CIS530: Assignment 3

Octavio E. Lima Alexander Dong

1 Introduction (5pt)

For this project, we incorporate different approaches of Part-of-Speech (P.O.S.) Tagger. That is, based on a sequence of words, we generate a sequence of their corresponding predicted P.O.S. Tags.

For illustration, the sequence of words `[‘<START>’,‘The’,‘Bird’,‘Flies’,‘<END>’]` should return the following if the model is correct: `[‘<S>’, ‘DT’, ‘NNP’, ‘VBZ’, ‘<E>’]`. To do this, we use methods known as Greedy Decoder, Beam Search, and Viterbi Algorithm. Our raw training data consists of two “flat” CSV files (`Train X` and `Train Y`) with two columns each. The columns in `Train X` are `ID` and `WORD`, whereas the columns in `Train Y` are the corresponding `IDs` and `P.O.S. TAGS`.

To evaluate our results, the main measures that we use are **(i)** Whole-Sentence Accuracy, **(ii)** Token Accuracy, and **(iii)** Unknown-Word Accuracy. For convenience, we refer to these as “WSA”, “TA”, and “UWA”. In summary, whereas the Trigram-Viterbi Algorithm is relatively difficult to implement, it yields the best results. Specifically, our Trigram-Based Viterbi smoothed by Linear Interpolation returns the following in the Development Set: `[WSA = 0.270, TA = 0.936, UWA = 0.320]`.

For comparison, the Bigram-Based Greedy algorithm returns `[WSA = 0.222, TA = 0.925, UWA = 0.280]`. Our Bigram- and Trigram-based Beam-2, Beam-3 Search outputs are better than Greedy. We conclude that the best method in our case is the Trigram-Viterbi Algorithm despite its complexity.

Lastly, the two Smoothing techniques that we use are: **(i)** Laplace and **(ii)** Linear Interpolation with normalized λs. The Linear Interpolation method has an advantage over Laplace, although the difference in performance between them is not substantial. This is further discussed in Section 6. We later show that the two techniques are both better than the absence of Smoothing.

	Dev		
Classifier	W.S.A.	T.A.	U.W.A.
Greedy (Bi.)	0.222	0.925	0.280
Greedy (Tri.)	0.224	0.926	0.283
Beam-2 (Bi.)	0.240	0.929	0.293
Beam-2 (Tri.)	0.246	0.930	0.299
Beam-3 (Bi.)	0.252	0.932	0.307
Beam-3 (Tri.)	0.262	0.933	0.309
Viterbi (Bi.)	0.246	0.930	0.181
Viterbi (Tri.)	0.270	0.936	0.320
Viterbi (Quad.)	0.264	0.933	0.303

Table 0. Main Results in the Development set; add-k smoothing with k = 0.00001

2 Data (5pt)

We work with 1,387 individual sentences in the training data; 462 in the development set; and 463 in the testing set. The distribution of words is highly skewed; meaning that a few words (such as “the”, “of”, “to”) appear much more often than others.

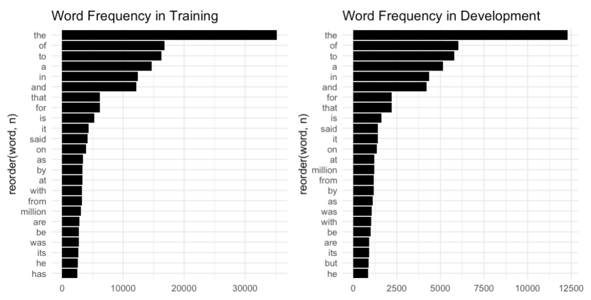


Figure 1. Frequency of 50 most common words in the Training and Development sets.

We also calculate the average and median number of words from all sentences. What is more, we calculate summary statistics from character counts — the mean and median number of characters. Table 1 indicates that the Training, Development, and Testing sets have similar distributions as observable below.

Table 1: Summary Statistics Before Preprocessing

	Train		Dev		Test	
Classifier	Mean	Median	Mean	Median	Mean	Median
Word Count	503	278	527	293	512	264
Char. Count	2,252	1,255	2,354	1,364	2,270	1,184

Table 1. Summary Statistics Before Preprocessing.

Regarding vocabulary size, we find that — with punctuation and numbers included — the training set contains 33,651 unique words, while the development and testing data sets contain respectively 18,591 and 18,259 unique words each.

We convert all words to lower case through RegEx and NumPy. Similarly, tokenization is an important step in NLP. In our case, the raw data was originally separated by white spaces, which facilitated the process; we used it to derive our Summary Statistics. We then removed certain punctuation marks to investigate word frequency using RegEx. This is important since many of the most frequent words were in fact marks such as commas, periods, and dashes.

Table 2: Summary Statistics After Preprocessing

	Train		Dev		Test	
Classifier	Mean	Median	Mean	Median	Mean	Median
Word Count	438	240	459	255	443	230
Char. Count	2,145	1,203	2,243	1,291	2,158	1,133

Table 2. Summary Statistics After Preprocessing.

As noticeable in Table 2 above, once punctuation marks and case are accounted for, the features of word and character counts decrease proportionally across the three data sets. On a final note, we decided to use all observations available to us. Even though this demands more computational power, we believe that this is beneficial. The reason is that more examples add more diversity to the model, which in turn decreases generalization error. Put another way, additional observations are generally helpful.

3 Handling Unknown Words (15pt)

The strategy that we use to handle "unknown" words (i.e., words that are not present in the training set) is to assign them tag probabilities based on their ending. As an example, words that end in "ness" or "ment" are more likely to be nouns than verbs. We apply this same approach to other cases and patterns. We do this based on Function 1 and following previous work developed by Dr. Thorsten Brants.

This approach benefits the P.O.S. Tagger because it allows the different models to consider all words that are passed to it, as opposed to having them constraint by their dictionaries. Specifically, without a reasonable technique to process unknown words, methods have scarce possibilities of tagging those same words.

Conversely, our trade off by not considering the full identity of the word is that our model will not **(i)** capture words that appear to be from a tag but are in fact from another, and **(ii)** words that can be from multiple tags depending on context. If the word **"silence"** does not exist in the training data, our model might fail to accurately predict it in *"they want to **silence** their opponent."* Here, **"silence"** is a verb despite resembling a noun. Our model would likely tag it as a noun nonetheless since it ends in "ence".

```
def unknown_words( tag_index , word ):
    Create zero−matrix of size (all_tags ,)
    if word is a digit:
        index of number reaches a higher prob
    if word ending resembles noun:
        indexes of noun reach higher probs
    if word end resembles gerund verb:
        index of gerund verb reaches a higher prob
    if word end resembles a verb:
        indexes of verb reach higher probs
    if word end resembles an adjective:
        index of adjectives reaches a higher prob
    otherwise:
      three most common tags reach higher probs
```

Function 1. Method for handling unknown words.

We call this function inside of each P.O.S. tagger method (e.g., Greedy, Beam-k, Viterbi). Precisely, if a word is present in the dictionary, we get its corresponding column in the emissions matrix and make inferences based on it. If the word does not exist in the dictionary, we tag it based on word endings and following the specifications above. Lastly, the function above takes as parameters the tag-to-index dictionary and word.

4 Smoothing (10pt)

The two techniques that we use for Smoothing are Laplace and Linear Interpolation. They are both used to remove 0% probabilities from matrices. For instance, this is done because Tag sequences that would theoretically never happen due to grammar, might still happen in the real world by accident. Therefore, assigning very low probability to these cases (as opposed to zero) would most likely improve one's model.

Specifically, Laplace Smoothing (also called add-k) consists of an assumption that every instance n in the model happens with a frequency of $n + k$ instead. In other words, before calculating probabilities in the Bigram, Trigram, and Emission matrices, we add k to each item, where k is generally a small number. In our case, we add `1e-6` (or 0.000001) to all individual counts in the three matrices.

The second method, Linear Interpolation, consists of using "(n −1)-gram probabilities to smooth n-gram probabilities."[1]. As adapted by Dr. Hockenmaier's CS-447 lecture, the sentence *"Bob was reading"* may have never been seen by the model. However, the model may have observed *"(UNK) was reading"* or simply *"(UNK) (UNK) reading."* That said, we may use a parameter λ to retrieve a linear combination of smoothed and unsmoothed probability matrices; thus eliminating 0% probabilities. For this specific example:

$$\begin{aligned}\tilde{P}(w_i|w_{i-1},w_{i-2}) =& \lambda_3 \cdot \hat{P}(w_i|w_{i-1},w_{i-2}) \\ &+ \lambda_2 \cdot \hat{P}(w_i|w_{i-1}) \\ &+ \lambda_1 \cdot \hat{P}(w_i) \\ &\text{for } \lambda_1 + \lambda_2 + \lambda_3 = 1\end{aligned}$$

For our Unigrams, Bigrams, and Trigrams respectively, we use λs with values of `0.000`, `0.901`, and `0.090`. We choose these values based on Dr. Brants's paper[2], *A Statistical Part-of-Speech Tagger*. More specifically this "technique successively removes each trigram from the training corpus and estimates best values for the λs from all other ngrams in the corpus." Lastly, we subtract 1 in the three cases to account for unseen words, thus avoiding overfitting in the training data.

Whereas smoothing techniques can be helpful to P.O.S.-Tagging models, there are some drawbacks that are worth mentioning. For instance, two of these drawbacks include **(i)** moving too much probability mass from seen to unseen events (commonly seen in Laplace Smoothing) and **(ii)** the time and computational resources that are spent finding the performance-maximizing value of the λs (eg, in Linear Interpolation).

[1]Hockenmaier, J. Lecture 4: Smoothing - UIUC. courses.engr.illinois.edu/cs447/fa2018/Slides/Lecture04.pdf

[2]Brants, Thorsten. "TNT - a Statistical Part-of-Speech Tagger." ArXiv.org, 13 Mar. 2000

Nonetheless, the benefits of these techniques far outweigh their disadvantages. This helps explain why these smoothing methods are popular and widely used. Having 0% probabilities for unseen words is both costly and detrimental to P.O.S.-Tagging models. Hence why Smoothing techniques should be implemented.

5 Implementation Details (5pt)

We use `constants.py`, `utils.py`, and `pos_tagger.py` to implement our models. In `utils`, two important helper functions are `linear_interpolation()` and `unknown_words()`. The first finds the performance-maximizing λs for one of our Smooothing techniques; the second helps us process unknown words. Lastly, we use flags in the `inference()` class of `pos_tagger` to specify which model to use. We apply multiprocessing with 6-to-8 cores.

6 Experiments and Results

Test Results (3pt)

Our best P.O.S.-Tagging model is a Trigram-based Viterbi Algorithm, smoothed by Linear Interpolation with the λs described in Section 4. Its performance was quite satisfactory, yielding an F1-Score of approximately 94.1 in the testing data set.

	Testing Data
	F1-Score
Viterbi	94.08

Table 3. Trigram Viterbi with Linear Interpolation

Smoothing (5pt)

Furthermore, we compare our Token Accuracy and Unknown-Word Accuracy by Smoothing Method. First, we use Laplace with a k of `0.00001`. We then use a Linear Interpolation Method with λs of `[0, 0.90199, 0.098009]` for Unigram, Bigram, and Trigram respectively. Lastly, we use no method of smoothing. For the later, we received some warning errors when normalizing, but these did not prevent the model from working. We observe below that the Linear Interpolation method outperforms Laplace, which in turn outperforms the absence of smoothing.

	Laplace		Linear Interp.		No Smoothing	
Classifier	T.A.	U.W.A.	T.A.	U.W.A.	T.A.	U.T.A.
Greedy (Bi.)	0.922	0.276	0.925	0.280	0.913	0.303
Greedy (Tri.)	0.923	0.284	0.926	0.283	0.916	0.305
Beam-2 (Bi.)	0.921	0.288	0.929	0.293	0.896	0.326
Beam-2 (Tri.)	0.922	0.296	0.930	0.299	0.898	0.329
Beam-3 (Bi.)	0.924	0.300	0.932	0.307	0.374	0.142
Beam-3 (Tri.)	0.925	0.301	0.933	0.309	0.448	0.150
Viterbi (Bi.)	0.23	0.177	0.930	0.181	0.104	0.001
Viterbi (Tri.)	0.930	0.311	0.936	0.320	0.029	0.000
Viterbi (Quad.)	0.926	0.297	0.933	0.303	0.042	0.140

Table 4. Dev. Set Results by Smoothing Technique

Bi-gram vs. Tri-gram (5pt)

Our overall results from the *trigram-based* Greedy, Beam-2, Beam-3, and Viterbi algorithms outperform those of their *bigram-based* counterparts. I.e., the mean values of WSA, TA, and UWA across methods are higher for Trigrams than for Bigrams. In addition, our *fourgram-based* averages are higher than the others overall, as seen below.

	Dev		
Classifier	W.S.A.	T.A.	U.W.A.
Bigram (Mean)	0.240	0.929	0.265
Trigram (Mean)	0.250	0.931	0.303
Fourgram (Mean)	0.259	0.930	0.308

Table 5. Dev Results: Bigrams vs. Trigrams vs. Fourgrams with Linear Interpolation Smoothing

Greedy vs. Viterbi vs. Beam (10pt)

Here, we compare our best-performing P.O.S. Taggers. These are the **(i)** bigram-based Greedy, **(ii)** trigram-based Beam-3, and **(iii)** trigram-based Viterbi Algorithm — all smoothed by Linear Interpolation. The trigram-based Viterbi model outperforms all other P.O.S. Taggers (including our fourgram-based Viterbi method).

Classifier	W.S.A.	T.A.	U.W.A.
Greedy*	0.224	0.926	0.283
Beam-k*	0.262	0.933	0.309
Viterbi*	0.270	0.936	0.320

Table 6. Dev Results: *Best-Performing Taggers

We further explore the Beam-k algorithm with different k-values, ranging from 2 to 6. We highlight the corresponding Precision, Sensitivity, and F1-Score calculations below.

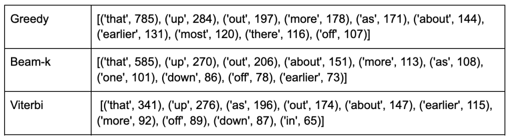
	Dev		
k	Precision	Recall	F1
2	0.858	0.762	0.782
3	0.861	0.773	0.793
4	0.861	0.772	0.793
5	0.862	0.779	0.800
6	0.862	0.777	0.798

Table 7. Precision, Sensitivity, and F1 by k-values.

7 Analysis

Error Analysis (9pt)

On the down side, our models tend to underperfom on words such as `["that", "up", "as"]`. This is understandable since these can belong to different tags depending on context. For instance, **"as"** may be **(i)** a preposition in *"She works as a developer,"* and **(ii)** an adverb in *"Nylon is cheaper than leather, and it's just as strong."*



Greedy	[('that', 785), ('up', 284), ('out', 197), ('more', 178), ('as', 171), ('about', 144), ('earlier', 131), ('most', 120), ('there', 116), ('off', 107)]
Beam-k	[('that', 585), ('up', 270), ('out', 206), ('about', 151), ('more', 113), ('as', 108), ('one', 101), ('down', 86), ('off', 78), ('earlier', 73)]
Viterbi	[('that', 341), ('up', 276), ('as', 196), ('out', 174), ('about', 147), ('earlier', 115), ('more', 92), ('off', 89), ('down', 87), ('in', 65)]

Table 8. Error Analysis

Confusion Matrix (5pt)

Common tags where our best model underperforms include `[[NNS, NN], [FW, NN], [POS, IN]]`. These could be further improved by adjusting the hyperparameters. That being said, these are sensible mistakes that do not undermine our overall performance.

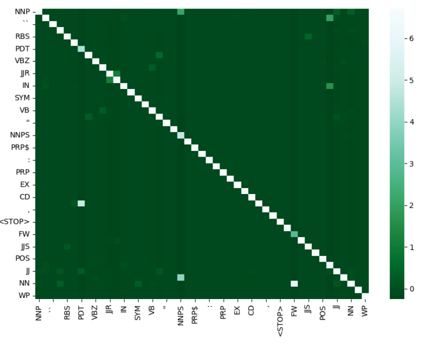


Figure 2. Confusion Matrix with Standardized Values

8 Conclusion (3pt)

We conclude that even our modest model, Greedy, did a reasonable job predicting P.O.S. tags. Needless to say, it could be improved much further with the more sophisticated algorithm, named Viterbi.