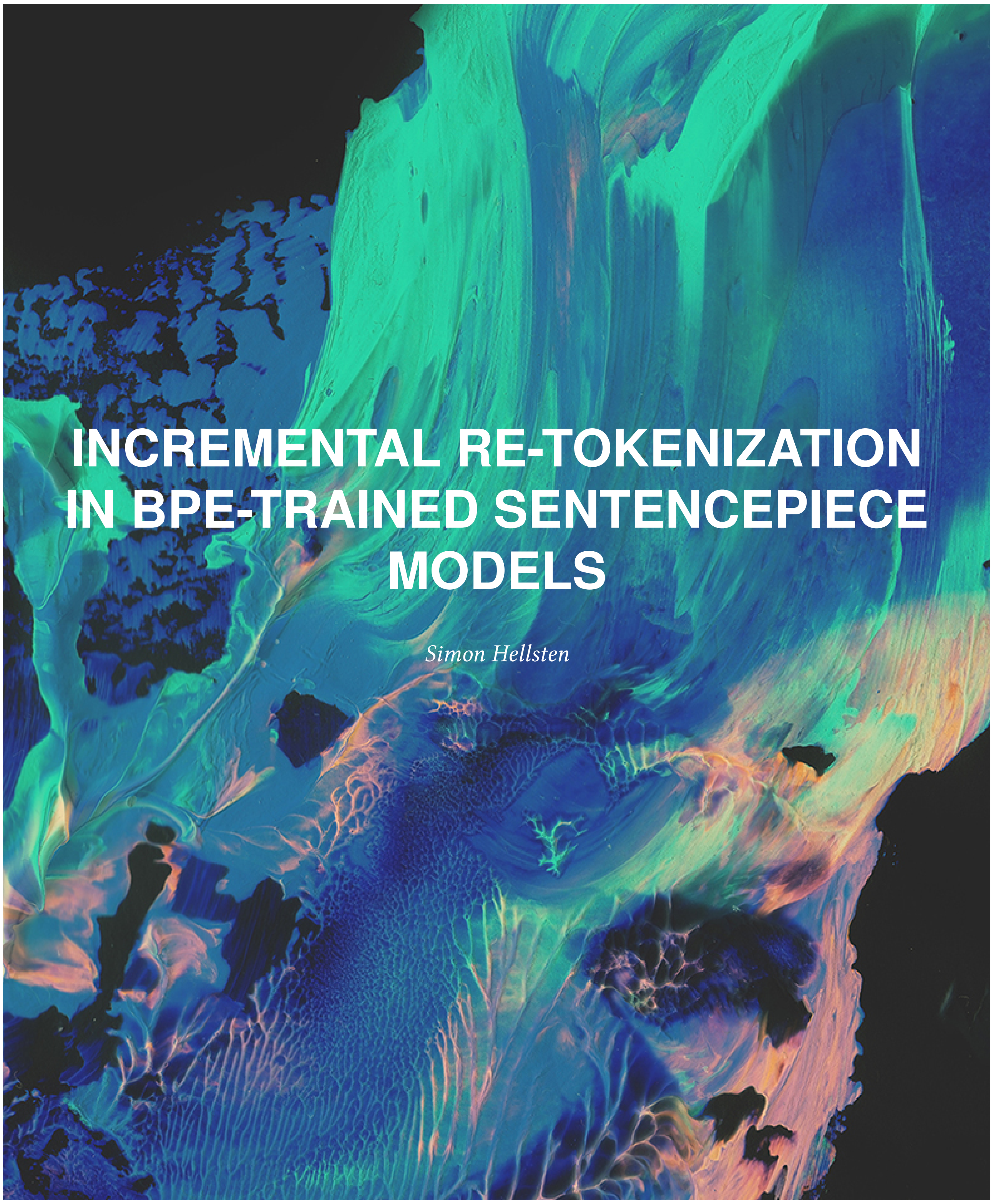
UMEÅ UNIVERSITY



INCREMENTAL RE-TOKENIZATION IN BPE-TRAINED SENTENCEPIECE MODELS

Simon Hellsten

Bachelor Thesis, 15 credits
BACHELOR OF SCIENCE IN COMPUTING SCIENCE

2024

Abstract

This bachelor's thesis in Computer Science explores the efficiency of an incremental re-tokenization algorithm in the context of BPE-trained SentencePiece models used in natural language processing. The thesis begins by underscoring the critical role of tokenization in NLP, particularly highlighting the complexities introduced by modifications in tokenized text. It then presents an incremental re-tokenization algorithm, detailing its development and evaluating its performance against a full text re-tokenization. Experimental results demonstrate that this incremental approach is more time-efficient than full re-tokenization, especially evident in large text datasets. This efficiency is attributed to the algorithm's localized re-tokenization strategy, which limits processing to text areas around modifications. The research concludes by suggesting that incremental re-tokenization could significantly enhance the responsiveness and resource efficiency of text-based applications, such as chatbots and virtual assistants. Future work may focus on predictive models to anticipate the impact of text changes on token stability and optimizing the algorithm for different text contexts.

Acknowledgements

I would like to express my special thanks to my supervisor Martin Berglund for his guidance and thoughtful feedback. I am also deeply grateful to my parents for their unwavering support.

And to my partner, Frida, thank you for your patience and love. I could not have done this without you.

Contents

1 Introduction

In the field of natural language processing (NLP), tokenization plays a fundamental role. It involves breaking down text into smaller units, known as tokens, which are used as the basis for various computational tasks. For instance, consider the string "`ab-c`", a tokenization method might break this into the three tokens: "`ab`", "`-`", and "`c`". One of the popular tokenization methods is Byte Pair Encoding (BPE) [1], which has been widely used in various NLP models, including the large language model GPT-3 [2]. In addition to BPE, this thesis also examines SentencePiece [3] which is a text tokenizer and detokenizer, that implements BPE and a unigram language model (the latter is not be covered by this thesis). The SentencePiece library has in turn been used to train new BPE tokenizers [4], such as the tokenizer used by the GPT-SW3 model [4, 5].

When changes occur in a tokenized text, the process of updating the tokens, referred to as re-tokenization, can introduce complexity. Consider a situation where the hyphen from our original string "`ab-c`" is removed, resulting in the string "`abc`". One might expect that simply removing the "`-`" token would suffice, leaving us with the two tokens "`ab`" and "`c`". But depending on the tokenization scheme, the correct way to tokenize the updated string could just as well be the tokens "`a`" and "`bc`", or even just a single token "`abc`". This example demonstrates that even a minor modification can affect neighboring tokens. Based on this observation, it can be further inferred that these changes may also influence subsequent tokens in both directions, potentially leading to a situation where no token remains unchanged.

In cases where re-tokenization is needed, the straightforward approach is to perform the tokenization process again for the entire text. This guarantees the consistency of the new tokens with the established tokenization scheme. However, re-tokenizing the whole text can be computationally demanding and time-consuming, especially for large volumes of text data. This highlights the necessity for more efficient methods for re-tokenization.

Expanding on the significance of this research, a more efficient re-tokenization method could provide various benefits in the field of NLP. For instance, in the context of machine translation [6], where documents are often updated or corrected, using an algorithm that can incrementally adjust the tokenization of revised sections instead of reprocessing the entire document could result in considerable time and computational savings. The same holds true in sentiment analysis [7], where things such as social media posts or customer reviews are continually updated. Furthermore, in applications such as simple text editing and grammar checking, where text is frequently modified, efficient re-tokenization can enhance the responsiveness and accuracy of these tools, allowing for real-time corrections and suggestions without the need to reprocess the entire text from scratch.

Considering these potential benefits, this thesis addresses the research question: "*Can an incremental algorithm efficiently re-tokenize text after minor changes in BPE-trained SentencePiece models without reprocessing the entire text?*" The objectives of this research are: (1) to develop an incremental algorithm for re-tokenization in BPE-trained SentencePiece models, (2) to reason for the correctness of the developed algorithm, and (3) to evaluate the performance of the developed algorithm against fully tokenizing the altered text using SentencePiece. In essence, this research aims to pave the way for more efficient approaches to tokenization, contributing meaningfully to the advancement of NLP.

This thesis employs a combination of literature review, algorithm development, and experimental evaluation. The literature review provides an overview of existing tokenization methods, the state-of-the-art in BPE-trained SentencePiece models, as well as some further necessary background and context for BPE and SentencePiece [8][9]. The central contribution is the development of an incremental re-tokenization algorithm for BPE-trained SentencePiece models, aimed at efficiently updating tokenized text following minor modifications. The development of this algorithm is detailed, and its performance is compared with the naive approach of a full re-tokenization. Results from these experiments show that the incremental algorithm consistently outperforms full re-tokenization in terms of processing time.

This research focuses on the development of an incremental algorithm for re-tokenization within the context of BPE-trained SentencePiece models. While the findings of this study may be applicable to other tokenization methods and NLP models, the scope of this research is limited to BPE and SentencePiece.

The thesis is organized as follows: Section 1 provides an overview of the study's context and its importance in the field of NLP, specifically focusing on the challenges and significance of tokenization and re-tokenization. Section 2 delves into various tokenization methods and their applications in advanced NLP models, highlighting the importance of efficient text processing. Section 3 lays the foundational knowledge about BPE and SentencePiece, which is crucial for understanding the subsequent development of the incremental re-tokenization algorithm. Section 4 introduces the incremental re-tokenization algorithm and provides reasoning for its correctness. Section 5 describes the methodology and conditions under which the algorithm's performance is empirically evaluated. Section 6 presents and interprets the findings from the experimental evaluation. Finally, Section 7 reviews the research findings, discusses their implications, and suggests directions for future research in the domain of efficient re-tokenization.

2 Related Work

Various tokenization methods have been proposed and employed in the field of NLP. Wu et al. [10] introduced Google's Neural Machine Translation system, which uses a WordPiece tokenization method to address the open vocabulary problem and improve the handling of rare words. The WordPiece method recursively splits words into subwords based on their frequency in a large corpus. This approach strikes a balance between character-delimited models and word-delimited models, offering both flexibility and efficiency. Schuster and Nakajima [11] initially applied WordPiece tokenization to Japanese and Korean voice search, demonstrating its effectiveness in handling multiple script languages and addressing the challenges of infinite vocabulary and ambiguities in scoring results.

Taku Kudo's work [3] focuses on improving neural machine translation (NMT) through subword regularization, which utilizes "multiple subword segmentations probabilistically sampled during training" to enhance model robustness. Additionally, Kudo introduces tokenization based on a unigram language model, a probabilistic model that predicts the likelihood of a sequence of subword units based on the frequencies of their occurrence in a corpus, offering a method to generate subword units with associated probabilities [3]. SentencePiece implements Kudo's unigram language model, providing an alternative to BPE for token segmentation [12].

The Transformer model, introduced by Vaswani et al. [13], has been influential in the development of state-of-the-art NLP models. The authors proposed a novel network architecture based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. This approach led to superior results in quality while being more parallelizable and requiring significantly less time to train [13]. The adoption of tokenization methods, such as WordPiece, in Transformer-based models has contributed to their high performance in various NLP tasks.

Following the advancements of Transformer models, the BERT model introduced by Devlin et al. [14] represents a significant leap in NLP by employing the WordPiece tokenization. This method allows BERT to efficiently handle a wide vocabulary, including rare words, by breaking them into smaller, manageable subwords. Crucially, BERT enhances understanding by analyzing text from both left and right contexts simultaneously, leveraging bidirectional training unlike previous models. This approach enables BERT to achieve superior performance in tasks such as question answering and language inference with minimal architectural adjustments [14]. The key takeaway here is BERT's demonstration of how advanced tokenization techniques, integrated within a Transformer-based framework, can substantially improve NLP applications, underscoring the importance of context-aware processing and efficient tokenization in pushing the boundaries of what's possible in language understanding.

Efficiency is not only an important consideration in tokenization but also in other text processing tasks, such as regular expression evaluation. In their work, Björklund, Martens, and Timm [15] specifically focus on the efficient incremental evaluation of regular expressions. This area is pivotal due to the widespread presence of the counting operator in regular expressions, a popular extension used in multiple programming languages and database applications. The authors proposed a new algorithm that significantly accelerates this process.

They demonstrated that exploiting counting operators can result in substantial speed-ups in different scenarios, such as both normal and incremental evaluation, and on both synthetic and real expressions. This research highlights the broader relevance of incremental algorithms in improving the efficiency of text processing tasks.

Concurrently with the development of this thesis, Berglund and van der Merwe's article "Formalizing BPE Tokenization" [16] has contributed to our understanding of BPE tokenization, a key focus of this thesis. Their work provides a comprehensive formalization of BPE tokenization, as implemented in frameworks like SentencePiece and HuggingFace, offering insight into the construction of tokenization rules and their implications. Moreover, they propose an algorithm for incremental evaluation of tokenizations, although they acknowledge that this may not always be practical due to computational constraints. This aligns closely with the research presented in this thesis, as their theoretical groundwork lays the foundation for practical exploration. While their work abstains from experimental studies, this thesis aims to fill this gap by conducting experiments to validate the intuitive understanding and efficiency of their incremental algorithm in real-world scenarios. My thesis thus extends their theoretical contributions with an empirical perspective, shedding light on the practical applications and limitations of the incremental update algorithm in BPE tokenization.

3 Background

This section provides the necessary foundation for understanding the core concepts and techniques related to the development of the proposed incremental algorithm for re-tokenization in BPE-trained SentencePiece models. This section is organized into two subsections, each addressing a critical aspect of the problem.

Section 3.1 delves into BPE [1], a widely-used tokenization method known for its effectiveness in handling rare words within natural language processing tasks. This part of the section discusses the principles of the BPE algorithm, its applications, and presents a pseudocode representation to illustrate its structure.

Section 3.2 explores SentencePiece, which is "a language-independent subword tokenizer and detokenizer designed for Neural-based text processing" [3]. This subsection offers an overview of the SentencePiece model, with an emphasis on BPE-trained models. In addition, it examines the C++ implementation of the encoder for BPE-trained models from Google's GitHub repository [12], providing the corresponding pseudocode to facilitate understanding. Additionally, Subsection 3.2.1 provides an example to practically and visually demonstrate how the tokens of a tokenization are constructed.

3.1 Byte Pair Encoding

The BPE algorithm, initially developed for data compression [17], uniquely adapts to the complexities of language processing in NLP. It begins by tokenizing input text and then progressively builds its vocabulary. This is achieved by repeatedly combining the most common pairs of characters or symbols in the text, simplifying large vocabularies by creating these new, efficient subword units [1].

In the context of NLP, the BPE algorithm is applied by first tokenizing the input text and appending an end-of-word symbol to each word [1]. The algorithm then initializes the vocabulary with unique characters from the input text, including the end-of-word symbol. The main part of the BPE algorithm, as presented in Algorithm 1, involves iteratively counting bigram frequencies and merging the most frequent character pairs to form new subword units. This process is repeated for a predefined number of merge operations (or until no further merges are possible), and the resulting subword units are added to the vocabulary. The list of these merge operations, referred to as a *merge list*, is crucial for the application of the BPE algorithm on new words [1].

Definition 3.1.1 (Merge list)**.** A merge list, in the context of the BPE algorithm, is an ordered sequence of the most frequent bigram pairs (x, y) identified in the input text corpus for merging. Each entry in the list represents a specific operation of merging a bigram pair into a new subword unit z, where x and y are individual characters or subwords, and z is the resulting subword from their merge. The order of the list encodes the priority of merge operations, with pairs positioned earlier in the list being prioritized for merging over those that appear later. This prioritization reflects the relative frequency or importance of the bigram pairs in the training data. The merge list encapsulates the subword units learned from the training data and is utilized for segmenting new words.

To apply the learned BPE merge operations to new words, the words are first broken down into individual characters, followed by the end-of-word symbol. The operations in the merge list are then applied iteratively in the same order as they were learned from the training data [1]. This approach allows the BPE algorithm to segment new words into known subword units from the merge list, thereby helping handle large vocabularies and out-of-vocabulary words.

Algorithm 1: Byte Pair Encoding (BPE) for Word Segmentation

```
   Input  : Text corpus $T$, number of merge operations $N$
   Output: Merge list $M$
1  Preprocess the input text corpus $T$;
2  Initialize the vocabulary $V$ with unique characters from $T$;
3  Initialize an empty merge list $M$;
4  for $i \leftarrow 1$ to $N$ do
5      Calculate bigram frequencies in the tokenized text $T$;
6      Find the most frequent bigram pair $(x, y)$;
7      Add the merge operation $(x, y) \longrightarrow z$ to the end of $M$;
8      Update the vocabulary $V$ with the new subword unit $z$;
9      Update the text representation $T$ by merging the bigram pair $(x, y)$ to form the
         new subword unit $z$;
10 end
11 return $M$
```

The use of BPE for word segmentation has been shown to improve the performance of NLP models, especially when dealing with rare and out-of-vocabulary words [10]. However, BPE has some limitations. The choice of the number of merge operations and the size of the training data can significantly impact the quality of the learned subword units [18]. Moreover, BPE may not always capture semantically meaningful subword units, as it is primarily based on frequency counts [19].

Several variants and extensions of the BPE algorithm have been proposed to address its limitations and improve its performance. Notable examples include WordPiece, which is employed in Google's BERT model [14], and SentencePiece, which unifies the tokenization of words and subwords [3].

In conclusion, BPE is an essential technique for word segmentation in NLP tasks, offering a practical approach to handle large vocabularies and out-of-vocabulary words. Despite its limitations, BPE has found widespread adoption in various NLP applications and inspired the development of related segmentation algorithms. Its impact on the field of NLP is expected to continue as researchers explore new techniques and improvements.

3.2 SentencePiece

SentencePiece is “a language-independent subword tokenizer and detokenizer designed for Neural-based text processing, including [NMT]”. Unlike traditional subword segmentation tools, SentencePiece can directly train subword models from raw sentences without the need for pre-tokenization into word sequences. This ability enhances the end-to-end and language-independent nature of the system, making it a pivotal tool in NLP tasks across various languages [3].

SentencePiece incorporates the BPE model, a tokenization method which (as stated in Section 3.1) was originally developed for data compression and later adapted for NLP. The BPE model in SentencePiece is distinctive as it operates directly on raw text, in contrast to traditional BPE approaches that require pre-tokenized text. This integration of BPE within SentencePiece enhances its efficiency and application scope in NLP tasks, particularly in the processing of languages without clear word boundaries [16][3]. SentencePiece’s BPE model manages the vocabulary to ID mapping, allowing for direct conversion of text into ID sequences, differing from the standalone BPE which focuses more on the frequency of bigrams for creating subword units [3].

The SentencePiece tokenization of a string is performed by starting from an initial tokenization state and sequentially applying the merge rules from the merge list. The rules are chosen based on their priority, and the process continues until no further rules can be applied. This method ensures that the highest-priority rule applicable at each step is used, leading to a unique and, using the terminology of Berglund & van der Merwe, “correct” tokenization for each input string [16].

The pseudocode representation of the SentencePiece BPE tokenization process, as outlined in Algorithm 2, provides a step-by-step breakdown of this procedure. It begins with initializing a priority queue and a list of symbols from the input string (lines 1-3). Symbol pairs are then created and added to the priority queue based primarily on their score (priority) and secondly their position in the string, simulating the application of merge rules from the merge list (lines 16-18). The core of the tokenization process involves iterating over this priority queue and merging symbols, ensuring that the highest-priority rule is applied at each step (lines 19-30). The final stage involves constructing the output list of tokens by traversing through the linked list of symbols, thereby achieving the final tokenization of the input string (lines 31-37). This interpretation of SentencePiece’s tokenization with the BPE model is primarily informed by an analysis of the C++ source code from the SentencePiece GitHub repository [12]. Further insight from the paper by Berglund & van der Merwe has been helpful in understanding the nuances of the tokenization process [16]. A Python implementation, imitating the SentencePiece tokenizer for pure BPE models, can be found in Appendix A.

Understanding the tokenization process of SentencePiece with BPE is fundamental to this thesis, especially concerning the development of an incremental algorithm for re-tokenization. The ability to perform incremental updates [16], as suggested by existing research, is vital for efficient NLP processing, especially in scenarios where minor modifications in a string necessitate tokenization adjustments.

Algorithm 2: SentencePiece BPE Tokenization Process

```
Input   : Normalized string $s$
Output: List of tokens $L$
1  Initialize an empty priority queue $Q$;
2  Initialize a list of symbols $S$ from characters of $s$;
3  Link each symbol in $S$ with its previous and next symbol;
4  Function maybe_add_new_symbol_pair(left, right):
5      if either left or right is -1 (invalid) then
6          return
7      end
8      Concatenate pieces from symbols at left and right to form new_piece;
9      Get the ID of new_piece from the model;
10     if piece ID is unknown then
11         return
12     end
13     Get score for the piece ID from the model;
14     Create a new symbol pair with left, right, score, and size of new_piece;
15     Add the symbol pair to $Q$;
16 for $i \leftarrow 1$ to length of $S - 1$ do
17     maybe_add_new_symbol_pair($i - 1$, $i$)
18 end
19 while $Q$ is not empty do
20     Pop the top element from $Q$ as top;
21     if either left or right piece of top is missing or size mismatch then
22         Continue to the next iteration
23     end
24     Merge left and right pieces of top into left;
25     Update next pointer of left to point to the next of right;
26     If next of left is valid, update its previous pointer to left;
27     Set right piece as empty (fully merged);
28     maybe_add_new_symbol_pair(previous of left, left);
29     maybe_add_new_symbol_pair(left, next of left);
30 end
31 Initialize an empty list $L$;
32 Set current_index to the index of the first symbol in $S$;
33 while current_index is not -1 do
34     Add the piece at current_index to $L$;
35     Update current_index to the symbol's next index;
36 end
37 return $L$
```

3.2.1 Tokenization example

This section provides a demonstration of the tokenization process for the text "*ragged edge*". Ahead of the tokenization process, each new word is marked with a special meta symbol "▁" (U+2581), replacing any space character as well, this according to how SentencePiece treats whitespace [12], subsequently creating a token for every character in the text. Note that the use of this meta character eliminates the need for the end-of-word characters mentioned in Section 3.1.

The tokenization process is initialized by finding the first occurrence of a merge rule with a corresponding bigram pair in "▁ragged▁edge", which in this case is "▁ e", as seen in Table 1b. The said pair appears before any of the other possible merges in the merge list (implying that the pair "▁ e" appears most frequently in the training data, assuming the merge list was created using BPE), thus the pair "▁ e" have the highest priority to be applied. In consequence, the tokens "▁" and "e" are merged, shown as the first step (1) in Table 1a.

Further, this procedure is repeated for the tokens that remain. Hence, the tokens "e" and "d" are merged next according to the merge list in Table 1b, since the bigram pair "e d" now has the highest priority as the first occurring merge rule. Eventually, no more merges are possible and the tokenization process concludes, resulting in the tokens: "▁ra", "gg", "ed", "▁ed", and "ge".

Table 1 Tokenization of the text "*ragged edge*".

(a) Tokenization steps, applying the merge rule with the highest priority in each step.

Step	Tokens											
0	▁	r	a	g	g	e	d	▁	e	d	g	e
1	▁	r	a	g	g	e	d	**▁e**		d	g	e
2	▁	r	a	g	g	**ed**		▁e		d	g	e
3	▁	**ra**		g	g	ed		▁e		d	g	e
4	**▁ra**			g	g	ed		▁e		d	g	e
5	▁ra			g	g	ed		▁e		d	**ge**	
6	▁ra			g	g	ed		**▁ed**			ge	
7	▁ra			**gg**		ed		▁ed			ge	

(b) Excerpt of the merge list, showing only the relevant merge rules and their priority.

Priority	Merge rule	
42	▁	e
57	e	d
68	r	a
74	a	g
108	▁	r
677	▁	ra
765	g	e
1517	▁e	d
5139	ra	g
9596	g	g
25546	▁ra	g
27817	g	ed

4 Incremental algorithm for re-tokenization

The incremental re-tokenization process, shown in Algorithm 3, is designed to update the tokenized representation of text after modifications. This algorithm operates on three distinct segments: the prefix, the affected text, and the suffix. The prefix consists of all tokens that precede the modified text, similarly the suffix consists of the tokens after the modified text, and the affected text is the part directly impacted by the modification.

In the context of this thesis, the notation '$\wr$' is used to denote the boundaries between tokens in a tokenized text. For example, if a string "abc" is tokenized into two tokens "a" and "bc", it is represented as a $\wr$ bc.

The core of the process is an iterative approach that gradually extends the affected text into the prefix and suffix regions, ensuring that the tokenization boundaries are accurately identified (lines 3-15). In each iteration, the algorithm extends the affected text one token from the prefix and suffix, if available, and then re-tokenizes this updated segment. The process continues until the newly tokenized segment seamlessly aligns with the tokens at the immediate boundaries of the affected text.

The algorithm initializes two index variables, i and j, which point to the end of the prefix and the beginning of the suffix, respectively. In each iteration, it tokenizes a concatenation of the latter part of the prefix (starting from i), the affected text, and the initial part of the suffix (up to j).

The termination condition of the loop is based on the alignment of the newly tokenized segment with the boundaries of the affected text. Specifically, the process concludes when the first token of this segment aligns with the token at the current prefix index or the prefix

Algorithm 3: Incremental Re-Tokenization Process

```
   Input  : The prefix tokens $P$, the affected text $A$, the suffix tokens $S$.
   Output: Updated tokenization $U$.
 1 Let $P = p_1 \wr \cdots \wr p_n$ and $S = s_1 \wr \cdots \wr s_m$;
 2 Initialize $i$ to $n$ and $j$ to 1;
 3 Loop
 4     Let $\omega$ be the concatenation $p_i \cdots p_n A s_1 \cdots s_j$;
 5     Tokenize $\omega$ into $v_1 \wr \cdots \wr v_k$;
 6     if ($v_1 = p_i$ or $i \leq 1$) and ($v_k = s_j$ or $j \geq m$) then
 7         Set $U$ to $p_1 \wr \cdots \wr p_{i-1} \wr v_1 \wr \cdots \wr v_k \wr s_{j+1} \wr \cdots \wr s_m$;
 8         return $U$;
 9     end
10     if $v_1 \neq p_i$ and $i > 1$ then
11         $i \leftarrow i - 1$;
12     end
13     if $v_k \neq s_j$ and $j < m$ then
14         $j \leftarrow j + 1$;
15     end
```

is fully included, and similarly, when the last token aligns with the token at the current suffix index or the suffix is fully included (line 6).

If the first token of the newly tokenized segment does not align with the corresponding prefix token, the algorithm decrements the prefix index i, effectively expanding the tokenization boundary backwards (lines 10-12). Conversely, if the last token of N does not align with the corresponding suffix token, the algorithm increments the suffix index j, expanding the boundary forwards (lines 13-15).

Upon successful alignment, the algorithm constructs the updated list of tokens (U) by merging the relevant segments: the unchanged prefix up to the adjusted index i, the newly tokenized text, and the unchanged suffix starting just after the adjusted index j. The algorithm aims to minimize the computational overhead by localizing re-tokenization to the areas of text directly affected by changes. For a code implementation of Algorithm 3, see Appendix B.

4.1 Correctness of the algorithm

This subsection aims to establish the correctness of the incremental re-tokenization algorithm proposed in Section 4, i.e. Algorithm 3. We address the fundamental question: When a change occurs in a text segment, under what conditions can we confidently re-tokenize only a part of the text, while ensuring that the result is consistent with the "correct" tokenization obtained by fully re-tokenizing the entire text?

BPE processes text in a sequential manner, iteratively merging the most frequent adjacent pairs of symbols (i.e. characters or tokens). This process is deterministic and local; each merge decision depends solely on the immediate neighboring characters or tokens. Consequently, a token's formation is influenced only by its adjacent context. Once a token is formed, its structure remains unchanged unless the characters within it are directly modified.

When a change (such as an insertion, deletion, or substitution) occurs in the text, it immediately affects the tokens encompassing and adjacent to the change. This localized impact is due to the alteration of the character sequences in the vicinity of the change. The 'ripple effect' of this alteration can propagate to nearby tokens, but it is often limited to the area close to where the character sequences are affected. Nonetheless, it is important to recognize that in certain text structures and contexts, even minor changes can extend this effect, potentially influencing the tokenization across a larger portion of the text or even the entire text.

The incremental algorithm exploits the localized nature of BPE's token dependencies. When the algorithm encounters an unchanged token during re-tokenization, it signifies that the local context leading to this token has remained unaltered by the change. This unchanged token thus acts as a reliable indicator, marking a point beyond which the original tokenization structure is preserved. It implies that the character sequences beyond this token, subject to subsequent merge operations, remain the same as in the original text.

Given the unchanged token as a boundary marker, the algorithm can limit re-tokenization to the segment of text up to this token. Since the unchanged token confirms that the preceding merge operations leading to it have occurred identically to the original tokenization process, it follows that the tokenization pattern beyond this token will also proceed as in the original. Thus, by re-tokenizing only up to the unchanged token, the algorithm ensures that the resulting tokens are consistent with what a full re-tokenization would produce.

In summary, the correctness of the incremental re-tokenization algorithm is founded on the principles of BPE's local dependencies and deterministic token formation. The algorithm leverages these principles in the hopes of efficiently updating the tokenization of text segments affected by changes, while ensuring consistency with the overall tokenization pattern of the complete text.

4.2 Incremental re-tokenization example

Table 2 Incremental re-tokenization steps when removing "*un*" from "*An unexceptional sentence.*".

		Tokens							
Before modification		`▁An`	`▁une`	`x`	`cep`	`tional`	`▁sent`	`ence`	`.`
After modification		`▁An`	`▁e`	`x`	`cep`	`tional`	`▁sent`	`ence`	`.`
Step	**Action**								
1	EXTEND		`▁e`						
	TOKENIZE		`▁e`						
2	EXTEND	`▁An`	`▁e`	`x`					
	TOKENIZE	`▁An`	`▁ex`						
3	EXTEND	`▁An`	`▁ex`		`cep`				
	TOKENIZE	`▁An`	`▁excep`						
4	EXTEND	`▁An`	`▁excep`			`tional`			
	TOKENIZE	`▁An`	`▁excep`			`tional`			
After re-tokenization		`▁An`	`▁excep`			`tional`	`▁sent`	`ence`	`.`

To demonstrate the incremental re-tokenization process, the effect of text modification within a text segment is examined through a practical example. Specifically, the focus lies on the modification that changes "*unexceptional*" to "*exceptional*" in the sentence "*An unexceptional sentence.*". This section uses the incremental re-tokenization algorithm to adjust the tokenization of the modified text segment, starting from the affected token and extending outward as necessary. The detailed steps of this process are presented in Table 2, which visually represents the progression from initial tokenization to the final re-tokenized result.

Upon modification, the word "*unexceptional*", previously tokenized into "`▁une` ≀ `x` ≀ `cep` ≀ `tional`", is altered to "*exceptional*", affecting the initial tokenization. The iterative process of extending and re-tokenizing the example sentence can be summarized as follows: (1) The re-tokenization process initiates at the token directly impacted by the change, which is "`▁e`" in this case. Further, the first re-tokenization of the current segment results in "`▁e`", the newly tokenized segment does not align with the pre-modification tokens and the process then continues to the next step. (2) The segment is extended to include adjacent tokens ("`▁An`" and "`x`"), and then re-tokenized.

The process continues extending and re-tokenizing until the newly tokenized segment aligns with the unchanged tokens from the original tokenization. This happens when we re-tokenize the text segment "*An exceptional*" into "`▁An` ≀ `▁excep` ≀ `tional`", where the boundary tokens "`▁An`" and "`tional`" now matches the original tokenization. Consequently, the re-tokenization results in the "correct" tokenization "`▁An` ≀ `▁excep` ≀ `tional` ≀ `▁sent` ≀ `ence` ≀ `.`", which can be verified by tokenizing the modified sentence in full.

Through the incremental re-tokenization process, it is possible to perform a re-tokenization which is localized to the text segment directly influenced by the change with minimal impact on the surrounding text, as shown in Table 2.

5 Experimental Setup

This section details the experimental setup designed to evaluate the performance of the proposed incremental re-tokenization algorithm compared to full text re-tokenization. The focus is on assessing the time efficiency of the algorithm under conditions that simulate real-world text editing scenarios.

The experiments are conducted on a PC running Windows 10, equipped with an AMD Ryzen 3900X 12-Core Processor and 16GB of RAM. For tokenization, a purely BPE-trained SentencePiece model is used: the `m2m100_1.2B` model from Meta's Hugging Face repository (specifically their `sentencepiece.bpe.model` file) [20].

Two text sources are utilized for the experiments: (1) The first 500 bytes of Section 1 (i.e. the introduction of the thesis) is used as a sample of a basic text-snippet. (2) A sequence of 500 randomly generated lowercase characters, without spaces, was created to represent unstructured text scenarios. Each text source starts as a one-byte string, incrementally growing by one byte in each iteration, up to 500 bytes.

For each iteration, a random token in the string is modified by removing a character and shuffling the remaining characters. Three re-tokenization methods are compared: Firstly, (1) a full tokenization using SentencePiece. Secondly, (2) an incremental re-tokenization using SentencePiece with Algorithm 3. And lastly, (3) another incremental re-tokenization but with a custom Python implementation of SentencePiece's tokenizer, included partly to validate Algorithm 2. In addition, the third method (3) represents a worst case scenario in terms of performance, as opposed to the second method (2) which exemplifies something closer to an ideal implementation.

Each method executes 100 times for every modified string, and the fastest run out of the 100 is recorded. (As per the documentation: "In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python's speed, but by other processes interfering with your timing accuracy." [21]) The whole process is repeated with 1000 randomly selected tokens, and the average time across these iterations is calculated for each method and string length. This approach aims to minimize variability due to random factors and provide a robust comparison of the methods.

The average times for each method are compared to assess their relative efficiency. The focus is on determining whether the incremental re-tokenization algorithm can offer a significant performance advantage over a full tokenization.

6 Results

This section presents the experimental results of the re-tokenization methods compared in this study. The results are divided into two separate experiments, each using different text material for the evaluation.

The first experiment utilizes the first 500 bytes of Section 1 as the dataset for re-tokenization. Figure 1 illustrates the time taken for re-tokenization as a function of the length of the text, ranging from 0 to 500 bytes.

The graph shown in Figure 1 indicates that the full re-tokenization method's time increases linearly with the text length. In contrast, both incremental methods exhibit a rel-

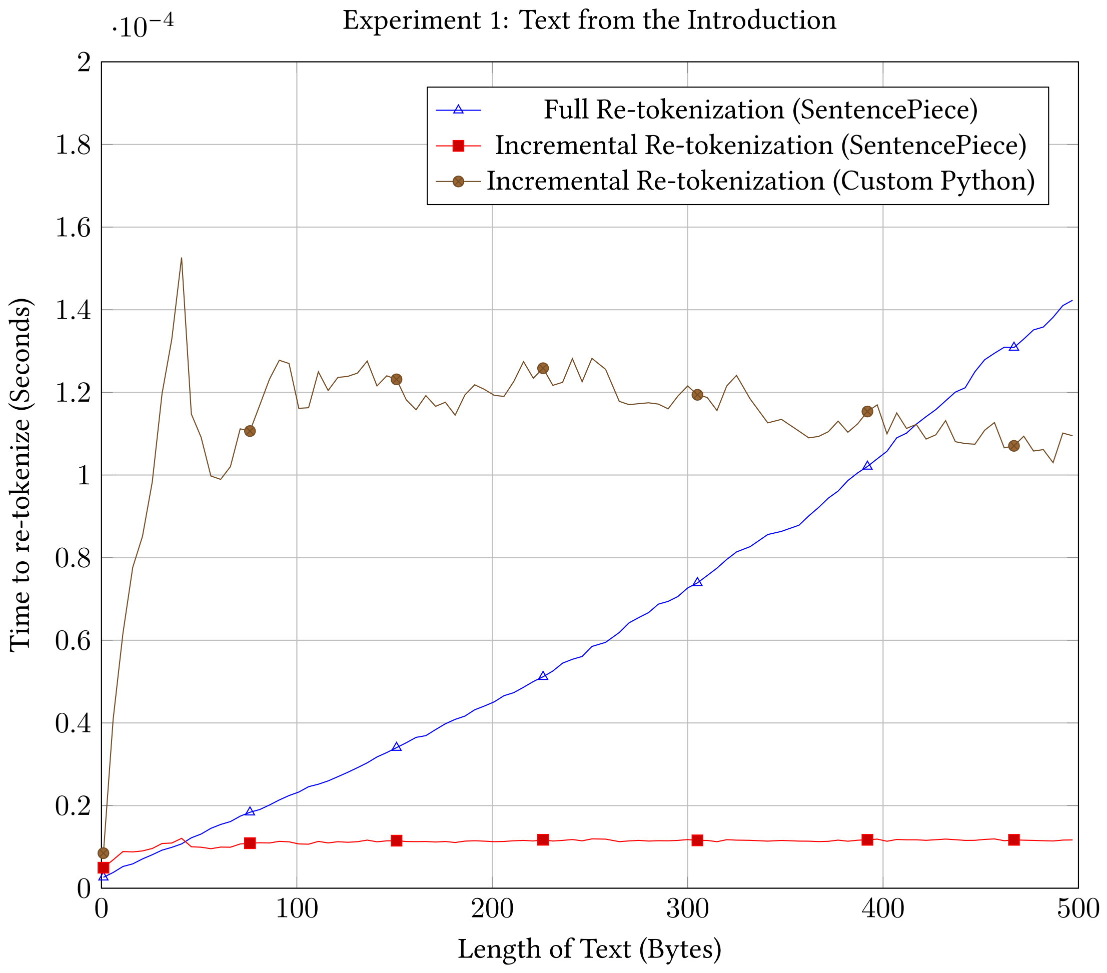


Figure 1: Performance comparison of three re-tokenization methods — full re-tokenization, incremental re-tokenization using SentencePiece, and incremental re-tokenization using a custom Python implementation — applied to the first 500 bytes of the thesis introduction. The graph illustrates the average time taken for re-tokenization as the length of the text increases.

atively constant time across various text lengths, with the custom Python implementation showing slightly fluctuating times.

In the second experiment, a string of 500 random lowercase characters was used to simulate a scenario with unstructured text. The results, as depicted in Figure 2, compare the time efficiency of the same three re-tokenization methods.

Similar to the first experiment, the time required for full re-tokenization using SentencePiece grows in proportion to the text length. The incremental re-tokenization methods, both the custom Python version and the one utilizing SentencePiece, show consistent times across different text lengths, with the custom Python implementation exhibiting minor variability.

Both these experiments show a clear distinction in time performance between the full re-tokenization approach and the incremental methods. The incremental re-tokenization methods demonstrate a significant reduction in the time variability across different text lengths when compared to the full re-tokenization method.

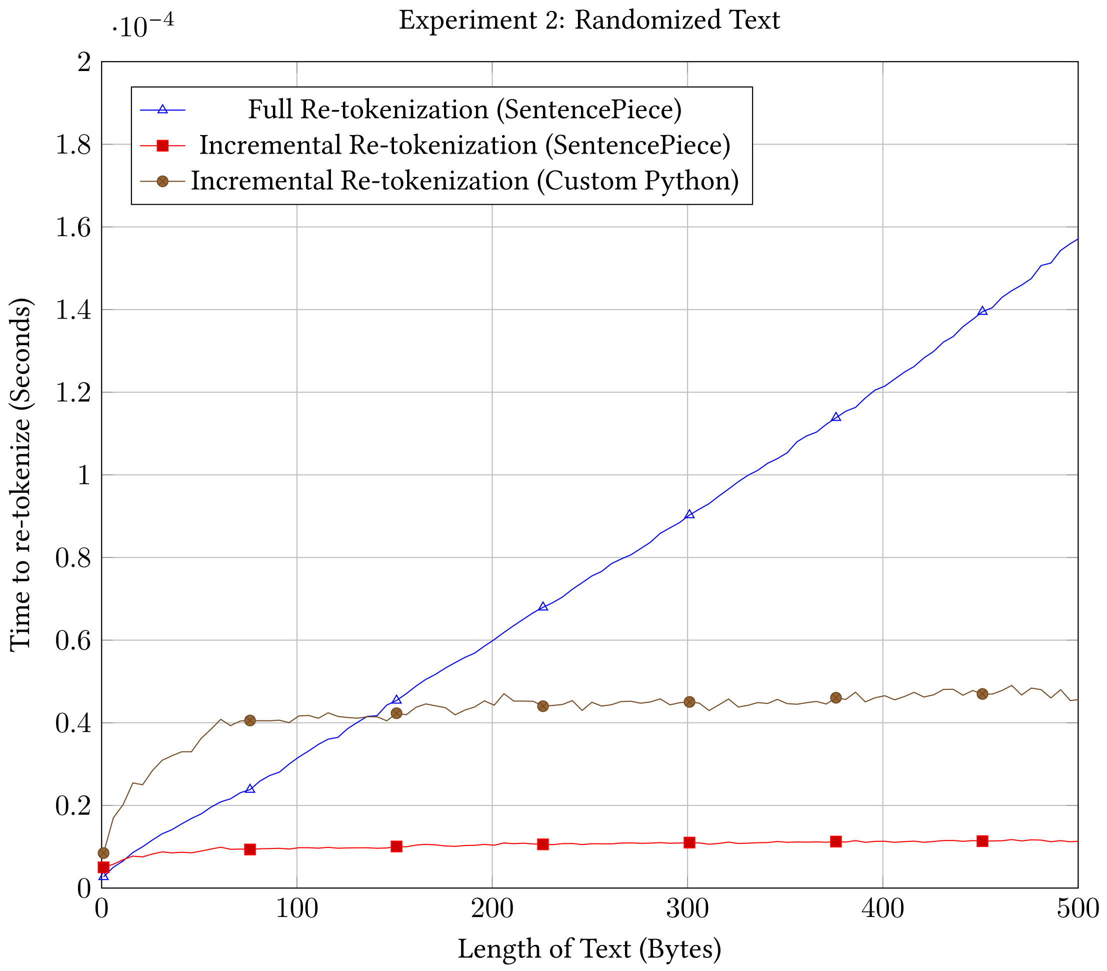


Figure 2: This graph shows the average time taken for re-tokenization of a string of up to 500 random lowercase characters using three different methods: full re-tokenization, incremental re-tokenization with SentencePiece's tokenizer, and incremental re-tokenization with a custom Python implementation. The x-axis represents the length of the text in bytes, highlighting the time efficiency of each method as the text length increases.

7 Discussion

The primary goal of this thesis was to explore the efficiency of an incremental re-tokenization algorithm in updating tokenized text data, especially in the context of BPE-trained SentencePiece models. We wanted to find out whether this algorithm could offer a more efficient alternative to full re-tokenization when a minor change had been made in the text. The research involved developing the algorithm, proving its correctness, analyzing its time complexity, and empirically comparing its performance with that of full re-tokenizations.

The research question was: *Can an incremental algorithm efficiently re-tokenize text after minor changes in BPE trained SentencePiece models without reprocessing the entire text?* The results from the experiments indicate a positive answer. Both implementations of the incremental re-tokenization algorithm consistently showed better time efficiency compared to full re-tokenization. Although, on short texts, i.e. less than about 400-500 bytes, the overhead and repeated tokenizations of some of the tokens may lead to slower execution times for the incremental re-tokenization process compared to a single tokenization on the whole text.

The experimental results validate the hypothesis that incremental re-tokenization can be more time-efficient than full re-tokenization. The full re-tokenization method exhibited a linear increase in time with text length, which is consistent with the expectation that the computational complexity should scale with the size of the dataset. However, it is worth noting that one might expect the time it takes to fully tokenize a string to grow with the complexity $O(n \log n)$ as the number of bytes (n) increases since SentencePiece's tokenizer utilizes a priority queue. But the effect of this appears to be negligible based on the results presented in Section 6. On the other hand, the incremental methods maintained a comparatively constant time, confirming that these methods localize the re-tokenization process to the parts of the text close to the modified parts. This local approach avoids unnecessary computation on unchanged parts of the text, which explains the observed efficiency.

7.1 Impact and Future Work

The incremental re-tokenization method explored in this thesis could enhance the way we interact with text-based technologies. By implementing this method, systems may become slightly faster and require fewer resources. This could in turn lead to smoother interactions with tools like chatbots, making them quicker to respond, or educational software & virtual assistants, where updates to content can be processed more swiftly. It is a step towards making our digital tools more agile and user-friendly, without any noticeable changes in how we use them every day.

In terms of what comes next, future research could delve into developing predictive models to determine the ripple effect of text modifications on token stability. By understanding the reach of these modifications, it may be possible to optimize memory usage, retaining only those tokens that are likely to be affected by changes. Exploratory work could also examine the benefits of varying the breadth of the initial re-tokenization scope and adjusting the expansion strategy during re-tokenization, potentially in asymmetrical ways for the left and right context. Such investigations would not only refine the algorithm's efficiency but could also reveal inherent patterns in how textual changes spread through tokenized data. This could lead to more sophisticated, context-aware algorithms that further streamline text processing tasks.

References

[1] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[3] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," *arXiv preprint arXiv:1808.06226*, 2018.

[4] F. Stollenwerk, "Training and evaluation of a multilingual tokenizer for gpt-sw3," *arXiv preprint arXiv:2304.14780*, 2023.

[5] A. Ekgren, A. C. Gyllensten, F. Stollenwerk, J. Öhman, T. Isbister, E. Gogoulou, F. Carlsson, A. Heiman, J. Casademont, and M. Sahlgren, "Gpt-sw3: An autoregressive language model for the nordic languages," *arXiv preprint arXiv:2305.12987*, 2023.

[6] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[7] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013, pp. 1631–1642.

[8] G. Lample and A. Conneau, "Cross-lingual language model pretraining," *arXiv preprint arXiv:1901.07291*, 2019.

[9] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[10] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[11] M. Schuster and K. Nakajima, "Japanese and korean voice search," in *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP).* IEEE, 2012, pp. 5149–5152.

[12] T. Kudo, "Google/sentencepiece: Unsupervised text tokenizer for neural network-based text generation." accessed: 2023-12-19. [Online]. Available: https://github.com/google/sentencepiece

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[15] H. Björklund, W. Martens, and T. Timm, "Efficient incremental evaluation of succinct regular expressions," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, 2015, pp. 1541–1550.

[16] M. Berglund and B. van der Merwe, "Formalizing bpe tokenization," *arXiv preprint arXiv:2309.08715*, 2023.

[17] P. Gage, "A new algorithm for data compression," *C Users Journal*, vol. 12, no. 2, pp. 23–38, 1994.

[18] T. Kudo, "Subword regularization: Improving neural network translation models with multiple subword candidates," *arXiv preprint arXiv:1804.10959*, 2018.

[19] B. Heinzerling and M. Strube, "Bpemb: Tokenization-free pre-trained subword embeddings in 275 languages," *arXiv preprint arXiv:1710.02187*, 2017.

[20] Meta, "M2m100 1.2b," accessed: 2023-12-19. [Online]. Available: https://huggingface.co/facebook/m2m100_1.2B

[21] Python Software Foundation, "timeit — measure execution time of small code snippets: timeit.timer.repeat," accessed: 2023-12-19. [Online]. Available: https://docs.python.org/3/library/timeit.html#timeit.Timer.repeat

A Tokenizer Implementation (in Python)

The code shown in Listing A.1 implements the encode function for BPE models in SentencePiece. The input string is assumed to be a normalized string.

```python
def my_encode(normalized: str) -> list[str]:
    if not normalized:
        return []

    agenda = []
    symbols = []

    # Inner function to add new symbol pairs to the agenda
    def maybe_add_new_symbol_pair(left, right):
        if left == -1 or right == -1:
            return
        piece = (l:=symbols[left].piece) + (r:=symbols[right].piece)
        piece_id = sp_model.piece_to_id(piece)
        unknown = sp_model.unk_id()
        if piece_id == unknown and any(unknown != sp_model.piece_to_id(p) for p in (l,r)):  # unknown token id
             check
            return
        piece_score = sp_model.get_score(piece_id)
        h = SymbolPair(left, right, piece_score, len(piece))
        heappush(agenda, h)

    # Splits the input into character sequence
    for index in range(len(normalized)):
        s = Symbol(piece=normalized[index])
        s.prev = index - 1
        s.next = index + 1 if index != len(normalized)-1 else -1
        symbols.append(s)

    # Initialize the agenda with symbol pairs
    for i in range(1, len(symbols)):
        maybe_add_new_symbol_pair(i - 1, i)

    # Process the agenda
    while agenda:
        top = heappop(agenda)
        if (not symbols[top.left].piece or not symbols[top.right].piece or
                len(symbols[top.left].piece) + len(symbols[top.right].piece) != top.size):
            continue

        symbols[top.left].piece += symbols[top.right].piece
        symbols[top.left].next = symbols[top.right].next
        if symbols[top.left].next >= 0:
            symbols[symbols[top.left].next].prev = top.left  # the merged symbol is now unlinked
        symbols[top.right].piece = ""  # symbol is now fully moved/merged

        maybe_add_new_symbol_pair(symbols[top.left].prev, top.left)  # try to add: 'symbol before top' + 'top'
        maybe_add_new_symbol_pair(top.left, symbols[top.left].next)  # try to add: 'top' + 'symbol after top'

    output = []
    index = 0
    while index != -1:
        if symbols[index].piece:
            output.append(symbols[index].piece)
        index = symbols[index].next
    return output
```

Listing A.1: Python implementation of SentencePiece's encode function (for BPE models).

B Incremental Re-Tokenization Implementation

Listing B.1 presents a Python implementation of Algorithm 3.

```python
def update(prefixes: list[str], affected_text: str, suffixes: list[str]) -> list[str]:
    P = prefixes or []
    S = suffixes or []
    n, m, = len(P)-1, len(S)-1
    i, j = n, 0

    while True:
        new_tokens = tokenize(''.join( P[i:] + [affected_text] + S[:j+1] ))

        if (new_tokens[:1] == P[i:i+1] or i <= 0) and (new_tokens[-1:] == S[j:j+1] or j >= m):
            return P[:i] + new_tokens + S[j+1:]

        if new_tokens[:1] != P[i:i+1] and i > 0:
            i -= 1
        if new_tokens[-1:] != S[j:j+1] and j < m:
            j += 1
```

Listing B.1: This `update`-function incrementally re-tokenizes a given text segment, utilizing specified prefix and suffix tokens. The resulting tokens will match the tokens obtained from a tokenization of the full text.

UMEÅ UNIVERSITY