

# LLM Lab 1 Report

**Name:** Shusrith S

**SRN:** PES1UG22AM155

---

## 1. Data Source

The data source used for this lab is the book *Moby Dick; Or the Whale* by Herman Melville. This text will be processed to perform encoding using Byte Pair Encoding (BPE) and Word Piece Encoding (WPE).

---

## 2. Preprocessing

The text file is read, and preprocessing is performed to convert the text into a format suitable for encoding. This involves converting the text to lowercase to avoid clashes between upper and lower case letters. Regular expressions are used to select only alphabetic characters and whitespaces for processing.

The corpus contains nearly **200k words**, which will be processed to build the encoding.

The text is converted to a list of characters, excluding whitespaces. Initially, there are **nearly 1 million characters** in the text.

A dictionary of words is created along with their frequencies. This dictionary represents the frequency distribution of words, which will be used for encoding.

---

## 3. Byte Pair Encoding (BPE)

### First Loop

Byte Pair Encoding (BPE) starts with finding the most frequent pairs of adjacent characters in the corpus. The **Counter** library is used to count the frequency of pairs of characters using a sliding window technique.

```

from collections import Counter
pairs = [(corpus[i], corpus[i + 1]) for i in range(len(corpus) - 1)]
pair_dict = Counter(pairs)
pair_dict

```

Once the pairs are created, the counter aggregates all pairs and gives the frequency of occurrence of each pair. For example, **(t, h)** appears **33,400 times** and is selected to be merged into a new token **th**. This new token is added to the vocabulary.

```

Counter({'t', 'h'): 33408,
        ('h', 'e'): 26991,
        ('e', 'p'): 2748,
        ('p', 'r'): 1941,
        ('r', 'o'): 5767,
        ('o', 'j'): 154,
        ('j', 'e'): 333,
        ('e', 'c'): 4345,
        ('c', 't'): 1416,
        ('t', 'g'): 395,
        ('g', 'u'): 714,
        ('u', 't'): 4395,
        ('t', 'e'): 8032,
        ('e', 'n'): 10039,
        ('n', 'b'): 879,
        ('b', 'e'): 4224,
        ('e', 'r'): 16298,

```

After merging the most frequent pair, the corpus is updated, and all occurrences of **t** followed by **h** are replaced by **th**. The size of the corpus reduces from 1 million characters to approximately **930k** characters.

## Second Loop

The same process is repeated, creating pairs using the sliding window technique and counting their frequency. The most common pair **(th, e)** is then merged into the token **the**, and the vocabulary is updated.

```
Counter({'th', 'e'): 20753,  
        ('e', 'p'): 2748,  
        ('p', 'r'): 1941,  
        ('r', 'o'): 5767,  
        ('o', 'j'): 154,  
        ('j', 'e'): 333,  
        ('e', 'c'): 4345,  
        ('c', 't'): 1381,  
        ('t', 'g'): 395,  
        ('g', 'u'): 714,  
        ('u', 't'): 3915,  
        ('t', 'e'): 8032,  
        ('e', 'n'): 10039,  
        ('n', 'b'): 879,  
        ('b', 'e'): 4224,  
        ('e', 'r'): 16298,  
        ('r', 'g'): 692,
```

---

## 4. Consolidating into a Function

The Byte Pair Encoding (BPE) algorithm can be consolidated into a function that accepts preprocessed text and converts it into a list of characters, creating a **corpus** that excludes whitespaces. The vocabulary initially consists of individual characters but grows as the algorithm merges frequent pairs. The function performs this by iterating through the corpus, generating pairs using a sliding window, and counting their frequencies with the **Counter** function. The most frequent pair is merged into a new token, and the corpus is updated by replacing occurrences of this pair. If the new token is not already in the vocabulary, it is added.

The algorithm continues this process for a set number of iterations, in this case, 100. Ideally, merging should continue as long as at least one token from the vocabulary exists in the corpus. If no such token is found, the loop terminates early, indicating that further merges are no longer meaningful. The algorithm allows for efficient tokenization by progressively merging frequent character pairs, ultimately creating a vocabulary of subword units that can be used for text encoding.

This function is run for 100 iterations. At the end of the process, the vocabulary contains **126 tokens**, indicating the number of iterations where new tokens were created.

```

from collections import Counter

def BPE(text):
    c = list(text)
    corpus = []
    for i in c:
        if i != " ":
            corpus.append(i)
    vocabulary = list(set(corpus))
    j = 0
    while j < 100:
        pairs = [(corpus[i], corpus[i + 1]) for i in range(len(corpus)
- 1)]
        pair_dict = Counter(pairs)

        if not pair_dict:
            break

        most_common1, most_common2 = pair_dict.most_common(1)[0][0]
        new_token = most_common1 + most_common2
        merged_corpus = []
        i = 0
        while i < len(corpus):
            if i < len(corpus) - 1 and corpus[i] == most_common1 and
corpus[i + 1] == most_common2:
                merged_corpus.append(new_token)
                i += 2
            else:
                merged_corpus.append(corpus[i])
                i += 1

        corpus = merged_corpus
        if new_token not in vocabulary:
            vocabulary.append(new_token)

        if not any(token in corpus for token in vocabulary):
            break
        j += 1
    return vocabulary

```

## Vocabulary - Some screenshots

'to',	'ac',	'av',
'wh',	'he',	'ur',
'it',	'all',	'was',
're',	'id',	'this',
'st',	'his',	'li',
'el',	'ver',	'ght',
'om',	'ic',	'od',
'ch',	'ith',	'ess',
'ow',	'ir',	'qu',
'il',	'ab',	'up',
'se',	'ap',	'ion',
'ad',	'am',	'sp',
'le',	'ent',	'fr',
'ly',	'et',	'est',
'im',	'ay',	'by',
'gh',	'em',	'inthe',
'that',	'ol',	'oun',
'be',	'ofthe	'ro',
'for',	'sh',	'ke',
'bu',	'ot',	'ul',

## Encoding New Text - Testing

The algorithm is tested with the unseen text **“suzuki honda maruti”**. The word is encoded using the generated BPE tokens, and we can observe how the word is broken into subwords efficiently.

```
def encode_word(word, bpe_tokens):
    encoded_word = list(word)
    encoded_word = [i for i in encoded_word if i != ' ']
    for token in bpe_tokens:
        i = 0

        while i < len(encoded_word) - 1:
            if encoded_word[i] + encoded_word[i + 1] == token:
                encoded_word[i] = token
                del encoded_word[i + 1]
            else:
                i += 1
    return encoded_word
```

```
['s', 'u', 'z', 'u', 'k', 'i', 'h', 'on', 'd', 'am', 'ar', 'u', 't', 'i']
```

---

## 5. Word Piece Encoding (WPE)

### First Loop

The original text is reused, and the preprocessing steps are repeated. This time, the **freq\_dict** dictionary is created, which contains the frequency of characters in the corpus. Similar to BPE, pairs of adjacent characters are created using a sliding window, and their frequencies are calculated using the **Counter** library.

Next, we calculate the **WPE score** for each pair using the formula:

$$\frac{\text{freq}(\text{character1}, \text{character2})}{\text{freq}(\text{character1}) \times \text{freq}(\text{character2})}$$

The pair with the highest score is selected, and the corpus is updated by merging that pair into a new token. In the first loop, (**z, z**) is the pair with the highest score, and it is merged into **zz**.

```
(( 'z', 'z' ), [0.0001421487603305785, 43])
```

## Second Loop

The process is repeated in the second loop. Pairs are generated, frequencies are calculated, and the pair with the highest WPE score is merged into a new token. The new token is added to the vocabulary and the corpus is updated.

```
(( 'q', 'u'), [3.6734699096896025e-05, 1580])
```

---

## 6. Consolidating Word Piece Encoding

The Word Piece Encoding (WPE) algorithm is consolidated into a function that utilizes the **wpe\_score** function to calculate the score for each character pair based on their frequencies. The pair with the highest score is selected for merging. The WPE function processes a given text corpus by first converting it into a list of characters, removing whitespaces, and building a frequency dictionary (**freq\_dict**). The algorithm then runs for a set number of iterations, such as 100, but ideally continues until no more meaningful tokens can be added to the vocabulary.

During each iteration, pairs of adjacent characters are formed, and their frequencies are computed using the **Counter** function. The **wpe\_score** function is called to identify the pair with the highest score, which is then merged into a new token. The corpus is updated with this new token, and **freq\_dict** is adjusted accordingly. The algorithm continues to merge pairs and update the corpus until it reaches the maximum number of iterations or no more merges are possible, resulting in a vocabulary of subword units that efficiently represent the text corpus.

```
from collections import Counter, defaultdict

def wpe_score(pair_dict, freq_dict):
    wpe_dict = {}

    for pair, count in pair_dict.items():
        elem1, elem2 = pair
        elem1_count = freq_dict[elem1]
        elem2_count = freq_dict[elem2]
        wpe_dict[pair] = count / (elem1_count * elem2_count)

    if wpe_dict:
        max_pair = max(wpe_dict, key=wpe_dict.get)
        return max_pair, wpe_dict[max_pair], pair_dict[max_pair]
    else:
        return None, None, None
```

```

def WPE(text):
    corpus = [char for char in text if char != " "]
    freq_dict = Counter(corpus)
    vocabulary = list(freq_dict.keys())

    iterations = 0

    while iterations < 100:
        pairs = [(corpus[i], corpus[i + 1]) for i in range(len(corpus)
- 1)]
        pair_dict = Counter(pairs)

        max_pair, max_score, max_count = wpe_score(pair_dict,
freq_dict)
        if max_pair is None:
            break

        new_token = ''.join(max_pair)

        merged_corpus = []
        i = 0
        while i < len(corpus):
            if i < len(corpus) - 1 and (corpus[i], corpus[i + 1]) ==
max_pair:
                merged_corpus.append(new_token)
                i += 2
            else:
                merged_corpus.append(corpus[i])
                i += 1
        corpus = merged_corpus

        if new_token not in vocabulary:
            vocabulary.append(new_token)
            freq_dict[new_token] = max_count

        freq_dict[max_pair[0]] -= max_count
        freq_dict[max_pair[1]] -= max_count

        iterations += 1

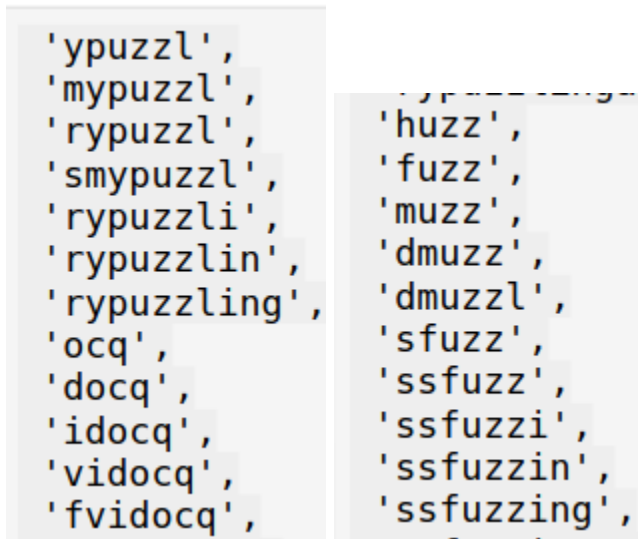
    return vocabulary

```

After running the algorithm for 100 iterations, the vocabulary contains **126 tokens**.



## Vocabulary - Some screenshots



```
'ypuzzl',  
'mypuzzl',  
'rypuzzl',  
'smpuzzl',  
'rypuzzli',  
'rypuzzlin',  
'rypuzzling',  
'ocq',  
'docq',  
'idocq',  
'vidocq',  
'fvidocq',  
'huzz',  
'fuzz',  
'muzz',  
'dmuzz',  
'dmuzzl',  
'sfuzz',  
'ssfuzz',  
'ssfuzzi',  
'ssfuzzin',  
'ssfuzzing',
```

## Encoding Unseen Words - Testing

The algorithm is tested with new words, and the encoding process is evaluated. The unseen word “**suzuki honda maruti**” is encoded using the Word Piece tokens.

```
def encode_word(word, wpe_tokens):  
    encoded_word = list(word)  
    encoded_word = [i for i in encoded_word if i != ' ']  
    for token in wpe_tokens:  
        i = 0  
        while i < len(encoded_word) - 1:  
            if encoded_word[i] + encoded_word[i + 1] == token:  
                encoded_word[i] = token  
                del encoded_word[i + 1]  
            else:  
                i += 1  
    return encoded_word  
encode_word("suzuki honda maruti", vocabulary_wpe)
```

```
['suz', 'u', 'k', 'i', 'hon', 'dam', 'ar', 'u', 't', 'i']
```

---

## 7. Comparison of Vocabularies

Upon comparing the vocabularies produced by **Byte Pair Encoding** and **Word Piece Encoding**, the following observations are made:

- **BPE Vocabulary:** The tokens are smaller in length and often represent high-frequency words like **the**, **is**, and **of**.
- **WPE Vocabulary:** The tokens tend to be larger and are more representative of the frequent subwords in the given corpus. Additionally, because WPE is probabilistic, it has a better tendency to handle **unseen words** more effectively.

WPE's approach of merging pairs based on their frequency and likelihood makes it more robust when dealing with rare or unseen words compared to BPE.

---

## 8. Conclusion

Both **Byte Pair Encoding** and **Word Piece Encoding** are powerful tokenization techniques for subword modeling, but each has its own strengths. BPE tends to produce smaller tokens, which are more useful for frequent, common words. On the other hand, WPE creates larger tokens, which are better suited for capturing the structure of rare or unseen words, making it more robust for generalization.

Thus, **Word Piece Encoding** is preferred in applications where handling unseen data or rare words is crucial, such as in large language models like BERT.