CS 276 / LING 286 Spring 2015

Programming Assignment 2

Due: Thursday, April 30, 2015, 11.59 PM PDT

1 Overview

The aim of this assignment is to build a Probabilistic Spell Corrector. The Spell Correctors we build will have 4 distinct parts - the channel model, the language model, the candidate generator and the candidate scorer.
The channel model basically learns a model of the errors that occur in a query - the probability of a character getting inserted into a query, deleted from a query, substituted by another character or transposed with a neighbouring character. We will use two different models which are described in more detail in the task description below.
The language model just keeps track of the probability of occurrence of unigrams and bigrams in the corpus. This could later be used to calculate the feasibility of candidate queries.
The candidate generator generates candidate queries given the query submitted by the user. For eg. Given a query 'Brute Willis', possible candidates are 'Bruce Willis', 'Brute Willis' and many others. Since most queries do not have more than 2 errors, you should try to build a candidate generator that generates atleast all 'reasonable' candidates within 2-edit distances from the input query (you certainly shouldn't stop at 2 if it's possible to build a better candidate generator). A brute force generator that builds all candidates within 2 edit distances wouldn't work since the number of candidates would be enormous. Try cooking up some pruning techniques.
The candidate scorer scores all the generated candidates using the language and the channel models and chooses the best candidate as the intended query.

1.1 Data

Original training corpus: 98,998 documents crawled from the stanford.edu domain. Block structure can be found in **data/corpus/**.

Language modeling corpus: 819,722 pairs of misspelled and matching correct queries. Each pair is within one edit distance apart. Misspelled queries and correct versions (tab-separated) in **data/edit1s.txt**.

Development dataset: 455 training queries with matching correct queries.
(Possibly) misspelled queries are in **data/queries.txt**
Correct queries are in **data/gold.txt**.
Google spell check's results are in **data/google.txt**.

1.2 Basic Theory

If a user types in a (possibly corrupted) query R, we want to find the query Q that the user intended to type in. To do this, we are going to use probabilities to find the most likely query that the user meant to enter.

The problem becomes finding the query that maximizes the conditional probability $P(Q|R)$, the probability that the user meant to search for Q when enterring R. Note that often $Q = R$, in which case there is no misspelling.

To estimate this probability, we are going to use Bayes' theorem. Note that since we are trying to maximize this probability with respect to a choice of the query Q, the probability $P(R)$ of seeing what the user actually entered will not vary, so we can disregard it. Thus, we have:

$$P(Q|R) \propto P(R|Q)P(Q)$$

The probability of seeing the query $P(Q)$ will be derived from a *language model* that we will estimate from our training corpus. At the same time, the probability of the user entering a particular sequence R, given that he meant to enter the query Q, $P(R|Q)$ is estimated from the *noisy channel model* of possible edits. As discussed in class, we will begin with a basic noisy channel model considering Damerau-Levenshtein distance with uniform edit probabilities. Later, we will consider more complicated models with non-uniform edit probabilities.

2 Spell Corrector Model with Uniform Edit Cost (Task 1 – 55%)

2.1 Language Models

The first step to building a language model is to estimate $P(Q)$ from the training corpus. The probability for a given sequence of terms is computed as follows:

$$P(w_1, w_2, ..., w_n) = P(w_1)P(w_2|w_1)P(w_3|w_2)...P(w_n|w_{n-1})$$

Here, the first term is the unigram probability, while the remaining ones are bigram probabilities. We suggest that you perform these calculations in log space to avoid numerical underflow, and recall $\log(a * b) = \log(a) + \log(b)$. Since the final operation is a monotonic function, $argmax_Q(..)$, you can make the comparison in log space.

2.1.1 Calculating Probabilities

This is a simple matter counting exactly how many bigrams and unigrams appear throughou the corpus. We will be using maximum likelihood estimates (MLEs) for both probabilities, outlined as follows.

Bigrams: $P_{MLE}(w_2|w_1) = \frac{P(w_1,w_2)}{P(w_1)} = \frac{count(w_1,w_2)}{count(w_1)}$.

Unigrams: $P_{MLE}(w_1) = \frac{count(w_1)}{T}$.

Here, $count(w_1, w_2)$ is the number of times w_2 immediately follows w_1 in the corpus, and $count(w_1)$ is the number of occurrences of w_1. T is the total number of tokens in the training corpus.

2.1.2 Smoothing

The unigram probabilities model will also serve as a dictionary, since we are making the assumption that our query language is derived from our document corpus. As a result, we do not need to perform Laplace add-one smoothing on our probabilities, since our candidates will be drawn from this very vocabulary.

To take into account the data sparsity problem where some bigrams that appear in the queries might not be in our training corpus, we interpolate unigram probabilities with the bigram probabilities to get our final interpolated conditional probabilities.

$$P_{int}(w_2|w_1) = \lambda P_{mle}(w_2) + (1 - \lambda)P_{mle}(w_2|w_1)$$

Try setting λ to a small value in the beginning, say 0.1, and later experiment with varying this parameter to see if you can get better correction accuracies on the development dataset. However, be careful not to overfit your development dataset. It might be useful to reserve a small portion of your development data to tune the parameters.

Note that, ngram interpolation is just one way of doing smoothing. See this link `http://www.stanford.edu/class/cs124/lec/languagemodeling.pdf`, starting from slide 47, for more information about other smoothing techniques.

2.2 Noisy Channel Model - Uniform Cost Edit Distance

Noisy channel model is a more challenging part in spelling correction, which models $P(R|Q)$, the probability that a user would enter a query R when he or she intended to enter Q, i.e. the noise in the communication of the user's intent to the query. To compute the noisy channel probability, we quantify the difference between the candidate query

Q and the actual input R using the Damerau-Levenshtein distance. In the Damerau-Levenshtein distance, the atomic operators defined are *insertion*, *deletion*, *substitution* and *transposition*.

The uniform cost edit distance model simplifies the computation of the noisy channel probability by assuming that any single edit in the Damerau-Levenshtein distance is equally likely, i.e., having the same probability. Again, try different values for that uniform probability, but in the beginning 0.01∼0.10 is appropriate.

A factor to consider in this model is that our input query, R, may indeed be the right one in a majority of cases. Experiment with the different assignments to $P(R|Q)$ in the case where $R = Q$, but a reasonable range is 0.90∼0.95.

Since the candidate generation component (next section) takes care of measuring the edit distance, in this part, all you have to do is calculate the probability of seeing R given the edit distance from Q.

2.3 Candidate Generation

Since we know that more than 97% of spelling errors are found within an edit distance of two from the input query R, we encourage you to consider those candidates. Note that this is the approach taken by Peter Norvig in his essay `http://norvig.com/spell-correct.html` on spelling correction. We can do better than this simple approach by aggressively narrowing down the search space while generating candidates.

A common misspelling is when words are accidentally joined together, or split, by a space. To deal with these cases, a simple approach would be: split or combine one or two words at first; Then for each word in the query, generate a set of words that are 1-edit distance away and make sure that the generated word is in dictionary. At last, do a Cartesian product of two candidate word sets at a time to get query candidates. Notice here we are making the assumption that all the words in a valid candidate query are found in our dictionary (as mentioned in the unigrams probability section).

The method mentioned above is a basic way of generating candidates. We can do some tricks to improve candidate generation efficiency. For instance, we can add space into alphabet so that 1-edit distance will include splitting a word into two words and combining two words. Please describe any generation optimizations you made in the write up. Solutions that both exhaustively generate and score all possible one and two edit distance candidates will not get full credit.

2.4 Candidate Scoring

When putting the probabilities of the language model and the channel model together to score the candidates (remember to use log space), we can use a parameter to relatively weight the different models.

$$P(Q|R) \propto P(R|Q)P(Q)^{\mu}$$

At first start with $\mu = 1$, and later, experiment with different values of μ to see which one gives you the best spelling correction accuracy. Again, be careful not to overfit your development dataset. It might be useful to reserve a small portion of your development data to tune the parameters.

3 Spell Corrector Model with Empirical Edit Cost (Task 2 – 25%)

After having gotten the spelling corrector working with the basic noisy channel version, turn your attention to a more principled approach to the edit probabilities. Here, we learn these edit probabilities from the provided data in **data/edit1s.txt**.

You are given a list of query pairs that are a *single* edit distance away from each other. You can devise a simple algorithm to detect what specific edit has been made to the queries and learn the probability of that specific edit taking place. The edit probability calculation is described in more details in the lecture handout `http://www.stanford.edu/class/cs276/handouts/spelling.pdf`.

An example of the information your model will learn: the probability of the letter e being substituted by the letter a for a correction,

$$P(sub[a,e]) = \frac{count(sub[a,e])}{count(e)}.$$

A few things to note, the insertion and deletion operator probabilities are conditioned on the character before the character being operated on. To account for an inevitable data sparsity problem, you need to use Laplace add-one smoothing for the error probabilities, as described in the lecture handout.

4 Running Steps

First, we call **buildmodels.sh** to build the models for the corrector. You should save the models in the most straightforward manner possible, so they can be read in during the next step. The third argument is optional. If you implement extra credit, run with "extra" as the third argument. Code should run when we call:

./buildmodels.sh <training corpus dir> <training edit1s file> <extra>(optional)

Then, we will call **runcorrector.sh**, the code should accept two arguments specifying the type of noisy channel probabilities to use and the file containing the queries to be corrected (one per line). This should output the "correct" queries one per line, in the same order as the test file. The third and fourth arguments are optional. If you want to run with extra credit, put "extra" as the third argument. If you want to compare with gold corrected queries, then add the gold file as another argument. Use the development set **data/queries.txt** as the queries file for your experiments. **Remember, it is an honour code violation to knowingly take a look at the test set**. Code should run when we call:

./runcorrector.sh <uniform | empirical > <queries file> <extra>(optional) <gold file>(optional)

You can then compare the result you get with the correct results in **data/gold.txt** or with the results from running Googles spell check on the queries in **data/google.txt**. As you can see, even Google does not get all of the queries right, can you do better?

5 Extra Credit

We have a couple of ideas here, but really any extensions that go above and beyond what is outlined here will be considered. Be sure to include a description in your assignment write-up.

Expanded edit model: we can take into account common spelling errors as a single edit in our model, for example the substitution **al** → **le**. Can you incorporate them into the channel model probabilities?

Alternate Smoothing: try other smoothing algorithm to better capture probabilities in the training corpus.

K-gram index To deal with unseen words, it is possible to develop a measure for the probability of that word being spelled correctly by developing a character k-gram index over your corpus. For example, a **q** not followed by a **u** should lead to a low probability. This index can also assist in *much* more efficient candidate generation.

Levenshtein Automata You can do *even* faster candidate generation using a Levenshtein transducer `http://en.wikipedia.org/wiki/Levenshtein_transducer` that uses a finite state automata for fuzzy matching of words. There is an experimental implementation `http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata` in Python `https://gist.github.com/491973`, but it needs to be generalized to perform the transposition operation too.

We will give more extra credit for best spell corrector in the entire class, which is based on both accuracy and running time computed on our hidden test data. 10% for the top 5 systems and 5% for the next 15 systems.

6 Grading

We will be evaluating your performance on a distinct test dataset, which will have a mixture of queries taken from realistic data.

Task 1 - 55% 55% for a correctly implemented solution with higher than 80% accuracy. We will give partial credit proportional to how well your solution does compared to that. Not linearly proportional though, as that would discredit any attempts to make the final small improvements. You will be penalized 10% if the running time is excessively long beyond the norm and 5% if the memory used is excessively large beyond the norm.

Task 2 - 25% In this case, 25% of the grade for accuracy that is better than 85%. Also partial credit will be awarded proportional to how well your model does. A correct implementation using the baseline method in the handling can get 89% for the empirical cost model.

Report - 20% Be sure to document any design decisions you made, and explain some rationale behind them. We give 5% for system design, 5% for discussing methods being used such as smoothing etc, 5% for discussing optimizations used for candidate generation, and 5% for discussing parameter tuning, e.g. plotting graphs showing accuracies as parameters vary.
Extra Credit - 20% Up to 10% more for implementing extensions, with an explanation in the report. It is not necessary for the extensions to radically improve accuracy to get credit. Extra 10% for the top 5 systems and 5% for the next 15 systems based on performance.

7 Deliverables

Code Should run seamlessly when we call **./buildmodels.sh** followed by **./runcorrector.sh**.
Report A write-up of the steps you took and any major implementation decisions you made in **report.pdf**.
Partner A list of people who worked together on the assignment, in **people.txt**. One line per student, containing the SUNet ID.

8 Submission Instructions

Go to the directory containing your code, and call
> **python submit.py**
Walk through the steps to submit the uniform edit cost model, empirical edit cost model, the report, and extra credit.

Be mindful of the fact that this script actually calls your **./buildmodels.sh** and **./runcorrector.sh** (with the arguments as described previously) on *your machine* using a test dataset. The corrector phase can take around 10 minutes.

Then, as a last step, you will have to submit your code, including the **report.pdf**. Zip your assignment directory using the **.zip** archiver (*without* the data we provided or your saved models) and upload it on the Coursera assignments page using the simple uploader you have used before. Your zip should be a few hundred kilobytes. Do not worry about making submissions for the other parts, we will populate those with scores using your reports and code.

Partners need only submit one copy named
<FirstPersons SUNetID>_<SecondPersons SUNetID>.zip.