Predicting Review Scores

In this paper, I will document my process for developing a classification model for Amazon reviews. This documentation will be live: I will be updating this simultaneously with trying new algorithms and updating my methods. It will include a description of what measures I am taking, insight into why I am taking such measures, as well as an analysis of the results of a given method.

The starter code Jupyter Notebook is a good place to start to get a sense of what the data format is and what a baseline K-nearest neighbors (KNN) accuracy is. Without making any changes to the code, KNN achieved an accuracy of roughly 0.4. This can certainly be improved upon, but we can examine the confusion matrix for further analysis of the model. The confusion matrix gives us the insight that the model performs fairly well on predicting five and one star reviews, but struggles to predict ratings in between. Upon inspecting the code, my first change is to remove the Time feature as the date of the review likely has little bearing on the score; our features are now just HelpfulnessNumerator, HelpfulnessDenominator, and Helpfulness, not even using Text to determine the customer's sentiment yet. My next step is to increase the value of K so the model compares data points to more than just the 3 nearest neighbors. These changes resulted in a significantly improved accuracy of roughly 0.55; however, analyzing the confusion matrix reveals that the model still struggles with ratings 2 – 4. Using only the Helpfulness features, the model is able to distinguish between very poor and very good ratings, but it has difficulty differentiating between less polar ratings.

This process was never intended to achieve a satisfactory end result; rather, it was meant to just play around with the current code to better understand what is going on. To begin with, while I already removed the irrelevant Time feature, it is fairly intuitive that Helpfulness alone cannot predict a customer's review score. Thus, it is necessary to explore the Summary and Text features. I hope to have the model associate specific words with higher vs lower scores. To do this, I must convert the text to a numerical form that the model can interpret. I can do this using *scikit-learn*'s TfidfVectorizer which allows me to convert the English words to vectors and weigh the words according to their frequency across all documents [1]. Further, I decided to stray away from KNN due to its limitations when it comes to large and dimensionally complex data. I chose to combine this vectorization approach with a decision tree model. Similar to KNN, a simple decision tree will also have trouble with high-volume data and high dimensionality, so instead I used an ensemble version of decision trees called Random Forest [2]. I will try out *scikit-learn*'s implementation of such a model called RandomForestClassifier [3]. This was in fact able to improve the accuracy, achieving a new score of 0.58; however, this achievement comes with a caveat: time. I left this model to run overnight and it ended up taking 762 minutes to complete its execution. Clearly, either vectorization, the Random Forest technique, or both of them are extremely computationally expensive (I am regretting not adding print statements that indicate the time taken for each step). In the IBM article [2], the key challenges section explains that while Random Forests are good at handling large datasets, they can still be slow at doing so. However, I believe that the approach I chose for text analysis may have been the bottleneck. Since it weighs words according to their frequency across all documents, it likely took an excessive amount of time calculating these weights due to there being millions of training examples.

It might make more sense to analyze each customer's response individually, avoiding the computational time needed for group analysis. Thus, my next method to test out will be a sentiment analysis of a customer's review to gain insight into their score through the connotation of their word choice. A commonly used library for natural language processing is *nltk*, containing a sentiment analysis tool called `SentimentIntensityAnalyzer` [4]. This takes a string of text as input and gives four scores ranging from -1 to 1 (I used the compound score). Using this tool, I am able to convert the review Summary and review Text to numerical values that can be meaningfully interpreted by the model. This is a similar strategy to the vectorization method used earlier, but it operates significantly faster. There are a couple minor additions that are worth noting: I am able to use more of *nltk*'s features to remove meaningless words using `nltk.corpus.stopwords` [5] and I am also able to reduce words to their roots using `nltk.stem.WordNetLemmatizer` [6]. These allow the sentiment analysis to avoid picking up too much distracting noise.

Another change that I am making to this iteration of the model is the classification model altogether. The initial KNN model performed well on 5 star scores, subpar on 1 star scores, and terrible on 2–4 star scores. The Random Forest model was less successful with the polar 5 and 1 star scores, but better with the middling 2–4 star scores. I am hoping that a new model, XGBoost, will be able to find a middle ground [7]. The main reason I am hopeful for such is because XGBoost is better than models like Random Forest when dealing with data that contains class imbalance [8]. This is made abundantly clear for our dataset through the bar plot provided in the starter Jupyter Notebook: there are over double the 5 star reviews compared to the next most frequent class. Using sentiment analysis and XGBoost, my score fell to about 0.56, indicating that the term frequency vectorization method used earlier may have been more powerful, with a significant sacrifice of computational time.

Unwilling to entertain a model that takes over 12 hours to train, I will stick with sentiment analysis as opposed to the term vectorization. However, this leaves me quite limited as my only features are Helpfulness, Summary Sentiment, and Text Sentiment, none of which are exceedingly valuable to my results thus far. Although, upon playing around with uniqueness in the data, I noticed that there are many repeat products and users. It seems reasonable to use these entities' historical data to strengthen unknown predictions. I can gather simple statistics (mean and variance) regarding repeat products and users and I added those as features, with one-time products and users receiving the overall mean and variance as placeholders. This insight was hugely beneficial, skyrocketing my local accuracy to roughly 0.677 (unfortunately, the Kaggle public score was only ~ 0.616; theoretically, though, the private score should be higher than my local score to balance it out).

Now that I have a model that I am content with, I can tune its hyperparameters. One option for this is manually trying slightly different models, but that is time consuming and requires intermittent monitoring. On the other hand, I can employ automated hyperparameter tuning, such as through grid search, offered by *scikit-learn* as `GridSearchCV` [9]. This brought me up to a high score of ~ 0.683 when tested locally, proving to have, at least somewhat, optimized my XGBoost model.

Citations

[1] https://scikit-learn.org/1.5/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

[2] https://www.ibm.com/topics/random-forest#:~:text=Random%20forest%20is%20a%20commonly,Decision%20trees

[3] https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[4] https://www.nltk.org/api/nltk.sentiment.vader.html

[5] https://pythonspot.com/nltk-stop-words/

[6] https://www.nltk.org/api/nltk.stem.WordNetLemmatizer.html?highlight=wordnet

[7] https://www.nvidia.com/en-us/glossary/xgboost/

[8] https://www.geeksforgeeks.org/difference-between-random-forest-vs-xgboost/

[9] https://scikit-learn.org/dev/modules/generated/sklearn.model_selection.GridSearchCV.html