NAML Project Report - Group 14

Music Genre Classification using
k-Nearest Neighbours
Nearest Centroid
Multiclass SVM
-

by Silvia Marino (10688672) and Francesco Panebianco (10632465)

prof. Edie Miglio e Francesco Regazzoni - Politecnico di Milano

February 2022

Contents

1 Introduction

Scope

The scope of this project is to create a music genre classifier using two machine learning methods: *k-Nearest Neighbours* and *Multiclass SVM*, which were assigned to our group. The project is part of the evaluation of the "Numerical Analysis for Machine Learning" course, taught in the first semester of the first year of Master's Degree in Computer Science and Engineering at Politecnico di Milano. Given the similarities between *k-Nearest Neighbours* and *Nearest Centroid*, we chose to implement the latter as well, comparing its performance to the former, even though it is outside the specification of the project.

The Dataset

The dataset assigned to our project is the notorious *GTZAN Genre Collection*[3], which contains 100 different extracts from 10 different music genres. The dataset genres are:

- Blues
- Classical
- Country
- Disco
- Hip Hop
- Jazz
- Metal
- Pop
- Reggae
- Rock

As it can be seen, genres that share similarities are included in the dataset (e.g. Blues and Jazz), but also dramatically different types of music such as Rock and Classical, which we expect the algorithms to distinguish with higher precision.

The dataset provides audio samples as *.wav* files (*Waveform Audio File Format*). WAV is the most common uncompressed audio file format in Microsoft Windows systems. It was developed by IBM and Microsoft, for storing an audio bitstream on PCs[1].
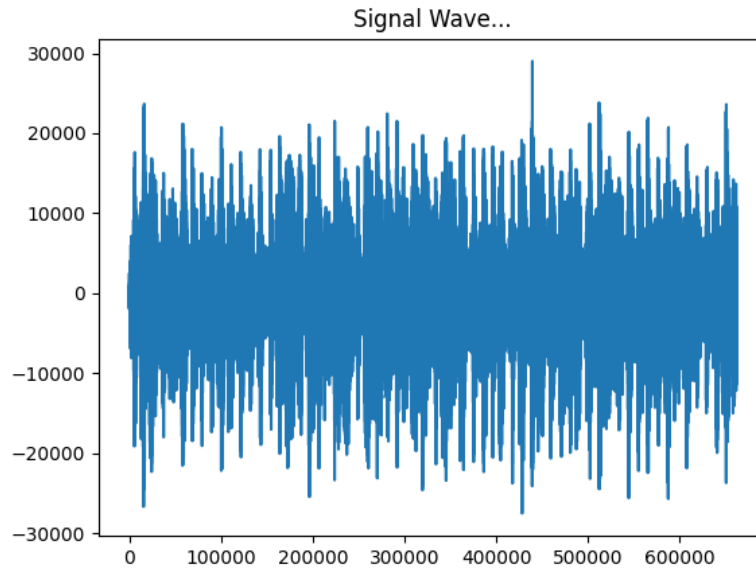
Figure 1: Plotted waveform of blues.00000.wav

This file format can be imported in *Python* by using the *wave* module, which provides convenient functions to work with the WAV sound format. In our case, the audio samples are provided as 22050Hz Mono 16-bit 30 second tracks[3], which are easily transformed into a numpy array of *int16*.

The picture above is the result of the following code:

```python
import wave
import matplotlib.pyplot as plt
import numpy as np

test_file = wave.open('genres/blues/blues.00000.wav', 'rb')
# Extract Raw Audio from Wav File
signal = test_file.readframes(-1)
signal = np.frombuffer(signal, dtype='int16')
test_file.close()

plt.figure(1)
plt.title("Signal Wave...")
plt.plot(signal)
plt.show()
```

Feature Extraction

For what concerns the features required to classify the genres, our first choice was the feature triplet of *Zero Crossing Rate*, *Average Energy* and *Silent Ratio* as defined in one of the papers provided by the project specification [2]. The extracted features were collected and exported in a csv file to avoid having to extract the feature multiple times, as the complete process took approximately 45 minutes on *Google Colab*.

Zero Crossing Rate
It indicates the frequency of signal amplitude sign change, which is in some way related to the average signal frequency. In practice, it is a key feature to classify percussive sounds as it's often correlated with the beat. ZCR records how many waves have passed for a certain time, by averaging the sign changes in the signal amplitude. The implemented formula is as follows:

$$\text{ZCR} = \frac{\sum_{n=1}^{N-1} \left| \text{sgn } x(n) - \text{sgn } x(n-1) \right|}{2N} \tag{1}$$

where sgn x is the *sign function*.

Average Energy
It indicates the loudness of the audio signal as a whole, being the average of the square amplitude of the audio signal.

$$\text{E}_{\text{avg}} = \frac{\sum_{n=0}^{N-1} x(n)^2}{N} \tag{2}$$

In practice, it corresponds to the momentum or the force of the music within the time slice where it is measured.

Silent Ratio
It indicates the proportion of the sound piece that is considered to be *silent*. Silence is defined as a period within which the absolute value of amplitude is below a certain threshold. If we indicate

$$SR = \frac{\sum_{x(n)<\text{thr}} 1}{N} \tag{3}$$

In our implementation, silence threshold is derived from the average energy as follows...

$$\text{thr} = 0.8\sqrt{E_{\text{avg}}} \tag{4}$$

Dataset Visualization

After exporting the feature dataset, we visualized it using built-in functions from *pandas* and *seaborn*. Full dataset visualization is available in a the Jupyter Notebook called *NAML_Project_Data_Visualization.ipynb*.

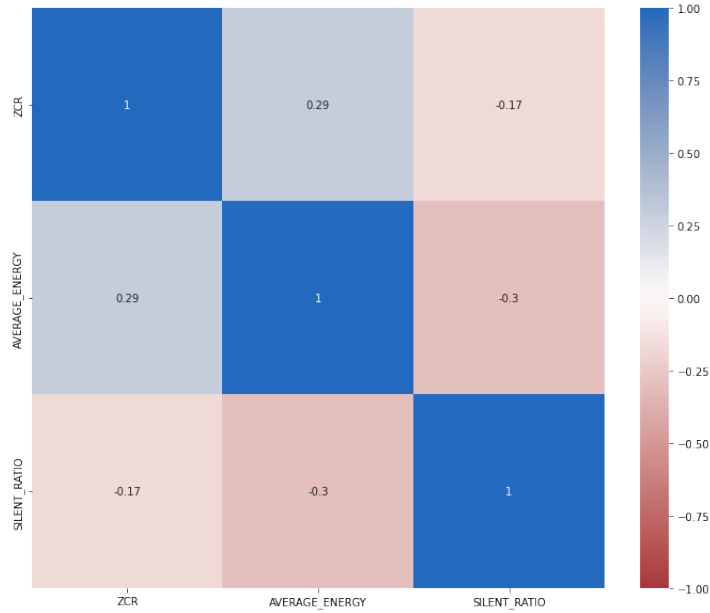|	ZCR	AVERAGE_ENERGY	SILENT_RATIO
count	1000.000000	1.000000e+03	1000.000000
mean	0.103768	2.631159e+07	0.827613
std	0.041886	2.396840e+07	0.024517
min	0.021714	6.178678e+04	0.767521
25%	0.070328	9.781419e+06	0.810039
50%	0.099618	1.834412e+07	0.824969
75%	0.132136	3.768007e+07	0.842175
max	0.275001	1.777716e+08	0.926341

Figure 2: Annotated Heatmap of the correlation of extracted features

From this visualization, it is evident that the three features are independent enough to be non-redundant in the classification.
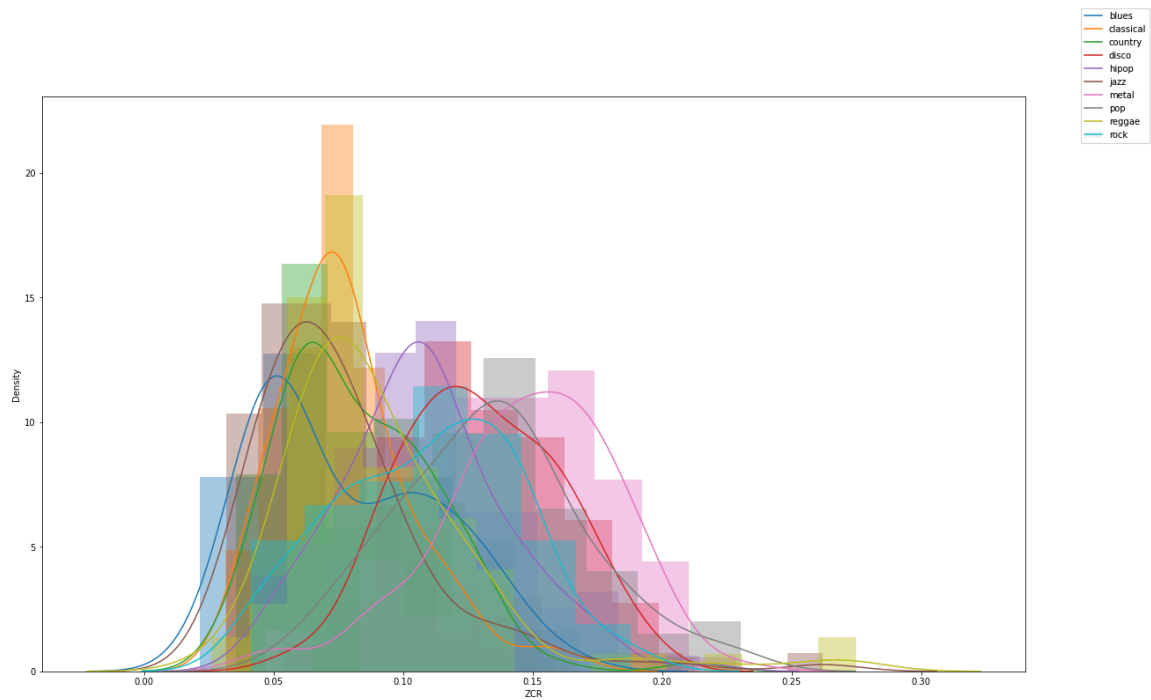
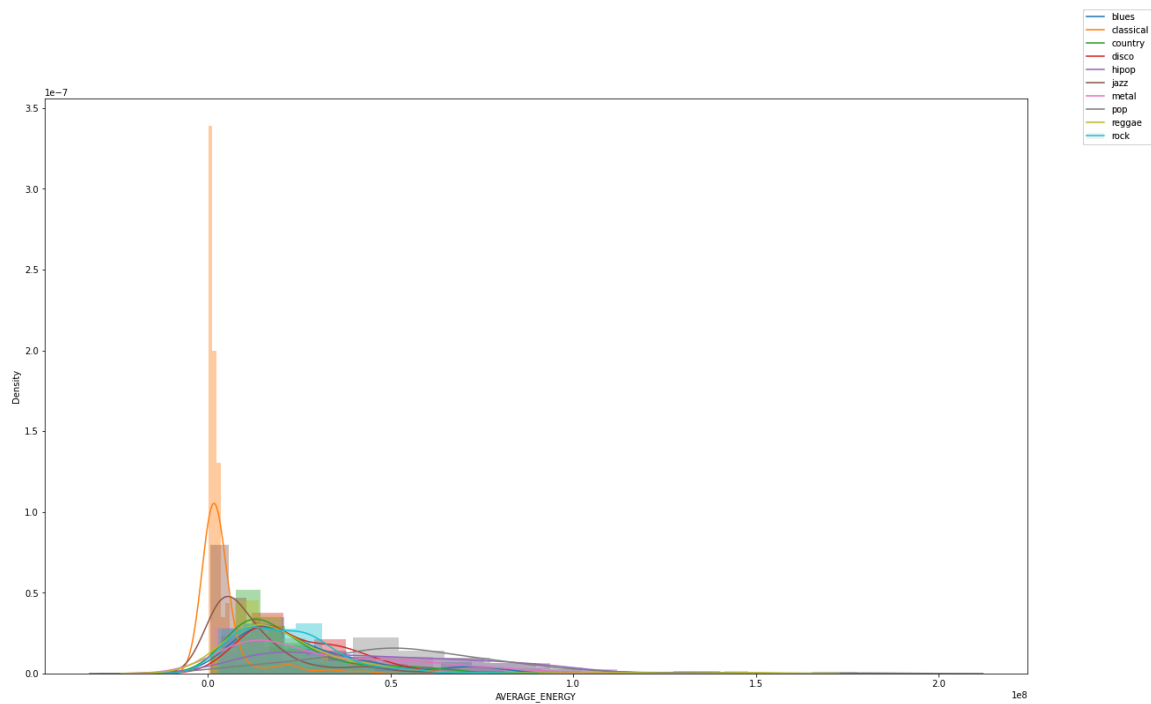Figure 3: Distribution Plot of ZCR feature

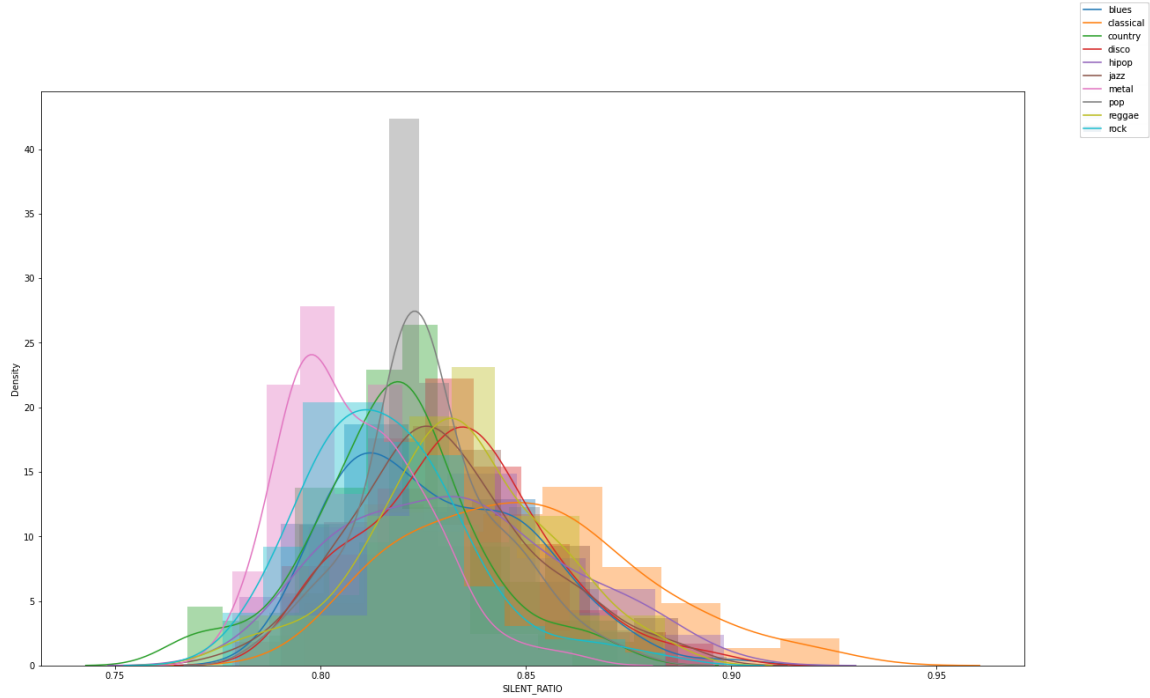Figure 4: Distribution Plot of Average Energy feature

Figure 5: Distribution Plot of Silent Ratio feature

Alternative Feature Set: Mel-Frequency Cepstral Coefficients

The second paper provided by the project specification [4] mentioned the possibility of using *Mel-Frequency Cepstral Coefficients* as features for music genre classification.

The **Mel-frequency Cepstrum** (MFC) is a representation of the short term power spectrum. The **Mel-frequency Cepstral Coefficients** (MFCCs) collectively make up an MFC. A Cepstrum is defined as a non-linear spectrum-of-a-spectrum.

The peculiarity of an MFC with respect to a generic cepstrum is, as the name suggests, the usage of the *Mel-frequency Scale*, which emulates the human hearing response more closely than the linearly spaced frequency bands used in the normal spectrum.

MFCC extraction is a relatively standardized algorithm, so we've chosen a popular python library to aid us in the extraction: *librosa*. As the library source code shows [5], the implemented extraction procedure is as follows:

1. Since a time series is provided, the spectrogram of the signal is computed.

2. The spectrogram is mapped to the mel scale

3. The log of the result is taken (dB conversion)

4. The *Discrete Cosine Transform* is applied

The Librosa implementation also supports a final processing step: Sinusoidal Liftering, which is reportedly considered beneficial to speech recognition applications. Since this is outside the scope of the classifier, no Liftering was applied.

The input signal is automatically divided into a certain number of windows, each with their MFCCs, most applications can deal with the first 13 MFCCs, as they carry the most relevant information for the section of the spectrum that can be heard by the human ear.
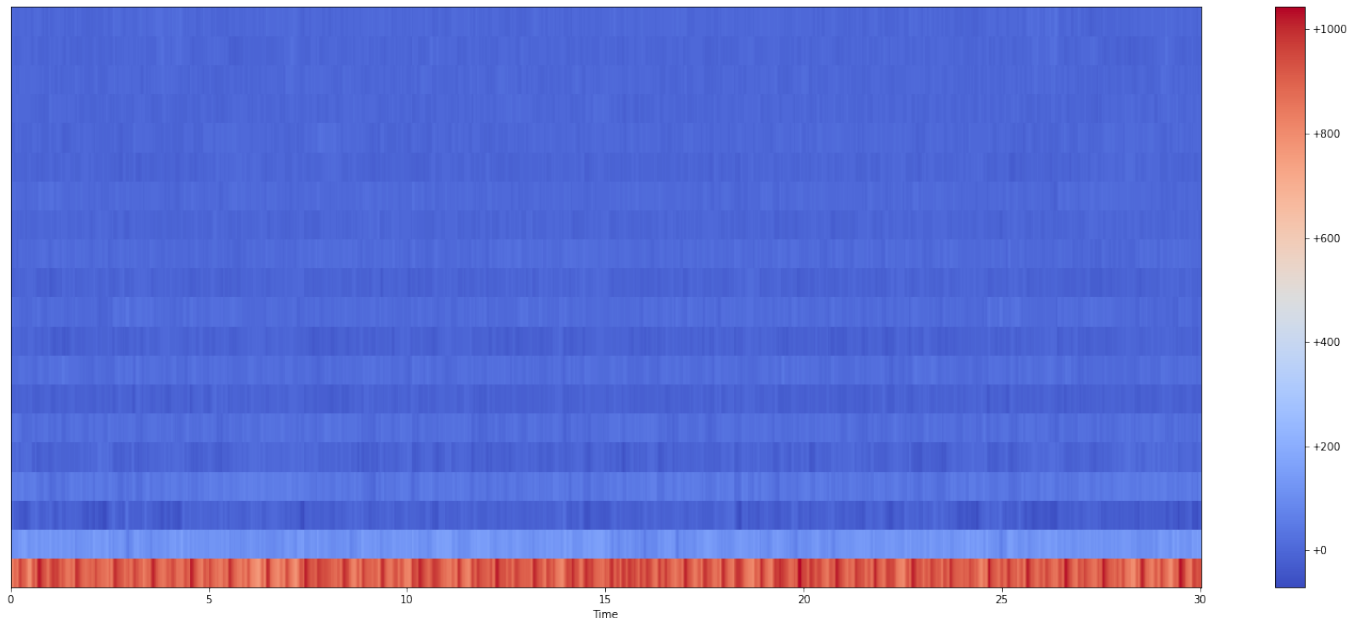

Figure 6: Mel-Frequency Cepstral Coefficents spectrogram of *blues.00000.wav*

The resulting MFCCs of the signal are returned in a matrix with rows corresponding to the 13 coefficients and columns corresponding to all windows of the original signal. This is a dynamic representation of the signal, but we need to make a choice to extract static features for our classifier. We first tried averaging the coefficients across all windows, but their correlation was too high. In the end we noticed that picking coefficients from a random window of the song extract gave us a solid feature set with low enough correlation for the classifier to perform in a satisfactory manner.

Librosa also allows for the computation of local estimates of n-order derivatives for MFCCs using the *librosa.feature.delta* function. We also included these features (namely order 1 and 2 deltas) from the same random window as the coefficients.

A specific Jupyter Notebook named *NAML Project Mel-frequency cepstrum.ipynb* handles the visualization and generation of the MFCC data for the dataset assigned to this project (the exported feature set is called *mfcc_dataset.csv*).

8

2 Classifiers

Distance Based Methods

In this paragraph we will explore the classifiers using the feature triplet (ZCR, Avg. Energy and Silent Ratio) using *k-Nearest Neighbours* and *Nearest Centroid*. The performance of both classifiers has been evaluated for different subsets of genres and using three different distance functions. Inspiration for the distance functions was taken from a recent computer science project work[7]. In the following formulas x is the current sample, y is the current centroid / training sample:

- **Euclidean Distance**:
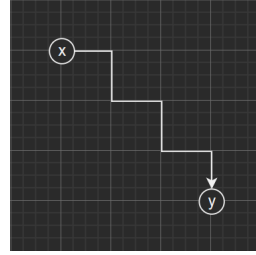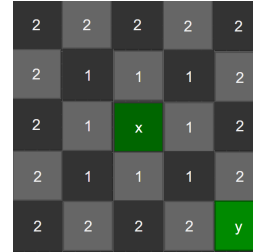
$$d(x, y) = \sqrt{\sum_{i=1}^{n} (y_i - x_i)^2} \tag{5}$$
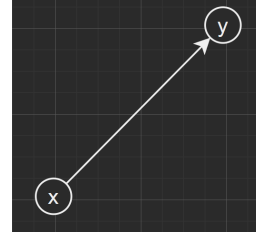
- **Manhattan Distance**:

$$d(x, y) = \sum_{i=1}^{n} \left| y_i - x_i \right| \tag{6}$$

- **Chebyshev Distance**:

$$d(x, y) = \max_{i} \left| y_i - x_i \right| \tag{7}$$

k-Nearest Neighbours Classifier

The implementation of the *k-Nearest Neighbours Classifier* was developed and tested in the jupyter notebook *NAML_Project_k_Nearest_Neighbours.ipynb* and was later ported to a standalone python script to be callable as a module (/knn/k_nearest_neighbours_classifier.py).

We calculate the distances between the input and all the data of the genres previously selected, we then store in an array the k minimum distances and the classes to which they belong. If k is greater than one, we count the occurrences of each class present in the k nearest points. In case of a tie, we return the class with the smallest distance.

k-Nearest Neighbours Classifier *Accuracy*		
Euclidean Distance	All 10 genres	6 genres	2 genres
k = 1	21.00%	35.04%	85.71%
k = 3	24.00%	35.04%	77.14%
k = 5	25.50%	29.06%	77.14%
k = 7	20.00%	35.90%	85.71%
k = 9	21.50%	30.77%	85.71%
Manhattan Distance	All 10 genres	6 genres	2 genres
k = 1	18.50%	32.48%	85.71 %
k = 3	20.00%	32.48%	77.14%
k = 5	23.50%	24.79%	85.71 %
k = 7	22.50%	29.06%	85.71 %
k = 9	21.00%	21.37%	85.71 %
Chebyshev Distance	All 10 genres	6 genres	2 genres
k = 1	21.5%	25.64%	88.57%
k = 3	23.5%	27.35%	62.86%
k = 5	20.5%	23.08%	71.43%
k = 7	18.00%	29.06%	88.57%
k = 9	17.50%	33.33%	88.57%

6 genres: ['blues', 'classical', 'country', 'disco', 'pop', 'rock']
2 genres: ['classical', 'pop']

Figure 7: Accuracy table for k-Nearest Neighbours

Nearest Centroid Classifier

The implementation of the *Nearest Centroid Classifier* is straightforward enough. It was developed and tested in the jupyter notebook *NAML_Project_Nearest_Centroid.ipynb* and was later ported to a standalone python script to be callable as a module (/nearest_centroid/nearest_centroid_classifier.py). To determine the centroid for each class, the training set is filtered and the mean of each feature is computed. A list called *centroids* collects the centroid vectors after their computation.
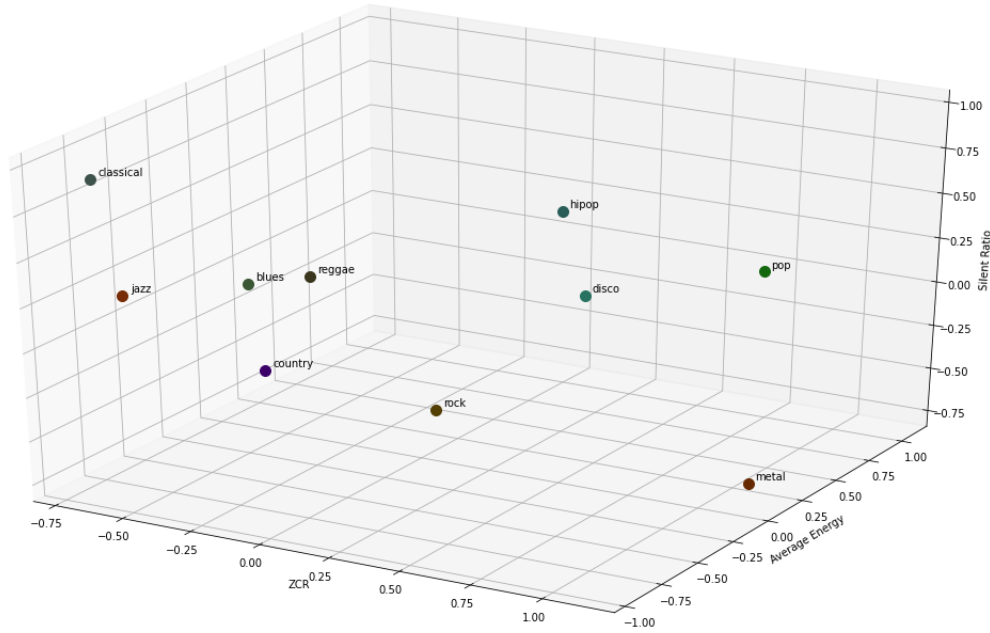

Figure 8: 3d scatterplot of class centroids

To classify the genre given the input and the possible genres to classify to, the algorithm determines the closest centroid among the available ones using the distance function passed as a parameter in the *classify* function. Here are the resulting classification accuracies:

Nearest Centroid Classifier *Accuracy*	All 10 genres	6 genres	2 genres
Euclidean Distance	34.00%	46.96%	95.45%
Manhattan Distance	33.50%	47.83%	95.45%
Chebyshev Distance	33.50%	45.22%	95.45%
6 genres: ['blues', 'classical', 'country', 'disco', 'pop', 'rock'] 2 genres: ['classical', 'pop']			

Figure 9: Classification accuracy for each distance/genre set

Multiclass SVM Classifiers

The implementation of a *Support Vector Machine* classifier can be accomplished in a variety of different ways. To explore them before moving to the full Multiclass application, we created a Jupyter Notebook called *NAML_Project_Experiments.ipynb*. After testing the binary classification performance, we adapted the code for the multi-class case, exploring both the *One-To-Rest* strategy and the *One-To-One* strategy.

Building Blocks

The implementation of a Support Vector Machine problem can be either derived from the primal or the dual formulation.

Primal Formulation - Hard Margin
The problem of computing the *maximum-margin hyperplane* exploiting *support vectors* (samples from either class lying on the "gutter" that separates the two classes) can be formulated as follows:

$$\min_{w,b} \|w\| \qquad\qquad \text{s.t.} \quad y_i(w^T x_i - b) \geq 1 \qquad\qquad (8)$$

Where $\mathbf{y_i} = \pm\mathbf{1}$ is the label of the class corresponding to sample $\mathbf{x_i}$.
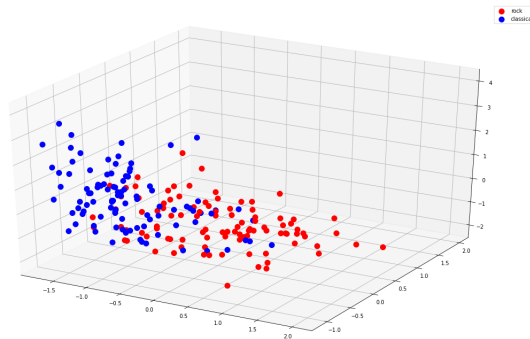

Figure 10: 3d scatterplot of rock and blues samples

Unfortunately, our samples are not linearly separable using a hard-margin, so this basic form of classification was not chosen.

Primal Formulation - Soft Margin

The soft-margin formulation involves the usage of the **hingeloss** function, which makes the optimization problem become

$$\min_{w,b} \quad \frac{1}{n} \sum_{i=1}^{n} \max\left(0, 1 - y_i(w^T x_i - b)\right) \tag{9}$$

The optimization problem is unconstrained, which enables us to apply the *Gradient Descent* method, particularly appropriate considering the convexity of the function. Most soft-margin implementations also feature a *penalization* term, but in our implementation such parameter ended up decreasing the accuracy. Hence, we decided to remove it.
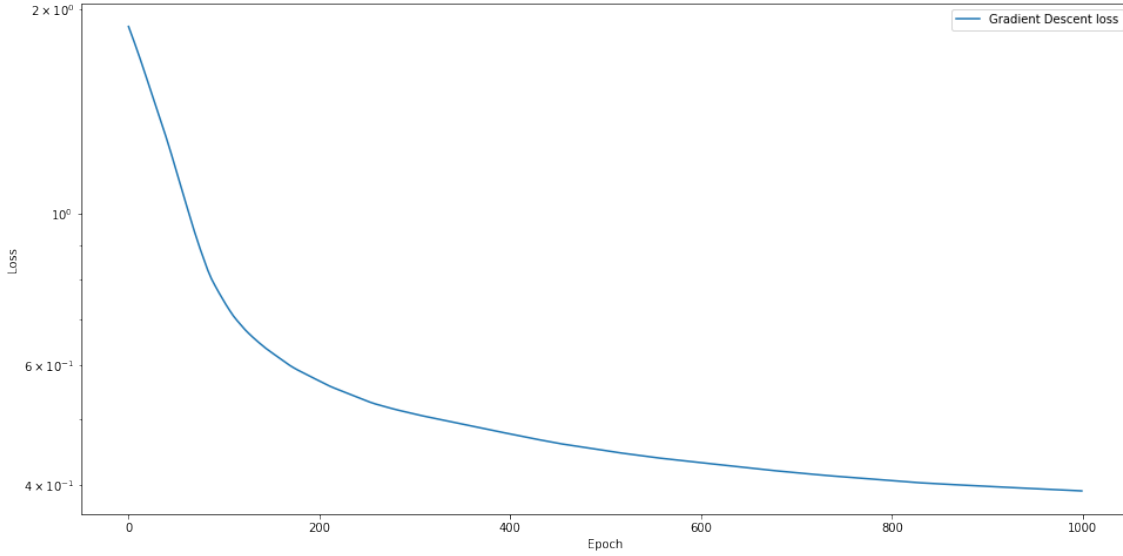

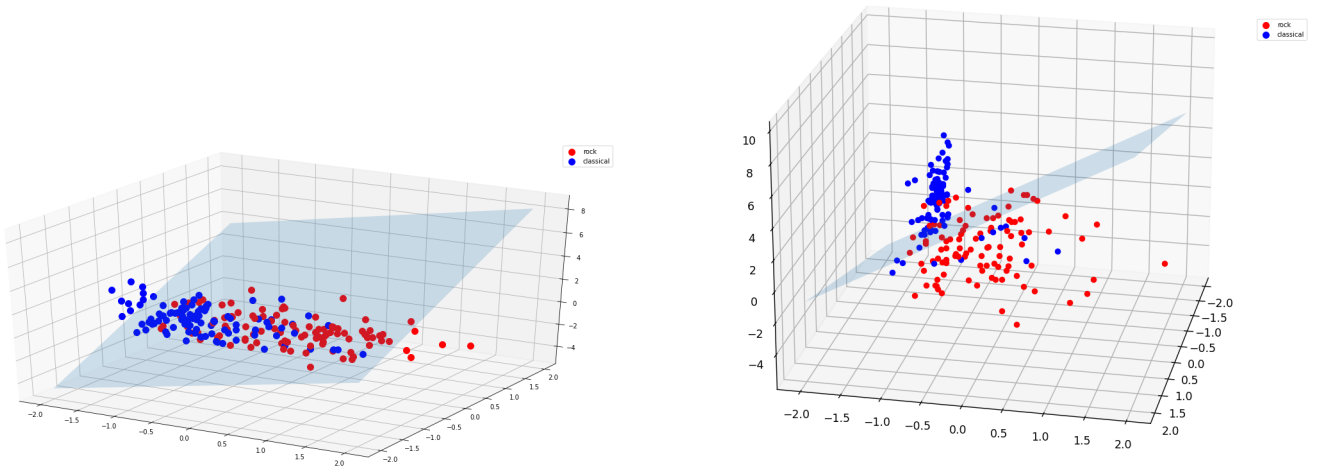Figure 11: Gradient Descent hingeloss history on rock/classical margin

Figure 12: Plots of the hyperplane corresponding to the optimal parameters (two different points of view)

13

Primal Formulation - Soft Margin with Feature Map

Before trying the dual formulation, which allows the usage of the Kernel Trick for non-linear classification margins, we tried a simple quadratic *feature map* to be applied to the primal formulation. This way, the three samples (ZCR, average energy and silent ratio) are mapped to a higher dimensional feature space.

$$\phi(x_1, x_2, x_3) = \begin{bmatrix} x_1 & x_2 & x_3 & x_1^2 & x_2^2 & x_3^2 & x_1x_2 & x_2x_3 & x_1x_3 \end{bmatrix}^T \tag{10}$$
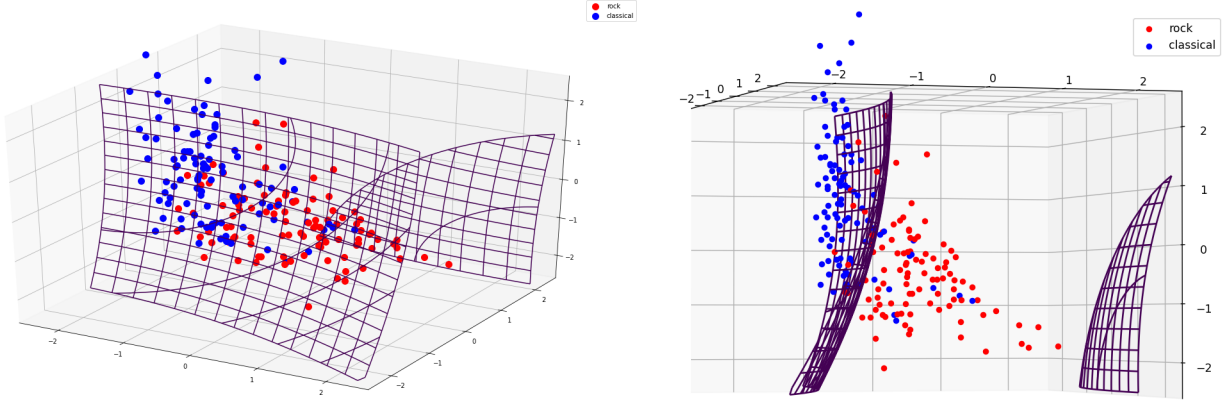

Figure 13: Plots of the non-linear margin corresponding to the optimal parameters (two different points of view)

Dual Formulation - Linear Margin

The dual formulation of the SVM is easily obtained from the soft-margin primal formulation by using *Lagrangian Multipliers...*

$$\max_{c_1,\ldots,c_n} \quad \sum_{i=1}^{n} c_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j y_i y_j x_i^T x_j \qquad \text{s.t.} \quad \sum_{i=1}^{n} c_i y_i = 0 \quad \wedge \quad c_i \geq 0$$

This time the optimization problem is constrained, so we had to use a *Quadratic Programming* library available for python. We ended up using *scipy.optimize*. Since the only constraint on the decision variable is linear, the computation of an iteration did not require the hessian, which was hardcoded to zero. The parameters of the optimization were method='trust-constr' (constrained minimization of scalar function) and jac='2-point' which enables the calculation of the jacobian of the constraint using finite differences [6]. Of course, since the dual formulation is a maximization problem, the objective function has to be multiplied by $-\mathbf{1}$ before being fed to the *scipy.optimize.minimize* method.

After running the computation, weights and bias can be derived from the following expressions

$$w = \sum_{i=1}^{n} c_i y_i x_i \qquad b = w^T x_i - y_i \text{ for any } i \text{ such that } x_i \text{ lies on the margin } (c_i > 0) \tag{11}$$

14

Dual Formulation - Kernel Trick

The kernel trick can intuitively be applied to the objective function by replacing the dot product of two samples $\mathbf{x_i^T x_j}$ with the kernel function $\mathbf{K(x_i, x_j)}$. The choice of the kernel function itself, unfortunately, was constrained by the performance of its computation. While a polynomial kernel was fast due to the matrix optimizations of *numpy*, a gaussian kernel was found expensive. The following kernel had the best results in terms of classification accuracy:

$$K(x_i, x_j) = (x_i^T x_j + 1)^3 \tag{12}$$

The computation of the weights (multiplied by the kernel) and the bias is as follows:

$$w_j^\phi = \sum_{i=1}^{n} c_i y_i K(x_j, x_i) \quad b = w_i^\phi - y_i \ \text{ for any } i \text{ such that } x_i \ \text{ lies on the margin } (c_i > 0) \tag{13}$$

From binary to multiclass

After testing all previously presented methods we noticed that the best results were provided by the classifiers that used *Kernel Trick with Dual Formulation* and *Feature Map with Primal Formulation*, so the first was chosen for the One-To-Rest classifier while the second was implemented in the One-To-One model.

One-To-Rest Classifier

The implementation and testing of this classifier was done in *Google Colab* on the *NAML_Project_Multiclass_OneToRest.ipynb* and was later ported to a python module for further usage. Even in this case, due to long execution time on *Google Colab* (around 15 minutes), the usage of the standalone python module is discouraged, as it takes way more than 15 minutes to train the model on personal devices:

Classifier *Accuracy*			
	All 10 genres	6 genres	2 genres
SVM Multiclass One-To-Rest	27.50 %	33.06 %	56.82 %

6 genres: ['blues', 'classical', 'country', 'disco', 'pop', 'rock']
2 genres: ['classical', 'pop']

Figure 14: Classification accuracy for each genre set

Each binary classifier checks if the input passed belongs to a specific genre or not. Specifically, it calculates a "score" that is positive if the input is classified to belong to the considered genre, and negative otherwise.

To decide which class to return, the one-to-rest classifier retrieves the highest score among the genres considered during initialization.

One-To-One Classifier

The implementation of the One-To-One classifier was done directly in *PyCharm* to better exploit debugging tools in the implementation of the object oriented architecture of the module.

Since the One-To-One classifier compares one genre to another, we decided to choose the actual predicted class using a **Binary Decision Tree**, which was implemented in a separate class using recursive algorithms. The amount of binary classifiers required for the multiclass problem of a given subset of n genres is

$$\# \text{ classifiers} = \binom{n}{2} = \frac{n!}{2!(n-2)!} \tag{14}$$

Since the maximum number of genres to classify is 10, the maximum number of binary classifiers to be instantiated is 45.

Once the class *MulticlassSVM_OTO* is instantiated, the __init__() method creates all combinations of classifiers and then recursively generates a binary decision tree, assigning a binary classifier to each node.

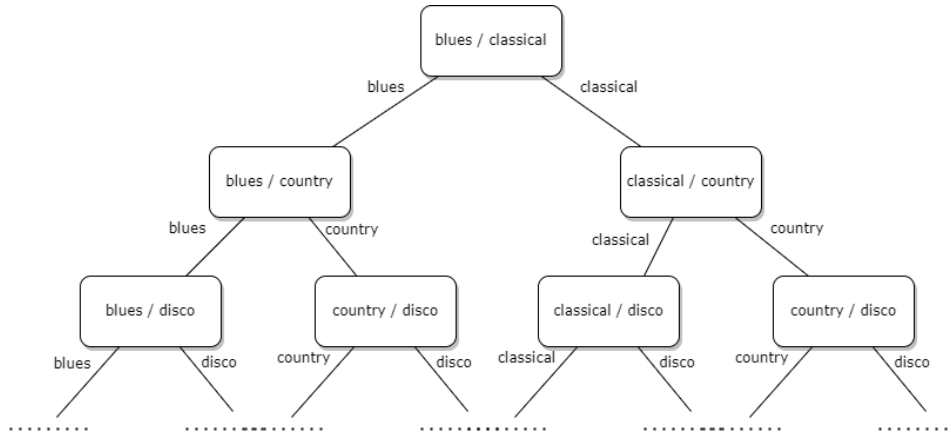Figure 15: Truncated example of decision tree as generated upon instantiation of DecisionBinaryTree

Multiclass SVM (OTO)	All 10 genres	6 genres	2 genres
Primal with Feature Map	37.00%	50.00%	95.24%
6 genres: ['blues', 'classical', 'country', 'disco', 'pop', 'rock'] 2 genres: ['classical', 'pop']			

Figure 16: Classification accuracy for each genre set

Classification with MFCCs

Once the implementation of the classifiers with the feature triplet was completed, we started working with the same classifiers changing the feature set to the extracted MFCCs. All MFCC classifiers were implemented and tested on *Google Colab* as jupyter notebook *MFCC_PCA.ipynb*. As the name of the notebook suggests, we were sure the usage of all 39 features without processing would have been redundant, so we started with a *Principal Component Analysis* on the data.
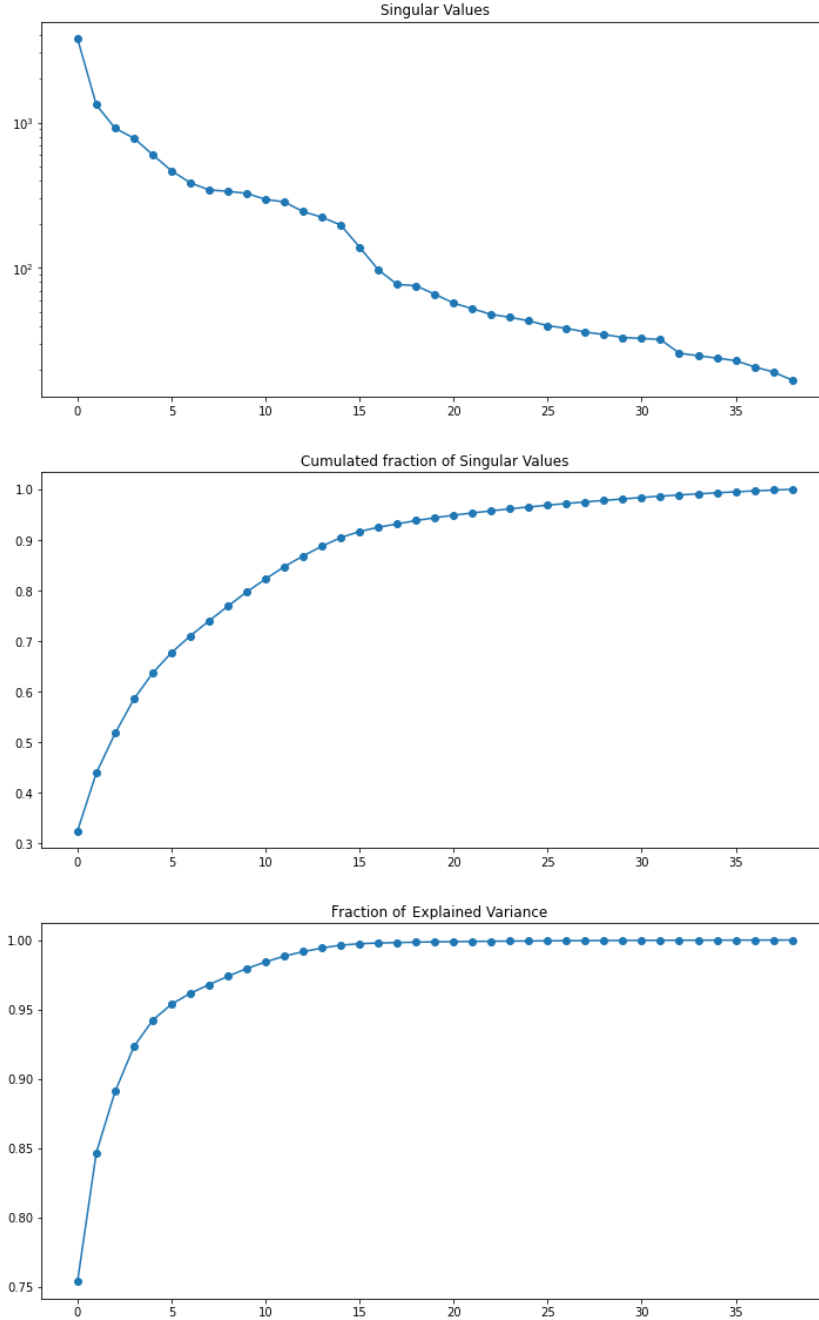

Figure 17: Results of the Principal Component Analysis performed on the MFCC dataset

From the plots, it is evident that around 10 principal components provide more than enough of the variance of the original data to be used for classification.

The accuracies for each technique are as follow:

MFCC *Accuracies*			
Nearest Centroids	All 10 genres	6 genres	2 genres
	29.00%	43.36%	84.44%
k-NearestNeighbours	All 10 genres	6 genres	2 genres
k = 1	23.50%	38.95%	93.33%
k = 3	24.50%	40.71%	95.56%
k = 5	26.00%	40.71%	93.33%
k = 7	26.50%	43.36%	91.11%
k = 9	24.50%	44.25%	91.11%
Multiclass SVM (OTO)	All 10 genres	6 genres	2 genres
Primal Linear Margin	21.00%	28.31%	91.11%

genres: ['blues', 'classical', 'country', 'disco', 'pop', 'rock']
genres: ['classical', 'pop']

Figure 18: Accuracies of classification with MFCC principal components

Please note that the One-To-One classifier implements the Linear Margin binary classifiers instead of the FeatureMap ones because a complete quadratic feature map would amount to 65-dimensional sample vectors. Moreover, considering the training time for the One-To-Rest classifier is way too long and the classification performance is poor, we did not include it in the MFCC classifiers notebook. Also, given that this alternative feature set does not improve accuracy compared to the feature triplet described before, we decided not to port the classification code to standalone python classes.

Modules User Guide

In this paragraph we will explain how to use the standalone python modules we developed. Some examples of the usage of the following classes and functions are available in the *tests* folder.

Feature Extraction Modules

- /feat_ext/sample_processing.py

```
# Used to load a wav audio signal located in "path" and of
# type "dtype" (e.g. int16, int32, ...)
load_audio_signal(path, dtype)

# Takes a audio signal and a sample rate and returns a 30 sec extract
# taken from the center of the signal
select_30sec_extract(signal, sampling_rate)

# Takes a audio signal and returns the feature triplet
# ZCR, avg_energy, silent_ratio as SEPARATE variables
get_feature_triplet(signal)


# Given a pandas dataframe of the extracted feature sets, returns
# dataset_train, labels_train, dataset_valid, labels_valid
# (in this exact order)
get_normalized_train_valid_sets(data:pd.DataFrame, feature_count=3)
```

- /feat_ext/full_feature_extraction.py

```
# Given an audio signal, returns its Zero Crossing Rate value
zero_crossing_rate(signal)

# Given an audio signal, returns its Average Energy value
average_energy(signal)

# Given an audio signal, returns its Silent Ratio value
silent_ratio(signal)

# Creates the full feature triplet set for the GTZAN dataset
# given the path to the genres folder and saves it in a file
# called 'triplet_dataset.csv' in the same folder as the module
create_dataset(path_to_genres_folder)
```

Utils for the distance based classifiers

- /utils/distances.py

```
# Returns the Manhattan Distance given x
# Assumption: x already contains the difference between the
# label and the prediction
manhattan(x)


# Returns the Chebyshev Distance given x
# Assumption: x already contains the difference between the
# label and the prediction
chebyshev(x)
```

k-Nearest Neighbours Classifier Module

- /knn/k_nearest_neighbours_classifier.py

```
class kNearestNeighboursClassifier:
        __init__() takes no parameters


        # Trains the k-Nearest Neighbours Classifier given
        # the training_set and the training_labels
        '''
        WARNING: dataset and labels are the COMPLETE dataset, even if you
        want to classify less than 10 genres
        '''
        self.train(training_set, training_labels)


        # Returns the name of the predicted genre given the
        # input sample, the list of possible genres,
        # the distance function and the number of
        # samples to check the distance from
        self.classify(input, genres_to_classify=genre_names,
                distance=np.linalg.norm, k=1)


        # Returns the confusion matrix (always 10x10)
        # of the validation set and validation_labels
        # given the list of possible genres, the distance function
        # and the number of samples to check the distance from
        self.confusion_matrix(validation_set, validation_labels,
                genres_to_classify=genre_names, distance=np.linalg.norm,k=1)


        # Returns the accuracy of the classifier given a confusion matrix
        @staticmethod
        compute_accuracy_from_matrix(c_mat)


        # Returns the accuracy of the classifier given
        # the validation set and the validation labels
        self.compute_accuracy(dataset_validation, labels_validation)
```

Nearest Centroid Classifier Module

- /nearest_centroid/nearest_centroid_classifier.py

```
class NearestCentroidClassifier:
        __init__() takes no parameters


        # Trains the Nearest Centroid Classifier given
        # the training set and the training labels
        '''
        WARNING: dataset and labels are the COMPLETE dataset, even if you
        want to classify less than 10 genres
        '''
        self.train(dataset, labels)


        # Returns the name of the predicted genre given the input sample,
        # the list of possible genres and the distance function.
        # debug_mode prints the distances between the sample
        # and the centroids
        self.classify(input, genres_to_classify=genre_names,
                distance=np.linalg.norm, debug_mode=False)


        # Returns the confusion matrix (always 10x10)
        # of the validation set and validation_labels
        # given the list of possible genres and the distance function
        self.confusion_matrix(validation_set, validation_labels,
                genres_to_classify=genre_names, distance=np.linalg.norm)


        # Returns the accuracy of the classifier given a confusion matrix
        @staticmethod
        compute_accuracy_from_matrix(c_mat)


        # Returns the accuracy of the classifier given
        # the validation set and the validation labels
        self.compute_accuracy(self, validation_set, validation_labels)
```

SVM One-To-One Module

Modules which are not described in this guide are auxiliary to the main module used for multiclass classification

- /svm/one_to_one/multiclass_oto_classifier.py

```python
class MulticlassSVM_OTO:
        # Instantiates the Multiclass SVM classifier given a list
        # of possible genres and a binary classifier type
        # (which extends SVMClassifier_OTO, can be either
        # PrimalFeatureMapClassifier or PrimalLinearClassifier)
        __init__(genre_list, classifier_type : type)

        # Trains all binary classifiers given a
        # training set and training set labels
        self.train_all(input_data, input_labels)

        # Returns the name of the predicted genre of an
        # input sample
        self.classify(input)

        # Returns the confusion matrix of the classifier
        # given the validation set and the validation labels
        self.confusion_matrix(input_data, input_labels)

        # Returns the accuracy of the classifier
        # given the validation set and the validation labels
        self.compute_accuracy(input_data, input_labels)

        # Returns the accuracy of a classifier given the confusion matrix
        @staticmethod
        accuracy_from_matrix(conf_mat)
```

SVM One-To-Rest Module

Modules which are not described in this guide are auxiliary to the main module used for multiclass classification

- /svm/one_to_rest/multiclass_otr_classifier.py

```
class MulticlassSVM_OTR:
        # Instantiates the Multiclass SVM classifier given a list
        # of possible genres
        __init__(genre_list)

        # Trains all binary classifiers given a
        # training set and training set labels
        self.train_all(dataset_train, labels_train)

        # Returns the name of the predicted genre of an
        # input sample
        self.classify(input)

        # Returns the confusion matrix of the classifier
        # given the validation set and the validation labels
        self.confusion_matrix(validation_data, validation_labels)

        # Returns the accuracy of the classifier
        # given the validation set and the validation labels
        self.compute_accuracy(input_data, input_labels)
```

References

[1] Fleischman E. *WAVE and AVI Codec Registries*. 1998. URL: `https://datatracker.ietf.org/doc/html/rfc2361`.

[2] Tamatjita Elizabeth Nurmiyati and Mahastama Aditya Wikan. "Comparison of Music Genre Classification Using Nearest Centroid Classifier and k-Nearest Neighbours". In: *International Conference on Information Management and Technology (ICIMTech)* (2016), pp. 118–123. DOI: `978-1-5090-3352-2`.

[3] Leben Jakob. *Music Analysis, Retrieval and Synthesis for Audio Signals*. URL: `http://marsyas.info/downloads/datasets.html`.

[4] Cast John, Schulze Chris, and Fauci Ali. *Music Genre Classification*. Tech. rep. Stanford University - Computer Science Machine Learning course CS229, 2013.

[5] *Librosa Github Repository: MFCC extraction details*. 2015. DOI: `10.5281/zenodo.4792298`. URL: `https://github.com/librosa/librosa/blob/main/librosa/feature/spectral.py`.

[6] *Scipy Optimization and root finding*. DOI: `10.1038/s41592-019-0686-2`. URL: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html`.

[7] Zhu Zijie and Zhang Mengtian. *K-Nearest Neighbors(KNN) Classification with Different Distance Metrics*. Tech. rep. Shanghai Jiao Tong University – Fundamentals of Data Science (CS245), 2020. URL: `http://bcmi.sjtu.edu.cn/home/niuli/teaching/2020_2_3.pdf`.