Senior Project

Implementing a Part of Speech Tagger with Hidden Markov Models

Sonya Fucci
Advisor: Deepak Kumar

Submitted in Partial Fulfillment of the Requirements of the BA in Computer Science
Bryn Mawr College
Spring 2020
Abstract

Part of speech tagging is a part of the natural language processing pipeline that classifies the part of speech of each token in a sentence. Several methods for part of speech tagging exist, and probability-based taggers are the most effective. A Hidden Markov Model is a probabilistic classification scheme that can be applied to part of speech tagging. This project implements a Hidden Markov Model that performs at a higher accuracy rate than the Natural Language Toolkit library implementation on the selected test corpus. Future explorations include a thorough analysis of out-of-vocabulary words and different methods of tagging them within a Hidden Markov Model implementation.
Acknowledgements

Deepak and Dianna: thank you for your unwavering support and confidence throughout my Bryn Mawr College career. I have had incredible opportunities through the Computer Science department, and I am immensely grateful for them. Thank you to Deepak for helping me reach the finish line and mentoring me through this project.

Robbie, Victor, and Francesca: they didn’t do anything to further my project, and at times had the opposite effect, but I love them anyway.

I must also thank my quarantine crew (Mom, Padraig, Victor, Nicki, and Ollie) for helping me de-stress and for their company during the final weeks. I must also thank the wine moms and several friends I hold dear: Carrie, Gabby, Auri, Eva, Amy, Arianna, and Helen. Nayem, of course, was so helpful, loving, and helped me grow in immeasurable ways throughout my college career.

And, of course, thank you to my wonderful family: Mom, Dad, Robbie, Victor, Francesca, Babulya, Dedulya, Padraig, and Sara.
1 Introduction

Part of Speech (POS) tagging is an important element of the Natural Language Processing pipeline [1]. The goal of part of speech tagging is to assign a part of speech to each word, or token, in a sentence. Part of speech tagging is performed once the text has been tokenized, or broken down into atomic structures. Each token is assigned a tag, or a part of speech classification that is part of a tagset of many tags. Upon successful part of speech tagging, the tags can facilitate further analyses on the corpus as a whole, with specific applications including speech synthesis and information retrieval [1].

I performed a comparative survey of different part-of-speech tagging techniques before implementing my own part-of-speech tagger using Hidden Markov Models. My Hidden Markov Model implementation yielded higher accuracy results than the standard library implementation, but was not as accurate as other probability-based taggers. These results shine light on the importance of the tagset being used and the ability to tag out-of-vocabulary words.

In this report I outline the part-of-speech tagging process and discuss several common tagging schemes. I then outline my Hidden Markov Model implementation and compare it to available library tagging schemes. I then outline future developments towards understanding the significance of out-of-vocabulary words in a text.

2 Background and Related Work

2.1 Part of Speech Tagging

Part of Speech Tagging is the process of classifying each token in a text with the correct part of speech. Part of speech taggers take in a sentence as input and produce the part of speech classification as output for each token in the sentence, as shown in Figure 1.

![Figure 1: Simple part of speech tagging pipeline.](image)

Consider the example in Figure 2. The input for part of speech tagging is the simple sentence “This is a cat.” The part of speech tagger assigns each token in the sentence a part of speech classification. A tagged sentence is represented as a list of tuples. Each tuple in the list contains a word and its corresponding part of speech classification, or tag. In the example in Figure 2, the word ‘This’ is a determiner, as indicated by the tag “DET”, ‘is’ is a verb, as indicated by the tag “VERB”, etc. The part of speech tagging in Figure 2 was performed with the Natural Language Toolkit’s, or NLTK’s, part of speech tagger [2]. NLTK is a robust Python library for natural language processing [2]. In addition to
providing standard natural language processing tools, including a part of speech tagger, NLTK provides access to several standard corpora often used in NLP [2, 3].

Figure 2: Example of part of speech tagging with input and output. The tagging was performed using NLTK’s default tagger [4].

Knowing a word’s part of speech is useful for many reasons. Many of these reasons stem from the fact that a word’s part of speech helps to convey the role that it plays in a sentence, and thus allows computers to derive meaning in further linguistic analyses [1]. Specific applications for part of speech tagging include speech synthesis, where the part of speech influences the pronunciation of certain words, and information retrieval [1].

2.1.1 Tagsets

As seen in Figure 2, a part of speech tagger outputs a part of speech classification for each word. These classifications are known as tags. For example, in Figure 2, the tags “DET”, for a determiner, “NOUN”, “VERB”, and “.” (for punctuation), are used. A tag is a part of a tagset, an unordered set of potential parts of speech a token could have [1].

Tagsets can vary greatly in size. The Brown University tagset, for instance, has 87 tags, Penn Treebank has 45 tags, and the universal tagset only has 12 elements [1, 5, 6]. The Penn Treebank tagset and the Universal tagset are seen in Table 1. When performing part of speech tagging, the choice of tagset is an important consideration. A larger tagset can convey more specific meaning and information about a token [1]. Additionally, non-universal tagsets are often tailored to a particular language and cannot be easily used on another language [5]. The Universal tagset, however, is intended to be general enough to be able to be used in several languages, and has significantly fewer tags, that are not language-specific, as a result[6].
Table 1: Penn Treebank Tagset, taken from [1] and the Universal Tagset [1, 6].

The chosen tagset has an impact on how the part of speech tagger classifies tokens. When comparing the Penn Treebank and Universal tagsets, it is useful to see how tagging differs for a simple sentence. Consider the sample sentence in Figure 3.

The great chef Bourdais has made a delicious meal.

Figure 3: Sample input sentence for part of speech tagging.

When tagged with NLTK’s default tagger using the Penn Treebank tagset, the sentence in Figure 3 has seven unique tags assigned to tokens out of ten tokens total. See Figure 4 for this classification.
The great chef Bourdain has made a delicious meal ‘.’

DT JJ NN NNP VBZ VBN DT JJ NN .

Figure 4: Tagged sentence in Figure 3 using the Penn Treebank Tagset and the NLTK default part of speech tagger.

When the same sentence is tagged with NLTK’s part of speech tagger but with the Universal tagset, the same ten tokens have only five unique tags, as shown below.

The great chef Bourdain has made a delicious meal ‘.’

DET ADJ NOUN NOUN VERB VERB DET ADJ NOUN .

Figure 5: Tagged sentence in Figure 3 using the Universal Tagset and tagged with the NLTK default part of speech tagger.

The differences in these two classifications lie in the way nouns and verbs are classified in the Penn Treebank tagset. There are four different classifications of nouns in the Penn Treebank tagset, including the distinction between singular and proper nouns. As a result, the proper noun “Bourdain” has a different tag than “chef” when using the Penn Treebank tagset in Figure 4, but when tagged with the Universal tagset they are both considered nouns. The same principle applies for verbs and punctuation.

2.2 Tagging Pipeline

As discussed above, which tagset to use is an important consideration for part of speech tagging and thus is included in the tagging pipeline. The Natural Language Toolkit’s gold standard part of speech tagger uses the Penn Treebank tagset by default, but a different tagset can be passed in as an optional parameter [3]. Figure 6 is an updated version of the part of speech tagging pipeline to reflect the importance of tagsets.

Figure 6: Example of part of speech tagging input and output including the tagset.

As sentences in a given text move through this pipeline, additional processing needs to be performed. The first necessary step in this processing is to perform tokenization, or splitting the corpus into individual tokens [5]. For the purposes of this project, a token is generally either a word or some form of punctuation, but this can make the task of tokenization challenging [5]. For this project, NLTK’s tokenizer will process the corpus prior to
part of speech tagging. The tokenized text is then passed into the Part-of-Speech tagger as input, as demonstrated by Figure 7.

![Figure 7: Example of part of speech tagging input and output including the tagset and tokenization.](image)

The main corpus for this project will be the Penn Treebank corpus as provided by the Natural Language Toolkit Library. The Penn Treebank corpus has a total of 100,676 tokens in 3,914 sentences that have already been pre-tagged with the Penn Treebank tagset with the optional parameter to modify the tagset. Of the 100,676 tokens in the corpus, they have tag distributions shown in Figure 8 (Universal tagset) and Figure 9 (Penn Treebank tagset). Nouns are the most common tag for both tagsets. The Universal tagset’s count of “NOUN” tags is over twice that of the most frequent tag in the Penn Treebank tagset, the singular noun ‘NN’. In the Universal tagset, nouns account for over 28% of tokens in the corpus, and singular nouns account for approximately 13% when using the Penn Treebank tagset.

![Figure 8: Distribution of tags for the Penn Treebank corpus using the Universal Tagset.](image)

2.3 Challenges of Part of Speech Tagging

Several processes in Natural Language Processing have to address ambiguity in a text [1]. Part of speech tagging is not exempt from this challenge. An ambiguous token in part of speech tagging is a token that can be classified with at least two different tags [1]. In English, this is apparent in words that have the same spelling but can have multiple parts of speech depending on the context [1]. For example the word “saw” can be either a noun or a verb. Humans disambiguate through context clues, and some cases can even be ambiguous to humans, but it is more challenging task for computers [1].
Figure 9: Distribution of tags for the Penn Treebank corpus using the Penn Treebank Tagset

Though most words in the English language are not ambiguous, words used more frequently are often more ambiguous [1]. In the Penn Treebank corpus, 28% of the tokens are tagged as nouns while using the universal tagset. 701 tokens in the Penn Treebank corpus have been tagged with at least two tags. While using the Penn Treebank tagset, the same corpus has 1187 ambiguous tokens, as demonstrated in Table 2.

<table>
<thead>
<tr>
<th>Tagset</th>
<th>Number of Ambiguous Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Penn Treebank</td>
<td>1187</td>
</tr>
<tr>
<td>Universal</td>
<td>701</td>
</tr>
</tbody>
</table>

Table 2: Number of ambiguous words in the Penn Treebank corpus by tagset.

In the Penn Treebank corpus, some words are “more ambiguous” than others. For the purposes of this project, an ambiguous token is a token that has been tagged with at least two different tags somewhere in the corpus. Some tokens are one tag the vast majority of its occurrences, whereas other tokens have fewer occurrences but have a more even distribution. The word “license” has one occurrence as a noun and one occurrence as a verb in the Penn Treebank corpus, for instance. On the other hand, the word “will” is tagged as a verb 278 times, but only once as a noun.

2.4 Tagging Algorithms

Part of speech tagging techniques generally fall into one of two categories. Algorithms are either rule-based or stochastic, or probability based. Rule-based part of speech tagging is one of the most easily understood techniques. Before tagging occurs, all of the classification rules are manually written [1]. On the other hand, probability-based tags classify words based on the likelihoods of a token having a certain tag. Rule-based taggers do not need any setup or training, aside from the hand-written rules. Stochastic taggers, however, generally go through a training and classification phase. The goal of this section is to explore some of the known techniques for part of speech tagging.
2.4.1 Rule-based Taggers

Rule-based taggers rely on a series of rules for what to do when it is classifying any given token [1]. These are highly tailored to a specific language. Sample rules for English are demonstrated in Table 3.

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Penn Treebank Tag</th>
<th>Universal Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>digits with or without a decimal</td>
<td>CD</td>
<td>NUM</td>
</tr>
<tr>
<td>is [Tt]he, [Aa], [Aa]n</td>
<td>DT</td>
<td>DET</td>
</tr>
<tr>
<td>ends with “able”</td>
<td>JJ</td>
<td>ADJ</td>
</tr>
<tr>
<td>ends with “ness”</td>
<td>NN</td>
<td>NOUN</td>
</tr>
<tr>
<td>ends with “ly”</td>
<td>RB</td>
<td>ADV</td>
</tr>
<tr>
<td>ends with “s”</td>
<td>NNS</td>
<td>NOUN</td>
</tr>
<tr>
<td>ends with “ing”</td>
<td>VBG</td>
<td>VERB</td>
</tr>
<tr>
<td>ends with “ed”</td>
<td>VBD</td>
<td>VERB</td>
</tr>
<tr>
<td>ends with “es”</td>
<td>VBZ</td>
<td>VERB</td>
</tr>
<tr>
<td>ends with “ould”</td>
<td>MD</td>
<td>VERB</td>
</tr>
<tr>
<td>ends with “’s”</td>
<td>NN</td>
<td>NOUN</td>
</tr>
<tr>
<td>else</td>
<td>NN</td>
<td>NOUN</td>
</tr>
</tbody>
</table>

Table 3: Regular expressions used in a rule-based tagger, compiled from [7] and [4] with tagset conversion assistance from [6].

There are a several implementations for rule-based taggers. Natural Language Toolkit allows the user to manually enter rules matching part of speech tags to regular expressions [4]. Jurafsky and Martin discuss a rule-based tagging pipeline that goes through two phases. The first phase determines all of the potential tags for a token, and the second phase consults more rules to disambiguate and assign a classification [1].

Some classification schemes revert to rule-based methods for handling unknown words. It is impossible to build a part of speech tagger that has every word in a language, as language is constantly evolving [1]. NLTK allows the user to add a default tag for a token when the original classification scheme cannot resolve its part of speech. This is not an ideal method for taggers that have the ability to consider a token in the context of the sentence [4]. In fact, morphemes, or the morphological structure of a token, can often be used to speculate at the unknown token [1].

2.4.2 Unigram Taggers

Stochastic taggers use probabilities to perform part-of-speech tagging. They often require a training phase prior to classifying a text [1].

One of the stochastic taggers is a Unigram tagger. NLTK’s Unigram tagger takes a tagged training dataset as input [7]. The Unigram Tagger then tags a token based on the most frequent tag for that token in its training set [7]. The Unigram tagger has thus been referred to as the “lookup tagger,” as the most frequent tag is calculated for each unique token during the training phase [4].
2.4.3 Hidden Markov Model Taggers

Hidden Markov Models (HMMs) are effective at solving classification problems probabilistically. Within the context of part of speech tagging, HMMs classify a token’s part of speech as one tag in the provided tagset [1]. This derivation of HMMs is closely based on [1]. It frames part of speech tagging as a maximization problem, where there are two critical sequences. The first sequence is the sentence, or sequence of tokens $w_1, w_2, ..., w_n$ the HMM will classify, also referred to as $w^n_1$ [1]. The second sequence is a set of corresponding tags for each token, which is referred to as $t^n_1$ [1]. The probability of sequence $w^n_1$ having classification $t^n_1$ is $P(t^n_1|w^n_1)$ [1]. Using these definitions, we can optimize this probability by choosing the tag sequence that has the highest probability.

Find $t^n_1$ such that $P(t^n_1|w^n_1)$ is maximized. (1)

Note that this is an approximation of the correct tag sequence [1]. While this statement of the problem is logical, it is unclear how to actually calculate $P(t^n_1|w^n_1)$. To derive a more computable formula, Bayes’ rule can be applied to the probability in (1), expanding the probability to the formula below [1].

Find $t^n_1$ such that \[
\frac{P(w^n_1|t^n_1)P(t^n_1)}{P(w^n_1)}
\]

is maximized. (2)

The formula in (2) can be simplified by removing the bottom term, $P(w^n_1)$. This value does not change during the classification process, as all of the potential tag sequences will be tagging the same sequence of words. The formula above is thus reduced to:

Find $t^n_1$ such that $P(w^n_1|t^n_1)P(t^n_1)$ is maximized. (3)

The model thus needs to find a way to calculate $P(w^n_1|t^n_1)$ and $P(t^n_1)$ for each potential sequence. This is a challenging task, and two key assumptions need to be made to make this model calculable. The first assumption regards $P(w^n_1|t^n_1)$. To simplify this term, we assume that the chance of encountering word $w_i$ only depends on the corresponding tag $t_i$. Using this first assumption, we approximate $P(w^n_1|t^n_1)$ as follows.

$$P(w^n_1|t^n_1) \approx \prod_{i=1}^{n} P(w_i|t_i)$$ (4)

The second assumption, also known as the Bigram or Markov assumption, simplifies the term $P(t^n_1)$ for computability. The Markov assumption states that the probability of a tag, $P(t_i)$, only depends on the tag immediately preceding it. In other words,

$$P(t_1, t_2, ..., t_i) = P(t_i|t_{i-1})$$ (5)
The Markov assumption simplifies our calculation for $P(t^n_1)$, defining its calculation as shown below.

$$P(t^n_1) \approx \prod_{i=1}^{n} P(t_i|t_{i-1})$$

Equation (6)

Thus we can substitute the terms in (3) with those derived in (4) and (6), reducing the original classification task to the following [1].

$$\text{Find } t^n_1 \text{ such that } \prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1}) \text{ is maximized.}$$

Equation (7)

The maximization defined in (7) is the model that HMMs use to perform a classification. The first term, $P(w_i|t_i)$ is known as the emissions probability [1]. The second term, $P(t_i|t_{i-1})$ is the transition probability, or the probability that $t_{i-1}$ is followed by $t_i$ [1]. In implementation, these probabilities can be approximated with a large training corpus. The details of the training phase and classification algorithm are discussed in section 3.

2.4.4 Comparing Taggers

I will do a comparison of the accuracies of the part of speech taggers presented on two different tagsets. The two tagsets for comparison are the 12-tag Universal tagset and the 45-tag Penn Treebank tagset shown in Table 1. To ensure an accurate comparison, I will be using the Penn Treebank corpus accessed via NLTK.

The following table charts the accuracies of NLTK’s rule-based, unigram, and HMM taggers on a text. As seen above, the Unigram tagger and HMM tagger each require a training phase before tagging. For these two taggers, the first 80% of sentences in the Penn Treebank corpus are the training data. As it follows, the test set for all three taggers is the remaining 20% of sentences in the corpus. The rule-based tagger, which requires no training phase, classifies tokens based on the regular expression rules in Table 3. The accuracies of each of these part-of-speech tagging methods are shown below.

<table>
<thead>
<tr>
<th>Tagger</th>
<th>Penn Treebank Tagset</th>
<th>Universal Tagset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regex</td>
<td>33.75%</td>
<td>37.75%</td>
</tr>
<tr>
<td>Unigram</td>
<td>86.08%</td>
<td>87.92%</td>
</tr>
<tr>
<td>NLTK’s HMM</td>
<td>36.47%</td>
<td>51.37%</td>
</tr>
</tbody>
</table>

Table 4: Results from all part of speech taggers described above on the last 20% of sentences on the Penn Treebank corpus.

As expected, there is higher accuracy when the Universal tagset is used. The Unigram tagger is overall the most effective tagger for this test corpus with the least amount of variance in accuracy depending on the tagset. The HMM tagger has the highest fluctuation...
in accuracy depending on the tagset, and the HMM tagger using the Penn Treebank tagset does worse than the regular expression tagger with the Universal tagset.

In further sections of this project, I will describe the process of building a HMM tagger and discuss the results of my implementation. I will be assessing my implementation with both Penn Treebank and Universal tagsets with two different corpora. One corpus is the same 20\% described above, whereas the second is the subset of the 20\% in which no sentences have out-of-vocabulary words.

3 Building a HMM Tagger

This section outlines all of the steps needed to implement a HMM tagger. First I will outline the key structures of a HMM tagger and illustrate how to compute them in the training phase. Next I will explain the Viterbi algorithm, the dynamic programming algorithm that assigns tags using the equations defined in section 3.1. Then I will work through an example sentence to demonstrate its functionality.

3.1 Key Components

As discussed in section 2.4.3, hidden Markov models define the task of part of speech tagging as follows:

\[
\text{Find tag sequence } t^n_i \text{ such that } \prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1}) \text{ is maximized.} \tag{8}
\]

There are two key probabilities in (8): the emissions probability and the transition probability. The emissions probability of token \(w_i\), \(P(w_i|t_i)\), is the chance that the Markov process token \(w_i\) given the current tag \(t_i\). In other words, of all the tokens with tag \(t_i\), what percentage of those are \(w_i\)? The transition probability, \(P(t_i|t_{i-1})\), is the probability that tag \(t_i\) follows tag \(t_{i-1}\).

To implement and expand on these probabilities, HMMs have the following components closely based on [1].

\[
\begin{align*}
Q: & \text{ set of possible states (the tagset). } |Q| = N. \\
V: & \text{ The vocabulary, or set of potential tokens. } |V| = M. \\
A: & \text{ transition probability matrix, of size } N \times N. \\
B: & \text{ emissions probability matrix, of size } N \times M. \\
\Pi: & \text{ starting probability array, of size } N.
\end{align*}
\]

Figure 10: Components of a HMM [1].

\(Q\) is the set of potential classifications. For the purposes of part of speech tagging, this is the tagset. \(V\), the vocabulary, is the set of tokens the tagger has emissions probabilities for.
The transition probabilities, as described earlier, are stored in $A$ [1]. An entry in this matrix, $A[t_{i-1}]\{t_i\}$, is equivalent to $P(t_i|t_{i-1})$ [1]. This entry is calculated by counting all of the occurrences of the sequence $t_{i-1}t_i$, and dividing that by the count of all occurrences of $t_{i-1}$:

$$A[i - 1][i] = \frac{C(t_{i-1}t_i)}{C(t_{i-1})}$$  \hspace{1cm} (9)

Each tag transition $t_it_j \forall i, j \in Q$ is accounted for. Thus $A$ can be represented as a $N\times N$ matrix, where $N = |Q|$ [1]. Note that this assumes that there is a preceding tag. For the first token in a sentence, this state to state transition is not entirely applicable. Thus, the model needs a separate structure to store the probabilities of a tag being the first state. This structure is $\Pi$, where $\Pi[i] = $ the likelihood the first tag in a sentence is $t_i$ [1].

The emissions probabilities are stored in matrix $B$ [1]. An entry in $B$, $B[t_i][v_i]$ represents the percentage of time a tag $t_i$ is word $v_i$ [1]. This entry is calculated by counting the number of times a word $v_i$ has tag $t_i$ and dividing that term by the total number of occurrences of tag $t_i$ [1].

$$B[t_i][v_i] = \frac{C(t_iv_i)}{C(t_i)}$$  \hspace{1cm} (10)

Each probability $P(v_j|t_i) \forall j \in V, i \in Q$ is accounted for. Thus $B$ is a $N$ by $M$ matrix.

### 3.2 Training a Hidden Markov Model

For the purposes of this project, 80% of pre-tagged Penn Treebank corpus will be used for training, as described in section 2.4.4. Each sentence is represented as a list of (word, tag) pairs. Within this context, the tags provided for each token is accepted as the gold standard.

#### 3.2.1 Computing $A$

$A$ is the transition probability matrix, as described above. Because an entry of $A$, as described in (9), does not count the occurrences of tokens, only the tags from the training corpus are necessary. As it follows, the first step is to process the training data such that it is flattened to one sequence of tags.

$$[[\{(w_1, t_1)\}, \{(w_2, t_2)\}, \ldots, \{(w_n, t_n)\}], \ldots, [[\{(w_1, t_1)\}, \ldots, \{(w_m, t_m)\}]]$$

Figure 11: Original training data format [7].

As seen above, $w_1$ is the first token in a sentence and $t_1$ is its corresponding tag, etc. Each list of tuples is its own sentence. The following steps are to flatten the list and strip the
words from the text. To account for the removed sentence barrier, fake sentence start and end tags are used. This is so that the first tag of one sentence does not depend on the last tag of the previous sentence.

\[ [t_{START}, t_{11}, t_{12}, \ldots, t_{1n}, t_{1,END}, \ldots, t_{m1}, \ldots, t_{mj}, t_{m,END}] \]

Figure 12: Processing necessary to calculating \( A \) [7]. In this figure, \( t_{ij} \) represents the \( j^{th} \) tag of sentence \( i \). There are \( m \) total sentences in this training corpus.

The figure above contains \( m \) training sentences, in which start and end tags buffer all of the original tags.

Now that the data has been processed to have a sequence of tags, we can then find all state to state transitions using \textit{bigrams}. The bigrams of a text is a set of pairs of a token and the token following it. For a list of elements \( E = e_1, e_2, \ldots, e_n \), the bigrams of \( E \) would be:

\[ [(e_1, e_2), (e_2, e_3), \ldots, (e_{n-1}, e_n)] \]

Figure 13: Bigrams of a list of elements

We can apply this principle to a list of tags. The bigrams would have this format:

\[ [(t_{1,START}, t_{11}), (t_{11}, t_{12}), \ldots, (t_{mj}, t_{m,END})] \]

Figure 14: Bigrams of the training corpus for part-of-speech tagging.

For this project, these bigrams are produced using NLTK’s Bigram library function [4]. Each bigram is a transition between two states in the text. Because we injected fake \textit{START} and \textit{END} tags, these bigrams include transitions to the first tag and from the last tag.

In effect, these bigrams are a list of state to state, or tag to tag, transitions. The next step is to count how many times each unique transition occurs. NLTK has a convenient tool for this as well, the frequency distribution tool, that calculates the number of times each bigram appears [4]. Sample values from the training corpus are displayed below:

\[
(NOUN, NOUN) : 6025 \\
(NOUN, VERB) : 3347 \\
(NUM, END) : 1
\]

Figure 15: Sample frequency distributions for the training corpus described in section 2.4.4 with the Universal Tagset.

Figure 15 shows some of the sample transition frequencies. For the training corpus described in section 2.4.4, there are 163 transition frequencies generated when using the Universal tagset. The total possible number of states is 196 because there are 14 tags (12 tags
in the Universal tagset and the START and END tags). It is possible that some transitions do not occur at all, and their frequencies are later treated as 0.

Now that we know the number of times a state-to-state transition occurs, populating cells in $A$ is simply a matter of counting and dividing. To find the transition probability from $t_{i-1}$ to $t_i$, or $P(t_i|t_{i-1})$ the following calculation is performed.

$$P(t_i|t_{i-1}) = \frac{C(t_{i-1}t_i)}{C(t_{i-1})}$$

The numerator for this term is either located in the list of frequency distributions or does not exist at all, and is zero. The denominator is the sum of all of the frequency distributions that start with tag $t_{i-1}$. For each tag to tag pair, this is performed until $A$ is fully populated.

### 3.2.2 Computing $\Pi$

Once $A$ has been computed, computing $\Pi$ can easily follow:

$$\forall i \in Q, \Pi[i] = A[START][i]$$

For this implementation, it is assumed that the transition probability from tag START to tag $i$ is the chance that a sentence will start with tag $i$ [1].

### 3.2.3 Computing $B$

Unlike $A$, the emissions probabilities need both the words and the tags from the training data. When processing the original training data, the words must be preserved. To facilitate our calculations, however, the training data will be converted into the following format:

$$[(t_{11}, w_{11}), (t_{12}, w_{12}), \ldots, (t_{mj}, w_{mj})]$$

Figure 16: Training data after processing to calculate $B$.

As in section 3.2.1, fake START and END tags are used on the boundaries of each sentence, but they are not shown in Figure 16.

The data requires no further processing; it is already in format where we can calculate the frequency distributions of the (tag, word) pairs similarly to how it was done to compute $A$. Sample values from the training corpus can be seen below.

$$\begin{align*}
(DET, each) : 27 \\
(NOUN, Jonas) : 1
\end{align*}$$

Figure 17: Sample frequency distributions needed to calculate $B$ using the training corpus described in section 2.4.4 with the Universal Tagset.
We then want to populate cells in $B$ with $P(w_i|t_i)$. This is defined in (10).

$$B[t_i][v_i] = \frac{C(t_i,v_i)}{C(t_i)}$$

As with computing $A$, the numerator for the above term is the frequency distribution. The denominator is calculated the same way as it is for $A$: by summing the frequency distributions that start with the tag $t_i$.

### 3.3 Finding the Most Likely Tag Sequence

For my part of speech tagger, the Viterbi algorithm is used to classify the tokens [1]. It is a dynamic programming algorithm that is often used in HMMs [1]. The Viterbi algorithm is used to calculate the probability of the most likely tag sequence, as illustrated in (7).

As input, the Viterbi algorithm takes the HMM components $A, B, Q, V,$ and $\Pi$ and the observation sequence $O$. $O$ is the sentence, or list of tokens, that the Viterbi algorithm will tag [1].

The Viterbi algorithm populates a Viterbi Matrix, which has dimensions $Nx|O|$. The Viterbi matrix is defined such that:

$$\text{Viterbi}[i][j] = \text{chance that } O_j \text{ has tag } t_i \quad (13)$$

The formula in (13) is based on the Viterbi cells for every token prior to $O_j$. The Viterbi algorithm populates this matrix, where the rightmost column contains the final probability of the most likely tag sequence. In order to retrieve the tag sequence, the Viterbi algorithm simultaneously populates a backpointers matrix of the same size.

The Viterbi algorithm is defined as follows (in [1]):

1. **Initialization step:**
   As previously described an entry $\text{Viterbi}[i][j] = \text{the likelihood that token } O_j \text{ has tag } t_i$. This is based on the tag before it, so the first column must be pre-populated. The first column of the Viterbi matrix represents the chances that the first token in a sentence has a certain tag.
   
   This is calculated as follows:
   $$\text{Viterbi}[i][1] = \Pi[i] \ast B[i][O_1] \forall i \in Q.$$
   
   The leftmost column of the back pointers matrix is set to $-1$, which will indicate that the tagging process is complete.

2. **Induction step:**
   Now that the setup is complete, the Viterbi algorithm iteratively populates the Viterbi matrix for subsequent tokens. The remaining Viterbi cells are populated as follows:
Viterbi[i][j] = max(Viterbi[i][j - 1] * A[i - 1][i] * B[i][Oj]) ∀i ∈ Q, 2 ≤ j ≤ N.

Consistent with (7), Viterbi[i][j] is the maximum probability of the following components:

(a) Viterbi[i][j − 1], probability from the previous token,
(b) A[i − 1][i], or the transition probability from the potential previous tag to the potential current tag, and
(c) B[i][Oj], or the emissions probability for token Oj given tag i.

This must be done for all possible i for each Viterbi cell, or all possible states. This is because the likelihood of a certain tag does not solely depend on the previous tag. The highest probability found in this process is what populates Viterbi[i][j]. The back pointer additionally gets populated; the argument, or previous tag that yielded the maximum probability, is stored in backpointers[i][j].

3. Termination step:

Viterbi terminates once the entire matrix is populated. The highest value in the last column represents the probability of the most likely tag sequence.

When the Viterbi algorithm is running, there is also an array of backpointers generated so the actual tag sequences can be determined [1]. The backpointer tracing starts at the last column and traces backwards to find the tags leading to the most likely sequence [1].

3.4 Example

For our example, we will be using a sample sentence similar to Figure 2, “This is a person.”.

We can display the relevant parts of A, B, and Q to walk through an example. We will be tagging this example sentence using the Universal tagset. The tags for this example are “DET”, “VERB”, “NOUN”, and “.”. The data below is a subset of the actual A, B, and Π after my HMM has been trained. Note that these numbers have been rounded for simplicity.

<table>
<thead>
<tr>
<th></th>
<th>DET</th>
<th>VERB</th>
<th>NOUN</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DET</td>
<td>.0053</td>
<td>.0406</td>
<td>.6373</td>
<td>.0174</td>
</tr>
<tr>
<td>VERB</td>
<td>.1359</td>
<td>.1654</td>
<td>.1069</td>
<td>.0344</td>
</tr>
<tr>
<td>NOUN</td>
<td>.0124</td>
<td>.1457</td>
<td>.2622</td>
<td>.2460</td>
</tr>
<tr>
<td>.</td>
<td>.0948</td>
<td>.0861</td>
<td>.1299</td>
<td>.0729</td>
</tr>
</tbody>
</table>

Table 5: Necessary subset of A for classifying the example sentence.
This is a person.

Table 6: Necessary subset of $B$ for classifying the example sentence.

<table>
<thead>
<tr>
<th>tag</th>
<th>$\Pi[tag]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DET</td>
<td>.23</td>
</tr>
<tr>
<td>VERB</td>
<td>.0092</td>
</tr>
<tr>
<td>NOUN</td>
<td>.26</td>
</tr>
<tr>
<td>.</td>
<td>.09</td>
</tr>
</tbody>
</table>

Table 7: Necessary subset of $\Pi$ for classifying the example sentence.

We now establish the empty Viterbi Matrix, which we are going to populate:

<table>
<thead>
<tr>
<th>This</th>
<th>is</th>
<th>a</th>
<th>person</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DET</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NOUN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VERB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 8: Empty Viterbi Matrix.

The first step is the initialization step. As discussed in the previous section, the first column of the Viterbi matrix is $\Pi[i] \times B[i][O_1]$.

To retrieve the necessary data, we refer to Tables 6 and 7. We determine the following information:

<table>
<thead>
<tr>
<th>tag</th>
<th>$\Pi[tag]$</th>
<th>$B[tag][This]$</th>
<th>$\Pi[tag] \times B[tag][This]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DET</td>
<td>.23</td>
<td>.0047</td>
<td>.001</td>
</tr>
<tr>
<td>VERB</td>
<td>.0092</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NOUN</td>
<td>.26</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.</td>
<td>.09</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9: Using $\Pi$ to find probabilities for the first token.

The first column of our Viterbi matrix can be extracted from the rightmost column of Table 9. Notice that the emissions probabilities eliminate all but one option for the first tag. We update the Viterbi Matrix accordingly.
Table 10: Viterbi Matrix after the setup phase.

The next step is to populate Viterbi cells for the column representing the token “is”.

To find Viterbi[.]is, we have to calculate all possible Viterbi probabilities for Viterbi[.]is. Once this is complete, the maximum value will be chosen. For this step, the emission probability remains constant: $B[.]is = 0$.

The Viterbi matrix gets updated with a result of 0, the maximum Viterbi probability, in this case.

Table 11: First calculation in the Viterbi matrix.

Note that this is relatively straightforward since the emission probability was always 0. The Viterbi matrix gets updated with a result of 0, the maximum Viterbi probability, in this case.

Table 12: Updated Viterbi Matrix after calculating Viterbi[.]is.

The next several values are additionally zero, so they will be skipped for this example. We will now calculate Viterbi[VERB][is]. For this step, the emission probability similarly remains constant: $B[VERB][is] = 0.05075$.

Table 13: Calculating Viterbi[VERB][is].
The highest value in the rightmost column of Table 13. This process continues for every Viterbi cell. The final Viterbi matrix for this example is as follows. Note that because the values in Tables 5, 6, and 7 are rounded for simplicity, the actual Viterbi matrix from my HMM for this example has slightly different numbers than seen below. Some Viterbi values in this example are also rounded for the purpose of simplicity.

<table>
<thead>
<tr>
<th></th>
<th>This</th>
<th>is</th>
<th>a</th>
<th>person</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DET</td>
<td>0.001</td>
<td>0</td>
<td>5.799</td>
<td>$10^{-8}$</td>
<td>0</td>
</tr>
<tr>
<td>NOUN</td>
<td>0</td>
<td>0</td>
<td>1.77</td>
<td>$10^{-11}$</td>
<td>0</td>
</tr>
<tr>
<td>VERB</td>
<td>0</td>
<td>2.06045</td>
<td>$10^{-6}$</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 14: The final Viterbi Matrix.

We can tell that the tag for the final token is “.” because it has the highest value in the rightmost column. While the Viterbi matrix is being populated, backpointers are also created to start from Viterbi[‘.’][‘.’] and find the tagging sequence [‘DET’, ‘VERB’, ‘DET’, ‘NOUN’, ‘.’].

Note that final Viterbi Matrix in Table 14 is relatively sparse. This sentence contains words that are very unambiguous, and that is expected. The choice of tagset also simplifies these probabilities, leading to a more straightforward classification.

### 3.5 Computing Accuracy

For the scope of this project, accuracy will be measured as follows.

$$\text{Accuracy} = \frac{\text{Number of tokens correctly tagged}}{\text{Total number of tokens}}.$$  

The gold-standard pre-tagged text is passed into the accuracy function as input in NLTK’s version [7]. For this project, I use the tags from the Penn Treebank corpus as the gold standard.

### 3.6 Out-of-Vocabulary Words

An *out-of-vocabulary* word is a token in the set of observations, $O$, that is not in $V$ [1]. Because of this, there is no known emissions probability for that token for any tag in the training corpus.

My implementation allows for a user to specify a default tag. For the purposes of this project, that default tag was “NOUN” for the Universal tagset and “NNP” for the Penn Treebank tagset. When the default tag is considered for an out-of-vocabulary word, the emissions probability is set to 1, and to 0 for all non-default tags. This is because nouns are a significant percentage of the Penn Treebank corpus, as seen in section 2.2. According
to Figures 8 and 9, ‘NOUN’ is the most frequently used tag for the universal tagset and ‘NNP’ is the third most frequently used tag for the Penn Treebank tagset.

4 Results and Analysis

My HMM tagger was compared to the data in Table 4 to determine its effectiveness. The accuracies for each tagger mentioned in section 2.4, in addition to my HMM is shown below.

<table>
<thead>
<tr>
<th>Tagger</th>
<th>Penn Treebank Tagset</th>
<th>Universal Tagset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regex</td>
<td>33.75%</td>
<td>37.75%</td>
</tr>
<tr>
<td>Unigram</td>
<td>86.08%</td>
<td>87.92%</td>
</tr>
<tr>
<td>NLTK’s HMM</td>
<td>36.47%</td>
<td>51.37%</td>
</tr>
<tr>
<td><strong>My HMM</strong></td>
<td><strong>72.86%</strong></td>
<td><strong>97.39%</strong></td>
</tr>
</tbody>
</table>

Table 15: Results from all part of speech taggers in this project on the last 20% of sentences on the Penn Treebank corpus.

The data from my HMM implementation shows that the tagset can be a factor in which classification scheme to use. The Unigram tagger has the highest accuracy when using the Penn Treebank tagset, whereas my HMM implementation achieved over 97% accuracy using the Universal tagset. Contrary to original expectations, my HMM significantly outperformed NLTK’s version. To understand why, I had to investigate the role of unknown words.

To more thoroughly understand the significance of unknown words, I additionally compared the accuracy of NLTK’s HMM tagger with my own HMM implementation with two tagsets and two corpora. The tagsets used are the Universal and Penn Treebank tagsets. The need to have two corpora used to compare NLTK and my implementations was a result of unexpected challenges from out-of-vocabulary words. Regarding out-of-vocabulary words, the ideal method of comparing the two taggers would be to have an implementation that does not tag out-of-vocabulary words and comparing the accuracy with one that does. During the implementation phase of the project, it became increasingly unclear how to implement a HMM that does not handle out-of-vocabulary words. As a result, two corpora were formed. The first corpus of comparison is the original 20% of the Penn Treebank sentences. The second corpus is a subset of the 20% of the Penn corpus in which no sentence has an out-of-vocabulary word.

The test set in the second corpus is significantly smaller than the original 20% of sentences. The original test set, 20% of sentences in the Penn corpus, contains 783 sentences and 20,039 tokens. The test set with no sentences containing out of vocabulary words has 129 sentences.

These numbers highlight the importance of handling out-of-vocabulary words, as the accuracy and magnitude of sentences a tagger can classify is significantly impacted by its
ability to tag these terms. As a result of having a default tag, I was able to tag 100% of the original test set, whereas if I did not accept out-of-vocabulary words, less than 17% of sentences would be valid input.

<table>
<thead>
<tr>
<th>Corpus</th>
<th>NLTK’s HMM</th>
<th>My HMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire 20%</td>
<td>36.47%</td>
<td>72.86%</td>
</tr>
<tr>
<td>20% - OOV sentences</td>
<td>95.13%</td>
<td>92.36%</td>
</tr>
</tbody>
</table>

Table 16: Accuracies of NLTK’s HMM tagger and my implementation with the Penn Treebank tagset.

<table>
<thead>
<tr>
<th>Corpus</th>
<th>NLTK’s HMM</th>
<th>My HMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire 20%</td>
<td>51.36%</td>
<td>93.30%</td>
</tr>
<tr>
<td>20% - OOV sentences</td>
<td>97.39%</td>
<td>97.39%</td>
</tr>
</tbody>
</table>

Table 17: Accuracies of NLTK’s HMM tagger and my implementation with the Universal tagset.

Tables 16-17 provide some interesting takeaways. First, the tagset makes a significant difference in tagger accuracy consistent with prior analyses. When tagging with the Penn Treebank tagset, which is much larger than the Universal tagset, there is a universal drop in accuracy. Where there are no out-of-vocabulary words, as seen in Table 17, the NLTK tagger performs with the exact same accuracy percentage as my implementation when using the Universal Tagset. This suggest that as the tagset gets smaller, there is less ambiguity and allows for a consistent accuracy across different implementations.

Minor differences in accuracy, as seen in the bottom row of Table 16, can be attributed to slight variations in the implementation. Further work is needed to determine what those variations are and how they impact the overall performance of a tagger.

When sentences containing out-of-vocabulary words are considered, NLTK’s accuracy varies greatly from mine. It is suspected that this is due to how NLTK handles, or fails to handle, out-of-vocabulary words. This data prompts a thorough investigation of how out-of-vocabulary words are handled in NLTK and how that impacts the accuracy.

### 5 Future Work

Tagging out-of-vocabulary words accurately in a HMM implementation is a much more involved task than originally anticipated. The only way this project handles out-of-vocabulary words is by assigning a default tag. Even though section 2.2 demonstrates that nouns make a significant percentage of the Penn Treebank corpus, it is worth further exploring other methods of classifying out-of-vocabulary words. Future work also includes a
more thorough comparison of my HMM to NLTK’s HMM tagger with regards to out-of-
vocabulary words. For the test sentences that contain out-of-vocabulary words, the default
tagger in my implementation outperforms NLTK’s implementation. Making the compar-
ison more thorough requires a thorough investigation of NLTK’s implementation. In ad-
dition, my implementation should have a baseline where out-of-vocabulary words are not
handled to be compared to NLTK’s.

6 Summary

Part of speech tagging is a part of the natural language processing pipeline that classi-
fies the part of speech of each token in a sentence. Several methods for part of speech tag-
ging exist, and probability-based taggers are the most effective. A Hidden Markov Model
is a probabilistic classification scheme that can be applied to part of speech tagging. This
project implements a Hidden Markov Model that performs at a higher accuracy rate than
the Natural Language Toolkit library implementation on the selected test corpus. Future
explorations include a thorough analysis of out-of-vocabulary words and different methods
of tagging them within a Hidden Markov Model implementation.
References


