CIS530: Assignment 3

Kyle Sullivan Federico Cimini

1 Introduction

The aim of this assignment is to construct a part-of-speech (POS) tagger that predicts the part of speech of each word in a sentence. To achieve this, we are using data from the Wall Street Journal.

We built a baseline tagger and experimented with the following attributes to find the best-performing tagger:

- Unigrams, bigrams, trigrams.
- Greedy, Beam search, Viterbi.
- Handling of unknown words through suffix groups and $< unk >$ token.
- Smoothing through add-k and linear interpolation.

In the end, we found out that the best-performing tagger uses trigrams, Viterbi, suffix groups, unk tokens and add-k smoothing, and has an F1 score of 95.80.

2 Data

The data used for this assessment corresponds to various documents extracted from articles in the Wall Street Journal. Each of these documents has several sentences in it, including punctuation and numbers. A summary of the statistics of the data can be found in Table 1.

Parameter	**Training**	**Dev**	**Test**
Number of documents	1,387	462	463
Number of tokens	697,862	243,483	237,045
Avg Document Length	503.14	527.02	511.98
Vocabulary Size (number of unique tokens)	37,506	20,706	20,180
Number of unique tags	60	60	-

Table 1: Data statistics from the World Street Journal documents

We interpret vocabulary size to be the number of distinct tokens in data. Dev and test data have a lower vocabulary size but can contain words that are not present in the training data.

For pre-processing, we decided not to change the casing of words, since we believe that upper or lower-case information is useful to determine certain parts of speech. For example, if we find a word that starts with a capital letter and is not located after a period, it most likely is a proper noun (NNP). We also decided not to deviate from normal tokenization (each token corresponds to a word).

Something we did that improved the quality of our model was to divide each document into sentences. After every period (or question mark or exclamation mark) we added STOP tokens to pad different sentences and avoid sentences influencing each other (especially when running trigrams).

3 Handling Unknown Words

To handle unknown words, we implemented two methods: *unk* tokens and suffix grouping.

For the *unk* token implementation, we identified tokens present less than k times in the data. This threshold k was established through hyperparameter tuning, and the best value for us was $k = 4$. But instead of only replacing the words with the *unk* token, we doubled the size of our training data, keeping the rare words with frequency lower than k, but also having sentences with those new *unk* tokens. We found that this strategy helped us reduce the loss of information in this method while improving our performance with new unknown words in the new data.

The second strategy was utilizing rules that classified unknown words checking for suffixes and other common characteristics of words:

- If the word has capitalization on first letter, and is not the first word of a sentence, it probably is a proper noun, so assign NNP if singular and NNPS if plural (if it ends in -*s*).

- If the word ends in -ly, it probably is an adverb, so assign RB.
- If it ends in -able, it probably is an adjective, so assign JJ.
- If it can be cast as an int or float, or if it's a spelled out number (checked using the library *word2num*), it probably is a number, so assign CD.

The impact of these strategies on the accuracy of all tokens and unknown tokens can be seen in Table 2. To compare this results, we used the same type of model, using tri-grams and beam 3.

Method: Trigram Beam 3	**Token acc**	**Unk token acc**
No handling	93.05	13.93
With Suffix Checks	94.84	59.03
With SC + Unk tokens	95.13	66.68

Table 2: Experimentation on unknown token handling

4 Smoothing

For smoothing we tried two different approaches: add-k smoothing and linear interpolation.

Add-k smoothing involves adding an additional k count to the frequency of all -grams, using the following formula:

$$P(w_i) = \frac{C(w_i) + k}{N + kV}$$

Where $P(w_i)$ is the probability of -gram w_i occurring, $C(w_i)$ is the frequency of the -gram, N represents the total number of -grams, and V is the number of distinct tags.

This approach is simplistic, and it made us lose some information on the true probability of the most common -grams. But even with the simplicity of this approach, we managed to improve the performance of our model. This is because this smoothing prevents our model from overfitting on training data, adding some noise that reduces variability when testing on different data.

Experimentation of this approach can be found in Section 6. We did hyperparameter tuning to figure out the best value for k, and found that $k = 0.01$ worked the best.

The second approach we tried was linear interpolation. We implemented this smoothing strategy using a deterministic algorithm from (Throsten, Brants: TnT)[?]. This didn't turn out to be very good, as this method performed worse than add-k smoothing.

5 Implementation Details

We implemented the tagger following the skeleton provided in the starter code, with some tweaks and implementations that helped with overall performance and efficiency.

In the main *pos_tagger.py* file, one can specify the hyperparameters when initializing the *POSTagger* class. These hyperparameters change both the strategies used for modeling, and adjust the parameters within those strategies. All the hyperparameters used for our model are:

- *unknown_token_handling* (bool): Determines if unknown tokens should be handled using the unk token strategy described in Section 3.
- *smoothing_strategy* (str): Determines which smoothing strategy to use. Can be '*add-k'* or *'linear_interpolation'*
- *model* (str): Type of -gram used for transition probabilities. Can be *'unigram'*, *'bigram'* or *'trigram'*.
- *method* (str): Type of decoder used for making predictions. Can be 'greedy', 'beam' or 'viterbi'.
- *suffix_bool* (bool): Determines if suffix rules are used or not.
- *smooth_k* (float): Determines the k used for add-k smoothing. Default is 0.
- *unk_threshold* (int): Determines thethreschold for unk token creation. Default is 3.
- *beam_k* (int): Determines the k value for beam search. Default is 2.

We additionally implemented helper functions in our main class that help with vectorizing bigrams and trigrams for beam search implementation, specifically. We found that these strategies reduced significantly the time it took to run this tagger.

6 Experiments and Results

Test Results The implementation that performed the best was an implementation using trigrams and viterbi. This is the version that we uploaded to the leaderboard. The overall F1 score and the hyperparamenters for this submission are seen in Table 3.

Best performing POS Tagger model	
Model	Trigrams
Method	Viterbi
Suffix handling	TRUE
smoothing_strategy	add-k
smooth_k	0.05
unk_threshold	4
F1 Score	**95.8032**

Table 3: Leaderboard model characteristics and result

Smoothing The results of the experiments regarding smoothing can be seen on Table 4. Using a base model of tri-grams and Viterbi, with unknown words handling, we compared the performances on known and unknown tokens for no smoothing, add-k smoothing (using different values of k).

Smoothing	**Overall token acc**	**Unk. token acc**
No smoothing	90.15	60.41
Add-k smoothing		
– k = 0.01	95.55	69.39
– k = 0.05	95.56	69.32
– k = 0.1	95.57	69.36
– k = 0.2	95.56	69.39
– k = 0.5	95.48	69.36
– k = 1	95.38	69.28
– k = 2	95.27	69

Table 4: Experimentation on different smoothing methods

Bi-gram vs. Tri-gram Performance experiments on bigrams and trigrams can be seen on Table 5. We used beam-3 search to get the predictions, with unknown word handling and smoothing.

Model	**Overall token acc**	**Unk. token acc**
Brigram	94.88	63.47
Trigram	95.13	66.68

Table 5: Leaderboard model characteristics and result

We see an increase in performance using trigram models. this is likely because the model has access to more accurate information when calculating the transition probabilities.

Greedy vs. Viterbi vs. Beam For the last experimentation, we tried different decoding implementations. Using a base trigram model with unknown word handling and smoothing, the results from this experimentation can be seen on Table .

Decoding	**Overall token acc**	**Unk. token acc**
Greedy	94.3	61.02
Beam		
– k=2	94.92	64.67
– k=3	95.34	66.73
– k=4	95.26	66.34
Viterbi	95.55	69.33

Table 6: Experimentation on different decoding methods

As we can see, as each strategy gets more complex, the performance of the model increases. what's interesting to note is that between Beam search and Viterbi there is not a significant improvement in performance. This means that for most applications a beam search model performs adequately well and should be used, as it can get predictions much faster. Even a greedy model, together with unknown token handling and the strategies we implemented, most of the time find the optimal solution.

7 Analysis

Error Analysis Looking at the predictions for Dev data, we can find specific groups of word where our model is not performing adequately.

First, if we look at the errors and group by actual tag, we can see the tags that have the highest percentage error. Between this, we can highlight those with a big number of errors. One big case is conjugated verbs (VBN, VBG, VBP), such as "reduced", "culminating" and "own", where our model is understandably predicting that these are adjectives or nouns. In 10 to 20% of cases our model cannot determine what these words are in the speech. The numbers for these and other groups of actual tags can be seen in Table 7.

Actual Tag	**Error Ratio**	**Total number of errors**
VBN	17%	875
VBG	13%	463
VBP	11%	336
JJ	9%	1392
NN	6%	1984
NNS	6%	860

Table 7: Some of the most common errors by actual tag

Second, looking at the predicted tag in the errors, we see that our model makes a mistake with plural proper nouns (NNPS). Most of this cases are proper nouns that end with "-s", or plural nouns that are part of a larger, singular noun that have many words, all of which our suffix rule classifies as NNPS. The numbers for these and other groups of predicted tags can be seen in Table 8.

Predicted tag	**Error Ratio**	**Total number of errors**
FW	70%	16
PDT	68%	78
NNPS	67%	737
LS	62%	13
RP	42%	371
RBR	22%	85
JJR	15%	135
VBN	14%	683
JJ	12%	1971
RB	10%	801

Table 8: Some of the most common errors by predicted tag

Adjectives are also usually a weakness in our model: 9% of adjectives are wrongfully tagged, and 12% of the times we predict adjective we are wrong. Most of the times we predict adjective they are nouns, and vice-versa. Words like "executive", "record", "maximum" are common errors for our tagger.

Looking at more full sentences, our model is pretty good in general at understanding context and only makes isolated errors. It's rare to see consecutive errors, only in situations like "Closed End Bond Funds Flexible Portfolio Funds Specialized Equity", which is actually just a cluster of proper nouns and not an actual English language phrase.

Confusion Matrix The full confusion matrix can be seen at Figure 1. We can appreciate most of the confusion happens with nouns and proper nouns.

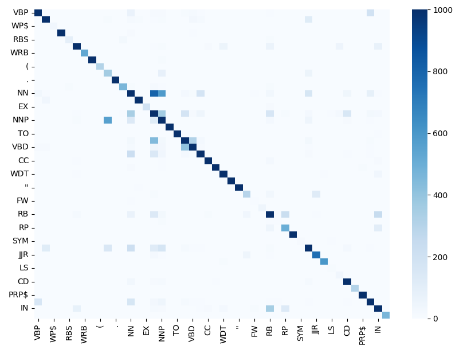

Figure 1: Confusion matrix

We can also take a closer look at a confusion matrix between the different type of verbs in Figure 2. We notice that most of the confusion happens between past tense and past principle, and between base and present.

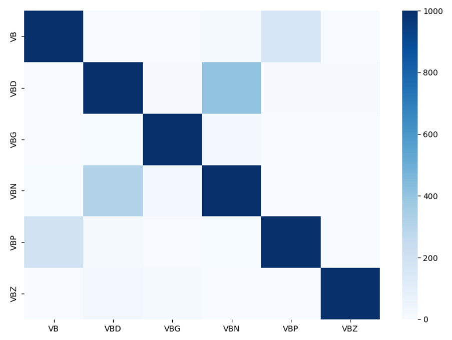

Figure 2: Confusion matrix for verbs

8 Conclusion

To conclude, we found that the best performing model is a model that uses trigrams and Viterbi enconding. Having said that, a trigram model using beam search with $k = 3$ performs much faster and has adequate similar performance for this POS tagging assignment.