2ID90 Spell Checker Assignment

Group 19: Daan de Graaf, Yoeri Poels

April 4, 2017

1 Introduction

In this project we explore the creation of a spell checker, used for correcting english sentences with at most 2 errors per sentence (and no consecutive incorrect words). We do this using an approach of natural language processing we were taught in this course: reasoning with probabilities about the words and sentences that have to be corrected. In this report we will describe how we approached the problem, how we tried to solve it, and why we solved it the way we did.

2 Overall approach

To correct a sentence, our approach was to go through all the words in the sentence and find the best possible word in this place. We did this using a combination of the noisy channel model[?] and the Bigram model[?].

For every word in the sentence, we start by generating a candidate set of words for our current word: we create this by taking all words with a Damerau-Levenshtein Distance of 1 (so a single deletion, insertion, substitution or transpostion) to the current word. If the current word is also in the vocabulary (and thus a valid word), we also add this to the candidate set. We then check for the probability of the typo of every word in the candidate set (and a constant NO_ERROR for no typos), how common the word is (which will be taken to the power of constant LAMBDA to give it a weight), and how well it fits by looking at its neighbouring words (which will be smoothed out: this will be explained in section 3). The following is a simple version of our algorithm:

```
String Spellchecker(phrase) {
    String correctPhrase; //the correct phrase
    for (every word in phrase) {
        candidateWords = words with Levenshtein Distance of 1 to word in vocabulary;
        if (word is in vocabulary) {
            add word to candidateWords, typo probability for word is NO_ERROR;
        }
        if (right neighbour of word is not in vocabulary) {
            bestWord = word;
            add bestWord to correctPhrase;
            continue;
        }
        for (every candidateWord in candidateWords) {
            candidateWord.probability = probability of typo
                    * (probability candidateWord occuring)^LAMBDA
                    * smoothed probability of candidateWord occuring before/after its neighbours;
        }
        bestWord = candidateWord from candidateWords with the highest probability;
        add bestWord to correctPhrase;
    }
    return correctPhrase;
}
```

Algorithm 1: Overall Approach

Details of these probabilities will be explained in section 3. The constants in section 4, and some enhancemenets in section 5.

3 Phrase generation and evaluation

During phrase generation we find the best suggestion for a given phrase. A suggestion is generated word by word from left to right, using the previous word in the phrase if available to create bi-grams. For each word in the phrase we calculate the probability that it is the correct suggestion. This probability is calculated in three stages:

1. Candidate words generation (using the confusion matrix to find likely typos)

2. Calculating the probability of the occurrence of each of the candidate words
3. Calculating the probability of the occurrence of the bi-gram ending with the candidate word

We then combine these probabilities to obtain the most likely word. Stringing these together then yields the final suggestion for the correct phrase.

3.1 Candidate words generation

Candidate words are all words w_c^i that have a Damerau-Levenshtein distance of exactly 1 from the word in the phrase (w_p), plus that word itself (because it might be correct) with a predefined probability for a typo (0.9 in our implementation). For each of these words we calculate $P(t|c)$, where $t \subseteq w_p \wedge c \subseteq w_c$, such that t is a typo and should be replaced by c. For our purposes, we use:

$$P(t|c) = \frac{confusionCount(t,c)}{biCharCount(t)}$$

Here $confusionCount(t,c)$ is the number of times that t is a typo and c was intended, according to the confusion matrix. $biCharCount(t)$ is the number of times that the biChar t occurs in the corpus. The intuition behind this formula is that it calculates how often t should be changed out of all the times it occurs.

3.2 Unigram occurrence probability

We now calculate the probability of occurrence of each candidate word using the number of times it occurs in the corpus:

$$P(w_c) = \frac{NGramCount(w_c)}{corpusSize}$$

We divide the number of times the candidate word occurs in corpus divided by the total number of words in the corpus. This measures how common a word is.

3.3 Bigram probability

At this stage we calculate how likely the combination of the candidate word and the previous word is. To calculate the probability of the bigram we use Kneser-Ney smoothing:

$$P_{KN}(w_i|w_{i-1}) = \frac{max(c(w_{i-1},w_i)-\delta,0)}{\sum'_w c(w_{i-1},w')} + \lambda_{w_{i-1}} \cdot P_{KN}(w_i)$$

Where for w_{i-1}:

$$\lambda_{w_{i-1}} = \frac{\delta}{\sum'_w c(w_{i-1},w')}|\{w' : 0 < c(w_{i-1},w'\}|$$

And for a unigram w_i:

$$P_{KN}(w_i) = \frac{|\{w' : 0 < c(w',w_i)\}|}{|\{(w',w'') : 0 < c(w',w'')\}|}$$

In order to express this in java, we derive:

$$\begin{aligned} c(x,y) &= getNGramCount(x,y) \\ \sum_w^{'} c(w_{i-1},w') &= biGram1Total.get(w_{i-1}) \\ |\{w' : 0 < c(w_{i-1},w'\}| &= biGram1.get(w_{i-1}) \\ |\{w' : 0 < c(w',w_i)\}| &= biGram2.get(w_i) \\ |\{(w',w'') : 0 < c(w',w'')\}| &= biGramCount \end{aligned}$$

Where $biGram1Total$ is almost identical to $biGram1$, only the first increments using the count attached to each biGram. Or in java code:

$$\begin{aligned} &biGram1.put(w1, biGram1.getOrDefault(w1,0)+1) \\ &biGram1Total.put(w1, biGram1.getOrDefault(w1,0)+count); \end{aligned}$$

Applying the appropriate substitutions yields a correct java implementation.

3.4 Combining

The probability of a typo $P(t|c)$ defines the noisy channel probability. Multiplying $P(w_c^i)$ and $P_{KN}(w_i|w_{i-1})$ yields the language model probability. To obtain the final probability we combine these two:

$$P = P(t|c) * P(w_c^i)^\lambda * P_{KN}(w_i|w_{i-1})$$

Where λ is a calibration parameter (set to 0.3 in our implementation).

Note that in our implementation we have tweaked this formula to improve accuracy, that formula looks like:

$$P = \frac{-1}{ln(P(t|c) * 0.8)} * P(w_c^i)^\lambda * P_{KN}(w_i|w_{i-1}) * P_{KN}(w_i - 1|w_{i+1})$$

This is explained in more detail in section *5. Advanced enhancements.*

Give a full discussion of how you have implemented phrase generation and have the best candidate sentence is selected. In particular, describe what rule the confusion matrices and bi-grams play a role in attaching a value/probability to a candidate sentence. Also describe what type of smoothing you use and how you have implemented it.

4 Results and evaluation

Our implementation corrects 31 out of 33 test cases of the training set, and 39 of the final set. The two failed test cases are:

1. "**boing** gloves shield the knuckles **nut** the head", corrected as "boxing gloves shield the knuckles **but** the head", where "but" should have been corrected as "not". This error is hard to catch, as it requires the spellchecker to use higher order ngrams. The spellchecker simply compairs the probability of "knuckles but the" with "knuckles not the", and in this slim context the spellchecker is unable to choose the right correction. According to the confusion matrix "not" is also confused with "nut" more often than "but".
2. "**laying** in the national footbal league was my dream" is unchanged by the spellchecker, where it should have corrected "laying" to "playing". But also in this case a higher order ngrams or semantic analysis would be necessary. "laying in" is a very common bigram, so there is no reason for the spell checker to correct this.

Given how hard it is to correct these errors, we are very satisfied with the performance of our spellchecker.

5 Advanced enhancements

In order to enhance the performance of our spell checker, we added a few features to its word-deciding process.

The spellchecker works from left to right, so while correcting words, its right neighbour has not been corrected yet. We simply check if this right neighbour is in the vocabulary: If not, this means it has to be corrected. Since we know that there are never 2 consecutive words with a spelling error, this implies the word currently being checked is already correct, and thus we conclude this and stop trying to find an alternative word.

Another enhancement is the way we deal with the probabilities of the typos. Since these greatly varied they would often have too much control over what word would be choosen as the correct word. As such, in order to make the differences in type probability not have too much influence, we evaluated them in the following way:

$$Typoscore(t|c) = \frac{-1}{0.8 \cdot \ln(P(t|c))}$$

This resulted in them not having too much control over the correct word but still being significant enough, which lead to much better results. We came to this formula by simply looking at the probabilities and reasoning what would be a reasonable range for them to be in, and created this formula to roughly map them to this range. These enhancemenets lead to the adapted, and final version of our overall algorithm:

6 Conclusions and contributions

Our spell checker ended up performing quite well. On the learning test set it corrected 31 out of 33 sentences, whereas on the final set it corrected 39 out of 50 sentences. We ended up mostly following the Noisy Channel Model and Bigram approach, with Kneser-Ney smoothing for the bigram model and some small enhancements

```
String Spellchecker(phrase) {
    Double NO_ERROR = 0.8; //typo probability for words with no change
    Double LAMBDA = 0.3; //weight for the occurence of a word
    String correctPhrase; //the correct phrase
    for (every word in phrase) {
        candidateWords = words with Levenshtein Distance of 1 to word in vocabulary;
        if (word is in vocabulary) {
            add word to candidateWords, typo probability for word is NO_ERROR;
        }
        if (right neighbour of word is not in vocabulary) {
            bestWord = word;
            add bestWord to correctPhrase;
            continue;
        }
        for (every candidateWord in candidateWords) {
            candidateWord.probability = Typoscore(probability of typo)
                                * (probability candidateWord occuring)^LAMBDA
                                * smoothed probability of candidateWord occuring before/after its neighbours;
        }
        bestWord = candidateWord from candidateWords with the highest probability;
        add bestWord to correctPhrase;
    }
    return correctPhrase;
}
```

Algorithm 2: Final Overall Approach

to improve its performance. The final approach is found at Algorithm 2 in section 5, with details regarding the implementation of some of these parts in secton 3.
Contributions:

	implementation	**documentation**	**total #hours**
Daan	50%	50%	22
Yoeri	50%	50%	22

References

[1] Wolfgang Ertel. *Introduction to Artificial Intelligence*. Springer Publishing Company, Incorporated, 1st edition, 2011.