به نام خدا

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

درس پردازش زبان طبیعی

پاسخ تمرین{1}

نام و نام خانوادگی: مریم حسینعلی

شماره دانشجویی: 610398209

QUESTION 1

PART 1

regex pattern r'\b\w+\b': \b is a word boundary, \w+ matches 1 or more word characters so regex pattern r'\b\w+\b' matches whole words in a text. (whole words = letters, digits, or underscores)

custom_tokenizer function is a **word-based** tokenizer. custome_tokenizer by using the regular expression pattern **`\b\w+\b`** finds all sequences of word characters in the input text.

disadvantages and problems of word-based tokenizer:

1. Words can have multiple meanings or be part of different phrases . Tokenizing only based on spaces can lead to ambiguity in token boundaries.

2. Words like "can't" or "won't" contain apostrophes, which can be challenging for tokenization. Tokenizing them as separate words like "can" and "'t" may lose the desired meaning.

3. Words like "state-of-the-art" can be tokenized incorrectly, breaking them into separate tokens and losing their desired meaning.

4. Tokenizing based on spaces might not handle punctuation marks correctly. For example, "I said, 'hello!'" might be tokenized into "I", "said," and "'hello!'" including punctuation in some tokens.

5. Some languages, like Chinese or Thai, do not use spaces between words. Tokenizing these languages based on spaces would be incorrect.

6. Words containing special characters, like email addresses or URLs, can be tokenized inconsistently if not handled properly.

7. Tokenization based on spaces might not handle rare words well, as they might be split into smaller tokens.

PART2

the result of tokenization is:

```python
import re

def custom_tokenizer(text):
    pattern = r'\b\w+\b'
    tokens = re.findall(pattern, text)
    return tokens

text = "Just received my M.Sc. diploma today, on 2024/02/10! Excited to embark on this new journey of knowledge and discovery. #MScGraduate
tokens = custom_tokenizer(text)
print(tokens)
```

```
['Just', 'received', 'my', 'M', 'Sc', 'diploma', 'today', 'on', '2024', '02', '10', 'Excited', 'to', 'embark', 'on', 'this', 'new', 'journey
', 'of', 'knowledge', 'and', 'discovery', 'MScGraduate', 'EducationMatters']
```

['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 tokenization output has several problems:

1. Words like "M.Sc." are not correctly tokenized, they are treated as separate tokens 'M' and 'Sc'.

2. Dates '2024/02/10' are not handled as separate tokens, which may be desired in certain cases.

3. Hashtags are not correctly tokenized. They should be treated as one token each.

4. Words like "new journey" are not tokenized as a single token, that should be treated as one token.

PART3

I want to solve first problem:

1. Words like "M.Sc." are not correctly tokenized, they are treated as separate tokens 'M' and 'Sc'.

```python
import re

def custom_tokenizer(text):
    pattern = r'\b(?:[A-Z][a-z]*\.?)+\b|\b\w+\b'
    tokens = re.findall(pattern, text)
    return tokens

text = "Just received my M.Sc. diploma today, on 2024/02/10! Excited to embark on this new journey of knowledge and discovery. #MScGraduate
tokens = custom_tokenizer(text)
print(tokens)
```

```
['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 tokenized expression is:

['Just', 'received', 'my', 'M.Sc', 'diploma', 'today', 'on', '2024', '02', '10', 'Excited', 'to', 'embark', 'on', 'this', 'new', 'journey', 'of', 'knowledge', 'and', 'discovery', 'MScGraduate', 'EducationMatters']

we can see ‘M.Sc’ has been tokenized correctly.

1. **\b(?:[A-Z][a-z]*\.?)+\b**:

- **\b**: Matches a word boundary.

- **(?:[A-Z][a-z]*\.?)+**: This matches a sequence of uppercase letters **[A-Z]** followed by zero or more lowercase letters **[a-z]***, and an optional dot **\.?**. This makes it possible to match abbreviations like "M." or "Sc." as well as the full abbreviation "M.Sc.".

- **\b**: word boundary to make sure the end of the abbreviation.

2. **|\b\w+\b:** This handles regular words by matching sequences of word characters (**\w+**) that are covered by word boundaries (**\b**). This makes sure the regular words are correctly tokenized.

پاسخ سوال دوم

PART1

BERT tokenizer:
BERT tokenizer uses something known as subword-based tokenization. Subword-tokenization splits unknown words into smaller words or characters such that the model can derive some meaning from the tokens.

GPT tokenizer:
GPT uses a byte pair encoding (BPE) tokenizer. Byte pair encoding is a type of subword tokenization that iteratively merges the most frequent pairs of consecutive bytes or characters in a corpus to create a vocabulary of subword units. This tokenizer allows GPT to handle a wide

range of words and subword units, making it effective for generating text and understanding context in a more fine-grained manner than traditional word-based tokenization.

so both BERT and GPT use subword-based tokenizers.

Why do LLMs like GPT and BERT use subword-based tokenizers?
Large language models like BERT and GPT use subword-based tokenization for several reasons:

1. Subword tokenization allows models to represent and process words that are not present in the vocabulary by breaking them down into smaller subword units. This helps improve the model's ability to handle rare and unseen words.

2. Subword tokenization provides a more flexible and adaptive way to represent words, capturing both morphological and semantic information. This can lead to more nuanced and contextually rich representations of words.

3. By breaking words into subword units, the vocabulary size can be kept relatively small while still being able to represent a wide range of words. This helps in reducing the computational and memory complexity of the model.

4. Subword tokenization can be applied across different languages, making the models more capable of handling multilingual tasks.

5. Subword tokenization can improve the model's ability to generalize to new and unseen data by capturing similarities between words at a subword level.

subword tokenization allows models to deal with a broad range of words, even those not seen during training.so subword-based tokenization enables LLM to achieve better performance on a wide range of natural language processing tasks.

PART2

BERT uses the WordPiece algorithm, and GPT uses the BPE algorithm, which are effective for rare or unknown words.

first let know about both algorithms

BPE:

- Initial Inventory: BPE starts with an inventory of all individual characters in the text.

- Build Language Model: A language model is created using this initial inventory.

- New Word Unit Creation: The algorithm iteratively creates new word units by combining existing units in the inventory. The selection criterion for the new word unit is based on which combination most increases the likelihood of the training data when added to the model.

- Iterative Process: This process continues until reaching a predefined number of word units or the likelihood increase falls below a certain threshold.

WordPiece:

- Initial Inventory: WordPiece starts with an inventory of all individual characters in the text.

- Build Language Model: A language model is created using this initial inventory.

- New Word Unit Creation: The algorithm iteratively creates new word units by combining existing units in the inventory. The selection criterion for the new word unit is based on which combination most increases the likelihood of the training data when added to the model.

- Iterative Process: This process continues until reaching a predefined number of word units or the likelihood increase falls below a certain threshold.

now lets see their differences:

Merge Criterion: While both BPE and WordPiece merge units based on certain criteria, BPE typically uses the frequency of unit pairs, merging the most frequent pairs first. In contrast, WordPiece considers the overall likelihood of the entire vocabulary, merging the pair that increases the likelihood the most.

Likelihood Calculation: BPE and WordPiece differ in how they calculate the likelihood increase. BPE looks at the frequency of the pair of units being merged, while WordPiece looks at the impact of merging the pair on the overall likelihood of the data.

Architecture: BERT uses a transformer architecture but only uses the encoder part, focusing on bidirectional context. but GPT uses the entire transformer architecture, including both the encoder and decoder, for autoregressive generation of text.

Training Objective:BERT is pre-trained using masked language modeling and next sentence prediction tasks, which helps it understand bidirectional context and relationships between sentences. GPT, on the other hand, is trained using a single task: predicting the next word in a sequence, which encourages it to generate coherent and contextually appropriate text.

Context Handling: BERT handles bidirectional context by masking some of the input tokens and predicting them based on the context provided by the surrounding tokens. GPT, being autoregressive, generates text one token at a time, conditioning each token on the previously generated tokens.

PART3

GPT tokenization:

```python
from transformers import GPT2Tokenizer


tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

with open('All_Around_the_Moon.txt', 'r') as file:
    text = file.read()

# tokenization
tokens = tokenizer.tokenize(text)
```

BERT tokenization:

```python
from transformers import BertTokenizer


tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')


with open('All_Around_the_Moon.txt', 'r') as file:
    text = file.read()

# tokenization
tokens = tokenizer.tokenize(text)
```

vocabulary size:

```python
from transformers import BertTokenizer, GPT2Tokenizer


bert_tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
print("BERT Vocabulary Size:", len(bert_tokenizer))


gpt2_tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
print("GPT-2 Vocabulary Size:", len(gpt2_tokenizer))
```

```
BERT Vocabulary Size: 30522
GPT-2 Vocabulary Size: 50257
```

The vocabulary sizes of BERT is 30522 and vocabulary sizes of GPT-2 is 50257 so they are different.

The vocabulary size of BERT is determined by the number of subword units in its vocabulary.

In the previous part of this question we saw GPT-2 uses BPE tokenizer, BPE merges the most frequent pairs of characters or subwords, which can result in a larger vocabulary size compared to WordPiece(which BERT uses).

So the difference in vocabulary size between BERT and GPT-2 is mainly because of their use of different tokenization algorithms and vocabulary construction methods.

sentences tokenization:

GPT:

```
sentence1 = "This darkness is absolutely killing! If we ever take this trip again, it must be about the time of the sNew Moon!"
tokenized_sentence1 = tokenizer.tokenize(sentence1)
print('sentence1',tokenized_sentence1,'\n')

sentence2 = "This is a tokenization task. Tokenization is the first step in a NLP pipeline. We will be comparing the tokens generated by eac
tokenized_sentence2 = tokenizer.tokenize(sentence2)
print('sentence2',tokenized_sentence2)
```

```
sentence1 ['This', 'Ġdarkness', 'Ġis', 'Ġabsolutely', 'Ġkilling', '!', 'ĠIf', 'Ġwe', 'Ġever', 'Ġtake', 'Ġthis', 'Ġtrip', 'Ġagain', ',', 'Ġi
t', 'Ġmust', 'Ġbe', 'Ġabout', 'Ġthe', 'Ġtime', 'Ġof', 'Ġthe', 'Ġs', 'New', 'ĠMoon', '!']

sentence2 ['This', 'Ġis', 'Ġa', 'Ġtoken', 'ization', 'Ġtask', '.', 'ĠToken', 'ization', 'Ġis', 'Ġthe', 'Ġfirst', 'Ġstep', 'Ġin', 'Ġa', 'ĠN',
'LP', 'Ġpipeline', '.', 'ĠWe', 'Ġwill', 'Ġbe', 'Ġcomparing', 'Ġthe', 'Ġtokens', 'Ġgenerated', 'Ġby', 'Ġeach', 'Ġtoken', 'ization', 'Ġmodel']
```

BERT:

```
sentence1 = "This darkness is absolutely killing! If we ever take this trip again, it must be about the time of the sNew Moon!"
tokenized_sentence1 = tokenizer.tokenize(sentence1)
print('sentence1',tokenized_sentence1,'\n')

sentence2 = "This is a tokenization task. Tokenization is the first step in a NLP pipeline. We will be comparing the tokens generated by eac
tokenized_sentence2 = tokenizer.tokenize(sentence2)
print('sentence2',tokenized_sentence2)
```

```
sentence1 ['this', 'darkness', 'is', 'absolutely', 'killing', '!', 'if', 'we', 'ever', 'take', 'this', 'trip', 'again', ',', 'it', 'must', '
be', 'about', 'the', 'time', 'of', 'the', 's', '##ne', '##w', 'moon', '!']

sentence2 ['this', 'is', 'a', 'token', '##ization', 'task', '.', 'token', '##ization', 'is', 'the', 'first', 'step', 'in', 'a', 'nl', '##p',
'pipeline', '.', 'we', 'will', 'be', 'comparing', 'the', 'token', '##s', 'generated', 'by', 'each', 'token', '##ization', 'model']
```

GPT:
sentence1 ['This', 'Ġdarkness', 'Ġis', 'Ġabsolutely', 'Ġkilling', '!', 'ĠIf', 'Ġwe', 'Ġever', 'Ġtake', 'Ġthis', 'Ġtrip', 'Ġagain', ',', 'Ġit', 'Ġmust', 'Ġbe', 'Ġabout', 'Ġthe', 'Ġtime', 'Ġof', 'Ġthe', 'Ġs', 'New', 'ĠMoon', '!']

sentence2 ['This', 'Ġis', 'Ġa', 'Ġtoken', 'ization', 'Ġtask', '.', 'ĠToken', 'ization', 'Ġis', 'Ġthe', 'Ġfirst', 'Ġstep', 'Ġin', 'Ġa', 'ĠN', 'LP', 'Ġpipeline', '.', 'ĠWe', 'Ġwill', 'Ġbe', 'Ġcomparing', 'Ġthe', 'Ġtokens', 'Ġgenerated', 'Ġby', 'Ġeach', 'Ġtoken', 'ization', 'Ġmodel']

Each token is separated by a space, and special tokens like 'Ġ' are used to indicate the beginning of a new word.

BERT:
sentence1 ['this', 'darkness', 'is', 'absolutely', 'killing', '!', 'if', 'we', 'ever', 'take', 'this', 'trip', 'again', ',', 'it', 'must', 'be', 'about', 'the', 'time', 'of', 'the', 's', '##ne', '##w', 'moon', '!']

sentence2 ['this', 'is', 'a', 'token', '##ization', 'task', '.', 'token', '##ization', 'is', 'the', 'first', 'step', 'in', 'a', 'nl', '##p', 'pipeline', '.', 'we', 'will', 'be', 'comparing', 'the', 'token', '##s', 'generated', 'by', 'each', 'token', '##ization', 'model']

The reasons behind these differences:

The GPT-2 tokenizer has a larger vocabulary size of 50,257, enabling it to include a more diverse set of words and subwords. This larger vocabulary size can result in different tokenizations compared to BERT.

The BERT tokenizer has a smaller vocabulary size of 30,522, which may lead to different tokenizations for out-of-vocabulary words or their split into subword units.

These differences in algorithms and vocabulary sizes between the GPT-2 and BERT tokenizers can influence how they process and represent text, leading to different tokenizations for the same input sentence.

QUESTION 3

PART1

This part is for preprocessing text data from the "Tarzan.txt" file and training an n-gram language model using the NLTK library.

First I uploaded the “Tarzan.txt” file in my jupyter environment.

preprocessing steps:

1) removing punctuation
2) tokenization
3) removing stopwords

in the first step I removed punctuations from the text, using the following function:

```
# remove punctuations function
def remove_punc(text):
    return ''.join([char for char in text if char not in string.punctuation])

# removeing punctuation from the text
clean_text = remove_punc(text)
```

After cleaning the text (by removing punctuations), I used the word_tokenize function from NLTK to tokenize the text and split it into individual words:

```
# Tokenization
tokens = word_tokenize(clean_text)
```

After tokenization, I removed stopwords from the text. NLTK has a list of stopwords for English and i used that list for removing stopwords:

```
# removing stopwords
stop_words = set(stopwords.words('english'))
tokens = [token for token in tokens if token.lower() not in stop_words]
```

The n-gram model was trained using bigrams (n=1),

The choice of n=1 is arbitrary and could be adjusted to any desired n.

```
# training n-gram model
n = 1
ngrams_list = list(ngrams(tokens, n))
# generating n-grams
```

We can make sure the text is tokenized correctly by counting and showing the most common unigrams(or any desired n-gram).

```
ngrams_freq = Counter(ngrams_list)

# print the most common n-grams (unigrams here)
print(ngrams_freq.most_common(10))
```

```
[(('Blake',), 411), (('upon',), 373), (('Tarzan',), 329), (('said',), 253), (('Ibn',), 238), (('would',), 228), (('Jad',), 227), (('Sir',), 205), (('one',), 203), (('man',), 199)]
```

PART2

when n is set to 2, we generate all possible bigrams of the text.

```
# training n-gram model
n = 2
ngrams_list = list(ngrams(tokens, n))
# generating n-grams
```

Now we calculates the probability of each bigram, and we print the most common bigrams, their frequencies, and probabilities:

```
# calculating the total number of unique bigrams
V = len(set(bigrams_list))

# calculating the probability of each bigram
bigram_prob = {}
for bigram in bigrams_freq:
    bigram_prob[bigram] = bigrams_freq[bigram] / len(bigrams_list)

most_common_bigrams = bigrams_freq.most_common(10)
for bigram, freq in most_common_bigrams:
    prob = bigram_prob[bigram]
    print(f"Bigram: {bigram}, Frequency: {freq}, Probability: {prob}")
```

```
Bigram: ('Ibn', 'Jad'), Frequency: 227, Probability: 0.006143270818110471
Bigram: ('Tarzan', 'Apes'), Frequency: 57, Probability: 0.0015425834212876512
Bigram: ('Project', 'Gutenberg™'), Frequency: 56, Probability: 0.001515520554247517
Bigram: ('Sir', 'Richard'), Frequency: 52, Probability: 0.00140726908608698
Bigram: ('Sir', 'James'), Frequency: 51, Probability: 0.0013802062190046846
Bigram: ('Sir', 'Malud'), Frequency: 39, Probability: 0.0010554518145652351
Bigram: ('Project', 'Gutenberg'), Frequency: 28, Probability: 0.0007577602771237585
Bigram: ('sir', 'knight'), Frequency: 27, Probability: 0.0007306974100836243
Bigram: ('Princess', 'Guinalda'), Frequency: 27, Probability: 0.0007306974100836243
Bigram: ('Knights', 'Nimmr'), Frequency: 25, Probability: 0.0006765716760033558
```

Data Sparsity: As 'n' increases, the frequency of specific n-gram sequences decreases, leading to data sparsity issues. This means that there may not be enough examples of certain word sequences in the training data for the model to learn effectively. Therefore, the model may

struggle to accurately estimate the probabilities of these rare n-grams, making its predictions less reliable.

To solve the data sparsity problem, I used Laplace smoothing. This technique involves adding a small value to the count of each n-gram. It helps the model by giving some probability to unseen n-grams, which otherwise would have a probability of zero.

Here are the probabilities after applying Laplace smoothing:

```
n = 2
bigrams_list = list(ngrams(tokens, n))
bigrams_freq = Counter(bigrams_list)

# calculating the total number of unique bigrams
V = len(set(bigrams_list))

# calculating the probability of each bigram using Laplace smoothing
k = 1  # Smoothing parameter
bigram_prob_smoothed = {}
for bigram in bigrams_freq:
    # Laplace smoothing formula
    bigram_prob_smoothed[bigram] = (bigrams_freq[bigram] + k) / (len(bigrams_list) + k*V)

most_common_bigrams = bigrams_freq.most_common(10)
for bigram, freq in most_common_bigrams:
    prob_smoothed = bigram_prob_smoothed[bigram]
    print(f"Bigram: {bigram}, Frequency: {freq}, Smoothed Probability: {prob_smoothed}")
```

```
Bigram: ('Ibn', 'Jad'), Frequency: 227, Smoothed Probability: 0.003310681303362955
Bigram: ('Tarzan', 'Apes'), Frequency: 57, Smoothed Probability: 0.0008421908578730324
Bigram: ('Project', 'Gutenberg™'), Frequency: 56, Smoothed Probability: 0.0008276703258407388
Bigram: ('Sir', 'Richard'), Frequency: 52, Smoothed Probability: 0.0007695881977115641
Bigram: ('Sir', 'James'), Frequency: 51, Smoothed Probability: 0.0007550676656792705
Bigram: ('Sir', 'Malud'), Frequency: 39, Smoothed Probability: 0.0005808212812917465
Bigram: ('Project', 'Gutenberg'), Frequency: 28, Smoothed Probability: 0.0004210954289365162
Bigram: ('sir', 'knight'), Frequency: 27, Smoothed Probability: 0.0004065748969042226
Bigram: ('Princess', 'Guinalda'), Frequency: 27, Smoothed Probability: 0.0004065748969042226
Bigram: ('Knights', 'Nimmr'), Frequency: 25, Smoothed Probability: 0.00037753383283963525
```

PART3

I decided not to remove stopwords, as it caused some problems in text generation.

For each sentence, I iteratively generated up to 10 more tokens. I did this by getting the last word in the current sentence and finding all bigrams that start with this word. I then selected the next word based on the bigram probabilities from the trained model.

The following code will generate 10 tokens for each of the two sentences using the trained bigram language model. Note that stopwords are not removed in this part, as removing stopwords causes some issues in text generation.

```
# first Sentence
sen1 = "Knowing well the windings of the trail he"
for _ in range(10):  # complete to 10 tokens

    # last word in the current text
    last_word = sen1.split()[-1]

    # getting all bigrams that start with last word
    possible_bigrams = [bigram for bigram in bigram_prob_smoothed if bigram[0] == last_word]
    if not possible_bigrams:
        break
    # choosing next word base on the probabilities
    next_bigram = max(possible_bigrams, key=lambda x: bigram_prob_smoothed[x])
    sen1 += " " + next_bigram[1]

# second sentence
sen2 = "For half a day he lolled on the huge back and"
for _ in range(10):  # complete to 10 tokens

    # last word in the current text
    last_word = sen2.split()[-1]
    # getting all bigrams that start with last word
    possible_bigrams = [bigram for bigram in bigram_prob_smoothed if bigram[0] == last_word]
    if not possible_bigrams:
        break

    # choosing the next word base on probabilities
    next_bigram = max(possible_bigrams, key=lambda x: bigram_prob_smoothed[x])
    sen2 += " " + next_bigram[1]
```

the generated sentences are:

```
print("First Sentence:", sen1)
print("Second Sentence:", sen2)
```

```
First Sentence: Knowing well the windings of the trail he had been a great apes the Sepulcher and the Sepulcher
Second Sentence: For half a day he lolled on the huge back and the Sepulcher and the Sepulcher and the Sepulcher and the
```

As we can see the generated text may not be very logical because it only considers bigrams. This can result in repetitive or odd sequences, especially with limited training data. This method lacks the ability to understand the meaning of the sentences, so the accuracy of the generated text is low.

PART4

In this part, I calculated the probabilities of word sequences for n=3 and n=5 using trigrams and 5-grams. I then applied Laplace smoothing to these probabilities. Finally, I generated the next 10 words for each sequence length to see how the model performed with longer sequences.

probabilities of word sequences for n=3 before Laplace smoothing:

```
Trigram: ('Project', 'Gutenberg™', 'electronic'), Frequency: 18, Probability: 0.0004871447902571042
Trigram: ('said', 'Ibn', 'Jad'), Frequency: 16, Probability: 0.00043301759133964815
Trigram: ('Project', 'Gutenberg', 'Literary'), Frequency: 13, Probability: 0.00035182679296346417
Trigram: ('Gutenberg', 'Literary', 'Archive'), Frequency: 13, Probability: 0.00035182679296346417
Trigram: ('Literary', 'Archive', 'Foundation'), Frequency: 13, Probability: 0.00035182679296346417
Trigram: ('Gutenberg™', 'electronic', 'works'), Frequency: 12, Probability: 0.00032476319350473615
Trigram: ('Project', 'Gutenberg™', 'License'), Frequency: 8, Probability: 0.00021650879566982408
Trigram: ('Sheik', 'Ibn', 'Jad'), Frequency: 7, Probability: 0.00018944519621109606
Trigram: ('menzil', 'Ibn', 'Jad'), Frequency: 7, Probability: 0.00018944519621109606
Trigram: ('Zeyd', 'Ibn', 'Jad'), Frequency: 7, Probability: 0.00018944519621109606
```

probabilities of word sequences for n=3 after Laplace smoothing:

```
Trigram: ('Project', 'Gutenberg™', 'electronic'), Frequency: 18, Smoothed Probability: 0.00025982905982905983
Trigram: ('said', 'Ibn', 'Jad'), Frequency: 16, Smoothed Probability: 0.00023247863247863248
Trigram: ('Project', 'Gutenberg', 'Literary'), Frequency: 13, Smoothed Probability: 0.00019145299145299145
Trigram: ('Gutenberg', 'Literary', 'Archive'), Frequency: 13, Smoothed Probability: 0.00019145299145299145
Trigram: ('Literary', 'Archive', 'Foundation'), Frequency: 13, Smoothed Probability: 0.00019145299145299145
Trigram: ('Gutenberg™', 'electronic', 'works'), Frequency: 12, Smoothed Probability: 0.00017777777777777779
Trigram: ('Project', 'Gutenberg™', 'License'), Frequency: 8, Smoothed Probability: 0.00012307692307692307
Trigram: ('Sheik', 'Ibn', 'Jad'), Frequency: 7, Smoothed Probability: 0.00010940017094017094
Trigram: ('menzil', 'Ibn', 'Jad'), Frequency: 7, Smoothed Probability: 0.00010940017094017094
Trigram: ('Zeyd', 'Ibn', 'Jad'), Frequency: 7, Smoothed Probability: 0.00010940017094017094
```

generated text using 3-grams:

```
First Sentence: Knowing well the windings of the trail he took short cuts swinging through the enemies lines How passed
Second Sentence: For half a day he lolled on the huge back and Usha tore through the neck and could not decipher their
```

probabilities of word sequences for n=5 before Laplace smoothing:

```
5-gram: ('Project', 'Gutenberg', 'Literary', 'Archive', 'Foundation'), Frequency: 13, Probability: 0.000351845837393093
5-gram: ('phrase', '“', 'Project', 'Gutenberg', '”'), Frequency: 4, Probability: 0.00010826025765941322
5-gram: ('United', 'States', 'check', 'laws', 'country'), Frequency: 3, Probability: 8.119519324455992e-05
5-gram: ('water', 'hole', 'smooth', 'round', 'rocks'), Frequency: 3, Probability: 8.119519324455992e-05
5-gram: ('use', 'anyone', 'anywhere', 'United', 'States'), Frequency: 2, Probability: 5.413012882970661e-05
5-gram: ('anyone', 'anywhere', 'United', 'States', 'parts'), Frequency: 2, Probability: 5.413012882970661e-05
5-gram: ('anywhere', 'United', 'States', 'parts', 'world'), Frequency: 2, Probability: 5.413012882970661e-05
5-gram: ('United', 'States', 'parts', 'world', 'cost'), Frequency: 2, Probability: 5.413012882970661e-05
5-gram: ('States', 'parts', 'world', 'cost', 'almost'), Frequency: 2, Probability: 5.413012882970661e-05
5-gram: ('parts', 'world', 'cost', 'almost', 'restrictions'), Frequency: 2, Probability: 5.413012882970661e-05
```

probabilities of word sequences for n=5 after Laplace smoothing:

```
5-gram: ('Project', 'Gutenberg', 'Literary', 'Archive', 'Foundation'), Frequency: 13, Smoothed Probability: 0.0001897018970189702
5-gram: ('phrase', '“', 'Project', 'Gutenberg', '”'), Frequency: 4, Smoothed Probability: 6.775067750677507e-05
5-gram: ('United', 'States', 'check', 'laws', 'country'), Frequency: 3, Smoothed Probability: 5.4200542005420054e-05
5-gram: ('water', 'hole', 'smooth', 'round', 'rocks'), Frequency: 3, Smoothed Probability: 5.4200542005420054e-05
5-gram: ('use', 'anyone', 'anywhere', 'United', 'States'), Frequency: 2, Smoothed Probability: 4.065040650406504e-05
5-gram: ('anyone', 'anywhere', 'United', 'States', 'parts'), Frequency: 2, Smoothed Probability: 4.065040650406504e-05
5-gram: ('anywhere', 'United', 'States', 'parts', 'world'), Frequency: 2, Smoothed Probability: 4.065040650406504e-05
5-gram: ('United', 'States', 'parts', 'world', 'cost'), Frequency: 2, Smoothed Probability: 4.065040650406504e-05
5-gram: ('States', 'parts', 'world', 'cost', 'almost'), Frequency: 2, Smoothed Probability: 4.065040650406504e-05
5-gram: ('parts', 'world', 'cost', 'almost', 'restrictions'), Frequency: 2, Smoothed Probability: 4.065040650406504e-05
```

generated text using 5-grams:

```
print( Second Sentence: , generated_text2)
First Sentence: Knowing well the windings of the trail he took short cuts swinging through the branches of the trees
Second Sentence: For half a day he lolled on the huge back and a suddenly down felled heard when I far up Apes
```

Comparison:

first let see original sentences:

1. Knowing well the windings of the trail he took short cuts, swinging through the branches of the trees
2. For half a day he lolled on the huge back, listening to Manu the Monkey chattering and scolding among the trees.

generated text using bigrams:

1. Knowing well the windings of the trail he had been a great apes the Sepulcher and the Sepulcher
2. For half a day he lolled on the huge back and the Sepulcher and the Sepulcher and the Sepulcher and the

generated text using trigrams:

1. Knowing well the windings of the trail he took short cuts swinging through the enemies lines How passed
2. For half a day he lolled on the huge back and Usha tore through the neck and could not decipher their

generated text using 5-grams:

1. Knowing well the windings of the trail he took short cuts swinging through the branches of the trees
2. For half a day he lolled on the huge back and a suddenly down felled heard when I far up Apes

When we compare the text generated by different models:

1. Bigrams (n=2):

- The generated text is not suitable and doesn't make complete sense.
- It struggles to create sentences that sound natural because it only considers two words at a time.

2. Trigrams (n=3):

- The generated text is more coherent compared to bigrams.
- It can produce sentences that make more sense and have better structure.

3. 5-grams (n=5):

- generated text is the most coherent and contextually relevant among the three.
- It can create sentences that closely resemble the original text.

results:

- Increasing n enables a model considering more words together, which can improve the quality of the generated text.
- higher n-grams can also lead to more complex models and potential issues with rare or unseen sequences.
- Trigrams and 5-grams generally perform better than bigrams in generating meaningful text, with 5-grams providing the highest quality but also requiring more computational resources.

PART5

increasing n is theoretically possible but not always practical due to several reasons

Firstly, as n increases, the number of unique n-grams in the training data also increases. This can lead to data sparsity issues, and many n-grams occur very rarely or are unseen in the training data and this makes it challenging for the model to accurately estimate probabilities for all possible n-grams.

Secondly, the computational complexity of the model increases with n as well. Storing and processing a larger number of n-grams requires more memory and processing power, which can be impractical for very large values of n.

Additionally, with a very large n, the model might start memorizing the training data instead of learning general patterns. This can lead to overfitting, where the model performs well on the training data but not very well on unseen data.

as n increases, the model becomes more complex and harder to interpret. This makes it challenging to understand why the model makes certain predictions.

QUESTION 4

PART1

test function:

I wrote the **test_ngram** function to classify reviews as positive or negative based on the frequencies of positive and negative n-grams in each review. It takes the test data, positive and negative n-gram frequencies, and the size of the n-grams as inputs. For each review, it calculates the total probability of positive and negative n-grams and predicts a label based on which probability is higher.

The test function returns a list of predicted labels for the test data, helping me evaluate the performance of the n-gram models on new reviews.

```
def test_ngram(data, positive_freq, negative_freq, n):

    pred_labels = []

    for index, row in data.iterrows():
        # tokenize the review into n-grams
        grams = get_ngrams(row['review'], n)

        # total probability of positive and negative n-grams
        positive_prob = sum([positive_freq[gram] for gram in grams])
        negative_prob = sum([negative_freq[gram] for gram in grams])

        # predict the label based on which probability is higher
        pred_label = 1 if positive_prob > negative_prob else 0


        pred_labels.append(pred_label)


    return pred_labels
```

PART2

i calculated the model accuracy as follows:

```
from sklearn.metrics import accuracy_score


actual_labels = test_data['polarity']


pred_labels = test_ngram(test_data, positive_freq, negative_freq, n)


accuracy = accuracy_score(actual_labels, pred_labels)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.7653631284916201
```

out of all the reviews in the test dataset, the model's predictions are correct for 76% of them.

During training, the model learns from a dataset where each review is labeled as positive or negative. It looks at which n-grams appear more frequently in positive reviews versus negative reviews.

When it's time to make predictions on new text, the model breaks the text into n-grams and calculates the probability that each n-gram is associated with positive or negative sentiment. It then combines these probabilities to decide whether the overall sentiment of the text is positive or negative.

The model achieved an accuracy of 76%, which means that it correctly predicted the sentiment for 76% of the reviews in the test dataset. This indicates that the model is **reasonably effective** at understanding the sentiment of text, but there is still room for improvement.