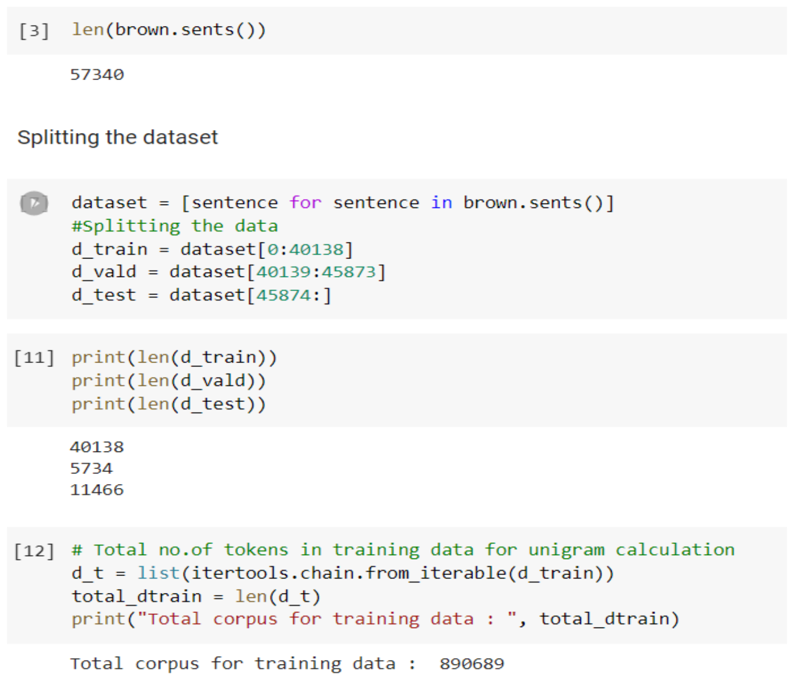
ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

IDS566: Advanced Text Analytics
Mini-Project #1

Problem 1: Language Modeling

(a) Write a code from scratch that learns unigram and bigram models on the training data as Python dictionaries. Report the perplexity of your unigram and bigram models on both the training data and test data.

Ans. We have used the brown corpus as per mentioned in the problem set. There are 57340 sentences in the brown corpus. We have split the data into training, validation and test sets. The training data has 40138, validation data has 5734 and test data has 11466 sentences. We are considering the total tokens as all the different tokens including the punctuations in the corpus. Therefore, we get the total corpus for the training dataset as 890689.



```
[3]  len(brown.sents())
```

```
57340
```

Splitting the dataset

```
dataset = [sentence for sentence in brown.sents()]
#Splitting the data
d_train = dataset[0:40138]
d_vald = dataset[40139:45873]
d_test = dataset[45874:]
```

```
[11] print(len(d_train))
     print(len(d_vald))
     print(len(d_test))
```

```
40138
5734
11466
```

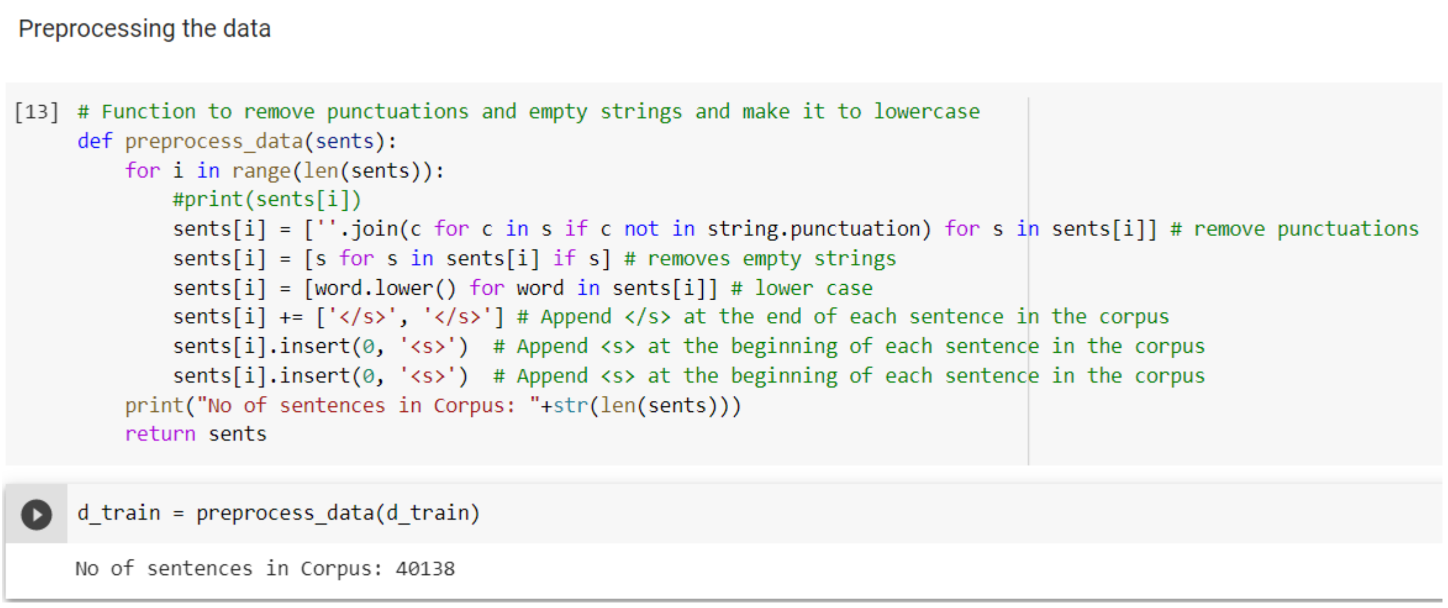
```
[12] # Total no.of tokens in training data for unigram calculation
     d_t = list(itertools.chain.from_iterable(d_train))
     total_dtrain = len(d_t)
     print("Total corpus for training data : ", total_dtrain)
```

```
Total corpus for training data :  890689
```

In the preprocessing step, we have removed the punctuations, empty strings, made all the words lowercase and appended <s><s> and </s> </s> at the beginning and end of each sentence. We have considered two sets of <s> and </s> since we are implementing the trigram model later as part of (e).

ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

Preprocessing Data



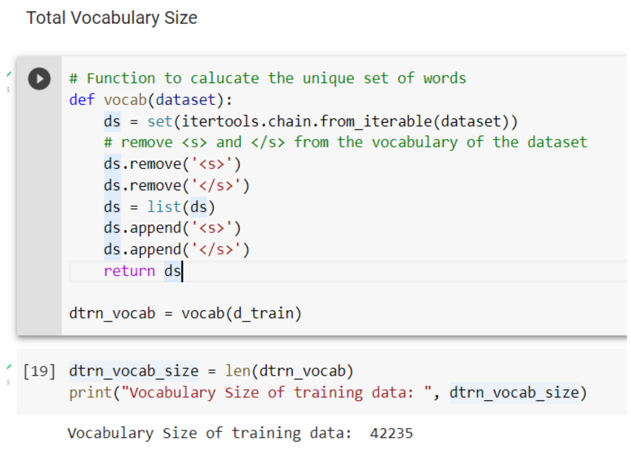
Preprocessing the data

```
[13] # Function to remove punctuations and empty strings and make it to lowercase
     def preprocess_data(sents):
         for i in range(len(sents)):
             #print(sents[i])
             sents[i] = [''.join(c for c in s if c not in string.punctuation) for s in sents[i]] # remove punctuations
             sents[i] = [s for s in sents[i] if s] # removes empty strings
             sents[i] = [word.lower() for word in sents[i]] # lower case
             sents[i] += ['</s>', '</s>'] # Append </s> at the end of each sentence in the corpus
             sents[i].insert(0, '<s>')  # Append <s> at the beginning of each sentence in the corpus
             sents[i].insert(0, '<s>')  # Append <s> at the beginning of each sentence in the corpus
         print("No of sentences in Corpus: "+str(len(sents)))
         return sents
```

```
d_train = preprocess_data(d_train)
```

```
No of sentences in Corpus: 40138
```

The function vocab() will return a list of all the unique words from the dataset passed to it. We have used it to calculate the vocabulary for the training data for now which contains 42235 unique words.



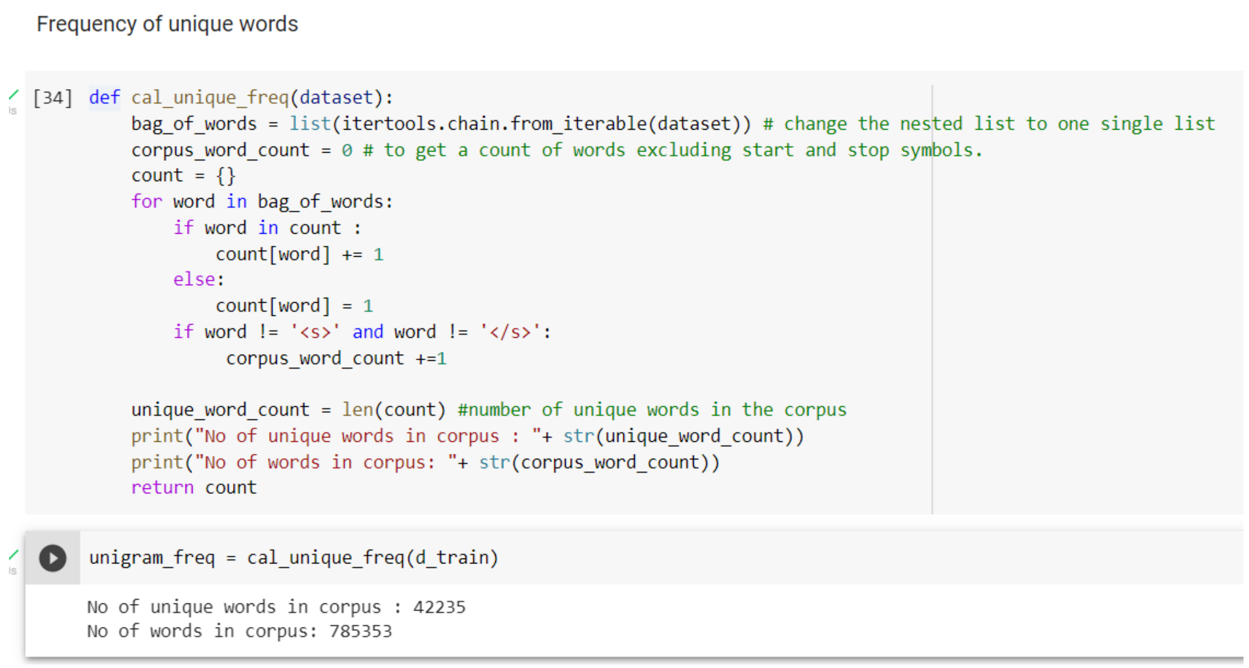
Total Vocabulary Size

```
# Function to calucate the unique set of words
def vocab(dataset):
    ds = set(itertools.chain.from_iterable(dataset))
    # remove <s> and </s> from the vocabulary of the dataset
    ds.remove('<s>')
    ds.remove('</s>')
    ds = list(ds)
    ds.append('<s>')
    ds.append('</s>')
    return ds

dtrn_vocab = vocab(d_train)
```

```
[19] dtrn_vocab_size = len(dtrn_vocab)
     print("Vocabulary Size of training data: ", dtrn_vocab_size)
```

```
Vocabulary Size of training data:  42235
```

The function cal_unique_freq() calculates the frequency of unique words from the dataset passed and returns a dictionary of unique words and their respective frequencies. We have calculated the unigram frequencies for the training dataset, and it is stored in the unigram_freq dictionary.



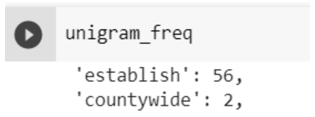
Frequency of unique words

```
[34] def cal_unique_freq(dataset):
         bag_of_words = list(itertools.chain.from_iterable(dataset)) # change the nested list to one single list
         corpus_word_count = 0 # to get a count of words excluding start and stop symbols.
         count = {}
         for word in bag_of_words:
             if word in count :
                 count[word] += 1
             else:
                 count[word] = 1
             if word != '<s>' and word != '</s>':
                  corpus_word_count +=1

         unique_word_count = len(count) #number of unique words in the corpus
         print("No of unique words in corpus : "+ str(unique_word_count))
         print("No of words in corpus: "+ str(corpus_word_count))
         return count
```

```
unigram_freq = cal_unique_freq(d_train)
```

```
No of unique words in corpus : 42235
No of words in corpus: 785353
```

The unigram frequencies look like below:



```
unigram_freq
```

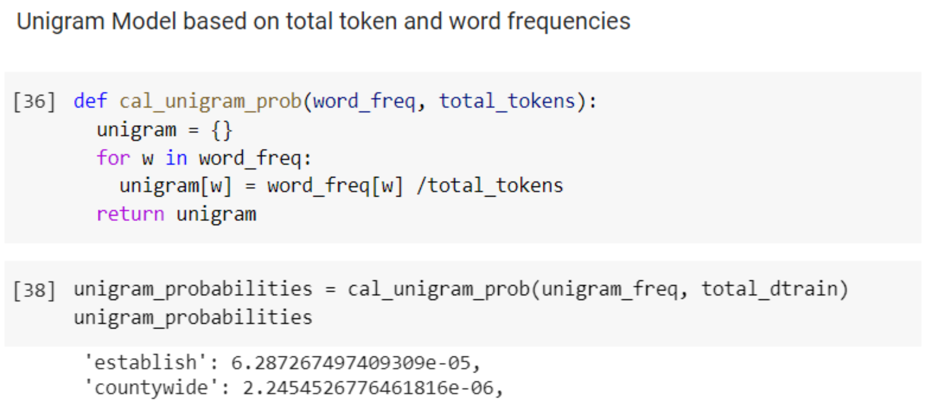
```
'establish': 56,
'countywide': 2,
```

Then we calculate the c=unigram probabilities using the maximum likelihood formula given below and we have implemented the same in the function cal_unigram_prob(). We pass the calculated unigram frequencies from the previous step and the total count of tokens in the dataset as parameters to the function and in return get a dictionary of words and their unigram probabilities.

$$\text{MLE} \quad P(w_i) = \frac{C(w_i)}{\sum_j C(w_j)} = \frac{C(w_i)}{N}$$

where P(wi) is the probability of a single word, C(wi) is the count of occurrences of that word and denominator is the total count of all words in the corpus.

Unigram Model



Unigram Model based on total token and word frequencies

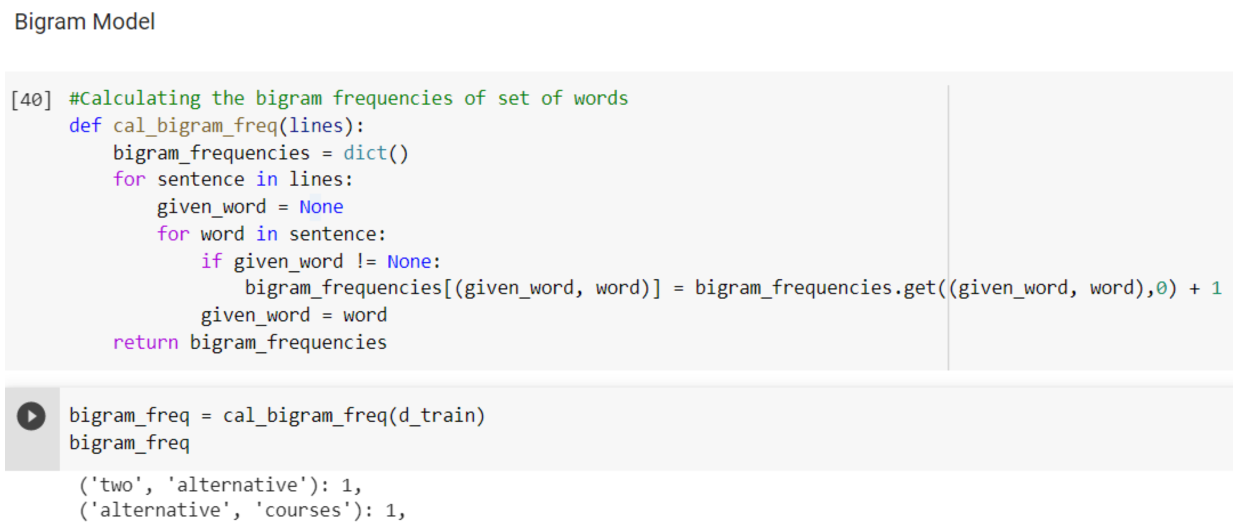
```
[36] def cal_unigram_prob(word_freq, total_tokens):
       unigram = {}
       for w in word_freq:
         unigram[w] = word_freq[w] /total_tokens
       return unigram
```

```
[38] unigram_probabilities = cal_unigram_prob(unigram_freq, total_dtrain)
     unigram_probabilities
```

```
      'establish': 6.287267497409309e-05,
      'countywide': 2.2454526776461816e-06,
```

Next, for the bigram model we calculate the bigram frequencies as the total occurrences of the previous word along with the current word. This is implemented in the function cal_bigram_freq. We pass the training dataset here as a parameter to get the bigram frequencies of the set of words. The output is a dictionary with each bigram and their respective frequencies.

Bigram Model



Bigram Model

```
[40] #Calculating the bigram frequencies of set of words
     def cal_bigram_freq(lines):
         bigram_frequencies = dict()
         for sentence in lines:
             given_word = None
             for word in sentence:
                 if given_word != None:
                     bigram_frequencies[(given_word, word)] = bigram_frequencies.get((given_word, word),0) + 1
                 given_word = word
         return bigram_frequencies
```

```
     bigram_freq = cal_bigram_freq(d_train)
     bigram_freq
```

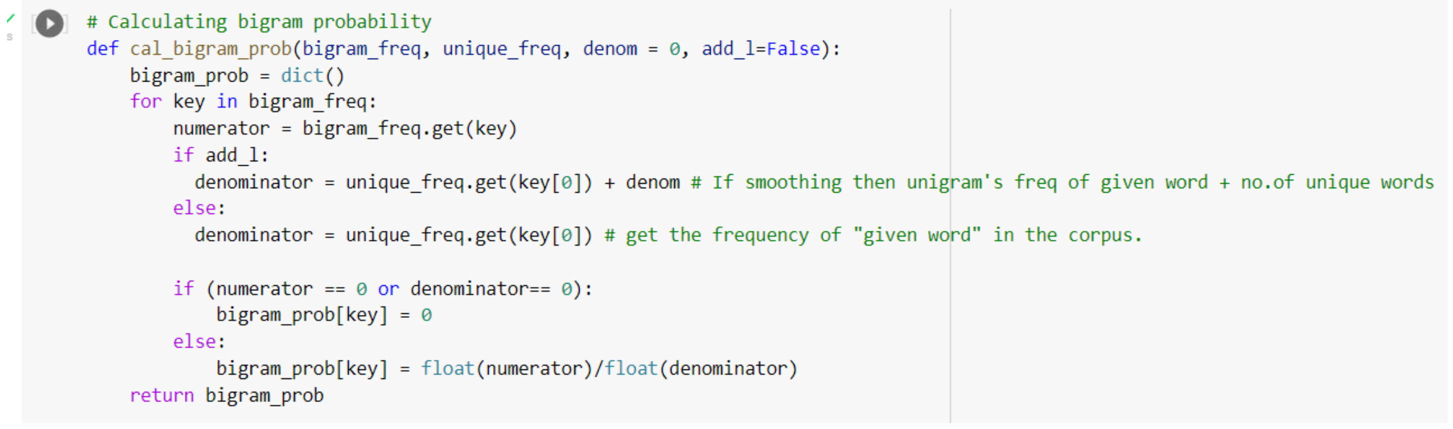
```
     ('two', 'alternative'): 1,
     ('alternative', 'courses'): 1,
```

Then we calculate the bigram probabilities using the conditional probability formula below and the same is implemented in the function cal_bigram_prob() which takes as input the unigram and bigram frequencies calculated in the previous steps along with additional parameters ‘denom’ and ‘add_l’ which are used for smoothing.

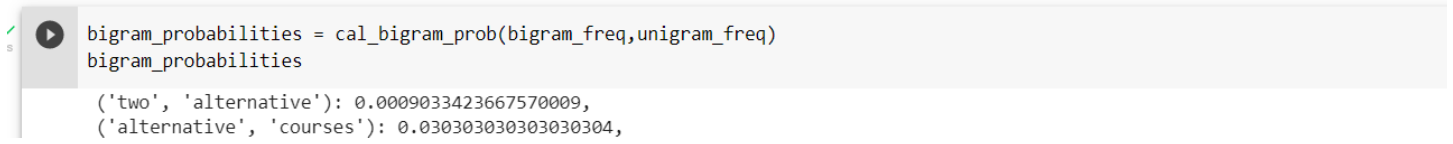
ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

$$P(\frac{w_i}{w_{i-1}}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$$

We then find the probability of a current word given the previous word using conditional probability. The numerator is the count of occurrences of the previous word along with the current word in the dataset and denominator is the total count of occurrences of the previous word alone.



```
# Calculating bigram probability
def cal_bigram_prob(bigram_freq, unique_freq, denom = 0, add_l=False):
    bigram_prob = dict()
    for key in bigram_freq:
        numerator = bigram_freq.get(key)
        if add_l:
          denominator = unique_freq.get(key[0]) + denom # If smoothing then unigram's freq of given word + no.of unique words
        else:
          denominator = unique_freq.get(key[0]) # get the frequency of "given word" in the corpus.

        if (numerator == 0 or denominator== 0):
            bigram_prob[key] = 0
        else:
            bigram_prob[key] = float(numerator)/float(denominator)
    return bigram_prob
```



```
bigram_probabilities = cal_bigram_prob(bigram_freq,unigram_freq)
bigram_probabilities
```

```
 ('two', 'alternative'): 0.0009033423667570009,
 ('alternative', 'courses'): 0.030303030303030304,
```

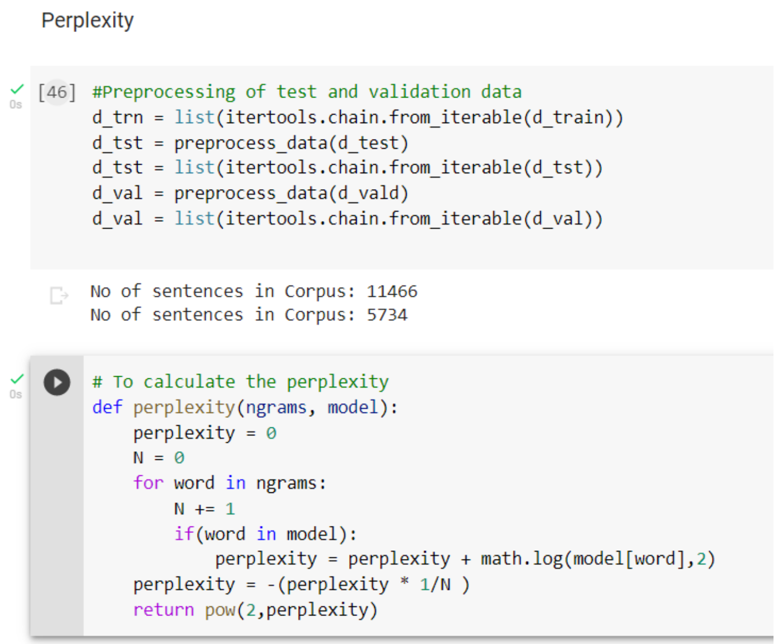
Perplexity

In order to evaluate how good our model is we find its perplexity.
Perplexity is the inverse probability of the test set, normalized by the number of words. It is equal to 2 to the power of entropy.
To calculate the perplexity of the models, we have used the function perplexity() which implements the below formula:

$$PPL = 2^H$$

$$PP(w_1, \ldots, w_n) = 2^{-\frac{1}{n}\log_2 q(w_1, \ldots, w_n)}$$

Before calculating the perplexity, we have preprocessed the validation and test data set as well as shown below.



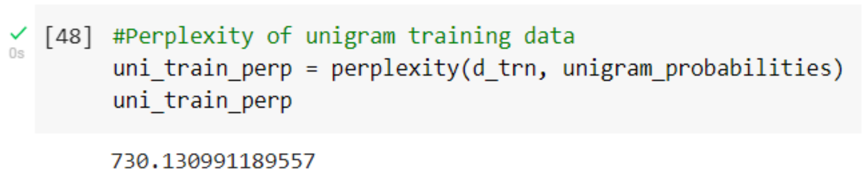
Perplexity

```
[46] #Preprocessing of test and validation data
     d_trn = list(itertools.chain.from_iterable(d_train))
     d_tst = preprocess_data(d_test)
     d_tst = list(itertools.chain.from_iterable(d_tst))
     d_val = preprocess_data(d_vald)
     d_val = list(itertools.chain.from_iterable(d_val))
```

```
No of sentences in Corpus: 11466
No of sentences in Corpus: 5734
```

```
# To calculate the perplexity
def perplexity(ngrams, model):
    perplexity = 0
    N = 0
    for word in ngrams:
        N += 1
        if(word in model):
            perplexity = perplexity + math.log(model[word],2)
    perplexity = -(perplexity * 1/N )
    return pow(2,perplexity)
```

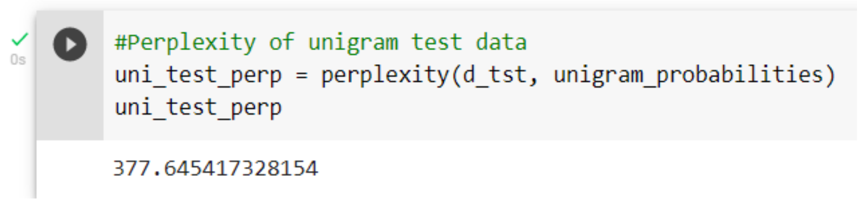
Perplexity of the unigram model on the training data :



```
[48] #Perplexity of unigram training data
     uni_train_perp = perplexity(d_trn, unigram_probabilities)
     uni_train_perp
```

```
730.130991189557
```

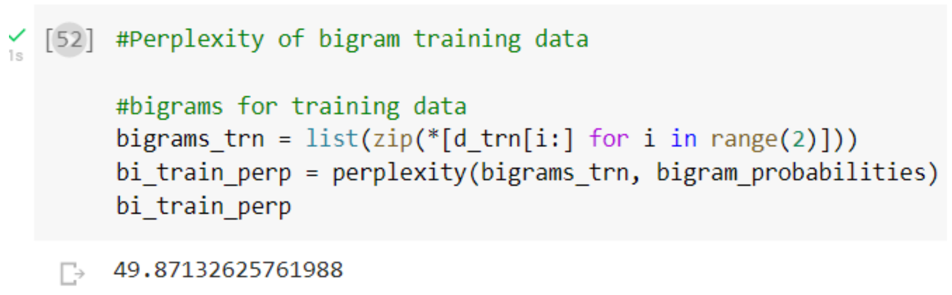
Perplexity of the unigram model on the test data :



```
#Perplexity of unigram test data
uni_test_perp = perplexity(d_tst, unigram_probabilities)
uni_test_perp
```

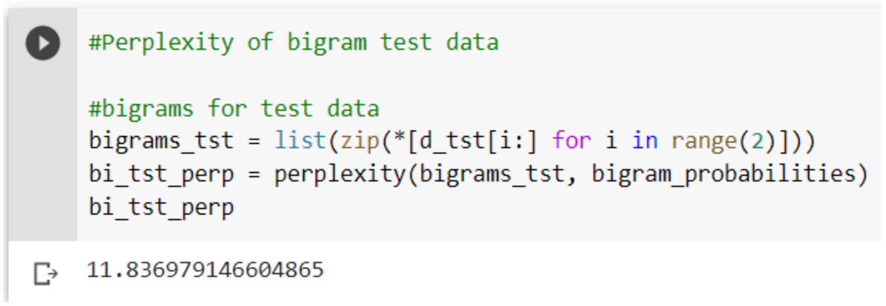
```
377.645417328154
```

Perplexity of the bigram model on the training data:



```
[52] #Perplexity of bigram training data

     #bigrams for training data
     bigrams_trn = list(zip(*[d_trn[i:] for i in range(2)]))
     bi_train_perp = perplexity(bigrams_trn, bigram_probabilities)
     bi_train_perp

     49.87132625761988
```

Perplexity of the bigram model on the test data:



```
#Perplexity of bigram test data

#bigrams for test data
bigrams_tst = list(zip(*[d_tst[i:] for i in range(2)]))
bi_tst_perp = perplexity(bigrams_tst, bigram_probabilities)
bi_tst_perp

11.836979146604865
```

Perplexity of upper n-gram models is lowest as they use the context. Lower the perplexity, better the model.

(b) Implement add-lambda smoothing method. With varying lambda values. Draw a curve that measures your perplexity change over different lambda values on the developing data.

Ans. In the above question, we have used the Maximum Likelihood Estimation(MLE) for training the parameters of the N-gram models. The problem with MLE is that it assigns zero probabilities to unseen words making the complete probability as zero. If there is a word in the test set which is not seen in the training dataset then the frequency of this word would be zero leading to zero probability.
In order to avoid this issue, smoothing is done. Smoothing takes some probability mass from the events seen in the training dataset and assigns it to the unseen events. Add-lambda smoothing or Laplace smoothing adds the lambda values to the count of all n-grams in the training set before normalizing into probabilities.

MLE:

$$P(w) = C(w)/N$$

$$P(w_n|w_{n-1}) = C(w_{n-1}\ w_n)/C(w_{n-1})$$

where P(w) is the unigram probability, $P(w_n|w_{n-1})$ is the bigram probability, C(w) is the count of occurrence of w in the training set, $C(w_{n-1}\ w_n)$ is the count of bigram $(w_{n-1}\ w_n)$ in the training set, N is the total number of word tokens in the training set.

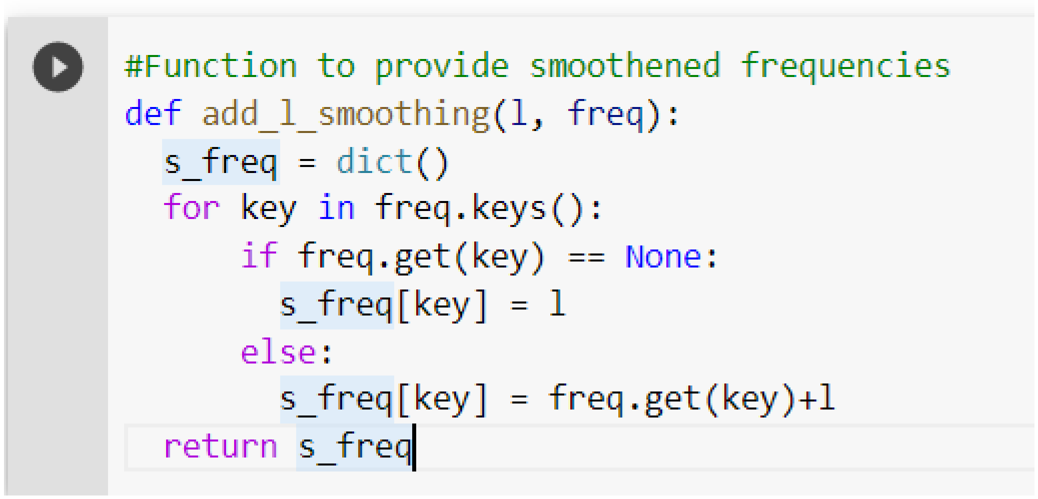
Add-1 smoothing for unigrams:

$P_{Laplace}(w) = (C(w)+1)/N+|V|$
Here, N is the total number of tokens in the training set and |V| is the size of the vocabulary represents the unique set of words in the training set.
As we have added 1 to the numerator, we have to normalize that by adding the count of unique words with the denominator in order to normalize.

Add-1 smoothing for bigrams:

$P_{Laplace}(w_n|w_{n-1}) = (C(w_{n-1}\ w_n)+1)/C(w_{n-1})+|V|$

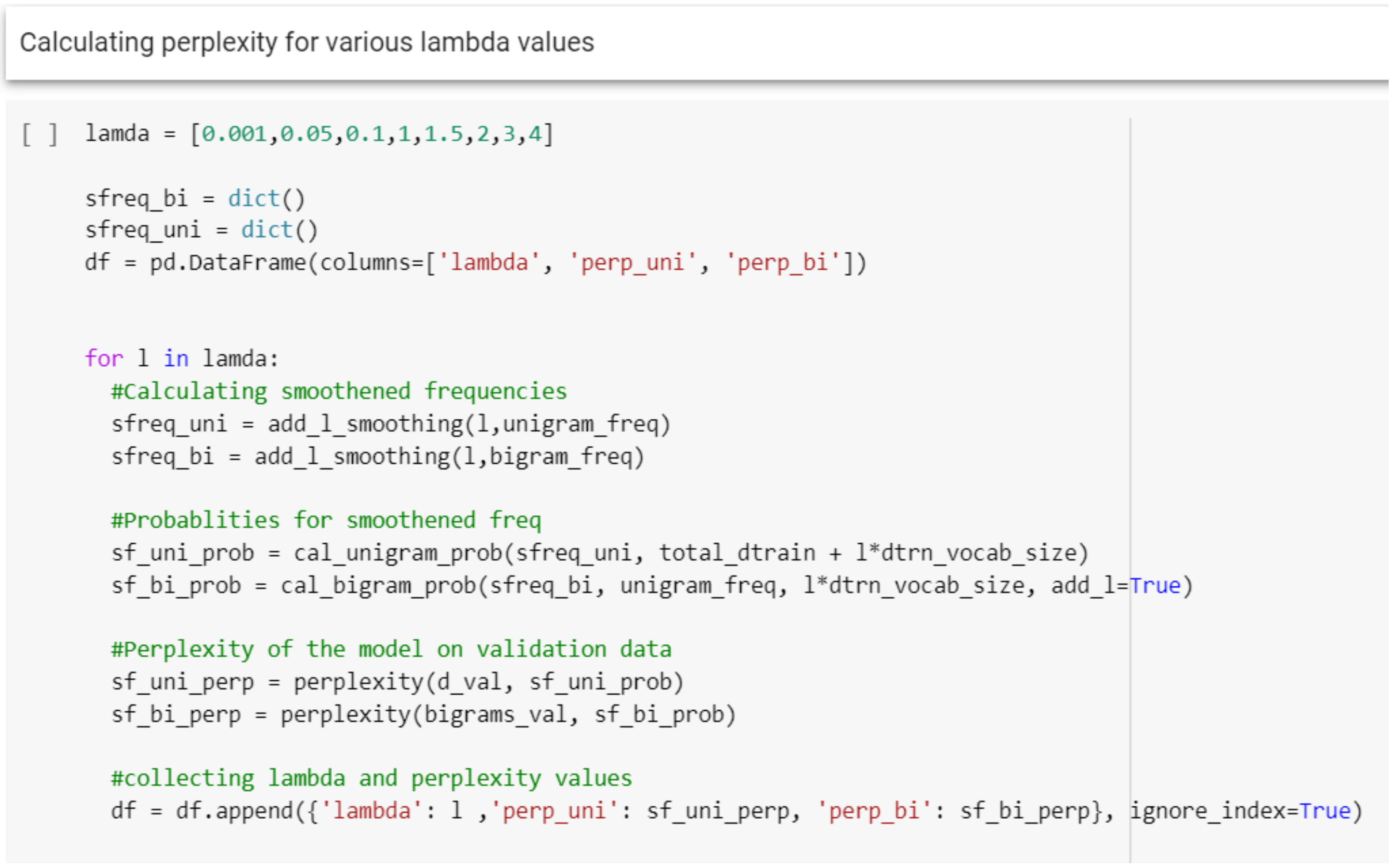
In our implementation, we have used the function add_l_smoothing() to get the smoothed frequencies when an unsmoothed frequency is passed to it.


```
#Function to provide smoothened frequencies
def add_l_smoothing(l, freq):
  s_freq = dict()
  for key in freq.keys():
      if freq.get(key) == None:
        s_freq[key] = l
      else:
        s_freq[key] = freq.get(key)+l
  return s_freq
```

We have used these different values of lambda: 0.001,0.05,0.1,1,1.5,2,3,4 to get the best lambda.
From the previously calculated unigram and bigram frequencies, we pass them to the add_l_smoothing() function to get the smoothed frequencies. These smoothed frequencies are then used to calculate the new probabilities for unigram and bigram models. In case of unigram models, the numerator would be the smoothed frequencies whereas the denominator would be the total tokens plus lambda times the vocabulary size of the training data. In case of the bigram models, the numerator would be the smoothed frequencies whereas the denominator would be the unigram frequency of the previous word plus lambda times the vocabulary size of the training data.

ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

After getting the probabilities for unigram and bigram models, we will calculate the perplexity of the models using the validation or developing data. The different values are shown in the table below.



Calculating perplexity for various lambda values

```
[ ]  lamda = [0.001,0.05,0.1,1,1.5,2,3,4]

     sfreq_bi = dict()
     sfreq_uni = dict()
     df = pd.DataFrame(columns=['lambda', 'perp_uni', 'perp_bi'])


     for l in lamda:
       #Calculating smoothened frequencies
       sfreq_uni = add_l_smoothing(l,unigram_freq)
       sfreq_bi = add_l_smoothing(l,bigram_freq)

       #Probablities for smoothened freq
       sf_uni_prob = cal_unigram_prob(sfreq_uni, total_dtrain + l*dtrn_vocab_size)
       sf_bi_prob = cal_bigram_prob(sfreq_bi, unigram_freq, l*dtrn_vocab_size, add_l=True)

       #Perplexity of the model on validation data
       sf_uni_perp = perplexity(d_val, sf_uni_prob)
       sf_bi_perp = perplexity(bigrams_val, sf_bi_prob)

       #collecting lambda and perplexity values
       df = df.append({'lambda': l ,'perp_uni': sf_uni_perp, 'perp_bi': sf_bi_perp}, ignore_index=True)
```

ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

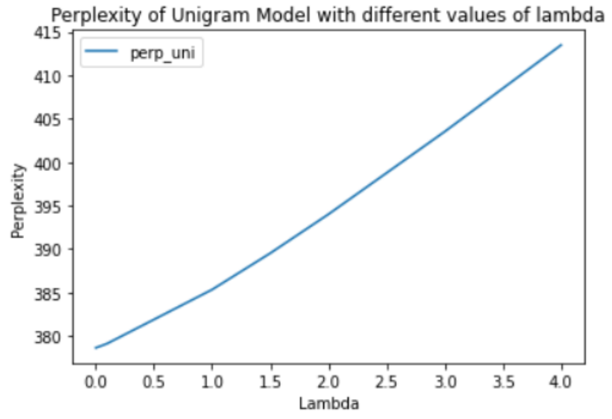
	lambda	perp_uni	perp_bi
0	0.001	378.655185	13.537326
1	0.050	378.893797	24.350821
2	0.100	379.151249	29.475224
3	1.000	385.342376	66.496547
4	1.500	389.527800	78.348777
5	2.000	394.005223	88.175619
6	3.000	403.511048	104.259137
7	4.000	413.462074	117.379624

We can see that, as the lambda values increase, the perplexity also increases for both the n-gram models. The lower the perplexity the better the model. Therefore, we will select a lambda value which gives the lowest perplexity for both our models i.e, lambda = 0.001.

Below is the graph for lambda vs perplexity for the unigram model:

```
# plotting the graph for different values of lambda
# and their respective perplexity for unigram model

df.plot(x='lambda', y='perp_uni', xlabel= 'Lambda', ylabel = 'Perplexity'
      , title = "Perplexity of Unigram Model with different values of lambda")
```

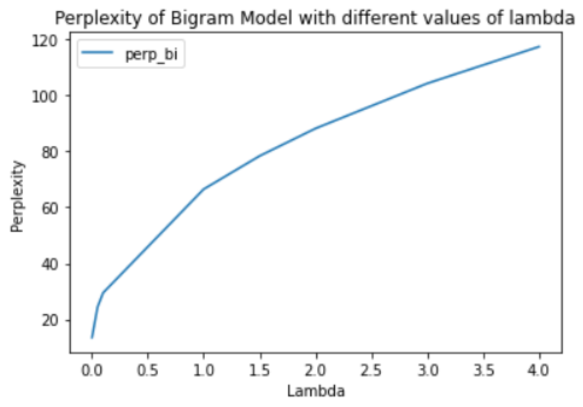
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f248dd57650>
```



<u>Below is the graph for lambda vs perplexity for the bigram model:</u>

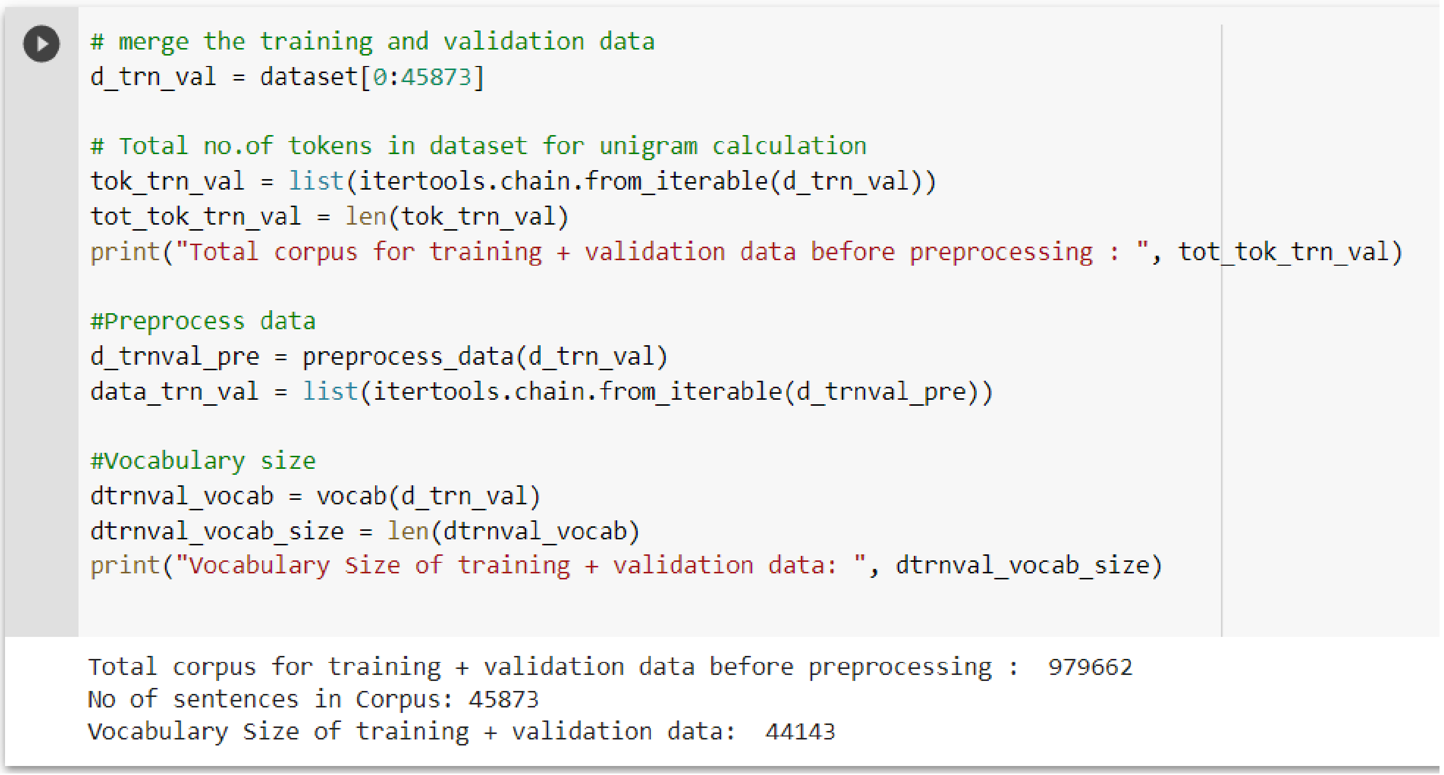
```
# plotting the graph for different values of lambda
# and their respective perplexity for bigram model

df.plot(x='lambda', y='perp_bi', xlabel= 'Lambda', ylabel = 'Perplexity'
      , title = "Perplexity of Bigram Model with different values of lambda")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f248dc7c510>
```



(c) Pick the best lambda value(s) and train again your unigram and bigram models on training data + developing data. Report new perplexity of your unigram and bigram models on the test data.

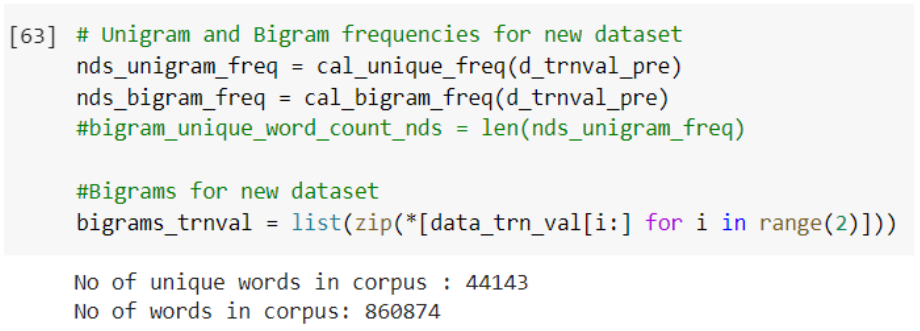
Ans. From the last question, we get the best lambda value as 0.001 for our n-gram models. Since we have to use a combined set of training and developing data to train our new models, we will first merge and preprocess them as shown below.



```
# merge the training and validation data
d_trn_val = dataset[0:45873]

# Total no.of tokens in dataset for unigram calculation
tok_trn_val = list(itertools.chain.from_iterable(d_trn_val))
tot_tok_trn_val = len(tok_trn_val)
print("Total corpus for training + validation data before preprocessing : ", tot_tok_trn_val)

#Preprocess data
d_trnval_pre = preprocess_data(d_trn_val)
data_trn_val = list(itertools.chain.from_iterable(d_trnval_pre))

#Vocabulary size
dtrnval_vocab = vocab(d_trn_val)
dtrnval_vocab_size = len(dtrnval_vocab)
print("Vocabulary Size of training + validation data: ", dtrnval_vocab_size)
```

```
Total corpus for training + validation data before preprocessing :  979662
No of sentences in Corpus: 45873
Vocabulary Size of training + validation data:  44143
```

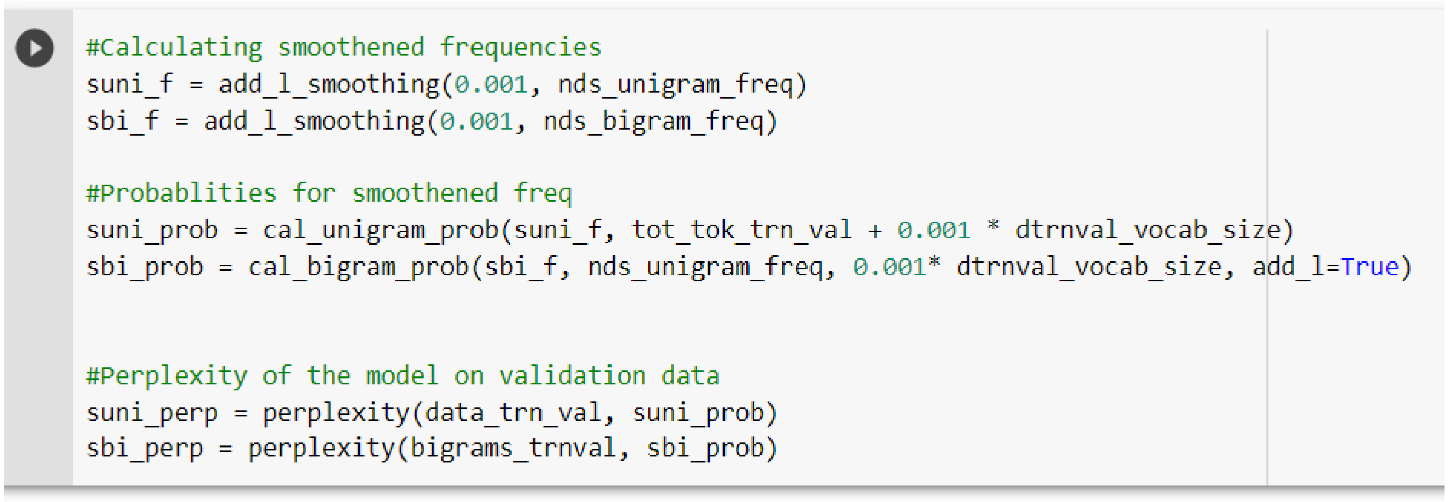
Then we will calculate the unigram and bigram frequencies of this new train dataset.



```
[63] # Unigram and Bigram frequencies for new dataset
     nds_unigram_freq = cal_unique_freq(d_trnval_pre)
     nds_bigram_freq = cal_bigram_freq(d_trnval_pre)
     #bigram_unique_word_count_nds = len(nds_unigram_freq)

     #Bigrams for new dataset
     bigrams_trnval = list(zip(*[data_trn_val[i:] for i in range(2)]))
```

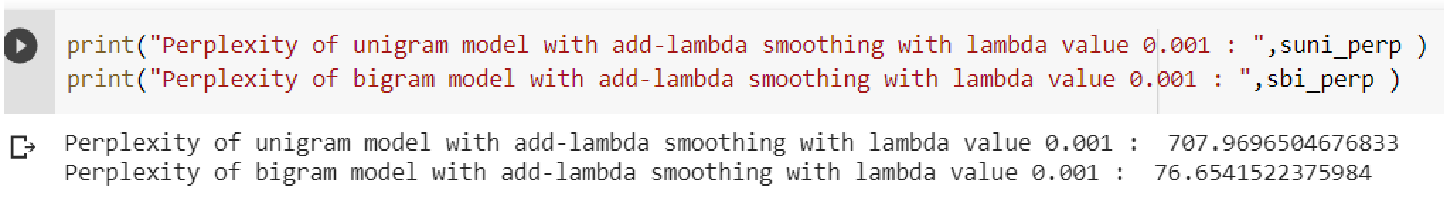
```
No of unique words in corpus : 44143
No of words in corpus: 860874
```

Now since we must apply add-lambda smoothing with the lambda value as 0.001, we will first compute the smoothed frequencies and then their probabilities.



```
#Calculating smoothened frequencies
suni_f = add_l_smoothing(0.001, nds_unigram_freq)
sbi_f = add_l_smoothing(0.001, nds_bigram_freq)

#Probablities for smoothened freq
suni_prob = cal_unigram_prob(suni_f, tot_tok_trn_val + 0.001 * dtrnval_vocab_size)
sbi_prob = cal_bigram_prob(sbi_f, nds_unigram_freq, 0.001* dtrnval_vocab_size, add_l=True)


#Perplexity of the model on validation data
suni_perp = perplexity(data_trn_val, suni_prob)
sbi_perp = perplexity(bigrams_trnval, sbi_prob)
```

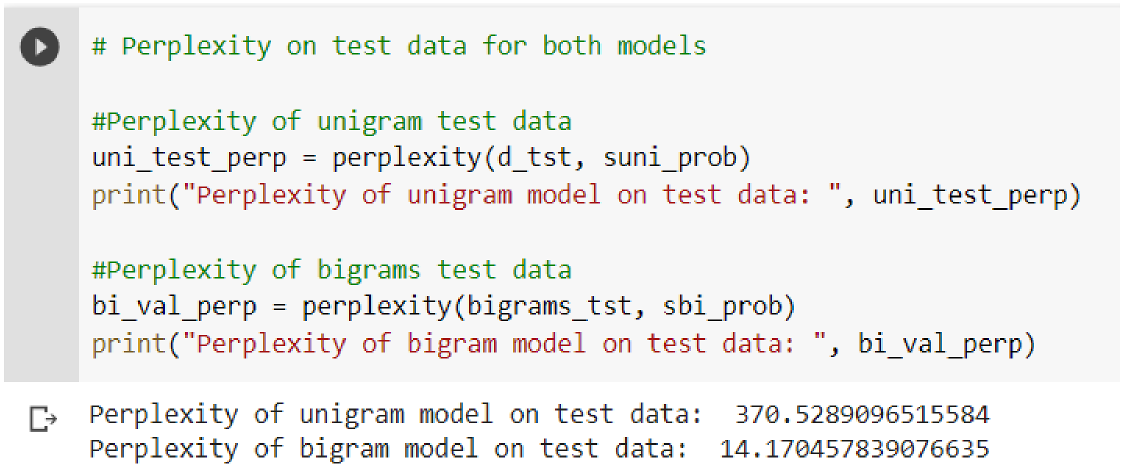
The perplexity of unigram and bigram models on the training data is as shown below:



```
print("Perplexity of unigram model with add-lambda smoothing with lambda value 0.001 : ",suni_perp )
print("Perplexity of bigram model with add-lambda smoothing with lambda value 0.001 : ",sbi_perp )
```

```
Perplexity of unigram model with add-lambda smoothing with lambda value 0.001 :  707.9696504676833
Perplexity of bigram model with add-lambda smoothing with lambda value 0.001 :  76.6541522375984
```

We will now calculate the perplexity on the test data for both the n-gram models.



```
# Perplexity on test data for both models

#Perplexity of unigram test data
uni_test_perp = perplexity(d_tst, suni_prob)
print("Perplexity of unigram model on test data: ", uni_test_perp)

#Perplexity of bigrams test data
bi_val_perp = perplexity(bigrams_tst, sbi_prob)
print("Perplexity of bigram model on test data: ", bi_val_perp)
```

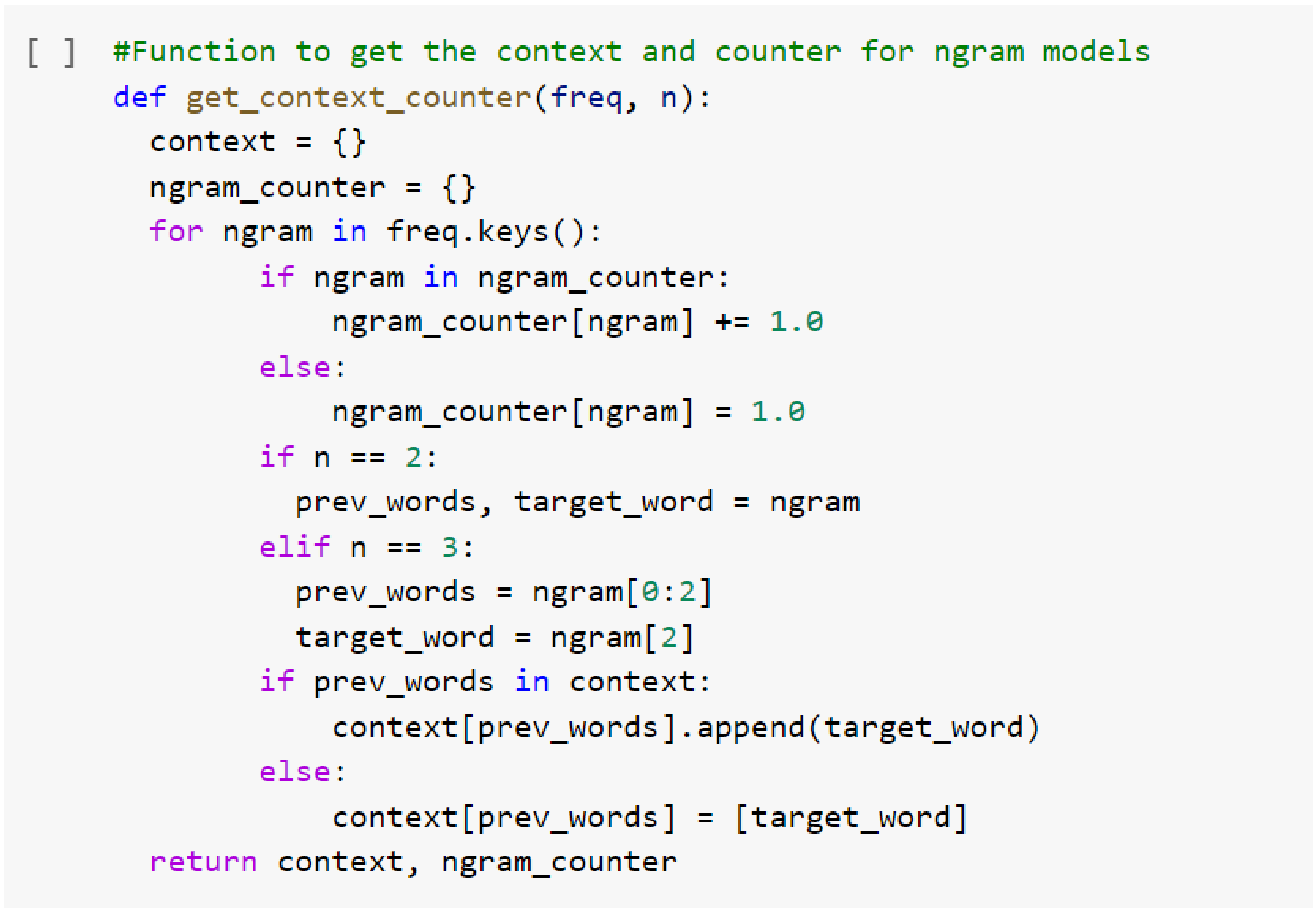
```
Perplexity of unigram model on test data:  370.5289096515584
Perplexity of bigram model on test data:  14.170457839076635
```

We see that for the test data, the perplexity is lower than before which implies that the model can generalize better with add-lambda smoothing.

ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

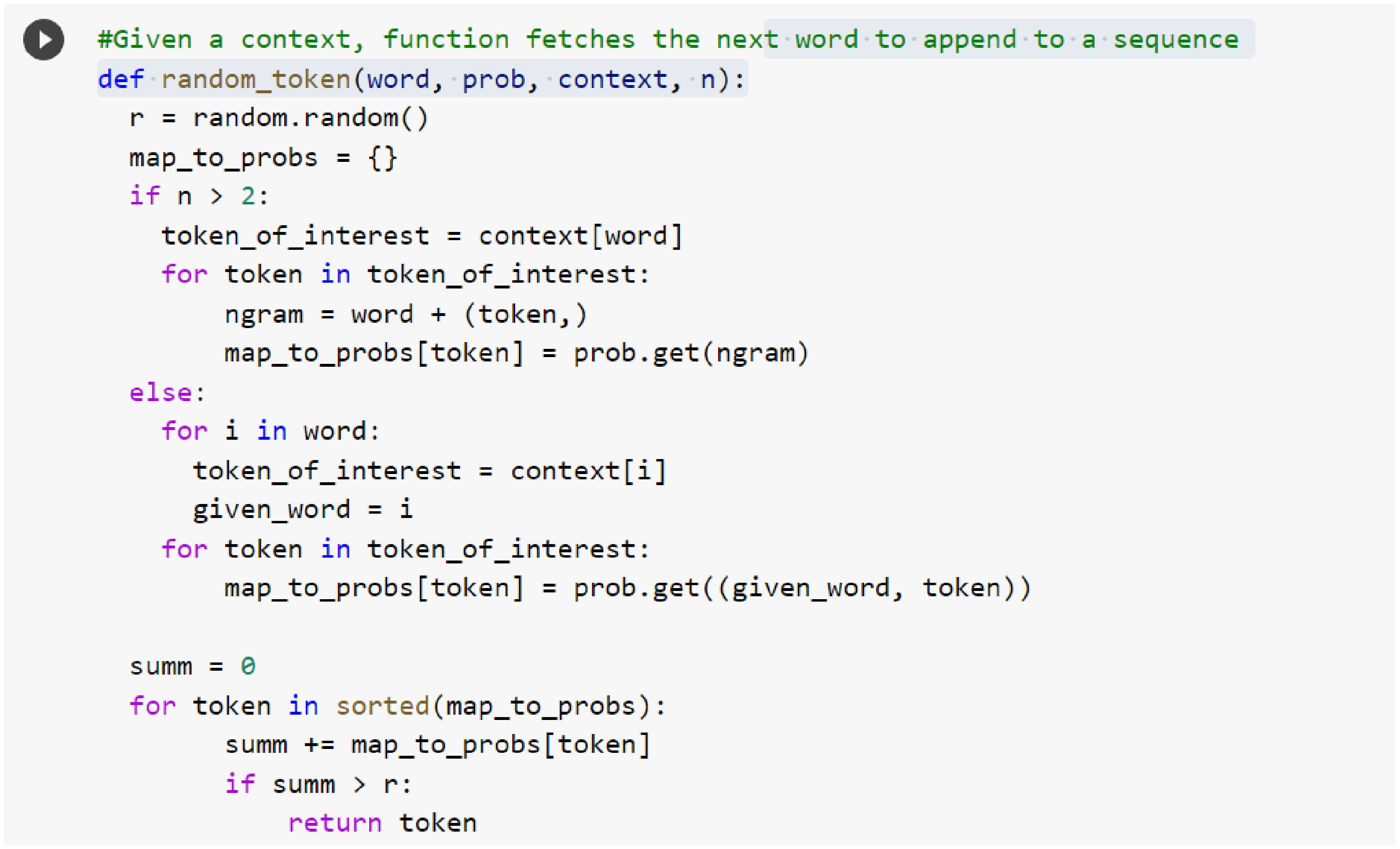
(d) Generate random sentences based on the unigram and bigram language models from part(c). Report 5 sentences per model by sampling words from each model continuously until meeting the stop symbol </s>.

Ans. We have created three functions, the first being to understand the context of words and counter for the ngram models. As the occurrence of a word increases, the frequency is added up and saved in the 'ngram_counter'. For n=2,3 the previous word and previous to previous words have been accommodated respectively. To comprehend the context, the previous word and target word are appended and saved in the dictionary 'context'. We pass the training data through the function to get the context and counter for ngram models. In the initialization we must specify, what is our n value for n-grams to create context and ngram_counter dictionaries.



```
[ ]  #Function to get the context and counter for ngram models
     def get_context_counter(freq, n):
       context = {}
       ngram_counter = {}
       for ngram in freq.keys():
             if ngram in ngram_counter:
                 ngram_counter[ngram] += 1.0
             else:
                 ngram_counter[ngram] = 1.0
             if n == 2:
               prev_words, target_word = ngram
             elif n == 3:
               prev_words = ngram[0:2]
               target_word = ngram[2]
             if prev_words in context:
                 context[prev_words].append(target_word)
             else:
                 context[prev_words] = [target_word]
       return context, ngram_counter
```

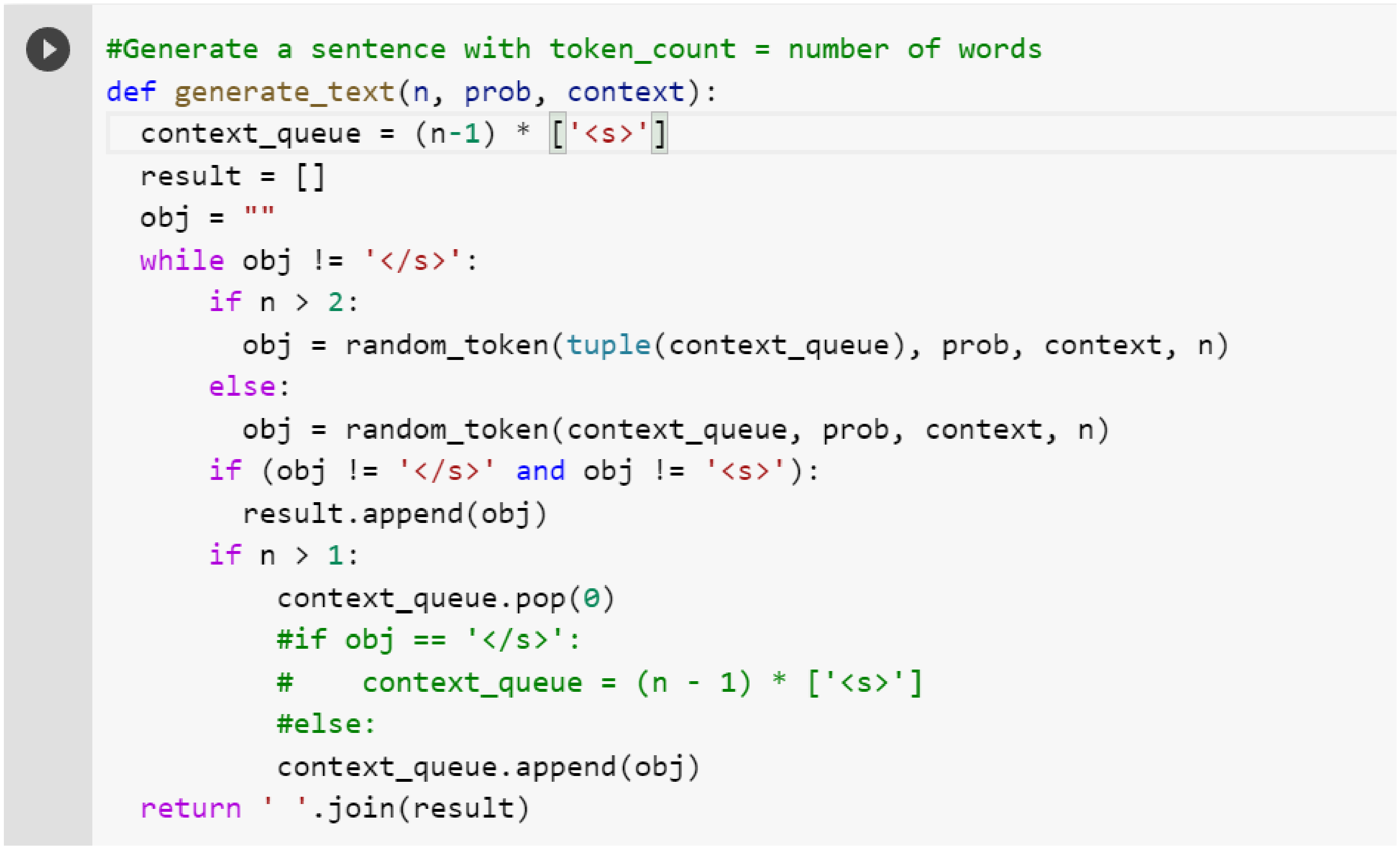
We have then defined a 'random_token' function to be able to randomly choose the next appropriate word based on the context of the training data. In the text generation function, given a context we "semi-randomly" select the next word to append in a sequence. After that we can generate our first random_token by using our "semi-random" approach.



```
#Given a context, function fetches the next word to append to a sequence
def random_token(word, prob, context, n):
  r = random.random()
  map_to_probs = {}
  if n > 2:
    token_of_interest = context[word]
    for token in token_of_interest:
        ngram = word + (token,)
        map_to_probs[token] = prob.get(ngram)
  else:
    for i in word:
      token_of_interest = context[i]
      given_word = i
    for token in token_of_interest:
        map_to_probs[token] = prob.get((given_word, token))

  summ = 0
  for token in sorted(map_to_probs):
        summ += map_to_probs[token]
        if summ > r:
            return token
```

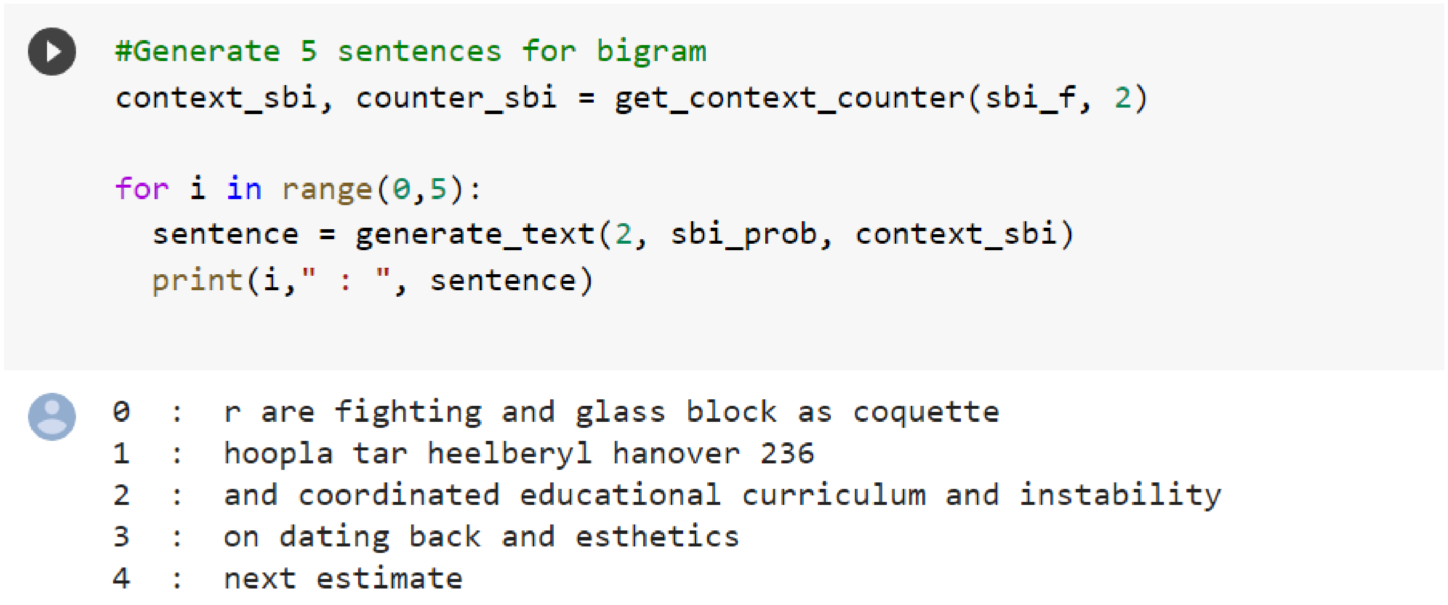
ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

Next, we have created a 'generate_text' function. We needed to reinitialize the contextual queue every time our model encounters </s>. Depending on the ngram model, we append the beginning of the sentence by (n-1) <s>. Using the random_tokens defined above, we try to generate the next possible words (tuples). The randomly generated word (or words, depending on the value of n) is appended to the result.



```python
#Generate a sentence with token_count = number of words
def generate_text(n, prob, context):
  context_queue = (n-1) * ['<s>']
  result = []
  obj = ""
  while obj != '</s>':
      if n > 2:
        obj = random_token(tuple(context_queue), prob, context, n)
      else:
        obj = random_token(context_queue, prob, context, n)
      if (obj != '</s>' and obj != '<s>'):
        result.append(obj)
      if n > 1:
          context_queue.pop(0)
          #if obj == '</s>':
          #    context_queue = (n - 1) * ['<s>']
          #else:
          context_queue.append(obj)
  return ' '.join(result)
```

Below are the Generate 5 sentences for bigram

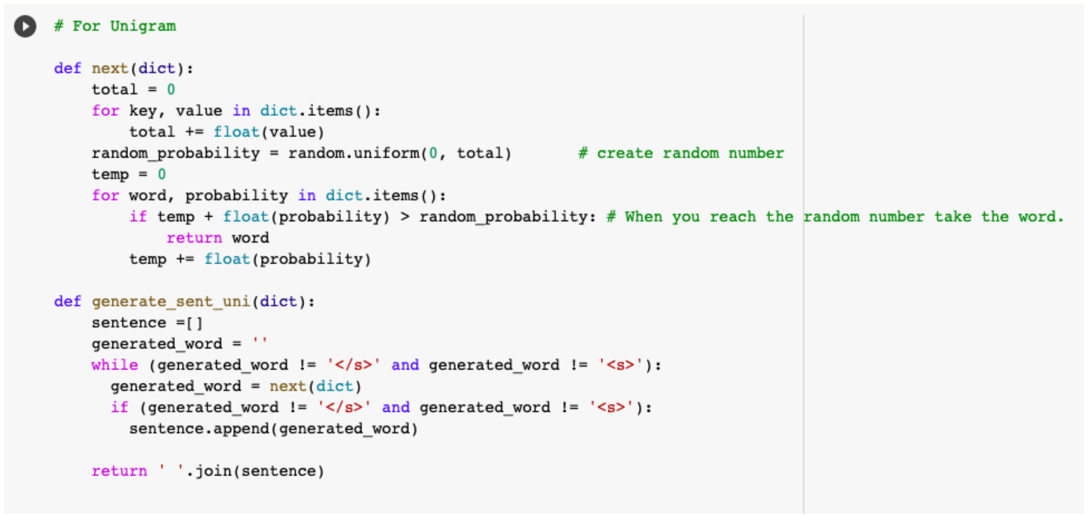


```
#Generate 5 sentences for bigram
context_sbi, counter_sbi = get_context_counter(sbi_f, 2)

for i in range(0,5):
  sentence = generate_text(2, sbi_prob, context_sbi)
  print(i," : ", sentence)
```

```
0  :  r are fighting and glass block as coquette
1  :  hoopla tar heelberyl hanover 236
2  :  and coordinated educational curriculum and instability
3  :  on dating back and esthetics
4  :  next estimate
```

For the Unigram Model,

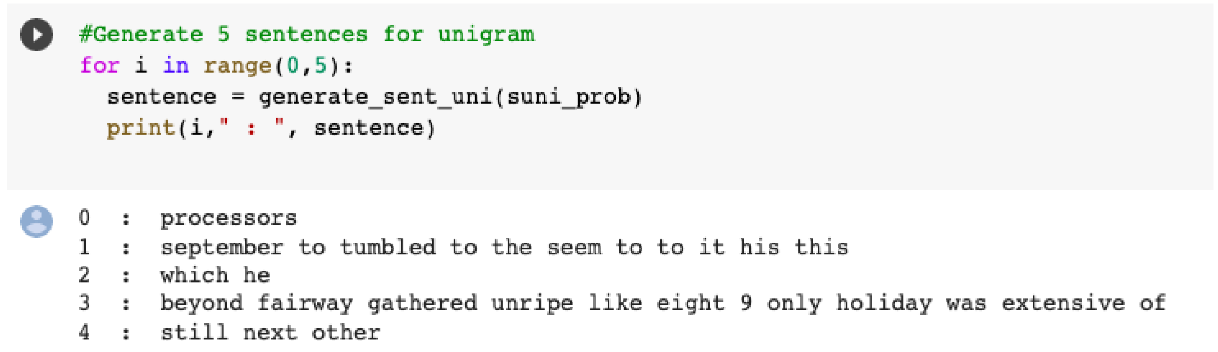
For the unigram model, using random generation of text and probability of the words based on training data, ‘next’ and ‘generate_sent_uni’ function are defined as:



```
# For Unigram

def next(dict):
    total = 0
    for key, value in dict.items():
        total += float(value)
    random_probability = random.uniform(0, total)       # create random number
    temp = 0
    for word, probability in dict.items():
        if temp + float(probability) > random_probability: # When you reach the random number take the word.
            return word
        temp += float(probability)

def generate_sent_uni(dict):
    sentence =[]
    generated_word = ''
    while (generated_word != '</s>' and generated_word != '<s>'):
      generated_word = next(dict)
      if (generated_word != '</s>' and generated_word != '<s>'):
        sentence.append(generated_word)

    return ' '.join(sentence)
```

ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

<u>Below are the Generate 5 sentences for unigram</u>



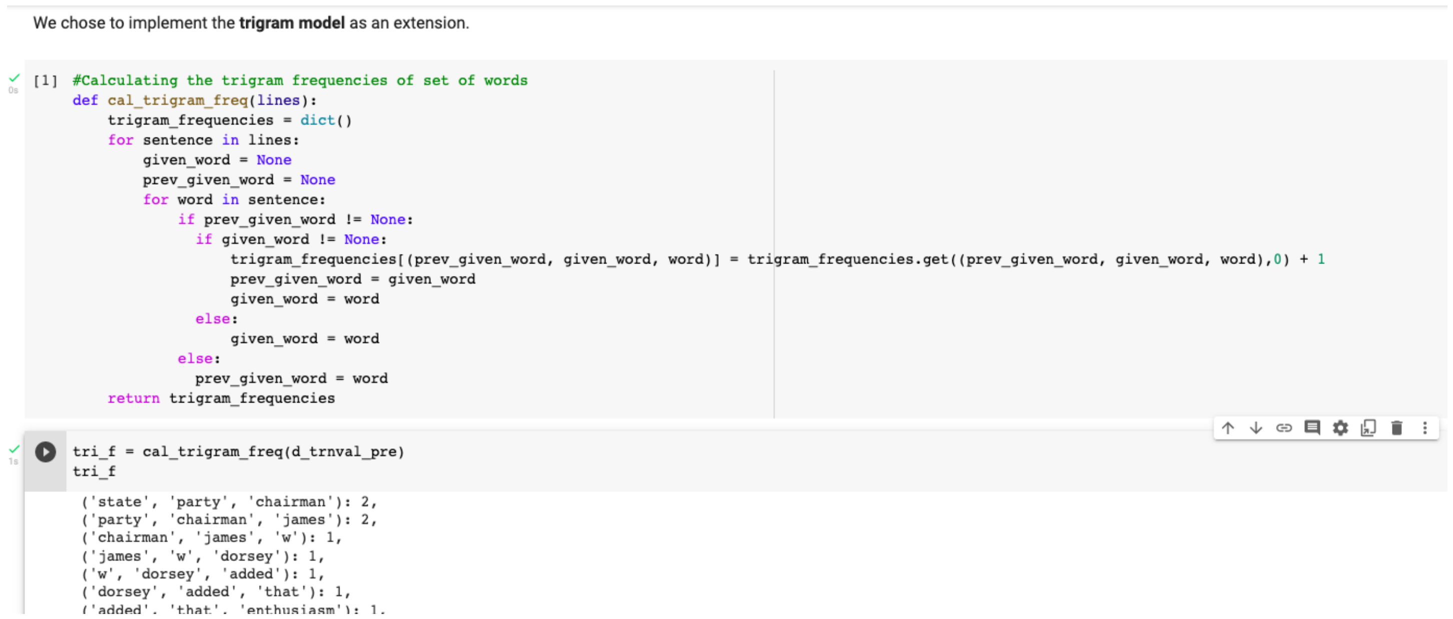
```
#Generate 5 sentences for unigram
for i in range(0,5):
  sentence = generate_sent_uni(suni_prob)
  print(i," : ", sentence)
```

```
0  :  processors
1  :  september to tumbled to the seem to to it his this
2  :  which he
3  :  beyond fairway gathered unripe like eight 9 only holiday was extensive of
4  :  still next other
```

(e) Choose at least one additional extension to implement. The available options are tri-gram, Good-Turing smoothing, interpolation method, and creative handling of unknown words. Verify quantitative improvement by measuring 1) the perplexity on test data; and qualitative improvement by retrying 2) the random sentence generation in part (d).

Ans. We have used the trigram model as an extension. For the trigram model, we appended the <s><s> and </s> </s> at the beginning and end of each sentence while preprocessing the brown corpus. We will be using the same vocabulary set we already built in the beginning of the project for the unigram and bigram models.

For the trigram model we calculate the trigram frequencies as the total occurrences of the previous word to the previous word(w_{n-2}), previous word(w_{n-1}) along with the current word(w_n). This is implemented in the function cal_trigram_freq. We pass the training and validation dataset here as a parameter to get the trigram frequencies of the set of words. The output is a dictionary with each bigram and their respective frequencies.



We chose to implement the **trigram model** as an extension.

```
[1] #Calculating the trigram frequencies of set of words
    def cal_trigram_freq(lines):
        trigram_frequencies = dict()
        for sentence in lines:
            given_word = None
            prev_given_word = None
            for word in sentence:
                if prev_given_word != None:
                  if given_word != None:
                      trigram_frequencies[(prev_given_word, given_word, word)] = trigram_frequencies.get((prev_given_word, given_word, word),0) + 1
                      prev_given_word = given_word
                      given_word = word
                  else:
                      given_word = word
                else:
                  prev_given_word = word
        return trigram_frequencies
```

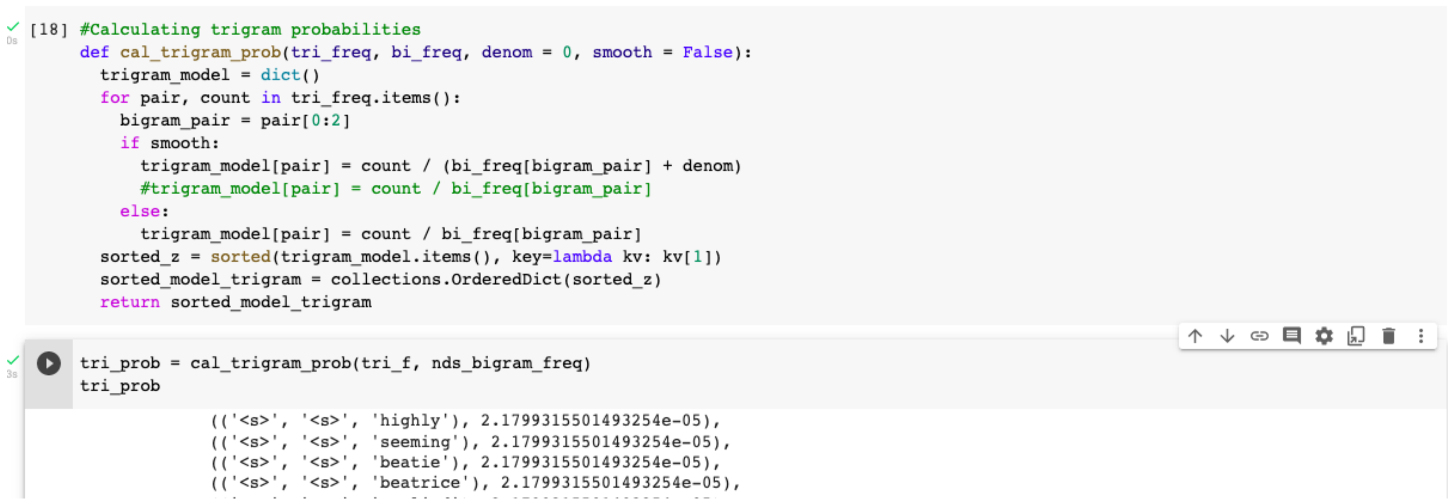
```
tri_f = cal_trigram_freq(d_trnval_pre)
tri_f
```

```
 ('state', 'party', 'chairman'): 2,
 ('party', 'chairman', 'james'): 2,
 ('chairman', 'james', 'w'): 1,
 ('james', 'w', 'dorsey'): 1,
 ('w', 'dorsey', 'added'): 1,
 ('dorsey', 'added', 'that'): 1,
 ('added', 'that', 'enthusiasm'): 1,
```

Then we calculate the trigram probabilities using the conditional probability formula below and the same is implemented in the function cal_trigram_prob() which takes as input the bigram and trigram frequencies calculated in the previous steps along with additional parameters 'denom' and smoothing is set to false(i.e. smoothing isn't considered).

$$P(w_n \mid w_{n-2}w_{n-1}) = \frac{count(w_{n-2}, w_{n-1}, w_n)}{count(w_{n-2}, w_{n-1})}$$

We find the probability of a current word given the previous word and previous to previous word using conditional probability. The numerator is the count of occurrences of the previous word to previous word and the previous word along with the current word in the dataset and denominator is the total count of occurrences of the previous word alone.


```
[18] #Calculating trigram probabilities
     def cal_trigram_prob(tri_freq, bi_freq, denom = 0, smooth = False):
       trigram_model = dict()
       for pair, count in tri_freq.items():
         bigram_pair = pair[0:2]
         if smooth:
           trigram_model[pair] = count / (bi_freq[bigram_pair] + denom)
           #trigram_model[pair] = count / bi_freq[bigram_pair]
         else:
           trigram_model[pair] = count / bi_freq[bigram_pair]
       sorted_z = sorted(trigram_model.items(), key=lambda kv: kv[1])
       sorted_model_trigram = collections.OrderedDict(sorted_z)
       return sorted_model_trigram
```

```
tri_prob = cal_trigram_prob(tri_f, nds_bigram_freq)
tri_prob
```

```
             (('<s>', '<s>', 'highly'), 2.1799315501493254e-05),
             (('<s>', '<s>', 'seeming'), 2.1799315501493254e-05),
             (('<s>', '<s>', 'beatie'), 2.1799315501493254e-05),
             (('<s>', '<s>', 'beatrice'), 2.1799315501493254e-05),
```

PERPLEXITY

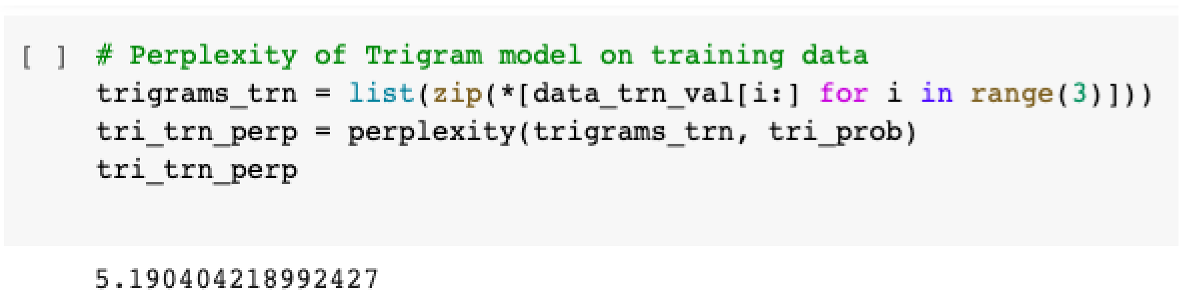
In order to evaluate how good our model is we find its perplexity as used as a metric above.

To calculate the perplexity of the models, we have used the function perplexity() which implements the below formula:

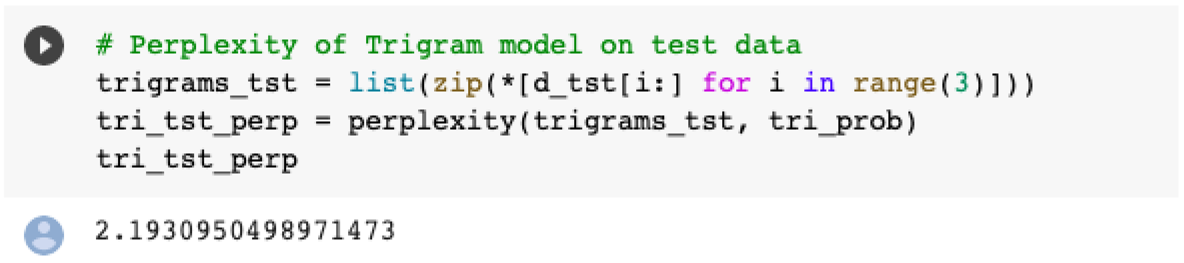
$$PP(w_1, \ldots, w_n) = 2^{-\frac{1}{n}\log_2 q(w_1,\ldots,w_n)}$$

We used the perplexity defined function to calculate the perplexity of the trigram model on training and test data set.

ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

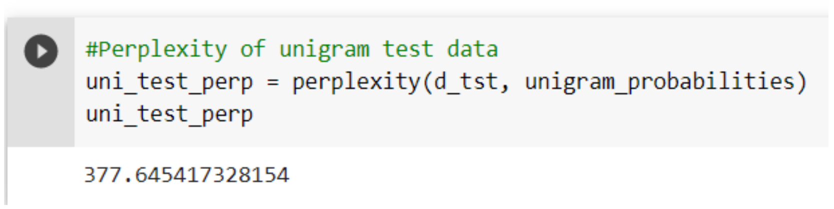


```
[ ] # Perplexity of Trigram model on training data
    trigrams_trn = list(zip(*[data_trn_val[i:] for i in range(3)]))
    tri_trn_perp = perplexity(trigrams_trn, tri_prob)
    tri_trn_perp

    5.190404218992427
```

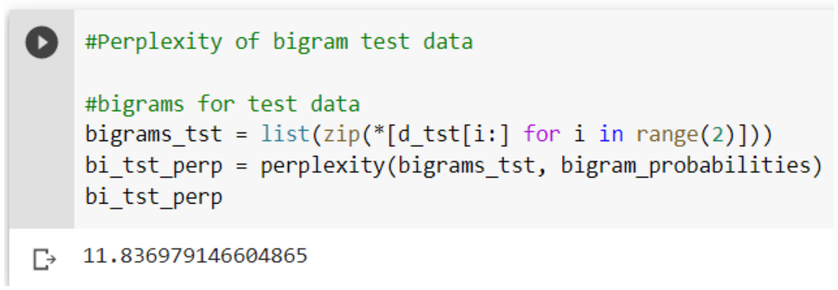


```
# Perplexity of Trigram model on test data
trigrams_tst = list(zip(*[d_tst[i:] for i in range(3)]))
tri_tst_perp = perplexity(trigrams_tst, tri_prob)
tri_tst_perp

2.1930950498971473
```

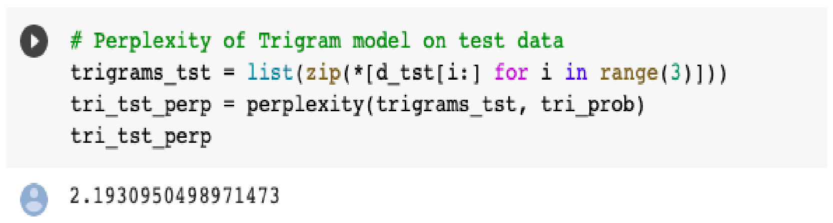
If we compare this perplexity to that of unigram and bigram models on test data,



```
#Perplexity of unigram test data
uni_test_perp = perplexity(d_tst, unigram_probabilities)
uni_test_perp

377.645417328154
```



```
#Perplexity of bigram test data

#bigrams for test data
bigrams_tst = list(zip(*[d_tst[i:] for i in range(2)]))
bi_tst_perp = perplexity(bigrams_tst, bigram_probabilities)
bi_tst_perp

11.836979146604865
```



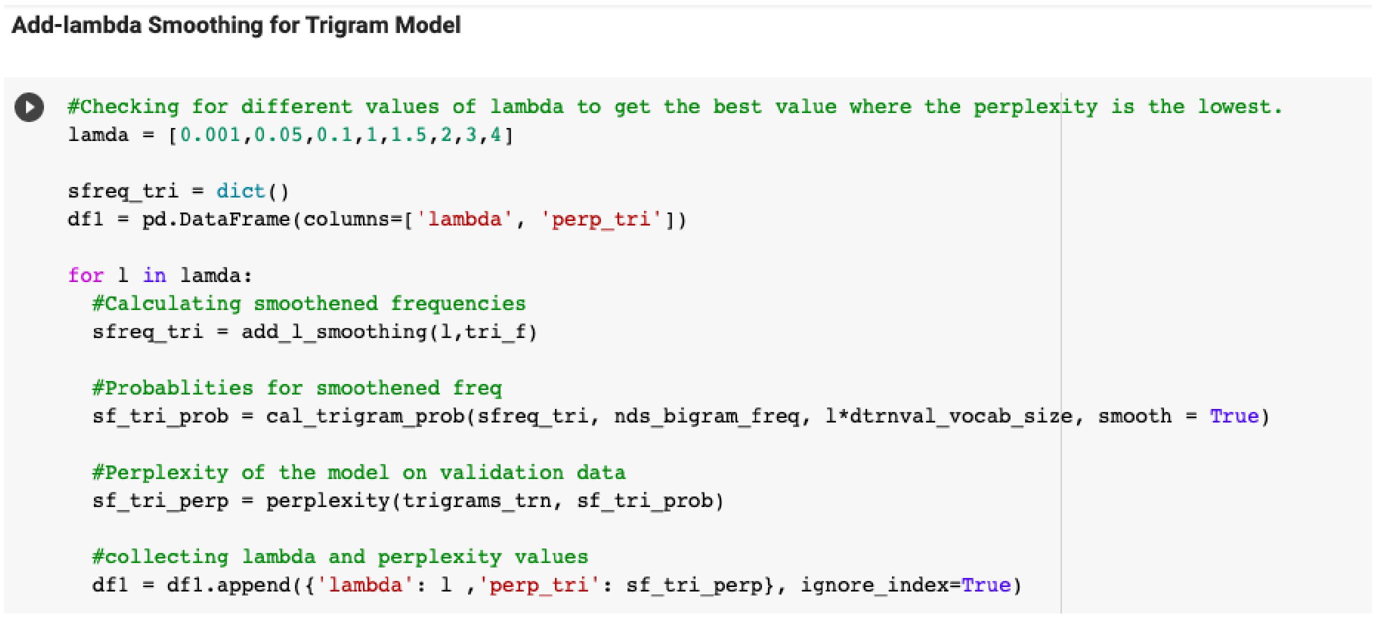
```
# Perplexity of Trigram model on test data
trigrams_tst = list(zip(*[d_tst[i:] for i in range(3)]))
tri_tst_perp = perplexity(trigrams_tst, tri_prob)
tri_tst_perp

2.1930950498971473
```

we find that the **perplexity of the upper n-gram model is lowest as they use the context, therefore the next word would not be that surprising.**

ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

Lambda Smoothing to Trigram

In order to overcome the problem with MLE that it assigns zero probabilities to unseen words, making the complete probability zero, we use the add lambda smoothing.Smoothing takes some probability mass from the events seen in the training dataset and assigns it to the unseen events. Add-lambda smoothing or Laplace smoothing adds the lambda values to the count of all n-grams in the training set before normalizing into probabilities.



Add-lambda Smoothing for Trigram Model

```
#Checking for different values of lambda to get the best value where the perplexity is the lowest.
lamda = [0.001,0.05,0.1,1,1.5,2,3,4]

sfreq_tri = dict()
df1 = pd.DataFrame(columns=['lambda', 'perp_tri'])

for l in lamda:
  #Calculating smoothened frequencies
  sfreq_tri = add_l_smoothing(l,tri_f)

  #Probablities for smoothened freq
  sf_tri_prob = cal_trigram_prob(sfreq_tri, nds_bigram_freq, l*dtrnval_vocab_size, smooth = True)

  #Perplexity of the model on validation data
  sf_tri_perp = perplexity(trigrams_trn, sf_tri_prob)

  #collecting lambda and perplexity values
  df1 = df1.append({'lambda': l ,'perp_tri': sf_tri_perp}, ignore_index=True)
```

In our implementation, we have used the function add_l_smoothing() to get the smoothed frequencies when an unsmoothed frequency is passed to it. We have used these different values of lambda: 0.001,0.05,0.1,1,1.5,2,3,4 to get the best lambda.

From the previously calculated trigram frequencies, we pass them to the add_l_smoothing() function to get the smoothed frequencies. These smoothed frequencies are then used to calculate the new probabilities for the trigram model.

After getting the probabilities for the model, we will calculate the perplexity of the models using the validation or developing data. The different values are shown in the table below.

ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

```
[ ] df1
```

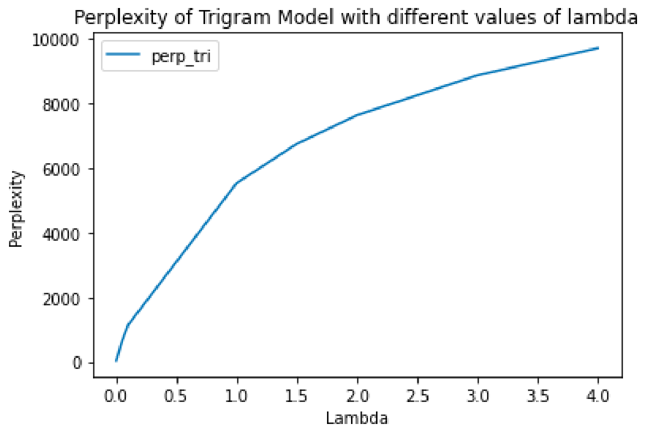
	lambda	perp_tri
0	0.001	36.057372
1	0.050	662.109225
2	0.100	1145.291467
3	1.000	5520.871187
4	1.500	6735.445538
5	2.000	7619.739674
6	3.000	8851.465872
7	4.000	9689.876795

We can see that, as the lambda values increase, the perplexity also increases. The lower the perplexity the better the model. Therefore, we will select a lambda value which gives the lowest perplexity for both our models i.e, lambda = 0.001.

<u>The graph of for lambda vs perplexity for the trigram model</u>:

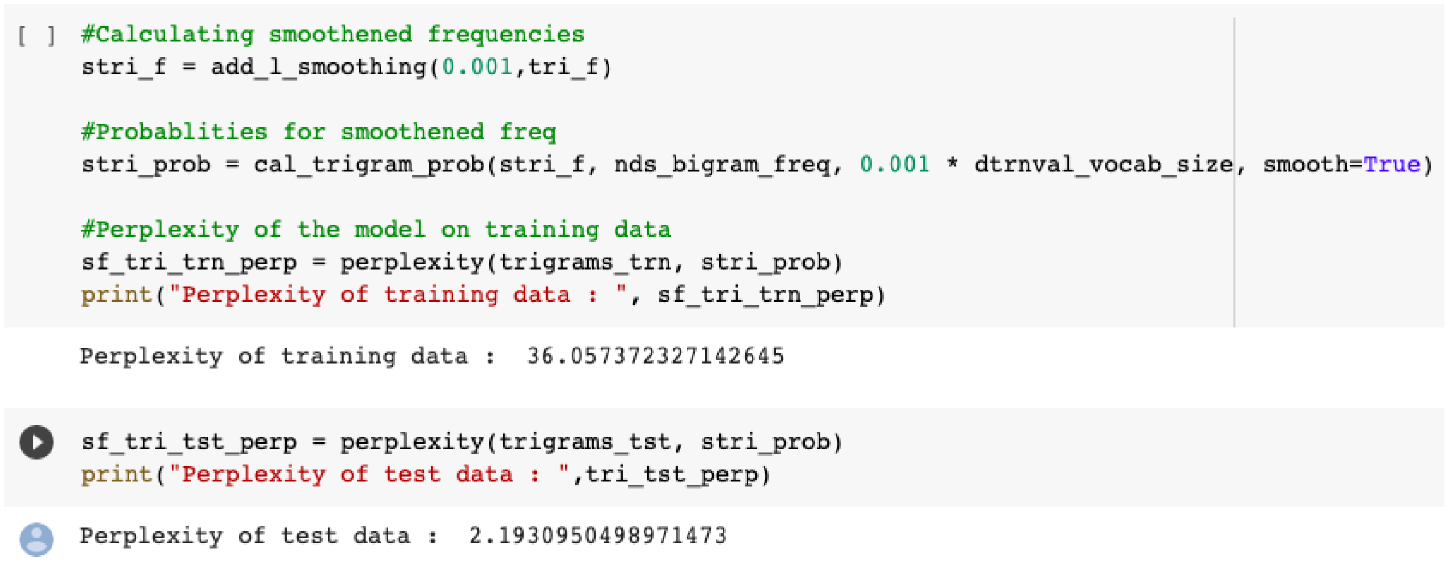
```
# plotting the graph for different values of lambda
# and their respective perplexity for triigram model

df1.plot(x='lambda', y='perp_tri', xlabel= 'Lambda', ylabel = 'Perplexity'
          , title = "Perplexity of Trigram Model with different values of lambda")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2473747c10>
```



ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

Taking lambda as 0.001 we train the model again and get the perplexity as:

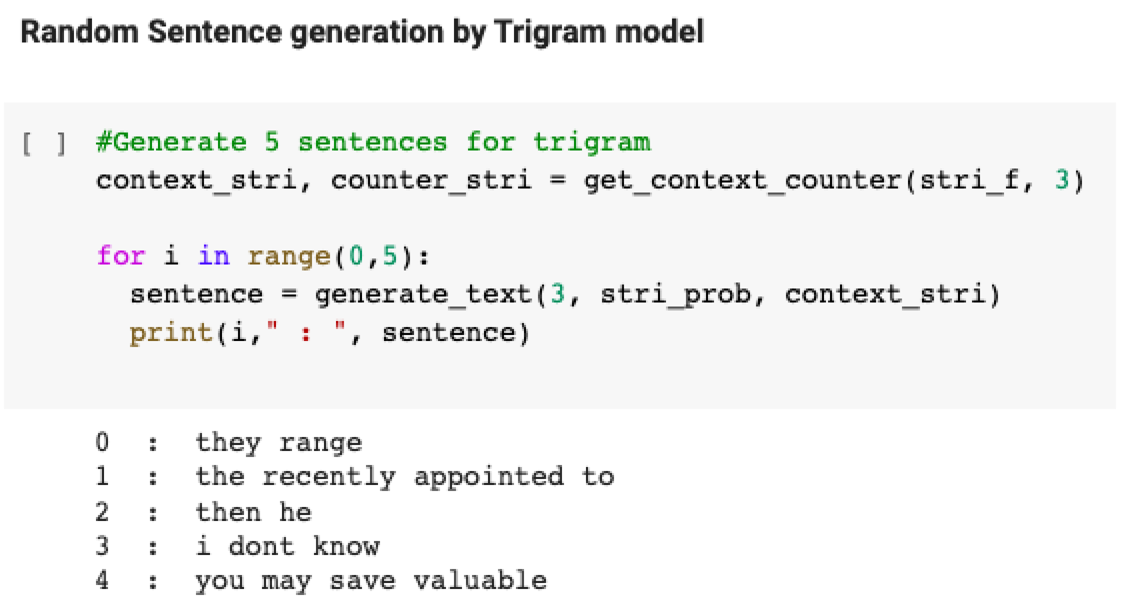


```
[ ] #Calculating smoothened frequencies
    stri_f = add_l_smoothing(0.001,tri_f)

    #Probablities for smoothened freq
    stri_prob = cal_trigram_prob(stri_f, nds_bigram_freq, 0.001 * dtrnval_vocab_size, smooth=True)

    #Perplexity of the model on training data
    sf_tri_trn_perp = perplexity(trigrams_trn, stri_prob)
    print("Perplexity of training data : ", sf_tri_trn_perp)

    Perplexity of training data :  36.057372327142645
```

```
    sf_tri_tst_perp = perplexity(trigrams_tst, stri_prob)
    print("Perplexity of test data : ",tri_tst_perp)

    Perplexity of test data :  2.1930950498971473
```

Generation of Random Sentences:

We use the get_context_counter and generate_text function as defined above for the trigram model and generate 5 random sentences.



Random Sentence generation by Trigram model

```
[ ] #Generate 5 sentences for trigram
    context_stri, counter_stri = get_context_counter(stri_f, 3)

    for i in range(0,5):
      sentence = generate_text(3, stri_prob, context_stri)
      print(i," : ", sentence)
```

```
0  :  they range
1  :  the recently appointed to
2  :  then he
3  :  i dont know
4  :  you may save valuable
```

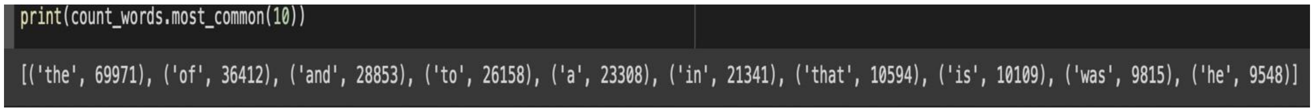
ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)

Problem 2: Application : Spell Checking

Step 1: We downloaded the "brown" corpus from the NLTK module, which is the dataset for this problem.

Step 2: Data Preprocessing: The words from Brown are all converted into lowercase. Following that, a regular expression is used to remove any punctuation or symbols. Any remaining white space is then removed.

Step 3: Calculate the frequency of each distinct word in the corpus.

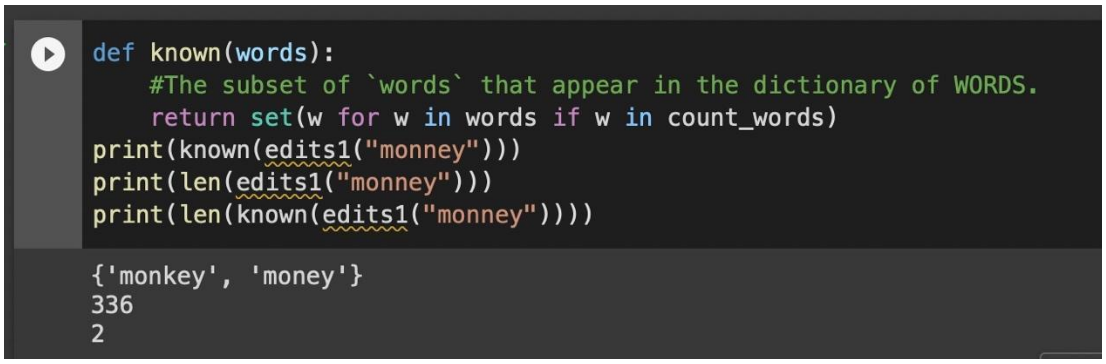


```
print(count_words.most_common(10))

[('the', 69971), ('of', 36412), ('and', 28853), ('to', 26158), ('a', 23308), ('in', 21341), ('that', 10594), ('is', 10109), ('was', 9815), ('he', 9548)]
```

Step 4: We used Peter Norvig's approach to spell check a word as our algorithm. Try every possible edit for a word, including Delete, Insert, Transpose, Replace, and Split. Every word is added to a list of words that could be used. We repeat this procedure for each word a second time to find words with a greater edit distance (for cases with two errors). A Unigram language model is used to estimate each possible word. Unigram probability is calculated for each word. The word with the highest probability is chosen from the list of possible words.

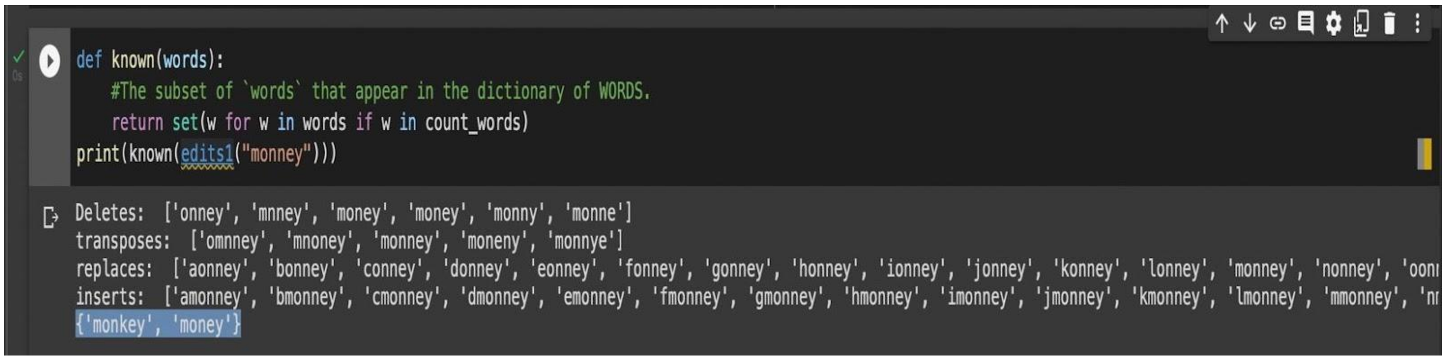
A delete (removing one letter), a transpose (swapping two letters next to each other), a replace (changing one letter for another), or an insert are all examples of simple word changes (adding a letter). edits1 returns a set of all the edited words (whether words or not) that can be made with a single simple edit.

If we limit ourselves to known words, that is, words from the dictionary, the set becomes much smaller.



```
def known(words):
    #The subset of `words` that appear in the dictionary of WORDS.
    return set(w for w in words if w in count_words)
print(known(edits1("monney")))
print(len(edits1("monney")))
print(len(known(edits1("monney"))))

{'monkey', 'money'}
336
2
```

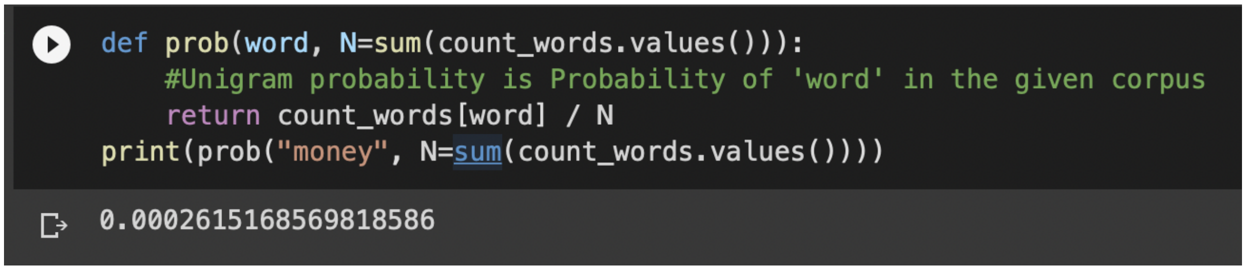
Corrections that require only two simple edits will also be considered. This generates a much larger set of possibilities, but only a few of them are commonly used words.



```
def known(words):
    #The subset of `words` that appear in the dictionary of WORDS.
    return set(w for w in words if w in count_words)
print(known(edits1("monney")))
```

```
Deletes:  ['onney', 'mnney', 'money', 'money', 'monny', 'monne']
transposes:  ['omnney', 'mnoney', 'monney', 'moneny', 'monnye']
replaces:  ['aonney', 'bonney', 'conney', 'donney', 'eonney', 'fonney', 'gonney', 'honney', 'ionney', 'jonney', 'konney', 'lonney', 'monney', 'nonney', 'oon
inserts:  ['amonney', 'bmonney', 'cmonney', 'dmonney', 'emonney', 'fmonney', 'gmonney', 'hmonney', 'imonney', 'jmonney', 'kmonney', 'lmonney', 'mmonney', 'n
{'monkey', 'money'}
```

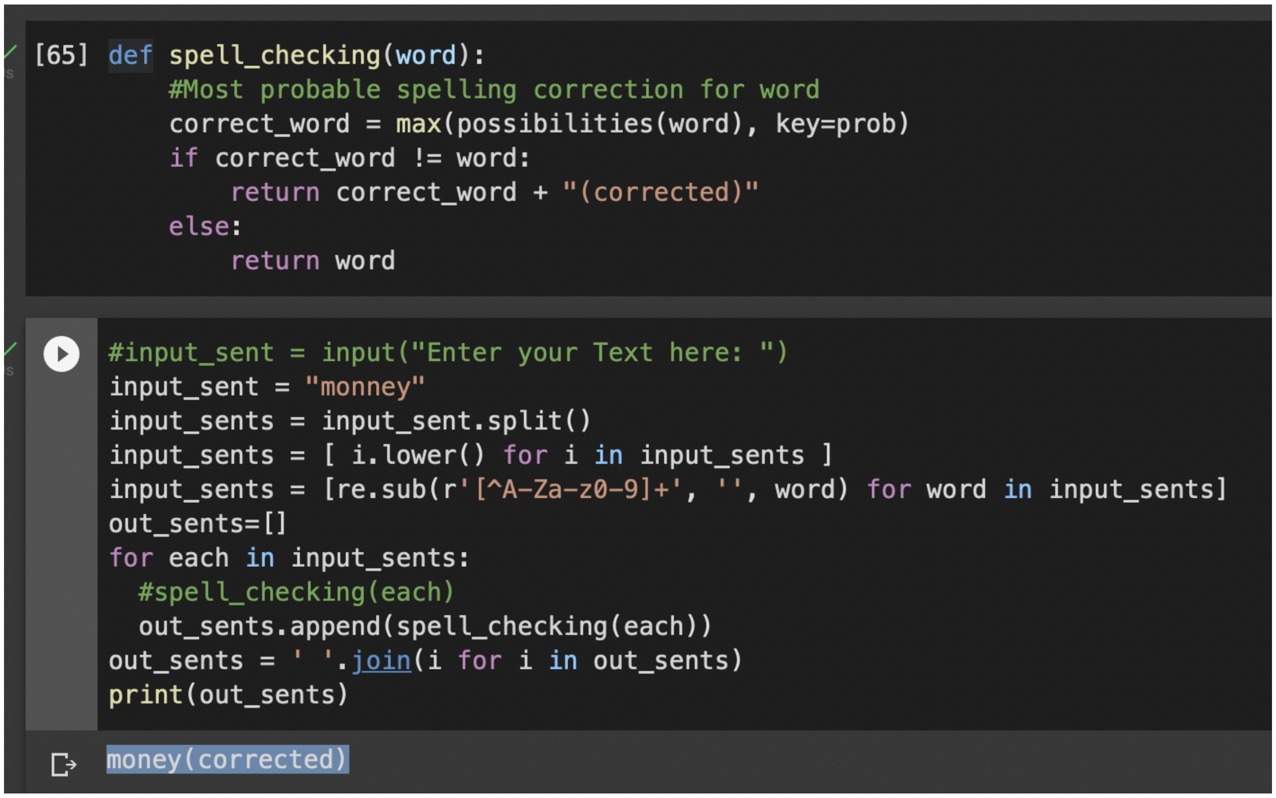
Step 5: Language Model. The Unigram probability of a "word" is found by dividing the number of times a word appears in the corpus by the total number of words in the corpus.



```
def prob(word, N=sum(count_words.values())):
    #Unigram probability is Probability of 'word' in the given corpus
    return count_words[word] / N
print(prob("money", N=sum(count_words.values())))
```

```
0.0002615168569818586
```

Step 6: Correction: To check the spelling of the word, we obtain all the possibilities of the word with one edit, two edits, or the same word if it is known (that is, in the data dictionary we have chosen). Then we choose the word that has the highest probability as our corrected word.



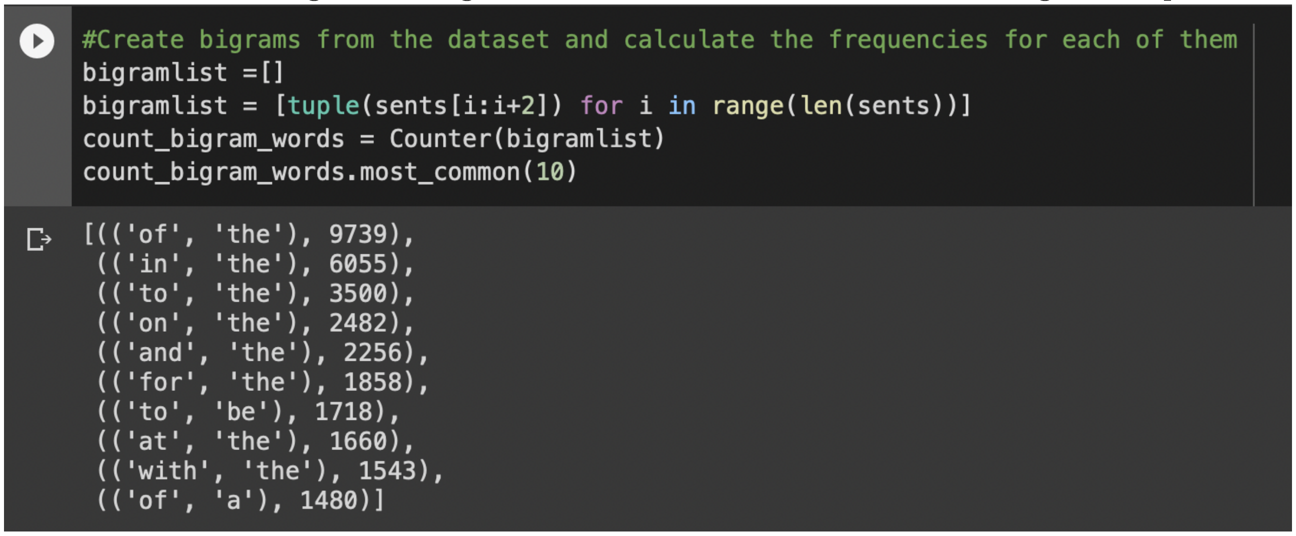
```
[65] def spell_checking(word):
         #Most probable spelling correction for word
         correct_word = max(possibilities(word), key=prob)
         if correct_word != word:
             return correct_word + "(corrected)"
         else:
             return word
```

```
#input_sent = input("Enter your Text here: ")
input_sent = "monney"
input_sents = input_sent.split()
input_sents = [ i.lower() for i in input_sents ]
input_sents = [re.sub(r'[^A-Za-z0-9]+', '', word) for word in input_sents]
out_sents=[]
for each in input_sents:
  #spell_checking(each)
  out_sents.append(spell_checking(each))
out_sents = ' '.join(i for i in out_sents)
print(out_sents)
```

```
money(corrected)
```

Step 7: Similarly, bigrams can be used to implement this.

The dataset is now used to generate bigrams, which are then used to calculate bigram frequencies.

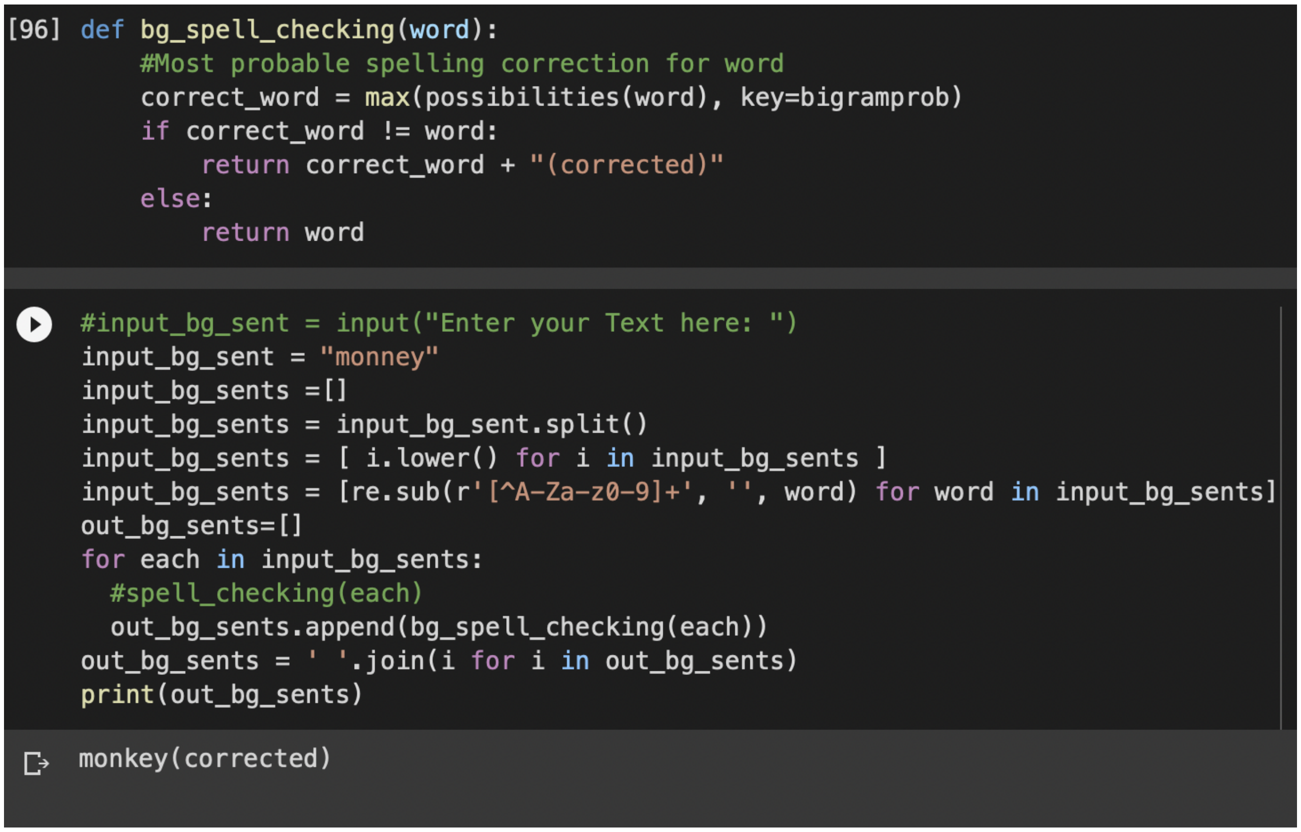


```
#Create bigrams from the dataset and calculate the frequencies for each of them
bigramlist =[]
bigramlist = [tuple(sents[i:i+2]) for i in range(len(sents))]
count_bigram_words = Counter(bigramlist)
count_bigram_words.most_common(10)
```

```
[(('of', 'the'), 9739),
 (('in', 'the'), 6055),
 (('to', 'the'), 3500),
 (('on', 'the'), 2482),
 (('and', 'the'), 2256),
 (('for', 'the'), 1858),
 (('to', 'be'), 1718),
 (('at', 'the'), 1660),
 (('with', 'the'), 1543),
 (('of', 'a'), 1480)]
```

The spell check is then performed using the bigram probabilities, considering two edits for the word, and selecting the word with the highest probability.

ANISHA VIJAYAN (UIN: 662618335)
SALONI KATARIA (UIN: 662519005)
JAHNAVI MUTHYALA (UIN:667960987)
NAINI NARAMA LNU (UIN:679008394)



```
[96] def bg_spell_checking(word):
         #Most probable spelling correction for word
         correct_word = max(possibilities(word), key=bigramprob)
         if correct_word != word:
             return correct_word + "(corrected)"
         else:
             return word
```

```
#input_bg_sent = input("Enter your Text here: ")
input_bg_sent = "monney"
input_bg_sents =[]
input_bg_sents = input_bg_sent.split()
input_bg_sents = [ i.lower() for i in input_bg_sents ]
input_bg_sents = [re.sub(r'[^A-Za-z0-9]+', '', word) for word in input_bg_sents]
out_bg_sents=[]
for each in input_bg_sents:
  #spell_checking(each)
  out_bg_sents.append(bg_spell_checking(each))
out_bg_sents = ' '.join(i for i in out_bg_sents)
print(out_bg_sents)
```

```
monkey(corrected)
```

Sources:
1. CS447: Natural Language Processing (J. Hockenmaier) Lecture 4: Smoothing http://courses.engr.illinois.edu/cs447
2. https://vitalflux.com/quick-introduction-smoothing-techniques-language-models/
3. Lecture02- Language Modeling.pdf
4. https://www.exploredatabase.com/2020/10/explain-add-1-laplace-smoothing-with-example.html
5. https://web.stanford.edu/~jurafsky/slp3/3.pdf