CS276 Programming Assignment 2

Raj Bandyopadhyay, Rafael Guerrero (rajb2, rplatero)

May 1st, 2013

Table of Contents

Introduction

In this assignment we have implemented a spelling corrector in Python, using a probabilistic approach to model the noisy channel. This report describes the different design options and trade-offs we have considered during the implementation of every module of the system i.e. model generation, candidate generation, filtering and scoring; as well as the results obtained.

Model Generation

Language Model

We scan the corpus files and generate these dictionaries:

- word_language_model: how probable is each word of the corpus
- biword_language_model: how probable is each biword of the corpus
- word_counter: how frequent is each word of the corpus
- biword_counter: how frequent is each biword of the corpus

We also generate word indexes, bi-character and tri-character indexes that will be used later on candidate generation.

Noisy-Channel Model

For the noisy-channel model we have followed the approach described by *Kernighan, Church and*

Gale in 'A Spelling Correction Program Based On Noisy Channel Model'

The model is represented by four confusion matrices (deletion, substitution, transposition and insertion), one unichar counter and one bichar counter. The entries of the confusion matrices are defined as:

- del[(x,y)] = count(xy typed as x)
- sub[(x,y)] = count(y typed as x)
- tra[(x,y)] = count(xy typed as yx)
- ins[(x,y)] = count(x typed as xy)

Candidate Generation

The approach we have followed for correcting a potential misspelled query phrase is as follows:

1. *Split the query phrase into bi-words and identify bi-words with one potential misspelled word* : A bi-word is considered as potentially misspelled when the bi-word language model shows no hits for it. Likewise a word is considered potentially misspelled when the word index shows no hits for it
2. *Generate candidates for the potential misspelled words using bigram and trigram character indices*: We are using an n-gram index approach; this guarantees that all candidates being considered for a given word, are also part of the dictionary, which reduces the size of the candidate set. Specifically:
    - When the word is 10 characters or less we use a bigram index.
    - When the word is greater than 10 characters, we use a trigram index.
    - If the word occurs in the dictionary, it is always included as a candidate

Optimizations

While the candidates are been selected using the bi-char or tri-char index we applied two more heuristics to prune the final set of candidates:

1. Assume that the first character of each query word is typed correctly
2. Jaccard Coefficient with cutoff threshold: If the Jaccard Coefficient between the word and the candidate is smaller or equal than the cutoff threshold (0.2 by default) then the candidate is discarded. If the potential misspelled word has more than 10 characters then we force the cutoff threshold to be at least 0.5. This is because for smaller words, a single edit can influence Jaccard distance drastically compared to longer words.
3. Edit Distance: we discard any candidate at edit distance greater than 2.

Handling spaces

We assume that every word has a potential missing space in it, and handle it as follows:

- Generate each possible split of the word.
- For each split
    - If one of the two parts occurs in the dictionary, add it as a candidate and don't generate any further candidates for that part. Generate candidates as before for the other part, but restrict them to edit distance 1 since the space counts as one edit.
    - If neither of the two parts occurs in the dictionary, skip candidate generation for this split and don't add either of them as candidates.
    - If both parts of the split occur in the dictionary, just add them as candidates and don't generate any more candidates.

Candidate Scoring

After generating a reduced list of potential candidates for each word, we rank them using a probabilistic approach for the Noisy Channel model. When we rank the bi-gram (R,Q) where Q is a candidate for R, we do the following calculation

rank(Q,R) = P(Q|R) = P(R|Q) * P(Q)

P(Q) comes from our language model, while P(R|Q) is modeled using one of the two options used to model the noisy channel P(w2|w21):

1. Uniform Noisy Channel

We model the cost of the edit operations as:

P(R|Q) = (cost)^d when Q != R
= 0.9 (default) when Q == R

Where d is the edit distance between R and Q, and cost is the uniform edit cost (set to 0.01, which was discovered through experimentation).

2. Empirical Noisy Channel

The cost of each edit operation (deletion, substitution, transposition and insertion) has been calculated following the approach described by *Kernighan et al.*

P(R|Q) = P_oper(R|Q) when Q != R
= 0.9 (default) when Q == R

where:

- edit_distance(R|Q) = 1
- one_edit_distance_operation(R|Q) = 'oper'
- P_oper(R|Q) = cost of changing Q into R as defined by the Noisy Channel Model

There may be candidates Q that are at edit distance 2 from R. This means that the noisy channel transformation from Q into R is: Q -> edit_operation_1 -> R' -> edit_operation_2 -> R. We have modelled this 2 edit-transformation as:

P(R|Q) = P_oper(R|Q) * uniform_edit_operation_cost

where:

- Popt(R|Q) models the cost of the first edit operation, using the empirical noisy model
- uniform_edit_operation_cost models the cost of the second edit operation

Using a constant for the second edit operation allowed us to avoid the computational burden of calculating the exact edit operation at the expenses of a less precise estimated cost.

Query Scoring

When best candidates have been selected we run a cross-product for the different bigrams involved,

generating a set of potential query candidates. We score each query phrase using the pre-computed Language Model e.g. if (w1,w2,w3) is a query phrase candidate we score it using:

$P(w1,w2,w3) = P(w1)*P(w2|w1)*P(w3|w2)$

When we hit a biword that is not part of our corpus we use n-gram interpolation with lambda = 0.2.

Combining candidates for multiple words

We limit the total number of candidates for a query phrase to M=500. and the number of candidates for a single word to M/N, where N is the number of words in the phrase. The candidates for each word in the query, as well as the entire query at the end are ranked using the scoring technique described above and the designated number of candidates is returned.

Extra Credit (Stupid Backoff)

In stupid backoff, we use just the weighted prior probability of a single word instead of the bi-word probability for bi-words that don't occur in the dictionary. i.e.

$P(w2|w1) = count(w1,w2)/count(w1)$ if (w1,w2) exists in the language model
$= alpha*count(w2)/T$ otherwise

As suggested in literature, alpha is set to 0.4.

Parameter Tuning

We attempted to manually tune many parameters: e.g. the lambda for interpolation, the uniform edit cost, the prior probability P(R|R) and so on. The following results show the numbers we obtained for parameters that made a significant difference through tuning.

1) lambda for interpolation

An overly small value of lambda drastically reduces accuracy, but beyond 0.2, it makes little difference. Hence, we stuck with 0.2.

lambda for interpolation	Accuracy (% correct on dev set)
0.1	50.8
0.2	78.9
0.4	78.6

2) uniform edit cost

0.01 was the optimal value we found for this cost.

uniform edit cost	Accuracy (% correct on dev set)
0.003	77.3
0.01	78.9
0.03	77.8
0.1	75.8

3) Jaccard coefficient cutoff

There is a tradeoff between accuracy and runtime. For small Jaccard cutoffs, the accuracy is high, but runtime increases due to large numbers of candidates. For large cutoffs, too few candidates are generated. We decided on 0.2 as a default cutoff.

jaccard cutoff	Accuracy (% correct on dev set)	Runtime (secs)
0.1	78.9	350
0.2	78.9	257
0.3	78	222
0.4	74.7	206