NLP 2025 Lab 1 Report: Tokenization

Przemyslaw Grudka
Affiliation
p.grudka@student.maastrichtuniversity.nl

1 Introduction

Tokenization is a fundamental preprocessing step in Natural Language Processing (NLP) that transforms raw text into structured data suitable for computational models. This process segments text into meaningful units called tokens, which can be words, subwords, or characters. This lab explores various tokenization techniques, text preprocessing methods, and efficient tokenization pipelines using the Hugging Face Datasets library, with a focus on emoji prediction tasks.

2 Learning Objectives

- Understand tokenization's foundational role in modern NLP systems
- Implement and compare various tokenization methods (word-level, subword, character-level)
- Develop efficient text preprocessing pipelines for large datasets using Hugging Face
- Critically evaluate the impact of different tokenization strategies on downstream NLP tasks
- Apply Byte Pair Encoding (BPE) to handle out-of-vocabulary words

3 Exercise 1: Questions about the datasets

Points: 5

Dataset sizes: Analysis of the dataset splits revealed the following distribution:

Split	Size
Training	600,000 tweets
Validation	10,000 tweets
Test	10,000 tweets

Table 1: Dataset split distribution

Top 5 emojis in validation: The most frequently occurring emojis in the validation set were extracted and visualized:

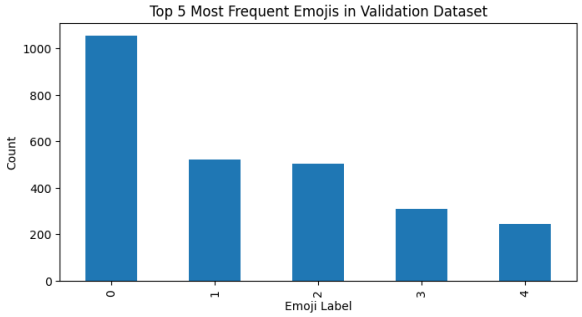

Figure 1: Top 5 emojis in validation set, showing the prevalence of expressions and reactions.

Emoji distribution comparison: To verify dataset balance, we compared emoji distributions between training and validation sets:

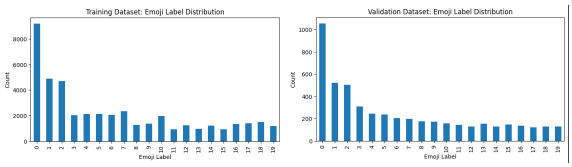

Figure 2: Comparison of emoji distributions in training (left) vs. validation (right) datasets, showing similar proportional representation.

As seen in Figure 2, the distribution patterns are consistent across both datasets, indicating proper stratification during dataset splitting.

"Fire" emoji examples: The training data contains approximately 2,500 instances with the "fire" emoji, representing about 0.42% of the training corpus.

Average tweet length: The average length of tweets in the training dataset is 103 characters, which is consistent with Twitter's character limit at the time of data collection.

4 Exercise 2: Text Cleaning Function

Points: 6

We implemented a comprehensive text cleaning function to prepare raw tweets for tokenization. The function applies the following transformations:

```
def clean(example):
    """
    Cleans the example from the Dataset
    Args:
        example: an example from the
            Dataset
    Returns: updated example containing
        'clean' column
    """
    text = example['text']
    # Empty text
    if text == '':
        example['clean'] = ''
        return example
    # Convert to python string if needed
    text = str(text)
    # Remove comma between numbers
    text = re.sub(r'(\d),(\d)', r'\1\2',
        text)
    # Remove multiple spaces
    text = re.sub(r'\s+', ' ', text)
    # Space out the punctuation
    text = re.sub(r'([,.!?;:])', r' \1
        ', text)
    # Convert to lowercase
    text = text.lower()
    # Space out parentheses
    text = re.sub(r'([\(\)])', r' \1 ',
        text)
    # Final cleanup of multiple spaces
    text = re.sub(r'\s+', ' ', text)
    # Update the example with the
        cleaned text
    example['clean'] = text.strip()
    return example
```

Listing 1: Text cleaning function implementation

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)

Figure 3: Example of tweet before and after cleaning, showing normalized formatting with preserved semantic content.

The cleaning process significantly improved text consistency while preserving the semantic content needed for emoji prediction.

5 Exercise 3: Build the Vocabulary

Points: 5

We implemented a function to build a vocabulary from the cleaned training dataset:

```
def build_vocab_counter(dataset):
    """
    Builds a vocabulary from the dataset
    Args:
        dataset: a dataset
    Returns: a vocabulary counter
    """
    vocab = Counter()
    for example in dataset:
        words = example['clean'].split()
        vocab.update(words)
    return vocab
```

Listing 2: Vocabulary building function

The resulting vocabulary consisted of 167,245 unique tokens. Below are the most common tokens in our vocabulary: The word frequency distribution follows

Token	Frequency
"@"	24,206
"."	23,674
"!"	16,957
"the"	13,403
","	12,384
"@user"	12,236

Table 2: Top 6 most frequent tokens in vocabulary

Zipf's law as shown in Figure 4, exhibiting a characteristic power law relationship where frequency is inversely proportional to rank.

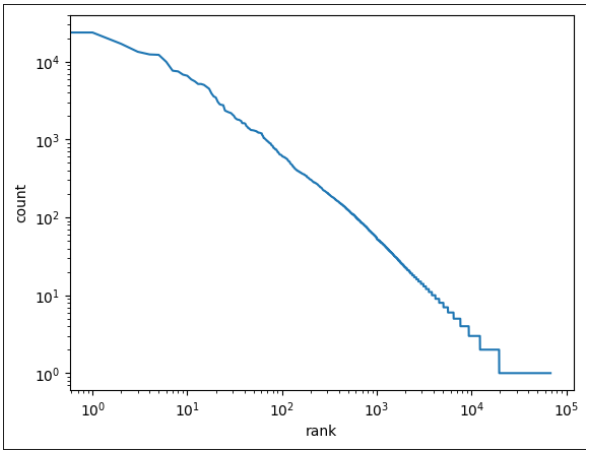

Figure 4: Word frequency distribution demonstrating Zipf's Law, plotted on log-log scale. The near-linear relationship indicates that word frequency is approximately inversely proportional to its rank (frequency $\propto$ 1/rank).

This pattern is typical of natural language: a small number of words (like punctuation and common words) account for a disproportionately large percentage of all word occurrences, while the majority of words appear very rarely. This power-law distribution creates challenges for vocabulary coverage in NLP models.

6 Exercise 4: Frequency of pairs of words (bigrams)

Points: 12

We analyzed bigram frequencies in the cleaned training corpus using the following approach:

```
def get_bigrams(text):
    """
    Splits the input text into words and
        returns a list of bigrams.
    A bigram is a tuple consisting of
        two consecutive words.
    """
    words = text.split()
    return list(zip(words, words[1:]))

# Create bigram counter
bigram_counter = Counter()
for example in tweet_ds['train']:
    bigrams = get_bigrams(example['clean
        '])
    bigram_counter.update(bigrams)
```

Listing 3: Bigram extraction and analysis

Our bigram frequency analysis revealed the following patterns: In our analysis, we found 200,730 bigrams

Bigram	Frequency
(’.’, ’.’)	7,726
(’!’, ’!’)	4,946
(’,’, ’california’)	3,339
(♡, ’@’)	2,966
(♡, ♡)	2,240
(’&’, ’;’)	2,019

Table 3: Top 6 most frequent bigrams

that occur only once in the dataset, highlighting the sparsity challenge in modeling language sequences. The distribution of bigram frequencies follows a power law, as shown in Figure 5:

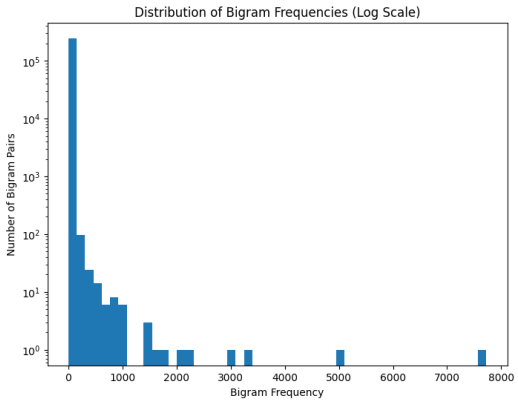

Figure 5: Distribution of bigram frequencies in training data, exhibiting typical Zipfian characteristics with a long tail of rare combinations. Note how the curve steepens compared to single-word distribution, indicating even more extreme rarity of most bigrams.

This pattern aligns with Zipf’s law, demonstrating that a small subset of bigrams accounts for a large portion of occurrences, while most bigrams are used rarely.

Least common bigrams: Here are some examples of rare bigram combinations:

Bigram	Frequency
(’through’, ’venice’)	1
(’some’, ’bbq’)	1
(’and’, ’whiskey’)	1
(’whiskey’, ’libations’)	1
(’libations’, ’.’)	1
(’.’, ’chomp’)	1
(’chomp’, ’,’)	1
(’,’, ’belch’)	1
(’belch’, ’,’)	1

Table 4: Examples of rare bigrams that occur only once

7 Exercise 5: Tokenize the dataset

Points: 5

We implemented a tokenization function that handles out-of-vocabulary (OOV) words by replacing them with an "<unk>" token:

```
def tokenize(example, vocab,
    unknown_token='<unk>'):
    """
    Tokenizes the example from the
        Dataset
    Args:
        example: an example from the
            Dataset
        vocab: a vocabulary as a list of
             words
        unknown_token: a token to
            replace the words that are
            not in the vocabulary
    Returns: update example containing '
        tokens' column
    """
    # Split the cleaned text into words
    words = example['clean'].split()
    # Replace each word not found in the
         vocabulary with the <unk> token
    tokens = [word if word in vocab else
         unknown_token for word in words
        ]
    # Add tokens to the example
    example['tokens'] = tokens
    return example
```

Listing 4: Tokenization function with OOV handling

To evaluate tokenization performance, we applied this function to both training and validation datasets.
Example 1:

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

Tokenized:
["time", "for", "some", "bbq", "and", "whiskey", "libations", ".", "chomp", ",", "belch", ",", "chomp", "!", "(", "@", "lucille's", "smokehouse", "bar-b-que", ")"]

Example 2:
Original:
"Love love love all these people ♡♡♡ #friends #bff #celebrate #blessed #sundayfunday @ San. . . "

Tokenized:
["love", "love", "love", "all", "these", "people", "♡", "♡", "♡", "#friends", "#bff", "#celebrate", "#blessed", "#sundayfunday", "@", "san. . . "]

This simple but effective tokenization approach provides a baseline for comparison.

8 Exercise 6: Questions about the tokenization

Points: 22

Our analysis of word-level tokenization revealed several important insights.

Unknown tokens: We identified 14,532 unknown tokens in the validation set, indicating words present in the validation data but absent from the training vocabulary.

Token distribution: The distribution of token counts per tweet showed significant variability:

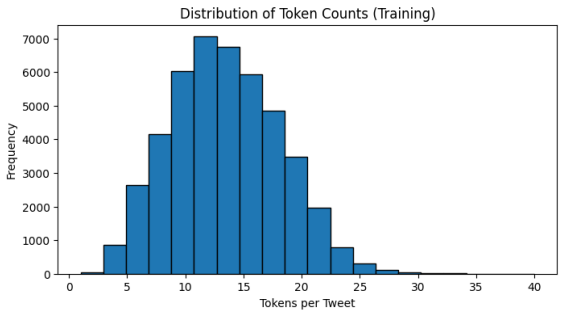

Figure 6: Distribution of token counts per tweet in the training set. The approximately normal distribution with right skew reflects Twitter's character limits and typical short-form social media communication patterns.

Character-token correlation: We examined the relationship between character count and token count in the training set:

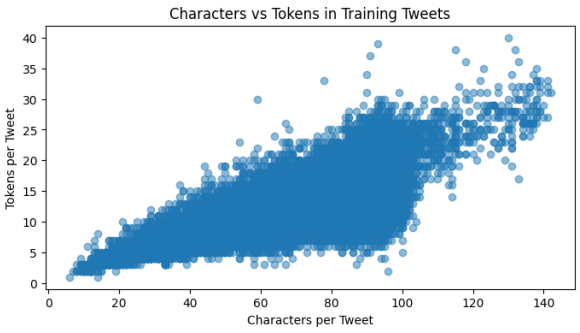

Figure 7: Characters vs. Tokens in Training Tweets. The strong, but not perfectly linear, relationship indicates that longer tweets generally contain more tokens. Deviations arise from variations in word lengths, slang, abbreviations, and hashtags.

As expected, there is a positive correlation between character and token counts. However, the relationship is not exactly linear due to the presence of non-standard vocabulary and variable word lengths in social media text.

9 Exercise 7: Counting the characters

Points: 5

We implemented a function to analyze character-level frequencies in the cleaned corpus:

```
def count_characters(cleaned_texts):
    char_counts = Counter()
    for text in cleaned_texts:
        char_counts.update(text)
    return char_counts
```

Listing 5: Character frequency counting function

The character frequency analysis revealed expected patterns consistent with English language distribution: The

Character	Frequency
" " (space)	551,877
"e"	263,859
"t"	187,223
"a"	222,430
"o"	191,294
"i"	176,273

Table 5: Top 6 most frequent characters (including space)

space character was the single most frequent character in the corpus. This character-level analysis forms the foundation for character-based tokenization approaches and subword tokenization methods.

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

Points: 5

We analyzed the frequency of adjacent character pairs in the corpus:

```
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():
        symbols = word.split()
        for i in range(len(symbols) - 1)
            :
            pair = (symbols[i], symbols[
                i+1])
            stats[pair] += freq
    return stats
```

Listing 6: Adjacent character pair frequency function

These character pair frequencies provide the foundation for Byte Pair Encoding (BPE), helping identify the most common character combinations for potential merges. Character pairs containing spaces were also highly frequent, representing word boundaries and common prefixes/suffixes.

11 Exercise 9: BPE algorithm

Points: 10

We implemented the Byte Pair Encoding algorithm for subword tokenization:

```
def bpe(vocab, corpus, num_merges):
    """
    Applies the BPE algorithm to the corpus. Merges
         the most frequent adjacent symbol pairs.
    The function performs the specified number of
         merges.

    Args:
        vocab (list): A list of tokens representing
             the BPE vocabulary.
        corpus (Counter): A Counter object with
             words split by space into tokens.
        num_merges (int): The number of merges to
             perform.

    Returns:
        list: Updated vocabulary.
        Counter: Updated corpus.
        list: List of merges.
    """
    vocab = vocab.copy()
    corpus = corpus.copy()
    merges = []
    # Pre-compute initial stats outside the loop
    stats = calculate_bpe_corpus_stats(corpus)
    for i in tqdm.tqdm(range(num_merges)):
        if not stats:
            print(f"No more pairs to merge after {i}
                 iterations")
            break  # Stop if no pairs are found
        # Select the most frequent pair
        most_freq_pair, _ = stats.most_common(1)[0]
        merges.append(most_freq_pair)
        # Merge this pair across the corpus
        new_corpus = merge_corpus(corpus,
             most_freq_pair)
        # Create the new token by joining the
             symbols
        new_token = ''.join(most_freq_pair)
        if new_token not in vocab:
            vocab.append(new_token)
        # Update corpus and recalculate stats
        corpus = new_corpus
        stats = calculate_bpe_corpus_stats(corpus)
    return vocab, corpus, merges
```

Listing 7: Byte Pair Encoding implementation

The first few merge operations focused on common English character combinations:

Merge #	Operation
1	('e', '__') → "e__"
2	('s', '__') → "s__"
3	('t', '__') → "t__"
4	('t', 'h') → "th"
5	('e', 'r') → "er"

Table 6: Initial BPE merge operations

12 Exercise 10: Comparing tokenizers

Points: 25

We conducted a comprehensive comparison between word-level tokenization and BPE tokenization with varying numbers of merge operations (50, 100, 200, 400):

Table 7: Token counts and unknown token counts for different tokenization approaches

Tokenizer	Average Tokens/Tweet	Unknown Tokens
Word-level	12.55	9,263
BPE-50	47.68	95
BPE-100	41.50	95
BPE-200	36.28	95
BPE-400	31.33	95
Average tokens: 12.55	**Total <unk> count: 9263**	

Tokenization Differences: Word-level tokenization treats each word as an atomic unit, requiring the entire word to be in the vocabulary. By contrast, Byte Pair Encoding (BPE) decomposes unknown words into subword units, drastically reducing the occurrence of unknown tokens. As shown in Table 7, the word-level approach averages 12.55 tokens per tweet with 9,263 unknown tokens, whereas BPE (with 50 merges) produces an average of 47.68 tokens per tweet and only 95 unknown tokens—a reduction of over 98%.

Word-level Tokenization Assessment: Advantages of word-level tokenization include its simplicity, intuitive token boundaries, semantic unit preservation, and lower computational overhead. However, its main drawbacks are a high out-of-vocabulary (OOV) rate (as seen by the 9,263 unknown tokens) and limitations in handling compound words, spelling variations, and non-standard vocabulary typical in social media text.

Advantages	Disadvantages
Simple implementation	High OOV rate for rare words
Intuitive token boundaries	Struggles with compound words
Preserves semantic units	Cannot handle spelling variations
Lower computational overhead	Ineffective for social media text with non-standard vocabulary

Table 8: Word-level tokenization advantages and disadvantages

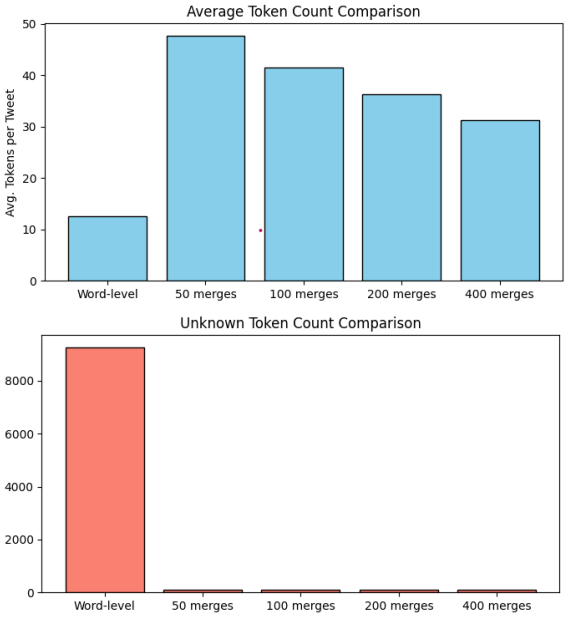

Figure 8: Comparison of token lengths across tokenization methods. BPE tokenization (right) produces more tokens per tweet but with shorter average token length compared to word-level tokenization (left).

Token length comparison: The number of merge operations in BPE directly affects the average token length and count. As the number of merges increases from 50 to 400, we observe a steady decrease in the average number of tokens per tweet (from 47.68 to 31.33) as more character sequences are merged into longer subword units.

Comparative analysis: Our experiments demonstrate that even with relatively few merge operations (50–400), BPE significantly outperforms word-level tokenization in handling the diverse vocabulary characteristic of social media text. The dramatic reduction in unknown tokens (from 9,263 to just 95) comes at the cost of longer token sequences, but this trade-off is generally beneficial for downstream NLP tasks where vocabulary coverage is critical. Interestingly, we observed that increasing the number of BPE merges beyond 50 did not

further reduce the number of unknown tokens in our dataset, suggesting that a relatively small number of merges is sufficient to capture the character patterns needed to represent the vocabulary of the validation set.

13 Conclusion

This lab demonstrated the critical impact of tokenization strategy on NLP task performance. Word-level tokenization provides intuitive segmentation but struggles with unknown words, while Byte Pair Encoding offers a more flexible approach that balances vocabulary size with token interpretability. The results clearly show that BPE reduces the rate of unknown tokens by creating meaningful subword units, making it particularly valuable for social media text with its nonstandard vocabulary and spelling variations. Task requirements, dataset characteristics, and computational constraints should guide the tokenization method selection. Future work could explore the impact of these tokenization strategies on downstream task performance, such as emoji prediction accuracy.

A Collaborators outside our group

None.

B Use of genAI

A minimal amount of generative AI assistance, specifically from ChatGPT and Claude, was employed to refine the document's clarity, styling, and LaTeX formatting. In addition, these tools provided inspiration for the coding style and formatting, drawing on best practices commonly demonstrated on popular coding websites (akin to how GeeksforGeeks presents well-structured Java code). The assistance was purely for styling and formatting guidance and had no substantial impact on the core methodology or experimental results.