Hidden Markov Models for Part of Speech Tagging

Master's Completion Research: Emre Yurtbay

April 2022

Introduction

Consider the following sentence:

The woman ate the orange.

Any sufficiently bright fifth grader could tell you the part of speech of each word in the above sentence: "The" is a determiner, "woman" is a noun, "ate" is a verb, "the" is a determiner, and "orange" is a noun. The "tag sequence" for this sentence is the following:

D N V D N

At first, it would seem simple enough to train a model to build a tag sequence for any given sentence. However, consider another sentence:

I saw the orange cat.

In the first sentence, the word "orange" functions as a noun, but in the second sentence, "orange" is an adjective. It is easy to see that the part of speech of many words is not constant; it depends on the context in which it appears in a sentence. In part of speech (POS) tagging, our goal is to build a model where the input is a sequence of words (that is, a sentence), such as the ones above, and the output is a sequence of tags representing the parts of speech of each of the words of the input sequence. We can formalize the POS tagging problem the following way. Let the inputs be a sequence of words $x_1, x_2, \ldots, x_n$, and a "tag set" Q, which contains all of the possible parts of speech for the language. Our output is the sequence of part of speech tags $y_1, y_2, \ldots, y_n$, where y_i corresponds with x_i, and each $y_i \in Q$. What makes this particular problem challenging to solve is something we have already alluded to: the problem of ambiguity. Many words in the English language, and many other languages, can take different parts of speech. Part of speech tagging can therefore be thought of as a disambiguation task; a model that accurately predicts the parts of speech for each word in a sentence must effectively take into account each word's context in a sentence. In this paper, we will use Hidden Markov Models (HMMs) to solve the POS tagging problem, and demonstrate how they are able to achieve extremely high accuracies.

I first learned about this problem while learning about stochastic processes. Hidden Markov models are an extension of the more basic Markov chain models that are some of the core structures found in stochastic processes. Part of Speech Tagging is a task that is natural for Hidden Markov models because we have a set of latent states of interest as well as observable states. The prediction task of the part of speech tagging problem is akin to the multi-class classification tasks we have become quite familiar with in CS 671. This project is also an extension on some of the latent variable models we learned in this class in the context of the EM algorithm. Finally, I also got the opportunity to learn about a new algorithm, the Viterbi algorithm, which we did not use in class but comes up all the time when statistical methods are applied to sequence data.

Related Work

I will be implementing as HMM class from scratch myself in Python 3. An implementation for HMMs is available in the natural language processing python package NLTK, but the HMMs I present here show no significant difference in performance compared to the those found in NLTK.

The Model

Hidden Markov Models (HMMs) belong to a class of models called probabilistic sequence models: given a sequence of words, they compute a probability distribution over possible sequences of labels and choose the best label. HMMs require that there exists an observable "process" X whose outcomes are influenced by the outcomes of some unobservable "process" Y. In our example, the observable process may be the sequence of words that make up the sentence, while the unobserved process Y is the sequence of parts of speech. We do not see the parts of speech directly; they are the latent or "hidden" states that we are ultimately trying to recover. The HMMs we will use here consist of several components, each of which will be discussed below.

HMM Components

The first component is $Q = \{q_1, q_2, \ldots, q_n\}$, a set of n states. These states correspond with the different parts of speech, (the hidden states) that each word in a sentence can be tagged as. In grade school in America, many of us are taught that there are 9 basic parts of speech in the English language: noun, verb, article, adjective, preposition, pronoun, adverb, conjunction, and interjection. However, English is a much richer language than this, and we can define more granular parts of speech as well, such a singular proper nouns, predeterminers, gerund verbs, modals, and many, many more. *In toto*, we will be using a set Q with 36 possible "states" when building our model.

Tag	Description	Example	Tag	Description	Example	Tag	Description	Example
CC	coord. conj.	*and, but, or*	NNP	proper noun, sing.	*IBM*	TO	"to"	*to*
CD	cardinal number	*one, two*	NNPS	proper noun, plu.	*Carolinas*	UH	interjection	*ah, oops*
DT	determiner	*a, the*	NNS	noun, plural	*llamas*	VB	verb base	*eat*
EX	existential 'there'	*there*	PDT	predeterminer	*all, both*	VBD	verb past tense	*ate*
FW	foreign word	*mea culpa*	POS	possessive ending	*'s*	VBG	verb gerund	*eating*
IN	preposition/ subordin-conj	*of, in, by*	PRP	personal pronoun	*I, you, he*	VBN	verb past participle	*eaten*
JJ	adjective	*yellow*	PRP$	possess. pronoun	*your, one's*	VBP	verb non-3sg-pr	*eat*
JJR	comparative adj	*bigger*	RB	adverb	*quickly*	VBZ	verb 3sg pres	*eats*
JJS	superlative adj	*wildest*	RBR	comparative adv	*faster*	WDT	wh-determ.	*which, that*
LS	list item marker	*1, 2, One*	RBS	superlatv. adv	*fastest*	WP	wh-pronoun	*what, who*
MD	modal	*can, should*	RP	particle	*up, off*	WP$	wh-possess.	*whose*
NN	sing or mass noun	*llama*	SYM	symbol	+,%, &	WRB	wh-adverb	*how, where*

Figure 1: A table of English Parts of Speech

Next, we have an $n \times n$ transition probability matrix A, which contains the probabilities from moving from one state (that is, part of speech) to another. For example, if adverbs ("RB") are followed by past tense verbs ("VBD") 60 percent of the time, $A_{RB,VBD} = 0.6$. These transitions are computed directly from the labeled training data.

The emission probability matrix E gives us the probability an observation w is generated from state q. In our case, the observations are words and the states are parts of speech. If 23 percent of all prepositions

("IN") in English are the word "of", $E_{IN,of} = 0.23$. The probabilities contained in this matrix may be slightly unintuitive, as we are calculating $P(w_i \mid t_i)$. This answers the question "if we were going to generate a part of speech t_i such as a modal, how likely is it that the modal will be word w_i?" The MLE of the emission probability is given by

$$P(w_i \mid t_i) = \frac{C(t_i, w_i)}{C(t_i)}$$

The E matrix in our example has 36 rows. The number of columns is equal to the number of unique words in the training corpus, which can be absolutely massive; naturally, this matrix is very sparse. It is important to note that the rows of E must sum to 1.

Finally, we have an initial probability distribution π over all states. This gives us the probability of each part of speech beginning a sentence. π is a vector of 36 entries, and the sum of all entries equals 1. Some π_j can equal 0: in English, sentences starting with past tense verbs ("VBD") are considered ungrammatical, so theoretically, $\pi_{VBD} = 0$.

The "Order" of an HMM

One other important consideration is the "order" of the HMM. Recall that a model that accurately predicts the parts of speech for each word in a sentence must effectively take into account each word's context in a sentence. To predict the part of speech of a word, the model will look at a number of words before it to help determine parts of speech. If the order of the HMM is 2 and the model is trying to tag word i, it will use the tag of word $i-1$ to help tag word i. (w_i, w_{i-1}) form what is known as a bigram, and thus order 2 HMMs are also known as "bigram HMMs". This is also where the "Markov property" of HMMs comes into play: in bigram HMMs, the tag for w_i only depends on the tag for w_{i-1}, and no other words before or after it. If we want to consider even more context, we can use trigram HMMs, in which tags for w_i depend not only on w_{i-1} but also w_{i-2}. Increasing the order of the HMM may lead to more accurate tags, but also to a much greater computational burden while training. HMMs can achieve remarkably accurate results even when their order is just 2 or 3.

Decoding with the Viterbi Algorithm

For a model with latent states, the task of determining the hidden variable sequence corresponding to the sequence of observations is called "decoding." To formalize this somewhat, given as an input an HMM and a sequence of observations $o_1, o_2, \ldots o_T$ (i.e. a sentence), we want to find the most probable sequence of states $q_1, q_2, \ldots q_T$. In the part of speech tagging example, the goal is to choose the tag sequence $t_1 \ldots t_n$ that is most probable given the observation sequence of n words, $w_1 \ldots w_n$. That is,

$$\{t^*\}_1^n = \arg \max_{t_1 \ldots t_n} P(t_1 \ldots t_n \mid w_1 \ldots w_n)$$

We can compute this probability using Bayes' rule

$$\{t^*\}_1^n = \arg \max_{t_1 \ldots t_n} \frac{P(w_1 \ldots w_n \mid t_1 \ldots t_n) P(t_1 \ldots t_n)}{P(w_1 \ldots w_n)}$$

Using a standard Bayesian trick, we can actually disregard the denominator when doing this computation.

$$\{t^*\}_1^n = \arg \max_{t_1 \ldots t_n} P(w_1 \ldots w_n \mid t_1 \ldots t_n) P(t_1 \ldots t_n)$$

Still this calculation is pretty difficult. We need to make a few simplifying assumptions. The first is the independence assumption:

$$P(w_1 \ldots w_n \mid t_1 \ldots t_n) = \prod_{i=1}^{n} P(w_i \mid t_i)$$

Algorithm 1: Trigram Viterbi.

Input: A second-order HMM M with states $Q = \{1, 2, \ldots, K\}$ given by its transition matrix A, emission probability matrix E (alphabet Σ), and probability distribution π on the (initial) states; a sequence X of length L (indexed from 0 to $L-1$).

Output: A sequence Z, with $|Z| = |X|$, that maximizes $p(Z, X)$.

$v[\ell', \ell, 1] \leftarrow (\pi_{\ell',\ell} \cdot E_{\ell'}(X_0) \cdot E_\ell(X_1))$ for every $\ell', \ell \in Q$;
for $i \leftarrow 2$ **to** $L-1$ **do**
  foreach $\ell' \in Q$ **do**
    foreach $\ell \in Q$ **do**
      $v[\ell', \ell, i] \leftarrow E_\ell(X_i) \cdot \max_{\ell'' \in Q}(v[\ell'', \ell', i-1] \cdot A_{\ell'',\ell',\ell})$;
      $bp[\ell', \ell, i] \leftarrow \operatorname{argmax}_{\ell'' \in Q}(v[\ell'', \ell', i-1] \cdot A_{\ell'',\ell',\ell})$;
$Z_{L-2} \leftarrow \operatorname{argmax}_{\ell' \in Q}(v[\ell', \ell, L-1])$;
$Z_{L-1} \leftarrow \operatorname{argmax}_{\ell \in Q}(v[\ell', \ell, L-1])$;
for $i \leftarrow L-3$ **downto** 0 **do**
  $Z_i \leftarrow bp[Z_{i+1}, Z_{i+2}, i+2]$;
return Z

Finally, we need to use the order of the HMM to make a further simplifying assumption. If we are working with an order 2 HMM, we must make the bigram assumption, which states that the probability of a tag is only dependent on the previous tag, rather than the entire tag sequence. This can be expressed as

$$P(q_1 \ldots q_n) = \prod_{i=1}^{n} P(q_i \mid q_{i-1})$$

If we are working with an order 3 HMM, we have the trigram assumption, that is,

$$P(q_1 \ldots q_n) = \prod_{i=1}^{n} P(q_i \mid q_{i-1}, q_{i-2})$$

We can improve the performance of our model even more by using a classic natural language processing procedure called smoothing. Say we are interested in trigram HMMs; then we are interested in $P(t_i \mid t_{i-1}, t_{i-2})$. Instead of using the classic rules of conditional probability to solve this, we can instead use the following formulation

$$P(t_i \mid t_{i-1}, t_{i-2}) = \lambda_1 * P(t_i \mid t_{i-1}, t_{i-2}) + \lambda_2 * P(t_i \mid t_{i-1}) + \lambda_3 * P(t_i)$$

subject to

$$\lambda_1 + \lambda_2 + \lambda_3 = 1, \lambda_i \geq 0$$

The λ values are fit by an algorithm called linear interpolation, which is implemented in the below code. When we use smoothing, we can combat the overfitting that may occur when we use high order HMMs.

Without loss of generality, let's consider order 2 HMMs without smoothing. Using our new assumptions, we have

$$\{t^*\}_1^n = \arg \max_{t_1 \ldots t_n} P(w_1 \ldots w_n \mid t_1 \ldots t_n) P(t_1 \ldots t_n) = \arg \max_{t_1 \ldots t_n} \prod_{i=1}^{n} P(w_i \mid t_i) P(t_i \mid t_{i-1})$$

This formulation is serendipitously convenient; $P(w_i \mid t_i)$ corresponds to the emission probabilities, and $P(t_i \mid t_{i-1})$ corresponds to the transition probabilities. So now, how can we do this decoding? For

HMMs, we use the Viterbi algorithm, a dynamic programming algorithm for obtaining the maximum a posteriori (MAP) probability estimate of the most likely sequence of hidden states that results in a sequence of observed events. The Viterbi procedure for trigram HMMs is given in Algorithm 1, though note that the bigram Viterbi is very similar. The Viterbi algorithm begins by setting up a dynamic programming lattice, with one column for each observation (that is, word) and one row for each possible state. Each cell of the lattice, which is called $v_t(j)$ represents the probability that the HMM is in state j after seeing the first t observations and passing through the most probable state sequence $q_1 \mid q_{t-1}$. Tracing backwards through the lattice, we can recover the most probable sequence of parts of speech for each word in the given sentence.

What is the run time of the trigram Viterbi algorithm? Line 1 has a run time of K^2, since we have nested for loops, each of which iterates K times. The for loop on line 2 runs on the order of m time since the sequence is of length m. The for loop on line 3 runs on the order of K time, since we iterate through every state. The for loop on line 4 runs on the order of K time as well, since we iterate through every state. Finally, finding the max value for ℓ'' in lines 5 and 6 take on the order of K time since we must iterate through every state again. The block from line 2 to 6 has run time of $m * K^3$. Finally, the for loop on line 9 runs on the order of m time. Hence, the run time of the algorithm is $O(mK^3)$

Hyperparameter Selection

The main model parameter we could theoretically tune is the order of the HMM. Increasing the order of the model would mean the model takes into account more context for each word, but this could lead to overfitting. In practice, increasing the order of the model also increases the computation burden significantly, to the point where we would have to sit around and wait a very long time for the models to fit. For the purposes of this project, we will only work with order 2 and 3 HMMs and not tune this parameter.

Experimental Setup and Metrics

The data we will be using in this demonstration is a text file of sentences, containing over 1 million words in total. Each line has exactly 1 word and exactly 1 tag. We call this a large collection of text a "corpus." Each word is tagged with 1 of 36 parts of speech by a human tagger (presumably a grammar expert). These tags are taken to be the ground truth - though one should note there could be errors made by the human taggers. Our test set is a small set of natural text, untagged. We also have a version of the test set that is tagged by a human tagger that is in the same form as the training corpus. This will be used as our validation set to determine test accuracy. The metric of importance here will be test accuracy, that is, what percent of words in the test data are labelled correctly by the HMM.

The corpus files are read in by two python functions I wrote specifically to handle this type of data. The rest of the functions build the HMM, and the Analysis/Experiment Code section of the notebook runs the experiments.

We will have 4 experiment types. We will test the performance of trigram HHMs with smoothing, trigram HMMs without smoothing, bigram HMMs with smoothing, and bigram HMMs without smoothing. This setup will allow us to compare the performance of trigram HMMs and bigram HMMs as well as the effect of smoothing.

Here, we will explain the experiment for trigram HMMs with smoothing; all of the other experiments will proceed similarly. First, we load in 1 percent of the training data, and train a trigram HMM on the data using smoothing. Then we will use this trained HMM to predict on the test data, and get a prediction accuracy score. Next, we will load in 5 percent of the entire training data, and train a trigram HMM on the data using smoothing. Again we will use this trained HMM to predict on the out of sample data, and

get a prediction accuracy score. We repeat this for 10, 25, 50, 75, and 100 percent of the data. This will give us a sense of how increasing the amount of training data increases the accuracy of our model.

Experimental Results

Table 1 and Table 2 show the results of our experiments.

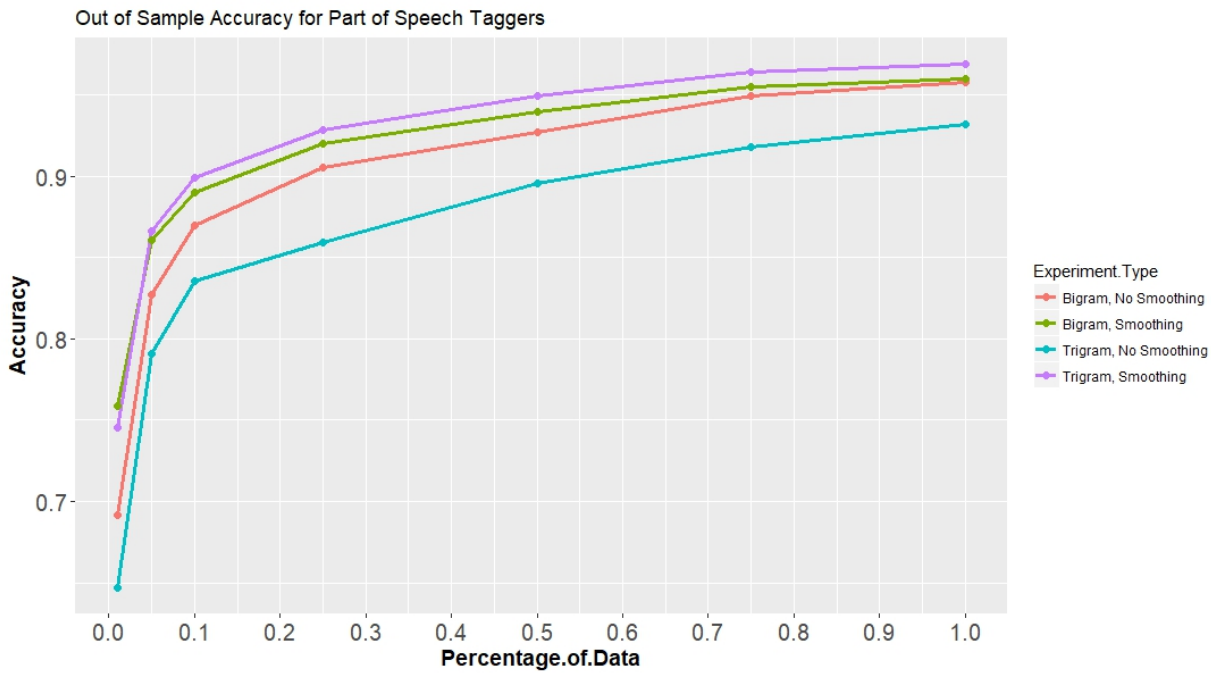
Table 1: Bigram HMM Results

Percent of Data	No Smoothing Accuracy	Smoothing Accuracy
1%	0.691	0.758
5%	0.827	0.860
10%	0.869	0.890
25%	0.905	0.919
50%	0.927	0.939
75%	0.949	0.954
100%	0.957	0.959

Table 2: Trigram HMM Results

Percent of Data	No Smoothing Accuracy	Smoothing Accuracy
1%	0.646	0.745
5%	0.790	0.866
10%	0.835	0.899
25%	0.859	0.928
50%	0.895	0.949
75%	0.917	0.964
100%	0.932	0.969

From Table 1, we see that the smoothed versions of the bigram HMM performs significantly better than its unsmoothed counterparts when the training data is small, but this advantage decreases as we use more training data. This points to the fact that bigram HMMs do not overfit to a high degree, so they benefit less from smoothing. When we use all of the training data, the bigram HMM achieves 96 percent accuracy, which is very impressive. Table 2 gives us the results for trigram HMMs. The effect of smoothing is much more apparent for trigram HMMs; smoothed trigram HMMs outperform their unsmoothed counterparts by a significant margin. Interestingly, the unsmoothed version of the bigram HMM actually achieves higher accuracies than the unsmoothed version of the trigram HMM. Still, the smoothed trigram HMM seems to have the best performance, achieving 97 percent accuracy when trained on the full data. This accuracy is very impressive; a trigram HMM could probably do better than the vast majority of literate English speakers. Finally, we see that as we increase the percentage of data on which we train each HMM, the accuracy improves, but the rate of increase in accuracy tapers off as the percent of the training data used increases. We can see these results in graphical form on the next page.



Errors and Mistakes

The most difficult part of this project was coding the Viterbi algorithm. I do not have much experience with dynamic programming, so learning this new technique took some time, but it was very rewarding. Implementing the linear interpolation to get the smoothing to work was also a challenge, but it helped me achieve better accuracies, so it was indeed worth it in the end.

References

[1] Michael Collins: Langauge Models.
http://www.cs.columbia.edu/mcollins/hmms-spring2013.pdf

[2] Michael Collins: Tagging Problems and Hidden Markov Models.
http://www.cs.columbia.edu/mcollins/hmms-spring2013.pdf

[3] Daniel Jurafsky & James H. Martin: Sequence Labeling for Parts of Speech and Named Entities
https://web.stanford.edu/jurafsky/slp3/8.pdf

[4] NLTK Project: Natural Language Toolkit Python Library
https://www.nltk.org