Inverted Index Construction

The file *invidx_cons.cpp* implements the creation of the inverted index. I have implemented the **Single In Memory Pass Indexing** algorithm to create the inverted index. This algorithm can work for arbitrarily large collection sizes, if there is enough disk space despite having limited RAM.

Algorithm 1 SPIMI-INVERT Algorithm

```
output_file ← new file()
dictionary ← new hash()
while free memory available do
    token ← next(token_stream)
    if term(token) ∉ dictionary then
        postings_list ← AddToDictionary(dictionary, term(token))
    else
        postings_list ← GetPostingsList(dictionary, term(token))
    end if
    AddToPostingsList(postings_list, docID(token))
end while
sorted_terms ← SortTerms(dictionary)
WriteBlockToDisk(sorted_terms, dictionary, output_file)
return output_file
```

I parse all the documents available in the collection, to get the token stream and till I have enough memory I keep using the algorithm mentioned above. As soon as memory is close to getting full, I write the postings list to disk and clear the memory.
From the next document onwards, I repeat the algorithm again till we are out of memory and so on.
This way at the end of this algorithm, we would have multiple temporary output files storing partial postings list. These files are stored in a folder named *temp*.

```
temp
├── temp-0
├── temp-1
├── temp-2
└── ....
```

After this a merge algorithm is employed which uses a merge tree to merge files in pairs, till we have a single temporary file representing the postings list and vocabulary of the entire collection.
This single file is then processed, creating the dictionary and the index while employing the appropriate compression algorithm. We also store the mappings from the document IDs to the integer space and also store the norms of the document vectors which will be used during search.

Estimating Memory of Postings List

The postings list is implemented as a

```
std::map<std::string, std::vector<std::pair<int, int>>>
```

This way the tokens are always sorted, To esimate the size of this Data Structure, I use the following Heurestic

$$\textit{size of postings list} = 8 * \textit{Number of (docID, frequency pairs} + 10 * \textit{Number of Tokens}$$

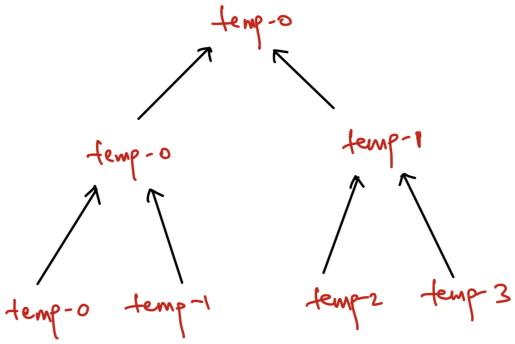

Figure 1: Merge Partial Postings List

This esimates the size of the postings list in bytes, Whenever the size exceeds *500 Mb*, we write the postings list to disk and start afresh from the next document.

Dictionary File

A sample dictionary file looks as follows, each row has a term followed by the number of terms it appears in and the offset in the index file, at the end we also store mappings of document IDs to integers and their precomputed vector norms

```
a 2 2
celeberation 1 6
correctness 1 8
course 1 10
day 2 12
for 1 16
good 1 18
independence 2 20

3  FR940104-1-00001  6.08319
2  FR940104-2-00001  4.86933
1 FT944-17126 4.60416
0  FR940104-0-00001  5.3946
```

Index File

A sample index file in Hex, is visualized as follows

```
01 00 82 81 81 81 80  ......
```

The first 2 bytes (01, 00) represent the type of tokenization and compression used to create this file. In this particular case we have used variable byte compression and the simple tokenizer, Each of the next bytes correspond to the variable byte code
eg. 82 $\Rightarrow$ 10000010 represents 2 in a single byte using variable byte encoding

```
00 00 00 00 00 02  ......
```

This represents a sample index file when we don't use any compression, the same same number 2 is now store using the standard 4 bytes, which wastes considerable space.

Tokenization

The following base delimiters, were used for the simple tokenizer and the pre-tokenization for Byte Pair Encoding

```
std::set<char>{'.' , ' ' , ':', ';', '\"', '\'', '.', '?', '!', ',', '\n', '~', '`', '(', ')', '/',
      '#', '*', '%', '+', '-', '[', ']', '{', '}', '@', '^'};
```

Byte Pair Encoding

Since Training the Byte Pair Tokenizer, is an expensive and time consuming operation, I pretrained my own implemenetation of the BPE Tokenizer, and stored the merges learnt in order which can be loaded later for tokenization. The learned merges look as follows:

```
...
yugoslav ia
at ta
dis count
at l
relig ious
j i
obser vers
grad u
bar ri
gree k
e a
compon ents
occas ion
new ly
...
```

These are stored in the file named *bpe_merges* and are loaded automatically for tokenization if needed.
To speed up tokenization I remember, the splits of a particular token which prevents iterating all merges again and again.

Search

During Search, we first load the dictionary, document ID mappings and the norms of the document vectors in memory.
Then we parse the query file and load all the queries into memory.
We generate the tokens from the query title and content. We maintain a map to store the partial document scores, for every term in the query we load the postings list from the index file using the offsets specified by the dictionary file, we use the tf-idf formulation to update the partial scores of all the documents.
Once all Query Terms are processed we normalize the document scores using the precomputed document vector norms that we stored during index construction.
To get the top K=100 documents, we maintain a min heap which always maintains the best 100 documents, this way we can retreive the top-k documents in $O(Nlogk)$ time.

Heuristics and Optimizations

We treat the Query Tile and Content differently, all terms in the title are considered important for scoring the documents, but for the content we try to pick only relevant terms, this speeds up query processing time significantly.
For all tokens in the query content, we sort them according to the increasing order of the count of documents they appear in.
We only consider the top 50% tokens in the query content according to this ordering.
Also since loading large postings list from the disk can be expensive, if a query term appears in more than 25% of the documents, we skip it during query processing to prevent loading large postings list from disk.

Results

Inverted Index Construction Time

Results are for all the entire training data, consisting of *2296* files containing *527939* documents

Compression	Tokenizer	Create Temporary Postings	Merge Partial Postings	Write Dictionary & Index File	Total Time
No Compression	Simple	784 s	120 s	897 s	1801 s
Variable Byte Encoding	Simple	717 s	69 s	229 s	1015 s
No Compression	BPE	5781 s	71 s	558 s	6410 s
Variable Byte Encoding	BPE	5823 s	36 s	281 s	6140 s

Index and Dictionary Sizes

Compression	Tokenization	Index File	Dictionary File
No Compression	Simple	811 MB	27 MB
Variable Byte Encoding	Simple	223 MB	26 MB
No Compression	BPE	860 MB	15 MB
Variable Byte Encoding	BPE	251 MB	15 MB

Average Query Processing Time

Compression	Tokenization	Average Time
No Compression	Simple	0.39 s
Variable Byte Encoding	Simple	0.69 s
No Compression	BPE	0.36 s
Variable Byte Encoding	BPE	0.65 s

Number of Correct Query Results

Simple Tokenization

On Average, 8 % results were relevant for the queries tested.

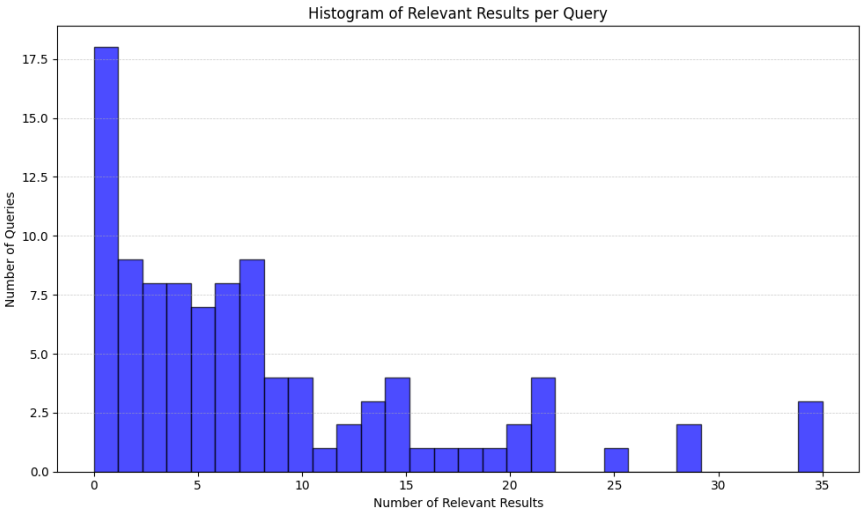

Figure 2: Number of Relevant Results in Top 100 Results

Byte Pair Encoding

Average Correct: 6

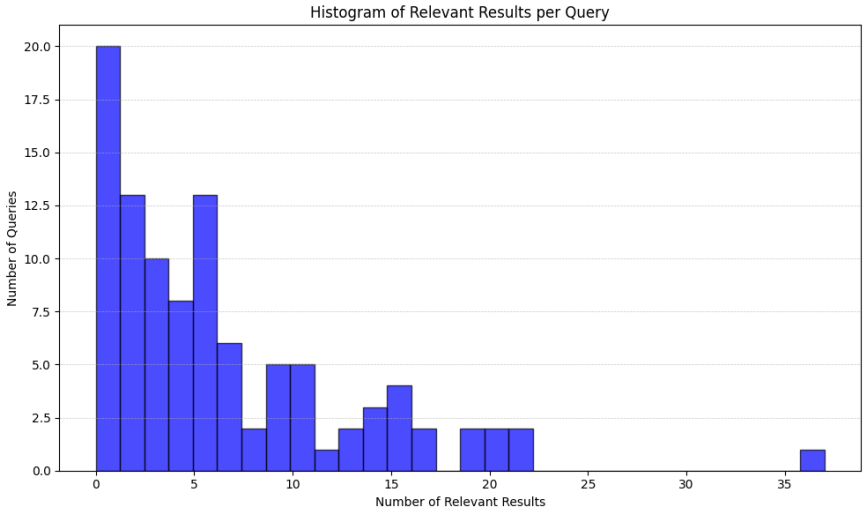

Figure 3: Number of Relevant Results in Top 100 Results

Number of Correct Query Results

Simple Tokenization

I computed Average F1@(100,50,20,10) for each query and then computed the average of these average F1 values Average F1: 0.0904361

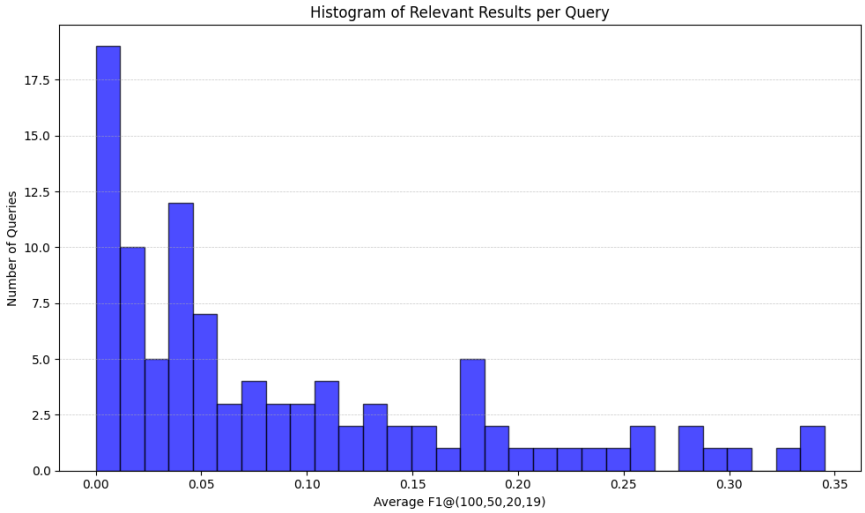

Figure 4: F1@(100,50,20,10)

Byte Pair Encoding

Average F1: 0.066461

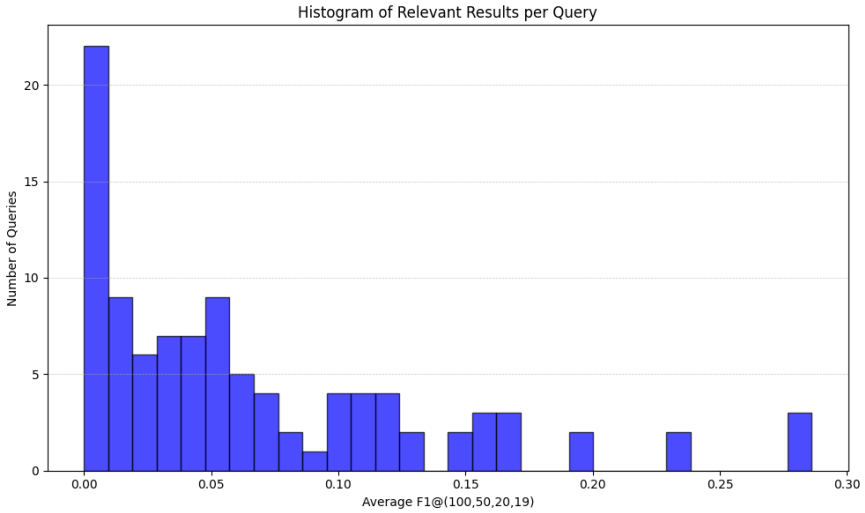

Figure 5: F1@(100,50,20,10)

Note: My Index and Dictionary files can be found at */home/cse/btech/cs1200336/COL764/A1/submission_files*