Programming Component - Building a Trigram Language Model (70 pts)

The instructions below are fairly specific and it is okay to deviate from implementation details. **However: You will be graded based on the functionality of each function. Make sure the function signatures (function names, parameter and return types/data structures) match exactly the description in this assignment.**

Please make sure you are developing and running your code using Python 3.

Introduction

In this assignment you will build a trigram language model in Python. You will complete the code provided in the file **'trigram_model.py'**. The main component of the language model will be implemented in the class *TrigramModel*. Parts of this class have already been provided for you and are explained below.

One important idea behind implementing language models is that the probability distributions are not precomputed. Instead, the model only stores the raw counts of n-gram occurrences and then computes the probabilities on demand. This makes smoothing possible.

The data you will work with is available in a folder named '**hw1_data'**. There are two data sets in this folder, which are described below in more detail.

Part 1 - extracting n-grams from a sentence (10 pts)

Complete the function `get_ngrams`, which takes a list of strings and an integer *n* as input, and returns padded *n*-grams over the list of strings. The result should be a list of Python tuples.

For example:

```
>>> get_ngrams(["natural","language","processing"],1)
[('START',), ('natural',), ('language',), ('processing',), ('STOP',)]
>>> get_ngrams(["natural","language","processing"],2)
('START', 'natural'), ('natural', 'language'), ('language', 'processing'), ('processi
ng', 'STOP')]
>>> get_ngrams(["natural","language","processing"],3)
[('START', 'START', 'natural'), ('START', 'natural', 'language'), ('natural', 'langua
ge', 'processing'), ('language', 'processing', 'STOP')]
```

Part 2 - counting n-grams in a corpus (10 pts)

We will work with two different data sets. The first data set is the Brown corpus, which is a sample of American written English collected in the 1950s. The format of the data is a plain text file **brown_train.txt**, containing one sentence per line. Each sentence has already been tokenized. For this assignment, no further preprocessing is necessary.

Don't touch **brown_test.txt** yet. We will use this data to compute the perplexity of our language model.

Reading the Corpus and Dealing with Unseen Words

This part has been implemented for you and are explained in this section. Take a look at the function `corpus_reader` in **trigram_model.py**. This function takes the name of a text file as a parameter and returns a Python generator object. Generators allow you to iterate over a collection, one item at a time without ever having to represent the entire data set in a data structure (such as a list). This is a form of *lazy evaluation*. You could use this function as follows:

```
>>> generator = corpus_reader("")
>>> for sentence in generator:
            print(sentence)

['the', 'fulton', 'county', 'grand', 'jury', 'said', 'friday', 'an', 'investigation',
'of', 'atlanta', "'s", 'recent', 'primary', 'election', 'produced', '``', 'no', 'evid
ence', "''", 'that', 'any', 'irregularities', 'took', 'place', '.']
['the', 'jury', 'further', 'said', 'in', 'term-end', 'presentments', 'that', 'the', '
city', 'executive', 'committee', ',', 'which', 'had', 'over-all', 'charge', 'of', 'th
e', 'election', ',', '``', 'deserves', 'the', 'praise', 'and', 'thanks', 'of', 'the',
'city', 'of', 'atlanta', "''", 'for', 'the', 'manner', 'in', 'which', 'the', 'electio
n', 'was', 'conducted', '.']
['the', 'september-october', 'term', 'jury', 'had', 'been', 'charged', 'by', 'fulton'
, 'superior', 'court', 'judge', 'durwood', 'pye', 'to', 'investigate', 'reports', 'of
', 'possible', '``', 'irregularities', "''", 'in', 'the', 'hard-fought', 'primary', '
which', 'was', 'won', 'by', 'mayor-nominate', 'ivan', 'allen', 'jr', '&', '.']
...
```

Note that iterating over this generator object works only once. After you are done, you need to create a new generator to do it again.

As discussed in class, there are two sources of data sparseness when working with language models: Completely unseen words and unseen contexts. One way to deal with unseen words is to use a pre-defined lexicon before we extract ngrams. The function `corpus_reader` has an optional parameter lexicon, which should be a Python set containing a list of tokens in the lexicon. All tokens that are not in the lexicon will be replaced with a special "UNK" token.

Instead of pre-defining a lexicon, we collect one from the training corpus. This is the purpose of the function `get_lexicon(corpus)`. This function takes a corpus iterarator (as returned by `corpus_reader`) as a parameter and returns a set of all words that appear in the corpus more than once. The idea is that words that appear only once are so rare that they are a good stand-in for words that have not been seen at all in unseen text. You do not have to modify this function.

Now take a look at the `__init__` method of TrigramModel (the constructor). When a new TrigramModel is created, we pass in the filename of a corpus file. We then iterate through the corpus *twice:* once to collect the lexicon, and once to count n-grams. You will implement the method to count n-grams in the next step.

Counting n-grams

Now it's your turn again. In this step, you will implement the method `count_ngrams` that should count the occurrence frequencies for ngrams in the corpus. The method already creates three instance variables of *TrigramModel*, which store the unigram, bigram, and trigram counts in the corpus. Each variable is a dictionary (a hash map) that maps the n-gram to its count in the corpus.
For example, after populating these dictionaries, we want to be able to query

```
>>> model.trigramcounts[('START','START','the')]

5478

>>> model.bigramcounts[('START','the')]

5478

>>> model.unigramcounts[('the',)]

61428
```

Where *model* is an instance of *TrigramModel* that has been trained on a corpus. Note that the unigrams are represented as one-element tuples (indicated by the , in the end). Note that the actual numbers might be slightly different depending on how you set things up.

Part 3 - Raw n-gram probabilities (10 pts)

Write the methods `raw_trigram_probability(trigram)`, `raw_bigram_probability(bigram)`, and `raw_unigram_probability(unigram)`.

Each of these methods should return an unsmoothed probability computed from the trigram, bigram, and unigram counts. This part is easy, except that you also need to keep track of the total number of words in order to compute the unigram probabilities.

Interlude - Generating text (OPTIONAL)

This part is a little trickier. Write the method `generate_sentence`, which should return a list of strings, randomly generated from the raw trigram model. You need to keep track of the previous two tokens in the sequence, starting with ("START","START"). Then, to create the next word, look at all words that appeared in this context and get the raw trigram probability for each.
Draw a random word from this distribution (think about how to do this -- I will give hints about how to draw a random value from a multinomial distribution on Piazza) and then add it to the sequence. You should stop generating words once the "STOP" token is generated. Here are some examples for how this method should behave:

```
model.generate_sentence()
['the', 'last', 'tread', ',', 'mama', 'did', 'mention', 'to', 'the', 'opposing', 'sec
tor', 'of', 'our', 'natural', 'resources', '.', 'STOP']

>>> model.generate_sentence()
['the', 'specific', 'group', 'which', 'caused', 'this', 'to', 'fundamentals', 'and',
'each', 'berated', 'the', 'other', 'resident', '.', 'STOP']
```

The optional t parameter of the method specifies the maximum sequence length so that no more tokens are generated if the "STOP" token is not reached before t words.

Part 4 - Smoothed probabilities (10 pts)

Write the method `smoothed_trigram_probability(self, trigram)` which uses linear interpolation between the raw trigram, unigram, and bigram probabilities (see lecture for how to compute this). Set the interpolation parameters to lambda1 = lambda2 = lambda3 = 1/3. Use the raw probability methods defined before.

Part 5 - Computing Sentence Probability (10 pts)

Write the method `sentence_logprob(sentence)`, which returns the log probability of an entire sequence (see lecture how to compute this). Use the `get_ngrams` function to compute trigrams and the `smoothed_trigram_probability` method to obtain probabilities. Convert each probability into logspace using `math.log2`. For example:

```
>>> math.log2(0.8)
-0.3219280948873623
```

Then, instead of multiplying probabilities, add the log probabilities. Regular probabilities would quickly become too small, leading to numeric issues, so we typically work with log probabilities instead.

Part 6 - Perplexity (10 pts)

Write the method `perplexity(corpus)`, which should compute the perplexity of the model on an entire corpus.
Corpus is a corpus iterator (as returned by the corpus_reader method).
Recall that the perplexity is defined as 2^{-l}, where l is defined as:

$$l = \frac{1}{M} \sum_{i=1}^{m} \log p(s_i)$$

Here M is the total number of words. So to compute the perplexity, sum the log probability for each sentence, and then divide by the total number of words in the corpus.

Run the perplexity function on the test set for the Brown corpus brown_test.txt (see main section at the bottom of the Python file for how to do this). The perplexity should be less than 400. Also try computing the perplexity on the training data (which should be a lot lower, unsurprisingly).
This is a form of intrinsic evaluation.

Part 7 - Using the Model for Text Classification (10 pts)

In this final part of the problem we will apply the trigram model to a text classification task. We will use a data set of essays written by non-native speakers of English for the ETS TOEFL test. These essays are scored according to skill level low, medium, or high. We will only consider essays that have been scored as "high" or "low". We will train a different language model on a training set of each category and then use these models to automatically score unseen essays. We compute the perplexity of each language model on each essay. The model with the lower perplexity determines the class of the essay.
The files **ets_toefl_data/train_high.txt** and **ets_toefl_data/train_low.txt** in the data zip file contain the training data for high and low skill essays, respectively. The directories **ets_toefl_data/test_high** and **ets_toefl_data/test_low** contain test essays (one per file) of each category.

Complete the method `essay_scoring_experiment`. The method should be called by passing two training text files, and two testing directories (containing text files of individual essays). It returns the accuracy of the prediction.
The method already creates two trigram models, reads in the test essays from each directory, and computes the perplexity for each essay. All you have to do is compare the perplexities and the returns the accuracy (correct predictions / total predictions).

On the essay data set, you should easily get an accuracy of > 80%.

What you need to submit

trigram_model.py
written.pdf or written.txt
Do not submit the data files.
Pack these files together in a .zip or .tgz file as described on top of this page.