NLP 2025 Lab 1 Report: Tokenization

Marta Adamowska[*] **Katarina Dvornák**[†] **Pavels Sorocenko**[‡]

1 Exercise 1: Questions about the datasets

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

training set -> (45000, 3)
test set -> (50000, 3)
validation set -> (5000, 3)

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

Labels of top 5 frequent emojis in in the validation dataset: 0, 1, 2, 3, 4.

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

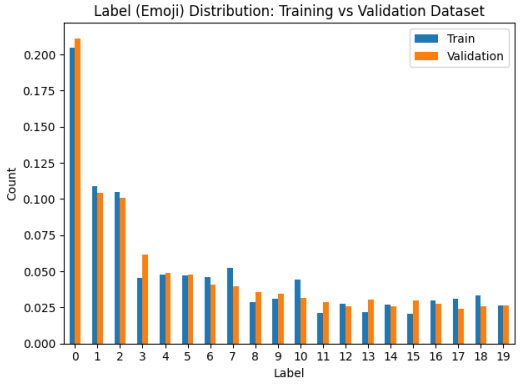

Figure 1: Comparison of the distributions of labels (emojis) between training and validation datasets

The counts for both the training and validation sets are illustrated in the bar plot shown in Figure 1. Since the validation set is significantly smaller than the training set, the data were normalized before plotting. Given that both sets originate from the same dataset and the values in the plot are similar, it is likely that the split was performed randomly.

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

There are 2146 examples with the fire emoji (label 4) in the training set.

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

The average length of a tweet in the training set is 71.02 characters.

2 Exercise 2: Write the text cleaning function

The text cleaning function consists of several cleaning steps. As the task required, three cleaning steps were performed:

```
1)  re.sub(r"(?<=\d),(?=\d)", "", text)
```

It selects commas and if a digit is detected before and after it, it removes itself.

```
2)  re.sub(r'([. . . .,!?;:(){}[\]<>])', r' \1 ', text)
```

Select the listed punctuation and space it out based on the '1' flag.

```
3) re.sub(' +', ' ', text)
```

Detects one or more spaces and substitutes for one single space.

In addition, seven unique cleaning steps were performed:

```
4) re.sub(r'\.+', r'.', text)
```

Replace multiple fullstops with one to perform the tokenization process faster.

```
5) re.sub(r'([@|#])', ' ', text)
```

Removes '@' (et character) and '#' (hashtag | diez | sharp character) to eliminate redundancy

```
6) re.sub(r'\b[Tt]he\b', ' ', text)
7) re.sub(r'\b[Aa]\b|\b[Aa]n\b', ' ', text)
```

[*] m.adamowska@student.maastrichtuniversity.nl
[†] k.dvornak@student.maastrichtuniversity.nl
[‡] p.sorocenko@student.maastrichtuniversity.nl

Removes 'the', 'an', 'a' articles from the text

`8) re.sub(r'\b&\b', 'and', text)`

Replaces '&' character with 'and' string if it is spaced out

`9) re.sub(r'[\n\t]', '', text)`

Removes tabs \t and escapes \n from a text

```
10) re.sub(r'[\u200B-\u200D\uFE0F\
     u200E\u200F]', '', text)
```

Removes variation selectors and zero-width characters that created and illusion of multiple spaces

It is important to note that the code removes redundant spaces in the end by repeating the third step.

3 Exercise 3: Build vocabulary

We built a vocabulary from the cleaned text in the training data set by counting all unique word occurances. The training set contains a total of 69'388 unique words, which indicates a high lexical diversity. This aligns with our expectations, given that Twitter uses informal language with hashtags, usernames, slang, etc.

```
def build_vocab_counter(dataset):
    vocab = Counter()
    for tweet in dataset:
        words = tweet["clean"].split()
        vocab.update(words)
    return vocab
```

The most frequent words are punctuation marks and common words such as "user". The most common words were: `('. . . ', 19238)`, `('!', 16957)`, `('.', 16089)`, `(',', 12383)`, and `('user', 12236)`. The least common words include usernames, slang, hashtags, and rare proper nouns. A large number of words appear only once. Some examples of least common words (frequency = 1) include: `('southbayla', 1)`, `('thedabberchick', 1)`, `('nector', 1)`, `('chefking1921express', 1)`, ...

The plotted frequency distribution follows Zipf's law. This confirms that a small number of words are frequent, while most words occur very rarely.

4 Exercise 4: Frequency of pairs of words

```
def get_bigram_frequencies(dataset):
    bigram_counter = Counter()

    for example in dataset:
        words = example['clean'].split()
        bigrams = zip(words, words[1:])
        bigram_counter.update(bigrams)

    return bigram_counter
```

In this exercise we computed the frequency of all consecutive word pairs (bigrams) from the cleaned tweets in the training data set. The 10 most frequent bigrams are:

```
(('!', '!'), 4947)
((',', 'California'), 3343)
(('&amp', ';'), 2020)
(('Los', 'Angeles'), 1738)
(('with', 'my'), 1015)
(('Angeles', ','), 998)
(('user', 'user'), 984)
(('Las', 'Vegas'), 982)
((',', 'CA'), 867)
((':', 'user'), 816)
```

Many of these pairs are commonly used or compound expressions. Repeated punctuation "!!" reflects expressivity in tweets, while "Los Angeles", "Las Vegas", and ", California" are locations which come up frequently in real-world tweets. HTML-escaped characters like "&" suggest that some tweets could not be fully decoded. Examples of bigrams that occur only once:

```
(('.', 'happylaborday'), 1)
(('happylaborday', 'Beer'), 1)
(('Beer', 'NV'), 1)
(('Pizza', '('), 1)
(('(', 'Five50'), 1)
(('Five50', '-'), 1)
(('mini', 'is'), 1)
(('one', 'deserves'), 1)
(('deserves', 'her'), 1)
(('her', 'Las'), 1)
```

These represent rare combinations of words, often coming from specific hashtags, usernames, event mentions, or brand names. A total of 226'946 bigrams in the training set occurred exactly once, indicating that the dataset contains a large amount of rare or unique language.

The logarithmic scaled histogram shows that most bigrams are low frequency, with a few occurring frequently. This aligns with our expectations of langauge in social media, that the data has a heavy-tailed distribution.

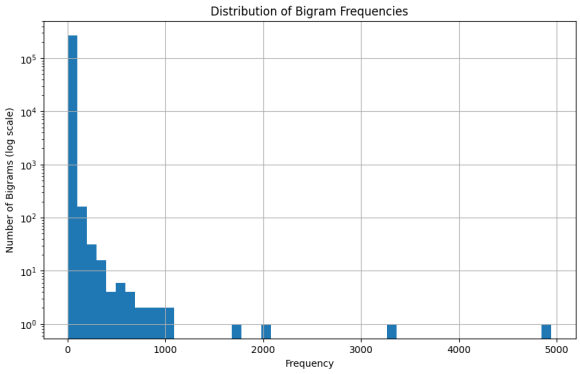

Figure 2: Distribution of Bigram Frequencies (log scale)

5 Exercise 5: Tokenize the dataset

```
tokens = text.split() # split text by spaces
# iterate through words
for i in range(len(tokens)):
    # if the word is not in vocabulary
    if tokens[i] not in vocab:
     # replace the word with the <unk> token
        tokens[i] = unknown_token
```

6 Exercise 6: Questions about the tokenization

6.1 How many unknown tokens are in the validation dataset after tokenization?

After tokenizaton there is 13081 unknown tokens are in the validation dataset, which is around 22% of all tokens.

6.2 What is the distribution of the number of tokens in the training dataset?

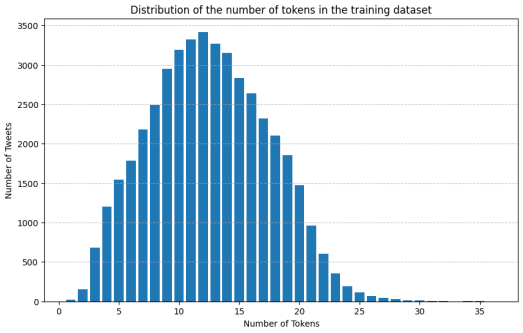

Figure 3: The distribution of the number of tokens in the training dataset

The distribution appears to follow a normal distribution centered around 12.5, as expected based on the Central Limit Theorem. Given the large number of tweets, the distribution of token counts can be approximated by a normal curve.

6.3 How the number of tokens corresponds to the number of characters in our dataset?

The total number of characters in the dataset is 7092437; the total number of tokens is 1264272. It means that the average token length in characters is 5.6.

6.4 How the size of the vocabulary affects the number of unknown tokens?

For the smaller vocabulary size, less words are in this set, so in new tweeds more word will be unrecognized (they will not be present in the vocabulary), so the number of unknown tokens will be higher. With a smaller vocabulary size, fewer words are included in the set. As a result, more words in new tweets will not be recognized (they will not be present in the vocabulary), leading to a higher number of unknown tokens.

6.5 How does the size of the vocabulary affect the number of tokens in the dataset?

The vocabulary size does not affect the number of tokens. Even if a token is not saved in the vocabulary set, it will be replaced by a unknown token <unk>, so the number of token in total will stay the same.

6.6 Think about the advantages and disadvantages of the tokenization method we used. What are the cases when it will not work well?

The tokenization method we used is straightforward to implement, as it replaces every unknown word with a special <unk> token. However, if the dataset contains many similar words (with similar letter sequences or formed through morpheme composition) this approach becomes inefficient. In such cases, a significant number of words may be classified as unknown, resulting in the loss of valuable information. A more effective strategy would be to use subword tokenization, which breaks words down into smaller, more meaningful units. This method focuses on common letter sequences that frequently appear across words, allowing for more efficient representation of the text without losing essential semantic details.

7 Exercise 7: Counting the characters

In this exercise, we built a character-level counter to analyze the frequency of each character in the cleaned training dataset. The function iterates

through the dataset and updates a Counter object with the characters.

The total number of unique characters is 552. This includes standard letters, punctuation, emojis, capital/ lowercase variants, and special symbols. The most frequent characters are common letters and whitespace:

```
[(' ', 505100), ('e', 243286), ('a', 199618),
('o', 183842), ('i', 161653),('t', 160278),
('s', 152020), ('n', 150202), ('r', 144876),
('l', 108238)]
```

Characters like punctuation and special symbols also appear frequently, reflecting the expressive language used typical of social media.

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

The method of calculating the statistics of character pairs is fairly simple.

```
for word, freq in corpus.items():

    chars = word.split()
    pairs = zip(chars, chars[1:])
    for pair in pairs:
        stats.update([pair] * freq)
```

While iterating though the corpus, each word is being split in characters. Then, using zip() method an array of pairs of adjacent characters is being created. Finally, while iterating through the array of pairs, each instance of the array of pairs is being added to the stats Counter() with corresponding frequencies (calculated by multiplying each pair by the frequency variable freq, that corresponds to the word that is being observed), that is returned in the end of the initial loop.

9 Exercise 9: BPE algorithm

BPE algorithm uses previous pieces of code. As input 3 parameters are taken: vocab - vocabulary, tokens that are induced + initial tokens, corpus - relevant corpus, num_merges - number of merges or number of iterations of BPE algorithm

```
most_common_pair = calculate_bpe
_corpus_stats(corpus).most_common(1)[0][0]
vocab.append("".join(most_common_pair))
corpus = merge_corpus(corpus,
most_common_pair)
merges.append(most_common_pair)
```

While iterating through the number of merges, the algorithm takes the most common pair of characters in the corpus, updates vocabulary with the merged pair, updates the corpus and adds the pair that was being merge to the array of merges, to track the history of merges

10 Exercise 10: Comparing tokenizers

10.1 What are the differences?

BPE algorithm constructs its vocabulary by iteratively merging the most frequent character sequences. This results in tokens that can represent subword units, including common prefixes and suffixes. In contrast, the previously used tokenizer treats entire words as individual tokens, storing whole words in the vocabulary.

10.2 Compare the number of tokens created by your tokenizers.

Number of tokens: 1st tokenizer = 1232674 BPE tokenizer = 4385033 As expected, the BPE tokenizer produces a larger number of tokens. This is because it includes not only individual characters and letters but also frequently occurring subword units. The first tokenizer, on the other hand, only includes complete words, resulting in a smaller vocabulary size.

10.3 Calculate the number of ‘<unk>‘ tokens in the validation dataset for each tokenizer.

Number of unknown tokens in validation dataset: 1st tokenizer = 10106 BPE tokenizer = 95 As anticipated, the BPE tokenizer yields significantly fewer unknown tokens. This is due to its ability to represent previously unseen words as combinations of known subword units, increasing its ability to generalize compared to the word-level tokenizer.

10.4 Compare the average length in tokens between different tokenizers.

The average length of tokens: 1st tokenizer = 5.75 BPE tokenizer = 1.62 The average token length is shorter for the BPE tokenizer because it segments text into smaller units (characters and subwords), whereas the word-level tokenizer processes larger units—entire words—resulting in longer token sequences per sentence or document.

10.5 What are the advantages and disadvantages of the BPE tokenizer?

One of the main advantages of Byte Pair Encoding is that it efficiently handles rare, unknown words and morphological variations of words that have already been saved by breaking them into smaller, known pieces. The mode is more general, because it captures common word parts like prefixes and suffixes, rather then whole longer words. It also reduced vocabulary size compared to full word-level models with similar coverage. However, the computational complexity of a language model is higher (especially for larger vocabularies). It may also be not efficient in languages with complex morphology or those lacking clear word boundaries and struggle with splitting idioms and colloquialisms into subwords.

References

A Collaborators outside our group

B Use of genAI

In completing this work, we used the content provided on Canvas as our primary resource. We also used ChatGPT to assist with clarifying concepts and tasks, resolving technical issues, enhancing the clarity of plots, and correcting grammatical and spelling errors.

Prompts used include:

- "Fix grammar and spelling mistakes in the following text:"
- "Make this plot more clear:"
- "Explain the error message:"
- "What does "number of tokens" in the following sentence mean: The distribution of the number of tokens in the training dataset"
- "How does the counter for characters and strings differ?"