UIC Search Engine

CS-494 Information Retrieval

Nancy Pitta
UIN: 672134497
University of Illinois at Chicago
Chicago, IL USA
npitta2@uic.edu

OVERVIEW

This document is a report for the search engine that was built as a part of course project for CS 494 - Information Retrieval. The project consists of building a search engine for UIC domain from scratch.This is built modularly and the execution starts from crawler followed by pages preprocessing, indexing and finally adding a command line prompt to take query as input from user. This project can be run by executing 'python search_engine.py'.

MODULES

The project is built in Python(3.7) and trained over 3700 documents. The main components in this engine are crawler, text preprocessor and search engine interface. The crawler starts at cs.uic.edu and runs till it gathers as many links as specified in the crawl limit variable. Once it crawls the crawl limit number of pages, which in our case is 6001, it stores the links information in a pickle file. Once that is done, the control goes to the next main component in our project preprocessor.

Crawler

Web crawling is the first step of developing the search engine and is important to gather the data required to create inverted index. Initially we start at cs.uic.edu and add this url to a queue that stores urls. The following extensions were ignored : 'pdf', '.doc', '.docx', '.ppt', '.pptx', '.xls', '.xlsx', '.css', '.js', '.aspx', '.png', '.jpeg', '.gif', '.svg', '.ico', '.mp4', '.avi', '.tgz', '.gz', 'tar' and 'zip' (ignoring all extensions without html content).

The crawler works on BFS algorithm by maintaining a queue data structure that stores the urls, whose content is yet to be crawled, to extract out links from the url's HTML content. The crawler starts with cs.uic.edu as the root node and searches for any out-links from the crawled URL's content belonging to the UIC domain. This crawler recursively checks the url queue, until it is empty, polls the first url from the queue and crawls its content for any out links. The crawler's performance is optimised by using deque operation instead of using a list to implement FIFO queue, as deque can remove elements from the head in constant time. Whereas a list can remove elements only from the tail in constant time. For Breadth first search algorithm, elements are popped from the head of the queue and hence deque was chosen.

For each link found in popped url's html content, it checks if it doesn't have the above mentioned extensions and whether the link starts with either 'http' or 'https'. Then it does some cleaning on the url string like removing the tail slash, removing query parameters appended to the URL (parameters that follow after '?' in the url string) and intra page anchors were removed so that each URL is a unique web page. Then it is converted to lower case and appended to the url queue. Now to avoid storing duplicate content, we check if this link is already crawled and finally we check if the url belongs to uic.edu domain.

We also store the fetched pages inside a separate directory with their names as the order number in which it was found by the crawler. We also maintain a dictionary 'pages_crawled' to keep track of the order in which url are crawled with index as key and url string as value. This dictionary is then stored in a pickle file for easier access, later on, to avoid long crawling time all over again.

The crawled pages' HTML content is stored in './FetchedPages' directory and the dict of crawled pages urls is dumped to a pickle file and is stored in './PickleFiles/'. The crawler can be executed by running the script 'uic_crawler.py'.The file 'error_log.txt' stores all the links that crawler could not parse/access along with the associated reason for the exception for that link.

The BFS traversal each time, checks if it has traversed the crawl_limit number of pages. The page limit was initially 3001 but then we increased it to 6000 to make the search engine more comprehensive. My crawler was unable to crawl more than 3560 pages so I imported few other pickle files from the GitHub accounts and added it to the pickled files directory in current project.

Preprocessor

The processing of the downloaded web pages is executed by the script 'preprocessor.py'. In this script, we preprocess the text mainly involving steps like removing stop words, stemming. The stop words set used in this project is the set of stop words from nltk.corpus. The stemmer used in this project is PorterStemmer, imported from nltk.stem. We use beautifulsoup library to parse the pages content. Then we create an inverted index which maps the words to corresponding list of documents in which it appears.

The fetched html files from directory ‘FetchedPages’, generated in the previous step, while crawling web pages starting from cs.uic.edu, are used to generate the word tokens. While parsing the fetched pages, we can skip parsing content that is not visible on the HTML page, as it wouldn’t contain any out-links that user can interact with or information that is useful to classify the document or to make it visible in search results. So we avoid parsing the content inside tags like ‘<style>’, ‘<script>’, ‘<head>’, ‘<meta>’ and ‘[document]’. We also avoid parsing HTML comments and other insignificant tags like ‘\s’, ‘\r’ and ‘\n’. We keep track of the list of words present in a document using ‘webpage_tokens’ dict. After fetching the text from only visible content, standard preprocessing steps were performed. Stop words were removed and the tokens present were stemmed. Any stemmed tokens of length 1 or 2 were dropped.

Inverted index was implemented as a dictionary of dictionaries, where the key-value pairs present are in the form of: { stemmed_token : { file_name : frequency_of_token_in_file } }. Each word token is the key of the inverted index and its corresponding value is a dictionary of webpages. The key of the internal dictionary is the file name in which the word token appears and its corresponding value is the frequency of the word token in the file.

The preprocessing step takes nearly XX minutes to compute the inverted index. We store this inverted index in a pickled file for future use in our search engine at ‘./PickledFiles’ directory named as ‘6000_inverted_index.pickle’ and also store the word tokens inside same directory as ‘6000_webpages_tokens.pickle’.

Vector Space Model

The search engine needs the web pages word tokens to match with the user input to find relevant documents and then it needs the inverted index to fetch and display the frequency of word token occurrence and finally we need to display the urls. To get all these information, we unload all these information from the pages crawled pickle file generated during crawling. We then get inverted index information from the ‘6000_inverted_index’ pickle file generated during text preprocessing. Finally to get the URL of the documents in which the word token is present, we need the ‘pages_crawled’ pickle file, which contains mappings from file names to the corresponding url where the current file/webpage (file contains the html content of the webpage at the url) can be found.

We are using TF-IDF as a measure of importance for a given word token in the corpus. The frequency of the most frequent token in each webpage is also calculated, which in turn is used in calculating the term frequency term of TF-IDF. So, we begin with calculating the tf-idf values for all the clean stemmed tokens in the inverted index. These values are stored as a dict of dicts just like the inverted index.

Next step is to calculate the document vector length for each of the webpages. To calculate document vector length, we utilise the webpage tokens dict present in ‘6000_webpages_tokens’ pickle file. Iterate over each of the url present in the webpage_tokens dictionary keys and for each url, fetch the list of clean stemmed tokens from the webpage_tokens values as webpage_tokens[url]. Now we calculate the squared sum of tf-idf scores for each of the stemmed word tokens in the given url, which can be accessed like tf-idf[token][url]. Then after calculating squared sum of all tf-idf scores, take a final square root of it and return it as the document length.

Next we generate a window using ‘tkinter’ and take the input from the user. Once the user enters the query and presses ‘Search’ button, we begin with tokenising the query, then preprocessing it to clean stemmed tokens.

For each of the stemmed tokens in the query, we calculate the tf-idf scores and we calculate query vector length (in a similar way we calculated the document vector length for web page tokens before) by taking the squared sum of tf-idf scores of all query tokens and taking a final squared root and returning it as query score.

Next we iterate over each token in the query, get its weight (which is its tf-idf score) and iterate over all pages the token is present in and fetch its tf-idf scores. Then we calculate similarity between the token and the document at the url by multiplying token’s tf-idf for this particular document with the tf-idf of token in the query. We store all these similarity values in a dictionary with key as the url and value as the similarity score. Finally for all calculated similarity scores for each url/page, we divide them by the length of the query and by the length of the tokens present in the given url/page.

CHALLENGES

- I am new to Python programming and hence I initially coded most of the part in a lengthier way rather than the Pythonic way but modified it later using various resources online. Optimising code written by comparing various data structures and comparing them was another major challenge. Using deque for faster access to the queue, storing files as pickle files for later use are some of the new concepts that I have learnt while doing this project.
- While executing the crawler initially it took the crawler almost half a day to crawl 1000 pages but it was still stuck. So finally I found that few extensions like ‘.pptx’, ‘.doc’, ‘.xlsx’, ‘.tar’, ‘.gz’, ‘.ppt’, ‘.pdf’, ‘.js’, ‘.png’, ‘.jpg’, ‘.jpeg’, ‘.zip’, ‘.mp4’, ‘.avi’, ‘.ico’, ‘.svg’, ‘.gif’ were irrelevant and slowing down the crawler. These extension types are consuming lot more time to be downloaded and then parsed. Initially tried parsing just 3000 documents but due to very low processing power of my laptop, it took hours and hours to crawl just 1000 pages.
- Due to the low processing power of my device, the crawl limit was set to 3580 but it took half a day to crawl these many pages and the results were not relevant. Hence I executed the code on google colab for faster crawling.
- Beautiful Soup parses many different types of words that are not English words, but basically junk tokens. This might have impacted the performance of the vector space model. This was one part where I was stuck and found many posts online to understand Beautiful Soup’s working and understood that only the

text content in the page has to be parsed but not content in other html tags like <meta>, <style> and the content inside it. Only the text visible on the browser needs to be parsed.

KEYWORDS

Search engine, Information retrieval, Preprocessor, crawler, Vector space model.

Weighting Schemes & Similarity Measures

1.1 Similarity Measure

We used cosine similarity in this project to rank relevant pages. Each document vector length was calculated and the user entered query vector length was also calculated. Cosine similarity takes into consideration the length of both the document and the query. The tokens that are not present in the query or the document do not affect the cosine similarity at all. Usually the query vectors are extremely sparse, making it easier to calculate the similarity value.

1.2 Alternative Similarity Measures

Other similarity measures that we could use instead of cosine similarity are inner product, dice coefficient and jaccard coefficient. Cosine similarity is a modification of inner product and does not give good results when used. Jaccard coefficient is better suited as a dissimilarity measure and hence I thought it wouldn't be feasible to use this as a similarity measure.

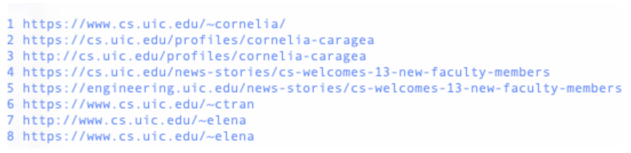
1.3 Weighting scheme

In this project, TF-IDF was used as the weighting scheme for the words in webpages. The code for this is initially written for assignment-2 and it is a relatively simple 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.

Evaluation of Queries

This search engine was evaluated by running five custom queries and top 10 webpages were retrieved for each query.

1. Query: Cornelia Caragea



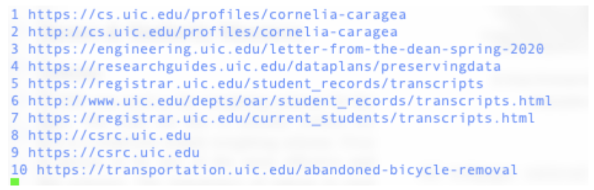
```
1 https://www.cs.uic.edu/~cornelia/
2 https://cs.uic.edu/profiles/cornelia-caragea
3 http://cs.uic.edu/profiles/cornelia-caragea
4 https://cs.uic.edu/news-stories/cs-welcomes-13-new-faculty-members
5 https://engineering.uic.edu/news-stories/cs-welcomes-13-new-faculty-members
6 https://www.cs.uic.edu/~ctran
7 http://www.cs.uic.edu/~elena
8 https://www.cs.uic.edu/~elena
```

Figure 1: **Search results for "Cornelia Caragea"**

Eight webpages were retrieved and all of them are directly related to the query and first three pages are directly related to the professor.

We can say that precision = 1.0 and recall is difficult to calculate since we don't know the total number of relevant URLs for the query in advance.

2. Query: info retrieval

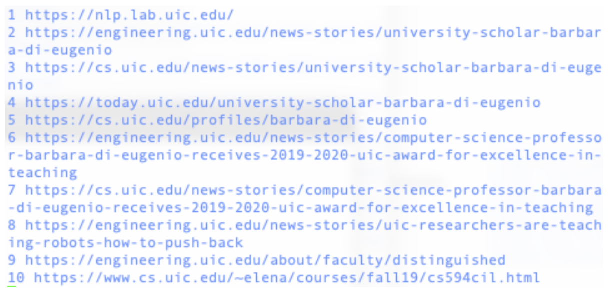


```
1 https://cs.uic.edu/profiles/cornelia-caragea
2 http://cs.uic.edu/profiles/cornelia-caragea
3 https://engineering.uic.edu/letter-from-the-dean-spring-2020
4 https://researchguides.uic.edu/dataplans/preservingdata
5 https://registrar.uic.edu/student_records/transcripts
6 http://www.uic.edu/depts/oar/student_records/transcripts.html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

Figure 1: **Search results for "info retrieval"**

All webpages retrieved are related to the query. Each page retrieved has query terms in them. Hence we can conclude that precision = 1.0

3. Query: NLP

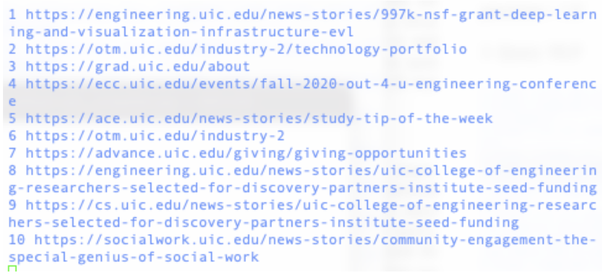


```
1 https://nlp.lab.uic.edu/
2 https://engineering.uic.edu/news-stories/university-scholar-barbar
a-di-eugenio
3 https://cs.uic.edu/news-stories/university-scholar-barbara-di-euge
nio
4 https://today.uic.edu/university-scholar-barbara-di-eugenio
5 https://cs.uic.edu/profiles/barbara-di-eugenio
6 https://engineering.uic.edu/news-stories/computer-science-professo
r-barbara-di-eugenio-receives-2019-2020-uic-award-for-excellence-in-
teaching
7 https://cs.uic.edu/news-stories/computer-science-professor-barbara
-di-eugenio-receives-2019-2020-uic-award-for-excellence-in-teaching
8 https://engineering.uic.edu/news-stories/uic-researchers-are-teach
ing-robots-how-to-push-back
9 https://engineering.uic.edu/about/faculty/distinguished
10 https://www.cs.uic.edu/~elena/courses/fall19/cs594cil.html
```

Figure 1: **Search results for "NLP"**

All webpages retrieved were relevant to NLP. Hence we can say that precision=1.0

4. Query: Deep learning

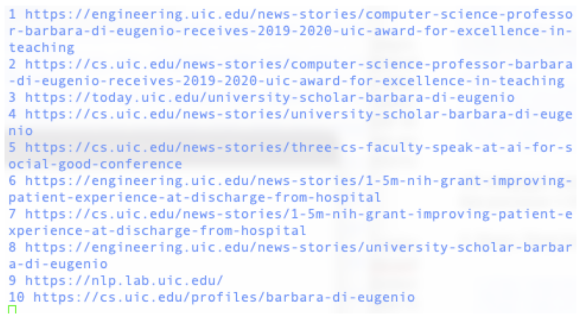


```
1 https://engineering.uic.edu/news-stories/997k-nsf-grant-deep-learn
ing-and-visualization-infrastructure-evl
2 https://otm.uic.edu/industry-2/technology-portfolio
3 https://grad.uic.edu/about
4 https://ecc.uic.edu/events/fall-2020-out-4-u-engineering-conferenc
e
5 https://ace.uic.edu/news-stories/study-tip-of-the-week
6 https://otm.uic.edu/industry-2
7 https://advance.uic.edu/giving/giving-opportunities
8 https://engineering.uic.edu/news-stories/uic-college-of-engineerin
g-researchers-selected-for-discovery-partners-institute-seed-funding
9 https://cs.uic.edu/news-stories/uic-college-of-engineering-researc
hers-selected-for-discovery-partners-institute-seed-funding
10 https://socialwork.uic.edu/news-stories/community-engagement-the-
special-genius-of-social-work
```

Figure 1: **Search results for "Deep Learning"**

All webpages retrieved were relevant to 'Deep learning'. Hence we can say that precision=1.0

5. Query: Barbara de Eugenio



```
1 https://engineering.uic.edu/news-stories/computer-science-professo
r-barbara-di-eugenio-receives-2019-2020-uic-award-for-excellence-in-
teaching
2 https://cs.uic.edu/news-stories/computer-science-professor-barbara
-di-eugenio-receives-2019-2020-uic-award-for-excellence-in-teaching
3 https://today.uic.edu/university-scholar-barbara-di-eugenio
4 https://cs.uic.edu/news-stories/university-scholar-barbara-di-euge
nio
5 https://cs.uic.edu/news-stories/three-cs-faculty-speak-at-ai-for-s
ocial-good-conference
6 https://engineering.uic.edu/news-stories/1-5m-nih-grant-improving-
patient-experience-at-discharge-from-hospital
7 https://cs.uic.edu/news-stories/1-5m-nih-grant-improving-patient-e
xperience-at-discharge-from-hospital
8 https://engineering.uic.edu/news-stories/university-scholar-barbar
a-di-eugenio
9 https://nlp.lab.uic.edu/
10 https://cs.uic.edu/profiles/barbara-di-eugenio
```

Figure 1: **Search results for “Barbara De Eugenio”**

All webpages retrieved were relevant to ‘Barbara de Eugenio’. Hence we can say that precision=1.0

RESULTS

• From the query evaluations, we can see that all relevant pages are fetched and the precision for all query results are good.

• Top three relevant pages are also the top results in the uic domain for any given query.

• While using TF-IDF weighting scheme, words that are most frequent in a document play a major role in determining its topic. From our queries containing words like ‘nlp’, ‘deep learning’ etc, all such words are more frequent only in the documents relevant to the topic and hence term frequency plays a major role in ranking these documents in our case.

• But there are few instances where there are few query results that are not relevant to the context intended by the query. For example in query “information retrieval”, the pages returned by the search don’t all correspond to the context of “CS-494 information retrieval”. Instead pages that contain even one occurrence of the word ‘information retrieval’ are returned. Other than the first url result, all other pages are just pages that have at least one occurrence of the query string.

Related Work

• I went through online resources and few parts in textbook to understand the terminology and the implementation of web crawler.

• While finding the implementation of crawler online, I came across BFS approach. So I used BFS traversal for the crawler to update its pages.

• I compared similarity measures like cosine similarity, jaccard coefficient and their use cases. For this project, I choose cosine similarity over Jaccard coefficient as it seemed feasible to me to calculate the similarity between documents and a query than calculating dissimilarity between query and all documents and then fetching from the bottom.

• I studied the Page rank algorithm and tried to incorporate into this project but found implementing it challenging and stuck. So I decided not to use the Page rank algo in this project.

FUTURE WORK

• While fetching visible content from webpages, spammers can change the color to match background color. Need to add workarounds for this.

• Few extensions like ‘.ppt’, ‘.pdf’ of the irrelevant file formats that were irrelevant and slowing down the crawler. These extension types are consuming lot more time to be downloaded and then parsed. Adding support for these extensions could be an improvement.

• Support to accept and crawl relative URLs need to be added.

• Page rank could be incorporated into this project for better results and performance.

• Owing to the dynamic nature of the webpages on the internet, pulling the data from these pages time to time can be added.

• Better GUI can be added for the front end of the application.