# COMP 479 - Project 4

François LaBerge

December 11, 2020

## 1 Different Behaviours of Ranking Schemes

## 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.

```python
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.

- parallelization - robots.txt - non html documents