SB 2.2 Statistical Machine Learning Practical

P135-P151-P782-P211

2491 Words

Contents

1 Introduction

This report focuses on forecasting the genre of a song using a dataset including 518 characteristics derived from 8,000 audio files. Each song is characterised by a vector of statistical summaries, including the mean, standard deviation, skewness, kurtosis, median, minimum, and maximum, obtained from time series data of musical properties such as the chromagrams or Mel-frequency cepstra via the `librosa` python package. We aim to create a classifier that can accurately determine the genre from a selection of eight categories: Electronic, Experimental, Folk, Hip-Hop, Instrumental, International, Pop, and Rock. As a benchmark, we reference the initial finding using a 5-nearest neighbour classifier with 35% prediction accuracy on the unseen test set.

2 Exploratory Data Analysis

The training dataset includes $6,000$ observations on 518 features. Our exploratory data analysis indicates that the classes in the training dataset are evenly distributed across all eight genres, as shown in Figure 1.

Figure 1: Distribution of Classes in the Training Set

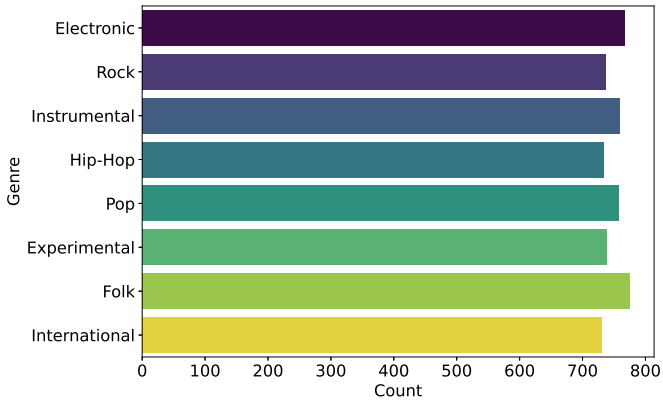


Figure 2: Distribution of Classes in the Training Set

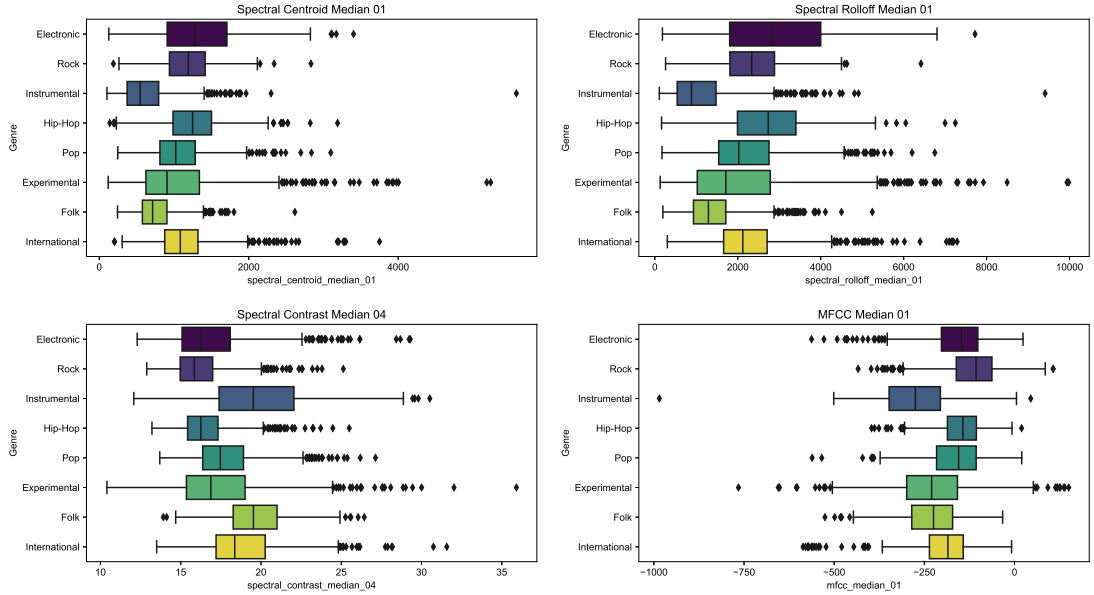


Figure 2 displays boxplots for the median of selected feature categories: spectral centroid, spectral rolloff, spectral contrast, and MFCC. Spectral centroid and spectral rolloff have unique distributions in particular genres, indicating their potential to distinguish certain music styles. In contrast, spectral contrast exhibits fluctuation but displays increased overlap between genres, suggesting a weaker capacity to distinguish between them.

As Figure 3 shows, many features are highly correlated. Though not necessarily problematic, it complicates attempts at feature engineering to reduce the number of features. Figure 4 shows that in order to capture 90% of the variance in the training set, we would need roughly 130 principal components. This would not be helpful in creating models that are easily interpretable, and in our experiments quickly led to reduced test accuracies compared to models that utilize the full training dataset. We therefore decide not to reduce the number of features.

Figure 3: Correlation between Selected Features for Categories in the Training Set

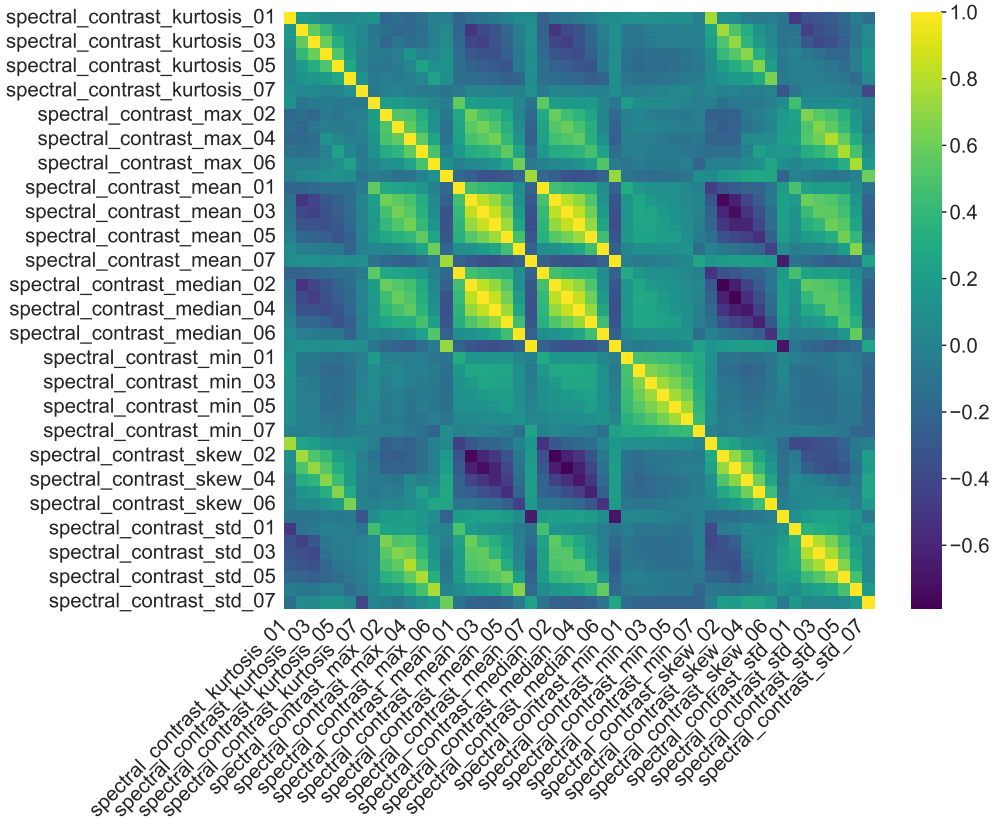
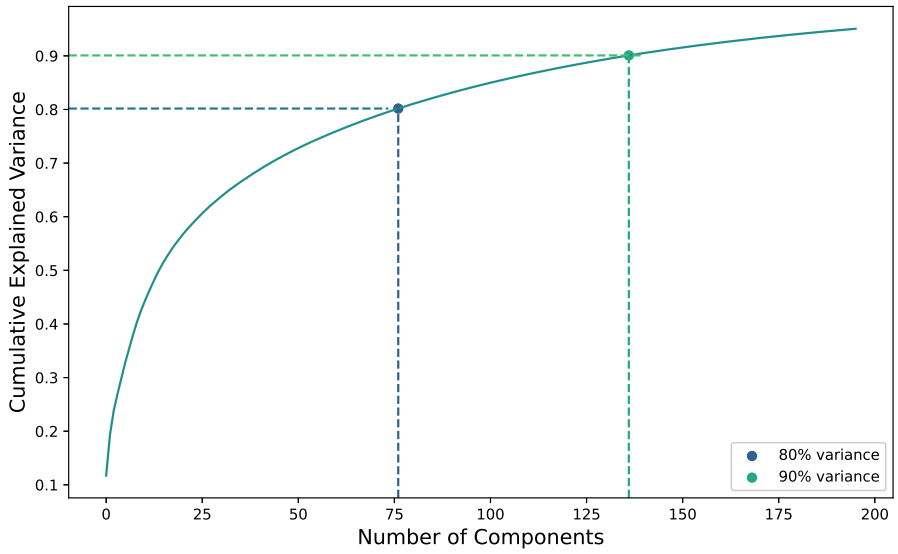


Figure 4: Cumulative Explained Variance by Principle Components



3 Data Preprocessing and Splitting

To estimate our test accuracy and tune hyperparameters, we set aside the unlabelled test data and split the training data as follows: we allocate 60% to the training subset and reserve 40 % for validation and pseudo-test subsets. The latter subset is split again into equal proportions, leaving us with a $60 - 20 - 20$ training-validation-pseudo-test overall split, ensuring a standard proportion for model evaluation. We choose not to stratify these splits based on our finding that class labels are balanced as shown in Figure 1. Given that we set the unlabelled test set aside and use the pseudo-test set to estimate the test accuracy on the unlabelled test set, we subsequently use *test accuracy* to refer exclusively to accuracy computed on the pseudo-test set, as the accuracy of our models on the real test set is unknowable. We also round the pseudo-test and training accuracies for all models except our final model to the nearest integer.

Considering the substantial differences in the ranges of the variables present in the unprocessed dataset, as evidenced by the drastically differing scales in Figure 2. This standardization is implemented using the `StandardScaler` from the `Scikit-learn` library. It is worth noting that we fit these three scalers only after splitting the training data to prevent information leakage, which could introduce bias and inaccurately inflate our models' performance metrics.

4 Baseline Models

We experiment with several common machine learning approaches as standalone models and obtain their training and test accuracies (Table 1). We outline their respective advantages and disadvantages (Table 2). We now briefly describe our model selection and implementation rationale but do not discuss results in detail in the name of brevity.

Table 1: Prediction accuracy (%) of selected base models on training and test datasets

Model	Training Accuracy (%)	Test Accuracy (%)
Multinomial Regression	84	51
Random Forest	100	56
KNN Bagging	52	40
SVC	82	59
XGBoost	100	58

Table 2: Comprehensive comparison of advantages and disadvantages of the models considered

Model	Advantages	Disadvantages	Comp. Complexity [1]
Multinomial Regression	Interpretability, efficiency in Training	Limited Capacity to Capture Complex Relationships, Susceptibility to Overfitting in High Dimensions	$\mathcal{O}(10,000 \cdot n \cdot f \cdot m)$
Random Forests	Handles non-linear data, robust to overfitting	Slow with many trees, complex models hard to interpret	$\mathcal{O}(200 \cdot n \cdot \log(n) \cdot f)$
KNN Bagging	Captures complex boundaries, robust	Computationally expensive, sensitive to noise	$\mathcal{O}(n \cdot f)$
SVC	Effective in high dimensions	Poor with noise and overlap, kernel choice	$\mathcal{O}(m^2 \cdot n^2 \cdot f)$ to $\mathcal{O}(m^2 \cdot n^3 \cdot f)$
XGBoost	High performance and speed, iterative learning	Can overfit, complex to understand	$\mathcal{O}(n_{\text{estimators}} \cdot n \cdot f \cdot \text{max\_depth})$
Neural Network	Flexible, models complex relationships	Requires significant computation, prone to overfitting	Depends on architecture

[1] n = number of samples, m = number of classes, f = number of features

4.1 Multinomial Logistic Regression

Initially, we use a multinomial logistic regression classifier that utilises a softmax function to generate class predictions. We choose the multinomial technique over the one-vs-rest strategy (OvR) as OvR performs poorly with insufficient differentiation for a particular class based on feature values. This can be seen in the provided examples shown in Figure 2. We specify a limit of $10,000$ iterations to guarantee convergence.

Although this model is highly interpretable and computationally simple (Table 2), we chose not to continue with it because we want our final model to capture non-linear relations. Furthermore, multinomial regression performs well if characteristics are independent from one another, which in our dataset is not the case (Figure 3).

4.2 K-Nearest Neighbours (KNN)

To probe whether more complex K-Nearest Neighbours models can outperform the benchmark basic KNN of accuracy of 35%, we implement KNN Bagging (Bootstrap Aggregating), which reduces the variance of individual models and exposes the constituent models to different parts of the dataset. We employ a `BaggingClassifier` on `KNeighborsClassifier` with `n_neighbor = 1` and `n_estimators = 10` base estimators, which overfits.

4.3 Support Vector Classifier (SVC)

Next, we investigate the use of a SVC that utilises a radial basis function (RBF) kernel based on the assumption of non-linear connections in the dataset, which is indirectly supported by the performance of the multinomial regression model. By using the RBF kernel, we transform our dataset into a higher-dimensional space, in which we expect to identify a linear decision boundary. The fitted SVC manages the high-dimensional data well and marginally outperforms most other base model but is computationally intensive.

4.4 Neural Networks (NN)

Table 3: Summary of Neural Network Model Performances

Model	No. HL	No. Neurons	LR [1]	DR [2]	Optimizer	OF [3]	Batch Norm	Train Acc. (%)	Test Acc. (%)
NN1	1	57	0.58	-	Adam	CEL	No	82	57
NN2	6	221	0.0007	0.27	Adam	CEL	Yes	79	57
NN3	10	2056	0.01	0.4	SGD M 0.9	CEL	Yes	76	47

[1] LR = Learning Rate
[2] DR = Dropout Rate
[3] OF = Objective Function

Additionally, we evaluate NNs motivated by the potential to capture complex, nonlinear relationships within high-dimensional datasets such as ours. We explore various fully-connected architectures, as outlined in Table 3. We use batch normalization to stabilize and accelerate training, dropout layers to regularize, and a cross-entropy loss function (CEL). Attempts at using convolutional architectures on our tabular data performed very poorly.

Although the accuracies in both `NN1` and `NN2` configurations are promising, the complexity of further optimization and the computational demands of deeper and more complex networks present practical constraints. Given these considerations, along with the negligible accuracy improvement over simpler models like Random Forests, SVCs, and Gradient Boosted Trees, we decide against adopting a neural network as our final model. Instead, we favor methods that offer a better balance between performance, computational efficiency, and interpretability.

4.5 Random Forests (RF)

As we implicitly assume that the relationships between our features and classes are non-linear given the performance of our multinomial logistic regression (Table 1) and the high-dimensional dataset, we decide to experiment with RFs. These models demonstrate robustness to correlation between variables by creating several decision trees and using a subset of features for each tree. They are attractive as they are able to capture non-linear relationships and scale well with high-dimensional datasets. We implement an RF classifier, configuring it with hyperparameters of: `n_estimators=200`, a maximum depth of 50 (`max_depth=50`), and `log_loss` as the criterion for quality of splits.

However, our RF models exhibits overfitting, suggesting poor generalizability. It is also computationally intensive, as each iteration considers a random subset of characteristics. As the number of trees and the depth of each tree rise, we choose to experiment with gradient boosted trees. Unlike RFs, which construct trees individually, boosted trees create forests consecutively, with each tree aiming to rectify the errors made by its predecessors.

4.6 Gradient-Boosted Decision Trees (XGB)

We initially implement a gradient-boosted decision tree classifier with the package `xgboost`, configuring it with specific hyperparameters: a `multi:softmax` objective function for the `num_class = 8` classes, with 200 trees `num_rounds = 200` of maximum depth of 30 (`max_depth = 30`), employing `mlogloss` (multi-class log loss) as the criterion for quality of splits. Compared to RFs, though the XGB model also overfits, the similarly high test accuracy of 58% and stark superiority in both computational complexity (Table 2) and interpretability make it an optimal base model to further experiment with.

5 Feature Subsetting and Stacked Modelling Approaches

After exploring the base models individually, we experiment with leveraging the hierarchical structure of the dataset. Given that for each musical feature, such as for example the chromagrams, seven summary statistics in each dimension are provided, we create feature subsets which include all summary statistics for each musical feature. On these feature subsets, we first fit separate models which make the final prediction via weighted voting from the validation accuracies (Table 4).

The gradient boosted decision tree implemented using XGBoost achieves the highest performance on the feature subsets, followed by the SVC. Nevertheless, training the models on subsets of features and making predictions via weighted voting does not enhance the performance of our models and is accompanied with increased computational cost.

Table 4: Hyperparameters and test accuracy (%) of models trained on subsets of features

Model	Final Prediction	Test Accuracy (%)
Multinomial Regression	Weighted Voting	53
Random Forest	Weighted Voting	54
KNN	Weighted Voting	51
SVC	Weighted Voting	57
XGBoost	Weighted Voting	57

Next, we try an ensemble technique. We choose the optimal model for each examined feature subset based on validation accuracy, weigh each model by the same and create an ensemble of these models (Table 5). The final prediction is made using weighted voting, obtaining 56.8% test accuracy.

Table 5: Best performing model and accuracy for each feature subset

Feature	Best Model	Test Accuracy (%)
chroma_cens	SVC	33
chroma_cqt	SVC	35
chroma_stft	RF	39
mfcc	SVC	56
rmse	RF	27
spectral_bandwidth	RF	33
spectral_centroid	SVC	38
spectral_contrast	SVC	47
spectral_rolloff	SVC	37
tonnetz	XGB	33
zcr	SVC	34
Test accuracy (ensemble)		56

From the previous feature subset ensemble method, we observe that for different feature subsets, different learners perform best with respect to accuracy (Table 4). Therefore, we decide to leverage the diverse advantages of multiple base learners, stack them, and construct a meta-learner for the final prediction. This strategy involves two main steps:

1. Generate meta-features by computing the class probability predictions from each base learner (e.g., RF, SVC, XGBoost) on the validation set
2. Train a logistic regression model (meta-learner) on these features.

We first generate the meta-features. We experimented with a RF and KNN comibination (Table 6), and a trio of RF, SVC, and XGB, however in the following we will focus on the latter. Each model predicts class probabilities for the validation set, which we consider as meta-features. For each feature subset identified as having good predictive potential, we train the base learners—specifically, RF, SVC, and XGB models. We choose these base learners for their complementary strengths: RF for its ensemble robustness, SVC for its effectiveness in high dimensions, and XGB for its performance in structured datasets. These three models also performed with highest accuracy for the respective feature subsets (Table 5). For each feature subset, we stack the class probabilities from RF, SVC, and XGB into a three-dimensional array, where each `slice` corresponds to one model's output. This yields three `slices`, each one being a two-dimensional array where rows represent samples and columns represent the predicted probabilities for each class by one of the base learners (RF, SVC, or XGB). When we stack these arrays, we are effectively layering these predictions to create a three-dimensional array, which we then average across the third dimension (across the base learners) to obtain our meta-features for the meta-learner to train on. By averaging, we aim to capture a common consensus. We follow an identical process to create the meta test set using the test set. For the meta-learner, we choose logistic regression, which learns to weigh these meta-features to make final predictions and is computationally efficient. Finally, we evaluate the meta-learner using the meta test set. The output of this process is a set of final predictions for the test set, and we calculate the test accuracy by comparing these predictions against the true labels.

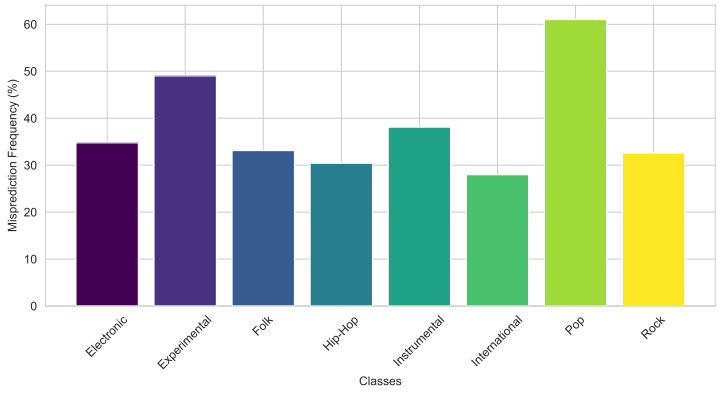
The stacked models, as presented in Table 6, achieve test accuracies that rank among the highest in our series of experiments. Particularly the stacked model of RF, XGBoost, and SVC, which utilized a Logistic Regression meta-learner, results in a test accuracy of 61.1%. Analyzing the misprediction frequencies per class displayed in Figure 5,

we observe that the stacked model performs better on the international class than our final model. However, on all remaining classes, the stacked model performs worse and the overall test accuracy is lower compared to our final model.

Table 6: Test Accuracies of Stacked Models

Model	Metalearner	Test Accuracy (%)
Random Forest and KNN	Logistic Regression	57
Random Forest, XGB, SVC	Logistic Regression	61

Figure 5: Stacked Model Misprediction Frequency by Class



Consequently, we eventually opt not to use this as our final model. The choice is motivated by factors of computing efficiency and model simplicity. The XGB model is a more practical option because of its better performance with respect to accuracy, lower computing burden, and clear nature, which makes it easier to use and analyse.

6 Final Model: Tuned Gradient-Boosted Trees (XGB)

The model we use to generate our final prediction is a tuned XGB model. We train our model using a hyperparameter tuning suite to find the optimal values for several key hyperparameters of our chosen implementation package `XGBoost`. Algorithm 1 describes the process of fitting numerous 'weak' decision trees in a sequential manner by calculating the gradient and Hessian of the loss function to guide the optimization of subsequent learners, thereby correcting the errors of the preceding ensemble. The algorithm incorporates regularization directly in the optimization process, which helps in preventing overfitting. Hyperparameters detailed in Algorithm 1 allow us to tune the model. The learning rate (η) is particularly significant as it dictates the adjustment size at each step, influencing both the speed of convergence and the risk of overshooting the optimal solution. By employing the `multi:softmax` objective, the algorithm predicts labels for each class. Similar to RFs, XGBoost optimizes between fitting the model closely to the training data and maintaining a generalization to avoid overfitting ("XGBoost Documentation — xgboost 2.0.3 documentation", 2024).

Algorithm 1 Gradient Boosting Algorithm, based on Hastie (2017)

Input: training set $\{(x_i, y_i)\}_{i=1}^N$, a differentiable loss function $L(y, F(x))$, a number of weak learners M, and a learning rate η

Loss Function: Multiclass Logloss := $L(y, F(x)) = -\sum_{k=1}^{K} \mathbb{I}(y = k) \log \left(\frac{\exp(F_k(x))}{\sum_{j=1}^{K} \exp(F_j(x))} \right)$

Objective function: 'multi:softmax' for multiclass classification with 8 classes

Hyperparameters:

Maximum depth of trees: trial.suggest_int('max_depth', 3, 100)

Learning rate η: trial.suggest_float('eta', 0.005, 0.4)

Subsample ratio of the training instances: trial.suggest_float('subsample', 0.6, 1.0)

Subsample ratio of columns when constructing each tree: trial.suggest_float('colsample_bytree', 0.6, 1.0)

Minimum loss reduction required to make a further partition on a leaf trial.suggest_float('colsample_bytree', 0.2, 0.7)

Initialize model with a constant value: $f_0(x) = \arg\min_\theta \sum_{i=1}^{N} L(y_i, \theta)$.

Tuning Process: `validation_accuracies` = [], `trials` = []

for i in 1,100 **do**

  params=suggest hyperparameter values

  for `num_boosting_round=5,000` **do**

    Compute the gradients and Hessians for training data:

    $\hat{g}_m(x_i) = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f(x)=f_{m-1}(x)}$

    $\hat{h}_m(x_i) = \left[\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)^2} \right]_{f(x)=f_{m-1}(x)}$

    Fit a base learner to the gradients and Hessians, including regularization:

    $\phi_m = \arg\min_\phi \sum_{i=1}^{N} \left[\frac{1}{2} \hat{h}_m(x_i) (\phi(x_i) - \frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)})^2 + \lambda \|\phi\|^2 \right]$.

    Scale the base learner's contribution with the learning rate: $f_m(x) = \eta \phi_m(x)$.

    Update the model: $f_m(x) = f_{m-1}(x) + f_m(x)$.

    Evaluate model on validation set and calculate accuracy: accuracy$(Y_{\text{val}}, f_m(X_{\text{val}}))$.

    Apply early stopping if validation accuracy does not improve for 15 rounds.

  end for

  Return trial[i] = accuracy, params

end for

Obtain optimal parameters :arg max `validation_accuracies`.params

Retrain on combined training and validation sets using optimal parameters for `num_boosting_round=10,000`

Output: The final model $f(x) = f_M(x) = \sum_{m=0}^{M} f_m(x)$.

Note: This entire process takes about 1 hour on a laptop CPU, prediction with the trained model takes milliseconds

6.1 Final Results

Given our tuning results, we arrive at the following values for our final model (section 9).

Table 7: Hyperparameter search space and selected values for model tuning.

Hyperparameter	Search space	Selected value
Max depth	{5, ..., 100}	86
Eta	{0.005, ..., 0.4}	0.089
Subsample	{0.6, ..., 1.0}	0.71
Colsample by tree	{0.6, ..., 1.0}	0.78
Gamma	{0.0, ..., 5.0}	0.32

Using this final model, we compute the **training and test set accuracies of 100% and 64%, respectively**.

Figure 6 displays the 10 most important features in the final model based on 4 different metrics: the features' weight, gain, cover, and total gain (Quinto, 2020). We can that individual features related to the Mel-frequency cepstra and spectra dominate across all four importance metrics.

Figure 6: Final Model Feature Importance

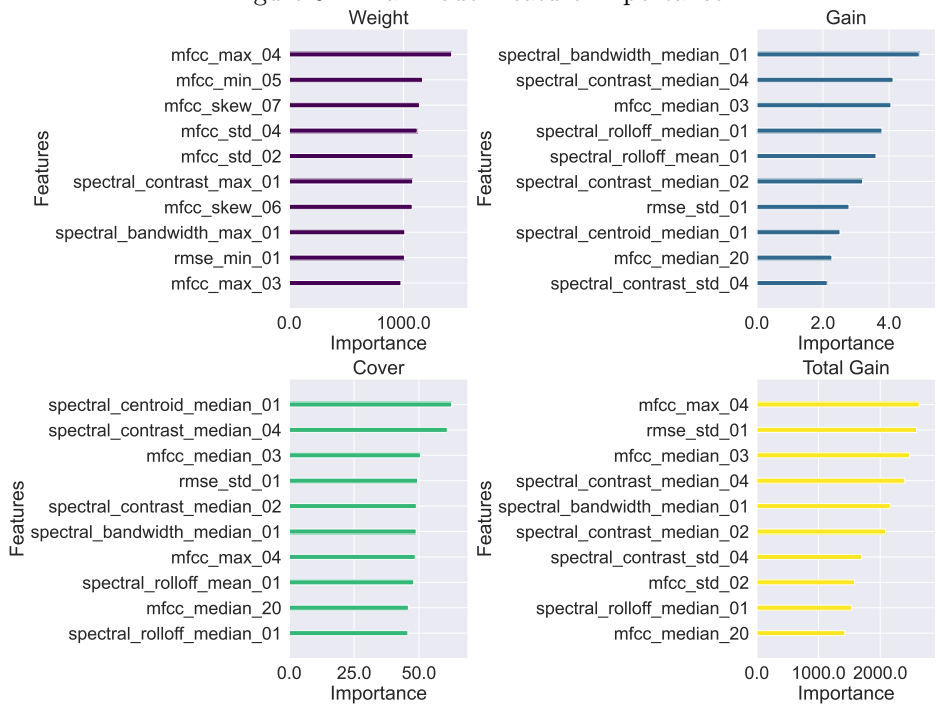
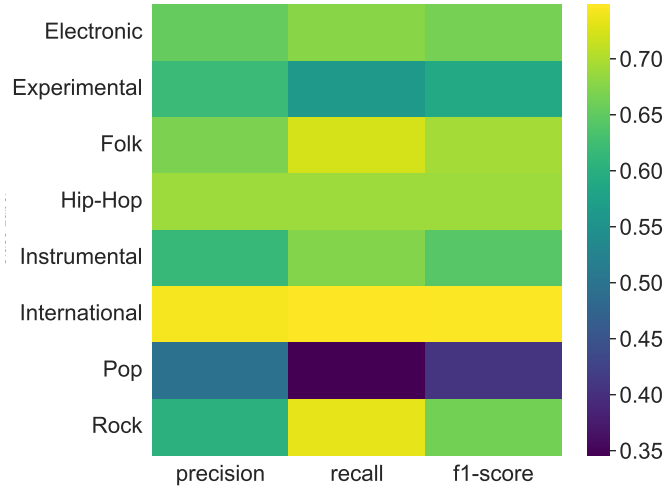


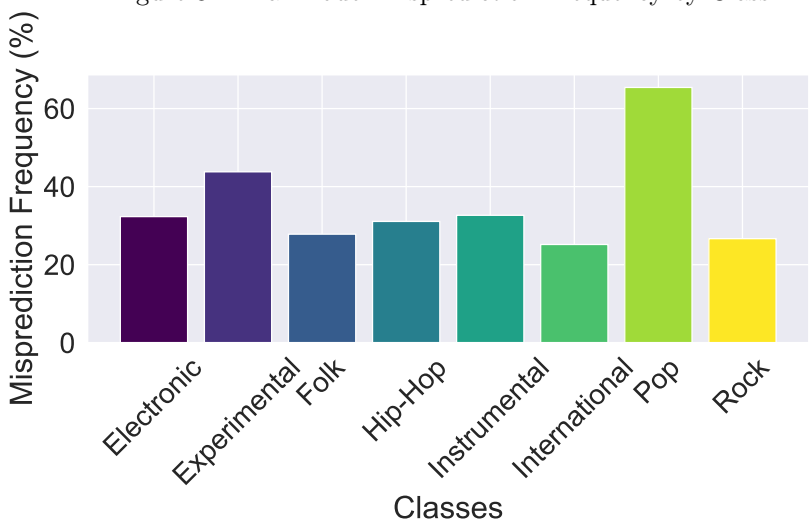
Figure 7 displays the performance of our final model across all 8 genres on our pseudo-test set, as measured by the precision, recall, and f1-scores, which are high for most genres, with the notable exception of genres *Pop* and *Experimental*. For these genres, our model exhibits poor recall, indicative of a high false negative rate.

Figure 7: Final Model Performance by Class



This poor recall is more clearly visible in Figure 8, where we can see that our model misclassifies more than 60% of observations from the *Pop* genre in the test set, and more than 40% of *Experimental* observations. As both genres are well-represented in our data (Figure 1), the poor performance of our model for them is the limiting factor in not achieving higher test set accuracy.

Figure 8: Final Model Misprediction Frequency by Class



If the real test data are similarly distributed to our training data (and therefore training, validation, and pseudo-test subsets), and contain identical genres, **we conservatively estimate our final model's accuracy on the test set to be 60%. This corresponds to a generalisation error of 40%.**

7 Conclusion

We implement various base models, experiment with feature subsetting and stacked modelling approaches before choosing to optimize the XGBoost model, given its high accuracies and computational simplicity. On our best performing final model, we achieve robust performance on most genres although our model overfits to the training data. If given actual audio files, future improvements can include training NNs directly on audio data, or better feature extraction with different libraries.

References

Hastie, T. (2017). *The elements of statistical learning: Data mining, inference, and prediction* (Second edition.). Springer. (Cit. on p. 10).

Quinto, B. (2020). *Next-Generation Machine Learning with Spark: Covers XGBoost, LightGBM, Spark NLP, Distributed Deep Learning with Keras, and More* (1st ed.). Apress L. P. https://doi.org/10.1007/978-1-4842-5669-5 (cit. on p. 11).

XGBoost Documentation — xgboost 2.0.3 documentation. (2024). Retrieved March 18, 2024, from https://xgboost.readthedocs.io/en/stable/ (cit. on p. 9).

8 Appendix A: Source Code

```
# Library Import
import getpass
import os
import shutil
from collections import Counter
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import xgboost as xgb
from matplotlib.ticker import FuncFormatter
from sklearn.decomposition import PCA
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler


# Function to generate final submission csv file
def generate_submission_csv(genre_predictions, filename="submission.csv"):
    submission_df = pd.DataFrame(data={
        "Id": range(len(genre_predictions)),
        "Genre": genre_predictions
    })
    submission_df.to_csv(filename, index=False)
    print(f"Submission file '{filename}' created successfully.")


# Function to compute pseudo test set accuracy
def calculate_pseudo_test_accuracy(predictions):
    print(f"Pseudo Test Set accuracy: {accuracy_score(Y_test, predictions):.2f}")


# Function to compute training set accuracy
def calculate_training_accuracy(predictions):
    print(f"Training Set accuracy: {accuracy_score(Y_train, predictions):.2f}")


# Load the training data and the test inputs
x_train = pd.read_csv('Data/X_train.csv', index_col=0, header=[0, 1, 2])
x_train_np = np.array(x_train)
y_train = pd.read_csv('Data/y_train.csv', index_col=0)
y_train_np = y_train.squeeze().to_numpy()  # Make y_train a NumPy array
```

```
x_test = pd.read_csv('Data/X_test.csv', index_col=0, header=[0, 1, 2])
x_test_np = np.array(x_test)

# Flatten the columns for easier wrangling
x_train_flat_columns = ['_'.join(col).strip() for col in x_train.columns.values]
x_train.columns = x_train_flat_columns

x_test_flat_columns = ['_'.join(col).strip() for col in x_test.columns.values]
x_test.columns = x_train_flat_columns

# Label-encode training labels
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train_np.ravel())  #

# Split training data into training and temporary validation sets
X_train, X_temp, Y_train, Y_temp = train_test_split(x_train, y_train_encoded, test_size=0.4,
↪  random_state=42)

# Split the temporary validation set into validation and pseudo test set
X_val, X_test, Y_val, Y_test = train_test_split(X_temp, Y_temp, test_size=0.5, random_state=42)

# Standardise respective subsets after splitting to avoid data leakage
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
X_real_test_scaled = scaler.transform(x_test)  # real test set to generate submission on

# Load best XGB model
final_model_name = 'Models/xgboost-64%-all-data'
final_booster = xgb.Booster()  # instantiate
final_booster.load_model(final_model_name)  # load
train_predictions = final_booster.predict(xgb.DMatrix(X_train_scaled))  # predict on train set
pseudo_test_predictions = final_booster.predict(xgb.DMatrix(X_test_scaled))  # predict on
↪  pseudo-test set
real_test_predictions = final_booster.predict(xgb.DMatrix(X_real_test_scaled))  # predict on
↪  real test set

# Decode numeric predictions to string labels
genre_predictions_decoded = label_encoder.inverse_transform(real_test_predictions.astype(int))

# Make submission csv with decoded predictions
generate_submission_csv(genre_predictions_decoded, filename="submission.csv")

# ######## MAKE PLOTS ########
```

```
export_username = "ts"  # Only save plots to dropbox on right machine


# Function to save plots to EPS for overleaf
def save_plot(plot, filename):
    username = getpass.getuser()
    filepath = "/Users/ts/Library/CloudStorage/Dropbox/Apps/Overleaf/SML Practical/Figures"
    filename += ".eps"
    if username == export_username:
        plot.savefig(os.path.join(filepath, filename), format='eps')  # Save as EPS
        print("Saved plot to {}".format(filename))


# Make EDA Plots

# PCA Plot
pca = PCA(n_components=0.95)
X_train_pca = pca.fit_transform(X_train_scaled)
idx_full_80 = np.where(np.cumsum(pca.explained_variance_ratio_) >= 0.8)[0][0]
idx_full_90 = np.where(np.cumsum(pca.explained_variance_ratio_) >= 0.9)[0][0]
pcaplot = plt.figure(figsize=(10, 6))

# Plot the cumulative explained variance
cumulative_variance = np.cumsum(pca.explained_variance_ratio_)
plt.plot(cumulative_variance, color=plt.cm.viridis(0.5))
plt.xlabel('Number of Components', fontsize=14)
plt.ylabel('Cumulative Explained Variance', fontsize=14)
plt.yticks(np.arange(0, 1, step=0.1))

y_80 = cumulative_variance[idx_full_80]
y_90 = cumulative_variance[idx_full_90]

# noinspection PyTypeChecker
plt.axvline(x=idx_full_80, ymax=y_80, color=plt.cm.viridis(0.3), linestyle='--')
# noinspection PyTypeChecker
plt.axhline(y=y_80, xmax=idx_full_80 / len(cumulative_variance), color=plt.cm.viridis(0.4),
    ↪ linestyle='--')
# noinspection PyTypeChecker
plt.axvline(x=idx_full_90, ymax=y_90, color=plt.cm.viridis(0.6), linestyle='--')
# noinspection PyTypeChecker
plt.axhline(y=y_90, xmax=idx_full_90 / len(cumulative_variance), color=plt.cm.viridis(0.7),
    ↪ linestyle='--')

# Scatter points with adjusted Viridis colors
plt.scatter(idx_full_80, y_80, color=plt.cm.viridis(0.3), label='80% variance')
```

```
plt.scatter(idx_full_90, y_90, color=plt.cm.viridis(0.6), label='90% variance')

plt.legend(loc='best')
save_plot(pcaplot, "pca")

# Class Balance Plot
viridis_colors = plt.cm.viridis(np.linspace(0, 1, 8))
custom_palette = [matplotlib.colors.rgb2hex(color) for color in viridis_colors]

class_bal = plt.figure(figsize=(10, 6))
sns.countplot(data=y_train, y='Genre', palette=custom_palette)
plt.xlabel('Count', fontsize=16)
plt.ylabel('Genre', fontsize=16)
plt.yticks(fontsize=16)
plt.xticks(fontsize=16)
plt.subplots_adjust(left=0.2, right=0.9, top=0.9, bottom=0.1)
save_plot(class_bal, "Class-Balance")

x_train_with_genre = x_train.merge(y_train, left_index=True, right_on='Id')  # Merge Genre
↪  labels on to training data
box1, axs = plt.subplots(nrows=2, ncols=2, figsize=(16, 9))  # Create the subplots
sns.boxplot(x='spectral_centroid_median_01', y='Genre', data=x_train_with_genre, ax=axs[0, 0],
↪  palette=custom_palette)
axs[0, 0].set_title('Spectral Centroid Median 01')
sns.boxplot(x='spectral_rolloff_median_01', y='Genre', data=x_train_with_genre, ax=axs[0, 1],
↪  palette=custom_palette)
axs[0, 1].set_title('Spectral Rolloff Median 01')
sns.boxplot(x='spectral_contrast_median_04', y='Genre', data=x_train_with_genre, ax=axs[1, 0],
↪  palette=custom_palette)
axs[1, 0].set_title('Spectral Contrast Median 04')
sns.boxplot(x='mfcc_median_01', y='Genre', data=x_train_with_genre, ax=axs[1, 1],
↪  palette=custom_palette)
axs[1, 1].set_title('MFCC Median 01')
sns.set(font_scale=2)  # Adjust the font scale for better readability
plt.tight_layout()

save_plot(box1, "boxplot-1")

# Correlation matrix
df_corr = X_train.filter(like='spectral_contrast')
corr_mat = df_corr.corr()
cormat = plt.figure(figsize=(16, 13))
sns.heatmap(corr_mat, cmap='viridis')
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels
plt.xlabel('')  # Remove x-axis title
```

```
plt.ylabel('')  # Remove y-axis title
plt.tight_layout()
save_plot(cormat, "correlation")

# Get decoded class labels for plots
y_test_decoded = label_encoder.inverse_transform(Y_test)
pseudo_test_preds_labels = label_encoder.inverse_transform(pseudo_test_predictions.astype(int))

calculate_training_accuracy(train_predictions)
calculate_pseudo_test_accuracy(pseudo_test_predictions)

# Make XGB Visualizations
# Retrieve column names
feature_names = x_train_flat_columns
# Custom formatter to one decimal place
formatter = FuncFormatter(lambda x, _: f'{x:.1f}')

# Define a list of colors for the bar plots
colors = plt.cm.viridis(np.linspace(0, 1, 4))

# Create the subplots with constrained_layout instead of tight_layout
importanceplots, axs = plt.subplots(nrows=2, ncols=2, figsize=(16, 12),
↪  constrained_layout=True)

# Define importance types and corresponding titles
importance_types = ['weight', 'gain', 'cover', 'total_gain']
titles = ['Weight', 'Gain', 'Cover', 'Total Gain']

# Plot importance for each type
for i, ax in enumerate(axs.flat):
    xgb.plot_importance(final_booster, importance_type=importance_types[i],
    ↪  max_num_features=10, ax=ax,
                        show_values=False, color=colors[i])
    ax.xaxis.set_major_formatter(formatter)
    ax.set_xlabel('Importance')
    ax.set_title(titles[i])
    ticks = ax.get_yticklabels()
    indices = [int(tick.get_text().replace('f', '')) for tick in ticks]
    new_labels = [feature_names[i] for i in indices]
    ax.set_yticklabels(new_labels)

save_plot(importanceplots, "XGB-Importance")

# Plot Misprediction Frequency by class
# Calculate mispredictions
```

```
mispredictions = (y_test_decoded != pseudo_test_preds_labels)

# Count the total occurrences for each class in the true test set
total_counts = Counter(y_test_decoded)

# Count mispredictions for each decoded class
mispredicted_counts = Counter(y_test_decoded[mispredictions])

# Calculate misprediction frequencies as a percentage
misprediction_freq = {class_label: (mispredicted_counts.get(class_label, 0) /
    ↪ total_counts[class_label]) * 100
                      for class_label in total_counts}

# Sort the classes by name to maintain consistent order
sorted_class_labels = sorted(total_counts.keys())

# Prepare colors, one for each class
colors = plt.cm.viridis(np.linspace(0, 1, len(sorted_class_labels)))

# Bar chart of misprediction frequencies (as percentages)
xgb_mispred_freq = plt.figure(figsize=(10, 6))
plt.bar(sorted_class_labels, [misprediction_freq[class_label] for class_label in
    ↪ sorted_class_labels], color=colors)
plt.xlabel('Classes')
plt.ylabel('Misprediction Frequency (%)')
plt.xticks(ticks=range(len(sorted_class_labels)), labels=sorted_class_labels, rotation=45)
plt.subplots_adjust(bottom=0.4)  # Increase the bottom margin

save_plot(xgb_mispred_freq, "xgb_mispred_freq")

# Classification Report Heatmap
# Plot the classification report as a heatmap
report_dict = classification_report(Y_test, pseudo_test_predictions, output_dict=True)
report_df = pd.DataFrame(report_dict).transpose()
# Extract unique class names in the correct order from y_test_decoded
unique_class_names = label_encoder.inverse_transform(sorted(np.unique(Y_test)))

# Drop the 'support' column and rows with averages, since we only want the individual classes
report_df = report_df.drop(columns=['support'])
class_report_df = report_df.iloc[:-3, :]
heatmap = plt.figure(figsize=(10, 8))
sns.heatmap(class_report_df, cmap='viridis', cbar=True, fmt='.2g',
            annot_kws={'color': 'black'},  # Add contrasting color for readability
            yticklabels=unique_class_names)
plt.ylabel('Class Label', fontsize=14)
```

```
heatmap.subplots_adjust(left=0.2)
save_plot(heatmap, "XGB-Heatmap")

# Copy code to overleaf
shutil.copy('submission.py', '/Users/ts/Library/CloudStorage/Dropbox/Apps/Overleaf/SML Practical/Code')
shutil.copy('tuning.txt', '/Users/ts/Library/CloudStorage/Dropbox/Apps/Overleaf/SML Practical/Code')
print("Source Code and Tuning log copied to Overleaf")
```

9 Appendix B: Tuning Log

```
/opt/homebrew/anaconda3/envs/sml-practical-env/bin/python /Users/ts/Git/sml-practical/Boosting.py
[I 2024-03-13 17:36:21,282] A new study created in memory with name: XGB
[I 2024-03-13 17:36:39,472] Trial 0 finished with value: 0.545 and parameters: {'max_depth': 26, 'eta': 0.2628146320956893, 'subsample': 0.7972713283357904, 'colsample_bytree': 0.816275172425584, 'gamma': 0.5549659330850426}. Best is trial 0 with value: 0.545.
[I 2024-03-13 17:39:28,514] Trial 1 finished with value: 0.575 and parameters: {'max_depth': 83, 'eta': 0.020858906103026254, 'subsample': 0.8283894956733198, 'colsample_bytree': 0.6947693012123138, 'gamma': 0.6296835502488504}. Best is trial 1 with value: 0.575.
[I 2024-03-13 17:39:42,549] Trial 2 finished with value: 0.5408333333333334 and parameters: {'max_depth': 42, 'eta': 0.3503523803700617, 'subsample': 0.7571401123742608, 'colsample_bytree': 0.7649244055417993, 'gamma': 0.6401155866869406}. Best is trial 1 with value: 0.575.
[I 2024-03-13 17:40:57,497] Trial 3 finished with value: 0.585 and parameters: {'max_depth': 94, 'eta': 0.04882205533678397, 'subsample': 0.6825997193130474, 'colsample_bytree': 0.6465288666735272, 'gamma': 0.5290770879296267}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:41:33,087] Trial 4 finished with value: 0.5533333333333333 and parameters: {'max_depth': 85, 'eta': 0.16435042060435348, 'subsample': 0.7992227416169095, 'colsample_bytree': 0.8015206920815696, 'gamma': 0.37576383381061895}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:41:51,269] Trial 5 finished with value: 0.555 and parameters: {'max_depth': 40, 'eta': 0.3625490378281487, 'subsample': 0.8056596870782603, 'colsample_bytree': 0.6592098943485736, 'gamma': 0.5428923898809522}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:43:02,761] Trial 6 finished with value: 0.5758333333333333 and parameters: {'max_depth': 75, 'eta': 0.04537194641541596, 'subsample': 0.679130829667404, 'colsample_bytree': 0.7800798339847403, 'gamma': 0.29785200155444036}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:43:37,170] Trial 7 finished with value: 0.5633333333333334 and parameters: {'max_depth': 59, 'eta': 0.1687200748829207, 'subsample': 0.6021802720485504, 'colsample_bytree': 0.6919453360127953, 'gamma': 0.6913331782269285}. Best is trial 3 with value: 0.585.
```

```
[I 2024-03-13 17:43:52,781] Trial 8 finished with value: 0.5541666666666667 and parameters: {'max_depth': 47, 'eta': 0.20592041602172748, 'subsample': 0.7850964466185477, 'colsample_bytree': 0.6347492211226203, 'gamma': 0.30452006633737644}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:44:56,698] Trial 9 finished with value: 0.5691666666666667 and parameters: {'max_depth': 80, 'eta': 0.053638238084523594, 'subsample': 0.6299265913130072, 'colsample_bytree': 0.8073265770602875, 'gamma': 0.5025625193126635}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:45:13,804] Trial 10 finished with value: 0.5733333333333334 and parameters: {'max_depth': 3, 'eta': 0.09787092974584016, 'subsample': 0.6954912139222134, 'colsample_bytree': 0.6104571344775431, 'gamma': 0.4094001086633518}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:45:45,405] Trial 11 finished with value: 0.5841666666666666 and parameters: {'max_depth': 100, 'eta': 0.09507499065807604, 'subsample': 0.6855439825040089, 'colsample_bytree': 0.7328098007409967, 'gamma': 0.20387053382531892}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:46:11,459] Trial 12 finished with value: 0.58 and parameters: {'max_depth': 99, 'eta': 0.10878967781596152, 'subsample': 0.6708412443823324, 'colsample_bytree': 0.7327974059637782, 'gamma': 0.21190555611177578}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:47:08,524] Trial 13 finished with value: 0.5691666666666667 and parameters: {'max_depth': 99, 'eta': 0.10312157388757899, 'subsample': 0.7261436795367667, 'colsample_bytree': 0.7354178428756479, 'gamma': 0.46965857279972395}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:50:00,051] Trial 14 finished with value: 0.5783333333333334 and parameters: {'max_depth': 66, 'eta': 0.013699170602828055, 'subsample': 0.6499973619411648, 'colsample_bytree': 0.6797753455414872, 'gamma': 0.2258549695623622}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:50:18,900] Trial 15 finished with value: 0.55 and parameters: {'max_depth': 100, 'eta': 0.2577701353965642, 'subsample': 0.724787824449863, 'colsample_bytree': 0.8446684042030116, 'gamma': 0.357447568001234}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:51:02,368] Trial 16 finished with value: 0.5733333333333334 and parameters: {'max_depth': 90, 'eta': 0.14467043019902703, 'subsample': 0.7040434094685657, 'colsample_bytree': 0.6013823119846831, 'gamma': 0.4318725185160022}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:52:16,208] Trial 17 finished with value: 0.5675 and parameters: {'max_depth': 68, 'eta': 0.07313349418536874, 'subsample': 0.7481364041005474, 'colsample_bytree': 0.7144302711094385, 'gamma': 0.5515528704276921}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:52:30,988] Trial 18 finished with value: 0.5608333333333333 and parameters: {'max_depth': 24, 'eta': 0.22617848171784177, 'subsample': 0.643091997057727, 'colsample_bytree': 0.6558976273068391, 'gamma': 0.273230979010732}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:52:58,779] Trial 19 finished with value: 0.575 and parameters: {'max_depth': 91, 'eta': 0.13761458188996406, 'subsample': 0.6142904735125763, 'colsample_bytree': 0.7596885970440854, 'gamma': 0.4977199521557925}. Best is trial 3 with value: 0.585.
```

```
[I 2024-03-13 17:53:14,549] Trial 20 finished with value: 0.5583333333333333 and parameters: {'max_depth': 72, 'eta': 0.30871067263417784, 'subsample': 0.6600098983765909, 'colsample_bytree': 0.6294804335721165, 'gamma': 0.6097567820884121}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:53:45,707] Trial 21 finished with value: 0.5816666666666667 and parameters: {'max_depth': 96, 'eta': 0.10683139040303158, 'subsample': 0.6790847374920324, 'colsample_bytree': 0.731838690051801, 'gamma': 0.2483221311097987}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:54:28,620] Trial 22 finished with value: 0.58 and parameters: {'max_depth': 90, 'eta': 0.0743354509349324, 'subsample': 0.6934492256509515, 'colsample_bytree': 0.7437030911944721, 'gamma': 0.26857552664760487}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:55:07,530] Trial 23 finished with value: 0.5741666666666667 and parameters: {'max_depth': 92, 'eta': 0.12410506717419809, 'subsample': 0.7174966297879752, 'colsample_bytree': 0.7146975849216959, 'gamma': 0.3383770158079019}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:56:03,516] Trial 24 finished with value: 0.5766666666666667 and parameters: {'max_depth': 79, 'eta': 0.0518842948117746, 'subsample': 0.6760993863842863, 'colsample_bytree': 0.7063719598727107, 'gamma': 0.20625694512079015}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:56:26,523] Trial 25 finished with value: 0.575 and parameters: {'max_depth': 100, 'eta': 0.17259723620387704, 'subsample': 0.6332147944883807, 'colsample_bytree': 0.6675141865655313, 'gamma': 0.23289155998045133}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:57:11,590] Trial 26 finished with value: 0.5725 and parameters: {'max_depth': 59, 'eta': 0.07087890107916804, 'subsample': 0.7415773237284217, 'colsample_bytree': 0.7554908422723174, 'gamma': 0.2670961668610108}. Best is trial 3 with value: 0.585.
[I 2024-03-13 17:58:08,972] Trial 27 finished with value: 0.5966666666666667 and parameters: {'max_depth': 86, 'eta': 0.08914110787027095, 'subsample': 0.7090217089902818, 'colsample_bytree': 0.7798683657238631, 'gamma': 0.3202196584688024}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:00:14,414] Trial 28 finished with value: 0.5741666666666667 and parameters: {'max_depth': 86, 'eta': 0.03245793946096251, 'subsample': 0.7140273678733143, 'colsample_bytree': 0.787966930641372, 'gamma': 0.3855273417003752}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:00:29,938] Trial 29 finished with value: 0.5516666666666666 and parameters: {'max_depth': 60, 'eta': 0.3944206280473697, 'subsample': 0.7780563279942019, 'colsample_bytree': 0.8232092013020121, 'gamma': 0.32518212939218916}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:01:26,371] Trial 30 finished with value: 0.5791666666666667 and parameters: {'max_depth': 30, 'eta': 0.0807346769397424, 'subsample': 0.7649894238773554, 'colsample_bytree': 0.8237359307483485, 'gamma': 0.45456702922807984}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:01:52,412] Trial 31 finished with value: 0.5758333333333333 and parameters: {'max_depth': 96, 'eta': 0.11488827657989104, 'subsample': 0.6800519725984875, 'colsample_bytree': 0.7753141802452551, 'gamma': 0.2461424389366125}. Best is trial 27 with value: 0.5966666666666667.
```

```
[I 2024-03-13 18:06:44,253] Trial 32 finished with value: 0.5758333333333333 and parameters: {'max_depth': 85, 'eta': 0.010331722743462404, 'subsample': 0.6959338208267418, 'colsample_bytree': 0.745784323164655, 'gamma': 0.5897231626354085}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:07:23,463] Trial 33 finished with value: 0.5833333333333334 and parameters: {'max_depth': 92, 'eta': 0.08818633534637826, 'subsample': 0.6610818663817299, 'colsample_bytree': 0.7253748211066992, 'gamma': 0.2931525731111545}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:09:00,445] Trial 34 finished with value: 0.5816666666666667 and parameters: {'max_depth': 76, 'eta': 0.03684964666584306, 'subsample': 0.8439590404723045, 'colsample_bytree': 0.717803181379646, 'gamma': 0.29482666065926644}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:09:29,955] Trial 35 finished with value: 0.5575 and parameters: {'max_depth': 86, 'eta': 0.19348262971763927, 'subsample': 0.6573468403879056, 'colsample_bytree': 0.6976338312657588, 'gamma': 0.5145249841220667}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:10:03,239] Trial 36 finished with value: 0.5725 and parameters: {'max_depth': 81, 'eta': 0.13622912208346832, 'subsample': 0.7322233121632405, 'colsample_bytree': 0.79301010805919, 'gamma': 0.3324049232959213}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:10:36,632] Trial 37 finished with value: 0.5666666666666667 and parameters: {'max_depth': 93, 'eta': 0.09177306256689026, 'subsample': 0.7067022256085407, 'colsample_bytree': 0.6850255185552782, 'gamma': 0.3854326368875961}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:11:50,259] Trial 38 finished with value: 0.57 and parameters: {'max_depth': 72, 'eta': 0.06007977105353304, 'subsample': 0.6641760471558428, 'colsample_bytree': 0.771453520773445, 'gamma': 0.6668007004926394}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:12:18,304] Trial 39 finished with value: 0.5675 and parameters: {'max_depth': 85, 'eta': 0.15156299229122341, 'subsample': 0.6897976250797817, 'colsample_bytree': 0.6439131254996642, 'gamma': 0.5824765976964921}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:13:18,379] Trial 40 finished with value: 0.5741666666666667 and parameters: {'max_depth': 94, 'eta': 0.039058749907474676, 'subsample': 0.623711570941751, 'colsample_bytree': 0.7510119771191417, 'gamma': 0.29576329021997716}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:13:50,717] Trial 41 finished with value: 0.5716666666666667 and parameters: {'max_depth': 96, 'eta': 0.0907664981742017, 'subsample': 0.6837328525962776, 'colsample_bytree': 0.735142734687972, 'gamma': 0.2413879488112296}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:14:16,457] Trial 42 finished with value: 0.5758333333333333 and parameters: {'max_depth': 88, 'eta': 0.11202202083192216, 'subsample': 0.6404723972501751, 'colsample_bytree': 0.7234936253398007, 'gamma': 0.20333936137570224}. Best is trial 27 with value: 0.5966666666666667.
```

```
[I 2024-03-13 18:14:36,068] Trial 43 finished with value: 0.5675 and parameters: {'max_depth': 81, 'eta': 0.17487937112644442, 'subsample': 0.6696024510658934, 'colsample_bytree': 0.7050317523986152, 'gamma': 0.26087383780695983}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:15:05,477] Trial 44 finished with value: 0.575 and parameters: {'max_depth': 95, 'eta': 0.1219829290483965, 'subsample': 0.65168357620829, 'colsample_bytree': 0.7657536123021709, 'gamma': 0.3041079108056126}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:15:46,591] Trial 45 finished with value: 0.575 and parameters: {'max_depth': 10, 'eta': 0.058197175388446334, 'subsample': 0.705587269509559, 'colsample_bytree': 0.6752623822174838, 'gamma': 0.3494433211556069}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:17:34,387] Trial 46 finished with value: 0.5808333333333333 and parameters: {'max_depth': 47, 'eta': 0.02522676082925976, 'subsample': 0.6750043279936795, 'colsample_bytree': 0.7307551389914038, 'gamma': 0.2818447369632301}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:18:09,832] Trial 47 finished with value: 0.5875 and parameters: {'max_depth': 77, 'eta': 0.09298697847604198, 'subsample': 0.6921783170086437, 'colsample_bytree': 0.6945485168422483, 'gamma': 0.4143290865615039}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:18:53,825] Trial 48 finished with value: 0.5683333333333334 and parameters: {'max_depth': 75, 'eta': 0.08161708070938356, 'subsample': 0.7349772046626463, 'colsample_bytree': 0.6240121180427641, 'gamma': 0.5295665522581865}. Best is trial 27 with value: 0.5966666666666667.
[I 2024-03-13 18:19:06,436] Trial 49 finished with value: 0.53 and parameters: {'max_depth': 66, 'eta': 0.2930047397457317, 'subsample': 0.7176313418474469, 'colsample_bytree': 0.6463957459794509, 'gamma': 0.48948472762402184}. Best is trial 27 with value: 0.5966666666666667.
Best trial: {'max_depth': 86, 'eta': 0.08914110787027095, 'subsample': 0.7090217089902818, 'colsample_bytree': 0.7798683657238631, 'gamma': 0.3202196584688024}
Retraining
Test set accuracy: 0.64
Total execution time: 52.59 minutes

Process finished with exit code 0
```