Spell checker

2ID90 Artificial Intelligence
Team 08
Wilco Brouwer (0816718)
Peter Wu (0783206)

April 2, 2015

1 Introduction

In this report we will describe our solution to the second assignment, which consists of designing a spell checker algorithm and its implementation in Java. The algorithm is provided with a sentence which contains zero to two misspelled words with an Damerau-Levenshtein edit distance of 1. In addition, misspelled words are not consecutive.

The expected output is a sentence where these errors have been corrected.

2 Approach

To remove spelling errors from a text, one has to generate possible corrections and evaluate them. Spelling suggestions are done on a word-by-word basis.

To obtain a list of possible words, *candidate words* will generated with a Damerau-Levenshtein distance of 1 to the original word. While doing this, a noisy channel probability can also be obtained which is needed later for word evaluation.

For each of word from the sentence, the word will be evaluated against the context of the word. This includes the previous and next words for n-grams. Since we only have the guarantee that the words next to the misspelled one are correct, tri-grams cannot reach out for the following two words before or after the current word.

Finally the most likely word may be chosen. If this is different from the original word, then the search may continue to find a possible second and third misspelled word.

2.1 Overview

The program is given a single sentence which is split in words. The indices of words that are surely correct are remembered (`readonly_words` in the listing below). Once a rating is found which is higher than the initial one, then the word is replaced.

When a word was modified, the hunt for more faulty words continue until all of them (less than three) are found. Otherwise, it is assumed that the sentence has no more spelling errors.

Pseudo-code:

```
words = input.split(' ')

# Initially a spelling suggestion can be proposed for all words.
readonly_words = [False] * len(words)

# Attempt to find wrongly spelled words.
for attempt in range(0, 3):

    for word_index, word in enumerate(words):
        # Skip words if these were modified before or if they were
        # consecutive to a previous word.
        if readonly_words[word_index]:
            continue

        # Finds the likelihood of the current word, it is described later
        rating_old = getLikelihood(word)

        # Generates a list of candidate words, it is described later.
        candidates = getCandidateWords(word):

        # Initially there is no best word.
        bestCandidate = None

        # Iterates through the candidates until one is found that is more
        # likely than the current word.
        for candidate in candidates:
            # Calculates the likelihood for the candidate word. If it is
            # better than the current word, then it is chosen.
            rating = getLikelihood(candidate)
            if rating > rating_old:
                bestCandidate = candidate

        if bestCandidate is not None:
            word[word_index = bestCandidate
            # Ensure that these words are not changed again in later
            # iterations. (Bounds checks are omitted for brevity.)
            readonly_words[word_index-1..word_index+1] = 0
        else:
            # no more suggestions, abort
            break

# words now contains the sentence with misspelled words corrected
```

2.2 Generation of candidate words

A naive approach to generate candidate words in the $getCandidateWords()$ function is to generate all possible combinations of the alphabet, and then take the intersection with the dictionary. This does not scale and uses $O(27^n)$ memory and time.

Instead we chose for a smarter approach. As we know that the misspelled word have a Damerau-Levenshtein distance of 1, candidates are generated by cycling through each letter and applying an insertion, deletion, substitution or transposition.

While doing these operations, the Noisy Channel Probability can also be obtained since we have a confusion matrix and know kind of operation was executed.

Pseudo-code (bounds checks are omitted, assume that such operations do not generate words. The actual Java implementation has decomposed methods for each of the operations):

```
def getCandidateWords(word):
    # Walk through the letters of a word. Words that are not in a
    # vocabulary will be dropped. While generating words, a Noisy Channel
    # Probability will also be calculated and attached to the word.
    for i in word[0..n]:
        # Insertion of a new character at position i.
        for character in ALPHABET:
            yield word[0..i] character word[i..n]

        # Deletion of a following character.
        yield word[0..i] word[i+1..n]

        # Substitution of the current character.
        for character in ALPHABET:
            yield word[0..i] character word[i+1..n]

        # Transposition of the previous and current characters.
        for character in ALPHABET:
            yield word[0..i-1] word[i+1] word[i] word[i+2..n]
```

2.3 Evaluation of words

The most important part of the algorithm is actually rating a word. For this, we will combine the channel model probability with the language model probability. The channel model probability $P(x|w)$ is obtained via $getCandidateWords()$ while the language model probability $P(w)$ depends on the frequency of prior words from a corpus.

The evaluation of words basically calculates $P(x|w) * P(w)$. $P(w)$ is calculated using $\frac{correctionCount}{errorsCount}$ where $correctionCount$ is the number of occurrences of the correction x given an error, and $errorsCount$ is the number of those errors.

Intuitively bigrams (or trigrams) are more precise than unigrams because these are more specific and use more details from the context. However, there are also less bigrams and even less trigrams to get information about. This could cause a skew in the results, so while it should be weighted more than unigrams, this parameter should not get too high.

3 Results and Explanation

4 Additional Resources

5 Statement of the Contributions

Peter wrote most of the Spell Checker implementationg and report.

6 Manual

7 References