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.

k1 = 0
b = 1

"America"
21571: 1.3795407055702045
21549: 1.3795407055702045
21546: 1.3795407055702045
21422: 1.3795407055702045
21391: 1.3795407055702045
21347: 1.3795407055702045
21216: 1.3795407055702045

"Population"
21536: 1.823781242691638
21532: 1.823781242691638
20620: 1.823781242691638
19278: 1.823781242691638
19274: 1.823781242691638
19262: 1.823781242691638
17940: 1.823781242691638

"South Korea and Japan"
19377: 4.014072347869391
17207: 4.014072347869391
17083: 4.014072347869391
16964: 4.014072347869391
16957: 4.014072347869391
16935: 4.014072347869391
16926: 4.014072347869391

"Democrats' welfare and healthcare reform policies"
1895: 5.159099087520678
2132: 4.689752373889834
8072: 3.5375230299776765
7892: 3.443395058912066
5868: 3.443395058912066
19660: 3.2497921867820283
18878: 3.2497921867820283

"Drug company bankruptcies"
16771: 2.875323165072604
4050: 2.79098294827621
9542: 2.5761203382107483
16994: 2.53618685452216
8209: 2.512296855305619
1805: 2.4862666284643806
18716: 2.4476552991944205

k1 = 0.5
b = 1

"America"
12099: 1.935355788706553
17822: 1.9009008469707789
12716: 1.899692988380118
14210: 1.874677869245071
16387: 1.8537548351404538
4235: 1.826573307943237
505: 1.8045235981988763

"Population"
20620: 2.1936322517858136
12534: 2.14018052647573
5215: 2.099258806801137
1040: 2.0728361080011504
9843: 2.053451505284918
9696: 2.0407286105628146
16503: 2.0096006086217346

"South Korea and Japan"
5810: 5.250622845095869
5908: 5.156440689996678
3199: 5.089169500103826
10614: 5.003214314178011
3949: 4.980494195435242
4056: 4.850301034085075
1772: 4.845014423488332

"Democrats' welfare and healthcare reform policies"
6940: 3.1688263102489564
18878: 3.1146022855712907
4006: 2.9939340412091244
5868: 2.9627110270780666
8072: 2.9150284751480537
18271: 2.8657831374169493
7219: 2.802556959713058

"Drug company bankruptcies"
6996: 2.449873356723786
4050: 2.449873356723786
16771: 2.102629374716065
12461: 2.102629374716065
8209: 2.102629374716065
2242: 2.102629374716065
21348: 2.0383637498880103

k1 = 1.5
b = 1

"America"
12099: 2.8558503962171384
17822: 2.724674561865235
12716: 2.720212216933342
14210: 2.629767108390487
16387: 2.556897109959499
4235: 2.4657952549517907
505: 2.3946931284314124

"Population"
20620: 2.6184332836323456
12534: 2.4850805318071303
5215: 2.387795164126649
1040: 2.3270622937128254
9843: 2.2835021378282425
9696: 2.2553568672926376
16503: 2.1879383654754

"South Korea and Japan"
5810: 6.974672552862035
5908: 6.682863579861867
3199: 6.563303192563523
10614: 6.283240457383521
3949: 6.185865174430873
4056: 5.876561607146206
1772: 5.817159551283573

"Democrats' welfare and healthcare reform policies"
20449: 3.5449957176696176
21577: 3.468269866348981
12455: 3.388800805621262
6940: 3.3636089557670665
14976: 3.342016812094061
7025: 3.311134288354226
1969: 3.311134288354226

"Drug company bankruptcies"
16771: 4.072646307098377
9542: 3.265269176252376
4050: 3.23771450642906
16994: 3.1520363075469358
1805: 3.0312581324410344
8209: 2.9761918336183975
12242: 2.91871910151405

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

```python
# 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/*