COMP 479 - Project 4

François LaBerge

December 11, 2020

1 Different Behaviours of Ranking Schemes

See the separate results file to view the full URLs. We use the naming convetion `informationNeed_query_rankingAlgorith`, for example the resukts for the first query of this document are int file `covid_Concordia COVID-19 faculty_tfidf.txt`.

1.1 Information Need: "which researchers at Concordia worked on COVID 19-related research?"

1.1.1 Query: "Concordia COVID-19 faculty"

The results of this query are not very efficient at retrieving relevant documents for the information need. It also highlights one of the problem encountered using term frequency in our ranking algorithms. Since "Concordia" and "faculty" appear very often in the top ranking pages of these queries they overpower the more relevant term of the query "Covid-19". For example, the first results for tf-idf contains "Concordia" 270 times and the one of bm25 counts 157. Although, we've chosen a value of 0.5 for the k1 parameter of bm25 it does not seem to be enough for such frequent terms.

1.1.2 Query: "SARS-CoV Concordia faculty"

For this query we get the same problems as the previous query since the problems terms, "Concordia" and "faculty" are still there.

1.1.3 Query: "Covid-19 research"

This query returns results that are much more relevant to the information need than the previous two queries since the results contain information about COVID-19. However, our results are not as good as the Concordia website's search engine. For this query, the two ranking algorithms return very similar results.

1.2 Information Need: "which departments at Concordia have research in environmental issues, sustainability, energy and water conservation?"

1.2.1 Query: "water management sustainability Concordia"

For this query, the BM25 ranking scheme returns results that are much better than the tf-idf results. The BM25 documents are pretty relevant while the tf-idf documents are almost completely irrelevant.

1.2.2 Query: "environmental issues sustainability energy water conservation"

We get a similar situation here compared to the previous query; BM25 returns much better results. We can probably attribute this difference in perfomance by the two tuning paramters of BM25. The `k1` parameter reduces relevancy of terms that are too frequent such as "Concordia". The `b` paramters reduces relevancy of very long documents, such as the top ranking documents of the tf-idf results.

2 Issues with the tf ranked postings list

We did not encounter any complicated issues while implementing the tf-ranked postings list. To keep the lists ranked during the indexing process we use python "heapq" library to create a priority queue that will keep our list sorted throughout the indexing process. The algorithm on page 45 of the 7th slide set was implemented to handle the ranking with a limited number of items. Here is a code snippet showing the implementation

```
def insert_posting(self, posting: Posting) -> None:
        if len(self._queue) < self._max_items:
                heapq.heappush(self._queue, (-posting.term_frequency, posting.doc_id, posting))
        elif -posting.term_frequency < self._queue[0][0]:
                heapq.heappop(self._queue)
                heapq.heappush(self._queue, (-posting.term_frequency, posting.doc_id, posting))
```

Note that the negative signs are necessary to turn the min heap into a max heap. For querying, the sorted postings list is retrieved using heapq's "nsmallest" method.

```
def get_postings(self) -> List[Posting]:
        return [element[2] for element in heapq.nsmallest(self._max_items, self._queue)]
```

3 Top 15 return functionality

Returning the top K results was very similar to ranking the postings list by TF. Again we used the algorithm on page 45 of the 7th slide set to maintain a bounded list only keeping the top k results. The implementation can be seen in the following code snippet.

```
def insert_posting(self, rank: float, doc_id: str):
        if len(self._queue) < self._max_items:
                heapq.heappush(self._queue, (rank, doc_id))
        elif rank > self._queue[0][0]:
                heapq.heappop(self._queue)
                heapq.heappush(self._queue, (rank, doc_id))
```

However, we did encounter issues when deciding which document to rank. As the amount of the documents in the inverted index is large, ranking every document in corpus would be rather resource intensive. Therefore, we decided to take some shortcuts. The functions used to perform ranking are the following,

tf-idf:

$$score(q,d) = \sum_{t \in q} tf_{t,d} \cdot log \frac{N}{df_t}$$

bm25:

$$score(q,d) = \sum_{t \in q} log \frac{N}{df_t} \cdot \frac{(k_1+1)tf_{t,d}}{k_1((1-b)+b\frac{L_d}{L_{ave}}+tf_{t,d})}$$

As one can see, both scoring methods computes the score of a document based on a sum over all query terms in the document. Therefore, if a document does not contain any terms in the query its score would be 0. For this reason, we have chosen to only ranked the documents that contain one or more of the query terms. After acquiring this list of document, they can be easily ranked using one of the two formulas seen above. The python implementation for this algorithm can be seen in the following snippet.

```
def rank(self, query: List[Token], inverted_index: InvertedIndex) -> List[Tuple[float, str]]:
        index: Index = Index()
        for query_token in query:
                postings = inverted_index.get_postings(query_token.term)
                df = postings.get_document_frequency()
                for posting in postings:
                        index.insert(posting.doc_id, query_token.term, posting.term_frequency, df)

        nb_documents = inverted_index.document_list.get_nb_documents()
        average_document_length = inverted_index.document_list.get_average_document_length()
        ranked_documents: RankedDocumentsQueue = RankedDocumentsQueue(self._nb_documents_to_return)
        for doc_id in index:
                document_length = inverted_index.document_list.get_document(doc_id).size
                rank = 0
                for term in index[doc_id]:
                        rank += self.score(term.df, term.tf, nb_documents, document_length, average_docu
                ranked_documents.insert_document(rank, doc_id)

        return ranked_documents.get_ranked_posting()
```

With the postings list limited to 50 posting, this makes the ranking process depend almost uniquely on the length of the query. These techniques reduce the time to process a query something much more acceptable.

4 Crawling

We used two libraries to handle web crawling and scraping. We opted for "Scrapy" as our web crawling framework and "Beatifulsoup" as our tool for scraping. Scrapy was a good choice for webscraping as it is usually used to scrape a single domain, which is our case in this project.

The crawling and scraping process is rather easy with Scrapy. By defining the `start_urls` variable of our Scrapy spider, the spider will start its crawling at with those URLs. In our case, we only specified "https://www.concordia.ca/" in `start_urls`. Behind the scenes, Scrapy will make the web request to the URL and pass the response body to the `parse(self, response, **kwargs)` method of our spider. From there we use Beatifulsoup to parse the content of that web request, allowing us to easily extract the text of the page and the URLs from the page to continue crawling. Beatifulsoup allows us to easily get the URLs by grabbing the `a` attribute of all `href` tags. In our case, since we did not want to crawl beyond the "wwww.concordia.ca" domain, we only grabbed link that began with "/", which indicates a link to the current domain. Those URLs are then yield by the `parse(self, response, **kwargs)` method for Scrapy to handle the web requests.

Adding the robot exclusion standard to our spider was very easy since Scrapy has a built-in feature for that. Although the option is disabled by default, we can force a Scrapy spider to obey robots.txt by adding the key-value pair `'ROBOTSTXT_OBEY': True,` to the `custom_settings` class variable of our spider.

We did encounter some limitations of the library we chose. For instance, in an attempt to increase the throughput of the spider, we made an attempt to use multithreading to allow the processing of multiple web pages at once. However, after some manual testing, we found out that our implementation did not improve the performance of our spider. After some research, we discovered that Scrapy was singly threaded. Therefore, a possible hypothesis to explain this phenomenon is that the time complexity associated with making the web requests is magnitudes higher than the time complexity of parsing the documents in adding them to the inverted index. This would indicate that the process spend most of its time making web requests on a single thread and all advantages of multi-threading are rendering insignificant by this difference in time complexity. Regardless, we have changed the Spider settings `'CONCURRENT_ITEMS'`, `'CONCURRENT_REQUESTS'`, and `'CONCURRENT_REQUESTS_PER_DOMAIN'` to a value of 1, because it seems to provide the least fluctuations in CPU usage.

We also encountered some problems with our spider when encountering a specific type of webpage. In a few instances, our spider requested pages that were not HTML document, but ".doc". Those documents proved to be extremely long to be parsed by Beatifulsoup. Those documents reduced the efficiency of our we should consider detecting and ignoring those pages altogether.