CS5740: Assignment 3

https://github.com/cornell-cs5740-20sp/assignment-3-asd

Chris Shei
cs856

Kushal Singh
ks2377

1 Introduction (5pt)

In this abstract, we implement and discuss several models for part-of-speech (POS) tagging. A POS tagger classifies each word in a sentence according to its lexical category. For example, consider the following sentence: "A shareholder filed suit". In this case, the correct POS tags would be: ("A", *DT*), ("shareholder", *NN*), ("filed", *VBD*), ("suit", *NN*), where *DT* is abbrev. for "Determiner", *NN* is abbrev. for "Noun", *VBD* is abbrev. for "Verb, past tense".

The data used in this assignment contains approx. 1 million words of text from the Wall Street Journal. The words and symbols that comprise each document are written in column format, where every document begins with the tag -DOCSTART-.

2 Data (5pt)

Table 1: Data Statistics

Metric	Value
Total Unprocessed Tokens (Training)	696475
Unique Vocab Size (Training)	37504
Number of Sentences (Training)	28837
Total Unprocessed Tokens (Dev)	243021
Unique Vocab Size (Dev)	20705
Number of Sentences (Dev)	10078
Total Unprocessed Tokens (Test)	236582
Number of Sentences (Test)	9818
Total Parts of Speech	46

From the training data statistics, we can observe that tags and words are fairly sparse. Excluding the START and STOP tags, there are 46 tags, with the most common tag being *NN* (accounting for 13.9% of all tags). The top 5 tags are *NN, IN, NNP, DT, JJ* and they comprise 49.0% of all tags. There are 37,504 unique words in the training data, of which 17,592 words (46.9%) only appear once.

For our preprocessing, we first loaded the data, and applied a mapper function to each word. This mapper function either: (1) returned the raw word, as is, if it appeared more than twice in the training set, OR (2) returned a token, based on its category, which is defined in the section below, OR (3) returned a suffix, if it was informative, which is also defined in the section below, OR (4) returned the RARE key to denote that it didn't meet any of the other criteria mentioned in options (1)-(3).

Then, we create unigrams, bigrams, trigrams, and quadgrams of words in sentences and tag them with how many times a sequence of words is followed by a particular tag. For instance, for unigrams, we count how many times an *NNP* is followed by another *NNP*, how many times a *DT* is followed by an *NN*, etc. Similarly, for bigrams, we count how many times an (*NNP*, NNP) sequence is followed by another *NNP*. The same logic holds for trigrams and quadgrams. We then store all of these dictionaries in a global dictionary ($transition - counts$) representing the total counts of transitioning from different n-gram sequences to a particular POS tag, where the keys 1, 2, 3, 4 correspond to the dictionaries for unigrams, bigrams, trigrams, and quadgrams, respectively.

We also create a dictionary mapping how many times a sequence of tags appears for each n-gram. For instance, in the unigram case, we store how many NNs, NNPs, NNSs, etc., appear. In the bigram case, we store how many (*RBS*, *VBN*), (*NNP*, *NNP*), etc., sequences actually appear. We then store all of these dictionaries in a global dictionary ($state - counts$), where the keys 1, 2, 3, 4 correspond to the dictionaries for unigrams, bigrams, trigrams, and quadgrams, respectively.

3 Handling Unknown Words (15pt)

To handle unknown and rare words, we leverage three different methods.

Categories: We categorize unknown words based on certain shared properties. Among other characteristics, these words are considered special, since they contain numbers, hyphens, and are case sensitive. The categories for these words include: Numbers (e.g. *25*), Words that only start with one capital

letter (e.g. *Grace*), Words with all capital letters (e.g. *FERC*), Words with all capital letters that end in '*s*' (e.g. *UFOs*), Words that contain both letters and numbers (e.g. *pre-1917*), Numbers that end in '*s*' (e.g. *1950s*), Words that are connected to a lowercase word by a hyphen (e.g. *dollar-denominated*), Upper-case words that are separated by a hyphen (e.g. *Chinese-American*, *No-Smoking*). Words in these categories are generally tagged as *CD* (cardinal number), *NNP* (noun, proper singular), *NNP*, *NNS* (noun, plural), *JJ* (adjective), *CD* or *NNS*, *JJ*, *NNP*, respectively.

Suffixes: Suffixes ended up being a strong predictor for part-of-speech tags. If an unknown word does not fit into one of the granular categories defined above, then we consider its suffix of length 2. We create a key, value pair of (*suffix, POS tag*) and append that to our suffix dictionary. We only consider a suffix to be informative if its (*suffix, POS tag*) appears more than 7 times. We observed that the total number of suffixes in the training data prior to and subsequently after applying the 7-count filter was equal to 850 and 109, respectively. The (*suffix, POS tag*) tuples that appeared the most were (*VBG, "-ng"*) at 974 times, (*VBN, "-ed"*) at 709 times, (*NNS, "-es"*) at 609 times, (*RB, "-ly"*) at 444 times.

RARE Key: If a particular word does not fall into any of the aforementioned categories detailed above, or did not have an informative suffix associated with it, then that word is replaced with the RARE key. In total, there were 697 such words in the training set.

The table below describes our performance on handling unknown words for the development set.

Table 2: Unknown Word Performance

Group	Count	Percent
Numbers	1736	99.5%
Numbers + 's'	11	72.8%
Uppercase sep. by hyphen	118	73.8%
Word + Hyphen + Lowercase	1314	72.7%
First Letter Uppercase	4990	79.6%
All Letters Uppercase	775	70.2%
All Letters Uppercase + 's'	2	100%
Words w/ letters and numbers	156	82.1%
Total Number of Suffixes	6104	80.9%
Pure Unknowns	328	66.2 %

4 Smoothing (10pt)

Smoothing flattens a probability distributions of particular n-grams, such that **all** word sequences can occur with some probability. The motivation behind this is so that unseen words don't have emission probabilities equal to 0, if the training data is sparse. We can achieve this by redistributing weights from high probability regions to zero probability regions.

For our first smoothing technique, we used Add-k (aka Laplace) smoothing, which pretends that we saw each sequence k more times than we actually did. In the case of add-1 smoothing, for example, we pretend that we saw each word one more time than we actually did (so unseen words aren't completely whitened). This is a non-maximum likelihood estimator because we are changing the counts of what we observed in our training data to hope to generalize better in our development and test data. The general form of Add-k smoothing is:

$$P_{add-k} = \frac{c(x_{i-1}, x_i) + k}{c(x_{i-1}) + kV}$$

The second smoothing technique we used was backoff (aka Katz) smoothing. In our implementation, we computed the adjusted counts of different n-grams, and for n-grams with nonzero count r, we discounted according to a discount ratio d_r, which is detailed in the Lecure 7 slide. We used this smoothing technique because it tends to perform better on large training sets and on n-grams with large counts.

5 Implementation Details (5pt)

We implement and discuss three models for POS tagging: Greedy, Viterbi, and Beam Search.

The greedy search algorithm maximizes the likelihood between neighboring states by searching for the most likely transition between the current state and the subsequent state. Beam search is an extension of greedy search, which finds the top k possible transitions between the $i - 1_{st}$ token and the i_{th} token. Viterbi uses dynamic programming to find the global optimal POS tags.

No optimizations were needed to improve runtime performance. The hyper-parameters used included: threshold for generating suffixes, n for optimal n-gram sequences, k for beam search, k in add-k smoothing.

6 Experiments and Results

Test Results (3pt) We generated this test score using our procedure for handling unknown words described above. The hyper-paramater values were $n = 3$ for trigram sequences, $k = 3$ for beam search, and $k = 0.2$ for add-k smoothing.

N-gram	Beam	Add-k	Test Accuracy
n = 3	k = 3	k = 0.2	0.94857

Smoothing (5pt)

	Add-K Smoothing	Unknown Acc
k=0.2	0.94857	0.8061
k=0.4	0.94848	0.7985
k=0.6	0.94829	0.7926
k=0.8	0.94807	0.7910
	Backoff	
	0.9402	0.7965
	No Smoothing	
	0.9438	0.7929

As shown in the table above, add-k smoothing with $k = 2$ had the best performance, but only marginally. Interestingly enough, no smoothing technique seems to perform better than backoff smoothing.

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

	Accuracy	
Methods	Overall	Unknown
Bigram, k=3	0.94216	0.79128
Trigram, k=3	0.94857	0.80487
Bigram, Viterbi	0.94220	0.79265
Trigram, Viterbi	0.94848	0.80229

Based on the results from the table above, we can see that increasing gram size does not show remarkable improvement in accuracy for overall and unknown words. The bigram and trigram models performed fairly similar to each other. Despite the smaller context window size, the bigram results seem to be higher than expected, suggesting that bigrams have enough context to achieve good results on the training set. Increasing gram size also increases the search space sparsity and potential noise for transition and emission probabilities.

Greedy vs. Viterbi vs. Beam (10pt)

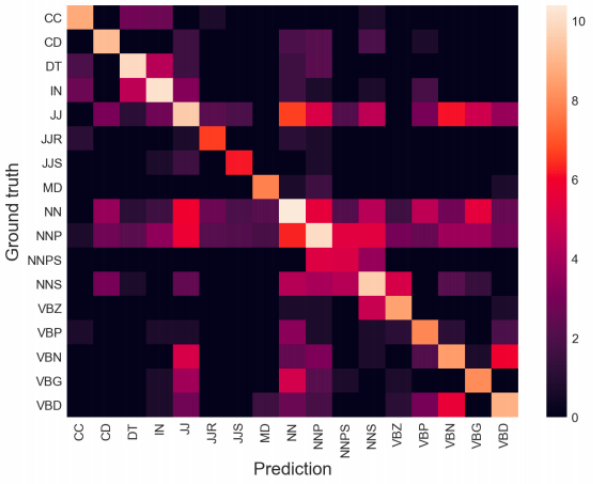
Method	Accuracy	Suboptimal	Unk Acc
Greedy	0.93965	0.785	0.7809
Beam, k=2	0.94109	0.702	0.7987
Beam, k=3	0.94857	0.701	0.8049
Beam, k=4	0.94762	0.705	0.8017
Beam, k=5	0.94513	0.709	0.7992
Viterbi	0.94798	0	0.8022

From the table, we can see that beam size had minimal impact on accuracy and solution optimality. Even though greedy rarely finds the optimal sequence, it still yielded a fairly high accuracy.

7 Analysis

Error Analysis (7pt) Based on the error analysis we conduced and the confusion matrix shown below, we observed that one of the most common misclassifications was when the tagger mapped *NN* and *NNP* to *JJ* (i.e. nouns to adjectives). There were also some *VBN* and *VBG* that were misclassified as *JJ* because a significant amount of *VBN* and *VBG* that end with *-ed* and *-ing* are actually adjectives. One such example was the word "retired", which was tagged as *VBD* when it was actually *JJ*. This type of error is hard to handle because it is an edge case in the English language, where a verb's past tense can act as as an adjective.

Confusion Matrix (5pt) With regards to the POS tags, there were two predominant clusters which align with normal English sentences: Nouns (e.g. *NN, NNP, NNPS, NNS*) and Verbs (e.g. *VBP, VBN, VBG, VBD*). Because the differences within each of these clusters are very nuanced, most misclassifications occurred within these clusters. For example, the word "Books" was classified as *NNS*, even though its actual POS tag is *NNPS*.



8 Conclusion (3pt)

We explored different search methods (greedy, beam, Viterbi), unknown words handling, and gram sizes. Handling unknown words proved crucial to predicting POS tags due to their direct impact on common words via transition emissions. To handle this, we grouped unknown words into one of several categories, used bigram suffixes where necessary, and added a RARE Key for words that didn't fit into either of the above handles. Gram size didn't play a huge factor in determining accuracy, possibly because context information was not as crucial as unknown word handling and smoothing. Greedy search showed the worst performance, while beam search had a very similar performance with Viterbi. Increasing width of beam search marginally improved accuracy. Next steps would be to leverage a better smoothing method to improve model performance.