NLP ASSIGNMENT

NIBHRIT GARG
19103110

Q1

Ques 1: Write a program that tokenizes the surface text and performs tasks:

a. Count the number of tokens

b. Word analysis

- Count the frequency of words

- Identify words belonging to different POS tags

```
import nltk
from nltk.tokenize import word_tokenize
```

✓ 12.9s Python

```
s = "Enter the text: This is first. This has $5.00. This is Mr. Nibhrit Garg."

#Tokenising
tokens = word_tokenize(s)

# Getting all tokens
print("Number of tokens:", len(tokens))
tokens
```

✓ 0.3s Python

```
Number of tokens: 19

['Enter',
 'the',
 'text',
 ':',
 'This',
 'is',
 'first',
 '.',
 'This',
 'has',
 '$',
 '5.00',
 '.',
 'This',
 'is',
 '$',
 '5.00',
 '.',
 'This',
 'is',
 'Mr.',
 'Nibhrit',
 'Garg',
 '.']
```

```
# Counting frequency of each word
print(nltk.FreqDist(tokens))
wordDict = dict()
for word in tokens:
    word=word.lower()
    wordDict[word] =  wordDict.get(word, 0) + 1 # storing the frequency of each word.
print(wordDict)
```

✓ 0.1s Python

```
<FreqDist with 14 samples and 19 outcomes>
{'enter': 1, 'the': 1, 'text': 1, ':': 1, 'this': 3, 'is': 2, 'first': 1, '.': 3, 'has': 1, '$': 1, '5.00': 1, 'mr.': 1, 'nibhrit': 1, 'garg': 1}
```

```
# Getting pos tag
from nltk import pos_tag
print(pos_tag(tokens))
```

✓ 0.7s Python

```
[('Enter', 'NNP'), ('the', 'DT'), ('text', 'NN'), (':', ':'), ('This', 'DT'), ('is', 'VBZ'), ('first', 'RB'), ('.', '.'), ('This', 'DT'), ('has', 'VBZ'), ('$', '$'), ('5.00', 'CD'),
('.', '.'), ('This', 'DT'), ('is', 'VBZ'), ('Mr.', 'NNP'), ('Nibhrit', 'NNP'), ('Garg', 'NNP'), ('.', '.')]
```

Q2

Ques 2 : Write a program that identifies whether a string or strings are part of the dicitonary or not.

```python
from nltk.corpus import words
```

✓ 0.2s Python

```python
import re
def check(text):
    wordTokens = word_tokenize(text)

    for word in wordTokens:
        x = re.search("[a-zA-Z0-9]",word)
        if not x:# removing special chars, punctuations;
            continue
        word = word.lower()
        if(not word in words.words()):
            print(word)
            return False
    return True

text = "A check for this sentence."

check(text)
```

✓ 1.2s Python

```
True
```

Q3

Ques 3: Write a program to find minimum edit distance between two strings.

```python
import numpy as np

def editDistance(str1, str2):
    n, m = len(str1), len(str2)
    dp = np.array([[0] * (n + 1)] * (m + 1))
    for i in range(n + 1):
        dp[i][0] = i
    for i in range(m + 1):
        dp[0][i] = i
    for i in range(len(str1)):
        for j in range(len(str2)):
            if str1[i] == str2[j]:
                dp[i + 1][j + 1] = dp[i][j]
            else:
                dp[i + 1][j + 1] = min(1 + min(dp[i][j + 1], dp[i + 1][j]), 2 + dp[i][j])
    return dp[n][m]
```

✓ 0.2s Python

```python
editDistance("intention", "execution")
```

✓ 0.2s Python

```
8
```

Q4

Ques 4: Write a program that takes a word and gives its morphological form as output

```python
from nltk.stem import PorterStemmer
words = ["wait", "waiting", "waited", "waits"]
ps = PorterStemmer()
for w in words:
    rootWord=ps.stem(w)
    print(rootWord)
```

✓ 0.1s Python

```
wait
wait
wait
wait
```

Q5

Ques 5: Given a string as input. Write a program to identify the most probable POS tag sequence for the input using Viterbi algorithm.

```
# Importing libraries
import nltk
import numpy as np
import pandas as pd
import random
from sklearn.model_selection import train_test_split
import pprint, time

#download the treebank corpus from nltk
nltk.download('treebank')

#download the universal tagset from nltk
nltk.download('universal_tagset')

# reading the Treebank tagged sentences
nltk_data = list(nltk.corpus.treebank.tagged_sents(tagset='universal'))

#print the first two sentences along with tags
print(nltk_data[:2])
# split data into training and validation set in the ratio 80:20
train_set,test_set =train_test_split(nltk_data,train_size=0.80,test_size=0.20,random_state = 101)
# create list of train and test tagged words
train_tagged_words = [ tup for sent in train_set for tup in sent ]
test_tagged_words = [ tup for sent in test_set for tup in sent ]
print(len(train_tagged_words))
print(len(test_tagged_words))
```

```
print(len(test_tagged_words))

#use set datatype to check how many unique tags are present in training data
tags = {tag for word,tag in train_tagged_words}
print(len(tags))
print(tags)

# check total words in vocabulary
vocab = {word for word,tag in train_tagged_words}

# compute Emission Probability
def word_given_tag(word, tag, train_bag = train_tagged_words):
    tag_list = [pair for pair in train_bag if pair[1]==tag]
    count_tag = len(tag_list)#total number of times the passed tag occurred in train_bag
    w_given_tag_list = [pair[0] for pair in tag_list if pair[0]==word]
#now calculate the total number of times the passed word occurred as the passed tag.
    count_w_given_tag = len(w_given_tag_list)


    return (count_w_given_tag, count_tag)

# compute  Transition Probability
def t2_given_t1(t2, t1, train_bag = train_tagged_words):
    tags = [pair[1] for pair in train_bag]
    count_t1 = len([t for t in tags if t==t1])
    count_t2_t1 = 0
    for index in range(len(tags)-1):
        if tags[index]==t1 and tags[index+1] == t2:
            count_t2_t1 += 1
    return (count_t2_t1, count_t1)

# creating t x t transition matrix of tags, t= no of tags
# Matrix(i, j) represents P(jth tag after the ith tag)

tags_matrix = np.zeros((len(tags), len(tags)), dtype='float32')
```

```python
tags_matrix = np.zeros((len(tags), len(tags)), dtype='float32')
for i, t1 in enumerate(list(tags)):
    for j, t2 in enumerate(list(tags)):
        tags_matrix[i, j] = t2_given_t1(t2, t1)[0]/t2_given_t1(t2, t1)[1]

print(tags_matrix)

def Viterbi(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)

        pmax = max(p)
        # getting state for which probability is maximum
        state_max = T[p.index(pmax)]
        state.append(state_max)
    return list(zip(words, state))
```

✓ 22.1s Python

```
[nltk_data] Downloading package treebank to
[nltk_data]     C:\Users\Nibhrit\AppData\Roaming\nltk_data...
[nltk_data]   Package treebank is already up-to-date!
[nltk_data] Downloading package universal_tagset to
[nltk_data]     C:\Users\Nibhrit\AppData\Roaming\nltk_data...
[nltk_data]   Package universal_tagset is already up-to-date!

Output exceeds the size limit. Open the full output data in a text editor
[[('Pierre', 'NOUN'), ('Vinken', 'NOUN'), (',', '.'), ('61', 'NUM'), ('years', 'NOUN'), ('old', 'ADJ'), (',', '.'), ('will', 'VERB'), ('join', 'VERB'), ('the', 'DET'), ('board',
'NOUN'), ('as', 'ADP'), ('a', 'DET'), ('nonexecutive', 'ADJ'), ('director', 'NOUN'), ('Nov.', 'NOUN'), ('29', 'NUM'), ('.', '.')], [('Mr.', 'NOUN'), ('Vinken', 'NOUN'), ('is',
'VERB'), ('chairman', 'NOUN'), ('of', 'ADP'), ('Elsevier', 'NOUN'), ('N.V.', 'NOUN'), (',', '.'), ('the', 'DET'), ('Dutch', 'NOUN'), ('publishing', 'VERB'), ('group', 'NOUN'), ('.',
'.')]]
80310
20366
12
{'CONJ', '.', 'X', 'VERB', 'DET', 'ADP', 'NOUN', 'PRT', 'ADV', 'NUM', 'PRON', 'ADJ'}
[[5.48847427e-04 3.51262353e-02 9.33040585e-03 1.50384188e-01
  1.23490669e-01 5.59824370e-02 3.49066973e-01 4.39077942e-03
  5.70801310e-02 4.06147093e-02 6.03732169e-02 1.13611415e-01]
 [6.00793920e-02 9.23720598e-02 2.56410260e-02 8.96899477e-02
  1.72191828e-01 9.29084867e-02 2.18538776e-01 2.78940029e-03
  5.25694676e-02 7.82104954e-02 6.87694475e-02 4.61323895e-02]
 [1.03786280e-02 1.60868734e-01 7.57255405e-02 2.06419379e-01
  5.68902567e-02 1.42225638e-01 6.16951771e-02 1.85085520e-01
  2.57543717e-02 3.07514891e-03 5.41995019e-02 1.76821072e-02]
 [5.43278083e-03 3.48066315e-02 2.15930015e-01 1.67955801e-01
  1.33609578e-01 9.23572779e-02 1.10589318e-01 3.06629837e-02
  8.38858187e-02 2.28360966e-02 3.55432779e-02 6.63904250e-02]
 [4.31220367e-04 1.73925534e-02 4.51343954e-02 4.02472317e-02
  6.03708485e-03 9.91806854e-03 6.35906279e-01 2.87480245e-04
  1.20741697e-02 2.28546783e-02 3.30602261e-03 2.06410810e-01]
```

```
 8.38858187e-02 2.28360966e-02 3.55432779e-02 6.63904250e-02]
[4.31220367e-04 1.73925534e-02 4.51343954e-02 4.02472317e-02
 6.03708485e-03 9.91806854e-03 6.35906279e-01 2.87480245e-04
 1.20741697e-02 2.28546783e-02 3.30602261e-03 2.06410810e-01]
[1.01240189e-03 3.87243740e-02 3.45482156e-02 8.47886596e-03
 3.20931405e-01 1.69577319e-02 3.23588967e-01 1.26550242e-03
 1.45532778e-02 6.32751212e-02 6.96026310e-02 1.07061505e-01]
[4.24540639e-02 2.40094051e-01 2.88252197e-02 1.49133503e-01
 1.31063312e-02 1.76826611e-01 2.62344331e-01 4.39345129e-02
...
 3.69020514e-02 6.83371304e-03 6.83371304e-03 7.06150308e-02]
[1.68932043e-02 6.60194159e-02 2.09708735e-02 1.14563107e-02
 5.24271838e-03 8.05825219e-02 6.96893215e-01 1.14563107e-02
 5.24271838e-03 2.17475723e-02 1.94174761e-04 6.33009672e-02]]
```

```python
# convert the matrix to a df for better readability
#the table is same as the transition table shown in section 3 of article
tags_df = pd.DataFrame(tags_matrix, columns = list(tags), index=list(tags))
display(tags_df)

# Let's test our Viterbi algorithm on a few sample sentences of test dataset
random.seed(1234)      #define a random seed to get same sentences when run multiple times

# choose random 10 numbers
rndom = [random.randint(1,len(test_set)) for x in range(10)]

# list of 10 sents on which we test the model
test_run = [test_set[i] for i in rndom]

# list of tagged words
test_run_base = [tup for sent in test_run for tup in sent]
```

```python
# Let's test our Viterbi algorithm on a few sample sentences of test dataset
random.seed(1234)      #define a random seed to get same sentences when run multiple times

# choose random 10 numbers
rndom = [random.randint(1,len(test_set)) for x in range(10)]

# list of 10 sents on which we test the model
test_run = [test_set[i] for i in rndom]

# list of tagged words
test_run_base = [tup for sent in test_run for tup in sent]

# list of untagged words
test_tagged_words = [tup[0] for sent in test_run for tup in sent]
#Here We will only test 10 sentences to check the accuracy
#as testing the whole training set takes huge amount of time
start = time.time()
tagged_seq = Viterbi(test_tagged_words)
end = time.time()
difference = end-start

print("Time taken in seconds: ", difference)

# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]

accuracy = len(check)/len(tagged_seq)
print('Viterbi Algorithm Accuracy: ',accuracy*100)
```

✓ 1m 20.7s Python

	CONJ	.	X	VERB	DET	ADP	NOUN	PRT	ADV	NUM	PRON	ADJ
CONJ	0.000549	0.035126	0.009330	0.150384	0.123491	0.055982	0.349067	0.004391	0.057080	0.040615	0.060373	0.113611
.	0.060079	0.092372	0.025641	0.089690	0.172192	0.092908	0.218539	0.002789	0.052569	0.078210	0.068769	0.046132
X	0.010379	0.160869	0.075726	0.206419	0.056890	0.142226	0.061695	0.185086	0.025754	0.003075	0.054200	0.017682
VERB	0.005433	0.034807	0.215930	0.167956	0.133610	0.092357	0.110589	0.030663	0.083886	0.022836	0.035543	0.066390
DET	0.000431	0.017393	0.045134	0.040247	0.006037	0.009918	0.635906	0.000287	0.012074	0.022855	0.003306	0.206411
ADP	0.001012	0.038724	0.034548	0.008479	0.320931	0.016958	0.323589	0.001266	0.014553	0.063275	0.069603	0.107062
NOUN	0.042454	0.240094	0.028825	0.149134	0.013106	0.176827	0.262344	0.043935	0.016895	0.009144	0.004659	0.012584
PRT	0.002348	0.045010	0.012133	0.401174	0.101370	0.019569	0.250489	0.001174	0.009393	0.056751	0.017613	0.082975
ADV	0.006982	0.139255	0.022886	0.339022	0.071373	0.119472	0.032196	0.014740	0.081458	0.029868	0.012025	0.130721
NUM	0.014281	0.119243	0.202428	0.020707	0.003570	0.037487	0.351660	0.026062	0.003570	0.184220	0.001428	0.035345
PRON	0.005011	0.041913	0.088383	0.484738	0.009567	0.022323	0.212756	0.014123	0.036902	0.006834	0.006834	0.070615
ADJ	0.016893	0.066019	0.020971	0.011456	0.005243	0.080583	0.696893	0.011456	0.005243	0.021748	0.000194	0.063301

```
Time taken in seconds:  79.92651629447937
Viterbi Algorithm Accuracy:  93.77990430622009
```

```python
#Check how a sentence is tagged by the two POS taggers
#and compare them
test_sent="Will can see Marry"
# pred_tags_rule=Viterbi_rule_based(test_sent.split())
pred_tags_withoutRules= Viterbi(test_sent.split())
# print(pred_tags_rule)
print(pred_tags_withoutRules)
#Will and Marry are tagged as NUM as they are unknown words for Viterbi Algorithm
```

✓ 1.6s Python

```
[('Will', 'CONJ'), ('can', 'VERB'), ('see', 'VERB'), ('Marry', 'CONJ')]
```

+ Code + Markdown

Q6

Ques 6: Explore different parsers and write code snippets to show their implementation.

```python
import nltk
from nltk.tag import pos_tag
from nltk.tokenize import word_tokenize
from nltk.tree import *
from nltk.draw import tree
from nltk import Nonterminal, nonterminals, Production, CFG
from nltk.parse import RecursiveDescentParser
text = "saw a dog"
words = text.split()
sentence = pos_tag(words)

grammar1 = nltk.CFG.fromstring("""
    S -> NP VP
    S -> VP
    VP -> V NP | V NP PP
    NP ->  Det N | Det N PP
    PP -> P NP
    V -> "saw" | "ate" | "walked" | "book" | "prefer" | "sleeps"
    Det -> "a" | "an" | "the" | "my" | "that"
    N -> "man" | "dog" | "cat" | "telescope" | "park" | "flight" | "apple"
    P -> "in" | "on" | "by" | "with"
     """)

rd = nltk.RecursiveDescentParser(grammar1)
result = rd.parse(words)
for p in result:
        print(p)
```

✓ 0.2s Python Python

```
(S (VP (V saw) (NP (Det a) (N dog))))
```

```python
nltk.parse.shiftreduce.demo()
```

✓ 0.3s Python

```
Parsing 'I saw a man in the park'
    [ * I saw a man in the park]
  S [ 'I' * saw a man in the park]
  R [ NP * saw a man in the park]
  S [ NP 'saw' * a man in the park]
  R [ NP V * a man in the park]
  S [ NP V 'a' * man in the park]
  R [ NP V Det * man in the park]
  S [ NP V Det 'man' * in the park]
  R [ NP V Det N * in the park]
  R [ NP V NP * in the park]
  R [ NP VP * in the park]
  R [ S * in the park]
  S [ S 'in' * the park]
  R [ S P * the park]
  S [ S P 'the' * park]
  R [ S P Det * park]
  S [ S P Det 'park' * ]
  R [ S P Det N * ]
  R [ S P NP * ]
  R [ S PP * ]
```

Q7

Ques 7: Write a program that takes sentence (atleast one word with multiple senses) as input and performs word sense disambiguation using the context

```
from nltk.wsd import lesk
from nltk.tokenize import word_tokenize
a1= lesk(word_tokenize('I like to eat bread jam'),'jam')
print(a1,a1.definition())
a2 = lesk(word_tokenize('I am stuck in a traffic jam'),'jam')
print(a2,a2.definition())
```

✓ 4.5s Python

```
Synset('jam.v.06') crowd or pack to capacity
Synset('jam.v.05') get stuck and immobilized
```

Q8

Ques 8: Write a program for word generation given a context

```
from nltk.util import ngrams
from nltk.corpus import brown

corpus_tokens = brown.words()

input_string ="I would like to have a"
input_string_tokens = word_tokenize(input_string)


def word_predictor(n):
    frequencies = nltk.FreqDist(ngrams(corpus_tokens, n))
    frequencies_list = [(k, v) for k, v in dict(frequencies).items()]
    frequencies_list = sorted(
        frequencies_list, key=lambda x: x[-1], reverse=True)
    ngram = tuple(ngrams(input_string_tokens, n-1))[-1]
    predictions = []
    count = 0
    print(ngram)
    for each in frequencies_list:
        if each[0][:-1] == ngram:
            count += 1
            predictions.append(each[0][-1])
            if count == 5:
                break

    if count < 5:
        while(count != 5):
            predictions.append("NONE")
            count += 1
```

```
            count += 1
            predictions.append(each[0][-1])
            if count == 5:
                break

    if count < 5:
        while(count != 5):
            predictions.append("NONE")
            count += 1

    return predictions


print("\nPredictions of the next word for the input line : \n")
print("Bigram model predictions   : ", word_predictor(2))
print("Trigram model predictions  : ", word_predictor(3))
print("4gram model predictions  : ", word_predictor(4))
```

✓ 42.8s Python

```
Predictions of the next word for the input line :

('a',)
Bigram model predictions   :  ['few', 'little', 'man', 'new', 'good']
('have', 'a')
Trigram model predictions  :  ['look', 'drink', 'chance', 'good', 'few']
('to', 'have', 'a')
4gram model predictions  :  ['few', 'specific', 'baby', 'complete', 'session']
```