Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Bhuvi Ghosh
60009210191

COURSE CODE: **DJ19DSL602** YEAR**: 2023-2024** DATE: **11 / 03 / 2024**

COURSE NAME: **Computational Linguistics Laboratory** CLASS: **T.Y.B.Tech (D1)**

Experiment 4

AIM: Implement Hidden Markov Model for Parts of Speech tagging using python.

Theory :-

Tagging is a kind of classification that may be defined as the automatic assignment of description to the tokens. Here the descriptor is called tag, which may represent one of the part-of-speech, semantic information and so on. Now, if we talk about Part-of-Speech (PoS) tagging, then it may be defined as the process of assigning one of the parts of speech to the given word. It is generally called POS tagging. In simple words, we cansay that POS tagging is a task of labelling each word in a sentence with its appropriate part of speech. We already know that parts of speech include nouns, verb, adverbs, adjectives, pronouns, conjunction and their sub-categories.

1. Rule-based POS Tagging

One of the oldest techniques of tagging is rule-based POS tagging. Rule-based taggers use dictionary or lexicon for getting possible tags for tagging each word. If the word has more than one possible tag, then rule-based taggers use hand-written rules to identify the correct tag. Disambiguation can also be performed in rule-based tagging by analyzing the linguistic features of a word along with its preceding as well as following words.

2. Stochastic POS Tagging

Another technique of tagging is Stochastic POS Tagging. Now, the question that arises here is which model can be stochastic. The model that includes frequency or probability (statistics) can be called stochastic. Any number of different approaches to the problem of part-of-speech tagging can be referred to as stochastic tagger.

Hidden Markov Model (HMM) POS Tagging

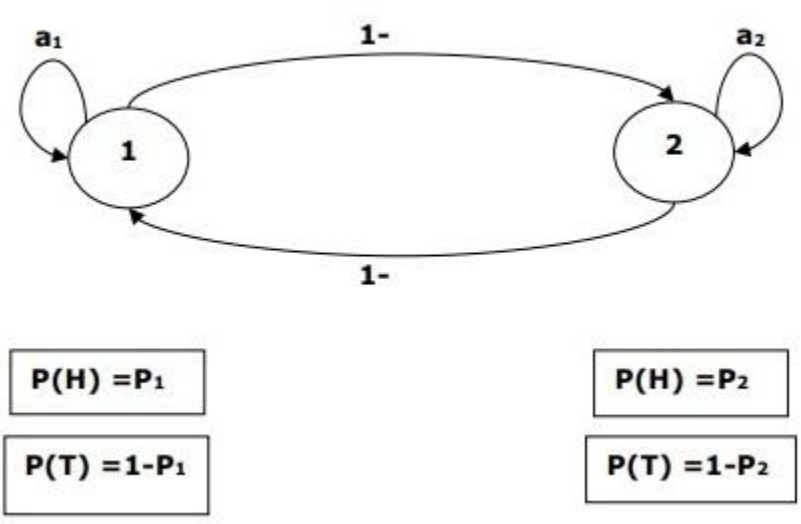
Before digging deep into HMM POS tagging, we must understand the concept of Hidden Markov Model (HMM).

Hidden Markov Model

An HMM model may be defined as the doubly-embedded stochastic model, where the underlying stochastic process is hidden. This hidden stochastic process can only be observed through another set of stochastic processes that produces the sequence of observations.

Example

For example, a sequence of hidden coin tossing experiments is done and we see only the observation sequence consisting of heads and tails. The actual details of the process - how many coins are used, the order in which they are selected - are hidden from us. By observing this sequence of heads and tails, we can build several HMMs to explain the sequence. Following is one form of Hidden Markov Model for this problem −



We assumed that there are two states in the HMM and each of the state corresponds to the selection of different biased coin. Following matrix gives the state transition probabilities –

$$A = \begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix}$$

Here,

aij = probability of transition from one state to another from i to j.

a11 + a12 = 1 and a21 + a22 =1

P1 = probability of heads of the first coin i.e. the bias of the first coin.

P2 = probability of heads of the second coin i.e. the bias of the second coin.

We can also create an HMM model assuming there are 3 coins or more.

This way, we can characterize HMM by the following elements −
N, the number of states in the model (in the above example N =2, only two states).
M, the number of distinct observations that can appear with each state in the above example M = 2, i.e., H or T).
A, the state transition probability distribution − the matrix A in the above example.
P, the probability distribution of the observable symbols in each state (in our example P1 and P2). the initial state distribution.

Use of HMM for POS Tagging

The POS tagging process is the process of finding the sequence of tags which is most likely to have generated a given word sequence. We can model this POS process by using a Hidden Markov Model (HMM), where tags are the hidden states that produced the observable output, i.e., the words.
Mathematically, in POS tagging, we are always interested in finding a tag sequence (C) which maximizes −
P (C|W)
Where,
C = C1, C2, C3... CT
W = W1, W2, W3, WT
On the other side of coin, the fact is that we need a lot of statistical data to reasonably estimate such kind of sequences. However, to simplify the problem, we can apply some mathematical transformations along with some assumptions.
The use of HMM to do a POS tagging is a special case of Bayesian interference. Hence, we will start by restating the problem using Bayes' rule, which says that the above-mentioned conditional probability is equal to −
(PROB (C1,..., CT) * PROB (W1,..., WT | C1,..., CT)) / PROB (W1,..., WT)
We can eliminate the denominator in all these cases because we are interested in finding the sequence C which maximizes the above value. This will not affect our answer. Now, our problem reduces to finding the sequence C that maximizes −
PROB (C1,..., CT) * PROB (W1,..., WT | C1,..., CT) (1)
Even after reducing the problem in the above expression, it would require large amount of data. We can make reasonable independence assumptions about the two probabilities in the above expression to overcome the problem.
First Assumption
The probability of a tag depends on the previous one (bigram model) or previous two (trigram model) or previous n tags (n-gram model) which, mathematically, can be explained as follows −
PROB (C1,..., CT) = Πi=1..T PROB (Ci|Ci-n+1…Ci-1) (n-gram model)
PROB (C1,..., CT) = Πi=1..T PROB (Ci|Ci-1) (bigram model)
The beginning of a sentence can be accounted for by assuming an initial probability for each tag.
PROB (C1|C0) = PROB initial (C1)
Second Assumption
The second probability in equation (1) above can be approximated by assuming that a word appears in a category independent of the words in the preceding or succeeding categories which

can be explained mathematically as follows −
PROB (W1,..., WT | C1,..., CT) = Πi=1..T PROB (Wi|Ci)
Now, on the basis of the above two assumptions, our goal reduces to finding a sequence C which maximizes
Πi=1...T PROB(Ci|Ci-1) * PROB(Wi|Ci)
Now the question that arises here is has converting the problem to the above form really helped us. The answer is - yes, it has. If we have a large tagged corpus, then the two probabilities in the above formula can be calculated as −
PROB (Ci=VERB|Ci-1=NOUN) = (# of instances where Verb follows Noun) / (# of instances where Noun appears) (2)
PROB (Wi|Ci) = (# of instances where Wi appears in Ci) /(# of instances where Ci appears) (3)

Algorithm for POS tagging Using HMM

1. use a corpus with word/tags like brown or wsj
2. add <start> and <end> tags before and after each sequence
3. compute emission probs based on maximum likelihood estimation
4. compute transition probs
5. implement viterbi to find the most likely tagging of a sentence (see it as an extension of shortest distance in a graph)
6. find a word/pos tag dictionary to constrain possible predictions

3. Transformation-based Tagging

Transformation based tagging is also called Brill tagging. It is an instance of the transformation-based learning (TBL), which is a rule-based algorithm for automatic tagging of POS to the given text. TBL, allows us to have linguistic knowledge in a readable form, transforms one state to another state by using transformation rules.

Lab Experiment to be performed in this session: -

1. Implement Hidden Markov Model for POS tagging using python

   Steps to be followed

   1. Upload Dataset
   2. Split Train and Test
   3. Calculate emission probability matrix for train
   4. Calculate transition probability matrix for train
   5. Implement Vertibi Algorithm for finding tag of unseen data

```
import numpy as np
import random

def generate_dataset(num_sentences, max_sentence_length, tags, word_tag_dict):
    dataset = []
    for _ in range(num_sentences):
        sentence_length = random.randint(1, max_sentence_length)
        sentence = []
        for _ in range(sentence_length):
            tag = random.choice(tags)
            word = random.choice(word_tag_dict[tag])
            sentence.append(f"{word}/{tag}")
        dataset.append(" ".join(sentence))
    return dataset
```

```
def split_data(data, train_ratio=0.8):
    split_idx = int(len(data) * train_ratio)
    train_data = data[:split_idx]
    test_data = data[split_idx:]
    return train_data, test_data
```

```
def calc_emission_probs(train_data):
    emission_probs = {}
    tag_counts = {}
    for sentence in train_data:
        words = sentence.split()
        for word, tag in [pair.split('/') for pair in words]:
            if tag not in emission_probs:
                emission_probs[tag] = {}
                tag_counts[tag] = 0
            if word not in emission_probs[tag]:
                emission_probs[tag][word] = 0
            emission_probs[tag][word] += 1
            tag_counts[tag] += 1

    for tag in emission_probs:
        for word in emission_probs[tag]:
            emission_probs[tag][word] /= tag_counts[tag]

    return emission_probs
```

```
def calc_transition_probs(train_data):
    transition_probs = {}
    tag_pairs = {}
    for sentence in train_data:
        words = sentence.split()
        tags = [pair.split('/')[1] for pair in words]
        prev_tag = '<START>'
        for curr_tag in tags:
            if prev_tag not in transition_probs:
                transition_probs[prev_tag] = {}
            if curr_tag not in transition_probs[prev_tag]:
                transition_probs[prev_tag][curr_tag] = 0
            transition_probs[prev_tag][curr_tag] += 1

            if (prev_tag, curr_tag) not in tag_pairs:
                tag_pairs[(prev_tag, curr_tag)] = 0
            tag_pairs[(prev_tag, curr_tag)] += 1

            prev_tag = curr_tag
        curr_tag = '<END>'
        if prev_tag not in transition_probs:
            transition_probs[prev_tag] = {}
        if curr_tag not in transition_probs[prev_tag]:
            transition_probs[prev_tag][curr_tag] = 0
        transition_probs[prev_tag][curr_tag] += 1

        if (prev_tag, curr_tag) not in tag_pairs:
            tag_pairs[(prev_tag, curr_tag)] = 0
        tag_pairs[(prev_tag, curr_tag)] += 1

    for prev_tag in transition_probs:
        total = sum(transition_probs[prev_tag].values())
        for curr_tag in transition_probs[prev_tag]:
            transition_probs[prev_tag][curr_tag] /= total

    return transition_probs
```

```
def viterbi(sentence, emission_probs, transition_probs, tags):
    words = sentence.split()
    num_words = len(words)
    num_tags = len(tags)

    viterbi_table = np.zeros((num_words, num_tags))
    backpointer_table = [[None] * num_tags for _ in range(num_words)]

    for tag_idx, tag in enumerate(tags):
        word = words[0]
        if word in emission_probs[tag]:
            viterbi_table[0, tag_idx] = emission_probs[tag][word] * transition_probs['<START>'][tag]
        else:
            viterbi_table[0, tag_idx] = 1e-10 * transition_probs['<START>'][tag]

    for word_idx in range(1, num_words):
        word = words[word_idx]
        for tag_idx, tag in enumerate(tags):
            max_prob = None
            prev_tag = None
            for prev_tag_idx, prev_tag in enumerate(tags):
                emission_prob = emission_probs[tag].get(word, 1e-10)
                transition_prob = transition_probs[prev_tag].get(tag, 1e-10)
                prob = viterbi_table[word_idx-1, prev_tag_idx] * emission_prob * transition_prob
                if max_prob is None or prob > max_prob:
                    max_prob = prob
                    prev_tag = prev_tag
            viterbi_table[word_idx, tag_idx] = max_prob
            backpointer_table[word_idx][tag_idx] = prev_tag

    max_prob = None
    best_tag = None
    for tag_idx, tag in enumerate(tags):
        prob = viterbi_table[num_words-1, tag_idx] * transition_probs[tag]['<END>']
        if max_prob is None or prob > max_prob:
            max_prob = prob
            best_tag = tag

    tag_sequence = [best_tag]
    for word_idx in range(num_words-1, 0, -1):
        best_tag = backpointer_table[word_idx][tags.index(best_tag)]
        tag_sequence.insert(0, best_tag)

    return tag_sequence
```

```
def main():
    tags = ['NOUN', 'VERB', 'ADJ', 'ADV', 'PRON', 'DET', 'CONJ', 'PREP']
    word_tag_dict = {
        'NOUN': ['book', 'apple', 'tree', 'car', 'dog'],
        'VERB': ['read', 'eat', 'run', 'drive', 'bark'],
        'ADJ': ['red', 'big', 'small', 'fast', 'loud'],
        'ADV': ['quickly', 'slowly', 'quietly', 'carefully', 'happily'],
        'PRON': ['he', 'she', 'it', 'they', 'we'],
        'DET': ['the', 'a', 'an', 'this', 'that'],
        'CONJ': ['and', 'or', 'but', 'yet', 'so'],
        'PREP': ['in', 'on', 'at', 'by', 'with']
    }

    num_sentences = 1000
    max_sentence_length = 10
    data = generate_dataset(num_sentences, max_sentence_length, tags, word_tag_dict)

    train_data, test_data = split_data(data)
    emission_probs = calc_emission_probs(train_data)
    transition_probs = calc_transition_probs(train_data)
    for sentence in test_data:
        words = sentence.split()
        tag_sequence = viterbi(' '.join([word.split('/')[0] for word in words]), emission_probs, transition_probs, tags)
        print(f"Sentence: {' '.join(words)}")
        print(f"Tags: {' '.join(tag_sequence)}")
        print()

if __name__ == '__main__':
    main()
```

```
Sentence: but/CONJ
Tags: CONJ

Sentence: happily/ADV in/PREP drive/VERB run/VERB yet/CONJ happily/ADV tree/NOUN the/DET
Tags: PREP PREP PREP PREP PREP PREP PREP DET

Sentence: and/CONJ yet/CONJ
Tags: PREP CONJ

Sentence: read/VERB it/PRON quietly/ADV loud/ADJ the/DET it/PRON on/PREP quickly/ADV on/PREP apple/NOUN
Tags: PREP PREP PREP PREP PREP PREP PREP PREP PREP NOUN

Sentence: quickly/ADV book/NOUN
Tags: PREP NOUN

Sentence: loud/ADJ they/PRON quietly/ADV and/CONJ the/DET bark/VERB carefully/ADV at/PREP eat/VERB
Tags: PREP PREP PREP PREP PREP PREP PREP PREP VERB

Sentence: at/PREP
Tags: PREP

Sentence: happily/ADV book/NOUN this/DET by/PREP by/PREP at/PREP
Tags: PREP PREP PREP PREP PREP PREP

Sentence: on/PREP he/PRON an/DET an/DET drive/VERB and/CONJ red/ADJ and/CONJ in/PREP
Tags: PREP PREP PREP PREP PREP PREP PREP PREP PREP

Sentence: it/PRON she/PRON drive/VERB
Tags: PREP PREP VERB

Sentence: and/CONJ but/CONJ
Tags: PREP CONJ

Sentence: we/PRON by/PREP they/PRON and/CONJ quickly/ADV fast/ADJ
Tags: PREP PREP PREP PREP PREP ADJ

Sentence: they/PRON read/VERB slowly/ADV by/PREP it/PRON in/PREP apple/NOUN drive/VERB
Tags: PREP PREP PREP PREP PREP PREP PREP VERB

Sentence: slowly/ADV they/PRON loud/ADJ yet/CONJ drive/VERB
Tags: PREP PREP PREP PREP VERB

Sentence: in/PREP car/NOUN we/PRON apple/NOUN fast/ADJ big/ADJ he/PRON loud/ADJ at/PREP at/PREP
Tags: PREP PREP PREP PREP PREP PREP PREP PREP PREP PREP

Sentence: so/CONJ the/DET bark/VERB at/PREP quickly/ADV
Tags: PREP PREP PREP PREP ADV

Sentence: car/NOUN he/PRON but/CONJ yet/CONJ fast/ADJ eat/VERB
Tags: PREP PREP PREP PREP PREP VERB
```

```
Start coding or generate with AI.
```