Song Genre Classification

COMP 6630 Machine Learning

Emma Ingram
Auburn University
eci0004@auburn.edu

Sean O'Connor
Auburn University
sfo0002@auburn.edu

ABSTRACT

For our course project, we were tasked with identifying a multi-class classification problem where a solution could be determined via the implementation of a multi-layered perceptron (MLP). After reviewing a variety of potential problems we came to settle on exploring the problem of song classification by genre. Our desired goal was for a user to be able to input a song and the proper genre be populated by the software. Our algorithm would make this determination by extracting different features and making a prediction based on training data. With this implementation, we would be able to effectively eliminate the manual classification of songs into their respective genres, which proves to be a tedious task. Overall, our software trains a neural network using the GTZAN dataset and provides a user interface that prompts for a Waveform Audio File Format (.wav), extracts the features, and classifies the song into its proper genre. With this approach, we were able to implement this classifier and achieve a thirty percent (30%) accuracy rate.

CCS CONCEPTS

• **Computing methodologies → Supervised learning by classification**; **Feature selection**; **Neural networks**.

KEYWORDS

datasets, neural networks, genre classification, music, multi-layered perceptron

ACM Reference Format:
Emma Ingram and Sean O'Connor. 2022. Song Genre Classification: COMP 6630 Machine Learning. In *Proceedings of (COMP6630 Machine Learning)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Manual classification of songs into their proper genre can become tedious and time-consuming, especially if the song list is considerably lengthy. To address this issue, our project aims to eliminate this manual classification with the implementation of multi-layered perceptrons and song features to accommodate our primary application domain: music listeners, song companies, and people who enjoy genre-specific music.

When it comes to identifying who our primary application domain is we had to determine who our solution would be best suited for. After analyzing our problem statement, we found the users of our classifiers to be music listeners and more importantly music companies. There are multiple music companies, such as Apple Music, Spotify, and Pandora, who could benefit from the utilization of this neural network. These domains could find this classifier to be of importance for a variety of reasons:

- Lead to better population of suggested music
- Automate playlist creation based on genre
- Classifier could be re-implemented for sub-genre classification

For all of these reasons, our classifier would provide more help than harm for the music industry. Our report will dive deeper into our project approach, analysis, and final conclusions.

2 BASELINE MODELS

Before implementing our multi-layer perceptron, we explored a few baseline models, discussed below. These models allowed us to gauge the performance of our project to see where we matched up against other model implementations. To quickly implement these specific models, we utilized the library sklearn.

2.1 Support Vector Machine

For our first baseline model, we utilized a Support Vector Machine. Useful for both regression and classification, this model is a supervised machine learning algorithm. The number of input features works to determine the number of dimensions the hyperplane of this model will be divided into. For example, if the number of input features is 4, then the hyperplane becomes a three-dimensional plane. For our use case we are dealing with 10 input features (song length, spectral rolloff, root mean squared, tempo, etc.) which means we will have created a nine dimensional hyperplane. Using the SVC() class from the sklearn.svm package we split our data into a 80/20 training and testing split. After running this class, we achieved around a twenty-nine percent accuracy (28.5%) and a thirty-one percent testing accuracy (31.0%).The confusion matrix and classification model for this baseline model is illustrated on Table 1 and Table 2.

Genres	0	1	2	3	4	5	6	7	8	9
0	0	3	5	1	0	2	5	0	4	0
1	0	14	2	0	0	0	3	0	1	0
2	0	1	10	4	0	2	2	0	1	0
3	0	0	5	5	0	0	5	0	5	0
4	0	0	3	4	0	0	4	5	4	0
5	0	3	4	0	0	1	9	0	3	0
6	0	10	0	2	0	1	7	0	0	0
7	0	0	1	1	0	0	0	11	7	0
8	0	0	3	0	0	0	0	3	14	0
9	0	5	0	4	0	2	6	1	2	0

Table 1: Confusion Matrix for SVM

	Precision	Recall	F1-Score	Support
Blues	0.00	0.00	0.00	20
Classical	0.39	0.70	0.50	20
Country	0.30	0.50	0.38	20
Disco	0.24	0.25	0.24	20
Hip Hop	0.00	0.00	0.00	20
Jazz	0.12	0.05	0.07	20
Metal	0.17	0.35	0.23	20
Pop	0.55	0.55	0.55	20
Raggae	0.34	0.70	0.46	20
Rock	0.00	0.00	0.00	20

Table 2: Classification Model for SVM

2.2 Polynomial Kernel

In addition to our support vector machine as our baseline model, we began creating a variety of support vector machine kernel baseline models to gain a better understanding of our model implementation expectations. The first type of kernel we created a baseline model for is the polynomial kernel. In essence, a kernel takes data as an input and transforms it into a higher dimension so that a linear equation can be derived from the higher dimension space. In particular, the polynomial kernel illustrates the similarity of vectors in a feature space spanning polynomials of the original values used in the kernel. For a polynomial kernel of degree d, the equation can be written as:

$$K(x, y) = tanh(\gamma \cdot x^T y + r)^d, \gamma > 0$$

To differentiate between different types of kernels in our baseline model implementation, we added the parameter to our SVC() class previously discussed. Instead of just SVC() it became SVC(kernel='poly'). After running the program, the polynomial kernel performed with around a twenty-one percent (20.625%) training accuracy and around a nineteen percent (18.5%) testing accuracy. The confusion matrix and classification model for this baseline model is illustrated on Table 3 and Table 4.

2.3 Gaussian Kernel

Next in our kernel model for our baseline model analysis we implemented the Gaussian kernel. This kernel is useful because it allows for data transformation without any prior knowledge. The equation utilized for a

Genres	0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	2	16	0	1	0
1	0	0	0	0	0	1	19	0	0	0
2	0	0	0	0	0	1	18	0	1	0
3	1	0	0	2	1	2	14	0	0	0
4	0	0	0	2	2	0	11	3	2	0
5	0	0	0	0	0	1	17	2	0	0
6	0	0	0	0	0	0	20	0	0	0
7	1	0	1	1	5	1	3	7	1	0
8	2	0	2	0	2	4	3	3	4	0
9	1	0	0	0	1	1	17	0	0	0

Table 3: Confusion Matrix for Poly. Kernel

Genres	0	1	2	3	4	5	6	7	8	9
0	0	3	5	1	0	2	5	0	4	0
1	0	14	2	0	0	0	3	0	1	0
2	0	1	10	4	0	2	2	0	1	0
3	0	0	5	5	0	0	5	0	5	0
4	0	0	3	4	0	0	4	5	4	0
5	0	3	4	0	0	1	9	0	3	0
6	0	10	0	2	0	1	7	0	0	0
7	0	0	1	1	0	0	0	11	7	0
8	0	0	3	0	0	0	0	3	14	0
9	0	5	0	4	0	2	6	1	2	0

Table 5: Confusion Matrix for Gaussian Kernel

	Precision	Recall	F1-Score	Support
Blues	0.17	0.05	0.08	20
Classical	0.00	0.00	0.00	20
Country	0.00	0.00	0.00	20
Disco	0.40	0.10	0.16	20
Hip Hop	0.18	0.10	0.13	20
Jazz	0.08	0.05	0.06	20
Metal	0.14	1.00	0.25	20
Pop	0.47	0.35	0.40	20
Raggae	0.44	0.20	0.28	20
Rock	0.00	0.00	0.00	20

Table 4: Classification Model for Poly. Kernel

	Precision	Recall	F1-Score	Support
Blues	0.00	0.00	0.00	20
Classical	0.39	0.70	0.50	20
Country	0.30	0.50	0.38	20
Disco	0.24	0.25	0.24	20
Hip Hop	0.00	0.00	0.00	20
Jazz	0.12	0.05	0.07	20
Metal	0.17	0.35	0.23	20
Pop	0.55	0.55	0.55	20
Raggae	0.34	0.70	0.46	20
Rock	0.00	0.00	0.00	20

Table 6: Classification Model for Gaussian Kernel

Gaussian model can be illustrated as:

$$K(x,y) = e^{-\frac{||x-y||^2}{2\sigma^2}}$$

In terms of our code, we utilized a different equation because the function call we implemented was SVC(kernel='rbf') to better our transformation because it incorporates the radial basis function. The improvement of the transformation is shown by the following equation:

$$K(x,y) = e^{\gamma||x-y||^2}]$$

The Gaussian kernel, or radial basis function kernel, performed with around a twenty-nine percent (28.5%) training accuracy and a thirty-one percent (31.0%) testing accuracy. The confusion matrix and classification model for this baseline model is illustrated on Table 5 and Table 6.

2.4 Sigmoid Kernel

The final type of kernel we were able to use in creating our baseline was the Sigmoid kernel. Unlike the polynomial and Gaussian kernel, the Sigmoid kernel has the ability to be used as an activation function in neural networks. The equation for this kernel uses the hyperbolic tangent function and is shown below:

$$K(x,y) = tanh(\gamma \cdot x^T y + r)$$

As with the previous three baseline models, this model utilized to SVC() class with the adjustment of SVC(kernel='sigmoid') being applied for this specific implementation. At runtime completion the Sigmoid kernel performed with around a fifteen percent (15.375%) training accuracy and around a fifteen percent (14.5%) testing accuracy. The confusion matrix and classification model for this baseline model is illustrated on Table 7 and Table 8.

Genres	0	1	2	3	4	5	6	7	8	9
0	3	3	3	1	1	0	1	0	8	0
1	3	13	1	0	1	0	0	0	2	0
2	0	1	1	0	0	0	0	0	18	0
3	2	0	4	1	1	0	0	0	12	0
4	0	0	2	6	2	0	1	0	9	0
5	5	1	1	2	2	0	4	0	5	0
6	5	7	0	0	0	0	4	0	4	0
7	0	0	5	11	0	0	0	0	4	0
8	0	0	10	5	0	0	0	0	5	0
9	3	4	2	1	0	0	2	0	8	0

Table 7: Confusion Matrix for Sigmoid Kernel

	Precision	Recall	F1-Score	Support
Blues	0.14	0.15	0.15	20
Classical	0.45	0.65	0.53	20
Country	0.03	0.05	0.04	20
Disco	0.04	0.05	0.04	20
Hip Hop	0.29	0.10	0.15	20
Jazz	0.00	0.00	0.00	20
Metal	0.33	0.20	0.25	20
Pop	0.00	0.00	0.00	20
Raggae	0.07	0.25	0.11	20
Rock	0.00	0.00	0.00	20

Table 8: Classification Model for Sigmoid Kernel

2.5 K-Nearest Neighbor

As our last baseline model, we used sklearn to implement a K-Nearest Neighbor model. This model is a supervised learning model that does not require any prior parameters. It creates k categories of data based on a central point in the data to classify data. For our code, we used to KNeighborsClassifier() class to create this baseline model. After running our k-nearest neighbor implementation, we recorded a performance with around a fourty-four percent (44.25%) training accuracy and around a twenty-four percent (23.5%) testing accuracy. The confusion matrix and classification report for this baseline model is illustrated on Table 9 and Table 10.

Genres	0	1	2	3	4	5	6	7	8	9
0	4	1	4	1	2	4	3	0	0	1
1	5	9	2	0	0	1	3	0	0	0
2	4	1	5	6	0	2	0	0	1	1
3	2	0	5	5	2	0	0	0	4	2
4	0	0	1	5	2	0	1	7	2	2
5	3	2	5	0	2	6	0	1	0	1
6	4	4	2	1	1	4	3	0	1	0
7	0	0	1	1	8	0	0	7	3	0
8	2	0	1	6	2	0	0	4	5	0
9	5	0	2	2	1	2	4	2	1	1

Table 9: Confusion Matrix for KNN

	Precision	Recall	F1-Score	Support
Blues	0.14	0.20	0.16	20
Classical	0.53	0.45	0.49	20
Country	0.18	0.25	0.21	20
Disco	0.19	0.25	0.21	20
Hip Hop	0.10	0.10	0.10	20
Jazz	0.32	0.30	0.31	20
Metal	0.21	0.15	0.18	20
Pop	0.33	0.35	0.34	20
Raggae	0.29	0.25	0.27	20
Rock	0.12	0.05	0.07	20

Table 10: Classification Model for KNN

2.6 Sklearn's Multi-Layer Perceptron

To gain a conclusive understanding of how our model values could be reported as, we implemented a Multi-Layer Perceptron using the scikit-learn library to compare the accuracy with ours written from scratch. Specifically, we used the MLPClassifier() and StandardScaler() classes from this library. The implementation performed with almost a one hundred percent (99.67%) training accuracy and a sixty-five percent (65.0%) testing accuracy. This classifier performed the best of all the baseline models. The confusion matrix and classification model for this baseline model is illustrated on Table 11 and Table 12.

Genres	0	1	2	3	4	5	6	7	8	9
0	22	0	6	3	0	1	3	0	4	1
1	1	35	0	1	0	3	0	0	0	0
2	0	0	31	0	0	3	0	0	1	5
3	0	1	2	24	3	0	1	6	1	2
4	0	0	0	2	31	0	0	0	7	0
5	1	2	0	0	0	34	0	0	0	3
6	0	0	0	1	1	0	35	0	1	2
7	0	0	0	2	2	0	0	29	6	1
8	0	0	4	3	5	2	0	2	22	2
9	2	0	6	9	2	2	1	2	1	15

Table 11: Confusion Matrix for Sklearn MLP

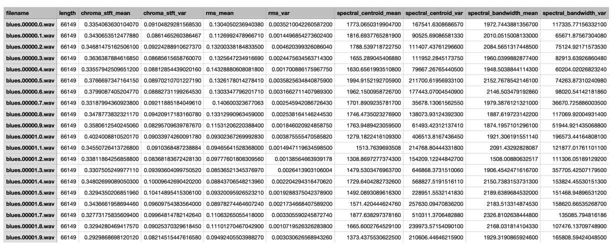
	Precision	Recall	F1-Score	Support
Blues	0.85	0.55	0.67	40
Classical	0.92	0.88	0.90	40
Country	0.63	0.78	0.70	40
Disco	0.53	0.60	0.56	40
Hip Hop	0.70	0.78	0.74	40
Jazz	0.76	0.85	0.80	40
Metal	0.88	0.88	0.88	40
Pop	0.74	0.72	0.73	40
Raggae	0.51	0.55	0.53	40
Rock	0.48	0.38	0.42	40

Table 12: Classification Model for SkLearn MLP

3 DATA

3.1 GTZAN

The data that we used comes from the GTZAN dataset, which is the most-used public dataset for evaluation in machine listening research for music genre recognition. It is a collection of one thousand, thirty-second songs split into ten genres - blues, classical, country, disco, hip-hop, jazz, metal, pop, reggae, and rock. It also contains 2 Comma Separated Values (.csv) files that contain features of the audio files. One of the .csv files contains a mean and variance computed over multiple features for each song. The other .csv file has the same structure but the songs were split into 3 second audio files, increasing the amount of data by a factor of ten. We used this dataset as-is for our project. We split it into training and testing data using an 80/20 split. An image of part of the contents in the .csv file along with how they appear is shown below in Figure 1



filename	length	chroma_stft_mean	chroma_stft_var	rms_mean	rms_var	spectral_centroid_mean	spectral_centroid_var	spectral_bandwidth_mean	spectral_bandwidth_var
blues.00000.0.wav	66149	0.3354063630104070	0.0910482928156853 0	0.1304050236940380	0.0035210042260587200	1773.0650319904700	167541.6308686570	1972.7443881356700	117335.7715633210 0
blues.00000.1.wav	66149	0.343065351247788 0	0.0861465260386467	0.1126992478966710	0.0014496854273602400	1816.6937765281900	90525.69086581330	2010.0515008133000	65671.87567304080
blues.00000.2.wav	66149	0.34681475162506100	0.09224288910627370	0.13200338184833500	0.00462039932608604 0	1788.539718722750	111407.43761296600	2084.5651317448500	75124.92171573530
blues.00000.3.wav	66149	0.3636387884616850	0.08685615658760070	0.1325647234916690	0.0024475634563714300	1655.289045406880	111952.2845173750	1960.0399882877400	82913.63926860480
blues.00000.4.wav	66149	0.33557942509651200	0.08812854439020160	0.14328880608081800	0.00170088617596775 0	1630.6561993510800	79667.26765440500	1948.5038844114300	60204.02026823240
blues.00000.5.wav	66149	0.3766697347164150	0.08970210701227190	0.1326178014278410	0.0035825634840875900	1994.9152192705900	211700.61956933100	2152.7678542146100	74263.87310240980
blues.00000.6.wav	66149	0.3799087405204770	0.08882731199264530	0.1303347796201710	0.0031662711407989300	1962.1500958726700	177443.07004540900	2146.503479192860	98020.54142181860
blues.00000.7.wav	66149	0.33187994360923800	0.09211885184049610	0.140600323677063	0.00254594208672643 0	1701.8909235781700	35678.13061562550	1979.3876121321000	36670.725886003500
blues.00000.8.wav	66149	0.3478773832321170	0.09420917183160780	0.13312990963459000	0.00253816414624453 0	1746.4735023276900	138073.93124392300	1887.619723142200	117069.92004931400
blues.00000.9.wav	66149	0.3580612540245060	0.08295709639787670	0.11531206220388400	0.00184602092485875 0	1763.9489423059500	61493.42312137410	1874.1957101296100	51944.921435068800
blues.00001.0.wav	66149	0.4024008810520170	0.09033974260091780	0.09302367269992830	0.00387555547058582 0	1279.1822416109300	406513.8167436450	1921.306191551140	196573.44164808100
blues.00001.1.wav	66149	0.34550726413726800	0.0910368487238884	0.09465641528368000	0.0014947119634598500	1513.7639693508	214768.80444331800	2091.43292828087	121877.01761101100
blues.00001.2.wav	66149	0.33811864256858800	0.08368183672428130	0.09777601808309560	0.00138564663939178	1308.8697277374300	154209.12244842700	1508.00880632517	111306.05189129200
blues.00001.3.wav	66149	0.3307505249977110	0.09393604099750520	0.08536521345376970	0.0026413903106004	1479.5303476963700	646868.3731510060	1906.4542471616700	357705.42507179500
blues.00001.4.wav	66149	0.34802699089050300	0.10009642690420200	0.08843706548213960	0.00220429431647062 0	1729.6404282732600	568827.5191516110	2150.7383153731300	153824.45530151300
blues.00001.5.wav	66149	0.3294350206851960	0.10414895415306100	0.09320095926523210	0.0019288375042378900	1492.0893089616300	228951.5532141830	2189.6389684532000	151468.94896531200
blues.00001.6.wav	66149	0.3436661958694460	0.09609754383564000	0.08978274464607240	0.0021734668407589200	1571.420444624760	257630.09470836200	2183.513314874530	158620.66535268700
blues.00001.7.wav	66149	0.32773175835609400	0.09964814782142640	0.11063265055418000	0.00330559024587274 0	1877.638297378160	510311.3706482880	2326.8102638444800	135085.794816186
blues.00001.8.wav	66149	0.3294280469417570	0.09025370329618450	0.11101270467042900	0.0010719526326283800	1665.6002764529100	239973.57154090100	2168.031814104330	107476.13709748900
blues.00001.9.wav	66149	0.2929868698120120	0.08214515447616580	0.09492405503988270	0.00303062656894326 0	1373.4375530622500	210606.44646215900	1929.3190865924600	165808.59424048500

Figure 1: Song Feature Initial Data File

3.2 Feature Extraction

We analyzed a variety of features that we extracted from the Waveform Audio File Format (.wav) files. To execute this, we used the Librosa python library and several of its function calls to calculate the proper values. We will go more in depth about our implementation of the Librosa library in our code walk through section of the report. The features we extracted along with their definitions include:

- Length of the Song
  - For our purposes, the length is the number of frames in the .wav file; however, to find the length of the file in seconds we must take the number of frames divided by the sample rate.
- Chroma Frequencies
  - Is a representation of an audio file split into the twelve categories of semitones within the musical octave.
- Root Mean Squared
  - Measures the average loudness of a sound within a certain time interval.
- Spectral Centroid
  - The spectral centroid takes the weighted mean of the culmination of frequencies within the sound. With this value, the center of mass (or centroid) of the sound can be found.
- Spectral Bandwidth
  - Spectral Bandwidth is the width of the band in the audio at the halfway point of a peak.
- Spectral Rolloff
  - Can be categorized as the shape of the sound. This value is the frequency below which a specified percentage of the total spectral energy.
- Zero-Crossing Rate
  - This can be found by the rate at which the signal from the audio file changes from positive to negative or vice versa.

- Harmony
  - Can be described as simultaneously occurring frequencies, pitches, or chords. Is said to represent the sounds color.
- Perceptual
  - Is an understanding of representations of shock waves that can be translated to the sound rhythm and emotion.
- Tempo
  - Characterizes the beats per minute. Tempo can also be illustrated as the speed or pace that music should be played.
- Mel-Frequency Cepstral Coefficients (MFCC's)
  - A combination of frequencies that together form the shape of the spectral envelope. Can be found by taking a derivation of the Fourier transformation.

4 MULTI-LAYER PERCEPTRON

To implement the multi-layer perceptron, we must first apply some pre-processing techniques to the data. We need to normalize the data, one-hot encode the data, and split it into training and testing data.

To normalize the data, we used min-max normalization. For every feature, the minimum value of that feature gets transformed into a 0, the maximum value gets transformed into a 1, and every other value gets transformed into a decimal between 0 and 1. This is done using the following equation:

$$\frac{value - min}{max - min}$$

After normalizing the data, we then applied one-hot encoding because our data labels are categorical. The target data labels are strings representing the genre in which the input data lies. For example, the label in the target data for a blues song would be 'blues'. Because this data is in categories, we needed a numerical representation for the neural network to interpret. We applied a mapping to the genres as follows:

- 'blues' $\longrightarrow$ 0
- 'classical' $\longrightarrow$ 1
- 'country' $\longrightarrow$ 2
- 'disco' $\longrightarrow$ 3
- 'hiphop' $\longrightarrow$ 4
- 'jazz' $\longrightarrow$ 5
- 'metal' $\longrightarrow$ 6
- 'pop' $\longrightarrow$ 7
- 'reggae' $\longrightarrow$ 8
- 'rock' $\longrightarrow$ 9

After normalizing and encoding the data, we can then split it into training and testing data. We chose to use an eighty-twenty split, meaning eighty percent (80%) of the data is used for training the classifier, while the other twenty percent (20%) is used for validating the classifier, or testing it. Because the data is in an order such that the first one hundred lines of the .csv feature file are songs from the 'blues' category, the next one hundred lines are songs from the 'classical' category, and so on, we shuffle the data before applying this train-test split.

The next step in applying a multi-layer perceptron to our problem is to initialize the weights used in the perceptron. We found that initializing the weights using Xavier weight initialization performed better than a completely random initialization. The Xavier initialization is calculated as a random number with a uniform probability distribution. We implemented the Xavier weight distribution using the following calculation:

$$weight = U[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$$

where U is a uniform probability distribution and n is the number of features, which in our case is fifty-eight. This initialization is used for both the hidden layer and the output layer.

Then, to train the classifier we first perform a forward pass through the hidden and output layer and then a backward pass through the output layer then the hidden layer. The forward pass uses the current weights to make a prediction. The input to the hidden layer is calculated by taking the dot product of the input data and weight W and adding bias w_0, as shown below:

$$h_i = X \cdot W + w_0$$

This input is then fed into the activation function for the hidden layer, which is the sigmoid function. This function uses the following equation:

$$h_o = \frac{1}{1 + e^{-h_i}}$$

Using the output from the hidden layer, we then calculate the input to the output layer by taking the dot product of the output of the hidden layer and weight V and adding bias v_0, as shown below:

$$o_i = h_o \cdot V + v_0$$

Now, we can make a prediction by passing the input of the output layer into the output layer's activation function, softmax, which uses the following equation:

$$\sigma(z_i) = \frac{e^{z_i}}{\Sigma_{j=1}^{K} e^{z_j}}$$

for i = 1,...,K and z = $(z_1,...,z_K)$ ϵ $\mathbb{R}^K$.

The result of passing o_i into the softmax function is the classifier's prediction for that given input. This completes the forward pass of the classifier.

To continue training the perceptron and improve its accuracy, a backward pass is then performed and the weights are updated accordingly. This is the backpropagation. First, the gradient descent with respect to the input of the output layer is calculated. This uses the gradient of the loss function, which is:

$$loss_{gradient} = -\frac{y_a}{y_p} + \frac{1 - y_a}{1 - y_p}$$

where y_a is the actual values of y and y_p is the predicted y values. This value is then multiplied by the gradient of the softmax function of the input of the ouptut layer, o_i, shown below:

$$softmax_{gradient} = softmax(o_i) * (1 - softmax(o_i))$$

Putting this together, we get the gradient descent with respect to the input of the output layer with the following equation:

$$grad_{o_i} = loss_{gradient}(y_a, y_p) * sigmoid_{gradient}(h_i)$$

We then calculate the gradient descent for the weights used in the output layer, V and v_0 using the following two equations:

$$grad_V = h_o^T \cdot grad_{o_i}$$

$$grad_{v_0} = \Sigma grad_{o_i}$$

Using these values, we can update the weights using the gradient descent as follows: V = V - learning_rate * $grad_V$ v_0 = v_0 - learning_rate * $grad_{v_0}$

After this we perform similar calculations to update the weights used in the hidden layer. We start by calculating the gradient descent with respect to the input of the hidden layer. In order to do this, we must first calculate the dot product of the gradient descent with respect to the input of the out put layer and the transpose of V, the weight used in the output layer. This is expressed with the following equation:

$$grad_{h_i} = grad_{o_i} \cdot V^T * sigmoid_gradient(h_i)$$

where the gradient of the sigmoid is expressed as:

$$sigmoid_{gradient} = sigmoid(h_i) * (1 - sigmoid(h_i))$$

Using these values, we can calculate the gradient descent for the weights uesed in the hidden layer, W and w_o, as shown below:

$$grad_W = X^T \cdot grad_{h_i}$$

$$grad_{w_0} = \Sigma grad_{h_i}$$

Now, we are able to update the weights as follows: W = W - learning_rate * $grad_W$ w_0 = w_0 - learning_rate * $grad_{w_0}$

This process, the forward pass and backward pass, are then repeated for each epoch to produce the multilayer perceptron. To predict the genre of a new input, we simply perform a forward pass through the hidden layer and output layer with the trained weights and return that value as the prediction.

5 IMPLEMENTATION

5.1 Tools

To implement our multi-layer perceptron, we used the Python language. We also used multiple libraries supported by Python to aid in our implementation. These libraries include:

- NumPy
  - NumPy is a library for Python that offers large, multi-dimensional arrays and matrices, as well as a large collection of high-level mathematical functions on operate on these arrays.
- Matplotlib
  - Matplotlib is a plotting library for Python that works alongside NumPy.
- Pandas
  - Pandas is a library for Python that is used for data manipulation and analysis. It offers data structures and operations for manipulating numerical tables and time series.
- math
  - `math` is a built-in module in Python. It allows the user to perform various mathematical calculations, such as numeric, trigonometric, logarithmic, and exponential calculations.
- Librosa

  - Librosa is a python package for music and audio analysis. It provides the building blocks necessary to create music information retrieval systems.

We used NumPy to store the data and perform manipulations such as dot products on the data. We used Matplotlib to plot accuracy graphs given varying values for the learning rate, number of epochs, and number of neurons in the hidden layer. We used Pandas to read in the data from the .csv file into a dataframe. We used Python's math library to perform square roots and other mathematical operations on our data. Lastly, we used Librosa to extract features from the .wav files.

5.2 Code

In our Python implementation of the multi-layer perception, we first defined some helper functions. These include:

- `accuracy_score(y_true, y_pred)`
  - This function calculates the accuracy of the classifier, given the actual values for `y` and the predicted values for `y`.
- `normalize(x)`
  - This function normalizes the values of `x` using min-max normalization.
- `to_categorical(x, n_col)`
  - This function performs the one-hot encoding on the data `x`.
- `sigmoid(x)`
  - This function calculates the sigmoid of the input data `x`.
- `sigmoid_gradient(x)`
  - This function calculates the gradient of the sigmoid of the input data `x`.
- `softmax(x)`
  - This function calculates the softmax of the input data `x`.
- `softmax_gradient(x)`
  - This function calculates the gradient of the softmax of the input data `x`.
- `loss(y_true, y_pred)`
  - This function calculates the loss given the actual values for `y` and the predicted values for `y`.
- `loss_gradient(y_true, y_pred)`
  - This function calculates the gradient of the loss function given the actual values for `y` and the predicted values for `y`.

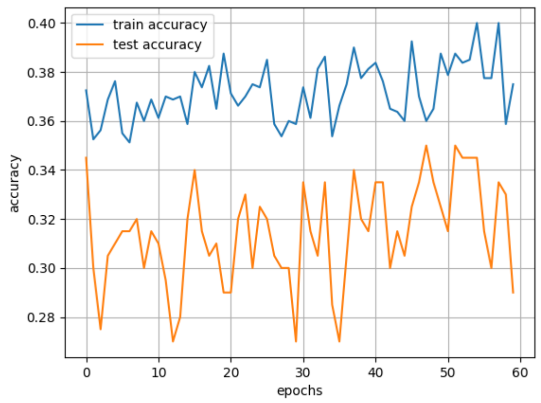


Figure 2: Number of Epochs

- `get_genre(songs)`
  - This function returns the string representation of music genres, given an array of integers representing genres of music.

The training data is read into a data frame from a .csv file using Pandas. Then the pre-processing discussed in the previous section is applied to this data, where it ultimately ends up stored in a NumPy array.

We then initialize our hyper-parameters, which include the number of neurons in the hidden layer, the number of epochs used to train the classifier, and the learning rate. We performed some tests by plotting the accuracy of our classifier with varying values for each of these hyper-parameters to decide what values to use for them. We found the optimal values to be 15 neurons in the hidden layer, 9400 epochs, and a learning rate of 0.01. Figure 2 below shows the training accuracy and testing accuracy for `n_hidden = 15`, `learning_rate = 0.01`, and varying values of `n_epochs`. We also initialize our weights using the aforementioned Xavier weight initialization method.

Then we create our `fit(X, y, weights` function that trains the weights for the classifier. This portion of the code uses the equations mentioned in the previous section. For each epoch, we perform a forward and backward pass through the hidden and output layers and update our weights according to the backpropagation calculations. To make a prediction using the trained classifier, it simply performs a forward pass over the input data and outputs the result.

We also created an interface for the user to use. This interface prompts for a song file (.wav), reads in the file and extracts the features. To extract the features from the .wav file, the program uses our `wav_extract(wav_in)` function in `wav_feature_extract.py`. This function reads in the .wav file using Librosa's load function, trims the leading and trailing silence of the audio, and stores the remainder in a variable called `audio_file`. Using this floating point time series with Librosa methods, we then extract the features mentioned in Section 3.2. All fifty-eight features are stored in a NumPy array which is then returned by the function. This data is then ready to be processed for our classifier. Back in `mlp.py`, the program normalizes this feature data, uses the classifier to make a prediction, and outputs the string representation of the prediction. This program continues to prompt for songs until the user inputs 'n' at the prompt.

Algorithm 1 is the general algorithm implemented by our code.

Algorithm 1 Multi-Layer Perceptron

```
1: Normalize data
2: One-hot encode data
3: Split data into train and test
4: Initialize weights
5: for each epoch do
6:     Forward pass on hidden layer
7:     Forward pass on output layer
8:     Backward pass on output layer
9:     Backward pass on hidden layer
10:    Update weights
11: end for
12: Predict (forward pass)
```

5.3 Testing

When classifying the training data, we found our multi-layer perceptron to perform with an accuracy of thirty-one percent (31%).

We tested our classifier using twenty percent (20%) of the data from the GTZAN dataset. This was a random subset of the data. When classifying this validation data, we found our multi-layer perceptron to perform with a twenty-nine percent (29%) accuracy. We found that utilizing a confusion matrix and a classification report produced the best understanding when analyzing our testing results. A confusion matrix is useful because it defines the performance of our classifier by providing a visualization of our results in terms of different evaluation metrics such as recall, precision, and accuracy. A classification report is useful because it gives a numerical representation for each genre of music given to the classifier. The numerical values are representative of our classifier's precision, recall, and F1-Score. The definition of each of these evaluation metrics is provided below.

- Precision: determines how precise a model is by quantifying the correct number of positive predictions made. The formula for Precision is listed below:

$$Precision = \frac{\#ofTruePositives}{\#ofTruePositives + \#ofFalsePositives}$$

- Recall: can be defined as the ratio between the number of positive samples correctly classified and the total number of positive examples. The equation for Recall can be illustrated as:

$$Recall = \frac{\#ofTruePositives}{\#ofTruePositives + \#ofFalseNegatives}$$

- Accuracy: is the fraction of predictions our model generated correctly. The equation for accuracy is:

$$Accuracy = \frac{\#ofcorrectpredictions}{\#oftotalpredictions}$$

- F1-Score: can be defined as the harmonic mean of precision and recall within a model. The equation for F1 score is:

$$F1score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Our classification report and confusion matrix for our Multi-Layered Perceptron is displayed in Table 13 and Table 14. It is evident that our classifier struggled the most with predicting a song of Hip Hop and Rock; however, our classifier proved to have the most success when correctly predicting a Pop song.

Our accuracy was lower than our goal and lower than sklearn's implementation of a multi-layer perceptron using the same dataset. We believe this to be because of multiple challenges faced when implementing our perceptron. We discuss these challenges in detail later in the paper.

Genres	0	1	2	3	4	5	6	7	8	9
0	1	0	4	1	0	0	11	0	1	0
1	1	8	0	1	0	0	4	0	0	0
2	3	1	1	6	0	0	1	1	2	0
3	0	0	4	7	2	0	1	4	2	0
4	1	0	0	3	0	0	3	9	3	1
5	1	3	6	1	0	1	6	0	2	0
6	0	4	0	3	2	0	13	0	0	0
7	0	0	1	1	2	0	0	17	3	0
8	0	0	1	1	1	0	0	6	14	0
9	0	0	4	3	0	0	8	3	6	0

Table 13: Confusion Matrix for MLP

	Precision	Recall	F1-Score	Support
Blues	0.14	0.05	0.08	19
Classical	0.50	0.57	0.53	14
Country	0.05	0.07	0.06	15
Disco	0.26	0.35	0.30	20
Hip Hop	0.00	0.00	0.00	20
Jazz	1.00	0.05	0.10	20
Metal	0.28	0.59	0.38	22
Pop	0.41	0.70	0.52	23
Raggae	0.41	0.61	0.49	23
Rock	0.00	0.00	0.00	24

Table 14: Classification Model for MLP

6 INSTALLATION AND SETUP

In this section, we will walk through how a new user can access and run our project via the remote repository GitHub with an Apache 2.0 license. As previously stated, the libraries we used for our project are Numpy, Pandas, Sklearn, Matplotlib, GitHub, and Librosa. We will assume the user has at least Python 3.9 installed on their respective machines.

6.1 Installing on Macintosh

For Mac users, please open terminal and follow the instructions below to properly install our project on your machine.

6.1.1 Installing Required Libraries. There are a multitude of libraries needing installation for the project to function properly. They are:

- NumPy
- Pandas
- Matplotlib
- Librosa
- GitHub
- SkLearn

We will utilize the 'pip' or 'pip3' command to install these libraries. For Git, we will use the package manager homebrew to install. If you do not have homebrew on your machine, refer to the following website to install Homebrew Installation. Please enter the following commands to successfully install on your machine.

(1) pip3 install numpy
(2) pip3 install pandas
(3) pip3 install matplotlib
(4) pip3 install librosa
(5) pip3 install git
(6) brew install sklearn

Once you have run these commands, we are now ready to clone our remote repository.

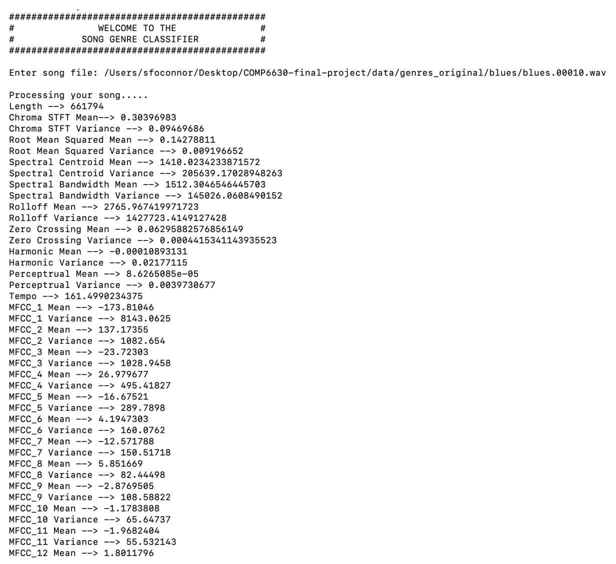
6.1.2 Cloning GitHub Remote Repository. Assuming you have successfully installed git on your local machine, we are ready to now clone the remote repository.

(1) Via Terminal, navigate to your desired directory to place repository folder
(2) Run the following command:
    - git clone https://github.com/emmaingram/COMP6630-final-project.git
(3) Installation should begin and at completion ensure all files were pulled properly. You should ensure you have the following python and data files/folders:
    - data folder containing .csv, .wav, and image files
    - baseline_metrics.py
    - mlp.py
    - wav_feat_extract.py

6.1.3 Running the Remote Repository. Finally, you should have all components installed and pulled that are required to run our software. To run our project, you will start with the mlp.py file. Run the following command:

- python3 mlp.py

Afterwards, you should be prompted to enter an audio file for genre prediction. Any .wav file should be compatible; however, you can use some of the provided .wav files in the data folder for testing. The interface after inputting a Waveform Audio File Format file should resemble Figure 3: Interface for mlp.py



```
##############################################
#               WELCOME TO THE               #
#            SONG GENRE CLASSIFIER           #
##############################################

Enter song file: /Users/sfoconnor/Desktop/COMP6630-final-project/data/genres_original/blues/blues.00010.wav

Processing your song.....
Length --> 661794
Chroma STFT Mean--> 0.30396983
Chroma STFT Variance --> 0.09469686
Root Mean Squared Mean --> 0.14278811
Root Mean Squared Variance --> 0.009196652
Spectral Centroid Mean --> 1410.0234233871572
Spectral Centroid Variance --> 205639.17028948263
Spectral Bandwidth Mean --> 1512.3046546445703
Spectral Bandwidth Variance --> 145026.0608490152
Rolloff Mean --> 2765.967419971723
Rolloff Variance --> 1427723.4149127428
Zero Crossing Mean --> 0.06295882576856149
Zero Crossing Variance --> 0.0004415341143935523
Harmonic Mean --> -0.00010893131
Harmonic Variance --> 0.02177115
Perceptrual Mean --> 8.6265085e-05
Perceptrual Variance --> 0.0039730677
Tempo --> 161.4990234375
MFCC_1 Mean --> -173.81046
MFCC_1 Variance --> 8143.0625
MFCC_2 Mean --> 137.17355
MFCC_2 Variance --> 1082.654
MFCC_3 Mean --> -23.72303
MFCC_3 Variance --> 1028.9458
MFCC_4 Mean --> 26.979677
MFCC_4 Variance --> 495.41827
MFCC_5 Mean --> -16.67521
MFCC_5 Variance --> 289.7898
MFCC_6 Mean --> 4.1947303
MFCC_6 Variance --> 160.0762
MFCC_7 Mean --> -12.571788
MFCC_7 Variance --> 150.51718
MFCC_8 Mean --> 5.851669
MFCC_8 Variance --> 82.44498
MFCC_9 Mean --> -2.8769505
MFCC_9 Variance --> 108.58822
MFCC_10 Mean --> -1.1783808
MFCC_10 Variance --> 65.64737
MFCC_11 Mean --> -1.9682404
MFCC_11 Variance --> 55.532143
MFCC_12 Mean --> 1.8011796
```

Figure 3: Interface For mlp.py

6.2 Installing on Windows

If you are using a Windows Operating System, please follow the following steps outlined in the Installing on Macintosh section. All steps are the same if being ran in the Command Prompt on your Windows Oporating System.

6.3 Other Installation Notes

(1) The python library, Librosa, is not compatible with Python 3.11. If you are running this version, you will have to install an older version (> 3.6 and < 3.11).
(2) Some systems require a call of python instead of python3 or pip instead of pip3.
(3) If you do not have Git installed on your Windows machine, refer to Git Installation
(4) If you do not have Python installed on your machine, refer to Python Installation

7 CHALLENGES

When determining challenges of our Multi-Layered Perceptron implementation for song-genre classification we can split them into two categories: expected challenges and actual challenges. Some of our expected challenges we did encounter; however, they were not as difficult or tedious as the unexpected challenges and problems we encountered during development. In the following subsections we will go into further detail of the variety of challenges we expected and actually encountered.

7.1 Expected Challenges (Pre-Implementation)

Prior to beginning the implementation of this project, there were several anticipated challenges we planned to encounter. First, we planned on encountering issues with the processing of the Waveform Audio File Format (.wav) because we had no prior experience with handling this type of file format. Next, we knew that regional differences in genre-classification could produce difficulties in determining the genre of a song with its correlating input features. This means that a song deemed as Pop in the United States could be categorized as a Rock song in another country such as Portugal. In addition, songs with overlapping genres within the same region could become difficult to categorize. For example, it may become difficult to categorize a Hip-Hop song compared to R&B because it could become relatively easy for the Hip-Hop song to be susceptible to misclassification as a R&B song. In general, we encountered every challenge that we expected; however, the extent of the difficulty in handling these challenges could be deemed as different than what we expected.

7.2 Actual Challenges (Post-Implementation)

Once we concluded our implementation of the song genre classifier via Multi-Layered Perceptrons, there were new problems created that we did not foresee. Primarily, the implementation of the Multi-Layered Perceptron itself proved to be the most difficult aspect of this project. In addition, it is a very viable possibility that this challenge hindered the final accuracy of our model. Some of the types of questions raised as a result of this challenge are as follows:

- What activation function should we use?
- What is the best approach to utilize in data normalization?
- How do we mathematically implement the back-propogation algorithm?
- How to we accurately implement hot coding for data manipulation?
- Which features should we use for our input layer?
  - Is it optimal to include all features?

- Should we just use MFCC's?
- Is mean and variance differentiable enough between song genres?
- Which input features are not beneficial?

It is clear to see there are a lot of issues we had to approach when it came to Multi-Layered Perceptron implementation since we were not able to utilize outside libraries such as Sklearn. This was definitely the most time consuming and difficult aspect of out project. In terms of other challenges, the feature extraction for the input layer from the .wav file was not as challenging as previously anticipated. For testing and training, those values came directly from the previously described .csv file. When it came to actual implementation and giving out algorithm a .wav file, we were able to use the python library, Librosa, and a couple other calculations to find these values. The regional differences between genres and the general overlapping of song genre is an issue we believe contributed to the unsuccessfulness of our algorithm implementation. The use of only mean and variance probably created a data set that was not differentiable enough between genres; therefore, our neural network encountered accuracy issues. Overall, we were able to accommodate and find solutions for most of these challenges; however, that is not to say that these issues did not contribute to the inaccurate predictions created when testing our algorithm.

8 ANALYSIS, CONCLUSIONS, AND IMPROVEMENTS

In conclusion, we were successfully able to implement a Multi-Layered Perceptron. We found that, for our implementation, the sigmoid activation function generated a better accuracy than the Rectified Linear Unit (ReLu) activation function. In addition, we also determined that adding additional hidden layers in our neural network does not always infer a more accurate model. In fact, when adding too many hidden layers we actually noticed our model accuracy decreased. To improve this model, there are a variety of additions and aspects that could be taken into consideration. We could use more input layers to try and improve the accuracy of our model. Currently, we have fifty-eight features in the input layer which are composed of mean and variance values previously outlined. To generate more values, we could begin to use individual values form each respective feature instead of wrapping it up into a mean and variance value. This could potentially make it easier for our model to differentiate between each genre. In addition to using values provided in the .csv file, we could utilize the spectrograms that are provided in the data set. By adding these charts and visualizations for each song file we could provide more information to allow for our model to yield a higher accuracy. Overall, we were able to generate a Multi-Layer Perceptron as a solution; however, our reported accuracy was relatively low and there is plenty of room for improvement. The link for out project was provided previously in the Installation section; however, you can also follow the link provided here: Ingram & O'Connor COMP 6630 Project. Our code is located in a GitHub remote repository under an Apache open source license. This project was very fulfilling and provided great learning experiences in understanding neural networks and how they can be applied to achieve solutions to modern day issues.