Natural Language Processing
Assignment 2 Report
Language Modeling

Team ID: 4

1. **Using the given data to create a processed corpus**

```
df['lemmatized_comments'] = df['Comment'].apply(lemmatize_sentence)
```

2. **Using the NLTK Tokenizer to clean the data set.**

Google Colab code used for cleaning the dataset.

```
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()

def lemmatize_sentence(sentence):
    sentences = sent_tokenize(sentence)  # Tokenize into sentences
    lemmatized_sentences = []

    for sent in sentences:
        words = word_tokenize(sent.lower())  # Tokenize and lowercase
        lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
        lemmatized_sentence = ' '.join(lemmatized_words)
        lemmatized_sentences.append(lemmatized_sentence)

    return ' '.join(lemmatized_sentences)

# Apply the lemmatize_sentence function to each row in the DataFrame
df['lemmatized_comments'] = df['Comment'].apply(lemmatize_sentence)

# Display the DataFrame with the lemmatized comments
print(df)
```

```
import re
import string
def remove_punctuation(text):
    # Use a regular expression to remove punctuation
      return re.sub(f'[{string.punctuation}]', '', text)

# Apply the remove_punctuation function to the 'text' column
new_df['Comment'] = new_df['Comment'].apply(remove_punctuation)
```

```
[ ]  new_df.to_csv('preprocessed_data.csv', encoding='utf-8')
```

```
[ ]  new_df = new_df.drop('lemmatized_sentences', axis=1)
```

```
[ ]  import pandas as pd
     new_df=pd.read_csv('/content/lemmatized_sentences (1).csv')
```

[Note]: Pre-processed data contains data in lowercase, without punctuations and without 'http links'

3. Dividing data into 80:20 (Training: Testing data)

```
[27] import numpy as np
     np.random.seed(0)
     train_set, validation_set = train_test_split(proc_sents, test_size=0.2)
```

4. Creating a dictionary for n-grams using training data:

```
     for sentence in train_set:
         for token in sentence:
             if(token not in unigrams): unigrams[token] = 0
             unigrams[token] += 1
```

```
[31] # word_to_int
     print({k: word_to_int[k] for k in list(word_to_int)[:3]})

     {'you': 0, 'dont': 1, 'have': 2}
```

```
[32] for sentence in train_set:
         for i in range(3 , len(sentence)):
             quadgram = (sentence[i-3] , sentence[i-2] , sentence[i-1], sentence[i])
             if(quadgram not in quadgrams): quadgrams[quadgram] = 0
             quadgrams[quadgram] += 1
```

```
[33] for sentence in train_set:
         for i in range(2 , len(sentence)):
             trigram = (sentence[i-2] , sentence[i-1], sentence[i])
             if(trigram not in trigrams): trigrams[trigram] = 0
             trigrams[trigram] += 1
```

```
[34] for sentence in train_set:
         for i in range(1 , len(sentence)):
             bigram = (sentence[i-1], sentence[i])
             if(bigram not in bigrams): bigrams[bigram] = 0
             bigrams[bigram] += 1
```

5. Perplexity of n-grams (before smoothing)

Calculating the perplexity score of the various models on the test data. (Unigram/Bigram/Trigram/Quadragram)

We know the formula for perplexity of a sentence in an ngram model is:

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1, w_2, w_3, \ldots, w_N)}} \tag{1}$$

$$\sim \sqrt[N]{\frac{1}{\prod_{i=1}^{N} P(w_i|w_1, w_2, \ldots, w_{i-1})}} \tag{2}$$

where N is the number of words in the sentence and P is the probability of the sentence. Here we use the Markov approximation to change the probability of the sentence to the product of the probabilities of the words only for a fix n words behind it. n being the order of the ngram model.

The way we get the probabilites is:

$$P(w_i|w_1, w_2, \ldots, w_{i-1}) = \frac{P(w_1, w_2, \ldots, w_{i-1}, w_i)}{P(w_1, w_2, \ldots, w_{i-1})} \tag{3}$$

$$= \frac{C(w_1, w_2, \ldots, w_{i-1}, w_i)}{C(w_1, w_2, \ldots, w_{i-1})} \tag{4}$$

where C is the count of the word tuple in the corpus.

So this example for the bigram would be:

$$P(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})} \tag{5}$$

> Our solution for the problem of tuples not in the corpus:

For the direct case, when we do have either the numerator $C(w_i, w_{i-1})$ or $C(w_{i-1})$ as zero in the corpus due to the tuple being missing, we replace the probability with some other expected value.

In this implementation, we estimated the value of probability for the missing tuples case by the **inverse of the vocabulary size.**.

The reason for this being that perplexity tries to measure the uncertainity in predicting the next word of a sentence, and its value corresponds to how many words from the corpus it is equally confused from. Hense if a said tuple is not present in the training data, then the model will be equally confused between all the words in the vocabulary, when trying to predict the next word.

Thus

$$P(w_i|w_1, w_2, \ldots, w_{i-1}) = \frac{1}{V}$$

where V is the vocabulary size, if the tuple $w_i, w_{i-1}, \ldots$ is not present in the training data.

> Our solution for numerical stability:

We know that the probability of a sentence is the product of the probabilities of the words in the sentence. But the problem with this is that the product of many small numbers can be very small and can cause numerical instability.

So we take the log of the probabilities and add them instead of multiplying them. This is equivalent to multiplying the probabilities, and gives the correct values instead of infinities.

We handled the case for Infinity as well.

a. Unigram

```
## unigrams
total_sum = sum(unigrams.values())
total_perp = 0
count_sent = 0

for sentence in validation_set:

    count_sent += 1
    if(len(sentence) < 1): continue
    else:
        perp = 1
        n = len(sentence)
        log_perp = 0
        for i in range(1 , n):
            n_gram = ( sentence[i])

            if(n_gram[0] not in unigrams) :
                log_perp += math.log(total_sum)

            else:
                log_perp += math.log(total_sum/unigrams[n_gram[0]])
        log_perp = (1/n)*log_perp
        total_perp += math.exp(log_perp)
print("average perplexity of bigram is: ",total_perp / count_sent)
```

```
average perplexity of bigram is:  2431.529369177447
```

b. Bigram

```
## Bigrams
total_perp = 0
count_sent = 0

for sentence in validation_set:

    count_sent += 1
    if(len(sentence) < 2): continue
    else:
        perp = 1
        n = len(sentence)
        log_perp = 0
        for i in range(1 , n):
            n_gram = (sentence[i-1], sentence[i])

            if(n_gram[0] not in unigrams) or (n_gram not in bigrams):
                log_perp += math.log(len(unigrams))

            else:
                log_perp += math.log(unigrams[n_gram[0]]/(bigrams[n_gram]))
        log_perp = (1/n)*log_perp
        total_perp += math.exp(log_perp)
print("average perplexity of bigram is: ",total_perp / count_sent)
```

```
average perplexity of bigram is:  592.2785718972964
```

c. Trigram

```
## Trigram
total_perp = 0
count_sent = 0

for sentence in validation_set:
    count_sent += 1
    if(len(sentence) < 3): continue
    else:
        n = len(sentence)
        log_perp = 0
        for i in range(2 , n):
            n_gram = (sentence[i-2], sentence[i-1], sentence[i])

            if(n_gram[:-1] not in bigrams) or (n_gram not in trigrams):
                log_perp += math.log(len(unigrams))

            else:
                log_perp += math.log(bigrams[n_gram[:-1]]/(trigrams[n_gram]))
        log_perp = (1/n)*log_perp
        total_perp += math.exp(log_perp)
print("average perplexity of trigram is: ",total_perp / count_sent)
```

```
average perplexity of trigram is:  1973.644316192001
```

d. Quadgram

```
## Quadgram
total_perp = 0
count_sent = 0

for sentence in validation_set:
#     print(sentence)
    count_sent += 1
    if(len(sentence) < 4): continue
    else:
        n = len(sentence)
        log_perp = 0
        for i in range(3 , n):
            n_gram = (sentence[i-3], sentence[i-2], sentence[i-1], sentence[i])

            if(n_gram[:-1] not in trigrams) or (n_gram not in quadgrams):
                log_perp += math.log(len(unigrams)) # 7 to 6
#                 print(len(quadgrams))

            else:
                log_perp += math.log(trigrams[n_gram[:-1]]/(quadgrams[n_gram]))
#                 print(trigrams[n_gram[:-1]]/(quadgrams[n_gram]), trigrams[n_gram[:-1]] , (quadgrams[n_gram]) )
        log_perp = (1/n)*log_perp
        total_perp += math.exp(log_perp)
print("average perplexity of quadgram is: ",total_perp / count_sent)
```

```
average perplexity of quadgram is:  6130.475783783798
```

6. Perplexity of n-grams (after Laplace-smoothing)

Here we use the method of laplace smoothign to smooth the probabilities of the tuples. This is done by adding 1 to the count of each tuple. This is done to avoid the problem of zero probabilities.

The formula for the probability of a tuple after laplace smoothing is:

$$P(w_i|w_1, w_2, \ldots, w_{i-1}) = \frac{C(w_1, w_2, \ldots, w_{i-1}, w_i) + 1}{C(w_1, w_2, \ldots, w_{i-1}) + V} \tag{6}$$

where V is the vocabulary size.

Here we apply the laplace smoothing method to all the previous models.

a. Unigram

```
total_sum = sum(unigrams.values())
total_perp = 0
count_sent = 0

for sentence in validation_set:
    count_sent += 1
    if len(sentence) < 1:
        continue
    else:
        perp = 1
        n = len(sentence)
        log_perp = 0
        for i in range(n):
            n_gram = sentence[i]

            if n_gram[0] in unigrams:
                smooth_prob = ((total_sum + len(unigrams))/(unigrams[n_gram[0]] + 1))
                log_perp += math.log(smooth_prob)
            else:
                log_perp += math.log(1 / (total_sum + len(unigrams)))

        log_perp = (1 / n) * log_perp
        total_perp += math.exp(log_perp)

print("Average perplexity of Laplace smoothed unigram is:", total_perp / count_sent)
```

```
Average perplexity of Laplace smoothed unigram is: 2652.872424332836
```

b. Bigram

```
## Bigram Laplace smoothing

total_perp = 0
count_sent = 0

for sentence in validation_set:
#     print(sentence)

    count_sent += 1
    if(len(sentence) < 2): continue
    else:
        perp = 1
        n = len(sentence)
        log_perp = 0
        for i in range(1 , n):
            n_gram = (sentence[i-1], sentence[i])

            smooth_prob = (unigrams.get(n_gram[0],0) + len(unigrams))/(bigrams.get(n_gram,0)+1)
            log_perp += math.log(smooth_prob)

        log_perp = (1/n)*log_perp

        total_perp += math.exp(log_perp)
print("average perplexity of Laplace smoothened bigram is: ",total_perp / count_sent)
```

```
average perplexity of Laplace smoothened bigram is:  2950.6115686115413
```

c. Trigram

```
## Trigram LAPLAACE SMOOTHING

total_perp = 0
count_sent = 0

for sentence in validation_set:

    count_sent += 1
    if(len(sentence) < 3): continue
    else:
        n = len(sentence)
        log_perp = 0
        for i in range(2 , n):
            n_gram = (sentence[i-2], sentence[i-1], sentence[i])

            smooth_prob = (bigrams.get(n_gram[:-1],0) + len(unigrams))/(trigrams.get(n_gram,0)+1)
            log_perp += math.log(smooth_prob)

        log_perp = (1/n)*log_perp

        total_perp += math.exp(log_perp)
print("average perplexity of Laplace smoothened trigram is: ",total_perp / count_sent)
```

```
average perplexity of Laplace smoothened trigram is:  10617.441382254596
```

d. Quadgram

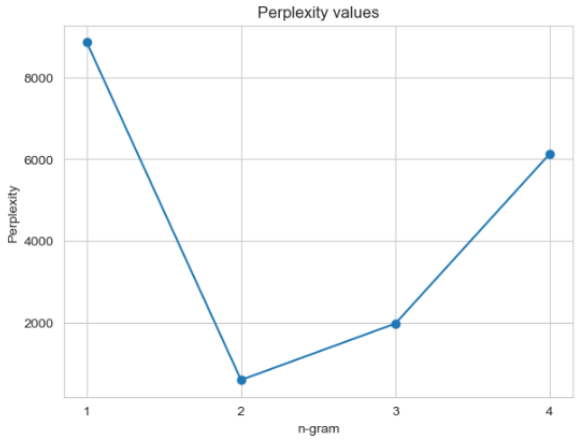
```
## Quadgram LAPLAACE SMOOTHING

total_perp = 0
count_sent = 0

for sentence in validation_set:
#     print(sentence)

    count_sent += 1
    if(len(sentence) < 4): continue
    else:
        n = len(sentence)
        log_perp = 0
        for i in range(3 , n):
            n_gram = (sentence[i-3], sentence[i-2], sentence[i-1], sentence[i])

            smooth_prob = (trigrams.get(n_gram[:-1],0) + len(unigrams))/(quadgrams.get(n_gram,0)+1)
            log_perp += math.log(smooth_prob)

        log_perp = (1/n)*log_perp

        total_perp += math.exp(log_perp)
print("average perplexity of Laplace smoothened quadgram is: ",total_perp / count_sent)
```

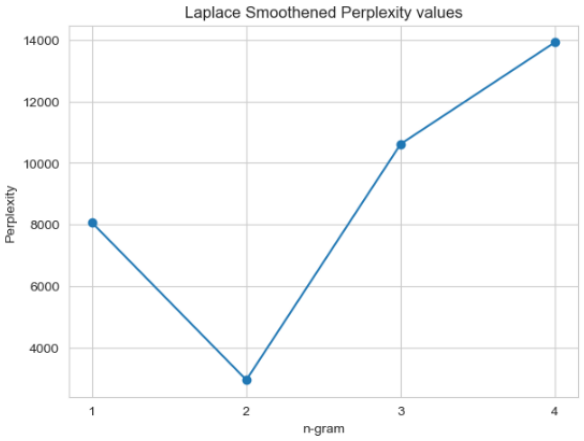
```
average perplexity of Laplace smoothened quadgram is:  13924.504467870463
```

7. Comparison of Perplexity of n-grams Before and After Smoothing:

	Before Smoothing	After Laplace Smoothing
Unigrams	2431.53	2652.87
Bigrams	592.28	2950.61
Trigrams	1973.64	10617.44
Quadgrams	6130.48	13924.50

Plotting the perplexity values





Conclusions and observations.

We can make two important conclusions:-

For both models, the perplexity value decreases with the increase in model size from 1 to 2, but it increases again after 2.

The perplexity of the Laplace model is higher than the unsmoothed model.

The perplexity value for the unigram model is much higher than the bigram model. This is because the unigram model does not consider the context of the word, and thus, the probability of the word is not dependent on the previous word. This is not the case for the bigram model, where the probability of the word is dependent on the previous word. Thus, the Bigram model can predict the next word better than the Unigram model.

However, the perplexity of models further increases instead of decreasing the logic. This happens as the model size increases to three and four, the combinations of tuples increase exponentially, and many combinations in the test data are absent in the training data. We give these combinations a very low probability, and thus, more and more misses lead to a higher perplexity value.

For example, you can see that many of the quadgram models' tuples are not present in the training data. (And their low probability increases the value of perplexity.)

The perplexity of the Laplace model is higher than that of the unsmoothed model.

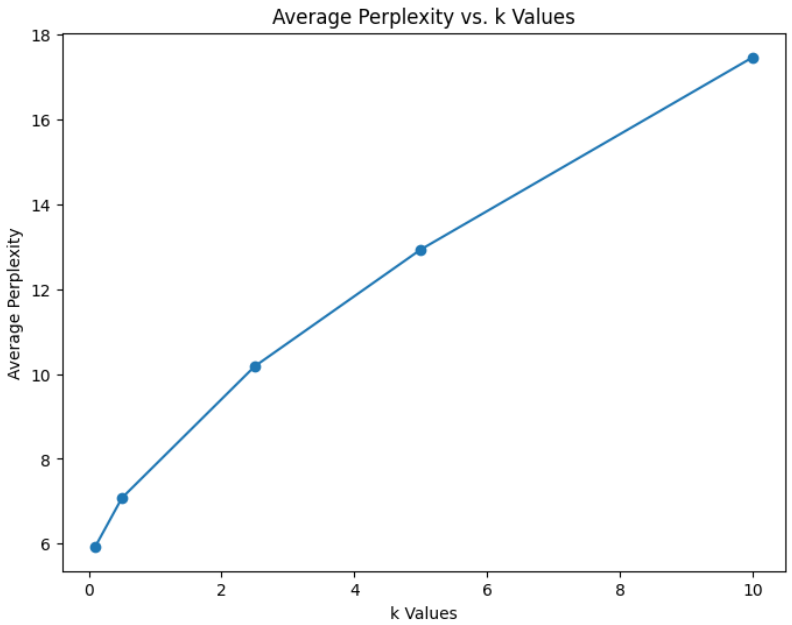
Laplace smoothing increases the value of the perplexity as the probability of each tuple decreases, we add V to the denominator for each probability. Thus, the probability of the sentence decreases, and the perplexity increases.

8. Two other smoothing methods - Good Turing and Add-k

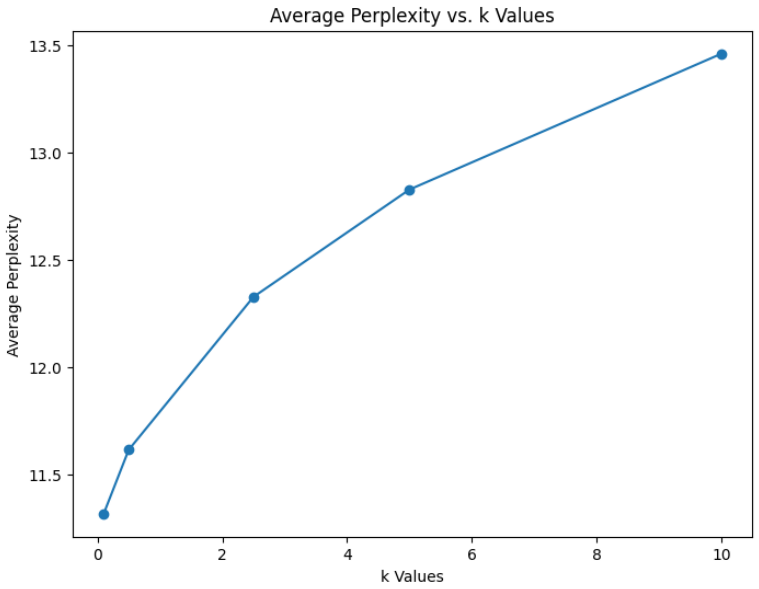
The formulae below are for the Bigram model:
P_Laplace(wi|wi−1) =c(wi−1,wi) +1/c(wi−1) +1*V
P_Add−k(wi|wi−1) =c(wi−1,wi) +k/c(wi−1) +k*V

We use 6 different values of k = 0.1, 0.5, 2.5, 5, and 10 to analyze the 2 smoothing techniques.

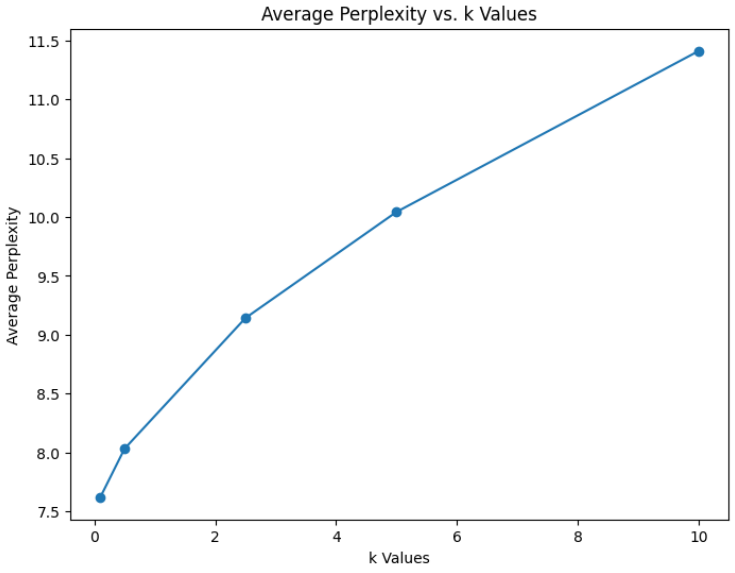
AVERAGE PERPLEXITY VS k for Quadgram:



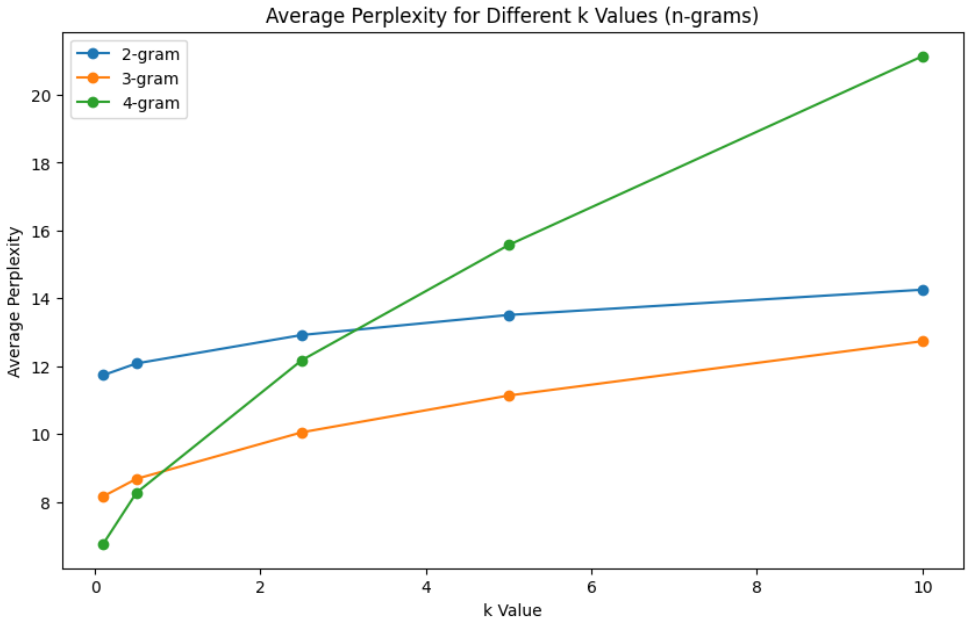
AVERAGE PERPLEXITY VS k for Bigram:



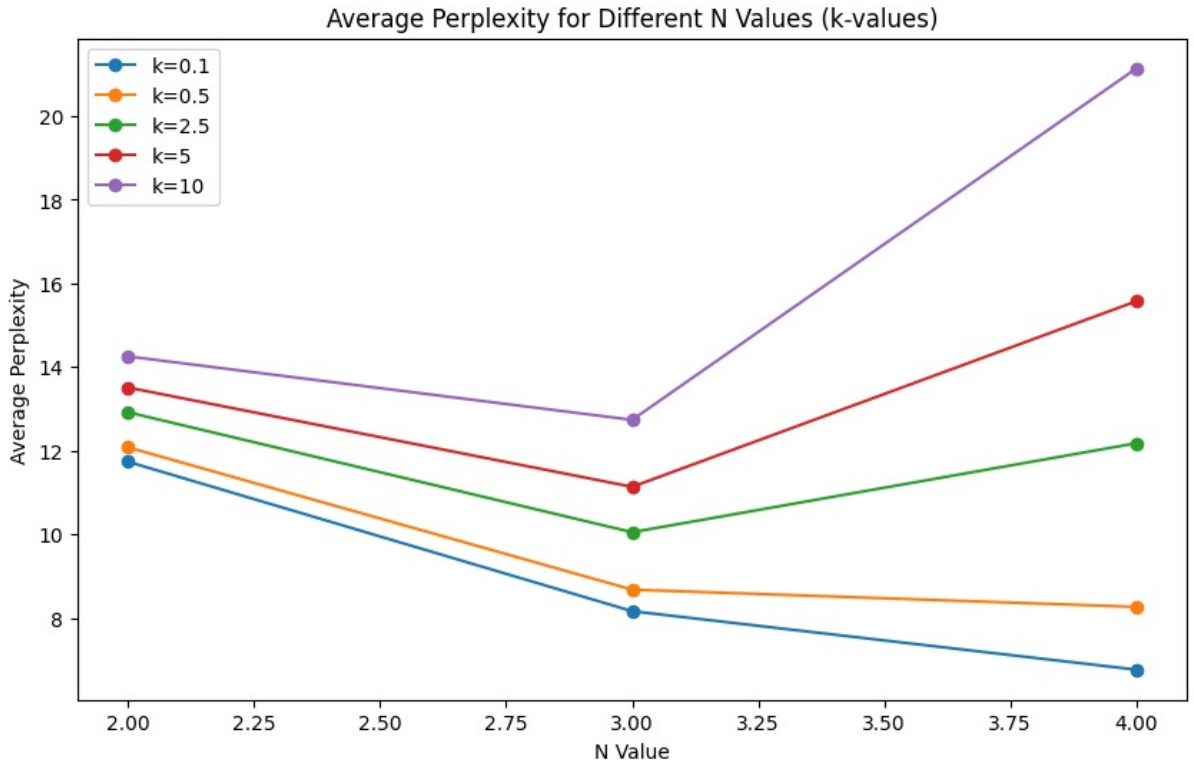
AVERAGE PERPLEXITY VS k for Trigram:



Average perplexity vs. k for different N(add-k smoothing)



Average Perplexity vs. N for different k values



Analysis:

- We used add-k, and the results for the method are shown above.
- perplexity values for Laplace are exceptional cases of add-k for the same n-gram model. Thus, add-k smoothing is preferable to Laplace because we would want to vary k to get lower perplexity values, and for Laplace, k is fixed, which is k = 1.
- The graphs for add-k are shown for 5 different values of k, and we can see that as k increases, the perplexity values increase.
- As we can see, for a particular N, the perplexity is minimal and then increases. So, there is an optimum N, which, from the graph of add-k smoothing, can be concluded as N (optimum) = 3.

Good Turing

- **For bigrams:**

```
bigram_freq ={}

for frequency in bigrams.values():
    if frequency not in bigram_freq:
        bigram_freq[frequency] = 1
    else:
        bigram_freq[frequency] += 1


bigram_freq = dict(sorted(bigram_freq.items()))

bigram_freq
```

```
# good turing (for bigrams)
total_perp = 0
count_sent = 0

for sentence in validation_set:

    count_sent += 1
    if(len(sentence) < 2): continue
    else:
        perp = 1
        n = len(sentence)
        log_perp = 0
        for i in range(1 , n):
            n_gram = (sentence[i-1], sentence[i])

            if(n_gram[0] not in unigrams) or (n_gram not in bigrams):
                log_perp += math.log(len(unigrams))

            else:
                if bigrams[n_gram]>100:
                    log_perp += math.log((unigrams[n_gram[0]])/(bigrams[n_gram]-0.75))
                elif (bigrams[n_gram] ==0):

                    log_perp += math.log((bigram_freq[1]/N0))



                else:
                    log_perp += math.log(((unigrams[n_gram[0]])
                    *bigram_freq[bigrams[n_gram]])/((bigrams[n_gram]+1)
                    *bigram_freq[bigrams[n_gram]+1]))

        log_perp = (1/n)*log_perp
        total_perp += math.exp(log_perp)
print("average perplexity of bigram after turing is: ",total_perp / count_sent)
```

```
average perplexity of bigram after turing is:  643.5744591759906
```

- **For Trigrams:**

```
trigram_freq ={}

for frequency in trigrams.values():
    if frequency not in trigram_freq:
        trigram_freq[frequency] = 1
    else:
        trigram_freq[frequency] += 1


trigram_freq = dict(sorted(bigram_freq.items()))

trigram_freq
```

```
# good turing (for trigrams)
total_perp = 0
count_sent = 0

for sentence in validation_set:

    count_sent += 1
    if(len(sentence) < 3): continue
    else:
        perp = 1
        n = len(sentence)
        log_perp = 0
        for i in range(2 , n):
            n_gram = (sentence[i-2], sentence[i-1], sentence[i])

            if(n_gram[0] not in bigrams) or (n_gram not in trigrams):
                log_perp += math.log(len(bigrams))

            else:
                if trigrams[n_gram]>100:
                    log_perp += math.log((bigrams[n_gram[0]])/(trigrams[n_gram]-0.75))
                elif (trigrams[n_gram] ==0):

                    log_perp += math.log((bigram_freq[1]/N0))



                else:
                    log_perp += math.log(((bigrams[n_gram[0]])
                    *bigram_freq[trigrams[n_gram]])/((trigrams[n_gram]+1)
                    *bigram_freq[trigrams[n_gram]+1]))

        log_perp = (1/n)*log_perp
        total_perp += math.exp(log_perp)
print("average perplexity of bigram after turing is: ",total_perp / count_sent)
```

```
average perplexity of bigram after turing is:  252658.39010734862
```

- **For Quadgrams:**

```
quadgram_freq ={}

for frequency in quadgrams.values():
    if frequency not in quadgram_freq:
        quadgram_freq[frequency] = 1
    else:
        quadgram_freq[frequency] += 1


quadgram_freq = dict(sorted(bigram_freq.items()))

quadgram_freq
```

```
[108] # good turing (for quadgrams)
      total_perp = 0
      count_sent = 0

      for sentence in validation_set:

          count_sent += 1
          if(len(sentence) < 4): continue
          else:
              perp = 1
              n = len(sentence)
              log_perp = 0
              for i in range(3 , n):
                  n_gram = (sentence[i-3], sentence[i-2], sentence[i-1], sentence[i])

                  if(n_gram[0] not in trigrams) or (n_gram not in quadgrams):
                      log_perp += math.log(len(trigrams))

                  else:
                      if quadgrams[n_gram]>100:
                          log_perp += math.log((trigrams[n_gram[0]])/(quadgrams[n_gram]-0.75))
                      elif (quadgrams[n_gram] ==0):

                          log_perp += math.log((bigram_freq[1]/N0))



                      else:
                          log_perp += math.log(((trigrams[n_gram[0]])
                          *bigram_freq[quadgrams[n_gram]])/((quadgrams[n_gram]+1)
                          *bigram_freq[quadgrams[n_gram]+1]))

              log_perp = (1/n)*log_perp
              total_perp += math.exp(log_perp)
      print("average perplexity of bigram after turing is: ",total_perp / count_sent)

      average perplexity of bigram after turing is:  399270.065792176
```

Results:

	Without Smoothing	Laplace Smoothing	Good Turing
Bigrams	592.28	2950.61	643.57
Trigrams	1973.64	10617.44	252658.39
Quadgrams	6130.48	13924.50	399270.06

GitHub repository for the assignment: [immortalcurse/NLP_Assignment2 (github.com)]
Google Colab Link: **Q1_Q8.ipynb**

Google Colab Link (with all the parts except preprocessing**):** **T4_NLP_A2(expect preprocessing).ipynb**

Contributors:

Team Members	**Roll no:**	**Contribution**
Abhay	21110004	12.5%
Adit	21110010	12.5%
Aishwarya	20110008	12.5%
Amaan	20110011	12.5%
Govardhan	20110070	12.5%
Gaurav Joshi	21110065	12.5%
Sanskriti	23120007	12.5%
Simran	20110200	12.5%