CIS530: Assignment 2

Michael Gao David Zhan

1 Introduction

In this project, we implement a Part-of-Speech (POS) tagger using a trigram Hidden Markov Model (HMM). POS tagging entails predicting tags which assign grammatical categories to each word in a text input, such as noun, verb, or adjective. The data used in this task is from the Penn Treebank, which consists of tagged words from the Wall Street Journal.

Maximum Likelihood Estimation (MLE) was applied for parameter estimation, incorporating Laplace (additive) and linear interpolation smoothing techniques. The performance of various inference methods—greedy search, beam search, and the Viterbi algorithm—was then compared. Different combinations of smoothing techniques, inference methods, and n-gram models were tested on a development dataset to optimize prediction accuracy.

2 Data

Our data consists of a train set, development set, and test set. The latter is given only as text input, while the other two are given with both inputs and tag labels. In total, there are 46 unique labels, including a `-DOCSTART-`, the equivalent of a `<START>` tag. No `<STOP>` tag is included in the data. Each set consists of a number of documents, which are each preceded with a `<START>`. We provide some statistics on the nature of these data sets:

Table 1: Data Summary

	Train	**Dev**	**Test**
Number of documents	1,387	462	463
Total tag count	696,475	243,021	236,582
Mean document length (tags)	502.1	526.0	511.0
Vocab size (unique words)	37506	20706	20180

We do not alter/preprocess the data besides ignoring `<STOP>`, since it does not exist in any outputs and thus offers no benefit for predictions.

3 Handling Unknown Words

We use a suffix tree to handle unknown words. When training the model, we add all words to a suffix tree. Words with common suffixes often share the same POS tag with a high probability; for example, "-ing" is a common suffix of present-tense verbs (e.g. jumping, trying, running). To handle unknown words, we use the suffix tree to match the maximum suffix, then count the numbers of each tag present in words which share that suffix. From those numbers, we calculate the emission probabilities and apply smoothing. This approach also allows us to calculate probabilities based on the present training set, rather than an external embedding.

4 Smoothing

We implement two forms of smoothing in our model: Laplace and linear interpolation. Laplace smoothing involves adding a small count to all frequency probability calculations to prevent 0 possibilities. In our unigram, bigram, and trigram methods, we add a Laplace factor of $\alpha = 0.01$ to all counts, then normalize appropriately. By doing so, we artificially increase the probabilities of unseen tag sequences, which slightly dilutes the probabilties of frequently seen sequences. However, with a $\alpha = 0.01$, the impact on the most frequent n-grams is minimal while allowing the model to now make inferences on unseen words.

Linear interpolation smoothing assigns weights to each n-gram and takes a linear combination of each lower gram probability. Higher order n-grams may overfit to specific tag sequences, and since unigrams and bigrams are more common, they provide more reliable fallbacks for unseen words. By weighting each n-gram, we dilute the probabilites of precise tagging sequences, but the model is able to handle unseen higher order sequences more reliably. We use bigram lambdas

$\lambda_1 = 0.1, \lambda_2 = 0.9$ and trigram lambdas $\lambda_1 = 0.1, \lambda_2 = 0.1, \lambda_3 = 0.8$.

5 Implementation Details

We implement each of the n-gram transition calculations by first computing counts of each tag sequence (e.g. `tag1,tag2,tag3`), then dividing by the total to normalize (and applying smoothing after, if specified). Each of these computations are vectorized using Numpy arrays for speed, although training takes a fraction of a second regardless. Emission calculations are computed similarly, by summing counts of `tag,word` pairs, then dividing and applying smoothing if specified.

A suffix tree is included in the model to handle unknown words using suffixes to estimate probable tags based on training words. Suffixes are filtered to am inimum length of 2 and maximum frequency of 5, such that short or infrequent suffixes to not unnecessarily inflate the tree's size. This improves each of memory, efficiency, and prediction accuracy.

In each inference method, we maintain log-space probabilities to avoid floating-point precision errors and underflow. The greedy and beam search inference implementations iterate over the desired input sequence, check the current beam, and compare the probability of adding each possible tag to the sequence. The top k tag sequences are then kept for the successive iteration. The viterbi algorithm is implemented using a 2-dimensional state lattice of `tag,word` for bigrams and `(tag1,tag2),word` for trigrams. We then use dynamic programming to check all transitions from the previous tag(s) and emissions to find the sequence with the maximum likelihood. We also attempted to vectorize computations wherever possible, but for our implementation there was not much opportunity for it.

6 Experiments and Results

Test Results Our final leaderboard submission had a 96.19 F1-score on the test data, using a trigram Viterbi model with linear interpolation with $\lambda_1 = 0.1, \lambda_2 = 0.1, \lambda_3 = 0.8$, trained on both the train and dev data. This was the best performance out of the submissions to the leaderboard (Table 2). For leaderboard submissions, we mainly used Viterbi since it makes optimal inferences, and trigrams because it performs better than bigrams. The interpolation submissions performed slightly better than Laplace, ceteris paribus.

Table 2: Test data F1-score.
** denotes final leaderboard submission.

Setup	**F1**
3-gram Viterbi, interpol., $\lambda = (0.1, 0.1, 0.8)$, train & dev data	96.19 **
3-gram Viterbi, interpol., $\lambda = (0.1, 0.1, 0.8)$, train data only	96.02
3-gram Viterbi, Laplace, $\alpha = 0.001$, train data only	95.89
3-gram Viterbi, Laplace, $\alpha = 0.01$, train data only	95.87
3-gram Beam ($k = 20$), Laplace, $\alpha = 0.001$, train data only	95.11

Smoothing The values for α and λ were derived empirically by testing values, as show in tables 3 and 4 below. For succinctness, we show a subset of the tests done to make the comparison apparent.

Table 3: Interpolation performance across $\lambda = (\lambda_1, \lambda_2, \lambda_3)$ using 3-gram beam search (k=10)

Setup	**T**	**U**
$\lambda = (0.33, 0.33, 0.34)$	95.43	71.10
$\lambda = (0.1, 0.1, 0.8)$	95.76	74.07
$\lambda = (0.1, 0.8, 0.1)$	95.42	72.04
$\lambda = (0.8, 0.1, 0.1)$	94.27	65.31

Table 4: Interpolation performance across α using 3-gram beam search (k=10)

Setup	**T**	**U**
$\alpha = 1$	93.43	73.71
$\alpha = 0.1$	95.38	74.62
$\alpha = 0.01$	95.66	74.49
$\alpha = 0.001$	95.63	74.51
$\alpha = 0.0001$	95.58	74.45

From table 3, we see that a higher λ_3 results in the highest total (T) and unknown word (U) accuracy of 95.76% and 74.07%, respectively. A higher λ_1 results in the lowest accuracies of 94.27% and 65.31%. Splitting λ values evenly or with a higher λ_2 sit in the middle.

From table 4, we see that $\alpha = 0.01$ has the highest total accuracy of 95.66%. $\alpha = 0.001$ has the highest unknown word accuracy of 74.51%, but $\alpha = 0.01$ is only 0.02% worse.

While running tests for the rest of this document, we use $\alpha = \alpha_{opt} = 0.01$ for Laplace and

$\lambda = \lambda_{opt} = (\lambda_{1,opt}, \lambda_{2,opt}, \lambda_{3,opt}) = (0.1, 0.1, 0.8)$ for interpolation smoothing.

As seen in table 5, without smoothing, all inference methods perform worse in total accuracy, while the difference in unknown word accuracy is inconsistent. However, as a more optimal method is used, the disparity is much less: for total accuracy, Greedy without smoothing is 0.37% worse than Laplace and 0.59% worse than interpolation, but Viterbi is only 0.16% worse than Laplace and 0.30% worse than interpolation.

Table 5: Performance with vs. without Smoothing, for total (T) and unknown-only (U) accuracy

Setup	**T**	**U**
3-gram Greedy, no smoothing	94.13	69.26
3-gram Greedy, Laplace α_{opt}	94.50	69.22
3-gram Greedy, interpol. λ_{opt}	94.72	68.77
3-gram Beam (k=10), no smoothing	95.43	74.43
3-gram Beam (k=10), Laplace α_{opt}	95.66	74.49
3-gram Beam (k=10), interpol. λ_{opt}	95.76	74.07
3-gram Viterbi, no smoothing	95.53	75.31
3-gram Viterbi, Laplace α_{opt}	95.69	75.48
3-gram Viterbi, interpol. λ_{opt}	95.83	74.69

In general, smoothing increases total accuracy, and interpolation does so more than Laplace. However, interpolation is consistently worse in unknown word accuracy than both no smoothing and Laplace smoothing.

Bi-gram vs. Tri-gram The results in Table 6 indicate that the trigram model slightly outperforms the bigram model in both overall token accuracy and accuracy on unknown words. This suggests that the trigram model captures more context, allowing it to better handle cases where word disambiguation relies on considering two preceding tags rather than just one.

However, the improvement is small (95.42 vs. 95.69 for Laplace, and 95.44 vs. 95.83 for interpolation), which implies that the trigram model captures more information, but the additional context does result significantly better performance. The bigram model still performs quite well, since a 95.43 percent accuracy is proficient, making it a competitive option when considering the tradeoff between model complexity and performance. However, for unknown words, the trigram model performed up to about 2% better, which can be significant depending on the desired task.

Table 6: Bigram vs. trigram performance

Setup	**T**	**U**
2-gram Viterbi, Laplace α_{opt}	95.42	73.54
3-gram Viterbi, Laplace α_{opt}	95.69	75.48
2-gram Viterbi, interpol. λ_{opt}	95.44	73.12
3-gram Viterbi, interpol. λ_{opt}	95.83	74.69

This indicates that while higher-order n-gram models (such as the trigram model) can improve performance slightly by leveraging more contextual information, the gains may be limited, particularly when compared to the additional computational complexity required.

Greedy vs. Viterbi vs. Beam Using Laplace smoothing with $\alpha_{opt} = 0.01$, Viterbi out-performs beam search (for all $k \in [1..20]$) for both total accuracy and unknown word accuracy. It is true that increasing the value of k for beam search does increase its accuracy in both categories, but the amount of increase significantly decreases as k increases. Initially, increasing k improves accuracy substantially. For example, from $k = 1$ (Greedy) to $k = 5$, the total accuracy jumped over 1% and unknown word accuracy increased over 5%. However, the tradeoffs in time exceed the additional benefit of less than 0.01% increase in total accuracy for each unit increase of k after $k = 20$. Our viterbi implementation takes almost 9 minutes, and beam search with $k = 20$ takes about a third of that. In table 7, we compare different inference methods including a subset of k values for beam search. Additionally, Greedy (with Laplace smoothing) will find the optimal tag sequence for an individual sentence 72.42% of the time.

Table 7: Greedy, Beam, Viterbi Comparison

Setup	**T**	**U**	**Time**
Greedy, Laplace α_{opt}	94.50	69.22	9sec
Beam (k=5), Laplace α_{opt}	95.63	74.24	33sec
Beam (k=10), Laplace α_{opt}	95.66	74.49	65sec
Beam (k=20), Laplace α_{opt}	95.69	74.70	3min
Viterbi, Laplace α_{opt}	95.69	75.48	9min

7 Analysis

Error Analysis The most common error classes for the Trigram Viterbi model are:

Adjective-Noun Confusion (JJ vs. NN): The model struggles to differentiate between adjectives

and nouns in certain contexts, especially when descriptive words can be mistaken for nouns. This type of error accounts for the most and third most frequent error for our model. For example, in the case of "closing," the model predicted a noun (NN), while the true tag was an adjective (JJ).

Proper Noun vs. Common Noun (NNP vs. NN): This error occurs when the model confuses capitalized proper nouns with regular common nouns. A typical example is the word "End," where the model predicted a common noun (NN), but the correct tag was a proper noun (NNP).

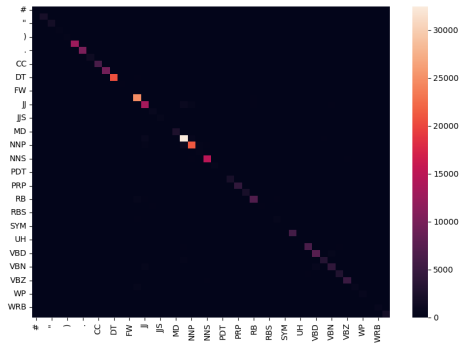
Verb Tense Confusion (VBN vs. VBD): The model sometimes confuses past participles with simple past tense verbs, likely due to similar sentence structures where both forms are plausible. For instance, "estimated" was tagged as a past participle (VBN) when it should have been marked as simple past tense (VBD).

Table 6 lists the 10 most common POS tagging errors for our model.

Table 8: 10 Most Common POS Tagging Errors

True Tag	Predicted Tag	Count
JJ	NN	613
NNP	NN	600
NN	JJ	595
NN	NNP	458
VBN	JJ	425
VBD	VBN	412
IN	RB	355
NNP	NNPS	320
VBN	VBD	311
RB	RP	243

Confusion Matrix The heatmap visualizes the performance of a POS tagger through a confusion matrix, where rows represent true tags and columns represent predicted tags. The diagonal elements show correct predictions, with brighter spots indicating higher number of predictions. The brightness is almost all concentrated along the diagonal, suggesting that the model is very accurate.



Off-diagonal spots highlight misclassifications, which there are very few. To the naked eye, there do not seem to be any misclassifications of significance, as aside from the center diagonal (or correct classifications), the rest of the squares are black.

8 Conclusion

In this project, we implemented a Part-of-Speech tagger using a Hidden Markov Model with trigram transitions and applied different smoothing and inference methods. Our results demonstrated that the Viterbi algorithm, especially when combined with linear interpolation smoothing, provided the best overall accuracy, reaching 95.83% for token accuracy and 74.69% for unknown words. This outcome highlights the advantage of using higher-order n-gram models for capturing more context, which is particularly beneficial for disambiguating tags in complex sentences.

The analysis also revealed that smoothing is crucial for model performance, as it significantly improves the handling of unseen word-tag pairs. Laplace smoothing, in particular, enabled the model to generalize better, as seen by the increase in unknown word accuracy compared to models without smoothing.

Error analysis identified common challenges such as distinguishing between adjectives and nouns (e.g."closing"), differentiating proper nouns from common nouns (e.g., "End"), and handling verb tense variations. Despite these challenges, the overall performance and efficiency of our trigram Viterbi model demonstrate its robustness in accurately tagging parts of speech, making it suitable for practical NLP tasks involving POS tagging. Future work could explore more advanced smoothing techniques or the incorporation of external embeddings to further improve accuracy, especially for handling rare or unknown words.