Identifying Genres of Music - Comparing Model Performances

Adam Lizerbram

***Abstract*—Utilizing several different models to compare the performance of the multiclass classification problem of identifying the genre of music from a given audio file.**

I. Introduction

THE purpose of this project is to test several different models on the same classification problem to compare how they perform. The problem is as follows: given a 30 second audio clip, identify which genre of music it belongs. This is a rather well-known problem in machine learning, so I wanted to try multiple models to see which ones perform the best, as well as get myself more familiar with using a variety of models.

II. Dataset

THE dataset I used is the GTZAN Dataset for Music Genre Classification, which contains 10,000 labeled 30-second audio clips, images of their spectrograms, and csv files with various numeric features of each entry. It includes audio from 10 different genres (Blues, Classical, Country, Disco, Hip-Hop, Jazz, Metal, Pop, Reggae, Rock), with 1,000 samples for each genre.

A. Data Visualization

There are several ways to visualize the data in our dataset. The Librosa library in Python has many useful functions for working with audio files.

1) Raw wav file: One way to visualize the data is by plotting the raw wav file. This is not very helpful as it does not provide any details, other than the amplitude or volume of the audio over time. Figure 1 is an example of what that looks like when plotted in Python.

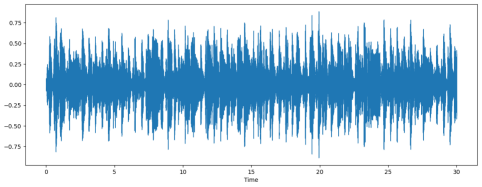

Fig. 1. Raw waveform visualization of an audio file.

2) Spectrogram: This is a far more useful way to visualize an audio file. A spectrogram is a plot of frequency over time, which is calculated using a Fourier transform, which extracts the most prominent frequencies from a function. The spectrogram is in the form of a heatmap, where the brighter colors represent stronger frequencies. See Figure 2 for how a spectrogram looks when plotted in Python.

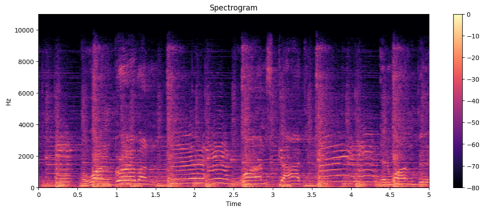

Fig. 2. Spectrogram visualization of an audio file.

3) Chromagram: This is a very powerful method of musical analysis on an audio file. It is similar to a spectrogram, but instead of plotting frequencies, it classifies the frequencies into musical pitches. The red colors represent more prominent notes at a given moment. The resulting plot almost looks like a MIDI display. See Figure 3 for what a chromagram looks like when plotted in Python.

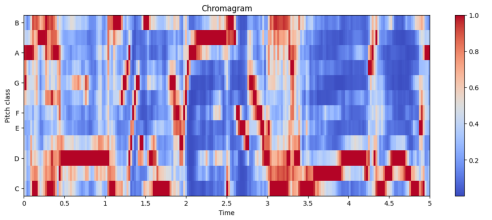

Fig. 3. Chromagram visualization of an audio file.

4) CSV File: The dataset provides a CSV file with the mean and variance of 30 numeric features that can be extracted from an audio file. It also has the labels for each entry. This is what will be used to train the models, as they require numeric data to train on. Figures 4 and 5 show part of what the CSV file looks like using a Pandas dataframe in Python. The data includes the original file name which is not useful for testing, but will be useful later on when we want to locate certain files which were guessed correctly or incorrectly. However, the rest of the numeric data and the labels are used in the training and testing phases.

	filename	length	chroma_stft_mean	chroma_stft_var	rms_mean	rms_var	spectral_centroid_mean	spectral_centroid_var	spectral_bandwidth_mean	spectral_bandwidth_var
0	blues.00000.0.wav	66149	0.335406	0.091048	0.130405	0.003521	1773.065032	167541.630869	1972.744388	117335.771563
1	blues.00000.1.wav	66149	0.343065	0.086147	0.112699	0.001450	1816.693777	90025.690866	2010.051501	65671.875673
2	blues.00000.2.wav	66149	0.346815	0.092243	0.132003	0.004620	1788.539719	111407.437613	2084.565132	75124.921716
3	blues.00000.3.wav	66149	0.363639	0.086856	0.132565	0.002448	1655.289045	111952.284517	1960.039988	82913.639269
4	blues.00000.4.wav	66149	0.335579	0.088129	0.143289	0.001701	1630.656199	79667.267654	1948.503884	60204.020268

Fig. 4. Provided CSV file with many features from the audio files.

B. Data Normalization

Before training the models, the data needs to be normalized to prevent any skewed or biased outcomes. This can be done

mfcc16_var	mfcc17_mean	mfcc17_var	mfcc18_mean	mfcc18_var	mfcc19_mean	mfcc19_var	mfcc20_mean	mfcc20_var	label
39.687145	-3.241280	36.488243	0.722209	38.099152	-5.050335	33.618073	-0.243027	43.771767	blues
64.748276	-6.055294	40.677654	0.159015	51.264091	-2.837699	97.030830	5.784063	59.943081	blues
67.336563	-1.768610	28.348579	2.378768	45.717648	-1.938424	53.050835	2.517375	33.105122	blues
47.739452	-3.841155	28.337118	1.218588	34.770935	-3.580352	50.836224	3.630866	32.023678	blues
30.336359	0.664582	45.880913	1.689446	51.363583	-3.392489	26.738789	0.536961	29.146694	blues

Fig. 5. Provided CSV file with many features from the audio files.

by simply subtracting the mean and dividing by the standard deviation for each feature. This results in each feature being roughly normally distributed with a mean of 0 and standard deviation of 1. This can be done using the StandardScalar class from the preprocessing sublibrary from the sklearn library in Python. There are plenty of other methods for normalizing data which would work just as well, but this method was simple and effective, as well as made logical sense since the data is roughly normally distributed to start.

Additionally, the labels need to be converted to numeric values for testing purposes. This can be done with the LabelEncoder class which also comes from sklearn.preprocessing. Figure 6 and 7 show examples of what this code looks like in order to perform these steps.

```
[ ]  1 scale = StandardScaler()
     2 X = scale.fit_transform(np.array(df.iloc[:,1:-1], dtype=float))
```

Fig. 6. Using the StandardScaler class to normalize the data.

```
1 classes = df.iloc[:,-1]     # column of labels
2 encoder = LabelEncoder()
3 y = encoder.fit_transform(classes)
```

Fig. 7. Using the LabelEncoder class to convert labels to numeric values.

C. Data Splitting

Splitting data is a very important step in machine learning, because the data needs to be split into three distinct subsets, each of which has a unique job in terms of evaluating the model. The first subset is the training data subset. This data will be used to directly train the model. In this case, I used approximately 65% of the data as part of the training subset. The second subset is the validation subset. This data will be used to validate the training data. In other words, the validation data is used to determine the success of training the model, and can be used to compare the success of different hyperparameter combinations in order to choose the best option. I used approximately 25% of the total data for validation. The final subset of the data is the testing subset. This is the data that is used purely to test the final model with the chosen hyperparameters. As soon as this data is chosen to be for testing, it will not be touched until the model is fully trained with the best hyperparameters. I used 10% of the data for testing. The train_test_split function from the model_selection sublibrary within the sklearn library is a useful way to randomize the data splitting process. Figure 8 shows what the code to split the dataset into the respective subsets looks like.

```
[ ]  1 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)
     2 X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.3)
```

Fig. 8. Using the train_test_split function to split the dataset into the three randomized subsets.

Now that we are more familiar with the dataset, normalized the data, and split the data into the three data subsets, we are ready to set up and train the models.

III. Model Descriptions and Training

THREE different models will be used to solve the classification problem. They are ordered in increasing complexity, and in increasing expected success.

A. K-Nearest Neighbors (KNN)

KNN is the most simple model that we will use. It is trained by simply plotting the training data, and when testing, the label for each test data point is voted for by the nearest k training data points. The value of k is a hyperparameter for this model. I want to try many different values for k to see which one performs the best. Figure 9 and 10 show the code and training results for this model, respectively. Figure 10 is a plot of validation accuracy over increasing values of k.

```
1 knn = KNeighborsClassifier(n_neighbors=5)  # You can adjust the number of neighbors
2 knn.fit(X_train, y_train)
3 knn.score(X_valid, y_valid)
```

Fig. 9. Creating one KNN model with $k = 5$.

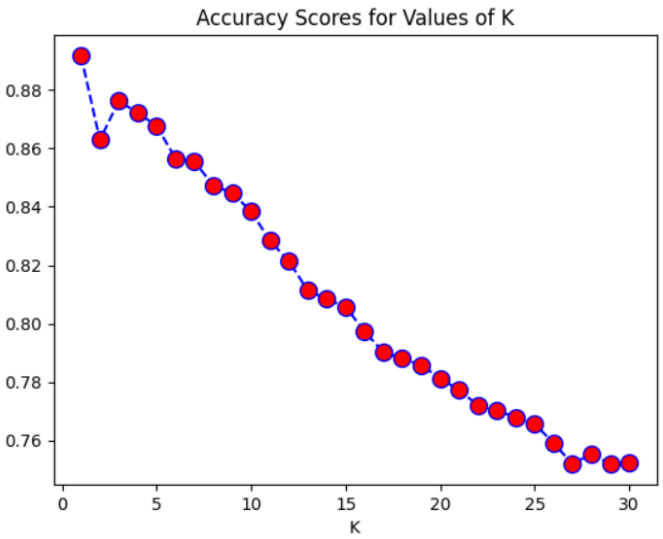

Fig. 10. Results of training KNN models with k values ranginf from 1 to 30.

From this plot, it is clear that $k = 1$ is by far the best value of k. It makes sense that as k increases, the accuracy decreases, since larger k values mean that further points can vote for the label of an unknown point, which causes more inaccuracies. By using $k = 1$, we achieve a validation accuracy of almost 90%, which is not bad at all for training purposes.

B. Support Vector Machine (SVM)

SVM is the next model that we will use. It is much more complex than KNN. The SVM uses a kernel function (specified as a hyperparameter) to decide how to draw lines between the classes of data points. Without a kernel, the classes must be linearly separable, but in most cases of real data, classes will likely not be linearly separable. The kernel decides the shape of the separated classes that it can create, and different kernels may better fit different datasets, depending on how the data is distributed. In this project, I test four different kernel functions, including Linear, Polynomial, RBF, and Sigmoid. Another hyperparameter, C, determines how lenient the model is when drawing these boundaries, or how much overlap it allows. Data will overlap more often than not, so allowing some overlap in separating the classes can help increase overall accuracy. Figure 11 shows the code to create one SVM model with fixed hyperparameters. Figure 12 shows a plot of the validation accuracy of the different kernels over increasing values of C.

```
svm = SVC(kernel='linear', C=1)
svm.fit(X_train, y_train)
svm.score(X_valid, y_valid)
```

Fig. 11. Creating an SVM model with the Linear kernel and $C = 1$.

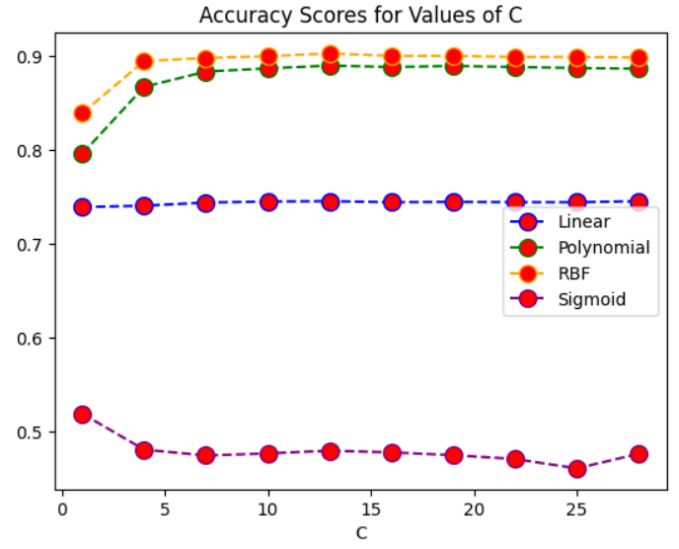

Fig. 12. Plot of validation accuracy of different kernels over increasing values of C.

From the Figure 12 plot, it is clear to see that the RBF kernel outperforms the others, with Polynomial close behind. For RBF, I found that the C value which gave the best validation accuracy is $C = 13$. This is the "sweet-spot" value of C for allowing enough overlap to find an optimal class separation, but not too much to reduce the accuracy. With these hyperparameters of using the RBF kernel with $C = 13$, a validation accuracy of just over 90% was achieved.

C. Neural Network (NN)

NN is the last model that will be used. It is the most complex model out of the three. It has many different hyperparameters to test, such as activation function, number of layers, number of neurons per layer, optimizer, etc. Testing all combinations of these hyperparameters would be extremely inefficient, so I will test one at a time and choose the best ones to use. Certain hyperparameters such as the activation functions and optimizer I will simply choose one without testing to save time and resources. For the activation functions of the hidden layers of the neural network, I will use the Relu function for the hidden layers, which is a popular choice due to its simplicity and training speed. For the output layer, I will use the softmax function, which works well with multiclassification since it calculates probabilities of a data point belonging to each class. I will also use the ADAM optimizer, which is another popular choice due to its efficiency and robustness.

As for the hyperparameters that I will test, there will be certain restrictions. For the number of neurons per layer, it is common to use powers of 2, and each layer has half as many neurons as the previous layer. In this case, I made it so the last hidden layer will have 64 neurons. Of course the output layer must have 10 neurons to match the 10 classes this problem has. Now for the number of hidden layers, even though theoretically more layers would give a better result, since each layer has twice as many neurons as the next layer, the running time to train a neural network will grow exponentially as more layers are added while causing small improvements to the accuracy. For this reason, I will only test between 2-5 hidden layers. An additional hyperparameter which was not previously mentioned is the use of Dropout, which ignores a given proportion of the training data in each epoch to help prevent overfitting. I will test a couple of different dropout values being 0 (no dropout), 0.2, and 0.5. While testing the various combinations of hyperparameters, I only trained the model with 50 epochs, which is on the low end to save time. A more accurate understanding of the effect of the hyperparameters could have been achieved using more epochs, but I needed to be efficient with this testing.

From testing these hyperparameters, I found that a neural network with 4 hidden layers and a dropout value of 0.2 performed the best in terms of validation accuracy. I ran these parameters for 500 epochs to obtain a properly trained neural network ready to be tested. Figure 13 shows the function I wrote to train a neural network. Figure 14 shows the training and validation accuracy over increasing epochs.

```
def trainNN(X_train, y_train, X_valid, y_valid, num_layers, dropout, epochs):
    model = Sequential()
    for i in range(num_layers):
        if i == 0:
            model.add(Dense(int(2**(5+num_layers)), activation='relu', input_shape=(X_train.shape[1],)))
        else:
            model.add(Dense(int(2**(5+num_layers-i)), activation='relu'))
        model.add(Dropout(dropout))
    model.add(Dense(10, activation='softmax'))
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    history = model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=epochs)
    return model, history
```

Fig. 13. Function for training a neural network with specified hyperparameters.

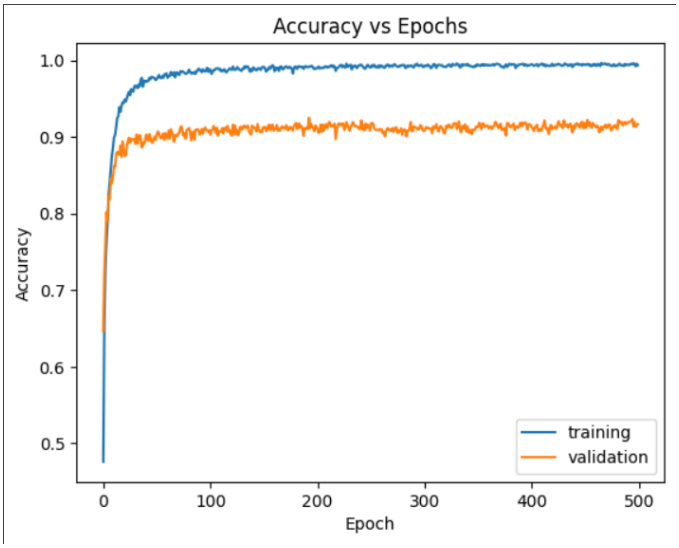

Fig. 14. Plot of training and validation accuracy over number of epochs.

With these epochs and hyperparameters, the model achieved a 92.5% validation accuracy, which is higher than the other two models. Now that we have found the best hyperparameters for each model and trained them, it is time to evaluate their performance on the testing data.

IV. Evaluation

THE metrics that will be used for evaluation purposes are the overall accuracy, F1 score (a combination of precision and recall), and confusion matrix for each model. These can be easily found with the classification_report and confusion_matrix functions from the metrics sublibrary of the sklearn library. The classification report also includes a support column, which means the number of samples for that genre that was used in testing.

A. K-Nearest Neighbors (KNN)

Here are the classification report and confusion matrix for the KNN model on the testing data, as shown in Figures 15 and 16 respectively. The classification report includes the accuracy, precision, recall, and F1 score as previously mentioned.

From these metrics, we can see that KNN performed well with classical, with the highest F1 score of 0.88. KNN performed the worst with rock, with the lowest F1 score of 0.64. The overall accuracy was 0.76, which although is much better than random guessing, is not the ideal percentage we are looking for. The KNN did not perform particularly well with any genre, since the highest being classical only had an F1 score of 0.88, which again is not bad, but for the best performing genre is not ideal. The result is interesting because the success of classical could be partially due to the support number being only 83, meaning that more classical samples were used in the training/validation process than expected. Since there are 1,000 samples of each genre, we expect around 100 samples of each genre in the testing subset. However, rock has exactly 100 samples in the testing set which is expected, yet it performed well below average. From the confusion

	precision	recall	f1-score	support
blues	0.89	0.74	0.81	115
classical	0.82	0.94	0.88	83
country	0.74	0.76	0.75	101
disco	0.58	0.79	0.67	99
hiphop	0.79	0.70	0.74	90
jazz	0.75	0.79	0.77	96
metal	0.81	0.84	0.82	104
pop	0.91	0.78	0.84	94
reggae	0.77	0.71	0.74	117
rock	0.66	0.61	0.64	100
accuracy			0.76	999
macro avg	0.77	0.77	0.76	999
weighted avg	0.77	0.76	0.76	999

Fig. 15. Classification report including precision, recall, F1 score, and accuracy for KNN.

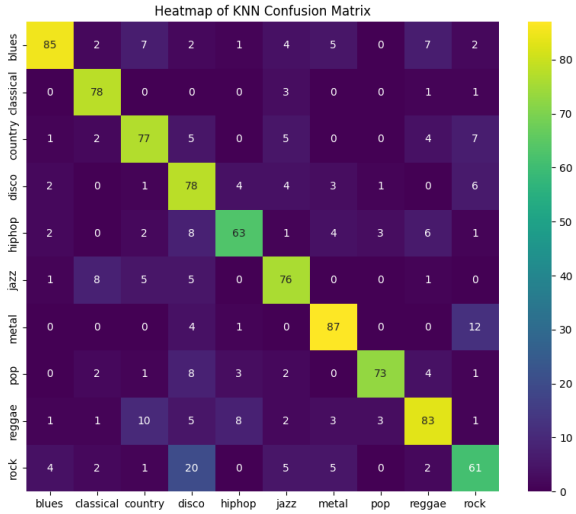

Fig. 16. Confusion matrix heatmap for KNN.

matrix, we can see that rock was often confused with disco and metal as well. Overall, it was expected for the accuracy of KNN to underperform, but it was interesting to see how much the F1 score varies between the best and worst performing genre.

B. Support Vector Machine (SVM)

Below is the classification report (Figure 17) and confusion matrix (Figure 18) for the SVM testing data.

Right away we can see a massive improvement in the SVM compared to the KNN. I expected the SVM to outperform KNN, but not to this degree. With an overall accuracy of 0.92, the SVM is clearly a very strong choice for tackling this classification problem. An accuracy of 0.92 was the goal that I had for at least one of the models to get, and SVM has succeeded. Above 0.90 accuracy is considered to be quite strong. With the SVM, we can see that metal performed the best with an F1 score of 0.96, and rock and disco tied for the worst with an F1 score of 0.88. Something very interesting about this is that rock is the worst performing genre for both KNN and SVM so far. We can see from the confusion matrix that rock and disco are confused occasionally, but overall all

	precision	recall	f1-score	support
blues	0.94	0.94	0.94	115
classical	0.89	0.95	0.92	83
country	0.92	0.90	0.91	101
disco	0.86	0.91	0.88	99
hiphop	0.92	0.92	0.92	90
jazz	0.83	0.93	0.88	96
metal	0.94	0.97	0.96	104
pop	0.97	0.90	0.93	94
reggae	0.97	0.91	0.94	117
rock	0.93	0.84	0.88	100
accuracy			0.92	999
macro avg	0.92	0.92	0.92	999
weighted avg	0.92	0.92	0.92	999

Fig. 17. Classification report including precision, recall, F1 score, and accuracy for SVM.

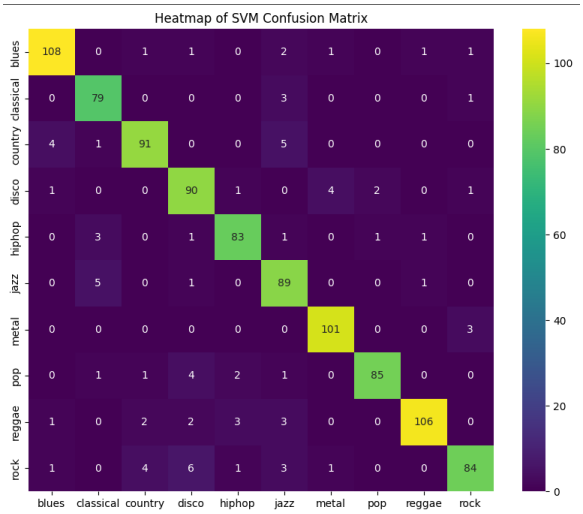

Fig. 18. Confusion matrix heatmap for SVM.

	precision	recall	f1-score	support
blues	0.96	0.92	0.94	115
classical	0.92	0.96	0.94	83
country	0.88	0.90	0.89	101
disco	0.88	0.91	0.90	99
hiphop	0.96	0.90	0.93	90
jazz	0.88	0.96	0.92	96
metal	0.89	0.97	0.93	104
pop	0.99	0.94	0.96	94
reggae	0.98	0.94	0.96	117
rock	0.91	0.85	0.88	100
accuracy			0.92	999
macro avg	0.93	0.93	0.92	999
weighted avg	0.93	0.92	0.93	999

Fig. 19. Classification report including precision, recall, F1 score, and accuracy for SVM.

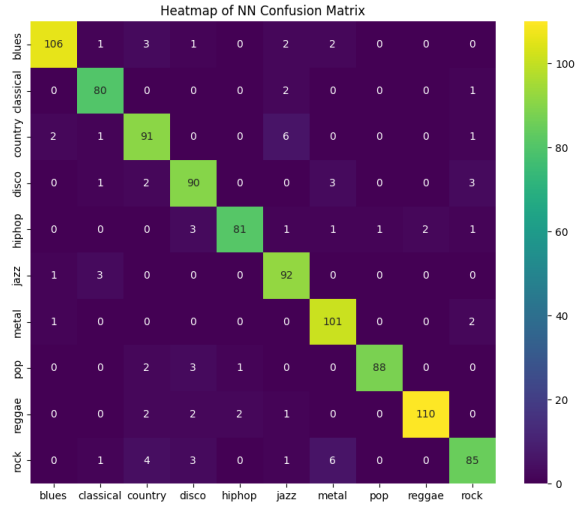

Fig. 20. Confusion matrix heatmap for SVM.

of the genres still performed well. From the confusion matrix, there are no big mistakes that stand out, and from looking at the F1 scores, the worst performing of 0.88 actually ties with the best performing genre in KNN. That alone proves how much better suited SVM is than KNN for this problem. Overall, I am very happy with the performance of the SVM, as it was easy to test and set up, and outperformed my expectations.

C. Neural Network (NN)

Finally, let's look at the testing data performance of the NN. This was the model that I had the highest hopes for. Below is the classification report (Figure 19) and confusion matrix (Figure 20) for the NN.

The results of the NN are about what I hoped for, with a high accuracy of 0.92 (macro avg and weighted avg 0.93). The F1 scores for the NN are the most consistent out of the 3 models. There is only one genre with the lowest F1 score of 0.88, which is surprisingly the rock genre again. The rock genre was consistently the worst performing genre for all 3 models, which is a very interesting result. Both pop and reggae are the highest scoring genres with an F1 score of 0.96. Another interesting observation is that none of the 3 models had the same best performing genre. From the NN confusion matrix, it looks similar to the SVM confusion matrix, in which there is no outstanding mistakes that the model makes. Technically, the NN slightly outperformed the SVM since the macro and weighted averages were higher. However, I found the NN much harder to test the hyperparameters and took much longer to fully train. Overall, the NN scored about as well as I expected.

V. Conclusion

In conclusion, the KNN and NN performed about as well as I expected, but SVM outperformed my expectations. I found it extremely interesting that rock was the genre that all models struggled with, but each model had a different genre that they excelled at. This was not expected, as I thought it would be somewhat random what each model does well and poorly in, based on the randomized splitting of data. However, the results were the same after randomizing the data and training the models numerous times. It could be related to the rock samples that the dataset provides, but in truth I am not entirely sure why this is the case. Overall, for this type of classification problem I would highly consider using SVM to

solve it due to the ease of building it and the speed of training, as well as its accuracy. Despite the NN performing slightly better, it took a long time to test the many combinations of hyperparameters and took a long time to fully train once I chose the hyperparameters.

VI. Moving Forward

UNFORTUNATELY I did not have time to implement visualizing the data that was incorrectly labeled by each model. If I had more time, this would be the first thing to do. This could provide an insight as to why the models struggled with rock. In addition, more models such as K-Means Clustering and Decision Trees or Random Forests could be tested on this same classification problem to compare their performances to the ones that I already tested. As for longer term use, perhaps other genres could be added to expand the capabilities of these models. Additionally, perhaps a similar model could be trained to identify a certain artist who created a given song, or even be able to identify the individual song like Shazam. Overall, this project was a great way for me to get familiar with using public datasets and the provided libraries in Python to train my own machine learning models.

References

[1] Blog about using a Convolutional Neural Network for this classification problem: https://www.clairvoyant.ai/blog/music-genre-classification-using-cnn

[2] GTZAN Dataset on Kaggle: https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification