NLP 2025

Lab 1: Tokenization

Tokenization is a fundamental step in **Natural Language Processing (NLP)** 🧠💬 that transforms raw text into structured data for computational models. In this lab, you will explore different **tokenization techniques** 📝, preprocess text data 🔍, and implement **tokenization pipelines** using popular NLP libraries 🏗️.

You will also gain **hands-on experience** with **Hugging Face Datasets** 🤗📚, while assessing the impact of tokenization choices on downstream NLP tasks.

By the end of this lab, you will have a **strong foundation** in tokenization techniques and be able to apply them effectively in **real-world NLP applications** 🌍.

🎯 Learning Goals

By the end of this lab, you should be able to:

✅ **Understand the role of tokenization in NLP** 🧠💡
✅ **Explain why tokenization is important** and how it affects text processing 📖🔍
✅ **Implement different tokenization techniques** – Apply **word** 📝, **subword** 🔢, and **character-level** 🔡 tokenization using built-in libraries.
✅ **Use Hugging Face Datasets** 🤗📊 – Load and preprocess text datasets efficiently.
✅ **Evaluate tokenization impact** 📉🔎 – Analyze how different tokenization methods influence model performance.
✅ **Identify challenges in tokenization** ❗🔍 – Recognize issues like **out-of-vocabulary (OOV) words**, **ambiguity**, and **multilingual tokenization** 🌍.

Score breakdown

Exercise	Points
Exercise 1	5
Exercise 2	6
Exercise 3	5
Exercise 4	12
Exercise 5	5
Exercise 6	22

Exercise	Points
Exercise 7	5
Exercise 8	5
Exercise 9	10
Exercise 10	25
Total	100

This score will be scaled down to 0.5 and that will be your final lab score.

📌 Instructions for Delivery (📅 Deadline: 11/Apr 18:00, 🎭 *wildcards possible*)

✅ Submission Requirements

- 📄 You need to submit a **PDF of your report** (use the templates provided in **LaTeX** 🖋️ (*preferred*) or **Word** 📄) and a **copy of your notebook** 📓 with the code.
- ⚡ Make sure that **all cells are executed properly** ⚙️ and that **all figures/results/plots** 📊 you include in the report are also visible in your **executed notebook**.

✅ Collaboration & Integrity

- 🕵️ While you may **discuss** the lab with others, you must **write your solutions with your group only**. If you **discuss specific tasks** with others, please **include their names** in the appendix of the report.
- 📜 **Honor Code applies** to this lab. For more details, check **Syllabus §7.2** ⚖️.
- 📢 **Mandatory Disclosure**:
  - Any **websites** 🌐 (e.g., **Stack Overflow** 💡) or **other resources** used must be **listed and disclosed**.
  - Any **GenAI tools** 🤖 (e.g., **ChatGPT**) used must be **explicitly mentioned**.
  - 🚨 **Failure to disclose these resources is a violation of academic integrity**. See **Syllabus §7.3** for details.

Preparation

In [39]:
```
# ! pip install -U datasets~=3.2.0
# ! python -m pip install -U matplotlib
```

In [40]:
```
import re
from collections import Counter
import numpy as np
import datasets
import matplotlib.pyplot as plt
```

```
import pandas as pd
import tqdm
```

0. Intro to regular expressions

In this introduction section, you can practice the use of regular expressions in python. You can find the documentation here: https://docs.python.org/3/library/re.html. The main functions of the re module are:

- `re.search()` - searches for a pattern in a string, returns the first match,
- `re.findall()` - similar to `search()`, but returns a list of all matches,
- `re.sub()` - replaces the matches with a string.

All above functions accept the regular expression pattern as their argument. The patterns are strings that represent the rules for matching the text. In python they start with `r` character, e.g. `r'\d'` is a pattern that matches a digit.

Let us start with a simple example. We will search for the word "world" in the string "Hello, world!".

In [41]:
```
text = "Hello, world!, world im the worlds king of the world"
pattern = r'world'
match = re.search(pattern, text)
print(match)
```

```
<re.Match object; span=(7, 12), match='world'>
```

The `search()` function returns a match object that tells us where the match was found (`span` argument) and the exact part of the string that matched the pattern (`group` argument).

Below you can find the examples from the lecture.

In [42]:
```
# Disjunctions
pattern = r'[wW]oodchuck' # matches both "woodchuck" and "Woodchuck"
pattern = r'[1234567890]' # matches any digit
pattern = r'[0-9]' # matches any digit
pattern = r'[A-Z]' # matches any uppercase letter
pattern = r'[a-z]' # matches any lowercase letter
pattern = r'[A-Za-z]' # matches any letter

# Disjunctions with pipe |
pattern = r'groundhog|Woodchuck' # matches both "woodchuck" and "Woodchuck"

# Negation (only when in [])
pattern = r'[^0-9]' # matches any character that is not a digit
pattern = r'[^Ss]' # matches any character that is not 'S' or 's'
pattern = r'a^b' # matches the string "a^b"

# Quantifiers (+, *, ?, .)
pattern = r'baa+' # matches "ba" followed by one or more "a" (e.g. "baa", "baaa", "
```

```
pattern = r'oo*h' # matches "o" followed by zero or more "o" and then "h" (e.g. "oh
pattern = r'colou?r' # matches "color" and "colour"
pattern = r'beg.n' # matches "begun", "begin", "begnn", ...

# Anchors (^, $)
pattern = r'^Hello' # matches "Hello" at the beginning of the string
pattern = r'world!$' # matches "world!" at the end of the string
```

1. Huggingface datasets

For this lab, we will use the **Hugging Face Datasets** library (Hugging Face Datasets), which provides an extensive collection of ready-to-use NLP datasets. The library is designed to be lightweight, efficient, and compatible with popular deep learning frameworks such as PyTorch and TensorFlow.

You can find the full documentation and tutorials here:
📌 Hugging Face Datasets Documentation

Why use Hugging Face Datasets?

- **Easy Access:** Load datasets with a single command without manual downloads.
- **Standardized Format:** Datasets come in a unified structure, making them easy to preprocess and integrate into ML pipelines.
- **Large Collection:** Provides datasets for a wide range of NLP tasks, including classification, translation, summarization, and more.
- **Seamless Integration:** Works with `transformers` and `sklearn` for preprocessing and model training.

Dataset for this lab: TweetEval - Emoji Subset

In this lab, we will work with the **TweetEval** dataset, specifically the **emoji** subset. The TweetEval dataset is a benchmark for evaluating NLP models on Twitter-related tasks, covering tasks such as sentiment analysis, hate speech detection, and irony detection.

For tokenization, we will focus only on the **text** (the content of the tweets), but we will also examine the **labels** to understand the dataset structure.

📎 The dataset description and details are available in its dataset card: **TweetEval Dataset**

💡 Exploring More Datasets Hugging Face provides a vast selection of datasets across different NLP tasks. You can browse and explore more at: 📎 Hugging Face Datasets Collection

In [43]:
```
tweet_ds = datasets.load_dataset('tweet_eval', 'emoji')
print(tweet_ds)
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 45000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 50000
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 5000
    })
})
```

The loaded dataset contains three subsets (“train”, “validation”, and “test”). Each consists of two columns: “text” and “label”. Label is an integer from 0 to 19 representing an emoji. See the dataset's card for more information. We can access the elements of the dataset like so:

In [44]:
```
for i in range(10):
    print(tweet_ds['train'][i])
```

```
{'text': 'Sunday afternoon walking through Venice in the sun with @user    @ Abbot K
inney, Venice', 'label': 12}
{'text': "Time for some BBQ and whiskey libations. Chomp, belch, chomp! (@ Lucille's
Smokehouse Bar-B-Que)", 'label': 19}
{'text': 'Love love love all these people    #friends #bff #celebrate #blessed #sund
ayfunday @ San…', 'label': 0}
{'text': '    @ Toys"R"Us', 'label': 0}
{'text': 'Man these are the funniest kids ever!! That face! #HappyBirthdayBubb @ FLI
PnOUT Xtreme', 'label': 2}
{'text': '#sandiego @ San Diego, California', 'label': 11}
{'text': 'My little      #ObsessedWithMyDog @ Cafe Solstice Capitol Hill', 'label':
0}
{'text': 'More #tinyepic things #tinyepicwestern, this one is crazy @user I may be o
ne of your…', 'label': 19}
{'text': 'Last night  @ Omnia Night Club At Caesars Palace', 'label': 0}
{'text': 'friendship at its finest. ....#pixar #toystory #buzz #woody #friends #frie
ndship #bff…', 'label': 7}
```

You can easily cast the dataset to the pandas DataFrame.

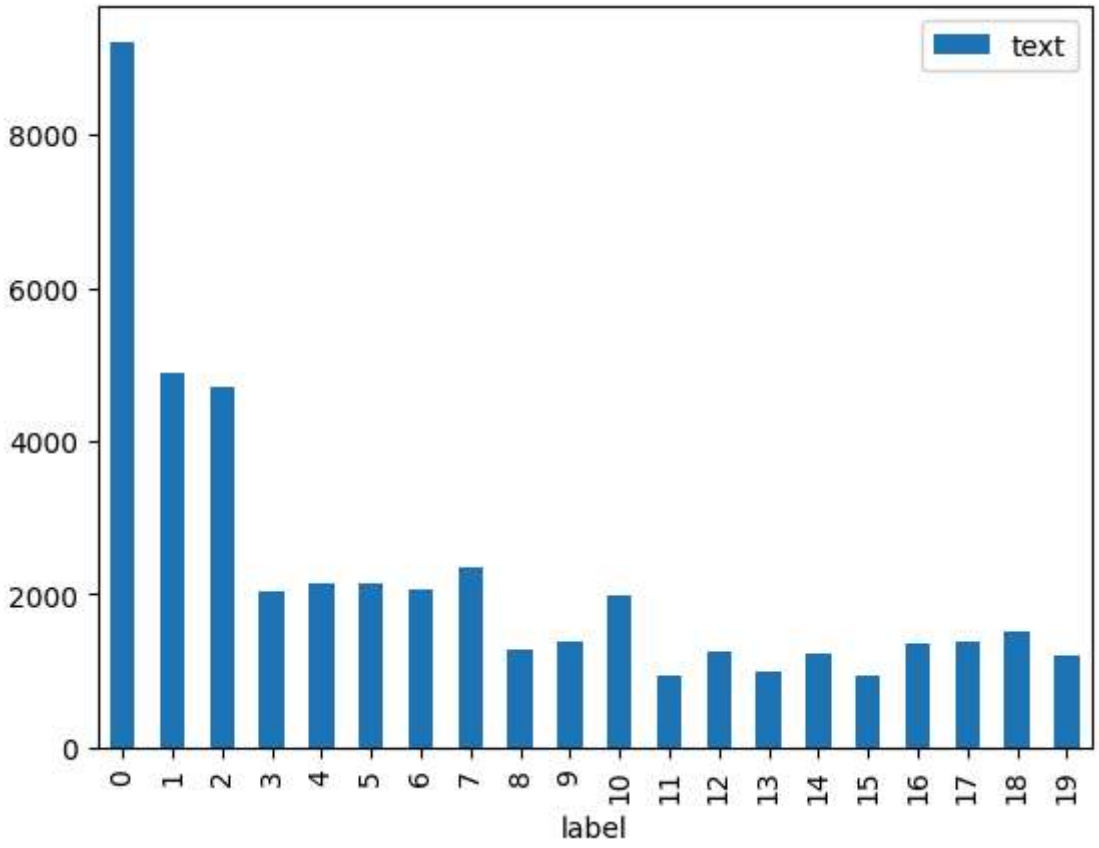
In [45]:
```
tweet_train_df = pd.DataFrame(tweet_ds['train'])
print(tweet_train_df)
```

```
                                                    text  label
0      Sunday afternoon walking through Venice in the...     12
1      Time for some BBQ and whiskey libations. Chomp...     19
2      Love love love all these people    #friends...      0
3                                        @ Toys"R"Us      0
4      Man these are the funniest kids ever!! That fa...      2
...                                                  ...    ...
44995  Here to celebrate the Nunez wedding! Love my b...      0
44996  1 night in Paris.... Wait... @ Paris Las Vegas...      1
44997  Be safe this weekend everyone. #happylaborday ...     11
44998          Pizza (@ Five50 - @user in Las Vegas, NV)      1
44999  my mini is perfect, no one deserves her @ Las ...     13

[45000 rows x 2 columns]
```

We can plot the distribution of the labels in the training subset.

In [46]:
```
tweet_train_df.groupby('label').count().plot.bar()
```

Out[46]: <Axes: xlabel='label'>



Dataset's filter function

We can filter the examples using `filter()` method. See this link for more details https://huggingface.co/docs/datasets/en/use_dataset. Here is an example of filtering the short tweets (less than 20 characters) from the `train` subset.

In [47]:
```
short_tweets = tweet_ds['train'].filter(lambda example: len(example['text']) < 20)
print(short_tweets)
```

```
Dataset({
    features: ['text', 'label'],
    num_rows: 506
})
```

In [48]:
```
for i in range(10):
    print(short_tweets[i])
```

```
{'text': '    @ Toys"R"Us', 'label': 0}
{'text': ' @ Columbia River', 'label': 12}
{'text': 'My weekend: @user 3', 'label': 6}
{'text': 'good day today', 'label': 3}
{'text': 'My last RT...', 'label': 2}
{'text': '@ On Lake Cowichan', 'label': 13}
{'text': '@ Macroplaza', 'label': 4}
{'text': '@ BART Train', 'label': 6}
{'text': ' 4 a tbh&amp;amp; rate', 'label': 0}
{'text': '@user Oh nice!!', 'label': 14}
```

Dataset's map function

Datasets library contains a very useful method map. It expects a function that will receive an example from the dataset. This function will be applied to all entries. We will calculate the length of the text (in characters) in each example.

In [49]:
```
def calculate_text_length(example):
    example['text_length'] = len(example['text'])
    return example
```

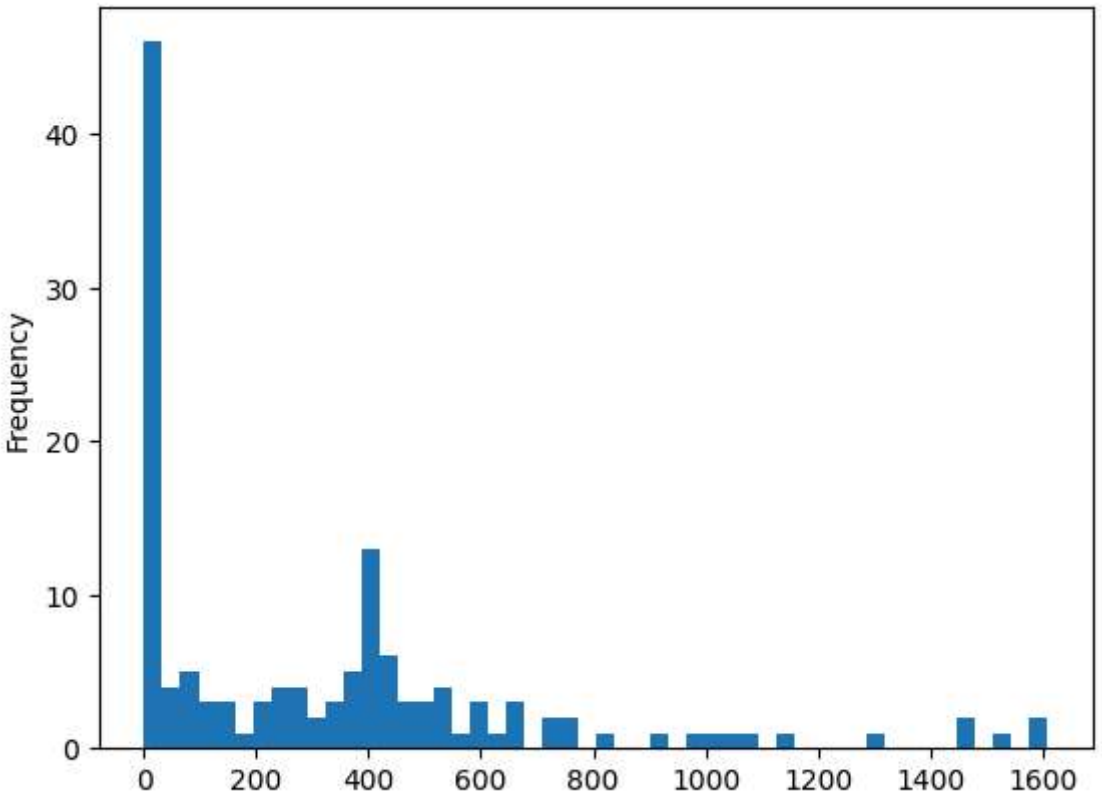
In [50]:
```
tweet_ds = tweet_ds.map(calculate_text_length)
print(tweet_ds)
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label', 'text_length'],
        num_rows: 45000
    })
    test: Dataset({
        features: ['text', 'label', 'text_length'],
        num_rows: 50000
    })
    validation: Dataset({
        features: ['text', 'label', 'text_length'],
        num_rows: 5000
    })
})
```

We can plot the histogram of the text lengths.

In [51]:
```
pd.DataFrame(tweet_ds['train']).groupby('text_length')['text_length'].count().plot.
```

Out[51]: <Axes: ylabel='Frequency'>



Exercise 1: Questions about the datasets

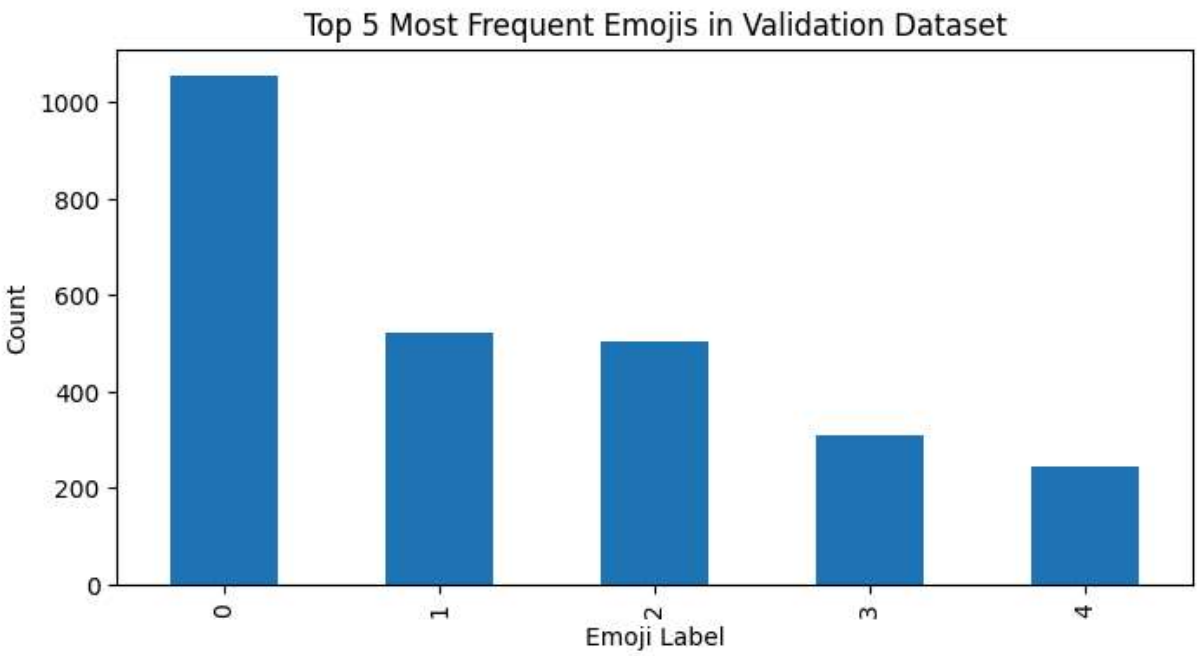
1. (1p) What is the size of the training, test and validation datasets?
2. (1p) What are the top 5 most frequent emojis in the validation dataset?
3. (1p) Compare the distributions of labels (emojis) between training and validation datasets.
4. (1p) How many examples with the "fire" emoji are in the training dataset?
5. (1p) What is the average length (in characters) of the tweets in the training dataset?

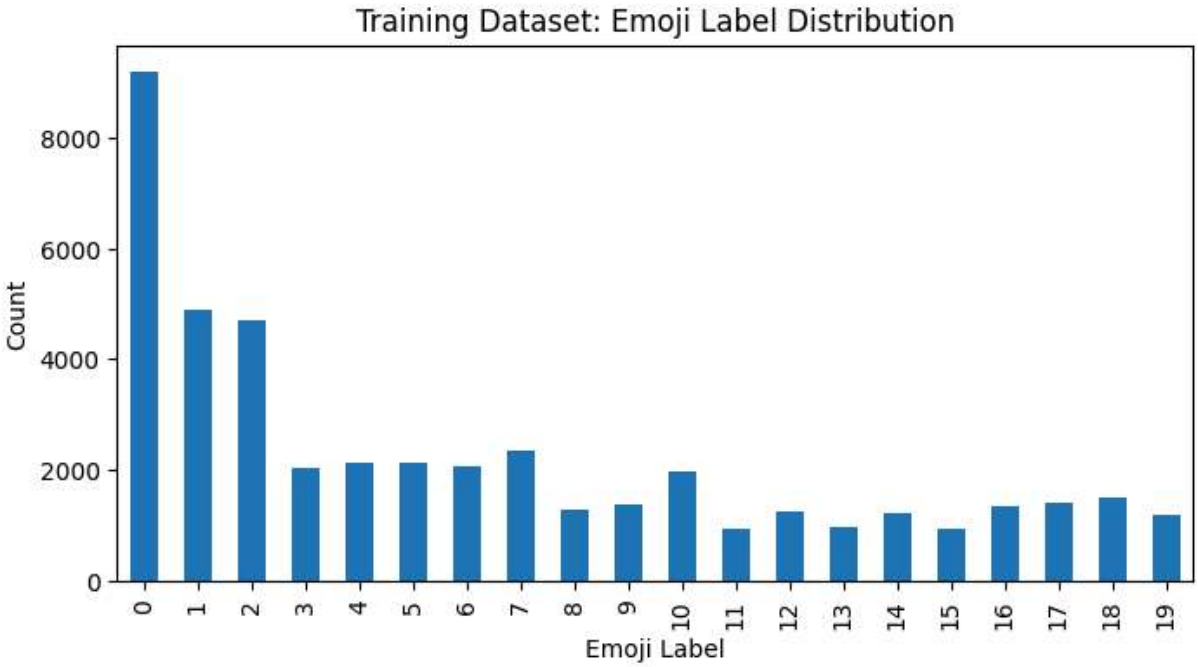
You can add cells here to answer the questions

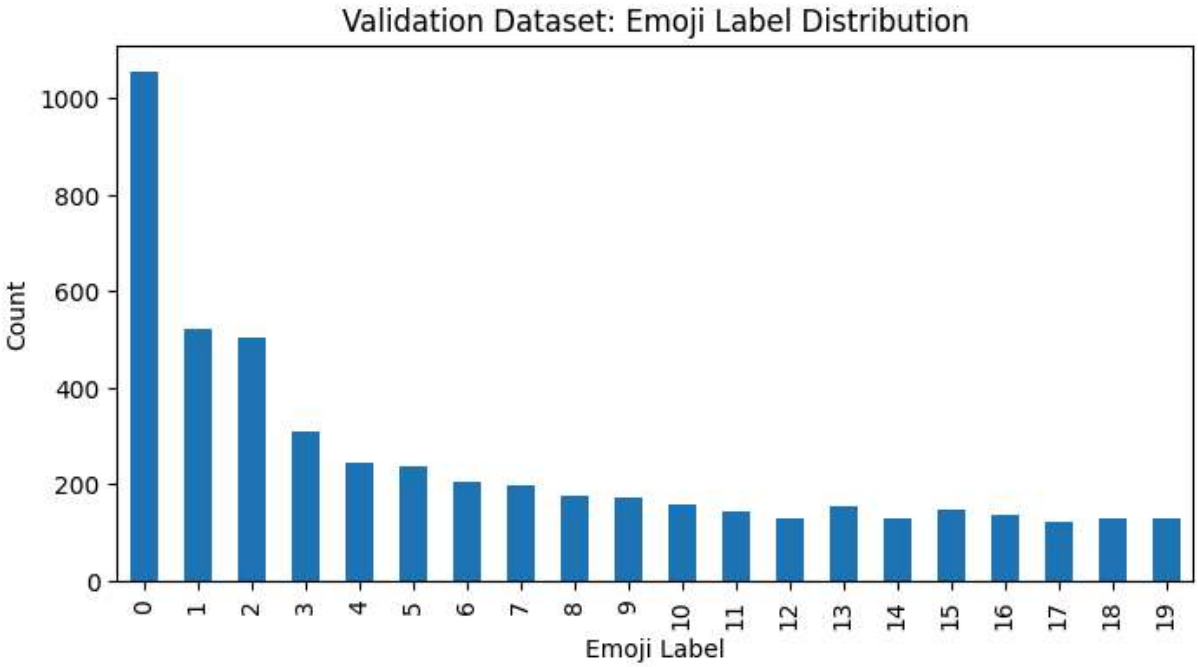
In [52]:
```
s_train = len(tweet_ds['train'])
s_val = len(tweet_ds['validation'])
s_test = len(tweet_ds['test'])

print("Training set size:", s_train)
print("Validation set size:", s_val)
print("Test set size:", s_test)
print()
#2

tweet_val_df = pd.DataFrame(tweet_ds['validation'])
top5_val = tweet_val_df['label'].value_counts().head(5)
top5_val.sort_index().plot.bar(figsize=(8, 4), title="Top 5 Most Frequent Emojis in
plt.xlabel("Emoji Label")
```

```
plt.ylabel("Count")
plt.show()

#3
train_counts = tweet_train_df['label'].value_counts().sort_index()
val_counts = tweet_val_df['label'].value_counts().sort_index()

plt.figure(figsize=(8, 4))
train_counts.plot(kind='bar', title='Training Dataset: Emoji Label Distribution')
plt.xlabel('Emoji Label')
plt.ylabel('Count')
plt.show()
plt.figure(figsize=(8, 4))
val_counts.plot(kind='bar', title='Validation Dataset: Emoji Label Distribution')
plt.xlabel('Emoji Label')
plt.ylabel('Count')
plt.show()

#4
fire_count = sum(1 for ex in tweet_ds['train'] if ex['label'] == 4)
print("Number of examples with the 'fire' emoji (label 4) in the training dataset:"
#5

avg_length = sum(len(ex['text']) for ex in tweet_ds['train']) / len(tweet_ds['train
print("Average tweet length in the training dataset (in characters):", avg_length)
```

```
Training set size: 45000
Validation set size: 5000
Test set size: 50000
```








```
Number of examples with the 'fire' emoji (label 4) in the training dataset: 2146
Average tweet length in the training dataset (in characters): 71.01691111111111
```

2. Tokenization

In this section we will preprocess the dataset by cleaning and tokenizing the entries. Datasets library contains a very useful method map. It expects a function that will receive an example from the dataset. This function will be applied to all entries.

2.1 Cleaning the text

Exercise 2: Write the text cleaning function

Include at least the following steps:

- (1p) remove comma between numbers, i.e. 15,000 -> 15000
- (1p) remove multiple spaces
- (1p) space out the punctuation (i.e. "hello, world." -> "hello , world .")
- (3x1p) three more cleaning steps of your choice

In [53]:
```
def clean(example):
    """
    Cleans the example from the Dataset
    Args:
        example: an example from the Dataset

    Returns: update example containing 'clean' column

    """
    text = example['text']

    # Empty text
    if text == '':
        example['clean'] = ''
        return example

    # 'text' from the example can be of type numpy.str_, let's convert it to a pyth
    text = str(text)

    ### YOUR CODE HERE

    # 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)
    # three more cleaning steps of your choice
    text = text.lower()
    text = re.sub(r'([\(\)])', r' \1 ', text)
    text = re.sub(r'\s+', ' ', text)

    ### YOUR CODE ENDS HERE

    # Update the example with the cleaned text
    example['clean'] = text.strip()
    return example
```

This is an example of applying the `clean()` function you just wrote to a single entry of the dataset. The function added a 'clean' field to the example.

In [54]:
```
print('Original tweet item:')
print(tweet_ds['train'][1]['text'])
```

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

```
Original tweet item:
Time for some BBQ and whiskey libations. Chomp, belch, chomp! (@ Lucille's Smokehous
e Bar-B-Que)
Cleaned tweet item:
time for some bbq and whiskey libations . chomp , belch , chomp ! ( @ lucille's smok
ehouse bar-b-que )
```

Let's finally use the `map()` method and apply your `clean()` function to all entries of the dataset. You can see that the `clean` column has been added to each split.

Below, we will apply your function to all entries in the dataset.

In [55]:
```
tweet_ds = tweet_ds.map(clean)
print(tweet_ds)
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label', 'text_length', 'clean'],
        num_rows: 45000
    })
    test: Dataset({
        features: ['text', 'label', 'text_length', 'clean'],
        num_rows: 50000
    })
    validation: Dataset({
        features: ['text', 'label', 'text_length', 'clean'],
        num_rows: 5000
    })
})
```

2.2 Build vocabulary

In the previous section, we implemented the cleaning of the dataset. Now, we will tokenize the text splitting it by spaces. We will build a vocabulary based on the cleaned text of the `train` split. We will investigate some properties of corpora (e.g. Zipf's law).

The function below builds a vocabulary from the dataset. It counts the occurrences of the words in the dataset using the Counter class. Check the documentation here collections.Counter.

Exercise 3: Build the vocabulary

(5p) Fill in the function below to build the vocabulary from the dataset. The function should return a `Counter` object with the words and their frequencies. The variable named `vocab` is already initialized as an empty `Counter` object.

In [56]:
```
def build_vocab_counter(dataset):
    """
```

```
    Builds a vocabulary from the dataset
    Args:
        dataset: a dataset

    Returns: a vocabulary

    """
    vocab = Counter()

    ### YOUR CODE HERE
    for example in dataset:
        words = example['clean'].split()

        vocab.update(words)

    return vocab

    ### YOUR CODE ENDS HERE
    return vocab
```

In [57]:
```
vocab_counter = build_vocab_counter(tweet_ds['train'])
print('Size of the vocabulary:', len(vocab_counter))
```

```
Size of the vocabulary: 68297
```

Because we created a counter, we can easily check the most and least common words in the vocabulary. Do the most common words make sense? How about the least common ones?

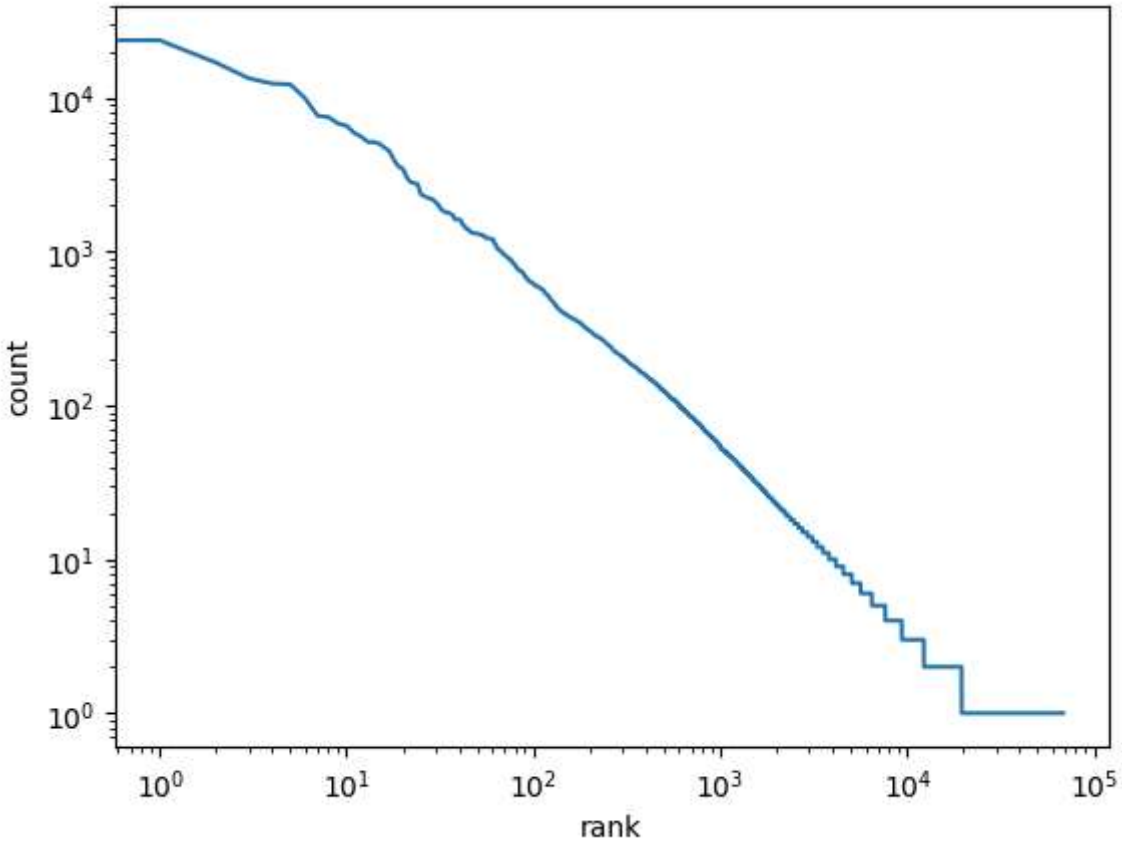
In [58]:
```
print('Most common:')
print(vocab_counter.most_common(10))
print('Least common:')
print(vocab_counter.most_common()[-10:])
```

```
Most common:
[('@', 24206), ('.', 23674), ('!', 16957), ('the', 13403), (',', 12384), ('@user', 1
2236), ('', 9936), ('to', 7653), ('my', 7495), ('a', 6814)]
Least common:
[('#southbayla', 1), ('thedabberchick', 1), ('nector', 1), ('chefking1921express',
1), ('#alabama', 1), ('#rolltide', 1), ('#bffweekend', 1), ('nunez', 1), ('#happylab
orday', 1), ('five50', 1)]
```

We can also plot the counts of the words. You can check the Power law if you are more interested.

In [59]:
```
import matplotlib.pyplot as plt

plt.loglog([val for word, val in vocab_counter.most_common()])
plt.xlabel('rank')
plt.ylabel('count')
```

Out[59]:
```
Text(0, 0.5, 'count')
```



The plot shows that the distribution of the words in the vocabulary follows the Zipf's law. The most frequent word occurs approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc.

We can also filter the vocabulary by the frequency of the words. We will only consider the most frequent words and mark the rest as the `<unk>` token. Here we set the maximum vocabulary size to 10,000. But in the later steps, you will experiment with different sizes.

In [60]:
```
max_vocab_size = 10000
vocab = vocab_counter.most_common(max_vocab_size)
# cast to list of words
vocab = [word for word, _ in vocab]
print(len(vocab))
```

```
10000
```

Exercise 4: Frequency of pairs of words (bigrams)

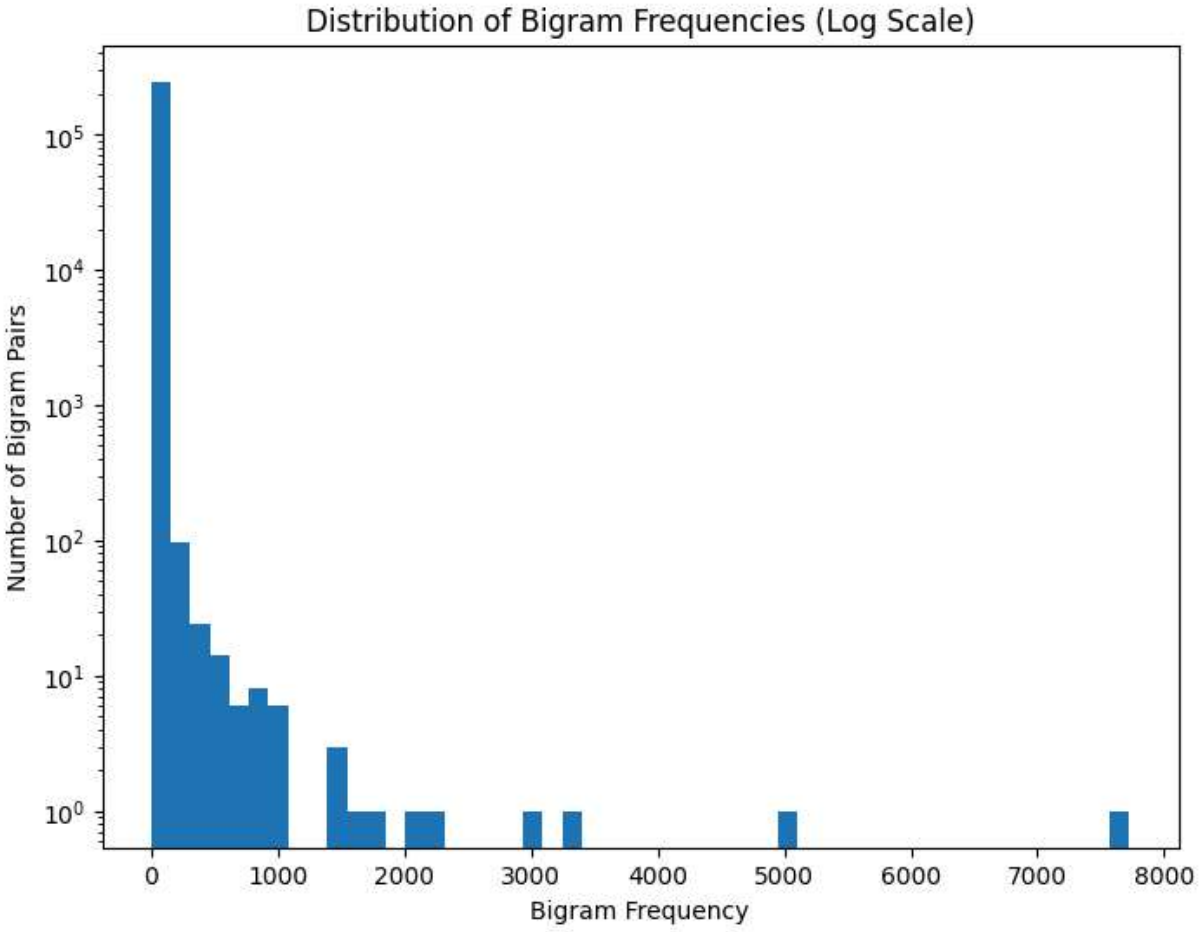
Calculate the frequency of (neighbouring) pairs of words in the training dataset.

- (5p) List the most and least common pairs. Do the most common pairs make sense?
- (2p) How many pairs occur only once in the dataset?
- (5p) Plot the distribution of the pair frequencies.

In [61]:
```
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:]))

bigram_counter = Counter()
for example in tweet_ds['train']:
    bigrams = get_bigrams(example['clean'])
    bigram_counter.update(bigrams)

# Print the top 10 most common bigrams
print("Most common bigrams (top 10):")
for bigram, freq in bigram_counter.most_common(10):
    print(bigram, ":", freq)

least_common = sorted(bigram_counter.items(), key=lambda x: x[1])[:10]
print("\nLeast common bigrams (10 examples):")
for bigram, freq in least_common:
    print(bigram, ":", freq)

#2
bigram_once_count = list(bigram_counter.values()).count(1)
print("\nNumber of bigrams that occur only once:", bigram_once_count)
#3

freqs = list(bigram_counter.values())

plt.figure(figsize=(8, 6))
plt.hist(freqs, bins=50)
plt.xlabel("Bigram Frequency")
plt.ylabel("Number of Bigram Pairs")
plt.title("Distribution of Bigram Frequencies (Log Scale)")
plt.yscale('log')
plt.show()
```

```
Most common bigrams (top 10):
('.', '.') : 7726
('!', '!') : 4946
(',', 'california') : 3339
('', '@') : 2966
('', '') : 2240
('&amp', ';') : 2019
('los', 'angeles') : 1722
('@', 'the') : 1621
('@user', '@') : 1467
('.', '@') : 1465

Least common bigrams (10 examples):
('through', 'venice') : 1
('some', 'bbq') : 1
('and', 'whiskey') : 1
('whiskey', 'libations') : 1
('libations', '.') : 1
('.', 'chomp') : 1
('chomp', ',') : 1
(',', 'belch') : 1
('belch', ',') : 1
(',', 'chomp') : 1

Number of bigrams that occur only once: 200730
```



2.3 Tokenize the dataset

The function below tokenizes the cleaned text (`example['clean']`) by splitting it on spaces. It replaces the words that are not in the vocabulary with the `<unk>` token.

Exercise 5: Tokenize the dataset

(5p) Fill in the function below to tokenize the dataset. The function will be applied to the dataset through the `map()` method, so it returns the updated example. Your task is to split the text by spaces and replace the words that are not in the vocabulary with the `<unk>` token.

In [62]:
```python
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

    """
    text = example['clean']
    tokens = None # list of tokens, your code should fill this variable

    ### YOUR CODE HERE
    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]



    ### YOUR CODE ENDS HERE

    example['tokens'] = tokens
    return example
```

In [63]:
```python
tweet_ds = tweet_ds.map(tokenize, fn_kwargs={'vocab': vocab})
print(tweet_ds)
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label', 'text_length', 'clean', 'tokens'],
        num_rows: 45000
    })
    test: Dataset({
        features: ['text', 'label', 'text_length', 'clean', 'tokens'],
        num_rows: 50000
    })
    validation: Dataset({
        features: ['text', 'label', 'text_length', 'clean', 'tokens'],
        num_rows: 5000
    })
})
```

Let us examine several entries from the dataset. We can see that the `tokens` column has been added to each example.

In [64]:
```
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, Ven
ice
Tokenized tweet:
['sunday', 'afternoon', 'walking', 'through', 'venice', 'in', 'the', 'sun', 'with',
'@user', '', '', '', '@', 'abbot', 'kinney', ',', 'venice']
Original tweet:
Time for some BBQ and whiskey libations. Chomp, belch, chomp! (@ Lucille's Smokehous
e Bar-B-Que)
Tokenized tweet:
['time', 'for', 'some', 'bbq', 'and', 'whiskey', 'libations', '.', '<unk>', ',', '<u
nk>', ',', '<unk>', '!', '(', '@', "lucille's", 'smokehouse', 'bar-b-que', ')']
Original tweet:
Love love love all these people    #friends #bff #celebrate #blessed #sundayfunday @
San…
Tokenized tweet:
['love', 'love', 'love', 'all', 'these', 'people', '', '', '', '#friends', '#bff',
'#celebrate', '#blessed', '#sundayfunday', '@', 'san…']
Original tweet:
    @ Toys"R"Us
Tokenized tweet:
['', '', '', '', '@', '<unk>']
Original tweet:
Man these are the funniest kids ever!! That face! #HappyBirthdayBubb @ FLIPnOUT Xtre
me
Tokenized tweet:
['man', 'these', 'are', 'the', 'funniest', 'kids', 'ever', '!', '!', 'that', 'face',
'!', '<unk>', '@', '<unk>', '<unk>']
Original tweet:
#sandiego @ San Diego, California
Tokenized tweet:
['#sandiego', '@', 'san', 'diego', ',', 'california']
Original tweet:
My little      #ObsessedWithMyDog @ Cafe Solstice Capitol Hill
Tokenized tweet:
['my', 'little', '', '', '', '', '', '<unk>', '@', 'cafe', 'solstice', 'capitol', 'h
ill']
Original tweet:
More #tinyepic things #tinyepicwestern, this one is crazy @user I may be one of you
r…
Tokenized tweet:
['more', '<unk>', 'things', '<unk>', ',', 'this', 'one', 'is', 'crazy', '@user',
'i', 'may', 'be', 'one', 'of', 'your…']
Original tweet:
Last night  @ Omnia Night Club At Caesars Palace
Tokenized tweet:
['last', 'night', '', '@', 'omnia', 'night', 'club', 'at', 'caesars', 'palace']
Original tweet:
friendship at its finest. ....#pixar #toystory #buzz #woody #friends #friendship #bf
f…
Tokenized tweet:
['friendship', 'at', 'its', 'finest', '.', '.', '.', '.', '.', '<unk>', '<unk>', '#b
uzz', '<unk>', '#friends', '#friendship', '#bff…']
```

Make sure that the tokenization works as you intended. If not, revisit the cleaning and tokenization functions.

Exercise 6: Questions about the tokenization

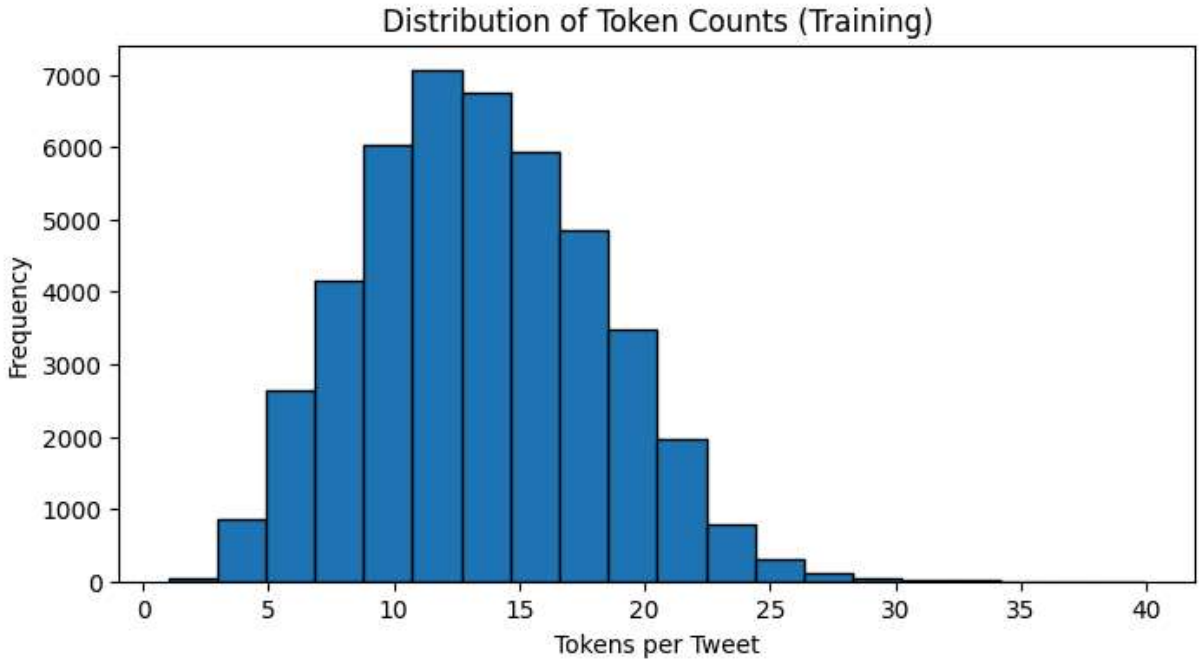
1. (3p) How many unknown tokens are in the validation dataset after tokenization?
2. (3p) What is the distribution of the number of tokens in the training dataset?
3. (4p) How the number of tokens corresponds to the number of characters in our dataset?
4. (4p) How the size of the vocabulary (`max_vocab_size`) affects the number of unknown tokens?
5. (4p) How does the size of the vocabulary affect the number of tokens in the dataset?
6. (4p) Think about the advantages and disadvantages of the tokenization method we used. What are the cases when it will not work well?

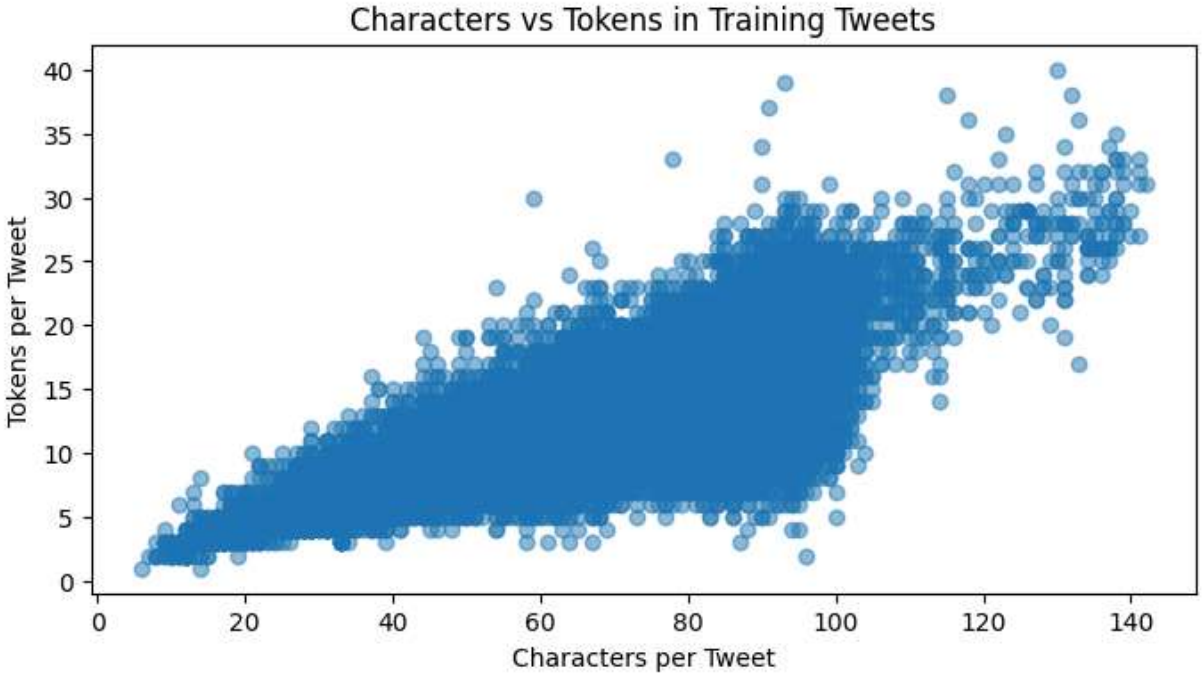
For answering these questions make sure to include a proper mix of numbers/plots/tables etc. and comments.

In [65]:
```
# 1. Count unknown tokens (<unk>) in the validation dataset.
unknown_token = '<unk>'
unknown_count = sum(token == unknown_token
                    for example in tweet_ds['validation']
                    for token in example['tokens'])
print("1. Unknown tokens in validation:", unknown_count)

# 2. Distribution of token counts per tweet in the training dataset.
train_token_counts = [len(example['tokens']) for example in tweet_ds['train']]
plt.figure(figsize=(8, 4))
plt.hist(train_token_counts, bins=20, edgecolor='black')
plt.xlabel("Tokens per Tweet")
plt.ylabel("Frequency")
plt.title("Distribution of Token Counts (Training)")
plt.show()

# 3. Relationship between tweet length in characters and number of tokens.
char_counts = [len(example['text']) for example in tweet_ds['train']]
plt.figure(figsize=(8, 4))
plt.scatter(char_counts, train_token_counts, alpha=0.5)
plt.xlabel("Characters per Tweet")
plt.ylabel("Tokens per Tweet")
plt.title("Characters vs Tokens in Training Tweets")
plt.show()
correlation = np.corrcoef(char_counts, train_token_counts)[0, 1]
print("3. Correlation between characters and tokens:", correlation)

# 4. How vocabulary size affects the number of unknown tokens.
# We test for different vocabulary sizes.
vocab_sizes = [5000, 10000, 15000]
unk_counts = []

for vocab_size in vocab_sizes:
    # Build vocabulary from training data (based on cleaned text).
    counter = build_vocab_counter(tweet_ds['train'])
    vocab_current = [word for word, _ in counter.most_common(vocab_size)]
```

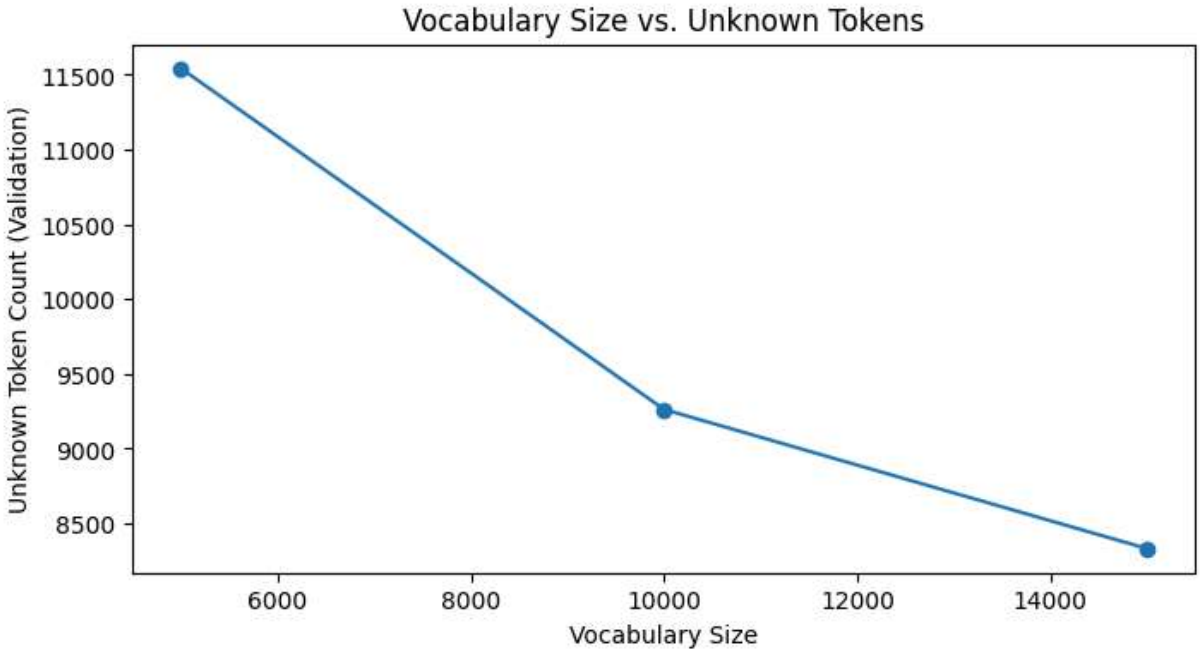
```
    # Re-tokenize the validation set using the current vocabulary.
    current_unk_count = 0
    for ex in tweet_ds['validation']:
        words = ex['clean'].split()
        # Replace words not in the vocabulary.
        tokens = [word if word in vocab_current else unknown_token for word in word
        current_unk_count += tokens.count(unknown_token)

    unk_counts.append(current_unk_count)

plt.figure(figsize=(8, 4))
plt.plot(vocab_sizes, unk_counts, marker='o')
plt.xlabel("Vocabulary Size")
plt.ylabel("Unknown Token Count (Validation)")
plt.title("Vocabulary Size vs. Unknown Tokens")
plt.show()

# 5. How vocabulary size affects the total number of tokens in the training dataset
total_tokens = []
for vocab_size in vocab_sizes:
    counter = build_vocab_counter(tweet_ds['train'])
    vocab_current = [word for word, _ in counter.most_common(vocab_size)]

    token_sum = 0
    for ex in tweet_ds['train']:
        words = ex['clean'].split()
        tokens = [word if word in vocab_current else unknown_token for word in word
        token_sum += len(tokens)
    total_tokens.append(token_sum)

plt.figure(figsize=(8, 4))
plt.plot(vocab_sizes, total_tokens, marker='o')
plt.xlabel("Vocabulary Size")
plt.ylabel("Total Token Count (Training)")
plt.title("Vocabulary Size vs. Total Token Count")
plt.show()

# 6. Advantages and disadvantages of the tokenization method.
print("\n6. Advantages:")
print(" - Simple and computationally efficient.")
print(" - Easy to implement, understand, and debug.")
print(" - Works well when the text is clean and the vocabulary covers most frequent

print("\nDisadvantages:")
print(" - Out-of-vocabulary words are all replaced with <unk>, losing detail.")
print(" - Does not capture subword or morphological features.")
print(" - Sensitive to vocabulary size choice (rare words can get lost).")
```

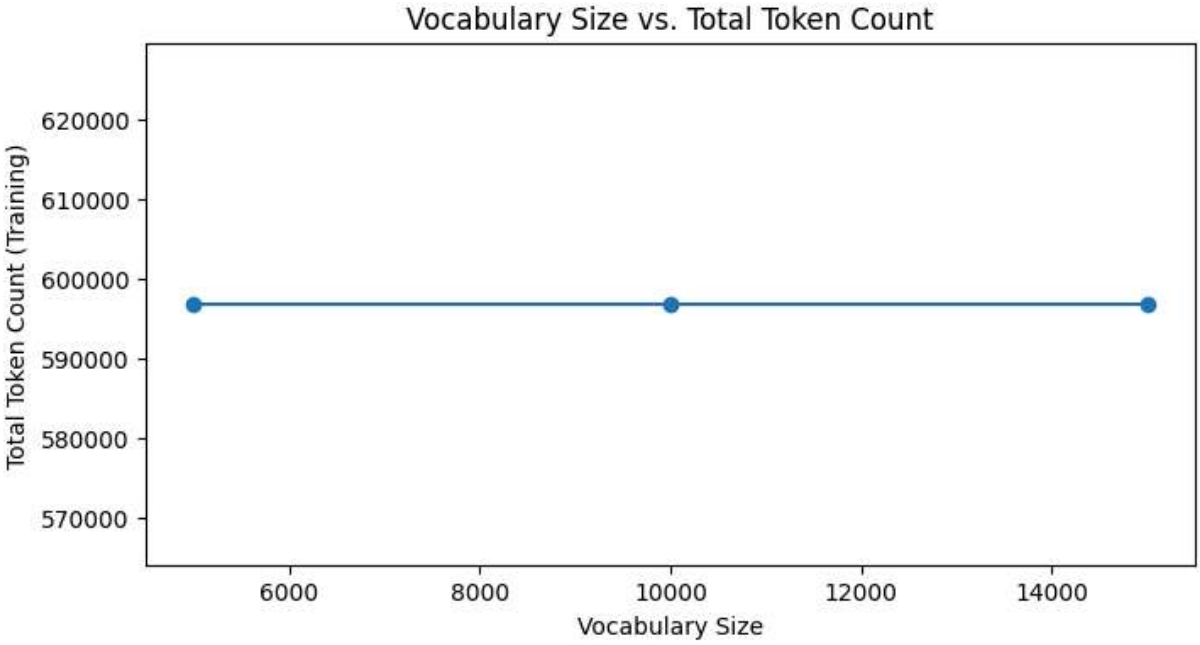
1. Unknown tokens in validation: 9263





3. Correlation between characters and tokens: 0.7768251298124685






```
6. Advantages:
 - Simple and computationally efficient.
 - Easy to implement, understand, and debug.
 - Works well when the text is clean and the vocabulary covers most frequent words.

Disadvantages:
 - Out-of-vocabulary words are all replaced with <unk>, losing detail.
 - Does not capture subword or morphological features.
 - Sensitive to vocabulary size choice (rare words can get lost).
```

3. Byte Pair Encoding

In this section, you will build the Byte Pair Encoding (BPE) tokenizer. BPE is an algorithm that replaces the most frequent pair of tokens (initially characters) with a new token. The

algorithm is configured by the number of merges that are performed. You can find the paper here Neural Machine Translation of Rare Words with Subword Units.

3.1 Finding the initial set of characters

BPE algorithm starts with the set of characters that occur in the dataset. We will build a character counter from the dataset.

Exercise 7: Counting the characters

(5p) In this exercise, we build a counter with the frequencies of all characters in the dataset. Iterate over the dataset and count the characters in the `clean` column. The function returns a `Counter` object with the characters and their frequencies.

In [66]:
```python
def build_character_counter(dataset):
    """
    Builds a character counter from the dataset
    Args:
        dataset: a dataset

    Returns: a character counter

    """
    char_counter = Counter()

    ### YOUR CODE HERE
    for example in dataset:
        # Use the cleaned text from each example.
        text = example['clean']
        char_counter.update(text)
    return char_counter


    ### YOUR CODE ENDS HERE

    return char_counter
```

The next cell applies the function to the training dataset and prints the size of the vocabulary and the most common characters.

In [67]:
```python
char_counter = build_character_counter(tweet_ds['train'])
print(len(char_counter))
print(char_counter.most_common(100))
```

```
512
[(' ', 551877), ('e', 263859), ('a', 222430), ('o', 191294), ('t', 187223), ('i', 17
6273), ('s', 172296), ('n', 157316), ('r', 152188), ('l', 122292), ('h', 106440),
('d', 81100), ('u', 79862), ('m', 73881), ('c', 73644), ('y', 70573), ('g', 62967),
('#', 56729), ('f', 54175), ('p', 51445), ('w', 47023), ('b', 46238), ('@', 36756),
('k', 31217), ('v', 30824), ('.', 23674), ('…', 19238), ('!', 16957), (',', 12384),
('', 11733), ("'", 10108), ('j', 6660), ('x', 4469), ('1', 4449), ('z', 4402), ('-',
3827), ('2', 3592), ('0', 3541), (':', 2784), ('_', 2585), (';', 2354), ('(', 2298),
('"', 2255), ('&', 2243), (')', 2092), ('q', 2078), ('6', 1852), ('/', 1711), ('?',
1624), ('5', 1561), ('3', 1430), ('4', 1096), ('7', 945), ('|', 908), ('9', 817),
('8', 761), ('•', 693), ('・', 642), ('*', 293), ('+', 288), ('$', 180), ('~', 156),
('−', 130), ('[', 127), ('é', 126), ('%', 126), (']', 118), ('=', 110), ('’', 84),
('“', 79), ('  ', 74), ('{', 63), ('}', 53), ('ñ', 44), ('♡', 31), ('”', 28), ('o',
25), ('▬', 24), ('и', 23), ('‿', 18), ('\\', 18), ('p', 17), ('^', 17), ('ン', 15),
('■', 15), ('н', 12), ('12 ,'), ('ᴵ', 12), ('a', 11), ('б', 10), ('ア', 10), ('¡', 1
0), ('ˢ', 10), ('á', 9), ('', 9), ('í', 9), ('к', 8), ('ー', 8), ('è', 8), ('ф', 8)]
```

We will filter the characters that occur less than 10 times in the dataset. We will also replace the space character with the `__` token. This is necessary because we want to preserve the spaces between the words in the tokenization process.

In [68]:
```
bpe_init_vocab = sorted([char for char, _ in char_counter.most_common() if char_cou
bpe_init_vocab[bpe_init_vocab.index(' ')] = '__'
print(bpe_init_vocab)
```

```
['__', '!', '"', '#', '$', '%', '&', "'", '(', ')', '*', '+', ',', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '=', '?', '@', '[',
'\\', ']', '^', '_', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '|', '}',
'~', '¡', 'é', 'ñ', 'ˢ', '‿', 'а', 'б', 'и', 'н', 'о', 'р', 'ᴵ', 'ᴵ', '−', '’', '“',
'”', '•', '…', '▬', '■', '♡', ' ', 'ア', 'ン', '・', '']
```

3.2 Training the BPE tokenizer

In this section, we will implement the BPE algorithm. We will start by initializing the BPE corpus. The corpus is a list of words from the dataset with their frequency. This makes it easier to find the most frequent pairs of neighbouring tokens (or characters in the beginning). Each word is split into characters and the space (the `__` token) is added at the end of each word.

In [69]:
```
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
```

In [70]:
```
bpe_corpus = init_bpe_corpus(tweet_ds['train'])
print(len(bpe_corpus))
```

```
68297
```

We can check the most common words in the corpus along with their frequencies.

In [71]:
```
bpe_corpus.most_common(30)
```

Out[71]:
```
[('@ __', 24206),
 ('. __', 23674),
 ('! __', 16957),
 ('t h e __', 13403),
 (', __', 12384),
 ('@ u s e r __', 12236),
 (' __', 9936),
 ('t o __', 7653),
 ('m y __', 7495),
 ('a __', 6814),
 ('i __', 6589),
 ('i n __', 5920),
 ('a n d __', 5588),
 ('y o u __', 5166),
 ('w i t h __', 5163),
 ('o f __', 5029),
 ('f o r __', 4737),
 ('t h i s __', 4490),
 ('c a l i f o r n i a __', 3915),
 ('a t __', 3576),
 ('i s __', 3467),
 ('l o v e __', 3046),
 ('… __', 2837),
 (': __', 2784),
 ('o n __', 2758),
 ('; __', 2354),
 ('( __', 2298),
 ('s o __', 2253),
 ('i t __', 2215),
 ('m e __', 2182)]
```

Our BPE implementation will consist of the following steps:

1. Calculate the frequency statistics of adjacent symbol pairs in the corpus.
2. Find the most frequent pair.
3. Merge the most frequent pair.
4. Repeat until the specified number of merges is reached.

The following function calculates the frequency statistics of adjacent symbol pairs in the corpus.

Exercise 8: Calculate the frequency statistics of adjacent symbol pairs

(5p) Fill in the function below to calculate the frequency statistics of adjacent symbol pairs in the corpus. The function returns a Counter object with the counts of adjacent token pairs. The pairs are represented as tuples of two tokens (e.g., `('cali', 'for')`).

In [72]:
```
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 tok

    Returns: a Counter object with the frequency statistics of adjacent symbol pair
    """
    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
```

We can check the most common pairs of characters in the initial corpus.

In [73]:
```
subset = Counter(dict(list(bpe_corpus.items())[:1000]))  # take a sample of 1000 cu
stats_subset = calculate_bpe_corpus_stats(subset)
print(stats_subset.most_common(10))
```

```
[(('e', '__'), 46280), (('t', 'h'), 30251), (('r', '__'), 25506), (('s', '__'), 2496
2), (('t', '__'), 24786), (('@', '__'), 24206), (('.', '__'), 23674), (('y', '__'),
21442), (('e', 'r'), 20476), (('h', 'e'), 19062)]
```

Next, we will implement the function that merges the most frequent pair of symbols in the corpus. The function takes the corpus and the most frequent pair of symbols as input and returns the updated corpus.

In [74]:
```
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., "l o w"),
                       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)

    # Process words in batches to reduce overhead
    batch_size = 1000
    words = list(corpus.items())

    for i in range(0, len(words), batch_size):
        batch = words[i:i+batch_size]
        for word, freq in batch:
            # More efficient way to find and replace pairs
            word_tokens = word.split()
            i = 0
            new_word = []
            while i < len(word_tokens) - 1:
                if word_tokens[i] == pair[0] and word_tokens[i+1] == pair[1]:
                    new_word.append(replacement)
                    i += 2
                else:
                    new_word.append(word_tokens[i])
                    i += 1

            # Add the last token if it wasn't part of a pair
            if i == len(word_tokens) - 1:
                new_word.append(word_tokens[i])

            new_word = ' '.join(new_word)
            new_corpus[new_word] += freq

    return new_corpus
```

The last step is to implement the BPE algorithm. The function takes the initial vocabulary, the corpus, and the number of merges as input. It returns the updated vocabulary, corpus, and the list of merges. Returning the list of merges is useful for the tokenization process - it makes it faster to tokenize the text. It contains the tuples of the two tokens that were merged. For example, ('to', 'day__') will merge the tokens 'to' and 'day__' into the 'today__' token.

Exercise 9: BPE algorithm

(10p) Implement the BPE algorithm in the following function. The function should return the updated vocabulary, corpus, and the list of merges. The function should perform the specified number of merges. The vocabulary is a list of tokens, the corpus is a Counter object with the words split by space into tokens, and the merges is a list of tuples with the merged tokens.

You should use the functions you implemented earlier in this section (`calculate_bpe_corpus_stats()`, `merge_corpus()`).

In [75]:
```
def bpe(vocab, corpus, num_merges):
    """
    Applies the BPE algorithm to the corpus. Merges the most frequent adjacent symb
    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
```

The following cell applies the BPE algorithm to the initial vocabulary and corpus. We will perform 100 merges at first, but you will experiment with different numbers.

In [81]:
```
bpe_vocab, updated_bpe_corpus, bpe_merges = bpe(bpe_init_vocab, bpe_corpus, num_mer
```

```
100%|██████████| 100/100 [00:46<00:00,  2.15it/s]
```

We can check the size of the BPE vocabulary and the most common tokens.

In [82]:
```
print(len(bpe_vocab))
print(bpe_vocab[:150])
```

```
193
['__', '!', '"', '#', '$', '%', '&', "'", '(', ')', '*', '+', ',', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '=', '?', '@', '[',
'\\', ']', '^', '_', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '|', '}',
'~', '¡', 'é', 'ñ', 'ˢ', '_', 'a', '6', 'и', 'н', 'o', 'p', 'ˈ', 'ᴵ', '–', '’', '“',
'”', '•', '…', '▬', '▄', '♡', ' ', 'ア', 'ン', '・', '', 'e__', 's__', 't__', 'th',
'er', 'in', 'y__', 'an', 'd__', 'or', 'a__', '@__', 'er__', '.__', 'on', 'o__', 'a
l', 'ou', 'ar', '…__', 'ing', 'en', 'us', '!__', 'st', 'ing__', 're', 'ch', 'the__',
'lo', ',__', 'user__', 'l__', '@user__', 'at', 'for', 'am', 'da', 'el', 'it', '__',
'om', 'ri', 'be', 'k__', 'on__', 'is__', 'la', 'ho', 'to__', 'ni', 'i__', 'in__', 'e
s__', 'ha', 'you', 'es']
```

We can also check the most common merges.

In [83]:
```
print(bpe_merges[:150])
```

```
[('e', '__'), ('s', '__'), ('t', '__'), ('t', 'h'), ('e', 'r'), ('i', 'n'), ('y', '_
_'), ('a', 'n'), ('d', '__'), ('o', 'r'), ('a', '__'), ('@', '__'), ('er', '__'),
('.', '__'), ('o', 'n'), ('o', '__'), ('a', 'l'), ('o', 'u'), ('a', 'r'), ('…', '_
_'), ('in', 'g'), ('e', 'n'), ('u', 's'), ('!', '__'), ('s', 't'), ('ing', '__'),
('r', 'e'), ('c', 'h'), ('th', 'e__'), ('l', 'o'), (',', '__'), ('us', 'er__'),
('l', '__'), ('@', 'user__'), ('a', 't'), ('f', 'or'), ('a', 'm'), ('d', 'a'), ('e',
'l'), ('i', 't'), ('', '__'), ('o', 'm'), ('r', 'i'), ('b', 'e'), ('k', '__'), ('o
n', '__'), ('i', 's__'), ('l', 'a'), ('h', 'o'), ('t', 'o__'), ('n', 'i'), ('i', '_
_'), ('in', '__'), ('e', 's__'), ('h', 'a'), ('y', 'ou'), ('e', 's'), ('an', 'd__'),
('l', 'i'), ('o', 'f'), ('m', 'y__'), ('o', 'o'), ('p', '__'), ('v', 'e__'), ('w',
'i'), ('da', 'y__'), ('i', 's'), ('i', 'c'), ('th', '__'), ('a', 't__'), ('i', 'l'),
('w', '__'), ('i', 'r'), ('u', 'n'), ('g', 'h'), ('a', 's__'), ('c', 'al'), ('n',
'e'), ('a', 's'), ('#', 's'), ('an', '__'), ('a', 'c'), ('r', 'o'), ('t', 'i'),
('t', 'o'), ('you', '__'), ('l', 'e'), ('wi', 'th__'), ('of', '__'), ('u', 'r'),
('e', 'd__'), ('cal', 'i'), ('a', 'd'), ('ch', '__'), ('for', '__'), ('cali', 'fo
r'), ('califor', 'ni'), ('p', 'p'), ("'", 's__'), ('th', 'is__')]
```

3.3 Tokenizing the text using BPE

With the tokenizer trained we can now tokenize the text using the BPE vocabulary. We will first build a function that tokenizes any text using our BPE tokenizer (vocabulary and merges). Next we will apply it to our dataset.

The following function tokenizes the text using the BPE vocabulary. It replaces the most frequent pairs of tokens with the new token. The function also replaces the tokens that are not in the vocabulary with the `<unk>` token.

In [84]:
```
def apply_bpe_tokenization(text, vocab, merges, unk_token='<unk>'):
    """
    Tokenizes the text using BPE vocabulary, preserving spaces as '__'.

    Args:
        text (str): The input text to be tokenized.
```

```
        vocab (set): A set containing the BPE vocabulary tokens.

    Returns:
        list: A list of tokens representing the input text.
    """

    words = re.split(r'\s', text)
    words = [' ' + ' '.join(list(word)) + (' __ ' if i < len(words) - 1 else ' ') f

    bpe_tokens = []

    for i, word in enumerate(words):
        for merge in merges:
            word = word.replace(' ' + ' '.join(merge) + ' ', ' ' + ''.join(merge) +
        bpe_tokens.extend(word.split())

    for i, token in enumerate(bpe_tokens):
        if token not in vocab:
            bpe_tokens[i] = unk_token
    return bpe_tokens


# A test example with a special character. Is the character tokenized correctly as
print(apply_bpe_tokenization(tweet_ds['train'][0]['clean'] + ' u', bpe_vocab, bpe_m
```

```
['s', 'un', 'day__', 'a', 'f', 't', 'er', 'n', 'o', 'on__', 'w', 'al', 'k', 'ing__',
'th', 'r', 'ou', 'gh', '__', 'v', 'en', 'ic', 'e__', 'in__', 'the__', 's', 'un', '_
_', 'with__', '@user__', '__', '__', '__', '@__', 'a', 'b', 'b', 'o', 't__', 'k', 'i
n', 'ne', 'y__', ',__', 'v', 'en', 'ic', 'e__', '<unk>']
```

The function below will apply our BPE tokenizer to the dataset. It will add a new column `bpe_tokens` to each example.

In [85]:
```
def tokenize_bpe(example, vocab, merges, unk_token='<unk>'):
    """
    Tokenizes the example from the Dataset using BPE
    Args:
        example: an example from the Dataset
        vocab: a BPE vocabulary

    Returns: update example containing 'bpe_tokens' column

    """
    text = example['clean']
    bpe_tokens = apply_bpe_tokenization(text, vocab, merges, unk_token)
    example['bpe_tokens'] = bpe_tokens
    return example

tweet_ds = tweet_ds.map(tokenize_bpe, fn_kwargs={'vocab': bpe_vocab, 'merges': bpe_
print(tweet_ds)
```

```
Map:   0%|          | 0/45000 [00:00<?, ? examples/s]
Map:   0%|          | 0/50000 [00:00<?, ? examples/s]
Map:   0%|          | 0/5000 [00:00<?, ? examples/s]
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label', 'text_length', 'clean', 'tokens', 'bpe_tokens'],
        num_rows: 45000
    })
    test: Dataset({
        features: ['text', 'label', 'text_length', 'clean', 'tokens', 'bpe_tokens'],
        num_rows: 50000
    })
    validation: Dataset({
        features: ['text', 'label', 'text_length', 'clean', 'tokens', 'bpe_tokens'],
        num_rows: 5000
    })
})
```

We will inspect the both tokenizations of several examples from the `validation` subset. Try to find the `<unk>` tokens in the printed examples.

In [86]:
```
for i in range(10):
    print('Original tweet:')
    print(tweet_ds['validation'][i]['text'])
    print('Word tokenization:')
    print(tweet_ds['validation'][i]['tokens'])
    print('BPE tokenization:')
    print(tweet_ds['validation'][i]['bpe_tokens'])
    print()
```

```
Original tweet:
A little throwback with my favourite person @ Water Wall
Word tokenization:
['a', 'little', 'throwback', 'with', 'my', 'favourite', 'person', '@', 'water', 'wal
l']
BPE tokenization:
['a__', 'l', 'it', 't', 'l', 'e__', 'th', 'ro', 'w', 'b', 'ac', 'k__', 'with__', 'my
__', 'f', 'a', 'v', 'ou', 'r', 'it', 'e__', 'p', 'er', 's', 'on__', '@__', 'w', 'a
t', 'er__', 'w', 'al', 'l']

Original tweet:
glam on @user yesterday for #kcon makeup using @user in #featherette,…
Word tokenization:
['glam', 'on', '@user', 'yesterday', 'for', '<unk>', 'makeup', 'using', '@user', 'i
n', '<unk>', ',', '…']
BPE tokenization:
['g', 'l', 'am', '__', 'on__', '@user__', 'y', 'e', 'st', 'er', 'day__', 'for__',
'#', 'k', 'c', 'on__', 'm', 'a', 'k', 'e', 'u', 'p__', 'us', 'ing__', '@user__', 'in
__', '#', 'f', 'e', 'a', 'th', 'er', 'e', 't', 't', 'e__', ',__', '…']

Original tweet:
Democracy Plaza in the wake of a stunning outcome #Decision2016 @ NBC News
Word tokenization:
['<unk>', 'plaza', 'in', 'the', 'wake', 'of', 'a', 'stunning', '<unk>', '<unk>',
'@', '<unk>', 'news']
BPE tokenization:
['d', 'e', 'm', 'o', 'c', 'r', 'ac', 'y__', 'p', 'la', 'z', 'a__', 'in__', 'the__',
'w', 'a', 'k', 'e__', 'of__', 'a__', 'st', 'un', 'n', 'ing__', 'ou', 't', 'c', 'om',
'e__', '#', 'd', 'e', 'c', 'is', 'i', 'on', '2', '0', '1', '6', '__', '@__', 'n',
'b', 'c', '__', 'ne', 'w', 's']

Original tweet:
Then &amp;amp; Now. VILO @ Walt Disney Magic Kingdom
Word tokenization:
['then', '&amp', ';', 'now', '.', '<unk>', '@', 'walt', 'disney', 'magic', 'kingdo
m']
BPE tokenization:
['th', 'en', '__', '&', 'am', 'p__', ';', '__', 'n', 'o', 'w__', '.__', 'v', 'il',
'o__', '@__', 'w', 'al', 't__', 'd', 'is', 'ne', 'y__', 'm', 'a', 'g', 'ic', '__',
'k', 'ing', 'd', 'om']

Original tweet:
Who never... @ A Galaxy Far Far Away
Word tokenization:
['who', 'never', '.', '.', '.', '@', 'a', 'galaxy', 'far', 'far', 'away']
BPE tokenization:
['w', 'h', 'o__', 'ne', 'v', 'er__', '.__', '.__', '.__', '@__', 'a__', 'g', 'al',
'a', 'x', 'y__', 'f', 'ar', '__', 'f', 'ar', '__', 'a', 'w', 'a', 'y']

Original tweet:
Dinner in FLA tonight // Pan-seared salmon over couscous veggie salad #yum #dinner #
florida #salmon…
Word tokenization:
['dinner', 'in', '<unk>', 'tonight', '//', '<unk>', 'salmon', 'over', '<unk>', 'vegg
ie', 'salad', '#yum', '#dinner', '<unk>', '<unk>']
BPE tokenization:
```

```
['d', 'in', 'n', 'er__', 'in__', 'f', 'l', 'a__', 't', 'on', 'i', 'gh', 't__', '/',
'/', '__', 'p', 'an', '-', 's', 'e', 'ar', 'ed__', 's', 'al', 'm', 'on__', 'o', 'v',
'er__', 'c', 'ou', 's', 'c', 'ou', 's__', 'v', 'e', 'g', 'g', 'i', 'e__', 's', 'al',
'a', 'd__', '#', 'y', 'u', 'm', '__', '#', 'd', 'in', 'n', 'er__', '#', 'f', 'l', 'o
r', 'i', 'd', 'a__', '#s', 'al', 'm', 'on', '…']

Original tweet:
It's my fav seniors last game congrats on beating west @ West Salem…
Word tokenization:
["it's", 'my', 'fav', 'seniors', 'last', 'game', 'congrats', 'on', 'beating', 'wes
t', '@', 'west', '<unk>']
BPE tokenization:
['it', "'s__", 'my__', 'f', 'a', 'v', '__', 's', 'en', 'i', 'or', 's__', 'la', 's',
't__', 'g', 'am', 'e__', 'c', 'on', 'g', 'r', 'at', 's__', 'on__', 'be', 'at', 'ing_
_', 'w', 'es', 't__', '@__', 'w', 'es', 't__', 's', 'al', 'e', 'm', '…']

Original tweet:
I got to to go formal with my best friend @ Phi Mu at JSU
Word tokenization:
['i', 'got', 'to', 'to', 'go', 'formal', 'with', 'my', 'best', 'friend', '@', 'phi',
'mu', 'at', '<unk>']
BPE tokenization:
['i__', 'g', 'o', 't__', 'to__', 'to__', 'g', 'o__', 'for', 'm', 'al', '__', 'with_
_', 'my__', 'be', 's', 't__', 'f', 'ri', 'en', 'd__', '@__', 'p', 'h', 'i__', 'm',
'u', '__', 'at__', 'j', 's', 'u']

Original tweet:
'Cause I Miss My Little Homies .#Throwback #CousinLove @ Indiana University
Word tokenization:
['<unk>', 'i', 'miss', 'my', 'little', 'homies', '.', '#throwback', '#cousinlove',
'@', '<unk>', 'university']
BPE tokenization:
["'", 'c', 'a', 'us', 'e__', 'i__', 'm', 'is', 's__', 'my__', 'l', 'it', 't', 'l',
'e__', 'h', 'om', 'i', 'es__', '.__', '#', 'th', 'ro', 'w', 'b', 'ac', 'k__', '#',
'c', 'ou', 's', 'in', 'lo', 've__', '@__', 'in', 'd', 'i', 'an', 'a__', 'u', 'ni',
'v', 'er', 's', 'it', 'y']

Original tweet:
Birthday Kisses @ Madison, Wisconsin
Word tokenization:
['birthday', 'kisses', '@', 'madison', ',', '<unk>']
BPE tokenization:
['b', 'ir', 'th', 'day__', 'k', 'is', 's', 'es__', '@__', 'm', 'ad', 'is', 'on__',
',__', 'wi', 's', 'c', 'on', 's', 'in']
```

Exercise 10: Comparing tokenizers

Train the BPE tokenizer with different number of merges. Compare the tokenization results with the word tokenization.

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.

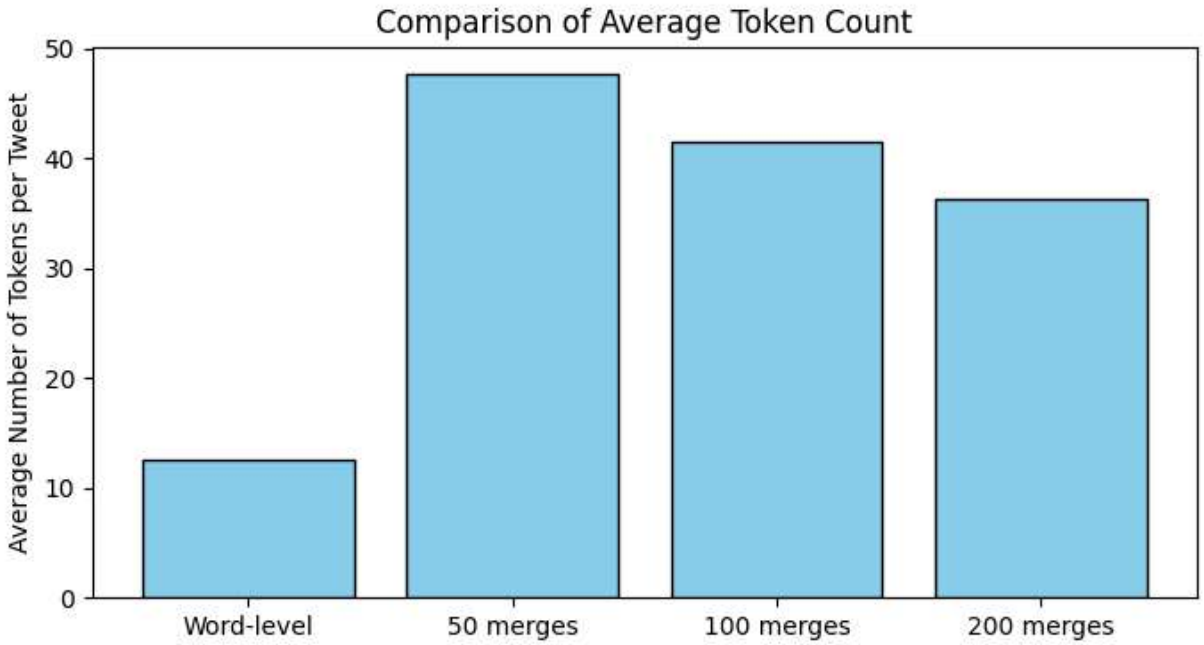
4. (5p) Compare the average length in tokens between different tokenizers.
5. (5p) What are the advantages and disadvantages of the BPE tokenizer?

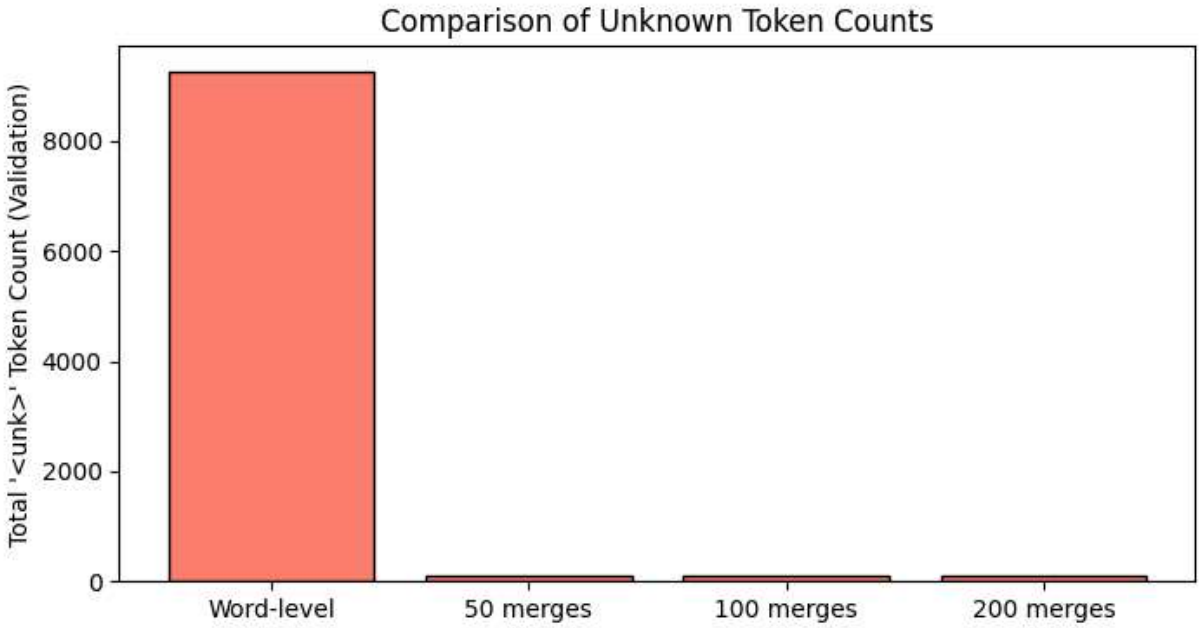
For answering these questions make sure to include a proper mix of numbers/plots/tables etc. and comments.

In [87]:
```
merge_values = [50, 100, 200]  # You can adjust these values as needed.
bpe_results = {}   # To store statistics for each BPE tokenizer.

for num_merges in merge_values:
    # Train the BPE tokenizer using num_merges merge operations.
    bpe_vocab_temp, updated_corpus_temp, bpe_merges_temp = bpe(bpe_init_vocab, bpe_

    # Tokenize the validation set using the current BPE tokenizer.
    bpe_token_counts = []
    bpe_unknown_count = 0
    for ex in tweet_ds['validation']:
        tokens = apply_bpe_tokenization(ex['clean'], bpe_vocab_temp, bpe_merges_tem
        bpe_token_counts.append(len(tokens))
        bpe_unknown_count += tokens.count('<unk>')

    avg_bpe_tokens = np.mean(bpe_token_counts)
    bpe_results[num_merges] = {
        'avg_tokens': avg_bpe_tokens,
        'unk_count': bpe_unknown_count
    }

# For the word-level tokenizer, we assume that we have already created a 'tokens' f
# using simple whitespace splitting and replacing unknown words.
word_token_counts = [len(ex['tokens']) for ex in tweet_ds['validation']]
avg_word_tokens = np.mean(word_token_counts)
word_unk_count = sum(ex['tokens'].count('<unk>') for ex in tweet_ds['validation'])

# Print word-level tokenization results.
print("Word-level Tokenization (Validation):")
print(f" Average number of tokens: {avg_word_tokens:.2f}")
print(f" Total '<unk>' tokens: {word_unk_count}")

# Print BPE tokenizer results.
print("\nBPE Tokenization Results:")
for num_merges, stats in bpe_results.items():
    print(f" {num_merges} merges: Average tokens = {stats['avg_tokens']:.2f}, <unk>

# (Optional) Plot the comparison in bar charts.
# Plot average token counts for word-level and BPE tokenizers.
labels = ['Word-level'] + [f'{m} merges' for m in merge_values]
avg_tokens_values = [avg_word_tokens] + [bpe_results[m]['avg_tokens'] for m in merg

plt.figure(figsize=(8, 4))
plt.bar(labels, avg_tokens_values, color='skyblue', edgecolor='black')
plt.ylabel("Average Number of Tokens per Tweet")
plt.title("Comparison of Average Token Count")
plt.show()
```

```python
# Plot unknown token counts for word-level and BPE tokenizers.
unk_values = [word_unk_count] + [bpe_results[m]['unk_count'] for m in merge_values]

plt.figure(figsize=(8, 4))
plt.bar(labels, unk_values, color='salmon', edgecolor='black')
plt.ylabel("Total '<unk>' Token Count (Validation)")
plt.title("Comparison of Unknown Token Counts")
plt.show()

# ------------------------------
# Advantages and Disadvantages Discussion:
# ------------------------------
print("\nAdvantages of BPE Tokenization:")
print(" - Breaks words into subwords, reducing the number of out-of-vocabulary toke
print(" - Can better handle morphological variations and rare words.")
print(" - Often improves model performance when dealing with large vocabularies.")

print("\nDisadvantages of BPE Tokenization:")
print(" - More complex to implement and train compared to simple whitespace tokeniz
print(" - Requires tuning the number of merge operations.")
print(" - May produce subword tokens that are less interpretable by humans.")
```

```
100%|██████████| 50/50 [00:27<00:00,  1.82it/s]
100%|██████████| 100/100 [00:57<00:00,  1.75it/s]
100%|██████████| 200/200 [01:35<00:00,  2.10it/s]
Word-level Tokenization (Validation):
 Average number of tokens: 12.55
 Total '<unk>' tokens: 9263

BPE Tokenization Results:
 50 merges: Average tokens = 47.68, <unk> count = 95
 100 merges: Average tokens = 41.50, <unk> count = 95
 200 merges: Average tokens = 36.28, <unk> count = 95
```





Advantages of BPE Tokenization:
- Breaks words into subwords, reducing the number of out-of-vocabulary tokens.
- Can better handle morphological variations and rare words.
- Often improves model performance when dealing with large vocabularies.

Disadvantages of BPE Tokenization:
- More complex to implement and train compared to simple whitespace tokenization.
- Requires tuning the number of merge operations.
- May produce subword tokens that are less interpretable by humans.