# Natural Language Processing

Student Name:
Pouya Haji Mohammadi Gohari

SID:810102113

Date of deadline
Monday 6th March, 2023
Grace:13 hours and 36 minutes used

Dept. of Computer Engineering

University of Tehran

## Contents

## 1 Question 1

### 1.1 part 1

There are numerous tokenizer where can be helpful in NLP[1] tasks.They can be categorized as follow:

- Words-based
- Subword-based
- Character-based

The provided tokenizer is indeed a word-based.Here are some examples to demonstrate what this code accomplishes:

| Example | Tokenized |
|---|---|
| ’I love my mother so much!’ | [’I’, ’love’, ’my’, ’mother’, ’so’, ’much’] |
| ’”Dr.Khosravi said I’m weak’ | [’Dr’, ’Khosravi’, ’said’, ’I’, ’m’, ’weak’] |

More example has been implemented in code
In the observed examples,the tokenizer incorrectly splits ’Dr.Khosravi’ into two seperated words. More precisely, if we are given ’U.S.A’ to tokenize, it will divides the characters.Furtheremore, this tokenizer wrongly splits ’2024/12/22’ to incorrect form ’[’2024’, ’12’, ’22’]’.Another issue with this tokenizer is when we handling words like ’I’m’ or ’It’s’ where it will be tokenized in [’I’, ’s’] or [’It’,’s’] instead of correct forms [’I’, ’am’] or [’It’,’is’].

### 1.2 part 2

When we apply given text:
”Just received my M.Sc. diploma today, on 2024/02/10! Excited to embark on this new journey of knowledge and discovery. #MScGraduate #EducationMatters.” The tokenizer will splits the sentence to: [’Just’, ’received’, ’my’, ’M’, ’Sc’, ’diploma’, ’today’, ’on’, ’2024’, ’02’, ’10’, ’Excited’, ’to’, ’embark’, ’on’, ’this’, ’new’, ’journey’, ’of’, ’knowledge’, ’and’, ’discovery’, ’MScGraduate’, ’EducationMatters’]
As we said eariler, issues can be categorized into following table 1: In most cases of NLP tasks, URLs will be

Table 1: Categorized Issues

| Issues | Example | Tokenized |
|---|---|---|
| Dot | ”M.Sc” | [’M’, ’Sc’] |
| Date formats | ”2024/02/10” | [’2024’, ’02’, ’10’] |
| Prices | ”$299.99” | [’299’, 99] |
| URLs | http://www.stanford.edu | [’http’, ’www’, ’stanford’, ’edu’] |

typically removed in tokenization phase since they do not generally contribute valuable features for analysing.

[1]Natural Language Processing

## 1.3 part 3

We plan to modify the pattern in the provided tokenizer to address issues with periods(dot ambiguity) and date formats.Firstly, it would be helpful to have review on metacharacters in Python's regular expressions(illustrated in table 2).

Table 2: Regular Expression Metacharacters and Descriptions

| Operators | Description |
|---|---|
| \w | Matches any alphanumeric character (word character) |
| \d | Matches any digit character |
| \s | Matches any whitespace character |
| \b | Matches a word boundary |
| . (dot) | Matches any single character except newline |
| ? | Matches 0 or 1 occurrence of the pattern |
| * | Matches 0 or more occurrences of the pattern |
| + | Matches 1 or more occurrences of the pattern |
| \W | Matches any non-word character |
| \D | Matches any non-digit character |
| \S | Matches any non-whitespace character |
| \B | Matches a non-word boundary |
| [· · · ] | Matches any single character inside the square brackets |
| \ | Escapes special characters |
| ^ and $ | Matches the start and end of a string, respectively |
| {n,m} | Matches at least n and at most m occurrences |
| \| | Alternation, matches either pattern |
| () | Groups patterns |
| \t, \n, \r | Matches tab, newline, return |

Now for addressing the preriods's issue, we can simply add \b[\w.]+\b to pattern.This pattern will capture words including dots in the middle of the word, but not at the begining or end.To address date's issue we can simply use the following pattern:

$$[0-9]\{4\}/[0-9]\{2\}/[0-9]\{2\} \tag{1}$$

This means 4 digits for year,followed by 2 digits for month and also another 2 digits for the day splilited by /.In total the modified code is:

```python
    def my_tokenizer(text):
        pattern = r'[0-9]{4}/[0-9]{2}/[0-9]{2}|\b[\w.]+\b'
        return re.findall(pattern, text)
```

The given text would be: ['Just', 'received', 'my', 'M.Sc', 'diploma', 'today', 'on', '2024/02/10', 'Excited', 'to', 'embark', 'on', 'this', 'new', 'journey', 'of', 'knowledge', 'and', 'discovery', 'MScGraduate', 'EducationMatters']
As you can see the output indicates we have solved two issues(period and date).

## 2 Question 2

### 2.1 part 1

GPT use BPE[2] tokenizer.This tokenizer is a form of subword-based tokenization that starts with a large corpus of text and iteratively merges the most frequent characters or character sequnce.This process will continue till merging can create more new-word or we reach to the certain vocab size.
Example of BPE: Assume the given-corpus is:

- low _ -> 5 times
- lowest _ -> 2 times
- newer _ -> 6 times
- wider _ -> 3 times
- new _ -> 2 times

vocabulary : _, d, e, i, l, n, o, r, s, t, w.
The procdure of BPE:
Step 1 -> _, d, e, i, l, n, o, r, s, t, w, er.
Step 2 -> _, d, e, i, l, n, o, r, s, t, w, er, er_
Step 3 -> _, d, e, i, l, n, o, r, s, t, w, er, er_, ne
Step 4 -> _, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
Step 5 -> _, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, newer_
Step 6 -> _, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, newer_, low_. The findings are very fascinating, as the term 'slow' does not appear in the dataset but BPE can generate this word!

There are several resons that LLM[3]'s like GPT using BPE tokenizer:

- We can generate some words that was not included in trainin corpus.
- Subword tokenization can reduce the vocab size needed to cover the majority of the text in a language.
- Instead of needing a massive list of words, which is the case with word-based tokenization, subword tokenization manages with far fewer entries.
- This tokenizer can alse recognize these words 'friend' and 'friendly' releated to each other although they are distinct.
- Adoptability to different languages is one of the advantage of BPE.BPE can use the same subword units for different tongues. This advantage is a very good fit for models learning multiple languages at once.

BERT where was introduced by Google, use WordPiece tokenizer.It is very similar to BPE.There are several reasons to use WordPiece tokenizer:

- Like BPE, it can handle out-of-vocabulary words(unseen words during training).

[2]Byte Pair Encoding
[3]Large Language Model

- Similart to BPE, it can cover the majority of text.
- Generalizability: It has a good performance on the new text.
- It can break break down words to meaningful subwords units where maked BERT adoptable across languages.

### 2.2 part 2

GPT use BPE tokenizer algorithem, the procedure is:

- Iterative Merging: BPE starts by tokenizing text at the character level and then iteratively merges the most frequent pair of adjacent characters or character sequences to form new tokens. This process will continue till we reach to certain vocab size.
- Frequency-Based: The decision for merging two characters are based on their frequency.The most frequent pairs are merged at first.

BERT use WordPiece tokenizer alogrithem, the procedure is:

- Minimizing Loss Function: WordPiece also starts with a base vocabulary of characters and iteratively adds the most beneficial token to the vocabulary based on a criterion, but its selection process is slightly different. It aims to minimize a loss function, which often involves choosing tokens that decrease the likelihood of the training data given the current vocabulary.
- Likelihood-Based: The decision for which token we will pick, is based on considering both frequency of token and how adding the token will affect the overall likelihood of the text.This makes WordPiece tokenizer a little bit complex than BPE.

Differences are :

- Chosing Tokens: In BPE we merging most frequent characters but in WordPiece even we use the frequency of the token we will use a evaluation metric to see adding that particular token will affect the likelihood.
- Final vocabulary: Both will generate different vocabulary.

### 2.3 part 3

We have used two method to normalized data: One with 're' library to normalize data and another method is using built-in normalizer in 'tokenizer' library. First Method to normalizing data:(my way)

```
    def normalizing_data(content):
        content = content.lower()
        content = re.sub(r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\\(\\)
           ↪ ,]|(?:%[0-9a-fA-F][0-9a-fA-F]))+', '', content)
        content = re.sub(r'www(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\\(\\),]|(?:%[0-9a-fA
           ↪ -F][0-9a-fA-F]))+', '', content)
        content = re.sub(r'[^\w\s]', '', content)
        content = re.sub(r'<[^>]+>', '', content)
        content = re.sub(r'\s+', ' ', content).strip()
        content = unicodedata.normalize('NFKD', content).encode('ascii', 'ignore')
           ↪ .decode('utf-8')
        return content
```

This piece of code are so useful since:

- It will convert the text to lower case.
- Deleting all sites(since they are nonesence words)
- Delete all HTML Tags!
- Removing all punctuations.
- Deleting whitespaces.
- Removing accents form text and convert them to a plain ASCII representation.

Now, the 'tokenizer' library allow us to add Tokenizer and models like BPE and WordPiece.To train a BPE and WordPiece models on our dataset, we should use BpeTrainer and also WordPieceTrainer.We can use pre tonizer to ensure the whitespaces and punctuations are eliminated from our text file!

```
    from tokenizers import Tokenizer
    from tokenizers.models import BPE, WordPiece
    from tokenizers.trainers import BpeTrainer, WordPieceTrainer
    from tokenizers.pre_tokenizers import Whitespace
```

The code has been implemented using these library is as follow:

```
    tokenizer = Tokenizer(BPE(unk_token="[UNK]"))
    trainer = BpeTrainer(special_tokens=["[UNK]", "[CLS]", "[SEP]", "[PAD]", "[
       ↪ MASK]"])
    tokenizer.pre_tokenizer = Whitespace()
    file = ['G:/Master SE/Term 2/NLP/CA\'s/CA1/My codes/
       ↪ normalized_with_my_method_All_Around_the_Moon.txt']
    tokenizer.train(file, trainer)
    tokenizer.save("bpe_tokenizer.json")
    vocab_size = tokenizer.get_vocab_size()
    print(f'vocab size for BPE is: {vocab_size}')
```

In the code above, the result has been saved in .json format.Furthermore, if a test sentence is fed to the model containing tokens where the model has not seen during training phase, it would be filled by 'UNK'. Some of special tokens are(yet not important in our task) is:

UNK (Unknown): Represents words or subwords that are not found in the tokenizer's vocabulary.

CLS (Classifier): A special token added at the beginning of each input sequence.(It is used for classification purposes)

SEP (Separator): Used to separate different sequences within the same input for example in question-answering models where a question and a passage are provided as inputs, or in tasks requiring the comparison of two sequences. It helps the model distinguish between different parts of the input.

PAD (Padding): Used to fill shorter sequences to match the length of the longest sequence in a batch of input data.

MASK : Specifically used in the training of models like BERT, where some percentage of the input tokens are replaced with the [MASK] token.

we can do the same procedure for WordPiece model as well:

```
    tokenizer = Tokenizer(WordPiece(unk_token="[UNK]"))
    trainer = WordPieceTrainer(special_tokens=["[UNK]", "[CLS]", "[SEP]", "[PAD]",
       ↪  "[MASK]"])
    tokenizer.pre_tokenizer = Whitespace()
    file = ['G:/Master SE/Term 2/NLP/CA\'s/CA1/My codes/
       ↪ normalized_with_my_method_All_Around_the_Moon.txt']
    tokenizer.train(file, trainer)
    tokenizer.save("wordpiece_tokenizer.json")
    tokenizer.get_vocab_size()
```

The vocab sizes for each method are represented in table 3:

Table 3: Vocab size for each tokenizer

| BPE | WordPiece |
|---|---|
| 14851 | 15998 |

Differentiate of vocab sizes are basd on differences between these two alogrithems(discussed in previous part). Furthermore, BPE tokenizer is frequency-based.However WordPiece approach is quite different since it use frequency and also WordPiece chooses next word based on that maximizies the likelihood of the training data given the current vocabulary.
BPE's smaller vocabulary might indicate that it has focused on merging the most common pairs to form tokens. WordPiece's larger vocabulary suggests that it has identified additional tokens that, while perhaps less frequent, contribute to a better overall understanding of the text.
Given sentence to tokenize with BPE and WordPiece models:

- ”This darkness is absolutely killing! If we ever take this trip again, it must be about the time of the sNew Moon!”
- This is a tokenization task. Tokenization is the first step in a NLP pipeline. We will be comparing the tokens generated by each tokenization model.

Tokenizing with BPE:

- 'this', 'darkness', 'is', 'absolutely', 'killing', 'if', 'we', 'ever', 'take', 'this', 'trip', 'again', 'it', 'must', 'be', 'about', 'the', 'time', 'of', 'the', 's', 'new', 'moon'

- ’this’, ’is’, ’a’, ’to’, ’ken’, ’ization’, ’task’, ’to’, ’ken’, ’ization’, ’is’, ’the’, ’first’, ’step’, ’in’, ’a’, ’n’, ’lp’, ’pi’, ’pe’, ’line’, ’we’, ’will’, ’be’, ’comparing’, ’the’, ’to’, ’k’, ’ens’, ’generated’, ’by’, ’each’, ’to’, ’ken’, ’ization’, ’model’

Tokenizing with WordPiece:

- ’this’, ’darkness’, ’is’, ’absolutely’, ’killing’, ’if’, ’we’, ’ever’, ’take’, ’this’, ’trip’, ’again’, ’it’, ’must’, ’be’, ’about’, ’the’, ’time’, ’of’, ’the’, ’sne’, ’##w’, ’moon’
- ’this’, ’is’, ’a’, ’to’, ’##ken’, ’##ization’, ’task’, ’to’, ’##ken’, ’##ization’, ’is’, ’the’, ’first’, ’step’, ’in’, ’a’, ’n’, ’##l’, ’##p’, ’pip’, ’##eli’, ’##ne’, ’we’, ’will’, ’be’, ’comparing’, ’the’, ’to’, ’##ken’, ’##s’, ’generated’, ’by’, ’each’, ’to’, ’##ken’, ’##ization’, ’model’

The approach two is when we use built-in normalizer in ’tokenizer’ library.Importing normalizer for BPE:

```
    from tokenizers.normalizers import Lowercase, NFD, StripAccents, Sequence
```

Adding normalizer to the game!

```
    normalizer = Sequence([
        Lowercase(),
        StripAccents(),
        NFD(),
    ])
    tokenizer.normalizer = normalizer
```

The interpretation of added things in the ’Sequence’:

- Lowercase: Convert all text to lowercase.
- StripAccents: Removes diacritics or accent marks from characters.
- NFD (Normalization Form Decomposition): Decomposes characters into their base characters and separate diacritical marks.

For WordPiece, we can simply add BertNormalizer:

```
    normalizer = Sequence([
        BertNormalizer()
    ])
    tokenizer.normalizer = normalizer
```

The vocab size for the second method has been shown in table 4:

Table 4: Vocab size for each tokenizer using built-in normalizer

| BPE | WordPiece |
|---|---|
| 14316 | 15259 |

Encoding given sentences using BPE tokenizer:

- ’this’, ’darkness’, ’is’, ’absolutely’, ’killing’, ’!’, ’if’, ’we’, ’ever’, ’take’, ’this’, ’trip’, ’again’, ’,’, ’it’, ’must’, ’be’, ’about’, ’the’, ’time’, ’of’, ’the’, ’s’, ’new’, ’moon’, ’!’
- ’this’, ’is’, ’a’, ’to’, ’ken’, ’ization’, ’task’, ’.’, ’to’, ’ken’, ’ization’, ’is’, ’the’, ’first’, ’step’, ’in’, ’a’, ’n’, ’lp’, ’pi’, ’pe’, ’line’, ’.’, ’we’, ’will’, ’be’, ’comparing’, ’the’, ’to’, ’k’, ’ens’, ’generated’, ’by’, ’each’, ’to’, ’ken’, ’ization’, ’model’, ’.’

Encoding given sentences with WordPiece tokenizer:

- ’this’, ’darkness’, ’is’, ’absolutely’, ’killing’, ’!’, ’if’, ’we’, ’ever’, ’take’, ’this’, ’trip’, ’again’, ’,’, ’it’, ’must’, ’be’, ’about’, ’the’, ’time’, ’of’, ’the’, ’sne’, ’##w’, ’moon’, ’!’
- ’this’, ’is’, ’a’, ’to’, ’##ken’, ’##ization’, ’task’, ’.’, ’to’, ’##ken’, ’##ization’, ’is’, ’the’, ’first’, ’step’, ’in’, ’a’, ’n’, ’##l’, ’##p’, ’pip’, ’##el’, ’##ine’, ’.’, ’we’, ’will’, ’be’, ’comparing’, ’the’, ’to’, ’##ken’, ’##s’, ’generated’, ’by’, ’each’, ’to’, ’##ken’, ’##ization’, ’model’, ’.’

From tokenizing the sentences, apply either first or second method will not make in significant difference result.

## 3 Question 3

### 3.1 part 1

Firsly we have normalized the Tarzan.text where the normalizer was implemented in the question 2.

```
    tarzan_content = normalizing_data(tarzan_content)
    tarzan_content = tarzan_content.translate(str.maketrans('', '', string.
       ↪ punctuation))
    tarzan_content
```

The second line is for deleting punctuations. We have implemented a class with some features for modeling our ngram:

```
    class ngram:
        def __init__(self, n, data_set):
            self.data = data_set
            self.n = n
```

This class will be initialized with n(for n grams) and a data set.In order to tokenize this we will use 'nltk' library.

```
        def tokenize_data(self):
            self.tokenized = word_tokenize(self.data)
```

The function word_tokenize will tokenize our dataset.

### 3.2 part 2

For getting ngram and frequency of each subsequence we define a new function called 'frequency':

```
        def frequency(self):
            subsequences = []
            for i in range(len(self.tokenized)- self.n + 1):
                subsequence = tuple(self.tokenized[i:i+self.n])
                subsequences.append(subsequence)
            return Counter(subsequences)
```

The Counter in 'collections' library will tell us the frequency of each pair.For calculating probability we must use Markov assumption as follows:

$$p(w_1 w_2 \cdots w_n) \approx p(w_i | w_{i-k} \cdots w_{i-1}) \tag{2}$$

In other words:

$$p(w_i | w_1 w_2 \cdots w_{i-1}) \approx p(w_i | w_{i-k} \cdots w_{i-1}) \tag{3}$$

This is formula is baesd on kgrams where from a given corpus to train the kgram model the probability will be calculated from this formula:

$$p(w_i | w_{i-k} \cdots w_{i-1}) = \frac{Count(w_{i-k} \cdots w_i)}{Count(w_{i-k} \cdots w_{i-1})} \tag{4}$$

But our freqeuncy function just tell us the seen ngrams.More precisely, assume for bigram modeling we have 'the project' in our dataset but we have not 'the the' therefore the frequency of this pair is zero. lets see how many unique tokens we have in our dataset with following feature in our class:

```
    def vocab_size(self):
        return len(set(self.tokenized))
```

The total unique tokens in our dataset is about 6842 and if we want to add unseen data like 'the the' we must add any combination of two unique words in our dataset.It means about $6842^2$ pairs where the memory will allow us to add unseen combination pairs for bigrams but for example in fivegram model we must have $6842^5$ combinations of any sequence in our vocabulary where the memory will not allow us to do add unseen data.This issue called data sparsity since the probability of unseen subsequences are zero, our ngram model during training phase did not see them.
Let's investigate this issue in bigrams: As illustrated in table 5, approximately 46M data has not been seen and

Table 5: Seen versus unseen data

| Frequency for seen data | length of unseen data |
|---|---|
| 39544 | 46773420 |

therefore 46M probabilities are zero. There are several approaches in order to solve this issue, some of them are:

- Add_1 smoothing
- Add_$\alpha$ smoothing

Add_1 approach:

$$p(w_i|w_{i-k}\cdots w_{i-1}) = \frac{Count(w_{i-k}\cdots w_i)+1}{Count(w_{i-k}\cdots w_{i-1})+|V|} \tag{5}$$

Add_$\alpha$ approach:

$$p(w_i|w_{i-k}\cdots w_{i-1}) = \frac{Count(w_{i-k}\cdots w_i)+\alpha}{Count(w_{i-k}\cdots w_{i-1})+\alpha|V|} \tag{6}$$

We know that when the zeros are so many, using add1 will not be a good strategy.Thus we use add_$\alpha$ smoothing. In order to calculate the probability with alpha smoothing we have add feature to our class:

```
        def add_alpba_probability(self, freq, n, alpha = 0.001):
            vocab_size = len(set(self.tokenized))
            probs = {}
            self.n = n-1
            prev_gram_freq = self.frequency()
            self.n = n
            for key,value in freq.items():
                probs[key] = ((value + alpha)/(prev_gram_freq[key[:-1]] + (alpha*
                   ↪ vocab_size)))
            return probs
```

## 3.3 part 3

In order to complete a sentence with our ngram model we have implemented following function:

```
    def complete_sentence_backoff(self, inc_sentence, method, min_lenght=10):
        inc_sentence = normalizing_data(inc_sentence)
        inc_sentence = inc_sentence.translate(str.maketrans('', '', string.
           ↪ punctuation))
        tokens = word_tokenize(inc_sentence)
        current_token = []
        sentence = tokens[:]
        i = self.n
        real_n = self.n
        while len(sentence) < min_lenght + len(tokens):
            candidates = []
            while not candidates:
                current_token = sentence[-i+1:]
                self.n = i
                freq = self.frequency()
                probs = self.add_alpba_probability(freq, i)
                candidates = [(pair, prob) for pair,prob in probs.items() if list(
                   ↪ pair[:i-1]) == current_token]
                i = i - 1
            self.n = real_n
            i = self.n
            if method == 'max':
                chosen_token = max(candidates, key= lambda x: x[1])[0][-1]
            else:
                weights = [0 for _ in range(len(candidates))]
                for k in range(len(candidates)):
                    weights[k] = candidates[k][1]
                chosen_token = random.choices(candidates, weights=weights, k=1)
                   ↪ [0][0][-1]
            sentence.append(chosen_token)
        return ' '.join(sentence)
```

We used backoff algorithem in order to complete sentence.It is a strategy to complete a sentence where we will explain it in the next part of this question. Firsly, we normalize our incomplete sentence and tokenize it.Let's assume we want a bigram model predict our next word.Based on the last word, candidates will be listed and due to their probability we pick one of them. If method is 'max', candidate with most probability will be picked otherwise it would be random based on their probability.
Incomplete sentences:

- Knowing well the windings of the trail he · · ·
- For half a day he lolled on the huge back and · · ·

Predicted sentences with maximum candidate:

- knowing well the windings of the trail he had been a great tourney and the great tourney and

- for half a day he lolled on the huge back and the great tourney and the great tourney and the great

Predicted sentences with random candidates:

- knowing well the windings of the trail he had found no other to trust to the castle what
- for half a day he lolled on the huge back and hurried through the black was moving silently into the difficulty

As you can see the completed sentence has a little meaning.

## 3.4 part 4

Backoff strategy is one of useful when we have data sparsity.Assume we have fivegram model where we want to use it in order to predict the last word based on four last tokens.Assume that four last tokens were not in our seen data then how can we generate the next token of this sentence?
This is where backoff algorithem will help us.It will use previous ngram models whenever it does not see ’n’ last tokens it would consider (n-1)gram to complete it and if (n-1)gram can not see the (n-1) last tokens it will use (n-2)gram and so on.
Complete sentence for trigram using max method:

- knowing well the windings of the trail he did not know that he had been the last event
- for half a day he lolled on the huge back and forth wagers were laid and woe betide the contender who

Complete sentence for trigram using random choice:

- knowing well the windings of the trail he came for ivory alone am i added blake when the
- for half a day he lolled on the huge back and forth slowly there were lapses in their twenties for to

Complete sentence for fivegram using max method:

- knowing well the windings of the trail he took short cuts swinging through the branches of the trees
- for half a day he lolled on the huge back and forth wagers were laid and woe betide the contender who

Complete sentence for fivegram using random choice:

- knowing well the windings of the trail he took short cuts swinging through the branches of the trees
- for half a day he lolled on the huge back and essayed to say eh and to yawn at the same

As you can see with fivegram the sentence has a better menaing than trigram and trigram has a better menaing than bigram.

## 3.5 part 5

The answer is no.There are several reasons that limit us not to increase parameter ’n’:

- Data Sparsity:If parameter ’n’ increases, it would lead the model to have much more number of zero probabilities. Thus it would have so much sparsity in our ngram models and also it means that our model will not see so many longer sequences to make accurate predicitions.
- Computational Complexity: The higer ’n’ gets, the model’s complexity will increase exponentialy and also increase our memory to save the sequences.
- Overfitting: Larger ’n’ value causes the model memorize specific ngram sequences rather than learning the pattern of the language.
- Context Relevance: In so many languages, the next word based on few last words is more predictive than a long distance context.

## 4 Question 4

### 4.1 part 1 and 2

We have created two different functions, where we plan to discuss each.
Prediction polarity function:

```python
    def predict_polarity(review, positive_freq, negative_freq, n):
        grams = get_ngrams(review, n)
        positive_prob = sum(np.log(positive_freq.freq(gram) + 1e-6) for gram in
           ↪ grams)
        negative_prob = sum(np.log(negative_freq.freq(gram) + 1e-6) for gram in
           ↪ grams)
        label = 1 if positive_prob >= negative_prob else 0
        return label
```

This function is based on Naïve Bayes classifier.The procedure is as follow:

- Tokenizing and ngrams: Firstly, the given reivew will be tokenized and based on 'n' we will get the ngrams from the written fucntion 'get_ ngrams'.
- Calculating probabilities: We will use 'positive_freq' and 'negative_freq' to calculate probability of how much likely that sentence is positive or negative.

$$\begin{aligned} \text{positive-prob} &= \prod_{i=1}^{N} p(w_i|w_{i-1}) \\ &= \prod_{i=1}^{N} \frac{count(w_i, w_{i-1})}{count(w_{i-1})} \end{aligned} \tag{7}$$

  Attention: This probability of a sentence is based on bigram. The dominator of this fraction is not important since all words are unique in that particular sentence.Therefore for comparison between probability that sentence being in negative class or positive class will not effected by this dominator.Thus we can delete it from our discriminator. (This can be applied to even ngram with higer n!)
  Log likelihood:

$$\begin{aligned} \text{positive-prob} &= \sum_{i=1}^{n} log(p(w_i|w_{i-1})) \\ &\propto \sum_{i=1}^{n} log\left(freq(w_i, w_{i-1})\right) \end{aligned} \tag{8}$$

  But since this probability can be zero and log would be $-\infty$ therefor to prevent this issue we added $1e-6$ to positive and negative probabilities.

- Decision making: We will compare the probability of positive and negative probability.If positive probability is bigger than negative probability, class 1 will be assigned to the positive probability if it's bigger than negative probability otherwise in class 0.

Accuracy function:

```python
    def test_ngram(test_data, positive_freq, negative_freq, n):
        correct_predictions = 0
        predicted_labels = []
        for _,row in test_data.iterrows():
            predicted = predict_polarity(row['review'], positive_freq,
               ↪ negative_freq, n)
            predicted_labels.append(predicted)
            if predicted == row['polarity']:
                correct_predictions += 1
        return correct_predictions/len(test_data), predicted_labels
```

In this piece of code we will check if predicted label is equal to the real ones or not.Furthermore, we counting the number of predicted labels are equal to the polarity of reviews.We also return predicted labels. The metrics from this function alongside builtin functions in ’sklearn’ library has been illustrated in table 6:

Table 6: Evaluation Matrix

| Percison | Recall | F1 score |
|---|---|---|
| 0.8125 | 0.49 | 0.61 |

The confusion matrix are shown in figure 7:

Table 7: Confusion Matrix

| | |
|---|---|
| 120 | 6 |
| 27 | 26 |

Interpreting the evaluation metrics:

- Precision: Precision of 0.8125 indicates that when the model predicts a sample as positive, it is correct about 81.25% of the time.
- Recall: The recall of approximately 0.4906 means that the model correctly identifies 49.06% of all actual positive cases. In other words, it misses out on approximately 50.94% of the positive cases, labeling them as negative.
- F1 Score: The F1 Score is the harmonic mean of precision and recall, and at 0.6117, it suggests a balance between precision and recall.
- Confusion Matrix:There are 120 instances where the model correctly predicted the negative class, there are 6 instances where the model incorrectly predicted the negative instances as positive , there are 27 instances where the model incorrectly labeled positive instances as negative, there are 26 instances where the model correctly identified the positive class.

## 5 Codes (Read This First)

My code is in ’My codes’ file including all questions in a particular jupyter notebook. Careful with the paths, if you are runing the code.More over, whole datas are gathered in the ’data’ file and all new files are in the ’My codes’ path.