CS637 - Natural Language Processing

Programming Assignment 1 – Spell Checker

Statistical Spell Checker

Gabriel Dulac-Arnold <gabe@squirrelsoup.net> [CS09F004]
Johannes H. Jensen <johannj@stud.ntnu.no> [CS09F005]

We based our spellchecker on the techniques presented in the paper:

Kernighan, M. D., Church, K. W. and Gale, W. A. (1990), "***A Spelling Correction Program Based on a Noisy Channel Model***", Proceedings of COLING '90, Helsinki.

For better accuracy we used bigrams for context-sensitive correction.

Our spellchecker is written in python[1] and we use the Natural Language Toolkit[2] (nltk) extensively throughout the program.

The spellchecker only attempts to correct non-words. Words that appear in the dictionary are assumed to be correct and ignored. The dictionary is a mix of the nltk dictionary and the local UNIX dictionary on the machine running the code. This was necessary because each dictionary was missing some particular sets of words (plurals, past tenses, etc).

For each encountered non-word, the spellchecker constructs a list of candidate words based on edit distances as described in the paper. Only candidate words found in the dictionary or corpus are considered. (We chose to include words from the corpus because we were concerned about the accuracy of our dictionary.) In the case where the program is unable to find any candidate words within edit distance 1 it proceeds to try distance 2. Finally it sorts the list of candidate words based on the *word*, *edit* and *context probability*.

The word probability, P(c), is calculated by counting the number of times the given word occurs in the Brown and Reuters corpora. It is not normalized to [0,1]. Since we're using probabilities to sort, normalizing to [0,1] is not necessary, as everything is relative.

To be able to complete the noisy channel method, we needed to take advantage of P(c|t), or the probability of a certain correction occuring considering the error that led to it. Instead of taking the tables from the paper, we decided to attempt to build them ourselves. Lacking the time and expertise to implement Good - Turing approximation on the probabilities, they were normalized with the '+1' method, and left as-is. Probabilities were therefore calculated by taking the number of times a certain typo appeared in the corpus, and normalizing it by the

1 http://www.python.org
2 http://www.nltk.org

method described in the paper.

Instead of using the boot-strapping method described in the paper, we resorted to a simpler single iteration method. We go through a certain corpus (in this case the Reuters corpus, as it is a typed corpus) and check for non-dictionary words. For every non-dictionary word, we create a set of noisy-channel candidates. We use the probability of the word appearing in the corpus to pick the best candidate, and store the typographical error that leads to that word in our confusion matrix. We recognize that using P(c) to help in computing P(c|t) is potentially biasing the results of our confusion matrix, but it seems to provide good results.

For context probability we used bigram counts from the same corpora. We only consider the two words word preceding and following the word in question. We can therefore use the frequency of these context bigrams to judge each candidate.

Finally the list of candidate words is sorted by P(c) * P(c|t) * P(c|corpus), and the most likely words are picked.