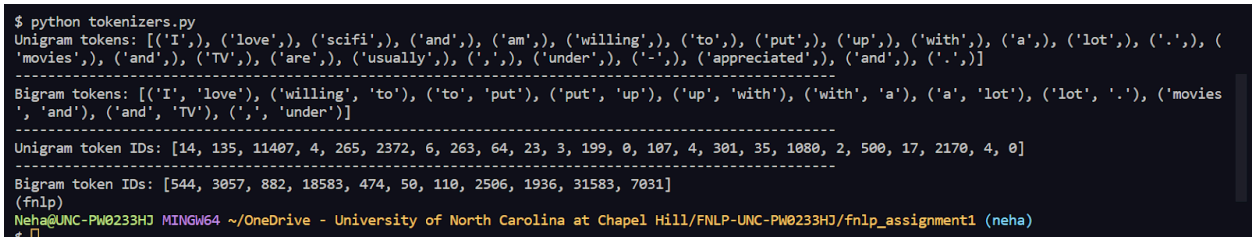
Tokenizer q1,q2
Logistic Regression q1,q2,q3
Embeddings q1,q2,q3,q4
Looking forward q1,q2

s2829893 contributed to Tokenizer q1,q2 Logistic Regression q1,q2,q3, Embeddings q1,q2,q3,q4, and Looking forward q1,q2. s2800414 contributed to Tokenizer q2, Embeddings q2, and Looking forward q1. s2829893 and s2800414 both completed implementations of logistic regression and embeddings code. s2829893 implementation was submitted due to speed and accuracy. s2800414 completed tokenization (tasks 1-2) code.

1. Tokenizer

1. **Report how this text is split into tokens under each method**



2. **What are the advantages and disadvantages of n-gram (n > 1) vs. unigram tokenization? List at least one advantage and disadvantage.**

The advantage of a N-Gram n>1 tokenization is that N-grams contextualize better than unigrams since they consider multiple words together. A bigram model considers the probability of a word given its immediate previous word while a trigram model considers the previous 2 words. A disadvantage is as n increases the number of unique n-grams in a corpus grows making it hard to have reliable probability to estimate unforeseen n-grams. This is why in class we learned smoothing techniques to address these missing probabilities.

Additionally, experimenting with different n-values showed that tokens may appear infrequently or not appear at all (e.g. n>3) resulting in a sparsity issue in our particular case. To work around this, we may need increasingly larger corpora of data to obtain reliable estimates, which is often computationally or financially demanding.

2. Logistic Regression

1. **Hyper-parameters and commands using a Bigram tokenizer and the CountFeatureExtractor**

python run_classifier.py --model LR --tokenizer NGRAM --ngrams 2 --feats COUNTER --learning_rate 0.1 --batch_size 20 --epochs 10

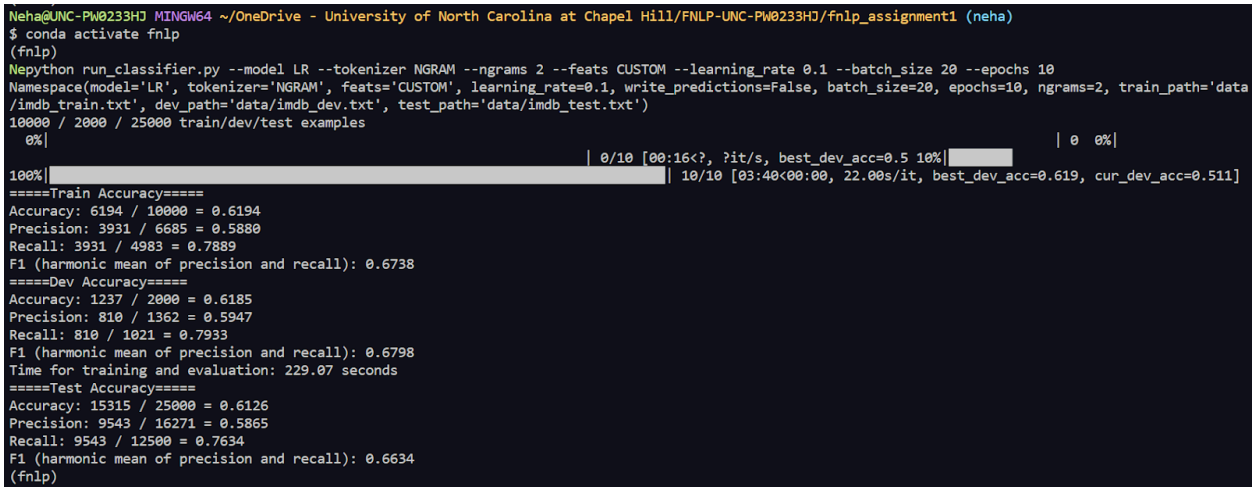
Terminal Output:

| 10/10 [01:09<00:00, 6.99s/it, best_dev_acc=0.848,
=====Train Accuracy=====
Accuracy: 9939 / 10000 = 0.9939
Precision: 4943 / 4964 = 0.9958
Recall: 4943 / 4983 = 0.9920
F1 (harmonic mean of precision and recall): 0.9939
=====Dev Accuracy=====
Accuracy: 1696 / 2000 = 0.8480
Precision: 862 / 1007 = 0.8560
Recall: 862 / 1021 = 0.8443
F1 (harmonic mean of precision and recall): 0.8501
Time for training and evaluation: 73.82 seconds
=====Test Accuracy=====
Accuracy: 21151 / 25000 = 0.8460
Precision: 10516 / 12381 = 0.8494
Recall: 10516 / 12500 = 0.8413
F1 (harmonic mean of precision and recall): 0.8453
(fnlp)

2. **Hyper-parameters and commands using CustomFeatureExtractor**

python run_classifier.py --model LR --tokenizer NGRAM --ngrams 2 --feats CUSTOM --learning_rate 0.1 --batch_size 20 --epochs 10

Terminal Output:



3. **Discuss what your feature extractor does, and why you believed the features would be informative.**

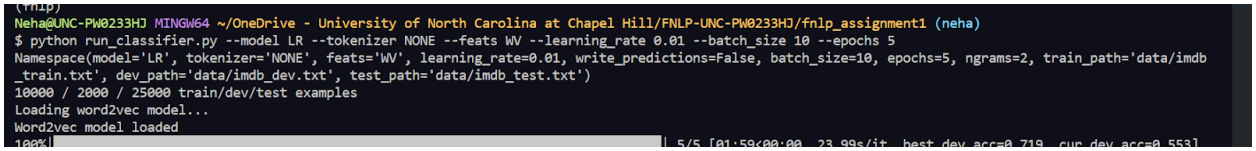
Our CustomFeatureExtractor counts not only individual words and bigrams, but also includes 2 extra features. These features are the total number of words in the review and the average length of those words. In the extract_features method, we first use the tokenizer to break the text into tokens and count how many times each token appears like the CountFeatureExtractor. After we calculate total_tokens and avg_word_length because we believe longer reviews may have more detailed opinions which could help with determining sentiment and shorter words might indicate casual or emotional writing while longer could suggest a more formal tone. We believe that these features would be useful in assessing whether a review was positive or negative because sentiment isn't only based on the words they use. It's also based on the way people write. However, our model's accuracy did not significantly improve with these features. Chapter 5 on Logistic Regression emphasizes that adding irrelevant or weakly correlated features can introduce noise which leads to worse generalization. One possible reason is the extra features may not have added useful information or may have made learning harder by adding noise.

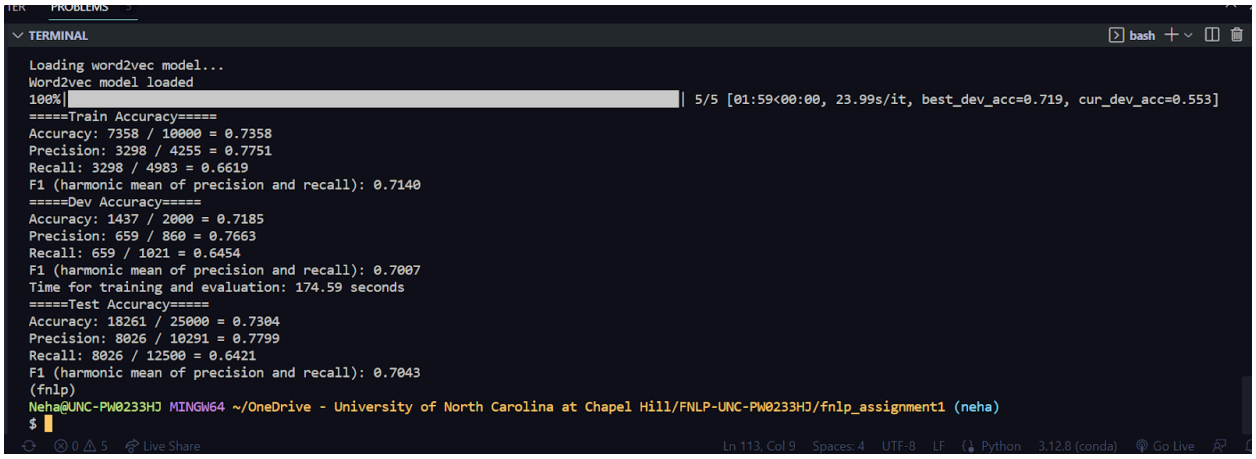
3. Embedding

1. **Hyper-parameters and commands using MeanPoolingWordVectorFeatureExtractor**

python run_classifier.py --model LR --tokenizer NONE --feats WV --learning_rate 0.01 --batch_size 10 --epochs 5

Terminal Output:





2. **Compare the general performance of embedding-based features to count-based features. Hypothesize as to why one performed better than the other, and what the advantages + disadvantages there are with that approach.**

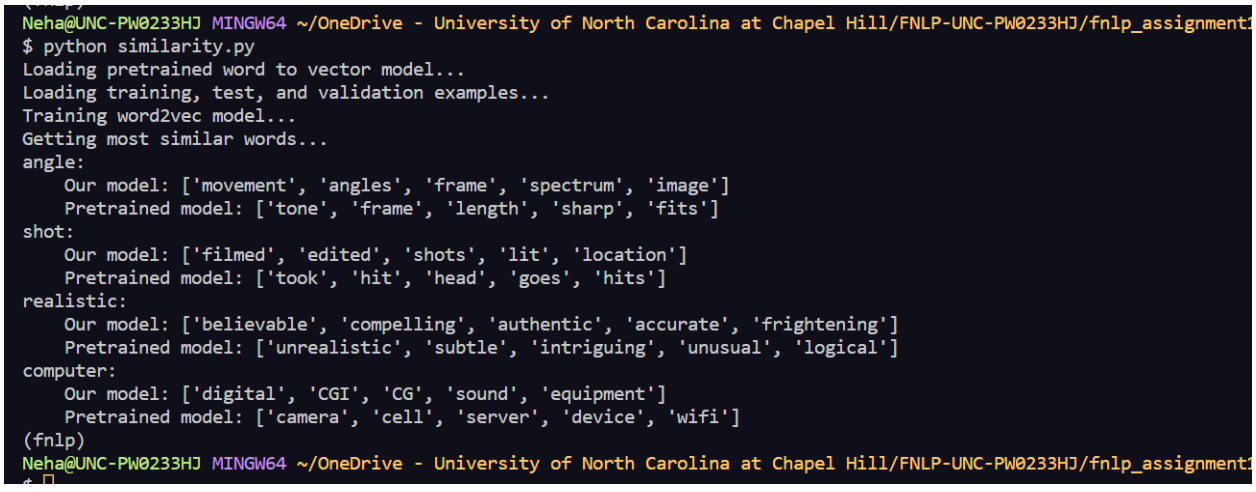
In our case, count-based features outperformed word embedding based features achieving higher accuracy, precision, recall, and F1-score across all datasets. The count-based model reached 84.6% accuracy on the test set compared to 73.04% accuracy for the Word2Vec model. We think one reason for the better performance of count-based features is that n-grams capture word co-occurrence patterns which is good for sentiment classification tasks where certain phrase structures are important like not good or very bad. Word embeddings capture the semantic relationship between words, but they may not be able to differentiate between positive and negative sentiment phrases effectively unless the training data is big enough. Outside of our project, word embeddings are generally more effective and more accurate because they are able to grasp the semantic and contextual relationship while count based purely rely on frequency (Jurafsky and Martin 2024). Word2Vec allows the model to understand word similarity and relationships even if certain words don't show up in the data, so if we had a bigger data set I believe we would have seen the word embeddings be more effective. We think in this case as we were just looking at movie reviews, count-based worked better because the contextual meaning was less critical, so it was quick and accurate as word embeddings require pre-trained models and more computational resources.

We also think that the source of the training data might be one of the factors in this case. The IMDB reviews, by the nature of them being public and "crowdsourced," usually employ a straightforward and linear writing style with frequent use of polarizing words, like "like, love, hate, great, awful, etc." This makes it easy for the n-gram models to just count frequently positive or frequently negative words and get a good estimate. The difficult examples for it would likely include some juxtaposition, e.g. I thought I'd ***hate*** this movie, but I actually ***liked*** it or This movie is ***so bad*** that is ***good***.

As a contrary example, if the movie reviews were sourced from a group of expert movie critics, our experiments might have had different results. Professional critics are quite notorious for using complex structures riddled with metaphors and analogies, where the context plays a key role in identifying a sentiment. They also might score the films differently across multiple aspects. For example: "While the camera work is absolutely stunning, the sound design and actors' performances leave something to be desired, which is ultimately what is important" A count-based model might pick up on "stunning" and classify the sentence as positive, missing the overall mixed sentiment. In contrast, a word embedding model could recognize the subtle contrast and better capture the intended meaning.

Therefore, we think the choice of the model architecture (e.g. n-gram vs word embeddings) should made depending on the particular task and the nature of the dataset.

3. Implement train word2vec model()- given a list of sentences, train a basic Word2Vec model and return it



4. **What trends do you see in the differences between your model and the pre-trained model? What might cause these differences?**

The Word2Vec model and the pretrained GloVe-Twitter-25 model have differences in their word associations because of the variations in the training data, vector size, and model parameters. Our model was trained specifically on IMDB data, so it led to associations like “shots” and “filmed” and “edited”, so all movies related. The pretrained model which was trained on the twitter data set which was much larger linked words like “shot”, “took”, and “hit” which had more of a literal and general meaning. The vector size of our model uses 300-dimensional embeddings while the GloVe model was limited to 25 dimensions. The higher the dimensionality allows our model to create better relationships with words. Also our model included all the words min_count = 1 which could have contributed to overfitting on the sentiment data. The pretrained model definitely had better filtering techniques for more general associations like “camera” and “device”. Our model captured stronger contextual relationships specific to movie-related while pre trained embeddings are more generalizable. Custom embeddings seem to be better suited for specific tasks.

4. Looking Forward

1. **The IMDB data is entirely in English. Imagine we wanted to predict the sentiment of Japanese reviews using your trained pipeline. Remember your word-splitter, tokenizer, count-based feature extractor, and model are all ‘trained’ or initialized based on your training data. For each of (word-splitter, tokenizer, count-feature extractor, learned weights) describe what your model’s default behaviour will be and the effect on performance.**

If we used our trained pipeline for sentiment analysis on Japanese text, there would be several issues. The word splitter would not work with Japanese text because of differences in writing systems. Japanese doesn’t have a whitespace in between words so it relies on the mix of three systems to differentiate the words. The most common words are usually written with Kanji (Chinese-origin characters), some of the Japanese names and particles are written in Hiragana, while the loan words and foreign names are written in the second alphabet called Katakana. Our word splitter would treat entire sentences as single tokens or break characters incorrectly which take away from the meaning of a sentence or word. The tokenizer also wouldn’t be able to run properly due to the ambiguity of most of the Kanji characters. For example, a character for a book 本 (“hon”) can also mean “origin” or “base” but when it comes after the character 日(“ni”) which means the Sun, it forms the word “nihon” which means Japan.

Most tokens would be mapped to an unknown token losing grammatical indicators that are important for meaning in Japanese. The tokenizer would fail to capture subwords and n-gram features and only recognize things like numbers as recognizable tokens. Because of this issue the count feature would not work because the feature space would consist of entirely 0s due to vocab mismatch. Unlike English, Japanese has important sentiment markers which include honorifics which indicate respect and sentence final markers. The models learned weights would also be ineffective because they are tied to English token Ids. Japanese has a fundamentally different word order. For example, the closest example of an English *to-be* verb (as in, “this ***is*** my friend”) in Japanese is ***desu*** which almost always appears at the end of the sentences.

2. **What would you suggest to improve your model’s performance in Japanese? Describe why you think these changes would improve performance.**

To enhance the model's performance on Japanese text, we would implement a morphological analyzer because Japanese does not have explicit word boundaries and has complex compound structures. This would segment words correctly and identify parts of speech since Japanese depends on these grammatical relationships. We would also want to train the data on Japanese data which would help the model learn language specific patterns and handle the several writing systems Japanese has. This is important because Japanese expresses sentiment differently from English. As stated before through grammatical markers and honorific forms rather than explicit sentiment words like good and bad. We would also implement a multilingual model like BERT which would provide pre-trained knowledge of both languages' semantic structures.