Spelling Correction

Dennis van der Schagt (0814249)
Rob Wu (0787817)

April 1, 2015

Abstract

1 The algorithm

To find the best suggestion for a spelling correction, potential spelling corrections need to be generated and evaluated.

We assume that the input text contains at most two misspelled words, which are not next to each other. To find the most appropriate spelling correction, we first rate the original text, and then re-evaluate the text with one or two words modified. We apply the noisy channel model and assume that a word contains only one typographical mistake (insertion, deletion or substitution of one character, or transposition of two consecutive characters). For each input word, a list of such corrections is generated. These corrections are called *candidates*. The output where words may be replaced by candidates is called a *suggestion*.

To evaluate a suggestion, the likelihood of a candidate is used, as well as the likelihood of the bigrams of candidates within the suggestion. These likelihoods are combined (the exact method is explained later) and form the *combined likelihood*. After calculating the likelihoods for every suggestion, the one with the maximal *combined likelihood* is presented as the final suggestion for spelling correction.

1.1 General flow

A naive way to implement the previous concepts is to generate a list containing every possible suggestion, and then iterate over the list to find the maximal value. Although this method is simple, it is not the best choice because it does not scale well: the number of suggestions is of the order $O(\textit{[number of candidates per word]}^2 \times \textit{[number of words in the input text]})$. This excessive memory demand can be solved by immediately evaluating the suggestion when it is generated, and replacing the then-best suggestion with the just-evaluated suggestion when its combined likelihood is higher.

Suggestions are generated as follows. First, we split the input text into a list of words. Then, for each word, we generate and store a list of candidates (as elaborated in section 1.2). This step requires $O(\textit{[number of candidates per word]} \times \textit{[number of words in the input text]})$ memory. If peak memory usage is a concern, this step can be omitted at the cost of speed (time) in the next step.

The actual suggestion generation is done by using the assumption that at most two non-consecutive errors occur in the input text. For every word, we iterate over the list of candidates and evaluate the suggestion that is equal to the original text, but with the given word replaced by a candidate. Similar logic is used to find and evaluate the second spelling error. The original word is also rated, since it is also possible that the input is correct, i.e. contains zero spelling errors.

The pseudo-code of the general structure is displayed below.

```
words = input.split(' ')
candidates = []
for word in words:
   # getCandidateWords generates a set of candidates for each word; its
   # exact implementation is explained later.
   candidates.append(getCandidateWords(word))

```

```
# The program will not output a suggestion if a word is not in the dictionary.
bestSuggestion = input if input in dictionary  else  ""

for error1 in range( 0, len(words) ):
    for candidate1 in candidates:
        # Evaluate the suggestion, with the word at position |error1| replaced
        # by |candidate1|.
        evaluate(error1, candidate1)

        # Try to find the next error. The next word after a corrected word is
        # assumed to be correct, so skip the word at the next position and
        # start iterating over the words from the next position, i.e. at
        # position |error1| + 2.
        for error2 in range( error1 + 2, len(words) ):

            for candidate2 in candidates:
                # Evaluate the suggestion, with the word at position |error2|
                # replaced with |candidate2|. Note that the word at position
                # |error1| is still |candidate1|.
                evaluate(error2, candidate2)

        # restore the original word at position |error2|, so that the next
        # iteration over |error1| will use the original word.
        restore(error2)

    # restore the original word at position |error1|
    restore(error1)

# At this point, every possible combination of candidates in the suggestion has
# been checked by the evaluate function, and the best suggestion has been
# selected. Show this suggestion.
print(bestSuggestion)
```

1.2 Generating candidates

As said before, we assume that a candidate is a word in the dictionary, and differs from the original word by at most one modification, which is either a character insertion/deletion/substitution or a transposition of consecutive characters. Each of these mutations are defined in terms of a range (start + end position) and replacement. A precise definition and explanation of the parameters for these operations (given a word) is shown in the pseudo-code below.

```
# Insert space (word separator) around word to simplify the implementation of
# the insertion/deletion logic (without these, we have to add and handle extra
# bound checks for the first and last character).
word = ' ' + word + ' '

# Insertion
# Start at 1 because insertion looks at the previous letter. It is safe to skip
# the first character because we have prepended a space before the word.
for i in range(1, len(word)):
    for newLetter in ALPHABET:
        collect(i - 1, i, word[i - 1] + newLetter)

# Deletion
# Exclude the first and last characters (spaces) because the body looks at the
# previous and next letter.
for i in range(1, len(word) - 1):
    collect(i - 1, i + 1, word[i - 1])

# Transposition
# Exclude the first character, because we do not want to break the word in two
# words (this would happen when the leading space is swapped with the second
# character). The last character is skipped for the same reason, and the
#
for i in range(1, len(word) - 2):
    collect(i, i + 2, word[i + 1] + word[i])

# Substitution
# Skip the first and last character (space) because replacing them with letters
# would be equivalent to adding letters before/after a word. This is already
# covered by the insertion logic.
for i in range(1, len(word) - 1):
    for newLetter in ALPHABET:
        collect(i, i + 1, newLetter)
```

In the pseudo-code, the `collect` function is invoked several times. This method generates a potential candidate (by replacing the characters in the given range by the given replacement) and checks whether the word is in the dictionary. If the word exists, the probability that the edit is correct is calculated and stored together with the candidate (this probability is used many times during the evaluation of suggestions, so it makes sense to calculate and store it in advance). This probability is also known as ”channel model probability” within the noisy channel framework, and is a product of two factors: the *prior* and the *edit probability*. The prior is simply the frequency of the word, and the edit probability is the smoothened edit frequency. This is derived from the confusion matrix from Kernighan's paper [1].

Some candidates can be generated in multiple ways, e.g. ”ass” can become ”as” by removing any of the two ”s” characters. This possibility is accounted for by summing the calculated probabilities (up to a maximum of 1).

1.3 Evaluating word suggestions

The previous sections explained how suggestions and candidates are generated, together with the channel model probability of each candidate. This section brings the pieces together.

We define the probability that a word at a certain position within the input is correct as the product of the *channel probability* and the bigram frequency (with the previous word). The latter is the quotient with the smoothened bigram count of the previous word together with the given word as numerator and the smoothened unigram count of the previous word as denominator. The first word does not have a previous word, so we will only use the channel probability in that case.

After determining the probability that a word within the suggestion is correct, we can take the product of these probabilities to get the probability that the suggestion is correct. Then the best choice is the probability that is closest to 1. An issue with this approach is that most application runtimes cannot store arbitrary-precision floating point numbers. If the input has many words and/or contains many low-frequency words and/or low-frequency edits, then the product of these small probabilities cannot be represented as a (small) floating-point number any more, and all calculated probabilities would effectively be zero.

1.3.1 Enhancing probability: likelihood

To solve the problem of representability of floating-point numbers, we apply a clever trick. The probability is a number in the range $[0, 1]$, but the application does not need the number to be constrained to this format. The only requirement is that the value needs to be monotonous and be bounded at at least one boundary.

So, we normalize all numbers by taking a logarithm of the probabilities and summing the result. A logarithm on domain $[0, 1]$ maps to the range $[-\infty, 0]$, where $-\infty$ means "improbable" and 0 "very likely". The number is by definition no longer a probability, so we call it "likelihood" to avoid any confusion.
Although this solution is not perfect (adding huge negative numbers will eventually result in negative infinity), it is an improvement over the previous situation, because the method does not suffer from the problem of loosing precision.

With this normalization, another optimization can be done. Instead of summing all numbers, we can re-use likelihood sums, by subtracting the original likelihood and adding the likelihood of the changed word. Then the likelihood calculation is no longer linear in terms of the word count, but constant!

2 Miscellaneous modifications to the original files

We were given a set of Java files to develop a spell checker. These have changed significantly. Obvious changes to unimplemented methods are not mentioned, the other changes are listed below.

- The input reader of the confusion matrix was replaced, and uses regular expressions to parse the input. This ensures that the input format complies with the documented expected format.
- The `getCharsCount` method was added to `ConfusionMatrixReader.java`. This method returns the number of occurrences of errors in the data that was used to create the confusion matrix.
- The `CorpusReader` class was modified so that the corpus size of the training data can be requested. This is simply the sum of all unigram counts, and used to calculate word frequency.
- The `calculateChannel` method of `SpellCorrector.java` has been removed. The probability is calculated in a private inner class within the same file, `IntermediateAnswer`. This class maintains the state of the spelling corrector, and decouples the state of the `correctPhrase` from the `SpellCorrector` class. This makes it easier to extend the class if wanted, e.g. to add support for multi-threading.
- The `getCandidateWords` method's signature of `SpellCorrector.java` was changed to match the fact that it returns a Map instead of a Set. It was also declared private, since it is merely an implementation detail of the class, and not intended to be used outside the class.
- `confusion_matrix.txt` was replaced with the confusion matrix from Kernighan [1].

3 Results

The spell checker can be configured by 3 parameters. The quality of the spell checker strongly depends on those parameters. Optimal values for those parameters were determined by running the spell checker on our own training data. The 3 parameters are called: probability-no-edit-needed, edit-probability-k-smoothing, and bigram-smoothing. Following is a description of those parameters and their influence on the output.

probability-no-edit-needed Every correction candidate to a word gets a probability. To be able to compare candidate probabilities with the probability of the original word, we have to assign an edit probability to the original word. We choose a value of 0.95 which means that we assume that 1 out of 200 words contains a spelling error. By choosing such a high probability we make sure that only words that are significantly better will replace the original word.

edit-probability-k-smoothing The confusion matrix contains all probable alterations that correct a word. Nevertheless, words sometimes contain spelling errors that are not in the confusion matrix. Initially we just didn't correct those errors but we discovered that by simply adding a constant k to all results the confusion matrix got smoothed enough to solve those errors. We now use a safe value of 1 instead of add-k smoothing. Increasing this value adds more smoothing but it also lowers the influence of the actual confusion matrix.

bigram-smoothing Many 2-word combinations of the correct sentences do not occur as bigrams in the N-gram input file. Many sentences were discarded because of this, even if the individual words had a high probability and were actually correct. To fix this, smoothing of the bigram data is needed. We first smoothed by adding 1 to every bigram count. This turned out to be too high. In the end we settled with add-0.01 smoothing. This makes sure that the bigram information stays relevant while still allowing new bigrams to be included in the suggested sentences.

To show the problem with add-1 smoothing, take the bigrams "is present" and "us present", which have respective counts 7 and 0. The unigrams "is" and "us" have respective counts 12157 and 628. We calculate a bigram's probabilility by dividing the smoothed bigram count by the smoothed count of the left word. This would assign a probability of 8/12158 to "is present" and a 1/628 probability to "us present". Here "us present" clearly has a much higher probability while, intuitively, "is present" should have a higher probability. Add-0.01 smoothing clearly performs better here with probabilities of 7.01/12157.01 and 0.01/628.01 for "is present" and "us present" respectively.

We used Peach's tests to assess the quality of our spell checker. Of those tests our spell checker passes 30 out of 33.

4 Reproducing the results

Following are the steps required to reproduce our results:

- Download the Netbeans project of our Spelling correction program from Peach.
- Compile the project, e.g. using ANT COMPILE.
- Run the project, e.g. using ANT RUN.
- Type a sentence consisting of lowercase Latin letters and spaces, with at most two spelling errors and press enter.
- Read the result. The corrected sentence will be displayed as "Answer: [corrected sentence].

5 Contributions

Dennis and Rob contributed equally to this assignment.

References

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