M.Sc. Computerlinguistik
Center for information and language processing (CIS)
Faculty of language and literature sciences
Ludwig-Maximiliams-Universität München

Master thesis

The effects of word segmentation quality on word alignments

Ane Berasategi
Matriculation number: 12006250

Supervisor: Masoud Jalili Sabet
Supervisor: Prof Dr.Hinrich Schütze
Submitted on: 2020 August

Abstract

Historically, many ways have been used to feed language to machines. Separating sentences, separating words, characters, each has its own advantages and drawbacks. In recent times, subword units have gained popularity, as a middle point between words and characters. However, this is based on the assumption that sentences are separated into words, that is, that space is a separating factor. This thesis explores the research done until the time of writing, describing various tokenization methods, replication of BPE and BPE dropout results, and also incorporates the possibility to create BPE units without spaces, as well as an improvement over the original BPE algorithm.

Declaration by the candidate

I hereby declare that this thesis is my own work and effort and that it has not been submitted anywhere for any award. Where other sources of information have been used, they have been marked.

The work has not been presented in the same or a similar form to any other testing authority and has not been made public.

I hereby also entitle a right of use (free of charge, not limited locally and for an indefinite period of time) that my thesis can be duplicated, saved and archived by the Ludwig Maximilians University (LMU) or any commissioned third party (e. g. *iParadigms Europe Limited*, provider of the plagiarism-detection service "Turnitin") exclusively in order to check it for plagiarism and to optimize the appraisal of results.

Munich, August 19, 2020

Acknowledgements

The development of this thesis was hit by the COVID-19 pandemic, reclusing me in my hometown and depriving me from face to face contact with my supervisor and exchange of ideas with other students. Writing this thesis has been a more solitary endeavor than what I would have hoped, but it still was grounded in the support of many people.

First of all, my biggest appreciation goes to my supervisor Masoud, with whom I had weekly video chats to discuss the progress, brainstorm ideas and discuss next steps. He has always been very helpful and attentive, and especially empathetic with my confinement when he displayed colorful and paradisiac beaches as his background, to make me jealous. Despite his many other obligations he has found time for me, and provided guidance, direction or resources. It has been a pleasure to work with you.

On the other side, my family has been always been a pillar for me, and more so during these difficult times. They have provided distraction, entertainment, cheering up, sharing ideas, progress or lack thereof, and helping me through the so common situation when a bug torments you for days or weeks, appearing in your nightmares, making you obsessed until people around you start worrying if you are having a breakdown. They ask you to explain it, and talking about it to someone outside of circle of competence makes you explain it in an alternative way, and they ask questions that make you think about the problem in a completely different way, and at some point you think: maybe I could quickly try this, wait a second...

Last but not least, friends and loved ones have also provided great support. Thank you for the regular checkups, early draft reviews, ranting opportunities, the burgers in the park. You have given me a hand to hold, things to do, a room to sit in surrounded by my favourite view.

Contents

List of Figures

1 Introduction

Natural language processing, NLP, is the task of processing natural language, transforming it, obtaining information from it, classifying it, translating it, and many other tasks. It is closely related to NLU, natural language understanding, which seeks to more theoretically understand the meaning of language. It could be said that linguistics is more involved in NLU, whereas computer science, algorithms and programming are the backbone of NLP.

There has been great progress in NLP since around 2017 until the time of writing, 2020, mostly driven by computational advances. Pre-2017 NLP mostly used convolutional or recurrent neural networks, built them from scratch with hundreds of thousands of hyperparameters and trained them on CPUs locally. The datasets used for these models were comparable in size with the models. In 2017, Vaswani et al [1] introduced a model neural network architecture, Transformers, that removed any sequentiallity and focused solely on attention. This architecture is very powerful and soon thereafter, BERT [2] was introduced, which enabled the pre-training of deep bidirectional Transformers. BERT is very computationally expensive and requires intensive training, which is why the authors published pre-trained weights on many NLP tasks that users could later fine tune for their own applications. This introduced the era of pre-training, when users do not train models from scratch anymore and use huge, already pre-trained models. Anyone who wishes to train state-of-the-art models at the time of writing requires very powerful GPUs or even TPUs, which are only available to large organizations given their enormous cost, and several days or even weeks of training, which produces a gigantic energy consumption, as a study from 2019 suggests [3]. Bigger datasets, bigger models, better embeddings have been trained and published, and they are more power hungry every year. Neural networks, especially the ones being used nowadays, given their size, require vast amounts of data in order to process and adapt their billions of hyperparameters.

It could be said that most of the advances in the past 3-5 years have been computational, and not so much linguistic. The deep learning pipeline has barely changed. First of all, a corpus is read and parsed. Then, embeddings are applied to the parsed corpus and fed to the neural network. This in turn is trained for a number of epochs, the weights typically saved for posteriority, and then the model is evaluated against a test set. Independently of the size or architecture of the neural network, the training time and the relatively new addition of embeddings, the main idea of machine learning has not changed.

This thesis does not introduce any new parameter in the main pipeline, but rather focuses on the corpus-embeddings step and explores how the corpus is parsed so that the embeddings are optimally applied to it. The definition and usage of embeddings is explained in the literature review section of this thesis, but in broad terms, embeddings are a mapping between characters and numbers. Computers cannot work with pictures, audio or language, they only process numbers, which is why for all data types, they first need to be transformed into numbers. How to make this transformation depends at great extent on the type of data: for instance, sequentiallity is very important in audio and language. An utterance or word generally loses its meaning when it is displaced in time. In images, however, the important metric is knowing which pixels are close to which pixels in order to get a meaning out of a group of pixels. For language, the numbers fed to the computer should not just be numbers: they should encode meaning. Embeddings map any text to a fixed size vector, say 50 numbers one after another, in a way that the numbers of similar words are also very similar. The embedding vector of gray and black are therefore very similar.

Historically, embeddings have been applied to words. That is, the word *black* has the same embedding vector regardless of the context, where it might be used to talk about colors, the sky, or race. This creates some problems since words can have different meanings depending on the context. After BERT, the chosen method has been subword embeddings, applying the mapping to parts of the word instead of the whole. For example, the word *extraterrestrial* would be first segmented into *extra-terrestr-ial*, and each unit would get assigned a different embedding. There have been many methods to obtain this segmentation, the most popular one being BPE [4].

This thesis has explores he details of the BPE algorithm, coded it from scratch to deeper understand it, and replicated its results. Some improvements have been done to parts of the algorithm used in the original paper, which are presented in this thesis. A newer version of BPE, published in 2019, has also been explored, analyzed, coded and its results replicated. Some of the hyperparameters used in this method have been tweaked and its performance slightly improved. BPE-dropout [5] improves the performance of BPE, which has been shown in this thesis for English-German and also for other language pairs, confirming its consistency. Moreover, the most novel addition of this thesis, is applying BPE without any word boundaries.

In English and Indo-European languages, the smallest unit in language that preserves meaning is the word. Given a sentence with 5 words, it is possible to assign one meaning to each word, and obtain the meaning of the sentence by uniting the meanings of the words. But in other languages where not words, but symbols are used, the whitespace between words does not have the same defining meaning as in Indo-European languages, and perhaps it would make sense to apply embeddings to a group of whitespace-separated symbols.

As a result, this thesis extends BPE to perform also in the case where multiple-word units are allowed. The evaluation method chosen to observe the performance of BPE is not prepared to deal with multi-word units, and the thesis shows that the performance is lower in this case. However, for a specific combination of hyperparameters, the results are comparable to those of BPE using whitespace as word boundary. Finally, it is also shown that BPE-dropout outperforms BPE also in this new case of no word boundaries.

2 Goals of the thesis

The goal of this thesis has been primarily to analyse the BPE algorithm, as a way to tokenize raw text and as opposed to other tokenization methods such as word tokenization, or other subword tokenizations. After reading this thesis the reader should have no doubts as to how the BPE algorithm works, given the extensive array of examples and codes that are presented throughout the thesis. This thesis also aims to replicate the BPE algorithm and obtain satisfactory performance results.

In order to evaluate the performance of the BPE algorithm, an alignment method from statistical machine translation is transferred into this scenario, to compare the difference between aligning words and subword units.

An additional goal is to analyse, understand and replicate an improvement over BPE called BPE dropout. The intricancies of the algorithm improvement, the motivation stemming from some of BPE's drawbacks will be explained, and examples and codes included as well. By iterating on some of the hyperparameters used to obtain BPE dropout results, this thesis will atempt to improve BPE dropout's results.

This thesis will also intend to improve the original BPE learning algorithm, making a dramatic improvement in performance. Furthermore, a new paradigm to obtain BPE units will be presented, namely removing the space boundary between words to admit creating BPE units among different words. This will be integrated into the algorithm, so that the end user can choose which mode to use, either the traditional space separated version employed in the BPE and BPE dropout paper, or the new one. It will be proven that BPE-dropout outperforms BPE even in this no-space case, confirming the general improvement of BPE-dropout.

The pipeline will be automated and easy to tweak in the sense that by changing some parameters in a global file, such as dropout rate, number of merges, space mode on or off, etc. the pipeline will automatically adapt to the specific scenario.

3 Literature review

This chapter outlines the literature review for this thesis. It contains basic explanations of the concepts explained, the historical development, graphs and figures, examples, different variants and comparisons among them, and references to all papers and articles mentioned. The chapter is divided into two sections, namely tokenization and translation. The topics handled in each section are independent, but come together in the development of this thesis.

3.1 Tokenization

3.1.1 Introduction

Tokenization is the first major step in language processing. The main idea is simplifying or compressing the input text into meaningful units, called tokens, creating a big vocabulary of tokens and shorter sequences, as illustrated in Figure 3.1. [6]

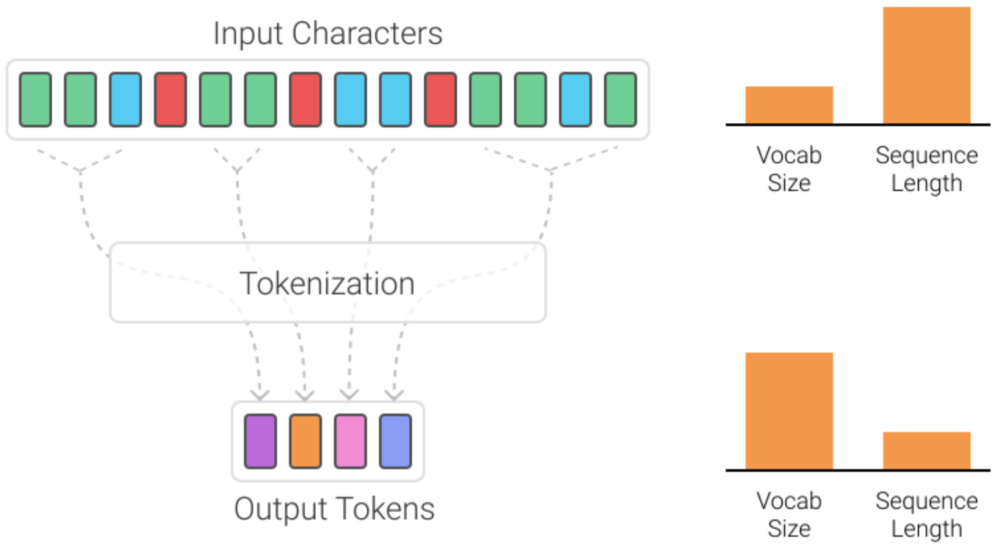


Figure 3.1: Tokenization of a sequence of text

Tokens in language are defined as units which have a semantic meaning, be it words, phrases, symbols or other elements. Here is an example of a simple way to tokenize text:

Raw text: I ate a burger today, and it was good.
Text after tokenization: ['I', 'ate', 'a', 'burger', 'today', 'and', 'it', 'was', 'good']

In this example, the tokenization process is simple. First, locate the word boundaries and split words by whitespaces. Next, remove symbols and punctuation marks as both contain no definitive meaning. However, tokenization in real life is not always that easy: some punctuation marks are relevant to the meaning of the words around them, for example *don't*: should it be tokenized as *do, n't* or *don, ', t*? There is no correct answer, different libraries or applications take a differente approach.

How to deal with punctuation marks?

Every language has its challenges. In English for example, there are possessors such as *aren't*, and contractions such as *Sarah's* and *O'Neill.* Because of this, it is imperative to know the language of the input text. *Language identification* is the task of identifying the language of the input text. Methods ranging from the initial k-gram algorithms used in cryptography (Konnheim, 1981), to more modern n-gram methods (Dunning, 1994) are commonly used. Once the language is identified, we can follow the rules for each case and deal with punctuation marks appropriately.

Other types of tokens

In the simple example above, tokens were words. Alternatively, tokens can be groups of words, characters or subwords (parts of a word). For example, take the word *smarter*:

- Sentence: the smarter computer
- Word tokens: the, smarter, computer
- Character tokens: t, h, e, s, m, a, r, t, e, r, c, o, m, p, u, t, e, r
- Subword tokens: the, smart, er, comput, er
- Subword tokens without word boundaries: the smart, er comput, er

The major question in the tokenization phase is: **what are the correct tokens to use?**. The following section explores these 4 types of tokenization methods and delves into the algorithms and code libraries available.

3.1.2 Tokenization algorithm types

The tokenization method depends heavily on the targeted application. This results in different applications requiring different tokenization algorithms. Nowadays, most deep learning architectures in NLP process raw text at the token level and as a first step, create embeddings for these tokens, which will be explained in more detail in the following section. In short, *the type of tokenization depends on the type of embedding.* Advantages and drawbacks of several tokenization methods are further explained in the following sections.

Word level tokenization

Word level tokenization is the first established type of tokenization. It is the most basic and also the most common form of tokenization. It splits a piece of text into individual words based on word boundaries; usually a specific delimiter consisting mostly of whitespace ' ' or other punctuation signs.

Conceptually, splitting on whitespace can also split an element which should be regarded as a single token, for example New York. This is mostly the case with names, borrowed foreign phrases, and compounds that are sometimes written as multiple words. Tokenization without word boundaries aims to address that problem. 3.1.2 on page 15

Word level algorithms

The simplest way to obtain word level tokenization is by splitting the sentence on the desired delimeter; most commonly this is whitespace. The `sentence.split()` function in Python or a Regex command `re.findall("[\w’]+", text)` achieves this in a simple way.

The natural language toolkit (NLTK) in Python provides a tokenize package which includes a *word_tokenize* function. The user can provide the language of the text, whereby if none is given, English is taken as default.

```
from nltk.tokenize import word_tokenize
sentence = u'I spent $2 yesterday'
sentence_tokenized = word_tokenize(sentence, language='English')
>>> sentence_tokenized = ['I', 'spent', '$', '2', 'yesterday']
```

Comparatively, SpaCy offers a similar functionality. It is possible to load the language model for different languages and model size. In this case, the English language (en) and small model size (sm) was loaded.

```
import spacy
sp = spacy.load('en_core_web_sm')
sentence = u'I spent $2 yesterday'
sentence_tokenized = sp(sentence)
>>> sentence_tokenized = ['I', 'spent', '$', '2', 'yesterday']
```

Other word level tokenization functions include Keras:

```
from keras.preprocessing.text import text_to_word_sequence
sentence_tokenized = text_to_word_sequence(sentence)
```

And Gensim:

```
from gensim.utils import tokenize
sentence_tokenized = list(tokenize(sentence))
```

Depending on the target application and framework, one might favor an algorithm over the other.

Word embeddings

As stated before, the goal of tokenization is to split the text into units with meaning. Typically, each token is assigned an embedding vector. Word2vec (Mikolov et al., 2013 [7]) is a way of transforming a word into a fixed-size vector representation, as shown in Figure 3.2.

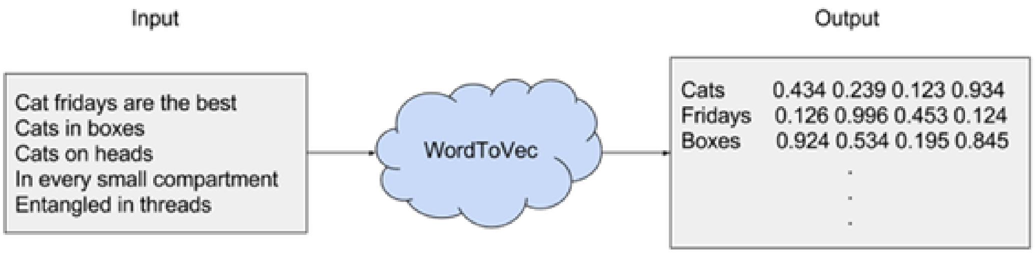

Input
Output
Cat fridays are the best
Cats in boxes
Cats on heads
In every small compartment
Entangled in threads
WordToVec
Cats 0.434 0.239 0.123 0.934
Fridays 0.126 0.996 0.453 0.124
Boxes 0.924 0.534 0.195 0.845

Figure 3.2: Representation of word embeddings

Apart from word2vec, there are other word embedding algorithms, namely *GloVe* or *fasttext*. When words are translated into a multi-dimensional (N) plane, each word can be compared relative to another.

As such, words with similar context will appear closer to one another. Here is a simplified example to illustrate the concept of word embeddings.

- Word: smart. Embedding: [2, 3, 1, 4]
- Word: intelligent. Embedding: [2, 3, 2, 3]
- Word: stupid. Embedding: [-2, -4, -1, -3]

In the example, the embeddings of *smart* and *intelligent* have a distance of 2, since the last two numbers in the vector differ by one respectively. If this was plotted in a four dimensional space, these words would be very close together. On the other hand, *stupid* is almost the opposite of *smart*. The distance in this case is much larger. In the plot, these words would sit roughly in opposite directions. Thus, with word embeddings, a sentence is transformed into a sequence of embedding vectors, which is very useful for NLP tasks.

Word level tokenization drawbacks

Word embeddings have some drawbacks. In many cases, a word can have more than one meaning: *well*, for example, can be used in these two scenarios.

> I'm doing quite well.
> The well was full of water.

In the first case, *well* is an adverb and in the second it is a noun. *well*'s embedding will probably be a mixture of the two, since word embeddings do not generalize to **homonyms**. Consequently, the true meaning of both words cannot be represented.

Another drawback is that word embeddings are not well equipped to deal with **out of vocabulary (oov) words**. Word embeddings are created based on limited vocabulary size known to the system. If a foreign or misspelled word is detected, it will be given a universal unknown <UNK> embedding, that will be the same for all unknown words. Therefore, all unknown words in NLP will be treated similarly as if they have the same meaning. The information within these words is lost due to the mapping from OOV to UNK.

Another issue with word tokens is the huge **vocabulary size**. Generally, pre-trained models are trained on a large volume of the text corpus. As such, if the vocabulary is built with all the unique words in such a large corpus, it creates a huge vocabulary. A current state-of-the-art deep learning architecture, Transformer XL [8], has a vocabulary size of 267,735. This opens the door to *character tokenization*, since in this case the vocabulary depends on the number of characters, which is significantly lower than the number of all different words.

These problems are not to be mistaken with tokenization problems, tokenization is merely a way to an end. In most cases however, they are used to create embeddings. And if embeddings from word tokens have drawbacks, the tokenization method is changed in order to create different tokens, in order to create other types of embeddings.

Character level tokenization

In this type of tokenization, instead of splitting a text into words, the splitting is done into characters, whereby *smarter* becomes *s-m-a-r-t-e-r* for instance. Karpathy, 2015 was the first to introduce a character level language model.

OOV words, misspellings or rare words are handled better, since they are broken down into characters and these characters are usually known in the vocabulary. In addition, the size of the vocabulary is significantly lower, namely 26 in the simple case where only the English characters are considered, though one might as well include all ASCII characters. Zhang et al. (2015) [9], who introduced the character CNN, consider all the alphanumeric character, in addition to punctuation marks and some special symbols.

Character level models are unrestricted in their vocabulary and see the input "as-is". Since the vocabulary is much lower, the model's performance is much better than in the word tokens case. Tokenizing sequences at the character level has shown some impressive results.

Radford et al. (2017) [10] from OpenAI showed that character level models can capture the semantic properties of text. Kalchbrenner et al. (2016) [11] from Deepmind and Leet et al. (2017) [12] both demonstrated translation at the character level. These are particularly compelling results as the task of translation captures the semantic understanding of the underlying text.

Character level algorithms

The previous libraries explored in the case of word tokenization (native python libraries, nltk, spacy, keras) have their own version for character level tokenization.

Character level tokenization drawbacks

When tokenizing a text at the character level, the sequences are longer, which takes longer to compute since the neural network needs to have significantly more parameters to allow the model to perform the conceptual grouping internally, instead of being handed the groups from the beginning.

It becomes challenging to learn the relationship between the characters to form meaningful words and, given that there is no semantic information among characters, characters are semantically void. This makes it complicated to generate character embeddings.

Sometimes the NLP task does not need processing at the character level, such as when doing a sequence tagging task or name entity recognition, the character level model will output characters, which requires post processing.

As an in-betweener between word and character tokenization, subword tokenization produces subword units, smaller than words but bigger than just characters.

Subword level tokenization

Subword tokenization is the task of splitting the text into subwords or n-gram characters. These algorithms leave most common words as they are, but they decompose rare words in meaningful subword units. The word *unfriendly* might be considered a rare word and decomposed as *un-friend-ly*. Words like *lower* might be segmented as *low-er*, *smartest* as *smart-est*, and so on. In the event of an OOV word such as *greoter*, this tokenizer will divide it into *greot-er* and effectivey obtain some semantic information. Very common subwords such as *ing*, *ion*, usually with a morphological sense, are learnt through repetition. This is especially useful in agglutinative languages such as Turkish, where you can form (almost) arbitrarily long complex words by stringing together some subwords.

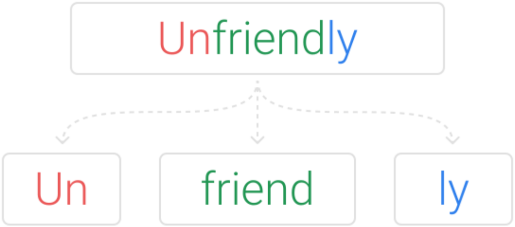


Figure 3.3: Representation of the word 'unfriendly' in subword units

At the time of writing (2020), the most powerful deep learning architectures are based on Transformers (Vaswani et al., 2017 [1]), and these rely on subword tokenization algorithms to prepare the vocabulary. BERT [2] makes the following tokenization. The ## symbol means that the rest of the token should be attached to the previous one, without space, for the case when we might need to decode predictions and reverse the tokenization.

Raw text: I have a new GPU.
Text after tokenization: ['i', 'have', 'a', 'new', 'gp', '##u', '.']

Subword level algorithms

Since Transformers are a relatively new architecture in 2020, subword tokenization is an active area of research. Nowadays four algorithms stand out: byte-pair encoding (BPE), unigram LM, WordPiece and SentencePiece.

Since BPE is the basis of the thesis, it will be explained in depth in the following section. A simple explanation of BPE and the rest of the algorithms follow below.

Huggingface, an open source NLP company, released Transformers and Tokenizers (Wolf et al., 2019 [13]), two popular NLP framework which include several subword tokenizers such as *ByteLevelBPETokenizer*, *CharBPETokenizer*, *SentencePieceBPETokenizer* and *BertWordPieceTokenizer*. The first refer to the first subword level algorithm, BPE, in addition to WordPiece and SentencePiece.

BPE

BPE (Sennrich et al., 2016 [4]) merges the most frequently occurring character or character sequences iteratively. This is roughly how the algorithm works:

1. Get a large enough corpus.
2. Define a desired subword vocabulary size.
3. Split word to sequence of characters and append a special token showing the beginning-of-word or end-of-word affix/suffix respectively.
4. Calculate pairs of sequences in the text and their frequencies. For example, ('t', 'h') has frequency X, ('h', 'e') has frequency Y.

5. Generate a new subword according to the pairs of sequences that occurs most frequently. For example, if ('t', 'h') has the highest frequency in the set of pairs, the new subword unit would become 'th'.

6. Repeat from step 3 until reaching subword vocabulary size (defined in step 2) or the next highest frequency pair is 1. Following the example, ('t', 'h') would be replaced by 'th' in the corpus, the pairs calculated again, the most frequent pair obtained again, and merged again.

BPE is based on a greedy and deterministic symbol replacement and can not provide multiple segmentations.

Unigram LM

Unigram language modelling (Kudo, 2018 [14]) is based on the assumption that all subword occurrences are independent and therefore subword sequences are produced by the product of subword occurrence probabilities. These are the steps of the algorithm:

1. Get a large enough corpus.

2. Define a desired subword vocabulary size.

3. Optimize the probability of word occurrence by giving a word sequence.

4. Compute the loss of each subword.

5. Sort the symbol by loss and keep top X % of word (X=80% for example). To avoid OOV instances, character level is recommended to be included as a subset of subwords.

6. Repeat step 3–5 until reaching the subword vocabulary size (defined in step 2) or there are no changes (step 5).

At a given step, Unigram LM computes a loss from the corpus and the current vocabulary. Then, for each subword, it evaluates how much the loss would augment if the subword was removed from the vocabulary. It sorts the subwords by this quantity (that represents how worse the loss becomes if the token is removed) and removes all the worst p tokens (for instance p could be 10% or 20%). It then repeats the process until the vocabulary has reached the desired size, always keeping the base characters, to be able to tokenize any word written with them, like BPE or WordPiece. Contrary to BPE and WordPiece that work out rules in a certain order that can then be applied in the same order when tokenizing new text, Unigram LM has several ways of tokenizing a new text.

Kudo argues that the unigram LM model is more flexible than BPE because it is based on a probabilistic LM and can output multiple segmentations with their probabilities. Instead of starting with a group of base symbols and learning merges with some rule, like BPE or WordPiece, it starts from a large vocabulary (for instance, all pretokenized words and the most common substrings) that it will trim down progressively. It is not used directly for any of the pretrained models in the library, but it is used in conjunction with SentencePiece, explained below.

WordPiece

WordPiece (Schuster and Nakajima, 2012 [15]) was initially used to solve Japanese and Korean voice problem, and is used in BERT [2]. It is similar to BPE in many ways, except that it forms a new subword based on likelihood, not on the next highest frequency pair. These are the steps of the algorithm:

1. Get a large enough corpus.
2. Define a desired subword vocabulary size.
3. Split word to sequence of characters.
4. Initialize the vocabulary with all the characters in the text.
5. Build a language model based on the vocabulary.
6. Generate a new subword unit by combining two units out of the current vocabulary to increment the vocabulary by one. Choose the new subword unit out of all the possibilities that increases the likelihood on the training data the most when added to the model.
7. Repeat step 5 until reaching subword vocabulary size (defined in step 2) or the likelihood increase falls below a certain threshold.

WordPiece and BPE only differ in step 6, since BPE merges the token combination that has the maximum frequency. This frequency stems from the combination of the tokens and not previous individual tokens. In WordPiece, the frequency of the two tokens are separately taken into account. If there are 2 tokens A and B, the score of this combination will be the following:

Score(A,B) = Frequency(A,B) / Frequency(A) * Frequency(B)

The token pair with the highest score will be selected. It might be the case that *Frequency('so', 'on')* is very high but their individual frequencies are also high. Hence with the WordPiece algorithm, 'soon' will not be merged as the overall score is low. In another example, *Frequency('Jag','gery')* might be low but if their individual frequencies are also low, 'Jag' and 'gery' might be joined to form 'Jaggery'. WordPiece evaluates what it loses by merging two symbols and makes sure it is worth it. BERT (Devlin et al., 2018 [2]) uses WordPiece as its tokenization method, yet the precise tokenization algorithm and/or code has not been made public. This example shows the tokenization step and how it handles OOV words.

original tokens = ["John", "Johanson", "'s", "house"]
bert tokens = ["[CLS]", "john", "johan", "##son", "'", "s", "house", "[SEP]"]

SentencePiece

All the tokenization methods so far required some form of pretokenization, which constitutes a problem, since not all languages use spaces to separate words. This is a problem that the architecture XLM [16] solves by using specific pretokenizers for each of those languages, Chinese, Japanese and Thai for instance. To solve this problem, SentencePiece (Kudo et al. 2018 [17]) treats the input as a raw stream, includes the space in the set of characters to use, then uses BPE or unigram LM to construct the appropriate vocabulary. It has an extensive Github repository with freely available code.

As the repository states, it is an unsupervised text tokenizer and detokenizer where the vocabulary size is predetermined prior to the neural model training. It implements subword units (e.g., BPE 3.1.2) and unigram LM 3.1.2) with the extension of direct training from raw sentences. It does not depend on language-specific pre or post-processing.

While conceptually similar to BPE, it does not use the greedy encoding strategy, thus achieving higher quality tokenization while reducing error induced by location-dependent factors as seen in BPE. SentencePiece sees ambiguity in character grouping as a source of regularization for downstream models during training, and uses a simple language model to evaluate the most likely character groupings instead of greedily picking the longest recognized strings like BPE does.

Approaching ambiguity in text as a regularization parameter for downstream models results in higher tokenization quality but adversely reduces the performance of the pipeline, at times making it the slowest part or bottleneck of an NLP system. While the assumption of ambiguity in tokenization seems natural, it appears the performance trade-off is not worth it, as Google itself opted not to use this strategy in their BERT language model. 3.1.2

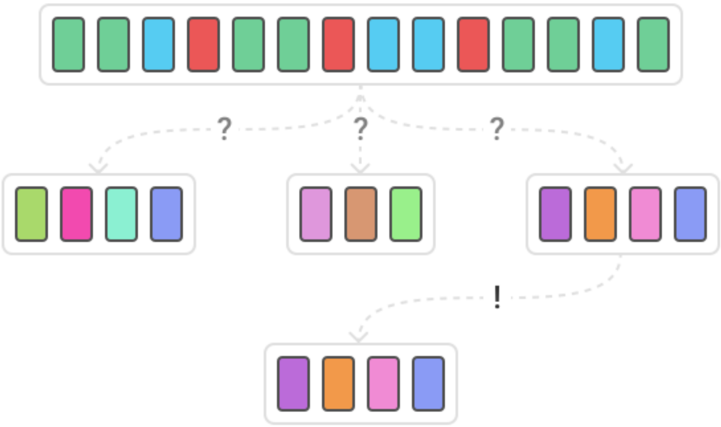


Figure 3.4: Representation of the SentencePiece tokenization in a sequence of text

The number of unique tokens in SentencePiece is predetermined, the segmentation model is trained such that the final vocabulary size is fixed, e.g., 8k, 16k, or 32k. This is different from BPE (Sennrich et al., 2015 [4]) which uses the number of merge operations instead. The number of merge operations is a BPE-specific parameter and not applicable to other segmentation algorithms, including unigram, word and character level algorithms. Most current deep learning architectures, such as ALBERT [18] or XLNet [19] use SentencePiece with Unigram LM.

Tokenization without word boundaries

Another type of tokenization, beyond word, character or subword, is tokenization without word boundaries. The three types of tokenization explored until now cannot create units among words, that is, they consider words separately.

When dealing with languages that do not include space tokenization, such as several Asian languages, an individual symbol can resemble a syllable rather than a word or letter. Most words are short (the most common length is 2 characters), and given the lack of standardization of word breaks in the writing system or lack of punctuation in certain languages, it is not always clear where word boundaries should be placed. As an example, in English:

Input sentence: the smarter computer
Subword tokens without word boundaries: the smart, er comput, er

An approach to handle this has been to abandon word-based indexing, and do all indexing from just short subsequences of characters (character n-grams), regardless of whether particular sequences cross word boundaries or not. Hence, at times, each character used is taken as a token in Chinese tokenization.

3.1.3 BPE

Byte Pair Encoding (BPE) (Sennrich et al., 2015 [4]), is a widely used tokenization method among Transformer-based models. The code is open source and there is an active repository on Github. It merges the most frequently occurring character or character sequences iteratively.

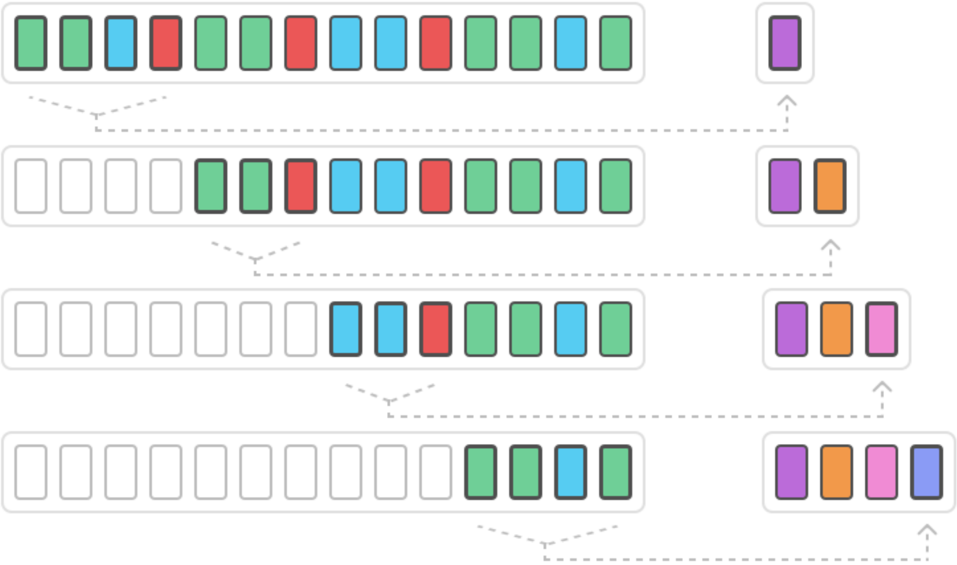


Figure 3.5: Representation of the BPE tokenization in a sequence of text

BPE enables the encoding of rare or OOV words with appropriate subword tokenization without introducing any 'unknown' tokens. One of the performance aspects in tokenization is the length of output sequences. Here, BPE is superior as it produces shorter sequences compared to character tokenization.

Minimal algorithm to learn BPE segmentations

Subsection 3.1.2 showed a simple algorithm to build subword units. In this section it will be explained in depth with an example. These are the steps of the algorithm:

1. Get a large enough corpus.

2. Define a desired subword vocabulary size.

3. Split word to sequence of characters and append a special token showing the beginning-of-word or end-of-word affix/suffix respectively.

4. Calculate pairs of sequences in the text and their frequencies.

5. Generate a new subword according to the pairs of sequences that occurs most frequently, and save it to the vocabulary.

6. Merge the most frequent pair in corpus.

7. Repeat from step 4 until reaching subword vocabulary size (defined in step 2) or the next highest frequency pair is 1.

Considering a simple corpus with a single line, and a desired subword vocabulary size of 10. The character '_' marks the beginning of each word. The following code shows the steps 1-3.

```
def read_corpus(corpus):
    tokens = [("_" + " ".join(token)) for token in corpus]
    return tokens

corpus = [’this is this.’]
vocab_size = 10
tokens = read_corpus(corpus)
>>> tokens = [’_t h i s _i s _t h i s .’]
```

Now we can calculate the pairs of characters and their frequencies, as well as the most popular pair. These are the steps 4-5 in the algorithm above.

```
from collections import Counter

def get_stats(tokens):
  pairs = Counter()
  for sent in tokens:
    for word in sent[1:].split(’ _’):
      symbols = (’_’ + word).split()
      for j in range(len(symbols) - 1):
        pairs[symbols[j], symbols[j+1]] += 1
  return pairs

pairs = get_stats(tokens)
>>> pairs = Counter({(’_t’, ’h’): 2, (’h’, ’i’): 2, (’i’, ’s’): 2,
                    (’_i’, ’s’): 1, (’i’, ’s,’): 1, (’_’, ’i’): 1, (’_i’, ’t’): 1})

most_frequent_pair = pairs.most_common(1)[0][0]
>>> most_frequent_pair = (’_t’, ’h’)

vocab = []
vocab.append(most_frequent_pair)
```

There we can see each bigram and its frequency. For example, ('_t', 'h') occurs twice in the corpus, and it is taken as the most frequently occurring bigram, which we can save into the merge_list. Now it is the time to merge this pair in the corpus as stated in step 6.

```python
import re

def merge_pair_in_corpus(tokens, pair):
  # convert list of sentences into one big string
  # in order to do the substitution once
  tokens = '\n'.join(tokens)

  # regex to capture the pair
  p = re.compile(r'(?<!\S)' + re.escape(' '.join(pair)) + r'(?!\S)')

  # substitute the unmerged pair by the merged pair
  tokens = p.sub(''.join(pair), tokens)

  tokens = tokens.split('\n')
  return tokens

tokens = merge_pair_in_corpus(tokens, most_frequent_pair)
>>> tokens = ['_th i s _i s _th i s .']
```

The subword unit ’_th’ has been created, saved in the vocabulary and merged in the corpus. The last step is iterating until the subword vocabulary size has been reached or until there are no pairs with bigger than 1. At each step, the object *pairs* is computed again since there might be new pairs such as (’_th’, ’i’) in this example. The whole minimal code would look like this:

```python
corpus = ['this is this.']
vocab_size = 10
vocab = []

tokens = read_corpus(corpus)

for _ in range(vocab_size):
  pairs = get_stats(tokens)

  # frequency of the most common pair is 1, break loop
  if pairs.most_common(1)[0][1] == 1:
      break

  most_frequent_pair = pairs.most_common(1)[0][0]
  vocab.append(most_frequent_pair)
  tokens = merge_pair_in_corpus(tokens, most_frequent_pair)

>>> tokens = ['_this _i s _this .']
>>> vocab = [('_t', 'h'), ('_th', 'i'), ('_thi', 's')]
```

In each step of the iteration, the *get_stats* function iterates all the characters in the corpus, so the complexity is O(len(corpus) * length of sentence), for an average sentence length. Obtaining the most frequent pair takes constant time, since the object *pairs* is a Counter object and includes a function to retrieve the most frequent item. At the step of *merge_pair_in_corpus*, the corpus is iterated in its entirety again, with a complexity of O(len(corpus) * len(sent)). Therefore, the algorithm has a complexity of O(num_merges * len(corpus) * len(sent)). Iterating num_merges amount of times cannot be avoided, but operating through all the characters in the corpus is computationally very expensive.

One of the contributions of this thesis is an optimization of this algorithm, as will be shown in the following chapters.

Applying BPE to OOV words

In the event of an OOV word, such as 'these', which the corpus used in the previous example does not know, the BPE algorithm can create some subword units from the corpus used before.

1. Split the OOV word into characters after inserting '_' in the beginning.

2. Compute the pair of character or character sequences in the OOV word.

3. Select the pairs present in the learned operations.

4. Merge the most frequent pair.

5. Repeat steps 2-4 until merging is possible.

And this is the code in Python for such an algorithm:

```python
oov = 'these'
oov = ['_' + ' '.join(list(oov))]

i = 0
while True:
  pairs = get_stats(oov)
  # find the pairs available in the vocab learnt before
  idx = [vocab.index(i) for i in pairs if i in vocab]

  if len(idx) == 0:
    print("BPE completed")
    break

  # choose the most frequent pair which appears in the OOV word
  best = merges[min(idx)]

  # merge the best pair
  oov = merge_vocab(best , oov)

>>> oov = '_th e s e'
```

'_th' is the only known merge in the vocabulary, the rest of the characters ('e', 's', 'e') are unknown to the vocabulary so it does not know how to create any subword units.

BPE dropout

BPE dropout (Provilkov et al., 2019 [5]) changes the BPE algorithm by stochastically corrupting the segmentation procedure of BPE, producing multiple segmentations within the same fixed BPE framework.

It exploits the innate ability of BPE to be stochastic: the merge table remains the same, but when applying it to the corpus, at each merge step some merges are randomly dropped with probability p, hence the name of BPE dropout. In the paper p=0.1 is used during training and p=0 during inference. For the Chinese and Japanese languages, p=0.6 is used in order to match the increase in length of segmented sentences as other languages.

It is hypothesized in the paper that exposing a model to different segmentations might result in better understanding of the whole words as well as their subword units. The performance improvement with respect to normal BPE is consistent no matter the vocabulary size, but it is shown that the impact from using BPE-Dropout vanishes when a corpora size gets bigger. These results are replicated and confirmed in later chapters.

Sentences segmented with BPE-Dropout are longer. There is a danger that models trained with BPE-Dropout might use more fine-grained segmentation during inference and hence slow down the process.

BPE drawbacks

Kudo (2018) [14] showed that BPE is a **greedy algorithm** that keeps the most frequent words intact, while splitting the rare ones into multiple tokens. BPE splits words into unique sequences, meaning that for each word, a model observes **only one segmentation**, meaning that if there is a segmentation error, all the following steps are erroneous. Additionally, subwords into which rare words are segmented end up poorly understood.

Although the problem of unique segmentation can be improved with the BPE Dropout method, it is still susceptible to the common problems of BPE, namely, the greediness of the algorithm and fragility regarding segmentation errors, problems which are explored in the following chapters.

3.2 Translation

A significant component of this thesis is the use of word alignments, a method employed in statistical machine translation in order to gauge how good a translation is. If a word in French gets aligned to its equivalent word in English, for example *maison-house*, then the translation is correct. It is a metric to evaluate if a translation has been successful.

This thesis is not related to translation, neither SMT nor its more modern approach neural machine translation. But the word alignment algorithm is used in order to compare word alignments against BPE alignments. Taking word alignments as the basis, alignment algorithms are used to see if these BPE units are also correctly aligned to the BPE units in the other language. For example, ideally the English suffix *-ment* would be aligned to the German word ending of *-keit*. But before diving deep into word alignments and how they work, it is important to understand the background of this algorithm, as well as its original purpose and a general notion of how statistical machine translation works.

3.2.1 Statistical machine translation (SMT)

SMT is a machine translation approach where translations are generated based on statistical models, whose parameters are derived from the analysis of bilingual text corpora. It can be done with rule-based approaches in a supervised way, or example-based approaches, in an unsupervised way. A document is translated according to the probabilistic distribution *P(e/s)* that a word *e* in the target language (for example English) is the translation of a word *s* in the source language (for example, Spanish).

- Suppose that s = gracias
- P(thanks|gracias) = 0.45
- P(appreciate|gracias) = 0.13
- P(water|gracias) = 0.00001

Given the word *gracias*, the translation algorithm would give a higher probability that it is aligned with *thanks*, and lower probabilities to other alignments. For each word, there usually is a list with alignment possibilities and its probabilities, and the one with the highest probability is taken as alignment. Note that since this translation method is statistical, there is no direct translation, there is no direct approval of one alignment and dismissal of the rest, since all is based on probabilities.

Typically, a translation model first translates the source language into a broken version of the target language, using an algorithm such as the expectation-maximization algorithm. Afterwards, a language model in the target language makes the broken language look more natural, in the way native speakers would speak it. A good language model will for example assign a higher probability to the sentence "the house is small" than to "small the is house". An example of a translation system from Spanish to English can be seen in Figure 3.6. In the first step, the translation model needs to know which words to align in a source-target sentence pair, which is handled by the word alignment step.

3.2.2 Word alignments

Word alignment between a parallel corpora is the NLP task of identifying translation relationships among the words in a parallel text, resulting in a graph between the two sides of the texts, with an arc

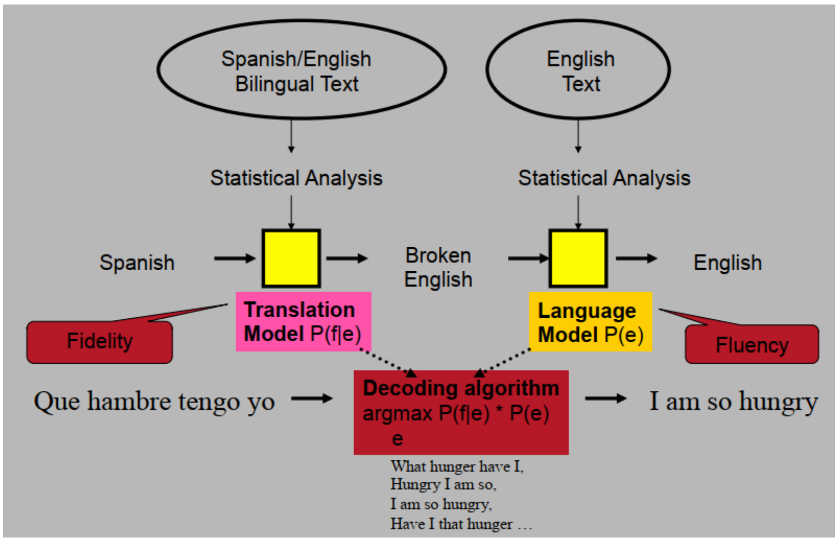


Figure 3.6: Example of Spanish-English SMT system.

between two words if they are translations of one another. Figure 3.7. shows an alignment example between a sentence in English and its counterpart in French. Alternatively, the alignments can also be displayed in a matrix as shown in Figure 3.8.

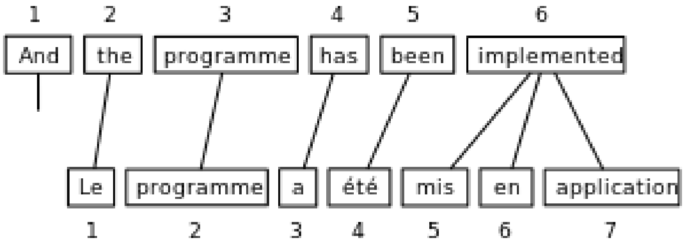


Figure 3.7: Word alignments between an English and French sentence.

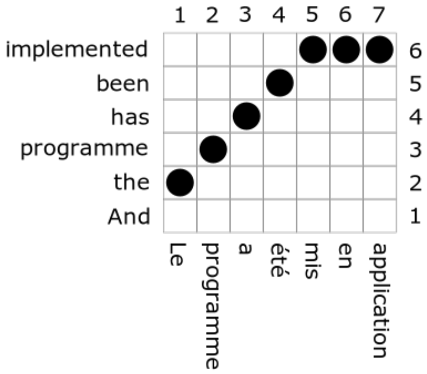


Figure 3.8: Word alignments between an English and French sentence in matrix form.

The most basic words usually have one-to-one alignments, such as *the-le* in this example, and words with the same root, *programme-programme* since they both come from Latin. However, in many cases one language might express something using one word but another might express with with many words, distributed throughout the sentence. Besides, as seen in this example, some words do not even have an alignment (*And*), and some have multiple alignments, in this case one-to-many: *implemented-mis en application.* There can also be many-to-one and many-to-many alignments.

The parameters of word alignment methods are usually estimated by observing word-aligned texts [20], and automatic word alignment is typically done by choosing that alignment which best fits a statistical machine translation model. A popular algorithm to find word alignments is the **expectation-maximization algorithm** [21]

For training, historically IBM models have been used. [22] These models are used in statistical machine translation to train a translation model, as well as an alignment model. They make use of the expectation-maximization algorithm explained above: in the expectation step, the translation probabilities within each sentence are computed; in the maximization step, these probabilities are accumulated to global translation probabilities.

This approach is an example of unsupervised learning, meaning that the system has no knowledge of the kind of output it is expected to produce, but tries to find values for the unobserved model and alignments which best explain the observed parallel text. In some cases, a small number of manually aligned sentences is used to help the model, as a way to explore supervised learning. [23] These models are able to more easily take advantage of combining many features within the data, such as context, syntactic structure, part-of-speech or translation lexicon information, which are difficult to integrate into the unsupervised models generally used.

Fastalign algorithm

Fastalign algorithm [24] is a simple log-linear reparametrization of IBM Model 2 that overcomes problems from both Model 1 and Model 2. Training this model is consistently ten times faster than Model 4. An open-source implementation of the alignment model described in this paper is available in Github.

Fastalign is a variation of the lexical translation. Lexical translation works as follows: given a source sentence f with length n, first generate the length of the target sentence m, where the target sentence is e. Then generate an alignment vector of length m that indicates which source word (or null token) each target word will be a translation of. Lastly, generate the m output words, where each word in e depends only on the word in f it is aligned with.

Fastalign's modification is that the distribution over alignments is parametrized by a null alignment probability and a precision parameter, which controls how strongly the model favors alignment points close to the diagonal (if we use the word alignment matrix like in the example above). It also adds a symmetrization step called GDFA.

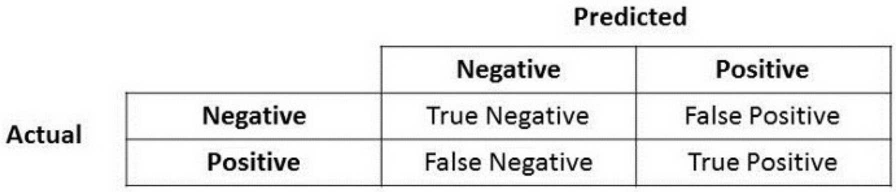
The paper [24] has more detailed information on training, inference and results.

Eflomal algorithm

The Eflomal alignment algorithm [25], presented in 2016, defines itself as an *Efficient Low-Memory Aligner.* Its code is available on Github, and it is a word alignment tool based on *Efmaral*, with some improvements, such as more compact data structures, yielding a lower memory requirement, and the fact that the estimation of alignment variable marginals is done one sentence at a time, which also saves a lot of memory at no detectable cost in accuracy. Eflomal differs from Fastalign in the sense that it is based on a hidden Markov Model, as opposite to Fastalign which is based on an IBM model. Eflomal also requires the simmetrization step GDFA, and has historically had better results than Fastalign. Technical details relevant to both *Efmaral* and *Eflomal* can be found in the cited article.

3.2.3 Alignment metrics

Evaluating alignments is usually done by the following metrics: precision, recall, F1 score and accuracy. It is assumed that the actual results are at a disposal, in this case, that the correct alignments are known. The system then makes a prediction for certain alignments. This confusion matrix shows the table for true positives, true negatives, false positives and false negatives.



		Predicted	
		Negative	Positive
Actual	Negative	True Negative	False Positive
	Positive	False Negative	True Positive

Figure 3.9: Confusion matrix for actual and predicted results

If an alignment is predicted and is also correct, that constitutes a true positive. If an alignment is predicted but is not correct, that constitutes a false positive. If a correct alignment was not predicted, it constitutes a false negative. And an alignment that is not present in the actual alignment list nor in the prediction list is a true negative. These are the formulas to calculate the precision, recall and F1 score:

- Precision = true positives / (true positives + false positives) = true positives / (total predicted positives)
- Recall = true positives / (true positives + false negatives) = true positives / (total actual positives)
- F1 score = 2 x precision x recall / (precision + recall)

In a nutshell, precision evaluates the following: out of all the predicted items, how many are correct? And recall: out of all the correct items, how many were predicted? The F1 score is a balance between the two, so that a system with a high precision but low recall might have a decent F1 score.

This is the traditional use of precision and recall. When talking specifically about alignments however, a slightly different definition is commonly provided:

- Precision = test alignments x possible alignments / test alignments
- Recall = test alignments x sure alignments / sure alignments
- AER = 1 - (test x sure + test x possible) / (test + sure)

AER, Alignment Error Rate [26] [22], is a commonly used metric for assessing sentence alignments. It combines precision and recall metrics together such that a perfect alignment must have all of the sure alignments and may have some possible alignments.

4 Methodology

This chapter explains the technical content of this thesis in broad strokes, the methodology used and the general idea of each method employed. For a more in-depth analysis and code snippets and explanations, refer to the Development chapter 5. The results, plots and graphs are displayed in the Results chapter 6. This thesis, first of all, aims to replicate the results of BPE and BPE dropout.

4.1 Replication of BPE

Using Sennrich et al.'s [4] code on Github, the first goal of the thesis is to replicate the BPE algorithm, and gauge how good the BPE units are by using an alignment algorithm and matching it against a gold standard. For that, these steps are undertaken:

1. Write learn-BPE from corpus algorithm
2. Write apply-BPE to corpus algorithm
3. Write extract alignment script
4. Write calculate alignment scores script

The corpus employed in this thesis is a 10k sentence English-German corpus. As an excerpt of three sentences from the corpora:

> English
> 21 The Committee on Transport and Tourism has adopted four amendments for the second reading .
> 22 They will certainly enhance the feeling of the right of movement by EU citizens and they will also certainly benefit disabled drivers .
> 23 The initial Commission proposal was adopted unamended by Parliament on first reading .
>
> German
> 21 Der Transportausschuß hat für die zweite Lesung vier Änderungsanträge beschlossen .
> 22 Sie werden bei den EU-Bürgern gewiß das Gefühl für das Recht auf Freizügigkeit stärken , und sie werden gewiß auch behinderten Fahrern Vorteile bringen .
> 23 Der ursprüngliche Vorschlag der Kommission wurde vom Parlament in erster Lesung ohne Änderungen verabschiedet .

Each step of the pipeline will be detailed as follows.

4.1.1 Learn-BPE algorithm

Sennrich's repository's code has some additional parameters that were not relevant for a minimal implementation of the BPE algorithm, so the script was adapted. These are the steps for a minimal algorithm to learn-BPE units:

1. Read corpus into tokens, parse index.
2. Count pair frequencies.
3. Start loop from 1 until desired vocabulary size. In this case, 10k merges.
    a) Get most frequent pair.
    b) Append most frequent pair to vocabulary.
    c) Merge pair in corpus.
    d) Count pair frequencies in corpus.
4. Write vocabulary to a file.

This step of the pipeline only has to be done once for each corpus, afterwards the vocabulary can be used in different ways. But this minimal algorithm, since it has to count all the pairs in the whole corpus in each iteration, takes a long time. One of the improvements of this thesis is optimizing the runtime of this algorithm, which will be explained in the subsequent sections of this chapter. With the given corpus, these are the 10 most frequent merges in the English language: _t h, _th e, o n, r e, t i, e n, e r, i n, i s, n d. And the 10 most frequent merges in German: e n, e r, c h, i e, e i, n d, n g, _d ie, s t, i ch. Throughout this thesis, the special symbol _ has been used to mark the beginning of each word. Therefore, *_t* only applies to those *t* characters that occur in word beginnings, not anywhere else in the word.

4.1.2 Apply-BPE algorithm

The learnt vocabulary can be applied to a corpus. In the case of this thesis, this corpus is the same as the one used to learn the BPE units. Different num_merges are declared. For example, for 500 merges, only the first 500 merges of the vocabulary are taken, and there are barely any meaningful text segments. For bigger merge values, more and more subword units get merged. In this thesis, merges range from 500 up to 8000. These are the steps for applying BPE merges to a corpus:

1. Load data, corpus and BPE vocabulary.
2. Start loop for all numbers of merges: 500 merges, 2000 merges, 8000 merges.
    a) Start loop from 1 until desired amount of merges: from 1 to 500 for example.
        i. Merge the current most frequent pair in corpus.
    b) Write merged corpus to .bpe file.

This is what the excerpts from the example corpora above look like after 100 merges, that is, after the 100 most common units in the language have been merged:

English
_the _Comm it t e e _on _T ran s p ort _and _T ouris m _has _a d o p t ed _f our _a m end ment s _for _the _sec ond _re a d ing _.
_the y _w ill _cer t a in ly _en h an ce _the _f e e ling _of _the _ri g h t _of _m o ve ment _b y _E U _citi z en s _and _the y _w ill _al s o _cer t a in ly _ben e f it _d is a bled _d ri ver s _.
_the _in iti al _Commission _pro p o s al _w as _a d o p t ed _u n a m end ed _b y _P ar li a ment _on _f ir st _re a d ing _.

German
_der _T rans p or t a ussch u ß _h at _für _die _z w eite _L es ung _v ier _Ä nder ung s ant r ä ge _besch l o ss en _.
_s ie _wer den _bei _den _E U - B ür ger n _ge w i ß _das _G e f ü h l _für _das _Re ch t _auf _F r ei z ü g i g k eit _st ä r k en _, _und _s ie _wer den _ge w i ß _au ch _beh inder ten _F a hr er n _V or tei l e _b r ingen _.
_der _ur s p r ü ng lich e _V or sch la g _der _Ko mm iss ion _w ur de _v o m _P ar la ment _in _er ster _L es ung _o h n e _Ä nder ungen _vera b sch ie det _.

In English, *_the*, *_and*, *ment* and other very common words and affix/suffixes have been merged. As for German, we can see very common words being merged such as *_die*, *_das*, *_bei* and so on. The same sentences after 1000 merges:

English
_the _Committee _on _T rans p ort _and _T ourism _has _adop ted _four _amendments _for _the _second _reading _.
_they _will _certain ly _en h ance _the _feeling _of _the _right _of _mo vement _by _EU _citizens _and _they _will _also _certain ly _bene f it _dis abled _d rivers _.
_the _initi al _Commission _proposal _was _adop ted _un amended _by _Parliament _on _first _reading _.

German
_der _T ransp ort aussch uß _hat _für _die _zweite _L esung _v ier _Änderungsant räge _beschlossen _.
_sie _werden _bei _den _EU - B ür gern _gew iß _das _Ge fü hl _für _das _Recht _auf _Freiz ü g igkeit _st är ken _, _und _sie _werden _gew iß _auch _beh inderten _F ahrern _Vorteile _b ringen _.
_der _ur sp rüngliche _Vorschlag _der _Kommission _wurde _vom _Parlament _in _erster _L esung _ohne _Änderungen _verab schiedet _.

We can see bigger subwords being merged, such as *reading*, *first*, *zweite* and so on. And the sentences after 4000 merges:

English
_the _Committee _on _Transport _and _Tourism _has _adopted _four _amendments _for _the _second _reading _.
_they _will _certainly _enhance _the _feeling _of _the _right _of _movement _by _EU _citizens _and _they _will _also _certainly _benefit _dis abled _drivers _.
_the _initial _Commission _proposal _was _adopted _un amended _by _Parliament _on _first _reading _.

German
_der _T ransp ortausschuß _hat _für _die _zweite _Lesung _vier _Änderungsanträge _beschlossen _.
_sie _werden _bei _den _EU-B ür gern _gew iß _das _Ge fü hl _für _das _Recht _auf _Freizügigkeit _stärken _, _und _sie _werden _gew iß _auch _behinderten _Fahrern _Vorteile _bringen _.
_der _ursprüngliche _Vorschlag _der _Kommission _wurde _vom _Parlament _in _erster _Lesung _ohne _Änderungen _verabschiedet _.

Most words are merged, except *_dis abled* in English, and *_T ransp ortausschuß* in German for instance. At this point, the corpus is not composed of words anymore, but rather of subwords.

4.1.3 Extract alignments

To evaluate if the BPE units are good, bilingual corpora are aligned and then compared against a gold standard. The motivation behind alignment and how it works can be found in 3.2. The challenge here lies in the fact that if the BPEs are of good quality, the alignment algorithm will align subword items correctly, and therefore in the subword-to-word mapping, the word alignments will have a high score relative to the gold standard.

On the first step, alignment, two algorithms have been used, namely *Fastalign* and *Eflomal*. The software installation guides can be found in the development section 5. These algorithms take English text and German as input and create an alignment file as output. For the example above:

English sentence: The Committee on Transport and Tourism has adopted four amendments for the second reading .
German sentence: Der Transportausschuß hat für die zweite Lesung vier Änderungsanträge beschlossen .
Alignment output: 0-0 1-1 2-1 3-1 4-1 5-1 6-2 7-9 8-7 9-8 10-3 11-4 12-5 13-6 14-10

English sentence: The initial Commission proposal was adopted unamended by Parliament on first reading .
German sentence: Der ursprüngliche Vorschlag der Kommission wurde vom Parlament in erster Lesung ohne Änderungen verabschiedet .
Alignment output: 0-0 1-1 2-4 3-2 3-3 4-5 5-13 6-11 6-12 7-6 8-7 9-8 10-9 11-10 12-14

Many words have one-to-one alignment, such as *four-vier* and *adopted-verabschiedet.* Some others have many-to-one alignments, such as *Committee on Transport and Tourism-Transportausschuß* and one-to-many alignments such as *unamended-ohne Änderungen.* In our case however, the input files are not composed by words, but rather by subwords. And the alignments are done among subwords. Using the example above but with subwords instead of words:

> English sentence: _the _Committee _on _Transport _and _Tourism _has _adopted _four _amendments _for _the _second _reading _.
> German sentence: _der _T ransp ortausschuß _hat _für _die _zweite _Lesung _vier _Änderungsanträge _beschlossen _.
> Alignment output: 0-0 1-1 2-1 3-1 4-1 5-1 1-2 2-2 3-2 4-2 5-2 1-3 2-3 3-3 4-3 5-3 6-4 7-11 8-9 9-10 10-5 11-6 12-7 13-8 14-12

Now there are subword alignments as opposed to word alignments. For instance, since the German word *Transportausschuß* is divided into three words, namely *T ransp ortausschuß*, the many-to-one alignment from the previous case is now a many-to-many alignment. The number of alignment has grown from last example. Because the gold standard against which the system is being evaluated consists of word alignments, it is necessary to map subword alignments into word alignments, which is the second step in this algorithm. Using English and German corpora and the alignment file as input files, this algorithm gives a word alignment file as output.

> English sentence: _the _Committee _on _Transport _and _Tourism _has _adopted _four _amendments _for _the _second _reading _.
> German sentence: _der _T ransp ortausschuß _hat _für _die _zweite _Lesung _vier _Änderungsanträge _beschlossen _.
> Subword alignment: 0-0 1-1 2-1 3-1 4-1 5-1 1-2 2-2 3-2 4-2 5-2 1-3 2-3 3-3 4-3 5-3 6-4 7-11 8-9 9-10 10-5 11-6 12-7 13-8 14-12
> Word alignment output: 0-0 1-1 2-1 3-1 4-1 5-1 6-2 7-9 8-7 9-8 10-3 11-4 12-5 13-6 14-10

4.1.4 Calculate alignment scores

In the final step, the alignment scores are computed against the gold standard. After loading the gold dataset, each alignment file is matched against this dataset, obtaining precision, recall, F1 score and AER metrics. In the case of 100 learnt symbols, there will be an associated score and so on for other numbers of learnt symbols. Additionally, the gold standard's scores are also computed as a baseline. To make it more visual, the scores are plotted and saved into a *.png* image file as well as *.csv* file with the exact numbers.

4.2 Replication of BPE-dropout

The difference between BPE-dropout and standard BPE is the fact that some merges do not take place. Based on this random factor, each time this system is carried out, new BPE merges are created. For example, if the most merge in English *(_t, h)* is not merged, the resulting file of BPE merges is vastly different than if the 10th most frequent merge does not take place. In order to obtain different instances, the dropout algorithm is run a number of times, in this case 10 times, the alignments extracted 10 times, and then the alignments aggregated. Regarding each specific step, the algorithm to create the merge list (learn_bpe) remains unchanged, the first slight change occurs when applying the BPE algorithm to the corpus.

In the apply-BPE algorithm, a random variable is created for every merge: if it falls below a threshold, the dropout threshold, the merge in question is discarded. Apart from this, the whole algorithm is run a number of times, creating a number of BPE files, usually ten throughout this thesis. When extracting alignments, since now there are ten BPE files instead of a single one, the whole algorithm is run ten times, and alignments for all numbers of merges multiplied by all dropout repetitions are saved. At this point, there are ten alignments for each sentence. Which alignments to pick, is the next step to be resolved. Three methods are selected:

- Create the union of all alignments.
- Create the intersection of all alignments.
- Create a threshold parameter, for example 0.5. If an alignment is present in 50% of all alignments for that sentence, it is added to the aggregated file.

To illustrate this with a example. Files 1, 2 and 3 are three alignment files for a given sentence pair.

- File 1: 0-0 0-1 1-1 1-2 2-3
- File 2: 0-0 0-1 1-2
- File 3: 0-0 1-1 1-3
- Union file: 0-0 0-1 1-1 1-2 1-3 2-3
- Intersection file: 0-0
- Aggregated file: 0-0 0-1 1-1 1-2

As it is visible in the example, the **union** case takes all alignments into account. By brute force, possibly most of the correct alignments will be present in the alignment, yielding a high recall, but the majority of the alignments in the union file will be incorrect, that is, it will have low precision. By contrast, the **intersection** case is the opposite. The file is much shorter since only the alignments present in *all* files are considered, which means that these alignments will mostly be correct. But also many of the actually correct alignments will not be present, because they might have been skipped, and therefore are not present in the intersection.

The method of creating an **aggregated file** with the threshold aims to alleviate this problem by creating a sort of middle point between union and intersection. Taking a threshold value closer to 0 will mean that almost all alignments will be accepted, and therefore the score will be closer to the score of the union. In the opposite end of the spectrum, a threshold value close to 1 means that only alignments present in most alignment files will be accepted, and this resembles the intersection. Many experiments have been done with threshold values ranging from 0.3 to 0.9, and the corresponding scores can be found in the Results chapter.

4.3 Improvement of learn-BPE algorithm

One of the drawbacks of the algorithm to learn-BPE units is that every time a pair of sequences is merged in the corpus, all sequence pairs and their frequencies has to be computed from scratch. This requires iterating over all characters from each sentence in the corpus, and for each iteration, a pair of sequences gets merged.

4.3.1 Updating only neighboring sequences

In order to understand the magnitude of this computation, let us explore the step of merging a pair in the corpus. We can use this small corpus as an example:

0 A start and the end .
1 The index of a document .
2 My name is Bob .

In the foremost step, all characters are separated into individual tokens, and the beginning of the work is marked:

0 _A _s t a r t _a n d _t h e _e n d .
1 _T h e _i n d e x _o f _a _d o c u m e n t .
2 _M y _n a m e _i s _B o b .

For the sake of the example, let us assume that ('n', 'd') is the most frequent pair of tokens. When merging them, the corpus is altered in the following way:

0 _A _s t a r t _a nd _t h e _e nd .
1 _T h e _i nd e x _o f _a _d o c u m e n t .
2 _M y _n a m e _i s _B o b .

In the brute force approach, each pair's frequency is computed again: the sequence pairs ('_T', 'h'), ('h', 'e'), etc. are all revisited and their frequencies counted from scratch. But this does not have to be the case, actually the only pairs that need to be updated are the ones surrounding ('n', 'd'). When viewing this merge from a different perspective, these are the changes that occur in the pairs of tokens:

- ('_a', 'n') in sentence 0 now becomes ('_a', 'nd')
- ('_e', 'n') in sentence 0 now becomes ('_e', 'nd')
- ('i', 'n') in sentence 1 now becomes ('i', 'nd')

- ('d', 'e') in sentence 1 now becomes ('nd', 'e')

The pair of tokens in the word 'and' which previously was ('_a', 'n'), now becomes ('_a', 'nd') since 'n' and 'd' have been merged. In this instance, the pair ('_a', 'n') no longer exists, and a new pair has been created: ('_a', 'nd'). And so on for the rest of the tokens. As a result, only the following frequency updates must be made:

- Reduce frequency of ('_a', 'n') by 1, increase frequency of ('_a', 'nd') by 1.
- Reduce frequency of ('_e', 'n') by 1, increase frequency of ('_e', 'nd') by 1.
- Reduce frequency of ('i', 'n') by 1, increase frequency of ('i', 'nd') by 1.
- Reduce frequency of ('d', 'e') by 1, increase frequency of ('n', 'de') by 1.

All the other pairs remain unchanged. This is the major improvement of this thesis regarding the learn-BPE algorithm, **the fact that only neighboring tokens of the merged pair need to be updated**. Now, instead of updating each pair in each sentence, it is only necessary to update the merge pair's surrounding tokens. The way to do this is by locating the merged pair in the sentence, and updating the previous and next tokens.

4.3.2 Saving indexes of pairs

If a pair is very frequent, it is safe to assume that it will be present in the majority of the sentences in the corpus. In the example above, the last sentence does not contain the ('n', 'd') pair. In a bigger corpus, the more merges are done, the rarer they become. It is therefore useful to only visit those sentences where the pair is present. If the merged pair only appears in 10% of the corpus' sentences, it is a waste of resources to visit all sentences. This can be solved by saving the index where each pair appears. This way, each pair has its frequency associated to it, as well as a list of indexes where it is present. Creating this index list can be done in the initial step of the algorithm when the corpus is read and iterated completely, each pair's frequency computed for the first time, and the indexes recorded.

This way, when accessing the most frequent pair in the corpus, we can also access to the sentences they are present in, and iterate only those.

4.4 BPE without word boundaries

Up until this point, the underlying principle of creating BPE units is that merges between words are not considered. But given the existence of languages without any clear word boundaries, this thesis also explores what would happen if merges between different words were allowed, which has not been done before in literature, to the knowledge of the author. Regarding the algorithm pipeline, all remains the same except the algorithm to create BPE merges. Now, instead of having a special token to mark the beginning of each word, the same token is used to replace whitespace. For instance, the space mode tokenization used until now and the new no-space mode tokenization:

- Raw sentence: The cake is delicious .
- Sentence after space mode tokenization: _T h e _c a k e _i s _d e l i c i o u s .
- Sentence after no-space mode tokenization: T h e _ c a k e _ i s _ d e l i c i o u s _ .

This way, merges with whitespace are possible, as well as merges between endings of some words and beginnings of the next. After learning BPE units using this method, these are the most common merges in English: e _, t h, s _, t _, n _, d _, th e_, e r, i n, y _. The most common BPE unit between words is *of_ the_*. As for German, the most common merges are: n _, e r, e n_, c h, e _, _ d, e i, u n, t _, er _. And the most common BPE unit between words is *, _da*.

Since now merges between words are possible, there are many more possible merges, and the algorithm is not as fast as in the space mode. The rest of the algorithms, namely applying BPE units to a corpus, extracting alignments, and calculating scores remain the same. However, it is important to consider that when aligning units this way, a unit containing multiple words might get aligned to a unit containing also multiple words, so the mapping subword to word is slightly altered, relative to the previous case where only parts of words were aligned.

5 Development

This chapter explains the development of the thesis in a deeper way, adding more detailed explanations, the code and algorithms previously explained in the Methodology section. 4 This chapter includes coding practices, how global variables are stored and treated, the tree directory of the project, and the various tasks that have been undertaken during this thesis, which include the following:

1. Replication of the BPE algorithm and results
2. Replication of the BPE-dropout algorithm and results
3. Improvement of the learn BPE algorithm
4. Study and implementation of BPE without word boundaries

5.1 Coding practices

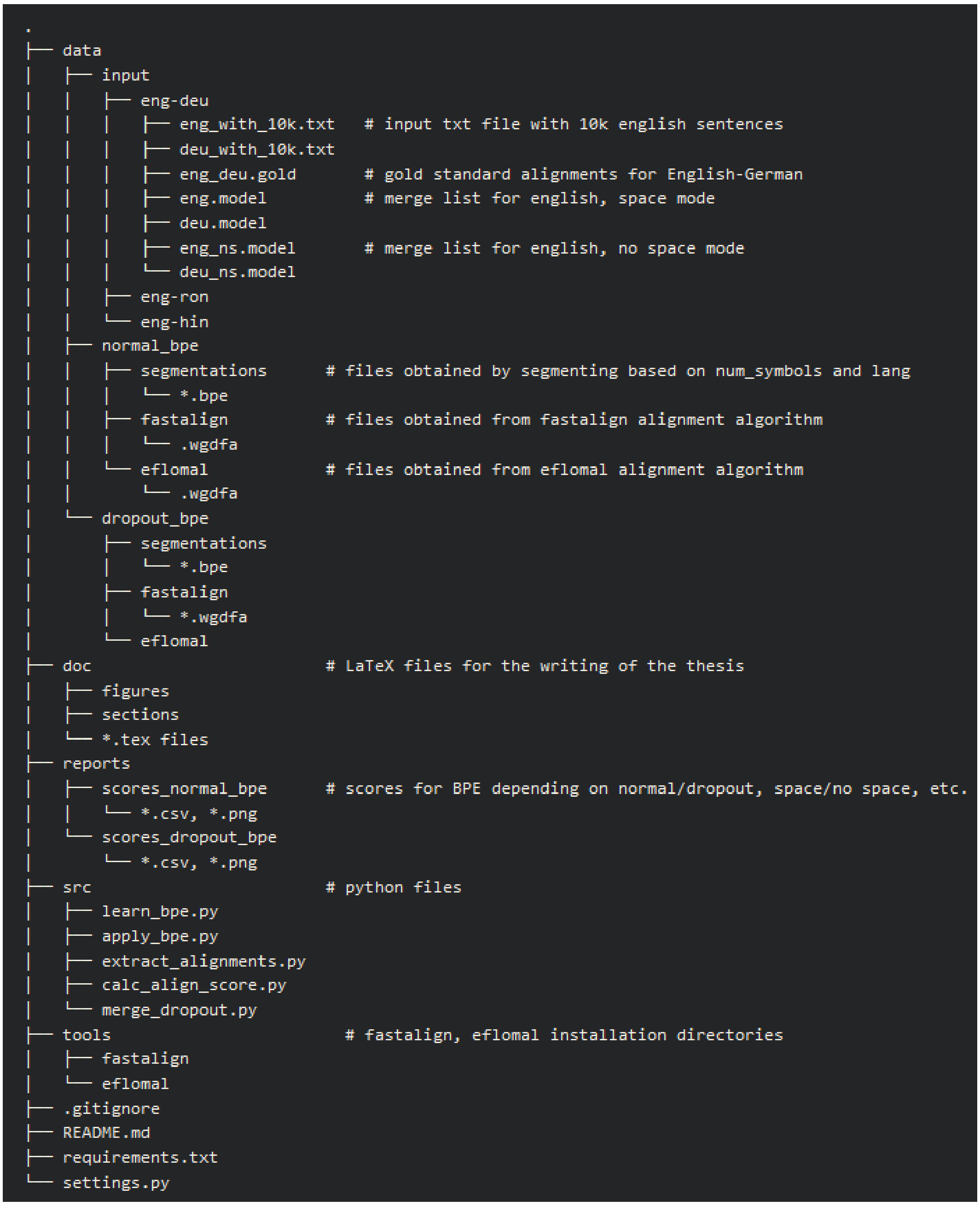
To ensure consistency, the parameters for the pipeline, called global variables, such as num_symbols, dropout, file paths, etc. have been written in *settings.py*. The word separator is a special Unicode character. The languages used as source and target languages are English and German respectively. The number of symbols to be learnt in the learn-BPE phase is set as 20000. When applying these merges to the corpus, in order to have different variants and explore the differences between merging few symbols or more, a number of key numbers are selected for these experiments, as seen in the variable *all_symbols*.

```
# settings.py
import os
import glob
from os.path import join

word_sep = u’\u2581’
source, target = ’eng’, ’deu’
learn_merges = 20000
merges = [100, 200, 500, 1000, 2000, 4000, 6000, 8000]

rootdir = os.getcwd()
if rootdir.split(os.sep)[-1] == ’src’:
    rootdir = os.sep.join(rootdir.split(os.sep)[:-1])
inputdir = join(rootdir, ’data’, ’input’, source+’-’+target)
bpedir = join(rootdir, ’data’, ’normal_bpe’)
baselinedir = join(rootdir, ’reports’, ’scores_normal_bpe’)
scoredir = join(rootdir, ’reports’, ’scores_normal_bpe’)
goldpath = join(inputdir, source+’_’+target+’.gold’)
inputpath = {}
for filename in glob.glob(join(inputdir, "*.txt")):
  inputpath[filename.split(os.sep)[-1].split(’_’)[0]] = filename
```

Regarding directories and paths, the tree view of the project is displayed in the following figure. The input data is saved into *data/input*, the rest of the intermediate data files such as *.bpe*, *.wgdfa* and others are saved into their respective folder depending on the dropout parameter. The models are saved in the general *data* folder with the *.model* extension. The LaTeX files for writing this thesis are saved in the *doc* folder, the score figures and csv files in *reports*, all Python scripts in *src*, the *Fastalign* and *Eflomal* installation files in *tools*, and README, requirements, gitignore and the global variable file *settings.py* are saved in the root folder.



```
.
├── data
│   ├── input
│   │   ├── eng-deu
│   │   │   ├── eng_with_10k.txt   # input txt file with 10k english sentences
│   │   │   ├── deu_with_10k.txt
│   │   │   ├── eng_deu.gold       # gold standard alignments for English-German
│   │   │   ├── eng.model          # merge list for english, space mode
│   │   │   ├── deu.model
│   │   │   ├── eng_ns.model       # merge list for english, no space mode
│   │   │   └── deu_ns.model
│   │   ├── eng-ron
│   │   └── eng-hin
│   ├── normal_bpe
│   │   ├── segmentations      # files obtained by segmenting based on num_symbols and lang
│   │   │   └── *.bpe
│   │   ├── fastalign          # files obtained from fastalign alignment algorithm
│   │   │   └── .wgdfa
│   │   └── eflomal            # files obtained from eflomal alignment algorithm
│   │       └── .wgdfa
│   └── dropout_bpe
│       ├── segmentations
│       │   └── *.bpe
│       ├── fastalign
│       │   └── *.wgdfa
│       └── eflomal
├── doc                        # LaTeX files for the writing of the thesis
│   ├── figures
│   ├── sections
│   └── *.tex files
├── reports
│   ├── scores_normal_bpe      # scores for BPE depending on normal/dropout, space/no space, etc.
│   │   └── *.csv, *.png
│   └── scores_dropout_bpe
│       └── *.csv, *.png
├── src                        # python files
│   ├── learn_bpe.py
│   ├── apply_bpe.py
│   ├── extract_alignments.py
│   ├── calc_align_score.py
│   └── merge_dropout.py
├── tools                        # fastalign, eflomal installation directories
│   ├── fastalign
│   └── eflomal
├── .gitignore
├── README.md
├── requirements.txt
└── settings.py
```

Figure 5.1: Project directories in tree mode

5.2 Replication of BPE results

The first programming task in this thesis has been to replicate the BPE results from Sennrich et al. [4]. In order to modularize the code in clear units, these have been the steps:

1. Write the learn-BPE from corpus algorithm
2. Write the apply-BPE to corpus algorithm
3. Write extract alignments script
4. Write calculate alignment scores script

In the following sections, each step will be explained in detail, with comments and examples for each function.

5.2.1 Learn-BPE algorithm

In learning BPE units, the first step is to read the corpus into an appropriate format. All the libraries and global variables imported are also added here for simplicity.

```
# learn_bpe.py
import os
import re
import sys
import codecs
from os.path import join
from collections import defaultdict, Counter
# import global variables from settings.py
sys.path.insert(1, os.path.join(sys.path[0], ’..’))
from settings import *

def read_corpus(corpus: list) -> list:
  ’’’
  Read corpus, strip index and new line characters.
  A word_sep symbol is added at the beginning of each word.
  example:
  tokens = [
    ’\_w e \_d o \_n o t \_b e l i e v e
    \_t h a t \_w e \_s h o u l d
    \_c h e r r y - p i c k \_.’,
    ...
]
  ’’’
  tokens = []
  for line in corpus:
    line = line.split(’\t’)[1].strip(’\r\n ’)
    line = line.split()
    line[0] = str.lower(line[0])

    # add word_sep to each beginning of word and join by space
    tokens.append(’ ’.join([word_sep + ’ ’.join(word) for word in line]))
  return tokens
```

Once the corpus is parsed, all pairs and their frequencies are computed and saved in a suitable data structure.

```
# learn_bpe.py
def get_stats(tokens: list) -> Counter:
  ’’’
  Count frequency of all bigrams and the frequency per index.
  pairs = {
    (’s’, ’h’): 5,
    (’h’, ’e’): 6
  }
  The last token ’.’ or word_sep. is not merged with anything.
  ’’’
  pairs = Counter()
  for i, sent in enumerate(tokens):
    # get stats for each word independently,
    # no bigrams between different words
    for word in sent[1:].split(’ ’+word_sep):
      symbols = symbols.split()
      for j in range(len(symbols) - 1):
        pairs[symbols[j], symbols[j + 1]] += 1
  return pairs
```

Following the algorithm explained in the previous chapter, 4.1.1, the big/outer loop iterates between extraction of highest frequency pairs and merging of corpus until the maximum number of merges is reached.

```
# learn_bpe.py
def merge_token(corpus: list, most_frequent: tuple) -> list:
  str_corpus = ’\n’.join(corpus)
  str_corpus = str_corpus.replace(’ ’.join(most_frequent), ’’.join(most_frequent))
  return str_corpus.split(’\n’)

def learn_bpe(argsinput: str) -> list:
  ’’’
  Learn BPE operations from vocabulary. Steps:
  1. split corpus into characters, count frequency
  2. count bigrams in corpus
  3. merge most frequent symbols
  4. Update bigrams in corpus
  ’’’
  corpus = read_corpus(argsinput)
  most_frequent_merges = []
  for i in range(learn_merges):
    pairs = get_stats(corpus)
    try:
      most_frequent = pairs.most_common(1)[0][0]
    except:
      # pairs is empty
      break
    most_frequent_merges.append(most_frequent)
    corpus = merge_token(corpus, most_frequent)
  return most_frequent_merges
```

After the loop ends, the merge list is saved into a *.bpe* file. The following code includes this function as well as the main function that calls out all functions.

```python
# learn_bpe.py
def write_bpe(lang: str, most_freq_merges: list):
  bpe_file = codecs.open(join(inputdir, lang+'.model'), 'w', encoding='utf-8')
  bpe_file.write(f"{lang} {len(most_freq_merges)}\n")
  bpe_file.write('\n'.join(' '.join(item) for item in most_freq_merges))
  return

if __name__ == '__main__':
  for lang in [source, target]:
    argsinput = codecs.open(inputpath[lang], encoding='utf-8')
    most_freq_merges = learn_bpe(argsinput)
    write_bpe(lang, most_freq_merges)
```

5.2.2 Apply-BPE algorithm

After learning BPE units, there is already a merge list written into the project data directory, which can now be loaded into memory in order to apply these merges to the corpus. The following function returns a list of the languages present in the project, BPE models and corpora for each language.

```python
# apply_bpe.py
import os
from os.path import join
import sys
import codecs
# import global variables from settings.py
sys.path.insert(1, os.path.join(sys.path[0], '..'))
from settings import *
from learn_bpe import read_bpe_model, read_corpus

def load_data() -> (list, list, list):
  langs = [source, target]
  bpe_models = []
  corpora = []
  for lang in langs:
    argsinput = codecs.open(inputpath[lang], encoding='utf-8')
    corpora.append(read_corpus(argsinput))
    bpe_model = codecs.open(join(inputdir, lang+'.model'), encoding='utf-8').readlines()
    bpe_model = [tuple(item.strip('\r\n ').split(' ')) for item in bpe_model]
    bpe_models.append(bpe_model[1:])
  return langs, bpe_models, corpora
```

Once this data is available, the merge list is applied to the corpus iteratively, for the symbols present in the global variable *all_symbols*. To avoid repetition, when aiming for 8000 merges, the loop is halted at 100, 500 merges respectively, in order to save the state of the corpus into the *.bpe* file. After writing the file in the system, the loop resumes again. A less efficient approach would be making the loop run until 100 merges, then start from scratch and run until 500, and so on. This method is equally effective but not nearly as efficient.

```python
# apply_bpe.py
def write_bpe(lang: str, num_symbols: int, merged_corpus: str):
  outputpath = join(bpedir, 'segmentations', f"{lang}_{num_symbols}.bpe")
  argsoutput = codecs.open(outputpath, 'w', encoding='utf-8')
  argsoutput.write(merged_corpus)
  return

def apply_bpe(langs: list, bpe_models: list, corpora: list):
  for lang, bpe_model, corpus in zip(langs, bpe_models, corpora):
    bpe_model = bpe_model[:max(merges)]
    k = 0
    str_corpus = '\n'.join(corpus)
    for j, bigram in enumerate(bpe_model):
      str_corpus = str_corpus.replace(' '.join(bigram), ''.join(bigram))
      if j + 1 == merges[k]:
        write_bpe(lang, merges[k], str_corpus)
        k += 1
  return

if __name__ == "__main__":
  os.makedirs(join(bpedir, 'segmentations'), exist_ok=True)
  langs, bpe_models, corpora = load_data()
  apply_bpe(langs, bpe_models, corpora)
```

After this step, in the directory *../data/normal_bpe/segmentations/* various files exist with the following format:

- eng_100.bpe
- eng_200.bpe
- ...
- deu_100.bpe
- deu_200.bpe
- ...

deu or *eng* refers to the language, and the numbers are the amount of merges that have been applied into this file. For examples, see 4.1.2.

5.2.3 Extract alignments

This step requires some prior installation of the alignment software. The thesis has been conducted in a Linux environment, so the installation guide is adapted to this case. The installation of the software in any other OS is however possible, as long as the user adapts the commands to their OS. Initially, if not present in the system, it is mandatory to install Cmake. The installation steps for *Fastalign* are as follows, in bash. The path *path/to/project* should be the path to where the README, data folders and so on are present. Here, a new folder named *tools* is created to save the alignment algorithms.

```
sudo apt-get install libgoogle-perftools-dev libsparsehash-dev
cd /path/to/project
mkdir tools
cd tools
git clone https://github.com/clab/fast_align.git
cd fast_align
mkdir build
cd build
cmake ..
make
```

And for *Eflomal*:

```
cd /path/to/project/tools
git clone https://github.com/robertostling/eflomal.git
cd eflomal
make
sudo make install
python3 setup.py install
```

The paths for the installation are then saved in the global variable file, as well as the *mode* to make the alignments, either of the two algorithms.

```python
# settings.py
mode = "fastalign" #fastalign, eflomal
fastalign_path = join(rootdir, "tools/fast_align/build/fast_align")
atools_path = join(rootdir, "tools/fast_align/build/atools")
eflomal_path = join(rootdir, "tools/eflomal")
```

In this step, as a general notion, the extract alignments script takes two files as input: English BPE file, German BPE file, and outputs an alignment file, with the extension .wgdfa. First of all, it is necessary to iterate through the different merge types that have been done before. There are BPE files with 100 merges, 200, 500, etc for both languages. At each iteration, a different alignment file is created. Regarding alignment algorithms, they work on parallel data, that is, they expect text in the following format:

Hello from England ||| Hallo aus Deutschland

Since the BPE files do not have this format, they are actually separated into two files, namely *deu* and *eng* files, first of all the function *create_parallel_text* creates a *.txt* file in the appropriate parallel format.

```python
# extract_alignments.py
from os.path import join
import os
import sys
import codecs
# import global variables from settings.py
sys.path.insert(1, os.path.join(sys.path[0], ’..’))
```

```
from settings import *
from subword_word import *

def create_parallel_text(sourcepath: str, targetpath: str, outpath: str):
  fa_file = codecs.open(outpath + ’.txt’, "w", "utf-8")
  fsrc = codecs.open(sourcepath, "r", "utf-8")
  ftrg = codecs.open(targetpath, "r", "utf-8")
  for sl, tl in zip(fsrc, ftrg):
    sl = sl.strip().split("\t")[-1]
    tl = tl.strip().split("\t")[-1]
    fa_file.write(f"{sl} ||| {tl}\n")
  fa_file.close()
  return
```

Fastalign and *eflomal* work by issuing a command on the OS terminal, and they generate forward and reverse alignments. This is handled by the *create_fwd_rev_files* function, which creates *.fwd* and *.rev* files with the corresponding file name.

```
# extract_alignments.py
def create_fwd_rev_files(outpath: str):
  if mode == "fastalign":
    os.system(f"{fastalign_path} -i {outpath}.txt -v -d -o > {outpath}.fwd")
    os.system(f"{fastalign_path} -i {outpath}.txt -v -d -o -r > {outpath}.rev")
  elif mode == "eflomal":
    os.system(f"cd {eflomal_path}; python align.py -i {outpath}.txt --model 3 -f {outpath}.fwd -r {outpath}.rev")
  return
```

Given these *.fwd* and *.rev* files, the alignment algorithm creates a type of union between these two, called *grow-diag-final-and*, handled by the *create_gdfa_file* function, creating files with the extension *.gdfa*. The previously generated files, *.fwd*, *.rec*, *.txt* and the intermediate file *_unnum.gdfa* are deleted from the system.

```
# extract_alignments.py
def create_gdfa_file(outpath: str):
  # create gdfa file from .fwd and .rev
  comm = f"{atools_path} -i {outpath}.fwd -j {outpath}.rev -c grow-diag-final-and > {outpath}_unnum.gdfa"
  os.system(comm)

  # parse _unnum.gdfa to .gdfa with "\t" separator
  with codecs.open(f"{outpath}_unnum.gdfa", "r", "utf-8") as fi:
    with codecs.open(f"{outpath}.gdfa", "w", "utf-8") as fo:
      for i, line in enumerate(fi):
        fo.write(f"{i}\t{line.strip()}\n")

  # delete unnecessary files
  comm = f"rm {outpath}_unnum.gdfa; rm {outpath}.fwd; rm {outpath}.rev; rm {outpath}.txt"
  os.system(comm)
  return
```

As explained in the *Extract alignment* subsection in the Methodology chapter 4.1.3, the alignment aligns whitespace-separated units, which are generally words but in this case are subword, or BPE units. In order to actually make alignments between words, the subword alignments need to be transformed into word alignments. The function *load_and_map_segmentations* loads the BPE files and maps each BPE unit to its corresponding word. The comments on the function display a simple example to illustrate this concept. This is an auxiliary function in order to map the alignments later. Afterwards, by calling *bpe_word_align*, the mapping from subword alignments to word alignments is made. Lastly, the new alignments are saved in a file with the extension *.wgdfa*.

```
# extract_alignments.py
def load_and_map_segmentations(num_symbols: int):
  '''
  Given a .bpe file composed of the corpus made of subword units such as
  corpus_eng = [
    '_We _do _no t _be li eve _.',
    '_Thi s _is _a _sent ence _.',
    ...
  ]
  Output: dictionary of each language and
  a list of indexes pointing to which word each element (_do) belongs to
  bpes = {
    'eng':[
      [0, 1, 2, 2, 3, 3, 3, 4],
      [0, 0, 1, 2, 3, 4, 5],
      ...
    ],
    ...
  }
  '''
  bpes = {}
  for lang in [source, target]:
    bpes[lang] = []
    corpus = codecs.open(lang+'_'+str(num_symbols)+'.bpe', encoding='utf-8')
    for sent in corpus:
      mapping = [0]
      i = 0
      for subw in sent.split()[1:]:
        if subw[0] == word_sep:
          i += 1
        mapping.append(i)
      bpes[lang].append(mapping)
  return bpes

def bpe_word_align(bpes: dict, bpe_aligns: list) -> str:
  '''
  Input: dictionary of bpes obtained as output of map_subword_to_word()
  Output: list of word alignments and their indexes
    "
      0   0-0 0-1 1-1 1-2 3-1 2-4 \n
      1   0-0 1-0 1-1 2-1 \n
      ...
    "
  '''
  all_word_aligns = ''
  for i, (sent1, sent2, bpe_al) in enumerate(zip(bpes[source], bpes[target],
                                                  bpe_aligns)):
    word_aligns = set()
    # iterate each alignment
    for al in bpe_al.split('\t')[1].split():
      firstal, secondal = al.split('-')
      new_al = str(sent1[int(firstal)]) + '-' + str(sent2[int(secondal)])
      word_aligns.add(new_al)
    all_word_aligns += str(i) + "\t" + ' '.join(word_aligns) + "\n"
  return all_word_aligns
```

The alignment algorithm is executed for two types of files. Firstly, for the raw corpus itself, the alignment algorithm is executed to align the English raw corpus with the German raw corpus, which will serve as baseline to check how good the BPE merges are. And then, the alignment algorithm is executed for the BPE files themselves.

```
# extract_alignments.py
def extract_alignments(input_mode=False: bool):
  for num_symbols in merges:
    if input_mode:
      print("Alignments for input files")
      sourcepath = inputpath[source]
      targetpath = inputpath[target]
      outpath = join(bpedir, mode, "input")
    else:
      print(f"Alignments for {num_symbols} symbols")
      sourcepath = join(bpedir, 'segmentations', f"{source}_{num_symbols}.bpe")
      targetpath = join(bpedir, 'segmentations', f"{target}_{num_symbols}.bpe")
      outpath = join(bpedir, mode, str(num_symbols))
    create_parallel_text(sourcepath, targetpath, outpath)
    create_fwd_rev_files(outpath)
    create_gdfa_file(outpath)
    # map alignment from subword to word
    bpes = load_and_map_segmentations(num_symbols)
    all_word_aligns = bpe_word_align(bpes, codecs.open(o+'.gdfa', encoding='utf-8'))
    os.system(f"rm {outpath}.gdfa")
    codecs.open(outpath+'.wgdfa', 'w', encoding='utf-8').write(all_word_aligns)
  return

if __name__ == "__main__":
  os.makedirs(join(bpedir, mode), exist_ok=True)
  if not os.path.isfile(join(bpedir, mode, 'input.wgdfa')):
    extract_alignments(input_mode=True)
  extract_alignments()
```

5.2.4 Calculate alignment scores

At this point in the pipeline, the alignments between subword units are obtained, mapped into word alignments, and the last step to be performed is to calculate the alignment scores. First of all, the gold alignment file is loaded and its alignments extracted.

```
# calc_align_scores.py
import os
from os.path import join
import sys
import glob
import random
import collections
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# import global variables from settings.py
sys.path.insert(1, os.path.join(sys.path[0], ’..’))
from settings import *

def load_gold(g_path: str) -> (dict, dict, float):
  gold_f = open(g_path, "r")
  pros = {}
  surs = {}
  all_count = 0.
  surs_count = 0.
  for line in gold_f:
    line = line.strip().split("\t")
    line[1] = line[1].split()
    pros[line[0]] = set()
    surs[line[0]] = set()
    for al in line[1]:
      pros[line[0]].add(al.replace(’p’, ’-’))
      if ’p’ not in al:
        surs[line[0]].add(al)
    all_count += len(pros[line[0]])
    surs_count += len(surs[line[0]])
  return pros, surs, surs_count
```

The next function, given an input path, calculates the precision, recall, F1 and AER score based on the gold standard.

```
# calc_align_scores.py
def calc_score(input_path: str, probs: dict, surs: dict, surs_count: float) -> (float,
                                                   float, floatl float):
  total_hit, p_hit, s_hit = 0., 0., 0.
  target_f = open(input_path, "r")
  for line in target_f:
    line = line.strip().split("\t")
    if line[0] not in probs: continue
    if len(line) < 2: continue
    line[1] = line[1].split()
    for pair in line[1]:
      if pair in probs[line[0]]:
        p_hit += 1
      if pair in surs[line[0]]:
        s_hit += 1
      total_hit += 1
  target_f.close()
  y_prec = round(p_hit / max(total_hit, 1.), 3)
  y_rec = round(s_hit / max(surs_count, 1.), 3)
  y_f1 = round(2. * y_prec * y_rec / max((y_prec + y_rec), 0.01), 3)
  aer = round(1 - (s_hit + p_hit) / (total_hit + surs_count), 3)
  return y_prec, y_rec, y_f1, aer
```

This step is done for the baseline, to obtain a measure of the standard version of the system, that is, the raw English and German corpora, and then for the BPE files itself.

```
# calc_align_scores.py
columns = [’num_symbols’, ’prec’, ’rec’, ’f1’, ’AER’]

def get_baseline_score(probs: dict, surs: dict, surs_count: int) -> pd.DataFrame:
  alfile = join(bpedir, mode, ’input.wgdfa’)
  score = [0]
  score.extend(list(calc_score(alfile, probs, surs, surs_count)))
  baseline_df = pd.DataFrame([score], columns=columns).round(decimals=3)
  return baseline_df

def calc_align_scores(probs: dict, surs: dict, surs_count: int):
  scores = []
  for num_symbols in merges:
    alfile = join(bpedir, mode, f"{num_symbols}.wgdfa")
    score = [int(num_symbols)]
    score.extend(list(calc_score(alfile, probs, surs, surs_count)))
    scores.append(score)
  df = pd.DataFrame(scores, columns=columns).round(decimals=3)
  return df
```

The scores for the baseline and the corresponding BPE files are obtained and stored in a DataFrame data structure. Next, this data is plotted and saved into *.png* and *.csv* files.

```
# calc_align_scores.py
def plot_scores(df: pd.DataFrame, baseline_df: pd.DataFrame, scoredir: str):
  # Use plot styling from seaborn.
  sns.set(style=’darkgrid’)
  # Increase the plot size and font size.
  sns.set(font_scale=1.5)
  plt.rcParams["figure.figsize"] = (12, 6)
  plt.clf()
  ax = plt.gca() # gca stands for ’get current axis’
  colors = [’magenta’, ’tab:blue’, ’tab:green’, ’tab:red’]
  df = df.sort_values(’num_symbols’)
  columns = list(df)
  for column, color in zip(columns[1:], colors):
    df.plot(kind=’line’, x=columns[0], y=column, color=color, ax=ax)
  for baseline_results, color in zip(list(baseline_df.iloc[0][1:]), colors):
    plt.axhline(y=baseline_results, color=color, linestyle=’dashed’)
  plt.savefig(join(scoredir+’.png’))
  return


```

```python
if __name__ == "__main__":
  # Calculate alignment quality scores based on the gold standard.
  # The output contains Precision, Recall, F1, and AER.
  probs, surs, surs_count = load_gold(goldpath)
  baseline_df = get_baseline_score(probs, surs, surs_count)
  df = calc_align_scores(probs, surs, surs_count, baseline_df)
  scorename = join(scoredir, 'scores')
  print(f"Scores saved into {scorename}")
  df.to_csv(scorename+'.csv', index=False)
  plot_scores(df, baseline_df, scorename)
```

5.3 Replication of BPE-dropout

The previous section has laid the backbone of the algorithms, the various files and functions. In this and the following sections, some modifications and improvements are introduced to the already existing Python scripts. For the sake of simplicity, the code snippets that follow only include the new changes, or the functions from the last section with new modifications, and the modifications are introduced by signalling the line numbers from previous scripts in which changes are introduced. For example, it might be mentioned that a certain line is added to line 21 of a certain function in a specific script. This refers to the function introduced in the previous section, and it will be referred as such. In the case that a function is altered in a significant way, the new version of the function will be shown, which overrules the previous version. The functions that remain unaltered are not shown.

When adding dropout to BPE, three new parameters come into play in the project, namely **dropout** rate, **dropout_samples**, that is, how many samples of the dropout system are considered, and **merge_threshold**, which serves its function with alignments later on. These values are saved into *settings.py*, as well as new data directories to store files resulting from BPE-dropout.

```python
# settings.py

dropout = 0.1
dropout_samples = 10
merge_threshold = [0.3, 0.5, 0.7, 0.9]

bpedir = join(rootdir, 'data', 'dropout_bpe' if dropout > 0 else 'normal_bpe')
scoredir = join(rootdir, 'reports', 'scores_' + ('dropout_bpe' if dropout > 0 else 'normal_bpe'))
```

5.3.1 Apply-BPE to corpus with dropout

The first modifications in the project occur in *apply_bpe.py*, where some merges are skipped, and the process is repeated 10 times. More theoretical insights regarding this approach can be found in the Methodology section 4.2. The function *apply_bpe* includes two new lines where a random number between 0 and 1 is generated. If this number is smaller than the *dropout* rate saved in *settings.py*, then that merge is not considered and the loop skips it. This means that if the *dropout* variable has the value of 0.1, 10% of merges are skipped.

Additionally, in the main function, the function *apply_bpe* is called *dropout_samples* times. To save the files accordingly, a new variable is introduced, namely *i*, that does nothing in the case where dropout=0, but when repeating the process, for instance if lang=eng, num_symbols=2000, and first iteration of dropout, that is, i=0, the files are saved as *eng_2000_0.bpe* instead, and so on for further iterations. This is effectively achieved by changing the variable *outputpath*. In the function *write_bpe* in the general pipeline 5.2.2, the following modifications are made:

```
# apply_bpe.py
import random

def write_bpe(lang: str, num_symbols: int, merged_corpus: str, i: int =-1):
  outputpath = join(bpedir, ’segmentations’, f"{lang}_{num_symbols}{f’_{i}’ if i != -1 else ’’}.bpe")
  argsoutput = codecs.open(outputpath, ’w’, encoding=’utf-8’)
  argsoutput.write(merged_corpus)
  return

def apply_bpe(langs: list, bpe_models: list, corpora: list, i: int =-1):
  for lang, bpe_model, corpus in zip(langs, bpe_models, corpora):
    bpe_model = bpe_model[:max(merges)]
    merges_copy = merges.copy()
    str_corpus = ’\n’.join(corpus)
    for j, bigram in enumerate(bpe_model):
      if random.uniform(0, 1) < dropout:
        continue
      str_corpus = str_corpus.replace(’ ’.join(bigram), ’’.join(bigram))
      if j + 1 == merges_copy[0]:
        write_bpe(lang, merges_copy.pop(0), str_corpus, i)
  return

if __name__ == "__main__":
  langs, bpe_models, corpora = load_data()
  if dropout > 0:
    for i in range(dropout_samples):
      apply_bpe(i)
  else:
      apply_bpe()
```

5.3.2 Extract alignments with dropout

The only change in this step is that the *extract_alignment* 5.2.3 function is called *dropout_samples* times, which changes the function to write the alignments in the new format, namely, changing the variables *sourcepath* and *targetpath*. Additionally, the gold standard's alignments do not need to be calculated since the baseline are the BPE scores rather than the gold standard scores. The rest of the alignment algorithm remains unchanged.

```
# extract_alignments.py
# change line 2
def extract_alignments(i: int =-1, input_mode: bool =False):

# change lines 11 and 12
sourcepath = join(bpedir, ’segmentations’, f"{source}_{num_symbols}_{’_’+str(i) if dropout else ’’}.bpe")
```

```
targetpath = join(bpedir, 'segmentations', f"{target}_{num_symbols}_{'_'+str(i) if dropout else ''}.bpe")

# change line 30
if dropout > 0:
  for i in range(dropout_samples):
    extract_alignments(i)
else:
  extract_alignments()
```

5.3.3 Calculate alignment scores with dropout

As explained in the Methodology section 4.2, variants of union, intersection and threshold are created. This is introduced with a new script, namely *merge_dropout.py*. First of all, the function *merge_dropout_alignments* opens all alignment files and creates a dictionary data structure with the union, intersection and threshold alignment files and saves them into *X_union.wgdfa*, *X_inter.wgdfa*, *X_thres.wgdfa* respectively.

```
# merge_dropout.py
import os
from os.path import join
import sys
import codecs
import pandas as pd
from tqdm import tqdm
from collections import Counter
# import global variables from settings.py
sys.path.insert(1, os.path.join(sys.path[0], '..'))
from settings import *
from calc_align_score import *

def merge_dropout_alignments():
  union_merge, inter_merge, thres_merge = {}, {}, {}
  for num_symbols in tqdm(merges, desc=f"merge_dropout: dropout={dropout}, union, inter, thres"):
    union_merge[num_symbols], inter_merge[num_symbols], thres_merge[num_symbols] = [], [], []
    for i in range(dropout_sampless):
      for j, line in enumerate(open(f'{num_symbols}_{i}.wgdfa', 'r').readlines()):
        al = frozenset(line.strip().split("\t")[1].split())

        # at the first iteration, just append the alignment
        if i == 0:
          union_merge[num_symbols].append(al)
          inter_merge[num_symbols].append(al)
          thres_merge[num_symbols].append(Counter(al))
          continue

        # do union, intersection or frequency addition
        union_merge[num_symbols][j] |= al
        inter_merge[num_symbols][j] &= al
        thres_merge[num_symbols][j] += Counter(al)

    # write to output
    unionfile = codecs.open(f'{num_symbols}_union.wgdfa', 'w')
    interfile = codecs.open(f'{num_symbols}_inter.wgdfa', 'w')
    thresfiles = {merge_t: codecs.open(f'{num_symbols}_thres_{merge_t}.wgdfa', 'w')
                                               for merge_t in merge_threshold}
    for i in range(len(union_merge[num_symbols])):
      unionfile.write(f"{i}\t{' '.join(union_merge[num_symbols][i])}\n")
      interfile.write(f"{i}\t{' '.join(inter_merge[num_symbols][i])}\n")
      # get alignments more common than the merge_threshold %
      for merge_t in merge_threshold:
        common_aligns = [k for k in thres_merge[num_symbols][i]
                        if thres_merge[num_symbols][i][k] > merge_t * dropout_samples]
        thresfiles[merge_t].write(f"{i}\t{' '.join(common_aligns)}\n")
  return
```

Afterwards, the function *calc_score_merges* opens these files and calculates the score, much in the way as the *calc_align_score* algorithm from the previous section.

```
# merge_dropout.py
def calc_score_merges():
  probs, surs, surs_count = load_gold(goldpath)
  baseline_df = pd.read_csv(join(baselinedir, f'scores_{source}_{target}.csv'))
  scorespath = join(scoredir, str(dropout))
  if not os.path.isdir(scorespath):
    os.mkdir(scorespath)

  for merge_type in ['union', 'inter']:
    scores = []
    for num_symbols in merges:
      mergefilepath = join(bpedir, mode, f'{num_symbols}_{merge_type}.wgdfa')
      score = [int(num_symbols)]
      score.extend(list(calc_score(mergefilepath, probs, surs, surs_count)))
      scores.append(score)

    df = pd.DataFrame(scores, columns=columns).round(decimals=3)
    scorename = join(scorespath, 'scores', merge_type)

    print(f"Scores saved into {scorename}")
    df.to_csv(scorename+'.csv', index=False)
    plot_scores(df, baseline_df, scorename)

  # threshold case, iterate all merge_thresholds saved
  for merge_t in merge_threshold:
    scores = []
    for num_symbols in merges:
      mergefilepath = join(bpedir, mode, f'{num_symbols}_thres_{merge_t}.wgdfa')
      score = [int(num_symbols)]
      score.extend(list(calc_score(mergefilepath, probs, surs, surs_count)))
      scores.append(score)

    df = pd.DataFrame(scores, columns=columns).round(decimals=3)
    scorename = join(scorespath, ’scores’, f"{merge_t}_thres")

    print(f"Scores saved into {scorename}")
    df.to_csv(scorename+’.csv’, index=False)
    plot_scores(df, baseline_df, scorename)
  return

if __name__ == "__main__":
    merge_dropout_alignments()
    calc_score_merges()
```

5.4 Improvement of learn-BPE algorithm

As explained in the methodology 4.3, the main improvement in the learn-BPE algorithm is to only update previous and next tokens to the merged pair, as well as saving the indexes where each pair occurs. These improvements are built on top of the code shown in 5.2.1.

In the *learn_bpe* function, the new *update_tokens* returns the updated pairs and the merged tokens, all in one step. The function *get_stats*, which is the function that iterates the whole corpus, only has to be performed once. This is however a modification of the previous *get_stats* function, since it computes the indexes of the pairs as well.

```
# learn_bpe.py
def learn_bpe(corpus: list, bpe_model: list): list:
  ’’’
  Learn BPE operations from vocabulary.
  Steps:
  1. split corpus into characters, count frequency
  2. count bigrams in corpus
  3. merge most frequent symbols
  4. Update bigrams in corpus
  ’’’
  tokens = read_corpus(corpus)
  pairs, idx = get_stats(tokens)
  most_frequent_merges = []
  for i in range(learn_merges):
    try:
      most_frequent = pairs.most_common(1)[0][0]
    except:
      # pairs is empty
      break
    most_freq_merges.append(most_frequent)
    tokens, idx, pairs = update_tokens(tokens, idx, pairs, most_frequent)
  return most_freq_merges
```

These are the modifications introduced in the *get_stats* function so that it saves the pair indexes. In the *idx* data structure, not only is the index saved, but also the amount of appearances of each pair in that sentence. This is done to ensure that if the pair (’t’, ’h’) in index 0 now becomes (’t’, ’he’) because of the (’h’, ’e’) merge, and the frequency of (’t’, ’h’) is reduced by one, it might be assumed that (’t’, ’h’) no longer appears in that sentence. But this would be a mistake, since there might be other instances of

(’t’, ’h’) in the sentence that are not altered by this merge. We only want to say that (’t’, ’h’) no longer appears in index 0 when all instances of (’t’, ’h’) have been merged with other sequences. This is the motivation behind storing the amount of appearances of each pair in a sentence.

```python
# learn_bpe.py
def get_stats(tokens: list) -> (Counter, dict):
  """
  Count frequency of all bigrams, their indexes and the frequency per index.
  pairs = {
    ('s', 'h'): 5,
    ('h', 'e'): 6
  }
  idx = {
    ('t', 'h'): {
      # keys are indexes in corpus, values are frequency of appearance
      0: 2,
      1: 3,
    }
  }
  """
  def get_pairs_idx(pairs, idx, symbols):
    symbols = symbols.split()
    for j in range(len(symbols) - 1):
      new_pair = symbols[j], symbols[j + 1]
      pairs[new_pair] += 1
      idx[new_pair][i] += 1
    return pairs, idx

  pairs = Counter()
  idx = defaultdict(lambda: defaultdict(int))
  for i, sent in enumerate(tokens):
    # get stats for each word independently, no bigrams between different words
    for word in sent[1:].split(' '+word_sep):
      pairs, idx = get_pairs_idx(pairs, idx, word_sep + word)
  return pairs, idx
```

The *update_tokens* function handles the situation where only the previous and after tokens are updated for each merged pair. Comments are included in the code for readability.

```python
# learn_bpe.py
def update_tokens(tokens, idx, pairs, pair):

  def update_freqs(pairs, idx, pair, new_pair=-1):
    # decrease freq from pairs
    pairs[pair] -= 1
    if pairs[pair] <= 0: del pairs[pair]
    # decrease freq from idx
    idx[pair][i] -= 1
    if idx[pair][i] <= 0: del idx[pair][i]
    if len(idx[pair]) <= 0: del idx[pair]
    if new_pair != -1:
      pairs[new_pair] += 1
      idx[new_pair][i] += 1
    return pairs, idx

  merged_pair = ''.join(pair)
  p = re.compile(r'(?<!\S)' + re.escape(' '.join(pair)) + r'(?!\S)')
  # only iterate the corpus indexes where the pair to be merged is present
  for i in list(idx[pair]).copy():

    # merge pair in the sentence
    sent = p.sub(merged_pair, tokens[i])
    # sentence remains unchanged. Delete pair from pairs and idx and continue
    if sent == tokens[i]:
      del pairs[pair]
      del idx[pair][i]
      if len(idx[pair]) <= 0:
        del idx[pair]
      continue
    tokens[i] = sent
    '''
    iterate sent by the position the merged_pair occurs.
    in each position, we need to reduce freq of previous and after tokens
    sentence before merge: 'h e l l o', pair: ('e', 'l')
    merged sent = 'h el l o'
    sent.split(merged_pair) -> ['h ', ' l o']
    we iterate the splitted sentence and in each occasion
    * decrease freq of previous token ('h', 'e')
        * create new token ('h', 'el')
    * decrease freq of after token ('l', 'l')
        * create new token ('el', 'l')
    * decrease freq of merged pair ('e', 'l')
    '''
    sent = sent.split(merged_pair)
    for k in range(len(sent[:-1])):
      if sent[k].split() and sent[k][-1] == ' ' and word_sep not in pair[0][0]:
        '''
        conditions to update the **previous** token:
        * if sent[k] is not empty. if it is, there's no previous token to update.
        * if the merged_pair is not the beginning of the word.
          * in this case, discard the last letter from the prev word to be merged with ...
          * our current pair. ... e _t h ... ('e', '_t') is discarded
        '''
        prev = (sent[k].split()[-1], pair[0])
        new_pair = (prev[0], merged_pair)
        pairs, idx = update_freqs(pairs, idx, prev, new_pair)

      if not sent[k+1].split() and word_sep not in pair[0][0]:
        '''
        conditions to update the **after** token when merged bigrams are consecutive:
        * when the pair's first character is not the beginning of the word
        * and when the next token is empty
        * there are consecutive merged pairs, merged_pair=('ssi'), sent='m i ssi ssi p p i'
```

```
            * in this case, delete the token between the merged_pair: ('i', 's')
            * and create a new pair ('ssi', 'ssi')
        '''
        if sent[k] and sent[k][-1] == word_sep:
          after = (word_sep+merged_pair, pair[0])
          new_pair = -1
        else:
          after = (pair[1], pair[0])
          new_pair = (merged_pair, merged_pair)
          pairs, idx = update_freqs(pairs, idx, after, new_pair)
      elif sent[k+1].split() and word_sep not in sent[k+1].split()[0]:
        '''
        conditions to update the **after** token in a more general case:
        * if sent[k] is not empty. if it is, there's no after token to update.
        * if the after token is a new word, we do not want to consider it.
        '''
        after = (pair[1], sent[k+1].split()[0])
        new_pair = (merged_pair, after[1])
        pairs, idx = update_freqs(pairs, idx, after, new_pair)
      # decrease freq of merged bigram
      pairs, idx = update_freqs(pairs, idx, pair)
  return tokens, idx, pairs
```

The performance improvements of this approach can be seen in the Results section.

5.5 BPE without word boundaries

As explained in the Methodology chapter, this addition considers BPE units between different words. The only change occurs when learning these BPE units and mapping multiple-word-units to multiple-word-units, the rest of the pipeline remains as is, other than changing the filenames and paths of the alignments and scores. A new Boolean variable is introduced in *settings.py*, so that by changing it to a positive value will make the whole pipeline act in space mode, that is, setting the space boundary and only allowing merges between words. On the contrary, no space mode allows merges between different words.

```
# settings.py
space = False
```

The following scripts for learning BPEs and extracting and mapping alignments are generalized in the sense that just by changing the Boolean variable in the settings file, the pipeline runs automatically and no further changes need to be done. The code snippets below display these functionality, accepting both modes.

In *learn_bpe.py*, the way to parse and tokenize the corpus is slightly different, now instead of having a special token to denote the beginning of a word, the special token denotes the whitespace.

```
# learn_bpe.py
def read_corpus(corpus: list) -> list:
  '''
  Read corpus, strip index and new line characters.
  In space mode, each word has a word_sep symbol at the beginning of the word.
  tokens = [
    ’w e \_d o \_n o t \_b e l i e v e \_t h a t \_w e \_s h o u l d.’,
  ]
  In no space mode, words are joined by word_sep.
  tokens = [
    ’w e \_ d o \_ n o t \_ b e l i e v e \_ t h a t \_ w e \_ s h o u l d .’,
  ]
  ’’’
  tokens = []
  for line in corpus:
    line = line.split(’\t’)[1].strip(’\r\n ’)
    line = line.split()
    line[0] = str.lower(line[0])
    if space:
      # add word_sep to each beginning of word and join by space
      tokens.append(’ ’.join([word_sep + ’ ’.join(word) for word in line]))
    else:
      # join all words by word_sep
      tokens.append(u’ \u2581 ’.join([’ ’.join(word) for word in line]))
  return tokens
```

When analysing pairs and their frequencies, each word’s last character and the following whitespace has to be considered, as well as this whitespace and the following word’s first character. This changes the *get_stats* function slightly.

```
# learn_bpe.py
def get_stats(tokens: list) -> (Counter, dict):
  ’’’
  Count frequency of all bigrams, their indexes and the frequency per index.
  pairs = {
    (’s’, ’h’): 5,
    (’h’, ’e’): 6
  }
  idx = {
    (’t’, ’h’): {
      # keys are indexes in corpus, values are frequency of appearance
      0: 2,
      1: 3,
    }
  }
  In space mode, the last token ’.’ or word_sep. is not merged with anything.
  ’’’
  def get_pairs_idx(pairs, idx, symbols):
    symbols = symbols.split()
    for j in range(len(symbols) - 1):
      new_pair = symbols[j], symbols[j + 1]
      pairs[new_pair] += 1
      idx[new_pair][i] += 1
    return pairs, idx

  pairs = Counter()
  idx = defaultdict(lambda: defaultdict(int))
  for i, sent in enumerate(tokens):
    if space:
      # get stats for each word independently, no bigrams between different words
      for word in sent[1:].split(u'\u2581'):
        pairs, idx = get_pairs_idx(pairs, idx, word_sep + word)
    else:
      # get bigram stats for the whole sentence
      pairs, idx = get_pairs_idx(pairs, idx, sent)
  return pairs, idx
```

And finally, after merging a pair in the corpus, when updating the tokens and the pairs, the fact that space boundaries exist or not also plays a role. Specifically, the no space mode enjoys more freedom, in the sense that no matter which pair is merged, the pair immediately before and the pair immediately after are merged, no matter what. In space mode however, this cannot be done if the previous or next pair belong to another word, which puts some restrictions. Only modified lines with respect to the code in the previous improve learn BPE algorithm 5.4. There are comments in the code for readability.

```
# learn_bpe.py
def update_tokens(tokens: list, idx: dict, pairs: Counter, pair: tuple) -> (list, dict, Counter):

# change line 46
if sent[k].split() and (sent[k][-1] == ' ' and word_sep not in pair[0][0] if space else True):
  '''
  conditions to update the **previous** token:
  * in space mode, if the merged_pair is not the beginning of the word.
    * in this case, discard the last letter from the prev word to be merged ...
    * with our current pair. ... e _t h ... ('e', '_t') is discarded
  '''

# change line 58
if space and not sent[k+1].split() and word_sep not in pair[0][0]:
  '''
  conditions to update the **after** token when merged bigrams are consecutive:
  * in space mode, when the pair's first character is not the beginning of the word
  '''

# change line 74
elif sent[k+1].split() and (word_sep not in sent[k+1].split()[0] if space else True):
  '''
  * in space mode, if the after token is a new word, we do not want to consider it.
  '''
```

The *apply_bpe.py* remains unchanged, aligning BPE files as well with the *Fastalign* or *Eflomal* algorithm as well. However, since now the alignments are among many words, the mapping is different than in the space case. The mapping is divided into subword to word mapping (map_subword_.to_word function) and multiword to word (map_multiword_to_word).

```
# extract_alignments.py
def map_subword_to_word(corpus, bpes, lang):
  '''
  SPACE MODE
  Input: list of sentences with subword separation
  corpus =  [
    '_We _do _no t _be li eve _.',
    '_Thi s _is _a _sent ence _.',
  ]
  Output: dictionary of each language and
  a list of indexes pointing to which word each element (_do) belongs to
  bpe = {
    source:
    [
      [0, 1, 2, 2, 3, 3, 3, 4],
      [0, 0, 1, 2, 3, 4, 5],
    ],
  }
  '''
  bpes[lang] = []
  for sent in corpus:
    mapping = [0]
    i = 0
    for subw in sent.split()[1:]:
      if subw[0] == word_sep:
        i += 1
      mapping.append(i)
    bpes[lang].append(mapping)
  return bpes
```

And the new *map_multiword_to_word* function.

```
# extract_alignments.py
def map_multiple_to_word(corpus, bpes, lang):
  '''
  NO SPACE MODE
  Input: list of sentences with subword separation
  corpus =  [
    'b u t_this_is_no t_w hat_hap pen s_.',
    'th e_ ice_cre am_.',
  ]
  Output: dictionary of each language and
  a list of indexes pointing to which word each element (t\_w) belongs to
  bpes = {
    source:
    [
      [[0], [0], [0,1,2,3], [3,4], [4,5], [5], [5,6]],
      [[0], [0], [1,2], [2]],
    ],
  }
  '''
  bpes[lang] = []
  for sent in corpus:
    sent_bpes = []
    j = 0
    for word in sent.split():
      if word == word_sep:
        # word is simply '_', doesn't belong to anything
        j += 1
        sent_bpes.append([])
        continue
      word_count = word.count(word_sep)
      if word_count == 0:
        sent_bpes.append([j])
        continue
      # multiple words in the element: t_this_is_no -> [0,1,2,3]
      if word[0] == word_sep:
        # word starts with '_' but there are no elements of the previous word in it
        j += 1
        word_count -= 1
      if word[-1] == word_sep:
        # word ends with '_' but there are no elements of the next word in it
        sent_bpes.append(list(range(j, j + word_count)))
      else:
        sent_bpes.append(list(range(j, j + word_count + 1)))
      j += word_count
    bpes[lang].append(sent_bpes)
  return bpes
```

Consequently, the function *load_and_map_segmentations* is altered in the following way:

```
# extract_alignments.py
def load_and_map_segmentations(num_symbols, i=-1):
  bpes = {}
  for lang in [source, target]:
    segmentpath = lang+'_'+str(num_symbols)+('_'+str(i) if i != -1 else '')+'.bpe'
    argsinput = codecs.open(segmentpath, encoding='utf-8')
    if space:
        bpes = map_subword_to_word(argsinput, bpes, lang)
    else:
        bpes = map_multiple_to_word(argsinput, bpes, lang)
  return bpes
```

Besides, the function *bpe_word_align* is also altered to account for the fact that if a multiword unit such as *from_the* is aligned with *aus_dem*, each one-to-one alignment must be considered, that is, from-aus, from-dem, the-aus, and the-dem.

```
def bpe_word_align(bpes, bpe_aligns):
  '''
  Input: dictionary of bpes obtained as output of map_subword_to_word()
  Output: list of word alignments and their indexes
    "
      0   0-0 0-1 1-1 1-2 3-1 2-4 \n
      1   0-0 1-0 1-1 2-1 \n
    "
  '''
  all_word_aligns = ''
  for i, (sent1, sent2, bpe_al) in enumerate(zip(bpes[source], bpes[target],
                                             bpe_aligns)):
    word_aligns = set()
    # iterate each alignment
    for al in bpe_al.split(’\t’)[1].split():
      firstal, secondal = al.split(’-’)
      if space:
        new_al = str(sent1[int(firstal)]) + ’-’ + str(sent2[int(secondal)])
        word_aligns.add(new_al)
      else:
        for el1 in sent1[int(firstal)]:
          for el2 in sent2[int(secondal)]:
            new_al = str(el1) + ’-’ + str(el2)
            word_aligns.add(new_al)
    all_word_aligns += str(i) + "\t" + ’ ’.join(word_aligns) + "\n"
  return all_word_aligns
```

The final step of calculating scores does not change.

6 Results

This chapter shows the results and analysis for each step of the development of this thesis, ranging from score plots for BPE alignments, to algorithm runtimes. The scores display four different metrics, namely precision, recall, F1 score and AER, which have been explained previously in the Translation chapter. 3.2.3

6.1 Replication of BPE

When replicating BPE, the best BPE result using *Fastalign* as alignment algorithm is a F1 score of 0.609, obtained after 1000 BPE merges. The baseline has a F1 score of 0.6. Baseline in this situation refers to the scores when aligning the raw corpus, leaving the words untouched and with no BPE units.

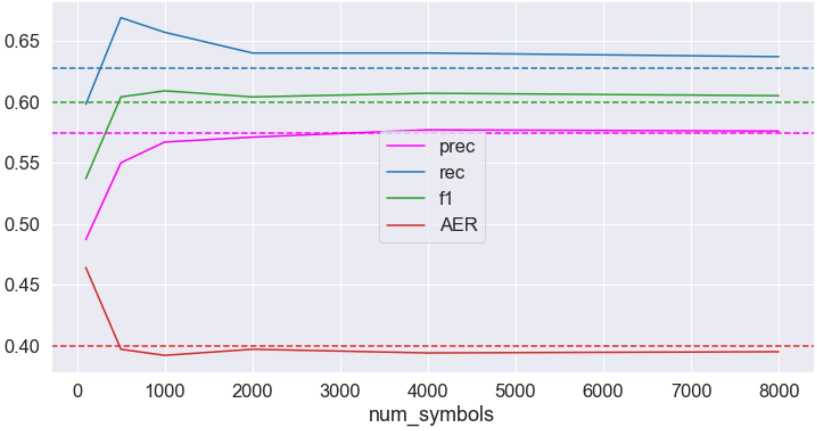


Figure 6.1: Scores of BPE over baseline, using Fastalign

In the case of *Eflomal*, the baseline F1 score is 0.72, and the best F1 score is 0.701, obtained after 6000 merges. When using *Eflomal*, BPE does not improve the baseline.

The alignment algorithms, *Fastalign* and *Eflomal*, require one language to be given as source and the other as target, and perform the alignment based on this. Throughout the thesis, English has been taken as the source language and German as the target. It might have been interesting to see that swapping the languages would yield different results, by swapping the languages in the global variables file. The difference between these two cases is barely perceptible, a difference of 0.03% throughout the scores across all numbers of merges. Therefore, it is assumed that swapping the source and target language has no effect in the experiment result scores.

6.2 Replication of BPE-dropout

BPE-dropout is meant to be an improvement of BPE, therefore the desired score comparison is that of BPE-dropout with respect to BPE. The BPE scores are taken as baseline, instead of the gold standard

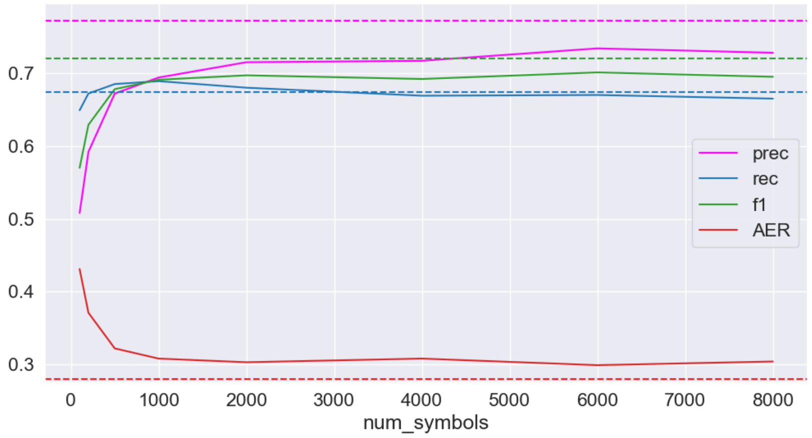


Figure 6.2: Scores of BPE over baseline, using Eflomal

scores as in the previous case. **The solid lines in the figures represent the BPE-dropout results, whereas the dotted lines correspond to BPE results**. As explained in the Methodology chapter 4.2, the BPE dropout pipeline is run a number of times, resulting in many possible segmentations. Out of these, three types of results are obtained: union, intersection and threshold.

- The **union** case takes all alignments into account, resulting in a big list of alignments per sentence. These most probably include the correct alignments, but there are many other wrong alignments. This case has low precision and high recall.
- The **intersection** case takes only those alignments which are present in all alignment files, resulting in a few number of alignments per sentence. Most probably, these alignments are correct, but there are also be many alignments missing. This case has high precision and low recall.
- The **threshold** case is a mixture between union and intersection. Given a threshold, for instance 0.7, an alignment is accepted if it is present in 70% of the alignment files. Smaller values for this variable resemble the union case, since more alignments are accepted. Higher threshold numbers resemble the intersection case, where alignments must be present in more and more files in order to be accepted.

The following figures 6.3, 6.4 and 6.5. show the results for a *dropout* rate of 10%, and the union, intersection and threshold cases.

It can be seen that the F1 score improves consistently the more BPE units have been merged. Since there are more merges, there are fewer units, most words are completely merged and the uncertainty of aligning BPE units is lower, since the sentence looks more and more similar to the raw sentence where there are just words. However, in no case does BPE-dropout improve BPE when union and intersection are considered exclusively. Regarding the threshold case, the following two figures show the scores for threshold 0.3 and 0.7, **which achieves the best F1 score n, namely 0.635, and an improvement of BPE of 3.5%**.

It is visible that the scores plot with the threshold at 0.3 resembles the union case more, and recall goes lower while the precision goes higher for larger threshold values. The optimal result is found at the

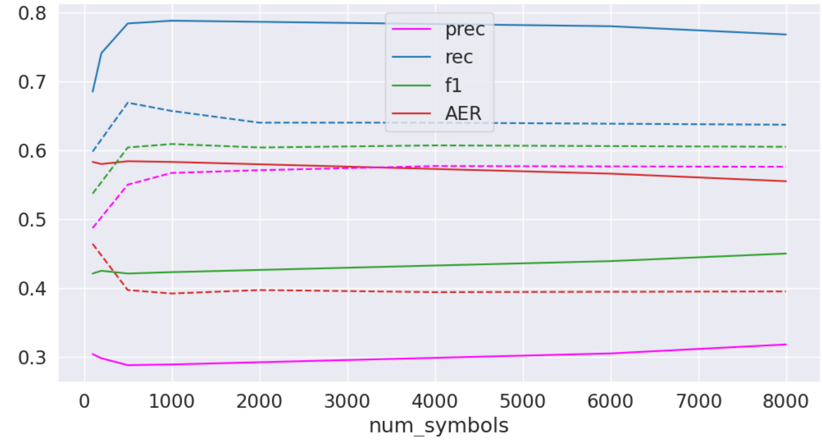


Figure 6.3: Scores for BPE-dropout with dropout rate 0.1, union mode

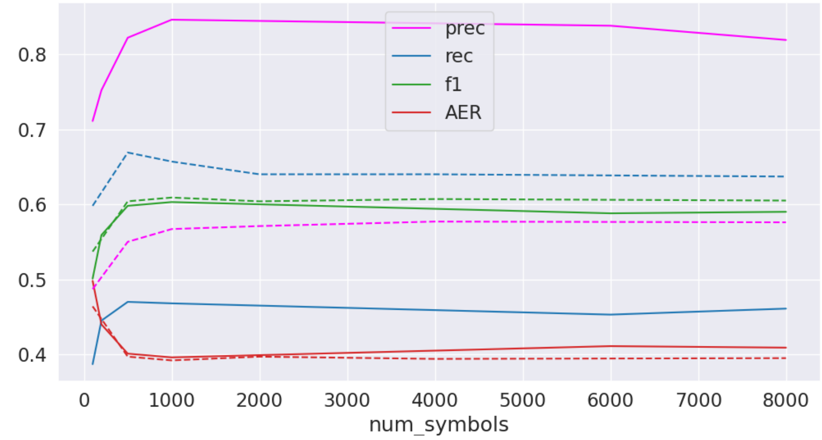


Figure 6.4: Scores for BPE-dropout with dropout rate 0.1, intersection mode

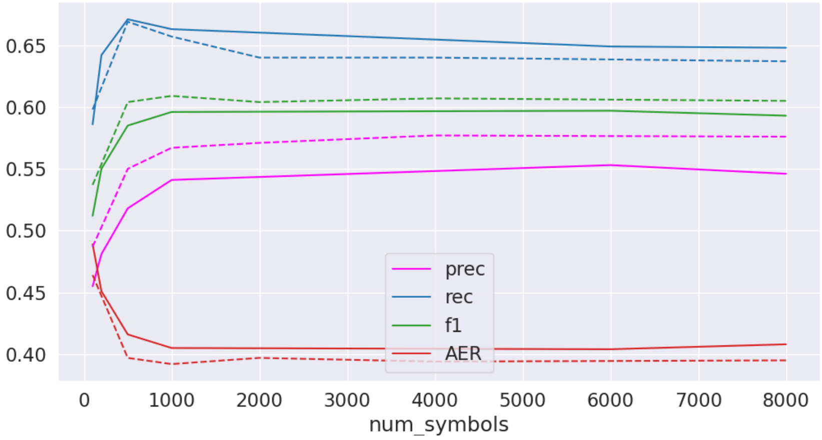


Figure 6.5: Scores for BPE-dropout with dropout rate 0.1, threshold mode at 0.3

threshold value of 0.7. This result is replicated when increasing the dropout rate up to 20% as well as 30%, and setting the alignment threshold at 0.5. **This shows that BPEs are robust to dropout**.

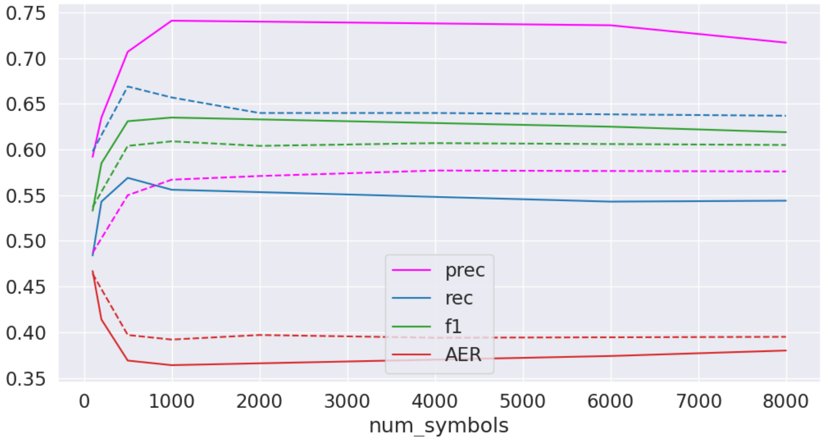


Figure 6.6: Scores for BPE-dropout with dropout rate 0.1, threshold mode at 0.7

In the BPE dropout paper [5], the authors only use one dropout percentage, namely 10% for all languages and 60% specifically for Chinese and Japanese to match the increase in length of segmented sentences for other languages. The paper hypothesizes that exposing a model to different segmentations might result in better understanding of the whole words as well as their subword units, which is proven by the paper and by this thesis as well. The paper authors also speculate the following:

> Results indicate that using BPE-Dropout on the source side is more beneficial than using it on the target side. We speculate it is more important for the model to understand a source sentence, than being exposed to different ways to generate the same target sentence.

This theory is also confirmed in this thesis. As the experiment of only applying BPE in English in the English-German language pair shown in the Figure 6.7, the improvement of BPE-dropout over BPE in this case is 3%.

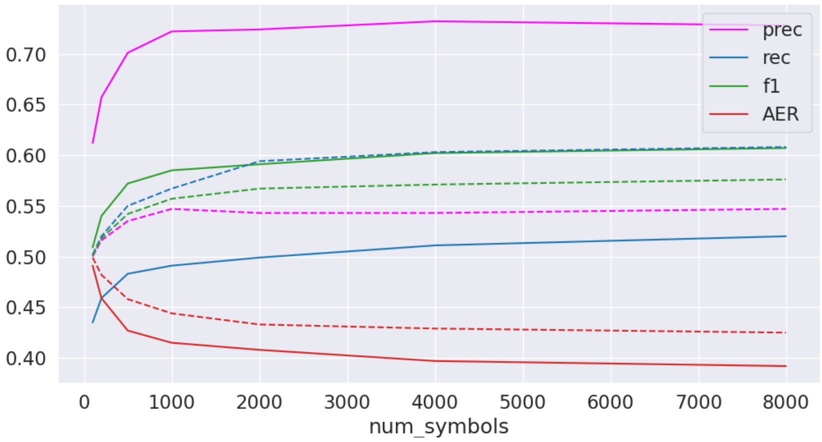


Figure 6.7: Scores for BPE-dropout with dropout rate 0.1, threshold at 0.7 and BPE only in source language

Regarding the improvements of BPE dropout over normal BPE, the authors conclude the following:

> The improvement with respect to normal BPE are consistent no matter the vocabulary size. But we see that the effect from using BPE-Dropout vanishes when a corpora size gets bigger.

This thesis has predominantly been run with a fixed size, relatively small corpus consisting of 10.000 sentences, whereas the paper authors have used much bigger corpora ranging from 133.000 sentences up to 2 million. As a result, the effect of varying corpus size has not been replicated. However, it can be seen that for bigger number of merges, the effects of BPE dropout are smaller, since given the bigger number of merges, the segmentations resemble the case where no dropout is applied.

As the BPE dropout paper states, *BPE dropout improves BPE consistently no matter how many merges are performed.* This result is replicated in this paper. Additionally, in the paper they use a fixed dropout rate and they give no information as per the threshold used in determining how to aggregate alignments. **By experimenting with different dropout and threshold rates, it was possible to improve the F1 score by selecting a dropout rate of 30%, which is specific to the corpus, in this case English and German, creating 30 segmentations and selecting an alignment threshold of 0.5. This results in a F1 score of 0.685, an improvement of 2% from the previous case**.

6.3 BPE-dropout for other language pairs

The better results of BPE-dropout over BPE proves that BPE-dropout performs better than BPE, at least in the English-German case, and the optimal hyperparameters for this case were using a dropout rate of 10%, an alignment threshold of 0.7, and around 2000 merges. The 2000 merge hyperparameter might be dependent on the corpus, smaller corpora might not even have 2000 merges to make, while bigger corpora might obtain their optimal results for more merges. **But does BPE-dropout outperform BPE for other language pairs at a dropout rate of 10% and an alignment threshold of 0.7?**

A couple of other language pairs were tested, namely English-Romanian, consisting of 50.000 sentences, and English-Hindi, with 3.000 sentences. Merge lists were created for English, Romanian and Hindi, each according to the particular case. For instance, the English merge list in the English-Romanian case is much richer than the merge list in the English-Hindi case, because the former has a much bigger corpus from which to learn frequent sequence pairs. Instead of using the rich English merge list obtained from the 50.000 sentences and applying it to all language pairs, namely German, Romanian and Hindi, it was decided that each language pair would see its own merge list created from the available corpus. The Romanian case, being the corpus around 16 times bigger, takes longer to compute segmentations and alignments but the results are consistent. **BPE-dropout outperforms BPE in two other language pairs for a dropout rate of 10%, an alignment threshold of 0.7 and with vastly different corpus sizes**. The BPE-dropout scores are shown in the following figures, the F1 score improvement over BPE for English-Romanian is of 1.5%, obtained at 4000 merges, and the F1 score improvement over BPE for English-Hindi is of 2.8%. It is worth noting that given the small size of the English-Hindi corpora, the merge list is only around 5000 merges long.

6.4 Improvement of the learn-BPE algorithm

By modifying the algorithm to update the corpus after a merge, by introducing indexes and frequency counts to reduce iteration, the learn-BPE algorithm's runtime was dramatically accelerated. In the

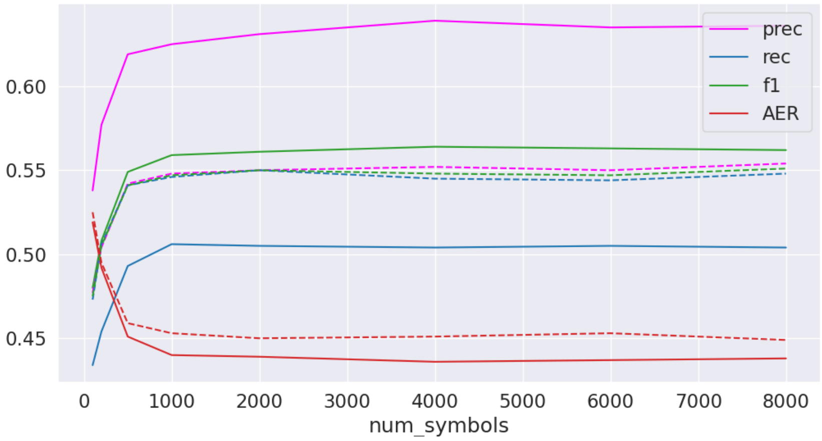


Figure 6.8: Scores for BPE-dropout on English-Romanian with dropout rate 0.1, threshold at 0.7

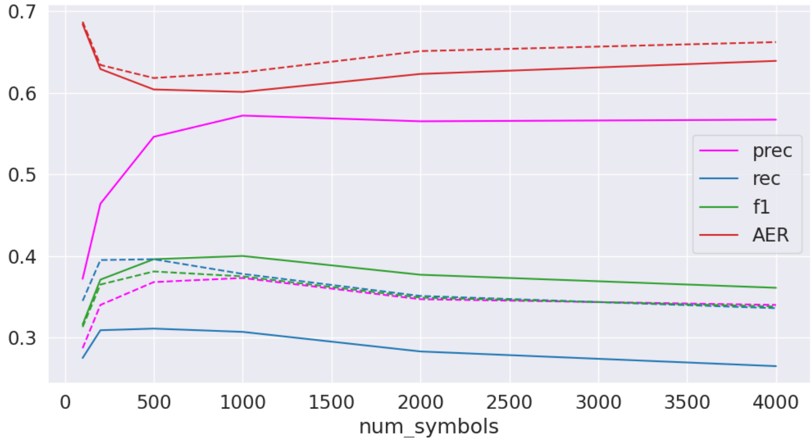


Figure 6.9: Scores for BPE-dropout on English-Hindi with dropout rate 0.1, threshold at 0.7

beginning, the algorithm starts by creating around 1.5 BPE merges per second, which is relatively slow because since the most frequent pairs are so frequent, most of the indexes in the corpus are visited and many update operations are done. The further down in the iterations, the less frequent merges are, the faster it is to merge them and update the corpus. The speed of merges per second gets faster and faster until by the time it is finished with the corpus, it is running at roughly 300 merges per second for English and 120 merges per second in German.

After the improvement, the algorithm, on average, needs **55s to learn BPE merges in English, and 1:25min in German on corpora composed of 10.000 sentences**. The time difference between English and German is due to the fact that the words in German are generally lower, with a higher variance, and there are therefore more merge possibilities. In contrast, Sennrich et al. [4]'s algorithm needs 2:15min for English and 3:20min for German. The algorithm presented here shows an improvement in speed of 2.5x for English and 2.4x for German. These tests have been performed on an Intel Core i5-7200U CPU at 2.5GHz, a 64-bit Windows OS with 8GB RAM.

A C++ implementation of the BPE algorithm from Sennrich et al. can be found on Github This package makes use of C++'s advantage to handle memory better than Python, for instance by memory mapping the input file to read it efficiently, and by using multi-threading to pre-compute the BPE splits of all words in the input file. When applied to this thesis' dataset, it barely needs a few seconds to learn BPE codes, making it much faster than the algorithm developed in this thesis. However, this package does not include BPE-dropout or any other word separation rather than *</w>* at the end of the word; it is not possible to create BPE tokens without word boundaries using this method. It is assumed that it would not be very complicated to integrate these two aspects into the package, but given that this algorithm was created by a PhD student, there are no guarantees of maintenance in the future. In contrast, this thesis' algorithm takes more time but depending on *dropout* and *space* parameters, the algorithm is adapted to both scenarios.

6.5 No-space results

Learning BPE units that can handle merges among words takes more time, since there are more possibilities to merge; each end of the word and each beginning of the word can now be merged. In the previous case, there was only a fixed number of merges that could be done inside a word, once the whole word was merged into one unit, no other merges were possible. Now however, that word can be merged with its previous or subsequent tokens.

As with the space case, the algorithm starts relatively slow, at around 0.7 BPE merges per second. While the most frequent merge in the space case is *_t, h* for English and *e, n* for German, in the no-space case the most frequent merges are *e, _* for English and *e, n* again for German. In the English case, the pair *e, _* is much more frequent than the pair *_t, h*, and it takes more time for this pair to get merged. Afterwards, the number of merges per second increases until plateauing at 50 merges per second in English, and slightly lower in German. The runtime is **3:50min for 10.000 merges in English and 4:10min in German**. Since there are more merge possibilities, the algorithm was also run **for 20.000 merges, with a runtime of 9:30min in English and 10min in German**.

Further down the pipeline, alignments are performed without any notion of what words are, and *Fastalign* and *Eflomal* are not thought to handle multiword-to-multiword alignments. It is therefore expected that the scores will be lower than in the space case, and it is also expected that the more BPE merges, the bigger the segmentations, the more words will be merged together. For instance, if the multiword unit *in_the_street* is aligned to *auf_der_Strasse*, the following mapping would be performed: *in-auf*, *in-der*, *in-Strasse*, *the-auf*, *the-der* and so on. This brings the precision down dramatically, since most of these alignments are incorrect. For the case of no dropout, the best result for BPE without space separations is a F1 score of 0.477, obtained at 200 merges, which is a very low number of merges compared to the previous best scores. The F1 score is also poor relative to the case where merges between words are no allowed. Regardless, given that there is no notion as to what a word constitutes, this result cannot be poorly regarded.

As the following figures show, BPE-dropout outperforms BPE also in no-space mode, making this improvement consistent regardless if word boundaries are considered or not. It can be therefore concluded, that **BPE-dropout outperforms BPE in all experiments done in this thesis, with word boundaries or without**. For a dropout rate of 10%, results already improve wtih an F1 score of 0.523 for 200 merges and an alignment threshold of 0.7. The maximum F1 score is obtained at a dropout rate of 20%, with an **F1 score of 0.559 for 500 merges and an alignment threshold of 0.5**, which constitutes an improvement of 10.5% with respect to no-space BPE with no dropout.

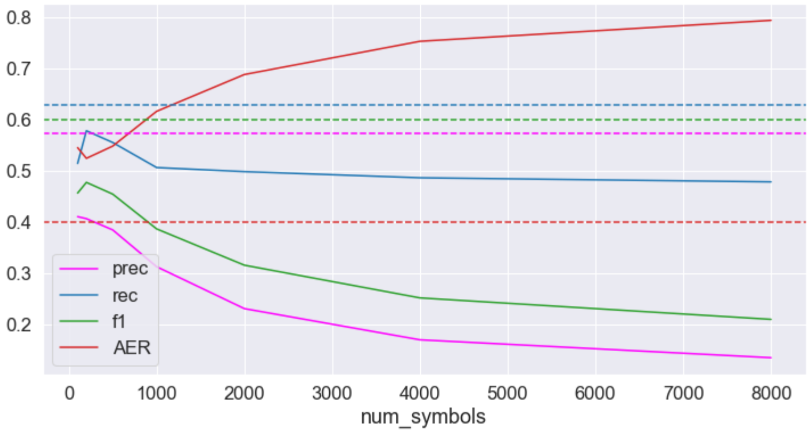


Figure 6.10: Scores for no-space BPE

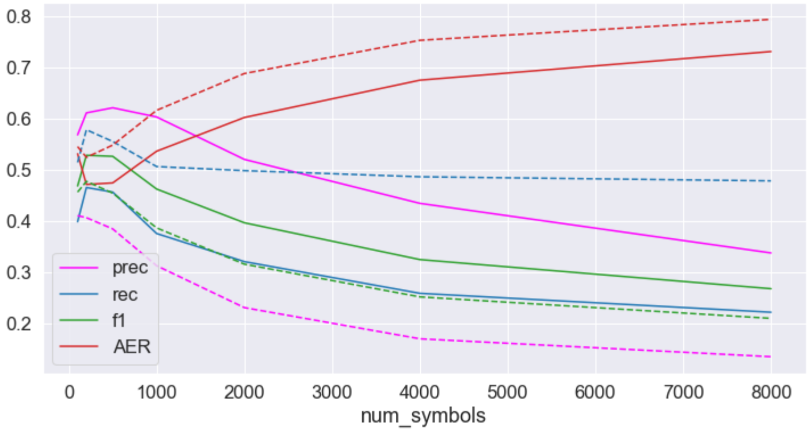


Figure 6.11: Scores for no-space BPE-dropout with dropout rate 0.1, threshold at 0.7

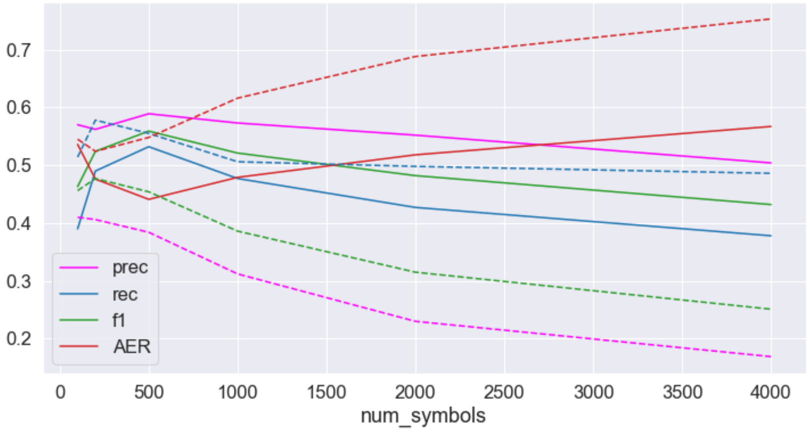


Figure 6.12: Scores for no-space BPE-dropout with dropout rate 0.2, threshold at 0.5

Experiments with higher dropout rates have also been conducted, namely a dropout rate of 50% which is very large, half the merges are dropped, with a F1 score of 0.529 for 4000 merges and an alignment threshold of 0.3. These parameters are considerably different than the ones previously. The ideal score happens at 4000 merges, which makes sense that it is a bigger number since so many merges are dropped. And the alignment threshold is also very low compared to previous cases, meaning that most alignments are accepted creating a sort of union.

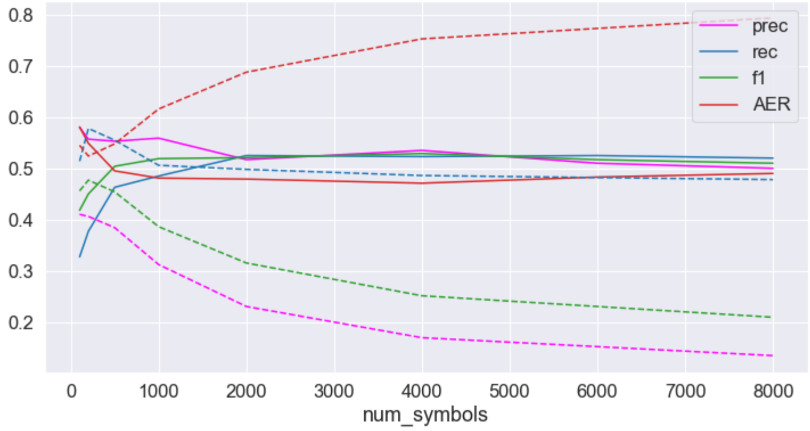


Figure 6.13: Scores for no-space BPE-dropout with dropout rate 0.5, threshold at 0.3

6.6 No-space results for other language pairs

Similarly to the case of BPE-dropout in space mode, where it was shown that BPE-dropout outperforms BPE with a dropout rate of 10% and an alignment threshold of 0.7, it is also interesting to see the performance of BPE-dropout in no-space mode in other language pairs, in this case with the optimal hyperparameters of no-space mode, namely a dropout rate of 20% and an alignment threshold of 0.5. The following figures show the performance of BPE-dropout given the baseline of BPE in no-space mode, for the language pairs English-Romanian and English-Hindi. At the ideal hyperparameter of 500 merges, the F1 score improvement of BPE-dropout over BPE in English-Hindi is of 6.2%, and 8.4% in English-Romanian.

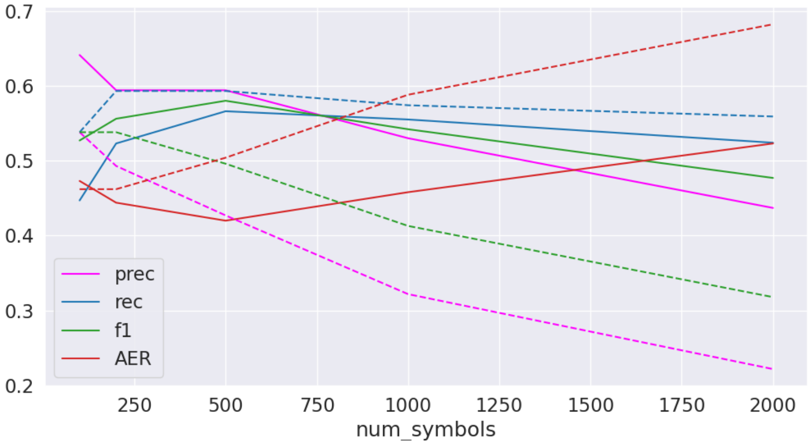


Figure 6.14: Scores for no-space BPE-dropout on English-Romanian with dropout rate 0.2, threshold at 0.5

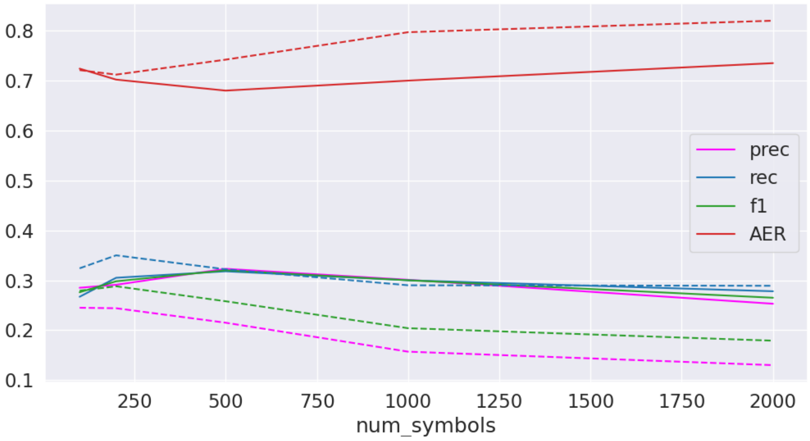


Figure 6.15: Scores for no-space BPE-dropout on English-Hindi with dropout rate 0.2, threshold at 0.5

6.7 Summary

The F1 score results of this thesis are summarized as follows:

Eng-Deu word alignment score: 0.6
Eng-Deu BPE score: 0.609
Eng-Deu BPE score, BPE only in Eng: 0.56
Eng-Deu BPE-dropout score: 0.635

The best hyperparameters for Eng-Deu BPE-dropout are: dropout rate 10%, 2000 merges, alignment threshold 0.7, 10 segmentations. Best scores for this setup are:

Eng-Deu BPE-dropout: 0.685
Eng-Deu BPE-dropout score, BPE only in Eng: 0.59
Eng-Ron BPE-dropout improvement over BPE: 1.5%
Eng-Hin BPE-dropout improvement over BPE: 2.8%

The improvements achieved in this thesis are outlined as follows:

Improvement of learn-BPE algorithm: speedup of 2.5x
Eng-Deu no-space BPE-dropout improvement over no-space BPE: 10.5%
Eng-Ron no-space BPE-dropout improvement over no-space BPE: 8.4%
Eng-Hin no-space BPE-dropout improvement over no-space BPE: 6.2%

For clarity and interpretability, all the codes and materials used for the development of this thesis are published in .

7 Future work

As future work, other language pairs could be explored, especially those not involving English, or other languages with different tokenization. It would be specifically interesting to try out the no space mode with languages without any space tokenization. The capabilities of BPE-dropout in no space mode would be tested this way.

Additionally, in this thesis only the sampling method of Dropout has been explored. This is the way to obtain different segmentations from the same raw corpus, which in this case is done very simply and randomly with a dropout rate. It is hypothesized that if a random sampling already improves the results of BPE, a slightly more intelligent one might improve it even further. Besides, different scoring methods could be explored other than the maximum pair frequency as in BPE. Regarding merging, instead of merging sequences based on their independent frequency, there might be an attempt to make this merging based on the sequence length. This is an example of how BPE would merge a long word:

i n d e p e n d e n t l y
in d e p e n d e n t l y
in d e p e n d e n t ly
...
indep end ent ly
independ ent ly
independent ly
independently

Instead of creating many short sequences, such as *indep end ent ly*, a different method would be to make a unit as big as possible and then merge short sequences to it. This way, the sequence *independent* would be created before than in the previous case.

i n d e p e n d e n t l y
in d e p e n d e n t l y
in de p e n d e n t l y
in dep e n d e n t l y
indep e n d e n t l y
indepe n d e n t l y
...
independ en t ly
independen t ly
independent ly
independently

8 Conclusion

This thesis has broadly analysed all tokenization algorithms, while explaining BPE in depth providing examples, code snippets to make the algorithm understandable, and adding use cases, resources and references. The replication process of the performance of BPE has also been minutely detailed. Regarding evaluation methods, the background of which has been explained in the literature review section, primarily Fastalign has been used, clarifying in detail how it works.

Additionally, the improvement over BPE, BPE-dropout has also been meticulously described, its algorithm outlined, the improvement over BPE illustrated, its results replicated, and its own drawbacks exposed as well. It has been shown that by iterating over some of the hyperparameters that BPE-dropout uses, its results have been slightly improved by 2%.

By going over the original BPE algorithm, this thesis has optimized the algorithm to learn BPE units by making it faster, namely 2.5 times faster. The algorithm has also been adapted to handle the case of space and no space tokenization, which was not present in the original algorithm.

To the knowledge of the author, this is the first piece of research dealing with no space tokenization, and no space BPE. This function is integrated into the pipeline, so that by tweaking some parameters in the global variable file in the repository, all the consequent pipeline is adapted to the specific case.

Throughout the thesis it is proven that BPE-dropout outperforms BPE even in no-space case, confirming the general improvement of BPE-dropout. This is also confirmed by seeing that BPE-dropout outperforms BPE in various languages, other than English-German.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[3] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp, 2019.

[4] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2015.

[5] Ivan Provilkov, Dmitrii Emelianenko, and Elena Voita. Bpe-dropout: Simple and effective subword regularization, 2019.

[6] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge university press, 2008.

[7] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context, 2019.

[9] Xiang Zhang and Yann LeCun. Text understanding from scratch, 2015.

[10] Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. Learning to generate reviews and discovering sentiment, 2017.

[11] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time, 2016.

[12] Jason Lee, Kyunghyun Cho, and Thomas Hofmann. Fully character-level neural machine translation without explicit segmentation. *Transactions of the Association for Computational Linguistics*, 5:365–378, 2017.

[13] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing, 2019.

[14] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75. Association for Computational Linguistics, July 2018.

[15] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152. IEEE, 2012.

[16] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining, 2019.

[17] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

[18] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations, 2019.

[19] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding, 2019.

[20] Peter F Brown, Stephen A Della Pietra, Vincent J Della Pietra, and Robert L Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311, 1993.

[21] Franz Josef Och, Christoph Tillmann, and Hermann Ney. Improved alignment models for statistical machine translation. In *1999 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, 1999.

[22] Philipp Koehn. *Statistical machine translation*. Cambridge University Press, 2009.

[23] Dániel Varga, Péter Halácsy, András Kornai, Viktor Nagy, László Németh, and Viktor Trón. Parallel corpora for medium density languages. *Amsterdam Studies In The Theory And History Of Linguistic Science Series 4*, 292:247, 2007.

[24] Chris Dyer, Victor Chahuneau, and Noah A. Smith. A simple, fast, and effective reparameterization of IBM model 2. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 644–648. Association for Computational Linguistics, jun 2013.

[25] Robert Östling and Jörg Tiedemann. Efficient word alignment with markov chain monte carlo. *The Prague Bulletin of Mathematical Linguistics*, 106(1):125–146, 2016.

[26] Rada Mihalcea and Ted Pedersen. An evaluation exercise for word alignment. In *Proceedings of the HLT-NAACL 2003 Workshop on Building and using parallel texts: data driven machine translation and beyond*, pages 1–10, 2003.