Report.

COMP 479 Deliverable 3

November 7th 2022

Christian Pangia-Henneveld

40034040

Explanation of Work

One of the main issues with my initial code was the project got too complicated and eventually searching for functions between my poorly scripts became impossible. A couple days before the due date, I decided to redo my project and separate it into 3 main classes that organized my code in a more readable way. I have a Query, BM25 and an Indexers class. These contain the logic for the “and” and “or” queries, all the calculations to create a ranked list for the indexer and the creation of the indexers themselves respectively. The downfall of my other projects have now become a strong suit of my third project.

Subproject 1 was pretty straightforward. We were tasked with creating an indexer inspired by SPIMI creation from our indexer in project2. Obviously changes have to be made to the indexer which were done first. As described we will directly output our term-DocID pairs into a dictionary instead of to a list which will then be converted into a dictionary. The only part here that doesn't necessarily follow how SPIMI is described is the merging of blocks. Since we are working with such small numbers we never have to separate our dictionaries into separate blocks

which are merged after. We also had to put the 10000 term-docID limit to give us a good basis for comparison in terms of times. The picture below demonstrates my results from the timing of both methods.As expected SPIMI runs faster than the indexer in P2. In my case more than 3 times faster but working over a whole corpus 3x longer becomes a lot longer. My SPIMI takes 30 seconds for the whole corpus so if the project 2 indexer can keep up it would take roughly 1 and a half minutes which we can definitely agree is much slower.

```
Enter Path to reutersC:\Users\cpang\PycharmProjects\Comp479_P3\Reuters\
--- 0.0034801000001607463 seconds ---

10 000 Term-DocID pairs for the P2 indexer
Enter Path to reutersC:\Users\cpang\PycharmProjects\Comp479_P3\Reuters\
--- 0.011485299999773018 seconds ---
```

Subproject 2 had us create an altered version of the SPIMI inspired dictionary we made in subproject 1. I use the word altered lightly because we don't actually need to alter it however we need certain things for the BM25 calculation which we can only get while creating the indexer which means another indexer would have to be created. I chose the route of doing two different functions for the indexers. One of which is just a quick SPIMI and doesn't take into account all this extra stuff needed and one which I call naive_indexer in my Indexers class which will calculate all these metrics for me while I create the inverted index. Realizing we needed these things after doing SPIMI actually caused more of a headache because despite having the formula for the equation I didn't have all the pieces

and it looked like I should have. The idf was simple, i kept track of all document lengths as well as the total amount of terms in the corpus while i was creating the indexer. I put the document lengths into a list which since i process docs sequentially line up exactly with the id of the document. I can then create a dictionary with the term as the key and the idf value for each word in the corpus. This one-time cost to create the dictionary allows an O(1) lookup when using it within the greater calculation of BM25 ranking and speeds up the whole process itself. The second part we needed to calculate was TF_{td} . TF_{td} wasn't difficult; it was just a matter of creating the postings list originally with duplicates in it and then calculating the frequency of that number in each posting list before removing them. To get the length of any document as I previously mentioned you just have to check the list at position docID-1 which returns the size of the document(amount of tokens in title and body) . The average was computed once by taking the total amount of terms in the document and dividing by the number of documents itself. With all these individual things calculated we are able to vary our parameters for K1 and B and then see our ranked results. I usually stuck with B=1 which meant term weight was fully scaled by my document length.This gave me rankings for my answers which were very similar to the list of documents given by the "and" query search on the same sentence. The guide for Okapi BM25[1] on Elastic suggested a k of 1.2 which I ended up sticking with.

[1] https://www.elastic.co/guide/en/elasticsearch/guide/current/pluggable-similarites.html

Another part of subproject 2 was the creation of functions which allowed multi-term queries with either “and” or “or” between each term. The only caveat was “or” had to be placed in a specific order. For the most part both of these functions were simple and we used our indexer and repeatedly searched for all the words in the query. Depending on which operation you were doing we take the union or intersection of the postings list for the terms. I order the lists in increasing size for the queries as I know intersecting the smallest list with the second smallest list and repeating it in this fashion is the quickest and most optimized way of intersection since we know the list can only be as large as the smallest list for “and”. The “or” query can still be optimized depending on how you approach it. I did have another oversight here, I read the requirements for having a descending order of frequency(the document with the most terms is first) but I didn't implement it. I caught this error when my “and” query document ID’s weren’t directly matching the first X document ID’s for my “or” query. I figured out how to do this initially one way with regular looping and such but a functional way using list comprehension I found on stack overflow cited in a comment in my code gives me the same results and slightly quicker runtime so I chose to go with this one.

Documented Sample runs

```
query = input("What is your query?")
Query = Query(indexer)

Query.query_operation(query, "and")
Query.query_operation(query, "or")

with open(query + '.txt', 'w') as convert_file:
    convert_file.write("And\n")
    convert_file.writelines(str(Query.result_and))
    convert_file.write("\nOr\n")
    convert_file.writelines(str(Query.result_or))
```

All sample queries tested will be given in files under the directory Ranked and Unranked testing. I chose to replicate the query given to us in project 2 and search for Samjens. I anticipated having 4 results as our dictionary is uncompressed which is exactly what I received. I also ran the word cooper and received 0 results which means our indexers are working similarly.Testing all queries given at the bottom of the assignment led me to my own test queries of adding on words to them to see the behavior of all my functions. I would see how the “and” list goes down by adding words as well as how adding words changes the document's ranks ever so slightly if you are using words that might commonly appear together. One thing to note is using queries with stopwords in the list for example the first one given to us, might results in large postings lists I made the choice to remove stopwords from the queries to limit the amount of articles that would show up and I do this in the ranked indexing as well to avoid documents

having negative values. Finally the word “Democrats’ ” was giving me issues for whatever reason it would give me a very strange posting list with what seemed like 100s of documents when pulling the results despite me checking the dictionary in the file being only 2 documents. I had to alter that query to Democrats without the apostrophe to run it smoothly. To troubleshoot it I printed it before and after entering the calculation function which does not change the dictionary and it gave different results so I chose to move on from this bug and focused on the rest of my project. Given more time I would like to see how the ranking of the Samjen documents differ with B and K1. The article with very few words and 2 occurrences rank slightly higher than the one with 11 and much more words. I wonder if b and k1 would have a significant effect on the ranking of these two docs. Regardless these are by far the most relevant out of the 4 that contain the word.

Finally, to reiterate all parameters for my calculations: my number of documents were 21578, total length of documents was 3128810 giving me an average document length of 145.3814533320975. BM25 used k1=1.2 and B=1.