CS986 Spotify Classification Problem 2025

Introduction / Abstract

This classification project aimed to predict song genres using machine learning models. The initial process examines the raw dataset, which contains various audio factors and metadata, and proceed with data cleaning — filling missing values, removing outliers, and merging rare classes to avoid stratification issues. Feature engineering plays a vital role in uncovering hidden relationships; for example, we derive "song_age," "energy_loudness," and others captures a track's unique characteristics. We then address class imbalances using SMOTE, ensuring the models does not overlook minority genres.

Moreover, we implement baseline models —such as Logistic Regression, Decision Trees, and k-Nearest Neighbors to establish an initial benchmark. Building on these foundations, we explore advanced algorithms, including Random Forests, Support Vector Machines, and gradient boosting frameworks (XGBoost, CatBoost). Finally, we employ ensemble methods, such as Voting and Stacking, that combine the strengths of multiple models. Each step's performance is assessed with key metrics (accuracy, F1 score), guiding model selection and refinement. Based on the aforementioned metrics, the final models were chosen: k-NN (baseline) & Voting Classification (advanced).

Our score is evaluated in Kaggle Leaderboards:

- k-NN (0.03)
- Voting Classifier (0.44)

Overall, this notebook demonstrates the value of preprocessing, feature engineering, and experimentation with diverse algorithms can significantly improve genre classification.

Library

```
In [ ]: import pandas as pd
import ast
from sklearn.preprocessing import OneHotEncoder
from sklearn.semi_supervised import LabelPropagation
from sklearn.preprocessing import LabelEncoder
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
```

```
In [6]: df_labeled_raw = pd.read_csv('CS98XClassificationTrain.csv')
df_unlabeled_raw = pd.read_csv('CS98XClassificationTest.csv')
```

Two datasets were utilised for this task. The training dataset has 453 rows & 15 columns, including the target variable 'top genre'. The testing has 113 rows & 14 columns, excluding the 'top genre' column.

1.0 Data Preprocessing & Manipulation

1.1 Handling Missing Values

Top Genre: 15 null values was found during data exploration in the training dataset, these values have been manually filled based on research

```
In [11]: # Manually fill missing values based on research:
genre_mapping = {
    'Unchained Melody': 'pop', 'Someone Elses Roses': 'adult standards', 'Drinks On The House':
    'Little Things Means A Lot': 'pop', 'The Lady Is A Tramp': 'jazz', 'If I Give My Heart To Y
    'Stairway Of Love': 'pop', 'You': 'pop', 'No Other Love': 'pop', "I've Waited So Long": 'po
    'Ain\'t That Just the Way': 'pop', 'I Promised Myself': 'pop'}
```

```
# Update the 'top genre' column
df_labeled_raw['top genre'] = df_labeled_raw['top genre'].fillna(df_labeled_raw['title'].map(ge
```

1.2 Handling Duplicates

In [13]:
```
print(df_labeled_raw.duplicated().sum())
print(df_unlabeled_raw.duplicated().sum())
```

```
0
0
```

1.3 Encoding Categorical Features

- Low-cardinality categorical features: Use One-Hot Encoding (OHE).
- High-cardinality categorical features: Use Target Encoding, Frequency Encoding, or Embedding Techniques.

In [16]:
```
# Select categorical columns (assuming 'top genre' and 'artist' are categorical)
categorical_columns = ["top genre", "artist"]

# Count unique values in each categorical feature
for col in categorical_columns:
    unique_values = df_labeled_raw[col].nunique()
    print(f"Feature: {col}, Unique Categories: {unique_values}")
```

```
Feature: top genre, Unique Categories: 87
Feature: artist, Unique Categories: 345
```

- 87 unique genres ("top genre") --> High-cardinality target (multi-class classification).
- 345 unique artists ("artist") --> Moderate-cardinality categorical feature.

With so many unique values, we call them “high cardinality” (for genre) and “moderate cardinality” (for artist). Some machine learning models (Tree-based models: Random Forest, XGBoost) can handle lots of categories better than others.

To make things simpler for the algorithms, we convert these text labels into numbers. For the top genre column, we use Label Encoding, which means each genre label gets a unique integer ID. For the artist column, we use One-Hot Encoding, which creates a set of new columns—one for each possible artist—marked with 0s and 1s. This way, the model can process all our data as numeric features instead of text.

1.4 Feature Engineering

We drop Id and title because they don’t help predict a song’s genre. Id is a unique identifier, and title doesn’t capture meaningful musical information. Retaining them would clutter the model with irrelevant details and risk overfitting to specific song names.

In [20]:
```
# Drop Irrelevant Columns
df_labeled = df_labeled_raw.drop(columns=["Id", "title"])
df_unlabeled = df_unlabeled_raw.drop(columns=["Id", "title"])
```

We created new features from existing song data to help the model understand additional relationships. For instance, converting duration from seconds to minutes, or computing “song_age” from 2025 minus the release year. We also compute loudness or energy ratios that may better capture relevant aspects for modeling. Infinite values (which can result from divisions by zero) are replaced with missing values (NaN). We filled missing numeric values with the median of each column, ensuring we do not lose data rows. This approach helps maintain consistent inputs for downstream modeling while avoiding anomalies or incomplete entries.

In [22]:
```
# Creating New Features
df_labeled['song_age'] = 2025 - df_labeled['year']
df_unlabeled['song_age'] = 2025 - df_unlabeled['year']
df_labeled["dur_minutes"] = df_labeled["dur"] / 60
df_unlabeled["dur_minutes"] = df_unlabeled["dur"] / 60
df_labeled["loudness_per_sec"] = df_labeled["dB"] / df_labeled["dur"]
df_unlabeled["loudness_per_sec"] = df_unlabeled["dB"] / df_unlabeled["dur"]
df_labeled["energy_dance_ratio"] = df_labeled["nrgy"] / (df_labeled["dnce"] + 1)
df_unlabeled["energy_dance_ratio"] = df_unlabeled["nrgy"] / (df_unlabeled["dnce"] + 1)
```

```
df_labeled["acous_per_loud"] = df_labeled["acous"] / (df_labeled["dB"] + 1)
df_unlabeled["acous_per_loud"] = df_unlabeled["acous"] / (df_unlabeled["dB"] + 1)

# Replace infinite values with NaN
df_labeled.replace([np.inf, -np.inf], np.nan, inplace=True)
df_unlabeled.replace([np.inf, -np.inf], np.nan, inplace=True)

# Fill NaN values only for numerical columns
numeric_cols = df_labeled.select_dtypes(include=np.number).columns
df_labeled[numeric_cols] = df_labeled[numeric_cols].fillna(df_labeled[numeric_cols].median())

numeric_cols_2 = df_unlabeled.select_dtypes(include=np.number).columns
df_unlabeled[numeric_cols_2] = df_unlabeled[numeric_cols_2].fillna(df_unlabeled[numeric_cols_2]
```

We apply a PowerTransformer (Yeo-Johnson) to selected numeric features so they better follow a normal (bell-shaped) distribution. This handles positive and negative values, reducing skew and stabilising variance. Fitting on the labeled dataset learns appropriate transformations, and then applying the same transformation to both datasets keeps their distributions consistent. This aims to make features more balanced.

In [24]:
```
from sklearn.preprocessing import PowerTransformer

# Select features to transform
features_to_transform = ['year', 'bpm', 'nrgy', 'dnce', 'dB', 'live', 'val', 'dur', 'acous', 's
                         'loudness_per_sec', 'energy_dance_ratio', 'acous_per_loud']

# Initialize PowerTransformer (using yeo-johnson method, which works for both positive and nega
pt = PowerTransformer(method='yeo-johnson')

# Apply PowerTransformer
df_labeled[features_to_transform] = pt.fit_transform(df_labeled[features_to_transform]) # (labe
df_unlabeled[features_to_transform] = pt.transform(df_unlabeled[features_to_transform]) # (unla
```

OneHotEncoder transforms “artist” into several columns (one per unique artist), using 0s and 1s to indicate each row’s artist. We first “fit” on the labeled set so the encoder learns all possible categories, then apply the same transformation to unlabeled data for consistency.

In [26]:
```
# For the feature columns -> Encoding with One-Hot Encoding
# One-Hot Encoding
col_category = ["artist"] # Select category feature

# Set OneHotEncoder by using the same categories
encoder = OneHotEncoder(handle_unknown="ignore", sparse_output=False)

# Fit & Transform
encoded_labeled = encoder.fit_transform(df_labeled[col_category])
encoded_unlabeled = encoder.transform(df_unlabeled[col_category])
encoded_labeled_df = pd.DataFrame(encoded_labeled, columns=encoder.get_feature_names_out())
encoded_unlabeled_df = pd.DataFrame(encoded_unlabeled, columns=encoder.get_feature_names_out())

# Handling with numeric columns
col_numeric = ["bpm", "nrgy", "dnce", "live", "val", "acous", "spch" , "pop",
               "song_age", "dur_minutes", "loudness_per_sec", "energy_dance_ratio", "acous_per_
df_labeled_numeric = df_labeled[col_numeric]
df_unlabeled_numeric = df_unlabeled[col_numeric]
```

1.5 Standardisation

Numeric columns were standardised. The scaled numeric data is combined with encoded categorical data:

In [29]:
```
# Normalisation (Standardisation)
scaler = StandardScaler()
df_labeled_numeric = pd.DataFrame(scaler.fit_transform(df_labeled_numeric), columns=col_numeric
df_unlabeled_numeric = pd.DataFrame(scaler.transform(df_unlabeled_numeric), columns=col_numeric

# Combine Numeric and Category dataset
merged_labeled_df = pd.concat([df_labeled_numeric, encoded_labeled_df], axis=1)
merged_unlabeled_df = pd.concat([df_unlabeled_numeric, encoded_unlabeled_df], axis=1)
```

1.6 Handling Imbalance

We address imbalanced genres by replacing extremely rare categories with "Unknown," then label-encode the target. Using SMOTE, we oversample minority classes for a more balanced dataset. Finally, we split the data into training and test sets, ensuring fair evaluation and improving the model's ability to handle underrepresented genres.

In [32]:
```
# Seperate Features and Target
y_train_labeled = df_labeled["top genre"]
X_train_labeled = merged_labeled_df
X_train_unlabeled = merged_unlabeled_df.copy()

# Find categories that appear only 1
rare_categories = y_train_labeled.value_counts().loc[lambda x: x == 1].index

# Replace them with 'Other'
y_train_labeled = y_train_labeled.replace(rare_categories, 'Unknown')

# For the target columns -> Encoding with LabelEncoder
le = LabelEncoder()
y_train_labeled_encoded = le.fit_transform(y_train_labeled)

# Use SMOTE for oversampling target to addressing imbalance target
smote = SMOTE(sampling_strategy='auto', k_neighbors=1, random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train_labeled, y_train_labeled_enco

# Split train/test dataset
X_train, X_test, y_train, y_test = train_test_split(X_train_resampled, y_train_resampled, test_
```

2.0 Feature Selection

Filter Methods (e.g., correlation, mutual information, chi-square test) - Perform standardisation after feature selection because these methods work independently of the model and are not affected by scaling.

Wrapper & Embedded Methods (e.g., Recursive Feature Elimination (RFE), Lasso, Decision Trees, Random Forests) - Perform standardisation before feature selection when using models sensitive to feature scales (e.g., logistic regression, SVM, k-NN). If using tree-based models (which are scale-invariant), standardisation is not required.

In [35]:
```
import numpy as np
import pandas as pd
from sklearn.feature_selection import RFE, mutual_info_classif
from sklearn.linear_model import LogisticRegression, LassoCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score

# Dictionary to store selected features for each method
selected_features = {}

# --- Feature Selection Methods ---
# Recursive Feature Elimination (RFE) with Logistic Regression
logreg = LogisticRegression(max_iter=500, random_state=42)
rfe = RFE(estimator=logreg, n_features_to_select=10)  # Adjust number of features
rfe.fit(X_train, y_train)
selected_features["RFE"] = X_train.columns[rfe.ranking_ <= np.mean(rfe.ranking_)].tolist()

# Lasso for Feature Selection
lasso = LassoCV(cv=5, random_state=42, max_iter=5000, tol=1e-4).fit(X_train, y_train)
lasso_coef = np.abs(lasso.coef_)
selected_features["Lasso"] = X_train.columns[lasso_coef > np.mean(lasso_coef)].tolist()

# Tree-Based Feature Selection (Random Forest)
random_forest = RandomForestClassifier(n_estimators=100, random_state=42)
random_forest.fit(X_train, y_train)

# Get feature importances and select features with importance greater than mean
feature_importances = np.array(random_forest.feature_importances_)
selected_features["RandomForest"] = X_train.columns[feature_importances > np.mean(feature_impor

# Mutual Information for Feature Selection
mutual_info = mutual_info_classif(X_train, y_train)
```

```
mutual_info_series = pd.Series(mutual_info, index=X_train.columns)
selected_features["Mutual Information"] = mutual_info_series[mutual_info_series > mutual_info_s
```

Now we have different sets of selected features from multiple methods, the next step is evaluating which selection works best for the classification task. Train & Evaluate Models on each feature Set - compare performance metrics (F1-score for imbalance target) on the test set.

In [37]:
```
# --- Model Evaluation ---
def evaluate_feature_set(feature_name, X_train, X_test, y_train, y_test):
    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    metrics = {"F1 Score": f1_score(y_test, y_pred, average='weighted')}
    return metrics

# Evaluate each feature selection method and store the results
evaluation_results = {}
for method, features in selected_features.items():
    X_train_selected = X_train[features]
    X_test_selected = X_test[features]
    evaluation_results[method] = evaluate_feature_set(method, X_train_selected, X_test_selected

# Convert evaluation results to DataFrame for easy readability
evaluation_df = pd.DataFrame(evaluation_results).T
evaluation_df = evaluation_df.sort_values(by="F1 Score", ascending=False)

print(evaluation_df.T)
```

```
               RFE  RandomForest  Mutual Information   Lasso
F1 Score  0.950053      0.942424             0.93746  0.5548
```

3.0 Train & Test Model

Select RFE-Selected Features for train model

In [40]:
```
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state
```

We use RFE to find the most important features and use these to train:

In [42]:
```
best_features = selected_features['RFE']

# Keep only the selected features
X_train_rfe = X_train[best_features]
X_val_rfe = X_val[best_features]
X_test_rfe = X_test[best_features]
print("Final Feature Count:", len(best_features))
```

```
Final Feature Count: 179
```

In [43]:
```
# Function to compute F1 score
def get_metrics(y_true, y_pred):
    return {"F1 Score": f1_score(y_true, y_pred, average='weighted')}

# Function to evaluate models
def evaluate_model(model, model_name, X_train, X_val, X_test, y_train, y_val, y_test):
    model.fit(X_train, y_train)

    # Predictions
    y_train_pred = model.predict(X_train)
    y_val_pred = model.predict(X_val)
    y_test_pred = model.predict(X_test)

    # Get performance for each dataset
    train_f1 = get_metrics(y_train, y_train_pred)
    val_f1 = get_metrics(y_val, y_val_pred)
    test_f1 = get_metrics(y_test, y_test_pred)

    # Print results
    results_df = pd.DataFrame({"Train": train_f1, "Validation": val_f1, "Test": test_f1})
    print(f"Results for {model_name}:\n{results_df}\n")
    return results_df
```

3.1 Baseline

We define three baseline models—Logistic Regression, Decision Tree, and k-Nearest Neighbors — each capturing different learning patterns: logistic regression for linear decision boundaries, decision trees for non-linear splits, and k-NN for instance-based learning. We found these three reveals which approach suits our data best before deeper refinement.

In [46]:
```python
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Define baseline models
baseline_models = {
    "Logistic Regression": LogisticRegression(C=0.1, max_iter=500, random_state=42),
    "Decision Tree": DecisionTreeClassifier(max_depth=70, random_state=42),
    "K-Nearest Neighbors": Pipeline([("scaler", StandardScaler()),("classifier", KNeighborsClas

# Evaluate Baseline Models
baseline_results = {}
print("\n===== Baseline Models Evaluation =====\n")
for name, model in baseline_models.items():
    baseline_results[name] = evaluate_model(model, name, X_train, X_val, X_test, y_train, y_val
```

```
===== Baseline Models Evaluation =====

Results for Logistic Regression:
             Train  Validation      Test
F1 Score  0.837198    0.811735  0.815444

Results for Decision Tree:
            Train  Validation      Test
F1 Score  0.99639    0.932091  0.910157

Results for K-Nearest Neighbors:
             Train  Validation      Test
F1 Score  0.994129    0.954786  0.944397
```

k-NN performed the best in terms of Train, Validation and Test.

3.2 Advanced

We define advanced models—RandomForest, SVM, XGBoost, CatBoost, and ensembles (Voting, Stacking). Each method captures data patterns differently: tree ensembles learn complex interactions, SVM focuses on robust margins, boosting refines errors at each stage, and ensemble techniques combine strengths.

In [50]:
```python
from sklearn.ensemble import RandomForestClassifier, VotingClassifier, StackingClassifier
from sklearn.svm import SVC
from xgboost import XGBClassifier
from catboost import CatBoostClassifier

# Define advanced models
rf_model = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)
svm_model = SVC(kernel='poly', C=1, probability=True, random_state=42)
xgb_model = XGBClassifier(n_estimators=100, learning_rate=0.1, max_depth=3, subsample=0.8, cols
catboost_model = CatBoostClassifier(iterations=100, learning_rate=0.2, depth=6, verbose=0, rand
stacking_model = StackingClassifier(estimators=[('RandomForest', rf_model), ('SVM', svm_model),
                                    final_estimator=LogisticRegression(max_iter=500, random_sta
voting_model = VotingClassifier(estimators=[('RandomForest', rf_model), ('SVM', svm_model), ('C
                                voting='soft')

# Advanced Models Dictionary
advanced_models = {"Random Forest": rf_model, "SVM": svm_model, "XGBoost": xgb_model,
                   "CatBoost": catboost_model, "Stacking Classifier": stacking_model, "Voting C

# Evaluate Advanced Models
```

```
advanced_results = {}
print("\n===== Advanced Models Evaluation =====\n")
for name, model in advanced_models.items():
    advanced_results[name] = evaluate_model(model, name, X_train, X_val, X_test, y_train, y_val
```

```
===== Advanced Models Evaluation =====

Results for Random Forest:
             Train  Validation     Test
F1 Score  0.915526    0.884651  0.86377

Results for SVM:
             Train  Validation      Test
F1 Score  0.959183    0.917354  0.905048

Results for XGBoost:
             Train  Validation      Test
F1 Score  0.999537    0.929351  0.925939

Results for CatBoost:
             Train  Validation      Test
F1 Score  0.959164    0.902142  0.904008

Results for Stacking Classifier:
             Train  Validation      Test
F1 Score  0.999079    0.945574  0.936981

Results for Voting Classifier:
             Train  Validation      Test
F1 Score  0.990571    0.949492  0.938034
```

Stacking and Voting Classifier was close, but we opted for Voting Classifier due to the upper edge in Train and Validation. The Voting Classifier outperformed the other five by combining outputs from RandomForest, SVM, XGBoost, CatBoost, and potentially stacking - it reduced each model's weaknesses. This approach was more consistent and effective than relying on any single model alone.

3.3 Refine Top Performing Models by Tuning Hyperparameters

Tried to tune hyperparameters by modifying hyperparameters with different models (for basic and advanced models). As a result, the best model for basic is K-NN, and the best model for advanced is the Voting Classifier.

3.4 Voting Classifier is the best Advanced Model (Using K-Fold Cross-Validation)

Created best model and used cross-validation for robust performance estimates

In [56]:
```
from sklearn.model_selection import KFold, cross_val_score

# Define models
rf_model = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)
svm_model = SVC(kernel='poly', C=1, probability=True, random_state=42)
catboost_model = CatBoostClassifier(iterations=100, learning_rate=0.2, depth=6, verbose=0, rand

# Create Voting Classifier
best_voting_model = VotingClassifier(estimators=[('RandomForest', rf_model), ('SVM', svm_model)

# K-Fold Cross Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Perform cross-validation on the training set
cv_scores = cross_val_score(best_voting_model, X_train_rfe, y_train, cv=kf, scoring='f1_weighte

# Train the model on the full training set
best_voting_model.fit(X_train_rfe, y_train)

# Evaluate on validation and test sets
y_val_pred = best_voting_model.predict(X_val_rfe)
y_test_pred = best_voting_model.predict(X_test_rfe)
y_train_pred = best_voting_model.predict(X_train_rfe)

# Get performance for each dataset
train_f1 = get_metrics(y_train, y_train_pred)
```

```
val_f1 = get_metrics(y_val, y_val_pred)
test_f1 = get_metrics(y_test, y_test_pred)

results_df = pd.DataFrame({"Cross-Validation Mean F1": [np.mean(cv_scores)], "Train F1": [train
                           "Validation F1": [val_f1], "Test F1": [test_f1]})
print(results_df)
```

```
   Cross-Validation Mean F1                           Train F1  \
0                  0.932643  {'F1 Score': 0.9868092405445982}

                     Validation F1                          Test F1
0  {'F1 Score': 0.9479572080114671}  {'F1 Score': 0.9296789699712937}
```

4.0 Final Model Selection and Implementation

4.1 k-NN

k-NN is the best model in terms of Train, Validation and Test. Among the baseline models, k-NN performed best because it uses distances between data points to classify songs. That approach aligned well with this dataset's feature spread, letting k-NN find good neighbors more accurately than the simpler linear (Logistic Regression) or split-based (Decision Tree) method. This can serve as a good starting point as it shows a high F1-score and validity - predicting the song's genres while minimising overfitting. However, it may not overall have an overview of the data.

In [60]:
```
knn_model  = Pipeline([("scaler", StandardScaler()),("classifier", KNeighborsClassifier(n_neigh

# Train
knn_model.fit(X_train_rfe, y_train)

# Export the predictions for the unlabeled data
X_unlabeled_final = X_train_unlabeled[best_features]
y_unlabeled_pred_knn = knn_model.predict(X_unlabeled_final)
y_unlabeled_labels_knn = le.inverse_transform(y_unlabeled_pred_knn)

# Save KNN predictions to CSV
test_pred_knn = pd.DataFrame({'Id': df_unlabeled_raw['Id'], 'top genre': y_unlabeled_labels_knn
test_pred_knn.to_csv("Predicted_Genres_KNN.csv", index=False)
```

4.2 Voting

Ultimately, the final advanced model was selected. After evaluating multiple advanced models, the Voting Classifier was chosen as the final model due to its superior performance across training, validation, and test sets. The Voting Classifier combines Random Forest, Support Vector Machine (SVM), and XGBoost in a soft-voting strategy, by taking each model's prediction into account and producing an overall "vote". This also takes the strengths of each model while offsetting any single model's weaknesses. This is because each algorithm specialises in different decision boundaries or features of the data, the combined result tends to be more stable and accurate than relying solely on a single classifier's output.

The model was optimised using Stratified K-Fold Cross-Validation (K=5), ensuring that performance estimates were reliable and not biased by a single train-test split. The cross-validation mean F1 score was 0.9326, demonstrating strong consistency across different data folds. After training on the full dataset, the model achieved an F1 score of 0.9868 on the training set, 0.9479 on the validation set, and 0.9297 on the test set. This indicates that the model maintains high generalisation capability without severe overfitting.

In [62]:
```
# Export the predictions for the unlabeled data
X_unlabeled_final = X_train_unlabeled[best_features]
y_unlabeled_pred_voting = best_voting_model.predict(X_unlabeled_final)
y_unlabeled_labels_voting = le.inverse_transform(y_unlabeled_pred_voting)

# Save KNN predictions to CSV
test_pred_voting = pd.DataFrame({'Id': df_unlabeled_raw['Id'], 'top genre': y_unlabeled_labels_
test_pred_voting.to_csv("Predicted_Genres_voting.csv", index=False)
```

5.0 Kaggle

On the Kaggle Leaderboards, the baseline model (k-NN) produced a score of 0.035. The chosen advanced model, Voting Classifier severely outperformed the baseline model, with a score of 0.44. This showcases that our model has an effective ability to predict the genre of the songs in the test dataset. The current model had successfully captured behaviours and patterns from the training dataset, further indicating that our feature selection and pre-processing was effective. However, the prediction score from the Kaggle leaderboards is based on 50% on the test data, and final results will be based on the other 50%. The final result of 0.44 could vary as the entrie dataset was not used.

6.0 Recommendation

To further enhance model performance and reduce the gap between training, validation, and test scores, the hyperparameter tuning should be optimised using techniques like grid search or Bayesian optimization to fine-tune parameters for each model in the Voting Classifier, ensuring a balance between bias and variance. This includes adjusting tree depth and estimators in Random Forest and XGBoost, as well as kernel selection in SVM. Moreover, feature selection and engineering should be applied to remove irrelevant or redundant features using techniques like PCA, helping the model focus on the most influential variables while avoiding unnecessary complexity. Lastly, regularisation techniques such as L1/L2 penalties, pruning for tree-based models, and early stopping in XGBoost can be implemented to prevent overfitting and improve generalisation. These strategies will help create a more robust model with consistent performance across different datasets.

7.0 Concluding Remarks

Overall, the Voting Classifier was the best model created to predict the top genre that a song belongs to. With different models created ranging from simple to advanced ones, the Voting Classifier stood out as the best algorithm with the most consistent F1 scores for all the train, validation, and test sets. The K-nearest neighbour model proved to be the most effective out of the baseline models, however, it could not compete with the voting classifier with its effective handling of complex data and flexibility. As well as the actual model building, different feature engineering methods were utilised such as one hot encoding to convert categorical data into a numerical format and power transformer to help stabilise variances.

The model was further able to fit the Kaggle dataset very well with a strong score of 0.44, suggesting it was able to successfully capture unknown patterns and trends. However as previously mentioned, it is important to note that the Kaggle dataset is based on only 50% of the test data, therefore introducing the possibility of a skewed result. Nonetheless, this report produced an impressive classifier model to help predict the top genre of a song.