Part of speech tagging is the process of assigning parts of speech to the words in a given string. A probabilistic tagger assigns the most likely tags based on a set of probabilities that it is given. These probabilities can be calculated simply by counting the tags and words in a corpus of pre-tagged data.

The disadvantage of this, though, is that it requires a large amount of pre-tagged data. Another way to find the probabilities the tagger needs is to seed a tagger with a small amount of pre-tagged data, and use it to tag a large corpus of untagged data. This tagged data can then be used to estimate the probabilities again. The program I wrote repeats this process over and over, making re-estimated probabilities each time. The final set of probabilities are given to the tagger.

A problem with this method, however, is that it might re-estimate some probabilities in a way that is not beneficial. Since humans tagged the set of tagged data, we know that the tagged data is correct. However, the algorithm does not know anything about language, and may change the probabilities in a way that does not reflect natural language. In this paper, I will examine whether keeping certain probabilities from the seed data helps the tagging be more accurate.

The first step to doing this is to make a tagger that can tag strings of words once it has the probabilities. Given a sentence of words $w_1 \ldots w_n$, a tagger needs to find the set of tags $t_1 \ldots t_n$ that is most likely to produce the given words. That is, it needs to find the string of tags $t_1 \ldots t_n$ that maximizes

$$P(t_1 \ldots t_n \mid w_1 \ldots w_n)$$

Of course, there are (number of tags)^n possibilities for $t_1 \ldots t_n$, so we cannot possibly go through them all. A bigram tagger computes the string of tags using the (incorrect)

assumption that the probability of each tag is dependent only on the tag and word next to it.

For each position *i* in the string, a bigram tagger needs to compute the tag t_i at position *i* that is most probable. It does this by using the forward and backward probabilities for the string. The forward probability $\alpha_i(t)$ is the probability of having tag *t* at position *i* after having seen the first *i* words, $w_1 \ldots w_i$. That is,

$$\alpha_i(t) = P(w_1 \ldots w_i \, , \, t_i = t)$$

The backward probability $\beta_i(t)$ is the probability of seeing the words $w_{i+1} \ldots w_n$ given that the tag at the i^{th} position is *t*. That is,

$$\beta_i(t) = P(w_{i+1} \ldots w_n \mid t_i = t)$$

These probabilities are computed recursively. We can calculate the probabilities based on a sum over all transitions from one state to the next:

$$\alpha_1(t) = P(\$ \rightarrow t) \;\; P(t \rightarrow w_1)$$

$$\alpha_i(t) = \sum_{u \text{ Tags}} \alpha_{i-1}(u) \;\; P(u \rightarrow t) \qquad P(t \rightarrow w_i)$$

and

$$\beta_n(t) = P(t \rightarrow \$)$$

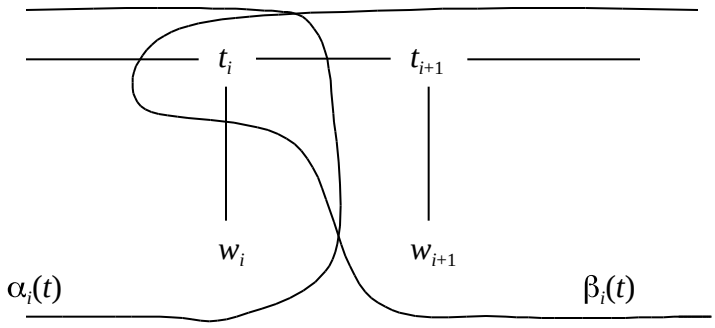
$$\beta_i(t) = \sum_{u \text{ Tags}} \beta_{i+1}(u) \;\; P(t \rightarrow u) \;\; P(u \rightarrow w_{i+1})$$

where \$ is the end of string tag, $P(t \rightarrow u)$ is the probability that tag *t* transitions to tag *u*, and $P(t \rightarrow w_i)$ is the probability that word w_i is emitted by tag *t*. The fact that these formulas compute the probabilities for the substrings shown in their definitions is due to the forward and backward probabilities themselves being in the sums.

Using the assumption that the tag at a position is dependent only on the positions next to it, we can now find the most probable tag t_i at position i. The forward probability $\alpha_i(t)$ is the probability of having tag t at position i after having seen the first i words. The backward probability $\beta_i(t)$ is the probability of seeing all words after position i given that we have tag t at position i. Thus, the product $\alpha_i(t)\ \beta_i(t)$ is the probability that tag t is at position i having seen the entire string of words. So all we have to do is choose t_i to be the tag t that gives us the maximum value of the product $\alpha_i(t)\ \beta_i(t)$. That is, we choose t_i to be

$$t_i = \arg\max_t(\alpha_i(t)\beta_i(t))$$

This can be seen visually with the following picture:



To implement the tagger, it is natural to make the forward and backward probabilities two dimensional arrays, indexed by the position i and tag t. However, since most of the P(tag → word) probabilities will be 0, we can optimize by calculating the forward and backward probabilities at each position i only for those tags that can actually emit the word w_i. Thus, in my implementation, I store the forward and backward probabilities as an array (or vector) indexed by the position i. Each entry of the array stores a map (or hashtable) mapping tags to probabilities. For example, the probability $\alpha_i(t)$ would be the number mapped to by t in the map stored at entry i of the array α. (In

C++, the STL square-bracket operators for map lookups seemed very elegant, since then the forward and backward probability tables could be used just like two-dimensional arrays.)

To get the probabilities used by the tagger, the most straightforward thing to do is to count the number of times we see each tag to tag transition and tag to word emission in a pre-tagged corpus. We can then divide by the total number of times we saw each tag to get the probabilities:

$$P(t \rightarrow u) = \frac{c(t \rightarrow u)}{c(t)}$$

$$P(t \rightarrow w) = \frac{c(t \rightarrow w)}{c(t)}$$

where $P(.)$ is probability, and $c(.)$ is the count of the number of times we saw the thing in parentheses. ($t \rightarrow u$) is a transition from tag t to tag u, and ($t \rightarrow w$) is an emission of word w from tag t.

Most counts, however, will be zero. So we need a way of smoothing the probabilities. If we do not smooth, the zeros could propagate into all of the forward and backward probabilities, and the program will not be able to tag anything. For the tag to tag transition probabilities, I made any probability that was 0 into a small number, around 10^{-10}. For the tag to word emission probabilities, I used a small amount of pre-tagged data to estimate $P(t \rightarrow \text{UNKNOWN})$ for each tag t, where $P(t \rightarrow \text{UNKNOWN})$ is the probability of t emitting an unknown word (i.e., a word that wasn't in any of the data). Then the emission probability for each tag t to any word that has not been encountered yet is $P(t \rightarrow \text{UNKNOWN})$. The emission probabilities for words w that are known is

$\hat{P}(t \rightarrow w) = P(t \rightarrow w)\ (1 - P(t \rightarrow \text{UNKNOWN}))$, where $P(t \rightarrow w)$ is the probability before smoothing.

If we do not have a large pre-tagged corpus, we cannot find the probabilities by straight counting. Another way to estimate the probabilities is to use a small amount of pre-tagged data to get seed probabilities for a tagger. We can then use this tagger to tag a large untagged corpus. This corpus, once it is tagged, can be used to re-estimate the probabilities using direct counts. The process of tagging the untagged data, and using the newly tagged data to re-estimate the probabilities, can be repeated however many times. The result is the probabilities that a tagger can use.

It turns out that in my tests, I found that my implementation's tagging accuracy actually goes *down* with more iterations. This leads me to believe I might have made a mistake implementing the algorithm or tagger, but I cannot find it.

One of the problems with this algorithm is that it does not know anything about natural language. Thus, in order to get a better result from the tag transition probabilities, it might change some word emission probabilities to things that are completely different from the natural language. For example, it might start tagging the word "away" as a verb if it happens that a verb fits best in all the places where the word "away" occurs in the data.

A possible solution to this is to not change the tag to word probabilities for any words that are in the seed data. Since the seed data was tagged by humans, we know the tags for those words are correct. Thus, it might be beneficial if we don't touch those probabilities when re-estimating. I wrote two versions of this: one where the only word emission probabilities that are changed are those for words not in the seed data, and one

where the probabilities are never changed. I did not find much difference between these two versions.

For the first version, where I changed the emission probabilities for words not in the seed data, I weighted the probabilities so that all of the words that were not in the seed data fit into the unknown words space of the seed data. That is, for words in the seed data, the probability is

$$\hat{P}(t \rightarrow w) = P(t \rightarrow w)\ (1 - P(t \rightarrow \text{UNKNOWN}_{\text{SEED}}))$$

where $\text{UNKNOWN}_{\text{SEED}}$ represents words not in the seed data. For words in the untagged data but not in the seed data, the probability is

$$\hat{P}(t \rightarrow w) = P(t \rightarrow w)\ \ P(t \rightarrow \text{UNKNOWN}_{\text{SEED}}))\ (1 - P(t \rightarrow \text{UNKNOWN}_{\text{UNTAGGED}}))$$

where $\text{UNKNOWN}_{\text{UNTAGGED}}$ represents words not in the untagged data. For the unknown words (that is, words not in the seed data or untagged data), the probability is

$$\hat{P}(t \rightarrow \text{UNKNOWN}) \ = P(t \rightarrow \text{UNKNOWN}_{\text{SEED}})\ \ P(t \rightarrow \text{UNKNOWN}_{\text{UNTAGGED}})$$

I implemented all the things in this paper in C++. The code is on CS, at /u/deigen/course/cg136/final/. For data, I used the Wall Street Journal DPG tagged corpus. My results are on the following page. In the table, the number of iterations means the number of times the probabilities were estimated using the untagged data.

Description	DPG Sections Used				Average Correct
	seed	untagged	estimating unknown probabilities	testing	
straight count of tagged data	2 - 21	-	24	23	0.937689
seed data only (no untagged)	2 - 5	-	24	23	0.874236
estimating with untagged — one iteration	2 - 5	6 - 21	24	23	0.910361
estimating with untagged — four iterations	2 - 5	6 - 21	24	23	0.904306
estimating with untagged, with emission probability freezing — one iteration	2 - 5	6 - 21	24	23	0.905889
estimating with untagged, with emission probability freezing — four iterations	2 - 5	6 - 21	24	23	0.904637

According to these results, freezing the emission probabilities makes the performance worse in the one iteration case. However, in the four iteration case, it makes the average correct a little better. Thus, according to these numbers, I believe that freezing the emission probabilities of the seed data probably does not produce better results. In addition, the difference in performance between freezing and not freezing the results is little.

References

Jurafsky, Daniel and Martin, James H., *Speech and Language Processing*, Upper Saddle River, NJ: Prentice Hall, 2000

Johnson, Mark and Geman, Stuart, *Probability and Statistics in Computational Linguistics, a brief review*, Dec. 7, 2000