Tools for Feature Extraction: Exploring essentia

MUS-15

Andrea Hanke

July 5, 2017

Introduction

In the research on Music Information Retrieval, it is attempted to automatically classify a piece of music based on the raw audio-file, without the manual work of a human listener. To achieve this, computational feature extraction is needed. For this essentia is an ideal tool.

Essentia is a C++ library with Python bindings for audio analysis. The software is fairly young (first version released in 2008, newest version 2.1 beta3 released on September 2016) and allows to add new algorithms easily. For this reason, and because essentia is open-source, continuing additions are made, making it an extensive library with many up-to-date and experimental algorithms. Essentia is mainly supported by the Music Technology Group of the Universitat Pompeu Fabra.

This report tries to demonstrate the usage of essentia with both Python and C++.

General Structure of essentia

Essentia provides two operating modes: The standard mode and the streaming mode. While the standard mode provides maximum control in C++, it is also recommended for research with Python, due to its interactive environment. The streaming mode allows to easily port your code from Python to C++ and provides easy-to-write extractors in C++ and Python.

The essentia-library covers a wide range of algorithms: From spectral, tonal and pitch analysis over loudness, dynamics and rythm analyisis to filters, extraction and segmentation of an audio file, just to name a few.

These functionalities are provided as individual processing blocks, which are called Algorithms, or essentia-algorithms in this report. An essentia-algorithm has three different types of attributes: Parameters, Inputs and Outputs. Every Algorithm can have any number of those attributes, including none at all. The Parameters can have default values, and are set when an Algorithm is instantiated. An instantiated Algorithm receives the Input-variables as paramters and returns the Output-variables after finishing its calculation.

Using essentia with Python

After importing the essentia-library:

```
import essentia
import essentia.standard
import essentia.streaming
```

its Algorithms can be accessed. To use an essentia-algorithm from the library one first always needs to create an instance of it. Some Algorithms require Parameter-values, whilst other Algorithms have default values for their Parameters, which can be configured when creating the Algorithm. For example the Algorithm *EasyLoader*, which loads a music file, requires the Parameter *filename* to be instantiated with the path to the music file. Optionally, *startTime* and *endTime* can be set to choose the time frame of the extracted audio.

instantiate EasyLoader

```
loader =
    essentia.standard.EasyLoader(
    filename = "../music/audio.mp3",
    startTime = musicStart,
    endTime = musicEnd)
```

The result of this instantiation is a Python function, which can be used like any other function in Python. Depending on the algorithm, multiple Input parameters need to be passed to the function for the calculation. In the case of *EasyLoader* we have no Input and one Output:

use EasyLoader

```
audio = loader()
```

On the essentia website all Algorithms are documented with a detailed description of their Parameters, Inputs and Outputs as well as the calculation they are performing.

Adding a new Algorithm to the library

In the following, I will describe how to add a new essentia-algorithm to the library with the example of *DanceabilityDetailed*. I added this Algorithm because the original Algorithm *Danceability* was not returning the full information that could be retrieved from the calculation.

To add a new Algorithm, first the two aptly named files

- *danceabilityDetailed.h* and
- *danceabilityDetailed.cpp*

need to be created.

The new Algorithm is a subclass of *essentia::Algorithms*, inheriting functions to handle the Input, Output and Parameter attributes. In *danceabilityDetailed.h* first all protected Input and Output variables are declared. In the constructor declaration those variables are set as Inputs and Outputs with *declareInput(...)* and *declareOutput(...)*. With *declareParameters()* the Parameters are declared. The two functions *compute()* and *configure()* and the variables *name*, *category* and *description* are inherited from *essentia::Algorithms*. Finally, functions and variables specific for the Algorithm are declared.

danceabilityDetailed.h

```
#ifndef ESSENTIA_DANCEABILITYDETAILED_H
#define ESSENTIA_DANCEABILITYDETAILED_H
#include "algorithm.h"
#include "essentiamath.h"

namespace essentia {
namespace standard {

class DanceabilityDetailed :
    public Algorithm{

protected:
  Input<std::vector<Real> > _signal;
  Output<Real> _danceability;
  Output<std::vector<Real> >
                    _dfaExponents;
  Output<std::vector<Real> > _dfaTaus;
  int _preferredSize, _actualSize;

public:
  DanceabilityDetailed() {
    //as seen in the paper
    _preferredSize = 36;
    _actualSize = _preferredSize;
    declareInput(
        _signal,
        "signal", "#d");
    declareOutput(
        _dfaTaus,
        "dfaTaus", "#d");
  }

  void declareParameters() {
    declareParameter(
        "minTau", "#d",
        "(0,inf)", 310.);
    declareParameter(
        "maxTau", "#d",
        "(0,inf)", 8800.);
  }
```

```
  void compute();
  void configure();

  static const char* name;
  static const char* category;
  static const char* description;

  protected:
  std::vector<int> _tau;
  Real stddev(const std::vector
        <Real>& array, int start,
            int end) const;
};

} // namespace standard
} // namespace essentia
```

Note, that most of the above code needs to be repeated in a similar fashion for the streaming mode of essentia. In the *danceabilityDetailed.cpp*-file the inherited variables and all functions are filled with meaning. Note that *configure* is called when the Algorithm is created. Here the Parameters are used to set corresponding internal variables. *compute()* is called when the created Algorithm is executed, here the actual calculation is defined.

danceabilityDetailed.cpp

```
#include "danceabilityDetailed.h"

using namespace std;
namespace essentia {
namespace standard {

const char* DanceabilityDetailed
  ::name= "DanceabilityDetailed";
const char* DanceabilityDetailed
  ::category = "Rhythm";
const char* DanceabilityDetailed
  ::description
    = DOC("Long\n\nDescription");


void DanceabilityDetailed
            ::configure() {
  Real minTau =
    parameter("minTau").toReal();
  Real maxTau =
    parameter("maxTau").toReal();
  Real tauIncrement =
    parameter(
        "tauMultiplier"
            ).toReal();

  if (minTau > maxTau) {
    throw EssentiaException(
     "Danceability: minTau cannot \
_____be larger than maximumTauInMs"
     );
  }

  // tau is the number of blocks of 10ms
  // we calculate each time
  _tau.clear();
  for (Real tau = minTau;
            tau <= maxTau;
            tau *= tauIncrement) {

    _tau.push_back(int(tau / 10.0));

  }
}
void DanceabilityDetailed::compute() {
    //using _tau
    ...
}
} // namespace standard
} // namespace essentia
```

Again, most of the above code needs to be repeated similarly for the streaming mode of essentia.

For more information on the *danceability*-Algorithm, please refer to Akshay Paranjape's report.

Outlook: One possible application of essentia

Considering the wide range of Algorithms essentia provides to analyse music, a possible application could be a DJ-program: With essentia

certain features of the songs can be extracted, such as beats per minute, meter, and danceability. These features can be used to classify the songs, possibly with machine learning. The DJ-program can use these classifications to determine which song to play next or to create playlists with certain themes. While songs are already classified by e.g. genre or composer, with essentia it is possible to automatically classify pieces of music by their raw audio-input and its features.

To gain first-hand experience in the use of essentia, I wrote a Python-code (see Appendix) with this application in mind, demonstrating the use of a few Algorithms that should be useful for such feature extractions. The Python code extracts beat positions, beats per minute, meter (currently just experimental), rubato, and novelty.

Conclusion

Essentia offers a huge variety of Algorithms for feature extraction and a great documentation on its website `http://essentia.upf.edu` with detailed descriptions of each algorithm, the use of essentia in both Python and C++, and how to create Algorithms.

The analysis of a song is quite fast: Executed on a laptop, my Python code takes only a few seconds per song, naturally depending on the song's length.

Overall, using the essentia-library was a worthwhile experience, as it is both easy to use and easy to extend. It has many potential applications in Music Information Retrieval.

References

[1] Full essentia homepage, root-site: `http://essentia.upf.edu`. Last visited: July 3rd, 2017.

[2] S. Streich, and P. Herrera. *Detrended Fluctuation Analysis of Music Signals: Danceability Estimation and further Semantic Characterization*. Proceedings of the AES 118th Convention, Barcelona, Spain, 2005.

Appendix 1

Python code for feature extraction

```
# -*- coding: utf-8 -*-
"""
Created on Mon May 1 20:19:04 2017

@author: andrea
"""

# import the essentia module. It is aptly named 'essentia'
import numpy as np
import math
import essentia
import essentia.standard
import essentia.streaming
import matplotlib.pyplot as plt
import time


from essentia.standard import*

playMusic = 'yes' # yes or no
musicIndex= -6


myMusicFiles = ['HouseLoop2016.wav','SherlockWhoYouReallyAre.wav',
    'PeriodicTableSong.wav', 'LastSled.wav', 'ItsGonnaBeOKAY.wav',
    'SingleLadies.wav', 'Thriller.wav',
    'TchaikovskyFlowersWaltz.wav', 'TchaikovskyFlowersWaltz.wav',
    'SherlockWhoYouReallyAre.wav', 'SherlockWhoYouReallyAre.wav',
    'ChaChaMusic.mp3', 'rumba.mp3', 'Tango.mp3']
myMusicStarts = [0,0, 8, 9, 0, 85, 14, 67, 334, 0, 64, 30, 0, 0]
myMusicEnds = [180,200, 164, 344, 222, 190, 344, 206, 377, 64,
    88, 90, 60, 60]
musicfile = '../audio/' + myMusicFiles[musicIndex]
musicStart = myMusicStarts[musicIndex]
musicEnd = myMusicEnds[musicIndex]

samplingrate = 44100


# let's define a small utility function
def play(audiofile):
    import os, sys

    # NB: this only works with linux!! mplayer rocks!
```

```
    if sys.platform == 'linux2':
        if playMusic == 'yes':
            cmd = 'mplayer ' + audiofile + ' -ss '
            + str(myMusicStarts[musicIndex]) + ' -endpos '
            + str(myMusicEnds[musicIndex]-myMusicStarts[musicIndex])
            os.system(cmd)
        else:
            print 'Not playing audio because you asked me not to do so'
    else:
        print 'Not playing audio...'

print 'analyzing the audio ' + musicfile


#play(musicfile)
loader = essentia.standard.EasyLoader(
    filename = musicfile, startTime = musicStart,
    endTime = musicEnd)

# and then we actually perform the loading:
audio = loader()
audioSize = audio.size

analysisStart = 0*samplingrate
analysisEnd = audioSize
analysisDelta = 1024#2205
numFreqIter = int(math.floor((analysisEnd-analysisStart)/analysisDelta))


#instantiate all algorithms that don't need yet unknown parameters:
getWindow = Windowing(type = 'hann')
# FFT() would give the complex FFT,
# here we just want the magnitude spectrum
getSpectrum = Spectrum()
getFrequencyBands = FrequencyBands()
getNoveltyCurve = NoveltyCurve()
getBpmHistogram = BpmHistogram()
getBeatTrackerDegara = BeatTrackerDegara()
getBeatogram = Beatogram()
getRythmExtractor = RhythmExtractor2013()
getMeter = Meter()
getBpmRubato = BpmRubato()

##################################################
## get rythm, bpm:                              ##
##################################################

'''
```

```
bpm (real) - the tempo estimation [bpm]
ticks (vector_real) - the estimated tick locations [s]
confidence (real) - confidence with which the ticks are
    detected (ignore this value if using 'degara' method)
estimates (vector_real) - the list of bpm estimates
    characterizing the bpm distribution for the signal [bpm]
bpmIntervals (vector_real) - list of beats interval [s]
'''
(bpm, ticks, confidence, estimates, bpmIntervals) = getRythmExtractor(audio)
print 'Beats per Minute:'
print bpm

################################################
## get meter !!!Meter is only experimental ##
################################################
ticksDegara = getBeatTrackerDegara(audio)
getBeatsLoudness = BeatsLoudness(beats=ticksDegara)
(loudness, loudnessBandRatio) = getBeatsLoudness(audio)
beatogram = getBeatogram(loudness, loudnessBandRatio)
meter = getMeter(beatogram)



print 'meter: '
print meter


################################################
# get rubato for bpm:                          ##
################################################

'''
rubatoStart (vector_real) - list of timestamps where
        the start of a rubato region was detected [s]
rubatoStop (vector_real) - list of timestamps where
        the end of a rubato region was detected [s]
rubatoNumber (integer) - number of detected rubato regions
'''
(rubatoStart, rubatoStop, rubatoNumber) = getBpmRubato(ticksDegara)
print 'rubato: '
print rubatoStart
################################################
# get novelty                                  ##
################################################

i=0
frame = audio[analysisStart + i*analysisDelta : analysisStart
                + (i+1)*analysisDelta]
```

```
spectrum = getSpectrum(getWindow(frame))
allFrequencyBands = getFrequencyBands(spectrum)

for i in range(1,numFreqIter-1):
    frame = audio[analysisStart + i*analysisDelta : analysisStart
                + (i+1)*analysisDelta]
    spectrum = getSpectrum(getWindow(frame))
    allFrequencyBands = np.append(allFrequencyBands,
                getFrequencyBands(spectrum))

allFrequencyBands = allFrequencyBands.reshape(numFreqIter-1, -1)
novelty = getNoveltyCurve(allFrequencyBands)


########################################
## Do something usefull with the data ##
########################################

from matplotlib.pyplot import plot, draw, show
#ion() # enables interactive mode


y1 = [0, novelty.size]
x1 = [0, 0]

fig = plt.figure(figsize=(15,5))
ax  = fig.add_subplot(111)
ax.set_xlim(0, novelty.size+1)
ax.set_ylim(audio.min()*novelty.max()/audio.max(), novelty.max())
plt.show(block=False)
#plt.show()

ax.plot(np.arange(0,audio.size)*novelty.size/audio.size,
    audio*novelty.max()/audio.max(), label=myMusicFiles[musicIndex],
        color='dodgerblue')

for i in range(0,rubatoNumber):
    xc = (rubatoStart[i]*novelty.size)/(musicEnd-musicStart)
    ax.axvline(x=xc, color="green")
    xc = (rubatoStop[i]*novelty.size)/(musicEnd-musicStart)
    ax.axvline(x=xc, color="green")
ax.plot(novelty, label="Novelty", color="red", linewidth=2.)



ax.set_ylim(audio.min()*novelty.max()/audio.max(), novelty.max())

show()
```

```
# get the canvas object
canvas = ax.figure.canvas
background = canvas.copy_from_bbox(ax.bbox)
plt.subplots_adjust(left=0.1, right=0.9, top=0.75, bottom=0.1)
leg = ax.legend(bbox_to_anchor=(0, 1.3), loc=2, borderaxespad=0.)

# add the progress line.
line = ax.axvline(x=0, color='r', animated=True)


starttime=time.time()
mytimer=0
mytimer_ref=0


def update(canvas, line, ax):
    # revert the canvas to the state before any progress line was drawn
    ax.lines = ax.lines[:2+2*rubatoNumber]
    t = time.time() - starttime #- (musicEnd - musicStart)
    mytimer = ((t + mytimer_ref)*novelty.size)/(musicEnd-musicStart)
    # update the progress line with its new position
    ax.axvline(x=mytimer, color='r')
    canvas.blit(ax.bbox)

def startGraph():
    global starttime
    starttime=time.time()
    global mytimer_ref
    mytimer_ref=0#event.xdata
    print "starttime",starttime, mytimer_ref
    timer.start()

def onclick(event):
    play(musicfile)
    startGraph()

timer=fig.canvas.new_timer(interval=5)
args=[canvas,line,ax]
timer.add_callback(update,*args) # every 100ms it calls update function
# when I click the mouse over a point, line goes to that point and start moving
cid1=line.figure.canvas.mpl_connect('button_press_event',onclick)

#fig.savefig("analysis.png")
plt.show()
print 'plot_finished'
```