CS 162 Homework 2 Coding

N-Gram Language Models

N-Gram Language Models are language models used to generate sequences of words based on the likelihood of that sequence appearing in the training corpus. The term n-gram refers to the a continuous sequence of n tokens for a given sample of text. For token to token prediction, the model estimates the likelihood of observing a particular word w_i given its context w_{i-1} where the length of the context is depending on the n of the n-gram model.
For this homework assignment, you will be implementing a N-Gram Language Model with Jelinek-Mercer Backoff Smoothing. You can find the starter code here: CS162_W24_homework_2

1 N-Gram Construction

We provide you with a training and development text files.

Coding Task 1 (4 points): Locate language_model.py in your starter code and implement the _get_ngrams() function in which you will construct the n-gram tuples your model will use.

2 Perplexity

Perplexity is a measure of how well a language model predicts a given sequence of words and captures the uncertainty of a generated sequence. Perplexity is calculated as

$$Perplexity = 2^{\frac{-1\sum_{i=1}^{N} log_2(P(w_i|...))}{N}}$$

where N is the count of the n-gram corpus and $P(w_i|...)$ is the conditional likelihood of a given word w_i based on the previous n words in the sequence.
One of the well studied issues with n-gram models is unseen n-gram. If a model encounters an n-gram that is not present in the training corpus, without smoothing the likelihood of that n-gram is 0 which causes the likelihood of that entire sequence to be 0. Smoothing for n-gram language models addresses unseen n-grams by assigning non-zero probability the model to assign some likelihood to these unseen example and therefore allow for novel events in the generation process. There are many methods of smoothing, and you will will be implementing two approaches, **Add-One** and **Jelinek-Mercer** smoothing.

2.1 Add-One Smoothing

Add-One Smoothing, also known as Laplace smoothing, is a smoothing technique that adds a small constant of one to the count of each event for both the numerator and denominator to ensure unseen events are still likely for the model to generate. The calculation is as follows

$$P_{add-1}(w_i|w_{i-1}) = \frac{c(w_{i-1}, w_i) + 1}{c(w_{i-1}) + |V|}$$

where V is the vocabulary. The generalized version of Add-One smoothing, is Add-k smoothing, in which you have a hyperparameter k the controls how much weight you want to add to an unseen instance. The generalized calculation is as follows

$$P_{add-k}(w_i|w_{i-1}) = \frac{c(w_{i-1}, w_i) + k}{c(w_{i-1}) + (|V| * k)}$$

2.2 Jelinek-Mercer Smoothing

Many common smoothing approaches like additive smoothing and Good-Turing smoothing make novel or unseen events too probable. To address this, the Jelinek-Mercer and other backoff smoothing technique were created to ensure that all unseen n-grams are not treated the same.
Jelinek and Mercer proposed a method of smoothing that interpolates between an n-gram and lower-order n-grams to estimate the likelihood of an unseen n-gram. It uses weights hyperparameter, λ, to control the interpolation. In a tri-gram model using Jelinek-Mercer smoothing, the probability of an unseen example using interpolation is

$$p_{interp}(w_i|w_{i-2}w_{i-1}) = \lambda_3 p(w_i|w_{i-2}w_{i-1}) + \lambda_2 p(w_i|w_{i-1}) + \lambda_1 p(w_i)$$

For higher-order n-gram language models with Jelinek-Mercer smoothing, it is defined as the linear interpolation between the n-order and (n-1)-order n-gram model is given as

$$p_{interp}(w_i|w_{i-(n-1)}...w_{i-1}) = \sum_{j=1}^{n} \lambda_j p(w_i|w_{i-(j-1)}...w_{i-1})$$

in which $\sum_{i=1}^{n} \lambda_1 = 1$ and all are ≥ 0. The weights are a hyperparmeter that we optimize on.

Coding Task 2 (10 points): Complete the sent_perplexity() function by implementing sentence level perplexity. Implement Jelinek-Mercer Backoff Smoothing for only tri-gram models when backoff smoothing is turned on. If backoff smoothing is turned off or if $n! = 3$, both no smoothing and implement Add-k Smoothing in which $k = 1$.

Coding Task 3 (4 points): Complete the corpus_perplexity() function by implementing corpus level perplexity using the previously defined sent_perplexity() function.

3 Greedy Search Generation

Greedy search generation for n-gram language models iteratively selects the most likely word at each step based on the probabilities estimated by the mode. For your model, you will use n-gram frequency to decide the next word in the generated sequence.

Coding Task 3 (2 points): Complete the greedy_search() function using the training frequency dictionary.