First Homework Assignment

of Computer Music Representation and Models

Anna Fusari
Filippo Marri
9 novembre 2023

General Description:

The aim of this homework is to create an algorithm able to classify music according to its genre. The main idea is to analyse the rhythmic features of a certain piece with a machine learning algorithm that has been trained with the songs contained in the dataset called GTZAN, unanimously considered as the database-cornerstone of automatic genre classification.

GTZAN Dataset

The first version of the **GTZAN** audio dataset has been compiled in 2001 by George Tzanetakis (G. Tzan) and Perry Cook when they laid the foundation for automatic genre classification with one of the most remarkable article that has ever been written on this subject: "Musical Genre Classification of Audio Signals". In such article, edited in 2002, they basically propose an accurate algorithm that has the same purpose as that of the problem of this homework (in their article, they consider also timber and melody actually) and they base their discussion on the tracks contained in GTZAN. Thus, let us see how this famous database is made.

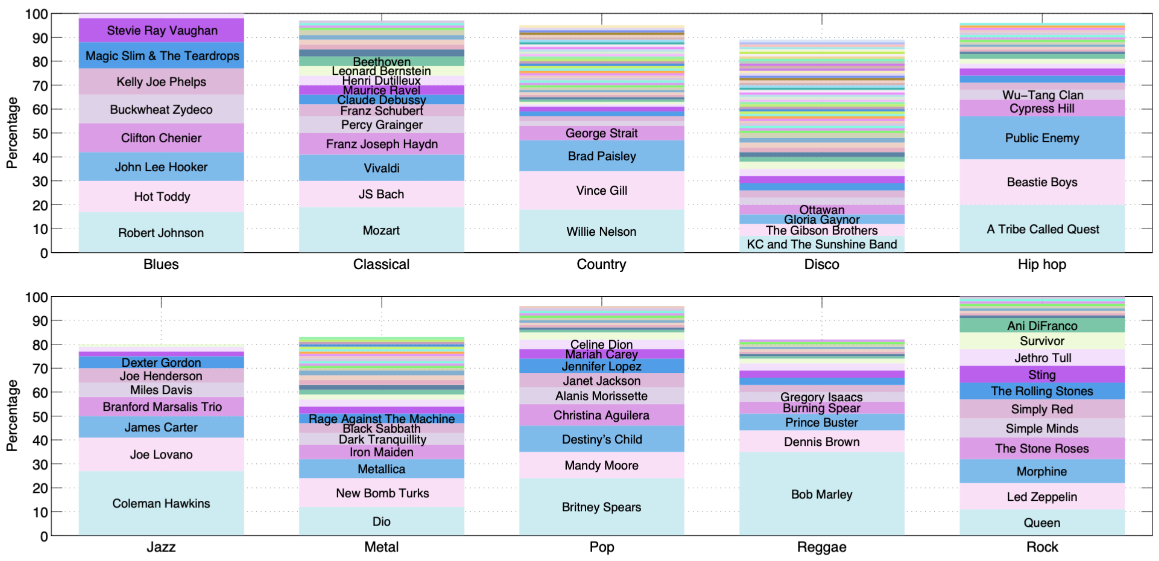
The database contains 1000 tracks of 30 second length where each track is a mono *.wav* audio-file sampled at $F_s = 22050\ Hz$. The *bit depth* (number of bit in which each sample is codified) is equal to 16 bit, therefore the *bit rate* (number of bit per second) will be equal to the product between the sample rate and the bit depth $F_S \cdot 16 = 22050 \cdot 16 = 352800\ \frac{bit}{s}$.

The songs are classified by 10 genres in groups made up by 100 elements. The genres are, in order:

0. Pop (some examples of what we can find under this label are *Every Breath You Take - Live Version* by Sting, some songs of Britney Spears, *Cloudbusting* by Kate Bush)
1. Metal (Dio, Metallica, Iron Maiden, Black Sabbath, …)
2. Classical (Some parts from "*Le Quattro Stagioni*" by Antonio Vivaldi, some sonatas - the eighth by Beethoven and the sixteenth by Mozart -, some orchestral pieces like Rhapsody in Blue by George Gershwin and so on)
3. Rock (Queen, Led Zeppelin, Rolling Stones, Jethro Tull, …)
4. Blues (Stevie Ray Vaughan, Robert Johnson, …)
5. Jazz (Coleman Hawkins, Miles Davis, …)
6. Hip-Hop (Beastie Boys, Public Enemy, …)
7. Reggae (Bob Marley, Dennis Brown, …)
8. Disco (*Get Down Tonight, Knock on Wood* by Amit Stewart, some songs by ABBA, some by Boney M., *Disco Inferno* by The Tramps and others)
9. Country (Willie Nelson, Vince Gill, …)

As it can be seen from the examples reported for every genre, we could say that this database is a very open-minded one. In fact, each group is not the stereotype version of itself but it has been created choosing, of course songs that characterised the genre, but also songs that are on the borderline between two genres. We can watch, for instance, at the pop genre: the main ones are Britney Spears's song but we have also Cloudbusting by Kate Bush that, in some part, it is closer to a classical piece than it is to a Britney Spears song.

To better understand how each group is organised, we can watch at the following histogram taken from the article "*The GTZAN dataset: Its contents,* *its faults, their effects on evaluation, and its future use"* by *Bob L. Sturm* where we can see, in percentage, how many songs of a certain author are chosen to create a group.



Artist composition for each genre. The missing elements in the columns (for the ones that not reach 100%) are songs that have not been identified by the author of the cited article.

Code analysis:

First of all we import all the libraries, the objects and the function that we need:

- **os:** this library allows us to work with operation of the operative system. We will use it to save the models in local.
- **libROSA:** it allows us to work with audio files.
- **numpy:** renamed **np** for practical reasons, it allows us to work with arrays.
- **matplotlib.pyplot:** renamed **plt**, is a procedural interface of the library matplotlib that allows us to plot graphs. The library is modelled closely after matLAB. Therefore, the syntax of the majority of plotting commands is similar matLAB's ones.
- **deeplake:** importing deeplake, we have the possibility to exploit the deeplake platform to work on datasets without download them locally.
- **pickle:** it is a library necessary to save and load the models.
- **tqdm:** we import the tqdm method from the library **tqdm.notebook**. It is essentially a function that provides a progress bar (designed in HTML, nicer to see) of a certain process.
- **MinMaxScaler:** it is an object of the library **sklearn.preprocessing** that allows us to transform features by scaling each feature to a given range.
- **SVC:** that stands for Support Vector Classification is the classifier that we import from the library **sklearn.svm**.
- **metrics:** it is a module of **sklearn** where we can find methods to compute the accuracy of our algorithm. Furthermore, we can find there also the methods to evaluate and display the confusion matrix.

Question One:

First of all, we import the dataset by loading its path on a variable called ds. In this way, we can access to the dataset, still stored online, working on that variable. It should be noted that we access to the dataset in read-only mode, given the fact that we don’t owe the permissions to write there.

Then, we use the method summary() to see how the database is organised. What we discover is a very clear hierarchical structure: we can watch at the dataset as a container of two other containers called tensors. In the former, the audio tensor, we find the 1000 audio tracks, while in the latter, the genre tensor, we find the list of genres.

Each tensor contains, in turns, other information organised in a sort of matrix fashion. We report in a table the inner organisation of each tensor.

Tensor	**HType** *(Class of tensor)*	**Shape**	**Data Type**	**Compression**
Audio	Audio	(1000, 66000:675808,1)	Float64	.wav
Genre	Class_label	(1000,1)	Unit32	None

To better understand how every element of the tensor is organised, we report the contents of the first element of the two arrays that we extract with the method data().

ds.genre[0].data()

```
{'value': array([0], dtype=uint32), 'text': ['pop']}
```

ds.audio[0].data()

```
{'value': array([[ 1474.],
        [ 1489.],
        [ 1479.],
        ...,
        [11993.],
        [13501.],
        [14854.]])}
```

We firstly focus our attention on the result of the extraction from the audio tensor. Watching at this result and knowing that the shape of an array is the number of elements in each dimension, to clarify how this tensor is organised we can think it as a cube where each position is identified by three coordinates: that's what in algebra is called tensor. By extracting *ds.audio[0].data()* we have fixed one of this three coordinates obtaining a result in **two** dimensions even though one of this two dimension is equal to one. If we want to be simple, we can think this result as a column vector concluding that the tensor audio is a sort of an irregular parallelepiped squeezed in two dimensions but defined on three. We call it irregular because, as we can see printing its shape, one of the two dimension has a size that can vary from 66000 to 675808 depending on the track.

Furthermore, to access the element identified by one triplets of the coordinates, (e.i. *ds.audio[0,1,0].data()*) we have to specify the key by which acceding to the element. In this case, there is only one key but we still have to specify it in order to extract only the value or else we obtain anyway that element but introduced by its key as it is reported in the following examples:

Command	print(ds.audio[0,1,0].data())	print(ds.audio[0,1,0].data())['value']
Result	`{'value': array(1489.)}`	`1489.0`

For the genre tensor is exactly the same but in two dimensions so we can think it as a rectangular squeezed in a line where each element is identified by one coordinate since the other one can be only 0. In other words, a column vector. Let us see it with an example: if we extract what is contained in the first element of the tensor "genre" by printing ds.genre[0].data(), we obtain what it follows:

```
{'value': array([0], dtype=uint32), 'text': ['pop']}
```

In order to extract one of these three things related to the first elements we have to add, in square brackets, the key that allows us to access the specific value.
e.i. writing `ds.genre[0].data()['text']`*, we obtain as output just* `['pop']`.
We can look at the genre tensor as a matrix whose elements can be seen in turn as vector where each element is identified by a word called key.

We note that the extrapolation made by data() is different for the two tensors: this is because data() returns data in the tensor in a format based on the tensor's base htype. By using the key "value" to extract the data we obtain the same result as if we use the function numpy(), while using the key "text" we obtain just a string.

A typical machine learning algorithm is trained with the 80% of the elements of a dataset and tested with the remaining 20%. Given the fact that Jupyter Notebook has not enough space to store all the tracks of the database, we create two subsets from the original dataset. We start creating two lists of indexes: one for the training *(train_index)* whose indexes will pick one track every 10 and one for the test *(test_index)* whose indexes will pick one track every 52. In this way we will respect the proportion but we will work with a dataset made up by 100 tracks instead of 1000.

We now create two other lists, one for training *(genre_train)* and one for testing *(genre_test)*, where we store the elements picked from the dataset by using the previously defined list. To work with a smarter syntax we use list comprehension method instead of a for loop. We print the shape of the lists to see whether or not we achieved the requested result.

At this point, we extract the audio tracks to finally create the subset. We use a more complex list comprehension that will be illustrated in detail. We analyse just the audio_train one since the process is exactly the same for the other.

The line code is the following:

```
    audio_train = np.array([ds.audio[i].data()['value'][:n_samples][:,0]
for i in tqdm(train_index)], dtype = None)
```

Let us, for now, focus on the ds.audio[i].data()['value'][:n_samples][:,0] part to understand what it returns. We are working on the i-th element of the tensor *audio* of the dataset, so we are selecting one column of the irregular parallelepiped. Then, we extract it from the dataset with the command data() and we chose the value key. What we obtain is a **two** dimension numpy array that has a shape equal to (x, 1) where x is the dimension of that specific column of the parallelepiped so the number of samples in which the track is sampled. In other words, the samples of the track organised in column vector shape where each column has its own size. However, we still have to abide by the request that every track must be long 29 seconds. Knowing that the track are sampled at $F_s = 22050\ Hz$, we crop every vector picking up just the first n_samples where n_samples is equal to $n\_samples = F_s * 29s$ with the instruction [:n_samples]. In this way, we obtain a **bi**-dimensional array where each track lasts perfectly 29 seconds. Another request is to obtain not bi-dimensional arrays, but mono-dimensional one so, in our case, we have to reshape the column vector into a row vector. The faster way, it is to impose that everything that concern the third dimension has to be equal to 0. In this way, the shape of our vector of samples will perfectly fit the request: `(639450,)`. At this point, we iterate the operations for every element selected by the list *train_index* with a list comprehension structure. We add the operator tqdm that stays for تقدّم (progress in Arabic) to get an HTML bar that shows us the progress of this operation that is a bit long. The result of this list comprehension will be transformed into an array by the np.array method that receives as first argument the result of the list comprehension and as second argument the data type: none for audio files.

Another way, faster when executed, to write this line could have done using the function enumerate(), but it has been preferred the explained solution that allows to better understand each step of the process.

Now, we print the shape of the two new-born arrays to check if they are as it is requested and we plot the first audio track to check if the array is perfectly loaded and if everything has worked correctly. For this point, we define a general plotting function that will come in handy in other points of the code.

Question Two:

First of all, we normalise the tracks in order to keep their dynamic into the interval $[-1,1]$. In this way, we improve the generalisation capability of the algorithm since the model has to work with audio files that shares the same codomain. The reason why the generalisation capability gets better is still unknown but, it has been observed from empirical results that the data-normalisation can improve the accuracy of a deep learning architecture. A possible explanation may be that elements that shares the same codomain are easier to compare but, at the moment, there are no mathematical inferences that sustains this thesis.

Anyway, we perform this kind of operation by creating a new object *MinMaxScaler* set with a feature_range equal to the desired interval. This object offers the possibility to scale a certain feature to a given range. The transformation is given by the following formula:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (1 - (-1)) + (-1)
```

The first line is just the formula for the normalisation between 0 and 1: each element of the vector is translated of a quantity equal to vector's minima and then divided for what we can call, borrowing the word from signal analysis, the vector's dynamic. *X_scaled* is simply the standardised vector (*X_std*) extended in the new range and re-centred around its zero summing the minimum extreme of the scaling interval. This procedure is triggered by calling the method *fit_transform()*.

We plot now the first sample calling the plot function defined before. We note that the track is so much amplified that seems to clip. We will analyse this sound wave later.

We start now writing the code for the novelty function. We remind that a *novelty function* is a time-function that shows a peak whenever an event occurs in the track. It is often defined to identify the onsets of a song. These are the exact instants where we can find musical changes (e.i. a note is played, a kick is hit). There are several procedure to find a novelty function given that there is not a univocal definition of it.

So we proceed analysing the preliminary section that we need in order to define the novelty function.

Firstly, we define a function that compute the local average: this function takes as inputs a vector x, where the signal is stored, and a parameter M that represents the width of the window given in samples. We initialise the vector *local_average* by creating an array filled with a number of zeros equals to the length of the vector x. Then we evaluate the local average considering an interval from -M to M cropped in correspondence of the edge of the vector support when the window overcome them. This function will return a mono-dimensional array, so a vector, where the n-th element of such vector is equal to the local average of the signal x evaluated, with a window of size M, in correspondence of the n-th sample of the signal x. Simply speaking, it is just a smoother version of the signal x.

We define then the *principal_argument* function that will implement the expression $y = \left(x + 0.5\right) \% \, 1 - 0.5$ that we will use to map the phase derivative into a range that goes from -0.5 to 0.5. In the code, we use the numpy function *mod()* considering that we have to evaluate the result of the expression element-wise.

We define now the novelty function. This function takes as argument:

- The signal **x** stored in an array;
- The sampling rate F_s;
- The window size **N** using to evaluate the STFT;
- The hop size **H**, so the step size in which the window of the STFT is to be shifted across the signal. Simply speaking, how much we can advance the analysis time origin from frame to frame;

- The size **M** of the window of the local average function;
- A boolean value called **norm** that, if it set true, normalised the result of the novelty function to the unity;
- A boolean value called **plot** that, if it set true, plot the novelty function at the end of the evaluation.

Let us now analyse the function's definition.

Since we want to find an algorithm that recognises the changes in phase (this is a rough explanation of what a phase-based novelty function does), we firstly evaluate the STFT just using the function from the library libROSA in order to extract the phase from its result. We briefly remind that the STFT is nothing but a DFT (actually, the FFT implementation) evaluated not on the entire time axis but just on a portion selected by a series of overlapping windows defined as $w\ (n)$. The size of the portion of the signal under the window that is not overlapped by the next window is chosen by the hop-size.

We report the definition:

$$\mathcal{X}(m,k) := \sum_{n=0}^{N-1} x(n+mH)w(n)e^{-2\pi i \frac{k}{N}n}$$

The function of libROSA takes as input the signal, the width of the window, the hop-size, the window length and the type of window. For this last parameter we choose an Hann Function in order to avoid the artefacts that a classic rectangular window would introduce. It returns a complex-value matrix with the coefficients of the STFT of the signal x. We save the result in the array *x_stft*. To understand how this matrix is organised we can think that each element of the matrix represents the complex coefficient of the STFT of the signal evaluated for a certain frequency (row) and in certain portion of time (column).

Secondly we compute the sampling frequency of the novelty rate. We just have to divide the sampling frequency of the signal F_s to the Hop Size keeping in mind that each result of the STFT is evaluated for the given Hop Size, so the number of samples per second in one hop-size will be $\frac{F_s}{H}$.

Then, we extract the phase from the STFT using the numpy function angle that returns, putted as input the matrix x_stft, a matrix that will contain the phase of each coefficient of the x_stft. We have to divide each element of the matrix by 2π because we want to normalise the phase interval to 1 and not to 2π as it is returned by the np.angle() function.

We evaluate now the proper novelty function. Since we want to see when something changes in the phase, we derive it. We remind that a derivative can be seen as how much a certain function vary around a certain point so, simply speaking, absolutely higher is the value of the derivative of the phase, stronger is the change in the phase function. Reminding that a continue derivative is defined as

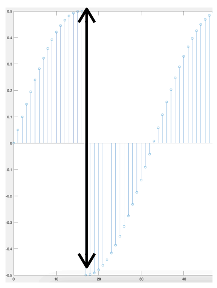
$$f'(x) := \lim_{h\to\infty} \frac{f(x+h)-f(x)}{h}$$

and given the fact that in discrete domain the minimum distance between two elements is equal to 1, we have

$$f'[n] := \lim_{h\to\infty} \frac{f[n+h]-f[n]}{h} = f[n+1]-f[n]$$

Hence, we can evaluate the derivative with the function *np.diff()* of numpy that returns the difference between two consecutive elements: exactly what we want. The term axis = 1 specifies the dimension upon which the operation is executed. In our case, along the time direction.

Then, we evaluate the principal argument of the result of the previous point. The function *principal_argument* will map the values of the derivative between -0.5 and 0.5. We do this kind of operation to deal with the phase discontinuities due to the fact that the phase is periodic.



Let us briefly see what happens.

The matrix of the phases contains values between -0.5 and 0.5 since we have normalised over 2π the result of np.phase that returns values between $-\pi$ and π. Considering how we defined the operation, the derivative of that vector will contain elements that will be included into the interval $(-1,1]$. We note that we can reach those extreme values just if there is jump in the phase equal to 1 in absolute value (so equal to 2π if we don't consider the normalisation). This discontinuity is present because the phase is periodic so every time that the phase goes over π (or below $-\pi$) there is a jump, given that the next sample starts from $-\pi$ (π). With the function *principal_argument()* we take the codomain of the phase derivative and we shift it of 0.5 obtaining $(-0.5,1.5]$. Then, we take the reminder of the division between the element and one obtaining a matrix whose elements are included in $[0,1)$. At this point we subtract 0.5 obtaining $[-0.5,0.5)$. As verification, we can print the minima and the maxima of the phase matrix of the first track noting that these values are very close to the boundaries of the codomain that we have said.

`np.min(x)`	`np.max(x)`
-0.999945645626568	0.999830998055425

`np.min(x+0.5)`	`np.max(x+0.5)`
-0.499945645626568	1.49983099805543

`np.min(np.mod(x + 0.5, 1))`	`np.max(np.mod(x + 0.5, 1))`
0.0	0.999999871269063

`np.min(np.mod(x + 0.5, 1) - 0.5)`	`np.max(np.mod(x + 0.5, 1) - 0.5)`
-0.5	0.499999871269063

So, as we see, this method is a sort of correction for discontinuities.

We repeat exactly the same operation to find the second derivative. After this sequence of operation, the final array will have peaks in correspondence of the value of the duplex of frequency and time where the derivative of the phase changes. Assuming that in correspondence of a changes in phase the behaviour of the phase signal is almost linear, the second derivative will show a peak in correspondence of that point.

We now sum all the contributes given by each frequency in order to obtain a phase-based novelty function that is just function of time.

We subtract the local average in order to emphasise the peaks and we apply the half-wave rectification by setting all the negative values equal to zero.

We write the function for normalising and plotting and, at the end, we return the result.

After having defined the function, we test it with the first track.

Let us now understand how to read the novelty phase-based function. Let us suppose to have an increasing behaviour of the phase of our signal in a certain window. This means that the rhythm of the song is speeding up in that portion of time. If we derive two times the phase signal, we obtain a function with a sort of peak in correspondence of the first sample of the window (assuming that the first derivative of the phase could be almost constant given that the window is so narrow and each portion of the phase selected by the window could be seen as linear). So, higher is the peak, higher is the speeding up. Following the same reason, we could say that lower (higher in module but negative) is the peak, higher is the rallentando. We have to remind that we take the absolute value, so we don’t know if each peak represents an absolute speeding up or an absolute slowing down. We can only detect the relative behaviour respect to the previous sample.

In light of this, we can try to analyse which is the influence of each parameters of the novelty function.

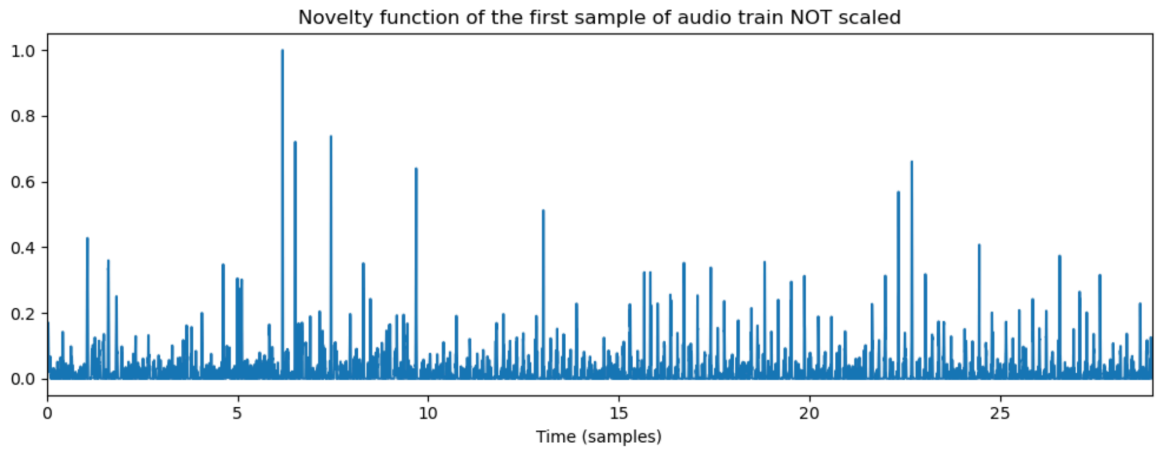
Type of window:

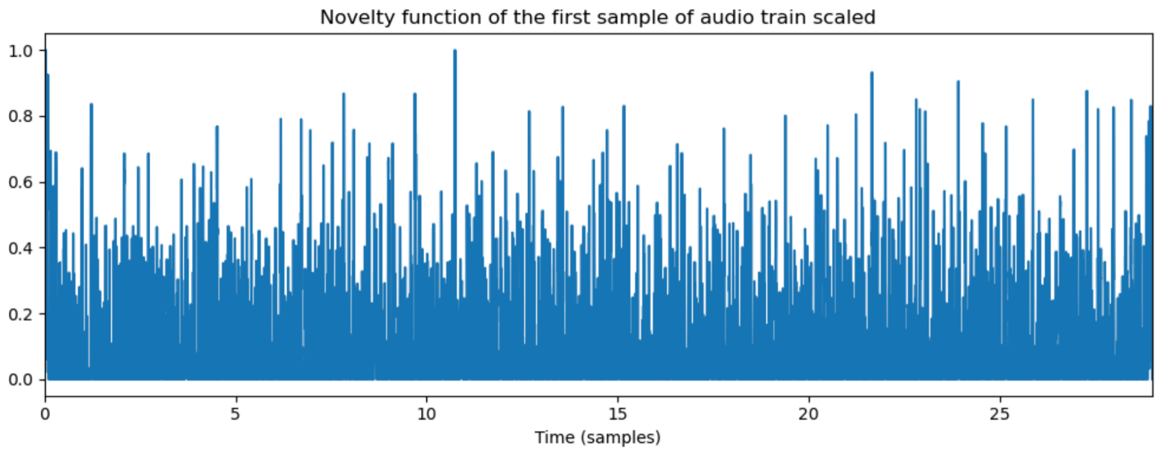
Firstly, we consider the type of window: sharper are the border of the window, preciser is the cropping but the more will be the artefacts introduced.

Hop size and window size:

We can also note how the hop size has a very strong influence on the result that we obtain: an higher hop-size, will give less overlap that, in turns, will give more analysis points and therefore smoother results along time. We have to underline that the role of H is played hand in hand with the one of the window size: if we increase the value of N we take into account more samples on which evaluate the STFT. This means that on the one hand we will work on a larger number of frequencies but, on the other hand, that the evaluation of the STFT will be less specific giving that we are substantially reducing the time resolution. It is generally better working with a larger number of frequencies because the frequency mean value evaluated on a larger number of frequencies it's surely closer to the frequency mean value of the whole song. To compensate the reduction of time resolution we can reduce the value of H: choosing a smaller hop-size we evaluate the STFT on a smaller number of samples at a time. Hence, it's better choosing relatively high values for N and relatively small values for H. However, we have to pay attention not to be too much extreme: if we choose a value of N that is too much high and a value of H that is too much low we lose accuracy, therefore all the peaks will be shorter. We do not appreciate this fact given that a small peak can be confused with "false positives" peak (so a peak owed to noise and not to an actual onset). We conclude evincing that a good trade-off is the perfect solution.

Keep talking about noise, we can note how the reshaping influences the noisiness of the novelty function: if we do not rescale the function, we obtain a signal with a very low noise contribute but, on the other hand, in that novelty function the spikes are not well defined. This phenomena is evident in the two following pictures where in the former the signal noise-ratio is low (especially from 10 seconds on), while in the latter the peaks related to the onsets are more defined and more distinguishable from the one related to noise.





Furthermore, introducing noise could be a good idea also to refine the algorithm. In some situation the on purpose addition of noise could be a preprocessing technique – applied to all the tracks before a model learns and makes inferences – or an augmentation technique, where it is applied only to tracks in the training set for strategic variation.

Mainly, noise is best suited for augmentation. In our specific case, could be useful to increase variability of some tracks with the aim of training to mitigate adversarial attacks and avoid overfitting. An easy and naive way to understand this concept is to think about what is happening with images: if we train an image identification algorithm just using perfectly defined images it will be more difficult for it to deal with new images where the subject is not identical to the image used for training. If we sully the image with some noise, the image becomes like what we see on the screen of an anagogic television when the signal is not optimum. The algorithm learns how to "abstract" the subject from the dirty image and, in this way, it is easier for it to detect that same subject in new images.

Question Three:

We call now the function that we have defined in the previous point into a new one in order to create a vector that will contain all the features related to a certain track.

Firstly, we compute the phase-based novelty function using the function defined in the previous point.

Then we evaluate mean value and standard deviation of the novelty function. Let us dwell on the meaning of this values related to the novelty function. If the mean value of the novelty function is close to the max of the novelty function, it means that we have events in rhythm that appears very often. We can expect such result from a metal piece like Muster of Puppets by Metallica or things with that kind of aggressive and fast-paced rhythm. For a piece like that one, we expect that the standard variation will be around zero given that, in certain part, the rhythm is all the same. On the other hand, a jazz piece like Take Five, will also have a very high mean value but with a high standard variation too because the rhythm scheme is still heavily rich but also variegate. On the contrary, for a piece like an adagio played by a string quartet, the mean square will be lower with a not so much high standard deviation.

After that, we evaluate the Tempogram. It is a tridimensional array that presents a certain portion of time along the zero dimension, a certain value of tempo measured in BPM along the one dimension and, for last, the quantification of how much the tempo tendency, specified by the one-coordinate, is present in the certain portion of time, specified by the zero-coordinate.

We also find the *zero_crossing_rate* of the signal that is the rate at which a signal transitions from positive to zero to negative or vice-versa.

It is defined as:

$$zrc = \frac{1}{N-1} \sum_{n=0}^{N-1} 1_{\mathbb{R}_{<0}} \left(x_n \cdot x_{n-1}\right)$$

where N is the length of the signal and $1_{\mathbb{R}_{<0}} \left(x_n, x_{n-1}\right)$ is an indicator function. It maps the elements of a certain subset to 0 or 1 according to the condition expressed to the 1 subscript. We can see its definition as:

$$1_{\mathbb{R}_{<0}}(x[n] \cdot x[n-1]) = \begin{cases} 1, & \text{if } (x[n] \cdot x[n-1]) \in \mathbb{R}_{<0} \\ 0, & \text{if } (x[n] \cdot x[n-1]) \notin \mathbb{R}_{<0} \end{cases}$$

It returns one if the sign of a certain sample is opposite respect to the sign of the signal at the previous sample. The librosa function takes as input also the hop-size to evaluate it. The ZCR provides us information about how many times the signal cross the zero over its duration. This is a rough way to estimate the instantaneous frequency of the signal. Also in this case, we find mean value and standard deviation. For a similar reason to the one previously exposed for the the novelty function, we can say that the jazz and classical music genre songs have generally low ZCR values mean values, while Pop and Metal music genre songs have high ZCR.

We introduce also the spectral flux. It measures the spectral changes between two consecutive samples. it is defined as the L2-norm between two normalised spectra of two consecutive samples.

$$Fl_{(i,i-1)} = \sum_{k=1}^{N} \left(EN_i(k) - EN_{i-1}(k)\right)^2$$

where $EN_i(k) = \frac{X_i(k)}{\sum_{l=1}^{N} X_i(l)}$ so the magnitude of the k-th sinusoid of the DFT of the signal x at the sample i normalised over the sum of all the magnitudes. We evaluate mean value and standard deviation also for the spectral flux.

As last feature, we add the tempo.

After we defining it, we create a function that creates the feature vector for the audio train set and for the audio test set. We prefer to create a function because it will become useful also for the next points. Also in this case we use the tqdm function to see in real time the advancement of the whole operation.

At the end we check the shape of both the feature vectors. Their first dimension is equal to 93169, reasonable result given that we have placed all the features one after the other allocating:

- 4996 cells for the novelty function;
- 1 for the sampling frequency of the novelty function;
- 1 for the mean value of the novelty function;
- 1 for the standard deviation of the novelty function;
- $2048 \cdot 40 = 81920$ for the tempogram;
- $1 \cdot 4996 = 4996$ for the zero crossing rate;
- 1 for the mean value of the zero crossing rate;
- 1 for the standard deviation of the zero crossing rate;
- 1249 for the spectral flux;
- 1 for the mean value of the spectral flux;
- 1 for the standard deviation of the spectral flux;
- 1 for the beat.

Summing, all these contributes we obtain exactly 93169.

On the base of the reason previously exposed for the noise, we can affirm that the feature vector must not be too much precise in the description of a song. If we add too much features, the description becomes too much accurate. So, if we don't preserve a certain degree of ambiguity by creating a vector that over-describe the track, the algorithm is not able to find the common elements between tracks and it risks to be too pernickety.

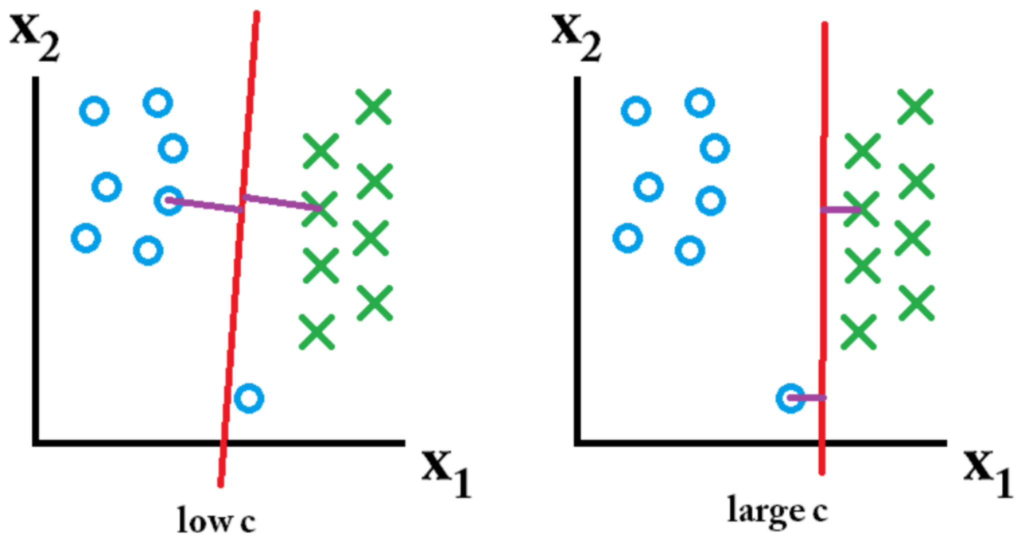
In a formal way, we call the incapacity of generalisation *overfitting*. We use this term to underline that the model *fits too closely* to the training dataset.

Question Four:

After this preparation part, it is high time we started creating our model. To do this, we train a Support Vector Machine: an algorithm developed by Владимир Вапник at the Bell's Laboratory from 1964 to 1995, that organises the objects into classes by defining a kind of demarcation line between their position in a multidimensional space. More precisely, it finds the best hyperplane (a subspace that has a dimension equal to one less the dimension of its ambient space) that maximise the distance between the end vectors that support it.

The strength of this algorithm is in its versatility: it is possible to change and even define different type of Kernels. But, what is it a Kernel? The word kernel means core, in German. It is in fact the core of the algorithm since it is the method that mapped the elements in the multidimensional space. Actually, "mapped" it is not the most precise word given the fact that the kernel methods do not properly do what a function does; they evaluate the inner products (that is the multidimensional version of the scalar product) between the images of all pairs of data in the feature space. The trick of using a kernel method instead of a proper function significantly decreases the computation cost.

So, we define our model choosing a linear kernel and a factor C equal to one. This means, very naively, that if we imagine our datas like a series of dots on the plane, we try to separate them into categories with straight lines. The hyperparameter (a parameter whose value is used to control the learning process) C tells to the SVM optimisation how much we want to avoid misclassification of each training example. We can understand how it influences the model by watching the following picture.



A low value of C allows the SVM to divide the space in order to find an hyperplane with the largest minimum margin. On the other hand a higher C allows to separate as many instances as possible. Given the fact that it is not likely to obtain both things, we have to find a good trade-off.

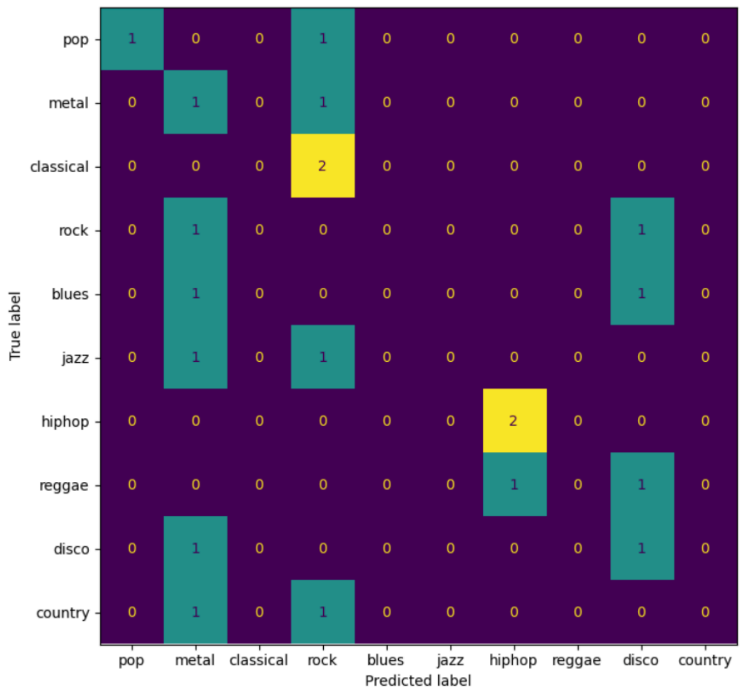
After creating the model, we fit it with the training set.

Then, we create a function that will save or create new models and load them. At the end we create a function that prints the accuracy of the model, so how much, in percentage, the prediction genres for one song corresponds to the right one.

For the training set, the accuracy is equal to 100%. This could be risky in terms of overfitting for a reason that will be explained later.

Question Five:

At this point, we are ready to perform classification. In order to do this, we use the predict method of the SVC model created before. It takes as input the data that will be classified: the feature vector of the audio test set. The function returns the labels of the data passed as argument based upon the trained data obtained from the model. Thus, in our case, the list of the genres of the 20 songs. After that we print the accuracy obtaining, as result, 25%. This is not a good value since the model has correctly classified just five songs out of twenty. Let us try to understand which are the mismatches by exploiting the confusion matrix. To visualise it in a most intuitive way, we print it with the method *ConfusionMatrixDisplay*. This matrix is graphical table that allows us to read where the model has classified the song in relation to their true genre. In the cells, identified by two coordinates, we can read the number of songs of the genre identified by the label relative to the row coordinate that has been classified by the model into the genre identified by the column coordinate. An example will follow:

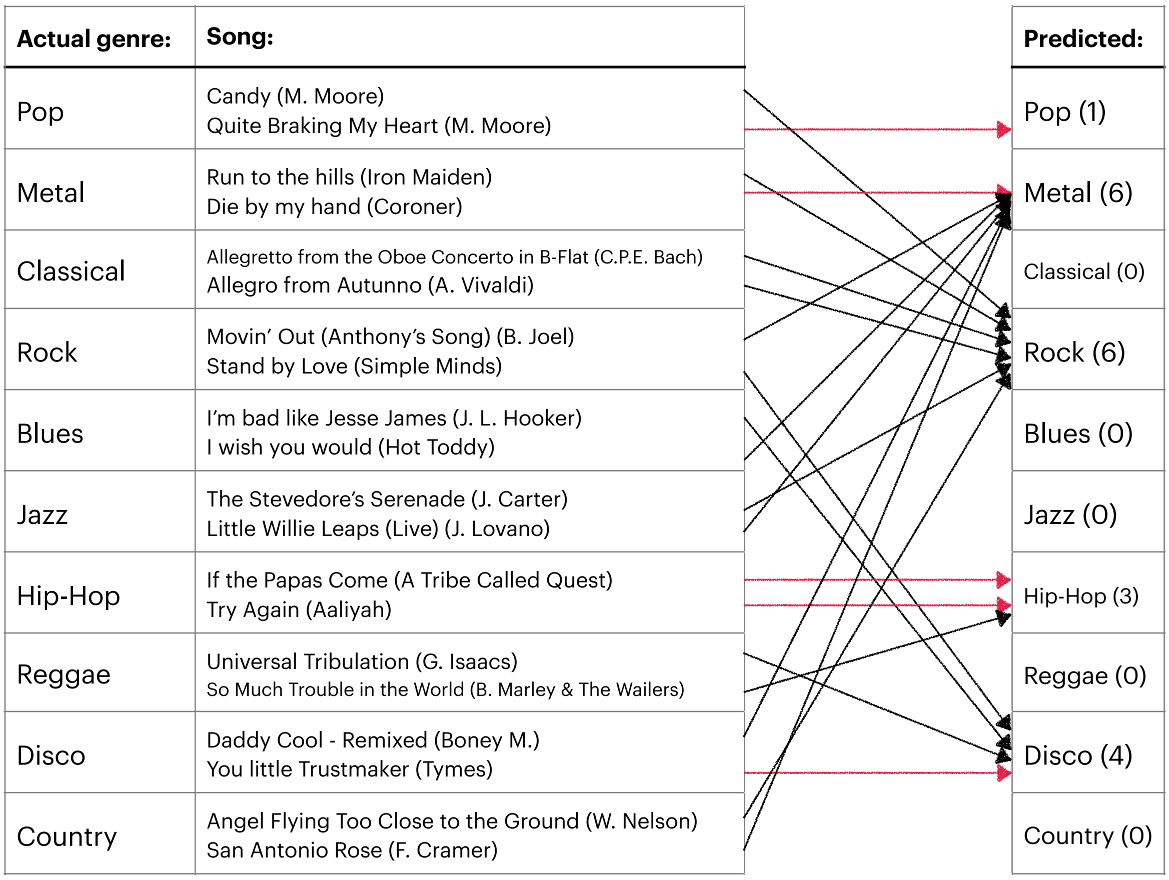


True label \ Predicted label	pop	metal	classical	rock	blues	jazz	hiphop	reggae	disco	country
pop	1	0	0	1	0	0	0	0	0	0
metal	0	1	0	1	0	0	0	0	0	0
classical	0	0	0	2	0	0	0	0	0	0
rock	0	1	0	0	0	0	0	0	1	0
blues	0	1	0	0	0	0	0	0	1	0
jazz	0	1	0	1	0	0	0	0	0	0
hiphop	0	0	0	0	0	0	2	0	0	0
reggae	0	0	0	0	0	0	1	0	1	0
disco	0	1	0	0	0	0	0	0	1	0
country	0	1	0	1	0	0	0	0	0	0

Let us, for instance, focus on the first "1" of the first row. This cell tells us that one pop song has been classified as a pop song. The other "1" in the first row tells us that a pop song has been classified as a rock one.

We note that, if the algorithm did perfectly its work, the confusion matrix would be diagonal.

Let's try to justify the obtained result focusing on some of the songs that have been classified.



Actual genre:	Song:
Pop	Candy (M. Moore) Quite Braking My Heart (M. Moore)
Metal	Run to the hills (Iron Maiden) Die by my hand (Coroner)
Classical	Allegretto from the Oboe Concerto in B-Flat (C.P.E. Bach) Allegro from Autunno (A. Vivaldi)
Rock	Movin' Out (Anthony's Song) (B. Joel) Stand by Love (Simple Minds)
Blues	I'm bad like Jesse James (J. L. Hooker) I wish you would (Hot Toddy)
Jazz	The Stevedore's Serenade (J. Carter) Little Willie Leaps (Live) (J. Lovano)
Hip-Hop	If the Papas Come (A Tribe Called Quest) Try Again (Aaliyah)
Reggae	Universal Tribulation (G. Isaacs) So Much Trouble in the World (B. Marley & The Wailers)
Disco	Daddy Cool - Remixed (Boney M.) You little Trustmaker (Tymes)
Country	Angel Flying Too Close to the Ground (W. Nelson) San Antonio Rose (F. Cramer)

Predicted:
Pop (1)
Metal (6)
Classical (0)
Rock (6)
Blues (0)
Jazz (0)
Hip-Hop (3)
Reggae (0)
Disco (4)
Country (0)

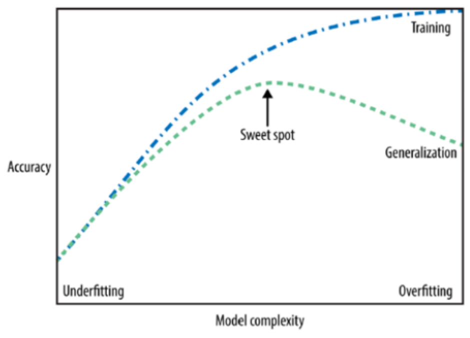
Keeping in mind that all the system is based on rhythm features, it is not surprising that a particular rhythm (with many electronic sounds and spoken instead of sung lyrics) as the hip-hop one is, has been perfectly detected by the model. We note how more than 50% of the songs have been mapped into metal or rock genre. We note that all the songs (apart from "I wish you would") that have been classified as metal, shares a particularly full spectra: probably, this density of frequencies has created some issues during the novelty function analysis. A possible solution to avoid this kind of problem, could be increasing the value of H to enhance the peaks and make them more distinguishable from the noise.

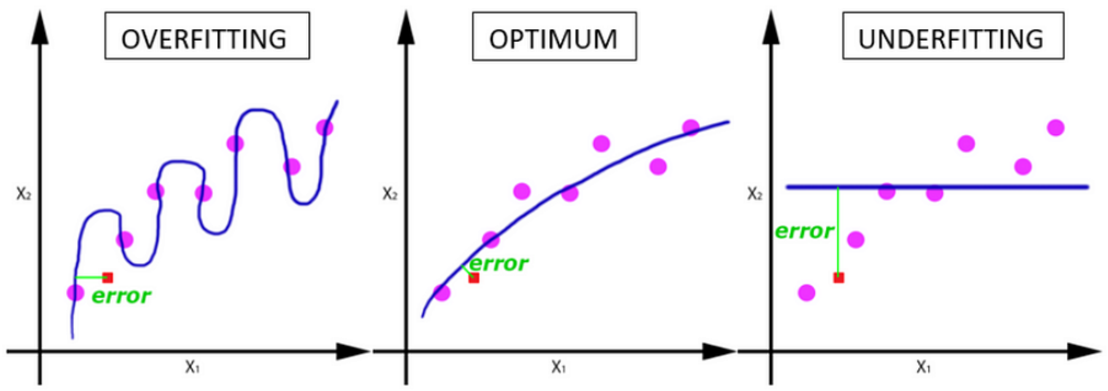
Another interesting thing we can note is that all the songs that have been mapped into rock have more or less the same tempo:

- Candy: 135.99917763 bpm
- Run to the hills: 130.8346519 bpm
- Allegretto by Bach: 130.8346519 bpm
- Allegro by Vivaldi: 130.8346519 bpm
- The Stevedore's Serenade: 130.8346519 bpm
- Angel Flying Too Close To the Ground: 130.8346519 bpm

We can suppose that this similarity of tempo play an important role in classifying all these songs under the same label.

As we can see from the result obtained, this is not the best classification we can get. We are probably facing an overfit situation given the fact that the accuracy of the training set is very high (100%) and the one of the testing set is low (25%). In order to solve this problem, we have to reduce the model complexity by decreasing C. In this way the model will not fit so closely to the particularities of the training set and it will be more able to generalise to new data. However, we have to find a good trade off in order to don't get underfitting. Coming back to the plane used in the previous question for our naives explanations, we can express the three situation in this way:



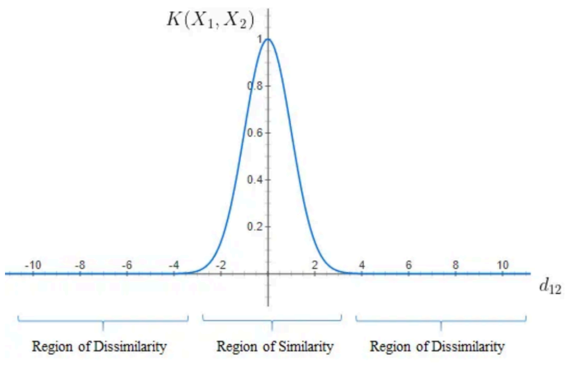


We try now to change some parameters to get better results. To do this, we simply recall the functions defined in the previous point and we pass them different parameters.

First of all we consider what we have said during our discussion about the conjugate trade-off that we have to find for N and H. So, we decide to increase the value of H letting N as it was. Thus, we choose $N = 2048$ and $H = 800$. Furthermore, we change also the type of Kernel choosing a radial basis function instead of a linear one. This kind of Kernel is very similar to a Gaussian distribution and it compute the similarity between two elements (a.k.a. how close are their representation in the multidimensional space) with the following formula:

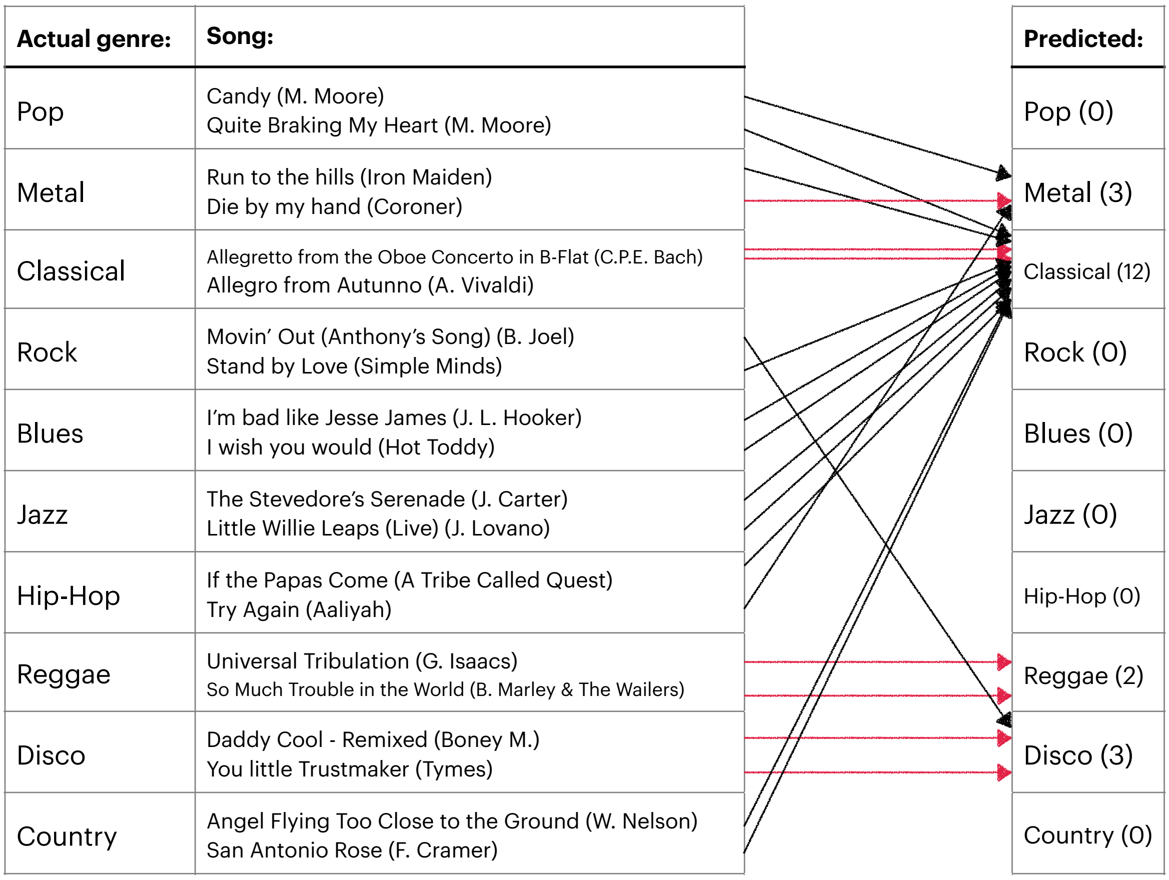
$$K\left(f\_vector[i], f\_vector[j]\right) = exp\left(-\frac{\|f\_vector[i] - f\_vector[j]\|^2}{2\sigma^2}\right)$$

As we can gather, farer are the two elements, lower is the result. When the two elements are superimposed, we obtain the maximum value that is 1. Now, we have to choose our hyperparameters C and γ where the second one is directly related to σ by an inverse proportional relation. As for a Gaussian the element at the denominator of the exponential changes the amplitude of the bell, also in this definition we can change the value of σ in order to obtain a larger or a thinner bell-shape. All the values that are under the bell (so, all the elements of the feature vector for which $K\left(f\_vector[i], f\_vector[j]\right) \neq 0$, are in the region of similarity. Since we don't want neither a particular larger

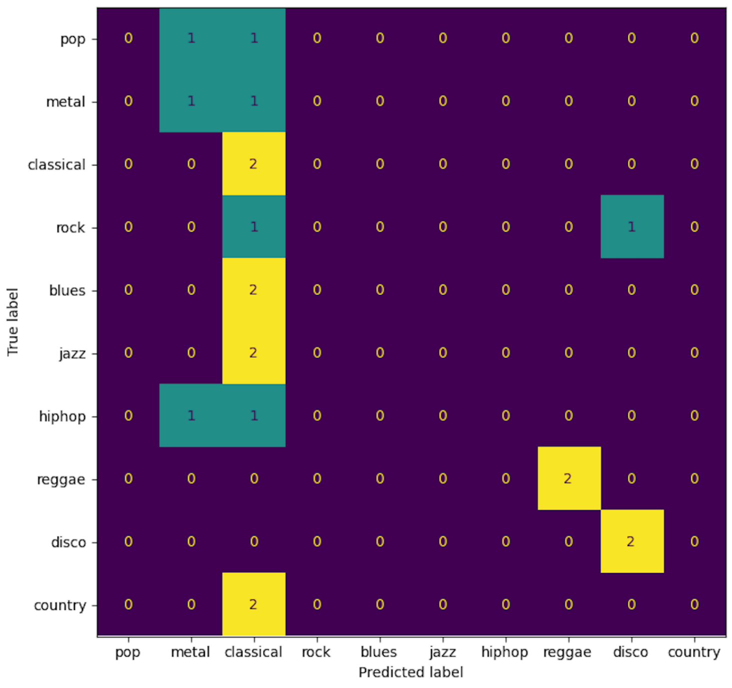


bell nor a particular narrow one, we set $\gamma = 1$, obtaining a bell-shape like the one in the figure on the left. Reminding which are the boundaries of the approximative support of a Gaussian, we can affirm that all the tracks for which $-\frac{3}{C} \propto -3\sigma \leq K\left(f\_vector[i], f\_vector[j]\right) \leq 3\sigma \propto \frac{3}{C}$, are similar.

By letting also C equal to 1, we obtain an accuracy of 35% for the following result.



Actual genre:	**Song:**	**Predicted:**
Pop	Candy (M. Moore) Quite Braking My Heart (M. Moore)	Pop (0)
Metal	Run to the hills (Iron Maiden) Die by my hand (Coroner)	Metal (3)
Classical	Allegretto from the Oboe Concerto in B-Flat (C.P.E. Bach) Allegro from Autunno (A. Vivaldi)	Classical (12)
Rock	Movin' Out (Anthony's Song) (B. Joel) Stand by Love (Simple Minds)	Rock (0)
Blues	I'm bad like Jesse James (J. L. Hooker) I wish you would (Hot Toddy)	Blues (0)
Jazz	The Stevedore's Serenade (J. Carter) Little Willie Leaps (Live) (J. Lovano)	Jazz (0)
Hip-Hop	If the Papas Come (A Tribe Called Quest) Try Again (Aaliyah)	Hip-Hop (0)
Reggae	Universal Tribulation (G. Isaacs) So Much Trouble in the World (B. Marley & The Wailers)	Reggae (2)
Disco	Daddy Cool - Remixed (Boney M.) You little Trustmaker (Tymes)	Disco (3)
Country	Angel Flying Too Close to the Ground (W. Nelson) San Antonio Rose (F. Cramer)	Country (0)



	pop	metal	classical	rock	blues	jazz	hiphop	reggae	disco	country
pop	0	1	1	0	0	0	0	0	0	0
metal	0	1	1	0	0	0	0	0	0	0
classical	0	0	2	0	0	0	0	0	0	0
rock	0	0	1	0	0	0	0	0	1	0
blues	0	0	2	0	0	0	0	0	0	0
jazz	0	0	2	0	0	0	0	0	0	0
hiphop	0	1	1	0	0	0	0	0	0	0
reggae	0	0	0	0	0	0	0	2	0	0
disco	0	0	0	0	0	0	0	0	2	0
country	0	0	2	0	0	0	0	0	0	0

As we can note, the reggae, disco and classical songs have been perfectly classified. Furthermore, there has not been any kind of wrong classification for the reggae genre: just the two reggae songs have been classified under the reggae label.

We also note that most of the songs that have been mismatched have been classified as classical ones.

A possible explanation to justify this sort of bias could be found looking at the list of the classical pieces used for training:

- *Nocturne* from Ainsi la Nuit (H. Dutilleux)
- *Ich steh mit einem Fuß in Grabe* (J.S. Bach)
- *Violin Concerto No. 1* (K. Szymanowski)
- *Shepherd's Hey* (P. Grainger)
- *Allegro ma non troppo* form the Oboe Concerto in E-Flat (J.S. Bach)
- *Finale Presto* from the Symphony No. 38 "Prague" (W.A. Mozart)
- *Allegro con grazia* from the Symphony No. 6 "Pathétique" (П.И. Чайко́вский)
- *Allegro di molto* from the Symphony No. 35 (J. Haydn)
- *Sonata XIII a 8 voci* (G. Gabrielli)
- *Finale, molto allegro* from the Symphony No. 41 "Jupyter" (W.A. Mozart)

Just barely knowing when the composers of these pieces lived, we can grasp how varied the set is. A piece like the Nocturne by Dutilleux (died in 2013) has a rhythmical pattern that has nothing in common with the one of the Sonata by Gabrielli (born in 1557), given the fact that the human perception of rhythm has been heavily evolving during 500 years. To fully understand how much the pieces are different we have just to listen to them: we will note that the first piece has an unusual rhythmic structure, so it is almost impossible to detect the tempo. On the other hand the Gabrielli's rhythm is perfectly regular and without any strong changes in the tempo (at least in the cropped part used for the set). Using the previously described kernel definition, we can suppose that is highly probable that all the feature vectors of the test-set have something that makes the classical label suitable from them. Not because their feature vectors are closer to the one of classical piece, but because, for how we trained the dataset, the classical label can be posed upon every piece that has not an explicitly distinguishable genre. Furthermore, given the fact that the chosen classical pieces have not a drum line, we can also suppose that the detection of their onset made by the novelty function is not so accurate.

We can conclude our report with a general note on this algorithm. We observe that, given the fact that the algorithm is overfitted, it is not so able to generalise. In order to improve its accuracy we have to avoid this phenomena. Mainly, there are two possibilities to do this: decreasing the value of the hyper-parameter C or training the model with more datas in case we cannot change its structure. From literature is well-known that the more training data we use, the more accurate the final model is. With our code we have proved the effect of C in the last two cells. They provide an example of how this hyper-parameter can change the result. In fact, by using same kernel (linear), same N and same H but decreasing the C value the accuracy related to the *training-set* decreases, while the one related to the *test-set* increases. So, this is a little demonstration on how decreasing C, avoiding overfitting, we have a model that is more able to generalise.

References:

We reported just the references used for the result's analysis and not the one used to write the code.

1. Deeplake, *deeplake,* https://docs.deeplake.ai/en/latest/deeplake.html#deeplake.load
2. Rochester Institute of Technology, *Normalisation and Generalization in Deep Learning*, https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=12521&context=theses
3. Bob L. Sturm, The GTZAN dataset: Its contents, its faults, their effects on evaluation, and its future use, 2013.
4. Fabien Gouyon, François Pachet, Olivier Delerue, *On the use of zero-crossing rate for an application of classification of percussive sounds,* 2000.
5. Drishti Sharma, *Analysis of Zero Crossing Rates of Different Music Genre Tracks,* https://www.analyticsvidhya.com/blog/2022/01/analysis-of-zero-crossing-rates-of-different-music-genre-tracks/
6. *SVMs Documentation,* https://scikit-learn.org/stable/modules/svm.html
7. Stack Exchange, *What is the influence of C in SVMs with linear Kernel?,* https://stats.stackexchange.com/questions/31066/what-is-the-influence-of-c-in-svms-with-linear-kernel
8. Toward Sai, *Underfitting & Overfitting—The Thwarts of Machine Learning Models' Accuracy,* https://towardsai.net/p/machine-learning/underfitting-overfitting-the-thwarts-of-machine-learning-modelsaccuracy
9. Steven Dye, *An intro to Kernels,* https://towardsdatascience.com/an-intro-to-kernels-9ff6c6a6a8dc
10. Sushanth Sreenivasa, *Radial Basis Function (RBF) Kernel: The Go-To Kernel,* https://towardsdatascience.com/radial-basis-function-rbf-kernel-the-go-to-kernel-acf0d22c798a
11. Scikit-learn, *Accuracy classification score*, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html