Web Search Engine on UIC Domain

CS 582 Term Project

Samujjwaal Dey
Computer Science Department
The University of Illinois at Chicago
Chicago, Illinois
sdey9@uic.edu

ABSTRACT

This document is a report for the final course project for CS582: Information Retrieval course at the University of Illinois at Chicago during the Spring 2020 term. The goal of the project is to design and implement a web search engine for the UIC domain. The search engine includes a web crawler, preprocessing and indexing of web pages, and an IR system implementing the vector-space model to retrieve webpages relevant to an input user query.

1 Software Description

The software is designed in Python(3.8), and all functions are modularized so that they can be extended further for use in other projects and future work. The search engine crawls, collects, and preprocesses 6000 webpages from the UIC domain, which takes around 3 hours to complete. So to execute the code without first crawling the UIC domain, all the pickle files needed to execute the code are included inside the 'PickleFiles' folder. The *search_query.py* script file can be executed from the terminal to execute the search engine right away. However, the python scripts to execute the web crawling and preprocess of webpages are also provided.

1.1 Web Crawler

The web crawler is executed by the script *uic_crawler.py*. Web traversal begins from the UIC-CS department page (https://www.cs.uic.edu/) and is restricted to crawl only within the UIC domain (https://www.uic.edu/). The crawler follows a Breadth-First Search (BFS) strategy with the UIC-CS page as the root node while maintaining a First in First Out queue to store URL links to traverse next. For each web page traversed, its link is appended to a list of crawled pages, the HTML content is downloaded and parsed using the *Beautiful Soup* library, and all the links present in the page are extracted. Each link encountered is appended to the FIFO queue only if it belongs to the UIC domain, does not have an irrelevant file extension, and is not already present in the queue.

The crawler performance is optimized by using a deque instead of a list to implement the FIFO queue, as a deque can remove elements from the head in constant time. In contrast, a list can remove elements only from the tail in constant time. For the BFS strategy, links are popped from the head of the queue, and hence deque was chosen.

22 file extensions were marked as irrelevant and ignored for web traversal as they took much time to download and did not point to valid web pages that can be parsed and preprocessed. The file extensions ignored are: '.pdf', '.doc', '.docx', '.ppt', '.pptx', '.xls', '.xlsx', '.css', '.js', '.aspx', '.png', '.jpg', '.jpeg', '.gif', '.svg', '.ico', '.mp4', '.avi', '.tar', '.gz', '.tgz', '.zip' .

The links were processed before being appended to the FIFO queue. The query parameters indicated by '?' and the intra-page anchors indicated by '#' were removed so that each URL added to the queue is a unique web page.

The BFS traversal of links gets terminated if the FIFO queue gets empty or on achieving the crawling limit. The crawling limit was initially set at 3000 pages but then finally increased to 6000 make the search engine more comprehensive. It took close to 3 hours to crawl 6000 pages.

Each web page is downloaded to the folder 'FetchedPages'. A dictionary saves the links of the pages that have been parsed and downloaded. The key for each link is the name of the file which stores the downloaded content of that link. The dictionary is saved on disk for later use by pickling it.

The file *error_log.txt* stores all the links that crawler could not parse/downloaded, along with the associated reason for the exception for that link.

1.2 Preprocessing

The preprocessing of the downloaded web pages is executed by the script *preprocessor.py*. For each downloaded page, a Beautiful Soup object is initialized. At first, all text on the webpage is extracted, and then all text that is never visible on a browser is excluded. To achieve this the text inside HTML tags <script>, <meta> and <style> are excluded. After this, the standard preprocessing steps are performed on the extracted text to tokenize the text: removal of all punctuations, numbers, and stop words. *Poster Stemmer* from the nltk library is used to stem each token. After stemming, any residual stop words and tokens with a length of 1 or 2 characters were removed. The tokens of each downloaded webpage are stored as a dictionary.

After all, the text preprocessing is complete, and tokens are formed, an inverted index is computed. This inverted index is a dictionary of dictionaries. Each token of the vocabulary of the corpus of web pages is the key, and its corresponding value is a dictionary of webpages where the token occurs. The filename of the downloaded web page is the key in the internal dictionary, and the count of the token on that particular page is the corresponding value.

The preprocessing of each downloaded web page and the computation of the inverted index takes around 20 minutes to complete. So, like the dictionary of all crawled pages in the crawler, the inverted index and the dictionary of tokens are saved to disk for later use.

1.3 Vector Space Model

The *search_query.py* is executed for taking in user queries and retrieving the most relevant webpages from the UIC domain.

At first, all the pickle files for the links crawled, tokens of webpages, and the inverted index are unpickled and extracted.

The Inverse Document Frequency for each webpage-token pair is calculated using the inverted index and stored as a separate dictionary. The frequency of the most frequent token in each webpage is calculated, which will be required to calculate the Term Frequency of each token for its corresponding webpage. Then finally, the TF-IDF value for each token in the corpus is calculated and stored as a dictionary of dictionaries like the inverted index.

Document length of each web page is calculated to form document vectors for each webpage.

A query is input by the user, and the query is preprocessed and tokenized, just like the webpage tokens. Then, the cosine similarity between the query each downloaded webpage is calculated. The webpages are then sorted in the decreasing order of cosine similarity scores and displayed to the user.

2 Challenges

- I am relatively new to Python programming language, so I was not aware of many trivial programming constructs in Python, like using a dictionary of dictionaries for the inverted index, using *deque* for faster access to the queue, storing files as pickle files for later use. But over the project, I learned many new Python tips and tricks for better programming.
- While executing an early version of the crawler, it took me 2 hours to crawl just 250 pages. I started printing out all the links that were being traversed and noticed many links were links to media, zip files, script files, or document/PDF files and not actual webpages. So, I made a list of all these irrelevant file formats and excluded such links from being fetched and parsed. The file extensions ignored are: '.pdf', '.doc', '.docx', '.ppt', '.pptx', '.xls', '.xlsx', '.css', '.js', '.aspx', '.png', '.jpg', '.jpeg', '.gif', '.svg', '.ico', '.mp4', '.avi', '.tar', '.gz', '.tgz', '.zip'. For a more comprehensive web crawler, document/PDF files could help in improving the precision of the search engine, but in my case, these files were just increasing the time to download and parse these files.
- When parsing each downloaded webpage and extracting the text from them, the Beautiful Soup object was returning all different kinds of words from the webpage. These were not English words and were forming junk tokens, which could have impacted the performance of the vector space model. So I searched online for a whole day and, in

the end, found a very helpful Stack Overflow post to address this issue. As per the post, on ignoring the text inside tags <script>, <meta>, <style>, etc. HTML tags, the text fetched are the words that are actually visible on the webpage on a browser. Practically a user is looking for only this text when searching on a search engine.

3 Weighting Scheme and Similarity Measure

3.1 Weighting Scheme

The weighting scheme used in the project was the simple TF-IDF of words in webpages. Although it is a relatively simple and straightforward weighting scheme, over the years, it has proven to be one of the most effective and efficient weighting schemes. The importance of tokens in each document is accounted for in a very unbiased manner, which was perfect for this project.

3.2 Similarity Measure

The similarity measure used to rank relevant webpages was the Cosine Similarity. Cosine similarity takes into consideration the length of both the document and the query. Moreover, the tokens that are not in either the document or the query do not affect their cosine similarity. Usually, the query is relatively short, and therefore its vector is extremely sparse, which means similarity scores can be calculated quickly.

3.3 Alternative Similarity measures

Other similarity measures instead of cosine similarity include Inner Product, Dice Coefficient, and Jaccard Coefficient. The Inner Product does not consider the length of the document or query and hence does not give good results when used. Cosine Similarity measure itself is a modification of the Inner Product measure. I have not tried the other similarity measures for implementing this project.

4 Evaluation of Queries

The IR system was evaluated using five custom queries and the top 10 webpages retrieved for each query.

1. Query: Cornelia Caragea NLP

```
Enter a search query: Cornelia Caragea NLP
1 https://www.cs.uic.edu/~cornelia/
2 https://cs.uic.edu/profiles/cornelia-caragea
3 http://cs.uic.edu/profiles/cornelia-caragea
4 https://nlp.lab.uic.edu/
5 https://engineering.uic.edu/news-stories/university-scholar-barbara-di-eugenio
6 https://cs.uic.edu/news-stories/university-scholar-barbara-di-eugenio
7 https://today.uic.edu/university-scholar-barbara-di-eugenio
8 https://cs.uic.edu/profiles/barbara-di-eugenio
9 https://cs.uic.edu/news-stories/cs-welcomes-13-new-faculty-members
10 https://engineering.uic.edu/news-stories/cs-welcomes-13-new-faculty-members
```

All webpages retrieved were related to the query. Five results are directly related to Prof. Cornelia Caragea, and rest are associated with NLP.
Precision = 1.0

2. Query: INfoRmaTioN reTrIEVal

```
Enter a search query: INfoRmaTioN reTrIEVal
1 https://cs.uic.edu/profiles/cornelia-caragea
2 http://cs.uic.edu/profiles/cornelia-caragea
3 https://engineering.uic.edu/letter-from-the-dean-spring-2020
4 https://registrar.uic.edu/student_records/transcripts
5 http://www.uic.edu/depts/oar/student_records/transcripts.html
6 https://registrar.uic.edu/current_students/transcripts.html
7 https://researchguides.uic.edu/dataplans/preservingdata
8 http://csrc.uic.edu
9 https://csrc.uic.edu
10 https://transportation.uic.edu/abandoned-bicycle-removal
```

Each retrieved web page has the query terms in them.
Precision = 1.0

3. Query: Internships & Jobs

```
Enter a search query: Internships & Jobs
1 https://career.las.uic.edu/internship-program
2 https://careerservices.uic.edu/students/internships
3 https://career.las.uic.edu/about
4 https://studyabroad.uic.edu/internships
5 https://engineering.uic.edu/news-stories/reaping-the-benefits-of-an-internship-experience
6 https://engineering.uic.edu/undergraduate/guaranteed-paid-internship-program
7 http://engineering.uic.edu/undergraduate/guaranteed-paid-internship-program
8 https://cs.uic.edu/news-stories/reaping-the-benefits-of-an-internship-experience
9 https://ecc.uic.edu/freshman-internship-program
10 https://ecc.uic.edu/students/freshman-internship-program
```

All webpages retrieved were related to internship programs or news stories about internships.
Precision = 1.0

4. Query: Fall Career Fair

```
Enter a search query: Fall Career Fair
1 https://ecc.uic.edu/employers/career-fairs
2 https://careerservices.uic.edu/students/career-programs-events
3 http://careerservices.uic.edu/students/career-programs-events
4 https://researchguides.uic.edu/copyright/fairuse
5 https://careerservices.uic.edu/most-popular/career-fair-tips
6 https://ecc.uic.edu/events-2/engineering-career-fair
7 https://ecc.uic.edu/engineering-career-fair
8 http://careerservices.uic.edu/uic-career-fair-descriptions
9 https://researchguides.uic.edu/copyright/myths
10 https://registrar.uic.edu/tuition/grad/index.html
```

Of all the webpages retrieved seven results are related to career fair events, tips, and descriptions career fairs.
Precision = 0.7

5. Query: research assistantships

Eight webpages retrieved are related to different assistantship opportunities available at UIC. Precision = 0.8

5 Results

- As seen from the evaluation of the sample queries, the precision for all queries is pretty good.
- Additionally, the top 3 most relevant webpages returned are actually the top results for all queries.
- Another point evident from the query results is that in a vector space model IR system based on the TF-IDF scheme, words that are more frequent in a document play major role in determining the topic of that particular document. Words like 'internship', 'career', and 'job' are not present in all documents, but they occur very frequently within their own documents. So, this reduces the ambiguity in retrieving the most similar documents as TF plays a significant role in ranking documents.
- But it is also evident from the results of the queries that the query results returned might not always be relevant to the context intended by the query. For instance, in query 2, the context of the query 'Information Retrieval' could be to search for the course' CS582: Information Retrieval'. But the pages returned have no connection with the course 'Information Retrieval'. Except for the first query result returned, all other webpages are just webpages that have the occurrences of words 'inform' and 'retriev' (the stemmed query tokens).
- An error that can be noticed when evaluating the query results was that some links are repeated in the results. Even though the crawler checks for duplicate links during link traversal, there are some links that are present in webpages with both 'http' and 'https' extensions. The crawler at present treats the links which point to the same page but have different HTTP format as distinct links. This can be worked on in the future.

6 Related Work

I read a few online resources to learn how to perform link traversal and web crawling, implementing Breadth-First Search in Python, parsing webpages' text, and efficiently calculating cosine similarities.

I also found a couple of papers, one discussing the integration of the Text Rank algorithm in a vector search model and the other discussing how to return results relevant to the context of the search query in a search engine.

7 Future Work

The precision of the web search engine implemented is pretty decent and acceptable, given the fact that it was developed from scratch.

But the performance of the search engine can be improved a lot by adding a host of other functionalities to the search engine as a whole.

- For the web crawler, relative URLs can be converted using the current page's URL, implementing Multi-Threaded Spidering and implementing Directed Spidering strategies like Topic-Directed or Link-Directed Spidering.
- Text Rank algorithm and a query context determining algorithm integrated with the IR system will further improve the precision and retrieve much more accurate pages.
- Also, the crawled pages need to be updated from time to time so that the search results are updated.
- A GUI application can be made as the front end for the search engine.