Assignment 2

Due: March 8, Midnight

Overview

In this assignment, your primary goal is to implement unigram and bigram language models and evaluate their performance. You'll use the equations from Chapter 3 of SLP (https://web.stanford.edu/~jurafsky/slp3/3.pdf); in particular you will implement maximum likelihood estimation (equations 3.11 and 3.12) with add-k smoothing (equation 3.25), as well as a perplexity calculation to test your models (equation 3.16, but explained more in this document and skeleton code). This assignment is heavily inspired by assignment from Rob Voigt's Computational Ethics for NLP course.
The skeleton code for this assignment is: `language_modeling.py`. It will run your model on some toy data (Sam I Am) accompanying the assignment and report its performance. All resources are available at: Ass2_NgramLM_NLPSpring2024

A Note on Object-Oriented Programming

Some of you may not be familiar with the idea of object-oriented programming, or how it plays out in python. You don't need to go deep into the details, but if you're interested in getting more detail on what's going on, you can start with Chapters 15-17 of Think Python (https://greenteapress.com/thinkpython2/thinkpython2.pdf).

For the purposes of this assignment, notice that there is a large structure at the top level (leftmost indentation) that is defined with a keyword `class`. This is us making a definition of a new type of object, an `NgramLanguageModel`. Doing so allows us to associate all the various data (for instance, counts from a corpus) and functions (for instance, to accumulate those counts or produce a probability) with a given "instance" of that object in a persistent manner.

Once the class is defined, we can produce an instance as follows:

```
ngram_lm = NgramLanguageModel()
```

The parenthesis on the end look like a function call, and that's because they are - specifically a special "constructor" function that creates an object of the `NgramLanguageModel` type. The above `ngram_lm` now contains an instance of that object, setup according to the special `__init__(self)` function at the top of the class (e.g., it has the `*_counts` dicts and the `k` value set).

A note on `self` : this is a special self-referential keyword variable, by which a class can reference the variables (== attributes of the class) and functions (== methods of the class) it contains from inside itself. In this assignment, you'll be updating the `self.unigram_counts` and `self.bigram_counts` variables - you have to use `self.` as a prefix to access them.

This is all for your information - the object-oriented setup is done for you in the skeleton code, your job is to modify the functions inside the class to do the various things we need to do.

Your Job

There are a number of ways to build a language model; the skeleton code provides a basic interface which I'd like you to stick with primarily, except for the changes required for given extension (below).

For this assignment the main things you need to do are to write the `train`,`predict_unigram`, `predict_bigram`, and `test_perplexity` functions.

`train`

The way it's set up, the `train` function need only accumulate counts, the resulting probabilities can be calculated at test time in the `predict_*` functions. This is slower at test time though, so one simple extension is to modify this (see section extension).

You can expect the training corpus to contain one sentence per line, already tokenized, so you can split it up on whitespace (e.g., `sentence.split()`). Important reminder that you must add `<s>` tokens to the beginning and `</s>` tokens to the end of every sentence.

`predict_unigram` ,`predict_bigram`

These functions should take a sentence (as a string), split it into tokens on whitespace, add start and end tokens, and then calculate the probability of that sentence sequence using a unigram or bigram language model, respectively.

Notes:

Use add-k smoothing in this calculation. This is very similar to maximum likelihood estimation, but adding `k` to the numerator and `k * vocab_size` to the denominator (see Equation 3.25 in the textbook).
Return log probabilities! As talked about in class, we want to do these calculations in log-space because of floating point underflow problems. Do this per word! Even a long sentence could easily get us to an underflow. So create a float that starts at 0.0, and for each word where you get the probability, do `+= math.log(prob)` onto that float variable. Return this sum in the end.

`test_perplexity`

This function takes the path to a new corpus as input and calculates its perplexity (normalized total log- likelihood) relative to a new test corpus.

The basic gist here is quite simple - use your `predict_*` functions to calculate sentence-level log probabilities and sum them up, then convert to perplexity by doing the following:

`math.exp(-1 * total_log_prob / N)`
Where `N` is the total number of words seen. There are some additional details given in the function docstring in the skeleton code.

At a minimum, your assignment should run, be able to train and test the models, and report respective perplexities.

Extension: 5% Bonus

If you manage to finish early, why not try this simple extension for an extra 5% score?

Speed up test time performance. You can change the structure of the NgramLanguageModel class to calculate all possible probabilities at training time, which will substantially increase the speed of test- time inference since that will become a matter of looking up log probabilities and adding them up rather than performing the calculation each time a given n-gram appears.

Compare and report the performance hike.

Code quality:

Please cite all sources, if you repurpose from any, with a description of what that code is doing and why was it helpful.

All code must be well-documented with comments; the purpose of each block of code should be made clear.

Submission

Upload in Canvas your completed `language_modeling.py` file, along with a ReadMe file for users. The Readme should briefly describe the project and specifically mention the command/s to run it on user end.