What If English Had No Spaces? Investigating the Challenges of NLP for Character-Based Languages

---

A Thesis
Presented to
the Established Interdisciplinary Committee for Mathematics and Computer Science
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Yilin Li

May 2021

Approved for the Committee
(Mathematics and Computer Science)

Mark Hopkins	Jamie Pommersheim

Table of Contents

Abstract

Tokenization is the one of the most critical tasks in natural language processing. For languages like English, whitespace separates each word so that a sentence has a natural segmentation. Character-based languages like Chinese and Japanese, however, do not use spaces to mark word boundaries. In this work, we aim to explore the challenges posed by character-based languages in NLP.

We used whitespace-deprived English to mimic character-based languages and we used 2 tokenization methods to train our GPT2 models: BBPE and Unigram LM. For each one, we trained two models, one on whitespace English text, the other on the same text with whitespaces removed. The whitespace model has 76.27% lower perplexity for the BBPE method and 11.90% lower perplexity for the Unigram LM method than their corresponding non-whitespace models. Overall, Unigram LM tokenization performs much better on non-whitespace English text which suggests that Unigram LM should be used when we deal with character-based languages.

Chapter 1

Motivation

1.1 Introduction to NLP

Language is one of the most critical characteristics which distinguish human beings from animals. Humans use language to communicate thoughts and intentions. The field of natural language processing, NLP for short, uses computers to handle tasks which utilize natural language as the information carrier, such as text understanding, text classification, text summarization, question answering, information extraction, etc. Computers are used to analyze, understand, and derive meanings from human language. NLP is considered a branch of artificial intelligence.

To get a sense of the challenges faced in NLP, take the following sentence as an example

Tom catches the cat with a hat.

In different contexts, we can interpret the sentence differently. One is saying that Tom catches the cat, using a hat as a tool. The other is saying that Tom catches the cat which wears a hat. One difficulty of NLP is clearly illustrated: it's hard for a computer to determine the latent boundaries of phrases. If it can't, then the whole sentence will likely be misinterpreted.

1.2 Tokenization

Tokenization is the task of segmenting text into pieces, called *tokens*, and it is almost always the first task we need to do when we're given text data. A token is a contiguous sequence of characters that are grouped together as a useful atomic unit for processing a sentence or passage.

Example 1.2.1. A simple example of tokenization.

He likes cats.
{'He', 'likes', 'cats.'}

which organizes the sentence into three pieces. This segmentation can then be used by NLP tools that operate at the token level.

How should we set the boundaries of tokens? We want to know whether a tokenization method fits whatever goal we have. Intuitively at least, {'He', 'likes', 'cats.'} seems like a better tokenization than {'He li', 'kes cats.'} if we want to understand the meaning of the sentence because it splits the sentence into tokens with independent significance.

1.2.1 How Machines Read

Human beings can understand language because we start to learn it when we are born. We learned the sound of the language from parents and we learned how to write when we went to school. Gradually, we learned the meanings of the language because we memorized and utilized them for years. However, machines don't have the same experience. How do they read?

It turns out that machines process a text as an integer sequence. For example, if we consider the tokens from Example 1.2.1 {'He', 'likes', 'cats'}, a machine might treat it as {24, 102, 72}. An important point in machine reading is that even though 'like' and 'likes' are considered strongly related by human beings, machines read them as two entirely different integers, which conceals the lexical similarities. More effective tokenization methods can help address this issue.

1.2.2 Whitespace Tokenization

A common tokenization method is *whitespace tokenization*. It is a simple algorithm that splits sentences into words like Example 1.2.1. However, there are downsides with this intuitive idea.

First, it doesn't deal well with punctuation.

Example 1.2.2.

Don't you love her? No.
{"Don't", "you", "love", "her?", "No."}

Here, question mark and period are attached to the words *her* and *No*. This can become a problem because a model will have to learn a different representation for a word and every possible punctuation symbol that could attach to it. This results in a lot of redundant tokens in the vocabulary.

Secondly, whitespace tokenization has the out-of-vocabulary problem. When we use a piece of text data to train a model, the content is limited. Therefore, the model can only recognize a finite number amount of tokens and when we apply the model in reality, encountering some unknown words, the model will have some trouble because it never saw these words during training and thus has no idea how to process it.

Third, whitespace tokenization cannot capture the intrinsic similarities between words like 'run', 'runner', 'running', or 'low', 'lower', 'lowest'. It will interpret them as totally unrelated words, but we as people can clearly identify the stem of the first three words — 'run'.

Moreover, whitespace tokenization is not possible in languages like Chinese and Japanese, which do not use spaces to mark word boundaries. For example, consider the following sentence:

做研究很快乐

"Doing research is very fun"

One tokenization could be:

做研究 很 快乐

{'doing research', 'very', 'fun'}

Another tokenization could be:

做 研究很快 乐

{'make', 'research is fast', 'happy'}

They result in totally different interpretations and it can be a challenge to automatically determine which tokenization is more appropriate.

1.2.3 Character Tokenization

Another intuitive tokenization method is *character tokenization*. The idea is really simple. Take the sentence 'cats run' as the example. It will be tokenized into {'c', 'a','t','s',' ','r','u','n'}, which is a sequence of individual characters of the sentence.

Character tokenization solves two drawbacks of whitespace tokenization. First, it avoids the redundant tokens, or it sets a limit to the size of the vocabulary of the tokens. Given a text corpus, we usually can only extract a fair amount of individual characters. For example, we often only need 26 letters, their Capital forms, plus punctuation symbols to build the vocabulary for an English document. Second, character tokenization avoids the out-of-vocabulary problem. It breaks down each word, known or unknown, into individual characters and represents each word as a sequence of characters.

However, character tokenization introduces new difficulties. Using character tokenization rapidly increases the length of the encoded input and output sentences. This will increase the computational expense. Moreover, individual characters are often semantically void. They are not representing anything meaningful other than a vocal sound.

1.2.4 Subword Tokenization

Subword tokenization can be treated as a hybrid of whitespace tokenization and character tokenization. It breaks down words into sub-units like 'er', 'est', 'ing', etc. An simple tokenization example is as follows:

Example 1.2.3.

unfriendly runner
{'un', 'friend', 'ly ', 'run', 'n', 'er'}
{25, 109, 9, 78, 3, 10}

running friend
{'run', 'n', 'ing ', 'friend'}
{78, 3, 41, 109}

Notice that in this example, a machine reads them as two sequences which share 3 integers: 78, 3, and 109. This is an improvement from whitespace tokenization where *unfriendly runner* will be read as {'unfriendly', 'runner'} and *running friend* will be read as {'running', 'friend'}. Whitespace tokenization will treat 'unfriendly', 'runner', 'running', 'friend' as 4 different integers, concealing the intrinsic relations between 'runner' and 'running', or between 'unfriendly' and 'friend'.

Subword tokenization methods usually learn to group tokens from the data itself. The token vocabulary usually also incorporates individual characters to avoid the out-of-vocabulary problem. The resulting vocabulary often consists of a hybrid of subwords, such as 'ing', and 'er', and full words like 'friend' and 'run'. Subword tokenization allows the model to capture intrinsic relationships between related words such as 'fast', 'faster', and 'fastest'. It also allows models to better understand unknown words by breaking them into subword tokens. In this paper, we will compare three popular subword tokenization methods: Byte Pair Encoding, Byte-level BPE, and Unigram LM. We introduce them in later sections.

In linguistics, morphology is the study of words, how they are formed, and their relationship to other words in the same language (Aronoff 2007). English belongs to morpheme-based morphology, where a morpheme is the smallest meaningful unit in a language. Some morphemes are words such as 'run'. However, some morphemes are prefixes and suffixes such as '-er' in 'runner'. In some sense, subword tokenization is similar to morphological analysis where both aim to find a method to deconstruct a word into smaller meaningful units. However, subword tokenizations in NLP would be more data-driven than the traditional study of morphological analysis.

1.3 Goal

As mentioned before, character-based languages like Chinese and Japanese do not use spaces to mark word boundaries. In this paper, we want to explore the influence of **missing whitespace** on subword tokenization methods, and the language models trained with them. We use whitespace-deprived English as the tool to mimic character-based languages. We use whitespace English and whitespace-deprived English to train different language models and evaluate models on popular metrics. We hope to find clues about how to create improved tokenization methods for non-whitespace languages based on the result of these experiments.

Chapter 2

Introduction and Review: Information Theory

2.1 Entropy

Entropy is a measure of the uncertainty of a random variable in information theory (Cover & Thomas 2006a). Let X be a discrete random variable with alphabet Σ_X and probability mass function p, where $p(x) = Pr\,(X = x)$.

Definition 2.1.1 (Entropy)**.** The *entropy* of a discrete random variable X is defined by

$$H(X) = -\sum_{x \in \Sigma_X} p(x) \log p(x) \tag{2.1}$$

Note that we use the convention $0 \cdot log0 = 0$ and we use base 2 to the log to count it in bits. Observe that the entropy can be treated as the expected value of random variable $\log \frac{1}{p(X)}$. Thus,

$$H(X) = E_p \log \frac{1}{p(X)} \tag{2.2}$$

In one perspective, we can treat entropy as a measure of the uncertainty of a random variable, the higher the entropy, the more uncertain the distribution. However, we can also treat entropy as a representation of the minimum average number of bits "in theory" to encode alphabet Σ_X. I will show them with the following two examples.

Example 2.1.1. Let's toss a coin with probability p_0 of landing heads. The result of the coin is random variable X. Let the alphabet $\Sigma_X = \{0, 1\}$, where 0 represents tails and 1 represents heads. The probability mass function of X is $p(1) = p_0$ and $p(0) = 1 - p_0$. Then we have

$$H(x) = -p_0 \log p_0 - (1 - p_0) log_{1-p_0}, \tag{2.3}$$

which can be viewed as a function of p_0. We show the graph of the function $H(p_0)$ in Figure 2.1. As we can observe, the entropy reaches maximum when $p_0 = 0.5$, which

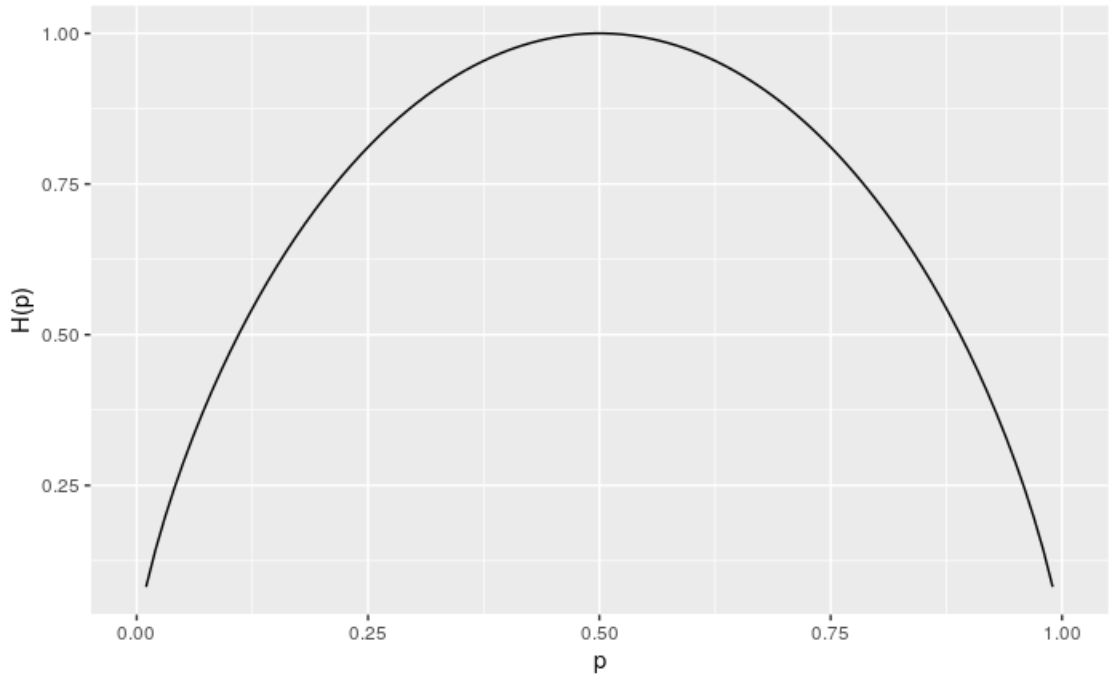

Figure 2.1: $H(p)$ vs. p.

is a fair coin. Note that a fair coin is the most uncertain case when tossing a coin. The entropy equals 0 when $p_0 = 0$ or $p_0 = 1$, which are deterministic cases.

Example 2.1.2. Consider encoding a language with 4 possible letters $\Sigma_X = \{a, b, c, d\}$, where

$$X = \begin{cases} a & \text{with probability } \frac{1}{2}, \\ b & \text{with probability } \frac{1}{4}, \\ c & \text{with probability } \frac{1}{8}, \\ d & \text{with probability } \frac{1}{8}. \end{cases} \tag{2.4}$$

The entropy of X is

$$H(X) = -\frac{1}{2}\log\frac{1}{2} - \frac{1}{4}\log\frac{1}{4} - \frac{1}{8}\log\frac{1}{8} - \frac{1}{8}\log\frac{1}{8} = 1.75 \text{ bits}. \tag{2.5}$$

Suppose a student comes up with 2 different ways to encode this language. Let's use $C(x)$ to represent encoding a letter with binary bits. First, we have

$$\begin{cases} C(a) = 00, \\ C(b) = 01, \\ C(c) = 10, \\ C(d) = 11. \end{cases} \tag{2.6}$$

The expected length of the code word is 2 bits obviously. The other way to encode

this language is

$$\begin{cases} C'(a) = 0, \\ C'(b) = 10, \\ C'(c) = 110, \\ C'(d) = 111. \end{cases} \tag{2.7}$$

The expected length of the code word is $\frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 = 1.75$ bits. Interestingly, it equals the entropy $H(X)$. Readers can easily check that both methods are *uniquely-decodable*, which ensures that the codewords can be recognized uniquely in the received signal so that the decoding is exactly a reverse of the encoding. We can see that the second method is a better encoding method compared to the first one, because it has a better average code word length. It turns out that the entropy is a lower-bound on the average code word length of any uniquely decodable encoding. However, it is not the case that we can always find a way to encode a language with the average code word length equaling the entropy because the length of a code word has to be an integer.

Definition 2.1.2 (relative entropy)**.** The *relative entropy*, or *Kullback-Leibler distance* between two probability mass functions $p(x)$ and $q(x)$ is defined as

$$\begin{aligned} D(p\|q) &= \sum_{x \in X} p(x) \log \frac{p(x)}{q(x)} \\ &= E_p \log(\frac{p(X)}{q(X)}) \end{aligned} \tag{2.8}$$

Note that we have $0 \cdot \log \frac{0}{0} = 0$, $0 \cdot \log \frac{0}{q} = 0$, $p \cdot \log \frac{p}{0} = \infty$, and $D(p\|q) \neq D(q\|p)$ in general. We use relative entropy as a way to assess the quality of language models we will later introduce.

We now introduce one of the most important inequalities in Information Theory, namely *Jensen's Inequality*

Theorem 2.1.1 (Jensen's Inequality)**.** If function f is convex and X is a random variable, then

$$E(f(X)) \geq f(E(X)), \tag{2.9}$$

Moreover, if f is strictly convex. then equality means X is constant.

Proof. We prove the theorem by applying induction on the number of elements in X.

For the base case, let's have two elements in X, then the equation becomes

$$p_1 f(x_1) + p_2 f(x_2) \geq f(p_1 x_1 + p_2 x_2), \tag{2.10}$$

which follows directly from the definition of the convex functions.

For the inductive step, assume the inequality holds for $k-1$ elements in X, then writing $p'_i = \frac{p_i}{1-p_k}$ for $i = 1, 2, \ldots, k-1$, we have

$$\begin{aligned} E(f(X)) &= p_1 f(x_1) + p_2 f(x_2) + \cdots + p_k f(x_k) \\ &= p_k f(x_k) + (1-p_k) \sum_{i=1}^{k-1} p'_i f(x_i) \\ &\geq p_k f(x_k) + (1-p_k) f\left(\sum_{i=1}^{k-1} p'_i x_i \right) \\ &\geq f\left(p_k x_k + (1-p_k) \sum_{i=1}^{k-1} p'_i x_i \right) \\ &= f\left(\sum_{i=1}^{k} p_i x_i \right). \end{aligned} \tag{2.11}$$

□

At this point, we could use *Jensen's Inequality* to prove an important property of relative entropy, a theorem called *Information Inequality*.

Theorem 2.1.2 (Information Inequality)**.** Let $p(x)$, $q(x)$, $x \in X$ be two probability mass function, then

$$D\left(p \| q\right) \geq 0 \tag{2.12}$$

with equality if and only if $p(x) = q(x)\ \forall x \in X$.

Proof. Let A be the domain of $p(x)$, X be the union of the domain of both $p(x)$ and $q(x)$, then

$$\begin{aligned} -D(p\|q) &= -\sum_{x \in A} p(x) \log \frac{p(x)}{q(x)} \\ &= \sum_{x \in A} p(x) \log \frac{q(x)}{p(x)} \\ &\leq \log \sum_{x \in A} p(x) \frac{q(x)}{p(x)} \\ &= \log \sum_{x \in A} q(x) \\ &\leq \log \sum_{x \in X} q(x) \\ &= \log 1 \\ &= 0. \end{aligned} \tag{2.13}$$

Notice that the first inequality follows from *Jensen's Inequality*. Since log functions are strictly concave, we have equality on the first inequality if and only if $\frac{q(x)}{p(x)}$ is constant everywhere. Thus, $\sum_{x \in A} p(x) = c \sum_{x \in A} q(x) = 1$. We have the equality on the second inequation only if $\sum_{x \in A} q(x) = \sum_{x \in X} q(x) = 1$, which means $c = 1$. Therefore, $D(p\|q) = 0$ if and only if $p(x) = q(x)$ everywhere. □

2.2 Entropy in language models

A statistical **language model**, or **LM**, is actually a distribution which assigns probability to a sentence or a sequence of words. Why would we assign probability to sentences? It turns out that these models are vital for many tasks like machine translation, speech recognition, auto-correction, etc. Take auto-correction as an example. The sentence:

the professor says that doing research is fun

should be more probable than

the prefassor say that doing research is fun

since *professor* is misspelled and the word *say* should end with "s".

Consider a language L which has a finite set of symbols. Then the probability of a given sequence of symbols $\omega_1, \ldots, \omega_n$ is

$$\begin{aligned} P(\omega_1, \ldots, \omega_n) &= P(\omega_1)P(\omega_2|\omega_1)\ldots P(\omega_n|\omega_1, \omega_2, \ldots, \omega_{n-1}) \\ &= \prod_{i=1}^{n} P(\omega_i|\omega_{i-1}, \ldots, \omega_1) \end{aligned} \tag{2.14}$$

Notice that a sequence has order, which implies that if we change the order of some symbols, the probability of the sequence may change.

Example 2.2.1. Consider the sentence: He has a dog. Then,

$$P(\text{He has a dog}) = P(\text{He})P(\text{has}|\text{He})P(\text{a}|\text{He, has})P(\text{dog}|\text{He, has, a}) \tag{2.15}$$

We can calculate the entropy of a probability distribution to determine how 'random' it is. How do we calculate the entropy of a sequence of words? One option would be having a variable that ranges over all finite sequences of length n.

$$H(\omega_1, \ldots, \omega_n) = -\sum_{W^n \in L} p(W^n) \log p(W^n), \tag{2.16}$$

here W^n represents a word sequence of length n.

We could then define the **entropy rate** of a sentence as the entropy of the word sequence divided by the number of words.

$$\frac{1}{n}H(\omega_1, \ldots, \omega_n) = -\frac{1}{n}\sum_{W^n \in L} p(W^n) \log p(W^n) \tag{2.17}$$

But to measure the entropy of a language, we need to consider sequences of arbitrary length. Therefore, the language L's entropy rate will be

$$\begin{aligned} H(L) &= \lim_{n\to\infty} \frac{1}{n} H(\omega_1, \ldots, \omega_n) \\ &= -\lim_{n\to\infty} \frac{1}{n} \sum_{W^n \in L} p(W^n) \log p(W^n) \end{aligned} \tag{2.18}$$

The Shannon-McMillan-Breiman theorem (Cover & Thomas 2006b, Algoet & Cover 1988) states that if the language is both stationary and ergodic,

$$H(L) = \lim_{n\to\infty} -\frac{1}{n}\log p(\omega_1, \ldots, \omega_n) \tag{2.19}$$

This illustrates the fact that it's not necessary to sum over all the possible finite sequences as long as we have a long enough sequence. This actually makes sense. The intuition behind the theorem is that if we have a long enough sequence, then it would contain all the other short sequences with respect to their own probabilities. A stationary ergodic process is a stochastic process that does not change its statistical properties with time and can be deduced from a single sufficient sample of the process. Regarding a stationary ergodic language, we mean that the use of word doesn't change, which in fact is not necessarily true in reality, and can be summarized with a large enough text file, say Wikipedia. Therefore, the above calculation is just an approximation to the correct distributions and entropies of natural language.

Given a language L, we can use a neural network to learn some language model m. Assume the language L has some intrinsic probability p for each sentence. Then we can calculate the **cross entropy** in order to evaluate the effectiveness of a language model, defined as

$$H(p, m) = \lim_{n\to\infty} -\frac{1}{n}\sum_{W^n\in L} p(W^n)\log m(W^n) \tag{2.20}$$

Again, by the Shannon-McMillan-Breiman theorem, we have

$$H(p, m) = \lim_{n\to\infty} -\frac{1}{n}\log m(\omega_1, \ldots, \omega_n) \tag{2.21}$$

Notice that we have the entropy of the true distribution of the language L as the lower bound of the cross entropy because the relative entropy $D(P\|M) \geq 0$ by the theorem of information inequality.

$$\begin{aligned}
H(p, m) &= \lim_{n\to\infty} -\frac{1}{n}\sum_{W^n\in L} p(W^n)\log m(W^n) \\
&= \lim_{n\to\infty} -\frac{1}{n}\sum_{W^n\in L} p(W^n)\left[\log p(W^n) + \log m(W^n) - \log p(W^n)\right] \\
&= \lim_{n\to\infty} -\frac{1}{n}\sum_{W^n\in L} p(W^n)\left[\log p(W^n) + \frac{\log m(W^n)}{\log p(W^n)}\right] \\
&= \lim_{n\to\infty} -\frac{1}{n}\sum_{W^n\in L} p(W^n)\log p(W^n) + \lim_{n\to\infty}\frac{1}{n}\sum_{W^n\in L} p(W^n)\frac{\log p(W^n)}{\log m(W^n)} \\
&= H(p) + D(p\|m)
\end{aligned} \tag{2.22}$$

Therefore, the approximation to the cross entropy of a model M on a sequence of words W of length n is

$$H(W) = -\frac{1}{N}\log m(\omega_1, \ldots, \omega_N) \tag{2.23}$$

The best way to evaluate a language model is to deploy it on a downstream task and measure how much the task improves. Such end-to-end evaluation is called **extrinsic evaluation**(Jurafsky & Martin 2009). Extrinsic evaluation is the only way to confirm such an improvement of the model. Thus we can compare two language models by running them separately on the same task, say machine translation, and see which gives a more accurate translation. However, running such downstream tasks is always very expensive. Instead, we need a quick tool to glance the performance of a language model and the potential improvements once we have it. An **intrinsic evalution** metric is one that measures the quality of a model independent of any application. **Perplexity** is always considered as a good candidate of the intrinsic evaluation metric because perplexity often correlates with extrinsic improvements(Jurafsky & Martin 2009). It does not guarantee such an improvement, but as a quick tool to check the potential improvement, perplexity works pretty well.

Perplexity is defined as the exponential of the approximation of the cross entropy

$$\begin{aligned} PPL(W) &= 2^{H(W)} \\ &= m(\omega_1, \ldots, \omega_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{m(\omega_1, \ldots, \omega_N)}} \\ &= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{m(\omega_i|\omega_1, \ldots, \omega_{i-1})}} \end{aligned} \tag{2.24}$$

Example 2.2.2. Recall the professor example in the beginning of this section. We can further show that the first sentence is more probable than the second one quantitatively by comparing their perplexity. We use our trained GPT2 model based on BBPE tokenization method to calculate probabilities of each token.

First sentence:

token	probability
'the'	9.33e-5
' professor'	20.11e-5
' says'	11.82e-5
' that'	188.04e-5
' doing'	4.84e-5
' research'	537.65e-5
' is'	2.15e-5
' fun'	10.72e-5

The perplexity of the first sentence is 5014.56

Second sentence:

token	probability
‘the’	9.33e-5
‘ pref’	2.58e-6
‘ass’	26.04e-5
‘or’	174.83e-5
‘ say‘	6.12e-5
‘ that’	422.67e-5
‘ doing’	5.59e-5
‘ research’	205.31e-5
‘ is’	2.60e-5
‘ fun’	23.56e-5

The perplexity of the second sentence is 5889.14

Clearly, the first sentence has a much lower perplexity than the second sentence, which indicates the fact that the first sentence is more probable.

Chapter 3

Tokenization Methods

3.1 Byte Pair Encoding

3.1.1 Origin

Byte Pair Encoding(BPE), one of the most frequently used tokenization algorithms in the field of natural language processing, was originally created by Philip Gage (Gage 1994) for data compression. It provides almost as much compression as the popular LZW method. BPE has slower speed in compression than LZW's, but it outspeeds LZW in expansion. The main advantage of BPE is the small, fast expansion routine, which is always utilized by applications with limited memory.

Like many compression algorithms, which replace frequently occurring bit patterns with shorter representations, BPE replaces common pairs of bytes, where each *byte* is defined as a single character or letter, by single bytes which do not occur in the original data. Notice that the algorithm requires a table of replacements in order to construct the original data.

Take the data *aaabdaaababc* as the example. As we count each pair of byte like *aa*, *ab*, etc, we observe that *aa* pair is the most frequent pair, appearing 4 times. Say we use Z to replace *aa*, then

$$aaabdaaababc \rightarrow ZabdZababc \tag{3.1}$$

Note that we replace the first occuring pair if two pairs overlap.
Then we count each pair again and observe that the *ab* pair is now the most frequent pair, and we replace it with Y:

$$ZabdZababc \rightarrow ZYdZYYc \tag{3.2}$$

Then we continue to count the pairs. It turns out at this point ZY is the most frequent pair, so we replace it with X:

$$ZYdZYYc \rightarrow XdXYc \tag{3.3}$$

Therefore, our original data *aaabdaaababc* is compressed to $XdXYc$. Notice that we stop when there's no pair that occurs more than once. The replacement table is

aa:	Z
ab:	Y
ZY:	X

3.1.2 BPE for Tokenization

The application of BPE to tokenization was introduced after the ascent in popularity of neural machine translation. The translation of rare words was still difficult to deal with because of the out-of-vocabulary problem. It turned out that to conquer further translation tasks, a mechanism which goes below the word level was required. Sennrich(Sennrich et al. 2016) therefore applied *byte pair encoding* to the task of word tokenization. It allows for the representation of an open vocabulary through a fixed-size vocabulary of variable-length character sequences, and thus avoids the out-of-vocabulary problem.

To apply BPE to the tokenization task, we merge characters or character sequences, and treat each token as a single byte.

First, we initialize a vocabulary with all the characters we need, and represent each word in the text data as a sequence of characters with an end of word symbol. Iteratively do the following: count the token pairs(notice that a single character is also treated as a token) and merge every occurrence of the most frequent pair and add the newly-merged token into the vocabulary until the vocabulary achieves the desired size.

Algorithm 1 BPE tokenization

1. Initialize the vocabulary.
2. Represent each word with a sequence of character plus an end of word symbol.
3. Count each pair of tokens
4. Merge the most frequent token pair, and add the new merged token into the vocabulary.
5. Repeat steps 3-4 until the vocabulary reaches the desired size.

Let's have an example to further illustrate the details.

Example 3.1.1. Apply BPE tokenization on a bag of words $\{low, lower, higher, lowest, bigger\}$

We first initialize the vocabulary with all 26 English letters, plus the end of word symbol #. The words looks

$$\{\text{'l o w \#'}, \text{'l o w e r \#'}, \text{'h i g h e r \#'}, \text{'l o w e s t \#'}, \text{'b i g g e r \#'}\} \tag{3.4}$$

After the first iteration, we count the token pairs and merge 'l' and 'o' since they appears the most times. The words then look like

$$\{\text{'lo w \#'}, \text{'lo w e r \#'}, \text{'h i g h e r \#'}, \text{'lo w e s t \#'}, \text{'b i g g e r \#'}\} \tag{3.5}$$

After the second iteration, we count the token pairs and merge 'lo' and 'w' since they appears the most times. The words then look like

$$\{\text{'low \#'}, \text{'low e r \#'}, \text{'h i g h e r \#'}, \text{'low e s t \#'}, \text{'b i g g e r \#'}\} \tag{3.6}$$

After the third iteration, we count the token pairs and merge 'e' and 'r' since they appears the most times. In fact the ,The words then look like

{'low #', 'low er #', 'h i g h er #', 'low e s t #', 'b i g g er #'} (3.7)

After the third iteration, we count the token pairs and merge 'er' and '#' since they appears the most times. In fact the ,The words then look like

{'low #', 'low er#', 'h i g h er#', 'low e s t #', 'b i g g er#'} (3.8)

We could just stop here as we want. Notice that we are obtaining interesting subwords like 'low' and 'er#', subwords that have meaning of their own.

3.1.3 Byte Level BPE

As researchers explored the application of BPE to various multilingual NLP tasks, they found that a base vocabulary that includes all the possible Unicode characters can be quite large. In fact there are 138K characters covering over 150 languages. It turns out that we can further squeeze the size of the vocabulary by converting all of the Unicode characters into their UTF-8 byte forms, and therefore limit the size of the base vocabulary to 256. This allows us to model a sentence in a sequence of bytes. If then we apply the byte pair encoding on the byte form of sentence, we reach the idea called *byte-level BPE*(Wang et al. 2019).

Take our last example.

Example 3.1.2. Apply BBPE tokenization on a bag of words $\{low, lower, higher, lowest, bigger\}$.

We first need to encode the words into their byte forms.

{'6C 6F 77 23', '6C 6F 77 65 72 23', ..., '62 69 67 67 65 72 23'} (3.9)

Notice that '23' is encoding the end of word symbol '#'.
Then, we just simply apply BPE on it and we will get our tokenizer.

3.2 Unigram Language Model

3.2.1 Parameter Estimation

Parameters are descriptive measures of an entire population. However, parameters are usually unknown since it's impossible to measure the entire population. We can estimate parameters by taking samples from the population. For example, a student received a biased coin and wanted to figure out how biased it is. He did an experiment where he tossed the coin a thousand times and observed that the coin landed head 600 times. One reasonable estimation of the probability of the biased coin landing head would be $P(\text{coin lands head}) = 0.6$ given his observations.

However, there are cases where the observed results can not give us the estimation fairly easily. Suppose the student received three biased coins A, B, C. His parents did

the following experiment for him: They first tossed coin A. If coin A landed head, they would toss coin B and record the result. If coin A landed tail, they would toss coin C and record the result. The student could only observe the resulting heads and tails but he had no idea which coin the result came from. This time, it's really hard for the student to estimate the probability for three coins because there are hidden variables.

3.2.2 EM Algorithm

The Expectation-Maximization(EM) algorithm (Dempster et al. 1977) is an iterative method to solve difficult maximum likelihood problems, involving unobserved variables. The key idea of the algorithm is as follows:

1. Consider the general problem of maximizing a tricky function $f(\theta)$ as shown by Figure 3.1. Suppose we have an initial guess of the maximal value θ'.

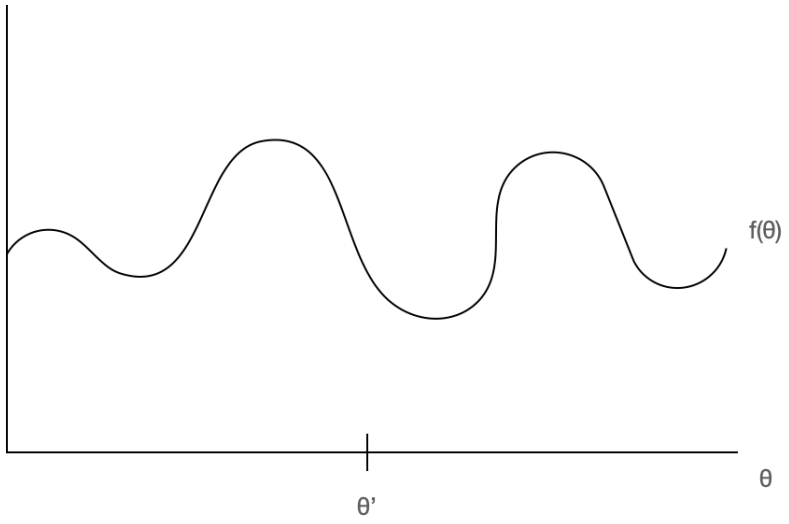

Figure 3.1: A tricky function $f(\theta)$with an initial guess θ'

2. No matter what the guess is, EM algorithm can always come up with a simpler function $g(\theta : \theta')$ shown by Figure 3.2 such that $g(\theta : \theta')$ is a lower bound of $f(\theta)$ and $g(\theta' : \theta') = f(\theta')$.

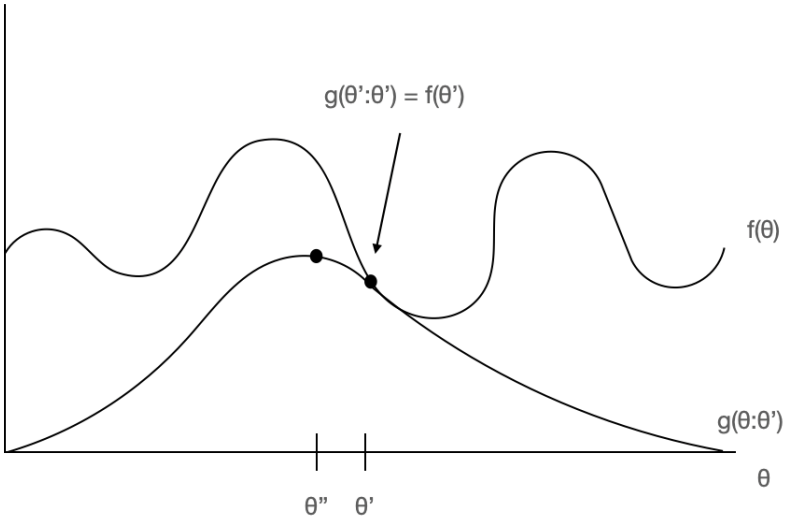

Figure 3.2: The simpler function by EM algorithm

3. Once we have the simpler function $g(\theta : \theta')$, we can easily find the maximum point of it because it's constructed with the last guessed maximum point. The maximum point of the simpler function $g(\theta : \theta')$ will be our next guess of the maximum of $f(\theta)$. Notice that θ'' is at least as good as θ' because

$$f(\theta') = g(\theta' : \theta') \leq g(\theta'' : \theta') \leq f(\theta''). \tag{3.10}$$

4. Based on the new guess θ'', we can construct the new simpler function $g(\theta : \theta'')$ and find a new guess of the maximum. Using this technique, our guesses never get worse. It doesn't guarantee to find a global maximum of $f(\theta)$, but it can find a local maximum.

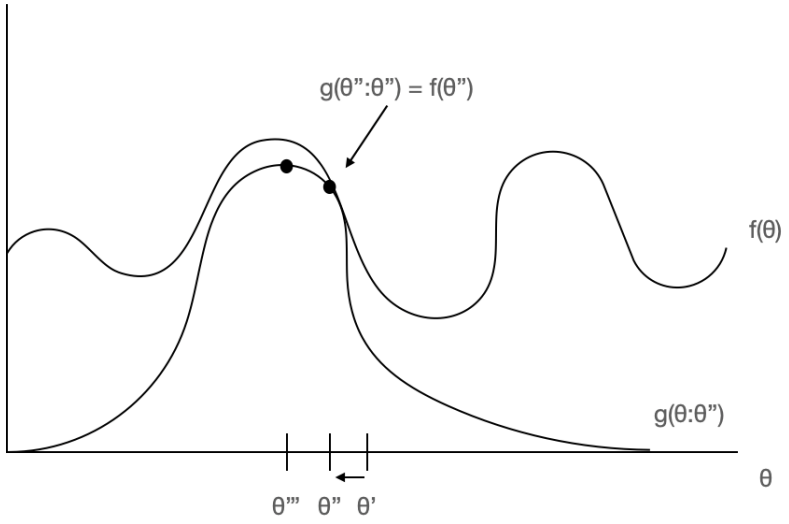

Figure 3.3: Construct another simpler function based on the new guess

Algorithm 2 EM Algorithm

1. Choose initialization $\theta^{(0)}$
2. E-step: Let $\theta^{(i)}$ be the approximation of θ from the i-th iteration. For the $i+1$-th iteration, calculate

$$\begin{aligned} Q(\theta, \theta^{(i)}) &= E_Z[\log P(Y, Z \mid \theta) \mid Y, \theta^{(i)}] \\ &= \sum_Z \log P(Y, Z \mid \theta) P(Z \mid Y, \theta^{(i)}) \end{aligned}$$

3. M-step: Find the value of θ which maximizes $Q(\theta, \theta^{(i)})$ and set it as the approximation $\theta^{(i+1)}$ for the next iteration

$$\theta^{(i+1)} = \underset{\theta}{\operatorname{argmax}}\, Q(\theta, \theta^{(i)})$$

4. Repeat steps 2-3 until the convergence condition is achieved.

Let me further illustrate the EM algorithm in detail. Say we have a model which contains hidden variables and we want to approximate the parameters of the model based on observed data from the model. Let Y represent the observed random variables and let Z represent the hidden random variables. Then $P(Y \mid \theta)$ is the probability

function of observed variables, where θ is the model's parameters. $P(Y, Z \mid \theta)$ is the joint probability function of observed and hidden variables. The EM algorithm aims to estimate $\hat{\theta} = \text{argmax}_\theta P(Y \mid \theta)$.

Definition 3.2.1 (Q function)**.** The *Q function* is defined as the expectation of the log-likelihood of the complete data $\log P(Y, Z \mid \theta)$ over the conditional distribution $P(Z \mid Y, \theta^{(i)})$,

$$Q(\theta, \theta^{(i)}) = E_Z[\log P(Y, Z \mid \theta) \mid Y, \theta^{(i)}] \tag{3.11}$$

How we derive the Q function and the EM algorithm? The following part will show readers the derivation of the EM algorithm.

When people want to estimate the parameters of a probabilistic model with hidden variables, one way is to maximize the log-likelihood function of the observed data, i.e.

$$\begin{aligned} L(\theta) &= \log P(Y \mid \theta) \\ &= \log \sum_Z P(Y, Z \mid \theta) \\ &= \log \left(\sum_Z P(Y \mid Z, \theta) P(Z \mid \theta) \right) \end{aligned} \tag{3.12}$$

In fact, EM algorithm is maximizing $L(\theta)$ iteratively and the approximate value of θ after i-th iteration is $\theta^{(i)}$. Therefore, consider the difference between the two likelihood functions:

$$\begin{aligned} L(\theta) - L(\theta^{(i)}) &= \log \left(\sum_Z P(Y \mid Z, \theta) P(Z \mid \theta) \right) - \log P(Y \mid \theta^{(i)}) \\ &= \log \left(\sum_Z P(Z \mid Y, \theta^{(i)}) \frac{P(Y \mid Z, \theta) P(Z \mid \theta)}{P(Z \mid Y, \theta^{(i)})} \right) - \log P(Y \mid \theta^{(i)}) \\ &= \log \left(\sum_Z P(Z \mid Y, \theta^{(i)}) \frac{P(Y \mid Z, \theta) P(Z \mid \theta)}{P(Z \mid Y, \theta^{(i)}) P(Y \mid \theta^{(i)})} \right) \\ &\geq \sum_Z P(Z \mid Y, \theta^{(i)}) \log \frac{P(Y \mid Z, \theta) P(Z \mid \theta)}{P(Z \mid Y, \theta^{(i)}) P(Y \mid \theta^{(i)})} \end{aligned} \tag{3.13}$$

Notice that the inequality comes from Jensens Inequality. Let

$$B(\theta, \theta^{(i)}) = L(\theta^{(i)}) + \sum_Z P(Z \mid Y, \theta^{(i)}) \log \frac{P(Y \mid Z, \theta) P(Z \mid \theta)}{P(Z \mid Y, \theta^{(i)}) P(Y \mid \theta^{(i)})}, \tag{3.14}$$

Then,

$$L(\theta) \geq B(\theta, \theta^{(i)}) \tag{3.15}$$

Importantly, notice that $B(\theta, \theta^{(i)})$ is a lower bound of function $L(\theta)$. If we plug in $\theta = \theta^{(i)}$ into Eqn 3.14, we have

$$L(\theta^{(i)}) = B(\theta^{(i)}, \theta^{(i)}), \tag{3.16}$$

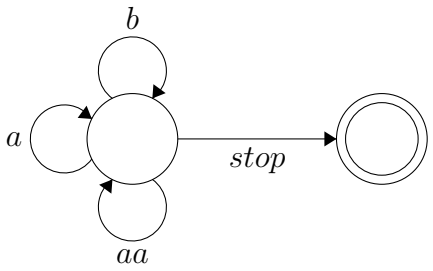

Figure 3.4: The simple FSA

since

$$\frac{P(Y \mid Z, \theta^{(i)})P(Z \mid \theta^{(i)})}{P(Z \mid Y, \theta^{(i)})P(Y \mid \theta^{(i)})} = 1 \tag{3.17}$$

Therefore, for each iteration, we choose the θ that can maximize $B(\theta, \theta^{(i)})$, which in turn maximizes $L(\theta)$ as well. So,

$$\theta^{(i+1)} = \underset{\theta}{\operatorname{argmax}}\, B(\theta, \theta^{(i)}) \tag{3.18}$$

By Eqn 3.18, 3.14, 3.11, we have

$$\begin{aligned}
\theta^{(i+1)} &= \underset{\theta}{\operatorname{argmax}} \left(L(\theta^{(i)}) + \sum_{Z} P(Z \mid Y, \theta^{(i)}) \log \frac{P(Y \mid Z, \theta)P(Z \mid \theta)}{P(Z \mid Y, \theta^{(i)})P(Y \mid \theta^{(i)})} \right) \\
&= \underset{\theta}{\operatorname{argmax}} \left(\sum_{Z} P(Z \mid Y, \theta^{(i)}) \log(P(Y \mid Z, \theta)P(Z \mid \theta)) \right) \\
&= \underset{\theta}{\operatorname{argmax}} \left(\sum_{Z} P(Z \mid Y, \theta^{(i)}) \log P(Y, Z \mid \theta) \right) \\
&= \underset{\theta}{\operatorname{argmax}}\, Q(\theta, \theta^{(i)})
\end{aligned} \tag{3.19}$$

3.2.3 Application of EM on a simple FSA

Let me illustrate it further by applying EM to a simple finite state machine shown by Figure 3.4, where we assume that the true probabilities are:

$$\begin{cases}
P(aa) = \dfrac{2}{10}, \\
P(a) = \dfrac{1}{10}, \\
P(b) = \dfrac{4}{10}, \\
P(stop) = \dfrac{3}{10}.
\end{cases} \tag{3.20}$$

Therefore, as we observe, the machine can generate string data such as “aaaa”, “aaba”, “bbaaaa”...

However, suppose we don't know the true probability of the transitions beforehand, Instead we receive a string dataset $S = \{\text{“}aaaa\text{”}, \text{“}aaba\text{”}, \text{“}bbaaaa\text{”}, ...\}$, where each string was generated by the FSA. For example, “aaba” could be constructed as 'aa'+'b'+'a'+'stop' or 'a'+'a'+'b'+'a'+'stop'.

Define a finite state machine $M = (\Sigma, Q, q_0, \delta, F)$ as Figure 3.4 shows. Notice that $\Sigma = \{aa, a, b, stop\}$, $\delta : Q \times \Sigma \to Q$. Define $\theta(\sigma) = P(\sigma), \forall \sigma \in \Sigma$. Let's have $\forall s \in S, \exists \pi$ such that $\pi(s)$ is the path of the real construction of s such as {'aa','b','a','stop'} for 'aaba'. At last, we define $c_\sigma(\pi(s)) =$ number of transition σ appears in $\pi(s)$.

Our ultimate goal in this task is to maximize the probability of the string data we're given, i.e. maximize $\prod_{s\in S} P(s;\theta)$. Here, the observed variable Y is every string in the string dataset $s \in S$ and the hidden variable Z is the real construction for each string $\pi(s)$. Therefore, we have

$$\begin{aligned}\underset{\theta}{\operatorname{argmax}} \prod_{s\in S} P(s;\theta) &= \underset{\theta}{\operatorname{argmax}} \sum_{s\in S} \log P(s;\theta) \\ &= \underset{\theta}{\operatorname{argmax}} \sum_{s\in S} \log \sum_{\pi(s)} P(s, \pi(s);\theta)\end{aligned} \tag{3.21}$$

Observe that

$$\begin{aligned}\log \sum_{\pi(s)} P(s, \pi(s);\theta) &= \log \sum_{\pi(s)} \left(\frac{P(\pi(s) \mid s;\theta')}{P(\pi(s) \mid s;\theta')}\right) P(s, \pi(s);\theta) \\ &= \log \sum_{\pi(s)} P(\pi(s) \mid s;\theta') \left(\frac{P(s, \pi(s);\theta)}{P(\pi(s) \mid s;\theta')}\right) \\ &\geq \sum_{\pi(s)} P(\pi(s) \mid s;\theta') \log \left(\frac{P(s, \pi(s);\theta)}{P(\pi(s) \mid s;\theta')}\right) \equiv g(\theta \mid s;\theta')\end{aligned} \tag{3.22}$$

The inequality is from Jensen Inequality where log is a convex function.

Therefore, summing up together every string we're given still satisfies the above inequality:

$$\sum_{s\in S} \log \sum_{\pi(s)} P(s, \pi(s);\theta) \geq \sum_{s\in S} g(\theta \mid s;\theta') \tag{3.23}$$

Then we have finished the E-step, where the right hand side of the above inequality is our Q-function in this case. Next, we only need to find the parameters that maximize

it:

$$\begin{aligned}
\operatorname*{argmax}_{\theta} \sum_{s\in S} g(\theta \mid s;\theta') &= \operatorname*{argmax}_{\theta} \sum_{s\in S}\sum_{\pi(s)} P(\pi(s) \mid s;\theta') \log\left(\frac{P(s,\pi(s);\theta)}{P(\pi(s) \mid s;\theta')}\right) \\
&= \operatorname*{argmax}_{\theta} \sum_{s\in S}\sum_{\pi(s)} P(\pi(s) \mid s;\theta') \log P(s,\pi(s);\theta) \\
&= \operatorname*{argmax}_{\theta} \sum_{s\in S} \mathop{\mathbf{E}}_{P(\pi(s)|s;\theta')} \log P(s,\pi(s);\theta) \\
&= \operatorname*{argmax}_{\theta} \sum_{s\in S} \mathop{\mathbf{E}}_{P(\pi(s)|s;\theta')} \log \prod_{\sigma\in\Sigma} \theta_{\sigma}^{c_{\sigma}(\pi(s))} \\
&= \operatorname*{argmax}_{\theta} \sum_{s\in S} \mathop{\mathbf{E}}_{P(\pi(s)|s;\theta')} \sum_{\sigma\in\Sigma} c_{\sigma}(\pi(s)) \log\theta_{\sigma} \\
&= \operatorname*{argmax}_{\theta} \sum_{\sigma\in\Sigma} \log\theta_{\sigma} \sum_{s\in S} \mathop{\mathbf{E}}_{P(\pi(s)|s;\theta')} c_{\sigma}(\pi(s))
\end{aligned} \tag{3.24}$$

Finally, the algorithm looks like the following,

Algorithm 3 EM on simple FSA

function EM(S)
  Guess a true probability $\hat{\theta}$
  repeat
    $\hat{\theta} = \operatorname{argmax}_{\theta} \sum_{\sigma\in\Sigma} \log\theta_{\sigma} \sum_{s\in S} \mathop{\mathbf{E}}_{P(\pi(s)|s;\hat{\theta})} c_{\sigma}(\pi(s))$
  until satisfied **return** $\hat{\theta}$.
end function

We implement the above algorithm on the FSA shown by Figure 3.4. Here's the different results if we feed different numbers of generated strings to the algorithm.

Num of strings	P(a)	P(aa)	P(b)	P(stop)
10	0.186	0.198	0.274	0.342
100	0.106	0.182	0.409	0.303
1000	0.119	0.188	0.386	0.306
10000	0.104	0.201	0.400	0.295

Table 3.1: Estimated probabilities for each symbol in FSA

Notice that the estimated probabilities by EM algorithm converges to the true probabilities with not too many training strings.

3.2.4 Unigram Language Model

Unigram language model (Kudo 2018) is a language model such that each subword occurs independently and therefore the probability of a subword sequence $\mathbf{x} =$

$(x_1, x_2, ..., x_M)$ is

$$\begin{aligned} P(\mathbf{x}) &= \prod_{i=1}^{M} p(x_i) \\ \forall i \ x_i \in \mathcal{V}, &\sum_{x \in \mathcal{V}} p(x) = 1, \end{aligned} \tag{3.25}$$

where $\mathcal{V}$ is the pre-determined vocabulary.

If the vocabulary $\mathcal{V}$ is given, we can estimate the subword occurrence probabilities $p(x_i)$ using EM algorithm, where sentence corpus are observed variables and real tokenizations for each sentences are hidden variables. Let $X^{(s)}$ be the s-th sentence in the sentence corpus and $S(X^{(s)})$ be a set of segmentation candidates built from the input sentence $X^{(s)}$. Then we have the log likelihood function for EM algorithm:

$$L(\theta) = \sum_{s=1}^{|D|} \log(P(X^{(s)}; \theta)) = \sum_{s=1}^{|D|} \log \left(\sum_{\mathbf{x} \in \mathcal{S}(X^{(s)})} P(\mathbf{x}; \theta) \right) \tag{3.26}$$

However, when we train a language model, the vocabulary will not be given as granted. We therefore hope to find them iteratively.

1. Heuristically make a seed vocabulary from the training corpus.
2. Fix a set of vocabulary. Find $p(x)$ using EM algorithm.
3. For each subword in the vocabulary, calculate how much the likelihood function reduced if the subword is removed from the vocabulary.
4. Sort subwords by top $\eta\%$ reducer of the likelihood function. We always keep the subwords of single character to avoid out-of-vocabulary problem.
5. Repeat step 2-4 until $|\mathcal{V}|$ reaches a desired size.

Notice that when generating a seed vocabulary, a natural choice is to use the union of all single characters and the most frequent subwords in the corpus.

Chapter 4

Experiment and Conclusion

4.1 Model Selection

The model we chose was GPT-2 (Radford et al. 2019), which is a general-purpose language model that can predict the next token given a sequence of input tokens. GPT-2 was created to be the basis for NLP systems across many tasks and domains. It intends to be trained on multiple domains and tasks so that it doesn't need extra data to reform the model in order to perform well in a specific task.

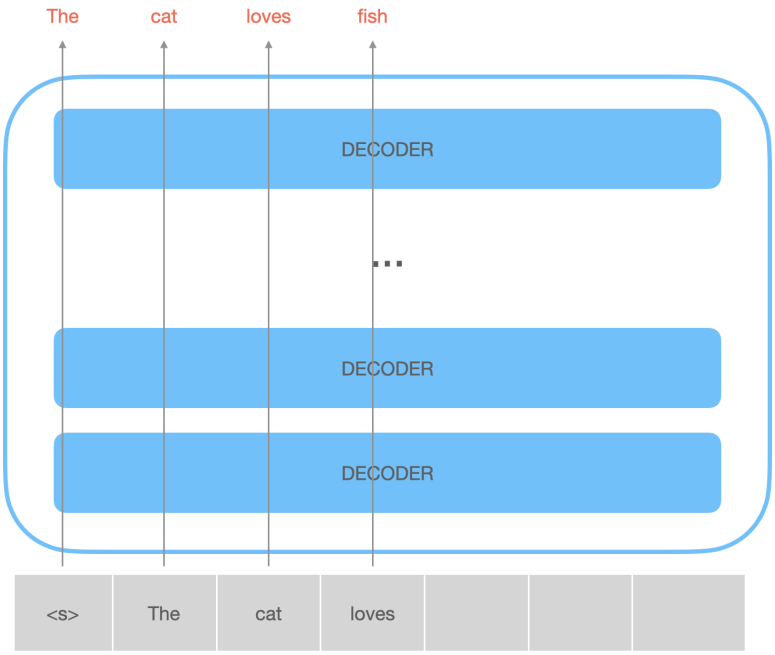

Figure 4.1: GPT-2 model at a high level

Figure 4.1 shows us how the GPT2 model works at a high level. The model is given a context, i.e. a sequence of tokens, and then it outputs the next token predicted by the model: 'fish'. This allows the use of GPT-2 for language generation.

Example 4.1.1. Here are some examples.

> Initial context: *A train carriage containing controlled nuclear materials was stolen in Cincinnati today. Its whereabouts are unknown.*
>
> GPT2 output: *The incident occurred on the downtown train line, which runs from Covington and Ashland stations.*

We choose GPT-2 for the following reasons.

1. GPT-2 is a general system which can perform multiple tasks. Therefore, we can explore the differences between word-based language and character-based language without the bias of specific tasks.

2. GPT-2 uses cross-entropy as the loss function, which we will use as the evaluation metric for models we train. BERT model, however, is hard to compute cross-entropy based on the model's desgin.

3. The performance of GPT-2 on many downstream tasks such as reading comprehension and question answering is competitive with supervised baselines in a zero-shot setting (Radford et al. 2019). Therefore, we can use trained GPT-2 models and fine-tune them on these downstream tasks to further evaluate differences of whitespace languages and non-whitespace languages for future work.

4.2 How it works?

4.2.1 Embeddings

The input to GPT-2 is a sequence of tokens. We first translate it into a corresponding sequence of ids using our pre-built tokenizers. Then we find the token embedding associated with each id by multiplying its one-hot vector with the token embeddings matrix in Figure 4.2. Notice that token embeddings will not be enough to capture the order of the given sequence of tokens. Without the positional embedding, GPT-2 would treat 'the dog bit the man' and 'the man bit the dog' as the same input. Therefore, we also give each position a unique position embedding according to the position embeddings matrix in Figure 4.2.

Finally, we sum the token and position embeddings and obtain a $1 \times C$ vector for each token, where we set $C = 768$ in our model.

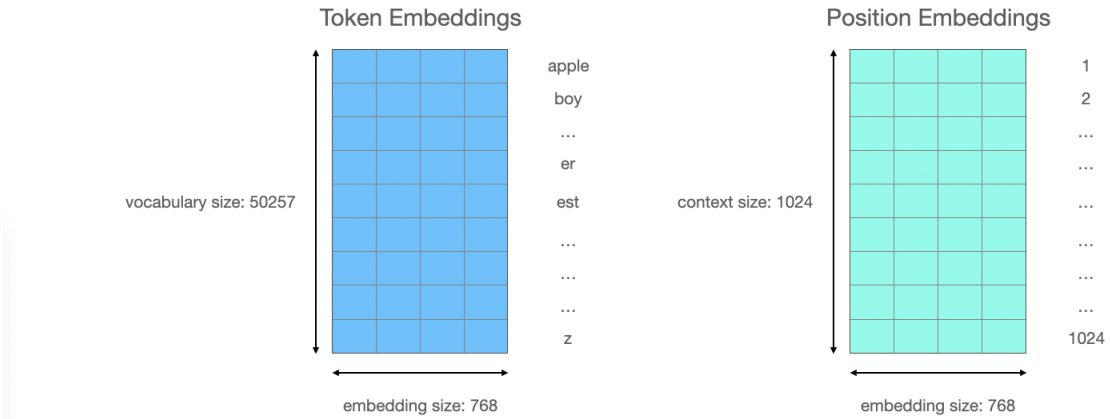

Figure 4.2: Input embeddings for GPT2 model

Notice that the entries of the token embeddings matrix and the position embeddings matrix will be trained during the training process, rather than set manually.

4.2.2 Masked Self-Attention

After the embedding process, we reach the Masked Self-Attention block, which is the first chunk of the decoder block. As shown by Figure 4.3, when we process the embedding of the token 'fun', the Masked Self-Attention block assigns each previous token a relevance score related to 'fun'. The score ranges from 0 to 1, where scores closer to 1 represent high relevance to 'fun' and scores closer to 0 represent less relevance to 'fun'. For example, the token 'professor' might be scored highly because it is the subject of the token 'fun'. The token 'research' might also be scored highly because it is being described by the token 'fun'. Then the block uses these scores to compute the weighted average of each previous embeddings. Basically we can treat Masked Self-Attention block as a means of weighting the importance of previous tokens.

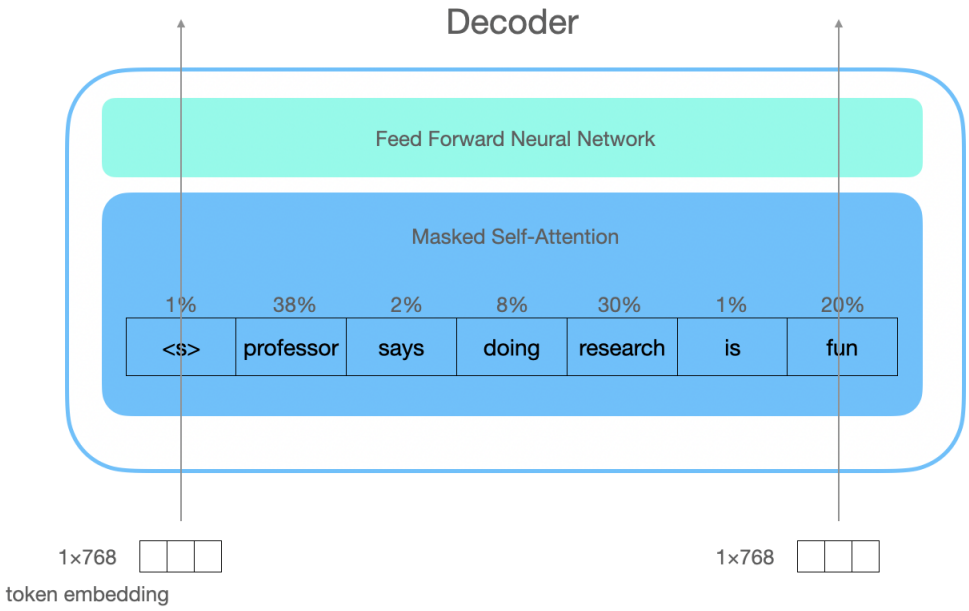

Figure 4.3: Decoder block at a high level when processing the token 'fun'

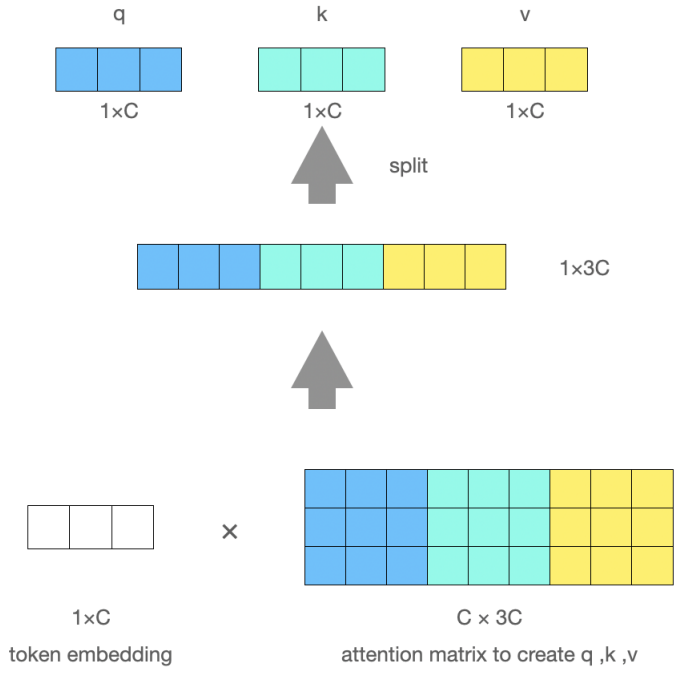

Figure 4.4: Creating q, k, v vector for a token. In the diagram, $C = 3$, but GPT-2 uses $C = 768$.

Let's explore more details in the Masked Self-Attention block. As shown by Figure 4.4, we multiply each token embedding with the attention matrix to create three vectors: q, k, v (short for "query", "key", and "value").

Then we can simply use matrix multiplication and apply the softmax function to get scores for each token embedding as illustrated by Figure 4.5. For example, the first row of the matrix after applying the softmax function represents the scores of the first token on the first five tokens. However, when we calculate the scores of a token, we need to mask the information of tokens after it so that we don't use the future to predict the past. Say we are given a sequence of length 4 (A,B,C,D), and the model uses A to predict B, and uses (A,B) to predict C. It is necessary to mask (B,C,D) when we use A to predict B because otherwise the prediction gains foresight.

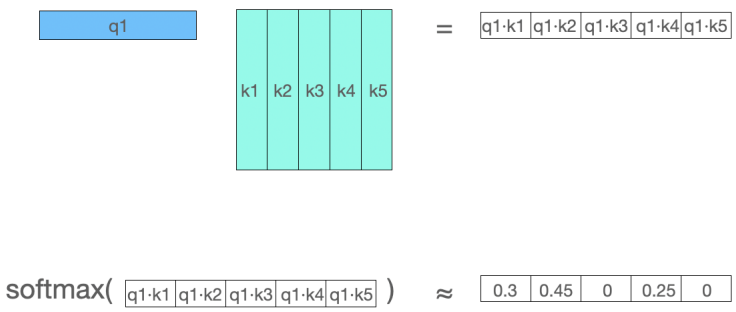

Figure 4.5: Calculating relevance scores for a token with query embedding q_1. k_1,...,k_5 are key embeddings for each token in the input.

Therefore, as illustrated by Figure 4.6, before we apply the softmax function, we replace every element above the matrix diagonal with a very small number, e.g. '-inf'. At last, once we have the scores, we can simply calculate the weighted averages for each token and then apply a projection matrix to the weighted averages to get the final output for the next layer as shown by Figure 4.7. We apply the projection matrix because it will let our model learn how to best map the resulting weighted average into a vector that can proceeded by the next layer.

Notice that the attention matrix and projection matrix are weight matrices that will be trained during the training process, rather than set manually.

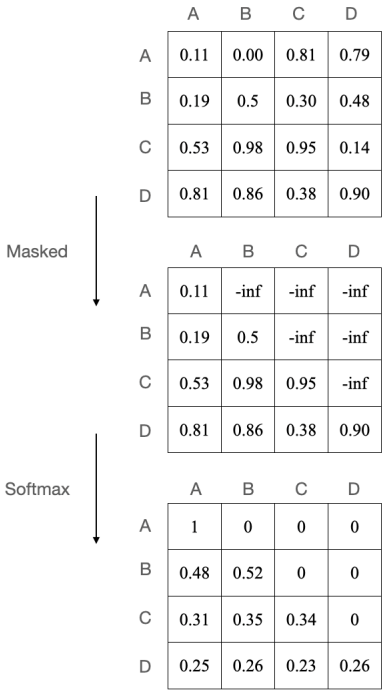

Figure 4.6: Applying masked function before softmax.

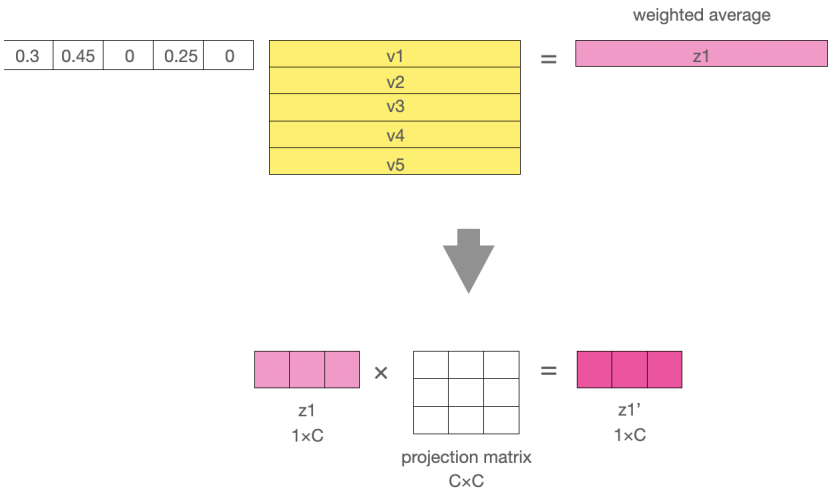

Figure 4.7: Calculating the weighted average and outputting the final vector after the projection step. v_1, ..., v_5 are value embeddings for each token in the input.

4.2.3 Feed Forward Neural Network

The second block in a decoder is Feed Forward Neural Network. This is the place the model processes information after the input has incorporated the context of the given sequence.

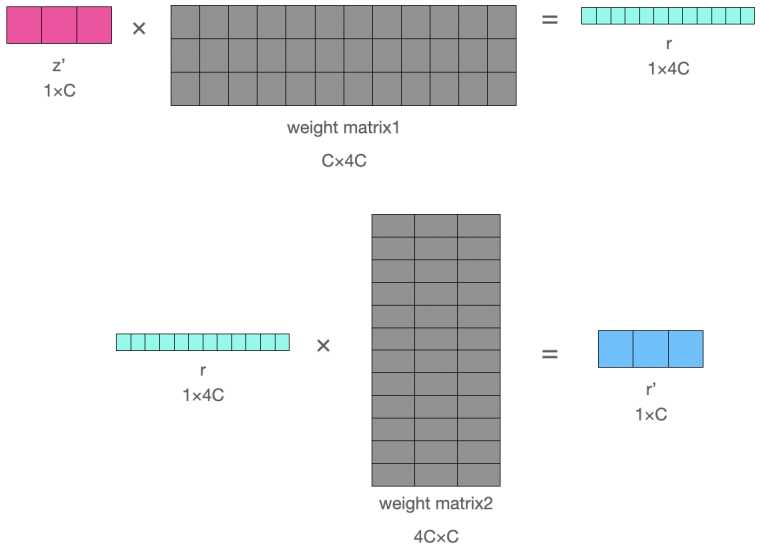

Figure 4.8: Feed Forward Neural Network at a high level

As noted by Figure 4.8, Feed Forward Neural Network block receives the final output z' from the Masked Self-Attention block. It maps the input vector into a vector that is 4 times larger and then applies another transformation back to a vector of the embedding size. Then it outputs r' to the next part of GPT-2. This process is simply a fully-connected neural network with 1 hidden layer whose width is 4 times as the input size. Both weight matrix1 and weight matrix2 are trained during the training process.

GPT2 model is built with 12 decoders stacked together as shown by Figure 4.3. The next decoder receives the output of the previous decoder. We multiply the output of the last decoder with the transpose of the token embedding matrix from Figure 4.2 to get a vector of the vocabulary size. If then we apply the softmax function on this vector, we will have the probability of each token in the vocabulary as the next predicted token, where we could choose the token with the highest probability as the predicted token.

4.3 Result and Analysis

We trained 4 different models with the English corpus of WMT'14 English-German data (Luong et al. 2015) and tested the models on its corresponding test datasets. We used the exactly same set of hyper-parameters as the original smallest GPT2 model (Radford et al. 2019) when we trained the GPT2 with the Byte-level BPE tokenization method. However, GPT2 model with Unigram LM tokenization method required more hyper-parameter tuning. We chose to use the SGD optimizer to train the model instead of the AdamW optimizer that the original model used. The evaluation of perplexities for each model is shown by Table 4.1 below.

Test dataset	BBPE whitespace	BBPE non-whitespace	UniLM whitespace	UniLM non-whitespace
newstest2012	109.7862	475.4579	30.2481	33.7664
newstest2013	95.0274	442.1453	32.2228	36.9390
newstest2014	127.5265	503.2211	30.5277	36.9851
newstest2015	114.5525	458.1697	31.6756	34.0421

Table 4.1: Perplexities of trained models on different testsets

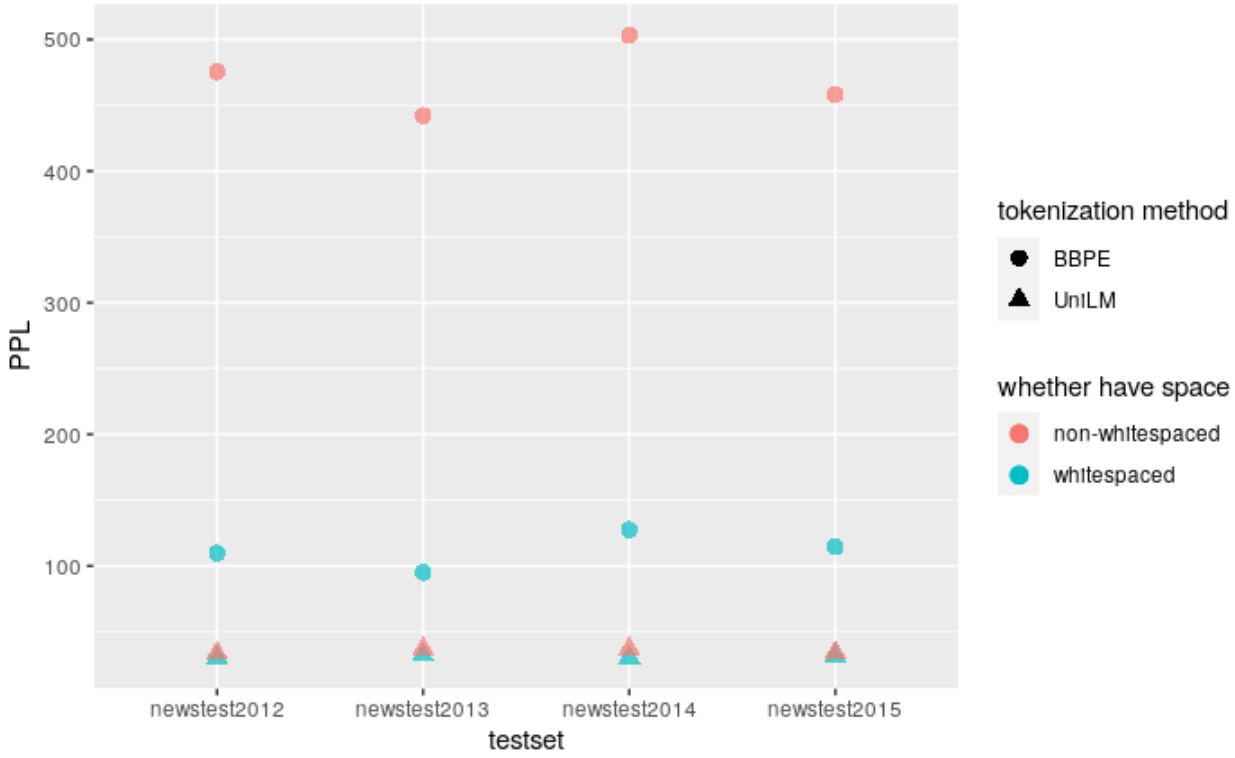

Figure 4.9: Perplexities of trained models on different testsets

Notice that on Figure 4.9, models trained with whitespace training data have lower perplexities than their corresponding non-whitespace models. Specifically, for the

Byte-level BPE tokenization method, the whitespace model decreases the perplexities by 76.27%, averaged across 4 test datasets. For the Unigram LM tokenization method, the whitespace model decreases the perplexities by 11.90%, averaged across 4 test datasets. Both whitespace and non-whitespace models have lower perplexities when the tokenization method is Unigram LM. We have the following opinions on these results:

1. Compared to the perplexity reported in the original GPT-2 paper, which is 19.93, both BBPE and Unigram LM models have much larger perplexities. One reason would be the size of the training data. Because of resource limitations, we only trained our models on a text corpus that is about 1% the size of the training data used by the original GPT-2 model. Therefore, our models are not as general as the original GPT-2 model.

2. We used whitespace-deprived English to mimic character-based languages like Chinese and Japanese. By our results, it is more challenging to train language models(like GPT-2) for whitespace-deprived English.

3. Unigram LM tokenization performs much better than BBPE on non-whitespace corpora. It might suggest that Unigram LM is a better tokenization method when we are dealing with character-based languages. We hypothesize that replacing BBPE with Unigram LM tokenization will get better results on downstream tasks with character-based languages. We are looking forward to testing it in the future.

4. Limited by time, we didn't train GPT-2 models with the BPE tokenization method. However, we conjecture that the result will be similar to the result for the BBPE.

4.4 Conclusion

In this work, we explored the challenges of character-based languages on natural language processing tasks. We used whitespace-deprived English to mimic character-based languages and trained GPT-2 models with both whitespace and corresponding non-whitespace corpora. For each corpus, we applied the Byte-level BPE and Unigram LM tokenization methods to train two different models.

We found that character-based languages are indeed harder to train language models. Unigram LM tokenization seems to be a much better tokenization method than BBPE on character-based languages.

References

Algoet, P. H., & Cover, T. M. (1988). A Sandwich Proof of the Shannon-McMillan-Breiman Theorem. *The Annals of Probability*, *16*(2), 899 – 909.

Aronoff, F. (2007). What is Morphology? *Forum for Modern Language Studies*, *43*(1), 93–93.

Cover, T. M., & Thomas, J. A. (2006a). *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. USA: Wiley-Interscience.

Cover, T. M., & Thomas, J. A. (2006b). *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. USA: Wiley-Interscience.

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, *39*(1), 1–38.

Gage, P. (1994). A new algorithm for data compression. *C Users J.*, *12*(2), 23–38.

Jurafsky, D., & Martin, J. H. (2009). *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition*. Upper Saddle River, N.J.: Pearson Prentice Hall.

Kudo, T. (2018). 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)*, (pp. 66–75). Melbourne, Australia: Association for Computational Linguistics.

Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. In *Empirical Methods in Natural Language Processing (EMNLP)*, (pp. 1412–1421). Lisbon, Portugal: Association for Computational Linguistics.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners.

Sennrich, R., Haddow, B., & Birch, A. (2016). Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (pp. 1715–1725). Berlin, Germany: Association for Computational Linguistics.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need.

Wang, C., Cho, K., & Gu, J. (2019). Neural machine translation with byte-level subwords. *CoRR*, *abs/1909.03341*.