1 Overview

The aim of this assignment is to build a probabilistic spell corrector. The spell corrector we build will have 4 distinct parts: the language model, the noisy channel (edit probability) model, the candidate generator, and the candidate scorer.

- The language model counts the occurrences of token unigrams and bigrams in the training corpus in order to determine their probabilities.
- The noisy channel model attempts to estimate the likelihood of the errors that may occur in a query: specifically, the probability of characters being mistakenly deleted, inserted, substituted, or transposed in a query term. In this assignment, we will use two different methods to estimate these probabilities.
- The candidate generator takes a query submitted by a user and produces variations on the query that can then be scored for their relative likelihood. For example, given a query ‘Brute Willis’, possible candidates include ‘Bruce Willis’, ‘Brute Wills’ and many others. (The original, unaltered query is also considered a candidate, to account for cases where no spell correction is necessary.)
- The candidate scorer uses the language model and noisy channel model to assign an overall probability score to each of the candidate corrections. The candidates are then ranked by their scores, and the highest-scoring candidate is chosen as the best proposed correction.

As with all programming assignments in CS 276, you may complete this assignment individually or in teams of two. You will be required to upload a written report via
Gradescope and to submit your code on corn (so that we may run it with our autograder). 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. The corpus is organized in a block structure found at data/corpus/, where you’ll find 10 files. Each line in a file represents the text of a single document. You will use the tokens in these documents to build a language model.

- **Query corpus**: 819,722 pairs of misspelled queries and their corresponding corrected versions, with each pair separated by an edit distance of at most one. The two queries are tab-separated in the file data/training_set/edit1s.txt. You will use this data to build a probability model for the "noisy channel" of spelling errors.

- **Development dataset**: 455 pairs of misspelled and corrected queries, which you will use to measure the performance of your model. There are three files in data/dev_set/: the (possibly) misspelled queries are in queries.txt, corrected versions are in gold.txt, and Google’s suggested spelling corrections are in 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/PHn30/pa2-tutorial

1.3 Recap of Spelling Correction Theory

If a user types in a (possibly corrupted) query $R$, our goal is to find the query $Q$ that the user actually intended to enter. To do this, we will combine several probability models to make an educated guess about what the user really meant to enter.

Stated formally, our problem is to find the query that maximizes the conditional probability $P(Q|R)$ — i.e., the probability that the user meant to enter $Q$ but actually typed $R$. Note that it is often the case that $Q = R$, in which case there is no misspelling. To estimate $P(Q|R)$, we can 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. (This is the term that would normally be in the denominator of the Bayes formula.) Thus, we have:

$$P(Q|R) \propto P(R|Q)P(Q)$$
The probability $P(Q)$ of seeing query $Q$ will be derived from a language model that we will build from our training corpus. The probability $P(R|Q)$ that the user entered a sequence of tokens $R$, given that he or she actually meant to enter sequence $Q$, is estimated from the noisy channel model of possible edits. As discussed in class, we will begin with a basic noisy channel model that uses Damerau-Levenshtein distance with an assumption that any edit is equally likely to occur. Later, we will consider a slightly more nuanced model with non-uniform edit probabilities.

2 Task-1: Spelling Correction with Uniform Edit Costs (55%)

2.1 Language Model

The first step in 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 of term $w_1$, while the remaining ones are bigram probabilities of later terms in the sequence. One important thing to remember is that many of the probabilities we will use in this type of model are very small, and when we multiply many very small numbers together, there is a risk of numerical underflow. Therefore, it is common practice (and strongly recommended here) to perform this type of probability calculation in log space — meaning that you should instead take the log of each probability and sum all of these values together. (Recall that $\log(ab) = \log(a) + \log(b)$, and that the log function increases monotonically. As a result, when comparing two probabilities $a$ and $b$, we can safely compare them in log space because $\log(a) > \log(b)$ if and only if $a > b$.)

2.1.1 Calculating Unigram and Bigram Probabilities

Deriving these probabilities is a simple matter of counting the unigrams and bigrams that appear throughout the corpus. We will be using maximum likelihood estimates (MLEs) for both probabilities, defined as follows:

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

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

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

2.1.2 Smoothing

The unigram probability 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. However, even if we have two
query terms that are both members of our query language, there is no guarantee that
their corresponding bigram appears in our training corpus. To handle this data sparsity
problem, we will interpolate unigram and bigram probabilities to get our final conditional
probability estimates.

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

Try setting $\lambda$ to a small value (say, 0.1) in the beginning, and experiment later with
varying this parameter to see if you can get better correction accuracies on the develop-
ment dataset. However, be careful not to overfit your development dataset. (You might
consider reserving a small portion of your development data to tune the parameters.)

Finally, note that n-gram interpolation is just one way of doing smoothing. Check out
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

The noisy channel model for edits is a little more complicated than the language model.
This model attempts to estimate the “noise” in the user’s attempt to communicate their
query — represented formally as $P(R|Q)$. To compute the noisy channel probability, we
first quantify the difference between the candidate query $Q$ and the actual input $R$ using
the Damerau-Levenshtein edit distance. In Damerau-Levenshtein distance, the atomic
operators defined are insertion, deletion, substitution and transposition.

The uniform cost edit model (which you will implement in this task) simplifies the com-
putation of the noisy channel probability by assuming that every individual edit in the
Damerau-Levenshtein distance has the same probability. You should try a range of val-
ues for your uniform edit probability, but in the beginning 0.01~0.10 is appropriate.
One important thing to remember in building your model is that the user’s input query
$R$ may indeed be the right one in a majority of cases (i.e. $R = Q$). As discussed in
lecture, we typically choose a relatively high fixed probability for $P(R|Q)$ in this case;
once again, you can experiment with different values, but a reasonable range is 0.90~0.95.

The noisy channel model that you construct here will be used when you rank candi-
dates for query corrections. Since the candidate generation component of your system
(described in the next section) takes care of measuring the edit distance between $Q$ and
the generated candidate $R$, all you need to do in this part is calculate the probability of
seeing some $R$ given its 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 user’s intended query, we encourage you to consider possible query cor-
rections that are within distance 2 of $R$. This is the approach taken by Peter Norvig in his essay (http://norvig.com/spell-correct.html) on spelling correction. However, it’s not really an option to use a pure “brute force” generator that produces all possible strings within distance 2 of $R$, because for any $R$ of non-trivial length, the number of candidates would be enormous.

We can improve on the naive approach by aggressively narrowing down the search space while generating candidates. There are lots of valid approaches to efficient candidate generation, but here are a few basic ideas:

- Begin by looking at each individual term in the query string $R$, and consider all possible edits that are distance 1 from that term.
- Remember that you might consider hyphens and/or spaces as elements of your character set. This will allow you to consider some relatively common errors, like when a space is accidentally inserted in a word, or two terms in the query were mistakenly separated by a space when they should actually be joined.
- Each time you generate an edit to a term, make sure that the edited term appears in the dictionary. (Remember that we have assumed that all words in a valid candidate query will be found in our training corpus, as mentioned above in section 2.1.2 above.)
- If you have generated possible edits to multiple individual terms, take the Cartesian product over these terms to produce a complete candidate query that includes edits to multiple terms. (But remember that you probably shouldn’t go beyond a total edit distance of 2 for the query overall!)

Again, there are many possible extensions and variations on the strategies mentioned here. We encourage you to explore some different options, and then describe in your written report the strategies that you ultimately used, and how you optimized their performance. Note that solutions that exhaustively generate and score *all* possible query candidates at edit distances 1 and 2 will not receive full credit.

### 2.4 Candidate Scoring

When combining probabilities from the language model and the noisy channel model (remember to use log space), we can use a parameter to weight the two models differently.

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

Start out with $\mu = 1$, and then experiment later with different values of $\mu$ to see which one gives you the best spelling correction accuracy. Again, be careful not to overfit your development dataset.

### 2.5 Notes on starter code and running your program

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 edit cost model. Therefore, we will ignore the query corpus (i.e. the data/training_set/edit1s.txt file) for this task.

As in PA1, you are welcome to change any of starter code for the assignment. When you are ready to submit, all of your 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. Here is an example of how to run the scripts:

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

The buildmodels.sh script needs the following arguments:

1. The path to the language modeling corpus (data/corpus).
2. The path to the query edits file (data/training_set/edit1s.txt).
3. Optionally, a third argument “extra.” If you implement extra credit, run with the string “extra” as the third argument.

The runcorrector.sh script needs the following arguments:

1. The type of noisy channel probabilities to use ("uniform" in case of task 1 and "empirical" in case of task 2).
2. The path to the file containing the queries to be corrected (data/dev_set/queries.txt).
3. Optionally, a third argument “extra” (to be used if you implement extra credit features, as described above).
4. Optionally, a fourth argument “gold.” If you want to compare your results with a “golden” set of corrected queries, then add the path to the gold data file as another argument. (Note that you may need to implement support for this type of comparison yourself, though some gold data from Google is provided for you.)

### 3 Task 2: Spelling Correction with Empirical Edit Costs (25%)

After your spelling corrector is working correctly with a basic noisy channel model, let’s turn our attention to a somewhat more realistic approach to edit probabilities. In this

---

6
task, we will learn these edit probabilities from the empirical error data provided in data/training_set/edit1s.txt.

As outlined in section 1.1 above, you have been given a list of query pairs that are precisely edit distance 1 from each other. The first step for this task is to devise a simple algorithm to determine which specific edit exists between the two queries in each pair. By aggregating the counts of all such edits over all queries, you can estimate the probability of each individual edit. The edit probability calculation is described in more detail in the lecture handout (http://web.stanford.edu/class/cs276/handouts/spell_correction.pdf). As an example, if you need to determine the probability of the letter e being (mistakenly) replaced by the letter a in a query, you should calculate:

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

Please note that the insertion and deletion operator probabilities are conditioned on the character before the character being operated on — which also means that you should devise an appropriate solution to handle the special case of insertions or deletions occurring at the beginning of a word. Finally, to account for the inevitable problem of data sparsity in our edit training file, you should apply Laplace add-one smoothing to the edit probabilities, as described in the lecture handout.

3.1 Notes on running your program

Compared to task 1, the only changes for task 2 are in the arguments passed to the buildmodels.sh and runcorrector.sh scripts. As a reminder, here are the command-line arguments expected by each script:

```
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 the scope of tasks 1 and 2 will be considered. Be sure to include a description of your work in your assignment report.

**Expanded edit model:** Pandu mentioned in lecture that there are sometimes spelling errors that may not be within a “naive” edit distance 2 of the correct phrase, but that may have a conceptual basis that makes them very common and understandable. (Substituting ph for f, or vice versa, is one such example.) Can you incorporate these types
of errors into the edit probabilities of your noisy channel model?

**Empirical edit costs 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](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 help you generate candidate corrections much more efficiently.


Finally, we will give a small amount of extra credit to the best spell correction systems, measured in terms of both accuracy and running time (as computed on our hidden test data). The top 5 systems according to either metric will receive 5% each, while the next 15 systems will receive 2.5% each.

## 5 Grading

You can use the released dev set of queries to evaluate the performance of your spell corrector. For purposes of grading, we will evaluate your system on a separate, hidden test set, which is around the same size as the dev set.

**Task 1:** 55% of your total grade for this assignment depends on a correctly implemented solution for task 1. You will receive full credit for this task if your system achieves at least 84% accuracy on the hidden test set. For lower accuracies, we will give partial credit on a non-linear scale (which disproportionately favors models that are closer to the threshold for full credit, as an encouragement to squeeze out more performance improvements). You will be penalized 10% if the running time of `runcorrector.sh` is
much larger than the class average, and 5% if the memory used by runcorrector.sh is much larger than the class average.

**Task 2:** 25% of your total grade is based on your implementation of task 2. Full credit will be granted for accuracy levels of 85% and above. (Again, for lower accuracy levels, we will give partial credit on a non-linear scale, with credit accruing more rapidly as your solution gets closer to the target.) You will be subject to the same penalties as above (10% and 5%) if your run time or memory usage, respectively, are far beyond the class average.

**Report:** 20% of your grade is based on the 1-2 page report that you will submit through Gradescope. Be sure to document any design decisions you made, and give some brief rationale for them. We will give 5% credit for overall system design, 5% for more detailed discussion of smoothing and related techniques, 5% for outlining optimizations that you used for candidate generation, and 5% for descriptions of your efforts to tune parameters in your system (e.g., a plot showing how accuracy varies as you change parameter values). Please keep your report concise: material that goes far beyond the 2-page limit may be penalized.

**Extra Credit:** 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. As described above, you can also get a small amount of extra credit if your system is a top performer in terms of accuracy or running time.

### 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 we've prepared. To submit your code, first make sure that all of your project files have been uploaded to a Farmshare machine (e.g. corn.stanford.edu). Then, from the **pa2-skeleton** directory, 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. Again, only one submission is required from students in teams, but remember to tag your partner in Gradescope when you submit.