CS506-midterm report

In this project, I set out to predict Amazon product review scores by leveraging both textual data (from the 'Summary' and 'Text' fields) and numerical features (Helpfulness). The main challenge was to effectively preprocess and combine these features to build an accurate predictive model. Throughout the process, I noticed certain patterns and implemented specific techniques to enhance the model's performance.

First, at the data processing stage, I thought that punctuation, numbers, and stopwords were unnecessary and could introduce noise when training the model. So, I wrote a function that removes punctuation, numbers, and stopwords while converting all the characters into lowercase. While removing numbers and stopwords might affect some accuracy, after I tested the model with and without the stopwords and numbers, I found that removing them was better. Also, I noticed that different word forms for the same word might affect the model, so I wanted the words to focus on their root forms. Therefore, I used a stemming algorithm to remove suffixes like “-s”, “-es”, and “-ing”. For this step, I used the `SnowballStemmer` method from the NLTK library. Additionally, I filled NaN values in the data with 0 or an empty string (`''`).

For choosing and transforming the features, I started with the text data ('Summary' and 'Text'). I used TF-IDF vectorization to vectorize my text data. Here, I performed a lot of tests on the parameters and how to implement this. First, I tried to combine the 'Summary' and 'Text', but later I found that they might have different information and should be considered separately. As for the tricks I used here, I changed the `ngram_range` to `(1, 2)`, which makes it consider both unigrams and bigrams. I think that is one of the keys that helped my features perform well because if we only consider unigrams by default, meaningful phrases like “not good” will be missed. Also, I found that when I do not limit the `max_features`, the model seems to overfit, and if that number is too low, the model's accuracy is low. Based on this pattern, I tried different `max_features` values for 'Summary' and 'Text' separately and used the values that worked the best. However, I still think there is room to improve on the `max_features` of my model, but due to time limitations, I did not continue to dive into that. Here, I made an assumption that no three-word phrases carry meaningful information, but making the vectorizer consider three-word phrases did make my model worse because it may consider some unimportant things.

For the numerical data, I only focused on the helpfulness data, which are `HelpfulnessNumerator` and `HelpfulnessDenominator`. I noticed that although `HelpfulnessNumerator` and `HelpfulnessDenominator` have a strong positive correlation, which makes using the ratio a good idea, after I graphed `HelpfulnessNumerator` vs. score, I found that higher helpfulness makes it more likely to be extreme (1 or 5 stars). So, I also included `HelpfulnessNumerator` and `HelpfulnessDenominator` in the features.

For the model training, I started by selecting only 25,000 data points for training first, and after finding a couple of models that worked well, I trained them with the whole training dataset. That saved me a lot of time, but I had to make the assumption that the random data I selected could generalize, or I might miss some good models that might work better with a larger training dataset. For the model selection, I tried many different models first, including `SGDClassifier`, `RandomForestClassifier`, `LinearSVC`, `GradientBoostingClassifier`, `LogisticRegression`, and `XGBClassifier`. For all of them, I set the random seed to 42 for reproduction, and for each of them, I tried a couple of different parameters. After that, I found that `LogisticRegression`, `XGBClassifier`, `LinearSVC`, and `GradientBoostingClassifier` were performing well. Especially with `LogisticRegression`, I found that increasing the maximum number of iterations improved performance, and for the `XGBClassifier`, I tried different evaluation metrics and chose the one with the highest `accuracy_score`. To evaluate the model, I found that for all models I had, the accuracy for 5 and 1 ratings was high, but for 2, 3, and 4, it was low. So, I considered adding class weights proportional to counts when training the model; however, the overall accuracy decreased after doing that, so I did not apply this in my final model.

After finding three models that worked overall well, I tried to train those models only on 'Summary' plus 'Text' and the three 'Helpfulness' data. Partly because I wanted to find out which features were most important, and partly because I wanted to see if models worked differently under different selected features. Here, I found that `LogisticRegression` works best when training on 'Summary' plus 'Text', and `LinearSVC` works best when training on the three 'Helpfulness' data. These patterns help thinking the stacking model. For the base case, I chose `LogisticRegression` and `LinearSVC` as the base estimators since they work well on different features, and I used `GradientBoostingClassifier` as the final estimator. I did not use the `XGBClassifier` because it takes too long to train the model. This is how I got my final model; the model evaluation is attached in this report.

Citations for all the methods used in the final model (excluding methods not used in the final model):

- **NLTK Library**: I used the methods `nltk.corpus.stopwords`, `nltk.stem.SnowballStemmer`, and `nltk.download('stopwords')` to remove the stopwords from the data and to convert all the words to their root forms.
- **Scikit-learn Library**: I used `sklearn.feature_extraction.text.TfidfVectorizer` to convert the text data to TF-IDF feature vectors, which allows me to perform regression and easily include it in the model. Also, from scikit-learn, I used `sklearn.svm.LinearSVC`, `sklearn.linear_model.LogisticRegression`,

`sklearn.ensemble.StackingClassifier`, and `sklearn.ensemble.GradientBoostingClassifier`. These four model methods were used to create my final model.

- **SciPy Library**: I used `scipy.sparse.hstack` and `scipy.sparse.csr_matrix` to horizontally stack sparse matrices and create compressed sparse row matrices.
- **Joblib Library**: I used `joblib.dump()` and `joblib.load()` to save and load the model for convenience.
- **Other External and Online Resources**: I used the documentation and books of the libraries I cited, and ChatGPT helped me to come up with ideas for possible models and help me with grammar of the report.