October 21, 2018

0.1 Keras neural network implementation for music genre classification

0.2 Abstract

In the rapidly growing and developing field of artificial intelligence, music classification has been present and growing for years. With companies like Spotify and Shazam making headlines for their state of the art algorithms for classifying music far beyond genre, the problem of what makes music differ from other kinds of music, has been presented from several different points of view. This report aims to cover a simple implementation for classifying music by genre by using the high-level neural network API of Keras to make the implementation easy to understand. Processing the data from actual music to numbers is not covered in the implementation but is touched through literature. The emphasis of the implementation is on simplicity and steps are covered from feature reduction to labeling the final data. Results achieved vary around 65% accuracy and logarithmic loss of around 0.18.

0.3 1. Introduction

In recent years, the study of deep learning and especially its applications in music classification has been a popular topic for researchers, with the amount of papers published related to the topic skyrocketing exponentially. (Bayle, 2018) Comparisons between traditional music recommendation systems and those using convolutional neural networks (CNNs) have shown that CNNs are indeed superior, especially in the context of new and unpopular songs. (Oord et al., 2014) Many papers covering the topic, understandably, cover it in a way that is difficult to understand for someone not familiar with the field. For example, talking about specific algorithms that are unfamiliar to the layman. The problem with classifying music is the music itself. What makes music a specific genre and more importantly, how can you measure it? A single 3 minute song also has way too much information in it, so classification should be possible from a single snapshot that's, for example, ten seconds long. (Despois, 2016) The problem with that, however, is that song can vary a lot in rhythm and melody, or in other words, the defining characteristics of a genre can vary between a moment in a song and another moment in the same song. With so many nyanses to consider, music classification and indeed even machine learning itself may become somewhat frightening for many. That is why this paper aims to produce a simple, easy-to-understand and "good enough" solution to the problem by the means of the high-level neural network API Keras. The learning objective of the project is the same - to learn to implement a simple, working, neural network that does what it is supposed to do. The ultimate research-question for this project is "How easy is it to implement a neural network to classify (music) data?"

0.4 2. Data analysis and preprocessing

The data provided came in a two matrices; one with 4363 rows, each representing a sample from a song with 264 features corresponding to rhythm patterns (indices 0 to 167), chroma (indices 168 to 215) and finally MFCCs or Mel-frequency cepstral coefficients (indices 216 to 263). This matrix was complemented by the corresponding label vector, with 4363 rows - one for each sample and values ranging from 1 to 10 each representing a music genre (Pop rock, electronic, rap, jazz, latin, RnB, international, country, reggae and blues). These are used in training and validating our classifying model. The data provided for making predictions came in a matrix of 6544 rows (samples) and 264 features. No label data was provided for this dataset. This will be the dataset the model will be tested against. We notice that the MFCC section of the raw data starts off with some quite differing values, which would make classifying the data more difficult. We aim to counter by standardizing the values. Under is provided the visualization of some of the raw data and a standardized version of the data. We notice that the MFCC section of the raw data starts off with some quite differing values, which would make classifying the data more difficult. We aim to counter by standardizing the values. The standardized plots below show for some much more easy-to-work-on data. The other other methods trialed for data-preprocessing with a weaker end result included using a MinMaxScaler to scale the data into a more workable format. With the amount of features (254) being quite high, PCA feature reduction was also tried, with n-values ranging from 3 to 150 but the method was deemed unsuccessful. Another attempt was made to reduce the features by feature selection and simply taking the mean of the three different sections of the data (rhythm, chroma and MFCCs). Both of these methods were also deemed to be unsuccessful. In the end the best result was reached by using all 264 features provided. It should be noted that the decision to not reduce features was only feasible due to the relatively low amount of data and that the feature amount was still manageable.

In [24]:
```
# Import libraries
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import itertools
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix
```

In [25]:
```
#READ THE DATA INTO MEMORY
data = pd.read_csv('train_data.csv', header=None)
data_labels = pd.read_csv('train_labels.csv', header=None) - 1
test_data = pd.read_csv('test_data.csv',header=None)

#SCALE THE DATA
standard_scaler = StandardScaler()
org_data = data #SAVE ORGINAL DATA FOR PLOTTING
data = standard_scaler.fit_transform(data)
test_data = standard_scaler.fit_transform(test_data)

reduced=data
```

```
test_reduced=test_data

reduced = pd.DataFrame(data=reduced) #MAKE INTO A DF

#ADD LABELS
reduced = pd.merge(data_labels, reduced, left_index=True, right_index=True)
reduced.rename(columns = {'0_x':'label', '0_y':'0'}, inplace = True)

reduced.head()
```

Out[25]:

	label	0	1	2	3	4	5	\
0	0	-1.571333	-1.423949	-0.935285	-1.169432	-0.772396	-1.258998	
1	0	-0.602171	0.267213	0.055395	-0.711935	-1.073589	-0.876857	
2	0	-0.584983	0.148239	0.606352	0.806748	0.075890	-0.140405	
3	0	0.193432	1.292285	0.854470	0.486911	0.341579	0.529897	
4	0	-0.825078	-0.712986	-0.865418	-1.029277	-1.073589	-1.243794	

	6	7	8	...	254	255	256	\
0	-0.811812	-0.985136	-0.581884	...	1.279245	1.266232	-0.158097	
1	-0.373028	0.155161	0.399480	...	-0.897103	-0.852736	0.074136	
2	-0.116555	0.220999	-0.266751	...	-0.406852	1.244176	0.375294	
3	0.640292	0.450361	0.375602	...	-0.975856	0.528008	-0.661580	
4	-0.874860	-1.021802	-0.959120	...	-0.189255	1.433153	-0.525024	

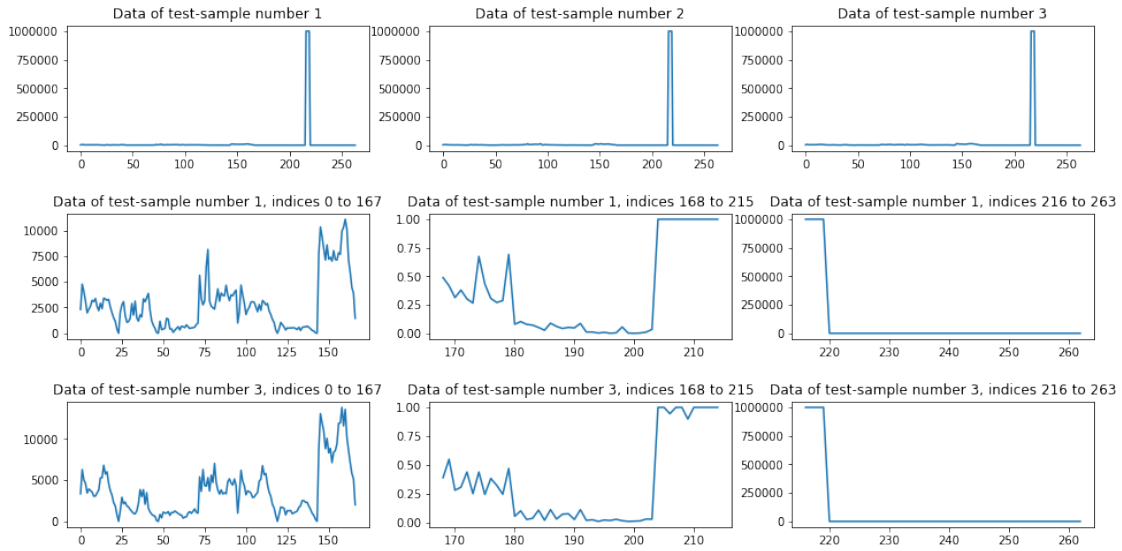
	257	258	259	260	261	262	263
0	0.297566	0.164625	0.499226	0.921775	-0.185416	0.092879	0.006096
1	1.103637	0.111916	-0.726099	0.269456	-1.177036	-0.694615	-0.223800
2	1.291144	0.824687	2.770780	-0.179373	-0.786025	0.560227	-0.548004
3	0.903560	-1.144311	0.899273	-1.229087	0.110236	2.960029	-0.956088
4	-1.507232	-0.966014	-0.535219	-0.432634	-1.336981	-1.421282	0.632431

[5 rows x 265 columns]

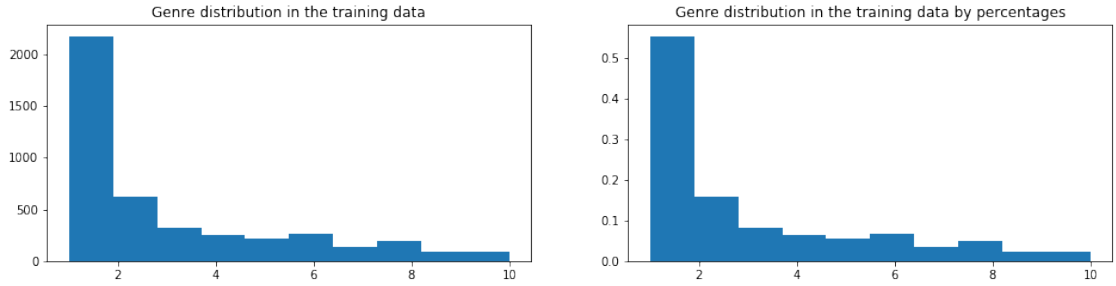
In [26]:
```
#Analysis of the input data
def plotRawData():
    fig = plt.figure(figsize=(16, 8))
    ax = fig.add_subplot(331)
    plt.plot(org_data.iloc[1,:])
    plt.title('Data of test-sample number 1')
    ax = fig.add_subplot(332)
    plt.plot(org_data.iloc[2,:])
    plt.title('Data of test-sample number 2')
    ax = fig.add_subplot(333)
    plt.plot(org_data.iloc[3,:])
    plt.title('Data of test-sample number 3')

    ax = fig.add_subplot(334)
    plt.plot(org_data.iloc[1,0:167])
```

```
        plt.title('Data of test-sample number 1, indices 0 to 167')
        ax = fig.add_subplot(335)
        plt.plot(org_data.iloc[1,168:215:])
        plt.title('Data of test-sample number 1, indices 168 to 215')
        ax = fig.add_subplot(336)
        plt.plot(org_data.iloc[1,216:263])
        plt.title('Data of test-sample number 1, indices 216 to 263')

        ax = fig.add_subplot(337)
        plt.plot(org_data.iloc[3,0:167])
        plt.title('Data of test-sample number 3, indices 0 to 167')
        ax = fig.add_subplot(338)
        plt.plot(org_data.iloc[3,168:215:])
        plt.title('Data of test-sample number 3, indices 168 to 215')
        ax = fig.add_subplot(339)
        plt.plot(org_data.iloc[3,216:263])
        plt.title('Data of test-sample number 3, indices 216 to 263')

        plt.subplots_adjust(hspace=0.5)

    def classDistribution():
        fig = plt.figure(figsize=(16,8))

        ax = fig.add_subplot(221)
        ax.hist(reduced['label']+1)
        plt.title("Genre distribution in the training data")

        ax = fig.add_subplot(222)
        ax.hist(reduced['label']+1, density=True)
        plt.title("Genre distribution in the training data by percentages")

        plt.show()
```

In [27]:
```
plotRawData() #RUN TO PLOT THE RAW DATA
```



In [28]: `classDistribution()` *#RUN TO PLOT THE DISTRIBUTION OF GENRES IN TRAINING DATA*



0.4.1 3. Methods and experiments

For the actual model, with the goal of the project steady in mind, the model was built using only slightly more than 10 lines of code for a simple, yet effective predictor. The first step is to split up the training data into the data and the corresponding labels (these were merged in a previous step for better visualization). We utilize the sklearn library method train_test_split() which splits our data into the data used to train our model and the data used to later validate it. We pass the function our data, the labels and argument "test_size" which determines how large chunk of our data will be used for validation. We use 0.2 or 20%. Now into the play comes the beautiful simplicity of the Keras API. We create a keras.Sequential, which is a linear neural network, which mean the data passed to the model will pass through its layers linearly - or one by one. We create two layers for the model, essentially leaving out hidden layers and only utilizing input and output -layers. The first layer is a Dense or fully connected -layer which mean it will take all the data we input into it and classify it accordingly. For this layer we use the activator 'elu' or "Exponential linear unit" - a good alternative for the more commonly used activator 'relu' or "Rectifier Linear

Unit" and often a more effective loss-minimizer (Pedamonti, 2018) Through trial and error, 'elu' was indeed deemed to provide higher accuracy and lower logloss. The node amount 128 was chosen half arbitrarily, with it being around half of the amount of features in one sample. This was also finally chosen through trial and error. For the second layer we use the same Dense -layer, however this time we utilize the 'softmax' activator, which gives us as an output, a matrix of shape (6544, 11) where each label is associated with a probability that it belongs to a certain genre. Here we use the n=10 since the amount must be the same as the amount of labels.

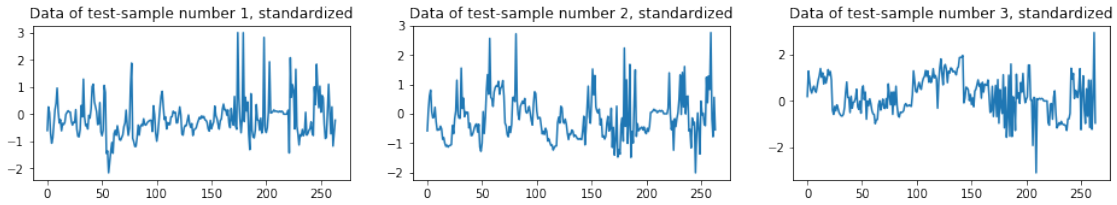
For compiling the model, we use the optimizer AdamOptimizer(), which was suggested by a Tensorflow neural network guide (Chollet, 2017). One other major optimizer included in the trial and error process was GradientDescentOptimizer, but it was deemed less accurate. Perhaps due to its linear nature. For the loss function we use sparse_categorical_crossentropy, which was also suggested by the aforementioned Tensorflow guide. It is a loss function used for this exact purpose, classifying data into categories. For the metrics we use accuracy, since it the most clear indicator of our success.

Finally we call the function fit() on our model, which fits our training data into our training labels. As the 'epoch' parameter we pass 4, which means we iterate over the dataset for four times. The amount of iterations was reach through trial and error, aiming to maximize accuracy all the while avoiding overfitting. The accuracy on the training data is roughly 70% and logloss of around 0.91.

For evaluating our model, we use the data previously cut off from our training data, and pass the data and its labels to the function evaluate(). This is a simple evaluation method that tries to predict the labels of data that it has not seen before. The accuracy on the validation data is roughly 72%. We can conduct that a small amount of underfitting might have happened, however, the difference is so minimal that we choose to ignore it. The logloss on the validation data is 0,88.

Finally we call the function predict(), giving it the test_data provided and saving it in a format suitable for Kaggle submission. For the logloss implementation, we get our answer without any problem, the 'softmax' activator automatically gives us the probabilities. For the label prediction, we choose argmax(predictions[i]), where i [0, 6544]. For improved accuracy a 'if' -clause is introduced so that only labels with over 25% confidence will be predicted with predictions where no genre passes the 25% confidence defaulting to 1, since from analyzing the data we know that if the test_data is of similar composition with the train_data, over 50% of the labels will be 1. The final accuracy on the test data is roughly 65% and the logloss is roughly 0.18.

```
In [29]: #DATA PREPROCESSING
         def plotStandardized():
             fig = plt.figure(figsize=(16, 8))
             ax = fig.add_subplot(331)
             plt.plot(data[1])
             plt.title('Data of test-sample number 1, standardized')
             ax = fig.add_subplot(332)
             plt.plot(data[2])
             plt.title('Data of test-sample number 2, standardized')
             ax = fig.add_subplot(333)
             plt.plot(data[3])
             plt.title('Data of test-sample number 3, standardized')

In [30]: plotStandardized() #RUN TO PLOT WHAT THE DATA LOOKS AFTER STANDARDIZED
```



In [31]:
```
#SEPARATE LABELS AND DATA
X = np.array(reduced.drop(['label'], axis=1))
y = np.array(reduced['label'])

#SEPARATE DATA INTO TRAINING AND VALIDATION DATA
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)


#BUILD THE MODEL THAT WILL PREDICT OUR LABELS (NOT RAN IN THIS CELL)
def build_model():
    model = keras.Sequential([
        keras.layers.Dense(128, activation=tf.nn.elu),
        keras.layers.Dense(10, activation=tf.nn.softmax)
        ])


    model.compile(optimizer=tf.train.AdamOptimizer(),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    model.fit(X_train, y_train, epochs=4)
    return model
```

In [32]:
```
#RUN TO BUILD OU
model = build_model()
```

```
Epoch 1/4
3490/3490 [==============================] - 1s 273us/step - loss: 1.4883 - acc: 0.5487
Epoch 2/4
3490/3490 [==============================] - 0s 132us/step - loss: 1.0853 - acc: 0.6564
Epoch 3/4
3490/3490 [==============================] - 0s 137us/step - loss: 0.9667 - acc: 0.6862
Epoch 4/4
3490/3490 [==============================] - 1s 152us/step - loss: 0.8911 - acc: 0.7163
```

In [33]:
```
#RUN TO VALIDATE THE MODEL WITH VALIDATION DATA. PRINTS OUT VALIDATION ACCURACY & LOG
test_loss, test_acc = model.evaluate(X_test, y_test)
print('Test accuracy:', test_acc)
print('Test logloss:', test_loss)
```

```
873/873 [==============================] - 0s 158us/step
Test accuracy: 0.630011454788
Test logloss: 1.19189186809
```

0.5 4. Results

The end result of the project is a fairly accurate model for predicting music genres. With the following figures. Training accuracy: 70% and training logloss: 0,91. Validation accuracy: 72% and validation logloss: 0,88. Final submission in Kaggle resulted in 65% accuracy and 0.18 logloss on the test data., which means a slightly lower accuracy and logloss in kaggle compared to the training and validation datasets. With a closer look at the confusion matrix, however, we can see the almost every label is mostly wrongly predicted as the most common label (pop rock). There are also several labels that are not predicted correctly even once (blues, international and latin) Hence, it would seem that the relatively high accuracy is merely due to the high amount of pop_rock samples in the training data and the test data.

In [34]:
```
#RUN TO DO PREDICTIONS ON THE TEST DATA.
predictions = model.predict(test_reduced)
```

In [35]:
```
#FUNCTION TO BUILD THE KAGGLE CSV FOR ACCURACY COMPETITION. ALSO INCLUDES THE IF CLAU
def predict_label():
    labels_predicted = np.ones((6544,2),dtype=np.int)


    for i in range(predictions.shape[0]):
        labels_predicted[i,0] = i+1
        if(max(predictions[i]) > 0.25):
            labels_predicted[i,1] = np.argmax(predictions[i])+1
    headers = "Sample_id,Sample_label"
    np.savetxt("label_pred.csv", labels_predicted,fmt='%i', header=headers, delimiter=
    print("BUILDING ACCURACY CSV SUCCESSFUL!")
    return labels_predicted


#RUN TO BUILDE THE KAGGLE CSV FOR LOGLOSS COMPETITION
def predict_logloss():

    id = [i for i in range(1,predictions.shape[0]+1)]
    predics = np.insert(predictions, 0, id, axis=1)

    fmt= ['%d','%.4f','%.4f','%.4f','%.4f','%.4f','%.4f','%.4f','%.4f','%.4f','%.4f']
    headers = "Sample_id,Class_1,Class_2,Class_3,Class_4,Class_5,Class_6,Class_7,Clas
    np.savetxt("label_all.csv", predics,fmt=fmt, header=headers, delimiter=",",comment
    print("BUILDING LOGLOSS CSV SUCCESSFUL!")
    return predics
```

In [36]:
```
x = predict_label() #RUN TO BUILD ACCURACY CSV FOR KAGGLE
#print(x) #REMOVE '#' TO PRINT OUT CSV HEAD AND TAIL
```

```
BUILDING ACCURACY CSV SUCCESSFUL!
```

```
In [37]: x = predict_logloss() #RUN TO BUILD LOGLOSS CSV FOR KAGGLE
         #print(x) #REMOVE '#' TO PRINT OUT CSV HEAD AND TAIL - NOT FORMATTED
```

```
BUILDING LOGLOSS CSV SUCCESSFUL!
```

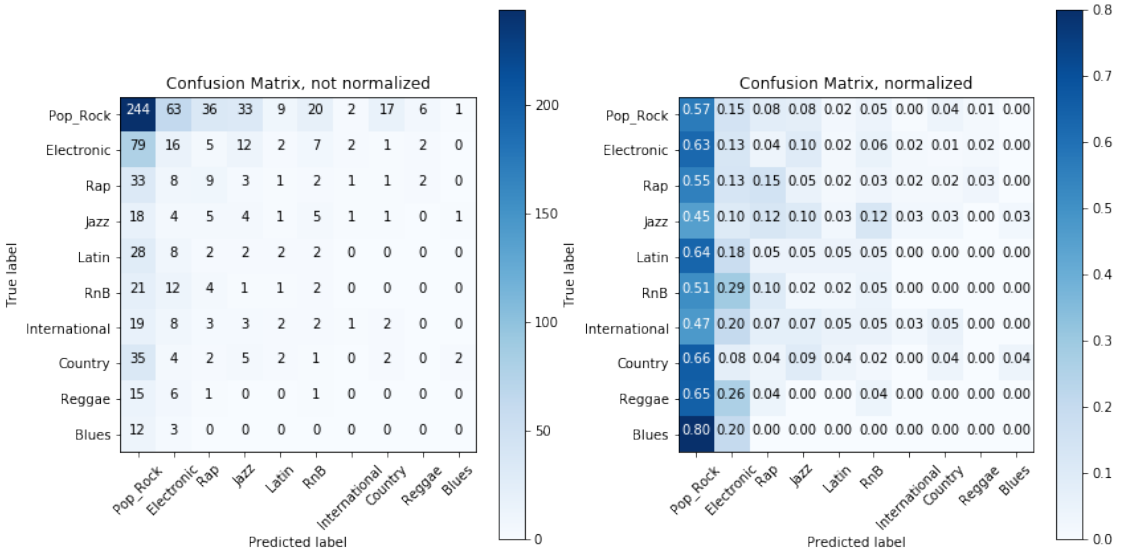
```
In [38]: #BUILD PREDICTIONS FOR TEST DATA (FOR CONFUSION MATRIX)
         def predict_test():
             predictions_test = model.predict(X_test)
             labels_predicted = np.ones(predictions_test.shape[0])
             for i in range(predictions_test.shape[0]):
                 if(max(predictions_test[i]) > 0.25):
                     labels_predicted[i] = np.argmax(predictions[i])
             return labels_predicted
```

```
In [39]: %matplotlib inline
         y_preds = predict_test()
         cm = confusion_matrix(y_test, y_preds)
```

```
In [40]: #CONFUSION MATRIX PLOTTING FUNCTION FROM SCIKIT-LEARN WEBSITE.
         #http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.h
         def plot_confusion_matrix(cm, classes,
                                   normalize=False,
                                   title='Confusion matrix',
                                   cmap=plt.cm.Blues):
             """
             This function prints and plots the confusion matrix.
             Normalization can be applied by setting `normalize=True`.
             """
             if normalize:
                 cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
                 #print("Normalized confusion matrix")
             #else:
                 #print('Confusion matrix, without normalization')

             #print(cm) REMOVE '#' TO PRINT OUT MATRIX AS TEXT ASWELL

             plt.imshow(cm, interpolation='nearest', cmap=cmap)
             plt.title(title)
             plt.colorbar()
             tick_marks = np.arange(len(classes))
             plt.xticks(tick_marks, classes, rotation=45)
             plt.yticks(tick_marks, classes)

             fmt = '.2f' if normalize else 'd'
             thresh = cm.max() / 2.
```

```
            for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
                plt.text(j, i, format(cm[i, j], fmt),
                         horizontalalignment="center",
                         color="white" if cm[i, j] > thresh else "black")

            plt.ylabel('True label')
            plt.xlabel('Predicted label')
            plt.tight_layout()
```

In [41]:
```
#RUN TO PLOT CONFUSION MATRICES
genres = ['Pop_Rock','Electronic','Rap','Jazz','Latin','RnB','International','Country
fig = plt.figure(figsize=(12,12))
ax = fig.add_subplot(221)
plot_confusion_matrix(cm, genres, title="Confusion Matrix, not normalized")

ax = fig.add_subplot(222)
plot_confusion_matrix(cm, genres, title="Confusion Matrix, normalized", normalize=Tru
```



0.6 5. Discussion/Conclusions

With a relatively simple solution accuracy of nearly two thirds can be achieved easily, although one still has to keep in mind the problematics stated in the results sections. The 65% accuracy is, however, in line with other similar experiments ie. (Goulart et al., 2012) using MFCCs reached a 61% accuracy on the 10-genre scale. With the accuracy possibly being distorted by the large amount of a single label in the datasets, I would put forward that logloss is a stronger measure of the success of this project - and indeed logloss does show that the project is quite successful. Logloss is thought to be a better measure of a models performance anyway. (Swalin, 2018) Especially this is true with imbalanced multiclass datasets, such as our datasets, since logloss doesn't

penalize as much as accuracy, for having the wrong labels probability higher. Using logloss eliminates the all or nothing of problem of accuracy. The research question was definitely answered and shown that building a model for predicting music genres can be done quite effortlessly using the Keras high-level API. It should, however, be stated that several articles reporting their successes in classifying music genres have reported much higher results ranging from 80% to even the high 90's (albeit not necessarily on 10 genres). (Goulart et al., 2012; Ogihara & Li, 2003; Haggblade et al., 2011) Further research could be done into how problems of the imbalanced datasets could be overcome and accuracies be raised into the possible 90%'s.

0.7 6. References

Bayle, Y. (2018) Deep Learning for Music. Accessible online. https://github.com/ybayle/awesome-deep-learning-music

Chollet, F. (2017) Train your first neural network: basic classification. Accessible online. https://www.tensorflow.org/tutorials/keras/basic_classification

Despois, J. (2016) Finding the genre of a song with Deep Learning. Accessible online. https://hackernoon.com/finding-the-genre-of-a-song-with-deep-learning-da8f59a61194

Goulart, A., Guido, R. & Maciel, C. (2012) Exploring different approaches for music genre classification

Haggblade, M., Hong, Y. & Kao, K. (2011) Music Genre Classification

Ogihara, M. & Li, Q. (2003) A Comparative Study on Content-Based Music Genre Classification

Oord, A., Dieleman, S. & Schauwen, B. (2014) Deep content-based music recommendations. Accessible online. http://papers.nips.cc/paper/5004-deep-content-based-

Pedamonti, D. (2018) Comparison of non-linear activation functions for deep neural networks on MNIST classification task. Accessible online. https://arxiv.org/pdf/1804.02763.pdf

Swalin, A. (2018) Choosing the Right Metric for Evaluatnig Machine Learning Models - part 2. Accessible online. https://medium.com/usf-msds/choosing-the-right-metric-for-evaluating-machine-learning-models-part-2-86d5649a5428