19MAT301: Mathematics for Intelligent Systems

End semester Project on

HMM, CRF and RNN for POS Tagging

Centre for Computational Engineering and Networking
Amrita Vishwa Vidyapeetham, Coimbatore

Arun Sivasubramanian (CB.EN.U4AIE19013)
Arun Prakash J (CB.EN.U4AIE19014)
P S Sai Ganesh (CB.EN.U4AIE19044)
Prashanth VR (CB.EN.U4AIE19047)

Table of Contents:

Introduction:

In probability theory, a Markov model is a stochastic model used to model pseudo-randomly changing systems. It is assumed that future states depend only on the current state, not on the events that occurred before it. In a special case called Hidden Markov Models, we have hidden transition states within observable states.

Natural language processing (NLP) refers to the branch of computer science—and more specifically, the branch of artificial intelligence or AI—concerned with giving computers the ability to understand text and spoken words in much the same way human beings can.

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 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 can say 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, verbs, adverbs, adjectives, pronouns, conjunction and their sub-categories.

A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes form a directed or undirected graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior.

This work gives an effective explanation of how Hidden Markov models and neural network techniques such as Recurrent Neural Networks can be used for POS tagging. The different models used help us to understand the necessity of each modification in the overall pipeline. In addition to this, we can also modify the models in order to cater to the needs of the model, subject to the availability of time and computational requirements.

Markov Chain:

A Markov chain, named after Andrey Markov, is a model which describes the probabilities of sequences of random variables or states, each of which can take on values from some set. These sets can be tags or words, or symbols representing anything like weather. The Markov chain assumes that if we want to predict what the future state is, then all that matters is the current state. The states before this current state do not have a direct impact on the future state except for the current state which is also known as Memory lessness property. For this reason, it is desirable for a proposed model to exhibit Markov property in the fields of predictive modelling and probabilistic forecasting.

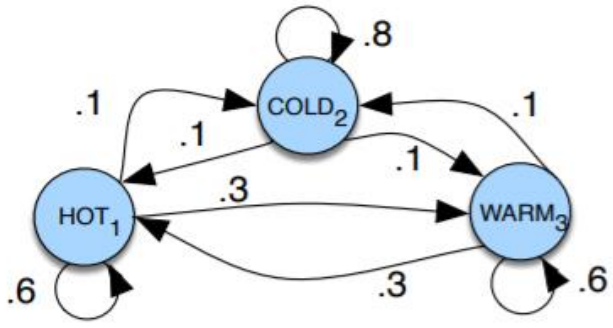

Fig. 1 A Markov chain for weather

In the above state diagram, to predict tomorrow's weather you could look into today's weather but not look at yesterday's weather. A start/ Initial probability distribution π is required where $\pi = [0.1,0.7,0.2]$ which would mean that a probability of 0.7 of starting in state 2 (cold).

In mathematical notations, considering a sequence of state variables $q_1, q_2 \ldots . q_i$, a Markov model represents the Markov assumption on the probabilities of this sequence i.e., for predicting the next state only the current state is required and the states before them doesn't matter.

Markov Assumption: $P(q_i = a | q_1 \ldots . q_{i-1}) = P(q_i = a | q_{i-1})$

The states in a Markov chain are represented as nodes and their transitions with the probabilities as edges. The values of the edges/ transition probabilities leaving a node must sum to 1.

Generally, a Markov chain is described using the following components:

1. $Q = q_1 q_2 \ldots q_n$, where n is the number of states
2. $A = a_{11} a_{12} \ldots a_{nn}$, a transition probability matrix where each state a_{ij} represent the probability of moving from state i to j
3. $\pi_k = \pi_1 \pi_2 \ldots \pi_n$, an initial probability distribution vector

Assumptions for Markov Chain:

1. The statistical system contains a finite number of states.
2. The states are mutually exclusive and collectively exhaustive.

3. The transition probability from one state to another state is constant over time.

Markov processes are fairly common in real-life problems and Markov chains can be easily implemented because of their memory lessness property. Using Markov chains can simplify the problem without affecting its accuracy.

Properties of Markov Chain:

1. A Markov chain is said to be Irreducible if we can go from one state to another in a single or more than one step.
2. A state in a Markov chain is said to be Periodic if returning to it requires a multiple of some integer larger than 1, the greatest common divisor of all the possible return path lengths will be the period of that state.
3. A state in a Markov chain is said to be Transient if there is a non-zero probability that the chain will never return to the same state, otherwise, it is Recurrent.
4. A state in a Markov chain is called Absorbing if there is no possible way to leave that state. Absorbing states do not have any outgoing transitions from it.

Types of Markov Chain:

1. Discrete-time Markov chains: The state of the process at time t is a countable set or the transitions occur at specific states. Generally, the term “Markov chain” is used for DTMC.
2. Continuous-time Markov chains: The state of the process at time t is a continuum (a range or series of things that are slightly different from

each other and that exist between two different states), which means changes are continuous in CTMC

Applications of Markov Chain:

Markov chains make the study of many real-world processes much simpler and easier to understand. Using the Markov chain, we can derive some useful results such as Stationary Distribution and many more.

1. MCMC (Markov Chain Monte Carlo), which gives a solution to the problems that come from the normalization factor, is based on the Markov Chain.
2. Markov Chains are used in information theory, search engines, speech recognition etc.
3. Markov chain has huge possibilities, future and importance in the field of Data Science and the interested readers are requested to learn this stuff properly for being a competent person in the field of Data Science.

Hidden Markov Models:

Hidden Markov models (HMMs) are a formal foundation for making probabilistic models. They provide a conceptual toolkit for building complex models just by drawing an intuitive picture. They are at the heart of a diverse range of programs, including profile searches, multiple sequence alignment and regulatory site identification. HMMs are the Legos of computational sequence analysis.

When we cannot observe the states themselves, but only the result of some probability function(observation) of the states we utilize HMM. HMM is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states.

Assumptions of HMM:

HMM too is built upon several assumptions and the following is vital.

- Output independence assumption: Output observation is conditionally independent of all other hidden states and all other observations when given the current hidden state.
- Emission Probability Matrix: Probability of hidden state generating output v_i given that state at the corresponding time was s_j.

Understanding HMM:

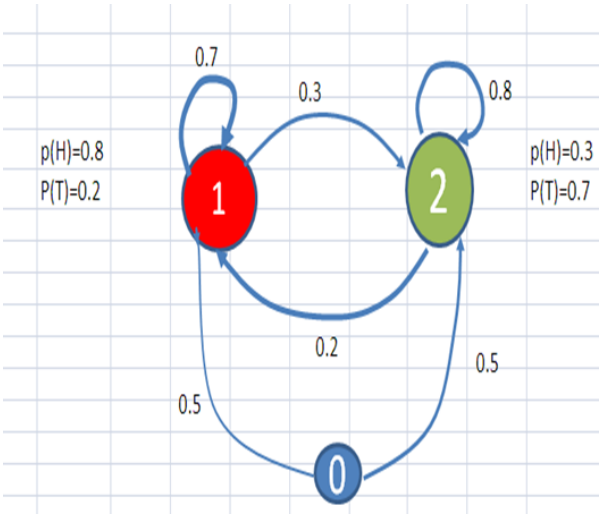

Fig. 2: A HM chain with coin toss as hidden state

Every Hidden Markov Model can be described as a finite state machine. Considering the diagram above, the following can be interpreted:

1. Set of states (S) = {0,1,2}
2. Set of hidden states (Q) = {Head, Tail}
3. State series over time = z∈ S_T

The transition from one state to another that you understand from a transition is called the observations since you observe them. The coin toss yielding a head/tail that influences what happens after reaching a state is called the hidden state since you can't observe it.

The probability for transition from one state to another is called the emission probability. In this case, the values on the edges are transition probabilities. The transition matrix will tell us what is the cumulative transition property for transition from one state sequence to another during an independent process.

The probability of influence of the hidden state variables on the observable states which explain the transition to-and-fro from the hidden states is called the emission probability.

Learning in HMMs involves estimating the state transition probabilities 'A' and the output emission probabilities 'B' that make an observed sequence most likely. Expectation-Maximization algorithms are used for this purpose.

An algorithm known as Baum-Welch algorithm, that falls under this category using the forward algorithm, is widely used.

Forward Probability (Likelihood Computation):

Given a task, being a climatologist in the year 279 studying the history of global warming, to determine the records of the weather in Baltimore for the summer of 2020, but there are no records found regarding this but we have record of Jason Eisner's dairy, which has a list of how many ice-creams Jason ate every day that summer. Also, we simply assume the weather to be either Hot (H) or Cold (C).

Our first problem is to compute the likelihood of a particular observation sequence. For a Markov chain, where the surface observations are the same as the hidden events, we could compute the probability of $[3, 1, 3]$ just by following the states labeled $[3, 1, 3]$ and multiplying the probabilities along the arcs. For a hidden Markov model,
things are not so simple. We want to determine the probability of an ice-cream observation sequence like $[3, 1, 3]$, but we don't know what the hidden state sequence is.

Let's start with a slightly simpler situation. Suppose we already knew the weather and wanted to predict how much ice cream Jason would eat. This is a useful part of many HMM tasks. For a given hidden state sequence (e.g., $[hot, hot, cold]$), we can easily compute the output likelihood of $[3, 1, 3]$.

Let's see how. First, recall that for hidden Markov models, each hidden state produces only a single observation. Thus, the sequence of hidden states and the sequence of observations have the same length.

Given this one-to-one mapping for a particular hidden state sequence $Q = q_1 q_2 \dots q_T$ and an observation sequence $O = o_1 o_2 \dots o_T$, the likelihood of the observation sequence is

$$P(O|Q) = \prod_{i=1}^{T} P(o_i|q_i)$$

The computation of the forward probability for our ice-cream observation $[3, 1, 3]$ from one possible hidden state sequence $[hot, hot, cold]$ is given as

$$P([3,1,3]|[hot, cold, cold]) = P(3|hot) \times P(1|hot) \times P(3|cold)$$

But of course, we don't actually know what the hidden state (weather) sequence was. We'll need to compute the probability of ice-cream events $[3, 1, 3]$ instead by summing over all possible weather sequences, weighted by their probability. First, let's compute the joint probability of being in a particular weather sequence Q and generating a particular sequence O of ice-cream events. In general, this is

$$P(O, Q) = P(O|Q) \times P(Q) = \prod_{i=1}^{T} P(o_i|q_i) \times \prod_{i=1}^{T} P(q_i|q_{i-1})$$

The computation of the joint probability of our ice-cream observation $[3, 1, 3]$ and one possible hidden state sequence $[hot, hot, cold]$ is given as

$$P([3,1,3],[hot,hot,cold]) = P(hot|start) \times P(hot|hot) \times P(cold|hot) \times P(3|hot) \times P(1|hot) \times P(3|cold)$$

Now that we know how to compute the joint probability of the observations with a particular hidden state sequence, we can compute the total probability of the observations just by summing over all possible hidden state sequences.

$$P(O) = \sum_{Q} P(O,Q) = \sum_{Q} P(O|Q)P(Q)$$

For an HMM with N hidden states and an observation sequence of T observations, there are N^Tpossible hidden sequences. For real tasks, where N and T are both large, N^T is a very large number, so we cannot compute the total observation likelihood by computing a separate observation likelihood for each hidden state sequence and then summing them.

Instead of using such an extremely exponential algorithm, we use an efficient forward $O(N^2T)$ algorithm called the forward algorithm. The forward algorithm is a kind of dynamic programming algorithm, that is, an algorithm that uses a table to store intermediate values as it builds up the probability of the observation sequence. The forward algorithm computes the observation probability by summing over the probabilities of all possible hidden state paths that could generate the observation sequence, but it does so efficiently by implicitly folding each of these paths into a single forward trellis.

Each cell of the forward algorithm trellis at $\alpha_t(j)$ represents the probability of being in state j after seeing the first t observations, given the automaton λ. The value of each cell at $\alpha_t(j)$ is computed by summing over the probabilities of every path that could lead us to this cell. Formally, each cell expresses the following probability:

$$\alpha_t(j) = P(o_1, o_2 \ldots o_t, q_t = j|\lambda)$$

Here, q_j = j means the t[th] state in the sequence of states is state j. We compute this probability at $\alpha_t(j)$ by summing over the extensions of all the paths that lead to the current cell. For a given state q_j at time t, the value at $\alpha_t(j)$ is computed as

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij}\, b_j(o_t)$$

The three factors that are multiplied in the above equation in extending the previous paths to compute the forward probability at time t are
$\alpha_{t-1}(i)$, the previous forward path probability
a_{ij}, the transition probability from previous state q_i to current state q_j
$b_j(o_t)$, the state observation likelihood of the observation symbol o_t given the current state j

Steps involved in the forward algorithm:

1. Initialization:

$$\alpha_1(j) = \pi_j\, b_j(o_1), 1 \leq j \leq N$$

2. Recursion:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij}\, b_j(o_t), 1 \le j \le N\,, 1 \le t \le N$$

3. Termination:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

Viterbi Algorithm (Decoding)

For any model, such as an HMM, that contains hidden variables, the task of determining which sequence of variables is the underlying source of some sequence of observations is called the decoding task. In the ice-cream domain, given a sequence of ice-cream observations $[3, 1, 3]$ and an HMM, the task of the decoder is to find the best hidden weather sequence [H H H].

We might propose to find the best sequence as follows: For each possible hidden state sequence (HHH, HHC, HCH, etc.), we could run the forward algorithm and compute the likelihood of the observation sequence given that hidden state sequence. Then we could choose the hidden state sequence with the maximum observation likelihood. It should be clear from the previous section that we cannot do this because there are an exponentially large number of state sequences.

Instead, the most common decoding algorithm for HMMs is the Viterbi algorithm. Like the forward algorithm, Viterbi is a kind of dynamic programming algorithm that makes use of a dynamic programming trellis.

The idea is to process the observation sequence left to right, filling out the trellis. Each cell of the trellis, $v_t(j)$, represents the probability that the HMM is in state j after seeing the first t observations and passing through the most probable state sequence $q_1,\ldots,q_{t-1}$, given the automaton λ. The value of each cell $v_t(j)$ is computed by recursively taking the most probable path that could lead us to this cell. Formally, each cell expresses the probability

$$v_t(j) = max_{i=1}^{N} P(q_1,\ldots,q_{t-1},o_1,\ldots,o_t,q_t = j|\lambda)$$

Note that we represent the most probable path by taking the maximum over all possible previous state sequences $max_{q_1,\ldots,q_{t-1}}$. Like other dynamic programming algorithms, Viterbi fills each cell recursively. Given that we had already computed the probability of being in every state at time t -1, we compute the Viterbi probability by taking the most probable of the extensions of the paths that lead to the current cell. For a given state q_t at time t, the value $v_t(j)$ is computed as:

$$v_t(j) = max_{i=1}^{N} v_{t-1}(i) a_{ij}\, b_j(o_t)$$

The three factors that are multiplied in the above equation for extending the previous paths to compute the Viterbi probability at time t are

$v_{t-1}(i)$, the previous Viterbi path probability from the previous time step

a_{ij}, the transition probability from previous state q_i to current state q_j

$b_j(o_t)$, the state observation likelihood of the observation symbol o_t given the current state j

Note that the Viterbi algorithm is identical to the forward algorithm except that it takes the max over the previous path probabilities whereas the forward algorithm takes the sum. Note also that the Viterbi algorithm has one component that the forward algorithm doesn't have: back pointers. The reason is that while the forward algorithm needs to produce an observation likelihood, the Viterbi algorithm must produce a probability and also the most likely state sequence. We compute this best state sequence by keeping track of the path of hidden states that led to each state and then at the end backtracking the best path to the beginning (the Viterbi backtrace).

Steps involved in the Viterbi recursion:

1. Initialization:

$$v_1(j) = \pi_j \, b_j, 1 \leq j \leq N$$

$$bt_1(j) = 0, 1 \leq j \leq N$$

2. Recursion:

$$v_t(j) = max_{i=1}^{N} v_{t-1}(i) a_{ij} \, b_j(o_t), 1 \leq j \leq N, 1 \leq t \leq N$$

$$bt_t(j) = argmax_{i=1}^{N} v_{t-1}(i) a_{ij} \, b_j(o_t), 1 \leq j \leq N, 1 \leq t \leq N$$

3. Termination:

The best score: $P *= max_{i=1}^{N} v_T(i)$

The start of backtrace: $qT *= argmax_{i=1}^{N} v_T(i)$

Recurrent Neural Networks:

Recurrent Neural Networks (RNNs) are frequently used to solve NLP tasks such as POS tagging. These networks process the input information sequentially, i.e., they perform repeated calculations for each token in the sequence considering the previous results.

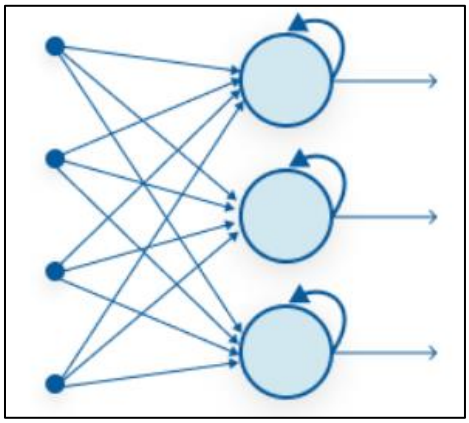

Fig 3: Recurrent Neural Network

Difference between RNNs and feed-forward networks:

Feed-forward networks such as CNN only consider the current input. So, they do not have any memory about the previous result. Therefore, it is difficult for them in predicting what comes next in a sequence. But they are able to classify an input with ease.

- RNNs differ in that they retain information about the input previously received.
- They are networks with feedback loops which allows the information to stay longer in a short-term memory. This sequential memory is preserved in the recurrent network's hidden state vector and represents the framework based on the past inputs and outputs.
- Unlike a feed-forward network, the same input may produce different outputs depending on the previous inputs.

So, in our case the input is in the form of X and Y where X contains all the information about the words in the form of embeddings and Y contains the

respective part of speech. The model that we build must be able to predict this part of speech based on the context of the sentence.

The three RNN models that we use to train the POS tagging problem are:

- Model 1: RNN with arbitrarily initialized, untrainable embeddings: In this model, the embedding weights are initialized arbitrarily. Further, the embeddings are frozen, i.e., the network is not allowed to train them.
- Model 2: RNN with arbitrarily initialized, trainable embeddings: In this the model is allowed to train the embeddings that are initialized arbitrarily.
- Model 3: RNN with trainable word2vec embeddings: In this model, word2vec word embeddings are used and the network is also allowed to train these embeddings.

One important point to make sure before training our model is that the input sequences must all be of the same length.

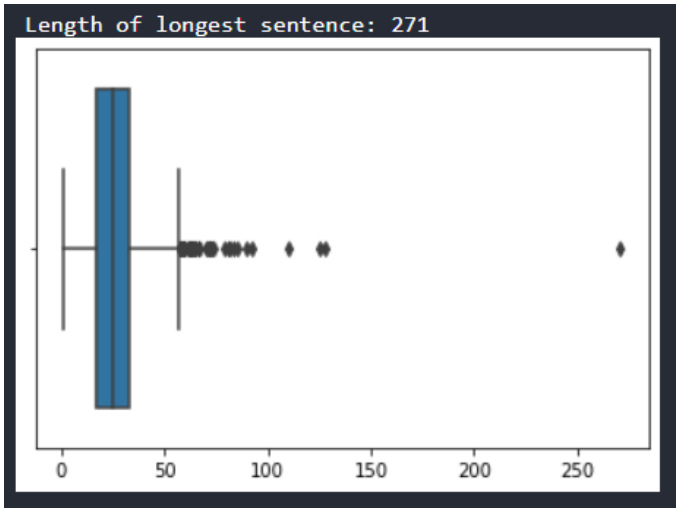

Fig 4: Box Whisker plot of sequence lengths

Therefore, all sentences are adjusted to the length of 100. If a sentence is smaller than 100 words, then the required zero padding is done in both X and Y.

Conditional Random Fields (CRF):

Conditional Random Fields is a class of discriminative models which makes predictions based on conditional probability that can be used for both classification and regression. CRFs are often used in tasks where context plays an important role. The prediction in CRF is modeled as a graphical model which implements dependencies between prediction. Some applications of CRFs include POS tagging, Named Entity Recognition (NER), gene finding, shallow parsing, object recognition and image segmentation in the field of computer vision.

The drawback using HMM for POS tagging is in its inability to deal with unknown words; proper names, acronyms, etc. As new common nouns and verbs are being introduced at a surprising rate, it would be beneficial to arbitrarily add features. Although we could try to hack the HMM to find ways to incorporate some of these, in general it's hard for generative models like HMMs to add arbitrary features directly into the model in a clean way. Conditional Independence (CI) asserted by Directed acyclic graphs as in case HMMs are not very natural.

One way to elegantly solve this problem is by looking at the neighbors. A Markov Random Field is a distribution that factorizes over an undirected graph such that its nodes represent the random variables x_i. The probability

of any configuration in a graph with n nodes is giving by the equation $p(x_1, x_2, ..., x_n) = \frac{1}{Z}\prod_{c \in C} \phi_c(x_c)$ where $\prod_{c \in C} \phi_c(x_c)$ is the product clique potentials, called the joint clique potential and these values depends on neighborhood values. Z in the above equation is the normalizer and is expressed as $Z = \sum_x \prod_{c \in C} \phi_c(x_c)$. C is the set of all maximal cliques. The normalizer is the sum of joint clique potential over all configurations.

The informal idea of CRF is to not make choices that do not matter.

$$p(y \mid x) = \frac{1}{Z}\prod_{c \in C} \phi_c(y_c, x)$$

$$where\ Z = \sum_y \prod_{c \in C} \phi_c(y_c, x)$$

The cliques are based on the structure in y.

Parts Of Speech (POS) Tagging:

Part-of-speech (POS) tagging is a popular Natural Language Processing process which refers to categorizing words in a text (corpus) in correspondence with a particular part of speech, depending on the definition of the word and its context.

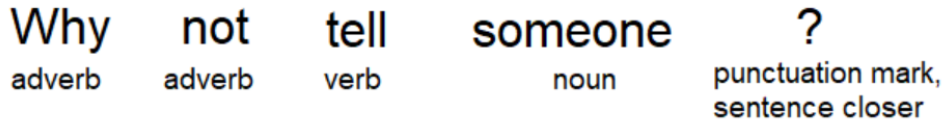

Fig. 5: A sentence with POS tags

In the above figure, we can see each word has its own lexical term written underneath, however, having to constantly write out these full terms when we perform text analysis can very quickly become cumbersome — especially as the size of the corpus grows. Then, we use a short representation referred to as “tags” to represent the categories.

Part-of-speech tags describe the characteristic structure of lexical terms within a sentence or text; therefore, we can use them for making assumptions about semantics. When we perform POS tagging, it’s often the case that our tagger will encounter words that were not within the vocabulary that was used. Consequently, augmenting your dataset to include unknown word tokens will aid the tagger in selecting appropriate tags for those words.

Using HMM for POS Tagging:

We have now understood that any HMM model is an FSM with hidden states. In our case, the unobservable states are the POS tags of a word. We may now think of these as hidden states since they are not directly observable from the corpus. Though a human may be capable of deciphering what POS applies to a specific word, a machine only sees the text, hence making it observable, and is unaware of whether that word POS tag is noun, verb, or something else which in-turn means they are unobservable.

Both the Markov Model and Hidden Markov model have transition probabilities that describe the transition from one hidden state to the next, however, the Hidden Markov Model also has something known as emission probabilities. The emission probabilities describe the transitions from the

hidden states in the model. We must remember the hidden states are the POS tags and the observable states are the words.

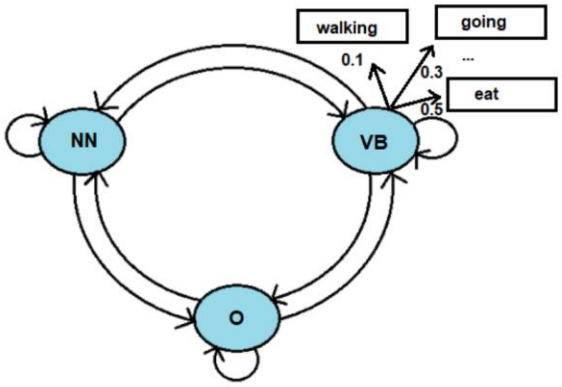

Fig. 6: Use of HMM for POS tagging

In Figure 4, we see that for the hidden VB state we have observable states. The emission probability from the hidden states VB to the observable word eat is 0.5 hence there is a 50% chance that the model would output this word when the current hidden state is VB.

CRF for POS tagging:

In CRFs, the input is a set of features (real numbers) derived from the input sequence using feature functions, the weights associated with the features (that are learned) and the previous label and the task is to predict the current label. The weights of different feature functions will be determined such that the likelihood of the labels in the training data will be maximized.

In CRF, a set of feature functions are defined to extract features for each word in a sentence. Some examples of feature functions are: is the first letter

of the word capitalised, what the suffix and prefix of the word, what is the previous word, is it the first or the last word of the sentence, is it a number etc. These set of features are called State Features**.** In CRF, we also pass the label of the previous word and the label of the current word to learn the weights. CRF will try to determine the weights of different feature functions that will maximise the likelihood of the labels in the training data. The feature function dependent on the label of the previous word is called the Transition Feature

In the case of POS tagging, the observed variables are words in sentences (represented as x). The unknown variables (represented as y) are the POS tags in the below equation.

$$p(y \mid x) = \frac{1}{Z} \prod_{c \in C} \phi_c (y_c, x)$$

$$where\ Z = \sum_{y} \prod_{c \in C} \phi_c (y_c, x)$$

About the dataset:

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.

For implementing our POS tagger, we use the treebank sentence tags of the NLTK package of python. This contains 95547 words along with their relevant POS tags in the form of a tuple.

Some other relevant information about data include:

```
# Importing libraries
import nltk
import numpy as np
import pandas as pd
import random
from sklearn.model_selection import train_test_split
import pprint, time
```

```
# reading the Treebank tagged sentences
nltk_data = list(nltk.corpus.treebank.tagged_sents(tagset='universal'))
```

```
# let's check some of the tagged data
print(nltk_data[1:5])
```

```
[[('Mr.', 'NOUN'), ('Vinken', 'NOUN'), ('is', 'VERB'), ('chairman', 'NOUN'), ('of', 'ADP'), ('Elsevier', 'NOUN'), ('N.V.',
'NOUN'), (',', '.'), ('the', 'DET'), ('Dutch', 'NOUN'), ('publishing', 'VERB'), ('group', 'NOUN'), ('.', '.')], [('Rudolph',
'NOUN'), ('Agnew', 'NOUN'), (',', '.'), ('55', 'NUM'), ('years', 'NOUN'), ('old', 'ADJ'), ('and', 'CONJ'), ('former', 'AD
J'), ('chairman', 'NOUN'), ('of', 'ADP'), ('Consolidated', 'NOUN'), ('Gold', 'NOUN'), ('Fields', 'NOUN'), ('PLC', 'NOUN'),
(',', '.'), ('was', 'VERB'), ('named', 'VERB'), ('*-1', 'X'), ('a', 'DET'), ('nonexecutive', 'ADJ'), ('director', 'NOUN'),
('of', 'ADP'), ('this', 'DET'), ('British', 'ADJ'), ('industrial', 'ADJ'), ('conglomerate', 'NOUN'), ('.', '.')], [('A', 'DE
T'), ('form', 'NOUN'), ('of', 'ADP'), ('asbestos', 'NOUN'), ('once', 'ADV'), ('used', 'VERB'), ('*', 'X'), ('*', 'X'), ('t
o', 'PRT'), ('make', 'VERB'), ('Kent', 'NOUN'), ('cigarette', 'NOUN'), ('filters', 'NOUN'), ('has', 'VERB'), ('caused', 'VER
B'), ('a', 'DET'), ('high', 'ADJ'), ('percentage', 'NOUN'), ('of', 'ADP'), ('cancer', 'NOUN'), ('deaths', 'NOUN'), ('among',
'ADP'), ('a', 'DET'), ('group', 'NOUN'), ('of', 'ADP'), ('workers', 'NOUN'), ('exposed', 'VERB'), ('*', 'X'), ('to', 'PRT'),
('it', 'PRON'), ('more', 'ADV'), ('than', 'ADP'), ('30', 'NUM'), ('years', 'NOUN'), ('ago', 'ADP'), (',', '.'), ('researcher
s', 'NOUN'), ('reported', 'VERB'), ('0', 'X'), ('*T*-1', 'X'), ('.', '.')], [('The', 'DET'), ('asbestos', 'NOUN'), ('fiber',
'NOUN'), (',', '.'), ('crocidolite', 'NOUN'), (',', '.'), ('is', 'VERB'), ('unusually', 'ADV'), ('resilient', 'ADJ'), ('onc
e', 'ADP'), ('it', 'PRON'), ('enters', 'VERB'), ('the', 'DET'), ('lungs', 'NOUN'), (',', '.'), ('with', 'ADP'), ('even', 'AD
V'), ('brief', 'ADJ'), ('exposures', 'NOUN'), ('to', 'PRT'), ('it', 'PRON'), ('causing', 'VERB'), ('symptoms', 'NOUN'), ('th
at', 'DET'), ('*T*-1', 'X'), ('show', 'VERB'), ('up', 'PRT'), ('decades', 'NOUN'), ('later', 'ADJ'), (',', '.'), ('researche
rs', 'NOUN'), ('said', 'VERB'), ('0', 'X'), ('*T*-2', 'X'), ('.', '.')]]
```

```
# split data into training and validation set in the ratio 95:5

train_set,test_set = train_test_split(nltk_data,train_size=0.95,test_size=0.05,random_state = 101)
```

```
# create list of train and test tagged words
train_tagged_words = [tup for sent in train_set for tup in sent]
test_tagged_words = [tup[0] for sent in test_set for tup in sent]
print(len(train_tagged_words))
print(len(test_tagged_words))
```

```
95547
5129
```

```
# let's check how many unique tags are present in training data
tags = {tag for word,tag in train_tagged_words}
print(len(tags))
print(tags)
```

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

```
# let's check how many words are present in vocabulary
vocab = {word for word,tag in train_tagged_words}
print(len(vocab))
```

```
12100
```

Models:

Solving the POS tagging problem using HMM, RNN and CRF, the following algorithms are proposed:

a) Using HMM:

1. Plain Vanilla Viterbi Algorithm
2. Viterbi Algorithm for OOV words
3. Viterbi Algorithm for OOV words with weighted transition probabilities
4. Rule based Viterbi for OOV
5. Modified Rule based Viterbi to return POS as Noun

b) Using RNN:

6. Vanilla RNN
7. Training RNN from scratch
8. Training RNN using pre-trained weights

c) Using CRF

Before going into the algorithms, let us see some important functions needed to be used as a foundation for upcoming algorithms.

To find the emission probability for a given word, we must be able to compute the word count given the tag. This will be a ratio of the word count given the tag given the total number of words having that tag. The function to do that is shown below.

```
# compute emission probability for a given word for a given tag
def word_given_tag(word,tag,train_bag= train_tagged_words):
    taglist = [pair for pair in train_bag if pair[1] == tag] #to find all words having that tag
    tag_count = len(taglist)
    #to find total occurences of a word belonging to any tag
    w_in_tag = [pair[0] for pair in taglist if pair[0]==word]
    word_count_given_tag = len(w_in_tag)

    return (word_count_given_tag,tag_count)
```

To compute the emission probability, we need to find out what is the conditional probability of finding the next tag given the previous tag. This will be a ratio of number of occurrences of that tag given the previous tag to the total occurrences of that tag. The function to do that is as follows

```
# compute transition probabilities of a previous and next tag
def t2_given_t1(t2,t1,train_bag=train_tagged_words):
    tags = [pair[1] for pair in train_bag] #finds all tags

    t1_tags = [tag for tag in tags if tag==t1]#finds a particular tag

    count_of_t1 = len(t1_tags)
    #finds a particular tag given the previous tag
    t2_given_t1 = [tags[index+1] for index in range(len(tags)-1)
                   if tags[index] == t1 and tags[index+1] == t2]

    count_t2_given_t1 = len(t2_given_t1)

    return(count_t2_given_t1,count_of_t1)
```

```
t2_given_t1('NOUN','DET')
```

```
(5284, 8281)
```

Once we compute the transition probabilities, we can create a TxT transition matrix from one tag to another. As described earlier, the column sum will be unity.

Code to calculate in brute force approach is shown below:

```
# creating t x t transition matrix of tags
# each column is t2, each row is t1
# thus M(i, j) represents P(tj given ti)

tags_matrix = np.zeros((len(tags), len(tags)), dtype='float32')
for i, t1 in enumerate(list(tags)):
    for j, t2 in enumerate(list(tags)):
        tags_matrix[i, j] = t2_given_t1(t2, t1)[0]/t2_given_t1(t2, t1)[1]
```

```
# convert the matrix to a df for better readability
tags_df = pd.DataFrame(tags_matrix, columns = list(tags), index=list(tags))
```

```
tags_df #table of transition probability
```

	X	DET	PRON	ADP	VERB	CONJ	.	ADV	NOUN	NUM	ADJ	PRT
X	0.076384	0.054742	0.055538	0.142584	0.203851	0.010662	0.163590	0.024984	0.062381	0.002864	0.017187	0.185232
DET	0.045405	0.005676	0.003744	0.009540	0.039850	0.000483	0.017993	0.012438	0.638087	0.022220	0.204323	0.000242
PRON	0.089969	0.009954	0.007657	0.022971	0.485452	0.005360	0.040965	0.034074	0.210949	0.006508	0.073124	0.013017
ADP	0.034427	0.324709	0.070031	0.016893	0.008340	0.000962	0.039025	0.014006	0.320967	0.062226	0.107024	0.001390
VERB	0.217506	0.134392	0.035786	0.092022	0.169249	0.005577	0.034934	0.081952	0.110070	0.022851	0.064988	0.030674
CONJ	0.008833	0.121339	0.058113	0.052534	0.156671	0.000465	0.034868	0.055323	0.349140	0.039981	0.118085	0.004649
.	0.026971	0.173335	0.066349	0.091342	0.089095	0.057538	0.093320	0.052324	0.222242	0.081003	0.043963	0.002427
ADV	0.023186	0.069891	0.014906	0.118582	0.343491	0.006956	0.137131	0.080490	0.031467	0.030474	0.129182	0.014243
NOUN	0.029175	0.012942	0.004607	0.176514	0.147667	0.042666	0.240604	0.017074	0.263564	0.009542	0.012248	0.043397
NUM	0.210542	0.003276	0.001489	0.036033	0.018761	0.013699	0.117332	0.002978	0.350208	0.184932	0.034247	0.026504
ADJ	0.021091	0.004943	0.000330	0.078267	0.011699	0.016971	0.063931	0.004778	0.699621	0.021256	0.066403	0.010710
PRT	0.013509	0.097858	0.017792	0.020099	0.405272	0.002306	0.043822	0.010214	0.247776	0.056672	0.083031	0.001647

Plain Vanilla Viterbi Algorithm:

The algorithm for this process is: Given a sequence of words, iterate through the sequence. For each word (starting from first word in sequence) calculate

the product of emission probabilities and transition probabilities for all possible tags. Based on that, we can assign the tag which has maximum probability. Then, we can move to the next word in sequence and repeat the process for whole corpus.

The algorithm code is as follows

```
# Vanilla Viterbi Algorithm
def Viterbi(words, train_bag = train_tagged_words):
    state = []

    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        for tag in T:
            if key == 0:
                #since word before start of a sent is unknown, prev tag is '.'
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)

        pmax = max(p)
        # getting state for which probability is maximum
        state_max = T[p.index(pmax)]
        state.append(state_max)
    return list(zip(words, state))
```

Testing the algorithm gives the following results:

```
# Let's test our Viterbi algorithm on a few sample sentences of test dataset

random.seed(1234)

# choose random 5 sents
rndom = [random.randint(1,len(test_set)) for x in range(5)]

# list of sents
test_run = [test_set[i] for i in rndom]

# list of tagged words
test_run_base = [tup for sent in test_run for tup in sent]

# list of untagged words
test_tagged_words = [tup[0] for sent in test_run for tup in sent]
```

```
# tagging the test sentences
start = time.time()
tagged_seq = Viterbi(test_tagged_words)
end = time.time()
difference = end-start

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

# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]

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

```
Time taken in seconds:  16.17097306251526
Vanilla Viterbi Algorithm Accuracy:  89.38053097345133
```

Though we get a good accuracy, we know that accuracy can be improved. The incorrectly labelled words on testing are:

```
# let's check the incorrectly tagged words
[j for i, j in enumerate(zip(tagged_seq, test_run_base)) if j[0] != j[1]]
```

```
[(('Contra', 'X'), ('Contra', 'NOUN')),
 (('command', 'VERB'), ('command', 'NOUN')),
 (('Honduras', 'X'), ('Honduras', 'NOUN')),
 (('Sandinista', 'X'), ('Sandinista', 'NOUN')),
 (('offensive', 'X'), ('offensive', 'NOUN')),
 (('rebel', 'X'), ('rebel', 'NOUN')),
 (('forces', 'VERB'), ('forces', 'NOUN')),
 (('Bucking', 'X'), ('Bucking', 'VERB')),
 (('drew', 'X'), ('drew', 'VERB')),
 (('Eveready', 'X'), ('Eveready', 'NOUN')),
 (('complaining', 'X'), ('complaining', 'VERB')),
 (('up', 'ADV'), ('up', 'PRT'))]
```

The error is high because of many unknown words present in the test data, but absent in the train data. So an efficient algorithm capable of handling Out of Vocabulary (OOV) words is essential.

Viterbi Algorithm for OOV words:

The algorithm remains the same. But in cases where the emission probability is 0, the product becomes zero. Thus, in those cases, we replace the probability score with the transition probability itself. The code to compute is shown below

```
# use transition probability of tags when emission probability is zero (in case of unknown words)

def Viterbi_1(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        p_transition =[] # list for storing transition probabilities
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)
            p_transition.append(transition_p)

        pmax = max(p)
        state_max = T[p.index(pmax)]


        # if probability is zero (unknown word) then use transition probability
        if(pmax==0):
            pmax = max(p_transition)
            state_max = T[p_transition.index(pmax)]

        else:
            state_max = T[p.index(pmax)]

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

Here, we need to find the maximum transition probability of all tags and replace the zeros with the appropriate value from the transition matrix. The results of this algorithm are:

```
# tagging the test sentences
start = time.time()
tagged_seq = Viterbi_1(test_tagged_words)
end = time.time()
difference = end-start

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

# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq)
print('Modified Viterbi_1 Accuracy: ',accuracy*100)
```

```
Time taken in seconds:  16.315754175186157
Modified Viterbi_1 Accuracy:  94.69026548672566
```

```
# let's check the tags for words
[j for i, j in enumerate(zip(tagged_seq, test_run_base)) if j[0] != j[1]]
```

```
[(('command', 'VERB'), ('command', 'NOUN')),
 (('Honduras', 'DET'), ('Honduras', 'NOUN')),
 (('Sandinista', 'VERB'), ('Sandinista', 'NOUN')),
 (('Eveready', 'VERB'), ('Eveready', 'NOUN')),
 (('*T*-252', 'VERB'), ('*T*-252', 'X')),
 (('up', 'ADV'), ('up', 'PRT'))]
```

We notice that a few nouns and verbs are getting correctly labelled and the accuracy has thus increased by approximately 5%.

Viterbi for OOV with weighted transition probabilities:

In this approach, we'll apply weights based on the probability of tag occurrence to the transition probabilities of tags and then use the resulting probability for predicting unknown words. This scheme will also take into account that some POS tags are more likely to occur as compared to others.

The list of tags along with their corr. probabilities are as follows

```
# lets create a list containing tuples of POS tags and POS tag occurance probability
tag_prob = []
total_tag = len([tag for word,tag in train_tagged_words])
for t in tags:
    each_tag = [tag for word,tag in train_tagged_words if tag==t]
    tag_prob.append((t,len(each_tag)/total_tag))

tag_prob
```

```
[('X', 0.06576867928872701),
 ('DET', 0.08666938784053921),
 ('PRON', 0.0273373313657153),
 ('ADP', 0.0978889970381069),
 ('VERB', 0.1351167488251855),
 ('CONJ', 0.02251248076862696),
 ('.', 0.11641391147812072),
 ('ADV', 0.031597015081582885),
 ('NOUN', 0.2862674913916711),
 ('NUM', 0.035145007169246546),
 ('ADJ', 0.06351847781719991),
 ('PRT', 0.0317644719352793)]
```

The modified algorithm is coded below

```
def Viterbi_1(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        p_transition =[] # list for storing transition probabilities

        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)

            # find POS tag occurance probability
            tag_p = [pair[1] for pair in tag_prob if pair[0]==tag ]

            # calculate the transition prob weighted by tag occurance probability.
            transition_p = tag_p[0]*transition_p
            p_transition.append(transition_p)

        pmax = max(p)
        state_max = T[p.index(pmax)]
```

```
        # if probability is zero (unknown word) then use weighted transition probability
        if(pmax==0):
            pmax = max(p_transition)
            state_max = T[p_transition.index(pmax)]

        else:
            state_max = T[p.index(pmax)]

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

The test results are:

```
# tagging the test sentences
start = time.time()
tagged_seq = Viterbi_1(test_tagged_words)
end = time.time()
difference = end-start

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

# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq)
print('Modified Viterbi_1 Accuracy: ',accuracy*100)
```

```
Time taken in seconds:  17.234552145004272
Modified Viterbi_1 Accuracy:  95.57522123893806
```

Thus, we see that we have got a much better accuracy by using weighted transition probabilties.

```
# let's check the incorrectly tagged words
[j for i, j in enumerate(zip(tagged_seq, test_run_base)) if j[0] != j[1]]
```

```
[(('command', 'VERB'), ('command', 'NOUN')),
 (('Sandinista', 'VERB'), ('Sandinista', 'NOUN')),
 (('Eveready', 'VERB'), ('Eveready', 'NOUN')),
 (('*T*-252', 'VERB'), ('*T*-252', 'X')),
 (('up', 'ADV'), ('up', 'PRT'))]
```

We have got a slightly higher accuracy with these weights, thanks to the weights from the original corpus.

Rule based Viterbi for OOV:

Rule based taggers are the most common approach for POS tagging. Here, we use certain regular expressions to identify the POS tags of unknown words occurring in the test data instead of simply using the transition probability. Some of these rules are based on English Grammar, while the other are to classify as nouns or numbers in case of proper nouns or words with numbers such as product model numbers. Note that here, we'll use the RexExp tagger of NLTK toolkit

```
# specify patterns for tagging
patterns = [
    (r'.*ing$', 'VERB'),              # gerund
    (r'.*ed$', 'VERB'),               # past tense
    (r'.*es$', 'VERB'),               # verb
    (r'.*\'s$', 'NOUN'),              # possessive nouns
    (r'.*s$', 'NOUN'),                # plural nouns
    (r'\*T?\*?-[0-9]+$', 'X'),        # X
    (r'^-?[0-9]+(.[0-9]+)?$', 'NUM'), # cardinal numbers
    (r'.*', 'NOUN')                   # nouns
]

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

The code with the modified rule is shown below:

```
#Backoff to rule based tagger in case unknown word is encountered.
def Viterbi_2(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)

        pmax = max(p)
        state_max = rule_based_tagger.tag([word])[0][1]
```

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


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

The results are really good for these rule-based taggers. It's shown below

```
# tagging the test sentences
start = time.time()
tagged_seq = Viterbi_2(test_tagged_words)
end = time.time()
difference = end-start

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

```
Time taken in seconds:  14.683353185653687
```

```
# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq)
print('Modified Viterbi_2 Accuracy: ',accuracy*100)
```

```
Modified Viterbi_2 Accuracy:  97.34513274336283
```

```
# let's check the incorrectly tagged words
[j for i, j in enumerate(zip(tagged_seq, test_run_base)) if j[0] != j[1]]
```

```
[(('command', 'VERB'), ('command', 'NOUN')),
 (('drew', 'NOUN'), ('drew', 'VERB')),
 (('up', 'ADV'), ('up', 'PRT'))]
```

We should notice the time taken to tag these words are also lesser than the previous approaches. And we only have 3 incorrectly labelled words, out of which two of them are nouns.

To encounter this problem, we'll perform another modification to existing rules.

Modified Rule based Viterbi to return POS as Noun:

We know that the rule-based tagger assigns 'NOUN' by default if word does not fall in any rule, to correct this let's assign the tags for any such word based purely on transition probability of tags. So, first we will modify the rule-based tagger to output 'NN' instead of 'NOUN' in case word does not satisfy any rules. We also observe that any capitalized word can still be defaulted as 'NOUN' so will add one more rule for that case.

```
# specify patterns for tagging
patterns = [
    (r'.*ing$', 'VERB'),              # gerund
    (r'.*ed$', 'VERB'),               # past tense
    (r'.*es$', 'VERB'),               # verb
    (r'.*\'s$', 'NOUN'),              # possessive nouns
    (r'.*s$', 'NOUN'),                # plural nouns
    (r'\*T?\*?-[0-9]+$', 'X'),        # X
    (r'^-?[0-9]+(.[0-9]+)?$', 'NUM'), # cardinal numbers
    (r'^[A-Z][a-z].*', 'NOUN'),       # NOUN
    (r'.*', 'NN')                     # default
]

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

Code to POS tag with modified rules is shown below

```
# modified Viterbi
def Viterbi_2(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        p_transition =[] # for storing transition probabilities
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)
```

```
            # find POS tag occurance probability
            tag_p = [pair[1] for pair in tag_prob if pair[0]==tag ]

            # calculate the transition prob weighted by tag occurance probability.
            transition_p = tag_p[0]*transition_p
            p_transition.append(transition_p)

        pmax = max(p)
        state_max = rule_based_tagger.tag([word])[0][1]
```

```
        # getting state for which probability is maximum
        if(pmax==0):
            state_max = rule_based_tagger.tag([word])[0][1] # assign based on rule based tagger

            # if unknown word does not satisfy any rule, find the tag with maximum transition probability
            if state_max == 'NN':
                pmax = max(p_transition)
                state_max = T[p_transition.index(pmax)]

        else:
             if state_max != 'X':
                # getting state for which probability is maximum
                state_max = T[p.index(pmax)]

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

Results show that tagging time as well as accuracy increases:

```
# tagging the test sentences
start = time.time()
tagged_seq = Viterbi_2(test_tagged_words)
end = time.time()
difference = end-start

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

```
Time taken in seconds:  14.343031883239746
```

```
# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq)
print('Modified Viterbi Algorithm Accuracy: ',accuracy*100)
```

```
Modified Viterbi Algorithm Accuracy:  98.23008849557522
```

We observe that much better accuracy is obtained now.

```
# let's check the incorrectly tagged words
[j for i, j in enumerate(zip(tagged_seq, test_run_base)) if j[0] != j[1]]
```

```
[(('command', 'VERB'), ('command', 'NOUN')), (('up', 'ADV'), ('up', 'PRT'))]
```

When we evaluate the same on the entire dataset, the accuracy, slightly decreases because of mild overfitting. But it should definitely be explored.

The results for running tagger on whole corpus is shown below.

```
test_tagged_words = [tup[0] for sent in test_set for tup in sent]
test_run_base = [tup for sent in test_set for tup in sent]
```

```
# tagging the test sentences
start = time.time()
tagged_seq = Viterbi(test_tagged_words)
end = time.time()
difference = end-start

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

# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq)
print('Modified Viterbi Algorithm Accuracy: ',accuracy*100)
```

```
Time taken in seconds:  652.3755822181702
Modified Viterbi Algorithm Accuracy:  94.22889452134919
```

```
# tagging the test sentences
start = time.time()
tagged_seq = Viterbi_2(test_tagged_words)
end = time.time()
difference = end-start

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

# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq)
print('Modified Viterbi Algorithm Accuracy: ',accuracy*100)
```

```
Time taken in seconds:  654.3451271057129
Modified Viterbi Algorithm Accuracy:  95.43770715539091
```

This tagger can be used for our own sample dataset whose tags are unknown. The procedure to do that is shown below.

```
f = open('C:/Users/ArrunPersonal/Codes/Semester5/Sem5_MISProject/SampleTextDoc.txt')
```

```
text = f.read()
```

```
sample_test_sent = text.splitlines()
```

```
f.close()
```

```
sample_test_sent[:-3]
```

```
["The UEFA Euro 2008 Final was the final match of Euro 2008, the thirteenth edition of the European Football Championship, UEFA's competition for national football teams. The match was played at Ernst-Happel-Stadion, Vienna, Austria, on 29 June 2008, and was contested by Germany and Spain. The sixteen-team tournament consisted of a group stage, from which eight teams qualified for the knockout phase. En route to the final, Germany finished second in Group B, with a defeat to Croatia and wins over Poland and Austria, after which they defeated Portugal and Turkey in the knockouts. Spain finished top of Group D with three wins, against Russia, Sweden and Greece, before defeating Italy on penalties in the quarter-final and a second victory over Russia in the semi-final.",
 '']
```

```
# list of untagged words
sample_test_words = [word for sent in sample_test_sent for word in sent.split()]
```

```
# tagging the test sentences
start = time.time()
sample_tagged_seq = Viterbi(sample_test_words)
end = time.time()
difference = end-start

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

```
Time taken in seconds:  43.91729426383972
```

```
sample_tagged_seq[:25]
```

```
[('The', 'DET'),
 ('UEFA', 'NOUN'),
 ('Euro', 'NOUN'),
 ('2008', 'NOUN'),
 ('Final', 'NOUN'),
 ('was', 'VERB'),
 ('the', 'DET'),
 ('final', 'ADJ'),
 ('match', 'NOUN'),
 ('of', 'ADP'),
 ('Euro', 'NOUN'),
 ('2008,', 'NOUN'),
 ('the', 'DET'),
 ('thirteenth', 'NOUN'),
 ('edition', 'NOUN'),
 ('of', 'ADP'),
 ('the', 'DET'),
 ('European', 'ADJ'),
 ('Football', 'NOUN'),
 ('Championship,', 'NOUN'),
 ("UEFA's", 'NOUN'),
 ('competition', 'NOUN'),
 ('for', 'ADP'),
 ('national', 'ADJ'),
 ('football', 'NOUN')]
```

We see that most words are getting tagged properly. Next, we'll look at POS tagging with RNN.

Before we go to RNN, we need to preprocess the treebank corpus as follows:

```
#Loading every sentence into X and Y variables
def load_sentences(tagged_sentences):
    X = []
    Y = []

    for sentence in tagged_sentences:
        X_sentence = []
        Y_sentence = []
        for word in sentence:
            X_sentence.append(word[0]) #word[0] contains the word
            Y_sentence.append(word[1]) #word[1] contains the corresponding tag

        X.append(X_sentence)
        Y.append(Y_sentence)

    return X,Y

X_train, Y_train = load_sentences(nltk_data)

print("Total number of tagged sentences: {}".format(len(X_train)))
```

```
Total number of tagged sentences: 3914

Example X:  ['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the', 'board', 'as', 'a', 'nonexecutive', 'dir
ector', 'Nov.', '29', '.']

Example Y:  ['NOUN', 'NOUN', ',', 'NUM', 'NOUN', 'ADJ', ',', 'VERB', 'VERB', 'DET', 'NOUN', 'ADP', 'DET', 'ADJ', '
NOUN', 'NOUN', 'NUM', '.']
```

```
from keras.preprocessing.text import Tokenizer

#Encoding X
word_tokenizer = Tokenizer()
word_tokenizer.fit_on_texts(X_train)

#Using the tokenizer to encode the input sequence
X_encoded = word_tokenizer.texts_to_sequences(X_train)

#Encoding Y
tag_tokenizer = Tokenizer()
tag_tokenizer.fit_on_texts(Y_train)
Y_encoded = tag_tokenizer.texts_to_sequences(Y_train)
```

```
#Look at first encoded data point
print("Original data:", "\n",)
print('X: ', X_train[0], '\n')
print('Y: ', Y_train[0], '\n')
print("Encoded data:", "\n")
print('X: ', X_encoded[0], '\n')
print('Y: ', Y_encoded[0], '\n')
```

```
Original data:
X: ['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the', 'board', 'as', 'a', 'nonexecutive', 'director', 'No
v.', '29', '.']

Y: ['NOUN', 'NOUN', '.', 'NUM', 'NOUN', 'ADJ', '.', 'VERB', 'VERB', 'DET', 'NOUN', 'ADP', 'DET', 'ADJ', 'NOUN', '
NOUN', 'NUM', '.']


Encoded data:
X: [5601, 3746, 1, 2024, 86, 331, 1, 46, 2405, 2, 131, 27, 6, 2025, 332, 459, 2026, 3]

Y: [1, 1, 3, 8, 1, 7, 3, 2, 2, 5, 1, 4, 5, 7, 1, 1, 8, 3]
```

This will give us the X and Y variables in terms of integers that can be used in the RNN model.

Next, we use word2vec model for further word embeddings, Currently, each word and the tags are encoded as integers. A more sophisticated technique is used to represent the input words (X) as word embeddings. We use a pre-trained word2vec model.

But, to represent each tag in Y, one hot encoding is done, since there are only 12 different tags in the dataset.

```
import gensim
from gensim.models import Word2Vec, KeyedVectors

#Path to word2vec model
path = '../input/googlenewsvectors/GoogleNews-vectors-negative300.bin'

word2vec = KeyedVectors.load_word2vec_format(path, binary=True)

#Assign word vectors from the pre trained word2vec model

EMBEDDING_SIZE  = 300  #Every word in word2vec model is represented using a
300 dimensional vector
VOCABULARY_SIZE = len(word_tokenizer.word_index) + 1
```

```
#Empty embedding matrix
embedding_weights = np.zeros((VOCABULARY_SIZE, EMBEDDING_SIZE))

#Word to index dictionary mapping
word2id = word_tokenizer.word_index

#Copying the vectors from word2vec model to the words present in our corpus
for word, index in word2id.items():
    try:
        embedding_weights[index, :] = word2vec[word]
    except KeyError:
        pass

print("Embeddings shape: {}".format(embedding_weights.shape))

from keras.utils.np_utils import to_categorical

# use Keras' to_categorical function to one-hot encode Y
Y_train = to_categorical(Y_train)

# print Y of the first output sequence
print("Shape of Y: {}".format(Y_train.shape))
```

```
Embedding's shape: (11388, 300)
Shape of Y: (3914, 100, 13)
```

Now, we need to split the X and Y arrays into train, test and validation sets.

```
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,
random_state=42)
X_train, X_validation, Y_train, Y_validation = train_test_split(X_train,
Y_train, test_size=0.2, random_state=4)
```

```
Training data :
Shape of input sequence: (2504, 100)
Shape of output sequence: (2504, 100, 13)
Validation data :
Shape of input sequence: (627, 100)
Shape of output sequence: (627, 100, 13)
Testing data :
Shape of input sequence: (783, 100)
Shape of output sequence: (783, 100, 13)
```

RNN Model 1 (Vanilla RNN):

```
from keras.utils.np_utils import to_categorical
from keras.models import Sequential
from keras.layers import Embedding
from keras.layers import Dense, Input
from keras.layers import TimeDistributed
from keras.layers import LSTM, GRU, Bidirectional, SimpleRNN, RNN

#Creating the architecutre

rnn_model = Sequential()

#Creating the embedding layer
#It is usually the first layer in any text problem
rnn_model.add(Embedding(input_dim = VOCABULARY_SIZE,
                        output_dim = EMBEDDING_SIZE,
                        input_length = MAX_SEQ_LENGTH,
                        trainable = False
))

#Add an RNN layer with 64 RNN cells
rnn_model.add(SimpleRNN(64,
              return_sequences = True  # True - Return whole sequence
))

#Add an output after each sequence
rnn_model.add(TimeDistributed(Dense(NUM_CLASSES, activation='softmax')))

rnn_model.compile(loss = 'categorical_crossentropy',
                  optimizer = 'adamax',
                  metrics = ['accuracy'])

#Summary of the model
rnn_model.summary()
```

Parameters in the embedding layer:

1. Vocabulary size: Number of unique words in data
2. Embedding size: Length of vector with which each word is represented
3. Input length: Length of input sequence
4. Trainable: False - Do not update/train the embeddings

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 100, 300)          3416400
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 100, 64)           23360
_________________________________________________________________
time_distributed (TimeDistri (None, 100, 13)           845
=================================================================
Total params: 3,440,605
Trainable params: 24,205
Non-trainable params: 3,416,400
_________________________________________________________________
```

Then we train our **model 1**,

```
rnn_training = rnn_model.fit(X_train, Y_train, batch_size=32, epochs=20,
validation_data=(X_validation, Y_validation))
```

```
Epoch 20/20
79/79 [==============================] - 5s 63ms/step - loss: 0.1625 - accuracy: 0.9496 - val_loss:
0.1734 - val_accuracy: 0.9462
```

The first RNN model gives us an accuracy of 94.62% with a training time of 94 seconds.

RNN Model 2 (RNN with trainable weights):

```
rnn_model = Sequential()

#Embedding layer
rnn_model.add(Embedding(input_dim = VOCABULARY_SIZE,
                        output_dim = EMBEDDING_SIZE,
                        input_length = MAX_SEQ_LENGTH,
                        trainable = True
))

#RNN layer
rnn_model.add(SimpleRNN(64,
              return_sequences = True
))

#Output after each sequence
rnn_model.add(TimeDistributed(Dense(NUM_CLASSES, activation='softmax')))

rnn_model.compile(loss = 'categorical_crossentropy',
                  optimizer = 'adamax',
                  metrics = ['accuracy'])
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 100, 300)          3416400
_________________________________________________________________
simple_rnn_2 (SimpleRNN)     (None, 100, 64)           23360
_________________________________________________________________
time_distributed_2 (TimeDist (None, 100, 13)           845
=================================================================
Total params: 3,440,605
Trainable params: 3,440,605
Non-trainable params: 0
_________________________________________________________________
```

Then we train our **model 2**,

```
rnn_training = rnn_model.fit(X_train, Y_train, batch_size=32, epochs=20,
validation_data=(X_validation, Y_validation))
```

```
Epoch 20/20
79/79 [==============================] - 9s 110ms/step - loss: 0.0343 - acc: 0.9913 - val_loss: 0.0
573 - val_accuracy: 0.9831
```

The second RNN model gives us an accuracy of 98.31% with a training time of 186 seconds.

RNN Model 3 (RNN with pre-trained weights):

```
rnn_model = Sequential()

#Embedding layer
rnn_model.add(Embedding(input_dim = VOCABULARY_SIZE,
                        output_dim = EMBEDDING_SIZE,
                        input_length = MAX_SEQ_LENGTH,
                        weights = [embedding_weights],
                        trainable = True
))

#RNN layer
rnn_model.add(SimpleRNN(64,
              return_sequences = True
))

#Output after each sequence
rnn_model.add(TimeDistributed(Dense(NUM_CLASSES, activation='softmax')))

rnn_model.compile(loss = 'categorical_crossentropy',
                  optimizer = 'adamax',
                  metrics = ['accuracy'])
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, 100, 300)          3416400
_________________________________________________________________
simple_rnn_3 (SimpleRNN)     (None, 100, 64)           23360
_________________________________________________________________
time_distributed_3 (TimeDist (None, 100, 13)           845
=================================================================
Total params: 3,440,605
Trainable params: 3,440,605
Non-trainable params: 0
_________________________________________________________________
```

Then we train our **model 3**,

```
rnn_training = rnn_model.fit(X_train, Y_train, batch_size=32, epochs=20,
validation_data=(X_validation, Y_validation))
```

```
Epoch 20/20
79/79 [==============================] - 8s 97ms/step - loss: 0.0235 - accuracy: 0.9935 - val_loss:
0.0366 - val_accuracy: 0.9887
```

The third RNN model gives us an accuracy of 98.87% with a training time of 192 seconds.

Training using CRF:

Reading the Corpus

```
[31] corpus = treebank.tagged_sents()
     tag_dict={}
     word_dict={}

     train_set,test_set = train_test_split(corpus,train_size=0.95,test_size=0.05,random_state = 101)

     for sent in train_set:
         for elem in sent:
             w = elem[0]
             tag= elem[1]
             if w not in word_dict:
                 word_dict[w]=0
             if tag not in tag_dict:
                 tag_dict[tag]=0
             word_dict[w]+=1
             tag_dict[tag]+=1

     print('Number of words(M): ',len(word_dict))
     print('Number of tags(N): ',len(tag_dict))
     print(tag_dict)
```

```
Number of words(M):  12100
Number of tags(N):  45
{'NNP': 8889, 'VBD': 2868, 'DT': 7750, 'NN': 12471, 'CC': 2151, 'MD': 896, 'RB': 2684, 'VB': 2447, '.': 3682, '-NONE-': 6264, 'VBG': 1380, 'JJ':
```

Feature extraction from the corpus

```
[32] def word2features(sent,i):
         word = sent[i][0]
         features ={
         'bias': 1.0,
         'word':word,
         'is_first': i == 0,
         'is_last': i == len(sent) - 1,
         'is_capitalized': word[0].upper() == word[0],
         'is_all_caps': word.upper() == word,
         'is_all_lower': word.lower() == word,
         'prefix-1': word[0],
         'prefix-2': word[:2],
         'prefix-3': word[:3],
         'suffix-1': word[-1],
         'suffix-2': word[-2:],
         'suffix-3': word[-3:],
         'prev_word': '' if i == 0 else sent[i - 1][0],
         'next_word': '' if i == len(sent) - 1 else sent[i + 1][0],
         'has_hyphen': '-' in word,
         'is_numeric': word.isdigit(),
         'capitals_inside': word[1:].lower() != word[1:]
         }

         return features
```

```
def sent2features(sent):
    return [word2features(sent,i) for i in range(len(sent))]

def sent2labels(sent):
    return [label for i,label in sent]

X_train=[sent2features(s) for s in train_set]
y_train=[sent2labels(s) for s in train_set]

X_test=[sent2features(s) for s in test_set]
y_test=[sent2labels(s) for s in test_set]
```

Training discriminative model

```
[34] crf = sklearn_crfsuite.CRF(
         algorithm='lbfgs',
         c1=0.1,
         c2=0.1,
         max_iterations=100,
         all_possible_transitions=True
     )

     crf.fit(X_train, y_train)

     # tagging the test sentences
     start = time.time()
     y_pred = crf.predict(X_test)
     labels=list(crf.classes_)
     score=metrics.flat_f1_score(y_test, y_pred, average='weighted', labels=labels)
     end = time.time()
     difference = end-start

     print("Time taken in seconds: ", difference)
     print("Accuracy:", score)
```

```
Time taken in seconds:  0.14121007919311523
Accuracy: 0.9673129601617199
```

Conclusion:

In this study we compared the performance of generative models (HMM), discriminative models (CRF) and Deep learning models (RNN) for the task of Parts-Of-Speech tagging on the treebank corpus dataset. RNN with trained weights performed the best in terms of accuracy whereas in terms of prediction time, CRF performed the fastest. HMM is approximately 100 times slower than CRF, but gives a higher accuracy. Thus, we encounter a tradeoff between time and accuracy and one must choose his model wisely.

References:

i) Speech and Language Processing. Daniel Jurafsky & James H. Martin.

ii) https://towardsdatascience.com/pos-tagging-using-crfs-ea430c5fb78b

iii) https://medium.com/analytics-vidhya/pos-tagging-using-conditional-random-fields-92077e5eaa31

iv) https://www.analyticsvidhya.com/blog/2021/02/markov-chain-mathematical-formulation-intuitive-explanation-applications/

v) https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

vi) https://towardsdatascience.com/pos-tagging-using-rnn-7f08a522f849