Predicting Genre for Audio Tracks from the Free Music Archive

MSc in Statistical Science

University of Oxford

P919-P268-P632-P299

March 2021

Kaggle Team Name: The Wise Beavers

1 Introduction

In this report, we try various methods to learn models that predict the musical genre of audio tracks using various engineered features that capture their musical properties. We compare a number of modeling approaches: generative classifiers, linear classifiers, KNN, ensembles, and neural networks. The goal is to obtain good classification accuracy on a held-out set of 2000 audio tracks available on Kaggle while using a set of 6000 audio tracks for training and parameter tuning.

2 The Data

The data contain 8000 audio tracks from 8 musical categories: Electronic, Rock, Instrumental, Hip-Hop, Pop, Experimental, Folk, and International. The true classifications of 2000 of these audio tracks have been withheld, but 1000 are available via submission to Kaggle. We use these to make our final model comparisons. In the remaining 6000-observation training set, the 8 classes are approximately balanced, with around 750 observations per class. The data include 11 overall features, and for each, 7 summary statistics (kurtosis, mean, median, max, min, std, and skew) have been computed over between one and twenty ordered windows over the audio clips. In total, there are 518 features and no missing data. While our focus is not on feature interpretation, we make a few relevant observations:

First, we note that these features occur on different scales. Figure 1 displays the standard deviations of all 518 features and then only of those with standard deviation under 100. The magnitude differences observed here reflect in part differences between the 11 overall features. For example, the largest maximum value observed across all 6000 observations and all windows for the `chroma_cens` feature is 0.979 while for `spectral_bandwidth`, it is 5075.348. This suggests that it may be advantageous to scale the data to aid faster convergence in optimization algorithms such as gradient descent.

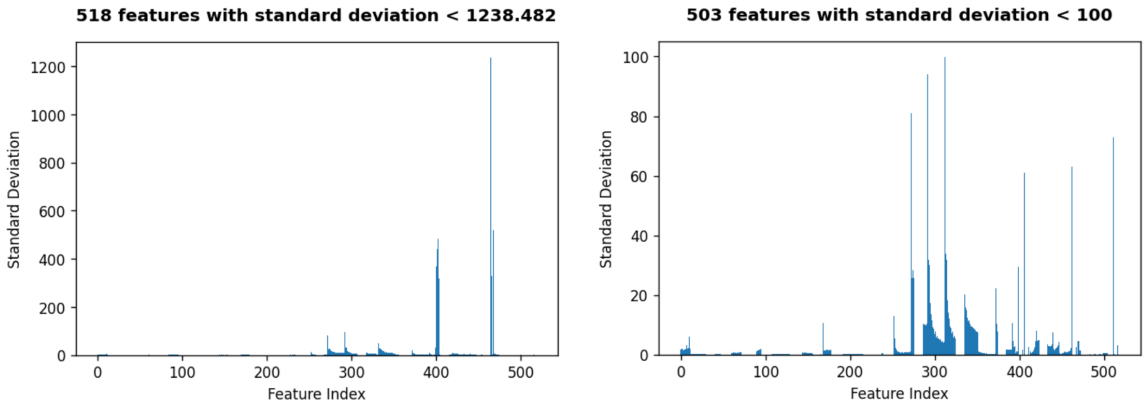

Figure 1: **Left** Standard deviations for all 518 features. **Right** Standard deviations for 503 features with standard deviation under 100 (those over 100 set to 0).

Second, within each of the 11 audio features, different summary statistics from the same or adjacent windows are moderately correlated, with `mean` and `median` particularly positively correlated. This is reflected by the lighter or darker colors on the diagonals in Figure 2, which shows these correlations for `chroma_cens`. Similar patterns occur for other features. This suggests that there is redundancy among the summary statistics and that we could benefit from some dimension reduction or feature selection.

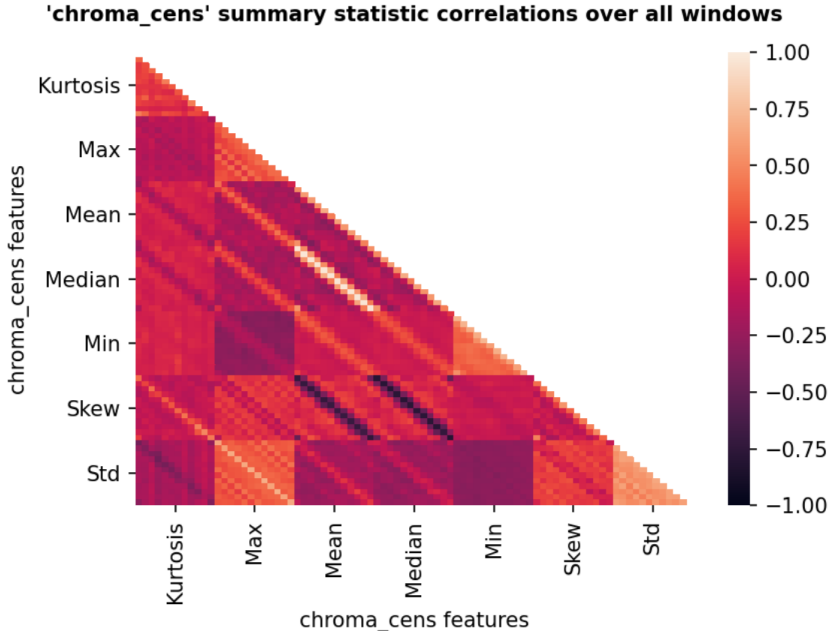

Figure 2: Heatmap of correlations for the `chroma_cens` feature. The plot includes correlations for all combinations across summary statistics and windows. Diagonal lines indicate particularly positive or negative correlations between features occurring in the same or adjacent windows.

Finally, Figure 3 shows the projections of the data onto the first two principal components and onto the first and second discriminant coordinates from Linear Discriminant Analysis. The principal component projection shows poor separation, indicating that the directions of greatest variability in the data do not separate the classes. On the other hand, LDA projections, which specifically maximize the distance between class means, do display moderate separation.

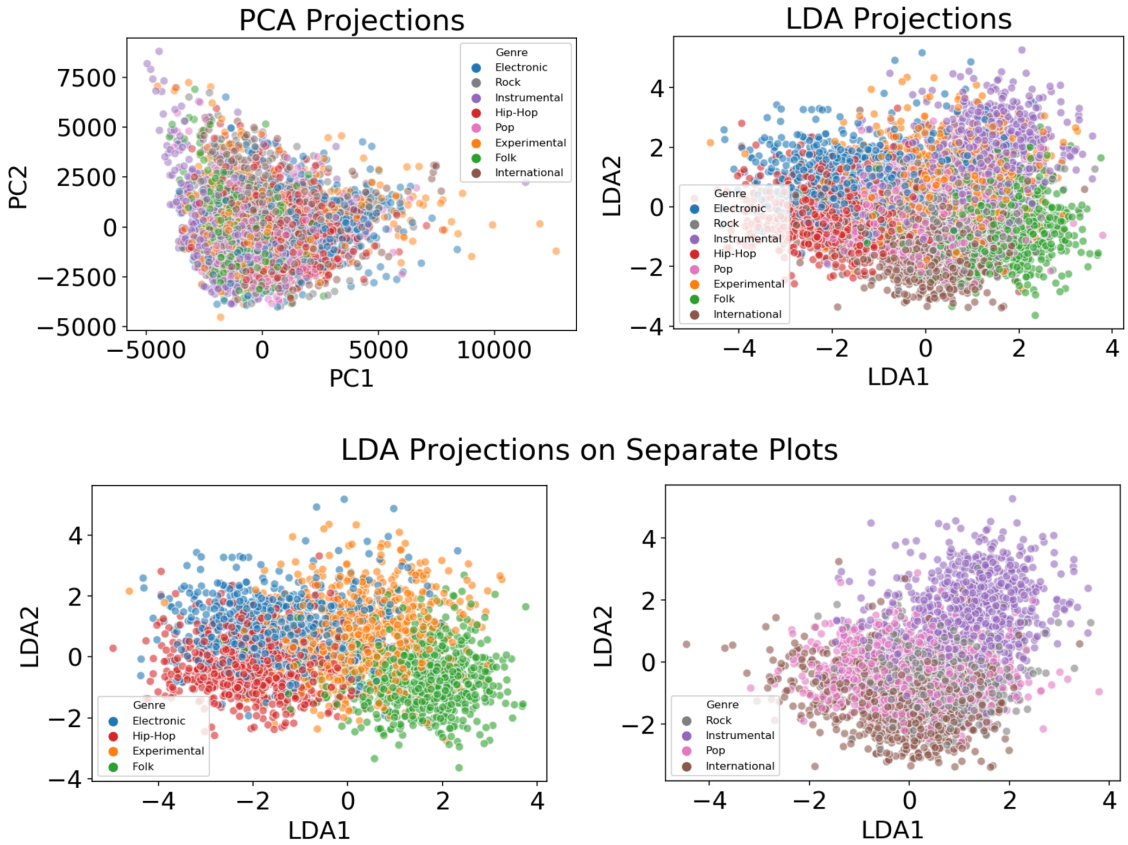

Figure 3: **Top Left:** Projection of the data onto the first two principal components displays poor separation. **Top Right:** Projection of the data onto the first two LDA discriminant coordinates of displays much better separation. Projections onto other discriminant coordinates also displayed more separation than PCA. **Bottom:** The LDA projections split onto two plots for clearer visibility. Choice of split is arbitrary.

2.1 The Test Data

We briefly check that the Kaggle test set looks similar to the training data. Figure 4 shows the PCA projections of the test data, and by informal comparison to Figure 3, these look similar. We also plot the feature means for the train data against those for the test data, and they again look similar.

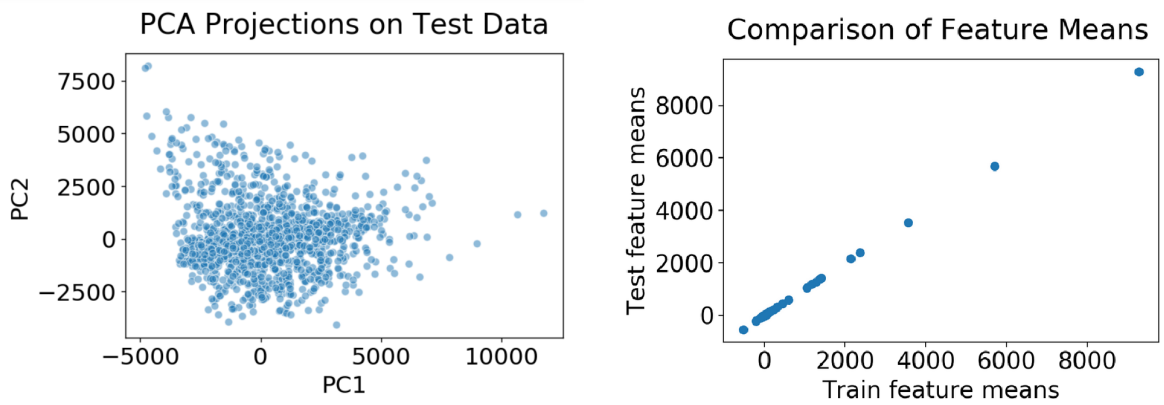

Figure 4: **Left:** Projection of the test data onto the first two principal components. **Right:** Scatterplot of the training set and test set means for each feature. Points along the $x = y$ line indicate close agreement. This agreement holds when we zoom in on the features with smaller means (not shown).

3 General Procedure

Our general pipeline for training and evaluating models is as follows:

1. Randomly split the initial set of 6000 observations for which we know the classes into a training set of 4800 observations and a test set of 1200 observations
2. Train a scaler on the train data and then use it to scale the train and test data.
3. Tune hyperparameters using a grid search and k-fold cross-validation on the training set.
4. Evaluate the model with the optimal hyperparameters on the held-out test set.
5. If performance is sufficiently better than baseline (usually 55% or greater), train model on full training set of 6000 observations and predict classes for the 2000 held-out Kaggle observations.
6. Use performance on public test set ("Kaggle Accuracy") to make final model selection.

Throughout, we use accuracy as our performance metric so as to be consistent with the metric used by Kaggle. All models were implemented using Python, and the code for our final model is available in the appendix.

Exceptions: In the case of Neural Networks, which were time-consuming to train and had a high potential to do well, we did not perform step 1. Instead, we trained and tuned the model on the full 6000 training examples (using cross-validation) and only evaluated performance on the public Kaggle leaderboard. In the case of Naive Bayes, there were no hyper-parameters to tune, so we did not have a validation step.

3.1 Dimensionality Reduction

Because the PCA projections exhibited poor separation, we focused on LDA for dimensionality reduction. We found that many classifiers performed as well or better when trained on the 7 LDA projections rather than all 518 features. This was also computationally efficient. Importantly, when we did this, we trained the LDA projection only on the size 4800 training set and evaluated test set performance by applying the learned LDA transformation and then the learned model. Learning the LDA transformations on the entire dataset beforehand would have resulted in overfitting.

Additionally, we briefly tried some other feature selection methods (ANOVA Correlation Coefficient, Recursive Feature Elimination, Tree-based Feature Selection), but this yielded little to no improvement. Combining methods by applying LDA projections to selected features also does only negligibly better than using the full data (improvement up to 1%). Going forward, we stick with LDA as our main dimension reduction method.

3.2 Scaling

We also examined three scaling methods (Standardization by mean and variance, Mix-max Normalization, and Quantile Transforms). However, we found that changing scaling methods had a negligible effect on performance (improvement up to 1%). We chose standardisation as the default scaling method, but occasionally switched to another method when it gave higher mean validation accuracy and held-out test accuracy.

4 Model Comparison

Before describing our final model, we overview results for a number of different classification approaches. Many attained $55\% - 57\%$ accuracy, but few performed better than that.

4.1 Baseline: Naive Bayes

As a baseline, we trained a Naive Bayes model on the unscaled data, treating the features as mutually independent and normally distributed. Its accuracy was 41.8% on the Kaggle data. This is better than random guessing, which, for 8 classes, would be 12.5%, but we would like to do better. In fact, after applying the Normal Quantile Transform scaling and the LDA projections, Naive Bayes achieved a 55.9% Kaggle accuracy – on par with many of our other models.

4.2 Generative and Linear Classifiers

For linear classifiers, polynomial input transformations are sometimes of interest. However, given the dimension of the data, we do not consider these below. It would likely lead to overfitting, as even with two-way interaction terms alone, we would have over $130,000$ features, much larger than the data size.

4.2.1 Linear and Quadratic Discriminant Analysis

LDA and QDA model the full generative distribution of the data and treat the class conditional densities as normally distributed. While LDA assumes a shared covariance matrix Σ, QDA takes an unconstrained covariance matrix Σ_k. The parameters are estimated using maximum likelihood with a closed-form solution, and for LDA, we used an eigendecomposition to speed up the prediction rule's computation.

To prevent overfitting, we used LDA/QDA with shrinkage. Note that while LDA was applied on the full standardised data, QDA was applied on the LDA projections to improve performance. A grid search suggested an optimal shrinkage parameter of 0.01 for both LDA and QDA, with a Kaggle accuracy of 56.2% and 56.0%, respectively.

4.2.2 Least Squares

We know that ERM under a squared loss can be found in closed-form. Again, to prevent overfitting, we added an L_2 penalty on the full standardised data. The grid search suggested an optimal regularisation parameter of 0.01, with a Kaggle accuracy of 54.9%.

4.2.3 Logistic Regression

Multi-class logistic regression (LR) uses the softmax function to model class conditional probabilities. ERM under the surrogate loss for multi-class LR has no closed-form, so we use the Stochastic Average Gradient method to estimate parameters. To fix overfitting, we add a penalty function on the full standardised data. A comparison of L_1, L_2, elastic net regularisation found that L_2 regularisation performed best. The grid search suggested an optimal regularisation parameter of 50 for L_2 penalty. This gave us a Kaggle accuracy of 55.4%.

Comment: Since we applied all the linear classifiers to the full standardised data, the magnitude of estimated gradients speak to feature importance. By ranking features by the average absolute gradients across the classes, the music feature `mfcc` and the summary statistics `mean` appear most frequently in the top-ranking features and thus contribute more to predictions.

4.3 K-Nearest Neighbors

Next, we tried KNN, a non-parametric approach which relies on measures of distance between points. On the full-dimensional data, KNN performed poorly ($< 50\%$) and became computationally expensive for large numbers of neighbours. On the LDA projections, it was both more efficient and more accurate (56.9% Kaggle accuracy). This improvement makes sense given that LDA projections maximize (Euclidean) distance between classes. A grid search over the number of neighbours k indicated that the optimal value was 59 neighbours. A comparison of $L_1, L_2 ... L_{10}$ norms found L_2 performed best.

4.4 Ensemble Methods

We also used Ensemble methods, which combine multiple models. These methods can be powerful, as they have the capacity to learn complex features. Though they often lack interpretability, this was not a concern here as the data is high-dimensional and understanding the features was not our focus. The ensemble methods we explored were Random Forests, Adaboost, Gradient Boosting, and a Voting Ensemble. As mentioned in Sec. 3.1, we used LDA to reduce the number of features as it gave the best results.

4.4.1 Bagging and Random Forests

A simple bagging model aggregating 95 decision trees yielded 55.2% accuracy on the 1200 observation test set, but a random forest approach did better. The advantage of Random Forest is that it further de-correlates the averaged decision-trees and reduces overfitting. To find the best hyperparameters for Random Forest we used a grid search with cross validation to have a realistic estimation of model performance. When trained on the LDA projections, this reached a Kaggle accuracy of 56.4%. Our final model had 175 trees, and each node used a maximum of 3 features to make each branching decision.

4.4.2 Adaboost and Gradient Boosting

To reduce the bias of our model we used boosting methods. For AdaBoost, each time a new weak learner is built, each datapoint receives a new weight depending on how well it is classified. Hyperparameters tuning led us to use 150 weak learners with a learning rate of 0.1. The performances remained limited with a 53.6% accuracy on the test set.

The second boosting method we used was Gradient Boosting. Instead of giving more importance to datapoints incorrectly classified, each new tree has parameters which are chosen to minimize the loss. This optimization step is done with gradient descent. The hyperparameters were tuned with grid search and cross-validation. We used 150 trees with depth 2. To prevent overfitting we also added an early stopping mechanism. If the losses on the validation sets were not diminishing quickly enough, the training stopped. Although it did the best among the ensemble methods, 57% on the Kaggle test set, Gradient Boosting still did not lead to major improvement.

4.4.3 Voting Ensemble

A voting ensemble combines predictions made by multiple models and takes the majority prediction as the final one. We applied this approach to 8 models on the LDA projections, but this yielded no improvement (56% on the Kaggle set). Figure 5 shows the performance by class of each "voter" on the 1200-observation test set. We see that their performance patterns are similar and that all do worst on the pop category. This is perhaps unsurprising given that pop is one of the classes least separated by the LDA projections (see Figure 3). Although this plot does not show whether the models are mis-classifiying the same exact observations, it does suggest that the ensemble may not add much because the classifiers tend to make the same errors. Figure 6, which displays accuracy for individual observations for the "Rock" group, further supports this notion.

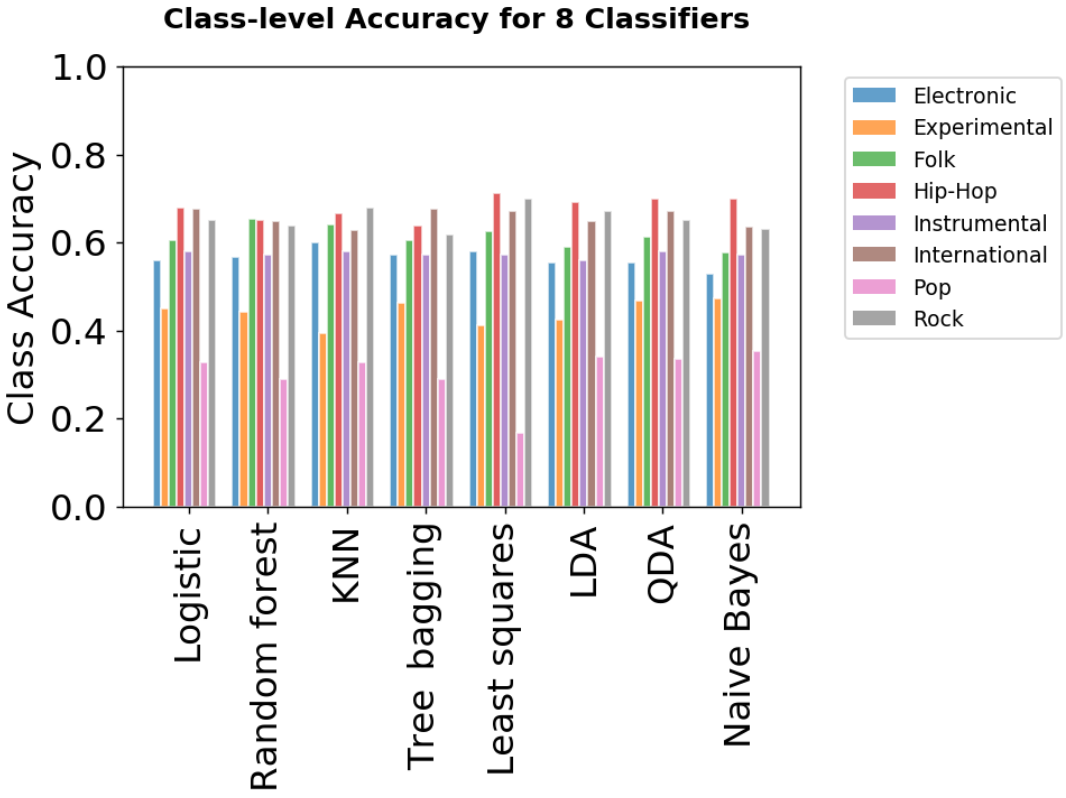

Figure 5: Accuracy by class for eight models trained on the LDA projections and used in the ensemble

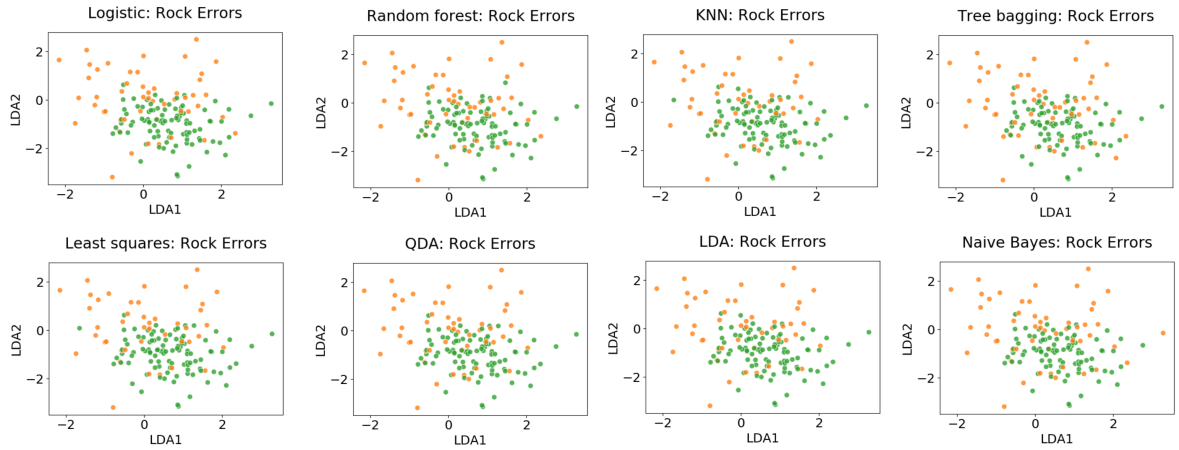

Figure 6: Correctly (green) and incorrectly (orange) classified points for the rock class for the held-out test set of 1200 observations. We see a similar pattern across the models in the voting ensemble, indicating they tend to misclassify the same points.

4.5 Best Model: Neural Networks

Our best model was a neural network. Since the data are high-dimensional, we focused on a Deep Learning Approach implemented in Pytorch. Note that we did not use any dimension reduction technique before the neural network as Deep Learning performs better in high dimensions (we checked this).

4.5.1 Implementation Details

The code used to obtain our neural network is provided in the appendix. We focused on fully connected Neural Networks. We used Cross-Validation with K-Folds (K = 3 so that the validation set was the same size as the test set on Kaggle) and made sure that the accuracy on the three validation sets used during K-fold was similar. To tune our many hyperparameters, we selected the model that had the best mean accuracy on those 3 validation sets. Because there is no implementation of GridSearchCV in Pytorch, we coded it ourselves with several for loops. This took many hours to run.

Our final network has the following shape: 3 hidden layers with 250 units per layer. Our learning rate was 0.0001 (quite small but it was the best), the batch size 40. The loss we used was the NLLLoss, the optimizer was Adam, and the dropout was 0.4 (used to prevent overfitting). To determine the number of epochs, we looked at where the mean accuracy on the three validation sets was the best. This gave us 225 epochs. The hyperparameters are summarized in Table 1.

Hyperparameter	**Value**
Activation Function	Relu
Batch Size	40
Number of Layers	3
Hidden units	250
Epochs	225
Learning Rate	0.0001
Dropout	0.4

Table 1: Hyperparameters obtained with 3-Fold Cross Validation.

4.5.2 Performance

The mean accuracy on the three validation sets was 62.3%, which was our best result. The mean accuracy on the three training sets was around 77%, indicating that our method still overfits the data a bit. After tuning our hyperparameters, we trained our model on the full training set (6000 observations) and achieved a Kaggle accuracy of 61.1%. As this result is close to the ones on the validation sets, we have evidence that our model does relatively well on new data. We can also see from Figure 7 that there is no sign of overfitting on the validation set (the validation accuracy does not decrease) and that the choice of our learning rate seems appropriate since the training accuracy converges.

5 Conclusion

To classify audio track genre, we tried a number of approaches and found that Neural Networks performed best for this problem. That said, no classifier we tried attained accuracy over 62% on the Kaggle set and many achieved around 56% – the differences in performance were not drastic. For models other than the neural network, using the LDA projections rather than the full data often improved performance.

Further work on this task might examine whether there is any advantage to first classifying observations into groups of genres (e.g. pop vs not pop) and then classifying within. Since there is some order in the features (they are obtained by moving a window through the audio tracks), Recurrent Neural Networks, which take into account this order, might be of interest. For this project, since we did not have access to any GPU, it was too time consuming to try to tune hypermarameters for a RNN.

For now, there remain many mis-classified points, particularly for the pop group. Speculatively speaking, the pop group's poor separability may reflect that that genre is particularly broad and often

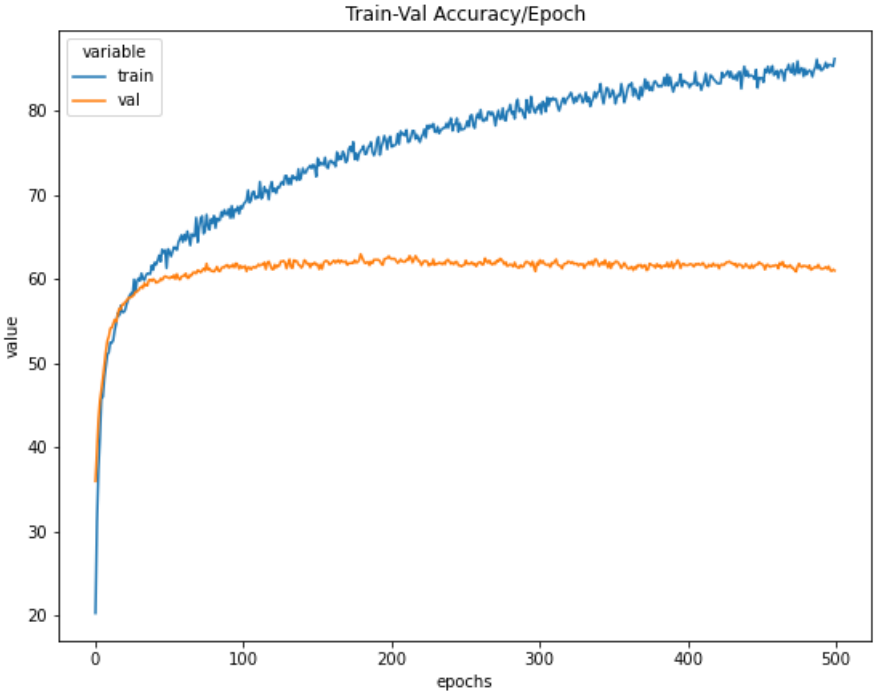

Figure 7: Accuracy evolution on training and validation set across epochs

draws on the other genres. In general, we see from the LDA projections that there is considerable overlap between classes, making the classification task a difficult one.

6 Code Appendix

This code is for our final model (the neural network). It was written in Python 3 using the Pytorch library. The seed was set to 15.

```
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import KFold
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

random.seed(15)
torch.manual_seed(15)


#Loading the data

root = '/Users/thomasdegouville/Documents/dev/SMLProject/'
dataX = pd.read_csv(root+'data/X_train.csv', index_col = 0, header = [0,1,2])
dataY = pd.read_csv(root+'data/y_train.csv', index_col = 0, dtype = 'category')

#Transform music categories to label to feed Neural Network
cat_map = dict(enumerate(dataY['Genre'].cat.categories))
dataY['cat'] = dataY['Genre'].cat.codes
Y = dataY.drop(['Genre'], axis = 1)

Xtest = pd.read_csv(root+'data/X_test.csv', index_col = 0, header = [0,1,2])


#Scale the data
#We used a standard scaler, the others (like MinMaxScaler) did not do well
scaler = StandardScaler()
dataX = scaler.fit_transform(dataX)
Xtest = scaler.transform(Xtest)
X, y, X_test = np.array(dataX), np.array(Y), np.array(Xtest)

#Define class datasets class to load the data

class myDataset(Dataset): #load training and validation sets
    def __init__(self, X_data, y_data):
        self.X_data = X_data
        self.y_data = y_data

    def __getitem__(self, index):
        return self.X_data[index], self.y_data[index]

    def __len__(self):
        return len(self.X_data)

class myDataset1(Dataset): #Load test set
    def __init__(self, X_data):
        self.X_data = X_data

    def __getitem__(self, index):
        return self.X_data[index]

    def __len__(self):
        return len(self.X_data)


#Accuracy measure for multiclass classification problem
def multi_acc(y_pred, y_test):
    y_pred_softmax = torch.log_softmax(y_pred, dim = 1)
    _, y_pred_tags = torch.max(y_pred_softmax, dim = 1)
    correct_pred = (y_pred_tags == y_test).float()
```

```
    acc = correct_pred.sum() / len(correct_pred)

    acc = torch.round(acc * 100)

    return acc



def train_model(X_train, y_train, X_val, y_val,
layers, hidden, batch, learning_rate):

    #Load the data
    train_dataset = myDataset(torch.from_numpy(X_train).float(),
     torch.from_numpy(y_train).long())
    val_dataset = myDataset(torch.from_numpy(X_val).float(),
     torch.from_numpy(y_val).long())
    train_loader = DataLoader(dataset=train_dataset,
                          batch_size=batch, shuffle = True)
    val_loader = DataLoader(dataset=val_dataset, batch_size=1, shuffle = False)

    #Define our NN architecture
    class Net(nn.Module):
        def __init__(self, dropout):
            super(Net,self).__init__()
            for i in range(layers):
                if i==0:
                    layer_in_size=518 #518 is the features dim
                else:
                    layer_in_size = hidden
                if i==layers - 1:
                    layer_out_size = 8 #8 is the number of categories
                else:
                    layer_out_size = hidden

                setattr(self,'dense_{}'.format(i),
                    nn.Linear(layer_in_size,layer_out_size))

            self.dropout=nn.Dropout(dropout)

        def forward(self,x):
            out=x
            for i in range(layers):
                if i==layers - 1:
                    out=torch.nn.functional.log_softmax(
                        getattr(self,'dense_{}'.format(i))
                        (self.dropout(out)), dim=1)
        #Use log_softmax function for the last layer as we use the NLLLoss
                else:
                    out=torch.nn.functional.relu(
                        getattr(self,'dense_{}'.format(i))(self.dropout(out)))
                    #for other layers we use the relu
            return out



    model = Net(dropout = 0.4) #Create model, the dropout was "manually" tuned



    criterion = nn.NLLLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    #define dictionnaries to store losses values
    accuracy_stats = {
    'train': [],
        "val": []
    }
    loss_stats = {
    'train': [],
    "val": []
    }

```

```
    #Train the model
    for epoch in range(500):
    #The maximum number of epochs is 500, after that it does not learn anymore
        model.train()
        train_epoch_loss = 0
        train_epoch_acc = 0
        for data in train_loader:
            inputs, output = data
            y_pred_train = model(inputs)
            optimizer.zero_grad()
            train_loss = criterion(y_pred_train, output.squeeze())
            train_acc = multi_acc(y_pred_train, output.squeeze())
            train_loss.backward() #Backward propagation
            optimizer.step()
            train_epoch_loss += train_loss.item()
            train_epoch_acc += train_acc.item()

        model.eval()
        val_epoch_loss = 0
        val_epoch_acc = 0
        with torch.no_grad(): #Check the results on the validation set
            val_epoch_loss = 0
            val_epoch_acc = 0
            for data in val_loader:
                inputs, output = data
                y_pred_val = model(inputs)
                val_loss = criterion(y_pred_val, output.squeeze(0))
                val_acc = multi_acc(y_pred_val, output.squeeze(0))
                val_epoch_loss += val_loss.item()
                val_epoch_acc += val_acc.item()

        loss_stats['train'].append(train_epoch_loss/len(train_loader))
        loss_stats['val'].append(val_epoch_loss/len(val_loader))
        accuracy_stats['train'].append(train_epoch_acc/len(train_loader))
        accuracy_stats['val'].append(val_epoch_acc/len(val_loader))
        if epoch%10==0:
            print(f'Epoch {epoch+0:03}: |
            Train Loss: {train_epoch_loss/len(train_loader):.5f} |
            Val Loss: {val_epoch_loss/len(val_loader):.5f} |
            Train Acc: {train_epoch_acc/len(train_loader):.3f}|
            Val Acc: {val_epoch_acc/len(val_loader):.3f}')
    return accuracy_stats['val']



def crossVal(dataX, y, layers, hidden, batch, learning_rate):
    kf = KFold(3) #Do the KFold with K = 3
    results = []
    for train_index, val_index in kf.split(dataX):
        #Split the training in 2 parts (train & validation)
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]
        results.append(train_model(X_train, y_train, X_val, y_val,
          layers, hidden, batch, learning_rate))
    results = np.mean(np.array(results), axis = 0)
    best_res = np.max(results)
    best_epoch = np.argmax(results)
    print("result acc", best_res)
    return best_res, best_epoch

def gridSearch(dataX, y, params):
    best_result = 0
    best_hidden, best_layers, best_batch, best_lr, best_ep = 0, 0, 0, 0, 0

    #Find the best hyperparameters with for loops
    for i in range(len(params['n_layers'])):
        for j in range(len(params['hidden_units'])):
            for k in range(len(params['batch_size'])):
                for u in range(len(params['learning_rate'])):
                    print("params", params['n_layers'][i],
                     params['hidden_units'][j],
                     params['batch_size'][k] ,params['learning_rate'][u])
                    CV_res = crossVal(dataX, y, params['n_layers'][i],
```

```
                    params['hidden_units'][j], params['batch_size'][k],
                     params['learning_rate'][u])
                    result = CV_res[0]
                    if result > best_result:
                        best_result = result
                        best_hidden = params['hidden_units'][j]
                        best_layers = params['n_layers'][i]
                        best_batch = params['batch_size'][k]
                        best_lr = params['learning_rate'][u]
                        best_ep = CV_res[1]
    print(best_hidden, best_layers, best_batch, best_result)
    return best_hidden, best_layers, best_batch, best_lr, best_ep


#parameters used in our GridSearch
params= {'hidden_units': [i for i in range(10, 500, 10)],
        'n_layers': [3,4,5,6,7],
        'batch_size': [20,30,40,50],
        'learning_rate': [0.01, 0.001, 0.0001, 0.00001]}


gridSearch_res = gridSearch(dataX, y, params)


def full_train(X_train, y_train, X_test, layers,
 hidden, batch, learning_rate, epochs):

    #Load the data
    train_dataset = myDataset(torch.from_numpy(X_train).float(),
     torch.from_numpy(y_train).long())
    train_loader = DataLoader(dataset=train_dataset,
                          batch_size=batch, shuffle = True)
    test_dataset = myDataset1(torch.from_numpy(X_test).float())
    test_loader = DataLoader(dataset=test_dataset, batch_size=1,
     shuffle = False)

    #Create same Neural Network
    class Net(nn.Module):
        def __init__(self, dropout):
            super(Net,self).__init__()
            for i in range(layers):
                if i==0:
                    layer_in_size=518 #518 is dim of features
                else:
                    layer_in_size = hidden
                if i==layers - 1:
                    layer_out_size = 8 #8 is the number of classes
                else:
                    layer_out_size = hidden

                setattr(self,'dense_{}'.format(i),
                nn.Linear(layer_in_size,layer_out_size))

            self.dropout=nn.Dropout(dropout)


        def forward(self,x):
            out=x
            for i in range(layers):
                if i==layers - 1:
                    out=torch.nn.functional.log_softmax(
                        getattr(self,'dense_{}'.format(i))
                        (self.dropout(out)), dim=1)
                else:
                    out=torch.nn.functional.relu(
                        getattr(self,'dense_{}'.format(i))(self.dropout(out)))
            return out




    model = Net(dropout = 0.4)
```

```
288
289
290
291     criterion = nn.NLLLoss()
292     optimizer = optim.Adam(model.parameters(), lr=learning_rate)
293     for epoch in range(epochs + 1):
294         model.train()
295         train_epoch_loss = 0
296         train_epoch_acc = 0
297         for data in train_loader:
298             inputs, output = data
299             y_pred_train = model(inputs)
300             optimizer.zero_grad()
301             train_loss = criterion(y_pred_train, output.squeeze())
302             train_acc = multi_acc(y_pred_train, output.squeeze())
303             train_loss.backward()
304             optimizer.step()
305             train_epoch_loss += train_loss.item()
306             train_epoch_acc += train_acc.item()
307     model.eval()
308     with torch.no_grad():
309         y_test = []
310         for data in test_loader:
311             inputs= data
312             y_pred_test = model(inputs)
313             _, y_pred_tags = torch.max(y_pred_test, dim = 1)
314             y_test.append(y_pred_tags.item())
315     return y_test
316
317
318 y_test = full_train(X, y, X_test, gridSearch_res[1],
319  gridSearch_res[0], gridSearch_res[2], gridSearch_res[3], gridSearch_res[4])
320
321 for i in range(len(y_test)):
322     #assign categories to labels to have the right .csv file
323     y_test[i] = cat_map[y_test[i]]
324 print(y_test)
325
326 def export_to_csv(y_hat, filename):
327     df = pd.DataFrame({'Genre': y_hat})
328     df.index.name = 'Id'
329     df.to_csv(filename)
330
331 export_to_csv(y_test, root+'Y_test_mlp.csv')
```