Natural Language Processing - CS6370
Spell Check Assignment Report

Aravind Sankar CS11B033
Sriram V CS11B058

September 28, 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 `del`$[X, Y]$, `add`$[X, Y]$, `sub`$[X, Y]$ *and* `rev`$[X, Y]$ after applying Laplace smoothing (Add-One). The `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.
- We computed the Jaccard's similarity for strings x and y by constructing the sets X and Y as the set of bigrams (letter) in strings x and y respectively. The jaccard similarity is computed as :

$$Sim(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

- 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

We tried 2 different approaches for the phrase/sentence spell checking problem and finally chose one of them.

4.1 Context words

The first approach is based on the idea of context sensitive spelling correction. We followed the paper - A Bayesian hybrid method for context-sensitive spelling correction by Andrew Golding. For the target word, they identify a confusion set, which is a set of possible words that could replace that word in a sentence. Since the paper deals with correcting spelling errors that result in valid words in the sentence, they assume that they have predefined confusion sets. In our case, since we're correcting misspelt words only, we use the top-20 candidates returned from our word checker as our confusion set. Let w_i represent each word in the confusion set. The idea is that, each word w_i in the confusion set will have a characteristic distribution of words that occur in its context. Thus, to classify an ambiguous target word, we look at the words that occur around it and see which w_i's distribution they closely follow. For this purpose, we look at the context words that occur in a window of $\pm k$ window of the target word.

$$P(w_i|c_{-k},...,c_1,...,c_k) = P(w_i)P(c_{-k},...,c_1,...,c_k)$$

$$P(w_i|c_{-k},...,c_1,...,c_k) = \prod_{j \in -k,...,1,...,k} p(c_j|w_i)$$

We assume that the occurrence of each context word is independent of every other context word.
The values of $P(c_j|w_i)$ were estimated using the brown corpus.
When we tested it for the sample test cases, we didn't get very good results as the context words in phrases were too general to uniquely identify words in the confusion set based on frequency in the corpus.

4.2 N-gram models

Approach 1

The next approach we tried was language models, more specifically n-gram models. A language model assigns a probability to a sentence, represented as a sequence of words. Suppose a sentence is represented as $W = w_1 w_2 w_3 ... w_n$, then we calculate $P(w)$ using the chair rule of probability as :

$$P(W) = P(w_1 w_2 w_3 ... w_n) = \prod_i P(w_i | w_1 w_2 ... w_{i-1})$$

As estimating each of the joint probabilities from the corpus won't be feasible due to data sparsity, we looked at n-gram models, where we approximate the product as (for a k-gram model),

$$P(W) \approx \prod_i P(w_i | w_{i-k} ... w_{i-1})$$

We use the trigram and bigram frequency counts obtained from the COCA dataset. As in the previous case, we obtain our confusion set from the output of our word checker. By substituting each word in the confusion set in place of the misspelt word in the phrase, we get a set of candidate phrases (or sentences). Using the trigram model, we find out the probability of each sentence, and rank them using these as scores. This approach used the general idea of language models to do spelling correction for phrases, but it was found the following approach gave much better results for spelling correction.

Approach 2

The idea behind our second approach based on language models was derived from the paper - Web-Scale N-gram Models for Lexical Disambiguation by Bergsma et.al. Out of the many techniques that paper describes, we adapt the **SUMLM** algorithm. We consider ngrams of sizes 5,4,3 and 2. Again, just like in the previous approach , we get a list of candidate sentences using the confusion set obtained from word checker.

Assuming the sentence has only one error, for a given sentence w, we find out possible n-grams which contain the target word, $\forall n \in \{2, 3, 4, 5\}$. This gives rise to a maximum of 14 ngrams, which contain the target word. We find the ngram counts of each of these grams, and take the *log* sum of each of these counts to get the score for that particular sentence.

Though this score gave correct phrases in the top-K results, it didn't rank them in the correct order. This was because we didn't take the likelihood factor (computed in word checker) into account. Without using likelihood to rank the scores, we had a few correction words that weren't so close to the misspelt word in terms of edit-distance and possible transformations (weren't that likely), but figured in the top of our list because they had larger ngram counts. To avoid this problem, we weighted our final score for each of the candidate words based on both the ngram model score and the likelihood score. This gave us the better results in comparison to the more traditional ngram model, as we include all possible ngram counts and hence cover even long range dependencies as much as possible.

This solved the case of one misspelt word per sentence/phrase.

When we had more than one misspelt word, we followed a 2 step procedure :

- First, we consider each misspelt word independent of the other one, and find a set of top-K (5 generally) corrections using the above procedure. This method has an obvious limitation that, it ignores ngrams in which the other misspelt word(s) also occurs, as such an ngrams which contain the other misspelt word.
- We take the top-K corrections for each of the misspelt words, and construction combinations or tuples. We then recompute the ngram model scores for each of these sentences, and rank these sentences to give our final set.