COL764 Assignment 1

Malik Hammad Faisal

September 5, 2024

Simple Tokenizer

It creates tokens by splitting the extracted text using whitespace and a set of punctuation marks (e.g., ",.:;"), treating them as delimiters.

Byte Pair Encoding (BPE) Tokenizer Dictionary Construction

1. **Initial Tokenization:** The text is first tokenized using a simple tokenizer. This tokenizer extracts word frequencies from the input text.

2. **Vocabulary Initialization:** An initial vocabulary is constructed using all ASCII characters (0-127). This includes letters, digits, and punctuation symbols.

3. **Encoding Words:** Each word in the text is represented as a sequence of tokens from the vocabulary. The algorithm maintains a list of token pairs within words and their respective frequencies.

4. **Token Pair Merging:** The core of BPE is the iterative merging of token pairs that appear frequently. For each token pair, its frequency is computed, and the pair with the highest frequency is merged into a new token. This process continues until the desired vocabulary size is reached.

5. **Priority Queue Management:** To efficiently find the most frequent token pairs, a priority queue is used. This queue stores the frequencies of token pairs, allowing fast access to the most frequent pair for merging.

6. **Updating Encodings:** After each merge, the word encodings are updated to reflect the new token, and the token pair frequencies are adjusted accordingly. Words that contain the merged tokens are re-encoded, and their corresponding token pairs are updated in the priority queue.

7. **Dictionary Output:** The final vocabulary and the list of merges are written to a file. The file contains the vocabulary followed by the pairs of tokens that were merged.

WordPiece Tokenizer Dictionary Construction:

The provided code implements a WordPiece tokenizer which closely resembles Byte Pair Encoding (BPE). Initially, it starts with a vocabulary that includes all ASCII characters (with a distinction for tokens starting with a space), and their alphanumeric classification is recorded. The process begins by tokenizing the input text using a simple tokenizer that splits words based on punctuation.

Token pairs are then iteratively merged, similar to BPE, based on their frequency and a scoring function:

- For each token pair, a score is calculated. If a token is alphanumeric, $\text{tok\_den} = \sqrt{\text{tok\_frequency}}$, otherwise $\text{tok\_den} = \text{tok\_frequency}$

  $$\text{score} = \frac{\text{pair\_freq}}{\text{tok1\_den} \times \text{tok2\_den}}$$

  This is done since in the function given on hugging face, pairs of highly occurring small tokens or letters are heavily penalized and they are not merged in atleast the initial 1000.

- Token pairs with the highest scores are merged, and the process is repeated until the target vocabulary size is reached. Only the vocabulary entries are stored in a dictionary file.

- The tokenization involves maintaining a frequency table of token pairs, updating the priority queue for optimal merging, and applying the merges to the encoded words in an efficient manner. The final output is a word dictionary suitable for use in WordPiece tokenization.

The key difference from BPE lies in how the scoring function penalizes pairs in which tokens occur a lot individually also. It also takes alphanumeric characters into account by penalizing certain tokens differently based on their types.

Creating Inverted Index

Tokenization

1. Simple Tokenizer only considers the words it has in its vocabulary and other words get classified as UNK.
2. BPE Tokenizer maintains a linked list of tokens. Token pairs are merged according to the merge priority. Priority is handled using balanced BST which is updated after each merge. This goes on till no more merges are possible.
3. WordPiece Tokenizer stores its vocabulary in a trie. It parses each word from left to right and creates longest token possible at each index. The next iteration starts after removing the token formed from the start of the word.
4. BPE and WordPiece were initialized with all ASCII characters so they do not need UNK tokens.

Inverted Index Creation Process

The given code constructs an inverted index from a collection of documents. The key steps involved are as follows:

1. **File Reading and Tokenization**:
   - A large document file is read line by line using an `ifstream` stream.
   - For each line, values are extracted from the JSON structure, and the relevant text is tokenized using one of the three tokenization methods:
     (a) Simple Tokenization,
     (b) Byte Pair Encoding (BPE),
     (c) WordPiece Tokenization.
   - The tokens are paired with the document ID and stored in `term_doc_pair`.
2. **Buffering and Sorting**:
   - Once the buffer reaches a pre-defined size limit (`MAX_SIZE`), the token-document pairs are sorted.
   - A `postings_list` is created, where for each term ID, a list of document IDs and their respective term frequencies is generated.
   - The postings list is written to a temporary file (e.g., `temp_inverted_index_X.idx`).
3. **Final Merging**:
   - After processing all documents, the temporary files are read back for merging.
   - For each term in the vocabulary, the corresponding document lists from different temporary files are merged.
   - The merged lists are written to the final inverted index file.
4. **Output Format**:
   - The final output consists of term-document frequency pairs written in binary format.
   - For each term, its posting list (containing document IDs and term frequencies) is stored efficiently in the output index file.

TF-IDF Retrieval Process

The given code retrieves documents using the TF-IDF scoring method. The key steps are as follows:

1. **Initialization**:
   - `doc_to_id`, `doc_id_to_name`, and `doc_norm` are initialized to map document names to IDs, store document names, and store document norms, respectively.
   - `tf` (term frequency) and `idf` (inverse document frequency) scores are calculated using the postings list.
   - `doc_norm` stores the squared sum of term frequencies for each document. Later, the square root of these sums is taken to normalize document scores.
2. **Query Processing**:
   - Queries are read from a file and tokenized.
   - For each token, the term frequency (`tf`) and inverse document frequency (`idf`) are used to compute the query's weight for that term:
     $$\texttt{weight} = (1 + \log_2(\texttt{token\_count})) \times \texttt{idf}.$$
   - The document scores are updated by accumulating the weighted `tf-idf` values for all matching tokens in the document.
3. **Score Normalization and Ranking**:
   - Document scores are normalized using the document norms. Query norm would be same for all documents for the same query so it is not used.:
     $$\texttt{normalized\_score} = \frac{\texttt{score}}{\texttt{doc\_norm}}.$$
   - The documents are ranked by their normalized scores, and the top 100 results are written to an output file.

Performance Comparison of BPE and WordPiece

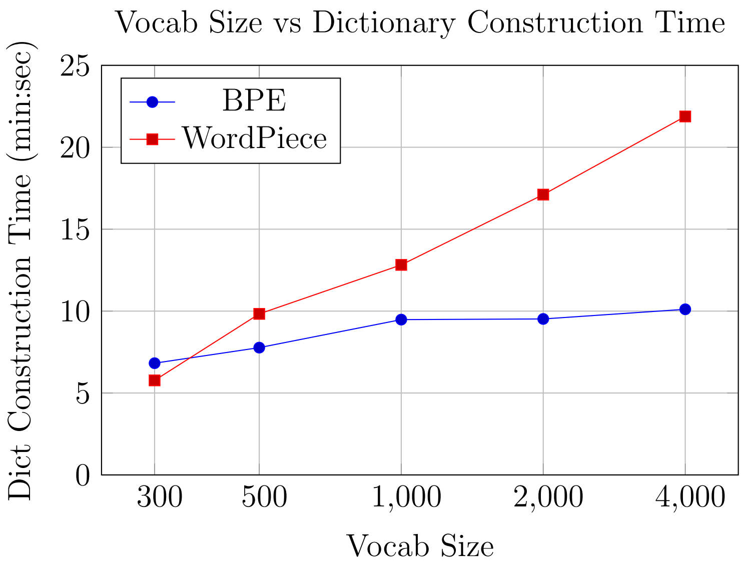

Figure 1: Vocab Size vs Dictionary Construction Time for BPE and WordPiece

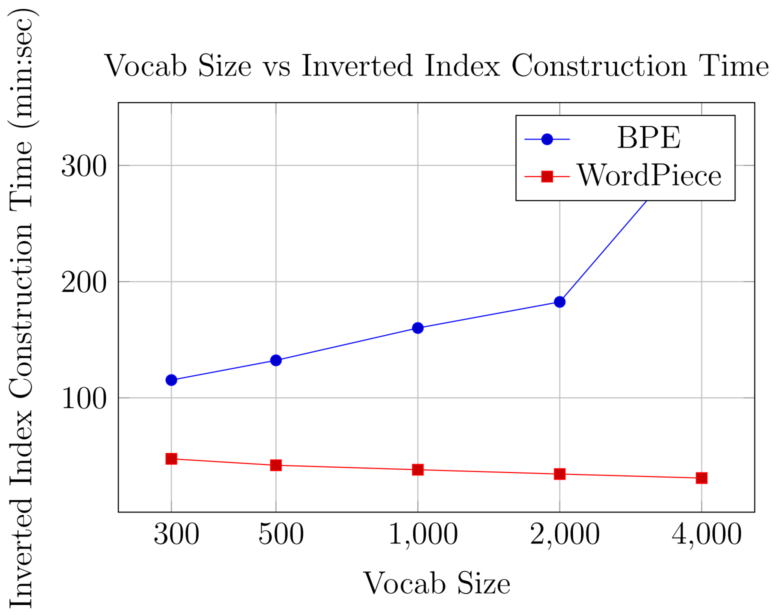

Figure 2: Vocab Size vs Inverted Index Construction Time for BPE and WordPiece

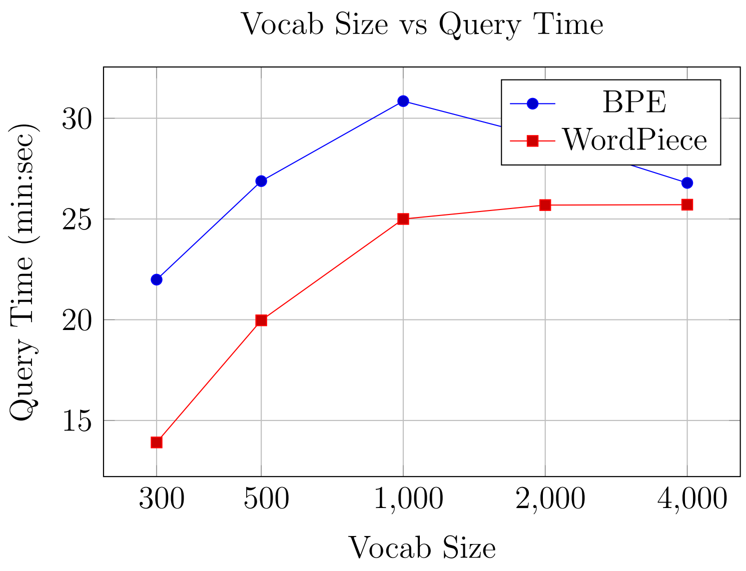

Figure 3: Vocab Size vs Query Time for BPE and WordPiece

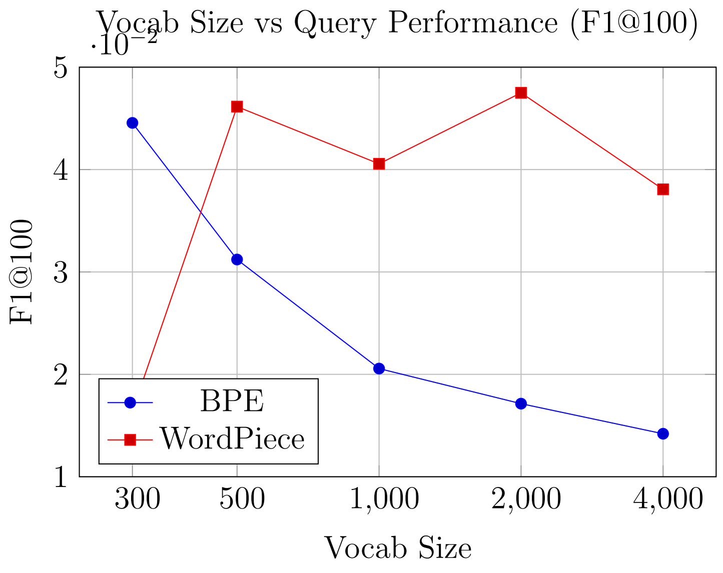

Figure 4: Vocab Size vs Query Performance (F1@100) for BPE and WordPiece

Results

Based on the analysis above, in the final submission BPE has vocab size 300, and WordPiece has size 500.

Tokenization	Dict Construction Time (min:sec)	Inverted Index Construction Time (min
Simple	0:04.44	0:31.01
BPE	0:06.82	1:55.31
WordPiece	0:09.83	0:42.00

Table 1: Dictionary and Inverted Index Construction Time

Tokenization	F1@10	F1@20	F1@50	F1@100
Simple	0.00809142	0.0111584	0.0165684	0.0215435
BPE	0.0105046	0.0177139	0.0307509	0.044556
WordPiece	0.00993683	0.0175744	0.0323414	0.0461473

Table 2: F1 Scores at Different Cutoffs for Different Tokenization Methods

Tokenization	Per-Query Time (sec)
Simple	0.83
BPE	0.88
WordPiece	0.80

Table 3: Per-Query Time Efficiency for Different Tokenization Methods

details of directories on HPC

/scratch/cse/btech/cs1210559/col764_ass1

It contains {simple,bpe,wordp}.{dict,idx} files which are dictionary and inverted index files for all tokenizers.