Spell checker

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

April 4, 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 containing 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 these. Spelling suggestions are made on a word-by-word basis.

Finding possible *candidate words* that can substitute a possibly misspelled word requires a method to obtain such words. This is described in section 2.2.

The sentences resulting from a substitution of a single word will be evaluated as a whole. This evaluation is based on Kernighan's Noisy Channel Model [1] and is further described in section 2.3. The result from such an evaluation is a probability that a sentence is correct. Note that this number can only be used to compare similar sentences with the same number of words and with only one word different. Comparing a different number of words will most likely favor the shorter phrase as it has fewer words to affect the total probability (this condition does not occur as words can be modified, but not added or removed). Comparing completely different sentences can favor the sentence with errors (but popular n-grams) over one that uses less common words. This typically does not occur as the task assumes that words next to an error cannot be modified and at most two errors exist.

If there is a better sentence found after evaluating all possible substitutions for a word, a recursion will follow on the new sentence. Under the assumptions of the task, at most two recursions are needed since there are no more errors. Initially we have tried to continue with the best word. This greedy approach may however miss cases because n-grams may increase the probability of neighbors.

Finally the sentence which has the best overall rating will be returned. It is possible that the sentence is unmodified if no word was ranked better than the original sentence.

2.1 Overview

The indices of words that are surely correct are remembered (`readonlyWords` in the listing below). Once a rating is found which is higher than the initial one, the word is replaced.

When a word was modified, the hunt for more faulty words continues until all of them (at most `MAX_TYPOS` which is 2 by default) are found. Otherwise, it is assumed that the sentence has no more spelling errors.

Pseudo-code:

```
# Attempt to find a better sentence by modifying words.
def findBetterWord(words, modifiedIndices):
    # Rate the initial sentence to compare it with alternatives.
    bestRating = evaluateSentence(words)
    bestSentence = words

    # For each word in the sentence, try alternatives for a better rating.
    for wordIndex, word in enumerate(words):
        # Skip words if these were modified before or if they were
        # consecutive to a previous word (canModifyWord definition
        # is omitted for brevity).
        if not canModifyWord(modifiedIndices, wordIndex):
            continue

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

        # Iterates through the candidates and remember the modification
        # that gave the best word. Initially assume that no candidate
        # results in a better sentence.
        bestCandidate = None
        for candidate in candidates:
            # Calculates the likelihood for the candidate word. If it is
            # better than the current word, remember it.
            rating = evaluateSentence(candidate)
            if rating > bestRating
                bestCandidate = candidate
                bestRating = rating

        # If a better word is found, use it and try to find more errors.
        if bestCandidate is not None:
            words[index] = bestCandidate
            # Ensure that this word (and its context) cannot be changed
            # given the assumption that no consecutive words have an error.
            modifiedIndices.append(wordIndex)

            # If there are more typos, try to find them.
            if modifiedIndices.length < MAX_TYPOS:
                # Recurse with the modified sentence to find better words
                newSentence = findBetterWord(bestCandidate, modifiedIndices)
                if evaluateSentence(newSentence) > bestRating:
                    bestSentence = newSentence

            # Make words writable for further iterations.
            modifiedIndices.pop()
        # else no more suggestions, do not recurse and try the next word.

    # Finish with the best sentence (possibly unmodified).
    return bestSentence

# Split the input on spaces and find the best sentence.
```

```
words = input.split(' ')
bestSentence = findBetterWord(words, [])
```

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 taking the intersection with the dictionary. This does not scale and costs $O(27^n)$ memory and time.

Instead we opted for a smarter approach. As we know that the misspelled words have a Damerau-Levenshtein distance of 1, candidates are generated by cycling through each letter and applying an insertion, deletion, substitution or transposition. We also know that all corrected words are in the vocabulary, so other words can be dropped from the candidates list.

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

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

```
def getWordMutations(word):
    # Walk through the letters of a word and mutate those.
    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]

def getCandidateWords(word):
    # Initially there are no candidate words
    candidateWords []
    # Start with an empty map for words -> channel probability
    channelProbs = {}

    # Obtain all possible word mutations and pick valid ones.
    for mutation, candidateWord in getWordMutations(word):
        # Skip words not in vocabulary or unmodified words:
        if candidateWord not in vocabulary or word == candidateWord:
            continue

        # Word is in vocabulary, keep it.
        candidateWords.append(candidateWord)

        # Find the channel probability. As multiple mutations can
        # result in the same word, remember the modification that
        # gives the highest rating. Do not sum those as it can
        # result in a bias for very small words.
```

```
            channelProbs[candidateWord] = max(
                channelProbs[candidateWord],
                calculateChannelProbability(mutation)
            )

    return candidateWords
```

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 n-grams from a corpus. The combination gives the word score $P(x|w) * P(w)$.

Initially we have used interpolation with lambda conditionals on context [2] to combine the n-gram probabilities, but this gave bad results because it put too much favor on high probabilities for unigrams. So instead the language model probability is now the result from combining the n-gram probabilities by simple multiplication. The weight of bigrams is not changed now. Since there are fewer bigrams in the corpus, it will naturally be ranked higher than unigrams. Note that the algorithm can also be scaled to use trigrams, fourgrams, etc just by modifying the $NGRAM\_N$ parameter.

Due to the usage of n-grams, the score of a sentence as a whole must be considered. This is the case because the $n-1$ words before and after a word can form new n-grams which have different probabilities.

Finally, all word scores are combined by simple multiplication. For this reason, words which are certainly incorrect (not in the vocabulary) should not be ranked with 0, but with a very small number (such as 10^{-99}). This ensures that words with at least two errors have the chance to be seen as a better result.

Pseudo-code to summarize the evaluation:

```
def evaluateSentence(words, correctedWordIndex):
    p = 1
    # Simply multiple all word probabilities to obtain a sentence rating
    for wordIndex, word in enumerate(words):
        p *= getWordLikelihood(words, wordIndex, correctedWordIndex)
    return p

def getWordLikelihood(words, wordIndex, correctedWordIndex):
    word = words[wordIndex]

    if wordIndex == correctedWordIndex:
        # If this is the possibly misspelled word, use the channel
        # probability to compare multiple candidates.
        p = channelProbs[word]
    else:
        # For other words, assume that they are correct and use a
        # high probability
        p = LM_PROBABILITY_UNMODIFIED

    # Calculate P(w), or, how often does it occur in the corpus
    # Apply add-1 smoothing under the assumption that there are many
    # unigrams and this does not significantly affect the chance,
    # it just ensures that it is non-zero.
    o *= (getNGramCount(word) + 1) / (unigramCount + 1)

```

```
26      # multiply by n-gram probabilities
27      for n in range(1, NGRAM\_N):
28          # calculate P(word|prefix)
29          p *= ngramProbability(word, words[wordIndex-n..wordIndex])
30
31      return p
32
33  # Calculates P(word|prefix) = #word / #(prefiix word)
34  def ngramProbability(word, prefix):
35      a = getNGramCount(word)
36      b = getNGramCount(prefix " " word)
37
38      # apply smoothing, but add a smaller number because "b" is
39      # typically very small.
40      return (a + 0.001) / (b + 1)
```

3 Steps to reproduce

1. Obtain the source code of the SpellChecker project.
2. Run `ant run` from the project directory to compile and execute the program.
3. At the `run:` step, type the sentence to be corrected and press Enter. This should have at most `MAX_TYPOS` errors (defaulting to two, but changeable in the source code of `SpellCorrector.java`.
4. The corrected sentence is printed as `Answer: <corrected sentence>`.

A mode exists where you can keep inputting lines and for each of the line, an answer will be displayed. This can be achieved by setting the environment variable `NO_PEACH` prior to executing `ant run`.

4 Results and Explanation

The algorithm can be tuned with the following parameters:

NGRAM_N The largest n (as in n-gram) that should be checked for. This depends on the input file *samplecnt.txt* (which is also dependent on *samplevoc.txt*).

MAX_TYPOS This parameter is set to 2 for the task, but it could be increased for other situations. The worst-case running time will exponentially increase with this parameter.

The program was tested against the *test-sentences.txt* dataset for which it only failed to correct *the development of diabetes **u**s present in mice that **h**arry a transgen(**e**)* (errors or additions are emphasized).

With the initial ngram text file, it failed to correct *kind retards* to *kind regards*.

5 Statement of the Contributions

Peter wrote most of the Spell Checker implementation and report.

6 References

[1] Kernighan et al. *A Spelling Correction Program Based on a Noisy Channel Model* 1990

[2] Presentation on Interpolation techniques by Daniel Jurafsky https://class.coursera.org/nlp/lecture/19