Indian Institute of Technology Gandhinagar

NLP Assignment-2
Team - Cloud 8

CS613 Assignment-2 Tasks and Results
18th September 2023

Authors

Aditi Agarwal 20110006
Harshvardhan Vala 20110075
Inderjeet Singh 20110080
Joy Makwana 20110086
Kalash Kankaria 20110088
Kanishk Singhal 20110091
Medhansh Singh 20110111
Nokzendi Aier 20110173

Under the guidance of
Professor. Mayank Singh

Tasks

Gitlink: [CS613-NLP/assignment-2]

Task 1-4. Preprocessing

The code for preprocessing Reddit comments ensures their cleanliness and suitability for further analysis. It employs NLTK for text processing and leverages parallel processing for efficiency.

We first created a preprocessing function, within the custom preprocessing function, each comment undergoes link and special character removal, followed by additional cleaning steps such as text pattern removal, whitespace stripping, and lowercase conversion.

- The distinguishing feature is the addition of three <s> tags at the start and three </s> tags at the end of each sentence within the comments. This format aids in sentence segmentation.
- After the preprocessing, we found out that there are some unnecessary words like “x000D”, “x200B” and also the comments which only had a hyperlink are storing null values after the processing.
- We removed the unnecessary words and empty processed comments from the dataset.
- Processed sentences are compiled, forming a new dataset stored in a CSV file.

We finally are splitting the dataset of processed comments from the CSV file into two subsets: a training set and a validation set, in an 80:20 ratio.

5. N-Grams, Probabilities and Perplexity

The code was structured into a Python class that takes a DataFrame as input, where each row represents a sentence. This class allows users to specify the value of 'n' to create n-grams, making it adaptable to different language modelling tasks. This class is implemented in `NGramProcessor.py`.

The workflow designed from user perspective is as follows:

1. The users gets the preprocessed dataset from the preprocessing pipeline.
2. Then we can define a ngram as the object of class `**NGramProcessor**` which takes training dataset and `n` as the input.
3. Then the '**train**' function of the class has to be called to populate the two (one in case of unigram) frequency dictionary which stores the ngram and their frequencies(counts).
4. To calculate the probability of an n gram in a training dataset, `**find_probability**` function is used which uses the frequency dictionary defined earlier to calculate the log probability of each n gram and also saves them to a csv file. *We do not need this to call this function to calculate perplexity though.*
5. We can then calculate the perplexity of a dataset with the `**calc_perplexity**' function which takes the dataset as input and calculates the log probability of each ngram in the input dataset for the function. It uses those probabilities to calculate the perplexity of each sentence and the average perplexity of the ngram and saves both to separate files.

There are also many optimizations in place to make the computation faster and to indicate the progress of computation in the terminal. The code is also very modularized with a function to find N-grams or perplexity of individual sentences for testing purposes. These functions can be reused again making the code more efficient and readable.

To calculate the average perplexity we first used the test data but as the train data is not a good representation of test data, there were sentences which had words which were never seen in train data. This caused the perplexity to become infinite. We then tried

finding perplexity on the train data itself and obtained results which matches the hypothesis that perplexity should decrease with unigram to quadgram.

Model	Perplexity
Unigram	2147.91
Bigram	124.35
Trigram	22.9
Quadgram	12.96

Table 1: *Average Perplexity of the **training dataset without smoothing***

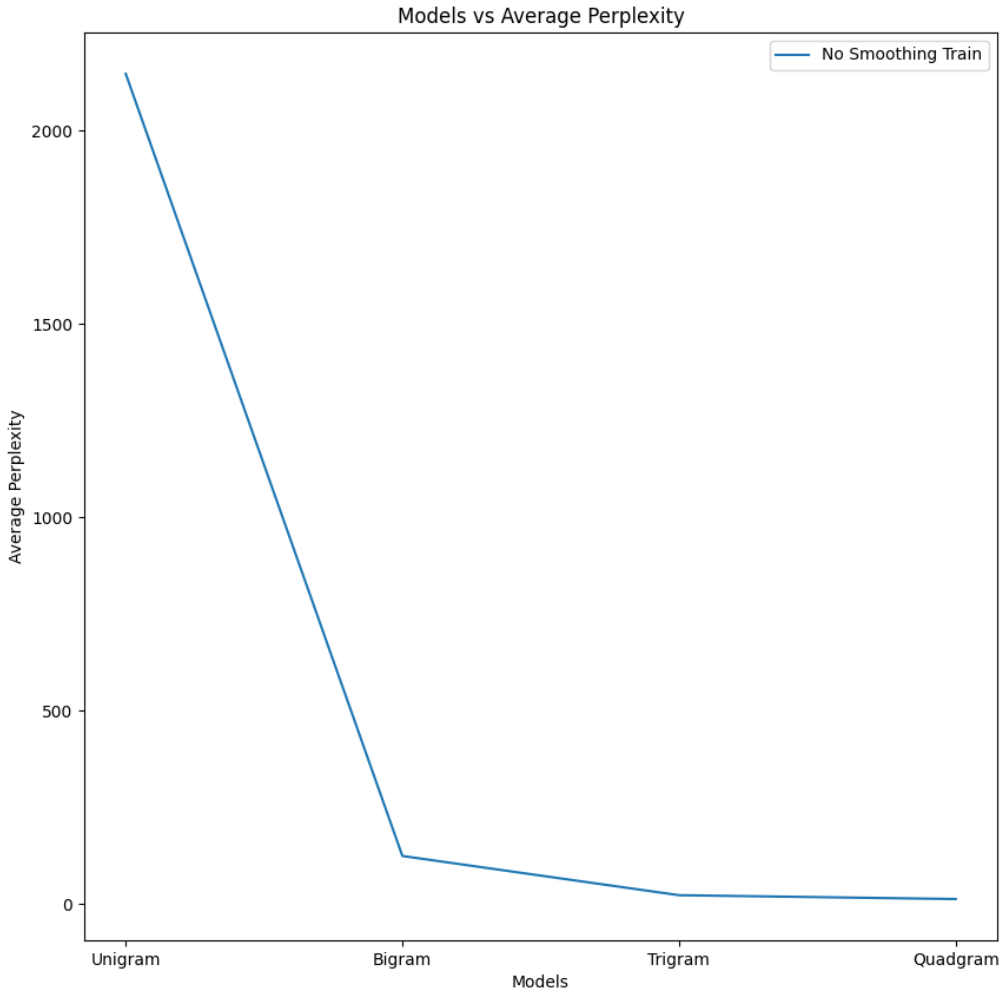

Figure 1: Average perplexity v/s NGram models without smoothing

6. Laplace Smoothing

The following is the tables and the graphs for the perplexity before and after using Laplace:

Model	Perplexity	Laplace Perplexity
Unigram	inf	2097.573
Bigram	inf	5464.032
Trigram	inf	21618.71
Quadgram	inf	32509.49

*Table 2: Average Perplexity for **test set***

The perplexity for test set without smoothing comes out to be infinity to lack of representation test data in train data. As train data does not have many n grams which occur in test data, majority of test data is unseen and the perplexity comes out to be infinite.

Model	Perplexity	Laplace Perplexity
Unigram	2147.912	1960.549
Bigram	124.3488	4059.216
Trigram	22.89613	13687.37
Quadgram	12.95586	19006.71

*Table 3: Average Perplexity for **train set***

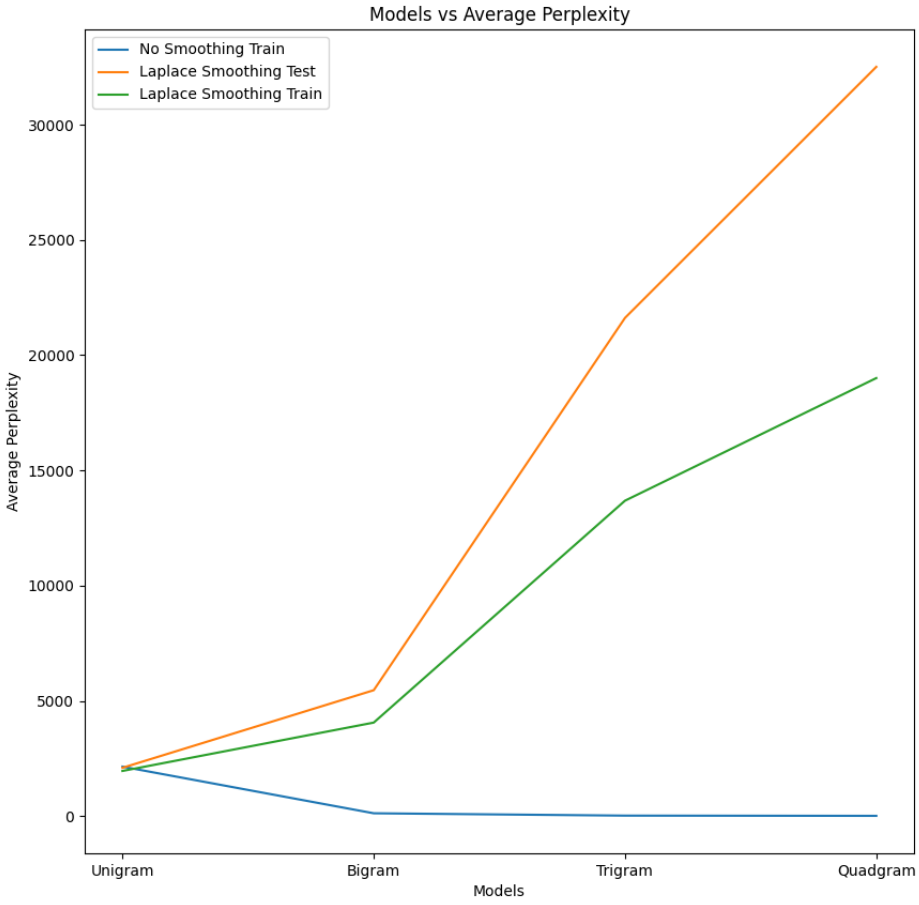

Figure 2: *Laplace Perplexity vs No Smoothing Perplexity on Training dataset*

The plots for Perplexity with no smoothing for the test set are not plotted as it is infinity. The changes have been compared and justified in the next section below.

7. Explaining Perplexity for Laplace Smoothing

Perplexity is a measure commonly used to evaluate the quality of language models, including N-gram models. It provides an estimate of how well the model can predict unseen or held-out data. A lower perplexity indicates that the model is better at predicting the test data.Here are possible justification for our observations of perplexity before and after laplace smoothing:

- Perplexity Before Smoothing:

  When we trained our N-gram model on the test data without smoothing, we encountered infinite perplexity for N-grams with N=1, 2, 3, and 4. This happens when the model assigns zero probability to certain sequences of words in the test data.

  In N-gram models, the problem of zero probabilities arises because they rely solely on the observed training data. If an N-gram in the test data was never seen in the training data, the model assigns a probability of zero, leading to infinite perplexity.

- Perplexity After Laplace Smoothing:

  Laplace (or add-one) smoothing is a technique used to address the zero probability problem. It adds a small constant (equals 1) to the count of each N-gram in the training data ensuring that no N-gram has a zero probability.

  After applying Laplace smoothing, the perplexity values became finite, indicating that the model can now make predictions for all N-grams in the test data.

  However, the perplexity values we observed are quite high for all N-grams, and especially for bigrams, trigrams, and quadgrams. This suggests that the model, even after smoothing, struggles to accurately predict the test data.

But why is perplexity increasing from Unigram to Quadgram after Laplace Smoothing?

- This might be because higher-order N-grams are more specific and have even fewer occurrences in the training data.
- Smoothing helps mitigate the problem of zero probabilities, but it doesn't address the issue of having a lot of missing N-grams (Sparse).
- So, as we move to higher-order N-grams, we have fewer training examples of those specific sequences, making it challenging to estimate their probabilities accurately.

In summary, applying Laplace smoothing made our N-gram model more robust by avoiding infinite perplexity values. However, the perplexity values we observed are still relatively high, and the increase in perplexity from unigram to quadgram is likely due to the increased sparsity and the limitations of Laplace smoothing in handling higher-order N-grams with limited training data. To further improve the model, we below explored more advanced smoothing techniques.

8. Bonus Smoothing

We here used 2 different smoothing techniques

- Good-Turing smoothing
- Additive smoothing

Plots for perplexity of using both of them:

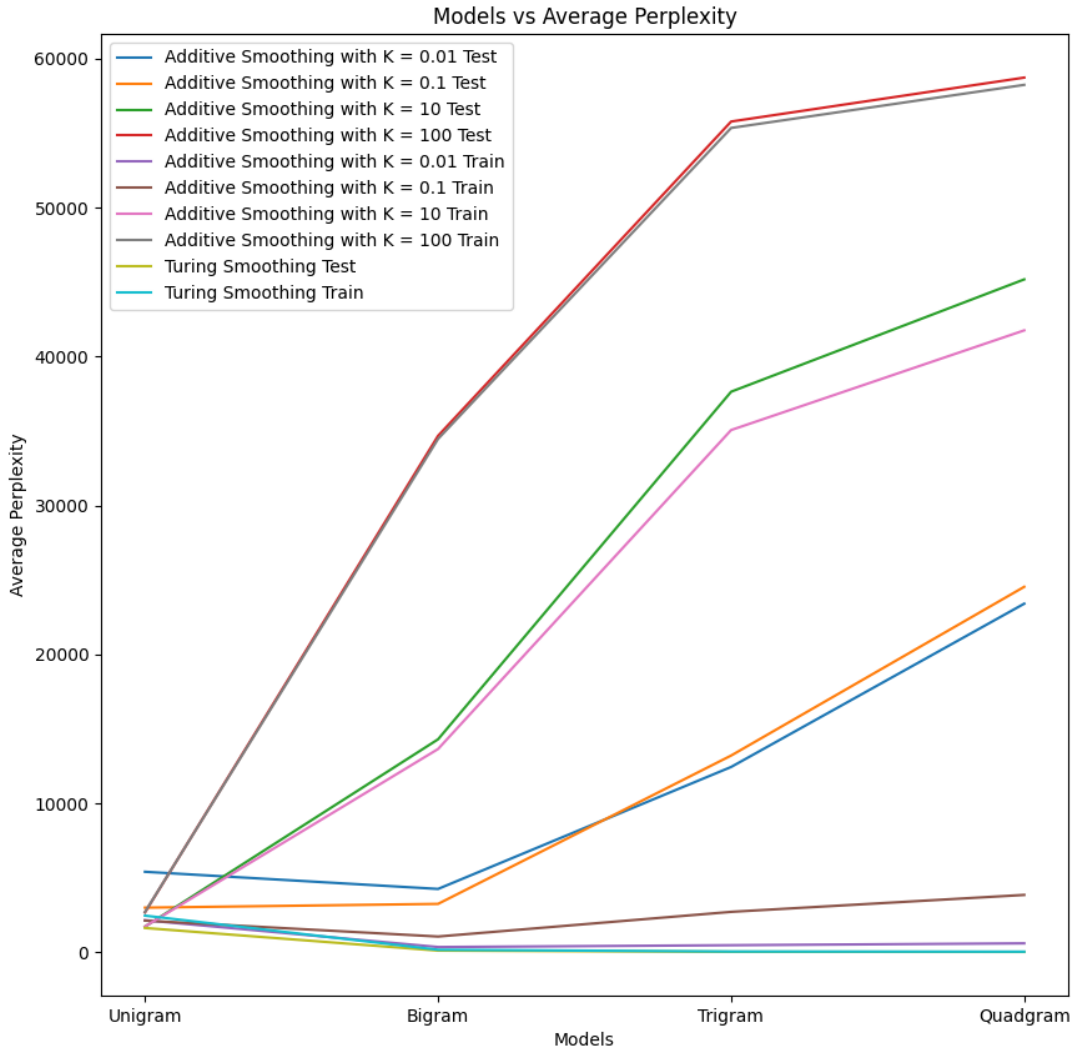

Figure 3: *Good Turing and Additive Smoothing Together*

Here are the observations and justifications of using both of them as smoothing techniques.

Good-Turing smoothing is a technique used to address the zero probability problem in N-gram models. It is known for providing more nuanced and effective smoothing compared to simple Laplace smoothing. Perplexity Plot after Turing Smoothing:

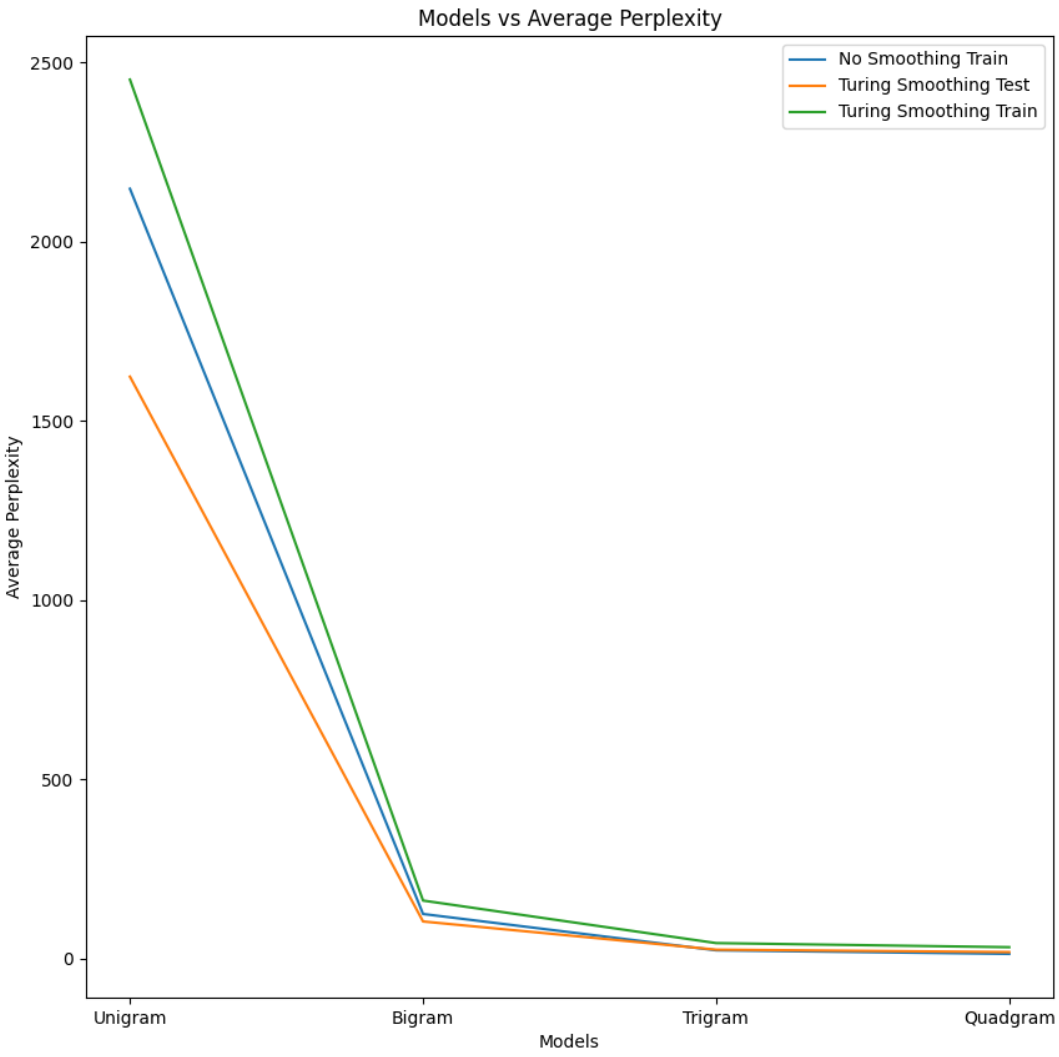

Figure 4: *Good Turing Smoothing vs No Smoothing*

We calculated the frequency of various counts that do not exist using *linear piecewise interpolation*. The scatter plot of the frequency of existing counts and interpolated counts can be seen here:

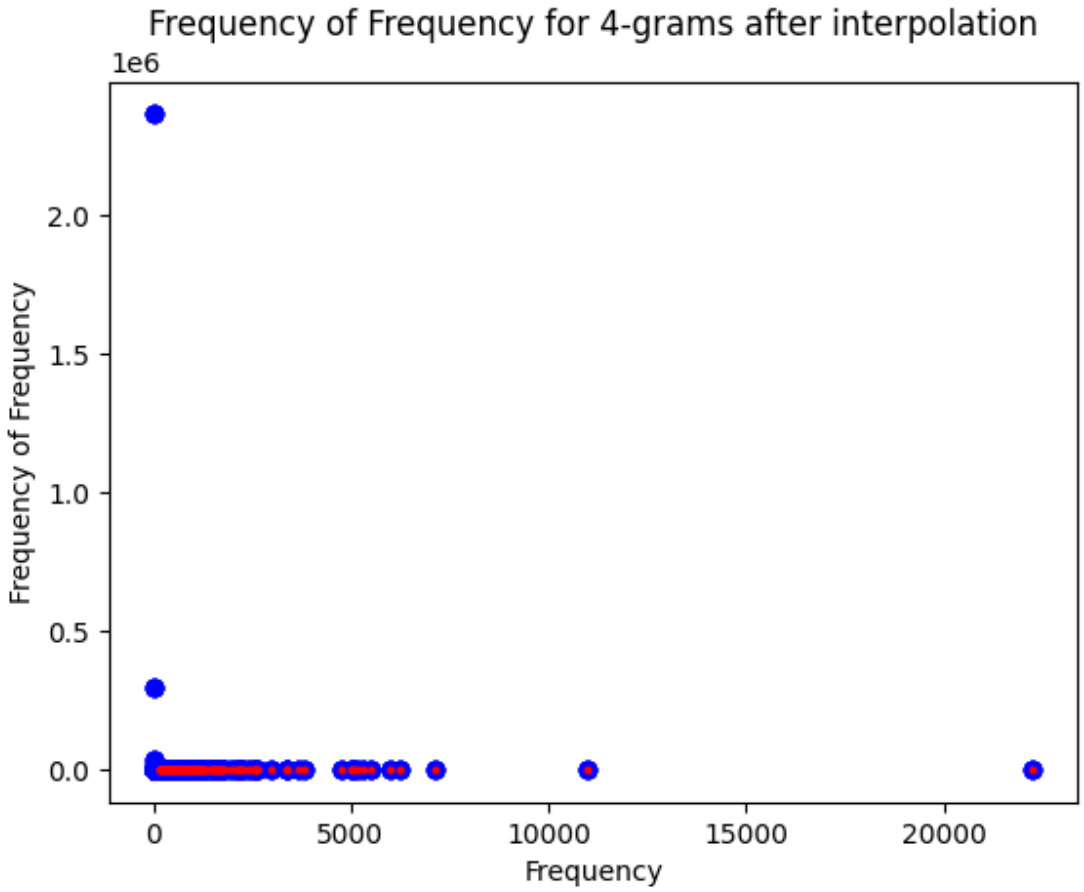

Figure 5: *Frequency of Frequency vs Frequency for quadgram*

Here the blue points are the existing points and the red points are the interpolated points.

Original Count	New Count
11	9.7
12	11.08
13	12.29
14	10.81
15	15.75
16	12.88
17	17.42
18	19.99
19	17.12
20	14.32

We could have simply subtracted a constant (usually 0.75) to get the updated counts instead, but the trend of data shows that using that method would have been quite inaccurate. The counts and their updated counts can be seen in this figure 5. Due to these inconsistencies in the data, we instead performed linear piecewise interpolation.

Figure 6: *Old counts vs New counts in Good Turing*

Our observations of decreasing perplexity with increasing N after applying Good-Turing smoothing can be explained as follows:

- Good-Turing smoothing adjusts the probability estimates of N-grams based on their observed frequencies in the training data.
- It takes into account the fact that some N-grams occur more frequently than others and aims to provide more accurate probability estimates.

Justification of Decreasing Perplexity with Increasing N:

With Good-Turing smoothing, the probability estimates for N-grams are more finely tuned based on their observed counts. This means that after this smoothing the model can retain higher probabilities to N-grams that occur frequently in the training data and lower probabilities to those that occur infrequently.

As we move from unigrams to higher-order N-grams (bigrams, trigrams, quadgrams), the probability estimates become increasingly accurate because Good-Turing smoothing adapts better to the specific patterns in the data.
Lower perplexity values indicate that the model is becoming more confident and accurate in predicting the test data. When you have lower perplexity values, it means that the model's predictions closely match the actual test data, which is a desirable outcome.
The decrease in perplexity from unigram to quadgram indicates that the model is improving in its ability to predict sequences of words as N increases, and it is handling the sparsity problem more effectively compared to Laplace smoothing.

In summary, the decreasing perplexity values you observed with increasing N after applying Good-Turing smoothing suggest that this smoothing technique is working effectively to improve the accuracy of your N-gram language model. It is better at capturing the nuances and patterns in the training data, resulting in more reliable predictions and lower perplexity values for higher-order N-grams.

Additive Smoothing - Additive smoothing, which is a variation of Laplace smoothing, is a technique that adds a constant k to the count of each unique event in the training data. This helps to address the zero probability problem and make the model more robust.

Our Observations:

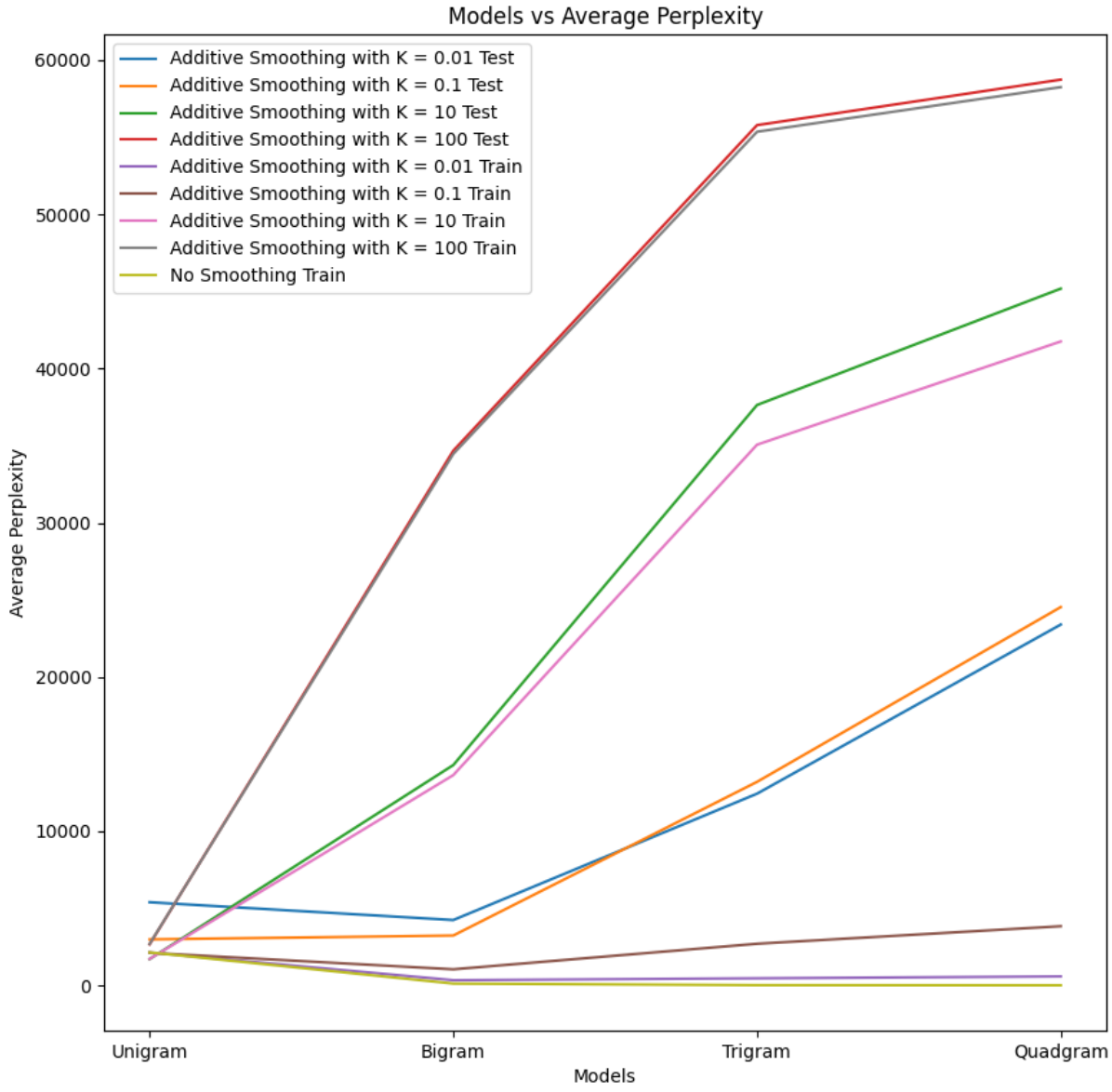

Figure 7: *Additive Smoothing vs No Smoothing*

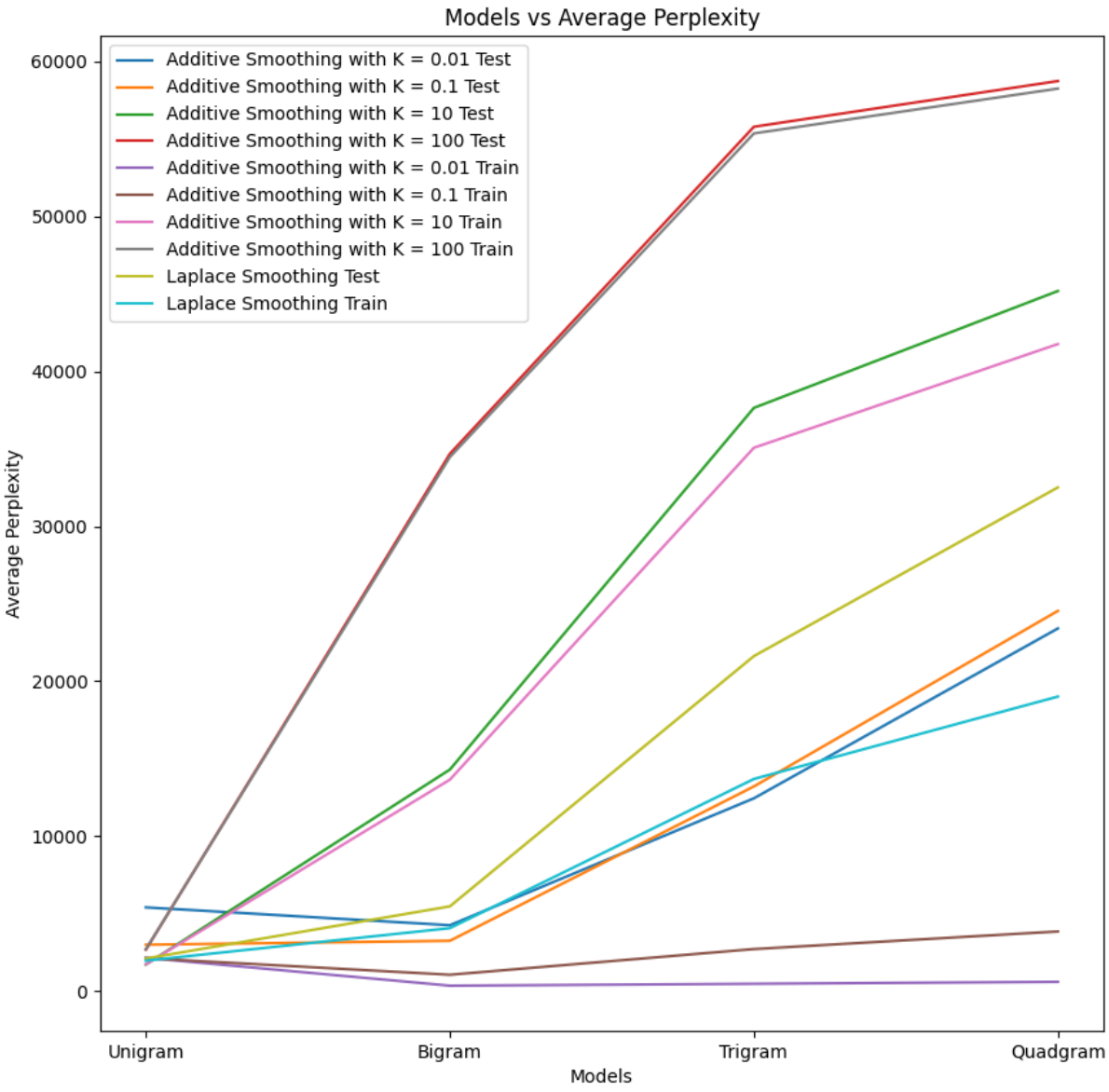

Figure 8: *Additive Smoothing vs Laplace Smoothing*

Our observation that smaller values of k result in lower perplexity, while larger values of k lead to higher perplexity, can be explained as follows:

Effect of Additive Smoothing Parameter (k):

The parameter k in additive smoothing controls the strength of smoothing. Smaller values of k result in more conservative smoothing, while larger values of k lead to more aggressive smoothing.

Lower Perplexity with Smaller k:
Smaller values of k lead to less aggressive smoothing. When we add a small constant like k to the counts, it has a relatively small impact on the estimated probabilities.
With less aggressive smoothing, the model's probability estimates remain closer to the original observed frequencies in the training data. This can result in lower perplexity because the model is essentially relying more on the training data and less on smoothing to make predictions.

Smaller k values tend to preserve the finer details and nuances of the training data, which can be particularly beneficial for capturing the intricacies of language.

Higher Perplexity with Larger k:
As we increase the value of k, smoothing becomes more aggressive. The constant k has a larger influence on the estimated probabilities, causing them to deviate further from the original training data frequencies.
More aggressive smoothing can result in higher perplexity values because the model's probability estimates are less tied to the specifics of the training data and more influenced by the smoothing parameter.

In summary, the observation that smaller values of k lead to lower perplexity while larger values of k lead to higher perplexity is consistent with how additive smoothing works. Smaller values of k result in milder smoothing, which preserves more of the original training data's characteristics and leads to lower perplexity. On the other hand, larger values of k introduce more smoothing, which can oversimplify the model's probability estimates and result in higher perplexity values. The choice of the optimal k value may depend on the specific characteristics of your data and the trade-off between smoothing and fidelity to the training data.

Contributions:

Inderjeet Singh Bhullar: Contributed in all the smoothing techniques specifically. Wrote the code for additional smoothing techniques and found trends in the perplexities achieved after different types of smoothing. Optimised the code for the module at various areas in order to get the code to run faster.

Joy Makwana: My significant contribution to this project involved developing a Reddit comment preprocessing pipeline using NLTK and parallel processing. This pipeline included a custom function to remove links, special characters, and apply text cleaning. I added tags for sentence segmentation and compiled processed sentences into a CSV dataset. After identifying and removing unnecessary words and null-valued comments, I split the dataset into training and validation sets in an 80:20 ratio, ensuring clean and suitable data for analysis.

Kanishk Singhal: Contributed in the implementation of `NGramProcessor` class. Worked on making ngrams, finding probabilities and perplexity of sentences. Adding multiprocessing code in preprocessing and ngram processing for faster execution. Modularized the code for easy readability and usage. Contributed in making the documentation.

Aditi Agarwal: Responsible for developing the basic NGramPreprocesser class which involved implementing all the ngrams from scratch and helper functions such as the function to create frequency dictionary and to split the preprocessed comments for each n-gram. Responsible for modularizing the code from start. Going through the data and making pre-processing decisions on the basis of creating n-grams.

Kalash Kankaria: Worked on the implementation of Laplace smoothing and Additive smoothing and subsequently calculating the perplexity scores after applying smoothing. Furthermore, I compared the various trends obtained after application of smoothing on train and test datasets.

Nokzendi Aier: Worked on implementing smoothing techniques. Plotted the results for the average perplexities, and did the analysis of the trends after smoothing on the train and test dataset. Documented the results and conclusions we found after analysing our model results.

Harshvardhan Vala: Worked on the preprocessing of the dataset by working on cleaning of comments and removal of unnecessary data. Subsequently, worked on the implementation of the additive smoothing and making inferences from the observed trends.

Medhansh Singh: Worked on the NGramPreprocessor class. Generated the Ngrams that were required. Made the helper functions which helped in creating the frequency dictionary and split the preprocessed comments for each n-gram. Helped in making preprocessing decisions for the creation of ngrams. Helped in drawing inferences from the observations.

Appendix

File Structure

The repository contains the following folders. Their descriptions are as follows:

- average_preplexity: This folder contains csv files that contains all the average perplexities over all the models for both train and test dataset
- dataset: This folder contains csv files that contain all the data. This includes the raw data, processed data, and train and test data subsets.
- perplexities: This folder contains subfolders for all the different smoothing techniques. Each subfolder (i.e., for each smoothing technique), contains csv files that contain the perplexities for each model after smoothing.
- Plots: This folder contains image files of plots for trends of different smoothing techniques over the different models.

The repository contains the following python files. Their descriptions are as follows:

- NGramProcessor.py: This file contains the class for the n-gram model. This includes all the methods to calculate the perplexities for the model as well.
- ngram_train.py: This file contains code that creates, trains, and gets the perplexities of the n-gram model.
- plot_saver.py: This file contains functions to plot and save the average perplexities.
- preprocessing.py: This file contains a function to preprocess the data and save it.