CS114 (Spring 2016) Homework 2
N-Gram Language Model

Due February 26, 2016

The overall goal of this assignment is for you to train the best language model possible given the same limited amount of data. Your grade will depend on the correctness of your implementation, the written report, the perplexity scores of your various models, and your models' performance on the "jumbled sentence task" (discussed below).

The `data` folder should contain the training set (`train-data.txt`), the development set (`dev-data.txt`), and two test sets (`test-data.txt` and `test-data-no-oov.txt`, which contains no words that are out of the vocabulary of the training set). The `src` folder contains the source code and `langmodel` contains the code you should actually edit and is where you should place any new files you create.

Assignment

1. Implement language models and come up with the best language model possible given the datasets described above.

   Note that if you train a trigram model, you must append two tokens of `<S>` to each sentence before training. For example:

   `P(I like running) = P(I|<S> <S>) • P(like|<S> I) • P(running|I like)`

   For simplicity of implementation, treat all punctuation as words and leave them as they are.

   For each language model you build, you will need to implement the following:

   - `train(Collection<List<String>> trainingSentences)` - Trains the model from the supplied collection of training sentences. Note that these sentence collections are disk-backed, so doing anything other than iterating over them will be very slow, and should be avoided.
   - `getWordProbability(List<String> sentence, int index)` - Returns the probability of the word at *index*, according to the model, within the specified sentence.

- `getSentenceLogProbability(List<String> sentence)` - Returns the probability, according to the model, of the specified sentence. Note that this method and the previous method should be consistent with one another, and in all likelihood this method will call that method. (This is already done for you in `LanguageModel.java`).
- `generateSentence()` - Returns a random sentence sampled according to the model. The generation method should be consistent with the language mode and should take into account the amount of context specified by your N-gram window (that is, a unigram model is just pulling words out of a bag, a bigram model should generate sentences where each bigram occurs in your language model, etc.)

Evaluation

You are given a shell script, `run`, that automatically launches the evaluator and runs all tests. If you edit this file, you can change the language model that is used for evaluation by editing the `model` parameter. E.g. `-model cs114.langmodel.ExampleUnigram` uses the ExampleUnigram model provided in `classes/cs114/langmodel`. `-model cs114.langmodel.MyAwesomeModel` will use the MyAwesomeModel language model provided a class file containing that model exists in `classes/cs114/langmodel`.

LanguageModelTester is the evaluation module. It tests the following:

1. Makes sure the probability sums to one given a random context.
2. Computes the perplexity on the train and test set. You should use the test set with no OOV words, which is provided. Edit the `-test` parameter in the run script to change which data set is used for testing.
3. The jumbled sentence task. This will use your LM to find which sentence is a real sentence out of 10 jumbled sentences. All of these tests are given to you. This should make you evaluation very consistent and easy to compare across models.

Tips and Tricks

1. It is useful to know that

$$\log \prod_{i=1}^{n} P(w_i) = \sum_{i=1}^{n} \log P(w_i)$$

2. After you turn the counts into the probabilities, you should store log-probabilities. Remember that exp log $P(w) = P(w)$, so it is easy to convert back and forth between probability and log probability.

3. You should use this formula for computing perplexity:

$$PP_M(W) = \exp(-\tfrac{\log P_M(W)}{n})$$

   because it uses log probability directly. Adding is much faster than multiplying, and the product of probabilities will lead to numerical underflow (a number too small to be represented by a Python float value). Using the sum of log probabilities helps avoid this problem.

4. Start small. Start with a bigram model with add-one smoothing and evaluate it. From there, it is not too hard to expand to trigram or 4-gram.

5. It is OK if your model does not perform too well compared to your classmates at the end as long as we see in the report that you have tried a good number of different models to make the performance better.

6. Do not train on the dev set ever. Instead, use it for tuning the interpolation weights, or deciding on your n (e.g. choosing between a 3-gram or 4-gram model).

7. It is a wise idea to cache the probability instead of computing it on the spot every time.

8. Look at `ExampleUnigram.java` to see how things should be done, including at the `generateWord` method to see how to generate a random word using the roulette method.

9. Look at the `Counter` classes provided to them as utility (`src/cs114/util`).

10. We encourage you to use an IDE to write and compile your code. Eclipse is a good one for Java. You can use `build.xml` in the package to create a project on Eclipse.

11. The "run" script will allow you to just run your stuff on the terminal as well.

12. Use subclasses as different models may share a lot of code.

13. Katz backoff yields good results and is easy enough to implement.

14. You must model the end of sentence symbol. Make sure you look at how it's done it in the example unigram LM.

Write-up

Write a short report on the language models that you have explored. You should at least describe the following:

- The models that you have tried (e.g. bigram with plus-one smoothing, trigram with Katz backoff, etc.)
- Perplexity on the training set, test set, and test set (no OOV) for each model
- Performance of each model on the jumbled sentence task

Submission Instructions

Zip up your report (PDF format, please!), with the cs114_hw2 folder that contain all your implementation and compiled LM classes, and submit it on LATTE.

Special Thanks

Special thanks to Te Rutherford for developing this assignment!