IITP-03 Assignment 1: POS Tagging

YongKyung Oh*
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213
yongkyuo@andrew.cmu.edu

Abstract

POS tagging forms an important part of NLP workflow for most modern day NLP systems. Within the NLP community, POS tagging is largely perceived as a solved problem, or at least well enough solved such that most people don't put much effort into improving POS tagging for its own sake. I use the hidden markov model (HMM) and viterbi algorithm to conduct POS tagging. I tried two directions. First, I develop own tri-gram model using HMM. Second I tried to improve the bi-gram / tri-gram viterbi model. To deal with the unsean data, I use the linear interpolation and good-turing estimator. Overall performance is slightly improved and I can see the way of how to improve the POS tagging.

1 Task 1

Task 1: A learning curve is a useful tool that lets you visualize how a model's performance depends on the amount of training data. The learning curve plots a performance measure evaluated on a fixed test set (y-axis) against the training dataset size (x-axis).

function VITERBI(*observations* of len T,*state-graph* of len N) **returns** *best-path*, *path-prob*

create a path probability matrix *viterbi[N,T]*
for each state s **from** 1 **to** N **do** ; initialization step
  $viterbi[s,1] \leftarrow \pi_s * b_s(o_1)$
  $backpointer[s,1] \leftarrow 0$
for each time step t **from** 2 **to** T **do** ; recursion step
  for each state s **from** 1 **to** N **do**
    $viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
    $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
$bestpathprob \leftarrow \max_{s=1}^{N} viterbi[s,T]$; termination step
$bestpathpointer \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T]$; termination step
$bestpath \leftarrow$ the path starting at state *bestpathpointer*, that follows backpointer[] to states back in time
return *bestpath*, *bestpathprob*

Figure 1: Overview of viterbi algorithm

Parts of speech (also known as POS, word classes, or syntactic categories) are useful because they reveal a lot about a word and its neighbors. Part-of-speech tagging is the process of assigning a part-of-speech marker to each word in an input text. The input to a tagging algorithm is a sequence of (tokenized) words and a tagset, and the output is a sequence of tags, one per token.

*UNIST, ok19925@unist.ac.kr

A hidden Markov model (HMM) allows us to talk about both observed events (like words that we see in the input) and hidden events (like part-of-speech tags) that we think of as causal factors in our probabilistic model.

The decoding algorithm for HMMs is the Viterbi algorithm. Viterbi algorithm for finding the optimal sequence of tags. Given an observation sequence and an HMM $\lambda = (A, B)$, the algorithm returns the state path through the HMM that assigns maximum likelihood to the observation sequence. [1]

1.1 Learning Curve

I tested the relationship between sample size and error rate. Sampling ratio is determined first, and shuffled sample data is selected according to the sample number. In the following graph, x-axis is sampling ratio and y-axis is error by word / error by sentence respectively. I tested 20 samples from 0.05 to 1.00.

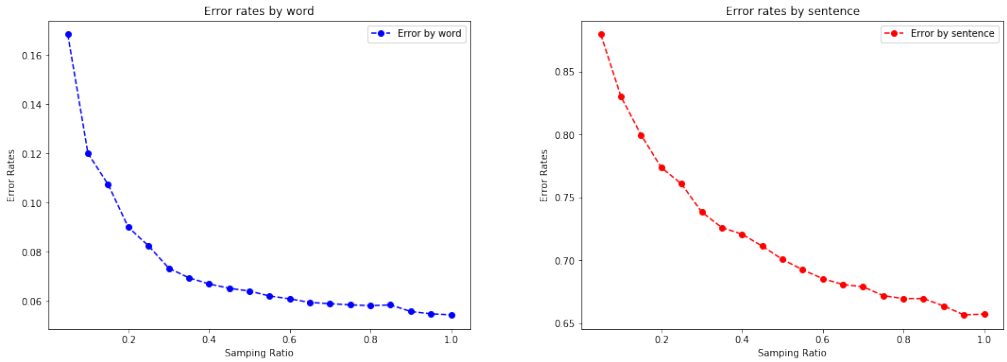

Figure 2: Error rate by sampling ratio

1.2 Analysis

In the graph, we can see the significant relationship between sample number and error rate. Even the slope become smaller, the error rate decrease when the sample size is bigger. To train emission probability and n-gram probability, the larger data help to develop a better model.

The relationship between train data and error ratio would be similar with the other languages. However, each language has different size of word / corpus space. It means that the minimum number of sample data can be differed by the language.

2 Task 2

I approached the task 2 with two different approaches. First, I develop own tri-gram model using HMM. Second I tried to improve the bigram / trigram viterbi model. To deal with the unsean data, I use the linear interpolation and good-turing estimator.

2.1 Trigram model

I developed trigram viterbi model. In the HMM process, I also consider the two consecutive tags before current tag. After that, I count the number of occurrence and calculate the MLE probability as HMM model. The biggest problem is the sparsity of data set. Comparing to unigram or bigram, the number of events increase exponentially.

In the dataset, there are 47 different tags including INITIAL_STATE ('init') and FINAL_STATE ('final). There are 2209(=47*47) possible states in bigram and 103823(=47*47*47) possible states in trigram. In reality, we can remove totally impossible stats to reduce the computation time.

> Trigram probabilities generated from a corpus usually cannot directly be used because of the sparsedata problem. This means that there are not enough instances

for each trigram to reliably estimate the probability. Furthermore, setting a probability to zero because the corresponding trigram never occured in the corpus has an undesired effect. It causes the probability of a complete sequence to be set to zero if its use is necessary for a new text sequence, thus makes it impossible to rank different sequences containing a zero probability. [2]

2.2 Imputation and Linear Interpolation

The estimates of trigram, bigram and unigram differ in their advantages and disadvantages. Unit estimates cannot be equal to zero for problems with numbers or denominators. Estimates are therefore always well defined and will always be greater than zero (if each word can be seen at least once in the training corpus, this is a reasonable assumption). However, the unigram estimate completely ignores context and thus deletes very valuable information. In contrast, trigram estimates use context, but many of the calculations have the problem of zero. The bigram estimates are between these two extremes.

We don't have the enough information for the unseen data. So, simply I impute the unseen cases with small value (1e-10). This assume that unseen event is rare compare to the seen event. So, the model can use more data regarding this.

$$\text{Trigrams} \quad \hat{P}(t_i|t_{i-1},t_{i-2}) = \frac{C(t_{i-2},t_{i-1},t_i)}{C(t_{i-2},t_{i-1})} \tag{8.26}$$

$$\text{Bigrams} \quad \hat{P}(t_i|t_{i-1}) = \frac{C(t_{i-1},t_i)}{C(t_{i-1})} \tag{8.27}$$

$$\text{Unigrams} \quad \hat{P}(t_i) = \frac{C(t_i)}{N} \tag{8.28}$$

The standard way to combine these three estimators to estimate the trigram probability $P(t_i|t_{i-1},t_{i-2})$ is via linear interpolation. We estimate the probability $P(t_i|t_{i-1}t_{i-2})$ by a weighted sum of the unigram, bigram, and trigram probabilities:

$$P(t_i|t_{i-1}t_{i-2}) = \lambda_3\hat{P}(t_i|t_{i-1}t_{i-2}) + \lambda_2\hat{P}(t_i|t_{i-1}) + \lambda_1\hat{P}(t_i) \tag{8.29}$$

We require $\lambda_1 + \lambda_2 + \lambda_3 = 1$, ensuring that the resulting P is a probability distribution. The λs are set by **deleted interpolation** (Jelinek and Mercer, 1980): we successively delete each trigram from the training corpus and choose the λs so as to maximize the likelihood of the rest of the corpus. The deletion helps to set the λs in such a way as to generalize to unseen data and not overfit. Figure 8.10 gives a deleted interpolation algorithm for tag trigrams.

Figure 3: Linear Interpolation Approach

---

set $\lambda_1 = \lambda_2 = \lambda_3 = 0$
foreach trigram t_1, t_2, t_3 with $f(t_1, t_2, t_3) > 0$
    depending on the maximum of the following three values:
        case $\frac{f(t_1,t_2,t_3)-1}{f(t_1,t_2)-1}$: increment λ_3 by $f(t_1, t_2, t_3)$
        case $\frac{f(t_2,t_3)-1}{f(t_2)-1}$: increment λ_2 by $f(t_1, t_2, t_3)$
        case $\frac{f(t_3)-1}{N-1}$: increment λ_1 by $f(t_1, t_2, t_3)$
    end
end
normalize $\lambda_1, \lambda_2, \lambda_3$

Figure 1: Algorithm for calculting the weights for context-independent linear interpolation $\lambda_1, \lambda_2, \lambda_3$ when the n-gram frequencies are known. N is the size of the corpus. If the denominator in one of the expressions is 0, we define the result of that expression to be 0.

Figure 4: Deleted Interpolation Algorithm

2.3 Good Turing Estimator

Good Turing estimation is more gentle way of dealing with unseen data. This approach calculate the probability of event using the relationship between counts of event and its probability. One of the problem is that it cannot deal with the number of event count is missing. To deal with this issue, I use smoothing function for the number of event count.

Good-Turing method

Intuition Use the counts of things you have seen *once* to estimate the count of things not seen.
i.e. use n-grams with frequency 1 to re-estimate the frequency of zero counts.

Suppose we have data with total count of events being N:

Standard notation:

r	=	frequency of event e
n_r	=	number of events e with frequency r $n_r = \|\{e \in E \| Count(e) = r\}\|$
N_r	=	total frequency of events occuring exactly r times $N_r = r \times n_r$

Observation: $N = \sum_{r=1}^{\infty} N_r$ $\qquad$ $N_0 = 0$

What we want: To recalculate the frequency r of an event (r^*)

Figure 5: Good Turing method

Good-Turing Estimates

The Good-Turing probability estimate for events with frequency r:

$$P_{GT}(\alpha) = \frac{r+1}{N}\frac{E[n_{r+1}]}{E[n_r]} \approx \frac{r+1}{N}\frac{n_{r+1}}{n_r} = \frac{1}{N} \times (r+1) \times \frac{n_{r+1}}{n_r}$$

We can think of this, as assigning frequency of r^* to events appearing r times:

$$r^* = (r+1) \times \frac{n_{r+1}}{n_r}$$

n_r : number of events with freq. r
n_{r+1} : number of events with freq. $r+1$

Figure 6: Good Turing estimate

Good-Turing methods provide a simple estimate of the total probability of the objects not seen. How this probability is divided among the objects depends on the objects, especially on any structure (such as the component words of a bigram) the objects may have.
Simple Good-Turing methods uses the simplest possible smooth of the frequency of frequencies, N_r, a straight line, together with a rule for switching from Turing estimates which are more accurate at low frequencies. [3]

We denote r is the number of count and N_r is the event that is observed r times. The value of N_r from data is also sparse. In order to deal with this issue, Simple Good Turing is suggested which use function $S : S(\log(N_r)) = a + b * \log(r)$. This function is fitted with the given data by logistic regression such that $b < -1$ (to guarantee that $r^* < r$. [2]

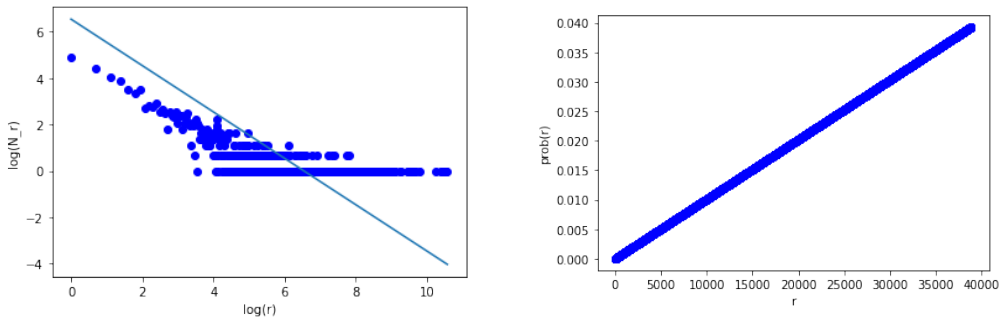

Figure 7: Simple Good Turing Fitting

3 Results

In the result, I found that the smoothing works well if we have a lot of unseen data. In the bi-gram model, smoothing approach improved the performance slightly. In the case of tri-gram model, accuracy was decreased, because the number of corpus is not enough or my model does not utilize it all.

		HMM	HMM +interpolation	HMM +Simple GT
Bigram	error rate by word	0.054091782	0.053643094	0.052945135
	error rate by sentence	0.655882353	0.655882353	0.651176471
Trigram	error rate by word	0.060996585	0.060797168	0.059775158
	error rate by sentence	0.724705882	0.715294118	0.712941176

Table 1: Test Results

Here we can see the best result is bigram with Simple Good Turing estimator. As I understand, smoothing algorithm is well implemented, but trigram model should be improved.

3.1 Additional Measure

POS tagging also can be evaluated by different measure such as precision, recall, F1- score and so on. I summarize the data as multi-class classification report. In the perspective of context, perplexity can be a important measure to compare the results. Here I compare the summary of baseline and best results.

By comparing two table, best model shows better results in the overall statistics. The tags distribute with very different numbers, the weight average can be potential measure to compare the result. We can see the best model shows better performance in weighted average precision, weighted average recall and weighted average F1-score.

[2]Implementation reference link for Simple Good-Turing estimation

	Precision	Recall	F1-score	Support
accuracy	-	-	0.9459	40117
macro avg	0.8151	0.8477	0.8106	40117
weighted avg	0.9508	0.9459	0.9479	40117

Table 2: Results Summary of baseline

	Precision	Recall	F1-score	Support
accuracy	-	-	0.9471	40117
macro avg	0.8674	0.8802	0.8478	40117
weighted avg	0.9508	0.9471	0.9484	40117

Table 3: Results Summary of best result

References

[1] Dan Jurafsky. *Speech & language processing*. Pearson Education India, 2000.

[2] Thorsten Brants. Tnt: a statistical part-of-speech tagger. In *Proceedings of the sixth conference on Applied natural language processing*, pages 224–231. Association for Computational Linguistics, 2000.

[3] William A Gale and Geoffrey Sampson. Good-turing frequency estimation without tears. *Journal of quantitative linguistics*, 2(3):217–237, 1995.