CSCI 4140: Natural Language Processing

CSCI/DASC 6040: Computational Analysis of Natural Languages

Spring 2025

Homework 3 - Tokenization

Due Sunday, February 16, at 11:59 PM

1. Byte-Pair Encoding (BPE) (60 pts)

Background

The rare/unknown word issue is ubiquitous in neural text generation. As we discussed in class, words not appearing in the vocabulary need to be replaced with a special `⟨unk⟩` symbol.

Byte-pair encoding (BPE) is a popular tokenization technique that addresses this issue. The idea is to encode text using a set of automatically constructed types, instead of using conventional (white-space-separated) word types. A type can be a character or a subword unit, and the types are built through a iterative process, which we now walk you through.

Walking through an example: Suppose we have the following tiny training data: `it unit unites`.

Training the Tokenizer:

- **First, we segment the data into characters.** Our initial types are the characters and the special beginning-of-word symbol: `{i, t, u, n, e, s, ⟨s⟩}`. Using the initial type set, the encoding of the training data is: `i t ⟨s⟩ u n i t ⟨s⟩ u n i t e s`. (Note that the beginning-of-word symbol here plays the same rule as the space token used in the lecture, and won't be prepended to a word at the beginning of a line)
- **In each training iteration, the most frequent type bigram is merged into a new symbol and then added to the type vocabulary.** Ties can be broken at random. The bigram count for our initial data is: `(i, t) : 3, (⟨s⟩, u): 2, (u, n) : 2, (n, i) : 2, (t, e) : 1, (e, s) : 1`. Note that we don't consider merges across white-space-separated words.
  - **Iteration 1:**

        - Bigram to merge (with frequency 3): `i t`
        - Updated data: `it ⟨s⟩ u n it ⟨s⟩ u n it e s`
    - **Iteration 2:**
        - Bigram to merge (with frequency 2): `⟨s⟩ u`
        - Updated data: `it ⟨s⟩u n it ⟨s⟩u n it e s`
    - **Iteration 3:**
        - Bigram to merge (with frequency 2): `⟨s⟩u n`
        - Updated data: `it ⟨s⟩un it ⟨s⟩un it e s`
- In this example, we end up with the type vocabulary `{i, t, u, n, e, s, ⟨s⟩, it, ⟨s⟩u, ⟨s⟩un}`. The stopping criterion can be defined through a target vocabulary size.

Applying to New Words:

- At inference-time, we encode text with BPE by first splitting the word into characters, and then iteratively applying the merge rules in the same order as learned in training. Suppose we want to encode the text **itunes unite**. This text is first split into characters, `i t u n e s ⟨s⟩ u n i t e`.
- Then,
    - **Iteration 1:**
        - Bigram to merge: `i t`
        - Encoded word: `it u n e s ⟨s⟩ u n it e`
    - **Iteration 2:**
        - Bigram to merge: `⟨s⟩ u`
        - Encoded word: `it u n e s ⟨s⟩u n it e`
    - **Iteration 3:**
        - Bigram to merge: `⟨s⟩u n`
        - Encoded word: `it u n e s ⟨s⟩un it e`
    - The above procedure is repeated until no merge rule can be applied. In this example we get the encoding: `it u n e s ⟨s⟩un it e`.

Your implementation of BPE (10 pts)

Now train a BPE tokenizer on the provided text file, `BPE-data.txt`. Use the **first 4000 lines** as the training set. **Run the algorithm until the frequency of the most frequent type bigram is two.**

Note that the above procedure isn't exactly how people use BPE in real-life. There are other implementation details like how to handle punctuation that aren't covered. If you would like to see a more comprehensive description of how BPE tokenizer works, please see the tutorial from Huggingface. It can also be helpful for implementing the algorithm, **but please refrain from copy-pasting code directly**.

In [1]:
```
def bpe_train(data): # train the BPE tokenizer
    """
    IMPLEMENT ME!
    """
    pass

def bpe_apply(data): # apply the BPE tokenizer
    """
    IMPLEMENT ME!
    """
    pass
```

Q1.1: Scatterplot (30 pts: 10 pts for code, 10 pts for each answer)

Please produce a scatterplot showing points `(x, y)`, each corresponding to an iteration of the algorithm, with `x` the current size of the type vocabulary (including the base vocabulary), and `y` the length of the training corpus (in tokens) under that vocabulary's types.

In [ ]:
```
"""
IMPLEMENT ME!
"""
```

- How many types do you end up with?
- What is the length of the training data under your final type vocabulary?

Your answers:

Q1.2: Rare word issue (10 pts)

Another way of overcoming the rare word issue is to encode text as a sequence of characters. Discuss the advantages and potential issues of character-level encoding compared to BPE.

Your answer:

Q1.3: Evaluate on test data (10 pts)

- Applying your tokenizer on the last 1000 lines of the data, how many tokens do you end up with?
- Do you encounter issues with words that didn't appear in the training set?
- More generally, when would you expect your tokenizer to fail when applying to an unseen text?

In [ ]:
```
"""
IMPLEMENT ME!
"""
```

Your answers:

2. WordPiece (50 pts)

Background

Another popular tokenization algorithm is **WordPiece**, which is used by a few BERT-based models. The idea of WordPiece is similar to BPE: if two tokens frequently appear together, then merge them into a new token. However, WordPiece normalizes the bigram frequency by the two tokens' individual frequency. In short, at each iteration, BPE merges the pair `(a, b)` that maximizes `freq(ab)`, whereas WordPiece merges the pair `(a, b)` that maximizes `freq(ab)/(freq(a) × freq(b))`.

Applying the tokenizer at inference-time is also different in WordPiece. In BPE, we apply each merge iteratively following the order during training. However, in WordPiece, we simply save the final vocabulary and greedily find the longest matching subword until the text is fully encoded. For example, given the vocabulary `{u, n, d, o, do, un, und}`, the word `undo` will be tokenized into `und o` in Wordpiece whereas in BPE it will be `un do` as the merge `u n` must be learned before `un d`.

Your implementation of WordPiece (10 pts)

Now implement WordPiece algorithm on the training set in the previous problem, `BPE-data.txt`. Again, feel free to refer to the Huggingface tutorial for a more detailed walkthrough, and note that the pre-tokenization procedure you are going to apply in this homework is slightly different from the standard approach.

Update: Run the algorithm until 4000 new merge rules are learned. (i.e., the vocabulary has size 4000 excluding the base vocabulary)

In [1]:
```
def wordpiece_train(data): # train the WordPiece tokenizer
    """
    IMPLEMENT ME!
    """
    pass

def wordpiece_apply(data): # apply the WordPiece tokenizer
    """
    IMPLEMENT ME!
    """
    pass
```

Q2.1: Scatterplot (20 pts: 10 pts for code, 10 pts for answers)

Please produce a scatterplot showing points `(x, y)`, each corresponding to an iteration of the algorithm, with `x` the current size of the type vocabulary (including the base vocabulary), and `y` the length of the training corpus (in tokens) under that vocabulary's types.

In [ ]:
```
"""
IMPLEMENT ME!
"""
```

- How many types do you end up with?
- What is the length of the training data under your final type vocabulary?

Your answers:

Q2.2: Evaluate on test data (10 pts)

- Applying your tokenizer on the last 1000 lines of the data, report the length of the tokenized data. Also, include the tokenized sequences for the following two sentences:
  - `Analysts were expecting the opposite, a deepening of the deficit.`
  - `Five minutes later, a second person arrived, aged around thirty, with knife wounds.`

In [ ]:
```
"""
IMPLEMENT ME!
"""
```

Your answer:

Q2.3 WordPiece vs. BPE (10 pts)

In terms of efficiency and performance, what are the advantages and disadvantages of WordPiece compared with BPE? There is no single correct answer here, just provide your thoughts and rationales, supported by empirical evidence.

Your answer: