UIC Web Search Engine

Information Retrieval

Vinay Kumar Valluru
Computer Science Department
University of Illinois at Chicago
Chicago, IL
vvallu3@uic.edu

ABSTRACT

This is a report for the final project of CS 582: Information Retrieval at UIC. The goal of this project is to design and implement a search engine for the UIC domain, that includes components for web crawling, webpage processing and indexing. The search engine should have an intelligent aspect and a friendly user interface.

I have implemented a multithreaded web-crawler and used TF-IDF weighting schema along with Matching Score ranking to determine the list of relevant webpages for a search query.

1. COMPONENTS

1.1 Introduction

The search engine is written in python programming language with independent components for each function of the search engine. I have written separate components for web crawling, indexing, searching and the user interface. Since web crawling is a time-consuming process, I have stored the webpages in separate files in json format along with their corresponding URLs.

Similarly, after the preprocessing is done on the webpages, their TF-IDF scores are calculated and stored in separate files in the json format for easy access. This is done to ensure that the search function can be used instantly without having to wait for the web crawling and indexing to complete. These components are explained in detail in the following sections.

1.2 Web Crawling

A web crawler is deployed on the UIC domain to gather web pages required for the search engine. The crawling starts from the computer science website on the UIC domain: https://www.cs.uic.edu/ and proceeds further. The traversal is done using a breadth first search strategy where all the links present on a webpage that belong to the UIC domain are processed in order before moving on to the links present on each of those web pages. This is achieved by implementing a First In First Out Queue to store the relevant URLs found by the crawler.

Each web page is parsed, and all the URLs present in the webpage are extracted. If the webpage is pointing towards a file instead of a website, the URL is eliminated. The following are the formats that I have ignored: ".docx", ".doc", ".mp4", ".jpg", ".jpeg", ".png", ".xls", ".pdf", ".xlsx", ".zip", ".exe", ".js", ".css", ".ppt". Apart from these, the URLs containing intra-page links ("#") are also removed, so that the same websites are not processed again. The URLs containing search parameters are also eliminated.

When a website returns a response within the timeout period, it is parsed, and the irrelevant tags are eliminated. Then the text is extracted and stored in a json file along with the corresponding URL of the page. This process is continued until 3000 webpages are stored making sure that no duplicate pages are stored.

Since web crawling can be a very time-consuming process, I have used the concept of multithreading to increase the process speed. I have set the number of threads to 30 which concurrently gather webpages. The code for the crawler can be found in 'web_crawler.py'.

1.3 Webpage Processing

All the webpages stored by the crawler are accessed for preprocessing. The text on each page is split into words, converted to lowercase and unnecessary symbols are removed. Then using the *PorterStemmer* and *stopwords* packages from the python *nltk* library, all the words are stemmed and any stop words are removed in every page.

Any tokens of length shorter than three are removed from every page.

The TF-IDF is calculated for all the pages and their values are stored in separate files for each webpage. This is done to ensure that every time a new search query is processed, the TF-IDF need not be calculated for all the webpages again. The preprocessing and creation of TF-IDF for all the webpages is done in the 'TF_IDF.py' file.

1.4 Query search

The code for the search function of the search engine is implemented in the 'search.py' file. Similar to the preprocessing steps followed on the webpages, the query is also tokenized, stemmed and stop words are removed. Any tokens of length less than three are also removed.

Based on the user query, the top 200 documents are selected and converted to TF-IDF vectors. Then the TF-IDF of the query is calculated and it is also converted to a TF-IDF vector. The cosine similarity between the query and these documents is calculated and the top 10 documents are returned.

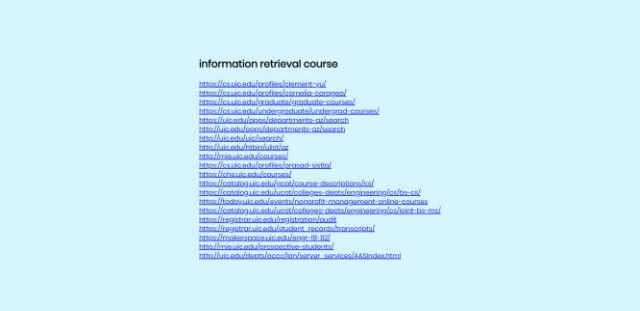
1.5 User Interface

I have used the *Flask* web development framework for python to create the user interface for the search engine. I have created a basic website using *html*, *css* and *javascript* and hosted it on the local host.

The homepage of the search engine looks as follows.

If the user has searched for the query "information retrieval course", the top 10 results are displayed as follows.

There is an option for the user to get more results which can be accessed by clicking on the "more" button.



The search engine can be accessed by running the 'app.py' file and opening the url: http://127.0.0.1:5000/ in the browser. I have created two html pages for the home page and the results page.

2. MAIN CHALLENGES

The main challenges encountered during the development of the search engine are:

- It was very difficult to crawl thousands of webpages at the beginning as it would take a lot of time to get the response and store the webpages one after the other. So, I had to learn to implement the multithreading concept in python and make the threads access the queue in parallel. This has reduced the time required to crawl to only a few seconds.
- I had to learn the hard way about html parsing and formatting the URLs correctly as I was getting a lot of broken links.

- It was also difficult to create TF-IDF vectors over such a large corpus as it took a lot of time to execute. So, I had to store the TF-IDF values for each webpage and then create vectors only of the top relevant webpages based on the query.
- Designing a user-friendly interface has also been a main challenge as I was not well-versed in html. I had to learn how to use Flask components to integrate the webpages with python and use python variables to display data on the page.
- I also had to eliminate several file types, search parameters and sub-components of the webpages from the URLs.

3. WEIGHTING SCHEME AND SIMILARITY

3.1 Weighting Scheme

I used the standard TF-IDF weighting scheme for this project. TF-IDF is one of the most effective weighting schemes to calculate the importance of words in a document. In terms of alternatives in TF-IDF, I could have used logarithmic TF, word frequency, etc. but simple TF-IDF weighting seemed the right choice.

3.2 Similarity Measure

The similarity measure I used to rank the webpages was Cosine Similarity. Other alternatives would have been inner product, etc. but cosine similarity is more comprehensive and often gives the best results. So, I had to choose cosine similarity measure for this project as it works very well in practice.

4. EVALUATION

I did a manual evaluation of the top 10 results for 5 sample queries:

- Query: “information retrieval course”
  It has returned links to professors working on or teaching information retrieval and also links to the course description which is pretty much what I was looking for.
- Query: “computer science courses”
  It has returned links to the list of graduate and undergraduate courses being offered at UIC, which is my expected result.
- Query: “career fair”
  It has returned links to the career services website and also to a list of upcoming events which is very accurate.
- Query: “student center east”
  It has returned links to the student centers website and also links to several events which is mostly accurate.
- Query: “spring 2019 courses”
  It has returned links to the courses offered in various departments which is very accurate.

5. RANKING USING MATCHING SCORE

Since it is difficult to calculate or store the TF-IDF vectors for all the webpages every time a query is searched, I have chosen to implement an intelligent aspect of my search engine i.e. Matching Score.
Matching score is a way to calculate the similarity between the query and the webpages. In this method, we need to add up TF-IDF values of the tokens that are in query for every webpage. For example, if the query is “hello world”, we need to check in every webpage if these words exists and if any of these words exists, then the TF-IDF value of that word is added to the matching score of that particular webpage. In the end I have sorted and got the top 20 webpages.

The TF-IDF vectors of these webpages is then calculated and cosine similarity is used with the TF-IDF vector of the query to rank them. The URLs of these webpages are then returned to the html page and displayed in the form of links.

6. RESULTS

When compared to a normal search engine, all the queries searched in my search engine returned similar results when using a domain search on the normal search engine. If the search queries contain ambiguous terms, the results may not be very accurate.

However, as discussed in section 4, for a query like “computer science courses” the first result itself is exactly what I was looking for. For most of the queries I have always found the relevant links in the top 10 results

displayed by the search engine. I rarely had to get more results to find a relevant link for the query.

7. RELATED WORK AND FUTURE CONSIDERATION

In case of related work, page rank can be used as a good algorithm to get relevant search results if we are not doing domain specific search. Since authorities did not affect the relevance of results much in a domain specific search engine, I have chosen not to implement page rank here.

Considerations for future work include query expansion, optimizing the query processing times, etc.

8. REFERENCES

[1] https://colorlib.com/wp/free-css3-html5-search-form-examples/

[2] https://projects.raspberrypi.org/en/projects/python-web-server-with-flask/7

[3] https://edmundmartin.com/multi-threaded-crawler-in-python/