CS159 - Assignment 2b

Due: Friday, February 15 at 5pm

http://www.smbc-comics.com/index.php?db=comics&id=2884

For the main part of this assignment we will be constructing a number of smoothed versions of a bigram language model and we will be evaluating its performance intrinsically using perplexity. 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 make grading easier. You will be submitting your code and a short write-up.

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 data resources for use in training and evaluating your system as well as some code that gives you a template to follow as you construct your language models. All of this can be found in the assignment starter at:

`http://www.cs.pomona.edu/~dkauchak/classes/cs159/assignments/assign2-starter.zip`

1.1 Data

In the `data` directory of the starter there are is a file called `sentences` that contains 110,000 English sentences that we will be using to train and evaluate our language model. These sentences were taken from a Simple English Wikipedia data set similar to the one we examined for the first assignment. I have done all the preprocessing work for you:

- you should identify individual tokens/words by just splitting on whitespace
- and each line in the file should be treated as a sentence

You'll also notice I've normalized the text in some cases, for example by lowercasing and converting numbers to a special token. Do not do any additional processing (i.e. sentence splitting or word tokenization) on this assignment since it will change your results.

1.2 Code

I have provided you with an interface to help structure your project and to make my life easier for grading. Your language models will implement this interface.

`LMModel`: Describes the interface that a language model **must** implement, specifically a `logProb` method for calculating the log probability of a sentence, `getPerplexity` method for determining the perplexity of the text in the file and `getBigramProb` which will give you the probability of a particular bigram. See the comments in the code for more detail.

Notice that the code I've provided for you is within the `nlp.lm` package. **All of your code for this assignment should also be in this package.**

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 train 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 need to add these internally for all of the methods since the data will not include them. 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.
  - `<s>` and `</s>` will be part of your vocabulary, but do NOT replace the first occurrences of `<s>` and `</s>`.

    For example, if you were given three sentences (I've used letters for words):

    ```
    a a a b
    a b b a
    c a a a
    ```

    this would become:

    ```
    <s> <UNK> a a <UNK> </s>
    <s> a b b a </s>
    <s> <UNK> a a a </s>
    ```

    and then you'd use this data to train your model. This would result in a vocabulary of 5 words, specifically `<s>, <UNK>, a, </s>` and `b`.

  During testing (i.e. in the `logProb`, `perplexity` and `getBigramProb` 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.

    For example, the sentence

    ```
    a b a a c a d
    ```

    would become

    ```
    <s> a b a a <UNK> a <UNK> </s>
    ```

- Smooth the bigrams using the supplied lambda. Because we're using the `<UNK>` symbol, you should NOT smooth the unigrams. To smooth the bigrams add lambda to each of the counts and then normalize appropriately.

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

Hints/Advice

- I *strongly* suggest that debug your code using examples that you've verified by hand. You can use your solutions from Assignment 2a for this with very minor changes to your code (e.g. comment out the part(s) in your code that add in the begin and end sentence words). You can then print out or use the debugger to look at individual probability distributions, i.e. $p(\cdot|a)$. You might also want to do a few extra examples by hand to test other cases. For example, make up three sentences or so with just 3-4 words in the vocabulary (like the above examples) and work out what the probabilities should be.

- There are many ways of calculating the probabilities, but my advice is to run through the data once and record all of the counts that you need. Then, go back through your stored counts and calculate the probabilities.

- You should use `HashMap`s (i.e. hashtables) and related structures such as `HashSet` wherever appropriate to store your counts and probabilities, otherwise, it's going to be too slow.

- 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).

- 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 its probability on the fly based on lambda and the size of your vocabulary. If this doesn't make sense, come talk to me.

- Be careful about underflow. When calculating the log prob of a sentences do *NOT* simply calculate the product of the bigram probabilities and then take the log. Instead, take the sum of the logs of the individual bigram probabilities. If you don't do it this way, you may have problems, particularly for longer sentences.

- When you do start training on 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're 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`.

- On my laptop, my implementation finishes in less than 2 seconds. If you find that yours is taking a long time (i.e. much more than a minute or so) you're probably doing something inefficiently and you should try and fix it.

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 the book).

Create a new class called `DiscountLMModel` that implements 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 enclose sentences in `<s>` and `</s>` and use a closed vocabulary with the same approach for using `<UNK>` as for `LambdaLMModel`.
- Unigram probabilities should be calculated normally. Note that `<s>, </s>` and `<UNK>` will all have their own unigram probabilities since they are part of the vocabulary.
- 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 its probability as the backoff factor (α in the book) times the unigram probability of the second word in the bigram. α **will be different for each word and will depend on how many bigrams were discounted that started with the first word in the unseen bigram**. This should be straightforward to calculate if 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. Think simple, with just a few sentences of length 4-5 and a vocabulary of just 3-4 words.
- 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. For the discount model you'll also need to aggregate the counts as the first step. 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.

- Training this language model also runs in less than 2 seconds for my implementation on my laptop. Again, if yours takes significantly longer than this (i.e. minutes) then you're probably doing something inefficiently and should try and figure out where.

4 Code specification

Make sure to follow the specifications outlined above. Do not change the `LMModel` interface and make sure that your constructors are as defined above. Do not include any `throws` statements in any of the public method declarations. This changes the interface and will make grading more of a pain.

5 Evaluation

Now that we have some working systems, I'd like for you to evaluate how well each of the systems are 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.

Create a file with your answers to the following question in a reasonable format. Make sure that your name(s) are at the top of the file and 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 (either as a table or a graph).
   - Now, calculate the perplexity on the test set for lambda in (.1, .01, .001, ..., .00000001) and provide the results in your write-up.
   - Discuss your results. In particular, what is the best lambda on the dev and test set? If you didn't cheat (i.e. picked the best lambda using the dev set) how far off from the optimal lambda would you have been? Add any other observations.

2. What is the best discount?

   Do the same as above except now 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. Performance

   Which model is better? Provide some quantitative and/or qualitative arguments (including data or examples) of which approach is better. Make sure to clearly explain you evaluation approach and your arguments.

4. 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 analysis [2 points]

Training data size

Investigate the impact on performance as you vary the training data size. Since we're using perplexity for evaluation, we need to be careful that it's a fair comparison, that is, all models should be distributing their probability mass over the same number of things (i.e. the same vocabulary). To investigate this, you'll need to modify your code slightly:

- Generate a vocabulary that you will use for *all* of your experiments. A reasonable choice would be the vocabulary from the largest test set.
- Modify your code so that you can optionally initialize it with a specified vocabulary list. I'll leave how you do this up to you, but make sure that your code still conforms to the specifications above.
- Run experiments for varying sizes of training data. Think a bit about what to use for the lambda and discount for these.

Provide data and analysis on your results.

More models [3 points]

Experiment with another smoothing techniques (for example, Kneser-Ney discounted backoff, interpolated models, Good-Turing discounting, Witten-Bell discounting). 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

The online submission site takes a single file. To do this:

- Create a base folder that will contain all of your work in. Name this folder with your last name(s) and the assignment number, e.g. `kauchak2` (or `kauchak-mezini2`, if I was working with a partner that had "Mezini" as a last name).

- Your writeup should go in the base folder. Make sure that the names of everyone in your group is in the writeup.
- Within the base folder, create a folder called `code` and place your code in there. Make sure you preserve the package directory structure, i.e. all your code should be in `nlp/lm/`.
- `zip` the base folder and submit this using the online submission mechanism. Only one person in the group should submit the material, though you should *tag* your partner in the submission system when submitting.

What to submit

You should submit the full working code including the `LMModel` interface 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
evaluation/write-up	20
style/commenting	10
extra credit	5
total	60 + 5 extra