DEPARTMENT OF
STATISTICS

Music Genre Classification with Machine Learning Models

MSc in Statistical Science, HT2021

Team Name: P231_P442_P648_P798

1 Introduction

This report studies into music genre classification with the provided 8000 audio tracks from Free Music Archive using 518 pre-computed music features extracted from various windows. For example, *mfcc max 04* in our data refers to the 4th maximum statistics from mel frequency cepstral coefficients (mfcc). We firstly conducted data cleaning and exploratory analysis on the training data of 6000 samples. Then, we implemented various machine learning models, tuned the hyperparameters and selected the best classification model according to cross-validation results on the training data, together with discussion about model efficiency. We used the final selected model to predict the genres of the 2000 tracks in the test set. With model interpretation, we analysed the variable importance. Lastly, we concluded by discussing the advantages and drawbacks of the model, and raised considerations for future work.

2 Exploratory Data Analysis (EDA)

The raw training data consists of 6000 observations with 518 features and there are 8 different music genre classes. Firstly, we performed simple data cleaning to make the data tidier and removed useless information. Upon investigation, we found and removed 13 repeated rows[1] in the training set. 24 columns were identified as containing very few (less than 1% of the number of samples) unique values. These 24 features correspond to the *chroma_cqt max* and *chroma_stft max* statistics, whose values are nearly all 1s. They were then removed from both the training set and test set as they are non-informative in the data. The full training data we later referred to would be the cleaned one, containing 5987 samples and 494 features. Then, the distribution of music genres was analysed. As Fig. 1 shows, counts of different classes are about the same, which indicates that the training data is approximately balanced.

Next, we performed Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA) on the training data **after standardisation**. They both transform the data to linear combinations of the original features. PCA aims to find the subspace explaining the most variance in the data while LDA tries to maximise the separation among classes. As shown in Fig. 2, the cumulative ratio of variance explained from PCA illustrates that 90% of the variance can be explained by 135 principal components, which is far less than the number of features in the data (494).

[1] Generally, duplicate rows may need further investigation but here since these duplicate observations are few, whose effect are negligible, we remove them to make the data tidyer.

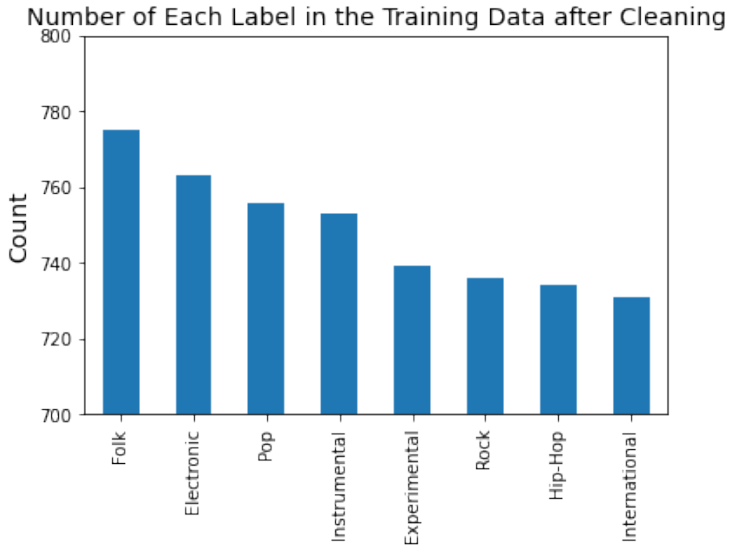

Fig. 1: Histogram of counts of different music genres in the training data.

With LDA, the data was reduced to 7-dimensional and the discriminant coordinates were computed, which indicate some information of how classes separate. We identified features with relatively large absolute values of the coefficients in the top discriminant coordinates as they may be important features. We inspected the corresponding class means of these features as the example shown in Table 1, where we could interpret some patterns in how the classes separate according to features' statistics. For example, *Electronic, Folk* and *Instrumental* tracks have relatively large negative mean *mfcc_mean_01* values, while *Experimental, Hip-Hop* and *Rock* samples have relatively large positive means for that feature.

Table 1: Class means of selected variables in the first discriminate coordinate. The means correspond to the 8 classes from left to right accordingly: *Electronic, Experimental, Folk, Hip-Hop, Instrumental, International, Pop* and *Rock*

Variable Name	*mfcc_mean_01*	*mfcc_median_05*	*spectral_bandwidth_01*	*zcr_mean_01*	...
Class Mean	-0.318, 0.404, -0.394, 0.441, -0.863, -0.023, 0.245, 0.704	-0.185, 0.079, 0.406, -0.190, 0.490, -0.420, -0.155, -0.052	0.646, -0.053, -0.394, 0.502, -0.893, 0.102, 0.067, 0.041	0.149, 0.204, -0.512, 0.257, 0.578, 0.239, 0.033, 0.243	...

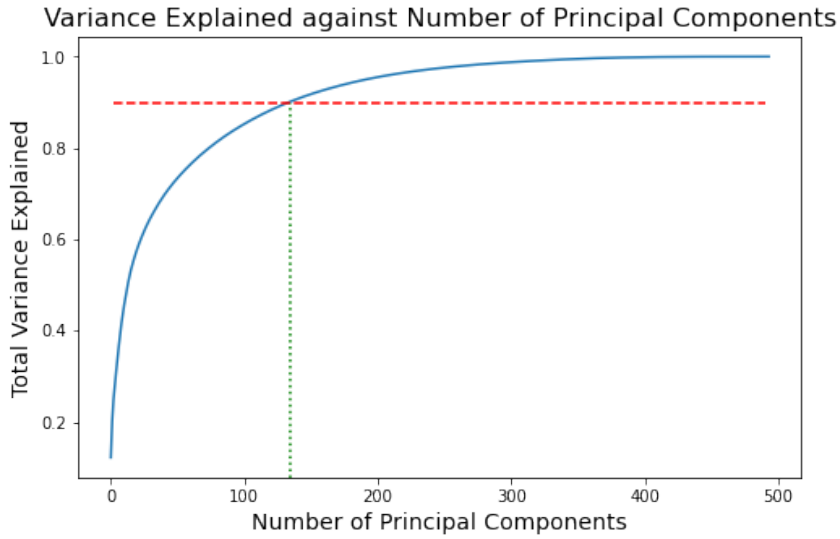

Fig. 2: Proportion of variance explained by the umber of PCs in PCA.

Both PCA and LDA can be used as dimensionality reduction methods to preprocess the data if lower dimensional space is desirable when training the models. PCA also decorrelates the feature space and it might help when multicollinearity becomes an issue. Thus, in model selection section, besides fitting model with training data in their original dimensions, we also discussed comparisons of models with PCA and LDA transformed data to see whether improvements could be made.

3 Model Selection

Firstly, we fitted a simple Naïve Bayes model as a baseline model. With 5-fold cross-validation (CV) applied to the dataset (5987 observations), the average classification accuracy for the baseline model is 41.66%. We then explored various machine learning algorithms to fit models including k-Nearest Neighbors (kNN), logistic regression, discriminant analysis, neural network (multi-layer perceptrons) and tree-based models. We tuned the essential hyperparameters with 5-fold CV to find the setting that gives the best generalisation performance for each modelling method. Table 2 summarises the best hyperparameters setting found for each method with CV results and the computation time required. We compared across different methods and selected the final model mainly according to the average CV accuracy, while we also considered the variance of the model's performance and discussed about the computational efficiency. Fig. 3 helps to directly visualise each method's performance after

hyperparameter tuning. It can be seen that tree-based XGBoost algorithm behaves well consistently and has the highest average classification accuracy, followed by neural network and logistic regression. Below we will describe how we tuned the hyperparameters for these top three methods. For all other methods, their hyperparameters were tuned with similar process.

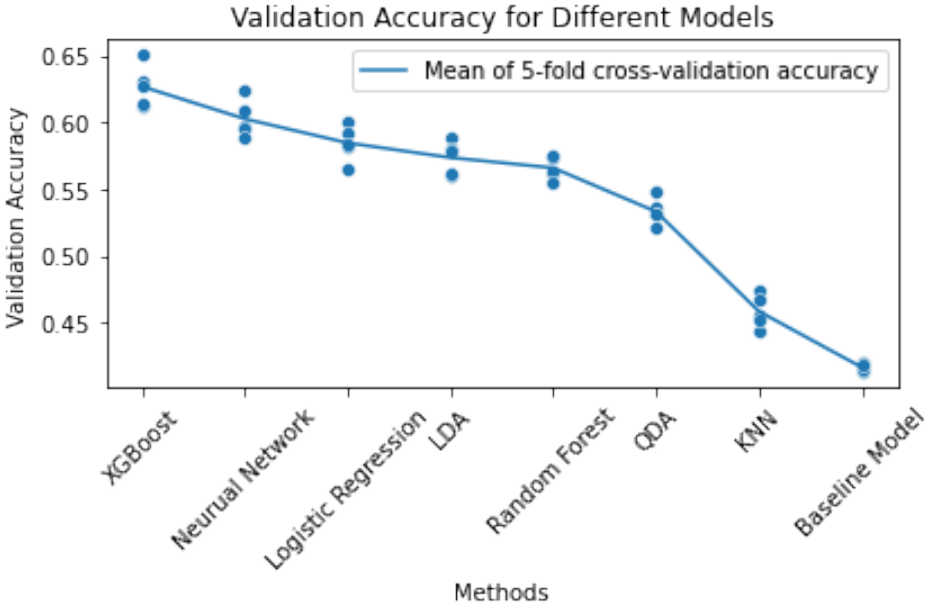

Fig. 3: Comparison of performance across different methods (each at their optimal hyperparameters setting after tuning) by 5-fold CV. x-axis arranges the methods in descending order of average CV accuracy.

3.1 Logistic Regression

Multiclass logistic regression was fitted and Newton-CG method, a variant of Newton method for high-dimensional problems, was applied as the optimiser. Hyperparameters C and tol were tuned. C is the inverse of L2 regularisation strength which controls the complexity of the model. tol is the stopping criteria for the optimisation algorithm which is essential to the convergence of the solution. After tuning these hyperparameters from ranges: $C = \{0.01, 0.1, 1\}$ and $tol = \{10^{-4}, 10^{-3}, 0.01, 0.1\}$, $C = 0.01$ and $tol = 10^{-4}$ were chosen as the optimal values with average cross-validation accuracy = 58.45%.

Table 2: Summary of statistics used for model comparison. Average and standard deviation were calculated based on 5 accuracy scores from 5-fold CV.[2]

Method	Best Hyperparameter Setting Selected by CV	Average Validation Accuracy	Standard Deviation Validation Accuracy	Time for One Fit (seconds)
Boosting Tree: XGBoost[3]	learning_rate: 0.1, n_estimators: 400, max_depth: 6, min_child_weight:1, gamma:0, colsample_bytree:1, subsample: 0.8, reg_alpha: 0.1	62.68%	1.30%	412 (An accuracy of 60% was obtained in 171 seconds by using early stopping)
Neural Network	1 layer of 512 neurons, α : 0.001, batch_size: 128, solver: 'adam', activation: 'ReLU'	60.73%	1.32%	49
Logistic Regression	C: 0.01, tol: 0.0001, max_iter: 300	58.45%	1.21%	<1
LDA	shrinkage: 0.2	57.36%	1.15%	<1
Random Forest	max_features: 50, n_estimators: 500	56.59%	0.79%	92
QDA	reg_param: 0.01	53.31%	0.89%	<1
kNN	k = 10	45.85%	1.08%	<1
Naïve Bayes (Baseline)	(Default)	41.66%	0.23%	<1

3.2 Neural Networks

We started tuning hyperparameters from the case of shallow neural network (with one hidden layer), and then increased the number of layers to 2 and 3 and tuned again. We fixed the activation function to be ReLU and optimiser as Adaptive Moment Estimation (Adam), which is an extension to stochastic gradient descent. Instead of using a constant learning rate for updates of all parameters throughout, Adam automatically adjusts each parameter's learning rate differently and adaptively. Using Adam alleviated us the need for tuning the learning rate hyperparameter although initial learning rate should still be

[2] Candidate values used for hyperparameter tuning:
LDA - shrinkage: {0.1, 0.2, ..., 1}; Random Forest - max_features: {1, 10,22, 50,100, 200,494}; QDA - reg_param: {0.0001, 0.001, 0.01, 1}; kNN - k: {1, 2, 5, 10, 20, 50, 100}.
Tuning ranges for logistic regression,neural network and XGBoost are discussed in details in following subsections.

[3] Hyperparameter names follow the ones used in Python package XGBoost with Scikit-Learn wrapper (Reference: https://xgboost.readthedocs.io/en/latest/python/python_api.html#module-xgboost.sklearn)

cautiously chosen because the upper bounds of learning rates are determined by the initial learning rate (init_learn_rate).

For each case (number of layers=1,2,3), we used CV to tune the number of neurons, the L2 regularisation parameter α, the batch size and init_learn_rate. The number of neurons and the regularisation term α determine the configuration of a neural network. They largely control the complexity of the neural networks. The batch size and init_learn_rate are gradient descent parameters which influence the optimisation. By using a grid search over a range of candidate hyperparameter settings, we obtained results as shown in Table 3. The optimal setting found is the one with single hidden layer and its hyperparameter values are shown in the table.

Table 3: Optimal Hyperparameter Values and Associated CV Accuracy [4]

Type of Network	Single Hidden Layer	Two Hidden Layers	Three Hidden Layers
Optimal Parameter Values	init_learn_rate: 10^{-3} Batch Size: 2^7 Number of Neurons: 2^9 α: 10^{-3}	init_learn_rate: 10^{-3} Batch Size: 2^7, Number of Neurons: 2^9, 2^7 α: 0.1	init_learn_rate: 10^{-3} Batch Size: 2^7 Number of Neurons: 2^8, 2^8, 2^8 α: 1
Average CV Accuracy	60.73%	60.34%	60.41%

3.3 Tree-Based Methods

The idea for tree-based method is that the data space could be split into disjoint regions hierarchically by feature values, where a tree structure forms and appropriate class labels are assigned accordingly to each of the leaf of the tree. Decision tree is a basic tree-based approach. However, even after tuning the tree complexity, the best accuracy we achieved was just 39.30%, not comparable with the baseline model. Apparently, a single tree is not an appropriate model here. Moving to random forest, which constructs many decision trees with bootstrapped samples and outputs the class by majority vote to achieve low variance, improved the average accuracy to 56.59%. However, this is still not ideal.

Then, we explored boosting and finally found the Extreme Gradient Boosting

[4] Candidate values used for hyperparameter tuning: init_learn_rate:$\{10^{-5}, 10^{-4}, ..., 1\}$, batch size: $\{2^5, 2^6, ..., 2^9\}$, number of neurons for each layer: $\{2^5, 2^6, ..., 2^9\}$, α: $\{10^{-4}, 10^{-3}, ..., 1\}$

method, commonly known as **XGBoost**, has the best performance among all tree-based models we attempted. Boosting constructs a series of weak classifiers (eg. small decision trees) sequentially and combines them adaptively to reduce bias and create a strong classifier finally. Compared to the basic Adaboost, XGBoost improved both the accuracy and speed significantly. XGBoost is an extension of gradient boosting, a type of boosting that aims to optimise the loss function from previous step using gradient descent algorithm in each iteration. The objective loss function to be minimised for iteration t is shown in (1)[5], for t=2, 3,...,*n_estimators*. XGBoost supports multi-core processing so it is much more efficient, and it offers more customised ways to build classifiers with a good control of overfitting given many useful hyperparameters.

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \eta^{(t)}(x_i)) + \Omega(f^{(t)}) \tag{1}$$

where

$$l(y_i, \eta^{(t)}(x_i)) = -\sum_{c=1}^{K} \mathbf{1}_{y_i=c} log(\eta_c^{(t)}(x_i))$$

$$\eta_c^{(t)}(x_i) = \frac{exp(\hat{y_{ic}}^{(t-1)} + f_c^{(t)}(x_i))}{\sum_{c=1}^{K} exp(\hat{y_{ic}}^{(t-1)} + f_c^{(t)}(x_i))}$$

with K being the number of classes and c is the lable encoder, $\hat{y_i}^{(t)} = (\hat{y_{i1}}^{(t)}, ..., \hat{y_{iK}}^{(t)})$ represents some raw boosting scores calculated and attached to different classes for observation i, and η applies softmax normalisation to convert them to probabilities respectively. $f^{(t)}$ is the tree constructed at iteration t with $f^{(t)}(x_i)$ giving the score we should add to minimise the error from previous tree. $\Omega(f^{(t)})$ is the regularisation term.

We tuned the important hyperparameters with cross-validation to select the best setting. Firstly, we applied a learning rate of 0.1, which typically works well. Learning rate largely determines how many number of estimators (trees) should be used. By CV, we found the optimal number of trees to use given learning rate=0.1 is around 400. Next, fixing the learning rate as 0.1 and *n_estimator* as 400, we grid searched to tune the tree-specific parameters: *max_depth*, *min_child_weight*, *gamma*, *subsample*, *colsample_bytree*. Table 4 summarises the meaning and functional effect of each hyperparameter as well as the values we selected. All these hyperparameters have effects on the complexity of the trees and can help avoid overfitting together. For example, we should not have large value of tree depth as it creates more complex trees tending to

[5] Adapted from original paper of XGBoost.Reference:https://arxiv.org/pdf/1603.02754.pdf

overfit. And if a new split cannot meet our desirable improvements, we would decide not to create that split to avoid overfitting, as what the *min_child_weight* controls.

In addition, we could add regularisation term to construct a more regularised tree. L1 regularisation is often used in XGBoost with high dimensional data and applying it could make the algorithm run faster. Therefore, after tuning the tree-specific hyperparameters and fixing their values as the best setting in Table 4, we tuned the L1 regularisation hyperparameter *reg_alpha* from the set $\{10^{-4}, 10^{-2}, 0.1, 1, 10\}$. The best *reg_alpha* found was 0.1. From Fig 4, we can see it is not good to put *reg_alpha* at a larger value like 1 and 10. Because when we tuned the tree-specific hyperparameters, they already put some 'pseudo' regularisation on the model to avoid overfitting, so setting *reg_alpha* at a large value will likely overdone the job and cause underfitting. By now, we have finished tuning the essential hyperparameters of XGBoost. The optimal hyperparameter values are:

$$\begin{aligned} &learning\_rate = 0.1, num\_of\_trees = 400, max\_depth = 6, subsample = 0.8 \\ &min\_child\_weight = 1, gamma = 0, colsample\_bytree = 1, reg\_alpha = 0.1 \end{aligned}$$

The average cross-validation accuracy of this selected model is 62.68% with a standard deviation of 1%. It is also the final model we used after comparing all the methods from Table 2.

In terms of running time, the best neural network is faster than XGBoost, but the latter is still acceptable and in fact its computation time highly depends on how many trees we use as we mentioned. We can use smaller number of trees like around 200 and the accuracy will not be affected too much, as from the example loss curve in Fig 5. With around 200 trees, we already reached a validation accuracy above 60.00% and that reduced the time to 171 seconds. Early stopping can also be applied in XGBoost to help reduce the computational cost and avoid overfitting at the meanwhile.

3.4 Model Trained with Transformed Data by PCA & LDA

Additionally, we attempted to build models on data preprocessed by PCA and LDA respectively, whose dimensions were significantly reduced as mentioned in EDA. Similarly, we tuned hyperparameters with CV on the transformed data. We compared the models' performance before and after dimensionality reduction for each classifier shown in Table 5. Neither PCA nor LDA transformation helped to find a better model. LDA transformation helped the kNN

Table 4: Table for tree-specific hyperparameters tuning. The candidate values were determined according to their ranges of typical values.This is a computationally heavy grid search.
Reference: *https://xgboost.readthedocs.io/en/latest/parameter.html#parameters-for-tree-booster.*

Hyperparameter	Explaination	Effect	Candidate Values	Best Setting Obtained from 5-Fold CV
max_depth	Maximum depth of every tree	Large values lead to more complex trees; reduce bias, increase variance.	3, 6, 9	max_depth: 6, min_child_weight: 1, gamma: 0, subsample: 0.8, colsample_bytree: 1
min_child_weight	Minimum sum of instances' Hessian weight required in child to make a split	Smaller values lead to more complex trees.	1, 3, 6	
gamma	Minimum loss reduction required to make a split	Smaller values lead to more complex trees.	0, 0.1, 0.3, 0.5	
subsample	Proportion of randomly subsampled observations to be used to build trees.	Larger values lead to more complex trees.	0.5, 0.8, 1	
colsample_bytree	Proportion of randomly subsampled columns to be used for to build trees.	Large values lead to more complex trees.	0.5, 0.8, 1	

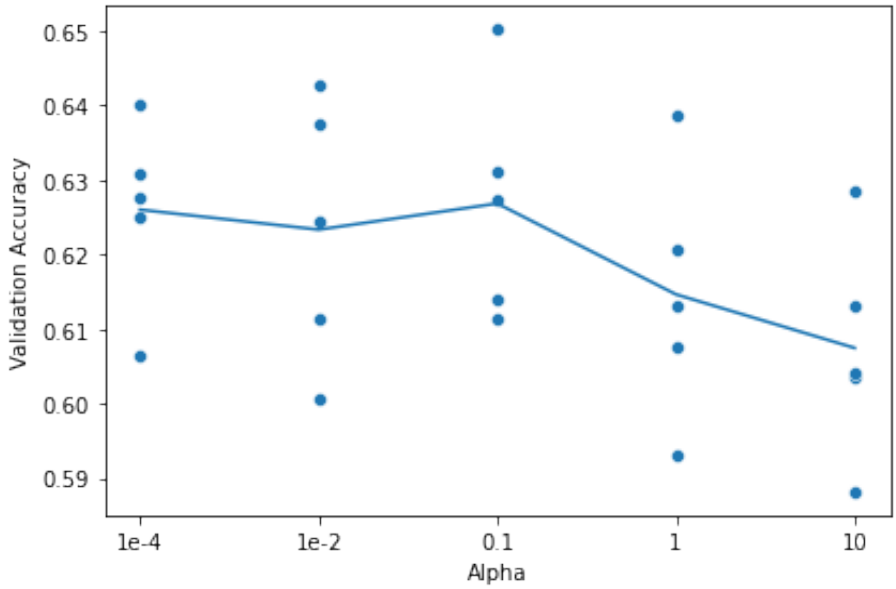

Fig. 4: Validation Accuracy for Different *reg_alpha* Values

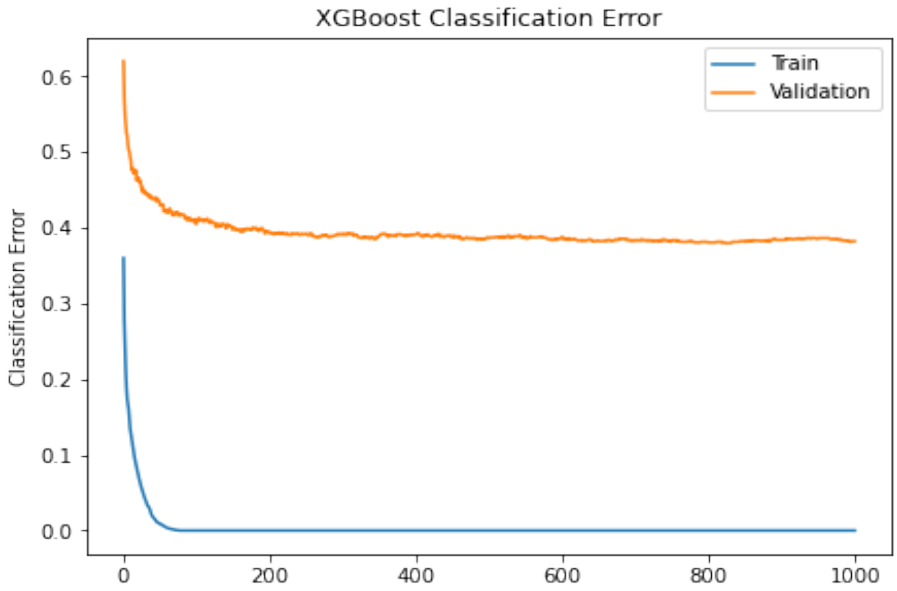

Fig. 5: Loss Curve on One Example Traning and Test Set.

improve performance significantly as kNN is not suitable for high dimensional data. But for models like XGBoost and neural network which could already handle high dimensional data well, their performance degraded after PCA or LDA transformation, likely due to the loss of information when reducing the dimensions. Comparing them all together, the best model to use is still the XGBoost model defined in section 3.3, and it is the final model we selected.

Table 5: Comparisons of Model Performances with Transformed Data

Method	Average Cross Validation Accuracy		
	Trained on Original (Cleaned) Data	Transformed by PCA	Transformed by LDA
XGBoost	62.68%	57.11%	53.02%
Neural Network	60.73%	57.10%	55.59%
Logistic Regression	58.45%	55.93%	56.38%
LDA	57.35%	54.97%	52.58%
Random Forest	56.59%	51.17%	54.02%
QDA	53.31%	53.23%	52.68%
kNN	45.84%	42.57%	55.84%

We retrained our final model on the full training data (5987 observations) and

used it to predict the labels on test set. Our model obtained a test accuracy of 61.60% on the public leaderboard from Kaggle competition.

4 Model Interpretation

We conducted feature importance analysis on the final XGBoost model we selected. There is a built-in feature importance measure in XGBoost with scikit-learn in Python based on the average gain in performance across all splits where the feature is used. Figure 6 shows the variable importance plot to identify the top 20 most important variables.

Additionally, we applied a model agnostic method to measure variable importance as an additional reference via Shapley values (SHAP)[6]. SHAP measures the features' importance by comparing predictions with current feature values to the one that changes the corresponding feature values to some non-informative baseline values. Figure 7 shows the top 20 most important features from SHAP. From the figures, both methods have identified the following features as important ones:

mfcc max 04, *mfcc mean 20*, *mfcc median 03*, *mfcc median 20*, *mfcc std 02*, *mfcc std 20*, *rmse std 01*, *spectral_centroid median 01*, *spectral_contrast median 02*, *spectral_contrast median 04*

From Figure 7, we could also interpret the importance of features to different classes. For example, feature *mfcc max 04* contributed the most to the prediction for *Hip-Hop* and *Instrumental*, while feature *rmse std 01* contributed the most to *Hip-Hop*.

[6] Package implementing SHAP in Python with detailed description of the method: https://github.com/slundberg/shap.

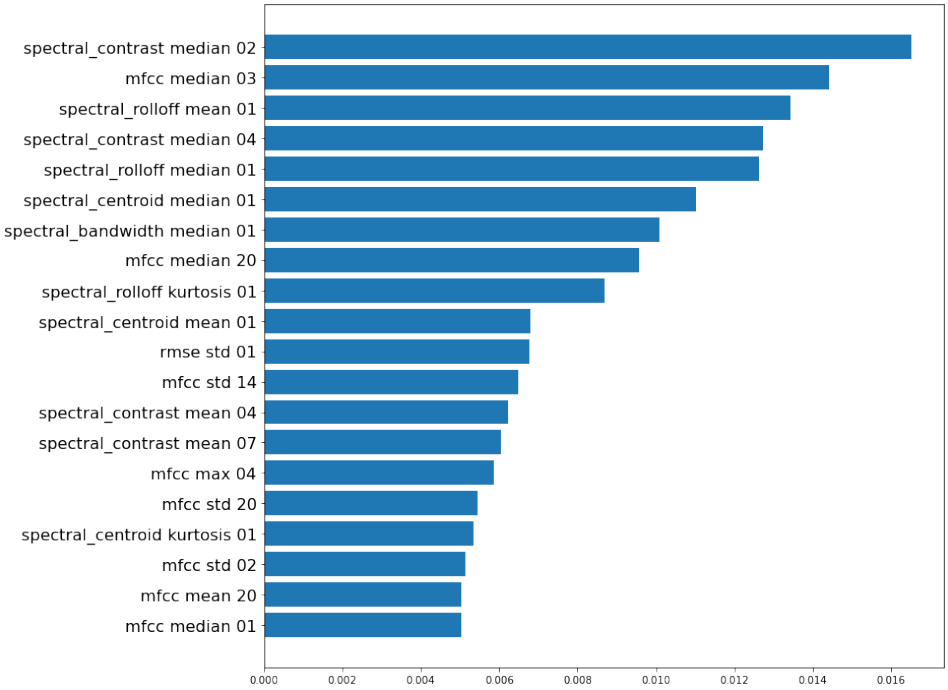

Fig. 6: Top 20 most important features according to built-in feature importance function.

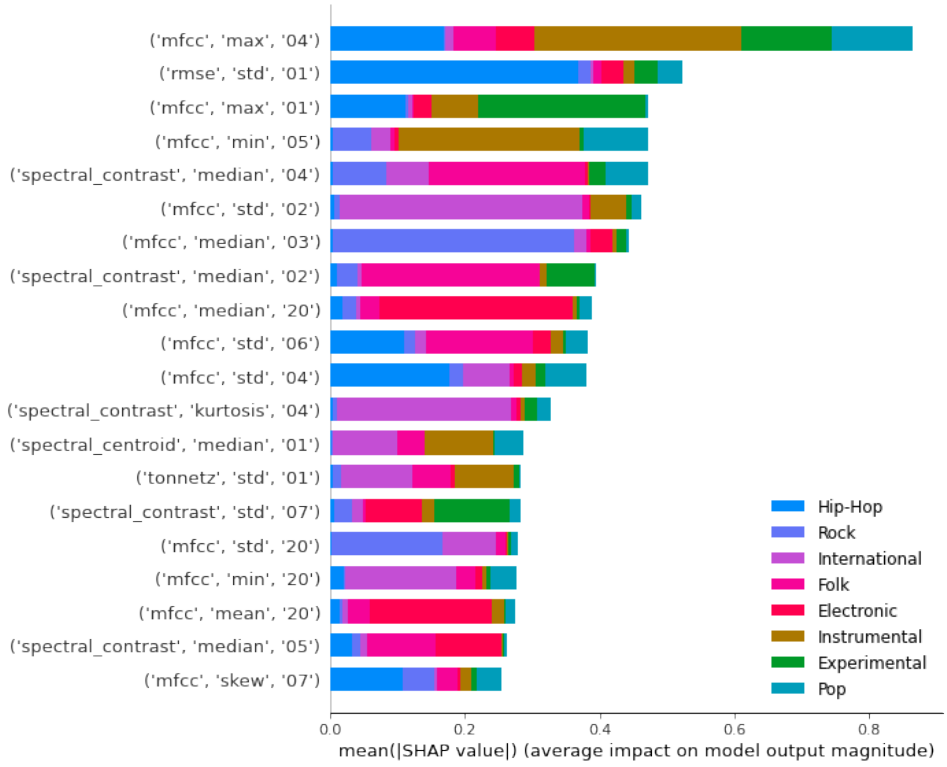

Fig. 7: Top 20 most important features according to SHAP values.Different colors on each bar represent the importance of each feature to the prediction of different music genres.

5 Conclusion and Future Work

After tuning hyperparameters and comparing various classifiers, we finally found that the XGboost model had the best performance. From the 5-fold CV, its best validation accuracy could reach 65.10% and on average the validation accuracy was 62.68%, which outperformed other models. However, boosting is generally computationally heavy. Though XGBoost is already designed to improve efficiency, it requires more computation time compared to other classifiers we summarised in Table 2. That may be a trade-off.But there are known more modern extensions of XGBoost like LightGBM and CatBoost that are designed to specifically improve efficiency. Though they are not widely used yet, we can explore such methods in the future work to improve efficiency.

In addition, there may also be space for improvement for neural network. Ensemble neural network can be explored and neural networks with customised connections, e.g. only connecting statistics extracted from the same music feature in the shallow layer and builds more layers after that, can be tried if we have gather more domain knowledge of features of music.

Appendix: Python Code

```
# Standard scientific Python imports
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

#load data
y_train = pd.read_csv('y_train.csv', index_col = 0, squeeze=True)
X_train = pd.read_csv('X_train.csv', index_col = 0, header=[0, 1, 2])
X_test = pd.read_csv( 'X_test.csv', index_col = 0, header=[0, 1, 2])

###Data Exploration###

#1. Data Cleaning and Preprocessing

#check for duplicate rows
sum(X_train.duplicated()) #13 duplicate rows
X_train_remove=X_train[X_train.duplicated()==False]#we remove them

##check for columns contains single/few values
X_train.loc[:,X_train.nunique()<=1].columns #all have at least 2 values
X_train.nunique().sort_values(ascending=True).head(n=30)
# get number of unique values for each column
counts = X_train.nunique()
# remove columns with very few values : 24 columns--cqt_max, stft_max
#delete if number of unique values less than 1% of the data
to_del = [i for i,v in enumerate(counts) if \
          (float(v)/X_train.shape[0]*100) < 1]
X_train_clean=X_train_remove.drop(X_train.columns[to_del],axis=1)
#dimension=5987(rows)*494(columns)
y_train_clean=y_train[X_train.duplicated()==False] #5987*1
X_test_clean=X_test.drop(X_train.columns[to_del],axis=1) #2000*494
#Final data we use: X_train_clean,y_train_clean,X_test_clean

# Label Count
y_train_clean.value_counts().plot(kind = 'bar')
plt.ylabel('Count', fontsize=14)
plt.title('Number of Each Label in the Training Data \
 after Cleaning', fontsize=14)
```

```
#PCA analysis:
scalar = StandardScaler() # Standardise before transformation
X_scaled = scalar.fit_transform(X_train_clean)
pca= PCA()
pca.fit(X_scaled)
var_explained = pca.explained_variance_ratio_

# Plot of Variance Explained
plt.figure(figsize=(10,6))
plt.plot(np.cumsum(var_explained)) # Cumulative Ratio
plt.axhline(0.9, xmin=0.05, xmax=0.95, c='r', linestyle='--')
plt.axvline(135, ymin=0, ymax=0.85, c='g',linestyle=':')
plt.xlabel('Number of Principal Components', fontsize=14)
plt.ylabel('Total Variance Explained', fontsize=14)
plt.title('Variance Explained against Number of \
           Principal Components', fontsize=16)

#LDA analysis:
lda = LinearDiscriminantAnalysis()
X_lda = lda.fit_transform(X_scaled, y_train_clean)

w = lda.coef_
l1 = np.where(abs(w[0])>2)[0] # Which features have large coefficients

means = lda.means_ # Class Means

print('The features having high weights in the 1st \
       discriminant coordinates are',l1,'.')
for i in l1:
    meani = means[:,i]
    print('The group means at the', i,'th feature are', meani)


### Model Selection ###

#5-fold cross validation set up
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
np.random.seed(0)
nfold = StratifiedKFold(n_splits=5)
#control the train/validation sets to be the same
```

```
#to make comparisons across models fairer.

#1. Baseline Model: Naive Bayes
# Naive Bayes (Gaussian)
from sklearn.naive_bayes import GaussianNB
clf_nb = GaussianNB()
nb=cross_val_score(clf_nb, X_train_clean,y_train_clean, cv=nfold)

#2. Logistic Regression:
from sklearn.linear_model import LogisticRegression
# hyperparameter tuning
Cs = [0.01, 0.1, 1]
tols = [0.0001, 0.001, 0.01, 0.1]
acc_cv_lr = np.zeros((len(Cs), len(tols)))
scalar = StandardScaler()
for (i, C) in enumerate(Cs):
    for (j, tol) in enumerate(tols):
        print('C=', C, 'tol=', tol)
        clf_lr = LogisticRegression(C=C, tol=tol, max_iter=300
        \, solver = 'newton-cg', verbose=True)
        #scale data before fitting model each time
        pipeline = Pipeline([('transformer', scalar), ('estimator', clf_lr)])
        acc_cv_lr[i][j] = (cross_val_score(pipeline, X_train_clean, \
        y_train_clean, cv = nfold, n_jobs=-1)).mean()

def find_index(a): # find location of maximum in a multi-dim array
    return np.unravel_index(np.argmax(a, axis=None), a.shape)

print('highest average cv accuracy:', acc_cv_lr.max())
ib, jb = find_index(acc_cv_lr)
print('best C:', Cs[ib])
print('best tol:', tols[jb])

# an example of how we calculated computational time and
# cv accuracy of best model for each method
# similar code for other methods
scalar = StandardScaler()
t1 = time.time()
clf_lr = LogisticRegression(C=0.01, max_iter=300, solver = 'newton-cg')
pipeline = Pipeline([('transformer', scalar), ('estimator', clf_lr)])
result_lr = cross_val_score(pipeline, X_train_clean,y_train_clean
```

```
\, cv = nfold, n_jobs=-1)
t2 = time.time()
print('average cv accuracy:', result_lr.mean())
print('cv accuracy sd:', result_lr.std())
print('cv time:', (t2 - t1) / 5)

#3. Neural Network:
import itertools
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import Pipeline

scaler = StandardScaler()
clf_nn = MLPClassifier(activation='relu', max_iter=300,
                       verbose=True,solver='adam')
pipeline = Pipeline(steps=[('sc', scaler), ('clf', clf_nn)])
param_grid = {
    'clf__hidden_layer_sizes': [x for x in
    \itertools.product((32, 64, 128, 256, 512),repeat=1)],
    'clf__alpha': np.array([0.0001, 0.001, 0.01, 0.1, 1]),
    'clf__batch_size': np.array([32, 64, 128, 256, 512]),
    'clf__learning_rate_init': np.array([0.00001, 0.0001, 0.001, 0.01, 0.1, 1])
}
search = GridSearchCV(pipeline, param_grid, n_jobs=-1)
search.fit(X_train_clean, y_train_clean)
print("Best parameter (CV score=\%0.3f):" \% search.best_score_)
print(search.best_params_)

## similar code for 2/3 layers with
'clf__hidden_layer_sizes': [x for x in
                            itertools.product((32, 64, 128, 256, 512),repeat=1)]
# replaced by
'clf__hidden_layer_sizes': [x for x in
                            itertools.product((32, 64, 128, 256, 512),repeat=2)]
# and
'clf__hidden_layer_sizes': [x for x in
                            itertools.product((32, 64, 128, 256, 512),repeat=3)]
# respectively

#4. XGBoost:
```

```
#1 st step: find the best number of trees given learning rate=0.1
import xgboost as xgb
#the xgboost package has a nice built in cross-validation method to
#easily get the optimum number of trees given a learning rate
#we ultise that good function and don't have to do grid-search here
def find_optimum_n_est(alg, X, y, useTrainCV=True, cvFolds=nfold,
    if useTrainCV:
        xgbParams = alg.get_xgb_params()
        xgTrain = xgb.DMatrix(X, label=y)
        cvresult = xgb.cv(xgbParams,
                    xgTrain,
                    num_boost_round=alg.get_params()['n_estimators'],
                    nfold=cvFolds,
                    stratified=True,
                    metrics={'merror'},
                    early_stopping_rounds=early_stopping_rounds,
                    seed=0,
                    callbacks=[xgb.callback.print_evaluation(show_stdv=False),
                    xgb.callback.early_stop(50)])

        print (cvresult)
        alg.set_params(n_estimators=cvresult.shape[0])

    # Fit the algorithm
    alg.fit(X, y, eval_metric='merror')

    # Predict
    dtrainPredictions = alg.predict(X)
    dtrainPredProb = alg.predict_proba(X)
#the above function defined will output the optimum n_estimators
#given a learning rate

#fix learning rate at 0.1 and use the find_optimum_n_est
#to find the optimum number of trees(n_estimators)
default_xgboost=XGBClassifier(
        eval_metric= 'merror',
        num_class=8,
        n_estimators=1000,
        nthread = 4,
        objective='multi:softmax',
        learning_rate =0.1,
```

```
        seed=0)
find_optimum_n_est(default_xgboost, X_train_clean, y_train,
                   early_stopping_rounds=50)
#optimum n_estimator outputted is 400.

#2nd step: Grid search with cross-validation to tune
#the tree-specific parameters together
param_tree = {'gamma':[0,0.1,0.3,0.5],
              'colsample_bytree':[0.5,0.8,1],
              'subsample':[0.5,0.8,1],
              'max_depth':range(3,10,3),
              'min_child_weight':range(1,6,2),
 }

from xgboost import XGBClassifier #scikit-learn wrapper of xgb
#easier to implement
from sklearn.model_selection import GridSearchCV

clf_xgboost_tune=XGBClassifier(eval_metric= 'merror',
                               num_class=8,
                               n_estimators=400,
                               nthread = 4,
                               objective='multi:softmax',
                               learning_rate =0.1,
                               seed=0)

gsearch_tree = GridSearchCV(estimator = clf_xgboost_tune,
                            param_grid=param_tree,scoring='accuracy',
                            n_jobs=4,verbose=True, cv=nfold)

gsearch_tree.fit(X_train_clean, y_train_clean)
gsearch_tree.cv_results_ #output detailed CV results with time
gsearch_tree.best_params_, gsearch_tree.best_score_
#output best parameter setting and corresponding mean CV accuracy

#Grid-search with CV is time consuming.The current one takes 1355.94 minutes
#If more computation resource is possible, we could tune the parameters
#with more values


#3rd step: Add in L1 regularizer
```

```
#set other parameters to the best setting found from gsearch_tree
#and tune the regularization parameter reg_alpha
#this step takes 94.5min
param_regularizer = {'reg_alpha':[1e-4, 1e-2, 0.1, 1, 10]}
clf_xgboost_alpha=XGBClassifier(
        eval_metric= 'merror',
        num_class=8,
        n_estimators=500,
        nthread = 4,
        objective='multi:softmax',
        learning_rate =0.1,gamma=0,
        colsample_bytree=1,subsample=0.8,
        seed=0,max_depth=6,min_child_weight=1)

gsearch_alpha = GridSearchCV(estimator = clf_xgboost_alpha,
                             param_grid = param_regularizer,
scoring='accuracy',n_jobs=4,verbose=True, cv=nfold)
gsearch_alpha.fit(X_train_clean, y_train_clean)
gsearch_alpha.cv_results_
gsearch_alpha.best_params_, gsearch_alpha.best_score_

####Variable Importance####
##Code for finding the most importance features among different models
#This is top 20 most important features for XGBoost found by
#built-in function in XGBoost
xgb1 = ['spectral_contrast median 02', 'mfcc median 03',
'spectral_rolloff mean 01', 'spectral_contrast median 04',
'spectral_rolloff median 01', 'spectral_centroid median 01',
'spectral_bandwidth median 01', 'mfcc median 20',
'spectral_rolloff kurtosis 01', 'spectral_centroid mean 01',
'rmse std 01', 'mfcc std 14', 'spectral_contrast mean 04',
'spectral_contrast mean 07', 'mfcc max 04', 'mfcc std 20',
'spectral_centroid kurtosis 01', 'mfcc std 02', 'mfcc mean 20',
'mfcc median 01']

#This is top 20 most importance features for XGBoost
#found by shap function
xgb2 = ['mfcc max 04', 'rmse std 01', 'mfcc max 01',
'mfcc min 05','spectral_contrast median 04', 'mfcc std 02',
'mfcc median 03','spectral_contrast median 02', 'mfcc median 20',
'mfcc std 06','mfcc std 04', 'spectral_contrast kurtosis 04',
```

```
'spectral_centroid median 01', 'tonnetz std 01',
'spectral_contrast std 07', 'mfcc std 20', 'mfcc min 20',
'mfcc mean 20', 'spectral_contrast median 05', 'mfcc skew 07']

#Find the common features between the two methods
set(xgb1) & set(xgb2)

####Code for Other Method###
## Take LDA as an exmaple. Similar code for other classifiers.
## hyperparameter tuning
# solver as 'svd' (defualt)
scalar = StandardScaler()
clf_lda = LinearDiscriminantAnalysis()
pipeline = Pipeline([('transformer', scalar), ('estimator', clf_lda)])
print('CV accuracy:', (cross_val_score(pipeline, X_train_clean,
       y_train_clean, cv = nfold, n_jobs=-1)).mean())

# solver as 'lsqr'/'eigen'
# tune on shrinkage (between 0 and 1)
shrs = np.arange(0, 1, 0.1)
solvers = ['lsqr', 'eigen']
acc_cv_lda = np.zeros((len(solvers), len(shrs)))
scalar = StandardScaler()
for (i, sol) in enumerate(solvers):
    for (j, shr) in enumerate(shrs):
        print('solver is', sol, 'shrinkage=', shr)
        clf_lda = LinearDiscriminantAnalysis(solver = sol, shrinkage = shr)
        pipeline = Pipeline([('transformer', scalar), ('estimator', clf_lda)])
        acc_cv_lda[i][j] = (cross_val_score(pipeline, X_train_clean,
                            y_train_clean, cv = nfold, n_jobs=-1)).mean()

print('highest average cv accuracy:', acc_cv_lda.max())
ib, jb = find_index(acc_cv_lda)
print('best solver:', solvers[ib])
print('best shrinakge:', shrs[jb])

####Code for model fit with PCA transformed data####
# take logistic regression as an example
# same method is applied on other classifiers
Cs = [0.01, 0.1, 1]
tols = [0.0001, 0.001, 0.01, 0.1]
```

```
acc_pcacv_lr = np.zeros((len(Cs), len(tols)))
scalar = StandardScaler()
pca = PCA(n_components = 0.9, svd_solver = 'full')
scalar2 = StandardScaler()
for (i, C) in enumerate(Cs):
    for (j, tol) in enumerate(tols):
        print('C=', C, 'tol=', tol)
        clf_lr = LogisticRegression(C=C, tol=tol, max_iter=300,
                                    solver = 'newton-cg', verbose=True)
        pipeline = Pipeline([('sc', scalar), ('pca', pca), ('sc2', scalar2),
                             ('estimator', clf_lr)])
        acc_pcacv_lr[i][j] = (cross_val_score(pipeline, X_train,y_train,
                              cv = nfold, n_jobs=-1)).mean()

print('highest average cv accuracy:', acc_pcacv_lr.max())
ib, jb = find_index(acc_pcacv_lr)
print('best C:', Cs[ib])
print('best tol:', tols[jb])

#### Code for model fit with LDA transformed data is similar to that for PCA
#### with two lines changed:
# replace
pca = PCA(n_components = 0.9, svd_solver = 'full')
# by
lda = LinearDiscriminantAnalysis()
# and replace
pipeline = Pipeline([('sc', scalar), ('pca', pca), ('sc2', scalar2)
\, ('estimator', clf_lr)])
# by
pipeline = Pipeline([('sc', scalar), ('lda', lda), ('sc2', scalar2)
\, ('estimator', clf_lr)])
```