Natural Language Processing Assignment # 2

Sentence Segmentation in Roman Urdu and Byte Pair Encoding Tokenizer

Student Name: Moeed Asif

Roll No: 21i-0483

Section : CS-A

Introduction:

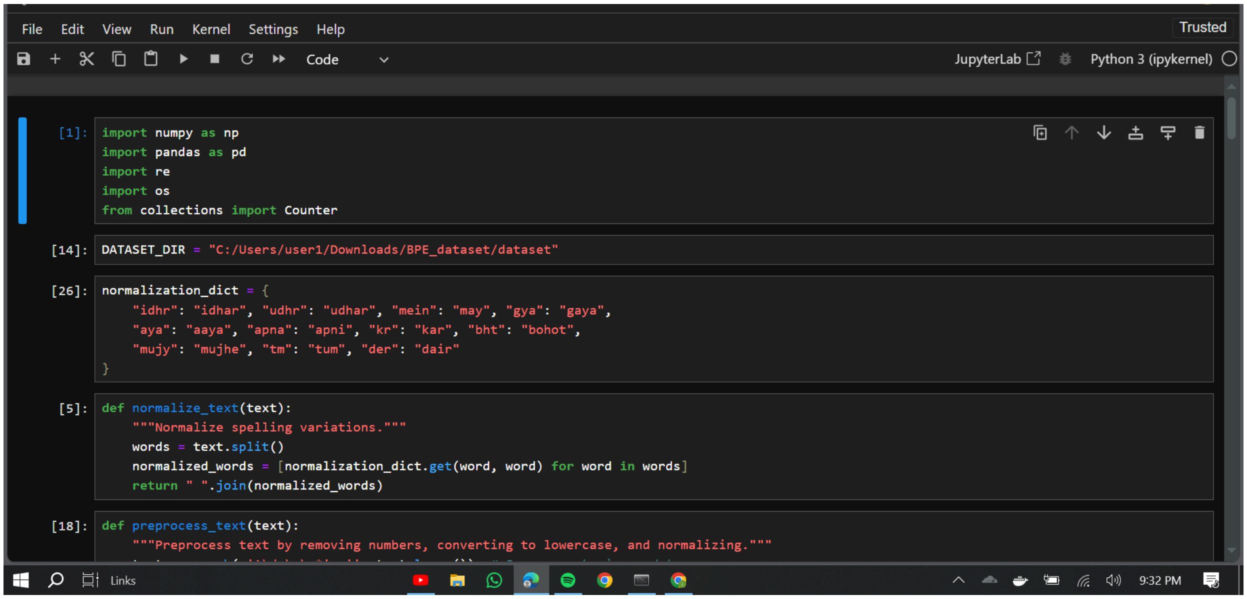
In this task, we need to implement Byte Pair Encoding (BPE) from scratch in Python to tokenize Roman Urdu diary entries. The aim is to preprocess the text, which reduces it to a unique set of characters, hence providing the basic vocabulary he used, and then iteratively to pair together the most frequent characters until he has a vocabulary of 1000 tokens. The resulting BPE model should then be applied to a separate set of 14-day diary entries to evaluate the effectiveness of vocabulary size reduction and coverage of Out-of-Vocabulary word. Some of the key challenges include identifying

Objectives:

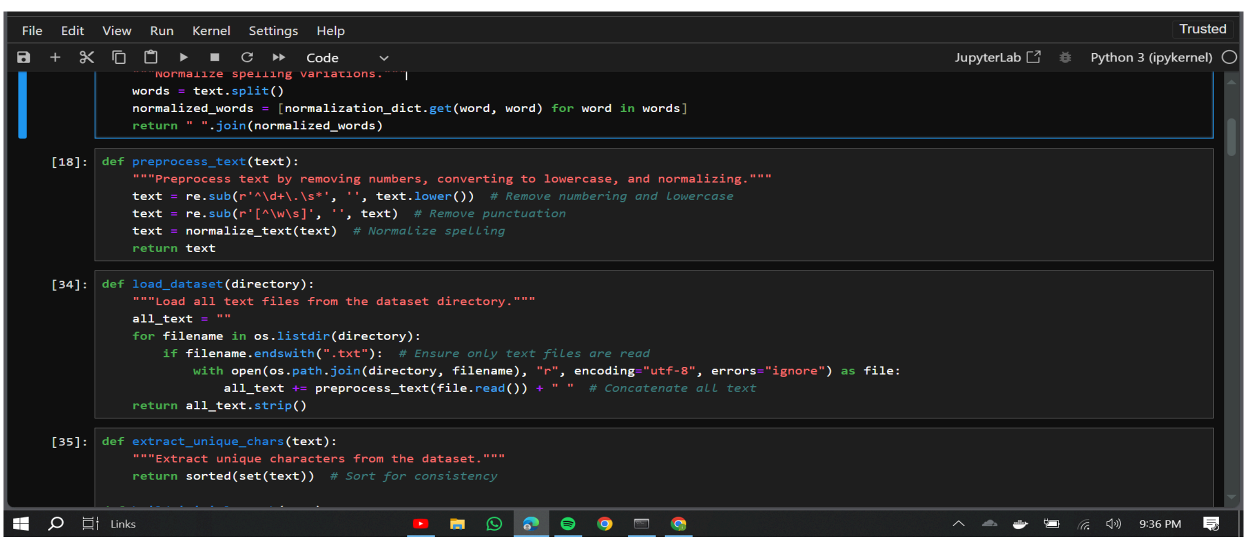
1. **Developed a BPE Tokenizer:** Implement Byte Pair Encoding from scratch in Python to tokenize Roman Urdu text effectively.
2. **Preprocess Text Data:** Convert text to lowercase, remove punctuation, and normalize spelling variations.
3. **Build Initial Vocabulary:** Extract unique characters from the dataset and assign unique IDs, starting **from 1, with <UNK> assigned to 0.**
4. **Train the BPE Model:** Learn frequent subword patterns from the dataset and iteratively merge the most common character pairs to reach a vocabulary size of **1000 tokens.**
5. **Test on Unseen Data:** Evaluate the trained model on a **14-day diary dataset,** identifying how many words remain unknown (<UNK> tokens).
6. **Reduce Out-of-Vocabulary (OOV) Words:** Improve tokenization efficiency by minimizing unknown words and ensuring better handling of spelling variations in Roman Urdu.
7. **Assess Model Performance:** Measure vocabulary reduction, token splits, and the model's ability to generalize on unseen text.

Implementation:

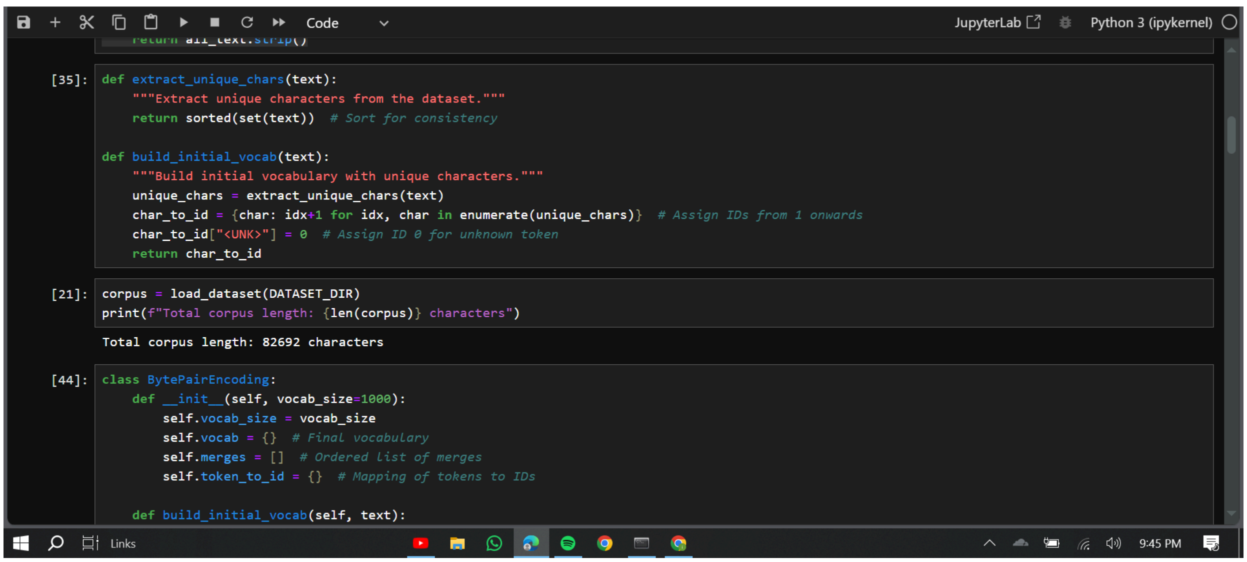
This code sets the dataset directory path and defines a **normalization_dict** to standardize common spelling variations in Roman Urdu. The **normalize_text** function processes a given text by splitting it into words and replacing any found in the dictionary with their standardized forms. If a word is not in the dictionary, it remains unchanged. This helps maintain consistency in text preprocessing, making it easier for further analysis, such as tokenization in Byte Pair Encoding (BPE) as shown in the screenshot below.



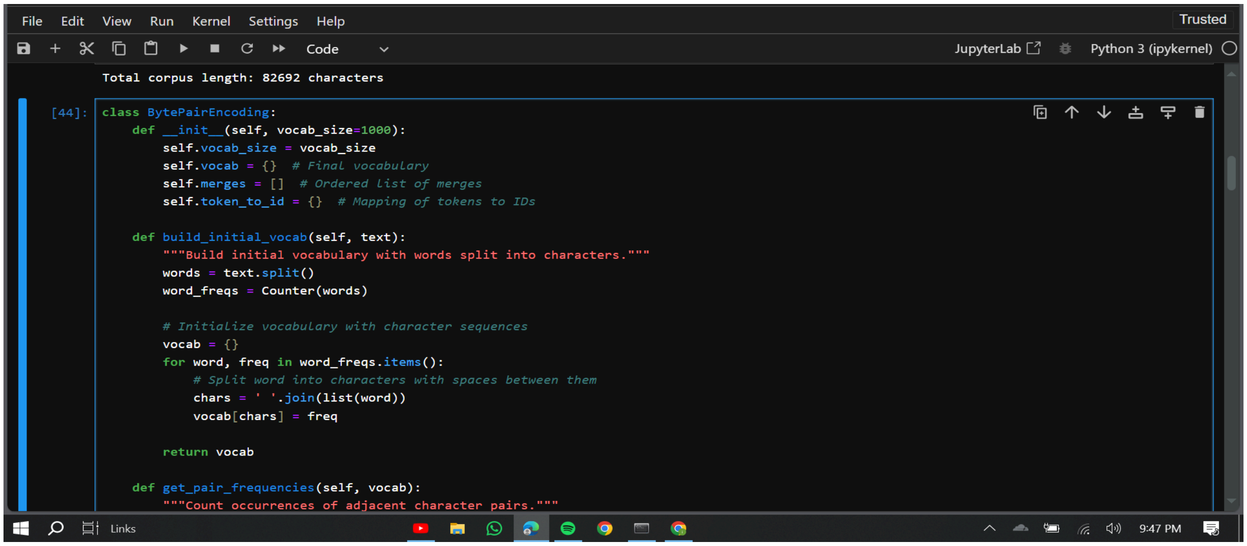
This code defines two functions for text preprocessing and dataset loading. The **preprocess_text** function removes numbering, converts text to lowercase, eliminates punctuation, and applies normalization using the normalize_text function. The **load_dataset** function reads all .txt files from a specified directory, processes their content using preprocess_text, and combines them into a single text string. This ensures the dataset is cleaned and standardized before further processing.



The code below defines two functions for handling character-level vocabulary in text processing. The extract_unique_chars function retrieves all unique characters from the dataset and sorts them for consistency. The build_initial_vocab function creates a character-to-ID mapping, assigning unique IDs to each character while reserving ID 0 for an unknown token (<UNK>). This helps in encoding text data for further processing, such as tokenization in a Byte Pair Encoding (BPE) algorithm.



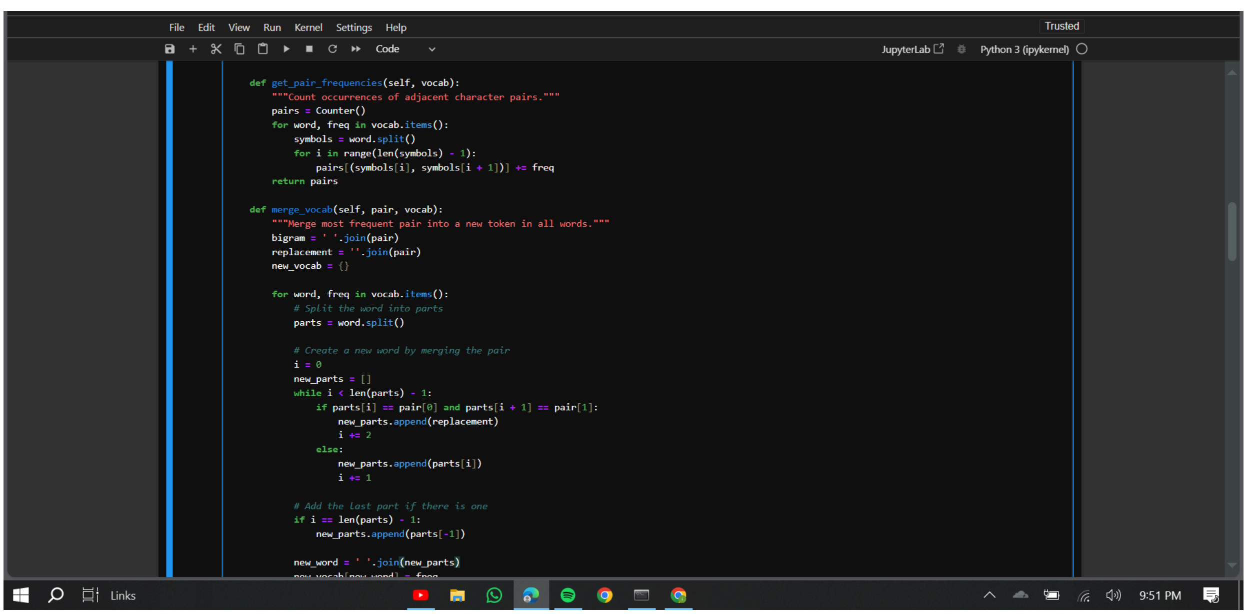
This piece of code initializes a Byte Pair Encoding (BPE) model and builds the initial vocabulary by splitting words into character sequences. The **__init__** method sets up the vocabulary size, final vocabulary, merge history, and token-to-ID mapping. The **build_initial_vocab** method processes the text by counting word frequencies and representing each word as a sequence of individual characters, forming the starting vocabulary for the BPE algorithm.



```
Total corpus length: 82692 characters
```

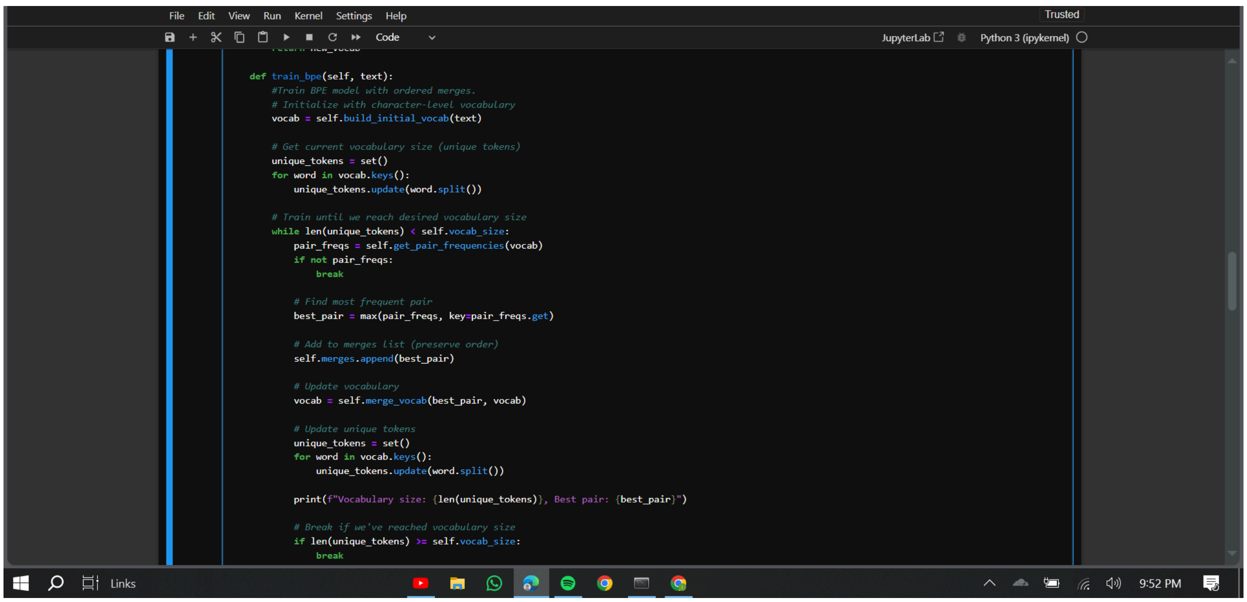
```python
class BytePairEncoding:
    def __init__(self, vocab_size=1000):
        self.vocab_size = vocab_size
        self.vocab = {}  # Final vocabulary
        self.merges = []  # Ordered list of merges
        self.token_to_id = {}  # Mapping of tokens to IDs

    def build_initial_vocab(self, text):
        """Build initial vocabulary with words split into characters."""
        words = text.split()
        word_freqs = Counter(words)

        # Initialize vocabulary with character sequences
        vocab = {}
        for word, freq in word_freqs.items():
            # Split word into characters with spaces between them
            chars = ' '.join(list(word))
            vocab[chars] = freq

        return vocab

    def get_pair_frequencies(self, vocab):
        """Count occurrences of adjacent character pairs."""
```

This code implements key functions for the Byte Pair Encoding (BPE) algorithm. The `get_pair_frequencies` method counts occurrences of adjacent character pairs in the vocabulary. The `merge_vocab` method merges the most frequent character pair into a new token throughout the vocabulary, updating word representations. This process iteratively refines the vocabulary by forming increasingly larger subword units, improving text tokenization efficiency

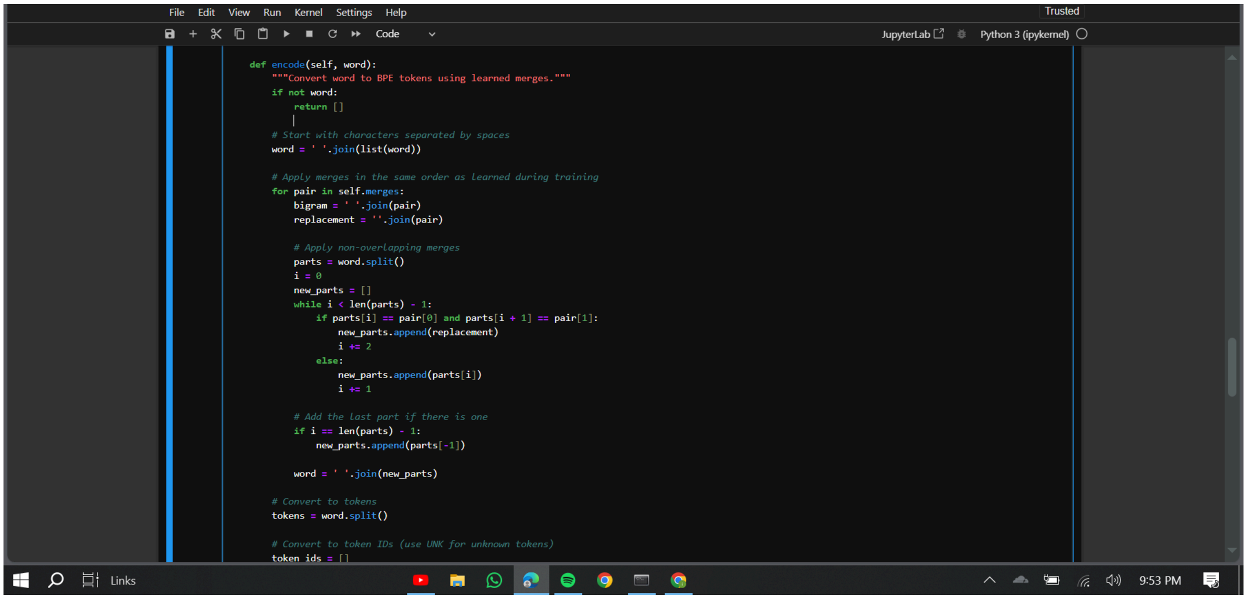


```python
    def get_pair_frequencies(self, vocab):
        """Count occurrences of adjacent character pairs."""
        pairs = Counter()
        for word, freq in vocab.items():
            symbols = word.split()
            for i in range(len(symbols) - 1):
                pairs[(symbols[i], symbols[i + 1])] += freq
        return pairs

    def merge_vocab(self, pair, vocab):
        """Merge most frequent pair into a new token in all words."""
        bigram = ' '.join(pair)
        replacement = ''.join(pair)
        new_vocab = {}

        for word, freq in vocab.items():
            # Split the word into parts
            parts = word.split()

            # Create a new word by merging the pair
            i = 0
            new_parts = []
            while i < len(parts) - 1:
                if parts[i] == pair[0] and parts[i + 1] == pair[1]:
                    new_parts.append(replacement)
                    i += 2
                else:
                    new_parts.append(parts[i])
                    i += 1

            # Add the last part if there is one
            if i == len(parts) - 1:
                new_parts.append(parts[-1])

            new_word = ' '.join(new_parts)
```

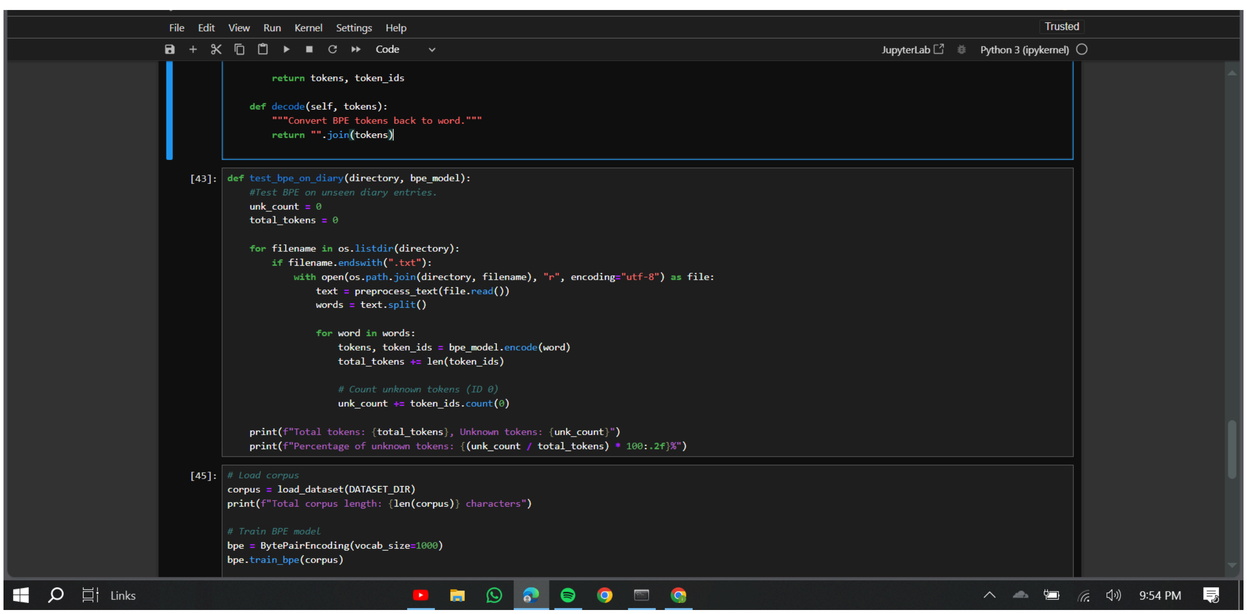
The `train_bpe` function trains a Byte Pair Encoding (BPE) model by iteratively merging the most frequent adjacent character pairs in the text until the vocabulary reaches the desired size. It maintains an ordered list of merges, updates the vocabulary after each merge, and constructs a token-to-ID mapping. The function ensures that single characters are added first, followed by merged tokens, creating an optimized subword representation for text tokenization.



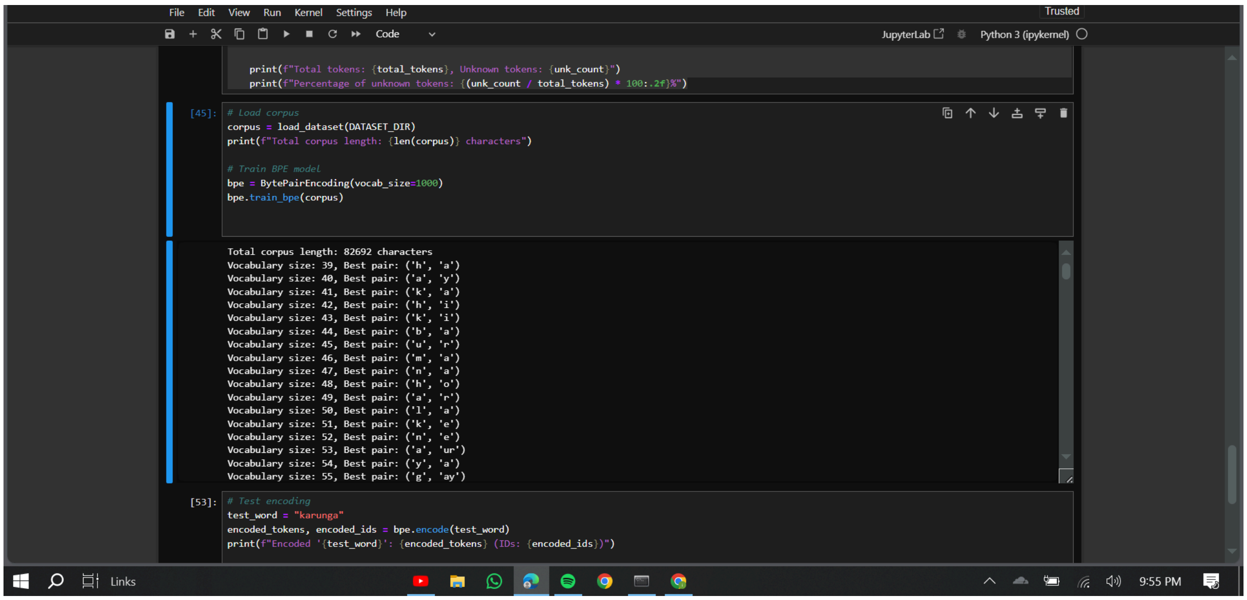
The encode function tokenizes a given word using the trained BPE model by applying learned merges in order. It starts by splitting the word into individual characters, then iteratively merges frequent character pairs according to the training process. Finally, it converts the resulting subword tokens into their corresponding token IDs, using an <UNK> token ID for any unseen tokens. This ensures consistent tokenization for text processing tasks.



The decode function reconstructs words from BPE tokens by simply joining them. The test_bpe_on_diary function evaluates the trained BPE model on unseen diary entries by tokenizing words and counting the total tokens and unknown tokens (mapped to ID 0). It then calculates the percentage of unknown tokens to assess the model's effectiveness in handling new text.



This code below in the screenshot loads a text dataset from the specified directory and preprocesses it into a single corpus. It then initializes a Byte Pair Encoding (BPE) model with a vocabulary size of 1000 and trains it on the corpus to learn token merges, creating an optimized vocabulary for text compression and representation.



This code tests the trained Byte Pair Encoding (BPE) model by encoding and decoding a sample word ("karunga") to verify its tokenization process. It then evaluates the model on unseen text files from a specified directory, measuring the percentage of unknown tokens to assess how well the BPE vocabulary generalizes to new data.

File Edit View Run Kernel Settings Help

Trusted

Code

JupyterLab Python 3 (ipykernel)

```
Vocabulary size: 69, Best pair: ('t', 'h')
Vocabulary size: 70, Best pair: ('t', 'ha')
Vocabulary size: 71, Best pair: ('u', 'n')
Vocabulary size: 72, Best pair: ('a', 'd')
Vocabulary size: 73, Best pair: ('c', 'h')
Vocabulary size: 74, Best pair: ('e', 'r')
Vocabulary size: 75, Best pair: ('s', 'i')
Vocabulary size: 76, Best pair: ('gay', 'a')
Vocabulary size: 77, Best pair: ('l', 'e')
Vocabulary size: 78, Best pair: ('r', 'a')
```

[53]:

```
# Test encoding
test_word = "karunga"
encoded_tokens, encoded_ids = bpe.encode(test_word)
print(f"Encoded '{test_word}': {encoded_tokens} (IDs: {encoded_ids})")

# Test decoding
decoded_word = bpe.decode(encoded_tokens)
print(f"Decoded '{encoded_tokens}': {decoded_word}")

# Test on unseen data
TEST_DIR = "C:/Users/user1/Desktop/NLP_A1"
test_bpe_on_diary(TEST_DIR, bpe)
```

```
Encoded 'karunga': ['karun', 'ga'] (IDs: [467, 326])
Decoded '['karun', 'ga']': karunga
Total tokens: 3625, Unknown tokens: 2
Percentage of unknown tokens: 0.06%
```

[ ]:

Links 9:56 PM