Name	Pranay Singhvi
UID No.	2021300126
Experiment No.	6

Experiment 6

Aim	To calculate emission and transition matrix and find Find POS tags of ws in a sentence using Viterbi decoding
Theory	In natural language processing, part-of-speech (POS) tagging is the process of assigning a grammatical category (such as noun, verb, adjective, etc.) to each word in a sentence. This is an important task in many NLP applications, as it helps in understanding the structure and meaning of text.

1. Hidden Markov Models (HMM):

Hidden Markov Models are statistical models used for modeling systems that transition between different states over time. In the context of natural language processing, HMMs are often used to model sequences of words and their associated POS tags.

2. Emission Matrix:

The emission matrix represents the probabilities of observing a certain word given a particular POS tag. Each row corresponds to a POS tag, and each column corresponds to a word. The cell (i, j) contains the probability of the word j being emitted by the POS tag i.

$$\text{Emission Matrix}[i, j] = P(\text{word } j \mid \text{POS tag } i)$$

3. Transition Matrix:

The transition matrix represents the probabilities of transitioning from one POS tag to another. Each row corresponds to a current POS tag, and each column corresponds to a possible next POS tag. The cell (i, j) contains the probability of transitioning from POS tag i to POS tag j.

$$\text{Transition Matrix}[i, j] = P(\text{POS tag } j \mid \text{POS tag } i)$$

4. Viterbi Decoding:

Viterbi decoding is a dynamic programming algorithm used to find the most likely sequence of hidden states in a Hidden Markov Model. In the context of POS tagging, Viterbi decoding helps find the most probable sequence of POS tags for a given sequence of words.

The algorithm involves calculating probabilities recursively and keeping track of the most probable path at each step. The steps are as follows:

- **Initialization:** Initialize the probability matrix for the first word using emission probabilities.
- **Recursion:** For each subsequent word, calculate the probability of reaching each state from the previous states by considering both emission and transition probabilities.
- **Backtracking:** Keep track of the most likely path at each step.
- **Termination:** The final state in the sequence is the one with the highest probability.

5. Application to POS Tagging:

Given a sentence, you can use the emission and transition matrices to apply Viterbi decoding and determine the most likely sequence of POS tags for each word in the sentence.

$$\text{POS Tags} = \text{Viterbi Decoding}(\text{Emission Matrix}, \text{Transition Matrix}, \text{Sentence})$$

This process helps in automatically assigning POS tags to words in a sentence based on statistical probabilities derived from training data.

Viterbi Algorithm:

1. **Initialization:**
   - Initialize the probability matrix, `V`, with dimensions `[num_states, num_words]`.
   - Set the initial probabilities in the first column based on the product of the initial probabilities of states and emission probabilities of the first word.
2. **Recursion:**
   - For each subsequent word in the sentence:
     - Calculate the probability of transitioning from the previous state to the current state, multiplied by the emission probability of the current word.
     - Update the probability matrix with the maximum probability and the corresponding backpointer.

$$V[i, j] = \max(V[k, j-1] \times \text{transition}[k, i] \times \text{emission}[i, \text{word}_j]), \text{ for all } i$$

3. **Backtracking:**
   - Start from the final state with the highest probability in the last column.
   - Follow the backpointers to trace back the most likely sequence of states.
4. **Termination:**
   - The final sequence of POS tags is obtained by backtracking from the state with the highest probability in the last column.

Explanation:

The Viterbi algorithm is based on dynamic programming principles. It efficiently computes the probability of the most likely sequence of states by breaking down the problem into smaller subproblems and reusing the solutions to these subproblems. The algorithm maintains a matrix of probabilities and backpointers, allowing it to track the most probable path at each step.

By considering both the transition probabilities between states and the emission probabilities for observing specific words, Viterbi decoding finds the sequence of hidden states (POS tags) that maximizes the overall probability of the observed sequence of words. The backtracking step then allows us to reconstruct this most likely sequence.

It's a fundamental algorithm used in various applications, including POS tagging, speech recognition, and bioinformatics. Its efficiency and optimality make it a valuable tool for finding the most probable sequence of hidden states in a Hidden Markov Model.

Advantages of Viterbi Algorithm:

1. **Efficiency:** The Viterbi algorithm is computationally efficient, with a time complexity that is linear with respect to the length of the sequence.
2. **Optimality:** It guarantees the optimal solution, meaning that the sequence it produces has the maximum probability given the model parameters.

Disadvantages of Viterbi Algorithm:

1. **Assumption of Independence:** Viterbi assumes that the observations (words) are conditionally independent given the hidden states. This may not always hold true in real-world language usage.
2. **Sensitivity to Model Parameters:** The performance of Viterbi decoding heavily depends on the accuracy of the emission and transition probabilities derived from the training data. Inaccuracies in these parameters can lead to suboptimal results.

1. Installation of NLTK and downloading the required corpus

In [ ]:
```
import re
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from prettytable import PrettyTable
from collections import defaultdict
warnings.filterwarnings('ignore')
```

2. Loading the corpus and preprocessing

In [ ]:
```
# load csv
df = pd.read_csv('exp5.csv', encoding='iso-8859-1')
df1 = df[df['Sentence #'].notna()]
print("There are",df1['Sentence #'].iloc[-1].split()[-1],"sentences in the dataset")
df.drop(['Sentence #', 'Tag'], axis=1, inplace=True)
df.head()
```

```
There are 47959 sentences in the dataset
```

Out[ ]:

	Word	POS
0	Thousands	NNS
1	of	IN
2	demonstrators	NNS
3	have	VBP
4	marched	VBN

In [ ]:
```
# print all unique values in POS column
print("Unique values in POS column:",df['POS'].unique())
```

```
Unique values in POS column: ['NNS' 'IN' 'VBP' 'VBN' 'NNP' 'TO' 'VB' 'DT' 'NN' 'CC' 'JJ' '.' 'VBD' 'WP'
 '``' 'CD' 'PRP' 'VBZ' 'POS' 'VBG' 'RB' ',' 'WRB' 'PRP$' 'MD' 'WDT' 'JJR'
 ':' 'JJS' 'WP$' 'RP' 'PDT' 'NNPS' 'EX' 'RBS' 'LRB' 'RRB' '$' 'RBR' ';'
 'UH' 'FW']
```

In [ ]:
```
def preprocess(text):
    text = text.lower()
    text = re.sub(r'[^\w\s]', '', text) # remove punctuation
    text = text.replace("\n", " ") # remove \n
    text = re.sub(r'\W', ' ', text)  # Remove non-w characters
    text = re.sub(r'\s+', ' ', text).strip()  # Remove extra whitespaces
    text = re.sub(r'\d', '', text)  # Remove digits
    return text
```

In [ ]:
```
tag_mapping = {
    'NN': 'NOUN',
    'NNS': 'NOUN',
    'NNP': 'NOUN',
    'NNPS': 'NOUN',
    'VB': 'VERB',
    'VBD': 'VERB',
    'VBG': 'VERB',
```

```
    'VBN': 'VERB',
    'VBP': 'VERB',
    'VBZ': 'VERB',
    'JJ': 'ADJ',
    'JJR': 'ADJ',
    'JJS': 'ADJ',
    'RB': 'ADV',
    'RBR': 'ADV',
    'RBS': 'ADV',
}
```

3. Building Vocabulary

In [ ]:
```
# convert the dataframe to a dictionary, make value field as list of all the tags of that w in the sentence
vocab = {}
for index, row in df.iterrows():
    w = row['Word']
    pos = row['POS']
    tag = tag_mapping.get(row['POS'], 'MODAL')
    # if only string
    if type(w) == str:
        if w == ';' or w == ':' or w == '``' or w == ',' or w == '.':
            continue
        else:
            w = preprocess(w)
    else:
        w = str(w)
        continue
    w = preprocess(w)
    if w in vocab and tag not in vocab[w]:
        vocab[w].append(tag)
    else:
        if w not in vocab:
            vocab[w] = [tag]
```

4. Calculating Emission & Transition Probabilities

In [ ]:
```
emission_matrix = defaultdict(lambda: defaultdict(int))
    # calculate the emission probability and store it in the emission matrix
for index, row in df.iterrows():
    w = row['Word']
    tag = tag_mapping.get(row['POS'], 'MODAL')
    if type(w) == str:
        w = preprocess(w)
    else:
        w = str(w)
        continue
    w = preprocess(w)
    emission_matrix[w][tag] += 1
```

In [ ]:
```
emission_table = PrettyTable()
emission_table.field_names = [""] + list(set(tag_mapping.values())) + ['MODAL']
for w in emission_matrix:
    total = sum(emission_matrix[w].values())
    prob = {tag: round(emission_matrix[w][tag] / total, 2) for tag in emission_table.field_names[1:]}
    emission_table.add_row([w] + list(prob.values()))
print("Emission Matrix:")
# print only first 10 rows
print(emission_table[:10])
```

```
Emission Matrix:
+---------------+-----+------+------+-----+-------+
|               | ADV | VERB | NOUN | ADJ | MODAL |
+---------------+-----+------+------+-----+-------+
|   thousands   | 0.0 | 0.0  | 1.0  | 0.0 |  0.0  |
|       of      | 0.0 | 0.0  | 0.0  | 0.0 |  1.0  |
| demonstrators | 0.0 | 0.0  | 1.0  | 0.0 |  0.0  |
|      have     | 0.0 | 1.0  | 0.0  | 0.0 |  0.0  |
|    marched    | 0.0 | 1.0  | 0.0  | 0.0 |  0.0  |
|    through    | 0.0 | 0.0  | 0.0  | 0.0 |  1.0  |
|     london    | 0.0 | 0.0  | 1.0  | 0.0 |  0.0  |
|       to      | 0.0 | 0.0  | 0.0  | 0.0 |  1.0  |
|    protest    | 0.0 | 0.48 | 0.52 | 0.0 |  0.0  |
|      the      | 0.0 | 0.0  | 0.0  | 0.0 |  1.0  |
+---------------+-----+------+------+-----+-------+
```

In [ ]:
```
transition_matrix = defaultdict(lambda: defaultdict(int))
previous_tag = None
for index, row in df.iterrows():
    tag = tag_mapping.get(row['POS'], 'MODAL')
    if previous_tag is not None:
        transition_matrix[previous_tag][tag] += 1
    previous_tag = tag
```

In [ ]:
```
print("\nTransition Matrix:")
trans_table = PrettyTable()
trans_table.field_names = [""] + list(set(tag_mapping.values())) + ['MODAL']
for tag in transition_matrix:
    total = sum(transition_matrix[tag].values())
    prob = {}
    for tg in set(tag_mapping.values()) | {'MODAL'}:
        prob[tg] = round(transition_matrix[tag][tg] / total, 2)
    trans_table.add_row([tag] + list(prob.values()))
print(trans_table)
```

```
Transition Matrix:
+-------+------+------+------+------+-------+
|       | ADV  | VERB | NOUN | ADJ  | MODAL |
+-------+------+------+------+------+-------+
|  NOUN | 0.01 | 0.55 | 0.18 | 0.25 |  0.01 |
| MODAL | 0.02 | 0.31 | 0.13 | 0.41 |  0.14 |
|  VERB | 0.05 | 0.54 | 0.18 | 0.16 |  0.07 |
|  ADJ  | 0.0  | 0.13 | 0.01 | 0.77 |  0.09 |
|  ADV  | 0.05 | 0.39 | 0.41 | 0.04 |  0.1  |
+-------+------+------+------+------+-------+
```

5. Predicting POS tags using Viterbi Algorithm

In [ ]:
```
# Viterbi Algorithm
def viterbi(ws, emission_matrix, transition_matrix):
    tags = list(set(tag_mapping.values())) + ['MODAL']
    pi = np.zeros((len(ws), len(tags)))
    bp = np.zeros((len(ws), len(tags)), dtype=int)
    for i, w in enumerate(ws):
        for j, tag in enumerate(tags):
            if i == 0:
                pi[i][j] = 1
            else:
                max_prob = -1
                max_prob_index = -1
                for k, prev_tag in enumerate(tags):
                    if emission_matrix[w][tag] == 0:
                        emission_matrix[w][tag] = 0.0001
                    if transition_matrix[prev_tag][tag] == 0:
                        transition_matrix[prev_tag][tag] = 0.0001
                    prob = pi[i-1][k] * emission_matrix[w][tag] * transition_matrix[prev_tag][tag]
                    if prob > max_prob:
                        max_prob = prob
                        max_prob_index = k
                pi[i][j] = max_prob
                bp[i][j] = max_prob_index
    max_prob = -1
    max_prob_index = -1
    for j, tag in enumerate(tags):
        if pi[-1][j] > max_prob:
            max_prob = pi[-1][j]
            max_prob_index = j
    predicted_tags = [tags[max_prob_index]]
    for i in range(len(ws)-1, 0, -1):
        max_prob_index = bp[i][max_prob_index]
        predicted_tags.append(tags[max_prob_index])
    return list(reversed(predicted_tags))
```

In [ ]:
```
sample_sentence = "The sun dipped below the horizon, casting a warm, golden glow across the tranquil, rippling wate
predicted_tags = viterbi(sample_sentence.split(), emission_matrix, transition_matrix)

table = PrettyTable(['Word', 'Predicted POS Tag'])

for word, tag in zip(sample_sentence.split(), predicted_tags):
    table.add_row([word, tag])

print("\nPredicted tags for the sample sentence:")
print(table)
```

```
Predicted tags for the sample sentence:
+-----------+-------------------+
|    Word   | Predicted POS Tag |
+-----------+-------------------+
|    The    |       MODAL       |
|    sun    |        NOUN       |
|   dipped  |        VERB       |
|   below   |       MODAL       |
|    the    |       MODAL       |
|  horizon, |        NOUN       |
|  casting  |        VERB       |
|     a     |       MODAL       |
|   warm,   |       MODAL       |
|   golden  |        NOUN       |
|    glow   |       MODAL       |
|   across  |       MODAL       |
|    the    |       MODAL       |
| tranquil, |        NOUN       |
|  rippling |       MODAL       |
|   waters  |        NOUN       |
|     of    |       MODAL       |
|    the    |       MODAL       |
|   lake.   |        NOUN       |
+-----------+-------------------+
```

6. Conclusion

In this experiment we learned how to calculate the emission and transition matrix for tagging Parts of Speech. We also learned how to find the POS tags of a given sentence using Viterbi decoding.