Project-4 Comp 479/6791
Fall 2020

Dept. of Computer Science and Software Engineering
Concordia University

Submitted to: - Dr. Sabine Bergler

Submitted by: Yashika Khurana
Student id: 40094722

Introduction

1. Crawler

In this module I used Scrapy to crawl the “https://www.concordia.ca”. Scrapy is a free and open-source web-crawling framework written in Python. It is easy to implement and can handles the requests asynchronously. The main advantage of using this tool is because it automatically adjusts crawling speed using own Auto-throttling mechanism. It also provides a built-in mechanism for extracting data by using selectors but for this project for scraping I have used BeautifulSoup.

It also have the feature to limit the no. of pages for crawling. The following command can be used if you want to limit the crawling no to 10.

```
“scrapy crawl "jobs" --set CLOSESPIDER_PAGECOUNT=10”.
```

To obey the standard for robot exclusion, we can give the command. While building the index, this command was used.

```
$ crapy crawl --set=ROBOTSTXT_OBEY='True' jobs
```

Where jobs is the name of spider used. ROBOTSTXT_OBEY=‘True' means we want to obey the standard for robot exclusion.

While crawling all the pages are stored in .html format.

All the urls are stored in temp_list.json and all the files have been given the unique id starting from 1. For example 1.html, 2.html, 3.html.

And while running the query on the index, the id’s are decoded into the url. So each id correspond to one url.

For example 1.html———->”https://www.concordia.ca/"

All the html files are stored in the same directory. For the submission purpose only 50 html files were submitted. But for practical purposes 2500 html files were used.

2. Extracting data and Pre-processing

To extract the text from the web pages, I have used BeautifulSoup. From the pages I extracted the ‘title’ and ‘body’. Text to generate the tokens.

After extracting the text, in the loadpages() method following steps were applied:

Step 1: Removal of punctuations and no. from the text. Following punctuations were removed from the text

```
!()[]{};:'"\©,<>./?#$%^|&*_+~-
```

Step 2: nltk.word_tokenize was used to tokenize the text

Step 3: Case folding of the tokens generated from Step 2.

Step 4: Using nltk.corpus stopwrods were removed from the output of Step 3.

Step 5: Then Porter Stemmer have been used to stem the generated tokens from Step 4

Step 6: Generation of tuple containing (token, id) pair.

I didn't removed the no. because I want to work with the search like "covid-19".

3. SPIMI

To build the index, SPIMI has been used. In this module the main focus is to build the blocks which have structure like ["block1": [{'df': 4}{docid1:tf},{docId2:tf),…,"block2": [{docid1:tf},{docId2:tf),…]. Where df stands for document frequency and tf stands for term frequency.

In the function generate_blocks(), this block is used to read the stream of tokens and generates the block based on the following parameters:

- For the implementation purpose block size is taken as 10,000 and then the total no of blocks reduced to 257.
- Then each block after generation is sorted and then save to final list which contains all the blocks (dynamic generation of all blocks till we cover the parameter value or if we reach to the end of the input token stream of token, id pair).

Merging:

Then all the blocks generated above will be considered for the merging to build the SPIMI unranked index. The approach used is to first find the lowest token present in all the blocks. Then that lowest token is compared with all blocks first value, so if it is present the docid,tf will be merged. The final structure of the merging is {"term1":[{'df': 4}{docid1:tf},{docid2:tf},…}. All the postings lists are ordered by term frequency values. The result of the final index is stored in final_index.json

4. Three Ranking criteria

1. BM25

In this module the output of the indexer of the given query is used to rank that documents with the help of BM25 formula which is:

$$RSV_d = \sum_{t \in q} \log \left[\frac{N}{\mathrm{df}_t} \right] \cdot \frac{(k_1 + 1)\mathrm{tf}_{td}}{k_1((1-b) + b \times (L_d / L_{ave})) + \mathrm{tf}_{td}}$$

With the help of this documents are ranked which help us to get the relevant documents first.

To compare the result with different k and b values query “Department" is tested with the following parameter settings:

(“Department”, k=1, b=0.2)

(“Department”, k=1, b=0.5)

(“Department”, k=1, b=0.8)

(“Department”, k=2, b=0.8)

After analyzing the different comparisons of the k and b values, the results were same except different ranking values. Therefore, further values k=1 and b =0.5 is used. First for the individual term rsv is calculated and for the common doc id, these rsv values are added before final ranking of the documents.

2. TF/IDF

To rank the documents according to the tf and idf value we could use simple

Formula 1 =tf/idf

and then we can do the ranking.

But for better measure quality of the ranking I have used the below formula:

TF = (1+log(tf)) and idf=log(N/df), then,
Formula 2 = tf-idf = TF* idf

By comparing the results, Formula 2 gives better results.

3. Different Measure used

The documents are also ranked on the basis of the term frequency given by the SPIMI (extra ranking to compare the better quality measure).

So for each query term its tf values are considered for the ranking.

Please note that for all the measures top 15 documents were only given back to the user.

4. Information need

Please note that to query on the indexer, all the queries have been pre- processed in the same order as I did while creating the tokens i.e. removal of punctuation, case folding, stop words removal, stemming.
well documented sample runs for your queries on the information needs:
(a) which researchers at Concordia worked on COVID 19-related research?
The two variations of the query are applied on three different ranking mechanism. The results of this query are stored in below mentioned files.

Variation 1 = “which researchers at Concordia worked on COVID 19-related research”

Test_Query1_bm25.json - Run on bm25 ranking criteria
Test_Query1_tf-idf.json - Run on better quality measure of tf-idf formula
Test_Query1_diffmeasure.json - Run on extra measure using tf values

Variation 2 = "researchers Concordia COVID 19”

Test_Query1_variationbm25.json - Run on bm25 ranking criteria
Test_Query1_variationtf-idf.json - Run on better quality measure of tf-idf formula
Test_Query1_variationdiffmeasure.json - Run on extra measure using tf values

(b) which departments at Concordia have research in environmental issues, sustainability, energy and water conservation?
The two variations of the query are applied on three different ranking mechanism. The results of this query are stored in below mentioned files.
Variation 1 = "which departments at Concordia have research in environmental issues, sustainability, energy and water conservation”

Test_Query2_bm25.json - Run on bm25 ranking criteria
Test_Query2_tf-idf.json - Run on better quality measure of tf-idf formula
Test_Query2_diffmeasure.json - Run on extra measure using tf values

Variation 2 = "departments Concordia research environmental issues sustainability energy water”

Test_Query2_variationbm25.json - Run on bm25 ranking criteria
Test_Query2_variationtf-idf.json - Run on better quality measure of tf-idf formula
Test_Query2_variationdiffmeasure.json - Run on extra measure using tf values

These sample when used to run with BM25, gives the good result in terms of first it gives the documents which are most relevant and then less relevant and so on, depending upon the rsv value for the particular docid.

5. Challenge Queries

In this module, we were given three queries
1. water management sustainability Concordia
2. Concordia COVID-19 faculty
3. SARS-CoV Concordia faculty

To run these queries on three different ranking criteria, all the queries have been pre-processed and stored in the file Challenge_Query(1/2/3).json

6. Experience

Different behaviour of ranking scheme: In this project I have worked with three different mechanism which helps me to get more clear understanding of the comparison of different ranking measures. In general bm25 performed better in terms of ranking. In general variation 2 performed better in terms of ranking because it contains concrete relevant terms only.

Ranking:To rank the documents based on only tf values does not give good results because for example, words which are the most common words such as “University” will have very high values, giving those words a very high importance. But using these words to compute the relevance produces bad results. Finding the importance of the word across all the documents and normalizing using that value represents the documents much better. So that’s the main reason of using better quality measure.

Top-15 return functionality: To return the top 15 documents for the given query to focus only on the top 15 most important document according to the measure used. This helps in limiting the no. as well as returning the important document instead of returning whole list. So for the user important documents matters the most. Top 15 criteria helped to check the relevancy of the document too.

Crawling and scraping of web projects: Finding the crawler was the major taking time. As we have to search for the crawler which can take the limit as well as it is should be configurable to work with more features. I have tried both Scrapy and Spicy and I found Scrapy better because of the features mentioned in the introduction.

Scarping was easier then crawling because of the learning curve. BeautifulSoup gives the easy handling methods to extract the data of the web pages.

7. Program structure

The program structure is divided into 2 file, i.e. ‘jobs.py’ which is crawler and scraper.py’.

To run the project - **Please simply run the “scraper**.py**” python file** which calls all the methods required to run the project. All the results are saved in the file as .json format. While running the code merging take time, please give some time to do the merging.

All the test queries and challenge queries are kept in different json file instead creating single annotated file for better readability.