Byte-Pair Encoding: Subword-based tokenization algorithm

Understand subword-based tokenization algorithm used by state-of-the-art NLP models — Byte-Pair Encoding (BPE)

Chetna Khanna · Follow
Published in Towards Data Science · 9 min read · Aug 13, 2021

Photo by Clark on Unsplash

The branch of Artificial Intelligence, **Natural Language Processing (NLP)**, is all about making machines understand and process human language. Processing human language is not an easy task for machines as machines work with numbers and not text. 💻 NLP is such a vast and widely studied branch of AI that every now and then we hear a new advancement in this domain. Researchers are trying hard to make the machines understand human language and the context behind it.

One of the main roles in understanding human language is played by the tokenizers. Tokenization algorithms can be word, subword, or character-based. Each type of tokenizer helps the machines process the text in a different way. Each one has an advantage over the other. If you want to know about the different types of tokenizers used in NLP then you can read this article. This article is a hands-on tutorial on TDS and will give you a good understanding of the topic. 😇

The popular one among these tokenizers is the **subword-based tokenizer**. This tokenizer is used by most state-of-the-art NLP models. So let's get started with knowing first what subword-based tokenizers are and then understanding the Byte-Pair Encoding (BPE) algorithm used by the state-of-the-art NLP models. 🙃

Subword-based tokenization

Subword-based tokenization is a solution between word and character-based tokenization. 😎 The main idea is to solve the issues faced by word-based tokenization (very large vocabulary size, large number of OOV tokens, and

different meaning of very similar words) and character-based tokenization (very long sequences and less meaningful individual tokens).

The subword-based tokenization algorithms do not split the frequently used words into smaller subwords. It rather splits the rare words into smaller meaningful subwords. For example, “boy” is not split but “boys” is split into “boy” and “s”. This helps the model learn that the word “boys” is formed using the word “boy” with slightly different meanings but the same root word.

Some of the popular subword tokenization algorithms are WordPiece, Byte-Pair Encoding (BPE), Unigram, and SentencePiece. We will go through Byte-Pair Encoding (BPE) in this article. BPE is used in language models like GPT-2, RoBERTa, XLM, FlauBERT, etc. A few of these models use space tokenization as the pre-tokenization method while a few use more advanced pre-tokenization methods provided by Moses, spaCY, ftfy. So, let’s get started. 🏃

Byte-Pair Encoding (BPE)

BPE is a simple form of data compression algorithm in which the most common pair of consecutive bytes of data is replaced with a byte that does not occur in that data. It was first described in the article “A New Algorithm for Data Compression” published in 1994. The below example will explain BPE and has been taken from Wikipedia.

Suppose we have data **aaabdaaabac** which needs to be encoded (compressed). The byte pair **aa** occurs most often, so we will replace it with **Z** as **Z** does not occur in our data. So we now have **ZabdZabac** where **Z = aa**. The next common byte pair is **ab** so let’s replace it with **Y**. We now have **ZYdZYac** where **Z = aa** and **Y = ab**. The only byte pair left is **ac** which appears as just one so we will not encode it. We can use recursive byte pair encoding to encode **ZY** as **X**. Our data has now transformed into **XdXac** where **X = ZY, Y = ab,** and **Z = aa**. It cannot be further compressed as there are no byte pairs

appearing more than once. We decompress the data by performing replacements in reverse order.

A variant of this is used in NLP. Let us understand the NLP version of it together. 🤗

BPE ensures that the most common words are represented in the vocabulary as a single token while the rare words are broken down into two or more subword tokens and this is in agreement with what a subword-based tokenization algorithm does.

Suppose we have a corpus that has the words (after pre-tokenization based on space) — old, older, highest, and lowest and we count the frequency of occurrence of these words in the corpus. Suppose the frequency of these words is as follows:

{“old”: 7, “older”: 3, “finest”: 9, “lowest”: 4}

Let us add a special end token “</w>” at the end of each word.

{“old</w>”: 7, “older</w>”: 3, “finest</w>”: 9, “lowest</w>”: 4}

The “</w>” token at the end of each word is added to identify a word boundary so that the algorithm knows where each word ends. This helps the algorithm to look through each character and find the highest frequency character pairing. I will explain this part in detail later when we will include “</w>” in our byte-pairs.

Moving on next, we will split each word into characters and count their occurrence. The initial tokens will be all the characters and the “</w>” token.

Number	Token	Frequency
5	e	16
6	r	3
7	f	9
8	i	9
9	n	9
10	s	13
11	t	13
12	w	4

Since we have 23 words in total, so we have 23 “</w>” tokens. The second highest frequency token is “e”. In total, we have 12 different tokens.

The next step in the BPE algorithm is to look for the most frequent pairing, merge them, and perform the same iteration again and again until we reach our token limit or iteration limit.

Merging lets you represent the corpus with the least number of tokens which is the main goal of the BPE algorithm, that is, compression of data. To merge, BPE looks for the most frequently represented byte pairs. Here, we are considering a character to be the same as a byte. This is a case in the English language and can vary in other languages. Now we will merge the most common byte pairs to make one token and add them to the list of tokens and recalculate the frequency of occurrence of each token. This means our frequency count will change after each merging step. We will keep on doing this merging step until we hit the number of iterations or reach the token limit size.

Iterations

Iteration 1: We will start with the second most common token which is “e”. The most common byte pair in our corpus with “e” is “e” and “s” (in the words finest and lowest) which occurred 9 + 4 = 13 times. We merge them to form a new token “es” and note down its frequency as 13. We will also reduce the count 13 from the individual tokens (“e” and “s”). This will let us know about the leftover “e” or “s” tokens. We can see that “s” doesn’t occur alone at all and “e” occurs 3 times. Here is the updated table:

Number	Token	Frequency
1	</w>	23
2	o	14
3	l	14
4	d	10
5	e	16 - 13 = 3
6	r	3
7	f	9
8	i	9
9	n	9
10	s	13 - 13 = 0
11	t	13
12	w	4
13	es	9 + 4 = 13

Iteration 2: We will now merge the tokens “es” and “t” as they have appeared 13 times in our corpus. So, we have a new token “est” with frequency 13 and we will reduce the frequency of “es” and “t” by 13.

Number	Token	Frequency
1	</w>	23
2	o	14
3	l	14
4	d	10
5	e	16 - 13 = 3
6	r	3
7	f	9
8	i	9
9	n	9
10	s	13 - 13 = 0
11	t	13 - 13 = 0
12	w	4
13	es	9 + 4 = 13 - 13 = 0
14	est	13

Iteration 3: Let's work now with the "</w>" token. We see that byte pair "est" and "</w>" occurred 13 times in our corpus.

Number	Token	Frequency
1	</w>	23 - 13 = 10
2	o	14
3	l	14
4	d	10
5	e	16 - 13 = 3
6	r	3
7	f	9
8	i	9
9	n	9
10	s	13 - 13 = 0
11	t	13 - 13 = 0
12	w	4
13	es	9 + 4 = 13 - 13 = 0
14	est	13 - 13 = 0
15	est</w>	13

Note: Merging stop token “</w>” is very important. This helps the algorithm understand the difference between the words like “estimate” and “highest”. Both these words have “est” in common but one has an “est” token in the end and one at the start. Thus tokens like “est” and “est</w>” would be handled differently. If the algorithm will see the token “est</w>” it will know that it is the token for the word “highest” and not for the word “estate”.

Iteration 4: Looking at the other tokens, we see that byte pairs “o” and “l” occurred 7 + 3 = 10 times in our corpus.

Number	Token	Frequency
1	</w>	23
2	o	14 - 10 = 4
3	l	14 - 10 = 4
4	d	10
5	e	16 - 13 = 3
6	r	3
7	f	9
8	i	9
9	n	9
10	s	13 - 13 = 0
11	t	13 - 13 = 0
12	w	4
13	es	9 + 4 = 13 - 13 = 0
14	est	13
15	ol	7 + 3 = 10

Iteration 5: We now see that byte pairs “ol” and “d” occurred 10 times in our corpus.

Number	Token	Frequency
1	</w>	23 - 13 = 10
2	o	14 - 10 = 4
3	l	14 - 10 = 4
4	d	10 - 10 = 0
5	e	16 - 13 = 3
6	r	3
7	f	9
8	i	9
9	n	9
10	s	13 - 13 = 0
11	t	13 - 13 = 0
12	w	4
13	es	9 + 4 = 13 - 13 = 0
14	est	13 - 13 = 0
15	est</w>	13
16	ol	7 + 3 = 10 - 10 = 0
17	old	7 + 3 = 10

If we now look at our table, we see that the frequency of “f”, “i”, and “n” is 9 but we have just one word with these characters, so we are not merging them. For the sake of the simplicity of this article, let us now stop our iterations and closely look at our tokens.

Number	Token	Frequency
1	</w>	10
2	o	4
3	l	4
4	e	3
5	r	3
6	f	9
7	i	9
8	n	9
9	w	4
10	est</w>	13
11	old	10

The tokens with 0 frequency count have been removed from the table. We can now see that the total token count is 11, which is less than our initial count of 12. This is a small corpus but in practice, the size reduces a lot. This list of 11 tokens will serve as our vocabulary.

You must have also noticed that when we add a token, either our count increases or decreases or remains the same. In practice, the token count first increases and then decreases. The stopping criteria can be either the count of the tokens or the number of iterations. We choose this stopping criterion such that our dataset can be broken down into tokens in the most efficient way.

Encoding and Decoding

Let us now see how we will decode our example. To decode, we have to simply concatenate all the tokens together to get the whole word. For example, the encoded sequence [“the</w>”, “high”, “est</w>”, “range</w>”, “in</w>”, “Seattle</w>”], we will be decoded as [“the”, “highest”, “range”, “in”, “Seattle”] and not as [“the”, “high”, “estrange”, “in”, “Seattle”]. Notice the presence of the “</w>” token in “est”.

For encoding the new data, the process is again simple. However, encoding in itself is computationally expensive. Suppose the sequence of words is [“the</w>”, “highest</w>”, “range</w>”, “in</w>”, “Seattle</w>”]. We will iterate through all the tokens we found in our corpus — longest to the shortest and try to replace substrings in our given sequence of words using these tokens. Eventually, we will iterate through all the tokens and our substrings will be replaced with tokens already present in our token list. If a few substrings will be left (for words our model did not see in training), we will replace them with unknown tokens.

In general, the vocabulary size is big but still, there is a possibility of an unknown word. In practice, we save the pre-tokenized words in a dictionary. For unknown (new) words, we apply the above-stated encoding method to tokenize the new word and add the tokenization of the new word to our dictionary for future reference. This helps us build our vocabulary even stronger for the future.

Isn’t it greedy? 🤔

In order to represent the corpus in the most efficient way, BPE goes through every potential option to merge at each iteration by looking at its frequency. So, yes it follows a greedy approach to optimize for the best possible solution.

Anyways, BPE is one of the most widely used subword-tokenization algorithms and it has a good performance despite being greedy. 💃

I hope this article helped you understand the idea and logic behind the BPE algorithm. 😍

References:

1. https://aclanthology.org/P16-1162.pdf
2. https://huggingface.co/transformers/tokenizer_summary.html

3. https://www.drdobbs.com/a-new-algorithm-for-data-compression/184402829
4. https://en.wikipedia.org/wiki/Byte_pair_encoding

Thank you, everyone, for reading this article. Do share your valuable feedback or suggestion. Happy reading! 📗 🖌

NLP | Data Science | Deep Learning | Artificial Intelligence | Machine Learning

Written by Chetna Khanna

Follow

690 Followers · Writer for Towards Data Science

Engineer — Data & ML | Love to read | Love to write | https://www.linkedin.com/in/chetna-khanna/

More from Chetna Khanna and Towards Data Science

Chetna Khanna in Towards Data Science

What and why behind fit_transform() vs transform() in...

Scikit-learn is the most useful library for machine learning in Python programming...

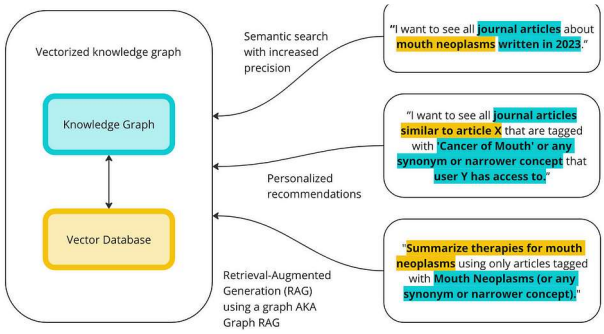
Aug 25, 2020 2.2K 55

Ahmed Besbes in Towards Data Science

What Nobody Tells You About RAGs

A deep dive into why RAG doesn't always work as expected: an overview of the...

Aug 23 1.6K 24



Steve Hedden in Towards Data Science

How to Implement Graph RAG Using Knowledge Graphs and...

A Step-by-Step Tutorial on Implementing Retrieval-Augmented Generation (RAG),...

Sep 6 1.1K 13

Chetna Khanna in Towards Data Science

Observational vs Experimental Study

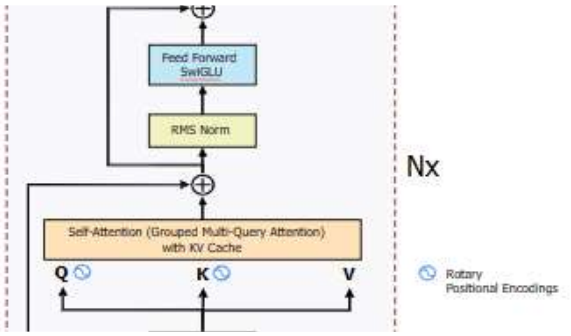
Is your statistical study observational or experimental? Let us find out.

Dec 5, 2020 94 5

See all from Chetna Khanna

See all from Towards Data Science

Recommended from Medium

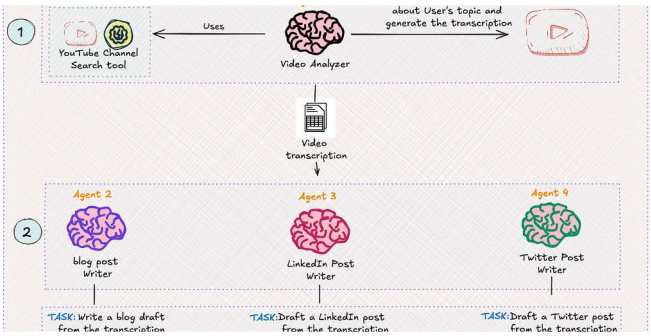


vignesh yaadav

Exploring and building the LLaMA 3 Architecture : A Deep Dive into...

Meta is stepping up its game in the artificial intelligence (AI) race with the introduction of...

Apr 19 · 274 · 5



Zoumana Keita in Towards Data Science

AI Agents—From Concepts to Practical Implementation in Python

This will change the way you think about AI and its capabilities

Aug 12 · 1.3K · 19

Lists

Predictive Modeling w/ Python

20 stories · 1548 saves

Practical Guides to Machine Learning

10 stories · 1877 saves

Natural Language Processing

1715 stories · 1287 saves

data science and AI

40 stories · 247 saves

Malik kashif

New York Street

Sep 15 808 24

Tayyib Ul Hassan Gondal in The Deep Hub

All you need to know about Tokenization in LLMs

In this blog, I'll explain everything about tokenization, which is an important step...

Jul 4 103

Vipra Singh

LLM Architectures Explained: NLP Fundamentals (Part 1)

Deep Dive into the architecture & building of real-world applications leveraging NLP...

Aug 15 1.5K 10

Abhishek Kumar Pandey

Word Piece Tokenization Algorithm

Easy:

Mar 30 3

See more recommendations