Victoria Lin

CS506 Midterm Report

The essential libraries for data manipulation, feature extraction, model training, and evaluation are imported. Then the code loads the *train.csv* and *test.csv* datasets, with *train_data* containing features and labels and *test_ids* providing test set identifiers. After merging *train_data* with *test_ids* on the *Id* column to create *test_data*, rows missing the target variable (*Score*) are removed from *train_data* to ensure label completeness. At this point, we have both datasets, *train_data* and *test_data*. To reduce computational load, a 70% random sample of *train_data* is selected. This initial setup provides a clean, manageable dataset ready for preprocessing, feature engineering, and model training.

The preprocessing is split into data cleaning, then generating several new artificial features based on the information provided.

A preprocessing function, *preprocess_1* standardizes data formats for *train_data* and *test_data*. It also calculates *HelpfulnessRatio* by dividing *HelpfulnessNumerator* by *HelpfulnessDenominator* (or setting it to 0 if the denominator is zero). It also creates logarithmic versions of these values to reduce skew. If a *Time* column exists, it converts it to datetime format and extracts *Year* and *Month*; otherwise, it assigns default values of 0. This preprocessing ensures consistent formats and meaningful features for model input.

Then the text data in *train_data* and *test_data* is tokenized and performs a sentiment analysis based on predefined word lists. Using *pandarallel* for parallel processing, it tokenizes *Text* and *Summary* columns into lowercase words, creating *TextTokens* and *SummaryTokens*. The tokenization is done using NLTK's word_tokenizer, which is a pre-trained tokenization model. The *preprocess_2* function then analyzes the tokens for sentiment by counting occurrences of positive, negative, and neutral words. It returns counts of these words and calculates a *SentimentText* score (positive minus negative words) for the main text and summary, capturing the general sentiment in each review. This was included because I noticed that there are reviews that use words such as “fantastic” or “terrible”, and these types of words with positive and negative connotations give us insight into the review's potential score. This approach enables efficient text processing and sentiment feature extraction.

Additional feature engineering to enhance text-based insights are introduced. The *preprocess_3* function begins by cleaning the *Text* and *Summary* columns, removing non-alphabetic characters while retaining exclamation marks and question marks, then converts all text to lowercase. It creates several new features: *TextLen* (character length of *CleanText*), *AvgWordLen* (average word length in *TextTokens*), *NumExclamations* (count of exclamation marks), and *NumQuestions* (count of question marks). These features capture various textual characteristics such as length, punctuation usage, and average word length that can provide the model with more contextual information about the review's tone and style.

The next step was to create embeddings of the text. Although I considered TF-IDF vectorization, I decided to not use this because it was computationally expensive and took a long time to run, even for a subset of the full data. GloVe word embeddings are embeddings that have already been created by a lexicon and rule-based approach (without neural networks). I downloaded a file that contained these embeddings, then loaded and applied these embeddings to each text data to generate semantic representations of documents. The *load_glove_embeddings* function reads the GloVe file, storing each word and its corresponding embedding vector in a dictionary (*embeddings_index*). The *get_document_embedding* function then calculates an embedding for each document by averaging the vectors of the words present in GloVe, providing a fixed-size semantic representation. Using a 300-dimensional GloVe model, this process generates *text_vectors_train* and *summary_vectors_train* for *train_data*, and similarly, *text_vectors_test* and *summary_vectors_test* for *test_data*. Each document embedding captures the overall meaning of the text, allowing the model to use these semantic features.

ProductId is encoded using HashVectorizer (to save space versus a one-hot encoding approach) and PCA is applied for dimensionality reduction, transforming high-dimensional product identifiers into a 20-component matrix for *train_data* and *test_data*, preserving variance while reducing sparsity. Additionally, the VADER sentiment analyzer is used on *CleanText* to extract sentiment scores (negative, neutral, positive, compound), adding these as features in *train_data*. I added these just to give some more robustness to the model, and to hopefully improve my personal sentiment analysis. Together, these steps enrich the dataset by capturing both product identifiers and nuanced sentiment information, aiding the model's ability to interpret and predict ratings effectively.

Then, we select key numeric features across *train_data* and *test_data* (e.g., *HelpfulnessRatio*, sentiment metrics, text length) and standardize them with *StandardScaler* to ensure uniform scale. The code then combines these scaled numeric features with text and summary embeddings (*text_vectors_train, summary_vectors_train*) and the reduced *ProductId* components (*reduced_ids*), creating the fully processed datasets (*X* for training and *X_test* for testing) ready for model input. The target variable, *Score*, is stored in *y* for training.

The fully preprocessed dataset is split into training and validation sets, with 80% of data in *X_train* and 20% in *X_val*. The target variable *y* is adjusted by shifting each value down by one (to start from zero) for compatibility with certain models like XGBoost. A logistic regression model from scikit-learn (*logreg*) is then trained on *X_train* and *y_train_shifted*, using the *lbfgs* solver and allowing a maximum of 1,000 iterations. Finally, predictions (*y_pred*) are made on *X_val*, and the model's accuracy and detailed classification report are printed to evaluate its performance on the validation set. I used this accuracy on the validation set in order to assess how good my approach is.

Finally, the code generates final predictions for the test set using the trained logistic regression model. Predictions (*y_test_pred*) are made on *X_test*, and the values are shifted up by one to match the original class labels. A submission file is then prepared, containing *Id* from *test_data* and the corresponding predicted *Score*. The results are saved to *submission.csv* for upload.