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

Master thesis

This is the unknown title of my master thesis

Version 0, 9 June 2020

submitted: June 25, 2020

by: Ane Berasategi

supervisor: Masoud Jalil Sabet

supervisor: Hinrich Schütze

Abstract

An *abstract* is a brief summary of a research article, thesis, review, conference proceeding or any in-depth analysis of a particular subject or discipline, and is often used to help the reader quickly ascertain the paper's purpose. When used, an abstract always appears at the beginning of a manuscript, acting as the point-of-entry for any given academic paper or patent application. Abstracting and indexing services for various academic disciplines are aimed at compiling a body of literature for that particular subject [**?**].

Task of the Thesis in the Original:

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 Otto von Guericke University of Magdeburg (OvGU) 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.

Magdeburg, June 25, 2020

Contents

List of Acronyms

ISO International Organization for Standardization

PDF Portable Document Format

List of Figures

List of Tables

1 Introduction

The introduction should present the topic of the thesis to specify the purpose and importance of the work. Other possible contents of an introduction are described in section 1 on page 5.

2 Goals of the thesis

3 Tokenization

3.1 Introduction

Tokenization is the first major step in language processing. Once the corpus is obtained, before starting to process the text, because we're dealing with language, we want to understand the meaning of the text. **Tokens** in the language have a semantic meaning, be it words, phrases, symbols or other elements, whereby a meaning is assigned to each token. Tokenization is the process of breaking a stream of text up into tokens. [1]

The main idea behind tokenization is simplifying or compressing the input text into meaningful units, tokens, creating a big vocabulary of tokens and a shorter sequence.

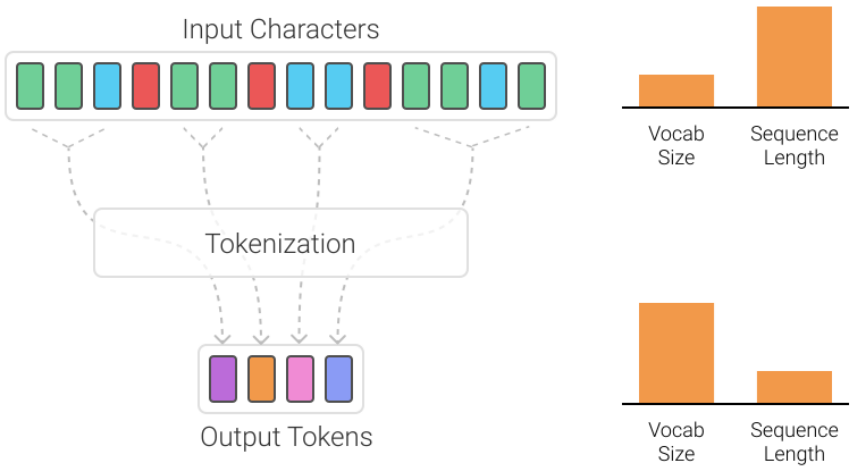


Figure 3.1: Tokenization of a sequence of text

The way to tokenize heavily depends on the task afterwards. Some applications might require different tokenization algorithms. Nowadays, most deep learning architectures in NLP process the 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 sections. 3.2.1 In short, the type of tokenization depends on the type of embedding. There are several, explained further in the following sections, and each has its advantages and drawbacks. 3.2

What is a token?

A token is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing. Here is an example of tokenization.

Input: I ate a burger today, and it was good.
Output: ['I', 'ate', 'a', 'burger', 'today', 'and', 'it', 'was', 'good']

In this example, the way to obtain tokens looks simple: just locate word boundaries, split by whitespace and get the symbols, and remove the punctuation marks, since punctuation marks have no definite meaning. However, this is not always the case.

How to deal with punctuation marks?

Each language has its own set of tricky cases, for example in English what is the correct approach when dealing with apostrophes for possession and contractions? For example words like *aren't*, *Sarah's* and *O'Neill*. Because of this, it is imperative to know the language of the text to be tokenized. *Language identification* is the task of identifying the language of the input text. From the initial k-gram algorithms used in cryptography (Konnheim, 1981), to more modern n-gram methods (Dunning, 1994) have been used. Once the language is known, we can follow the rules for each case and deal with punctuation marks appropriately.

Other types of tokens

Additionaly, tokens can be either characters or **subwords**. For example, the word *smarter*:

Sentence: the smarter computer Word: the, smarter, computer
Word without word boundaries: the smart, er comput, er
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

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.

3.2 Tokenization algorithm types

3.2.1 Word level tokenization

Word level tokenization was the first tokenization type used, and is also the most common. It splits a piece of text into individual words based on word bounderies, usually a specific delimiter, mostly 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. Word level tokenization without word boundaries aims to address that problem. 3.2.4 on page 15

Word level algorithms

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

The natural language toolkit (NLTK) in python provides a tokenize package which includes a *word_tokenize* function, which requires the user to give in the language of the text. 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']
```

Similarly, SpaCy offers a similar functionality. It is possible to load the language model for a different language and model size.

```
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']
```

Keras also offers a similar functionality:

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

As does Gensim:

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

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 [2]) is a way of transforming a word into a fixed-size vector representation, as shown in the picture below.

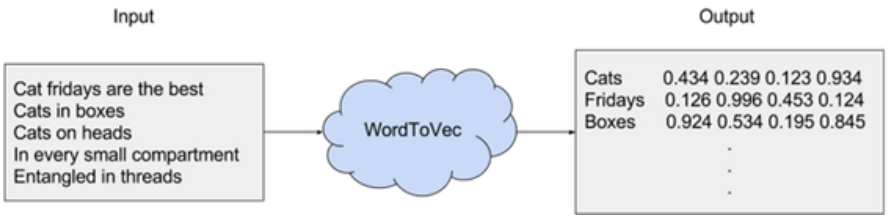


Figure 3.2: Representation of word embeddings

If viewed abstractly in its N dimensions, each word's representation is close to similar words. As a simple example:

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

The embedding numbers are just as an example, to illustrate the distances between words. For example, *smart* and *intelligent* have a distance of 2, since the last 2 numbers in the vector differ by one respectively. If this was plotted in a 4 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 higher. In the plot, these words would be very far apart.

As well as word2vec, there are other word embedding algorithms such as GloVe or fasttext.

Thus, with word embeddings, a sentence is transformed into a sequence of embedding vectors, which is very useful for NLP tasks.

Word level drawbacks

Word embeddings have some drawbacks however. In many cases, a word can have more than one meaning: *well*, for example, can be used in these 2 scenarios.

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

In the first case, well is an adverb while in the second it's a noun. *well*'s embedding will probably be a mixture of the two, since word embeddings don't generalize to homonyms.

Another drawback is that word embeddings aren't well equipped to deal with out of vocabulary (oov) words. Word embeddings are created with a certain vocabulary size, that is, a certain number of words are known. If afterwards a new word arrives which isn't present in the vocabulary, because it's a foreign word, or a misspelled word, it will be given an unknown <UNK> embedding, that will be the same for all unknown words. Therefore all unknown words have the same embedding, that is, the NLP task will treat all these words as if they had the same meaning. The information within these words is lost due to the mapping from OOV to UNK.

Another issue with word tokens is related to the 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. 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 ends, they're mostly used to create embeddings. And if embeddings from word tokens are problematic, the tokenization method is changed in order to create different tokens, in order to create other types of embeddings.

3.2.2 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're broken down into characters and these characters are usually known. 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) [3], who introduced the character CNN, consider all the alphanumeric character, in addition to puctuation 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, as stated below.

Radfor et al. (2017) [4] from OpenAI showed that character level models can capture the semantic properties of text. Kalchbrenner et al.(2016) [5] from Deepmind and Leet et al. (2017) [6] 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. Spelling mistakes, rare words and morphological aspects are very well handled with this type of tokenization.

Character level algorithms

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

Character level drawbacks

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

In addition, it becomes challenging to learn the relationship between the characters to form meaningful words. Besides, there is no semantic information among characters, characters are semantically void.

Sometimes the NLP task doesn't 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.

3.2.3 Subword level tokenization

Subword tokenization is the task of splitting the text into subwords or n-gram characters. For example, words like lower can be segmented as low-er, smartest as smart-est, and so on. In the event of an OOV word such as *eorner*, this tokenizer will divide it into *eorn, er* and effectivey obtain some semantic information. Very common subwords such as *ing*, *ion*, usually with a morphological sense, are learnt through repetition. The word *unfriendly* would be split into *un, friend, ly*.

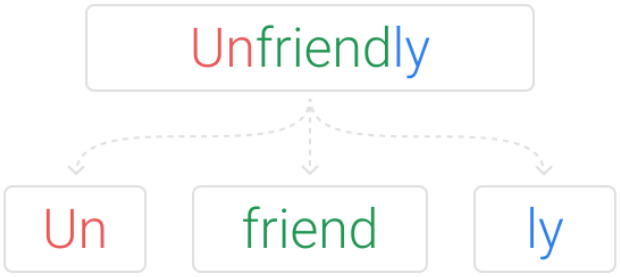


Figure 3.3: Representation of the word 'unfriendly' in subword units

Nowadays, as of 2020, the most powerful deep learning architectures are based on Transformers (Vaswani et al., 2017 [7]), and these rely on subword tokenization algorithms to prepare the vocabulary.

Subword level algorithms

Since Transformers are a relatively architecture at the time of writing, subword tokenization is an active area of research. Nowadays three 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, a popular NLP framework, released Transformers and Tokenizers (Wolf et al., 2019 [8]). They 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 [9]) to build a subword dictionary. This is roughly how the algorithm works:

1. Get a large enough corpus.
2. Define a desired subword vocabulary size.
3. Initialize the word unit inventory with all the characters in the text.
4. Split word to sequence of characters and appending an beginning-of-word or end-of-word affix/suffix respectively.
5. Calculate pairs of sequences and their frequencies
6. Generate a new subword according to the pairs of sequences that occurs most frequently.
7. Repeat step 4 until reaching subword vocabulary size (defined in step 2) or the next highest frequency pair is 1.

Unigram LM

Unigram language modeling (Kubo, 2018 [10]) is based on the assumption that all subword occurrences are independent and subword sequences are produced by the product of subword occurrence probabilities. Both WordPiece and Unigram Language Model leverages languages model to build subword vocabulary. 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 (e.g. X can be 80). To avoid out-of-vocabulary, character level is recommend to be included as subset of subword.
6. Repeating step 3–5until reaching subword vocabulary size which is defined in step 2 or no change in step 5.

The author argues that BPE is based on a greedy and deterministic symbol replacement, which can't provide multiple segmentations with probabilities. The unigram LM model is more flexible than BPE because it's based on a probabilistic LM and can output multiple segmentations with their probabilities.

WordPiece

WordPiece (Schuster and Nakajima, 2012 [11]) was initially used to solve Japanese and Korean voice problem. It's 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 word unit inventory with all the characters in the text.
5. Build a languages model based on the vocabulary.
6. Generate a new word unit by combining two units out of the current word inventory to increment the word unit inventory by one. Choose the new word unit out of all the possible ones that increases the likelihood on the training data the most when added to the model.
7. Goto step 5 until reaching subword vocabulary size (defined in step 2) or the likelihood increase falls below a certain threshold.

The BPE algorithm only differs in step 6, since it merges the token combination that has the maximum frequency. This frequency stems from the combination of the tokens, not previous individual tokens. In WordPiece, the frequency of the two tokens separatedly is also taken into account, apart from the frequency of the new token. If there are 2 tokens A & B, the score of this combination will be the following, and the token pair with the highest score will be selected.

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

It might be the case that Frequency('so','on') is very high but their separate frequencies are also high, hence if using WordPiece, 'soon' won't be merged as the overall score is low. Yet if Frequency('Jag','gery') might be low but if their separate frequencies are also low, 'Jag' and 'gery' might be joined to form 'Jaggery'.

BERT (Devlin et al., 2018 [12]) uses WordPiece as its tokenization method, yet the tokenization method has not been done 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

SentencePiece (Kudo et al. 2018 [13]) is a subword tokenization type first introduced by Google in 2017, mainly led by Kubo, the researcher under Unigram LM tokenization 3.2.3. It has an extensive Github repository with freely available code.

As the repository states, it's 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 3.2.3) and unigram LM 3.2.3) with the extension of direct training from raw sentences. It doesn't depend on language-specific pre/postprocessing.

While conceptually similar to BPE, it doesn't go for the greedy encoding strategy, allowing it to achieve higher quality tokenization and reduce error induced by location-dependence 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.

This results in very high quality tokenization, but comes at great cost to performance, at times making it the slowest part of an NLP pipeline. 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. 3.2.3

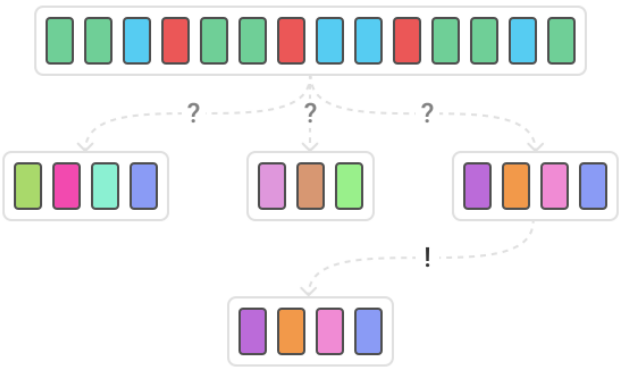


Figure 3.4: Representation of the sentencepiece tokenization in a sequence of text

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 [9]) which uses the number of merge operations. The number of merge operations is a BPE-specific parameter and not applicable to other segmentation algorithms, including unigram, word and character.

3.2.4 Tokenization without word boundaries

Another type of tokenization, beyond word, character or subword, is tokenization without subword boundaries. The three types of tokenization explored until now cannot create

units among words, that is, they consider words separatedly.

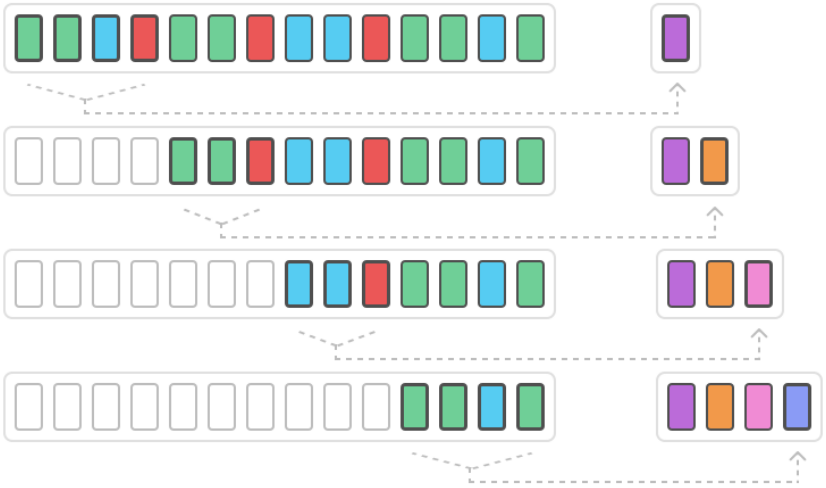
Word and character level tokenization are deterministic, meaning that for a given word, its tokenization will always be the same. But in subword tokenization, a word like *imagination* might have different tokenization, for example: *im, agina, tion*, *imagina, tion*, etc. It depends on the corpus that the subword tokenization algorithm is trained on, but there is no foolproff and/or unique tokenization.

Besides, when dealing with languages that don't include space tokenization, such as several Asian languages, an individual symbol can resemble a syllable rather than a word or letter. Additionally, most words are short (the most common length is 2 characters), and given the lack of standardization of word breaking in the writing system or lack of punctuation in certain languages, it is not always clear where word boundaries should be placed.

An approach to handle this has been to abandon word-based indexing, and do all indexing from just short subsequences of characters (character -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.

3.3 BPE

Byte Pair Encoding (BPE) (Sennrich et al., 2015 [9]), is a widely used tokenization method among Transformer-based models. It's a word segmentation algorithm that merges the most frequently occurring character or character sequences iteratively. The code is open source and there is an active repository on Github. Here is an example of a sequence in its original form and after the BPE merges, much shorter than the original.



BPE enables the encoding of rare words with appropriate subword tokenization without introducing any 'unknown' tokens.

BPE addresses the issues of word and character tokenizers in the following ways: regarding OOV words, BPE segments OOV as subwords and represents the word in terms of these subwords. Regarding the longer sequence length which was one of the drawbacks of character tokenization, in this case the length of input and output sentences after BPE are shorter compared to character tokenization.

3.3.1 Minimal algorithm to learn BPE segmentations

These are the steps to learn BPE segmentations:

1. Split the words in the corpus into characters after appending </w>, or another special symbol to show the beginning or end of the word.
2. Compute the frequency of each pair of characters or character sequences in corpus and obtain the most frequent one.
3. Save the best pair to the vocabulary.
4. Merge the most frequent pair in corpus.
5. Repeat steps 3 to 5 for a certain number of iterations.

As an example, let's consider a simple corpus with a single line, and the character '_' to mark the beginning of each word, so as not to merge anything between different words later. The following code shows the first step of the algorithm.

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

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

After that, we can initialize the vocabulary, which for now consists solely of the unique characters in the corpus. The following code snippet shows the second and third step.

```
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
```

```
merge_list = []
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, ('t', '?'): 1})

most_frequent = pairs.most_common(1)[0][0]
merge_list.append(most_frequent)
>>> most_frequent = ('_t', 'h')
```

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's the time to merge this pair in the corpus.

```
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)
>>> tokens = ['_th i s _i s _th i s .']
```

Now the only thing left to do is to iterate this a number of times, which is usually decided by the user, until a certain number of pairs have been found. At each step, the object *pairs* is calculated again since there might be new pairs such as ('_th', 'i') in this example. The whole minimal code would look like this:

```
corpus = ['this is this.']
merge_list = []
num_merges = 10

tokens = read_corpus(corpus)

for _ in range(num_merges):

    pairs = get_stats(tokens)
    most_frequent = pairs.most_common(1)[0][0]
    merge_list.append(most_frequent)
    tokens = merge_pair_in_corpus(tokens, most_frequent)
```

In each step of the iteration, the *get_stats* function iterates all the characters in the corpus, so the complexity is O(len(corpus) * len(sent)), for an average sentence length. Obtaining the most frequent pair takes constant time, since the object *pairs* is a Counter object and has a built-in 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)). The num_merges part cannot be avoided, but operating through all the characters of the corpus is very computationally expensive. In this thesis this algorithm has been optimized, as will be shown below.

3.3.2 Applying BPE to OOV words

In the event of an OOV word, such as *these*, which the corpus used in the previous example doesn't know, the BPE algorithm can create some subword units from the corpus used before. This is Sennrich et al.'s approach [9]:

1. Split the OOV word into characters after inserting '_' in the beginning.
2. Compute the pair of character or character sequences in a word.
3. Select the pairs present in the learned operations.
4. Merge the most frequent pair.
5. Repeat steps 2 and 3 until merging is possible.

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

```
oov = 'these'
oov = ['_' + ' '.join(list(oov))]
i = 0
while True:
    pairs = get_stats(oov)
    # find the pairs available in the learned operations
    idx = [merge_list.index(i) for i in pairs if i in merge_list]

    if len(idx) == 0:
        print("BPE completed")
        break
    # choose the most frequent learned operation
    best = merges[min(idx)]

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

After some iteration, the OOV word *these* is transformed into *th e s e*, since *th* is the only known merge in the merge_list.

Different subword tokenization strategies change this in subtle ways: BPE scores tokens simply regarding their count or frequency in the corpus, and uses eager encoding, while character tokenization simple has a max n-gram size of 1. This strategy is very similar to phrase extraction using collocation, which counts word n-grams, scores them, and exports a model in similar fashion, except the sequence items are words instead of characters. As it turns out, using a phrase extraction library it's possible to learn an effective BPE vocabulary for languages like Japanese which aren't whitespace separated by simply changing the word-regex to single characters, helping give this connection some credibility.

A finding like this begs the question - should BPE have a more intelligent scoring phase when choosing its vocab from n-gram counts? Mutual information as a scoring function was very valuable in achieving better phrase extraction, and it seems that it would be useful here as well. And even more than that, is there an abstract tokenizer we can define that fulfills each of these strategies and more via hyperparameters like n-gram length, scoring strategy, etc? Could this abstract tokenizer have a common implementation across platforms like Tensorflow, PyTorch, the web, etc?

3.3.3 BPE dropout

BPE dropout (Provilkov et al., 2019 [14]) slightly change 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, the segmentation procedure is altered. During segmentation, at each merge step some merges are randomly dropped with probability p, hence the name of BPE dropout. In the paper they use p=0.1 during training and p=0 during inference. For Chinese and Japanese, they use p=0.6 to match the increase in length of segmented sentences for other languages.

The paper hypothesize that exposing a model to different segmentations might result in better understanding of the whole words as well as their subword units.

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

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

Sentences segmented with BPE-Dropout are longer. There's a danger that models trained with BPE-Dropout might use more fine-grained segmentation in inference and

hence slow inference down.

3.3.4 BPE drawbacks

Kudo (2018) [10] 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**, which may prevent a model from better learning the compositionality of words and being robust to segmentation errors. Subwords into which rare words are segmented end up poorly understood.

This problem was solved by BPE Dropout [14], which corrupts the segmentation process and produces several segmentations.

4 Guidelines for Thesis Composition

5 Summary

The summary is the last section of the text and summarizes the results of the work (see also section **??** from page **??**).

Bibliography

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

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

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

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

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

[6] 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.

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

[8] 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.

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

[10] 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, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[11] 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.

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

[13] 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.

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

A Diagrams

Possible contents for an attachment as well as its formal design are described in more detail in section **??** on page **??**.

B Software for Using the LaTeX Document Markup Language

B.1 Microsoft Windows

Statements for the Bachelor's/Master's Thesis

Guidance for the Preparation of Degree Theses

submitted: Month Year
by: John Doe

1. This guidance is a great help for all students who write a thesis for the first time.
2. The brief text pointing to main statements makes it easy to read and understand this guidance within a short time.
3. The consideration of these guidelines expedites the review and appraisal of the thesis by the supervisor and the examiners.
4. ...

Signature