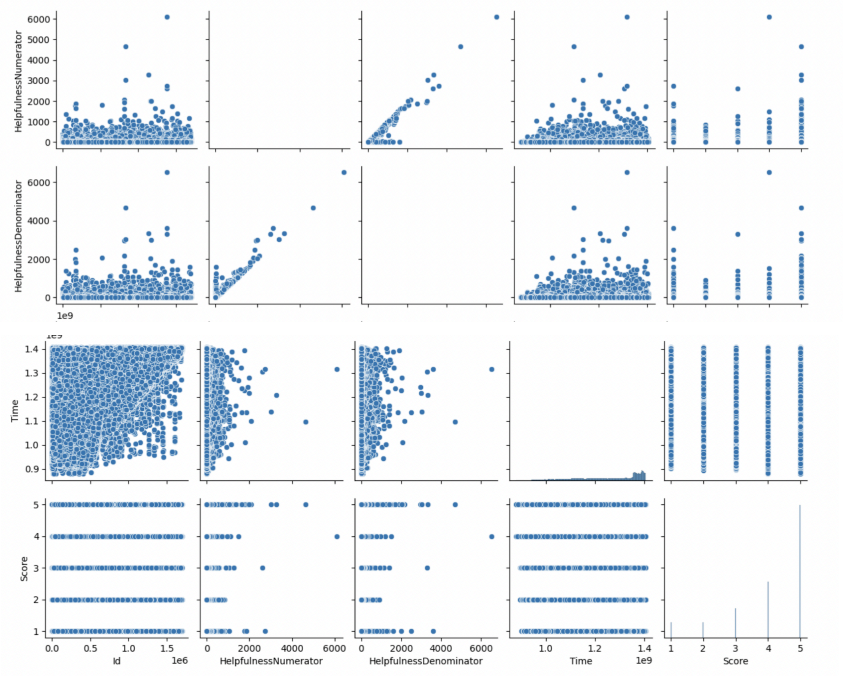
Predicting Amazon Movie Review Ratings: Model Creation Workflow

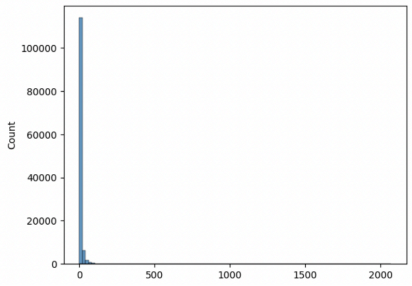
In this midterm Kaggle challenge (username: Kathyzyx), my goal was to predict the star rating (ranging from 1 to 5 stars) associated with user reviews from Amazon Movie Reviews. Throughout the development process, I focused on data exploration, feature engineering, preprocessing, and model selection methods that will enhance predictive performance while optimizing runtime efficiency.

I. Data Exploration and Splitting

To understand the given dataset, I used a series of visualization tools. Firstly, I plotted pairplots, also known as scatterplot matrices, to help reveal the pairwise correlations between the given features: Helpfulness Numerator, Helpfulness Denominator, Time, and Score. Notably, the plots highlighted that while Helpfulness Numerator and Denominator were correlated as shown by its linear trend (as expected), the relationship between Time and Score was not as direct, suggesting a need for additional time-based features or preprocessing techniques to reveal the more fine-grained correlations.



Additionally, a histogram analysis of reviews per user (shown below) revealed that the majority of users contributed only a few reviews, with a small subset of prolific reviewers. This leftwards skew in user review contributions indicates that there may not be enough reviews per user to reveal a significant trend if we focus on UserID as a feature. This also reaffirms our focus on extracting discriminative information from text-based features.



Finally, I compared the train and test data sets to check for any overlaps. This analysis revealed a notable overlap between products in the training and test set for us to leverage. To prevent information leakage between the training and test set, I first separated the two, yielding 1697533 training samples and 212192 testing samples. To further validate models, I created a validation split, containing 20% of the training set, and leave the remaining 80% of the training set for model fitting.

II. Adding Features (Trick #1)

Feature engineering played a crucial role in improving my model's accuracy. I started by extracting hand-crafted features that I thought were intuitively significant to score-determination:

Helpfulness Features: Helpfulness Ratio and Down Votes
Helpfulness Ratio: As shown by the pairplots from above, there is a strong pairwise relationship between Helpfulness Numerator and Denominator so I generated a feature that would utilize the ratio between the two.
Downvotes: We can assume that reviews where the customers did not find the product to be helpful would be associated with lower scores.

Time features: TimeYear, TimeMonth, TimeDays
Here, I assumed that as time passed, there may exist changes in the products (most likely from improvements over time) season-dependent changes in reviews. Here, I divided the timestamp into years, months, and days to enable a closer analysis of how time may impact customer reviews. As shown my the pairplot, there does not seem to be a much of a correlation between the time stamp and the score, which may be because we need to quantify time in more detail.

Statistical Features: ProductID Average Score and Variance
Average Score and Score Variance: Aggregated scores per product helped measure consistency in product quality.
Review Count: The number of reviews per product was used as a proxy for popularity.

Text Sentiment Features:
For text analysis, I chose to utilize stneiment polarity using the **TextBlob** library, where sentiments for both the summary and the full review text were calculated, ranging from -1 (negative) to 1 (positive).
A discrete classification was introduced to simplify sentiment, categorizing reviews as positive, neutral, or negative based on predefined thresholds.

1. Sentiment Score (Polarity): I used the get_sentiment function from TextBlob to calculate the sentiment polarity of the review text and summary because it provides a quantitative measure of the review's emotional tone. Reviews that are more positive are likely associated with higher ratings, while negative sentiments might indicate lower ratings. This feature helps the model capture the underlying mood or opinion of the user towards the product.
2. Discrete Sentiment Score: The get_sentiment_discrete function converts the continuous sentiment polarity score into discrete values: 1 for positive sentiment, -1 for negative, and 0 for neutral. The threshold of 0.05 is used to differentiate neutral sentiment from positive or negative. Discrete sentiment can be useful because it simplifies the interpretation of sentiment as a categorical variable. Discretizing the sentiment helps the model focus on clear sentiment classifications, which may improve classification performance for predicting ratings.
3. Subjectivity: The subjectivity score ranges from 0 (completely objective) to 1 (highly subjective). Subjectivity provides insight into how personal or opinionated the review is. Reviews with higher subjectivity may indicate stronger opinions (whether positive or negative), which could correlate with extreme ratings (1 or 5 stars). On the other hand, more objective reviews might correlate with moderate ratings. Including subjectivity as a feature can help the model differentiate between opinion-heavy reviews and more factual, balanced reviews.

Text Length: The length of the review can provide indirect information about the reviewer’s engagement or strength of opinion. Longer reviews may indicate more effort and possibly stronger sentiments, which might correspond with either very high or very low ratings. This feature adds depth to the model by providing a simple proxy for engagement level.

Punctuation Counts: Exclamations, questions, and capital letters were counted to infer tone intensity. Longer reviews or reviews with more exclamatory or capitalized words are more intense and reflect stronger positive or negative opinions, thus impacting the rating.
1. Exclamation and Question Counts: Punctuation marks, especially exclamations and questions, can reveal the emotional intensity of a review.
2. Capital Letters Count: Users often capitalize words for emphasis, indicating strong emotions. By quantifying the number of capital letters, the model can detect reviews where the reviewer has emphasized certain opinions. This feature contributes additional sentiment information beyond what’s provided by standard text sentiment analysis.

TF-IDF Vectorization: Common unigrams and bigrams were extracted using sklearn's TF-IDF, adding 1,400 features to the dataset, reflecting word importance. The most common unigrams and bigrams extracted through TF-IDF are meaningful indicators of the review’s sentiment and content, thus correlating with the star rating. This assumes that the frequency and context of specific words hold predictive power for ratings.

I generated word clouds for positive (score ≥ 4) and negative reviews (score ≤ 2). These word clouds provided qualitative insights, showing that common terms like "movie," "film," and "watch" appeared frequently in both positive and negative reviews, but their surrounding context helped differentiate sentiment.

III. Preprocessing Techniques (Trick #2)

The 80/20 train-validation split sufficiently represents the data, providing a robust measure of model performance and generalization capability. This assumes that the validation set is representative of unseen data

Normalization Strategies

Given the variety of features, several normalization methods were employed:

1. Z-Normalization: Applied to most features to standardize them to a mean of 0 and a standard deviation of 1.
2. Min-Max Scaling: Used for time-based features to preserve their ordinal nature without distorting intervals (e.g., January to December).
3. Nonlinear Transformation: To handle skewed distributions in engagement metrics (e.g., helpfulness votes), a power transformation was applied to make data more uniform, minimizing outliers' influence.

Handling Missing Values

Missing values, particularly in text-based features, were filled with zeros. Random forests can handle missing values natively, but this approach ensured uniformity across models.

IV. Modeling Approach (Trick #3)

Initially I did not know which type of model would be the best of this dataset, so I attempted to run three different types of models: SVM (one we learned in class), logistic regression, and one ensembling (bagging with random forests). Theoretically, I had already expected ensembles to work better since they combine the predictions from multiple classifiers.

Since Random Forests were not covered in class, I will detail them here. Random Forests are a bagging ensemble which means they consist of multiple weak learners (decision tree classifiers). We obtain an independent prediction from each classifier and the final prediction of the Random Forest is the majority vote. To further diversify the information modeling of each model, we may choose different subsets of the training data to fit each weak learner. For this implementation, I adopted sklearn's RandomForestClassifier class.

Hyperparameters for each model were tuned through grid search:

Random Forest: Tested tree numbers (50, 100, 200, 300) and minimum samples per split to balance performance.
Evaluation Metrics: Metrics included accuracy, F1-score, and ROC-AUC, providing a balanced assessment of model performance.

Parallel Processing Trick #4

Because I observed a considerably long run time, especially with larger parameters and with ensembles, I decided to utilize the pandarallel library. Pandarallel allows me to parallelize my Pandas operations on all available CPU cores, which significantly sped up my data processing tasks. This reduced the processing time from hours to under 10 minutes.

References

1. https://medium.com/@coldstart_coder/understanding-and-implementing-tf-idf-in-python-a325d1301484

2. https://levity.ai/blog/text-classifiers-in-machine-learning-a-practical-guide