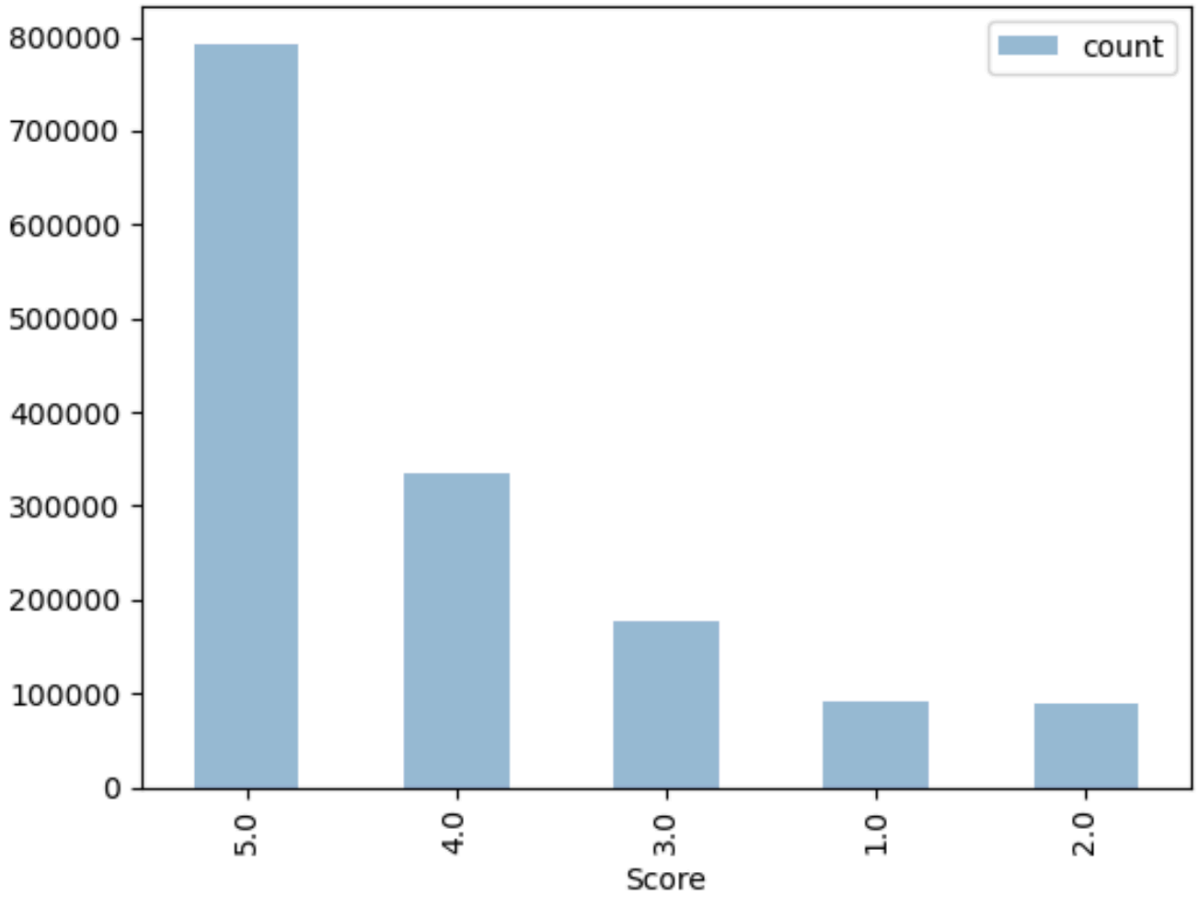
CS506 Midterm Report

Daniel Matuzka

The model I started with in the beginning was **KNNClassifier.** I tuned the *neighbors* parameter to 50 (golden middle between accuracy and time-efficiency for quick testing at the start) and scaled all of the features I added for better performance. I decided to focus on the features first and work on selecting the model later. The first feature I introduced to my model was 'Helpfulness', which was calculated as a ratio between 'HelpfulnessNumerator' and 'HelpfulnessDenominator' (if it appeared to be infinity because of 0 user interaction, I put it back to 0 for better reflection) . Higher values in these columns, especially in the 'Helpfulness' category, appeared to correlate with higher-rating reviews. Just these features achieved a decent accuracy of predicting 5.0 'Score' on a review, which appeared to be the most common out of all (see the distribution below). I also experimented with average helpfulness and later on added features based on deviation from average helpfulness for the product and for the user.



The next step was adding popularity features, such as 'ProductReviewCount' and 'ReviewCountByUser'. This further slightly increased accuracy as it appeared that products with similar popularity often tend to have similar ratings and more active users have similar tastes to one another. The included column 'Time' also showed positive correlation with the score.

Another basic feature I implemented was ‘ReviewLength’ and it also appeared to positively affect my model.

Starting from now on, I began looking into how I could extract more useful information from ‘Summary’ and ‘Text’ columns as they were the main determinants of how the reviewer felt and, therefore, the score they gave. For the future text-based features I combined the two columns into one in order to process them both together at the same time. One of the more simple features included ‘ExclamationCount’ and ‘QuestionCount’ where I counted the number of exclamation and question marks respectively. Use of punctuation signs like these two correlates with the emotions the reviewer felt after watching the movie, such as excitement or unsureness. This turned out to be quite helpful and increased the overall accuracy by around 1.5%. Looking into ways to analyze the sentiment of the reviews, I found **TextBlob** - NLP tool, one of the functionalities of which is assessing sentiment of the provided text. It would return a sentiment polarity score from -1 (negative) to 1 (positive) based on reading the ‘CombinedTextSummary’ field - ‘Sentiment’ column in the dataframe. Adding this feature made the most difference during the midterm as it made the accuracy jump up by around 13%. The model now was predicting 5.0 and 1.0 really well, accuracy of guessing the 5.0 even went slightly over 90%. However, the mid-range scores were still quite mixed up as the differences in sentiment polarity were not that high in-between them for the model to consistently make the right predictions. In addition, the model was false-predicting 5.0 score a lot. I tried using some resampling methods to balance out the training set, but it only led to the decrease of the score. The main reason for this was that 5.0 was still the majority score to be predicted in the testing set and resampling decreased the accuracy of predicting 5’s. My features could not yet distinguish between 2.0, 3.0 and 4.0 well enough to compensate for that.

I was experimenting with a lot of models at this time, such as **DecisionTreeClassifier, BalancedRandomForestClassifier,** and looked into creating my own hybrid ensembles from two models (one balanced and one unbalanced). Ultimately, the one that provided the most accuracy was **HistGradientBoostingClassifier**. In addition, it also ran quite fast, so I ended up sticking with it for the rest of the midterm, just hypertuned the parameters later on. The model builds an ensemble of decision trees which also learn from the mistakes of one another. As I discovered, it has its own unique way of grouping features into ‘histograms’ which frees up memory and, therefore, increases speed and performance.

Some new ideas I got for text features included ‘NegationCount’ and ‘PositiveWordCount’. I wanted to add more features reflecting the sentiment so that they could, along with the ‘Sentiment’, help the model better distinguish between mid-range scores. My thought process was that scores like 3.0 would include a lot of contradictions, comparing good about the movie against the bad. I created a small list of negation words and added a feature that would reflect how often they are used within a text. Turned out this was the case and it was a helpful feature for the model. On a similar note, that is how I thought about introducing the opposite feature and came up with ‘PositiveWordCount’. **NLTK** has an *opinion_lexicon* which

helped me count the number of positive words used in reviews. This also slightly helped to distinguish 2's from 3's and 3's from 4's, for example.

I then tried working with another sentiment analysis library called **VADER** and adding features for 'Positivity', 'Neutrality', 'Compound' and 'Neutrality'. In the end, it appeared that all of them were redundant and led to worse performance so I ended up scraping all of them. I did not use anything from the package for my final model.

Finally, I decided to introduce TF-IDF features into my model. They gave me the final big jump from around 60% accuracy all the way to around 66%. I operated on the assumption that reviews with similar ratings would often tend to have similar phrases used, and the term-document matrix helped me put it into actual 'features' that my model could use for learning. It paid off and especially helped with accuracy for scores of 2, 3 and 4.

After this I was done with feature adding and put some time into hypertuning the parameters for my model. As I also was slightly limited by time, I chose to select parameters that would make the process of fitting the model not that long. Having additional 1000 features introduced from TF-IDF, even the usually fast-running **HistGradientBoostingClassifier** had to take its time. Ultimately, I ended up getting these parameters: *max_iter=1500, learning_rate=0.05, max_leaf_nodes=100* and *min_samples_leaf=60*. This, in the end, gave me the final model I used to make my best prediction for the competition.

Used Resources

TextBlob - https://textblob.readthedocs.io/en/dev/quickstart.html#sentiment-analysis

HistGradientBoostingClassifier - https://scikit-learn.org/dev/modules/generated/sklearn.ensemble.HistGradientBoostingClassifier.html