Overview:

For my highest submission, I got an accuracy score of 0.55274. This is about a 15% improvement from the initial starter code which gave me an accuracy of about 0.40907. This code improved the starter code, but mostly built off of it.

Imports: Aside from the initial imports, I also imported these:

```
from sklearn.model_selection import GridSearchCV, cross_val_score
from sklearn.preprocessing import StandardScaler
```

GridSearchCV was important for model-tuning. Specifically, it would help find the best hyperparameters for a given model. Cross_val_score was imported for cross-validation. This was to provide a simpler way in assessing how well the model generalizes to an independent dataset, reducing the risk of overfitting.

StandardScalar was used to standardize features by removing the mean and scaling to unit variance. Standardization is important for distance-based algorithms like KNN, as it normalizes feature scales to improve model performance. There are a couple reasons why I used this:

- Improves Model Performance: KNN specifically can perform better when the features are on a similar scale.
- Speeds Up Convergence: Gradient-based algorithms like Gradient Descent converge faster when the features are standardized.
- Reduces Bias: It helps in reducing bias towards features with larger scales, ensuring that all features contribute equally to the model.

In my testing, I also tried MinMaxScaler as another scaling method, but that got me a similar score, but did not surpass my highest score. So in the end, I stuck with StandardScalar.

Feature Addition: In terms of adding features and sampling + splitting the data, I did not make any changes from the starter code.

Feature Scaling:

```
# Scaling Features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_select)
X_test_scaled = scaler.transform(X_test_select)
X_submission_scaled = scaler.transform(X_submission_select)
```

This part of the code took the results from the feature selection part of the code and used it for scaling. First, we compute the mean and standard deviation on the training data and apply the

transformation. This is essential for ensuring that the scaling parameters are learned from the training data, which is crucial for maintaining consistency. Then, to make sure that the test and submission data are scaled using the same parameters as the training data, we apply the same transformation to the test and submission data. This aids in maintaining consistency and preventing data leakage.

Hyperparameters: The next issue I tackled was hyperparameters. This was something I learned from assignment 5 and thought it could potentially help me with the midterm as well. The idea was that too few neighbors can lead to a model that's sensitive to noise (overfitting), while too many can smooth out distinctions between classes, making the model underfit. So finding the ideal number of neighbors would boost the accuracy score.

```
# Hyperparameter Tuning using GridSearchCV for KNN
param_grid = {'n_neighbors': range(1,100)}
grid_search = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5)
grid_search.fit(X_train_scaled, Y_train)

print("Best number of neighbors: ",
grid_search.best_params_['n_neighbors'])
```

I specified a range from (1, 100) that GridSearchCV will test. This range was chosen specifically to go through possible values with a high enough range to get a good answer, but with a low enough range that the processing time wouldn't take a significant chunk of time. This range made sure the best number of neighbors was found within a broad interval without testing every possible number. I also had GridSearchCV divide the dataset into 5 subsets, training on 4 and validating on 1, rotating through each subset to ensure thorough testing.

By automatically iterating over each value, GridSearchCV tries each one in combination with cross-validation, giving an averaged score for each parameter. This approach allows us to compare each value of n_neighbors based on accuracy without manually testing each option. The best-performing configuration can then be directly accessed, which allows the immediate selection of the optimal n_neighbors for the final model.

GridSearchCV helped to find the optimal n_nieghbors, and allowed me to maximize the performance of the KNN model.

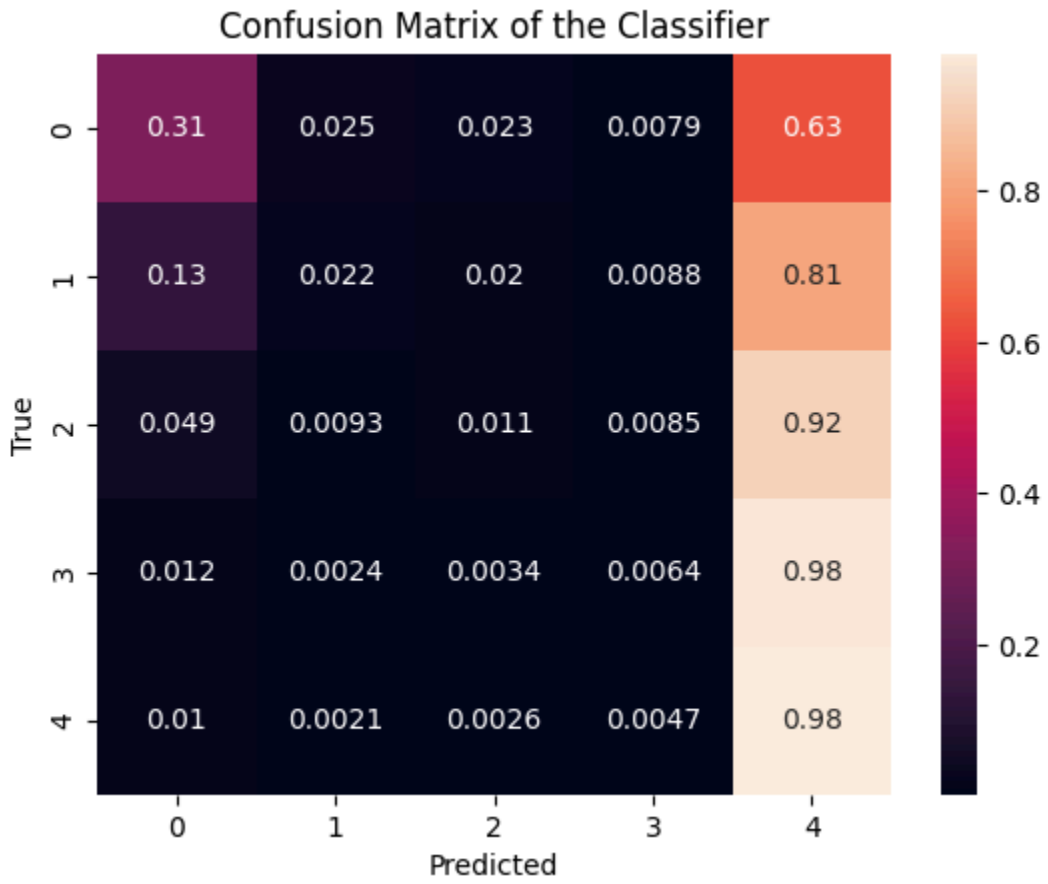
Cross-validation:

```
# Evaluate with cross-validation
cv_scores = cross_val_score(grid_search.best_estimator_,
```

```
X_train_scaled, Y_train, cv=5)
print("Cross-validation accuracy: ", cv_scores.mean())
```

This part of the code is straight forward, it cross-validated the scores we got from the GridSearch. Cross-validation further validates the model's performance by providing an average accuracy score across multiple folds. This helps ensure that the model generalizes well to new data and that its performance is not overestimated.

The rest of the code plotted a confusion matrix as well as the submission file for the code. This was the confusion matrix for my submission file:



Closing Remarks: While I implemented some methods to increase the score from the initial starter code, there were a couple more things I would have tried or done differently. Unfortunately, this code took my laptop *6 hours* to complete, which made me more cautious to add any additional parameters for testing. A couple things that I did not get to test (but was in the middle of) would have been a different classifier model instead of KNN, sentiment analysis, and adding additional features.