CS 276 / LING 286 Spring 2016

Programming Assignment 2

Due: Thursday, April 28, 2016, 11.59 PM PDT

1 Overview

This assignment should be done in teams of two or individually. The aim of this assignment is to build a Probabilistic Spell Corrector. The Spell Corrector 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 neighboring 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 example, given a query 'Brute Willis', possible candidates are 'Bruce Willis', 'Brute Wills' and many others. One of the candidates could be the original query itself.
- The candidate scorer scores all the generated candidates using the language and the channel models and chooses the best candidate as the intended query.

You are required to submit your code on corn (which we will run with an auto-grader) and a write-up via Gradescope. See later sections for more details.

1.1 Data

The dataset you will be working with for this assignment is available as a zip file at: http://web.stanford.edu/class/cs276/pa/pa2-data.zip. It consists of the following parts:

- **Language modeling corpus**: 99,904 documents crawled from the stanford.edu domain. Block structure can be found in **data/corpus/**, which contains 10 files. Each line in a file represents a new document. This data will be used to learn the language model.
- **Query 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/training_set/edit1s.txt**. This data will be used to learn the channel model.
- **Development dataset**: 455 query with matching correct queries that you can use to measure performance of your model
  (Possibly) misspelled queries are in **data/dev_set/queries.txt**
  Correct queries are in **data/dev_set/gold.txt**.
  Google spell check's results are in **data/dev_set/google.txt**.

1.2 Starter Code and Tutorial

Download the Java starter code from: http://web.stanford.edu/class/cs276/pa/pa2-skeleton.zip. You can use ant or Eclipse to build the code. An accompanying tutorial video that will help you walk through the code was put together by the teaching staff last year and is available here: `https://www.coursera.org/learn/cs276/lecture/BmXrp/pa2-tutorial`. Make sure you are signed up for the course on Coursera.

1.3 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 entering 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 Task-1: Spell Corrector Model with Uniform Edit Cost (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 $\log(a \times b) = \log(a) + \log(b)$. 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 of counting exactly how many bigrams and unigrams appear throughout 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 n-gram 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. It 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 (which we will implement in this task) 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. Try different values for that uniform probability, but in the beginning 0.01$\sim$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$\sim$0.95.

Noisy channel model constructed in this stage will be used during candidate generation and ranking. Since the candidate generation component (next section) takes care of measuring the edit distance between Q and the generated candidate R, 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. 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 brute force generator that builds all candidates within 2 edit distance wouldn't work since the number of candidates would be enormous.

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 strategy 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 your candidate generation strategy and and optimization you made in the report. 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.

2.5 Note about starter code

You can find some sample skeleton code under the `pa2-skeleton` directory. To understand the code structure, we suggest you start by looking at the top level classes `edu.stanford.cs276.BuildModels` (which constructs a language model and a channel model and stores these models to disk) and `edu.stanford.cs276.RunCorrector` (which uses the models to perform spelling correction on a given query dataset). Note that for this task, we are using a uniform cost model. Therefore, we will ignore the query corpus (i.e. the data/training_set/edit1s.txt file) for this task.

All of your submitted code for this PA should reside in a directory `pa2-skeleton`. Under the `pa2-skeleton` directory, you will find two shell scripts, named `buildmodels.sh` and `runcorrector.sh`. Each script invokes the necessary Java program for the model building and spell correction parts of this task. Prior to running the scripts, be sure to compile using `ant` or Eclipse. You may not change the input arguments to these scripts for the sake of our auto-grader. For example for task-1 you can run the scripts in this way:

```
cd pa2-skeleton
ant
```

```
./buildmodels.sh <LM corpus> <edit1s file> <extra>(optional)
./runcorrector.sh uniform <dev queries> <extra>(optional) <gold>(optional)
```

`buildmodels.sh` script needs the following arguments: path to the language modeling corpus (**data/corpus/**) and the query edits file (**data/training_set/edit1s.txt**) and optionally, a third argument "extra". If you implement extra credit, run with "extra" as the third argument. The `runcorrector.sh` script needs the following arguments: the type of noisy channel probabilities to use ("uniform" in case of task-1 and "empirical" in case of task-2) and the file containing the queries to be corrected (**data/dev_set/queries.txt**). 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 set of corrected queries, then add the gold file as another argument.

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

After having 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/training_set/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://web.stanford.edu/class/cs276/handouts/spell_correction.pdf`. As an example of the information your model will learn the probability of the letter e being substituted by the letter a for a correction is

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

Please note that 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.

3.1 Note about starter code

Compared to task-1, the only change for task-2 is the arguments passed to the `buildmodels.sh` and `runcorrector.sh` scripts . For task-2 you can run the scripts in this way:

```
cd pa2-skeleton
ant
./buildmodels.sh <LM corpus> <edit1s file> <extra>(optional)
./runcorrector.sh empirical <dev queries> <extra>(optional) <gold>(optional)
```

4 Extra Credit

We have listed a few 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 report.

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?

Empirical Edit Cost using Wikipedia: in task-2, you used the dataset of queries 1 edit distance apart to learn edit probabilities. If you look at the queries in this dataset, you will observe that most of these queries are related to the Stanford corpus, the same corpus used to build the language model. It would be interesting to explore what happens if the channel model and language model are learned from different datasets (and hence different distributions of the underlying data). To this end, you can use a dataset of spelling errors collected from Wikipedia and available on Peter Norvig's website `http://norvig.com/ngrams/spell-errors.txt`

Alternate Smoothing: try other smoothing algorithms (such as Kneser-Ney smoothing) 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 in Python `https://gist.github.com/491973`, but it needs to be generalized to perform the transposition operation too. This tutorial might be helpful: `http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata`

We will give more extra credit for the best spell correctors based on both accuracy and running time computed on our hidden test data. We will give 10% for the top 5 systems and 5% for the next 15 systems.

5 Grading

In addition to the performance on the released dev set, we will also evaluate the performance of your spell corrector on a distinct test dataset, which has around the same size as the dev dataset.

Task 1 - 55%: 55% of the grade for a correctly implemented solution for Task 1. It is possible to get higher than 80% accuracy on the dev set for this task. For lower accuracies, we will give partial credit proportionally to how well your model does (not linearly proportional though, as that would insufficiently encourage attempts to make the final small improvements). You will be penalized 10% if the running time of `runcorrector.sh` is excessively long beyond the norm and 5% if the memory used by `runcorrector.sh` is excessively large beyond the norm.

Task 2 - 25%: 25% of the grade for a correctly implemented solution for Task 2. It is possible to get higher than 85% accuracy for this task. For lower accuracies, we will give partial credit proportionally to how well your model does (not linearly proportional though, as that would insufficiently encourage attempts to make the final small improvements). Again, you will be penalized 10% if the running time of `runcorrector.sh` is excessively long beyond the norm and 5% if the memory used by `runcorrector.sh` is excessively large beyond the norm.

Report - 20%: Write a 1-2 page report and submit it alongside with your code. 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. Please note if you go drastically beyond the 2-page limit, we may penalize you for an overly long report.

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. We will give 10% for the top 5 systems and 5% for the next 15 systems based on performance.

6 Submission Instructions

6.1 Code submission

Before, submitting **please make sure that** your code follows the following directory structure to ensure auto-grading goes smoothly:

```
pa2-skeleton
├── build.xml
├── buildmodels.sh
├── runcorrector.sh
└── src
    └── <any source files of your program>
```

Note that we will compile and run your program on Farmshare using `ant` and our standard `build.xml`. If you did not complete the assignment on Farmshare, please verify your code compiles and runs on it before submission. You will submit your code using a Unix script that weve prepared. To submit your code, first put your files in a directory on Farmshare (e.g. corn.stanford.edu). Then, from your parent directory (this would be `pa2-skeleton` in the above directory structure), submit using the submission script:

```
cd pa2-skeleton
/afs/ir/class/cs276/bin/submit
```

If you are working in a team, only one team member needs to submit, but remember to indicate your partners SUNetID when prompted by the submit script.

6.2 Uploading Report to Gradescope

Please upload your report to Gradescope under the Assignments section. Again, if you are working in a team, only one team member needs to submit, but remember to add your partner as a team member for the assignment on Gradescope.