EXP 6

Name: Bodhisatya Ghosh
UID: 2021700026
Branch: CSE DS
Experiment: 6
Batch: A

AIM: POS Tagging using Viterbi Algorithm

THEORY: POS Tagging using Viterbi Algorithm

POS tagging, or Part-of-Speech tagging, is the process of assigning grammatical tags (such as noun, verb, adjective, etc.) to words in a sentence. It is an important task in natural language processing and is used in various applications like text analysis, information retrieval, and machine translation.

The Viterbi algorithm is a dynamic programming algorithm that is commonly used for POS tagging. It finds the most likely sequence of hidden states (POS tags) that corresponds to a sequence of observed words in a sentence. The algorithm is based on the assumption that the current POS tag depends only on the previous POS tag, making it a first-order Markov process.

The Viterbi algorithm works by calculating the probability of each possible tag sequence for a given sentence and selecting the sequence with the highest probability. It uses a combination of transition probabilities (the probability of transitioning from one tag to another) and emission probabilities (the probability of a word being assigned a particular tag) to calculate these probabilities.

The algorithm starts with an initial probability distribution and iteratively calculates the probability of each tag sequence by considering the probabilities of transitioning from the previous tag to the current tag and the emission probability of the current word given the current tag. It keeps track of the highest probability for each tag at each position in the sentence and the corresponding backpointer (the previous tag that leads to the highest probability).

Once the algorithm has processed the entire sentence, it backtracks from the last position to find the most likely tag sequence by following the backpointers. This sequence represents the POS tags for the given sentence.

The Viterbi algorithm is widely used for POS tagging due to its efficiency and accuracy. However, it may encounter challenges with unknown words or words that have multiple possible POS tags. To address these challenges, rule-based taggers or additional statistical models can be incorporated into the algorithm.

Overall, the Viterbi algorithm is a powerful tool for POS tagging and plays a crucial role in various natural language processing tasks.

CODE

In [ ]:
```
# 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])
```

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

```
[[('Pierre', 'NOUN'), ('Vinken', 'NOUN'), (',', '.'), ('61', 'NUM'), ('years', 'NOUN'), ('old', 'ADJ'), (',', '.'), ('will', 'V
ERB'), ('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', 'NOU
N'), ('of', 'ADP'), ('Elsevier', 'NOUN'), ('N.V.', 'NOUN'), (',', '.'), ('the', 'DET'), ('Dutch', 'NOUN'), ('publishing', 'VER
B'), ('group', 'NOUN'), ('.', '.')]]
```

In [ ]:

```
#print each word with its respective tag for first two sentences
for sent in nltk_data[:2]:
  for tuple in sent:
    print(tuple)
```

```
('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')
('.', '.')
```

In [ ]:

```
# 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)
```

In [ ]:

```
# 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))
```

```
80310
20366
```

In [ ]:
```python
# check some of the tagged words.
train_tagged_words[:5]
```

Out[ ]:
```
[('Drink', 'NOUN'),
 ('Carrier', 'NOUN'),
 ('Competes', 'VERB'),
 ('With', 'ADP'),
 ('Cartons', 'NOUN')]
```

In [ ]:
```python
#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}
```

```
12
{'ADJ', 'DET', 'ADV', '.', 'NUM', 'CONJ', 'PRON', 'PRT', 'VERB', 'NOUN', 'X', 'ADP'}
```

In [ ]:
```python
# 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)
```

In [ ]:
```python
# 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)
```

In [ ]:
```
# 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')
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)
```

In [ ]:
```
# 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)
```

In [ ]:
```
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))
```

In [ ]:
```
# 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]
```

In [ ]:
```
#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)
```

```
Time taken in seconds:  29.396782636642456
Viterbi Algorithm Accuracy:  93.77990430622009
```

In [ ]:
```
#Code to test all the test sentences
#(takes alot of time to run s0 we wont run it here)
# tagging the test sentences()
test_tagged_words = [tup for sent in test_set for tup in sent]
```

```
test_untagged_words = [tup[0] for sent in test_set for tup in sent]
test_untagged_words

start = time.time()
tagged_seq = Viterbi(test_untagged_words)
end = time.time()
difference = end-start

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

# accuracy
check = [i for i, j in zip(test_tagged_words, test_untagged_words) if i == j]

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

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[14], line 9
      6 test_untagged_words
      8 start = time.time()
----> 9 tagged_seq = Viterbi(test_untagged_words)
     10 end = time.time()
     11 difference = end-start

Cell In[11], line 15, in Viterbi(words, train_bag)
     12     transition_p = tags_df.loc[state[-1], tag]
     14 # compute emission and state probabilities
---> 15 emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
     16 state_probability = emission_p * transition_p
     17 p.append(state_probability)

Cell In[7], line 3, in word_given_tag(word, tag, train_bag)
      2 def word_given_tag(word, tag, train_bag = train_tagged_words):
----> 3     tag_list = [pair for pair in train_bag if pair[1]==tag]
      4     count_tag = len(tag_list)#total number of times the passed tag occurred in train_bag
      5     w_given_tag_list = [pair[0] for pair in tag_list if pair[0]==word]

Cell In[7], line 3, in <listcomp>(.0)
      2 def word_given_tag(word, tag, train_bag = train_tagged_words):
----> 3     tag_list = [pair for pair in train_bag if pair[1]==tag]
      4     count_tag = len(tag_list)#total number of times the passed tag occurred in train_bag
      5     w_given_tag_list = [pair[0] for pair in tag_list if pair[0]==word]

KeyboardInterrupt: 
```

In [ ]:

```
#To improve the performance,we specify a rule base tagger for unknown words
# specify patterns for tagging
patterns = [
    (r'.*ing$', 'VERB'),              # gerund
    (r'.*ed$', 'VERB'),               # past tense
    (r'.*es$', 'VERB'),               # verb
    (r'.*\'s$', 'NOUN'),              # possessive nouns
    (r'.*s$', 'NOUN'),                # plural nouns
    (r'\*T?\*?-[0-9]+$', 'X'),        # X
    (r'^-?[0-9]+(.[0-9]+)?$', 'NUM'), # cardinal numbers
```

```
    (r'.*', 'NOUN')                   # nouns
]

# rule based tagger
rule_based_tagger = nltk.RegexpTagger(patterns)
```

In [ ]:

```
#modified Viterbi to include rule based tagger in it
def Viterbi_rule_based(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)
        state_max = rule_based_tagger.tag([word])[0][1]


        if(pmax==0):
            state_max = rule_based_tagger.tag([word])[0][1] # assign based on rule based tagger
        else:
            if state_max != 'X':
                # getting state for which probability is maximum
                state_max = T[p.index(pmax)]


        state.append(state_max)
    return list(zip(words, state))
```

In [ ]:
```
#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
```

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

CONCLUSION

The experiment conducted in this notebook focused on POS tagging using the Viterbi algorithm. The results showed that the Viterbi algorithm achieved a high accuracy in assigning POS tags to words. Overall, the experiment demonstrated the effectiveness of the Viterbi algorithm for POS tagging and highlighted its importance in various NLP applications