Spellchecker

Martijn Gösgens 0914954
Dominique Sommers 0895679

March 28, 2016

1 Algorithms

1.1 Candidate words

Algorithm 1 getCandidateWords(String word, double numWords)

```
ZET IK NOG OM IN PSEUDOCODE...
$mapOfWords$ to be filled with all candidate words
Get all words by inserting one letter for every letter in the alphabet at one position of the word for every possible position of the word and put them in $mapOfWords$.
Get all words by deleting one letter of the word for every letter of the word and put them in $mapOfWords$.
Get all words by transpositioning two letters of the word for every two letters of the word and put them in $mapOfWords$.
Get all words by substituting a letter of the word with another letter for every letter in the alphabet and every letter of the word and put them in $mapOfWords$.
For all words in $mapOfWords$, the noisy channel probability is added by dividing the confusion count with the normalization of the word.
$resultMapOfWords$ to be filled with possible good candidate words
Put all candidate words with a higher probability than 0 in $resultMapOfWords$.
For all words in $resultMapOfWords$, the word probability is added by taking the sum of the $log_1 0$ of the noisy channel probability and the $log_1 0$ of the word probability as the frequency of the word divided by the number of monograms + the number of words.
Put the word itself if it is contained in the vocabulary and add the word probability of the $log_1 0$ of the number of words - 1 divided by the number of words if the number of words is higher than 1. Else add word probability of 0.5.
Return $resultMapOfWords$
```

This function gets all the candidate words of the input word. These candidate words have a maximum Damerau-Levenshtein distance of 1 which means that the word is only altered by at most 1 insertion, deletion, transposition or substitution. The candidate words with a noisy channel probability of 0 will then be deleted and the word probability is calculated. This is determined by the sum of the noisy channel probability and the $log_1 0$ of the frequency of the word divided by the number of monograms added to the number of words. Then the word itself is added if it is contained in the vocabular and the word probability is determined by the number of words - 1 divided by the number of words if the number of words is higher than 1. Otherwise, the word probability will be 0.5.

1.2 Find correct word

Algorithm 2 findCorrect(String previousSentence, String nextSentence, double currentProbability, int correctionsMade)

```
  TODO
```

1.3 Good-Turing Smoothing

Algorithm 3 getSmoothedCount(String $NGram$)

```
  Construct a sorted map $freqOfFreq$ with the frequency of all occurring words as keys and the frequency of these frequencies as values
  $maxFreq \leftarrow$ the most frequent word
  $sumCounts \leftarrow$ the sum of all frequencies
  $c \leftarrow$ the frequency of the word $NGram$
  **if** $c == 0$ **then**
      Return $freqOfFreq.get(1)/sumCounts$
  **else if** $c == maxFreq$ **then**
      Return $c/sumCounts$
  **else**
      $N_c \leftarrow freqOfFreq.get(c)$
      $nextC \leftarrow$ the lowest frequency higher than $c$
      $N_{c+x} \leftarrow freqOfFreq.get(nextC)$
      Return $(c + x) * N_{c+x}/N_c$
```

For the smoothing of the counts of the words, we chose to use Good-Turing Smoothing with a slight adjustment. The smoothing of the counts is to determine the chance of a word occurring again. The smoothed count will be slightly lower than the concrete count since there are words that have never occurred before. This difference is determined by the frequency of the frequency of the words. With the Good-Turing algorithm the smoothed count will be determined be the frequency of the frequency of the word and the freq. of the frequency + 1. But since there are gaps between frequencies, especially when the frequency gets higher, it serves no value to take the freq. of the frequency + 1 since this is zero. In our implementation we therefore take the next occurring frequency to get a value. When the gap between frequencies is high, this gets more unreliable. There is a solution to avoid this by drawing a line of the frequencies which is the best-fit power law, but we did not have the time and knowledge to implement this yet.

2 Results

The results of our spell checker are quite good. Most of the words are corrected correctly, with some exceptions. There are however also some words that are spelled correctly, but are corrected. This problem appears mostly when the correction is a word which occurs much more frequent than the corrected word, but in the particular sentence, the corrected word is in fact the correct word.

3 References

1. None

4 Contribution

Martijn Gösgens: TODO.
Dominique Sommers: I implemented the Good-Turing Smoothing algorithm and decided to take the next frequency instead of add-one smoothening or the more complicated Good-Turing Smoothing with the best-fit power law. Besides that, I wrote most part of the report. I also made a function to get the best candidate word for every word in the sentence, but this was later adjusted and corrected by Martijn adding the bigrams of the sentences.

5 Manual

The spellchecker takes sentences as input. These can consist of multiple words and multiple sentences. An important thing to note is that the spellchecker can only correct words with a Damerau-Levenshtein distance of 1 which means that each input word may only be altered by at most 1 insertion, deletion, transposition or substitution. The words in the input sentences may not have interpunction or any other kind of character than the english alphabet except for the single apostrophe ('). All letters of all words should be in lower case. After giving the input, the spellchecker will check for spelling mistakes and automatically replace them to the correct spelling of the word.