Natural Language Processing - CS6370
Spell Check Assignment Report

Aravind Sankar CS11B033
Sriram V CS11B058

September 26, 2014

1 Introduction

This assignment involved designing an efficient spell checker. This spell checking assignment was divided into three parts :

- Word checker where spelling corrections are given for a misspelled word not present in the dictionary.
- Phrase checker where spelling corrections are given for misspelled word(s) present in phrases.
- Sentence checker where spelling corrections are given misspelled word(s) present in sentences.

  The Spell Checker was implemented in *Python*.

2 Corpora used :

- The Unix dictionary for american english, which has close to 73,000 valid english words was used as the dictionary to identify if a given word has a spelling error.
- Corpus of Contemporary American English (COCA) dataset (from here , which has 1,000,000 most frequent n-grams. This was used in the phrase and sentence spell checkers.
- Natural Language Corpus Data: Beautiful Data (from here) was used to obtain word and letter frequencies.
- The brown corpus was also initially used for learning context words, but isn't part of the final spell checker model.

3 Word Checker

The basic idea of approach for Word Checker was obtained from the paper by Kernighan et.al which was titled - A Spelling Correction Program Based on a Noisy Channel Model. This paper doesn't take the context in which a misspelt word appears into account, and provides spelling corrections for standalone words.
Given a misspelt word t, we first find out a set of candidate corrections, and then find the probability of a correction c given t using the Bayesian model as,

$$Pr(c|t) = Pr(c)Pr(t|c)$$

where $Pr(c)$ represents the prior probability of the correction word c estimated from a corpus , and $Pr(t|c)$ is the likelihood term which accounts for spelling transformations on the letters of the word. The spelling transformations include letter insertions, deletions, substitutions and transpositions.
$Pr(t|c)$ is estimated using the confusion matrices given in the paper. We used the 4 confusion matrices $\texttt{del}[X, Y]$, $\texttt{add}[X, Y]$, $\texttt{sub}[X, Y]$ *and* $\texttt{rev}[X, Y]$ after applying Laplace smoothing (Add-One). The $\texttt{chars}[X, Y]$ matrix (not given in the paper) was estimated using letter bigram counts obtained from here. This was appropriately scaled down using the ratio of sizes of the corpora.
The paper only considers correction words c within edit distance (Damerau–Levenshtein distance DL) 1 of the misspelt word t. We extend this and consider correction words upto edit distance 3, beyond which we felt there would hardly be any corrections possible.
When we consider 3 edits in a word, we assume that the likelihood of occurrences of each of them are independent (Naive Bayes assumption), and hence we take the product of likelihood of each of the 3 edits.
Instead of searching the entire set of words in the dictionary to find out words at edit distance < 3, we applied a few techniques for candidate selection and pruning the search space :

- We create an inverted index of bigrams to set of words that contain the bigram. The set of words that contain the bigram, are also indexed by the length of the word.

- At query time, all possible bigrams of the misspelt word are found and we use the index structure to find all words that contain any of the bigrams, and also within length of $\pm$ 3 from the misspelt word.
- The set of candidate words obtained in the previous step are ranked according to the Jaccard Similarity index, and top-200 words are chosen for next stage.
- In this stage, we need to compute the likelihood scores for the candidates corrections obtained in the previous step. The dynamic programming solution for computing the DL distance has $O(n^2)$ complexity.
- We use the `trie` data structure to store these candidate words, and efficiently search the tree and return words within a given edit distance threshold (3 here). This gives a great reduction in the time taken in comparison to the brute force approach of computing edit distance for each of the words.
- Storing the words in a trie helps to prune out subtrees based on the threshold edit distance. We also compute the likelihood of each correction in the process. (Can explain something here).

Another approach using a BK tree was also tried to compute edit distance efficiently, but we found it to be inferior in comparison to our earlier approach, in terms of performance.
try to put some more fart :P.
Word checker MRR results : TODO

4 Phrase and Sentence Checker