CSE556 NLP Assignment-1

Arnav Goel *2021519*
Medha Hira *2021265*
Siddharth Rajput *2021102*
Amil Bhagat *2021309*

1. TASK 1

1.1 Introduction

The Tokenizer class implements a simple version of the Byte Pair Encoding (BPE) algorithm. BytePair encoding (BPE) is a subword tokenization algorithm introduced by Sennrich et al. (2016), which iteratively merges frequent pairs of characters to create a vocabulary of subword units. The implemented Tokenizer class consists of methods to learn the vocabulary and tokenize input text using the learned rules.

1.2 Code Implementation

In the tokenizer class, we have initialized

- **the *vocabulary***, a dictionary which would hold the current vocabulary for the corpus along with their respective frequencies of occurrence in the corpus.
- **the *bpe rules*,** a list that will store the pairs of tokens merged during the BPE process. These are the "rules" learned by the tokenizer and can be applied to new text. The number of rules depends on the number of merges allowed by the user
- ***old vocabulary***, a set that keeps track of unique characters (and later, unique tokens) observed in the input data. This is used for tracking the initial vocabulary before any merges.

Methods:-

- **get_vocab():-**
  The get_vocab method processes the text to initialize the vocabulary. It performs several preprocessing steps such as lowercasing, stripping newlines and removing all punctuations. A space separates each character in a word, and an end-of-word symbol($)

is appended. The resulting tokens are added to the vocab dictionary with their frequencies. Unique characters are also added to self.old_vocab to keep track of the initial character set before any BPE merges.

- **get_stats()**
  The get_stats method computes the frequency of adjacent pairs of tokens across the entire vocabulary. It returns a dictionary ('pairs') where keys are tuples representing token pairs, and values are the cumulative frequencies of these pairs in the vocabulary. This information is crucial for identifying the most frequent pairs to merge during the BPE algorithm.

- **merge_vocab()**
  The merge_vocab takes a pair of tokens('pair'). It then initialises v_out, which will hold the updated vocabulary after the specified pair of tokens has been merged. We then join the two tokens in 'pair' with a space between them, forming a string that represents the pair as it would appear in the tokenized vocabulary. Re.escape() is used to escape any characters in this string that could have special meanings in regex.

  - ***p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')***
    Compiles a regular expression pattern that matches the specified bigram only when it occurs as a whole token within a word. The pattern uses lookbehind (?<!\S) and lookahead (?!\S) assertions to ensure that the bigram is not preceded or followed by any non-whitespace characters (\S stands for any non-whitespace character). This effectively isolates the bigram as an independent sequence of tokens for accurate matching.

  We then iterate over each word in the current vocabulary and use the compiled regex pattern 'p' to search for the 'bigram' within each word. We use the sub() method to replace every occurrence of the 'bigram' with the merged version of the pair. The already existing vocabulary is then updated with the new, merged tokens.

- **learn_vocablury()**
  The learn_vocablury method in the Tokenizer class initializes the vocabulary using the get_vocab method. Then it iterates over a specified number of merges to learn split rules and frequencies through the BytePair encoding (BPE) algorithm. In each merge iteration, it calculates the frequency of adjacent token pairs with get_stats, merges the most frequent pair with merge_vocab, and updates the vocabulary. This iterative process

continues until either no more pairs are found or the specified number of merges is reached, ensuring effective learning of split rules tailored to the provided data.

- **tokenize()**
  Given a sample text, this method tokenizes it according to the learned BPE rules. If no bpe_rules are learned, we fall back and return the sample as it is. We preprocess the text here in a similar way as we do in the get_vocabulary method. Now, as we iterate over each word in the preprocessed corpus, we check for all the bpe_rules learnt and if in the word that we are iterating over, we replace the found tokenizable pair with the learnt merged tokens for the same. We keep adding the merged tokens and return them.

- **print_merge_rules()**
  The print_merge_rules method in the Tokenizer class writes the learned merge rules to a file specified by filename, with each rule in the format "token1,token2". It iterates over bpe_rules and writes each rule to the file along with its sequential index.

- **print_tokens()**
  The print_tokens method in the Tokenizer class writes all learned tokens to a file specified by filename. It iterates over the bpe_rules list to add merged token pairs to old_vocab, sorts the tokens, and writes them to the file. This ensures all tokens are present in the output file for reference.

- **Main Function**
  The main function reads the corpus from a file, initializes a Tokenizer object, learns the vocabulary and split rules, and prints the merge rules to a file named merge_rules.txt. It generates tokens and saves them to tokens.txt, then tokenizes samples, converts them to lowercase, and saves them to tokenized_samples.txt.

2. TASK 2

2.1 Implementing a Bigram Language Model from Scratch:

INITIALIZATION:

In the BigramLM class, we have initialized bigram frequency, total frequency, a set containing the vocabulary, vocabulary, and corpus size, a little of the corpus, and one of the labels.

PREPROCESS TEXT:
We have also created a function to preprocess the text. This is done by lowercasing the text and removing the non-alphanumeric characters from the text.

TRAIN LANGUAGE MODEL:
Next is a function used to train the Bigram language model. After opening the corpus and the label files, we allow the files to be read. Here, we parse through the lines, preprocess it to generate tokens and append it to the corpus. The labels are also accurately appended in the labels list.
The vocab is a set that consists of all the words present in the corpus; further, vocab_size is the size of the **vocabulary set,** as mentioned earlier. The corpus size is the size of the aforementioned list (corpus).
For each sentence in the corpus and all the works in the sentence, we create a bigram. The bigram consists of the current and next word (aka 2 consecutive words constitute a bigram). The frequency of this bigram is updated, and so is the frequency of the current word (in unigram frequency).
We have handled the end of sentence and start of sentence. **We add <end> and <start> tokens.**
This means that the first word of the sentence can have a bigram: {**<start>, <1st word of sentence>}** and the last word of the sentence has a bigram: **{ <last word of the sentence>, <end>}**.

SAVING THE MODEL:
This function saves the bigram LM in a pickle file.

LOAD MODEL:
This function loads the model from the pickle file.

BIGRAM PROBABILITY:
If a bigram is not present in the corpus, we return zero. Otherwise, we output the frequency of the bigram ({first word, second word}) divided by the total frequency of the first word.

The problem is that when 0 is multiplied by a number, we get the output as zero, and the penalty is very heavy. (When using the Bayes rule)

2.2 Implementing Smoothing Algorithms.

BIGRAM PROBABILITY USING LAPLACE SMOOTHING:

bigram_freq_smoothed = self.bigram_freq.get(bigram, 0) + 1

This means that if "bigram" is not a key in the dictionary "bigram_freq", then we get 0+1. Here, we focus on allocating a frequency of at least one to all the incoming bigrams.

previous_word_freq_smoothed = self.total_freq.get(previous_word, 0) + self.vocab_size

Here, we add "vocab_size" as the frequency of all the bigrams has increased by 1.

KNESER NEY SMOOTHING:

bigram = (previous_word, next_word)

- The continuation count is the sum of all bigrams where the second word is next_word.
- Unique continuation is a set of all possible prev and next_word tokens.

continuation_probability = max((continuation_count - d) / unique_continuations, 0)

- The standard definition of continuation count (C(*, w)) refers to the number of distinct preceding words that the next_word has appeared with.
- Lambda_context is the multiplication of the discount (a value between 0 and 1), continuation count divided by the unigram count.
- Discounted is the max of count_bigram - d and 0. Thus:

*final probability = discounted + lambda_context * continuation_probability*

Comparing the probabilities obtained from both and giving an argument for which is better:
We see a significant probability difference between Kneser-Ney and Laplace smoothing for the same bigram. This reflects the different approaches taken for these methods.

Kneser-Ney Smoothing: This method is particularly effective for natural language processing because it considers not only the frequency of the bigram itself but also the distinctiveness of the bigram's second element following other elements. Kneser-Ney smoothing would assign a higher probability to this bigram, even if the specific combination of (words even if it is not frequently observed). The probability can be quite high, as it effectively leverages the broader context of two words being a common continuation.

Laplace Smoothing (Add-One Smoothing): This method adds a uniform count to all bigrams, including those not observed in the dataset, to avoid zero probabilities. It's a simpler method that does not account for the context-specific prevalence of individual words. Thus, for a bigram that is not very common in the presence of a large vocabulary, the probability adjustment by Laplace smoothing can be relatively small, leading to a low probability estimate.

Kneser-Ney's probability reflects its better handling of language's contextual nature, making it generally more effective for language modelling, especially in predicting the likelihood of word sequences in natural language use.

2.3 Emotion Task

- The function ***"get_emotion_score_bigram"*** has a dictionary to store the bigram probabilities and an emotion probability dictionary, which stores the emotion score for a given bigram string. While making the bigram string, care is taken while processing the start and end tokens.
- ***"combine_pt_and_emotion"*** combines the bigram probabilities and emotion scores. The returned combined_prob_dict = {"joy": {}, "love": {}, "sadness": {}, "surprise": {}, "fear": {}, "anger": {}}
- We add the bigram probability and the respective emotion scores.

- The combined_prob_dict[curr_emo][prev_word][next_word] = the final calculated probability.
- Next, we have created a function called: ***"normalise_combined_prob,"*** this takes the combined dictionary returned by ***"combine_pt_and_emotion"*** as the input and normalizes the probabilities. The dictionary returned by this function will be used further in the code.
- The ***"generate_sentence"*** function generates a sentence of a max length of 15 or the end of the sentence for emotion given using the bigram model (minimum length = 7).
- We initialize our list (this will contain all the tokens/words that will make up the final sentence) with the <start> token to mark the beginning of the sentence (we can all use this to search for all the bigrams of type: {<start>, 1st word of a sentence}).
- If the sentence is just being started, we can randomly choose any of the bigrams where <start> is the first word.
- Otherwise, we get a list of all possible next words; we use the bigram probabilities to choose the top k of these words. In this case, if we encounter an end-of-sentence token, we ignore it if the minimum length is not reached; otherwise, we pick it up and end our sentence. In a case where there are no more probable words, we break.
- After getting the top n words (n = min(k, len(of all the next possible words), we store the words and their respective probabilities. If the picked word is <end>, we can return the made sentence; else, update the prev_word to be the one just picked.
- This process continues and makes the complete sentence.
- We generate 50 sentences for each emotion and save them to a unique file called "gen_<emotion>.txt".
- For each file, we generate a histogram of the emotion scores to verify our generated samples and their scores.

2.4 Extrinsic Evaluation

A. Please see the code for the generation method and see the txt files.

B. We define a **SVC Class** and **define a parameter grid**:

```python
# Define the parameters for the grid search
param_grid = {'C': [0.1, 1, 10, 100],
              'gamma': [1, 0.1, 0.01, 0.001],
              'kernel': ['rbf', 'linear', 'poly', 'sigmoid']}

# Grid Search on the SVC model
grid = GridSearchCV(SVC(), param_grid, refit=True, verbose=2, cv=5, n_jobs=-1)
```

param_grid = {'C': [0.1, 1, 10, 100],
'gamma': [1, 0.1, 0.01, 0.001],
'kernel': ['rbf', 'linear', 'poly', 'sigmoid']}

- We employ **5-fold cross-validation** and train it to get the best hyperparameters as follows:
{'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}

C. Please take samples

2.5 Final Evaluation

1. Top 5 Bigrams Before Smoothing and after Applying the 2 smoothing techniques:

```
Top 5 most probable bigrams without smoothing:
('href', 'http') :  1.0
('mooshilu', '<end>') :  1.0
('tychelle', 'to') :  1.0
('hang', 'out') :  1.0
('nonexistent', 'social') :  1.0
--------------------

Top 5 most probable bigrams with laplace smoothing:
('<start>', 'i') :  0.2693830629710052
('i', 'feel') :  0.11043610327619874
('feel', 'like') :  0.0350976507217662
('i', 'am') :  0.03189412019960946
('<start>', 'im') :  0.02720653978796781
--------------------

Top 5 most probable bigrams with kneser-ney smoothing:
('href', 'http') :  0.98
('don', 't') :  0.9745762711864406
('didn', 't') :  0.9722222222222222
('sort', 'of') :  0.9705882352941176
('doesn', 't') :  0.9444444444444444
```

---

2. Reasoning for Method Used for Including Emotion Component:

- This method returns a dictionary combining the bigram_probabilities and emotion_probabilities from the previous method.
- The structure of the combined dictionary is: (let's say for a bigram -> (prev_word, next_word_1))

```
{"emotion":
  {"prev_word" :
    {"next_word_1" : prob_1, "next_word_2" : prob_2}},
}
```

- For getting prob_1 for any bigra, we add its bigram_probability to the emotion score of that bigram for that particular emotion, i.e.

$$P_{final|e}(w_{i-1}, w_i) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})} + P_e$$

- The above equation is our bigram-level incorporation of emotion scores where P_{e} corresponds to the emotion score returned by `utils.py` for our input bigram for emotion e.

- This is the description of our combination method from our notebook.
- The rationale is based on time and efficiency. Adding increases the weight of a particular emotion. Our beta is not some constant but actually, the emotion score generated by the function for an input bigram.

Then let's say the current sentence is: **S = x1, x2, x3**

Now for the next token, we know the emotion scores of all bigrams where the first token is x3 (we calculate this and store it as a lookup table). Then we add this score to the calculated prob P(x4 | x3). This is not normalised. We then normalise for all such x3 to get a valid probability distribution. This is then used to pick the next token x4 which will convey the emotion hopefully.

- On the unigram level, it took a lot of time to make the look-up table for each emotion. On the sample level, considering each sample's emotion of varying length would mean a lot of calls to the emotion_scores() function.
- Thus, we came upon the use of the bigram level. **Our total method runs in only 12 minutes, which is much less than that of other choices.**
- Our final emotion scores for generated sentences are, **on average, above 0.95,** which shows the accuracy of our choice. This is because of the addition of emotion scores, and then **top-k sampling for each bigram (wi-1, wi)** ensures only those words which are of that emotion of high probability are selected.
- Top-k sampling ensures diversity in the generation of sentences compared to greedy sampling. A higher k will mean more diverse (but less emotionally correct) sentences. We choose k as default 6.

---

3. Two generated samples for each emotion:

Anger:

```
i pissed and grumpy i joke about talking to talk in regards to hate people
a hostile manner now sigh i pissed my selfish bitch if it strikes your comments
```

Fear:

```
ive entered in portland land and fearful that maybe something moving inside the unseen ambient
ive accomplished the perfect fit looking out pretty confident that rich every believer should hear
i celebrate today as comfortable being comfy feels so happy
```

Joy:

Love:

```
i embrace the longing urge to touch with the longing to wanting the gentle breeze
i empathize with girly stuff and loved by beloved your sweet nostalgic lately
```

Sadness:

```
im sorry for losing again after failing to suffer i missed certain goals
i missed about life changes cause of pathetic
i forgot my grief i exhausted all unloved he gets to numbing pleasures more suffering
```

Surprise:

```
i feel amazed to be funny sensitive skin so curious to be impressed to be
i feel surprised to be amazed at all the unexpected long
```

4.

The final accuracy we get on the test set is between **75-82%**

Macro-F1 score is also between **0.75 and 0.80**

Example:

```
Accuracy of our best SVM model on generated samples is:  0.8033333333333333
--------------------

Macro F1 score of our best SVM model on generated samples is:  0.8013763227922519
--------------------

Confusion matrix of our best SVM model on generated samples is:
[[37  2  3  2  6  0]
 [ 4 35  2  2  5  2]
 [ 7  3 30  0 10  0]
 [ 1  0  0 49  0  0]
 [ 4  0  3  1 42  0]
 [ 1  1  0  0  0 48]]
--------------------

Classification report of our best SVM model on generated samples is:
              precision    recall  f1-score   support

       anger       0.69      0.74      0.71        50
        fear       0.85      0.70      0.77        50
         joy       0.79      0.60      0.68        50
        love       0.91      0.98      0.94        50
     sadness       0.67      0.84      0.74        50
    surprise       0.96      0.96      0.96        50
...
weighted avg       0.81      0.80      0.80       300

--------------------
```

---

5. Picking a sentence for emotion and rationale as to why generated:

A. Sadness: *"the hurt when you should be discouraged why i forget it broken again in vain"*

B. Fear: *"ive entered in portland land and fearful that maybe something moving inside the unseen ambient"*

C. Joy: *"i celebrate today as comfortable being comfy feels so happy"*

D. Anger: *"i pissed and grumpy i joke about talking to talk in regards to hate people"*

E. Surprise: *"i feel surprised at the amazingness of dazed from the amazing because of shocked that"*

F. Love: *“i desire and liked him i embrace the gentle breeze and caring”*

The given sentences for each emotion, have a pattern. They are centred around a word or couple of words which very strongly emote their corresponding emotion. This is because our approach captures the emotional information at the bigram level and the probabilities get influenced accordingly by capturing the context in the bigram. Thus, we see our approach capturing the emotional information very accurately in the generated sentences even though the grammar and language are a bit skewed.

---

6. Credit Statement:

While each member contributed equally to all tasks, we are mentioning a division of tasks just for the sake of the question here.

- **Medha Hira (Roll No: 2021265)** - Task 2, Subtask 1 and 2
- **Arnav Goel (Roll No: 2021519)** - Task 2, Subtask 3, Extrinsic and Final Evaluation
- **Siddharth Rajput (Roll No: 2021102)** - Task 1 and Extrinsic Evaluation
- **Amil Bhagat (Roll No: 2021309)** - Task 1 and Extrinsic Evaluation