CS 6741 - NLP Assignment 1
Brandon Plaster

Using the code.

All the new code for this assignment exists in “POSTaggerTester.java”.

Command Line Options

- Required:
  - -path <DATA PATH>
    - where <DATA PATH> is the location of the root folder (typically “wsj/”)
    - Note: Twitter data file (named pos.txt) should be located in path folder.
- Optional:
  - -decoder <DECODER NAME>
    - where <DECODER NAME> is either “viterbi” or “greedy”
    - defaults to “greedy”
  - -tagger <TAGGER NAME>
    - where <TAGGER NAME> is either “hmm” or “frequent”
    - defaults to “frequent”
  - -traincorpus <CORPUS NAME>
    - where <CORPUS NAME> is either “penn”, “twitter”, or “both”
    - defaults to “penn”
  - -testcorpus <CORPUS NAME>
    - where <CORPUS NAME> is either “penn” or “twitter”
    - defaults to “penn”
  - -test <TEST SET>
    - where <TEST SET> is either “validation” or “test”
    - defaults to “test”

Problem 1. Building a Sequence Model

I began by following along with the simple trigram HMM model described by Collins [1] where the score of a tagset is the product of the emission $p(w|t)$ (probability of a word w given a tag t) and the transition $p(t_3|t_2,t_1)$ (probability of a tag t_3 given preceeding tags t_2 and t_1).

To determine the emission probability, using bayes theorem I rewrote the emission as:

$$p(w|t) = \frac{p(t|w)*p(w)}{p(t)}$$

For $p(w)$, for known words I kept a word count, and for unknown words, I set the probability to be the average probability of all known words, which is just $\frac{1}{total\ number\ of\ words}$. Initially, before creating a more sophisticated unknown words model, for $p(t|w)$, I used the distribution of $p(t)$ over all known words.

CS 6741 - NLP Assignment 1
Brandon Plaster

To determine the transition probability, initially I used a simplistic model where:

$$p(t_3|t_2,t_1) = \frac{count(t_3,t_2,t_1)}{count(t_2,t_1)}$$

This resulted in a NULL pointer error because there did not exist all cases with the stop state, so attempting to count trigrams with the stop state resulted in 0, which caused the stop state not to be set, and so a NULL pointer was reached. My initial method for dealing with this was a simple smoothing of counts by incrementing all possible trigrams and bigrams by a fixed number that I allowed to be a tunable parameter.

At this point, the results for the known words increased to ~94% but the unknown words were still below 50%. To help the unknown word model, I decided to implement a suffix-based model shown by Brants [2]. The suffix model replaces the unknown word model used in $p(t|w)$ by replacing it with a combined distribution of $p(t|suffix)$ for suffixes of length [0, m]. Brants uses a recursive implementation that is based on weights given to each suffix length (although, he finds that it's best if the weights are all the same). However, I found that I got better results by simply averaging, then normalizing the $p(t|suffix)$ over all the suffixes. Similarly to Brants, I found that having a maximum suffix length of 10 optimized my results. Though, another improvement on his paper that I found was to not count the suffixes of words that were less than three letters. In reference to the capitalization of unknown words, initially, I tried keeping a single counter, but I found that my results improved using separate counters for lowercase and capitalized unknown words.

From here, I decided to revisit my simplified smoothing method. I went about following Brants [2] again by replacing my calculation of $p(t_3|t_2,t_1)$ with a weighted summation of probabilities that, unlike my original method which only factored in trigram probabilities, also took into account the effects of the bigram and unigram probabilities. This resulted in:

$$p(t_3|t_2,t_1) = \lambda_1 P(t_3) + \lambda_2 P(t_3|t_2) + \lambda_3 P(t_3|t_2,t_1)$$

where the maximum likelihoods (P) are and weights (λ) were determined based on the associated distributions. I then tuned the model using my original smoothing counter (between a range of [0,1]) and I noticed that at this point, the overall simplified smoothing was actually hurting the model, and because of the new interpolation, it was no longer needed, so I commented it out.

With the new interpolation and the new unknown word model, the results were significantly improved from the base model as seen below:

CS 6741 - NLP Assignment 1
Brandon Plaster

Tagger	Validation Set % (known / unknown)	Test Set % (known / unknown)	Suboptimalities (validation / test)
Most Frequent	91.8 / 38.0	92.2 / 38.8	0 / 0
HMM	94.9 / 79.5	95.1 / 72.7	353 / 573

As can be seen, the HMM model had many suboptimalities, which are addressed in Problem 3.

Problem 2. Building a MaxEnt Tagger

Incomplete

Problem 3. Building a Sequence Decoder

Building a viterbi decoder actually turned out to be a very time consuming task because of how the original base code implemented its trellis. That, coupled with my unfamiliarity with java, resulted in my need to reimplement it several times before finally succeeding. This, however, did force me to understand the implementation a lot more, and resulted in digging deeper into the representation of the language in the trellis.

Initially, I had tried passing in all the possible tags, then creating a matrix of all the tags, similar to what was shown in the lecture notes. The issue with simply iterating over this was that the way the trellis stores the transition probabilities. Each “state” in the trellis is actually a set of two tags, and so a transition in the trellis, I discovered, was actually the transition between a set of tags to another set of tags. As there is no direct method of simply retrieving the transition probability between one tag and another tag, I needed to rethink how I was implementing the viterbi decoder. In order to debug these issues, I printed out trellis’ for short sentences that were still deemed suboptimal. Eventually, I realized, by dynamically creating each set of possible transitions from each tag at a specific position in a sentence, then I could create a non-square array that would contain all possible non-zero sequences of the sentence.

The comparison of before and after results are below:

Decoder	Validation Set % (known / unknown)	Test Set % (known / unknown)	Suboptimalities (validation / test)
Greedy	94.9 / 79.5	95.1 / 72.7	353 / 573
Viterbi	95.9 / 82.8	96.4 / 78.9	0 / 0

CS 6741 - NLP Assignment 1
Brandon Plaster

Problem 4. Out-of-domain Experiments

A note on running the twitter data: compared to running the Penn Treebank data, running the Twitter test data takes quite a bit longer (about 3x). Although this only increased the run time from roughly 1 minute to 3 minutes on my computer, it may vary based on the computer.

Below are the results for the out-of-domain experiments:

Training Set	Test Set	Validation Set % (known / unknown)	Test Set % (known / unknown)
Penn	Penn	95.9 / 82.8	96.4 / 78.9
Twitter	Twitter	-	78.9 / 49.1
Penn	Twitter	-	74.3 / 25.4
Penn + Twitter	Twitter	-	83.1 / 47.4

As can be seen, the results for running the Twitter data are abysmal. This can especially be seen in the set of unknown words, as there are a significantly higher number of unknown words in the dataset. This can be explained by it being a much smaller training set and there being a much greater number of possible words in the test set than in the Penn Treebank. This is because tweets oftentimes contain misspellings, hashtags, urls, and usernames. A possible solution to the hashtag problem, though requiring domain specificity, is that by removing the hashtags, it will decrease the number of possible words, since usually what follows a hashtag is a word. Alternatively, by simply looking at the first letter of a word, if you see that it begins with a hashtag, then you can classify it as an HT, or if it begins with an “@”, then you know it is a USR. Another possibility is by including a dictionary for looking up misspellings, it’s possible to replace misspelled words to have a higher likelihood of guessing the tag.

A simple test, where I looked at the first character of a word and chose the tag to be HT or USR based on if it was # or @, respectively, gave the improved results:

Training Set	Test Set	Before Correction % (known / unknown)	After Correction % (known / unknown)
Twitter	Twitter	78.9 / 49.1	80.2 / 53.6
Penn	Twitter	74.3 / 25.4	78.1 / 42.9
Penn + Twitter	Twitter	83.1 / 47.4	85.0 / 60.2

The unknown word issue is even more significant when training on the Penn Treebank alone, because many of the words and symbols that are unique to Twitter are not found in the Treebank. Even more so is the fact that not all the tags that exist in the Twitter set actually exist in the Penn Treebank, which means that it has no way of guessing an accurate tag for missing tags.

Also, defined by the assignment problem, there are more sentences in the test set than that of the Penn Treebanks test set and less in the training set, which inherently increases the number of unknown words. Sentences are also ill-defined because many of the tweets contain multiple sentences, but are deemed as a single sentence in the test and training sets. One solution for this would be to split up the tweets by periods as well. The size of the training set is also a significant factor in the known words accuracy. This can be shown by just increasing the size of the training set by having both the Penn and Twitter sets, the known word accuracy increases.

Another issue is that there are many all-uppercase words, which impact the results of the unknown word classes because there are two counters that distinguish between a capitalized and lowercase word. A possible solution to this would be to lowercase all words that are fully capitalized and not consider those as capitalized words.

References

[1] http://www.cs.columbia.edu/~mcollins/courses/nlp2011/notes/hmms.pdf
[2] http://www.coli.uni-saarland.de/~thorsten/publications/Brants-ANLP00.pdf