CS114 (Spring 2018) Homework 2
N-Gram Language Models

Due February 16, 2018

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`), 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), and a lot of jumbled data (`jumble`). You should only create more models similar to the one in `unigram.py`, while leaving the other files intact.

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 bigram model, you must append a tokens of `<S>` to each sentence before training. For example:

   $P(\texttt{I like running}) = P(\texttt{I}| < \texttt{S} >) \times P(\texttt{like}|\texttt{I}) \times P(\texttt{running}|\texttt{like})$

   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(trainingSentences)` - Trains the model from the supplied collection of training sentences.
   - `getWordProbability(sentence, index)` - Returns the probability of the word at `index`, according to the model, within the specified `sentence`.

- `getSentenceLogProbability(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).
- `generateSentence()` - Returns a random sentence sampled according to the model. The generation method should be consistent with the language model 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.).

There is no set number of language models you need to create, although a good guideline is that your language models should cover multiple values of n (e.g. bigram, trigram, etc.) and multiple smoothing methods (e.g. add-one, backoff, interpolation, 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 unigram.Unigram` uses the `Unigram` model provided.

`tester.py` tests the following things:

1. Makes sure the probability sums to one given a random context.
2. Computes the perplexity on the training and test sets. 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 your evaluation very consistent and easy to compare across models.

Tips and Tricks

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

2. 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.

3. 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).

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

5. Look at `unigram.py` to see how things should be done, including at the `generateWord` method to see how to generate a random word using the roulette method.

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

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

8. You must model the end of sentence symbol. Make sure you look at how its 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 interpolation, 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 all your new language models. You don't need to include any data or any of the code that was provided to you.