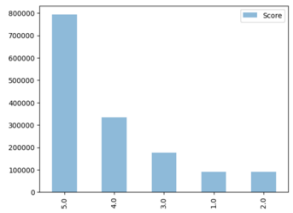
Predicting the star rating associated with user reviews from Amazon Movie Reviews

Examining Data and Feature Selection / Engineering:
train.csv shape is (1697533, 9) / test.csv shape is (212192, 2)



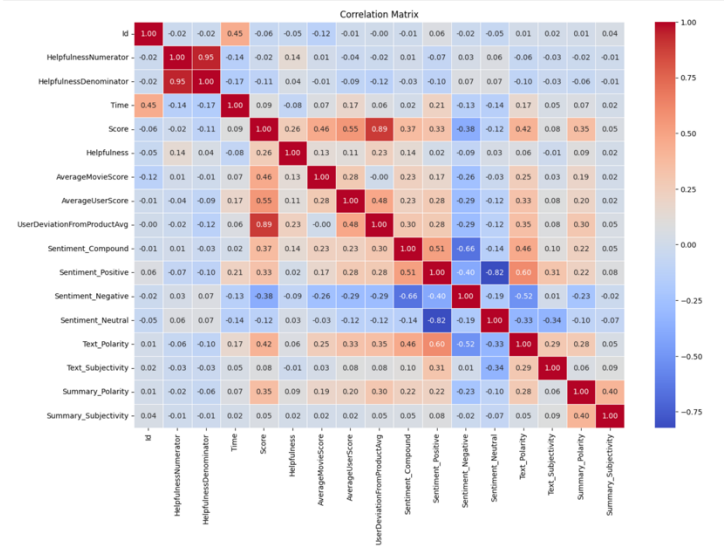
The dataset is huge (1697533 rows), so computing for feature selection and fitting model will naturally take a long time. I encountered an issue using Kaggle notebook that the runtime exceeded 12 hours and server shut down automatically. The way I solved this issue is introducing parallel computing using joblib to utilize all the cores on my laptop instead of 1, and reduced the training time significantly (around 20-30 times faster). Setting n_jobs = -1 utilized all the cores.

Feature Selection / Engineering:
'Helpfulness': The helpfulness features has a decent correlation with score (0.26), I printed some low helpfulness text and finded out these text tend to have less correlation with the score, that their response doesn't reflect if the movie is good or not.
'AverageMovieScore': I used ProductId and correspond it with the average score of the product (movie). If a movie is good, then it will natually has a higher score, and vice versa.
'AverageUserScore': I used UserId and correspond it with the average score of the each user. If a user tends to give higher scores, then the probability of the user to give high score will be high, and vice versa.
'UserDeviationFromProductAvg': This parameter measures the difference between the score user gives and the average score of the movie. This will show how generous or critical a user tends to be relative to other users for the same product.
Sentiment Analysis:
I used two sentiment library to analyze the text and summary: Vader Sentiment and TextBlob. Initially I included Afinn, but I found these two libraries are performing better than Afinn sentiment analysis, so I decided to include only Vader and TextBlob.
VADER Sentiment['Sentiment_Compound', 'Sentiment_Positive', 'Sentiment_Negative', 'Sentiment_Neutral']: Vader sentiment was exceptionally good in terms of analyzing sentiment for short text, and handle informal languages and includes slang. So it is good for social media and movie review, so it fits well for this assignment. Vader sentiment generated four features that will measure the portion of positive, negative and neutral text for each 'Text', and it will normalize between 0 and 1. And compound is the combined sentiment score between -1 and 1. From the correlation matrix, Vader sentiment has the correlation with score of high 0.3s for compound, positive and negative, which is higher than the 0.26 from Afinn.

TextBlob Sentiment['Text_Polarity', 'Text_Subjectivity', 'Summary_Polarity', 'Summary_Subjectivity']: I used textblob because it is a fast computing sentiment score that calculates polarity and subjectivity for the text. So it works well with the summary as they are typically short and the sentiment is quite clear (neutral, positive or negative). The subjectivity is also an interesting feature, I found some people wrote reviews with objective tones, the subjectivity can act as a weight for polarity in the models, while 0.08 correlation is not high, I tested that the addition of subjectivity indeed made my prediction accuracy better.

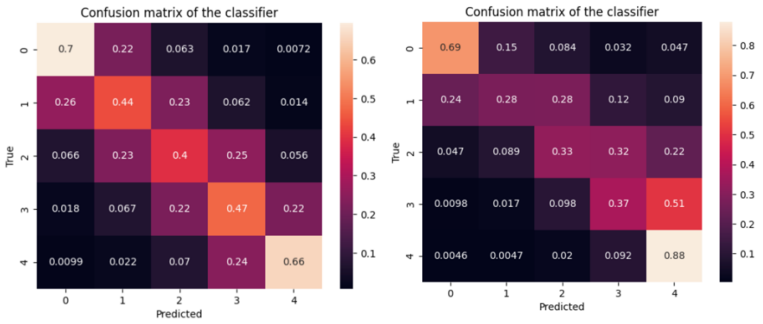
Min-Max scaling: Improves Convergence in Gradient-Based Algorithms, I used XGBoost, which use optimization techniques such as gradient descent to minimize a loss function. If the features have varying scales, the gradients might fluctuate widely, leading to slower convergence or a suboptimal solution.

Correlation Matrix:



Resampling:

Given the data is imbalanced as most of the scores are 5, I tried to apply resampling to deal with the imbalanced data. I first tried resampling by adding weight on XGBoost with the distribution of the data, and it gives a more diagonal like confusion matrix, however the accuracy is lower because it is predicting less score of 5, which is the nature of the data. (left for balanced, right for not balanced)



Resampling by oversampling others:

Oversampling the number of score of 4 to 0.8 of number of score 5 will yield an accuracy of 0.6443, lower than the 0.6499 for not doing resampling. I decided to not do resampling because predicting more 5 does improve the accuracy.

Model Selection:

I tried KNN, Random Forest and XGBoost, as well as stacking KNN with XGBoost to improve prediction to combine the strength of both models. However, I found stacking KNN with XGBoost does not make any difference comparing with just using XGBoost, looks like XGBoost fits better for all the parameters.

Comparing Random Forest with XGBoost, random forest gives an accuracy of 0.618, while XGBoost has an accuracy of 0.6499 on local testing set.

XGBoost is based on gradient boosting where multiple weak learners (typically decision trees) are sequentially trained. Each subsequent tree tries to correct the errors made by the previous trees by focusing more on the data points that were mispredicted. XGBoost is great in terms of preventing overfitting, the KNN model overfits much more than the XGBoost model when submitting to Kaggle public test, resulting in a less discrepancy between local test and kaggle test.

Fine tuning:
I used RandomizedSearchCV to random search the parameters with 5 fold cross validation totaling of 500 fits. The fine-tuned model improved by approximately 0.5-1.0% accuracy. The parameters includes:
colsample_bytree: controls the fraction of features
gamma: a regularization parameter that controls whether a given node will be split based on the reduction in loss after the split
learning_rate: controls how much each tree contributes to the final model. I tried to further reduce the learning_rate to pervent overfitting, in the end 0.0889 works the best.
max_depth: controls the maximum depth of each tree
min_child_weight: specifies the minimum sum of instance weights (hessian) needed in a child node
n_estimators: specifies the number of trees (or boosting rounds) that are built in the model
reg_alpha: the L1 regularization term on the leaf weights of the trees. It adds a penalty for large leaf weights, encouraging sparsity in the model. I used 0.07 to help to control overfitting by reducing the model's complexity.
reg_lambda: the L2 regularization term on the leaf weights, it penalizes the magnitude of the leaf weights.
Subsample: controls the fraction of the training data that is randomly sampled for each tree, I used 0.901 to introduce some randomness to prevent overfitting.

Links for external libraries:
https://xgboost.readthedocs.io/en/stable/
https://github.com/cjhutto/vaderSentiment
https://textblob.readthedocs.io/en/dev/