Google Search Scraping Automation

1. Objective:

Development of a web scraping tool in Python using libraries of choice. The tool would allow searching on Google of user specified topic, and then scrape relevant data from multiple sources. Final form required to be in a neat format in a txt file, automatically generated.

2. Overview:

The final code is a web scraping tool that allows users to search for a topic on Google and retrieve a specified number of search results. The code employs various libraries to simulate different user agents and avoid detection while making requests to Google. It then extracts the URLs from the search results and proceeds to fetch the first paragraphs of articles from those URLs. The script filters out non-article pages and ensures that each paragraph is unique before saving them into a text file. Additionally, the code creates a .docx file with the extracted paragraphs, organized under the topic heading. Overall, this web scraping tool provides users with a convenient way to gather information on a given topic from Google search results and store the content for further analysis or reference.

3. Methodology

3.1 Web Scraping:

I leveraged the requests library to make HTTP requests to the Google search engine and retrieve search results. Then, I used BeautifulSoup to parse the HTML content of the search results page and extract relevant information, such as the URLs of search results and the content of web pages.

3.2 User-Agent Rotation:

To avoid being blocked by Google, I simulated different user agents for each request using a list of predefined user agent strings. This strategy made the requests appear as if they were coming from different web browsers and devices.

3.3 URL Parsing:

I utilized the urlparse function from the urllib.parse module to extract the domain name from each URL. This helped me identify unique articles and avoid duplication in the extracted content.

3.4 Article Content Extraction:

For each URL found in the search results, the script utilizes the newspaper library to download, parse, and extract the main text of the articles. It checks if the article can be successfully parsed before attempting to retrieve the first paragraph.

3.5 Data Filtering and Handling:
I applied various filtering techniques, such as checking for the existence of articles and ensuring that extracted paragraphs met specific word count criteria. I also processed and cleaned the extracted content before storing it in files.

3.6 Text File and Document Creation:
The script dynamically creates a text file to store the extracted paragraphs and further converts the data into a well-formatted Microsoft Word document (.docx) using the docx library.

4. Key Findings and Issues

Amongst the first issues I faced was the difficulty of the use of different libraries with different HTML code structures. Initially I tested the libraries lxml and newspaper for the entire task, as well as switching to just one website to search for articles on. This did yield partially successful results, however there were constant issues with certain websites not being scraped or much of the scraped data not being relevant. So, I decided to make use of 2 libraries to perform different tasks. Newspaper for the confirmation of an article published on the current URL and to scrape just the URL and append it to a list, and bs4 to scrape data from these URLs later.
In addition to this, there was also the issue of being blocked by Google for sending too many requests at one time. To tackle this, I used 3 approaches: introduced a time delay of 1 sec before each search prompt using time library, customized the number of search queries so that it could be set to a minimal number when possible, and made use of rotating user agents to bypass being blocked by Google.

There were also some issues with the final body of text in txt format, which often had irrelevant data and special characters. I added a few checks and conditions to remove these once the final text had been written to the file.

I found out about the docx Python library and how to utilize it for creation and manipulation of word files. Using this, the code formats, and styles the text from the txt file into a word file and stores a neat and presentable final form of the scraped data for the user.

5. Conclusion:

Through testing I was able to understand what to improve and target while scraping data. For instance, for the actual scraped paragraphs from each site, I set criteria that for the data to scrape, it had to be inside a <p> tag and consist of at least 20 words. This can be improved using other similar checks and manual methods, however, although my understanding of NLP is limited, I believe it could be better utilized to perform this task more efficiently.