NLP 2025 Lab 1 Report: Tokenization

Nichita Bulgaru
Group 52
n.bulgaru@

Timur Jercaks
Group 52
t.jercaks@

Dmitrii Sakharov
Group 52
d.sakharov@

Abstract

This report details the experiments conducted for Lab 1: Tokenization in the NLP 2025 course. We explore the TweetEval dataset (emoji subset). Various text cleaning and tokenization techniques are implemented, including word-level tokenization with a fixed vocabulary and Byte Pair Encoding (BPE). We analyze the characteristics of the dataset and the vocabulary generated. Finally, we compare the different tokenization methods based on metrics like the number of unknown tokens, total tokens, and average token lengths, discussing their respective advantages and disadvantages.

1 Exercise 1: Questions about the datasets

1.1 What is the size of the training, test and validation datasets?

```
import pandas as pd
import datasets # Assuming tweet_ds is loaded

df_train = pd.DataFrame(tweet_ds['train'])
df_validation = pd.DataFrame(tweet_ds['validation'])
df_test = pd.DataFrame(tweet_ds['test'])

print("Training Dataset Size: ", df_train.shape[0])
print("Val Dataset Size: ", df_validation.shape[0])
print("Test Dataset Size: ", df_test.shape[0])
```

The resulting sizes are shown in Table 1.

Dataset Split	Size (Number of Examples)
Training	45000
Validation	5000
Test	50000

Table 1: Dataset Sizes

1.2 What are the top 5 most frequent emojis in the validation dataset?

The frequency of labels (emojis) in the validation set was calculated.

```
# df_validation is defined above
print(df_validation['label'].value_counts()[:5])
```

The top 5 most frequent emoji labels and their counts are listed in Table 2.

Label	Number of Occurrences
0	1056
1	521
2	504
3	308
4	243

Table 2: Top 5 Most Frequent Emoji Labels in Validation Set

1.3 Compare the distributions of labels (emojis) between training and validation datasets.

The distributions of emoji labels in the training and validation sets were compared by plotting their normalized frequencies.

```
train_counts = df_train['label'].value_counts(
                    normalize=True).sort_index()
val_counts = df_validation['label'].value_counts(
                    normalize=True).sort_index()

dist_df = pd.DataFrame({
    'Train': train_counts,
    'Validation': val_counts
})

dist_df.plot.bar(title='Label Distribution: Train vs Validation',
                figsize=(10, 6))
plt.show()
```

Figure 1 shows the comparison. The distributions appear very similar, indicating that the validation set is likely a representative sample of the training set in terms of label distribution.

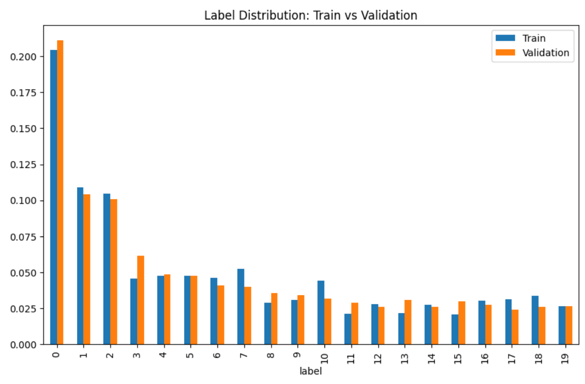

Figure 1: Comparison of Label Distributions (Normalized) between Training and Validation Sets.

1.4 How many examples with the "fire" emoji are in the training dataset?

The number of examples corresponding to the "fire" emoji (label 4) in the training set was counted.

```
print(df_train[df_train["label"]==4].shape[0])
```

There are 2146 examples with the "fire" emoji in the training dataset.

1.5 What is the average length (in characters) of the tweets in the training dataset?

```
print(df_train["text_length"].mean())
```

The average length of tweets in the training dataset is approximately 71.02 characters

2 Exercise 2: Write the text cleaning function

The following function ‘clean()‘ was changed in order to preprocess the tweet text. It performs the required steps: removing commas between digits, normalizing whitespace, spacing out punctuation, adding spaces between special characters, lemmatization, lowering case and removing whitespaces.

```
import re

def clean(example):
    text = example['text']
    if text == '':
        example['clean'] = ''
        return example
    text = str(text)

    # remove comma between numbers
    text = re.sub(r'(?<=\d),(?=\d)', '', text)

    # remove multiple spaces
    text = re.sub(r'[\u200d\u200c\u2764\ufe0f]+',
    ' ', text)
    pattern1 = r"\s{2,}"
    text = re.sub(pattern1, ' ', text)

    # space out the punctuation (dots and commas)
    pattern2 = r"\."
    pattern3 = r"\,"
    text = re.sub(pattern2, ' .', text)
    text = re.sub(pattern3, ' ,', text)

   # space between special characters (@, ?, !, ', ())
    pattern4 = r"\@"
    pattern5 = r"\!"
    pattern6 = r"\?"
    pattern7 = r"\'"
    pattern8 = r"\("
    pattern9 = r"\)"
    text = re.sub(pattern4, ' @ ', text)
    text = re.sub(pattern5, ' ! ', text)
    text = re.sub(pattern6, ' ? ', text)
    text = re.sub(pattern7, " ' ", text)
    text = re.sub(pattern8, " ( ", text)
    text = re.sub(pattern9, " ) ", text)

    # lemmatization
    pattern_be = r"\b(am|are|is)\b"
    pattern_have = r"\b(have|has|had)\b"
    pattern_do = r"\b(do|does|did|done)\b"
    pattern_go = r"\b(went|going|goes)\b"
   text = re.sub(pattern_be, "be", text, flags=re.IGNORECASE)
   text = re.sub(pattern_have, "have", text, flags=re.IGNORECASE)
   text = re.sub(pattern_do, "do", text, flags=re.IGNORECASE)
   text = re.sub(pattern_go, "go", text, flags=re.IGNORECASE)

    # lower case
    text = text.lower()

    # final whitespace removal
    pattern1 = r"\s{2,}"
    text = re.sub(pattern1, ' ', text)

    example['clean'] = text.strip()
    return example
```

The three additional steps chosen were:

1. Spacing out other common special characters (@, !, ?, ’, (,)).
2. Performing basic lemmatization for common irregular verbs (’be’, ’have’, ’do’, ’go’) to reduce vocabulary size slightly.
3. Converting the entire text to lowercase for consistency.

A final whitespace normalization step ensures no triple spaces are left after adding spaces around special characters.

```
print('Original tweet item:')
print(tweet_ds['train'][1:10]['text'])
print('Cleaned tweet item:')
print(clean(tweet_ds['train'][1:10])['clean'])
```

Truncuated tweets to just show that preprocessing worked.

Original tweet item:

"Time for some BBQ and whiskey libations. Chomp, belch, chomp! (@ Lucille’s Smokehouse Bar-B-Que)"...

Cleaned tweet item:

"time for some bbq and whiskey libations . chomp

, belch , chomp ! (@ lucille ' s smokehouse bar-b-que) "...

3 Exercise 3: Build the vocabulary

This function was implemented to count word frequencies in the cleaned training data. It uses a regular expression to find sequences that are either words (\w+), mentions (@\w+), or hashtags (#\w+).

```
from collections import Counter
import re

def build_vocab_counter(dataset):
    vocab = Counter()
    for n in dataset["clean"]:
      # Regex finds words, @mentions, and #hashtags
        tokens = re.findall(r'@\w+|#\w+|\w+', n)
        vocab.update(tokens)
    return vocab

vocab_counter=build_vocab_counter(tweet_ds['train'])
print('Size of the vocabulary:', len(vocab_counter))
# Output: Size of the vocabulary: 59247

print('Most common:')
print(vocab_counter.most_common(10))
# Output: Most common:
# [('the', 13875), ('user', 12236), ('i', 8489),
# ('to', 7841), ('my', 7648), ('a', 7022),
# ('be', 6332), ('in', 6103),
# ('and', 5799), ('you', 5755)]

print('Least common:')
print(vocab_counter.most_common()[-10:])
# Output: Least common:
# [('#southbayla', 1), ('thedabberchick', 1),
# ('nector', 1), ('chefking1921express', 1),
# ('#alabama', 1), ('#rolltide', 1),
# ('#bffweekend', 1), ('nunez', 1),
# ('#happylaborday', 1), ('five50', 1)]
```

The total vocabulary size based on the cleaned training data is 59,247 unique tokens (words, mentions, hashtags). The most common tokens are typical English stop words ('the', 'i', 'to', 'a', 'in', 'and', 'you') along with the placeholder '@user' and the lemmatized verb 'be'. This is expected in a large text corpus. The least common tokens appear to be rare hashtags, usernames, specific names, or potentially typos, each occurring only once. This long tail of rare words is also characteristic of natural language.

The distribution of word frequencies follows Zipf's law, as shown in Figure 2, where a few words are very frequent, and many words are very rare.

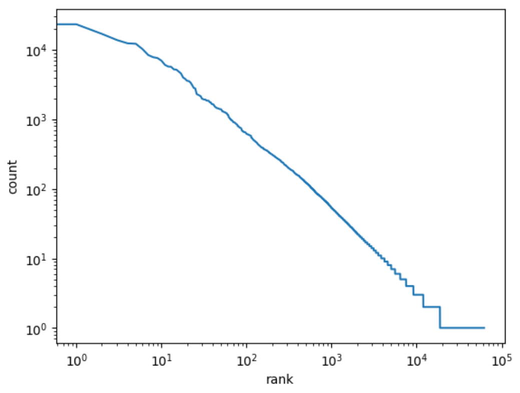

Figure 2: Log-log plot of word frequencies vs. rank, illustrating Zipf's Law.

4 Exercise 4: Frequency of pairs of words

The frequency of adjacent word pairs (bigrams) in the cleaned training data was calculated.

```
from collections import Counter
import re
import matplotlib.pyplot as plt

counter = Counter()
for n in tweet_ds["train"]:
    cleaned = n["clean"]
   tokens = re.findall(r'@\w+|#\w+|\w+', cleaned)
    pairs = [(tokens[i], tokens[i+1]) for i in
    range(len(tokens) - 1)]
    counter.update(pairs)

print(counter.most_common(10))
# Output:
# [(('los', 'angeles'), 1749), (('it', 's'), 1314),
# (('i', 'm'), 1153), (('with', 'my'), 1085),
# (('user', 'user'), 1044), (('in', 'the'), 1028),
# (('las', 'vegas'), 1006), (('for', 'the'), 884),
# (('i', 'love'), 864), (('thank', 'you'), 849)]

print(counter.most_common()[-10:])
# Output:
# [(('everyone', '#happylaborday'), 1),
# (('#happylaborday', 'beer'), 1),
# (('beer', 'nv'), 1), (('pizza', 'five50'), 1),
# (('five50', 'user'), 1), (('mini', 'be'), 1),
# (('perfect', 'no'), 1), (('one', 'deserves'), 1),
# (('deserves', 'her'), 1), (('her', 'las'), 1)]
# - How many pairs occur only once in the dataset?
least_common = [n for n in counter.most_common()
if n[1] == 1]
print(len(least_common))
# Output: 196099

plt.figure(figsize=(10, 6))
plt.hist(list(counter.values()), bins=50, log=True)
plt.title('Distribution of Word Pair Frequencies')
plt.xlabel('Frequency')
plt.ylabel('Number of Pairs (log scale)')
plt.grid(True)
plt.show() # In notebook context
# plt.savefig('bigram_freq.png') # To save figure
```

The most common bigrams, such as '('los', 'angeles')', '('it', 's')', '('i', 'm')', '('in', 'the')', '('thank', 'you')', represent common phrases, contractions, or collocations involving frequent words. This makes sense linguistically. The least common bigrams involve rare words or specific context-dependent sequences, each occurring only once.

There are 196,099 unique bigrams that occur only once in the training dataset. This highlights the sparsity of language data; many word combinations are unique or extremely rare.

The distribution of bigram frequencies, shown in Figure 3, is highly skewed, similar to the unigram (single word) distribution, following a power law. A vast majority of bigrams occur very infrequently.

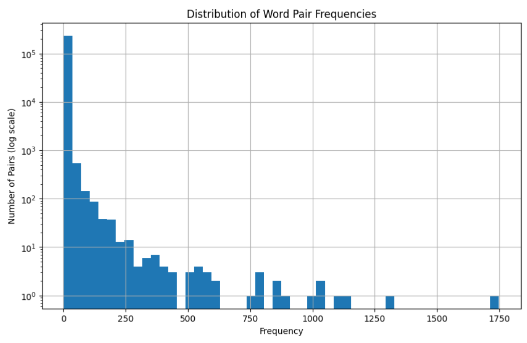

Figure 3: Histogram of Bigram Frequencies (Log Scale).

5 Exercise 5: Tokenize the dataset

The ‘tokenize‘ function splits the cleaned text by spaces and replaces words not found in the provided vocabulary (limited to the top 10,000 most frequent words from training) with the ‘<unk>‘ token.

```
def tokenize(example, vocab, unknown_token='<unk>'):
    text = example['clean']
    tokens = None
    ### YOUR CODE HERE
    tokens = [n if n in vocab else "<unk>"
    for n in text.split(" ")]
    ### YOUR CODE ENDS HERE
    example['tokens'] = tokens
    return example

tweet_ds = tweet_ds.map(tokenize,
fn_kwargs={'vocab': vocab})

for i in range(10):
    print('Original tweet:')
    print(tweet_ds['train'][i]['text'])
    print('Tokenized tweet:')
    print(tweet_ds['train'][i]['tokens'])

#Original tweet:
#Sunday afternoon walking through Venice
# in the sun with @user  @ Abbot Kinney, Venice
#Tokenized tweet:
#['sunday', 'afternoon', 'walking', 'through',
#'venice', 'in', 'the', 'sun', 'with', '<unk>',
#'user', '<unk>', 'abbot', 'kinney', '<unk>',
#'venice']
```

This function implements basic word-level tokenization with out-of-vocabulary handling using a predefined vocabulary and a special unknown token.

6 Exercise 6: Questions about the tokenization

We analyze the results of the word-level tokenization with a vocabulary size of 10,000.

1. **Unknown tokens in validation:** The number of ‘<unk>‘ tokens in the validation set was counted.

```
counter = 0
for n in tweet_ds["validation"]:
  counter += n["tokens"].count("<unk>")
print(counter)
# Output: 21851
```

   There are 21,851 ‘<unk>‘ tokens in the validation dataset using a vocabulary size of 10,000 derived from the training set. That is a lot, a fair share of words in validation are not in the training set.

2. **Distribution of token counts (training):** The distribution of the number of tokens per tweet in the training set is shown in Figure 4.

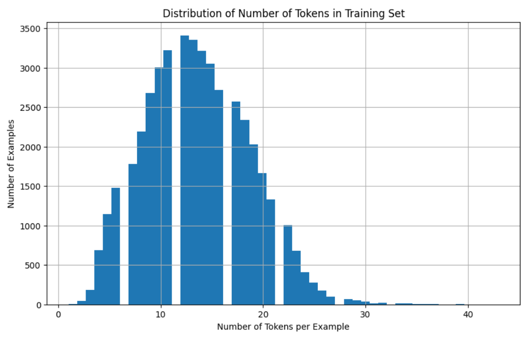

Figure 4: Distribution of the Number of Tokens per Tweet (Training Set).

   The distribution is right-skewed, with mode around 13 tokens.

3. **Tokens vs. Characters:** Figure 5 shows the relationship between the number of characters and the number of tokens per tweet. There is a clear positive correlation: longer tweets

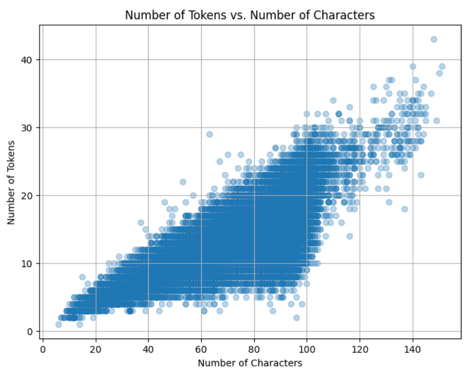

Figure 5: Number of Tokens vs. Number of Characters per Tweet.

(more characters) tend to have more tokens. However, the relationship isn't perfectly linear due to variations in word length. Also, all unknown tokens are replaced with just <unk> and we know that we have a bunch of them.

4. **Vocabulary Size vs. Unknown Tokens:** Figure 6 shows how the total number of '<unk>' tokens across the entire dataset changes with vocabulary size. As the vocabulary size in-

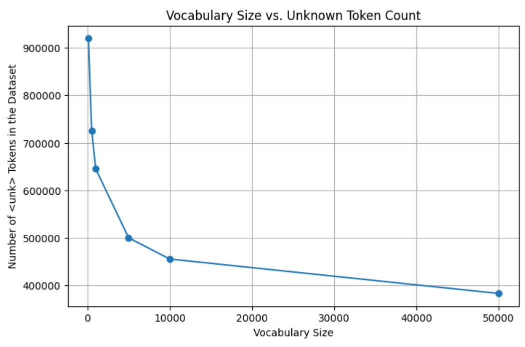

Figure 6: Effect of Vocabulary Size on the Total Number of '<unk>' Tokens.

creases, the number of '<unk>' tokens decreases significantly obviously, as more words from the dataset are included in the vocabulary. The decrease follows a such a curve due to Zipf's law.

5. **Vocabulary Size vs. Number of Tokens:** Figure 7 shows how the total number of known (non-'<unk>') tokens changes with vocabulary size.

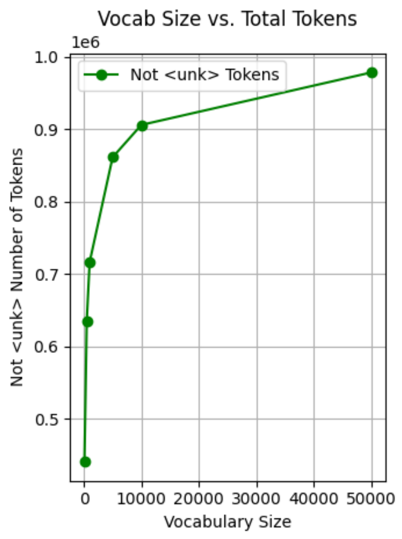

Figure 7: Effect of Vocabulary Size on the Total Number of Known (Non-'<unk>') Tokens.

Conversely to the '<unk>' count, the number of known tokens increases as the vocabulary size grows for obvious reasons (we find more rare words etc.)

6. **Advantages/Disadvantages:**
Advantages: The main advantage is that this tokenization method is super simple and fast. Also, using a fixed vocabulary is a really intuitive idea, and we can control what we consider words and what not.
Disadvantages: It really struggles with out-of-vocabulary) words, just turning them all into a single <unk> token, so we lose a lot of info. It's not great at handling different forms of the same word—like "run," "running," and "ran", because these words can be turned into different tokens and it is bad, because we want to understand which words have similar meaning. It is language dependent since it relies on spaces, so it doesn't work at all for languages like Chinese or Japanese. Also, it treats hashtags and mentions as single chunks, which isn't always the best approach.

7 Exercise 7: Counting the characters

This function counts the frequency of each character in the cleaned training data. Then we find the most common symbols, filter out rare occurring characters and replace spaces with the underscore.

```
from collections import Counter

def build_character_counter(dataset):
    char_counter = Counter()
    ### YOUR CODE HERE
    for n in dataset["clean"]:
        for j in n:
            char_counter[j]+=1

    ### YOUR CODE ENDS HERE
    return char_counter

char_counter = build_character_counter(tweet_ds['train'])
print(len(char_counter))
print(char_counter.most_common(100))
# Output: 511
# [(' ', 566053), ('e', 267923), ('a', 220986)..]

bpe_init_vocab = sorted([char for char, _
in char_counter.most_common() if
char_counter[char] >= 10])
bpe_init_vocab[bpe_init_vocab.index(' ')] = '__'
print(bpe_init_vocab)
# ['__', '!', '"',
#'#', '$', '%', '&', "'
```

There are 511 unique characters in the cleaned training dataset. The most common character is the space, followed by common English letters like 'e', 'a', 'o', 't', 'i'. Special characters like '' and '@' are also relatively frequent due to the nature of tweets.

7.1 Training BPE Tokenizer

```
def init_bpe_corpus(dataset):
    """
    Initializes the BPE corpus
    Args:
        dataset: a dataset

    Returns: a BPE corpus

    """

    corpus = Counter()
    for example in dataset:
        words = example['clean'].split()
        words = [' '.join(list(word)) +
        ' __' for word in words]
        corpus.update(words)
    return corpus

bpe_corpus = init_bpe_corpus(tweet_ds['train'])
print(len(bpe_corpus))
# 68031
bpe_corpus.most_common(30)
[('@ __', 36756),
 ('. __', 20870),
 ('! __', 16957),
 ('t h e __', 13388), .....
```

First, we initialized BPE corpus and checked for its length and the most common words.

8 Exercise 8: Calculate the frequency statistics of adjacent symbol pairs

The following function computes the frequency of adjacent characters (or tokens after merges) within words in the BPE corpus.

```
from collections import Counter

def calculate_bpe_corpus_stats(corpus):
    """
    Calculates the frequency statistics
    of adjacent symbol pairs in the corpus.
    Args:
        corpus: a BPE corpus as a Counter object
      with words split by space into tokens (initially characters)

    Returns: a Counter object with the frequency
    statistics of adjacent symbol pairs
    """
    stats = Counter()

    for word, freq in corpus.items():

        ### YOUR CODE HERE
        word = word.split(" ")
        pairs = [(word[index], word[index+1])
        for index in range(len(word)-1)]
        for pair in pairs:
            stats[pair] += freq
        ### YOUR CODE ENDS HERE

    return stats

stats = calculate_bpe_corpus_stats(bpe_corpus)
print(stats.most_common(10))
# [(('e', '__'), 80798), (('s', '__'), 53536) ...

def merge_corpus(corpus, pair):
    """
   Merges the most frequent pair of symbols in the corpus.
    Args:
      corpus (dict): Keys are words as space-separated symbols (e.g.
                and values are the frequency counts.
        pair (tuple): A pair of symbols to merge.

    Returns:
      dict: Updated corpus after merging the pair of symbols.
    """
    new_corpus = Counter()
    bigram = " ".join(pair)
    replacement = "".join(pair)
    for word, freq in corpus.items():
      new_word = word.replace(bigram, replacement)
        new_corpus[new_word] = freq
    return new_corpus
```

This function goes through each "word" in the text (basically, chunks of text separated by spaces) and looks at every pair of symbols that are next to each other. It then keeps track of how often each pair shows up by adding to a stats counter, based on how many times the word they're in appears. At the start, the most common pairs are usually things like frequent letters followed by the special word-ending marker underscore (like ('e', 'underscore')), or common letter combos such as ('t', 'h') for "th" or ('e', 'r') for "er".

Then we merge the most frequent pair of symbols in the corpus.

9 Exercise 9: BPE algorithm

The bpe function implements the Byte Pair Encoding algorithm. It iteratively finds the most frequent adjacent pair of symbols in the corpus, merges them into a new symbol, updates the corpus, and adds the new symbol to the vocabulary.

```
from collections import Counter
```

```
def bpe(vocab, corpus, num_merges):
    vocab = vocab.copy()
    corpus = corpus.copy()
    merges = []

    for i in tqdm.tqdm(range(num_merges)):
         ### YOUR CODE HERE
       stats = calculate_bpe_corpus_stats(corpus)
     most_common_pair = stats.most_common(1)[0][0]

     corpus = merge_corpus(corpus, most_common_pair)
        merges.append(most_common_pair)

        token = "".join(most_common_pair)

        if (token not in vocab):
            vocab.append(token)


        ### YOUR CODE ENDS HERE

    return vocab, corpus, merges

# Example run with 100 merges (from notebook)
bpe_vocab, updated_bpe_corpus, bpe_merges =
bpe(bpe_init_vocab,
bpe_corpus, num_merges=100)
print(len(bpe_vocab))
# Output: 192
print(bpe_merges[:10])
# Output:
# [('e', '__'), ('s', '__'), ('t', '__'), ('t', 'h'),
# ('e', 'r'), ('y', '__'), ('i', 'n'), ('@', '__'),
# ('a', 'n'), ('d', '__')]
```

The function successfully implements the BPE merging process. After 100 merges, the vocabulary size increased from the initial character set size (around 92 after filtering) to 192. The first few merges combine frequent characters with the end-of-word marker underdscore or common digraphs like ‘th‘ and ‘er‘.

10 Exercise 10: Comparing tokenizers

We compare the word-level tokenizer (vocabulary size 10,000) with BPE tokenizers trained with varying numbers of merges (50, 100, 250, 500, 750, 1000). The analysis focuses on the validation set.

```
# 1. (5p) What are the differences?
# 2. (5p) Compare the number of tokens created by
# your tokenizers.
# 3. (5p) Calculate the number of `<unk>` tokens
# in the validation dataset for each tokenizer.
# 4. (5p) Compare the average length in tokens
# between different tokenizers. (avg length of
the token or the avg length of the tokens in
the sentence?)
```

```
merge_steps = [50, 100, 250, 500, 750, 1000]
unk_count = []
token_count = []
avg_length = []
avg_token_length = []

for num_merges in merge_steps:
    new_vocab, updated_bpe_corpus, new_merges
    = bpe(bpe_init_vocab, bpe_corpus, num_merges)

    total_unk = 0
    total_tokens = 0
 total_examples = 0
 total_token_chars = 0

 tokenized_ds = tweet_ds.map(tokenize_bpe,
fn_kwargs={'vocab': new_vocab, 'merges': new_merges})

 for example in tokenized_ds["validation"]:
     tokens = example["bpe_tokens"]
     total_unk += tokens.count("<unk>")
     total_tokens += len(tokens)
   total_token_chars += sum(len(token) for token in tokens)

     total_examples += 1

 unk_count.append(total_unk)
 token_count.append(total_tokens)
avg_length.append(total_tokens / total_examples)
avg_token_length.append(total_token_chars / total_tokens)

print(f"Merges: {num_merges} → <unk>: {total_unk},
Total tokens: {total_tokens} Avg sent len: {avg_length[-1]},
 Avg token len: {avg_token_length[-1]}")
```

1. **Differences:** Word tokenization breaks text into pieces using spaces and turns each word into a token from a set list, or it uses <unk> if the word isn't in that list. It basically sees each whole word as one unit. On the other hand, BPE begins with individual characters and then keeps combining the most common pairs of symbols next to each other to form subword units. This way, it can build any word using the subwords or characters it's learned, which means it doesn't need <unk> tokens as long as the characters are known. BPE tokens can be anything from a single character to longer chunks of a word.

2. **Number of Tokens:** Figure 8 shows the total number of tokens produced by BPE on the validation set for different numbers of merges. As the number of merges

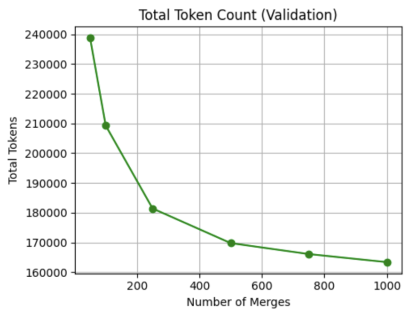

Figure 8: Total Number of Tokens (Validation Set) vs. Number of BPE Merges.

increases, BPE learns longer, more frequent subwords. This leads to a decrease in the total number of tokens needed to represent the same text, as frequent sequences are represented by single, longer tokens instead of multiple shorter ones. The word tokenizer (with vocab 10k) produced 112,431 tokens on the validation set (calculated from avg length 22.4862 * 5000 examples, see Q4). BPE consistently produces more tokens initially (starting from characters) but the count decreases with more merges, eventually becoming comparable to or less than the word tokenizer if enough merges are performed to capture common words.

3. **Number of ‘<unk>‘ Tokens:** Figure 9 compares the ‘<unk>‘ counts. The word tokenizer (vocab 10k) produced 21,851 ‘<unk>‘ tokens. BPE, by design, produces very few ‘<unk>‘ tokens (only 95 in our runs, likely due to characters filtered out initially). This is a major advantage of BPE – its ability to handle OOV words by breaking them down into known subwords or characters. The number of merges doesn’t affect the count because it doesn’t matter how many folds we make, if we don’t some of validation tokens in training, we will still not have them in training with another merge amount.

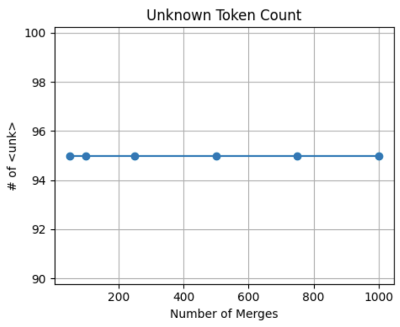

Figure 9: Number of ‘<unk>‘ Tokens (Validation Set) vs. Number of BPE Merges.

4. **Average Length:** Figures 10 and 11 show the average sentence length (in tokens) and average token length (in characters) for BPE. The average sentence length for the word tokenizer (vocab 10k) is approx. 15 tokens based on the Figure 4. For BPE, the average sentence length decreases as merges increase (from 47.7 to 32.7 tokens), so we can see that BPE tokens are shorter and they get longer when number of merges increse. Conversely, the average token length increases with more merges (from 1.68 to 2.46 characters), as merges create longer subword units.

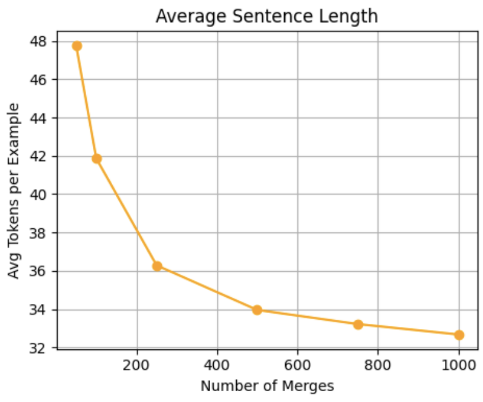

Figure 10: Average Sentence Length (Tokens) vs. Number of BPE Merges.

5. **BPE Advantages/Disadvantages:**
Advantages: BPE does a great job at solving the OOV issue for words made up of characters it already knows. You can control the size of the vocabulary by deciding how many merges to do, which lets you find a good balance between detail and keeping the vocabulary manageable. Plus, it often picks up on meaningful subword pieces—like parts of words that carry meaning—which can really help with tasks down the line, especially for languages with a lot of word variations.
Disadvantages: The BPE training process (finding frequent pairs iteratively) can be computationally more expensive than simple word counting. The resulting subwords might not always align with linguistic morphemes and can sometimes be arbitrary splits based purely on frequency. The tokenization of a word can depend on the context of merges performed, and the greedy nature of the algorithm doesn’t guarantee an optimal segmentation. It still relies on an initial character vocabulary; truly unknown characters would still become ‘<unk>‘.

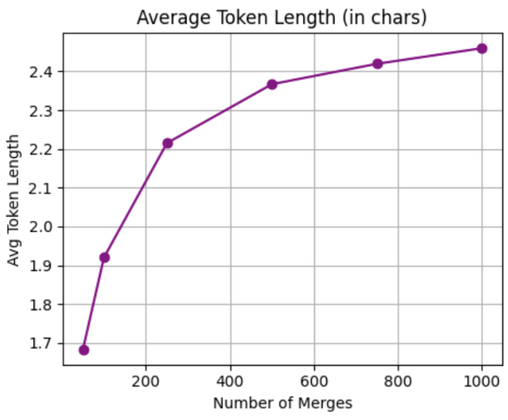

Figure 11: Average Token Length (Characters) vs. Number of BPE Merges.

References

A Collaborators outside our group

None.

B Use of genAI

ChatGPT (model o3-mini-high) was used occasionally during the lab, specifically for:

- Double-checking if code implementations correctly matched task descriptions. Example prompt: *"Does this Python function correctly implement spacing punctuation as described in Exercise 2? [code snippet]"*
- Clarifying syntax and usage examples of Python libraries (mainly Pandas, Matplotlib, and the `re` module). Example prompt: *"How to retrieve the top 5 most frequent values using from a column labeled ’label’ in a DataFrame? [code snippet]"*
- Moving chunks of code from ipynb to the report.
- Checking for spelling.