HINDI POS TAGGER

By

N.S.S.L.SRI RAMYA ***421249***

V.PRAVALLIKA ***421270***

Under the guidance of

Dr. Hima Bindu

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGYANDHRA PRADESH

TADEPALLIGUDEM-534102, INDIA

APRIL-2023

HINDI POS TAGGER

Thesis submitted to

National Institute of Technology Andhra Pradesh

for the award of the degree

of

Bachelor of Technology

by

N.S.S.L.SRI RAMYA ***421249***

V.PRAVALLIKA ***421270***

Under the guidance of

Dr. Hima Bindu

DEPARTMENT COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGYANDHRA PRADESH

TADEPALLIGUDEM-534102, INDIA

APRIL-2023

DECLARATION

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

(Signature)

N.S.S.L. SRI RAMYA

421249

Date: 02/05/2023

(Signature)

V.PRAVALLIKA

421270

Date: 02/05/2023

CERTIFICATE

It is certified that the work contained in the thesis titled "**Title of the Thesis**," by "N.S.S.L. Sri Ramya & V. Pravallika, bearing Roll No:421249 & 421270" has been carried out under my/our supervision and that this work has not been submitted elsewhere for a degree.

Signature of Supervisor(s)
Dr. HIMA BINDU
CSE
N.I.T. Andhra Pradesh
MAY,2023

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

POS tagging is a task of labelling each word in a sentence with its appropriate part of speech.It is an important natural language processing application used in machine translation, word sense disambiguation, question answering parsing, and so on. The genesis of POS tagging is based on the ambiguity of many words in terms of their part of speech in a context.

Steps in POS Tagging:

1) We first Obtain a tagged data set for Training.

2)Then Hidden Markov Model is used to identify Transmission and Emission Probabilities
3)Apply Viterbi Algorithm on Testing Data Set

4)Output the tagging sequence with the highest probability

How it Works:

The user inputs a Hindi Sentence in the space provided in the website. The sentence will be decomposed to its respective parts of speech and an output is obtained. This output sentence contains the correct parts of speech tagging of the given sentence. The user can change the context and input another sentence then parts of speech will displayed accordingly.

TABLE OF CONTENTS

NIT Andhra Pradesh

1. Introduction

Part-of-speech tagging is the process of assigning a part-of-speech to each word in part-of-speech tagging a text. The input is a sequence x1, x2,..., xn of (tokenized) words and a tag set, and the output is a sequence y1, y2,..., yn of tags, each output yi corresponding exactly to one input xi.

Tagging is a disambiguation task; words are ambiguous —have more than one possible part-of-speech—and the goal is to find the correct tag for the situation.

Example:

I threw the ***stone*** across the lake. In this sentence the word stone acts as ***Noun.***

The ***stone*** wall was hard to climb. In this sentence the word stone acts as ***Adjective***.

The POS tagging model must be able to give the most probable output with no ambiguity.

We use Hidden Markov Model and apply Viterbi algorithm to make the model more efficient in terms of run time.

Steps in POS Tagging:

- **Collect a dataset of annotated text:** This dataset will be used to train and test the POS tagger. The text should be annotated with the correct POS tags for each word.
- **Preprocess the text:** This may include tasks such as tokenization (splitting the text into individual words), lowercasing, and removing punctuation.

- **Divide the dataset into training and testing sets:** The training set will be used to train the POS tagger, and the testing set will be used to evaluate its performance.

- **Train the POS tagger:** This may involve building a statistical model, such as a hidden Markov model (HMM), or defining a set of rules for a rule-based or transformation-based tagger. The model or rules will be trained on the annotated text in the training set.
- **Test the POS tagger:** Use the trained model or rules to predict the POS tags of the words in the testing set. Compare the predicted tags to the true tags and calculate metrics such as precision and recall to evaluate the performance of the tagger.
- **Fine-tune the POS tagger:** If the performance of the tagger is not satisfactory, adjust the model or rules and repeat the training and testing process until the desired level of accuracy is achieved.
- **Use the POS tagger:** Once the tagger is trained and tested, it can be used to perform POS tagging on new, unseen text. This may involve preprocessing the text and inputting it into the trained model or applying the rules to the text. The output will be the predicted POS tags for each word in the text.

Back in the days, the POS annotation was manually done by human annotators but being such a laborious task, today we have automatic tools that are capable of tagging each word with an appropriate POS tag within a context.

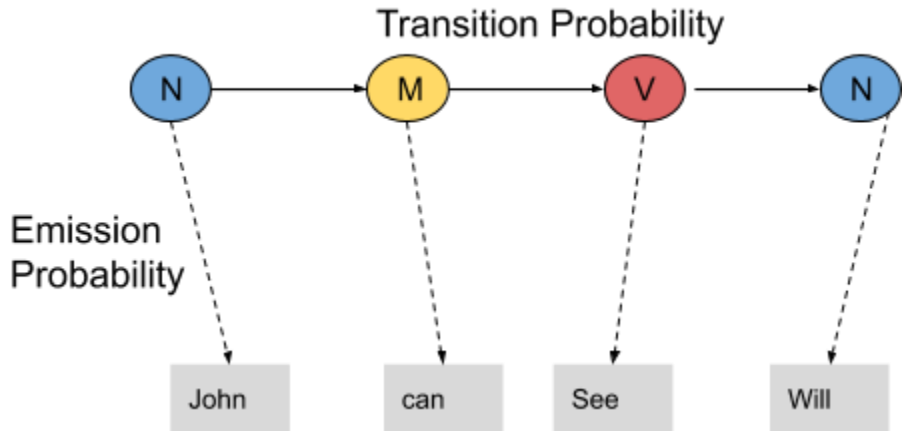
Nowadays, manual annotation is typically used to annotate a small corpus to be used as training data for the development of a new automatic POS tagger. Annotating modern multi-billion-word corpora manually is unrealistic and automatic tagging is used instead.

POS tags give a large amount of information about a word and its neighbors. Their applications can be found in various tasks such as information retrieval, parsing, Text to Speech (TTS) applications, information extraction, linguistic research for corpora. They are also used as an intermediate step for higher-level NLP tasks such as parsing, semantics analysis, translation, and many more, which makes POS tagging a necessary function for advanced NLP applications.

2. POS tagging with Hidden Markov Model

HMM (Hidden Markov Model) is a Stochastic technique for POS tagging. Hidden Markov models are known for their applications to reinforcement learning and temporal pattern recognition such as speech, handwriting, gesture recognition, musical score following, partial discharges, and bioinformatics.

Let us consider an example proposed by Dr.Luis Serrano and find out how HMM selects an appropriate tag sequence for a sentence.



In this example, we consider only 3 POS tags that are noun, model and verb. Let the sentence " Ted will spot Will " be tagged as noun, model, verb and a noun and to calculate the probability associated with this particular sequence of tags we require their Transition probability and Emission probability.

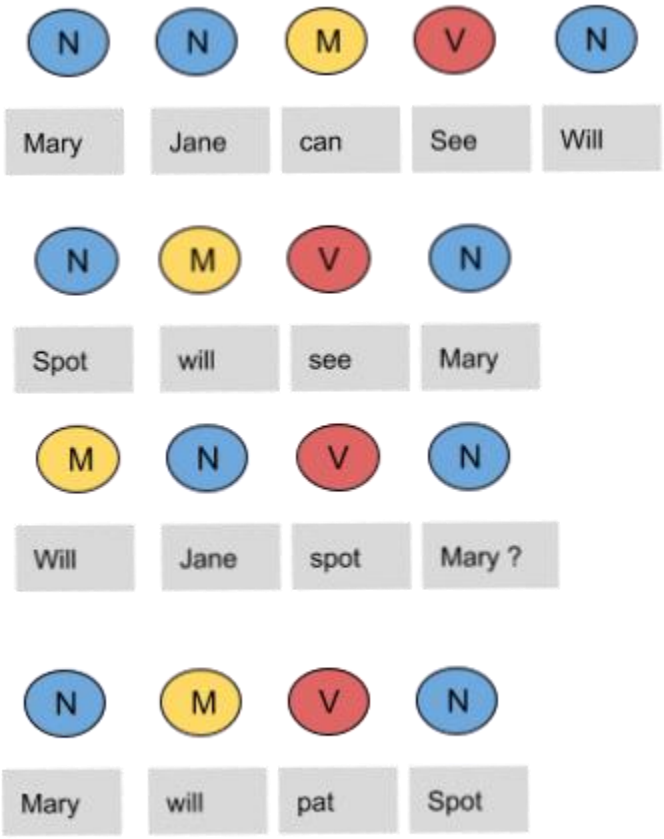
The transition probability is the likelihood of a particular sequence for example, how likely is that a noun is followed by a model and a model by a verb and a verb by a noun. This probability is known as Transition probability. It should be high for a particular sequence to be correct.

Now, what is the probability that the word Ted is a noun, will is a model, spot is a verb and Will is a noun. These sets of probabilities are Emission probabilities and should be high for our tagging to be likely.

Let us calculate the above two probabilities for the set of sentences below

- Mary Jane can see Will
- Spot will see Mary
- Will Jane spot Mary?
- Mary will pat Spot

Note that Mary Jane, Spot, and Will are all names.



In the above sentences, the word Mary appears four times as a noun. To calculate the emission probabilities, let us create a counting table in a similar manner.

Words	Noun	Model	Verb
Mary	4	0	0
Jane	2	0	0
Will	1	3	0
Spot	2	0	1
Can	0	1	0
See	0	0	2
pat	0	0	1

Now let us divide each column by the total number of their appearances for example, ‘noun’ appears nine times in the above sentences so divide each term by 9 in the noun column. We get the following table after this operation

Words	Noun	Model	Verb
Mary	4/9	0	0
Jane	2/9	0	0
Will	1/9	3/4	0
Spot	2/9	0	1/4
Can	0	1/4	0
See	0	0	2/4
pat	0	0	1

From the above table, we infer that

The probability that Mary is Noun = 4/9

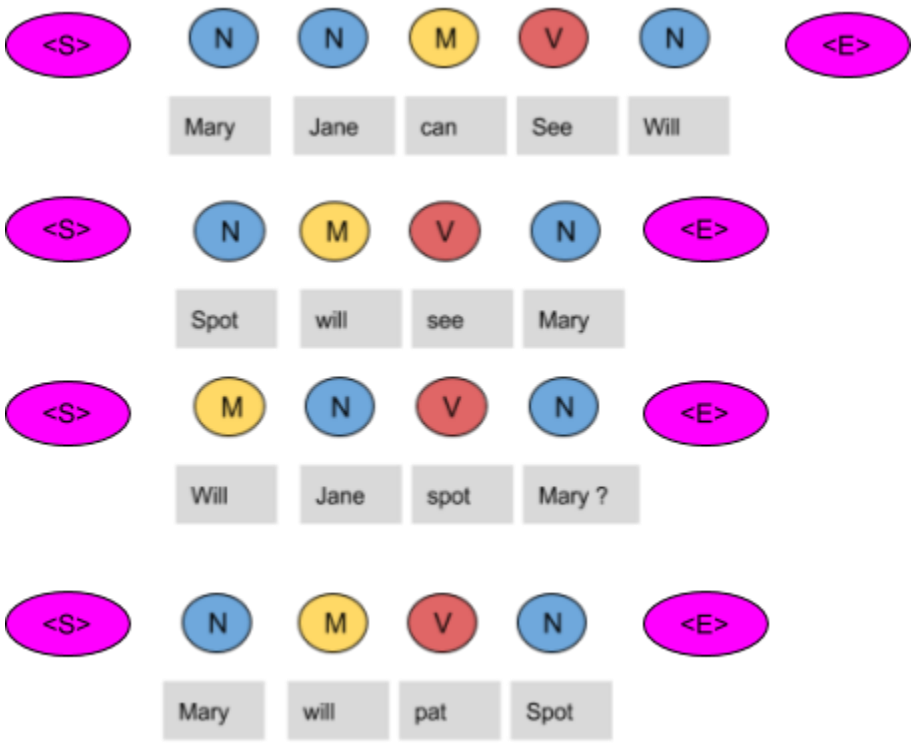
The probability that Mary is Model = 0

The probability that Will is Noun = 1/9

The probability that Will is Model = 3/4

In a similar manner, you can figure out the rest of the probabilities. These are the emission probabilities.

Next, we have to calculate the transition probabilities, so define two more tags <S> and <E>. <S> is placed at the beginning of each sentence and <E> at the end as shown in the figure below.

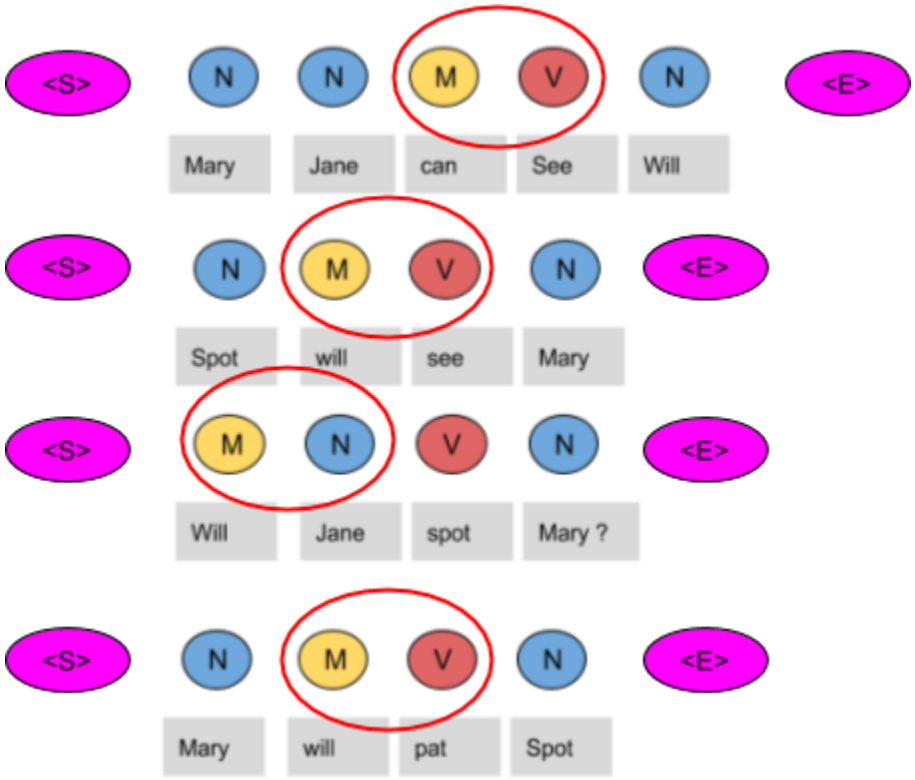


Let us again create a table and fill it with the co-occurrence counts of the tags.

	N	M	V	<E>
<S>	3	1	0	0
N	1	3	1	4
M	1	0	3	0
V	4	0	0	0

In the above figure, we can see that the <S> tag is followed by the N tag three times, thus the first entry is 3.The model tag follows the <S> just once, thus the second entry is 1. In a similar manner, the rest of the table is filled.

Next, we divide each term in a row of the table by the total number of co-occurrences of the tag in consideration, for example, The Model tag is followed by any other tag four times as shown below, thus we divide each element in the third row by four.



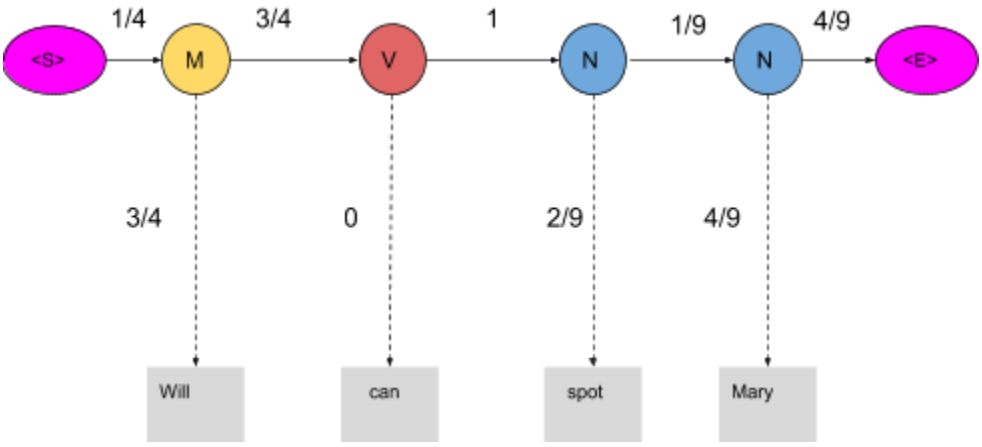
	N	M	V	<E>
<S>	3/4	1/4	0	0
N	1/9	3/9	1/9	4/9
M	1/4	0	3/4	0
V	4/4	0	0	0

These are the respective transition probabilities for the above four sentences. Now how does the HMM determine the appropriate sequence of tags for a particular sentence from the above tables? Let us find it out.

Take a new sentence and tag them with wrong tags. Let the sentence, ‘ Will can spot Mary’ be tagged as-

- Will as a model
- Can as a verb
- Spot as a noun
- Mary as a noun

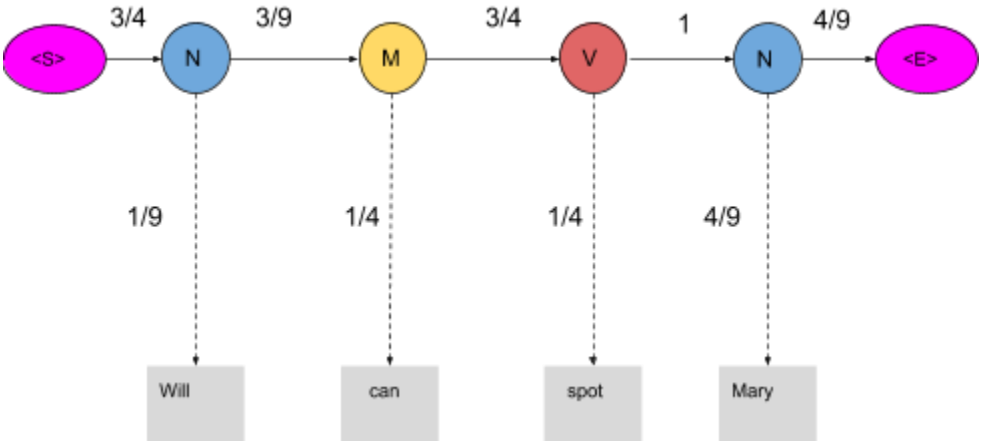
Now calculate the probability of this sequence being correct in the following manner.



The probability of the tag Model (M) comes after the tag <S> is ¼ as seen in the table. Also, the probability that the word Will is a Model is 3/4. In the same manner, we calculate each and every probability in the graph. Now the product of these probabilities is the likelihood that this sequence is right. Since the tags are not correct, the product is zero.

1/4*3/4*3/4*0*1*2/9*1/9*4/9*4/9=0

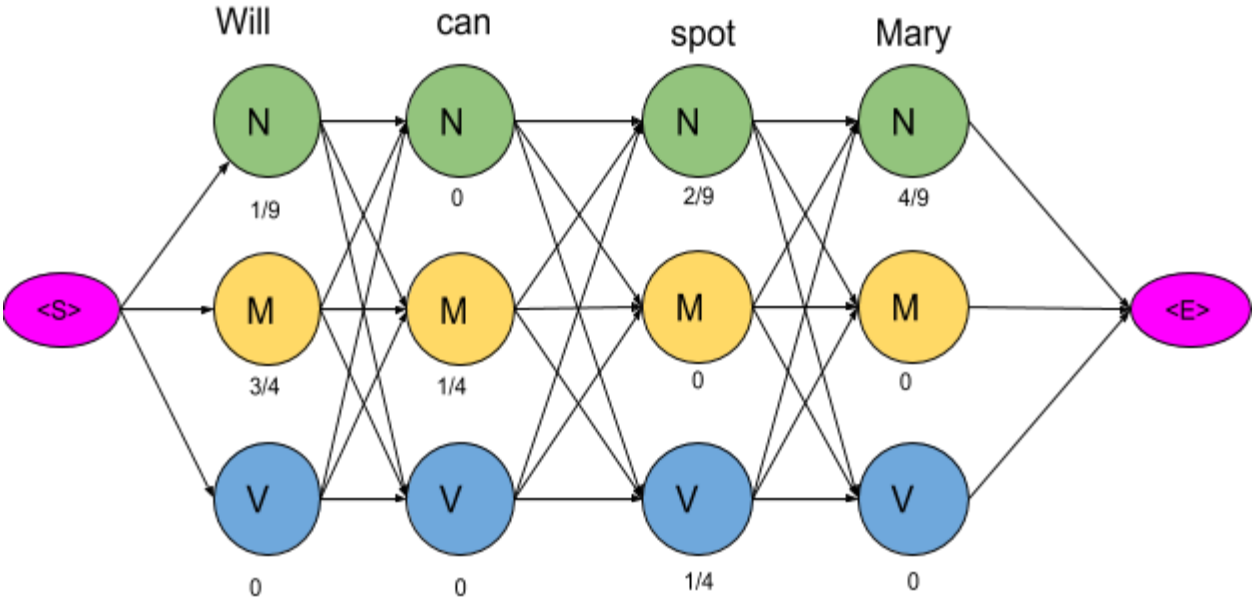
When these words are correctly tagged, we get a probability greater than zero as shown below



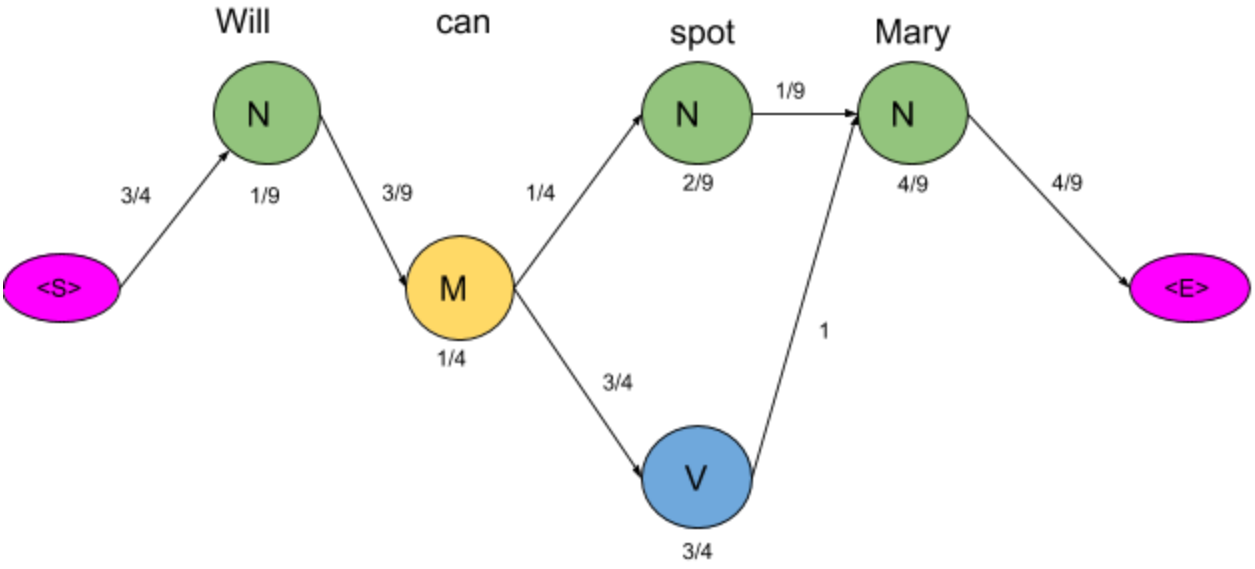
Calculating the product of these terms we get,

3/4*1/9*3/9*1/4*3/4*1/4*1*4/9*4/9=0.00025720164

For our example, keeping into consideration just three POS tags we have mentioned, 81 different combinations of tags can be formed. In this case, calculating the probabilities of all 81 combinations seems achievable. But when the task is to tag a larger sentence and all the POS tags in the Penn Treebank project are taken into consideration, the number of possible combinations grows exponentially and this task seems impossible to achieve. Now let us visualize these 81 combinations as paths and using the transition and emission probability mark each vertex and edge as shown below.



The next step is to delete all the vertices and edges with probability zero, also the vertices which do not lead to the endpoint are removed. Also, we will mention-



Now there are only two paths that lead to the end, let us calculate the probability associated with each path.

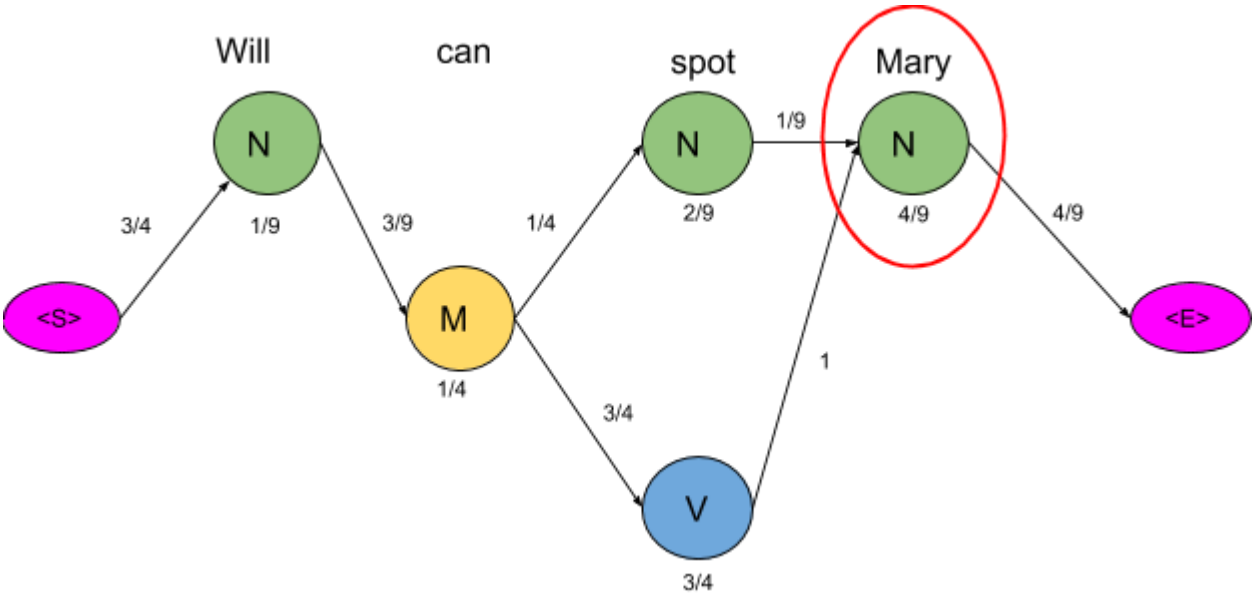
<S>→N→M→N→N→<E> =**3/4*1/9*3/9*1/4*1/4*2/9*1/9*4/9*4/9=0.00000846754**

<S>→N→M→N→V→<E>=**3/4*1/9*3/9*1/4*3/4*1/4*1*4/9*4/9=0.00025720164**

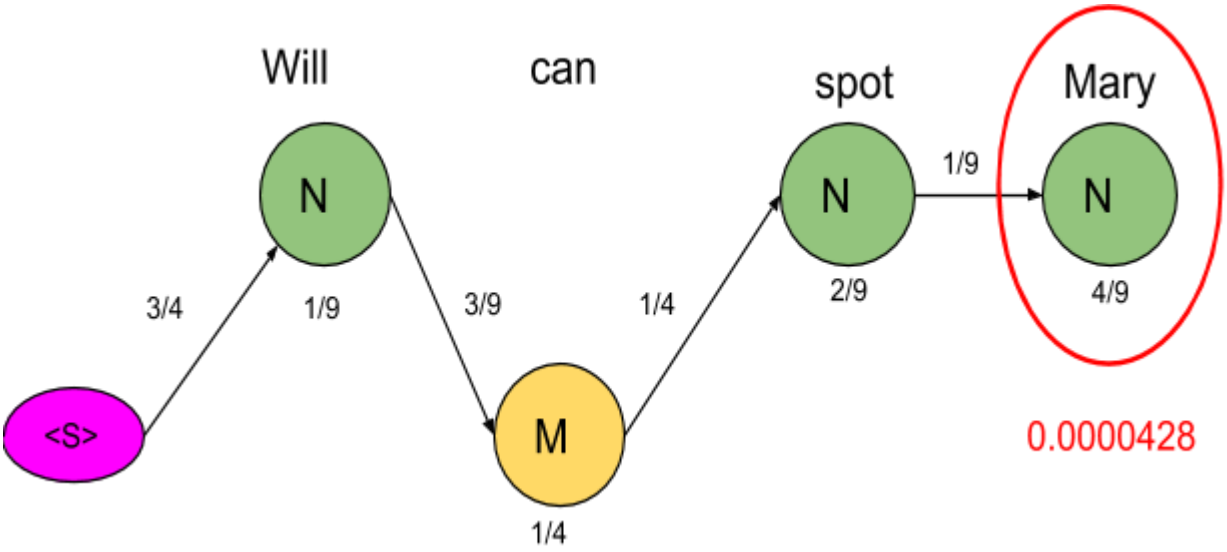
Clearly, the probability of the second sequence is much higher and hence the HMM is going to tag each word in the sentence according to this sequence.

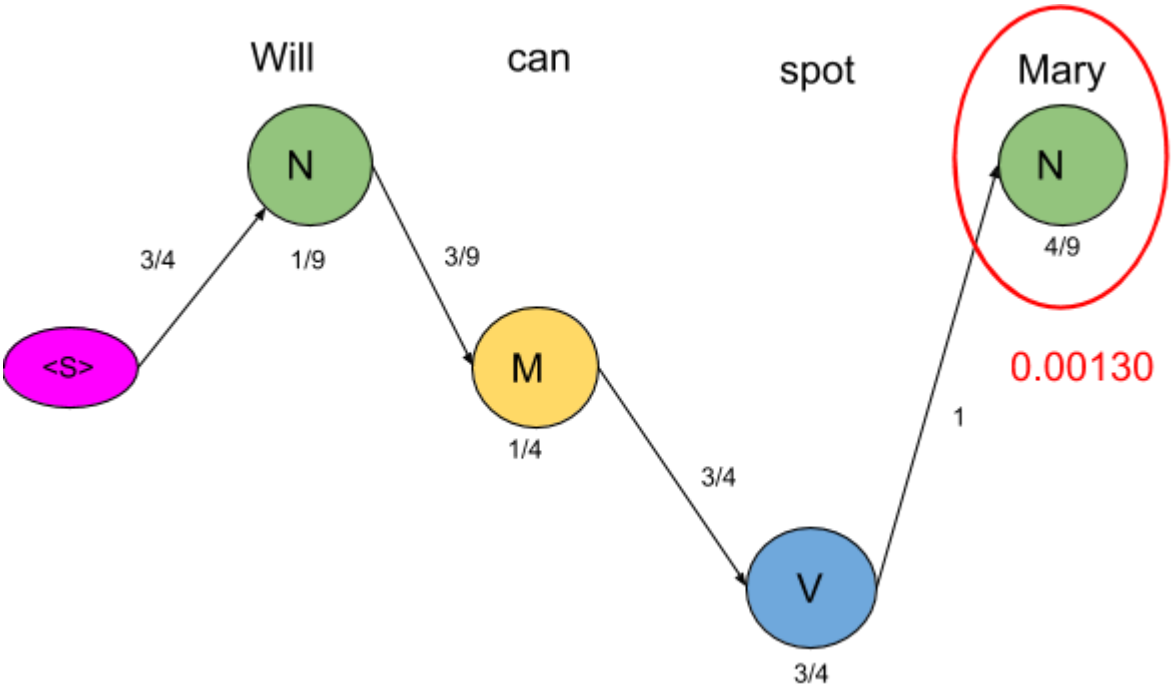
3.Optimizing HMM with Viterbi Algorithm

In the previous section, we optimized the HMM and brought our calculations down from 81 to just two. Now we are going to further optimize the HMM by using the Viterbi algorithm. Let us use the same example we used before and apply the Viterbi algorithm to it.

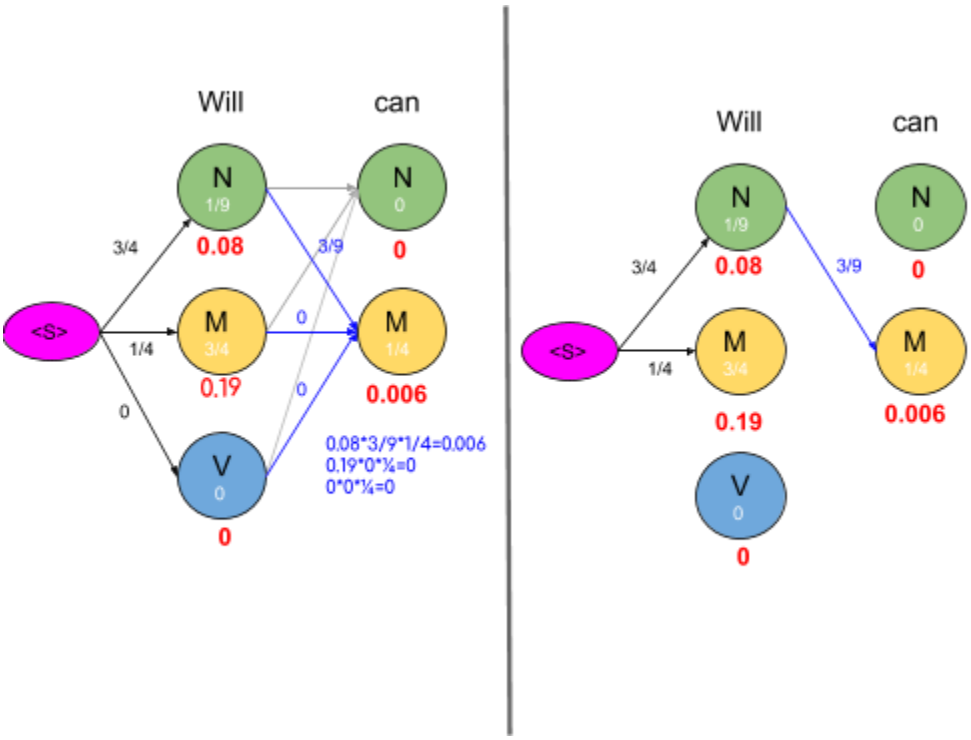


Consider the vertex encircled in the above example. There are two paths leading to this vertex as shown below along with the probabilities of the two mini-paths.

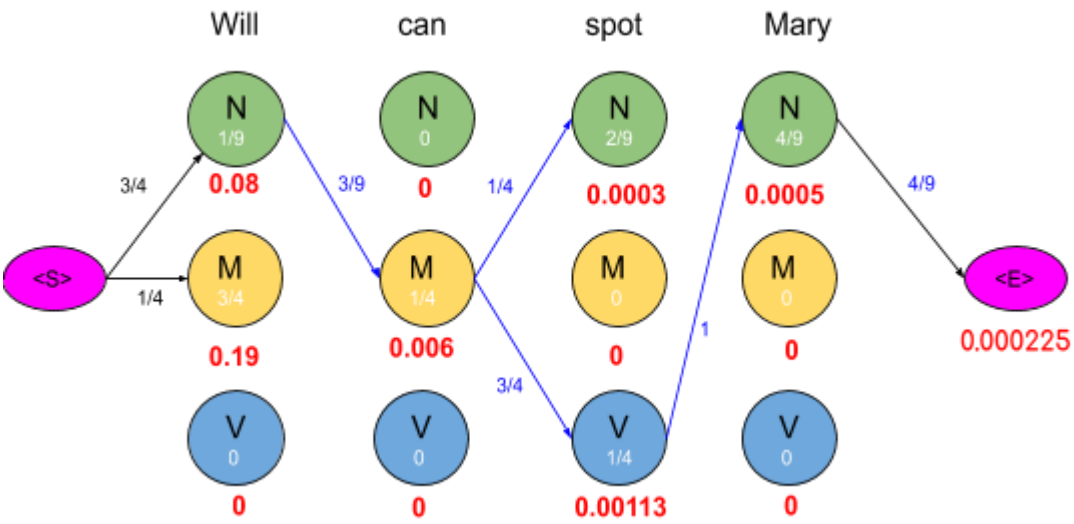




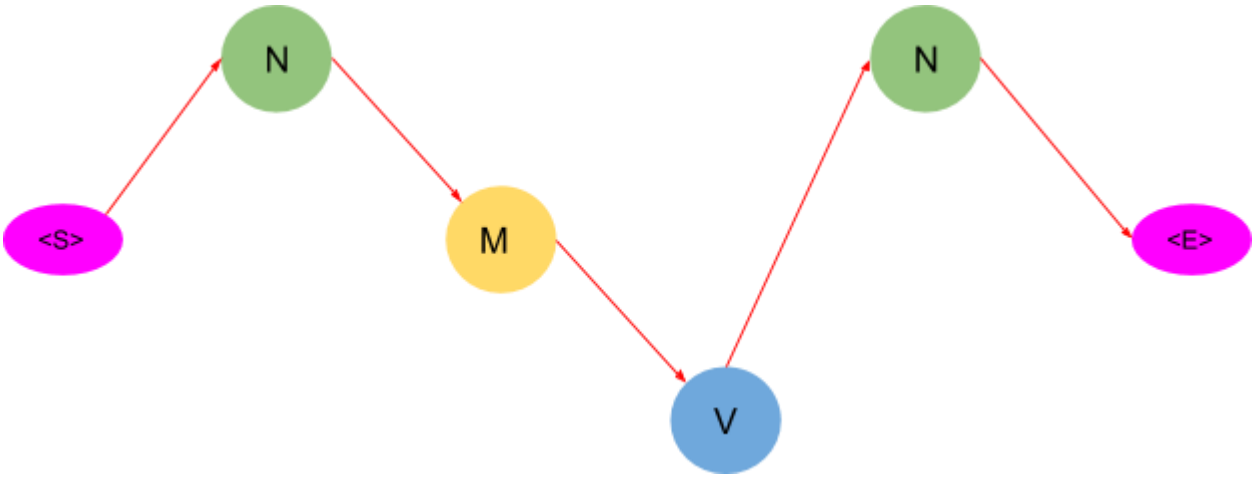
Now we are really concerned with the mini path having the lowest probability. The same procedure is done for all the states in the graph as shown in the figure below.



As we can see in the figure above, the probabilities of all paths leading to a node are calculated and we remove the edges or path which has lower probability cost. Also, you may notice some nodes having the probability of zero and such nodes have no edges attached to them as all the paths are having zero probability. The graph obtained after computing probabilities of all paths leading to a node is shown below:



To get an optimal path, we start from the end and trace backward, since each state has only one incoming edge, This gives us a path as shown below



As you may have noticed, this algorithm returns only one path as compared to the previous method which suggested two paths. Thus by using this algorithm, we saved us a lot of computations.

After applying the Viterbi algorithm the model tags the sentence as following-

- Will as a noun
- Can as a model
- Spot as a verb
- Mary as a noun

These are the right tags so we conclude that the model can successfully tag the words with their appropriate POS tags.

4. Experimental Procedure

Initially a tagged dataset is collected. It contains all the sentences with their corresponding tags.

Now we take any sentence as input from the website and we tag the sentence using veterbi algorithm and we print the output i.e the tagged sentence on the screen.

```
import sys
import math
from decimal import *
import codecs

tag_list = set()
tag_count = {}
word_set = set()


def parse_traindata():
    fin = "train_data/train_data.txt"
    output_file = "model/hmmmodel.txt"
    wordtag_list = []

    try:
        # read the training data file #
        input_file = codecs.open(fin, mode = 'r', encoding="utf-8")
        lines = input_file.readlines()
        # pushing words of a line into a list #
        for line in lines:
            line = line.strip('\n')
            data = line.split(" ")
            wordtag_list.append(data)
```

```
        input_file.close()
        return wordtag_list
    except IOError:
        fo = codecs.open(output_file,mode = 'w',encoding="utf-8")
        fo.write("File not found: {}".format(fin))
        fo.close()
        sys.exit()
def transition_count(train_data):
    global tag_list
    global word_set
    transition_dict = {}
    global tag_count
    for value in train_data:
        previous = "start"
        for data in value:
            # we store words and their corresponding tags #
            i = data[::-1]
            word = data[:-i.find("/") - 1]
            word_set.add(word.lower())
            data = data.split("/")
            tag = data[-1]
            tag_list.add(tag)

            # store frequency of each tag #

            if tag in tag_count:
                tag_count[tag] += 1
            else:
                tag_count[tag] = 1
```

```
            # store the frequency of each combination of tags #

            if (previous + "~tag~" + tag) in transition_dict:
                transition_dict[previous + "~tag~" + tag] += 1
                previous = tag
            else:
                transition_dict[previous + "~tag~" + tag] = 1
                previous = tag

    return transition_dict
def transition_probability(train_data):
    count_dict = transition_count(train_data)
    prob_dict = {}
    for key in count_dict:
        den = 0
        val = key.split("~tag~")[0]



      # Probabilty of a tagA to be followed by tagB out of all possible tags #
        for key_2 in count_dict:
            if key_2.split("~tag~")[0] == val:
                den += count_dict[key_2]
        prob_dict[key] = Decimal(count_dict[key])/(den)
    return prob_dict
def transition_smoothing(train_data):
    transition_prob = transition_probability(train_data)
    for tag in tag_list:
        # if a tag does not occur as a start tag, then set its probability to be a start tag to minimum
value #
```

```
        if "start" + tag not in  transition_prob:
            transition_prob[("start" + "~tag~" + tag)] = Decimal(1) / Decimal(len(word_set) +
tag_count[tag])
    for tag1 in tag_list:
        for tag2 in tag_list:
            # if a particular tag combination does not exist in the dictionary, we set its probability to
minimum#
            if (tag1 +"~tag~" + tag2) not in transition_prob:
                transition_prob[(tag1+"~tag~"+tag2)] = Decimal(1)/Decimal(len(word_set) +
tag_count[tag1])
    return transition_prob

def emission_count(train_data):
    count_word = {}
    for value in train_data:
        for data in value:
            i = data[::-1]
            word = data[:-i.find("/") - 1]
            tag = data.split("/")[-1]
            # map the words in the training set to their tagged POS #
            if word.lower() + "/" + tag in count_word:
                count_word[word.lower() + "/" + tag] +=1
            else:
                count_word[word.lower() + "/" + tag] = 1
    return count_word


def emission_probability(train_data):
    global tag_count
```

```
    word_count = emission_count(train_data)
    emission_prob_dict = {}

    # calculate probability of a word to be a certain Tag out of all the possible tags that it can be #
    for key in word_count:
        emission_prob_dict[key] = Decimal(word_count[key])/tag_count[key.split("/")[-1]]
    return emission_prob_dict
train_data = parse_traindata()
transition_model = transition_smoothing(train_data)
emission_model = emission_probability(train_data)

fout = codecs.open("model/hmmmodel.txt", mode ='w', encoding="utf-8")
for key, value in transition_model.items():
    fout.write('%s:%s\n' % (key, value))

fout.write(u'Emission Model\n')
for key, value in emission_model.items():
    fout.write('%s:%s\n' % (key, value))
```

The code for testing the model is:

```
import sys
from decimal import *
import codecs
from flask import Flask, jsonify, request
```

```
from flask_cors import CORS
app = Flask(__name__)
CORS(app)
@app.route('/')
def verify():
    # data = request.get_json()
    # l = sorts(data)
    return jsonify({"name": "verfied ok"})


tag_set = set()
word_set = set()


def parse_traindata():
    fin = "model/hmmmodel.txt"
    output_file = "outputs/prav.txt"
    transition_prob = {}
    emission_prob = {}
    tag_list = []
    tag_count = {}
    global tag_set
    try:
        input_file = codecs.open(fin, mode='r', encoding="utf-8")
        lines = input_file.readlines()
        flag = False
        for line in lines:
            line = line.strip('\n')
            if line != "Emission Model":
```

```
            i = line[::-1]
            key_insert = line[:-i.find(":")-1]
            value_insert = line.split(":")[-1]

            # for transition probabilities #
            if flag == False:
                transition_prob[key_insert] = value_insert

                if (key_insert.split("~tag~")[0] not in tag_list) and (key_insert.split("~tag~")[0] !=
"start"):
                    tag_list.append(key_insert.split("~tag~")[0])
                    # print(key_insert.split("~tag~")[0])

            else:
                # for emission probabilities #
                emission_prob[key_insert] = value_insert
                val = key_insert.split("/")[-1]
                j = key_insert[::-1]
                word = key_insert[:-j.find("/")-1].lower()
                word_set.add(word)
                if val in tag_count:
                    tag_count[val] += 1
                else:
                    tag_count[val] = 1
                tag_set.add(val)

        else:
            flag = True
            continue
```

```
        input_file.close()
        return tag_list, transition_prob, emission_prob, tag_count, word_set


    except IOError:
        fo = codecs.open(output_file, mode='w', encoding="utf-8")
        fo.write("File not found: {}".format(fin))
        fo.close()
        sys.exit()
def viterbi_algorithm(sentence, tag_list, transition_prob, emission_prob, tag_count, word_set):
    global tag_set
    # Get words from each sentence #
    sentence = sentence.strip("\n")
    word_list = sentence.split(" ")
    current_prob = {}
    for tag in tag_list:
        # transition probability #
        tp = Decimal(0)
        # Emission probability #
        em = Decimal(0)
        # Storing the probability of every tag to be starting tag #
        if "start~tag~"+tag in transition_prob:
            tp = Decimal(transition_prob["start~tag~"+tag])
        # Check for word in training data. If present, check the probability of the first word to be of
given tag#
        if word_list[0].lower() in word_set:
            if (word_list[0].lower()+"/"+tag) in emission_prob:
                em = Decimal(emission_prob[word_list[0].lower()+"/"+tag])
                # Storing probability of current combination of tp and em #
```

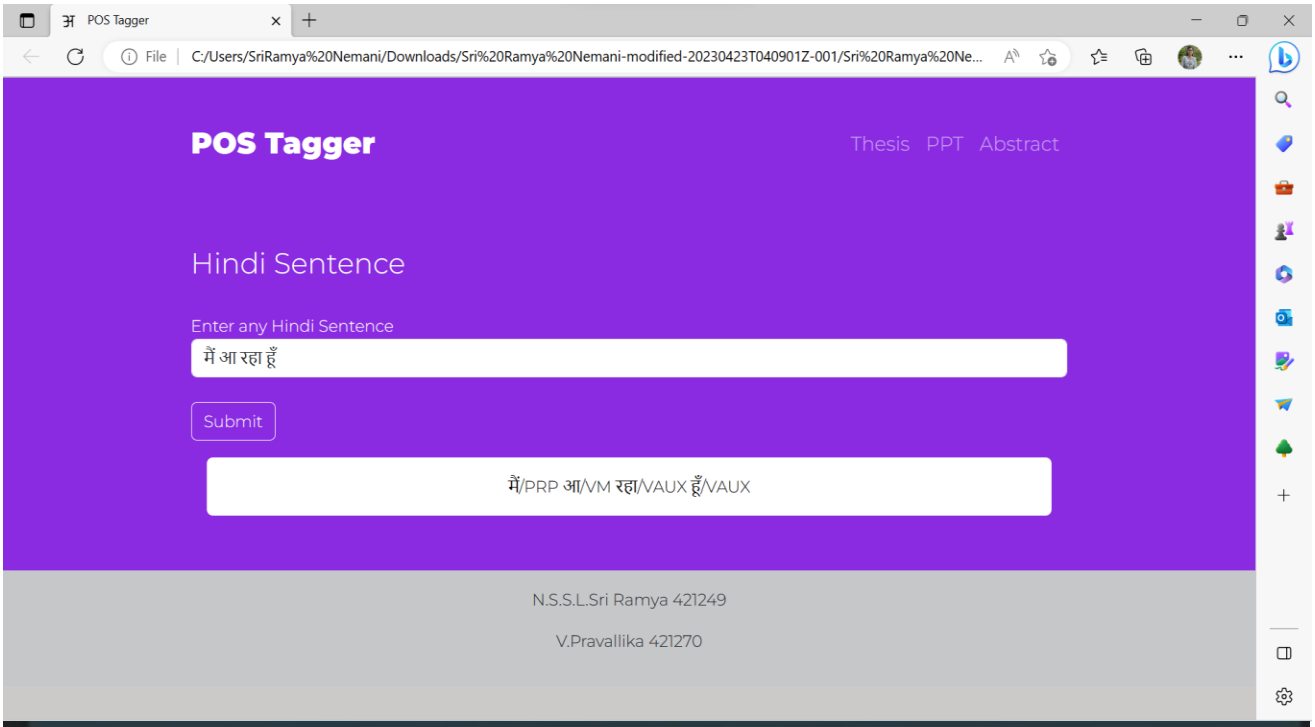
```
                current_prob[tag] = tp * em
            # Check for word in training data. If absent then probability is just tp#
            else:
                em = Decimal(1) / (tag_count[tag] + len(word_set))
                current_prob[tag] = tp


        if len(word_list) == 1:
            # Return max path if only one word in sentence #
            max_path = max(current_prob, key=current_prob.get)
            return max_path
        else:
            # Tracking from second word to last word #
            for i in range(1, len(word_list)):
                previous_prob = current_prob
                current_prob = {}
                locals()['dict{}'.format(i)] = {}
                previous_tag = ""
                for tag in tag_list:
                    if word_list[i].lower() in word_set:
                        if word_list[i].lower()+"/"+tag in emission_prob:
                            em = Decimal(
                                emission_prob[word_list[i].lower()+"/"+tag])
                            # Find the maximum probability using previous node's(tp*em)[i.e probability of
reaching to the previous node] * tp * em (Bigram Model) #
                            max_prob, previous_state = max((Decimal(previous_prob[previous_tag]) *
Decimal(
                                transition_prob[previous_tag + "~tag~" + tag]) * em, previous_tag) for
previous_tag in previous_prob)
                            current_prob[tag] = max_prob
```

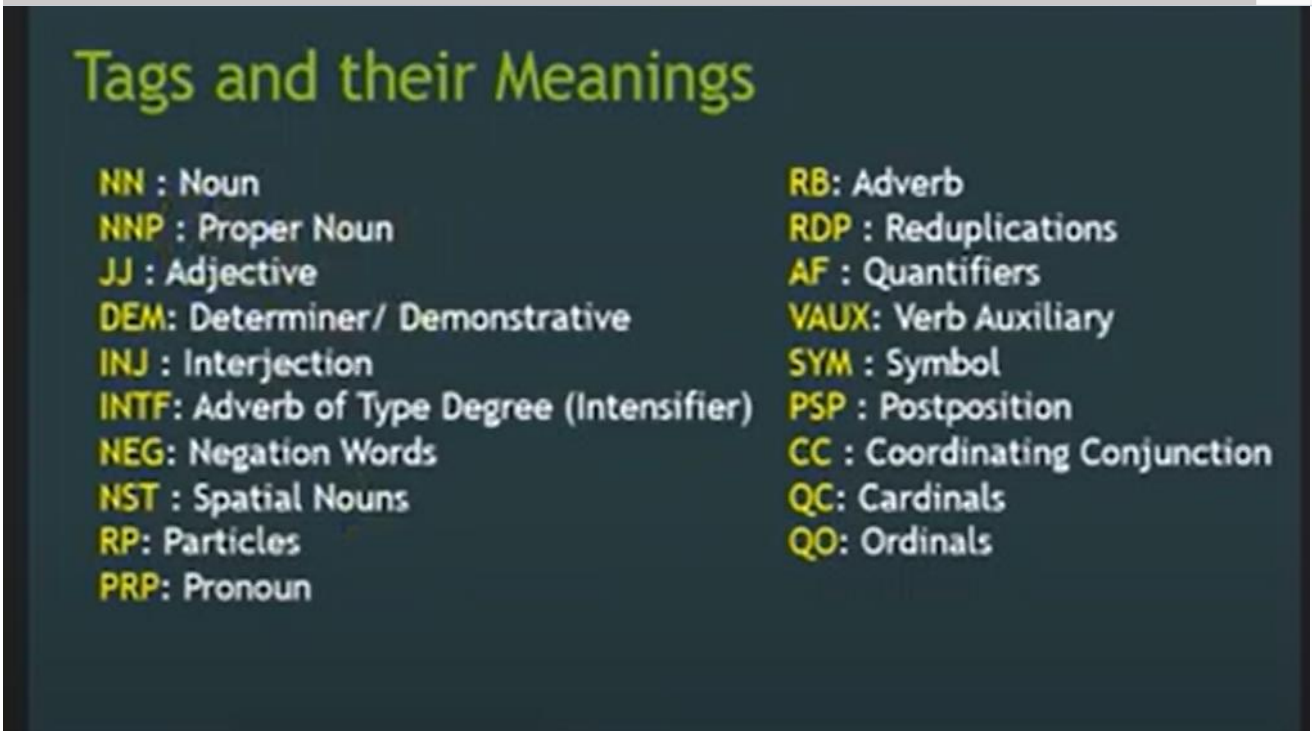
```
                locals()['dict{}'.format(i)
                        ][previous_state + "~" + tag] = max_prob
                previous_tag = previous_state
            else:
                em = Decimal(1) / (tag_count[tag] + len(word_set))
                max_prob, previous_state = max((Decimal(previous_prob[previous_tag]) *
Decimal(
                    transition_prob[previous_tag+"~tag~"+tag]) * em, previous_tag) for previous_tag
in previous_prob)
                current_prob[tag] = max_prob
                locals()['dict{}'.format(i)
                        ][previous_state + "~" + tag] = max_prob
                previous_tag = previous_state

        # if last word of sentence, then return path dicts of all words #
        if i == len(word_list)-1:
            max_path = ""
            last_tag = max(current_prob, key=current_prob.get)
            max_path = max_path + last_tag + " " + previous_tag
            for j in range(len(word_list)-1, 0, -1):
                for key in locals()['dict{}'.format(j)]:
                    data = key.split("~")
                    if data[-1] == previous_tag:
                        max_path = max_path + " " + data[0]
                        previous_tag = data[0]
                        break
            result = max_path.split()
            result.reverse()
            return " ".join(result)
```

```
tag_list, transition_model, emission_model, tag_count, word_set = parse_traindata()


@app.route('/out', methods=['POST'])
def sort():
    data = request.get_json()
    so = data["name"]

    j = str(so)

    filr = open("test_data/ab.txt", "w", encoding="utf-8")
    filr.write(j)
    filr = open("test_data/ab.txt", "r", encoding="utf-8")
    fin = sys.argv[1] if len(sys.argv) > 1 else "test_data/ab.txt"
    input_file = codecs.open(fin, mode='r', encoding="utf-8")
    fout = codecs.open("outputs/prav.txt", mode='w', encoding="utf-8")
    for sentence in input_file.readlines():
        path = viterbi_algorithm(
            sentence, tag_list, transition_model, emission_model, tag_count, word_set)
        sentence = sentence.strip("\n")
        word = sentence.split(" ")
        tag = path.split(" ")
        for j in range(0, len(word)):
            if j == len(word)-1:
                fout.write(word[j] + "/" + tag[j] + u'\n')
            else:
                fout.write(word[j] + "/" + tag[j] + " ")

    fout = codecs.open("outputs/prav.txt", mode='r', encoding="utf-8")
```

```
    a = str(fout.readlines())
    return jsonify({"name": a})


app.run(debug=True)
# predicted = codecs.open("outputs/prav.txt", mode ='r', encoding="utf-8")
# expected = codecs.open("test_data/test_tagged.txt", mode ='r', encoding="utf-8")

# c = 0
# total = 0
# for line in predicted.readlines():
#  u = line.split(" ")
# total += len(u)
#   a = expected.readline().split(" ")
#   for i in range(len(u)):
#       if(a[i]!=u[i]):
# c+=1

# print("Wrong Predictions = ",c)
# print("Total Predictions = ",total)
# print("Accuracy is = ",100 - (c/total * 100),"%")
```

5. Results and Discussion





6. Conclusion and future scope

The size of dataset considered for the experiments are taken randomly and there is no specific dependency on the size of training set.

As we can see, the Modifications along with hard coded rules gives a considerable improvement in accuracy even for such a small set of training data. A model trained with small dataset cannot predict more accurately because of low accuracy of transition probabilities. The increase in size of training set will definitely increase the accuracy as it will more accurately capture the transition probabilities. However our attempt is to get the optimum accuracy with least amount of resources.

7. References

https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf

Speech and Language Processing -Third Edition draft by Daniel Jurafsky & James H. Martin

Chapter 8 (page no : 160 to 174)

https://ieeexplore.ieee.org/document/9377146

Authors: Diganta Baishya & Rupam Baruah

conclusion

https://www.mygreatlearning.com/blog/pos-tagging/

Authors: Great Learning Team

- Techniques for POS tagging
- POS tagging with Hidden Markov Model
- Optimizing HMM with Viterbi Algorithm