TextTokenization

January 20, 2026

1 Text tokenization using BPE

In this assignment, you will complete two main tasks:

1. Train a character-level BPE algorithm for subword tokenization on the text of Moby Dick and plot the dependency between the number of tokens discovered and the minimum frequency threshould used for training.
2. Compute the frequency vs. rank distribution of the tokens in Moby Dick. For this, you will need to tokenize the document and create a vocabulary mapping token types to their document frequency.

1.1 Write Your Name Here:

2 Submission instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Edit -> Clear Output of All Cells. This will clear all the outputs from all cells (but will keep their content).
4. Select Run -> Run All Cells. This will run all the cells in order, and may take several seconds.
5. Once you've rerun everything, select File -> Save and Export Notebook As -> PDF and download a PDF version showing the code and the output of all cells, and save it in the same folder that contains the notebook file.
6. Look at the PDF file and make sure all your solutions and outputs are there, displayed correctly.
7. Submit **both** the PDF and your notebook file .ipynb on Canvas. If you encounter issues converting to PDF (make sure 'nbconvert' is installed first), saving as HTML is acceptable too (PDF is preferable).
8. Make sure your your Canvas submission contains the correct files by downloading it after posting it on Canvas.

2.1 Virtual Environments and Packages

It is recommended that you install the packages required for this class in a *virtual environment*, for which I recommend the Python virtual environment tool `venv` (other package managers exist that you can use, such as 'conda'). To use `venv` (to manage the environment) and `pip` (to install packages) please see this quick guide. You can first create a folder named "itcs6101" for the course, inside which you will place all your homework assignment folders. A prototypical setup in a Unix-based system looks like this:

1. *mkdir itcs6101* # this creates the course folder.
2. *cd itcs6101* # this changes the current directory to that folder.
3. *python3 -m venv .venvnlp* # this creates the virtual environment named 'venlp' (you can use any name you want).
4. *source .venvnlp/bin/activate* # this activates the virtual environment 'venvnlp'. After this, anything you install with pip will be installed in this environment.
5. *python3 -m pip install –upgrade pip* # this installs the 'pip' package manager.
6. *pip install jupyterlab* # this installs the 'jupyterlab' package.
7. *pip install nbconvert* # this installs the 'nbconvert' package, needed to convert notebooks to PDF.

We will use 'jupyterlab' to edit the notebooks for the homework assignments. You can similarly use the 'pip' command (inside the activated '.venvnlp' environment) to install other packages required for homework assignments.

2.2 Byte-Pair Encoding (BPE) algorithm

Create a Python implementation of the Byte-Pair Encoding (BPE) algorithm for subword tokenization. The code should be structured in two functions:

- A training function `bpe_train(corpus, K)`.
- A tokenization function `bpe_encode(text, rules)`.

The behavior of each function is described in its own section below.

2.2.1 The BPE training function (60 points)

Implement the training function `bpe_train(corpus, K, F)`, where `corpus` is the name of a text file containing the training corpus, and `K` is the number of token merges to be done by the BPE algorithm, and `F` is the minimum frequency a pair needs to have in order for it to be merged. The function should first preprocess the text corpus by splitting it into words using whitespaces as separators, for example as done by the Python `split()` method. Each word that is preceded by a whitespace in the original text corpus should be prepended the whitespace character ' '. The BPE algorithm will then be run on the resulting set of words. An initial vocabulary `V` should be created to contain all unique *characters* that occur across all the words. The function should return the final vocabulary `V`, a list `rules` that contains all token merges, in the order in which they were created, and the freuqncy `fmax` of the last pair of tokens that were merged.

For efficiency, in my implementation I maintain the corpus at each iteration as a dictionary `word2freq` mapping the tokenized version of each unique word in the corpus to the number of times that word apepars in the corpus. Furthermore, I implemented auxiliary functions, such as:

- `get_word_list(text)` splits the initial text in the corpus into words and prepends a space to words that were preceded by a whitespace in the original text.
- `get_stats(word2freq)` computes the frequency for every unique pair of tokens appearing in the current version of the corpus.

- `merge_pair(pair, word2freq)` creates and returns the new tokenized version of the corpus, by merging a pair of tokens everywhere it is seen in the corpus.

[16]:
```
# Any necessary imports go here.


# Any auxiliary functions go here.









def bpe_train(corpus_filename, K = None, F = None):
    if K == None and F == None:
        K, F = 10000, 5

    # Empty vocabulary, empty set of merge rules, zero frequency
    V, rules, fmax = set([]), [], 0

    # YOUR CODE HERE

    return V, rules, fmax
```

2.2.2 The BPE tokenization function (25 points)

Implement the BPE tokenization function `bpe_encode(text, rules)`, where `text` is a Python string, and `rules` is a list of token merges created by `bpe_train`. The function `bpe_encode` should preprocess the text in the same way as was done in `bpe_train`, namely split it into words using whitespaces as separators. Each word that is preceded by a whitespace in the original text input is then prepended the whitespace character ' '. At the beginning, each word is broken into a list of

character tokens. The function then iterates through all the token merges from `rules`, and applies each of them on the current tokenization of each word, whenever possible. The function should return the tokenized version of `text` as a list of tokens.

[17]:
```
def bpe_encode(text, rules):
    final_tokens = []

    # YOUR CODE HERE









    return final_tokens
```

2.2.3 Testing the BPE tokenizer (20 points)

The code below shows the tokenization obtained on two sentences, when using a BPE algorithm trained for K = 1000 merges. When 'whaling' is inside the sentence, the tokenizer recognizes it as one token. However, when 'whaling' starts the sentence, the tokenizer splits it into 3 tokens.

1. Explain the behavior above.
2. Does increasing K to 5000 change how 'whaling' at the start of the sentence is tokenized? Is it ever going to be tokenized as one token? Explain.

[ ]:
```
corpus_file = '../data/melville-moby_dick.txt'

print("--- bpe_train ---")
V, rules, fmax = bpe_train(corpus_file, K = 1000)
print(f"Frequency of last merge: {fmax}")
print(f"Number of merges: {len(rules)}")
```

```
# Print the 50 longest tokens. You should obtain the list below:
# [' Whale', ' other', ' still', ' sight', ' while', ' would', ' whale,', '␣
 ↪almost', ' little', ' before', ' length', ' though', ' matter', ' should', '␣
 ↪seemed', ' remain', ' called', ' consid', ' things', ' Pequod', ' fisher', '␣
 ↪moment', 'viathan', 'ueequeg', ' sudden', ' whales', ' there,', ' without',␣
 ↪' another', ' present', ' nothing', ' himself', ' instant', ' captain', '␣
 ↪thought', 'antucket', ' certain', ' Captain', ' towards', ' strange', '␣
 ↪whaling', ' through', ' CHAPTER', ' against', ' between', ' Queequeg', '␣
 ↪harpoone', ' Starbuck', ' Nantucket', ' something']
print(sorted(V, key = len)[-50:])

# Print the first 30 rules
print(rules[:30])

# Print last 30 rules
print(rules[-30:])

print("\n--- bpe_encode ---")
text = "it is a dangerous business to be in whaling if you are clumsy."
tokens = bpe_encode(text, rules)
print(f"Text: '{text}'")
print(f"Tokens: {tokens}") # You should obtain Tokens: ['it', ' is', ' a', '␣
 ↪d', 'ang', 'er', 'ous', ' b', 'us', 'iness', ' to', ' be', ' in', '␣
 ↪whaling', ' if', ' you', ' are', ' cl', 'um', 's', 'y.']

text = "whaling is a dangerous business."
tokens = bpe_encode(text, rules)
print(f"Text: '{text}'")
print(f"Tokens: {tokens}") # You should obtain Tokens: ['w', 'ha', 'ling', '␣
 ↪is', ' a', ' d', 'ang', 'er', 'ous', ' b', 'us', 'in', 'es', 's.']
```

2.2.4 Maximum token length vs. Token merges (20 points)

Train the BPE tokenizer for K = 500, 1000, 2500, and 5000. For each value of K:

1. Report the list of the 30 longest tokens in the created vocabulary.
2. Report the maximum token length.

```
[ ]: # YOUR CODE HERE
```

2.3 Token merges vs. Frequency (20 points)

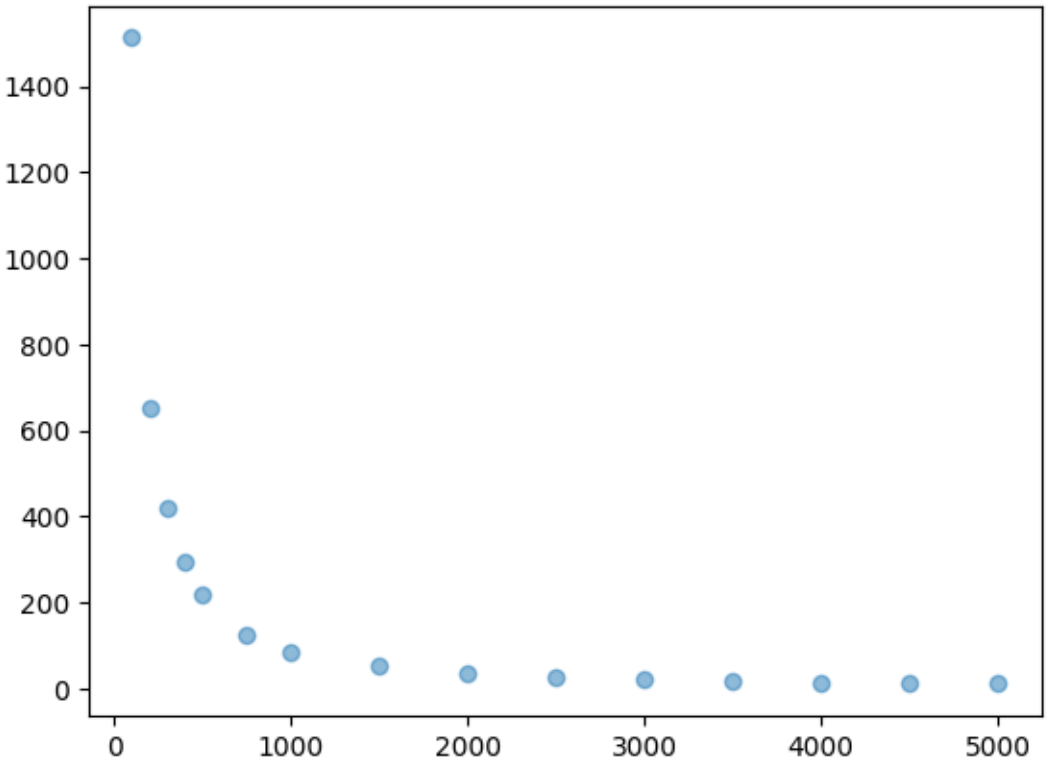
Implement an experimental version of the training function `bpe_train_exp(corpus, M)`, where the argument `M` is a list of integers in increasing order, each representing a number of merges. For each entry in this list, record the frequency of the last pair of tokens that was merged. The algorithm should stop when reaching the largest number of merges, which is represented be the last

element in the list `M`. The function should return a list `m2f` of tuples, where each tuple records in the first element the number of merges, and in the second element the corresponding frequency.

[22]:
```
def bpe_train_exp(corpus_filename, M):
    # Empty vocabulary, empty set of merge rules, empty result.
    V, rules, m2f = set([]), [], []

    # YOUR CODE HERE









    return V, rules, m2f
```

[ ]:
```
print("--- bpe_train_exp ---")
V, rules, m2f = bpe_train_exp(corpus_file, M = [100, 200, 300, 400, 500, 750,␣
 ↪1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000])
print(m2f) # You should obtain the list [(100, 1512), (200, 650), (300, 419),␣
 ↪(400, 294), (500, 219), (750, 126), (1000, 85), (1500, 52), (2000, 36),␣
 ↪(2500, 27), (3000, 21), (3500, 17), (4000, 15), (4500, 13), (5000, 11)]
```

[28]:
```
import matplotlib.pyplot as plt

freqs = [e[1] for e in m2f]
merges =[e[0] for e in m2f]
plt.scatter(merges, freqs, c='#1f77b4', alpha = 0.5)
plt.show()
```



2.4 Token distributions in Moby Dick (30 points)

Write a function `token_distribution(corpus, K)`, where `corpus` is the name of a file that contains a text corpus, and `K` is the number of merges in a BPE tokenizer. The function does the following: 1. It trains a BPE tokenizer for `K` merges on the lowercased version of the text in `corpus`.

2. It uses the trained tokenizer to tokenize the lowercased version of the text in `corpus`.
3. It creates a dictionary mapping each unique token that has at least 2 non-whitespace characters to its frequency in the corpus.
4. It returns a list of tuples (token, frequency) containing all the items in the dictionary above, sorted in decreasing order of frequency.

[ ]:
```
def token_distribution(corpus, K):
    tf = []
```

```
    # YOUR CODE HERE




    return tf
```

2.4.1 Plot the frequency vs. rank for the token distribution

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

corpus_file = '../data/melville-moby_dick.txt'
tf = token_distribution(corpus_file, 5000)

topK = 75
ranks = range(1, topK + 1)
freqs = [e[1] for e in tf[:topK]]

plt.scatter(ranks, freqs, c='#1f77b4', alpha=0.5)
plt.show()
```

2.5 Analysis and insights (10 points)

Include a nicely formatted analysis of the results that you obtained in the experiments above, and any additional experiments you run.

2.6 Bonus points

Anything extra goes here. For example:

1. Train the BPE algorithm on the first 1 billion characters of the English Wikipedia dump. If that is too time consuming, train it on the first 10 or 100 million of characters form the same dataset. Evaluate and compare the tokenization produced on various examples.
2. Write code Li (1992) showing that just random typing of letters including a space will generate "words" with a Zipfian distribution. Generate at least 1 million characters before your compute word frequencies.

- Show mathematically that random typing results in a Zipf's distribution by computing probabilities for all words that contain just 1 letter, 2 letters, …

3. Implement the WordPiece algorithm using the greedy criterion shon on the slides, and compare its tokenization with the one does by the BPE algorithm.
4. It was proposed in class that once the vocabulary is created by the BPE training algorithm, we could discard the merge rules and tokenize each string by greedily identifying the largest token in the vocabulary that is contained in the string, then repeat the procedure with the

rest of the string. Implement this procedure and compare its output with the output of the regular BPE tokenizer.

5. Compute the token distributions in Moby Dick using the `tiktoken` tokenizer of GPT-4.

[ ]: