COMP472 – Project 3 Report

Rhina Kim – 40130779

November 8, 2022

# Overview

The purpose of this project is to compile and compute execution time for naïve indexer and SPIMI indexer constructions, and to retrieve and analyze BM25 ranked results and Boolean unranked results based on given queries defined. Given the Reuter’s Corpus “Reuters-21578”, both naïve indexer and SPIMI indexer extract the raw text of each article from the corpus, tokenize the text for all articles, and compose an inverted index, which is then used for analysis of ranked and unranked retrieval.

The first part of this project focuses on computing the execution time for naïve indexer and SPIMI indexer constructions. Naïve indexer includes sorting of *term-docID* pairs before constructing an index, whereas SPIMI indexer is composed by directly appending the *term-docID* pairs to the postings list. Since SPIMI indexer uses HashMap to store term-docID pairs into its inverted index, the execution time for sorting after index construction becomes faster than that of naïve indexer.

The second part of this project focuses on constructing ranked and unranked search engines: BM25 and Boolean search engines, respectively. For BM25, given the custom queries created by me and sample queries given by professor, this report showcases the ranked results of retrieved documents and analyzes the influence on tuning parameters (*b* and *k1*) on result scores. For Boolean retrieval, custom queries and sample queries are used again to observe the result documents retrieved based on intersection of the given queries (AND) and union of the given queries (OR).

The preprocessing (such as applying stemming, lowercasing, etc.) was omitted for this project since it was not the necessary topic for this project. However, removing stop words are implemented when input queries are processed and tokenized to retrieve refined results.

The results for these pipeline steps are specified in the DEMO file.

# Technologies

**Language**

* Python 3.8

**Interpreter**

* Pypy3

**Python packages**

* Beautiful Soup 4.9
* NLTK 3.7
  + *nltk.word\_tokenize*
  + *nltk.corpus.stopwords*
* *math.log()*
* *collections.Counter()*

Python 3.8 is used as programming language for this project since it supports natural language processing with Python’s NLTK package. Moreover, pypy3 is used as a replacement for original Python 3 interpreter since its runtime interpreter is faster. As this project requires iteration of massive number of large files, using pypy3 as an alternative interpreter seemed to be an optimal choice.

NLTK is brought to this project to tokenize, and to remove stop words from the corpus and input queries. Beautiful Soup is given for extracting the text data from *.sgm* files which is composed of markup-languages like HTML and XML.

# Program Design and Result Analysis

## **Subproject I**

Before constructing both Naïve and SPIMI indexer, the program first reads corpus from Reuter’s collection, extract its raw texts, and returns list of (*term, docID*) pairs which will be used as parameter for Naïve Indexer and SPIMI Indexer construction.

sgm\_files = filter\_files(DIRECTORY)

documents = extract\_documents\_from\_corpus(sgm\_files)

term\_docID\_pairs = tokenize(documents)

test\_corpus = make\_test\_corpus(term\_docID\_pairs, max\_terms=10000)

Since we only need 10K amount of (*term, docID*) pairs for this subproject section, the function “*make\_test\_corpus*()” slices off all (*term, docID*) pairs retrieved into 10K.

def make\_test\_corpus(term\_docId\_pairs, max\_terms=0):

*# [DEMO] Use only 10K terms for testing purpose*

if max\_terms:

print("Reduced the number of terms in test corpus to " + str(max\_terms))

return term\_docId\_pairs[:max\_terms]

else:

return term\_docId\_pairs

**Important**: All the punctuations are removed during the tokenization process!!

**Naïve Indexer:**

def naive\_indexer(test\_corpus, remove\_duplicate=True):

startTime = time.time() *# Start the time*

*# Remove duplicates*

if remove\_duplicate:

test\_corpus = remove\_list\_duplicates(test\_corpus)

*# Sort (based on term and docID in ascending order)*

test\_corpus.sort(key=lambda posting: (posting[0], posting[1]))

*# make inverted index*

index = create\_inverted\_index(test\_corpus)

endTime = time.time()

elapsedTime = endTime - startTime

print("==> Execution Time: " + str(elapsedTime))

return index, elapsedTime

Constructing Naïve indexer is composed of following steps:

1. Remove duplicated (*term, documentID*) pairs
2. Sort the list of (*term, documentID*) pairs
3. Create inverted index from the list

**SPIMI Indexer:**

def spimi\_indexer(test\_corpus, remove\_duplicate=True):

startTime = time.time() *# Start the time*

*# Remove duplicates*

if remove\_duplicate:

test\_corpus = remove\_index\_duplicates(test\_corpus)

*# make inverted index*

index = create\_inverted\_index(test\_corpus)

*# Sort (based on the and docID in ascending order)*

index = dict(sorted(index.items(), key=lambda x:(x[0], x[1].sort(key = lambda y: y))))

endTime = time.time()

elapsedTime = endTime - startTime

print("==> Execution Time: " + str(elapsedTime))

return index, elapsedTime

Constructing SPIMI indexer is composed of following steps:

1. Remove duplicated (*term, documentID*) pairs
2. Create inverted index from the list
3. Sort the inverted index composed of hash table

The main difference between Naïve indexer and SPIMI indexer relies on the fact that whether sorting comes before creating inverted index or not. For the Naïve indexer, the sorting occurs before constructing inverted index, where list of (*term, documentID*) pairs are sorted. Since sorting of lists with enormous element takes more time than sorting hash table, the execution time for Naïve indexer is slower than that of SPIMI indexer.

The result execution time was:

* **10000 *term-docID* pairs** – with no duplicate pairs (from there 4008 *term-docID* pairs were removed):
  + Using **Naïve** indexer: **0.00555 seconds**
  + Using **SPIMI** indexer: **0.00457 seconds**
* **10000 *term-docID* pairs** – with duplicates:
  + Using **Naïve** indexer: **0.00735 seconds**
  + Using **SPIMI** indexer: **0.00376 seconds**
* **2597951 *term-docID* pairs** from entire corpus – with no duplicate pairs (from there 986711 *term-docID* pairs were removed):
  + Using **Naïve** indexer: **5.15272seconds**
  + Using **SPIMI** indexer: **1.62113 seconds**
* **2597951 *term-docID* pairs** from entire corpus – with duplicates
  + Using **Naïve** indexer: **3.70321 seconds**
  + Using **SPIMI** indexer: **1.23957 seconds**

In any situations, we can conclude that the execution time for SPIMI indexer is faster than that of Naïve indexer, mostly due to sorting. The above result is specified in “*time\_analysis.txt*” under *output\_test/subproject(1-A)/* and *output\_test/subproject(1-B)/* directories inside the Deliverables.

For **Subproject I – (b)**, the compiled inverted index is shown under *output\_test/subproject(1-A)/inverted\_index/* and *output\_test/subproject(1-B)/inverted\_index/* directories which stores 4 cases:

* using Naïve indexer allowing duplicated *term-docID pairs*
* using Naïve indexer without duplicates
* using SPIMI indexer allowing duplicated *term-docID pairs*
* and using SPIMI indexer without duplicates.

## **Subproject II**

For this subproject, ranked BM25 and unranked Boolean search engines were implemented, using SPIMI indexer returned from the Subproject I. SPIMI indexer was used since it had faster index construction time. The index constructed for this subproject section did not allow for any duplicated postings for the purpose of having better result score.

Five custom and sample queries were used for this project, which was:

sample\_queries1 = "America"

sample\_queries2 = "population"

sample\_queries3 = "South Korea and Japan"

sample\_queries4 = "Democrats' welfare and healthcare reform policies"

sample\_queries5 = "Drug company bankruptcies"

sample\_queries6 = "George Bush"

where *sample\_queries* 1 to 3 are custom queries that I have created for this experiment, and *sample\_queries* 4 to 6 are the queries given by our professor.

The tuning parameter used for this experiment was:

**Case 1: k1=0, b=1**

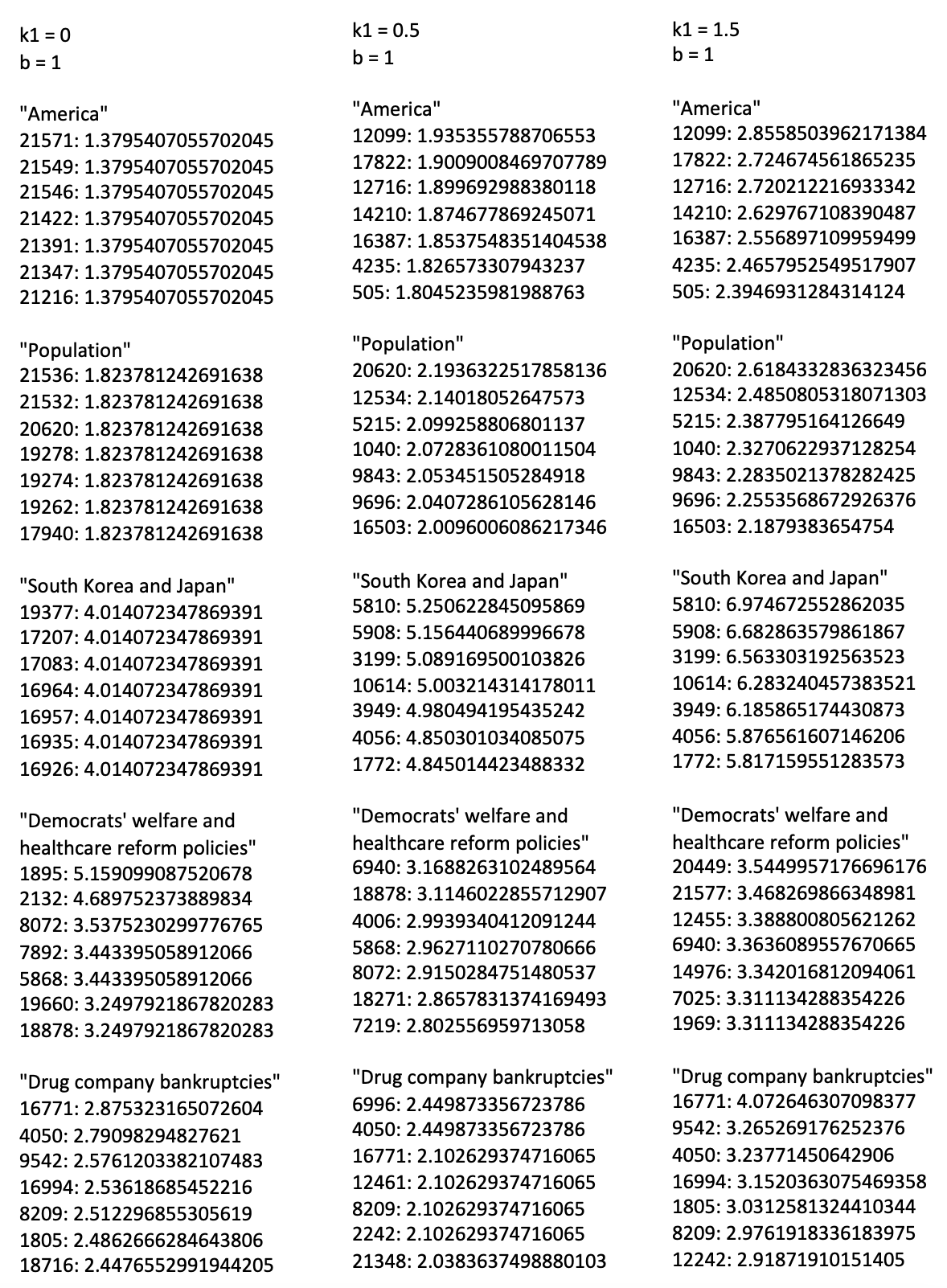
**Case 2: k1=0.5, b=1**

**Case 3: k1=1.5, b=1**

## **BM25**

With change in k1 values, change in ranking of document score was apparent. For example, the single query term like “America” or “population”, rank of the document was identical. However, when k1=0, ranking of the document changed drastically. I assume this is because k1=0 does not consider term frequencies inside document whereas setting k1 >= 0 does. Result with k1 = 0 was different from results with k1 >= 0 because it did not consider the frequency of terms inside one document. k1 >= 0 consider the fact that how many extra times the term is occurring in the document, which adds extra score to the RVSd result, thus bringing different document ranking.

The result score for queries with multiple terms like “Drug companies’ bankruptcies” varied a lot more since it sums up frequency of each token among every documents.



Input queries are tokenized, punctuation and stop words for input queries are removed before passing them to BM25.

*# Tokenize and remove stop words from given queries*

def query\_process(input\_query):

*# Tokenize*

tokens = S2\_get\_tokens\_list(input\_query)

*# Remove stopwords from queries*

stop\_words = list(set(stopwords.words('english')))

tokens[:] = [token for token in tokens if token not in stop\_words]

return tokens

The output of BM25 scores for each different queries and tuning parameters are saved as:

“*ranked\_query(****query#****)-k1****(#k1val****)-b****(#bval****).txt*” under the directory *output\_test/subproject(2-Ranked)/ranked\_results/*

## **Boolean Unranked Retrieval**

Using same indexer (SPIMI indexer) and same list of custom and sample queries specified above, Boolean unranked search engine was implemented.

When input queries are tokenized, the program search for postings lists for each query token. The returned postings list is appended to “*postings\_total = []*” which is a nested list composed of total postings lists returned from all query tokens.

Ex) postings\_total = []

inverted\_index = {“term1”: [34, 56, 677, 2357], “term2”: [34, 453, 565], …}

for token in query\_tokens:

postings\_total.append(inverted\_index[token])

The intersection (AND) of this result postings list is computed as following code:

common\_postings = sorted(list(set.intersection(\*[set(postings) for postings in postings\_total])))

which returns one list “*common\_postings*” composed of all the postings which shares all the query tokens.

The union (OR) of this result postings list is computed as following code:

union\_postings = sorted(list(set.union(\*[set(postings) for postings in postings\_total])))

which returns one list “*union\_postings*” composed of all the postings which shares any of the query tokens.

Since we need ranked results for the Boolean search with union (OR), the program computes the rank based on how many keywords the document contains:

for docID in union\_postings:

freq\_docID = sum(postings.count(docID) for postings in postings\_total)

union\_postings\_ranked[docID] = freq\_docID

which returns dictionary of *docID* and the number of distinct query\_tokens that exists within that *docID*.

If Boolean search engines retrieves 0 intersection or union postings, or if query token of less than one is given which is not enough to compute intersection or union of its postings list, it will leave the result list as blank.

The output of Boolean retrieval results for each different query are saved as:

“*AND\_query(#****query)****.txt*” OR “*OR\_query(#****query)****.txt*” OR

under the directory *output\_test/subproject(2-Unranked)/unranked\_results/*