CS6370: Natural Language Processing

Spell Check Assignment Report

Indian Institute of Technology Madras

Chinni Chaitanya, EE13B072
ee13b072@smail.iitm.ac.in

Venkatesh Maligireddy, EE13B041
ee13b041@smail.iitm.ac.in

Swaroop Kotni, EE13B030
ee13b030@smail.iitm.ac.in

Pronnoy Noel, EE13B029
ee13b029@smail.iitm.ac.in

October 5, 2016

Contents

1 Introduction

This assignment attempts to implement *spell checker and correction* programs for text, which will correct the erroneous word[1] present in it. The assignment is categorized into the following three parts,

- **Word checker**, a program which checks and corrects standalone erroneous words[2].
- **Phrase checker**, a program which examines phrases and corrects the erroneous words. We assume *phrases* contain about 5 words.
- **Sentence checker**, a program which examines sentences and corrects the erroneous words. We assume *sentences* contain about 30 words.

In all the three cases, we assume that there is only **one** erroneous word.

1.1 Directory structure

The submission contains the following folder structure,

```
/src
....bayesian.py
....editex.py
....stringTokenize.py
....trie.py
....phrase_check.py
....word_check.py
README
Report_Team10.pdf
```

2 Data generation

2.1 Corpa used for data generation

We have used *brown corpus* and *american-english* dictionary available in Linux and generated all the required data (confusion sets, context words frequencies etc.,)

3 Word checker

The *word checker* aims to implement a program to inspect the standalone erroneous word given as input, and come up with a set of possible candidates with their probability of correctness and ranking. The probability of each word is

[1] In phrases and sentences, the erroneous word need not be misspelled but might be contextually incorrect.

[2] The word given might be correct word also.

calculated both ***statistically (Bayesian probability)*** [1] as well as ***phonetically (editex)*** [2]. All the main ideas regarding ranking and probabilities are taken from [1] and [2].
This whole implementation can be broken into the following four sub-tasks,

- **Task-1** Generation of candidate set for the given erroneous word
- **Task-2** Calculate the Bayesian probability for each word in the candidate set
- **Task-3** Calculate the phonetic probability for each word in the candidate set
- **Task-4** Combine both Bayesian and phonetic probabilities and rank the words in the candidate set

3.1 Generation of candidates

The aim of this task is to generate the set of candidate words or the candidate set for the erroneous word given. A crude way to do this is to generate all possible strings for the given erroneous word by adding, deleting, substituting and reversing the letters in the given word and check whether that string exists in the dictionary. This works, but is computationally intensive. Hence, we need to intelligently store the words in the dictionary such that, the search doesn't take much time. Instead of the additional step to generate the possible strings and check if they exist in the dictionary, we could intelligently fetch the words from the dictionary itself by some measure. One such measure is *Levenshtein distance*. The following sub-sections attempt to address these ideas.

3.1.1 Levenshtein distance

Levenshtein Distance between two strings is given by the minimum number of edits, substitution, insertion, deletion required to change one string to another. We have implemented the *dynamic programming* approach to calculate the Levenshtein distance between the candidates and the incorrect word.

- Levenshtein distance follows the triangular inequality. That is,
  $d(x, y) + d(y, z) \geq d(x, z)$.
- It is commutative. That is,
  $d(x, y) = d(y, x)$.
- All the values in the last row are greater than the Levenshtein distance except the last value (used for searching in *Trie*).

3.1.2 Burkhard-Keller Trees or BK-Trees

BK-Tree is a tree data structure used for spell checking based on the Levenshtein distance between two words. All the words in the dictionary are iterated and inserted one by one into the tree. The insertion goes as follows,
The first word of dictionary becomes the root word of tree with an associated

dictionary to it. Then to insert the next word, the Levenshtein distance is calculated between root and current word and the current word is inserted at the *key* equal to the levenshtein distance in the root dictionary. If that key already exists, the same process is repeated for the word at that Levenshtein distance. The triangular inequality of the levenshtein distance is used while searching in the bk-tree to get the candidate words.

3.1.3 Trie structure

This is also a tree data structure. The root word is null. Each node is associated with a children dictionary. Now to insert the word, the first letter of the word is checked if present in the children of the root node. If not present, it is created. If present, it will go to the next character in the word and repeats the same with the children dictionary of the correct letter.
The property of the levenshtein distance that the last row of the matrix have all values except the last value greater than the levenshtein distance is used to search in the trie to get the candidates.

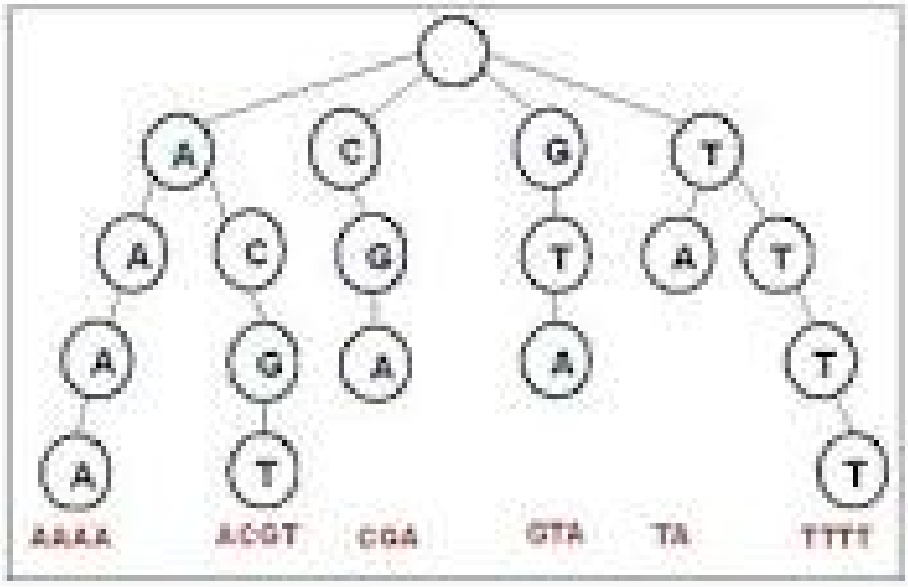

Figure 1: Trie data structure sample

3.2 Ranking the candidates statistically

3.2.1 Bayesian probability

Given the typo t, the possible candidates (c_i) were generated from the Trie data structure which was generated from the dictionary. Now the candidates are ranked based on posterior probability score $P(c|t)$, where c is the candidate word Using the Bayesian rule the probability that the suggested correct word

c, produces the typo c is given by,

$$P(c|t) = P(t|c) \times \frac{P(c)}{P(t)}$$

Each candidate correction, is scored by $Pr(c) \times Pr(t|c)$, and then normalized by the sum of the scores for all proposed candidates. The probability $P(t)$ cancels out during normalization.
The frequency of the correct word, $P(c)$ is obtained form the word frequency list which are generated by files from,
`http://opus.lingfil.uu.se/OpenSubtitles2016.php`
For smoothing purposes the frequency of c is modified to,

$$P'(c) = \frac{N \times P(c) + 0.5}{N}$$

where N is the total number of words in the corpus.

3.2.2 Computing $P(t|c)$

The conditional probabilities $P(t|c)$, are computed from the four confusion matrices namely deletion, insertion, substitution and reversal,

$$P(t|c) = \begin{cases} \frac{del[c_{p-1}, c_p]}{chars[c_{p-1}, c_p]}, \ if\ deletion \\ \frac{add[c_{p-1}, t_p]}{chars[c_{p-1}]}, \ if\ insertion \\ \frac{sub[t_p, c_p]}{chars[c_p]}, \ if\ substitution \\ \frac{rev[c_p, c_{p+1}]}{chars[c_p, c_{p+1}]}, \ if\ reversal \end{cases}$$

3.2.3 Generating single occurrences

We generate the single occurrences by parsing every word in the corpus and counting the number of times a character has occurred in the corpus.

3.2.4 Generating co-occurrences

Similarly for character co-occurrences we parse every word of the corpus and search for a pattern of consecutive characters (say x, y). For example consider ae, we search in the whole corpus for occurrence of the pattern ae and the number of occurrences of ae gives the co-occurrences of ae. Similarly we find the pattern for all the possible co-occurrences.
The confusion matrices are pre-computed from the training corpus 1988 AP corpus.

From the list of candidates we receive from the Trie data structure, we use Levenshtein edit distance to identify the deletions, substitutions etc for each candidate. If multiple edits are present, we simply multiply all the probabilities to obtain $P(t|c)$.

We then compute $P(c|t) \times P(c)$ for all the candidates and normalize it. This gives the probability that each of the candidate word is actually the correct word for a given typo t.
Based on the probability score the candidate words are ranked.

3.3 Ranking the candidates phonetically

The aim of the phonetic ranking is measure how similar the two given words sound, when pronounced. There are many methods which attempt to calculate the phonetic similarity. For example, *Soundex, Phonix, Q-gram method, Agrep, Idapist* and *Editex*. Based on the results of performance of each method and the simplicity of implementation as per [2], we chose to implement the ***Editex*** method for the phonetic ranking.

3.3.1 Editex

Editex is the phonetic distance measure of two words, similar to the *Levenshtein distance*. **Lesser the phonetic distance, greater is the phonetic similarity between two words**. This method is an improved implementation of *Soundex* and *Phonix*. All the letters in the English literature are classified into groups[3]. Each group contains letters, which generally sound similar in words when pronounced. *Editex* takes into account the silent letters like `'h'` and `'w'` also unlike Soundex or Phonix. By comparing the letters in the candidate word and erroneous word, the phonetic distance is computed, which is similar to the Levenshtein distance. This computation is implemented using the concept of *dynamic programming*.

3.4 Combined ranking of candidates using both statistical and phonetic ranking

The probabilities obtained from both the Bayesian and Editex are used to compute the final probability for the candidate. This operation is a function of the probabilities of both Bayesian and Editex.
We must give importance to both the probabilities since we should prefer the candidate which is statistically very frequently occurred (measured with Bayesian probability) and also sounds similar (phonetic probability) to the erroneous word.
If a candidate has very high phonetic probability but low Bayesian probability (or vice-vesra), and another candidate which has average phonetic as well as Bayesian probabilities, intuitively, we prefer the second candidate. Hence, to account for the best candidate, **we computed the final probability of each candidate as the product of it's Bayesian and phonetic probabilities**. The final ranking of candidates is done in the descending order of their corresponding final probabilities.

[3]Each letter can be in multiple groups.

4 Phrase and Sentence checker

The *context checker* attempts to detect and suggest a correction to an erroneous word present in a phrase or sentence. Here, we cannot assume that the erroneous word is misspelled. It might be contextually incorrect like for example, the word **peas* in the sentence, *"He was awarded the Nobel peas prize"*. The word *peas* is not misspelled. But it is contextually incorrect. So the aim of this *context checker* is to detect the erroneous word *peas*, and suggest a correction, *peace*.
Also, the sentence might contain correct words which are combined for example, consider the sentence, *"Standing in the halloffame"*. Here **halloffame* is the erroneous word but is combined by correct words. Our context checker must detect it and suggest a correction *hall of fame*.
The suitable candidate is predicted using the *Bayesian hybrid* method and using *collocations* as described in the paper [3].

4.1 Text tokenization

This attempts to tokenize the input sentence and detect the words like *halloffame* and split them into *hall*, *of* and *fame*. But if the split is not possible or if the word already exists in the dictionary (for example *cupboard*), tokenization will not alter the word. This is coded using the concept of *dynamic programming*.

4.2 Context based probability calculation

We started implementing context based spell check algorithm on similar lines with the one described in [3]. After tokenizing the sentence, we are checking for the common confusion words that usually occur. We then take confusion sets for each ambiguous word from the data present beforehand. For each word in the confusion set of a target word, we take a set of context words within a ± 3 window frame and find their co-occurrences with the target word from the corpus(these word co-occurrence frequencies were generated beforehand) and find the posterior probability of words in confusion set applying bayesian rule and taking the independence of likelihoods. The word which gives maximum probability is taken as the target word.

$$P(w_j|c_{[-3,3]}) = \prod_{i=-3}^{+3} P(c_i|w_j)P(w_j)$$

4.3 Collocations

Firstly we generated frequently occurring parts of speech tags along with a word for all the words in our corpus. We took a window length of 2 word distance from target word in all possible ways that includes our word and count the POS

sequences that occur with their frequencies.
For a given sentence, similar to context words, we tokenize the sentence and check for common confusion words. We take the confusion sets for these ambiguous words. For each word in a confusion set we take all possible sequences of words with the target word within a distance of 2 and calculate their occurrences. We find the POS tags for these cases and find the frequencies of these words from the trained data. We take the average of these frequencies for each word and the one that gives maximum probability among the confusion set as the target word.

4.4 Bayesian hybrid

To get the complementary benefits from both the above methods, We take the product of probability of both co-occurrences of both context words and collocations) from previous results for each target word and the word which gives maximum probability is taken among the confusion set.

References

[1] Mark D. Kemighan , Kenneth W. Church, William A. Gale. *A Spelling Correction Program Based on a Noisy Channel Model*

[2] Justin Zobel, Philip Dart. *Phonetic String Matching: Lessons from Information Retrieval*

[3] Andrew R. Golding. *A Bayesian hybrid method for context-sensitive spelling correction*