IIT Madras

CS6370

Natural Language Processing

Spellcheck

Team:	*Roll No:*
Abhishek Naik	CS13B030
Mohan Bhambhani	CS13B036
Shiva Krishna Reddy	CS13B051
YSSV Sasikiran	CS13B055

1 Introduction

In this assignment we deal with the classic problem of Spell Check. The Spell Check can be categorised into three parts:

1. **Word spell check** - Suggest corrections for stand alone erroneous words.
2. **Phrase spell check** - Words present in phrases need to be checked for spelling errors and corrected. These words could be part of the dictionary, but are not the intended words in the context of the phrase.
3. **Sentence spell check** - Similar to phrase, entire sentence needs to be checked for spelling errors.

We deal with multiple approaches to each of the above sub-problems and use the models that performed best or a hybrid of the best models by weighted scores.

The assignment is implemented in *Python*.

2 Corpora used

1. Brown Corpus : The Brown Corpus from here in `nltk` is annotated with part-of-speech tags, and is used in the context-sensitive bigram model.
2. COCA : Corpus of Contemporary American English (COCA) from here contains 1,000,000 most frequent n-grams. This was used for contextua l n-grams spell check.
3. Natural Language Corpus Data: Beautiful Data, from here, which has some million most frequent words, along with counts for all 2-letter (lowercase) bigrams, 3-letter trigrams, etc.

3 Assumptions

We make some reasonable simplifying assumptions on the errors in words and phrases to boost performance with respect to accuracy and computation time.

For a word the error is defined as the Damerau-Levenshtein distance from the correct intended word. The assumptions are as follows:

1. Words of length less than 5 may have atmost 2 errors.
2. For any incorrect word of length more than 5, there is atleast one trigram of letters that occur in the correct word. We make use of this assumption during the candidate generation.
3. As specified in the problem statement, it is being assumed that there is *only one incorrect word* per query (phrase or sentence).
4. In case of splitting, a correct word is split only in 2 parts, not more.

4 Word Spell Check

Given a standalone incorrect word, to find the correct word that could have resulted in that typo. This is a 2-stage process:

1. Candidate generation
2. Candidate ranking

4.1 Candidate generation

The following methods were considered:

4.1.1 Peter Norvig's method

The Norvig approach [4] to generate the candidates was followed, wherein a simple edit to a word is a deletion (remove one letter), a transposition (swap two adjacent letters), a replacement (change one letter to another) or an insertion (add a letter), which is also the definition of the Damerau–Levenshtein distance.
An edit-distance of 1 can be a big set. For a word of length n and alphabet a, there will be n deletions, $n - 1$ transpositions, $a * n$ alterations, and $a * (n + 1)$ insertions, for a total of $2n + 2an + a - 1$ (of which a few are typically duplicates).

Clearly, this grows polynomially in length of the word, and also isn't language independent. (For ex: English has $a = 26$ letters, and Chinese has 70,000 Unicode Han characters.

We initially used this approach, with upto 3 edit-distances. It resulted in a large number of candidates, but that is ok, as the percentage of them present in a dictionary are quite less.

An obvious limitation of this approach was that it does not even generate candidates of higher edit-distances for consideration, because the computation becomes more and more exhaustive as the size if the string and the extent of edit-distance grows.

4.1.2 FAROO's implementation

A 'symmetric delete' operation [1] is performed in the following manner:

1. Generate terms with an edit distance ≤ 2 (deletes only). Store the shorter words with the words that generated them. Ex : delete(sun, 1) = sn, and delete(sun, 1) = sn, so the dictionary entry is {'sn' : ['sun', 'sin']} .
   This has to be done only once during a pre-calculation step.
2. Generate terms with an edit distance ≤ 2 (deletes only) from the input term and search them in the dictionary.

Following these operations, comparing both sets of terms is equivalent to performing Norvig's addition, deletion, insertion and transposition.[1]

The number 'x' of deletes for a single dictionary entry depends on the maximum edit distance: $x = n$ for edit distance=1, $x = n * (n - 1)/2$ for edit distance=2.

4.1.3 Metaphone and Double Metaphone

Metaphone is a phonetic algorithm for generating encoding of words depicting their English pronunciations. Metaphone codes use the 16 consonant symbols 0BFHJKLM-NPRSTWXY. The '0' represents "th", 'X' represents "sh" or "ch", and the others represent their usual English pronunciations.

It does not produce phonetic representations of an input word or name; rather, the output is an intentionally approximate phonetic representation. The approximate encoding is necessary to account for the way speakers vary their pronunciations and misspell or otherwise vary words and names they are trying to spell.

The Double Metaphone phonetic encoding algorithm is the second generation of the Metaphone algorithm. It is called "Double" because it can return both a primary and a secondary code for a string; this accounts for some ambiguous cases as well as for multiple variants of surnames with common ancestry. For example, encoding the name "Smith" yields a primary code of SM0 and a secondary code of XMT,

[1] The complete explanation is avoided due to reasons of brevity. It can be looked up in [1], or the team can explain it during the viva.

while the name “Schmidt” yields a primary code of XMT and a secondary code of SMT–both have XMT in common.

Double Metaphone tries to account for myriad irregularities in English of Slavic, Germanic, Celtic, Greek, French, Italian, Spanish, Chinese, and other origin. Thus it uses a much more complex ruleset for coding than its predecessor; for example, it tests for approximately 100 different contexts of the use of the letter C alone.

We used the candidates returned by the Double Metaphone algorithm, and boosted their scores while ranking, to account for the phonetic errors that occur in spelling mistakes.

4.1.4 Inverted index of trigrams

The above mentioned methods make assumptions on the number of errors by limiting it to two. Here we make a relaxation on the assumption and assume that the misspelt word has atleast one trigram which has no errors.

1. We make a preprocessing on the dictionary to store an inverted index of all the trigrams possible and the words that contain that trigram.
2. For a misspelt word to be corrected, we split the word into all possible trigrams and extract the union of all the words linked to each trigram and use it as the candidate set.

This way we get words containing larger edit distances than 2. This model is particularly useful for longer misspelt words where the edit distance could typically be more than 2. To efficiently implement this, we used the dictionary data structure which uses hashing. The hashing is $O(1)$ in the average case. Here we blow up on the number of candidates genetrated, but every possible mistake is recovered.

4.1.5 Final approach used

All the above approaches have individual merits and demerits, and were combined to generate the best possible set of candidates for an incorrect word.

- The main candidate generation was performed with the inverted index of trigrams, which gives an list of candidates from which the typo could have generated from. The main advantage of this approach is that there is no constraint of an upper bound of edit-distance, as with Norvig’s and FAROO’s. This approach was used only when length of the input word is greater than 6.

- The Norvig approach was also used[2] to supplement the candidate generation because for words of size ≤ 5, say, the probability of even one trigram matching the correct word is close to zero. This results in a lot of 'bad' candidates.
- Finally, as mentioned earlier, the candidates generated by the phonetic algorithm are given a boosted score while ranking.

4.2 Candidate ranking

The candidates obtained by the above method are scored using the approach in [5] and the top candidates are found based on the score assignes.For a given misspelt word t, we find the value proportional to the posterior probability of each correct word, i.e

$$\begin{aligned} Pr(c_i|t) &\propto Pr(t|c_i).Pr(c_i) \\ \implies log(Pr(c_i|t)) &\propto \log\left(Pr(t|c_i)\right) + \log\left(Pr(c_i)\right) \end{aligned}$$

We make the Naive Bayes assumption on the $Pr(t|c_i)$ and approximate it as :

$$p(t|c_i) = \prod_j p(t_j|c) \text{ where } t_j \text{ is an error}$$

Hence we get:

$$log(Pr(c_i|t)) \propto \sum_j \log\left(Pr(t_j|c_i)\right) + \log\left(Pr(c_i)\right)$$

Firstly we find the t_js by using Dynamix Programming. We fill a matrix M containing all the edit distance information, i.e M[i,j] represents edit distance of $c_{i:n}$, $t_{j:n}$. We use this matrix to recursively find positions at which errors occur. Then each of the $\log\left(Pr(t_j|c_i)\right)$ is found by using the *rev[x,y]* , *del[x,y]* , *add[x,y]* and *sub[x,y]* mentioned in the paper[5] and dividing with the counts from [4].

The following is the algorithm:

- Given, a typo and a candidate, we first find all the edits between the typo and the candidate by backtracking the dynamic programming matrix generated while finding the Damerau–Levenshtein distance.

[2]FAROO's faster algorithm for candidate generation has large pre-processing overheads, and gives noticeable speedup for use-cases with 1000s of corrections. For our application, it would not provide a competitive edge, and Norvig's approach used instead.

- Then for each edit, we estimate the probability of that edit happening using the bayesian approach mentioned in [5].
- We use the assumption that each edit is independent of the other edits and hence we calculate the likelihood of the candidate given the typo by multiplying all the individual edit probabilities.
- We use the relative number of occurrences of the candidate as the prior.
- We add an additional smoothing parameter for reducing the dependence of probability score on number of edits.
- For all the candidates which are generated by the phonetic algorithm, we add an additional weight while calculating the prior for those candidates.
- Finally, we sort the candidates based on the probability rank generated.

5 Phrase and Sentence Spell Check

Phrase correction is a harder problem to solve compared to the non-word correction. Here we need to look at the context of the word in the sentence by examining the nearby words and then decide which word best fits the context. For this we obtained a set of commonly confused words compiled from a few sources and for every occurrence of a potentially confused words we compute the score of each of the possible words and assign the word with the highest score.
We used the following approaches:

5.1 Hidden Markov model of Bi-gram words

A Hidden Markov model is a statistical model where the system is modeled as a Markov process with hidden states. For context sensitive spell check, we employ HMMs in the following way based on an idea in [2].

- We model the Parts of Speech(POS) of a word as the hidden variable and the word itself as the observed variable.
- We first find the transition probability matrix for the parts of speech from the training data. We also find the emission probability of a word from a POS. We use the brown tagged corpus for the training purpose and estimate the transition and emission as follows.

$$Transition\ Probability(a,b) = \frac{\text{No.of times tag(b) follows tag(a)}}{\text{No. of times tag(a) occurs}}$$
$$Emission\ Probability(w,a) = \frac{\text{No.of times word w occurs with tag(a)}}{\text{No. of times tag(a) occurs}}$$

- Then for each word in the given sentence(may contain context-sensitive mistakes), if the word is present in any confusion set, we replace the word with all possible candidates from the confusion set and get the probability of the sentence being generated with each confusion word.
- We then choose that confusion word which gives highest probability of that sentence being generated.

In this approach, we find the probability of a candidate sentence being generated in the following way.

- For each word in the candidate sentence, we first generate all possible tags of that word from the dictionary. We append a starting tag at the beginning of the sentence. We also initialize the probability of generation to 1.
- Then for each possible tag of a current word, we find the probability of that tag being generated from all possible tags of the previous word multiplied with probability of that tag emitting the current word.
- For all the tags of the current word, we store the previous tag which gave highest probability of generation of sentence until that word along with the probability value.
- We then find the tag sequence which gives highest probability of generation of sentence by backtracking from the maximum probable tag of the last word.

However this model does not give good results when all the confusion words have the same POS tag since the score will be based only on the $Pr\left(y_i|POS_i\right)$ This is not acceptable since confusion words could be the same for significant number of cases (Ex: *between* and *among*).

Hence we used the Web-Scale N-gram Model described below

5.2 Web-Scale N-gram Model

This model is used only in the cases where all the potential words of a confused word have the same POS tag. In such a scenario the Markov Model simply assigns the $Pr\left(W_k|POS_K\right)$ as the score to each W_k. This model is based on [6], specifically the

SUMLM approach. Here we make use of the counts of 2-grams, 3-grams, 4-grams and 5-grams from the data provided by COCA[3] containing about a million ngrams' counts.

We assume that the phrase provided has already been accounted for all the spelling mistakes of non-words i.e words not occuring in the dictionary. For a confused word W_0 in a given sentence, we create the n-grams $2 \leq n \leq 5$ containing W_0, resulting in a total of at most 14 n-grams(2+3+4+5). For example for 5-grams :

$$W_{-4}W_{-3}W_{-2}W_{-1}\mathbf{W_0}$$
$$W_{-3}W_{-2}W_{-1}\mathbf{W_0}W_1$$
$$W_{-2}W_{-1}\mathbf{W_0}W_1W_2$$
$$W_{-1}\mathbf{W_0}W_1W_2W_3$$
$$\mathbf{W_0}W_1W_2W_3W_4$$

Now let $Y = \{Y_1, Y_2, ..Y_k\}$ be the set of possible confusion words of W_0. Then we replace W_0 with each of the Y_k and find the count of the resulting ngram. We add these counts from all ngrams ($n = 2, 3, 4, 5$) for each Y_k and use this as the score.

This model is similar to the naive bayes model, except that the prior log counts are not considered. This model also gave good results but suffers from the sparsity of the data. Hence we used this model only when the POS tags are same for all $Y_k \in Y$. Furthermore we have weighted the results using both the score from the corpus data and the score from the stand alone word check and this gave better results.

6 Word Segmentation

Word segmentation[7] deals with the problem of splitting a word which is missing spaces. As an example, the word "*footballground*" must be split as "*football ground*". For this we follow the below approach.

- We first generate all the possible splits of the word such that length of each split is less than a maximum limit($L = 20$).During the split generation, we memorize the call to the function with same inputs to reduce the split time.
- We then find the probability of the sequence of split words being generated in sequence using the Bi-gram counts and return that split which has the highest

probability.
$Pr(w_1, w_2, ..., w_n) = \Pi_{i=1}^{n} Pr(w_i|w_{i-1})$

7 Final Combination

We describe the steps in the final algorithm given a spelling error to be corrected:

1. For word spell check, we just pass the typo to the program and use the above given algorithms for candidate generation and ranking and return the top 10 ranked words and their scores.

2. For phrase and sentence spell check, we first try to resolve the errors where a single word was split as two words.(Ex:*hand bags* must be written as *handbags*). We achieve this by combining each consecutive word in the sentence and giving it to segment algorithm. If the segment algorithm returns the combined word with high probability, we merge the words and return.

3. We then check each word in the dictionary. If the word is not present in dictionary it may be a spell error or a segmentation error.

4. For each word which is not in dictionary, we give the word to the segment function, and if it divides the word with a probability greater than threshold, we consider that as error and return.

5. If the segment function did not split the word, we then pass the word to word spell check algorithm to obtain the ten most relevant words. We then replace the word with the most relevant and add all the other 9 words in the confusion set of the most relevant word. We then pass to the context spell check algorithm.

6. If all the words are present in dictionary, we may still have a word which is in dictionary but may not be in confusion set.(Ex: *cort* and *cost*).

7. In this scenario, for each word w_i, we give this word to the spell check and get back ten most relevant words. We then add these words to the confusion set of the first word.

8. Now, if the first word is not in confusion set, we directly use the N-Gram based method to correct the word. This is because we may still have many words which are in the top 10 and have the same parts of speech for which bi-gram HMM does not work good.

9. If the first word is in confusion set, we directly call the hybrid method for ranking the words.
10. We finally output the top 3 words for each confusion word in each phrase separately.

8 Results

Both the stand alone word error correction and the phrase/sentence correction gave good results on the input data provided. An **MRR of 1.0** was obtained for almost all the word test cases. As for the phrase errors, we got an **MRR of 1.0** for most of the cases. However we got a MRR of 0.5 for phrases like *roff of the house* where our result was *off* followed by *roof*. This can be attributed to a large number of n-grams in the training data with the word *off*. We could not perform well on a few phrases like *coyote fox* which was almost unseen by the training corpus. Hence the sparsity of the data was the major reason for some of these extreme cases.

References

[1] Faroo - fast candidate generation. http://blog.faroo.com/2012/06/07/improved-edit-distance-based-spelling-correction/.

[2] A mixed trigrams approach for context sensitive spell checking. http://nlp.cs.uic.edu/PS-papers/spell-cicling07.pdf.

[3] N-grams data : Corpus of contemporary american english. http://www.ngrams.info/download_coca.asp.

[4] Peter norvig's basic spell check. http://norvig.com/spell-correct.html.

[5] Spelling correction program based on noisy channel model. http://www.aclweb.org/anthology/C90-2036.

[6] Web-scale n-gram models for lexical disambiguation. http://www.clsp.jhu.edu/~sbergsma/Pubs/bergsmaLinGoebel_WebNGram_IJCAI09.pdf.

[7] Word segmentation. http://norvig.com/ngrams/ch14.pdf.