CP 423 Text Retrieval & Search Engine – Assignment 1

Bryan Gadd (170451160), Herteg Kohar (190605160), Kelvin Kellner (190668940)

What each member contributed:

- Bryan Gadd (170451160)- Q2 and report.
    - Q2: Added coauthor for JSON+ changing Selenium to use Edge browser (vs Chrome).
    - Added brief understanding of questions.
- Herteg Kohar (190605160)- Q1, Q2, Q3 and report.
    - Q1: Implemented all the Q1 code for retrieving the URLs through various depths. Set up the functions for hashing and saving of the html content for this question and templated it for the other questions.
    - Q2: Utilized the templated hashing and file saving from the first question.
    - Implemented the bonus through selenium and coded the parsing of the HTML with BeautifulSoup.
    - Q3: Made the functions for hashing and saving of the files.
- Kelvin Kellner (190668940) - Q3 and report.
    - Performed all Q3-specific code (binary sequencing of HTML page content, optimization function code, displaying results via plot and .txt file)
        - Q3 starter code was based on a copy of Q1 code (accepting program arguments, parsing HTML page content, hashing URLs, and logging)
    - Wrote Q3 understanding, implementation, and issues sections of the report

Q1:

How we understand the question:

This question involves creating a general web crawler that will download the web content and branch off to explore other links that it finds within the web page. Some arguments are required for the program including a 'maxdepth' option which will limit how many links the crawler will branch off into (how deep it will go). Additionally, a 'rewrite' option will determine if the text file is re-written or not, and 'verbose' to print the URL and depth for debugging and watching the crawler in action. The 'requests' library was used to call the GET requests in order to get the HTML content from each of the links at each respective depth.

How we implement the question:

This question was split into two parts, one function handled the crawling of the page (retrieve, pull any links) while a second function handled hashing and saving the information to the text file. We used a universal regex to identify new links to explore.

Any challenges/issues that we had:

The first time testing this question was getting a 429-status code from a website. If a 429-status code is encountered the function exits and prints "Too many requests" to the console. Another issue was when the page had hrefs to endpoints on the same site. So, in this case the base URL was concatenated to the front of the endpoint in order to ensure it could be reached through requests.

<u>Q2</u>:

How we understand the question:

This question describes creating a web crawler that pulls specific fields of information from a certain site (Google Scholar). This seems like an example of pulling specific information from a specific site. To pull all the required information we will need to search the HTML document and pull the tags/elements that fit the description for each field (name, institution, coauthors, etc.).

How we implement the question (explain how we coded?):

This question was split into two parts, one function to parse the HTML content from each of the google scholar profile pages and another for the bonus in order to press the 'Show More' button enough times to be able to view all papers associated with the profile. Selenium was used in order to press the 'Show More' button until it was disabled. Once the button was disabled this was the indicator as to when all the papers were visible. The id of the button was found which was "gsc_bpf_more". This element was then found and pressed until it was no longer able to be pressed. To do this the click method was invoked on the element in a while loop which also included a call to sleep for a second between clicks in order to avoid misclicks. After this step, the HTML content had all the desired content to scrape and parse and using the driver object from Selenium the content was then passed to a function using 'BeautifulSoup' to parse the page. We began by parsing the profile specifics such as the researcher's name and profile picture and such things. Once we got to their papers a for-loop was used to iterate over all the rows and extract the paper-related information. Once the parsing was complete the HTML content was placed into a hash of the URL as its filename and the data scraped was placed into a JSON file with the hash of its URL as the filename.

Any challenges/issues that we had:

We initially coded the entire question in 'BeautifulSoup' but then looking into the bonus realized we needed to use 'Selenium' to simulate the "Show More" button. Luckily, we were able to combine both API's instead of rewriting the entire crawler for 'Selenium'. Another issue was the quick button presses would not be registered one after another since it took time for the new content to be displayed. The solution to this was to add a pause between button presses. This was achieved using the sleep function from the python 'time' module where a pause of 1 second was used between button presses.

<u>Q3</u>:

How we understand the question:

The focus of this question is to use a web crawler to identify and extract the "main content block" of a web page. The method is based on an observation that the main content block is usually the section with the fewest HTML tags per word/token, as opposed to headers, footers, ads, navigation, etc. which often have more HTML tags. The method replaces all tokens in the HTML with either a '1' for

HTML tag or a ‘0’ otherwise, and seeks to optimize a function f(i, j) whose optimum value represents the section with the most ‘0’s within range i,j and the fewest ‘1’s within i,j. See lesson or textbook 3.8 for the formula. All calculated function values must be plotted on either a 2D heat map or 3D surface map, and the resulting “content block” must be outputted to a file with the appropriate naming convention.

How we implement the question:

Our goal was to use modular programming to build each consecutive step toward the solution. Our docstrings describe a detailed breakdown of each step. The steps are loosely as follows:

In short, the program calls functions to 1) get HTML content of page 2) create list of spans for each token and each HTML tag in text 3) each token span is put in a new list with either a 1 or 0 flag indicating whether or not the token is part of an HTML tag, which is determined by spans 4) optimization function algorithm uses a clever trick to try all combinations of i and j for the function given in the lesson and textbook, the highest scoring combination is given as a token span 5) the resulting token span is then used to extract the content block from the webpage and block is saved to a text file 6) all optimization function outputs are plotting in a 2D plot using matplotlib.

Any challenges/issues that we had:

Coding for this question involved a lot of trial and error. Here were the roadblocks we encountered and the refinements we made to our approach in solving them:

- Our first iteration of binary sequencing used regex to replace parts of the original string to produce a new string with 0s and 1s in place of the appropriate tokens. However, this string alone had no information on how to retrieve the optimal content block from the original text.
    - We decided to use spans of token positions and generate a mapping of either 0 or 1 for each token span to keep track of everything, rather than converting to a string directly.
- Brute forcing all possible combinations of f(i,j) was too slow for my computer to output. Designing, coding, testing, and debugging a new algorithm for the optimization function took time and multiple iterations, as problems kept occurring.
    - We sketched our idea on paper, it was coded and tested. Each incorrect version was inspected and debugged. We restarted from scratch once for this portion of code.
- After correctly getting the optimum i & j values, there was an error where we were attempting to use a span indicating start/end TOKENs to retrieve the start/end CHARACTERs of the text.
    - It took time to realize why the output was incorrect. However, the issue was easy to resolve when discovered by using the token spans we created and mentioned earlier.
- After getting the 2D plot working correctly, we tried to generate a 3D plot as well, but the first solution we attempted via online resources loaded very slowly.
    - As our 2D plot worked fine, we did not bother to try to make a 3D plot a second time.