AY:2024-25

Class:	BE	Semester:	VII
Course Code:	CSDOL7011	Course Name:	Natural Language Processing

Name of Student:	Parth Raut
Roll No.:	40
Experiment No.:	5
Title of the Experiment:	N-Gram Model Implementation
Date of Performance:	
Date of Submission:	

Evaluation

Performance Indicator	Max. Marks	Marks Obtained
Performance	5	
Understanding	5	
Journal work and timely submission	10	
Total	20	

Performance Indicator	Exceed Expectations (EE)	Meet Expectations (ME)	Below Expectations (BE)
Performance	4-5	2-3	1
Understanding	4-5	2-3	1
Journal work and timely submission	8-10	5-8	1-4

Checked by

Name of Faculty :

Signature :

Date :

Experiment 5

Aim: Implement N-Gram model for the given text input.

Objective: To study and implement N-gram Language Model.

Theory:

A language model supports predicting the completion of a sentence.
Eg:

- Please turn off your cell _____
- Your program does not ______

Predictive text input systems can guess what you are typing and give choices on how to complete it.

N-gram Models:
Estimate probability of each word given prior context.
P(phone | Please turn off your cell)

- Number of parameters required grows exponentially with the number of words of prior context.
- An N-gram model uses only N1 words of prior context.
  - Unigram: P(phone)
  - Bigram: P(phone | cell)
  - Trigram: P(phone | your cell)

- The Markov assumption is the presumption that the future behavior of a dynamical system only depends on its recent history. In particular, in a kth-order Markov model, the next state only depends on the k most recent states, therefore an N-gram model is a (N1)-order Markov model.

N-grams: a contiguous sequence of n tokens from a given piece of text

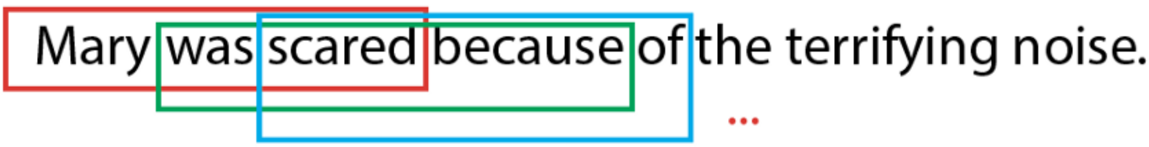

Fig. Example of Trigrams in a sentence

Code:

Necessary Imports

In [1]:
```
import nltk, re, pprint, string
from nltk import word_tokenize, sent_tokenize
string.punctuation = string.punctuation +'“'+'”'+'-'+'’'+'‘'+'—'
string.punctuation = string.punctuation.replace('.', '')
file = open('./dataset.txt', encoding = 'utf8').read()
```

Preprocess of the Data

In [2]:
```
file_nl_removed = ""
for line in file:
  line_nl_removed = line.replace("\n", " ")
  file_nl_removed += line_nl_removed
file_p = "".join([char for char in file_nl_removed if char not in string.punctuation])
```

Statistics of the Data

In [3]:
```
sents = nltk.sent_tokenize(file_p)
print("The number of sentences is", len(sents))

words = nltk.word_tokenize(file_p)
print("The number of tokens is", len(words))

average_tokens = round(len(words)/len(sents))
print("The average number of tokens per sentence is",
average_tokens)

unique_tokens = set(words)
print("The number of unique tokens are", len(unique_tokens))
```

```
The number of sentences is 981
The number of tokens is 27361
The average number of tokens per sentence is 28
The number of unique tokens are 3039
```

Building the N-Gram Model

In [4]:
```
from nltk.util import ngrams
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
```

In [5]:
```
unigram=[]
bigram=[]
trigram=[]
fourgram=[]
tokenized_text = []
```

In [6]:
```
for sentence in sents:
    sentence = sentence.lower()
    sequence = word_tokenize(sentence)
    for word in sequence:
        if (word =='.'):
            sequence.remove(word)
        else:
            unigram.append(word)
    tokenized_text.append(sequence)
    bigram.extend(list(ngrams(sequence, 2)))
    trigram.extend(list(ngrams(sequence, 3)))
    fourgram.extend(list(ngrams(sequence, 4)))

#removes ngrams containing only stopwords
def removal(x):
    y = []
    for pair in x:
        count = 0
        for word in pair:
            if word in stop_words:
                count = count or 0
            else:
                count = count or 1
        if (count==1):
            y.append(pair)
    return(y)
```

```
bigram = removal(bigram)
trigram = removal(trigram)
fourgram = removal(fourgram)
freq_bi = nltk.FreqDist(bigram)
freq_tri = nltk.FreqDist(trigram)
freq_four = nltk.FreqDist(fourgram)
print("Most common n-grams without stopword removal and without add-1 smoothing: \n")
print ("Most common bigrams: ", freq_bi.most_common(5))
print ("\nMost common trigrams: ", freq_tri.most_common(5))
print ("\nMost common fourgrams: ", freq_four.most_common(5))
```

```
Most common n-grams without stopword removal and without add-1 smoothing:

Most common bigrams:  [(('said', 'the'), 209), (('said', 'alice'), 115), (('the', 'queen'), 65), (('the', 'king'), 60), (('a', 'little'), 59)]

Most common trigrams:  [(('the', 'mock', 'turtle'), 51), (('the', 'march', 'hare'), 30), (('said', 'the', 'king'), 29), (('the', 'white', 'rabbit'), 21), (('said', 'the', 'hatter'), 21)]

Most common fourgrams:  [(('said', 'the', 'mock', 'turtle'), 19), (('she', 'said', 'to', 'herself'), 16), (('a', 'minute', 'or', 'two'), 11), (('said', 'the', 'march', 'hare'), 8), (('will', 'you', 'wont', 'you'), 8)]
```

Script for downloading the stopwords using NLTK

In [7]:
```
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
```

Print 10 Unigrams and Bigrams after removing stopwords

In [8]:
```
print("Most common n-grams with stopword removal and without add-1 smoothing: \n")
unigram_sw_removed = [p for p in unigram if p not in stop_words]
fdist = nltk.FreqDist(unigram_sw_removed)
print("Most common unigrams: ", fdist.most_common(10))
bigram_sw_removed = []
bigram_sw_removed.extend(list(ngrams(unigram_sw_removed, 2)))
fdist = nltk.FreqDist(bigram_sw_removed)
print("\nMost common bigrams: ", fdist.most_common(10))
```

```
Most common n-grams with stopword removal and without add-1 smoothing:
```

Add-1 smoothing

In [9]:
```
ngrams_all = {1:[], 2:[], 3:[], 4:[]}
for i in range(4):
    for each in tokenized_text:
        for j in ngrams(each, i+1):
            ngrams_all[i+1].append(j);
ngrams_voc = {1:set([]), 2:set([]), 3:set([]), 4:set([])}
for i in range(4):
    for gram in ngrams_all[i+1]:
        if gram not in ngrams_voc[i+1]:
            ngrams_voc[i+1].add(gram)
total_ngrams = {1:-1, 2:-1, 3:-1, 4:-1}
total_voc = {1:-1, 2:-1, 3:-1, 4:-1}
for i in range(4):
    total_ngrams[i+1] = len(ngrams_all[i+1])
    total_voc[i+1] = len(ngrams_voc[i+1])

ngrams_prob = {1:[], 2:[], 3:[], 4:[]}
for i in range(4):
    for ngram in ngrams_voc[i+1]:
        tlist = [ngram]
        tlist.append(ngrams_all[i+1].count(ngram))
        ngrams_prob[i+1].append(tlist)

for i in range(4):
    for ngram in ngrams_prob[i+1]:
        ngram[-1] = (ngram[-1]+1)/(total_ngrams[i+1]+total_voc[i+1])
```

Prints top 10 unigram, bigram, trigram, fourgram after smoothing

In [10]:
```
print("Most common n-grams without stopword removal and with add-1 smoothing: \n")
for i in range(4):
    ngrams_prob[i+1] = sorted(ngrams_prob[i+1], key = lambda x:x[1], reverse = True)

print ("Most common unigrams: ", str(ngrams_prob[1][:10]))
print ("\nMost common bigrams: ", str(ngrams_prob[2][:10]))
print ("\nMost common trigrams: ", str(ngrams_prob[3][:10]))
print ("\nMost common fourgrams: ", str(ngrams_prob[4][:10]))
```

Next word Prediction

In [11]:
```
str1 = 'after that alice said the'
str2 = 'alice felt so desperate that she was'
```

In [12]:
```
token_1 = word_tokenize(str1)
token_2 = word_tokenize(str2)
ngram_1 = {1:[], 2:[], 3:[]}   #to store the n-grams formed
ngram_2 = {1:[], 2:[], 3:[]}
for i in range(3):
    ngram_1[i+1] = list(ngrams(token_1, i+1))[-1]
    ngram_2[i+1] = list(ngrams(token_2, i+1))[-1]
print("String 1: ", ngram_1,"\nString 2: ",ngram_2)
```

```
String 1:  {1: ('the',), 2: ('said', 'the'), 3: ('alice', 'said', 'the')}
String 2:  {1: ('was',), 2: ('she', 'was'), 3: ('that', 'she', 'was')}
```

In [13]:
```
for i in range(4):
    ngrams_prob[i+1] = sorted(ngrams_prob[i+1], key = lambda x:x[1], reverse = True)

pred_1 = {1:[], 2:[], 3:[]}
for i in range(3):
    count = 0
    for each in ngrams_prob[i+2]:
        if each[0][:-1] == ngram_1[i+1]:
#to find predictions based on highest probability of n-grams

            count +=1
            pred_1[i+1].append(each[0][-1])
            if count ==5:
                break
    if count<5:
        while(count!=5):
            pred_1[i+1].append("NOT FOUND")
#if no word prediction is found, replace with NOT FOUND
            count +=1
for i in range(4):
    ngrams_prob[i+1] = sorted(ngrams_prob[i+1], key = lambda x:x[1], reverse = True)

pred_2 = {1:[], 2:[], 3:[]}
```

```
pred_2 = {1:[], 2:[], 3:[]}
for i in range(3):
    count = 0
    for each in ngrams_prob[i+2]:
        if each[0][:-1] == ngram_2[i+1]:
            count +=1
            pred_2[i+1].append(each[0][-1])
            if count ==5:
                break
    if count<5:
        while(count!=5):
            pred_2[i+1].append("\0")
            count +=1
```

In [14]:

```
print("Next word predictions for the strings using the probability models of bigrams, trigrams, and fourgrams\n")
print("String 1 - after that alice said the-\n")
print("Bigram model predictions: {}\nTrigram model predictions: {}\nFourgram model predictions: {}\n" .format(pred_1[1], pre
print("String 2 - alice felt so desperate that she was-\n")
print("Bigram model predictions: {}\nTrigram model predictions: {}\nFourgram model predictions: {}" .format(pred_2[1], pred_
```

```
Next word predictions for the strings using the probability models of bigrams, trigrams, and fourgrams

String 1 - after that alice said the-

Bigram model predictions: ['queen', 'king', 'gryphon', 'mock', 'hatter']
Trigram model predictions: ['king', 'hatter', 'mock', 'caterpillar', 'gryphon']
Fourgram model predictions: ['NOT FOUND', 'NOT FOUND', 'NOT FOUND', 'NOT FOUND', 'NOT FOUND']

String 2 - alice felt so desperate that she was-

Bigram model predictions: ['a', 'the', 'not', 'that', 'going']
Trigram model predictions: ['now', 'quite', 'a', 'walking', 'looking']
Fourgram model predictions: ['now', 'losing', 'quite', 'dozing', 'walking']
```

Conclusion:

The N-gram language model was implemented for text analysis, generating unigrams, bigrams, trigrams, and fourgrams from a dataset. Key statistics included 981 sentences, 27,361 tokens, and 3,039 unique tokens. Most common n-grams were identified, with bigrams like ('said', 'the') and trigrams like ('the', 'mock', 'turtle') being the most frequent. Add-1 smoothing was applied to enhance the probability distribution. Next-word predictions were made for two input strings, yielding various bigram and trigram predictions. The model effectively captures word patterns but showed limitations in fourgram predictions, indicating a need for more data or refinement for improved accuracy.