What is Tokenization in NLP? Here's All You Need To Know

ADVANCED ALGORITHM NLP PYTHON TEXT UNSTRUCTURED DATA

Highlights

- Tokenization is a key (and mandatory) aspect of working with text data
- We'll discuss the various nuances of tokenization, including how to handle Out-of-Vocabulary words (OOV)

Introduction

Language is a thing of beauty. But mastering a new language from scratch is quite a daunting prospect. If you've ever picked up a language that wasn't your mother tongue, you'll relate to this! There are so many layers to peel off and syntaxes to consider – it's quite a challenge.

And that's exactly the way with our machines. In order to get our computer to understand any text, we need to break that word down in a way that our machine can understand. That's where the concept of tokenization in Natural Language Processing (NLP) comes in.

Simply put, we can't work with text data if we don't perform tokenization. Yes, it's really that important!

And here's the intriguing thing about tokenization – it's not *just* about breaking down the text. Tokenization plays a significant role in dealing with text data. So in this article, we will explore the depths of

tokenization in Natural Language Processing and how you can implement it in Python.

I recommend taking some time to go through the below resource if you're new to NLP:

- Introduction to Natural Language Processing (NLP)

Table of Contents

A Quick Rundown of Tokenization

Tokenization is a common task in Natural Language Processing (NLP). It's a fundamental step in both traditional NLP methods like Count Vectorizer and Advanced Deep Learning-based architectures like Transformers.

> *Tokens are the building blocks of Natural Language.*

Tokenization is a way of separating a piece of text into smaller units called tokens. Here, tokens can be either words, characters, or subwords. Hence, tokenization can be broadly classified into 3 types – word, character, and subword (n-gram characters) tokenization.

For example, consider the sentence: "Never give up".

The most common way of forming tokens is based on space. Assuming space as a delimiter, the tokenization of the sentence results in 3 tokens – Never-give-up. As each token is a word, it becomes an example of Word tokenization.

Similarly, tokens can be either characters or subwords. For example, let us consider "smarter":

1. Character tokens: s-m-a-r-t-e-r
2. Subword tokens: smart-er

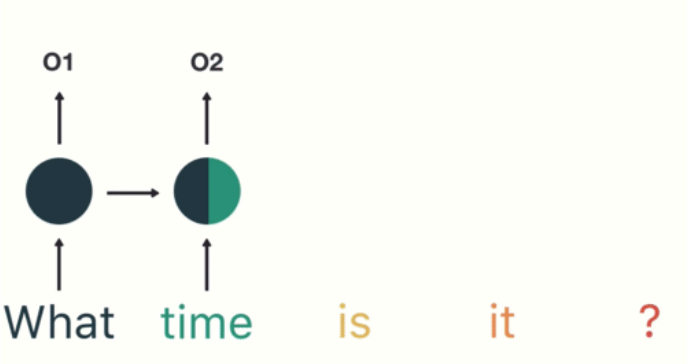
But then is this necessary? Do we really need tokenization to do all of this?

The True Reasons behind Tokenization

As tokens are the building blocks of Natural Language, the most common way of processing the raw text happens at the token level.

For example, Transformer based models – the State of The Art (SOTA) Deep Learning architectures in NLP – process the raw text at the token level. Similarly, the most popular deep learning architectures for NLP

like RNN, GRU, and LSTM also process the raw text at the token level.



Working of Recurrent Neural Network

As shown here, RNN receives and processes each token at a particular timestep.

Hence, Tokenization is the foremost step while modeling text data. Tokenization is performed on the corpus to obtain tokens. The following tokens are then used to prepare a vocabulary. Vocabulary refers to the set of unique tokens in the corpus. Remember that vocabulary can be constructed by considering each unique token in the corpus or by considering the top K Frequently Occurring Words.

> *Creating Vocabulary is the ultimate goal of Tokenization.*

One of the simplest hacks to boost the performance of the NLP model is to create a vocabulary out of top K frequently occurring words.

Now, let's understand the usage of the vocabulary in Traditional and Advanced Deep Learning-based NLP methods.

- Traditional NLP approaches such as Count Vectorizer and TF-IDF use vocabulary as features. Each word in the vocabulary is treated as a unique feature:

	I	play	cricket	football	tennis
Doc 1	1	1	1	1	1
Doc 2	1	1	0	1	0
Doc 3	0	1	1	0	0
Doc 4	1	1	0	0	1

Traditional NLP: Count Vectorizer

- In Advanced Deep Learning-based NLP architectures, vocabulary is used to create the tokenized input sentences. Finally, the tokens of these sentences are passed as inputs to the model

Which Tokenization Should you use?

As discussed earlier, tokenization can be performed on word, character, or subword level. It's a common question – which Tokenization should we use while solving an NLP task? Let's address this question here.

Word Tokenization

Word Tokenization is the most commonly used tokenization algorithm. It splits a piece of text into individual words based on a certain delimiter. Depending upon delimiters, different word-level tokens are formed. Pretrained Word Embeddings such as Word2Vec and GloVe comes under word tokenization.

But, there are few drawbacks to this.

Drawbacks of Word Tokenization

One of the major issues with word tokens is dealing with **Out Of Vocabulary (OOV) words**. OOV words refer to the new words which are encountered at testing. These new words do not exist in the vocabulary. Hence, these methods fail in handling OOV words.

But wait – don't jump to any conclusions yet!

- A small trick can rescue word tokenizers from OOV words. The trick is to form the vocabulary with the Top K Frequent Words and replace the rare words in training data with **unknown tokens (UNK)**. This helps the model to learn the representation of OOV words in terms of UNK tokens
- So, during test time, any word that is not present in the vocabulary will be mapped to a UNK token. This is how we can tackle the problem of OOV in word tokenizers.
- The problem with this approach is that the entire information of the word is lost as we are mapping OOV to UNK tokens. The structure of the word might be helpful in representing the word accurately. And another issue is that every OOV word gets the same representation

Another issue with word tokens is connected to the size of the vocabulary. Generally, pre-trained models are trained on a large volume of the text corpus. So, just imagine building the vocabulary with all the unique words in such a large corpus. This explodes the vocabulary!

This opens the door to Character Tokenization.

Character Tokenization

Character Tokenization splits apiece of text into a set of characters. It overcomes the drawbacks we saw above about Word Tokenization.

- Character Tokenizers handles OOV words coherently by preserving the information of the word. It breaks down the OOV word into characters and represents the word in terms of these characters
- It also limits the size of the vocabulary. Want to talk a guess on the size of the vocabulary? 26 since the vocabulary contains a unique set of characters

Drawbacks of Character Tokenization

Character tokens solve the OOV problem but the length of the input and output sentences increases rapidly as we are representing a sentence as a sequence of characters. As a result, it becomes challenging to learn the relationship between the characters to form meaningful words.

This brings us to another tokenization known as Subword Tokenization which is in between a Word and Character tokenization.

Subword Tokenization

Subword Tokenization splits the piece of 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.

Transformed based models – the SOTA in NLP – rely on Subword Tokenization algorithms for preparing vocabulary. Now, I will discuss one of the most popular Subword Tokenization algorithm known as Byte Pair Encoding (BPE).

Welcome to Byte Pair Encoding (BPE)

Byte Pair Encoding (BPE) is a widely used tokenization method among transformer-based models. BPE addresses the issues of Word and Character Tokenizers:

- BPE tackles OOV effectively. It segments OOV as subwords and represents the word in terms of these subwords
- The length of input and output sentences after BPE are shorter compared to character tokenization

BPE is a word segmentation algorithm that merges the most frequently occurring character or character sequences iteratively. Here is a step by step guide to learn BPE.

Steps to learn BPE

1. Split the words in the corpus into characters after appending </w>
2. Initialize the vocabulary with unique characters in the corpus
3. Compute the frequency of a pair of characters or character sequences in corpus

4. Merge the most frequent pair in corpus
5. Save the best pair to the vocabulary
6. Repeat steps 3 to 5 for a certain number of iterations

We will understand the steps with an example.

Consider a corpus:

Corpus

low	lower	newest
low	lower	newest
low	widest	newest
low	widest	newest
low	widest	newest

1a) Append the end of the word (say </w>) symbol to every word in the corpus:

Corpus

low</w>	lower</w>	newest</w>
low</w>	lower</w>	newest</w>
low</w>	widest</w>	newest</w>
low</w>	widest</w>	newest</w>
low</w>	widest</w>	newest</w>

1b) Tokenize words in a corpus into characters:

Corpus

l o w </w>	l o w e r </w>	n e w e s t </w>
l o w </w>	l o w e r </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>

2. Initialize the vocabulary:

Vocabulary

d	e	i	l	n	o	s	t	w

Iteration 1:

3. Compute frequency:

Frequency

d-e (3)	l-o (7)	t-</w> (8)
e-r (2)	n-e (5)	w-</w> (5)
e-s (8)	o-w (7)	w-e (7)
e-w (5)	r-</w> (2)	w-i (3)
i-d (3)	s-t (8)	

4. Merge the most frequent pair:

Corpus

l o w </w>	l o w e r </w>	n e w es t </w>
l o w </w>	l o w e r </w>	n e w es t </w>
l o w </w>	w i d es t </w>	n e w es t </w>
l o w </w>	w i d es t</w>	n e w es t </w>
l o w </w>	w i d es t </w>	n e w es t </w>

5. Save the best pair:

Vocabulary

d	e	i	l	n	o	s	t	w
es								

Repeat steps 3-5 for every iteration from now. Let me illustrate for one more iteration.

Iteration 2:

3. Compute frequency:

Frequency

d-es (3)	l-o (7)	w-</w> (5)
e-r (2)	n-e (5)	w-es (5)
e-w (5)	o-w (7)	w-e (2)
es-t (8)	r-</w> (2)	w-i (3)
i-d (3)	t-</w> (8)	

4. Merge the most frequent pair:

Corpus

l o w </w>	l o w e r </w>	n e w est </w>
l o w </w>	l o w e r </w>	n e w est </w>
l o w </w>	w i d est </w>	n e w est </w>
l o w </w>	w i d est </w>	n e w est </w>
l o w </w>	w i d est </w>	n e w est </w>

5. Save the best pair:

Vocabulary

d	e	i	l	n	o	s	t	w
es	est							

After 10 iterations, BPE merge operations looks like:

Vocabulary

d	e	i	l	n	o	s	t	w
es	est	est</w>	lo	low	low</w>	ne	new	newest</w>

Pretty straightforward, right?

Applying BPE to OOV words

But, how can we represent the OOV word at test time using BPE learned operations? Any ideas? Let's answer this question now.

> At test time, the OOV word is split into sequences of characters. Then the learned operations are applied to merge the characters into larger known symbols.
>
> – Neural Machine Translation of Rare Words with Subword Units, 2016

Here is a step by step procedure for representing OOV words:

1. Split the OOV word into characters after appending </w>
2. Compute 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

Let's see all this in action next!

Implementing Tokenization – Byte Pair Encoding in Python

We are now aware of how BPE works – learning and applying to the OOV words. So, its time to implement our knowledge in Python.

The python code for BPE is already available in the original paper itself (Neural Machine Translation of Rare Words with Subword Units, 2016)

Reading Corpus

We'll consider a simple corpus to illustrate the idea of BPE. Nevertheless, the same idea applies to another corpus as well:

```
#importing library
import pandas as pd

#reading .txt file
text = pd.read_csv("sample.txt",header=None)

#converting a dataframe into a single list
corpus=[]
for row in text.values:
    tokens = row[0].split(" ")
    for token in tokens:
        corpus.append(token)
```

view raw

bpe_1.py hosted with ❤ by GitHub

Text Preparation

Tokenize the words into characters in the corpus and append </w> at the end of every word:

```
#initlialize the vocabulary
vocab = list(set(" ".join(corpus)))
vocab.remove(' ')

#split the word into characters
corpus = [" ".join(token) for token in corpus]

#appending </w>
corpus=[token+' </w>' for token in corpus]
```

view raw

bpe_2.py hosted with ❤ by GitHub

Learning BPE

Compute the frequency of each word in the corpus:

```
import collections

#returns frequency of each word
corpus = collections.Counter(corpus)

#convert counter object to dictionary
corpus = dict(corpus)
print("Corpus:",corpus)
```

view raw

bpe_3.py hosted with ❤ by GitHub

Output:

```
Corpus: {'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w e s t </w>': 6, 'w i d e s t </w>': 3}
```

Let's define a function to compute the frequency of a pair of character or character sequences. It accepts the corpus and returns the pair with its frequency:

```
#computer frequency of a pair of characters or character sequences
#accepts corpus and return frequency of each pair
def get_stats(corpus):
    pairs = collections.defaultdict(int)
    for word, freq in corpus.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
            pairs[symbols[i],symbols[i+1]] += freq
    return pairs
```

view raw

bpe_4.py hosted with ❤ by GitHub

Now, the next task is to merge the most frequent pair in the corpus. We will define a function that accepts the corpus, best pair, and returns the modified corpus:

```
#merges the most frequent pair in the corpus
#accepts the corpus and best pair
#returns the modified corpus
import re
def merge_vocab(pair, corpus_in):
    corpus_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')

    for word in corpus_in:
        w_out = p.sub(''.join(pair), word)
        corpus_out[w_out] = corpus_in[word]
```

```
13
14      return corpus_out
```

view raw

bpe_5.py hosted with ❤ by GitHub

Next, its time to learn BPE operations. As BPE is an iterative procedure, we will carry out and understand the steps for one iteration. Let's compute the frequency of bigrams:

```
#compute frequency of bigrams in a corpus
pairs = get_stats(corpus)
print(pairs)
```

view raw

bpe_6.py hosted with ❤ by GitHub

Output:

```
{('d', 'e'): 3,
 ('e', 'r'): 2,
 ('e', 's'): 8,
 ('e', 'w'): 5,
 ('i', 'd'): 3,
 ('l', 'o'): 7,
 ('n', 'e'): 5,
 ('o', 'w'): 7,
 ('r', '</w>'): 2,
 ('s', 't'): 8,
 ('t', '</w>'): 8,
 ('w', '</w>'): 5,
 ('w', 'e'): 7,
 ('w', 'i'): 3})
```

Find the most frequent pair:

```
#compute the best pair
best = max(pairs, key=pairs.get)
print("Most Frequent pair:",best)
```

view raw

bpe_7.py hosted with ❤ by GitHub

Output: ('e', 's')

Finally, merge the best pair and save to the vocabulary:

```
#merge the frequent pair in corpus
corpus = merge_vocab(best, corpus)
print("After Merging:", corpus)

#convert a tuple to a string
best = "".join(list(best))

#append to merge list and vocabulary
merges = []
merges.append(best)
vocab.append(best)
```

view raw

bpe_8.py hosted with ❤ by GitHub

Output:

```
After Merging: {'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w es t </w>': 5, 'w i d es t </w>': 3}
```

We will follow similar steps for certain iterations:

```
num_merges = 10
for i in range(num_merges):

```

```
    #compute frequency of bigrams in a corpus
    pairs = get_stats(corpus)

    #compute the best pair
    best = max(pairs, key=pairs.get)

    #merge the frequent pair in corpus
    corpus = merge_vocab(best, corpus)

    #append to merge list and vocabulary
    merges.append(best)
    vocab.append(best)

#convert a tuple to a string
merges_in_string = ["".join(list(i)) for i in merges]
print("BPE Merge Operations:",merges_in_string)
```

view raw

bpe_9.py hosted with ❤ by GitHub

Output:

```
BPE Merge Operations: ['es', 'est', 'est</w>', 'lo', 'low', 'ne', 'new', 'newest</w>', 'low</w>', 'wi', 'wid']
```

The most interesting part is yet to come! That's applying BPE to OOV words.

Applying BPE to OOV word

Now, we will see how to segment the OOV word into subwords using learned operations. Consider OOV word to be "lowest":

```
#applying BPE to OOV
oov ='lowest'

#tokenize OOV into characters
oov = " ".join(list(oov))

#append </w>
oov = oov + ' </w>'

#create a dictionary
oov = { oov : 1}
```

view raw

bpe_10.py hosted with ❤ by GitHub

Applying BPE to an OOV word is also an iterative process. We will implement the steps discussed earlier in the article:

```
i=0
while(True):

    #compute frequency
    pairs = get_stats(oov)

    #extract keys
    pairs = pairs.keys()

    #find the pairs available in the learned operations
    ind=[merges.index(i) for i in pairs if i in merges]

    if(len(ind)==0):
        print("\nBPE Completed...")
        break

    #choose the most frequent learned operation
    best = merges[min(ind)]

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

```
22
23      print("Iteration ",i+1, list(oov.keys())[0])
24      i=i+1
```

view raw

bpe_11.py hosted with ❤ by GitHub

Output:

```
Iteration  1 : l o w es t </w>
Iteration  2 : l o w est </w>
Iteration  3 : l o w est</w>
Iteration  4 : lo w est</w>
Iteration  5 : low est</w>

BPE Completed...
```

As you can see here, the unknown word “lowest” is segmented as low-est.

End Notes

Tokenization is a powerful way of dealing with text data. We saw a glimpse of that in this article and also implemented tokenization using Python.

Go ahead and try this out on any text-based dataset you have. The more you practice, the better your understanding of how tokenization works (and why it’s such a critical NLP concept). Feel free to reach out to me in the comments below if you have any queries or thoughts on this article.

Article Url - https://www.analyticsvidhya.com/blog/2020/05/what-is-tokenization-nlp/

Aravind Pai

Aravind is a sports fanatic. His passion lies in developing data-driven products for the sports domain. He strongly believes that analytics in sports can be a game-changer