ELEC 301 Final Project Report

Rosemary Lach, Elijah Schwartz, Eunice Tan, Nick Thevenin

I. Introduction

Project Name: *Music Genre Classification*

The goal of this project was to implement a model that is able to classify music genres with a high degree of accuracy. The data used in the project was given in the form of audio files. The training data was labeled with the known music genre, making this an instance of supervised learning.

As a group, we created a GitHub repository to house all the needed code for data exploration, feature extraction, model training and testing, along with any additional resources. Our main programming language was Python, which we chose based on its strong capabilities for data analysis, and access to libraries. The code for the project can be found here: https://github.com/rosemarylach/elec301project.git.

Participation in this project included competing in a Kaggle competition alongside the rest of the class. Competition rules permitted a maximum of 4 prediction entries a day, with two of those being our final submissions for judging. Competition leaderboards allowed for us to monitor our team's learning accuracy score while comparing to the rest of the teams. Our final accuracy scores and leaderboard placement will be discussed later in the report.

II. Data Exploration

As stated earlier, the given training data was labeled with the appropriate genre of music. This was a nice part of our data exploration process, as most of our models (and machine learning algorithms in general) perform better for supervised learning. One qualm we had with the labeling method was extracting the genre from the filenames. While we believe the filename structure was the most efficient and easy way to store this information, the fact that the number labels after the genre were of different length made it difficult to extract this information. We propose using a 3-digit number for all files, for example blues001.wav instead of blues1.wav, so feature names can be more easily extracted using simple list splicing.

To test our data ourselves, we had to cut data from our training data, providing us with even less to use at times. In order to do this, we set aside part of our dataset (typically 20-30%) and used it as a testbench, as we only had so many submissions per day. This was especially useful as the deadline approached and we only had a very limited amount of submissions. As a result, we were able to freely test our models without the submission constraint and find more optimal parameters for our models. Initially we separated our data by creating entirely new csv files for split training and test data and running our custom script accuracy.py to test performance. However, later we discovered that sklearn had functions to split and test accuracy within the files themselves that was much quicker, so we transitioned to using that instead.

The biggest hurdle with using raw data was formatting, processing and extracting the necessary features from it. Our experience with this is discussed below. Based on our accuracy results, we are content with how we handled the data. To fit our models better, we would have appreciated more training data, mainly so that we could have more samples in our test set when we split the data.

III. Feature Extraction

The majority of our models, and all of our successful models, were not run on the raw data itself, but rather a collection of time-domain and frequency-domain features of the provided .wav files. The features that we chose to extract were based on the same features used in Chillara's study, and we decided to use the same parameters because we figured they would provide a good baseline since they had been successfully tested before, and could easily be extracted using libraries. If these proved to give bad predictions we intended to update our feature selection process, but when these gave very good results, we decided to keep them as they were because the additional code runtime was not worth what we believed would be minimal improvement to our models.

Our feature extraction process takes place in the file *feature_extraction.ipynb*, where we use the librosa library to read every .wav file and directly pull our desired features out of the signal. Since librosa abstracts much of this process away, we were able to focus primarily on the feature selection and organization of the data in a way that was simple to use for all of our models.

The code was structured to build a pandas dataframe of all of the desired features in both the training and test data. Additionally, the training dataframe included the genres to be used as labels during the learning, and the test dataframe included the .wav filename to be used when building up the final predictions csv. The dataframes contained one row for every sound sample, and a column for each feature. We chose this structure because it provided the simplest way to load this data into our model since it was the same general format required by the learning models in the scikit-learn library. After constructing the dataframes row by row, we exported them to separate .csv files that were saved to our main directory. This proved to be very useful because it allowed us to work independently to implement models in separate files and simply read the csv directly into a new dataframe each time we ran a new algorithm. This also significantly reduced the runtime of our code, because the feature extraction process took multiple minutes to run, but once we did it the

one time we didn't have to do it again. Lastly, to prepare these features for going through our models, we had to normalize the data. This step was performed in the code for each of the individual models, but it proved essential for all of our models to work correctly.

We did this feature extraction process on both the labeled training data and the unlabeled test data to create a separate csv file for each.

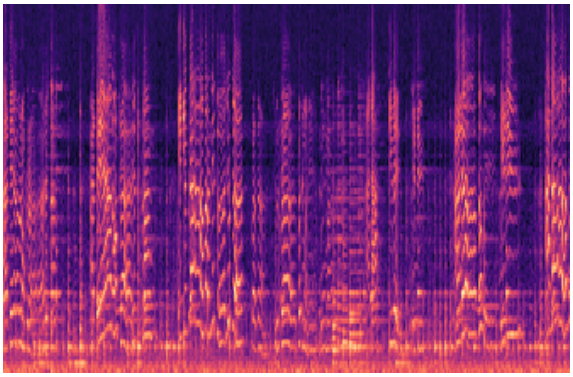
Short descriptions of our 91 selected time and frequency domain features are outlined below. The starred signals indicate that our stored value was just a mean and standard deviation that represented the measurements taken at multiple frames in the data.

- **Central Moments:** mean, standard deviation, skewness, and kurtosis of the signal
- **Zero Crossing Rate*:** the rate at which a signal changes from positive to negative or negative to positive. We added a slight offset to the data before calculating this to minimize the effect of noise.
- **Root Mean Squared Energy*:** Defined as $\sum_{n=1}^{N} |\, x(n) \,|^2$.
- **Tempo:** beats per minute of the recording
- **Mel-Frequency Cepstral Coefficients*:** a representation of the short-term power spectrum of a sound**.** We calculated 20 coefficients to take mean and standard deviation of to provide 40 features.
- **Chroma*:** A vector corresponding to the total energy of the signal in each of the 12 pitches (C, C♯, D, D♯, E, F, F♯, G, G♯, A, A♯, B). We calculated the means and standard deviations to provide 24 features
- **Spectral Centroid*:** The frequency around which most of the energy is centered.
- **Spectral Contrast*:** Measures the decibel difference between spectral peaks and valleys in each frequency subband. We calculated the means and standard deviations at 7 frequency bands to provide 14 features
- **Spectral Rolloff*:** The frequency where a certain percentage of the total spectral energy (in our case, 85%) lies below.

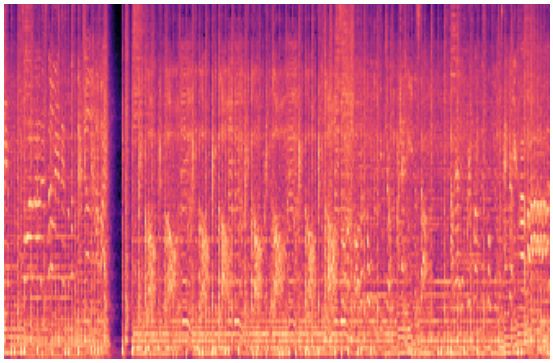
In addition to extracting these features, another entirely separate method we used to summarize each sample was to create a MEL spectrogram. We thought this would be a good idea because some of the best music genre classification results we found in our readings came from doing image classification on these spectrograms. We discuss our attempt to do this ourselves in the next section. The MEL spectrogram was chosen over a standard spectrogram, because it is designed to model human hearing perception which is more ideal for genre classification. We also used librosa to generate and plot each MEL spectrogram, and then saved each to a separate png file in either trainspecs or testspecs depending on which dataset the sample was from. These pngs could then be used later to classify genre based on the images instead of the signal features described above.

Some challenges with this process as opposed to our main feature extraction was that because we were plotting an image for each sample, the code took much longer to run. Also, since we had to worry about processing these images later, we had to take steps to remove axes, borders, and labels so that none of this information would confound our results or increase the runtime of our model implementations with no benefit.

All of our MEL spectrogram files can be found in our git repository, but two (for blues and metal) are provided in this paper to highlight the contrast in some of the images between genres, which demonstrates why we thought this was a useful feature to extract.



Blues Spectrogram



Metal Spectrogram

IV. Model Selection and Testing

For our models, we used K nearest neighbors (KNN), multinomial logistic regression (MLR), support vector machine (SVM), a multi-layer perceptron neural network (MLP), and a random forest classifier. These models were chosen based on their effectiveness in multi-class classification problems, ease of implementation, and their overall speed on large datasets. In each model, we set aside a fraction of our training dataset for the purpose of viewing our models' expected performance on the larger testing dataset. We also used these results to employ various methods for visualizing our data, such as confusion matrices, that were formed from these samples of the training data. All of our models were implemented using the library scikit-learn.

One thing to note is that our data features are not linearly separable. While this is intuitively clear based on the nature of the problem, we tested this by training an SVM using an extremely large regularization parameter, which made the minimum margin very small. However, even this could not make our model 100% accurate, meaning that our features are not linearly separable. This is important to note because it means that classifiers that assume linear separation, such as logistic regression, will be less accurate.

a. K Nearest Neighbors

The K-Nearest Neighbors algorithm was selected due to its simplicity, which leads to ease of implementation and testing. KNN is essentially based on "closeness" of data points; in theory it predicts data labels based on the labels of the majority of its closest neighbors. We went over this algorithm in class, which also contributed to us using it for the project as it was familiar to us already.

Initially we assumed K-Nearest Neighbors wouldn't perform the best based on its simplicity compared to our other models. The results were actually surprising, as one of our best accuracy scores was achieved from a KNN model. As our data was non-linearly separable, it does make sense KNN would give solid predictions.

For number of neighbors, k = 5: the accuracy was 73.3%.
For number of neighbors, k = 10: the accuracy was 77.7%.
For number of neighbors, k = 17: the accuracy was 71.1%.

The optimal k being 10 was not outside of what we thought was reasonable, but we expected around k=25 to perform better because online resources said that often, optimal k is about sqrt(n) where n is the number of training samples. However, when testing on the split data, we found that the k values above performed better, which is why we chose to submit those to Kaggle instead. Since selection of k is not an exact science, we were not too disappointed that our initial baseline was off because the final k values were still very successful.

We attempted to run principal component analysis (PCA) alongside our KNN model to reduce the dimensionality of our data because typically KNN can be negatively affected by unimportant data contributing to the distances. The results of doing this proved to be insignificant as no noticeable improvements to the model's accuracy were found. This is likely due to how we extracted our features initially; if we already identified the most important data features, dimensionality reduction methods like PCA are redundant. Because of this, we decided not to use PCA for any of our other models.

b. Multinomial Logistic Regression

We selected Multinomial Logistic Regression because of its speed and elegance compared to a one-vs-all approach of many binary classifiers. Therefore, we used a cross-entropy calculation in our model, with the minimum iterations needed for the solvers to converge. This way we were able to avoid overfitting our data to the training set. This model achieved 71% accuracy on the test data submission.

c. Support Vector Machine

We selected three lambdas for this model: 0.1, 0.01, and 0.001. These values were based on standard lambdas for machine learning for SVM. It was chosen due to its coverage

in class, and personally, it was also one of the easier models to understand and implement due to the use of sklearn.
For lambda = 0.1: the accuracy was 52.2%.
For lambda = 0.01: the accuracy was 75.6%.
For lambda = 0.001: the accuracy was 64.4%.

d. Multi-Layer Perceptron Neural Network and Spectrogram Image Classification

This was one of the models not covered in class, but many sources, including RichB and the Internet, encouraged us to step out of our comfort zone, so we chose this model both to challenge ourselves and because it is an effective classification tool for models that are not linearly separable. Essentially, it consists of an input and an output layer that contain neurons for each of the input parameters, and a single neuron for the output classification result. The part that makes this algorithm unique are the neurons in the hidden layers which transform the values from the previous layer with a weighted linear summation and run the result through a non-linear activation function - a method that we learned was very powerful in class. Our hidden layers in our Kaggle submission had 100, 50, and 25 neurons respectively (chosen based on suggestions from online resources) and its accuracy was 75.5. However, after doing more tests with our own split data, we found that a single hidden layer with 100 neurons tended to perform better. We believe that, although this model is commonly used in all sorts of classification in industry, it is typically used in cases in which there is a very large dataset—much larger than the one thousand music samples we used. Because of this, although using a neural network is very effective, it is not the best for this type of dataset.

The multi-layer perceptron network was the model we used to test our alternative method of classification of the MEL spectrogram images. The algorithm itself functioned almost identically to when we used the extracted feature vectors, but instead we loaded each png file into a numpy vector and ran that through the system instead. Unfortunately, this did not perform anywhere close to as well as the papers we read, which said this was the best classification method for them. In fact, it's accuracy of only 40% was significantly

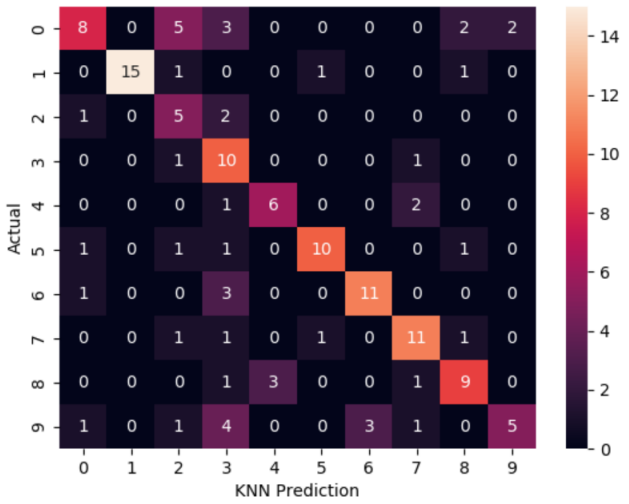
worse than all of our other algorithms that used the standard 91 extracted features. Also, since the image file vectors are much longer than 91 elements long, the algorithm took much longer to run at about 10 minutes vs the 30 seconds for our other methods. For both these reasons, it seemed like it would not be worth it to explore the spectrogram image classification any further because it would take a lot of effort to make our accuracy competitive with our other methods, and the runtime was prohibitive of us being able to quickly test different versions in the timeframe for this project. If we were to continue with this method, however, the next thing we would attempt would be to analyze the spectrograms in small chunks instead of each image as a whole because short intervals of time in a genre are more likely to be similar to one another than an entire 30 second clip. This is similar to how we took the means and standard deviations of smaller frames within the samples when extracting our other features instead of pulling the features from the entire signal. While this additional work on image classification would have been interesting, analyzing these smaller chunks would be even more computationally costly than what we previously ran, and is simply not feasible for this project.

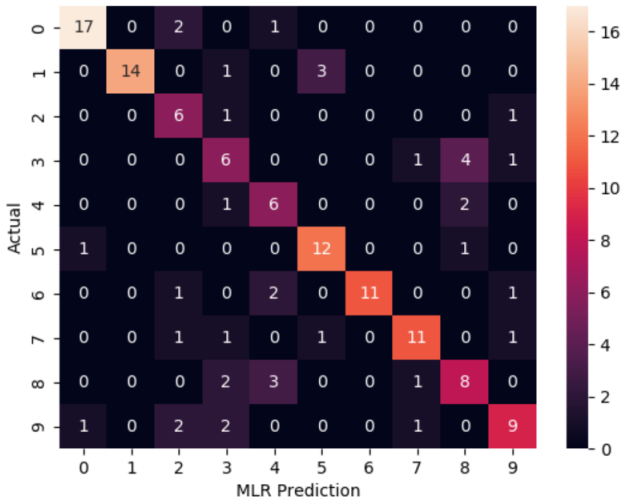
e. Random Forest Classifier

For our second model not introduced in class we decided to use a random forest classifier. We believed that this model would work well on our training data due to the fact that it relies on the predictions of many decision trees, which organize our features by information gain and determine the final importance of each feature. These decision trees work well on non-linearly separable data, as opposed to logistic regression. We found that increasing the number of decision trees slightly increased model accuracy with a trade-off of runtime.

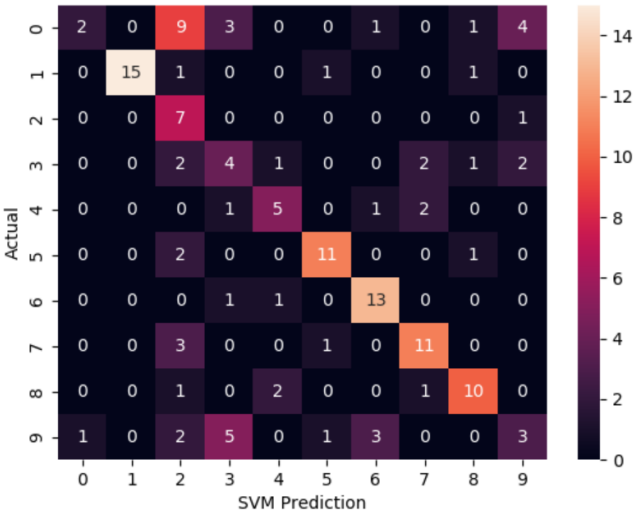
For N_trees = 100: 69.2% accuracy on sample test set.

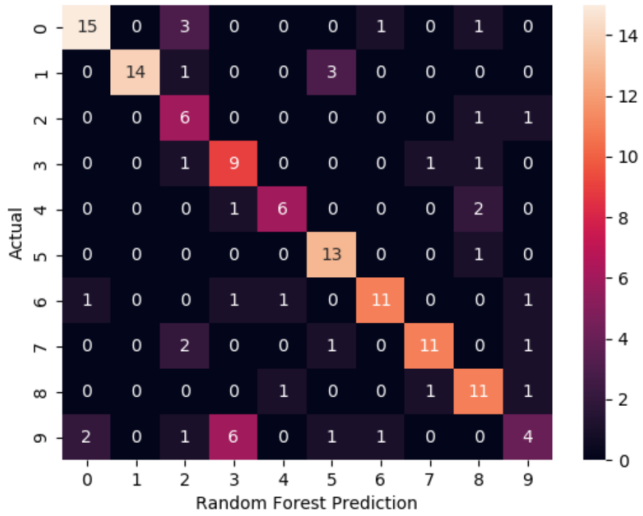
For N_trees = 1000: 71.4% accuracy on sample test set.



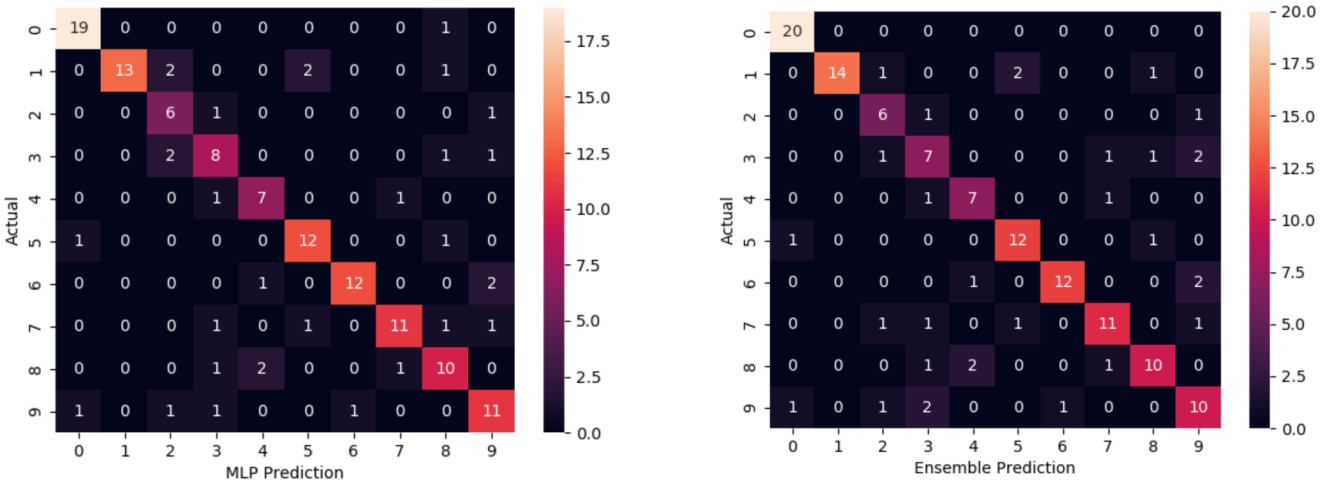


Confusion Matrices for K-NN and MLR classifications





Confusion Matrices for SVM and Random Forest Classifications



Confusion matrices for MLP and Ensemble classifications

As evidenced by our confusion matrices, many of our models were effective in accurately predicting samples from classical, metal, pop, and jazz genres. Classical was clearly the most distinct genre, which makes sense: it is very unique compared to the other ones on the list. On the other hand, rock was not very accurate in most confusion matrices. Disco has varying results: in some models, like KNN, it performed very well, but in others—especially SVM—it performed much worse. Our models also often confused blues for country, likely due to their foundational similarities.

V. Final Design

f. Ensemble Learning

We chose ensemble learning because it uses different models all together, resulting in a more effective algorithm with better prediction capabilities. Because we have a lot of different models which work differently, a lot of the predictions from these previous models have different types of errors (see the disco genre, which KNN could predict very well, but SVM had a lot of trouble with vs the metal genre which was the other way around). Because of this diversity, using ensemble learning was a decisive next step for us. We took the highest-performing parameters from the different models presented: for example, for KNN, k = 10 had the best performance, and therefore, we implemented that parameter into our ensemble learning model. Another example is from SVM. While testing with different lambda values, we found that lambda = 0.01 had a high degree of accuracy, so we also used that value when using the ensemble learning model. With these parameters, we were able to achieve an accuracy of 80% on the public leaderboard.
In total, the models that we used were as follows:

- KNN (10 neighbors)
- Logistic Regression
- SVM (gamma = 0.01)
- MLP (1 hidden layer with 100 neurons, initial learning rate of 0.01)
- Random forest (15 max features)

To calculate the final prediction for ensemble learning, instead of allowing each of the classification algorithms to make a single prediction decision, we had them all return a value for each of the possible genre categories corresponding to the probability that the model thought the test sample belonged in that category. We then added these probabilities together for every model, and classified the sample as the genre with the highest sum. This method works because if any one method is very confident that a particular sample belongs in a certain category, it will drive the final sum up.

One final modification we made to this process was to provide slight weights to some of the models when calculating the probability sums, because that places more emphasis on the models we are more confident in when selecting the genre (such as random forest because of its already high accuracy when running on its own). Based on the individual performances of each of our models, and after some trial and error, we selected a weight of 1.3 for random forest, 1.1 for KNN and MLP, and 1 for SVM and logistic regression.

Another method we tried to combine the data was to take the prediction of the most confident model based on the returned probabilities, and use that instead of summing the probabilities of each model. This is slightly different as it essentially selects the best model to use for classifying each sample, which could be different depending on the genre of the sample. This version of ensemble learning received a score of 78.8% on Kaggle which was mostly comparable with the version above.

Theoretically, the fact that ensemble learning is most effective makes sense because it is a multiple classifier system: it combines these models in the form of hypotheses and aggregates them together to create a better predictive model.

In all, the process to produce this final and best prediction looked like this:

1. Extract features exactly as described in the Feature Extraction section and load those features from the saved training features csv
2. Split our labeled data into a training and test set
3. Run all classification algorithms on the split training data with the optimal parameters we found when testing them individually
4. Combine the results of each algorithm as described above with weights based on each model's individual performances
5. Test the predictions of the model on the split test data, and note the accuracy percentage
6. Repeat steps 2 through 5 while tweaking the parameters of the models, and the weight values until the accuracy levels off at the highest value we could

achieve (around 78% on average, but with some variation depending on the initial training/test split)

7. Using the parameters optimized in step 6, train the ensemble learning model on all labeled training data
8. Load the features from the saved test features csv, and use the model generated in step 7 to predict their labels
9. Load the predictions into a csv and submit to Kaggle

VI. Conclusion

During the scope of this project, we effectively classified the genres of music audio files with an accuracy of 80%.

Most of our models performed around the 70-80% accuracy mark, and we believe that a large reason for this was the preprocessing process. As a result of the extensive feature extraction method incorporated—with a total of 91 features, our data was pretty accurate. The importance of feature extraction was unexpected (as we did not cover much of this in class) but a welcome surprise, and it was the basis for most of our success when it came to the models used, especially since our data is not linearly separable. And when it comes to music—with its spectrum of genres (even now, it is debatable whether genres really exist because the 'boundaries' between these types of music are so thin: there are so many songs and pieces that can fall in one category or another, and sometimes it is difficult to tell).

We discussed other methods of preprocessing, like subsampling, upsampling, and resampling, but ultimately decided that the feature extraction was more important and required more focus as a team. Additionally, even though the genre classification was unbalanced, it wasn't a glaring issue at first glance. Given more data and time, then, balancing data might have been a good idea. We also considered using spectrograms (and performing classification on the images themselves) instead of pure feature extraction, but they resulted in bad initial predictions, leading us to stick with feature extraction.

The project was coded in Python, but feature extraction—as a result—was a lengthy process and we had to wait a significant amount of time for its completion (so that we could begin the model selection, data exploration, and training). Potentially, we could have used C to expedite the process, but we were also more comfortable with Python and therefore chose to use that programming language, even if it can be slower.

We chose models based on perceived effectiveness and what was covered in class. For example, KNN works well with non-linearly separable data, so we chose that model as a starting point and worked from there, using resources like sklearn and its optimized algorithms for classification in order to draw conclusions about the data. With these models, we explored changing parameters—like lambda value in SVM and the use of PCA along with KNN (which turned out to be insignificant when classifying data).

Hypothetically, if we had more time, we could have researched more models than what was covered in class and potentially coded our own models from scratch as an exercise in thorough understanding of these models. Then, we could have compared our efforts with that of sklearn and other resources on the Internet.

Another constraint was the limited amount of data given. As stated above, algorithms like artificial neural networks are most effective with a larger dataset, and although we were able to extract a large number of features from the data, there were only so many wav. files to extract data from, and some of the genres were unevenly represented (there were much more disco files than there were rock files). As an example, if we had more data, the MLP model would probably work much better.

Overall, we are happy with the direction and results of our project given the constraints. Our first success was choosing significant features, our second was implementing effective models, and our third was the analysis of such data and the accuracy of our music genre classification, which we are very proud of.

VII. References

https://scikit-learn.org/stable/auto_examples/neighbors/plot_nca_dim_reduction.html

https://www.geeksforgeeks.org/hyperparameters-of-random-forest-classifier/

https://www.geeksforgeeks.org/advantages-and-disadvantages-of-logistic-regression/

https://medium.com/@b.terryjack/tips-and-tricks-for-multi-class-classification-c184ae1c8ffc

https://en.wikipedia.org/wiki/Multiclass_classification

https://towardsdatascience.com/comprehensive-guide-to-multiclass-classification-with-sklearn-127cc500f362

https://towardsdatascience.com/how-to-balance-a-dataset-in-python-36dff9d12704

https://data-flair.training/blogs/python-project-music-genre-classification/

https://www.analyticsvidhya.com/blog/2021/06/music-genres-classification-using-deep-learning-techniques/

https://towardsdatascience.com/comprehensive-guide-to-multiclass-classification-with-sklearn-127cc500f362

http://www.scholarpedia.org/article/Ensemble_learning

http://www.scholarpedia.org/article/Ensemble_learning#Data_Fusion

https://1library.net/document/yr3wv8jy-music-genre-classification-using-machine-learning-algorithms-comparison.html?utm_source=related_list