INDIAN INSTITUTE OF TECHNOLOGY MADRAS

CS6370 – Natural Language Processing
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*.

The source code present in `/src` contains the following files and directories:

- `word_check.py` – Performs spelling corrections on words.
- `phrase_check.py` – Performs spelling corrections on phrases and sentences. Uses the above file as a library.
- `metaphone.py` – Contains the method to compute the double metaphone of a word. Imported into `word_check.py`
- `data` – This folder contains data that is used to build indices as a preprocessing step.

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 [1] 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 (`del[X,Y]`, `add[X,Y]`, `sub[X,Y]` and `rev[X,Y]`) given in [1], which can be found in the `data` directory in the source folder as well. However, we found that the provided matrices were too sparse to be of much use at edit distances greater than 1, and hence we augmented the ones provided in the paper by applying Laplace Smoothin (Add-One), to result in the 4 confusion matrices `addoneDelXY`, `addoneAddXY`, `addoneSubXY` and `addoneRevXY`. 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 to produce the final matrix `newCharsXY`. Further, the sum of the individual rows too was precomputed and saved to a fifth matrix `sumnewCharsXY`, in the interest of time.

The paper only considers correction words c within an edit distance (the Damerau-Levenshtein distance, DL) of 1 from 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.

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 :

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

- Here, we observed that when jaccard's similarity was used, it fails to rank candidates properly when the misspelt word is small in length. This is because, if we have one spelling error, then 2 bigrams are lost, which leads a loss in rank and longer words get better tanks.
- Hence we modified the similarity as :

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

  where Y is the set of bigrams of the correction (or candidate) word. This also gives more importance to the corrected word, unlike the Jaccard Similarity measure which is symmetric.
- 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 a complexity of $O(mn)$, where the length of the source and target words are m and n.
- 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).
- The `trie` has a node for each letter of a word, and joins together nodes for words that share the same prefix. The dynamic programming solution to compute edit distance requires us to calculate the matrix row by row, for each letter in the source word. Thus, the `trie` gives us the advantage of having to compute each row only once, for all words that require this row.
- This reduces the complexity of the computation to $O(m$<*no. of nodes*>$)$, which is much, much lesser than the complexity of the brute force approach of computing edit distance for each of the words ($O(nm^2)$, where m is the length of the longest word in the dictionary, and n is the number of words in teh candidate set).
- Storing the words in a `trie` helps prune out subtrees by using the edit distance as a threshold. The task of having to compute the likelihoods of each correction is also performed simultaneously, to speed up computation and reduce complexity.
- In addition to using the prior probablity and likelihood to rank the words, we also calculate a phonetic score for each of the candidate words, which we multiply with the existing score.
- The phonetic score is computed by evaluating the double metaphones of the misspelt word and each of the candidates, and assigning an appropriate score depending on whether they had a primary-primary match, primary-secondary match or no match.

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. Our distance function too inhibited our use of the

BK tree, because we switched to a DL distance function that considers adjacent transpositions only, which is a non-metric function, whereas the BK tree relies on a metric function to prune efficiently.

3.1 LIKELIHOOD COMPUTATION

It turns out that unlike for the simple case of an edit distance of 1 as handled in [1], likelihood computation is a trivial task. In the case of greater edit distances, the problem becomes more involved since we need to break down the edits into a series of single edits, as we have confusion matrices for single edits only. Further, the order of these edits also comes into the picture. There is also the issue of multiple edit paths to convert a correct word into the misspelt word.

Hence, we could look at the problem in two ways – a forward and a backward approach. The backward approach essentially involves traversing every edit path in reverse, multiplying single edit Naive Bayes probabilities along the way, as mentioned in the paper. Hence, this can be looked at as an error graph traversal of sorts and can be computed using a Depth-First Search (DFS) procedure. For this, at every node, we need to keep track of its parent nodes, and the edge labels (whether we arrived at that node via an insert, delete, subsitution or transposition), and accordingly move to the parent node. We initially implemented this, but the overhead involved made us switch over to the online, forward approach, which could be modelled as a dynamic programming problem similar to the edit distance one, and hence could be computed simultaneously along with the edit distance.

At an i, j in the edit distance matrix, let the correct word seen till now be s, and the misspelt word be tp, where t is a string and p is the letter to be inserted. Then,

$$Pr(tp|s) = Pr(tp|sp)Pr(sp|s) + Pr(tp|t)Pr(t|s)$$

Here, the first term in the first product and the second one in the second product are single edits and hence, are computed at this step. $Pr(tp|sp) = Pr(t|s)$, and this value has been previously computed as part of the DP solution. Similarly, for deletes it is:

$$Pr(t|sp) = Pr(t|s)Pr(s|sp) + Pr(t|tp)Pr(tp|s)$$

The equation considered for substitutions works out to:

$$Pr(td|sp) = Pr(td|sd)Pr(sd|sp) + Pr(td|tp)Pr(tp|sp)$$

And for transpositions the equation considered is:

$$Pr(tjn|snj) = Pr(tjn|sjn)Pr(sjn|snj) + Pr(tjn|tnj)Pr(tnj|snj)$$

Here too, we see that the first term in the first product and the second one in the second product are both equal to $Pr(t|s)$ (which has already been computed), and the remaining products are all one edit distance away, and hence can be computed easily.

Thus, this algorithm was run simultaneously with the edit distance DP solution, to speed up the computation process. All the accessed entries were indexed to reduce the complexity of a single iteration to $O(1)$.

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, as is described in [2]. 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 [3]. 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.

5 RESULTS

5.1 WORD CHECKER

For the words given in `words.tsv`, our spell checker obtains an **MMR of 1** for all the words except *failr* and *thruout*. In the case of the former, our model gives *failure* as its first suggestion, which we believe is a better suggestion than the expected word *fairy*, as phonetics is being taken into account. In the latter case, we obtain an **MMR of 0.5**, with *throat* being the primary suggestion mainly because of the significant difference in edit distances (3 vs 1 respectively). The word *throughout* gets the second position, once again due to the influence of the phonetic score that has been computed using the double metaphones of the misspelt word and its suggestion.

5.2 PHRASE CHECKER

For the phrases specified in `phrases.tsv`, our spell checker once again obtains an **MMR of 1** for all the phrases specified, except for the cases where the errors are due to string concatenation (for example, *halloffame*) or are grammatical in nature (such as the insertion of 'the' in *president of united states*). The first issue can be tackled by making use of a word-splitting algorithm to test every word not present in the dictionary, before performing the actual spell check. The second case deals with grammar and syntax, and could thus be tackled by suggesting insertions based on either N-gram models or context windows.

5.3 SENTENCE CHECKER

Finally, for the sentences mentioned in `sentences.tsv`, our model yet again gives an **MMR of 1** in all cases, if we choose to disregard the insertion of 'not' before 'to' in the misspelled sentence, *To be or to bea is not the question*. This issue of insertion can be handled as mentioned above, for the case of phrases.

REFERENCES

[1] Kernighan, M.D., Church, K.W., Gale, W.A.: A spelling correction program based on a noisy channel model. In: Proceedings of the 13th Conference on Computational Linguistics - Volume 2. pp. 205-210. COLING '90, Association for Computational Linguistics, Stroudsburg, PA, USA (1990)

[2] Golding, A.R., Golding, A.I.: A bayesian hybrid method for context-sensitive spelling correction. In: In Proceedings of the Third Workshop on Very Large Corpora. pp. 39-53 (1995)

[3] Bergsma, S., Lin, D., Goebel, R.: Web-scale N-gram models for lexical disambiguation. In: Proceedings for the 21st International Joint Conference on Artificial Intelligence. pp. 1507-1512 (2009)