Exercise Track
Lepatriinu

Audio and Music Processing

June 23, 2013

Wolfgang Küllinger (0955711)
Fabian Jordan (0855941)

Contents

1 Architecture

1.1 General overview

Our project is called “Lepatriinu” which is the Estonian word for “lady beetle” (German: “Marienkäfer”) in order to express our sympathy for Austrian lady beetles, which are threatened by the superior species of Asian lady beetles.

Generally the structure of our project is object oriented. Therefore the functionality is split in several classes. The given framework simple calls methods which are defined in the package `at.jku.amp.lepatriinu`.

Additional to the functionality the project comes up with a graphical user interface which simplifies working on the audio data and represents the results in a proper fashion.

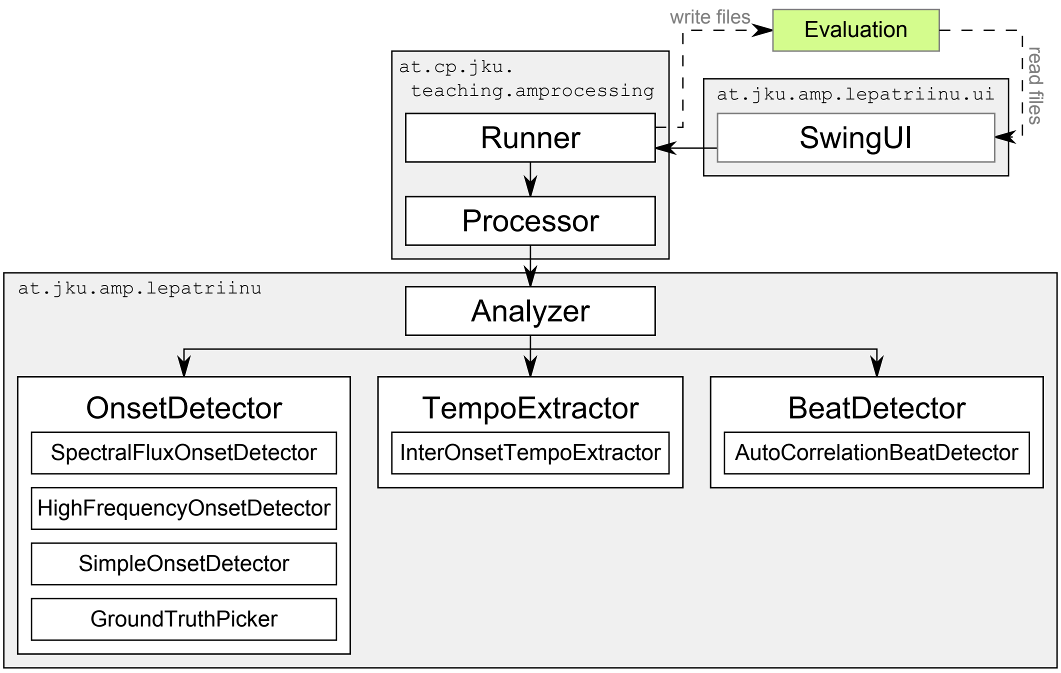

Figure 1.1: Class diagram

1.2 `Analyzer`

The *Analyzer* class defines all the constants that are needed for any algorithm in the project to succeed. In order to make central configurations possible, the constants are collected in this single class.

Furthermore it provides the central interface to the `at.cp.jku.teaching.amprocessing` project. It is initialized with a pre-processed (e.g. FFT) audio file of type `Audiofile`. The order of usage is important. In the first place, onset detection can be done. This information can directly be retrieved from the audio file. The found onsets are the base for tempo extraction. Therefore the onset list has to be provided as parameter of the tempo extraction function. Finally beat detection can be performed. In order to be able to use the best algorithms both the onset list as well as the calculated tempo should be provided.

1. Onset detection: `onsets = performOnsetDetection()`
2. Tempo extraction: `tempo = performTempoExtraction(onsets)`
3. Beat Detection: `beats = performBeatDetection(onsets, tempo)`

1.3 `OnsetDetector`

OnsetDetector is the abstract superclass of all onset detection algorithm's classes. It provides two different peak picking methods which can be used by any sub class.

1. The first peak picking method is taken from lecture slides (5.40) and called *Adaptive Thresholding*.
2. The second one is a slight modification of the adaptive thresholding algorithm. Instead of a threshold it simply isolates the highest magnitudes by zeroing everything but the peak as well as close neighbors that are lower. As a result a clear list of onsets remains. We called this method *mountain climbing*.

Constants

- `USE_MOUNTAIN_PEAKPICK` defines which peakpicking method to use. If constant is `true`, *mountain peakpicking* is chosen. Otherwise *adaptive thresholding* is performed.
- `THRESHOLD_RANGE` is used to specify the `int` size of the shifted slice used in *adaptive*

thresholding as well as the range of zeroed neighbors in *mountain climbing*. The best results were produced by a range of `5`.

- `PEAKPICK_USE_MEAN` defines whether to use mean (`true`) or median (`false`) in order to select the threshold that is later on applied to all the onset calculations of *adaptive thresholding*.
- `THRESHOLD` is used to define a fixed `int` threshold for choosing the magnitude peaks when using *mountain climbing*. The value that produced the best results in our experiments was `13`.

1.3.1 `SimpleOnsetDetector`

This very simplistic method is taken from the given framework. Generally it calculates the distances between every pair of spectral data bins and validates it against a fixed threshold of 10.

1.3.2 `SpectralFluxOnsetDetector`

This onset detection function implements the idea of spectral differences ("spectral flux"). Therefore the idea of slide 4.20 was implemented. Because this method evolved from a copy of `SimpleOnsetDetector` in the first place, two minimally different versions of this algorithm were implemented.

The mode can be chosen by the constant `FLUX_USE_TOTAL_ENERGY`.

- `true`: like the `SimpleOnsetDetector` the first version uses the total energy of each bin in order to calculate the differences, which are subsequently processed according to the formula.
- `false`: unlike the first method but according to the slides, the second method uses differences of magnitudes between bins. This has the unfortunate attitude to need an additional loop.

1.3.3 `HighFrequencyOnsetDetector`

As an alternative to spectral differences the *HighFrequencyOnsetDetector* implements the idea of so called "high frequency content" (slide 4.19). Again there are two modes that qualify for usage. These can be switched by the `boolean` constant `HIFQ_USE_WPHACK`.

- `false`: this mode implements the formula on the slides.

- `true`: unlike the original idea, this mode uses a formula which was found on Wikipedia[1]. Essentially the difference is the calculation of the mean, which is performed by the formula on the slides but left away in the formula from Wikipedia.

1.3.4 `GroundTruthPicker`

We designed the *GroundTruthPicker* just for experimenting with the best onsets we could possibly get: The original ground truth. It simply reads in the corresponding ground truth file from the input directory and parses it into a `List<Double>`, which it returns. To find the right file we had to expose the file name in the `Audiofile`. But as it turned out, the evaluation of the ground truth isn't even as true as we thought it would be - it was only considered with an average F-Measure of 92.5% over all 18 data sets.

1.4 `TempoExtractor`

TempoExtractor is the abstract superclass of which all tempo extractors should be derived.

1.4.1 `InterOnsetTempoExtractor`

As the name suggests, *InterOnsetTempoExtractor* performs the Inter Onset Intervals as described in the lecture slides 5.10 through 5.14. The method calculates the most common gap size between the onsets inside a set of onsets.

Because of the inaccuracy that evolves when adding vague `double` values and the necessity to provide some sort of categorization and clustering an indirect mapping from the `double` value of the gap size to the `int` value of the number of occurrencies was established. In order to reach this goal actually two mappings are done:

1. `double` $\rightarrow$ `int`: specified by the constant `TEMPO_KEY_TOLERANCE`, given distances are separated into different categories which are consecutively numbered using the `int` values. To be more precise, if `distance` $\pm$ `TEMPO_KEY_TOLERANCE` meets one key in the map, it is put into the same category. Otherwise it is put into a new one.
2. `int` $\rightarrow$ `int`: The *value* of the mapping in 1 is the *key* of the second mapping, which provides the number of occurencies.

[1] `https://en.wikipedia.org/wiki/High_Frequency_Content`

Of course this is to be done if the distance is within the range of usual tempi. Therefore distances that are smaller than `MIN_TEMPO` or greater than `MAX_TEMPO` are not taken into account.

The tempo then is returned in *beats per minute* (bpm).

1.5 `BeatDetector`

Also *BeatDetector* is an abstract class, thats purpose is to be the superclass of all beat detection algorithms.

1.5.1 `AutoCorrelationBeatDetector`

This subclass implements beat detection based on auto correlation. It accommodates an enumeration, named *Mode*, which allows to switch between three different ideas and implementations:

1. `AUTO_TEMPO_CORRELATION` simply implements the idea of the pulse train, described on slides 5.15 through 5.19. To tackle the inaccuracy the list of onsets will always carry along we use a threshold, called `AUTO_PHASE_TOLERANCE` which, after excessive testing we found best to be set around 0.21 and pushed it up to 48% (average over all 18 provided data sets).
2. `AUTO_ONSET_CORRELATION` represents the implementation of the IOI idea presented on lecture slides 5.10 through 5.14.
3. `AUTO_SPECTRAL_CORRELATION` was the attempt to improve the onset correlation by trying to reinterpret the formulas as to be the spectral data instead of the onsets. It didn't work out that well (less than 4% over 18 data sets).

1.6 Best practices

The best results were reached by using the following configuration:

- Onset detection:
  - onset detector: `SpectralFluxOnsetDetector` (see 1.3.2)
  - mode: `USE_TOTAL_ENERGY = true`
  - peak picking: `USE_MOUNTAIN_PEAKPICK = true` (see 1.3)

  - constants: `THRESHOLD = 13` and `THRESHOLD_RANGE = 5`

- Tempo extraction:
  - tempo extractor: `InterOnsetTempoExtractor` (see 1.4.1)
  - constants: `MIN_TEMPO = 0.3` (= 200 bpm), `MAX_TEMPO = 1` (= 60 bpm) and `TEMPO_KEY_TOLERANCE = 0.0195`.

- Beat detection:
  - beat detector: `AutoCorrelationBeatDetector` (see 1.5.1)
  - mode: `AUTO_FUNCTION = AUTO_TEMPO_CORRELATION`
  - constant: `AUTO_PHASE_TOLERANCE = 0.21`

2 User interface

2.1 Purpose and functionality

As the pure command line interface provided too few possibilities to see the overall performance of our algorithms and lacks a simple way to execute batches of runs, we decided to develop a small GUI in Swing to match these two requirements.

The `SwingUI` only needs the `FileUtils` to run properly. As it starts the `Runner` by simply calling its main method, there is no need to make big adaptations to the `Runner` itself. Only, in order to prevent the program from shutting down at any occuring error we had to replace the `System.exit(0)` calls by simple `RuntimeExceptions`. And, in order to make it run properly on Windows systems, we changed some code regarding paths, that had hardcoded Unix path seperators ("/").

Starting the program will allow you to select one or multiple files, which should be located in the "`./data/`" folder. On the bottom of the window three checkboxes let you choose, whether the output files should be generated. The "Run" buttons on the top are pretty self-explanatory.

2.2 Outputs

The outputs are displayed seperately for Onsets, Tempo and Beat, to keep a straight interface. Each of the three output tabs has the same layout:

- In the upper half, each output file will be displayed. For onsets and beat, only the respective "`*.eval`" files appear, whereas for the tempo the "`*.bpms`" as well as the "`*.bpms.eval`" file.
- In the bottom half, the selected file will be displayed as well as the summary of this set of evaluations. To make differences easy to spot at first look, the summaries are color coded pretty straight forward: The greener, the better.

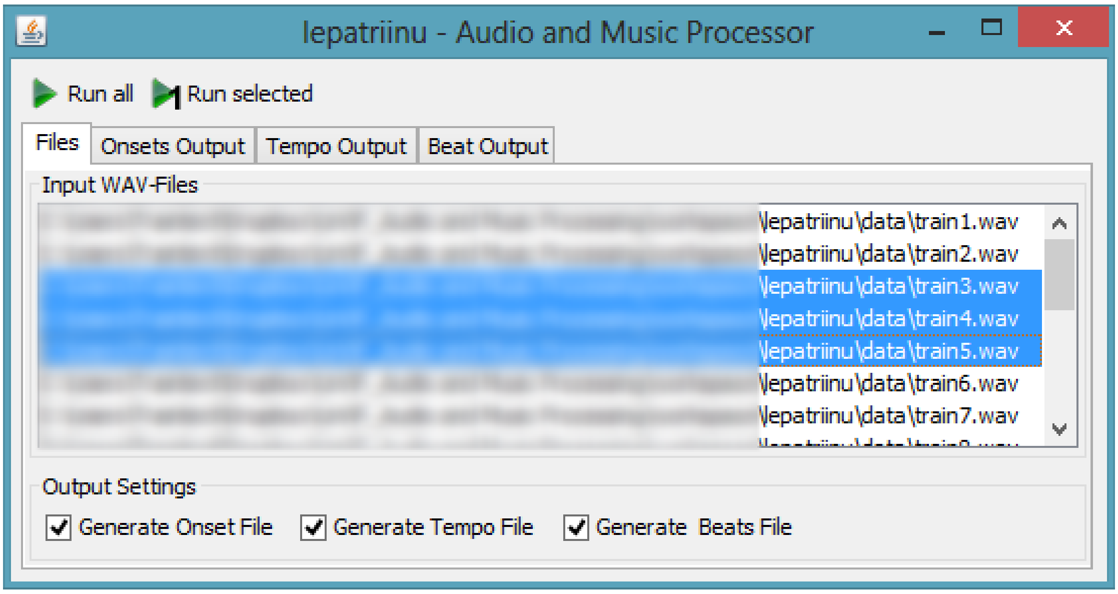

Figure 2.1: Graphical user interface

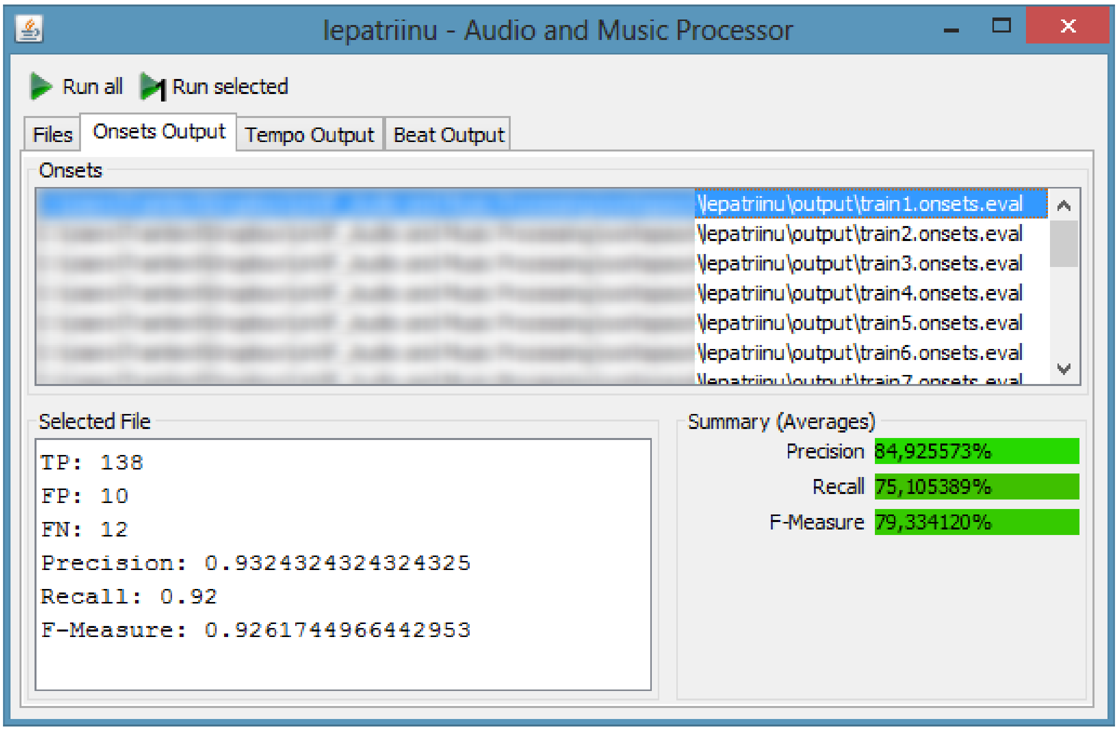

Figure 2.2: The output of the onsets