STOCHASTIC PROCESS

[Final Report]

DEC **2022**

WORD PREDICTION

Markov Chain

PRESENTED BY

Ebtssam Hassan	20201491232
Rahma Ezzat	20201381362
Aya Mohammed	20201379022
Asmaa Hamdy	20201032790
Ziad Ahmed	20201376238
Abdelrahman salah	20201444840
Omar Mekkawy	20201375851

PRESENTED TO

Dr. Amr Amin

Markov Model

a process where the next state depends only on the current state.
For example, let's say that tomorrow's weather depends only on today's weather or today's stock price depends only on yesterday's stock price. Mathematically speaking, the conditional probability distribution of the following state depends on the current state and not the past states. That is s(t) depends only on s(t-1), where s(t) is the state at time t. This is what is called the first-order Markov model.
In general, if the current state of a system depends on n previous states, it is called the n-th order Markov model.

Next Word Prediction

- *Introduction*

The next word predictor is a keyboard-based project, it will be helpful for the intermediate level student who is sending mail to the administration, or teacher for some time where they need to assure there is no wrong spelling or something in a research paper, they also need to next word predictor software. Sometimes they forget what to say, this next word predictor will help them. It is based on Markov chains.
Markov chain maintains a sequence of possible even where prediction or probabilities for the next stage are based solely on its previous event stage, not the state before.

Now let's take our understanding of the Markov model and do something interesting. Suppose we want to build a system that when given an incomplete sentence, the system tries to predict the next word in the sentence. So, how do we take a word prediction case as in this one and model it as a Markov model problem? Treat every word as a state and predict the next word based on the previous state, as simple as that. This case is a perfect fit for the Markov chain.

We need the probability distribution. Let's understand this better with a simple example. Consider the three simple sentences :

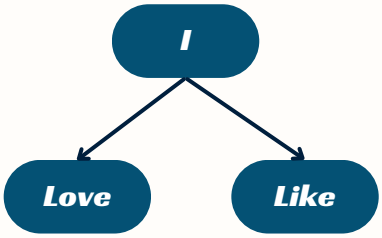
- ***I like Photography.***
- ***I like Science.***
- ***I love Mathematics.***

All the unique words from the above sentences are 'I', 'like', 'love', 'Photography', 'Science', and 'Mathematics' could form different states.

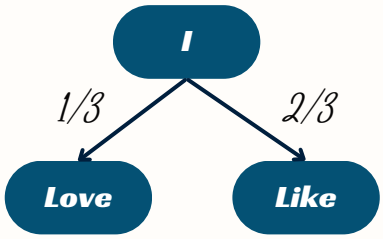
The probability distribution is all about determining the probability of transition from one state to another, in our case, it is from one word to another.

In our scenario, it is clear from the above examples that the first word always starts out with the word 'I'. So there is a 100% chance that the first word of the sentence will be 'I'.

For the second state, we have to choose between the words 'like' and 'love'. Probability distribution now is all about determining the probability that the next word will be 'like' or 'love' given that the previous word is 'I'.

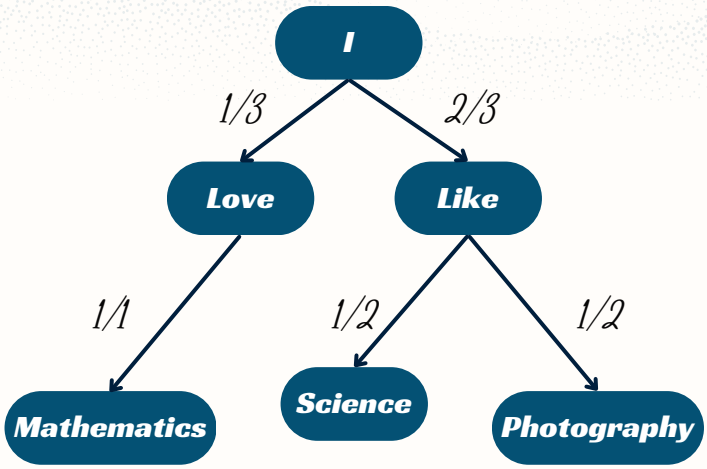


For our example, we can see that the word 'like' appears in 2 of the 3 sentences after 'I' whereas the word 'love' appears only once. Hence there is approximately 67% (2/3) probability that 'like' will succeed after 'I' and 33% (1/3) probability for 'love'.



Similarly, there is a 50-50 chance for 'Science' and 'Photography' to succeed 'like'. And 'love' will always be followed by 'Mathematics' in our case.

- ***State transition diagram for our sample data***



- ***Representing the above work Mathematically as conditional probabilities***

  - ***P(like | I) = 0.67***
  - ***P(love | I) = 0.33***
  - ***P(Photography | like) = P(Science | like) = 0.5***
  - ***P(Mathematics | love) = 1***

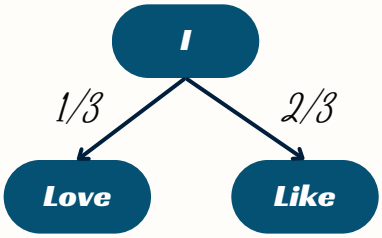
This is how we build a probability distribution from sample data. In this example, you can also see that Markov chains don't look back at what already has occurred and only predict what to do based on the current state. Which is called memorylessness.

- ***Word prediction with Markov chains in Python***

We have all seen the word predictor on our mobile keyboards and pressed on the next prediction until it creates a ridiculous story. But how do they work and how do we create one ourselves with Python & Numpy?

- ***How are we going to store the data?***

The way we are going to use the Markov chain is by setting each word as a state and the next word as a probability of the next state.



To store the data efficiently we need to create an adjacency list (E.4). Which is a fancy way of saying that we store a list of words, and each word contains a list with all probabilities. which will look like the following in Python code:

- ***lexicon = {'I' : {'like': 2/3, 'love': 1/3}}***

- How do predict the next word?

1. Split the sentence into words.
2. Select the last word in the sentence.
3. Look up probabilities of the last word in the lexicon.
4. Select the next word based on the found probability.

MAKING OUR OWN MARKOV CHAIN

- Create the lexicon.

We first need to create a function to populate the lexicon the right way. The following function will count the occurrence and the combination of the words.

```python
import numpy as np
#markov chain stored as adjacency list
lexicon = {}

#current => input word
#next_word => output word
def update_lexicon(current , next_word):
    #add input word to the lexicon if it isn't in there
    if current not in lexicon:
        lexicon.update({current: {next_word: 1}})
        return

    #recieve probs of the input word
    options = lexicon[current]

    #check if output word is in the prob list
    if next_word not in options:
        options.update({next_word: 1})
    else:
        options.update({next_word: options[next_word] + 1})

    #update the lexicon
    lexicon[current] = options
```

- Populating the lexicon.

The following step is to populate the lexicon with the actual data parsed from the dataset.

```
#populate words with actual data parsed from dataset
with open(r'C:\Users\Ahmed kamal\Desktop\dataS.txt' , 'r' ,encoding="utf8") as dataset:
    for line in dataset:
        words = line.strip().split(' ')
        for i in range(len(words) - 1):
            update_lexicon(words[i], words[i+1])
```

This piece of code will loop through all the lines of the file and add the words to the lexicon.

- Adjusting probability.

To be able to use the Numpy random. a choice method with a probability array we need to scale the counter from a 1-infinite (int) to 0–1 (float).

```
#adjust prob => we need to scale the counter from 1-infinite(int) to 0-1(float)
for words, transition in lexicon.items():
    transition = dict((key, value / sum(transition.values())) for key, value in transition.items())
    lexicon[words] = transition
```

- Predicting the next word.

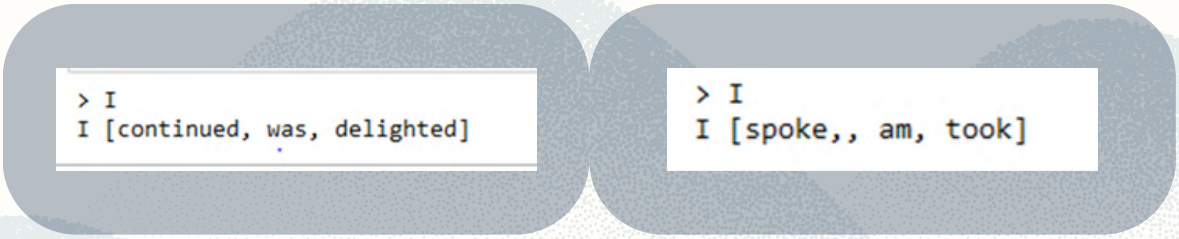
Now we are finally there! Now we can predict the next word based on the user their input.

```
#predicting the next word based on the user input
line = input('> ')
word = line.strip().split(' ')[-1]
if word not in lexicon:
    print('Word not found')
else:
    options = lexicon[word]
    predicted = np.random.choice(list(options.keys()), p = list(options.values()))
    predicted1 = np.random.choice(list(options.keys()), p = list(options.values()))
    predicted2 = np.random.choice(list(options.keys()), p = list(options.values()))
    print(line + ' [' + predicted + ', ' + predicted1 + ', ' + predicted2 + ']')
```

- Now we can use our word predictor project .

First, enter our word to predict the next word

If our input is " I " then the model predict three words by random like the keyboard



PART OF SPEECH TAGGING

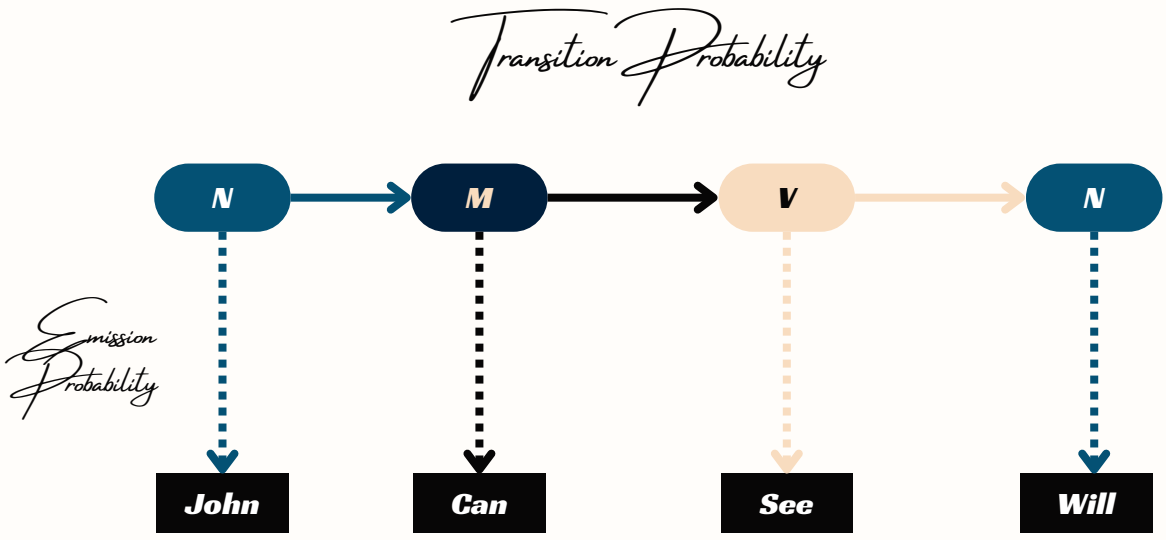
Hidden Markov Model

- # What is Part of Speech (POS) tagging?

Back in elementary school, we have learned the differences between the various parts of speech tags such as nouns, verbs, adjectives, and adverbs. Associating each word in a sentence with a proper POS (part of speech) is known as POS tagging or POS annotation. POS tags are also known as word classes, morphological classes, or lexical tags.
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.
In this,
you will learn how to use

- 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.
find out how HMM selects an appropriate tag sequence for a sentence.



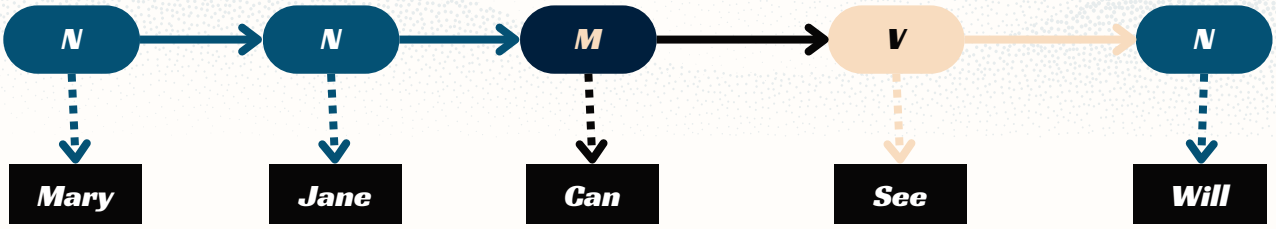
In this example, we consider only 3 POS tags that are noun, model and verb. Let the sentence " Ted will spot Will " be tagged as noun, model, verb and a noun and to calculate the probability

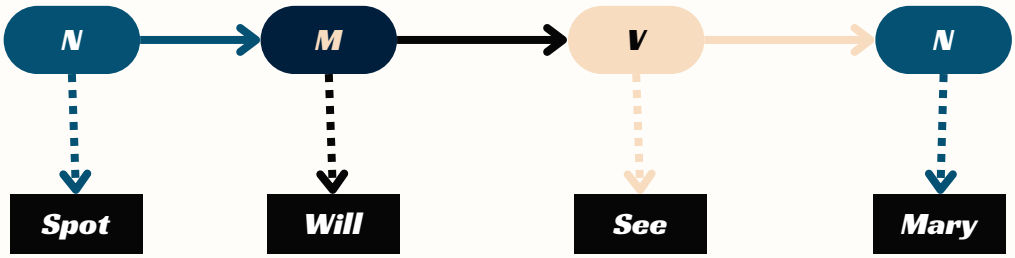
associated with this particular sequence of tags we require their Transition probability and Emission probability.
The transition probability is the likelihood of a particular sequence for example, how likely is that a noun is followed by a model and a model by a verb and a verb by a noun. This probability 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.

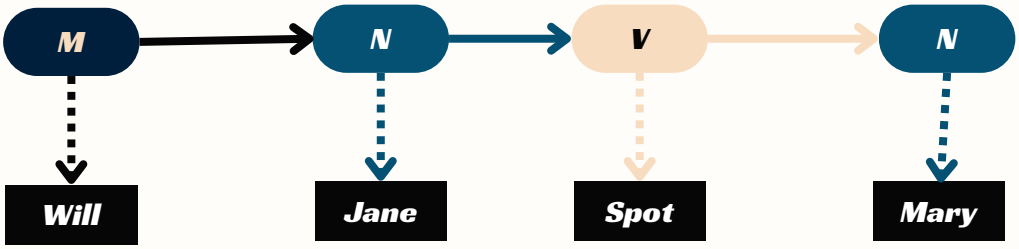
Let us calculate the above two probabilities
for the set of sentences below

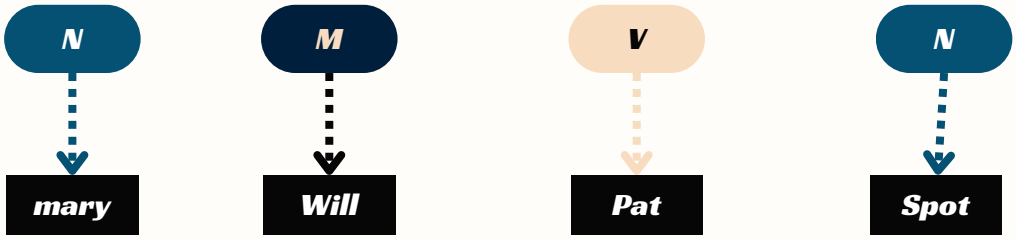
- ***Mary Jane can see Will***
- ***Spot will see Mary***
- ***Will Jane spot Mary?***
- ***Mary will pat Spot***

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.

TAG / WORDS	MARY	JANE	WILL	SPOT	CAN	SEE	PAT
NOUN	4	2	1	2	0	0	0
MODEL	0	0	3	0	1	0	0
VERB	0	0	0	1	0	2	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.

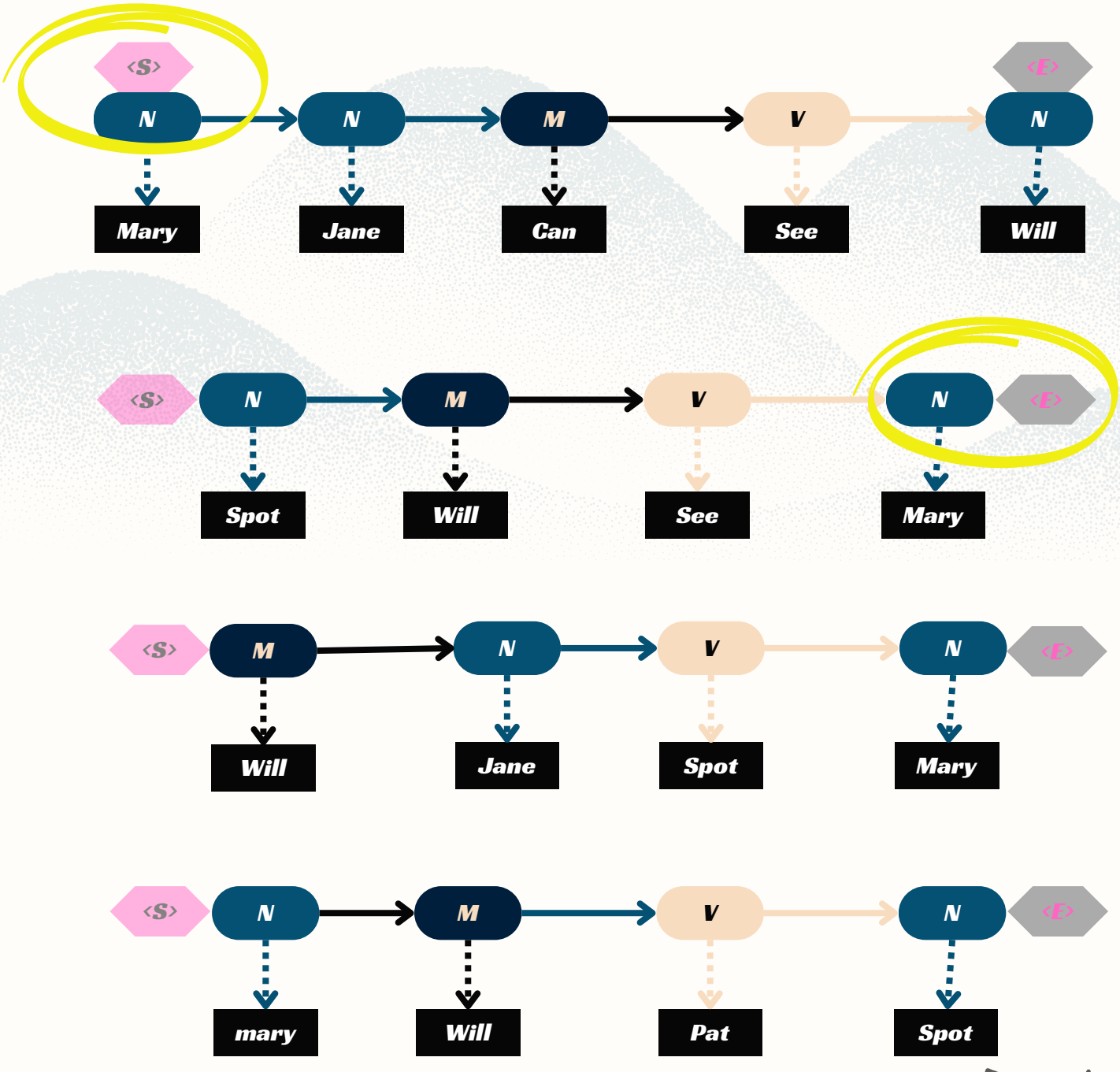
TAG / WORDS	MARY	JANE	WILL	SPOT	CAN	SEE	PAT
NOUN	4/9	2/9	1/9	2/9	0	0	0
MODEL	0	0	3/4	0	1/4	0	0
VERB	0	0	0	1/4	0	2/4	1/4

- 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***

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.



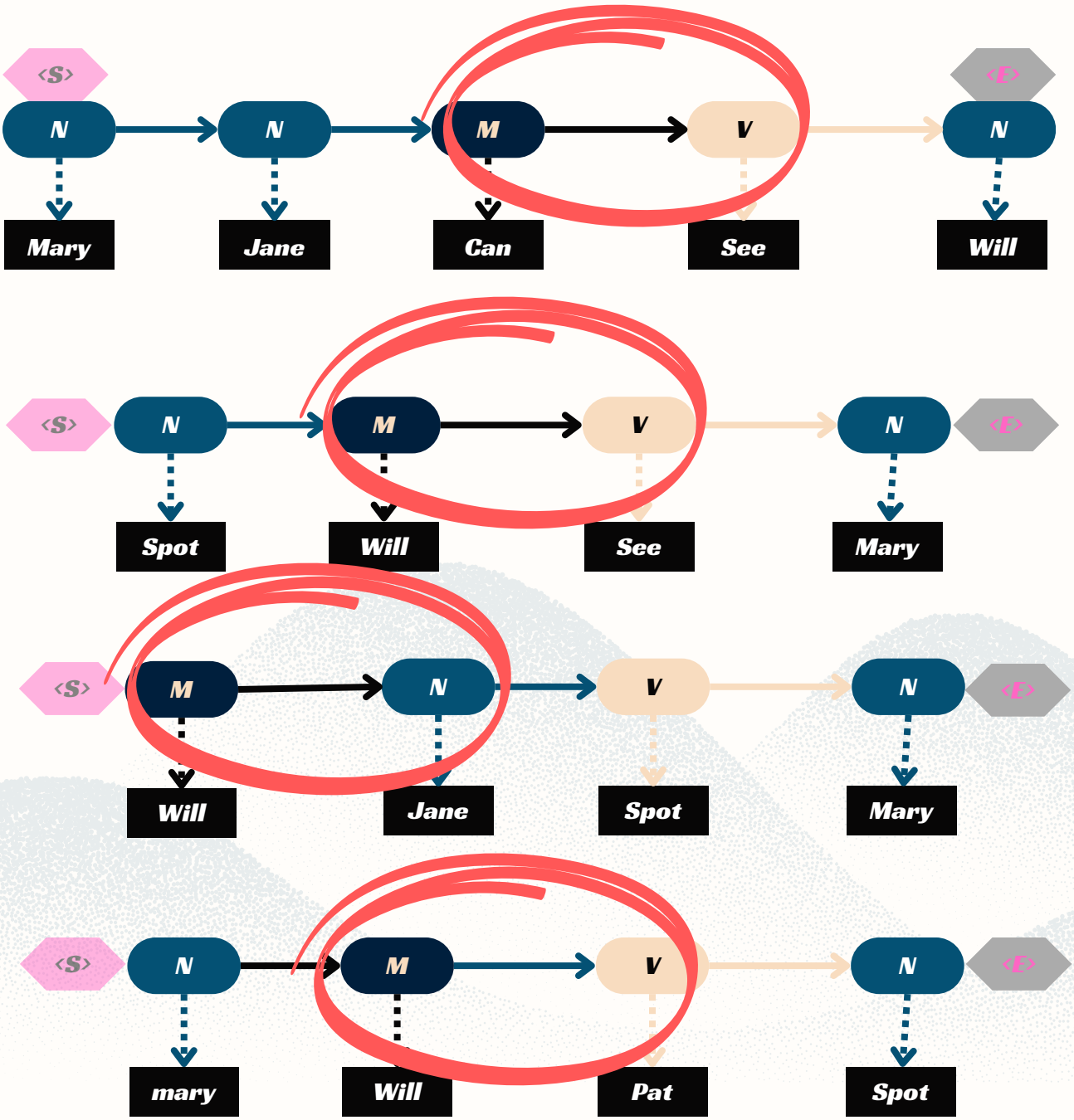
MORE ON THE OTHER SIDE

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.


‹S›
N
N
M
V
N
‹E›
Mary
Jane
Can
See
Will
‹S›
N
M
V
N
‹E›
Spot
Will
See
Mary
‹S›
M
N
V
N
‹E›
Will
Jane
Spot
Mary
‹S›
N
M
V
N
‹E›
mary
Will
Pat
Spot

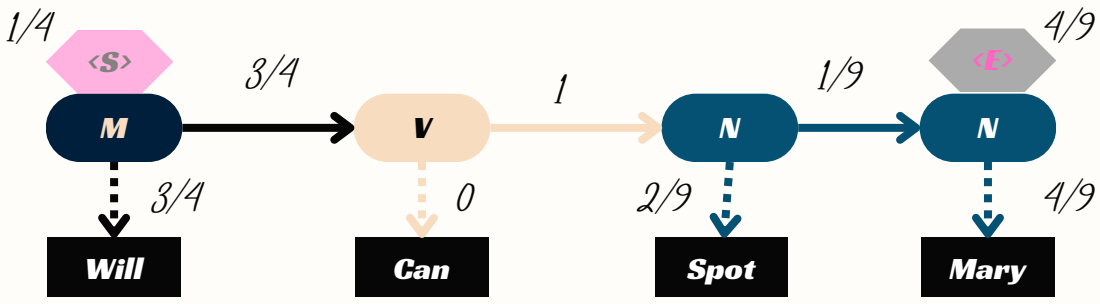
	N	M	V	<E>
<S>	3/4	1/4	0	0
N	1/9	3/9	1/9	4/9
M	1/4	0	3/4	0
V	4/4	0	0	0

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.

- POS tagging with Hidden Markov Model.

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

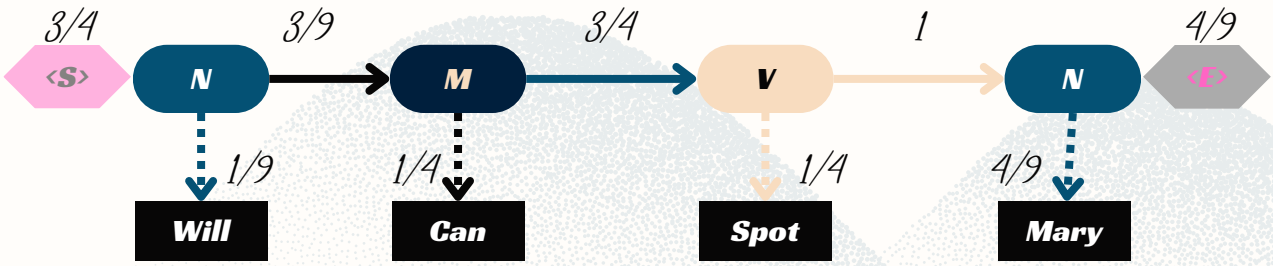
Now calculate the probability of this sequence being correct in the following manner.



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.

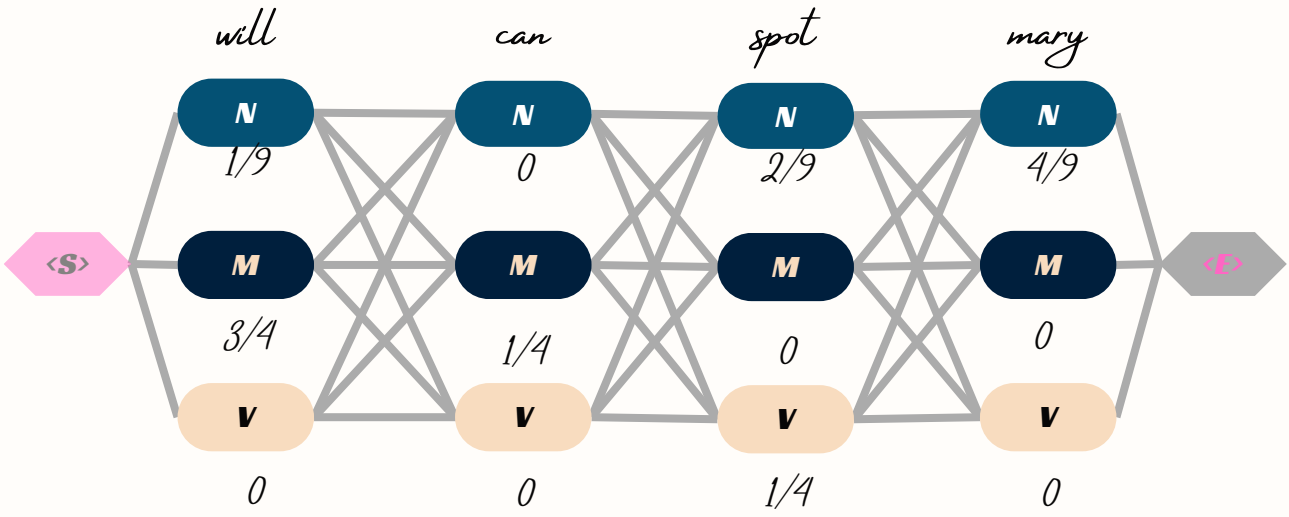
- **1/4*3/4*3/4*0*1*2/9*1/9*4/9*4/9=0**

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

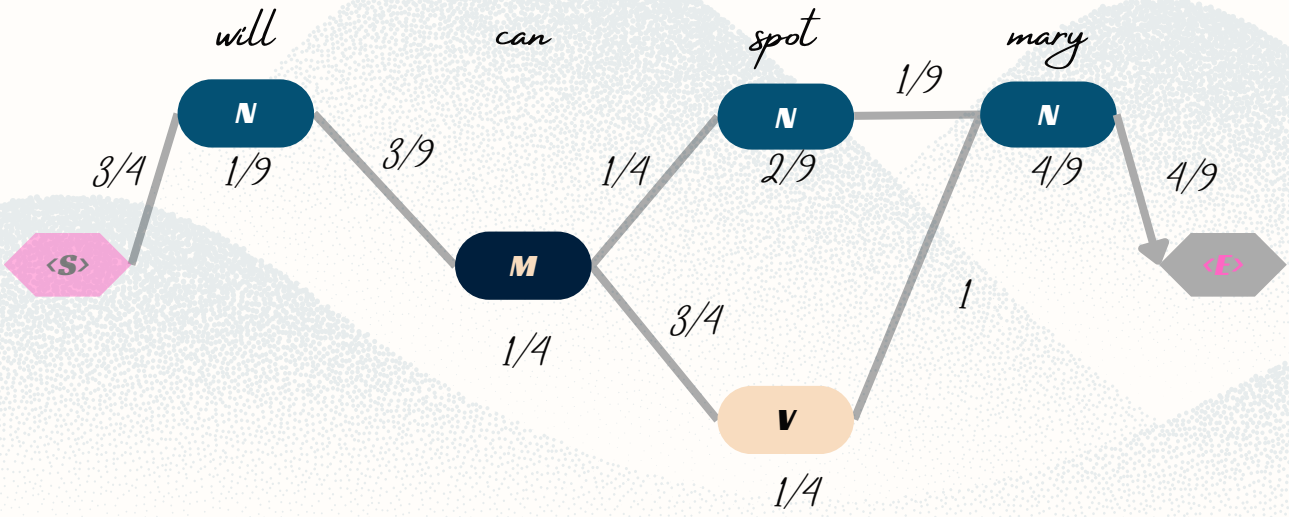


- **3/4*1/9*3/9*1/4*3/4*1/4*1*4/9*4/9=0.00025720164**

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:



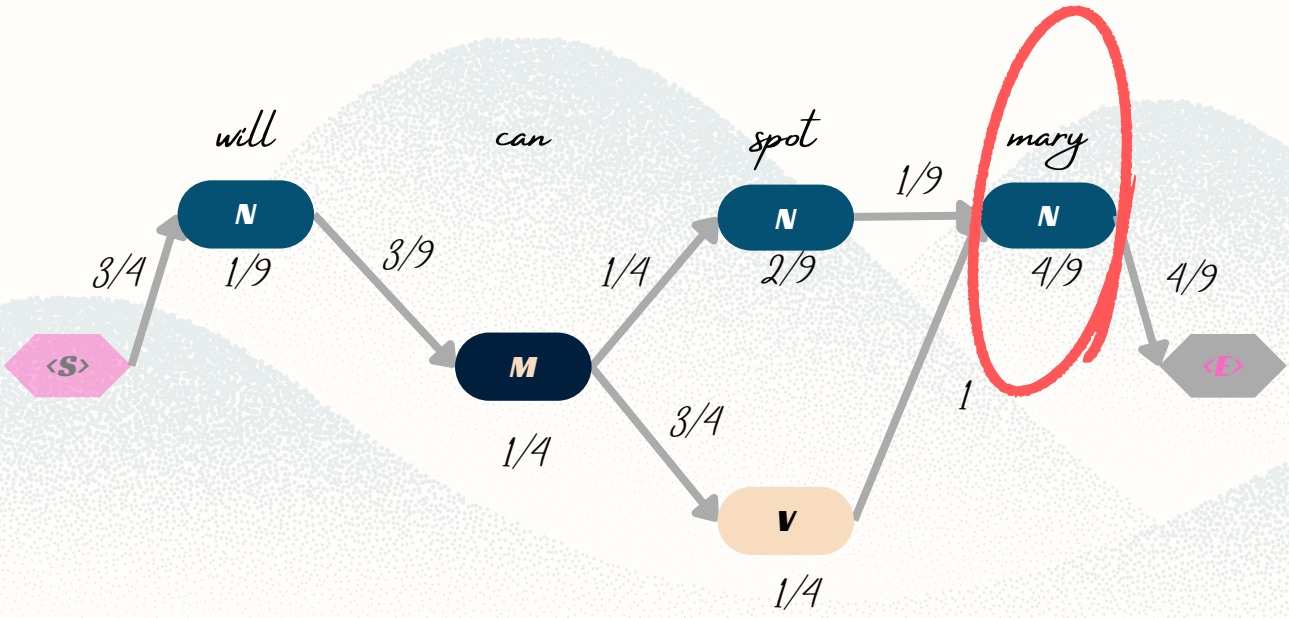
- Now there are only two paths that lead to the end, let us calculate the probability associated with each path.

  - **<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**

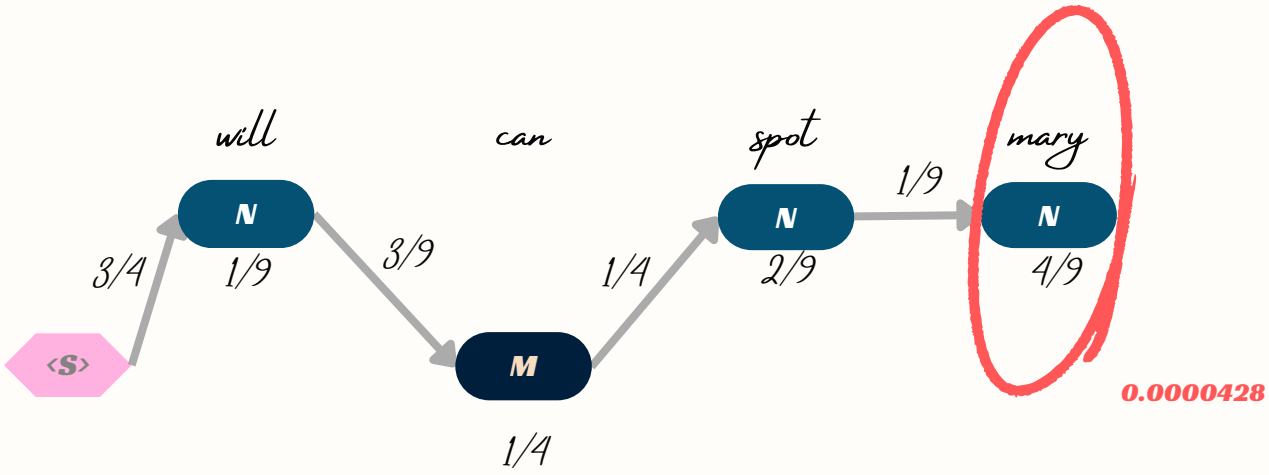
- *Clearly, the probability of the second sequence is much higher and hence the HMM is going to tag each word in the sentence according to this sequence.*

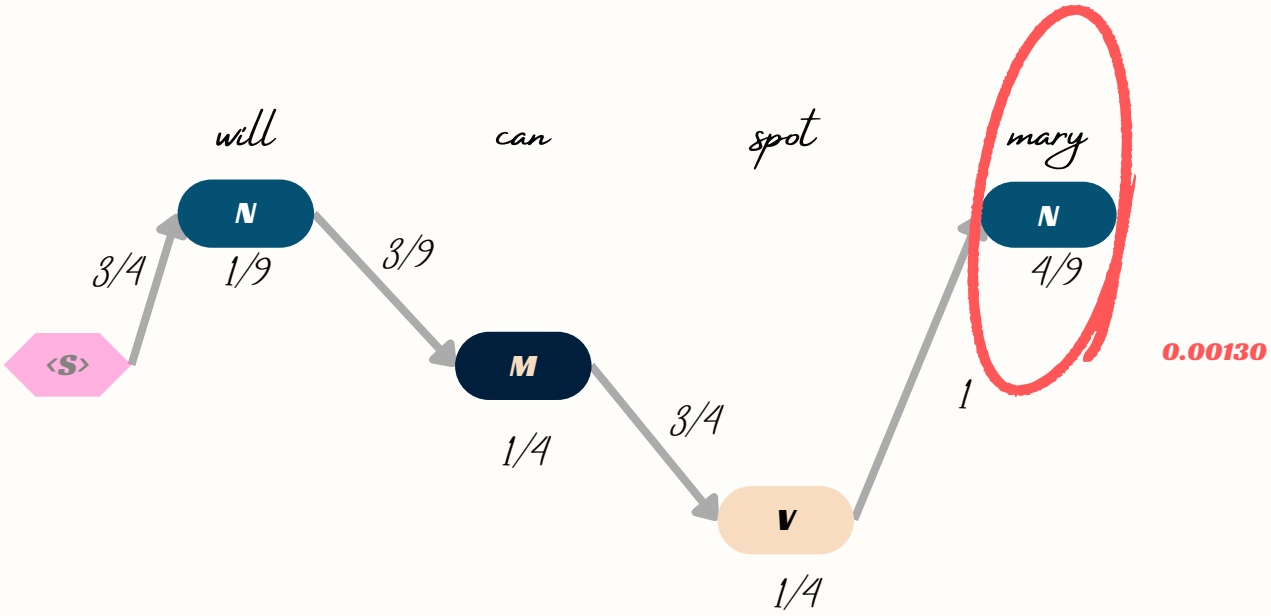
Optimizing HMM with Viterbi Algorithm

The Viterbi algorithm is a dynamic programming algorithm for finding the most likely sequence of hidden states —called the Viterbi path—that results in a sequence of observed events, especially in the context of hidden Markov models (HMM). In the previous section, we optimized the HMM and bought 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.

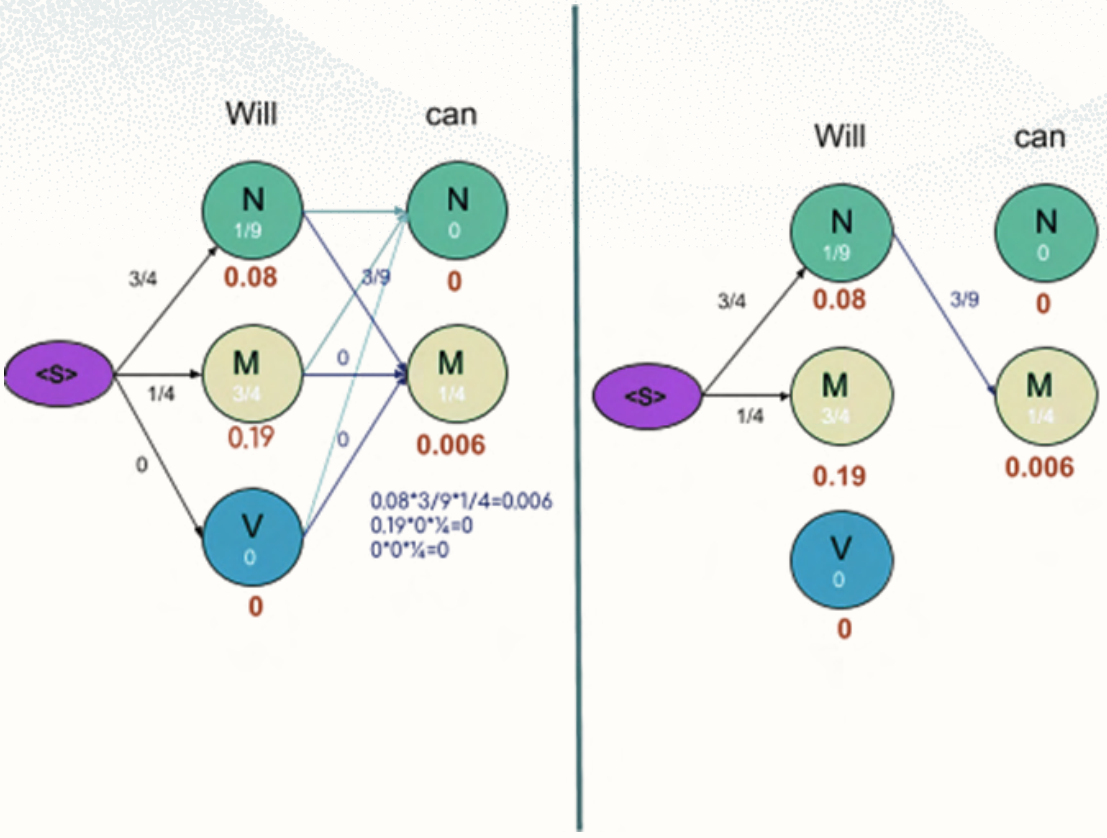


Consider the vertex encircled in the above example. There are two paths leading to this vertex as shown below along with the probabilities of the two mini-paths.

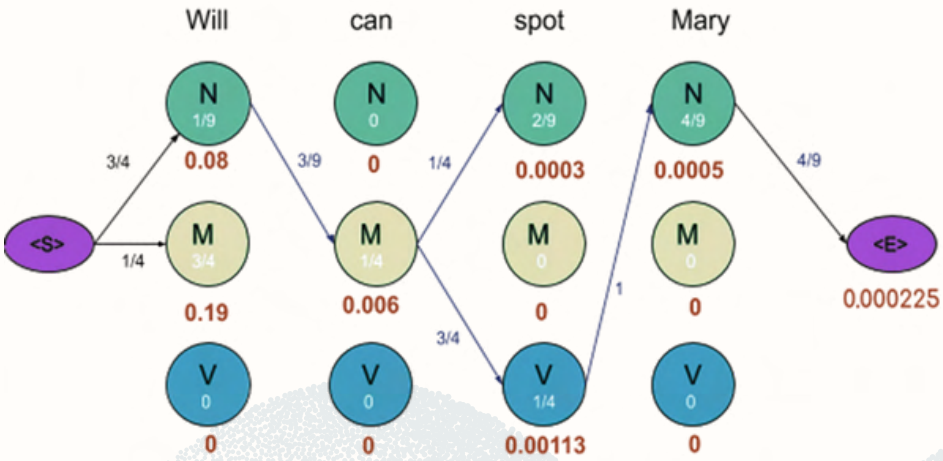




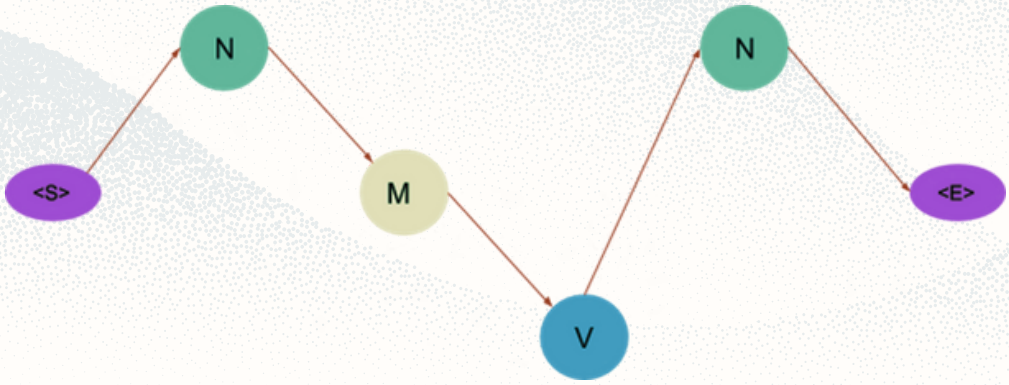
Now we are really concerned with the mini path having the lowest probability. The same procedure is done for all the states in the graph as shown in the figure below



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.

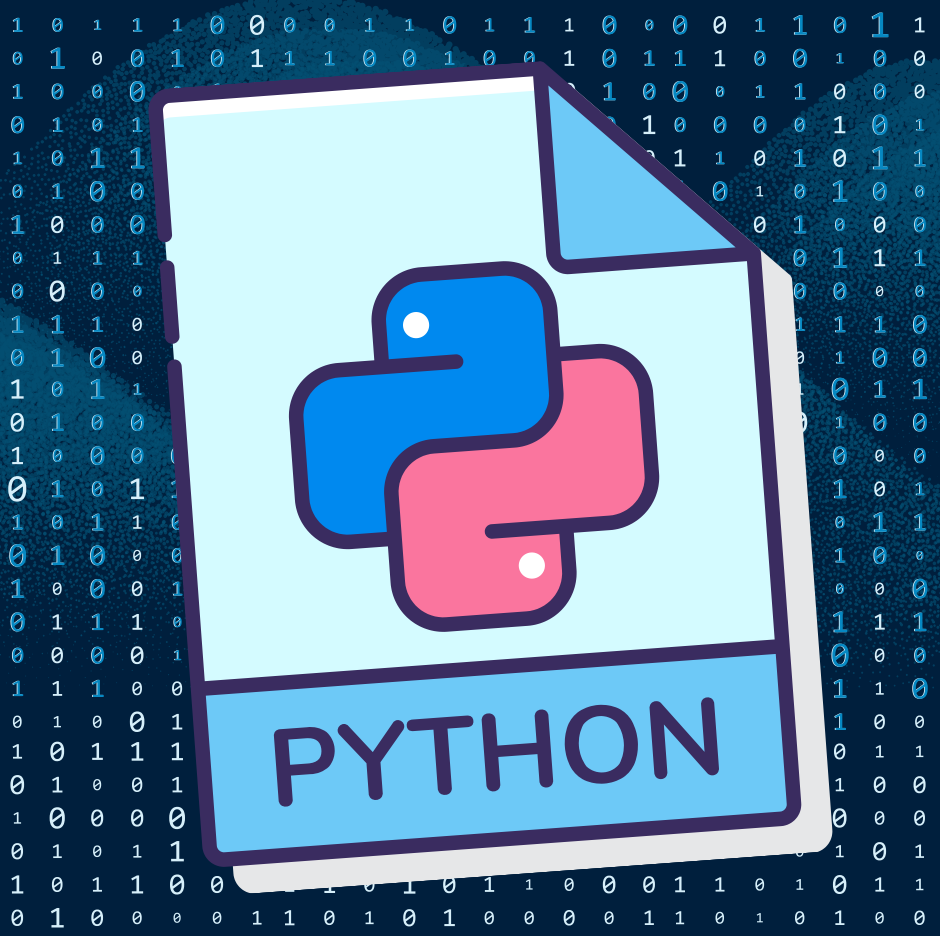
- ***After applying the Viterbi algorithm the model tags the sentence as following:***

  - ***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.

Implementation

With



Our model will learn from real text

So we will do a little bit of data preparation before running the model

```
import re
alphabets= "([A-Za-z])"
prefixes = "(Mr|St|Mrs|Ms|Dr)[.]"
suffixes = "(Inc|Ltd|Jr|Sr|Co)"
starters = "(Mr|Mrs|Ms|Dr|He\s|She\s|It\s|They\s|Their\s|Our\s|We\s|But\s|However\s|That\s|This\s|Wherever)"
acronyms = "([A-Z][.][A-Z][.](?:[A-Z][.])?)"
websites = "[.](com|net|org|io|gov)"
digits = "([0-9])"
```

```
def split_into_sentences(text):
    text = " " + text + "  "
    text = text.replace("\n"," ")
    text = re.sub(prefixes,"\\1<prd>",text)
    text = re.sub(websites,"<prd>\\1",text)
    text = re.sub(digits + "[.]" + digits,"\\1<prd>\\2",text)
    if "..." in text: text = text.replace("...","<prd><prd><prd>")
    if "Ph.D" in text: text = text.replace("Ph.D.","Ph<prd>D<prd>")
    text = re.sub("\s" + alphabets + "[.] "," \\1<prd> ",text)
    text = re.sub(acronyms+" "+starters,"\\1<stop> \\2",text)
    text = re.sub(alphabets + "[.]" + alphabets + "[.]" + alphabets + "[.]","\\1<prd>\\2<prd>\\3<prd>",text
    text = re.sub(alphabets + "[.]" + alphabets + "[.]","\\1<prd>\\2<prd>",text)
    text = re.sub(" "+suffixes+"[.] "+starters," \\1<stop> \\2",text)
    text = re.sub(" "+suffixes+"[.]"," \\1<prd>",text)
    text = re.sub(" " + alphabets + "[.]"," \\1<prd>",text)
    if "”" in text: text = text.replace(".”","”.")
    if "\"" in text: text = text.replace(".\"","\".")
    if "!" in text: text = text.replace("!\"","\"!")
    if "?" in text: text = text.replace("?\"","\"?")
    text = text.replace(".",".<stop>")
    text = text.replace("?","?<stop>")
    text = text.replace("!","!<stop>")
    text = text.replace("<prd>",".")
    sentences = text.split("<stop>")
    sentences = sentences[:-1]
    sentences = [s.strip() for s in sentences]
    return sentences
```

In the pieces of code above we are using "regex" to split any raw text into sentences.

```
df = pd.DataFrame()
df=df.append(split_into_sentences(txt))
from nltk import word_tokenize,pos_tag
l =int(df.shape[0])
dic1={'sentence':[],'word':[],'pos':[]}
for i in range(0,l):
    text = word_tokenize(str(df.values[i]))
    for j in pos_tag(text):
        dic1['sentence'].append(i)
        dic1['word'].append(j[0])
        dic1['pos'].append(j[1])
df2 = pd.DataFrame(dic1)
df2.head()
```

then into words and give them their POS tags

these are the main libraries we need

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import seaborn as sns
import random
from matplotlib import pyplot as plt # show graph
from sklearn.model_selection import GroupShuffleSplit
import warnings
warnings.filterwarnings('ignore')
from hmmlearn import hmm
from sklearn.metrics import confusion_matrix, classification_report, precision_score
```

#1

```
data = pd.read_csv("C:/Users/Kimo Store/OneDrive/Desktop/words_pos.csv" , encoding='latin1' , index_col=[0])
data.head()
```

	sentence	word	pos
0	0	Mr	NNP
1	0	and	CC
2	0	Mrs	NNP
3	0	Dursley	NNP
4	0	.	.

loading data

#2

```
tags = list(set(data.pos.values)) #Read POS values
words = list(set(data.word.values))
len(tags), len(words)
```

```
(38, 872)
```

Get the numbers of tags & words inside the whole data.

We cannot split data normally with train_test_split because doing that makes some parts of a sentence in the training set while some others are in the testing set. Instead, we use GroupShuffleSplit.

```
y = data.pos
X = data.drop('pos', axis=1)

gs = GroupShuffleSplit(n_splits=2, test_size=.33, random_state=42)
train_ix, test_ix = next(gs.split(X, y, groups=data['sentence']))

data_train = data.loc[train_ix]
data_test = data.loc[test_ix]

data_train
```

	sentence	word	pos
0	0	Mr	NNP
1	0	and	CC
2	0	Mrs	NNP
3	0	Dursley	NNP
4	0	,	,
5	0	of	IN
6	0	number	NN
7	0	four	CD
8	0	,	,
9	0	Privet	NNP
10	0	Drive	NNP

After checking the data after splitting, it seems to be fine. Check the number of tags & words in the training set.

```
tags = list(set(data_train.pos.values)) #Read POS values
words = list(set(data_train.word.values))
len(tags), len(words)
```

```
(36, 682)
```

The number of tags is enough but the number of words is not enough Because of that we need to randomly add some UNKNOWN words into the training dataset then we recalculate the word list and create map from them to number.

```
dfupdate = data_train.sample(frac=.15, replace=False, random_state=42)
dfupdate.word = 'UNKNOWN'
data_train.update(dfupdate)
words = list(set(data_train.word.values))
# Convert words and tags into numbers
word2id = {w: i for i, w in enumerate(words)}
tag2id = {t: i for i, t in enumerate(tags)}
id2tag = {i: t for i, t in enumerate(tags)}
len(tags), len(words)
```

```
(36, 605)
```

the input of the training is just a dataset (Words). We cannot map back the states to the POS tag.
That's why we have to calculate the model parameters for hmmlearn.hmm.MultinomialHMM manually by calculating

- ***initial matrix***
- ***transition matrix***
- ***emission matrix***

```
count_tags = dict(data_train.pos.value_counts())
count_tags_to_words = data_train.groupby(['pos']).apply(lambda grp: grp.groupby('word')['pos'].count().to_dict()).to_dict()
count_init_tags = dict(data_train.groupby('sentence').first().pos.value_counts())

# TODO use panda solution
count_tags_to_next_tags = np.zeros((len(tags), len(tags)), dtype=int)
sentences = list(data_train.sentence)
pos = list(data_train.pos)
for i in range(len(sentences)) :
    if (i > 0) and (sentences[i] == sentences[i - 1]):
        prevtagid = tag2id[pos[i - 1]]
        nexttagid = tag2id[pos[i]]
        count_tags_to_next_tags[prevtagid][nexttagid] += 1
```

```
mystartprob = np.zeros((len(tags),))
mytransmat = np.zeros((len(tags), len(tags)))
myemissionprob = np.zeros((len(tags), len(words)))
num_sentences = sum(count_init_tags.values())
sum_tags_to_next_tags = np.sum(count_tags_to_next_tags, axis=1)
for tag, tagid in tag2id.items():
    floatCountTag = float(count_tags.get(tag, 0))
    mystartprob[tagid] = count_init_tags.get(tag, 0) / num_sentences
    for word, wordid in word2id.items():
        myemissionprob[tagid][wordid]= count_tags_to_words.get(tag, {}).get(word, 0) / floatCountTag
    for tag2, tagid2 in tag2id.items():
        mytransmat[tagid][tagid2]= count_tags_to_next_tags[tagid][tagid2] / sum_tags_to_next_tags[tagid]
```

```
model = hmm.CategoricalHMM(n_components=len(tags), algorithm='viterbi', random_state=42)
model.startprob_ = mystartprob
model.transmat_ = mytransmat
model.emissionprob_ = myemissionprob
```

Initialize a HMM

As some words may never appear in the training set, we need to transform them into UNKNOWN first. Then we split data_test into samples & lengths and send them to HMM.

```
data_test.loc[~data_test['word'].isin(words), 'word'] = 'UNKNOWN'
data_test.loc[~data_test['pos'].isin(tags),'pos'] = random.choice(tags)
print(data_test)
word_test = list(data_test.word)
samples = []
for i, val in enumerate(word_test):
    samples.append([word2id[val]])
```

	sentence	word	pos
54	2	Mr	NNP
55	2	Dursley	NNP
56	2	was	VBD
57	2	the	DT
58	2	UNKNOWN	NN
59	2	of	IN
60	2	a	DT
61	2	UNKNOWN	NN
62	2	called	VBN
63	2	Grunnings	NNP
64	2	,	,
65	2	which	WDT
66	2	made	VBD
67	2	drills	NNS
234	9	The	DT
235	9	Dursleys	NNP
236	9	UNKNOWN	VBD
237	9	to	TO

```
# TODO use panda solution
lengths = []
count = 0
sentences = list(data_test.sentence)
for i in range(len(sentences)) :
    if (i > 0) and (sentences[i] == sentences[i - 1]):
        count += 1
    elif i > 0:
        lengths.append(count)
        count = 1
    else:
        count = 1
len(lengths),len(samples)
```

```
(60, 1022)
```

```
pos_predict = model.predict(samples , lengths)
pos_predict
```

```
array([28, 28, 12, ...,  3, 12,  3], dtype=int64)
```

```
predicted_sequence = []
for i in pos_predict:
    predicted_sequence.append(id2tag[i])
predicted_sequence
```

```
['NNP',
 'NNP',
 'VBD',
 'DT',
 'NN',
 'IN',
 'DT',
 'NN',
 'VBN',
 'NNP',
 ',',
 'WDT',
 'VBD',
 'NNS',
 'DT',
 'NNP',
 'VBD',
 'RB',
 'RB',
```

```
for x,y in zip(data_test['word'],predicted_sequence):
    print(("Word = "+str(x),"Predicted POS = "+str(y)))
```

```
('Word = Mr', 'Predicted POS = NNP')
('Word = Dursley', 'Predicted POS = NNP')
('Word = was', 'Predicted POS = VBD')
('Word = the', 'Predicted POS = DT')
('Word = UNKNOWN', 'Predicted POS = NN')
('Word = of', 'Predicted POS = IN')
('Word = a', 'Predicted POS = DT')
('Word = UNKNOWN', 'Predicted POS = NN')
('Word = called', 'Predicted POS = VBN')
('Word = Grunnings', 'Predicted POS = NNP')
('Word = ,', 'Predicted POS = ,')
('Word = which', 'Predicted POS = WDT')
('Word = made', 'Predicted POS = VBD')
('Word = drills', 'Predicted POS = NNS')
('Word = The', 'Predicted POS = DT')
('Word = Dursleys', 'Predicted POS = NNP')
('Word = UNKNOWN', 'Predicted POS = VBD')
('Word = to', 'Predicted POS = RB')
('Word = think', 'Predicted POS = RB')
```

```
pd.set_option('display.max_rows', None)
data_test.head(100)
```

	sentence	word	pos
54	2	Mr	NNP
55	2	Dursley	NNP
56	2	was	VBD
57	2	the	DT
58	2	UNKNOWN	NN
59	2	of	IN
60	2	a	DT
61	2	UNKNOWN	NN
62	2	called	VBN
63	2	Grunnings	NNP
64	2	,	,

```
tags_test = list(data_test.pos)
pos_test = np.zeros((len(tags_test), ), dtype=int)
# print(pos_test)
# print(tags_test)
# print(len(pos_test))
# print(len(tags_test))

for i, val in enumerate(tags_test):
    pos_test[i] = tag2id[val]
len(pos_predict), len(pos_test), len(samples), len(word_test)
```

```
(1019, 1022, 1022, 1022)
```

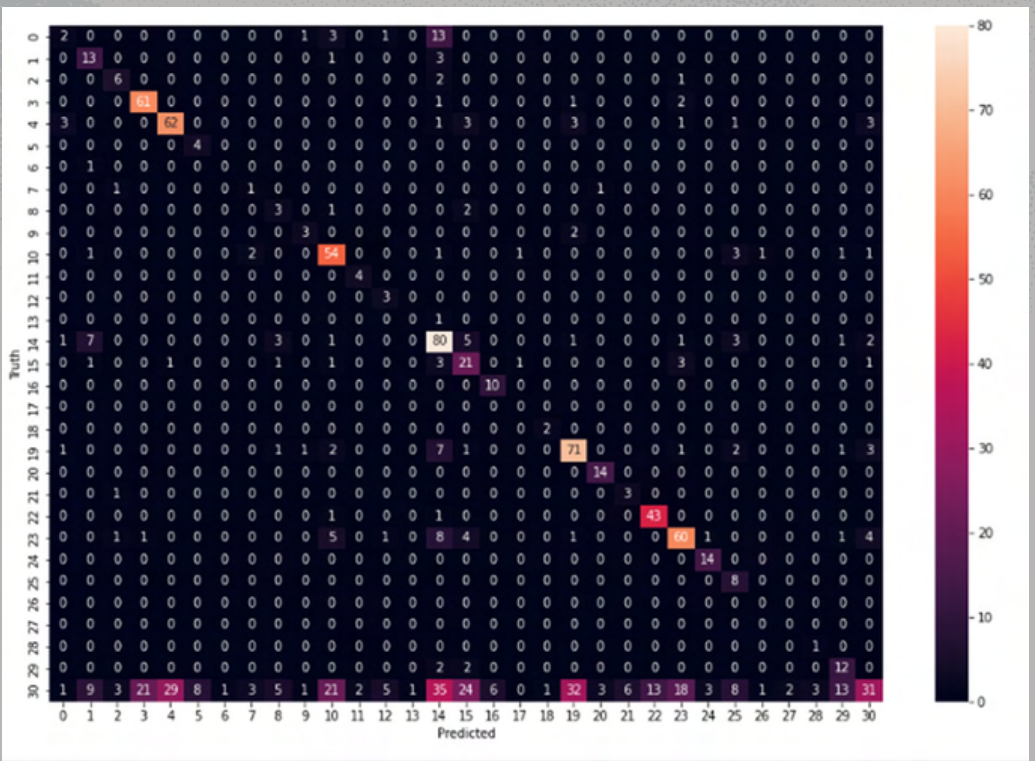
```
def reportTest(y_pred, y_test):
    print("The precision is {}".format(precision_score(y_test, y_pred, average='weighted')))

min_length = min(len(pos_predict), len(pos_test))
reportTest(pos_predict[:min_length], pos_test[:min_length])
```

```
The precision is 0.7625864734372477
```

```
cm = confusion_matrix(pos_predict[:min_length] , pos_test[:min_length])
plt.figure(figsize = (15 , 10))
sns.heatmap(cm , annot = True )
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

```
Text(114.0, 0.5, 'Truth')
```



Thanks