Spell Correction Assignment Report

Rahul Kejriwal (CS14B023)
Srinidhi Prabhu (CS14B028)

March 19, 2018

1 Introduction

The Spelling Correction problem was introduced in the 90s and Kernighan [1] was the first one to make significant progress in this direction. He introduced the noisy channel model which is still widely used today.

Formally, the problem requires a system to take as input incorrect words (generally rejected by a system that checks for incorrect spellings) and output a ranked list of plausible corrections. Generally, it is assumed that the words given as input are indeed incorrect.

This problem has two variants:

- **Context-insensitive Spell Correction:** In this case, only the incorrectly spelled words are given as input to the system.
- **Context-sensitive Spell Correction:** In this case, the incorrectly spelled words along with a few context words are given to the system. Such systems can also correct real-word spelling errors (errors where the incorrectly spelled word is also a word in the dictionary) and can also fix minor grammatical errors (like of/off).

2 Methodology

Spell Correction using Noisy Channel Models consists of 2 phases:

1. Candidate Generation
2. Ranking of Candidates

Here, ranking of candidates generally uses posterior probabilities:

$$\begin{aligned} correction &= \underset{c \in Candidates}{\operatorname{argmax}} \; P(c|wrong\_word) \\ &= \underset{c \in Candidates}{\operatorname{argmax}} \; P(wrong\_word|c)P(c) && [Baye's\ Theorem] \\ &= \underset{c \in Candidates}{\operatorname{argmax}} \; log(P(wrong\_word|c)) \; + log(P(c)) \end{aligned}$$

As an optimization, a tunable hyperparameter λ is introduced to weight the contributions of the likelihood and prior terms. This allows us to trust one measure more than the other. This approach is given in Jurafsky & Martin [2].

$$correction = \underset{c \in Candidates}{\operatorname{argmax}} \; log(P(wrong\_word|c)) + \lambda log(P(c))$$

In order to completely describe the model, we need to state the following:

1. Candidate Generation Strategy and Dictionaries used
2. Prior estimation
3. Likelihood estimation

2.1 Word-Level Spell Check

1. **Dictionaries Used:** We used the two given word lists along with the YAWL (Yet Another Word List) list of valid Scrabble words [3]. We then filtered out unacceptable spellings using WordNet. Our final list had 123436 valid words.

2. **Candidate Generation:** Spelling errors are of two kinds:

   - *Typographic Errors:* Such errors occur due to typos. Transposition of adjacent characters and substitution of characters by their adjacent character on the keyboard are a few examples of such errors.
   - *Orthographic Errors:* Such errors occur due to cognitive failures. The use of a similar sounding word or alternative spellings with similar pronunciations are examples of this kind of errors.

   Typically, typographic errors are characterized by small edit distances to the correct word and orthographic errors are characterized by similar pronunciations to the correct word.

   We combine two strategies for candidate generation to account for both kinds of errors:

(a) To handle typographic errors, we generate all valid words upto 2 edit-distance away from the incorrect word. As all possible edits can be numerous, we speed up this process by only allowing valid prefixes during the edits generation process. This approach is given by Norvig [4].

(b) To handle orthographic errors, we use phonetic hashes of words as produced by the Double Metaphone algorithm. We maintain an index of all valid words based on their phonetic hashes. At runtime, we retrieve all valid words having the same phonetic hash as the incorrect word. This method allows us to retrieve the right word in most cases.

The combination of these methods gave us pretty high word-processing speeds. We were able to process 160-200 incorrect words per second.

3. **Prior Estimation:** We used unigram word frequencies from Norvig's *count_1w.txt* [3]. We then did Laplace Add-1 smoothing. We computed and stored log priors of all valid words.

4. **Likelihood Estimation:** We didn't actually estimate the likelihood probabilities - $P(wrong\_word|correction)$, rather we used a surrogate for this metric. We computed the Damerau-Levenshtein distance between the incorrect word and possible corrections and then we normalized the cost by the word length. The higher this metric, the lower is the chance that the candidate is a good correction of the current word. Thus, we used the negative of this quantity as a surrogate for likelihood.

   The rationale behind normalization of the edit distance is that we are possibly ok with higher edit distance as long as the incorrect word under consideration is long as well. It is to be expected that longer words will have higher number of errors.

   The Damerau-Levenshtein distance can use variable cost schemes for insertion and deletion of particular characters, substitution of a particular character with another and for transposition of a pair of characters. However, it is to be noted that these costs are context independent. The cost of replacing an 'e' with an 'i' remains the same regardless of the previous character. Although Kernighan's intial paper [1] uses probabilities conditioned on the previous character, we did not accomodate such context information in our cost schemes. The reason is that available edit distance libraries don't support such cost schemes and more importantly, estimating such costs based on context requires very large number of erroneous and correct word pairs to estimate counts of each possible error in each possible context.

Thus, our final scoring function became:

$$score(candidate|wrong\_word) = -\frac{dam\_lev(wrong\_word, candidate)}{len(candidate)} + \lambda log(P(candidate))$$

where $dam\_lev(w, c)$ computes the Damerau-Levenshtein distance of w from c and $len(w)$ computes length of word w.

2.2 Phrase-Level and Sentence-Level Spell Check

1. **Dictionaries Used:** We used the same dictionary as the one used for the correction of words.

2. **Methodology:**

   (a) **Identification:** In a phrase or sentence, we do not know the word that is to be corrected beforehand. So, the first task would be to identify which word in the phrase to correct. To identify the wrong word, we follow the below procedure:

      i. Scan all words in the phrase or sentence.

      ii. If there is a word that is not in the dictionary, then that is the word to be corrected.

      iii. If all words are in the dictionary, we check if there are words that have homophones. The list of homophones is pre-built. If replacing the word with the homophone gives us a good enough confidence(more details on this below), then we identify this word as the one to be corrected.

      iv. If all words are in the dictionary and none of them can be replaced by a homophone, then we identify the word with the least confidence as the one to be corrected.

   (b) **Calculation of confidence:** We identified two methods to calculate confidence:

   - **Bigrams:** We use bigram frequencies to calculate the confidence of a word in a context. Suppose the words present in a phrase/sentence are in the order A B C. We define the confidence as follows:

$$confidence(B) = (bigram(A, B) + bigram(B, C)) + \alpha * unigram(B)$$

     where *bigram(A, B)* is the bigram frequency of the word B occurring after word A, and *unigram(B)* is the unigram frequency of the word B. The sum of bigram counts is a measure of the likelihood of the word B occurring in the context and the unigram count is like the prior of the word B occurring. The weight α is a hyperparameter that is tuned.

   - **Word vectors:** We find the word vectors for all the words surrounding the given word upto a given window k. We find an averaged representation of all the word vectors and then find the similarity of the word with its context. This gives a measure of the similarity of a word with its context. For this assignment, we use 50-dimensional GloVe word embeddings. [5]

$$confidence(B) = \vec{B} \bullet \vec{context}$$

where $\vec{B}$ is the word and $\vec{context}$ is the vector average of all words around the word B up to a window of size k. We use the dot product as a method to compute similarity between two vectors.

(c) **Correction:** To correct a word, we first generate all candidates for the incorrect word using the candidate generation described in the word check section. We then calculate the confidence scores for each candidate by replacing the incorrect word with the candidate. We normalize the scores by dividing by a quantity proportional to the edit distance. This is because candidates with lower edit distance are more likely to be the intended word. The candidates are sorted in descending order of their confidences and the top 3 suggestions are output.

3 Experiments and Results

3.1 Word-Level Spell Check

We discuss two models we made - one for the interim submission and the other for the final submission. The only difference between them was the cost schemes used for various edits in the Damerau-Levenshtein distance function.

1. The initial model, which we refer to as **AgnosticModel** uses uniform costs for each possible edit.

2. The final model, which we refer to as **DiscriminativeModel** uses non-uniform costs for different edits.

   We had around 14K pairs of words where the incorrect spelling was 1 edit distance away from the actual spelling. Using them we estimated the frequency of different kinds of edits. We had to estimate frequencies of more than 1400 possible kinds of edits and thus, the data was sparse. In order to compensate for that, first we added one to the count of each possible edit and then, we used:

   $$cost(edit) = log(1000) - log(freq(edit))$$

   We used log(1000) as 1000 was around the maximum edit frequency we encountered.

We report here the MRR we get on the evaluation set as well as for a few other datasets available from Norvig [3] and Roger Mitton [6]. We compare our models to a few openly available python libraries that perform spell correction - PyEnchant and autocorrect. Bracketed values besides the dataset denote number of test instances for that dataset.

Dataset	DiscriminativeModel	AgnosticModel	PyEnchant	autocorrect
Speed	185 words/sec	185 words/sec	9 words/sec	18 words/sec
Interim Dataset (40)	0.810	0.784	0.796	0.667
aspell (531)	0.642	0.637	0.598	0.390
wikipedia (2455)	0.762	0.740	0.822	0.648
holbrook-missp (1771)	0.350	0.332	0.321	0.200
birkbeck (36133)	0.457	0.450	0.385	0.291
spell-errors (39710)	0.467	0.459	0.405	0.304

Note: DiscriminativeModel and **AgnosticModel** output 10 suggestions per word, **PyEnchant** outputs variable number of suggestions based on its confidence scores and autocorrect only outputs one suggestion.

3.2 Phrase-Level and Sentence-Level Spell Check

Corresponding to the two confidence measures described in section 2.2, we get two models of correction. Let the two models be as follows:

1. **BigramModel** which uses the bigram frequencies as a measure of confidence.
2. **VectorModel** which uses similarity of word vectors as a measure of confidence.

The interim submission used the VectorModel for both phrases and sentences. The final submission, however, would use the BigramModel for phrases and the VectorModel for sentences. We compare the two models on two aspects:

1. Detection Accuracy(DA): The accuracy with which the model is able to detect the wrong word.
2. MRR: The mean of the reciprocal rank of the correct word among the list of suggested corrections.

The values are reported on the interim public dataset.

The measures for the phrases dataset(46 phrases) are as follows:

	DA	MRR
BigramModel	0.913	0.514
VectorModel(k=4)	0.826	0.460

The measures for the sentences dataset(40 sentences) are as follows:

	DA	MRR
BigramModel	0.625	0.233
VectorModel(k=4)	0.575	0.296

The measures for the VectorModel for different k over the sentences dataset is as follows. The Detection Accuracy for all values of k was 0.575.

k	MRR
1	0.270
2	0.292
3	0.3
4	0.296
5	0.296
6	0.3
7	0.296
8	0.296

4 Discussion

4.1 Word-Level Spell Check

It is natural to think that the use of non-uniform costs that capture the probabilities of different possible edits should improve model performance. However, in our experiments, we see that the gain is very less. Using vanilla edit distances with uniform costs also appears to discriminate between candidates well. This might be because our cost scheme does not reflect the true probabilities of different edits.

Also interestingly, the use of simple edit-distance based metrics in place of likelihood probabilities $P(wrong\_word|candidate)$ works surprisingly well. The **AgnosticModel** requires almost no data to be trained as it does not use frequencies of different edits in incorrect spellings. Using only a dictionary of valid spellings and a unigram distribution of words, our **AgnosticModel** was able to outperform existing libraries in many cases and at much higher word-processing speeds.

The use of Double Metaphone algorithm for candidate-generation was also quite helpful as it often compensated for typographic errors that caused the incorrect spelling to have more than 2 edit distance from the correct spelling.

4.2 Phrase-Level and Sentence-Level Spell Check

We see that the BigramModel works better for the case of phrases while the VectorModel works better for the case of sentences. This is an expected result. There would be about 4-5 words in a phrase and it is highly likely that the corrected word largely depends on the word that precedes or succeeds it. However, in case of sentences, the word related to the wrong word may be present 3-4 words away from the wrong word. Using the VectorModel for phrases might bring in a lot of other words which would be a source of noise. This explains why the BigramModel works better for phrases while the VectorModel works better for sentences.

5 Error-Analysis

5.1 Word-Level Spell Check

Using non-uniform edit costs in our **DiscriminativeModel** was able to fix a few errors that **AgnosticModel** was unable to correct:

Incorrect Word	**AgnosticModel**	**DiscriminativeModel**
aukward	award awkward ...	awkward award ...
peice	price peace ...	piece pic ...
sieze	size site ...	size seize ...
tierd	there tired ...	tired tiered ...

However, we also notice a few kinds of errors where our model performs poorly:

Incorrect Word	**DiscriminativeModel**
letf	lf etf elf let left ...
rember	ember amber member ...
revelent	prevalent reverent ...
tounge	tung lounge tonga tongue ...
leeiissy	less lies ...
vouture	voter vulture ...

A few points to note here:

- The first case *letf* shows that our system has learned that deletion of characters like ‘l’ and ‘e’ is more proabable as compared to the transposition of ‘tf’. In contrast, **AgnosticModel** does not make this error. This shows the limitation of likelihood estimates based on frequencies of edits seen in incorrect spellings.
- For *revelent* and *tounge*, we see that the system is unable to recognize the reordering of characters at long-distances (not transpositions). This is a limitation of the edit distance that we used which cannot capture long-range edits.
- For *leeiissy*, it is observed that the Double Metaphone algorithm correctly retrieves ‘lazy’ however our scoring function heavily penalizes this correction due to the high edit distance.
- For *vouture*, we see that one syllable *ture* replaces the correct *cher*. Here, even the Double Metaphone algorithm fails. It might be necessary to study edits at the level of syllables rather than at the letter level for handling such kinds of orthographic errors.

5.2 Phrase-Level and Sentence-Level Spell Check

For the phrase and sentence correction case, the model fails to give good suggestions when the correct word is at a large edit distance from the wrong word. In some cases, the correct word may not be part of the candidates because it is at a larger edit distance and a phoneme hash may not be the same as the correct word. In cases when the correct word is generated as a candidate, its score would be penalized more because it is at a larger edit distance. A word with a smaller edit distance that fits into the context would be preferred in such cases. We also find that there are some cases where the BigramModel gives better suggestions on sentences and VectorModel gives better suggestions on phrases.

6 Conclusion

In this work, we have aimed to develop a spell checker and correction system. Our results show that the word spell checker achieves performances close to some of the well known Python libraries. However, our phrase and sentence checkers are far from the state-of-the-art spell checkers. The reasons for this can be attributed to the compute power available to us. The suggestions could be improved by using higher dimensional word vectors or using trigram frequencies for estimation of confidence.

7 Future Work

1. Our cost scheme for different possible edits in Damerau-Levenshtein distance ignored the surrounding characters of the edit. Kernighan [1] in his paper used probabilities of edits conditioned on the character preceding the edit. This “context” information might be used for better likelihood estimates.

2. Often the phonetic hash of the incorrect spelling is different from that of the correct spelling but the hashes are only a small edit distance away from each other. The candidate generation process can be modified to include words whose phonetic hash are at 1 edit distance away from the phonetic hash of the true spelling.

3. Use of trigram frequencies to estimate confidence instead of bigram frequencies may be explored.

4. Use of higher dimensional word vectors to estimate confidence may be explored.

5. Use of weighted vectors to obtain confidence scores instead of plain average of vectors may be explored.

6. Use a combination of BigramModel and VectorModel and use the confidence output from both of them to weight the results and output suggestions.

References

[1] Mark D Kernighan, Kenneth W Church, and William A Gale. A spelling correction program based on a noisy channel model. In *Proceedings of the 13th conference on Computational linguistics-Volume 2*, pages 205–210. Association for Computational Linguistics, 1990.

[2] James H. Martin Daniel Jurafsky. *Speech and Language Processing.* 2017.

[3] Peter Norvig. Natural language corpus data: Beautiful data. http://norvig.com/ngrams/. (Accessed on 03/19/2018).

[4] Peter Norvig. Natural language corpus data. In Toby Segaran and Jeff Hammerbacher, editors, *Beautiful Data*, chapter 14, pages 219–242. O'Reilly, Oxford, 2009.

[5] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.

[6] Roger Mitton. Corpora of misspellings for download. http://www.dcs.bbk.ac.uk/ ROGER/corpora.html. (Accessed on 03/19/2018).