Natural Language Processing
Fall 2018
Professor James Marting
Assignment 2 (POS tagging) Report

Student: **Keval D. Shah**

1. Dividing main training data:-

- To keep a track of the progress with accuracy and analyze the effect of an implementation, I have divided the main set into two sections in the ratio 80:20, calling them training set and dev set respectively. I shall train on the main set when building the model for the test set.

2. Implementing the baseline approach:-

- To implement baseline, I have maintained a dictionary of word-tag pair corresponding to the frequency of occurrence. While validating, a word is assigned the tag for which it had the highest frequency in the word-tag dictionary.
- In the case of encountering a new word in the dev set, I would simply assign it the tag that occurred the most number of times.
- The accuracy for baseline was found to be **86%**. If I further filter out the repeating sentences (with same tags assigned to each word), This accuracy falls to **84%**.
- This accuracy is surprisingly high because of the nature of the dataset and the language. The major drawback in this is assigning tags to unseen words and words that occur with a new tag. Hence, we need to use:
  - Viterbi-bigram: for relating previous-current word states.
  - Use of <unk>: to decide the right tag for unseen words.
  - smoothing: to give weightage to unseen words and seen words with unseen tags.

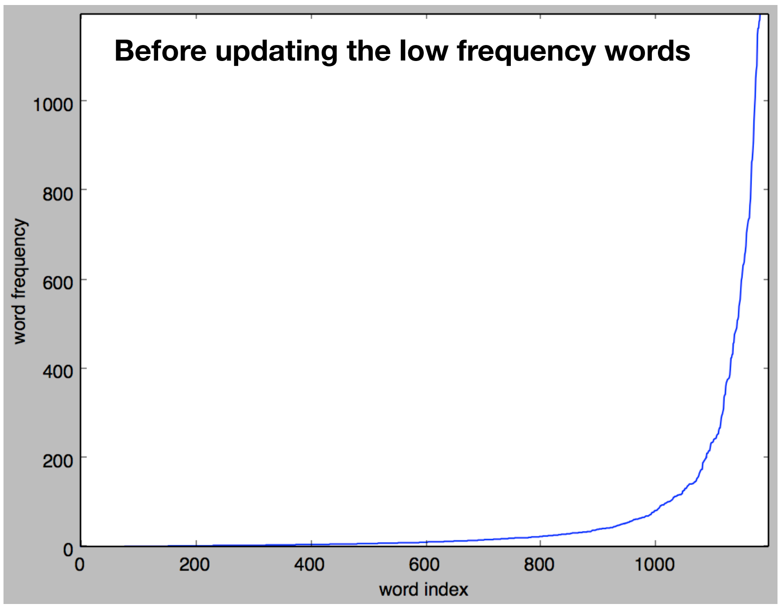
3. Implementing a tagger with bigrams-Viterbi, “Add-1” smoothing and <unk> placeholder:-

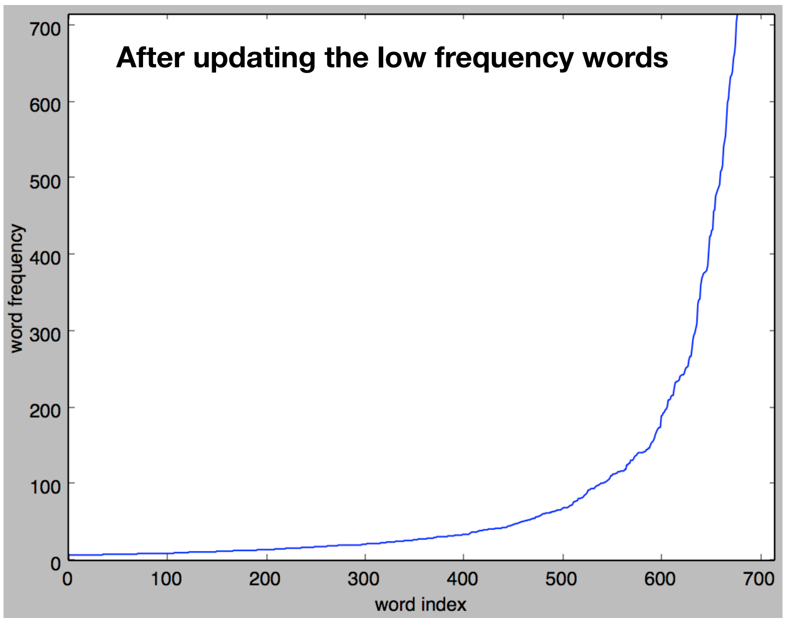
- Firstly, we have to add an <unk> word to the unique word list and set counts equal to zero for all tags.
- Next step was adding a <s> tag to the list of all tags. The <s> tag will aid in deciding the tag of a given word if it appears at the start of the sentence.
- We create a word likelihood probability matrix W2T2 with dimension (36 x #unique_words). Next, create a state transition probability matrix T2T1 (i.e. tag1 x tag2) with dimension (37 x 36) since <s> would not appear in the tag2 position.
- In the next step we calculate the probability with ‘Add-1’ smoothing. All the required probabilities are ready for implementing the Viterbi algorithm. Note that the Viterbi algorithm acts on one sentence at a time and my implementation if self-written and non-recursive.
- I create a Viterbi matrix and a Backtrack matrix of dimension (36 x sentence_length). First, we iterate through each element of this matrix and calculate the product of W2T2 and T2T1 probability. In another iteration, we iterate through each element (starting from except first column) and find the index of the previous column element that assigns the maximum probability till that path. We store this index in the corresponding backtrack matrix. (These operations can be done in a single loop as well)
- In the final iterations, we check the maximum probability in the last column of the Viterbi matrix and get the corresponding index from backtrack matrix. Now, trace this index till the very start and we have our final sequence of tags for this sentence.
- The accuracy for this implementation was found to be **90%** on the dev set.

4. Extending the previous model for better performance:- *(reference: Textbook-3.3.1)*

- Create a copy of the wordlist that maps the words to their corresponding tag counts. In this copy, replace the words to <unk> that have total occurrence count less than a predefined threshold. This threshold is a small number between the range 5-15.

- By doing this, we are actually reducing the long tail of the word-count array and tackling the assignment of new words in a way similar to those of rare words. The graphs below compare the tail length after applying this update.
- The accuracy with this extension of <unk> tag was found to be **93%** on the dev set.


Before updating the low frequency words
word frequency
word index


After updating the low frequency words
word frequency
word index