Classification of music genres
Matlab Audio Project

Matteo De Filippis

1 Introduction

The aim for this project was to collect eighteen songs belonging to three different genres, in order to extract the audio features and apply a k-Nearest Neighbor algorithm to classify them. I chose one-minute-long extracts of rock, jazz and classical music since full-length songs took to long to compute (up to an hour and 15 minutes). The audio features I extracted were both time-domain audio features (*Energy* and *Zero-Crossing rate*), and frequency-domain ones (*Spectral Centroid, Spectral Spread, Spectral Rolloff* and *Mel-Frequency Cepstral Coefficients*).
Then I divided the songs into two sets, a train set and a test set, in order to apply the k-Nearest Neighbor algorithm to classify the test set, using different k's to discover the optimal one. I applied the kNN algorithm using time and frequency features, eventually using them together.

2 Time-Domain Features Extraction

I saved the different songs into three matrices, one for each genre, to have the files already saved into Matlab and avoid using the audioread function each time a song had to be used. Thus, I created six-column matrices, where the number of the rows was the same as the shortest song. This somewhat helped the computational speed of the program, as every song was cut to the same length of the shortest one. Then, I applied the nonzeros and the normalize functions to eliminate useless data and to allow better performances when applying the kNN algorithm, both on speed and accuracy.

```
    [y,fs] = audioread(paths(i));
    y = sum(y,2);
    y = nonzeros(y);
    y = normalize(y);
```

Listing 1: Extract from saveJazzSongs.m

After this first input part, I windowized the files, to extract the time-domain features from each frame and I plotted the results into *.png* files, saved into the folder *"plots/timefeatures"*.
As said before, the time-domain features extracted were energy, obtained as

$$E(i) = \frac{1}{W_L} \sum_{n=1}^{W_L} |x_i(n)|^2$$

and zero-crossing rate, obtained as

$$E(i) = \frac{1}{2W_L} \sum_{n=1}^{W_L} |sgn[x_i(n)] - sgn[x_i(n-1)]|$$

The code can be found in *timeDomainFeatures.m*, *feature_zcr.m* and *feature_energy.m*.
The features extracted for each file were then concatenated into two vectors, one for train sets (e.g. *trainJazzE*, containing the energy values for the three train songs) and one for test sets (e.g. *testJazzZ*, containing the ZCR values for the three train songs), in order to be used in kNN algorithm.

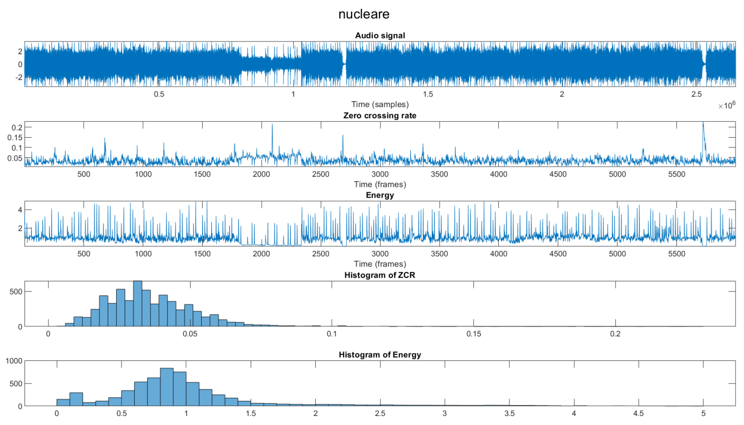

Figure 1: Example of a plot with time-domain features

3 Frequency-Domain Features Extraction

The extraction of frequency-domain features was similarly carried out.
I windowized the files and also applied a Hamming window, in order to have better performance and to smooth data. Then I extracted the features from each window and plotted the results into *.png* files, saved into the folder *"plots/freqfeatures"*.

The frequency-domain features extracted were spectral centroid, obtained as

$$C_i = \frac{\sum_{k=1}^{Wf_L} kX_i(k)}{\sum_{k=1}^{Wf_L} X_i(k)}$$

spectral spread, obtained as

$$S_i = \sqrt{\frac{\sum_{k=1}^{Wf_L} (k - C_i)^2 X_i(k)}{\sum_{k=1}^{Wf_L} X_i(k)}}$$

spectral rolloff, which is the mth Discrete Fourier Transform coefficient that satisfies the equation

$$\sum_{k=1}^{m} X_i(k) = C \sum_{k=1}^{Wf_L} X_i(k)$$

and Mel-Frequency Cepstral Coefficients.
The code can be found in *frequencyDomainFeatures.m* and the respective functions.
The features extracted for each file were then concatenated into two vectors (or matrices for the MFCCs), one for train sets (e.g. *trainJazzCeps*, containing the first 13 MFCCs for the three train songs) and one for test sets (e.g. *testJazzR*, containing the Spectral Rolloff values for the three test songs), in order to be used in kNN algorithm.

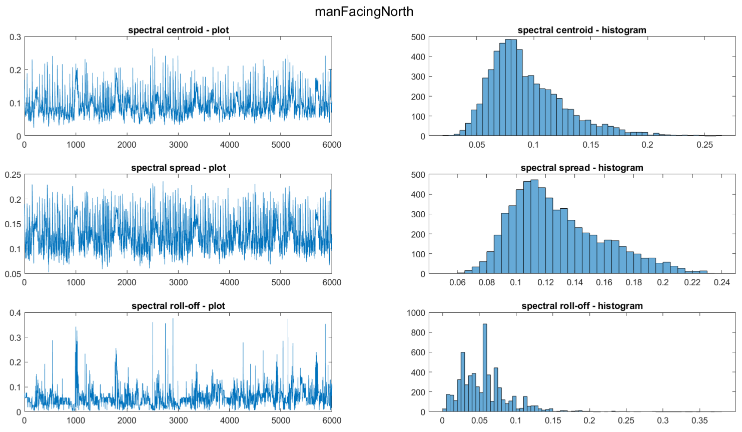

Figure 2: Example of a plot with frequency-domain features

4 kNN algorithm

The kNN algorithm, which was implemented using different features (every function can be found in *kNN* folder), was firstly used with each feature separately. This way I found the best parameters (which were the MFCCs) and the best k values. I plotted the results into *kNN_separate_features.png*. Then, I concatenated all the time features in a matrix, all the frequency features in another, and then all the features together. As expected, the best result was achieved using all the features together. A plot with the results can be found in *kNN_freq_time_all.png*. All the functions returned the accuracy rate for each k parameter and the best value for every considerated feature. To not slow down the program, the k values used were only [1 5 10 15 20].

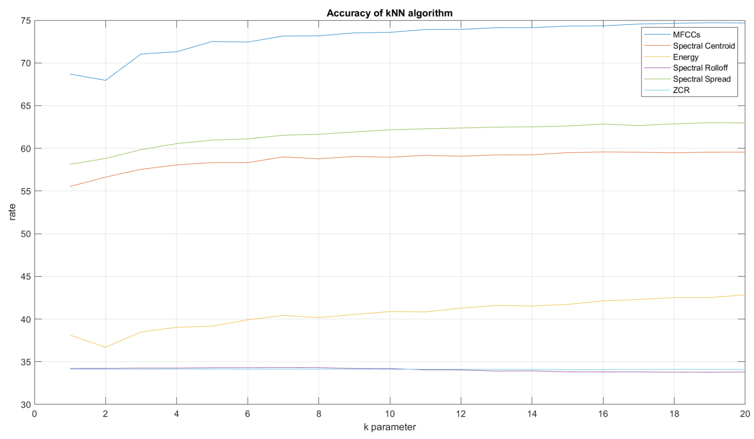

Figure 3: kNN algorithm results applied to each different feature

5 Conclusion

The elapsed time, using one-minute-long extracts, was around 12/13 minutes, so this helped a lot with the computational time. However, even if they were quite good (in some cases the accuracy rate exceeded 70%) the results were not the ones expected. In fact the accuracy rate steadily grows as k values grow (I even checked with k values higher than the ones that were used), while I would expect the best rate using an average k value (using too many neighbors could lead to considering wrong values from other classes, and using too little could raise considerable noise problems). So, some improvements could be applied (such as using average frames features instead of having frame-level inputs, or including the first derivative to have better accuracy). For lack of time, this couldn't be done. However the results and the individual values and graphs of every feature were mainly in accordance with the expectations.

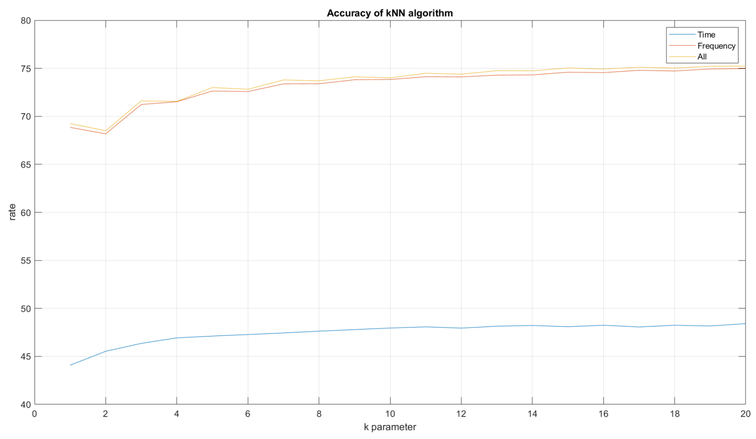

Figure 4: kNN algorithm results applied to concatenated features