NLP 201: Assignment 2

Ethan Sin

November 25, 2023

1 Introduction

This report will cover the three parts of our assignment, including the findings from the programming assignment in parts one and two, and the experimentation with GPT-3 in part three. The first two parts of the assignment involve building and observing the qualities of an n-gram language model. The description of the code will be mostly left within comments of the file, and this report will focus more on the process of building and what was learned from that process.

2 Part 1

Part 1 of the assignment contained the majority of the programming. The task was to create three n-gram language models without implementation of a smoothing method. The three were to be unigram, bigram, and trigram based models. The end goal is to train the model to estimate the probability of any given corpus. This would be measured by taking the probability of each n-gram in the corpus to calculate perplexity.

2.1 The Model

At the center of my model is a three-layer nested dictionary that allows the model to easily access the count of any relevant n-gram. The top layer is the count of every unigram stored in a special key named '<count>', and along with that within each unigram is the second layer with the counts of all words that follow each given unigram and their counts in that context. The final layer is the similar, except each word only contains counts as we are only counting up to trigrams. Before the words are added to this dictionary, their total occurrences in the corpus is calculated and they are turned into a special '<unk>' token as per the assignment instructions. With this implementation, finding the probability of a trigram like 'I drink coffee' is as simple as taking model.gram_counts['I']['drink']['coffee'] and dividing it by model.gram_counts['I']['drink']['<count>'].

Within the model's class also contains functions to easily access different degree n-gram probabilities, the n-gram counts, and to covert tokens that are not seen in the corpus the model is estimating into '<unk>' before finding the counts. Outside of the model are the rest of the math functions that are required to calculate the model's perplexity on a corpus. I first created a maximum likelihood estimate function that simply would calculate

the likelihood of each n-gram in an utterance and add them all together for perplexity to be calculated. Since it would return one number, I had converted each probability into a log probability before summing them together. As this function already summed the log probabilities, the perplexity function only had to take the product of that sum and -1/N where N is the total number of tokens in the input ignoring start tokens.

Since my implementation is better suited for calculating individual sentences, instead of flattening the corpora to calculate their perplexities, I wrote a function that would multiply each sentence in the corpus by a weight that is the proportion of the sentence's length compared to the length of the entire corpus minus the start tokens. This function is also where a corpus' perplexity can return as infinity. This will occur if in any point in the calculation a KeyError is returned, likely meaning that the model is trying to calculate the probability of a bigram or trigram that does not exist in the train set.

As there are no hyperparameters to consider in this part, the only tuning that was needed was to match the sample perplexities that were given in the assignment instructions. I was able to match all of these perplexities exactly, and a block of code is left in the file to show that they do match.

2.2 Experimentation and Conclusions

For this part, the main experimentation is to use the unigram, bigram, and trigram models to calculate perplexities on the train, validation, and test sets. The results I achieved are displayed here:

Dataset	Unigram	Bigram	Trigram
Train Set	1059.06	77.94	6.95
Validation Set	966.88	inf	inf
Test Set	974.27	inf	inf

Table 1: Perplexities

There are several key observations that can be made from this data. The first is that the perplexities clearly get significantly lower as the degree of the n-gram becomes larger. This is expected, as a unigram model can only give a probability of a single word occurring against the count of all the words in the train corpus, while a model that takes into account knowing a word that comes before it can calculate a probability that is conditional on that knowledge. For example, assume that in the 'I like coffee' example: 'coffee' occurs 5 times in the whole corpus, 'like' occurs 20 times, and the size of the corpus is 10,000 words. In this case, a unigram model can only tell you that the likelihood of 'coffee' is 5/10,000. A bigram model would be able to narrow that probability down to as high as 1/20 if out of those 5 times the model saw 'coffee', even just one of them occurred after 'like'. (Jurafsky and Martin, 2019)

Another observation is that bigram and trigram models give a perplexity of infinity on any set that is not the train set. This is also to be expected, as even though our model does some coverage of unknown tokens by converting tokens with 3 or less occurrences to '<unk>', there are still many occurrences of bigram or trigram sequences where a token that occurred 3 or less times was not present in that sequence. Thus, the model would

have no count for that sequence and therefore not be able to give a probability at all, giving a perplexity of infinity.

The final observation I made is that the perplexity of the unigram model on the validation set and test set are lower than the train set. This was unexpected, as intuitively I would think that a model would get a better perplexity on data that it has seen every data point of. However, after thinking about this result it began to make sense. If we are converting tokens to '<unk>' and having them all contribute to the same count, the unigram count for '<unk>' should end up being fairly high comparatively to many other words, especially taking into account Zipf's law. Additionally, since I had converted all tokens in the input corpus that were not found in the train set to '<unk>', there are bound to be many more tokens that are changed to '<unk>' in the validation and test set, as our train data has not seen many of the words that exist in these other sets. As such, the unigram model will be slightly less confused when seeing a sentence as it expects to see '<unk>' as more likely than most of the other words that it would recognize.

3 Part 2

Part 2 introduces a way to address some of the issues found with the models in part 1. The task is to implement smoothing with linear interpolation into the model and experiment with this method's hyperparameters to improve the model. The idea is to use all three models from part 1 in tandem with each other to calculate the probability of a corpus and therefore its perplexity as well. The way it works is by calculating the probability of each word by taking the sum of its probability from each n-gram model, with each individual probability multiplied by a set of weights manually determined (denoted as lambda) that sum to 1. The task is then to use this model to experiment the effects on perplexity for different combinations of weights, different training data sizes, and different parameters of which tokens to change into '<unk>'.

3.1 The Model

This is a fairly simple addition to the models that were built in part 1. As the lambda weights are applied at the calculation of each individual probability, the functions in part 1 were slightly modified. The function to calculate the probabilities of a sentence now returns a list of the probabilities of each word with the lambda weights applied, and a separate function was made to turn each probability in the list into a log probability and then sum it up. In addition, a KeyValue error would have the count function in the model return 0 rather than make the perplexity return as infinity now. The rest of the model remains mostly the same. It successfully matches the given perplexity sample for debugging.

3.2 Experiment 1

The first experiment is to try different combinations of lambda for the 3 probabilities on the validation set to inform which combination should be used on the test set as a final evaluation. In this section, I will format these lambda weights as such: [0.1, 0.3, 0.6], where the weight for unigram, bigram, and trigram probabilities will be displayed

respectively. As per the assignment instructions, I tested only 5 different combinations of weights on the validation set. I started with the combinations [0.1, 0.3, 0.6] and [0.3, 0.3, 0.4] and adjusted based on patterns of the perplexities that I saw. Here are my results:

Dataset	[0.1, 0.3, 0.6]	[0.3, 0.3, 0.4]	[0.4, 0.3, 0.3]	[0.2, 0.3, 0.5]	[0.3, 0.4, 0.3]
Validation	104.3	93.57	94.6	95.88	88.13

Table 2: Lambda Weight Tests on the Validation Set

The main finding here that I found is that weighing the bigram higher than the unigram or trigram led to the best result. Initially, the weights were adjusted with little understanding of how they would end up affecting the model and I tried different values raising and lowering the extreme ends of the weights. Once I saw that giving more weight to the bigram yielded notably improved results, I began to understand what was going on. I realized that each n-gram probability contributed something different to the model. The unigram acts as the agent of 'smoothing' the model, allowing any word to have some probability even if it does not have any probability as a bigram or a trigram. The weakness of the unigram is that on its own it cannot give a conditional probability and will therefore be a very low probability with a high perplexity. The trigram is the other end of the spectrum, giving a much more informative conditional probability seeing the previous two words. The problem with the trigram is that it is much more likely to retrieve a probability of 0. The bigram gives a balance between them, having more of a likelihood to not return 0 while still giving a more informative conditional probability for a larger amount of data. To illustrate the difference in how much data the trigram will find a 0 probability for compared to the bigram, I took the percentage of sentences in the validation set that had a perplexity of infinity from the models created in part 1. The bigram model had 0.884 of sentences as infinite perplexity, and my trigram model had 0.998 of sentences with infinite perplexity.

I then used my model with the weights [0.3, 0.4, 0.3] to calculate the perplexity on the test set, and the result was 87.83. If I could test more, I would like to see what a very high lambda on bigrams with low weights on unigrams and trigrams would result in.

3.3 Experiment 2

This experiment was to train the model on only half the dataset and compare the new perplexities. Before experimenting, there are reasons for the model to result in both a higher or lower perplexity. It could be lower because the number of '<unk>' would be higher, and the number of tokens converted to '<unk>' would be higher. Each token that is seen as '<unk>' would then give a slightly higher probability, and more tokens will have that higher probability. On the other hand, perplexity could be higher because many of the bigrams and trigrams that were seen due to the deleted half of the training set will no longer be seen. Many more bigrams and trigrams will then result in a probability of 0. The experiment appears to support the hypothesis that the missing bigram and trigram counts affect the perplexity more, and that having a smaller training dataset will result in higher perplexities. The result with the half training dataset is a perplexity of 98.79 on the test set compared to the 87.83 perplexity with the full training dataset.

3.4 Experiment 3

This final experiment is to train the model where the training data is preprocessed to make '<unk>' tokens at different thresholds. The original threshold is below 3 occurrences overall, and the tested thresholds will be below 5 occurrences and exactly 1 occurrence. These are the results:

Dataset	< 3	< 5	$== 1$
Test Set	87.82	70.59	110.25

Table 3: Test Set Perplexities with different '<unk>' Thresholds

It appears that the generation of more '<unk>' tokens lead to lower perplexities. This makes sense in tandem with the discussion in part 1 of the unigram perplexities becoming lower on unseen datasets. This leads to both more '<unk>' tokens in the datasets and more '<unk>' counts in unigrams, bigrams, and trigrams. That means more tokens will receive the relatively high likelihood of the '<unk>' token, and that already high likelihood will become even higher as the number of '<unk>' tokens become proportionally larger compared to the whole dataset. This gives a valuable implication. A lower perplexity is not necessarily a more useful model. It only serves to tell the model to not be so confused when it sees something it does not recognize. It is then contextual to determine whether or not that is what the model should do.

4 Part 3

In this last part of the assignment, the task is to test GPT-3 models with zero-shot prompting and few-shot prompting, and compare the performance. Then, compare the performances with other models available on the site and the performance with different tasks. I will report my findings in the following sections.

4.1 "text-davinci-002"

For the prompt given for our zero-shot prompting example, the model gave me the correct answer for "Who succeeded Joseph Willard as president?". I also tried asking "Who died in 1803?" and "What was Samuel Webber's ideology?", which both yielded correct answers. These, however, were all in the form of a complete sentence. For example, "Samuel Webber succeeded Joseph Willard as president". With the few-shot prompting example, the model would start giving brief, concise answers, imitating the examples given.

4.2 "davinci"

I repeated the above prompts using the older "davinci" model. For zero-shot prompting, it would still give the correct answer but it would continue to generate a new passage afterward and eventually only continue to build on that new passage until the limit was reached. With few-shot prompting, the model would still give the correct answer, but continued to generate more questions and attempted to answer them. It seemed that it

stayed on task more by giving examples of the task, but it did not know when the task was over.

4.3 Sentiment Analysis

I then gave "text-davinci-002" my own prompt, starting the prompt with "Read this review and determine its sentiment", then copy pasting the body text of a review from IMDb, then writing "Sentiment:". It gave the correct answer but also went on to summarize the review. Another submission of the same prompt went on to pull key phrases from the review along with the sentiment. I then changed the prompt to be few-shot with one example, with the example only stating that the review is positive and nothing else. The model would then mimic it and give me only "Positive" as an answer. When I did the same experiment with "davinci", it first gave me a tag "featured trending" before giving me the sentiment. Then, it seemed to just generate more text that would be likely to be on an IMDb page, completely irrelevant to the review. When I submitted the same prompt again, it generated irrelevant information and did not ever give a sentiment. With few-shot prompting, it gave "????" as the sentiment and then began to summarize various movie characters that are tangentially related to the review. It then gave information about an explicit movie that has no relevance and should not be shared. I ran the same few-shot prompt with similar results. Only the first submission actually contained an accurate sentiment label.

References

Jurafsky, D. and Martin, J. H. (2019). *Speech and Language Processing.* Stanford University Press.