CS114 (Spring 2020) Programming Assignment 3
N-gram Language Models

Due March 20, 2020

The overall goal of this assignment is for you to train the best language model possible given the same limited amount of data. The `data` folder should contain the training set (`train-data.txt`), the development set (`dev-data.txt`), and a lot of jumbled data (`jumble-dev`). Test sets (for sentences and jumbled data) have been held out and are not given to you. You are also given `train-data_small.txt` and `test-data_small.txt`, the toy corpus from HW2, in the same format. You should only create more models similar to the one in `unigram.py`, while leaving the other files intact.

Assignment

Unigram model

The unigram model, `unigram.py`, is given to you; you do not need to (and should not) change anything. However, before creating any new models, you should read the code and understand what it is doing. In particular, note the following:

- `train(self, trainingSentences)` - Trains the model from the supplied collection of `trainingSentences`. First, we count all the words in the training data. Note that one symbol `LanguageModel.STOP` is counted at the end of each sentence, and the unknown word `LanguageModel.UNK` is also given a count of 1.

  Then we build `self.prob_counter` as a `scipy.sparse.lil_matrix`. For the unigram model, this is not necessary, but for bigram models with large vocabularies, it is impossible to store all of the probabilities in a Numpy array without running out of memory. However, using a sparse matrix (here, in list of lists format), we can take advantage of the fact that most of the bigram probabilities are 0, and store only the non-zero probabilities, saving a lot of space. There are a few idiosyncrasies (e.g. division becomes multiplication by the reciprocal), but for the most part, we can use Scipy sparse matrices similarly to Numpy arrays.

As in PA2, `self.word_dict` should be used to translate between indices and words; by popular demand, this time we can use a {`word:index`} dictionary rather than {`index:word`}. Note the use of `self.total` to store the denominator—in this case, the total number of words in the training data. Finally, we divide by `self.total` to get the probabilities $P($`word`$)$ in `self.prob_counter`.

- `getWordProbability(self, sentence, index)` - Returns the probability of the word at `index`, according to the model, within the specified `sentence`. Note that if the index is at the end of the sentence, we return the probability of `LanguageModel.STOP`, and if we have not seen the word at `index` before, we return the probability of `LanguageModel.UNK`.

- `generateWord(self, context)` - Returns, for a given `context`, a random word, according to the probabilities in the model. The `context` is a list of the previous words in the sentence. For the unigram model, the `context` is ignored, so we simply use `numpy.random.choice` to generate a word using the unigram probabilities in `self.prob_counter`.

Bigram model

Now that you are familiar with how the unigram model works, your first task is to create a bigram model in `bigram.py`. Whereas a unigram model uses probabilities of words $P($`word`$)$, a bigram model uses conditional probabilities $P($`word`$|$`previous_word`$)$. Therefore, instead of just storing a single probability as `self.probCounter[word]`, we can set
`self.prob_counter[previous_word][word]` $= P($`word`$|$`previous_word`$)$.
Similarly, `self.total[previous_word]` stores the count of `previous_word` in the training data.

Some notes:

- For a word at the beginning of the sentence (i.e. the `index` is 0 in `getWordProbability`, or the `context` is empty in `generateWord`), the previous "word" will be `LanguageModel.START`.

- When building a conditional probability table (as in HW2), there is no row for `</S>`, and no column for `<S>`. Probably the easiest way to account for this in your code is to use a single index to represent `LanguageModel.START` when used for a row and `LanguageModel.STOP` when used for a column.

- If you get a divide-by-zero warning when calculating the (dev) test set/jumbled sentence perplexity, that is good! Do not worry about that.

Bigram model with add-k smoothing

Your next task is to create a bigram model with add-k smoothing, in `bigram_add_k.py`. A naïve implementation would involve calculating the adjusted probabilities $\hat{P}(w_n|w_{n-1}) = \dfrac{\text{count}(w_{n-1}, w_n) + k}{\text{count}(w_{n-1}) + k|V|}$ and storing them in `self.prob_counter`. Do not do this! (You will run out of memory.) Instead, we will use some tricks:

1. Collect the bigram counts as before, such that `self.prob_counter` only contains those bigrams that have non-zero actual counts in the training data.
2. Add $k|V|$ to `self.total[previous_word]`, so that after dividing, the values of `self.prob_counter` have form $\dfrac{\text{count}(w_{n-1}, w_n)}{\text{count}(w_{n-1}) + k|V|}$.
3. Then, whenever we need a probability (array), we can add the missing $\dfrac{k}{\text{count}(w_{n-1}) + k|V|}$ inside of `getWordProbability` and `generateWord`.

Otherwise, your procedure should be the same as for the unsmoothed bigram model.

Setting the value of k

Which value of k should you use? When you are first writing your code, you can use $k = 1$ (i.e., add-one smoothing) for simplicity. After you get your algorithm working, though, you should try to find the value of k that results in the best performance on the dev set, either in terms of minimum perplexity, or maximum accuracy on the jumble task, or both.

Bigram model with interpolation

Finally, you should create a bigram model with (simple linear) interpolation, in `bigram_interpolation.py`. If you implemented `bigram.py` correctly, you should be able to leave `train` and `getWordProbability` (and `__init__`) as is, only needing to fill in `generateWord`.

Tuning the interpolation weights

As above, you should experiment with finding the values of λ that optimize the dev set perplexity and/or jumble task accuracy. Both books mention the EM (expectation-maximization) algorithm, but for now, trial and error is fine.

Evaluation

You are given a shell script, `run`, that automatically launches the evaluator and runs some 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 (dev) 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.

Write-up

Write a short report on the language models that you have explored. You should at least describe the following (which will count toward your grade):

- How you set the value of k/tuned the interpolation weights
- Perplexity on the training set and dev set for each model
- Performance of each model on the jumbled sentence task

Note that you only need to report on models trained/tested on the full data (you do not need to include any results on the toy data).

Please also include the following (which will *not* count toward your grade):

- (About) how many hours you spent working on this assignment
- Any parts of the assignment you found particularly easy or difficult
- Any other comments on the assignment you would like to make

Submission Instructions

You should submit four files: your report (in PDF format), `bigram.py`, `bigram_add_k.py`, and `bigram_interpolation.py`. You don't need to include any data or any of the other code that was provided to you.