Computer Music Languages and Systems

Homework n° 1, audio effect classification

10574752

Davide Lionetti

10612929

Jonathan Rocco

Master of Science in Music and Acoustic Engineering

POLITECNICO
MILANO 1863

Abstract

Our group worked on **Assignment 3**: implement a classifier system able to predict the audio effect used in recordings of electric guitar and bass. We divided our work in steps as presented in the sections of this document, from the choice of the feature through the selection of a classification method, for each phase we present the decisions we made. In general, we used as a reference the second laboratory on Classification, mainly because we implemented the Support Vector Machine with majority voting for multiclass classification.

Chapter 1

Reasoning and implementation

1.1 Feature choice

The first attempt in trying to classify those three classes was by using one low-level descriptor: the **Spectral Centroid** (**SC**).
This synthetic descriptor represents where the center of mass of the spectrum is located. It is calculated through the following formula:

$$SC = \frac{\sum_{n=0}^{N-1} f(n)s(n)}{\sum_{n=0}^{N-1} s(n)}$$

where $f(n)$ is the frequency in Hz corresponding to bin n, $s(n)$ represents the spectral value at bin n and N is the total number of frequency bins.
We opted, at first glance, to use this feature because since the spectrum in a distorted audio sample is distributed along an higher range of frequencies, the spectral centroid is therefore higher than a noFX sample. But this descriptor, despite working well in recognizing the difference between distortion vs tremolo and distorsion vs noFX, it doesn't work that well when we have to distinguish between tremolo vs noFX.
However, in trying this feature, we used a built-in function, already implemented in librosa, called *librosa.feature.spectral_centroid*. This function takes in input: the audio samples array obtained with the function *librosa.load*, the audio sampling rate, the FFT window size, the hop length for STFT, the window length and the type of window used for analyzing the audio. The output of the function is an array containing the centroid frequencies.
Then, we looked for another low-level descriptor, and the final choice fell to the **Root Mean Square** (**RMS**), for both tremolo detection and distorsion detection. The RMS is defined as the square root of the mean energy of the

signal, through the following formula:

$$RMS = \sqrt{\frac{1}{N}\sum_n |x(n)|^2}$$

where $|x(n)|$ is the amplitude of the signal evaluated at the time frame n and N represents the number of frames in which the signal is considered.

In other terms, the RMS represents the average "power" of the signal.

We decided to use this spectral feature because the difference between the three classes can be well explained in terms of the mean energy of the waveform in the frequency spectrum.

Indeed, by simply using a spectrum analyzer, we can see that, for example, for the tremolo audio samples, the waveform is rapidly oscillating in amplitude. In the distortion audio samples, instead, the waveform is distributed along more frequencies in the spectrum. On that note, we'll expect a lower energy in the tremolo case, an higher energy in the distortion one, and an intermediate energy in the case of samples with no effects.

For implementing this feature we used a built-in function. The Root Mean Square is one of the functions for spectral feature extraction already implemented in Librosa. The function is *librosa.feature.rms.*

This function takes in input: the audio samples array obtained with the function *librosa.load*, the length of analysis frame (in samples) for energy calculation and the hop length for STFT. The output of the function is an array containing all the RMS values, one for each frame.

1.2 Dataset selection

For the definition of the datasets, we decided to use only *monophonic bass* and *monophonic guitar* from **IDMT-SMT-AUDIO-EFFECTS** in order to have more homogeneous sample's features. While for the train and test sets we divided the whole classes in about half, as shown in the following table:

	Train (# elem)	Test (# elem)
Distortion	1394	1238
Tremolo	1194	1228
NoFX	559	555

Table 1.1: Train and test datasets sizes

1.3 Multi-class Weighted Support Vector Machine

SVMs are generally binary classifier, so, in order to manage 3 classes, we used one SVC for each couple of class: Distortion/Tremolo, Distortion/NoFX and Tremolo/NoFX. Each one misclassify the missing class, hence, applying majority voting, we are able to train and test the SVM with all the instances of the samples.

In our training set we face the problem of having an unbalance amount of tracks for every class type. It contains half of no-effect tracks compared to the effected ones, hence we have found a way to solve this problem. The SVM model works well also with unbalanced training-set taking care of setting efficiently the right value of an internal private variable called C.
C determines the number of tolerated severity apply to the margin violation (and to the hyperplane), it controls the trade-off between maximizing the separation margin between classes minimizing the number of misclassified instances. Specifically, each example in the training dataset has its own penalty term C used in the calculation of the margin when fitting the SVM model. The value of an example's C-value can be calculated as a weighting of the global C-value, where the weight is defined proportional to the class distribution.

$$C_i = weight_i * C$$

By default it is suppose a balanced training-set, each class has the same weighting, which means that the softness of the margin is symmetrical.
A larger weighting factor C can be used for the minority class, allowing the margin to be softer (smaller penalty for misclassified examples), whereas a smaller C can be used for the majority class, forcing the margin to be harder and preventing misclassified examples. This has the effect of encouraging the margin to contain the majority class with less flexibility, but allow the minority class to be flexible with misclassification of majority class examples onto the minority class side if needed.

1.4 Cross-validation for SVM parameters

In order to select appropriate parameters for the SVM, we tested our classifier through Cross-validation using the function GridSearchCV from scikit-learns, in doing so we were able to fine-tune the machine learning algorithm and to avoid overfitting the model. We decided also to run the grid-search only on the classifier between Tremolo and NoFX, because those were the classes that would presents more problems during the classification. The code snippet and the result is shown below, the score measures are precision and recall, while the C parameters were chosen arbitrarily within an order of magnitude:

```
tuned_parameters = [
  {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
  {'C': [1, 10, 100, 1000], 'kernel': ['rbf']},
 ]

scores = ['precision', 'recall']

for score in tqdm(scores):
    print("# Tuning hyper-parameters for %s" % score)
    print()

    clf = GridSearchCV(
        SVC(), tuned_parameters, scoring='%s_micro' % score
    )
    clf.fit(X_train12, y_train_12)

    print("Best parameters set found on development set:")
    print()
    print(clf.best_params_)
    [...]
```

Output:

```
# Tuning hyper-parameters for precision

Best parameters set found on development set:

{'C': 1000, 'kernel': 'rbf'}

Grid scores on development set:

0.759 (+/-0.121) for {'C': 1, 'kernel': 'linear'}
0.735 (+/-0.073) for {'C': 10, 'kernel': 'linear'}
0.710 (+/-0.103) for {'C': 100, 'kernel': 'linear'}
0.689 (+/-0.085) for {'C': 1000, 'kernel': 'linear'}
0.916 (+/-0.044) for {'C': 1, 'kernel': 'rbf'}
```

```
0.984 (+/-0.035) for {'C': 10, 'kernel': 'rbf'}
0.993 (+/-0.019) for {'C': 100, 'kernel': 'rbf'}
0.993 (+/-0.016) for {'C': 1000, 'kernel': 'rbf'}
[...]

# Tuning hyper-parameters for recall

Best parameters set found on development set:

{'C': 1000, 'kernel': 'rbf'}

Grid scores on development set:

0.759 (+/-0.121) for {'C': 1, 'kernel': 'linear'}
0.735 (+/-0.073) for {'C': 10, 'kernel': 'linear'}
0.710 (+/-0.103) for {'C': 100, 'kernel': 'linear'}
0.689 (+/-0.085) for {'C': 1000, 'kernel': 'linear'}
0.916 (+/-0.044) for {'C': 1, 'kernel': 'rbf'}
0.984 (+/-0.035) for {'C': 10, 'kernel': 'rbf'}
0.993 (+/-0.019) for {'C': 100, 'kernel': 'rbf'}
0.993 (+/-0.016) for {'C': 1000, 'kernel': 'rbf'}
[...]
```

1.5 Result

In order to control the effectiveness of the feature, we first compared the plots of the rms of the three different classes. In these figures we saw that the *distortion* effect has a long sustain phase, meanwhile *NoFX* and *tremolo* rapidly decay. These last two classes rms has the same envelope but *tremolo* has an oscillating behaviour due to its characteristic amplitude oscillation.

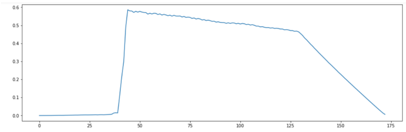

Figure 1.1: Distortion

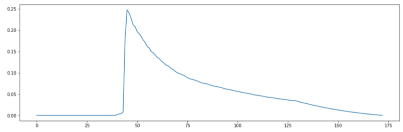

Figure 1.2: No Effects

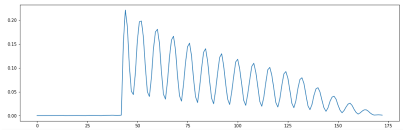

Figure 1.3: Tremolo

We also compared the train and test set of the audio samples belonging to the same class and we saw that the rms are very similar to each other. Then, at the end of the feature computation and voting we calculated and visualised the confusion matrix:

rms	distortion	tremolo	NoFx
distortion	1219	19	0
tremolo	4	1224	0
NoFX	7	1	547

As the table above shows, the rms it is enough for a good separation and then identification of the three different effects, meanwhile using the other features we took in consideration we weren't able to discern all the three classes properly. In particular with the spectral centroid classification the *distortion* were well separated from the other but *NoFx* were confused with *tremolo* more than the half of tests:

sp. centroid	distortion	tremolo	NoFx
distortion	1238	0	0
tremolo	6	1167	55
NoFX	0	352	203