CS159 - Assignment 2
Due: Wednesday, Feb. 16 at midnight

In this assignment we will be constructing a number of smoothed versions of a bigram language model and will be evaluating its performance based on intrinsically as well as for its usefulness for context dependent spelling correction. For this assignment you will be programming in Java. I have provided some starter code and have also given you a specification to follow that will help guide you through the process and also make grading easier. You will be submitting your code and a short write-up in the dropbox.

You may (and I would encourage you to) work with a partner on this assignment. If you do, you must both be there when either of you are working on the project and you should only be coding on one computer (i.e. pair programming). If you would like a partner, but don't have one, e-mail me asap and I will try and pair you up.

1 Getting Started

First, read through this ***entire*** handout before getting started!

I have provided you with a number of resources for use in training and evaluating your system as well as some initial code that handles some of the I/O and also gives you a template to follow as you construct your language models. All of this can be found in the normal place at:

```
/common/cs/cs159/assignments/assignment2
```

on the Pomona CS network.

1.1 Data

In the `data` directory of the starter there are three files.

- `sentences` contains 110,000 English sentences that we will be using to train and evaluate our language model. These sentences were taken from Simple English Wikipedia data set similar to the one we play with for the first assignment. I have done all the preprocessing work for you, so you can identify individual tokens/words by just splitting on whitespace. You'll also notice I've normalized the text in some cases, for example by lowercasing and converting numbers to a special token.

- `spelling.200.incorrect` contains 200 examples of words being incorrectly spelled in context along with the corrected spelling for the word. This file can be read using the `SpellingExampleReader` class (see below) and we will use it to evaluate the performance of our language models on an applied task.
- `word_list.txt` contains a list of English words that we will be using as candidates during spelling correction. This file should be used as input to the `SimWordGenerator` class.

1.2 Code

I have provided you some basic code to help you structure your project and also so you don't have to waste time with some of the basic I/O. I have included six classes in the starter. I will give a brief description of their uses/purpose here. See the comments in the code for more detail.

- `LMModel`: Describes the interface that a language model must implement, specifically a `logProb` method for calculating the log probability of a sentence and `getPerplexity` method for determining the perplexity of the text in the file based on the language model.
- `SpellingExample`: A class that stores all of the relevant information for our spelling correction examples including the incorrectly spelled word, the left and right context and the actual correct spelling of the word.
- `SpellingExampleReader`: Given the file described above and supplied with the starter with incorrect spelling example, this class will iterate over them, creating a `SpellingExample` object for each example in the file.
- `SimWordGenerator`: This class will be useful when you attempt to determine the correct spelling of a spelling example. You can construct a new `SimWordGenerator` with the list of English words supplied above and then the `getSimThreshold` method will return a list of similar words to the incorrectly spelled word. In our case, we'll use a threshold of 2, but you can play with others if you'd like.
- `SpellCorrector`: This is a starter class that you will fill in the details for to accomplish correct spelling examples. You may add methods to this class but do not change the headers for those that are provided.
- `EditDistance`: Used to calculate the edit distance between two words. This is only used by the `SimWordGenerator` class, so you won't need to use it explicitly.

2 Building your first language model

Implement a bigram language model that is smoothed by adding a small λ to all of the frequency counts in a class called `LambdaLMModel`. This class must:

- `implement` the `LMModel` interface

- Include a constructor as follows:

  `LambdaLMModel(String filename, double lambda)`

  which will learn a new bigram language model from the text in `filename` smoothing with `lambda`

- Use the symbol `<s>` to mark the beginning of sentences and `</s>` to mark the end. You'll have to do this somewhere in this class since the data I've provided you does not include it. Don't forget to also do this in the `logProb` and `perplexity` methods when you're given a new sentence.

- Utilize a fixed vocabulary based on the training data provided and use the `<UNK>` symbol to mark unknown or out of vocabulary words. During training (i.e. when you are learning the probabilities) you should replace the *first* occurrence of each word with `<UNK>` and train the model using that. During testing, (i.e. in the `logProb` and `perplexity` methods) if you see a word that was not seen during training, you should replace it with the `<UNK>` symbol. Notice that if you only saw a word once during training, it would be an unknown word. See pg. 95 from the book for a description of this approach.

- Smooth the bigrams using the supplied lambda. Because we're using the `<UNK>` symbol, you do NOT need to smooth the unigrams. To smooth the bigrams by adding lambda to the accounts and normalizing appropriately.

- Any logs should be log base 10 (use `Math.log10`).

Hints/Advice

- You should use `HashMap`s (i.e. hashtables) and related structures such as `HashSet` wherever appropriate to store your counts and probabilities.

- There are a number of possible approaches for storing the bigrams. You could use a single `HashMap` with the key being the bigram, however, a better way to do it is to use a hashtable of hashtables. The main hashtable is keyed off of the first word and the value is another hashtable. The second hashtable is keyed off of the second word in the bigram and has the value as the probability. This approach will make it much easier for later parts of this assignment (don't say I didn't warn you).

- Make sure you understand how the lambda smoothing approach works and that you're normalizing your probabilities correctly.

- You should only store bigrams that you've actually seen in the table. During testing, if you encounter a bigram that you have not seen before, then you can calculate it's probability on the fly based on lambda and the size of your vocabulary. If this doesn't make sense, come talk to me.

- These things can be very, very hard to debug and to determine if you've got the right answer. I strongly suggest that you come up with a set of very simple sentences (for example using just one letter "words") and calculate the correct probabilities by hand. Then, train your

model on this and print out the probabilities and compare your answers. If you just start training on the entire data set, you're going to have a hard time telling if you're doing it right.

- Be careful about underflow. I would encourage you to just do everything in logs, including storing the bigram table. For example if you wanted to calculate $log(a/b)$ then you would calculate it as $log(a) - log(b)$. If you don't do it this way, you may have problems.
- When you do start training on the larger amounts of text, you may need to increase your heap size (you'll probably get an out of memory exception if you don't). If you've running on the command-line, just add `-Xmx2G` after `java` (2G specified 2 GB of heap space which should be plenty). If you're using Eclipse, under "Run Configurations...", select the "Arguments" tab and under "VM arguments:" included `Xmx2G`.

3 A better language model?

The language model above does not use the unigram probabilities at all. We're going to try and improve this and construct a language model that backs off to the unigram probabilities. Specifically, we're going to construct an absolute discounted backoff language model (see the lecture notes and pg. 110 in the book).

Create a new class called `DiscountLMModel` that implements the a backoff, discounted bigram model with the following features:

- `implements` the `LMModel` interface.
- Includes a constructor

  `DiscountLMModel(String filename, double discount)`

  which will learn a new bigram language model from the text in `filename` discounting each bigram count by `discount` to be used for backoff calculations.
- Like the language model above, we we enclose sentences in `<s>` and `</s>` and use a closed vocabulary with the same approach for using `<UNK>`.
- Unigram probabilities should be calculated normally.
- During training, when calculating bigram probabilities, you will discount the actual bigram count by `discount` (.75 is a good place to start). During testing, if you encounter an unseen bigram, you will calculate it's probability as the backoff factor (α in the book) times the unigram probability of the second word in the bigram. *alpha* will be different for each bigram and will depend on how bigrams were disctounted that started with the first word in the unseen bigram. This should be straightforward to calculate is you represented the bigram probabilities as a hash of hashes like I suggested.
- Again, make sure to use log base 10 for all of your calculations.

Hints/Advice

- Again, I strongly encourage you to work out the probabilities by hand on a small example and then compare them to the system's output.
- If your training for the first language model was done in two steps, first collecting the counts, then going back and calculating the probabilities, I would encourage you to reuse code. The best way to do this would be to create an intermediary abstract class (say something like `LMBase`) that has appropriate protected variables for aggregating counts and the count collection method. You can then `extend` this class with both your language model classes.

4 Language model-based spelling correction

Once you've got your language models working, we're going to use them to try and do some spelling correction. Fill in the details for the `SpellingCorrector` class. You'll just need to fill in some details for the constructor and the `guessCorrection` class. Do not change the method headers for these, but feel free to add additional private methods if you'd like.

To figure out the "best" spelling correction using your language model, do the following:

1. Get a list of possible corrections using the `SimWordGenerator` class
2. For each of these words, create a new sentence with that word in the appropriate place.
3. Get the probability of each of these sentences using your language model.
4. Pick the word that results in the best language model score.
5. If no options exists, just return the empty string.

5 Evaluation

Now that we have some working systems, I'd like for you to evaluate how well each of the systems is doing, with a variety of parameters and include a writeup (with data) describing the results from the experiments below. In each case, provide the results of your experiments and a short paragraph analyzing your results. Make sure to number your sections so it's easy to review.

1. What is the best lambda smoothing parameter?

   Split the sentences into three parts: 90,000 training, 10,000 development and 10,000 testing (the command-line commands `head` and `tail` may be useful). Calculate the perplexity on the development set for lambda in (.1, .01, .001, ..., .00000001) and provide the results in your write-up. What is the best lambda?

   Now, cheat and train on the 90,000 but find the best lambda on the test set. Is it the same as the one you found on the development set?

2. What is the best discount?

   Using the same splits as above, calculate the perplexities on the development set for the discounted model with discounts in (0.99, 0.9, 0.75, 0.5, 0.25, 0.1). Provide the results and analysis in your write-up

3. Spelling correction

   Using the best parameters from the previous two experiments, examine the spelling correction data set. To compare the approaches, calculate the precision of each approach on the data set, where precision is the number correct divided by the total number of examples. Provide the results and an analysis in your writeup. Do the spelling correction results have the same outcome as the perplexity results (i.e. which system is better)?

4. Training data size

   For both of the approaches, vary the training data size and see how it affects both the perplexity and spelling correction performance. Provide data and analysis.

5. Wrap-up

   Very briefly answer the following questions: how long did you spend on this assignment? what was the most fun part? least fun part? how would you improve it if I had to give it again?

6 Extra Credit: More language models!

Experiment with up to two other smoothing techniques (for example, Kneser-Ney discounted back-off, interpolated models, Good-Turing discounting, Witten-Bell discounting). You will receive up to 3 points extra credit per smoothing technique (depending on difficulty). If you do this, just create other classes that implement the `LMModel` class. Do NOT change either of the classes above.

Include an additional section in your writeup where you provide results and analysis of the performance of your new approach(es). You don't necessarily have to answer all of the questions above, but you should provide some results.

When you're done

When you're all done, follow the directions on the course web page for submitting your work in the dropbox. Make sure that your code compiles, that your files are named as specified and that all your methods have the same name and parameters as specified. If you get an error, try changing the name of the folder to include a version number and resubmit.

All of your code and your writeup should be in a folder.

If you worked with a partner, put both people's last names on the submitted directory, but only submit one copy.

What to submit

You should submit the full working code including the six original classes I specified (you will have modified `SpellCorrector`) as well the `LambdaLMModel` and `DiscountLMModel` classes. You likely will have written additional code to run experiments, etc. You don't need to submit this, though if you do, make sure it compiles, etc. and I reserve the right to look at it for style and commenting :)

Commenting and code style

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name (or names) and the assignment number should be at the top of each class file you modify.
- Each class and method should have appropriate JavaDoc comments.
- If anything is complicated, put a short note in there to help me out if there are any issues.

This is a non-trivial assignment and it can get complicated, which makes code style and comments very important so that I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.

Grading

Part	points
LambdaLM	15
DiscountLM	15
SpellCorrector	5
evaluation/write-up	20
style/commenting	10
extra credit	6
total	65 + 6 extra