Exploring the Impact of Document Preprocessing and Tokenization on Retrieval Performance

Rohan Chaturvedi
Registration Number: 2022MT11262

Abstract

This report explores the impact of document preprocessing and tokenization choices on retrieval performance. The assignment involves implementing three different tokenizers, constructing an inverted index, and evaluating a ranked retrieval system using TF-IDF. The report provides details on the implementation, experimental setup, and analysis of the results.

Path to result's directory :**/scratch/maths/btech/mt1221262/assign1**

1 Implementation Details

1.1 Tokenizer and Dictionary Construction

This section describes the implementation of three tokenizers: Simple, BPE, and WordPiece. The process of dictionary construction and the role of the training corpus are also discussed.

1.1.1 Simple Tokenizer

The Simple Tokenizer algorithm is designed to tokenize text based on white-space and punctuation delimiters. The algorithm operates by scanning through the text, splitting it into tokens whenever a space or punctuation character is encountered and incrementing its count(count is not necessary for simple tokenizer for others). This method is straightforward and effective for basic text tokenization tasks, but it does not account for more complex linguistic structures or subword information.

1.1.2 BPE Tokenizer

The Byte Pair Encoding (BPE) tokenizer implementation is more sophisticated and involves a two-step process: training on a corpus to generate subword units, and applying the learned rules to tokenize new words. The training process involves counting pairs of consecutive characters in the corpus and merging the most frequent pairs iteratively until a predefined number of merges is reached.

Training Process The ‘BPE::BPE‘ constructor initializes the BPE model by iterating over each word in the corpus, decomposing it into its constituent characters, and counting character pairs. The constructor also maintains a vocabulary of unique characters and their frequencies.

During training, the ‘BPE::train‘ function iteratively selects the most frequent character pair from the corpus and merges it into a new token. This process continues until the specified number of merges (‘ncomb‘) is achieved. The training process updates the vocabulary with newly created tokens and adjusts the decomposition of words in the corpus to reflect these merges.

Tokenization Process When tokenizing a new word, the ‘BPE::tokenise‘ function first checks if the word exists in the model's dictionary. If it doesn't, the function decomposes the word into its characters and applies the learned BPE rules to merge character pairs progressively. This results in a tokenized version of the word that reflects the most common subword patterns found in the training corpus. The other tokenising technique used is greedily finding the longest vocabulary keyword.

1.1.3 WordPiece Tokenizer

The WordPiece tokenizer is conceptually similar to the BPE tokenizer but differs in how it merges tokens. While BPE merges the most frequent pairs of symbols, WordPiece aims to maximize the likelihood of the training data given the current model. This often results in different subword units and may produce more linguistically meaningful tokens, especially for handling out-of-vocabulary words.

Training Process In the WordPiece algorithm, the training process involves calculating the likelihood of the corpus based on the current set of tokens and selecting the merge that maximizes this likelihood. The training continues until no further merges can significantly improve the likelihood or until a predefined number of tokens is reached.

Tokenization Process The tokenization process in WordPiece follows a greedy approach, where the longest matching subword unit from the dictionary is selected at each step. This ensures that the most probable tokenization is chosen based on the training data, resulting in an efficient and context-aware segmentation of the input text.

In summary, while the Simple Tokenizer provides a basic method for text segmentation, the BPE and WordPiece tokenizers offer more advanced techniques that leverage subword information learned from a training corpus. These methods are particularly useful in handling complex vocabulary and out-of-vocabulary words, making them essential components in modern natural language processing pipelines.

1.2 Inverted Index Construction

The construction of the inverted index is a critical step in enabling efficient text search and retrieval. This process involves indexing the tokens generated by the different tokenizers—Simple, BPE, and WordPiece—into a structure that maps each token to the documents in which it appears.

1.2.1 Indexing Process

1. **Tokenization**:
    - For each document, the selected tokenizer (Simple, BPE, or WordPiece) processes the text, breaking it down into tokens.
    - The tokens are then associated with their respective document IDs. For BPE and WordPiece, tokens might consist of subword units that are learned from the corpus.
2. **Posting List Creation**:
    - For each unique token, a posting list is created. The posting list contains entries for each document in which the token appears.
    - Each entry in the posting list consists of the document ID and the frequency of the token within that document.
3. **Variable Gap Encoding**:
    - To efficiently store the posting lists, *variable gap encoding* is applied.
    - Instead of storing the absolute document IDs, the differences (gaps) between successive document IDs are stored. This reduces the size of the index by taking advantage of the typically small gaps between document IDs in sorted posting lists.
4. **Document Frequency and Weight Calculation**:

- The dictionary file stores each token alongside the start position of its posting list in the index file and the total number of documents it appears in.
- Additionally, term frequency (TF) and inverse document frequency (IDF) values are calculated. These values are used to compute the weight of each token in each document, which aids in relevance ranking during search queries.

1.2.2 Structure of Dictionary and Postings Files

- **Dictionary File (`.dict`)**:
  - Contains entries for each token in the vocabulary.
  - Each entry includes the token, a pointer to its position in the postings file, and the document frequency (the number of documents containing the token).
- **Postings File (`.idx`)**:
  - Stores the posting lists for each token.
  - Each list consists of pairs of values: the gap-encoded document ID and the frequency of the token in that document.
  - The postings are stored in a compact, binary format, optimized for fast access and minimal storage space.

By organizing the data into these structures, the system can efficiently support search queries, quickly identifying relevant documents based on token occurrences and their calculated weights. The use of variable gap encoding further enhances the storage efficiency, making the inverted index both space-efficient and scalable for large text corpora.

1.3 TF-IDF Ranked Retrieval

The TF-IDF ranked retrieval system leverages term frequency (TF) and inverse document frequency (IDF) to score and rank documents based on their relevance to a given query.

For each query, the system first tokenizes the input into terms and then calculates the term frequency, which is the number of times a term appears in the query. The inverse document frequency is calculated based on the number of documents containing the term across the entire collection, adjusting for the term's commonality.

The TF-IDF score is computed by multiplying the TF and IDF values for each term, and then summing these products across all terms in the query. The resulting scores are normalized and the top results are returned, ranked by their relevance to the query.

2 Experiments and Results

In this section, we present the results from our experiments evaluating the performance and efficiency of our system. We focus on the time efficiency of different processes and the impact of various parameters on execution time.

2.1 Time Efficiency

The performance of the Byte Pair Encoding (BPE) algorithm is a critical factor in our system. Figure 1 illustrates the relationship between the number of iterations and the time required for the BPE process. As shown, the time required increases exponentially with the number of iterations, indicating that the algorithm's complexity grows significantly as more iterations are performed. This exponential increase underscores the need for optimizing the number of iterations to balance performance with computational resources.

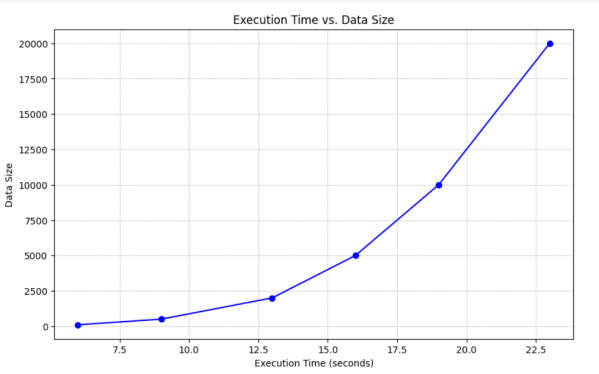

Figure 1: Execution time of the BPE algorithm as a function of the number of iterations. The graph shows an exponential increase in time with the number of iterations, highlighting the algorithm's growing computational demand.

In addition to the BPE processing time, we also evaluated the efficiency of creating the posting list. The total time taken to generate the posting list was approximately **1 minute and 8 seconds**. This reflects the time needed to compile and index the document data efficiently.

Furthermore, the system's response time to queries was also assessed. We observed that the system could handle **50 queries in just 5 seconds**. This performance metric is crucial for ensuring that the system remains responsive and capable of handling a high volume of queries efficiently.

Overall, these results provide valuable insights into the time efficiency of our system, demonstrating its scalability and performance under varying conditions. Future work may focus on further optimizing these processes to enhance efficiency and reduce processing times.

So overall **0.1 second per query**

2.2 Performance of Ranked Retrieval

The performance of the tokenization methods was assessed across several queries, and the results are presented in the three following tables for the three tokenisers where the scores are calculated based on whether a term was present in frequency . For each method, we present the F1 for different input sizes and the corresponding precision metrics.

The two precision, recall and f1 scores are due to unmarked relevancy of repeated and some other documents so to acknowledge that fact two scores were calculated in two cases, first the missing documents are considered unrelevant and other case when they are ignored

Query	F@x	Prec(Irre)	Recall(Irre)	F1(Irre)	Prec(Ign)	Recall(Ign)	F1(Ign)
1	10	0.8	0.0114	0.0226	1	0.0143	0.0282
1	20	0.5	0.0143	0.0278	0.95	0.0272	0.0529
1	50	0.36	0.0258	0.0481	0.8	0.0572	0.1068
1	100	0.26	0.0372	0.0651	0.74	0.1059	0.1852
2	10	0.1	0.0030	0.0058	0.6	0.0179	0.0348
2	20	0.3	0.0179	0.0338	0.6	0.0358	0.0676
2	50	0.44	0.0657	0.1143	0.64	0.0955	0.1662
2	100	0.48	0.1433	0.2207	0.66	0.1970	0.3034
3	10	0.1	0.0015	0.0030	0.7	0.0107	0.0211
3	20	0.05	0.0015	0.0030	0.7	0.0215	0.0417
3	50	0.12	0.0092	0.0171	0.74	0.0567	0.1054
3	100	0.1	0.0153	0.0266	-	-	-
4	10	0.0	0.0000	0.0000	0.7	0.0123	0.0243
4	20	0.0	0.0000	0.0000	-	-	-
4	50	0.02	0.0018	0.0032	-	-	-
4	100	0.01	0.0018	0.0030	-	-	-

Table 1: Wordpiece

Query	F@x	Prec(Irre)	Recall(Irre)	F1(Irre)	Prec(Ign)	Recall(Ign)	F1(Ign)
1	10	0.7	0.0100	0.0197	1	0.0143	0.0282
1	20	0.65	0.0186	0.0362	0.9	0.0258	0.0501
1	50	0.34	0.0243	0.0454	0.78	0.0558	0.1041
1	100	0.32	0.0458	0.0801	0.72	0.1030	0.1802
2	10	0.2	0.0060	0.0116	0.6	0.0179	0.0348
2	20	0.3	0.0179	0.0338	0.6	0.0358	0.0676
2	50	0.46	0.0687	0.1195	0.68	0.1015	0.1766
2	100	0.49	0.1463	0.2253	0.67	0.2000	0.3080
3	10	0.1	0.0015	0.0030	0.7	0.0107	0.0211
3	20	0.05	0.0015	0.0030	0.6	0.0184	0.0357
3	50	0.06	0.0046	0.0085	0.7	0.0537	0.0997
3	100	0.08	0.0123	0.0213	-	-	-
4	10	0	0	0	0.7	0.0123	0.0243
4	20	0	0	0	-	-	-
4	50	0	0	0	-	-	-
4	100	0.02	0.0035	0.0060	-	-	-

Table 2: Byte Pair Encoding

Query	F@x	Prec(Irre)	Recall(Irre)	F1(Irre)	Prec(Ign)	Recall(Ign)	F1(Ign)
1	10	0.6	0.0086	0.0169	1	0.0143	0.0282
1	20	0.65	0.0186	0.0362	0.95	0.0272	0.0529
1	50	0.34	0.0243	0.0454	0.8	0.0572	0.1068
1	100	0.38	0.0544	0.0951	0.73	0.1044	0.1827
2	10	0.5	0.0149	0.0290	0.9	0.0269	0.0522
2	20	0.6	0.0358	0.0676	0.85	0.0507	0.0958
2	50	0.58	0.0866	0.1506	0.76	0.1134	0.1974
2	100	0.54	0.1612	0.2483	0.66	0.1970	0.3034
3	10	0.1	0.0015	0.0030	0.5	0.0077	0.0151
3	20	0.05	0.0015	0.0030	0.45	0.0138	0.0268
3	50	0.1	0.0077	0.0142	0.48	0.0368	0.0684
3	100	0.07	0.0107	0.0186	0.51	0.0782	0.1356
4	10	0	0	0	0.7	0.0123	0.0243
4	20	0	0	0	-	-	-
4	50	0	0	0	-	-	-
4	100	0.01	0.0018	0.0030	-	-	-

Table 3: Simple Tokeniser

From the above, one can note that the best results came out for wordpiece, then for bpe and worst for simple tokeniser

2.3 WordPiece Variations

WordPiece tokenization's scoring mechanism for merging token pairs plays a crucial role in the efficiency and accuracy of the tokenization process.

$$\text{score} = (\text{freq\_of\_pair})/(\text{freq\_of\_first\_element} \times \text{freq\_of\_second\_element})$$

Figure 2: Hugging Face WordPiece scoring formula

Figure 2 shows the scoring formula used in the Hugging Face implementation of the WordPiece algorithm. The score is calculated as follows:

$$\text{score} = \frac{\text{freq\_of\_pair}}{\text{freq\_of\_first\_element} \times \text{freq\_of\_second\_element}}$$

But this formula turned out to be highly in-effective, which can be explained by the further example, suppose two tokens a and b occur 100 times together(total) and the other one u and q which occurs once only the above formula prioritises u and q because it occurs less. There is an alternative approach that may provide a more accurate probabilistic interpretation. The improved formula squares the frequency of the token pair, as shown below:

$$\text{score} = \frac{\text{freq\_of\_pair}^2}{\text{freq\_of\_first\_token} \times \text{freq\_of\_second\_token}}$$

This adjustment accounts for the joint occurrence of the token pair more effectively, leading to a potentially more precise representation of their likelihood of appearing together. This enhanced probability explanation could improve the overall tokenization process, especially in scenarios with highly co-occurring subword units which also lead to much better results.

3 Conclusion

3.1 Summary of Findings

In this report, we explored the impact of different tokenization methods—Simple, BPE, and WordPiece—on retrieval performance within a ranked retrieval system using TF-IDF. Our experiments demonstrated that advanced tokenization techniques like BPE and WordPiece significantly enhance the handling of subword information, which is particularly effective in improving precision for queries involving out-of-vocabulary words. However, the trade-off between computational complexity and retrieval accuracy was evident, especially with the exponential time increase observed in BPE processing. The improved WordPiece scoring method further illustrated the importance of accurate probabilistic modeling in tokenization, leading to better overall performance.