COL764

Information Retreival And Web Search

Assignment 1

Team Members:

Ayushman Kumar Singh (2021CS51004) cs5211004@cse.iitd.ac.in

Instructor:

Srikanta Bedathur

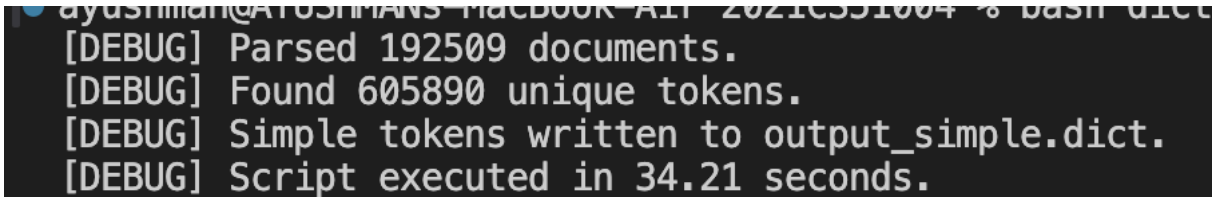
2024/2025 – 1st Semester

Contents

1 Task 1: Encoding and Tokenization

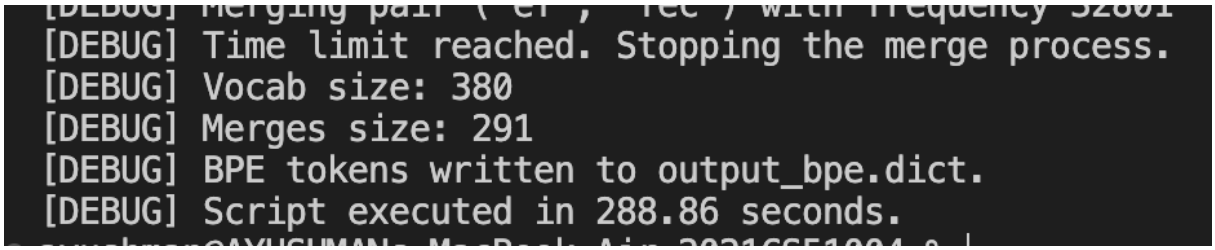
1.1 Normal Encoder! Removal of Non ACSII

- Normal Encoding is done by splitting on the given delimiters and then only considering those **tokens which consists of all ASCII characters**.
- This is one instance of the run where 605890 sized vocabulary was created.



1.2 Byte Pair Encoding! Some Notes and Tunings

- I considered only 50 percent of the documents for training over the BPE. kept ten maximum vocabulary size to 50 thousand and ran it for 287 seconds.
- I was able to achieve around **350 merges in 287 seconds (5 minutes)**. I wrote the vocabulary in the output.dict file. The encoder was run over Simple Tokenizer as base which removed tokens containing non-ascii characters. The vocabulary size was around 400.
- **Multiprocessing for word count calculation. MapReduce?** We exploited the parallelism oppurtunity for word counting across various documents.
- Exploited collections in python to achieve low latency over dictionary operations. Tokenization was carried out by simulating the merges over split of each token created by the Simple Tokenizer.
- **The Runs** I was able to achieve **291 merges** and **vocab size of 380** in **287 seconds**.



1.3 WordPiece Encoding! HuggingFace

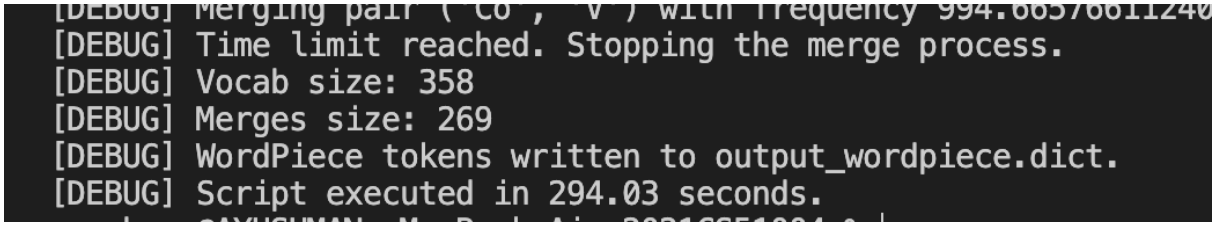
- I considered only 50 percent of the documents for training over the BPE. Kept ten maximum vocabulary size to 50 thousand and ran it for 287 seconds.
- **The formula to calculate the best pair?** The given formula is:
  $pair\_freq/(letter\_freqs[pair[0]] * letter\_freqs[pair[1]])$

at at HuggingFace tutorial page. I tried to use:
$pair\_freq/math.log2((letter\_freqs[pair[0]] * letter\_freqs[pair[1]]))$
or
$pair\_freq/math.sqrt((letter\_freqs[pair[0]] * letter\_freqs[pair[1]]))$

or
$(pair\_freq * pair\_freq)/letter\_freqs[pair[0]] * letter\_freqs[pair[1]]$

and got better result since it gave lower weight to independent occurence of irrelevant symbols. However i have submitted the former one.

- I was able to achieve around **250 merges in 287 seconds (5 minutes)**. I wrote the vocabulary in the output.dict file. The encoder was run over Simple Tokenizer as base which removed tokens containing non-ascii characters. The vocabulary size was around 300.
- **Multiprocessing for word count calculation. MapReduce?** We exploited the parallelism oppurtunity for word counting across various documents.
- Exploited collections in python to achieve low latency over dictionary operations. Tokenization was carried out by simulating the merges over split of each token created by the Simple Tokenizer.
- **The Runs** I was able to achieve **269 merges** and **vocab size of 358** in **287 seconds**.



2 Task 2: Inverted Index Creation!

2.1 Compression Techniques and Contents!

- **Compression techniques:** For **Simple Tokenizer** I **used the compression technique**, because the length of the inverted index is huge, since there are lots of terms. For BPE and WordPiece I did **not use any compression** before writing to the index file.
- **Contents of .dict file** This contains the vocabulary of the tokenizer. For BPE and WordPiece the length of the vocaubulary is around **400**. For Simple tokenizer the length of the vocabulary is around **5 lakhs.**
- **What information I stored in .idx file?**
  - Encoding method used type

  - length of dictionary containing the merges (for BPE and WordPiece, it is non zero)
  - the merge dictionary
  - length of inverted index
  - ¡length of term, term, postings¿ for each of the term. These are compressed in case of Simple Tokenizer.
  - postings included document_id and frequency (term frequencies)(compressed in case of SimpleTokeniser)

- So in all, I passed the merge dictionary (in case of BPE and WordPiece) and term frequencies integrated in the postings list.

2.2 Multiprocessing

- **Exploiting oppurtunities of parellelism! MapReduce?** We could have Tokenized each documents independently as they do not have interdependence. I divided the document chunk in *ncpu* chunks and processed (Tokenized) them independently.

- I merged the inverted_index generated by each chunk as done in **MapReduce Operations**.

2.3 Time Division between learning and Tokenization

- I allocated 280 seconds for training the tokeniser in case of BPE and WordPiece. It generated around 350 merge operations for BPE and WordPiece. The rest of the time was donated to tokenize the document, and then write inverted index to the .idx file.

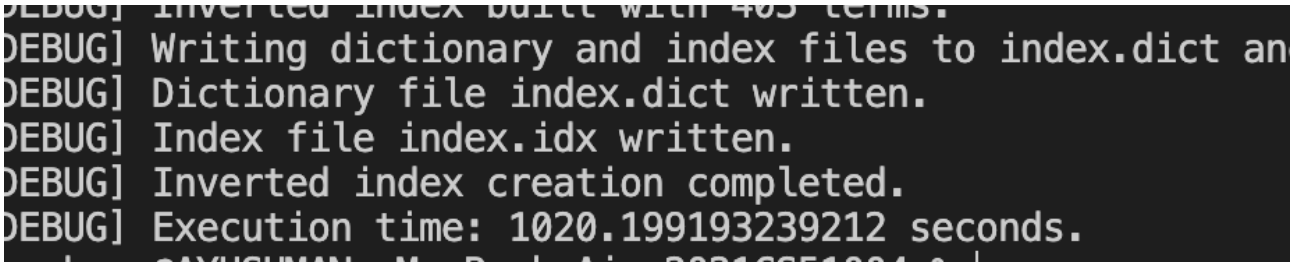
- Writing of inverted index to the .idx file was done in a compressed manner in case of SimpleTokeniser and in a non compressed manner in case of BPE and WordPiece. **Why?** Because compression takes time, while in case of BPE and WordPiece I have to give more time to learning and tokenization.

2.4 Giving more weight to title?

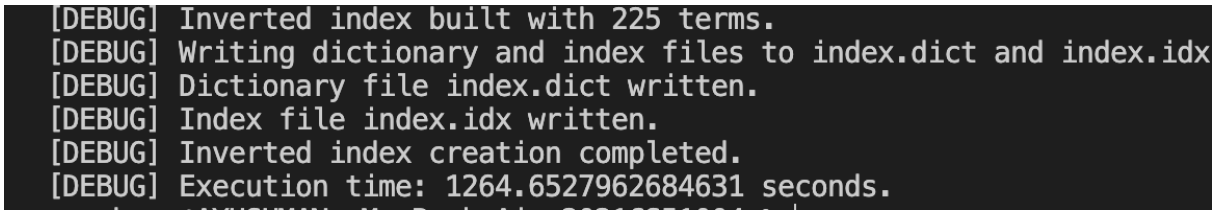
- I repeated the title two times in the corpus. This would increase the term frequency while having no affect on the inverse document frequency. This would lead to better weights for more important terms.

2.5 Runs

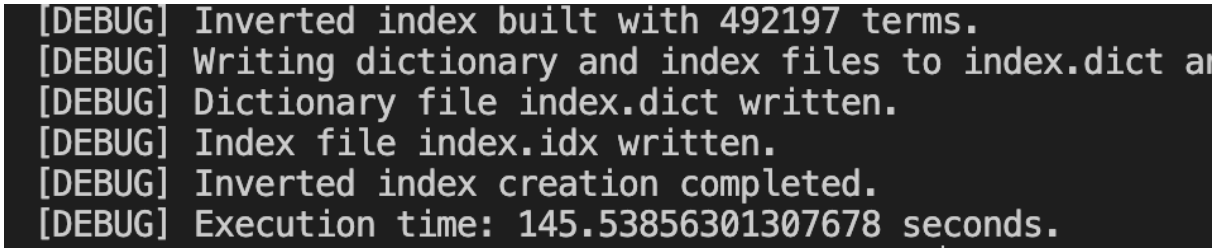
- With BPE tokenization and 8 CPU cores. **Time Taken = 1020 Seconds** with **280 seconds training** time



- With WordPiece tokenization and 8 CPU cores. **Time Taken = 1264 Seconds** with **280 seconds training** time



- With Simple Tokenisation and 8 CPU cores. **Time Taken = 145 seconds** with **1 second of training time**



3 Task 3: TF-IDF Search!

3.1 The "Metrics"

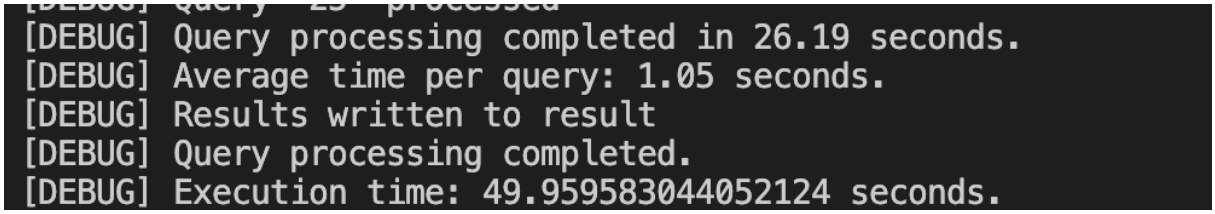
- **Notes: We First Look at the Retrieval Efficiency** What is total query processing time?
  This does not include document norm calculation and loading of index, dictionary files. This also does not include weight calculation of document terms.

  But this does include **query file load, query tokenization, query processing to calculate query term weights, searching for top 100 results and writing the top 100 results to result file.**

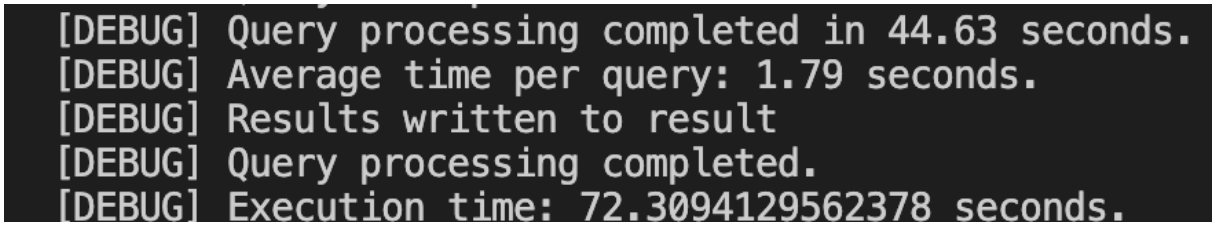
- We generate top 100 best result for each query. The time taken for query file load, query tokenization, query processing to calculate query term weights, searching for top 100 results and writing the top 100 results to result file, is considered as total time to process the 25 queries.

- We divide the above time by total number of queries to get the retrieval efficiencies.

- **Simple Tokenization:** Average Query processing time is **1.05 seconds** and total processing time is **26.19 seconds**. Total time for the program to run is **50 seconds**



- **BPE Tokenization:** Average Query processing time is **1.79 seconds** and total processing time is **44.63 seconds**. Total time for the program to run is **72 seconds**



3.2 Giving more weight to Query title!

- I repeated the title two times. This would increase the term frequency and would lead to better weights for more important terms of the query. This would likely give more rank to documents which contain them.

3.3 Loading the Index and Compression

- For simple tokenizer I load and then decompress the the bytes to get the postings list. For BPE and WordPiece I directly read. Why? Simple Tokenization contains a lots of terms leading to a very high loading time. Compression significantly improved performance in case we have a lot of data to read.

- Why did not I use the compression techniques for BPE and WordPiece? Because that would have caused me to compress it while writing, thus eating up the training time.

3.4 Evaluating Similarity! How to improve the performance?

- I calculated the **document norm (i.e., sum of weights of each term for a doument) before hand** only. For each query I had to calculate the query norm (i.e., sum of weights of query term for a query) which could not be waived off..

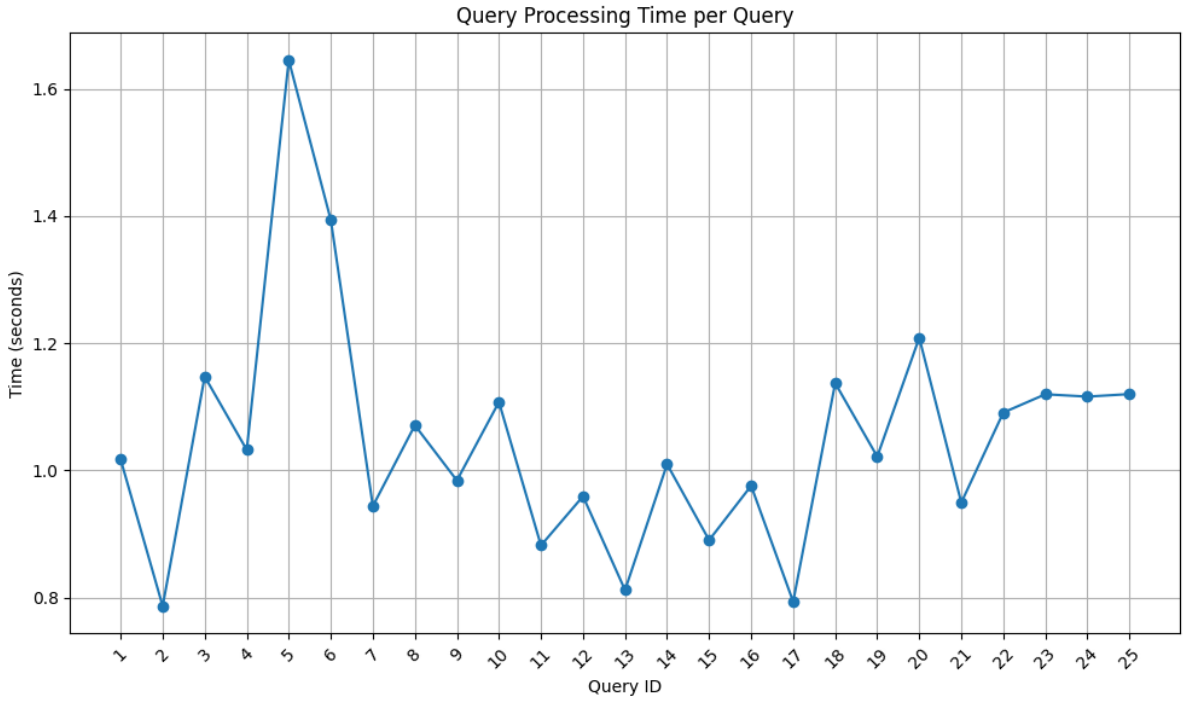
- I calculate the query_term weights for each query. Then loop across all the douments, calculating the cosine similarity with, it. **Note that while taking the dot product, I only consider terms which are present in both the document and the query instead of iterating over all of the vocabulary.** Then divide the whole thing by already calculated document norm and the query norm.

3.5 The "Metrics" Revisited!

We here plot the time taken by different queries over a differnt run.

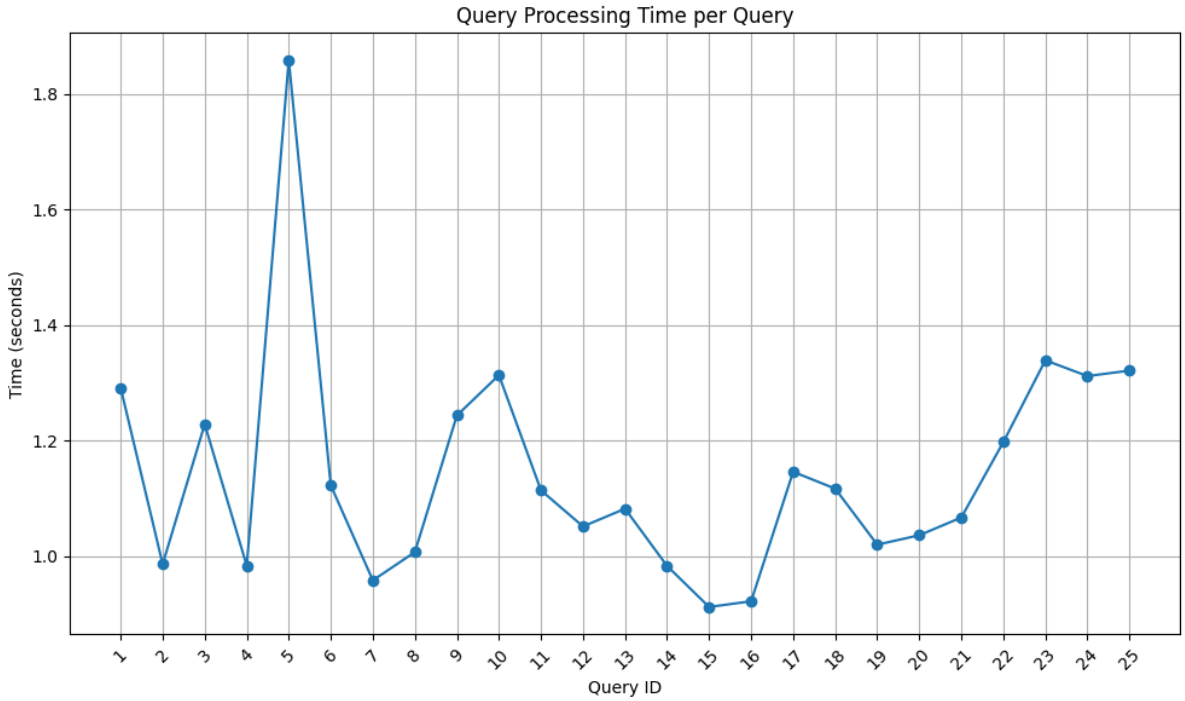
- WordPiece tokenisation:

```
[DEBUG] Query processing completed in 26.21 seconds.
[DEBUG] Average time per query: 1.05 seconds.
[DEBUG] Results written to result
[DEBUG] Query processing completed.
[DEBUG] Execution time: 60.89474296569824 seconds.
```



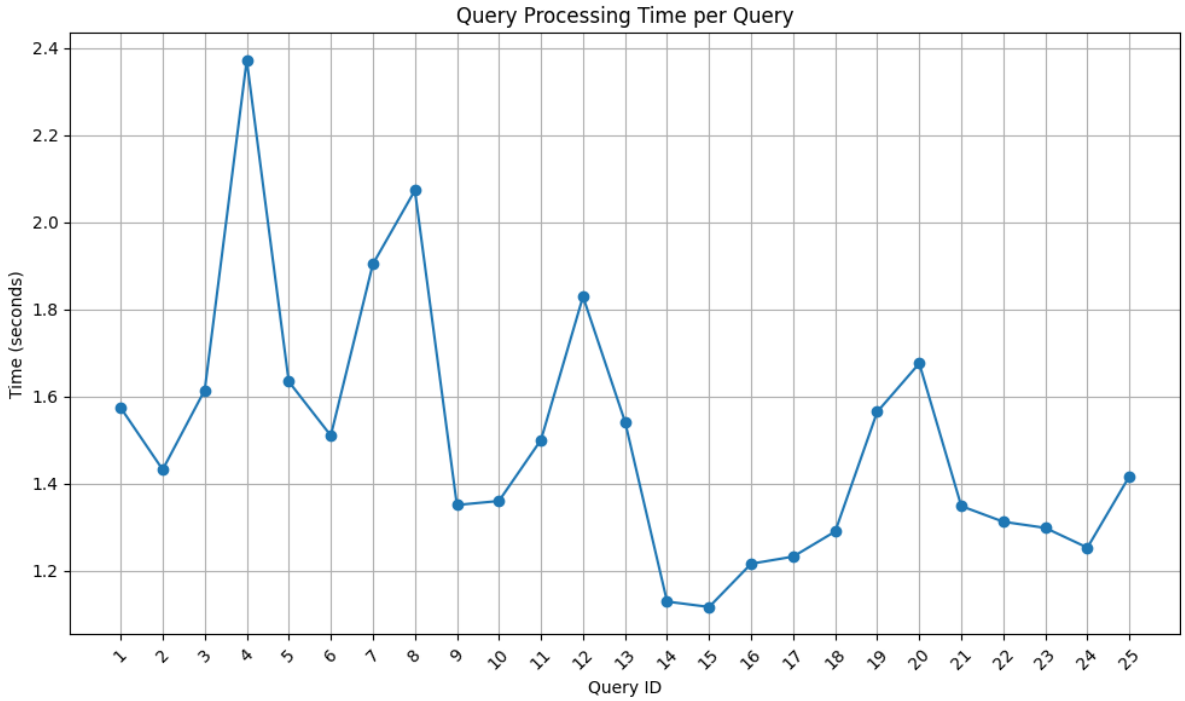
- Simple tokenisation:

```
[DEBUG] Query processing completed in 28.61 seconds.
[DEBUG] Average time per query: 1.14 seconds.
[DEBUG] Results written to result
[DEBUG] Query processing completed.
[DEBUG] Execution time: 80.48123383522034 seconds.
```



- BPE

```
[DEBUG] Query processing completed in 37.54 seconds.
[DEBUG] Average time per query: 1.50 seconds.
[DEBUG] Results written to result
[DEBUG] Query processing completed.
[DEBUG] Execution time: 84.67836713790894 seconds.
```


Query Processing Time per Query
Time (seconds)
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Query ID