NLP Project

Primary Focus: Machine Translation (Kannada to English, and vice versa):

Currently, we are focusing on DATA-PREPROCESSING and REQUIRED CHANGES TO MAKE before sending the data into the model.

This is done in 3 steps:

1. Preprocessing the text - This part is essentially doing the cleaning of the text before we perform any other steps in it.
    - This is important so that we can bring in CONSISTENCY in the data.
2. Tokenizing the texts - Once the text is cleaned then we will perform tokenization. This refers to breaking the text into smaller units (Tokens).
    - This allows the model to understand the building blocks of the language it is processing, making it easier to translate each part accurately.
    - creates a vocabulary of unique tokens from the training dataset.
    - In models like Transformers, tokenization allows the model to focus on relevant parts of the input when generating translations.
3. Encoding the texts - This refers to converting the tokens into numerical representations.
    - Machine learning models, including neural networks, can only process numerical data. Encoding is essential for allowing the model to work with textual input effectively.
    - Encoding allows the model to create a vocabulary of known tokens, mapping each token to a unique numerical value.

**** For our case, we will be converting each word into vector representation (this is called text vectorization). Keep in mind, encoding is the superset of text vectorization. They are not the same, because encoding can be normal**

integers too, not necessarily being vectors. However I will be using them inter-changeably.

Therefore the flow will be:

PREPROCESSING —> TOKENIZATION —> ENCODING (text vectorization)

**cleaner text will be easier to be tokenized, hence first we will preprocess and clean the text, then we will tokenize it and then vectorized it.

PREPROCESSING APPROACH:

1. Lowercasing

- **What it does**: Converts all characters in the text to lowercase.
- **Why**: This helps to ensure uniformity, as "Word" and "word" would be treated as the same token.
- **Example**:
  - Input: `"The Quick Brown Fox"`
  - Output: `"the quick brown fox"`

2. Removing Punctuation

- **What it does**: Eliminates punctuation marks from the text.
- **Why**: Punctuation can interfere with tokenization and doesn't add meaning in many cases.
- **Example**:
  - Input: `"Hello, world! How's it going?"`
  - Output: `"Hello world Hows it going"`

3. Removing Special Characters

- **What it does**: Deletes any characters that aren't letters or numbers (like @, #, $, etc.).

- **Why**: These characters often don't add value and can complicate tokenization.
- **Example**:
  - Input: `"#Amazing@Day$2024!"`
  - Output: `"AmazingDay2024"`

4. Expanding Contractions

- **What it does**: Changes contractions to their full forms.
- **Why**: This helps ensure that the model understands the words better.
- **Example**:
  - Input: `"I can't believe it's true."`
  - Output: `"I cannot believe it is true."`

5. Removing Stop Words

- **What it does**: Eliminates common words that may not contribute significant meaning (like "is", "the", "and").
- **Why**: This can reduce noise and improve processing efficiency.
- **Example**:
  - Input: `"The cat sat on the mat."`
  - Output: `"cat sat mat."` *(if "the" and "on" are stop words)*

6. Lemmatization/Stemming

- **What it does**: Reduces words to their base or root form.
- **Why**: This helps in treating different forms of a word as the same (e.g., "running", "ran", "runs" all relate to "run").
- **Example**:
  - Input: `"running runner ran"`
  - Output: `"run run run"`

7. Handling Numbers

- **What it does**: Decides how to treat numerical values (keep them, convert to words, or remove).
- **Why**: Depending on the context, you may want to keep, replace, or remove numbers.
- **Example**:
  - Input: `"I have 2 cats and 1 dog."`
  - Output: `"I have two cats and one dog."` *(if converting to words)*

8. Removing Extra Whitespace

- **What it does**: Trims extra spaces, tabs, or line breaks.
- **Why**: This cleans up the text, making it uniform.
- **Example**:
  - Input: `"Hello world! How are you? "`
  - Output: `"Hello world! How are you?"`

Additionally, we can add: ***POS TAGGING (parts of speech tagging) and PADDING THE SENTENCES.***

POS tagging can be very useful for enhancing the understanding of sentence structure and context in machine translation, which can lead to better translations. However, it does introduce complexity into the preprocessing pipeline

Padding the sentence is required to ensure all the sentences are of equal length. Generally what we do is we take the required length of the sentence to be the length of the longest sentence, and then we just PAD the remaining ones,

TOKENIZATION APPROACH:

We will perform subword tokenization, and use byte-pair encoding to perform that

- Byte Pair Encoding (generally referred to as BPE) is different from the encoding (vectorization we learned earlier.)
    - **BPE** is concerned with how text is split into manageable subword tokens, while **encoding** focuses on transforming those tokens into numerical formats that models can process.

WHY BPE :

If you implement your own tokenizer using Byte Pair Encoding (BPE), you can generate tokens for any language, including languages with different scripts, grammatical structures, and characteristics. This is because of the way in which BPE generates its tokens (see below, I have attached token generation for both kannada and english)

The tokenizer can break OOV words down into subword tokens based on the frequency of character pairs. For instance, if the word "ನೆನೆಸುವುದು" (to reminisce) is not in your vocabulary, the tokenizer may break it down into smaller known components like "ನೆನೆ" (reminisce) + "ಸು" (to).

Overall it will look something like this:

- Original sentence: "I love natural language processing."
- BPE generates tokens: `["I", "lov", "e", "natural", "language", "processing"]`.

HOW IT HAPPENS:

1. **Initial Tokenization**:
    - Start by breaking the text down into individual characters.
    - For simplicity, let's break it down into characters, including spaces. We'll add a space between each character for clarity.

```
Original Text: "I love natural language processing."
Initial Tokens: ['I', ' ', 'l', 'o', 'v', 'e', ' ', 'n',
'a', 't', 'u', 'r', 'a', 'l', ' ', 'l', 'a', 'n', 'g',
```

```
'u', 'a', 'g', 'e', ' ', 'p', 'r', 'o', 'c', 'e', 's',
's', 'i', 'n', 'g', '.']
```

2. **Count Frequency of Pairs**:
   - Count the frequency of adjacent character pairs (bigrams). For example, "I " (the space after "I"), " l", "lo", "ov", etc.
   - Here are some example pairs with their counts:

```
('I', ' ') → 1
(' ', 'l') → 1
('l', 'o') → 1
('o', 'v') → 1
('v', 'e') → 1
('e', ' ') → 1
(' ', 'n') → 1
('n', 'a') → 1
...
('g', 'e') → 1
('e', ' ') → 1
(' ', 'p') → 1
('p', 'r') → 1
...
```

3. **Merge Most Frequent Pair**:
   - Identify the most frequent pair of characters and merge them into a single token. In this case, let's assume that the most frequent pair is `('l', 'o')`.
   - After merging, the token list will change.

```
After merging 'l' and 'o':
Tokens: ['I', ' ', 'lo', 'v', 'e', ' ', 'n', 'a', 't',
'u', 'r', 'a', 'l', ' ', 'l', 'a', 'n', 'g', 'u', 'a',
'g', 'e', ' ', 'p', 'r', 'o', 'c', 'e', 's', 's', 'i',
'n', 'g', '.']
```

4. **Repeat the Process**:

- Continue counting pairs and merging the most frequent pairs iteratively.
- For example, after merging `('l', 'a')` and `('n', 'g')`, the process continues until a set number of merges is achieved or no more frequent pairs exist.

5. **Final Tokens**:
   - After several iterations, the final tokens may look like this (the exact tokens depend on the frequency of pairs):

```
Final Tokens: ['I', 'lov', 'e', 'natural', 'language',
'processing']
```

How BPE Works for Kannada

1. **Input Text**:
   - Start with a Kannada sentence. For example, "ನಾನು ನಿಂತಿರುವೆನು" (I am standing).
2. **Initial Tokenization**:
   - Break down the text into characters. In Kannada, characters can include consonants, vowels, and diacritics.

```
less
Copy code
Initial Tokens: ['ನ', 'ಾ', 'ಂ', 'ಉ', ' ', 'ನ', 'ಿ',
'ಂ', 'ಟ', 'ಿ', 'ರ', 'ು', 'ವ', 'ೆ', 'ನ', 'ು']
```

3. **Count Frequency of Pairs**:
   - Count the frequency of adjacent character pairs (bigrams). This is important because the goal is to find and merge the most frequent subword structures.

```
lua
Copy code
Example pairs might include:
```

```
('ನ', 'ಾ'), ('ಾ', 'ಂ'), ('ಂ', 'ು'), ('ನ', 'ಿ'), etc.
```

4. **Merge Most Frequent Pair**:
   - Identify the most frequent pair and merge them into a new token.
   - For instance, if ('ನ', 'ಾ') is the most frequent pair, it may be merged to form a token like "ನಾ".

```
arduino
Copy code
After merging 'ನ' and 'ಾ':
Tokens: ['ನಾ', 'ಂ', 'ಉ', ' ', 'ನ', 'ಿ', 'ಂ', 'ಟ', 'ಿ',
'ರ', 'ು', 'ವ', 'ೆ', 'ನ', 'ು']
```

5. **Repeat the Process**:
   - Continue counting pairs and merging them until you reach a specified number of merges or until no more frequent pairs exist.
6. **Final Tokens**:
   - After several iterations, you might end up with meaningful tokens like:

```
less
Copy code
Final Tokens: ['ನಾನು', 'ನಿಂತ', 'ಿರುವೆನು']
```

AFTER WE HAVE TOKEINZED THE TEXT, WE WILL GENERATE THE VOCABULARY, WHERE THE VOCABULARY WILL CONSIST OF ALL THE UNIQUE TOKENS WE HAVE GENERATED.

Include special tokens for handling OOV words (like `<OOV>`), padding (like `<PAD>`), and beginning/end of sequence markers (like `<BOS>`, `<EOS>`), if applicable.

VECTORIZATION APPROACH:

** REMEMBER, before we go ahead, just keep in mind, that the vectors we create for the words, are also referred to as embeddings.

There are two kinds of embeddings, ***STATIC and CONTEXTUAL embeddings.***

Here is the difference :

Static Embeddings

Description:

- Static embeddings provide a fixed vector representation for each word in the vocabulary, regardless of the context in which the word appears. For example, the word "bank" would have the same embedding whether it refers to a financial institution or the side of a river.

Pros:

- **Simplicity**: Easier to implement and understand.
- **Efficiency**: Requires less computational power and memory compared to contextual embeddings.
- **Good for Smaller Datasets**: If you have a limited dataset, static embeddings can provide a solid starting point.

Cons:

- **Context Ignorance**: Lacks the ability to capture different meanings of words based on their context, which can be critical in tasks like machine translation where context heavily influences meaning.
- **OOV Issues**: Static embeddings may struggle with out-of-vocabulary words, especially in morphologically rich languages like Kannada.

Contextual Embeddings

Description:

- Contextual embeddings provide a dynamic representation for each word, depending on the surrounding context in which it appears. For instance, the

embedding for "bank" would change based on whether it appears in a sentence about finance or nature.

Pros:

- **Context Awareness**: Can capture nuances of meaning based on context, significantly improving translation accuracy.
- **Handling OOV Words**: Often better equipped to handle OOV words since they can derive meaning from subword tokens and context.
- **Rich Representation**: Captures complex relationships between words and phrases.

Cons:

- **Complexity**: More complicated to implement and may require more extensive training data and computational resources.
- **Resource Intensive**: Requires more memory and processing power, especially for larger models like BERT or GPT.

Integration with Static and Contextual Embeddings

- **Static Embeddings with BPE**:
  - If you use BPE for tokenization and static embeddings like Word2Vec or GloVe, you can create a vocabulary that captures subwords, allowing your model to handle OOV words better. However, the static embeddings won't adapt based on context, which could be limiting for translation tasks.
- **Contextual Embeddings with BPE**:
  - When paired with contextual embeddings (like those from models such as BERT or GPT), BPE not only handles OOV but also enhances the model's ability to understand the context in which those subwords are used. This combination allows for a richer representation and more accurate translations.

THEREFORE, OUR AIM WILL BE TO GENERATE CONTEXTUAL EMBEDDINGS.