Web Search Engine on UIC Domain

CS 582 Term Project

Anjani Sruti Doradla
Computer Science
University of Illinois Chicago
Chicago, Illinois
adorad4@uic.edu

ABSTRACT

This document is a report for the final course project for CS 582: Information Retrieval course at the University of Illinois Chicago. The goal of this project is to create and put into operation an online search engine for the UIC domain. A web crawler, preprocessing, indexing of web pages, and an IR system that uses the vector space model to obtain web pages pertinent to a user query in the Command Line Interface are all included in the search engine.

1. SOFTWARE DESCRIPTION

The software is designed in Python and all its functions have been modularized to allow for future expansion and use in further projects. In about two hours, the search engine retrieves, crawls, and preprocesses 3500 web pages from the UIC domain. The 'PickleFile' folder contains all the pickle files required to run the function, allowing you to do so without first browsing the UIC domain. The search engine can be launched immediately by running the search_engine.py file from the terminal. However, individual python files are also provided to carry out web crawling and preprocessing.

1.1 Web Crawler

This module is in charge of crawling web pages to gather all the links and page material that will be utilized to populate the search engine's results page. The crawler is implemented by executing the crawler.py. The domain is – "uic.edu" and the traversal starts from the UIC-CS department page – https://www.cs.uic.edu/.

While keeping a First in First Out queue to store URL links to explore next, the crawler uses a Breadth First Search (BFS) approach using the UIC-CS page as the root node. The BFS strategy uses the base URL as the only element.

We do the following checks to see if the URL-pointed web page can be accessed in each traversal iteration by popping the leftmost element from the deque**.**

- Check if the URL belongs to the UIC domain.
- Check if the web page linked to the URL is an HTML page.

A deque is used because it can remove elements from the head in constant time and for the BFS method of traversing, links are popped from the head of the queue. The HTML content is downloaded and parsed using the Beautiful Soup library for each web page visited, and all links present on the page are extracted. The link for each web page visited is then appended to a list of crawled pages.

Some file extensions such as – '.pdf', '.doc', '.docx', '.ppt', '.pptx', '.xls', '.xlsx', '.css', '.js', '.aspx', '.png', '.jpg', '.jpeg', '.gif' were irrelevant to the search and crawling that needed to be ignored during the web traversal because they took a long time to download and did not link to legitimate web pages that could be parsed and processed. Before being added to the FIFO queue, the links were processed. To ensure that each URL added to the queue is a distinct web page, the query parameters denoted by "?" and the intra-page anchors denoted by "#" were eliminated.

The document is then parsed using the Beautiful Soup library to remove any unnecessary SGML tags (for example, <TITLE>, <DOC>, <TEXT>, etc.) mentioned above and to obtain the other URLs in the document. These URLs are extracted and added to the list of URLs to be crawled if they have not already been crawled.

When the URL queue, that is, the deque is empty, in this case, 3500 web pages, the web traversal procedure continues and took two hours to complete the whole process.

Every website page is downloaded to the "FetchedPages" folder. Links are downloaded and processed web pages are saved in dictionaries. The name of the file that contains the link's downloaded content serves as the key for each link. The dictionary is pickled and stored on disk for future use, that is, the crawled pages are pre-processed in the next steps.

1.2 Pre-Processing

This module takes care of pre-processing of the fetched web pages which does text processing, removes stopwords, and performs stemming using the nltk library. This module calculates the

inverted index and stores it. Nltk is a standard Python package that offers a variety of NLP methods.

The preprocessing of all the crawled pages is implemented by a python file – pre_processor.py. A Beautiful Soup object using the Beautiful Soup library is created from scratch for each page that is crawled. The text in the HTML tags - 'style', 'script', 'head', 'meta', '[document]', 'link', 'noscript', and 'svg' are excluded because they are not visible on a browser. All the text on the webpage is extracted.

Later, on the extracted text – tokenization is performed where the text is converted into tokens. The process of tokenization involves converting the text into lowercase, removing all the punctuations, separating the numbers from the text and replacing them with " ", and splitting on whitespace.

For stemming, Porter Stemmer from the nltk library is used to stem each token that is obtained after tokenization. After stemming, stop words and very short tokens were removed less than the length of one or two characters are removed. The resulting set of tokens for each webpage was saved as a dictionary.

After the text preprocessing is complete and the tokens are created from the extracted text, an inverted index is created. This inverted index is a dictionary that maps each token to a dictionary of web pages where the token appears. The key in the internal dictionary of the inverted index is the filename – (for example – FetchedPage 123) of the webpage and the value is the number of occurrences of the token on that page, that is, the term frequency in that particular page.

After the inverted index of the corpus is created, it is saved in a pickle file for all the 3500 documents for future use. The pickle file is stored in a folder called "Pickle Files".

1.3 Vector Space Model

This module takes care of taking in user input as a query and retrieving the top 10 relevant web pages from the UIC domain.

At the beginning of the process, the pickle files containing the saved links, tokens, and inverted index are loaded and extracted. The data from these files will be used to calculate tf-idf and cosine similarity.

The Document Frequency is calculated which is the value of how many times is the token present in each document. The Inverse Document Frequency is calculated which is a webpage-token pair using the inverted index and term frequency. Document frequency and Inverse Document Frequency are both stored in dictionaries with the respective values calculated.

The term frequency is already calculated in the pre-processing step and is stored in the inverted index. This dictionary is loaded from the pickle file and the term frequency is used to calculate tf-idf.

Since we have obtained all the values for calculating tf-idf, the tf-idf is calculated for each token in the corpus of UIC domain and stored as a dictionary like the inverted index dictionary.

Similarly, document length and document length vectors are calculated and stored in a dictionary for calculating cosine similarity.

A query input is taken from the user and pre-processing is performed on the query. The pre-processing included converting the text into lowercase, removing all the punctuations, separating the numbers from the text, replacing them with " ", and splitting on whitespace. Stemming is also performed on the query to keep it uniform and comprehensible for the search.

Next, the cosine similarity between the user's query and each webpage in the corpus is calculated.

This module provides a command-line interface for users to enter a query and receive a set of top 10 related URLs. The module loads the pickle files containing the inverted index, webpage tokens, and crawled pages and applies cosine similarity to the user's query and all indexed web pages to find the best matching URLs. These URLs are then displayed to the user with a choice of the user to continue or exit.

2. CHALLENGES

- As a beginner in the Python programming language, I was not familiar with some common programming constructs such as using a dictionary of dictionaries for the inverted index and storing them as pickle files for later use.
- The crawler took a lot of time to load, around 2 hours to finish and had to ignore many extensions to achieve to finish in 2 hours. Otherwise, the crawler took a lot of time to execute and parse the files. I would have used more than 3500 pages for a more comprehensible search engine if the crawler took less amount of time.
- Implementing the PageRank algorithm required some careful thought because it was being applied to a subset of web pages rather than the entire web. This did not work out well which did not give proper results.
- Pre-Processing everything and reading tokens multiple times for testing led to running the crawling file many times. This led to more time for crawling and waiting for all the pages to be processed. This could be avoided by saving the files in another way.
- When processing each crawled webpage and extracting the text, the Beautiful Soup object was returning a variety of non-English words that were forming "junk" tokens. These could have negatively impacted the performance of the vector space model.

- One of the challenges I faced was designing the software and dividing it into different modules based on their functionality, as well as incorporating object-oriented programming to improve the readability of the code since the format of assignments and the project was different.

3. WEIGHTING SCHEME AND SIMILARITY MEASURE

3.1 Weighting Scheme

The weighting scheme used in this project is tf-idf. The tf-idf scheme is easy to compute, has some basic metrics to extract the most descriptive terms in a document, and can easily compute the similarity between two documents, that is a fetched page and a query used in the project. It assigns a weight to each term in a document based on its importance. This weight is proportional to the term's frequency within the document and inversely proportional to its frequency across all documents in the corpus.

3.2 Alternative Weighting Scheme

Document frequency-inverse corpus frequency (DF-ICF) can be used to find the importance of a particular document within a given corpus. However, for the project at hand, it is not necessary to find the statistics between a document and the entire corpus. Therefore, using TF-IDF is more suitable and will provide reliable results for this project.

3.3 Similarity Measure

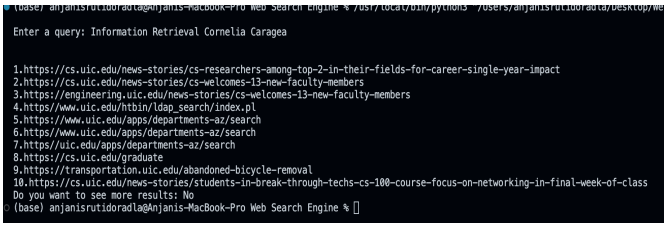
The similarity measure used to rank relevant web pages was cosine similarity. Cosine similarity is a measure that considers the lengths of the document and query, regardless of their size. It is often used in web search engines. This measure also considers the lengths of both the document and the query and ignores tokens that are not present in either. Because queries are typically short, their vectors are often very scattered, which means that the cosine similarity score can be calculated quickly. This makes cosine similarity a good choice for ranking web pages in this project.

3.4 Alternative Similarity Measure

In contrast, the inner product is the number of matched query terms in the document, which can be used to calculate similarity but does not take into account the lengths of the query and document. As a result, it may not produce reliable results. Therefore, cosine similarity is a more appropriate measure for this project and will be used in the web search engine.

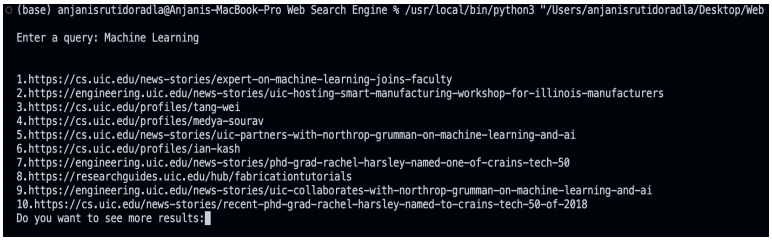
4. EVALUATION OF QUERIES

1. Query 1 – Information Retrieval Cornelia Caragea



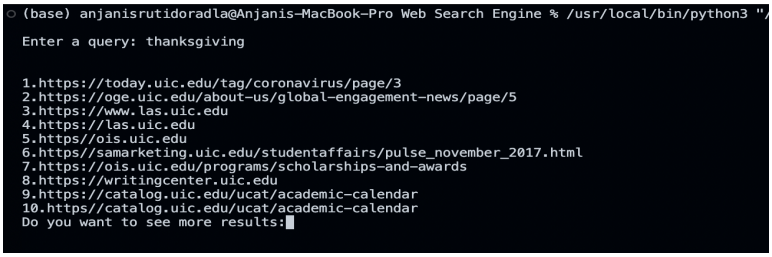
All the webpages shown above were closely related to 'Cornelia Caragea' since the stemmed words were present greater than the stemmed words of 'Information Retrieval'. Precision – 0.7

2. Query 2 – Machine Learning



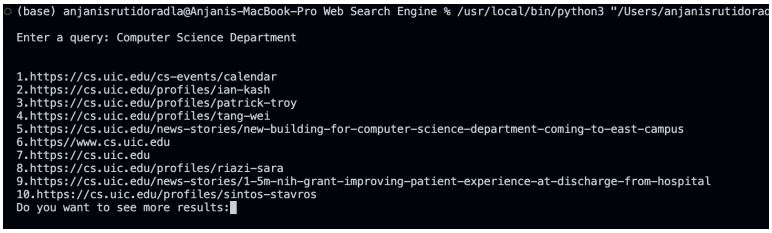
All the web pages shown above were closely related to 'Machine Learning'. Precision – 1.0

3. Query 3 – Thanksgiving



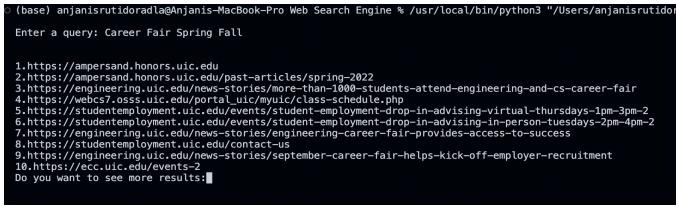
All the web pages shown above were closely related to 'Thanksgiving'. Precision – 0.9

4. Query 4 – Computer Science Department



All the web pages shown above were closely related to 'Computer Science Department'. Precision – 0.9

5. Query 5 – Career Fair Spring Fall



All the web pages shown above were closely related to 'Career Fair Spring Fall'. Precision – 1.0

The results were moreover relevant to what is required in the query provided by the user.

5. RESULTS

- The search engine performed well on custom queries, with many of the results being relevant to the search terms. Most of the time, the search engine was able to find the pages that we were looking for. In addition to this, the top five URLs retrieved are the most relevant.
- The user can also select to search for more results by choosing 'Yes' or 'Y' as a choice to display the next ten relevant URLs until the searches become zero.
- The HTTP and HTTPS are handled. Some URLs are the same when they were displayed reducing the efficiency of the search. The duplicate URLs which just differed on HTTP and HTTPS are also taken care of. Although, some links are seen to be the same because the same links are present in one or more documents.
- Some of the search results were not relevant to the query context due to stemming. For example, the query "cocktail party" would be stemmed as "parti", which could result in search results containing the word "parties". These results would not be relevant to the original query context. Stemming can sometimes produce unwanted results like this, so it is important to carefully consider whether to use it in a particular application.

6. ERROR ANALYSIS

- Despite handling all the duplication issues, some URLs got repeated in the search. This shows that there is some dependency among the results and needs to be taken care to avoid duplicate information.
- I noticed from the above queries that the use of tf-idf and cosine similarity tends to give more weight to term frequencies, which can impact the results of queries containing common terms like 'computer', 'graduate', and 'courses'. This is because these terms are likely to appear frequently in many documents, which could make them more important in the calculation of similarity scores. As a result, some search results may not be as relevant as they could be.

7. RELATED WORK

To build the web crawler, I researched various third-party crawlers and some websites. Using all the available resources online and some research papers, I developed a Search Engine from scratch using the libraries provided by Python. In addition, I worked on the assignments provided in class and incorporated any relevant parts into the search engine. This allowed me to gain a better understanding of the concepts available online and integrate all the ideas and techniques used in the search engine together.

8. FUTURE WORK

In spite of handling all the technicalities of a Web Search Engine, there could be other functionalities that can be added which are –

- Increase the number of web pages crawled for better search results and a bigger pool of URLs.
- Experiment with other ranking algorithms like Page Rank for a better indexing scheme.
- A better UI can be made for user understanding to enhance search experience and better readability of resources.
- A better Web Crawler like Robot Parser can be used to parse robot.txt files and a better way like distributed crawling can be used to decrease the loading time of the fetched files.