CS276: Information Retrieval and Web Search **Spring 2013**

Programming Assignment 2

Botao Hu (botaohu), Jiayuan Ma (jiayuanm) Completed Date: *May 4, 2013*

1 Models

We model the probabilty of a correct query Q given the typo query W as $P(Q|W) \propto P(Q)P(W|Q)$. We call $P(Q)$ the language model and $P(W|Q)$ the edit channel model.

1.1 Language Model

We build our language model using the provided corpus, and count all the unigrams and bigrams that appeared in the corpus. Using these counts, we can estimate the parameters in our language model by calculating

$$P_{mle}(w_i|w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_i)} \qquad P_{mle}(w_i) = \frac{count(w_i)}{\text{total number of terms}} \tag{1}$$

To deal with the data sparsity problem, we use N-gram interpolation to obtain the final conditional probability in our language model.

$$P_{int}(w2|w1) = \lambda P_{mle}(w2) + (1-\lambda)P_{mle}(w2|w1) \tag{2}$$

In our experiment, we use the default $\lambda = 0.2$. In addition, we use all the unigram in the corpus to build a bigram-character index to support efficient candidate word generation.

1.2 Edit Channel Model

We have two versions of the edit channel model: one is the uniform model, i.e. the error rate is a const paramemter that we set to 0.05.

The other one is the empirical model. Based on training data of 1-distance edit pairs, we can easily model edit cost of each word using singleton potential in our Markov chain model. The edit cost is given as follows.

$$P(x|w) = \begin{cases} \frac{del[w_{i-1},w_i]}{count[w_{i-1},w_i]} & \text{if deletion} \\ \frac{ins[w_{i-1},x_i]}{count[w_{i-1}]} & \text{if insertion} \\ \frac{sub[w_i,x_i]}{count[w_i]} & \text{if substitution} \\ \frac{trans[w_i,w_{i+1}]}{count[w_i,w_{i+1}]} & \text{if transposition} \end{cases} \tag{3}$$

where w is the correct word and x is the typo.

All these count statistics come from the provided training file. For the unseen edits, we use Laplace smoothing to deal with the data sparsity issue.

2 Candidate Word Generation

2.1 Bigram Indexing

We use bigram index to efficiently compute Jaccard distance between words, threshold Jaccard distances to obtain a list of candidates, and filter the candidates using the exact Damerau-Levenshtein distance. We tune the threshold parameter of Jaccard distance to be 0.5 to balance between efficiency and accuracy.

2.2 Levenshtein Automata (Extra credit)

We observe that Jaccard distance is not a good approximation of Damerau-Levenshtein distance, especially when the words are short and the correct edits are transpositions. To fix this problem, we implement the Levenshtein Automata to support *fast* and *accurate* candidate generation for words.

We use a Levenshtein transducer that uses a finite state automata for fuzzy matching of words. Based the experimental implementation provided in the assignment, we generalized it to perform the transposition operation by add an extra intermediate state $(i+1, e, 1)$ to respresent the transportation of two characters $w_i w_{i+1}$, i.e., we create the path $(i, e)\underbrace{\rightarrow}_{w_{i+1}}(i+1, e, 1)\underbrace{\rightarrow}_{w_i}(i+2, e+1)$ where state (i, e) respresent we edit the first i characters of the correct string with e edits.

Listing 1: Exerpt from automata.py

```
# Deletion
nfa.add_transition((i, e), NFA.ANY, (i, e + 1))
# Insertion
nfa.add_transition((i, e), NFA.EPSILON, (i + 1, e + 1))
# Substitution
nfa.add_transition((i, e), NFA.ANY, (i + 1, e + 1))
# Transportation
if i < len(term) - 1:
  nfa.add_transition((i, e), term[i + 1], (i + 1, e, 1))
  nfa.add_transition((i + 1, e, 1), c, (i + 2, e + 1))
```

2.3 Combined and Spilted Words

The space may be inserted in words. So we try to combine the adjacent two words into one candidate.

In the other hand, the space may be removed between two adjacent words. We try to enumerate all positions of one typo word to spilt it into two words as one candicate (w_1, w_2). Note that we have to add the language model $P(w_2|w_1)$ in the final model in this case.

2.4 Ranking and Filtering Candidate Words

Given a candidate word c and a typo word w, we can use the language model and edit channel model to evaluate a score that is the probabilty of changing c to w, i.e., $P(w|c)P(c)$. We sort all candidates by the score and only remains the first $cand_k = 20$ items to be candidates of the typo word w.

3 Candidate Query Scoring Using Markov-Chain Inference

Given the proposed candidate query, we compute our language model using

$$P(w_1, w_2, \ldots, w_n) = P(w_1)P(w_2|w_1)P(w_3|w_2) \cdot P(w_n|w_{n-1}) \tag{4}$$

which is essentially a Markov-Chain model. We can minimize the negative log-likelihood

$$-\log P(w_1, w_2, \ldots, w_n) = -\log P(w_1) - \sum_{i=2}^{n} \log P(w_i|w_{i-1}) \tag{5}$$

using dynamic programming (Viterbi algorithm) on markov chain. Based on this approach, we can find a global minimum without scoring all the possible candidate queries. So the algorithm complexity is linear to the number of words in the query and the square of the number of candidates of one word. Because we only remains the top $cand_k$ candidates in each step, the speed of inference is fast.

We think that this dynamic programming implementation is one of the most important contribution of our works.

4 Experiments & Conclusion

The uniform model achieves 381/455 = 0.83 accuracy on the provided development queries. The emprical model achieves 389/455 = 0.85 accuracy on the provided development queries. Our ultimate model (Empirical Edit Cost, Levenshtein Automata with Dynamic Programming) achieves 409/455 = 0.90 accuracy on the provided development queries.