

Parts of Speech Tagging Using Dynamic Programming

— Final Project Presentation —

Instructor : Prof. Jonathan Mwaura

Team



Atharva Sapre

I had previously enrolled for NLP. The holistic nature of the course really helped me nurture my interest in this domain. Being currently enrolled for Algorithms, I was looking for a way to implement a greedy/dynamic approach to solve a real-world ML problem. After a fair amount of research, I found that the problem of POS tagging could be solved using a probabilistic and dynamic programming approach with the help of a Hidden Markov Model.



Anushka Patil

Speech Recognition and Machine Translation have always intrigued me and have been a few of the topics related to NLP that I have intended to work on. While reading about a few papers related to these topics I came across Parts of Speech Tagging which gives us an understanding of the structure of terms within corpus which can be used to make assumptions about the semantics. I thought of using one of the teachings from this course to solve a problem in my area of interest.



Sriram Hariharan

To me, the field of NLP has always been fascinating, since I started inclining towards ML and AI. This excitement and interest in learning about the process behind the screen pushed me to take up this project which aims at processing data, cleaning it, and applying algorithms to help the machine understand the text by deducing the POS represented by each word. Hence we dug deeper to understand a few solutions to find POS in text data and found that Viterbi uses a DP approach.



Pareekshit Reddy G

I came across parts of speech tagging while reading through articles on the Data science blog. It piqued my interest, and I decided to turn it into a project using my knowledge of algorithms. Annotating billion word document manually is impractical, so automatic tagging is used. Implementing this project with DP would improve my understanding of the subject and my confidence.

Table of Contents

- Introduction
- Problem Statement
- Dataset
- Methodology
- Comparison
- Conclusion and Future Scope

Introduction

- A POS tag is a tag that indicates the part of speech for a word.
- Part-of-Speech tagging in itself may not be the solution to any particular NLP problem. It is a pre-requisite to simplify a lot of different problems.
- POS tagging has uses in the following NLP tasks:
 - Text to Speech Conversion : We need to know which word is being used in order to pronounce the text correctly.
 - Word Sense Disambiguation : Word-sense disambiguation is identifying which sense of a word (that is, which meaning) is used in a sentence, when the word has multiple meanings.
 - Semantic Analysis : POS tagging is a vital step in parsing and semantic analysis - enabling efficient parsing algorithms.

Problem Statement

- The most fundamental models in the field of Natural Language Processing (NLP) are based on Bag of Words. These models, however, fall short of capturing the syntactic relationships between words.
- Building lemmatizers, which are tools for breaking down words to their basic forms, need POS Tagging as well.
- POS tagging is the process of marking up a word in a corpus to a corresponding part of a speech tag, based on its context and definition.
- This task is not straightforward, as a particular word may have a different part of speech based on the context in which the word is used.
- Our aim is to use the probabilistic POS tagging model - Hidden Markov Model to compute the joint probability of a set of hidden states (POS tags of words in corpus) given a set of observed states.
- Then we use the Viterbi algorithm, which is a dynamic programming algorithm for obtaining the maximum a posteriori probability estimate of the most likely sequence of hidden states.

Dataset

- This project makes use of 2 datasets collected from Wall Street Journal (WSJ).
- One data set (WSJ-2_21.pos) will be used for training. The training set will be used to create the emission, transmission and tag counts.
- The other (WSJ-24.pos) for testing.
- The tagged training data has been preprocessed to form a vocabulary (hmm_vocab.txt).

Methodology

- First we create a Baseline model (bruteforce) for POS Tagging.
- We compute a few dictionaries that will help you to generate the tables which will help in predicting the parts of speech tag of each word. They are as follows:
 - Transition counts - This dictionary computes the number of times each tag happened next to another tag.
 - Emission counts - This dictionary will compute the probability of a word given its tag.
 - Tag counts - The dictionary will store tag as the key and number of times each tag appeared as the value.
- The Baseline Model works as follows :
- Choose the most prevalent POS for that word in the training set as the part of speech for that word.
- Check to see if the actual POS of the word is the same each time you anticipate using the word's most frequent POS. In that case, the prediction was accurate.
- By dividing the number of correctly predicted words by the total number of words for which we predicted the POS tag, we determine the accuracy.

Methodology

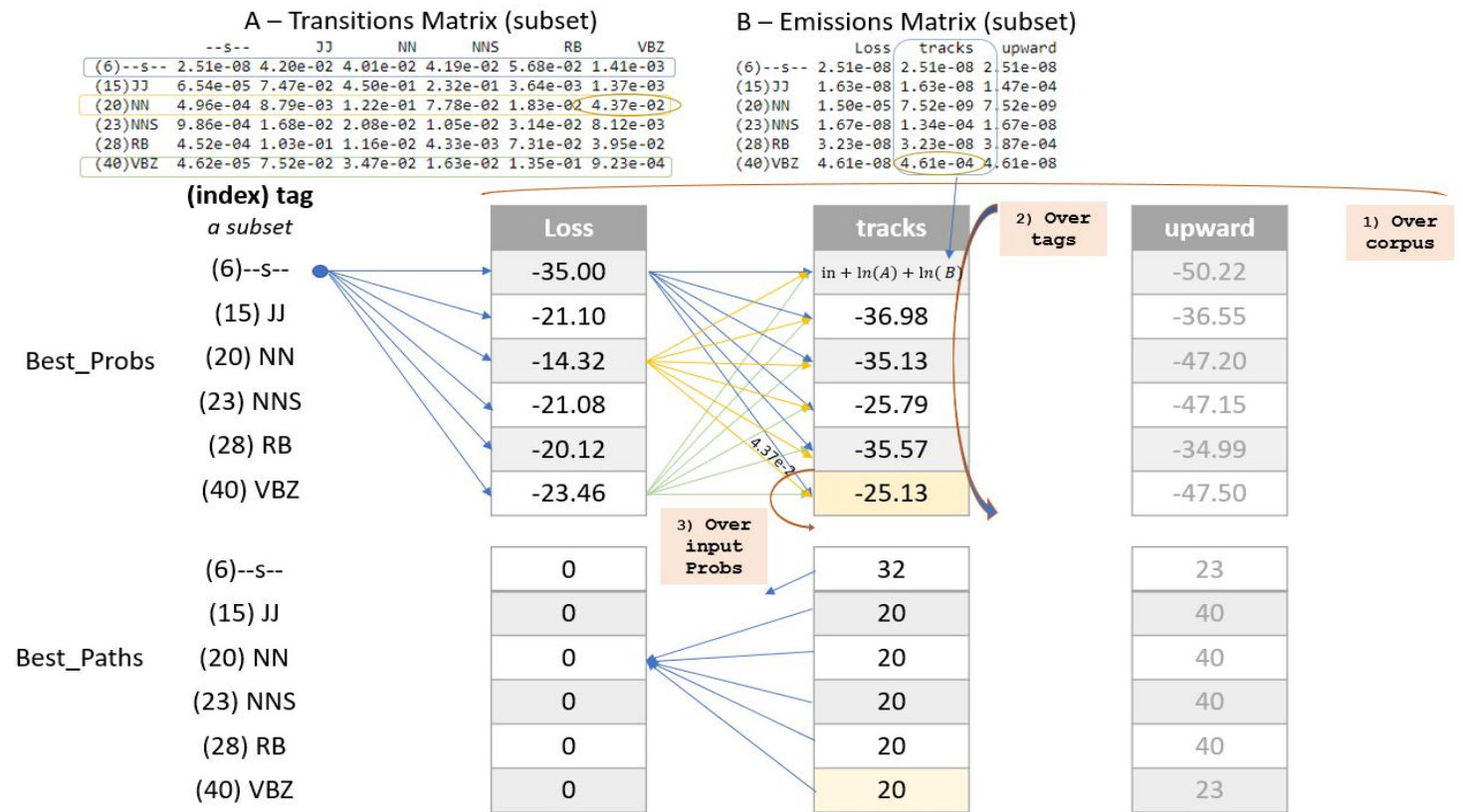
- 1) The next model we implemented was the Hidden Markov Model (HMM) using the viterbi algorithm.
- 2) The Markov Model contains a number of states and the probability of transition between those states. For this project the states are the parts-of-speech.
- 3) A Markov Model utilizes a transition matrix and an emission matrix.
- 4) We also generate 2 more matrices best_paths and best_probs using the above 2 matrices.
- 5) Best_probs and Best_paths matrices stores the probability of each path happening and the best paths up to that point.
- 6) HMM methodology:
 - a) Generation of Emission matrix and transition matrix.
 - b) Matrix initialization for best_paths and best_probs.
 - c) Run the viterbi forward algorithm to fill the matrices.
 - d) Run the viterbi backward algorithm to find the most probable sequence of POS tags.

Methodology

- After HMM, we implement the Viterbi algorithm which makes use of dynamic programming. We use transition and emission matrices from HMM to implement Viterbi.
- This is decomposed into 3 steps :
 - Initialization - In this part we initialize the best_paths and best_probabilities matrices that you will be populating in Viterbi feed_forward.
 - Viterbi Feed forward - Calculate the probability of each path happening and the best paths up to that point, at each step. For each word, compute a probability for each possible tag. Unlike the previous algorithm predict_pos (the 'warm-up' exercise), this will include the path up to that (word,tag) combination.
 - Viterbi Feed backward - The Viterbi backward algorithm gets the predictions of the POS tags for each word in the corpus using the best_paths and the best_probs matrices. Returns a list of predicted POS tags for each word in the corpus.
- By dividing the number of correctly predicted words by the total number of words for which we predicted the POS tag, we determine the accuracy.

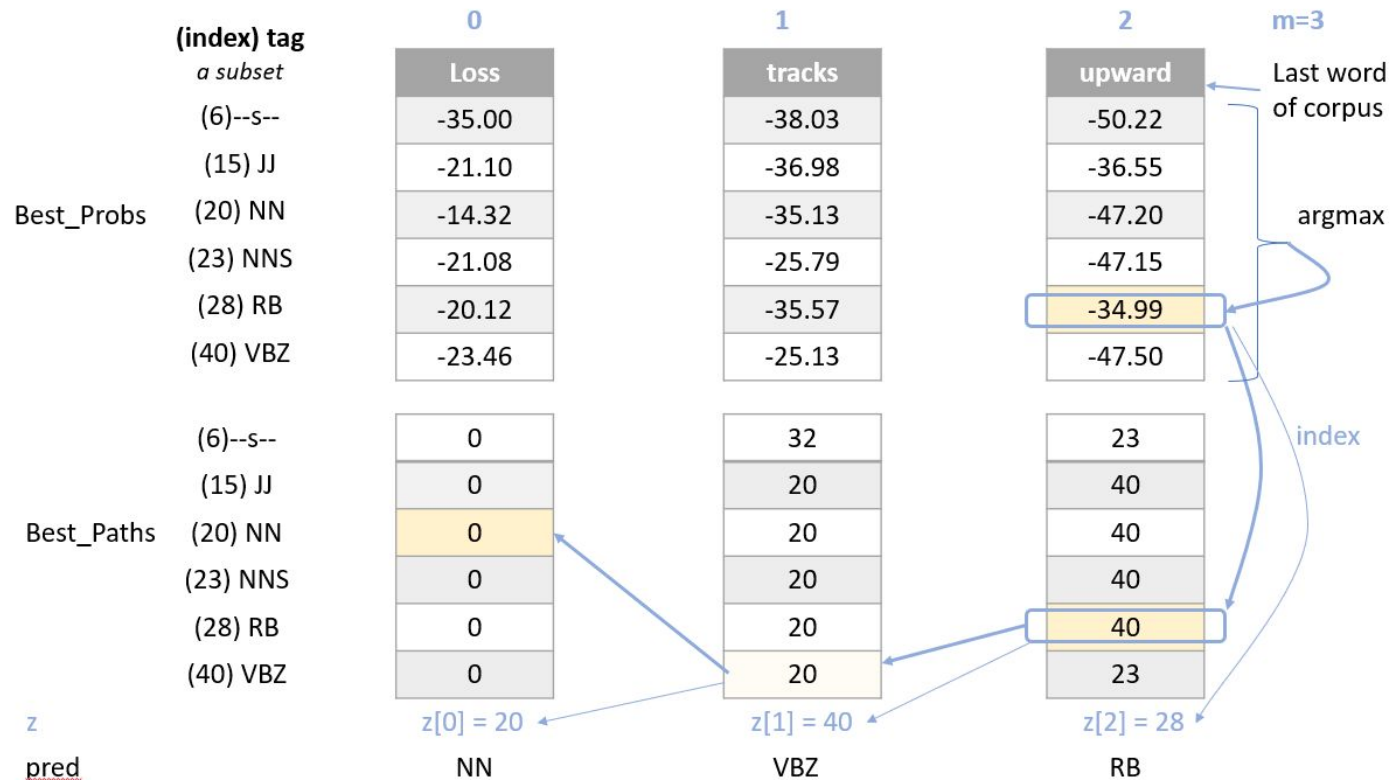
Methodology

Generation of Best paths and Best probabilities matrix (Viterbi Forward):



Methodology

Tracing the optimal path (Most probable sequence of POS tags)



Comparison

- The brute force (baseline) model for POS tagging generated an accuracy metric of 88.89%.
- The HMM Model with Viterbi algorithm using Dynamic Programming generated a prediction accuracy of 95.32% for the same dataset.
- By computing probabilities for each conceivable sequence of tags, the brute force approach to addressing this question entails selecting the sequence with the highest likelihood.
- However, All combinations of tags are possible. Hence, for a sentence with N words, and T tags the search space of possible state sequences X is $O(N^T)$.
- We will utilise the Viterbi method, a Dynamic Programming algorithm, to identify the most probable succession of hidden states that leads to a succession of observations, which requires $O(N^2 * T)$.

Conclusion

- A large problem can be solved using the dynamic programming technique by being divided into a number of smaller, simpler subproblems, each of which is solved only once, and then the solutions are stored in a memory-based data structure.
- The Viterbi approach is definitely more appropriate for POS tagging. This is because the applications of POS tagging involve working with complex and huge datasets, hence it's important to think about the space and time complexity of the program to ensure that we get timely results.
- The complexity of Brute force method is $O(N^T)$, where as the complexity of HMM with Viterbi using DP is $O(N^2 \cdot T)$. Generally there are 46 or more tags involved. Hence, $O(N^T)$ is definitely significantly greater than $O(N^2 \cdot T)$.
- Hence, for POS tagging on larger datasets, and looking at the prediction accuracy generated by both the Brute force and HMM with Viterbi methods, we can conclude that HMM using Viterbi with Dynamic Programming is definitely more efficient than the Brute Force method for POS tagging.

Future Work

- We can use the Baum-Welch algorithm that utilizes EM to give full conditional likelihood to the hidden parameters resulting in much better accuracy at the cost of time.
- We can work on researching more about the different ways of achieving POS tagging. One of which is Transformation-based Learning (TBL).
- TBL is a rule-based algorithm for automatic tagging of POS to the given text. It allows us to have linguistic knowledge in a readable form, transforms one state to another state by using transformation rules.
- Complexity in tagging is reduced because in TBL we make use of machine learned and human-generated rules. It is considered to be much faster than HMM tagger.
- Since, this is a sequence based modelling technique we can look into using RNNs and transformers to speed up the process of POS tagging and achieve higher predictive accuracy.

Thank You!

FINAL PROJECT REPORT

Parts of Speech Tagging using Dynamic Programming

Instructor: Prof. Jonathan Mwuara

Team: Atharva Vinay Sapre, Anushka Atul Patil, Pareekshit Reddy Gaddam, Sriram Hariharan
Neelakantan

YouTube Link: <https://youtu.be/CrhiP8a1S0c>

Github Link:

<https://github.com/atharvasapre/Parts-of-Speech-Tagging-using-Dynamic-Programming>

Introduction:

The ability of robots to mimic or improve human intelligence, such as reasoning and experience-based learning was possible using artificial intelligence (AI). AI has enabled machines to understand human language, speech, and images to help us process chunks of unstructured data and derive insights, deduct predictions, generate data, etc. One major branch of AI is Natural Language Processing(NLP) which studies how computers and human language interact, with a focus on how to train computers to process and analyze massive volumes of natural language data.

Primitive models of NLP used Bag of Words (BoW) to analyze unique word occurrences in text/document. The Bag of Words method fails to capture the syntactic relationship between words. Understanding the relationship between words is significant in order to interpret the context of the word in a sentence from a document. Like in any language, a sentence makes sense only when divided into parts of speech. Hence, finding parts of speech of a sentence being processed is imperative for deriving its intended meaning. Therefore, the Parts of Speech (POS) tagging method proved efficient in NLP. According to its definition and context, POS tagging assigns a word from a corpus to the appropriate part of a speech tag. Building parse trees, which are necessary for creating Named Entity Recognizers and discovering relationships between words, is made easier with the help of POS tags. Building lemmatizers, which are tools for breaking down words to their basic forms, need POS Tagging as well.

Context:

Anushka Patil -

In my undergraduate degree, I got to learn about Data Science and its applications in a variety of fields which really blew my mind. Speech Recognition and Machine Translation have always

intrigued me and have been a few of the topics related to Natural Language Processing that I have intended to work on. While reading about a few papers related to these topics I came across Parts of Speech Tagging. This basically involves categorizing words in a text with their parts of speech depending on the word definition and the context in which the word was used. POS tagging gives us an understanding of the structure of the terms within a dataset which in turn can be used to make assumptions about the semantics. Hence, I thought of using one of the teachings from this course (i.e Dynamic Programming) to solve a problem in my area of interest (POS tagging), which was the motivation behind this project topic for me.

Pareekshit Reddy -

As a Northeastern University Master of Data Science student, I was always inspired by the work being done in the field of deep learning, which included NLP. I came across this topic called parts of speech tagging while reading through some articles on the Data science blog one day. It piqued my interest, and I decided to turn it into a project using my knowledge of algorithms. So, POS tagging, also known as POS annotation, is the process of assigning a correct POS (part of speech) to each word in a sentence. Information retrieval, parsing, Text to Speech applications, information retrieval, and corpus linguistic study are among their applications. They are also an intermediate step for higher-level NLP tasks like parsing, semantic analysis, and translation, making POS tagging a necessary function for advanced NLP applications. Originally, tagging was done by humans, but because it takes so long, we can automate the process with an appropriate POS tag. Annotating a small corpus manually for use as training data in the development of a new automatic POS tagger is now a common practice. Annotating a billion-word document manually is impractical, so automatic tagging is frequently used. Implementing this project with Dynamic programming would definitely improve my understanding of the subject and my confidence while facing interviews.

Atharva Sapre -

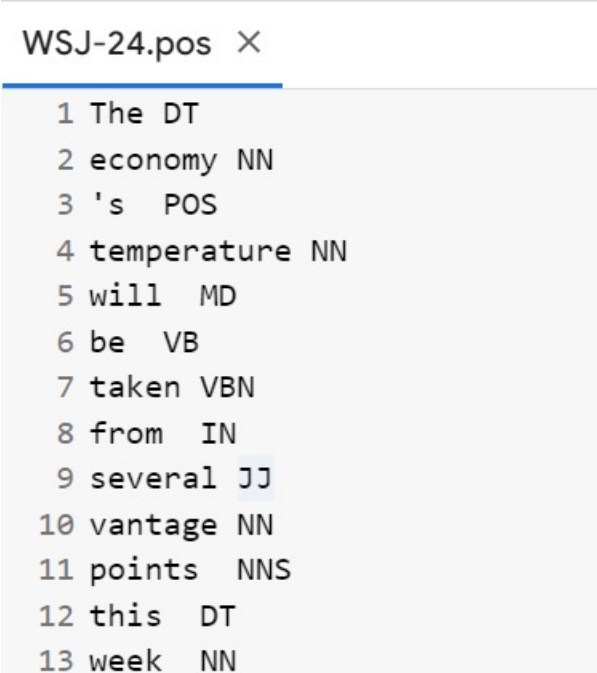
Being a student from a Data Science background, machine learning has become an integral part of my life. I had previously enrolled in the NLP course. The holistic and hierarchical nature of the course really helped me nurture my interest in this domain. I also plan to pursue my career in this field. Being currently enrolled in the Algorithms course, I was looking for a way to implement a greedy or a dynamic approach to solve a real-world machine learning problem. After a fair amount of research, I found that the problem of POS tagging could be solved using a probabilistic and dynamic programming approach with the help of a Hidden Markov Model. Of course, there are way more efficient approaches such as the use of RNNs for POS tagging. But I feel that implementing it in a traditional way with the incorporation of algorithmic concepts such as dynamic programming will help me strengthen my understanding of Algorithms, NLP as well as Probability.

Sriram Hariharan -

The sole purpose of Artificial Intelligence has been to mimic human behavior, understanding, and course of action and replicate it as closely as possible to help humans in performing significant tasks. One such important sub-branch of AI emerged 50 years ago, called Natural Language processing which was majorly based on linguistics. To me, the field of NLP has always been fascinating, from when I started inclining towards ML and AI. The idea of the computers trying to make sense of Text data, language, speech, etc, in the context of how humans would interpret, is something I wouldn't have imagined before getting to know about NLP. This excitement and interest in learning more about the process behind the screen pushed me to take up this project which aims at processing the data, cleaning it, and applying algorithms to help the machine understand the text by deducing the Parts of speech represented by each word. Therefore, we dug deeper to understand a few solutions to find POS in text data and found that Viterbi uses a Dynamic Program approach. Given that Dynamic programming is a widely used effective method for solving complex problems and I could use some practice to face coding interviews, I thought this project will be really useful to achieve both.

Analysis:**Dataset:**

We have used 2 datasets, one for training and another for testing from Wall Street Journal (WSJ). The data looks like as below:



WSJ-24.pos X

1	The	DT
2	economy	NN
3	's	POS
4	temperature	NN
5	will	MD
6	be	VB
7	taken	VBN
8	from	IN
9	several	JJ
10	vantage	NN
11	points	NNS
12	this	DT
13	week	NN

Each line has <word> , <POS Tag> format.

Brute Force Method:

To find the most probable of the hidden states, we can use the Brute force method by enumerating every possible path, and calculating the probability for every combination of hidden and observed events.

Some Definitions:

1. Transition Count: This contains the number of times each tag happened next to another tag, i.e., a tag at time t is the same as a tag at time $t-1$. We can use this to calculate probabilities of an event 's' at time t , given that the event 'e' happened at $t-1$, for such possible events.
2. Emission Count: This is a dictionary that has:
 - a. Key : (tag, word)
 - b. Value: The frequency(count) with which that pair appeared in your training set.
 This helps compute the probability of a particular output given a particular state. This probability links the hidden variable with the observed variable.

Algorithm:

Step 1: Read train data and create a vocabulary from the text.

Step 2: Create and initialize Transition Count and Emission Count Dictionaries with corresponding count values.

Step 3: For each word, calculate the maximum emission count and return the corresponding POS.

Step 4: Compute accuracy from the POS predicted from Step 3 and the original POS from training data.

emission examples:

(('DT', 'any'), 721)

(('NN', 'decrease'), 7)

(('NN', 'insider-trading'), 5)

transition examples:

(('--s--', 'IN'), 5050)

(('IN', 'DT'), 32364)

(('DT', 'NNP'), 9044)

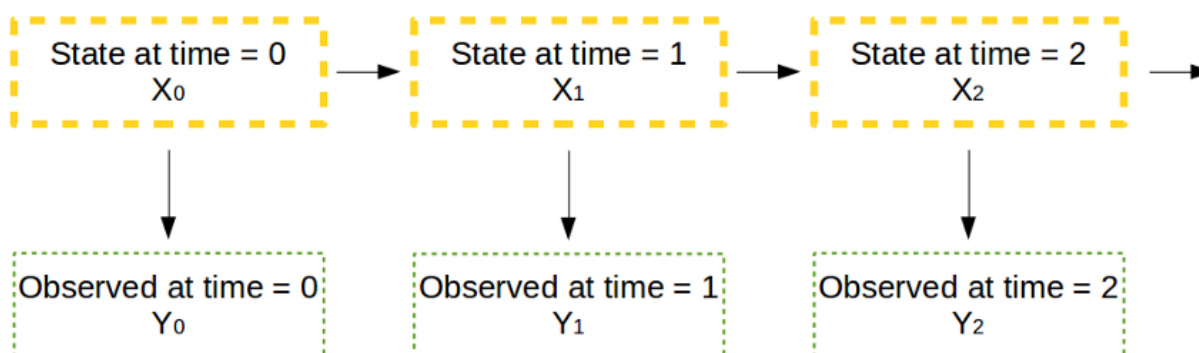
The above images show examples of what the emission and transition dictionaries look like.

The time complexity of this approach is $O(N^T)$, where N is the number of POS tags, T is the number of words in the text.

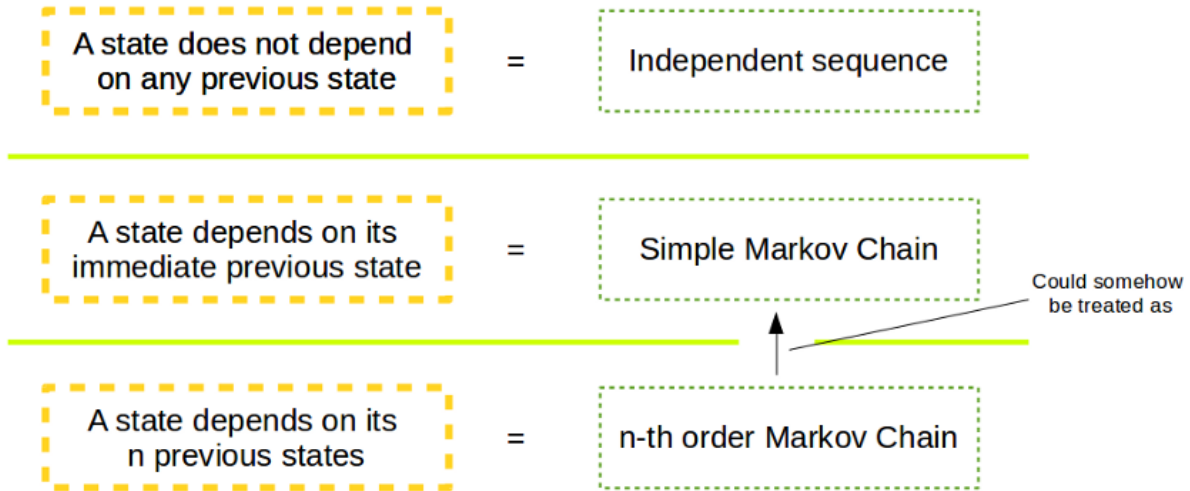
The Hidden Markov Model:

To tag parts of speech, we used the traditional Hidden Markov Model (HMM) sequence labeling technique. The HMM is a probabilistic sequence model that is an improvement over the Markov chain. When attempting to forecast the future of a Markov chain, the extreme assumption is that the current state is all that matters. It enables us to discuss both hidden and visible events, as well as part-of-speech tags, which we regard as incidental components in our probabilistic model. Observed events are words in the text that we see and POS is hidden from view. HMM is based on the assumption that the system under consideration is a chained Markov process with unobserved (hidden) states.

It is used to model a sequence as the result of a discrete stochastic process that moves through a number of 'hidden' states to the observer. Each observable piece of data is linked to a hidden reality. Each "hidden truth" is referred to as a "state variable," and each "observed data" is referred to as an "observed variable" in a formal HMM discussion.



The orange boxes above that are "connected" from a Markov chain. Every state in the chain is "connected" to the observed variable. For example, the state at time = 1 is dependent on the state at time = 0. Any state in the Markov model is only dependent on some of its previous states, which is a key premise that contributes to its simplicity.



The orange boxes above that are "connected" from a Markov chain. Every state in the chain is "connected" to the observed variable. For example, the state at time = 1 is dependent on the state at time = 0. Any state in the Markov model is only dependent on some of its previous states, which is a key premise that contributes to its simplicity.

$$X_{0:T}^* = \operatorname{argmax}_{X_{0:T}} P[X_{0:T}|Y_{0:T}]$$

The Viterbi algorithm:

The Viterbi algorithm is a dynamic programming algorithm that computes the best a posteriori probability estimate of the most probable sequence of hidden states, known as the Viterbi path. It views the computation as a probability product. This algorithm's time complexity is $O(N^2T)$.

Its purpose is to make an inference based on a trained model and some observed data. The following recursive formula is used to find the best set of states.

$$\mu(X_k) = \max_{X_{0:k-1}} P[X_{0:k}, Y_{0:k}] = \max_{X_{k-1}} \mu(X_{k-1})P[X_k|X_{k-1}]P[Y_k|X_k]$$

Let us substitute the value of $k=1,2,3$ and go over each possibility step by step:

$$\mu(X_0) = P[Y_0|X_0]P[X_0]$$

$$\mu(X_1) = \max_{X_0} \mu(X_0)P[X_1|X_0]P[Y_1|X_1]$$

$$\mu(X_2) = \max_{X_1} \mu(X_1)P[X_2|X_1]P[Y_2|X_2]$$

$$\mu(X_3) = \max_{X_2} \mu(X_2)P[X_3|X_2]P[Y_3|X_3]$$

- Given the initial observed data, the first line of the above figure produces a probability distribution for seeing different initial states.
- The second line selects the best initial state that maximizes the product of the right-hand side terms and leaves the initial state as a free parameter that can be determined in the next step.
- The third formula selects the best first state, leaving the second state to the fourth formula.
- We can Visualize this process with the diagrams called trellis.

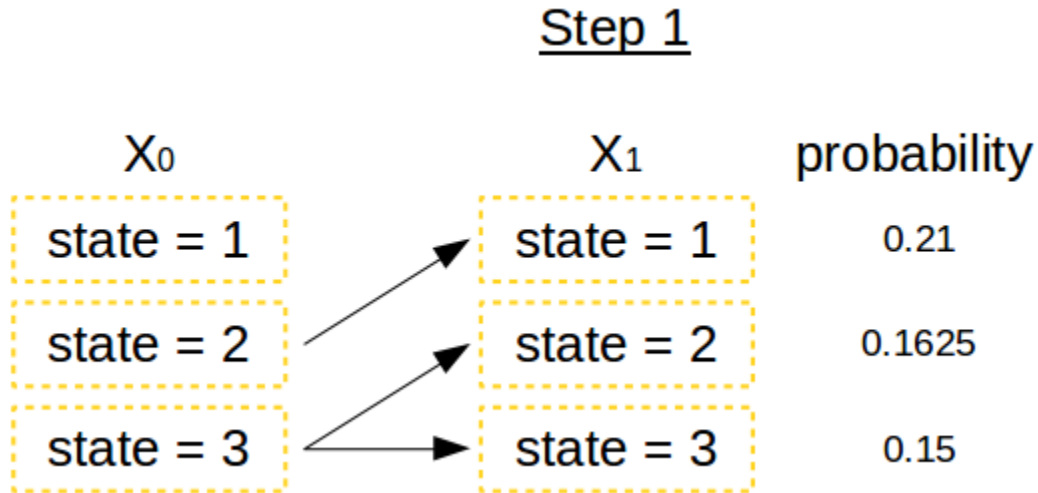
Algorithm:

1. Assume that each time step has only three possible states from which to choose.

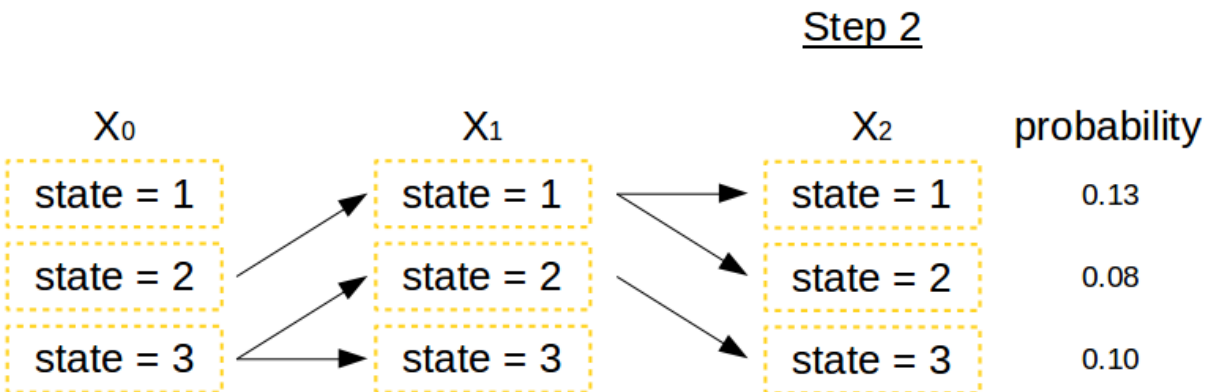
Step 0

X_0	probability
state = 1	$P[Y_0 X_0=1]P[X_0=1] = 0.4$
state = 2	0.35
state = 3	0.25

- Step 0 is simply a list of all possible states at time 0 and their probability value values; we do not choose which state to use at this point.



- The best possible initial state is chosen for each possible first state.
- In step 1, we iterate through all possible first states (state at time = 1) to find the best initial state (state at time = 0), as shown in the graphs above. The procedure is then repeated to complete step 2.



- We can now see some paths.
- If we stop at step 2, the most likely ending state is state = 1, and the remaining previous states can be retraced using the arrows, which are state 2 at time 0, state 1 at time 1, and state 1 at time 2. The second most likely path is 3-2-3, with 2-1-2 being the least likely. It is extremely unlikely that the path begins with state 1.
- The time complexity of the Viterbi algorithm is $O(N^2 \cdot T)$

We have compared this algorithm to the brute force method and we can see the results below:

	Brute Force Method	Viterbi algorithm
Accuracy (%)	88.89%	95.32%
Time Complexity	$O(N^T)$	$O(N^2 * T)$

Conclusion and Future Work:

Thus, we find out the likelihood of a given sequence of POS tags using the forward algorithm given the hidden states by adding the log values of their probabilities. Once we fill the entire best_probs matrix, we find the cell in the last column with a maximum value of likelihood and then use the best_paths matrix as a back pointer to traverse from the last word to the starting pointer using the backward algorithm. The Viterbi algorithm utilizes the concept of dynamic programming which makes this algorithm far more efficient than the traditional brute force method. The brute force method calculates the likelihood of each and every sequence of POS tags and thus requires a hefty time of $O(N^T)$ where N is the number of POS tags and T is the length of the sentence. On the contrary, the Viterbi algorithm eliminates the pathways with low likelihood and achieves a time complexity of $O(N^2 * T)$. To increase the accuracy further, we can use the Baum-Welch algorithm or sequential machine learning models like RNNS, LSTMS, and transformers.

Atharva Sapre:

My coursework in my bachelor's degree intrigued me to explore the world of Data Science. After taking a master's level course in NLP, I was really fascinated by how we can interpret and analyze natural human languages through algorithms to extract valuable information. Enrolling in the Algorithms course really helped me get better with coding as well as problem-solving skills. It helped me develop a different perspective on looking at problems and finding efficient ways to solve them. This project has played a major role in strengthening my grasp on concepts such as dynamic programming, data preprocessing, and python data manipulation. The hidden Markov Model involves rigorous probability calculations and thus, I also revised probability concepts and I'm way more confident working with probability than I previously was. Using dynamic programming for solving machine learning problems such as POS tagging was something I had never thought of, before. I found it challenging yet fulfilling. Being an aspiring

Data Scientist, my job would be to build efficient machine learning models. Thus, I'll be exploring more about how I can implement algorithmic concepts in machine learning that can help in model optimization and model tuning.

Pareekshit Reddy G:

As a data science and machine learning enthusiast, POS tagging project has given me a lot of insight into how dynamic programming can make a model work efficiently and accurately. I also learned about Markov models and expanded my knowledge of probability and statistics. Implementing the Viterbi algorithm for inducing dynamic programming part into the project improved my coding abilities and problem-solving abilities. Following this course, I am more motivated to conduct research in the field of deep learning. Now I understand how keyboards in electronic devices such as computers and smartphones auto-correct and predict the next likely word in a sentence as I type. This boosts my confidence in my career path and motivates me to contribute to this field. In the future, we plan to improve on this project. Since the Hidden Markov only considers the previous state to determine the probability of the current state, In the future we would like to experiment with considering two words prior or three words prior to calculate the current state probability and determine our model's accuracy. Now LSTMs and RNNs are being widely used in the market for the Parts of Speech Tagging as it is much more efficient and accurate, We also plan to implement these deep learning techniques in near future.

Sriram Hariharan:

For someone who is really interested in AI and applications such as NLP, this project has helped me learn a whole lot about the basic building blocks of any NLP system. The concepts such as Markov chains and Hidden Markov models are some of the significant topics in statistics and this project gave me an opportunity to learn them and apply my knowledge in algorithms to write a DP approach to solving the problem statement. Although I have read about the implementation of the Viterbi Algorithm for POS Tagging, I really didn't have a chance before to apply the concept through coding directly and witnessing the result practically. Having studied the Deep Learning course last semester, I saw this problem statement as one that could be solved only using Neural networks. But when I started researching, I found out that DP could be used to solve the problem, but a neural network approach is still better. In the future, we could try complementing the model with a dependency parser and using Deep neural network models such as RNNs or LSTM models for better results.

Anushka Patil:

Although earlier I have worked on few Machine Learning and Natural Language Processing based projects, from scratch as well as using builtin libraries, I had never used Dynamic Programming to implement these projects. This course definitely motivated me to research more blogs and papers to implement Machine Learning related algorithms with programming styles to improve computation time and space. Using the dynamic programming technique, a huge

problem can be broken down into several smaller, simpler subproblems, each of which is solved once, and the solutions are then stored in a memory-based data structure. For POS tagging, the Viterbi method is undoubtedly more appropriate. This is due to the fact that POS tagging applications require working with complicated and large datasets, thus it's crucial to consider the program's space and time complexity to guarantee that we acquire results quickly.

We can deduce that HMM using Viterbi with Dynamic Programming is certainly more effective than the Brute Force method for POS tagging after comparing the prediction accuracy produced by both the Brute force and HMM with Viterbi methods, as well as the space complexities required for both. Looking at the promising results after applying Dynamic Programming to NLP problem, I am excited to see more of its applications in other Data Science based problems, and I will certainly read more into it.

Appendix:

1. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.695.1924&rep=rep1&type=pdf>
2. [An Evaluation of POS Tagging for Tweets Using HMM Modelling](#)
3. <https://medium.com/analytics-vidhya/baum-welch-algorithm-for-training-a-hidden-markov-model-part-2-of-the-hmm-series-d0e393b4fb86>
4. <https://www.sciencedirect.com/topics/medicine-and-dentistry/hidden-markov-model>
5. <https://www.sciencedirect.com/topics/mathematics/viterbi-algorithm>
6. <https://www.mygreatlearning.com/blog/pos-tagging/>
7. <https://towardsdatascience.com/implementing-part-of-speech-tagging-for-english-words-using-viterbi-algorithm-from-scratch-9ded56b29133>

Project Code:

1. Brute Force Algorithm:

```
# -*- coding: utf-8 -*-
"""BruteForce

Automatically generated by Colaboratory.

Original file is located at
https://colab.research.google.com/drive/1n-X20v3gIEII_fEMy3_hiSW8mCeUzzjC
"""
```

```

# Importing packages and loading in the data set
import pandas as pd
from collections import defaultdict
import math
import numpy as np
import string

import string

# Punctuation characters
punct = set(string.punctuation)

# Morphology rules used to assign unknown word tokens
noun_suffix = ["action", "age", "ance", "cy", "dom", "ee", "ence", "er", "hood", "ion", "ism", "ist", "ity", "ling", "ment",
"ness", "or", "ry", "scape", "ship", "ty"]
verb_suffix = ["ate", "ify", "ise", "ize"]
adj_suffix = ["able", "ese", "ful", "i", "ian", "ible", "ic", "ish", "ive", "less", "ly", "ous"]
adv_suffix = ["ward", "wards", "wise"]

class bruteforce:
    def __init__(self, training, voc, test_corpus, test):
        # Punctuation characters
        self.punct = set(string.punctuation)

        # Morphology rules used to assign unknown word tokens
        self.noun_suffix = ["action", "age", "ance", "cy", "dom", "ee", "ence", "er", "hood", "ion", "ism", "ist", "ity", "ling", "ment",
"ness", "or", "ry", "scape", "ship", "ty"]
        self.verb_suffix = ["ate", "ify", "ise", "ize"]
        self.adj_suffix = ["able", "ese", "ful", "i", "ian", "ible", "ic", "ish", "ive", "less", "ly", "ous"]
        self.adv_suffix = ["ward", "wards", "wise"]

        with open(training, 'r') as f:
            training_corpus = f.readlines()

        with open(voc, 'r') as f:
            voc_1 = f.read().split("\n")

        # vocab: dictionary that has the index of the corresponding words
        self.vocab = {}

```

```

# Get the index of the corresponding words.
for i, word in enumerate(sorted(voc_l)):
    self.vocab[word] = i

#Reading the test corpus
with open(test_corpus, 'r') as f:
    self.y = f.readlines()

#corpus without tags, preprocessed
self._, self.prep = self.preprocess(self.vocab, test)

self.cnt = 0
for k,v in self.vocab.items():
    self.cnt += 1
    if self.cnt > 20:
        break

#create dictionaries
self.emission_counts, self.transition_counts, self.tag_counts = self.create_dictionaries(training_corpus, self.vocab)

# get all the POS states
self.states = sorted(self.tag_counts.keys())

accuracy_predict_pos = self.predict_pos(self.prep, self.y, self.emission_counts, self.vocab, self.states)
print(f"Accuracy of prediction using predict_pos is {accuracy_predict_pos:.4f}")

def get_word_tag(self, line, vocab):
    if not line.split():
        word = "--n--"
        tag = "--s--"
        return word, tag
    else:
        word, tag = line.split()
        if word not in vocab:
            word = self.assign_unk(word)
        return word, tag
    return None

def preprocess(self, vocab, data_fp):
    orig = []
    prep = []

```

```

with open(data_fp, "r") as data_file:
    for cnt, word in enumerate(data_file):
        if not word.split():
            orig.append(word.strip())
            word = "--n--"
            prep.append(word)
            continue
        elif word.strip() not in vocab:
            orig.append(word.strip())
            word = self.assign_unk(word)
            prep.append(word)
            continue
        else:
            orig.append(word.strip())
            prep.append(word.strip())

assert(len(orig) == len(open(data_fp, "r").readlines()))
assert(len(prepare) == len(open(data_fp, "r").readlines()))
return orig, prep

def assign_unk(self, tok):
    if any(char.isdigit() for char in tok):
        return "--unk_digit--"
    elif any(char in self.punct for char in tok):
        return "--unk_punct--"
    elif any(char.isupper() for char in tok):
        return "--unk_upper--"
    elif any(tok.endswith(suffix) for suffix in self.noun_suffix):
        return "--unk_noun--"
    elif any(tok.endswith(suffix) for suffix in self.verb_suffix):
        return "--unk_verb--"
    elif any(tok.endswith(suffix) for suffix in self.adj_suffix):
        return "--unk_adj--"
    elif any(tok.endswith(suffix) for suffix in self.adv_suffix):
        return "--unk_adv--"
    return "--unk--"

def create_dictionaries(self, training_corpus, vocab):
    emission_counts = defaultdict(int)
    transition_counts = defaultdict(int)

```

```

tag_counts = defaultdict(int)
prev_tag = '--s--'
i = 0
for word_tag in training_corpus:
    i += 1
    if i % 50000 == 0:
        print(f"word count = {i}")
    word,tag = self.get_word_tag(word_tag,vocab)
    transition_counts[(prev_tag,tag)] += 1
    emission_counts[(tag,word)] += 1
    tag_counts[tag] += 1
    prev_tag = tag

return emission_counts, transition_counts, tag_counts

def predict_pos(self, prep, y, emission_counts, vocab, states):
    """
    Input:
        prep: a preprocessed version of 'y'. A list with the 'word' component of the tuples.
        y: a corpus composed of a list of tuples where each tuple consists of (word, POS)
        emission_counts: a dictionary where the keys are (tag,word) tuples and the value is the count
        vocab: a dictionary where keys are words in vocabulary and value is an index
        states: a sorted list of all possible tags for this assignment
    Output:
        accuracy: Number of times you classified a word correctly
    """

    # Initialize the number of correct predictions to zero
    correct_pred=0

    # Get the (tag, word) tuples, stored as a set
    y_tup=set(emission_counts.keys())
    # Get the number of (word, POS) tuples in the corpus 'y'
    number=len(y)

    # Split the (word, POS) string into a list of two items
    for w, y_tup in zip(prepare,y):
        l=y_tup.split()

    # Verify that y_tup contain both word and POS

```

```

if(len(l)==2):
    # Set the true POS label for this word
    true=l[1]

    # If the y_tup didn't contain word and POS, go to next word
else:
    pass
# If the word is in the vocabulary...
final_ct=0
final_pos=""
if w in vocab:
    for pos in states:
        k=(pos,w)
        ### START CODE HERE (Replace instances of 'None' with your code) ###
        # define the key as the tuple containing the POS and word
        # check if the (pos, word) key exists in the emission_counts dictionary
        if k in emission_counts.keys():

            count=emission_counts[k]

            # get the emission count of the (pos,word) tuple

            # keep track of the POS with the largest count
            if count>final_ct:
                final_ct=count
                # update the final count (largest count)
                final_pos=pos
                # update the final POS

            # If the final POS (with the largest count) matches the true POS:
            if final_pos== true:
                correct_pred+=1
                # Update the number of correct predictions

        ### END CODE HERE ###
    accuracy = correct_pred/ number

return accuracy

obj= bruteforce(training="WSJ-2_21.pos", voc= "hmm_vocab.txt", test_corpus="WSJ-24.pos", test="test.words.txt")

```

2. Viterbi Algorithm

```
# -*- coding: utf-8 -*-
"""Viterbi Algorithm
Automatically generated by Colaboratory.
Original file is located at
    https://colab.research.google.com/drive/1iz9DaXevey0j3L9NnNGOwLPDZkIIMn2u
"""

# Importing packages and loading in the data set
import pandas as pd
from collections import defaultdict
import math
import numpy as np
import string

class HMM:
    def __init__(self, training, voc, test_corpus, test):

        # Punctuation characters
        self.punct = set(string.punctuation)

        # Morphology rules used to assign unknown word tokens
        self.noun_suffix = ["action", "age", "ance", "cy", "dom", "ee", "ence", "er", "hood", "ion", "ism", "ist", "ity", "ling", "ment",
                             "ness", "or", "ry", "sape", "ship", "ty"]
        self.verb_suffix = ["ate", "ify", "ise", "ize"]
        self.adj_suffix = ["able", "ese", "ful", "i", "ian", "ible", "ic", "ish", "ive", "less", "ly", "ous"]
        self.adv_suffix = ["ward", "wards", "wise"]

        with open(training, 'r') as f:
            training_corpus = f.readlines()

        with open(voc, 'r') as f:
            voc_l = f.read().split('\n')

        # vocab: dictionary that has the index of the corresponding words
        self.vocab = {}

        # Get the index of the corresponding words.
        for i, word in enumerate(sorted(voc_l)):
```



```

self.vocab[word] = i

#Reading the test corpus
with open(test_corpus, 'r') as f:
    self.y = f.readlines()

#corpus without tags, preprocessed
self._, self.prep = self.preprocess(self.vocab, test)

self.cnt = 0
for k,v in self.vocab.items():
    self.cnt += 1
    if self.cnt > 20:
        break

#create dictionaries
self.emission_counts, self.transition_counts, self.tag_counts = self.create_dictionaries(training_corpus, self.vocab)

# get all the POS states
self.states = sorted(self.tag_counts.keys())

#create transition matrix
alpha = 0.001
self.A = self.create_transition_matrix(alpha, self.tag_counts, self.transition_counts)

#creating emission matrix
self.B = self.create_emission_matrix(alpha, self.tag_counts, self.emission_counts, list(self.vocab))

#initilaize best_probs and best_paths matrices
self.best_probs, self.best_paths = self.initialize(self.states, self.tag_counts, self.A, self.B, self.prep, self.vocab)

# this will take a few minutes to run => processes ~ 30,000 words VITERBI FWD
self.best_probs, self.best_paths = self.viterbi_forward(self.A, self.B, self.prep, self.best_probs, self.best_paths, self.vocab)

#Running Viterbi Backward
self.pred = self.viterbi_backward(self.best_probs, self.best_paths, self.prep, self.states)

def get_word_tag(self,line, vocab):
    if not line.split():
        word = "--n--"
        tag = "--s--"
        return word, tag

```

```

else:
    word, tag = line.split()
    if word not in vocab:
        word = self.assign_unk(word)
    return word, tag
return None

def preprocess(self, vocab, data_fp):
    """
    Input:
        data_fp: file pointer to test data
        vocab: a dictionary where keys are words in vocabulary and value is an index

    Output:
        orig: original data with words and the assigned POS tags
        prep: Data without the POS tags for testing
    """
    orig = []
    prep = []

    with open(data_fp, "r") as data_file:

        for cnt, word in enumerate(data_file):
            if not word.strip():
                orig.append(word.strip())
                word = "--n--"
                prep.append(word)
                continue
            elif word.strip() not in vocab:
                orig.append(word.strip())
                word = self.assign_unk(word)
                prep.append(word)
                continue
            else:
                orig.append(word.strip())
                prep.append(word.strip())

    assert(len(orig) == len(open(data_fp, "r").readlines()))
    assert(len(prepare) == len(open(data_fp, "r").readlines()))

    return orig, prep

```

```

def assign_unk(self, tok):
    #Assign unk tags
    if any(char.isdigit() for char in tok):
        return "--unk_digit--"
    elif any(char in self.punct for char in tok):
        return "--unk_punct--"
    elif any(char.isupper() for char in tok):
        return "--unk_upper--"
    elif any(tok.endswith(suffix) for suffix in self.noun_suffix):
        return "--unk_noun--"
    elif any(tok.endswith(suffix) for suffix in self.verb_suffix):
        return "--unk_verb--"
    elif any(tok.endswith(suffix) for suffix in self.adj_suffix):
        return "--unk_adj--"
    elif any(tok.endswith(suffix) for suffix in self.adv_suffix):
        return "--unk_adv--"
    return "--unk--"

def create_dictionaries(self, training_corpus, vocab):
    """
    Input:
        prep: a preprocessed version of 'y'. A list with the 'word' component of the tuples.
        training_corpus: a corpus composed of a list of tuples where each tuple consists of (word, POS)
        vocab: a dictionary where keys are words in vocabulary and value is an index

    Output:
        emission_counts: a dictionary where the keys are (tag,word) tuples and the value is the count
        transition_counts: a dictionary where the keys are (prev_tag,curr_tag) tuples and the value is the count
        tag_counts: a dictionary where the keys are tags and the value is the count
    """
    emission_counts = defaultdict(int)
    transition_counts = defaultdict(int)
    tag_counts = defaultdict(int)

    prev_tag = '--s--'

    i = 0

    for word_tag in training_corpus:

        i += 1

```

```

        if i % 50000 == 0:
            print(f"word count = {i}")
        word,tag = self.get_word_tag(word_tag,vocab)
        transition_counts[(prev_tag,tag)] += 1
        emission_counts[(tag,word)] += 1
        tag_counts[tag] += 1
        prev_tag = tag

    return emission_counts, transition_counts, tag_counts

def create_transition_matrix(self, alpha, tag_counts, transition_counts):
    """
    Input:
        alpha: number used for smoothing
        tag_counts: a dictionary mapping each tag to its respective count
        transition_counts: transition count for the previous word and tag
    Output:
        A: matrix of dimension (num_tags,num_tags)
    """
    # Write your code here
    tags=sorted(tag_counts.keys())
    number=len(tags)
    A=np.zeros((number,number))
    prev_keys=set(transition_counts.keys())
    for i in range(number):
        for j in range(number):
            c=0
            k=(tags[i],tags[j])
            if k in transition_counts:
                c=transition_counts[k]
            prev_tag_count=tag_counts[tags[i]]

            A[i,j]=(c + alpha) / (prev_tag_count + alpha * number)

    return A

def create_emission_matrix(self, alpha, tag_counts, emission_counts, vocab):
    """
    Input:
        alpha: tuning parameter used in smoothing
        tag_counts: a dictionary mapping each tag to its respective count

```

```

    emission_counts: a dictionary where the keys are (tag, word) and the values are the counts
    vocab: a dictionary where keys are words in vocabulary and value is an index
Output:
    B: a matrix of dimension (num_tags, len(vocab))
'''
# Write your code here
number=len(tag_counts)
tags=sorted(tag_counts.keys())
tot_word_ct=len(vocab)
B = np.zeros((number, tot_word_ct))

keys=set(list(emission_counts.keys()))

for i in range(number):
    for j in range(tot_word_ct):
        c=0
        key=(tags[i],vocab[j])
        if key in emission_counts.keys():
            c=emission_counts[key]
        pos_tag_ct= tag_counts[tags[i]]

        B[i,j] = (c + alpha) / (pos_tag_ct + alpha * tot_word_ct)
    return B

def initialize(self, states, tag_counts, A, B, corpus, vocab):
    #helper function to initialize the matrix
    no_of_tags = len(tag_counts)
    best_probs = np.zeros((len(tag_counts), len(corpus)))
    best_paths= np.zeros((len(tag_counts), len(corpus)))

    s_idx = states.index("--s--")
    for i in range(no_of_tags):
        if A[s_idx,i]==0:
            best_probs[i,0]=float('-inf')
        else:
            best_probs[i,0]=math.log(A[s_idx,i])+math.log(B[i,vocab[corpus[0]]])
    return best_probs, best_paths

def viterbi_forward(self, A, B, test_corpus, best_probs, best_paths, vocab):
    '''
    Input:
        A, B: The transon and emission matrices respectively

```

```

test_corpus: a list containing a preprocessed corpus
best_probs: an initilized matrix of dimension (num_tags, len(corpus))
best_paths: an initilized matrix of dimension (num_tags, len(corpus))
vocab: a dictionary where keys are words in vocabulary and value is an index
Output:
    best_probs: a completed matrix of dimension (num_tags, len(corpus))
    best_paths: a completed matrix of dimension (num_tags, len(corpus))
'''

# Write your code here
num_tags = best_probs.shape[0]
for i in range(1, len(test_corpus)):
    for j in range(num_tags):
        best_prob_i = float('-inf')
        best_path_i = None
        for k in range(num_tags):
            prob = best_probs[k, i-1] + math.log(A[k, j]) + math.log(B[j, vocab.get(test_corpus[i])])
            if prob > best_prob_i:
                best_prob_i = prob
                best_path_i = k
        best_probs[j, i] = best_prob_i
        best_paths[j, i] = best_path_i
return best_probs, best_paths

def viterbi_backward(self, best_probs, best_paths, corpus, states):
    '''
    Input:
        corpus: a list containing a preprocessed corpus
        best_probs: an initilized matrix of dimension (num_tags, len(corpus))
        best_paths: an initilized matrix of dimension (num_tags, len(corpus))
        states: a list of all possible POS tags
    Output:
        best_probs: a completed matrix of dimension (num_tags, len(corpus))
        best_paths: a completed matrix of dimension (num_tags, len(corpus))
    '''
    m = best_paths.shape[1]
    tracker = [None] * m
    num_tags = best_probs.shape[0]
    best_p = -999999999999
    pos_pred = [None] * m
    for k in range(1, num_tags-1):
        if best_probs[k, m-1] < best_probs[k-1, m-1]:
            best_p = best_probs[k-1, m-1]

```

```

    tracker[m - 1] = int(k)
pos_pred[m-1] = states[k]
for i in range(m-1, 0, -1):
    tracker[i-1]=best_paths[int(tracker[i]), i]
    pos_pred[i-1] = states[int(tracker[i-1])]
return pos_pred

def compute_accuracy(pred, y):
    """
    Input:
        pred: a list of the predicted parts-of-speech
        y: a list of lines where each word is separated by a '\t' (i.e. word \t tag)
    Output:

    """
    num_correct = 0
    total = 0

    # Zip together the prediction and the labels
    for prediction, y in zip(pred, y):
        # Split the label into the word and the POS tag
        word_tag_tuple = y.split()

        # Check that there is actually a word and a tag
        # no more and no less than 2 items
        if len(word_tag_tuple)!=2: # complete this line
            continue

        # store the word and tag separately
        word, tag = word_tag_tuple

        # Check if the POS tag label matches the prediction
        if prediction == tag: # complete this line

            # count the number of times that the prediction
            # and label match
            num_correct += 1

        # keep track of the total number of examples (that have valid labels)
        total += 1

    p=(num_correct/total)*100

```

```

return print("The accuracy of the Viterbi Algorithm on test dataset is "+str(np.round(p,3))+"%")

obj= HMM(training="WSJ-2_21.pos", voc= "hmm_vocab.txt", test_corpus="WSJ-24.pos", test="test.words.txt")
compute_accuracy(obj.pred, obj.y)

"""**Matrix Visualization:**"""

# Try viewing emissions using sample words
cidx = ['725','adroitly','engineers', 'promoted', 'synergy']

# Get the integer ID for each word
cols = [obj.vocab[a] for a in cidx]

# Choose POS tags to show in a sample dataframe
rvals =['CD','NN','NNS', 'VB','RB','RP']

# For each POS tag, get the row number from the 'states' list
rows = [obj.states.index(a) for a in rvals]

# Get the emissions for the sample of words, and the sample of POS tags
B_sub = pd.DataFrame(obj.B[np.ix_(rows,cols)], index=rvals, columns = cidx )
print(B_sub)

# Visualizing transition matrix
print(f"A at row 0, col 0: {obj.A[0,0]:.9f}")
print(f"A at row 3, col 1: {obj.A[3,1]:.4f}")

print("View a subset of transition matrix A")
A_sub = pd.DataFrame(obj.A[30:35,30:35], index=obj.states[30:35], columns = obj.states[30:35] )
print(A_sub)

#Visualizing a subset of the best_probs matrix
print("View a subset of best_prob matrix A")
A_sub = pd.DataFrame(obj.best_probs[30:35,30:35], index=obj.states[30:35], columns = cidx )
print(A_sub)

#Visualizing a subset of the best_paths matrix
print("View a subset of best_prob matrix A")
A_sub = pd.DataFrame(obj.best_paths[30:35,30:35], index=obj.states[30:35], columns = cidx )
print(A_sub)

#Checking true and predicted POS tag values for first test sentence

```



```
print("Predicted POS sequence: ",obj.pred[0:13])
print("True POS sequence: ",["".join(reversed(i[-2:-4:-1])) for i in obj.y[0:13]])

# Commented out IPython magic to ensure Python compatibility.
# %%shell
# jupyter nbconvert --to html //Your notebook path file.ipynb
```