UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU DE MATEMÀTIQUES

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Supervised Learning for Genre Classification of Audio Tracks

Autor: Ángel Bergantiños Yeste

Director: Dr. Sergio Escalera Guerrero

Realitzat a: Departament of Applied Mathematics and Analysis

Barcelona, 24 de juny de 2018

Abstract

Music is a form of art that accompanies all of us every day and, with the appearance of on line services such as Spotify or Tidal, music analysis has become crucial to these services to recommend new music to users and to classify all new tracks uploaded every day.

In this dissertation, we provide with a system for multi-class classification of genres for audio tracks. We base on standard MFCC audio descriptors to then define a compact audio track feature vector representation. Different machine learning classifiers are tested to perform final genre classification, also providing an analysis of the relevance of the different features.

Resum

La música és una forma d'art que ens acompanya dia a dia i, amb l'aparició de serveis en línia com Spotify o Tidal, l'anàlisi musical s'ha tornat crucial perquè aquests serveis puguin recomanar nova música als usuaris, així com classificar totes les noves cançons que són pujades cada dia.

En aquesta dissertació, proporcionem un Sistema per la classificació per gènere de pistes d'àudio. Ens basem en els descriptors d'àudio MFCC estàndard per definir la representació com un vector de característiques compacta . Farem servir diversos classificadors de Machine Learning per a realitzar la classificació per gènere final, així com proporcionar un anàlisi de la rellevància de les diferents característiques.

Resumen

La música es un arte que nos acompaña a diario y, con la aparición de servicios online como Spotify o Tidal, el análisis musical se ha convertido en algo crucial para que estos servicios puedan recomendar nueva música a los usuarios, así como clasificar todas las nuevas canciones que son subidas cada día.

En esta disertación, proporcionamos un sistema para la clasificación por géneros de pistas de audio. Nos basamos en los descriptores de audio MFCC estándar para definir la representación como un vector de características compacto de la pista de audio. Usaremos diversos clasificadores de machine Learning para realizar la clasificación por género final, así como proporcionar un análisis de la relevancia de las diferentes características.

Contents

Chapter 1

Introduction

Music has been around since the dawn of humanity. It has changed its instruments, its representation, its style... But in any time in history, music has been an inherent condition to humanity.

Thanks to digital music, it can be heard almost everywhere at all time, allowing you to enjoy songs that really fit your taste, as well as find music you never thought you would love.

The amount of music that is around us today is way bigger than in any time in history, which makes it really difficult if you want to find music similar to yours. Because of that, music classification has become crucial, not only for people who want to find the next hit that they will love for the rest of their lives, but for companies that base their business in either music streaming or music distribution.

1.1 History of digital audio

Even though digital audio became available in 1938 as telephone technology, it wasn't until the 60s that mankind was able to record digital audio and store it in a computer.

Digital audio became possible after Harry Nyquist and Claude Shannon discovered what was known as Nyquist-Shannon Sampling Theorem, which was also

discovered by E. T. Whittaker, Vladimir Kotelnikov and others whose name hasn't been cataloged.

This theorem was, and still is, used to convert an analog signal (continuous) into a digital signal (discrete), dividing the analog signal into smaller pieces called "samples" and analyzing every sample to get a value, that will represent all frequencies in the signal.

Years later, in the 1950s and 1960s, the technology to record digital audio kept improving, but it was still too expensive to be used for the great public.

It wasn't until the 70s, that digital audio started to become mainstream, thanks to Thomas Stockham who, in 1976, built which is considered the first digital audio recorder: a 4-channel, 16-bit system that sampled at 50 KHz.

Years later, in 1982, Phillips and Sony released the CD, which allowed audio to be distributed easily, but it wasn't until the mid-80s, thanks to companies such as Mitsubishi and Sony, released the first digital audio recorder into the mainstream market.

In 1933, one of the most popular audio formats was invented: mp3, which allowed reducing audio size and making files more portable.

After that, and thanks to the release of the first iPod in 2001 and its success, digital audio became portable and easy to listen for almost everyone. [1]

1.2 History of audio classification

First methods used to classify audio

1.3 State of the art of audio classification

With the appearance of Machine Learning and Deep Learning, most of the studies in the field have switched to this approach.

1.4 Summary of the proposal

Considering this, our proposal is to develop an algorithm capable of classifying a relatively small dataset of songs of different genres.

In order to work with the dataset, we will follow the techniques that have been found to be more reliable when it comes to data representation, MFCC, which will extract the most significative features of each track and help us avoid working with all the raw data.

After this step, we will try some of the algorithms that are being used today to classify them, based on the field of machine learning, as we consider that deep learning will be too expensive in terms of computational power.

This dissertation will focus only in the classification part, although it could be used later to bigger projects such as recommenders.

Chapter 2

Method

In this chapter, we will explain the methods we are going to use later to try to classify our dataset, which will be presented later.

The songs we get can't be used as raw data, thus we need to treat the tracks to be able to work with them. This will be accomplished using MFCC, to extract audio features; PCA, to reduce the dimensionality of the matrix given by MFCC; and Decision Trees, to get the relevance of each feature and know which ones is more useful.

2.1 MFCC

After investigation, we found out that most of the projects involving audio analysis were using MFCC to extract features from the audio files.

MFCC (Mel Frequency Cepstral Coefficients) is usually used to extract features from human talk, but has been used lately for all kinds of sound. MFCC were defined by Paul Mermelstein and S. Davis in 1980.

Although it was first developed to recognize monosyllabic words in spoken form, its characteristics make it useful for all kinds of sounds.

The algorithm works as follows: [2] [3]

1. **Divide the signal in several same-sized intervals.**

   This step will take the audio file and segment it into frames of the same size. The size of the frame will depend on the characteristics of the file, but it usually uses a frame of 20 to 30 ms.

2. **Take the Fourier Transform of each interval.**

   Fourier Transform will take the frequencies of the interval and decomposes it into a finite domain of components that form the original signal.

3. **Convert the values to Mel Scale.**

   Once we have taken the Fourier Transform, we have to map the values into mel scale. This scale represents pitches which, when being judged by listeners, will be of equal distance.[4]

   To convert the frequencies (Hz) we get from the last step to mels, we use the following formula:

$$m = 2595 \cdot \log_{10}\left(1 + \frac{F}{7000}\right)$$

   This will map every frequency in its corresponding Mel Scale, which will look like the following image, but with more or less filters, depending on the amount of coefficients we want to get.[5]

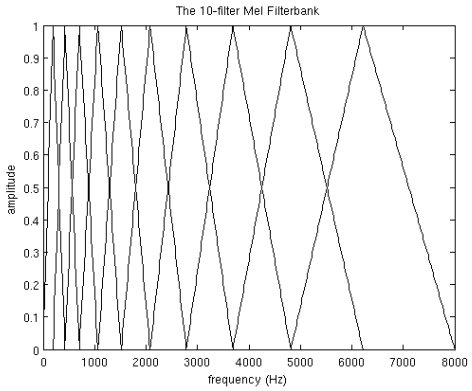


Figure 2.1: Mel Filterbank.

4. **Take power logs of each mel frequency.**

5. **Apply the discrete cosine transform (DCT) to all Mel logs.**

   Now, in order to convert the values back into the time domain, we need to apply the discrete cosine transform to all values.

   This is done using the following formula:

$$C_n = \sum_{k-1}^{k} (logD_k) \cos\left[m(k - \frac{1}{2})\frac{\pi}{k}\right]$$

   The resulting values will be MFCC.

Using this, we will end up with a matrix which size will be determined by the number of coefficients we want and the length of the audio sample.

2.2 Feature Vector representation

Once we have extracted the features using MFCC, we have to decide what are we going to do with them, given that the amount of features we get will always be, in our case, bigger than the dataset we can work with.

We will work with two different representations of these features: using all the raw data and creating histograms of each component.

2.2.1 Naive

The first method we will try will use all the values we get from MFCC. This method will take the matrix whole matrix and convert it into a 1-dimensional array, created by concatenating each row, which size will depend on the length of the song, one after another.

This way, we will have our dataset converted into a matrix of as many rows as songs it has by the length of each array.

The amount of information we will have to work with will be enormous, but we will use it to have a first approximation of the accuracy of our classifier.

2.2.2 Component histograms

As we said before, we want to reduce the amount of values we have, but being able to still have the most information we can, as well as remove the effect of time in our experiment.

In order to do that, we will have as many histograms as coefficients we use, and will be built following this procedure:

1. Take maximum and minimum values of all dataset.
2. Divide the interval in as many steps as you want.
3. Create a histogram for each coefficient.
4. Put every value of the corresponding row into its interval.
5. Divide every final value by the amount of values you have.
6. Concatenate each histogram into a 1-dimensional array.

This way, each song will be represented by an array with its size depending on the number of coefficients and the amount of steps we take.

2.3 Dimensionality reduction - PCA

Once we have both representations of the feature vector, we will try one last modification of it.

This will be done by applying PCA (Principal Component Analysis) to the matrix we have, which will reduce the size of it even more.

The objective of this procedure is to have a feature vector smaller than the size of the dataset, which we expect it will help classification.

PCA works by orthogonally transforming a set which may have correlation into a new one linearly uncorrelated. This procedure will be done following these steps: [6] [7]

1. **Standardize.**

   First, we want to have all our data to be standardized, in order to make the following step easier to calculate.

2. **Calculate the covariance matrix.**

   Now, we have to create a matrix which will be composed of the covariance of each one of the features, following this diagram:

$$\begin{bmatrix} cov(x_1, x_1) & \dots & cov(x_1, x_n) \\ \vdots & \ddots & \vdots \\ cov(x_n, x_1) & \dots & cov(x_n, x_n) \end{bmatrix}$$

3. **Find the eigenvectors and eigenvalues of the matrix.** To find the eigenvectors and eigenvalues, we need to solve the following equation:

$$[Covariancematrix] \cdot [Eigenvector] = [eigenvalue] \cdot [Eigenvector]$$

   With this, we will end up having as many eigenvectors as we need for the dimension of our data, which will vary depending on the experiment.

4. **Re-arrange data.**

   Once we have the new matrix, we multiply the original by the eigenvectors, which will re-orient the data, having the original matrix converted to a less dimensional one.

This will be done using sklearn python library, but will be explained later on.

2.4 Feature relevance

Even after reducing the dimensionality of our feature vectors, we can end up having information that doesn't give us meaningful information, so we will want to focus in the features that will help our program to give the best results, which will be measured using the accuracy of the predictions, as we will explain later.

In order to detect the most important MFCC we want to get, we will use algorithms based in decision trees, more precisely, Extra Trees.

Extra Trees algorithm (Extremely Randomized Trees) works similar to Random Trees but instead of choosing the best split from a random subset of the training set, they are chosen at random from the random subset for each tree. Apart from that difference, both help reducing variance, in expense of higher bias.

Given the number of features and how different two samples of the same genre can be, we don't really mind the increase in the bias if we can get, in return, a smaller variance, which can help training our program.

The algorithm works as follows: [8]

Split_a_node(S)
Input: the local learning subset S corresponding to the node we want to split
Output: a split $[a < a_c]$ or nothing
– If **Stop_split**(S) is TRUE then return nothing.
– Otherwise select K attributes $\{a_1, \ldots, a_K\}$ among all non constant (in S) candidate attributes;
– Draw K splits $\{s_1, \ldots, s_K\}$, where $s_i = \textbf{Pick\_a\_random\_split}(S, a_i), \forall i = 1, \ldots, K$;
– Return a split s_* such that $\text{Score}(s_*, S) = \max_{i=1,\ldots,K} \text{Score}(s_i, S)$.

Pick_a_random_split(S,a)
Inputs: a subset S and an attribute a
Output: a split
– Let $a^S_{\max}$ and $a^S_{\min}$ denote the maximal and minimal value of a in S;
– Draw a random cut-point a_c uniformly in $[a^S_{\min}, a^S_{\max}]$;
– Return the split $[a < a_c]$.

Stop_split(S)
Input: a subset S
Output: a boolean
– If $|S| < n_{\min}$, then return TRUE;
– If all attributes are constant in S, then return TRUE;
– If the output is constant in S, then return TRUE;
– Otherwise, return FALSE.

Figure 2.2: ExtraTrees Algorithm.

Chapter 3

Evaluation design

Now that we have explain all the methods we are going to follow, we are going to define how the experiment is going to take place.

In the following chapter, we will explain what dataset we will use to test our classifier, how we are going to divide it in smaller sets and what will they have, as well as what are we going after with our classifier, in terms of results.

Finally, we will explain what techniques and methods will be used in each experiment, as well as the parameters we want to try.

3.1 Dataset

For the realization of the project, we needed a large set of songs and genres to be able to train our algorithm in a proper way.

Initially, we wanted to use a relatively small amount of songs (100) of 4 different genres, all of them royalty free, taken from Free Music Archive[9]. The problem was that the set we ended up with was too small to make the program work as intended.

We decided to change the set to an already made one, so we looked for data sets build for our purpose and ended up finding Marsyas[10], a website in which we could find 1000 songs of 10 different genres, all of them 30 seconds long and

with a similar set of properties (which will be explained later). The genres in the dataset are some of the most common genres in music, which are as follow:

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

All the music in the data set is available for everyone and it can be used for investigation without any charge.
All songs are ".au" files, which is a format used by the program Audacity. To work with them, we need to know a few basics of digital audio, so I will explain what each one of the terms we will need when we extract the features of each song.

- **Audio frame:** Contains information in a given time.
- **Sample rate:** Number of samples taken from a continuous signal in order to produce a discrete signal.
- **Channels:** Number of streams in which the audio is sent.
- **Frame size:** Size of each frame. Sample rate * # of channels.
- **Frame rate:** Number of frames per second. Frame size/s.

In our data set, all songs have the following properties:

- **Sample rate:** 22050Hz
- **Channels:** 1 (Mono)
- **Frame rate:** 22050 fps

To make the program able to work with other formats and songs, we will take all this information when we extract the features.
This is accomplished forcing the load function from librosa to take the Sample Rate as 22050 and converting the signal to Mono-channel.

3.2 Evaluation protocol

Now that we have our dataset, we will explain how we are going to divide it in order to train our program. We will only use train and test sets, because we think adding a validation set will be useless in such a small dataset.

Considering this, our train and test set will follow 10-fold Crossvalidation, which will divide the original dataset in two smaller sets: the train set will have 90% of the songs; the test set, will have the remaining 10%.

Although this method is supposed to create these sets at random, we will always use the same sets, to be able to compare results between different methods and find which one is the best. Once we find which one works best, we will try it with other sets, to find a more fitting value.

To test the results, once we have trained our model, we will check if each one of the samples in the test set can be predicted correctly. With that, we will create a confusion matrix that will help us identify what genres are often mixed up.

The number of songs correctly classified will tell us how good our program is working.

3.3 Mehtods and parameters

Some of the steps we mentioned before are quite difficult to program so, in order to focus in the main experiment, we will use two already existing python libraries: librosa[11] and sklearn[12].

Librosa gives us the majority of audio analysis tasks already built in, so we only need to tweak the parameters we need to get the information we need out of every song.

From this library, we will only use two methods:

- **load:** This function loads the audio file, modifying the properties of the file we need to have all files following the same standards. The most important parameters we need are:
  - **sr:** changes the sample rate
  - **mono:** converts the file to mono-channel
  - **duration:** crops the song into a smaller length. The size of the matrix depends on the length of the file, so we need to make all songs last the same to work with them.
- **mfcc:** calculates the MFCC of the audio file we have loaded. The function automatically tweaks all the parameters it needs to make a small enough matrix, but without losing huge amounts of information.
  In this case, each interval is about 0.02 seconds long.
  From all the parameters that can be tuned, we only care about n_mfcc, which is the amount of MFCCs the algorithm will return. All the order parameters modify the properties of the song but, as we did already tune them with the "load" method, we don't need to do it now.

Knowing this, the experiments that we will carry out will be determined by the number of MFCC we calculate.

By default, MFCC returns 20 features for each interval, but this can lead to having data that won't give relevant information, as well as take a lot of time to compute in a laptop. For this reasons, we will use the following values for our experiments: 5, 10, 15 and 20 (in case it works best and we decide to keep using

this amount). We wanted to use values in between, but the time it will take to perform the test versus the improvement we could get makes it purposeless.

All the experiments will be done using these 4 values but, if we find that one of them has far better results than the others, we will stick to that value, to make experiments faster.

Sklearn, on the other side, will be used for all the algorithms involving machine learning.

The functions we will use from sklearn library are the following:

- **PCA:** this method will help us apply the method we explained before, to reduce the dimension of the matrix of values we have.
  The main parameter we are going to tune will be as follow:

  - n_components: number of components there will be after we apply PCA to our set.

- **SVC:** (Support Vector Classifier) this is the algorithm we are going use in most of the experiments to create a model which we can use to classify our dataset. It is part of the SVM module of sklearn and is the one we think will give the best results, considering our problem.
  To make it work properly with our dataset, we will have to tune the following parameters:

  - **kernel:** the kernel type we are going to use. We will perform an experiment with all the kernels sklearn offers us: rbf, linear, poly, and sigmoid, but will be explained later.
  - **C:** the penalty parameter. This will be the most important when using the 'linear' kernel, as gamma has not effect in it.
  - **gamma:** the kernel coefficient. It will allow us to tune the variance of the classifier.

- **fit:** fits the model
- **predict:** given a model and a sample, predicts its value.
- **ExtraTreesClassifier:** this class will help us classify the features by its relevance, using the method we explained before.

With PCA, we will only use one value, which will be 100, to fit the 10% of features it is recommended. We could try other values, but the time it would take to test every experiment with each different resulting matrix would be too long and won't give us enough improvement to justify it.

In terms of SVC, the parameters will be taken by trial and error, testing what value of each parameter (gamma for rbf, sigmoid, and poly; C for linear) works better for each feature representation and each kernel. How we test different values will be explained deeply later on.

There are others methods from the library that we will use, but most of them are implementations of algorithms we will use in our experiments, so we are going to only show them:

- **GaussianNB:** implements Naïve-Bayes
- **AdaBoostClassifier:** implements AdaBoost
- **cross_val_score:** will be used to check the accuracy of the AdaBoost Classifier.
- **LeaveOneOut:** Once we have our model, we will use this method to test the accuracy with all the dataset.

In this case, all parameters will remain untouched, as we want the results to be used as complimentary.

Chapter 4

Results

Once we have all the experiments set up, we can already start them.

In this chapter, we will show the results of each experiment we conduct, in which we will have a value based on the accuracy, which will be given by a percentage, as well as a confusion matrix that will show us which genres are most commonly confused.

The order of the experiments will be in the same order as the one we used to explain them, ignoring the results we get.

4.1 Classification results

The first experiment we will perform will be done using the naïve representation of the data, which is the experiment we expect worse results, given that the dimensionality of each song is way bigger than the size of our dataset.

Naive

For that, we will load all songs from our dataset, calculate the MFCC and store them in a numpy array, where each component will be a tuple containing the matrix that we have converted into a 1-dimensional array in the first component,

and the genre in the second, which will be an unsigned integer going from 0 to 9, following this:

0. Blues
1. Classical
2. Country
3. Disco
4. Hip hop
5. Jazz
6. Metal
7. Pop
8. Reggae
9. Rock

The extraction of the data will be done 4 times, to have 5, 10, 15 and 20 features, using the following code, which will be used taking each song of our dataset from each genre folder:

```
if filename.endswith((".au",".mp3",".wav",".aiff")):
    audio, fs = librosa.load(filename, duration=29.0, mono=True, sr=22050)#loads the first 30 seconds of the song
    mfcc = librosa.feature.mfcc(audio, sr=fs, n_mfcc=char) #calculates MFCC of the song
    #The size of the matrix will be number_of_MFCCs*1292, which is a feature every 0.02s
    #By default, it's 20*1291

    """The first comlumn gives useless information, so we remove it
    Works as offset"""
    mfcc = np.delete(mfcc, 0, axis=1)

    mfcc=np.transpose(mfcc)
    mfcc=mfcc.flatten() #converts the matrix into a 1D array

    char_gen.append((mfcc,genre[folder])) #adds the array to a list

    if cont%10==0: #Shows the percentage of completion of each genre
        print cont%100,
```

Figure 4.1: Code used to extract MFCC

Now that we have our dataset ready to work with, we have to create the train and test sets that we will use for the experiment.

sklearn needs two different train and test sets in order to work properly: one of them, will have the arrays of features, while the other will have, for each position of the first array, its genre.

As we said before, we are using 10-fold Crossvalidation, but we want all experiments to give us results we can compare. For this reason, we will divide the

original dataset using the following division:

```
def train_test(mfcc, char, n_set):
    train_mfcc, test_mfcc=[],[]
    train_genre, test_genre=[], []

    for i in range(1000):
        if (i+n_set)%10==0:
            test_mfcc.append(mfcc[i])
            test_genre.append(char[i][1])
        else:
            train_mfcc.append(mfcc[i])
            train_genre.append(char[i][1])

    train_mfcc = np.array(train_mfcc)
    test_mfcc = np.array(test_mfcc)
    train_genre = np.array(train_genre)
    test_genre = np.array(test_genre)
    return train_mfcc, train_genre, test_mfcc, test_genre
```

Figure 4.2: Code used to divide dataset into train and test sets.

Once we have our train and test sets, we can start classifying.

First, we will show a matrix with the best results we get from each experiment. Then, we will show each one of them and how we ended up having them.

Table 4.1: Naive representation with default parameters

n_mfcc	**rbf**	**Linear**	**Poly**	**Sigmoid**
5	15	38	38	10
10	15	44	41	10
15	22	47	39	11
20	24	46	41	11

As we can see, Linear gives us the best results with the default parameters, with both 15 and 20 features giving the highest accuracy, so these are the ones that we will use for the following experiments using this dataset representation.

The best accuracy comes from having 15 features and using linear kernel. The following confusion matrix, where each row represents the actual genre and the

columns, the genre it has predicted them to be, shows us better detail of it:

5	1	1	0	0	0	2	0	0	1
0	10	0	0	0	0	0	0	0	0
2	0	4	0	1	1	0	1	0	1
1	0	1	3	1	0	0	0	0	4
0	0	2	2	0	0	2	2	1	1
1	3	1	0	0	5	0	0	0	0
1	0	0	0	0	1	8	0	0	0
0	0	0	1	0	0	0	9	0	0
2	1	0	1	1	2	0	0	2	1
4	0	1	1	1	0	1	0	1	1

Figure 4.3: Confusion matrix of naive representation.

As expected, classical music is the genre that gives us best accuracy, given its difference between more modern genres, and it's also one of the genres other songs aren't usually confused to.

Pop also gives us great results, but genres such as rock and blues are the ones that most songs are predicted into, given that both are genres other genres evolved from.

Now, we will try tweaking gamma for rbf, poly and sigmoid methods and C for linear. The values will be taken by trial and error, trying different values until we find the best for each method and number of features. This will be done by creating different values, first, by powers of ten, between 10^{-10}) to 10^{5}, once we find the best, we should calculate the accuracy in a range of values around it until the accuracy remains constant.

Given that we are doing all the experiments in a laptop, the time it will take to do it would be too long, so we will only get the value from the first iteration.

Table 4.2: Naive representation using 15 MFCC with custom parameters

n_mfcc	**rbf** $\gamma = 10^{-7}$	**Linear** C=1	**Poly** $\gamma = 10^{-7}$	**Sigmoid** $\gamma = 10^{-9}$
15	50	47	42	34

Table 4.3: Naive representation using 20 MFCC with custom parameters

n_mfcc	**rbf** $\gamma = 10^{-7}$	**Linear** C=1	**Poly** $\gamma = 10^{-7}$	**Sigmoid** $\gamma = 10^{-9}$
20	47	46	44	34

As we can see, linear gave us the same results before and after tweaking its C, which made that, although its accuracy was higher at the beginning, one we start changing the gamma value, rbf gives us better results.

Now that we have found the best result using naïve representation, we will try to reduce its dimensionality using PCA, to find if we can improve it.

Given that, in all cases, poly and sigmoid kernels give us worst results, from now on we will only use rbf and Linear kernels, which will allow us to test more experiments with more values.

A we explained before, this will be done using the function given by sklearn library, which reduces the size of each song's vector to the amount of features we want. Once we have reduced the size of each vector, we can apply the same method as before, which gives us the following results:

Table 4.4: Naive representation with dataset and default values

n_mfcc	**rbf**	**Linear**
5	12	34
10	12	42
15	12	44
20	12	41

As we can see, the results are worse than with the raw data, which was expected considering that we are removing information from our feature vectors. rbf seems to be the method in which PCA has more effect, reducing its accuracy from more than 20% in some cases to 12% in all of them. Linear is also affected, but in a smaller degree.

Having this, we are going to try to tune the parameters of the classifiers, to see if we can improve these results.

Table 4.5: Naive representation with PCA using 5 MFCC and custom parameters

n_mfcc	**rbf** $\gamma = 10^{-8}$	**Linear** C=1
5	39	39

Table 4.6: Naive representation with PCA using 10 MFCC and custom parameters

n_mfcc	**rbf** $\gamma = 10^{-5}$	**Linear** $C = 10^{3}$
10	46	47

Table 4.7: Naive representation with PCA using 15 MFCC and custom parameters

n_mfcc	**rbf** $\gamma = 10^{-7}$	**Linear** $C = 10^{-2}$
15	46	45

Table 4.8: Naive representation with PCA using 20 MFCC and custom parameters

n_mfcc	**rbf** $\gamma = 10^{-3}$	**Linear** $C = 10^{-4}$
20	44	46

Histograms

Once we have tried with all the data, we are going to try the same experiments, but using the histogram representation, as we explained before.

The code of the features extractions will work similarly to the experiment before, but taking both, the minimum and maximum value of each MFCC, as we can see here:

```
if filename.endswith((".au",".mp3",".wav",".aiff")):
    audio, fs = librosa.load(filename, duration=29.0, mono=True, sr=22050)#loads the first 30 seconds of the song
    mfcc = librosa.feature.mfcc(audio, sr=fs, n_mfcc=char) #calculates MFCC of the song
    #The size of the matrix will be number_of_MFCCs*1292, which is a feature every 0.02s
    #By default, it's 20*1291

    """The first comlumn gives useless information, so we remove it
    Works as offset"""
    mfcc = np.delete(mfcc, 0, axis=1)

    maxs.append(np.amax(mfcc)) #gets the max value of the matrix
    mins.append(np.amin(mfcc)) #gets the min value of the matrix

    char_gen.append((mfcc,genre[folder])) #adds the array to a list

    if cont%10==0: #Shows the percentage of completion of each genre
        print cont%100,
```

Figure 4.4: Code used to extract MFCC and maximum and minimum values of each song.

As we can see, the main difference is that we don't reshape the matrix into a

1-dimensional array, as we need to know which values are from each feature, but instead, we take the maximum and minimum value of each song to, later on, find the values that we will use to calculate the size of the intervals of the histogram, dividing the range of all values by the amount of intervals we want.

To create each histogram, we have came up with the following code, which will take all the values from each feature and put it in its corresponding interval. This will be done using this:

```
def histogram(step, char, amount):
    histogram = np.zeros(amount)
    for c in char:
        histogram[int(c/step)] += 1.
    histogram = histogram/len(char)
    histogram = np.ndarray.tolist(histogram)
    return histogram
```

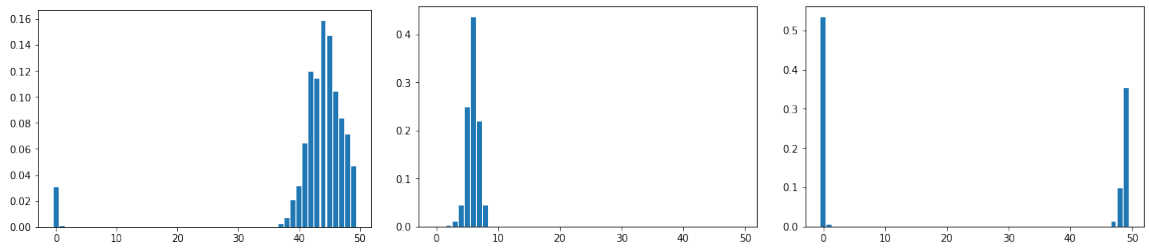
Figure 4.5: Code to generate the histograms of a song

Once we have all of them, we will join each histogram into a 1-dimensional array, shaped [histogram1][histogram2]...[histogramN]:

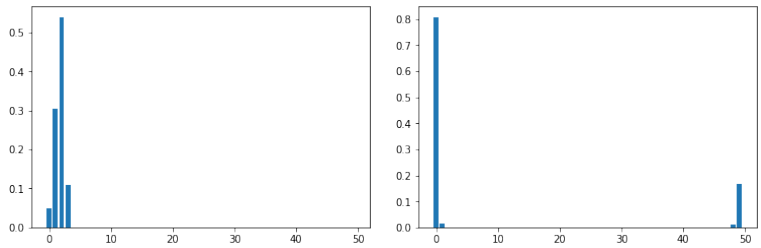
```
def more_hist(char, step, amount):
    hist = []
    temp_char=[]
    for c in char:
        tmp = c[0]
        temp = []
        for v in tmp:
            temp+=histogram(step, v, amount)
        temp_char.append(temp)
        hist.append((temp, c[1]))
    return hist, temp_char
```

Figure 4.6: Code to generate the histograms of all songs and create a new dataset

The histograms we will end up being similar to these:



(a) Feature 1 (b) Feature 2 (c) Feature 3



(d) Feature 4 (e) Feature 5

Figure 4.7: Histograms of a song with 5 MFCC

As we explained before, we will only use rbf and linear kernel, dividing the results in two different tables, each one for a different kernel, and using intervals in tens.

Using rbf kernel with default parameters, we get these results:

Table 4.9: Histogram representation using rbf kernel

n_mfcc	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**
5	24	37	33	35	34	35	31	32	32
10	24	37	37	33	35	35	33	31	31
15	24	37	37	35	37	33	31	31	29
20	24	37	37	35	37	32	30	31	30

Using linear, we get these:

Table 4.10: Histogram representation using linear kernel

n_mfcc	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**
5	25	41	41	45	41	41	40	41	41
10	25	42	44	50	50	50	48	49	51
15	25	42	44	53	51	50	54	57	59
20	25	42	44	53	52	51	54	55	57

Once we have tried the default values, we can start tweaking rbf's gamma and linear's C. In this case, we will only tweak the value with the amount of intervals that gave us the best results, because it would take too long to tweak those parameters for each one of the possibilities. As we can see, in all cases, except the first (40), the best case is to have 90 intervals which, if we tune the values, we get the following results:

Table 4.11: Histogram representation using 5 MFCC and custom parameters

n_mfcc	**rbf** $\gamma = 1$	**Linear** $C = 10^1$
5	55	48

Table 4.12: Histogram representation using 10 MFCC and custom parameters

n_mfcc	**rbf** $\gamma = 1$	**Linear** $C = 1$
10	61	58

Table 4.13: Histogram representation using 15 MFCC and custom parameters

n_mfcc	**rbf** $\gamma = 1$	**Linear** $C = 10$
15	70	60

Table 4.14: Histogram representation using 20 MFCC and custom parameters

n_mfcc	**rbf** $\gamma = 1$	**Linear** $C = 10$
20	65	60

As we can see, the amount of MFCC has to stay between 10 and 20, all of them giving the best results using rbf kernel and a gamma value. Now that we have this approximation, we will try to get a more specific value. For that, we will use 15 MFCC with 90 intervals and try to find a better accuracy by calculating it with gamma values between 0.5 and 1.5, which give us these results:

Table 4.15: Histogram representation using 15 MFCC with different gamma values

n_mfcc	$\gamma = 0.5$	$\gamma = 0.6$	$\gamma = 0.7$	$\gamma = 0.8$	$\gamma = 0.9$
15	66	67	68	68	69
n_mfcc	$\gamma = 1.1$	$\gamma = 1.2$	$\gamma = 1.3$	$\gamma = 1.4$	$\gamma = 1.5$
15	70	70	70	71	69

After that, we tried to tune more the value, but we found that variations in gamma don't really give us better results in this case so, from now on, we will be using $\gamma = 1.4$.

For further analysis, we will take a look at the confusion matrix of the last case, where we find this results:

5	0	2	0	0	0	2	0	1	0
0	9	0	0	0	0	0	0	0	1
0	0	8	0	0	1	0	1	0	0
0	0	2	6	0	0	0	1	1	0
0	0	0	1	5	0	1	1	2	0
0	0	0	0	0	8	0	0	0	2
1	0	0	0	0	0	8	0	0	1
1	1	1	0	0	0	0	7	0	0
1	0	0	0	1	0	0	0	7	1
0	0	1	1	0	1	0	0	0	7

Figure 4.8: Confusion matrix of histogram representation.

With this parameters, we can see now that most of the genres are predicted correctly, except Blues and Hip Hop.

Blues can be expected to be the hardest genre to classify, as is the one that more other genres have evolved from. As we see, it confuses those songs with Country and Metal, which is the most curious confusion, given the differences in vocals and the frequencies it tends to sound in.

Hip Hop gets mostly confused with Reggae, which is understandable considering that some rap scenes in Africa have been influenced by Reggae, taking some of its style in it.

Finally, we see that Rock remains as the genre that more songs are wrongly predicted to, as many features of Rock can be seen in other genres, either in its base or in small fragments.

Now, we will try to repeat the experiment but using PCA to reduce the histogram vector, as we did with the naive representation. In this case, this reduction will not be as big, as the size of the vector is way smaller, but it's still greater than the amount of samples in our dataset.

As before, in order to speed up the experiment, we will only try the experiments that gave us better results, starting with 15 MFCC and different amounts of intervals, which give us these values:

Table 4.16: Histogram representation with PCA using rbf kernel

n_mfcc	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**
15	24	37	37	34	38	36	34	31	30

Table 4.17: Histogram representation with PCA using linear kernel

n_mfcc	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**
15	25	42	44	53	51	50	54	57	59

As we can see, the results we get with the default values in SVM are virtually the same, with some changes using rbf, but within margin of error. We will now tune the parameters to see if we can get any improvement.

Table 4.18: Histogram representation with PCA using 15 MFCC and custom parameters

n_mfcc	**rbf** $\gamma = 1$	**Linear** $C = 10$
15	70	65

Considering the last two experiments, we can see that PCA does not affect the results if we are using histogram representation. This can be caused because we are already reducing the dimensionality of the data in a way that it has already removed most of the correlation between the songs, so the changes PCA can apply are almost non-existent.

With these experiments, we have found that the best accuracy is given using the histogram representation, with or without PCA, and with around 90 intervals

in each histogram. For classification, rbf kernel using a gamma = 1.4, we have the best possible accuracy of all the methods we have tried.

To confirm this result, we will try using other train and test sets, to see if it is consistent. This will be done by creating 10 different train and test sets and then testing the algorithm with them, which gives us pretty similar accuracies, as we can see here:

Table 4.19: Test of the final application with different datasets

set 1	**set 2**	**set 3**	**set 4**	**set 5**
61	69	74	64	75
set 6	**set 7**	**set 8**	**set 9**	**set 10**
71	72	65	71	70

Finally, we will try Leave One Out, which value will be an approximate accuracy of how the model will work in a more realistic environment, comparing each song with a model trained with all the other songs. With that, we get an accuracy of 69%.

Extra

To finish this experiments, we will try other methods to get other approximations to the problem, which will be: Naive-Bayes Classifier, Adaptive Boost Classifier (AdaBoost) and Leave One Out,to check what other algorithms will work with our problem. All of them will be tested with default parameters, given by sklearn library, following the characteristics we have found to be the most successful for our experiment.

First,we will use AdaBoost, which gives us a really low accuracy of only 18%. This can happen due to

Then, we will be using Naive-Bayes, which will give us an increase when we compare it with AdaBoost (56%), but still, we achieve lower accuracy than SVM.

4.2 Feature relevance analysis

Once that we have explained how we have built the model, as well as the algorithms and the results we have gotten, we are going to analyze what features are the most relevant. In order to do that, we will use ExtraTrees Classifier, which will give a percentage of how important each feature is in the classifier.

```
Feature ranking:
1. feature 3 (0.011840)
2. feature 0 (0.011585)
3. feature 0 (0.010474)
4. feature 3 (0.010153)
5. feature 1 (0.010096)
6. feature 8 (0.010084)
7. feature 0 (0.009707)
8. feature 10 (0.009629)
9. feature 2 (0.009549)
10. feature 1 (0.009488)
11. feature 2 (0.009082)
12. feature 0 (0.008644)
13. feature 3 (0.008399)
14. feature 3 (0.008375)
15. feature 11 (0.008210
16. feature 9 (0.008207)
17. feature 0 (0.008100)
18. feature 0 (0.008054)
19. feature 0 (0.007974)
20. feature 1 (0.007854)
21. feature 7 (0.007851)
22. feature 2 (0.007696)
23. feature 2 (0.007666)
24. feature 3 (0.007388)
25. feature 1 (0.007354)
```

Figure 4.9: Sample of feature relevance.

The algorithm returns the values given the position in the array, but we are interested in what feature of the MFCC gives the most relevant information, so we

will focus in that by dividing the array into as many pieces as features we have used, in this case, 15. One other thing to consider is that many of the positions in the array don't give information relevant to the model, so we will only consider those that give a relevance greater than 0%. Considering this, we have that only 274 of the 4,500 features are important to us, which are divided like this:

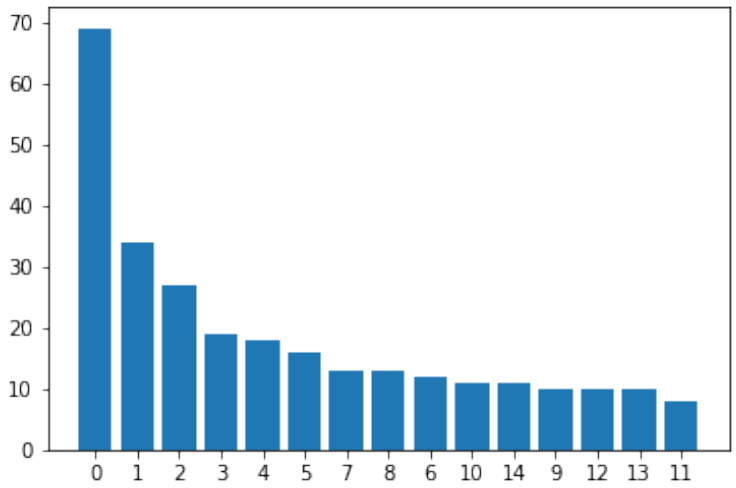


Figure 4.10: Relevance by feature.

In which every bar represents the amount of elements of this feature that are relevant in any form.

As we can see, the most relevant features are those in the first coefficients, which are the ones that represent lower frequencies. When we go higher, songs tend to not use these frequencies, as most instruments are unable to get that high.

Instruments that are typically used to measure rhythm, like bass or drums, also tend to use lower frequencies, and is also one of the elements that better represent the difference between genres.

Chapter 5

Conclusions

The objective of this dissertation was to achieve a model able of extracting the most relevant audio features of an audio track in order to classify different music tracks into its respective genre, using machine learning algorithms.

With our limited resources, we have been able to extract those features using MFCC approach, which classifies the frequencies that form each song into a scale that can be used to analyze its characteristics, which later where used to try different approaches, differentiated by how we used this data in our experiments, based on using either raw data or histogram representation of it, as well as reducing both using PCA.

With that, we then applied different versions of Support Vector Machine, both with default parameters and custom ones, that gave us a maximum accuracy of 50% using naive representation, and over 70% using histogram representation, which was both achieved using different values of gamma on rbf kernel.

Other algorithms were also used, but both of them (Naive-Bayes and AdaBoost) achieved worse accuracy than SVM.

If we wanted to improve even more our accuracy, we would need a bigger database, with which we could reduce underfitting, and a more powerful machine capable of trying different cases faster, to tune our model better.

Bibliography

[1] Audio Engineering Society

www.aes.org/aeshc/docs/audio.history.timeline.html

[2] MIR LAB

http://mirlab.org/jang/books/audioSignalProcessing

/speechFeatureMfcc.asp?title=12-2%20MFCC

[3] AIRCC Digital Library

http://aircconline.com/sipij/V4N4/4413sipij08.pdf

[4] Simon Fraser University

https://www.sfu.ca/sonic-studio/handbook/Mel.html

[5] Practical Cryptography

http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-

[6] PennState: Applied Multivariate Statistical Analysis

https://newonlinecourses.science.psu.edu/stat505/node/51/

[7] Sebastian Raschka

https://sebastianraschka.com/Articles/2014_pca_step_by_step.html

[8] Pierre Geurts, Damien Ernst, and Louis Wehenkel *Extremely randomized trees*. Liège, 2005.

[9] Free Music Archive

http://freemusicarchive.org/

[10] Marsyas

http://marsyasweb.appspot.com/download/data_sets/

[11] Librosa

https://librosa.github.io/librosa/

[12] Scikit learn

http://scikit-learn.org/stable/modules/classes.html