.com/jobs/search?keywords=%22Data Scientist%22&location=England&position=1&pageNum=0

  This refers to the position of a “Data Scientist” in England and it directs on the first page of the LinkedIn search engine.

- After that is done, we request a connection with the HTTP server by using the link that was already constructed.

- When we get directed to the first page of LinkedIn, we scroll down 5 times, in order to load the maximum number of job advertisements that we can for the country and the search term that we selected above.

- Afterwards, we open the source code and parse the HTML code in order to get it all in a script back on our computer, by using the XML library.

- From that script, we select only the internal href links that contain the job advertisements.

- After constructing this list of internal, anchor links within the site, we access each and every one of them, in order to parse the job descriptions and characteristics of interest and store them in a huge data frame, which is done by using the library rvest. This library allows us to access the HTML, CSS and JavaScript code hierarchically and get to the script that interests us (in this case the job description, as well as Industry and Seniority Level variables).

- Within that loop a pattern-searching vector is applied to the description of the job which tries to detect words of interest (such as R, Python, SAS, etc.), considering lowercase, as well as uppercase terms that could be included.

- After that procedure is done, the algorithm terminates the server connection to the site and returns the data frame that the user asked.

In conclusion, this algorithm is time-efficient, meaning that it can do a job which manually took 10 days to complete, within a mere 40 minutes, however it is not as smart as a human being, which means that it will not discard many of the repeating jobs that appear in many countries, or ambiguous jobs from recruiting agencies, or search the internet and infer the true seniority level, or the industry to which a job belongs to. But other than that, it does the job adequately.

Two HTTP-based errors:

Since this application is web-based, it means that it can get faulty at times, and especially when a connection cannot be established due to errors in the HyperText Transfer Protocol (HTTP). Two of the most common errors are the following:

- HTTP Error 500: Internal Server Error. It means that the request of accessing a page has been unsuccessful and in order to correct it, it is required to reload the image of the browser used in Docker.

- HTTP Error 429: Too Many Requests. This is a way for each site to protect itself from a Distributed Denial-of-Service (DDoS) attack, which, in essence exhausts a website's resources by constantly bombarding it with connection requests. Usually there is a time out, which can last from a couple of minutes to a whole day. Thus, in order to bypass it, one solution would be to wait the period of time, but as time can be pressing, one can change their IP address, by either restarting their router, thus allowing the DHCP to reallocate a new public IP, or by running the following lines of code in the Command Prompt (*cmd*) in order to refresh and switch an old IP address to a new one:

```
netsh winsock reset
netsh int ip reset
ipconfig /release
ipconfig /renew
ipconfig /flushdns
```