CS 276 / LING 286 Spring 2013

Programming Assignment 2

Due: Thursday, May 2, 2013, 4 p.m. PDT

Data

Original training corpus 98,998 documents crawled from the stanford.edu domain. Block structure 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 in **data/queries.txt**
Correct queries in **data/gold.txt**.
Google's spell check's results in **data/google.txt**.

Basic Theory

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

We are trying to find the query that maximizes the conditional probability $P(Q|R)$, the probability that the user meant to search for Q when (s)he entered R. Note that often $Q = R$, in which case the correction is that there is no misspelling.

To estimate this probability, were going to use Bayes' theorem to tease apart the confounded factors. Note that since were 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, and so we can disregard it.

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

The probability of seeing the query $P(Q)$ will be derived from a language model that we will estimate from our training corpus of documents. And the probability of the user entering a particular sequence R, given that (s)he meant to enter the query Q, $P(R|Q)$ is estimated from the noisy channel model of possible edits. We will spend our time optimizing this piece. 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.

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. Then, we will call **runcorrector.sh**, and your code should accept two arguments specifying which sort of noisy channel probabilities to use and the file containing the queries to be corrected (one per line). This should output the 'correct' queries on **stdout**, one per line, in the same order as the test file.

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?

Language Models

The first step you are undertaking is building a language model to estimate $P(Q)$ from the training document corpus. We will be using bigram and unigram probabilities with the interpolation covered in class to handle smoothing.

The probability for a given sequence of terms is simply given by the bigram probabilities of each bigram in the sequence:

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

Note that here we are using the unigram probability for the first term. Since these queries are sampled from our document corpus, there is no meaning to applying a special bigram context for a word beginning the query.

We suggest that you perform these calculations in log space to avoid numerical underflow, and recall $\log(a * b) = \log(a) + \log(b)$. And since the final operation is an $argmax_Q(..)$, you can make the comparison in log space, since it is a monotonic function.

Calculating Probabilities

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

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

Unigrams: $P_{mle}(w_1) = \frac{count(w_1)}{T}$, where T = total number of terms.

The unigram probabilities model will also serve as a dictionary, since were 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.

N-gram Interpolation

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.

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

Well start with $\lambda = 0.2$ as suggested in lecture. Before submitting though you should experiment with this value and see if you can get better relative probabilities on the development dataset. For the statisticians, make a held-out validation set if you like, but there is not much data, and over-fitting is not a major concern in this assignment.

Noisy Channel Model

Now we get to the unique challenge in spelling correction, modeling the probability $P(R|Q)$, that is concerned with the probability that the user would enter a query R when (s)he intended to enter the query Q. We will arrive at that probability using a noisy channel model that hypothesizes that there is some noise in the communication of the user's intent to the query, and we can quantify the probabilities involved in the generation of that noise.

Crucial to the calculation of this probability is a quantification of the difference between the candidate query Q and the actual input R. When comparing strings, this is the perfect opportunity for an edit distance to be used. Specifically in our case, we will be using the Damerau-Levenshtein distance. There will be a further elaboration of this distance in the next section.

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.

Uniform Cost Edit Distance

In this basic model, we assume that any single edit using an operator defined in the Damerau-Levenshtein distance has a particular uniform probability. We use that probability as our channel model estimate of seeing the candidate query Q. Again, play with different values for this probability, but for starters, 0.01∼0.10 is appropriate.

The atomic operators defined by the Damerau-Levenshtein Distance are insertion, deletion, substitution and transposition. Since the candidate generation component takes care of measuring the edit distance, in this regime all you have to do is calculate the probability of seeing R given the edit distance from Q.

Empirical Cost Edit Distance

After having gotten the spelling corrector working with the basic noisy channel version, turn your attention to this. Here, we apply a more principled approach to the edit probabilities. Namely, we learn them 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 accordingly, learn the probability of that specific edit taking place. We go into the edit probability calculation in some detail on the lecture handout.

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 is also described on the lecture handout.

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 on spelling correction. We can do better than this simple approach by aggressively narrowing down the search space while generating candidates. 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.

A common misspelling is when words are accidentally joined together, or split, by a space. To deal with these cases, you might need to be creative in the alphabet that you consider valid for the set of operators youre dealing with. Hint: as a first step, treat the entire query as a single string that you can make one, or two, edits to. Split and/or combine one or two words. After that, you can treat the words individually and perform your next steps. Also, 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).

Candidate Scoring

Finally, when we put 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.

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

You can optimize this parameter μ on a development section of your training data, or just initialize it with $\mu = 1$.

Implementation Hints

Check out the sample code included in **toy_code.py** if youre planning on working in Python. We make use of the **marshal** module to serialize data to a file and **itertools.izip(..)** to walk through pairs in a list. The documentation for these modules may be helpful, and there are use examples for them all in **toy_code.py**.

Note that running the final **runcorrector.sh** on the full dev or test set might take around 20 minutes, and you might want to run it on a remote machine. Do not leave your submission right till the last minute! We will not be able to give you an extension just to run your code.

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** $\rightarrow$ **le**. Can you incorporate them into the channel model probabilities?
Alternate Smoothing Try an algorithm like stupid backoff instead of the interpolation in the language model 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 that uses a finite state automata for fuzzy matching of words. There is an experimental implementation in Python with code, but it needs to be generalized to perform the transposition operation too.

Grading

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

Uniform edit cost model 20% for a correctly implemented solution clearing 0.65 accuracy. 5% for better tuning of this model over the baseline. You should be able to get to 0.65 without much untoward tweaking. 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.
Empirical edit cost model In this case, 20% of the grade, with room for improvement. Naturally, partial credit will be awarded proportional to how well your model does.
Report 20% of your grade. Be sure to document any design decisions you made, and explain some rationale behind them.
Efficiency 15%, both general running time of the entire operation and qualitative analysis of steps like candidate generation.
Correctness 10% , a check that you calculate your probabilities correctly in an error-free manner.
Parameter Tuning 10% , a sanity check to see that your parameters seem appropriately adjusted.
Extra Credit Up to 20% more for implementing extensions, with an explanation in the report. Its not necessary for the extensions to radically improve accuracy to get credit.

Deliverables

Code Should run seamlessly when we call
./buildmodels.sh [extra] <**training corpus dir**> <**training edit1s file**>
(note: the optional **extra** parameter is used for building the extra credit model)
Followed by
./runcorrector.sh <**uniform** | **empirical** | **extra**> <**queries file**>
(note: use **extra** as the first parameter when testing the extra credit)
The corrector should output the predicted intended query on **stdout**, one per line. Use the development set **data/queries.txt** as the queries file for your experiments. Remember, its an honour code violation to knowingly take a look at the test set.
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.

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.

You can use **stderr** to print out how many queries are done (do not print the queries themselves) to keep track of progress. You can also run this submission on **corn** or [pick your favourite] cloud machine to let it run in the background.

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.