madmom Documentation

Release 0.16.1

madmom development team

Nov 14, 2018

Contents

CHAPTER 1

Introduction

Madmom is an audio signal processing library written in Python with a strong focus on music information retrieval (MIR) tasks. The project is on GitHub.

It's main features / design goals are:

- ease of use,
- rapid prototyping of signal processing workflows,
- most things are modeled as numpy arrays (enhanced by additional methods and attributes),
- simple conversion of a workflow to a running program by the use of processors,
- no dependencies on other software packages (not even for machine learning stuff),
- inclusion of reference implementations for several state-of-the-art algorithms.

Madmom is a work in progress, thus input is always welcome.

The available documentation is limited for now, but *you can help to improve it*.

CHAPTER 2

Installation

Please do not try to install from the .zip files provided by GitHub. Rather install either install *from package* (if you just want to use it) or *from source* (if you plan to use it for development). Whichever variant you choose, please make sure that all *prerequisites* are installed.

2.1 Prerequisites

To install the `madmom` package, you must have either Python 2.7 or Python 3.3 or newer and the following packages installed:

- numpy
- scipy
- cython
- mido (for MIDI handling)
- pytest (to run the tests)
- pyaudio (to process live audio input)

If you need support for audio files other than `.wav` with a sample rate of 44.1kHz and 16 bit depth, you need `ffmpeg` (`avconv` on Ubuntu Linux has some decoding bugs, so we advise not to use it!).

Please refer to the `requirements.txt` file for the minimum required versions and make sure that these modules are up to date, otherwise it can result in unexpected errors or false computations!

2.2 Install from package

The instructions given here should be used if you just want to install the package, e.g. to run the bundled programs or use some functionality for your own project. If you intend to change anything within the *madmom* package, please follow the steps in *the next section*.

The easiest way to install the package is via pip from the PyPI (Python Package Index):

```
pip install madmom
```

This includes the latest code and trained models and will install all dependencies automatically.

You might need higher privileges (use su or sudo) to install the package, model files and scripts globally. Alternatively you can install the package locally (i.e. only for you) by adding the --user argument:

```
pip install --user madmom
```

This will also install the executable programs to a common place (e.g. /usr/local/bin), which should be in your $PATH already. If you installed the package locally, the programs will be copied to a folder which might not be included in your $PATH (e.g. ~/Library/Python/2.7/bin on Mac OS X or ~/.local/bin on Ubuntu Linux, pip will tell you). Thus the programs need to be called explicitely or you can add their install path to your $PATH environment variable:

```
export PATH='path/to/scripts':$PATH
```

2.3 Install from source

If you plan to use the package as a developer, clone the Git repository:

```
git clone --recursive https://github.com/CPJKU/madmom.git
```

Since the pre-trained model/data files are not included in this repository but rather added as a Git submodule, you either have to clone the repo recursively. This is equivalent to these steps:

```
git clone  https://github.com/CPJKU/madmom.git
cd madmom
git submodule update --init --remote
```

Then you can simply install the package in development mode:

```
python setup.py develop --user
```

To run the included tests:

```
python setup.py pytest
```

2.4 Upgrade of existing installations

To upgrade the package, please use the same mechanism (pip vs. source) as you did for installation. If you want to change from package to source, please uninstall the package first.

2.4.1 Upgrade a package

Simply upgrade the package via pip:

```
pip install --upgrade madmom [--user]
```

If some of the provided programs or models changed (please refer to the CHANGELOG) you should first uninstall the package and then reinstall:

```
pip uninstall madmom
pip install madmom [--user]
```

2.4.2 Upgrade from source

Simply pull the latest sources:

```
git pull
```

To update the models contained in the submodule:

```
git submodule update
```

If any of the `.pyx` or `.pxd` files changed, you have to recompile the modules with Cython:

```
python setup.py build_ext --inplace
```

CHAPTER 3

Usage

3.1 Executable programs

The package includes executable programs in the /bin folder. These are standalone reference implementations of the algorithms contained in the package. If you just want to try/use these programs, please follow the *instruction to install from a package*.

All scripts can be run in different modes: in single file mode to process a single audio file and write the output to STDOUT or the given output file:

```
DBNBeatTracker single [-o OUTFILE] INFILE
```

If multiple audio files should be processed, the scripts can also be run in batch mode to write the outputs to files with the given suffix:

```
DBNBeatTracker batch [-o OUTPUT_DIR] [-s OUTPUT_SUFFIX] FILES
```

If no output directory is given, the program writes the output files to same location as the audio files.

Some programs can also be run in online mode, i.e. operate on live audio signals. This requires pyaudio to be installed:

```
DBNBeatTracker online [-o OUTFILE] [INFILE]
```

The pickle mode can be used to store the used parameters to be able to exactly reproduce experiments.

Please note that the program itself as well as the modes have help messages:

```
DBNBeatTracker -h

DBNBeatTracker single -h

DBNBeatTracker batch -h

DBNBeatTracker online -h
```

(continues on next page)

(continued from previous page)

```
DBNBeatTracker pickle -h
```

will give different help messages.

3.2 Library usage

To use the library, *installing it from source* is the preferred way. Installation from package works as well, but you're limited to the functionality provided and can't extend the library.

The basic usage is:

```
import madmom
import numpy as np
```

To learn more about how to use the library please follow the *tutorials*.

CHAPTER 4

Tutorials

This page gives instructions on how to use the package. They are bundled as a loose collection of jupyter (IPython) notebooks.

You can view them online:

https://github.com/CPJKU/madmom_tutorials

CHAPTER 5

Development

As an open-source project by researchers for researchers, we highly welcome any contribution!

5.1 What to contribute

5.1.1 Give feedback

To send us general feedback, questions or ideas for improvement, please post on our mailing list.

5.1.2 Report bugs

Please report any bugs at the issue tracker on GitHub. If you are reporting a bug, please include:

- your version of madmom,
- steps to reproduce the bug, ideally reduced to as few commands as possible,
- the results you obtain, and the results you expected instead.

If you are unsure whether the experienced behaviour is intended or a bug, please just ask on our mailing list first.

5.1.3 Fix bugs

Look for anything tagged with "bug" on the issue tracker on GitHub and fix it.

5.1.4 Features

Please do not hesitate to propose any ideas at the issue tracker on GitHub. Think about posting them on our mailing list first, so we can discuss it and/or guide you through the implementation.

Alternatively, you can look for anything tagged with "feature request" or "enhancement" on the issue tracker on GitHub.

5.1.5 Write documentation

Whenever you find something not explained well, misleading or just wrong, please update it! The *Edit on GitHub* link on the top right of every documentation page and the *[source]* link for every documented entity in the API reference will help you to quickly locate the origin of any text.

5.2 How to contribute

5.2.1 Edit on GitHub

As a very easy way of just fixing issues in the documentation, use the *Edit on GitHub* link on the top right of a documentation page or the *[source]* link of an entity in the API reference to open the corresponding source file in GitHub, then click the *Edit this file* link to edit the file in your browser and send us a Pull Request.

For any more substantial changes, please follow the steps below.

5.2.2 Fork the project

First, fork the project on GitHub.

Then, follow the *general installation instructions* and, more specifically, the *installation from source*. Please note that you should clone from your fork instead.

5.2.3 Documentation

The documentation is generated with Sphinx. To build it locally, run the following commands:

```
cd docs
make html
```

Afterwards, open `docs/_build/html/index.html` to view the documentation as it would appear on readthedocs. If you changed a lot and seem to get misleading error messages or warnings, run `make clean html` to force Sphinx to recreate all files from scratch.

When writing docstrings, follow existing documentation as much as possible to ensure consistency throughout the library. For additional information on the syntax and conventions used, please refer to the following documents:

- reStructuredText Primer
- Sphinx reST markup constructs
- A Guide to NumPy/SciPy Documentation

CHAPTER 6

Citation

If you use madmom in your work, please consider citing it:

```
@inproceedings{madmom,
   Title = {{madmom: a new Python Audio and Music Signal Processing Library}},
   Author = {B{\"o}ck, Sebastian and Korzeniowski, Filip and Schl{\"u}ter, Jan and␣
↪Krebs, Florian and Widmer, Gerhard},
   Booktitle = {Proceedings of the 24th ACM International Conference on
   Multimedia},
   Month = {10},
   Year = {2016},
   Pages = {1174--1178},
   Address = {Amsterdam, The Netherlands},
   Doi = {10.1145/2964284.2973795}
}
```

CHAPTER 7

madmom.audio

This package includes audio handling functionality and low-level features. The definition of "low" may vary, but all "high"-level features (e.g. beats, onsets, etc. – basically everything you want to evaluate) should be in the *madmom.features* package.

7.1 Notes

Almost all functionality blocks are split into two classes:

1. A data class: instances are signal dependent, i.e. they operate directly on the signal and show different values for different signals.
2. A processor class: for every data class there should be a processor class with the exact same name and a "Processor" suffix. This class must inherit from madmom.Processor and define a process() method which returns a data class or inherit from madmom.SequentialProcessor or ParallelProcessor.

The data classes should be either sub-classed from numpy arrays or be indexable and iterable. This way they can be used identically to numpy arrays.

7.2 Submodules

7.2.1 madmom.audio.signal

This module contains basic signal processing functionality.

`madmom.audio.signal.`**`smooth`**`(`*signal*, *kernel*`)`

Smooth the signal along its first axis.

Parameters

signal [numpy array] Signal to be smoothed.

kernel [numpy array or int] Smoothing kernel (size).

Returns

numpy array Smoothed signal.

Notes

If *kernel* is an integer, a Hamming window of that length will be used as a smoothing kernel.

`madmom.audio.signal.`**`adjust_gain`**`(`*signal*, *gain*`)`

” Adjust the gain of the signal.

Parameters

signal [numpy array] Signal to be adjusted.

gain [float] Gain adjustment level [dB].

Returns

numpy array Signal with adjusted gain.

Notes

The signal is returned with the same dtype, thus rounding errors may occur with integer dtypes.

gain values > 0 amplify the signal and are only supported for signals with float dtype to prevent clipping and integer overflows.

`madmom.audio.signal.`**`attenuate`**`(`*signal*, *attenuation*`)`

Attenuate the signal.

Parameters

signal [numpy array] Signal to be attenuated.

attenuation [float] Attenuation level [dB].

Returns

numpy array Attenuated signal (same dtype as *signal*).

Notes

The signal is returned with the same dtype, thus rounding errors may occur with integer dtypes.

`madmom.audio.signal.`**`normalize`**`(`*signal*`)`

Normalize the signal to have maximum amplitude.

Parameters

signal [numpy array] Signal to be normalized.

Returns

numpy array Normalized signal.

Notes

Signals with float dtypes cover the range [-1, +1], signals with integer dtypes will cover the maximally possible range, e.g. [-32768, 32767] for np.int16.

The signal is returned with the same dtype, thus rounding errors may occur with integer dtypes.

`madmom.audio.signal.`**`remix`**`(`*signal*, *num_channels*`)`

Remix the signal to have the desired number of channels.

Parameters

signal [numpy array] Signal to be remixed.

num_channels [int] Number of channels.

Returns

numpy array Remixed signal (same dtype as *signal*).

Notes

This function does not support arbitrary channel number conversions. Only down-mixing to and up-mixing from mono signals is supported.

The signal is returned with the same dtype, thus rounding errors may occur with integer dtypes.

If the signal should be down-mixed to mono and has an integer dtype, it will be converted to float internally and then back to the original dtype to prevent clipping of the signal. To avoid this double conversion, convert the dtype first.

`madmom.audio.signal.`**`resample`**`(`*signal*, *sample_rate*, ***kwargs*`)`

Resample the signal.

Parameters

signal [numpy array or Signal] Signal to be resampled.

sample_rate [int] Sample rate of the signal.

kwargs [dict, optional] Keyword arguments passed to `load_ffmpeg_file()`.

Returns

numpy array or Signal Resampled signal.

Notes

This function uses `ffmpeg` to resample the signal.

`madmom.audio.signal.`**`rescale`**`(`*signal*, *dtype=<type 'numpy.float32'>*`)`

Rescale the signal to range [-1, 1] and return as float dtype.

Parameters

signal [numpy array] Signal to be remixed.

dtype [numpy dtype] Data type of the signal.

Returns

numpy array Signal rescaled to range [-1, 1].

madmom.audio.signal.**trim**(*signal*, *where='fb'*)
Trim leading and trailing zeros of the signal.

Parameters

signal [numpy array] Signal to be trimmed.

where [str, optional] A string with 'f' representing trim from front and 'b' to trim from back. Default is 'fb', trim zeros from both ends of the signal.

Returns

numpy array Trimmed signal.

madmom.audio.signal.**energy**(*signal*)
Compute the energy of a (framed) signal.

Parameters

signal [numpy array] Signal.

Returns

energy [float] Energy of the signal.

Notes

If *signal* is a *FramedSignal*, the energy is computed for each frame individually.

madmom.audio.signal.**root_mean_square**(*signal*)
Compute the root mean square of a (framed) signal. This can be used as a measurement of power.

Parameters

signal [numpy array] Signal.

Returns

rms [float] Root mean square of the signal.

Notes

If *signal* is a *FramedSignal*, the root mean square is computed for each frame individually.

madmom.audio.signal.**sound_pressure_level**(*signal*, *p_ref=None*)
Compute the sound pressure level of a (framed) signal.

Parameters

signal [numpy array] Signal.

p_ref [float, optional] Reference sound pressure level; if 'None', take the max amplitude value for the data-type, if the data-type is float, assume amplitudes are between -1 and +1.

Returns

spl [float] Sound pressure level of the signal [dB].

Notes

From http://en.wikipedia.org/wiki/Sound_pressure: Sound pressure level (SPL) or sound level is a logarithmic measure of the effective sound pressure of a sound relative to a reference value. It is measured in decibels (dB) above a standard reference level.

If *signal* is a *FramedSignal*, the sound pressure level is computed for each frame individually.

exception madmom.audio.signal.**LoadAudioFileError**(*value=None*)
Deprecated as of version 0.16. Please use madmom.io.audio.LoadAudioFileError instead. Will be removed in version 0.18.

madmom.audio.signal.**load_wave_file**(**args*, ***kwargs*)
Deprecated as of version 0.16. Please use madmom.io.audio.load_wave_file instead. Will be removed in version 0.18.

madmom.audio.signal.**write_wave_file**(**args*, ***kwargs*)
Deprecated as of version 0.16. Please use madmom.io.audio.write_wave_file instead. Will be removed in version 0.18.

madmom.audio.signal.**load_audio_file**(**args*, ***kwargs*)
Deprecated as of version 0.16. Please use madmom.io.audio.load_audio_file instead. Will be removed in version 0.18.

class madmom.audio.signal.**Signal**(*data*, *sample_rate=None*, *num_channels=None*, *start=None*, *stop=None*, *norm=False*, *gain=0.0*, *dtype=None*, ***kwargs*)
The *Signal* class represents a signal as a (memory-mapped) numpy array and enhances it with a number of attributes.

Parameters

data [numpy array, str or file handle] Signal data or file name or file handle.

sample_rate [int, optional] Desired sample rate of the signal [Hz], or 'None' to return the signal in its original rate.

num_channels [int, optional] Reduce or expand the signal to *num_channels* channels, or 'None' to return the signal with its original channels.

start [float, optional] Start position [seconds].

stop [float, optional] Stop position [seconds].

norm [bool, optional] Normalize the signal to maximum range of the data type.

gain [float, optional] Adjust the gain of the signal [dB].

dtype [numpy data type, optional] The data is returned with the given dtype. If 'None', it is returned with its original dtype, otherwise the signal gets rescaled. Integer dtypes use the complete value range, float dtypes the range [-1, +1].

Notes

sample_rate or *num_channels* can be used to set the desired sample rate and number of channels if the audio is read from file. If set to 'None' the audio signal is used as is, i.e. the sample rate and number of channels are determined directly from the audio file.

If the *data* is a numpy array, the *sample_rate* is set to the given value and *num_channels* is set to the number of columns of the array.

The *gain* can be used to adjust the level of the signal.

If both *norm* and *gain* are set, the signal is first normalized and then the gain is applied afterwards.

If *norm* or *gain* is set, the selected part of the signal is loaded into memory completely, i.e. .wav files are not memory-mapped any more.

Examples

Load a mono audio file:

```
>>> sig = Signal('tests/data/audio/sample.wav')
>>> sig
Signal([-2494, -2510, ...,   655,   639], dtype=int16)
>>> sig.sample_rate
44100
```

Load a stereo audio file, down-mix it to mono:

```
>>> sig = Signal('tests/data/audio/stereo_sample.flac', num_channels=1)
>>> sig
Signal([ 36,  36, ..., 524, 495], dtype=int16)
>>> sig.num_channels
1
```

Load and re-sample an audio file:

```
>>> sig = Signal('tests/data/audio/sample.wav', sample_rate=22050)
>>> sig
Signal([-2470, -2553, ...,   517,   677], dtype=int16)
>>> sig.sample_rate
22050
```

Load an audio file with *float32* data type (i.e. rescale it to [-1, 1]):

```
>>> sig = Signal('tests/data/audio/sample.wav', dtype=np.float32)
>>> sig
Signal([-0.07611, -0.0766 , ...,  0.01999,  0.0195 ], dtype=float32)
>>> sig.dtype
dtype('float32')
```

num_samples
Number of samples.

num_channels
Number of channels.

length
Length of signal in seconds.

write (*filename*)
Write the signal to disk as a .wav file.

Parameters

filename [str] Name of the file.

Returns

filename [str] Name of the written file.

energy()
Energy of signal.

root_mean_square()
Root mean square of signal.

rms()
Root mean square of signal.

sound_pressure_level()
Sound pressure level of signal.

spl()
Sound pressure level of signal.

class madmom.audio.signal.**SignalProcessor**(*sample_rate=None*, *num_channels=None*, *start=None*, *stop=None*, *norm=False*, *gain=0.0*, *dtype=None*, ***kwargs*)

The *SignalProcessor* class is a basic signal processor.

Parameters

sample_rate [int, optional] Sample rate of the signal [Hz]; if set the signal will be re-sampled to that sample rate; if 'None' the sample rate of the audio file will be used.

num_channels [int, optional] Number of channels of the signal; if set, the signal will be reduced to that number of channels; if 'None' as many channels as present in the audio file are returned.

start [float, optional] Start position [seconds].

stop [float, optional] Stop position [seconds].

norm [bool, optional] Normalize the signal to the range [-1, +1].

gain [float, optional] Adjust the gain of the signal [dB].

dtype [numpy data type, optional] The data is returned with the given dtype. If 'None', it is returned with its original dtype, otherwise the signal gets rescaled. Integer dtypes use the complete value range, float dtypes the range [-1, +1].

Examples

Processor for loading the first two seconds of an audio file, re-sampling it to 22.05 kHz and down-mixing it to mono:

```
>>> proc = SignalProcessor(sample_rate=22050, num_channels=1, stop=2)
>>> sig = proc('tests/data/audio/sample.wav')
>>> sig
Signal([-2470, -2553, ...,  -173,  -265], dtype=int16)
>>> sig.sample_rate
22050
>>> sig.num_channels
1
>>> sig.length
2.0
```

process(*data*, ***kwargs*)
Processes the given audio file.

Parameters

data [numpy array, str or file handle] Data to be processed.

kwargs [dict, optional] Keyword arguments passed to `Signal`.

Returns

signal [`Signal` instance] `Signal` instance.

static add_arguments(*parser*, *sample_rate=None*, *mono=None*, *start=None*, *stop=None*, *norm=None*, *gain=None*)

Add signal processing related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

sample_rate [int, optional] Re-sample the signal to this sample rate [Hz].

mono [bool, optional] Down-mix the signal to mono.

start [float, optional] Start position [seconds].

stop [float, optional] Stop position [seconds].

norm [bool, optional] Normalize the signal to the range [-1, +1].

gain [float, optional] Adjust the gain of the signal [dB].

Returns

argparse argument group Signal processing argument parser group.

Notes

Parameters are included in the group only if they are not 'None'. To include *start* and *stop* arguments with a default value of 'None', i.e. do not set any start or stop time, they can be set to 'True'.

madmom.audio.signal.**signal_frame**(*signal*, *index*, *frame_size*, *hop_size*, *origin=0*)

This function returns frame at *index* of the *signal*.

Parameters

signal [numpy array] Signal.

index [int] Index of the frame to return.

frame_size [int] Size of each frame in samples.

hop_size [float] Hop size in samples between adjacent frames.

origin [int] Location of the window center relative to the signal position.

Returns

frame [numpy array] Requested frame of the signal.

Notes

The reference sample of the first frame (index == 0) refers to the first sample of the *signal*, and each following frame is placed *hop_size* samples after the previous one.

The window is always centered around this reference sample. Its location relative to the reference sample can be set with the *origin* parameter. Arbitrary integer values can be given:

- zero centers the window on its reference sample

- negative values shift the window to the right
- positive values shift the window to the left

An *origin* of half the size of the *frame_size* results in windows located to the left of the reference sample, i.e. the first frame starts at the first sample of the signal.

The part of the frame which is not covered by the signal is padded with zeros.

This function is totally independent of the length of the signal. Thus, contrary to common indexing, the index '-1' refers NOT to the last frame of the signal, but instead the frame left of the first frame is returned.

class `madmom.audio.signal.`**`FramedSignal`**(*signal*, *frame_size=2048*, *hop_size=441.0*, *fps=None*, *origin=0*, *end='normal'*, *num_frames=None*, ***kwargs*)

The `FramedSignal` splits a `Signal` into frames and makes it iterable and indexable.

Parameters

signal [`Signal` instance] Signal to be split into frames.

frame_size [int, optional] Size of one frame [samples].

hop_size [float, optional] Progress *hop_size* samples between adjacent frames.

fps [float, optional] Use given frames per second; if set, this computes and overwrites the given *hop_size* value.

origin [int, optional] Location of the window relative to the reference sample of a frame.

end [int or str, optional] End of signal handling (see notes below).

num_frames [int, optional] Number of frames to return.

kwargs [dict, optional] If no `Signal` instance was given, one is instantiated with these additional keyword arguments.

Notes

The `FramedSignal` class is implemented as an iterator. It splits the given *signal* automatically into frames of *frame_size* length with *hop_size* samples (can be float, normal rounding applies) between the frames. The reference sample of the first frame refers to the first sample of the *signal*.

The location of the window relative to the reference sample of a frame can be set with the *origin* parameter (with the same behaviour as used by `scipy.ndimage` filters). Arbitrary integer values can be given:

- zero centers the window on its reference sample,
- negative values shift the window to the right,
- positive values shift the window to the left.

Additionally, it can have the following literal values:

- 'center', 'offline': the window is centered on its reference sample,
- 'left', 'past', 'online': the window is located to the left of its reference sample (including the reference sample),
- 'right', 'future', 'stream': the window is located to the right of its reference sample.

The *end* parameter is used to handle the end of signal behaviour and can have these values:

- 'normal': stop as soon as the whole signal got covered by at least one frame (i.e. pad maximally one frame),

- 'extend': frames are returned as long as part of the frame overlaps with the signal to cover the whole signal.

Alternatively, *num_frames* can be used to retrieve a fixed number of frames.

In order to be able to stack multiple frames obtained with different frame sizes, the number of frames to be returned must be independent from the set *frame_size*. It is not guaranteed that every sample of the signal is returned in a frame unless the *origin* is either 'right' or 'future'.

If used in online real-time mode the parameters *origin* and *num_frames* should be set to 'stream' and 1, respectively.

Examples

To chop a `Signal` (or anything a `Signal` can be instantiated from) into overlapping frames of size 2048 with adjacent frames being 441 samples apart:

```
>>> sig = Signal('tests/data/audio/sample.wav')
>>> sig
Signal([-2494, -2510, ...,   655,   639], dtype=int16)
>>> frames = FramedSignal(sig, frame_size=2048, hop_size=441)
>>> frames
<madmom.audio.signal.FramedSignal object at 0x...>
>>> frames[0]
Signal([    0,     0, ..., -4666, -4589], dtype=int16)
>>> frames[10]
Signal([-6156, -5645, ...,  -253,   671], dtype=int16)
>>> frames.fps
100.0
```

Instead of passing a `Signal` instance as the first argument, anything a `Signal` can be instantiated from (e.g. a file name) can be used. We can also set the frames per second (*fps*) instead, they get converted to *hop_size* based on the *sample_rate* of the signal:

```
>>> frames = FramedSignal('tests/data/audio/sample.wav', fps=100)
>>> frames
<madmom.audio.signal.FramedSignal object at 0x...>
>>> frames[0]
Signal([    0,     0, ..., -4666, -4589], dtype=int16)
>>> frames.frame_size, frames.hop_size
(2048, 441.0)
```

When trying to access an out of range frame, an IndexError is raised. Thus the FramedSignal can be used the same way as a numpy array or any other iterable.

```
>>> frames = FramedSignal('tests/data/audio/sample.wav')
>>> frames.num_frames
281
>>> frames[281]
Traceback (most recent call last):
IndexError: end of signal reached
>>> frames.shape
(281, 2048)
```

Slices are FramedSignals itself:

```
>>> frames[:4]
<madmom.audio.signal.FramedSignal object at 0x...>
```

To obtain a numpy array from a FramedSignal, simply use np.array() on the full FramedSignal or a slice of it. Please note, that this requires a full memory copy.

```
>>> np.array(frames[2:4])
array([[    0,      0, ..., -5316, -5405],
       [ 2215,   2281, ...,   561,   653]], dtype=int16)
```

frame_rate
Frame rate (same as fps).

fps
Frames per second.

overlap_factor
Overlapping factor of two adjacent frames.

shape
Shape of the FramedSignal (num_frames, frame_size[, num_channels]).

ndim
Dimensionality of the FramedSignal.

energy()
Energy of the individual frames.

root_mean_square()
Root mean square of the individual frames.

rms()
Root mean square of the individual frames.

sound_pressure_level()
Sound pressure level of the individual frames.

spl()
Sound pressure level of the individual frames.

class madmom.audio.signal.**FramedSignalProcessor**(*frame_size=2048*, *hop_size=441.0*, *fps=None*, *origin=0*, *end='normal'*, *num_frames=None*, ***kwargs*)

Slice a Signal into frames.

Parameters

frame_size [int, optional] Size of one frame [samples].

hop_size [float, optional] Progress *hop_size* samples between adjacent frames.

fps [float, optional] Use given frames per second; if set, this computes and overwrites the given *hop_size* value.

origin [int, optional] Location of the window relative to the reference sample of a frame.

end [int or str, optional] End of signal handling (see `FramedSignal`).

num_frames [int, optional] Number of frames to return.

Notes

When operating on live audio signals, *origin* must be set to 'stream' in order to retrieve always the last *frame_size* samples.

Examples

Processor for chopping a `Signal` (or anything a `Signal` can be instantiated from) into overlapping frames of size 2048, and a frame rate of 100 frames per second:

```
>>> proc = FramedSignalProcessor(frame_size=2048, fps=100)
>>> frames = proc('tests/data/audio/sample.wav')
>>> frames
<madmom.audio.signal.FramedSignal object at 0x...>
>>> frames[0]
Signal([    0,     0, ..., -4666, -4589], dtype=int16)
>>> frames[10]
Signal([-6156, -5645, ...,  -253,   671], dtype=int16)
>>> frames.hop_size
441.0
```

process (*data*, ***kwargs*)
Slice the signal into (overlapping) frames.

Parameters

data [`Signal` instance] Signal to be sliced into frames.

kwargs [dict, optional] Keyword arguments passed to `FramedSignal`.

Returns

frames [`FramedSignal` instance] FramedSignal instance

static add_arguments (*parser*, *frame_size=2048*, *fps=None*, *online=None*)
Add signal framing related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

frame_size [int, optional] Size of one frame in samples.

fps [float, optional] Frames per second.

online [bool, optional] Online mode (use only past signal information, i.e. align the window to the left of the reference sample).

Returns

argparse argument group Signal framing argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

class madmom.audio.signal.**Stream** (*sample_rate=None*, *num_channels=None*, *dtype=<type 'numpy.float32'>*, *frame_size=2048*, *hop_size=441.0*, *fps=None*, ***kwargs*)
A Stream handles live (i.e. online, real-time) audio input via PyAudio.

Parameters

sample_rate [int] Sample rate of the signal.

num_channels [int, optional] Number of channels.

dtype [numpy dtype, optional] Data type for the signal.

frame_size [int, optional] Size of one frame [samples].

hop_size [int, optional] Progress *hop_size* samples between adjacent frames.

fps [float, optional] Use given frames per second; if set, this computes and overwrites the given *hop_size* value (the resulting *hop_size* must be an integer).

queue_size [int] Size of the FIFO (first in first out) queue. If the queue is full and new audio samples arrive, the oldest item in the queue will be dropped.

Notes

Stream is implemented as an iterable which blocks until enough new data is available.

shape
Shape of the Stream (None, frame_size[, num_channels]).

7.2.2 madmom.audio.filters

This module contains filter and filterbank related functionality.

madmom.audio.filters.**hz2mel**(*f*)
Convert Hz frequencies to Mel.

Parameters

f [numpy array] Input frequencies [Hz].

Returns

m [numpy array] Frequencies in Mel [Mel].

madmom.audio.filters.**mel2hz**(*m*)
Convert Mel frequencies to Hz.

Parameters

m [numpy array] Input frequencies [Mel].

Returns

f: numpy array Frequencies in Hz [Hz].

madmom.audio.filters.**mel_frequencies**(*num_bands*, *fmin*, *fmax*)
Returns frequencies aligned on the Mel scale.

Parameters

num_bands [int] Number of bands.

fmin [float] Minimum frequency [Hz].

fmax [float] Maximum frequency [Hz].

Returns

mel_frequencies: numpy array Frequencies with Mel spacing [Hz].

madmom.audio.filters.**log_frequencies**(*bands_per_octave*, *fmin*, *fmax*, *fref=440.0*)
Returns frequencies aligned on a logarithmic frequency scale.

Parameters

bands_per_octave [int] Number of filter bands per octave.

fmin [float] Minimum frequency [Hz].

fmax [float] Maximum frequency [Hz].

fref [float, optional] Tuning frequency [Hz].

Returns

log_frequencies [numpy array] Logarithmically spaced frequencies [Hz].

Notes

If *bands_per_octave* = 12 and *fref* = 440 are used, the frequencies are equivalent to MIDI notes.

`madmom.audio.filters.`**`semitone_frequencies`**(*fmin*, *fmax*, *fref=440.0*)
Returns frequencies separated by semitones.

Parameters

fmin [float] Minimum frequency [Hz].

fmax [float] Maximum frequency [Hz].

fref [float, optional] Tuning frequency of A4 [Hz].

Returns

semitone_frequencies [numpy array] Semitone frequencies [Hz].

`madmom.audio.filters.`**`hz2midi`**(*f*, *fref=440.0*)
Convert frequencies to the corresponding MIDI notes.

Parameters

f [numpy array] Input frequencies [Hz].

fref [float, optional] Tuning frequency of A4 [Hz].

Returns

m [numpy array] MIDI notes

Notes

For details see: at http://www.phys.unsw.edu.au/jw/notes.html This function does not necessarily return a valid MIDI Note, you may need to round it to the nearest integer.

`madmom.audio.filters.`**`midi2hz`**(*m*, *fref=440.0*)
Convert MIDI notes to corresponding frequencies.

Parameters

m [numpy array] Input MIDI notes.

fref [float, optional] Tuning frequency of A4 [Hz].

Returns

f [numpy array] Corresponding frequencies [Hz].

`madmom.audio.filters.`**`midi_frequencies`**(*fmin*, *fmax*, *fref=440.0*)
Returns frequencies separated by semitones.

Parameters

fmin [float] Minimum frequency [Hz].

fmax [float] Maximum frequency [Hz].

fref [float, optional] Tuning frequency of A4 [Hz].

Returns

semitone_frequencies [numpy array] Semitone frequencies [Hz].

`madmom.audio.filters.`**`hz2erb`**(*f*)

Convert Hz to ERB.

Parameters

f [numpy array] Input frequencies [Hz].

Returns

e [numpy array] Frequencies in ERB [ERB].

Notes

Information about the ERB scale can be found at: https://ccrma.stanford.edu/~jos/bbt/Equivalent_Rectangular_Bandwidth.html

`madmom.audio.filters.`**`erb2hz`**(*e*)

Convert ERB scaled frequencies to Hz.

Parameters

e [numpy array] Input frequencies [ERB].

Returns

f [numpy array] Frequencies in Hz [Hz].

Notes

Information about the ERB scale can be found at: https://ccrma.stanford.edu/~jos/bbt/Equivalent_Rectangular_Bandwidth.html

`madmom.audio.filters.`**`frequencies2bins`**(*frequencies*, *bin_frequencies*, *unique_bins=False*)

Map frequencies to the closest corresponding bins.

Parameters

frequencies [numpy array] Input frequencies [Hz].

bin_frequencies [numpy array] Frequencies of the (FFT) bins [Hz].

unique_bins [bool, optional] Return only unique bins, i.e. remove all duplicate bins resulting from insufficient resolution at low frequencies.

Returns

bins [numpy array] Corresponding (unique) bins.

Notes

It can be important to return only unique bins, otherwise the lower frequency bins can be given too much weight if all bins are simply summed up (as in the spectral flux onset detection).

`madmom.audio.filters.`**`bins2frequencies`**(*bins*, *bin_frequencies*)
Convert bins to the corresponding frequencies.

Parameters

bins [numpy array] Bins (e.g. FFT bins).

bin_frequencies [numpy array] Frequencies of the (FFT) bins [Hz].

Returns

f [numpy array] Corresponding frequencies [Hz].

`class` `madmom.audio.filters.`**`Filter`**(*data*, *start=0*, *norm=False*)
Generic Filter class.

Parameters

data [1D numpy array] Filter data.

start [int, optional] Start position (see notes).

norm [bool, optional] Normalize the filter area to 1.

Notes

The start position is mandatory if a Filter should be used for the creation of a Filterbank.

`classmethod band_bins`(*bins*, ***kwargs*)
Must yield the center/crossover bins needed for filter creation.

Parameters

bins [numpy array] Center/crossover bins used for the creation of filters.

kwargs [dict, optional] Additional parameters for for the creation of filters (e.g. if the filters should overlap or not).

`classmethod filters`(*bins*, *norm*, ***kwargs*)
Create a list with filters for the given bins.

Parameters

bins [list or numpy array] Center/crossover bins of the filters.

norm [bool] Normalize the area of the filter(s) to 1.

kwargs [dict, optional] Additional parameters passed to `band_bins()` (e.g. if the filters should overlap or not).

Returns

filters [list] Filter(s) for the given bins.

`class` `madmom.audio.filters.`**`TriangularFilter`**(*start*, *center*, *stop*, *norm=False*)
Triangular filter class.

Create a triangular shaped filter with length *stop*, height 1 (unless normalized) with indices <= *start* set to 0.

Parameters

start [int] Start bin of the filter.

center [int] Center bin of the filter.

stop [int] Stop bin of the filter.

norm [bool, optional] Normalize the area of the filter to 1.

classmethod band_bins(*bins*, *overlap=True*)

Yields start, center and stop bins for creation of triangular filters.

Parameters

bins [list or numpy array] Center bins of filters.

overlap [bool, optional] Filters should overlap (see notes).

Yields

start [int] Start bin of the filter.

center [int] Center bin of the filter.

stop [int] Stop bin of the filter.

Notes

If *overlap* is 'False', the *start* and *stop* bins of the filters are interpolated between the centre bins, normal rounding applies.

class madmom.audio.filters.**RectangularFilter**(*start*, *stop*, *norm=False*)

Rectangular filter class.

Create a rectangular shaped filter with length *stop*, height 1 (unless normalized) with indices < *start* set to 0.

Parameters

start [int] Start bin of the filter.

stop [int] Stop bin of the filter.

norm [bool, optional] Normalize the area of the filter to 1.

classmethod band_bins(*bins*, *overlap=False*)

Yields start and stop bins and normalization info for creation of rectangular filters.

Parameters

bins [list or numpy array] Crossover bins of filters.

overlap [bool, optional] Filters should overlap.

Yields

start [int] Start bin of the filter.

stop [int] Stop bin of the filter.

class madmom.audio.filters.**Filterbank**(*data*, *bin_frequencies*)

Generic filterbank class.

A Filterbank is a simple numpy array enhanced with several additional attributes, e.g. number of bands.

A Filterbank has a shape of (num_bins, num_bands) and can be used to filter a spectrogram of shape (num_frames, num_bins) to (num_frames, num_bands).

Parameters

data [numpy array, shape (num_bins, num_bands)] Data of the filterbank .

bin_frequencies [numpy array, shape (num_bins,)] Frequencies of the bins [Hz].

Notes

The length of *bin_frequencies* must be equal to the first dimension of the given *data* array.

classmethod from_filters(*filters*, *bin_frequencies*)
Create a filterbank with possibly multiple filters per band.

Parameters

filters [list (of lists) of Filters] List of Filters (per band); if multiple filters per band are desired, they should be also contained in a list, resulting in a list of lists of Filters.

bin_frequencies [numpy array] Frequencies of the bins (needed to determine the expected size of the filterbank).

Returns

filterbank [*Filterbank* instance] Filterbank with respective filter elements.

num_bins
Number of bins.

num_bands
Number of bands.

corner_frequencies
Corner frequencies of the filter bands.

center_frequencies
Center frequencies of the filter bands.

fmin
Minimum frequency of the filterbank.

fmax
Maximum frequency of the filterbank.

class madmom.audio.filters.**FilterbankProcessor**(*data*, *bin_frequencies*)
Generic filterbank processor class.

A FilterbankProcessor is a simple wrapper for Filterbank which adds a process() method.

See also:

Filterbank

process(*data*)
Filter the given data with the Filterbank.

Parameters

data [2D numpy array] Data to be filtered.

Returns

——-

filt_data [numpy array] Filtered data.

Notes

This method makes the `Filterbank` act as a `Processor`.

static add_arguments(*parser*, *filterbank=None*, *num_bands=None*, *crossover_frequencies=None*, *fmin=None*, *fmax=None*, *norm_filters=None*, *unique_filters=None*)

Add filterbank related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

filterbank [`audio.filters.Filterbank`, optional] Use a filterbank of that type.

num_bands [int or list, optional] Number of bands (per octave).

crossover_frequencies [list or numpy array, optional] List of crossover frequencies at which the *spectrogram* is split into bands.

fmin [float, optional] Minimum frequency of the filterbank [Hz].

fmax [float, optional] Maximum frequency of the filterbank [Hz].

norm_filters [bool, optional] Normalize the filters of the filterbank to area 1.

unique_filters [bool, optional] Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

Returns

argparse argument group Filterbank argument parser group.

Notes

Parameters are included in the group only if they are not 'None'. Depending on the type of the *filterbank*, either *num_bands* or *crossover_frequencies* should be used.

class madmom.audio.filters.**MelFilterbank**(*bin_frequencies*, *num_bands=40*, *fmin=20.0*, *fmax=17000.0*, *norm_filters=True*, *unique_filters=True*, ***kwargs*)

Mel filterbank class.

Parameters

bin_frequencies [numpy array] Frequencies of the bins [Hz].

num_bands [int, optional] Number of filter bands.

fmin [float, optional] Minimum frequency of the filterbank [Hz].

fmax [float, optional] Maximum frequency of the filterbank [Hz].

norm_filters [bool, optional] Normalize the filters to area 1.

unique_filters [bool, optional] Keep only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

Notes

Because of rounding and mapping of frequencies to bins and back to frequencies, the actual minimum, maximum and center frequencies do not necessarily match the parameters given.

class madmom.audio.filters.**LogarithmicFilterbank**(*bin_frequencies*, *num_bands=12*, *fmin=30.0*, *fmax=17000.0*, *fref=440.0*, *norm_filters=True*, *unique_filters=True*, *bands_per_octave=True*)

Logarithmic filterbank class.

Parameters

bin_frequencies [numpy array] Frequencies of the bins [Hz].

num_bands [int, optional] Number of filter bands (per octave).

fmin [float, optional] Minimum frequency of the filterbank [Hz].

fmax [float, optional] Maximum frequency of the filterbank [Hz].

fref [float, optional] Tuning frequency of the filterbank [Hz].

norm_filters [bool, optional] Normalize the filters to area 1.

unique_filters [bool, optional] Keep only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

bands_per_octave [bool, optional] Indicates whether *num_bands* is given as number of bands per octave ('True', default) or as an absolute number of bands ('False').

Notes

num_bands sets either the number of bands per octave or the total number of bands, depending on the setting of *bands_per_octave*. *num_bands* is used to set also the number of bands per octave to keep the argument for all classes the same. If 12 bands per octave are used, a filterbank with semitone spacing is created.

madmom.audio.filters.**LogFilterbank**

alias of *madmom.audio.filters.LogarithmicFilterbank*

class madmom.audio.filters.**RectangularFilterbank**(*bin_frequencies*, *crossover_frequencies*, *fmin=30.0*, *fmax=17000.0*, *norm_filters=True*, *unique_filters=True*)

Rectangular filterbank class.

Parameters

bin_frequencies [numpy array] Frequencies of the bins [Hz].

crossover_frequencies [list or numpy array] Crossover frequencies of the bands [Hz].

fmin [float, optional] Minimum frequency of the filterbank [Hz].

fmax [float, optional] Maximum frequency of the filterbank [Hz].

norm_filters [bool, optional] Normalize the filters to area 1.

unique_filters [bool, optional] Keep only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

class madmom.audio.filters.**SemitoneBandpassFilterbank**(*order=4*, *passband_ripple=1*, *stopband_rejection=50*, *q_factor=25*, *fmin=27.5*, *fmax=4200.0*, *fref=440.0*)

Time domain semitone filterbank of elliptic filters as proposed in [1].

Parameters

order [int, optional] Order of elliptic filters.

passband_ripple [float, optional] Maximum ripple allowed below unity gain in the passband [dB].

stopband_rejection [float, optional] Minimum attenuation required in the stop band [dB].

q_factor [int, optional] Q-factor of the filters.

fmin [float, optional] Minimum frequency of the filterbank [Hz].

fmax [float, optional] Maximum frequency of the filterbank [Hz].

fref [float, optional] Reference frequency for the first bandpass filter [Hz].

Notes

This is a time domain filterbank, thus it cannot be used as the other time-frequency filterbanks of this module. Instead of `np.dot()` use `scipy.signal.filtfilt()` to filter a signal.

References

[1]

`num_bands`
Number of bands.

`fmin`
Minimum frequency of the filterbank.

`fmax`
Maximum frequency of the filterbank.

7.2.3 madmom.audio.comb_filters

This module contains comb-filter and comb-filterbank functionality.

class `madmom.audio.comb_filters.`**`CombFilterbankProcessor`**
CombFilterbankProcessor class.

Parameters

filter_function [filter function or str] Filter function to use {feed_forward_comb_filter, feed_backward_comb_filter} or a string literal {'forward', 'backward'}.

tau [list or numpy array, shape (N,)] Delay length(s) [frames].

alpha [list or numpy array, shape (N,)] Corresponding scaling factor(s).

Notes

tau and *alpha* must have the same length.

Examples

Create a processor and then filter the given signal with it. The direction of the comb filter function can be given as a literal:

```
>>> x = np.array([0, 0, 1, 0, 0, 1, 0, 0, 1])
>>> proc = CombFilterbankProcessor('forward', [2, 3], [0.5, 0.5])
>>> proc(x)
array([[ 0. ,  0. ],
       [ 0. ,  0. ],
       [ 1. ,  1. ],
       [ 0. ,  0. ],
       [ 0.5,  0. ],
       [ 1. ,  1.5],
       [ 0. ,  0. ],
       [ 0.5,  0. ],
       [ 1. ,  1.5]])
```

```
>>> proc = CombFilterbankProcessor('backward', [2, 3], [0.5, 0.5])
>>> proc(x)
array([[ 0.   ,  0.   ],
       [ 0.   ,  0.   ],
       [ 1.   ,  1.   ],
       [ 0.   ,  0.   ],
       [ 0.5  ,  0.   ],
       [ 1.   ,  1.5  ],
       [ 0.25 ,  0.   ],
       [ 0.5  ,  0.   ],
       [ 1.125,  1.75 ]])
```

process (*self*, *data*)
Process the given data with the comb filter.

Parameters

data [numpy array] Data to be filtered/processed.

Returns

comb_filtered_data [numpy array] Comb filtered data with the different taus aligned along the (new) last dimension.

madmom.audio.comb_filters.**comb_filter** (*signal*, *filter_function*, *tau*, *alpha*)
Filter the signal with a bank of either feed forward or backward comb filters.

Parameters

signal [numpy array] Signal.

filter_function [{feed_forward_comb_filter, feed_backward_comb_filter}] Filter function to use (feed forward or backward).

tau [list or numpy array, shape (N,)] Delay length(s) [frames].

alpha [list or numpy array, shape (N,)] Corresponding scaling factor(s).

Returns

comb_filtered_signal [numpy array] Comb filtered signal with the different taus aligned along the (new) last dimension.

Notes

tau and *alpha* must be of same length.

Examples

Filter the given signal with a bank of resonating comb filters.

```
>>> x = np.array([0, 0, 1, 0, 0, 1, 0, 0, 1])
>>> comb_filter(x, feed_forward_comb_filter, [2, 3], [0.5, 0.5])
array([[ 0. ,  0. ],
       [ 0. ,  0. ],
       [ 1. ,  1. ],
       [ 0. ,  0. ],
       [ 0.5,  0. ],
       [ 1. ,  1.5],
       [ 0. ,  0. ],
       [ 0.5,  0. ],
       [ 1. ,  1.5]])
```

Same for a backward filter:

```
>>> comb_filter(x, feed_backward_comb_filter, [2, 3], [0.5, 0.5])
array([[ 0.   ,  0.   ],
       [ 0.   ,  0.   ],
       [ 1.   ,  1.   ],
       [ 0.   ,  0.   ],
       [ 0.5  ,  0.   ],
       [ 1.   ,  1.5  ],
       [ 0.25 ,  0.   ],
       [ 0.5  ,  0.   ],
       [ 1.125,  1.75 ]])
```

madmom.audio.comb_filters.**feed_backward_comb_filter**(*signal*, *tau*, *alpha*)

Filter the signal with a feed backward comb filter.

Parameters

signal [numpy array] Signal.

tau [int] Delay length.

alpha [float] Scaling factor.

Returns

comb_filtered_signal [numpy array] Comb filtered signal, float dtype.

Notes

y[n] = x[n] + α * y[n - τ] is used as a filter function.

Examples

Comb filter the given signal:

```
>>> x = np.array([0, 0, 1, 0, 0, 1, 0, 0, 1])
>>> feed_backward_comb_filter(x, tau=3, alpha=0.5)
array([ 0.  ,  0.  ,  1.  ,  0.  ,  0.  ,  1.5 ,  0.  ,  0.  ,  1.75])
```

madmom.audio.comb_filters.**feed_forward_comb_filter**(*signal*, *tau*, *alpha*)
Filter the signal with a feed forward comb filter.

Parameters

signal [numpy array] Signal.

tau [int] Delay length.

alpha [float] Scaling factor.

Returns

comb_filtered_signal [numpy array] Comb filtered signal, float dtype

Notes

y[n] = x[n] + α * x[n - τ] is used as a filter function.

Examples

Comb filter the given signal:

```
>>> x = np.array([0, 0, 1, 0, 0, 1, 0, 0, 1])
>>> feed_forward_comb_filter(x, tau=3, alpha=0.5)
array([ 0. ,  0. ,  1. ,  0. ,  0. ,  1.5,  0. ,  0. ,  1.5])
```

7.2.4 madmom.audio.stft

This module contains Short-Time Fourier Transform (STFT) related functionality.

madmom.audio.stft.**fft_frequencies**(*num_fft_bins*, *sample_rate*)
Frequencies of the FFT bins.

Parameters

num_fft_bins [int] Number of FFT bins (i.e. half the FFT length).

sample_rate [float] Sample rate of the signal.

Returns

fft_frequencies [numpy array] Frequencies of the FFT bins [Hz].

madmom.audio.stft.**stft**(*frames*, *window*, *fft_size=None*, *circular_shift=False*, *include_nyquist=False*, *fftw=None*)
Calculates the complex Short-Time Fourier Transform (STFT) of the given framed signal.

Parameters

frames [numpy array or iterable, shape (num_frames, frame_size)] Framed signal (e.g. `FramedSignal` instance)

window [numpy array, shape (frame_size,)] Window (function).

fft_size [int, optional] FFT size (should be a power of 2); if 'None', the 'frame_size' given by *frames* is used; if the given *fft_size* is greater than the 'frame_size', the frames are zero-padded, if smaller truncated.

circular_shift [bool, optional] Circular shift the individual frames before performing the FFT; needed for correct phase.

include_nyquist [bool, optional] Include the Nyquist frequency bin (sample rate / 2) in returned STFT.

fftw [`pyfftw.FFTW` instance, optional] If a `pyfftw.FFTW` object is given it is used to compute the STFT with the FFTW library. Requires 'pyfftw'.

Returns

stft [numpy array, shape (num_frames, frame_size)] The complex STFT of the framed signal.

`madmom.audio.stft.`**`phase`**`(`*stft*`)`

Returns the phase of the complex STFT of a signal.

Parameters

stft [numpy array, shape (num_frames, frame_size)] The complex STFT of a signal.

Returns

phase [numpy array] Phase of the STFT.

`madmom.audio.stft.`**`local_group_delay`**`(`*phase*`)`

Returns the local group delay of the phase of a signal.

Parameters

phase [numpy array, shape (num_frames, frame_size)] Phase of the STFT of a signal.

Returns

lgd [numpy array] Local group delay of the phase.

`madmom.audio.stft.`**`lgd`**`(`*phase*`)`

Returns the local group delay of the phase of a signal.

Parameters

phase [numpy array, shape (num_frames, frame_size)] Phase of the STFT of a signal.

Returns

lgd [numpy array] Local group delay of the phase.

`class` `madmom.audio.stft.`**`ShortTimeFourierTransform`**`(`*frames*, *window=<function hanning>*, *fft_size=None*, *circular_shift=False*, *include_nyquist=False*, *fft_window=None*, *fftw=None*, ***kwargs*`)`

ShortTimeFourierTransform class.

Parameters

frames [*`audio.signal.FramedSignal`* instance] Framed signal.

window [numpy ufunc or numpy array, optional] Window (function); if a function (e.g. *np.hanning*) is given, a window with the frame size of *frames* and the given shape is created.

fft_size [int, optional] FFT size (should be a power of 2); if 'None', the *frame_size* given by *frames* is used, if the given *fft_size* is greater than the *frame_size*, the frames are zero-padded accordingly.

circular_shift [bool, optional] Circular shift the individual frames before performing the FFT; needed for correct phase.

include_nyquist [bool, optional] Include the Nyquist frequency bin (sample rate / 2).

fftw [`pyfftw.FFTW` instance, optional] If a `pyfftw.FFTW` object is given it is used to compute the STFT with the FFTW library. If 'None', a new `pyfftw.FFTW` object is built. Requires 'pyfftw'.

kwargs [dict, optional] If no *audio.signal.FramedSignal* instance was given, one is instantiated with these additional keyword arguments.

Notes

If the `Signal` (wrapped in the `FramedSignal`) has an integer dtype, the *window* is automatically scaled as if the *signal* had a float dtype with the values being in the range [-1, 1]. This results in same valued STFTs independently of the dtype of the signal. On the other hand, this prevents extra memory consumption since the data-type of the signal does not need to be converted (and if no decoding is needed, the audio signal can be memory-mapped).

Examples

Create a *ShortTimeFourierTransform* from a `Signal` or `FramedSignal`:

```
>>> sig = Signal('tests/data/audio/sample.wav')
>>> sig
Signal([-2494, -2510, ...,   655,   639], dtype=int16)
>>> frames = FramedSignal(sig, frame_size=2048, hop_size=441)
>>> frames
<madmom.audio.signal.FramedSignal object at 0x...>
>>> stft = ShortTimeFourierTransform(frames)
>>> stft
ShortTimeFourierTransform([[-3.15249+0.j     ,  2.62216-3.02425j, ...,
                            -0.03634-0.00005j,  0.0367 +0.00029j],
                           [-4.28429+0.j     ,  2.02009+2.01264j, ...,
                            -0.01981-0.00933j, -0.00536+0.02162j],
                           ...,
                           [-4.92274+0.j     ,  4.09839-9.42525j, ...,
                             0.0055 -0.00257j,  0.00137+0.00577j],
                           [-9.22709+0.j     ,  8.76929+4.0005j , ...,
                             0.00981-0.00014j, -0.00984+0.00006j]],
                          dtype=complex64)
```

A ShortTimeFourierTransform can be instantiated directly from a file name:

```
>>> stft = ShortTimeFourierTransform('tests/data/audio/sample.wav')
>>> stft
ShortTimeFourierTransform([[...]], dtype=complex64)
```

Doing the same with a Signal of float data-type will result in a STFT of same value range (rounding errors will occur of course):

```
>>> sig = Signal('tests/data/audio/sample.wav', dtype=np.float)
>>> sig
Signal([-0.07611, -0.0766 , ...,  0.01999,  0.0195 ])
>>> frames = FramedSignal(sig, frame_size=2048, hop_size=441)
>>> frames
<madmom.audio.signal.FramedSignal object at 0x...>
>>> stft = ShortTimeFourierTransform(frames)
>>> stft
ShortTimeFourierTransform([[-3.1524 +0.j     ,  2.62208-3.02415j, ...,
                            -0.03633-0.00005j,  0.0367 +0.00029j],
                           [-4.28416+0.j     ,  2.02003+2.01257j, ...,
                            -0.01981-0.00933j, -0.00536+0.02162j],
                           ...,
                           [-4.92259+0.j     ,  4.09827-9.42496j, ...,
                             0.0055 -0.00257j,  0.00137+0.00577j],
                           [-9.22681+0.j     ,  8.76902+4.00038j, ...,
                             0.00981-0.00014j, -0.00984+0.00006j]],
                          dtype=complex64)
```

Additional arguments are passed to `FramedSignal` and `Signal` respectively:

```
>>> stft = ShortTimeFourierTransform('tests/data/audio/sample.wav', frame_
↪size=2048, fps=100, sample_rate=22050)
>>> stft.frames
<madmom.audio.signal.FramedSignal object at 0x...>
>>> stft.frames.frame_size
2048
>>> stft.frames.hop_size
220.5
>>> stft.frames.signal.sample_rate
22050
```

bin_frequencies
Bin frequencies.

spec (***kwargs*)
Returns the magnitude spectrogram of the STFT.

Parameters

kwargs [dict, optional] Keyword arguments passed to *audio.spectrogram.Spectrogram*.

Returns

spec [*audio.spectrogram.Spectrogram*] *audio.spectrogram.Spectrogram* instance.

phase (***kwargs*)
Returns the phase of the STFT.

Parameters

kwargs [dict, optional] keyword arguments passed to *Phase*.

Returns

phase [*Phase*] *Phase* instance.

madmom.audio.stft.**STFT**
alias of *madmom.audio.stft.ShortTimeFourierTransform*

class madmom.audio.stft.**ShortTimeFourierTransformProcessor**(*window=<function hanning>, fft_size=None, circular_shift=False, include_nyquist=False, **kwargs*)

ShortTimeFourierTransformProcessor class.

Parameters

window [numpy ufunc, optional] Window function.

fft_size [int, optional] FFT size (should be a power of 2); if 'None', it is determined by the size of the frames; if is greater than the frame size, the frames are zero-padded accordingly.

circular_shift [bool, optional] Circular shift the individual frames before performing the FFT; needed for correct phase.

include_nyquist [bool, optional] Include the Nyquist frequency bin (sample rate / 2).

Examples

Create a *ShortTimeFourierTransformProcessor* and call it with either a file name or a the output of a (Framed-)SignalProcessor to obtain a *ShortTimeFourierTransform* instance.

```
>>> proc = ShortTimeFourierTransformProcessor()
>>> stft = proc('tests/data/audio/sample.wav')
>>> stft
ShortTimeFourierTransform([[-3.15249+0.j     ,  2.62216-3.02425j, ...,
                             -0.03634-0.00005j,  0.0367 +0.00029j],
                            [-4.28429+0.j     ,  2.02009+2.01264j, ...,
                             -0.01981-0.00933j, -0.00536+0.02162j],
                            ...,
                            [-4.92274+0.j     ,  4.09839-9.42525j, ...,
                              0.0055 -0.00257j,  0.00137+0.00577j],
                            [-9.22709+0.j     ,  8.76929+4.0005j , ...,
                              0.00981-0.00014j, -0.00984+0.00006j]],
                           dtype=complex64)
```

process(*data, **kwargs*)

Perform FFT on a framed signal and return the STFT.

Parameters

data [numpy array] Data to be processed.

kwargs [dict, optional] Keyword arguments passed to *ShortTimeFourierTransform*.

Returns

stft [*ShortTimeFourierTransform*] *ShortTimeFourierTransform* instance.

static add_arguments(*parser, window=None, fft_size=None*)

Add STFT related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser.

window [numpy ufunc, optional] Window function.

fft_size [int, optional] Use this size for FFT (should be a power of 2).

Returns

argparse argument group STFT argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

`madmom.audio.stft.STFTProcessor`
alias of *madmom.audio.stft.ShortTimeFourierTransformProcessor*

class `madmom.audio.stft.Phase`(*stft*, ***kwargs*)
Phase class.

Parameters

stft [*ShortTimeFourierTransform* instance] *ShortTimeFourierTransform* instance.

kwargs [dict, optional] If no *ShortTimeFourierTransform* instance was given, one is instantiated with these additional keyword arguments.

Examples

Create a *Phase* from a *ShortTimeFourierTransform* (or anything it can be instantiated from:

```
>>> stft = ShortTimeFourierTransform('tests/data/audio/sample.wav')
>>> phase = Phase(stft)
>>> phase
Phase([[ 3.14159, -0.85649, ..., -3.14016,  0.00779],
       [ 3.14159,  0.78355, ..., -2.70136,  1.81393],
       ...,
       [ 3.14159, -1.16063, ..., -0.4373 ,  1.33774],
       [ 3.14159,  0.42799, ..., -0.0142 ,  3.13592]], dtype=float32)
```

bin_frequencies
Bin frequencies.

local_group_delay(***kwargs*)
Returns the local group delay of the phase.

Parameters

kwargs [dict, optional] Keyword arguments passed to *LocalGroupDelay*.

Returns

lgd [*LocalGroupDelay* instance] *LocalGroupDelay* instance.

lgd(***kwargs*)
Returns the local group delay of the phase.

Parameters

kwargs [dict, optional] Keyword arguments passed to *LocalGroupDelay*.

Returns

lgd [*LocalGroupDelay* instance] *LocalGroupDelay* instance.

class madmom.audio.stft.**LocalGroupDelay**(*phase*, ***kwargs*)
Local Group Delay class.

Parameters

stft [*Phase* instance] *Phase* instance.

kwargs [dict, optional] If no *Phase* instance was given, one is instantiated with these additional keyword arguments.

Examples

Create a *LocalGroupDelay* from a *ShortTimeFourierTransform* (or anything it can be instantiated from:

```
>>> stft = ShortTimeFourierTransform('tests/data/audio/sample.wav')
>>> lgd = LocalGroupDelay(stft)
>>> lgd
LocalGroupDelay([[-2.2851 , -2.25605, ...,  3.13525,  0. ],
                 [ 2.35804,  2.53786, ...,  1.76788,  0. ],
                 ...,
                 [-1.98..., -2.93039, ..., -1.77505,  0. ],
                 [ 2.7136 ,  2.60925, ...,  3.13318,  0. ]])
```

bin_frequencies
Bin frequencies.

madmom.audio.stft.**LGD**
alias of *madmom.audio.stft.LocalGroupDelay*

7.2.5 madmom.audio.spectrogram

This module contains spectrogram related functionality.

madmom.audio.spectrogram.**spec**(*stft*)
Computes the magnitudes of the complex Short Time Fourier Transform of a signal.

Parameters

stft [numpy array] Complex STFT of a signal.

Returns

spec [numpy array] Magnitude spectrogram.

class madmom.audio.spectrogram.**Spectrogram**(*stft*, ***kwargs*)
A *Spectrogram* represents the magnitude spectrogram of a *audio.stft.ShortTimeFourierTransform*.

Parameters

stft [*audio.stft.ShortTimeFourierTransform* instance] Short Time Fourier Transform.

kwargs [dict, optional] If no *audio.stft.ShortTimeFourierTransform* instance was given, one is instantiated with these additional keyword arguments.

Examples

Create a *Spectrogram* from a *audio.stft.ShortTimeFourierTransform* (or anything it can be instantiated from:

```
>>> spec = Spectrogram('tests/data/audio/sample.wav')
>>> spec
Spectrogram([[ 3.15249,  4.00272, ...,  0.03634,  0.03671],
             [ 4.28429,  2.85158, ...,  0.0219 ,  0.02227],
             ...,
             [ 4.92274, 10.27775, ...,  0.00607,  0.00593],
             [ 9.22709,  9.6387 , ...,  0.00981,  0.00984]], dtype=float32)
```

num_frames
Number of frames.

num_bins
Number of bins.

bin_frequencies
Bin frequencies.

diff(***kwargs*)
Return the difference of the magnitude spectrogram.

Parameters

kwargs [dict] Keyword arguments passed to *SpectrogramDifference*.

Returns

diff [*SpectrogramDifference* instance] The differences of the magnitude spectrogram.

filter(***kwargs*)
Return a filtered version of the magnitude spectrogram.

Parameters

kwargs [dict] Keyword arguments passed to *FilteredSpectrogram*.

Returns

filt_spec [*FilteredSpectrogram* instance] Filtered version of the magnitude spectrogram.

log(***kwargs*)
Return a logarithmically scaled version of the magnitude spectrogram.

Parameters

kwargs [dict] Keyword arguments passed to *LogarithmicSpectrogram*.

Returns

log_spec [*LogarithmicSpectrogram* instance] Logarithmically scaled version of the magnitude spectrogram.

class madmom.audio.spectrogram.**SpectrogramProcessor**(***kwargs*)
SpectrogramProcessor class.

process(*data*, ***kwargs*)
Create a Spectrogram from the given data.

Parameters

data [numpy array] Data to be processed.

kwargs [dict] Keyword arguments passed to `Spectrogram`.

Returns

spec [`Spectrogram` instance] Spectrogram.

class madmom.audio.spectrogram.**FilteredSpectrogram**(*spectrogram, filterbank=<class 'madmom.audio.filters.LogarithmicFilterbank'>, num_bands=12, fmin=30.0, fmax=17000.0, fref=440.0, norm_filters=True, unique_filters=True, **kwargs*)

FilteredSpectrogram class.

Parameters

spectrogram [`Spectrogram` instance] Spectrogram.

filterbank [`audio.filters.Filterbank`, optional] Filterbank class or instance; if a class is given (rather than an instance), one will be created with the given type and parameters.

num_bands [int, optional] Number of filter bands (per octave, depending on the type of the *filterbank*).

fmin [float, optional] Minimum frequency of the filterbank [Hz].

fmax [float, optional] Maximum frequency of the filterbank [Hz].

fref [float, optional] Tuning frequency of the filterbank [Hz].

norm_filters [bool, optional] Normalize the filter bands of the filterbank to area 1.

unique_filters [bool, optional] Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

kwargs [dict, optional] If no `Spectrogram` instance was given, one is instantiated with these additional keyword arguments.

Examples

Create a `FilteredSpectrogram` from a `Spectrogram` (or anything it can be instantiated from. Per default a `madmom.audio.filters.LogarithmicFilterbank` with 12 bands per octave is used.

```
>>> spec = FilteredSpectrogram('tests/data/audio/sample.wav')
>>> spec
FilteredSpectrogram([[ 5.66156, 6.30141, ..., 0.05426, 0.06461],
                     [ 8.44266, 8.69582, ..., 0.07703, 0.0902 ],
                     ...,
                     [10.04626, 1.12018, ..., 0.0487 , 0.04282],
                     [ 8.60186, 6.81195, ..., 0.03721, 0.03371]],
                    dtype=float32)
```

The resulting spectrogram has fewer frequency bins, with the centers of the bins aligned logarithmically (lower frequency bins still have a linear spacing due to the coarse resolution of the DFT at low frequencies):

```
>>> spec.shape
(281, 81)
>>> spec.num_bins
81
>>> spec.bin_frequencies
array([    43.06641,    64.59961,    86.13281,   107.66602,
         129.19922,   150.73242,   172.26562,   193.79883, ...,
       10551.26953, 11175.73242, 11843.26172, 12553.85742,
       13285.98633, 14082.71484, 14922.50977, 15805.37109])
```

The filterbank used to filter the spectrogram is saved as an attribute:

```
>>> spec.filterbank
LogarithmicFilterbank([[0., 0., ..., 0., 0.],
                       [0., 0., ..., 0., 0.],
                       ...,
                       [0., 0., ..., 0., 0.],
                       [0., 0., ..., 0., 0.]], dtype=float32)
>>> spec.filterbank.num_bands
81
```

The filterbank can be chosen at instantiation time:

```
>>> from madmom.audio.filters import MelFilterbank
>>> spec = FilteredSpectrogram('tests/data/audio/sample.wav',
↪filterbank=MelFilterbank, num_bands=40)
>>> type(spec.filterbank)
<class 'madmom.audio.filters.MelFilterbank'>
>>> spec.shape
(281, 40)
```

bin_frequencies
Bin frequencies.

class madmom.audio.spectrogram.**FilteredSpectrogramProcessor**(*filterbank=<class 'madmom.audio.filters.LogarithmicFilterbank'>, num_bands=12, fmin=30.0, fmax=17000.0, fref=440.0, norm_filters=True, unique_filters=True, **kwargs*)

FilteredSpectrogramProcessor class.

Parameters

filterbank [*audio.filters.Filterbank*] Filterbank used to filter a spectrogram.

num_bands [int] Number of bands (per octave).

fmin [float, optional] Minimum frequency of the filterbank [Hz].

fmax [float, optional] Maximum frequency of the filterbank [Hz].

fref [float, optional] Tuning frequency of the filterbank [Hz].

norm_filters [bool, optional] Normalize the filter of the filterbank to area 1.

unique_filters [bool, optional] Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

process (*data*, ***kwargs*)
Create a FilteredSpectrogram from the given data.

Parameters

data [numpy array] Data to be processed.

kwargs [dict] Keyword arguments passed to *FilteredSpectrogram*.

Returns

filt_spec [*FilteredSpectrogram* instance] Filtered spectrogram.

class madmom.audio.spectrogram.**LogarithmicSpectrogram** (*spectrogram*, *log=<ufunc 'log10'>*, *mul=1.0*, *add=1.0*, ***kwargs*)

LogarithmicSpectrogram class.

Parameters

spectrogram [*Spectrogram* instance] Spectrogram.

log [numpy ufunc, optional] Logarithmic scaling function to apply.

mul [float, optional] Multiply the magnitude spectrogram with this factor before taking the logarithm.

add [float, optional] Add this value before taking the logarithm of the magnitudes.

kwargs [dict, optional] If no *Spectrogram* instance was given, one is instantiated with these additional keyword arguments.

Examples

Create a *LogarithmicSpectrogram* from a *Spectrogram* (or anything it can be instantiated from. Per default *np.log10* is used as the scaling function and a value of 1 is added to avoid negative values.

```
>>> spec = LogarithmicSpectrogram('tests/data/audio/sample.wav')
>>> spec
LogarithmicSpectrogram([[...]], dtype=float32)
>>> spec.min()
LogarithmicSpectrogram(0., dtype=float32)
```

filterbank
Filterbank.

bin_frequencies
Bin frequencies.

class madmom.audio.spectrogram.**LogarithmicSpectrogramProcessor** (*log=<ufunc 'log10'>*, *mul=1.0*, *add=1.0*, ***kwargs*)

Logarithmic Spectrogram Processor class.

Parameters

log [numpy ufunc, optional] Loagrithmic scaling function to apply.

mul [float, optional] Multiply the magnitude spectrogram with this factor before taking the logarithm.

add [float, optional] Add this value before taking the logarithm of the magnitudes.

process(*data*, ***kwargs*)
Perform logarithmic scaling of a spectrogram.

Parameters

data [numpy array] Data to be processed.

kwargs [dict] Keyword arguments passed to *LogarithmicSpectrogram*.

Returns

log_spec [*LogarithmicSpectrogram* instance] Logarithmically scaled spectrogram.

static add_arguments(*parser*, *log=None*, *mul=None*, *add=None*)
Add spectrogram scaling related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

log [bool, optional] Take the logarithm of the spectrogram.

mul [float, optional] Multiply the magnitude spectrogram with this factor before taking the logarithm.

add [float, optional] Add this value before taking the logarithm of the magnitudes.

Returns

argparse argument group Spectrogram scaling argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

class madmom.audio.spectrogram.**LogarithmicFilteredSpectrogram**(*spectrogram*, ***kwargs*)
LogarithmicFilteredSpectrogram class.

Parameters

spectrogram [*FilteredSpectrogram* instance] Filtered spectrogram.

kwargs [dict, optional] If no *FilteredSpectrogram* instance was given, one is instantiated with these additional keyword arguments and logarithmically scaled afterwards, i.e. passed to *LogarithmicSpectrogram*.

See also:

FilteredSpectrogram, *LogarithmicSpectrogram*

Notes

For the filtering and scaling parameters, please refer to *FilteredSpectrogram* and *LogarithmicSpectrogram*.

Examples

Create a *LogarithmicFilteredSpectrogram* from a *Spectrogram* (or anything it can be instantiated from. This is mainly a convenience class which first filters the spectrogram and then scales it logarithmically.

```
>>> spec = LogarithmicFilteredSpectrogram('tests/data/audio/sample.wav')
>>> spec
LogarithmicFilteredSpectrogram([[0.82358, 0.86341, ..., 0.02295, 0.02719],
                                [0.97509, 0.98658, ..., 0.03223, 0.0375 ],
                                ...,
                                [1.04322, 0.32637, ..., 0.02065, 0.01821],
                                [0.98236, 0.89276, ..., 0.01587, 0.0144 ]],
                               dtype=float32)
>>> spec.shape
(281, 81)
>>> spec.filterbank
LogarithmicFilterbank([[...]], dtype=float32)
>>> spec.min()
LogarithmicFilteredSpectrogram(0.00831, dtype=float32)
```

filterbank
Filterbank.

bin_frequencies
Bin frequencies.

class madmom.audio.spectrogram.**LogarithmicFilteredSpectrogramProcessor**(*filterbank=<class 'madmom.audio.filters.Logarithmi, num_bands=12, fmin=30.0, fmax=17000.0, fref=440.0, norm_filters=True, unique_filters=True, mul=1.0, add=1.0, **kwargs*)

Logarithmic Filtered Spectrogram Processor class.

Parameters

filterbank [*audio.filters.Filterbank*] Filterbank used to filter a spectrogram.

num_bands [int] Number of bands (per octave).

fmin [float, optional] Minimum frequency of the filterbank [Hz].

fmax [float, optional] Maximum frequency of the filterbank [Hz].

fref [float, optional] Tuning frequency of the filterbank [Hz].

norm_filters [bool, optional] Normalize the filter of the filterbank to area 1.

unique_filters [bool, optional] Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

mul [float, optional] Multiply the magnitude spectrogram with this factor before taking the logarithm.

add [float, optional] Add this value before taking the logarithm of the magnitudes.

process (*data*, ***kwargs*)

Perform filtering and logarithmic scaling of a spectrogram.

Parameters

data [numpy array] Data to be processed.

kwargs [dict] Keyword arguments passed to *LogarithmicFilteredSpectrogram*.

Returns

log_filt_spec [*LogarithmicFilteredSpectrogram* instance] Logarithmically scaled filtered spectrogram.

class madmom.audio.spectrogram.**SpectrogramDifference** (*spectrogram*, *diff_ratio=0.5*, *diff_frames=None*, *diff_max_bins=None*, *positive_diffs=False*, *keep_dims=True*, ***kwargs*)

SpectrogramDifference class.

Parameters

spectrogram [*Spectrogram* instance] Spectrogram.

diff_ratio [float, optional] Calculate the difference to the frame at which the window used for the STFT yields this ratio of the maximum height.

diff_frames [int, optional] Calculate the difference to the *diff_frames*-th previous frame (if set, this overrides the value calculated from the *diff_ratio*)

diff_max_bins [int, optional] Apply a maximum filter with this width (in bins in frequency dimension) to the spectrogram the difference is calculated to.

positive_diffs [bool, optional] Keep only the positive differences, i.e. set all diff values < 0 to 0.

keep_dims [bool, optional] Indicate if the dimensions (i.e. shape) of the spectrogram should be kept.

kwargs [dict, optional] If no *Spectrogram* instance was given, one is instantiated with these additional keyword arguments.

Notes

The first *diff_frames* frames will have a value of 0.

If *keep_dims* is 'True' the returned difference has the same shape as the spectrogram. This is needed if the diffs should be stacked on top of it. If set to 'False', the length will be *diff_frames* frames shorter (mostly used by the SpectrogramDifferenceProcessor which first buffers that many frames.

The SuperFlux algorithm *[1]* uses a maximum filtered spectrogram with 3 *diff_max_bins* together with a 24 band logarithmic filterbank to calculate the difference spectrogram with a *diff_ratio* of 0.5.

The effect of this maximum filter applied to the spectrogram is that the magnitudes are "widened" in frequency direction, i.e. the following difference calculation is less sensitive against frequency fluctuations. This effect is exploited to suppress false positive energy fragments originating from vibrato.

References

[1]

Examples

To obtain the SuperFlux feature as described above first create a filtered and logarithmically spaced spectrogram:

```
>>> spec = LogarithmicFilteredSpectrogram('tests/data/audio/sample.wav',
↪                                          num_bands=24, fps=200)
>>> spec
LogarithmicFilteredSpectrogram([[0.82358, 0.86341, ..., 0.02809, 0.02672],
                                [0.92514, 0.93211, ..., 0.03607, 0.0317 ],
                                ...,
                                [1.03826, 0.767  , ..., 0.01814, 0.01138],
                                [0.98236, 0.89276, ..., 0.01669, 0.00919]],
                                dtype=float32)
>>> spec.shape
(561, 140)
```

Then use the temporal first order difference and apply a maximum filter with 3 bands, keeping only the positive differences (i.e. rise in energy):

```
>>> superflux = SpectrogramDifference(spec, diff_max_bins=3,
↪                      positive_diffs=True)
>>> superflux
SpectrogramDifference([[0.     , 0. , ...,  0. ,  0. ],
                       [0.     , 0. , ...,  0. ,  0. ],
                       ...,
                       [0.01941, 0. , ...,  0. ,  0. ],
                       [0.     , 0. , ...,  0. ,  0. ]], dtype=float32)
```

bin_frequencies
Bin frequencies.

positive_diff()
Positive diff.

class madmom.audio.spectrogram.**SpectrogramDifferenceProcessor**(*diff_ratio=0.5, diff_frames=None, diff_max_bins=None, positive_diffs=False, stack_diffs=None, **kwargs*)

Difference Spectrogram Processor class.

Parameters

diff_ratio [float, optional] Calculate the difference to the frame at which the window used for the STFT yields this ratio of the maximum height.

diff_frames [int, optional] Calculate the difference to the *diff_frames*-th previous frame (if set, this overrides the value calculated from the *diff_ratio*)

diff_max_bins [int, optional] Apply a maximum filter with this width (in bins in frequency dimension) to the spectrogram the difference is calculated to.

positive_diffs [bool, optional] Keep only the positive differences, i.e. set all diff values < 0 to 0.

stack_diffs [numpy stacking function, optional] If 'None', only the differences are returned. If set, the diffs are stacked with the underlying spectrogram data according to the *stack* function:

- `np.vstack` the differences and spectrogram are stacked vertically, i.e. in time direction,
- `np.hstack` the differences and spectrogram are stacked horizontally, i.e. in frequency direction,
- `np.dstack` the differences and spectrogram are stacked in depth, i.e. return them as a 3D representation with depth as the third dimension.

process (*data*, *reset=True*, ***kwargs*)
Perform a temporal difference calculation on the given data.

Parameters

data [numpy array] Data to be processed.

reset [bool, optional] Reset the spectrogram buffer before computing the difference.

kwargs [dict] Keyword arguments passed to *SpectrogramDifference*.

Returns

diff [*SpectrogramDifference* instance] Spectrogram difference.

Notes

If *reset* is 'True', the first *diff_frames* differences will be 0.

reset ()
Reset the SpectrogramDifferenceProcessor.

static add_arguments (*parser*, *diff=None*, *diff_ratio=None*, *diff_frames=None*, *diff_max_bins=None*, *positive_diffs=None*)
Add spectrogram difference related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

diff [bool, optional] Take the difference of the spectrogram.

diff_ratio [float, optional] Calculate the difference to the frame at which the window used for the STFT yields this ratio of the maximum height.

diff_frames [int, optional] Calculate the difference to the *diff_frames*-th previous frame (if set, this overrides the value calculated from the *diff_ratio*)

diff_max_bins [int, optional] Apply a maximum filter with this width (in bins in frequency dimension) to the spectrogram the difference is calculated to.

positive_diffs [bool, optional] Keep only the positive differences, i.e. set all diff values < 0 to 0.

Returns

argparse argument group Spectrogram difference argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

Only the *diff_frames* parameter behaves differently, it is included if either the *diff_ratio* is set or a value != 'None' is given.

class madmom.audio.spectrogram.**SuperFluxProcessor**(***kwargs*)

Spectrogram processor which sets the default values suitable for the SuperFlux algorithm.

class madmom.audio.spectrogram.**MultiBandSpectrogram**(*spectrogram*, *crossover_frequencies*, *fmin=30.0*, *fmax=17000.0*, *norm_filters=True*, *unique_filters=True*, ***kwargs*)

MultiBandSpectrogram class.

Parameters

spectrogram [*Spectrogram* instance] Spectrogram.

crossover_frequencies [list or numpy array] List of crossover frequencies at which the *spectrogram* is split into multiple bands.

fmin [float, optional] Minimum frequency of the filterbank [Hz].

fmax [float, optional] Maximum frequency of the filterbank [Hz].

norm_filters [bool, optional] Normalize the filter bands of the filterbank to area 1.

unique_filters [bool, optional] Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

kwargs [dict, optional] If no *Spectrogram* instance was given, one is instantiated with these additional keyword arguments.

Notes

The MultiBandSpectrogram is implemented as a *Spectrogram* which uses a *audio.filters.RectangularFilterbank* to combine multiple frequency bins.

class madmom.audio.spectrogram.**MultiBandSpectrogramProcessor**(*crossover_frequencies*, *fmin=30.0*, *fmax=17000.0*, *norm_filters=True*, *unique_filters=True*, ***kwargs*)

Spectrogram processor which combines the spectrogram magnitudes into multiple bands.

Parameters

crossover_frequencies [list or numpy array] List of crossover frequencies at which a spectrogram is split into the individual bands.

fmin [float, optional] Minimum frequency of the filterbank [Hz].

fmax [float, optional] Maximum frequency of the filterbank [Hz].

norm_filters [bool, optional] Normalize the filter bands of the filterbank to area 1.

unique_filters [bool, optional] Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

process(*data*, ***kwargs*)

Return the a multi-band representation of the given data.

Parameters

data [numpy array] Data to be processed.

kwargs [dict] Keyword arguments passed to `MultiBandSpectrogram`.

Returns

multi_band_spec [`MultiBandSpectrogram` instance] Spectrogram split into multiple bands.

class madmom.audio.spectrogram.**SemitoneBandpassSpectrogram**(*signal*, *fps=50.0*, *fmin=27.5*, *fmax=4200.0*)

Construct a semitone spectrogram by using a time domain filterbank of bandpass filters as described in *[1]*.

Parameters

signal [Signal] Signal instance.

fps [float, optional] Frame rate of the spectrogram [Hz].

fmin [float, optional] Lowest frequency of the spectrogram [Hz].

fmax [float, optional] Highest frequency of the spectrogram [Hz].

References

[1]

7.2.6 madmom.audio.chroma

This module contains chroma related functionality.

class madmom.audio.chroma.**DeepChromaProcessor**(*fmin=65*, *fmax=2100*, *unique_filters=True*, *models=None*, ***kwargs*)

Compute chroma vectors from an audio file using a deep neural network that focuses on harmonically relevant spectral content.

Parameters

fmin [int, optional] Minimum frequency of the filterbank [Hz].

fmax [float, optional] Maximum frequency of the filterbank [Hz].

unique_filters [bool, optional] Indicate if the filterbank should contain only unique filters, i.e. remove duplicate filters resulting from insufficient resolution at low frequencies.

models [list of filenames, optional] List of model filenames.

Notes

Provided model files must be compatible with the processing pipeline and the values of *fmin*, *fmax*, and *unique_filters*. The general use case for the *models* parameter is to use a specific model instead of an ensemble of all models.

The models shipped with madmom differ slightly from those presented in the paper (less hidden units, narrower frequency band for spectrogram), but achieve similar results.

References

[1]

Examples

Extract a chroma vector using the deep chroma extractor:

```
>>> dcp = DeepChromaProcessor()
>>> chroma = dcp('tests/data/audio/sample2.wav')
>>> chroma
array([[0.01317, 0.00721, ..., 0.00546, 0.00943],
       [0.36809, 0.01314, ..., 0.02213, 0.01838],
       ...,
       [0.1534 , 0.06475, ..., 0.00896, 0.05789],
       [0.17513, 0.0729 , ..., 0.00945, 0.06913]], dtype=float32)
>>> chroma.shape
(41, 12)
```

class madmom.audio.chroma.**CLPChroma**(*data*, *fps=50*, *fmin=27.5*, *fmax=4200.0*, *compression_factor=100*, *norm=True*, *threshold=0.001*, ***kwargs*)

Compressed Log Pitch (CLP) chroma as proposed in [1] and [2].

Parameters

data [str, Signal, or SemitoneBandpassSpectrogram] Input data.

fps [int, optional] Desired frame rate of the signal [Hz].

fmin [float, optional] Lowest frequency of the spectrogram [Hz].

fmax [float, optional] Highest frequency of the spectrogram [Hz].

compression_factor [float, optional] Factor for compression of the energy.

norm [bool, optional] Normalize the energy of each frame to one (divide by the L2 norm).

threshold [float, optional] If the energy of a frame is below a threshold, the energy is equally distributed among all chroma bins.

Notes

The resulting chromagrams differ slightly from those obtained by the MATLAB chroma toolbox [2] because of different resampling and filter methods.

References

[1], [2]

class madmom.audio.chroma.**CLPChromaProcessor**(*fps=50*, *fmin=27.5*, *fmax=4200.0*, *compression_factor=100*, *norm=True*, *threshold=0.001*, ***kwargs*)

Compressed Log Pitch (CLP) Chroma Processor.

Parameters

fps [int, optional] Desired frame rate of the signal [Hz].

fmin [float, optional] Lowest frequency of the spectrogram [Hz].

fmax [float, optional] Highest frequency of the spectrogram [Hz].

compression_factor [float, optional] Factor for compression of the energy.

norm [bool, optional] Normalize the energy of each frame to one (divide by the L2 norm).

threshold [float, optional] If the energy of a frame is below a threshold, the energy is equally distributed among all chroma bins.

process(*data*, ***kwargs*)

Create a CLPChroma from the given data.

Parameters

data [Signal instance or filename] Data to be processed.

Returns

clp [*CLPChroma* instance] CLPChroma.

CHAPTER 8

madmom.features

This package includes high-level features. Your definition of "high" may vary, but we define high-level features as the ones you want to evaluate (e.g. onsets, beats, etc.). All lower-level features can be found the *madmom.audio* package.

8.1 Notes

All features should be implemented as classes which inherit from Processor (or provide a XYZProcessor(Processor) variant). This way, multiple Processor objects can be chained/combined to achieve the wanted functionality.

class madmom.features.**Activations**(*data*, *fps=None*, *sep=None*, *dtype=<type 'numpy.float32'>*)

The Activations class extends a numpy ndarray with a frame rate (fps) attribute.

Parameters

data [str, file handle or numpy array] Either file name/handle to read the data from or array.

fps [float, optional] Frames per second (must be set if *data* is given as an array).

sep [str, optional] Separator between activation values (if read from file).

dtype [numpy dtype] Data-type the activations are stored/saved/kept.

Notes

If a filename or file handle is given, an undefined or empty separator means that the file should be treated as a numpy binary file. Only binary files can store the frame rate of the activations. Text files should not be used for anything else but manual inspection or I/O with other programs.

Attributes

fps [float] Frames per second.

classmethod load(*infile*, *fps=None*, *sep=None*)

Load the activations from a file.

Parameters

infile [str or file handle] Input file name or file handle.

fps [float, optional] Frames per second; if set, it overwrites the saved frame rate.

sep [str, optional] Separator between activation values.

Returns

:class:‘Activations‘ instance `Activations` instance.

Notes

An undefined or empty separator means that the file should be treated as a numpy binary file. Only binary files can store the frame rate of the activations. Text files should not be used for anything else but manual inspection or I/O with other programs.

save (*outfile*, *sep=None*, *fmt='%.5f'*)

Save the activations to a file.

Parameters

outfile [str or file handle] Output file name or file handle.

sep [str, optional] Separator between activation values if saved as text file.

fmt [str, optional] Format of the values if saved as text file.

Notes

An undefined or empty separator means that the file should be treated as a numpy binary file. Only binary files can store the frame rate of the activations. Text files should not be used for anything else but manual inspection or I/O with other programs.

If the activations are a 1D array, its values are interpreted as features of a single time step, i.e. all values are printed in a single line. If you want each value to appear in an individual line, use ‘n’ as a separator.

If the activations are a 2D array, the first axis corresponds to the time dimension, i.e. the features are separated by *sep* and the time steps are printed in separate lines. If you like to swap the dimensions, please use the *T* attribute.

class madmom.features.**ActivationsProcessor** (*mode*, *fps=None*, *sep=None*, ***kwargs*)

ActivationsProcessor processes a file and returns an Activations instance.

Parameters

mode [{‘r’, ‘w’, ‘in’, ‘out’, ‘load’, ‘save’}] Mode of the Processor: read/write.

fps [float, optional] Frame rate of the activations (if set, it overwrites the saved frame rate).

sep [str, optional] Separator between activation values if saved as text file.

Notes

An undefined or empty (“”) separator means that the file should be treated as a numpy binary file. Only binary files can store the frame rate of the activations.

process (*data*, *output=None*, ***kwargs*)
Depending on the mode, either loads the data stored in the given file and returns it as an Activations instance or save the data to the given output.

Parameters

data [str, file handle or numpy array] Data or file to be loaded (if *mode* is 'r') or data to be saved to file (if *mode* is 'w').

output [str or file handle, optional] output file (only in write-mode)

Returns

:class:'Activations' instance *Activations* instance (only in read-mode)

static add_arguments (*parser*)
Add options to save/load activations to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser.

Returns

parser_group [argparse argument group] Input/output argument parser group.

8.2 Submodules

8.2.1 madmom.features.beats

This module contains beat tracking related functionality.

class madmom.features.beats.**RNNBeatProcessor** (*post_processor=<function average_predictions>*, *online=False*, *nn_files=None*, ***kwargs*)
Processor to get a beat activation function from multiple RNNs.

Parameters

post_processor [Processor, optional] Post-processor, default is to average the predictions.

online [bool, optional] Use signal processing parameters and RNN models suitable for online mode.

nn_files [list, optional] List with trained RNN model files. Per default ('None'), an ensemble of networks will be used.

References

[1]

Examples

Create a RNNBeatProcessor and pass a file through the processor. The returned 1d array represents the probability of a beat at each frame, sampled at 100 frames per second.

```
>>> proc = RNNBeatProcessor()
>>> proc
<madmom.features.beats.RNNBeatProcessor object at 0x...>
>>> proc('tests/data/audio/sample.wav')
array([0.00479, 0.00603, 0.00927, 0.01419, ... 0.02725], dtype=float32)
```

For online processing, *online* must be set to 'True'. If processing power is limited, fewer number of RNN models can be defined via *nn_files*. The audio signal is then processed frame by frame.

```
>>> from madmom.models import BEATS_LSTM
>>> proc = RNNBeatProcessor(online=True, nn_files=[BEATS_LSTM[0]])
>>> proc
<madmom.features.beats.RNNBeatProcessor object at 0x...>
>>> proc('tests/data/audio/sample.wav')
array([0.03887, 0.02619, 0.00747, 0.00218, ... 0.04825], dtype=float32)
```

class madmom.features.beats.**MultiModelSelectionProcessor**(*num_ref_predictions*, ***kwargs*)

Processor for selecting the most suitable model (i.e. the predictions thereof) from a multiple models/predictions.

Parameters

num_ref_predictions [int] Number of reference predictions (see below).

Notes

This processor selects the most suitable prediction from multiple models by comparing them to the predictions of a reference model. The one with the smallest mean squared error is chosen.

If *num_ref_predictions* is 0 or None, an averaged prediction is computed from the given predictions and used as reference.

References

[1]

Examples

The MultiModelSelectionProcessor takes a list of model predictions as it's call argument. Thus, *ppost_processor* of *RNNBeatProcessor* hast to be set to 'None' in order to get the predictions of all models.

```
>>> proc = RNNBeatProcessor(post_processor=None)
>>> proc
<madmom.features.beats.RNNBeatProcessor object at 0x...>
```

When passing a file through the processor, a list with predictions, one for each model tested, is returned.

```
>>> predictions = proc('tests/data/audio/sample.wav')
>>> predictions
[array([0.00535, 0.00774, ..., 0.02343, 0.04931], dtype=float32),
 array([0.0022 , 0.00282, ..., 0.00825, 0.0152 ], dtype=float32),
 ...,
 array([0.005  , 0.0052 , ..., 0.00472, 0.01524], dtype=float32),
 array([0.00319, 0.0044 , ..., 0.0081 , 0.01498], dtype=float32)]
```

We can feed these predictions to the MultiModelSelectionProcessor. Since we do not have a dedicated reference prediction (which had to be the first element of the list and *num_ref_predictions* set to 1), we simply set *num_ref_predictions* to 'None'. MultiModelSelectionProcessor averages all predictions to obtain a reference prediction it compares all others to.

```
>>> mm_proc = MultiModelSelectionProcessor(num_ref_predictions=None)
>>> mm_proc(predictions)
array([0.00759, 0.00901, ..., 0.00843, 0.01834], dtype=float32)
```

process (*predictions*, ***kwargs*)

Selects the most appropriate predictions form the list of predictions.

Parameters

predictions [list] Predictions (beat activation functions) of multiple models.

Returns

numpy array Most suitable prediction.

Notes

The reference beat activation function must be the first one in the list of given predictions.

madmom.features.beats.**detect_beats** (*activations*, *interval*, *look_aside=0.2*)

Detects the beats in the given activation function as in *[1]*.

Parameters

activations [numpy array] Beat activations.

interval [int] Look for the next beat each *interval* frames.

look_aside [float] Look this fraction of the *interval* to each side to detect the beats.

Returns

numpy array Beat positions [frames].

Notes

A Hamming window of 2 * *look_aside* * *interval* is applied around the position where the beat is expected to prefer beats closer to the centre.

References

[1]

class madmom.features.beats.**BeatTrackingProcessor** (*look_aside=0.2*, *look_ahead=10.0*, *fps=None*, *tempo_estimator=None*, ***kwargs*)

Track the beats according to previously determined (local) tempo by iteratively aligning them around the estimated position *[1]*.

Parameters

look_aside [float, optional] Look this fraction of the estimated beat interval to each side of the assumed next beat position to look for the most likely position of the next beat.

look_ahead [float, optional] Look *look_ahead* seconds in both directions to determine the local tempo and align the beats accordingly.

tempo_estimator [TempoEstimationProcessor, optional] Use this processor to estimate the (local) tempo. If 'None' a default tempo estimator will be created and used.

fps [float, optional] Frames per second.

kwargs [dict, optional] Keyword arguments passed to `madmom.features.tempo.TempoEstimationProcessor` if no *tempo_estimator* was given.

Notes

If *look_ahead* is not set, a constant tempo throughout the whole piece is assumed. If *look_ahead* is set, the local tempo (in a range +/- *look_ahead* seconds around the actual position) is estimated and then the next beat is tracked accordingly. This procedure is repeated from the new position to the end of the piece.

Instead of the auto-correlation based method for tempo estimation proposed in *[1]*, it uses a comb filter based method *[2]* per default. The behaviour can be controlled with the *tempo_method* parameter.

References

[1], *[2]*

Examples

Create a BeatTrackingProcessor. The returned array represents the positions of the beats in seconds, thus the expected sampling rate has to be given.

```
>>> proc = BeatTrackingProcessor(fps=100)
>>> proc
<madmom.features.beats.BeatTrackingProcessor object at 0x...>
```

Call this BeatTrackingProcessor with the beat activation function returned by RNNBeatProcessor to obtain the beat positions.

```
>>> act = RNNBeatProcessor()('tests/data/audio/sample.wav')
>>> proc(act)
array([0.11, 0.45, 0.79, 1.13, 1.47, 1.81, 2.15, 2.49])
```

process (*activations*, ***kwargs*)
Detect the beats in the given activation function.

Parameters

activations [numpy array] Beat activation function.

Returns

beats [numpy array] Detected beat positions [seconds].

static add_arguments (*parser*, *look_aside=0.2*, *look_ahead=10.0*)
Add beat tracking related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

look_aside [float, optional] Look this fraction of the estimated beat interval to each side of the assumed next beat position to look for the most likely position of the next beat.

look_ahead [float, optional] Look *look_ahead* seconds in both directions to determine the local tempo and align the beats accordingly.

Returns

parser_group [argparse argument group] Beat tracking argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

class madmom.features.beats.**BeatDetectionProcessor**(*look_aside=0.2*, *fps=None*, ***kwargs*)

Class for detecting beats according to the previously determined global tempo by iteratively aligning them around the estimated position *[1]*.

Parameters

look_aside [float] Look this fraction of the estimated beat interval to each side of the assumed next beat position to look for the most likely position of the next beat.

fps [float, optional] Frames per second.

See also:

BeatTrackingProcessor

Notes

A constant tempo throughout the whole piece is assumed.

Instead of the auto-correlation based method for tempo estimation proposed in *[1]*, it uses a comb filter based method *[2]* per default. The behaviour can be controlled with the *tempo_method* parameter.

References

[1], *[2]*

Examples

Create a BeatDetectionProcessor. The returned array represents the positions of the beats in seconds, thus the expected sampling rate has to be given.

```
>>> proc = BeatDetectionProcessor(fps=100)
>>> proc
<madmom.features.beats.BeatDetectionProcessor object at 0x...>
```

Call this BeatDetectionProcessor with the beat activation function returned by RNNBeatProcessor to obtain the beat positions.

```
>>> act = RNNBeatProcessor()('tests/data/audio/sample.wav')
>>> proc(act)
array([0.11, 0.45, 0.79, 1.13, 1.47, 1.81, 2.15, 2.49])
```

class madmom.features.beats.**CRFBeatDetectionProcessor**(*interval_sigma=0.18, use_factors=False, num_intervals=5, factors=array([0.5, 0.67, 1., 1.5, 2.]), **kwargs*)

Conditional Random Field Beat Detection.

Tracks the beats according to the previously determined global tempo using a conditional random field (CRF) model.

Parameters

interval_sigma [float, optional] Allowed deviation from the dominant beat interval per beat.

use_factors [bool, optional] Use dominant interval multiplied by factors instead of intervals estimated by tempo estimator.

num_intervals [int, optional] Maximum number of estimated intervals to try.

factors [list or numpy array, optional] Factors of the dominant interval to try.

References

[1]

Examples

Create a CRFBeatDetectionProcessor. The returned array represents the positions of the beats in seconds, thus the expected sampling rate has to be given.

```
>>> proc = CRFBeatDetectionProcessor(fps=100)
>>> proc
<madmom.features.beats.CRFBeatDetectionProcessor object at 0x...>
```

Call this BeatDetectionProcessor with the beat activation function returned by RNNBeatProcessor to obtain the beat positions.

```
>>> act = RNNBeatProcessor()('tests/data/audio/sample.wav')
>>> proc(act)
array([0.09, 0.79, 1.49])
```

process(*activations, **kwargs*)

Detect the beats in the given activation function.

Parameters

activations [numpy array] Beat activation function.

Returns

numpy array Detected beat positions [seconds].

static add_arguments(*parser, interval_sigma=0.18, use_factors=False, num_intervals=5, factors=array([0.5, 0.67, 1., 1.5, 2.])*)

Add CRFBeatDetection related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

interval_sigma [float, optional] allowed deviation from the dominant beat interval per beat

use_factors [bool, optional] use dominant interval multiplied by factors instead of intervals estimated by tempo estimator

num_intervals [int, optional] max number of estimated intervals to try

factors [list or numpy array, optional] factors of the dominant interval to try

Returns

parser_group [argparse argument group] CRF beat tracking argument parser group.

class madmom.features.beats.**DBNBeatTrackingProcessor**(*min_bpm=55.0*, *max_bpm=215.0*, *num_tempi=None*, *transition_lambda=100*, *observation_lambda=16*, *correct=True*, *threshold=0*, *fps=None*, *online=False*, ***kwargs*)

Beat tracking with RNNs and a dynamic Bayesian network (DBN) approximated by a Hidden Markov Model (HMM).

Parameters

min_bpm [float, optional] Minimum tempo used for beat tracking [bpm].

max_bpm [float, optional] Maximum tempo used for beat tracking [bpm].

num_tempi [int, optional] Number of tempi to model; if set, limit the number of tempi and use a log spacing, otherwise a linear spacing.

transition_lambda [float, optional] Lambda for the exponential tempo change distribution (higher values prefer a constant tempo from one beat to the next one).

observation_lambda [int, optional] Split one beat period into *observation_lambda* parts, the first representing beat states and the remaining non-beat states.

threshold [float, optional] Threshold the observations before Viterbi decoding.

correct [bool, optional] Correct the beats (i.e. align them to the nearest peak of the beat activation function).

fps [float, optional] Frames per second.

online [bool, optional] Use the forward algorithm (instead of Viterbi) to decode the beats.

Notes

Instead of the originally proposed state space and transition model for the DBN *[1]*, the more efficient version proposed in *[2]* is used.

References

[1], *[2]*

Examples

Create a DBNBeatTrackingProcessor. The returned array represents the positions of the beats in seconds, thus the expected sampling rate has to be given.

```
>>> proc = DBNBeatTrackingProcessor(fps=100)
>>> proc
<madmom.features.beats.DBNBeatTrackingProcessor object at 0x...>
```

Call this DBNBeatTrackingProcessor with the beat activation function returned by RNNBeatProcessor to obtain the beat positions.

```
>>> act = RNNBeatProcessor()('tests/data/audio/sample.wav')
>>> proc(act)
array([0.1 , 0.45, 0.8 , 1.12, 1.48, 1.8 , 2.15, 2.49])
```

reset()
Reset the DBNBeatTrackingProcessor.

process_offline(*activations*, ***kwargs*)
Detect the beats in the given activation function with Viterbi decoding.

Parameters

activations [numpy array] Beat activation function.

Returns

beats [numpy array] Detected beat positions [seconds].

process_online(*activations*, *reset=True*, ***kwargs*)
Detect the beats in the given activation function with the forward algorithm.

Parameters

activations [numpy array] Beat activation for a single frame.

reset [bool, optional] Reset the DBNBeatTrackingProcessor to its initial state before processing.

Returns

beats [numpy array] Detected beat position [seconds].

process_forward(*activations*, *reset=True*, ***kwargs*)
Detect the beats in the given activation function with the forward algorithm.

Parameters

activations [numpy array] Beat activation for a single frame.

reset [bool, optional] Reset the DBNBeatTrackingProcessor to its initial state before processing.

Returns

beats [numpy array] Detected beat position [seconds].

process_viterbi(*activations*, ***kwargs*)
Detect the beats in the given activation function with Viterbi decoding.

Parameters

activations [numpy array] Beat activation function.

Returns

beats [numpy array] Detected beat positions [seconds].

`static add_arguments`(*parser*, *min_bpm=55.0*, *max_bpm=215.0*, *num_tempi=None*, *transition_lambda=100*, *observation_lambda=16*, *threshold=0*, *correct=True*)

Add DBN related arguments to an existing parser object.

Parameters

parser [argparse parser instance] Existing argparse parser object.

min_bpm [float, optional] Minimum tempo used for beat tracking [bpm].

max_bpm [float, optional] Maximum tempo used for beat tracking [bpm].

num_tempi [int, optional] Number of tempi to model; if set, limit the number of tempi and use a log spacing, otherwise a linear spacing.

transition_lambda [float, optional] Lambda for the exponential tempo change distribution (higher values prefer a constant tempo over a tempo change from one beat to the next one).

observation_lambda [float, optional] Split one beat period into *observation_lambda* parts, the first representing beat states and the remaining non-beat states.

threshold [float, optional] Threshold the observations before Viterbi decoding.

correct [bool, optional] Correct the beats (i.e. align them to the nearest peak of the beat activation function).

Returns

parser_group [argparse argument group] DBN beat tracking argument parser group

8.2.2 madmom.features.beats_crf

This module contains the speed crucial Viterbi functionality for the CRFBeatDetector plus some functions computing the distributions and normalisation factors.

References

`madmom.features.beats_crf.best_sequence`(*activations*, *interval*, *interval_sigma*)

Extract the best beat sequence for a piece with the Viterbi algorithm.

Parameters

activations [numpy array] Beat activation function of the piece.

interval [int] Beat interval of the piece.

interval_sigma [float] Allowed deviation from the interval per beat.

Returns

beat_pos [numpy array] Extracted beat positions [frame indices].

log_prob [float] Log probability of the beat sequence.

`madmom.features.beats_crf.initial_distribution`(*num_states*, *interval*)

Compute the initial distribution.

Parameters

num_states [int] Number of states in the model.

interval [int] Beat interval of the piece [frames].

Returns

numpy array Initial distribution of the model.

`madmom.features.beats_crf.normalisation_factors(activations, transition_distribution)`

Compute normalisation factors for model.

Parameters

activations [numpy array] Beat activation function of the piece.

transition_distribution [numpy array] Transition distribution of the model.

Returns

numpy array Normalisation factors for model.

`madmom.features.beats_crf.transition_distribution(interval, interval_sigma)`

Compute the transition distribution between beats.

Parameters

interval [int] Interval of the piece [frames].

interval_sigma [float] Allowed deviation from the interval per beat.

Returns

numpy array Transition distribution between beats.

`madmom.features.beats_crf.viterbi(__Pyx_memviewslice pi, __Pyx_memviewslice transition, __Pyx_memviewslice norm_factor, __Pyx_memviewslice activations, int tau)`

Viterbi algorithm to compute the most likely beat sequence from the given activations and the dominant interval.

Parameters

pi [numpy array] Initial distribution.

transition [numpy array] Transition distribution.

norm_factor [numpy array] Normalisation factors.

activations [numpy array] Beat activations.

tau [int] Dominant interval [frames].

Returns

beat_pos [numpy array] Extracted beat positions [frame indices].

log_prob [float] Log probability of the beat sequence.

8.2.3 madmom.features.beats_hmm

This module contains HMM state spaces, transition and observation models used for beat, downbeat and pattern tracking.

Notes

Please note that (almost) everything within this module is discretised to integer values because of performance reasons.

class madmom.features.beats_hmm.**BeatStateSpace**(*min_interval*, *max_interval*, *num_intervals=None*)

State space for beat tracking with a HMM.

Parameters

min_interval [float] Minimum interval to model.

max_interval [float] Maximum interval to model.

num_intervals [int, optional] Number of intervals to model; if set, limit the number of intervals and use a log spacing instead of the default linear spacing.

References

[1]

Attributes

num_states [int] Number of states.

intervals [numpy array] Modeled intervals.

num_intervals [int] Number of intervals.

state_positions [numpy array] Positions of the states (i.e. 0. . . 1).

state_intervals [numpy array] Intervals of the states (i.e. 1 / tempo).

first_states [numpy array] First state of each interval.

last_states [numpy array] Last state of each interval.

class madmom.features.beats_hmm.**BarStateSpace**(*num_beats*, *min_interval*, *max_interval*, *num_intervals=None*)

State space for bar tracking with a HMM.

Model *num_beat* identical beats with the given arguments in a single state space.

Parameters

num_beats [int] Number of beats to form a bar.

min_interval [float] Minimum beat interval to model.

max_interval [float] Maximum beat interval to model.

num_intervals [int, optional] Number of beat intervals to model; if set, limit the number of intervals and use a log spacing instead of the default linear spacing.

References

[1]

Attributes

num_beats [int] Number of beats.

num_states [int] Number of states.

num_intervals [int] Number of intervals.

state_positions [numpy array] Positions of the states.

state_intervals [numpy array] Intervals of the states.

first_states [list] First states of each beat.

last_states [list] Last states of each beat.

class madmom.features.beats_hmm.**MultiPatternStateSpace**(*state_spaces*)

State space for rhythmic pattern tracking with a HMM.

Model a joint state space with the given *state_spaces* by stacking the individual state spaces.

Parameters

state_spaces [list] List with state spaces to model.

References

[1]

madmom.features.beats_hmm.**exponential_transition**(*from_intervals*, *to_intervals*, *transition_lambda*, *threshold=2.220446049250313e-16*, *norm=True*)

Exponential tempo transition.

Parameters

from_intervals [numpy array] Intervals where the transitions originate from.

to_intervals Intervals where the transitions terminate.

transition_lambda [float] Lambda for the exponential tempo change distribution (higher values prefer a constant tempo from one beat/bar to the next one). If None, allow only transitions from/to the same interval.

threshold [float, optional] Set transition probabilities below this threshold to zero.

norm [bool, optional] Normalize the emission probabilities to sum 1.

Returns

probabilities [numpy array, shape (num_from_intervals, num_to_intervals)] Probability of each transition from an interval to another.

References

[1]

class madmom.features.beats_hmm.**BeatTransitionModel**(*state_space*, *transition_lambda*)

Transition model for beat tracking with a HMM.

Within the beat the tempo stays the same; at beat boundaries transitions from one tempo (i.e. interval) to another are allowed, following an exponential distribution.

Parameters

state_space [*BeatStateSpace* instance] BeatStateSpace instance.

transition_lambda [float] Lambda for the exponential tempo change distribution (higher values prefer a constant tempo from one beat to the next one).

References

[1]

class madmom.features.beats_hmm.**BarTransitionModel**(*state_space*, *transition_lambda*)

Transition model for bar tracking with a HMM.

Within the beats of the bar the tempo stays the same; at beat boundaries transitions from one tempo (i.e. interval) to another following an exponential distribution are allowed.

Parameters

state_space [`BarStateSpace` instance] BarStateSpace instance.

transition_lambda [float or list] Lambda for the exponential tempo change distribution (higher values prefer a constant tempo from one beat to the next one). None can be used to set the tempo change probability to 0. If a list is given, the individual values represent the lambdas for each transition into the beat at this index position.

Notes

Bars performing tempo changes only at bar boundaries (and not at the beat boundaries) must have set all but the first *transition_lambda* values to None, e.g. [100, None, None] for a bar with 3 beats.

References

[1]

class madmom.features.beats_hmm.**MultiPatternTransitionModel**(*transition_models*, *transition_prob=None*)

Transition model for pattern tracking with a HMM.

Add transitions with the given probability between the individual transition models. These transition models must correspond to the state spaces forming a `MultiPatternStateSpace`.

Parameters

transition_models [list] List with `TransitionModel` instances.

transition_prob [numpy array or float, optional] Probabilities to change the pattern at pattern boundaries. If an array is given, the first dimension corresponds to the origin pattern, the second to the destination pattern. If a single value is given, a uniform transition distribution to all other patterns is assumed. Set to None to stay within the same pattern.

class madmom.features.beats_hmm.**RNNBeatTrackingObservationModel**(*state_space*, *observation_lambda*)

Observation model for beat tracking with a HMM.

Parameters

state_space [`BeatStateSpace` instance] BeatStateSpace instance.

observation_lambda [int] Split one beat period into *observation_lambda* parts, the first representing beat states and the remaining non-beat states.

References

[1]

log_densities(*observations*)
Compute the log densities of the observations.

Parameters

observations [numpy array, shape (N,)] Observations (i.e. 1D beat activations of the RNN).

Returns

numpy array, shape (N, 2) Log densities of the observations, the columns represent the observation log probability densities for no-beats and beats.

class madmom.features.beats_hmm.**RNNDownBeatTrackingObservationModel**(*state_space*, *observation_lambda*)
Observation model for downbeat tracking with a HMM.

Parameters

state_space [*BarStateSpace* instance] BarStateSpace instance.

observation_lambda [int] Split each (down-)beat period into *observation_lambda* parts, the first representing (down-)beat states and the remaining non-beat states.

References

[1]

log_densities(*observations*)
Compute the log densities of the observations.

Parameters

observations [numpy array, shape (N, 2)] Observations (i.e. 2D activations of a RNN, the columns represent 'beat' and 'downbeat' probabilities)

Returns

numpy array, shape (N, 3) Log densities of the observations, the columns represent the observation log probability densities for no-beats, beats and downbeats.

class madmom.features.beats_hmm.**GMMPatternTrackingObservationModel**(*pattern_files*, *state_space*)
Observation model for GMM based beat tracking with a HMM.

Parameters

pattern_files [list] List with files representing the rhythmic patterns, one entry per pattern; each pattern being a list with fitted GMMs.

state_space [*MultiPatternStateSpace* instance] Multi pattern state space.

References

[1]

log_densities(*observations*)
Compute the log densities of the observations using (a) GMM(s).

Parameters

observations [numpy array] Observations (i.e. multi-band spectral flux features).

Returns

numpy array, shape (N, num_gmms) Log densities of the observations, the columns represent the observation log probability densities for the individual GMMs.

8.2.4 madmom.features.chords

This module contains chord recognition related functionality.

madmom.features.chords.**majmin_targets_to_chord_labels**(*targets, fps*)

Converts a series of major/minor chord targets to human readable chord labels. Targets are assumed to be spaced equidistant in time as defined by the *fps* parameter (each target represents one 'frame').

Ids 0-11 encode major chords starting with root 'A', 12-23 minor chords. Id 24 represents 'N', the no-chord class.

Parameters

targets [iterable] Iterable containing chord class ids.

fps [float] Frames per second. Consecutive class

Returns

chord labels [list] List of tuples of the form (start time, end time, chord label)

class madmom.features.chords.**DeepChromaChordRecognitionProcessor**(*model=None, fps=10, **kwargs*)

Recognise major and minor chords from deep chroma vectors *[1]* using a Conditional Random Field.

Parameters

model [str] File containing the CRF model. If None, use the model supplied with madmom.

fps [float] Frames per second. Must correspond to the fps of the incoming activations and the model.

References

[1]

Examples

To recognise chords in an audio file using the DeepChromaChordRecognitionProcessor you first need to create a madmom.audio.chroma.DeepChromaProcessor to extract the appropriate chroma vectors.

```
>>> from madmom.audio.chroma import DeepChromaProcessor
>>> dcp = DeepChromaProcessor()
>>> dcp
<madmom.audio.chroma.DeepChromaProcessor object at ...>
```

Then, create the DeepChromaChordRecognitionProcessor to decode a chord sequence from the extracted chromas:

```
>>> decode = DeepChromaChordRecognitionProcessor()
>>> decode
<madmom.features.chords.DeepChromaChordRecognitionProcessor object at ...>
```

To transcribe the chords, you can either manually call the processors one after another,

```
>>> chroma = dcp('tests/data/audio/sample2.wav')
>>> decode(chroma)
...
array([(0. , 1.6, 'F:maj'), (1.6, 2.5, 'A:maj'), (2.5, 4.1, 'D:maj')],
      dtype=[('start', '<f8'), ('end', '<f8'), ('label', 'O')])
```

or create a *SequentialProcessor* that connects them:

```
>>> from madmom.processors import SequentialProcessor
>>> chordrec = SequentialProcessor([dcp, decode])
>>> chordrec('tests/data/audio/sample2.wav')
...
array([(0. , 1.6, 'F:maj'), (1.6, 2.5, 'A:maj'), (2.5, 4.1, 'D:maj')],
      dtype=[('start', '<f8'), ('end', '<f8'), ('label', 'O')])
```

class madmom.features.chords.**CNNChordFeatureProcessor**(***kwargs*)

Extract learned features for chord recognition, as described in *[1]*.

References

[1]

Examples

```
>>> proc = CNNChordFeatureProcessor()
>>> proc
<madmom.features.chords.CNNChordFeatureProcessor object at 0x...>
>>> features = proc('tests/data/audio/sample2.wav')
>>> features.shape
(41, 128)
>>> features
array([[0.05798, 0.     , ..., 0.02757, 0.014  ],
       [0.06604, 0.     , ..., 0.02898, 0.00886],
       ...,
       [0.00655, 0.1166 , ..., 0.00651, 0.     ],
       [0.01476, 0.11185, ..., 0.00287, 0.     ]])
```

class madmom.features.chords.**CRFChordRecognitionProcessor**(*model=None, fps=10, **kwargs*)

Recognise major and minor chords from learned features extracted by a convolutional neural network, as described in *[1]*.

Parameters

model [str] File containing the CRF model. If None, use the model supplied with madmom.

fps [float] Frames per second. Must correspond to the fps of the incoming activations and the model.

References

[1]

Examples

To recognise chords using the CRFChordRecognitionProcessor, you first need to extract features using the CNNChordFeatureProcessor.

```
>>> featproc = CNNChordFeatureProcessor()
>>> featproc
<madmom.features.chords.CNNChordFeatureProcessor object at 0x...>
```

Then, create the CRFChordRecognitionProcessor to decode a chord sequence from the extracted features:

```
>>> decode = CRFChordRecognitionProcessor()
>>> decode
<madmom.features.chords.CRFChordRecognitionProcessor object at 0x...>
```

To transcribe the chords, you can either manually call the processors one after another,

```
>>> feats = featproc('tests/data/audio/sample2.wav')
>>> decode(feats)
...
...
array([(0. , 0.2, 'N'), (0.2, 1.6, 'F:maj'),
       (1.6, 2.4..., 'A:maj'), (2.4..., 4.1, 'D:min')],
      dtype=[('start', '<f8'), ('end', '<f8'), ('label', 'O')])
```

or create a *madmom.processors.SequentialProcessor* that connects them:

```
>>> from madmom.processors import SequentialProcessor
>>> chordrec = SequentialProcessor([featproc, decode])
>>> chordrec('tests/data/audio/sample2.wav')
...
...
array([(0. , 0.2, 'N'), (0.2, 1.6, 'F:maj'),
       (1.6, 2.4..., 'A:maj'), (2.4..., 4.1, 'D:min')],
      dtype=[('start', '<f8'), ('end', '<f8'), ('label', 'O')])
```

8.2.5 madmom.features.downbeats

This module contains downbeat and bar tracking related functionality.

class madmom.features.downbeats.**RNNDownBeatProcessor**(***kwargs*)

Processor to get a joint beat and downbeat activation function from multiple RNNs.

References

[1]

Examples

Create a RNNDownBeatProcessor and pass a file through the processor. The returned 2d array represents the probabilities at each frame, sampled at 100 frames per second. The columns represent 'beat' and 'downbeat'.

```
>>> proc = RNNDownBeatProcessor()
>>> proc
<madmom.features.downbeats.RNNDownBeatProcessor object at 0x...>
>>> proc('tests/data/audio/sample.wav')
...
array([[0.00011, 0.00037],
       [0.00008, 0.00043],
       ...,
       [0.00791, 0.00169],
       [0.03425, 0.00494]], dtype=float32)
```

class madmom.features.downbeats.**DBNDownBeatTrackingProcessor**(*beats_per_bar*, *min_bpm=55.0*, *max_bpm=215.0*, *num_tempi=60*, *transition_lambda=100*, *observation_lambda=16*, *threshold=0.05*, *correct=True*, *fps=None*, ***kwargs*)

Downbeat tracking with RNNs and a dynamic Bayesian network (DBN) approximated by a Hidden Markov Model (HMM).

Parameters

beats_per_bar [int or list] Number of beats per bar to be modeled. Can be either a single number or a list or array with bar lengths (in beats).

min_bpm [float or list, optional] Minimum tempo used for beat tracking [bpm]. If a list is given, each item corresponds to the number of beats per bar at the same position.

max_bpm [float or list, optional] Maximum tempo used for beat tracking [bpm]. If a list is given, each item corresponds to the number of beats per bar at the same position.

num_tempi [int or list, optional] Number of tempi to model; if set, limit the number of tempi and use a log spacing, otherwise a linear spacing. If a list is given, each item corresponds to the number of beats per bar at the same position.

transition_lambda [float or list, optional] Lambda for the exponential tempo change distribution (higher values prefer a constant tempo from one beat to the next one). If a list is given, each item corresponds to the number of beats per bar at the same position.

observation_lambda [int, optional] Split one (down-)beat period into *observation_lambda* parts, the first representing (down-)beat states and the remaining non-beat states.

threshold [float, optional] Threshold the RNN (down-)beat activations before Viterbi decoding.

correct [bool, optional] Correct the beats (i.e. align them to the nearest peak of the (down-)beat activation function).

fps [float, optional] Frames per second.

References

[1]

Examples

Create a DBNDownBeatTrackingProcessor. The returned array represents the positions of the beats and their position inside the bar. The position is given in seconds, thus the expected sampling rate is needed. The position inside the bar follows the natural counting and starts at 1.

The number of beats per bar which should be modelled must be given, all other parameters (e.g. tempo range) are optional but must have the same length as *beats_per_bar*, i.e. must be given for each bar length.

```
>>> proc = DBNDownBeatTrackingProcessor(beats_per_bar=[3, 4], fps=100)
>>> proc
<madmom.features.downbeats.DBNDownBeatTrackingProcessor object at 0x...>
```

Call this DBNDownBeatTrackingProcessor with the beat activation function returned by RNNDownBeatProcessor to obtain the beat positions.

```
>>> act = RNNDownBeatProcessor()('tests/data/audio/sample.wav')
>>> proc(act)
array([[0.09, 1. ],
       [0.45, 2. ],
       ...,
       [2.14, 3. ],
       [2.49, 4. ]])
```

process (*activations*, ***kwargs*)
Detect the (down-)beats in the given activation function.

Parameters

activations [numpy array, shape (num_frames, 2)] Activation function with probabilities corresponding to beats and downbeats given in the first and second column, respectively.

Returns

beats [numpy array, shape (num_beats, 2)] Detected (down-)beat positions [seconds] and beat numbers.

static add_arguments (*parser*, *beats_per_bar*, *min_bpm=55.0*, *max_bpm=215.0*, *num_tempi=60*, *transition_lambda=100*, *observation_lambda=16*, *threshold=0.05*, *correct=True*)
Add DBN downbeat tracking related arguments to an existing parser object.

Parameters

parser [argparse parser instance] Existing argparse parser object.

beats_per_bar [int or list, optional] Number of beats per bar to be modeled. Can be either a single number or a list with bar lengths (in beats).

min_bpm [float or list, optional] Minimum tempo used for beat tracking [bpm]. If a list is given, each item corresponds to the number of beats per bar at the same position.

max_bpm [float or list, optional] Maximum tempo used for beat tracking [bpm]. If a list is given, each item corresponds to the number of beats per bar at the same position.

num_tempi [int or list, optional] Number of tempi to model; if set, limit the number of tempi and use a log spacing, otherwise a linear spacing. If a list is given, each item corresponds to the number of beats per bar at the same position.

transition_lambda [float or list, optional] Lambda for the exponential tempo change distribution (higher values prefer a constant tempo over a tempo change from one beat to the next one). If a list is given, each item corresponds to the number of beats per bar at the same position.

observation_lambda [float, optional] Split one (down-)beat period into *observation_lambda* parts, the first representing (down-)beat states and the remaining non-beat states.

threshold [float, optional] Threshold the RNN (down-)beat activations before Viterbi decoding.

correct [bool, optional] Correct the beats (i.e. align them to the nearest peak of the (down-)beat activation function).

Returns

parser_group [argparse argument group] DBN downbeat tracking argument parser group

class madmom.features.downbeats.**PatternTrackingProcessor**(*pattern_files*, *min_bpm=(55, 60)*, *max_bpm=(205, 225)*, *num_tempi=None*, *transition_lambda=100*, *fps=None*, ***kwargs*)

Pattern tracking with a dynamic Bayesian network (DBN) approximated by a Hidden Markov Model (HMM).

Parameters

pattern_files [list] List of files with the patterns (including the fitted GMMs and information about the number of beats).

min_bpm [list, optional] Minimum tempi used for pattern tracking [bpm].

max_bpm [list, optional] Maximum tempi used for pattern tracking [bpm].

num_tempi [int or list, optional] Number of tempi to model; if set, limit the number of tempi and use a log spacings, otherwise a linear spacings.

transition_lambda [float or list, optional] Lambdas for the exponential tempo change distributions (higher values prefer constant tempi from one beat to the next one).

fps [float, optional] Frames per second.

Notes

min_bpm, *max_bpm*, *num_tempo_states*, and *transition_lambda* must contain as many items as rhythmic patterns are modeled (i.e. length of *pattern_files*). If a single value is given for *num_tempo_states* and *transition_lambda*, this value is used for all rhythmic patterns.

Instead of the originally proposed state space and transition model for the DBN *[1]*, the more efficient version proposed in *[2]* is used.

References

[1], *[2]*

Examples

Create a PatternTrackingProcessor from the given pattern files. These pattern files include fitted GMMs for the observation model of the HMM. The returned array represents the positions of the beats and their position inside the bar. The position is given in seconds, thus the expected sampling rate is needed. The position inside the bar follows the natural counting and starts at 1.

```
>>> from madmom.models import PATTERNS_BALLROOM
>>> proc = PatternTrackingProcessor(PATTERNS_BALLROOM, fps=50)
>>> proc
<madmom.features.downbeats.PatternTrackingProcessor object at 0x...>
```

Call this PatternTrackingProcessor with a multi-band spectrogram to obtain the beat and downbeat positions. The parameters of the spectrogram have to correspond to those used to fit the GMMs.

```
>>> from madmom.audio.spectrogram import LogarithmicSpectrogramProcessor,␣
↪SpectrogramDifferenceProcessor, MultiBandSpectrogramProcessor
>>> from madmom.processors import SequentialProcessor
>>> log = LogarithmicSpectrogramProcessor()
>>> diff = SpectrogramDifferenceProcessor(positive_diffs=True)
>>> mb = MultiBandSpectrogramProcessor(crossover_frequencies=[270])
>>> pre_proc = SequentialProcessor([log, diff, mb])
```

```
>>> act = pre_proc('tests/data/audio/sample.wav')
>>> proc(act)
array([[0.82, 4.  ],
       [1.78, 1.  ],
       ...,
       [3.7 , 3.  ],
       [4.66, 4.  ]])
```

process (*features*, ***kwargs*)

Detect the (down-)beats given the features.

Parameters

features [numpy array] Multi-band spectral features.

Returns

beats [numpy array, shape (num_beats, 2)] Detected (down-)beat positions [seconds] and beat numbers.

static add_arguments (*parser*, *pattern_files=None*, *min_bpm=(55, 60)*, *max_bpm=(205, 225)*, *num_tempi=None*, *transition_lambda=100*)

Add DBN related arguments for pattern tracking to an existing parser object.

Parameters

parser [argparse parser instance] Existing argparse parser object.

pattern_files [list] Load the patterns from these files.

min_bpm [list, optional] Minimum tempi used for beat tracking [bpm].

max_bpm [list, optional] Maximum tempi used for beat tracking [bpm].

num_tempi [int or list, optional] Number of tempi to model; if set, limit the number of states and use log spacings, otherwise a linear spacings.

transition_lambda [float or list, optional] Lambdas for the exponential tempo change distribution (higher values prefer constant tempi from one beat to the next one).

Returns

parser_group [argparse argument group] Pattern tracking argument parser group

Notes

pattern_files, *min_bpm*, *max_bpm*, *num_tempi*, and *transition_lambda* must have the same number of items.

class madmom.features.downbeats.**LoadBeatsProcessor**(*beats*, *files=None*, *beats_suffix=None*, ***kwargs*)

Load beat times from file or handle.

process(*data=None*, ***kwargs*)

Load the beats from file (handle) or read them from STDIN.

process_single()

Load the beats in bulk-mode (i.e. all at once) from the input stream or file.

Returns

beats [numpy array] Beat positions [seconds].

process_batch(*filename*)

Load beat times from file.

First match the given input filename to the beat filenames, then load the beats.

Parameters

filename [str] Input file name.

Returns

beats [numpy array] Beat positions [seconds].

Notes

Both the file names to search for the beats as well as the suffix to determine the beat files must be given at instantiation time.

static add_arguments(*parser*, *beats=<open file '<stdin>', mode 'r'>*, *beats_suffix='.beats.txt'*)

Add beat loading related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

beats [FileType, optional] Where to read the beats from ('single' mode).

beats_suffix [str, optional] Suffix of beat files ('batch' mode)

Returns

argparse argument group Beat loading argument parser group.

class madmom.features.downbeats.**SyncronizeFeaturesProcessor**(*beat_subdivisions*, *fps*, ***kwargs*)

Synchronize features to beats.

First, divide a beat interval into *beat_subdivision* divisions. Then summarise all features that fall into one subdivision. If no feature value for a subdivision is found, it is set to 0.

Parameters

beat_subdivisions [int] Number of subdivisions a beat is divided into.

fps [float] Frames per second.

process (*data*, ***kwargs*)

Synchronize features to beats.

Average all feature values that fall into a window of beat duration / beat subdivisions, centered on the beat positions or interpolated subdivisions, starting with the first beat.

Parameters

data [tuple (features, beats)] Tuple of two numpy arrays, the first containing features to be synchronized and second the beat times.

Returns

numpy array (num beats - 1, beat subdivisions, features dim.) Beat synchronous features.

class madmom.features.downbeats.**RNNBarProcessor** (*beat_subdivisions=(4, 2)*, *fps=100*, ***kwargs*)

Retrieve a downbeat activation function from a signal and pre-determined beat positions by obtaining beat-synchronous harmonic and percussive features which are processed with a GRU-RNN.

Parameters

beat_subdivisions [tuple, optional] Number of beat subdivisions for the percussive and harmonic feature.

References

[1]

Examples

Create an RNNBarProcessor and pass an audio file and pre-determined (or given) beat positions through the processor. The returned tuple contains the beats positions and the probability to be a downbeat.

```
>>> proc = RNNBarProcessor()
>>> proc
<madmom.features.downbeats.RNNBarProcessor object at 0x...>
>>> beats = np.loadtxt('tests/data/detections/sample.dbn_beat_tracker.txt')
>>> downbeat_prob = proc(('tests/data/audio/sample.wav', beats))
>>> np.around(downbeat_prob, decimals=3)
...
array([[0.1  , 0.378],
       [0.45 , 0.19 ],
       [0.8  , 0.112],
       [1.12 , 0.328],
       [1.48 , 0.27 ],
       [1.8  , 0.181],
       [2.15 , 0.162],
       [2.49 ,   nan]])
```

process (*data*, ***kwargs*)
Retrieve a downbeat activation function from a signal and beat positions.

Parameters

data [tuple] Tuple containg a signal or file (handle) and corresponding beat times [seconds].

Returns

numpy array, shape (num_beats, 2) Array containing the beat positions (first column) and the corresponding downbeat activations, i.e. the probability that a beat is a downbeat (second column).

Notes

Since features are synchronized to the beats, and the probability of being a downbeat depends on a whole beat duration, only num_beats-1 activations can be computed and the last value is filled with 'NaN'.

class madmom.features.downbeats.**DBNBarTrackingProcessor** (*beats_per_bar=(3, 4)*, *observation_weight=100*, *meter_change_prob=1e-07*, ***kwargs*)
Bar tracking with a dynamic Bayesian network (DBN) approximated by a Hidden Markov Model (HMM).

Parameters

beats_per_bar [int or list] Number of beats per bar to be modeled. Can be either a single number or a list or array with bar lengths (in beats).

observation_weight [int, optional] Weight for the downbeat activations.

meter_change_prob [float, optional] Probability to change meter at bar boundaries.

Examples

Create a DBNBarTrackingProcessor. The returned array represents the positions of the beats and their position inside the bar. The position inside the bar follows the natural counting and starts at 1.

The number of beats per bar which should be modelled must be given, all other parameters (e.g. probability to change the meter at bar boundaries) are optional but must have the same length as *beats_per_bar*.

```
>>> proc = DBNBarTrackingProcessor(beats_per_bar=[3, 4])
>>> proc
<madmom.features.downbeats.DBNBarTrackingProcessor object at 0x...>
```

Call this DBNDownBeatTrackingProcessor with beat positions and downbeat activation function returned by RNNBarProcessor to obtain the positions.

```
>>> beats = np.loadtxt('tests/data/detections/sample.dbn_beat_tracker.txt')
>>> act = RNNBarProcessor()(('tests/data/audio/sample.wav', beats))
>>> proc(act)
array([[0.1 , 1. ],
       [0.45, 2. ],
       [0.8 , 3. ],
       [1.12, 1. ],
       [1.48, 2. ],
       [1.8 , 3. ],
```

(continues on next page)

(continued from previous page)

```
[2.15, 1. ],
[2.49, 2. ]])
```

process (*data*, ***kwargs*)
Detect downbeats from the given beats and activation function with Viterbi decoding.

Parameters

data [numpy array, shape (num_beats, 2)] Array containing beat positions (first column) and corresponding downbeat activations (second column).

Returns

numpy array, shape (num_beats, 2) Decoded (down-)beat positions and beat numbers.

Notes

The position of the last beat is not decoded, but rather extrapolated based on the position and meter of the second to last beat.

classmethod add_arguments (*parser*, *beats_per_bar*, *observation_weight=100*, *meter_change_prob=1e-07*)
Add DBN related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

beats_per_bar [int or list, optional] Number of beats per bar to be modeled. Can be either a single number or a list with bar lengths (in beats).

observation_weight [float, optional] Weight for the activations at downbeat times.

meter_change_prob [float, optional] Probability to change meter at bar boundaries.

Returns

parser_group [argparse argument group] DBN bar tracking argument parser group

8.2.6 madmom.features.key

This module contains key recognition related functionality.

madmom.features.key.**key_prediction_to_label** (*prediction*)
Convert key class id to a human-readable key name.

Parameters

prediction [numpy array] Array containing the probabilities of each key class.

Returns

str Human-readable key name.

class madmom.features.key.**CNNKeyRecognitionProcessor** (*nn_files=None*, ***kwargs*)
Recognise the global key of a musical piece using a Convolutional Neural Network as described in [1].

Parameters

nn_files [list, optional] List with trained CNN model files. Per default ('None'), an ensemble of networks will be used.

References

[1]

Examples

Create a CNNKeyRecognitionProcessor and pass a file through it. The returned array represents the probability of each key class.

```
>>> proc = CNNKeyRecognitionProcessor()
>>> proc
<madmom.features.key.CNNKeyRecognitionProcessor object at 0x...>
>>> proc('tests/data/audio/sample.wav')
array([[0.03426, 0.0331 , 0.02979, 0.04423, 0.04215, 0.0311 , 0.05225,
        0.04263, 0.04141, 0.02907, 0.03755, 0.09546, 0.0431 , 0.02792,
        0.02138, 0.05589, 0.03276, 0.02786, 0.02415, 0.04608, 0.05329,
        0.02804, 0.03868, 0.08786]])
```

8.2.7 madmom.features.notes

This module contains note transcription related functionality.

Notes are stored as numpy arrays with the following column definition:

'note_time' 'MIDI_note' ['duration' ['MIDI_velocity']]

class madmom.features.notes.**RNNPianoNoteProcessor**(***kwargs*)

Processor to get a (piano) note activation function from a RNN.

Examples

Create a RNNPianoNoteProcessor and pass a file through the processor to obtain a note onset activation function (sampled with 100 frames per second).

```
>>> proc = RNNPianoNoteProcessor()
>>> proc
<madmom.features.notes.RNNPianoNoteProcessor object at 0x...>
>>> act = proc('tests/data/audio/sample.wav')
>>> act.shape
(281, 88)
>>> act
array([[-0.00014,  0.0002 , ..., -0.     ,  0.     ],
       [ 0.00008,  0.0001 , ...,  0.00006, -0.00001],
       ...,
       [-0.00005, -0.00011, ...,  0.00005, -0.00001],
       [-0.00017,  0.00002, ...,  0.00009, -0.00009]], dtype=float32)
```

class madmom.features.notes.**NotePeakPickingProcessor**(*threshold=0.5, smooth=0.0, pre_avg=0.0, post_avg=0.0, pre_max=0.0, post_max=0.0, combine=0.03, delay=0.0, online=False, fps=100, **kwargs*)

This class implements the note peak-picking functionality.

Parameters

threshold [float] Threshold for peak-picking.

smooth [float, optional] Smooth the activation function over *smooth* seconds.

pre_avg [float, optional] Use *pre_avg* seconds past information for moving average.

post_avg [float, optional] Use *post_avg* seconds future information for moving average.

pre_max [float, optional] Use *pre_max* seconds past information for moving maximum.

post_max [float, optional] Use *post_max* seconds future information for moving maximum.

combine [float, optional] Only report one note per pitch within *combine* seconds.

delay [float, optional] Report the detected notes *delay* seconds delayed.

online [bool, optional] Use online peak-picking, i.e. no future information.

fps [float, optional] Frames per second used for conversion of timings.

Returns

notes [numpy array] Detected notes [seconds, pitch].

Notes

If no moving average is needed (e.g. the activations are independent of the signal's level as for neural network activations), *pre_avg* and *post_avg* should be set to 0. For peak picking of local maxima, set *pre_max* >= 1. / *fps* and *post_max* >= 1. / *fps*. For online peak picking, all *post_* parameters are set to 0.

Examples

Create a PeakPickingProcessor. The returned array represents the positions of the onsets in seconds, thus the expected sampling rate has to be given.

```
>>> proc = NotePeakPickingProcessor(fps=100)
>>> proc
<madmom.features.notes.NotePeakPickingProcessor object at 0x...>
```

Call this NotePeakPickingProcessor with the note activations from an RNNPianoNoteProcessor.

```
>>> act = RNNPianoNoteProcessor()('tests/data/audio/stereo_sample.wav')
>>> proc(act)
array([[ 0.14, 72.  ],
       [ 1.56, 41.  ],
       [ 3.37, 75.  ]])
```

process (*activations*, ***kwargs*)

Detect the notes in the given activation function.

Parameters

activations [numpy array] Note activation function.

Returns

onsets [numpy array] Detected notes [seconds, pitches].

8.2.8 madmom.features.onsets

This module contains onset detection related functionality.

madmom.features.onsets.**wrap_to_pi**(*phase*)
Wrap the phase information to the range -$\pi \ldots \pi$.

Parameters

phase [numpy array] Phase of the STFT.

Returns

wrapped_phase [numpy array] Wrapped phase.

madmom.features.onsets.**correlation_diff**(*spec*, *diff_frames=1*, *pos=False*, *diff_bins=1*)
Calculates the difference of the magnitude spectrogram relative to the N-th previous frame shifted in frequency to achieve the highest correlation between these two frames.

Parameters

spec [numpy array] Magnitude spectrogram.

diff_frames [int, optional] Calculate the difference to the *diff_frames*-th previous frame.

pos [bool, optional] Keep only positive values.

diff_bins [int, optional] Maximum number of bins shifted for correlation calculation.

Returns

correlation_diff [numpy array] (Positive) magnitude spectrogram differences.

Notes

This function is only because of completeness, it is not intended to be actually used, since it is extremely slow. Please consider the superflux() function, since if performs equally well but much faster.

madmom.features.onsets.**high_frequency_content**(*spectrogram*)
High Frequency Content.

Parameters

spectrogram [Spectrogram instance] Spectrogram instance.

Returns

high_frequency_content [numpy array] High frequency content onset detection function.

References

[1]

madmom.features.onsets.**spectral_diff**(*spectrogram*, *diff_frames=None*)
Spectral Diff.

Parameters

spectrogram [Spectrogram instance] Spectrogram instance.

diff_frames [int, optional] Number of frames to calculate the diff to.

Returns

spectral_diff [numpy array] Spectral diff onset detection function.

References

[1]

`madmom.features.onsets.`**`spectral_flux`**(*spectrogram*, *diff_frames=None*)

Spectral Flux.

Parameters

spectrogram [`Spectrogram` instance] Spectrogram instance.

diff_frames [int, optional] Number of frames to calculate the diff to.

Returns

spectral_flux [numpy array] Spectral flux onset detection function.

References

[1]

`madmom.features.onsets.`**`superflux`**(*spectrogram*, *diff_frames=None*, *diff_max_bins=3*)

SuperFlux method with a maximum filter vibrato suppression stage.

Calculates the difference of bin k of the magnitude spectrogram relative to the N-th previous frame with the maximum filtered spectrogram.

Parameters

spectrogram [`Spectrogram` instance] Spectrogram instance.

diff_frames [int, optional] Number of frames to calculate the diff to.

diff_max_bins [int, optional] Number of bins used for maximum filter.

Returns

superflux [numpy array] SuperFlux onset detection function.

Notes

This method works only properly, if the spectrogram is filtered with a filterbank of the right frequency spacing. Filter banks with 24 bands per octave (i.e. quarter-tone resolution) usually yield good results. With *max_bins* = 3, the maximum of the bins k-1, k, k+1 of the frame *diff_frames* to the left is used for the calculation of the difference.

References

[1]

`madmom.features.onsets.`**`complex_flux`**(*spectrogram*, *diff_frames=None*, *diff_max_bins=3*, *temporal_filter=3*, *temporal_origin=0*)

ComplexFlux.

ComplexFlux is based on the SuperFlux, but adds an additional local group delay based tremolo suppression.

Parameters

spectrogram [`Spectrogram` instance] `Spectrogram` instance.

diff_frames [int, optional] Number of frames to calculate the diff to.

diff_max_bins [int, optional] Number of bins used for maximum filter.

temporal_filter [int, optional] Temporal maximum filtering of the local group delay [frames].

temporal_origin [int, optional] Origin of the temporal maximum filter.

Returns

complex_flux [numpy array] ComplexFlux onset detection function.

References

[1]

`madmom.features.onsets.`**`modified_kullback_leibler`**`(`*spectrogram, diff_frames=1, epsilon=2.220446049250313e-16*`)`

Modified Kullback-Leibler.

Parameters

spectrogram [`Spectrogram` instance] `Spectrogram` instance.

diff_frames [int, optional] Number of frames to calculate the diff to.

epsilon [float, optional] Add *epsilon* to the *spectrogram* avoid division by 0.

Returns

modified_kullback_leibler [numpy array] MKL onset detection function.

Notes

The implementation presented in *[1]* is used instead of the original work presented in *[2]*.

References

[1], *[2]*

`madmom.features.onsets.`**`phase_deviation`**`(`*spectrogram*`)`

Phase Deviation.

Parameters

spectrogram [`Spectrogram` instance] `Spectrogram` instance.

Returns

phase_deviation [numpy array] Phase deviation onset detection function.

References

[1]

`madmom.features.onsets.`**`weighted_phase_deviation`**`(`*spectrogram*`)`

Weighted Phase Deviation.

Parameters

spectrogram [`Spectrogram` instance] `Spectrogram` instance.

Returns

weighted_phase_deviation [numpy array] Weighted phase deviation onset detection function.

References

[1]

`madmom.features.onsets.`**`normalized_weighted_phase_deviation`**(*spectrogram*, *epsilon=2.220446049250313e-16*)

Normalized Weighted Phase Deviation.

Parameters

spectrogram [`Spectrogram` instance] `Spectrogram` instance.

epsilon [float, optional] Add *epsilon* to the *spectrogram* avoid division by 0.

Returns

normalized_weighted_phase_deviation [numpy array] Normalized weighted phase deviation onset detection function.

References

[1]

`madmom.features.onsets.`**`complex_domain`**(*spectrogram*)

Complex Domain.

Parameters

spectrogram [`Spectrogram` instance] `Spectrogram` instance.

Returns

complex_domain [numpy array] Complex domain onset detection function.

References

[1]

`madmom.features.onsets.`**`rectified_complex_domain`**(*spectrogram*, *diff_frames=None*)

Rectified Complex Domain.

Parameters

spectrogram [`Spectrogram` instance] `Spectrogram` instance.

diff_frames [int, optional] Number of frames to calculate the diff to.

Returns

rectified_complex_domain [numpy array] Rectified complex domain onset detection function.

References

[1]

class madmom.features.onsets.**SpectralOnsetProcessor**(*onset_method='spectral_flux'*, ***kwargs*)

The SpectralOnsetProcessor class implements most of the common onset detection functions based on the magnitude or phase information of a spectrogram.

Parameters

onset_method [str, optional] Onset detection function. See *METHODS* for possible values.

kwargs [dict, optional] Keyword arguments passed to the pre-processing chain to obtain a spectral representation of the signal.

Notes

If the spectrogram should be filtered, the *filterbank* parameter must contain a valid Filterbank, if it should be scaled logarithmically, *log* must be set accordingly.

References

[1], *[2]*

Examples

Create a SpectralOnsetProcessor and pass a file through the processor to obtain an onset detection function. Per default the spectral flux *[1]* is computed on a simple Spectrogram.

```
>>> sodf = SpectralOnsetProcessor()
>>> sodf
<madmom.features.onsets.SpectralOnsetProcessor object at 0x...>
>>> sodf.processors[-1]
<function spectral_flux at 0x...>
>>> sodf('tests/data/audio/sample.wav')
...
array([ 0. , 100.90121, ..., 26.30577, 20.94439], dtype=float32)
```

The parameters passed to the signal pre-processing chain can be set when creating the SpectralOnsetProcessor. E.g. to obtain the SuperFlux *[2]* onset detection function set these parameters:

```
>>> from madmom.audio.filters import LogarithmicFilterbank
>>> sodf = SpectralOnsetProcessor(onset_method='superflux', fps=200,
...                               filterbank=LogarithmicFilterbank,
...                               num_bands=24, log=np.log10)
>>> sodf('tests/data/audio/sample.wav')
...
array([ 0. , 0. , 2.0868 , 1.02404, ..., 0.29888, 0.12122], dtype=float32)
```

classmethod add_arguments(*parser*, *onset_method=None*)

Add spectral onset detection arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

onset_method [str, optional] Default onset detection method.

Returns

parser_group [argparse argument group] Spectral onset detection argument parser group.

class madmom.features.onsets.**RNNOnsetProcessor**(***kwargs*)

Processor to get a onset activation function from multiple RNNs.

Parameters

online [bool, optional] Choose networks suitable for online onset detection, i.e. use unidirectional RNNs.

Notes

This class uses either uni- or bi-directional RNNs. Contrary to [1], it uses simple tanh units as in [2]. Also the input representations changed to use logarithmically filtered and scaled spectrograms.

References

[1], *[2]*

Examples

Create a RNNOnsetProcessor and pass a file through the processor to obtain an onset detection function (sampled with 100 frames per second).

```
>>> proc = RNNOnsetProcessor()
>>> proc
<madmom.features.onsets.RNNOnsetProcessor object at 0x...>
>>> proc('tests/data/audio/sample.wav')
array([0.08313, 0.0024 , ... 0.00527], dtype=float32)
```

class madmom.features.onsets.**CNNOnsetProcessor**(***kwargs*)

Processor to get a onset activation function from a CNN.

Notes

The implementation follows as closely as possible the original one, but part of the signal pre-processing differs in minor aspects, so results can differ slightly, too.

References

[1]

Examples

Create a CNNOnsetProcessor and pass a file through the processor to obtain an onset detection function (sampled with 100 frames per second).

```
>>> proc = CNNOnsetProcessor()
>>> proc
<madmom.features.onsets.CNNOnsetProcessor object at 0x...>
>>> proc('tests/data/audio/sample.wav')
array([0.05369, 0.04205, ... 0.00014], dtype=float32)
```

madmom.features.onsets.**peak_picking**(*activations*, *threshold*, *smooth=None*, *pre_avg=0*, *post_avg=0*, *pre_max=1*, *post_max=1*)

Perform thresholding and peak-picking on the given activation function.

Parameters

activations [numpy array] Activation function.

threshold [float] Threshold for peak-picking

smooth [int or numpy array, optional] Smooth the activation function with the kernel (size).

pre_avg [int, optional] Use *pre_avg* frames past information for moving average.

post_avg [int, optional] Use *post_avg* frames future information for moving average.

pre_max [int, optional] Use *pre_max* frames past information for moving maximum.

post_max [int, optional] Use *post_max* frames future information for moving maximum.

Returns

peak_idx [numpy array] Indices of the detected peaks.

See also:

`smooth()`

Notes

If no moving average is needed (e.g. the activations are independent of the signal's level as for neural network activations), set *pre_avg* and *post_avg* to 0. For peak picking of local maxima, set *pre_max* and *post_max* to 1. For online peak picking, set all *post_* parameters to 0.

References

[1]

class madmom.features.onsets.**PeakPickingProcessor**(***kwargs*)

Deprecated as of version 0.15. Will be removed in version 0.16. Use either `OnsetPeakPickingProcessor` or `NotePeakPickingProcessor` instead.

process(*activations*, ***kwargs*)

Detect the peaks in the given activation function.

Parameters

activations [numpy array] Onset activation function.

Returns

peaks [numpy array] Detected onsets [seconds[, frequency bin]].

static add_arguments(*parser*, ***kwargs*)
Deprecated as of version 0.15. Will be removed in version 0.16. Use either `OnsetPeakPickingProcessor` or `NotePeakPickingProcessor` instead.

class madmom.features.onsets.**OnsetPeakPickingProcessor**(*threshold=0.5*, *smooth=0.0*, *pre_avg=0.0*, *post_avg=0.0*, *pre_max=0.0*, *post_max=0.0*, *combine=0.03*, *delay=0.0*, *online=False*, *fps=100*, ***kwargs*)

This class implements the onset peak-picking functionality. It transparently converts the chosen values from seconds to frames.

Parameters

threshold [float] Threshold for peak-picking.

smooth [float, optional] Smooth the activation function over *smooth* seconds.

pre_avg [float, optional] Use *pre_avg* seconds past information for moving average.

post_avg [float, optional] Use *post_avg* seconds future information for moving average.

pre_max [float, optional] Use *pre_max* seconds past information for moving maximum.

post_max [float, optional] Use *post_max* seconds future information for moving maximum.

combine [float, optional] Only report one onset within *combine* seconds.

delay [float, optional] Report the detected onsets *delay* seconds delayed.

online [bool, optional] Use online peak-picking, i.e. no future information.

fps [float, optional] Frames per second used for conversion of timings.

Returns

onsets [numpy array] Detected onsets [seconds].

Notes

If no moving average is needed (e.g. the activations are independent of the signal's level as for neural network activations), *pre_avg* and *post_avg* should be set to 0. For peak picking of local maxima, set *pre_max* >= 1. / *fps* and *post_max* >= 1. / *fps*. For online peak picking, all *post_* parameters are set to 0.

References

[1]

Examples

Create a PeakPickingProcessor. The returned array represents the positions of the onsets in seconds, thus the expected sampling rate has to be given.

```
>>> proc = OnsetPeakPickingProcessor(fps=100)
>>> proc
<madmom.features.onsets.OnsetPeakPickingProcessor object at 0x...>
```

Call this OnsetPeakPickingProcessor with the onset activation function from an RNNOnsetProcessor to obtain the onset positions.

```
>>> act = RNNOnsetProcessor()('tests/data/audio/sample.wav')
>>> proc(act)
array([0.09, 0.29, 0.45, ..., 2.34, 2.49, 2.67])
```

reset()

Reset OnsetPeakPickingProcessor.

process_offline(*activations*, ***kwargs*)

Detect the onsets in the given activation function.

Parameters

activations [numpy array] Onset activation function.

Returns

onsets [numpy array] Detected onsets [seconds].

process_online(*activations*, *reset=True*, ***kwargs*)

Detect the onsets in the given activation function.

Parameters

activations [numpy array] Onset activation function.

reset [bool, optional] Reset the processor to its initial state before processing.

Returns

onsets [numpy array] Detected onsets [seconds].

process_sequence(*activations*, ***kwargs*)

Detect the onsets in the given activation function.

Parameters

activations [numpy array] Onset activation function.

Returns

onsets [numpy array] Detected onsets [seconds].

static add_arguments(*parser*, *threshold=0.5*, *smooth=None*, *pre_avg=None*, *post_avg=None*, *pre_max=None*, *post_max=None*, *combine=0.03*, *delay=0.0*)

Add onset peak-picking related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

threshold [float] Threshold for peak-picking.

smooth [float, optional] Smooth the activation function over *smooth* seconds.

pre_avg [float, optional] Use *pre_avg* seconds past information for moving average.

post_avg [float, optional] Use *post_avg* seconds future information for moving average.

pre_max [float, optional] Use *pre_max* seconds past information for moving maximum.

post_max [float, optional] Use *post_max* seconds future information for moving maximum.

combine [float, optional] Only report one onset within *combine* seconds.

delay [float, optional] Report the detected onsets *delay* seconds delayed.

Returns

parser_group [argparse argument group] Onset peak-picking argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

8.2.9 madmom.features.tempo

This module contains tempo related functionality.

`madmom.features.tempo.`**`smooth_histogram`**(*histogram*, *smooth*)

Smooth the given histogram.

Parameters

histogram [tuple] Histogram (tuple of 2 numpy arrays, the first giving the strengths of the bins and the second corresponding delay values).

smooth [int or numpy array] Smoothing kernel (size).

Returns

histogram_bins [numpy array] Bins of the smoothed histogram.

histogram_delays [numpy array] Corresponding delays.

Notes

If *smooth* is an integer, a Hamming window of that length will be used as a smoothing kernel.

`madmom.features.tempo.`**`interval_histogram_acf`**(*activations*, *min_tau=1*, *max_tau=None*)

Compute the interval histogram of the given (beat) activation function via auto-correlation as in *[1]*.

Parameters

activations [numpy array] Beat activation function.

min_tau [int, optional] Minimal delay for the auto-correlation function [frames].

max_tau [int, optional] Maximal delay for the auto-correlation function [frames].

Returns

histogram_bins [numpy array] Bins of the tempo histogram.

histogram_delays [numpy array] Corresponding delays [frames].

References

[1]

madmom.features.tempo.**interval_histogram_comb**(*activations*, *alpha*, *min_tau=1*, *max_tau=None*)

Compute the interval histogram of the given (beat) activation function via a bank of resonating comb filters as in [1].

Parameters

activations [numpy array] Beat activation function.

alpha [float or numpy array] Scaling factor for the comb filter; if only a single value is given, the same scaling factor for all delays is assumed.

min_tau [int, optional] Minimal delay for the comb filter [frames].

max_tau [int, optional] Maximal delta for comb filter [frames].

Returns

histogram_bins [numpy array] Bins of the tempo histogram.

histogram_delays [numpy array] Corresponding delays [frames].

References

[1]

madmom.features.tempo.**dominant_interval**(*histogram*, *smooth=None*)

Extract the dominant interval of the given histogram.

Parameters

histogram [tuple] Histogram (tuple of 2 numpy arrays, the first giving the strengths of the bins and the second corresponding delay values).

smooth [int or numpy array, optional] Smooth the histogram with the given kernel (size).

Returns

interval [int] Dominant interval.

Notes

If *smooth* is an integer, a Hamming window of that length will be used as a smoothing kernel.

madmom.features.tempo.**detect_tempo**(*histogram*, *fps*)

Extract the tempo from the given histogram.

Parameters

histogram [tuple] Histogram (tuple of 2 numpy arrays, the first giving the strengths of the bins and the second corresponding delay values).

fps [float] Frames per second.

Returns

tempi [numpy array] Numpy array with the dominant tempi [bpm] (first column) and their relative strengths (second column).

class madmom.features.tempo.**TempoHistogramProcessor**(*min_bpm*, *max_bpm*, *hist_buffer=10.0*, *fps=None*, *online=False*, ***kwargs*)

Tempo Histogram Processor class.

Parameters

min_bpm [float] Minimum tempo to detect [bpm].

max_bpm [float] Maximum tempo to detect [bpm].

hist_buffer [float] Aggregate the tempo histogram over *hist_buffer* seconds.

fps [float, optional] Frames per second.

Notes

This abstract class provides the basic tempo histogram functionality. Please use one of the following implementations:

- *CombFilterTempoHistogramProcessor*,
- *ACFTempoHistogramProcessor* or
- *DBNTempoHistogramProcessor*.

min_interval
Minimum beat interval [frames].

max_interval
Maximum beat interval [frames].

intervals
Beat intervals [frames].

reset()
Reset the tempo histogram aggregation buffer.

class madmom.features.tempo.**CombFilterTempoHistogramProcessor**(*min_bpm=40.0*, *max_bpm=250.0*, *alpha=0.79*, *hist_buffer=10.0*, *fps=None*, *online=False*, ***kwargs*)

Create a tempo histogram with a bank of resonating comb filters.

Parameters

min_bpm [float, optional] Minimum tempo to detect [bpm].

max_bpm [float, optional] Maximum tempo to detect [bpm].

alpha [float, optional] Scaling factor for the comb filter.

hist_buffer [float] Aggregate the tempo histogram over *hist_buffer* seconds.

fps [float, optional] Frames per second.

online [bool, optional] Operate in online (i.e. causal) mode.

reset()
Reset to initial state.

process_offline(*activations*, ***kwargs*)
Compute the histogram of the beat intervals with a bank of resonating comb filters.

Parameters

activations [numpy array] Beat activation function.

Returns

histogram_bins [numpy array] Bins of the beat interval histogram.

histogram_delays [numpy array] Corresponding delays [frames].

process_online(*activations*, *reset=True*, ***kwargs*)
Compute the histogram of the beat intervals with a bank of resonating comb filters in online mode.

Parameters

activations [numpy float] Beat activation function.

reset [bool, optional] Reset to initial state before processing.

Returns

histogram_bins [numpy array] Bins of the tempo histogram.

histogram_delays [numpy array] Corresponding delays [frames].

class madmom.features.tempo.**ACFTempoHistogramProcessor**(*min_bpm=40.0*, *max_bpm=250.0*, *hist_buffer=10.0*, *fps=None*, *online=False*, ***kwargs*)
Create a tempo histogram with autocorrelation.

Parameters

min_bpm [float, optional] Minimum tempo to detect [bpm].

max_bpm [float, optional] Maximum tempo to detect [bpm].

hist_buffer [float] Aggregate the tempo histogram over *hist_buffer* seconds.

fps [float, optional] Frames per second.

online [bool, optional] Operate in online (i.e. causal) mode.

reset()
Reset to initial state.

process_offline(*activations*, ***kwargs*)
Compute the histogram of the beat intervals with the autocorrelation function.

Parameters

activations [numpy array] Beat activation function.

Returns

histogram_bins [numpy array] Bins of the beat interval histogram.

histogram_delays [numpy array] Corresponding delays [frames].

process_online(*activations*, *reset=True*, ***kwargs*)
Compute the histogram of the beat intervals with the autocorrelation function in online mode.

Parameters

activations [numpy float] Beat activation function.

reset [bool, optional] Reset to initial state before processing.

Returns

histogram_bins [numpy array] Bins of the tempo histogram.

histogram_delays [numpy array] Corresponding delays [frames].

class madmom.features.tempo.**DBNTempoHistogramProcessor**(*min_bpm=40.0*, *max_bpm=250.0*, *hist_buffer=10.0*, *fps=None*, *online=False*, ***kwargs*)

Create a tempo histogram with a dynamic Bayesian network (DBN).

Parameters

min_bpm [float, optional] Minimum tempo to detect [bpm].

max_bpm [float, optional] Maximum tempo to detect [bpm].

hist_buffer [float] Aggregate the tempo histogram over *hist_buffer* seconds.

fps [float, optional] Frames per second.

online [bool, optional] Operate in online (i.e. causal) mode.

reset()

Reset DBN to initial state.

process_offline(*activations*, ***kwargs*)

Compute the histogram of the beat intervals with a DBN.

Parameters

activations [numpy array] Beat activation function.

Returns

histogram_bins [numpy array] Bins of the beat interval histogram.

histogram_delays [numpy array] Corresponding delays [frames].

process_online(*activations*, *reset=True*, ***kwargs*)

Compute the histogram of the beat intervals with a DBN using the forward algorithm.

Parameters

activations [numpy float] Beat activation function.

reset [bool, optional] Reset DBN to initial state before processing.

Returns

histogram_bins [numpy array] Bins of the tempo histogram.

histogram_delays [numpy array] Corresponding delays [frames].

class madmom.features.tempo.**TempoEstimationProcessor**(*method='comb'*, *min_bpm=40.0*, *max_bpm=250.0*, *act_smooth=0.14*, *hist_smooth=9*, *fps=None*, *online=False*, *histogram_processor=None*, ***kwargs*)

Tempo Estimation Processor class.

Parameters

method [{'comb', 'acf', 'dbn'}] Method used for tempo estimation.

min_bpm [float, optional] Minimum tempo to detect [bpm].

max_bpm [float, optional] Maximum tempo to detect [bpm].

act_smooth [float, optional (default: 0.14)] Smooth the activation function over *act_smooth* seconds.

hist_smooth [int, optional (default: 7)] Smooth the tempo histogram over *hist_smooth* bins.

alpha [float, optional] Scaling factor for the comb filter.

fps [float, optional] Frames per second.

histogram_processor [*TempoHistogramProcessor*, optional] Processor used to create a tempo histogram. If 'None', a default combfilter histogram processor will be created and used.

kwargs [dict, optional] Keyword arguments passed to *CombFilterTempoHistogramProcessor* if no *histogram_processor* was given.

Examples

Create a TempoEstimationProcessor. The returned array represents the estimated tempi (given in beats per minute) and their relative strength.

```
>>> proc = TempoEstimationProcessor(fps=100)
>>> proc
<madmom.features.tempo.TempoEstimationProcessor object at 0x...>
```

Call this TempoEstimationProcessor with the beat activation function obtained by RNNBeatProcessor to estimate the tempi.

```
>>> from madmom.features.beats import RNNBeatProcessor
>>> act = RNNBeatProcessor()('tests/data/audio/sample.wav')
>>> proc(act)
array([[176.47059,   0.47469],
       [117.64706,   0.17667],
       [240.     ,   0.15371],
       [ 68.96552,   0.09864],
       [ 82.19178,   0.09629]])
```

min_bpm
Minimum tempo [bpm].

max_bpm
Maximum tempo [bpm].

intervals
Beat intervals [frames].

min_interval
Minimum beat interval [frames].

max_interval
Maximum beat interval [frames].

reset()
Reset to initial state.

process_offline(*activations*, ***kwargs*)
Detect the tempi from the (beat) activations.

Parameters

activations [numpy array] Beat activation function.

Returns

tempi [numpy array] Array with the dominant tempi [bpm] (first column) and their relative strengths (second column).

process_online(*activations*, *reset=True*, ***kwargs*)
Detect the tempi from the (beat) activations in online mode.

Parameters

activations [numpy array] Beat activation function processed frame by frame.

reset [bool, optional] Reset the TempoEstimationProcessor to its initial state before processing.

Returns

tempi [numpy array] Array with the dominant tempi [bpm] (first column) and their relative strengths (second column).

interval_histogram(*activations*, ***kwargs*)
Compute the histogram of the beat intervals.

Parameters

activations [numpy array] Beat activation function.

Returns

histogram_bins [numpy array] Bins of the beat interval histogram.

histogram_delays [numpy array] Corresponding delays [frames].

dominant_interval(*histogram*)
Extract the dominant interval of the given histogram.

Parameters

histogram [tuple] Histogram (tuple of 2 numpy arrays, the first giving the strengths of the bins and the second corresponding delay values).

Returns

interval [int] Dominant interval.

static add_arguments(*parser*, *method=None*, *min_bpm=None*, *max_bpm=None*, *act_smooth=None*, *hist_smooth=None*, *hist_buffer=None*, *alpha=None*)
Add tempo estimation related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser.

method [{‘comb’, ‘acf’, ‘dbn’}] Method used for tempo estimation.

min_bpm [float, optional] Minimum tempo to detect [bpm].

max_bpm [float, optional] Maximum tempo to detect [bpm].

act_smooth [float, optional] Smooth the activation function over *act_smooth* seconds.

hist_smooth [int, optional] Smooth the tempo histogram over *hist_smooth* bins.

hist_buffer [float, optional] Aggregate the tempo histogram over *hist_buffer* seconds.

alpha [float, optional] Scaling factor for the comb filter.

Returns

parser_group [argparse argument group] Tempo argument parser group.

Notes

Parameters are included in the group only if they are not 'None'.

CHAPTER 9

madmom.io

Input/output package.

`madmom.io.open_file(*args, **kwds)`
Context manager which yields an open file or handle with the given mode and closes it if needed afterwards.

Parameters

filename [str or file handle] File (handle) to open.

mode: {'r', 'w'} Specifies the mode in which the file is opened.

Yields

Open file (handle).

`madmom.io.load_events(*args, **kwargs)`
Load a events from a text file, one floating point number per line.

Parameters

filename [str or file handle] File to load the events from.

Returns

numpy array Events.

Notes

Comments (lines starting with '#') and additional columns are ignored, i.e. only the first column is returned.

`madmom.io.write_events(events, filename, fmt='%.3f', delimiter='\t', header=None)`
Write the events to a file, one event per line.

Parameters

events [numpy array] Events to be written to file.

filename [str or file handle] File to write the events to.

fmt [str or sequence of strs, optional] A single format (e.g. '%.3f'), a sequence of formats, or a multi-format string (e.g. '%.3f %.3f'), in which case *delimiter* is ignored.

delimiter [str, optional] String or character separating columns.

header [str, optional] String that will be written at the beginning of the file as comment.

`madmom.io.`**`load_onsets`**(**args*, ***kwargs*)

Load a events from a text file, one floating point number per line.

Parameters

filename [str or file handle] File to load the events from.

Returns

numpy array Events.

Notes

Comments (lines starting with '#') and additional columns are ignored, i.e. only the first column is returned.

`madmom.io.`**`write_onsets`**(*events*, *filename*, *fmt='%.3f'*, *delimiter='\t'*, *header=None*)

Write the events to a file, one event per line.

Parameters

events [numpy array] Events to be written to file.

filename [str or file handle] File to write the events to.

fmt [str or sequence of strs, optional] A single format (e.g. '%.3f'), a sequence of formats, or a multi-format string (e.g. '%.3f %.3f'), in which case *delimiter* is ignored.

delimiter [str, optional] String or character separating columns.

header [str, optional] String that will be written at the beginning of the file as comment.

`madmom.io.`**`load_beats`**(**args*, ***kwargs*)

Load the beats from the given file, one beat per line of format 'beat_time' ['beat_number'].

Parameters

filename [str or file handle] File to load the beats from.

downbeats [bool, optional] Load only downbeats instead of beats.

Returns

numpy array Beats.

`madmom.io.`**`write_beats`**(*beats*, *filename*, *fmt=None*, *delimiter='\t'*, *header=None*)

Write the beats to a file.

Parameters

beats [numpy array] Beats to be written to file.

filename [str or file handle] File to write the beats to.

fmt [str or sequence of strs, optional] A single format (e.g. '%.3f'), a sequence of formats (e.g. ['%.3f', '%d']), or a multi-format string (e.g. '%.3f %d'), in which case *delimiter* is ignored.

delimiter [str, optional] String or character separating columns.

header [str, optional] String that will be written at the beginning of the file as comment.

madmom.io.**load_downbeats**(*filename*)
Load the downbeats from the given file.

Parameters

filename [str or file handle] File to load the downbeats from.

Returns

numpy array Downbeats.

madmom.io.**write_downbeats**(*beats*, *filename*, *fmt=None*, *delimiter='\t'*, *header=None*)
Write the downbeats to a file.

Parameters

beats [numpy array] Beats or downbeats to be written to file.

filename [str or file handle] File to write the beats to.

fmt [str or sequence of strs, optional] A single format (e.g. '%.3f'), a sequence of formats (e.g. ['%.3f', '%d']), or a multi-format string (e.g. '%.3f %d'), in which case *delimiter* is ignored.

delimiter [str, optional] String or character separating columns.

header [str, optional] String that will be written at the beginning of the file as comment.

Notes

If *beats* contains both time and number of the beats, they are filtered to contain only the downbeats (i.e. only the times of those beats with a beat number of 1).

madmom.io.**load_notes**(**args*, ***kwargs*)
Load the notes from the given file, one note per line of format 'onset_time' 'note_number' ['duration' ['velocity']].

Parameters

filename: str or file handle File to load the notes from.

Returns

numpy array Notes.

madmom.io.**write_notes**(*notes*, *filename*, *fmt=None*, *delimiter='\t'*, *header=None*)
Write the notes to a file.

Parameters

notes [numpy array, shape (num_notes, 2)] Notes, row format 'onset_time' 'note_number' ['duration' ['velocity']].

filename [str or file handle] File to write the notes to.

fmt [str or sequence of strs, optional] A sequence of formats (e.g. ['%.3f', '%d', '%.3f', '%d']), or a multi-format string, e.g. '%.3f %d %.3f %d', in which case *delimiter* is ignored.

delimiter [str, optional] String or character separating columns.

header [str, optional] String that will be written at the beginning of the file as comment.

Returns

numpy array Notes.

`madmom.io.load_segments(filename)`

Load labelled segments from file, one segment per line. Each segment is of form <start> <end> <label>, where <start> and <end> are floating point numbers, and <label> is a string.

Parameters

filename [str or file handle] File to read the labelled segments from.

Returns

segments [numpy structured array] Structured array with columns 'start', 'end', and 'label', containing the beginning, end, and label of segments.

`madmom.io.write_segments(segments, filename, fmt=None, delimiter='\t', header=None)`

Write labelled segments to a file.

Parameters

segments [numpy structured array] Labelled segments, one per row (column definition see SEGMENT_DTYPE).

filename [str or file handle] Output filename or handle.

fmt [str or sequence of strs, optional] A sequence of formats (e.g. ['%.3f', '%.3f', '%s']), or a multi-format string (e.g. '%.3f %.3f %s'), in which case *delimiter* is ignored.

delimiter [str, optional] String or character separating columns.

header [str, optional] String that will be written at the beginning of the file as comment.

Returns

numpy structured array Labelled segments

Notes

Labelled segments are represented as numpy structured array with three named columns: 'start' contains the start position (e.g. seconds), 'end' the end position, and 'label' the segment label.

`madmom.io.load_chords(filename)`

Load labelled segments from file, one segment per line. Each segment is of form <start> <end> <label>, where <start> and <end> are floating point numbers, and <label> is a string.

Parameters

filename [str or file handle] File to read the labelled segments from.

Returns

segments [numpy structured array] Structured array with columns 'start', 'end', and 'label', containing the beginning, end, and label of segments.

`madmom.io.write_chords(segments, filename, fmt=None, delimiter='\t', header=None)`

Write labelled segments to a file.

Parameters

segments [numpy structured array] Labelled segments, one per row (column definition see SEGMENT_DTYPE).

filename [str or file handle] Output filename or handle.

fmt [str or sequence of strs, optional] A sequence of formats (e.g. ['%.3f', '%.3f', '%s']), or a multi-format string (e.g. '%.3f %.3f %s'), in which case *delimiter* is ignored.

delimiter [str, optional] String or character separating columns.

header [str, optional] String that will be written at the beginning of the file as comment.

Returns

numpy structured array Labelled segments

Notes

Labelled segments are represented as numpy structured array with three named columns: 'start' contains the start position (e.g. seconds), 'end' the end position, and 'label' the segment label.

`madmom.io.load_key` (*filename*)

Load the key from the given file.

Parameters

filename [str or file handle] File to read key information from.

Returns

str Key.

`madmom.io.write_key` (*key*, *filename*, *header=None*)

Write key string to a file.

Parameters

key [str] Key name.

filename [str or file handle] Output file.

header [str, optional] String that will be written at the beginning of the file as comment.

Returns

key [str] Key name.

`madmom.io.load_tempo` (*filename*, *split_value=1.0*, *sort=None*, *norm_strengths=None*, *max_len=None*)

Load tempo information from the given file.

Tempo information must have the following format: 'main tempo' ['secondary tempo' ['relative_strength']]

Parameters

filename [str or file handle] File to load the tempo from.

split_value [float, optional] Value to distinguish between tempi and strengths. *values* > *split_value* are interpreted as tempi [bpm], *values* <= *split_value* are interpreted as strengths.

sort [bool, deprecated] Sort the tempi by their strength.

norm_strengths [bool, deprecated] Normalize the strengths to sum 1.

max_len [int, deprecated] Return at most *max_len* tempi.

Returns

tempi [numpy array, shape (num_tempi[, 2])] Array with tempi. If no strength is parsed, a 1-dimensional array of length 'num_tempi' is returned. If strengths are given, a 2D array with tempi (first column) and their relative strengths (second column) is returned.

madmom.io.**write_tempo**(*tempi*, *filename*, *delimiter='\t'*, *header=None*, *mirex=None*)
Write the most dominant tempi and the relative strength to a file.

Parameters

tempi [numpy array] Array with the detected tempi (first column) and their strengths (second column).

filename [str or file handle] Output file.

delimiter [str, optional] String or character separating columns.

header [str, optional] String that will be written at the beginning of the file as comment.

mirex [bool, deprecated] Report the lower tempo first (as required by MIREX).

Returns

tempo_1 [float] The most dominant tempo.

tempo_2 [float] The second most dominant tempo.

strength [float] Their relative strength.

9.1 Submodules

9.1.1 madmom.io.audio

This module contains audio input/output functionality.

exception madmom.io.audio.**LoadAudioFileError**(*value=None*)
Exception to be raised whenever an audio file could not be loaded.

madmom.io.audio.**decode_to_disk**(*infile*, *fmt='f32le'*, *sample_rate=None*, *num_channels=1*, *skip=None*, *max_len=None*, *outfile=None*, *tmp_dir=None*, *tmp_suffix=None*, *cmd='ffmpeg'*)
Decode the given audio file to another file.

Parameters

infile [str] Name of the audio sound file to decode.

fmt [{'f32le', 's16le'}, optional] Format of the samples: - 'f32le' for float32, little-endian, - 's16le' for signed 16-bit int, little-endian.

sample_rate [int, optional] Sample rate to re-sample the signal to (if set) [Hz].

num_channels [int, optional] Number of channels to reduce the signal to.

skip [float, optional] Number of seconds to skip at beginning of file.

max_len [float, optional] Maximum length in seconds to decode.

outfile [str, optional] The file to decode the sound file to; if not given, a temporary file will be created.

tmp_dir [str, optional] The directory to create the temporary file in (if no *outfile* is given).

tmp_suffix [str, optional] The file suffix for the temporary file if no *outfile* is given; e.g. ".pcm" (including the dot).

cmd [{'ffmpeg', 'avconv'}, optional] Decoding command (defaults to ffmpeg, alternatively supports avconv).

Returns

outfile [str] The output file name.

madmom.io.audio.**decode_to_pipe**(*infile*, *fmt='f32le'*, *sample_rate=None*, *num_channels=1*, *skip=None*, *max_len=None*, *buf_size=-1*, *cmd='ffmpeg'*)

Decode the given audio and return a file-like object for reading the samples, as well as a process object.

Parameters

infile [str] Name of the audio sound file to decode.

fmt [{'f32le', 's16le'}, optional] Format of the samples: - 'f32le' for float32, little-endian, - 's16le' for signed 16-bit int, little-endian.

sample_rate [int, optional] Sample rate to re-sample the signal to (if set) [Hz].

num_channels [int, optional] Number of channels to reduce the signal to.

skip [float, optional] Number of seconds to skip at beginning of file.

max_len [float, optional] Maximum length in seconds to decode.

buf_size [int, optional] Size of buffer for the file-like object: - '-1' means OS default (default), - '0' means unbuffered, - '1' means line-buffered, any other value is the buffer size in bytes.

cmd [{'ffmpeg','avconv'}, optional] Decoding command (defaults to ffmpeg, alternatively supports avconv).

Returns

pipe [file-like object] File-like object for reading the decoded samples.

proc [process object] Process object for the decoding process.

Notes

To stop decoding the file, call close() on the returned file-like object, then call wait() on the returned process object.

madmom.io.audio.**decode_to_memory**(*infile*, *fmt='f32le'*, *sample_rate=None*, *num_channels=1*, *skip=None*, *max_len=None*, *cmd='ffmpeg'*)

Decode the given audio and return it as a binary string representation.

Parameters

infile [str] Name of the audio sound file to decode.

fmt [{'f32le', 's16le'}, optional] Format of the samples: - 'f32le' for float32, little-endian, - 's16le' for signed 16-bit int, little-endian.

sample_rate [int, optional] Sample rate to re-sample the signal to (if set) [Hz].

num_channels [int, optional] Number of channels to reduce the signal to.

skip [float, optional] Number of seconds to skip at beginning of file.

max_len [float, optional] Maximum length in seconds to decode.

cmd [{'ffmpeg', 'avconv'}, optional] Decoding command (defaults to ffmpeg, alternatively supports avconv).

Returns

samples [str] Binary string representation of the audio samples.

`madmom.io.audio.get_file_info` (*infile*, *cmd='ffprobe'*)

Extract and return information about audio files.

Parameters

infile [str] Name of the audio file.

cmd [{'ffprobe', 'avprobe'}, optional] Probing command (defaults to ffprobe, alternatively supports avprobe).

Returns

dict Audio file information.

`madmom.io.audio.load_ffmpeg_file` (*filename*, *sample_rate=None*, *num_channels=None*, *start=None*, *stop=None*, *dtype=None*, *cmd_decode='ffmpeg'*, *cmd_probe='ffprobe'*)

Load the audio data from the given file and return it as a numpy array.

This uses ffmpeg (or avconv) and thus supports a lot of different file formats, resampling and channel conversions. The file will be fully decoded into memory if no start and stop positions are given.

Parameters

filename [str] Name of the audio sound file to load.

sample_rate [int, optional] Sample rate to re-sample the signal to [Hz]; 'None' returns the signal in its original rate.

num_channels [int, optional] Reduce or expand the signal to *num_channels* channels; 'None' returns the signal with its original channels.

start [float, optional] Start position [seconds].

stop [float, optional] Stop position [seconds].

dtype [numpy dtype, optional] Numpy dtype to return the signal in (supports signed and unsigned 8/16/32-bit integers, and single and double precision floats, each in little or big endian). If 'None', np.int16 is used.

cmd_decode [{'ffmpeg', 'avconv'}, optional] Decoding command (defaults to ffmpeg, alternatively supports avconv).

cmd_probe [{'ffprobe', 'avprobe'}, optional] Probing command (defaults to ffprobe, alternatively supports avprobe).

Returns

signal [numpy array] Audio samples.

sample_rate [int] Sample rate of the audio samples.

`madmom.io.audio.load_wave_file` (*filename*, *sample_rate=None*, *num_channels=None*, *start=None*, *stop=None*, *dtype=None*)

Load the audio data from the given file and return it as a numpy array.

Only supports wave files, does not support re-sampling or arbitrary channel number conversions. Reads the data as a memory-mapped file with copy-on-write semantics to defer I/O costs until needed.

Parameters

filename [str] Name of the file.

sample_rate [int, optional] Desired sample rate of the signal [Hz], or 'None' to return the signal in its original rate.

num_channels [int, optional] Reduce or expand the signal to *num_channels* channels, or 'None' to return the signal with its original channels.

start [float, optional] Start position [seconds].

stop [float, optional] Stop position [seconds].

dtype [numpy data type, optional] The data is returned with the given dtype. If 'None', it is returned with its original dtype, otherwise the signal gets rescaled. Integer dtypes use the complete value range, float dtypes the range [-1, +1].

Returns

signal [numpy array] Audio signal.

sample_rate [int] Sample rate of the signal [Hz].

Notes

The *start* and *stop* positions are rounded to the closest sample; the sample corresponding to the *stop* value is not returned, thus consecutive segment starting with the previous *stop* can be concatenated to obtain the original signal without gaps or overlaps.

`madmom.io.audio.`**`write_wave_file`**(*signal*, *filename*, *sample_rate=None*)

Write the signal to disk as a .wav file.

Parameters

signal [numpy array or Signal] The signal to be written to file.

filename [str] Name of the file.

sample_rate [int, optional] Sample rate of the signal [Hz].

Returns

filename [str] Name of the file.

Notes

sample_rate can be 'None' if *signal* is a `Signal` instance. If set, the given *sample_rate* is used instead of the signal's sample rate. Must be given if *signal* is a ndarray.

`madmom.io.audio.`**`load_audio_file`**(*filename*, *sample_rate=None*, *num_channels=None*, *start=None*, *stop=None*, *dtype=None*)

Load the audio data from the given file and return it as a numpy array. This tries load_wave_file() load_ffmpeg_file() (for ffmpeg and avconv).

Parameters

filename [str or file handle] Name of the file or file handle.

sample_rate [int, optional] Desired sample rate of the signal [Hz], or 'None' to return the signal in its original rate.

num_channels: int, optional Reduce or expand the signal to *num_channels* channels, or 'None' to return the signal with its original channels.

start [float, optional] Start position [seconds].

stop [float, optional] Stop position [seconds].

dtype [numpy data type, optional] The data is returned with the given dtype. If 'None', it is returned with its original dtype, otherwise the signal gets rescaled. Integer dtypes use the complete value range, float dtypes the range [-1, +1].

Returns

signal [numpy array] Audio signal.

sample_rate [int] Sample rate of the signal [Hz].

Notes

For wave files, the *start* and *stop* positions are rounded to the closest sample; the sample corresponding to the *stop* value is not returned, thus consecutive segment starting with the previous *stop* can be concatenated to obtain the original signal without gaps or overlaps. For all other audio files, this can not be guaranteed.

9.1.2 madmom.io.midi

This module contains MIDI functionality.

`madmom.io.midi.`**`tick2second`**(*tick*, *ticks_per_beat=480*, *tempo=500000*)
Convert absolute time in ticks to seconds.

Returns absolute time in seconds for a chosen MIDI file time resolution (ticks/pulses per quarter note, also called PPQN) and tempo (microseconds per quarter note).

`madmom.io.midi.`**`second2tick`**(*second*, *ticks_per_beat=480*, *tempo=500000*)
Convert absolute time in seconds to ticks.

Returns absolute time in ticks for a chosen MIDI file time resolution (ticks/pulses per quarter note, also called PPQN) and tempo (microseconds per quarter note).

`madmom.io.midi.`**`bpm2tempo`**(*bpm*, *time_signature=(4, 4)*)
Convert BPM (beats per minute) to MIDI file tempo (microseconds per quarter note).

Depending on the chosen time signature a bar contains a different number of beats. These beats are multiples/fractions of a quarter note, thus the returned BPM depend on the time signature.

`madmom.io.midi.`**`tempo2bpm`**(*tempo*, *time_signature=(4, 4)*)
Convert MIDI file tempo (microseconds per quarter note) to BPM (beats per minute).

Depending on the chosen time signature a bar contains a different number of beats. These beats are multiples/fractions of a quarter note, thus the returned tempo depends on the time signature.

`madmom.io.midi.`**`tick2beat`**(*tick*, *ticks_per_beat=480*, *time_signature=(4, 4)*)
Convert ticks to beats.

Returns beats for a chosen MIDI file time resolution (ticks/pulses per quarter note, also called PPQN) and time signature.

`madmom.io.midi.`**`beat2tick`**(*beat*, *ticks_per_beat=480*, *time_signature=(4, 4)*)
Convert beats to ticks.

Returns ticks for a chosen MIDI file time resolution (ticks/pulses per quarter note, also called PPQN) and time signature.

`class` `madmom.io.midi.`**`MIDIFile`**(*filename=None*, *file_format=0*, *ticks_per_beat=480*, *unit='seconds'*, *timing='absolute'*, ***kwargs*)
MIDI File.

Parameters

filename [str] MIDI file name.

file_format [int, optional] MIDI file format (0, 1, 2).

ticks_per_beat [int, optional] Resolution (i.e. ticks per quarter note) of the MIDI file.

unit [str, optional] Unit of all MIDI messages, can be one of the following:

- 'ticks', 't': use native MIDI ticks as unit,
- 'seconds', 's': use seconds as unit,
- 'beats', 'b' : use beats as unit.

timing [str, optional] Timing of all MIDI messages, can be one of the following:

- 'absolute', 'abs', 'a': use absolute timing.
- 'relative', 'rel', 'r': use relative timing, i.e. delta to

previous message.

Examples

Create a MIDI file from an array with notes. The format of the note array is: 'onset time', 'pitch', 'duration', 'velocity', 'channel'. The last column can be omitted, assuming channel 0.

```
>>> notes = np.array([[0, 50, 1, 60], [0.5, 62, 0.5, 90]])
>>> m = MIDIFile.from_notes(notes)
>>> m
<madmom.io.midi.MIDIFile object at 0x...>
```

The notes can be accessed as a numpy array in various formats (default is seconds):

```
>>> m.notes
array([[ 0. , 50. ,  1. , 60. ,  0. ],
       [ 0.5, 62. ,  0.5, 90. ,  0. ]])
>>> m.unit ='ticks'
>>> m.notes
array([[  0.,   50., 960.,   60.,    0.],
       [480.,   62., 480.,   90.,    0.]])
>>> m.unit = 'seconds'
>>> m.notes
array([[ 0. , 50. ,  1. , 60. ,  0. ],
       [ 0.5, 62. ,  0.5, 90. ,  0. ]])
>>> m.unit = 'beats'
>>> m.notes
array([[ 0., 50.,  2., 60.,  0.],
       [ 1., 62.,  1., 90.,  0.]])
```

```
>>> m = MIDIFile.from_notes(notes, tempo=60)
>>> m.notes
array([[ 0. , 50. ,  1. , 60. ,  0. ],
       [ 0.5, 62. ,  0.5, 90. ,  0. ]])
>>> m.unit = 'ticks'
>>> m.notes
array([[  0.,   50., 480.,   60.,    0.],
       [240.,   62., 240.,   90.,    0.]])
>>> m.unit = 'beats'
>>> m.notes
```

(continues on next page)

(continued from previous page)

```
array([[ 0. , 50. ,  1. , 60. ,  0. ],
       [ 0.5, 62. ,  0.5, 90. ,  0. ]])
```

```
>>> m = MIDIFile.from_notes(notes, time_signature=(2, 2))
>>> m.notes
array([[ 0. , 50. ,  1. , 60. ,  0. ],
       [ 0.5, 62. ,  0.5, 90. ,  0. ]])
>>> m.unit = 'ticks'
>>> m.notes
array([[   0.,    50., 1920.,    60.,     0.],
       [ 960.,    62.,  960.,    90.,     0.]])
>>> m.unit = 'beats'
>>> m.notes
array([[ 0., 50.,  2., 60.,  0.],
       [ 1., 62.,  1., 90.,  0.]])
```

```
>>> m = MIDIFile.from_notes(notes, tempo=60, time_signature=(2, 2))
>>> m.notes
array([[ 0. , 50. ,  1. , 60. ,  0. ],
       [ 0.5, 62. ,  0.5, 90. ,  0. ]])
>>> m.unit = 'ticks'
>>> m.notes
array([[  0.,   50., 960.,   60.,    0.],
       [480.,   62., 480.,   90.,    0.]])
>>> m.unit = 'beats'
>>> m.notes
array([[ 0. , 50. ,  1. , 60. ,  0. ],
       [ 0.5, 62. ,  0.5, 90. ,  0. ]])
```

```
>>> m = MIDIFile.from_notes(notes, tempo=240, time_signature=(3, 8))
>>> m.notes
array([[ 0. , 50. ,  1. , 60. ,  0. ],
       [ 0.5, 62. ,  0.5, 90. ,  0. ]])
>>> m.unit = 'ticks'
>>> m.notes
array([[  0.,   50., 960.,   60.,    0.],
       [480.,   62., 480.,   90.,    0.]])
>>> m.unit = 'beats'
>>> m.notes
array([[ 0., 50.,  4., 60.,  0.],
       [ 2., 62.,  2., 90.,  0.]])
```

tempi

Tempi (mircoseconds per quarter note) of the MIDI file.

Returns

tempi [numpy array] Array with tempi (time, tempo).

Notes

The time will be given in the unit set by *unit*.

time_signatures

Time signatures of the MIDI file.

Returns

time_signatures [numpy array] Array with time signatures (time, numerator, denominator).

Notes

The time will be given in the unit set by *unit*.

notes
Notes of the MIDI file.

Returns

notes [numpy array] Array with notes (onset time, pitch, duration, velocity, channel).

classmethod from_notes (*notes, unit='seconds', tempo=500000, time_signature=(4, 4), ticks_per_beat=480*)
Create a MIDIFile from the given notes.

Parameters

notes [numpy array] Array with notes, one per row. The columns are defined as: (onset time, pitch, duration, velocity, [channel]).

unit [str, optional] Unit of *notes*, can be one of the following:

- 'seconds', 's': use seconds as unit,
- 'ticks', 't': use native MIDI ticks as unit,
- 'beats', 'b' : use beats as unit.

tempo [float, optional] Tempo of the MIDI track, given in bpm or microseconds per quarter note. The unit is determined automatically by the value:

- *tempo* <= 1000: bpm
- *tempo* > 1000: microseconds per quarter note

time_signature [tuple, optional] Time signature of the track, e.g. (4, 4) for 4/4.

ticks_per_beat [int, optional] Resolution (i.e. ticks per quarter note) of the MIDI file.

Returns

:class:'MIDIFile' instance *MIDIFile* instance with all notes collected in one track.

Notes

All note events (including the generated tempo and time signature events) are written into a single track (i.e. MIDI file format 0).

save (*filename*)
Save to MIDI file.

Parameters

filename [str or open file handle] The MIDI file name.

madmom.io.midi.**load_midi** (*filename*)
Load notes from a MIDI file.

Parameters

filename: str MIDI file.

Returns

numpy array Notes ('onset time' 'note number' 'duration' 'velocity' 'channel')

madmom.io.midi.**write_midi**(*notes*, *filename*, *duration=0.6*, *velocity=100*)

Write notes to a MIDI file.

Parameters

notes [numpy array, shape (num_notes, 2)] Notes, one per row (column definition see notes).

filename [str] Output MIDI file.

duration [float, optional] Note duration if not defined by *notes*.

velocity [int, optional] Note velocity if not defined by *notes*.

Returns

numpy array Notes (including note length, velocity and channel).

Notes

The note columns format must be (duration, velocity and channel optional):

'onset time' 'note number' ['duration' ['velocity' ['channel']]]

CHAPTER 10

madmom.ml

Machine learning package.

10.1 Submodules

10.1.1 madmom.ml.crf

This module contains an implementation of Conditional Random Fields (CRFs)

class madmom.ml.crf.**ConditionalRandomField**(*initial*, *final*, *bias*, *transition*, *observation*)

Implements a linear-chain Conditional Random Field using a matrix-based definition:

$$P(Y|X) = exp[E(Y,X)]/_{Y'}[E(Y',X)]$$

$$E(Y,X) =_{i=1}^{N} [y_{n-1}^T A y_n + y_n^T c + x_n^T W y_n] + y_0^T + y_N^T,$$

where Y is a sequence of labels in one-hot encoding and X are the observed features.

Parameters

initial [numpy array] Initial potential (π) of the CRF. Also defines the number of states.

final [numpy array] Potential (τ) of the last variable of the CRF.

bias [numpy array] Label bias potential (c).

transition [numpy array] Matrix defining the transition potentials (A), where the rows are the 'from' dimension, and columns the 'to' dimension.

observation [numpy array] Matrix defining the observation potentials (W), where the rows are the 'observation' dimension, and columns the 'state' dimension.

Examples

Create a CRF that emulates a simple hidden markov model. This means that the bias and final potential will be constant and thus have no effect on the predictions.

```
>>> eta = np.spacing(1)  # for numerical stability
>>> initial = np.log(np.array([0.7, 0.2, 0.1]) + eta)
>>> final = np.ones(3)
>>> bias = np.ones(3)
>>> transition = np.log(np.array([[0.6, 0.2, 0.2],
...                                [0.1, 0.7, 0.2],
...                                [0.1, 0.1, 0.8]]) + eta)
>>> observation = np.log(np.array([[0.9, 0.5, 0.1],
...                                 [0.1, 0.5, 0.1]]) + eta)
>>> crf = ConditionalRandomField(initial, final, bias,
...                              transition, observation)
>>> crf
<madmom.ml.crf.ConditionalRandomField object at 0x...>
```

We can now decode the most probable state sequence given an observation sequence. Since we are emulating a discrete HMM, the observation sequence needs to be observation ids in one-hot encoding.

The following observation sequence corresponds to "0, 0, 1, 0, 1, 1":

```
>>> obs = np.array([[1, 0], [1, 0], [0, 1], [1, 0], [0, 1], [0, 1]])
```

Now we can find the most likely state sequence:

```
>>> crf.process(obs)
array([0, 0, 1, 1, 1, 1], dtype=uint32)
```

process (*observations*, ***kwargs*)

Determine the most probable configuration of Y given the state sequence x:

$$y^* = argmax_y P(Y = y|X = x)$$

Parameters

observations [numpy array] Observations (x) to decode the most probable state sequence for.

Returns

y_star [numpy array] Most probable state sequence.

10.1.2 madmom.ml.gmm

This module contains functionality needed for fitting and scoring Gaussian Mixture Models (GMMs) (needed e.g. in madmom.features.beats).

The needed functionality is taken from sklearn.mixture.GMM which is released under the BSD license and was written by these authors:

- Ron Weiss <ronweiss@gmail.com>
- Fabian Pedregosa <fabian.pedregosa@inria.fr>
- Bertrand Thirion <bertrand.thirion@inria.fr>

This version works with sklearn v0.16 (and hopefully onwards). All commits until 0650d5502e01e6b4245ce99729fc8e7a71aacff3 are incorporated.

madmom.ml.gmm.**logsumexp** (*arr*, *axis=0*)

Computes the sum of arr assuming arr is in the log domain.

Parameters

arr [numpy array] Input data [log domain].

axis [int, optional] Axis to operate on.

Returns

numpy array log(sum(exp(arr))) while minimizing the possibility of over/underflow.

Notes

Function copied from sklearn.utils.extmath.

`madmom.ml.gmm.`**`pinvh`**(*a*, *cond=None*, *rcond=None*, *lower=True*)

Compute the (Moore-Penrose) pseudo-inverse of a hermetian matrix.

Calculate a generalized inverse of a symmetric matrix using its eigenvalue decomposition and including all 'large' eigenvalues.

Parameters

a [array, shape (N, N)] Real symmetric or complex hermetian matrix to be pseudo-inverted.

cond, rcond [float or None] Cutoff for 'small' eigenvalues. Singular values smaller than rcond * largest_eigenvalue are considered zero. If None or -1, suitable machine precision is used.

lower [boolean] Whether the pertinent array data is taken from the lower or upper triangle of *a*.

Returns

B [array, shape (N, N)]

Raises

LinAlgError If eigenvalue does not converge

Notes

Function copied from sklearn.utils.extmath.

`madmom.ml.gmm.`**`log_multivariate_normal_density`**(*x*, *means*, *covars*, *covariance_type='diag'*)

Compute the log probability under a multivariate Gaussian distribution.

Parameters

x [array_like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

means [array_like, shape (n_components, n_features)] List of n_features-dimensional mean vectors for n_components Gaussians. Each row corresponds to a single mean vector.

covars [array_like] List of n_components covariance parameters for each Gaussian. The shape depends on *covariance_type*:

- (n_components, n_features) if 'spherical',
- (n_features, n_features) if 'tied',
- (n_components, n_features) if 'diag',
- (n_components, n_features, n_features) if 'full'.

covariance_type [{'diag', 'spherical', 'tied', 'full'}] Type of the covariance parameters. Defaults to 'diag'.

Returns

lpr [array_like, shape (n_samples, n_components)] Array containing the log probabilities of each data point in *x* under each of the n_components multivariate Gaussian distributions.

class madmom.ml.gmm.**GMM**(*n_components=1*, *covariance_type='full'*)

Gaussian Mixture Model

Representation of a Gaussian mixture model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a GMM distribution.

Initializes parameters such that every mixture component has zero mean and identity covariance.

Parameters

n_components [int, optional] Number of mixture components. Defaults to 1.

covariance_type [{'diag', 'spherical', 'tied', 'full'}] String describing the type of covariance parameters to use. Defaults to 'diag'.

See also:

sklearn.mixture.GMM

Attributes

'weights_' [array, shape (n_components,)] This attribute stores the mixing weights for each mixture component.

'means_' [array, shape (n_components, n_features)] Mean parameters for each mixture component.

'covars_' [array] Covariance parameters for each mixture component. The shape depends on *covariance_type*.:

```
- (n_components, n_features)             if 'spherical',
- (n_features, n_features)               if 'tied',
- (n_components, n_features)             if 'diag',
- (n_components, n_features, n_features) if 'full'.
```

'converged_' [bool] True when convergence was reached in fit(), False otherwise.

score_samples(*x*)

Return the per-sample likelihood of the data under the model.

Compute the log probability of x under the model and return the posterior distribution (responsibilities) of each mixture component for each element of x.

Parameters

x: array_like, shape (n_samples, n_features) List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

log_prob [array_like, shape (n_samples,)] Log probabilities of each data point in *x*.

responsibilities [array_like, shape (n_samples, n_components)] Posterior probabilities of each mixture component for each observation.

score (*x*)
Compute the log probability under the model.

Parameters

x [array_like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

log_prob [array_like, shape (n_samples,)] Log probabilities of each data point in *x*.

fit (*x*, *random_state=None*, *tol=0.001*, *min_covar=0.001*, *n_iter=100*, *n_init=1*, *params='wmc'*, *init_params='wmc'*)
Estimate model parameters with the expectation-maximization algorithm.

A initialization step is performed before entering the em algorithm. If you want to avoid this step, set the keyword argument init_params to the empty string '' when creating the GMM object. Likewise, if you would like just to do an initialization, set n_iter=0.

Parameters

x [array_like, shape (n, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

random_state: RandomState or an int seed (0 by default) A random number generator instance.

min_covar [float, optional] Floor on the diagonal of the covariance matrix to prevent overfitting.

tol [float, optional] Convergence threshold. EM iterations will stop when average gain in log-likelihood is below this threshold.

n_iter [int, optional] Number of EM iterations to perform.

n_init [int, optional] Number of initializations to perform, the best results is kept.

params [str, optional] Controls which parameters are updated in the training process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars.

init_params [str, optional] Controls which parameters are updated in the initialization process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars.

10.1.3 madmom.ml.hmm

This module contains Hidden Markov Model (HMM) functionality.

Notes

If you want to change this module and use it interactively, use pyximport.

```
>>> import pyximport
>>> pyximport.install(reload_support=True,
...                   setup_args={'include_dirs': np.get_include()})
...
(None, <pyximport.pyximport.PyxImporter object at 0x...>)
```

class madmom.ml.hmm.**DiscreteObservationModel**
Simple discrete observation model that takes an observation matrix of the form (num_states x num_observations) containing P(observation | state).

Parameters

observation_probabilities [numpy array] Observation probabilities as a 2D array of shape (num_observations, num_states). Has to sum to 1 over the second axis, since it represents P(observation | state).

Examples

Assuming two states and three observation types, instantiate a discrete observation model:

```
>>> om = DiscreteObservationModel(np.array([[0.1, 0.5, 0.4],
...                                         [0.7, 0.2, 0.1]]))
>>> om
<madmom.ml.hmm.DiscreteObservationModel object at 0x...>
```

If the probabilities do not sum to 1, it throws a ValueError:

```
>>> om = DiscreteObservationModel(np.array([[0.5, 0.5, 0.5],
...                                         [0.5, 0.5, 0.5]]))
...
Traceback (most recent call last):
...
ValueError: Not a probability distribution.
```

densities (*self*, *observations*)
Densities of the observations.

Parameters

observations [numpy array] Observations.

Returns

numpy array Densities of the observations.

log_densities (*self*, *observations*)
Log densities of the observations.

Parameters

observations [numpy array] Observations.

Returns

numpy array Log densities of the observations.

madmom.ml.hmm.**HMM**
alias of *madmom.ml.hmm.HiddenMarkovModel*

class madmom.ml.hmm.**HiddenMarkovModel**
Hidden Markov Model

To search for the best path through the state space with the Viterbi algorithm, the following parameters must be defined.

Parameters

transition_model [*TransitionModel* instance] Transition model.

observation_model [*ObservationModel* instance] Observation model.

initial_distribution [numpy array, optional] Initial state distribution; if 'None' a uniform distribution is assumed.

Examples

Create a simple HMM with two states and three observation types. The initial distribution is uniform.

```
>>> tm = TransitionModel.from_dense([0, 1, 0, 1], [0, 0, 1, 1],
...                                  [0.7, 0.3, 0.6, 0.4])
>>> om = DiscreteObservationModel(np.array([[0.2, 0.3, 0.5],
...                                          [0.7, 0.1, 0.2]]))
>>> hmm = HiddenMarkovModel(tm, om)
```

Now we can decode the most probable state sequence and get the log-probability of the sequence

```
>>> seq, log_p = hmm.viterbi([0, 0, 1, 1, 0, 0, 0, 2, 2])
>>> log_p
-12.87...
>>> seq
array([1, 1, 0, 0, 1, 1, 1, 0, 0], dtype=uint32)
```

Compute the forward variables:

```
>>> hmm.forward([0, 0, 1, 1, 0, 0, 0, 2, 2])
array([[ 0.34667,  0.65333],
       [ 0.33171,  0.66829],
       [ 0.83814,  0.16186],
       [ 0.86645,  0.13355],
       [ 0.38502,  0.61498],
       [ 0.33539,  0.66461],
       [ 0.33063,  0.66937],
       [ 0.81179,  0.18821],
       [ 0.84231,  0.15769]])
```

forward(*self*, *observations*, *reset=True*)

Compute the forward variables at each time step. Instead of computing in the log domain, we normalise at each step, which is faster for the forward algorithm.

Parameters

observations [numpy array, shape (num_frames, num_densities)] Observations to compute the forward variables for.

reset [bool, optional] Reset the HMM to its inital state before computing the forward variables.

Returns

numpy array, shape (num_observations, num_states) Forward variables.

forward_generator(*self*, *observations*, *block_size=None*)

Compute the forward variables at each time step. Instead of computing in the log domain, we normalise at each step, which is faster for the forward algorithm. This function is a generator that yields the forward variables for each time step individually to save memory. The observation densities are computed block-wise to save Python calls in the inner loops.

Parameters

observations [numpy array] Observations to compute the forward variables for.

block_size [int, optional] Block size for the block-wise computation of observation densities. If 'None', all observation densities will be computed at once.

Yields

numpy array, shape (num_states,) Forward variables.

reset (*self*, *initial_distribution=None*)
Reset the HMM to its initial state.

Parameters

initial_distribution [numpy array, optional] Reset to this initial state distribution.

viterbi (*self*, *observations*)
Determine the best path with the Viterbi algorithm.

Parameters

observations [numpy array] Observations to decode the optimal path for.

Returns

path [numpy array] Best state-space path sequence.

log_prob [float] Corresponding log probability.

class madmom.ml.hmm.**ObservationModel**
Observation model class for a HMM.

The observation model is defined as a plain 1D numpy arrays *pointers* and the methods *log_densities()* and *densities()* which return 2D numpy arrays with the (log) densities of the observations.

Parameters

pointers [numpy array (num_states,)] Pointers from HMM states to the correct densities. The length of the array must be equal to the number of states of the HMM and pointing from each state to the corresponding column of the array returned by one of the *log_densities()* or *densities()* methods. The *pointers* type must be np.uint32.

See also:

ObservationModel.log_densities, *ObservationModel.densities*

densities (*self*, *observations*)
Densities (or probabilities) of the observations for each state.

This defaults to computing the exp of the *log_densities*. You can provide a special implementation to speed-up everything.

Parameters

observations [numpy array] Observations.

Returns

numpy array Densities as a 2D numpy array with the number of rows being equal to the number of observations and the columns representing the different observation log probability densities. The type must be np.float.

log_densities (*self*, *observations*)
Log densities (or probabilities) of the observations for each state.

Parameters

observations [numpy array] Observations.

Returns

numpy array Log densities as a 2D numpy array with the number of rows being equal to the number of observations and the columns representing the different observation log probability densities. The type must be np.float.

class madmom.ml.hmm.**TransitionModel**

Transition model class for a HMM.

The transition model is defined similar to a scipy compressed sparse row matrix and holds all transition probabilities from one state to an other. This allows an efficient Viterbi decoding of the HMM.

Parameters

states [numpy array] All states transitioning to state s are stored in: states[pointers[s]:pointers[s+1]]

pointers [numpy array] Pointers for the *states* array for state s.

probabilities [numpy array] The corresponding transition are stored in: probabilities[pointers[s]:pointers[s+1]].

See also:

`scipy.sparse.csr_matrix`

Notes

This class should be either used for loading saved transition models or being sub-classed to define a specific transition model.

Examples

Create a simple transition model with two states using a list of transitions and their probabilities

```
>>> tm = TransitionModel.from_dense([0, 1, 0, 1], [0, 0, 1, 1],
...                                 [0.8, 0.2, 0.3, 0.7])
>>> tm
<madmom.ml.hmm.TransitionModel object at 0x...>
```

TransitionModel.from_dense will check if the supplied probabilties for each state sum to 1 (and thus represent a correct probability distribution)

```
>>> tm = TransitionModel.from_dense([0, 1], [1, 0], [0.5, 1.0])
...
Traceback (most recent call last):
...
ValueError: Not a probability distribution.
```

classmethod from_dense(*cls*, *states*, *prev_states*, *probabilities*)

Instantiate a TransitionModel from dense transitions.

Parameters

states [numpy array, shape (num_transitions,)] Array with states (i.e. destination states).

prev_states [numpy array, shape (num_transitions,)] Array with previous states (i.e. origination states).

probabilities [numpy array, shape (num_transitions,)] Transition probabilities.

Returns

:class:‘TransitionModel‘ instance TransitionModel instance.

log_probabilities
Transition log probabilities.

make_dense (*states*, *pointers*, *probabilities*)
Return a dense representation of sparse transitions.

Parameters

states [numpy array] All states transitioning to state s are returned in: states[pointers[s]:pointers[s+1]]

pointers [numpy array] Pointers for the *states* array for state s.

probabilities [numpy array] The corresponding transition are returned in: probabilities[pointers[s]:pointers[s+1]].

Returns

states [numpy array, shape (num_transitions,)] Array with states (i.e. destination states).

prev_states [numpy array, shape (num_transitions,)] Array with previous states (i.e. origination states).

probabilities [numpy array, shape (num_transitions,)] Transition probabilities.

See also:

TransitionModel

Notes

Three 1D numpy arrays of same length must be given. The indices correspond to each other, i.e. the first entry of all three arrays define the transition from the state defined prev_states[0] to that defined in states[0] with the probability defined in probabilities[0].

make_sparse (*states*, *prev_states*, *probabilities*)
Return a sparse representation of dense transitions.

This method removes all duplicate states and thus allows an efficient Viterbi decoding of the HMM.

Parameters

states [numpy array, shape (num_transitions,)] Array with states (i.e. destination states).

prev_states [numpy array, shape (num_transitions,)] Array with previous states (i.e. origination states).

probabilities [numpy array, shape (num_transitions,)] Transition probabilities.

Returns

states [numpy array] All states transitioning to state s are returned in: states[pointers[s]:pointers[s+1]]

pointers [numpy array] Pointers for the *states* array for state s.

probabilities [numpy array] The corresponding transition are returned in: probabilities[pointers[s]:pointers[s+1]].

See also:

TransitionModel

Notes

Three 1D numpy arrays of same length must be given. The indices correspond to each other, i.e. the first entry of all three arrays define the transition from the state defined prev_states[0] to that defined in states[0] with the probability defined in probabilities[0].

num_states
Number of states.

num_transitions
Number of transitions.

10.1.4 madmom.ml.nn

Neural Network package.

madmom.ml.nn.**average_predictions**(*predictions*)
Returns the average of all predictions.

Parameters

predictions [list] Predictions (i.e. NN activation functions).

Returns

numpy array Averaged prediction.

class madmom.ml.nn.**NeuralNetwork**(*layers*)
Neural Network class.

Parameters

layers [list] Layers of the Neural Network.

Examples

Create a NeuralNetwork from the given layers.

```
>>> from madmom.ml.nn.layers import FeedForwardLayer
>>> from madmom.ml.nn.activations import tanh, sigmoid
>>> l1_weights = np.array([[0.5, -1., -0.3 , -0.2]])
>>> l1_bias = np.array([0.05, 0., 0.8, -0.5])
>>> l1 = FeedForwardLayer(l1_weights, l1_bias, activation_fn=tanh)
>>> l2_weights = np.array([-1, 0.9, -0.2 , 0.4])
>>> l2_bias = np.array([0.5])
>>> l2 = FeedForwardLayer(l2_weights, l2_bias, activation_fn=sigmoid)
>>> nn = NeuralNetwork([l1, l2])
>>> nn
<madmom.ml.nn.NeuralNetwork object at 0x...>
>>> nn(np.array([[0], [0.5], [1], [0], [1], [2], [0]]))
...
array([0.53305, 0.36903, 0.265 , 0.53305, 0.265 , 0.18612, 0.53305])
```

process(*data*, *reset=True*, ***kwargs*)
Process the given data with the neural network.

Parameters

data [numpy array, shape (num_frames, num_inputs)] Activate the network with this data.

reset [bool, optional] Reset the network to its initial state before activating it.

Returns

numpy array, shape (num_frames, num_outputs) Network predictions for this data.

reset()
Reset the neural network to its initial state.

class madmom.ml.nn.**NeuralNetworkEnsemble**(*networks*, *ensemble_fn=<function average_predictions>*, *num_threads=None*, ***kwargs*)

Neural Network ensemble class.

Parameters

networks [list] List of the Neural Networks.

ensemble_fn [function or callable, optional] Ensemble function to be applied to the predictions of the neural network ensemble (default: average predictions).

num_threads [int, optional] Number of parallel working threads.

Notes

If *ensemble_fn* is set to 'None', the predictions are returned as a list with the same length as the number of networks given.

Examples

Create a NeuralNetworkEnsemble from the networks. Instead of supplying the neural networks as parameter, they can also be loaded from file:

```
>>> from madmom.models import ONSETS_BRNN_PP
>>> nn = NeuralNetworkEnsemble.load(ONSETS_BRNN_PP)
>>> nn
<madmom.ml.nn.NeuralNetworkEnsemble object at 0x...>
>>> nn(np.array([[0], [0.5], [1], [0], [1], [2], [0]]))
...
array([0.00116, 0.00213, 0.01428, 0.00729, 0.0088 , 0.21965, 0.00532])
```

classmethod load(*nn_files*, ***kwargs*)
Instantiate a new Neural Network ensemble from a list of files.

Parameters

nn_files [list] List of neural network model file names.

kwargs [dict, optional] Keyword arguments passed to NeuralNetworkEnsemble.

Returns

NeuralNetworkEnsemble NeuralNetworkEnsemble instance.

static add_arguments(*parser*, *nn_files*)
Add neural network options to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

nn_files [list] Neural network model files.

Returns

argparse argument group Neural network argument parser group.

madmom.ml.nn.layers

This module contains neural network layers for the ml.nn module.

`class madmom.ml.nn.layers.AverageLayer`

Average layer.

Parameters

axis [None or int or tuple of ints, optional] Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

dtype [data-type, optional] Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one.

`activate()`

Activate the layer.

Parameters

data [numpy array] Activate with this data.

Returns

numpy array Averaged data.

`class madmom.ml.nn.layers.BatchNormLayer`

Batch normalization layer with activation function. The previous layer is usually linear with no bias - the BatchNormLayer's beta parameter replaces it. See *[1]* for a detailed understanding of the parameters.

Parameters

beta [numpy array] Values for the *beta* parameter. Must be broadcastable to the incoming shape.

gamma [numpy array] Values for the *gamma* parameter. Must be broadcastable to the incoming shape.

mean [numpy array] Mean values of incoming data. Must be broadcastable to the incoming shape.

inv_std [numpy array] Inverse standard deviation of incoming data. Must be broadcastable to the incoming shape.

activation_fn [numpy ufunc] Activation function.

References

[1]

`activate()`

Activate the layer.

Parameters

data [numpy array] Activate with this data.

Returns

numpy array Normalized data.

class madmom.ml.nn.layers.**BidirectionalLayer**

Bidirectional network layer.

Parameters

fwd_layer [Layer instance] Forward layer.

bwd_layer [Layer instance] Backward layer.

activate()

Activate the layer.

After activating the *fwd_layer* with the data and the *bwd_layer* with the data in reverse temporal order, the two activations are stacked and returned.

Parameters

data [numpy array, shape (num_frames, num_inputs)] Activate with this data.

Returns

numpy array, shape (num_frames, num_hiddens) Activations for this data.

class madmom.ml.nn.layers.**Cell**

Cell as used by LSTM layers.

Parameters

weights [numpy array, shape (num_inputs, num_hiddens)] Weights.

bias [scalar or numpy array, shape (num_hiddens,)] Bias.

recurrent_weights [numpy array, shape (num_hiddens, num_hiddens)] Recurrent weights.

activation_fn [numpy ufunc, optional] Activation function.

Notes

A Cell is the same as a Gate except it misses peephole connections and has a *tanh* activation function. It should not be used directly, only inside an LSTMLayer.

class madmom.ml.nn.layers.**ConvolutionalLayer**

Convolutional network layer.

Parameters

weights [numpy array, shape (num_feature_maps, num_channels, <kernel>)] Weights.

bias [scalar or numpy array, shape (num_filters,)] Bias.

stride [int, optional] Stride of the convolution.

pad [{'valid', 'same', 'full'}] A string indicating the size of the output:

- **full** The output is the full discrete linear convolution of the inputs.
- **valid** The output consists only of those elements that do not rely on the zero-padding.
- **same** The output is the same size as the input, centered with respect to the 'full' output.

activation_fn [numpy ufunc] Activation function.

activate()
Activate the layer.

Parameters

data [numpy array (num_frames, num_bins, num_channels)] Activate with this data.

Returns

numpy array Activations for this data.

class madmom.ml.nn.layers.**FeedForwardLayer**
Feed-forward network layer.

Parameters

weights [numpy array, shape (num_inputs, num_hiddens)] Weights.

bias [scalar or numpy array, shape (num_hiddens,)] Bias.

activation_fn [numpy ufunc] Activation function.

activate()
Activate the layer.

Parameters

data [numpy array, shape (num_frames, num_inputs)] Activate with this data.

Returns

numpy array, shape (num_frames, num_hiddens) Activations for this data.

class madmom.ml.nn.layers.**GRUCell**
Cell as used by GRU layers proposed in *[1]*. The cell output is computed by

$$h = tanh(W_{xh} * x_t + W_{hh} * h_{t-1} + b).$$

Parameters

weights [numpy array, shape (num_inputs, num_hiddens)] Weights of the connections between inputs and cell.

bias [scalar or numpy array, shape (num_hiddens,)] Bias.

recurrent_weights [numpy array, shape (num_hiddens, num_hiddens)] Weights of the connections between cell and cell output of the previous time step.

activation_fn [numpy ufunc, optional] Activation function.

Notes

There are two formulations of the GRUCell in the literature. Here, we adopted the (slightly older) one proposed in *[1]*, which is also implemented in the Lasagne toolbox.

It should not be used directly, only inside a GRULayer.

References

[1]

activate()
Activate the cell with the given input, previous output and reset gate.

Parameters

data [numpy array, shape (num_inputs,)] Input data for the cell.

prev [numpy array, shape (num_hiddens,)] Output of the previous time step.

reset_gate [numpy array, shape (num_hiddens,)] Activation of the reset gate.

Returns

numpy array, shape (num_hiddens,) Activations of the cell for this data.

class madmom.ml.nn.layers.**GRULayer**

Recurrent network layer with Gated Recurrent Units (GRU) as proposed in *[1]*.

Parameters

reset_gate [*Gate*] Reset gate.

update_gate [*Gate*] Update gate.

cell [*GRUCell*] GRU cell.

init [numpy array, shape (num_hiddens,), optional] Initial state of hidden units.

Notes

There are two formulations of the GRUCell in the literature. Here, we adopted the (slightly older) one proposed in [1], which is also implemented in the Lasagne toolbox.

References

[1]

activate()

Activate the GRU layer.

Parameters

data [numpy array, shape (num_frames, num_inputs)] Activate with this data.

reset [bool, optional] Reset the layer to its initial state before activating it.

Returns

numpy array, shape (num_frames, num_hiddens) Activations for this data.

reset()

Reset the layer to its initial state.

Parameters

init [numpy array, shape (num_hiddens,), optional] Reset the hidden units to this initial state.

class madmom.ml.nn.layers.**Gate**

Gate as used by LSTM layers.

Parameters

weights [numpy array, shape (num_inputs, num_hiddens)] Weights.

bias [scalar or numpy array, shape (num_hiddens,)] Bias.

recurrent_weights [numpy array, shape (num_hiddens, num_hiddens)] Recurrent weights.

peephole_weights [numpy array, shape (num_hiddens,), optional] Peephole weights.

activation_fn [numpy ufunc, optional] Activation function.

Notes

Gate should not be used directly, only inside an LSTMLayer.

activate()

Activate the gate with the given data, state (if peephole connections are used) and the previous output (if recurrent connections are used).

Parameters

data [scalar or numpy array, shape (num_hiddens,)] Input data for the cell.

prev [scalar or numpy array, shape (num_hiddens,)] Output data of the previous time step.

state [scalar or numpy array, shape (num_hiddens,)] State data of the {current | previous} time step.

Returns

numpy array, shape (num_hiddens,) Activations of the gate for this data.

class madmom.ml.nn.layers.**LSTMLayer**

Recurrent network layer with Long Short-Term Memory units.

Parameters

input_gate [*Gate*] Input gate.

forget_gate [*Gate*] Forget gate.

cell [*Cell*] Cell (i.e. a Gate without peephole connections).

output_gate [*Gate*] Output gate.

activation_fn [numpy ufunc, optional] Activation function.

init [numpy array, shape (num_hiddens,), optional] Initial state of the layer.

cell_init [numpy array, shape (num_hiddens,), optional] Initial state of the cell.

activate()

Activate the LSTM layer.

Parameters

data [numpy array, shape (num_frames, num_inputs)] Activate with this data.

reset [bool, optional] Reset the layer to its initial state before activating it.

Returns

numpy array, shape (num_frames, num_hiddens) Activations for this data.

reset()

Reset the layer to its initial state.

Parameters

init [numpy array, shape (num_hiddens,), optional] Reset the hidden units to this initial state.

cell_init [numpy array, shape (num_hiddens,), optional] Reset the cells to this initial state.

class madmom.ml.nn.layers.**Layer**
Generic callable network layer.

activate()
Activate the layer.

Parameters

data [numpy array] Activate with this data.

Returns

numpy array Activations for this data.

reset()
Reset the layer to its initial state.

class madmom.ml.nn.layers.**MaxPoolLayer**
2D max-pooling network layer.

Parameters

size [tuple] The size of the pooling region in each dimension.

stride [tuple, optional] The strides between successive pooling regions in each dimension. If None *stride* = *size*.

activate()
Activate the layer.

Parameters

data [numpy array] Activate with this data.

Returns

numpy array Max pooled data.

class madmom.ml.nn.layers.**PadLayer**
Padding layer that pads the input with a constant value.

Parameters

width [int] Width of the padding (only one value for all dimensions)

axes [iterable] Indices of axes to be padded

value [float] Value to be used for padding.

activate()
Activate the layer.

Parameters

data [numpy array] Activate with this data.

Returns

numpy array Padded data.

class madmom.ml.nn.layers.**RecurrentLayer**
Recurrent network layer.

Parameters

weights [numpy array, shape (num_inputs, num_hiddens)] Weights.

bias [scalar or numpy array, shape (num_hiddens,)] Bias.

recurrent_weights [numpy array, shape (num_hiddens, num_hiddens)] Recurrent weights.

activation_fn [numpy ufunc] Activation function.

init [numpy array, shape (num_hiddens,), optional] Initial state of hidden units.

activate()
Activate the layer.

Parameters

data [numpy array, shape (num_frames, num_inputs)] Activate with this data.

reset [bool, optional] Reset the layer to its initial state before activating it.

Returns

numpy array, shape (num_frames, num_hiddens) Activations for this data.

reset()
Reset the layer to its initial state.

Parameters

init [numpy array, shape (num_hiddens,), optional] Reset the hidden units to this initial state.

class madmom.ml.nn.layers.**ReshapeLayer**
Reshape Layer.

Parameters

newshape [int or tuple of ints] The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

order [{'C', 'F', 'A'}, optional] Index order or the input. See np.reshape for a detailed description.

activate()
Activate the layer.

Parameters

data [numpy array] Activate with this data.

Returns

numpy array Reshaped data.

class madmom.ml.nn.layers.**StrideLayer**
Stride network layer.

Parameters

block_size [int] Re-arrange (stride) the data in blocks of given size.

activate()
Activate the layer.

Parameters

data [numpy array] Activate with this data.

Returns

numpy array Strided data.

class madmom.ml.nn.layers.**TransposeLayer**

Transpose layer.

Parameters

axes [list of ints, optional] By default, reverse the dimensions of the input, otherwise permute the axes of the input according to the values given.

activate()

Activate the layer.

Parameters

data [numpy array] Activate with this data.

Returns

numpy array Transposed data.

madmom.ml.nn.layers.**convolve**

Convolve the data with the kernel in 'valid' mode, i.e. only where kernel and data fully overlaps.

Parameters

data [numpy array] Data to be convolved.

kernel [numpy array] Convolution kernel

Returns

numpy array Convolved data

madmom.ml.nn.activations

This module contains neural network activation functions for the ml.nn module.

madmom.ml.nn.activations.**linear**(*x*, *out=None*)

Linear function.

Parameters

x [numpy array] Input data.

out [numpy array, optional] Array to hold the output data.

Returns

numpy array Unaltered input data.

madmom.ml.nn.activations.**tanh**(*x*, *out=None*)

Hyperbolic tangent function.

Parameters

x [numpy array] Input data.

out [numpy array, optional] Array to hold the output data.

Returns

numpy array Hyperbolic tangent of input data.

madmom.ml.nn.activations.**sigmoid**(*x*, *out=None*)

Logistic sigmoid function.

Parameters

x [numpy array] Input data.

out [numpy array, optional] Array to hold the output data.

Returns

numpy array Logistic sigmoid of input data.

`madmom.ml.nn.activations.`**`relu`**(*x*, *out=None*)

Rectified linear (unit) transfer function.

Parameters

x [numpy array] Input data.

out [numpy array, optional] Array to hold the output data.

Returns

numpy array Rectified linear of input data.

`madmom.ml.nn.activations.`**`elu`**(*x*, *out=None*)

Exponential linear (unit) transfer function.

Parameters

x [numpy array] Input data.

out [numpy array, optional] Array to hold the output data.

Returns

numpy array Exponential linear of input data

References

[1]

`madmom.ml.nn.activations.`**`softmax`**(*x*, *out=None*)

Softmax transfer function.

Parameters

x [numpy array] Input data.

out [numpy array, optional] Array to hold the output data.

Returns

numpy array Softmax of input data.

CHAPTER 11

madmom.utils

Utility package.

madmom.utils.**suppress_warnings**(*function*)
Decorate the given function to suppress any warnings.

Parameters

function [function] Function to be decorated.

Returns

decorated function Decorated function.

madmom.utils.**filter_files**(*files*, *suffix*)
Filter the list to contain only files matching the given *suffix*.

Parameters

files [list] List of files to be filtered.

suffix [str] Return only files matching this suffix.

Returns

list List of files.

madmom.utils.**search_path**(*path*, *recursion_depth=0*)
Returns a list of files in a directory (recursively).

Parameters

path [str or list] Directory to be searched.

recursion_depth [int, optional] Recursively search sub-directories up to this depth.

Returns

list List of files.

madmom.utils.**search_files**(*files*, *suffix=None*, *recursion_depth=0*)
Returns the files matching the given *suffix*.

Parameters

files [str or list] File, path or a list thereof to be searched / filtered.

suffix [str, optional] Return only files matching this suffix.

recursion_depth [int, optional] Recursively search sub-directories up to this depth.

Returns

list List of files.

Notes

The list of returned files is sorted.

`madmom.utils.`**`strip_suffix`**(*filename*, *suffix=None*)

Strip off the suffix of the given filename or string.

Parameters

filename [str] Filename or string to strip.

suffix [str, optional] Suffix to be stripped off (e.g. '.txt' including the dot).

Returns

str Filename or string without suffix.

`madmom.utils.`**`match_file`**(*filename*, *match_list*, *suffix=None*, *match_suffix=None*, *match_exactly=True*)

Match a filename or string against a list of other filenames or strings.

Parameters

filename [str] Filename or string to match.

match_list [list] Match to this list of filenames or strings.

suffix [str, optional] Suffix of *filename* to be ignored.

match_suffix [str, optional] Match only files from *match_list* with this suffix.

match_exactly [bool, optional] Matches must be exact, i.e. have the same base name.

Returns

list List of matched files.

Notes

Asterisks "*" can be used to match any string or suffix.

`madmom.utils.`**`combine_events`**(*events*, *delta*, *combine='mean'*)

Combine all events within a certain range.

Parameters

events [list or numpy array] Events to be combined.

delta [float] Combination delta. All events within this *delta* are combined.

combine [{'mean', 'left', 'right'}] How to combine two adjacent events:

- 'mean': replace by the mean of the two events

- 'left': replace by the left of the two events
- 'right': replace by the right of the two events

Returns

numpy array Combined events.

`madmom.utils.quantize_events`(*events*, *fps*, *length=None*, *shift=None*)

Quantize the events with the given resolution.

Parameters

events [list or numpy array] Events to be quantized.

fps [float] Quantize with *fps* frames per second.

length [int, optional] Length of the returned array. If 'None', the length will be set according to the latest event.

shift [float, optional] Shift the events by *shift* seconds before quantization.

Returns

numpy array Quantized events.

`madmom.utils.quantize_notes`(*notes*, *fps*, *length=None*, *num_pitches=None*, *velocity=None*)

Quantize the notes with the given resolution.

Create a sparse 2D array with rows corresponding to points in time (according to *fps* and *length*), and columns to note pitches (according to *num_pitches*). The values of the array correspond to the velocity of a sounding note at a given point in time (based on the note pitch, onset, duration and velocity). If no values for *length* and *num_pitches* are given, they are inferred from *notes*.

Parameters

notes [2D numpy array] Notes to be quantized. Expected columns: 'note_time' 'note_number' ['duration' ['velocity']] If *notes* contains no 'duration' column, only the frame of the onset will be set. If *notes* has no velocity column, a velocity of 1 is assumed.

fps [float] Quantize with *fps* frames per second.

length [int, optional] Length of the returned array. If 'None', the length will be set according to the latest sounding note.

num_pitches [int, optional] Number of pitches of the returned array. If 'None', the number of pitches will be based on the highest pitch in the *notes* array.

velocity [float, optional] Use this velocity for all quantized notes. If set, the last column of *notes* (if present) will be ignored.

Returns

numpy array Quantized notes.

`madmom.utils.expand_notes`(*notes*, *duration=0.6*, *velocity=100*)

Expand notes to include duration and velocity.

The given duration and velocity is only used if they are not set already.

Parameters

notes [numpy array, shape (num_notes, 2)] Notes, one per row. Expected columns: 'note_time' 'note_number' ['duration' ['velocity']]

duration [float, optional] Note duration if not defined by *notes*.

velocity [int, optional] Note velocity if not defined by *notes*.

Returns

notes [numpy array, shape (num_notes, 2)] Notes (including note duration and velocity).

class madmom.utils.**OverrideDefaultListAction**(*sep=None*, **args*, ***kwargs*)

An argparse action that works similarly to the regular 'append' action. The default value is deleted when a new value is specified. The 'append' action would append the new value to the default.

Parameters

sep [str, optional] Separator to be used if multiple values should be parsed from a list.

madmom.utils.**segment_axis**(*signal*, *frame_size*, *hop_size*, *axis=None*, *end='cut'*, *end_value=0*)

Generate a new array that chops the given array along the given axis into (overlapping) frames.

Parameters

signal [numpy array] Signal.

frame_size [int] Size of each frame [samples].

hop_size [int] Hop size between adjacent frames [samples].

axis [int, optional] Axis to operate on; if 'None', operate on the flattened array.

end [{'cut', 'wrap', 'pad'}, optional] What to do with the last frame, if the array is not evenly divisible into pieces; possible values:

- 'cut' simply discard the extra values,
- 'wrap' copy values from the beginning of the array,
- 'pad' pad with a constant value.

end_value [float, optional] Value used to pad if *end* is 'pad'.

Returns

numpy array, shape (num_frames, frame_size) Array with overlapping frames

Notes

The array is not copied unless necessary (either because it is unevenly strided and being flattened or because end is set to 'pad' or 'wrap').

The returned array is always of type np.ndarray.

Examples

```
>>> segment_axis(np.arange(10), 4, 2)
array([[0, 1, 2, 3],
       [2, 3, 4, 5],
       [4, 5, 6, 7],
       [6, 7, 8, 9]])
```

11.1 Submodules

11.1.1 madmom.utils.midi

This module contains MIDI functionality, but is deprecated as of version 0.16. Please use madmom.io.midi instead. This module will be removed in version 0.18.

Almost all code is taken from Giles Hall's python-midi package: https://github.com/vishnubob/python-midi

It combines the complete package in a single file, to make it easier to distribute. Most notable changes are *MIDITrack* and *MIDIFile* classes which handle all data i/o and provide a interface which allows to read/display all notes as simple numpy arrays. Also, the EventRegistry is handled differently.

The last merged commit is 3053fefe.

Since then the following commits have been added functionality-wise:

- 0964c0b (prevent multiple tick conversions)
- c43bf37 (add pitch and value properties to AfterTouchEvent)
- 40111c6 (add 0x08 MetaEvent: ProgramNameEvent)
- 43de818 (handle unknown MIDI meta events gracefully)

Additionally, the module has been updated to work with Python3.

madmom.utils.midi.**byte2int**(*byte*)

Convert a byte-character to an integer.

madmom.utils.midi.**read_variable_length**(*data*)

Read a variable length variable from the given data.

Parameters

data [bytearray] Data of variable length.

Returns

length [int] Length in bytes.

madmom.utils.midi.**write_variable_length**(*value*)

Write a variable length variable.

Parameters

value [bytearray] Value to be encoded as a variable of variable length.

Returns

bytearray Variable with variable length.

class madmom.utils.midi.**EventRegistry**

Class for registering Events.

Event classes should be registered manually by calling EventRegistry.register_event(EventClass) after the class definition.

Normal events are registered in the *events* dictionary and use the event's *status_msg* as a key; meta events are registered in the *meta_events* dictionary and use their *meta_command* as key.

classmethod register_event(*event*)

Registers an event in the registry.

Parameters

event [*Event* instance] Event to be registered.

class madmom.utils.midi.**Event**(***kwargs*)

Generic MIDI Event.

class madmom.utils.midi.**ChannelEvent**(***kwargs*)

Event with a channel number.

class madmom.utils.midi.**NoteEvent**(***kwargs*)

NoteEvent is a special subclass of Event that is not meant to be used as a concrete class. It defines the generalities of NoteOn and NoteOff events.

pitch

Pitch of the note event.

velocity

Velocity of the note event.

class madmom.utils.midi.**NoteOnEvent**(***kwargs*)

Note On Event.

class madmom.utils.midi.**NoteOffEvent**(***kwargs*)

Note Off Event.

class madmom.utils.midi.**AfterTouchEvent**(***kwargs*)

After Touch Event.

pitch

Pitch of the after touch event.

value

Value of the after touch event.

class madmom.utils.midi.**ControlChangeEvent**(***kwargs*)

Control Change Event.

control

Control ID.

value

Value of the controller.

class madmom.utils.midi.**ProgramChangeEvent**(***kwargs*)

Program Change Event.

value

Value of the Program Change Event.

class madmom.utils.midi.**ChannelAfterTouchEvent**(***kwargs*)
Channel After Touch Event.

value
Value of the Channel After Touch Event.

class madmom.utils.midi.**PitchWheelEvent**(***kwargs*)
Pitch Wheel Event.

pitch
Pitch of the Pitch Wheel Event.

class madmom.utils.midi.**SysExEvent**(***kwargs*)
System Exclusive Event.

class madmom.utils.midi.**MetaEvent**(***kwargs*)
MetaEvent is a special subclass of Event that is not meant to be used as a concrete class. It defines a subset of Events known as the Meta events.

class madmom.utils.midi.**MetaEventWithText**(***kwargs*)
Meta Event With Text.

class madmom.utils.midi.**SequenceNumberMetaEvent**(***kwargs*)
Sequence Number Meta Event.

class madmom.utils.midi.**TextMetaEvent**(***kwargs*)
Text Meta Event.

class madmom.utils.midi.**CopyrightMetaEvent**(***kwargs*)
Copyright Meta Event.

class madmom.utils.midi.**TrackNameEvent**(***kwargs*)
Track Name Event.

class madmom.utils.midi.**InstrumentNameEvent**(***kwargs*)
Instrument Name Event.

class madmom.utils.midi.**LyricsEvent**(***kwargs*)
Lyrics Event.

class madmom.utils.midi.**MarkerEvent**(***kwargs*)
Marker Event.

class madmom.utils.midi.**CuePointEvent**(***kwargs*)
Cue Point Event.

class madmom.utils.midi.**ProgramNameEvent**(***kwargs*)
Program Name Event.

class madmom.utils.midi.**UnknownMetaEvent**(***kwargs*)
Unknown Meta Event.

Parameters

meta_command [int] Value of the meta command.

class madmom.utils.midi.**ChannelPrefixEvent**(***kwargs*)
Channel Prefix Event.

class madmom.utils.midi.**PortEvent**(***kwargs*)
Port Event.

class madmom.utils.midi.**TrackLoopEvent**(***kwargs*)
Track Loop Event.

class madmom.utils.midi.**EndOfTrackEvent**(***kwargs*)
End Of Track Event.

class madmom.utils.midi.**SetTempoEvent**(***kwargs*)
Set Tempo Event.

microseconds_per_quarter_note
Microseconds per quarter note.

class madmom.utils.midi.**SmpteOffsetEvent**(***kwargs*)
SMPTE Offset Event.

class madmom.utils.midi.**TimeSignatureEvent**(***kwargs*)
Time Signature Event.

numerator
Numerator of the time signature.

denominator
Denominator of the time signature.

metronome
Metronome.

thirty_seconds
Thirty-seconds of the time signature.

class madmom.utils.midi.**KeySignatureEvent**(***kwargs*)
Key Signature Event.

alternatives
Alternatives of the key signature.

minor
Major / minor.

class madmom.utils.midi.**SequencerSpecificEvent**(***kwargs*)
Sequencer Specific Event.

class madmom.utils.midi.**MIDITrack**(*events=None*)
MIDI Track.

Parameters

events [list] MIDI events.

Notes

All events are stored with timing information in absolute ticks. The events must be sorted. Consider using *from_notes()* method.

Examples

Create a MIDI track from a list of events. Please note that the events must be sorted.

```
>>> e1 = NoteOnEvent(tick=100, pitch=50, velocity=60)
>>> e2 = NoteOffEvent(tick=300, pitch=50)
>>> e3 = NoteOnEvent(tick=200, pitch=62, velocity=90)
>>> e4 = NoteOffEvent(tick=600, pitch=62)
```

(continues on next page)

(continued from previous page)

```
>>> t = MIDITrack(sorted([e1, e2, e3, e4]))
>>> t
<madmom.utils.midi.MIDITrack object at 0x...>
>>> t.events
[<madmom.utils.midi.NoteOnEvent object at 0x...>,
 <madmom.utils.midi.NoteOnEvent object at 0x...>,
 <madmom.utils.midi.NoteOffEvent object at 0x...>,
 <madmom.utils.midi.NoteOffEvent object at 0x...>]
```

It can also be created from an array containing the notes. The *from_notes* method also takes care of creating tempo and time signature events.

```
>>> notes = np.array([[0.1, 50, 0.3, 60], [0.2, 62, 0.4, 90]])
>>> t = MIDITrack.from_notes(notes)
>>> t
<madmom.utils.midi.MIDITrack object at 0x...>
>>> t.events
[<madmom.utils.midi.SetTempoEvent object at 0x...>,
 <madmom.utils.midi.TimeSignatureEvent object at 0...>,
 <madmom.utils.midi.NoteOnEvent object at 0x...>,
 <madmom.utils.midi.NoteOnEvent object at 0x...>,
 <madmom.utils.midi.NoteOffEvent object at 0x...>,
 <madmom.utils.midi.NoteOffEvent object at 0x...>]
```

data_stream
MIDI data stream representation of the track.

classmethod from_stream(*midi_stream*)
Create a MIDI track by reading the data from a stream.

Parameters

midi_stream [open file handle] MIDI file stream (e.g. open MIDI file handle)

Returns

:class:'MIDITrack' instance *MIDITrack* instance

classmethod from_notes(*notes*, *tempo=120*, *time_signature=(4, 4)*, *resolution=480*)
Create a MIDI track from the given notes.

Parameters

notes [numpy array] Array with the notes, one per row. The columns must be: (onset time, pitch, duration, velocity, [channel]).

tempo [float, optional] Tempo of the MIDI track, given in beats per minute (bpm).

time_signature [tuple, optional] Time signature of the track, e.g. (4, 4) for 4/4.

resolution [int] Resolution (i.e. ticks per quarter note) of the MIDI track.

Returns

:class:'MIDITrack' instance *MIDITrack* instance

Notes

All events including the generated tempo and time signature events is included in the returned track (i.e. as defined in MIDI format 0).

class madmom.utils.midi.**MIDIFile**(*tracks=None*, *resolution=480*, *file_format=0*)

MIDI File.

Parameters

tracks [list] List of *MIDITrack* instances.

resolution [int, optional] Resolution (i.e. microseconds per quarter note).

file_format [int, optional] Format of the MIDI file.

Notes

Writing a MIDI file assumes a tempo of 120 beats per minute (bpm) and a 4/4 time signature and writes all events into a single track (i.e. MIDI format 0).

Examples

Create a MIDI file from an array with notes. The format of the note array is: 'onset time', 'pitch', 'duration', 'velocity', 'channel'. The last column can be omitted, assuming channel 0.

```
>>> notes = np.array([[0, 50, 1, 60], [0.5, 62, 0.5, 90]])
>>> m = MIDIFile.from_notes(notes)
>>> m
<madmom.utils.midi.MIDIFile object at 0x...>
```

The notes can be accessed as a numpy array in various formats (default is seconds):

```
>>> m.notes()
array([[ 0. , 50. ,  1. , 60. ,  0. ],
       [ 0.5, 62. ,  0.5, 90. ,  0. ]])
>>> m.notes(unit='ticks')
array([[  0.,   50., 960.,  60.,    0.],
       [480.,   62., 480.,  90.,    0.]])
>>> m.notes(unit='beats')
array([[ 0., 50.,  2., 60.,  0.],
       [ 1., 62.,  1., 90.,  0.]])
```

```
>>> m = MIDIFile.from_notes(notes, tempo=60)
>>> m.notes(unit='ticks')
array([[  0.,   50., 480.,  60.,    0.],
       [240.,   62., 240.,  90.,    0.]])
>>> m.notes(unit='beats')
array([[ 0. , 50. ,  1. , 60. ,  0. ],
       [ 0.5, 62. ,  0.5, 90. ,  0. ]])
```

```
>>> m = MIDIFile.from_notes(notes, tempo=60, time_signature=(2, 2))
>>> m.notes(unit='ticks')
array([[  0.,   50., 960.,  60.,    0.],
       [480.,   62., 480.,  90.,    0.]])
>>> m.notes(unit='beats')
array([[ 0. , 50. ,  1. , 60. ,  0. ],
       [ 0.5, 62. ,  0.5, 90. ,  0. ]])
```

```
>>> m = MIDIFile.from_notes(notes, tempo=240, time_signature=(3, 8))
>>> m.notes(unit='ticks')
array([[  0.,   50., 960.,   60.,    0.],
       [480.,   62., 480.,   90.,    0.]])
>>> m.notes(unit='beats')
array([[ 0., 50.,  4., 60.,  0.],
       [ 2., 62.,  2., 90.,  0.]])
```

ticks_per_quarter_note
Number of ticks per quarter note.

tempi(*suppress_warnings=False*)
Tempi of the MIDI file.

Returns

tempi [numpy array] Array with tempi (tick, seconds per tick, cumulative time).

time_signatures(*suppress_warnings=False*)
Time signatures of the MIDI file.

Returns

time_signatures [numpy array] Array with time signatures (tick, numerator, denominator).

notes(*unit='s'*)
Notes of the MIDI file.

Parameters

unit [{'s', 'seconds', 'b', 'beats', 't', 'ticks'}] Time unit for notes, seconds ('s') beats ('b') or ticks ('t')

Returns

notes [numpy array] Array with notes (onset time, pitch, duration, velocity, channel).

data_stream
MIDI data stream representation of the MIDI file.

write(*midi_file*)
Write a MIDI file.

Parameters

midi_file [str] The MIDI file name.

classmethod from_file(*midi_file*)
Create a MIDI file instance from a .mid file.

Parameters

midi_file [str] Name of the .mid file to load.

Returns

:class:'MIDIFile' instance *MIDIFile* instance

classmethod from_notes(*notes*, *tempo=120*, *time_signature=(4, 4)*, *resolution=480*)
Create a MIDIFile from the given notes.

Parameters

notes [numpy array] Array with the notes, one per row. The columns must be: (onset time, pitch, duration, velocity, [channel]).

tempo [float, optional] Tempo of the MIDI track, given in beats per minute (bpm).

time_signature [tuple, optional] Time signature of the track, e.g. (4, 4) for 4/4.

resolution [int] Resolution (i.e. ticks per quarter note) of the MIDI track.

Returns

:class:'MIDIFile' instance *MIDIFile* instance with all notes collected in one track.

Notes

All note events (including the generated tempo and time signature events) are written into a single track (i.e. MIDI file format 0).

static add_arguments(*parser*, *length=None*, *velocity=None*, *channel=None*)

Add MIDI related arguments to an existing parser object.

Parameters

parser [argparse parser instance] Existing argparse parser object.

length [float, optional] Default length of the notes [seconds].

velocity [int, optional] Default velocity of the notes.

channel [int, optional] Default channel of the notes.

Returns

argparse argument group MIDI argument parser group object.

madmom.utils.midi.**process_notes**(*data*, *output=None*)

This is a simple processing function. It either loads the notes from a MIDI file and or writes the notes to a file.

The behaviour depends on the presence of the *output* argument, if 'None' is given, the notes are read, otherwise the notes are written to file.

Parameters

data [str or numpy array] MIDI file to be loaded (if *output* is 'None') / notes to be written.

output [str, optional] Output file name. If set, the notes given by *data* are written.

Returns

notes [numpy array] Notes read/written.

CHAPTER 12

madmom.processors

This module contains all processor related functionality.

12.1 Notes

All features should be implemented as classes which inherit from Processor (or provide a XYZProcessor(Processor) variant). This way, multiple Processor objects can be chained/combined to achieve the wanted functionality.

class madmom.processors.**Processor**

Abstract base class for processing data.

classmethod load(*infile*)

Instantiate a new Processor from a file.

This method un-pickles a saved Processor object. Subclasses should overwrite this method with a better performing solution if speed is an issue.

Parameters

infile [str or file handle] Pickled processor.

Returns

:class:‘Processor‘ instance Processor.

dump(*outfile*)

Save the Processor to a file.

This method pickles a Processor object and saves it. Subclasses should overwrite this method with a better performing solution if speed is an issue.

Parameters

outfile [str or file handle] Output file for pickling the processor.

process(*data*, ***kwargs*)

Process the data.

This method must be implemented by the derived class and should process the given data and return the processed output.

Parameters

data [depends on the implementation of subclass] Data to be processed.

kwargs [dict, optional] Keyword arguments for processing.

Returns

depends on the implementation of subclass Processed data.

class madmom.processors.**OnlineProcessor**(*online=False*)

Abstract base class for processing data in online mode.

Derived classes must implement the following methods:

- process_online(): process the data in online mode,
- process_offline(): process the data in offline mode.

process(*data*, ***kwargs*)

Process the data either in online or offline mode.

Parameters

data [depends on the implementation of subclass] Data to be processed.

kwargs [dict, optional] Keyword arguments for processing.

Returns

depends on the implementation of subclass Processed data.

Notes

This method is used to pass the data to either *process_online* or *process_offline*, depending on the *online* setting of the processor.

process_online(*data*, *reset=True*, ***kwargs*)

Process the data in online mode.

This method must be implemented by the derived class and should process the given data frame by frame and return the processed output.

Parameters

data [depends on the implementation of subclass] Data to be processed.

reset [bool, optional] Reset the processor to its initial state before processing.

kwargs [dict, optional] Keyword arguments for processing.

Returns

depends on the implementation of subclass Processed data.

process_offline(*data*, ***kwargs*)

Process the data in offline mode.

This method must be implemented by the derived class and should process the given data and return the processed output.

Parameters

data [depends on the implementation of subclass] Data to be processed.

kwargs [dict, optional] Keyword arguments for processing.

Returns

depends on the implementation of subclass Processed data.

reset()

Reset the OnlineProcessor.

This method must be implemented by the derived class and should reset the processor to its initial state.

class madmom.processors.**OutputProcessor**

Class for processing data and/or feeding it into some sort of output.

process(*data*, *output*, ***kwargs*)

Processes the data and feed it to the output.

This method must be implemented by the derived class and should process the given data and return the processed output.

Parameters

data [depends on the implementation of subclass] Data to be processed (e.g. written to file).

output [str or file handle] Output file name or file handle.

kwargs [dict, optional] Keyword arguments for processing.

Returns

depends on the implementation of subclass Processed data.

class madmom.processors.**SequentialProcessor**(*processors*)

Processor class for sequential processing of data.

Parameters

processors [list] Processor instances to be processed sequentially.

Notes

If the *processors* list contains lists or tuples, these get wrapped as a SequentialProcessor itself.

insert(*index*, *processor*)

Insert a Processor at the given processing chain position.

Parameters

index [int] Position inside the processing chain.

processor [*Processor*] Processor to insert.

append(*other*)

Append another Processor to the processing chain.

Parameters

other [*Processor*] Processor to append to the processing chain.

extend(*other*)

Extend the processing chain with a list of Processors.

Parameters

other [list] Processors to be appended to the processing chain.

process (*data*, ***kwargs*)
Process the data sequentially with the defined processing chain.

Parameters

data [depends on the first processor of the processing chain] Data to be processed.

kwargs [dict, optional] Keyword arguments for processing.

Returns

depends on the last processor of the processing chain Processed data.

class madmom.processors.**ParallelProcessor** (*processors*, *num_threads=None*)
Processor class for parallel processing of data.

Parameters

processors [list] Processor instances to be processed in parallel.

num_threads [int, optional] Number of parallel working threads.

Notes

If the *processors* list contains lists or tuples, these get wrapped as a `SequentialProcessor`.

process (*data*, ***kwargs*)
Process the data in parallel.

Parameters

data [depends on the processors] Data to be processed.

kwargs [dict, optional] Keyword arguments for processing.

Returns

list Processed data.

class madmom.processors.**IOProcessor** (*in_processor*, *out_processor=None*)
Input/Output Processor which processes the input data with the input processor and pipes everything into the given output processor.

All Processors defined in the input chain are sequentially called with the 'data' argument only. The output Processor is the only one ever called with two arguments ('data', 'output').

Parameters

in_processor [`Processor`, function, tuple or list] Input processor. Can be a `Processor` (or subclass thereof like `SequentialProcessor` or `ParallelProcessor`), a function accepting a single argument ('data'). If a tuple or list is given, it is wrapped as a `SequentialProcessor`.

out_processor [`OutputProcessor`, function, tuple or list] OutputProcessor or function accepting two arguments ('data', 'output'). If a tuple or list is given, it is wrapped in an `IOProcessor` itself with the last element regarded as the *out_processor* and all others as *in_processor*.

process (*data*, *output=None*, ***kwargs*)
Processes the data with the input processor and pipe everything into the output processor, which also pipes it to *output*.

Parameters

data [depends on the input processors] Data to be processed.

output: str or file handle Output file (handle).

kwargs [dict, optional] Keyword arguments for processing.

Returns

depends on the output processors Processed data.

madmom.processors.**process_single**(*processor*, *infile*, *outfile*, ***kwargs*)
Process a single file with the given Processor.

Parameters

processor [*Processor* instance] Processor to be processed.

infile [str or file handle] Input file (handle).

outfile [str or file handle] Output file (handle).

madmom.processors.**process_batch**(*processor*, *files*, *output_dir=None*, *output_suffix=None*, *strip_ext=True*, *num_workers=4*, *shuffle=False*, ***kwargs*)
Process a list of files with the given Processor in batch mode.

Parameters

processor [*Processor* instance] Processor to be processed.

files [list] Input file(s) (handles).

output_dir [str, optional] Output directory.

output_suffix [str, optional] Output suffix (e.g. '.txt' including the dot).

strip_ext [bool, optional] Strip off the extension from the input files.

num_workers [int, optional] Number of parallel working threads.

shuffle [bool, optional] Shuffle the *files* before distributing them to the working threads

Notes

Either *output_dir* and/or *output_suffix* must be set. If *strip_ext* is True, the extension of the input file names is stripped off before the *output_suffix* is appended to the input file names.

Use *shuffle* if you experience out of memory errors (can occur for certain methods with high memory consumptions if consecutive files are rather long).

class madmom.processors.**BufferProcessor**(*buffer_size=None*, *init=None*, *init_value=0*)
Buffer for processors which need context to do their processing.

Parameters

buffer_size [int or tuple] Size of the buffer (time steps, [additional dimensions]).

init [numpy array, optional] Init the buffer with this array.

init_value [float, optional] If only *buffer_size* is given but no *init*, use this value to initialise the buffer.

Notes

If *buffer_size* (or the first item thereof in case of tuple) is 1, only the un-buffered current value is returned.

If context is needed, *buffer_size* must be set to >1. E.g. SpectrogramDifference needs a context of two frames to be able to compute the difference between two consecutive frames.

reset (*init=None*)
Reset BufferProcessor to its initial state.

Parameters

init [numpy array, shape (num_hiddens,), optional] Reset BufferProcessor to this initial state.

process (*data*, ***kwargs*)
Buffer the data.

Parameters

data [numpy array or subclass thereof] Data to be buffered.

Returns

numpy array or subclass thereof Data with buffered context.

buffer (*data*, ***kwargs*)
Buffer the data.

Parameters

data [numpy array or subclass thereof] Data to be buffered.

Returns

numpy array or subclass thereof Data with buffered context.

madmom.processors.**process_online** (*processor*, *infile*, *outfile*, ***kwargs*)
Process a file or audio stream with the given Processor.

Parameters

processor [`Processor` instance] Processor to be processed.

infile [str or file handle, optional] Input file (handle). If none is given, the stream present at the system's audio inpup is used. Additional keyword arguments can be used to influence the frame size and hop size.

outfile [str or file handle] Output file (handle).

kwargs [dict, optional] Keyword arguments passed to `audio.signal.Stream` if *in_stream* is 'None'.

Notes

Right now there is no way to determine if a processor is online-capable or not. Thus, calling any processor with this function may not produce the results expected.

madmom.processors.**pickle_processor** (*processor*, *outfile*, ***kwargs*)
Pickle the Processor to a file.

Parameters

processor [`Processor` instance] Processor to be pickled.

outfile [str or file handle] Output file (handle) where to pickle it.

`madmom.processors.io_arguments`(*parser*, *output_suffix='.txt'*, *pickle=True*, *online=False*)

Add input / output related arguments to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

output_suffix [str, optional] Suffix appended to the output files.

pickle [bool, optional] Add a 'pickle' sub-parser to the parser.

online [bool, optional] Add a 'online' sub-parser to the parser.

CHAPTER 13

madmom.evaluation

Evaluation package.

`madmom.evaluation.`**`find_closest_matches`**(*detections*, *annotations*)

Find the closest annotation for each detection.

Parameters

detections [list or numpy array] Detected events.

annotations [list or numpy array] Annotated events.

Returns

indices [numpy array] Indices of the closest matches.

Notes

The sequences must be ordered.

`madmom.evaluation.`**`calc_errors`**(*detections*, *annotations*, *matches=None*)

Errors of the detections to the closest annotations.

Parameters

detections [list or numpy array] Detected events.

annotations [list or numpy array] Annotated events.

matches [list or numpy array] Indices of the closest events.

Returns

errors [numpy array] Errors.

Notes

The sequences must be ordered. To speed up the calculation, a list of pre-computed indices of the closest matches can be used.

madmom.evaluation.**calc_absolute_errors**(*detections*, *annotations*, *matches=None*)
Absolute errors of the detections to the closest annotations.

Parameters

detections [list or numpy array] Detected events.

annotations [list or numpy array] Annotated events.

matches [list or numpy array] Indices of the closest events.

Returns

errors [numpy array] Absolute errors.

Notes

The sequences must be ordered. To speed up the calculation, a list of pre-computed indices of the closest matches can be used.

madmom.evaluation.**calc_relative_errors**(*detections*, *annotations*, *matches=None*)
Relative errors of the detections to the closest annotations.

Parameters

detections [list or numpy array] Detected events.

annotations [list or numpy array] Annotated events.

matches [list or numpy array] Indices of the closest events.

Returns

errors [numpy array] Relative errors.

Notes

The sequences must be ordered. To speed up the calculation, a list of pre-computed indices of the closest matches can be used.

class madmom.evaluation.**EvaluationMixin**
Evaluation mixin class.

This class has a *name* attribute which is used for display purposes and defaults to 'None'.

METRIC_NAMES is a list of tuples, containing the attribute's name and the corresponding label, e.g.:

The attributes defined in *METRIC_NAMES* will be provided as an ordered dictionary as the *metrics* property unless the subclass overwrites the property.

FLOAT_FORMAT is used to format floats.

metrics
Metrics as a dictionary.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

Notes

This is a fallback method formatting the *metrics* dictionary in a human readable way. Classes inheriting from this mixin class should provide a method better suitable.

class madmom.evaluation.**SimpleEvaluation**(*num_tp=0*, *num_fp=0*, *num_tn=0*, *num_fn=0*, *name=None*, ***kwargs*)

Simple Precision, Recall, F-measure and Accuracy evaluation based on the numbers of true/false positive/negative detections.

Parameters

num_tp [int] Number of true positive detections.

num_fp [int] Number of false positive detections.

num_tn [int] Number of true negative detections.

num_fn [int] Number of false negative detections.

name [str] Name to be displayed.

Notes

This class is only suitable for a 1-class evaluation problem.

num_tp
Number of true positive detections.

num_fp
Number of false positive detections.

num_tn
Number of true negative detections.

num_fn
Number of false negative detections.

num_annotations
Number of annotations.

precision
Precision.

recall
Recall.

fmeasure
F-measure.

accuracy
Accuracy.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

class madmom.evaluation.**Evaluation**(*tp=None*, *fp=None*, *tn=None*, *fn=None*, ***kwargs*)

Evaluation class for measuring Precision, Recall and F-measure based on numpy arrays or lists with true/false positive/negative detections.

Parameters

tp [list or numpy array] True positive detections.

fp [list or numpy array] False positive detections.

tn [list or numpy array] True negative detections.

fn [list or numpy array] False negative detections.

name [str] Name to be displayed.

num_tp

Number of true positive detections.

num_fp

Number of false positive detections.

num_tn

Number of true negative detections.

num_fn

Number of false negative detections.

class madmom.evaluation.**MultiClassEvaluation**(*tp=None*, *fp=None*, *tn=None*, *fn=None*, ***kwargs*)

Evaluation class for measuring Precision, Recall and F-measure based on 2D numpy arrays with true/false positive/negative detections.

Parameters

tp [list of tuples or numpy array, shape (num_tp, 2)] True positive detections.

fp [list of tuples or numpy array, shape (num_fp, 2)] False positive detections.

tn [list of tuples or numpy array, shape (num_tn, 2)] True negative detections.

fn [list of tuples or numpy array, shape (num_fn, 2)] False negative detections.

name [str] Name to be displayed.

Notes

The second item of the tuples or the second column of the arrays denote the class the detection belongs to.

tostring(*verbose=False*, ***kwargs*)

Format the evaluation metrics as a human readable string.

Parameters

verbose [bool] Add evaluation for individual classes.

Returns

str Evaluation metrics formatted as a human readable string.

class madmom.evaluation.**SumEvaluation**(*eval_objects*, *name=None*)

Simple class for summing evaluations.

Parameters

eval_objects [list] Evaluation objects.

name [str] Name to be displayed.

num_tp
Number of true positive detections.

num_fp
Number of false positive detections.

num_tn
Number of true negative detections.

num_fn
Number of false negative detections.

num_annotations
Number of annotations.

class madmom.evaluation.**MeanEvaluation**(*eval_objects*, *name=None*, ***kwargs*)
Simple class for averaging evaluation.

Parameters

eval_objects [list] Evaluation objects.

name [str] Name to be displayed.

num_tp
Number of true positive detections.

num_fp
Number of false positive detections.

num_tn
Number of true negative detections.

num_fn
Number of false negative detections.

num_annotations
Number of annotations.

precision
Precision.

recall
Recall.

fmeasure
F-measure.

accuracy
Accuracy.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

madmom.evaluation.**tostring**(*eval_objects*, ***kwargs*)
Format the given evaluation objects as human readable strings.

Parameters

eval_objects [list] Evaluation objects.

Returns

str Evaluation metrics formatted as a human readable string.

madmom.evaluation.**tocsv**(*eval_objects*, *metric_names=None*, *float_format='{:.3f}'*, ***kwargs*)

Format the given evaluation objects as a CSV table.

Parameters

eval_objects [list] Evaluation objects.

metric_names [list of tuples, optional] List of tuples defining the name of the property corresponding to the metric, and the metric label e.g. ('fp', 'False Positives').

float_format [str, optional] How to format the metrics.

Returns

str CSV table representation of the evaluation objects.

Notes

If no *metric_names* are given, they will be extracted from the first evaluation object.

madmom.evaluation.**totex**(*eval_objects*, *metric_names=None*, *float_format='{:.3f}'*, ***kwargs*)

Format the given evaluation objects as a LaTeX table.

Parameters

eval_objects [list] Evaluation objects.

metric_names [list of tuples, optional] List of tuples defining the name of the property corresponding to the metric, and the metric label e.g. ('fp', 'False Positives').

float_format [str, optional] How to format the metrics.

Returns

str LaTeX table representation of the evaluation objects.

Notes

If no *metric_names* are given, they will be extracted from the first evaluation object.

madmom.evaluation.**evaluation_io**(*parser*, *ann_suffix*, *det_suffix*, *ann_dir=None*, *det_dir=None*)

Add evaluation input/output and formatting related arguments to an existing parser object.

Parameters

parser [argparse parser instance] Existing argparse parser object.

ann_suffix [str] Suffix of the annotation files.

det_suffix [str] Suffix of the detection files.

ann_dir [str, optional] Use only annotations from this folder (and sub-folders).

det_dir [str, optional] Use only detections from this folder (and sub-folders).

Returns

io_group [argparse argument group] Evaluation input / output argument group.

formatter_group [argparse argument group] Evaluation formatter argument group.

13.1 Submodules

13.1.1 madmom.evaluation.alignment

13.1.2 madmom.evaluation.beats

This module contains beat evaluation functionality.

The measures are described in *[R84778d1bb8cd-1]*, a Matlab implementation exists here: http://code.soundsoftware.ac.uk/projects/beat-evaluation/repository

Notes

Please note that this is a complete re-implementation, which took some other design decisions. For example, the beat detections and annotations are not quantised before being evaluated with F-measure, P-score and other metrics. Hence these evaluation functions DO NOT report the exact same results/scores. This approach was chosen, because it is simpler and produces more accurate results.

References

exception `madmom.evaluation.beats.`**`BeatIntervalError`**(*value=None*)
Exception to be raised whenever an interval cannot be computed.

`madmom.evaluation.beats.`**`array`**(*metric*)
Decorate metric to convert annotations and detections to numpy arrays.

`madmom.evaluation.beats.`**`score_10`**(*metric*)
Metric to decorate

`madmom.evaluation.beats.`**`score_1100`**(*metric*)
Metric to decorate

`madmom.evaluation.beats.`**`variations`**(*sequence, offbeat=False, double=False, half=False, triple=False, third=False*)
Create variations of the given beat sequence.

Parameters

sequence [numpy array] Beat sequence.

offbeat [bool, optional] Create an offbeat sequence.

double [bool, optional] Create a double tempo sequence.

half [bool, optional] Create half tempo sequences (includes offbeat version).

triple [bool, optional] Create triple tempo sequence.

third [bool, optional] Create third tempo sequences (includes offbeat versions).

Returns

list Beat sequence variations.

`madmom.evaluation.beats.`**`calc_intervals`**(*events, fwd=False*)
Calculate the intervals of all events to the previous/next event.

Parameters

events [numpy array] Beat sequence.

fwd [bool, optional] Calculate the intervals towards the next event (instead of previous).

Returns

numpy array Beat intervals.

Notes

The sequence must be ordered. The first (last) interval will be set to the same value as the second (second to last) interval (when used in *fwd* mode).

`madmom.evaluation.beats.`**`find_closest_intervals`**(*detections*, *annotations*, *matches=None*)

Find the closest annotated interval to each beat detection.

Parameters

detections [list or numpy array] Detected beats.

annotations [list or numpy array] Annotated beats.

matches [list or numpy array] Indices of the closest beats.

Returns

numpy array Closest annotated beat intervals.

Notes

The sequences must be ordered. To speed up the calculation, a list of pre-computed indices of the closest matches can be used.

The function does NOT test if each detection has a surrounding interval, it always returns the closest interval.

`madmom.evaluation.beats.`**`find_longest_continuous_segment`**(*sequence_indices*)

ind the longest consecutive segment in the given sequence.

Parameters

sequence_indices [numpy array] Indices of the beats

Returns

length [int] Length of the longest consecutive segment.

start [int] Start position of the longest continuous segment.

`madmom.evaluation.beats.`**`calc_relative_errors`**(*detections*, *annotations*, **args*, ***kwargs*)

Errors of the detections relative to the closest annotated interval.

Parameters

detections [list or numpy array] Detected beats.

annotations [list or numpy array] Annotated beats.

matches [list or numpy array] Indices of the closest beats.

Returns

numpy array Errors relative to the closest annotated beat interval.

Notes

The sequences must be ordered! To speed up the calculation, a list of pre-computed indices of the closest matches can be used.

`madmom.evaluation.beats.`**`pscore`**(*detections*, *annotations*, **args*, ***kwargs*)

Calculate the P-score accuracy for the given detections and annotations.

The P-score is determined by taking the sum of the cross-correlation between two impulse trains, representing the detections and annotations allowing for a tolerance of 20% of the median annotated interval *[1]*.

Parameters

detections [list or numpy array] Detected beats.

annotations [list or numpy array] Annotated beats.

tolerance [float, optional] Evaluation tolerance (fraction of the median beat interval).

Returns

pscore [float] P-Score.

Notes

Contrary to the original implementation which samples the two impulse trains with 100Hz, we do not quantise the annotations and detections but rather count all detections falling withing the defined tolerance window.

References

[1]

`madmom.evaluation.beats.`**`cemgil`**(*detections*, *annotations*, **args*, ***kwargs*)

Calculate the Cemgil accuracy for the given detections and annotations.

Parameters

detections [list or numpy array] Detected beats.

annotations [list or numpy array] Annotated beats.

sigma [float, optional] Sigma for Gaussian error function.

Returns

cemgil [float] Cemgil beat tracking accuracy.

References

[1]

`madmom.evaluation.beats.`**`goto`**(*detections*, *annotations*, **args*, ***kwargs*)

Calculate the Goto and Muraoka accuracy for the given detections and annotations.

Parameters

detections [list or numpy array] Detected beats.

annotations [list or numpy array] Annotated beats.

threshold [float, optional] Threshold.

sigma [float, optional] Allowed std. dev. of the errors in the longest segment.

mu [float, optional] Allowed mean. of the errors in the longest segment.

Returns

goto [float] Goto beat tracking accuracy.

Notes

[1] requires that the first correct beat detection must occur within the first 3/4 of the excerpt. In order to be able to deal with audio with varying tempo, this was altered that the length of the longest continuously tracked segment must be at least 1/4 of the total length *[2]*.

References

[1], *[2]*

`madmom.evaluation.beats.`**`cml`**(*detections*, *annotations*, **args*, ***kwargs*)

Calculate the cmlc and cmlt scores for the given detections and annotations.

Parameters

detections [list or numpy array] Detected beats.

annotations [list or numpy array] Annotated beats.

phase_tolerance [float, optional] Allowed phase tolerance.

tempo_tolerance [float, optional] Allowed tempo tolerance.

Returns

cmlc [float] Longest continuous segment of correct detections normalized by the maximum length of both sequences (detection and annotations).

cmlt [float] Same as cmlc, but no continuity required.

References

[1], *[2]*

`madmom.evaluation.beats.`**`continuity`**(*detections*, *annotations*, **args*, ***kwargs*)

Calculate the cmlc, cmlt, amlc and amlt scores for the given detections and annotations.

Parameters

detections [list or numpy array] Detected beats.

annotations [list or numpy array] Annotated beats.

phase_tolerance [float, optional] Allowed phase tolerance.

tempo_tolerance [float, optional] Allowed tempo tolerance.

offbeat [bool, optional] Include offbeat variation.

double [bool, optional] Include double and half tempo variations (and offbeat thereof).

triple [bool, optional] Include triple and third tempo variations (and offbeats thereof).

Returns

cmlc [float] Tracking accuracy, continuity at the correct metrical level required.

cmlt [float] Same as cmlc, continuity at the correct metrical level not required.

amlc [float] Same as cmlc, alternate metrical levels allowed.

amlt [float] Same as cmlt, alternate metrical levels allowed.

See also:

`cml()`

`madmom.evaluation.beats.`**`information_gain`**(*detections*, *annotations*, **args*, ***kwargs*)

Calculate information gain for the given detections and annotations.

Parameters

detections [list or numpy array] Detected beats.

annotations [list or numpy array] Annotated beats.

num_bins [int, optional] Number of bins for the beat error histogram.

Returns

information_gain [float] Information gain.

error_histogram [numpy array] Error histogram.

References

[1]

`madmom.evaluation.beats.`**`tostring`**(*obj*)

Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

`class` `madmom.evaluation.beats.`**`BeatEvaluation`**(*detections*, *annotations*, *fmeasure_window=0.07*, *pscore_tolerance=0.2*, *cemgil_sigma=0.04*, *goto_threshold=0.175*, *goto_sigma=0.1*, *goto_mu=0.1*, *continuity_phase_tolerance=0.175*, *continuity_tempo_tolerance=0.175*, *information_gain_bins=40*, *offbeat=True*, *double=True*, *triple=True*, *skip=0*, *downbeats=False*, ***kwargs*)

Beat evaluation class.

Parameters

detections [str, list or numpy array] Detected beats.

annotations [str, list or numpy array] Annotated ground truth beats.

fmeasure_window [float, optional] F-measure evaluation window [seconds]

pscore_tolerance [float, optional] P-Score tolerance [fraction of the median beat interval].

cemgil_sigma [float, optional] Sigma of Gaussian window for Cemgil accuracy.

goto_threshold [float, optional] Threshold for Goto error.

goto_sigma [float, optional] Sigma for Goto error.

goto_mu [float, optional] Mu for Goto error.

continuity_phase_tolerance [float, optional] Continuity phase tolerance.

continuity_tempo_tolerance [float, optional] Ccontinuity tempo tolerance.

information_gain_bins [int, optional] Number of bins for for the information gain beat error histogram.

offbeat [bool, optional] Include offbeat variation.

double [bool, optional] Include double and half tempo variations (and offbeat thereof).

triple [bool, optional] Include triple and third tempo variations (and offbeats thereof).

skip [float, optional] Skip the first *skip* seconds for evaluation.

downbeats [bool, optional] Evaluate downbeats instead of beats.

Notes

The *offbeat*, *double*, and *triple* variations of the beat sequences are used only for AMLc/AMLt.

global_information_gain
Global information gain.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

class madmom.evaluation.beats.**BeatMeanEvaluation**(*eval_objects*, *name=None*, ***kwargs*)
Class for averaging beat evaluation scores.

fmeasure
F-measure.

pscore
P-score.

cemgil
Cemgil accuracy.

goto
Goto accuracy.

cmlc
CMLc.

cmlt
CMLt.

amlc
AMLc.

amlt
AMLt.

information_gain
Information gain.

error_histogram
Error histogram.

global_information_gain
Global information gain.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

madmom.evaluation.beats.**add_parser**(*parser*)
Add a beat evaluation sub-parser to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

Returns

sub_parser [argparse sub-parser instance] Beat evaluation sub-parser.

parser_group [argparse argument group] Beat evaluation argument group.

13.1.3 madmom.evaluation.chords

This module contains chord evaluation functionality.

It provides the evaluation measures used for the MIREX ACE task, and tries to follow *[Raff97c8dd6dc-1]* and *[Raff97c8dd6dc-2]* as closely as possible.

Notes

This implementation tries to follow the references and their implementation (e.g., https://github.com/jpauwels/MusOOEvaluator for *[Raff97c8dd6dc-2]*). However, there are some known (and possibly some unknown) differences. If you find one not listed in the following, please file an issue:

- Detected chord segments are adjusted to fit the length of the annotations. In particular, this means that, if necessary, filler segments of 'no chord' are added at beginnings and ends. This can result in different segmentation scores compared to the original implementation.

References

madmom.evaluation.chords.**encode**(*chord_labels*)
Encodes chord labels to numeric interval representations.

Parameters

chord_labels [numpy structured array] Chord segments in *madmom.io.SEGMENT_DTYPE* format

Returns

encoded_chords [numpy structured array] Chords in *CHORD_ANN_DTYPE* format

madmom.evaluation.chords.**chords**(*labels*)
Transform a list of chord labels into an array of internal numeric representations.

Parameters

labels [list] List of chord labels (str).

Returns

chords [numpy.array] Structured array with columns 'root', 'bass', and 'intervals', containing a numeric representation of chords (*CHORD_DTYPE*).

madmom.evaluation.chords.**chord**(*label*)

Transform a chord label into the internal numeric represenation of (root, bass, intervals array) as defined by *CHORD_DTYPE*.

Parameters

label [str] Chord label.

Returns

chord [tuple] Numeric representation of the chord: (root, bass, intervals array).

madmom.evaluation.chords.**modify**(*base_pitch*, *modifier*)

Modify a pitch class in integer representation by a given modifier string.

A modifier string can be any sequence of 'b' (one semitone down) and '#' (one semitone up).

Parameters

base_pitch [int] Pitch class as integer.

modifier [str] String of modifiers ('b' or '#').

Returns

modified_pitch [int] Modified root note.

madmom.evaluation.chords.**pitch**(*pitch_str*)

Convert a string representation of a pitch class (consisting of root note and modifiers) to an integer representation.

Parameters

pitch_str [str] String representation of a pitch class.

Returns

pitch [int] Integer representation of a pitch class.

madmom.evaluation.chords.**interval**(*interval_str*)

Convert a string representation of a musical interval into a pitch class (e.g. a minor seventh 'b7' into 10, because it is 10 semitones above its base note).

Parameters

interval_str [str] Musical interval.

Returns

pitch_class [int] Number of semitones to base note of interval.

madmom.evaluation.chords.**interval_list**(*intervals_str*, *given_pitch_classes=None*)

Convert a list of intervals given as string to a binary pitch class representation. For example, 'b3, 5' would become [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0].

Parameters

intervals_str [str] List of intervals as comma-separated string (e.g. 'b3, 5').

given_pitch_classes [None or numpy array] If None, start with empty pitch class array, if numpy array of length 12, this array will be modified.

Returns

pitch_classes [numpy array] Binary pitch class representation of intervals.

madmom.evaluation.chords.**chord_intervals**(*quality_str*)

Convert a chord quality string to a pitch class representation. For example, 'maj' becomes [1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0].

Parameters

quality_str [str] String defining the chord quality.

Returns

pitch_classes [numpy array] Binary pitch class representation of chord quality.

madmom.evaluation.chords.**merge_chords**(*chords*)

Merge consecutive chord annotations if they represent the same chord.

Parameters

chords [numpy structured arrray] Chord annotations to be merged, in *CHORD_ANN_DTYPE* format.

Returns

merged_chords [numpy structured array] Merged chord annotations, in *CHORD_ANN_DTYPE* format.

madmom.evaluation.chords.**evaluation_pairs**(*det_chords*, *ann_chords*)

Match detected with annotated chords and create paired label segments for evaluation.

Parameters

det_chords [numpy structured array] Chord detections with 'start' and 'end' fields.

ann_chords [numpy structured array] Chord annotations with 'start' and 'end' fields.

Returns

annotations [numpy structured array] Annotated chords of evaluation segments.

detections [numpy structured array] Detected chords of evaluation segments.

durations [numpy array] Durations of evaluation segments.

madmom.evaluation.chords.**score_root**(*det_chords*, *ann_chords*)

Score similarity of chords based on only the root, i.e. returns a score of 1 if roots match, 0 otherwise.

Parameters

det_chords [numpy structured array] Detected chords.

ann_chords [numpy structured array] Annotated chords.

Returns

scores [numpy array] Similarity score for each chord.

madmom.evaluation.chords.**score_exact**(*det_chords*, *ann_chords*)

Score similarity of chords. Returns 1 if all chord information (root, bass, and intervals) match exactly.

Parameters

det_chords [numpy structured array] Detected chords.

ann_chords [numpy structured array] Annotated chords.

Returns

scores [numpy array] Similarity score for each chord.

`madmom.evaluation.chords.`**`reduce_to_triads`**(*chords*, *keep_bass=False*)

Reduce chords to triads.

The function follows the reduction rules implemented in *[1]*. If a chord chord does not contain a third, major second or fourth, it is reduced to a power chord. If it does not contain neither a third nor a fifth, it is reduced to a single note "chord".

Parameters

chords [numpy structured array] Chords to be reduced.

keep_bass [bool] Indicates whether to keep the bass note or set it to 0.

Returns

reduced_chords [numpy structured array] Chords reduced to triads.

References

[1]

`madmom.evaluation.chords.`**`reduce_to_tetrads`**(*chords*, *keep_bass=False*)

Reduce chords to tetrads.

The function follows the reduction rules implemented in *[1]*. If a chord does not contain a third, major second or fourth, it is reduced to a power chord. If it does not contain neither a third nor a fifth, it is reduced to a single note "chord".

Parameters

chords [numpy structured array] Chords to be reduced.

keep_bass [bool] Indicates whether to keep the bass note or set it to 0.

Returns

reduced_chords [numpy structured array] Chords reduced to tetrads.

References

[1]

`madmom.evaluation.chords.`**`select_majmin`**(*chords*)

Compute a mask that selects all major, minor, and "no chords" with a 1, and all other chords with a 0.

Parameters

chords [numpy structured array] Chords to compute the mask for.

Returns

mask [numpy array (boolean)] Selection mask for major, minor, and "no chords".

`madmom.evaluation.chords.`**`select_sevenths`**(*chords*)

Compute a mask that selects all major, minor, seventh, and "no chords" with a 1, and all other chords with a 0.

Parameters

chords [numpy structured array] Chords to compute the mask for.

Returns

mask [numpy array (boolean)] Selection mask for major, minor, seventh, and "no chords".

`madmom.evaluation.chords.`**`adjust`**(*det_chords*, *ann_chords*)

Adjust the length of detected chord segments to the annotation length.

Discard detected chords that start after the annotation ended, and shorten the last detection to fit the last annotation; discared detected chords that end before the annotation begins, and shorten the first detection to match the first annotation.

Parameters

det_chords [numpy structured array] Detected chord segments.

ann_chords [numpy structured array] Annotated chord segments.

Returns

det_chords [numpy structured array] Adjusted detected chord segments.

`madmom.evaluation.chords.`**`segmentation`**(*ann_starts*, *ann_ends*, *det_starts*, *det_ends*)

Compute the normalized Hamming divergence between chord segmentations as defined in *[1]* (Eqs. 8.37 and 8.38).

Parameters

ann_starts [list or numpy array] Start times of annotated chord segments.

ann_ends [list or numpy array] End times of annotated chord segments.

det_starts [list or numpy array] Start times of detected chord segments.

det_ends [list or numpy array] End times of detected chord segments.

Returns

distance [float] Normalised Hamming divergence between annotated and detected chord segments.

References

[1]

class `madmom.evaluation.chords.`**`ChordEvaluation`**(*detections*, *annotations*, *name=None*, ***kwargs*)

Provide various chord evaluation scores.

Parameters

detections [str] File containing chords detections.

annotations [str] File containing chord annotations.

name [str, optional] Name of the evaluation object (e.g., the name of the song).

`length`

Length of annotations.

`root`

Fraction of correctly detected chord roots.

`majmin`

Fraction of correctly detected chords that can be reduced to major or minor triads (plus no-chord). Ignores the bass pitch class.

majminbass
Fraction of correctly detected chords that can be reduced to major or minor triads (plus no-chord). Considers the bass pitch class.

sevenths
Fraction of correctly detected chords that can be reduced to a seventh tetrad (plus no-chord). Ignores the bass pitch class.

seventhsbass
Fraction of correctly detected chords that can be reduced to a seventh tetrad (plus no-chord). Considers the bass pitch class.

undersegmentation
Normalized Hamming divergence (directional) between annotations and detections. Captures missed chord segments.

oversegmentation
Normalized Hamming divergence (directional) between detections and annotations. Captures how fragmented the detected chord segments are.

segmentation
Minimum of *oversegmentation* and *undersegmentation*.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns

eval_string [str] Evaluation metrics formatted as a human readable string.

class madmom.evaluation.chords.**ChordSumEvaluation**(*eval_objects*, *name=None*)
Class for averaging Chord evaluation scores, considering the lengths of the pieces. For a detailed description of the available metrics, refer to ChordEvaluation.

Parameters

eval_objects [list] Evaluation objects.

name [str, optional] Name to be displayed.

length()
Length of all evaluation objects.

class madmom.evaluation.chords.**ChordMeanEvaluation**(*eval_objects*, *name=None*)
Class for averaging chord evaluation scores, averaging piecewise (i.e. ignoring the lengths of the pieces). For a detailed description of the available metrics, refer to ChordEvaluation.

Parameters

eval_objects [list] Evaluation objects.

name [str, optional] Name to be displayed.

length()
Number of evaluation objects.

madmom.evaluation.chords.**add_parser**(*parser*)
Add a chord evaluation sub-parser to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

Returns

sub_parser [argparse sub-parser instance] Chord evaluation sub-parser.

13.1.4 madmom.evaluation.key

This module contains key evaluation functionality.

madmom.evaluation.key.**key_label_to_class**(*key_label*)

Convert key label to key class number.

The key label must follow the MIREX syntax defined at http://music-ir.org/mirex/wiki/2017:Audio_Key_Detection: *tonic mode*, where tonic is in {C, C#, Db, ... Cb} and mode in {'major', 'maj', 'minor', 'min'}. The label will be converted into a class id based on the root pitch id (c .. 0, c# .. 1, ..., cb ... 11) plus 12 if in minor mode.

Parameters

key_label [str] Key label.

Returns

key_class [int] Key class.

Examples

```
>>> from madmom.evaluation.key import key_label_to_class
>>> key_label_to_class('D major')
2
```

```
>>> key_label_to_class('D minor')
14
```

madmom.evaluation.key.**error_type**(*det_key*, *ann_key*, *strict_fifth=False*)

Compute the evaluation score and error category for a predicted key compared to the annotated key.

Categories and evaluation scores follow the evaluation strategy used for MIREX (see http://music-ir.org/mirex/wiki/2017:Audio_Key_Detection). There are two evaluation modes for the 'fifth' category: by default, a detection falls into the 'fifth' category if it is the fifth of the annotation, or the annotation is the fifth of the detection. If *strict_fifth* is *True*, only the former case is considered. This is the mode used for MIREX.

Parameters

det_key [int] Detected key class.

ann_key [int] Annotated key class.

strict_fifth: bool Use strict interpretation of the 'fifth' category, as in MIREX.

Returns

score, category [float, str] Evaluation score and error category.

class madmom.evaluation.key.**KeyEvaluation**(*detection*, *annotation*, *strict_fifth=False*, *name=None*, ***kwargs*)

Provide the key evaluation score.

Parameters

detection [str] File containing detected key

annotation [str] File containing annotated key

strict_fifth [bool, optional] Use strict interpretation of the 'fifth' category, as in MIREX.

name [str, optional] Name of the evaluation object (e.g., the name of the song).

tostring(***kwargs*)
Format the evaluation as a human readable string.

Returns

str Evaluation score and category as a human readable string.

class madmom.evaluation.key.**KeyMeanEvaluation**(*eval_objects*, *name=None*)
Class for averaging key evaluations.

Parameters

eval_objects [list] Key evaluation objects.

name [str, optional] Name to be displayed.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

Notes

This is a fallback method formatting the *metrics* dictionary in a human readable way. Classes inheriting from this mixin class should provide a method better suitable.

madmom.evaluation.key.**add_parser**(*parser*)
Add a key evaluation sub-parser to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

Returns

sub_parser [argparse sub-parser instance] Key evaluation sub-parser.

13.1.5 madmom.evaluation.notes

This module contains note evaluation functionality.

madmom.evaluation.notes.**remove_duplicate_notes**(*data*)
Remove duplicate rows from the array.

Parameters

data [numpy array] Data.

Returns

numpy array Data array with duplicate rows removed.

Notes

This function removes only exact duplicates.

madmom.evaluation.notes.**note_onset_evaluation**(*detections*, *annotations*, *window=0.025*)
Determine the true/false positive/negative note onset detections.

Parameters

detections [numpy array] Detected notes.

annotations [numpy array] Annotated ground truth notes.

window [float, optional] Evaluation window [seconds].

Returns

tp [numpy array, shape (num_tp, 2)] True positive detections.

fp [numpy array, shape (num_fp, 2)] False positive detections.

tn [numpy array, shape (0, 2)] True negative detections (empty, see notes).

fn [numpy array, shape (num_fn, 2)] False negative detections.

errors [numpy array, shape (num_tp, 2)] Errors of the true positive detections wrt. the annotations.

Notes

The expected note row format is:

'note_time' 'MIDI_note' ['duration' ['MIDI_velocity']]

The returned true negative array is empty, because we are not interested in this class, since it is magnitudes bigger than true positives array.

class madmom.evaluation.notes.**NoteEvaluation**(*detections*, *annotations*, *window=0.025*, *delay=0*, ***kwargs*)
Evaluation class for measuring Precision, Recall and F-measure of notes.

Parameters

detections [str, list or numpy array] Detected notes.

annotations [str, list or numpy array] Annotated ground truth notes.

window [float, optional] F-measure evaluation window [seconds]

delay [float, optional] Delay the detections *delay* seconds for evaluation.

mean_error
Mean of the errors.

std_error
Standard deviation of the errors.

tostring(*notes=False*, ***kwargs*)

Parameters

notes [bool, optional] Display detailed output for all individual notes.

Returns

str Evaluation metrics formatted as a human readable string.

class madmom.evaluation.notes.**NoteSumEvaluation**(*eval_objects*, *name=None*)
Class for summing note evaluations.

errors
Errors of the true positive detections wrt. the ground truth.

class madmom.evaluation.notes.**NoteMeanEvaluation**(*eval_objects*, *name=None*, ***kwargs*)
Class for averaging note evaluations.

mean_error
Mean of the errors.

std_error
Standard deviation of the errors.

tostring(***kwargs*)
Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

madmom.evaluation.notes.**add_parser**(*parser*)
Add a note evaluation sub-parser to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

Returns

sub_parser [argparse sub-parser instance] Note evaluation sub-parser.

parser_group [argparse argument group] Note evaluation argument group.

13.1.6 madmom.evaluation.onsets

This module contains onset evaluation functionality described in *[Re366ebbe117c-1]*:

References

madmom.evaluation.onsets.**onset_evaluation**(*detections*, *annotations*, *window=0.025*)
Determine the true/false positive/negative detections.

Parameters

detections [numpy array] Detected notes.

annotations [numpy array] Annotated ground truth notes.

window [float, optional] Evaluation window [seconds].

Returns

tp [numpy array, shape (num_tp,)] True positive detections.

fp [numpy array, shape (num_fp,)] False positive detections.

tn [numpy array, shape (0,)] True negative detections (empty, see notes).

fn [numpy array, shape (num_fn,)] False negative detections.

errors [numpy array, shape (num_tp,)] Errors of the true positive detections wrt. the annotations.

Notes

The returned true negative array is empty, because we are not interested in this class, since it is magnitudes bigger than true positives array.

class madmom.evaluation.onsets.**OnsetEvaluation**(*detections*, *annotations*, *window=0.025*, *combine=0*, *delay=0*, ***kwargs*)

Evaluation class for measuring Precision, Recall and F-measure of onsets.

Parameters

detections [str, list or numpy array] Detected notes.

annotations [str, list or numpy array] Annotated ground truth notes.

window [float, optional] F-measure evaluation window [seconds]

combine [float, optional] Combine all annotated onsets within *combine* seconds.

delay [float, optional] Delay the detections *delay* seconds for evaluation.

mean_error

Mean of the errors.

std_error

Standard deviation of the errors.

tostring(***kwargs*)

Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

class madmom.evaluation.onsets.**OnsetSumEvaluation**(*eval_objects*, *name=None*)

Class for summing onset evaluations.

errors

Errors of the true positive detections wrt. the ground truth.

class madmom.evaluation.onsets.**OnsetMeanEvaluation**(*eval_objects*, *name=None*, ***kwargs*)

Class for averaging onset evaluations.

mean_error

Mean of the errors.

std_error

Standard deviation of the errors.

tostring(***kwargs*)

Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

madmom.evaluation.onsets.**add_parser**(*parser*)

Add an onset evaluation sub-parser to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

Returns

sub_parser [argparse sub-parser instance] Onset evaluation sub-parser.

parser_group [argparse argument group] Onset evaluation argument group.

13.1.7 madmom.evaluation.tempo

This module contains tempo evaluation functionality.

`madmom.evaluation.tempo.`**`sort_tempo`**(*tempo*)

Sort tempi according to their strengths.

Parameters

tempo [numpy array, shape (num_tempi, 2)] Tempi (first column) and their relative strength (second column).

Returns

tempi [numpy array, shape (num_tempi, 2)] Tempi sorted according to their strength.

`madmom.evaluation.tempo.`**`tempo_evaluation`**(*detections*, *annotations*, *tolerance=0.04*)

Calculate the tempo P-Score, at least one and all tempi correct.

Parameters

detections [list of tuples or numpy array] Detected tempi (rows, first column) and their relative strengths (second column).

annotations [list or numpy array] Annotated tempi (rows, first column) and their relative strengths (second column).

tolerance [float, optional] Evaluation tolerance (max. allowed deviation).

Returns

pscore [float] P-Score.

at_least_one [bool] At least one tempo correctly identified.

all [bool] All tempi correctly identified.

Notes

All given detections are evaluated against all annotations according to the relative strengths given. If no strengths are given, evenly distributed strengths are assumed. If the strengths do not sum to 1, they will be normalized.

References

[1]

class `madmom.evaluation.tempo.`**`TempoEvaluation`**(*detections*, *annotations*, *tolerance=0.04*, *double=True*, *triple=True*, *sort=True*, *max_len=None*, *name=None*, ***kwargs*)

Tempo evaluation class.

Parameters

detections [str, list of tuples or numpy array] Detected tempi (rows) and their strengths (columns). If a file name is given, load them from this file.

annotations [str, list or numpy array] Annotated ground truth tempi (rows) and their strengths (columns). If a file name is given, load them from this file.

tolerance [float, optional] Evaluation tolerance (max. allowed deviation).

double [bool, optional] Include double and half tempo variations.

triple [bool, optional] Include triple and third tempo variations.

sort [bool, optional] Sort the tempi by their strengths (descending order).

max_len [bool, optional] Evaluate at most *max_len* tempi.

name [str, optional] Name of the evaluation to be displayed.

Notes

For P-Score, the number of detected tempi will be limited to the number of annotations (if not further limited by *max_len*). For Accuracy 1 & 2 only one detected tempo is used. Depending on *sort*, this can be either the first or the strongest one.

`tostring(**kwargs)`
Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

`class madmom.evaluation.tempo.TempoMeanEvaluation(eval_objects, name=None, **kwargs)`
Class for averaging tempo evaluation scores.

`pscore`
P-Score.

`any`
At least one tempo correct.

`all`
All tempi correct.

`acc1`
Accuracy 1.

`acc2`
Accuracy 2.

`tostring(**kwargs)`
Format the evaluation metrics as a human readable string.

Returns

str Evaluation metrics formatted as a human readable string.

`madmom.evaluation.tempo.add_parser(parser)`
Add a tempo evaluation sub-parser to an existing parser.

Parameters

parser [argparse parser instance] Existing argparse parser object.

Returns

sub_parser [argparse sub-parser instance] Tempo evaluation sub-parser.

parser_group [argparse argument group] Tempo evaluation argument group.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

CHAPTER 15

Acknowledgements

Supported by the European Commission through the GiantSteps project (FP7 grant agreement no. 610591) and the Phenicx project (FP7 grant agreement no. 601166) as well as the Austrian Science Fund (FWF) project Z159.

Bibliography

[1] Meinard Müller, "Information retrieval for music and motion", Springer, 2007.

[1] Sebastian Böck and Gerhard Widmer "Maximum Filter Vibrato Suppression for Onset Detection" Proceedings of the 16th International Conference on Digital Audio Effects (DAFx), 2013.

[1] Meinard Müller, "Information retrieval for music and motion", Springer, 2007.

[1] Filip Korzeniowski and Gerhard Widmer, "Feature Learning for Chord Recognition: The Deep Chroma Extractor", Proceedings of the 17th International Society for Music Information Retrieval Conference (ISMIR), 2016.

[1] Meinard Müller, "Information retrieval for music and motion", Springer, 2007.

[2] Meinard Müller and Sebastian Ewert, "Chroma Toolbox: MATLAB Implementations for Extracting Variants of Chroma-Based Audio Features", Proceedings of the International Conference on Music Information Retrieval (ISMIR), 2011.

[1] Sebastian Böck and Markus Schedl, "Enhanced Beat Tracking with Context-Aware Neural Networks", Proceedings of the 14th International Conference on Digital Audio Effects (DAFx), 2011.

[1] Sebastian Böck, Florian Krebs and Gerhard Widmer, "A Multi-Model Approach to Beat Tracking Considering Heterogeneous Music Styles", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2014.

[1] Sebastian Böck and Markus Schedl, "Enhanced Beat Tracking with Context-Aware Neural Networks", Proceedings of the 14th International Conference on Digital Audio Effects (DAFx), 2011.

[1] Sebastian Böck and Markus Schedl, "Enhanced Beat Tracking with Context-Aware Neural Networks", Proceedings of the 14th International Conference on Digital Audio Effects (DAFx), 2011.

[2] Sebastian Böck, Florian Krebs and Gerhard Widmer, "Accurate Tempo Estimation based on Recurrent Neural Networks and Resonating Comb Filters", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[1] Sebastian Böck and Markus Schedl, "Enhanced Beat Tracking with Context-Aware Neural Networks", Proceedings of the 14th International Conference on Digital Audio Effects (DAFx), 2011.

[2] Sebastian Böck, Florian Krebs and Gerhard Widmer, "Accurate Tempo Estimation based on Recurrent Neural Networks and Resonating Comb Filters", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[1] Filip Korzeniowski, Sebastian Böck and Gerhard Widmer, "Probabilistic Extraction of Beat Positions from a Beat Activation Function", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2014.

[1] Sebastian Böck, Florian Krebs and Gerhard Widmer, "A Multi-Model Approach to Beat Tracking Considering Heterogeneous Music Styles", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2014.

[2] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[R974a50e873b6-1] Filip Korzeniowski, Sebastian Böck and Gerhard Widmer, "Probabilistic Extraction of Beat Positions from a Beat Activation Function", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2014.

[1] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[1] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[1] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[1] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[1] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[1] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[1] Sebastian Böck, Florian Krebs and Gerhard Widmer, "A Multi-Model Approach to Beat Tracking Considering Heterogeneous Music Styles", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2014.

[1] Sebastian Böck, Florian Krebs and Gerhard Widmer, "Joint Beat and Downbeat Tracking with Recurrent Neural Networks" Proceedings of the 17th International Society for Music Information Retrieval Conference (ISMIR), 2016.

[1] Florian Krebs, Sebastian Böck and Gerhard Widmer, "Rhythmic Pattern Modeling for Beat and Downbeat Tracking in Musical Audio", Proceedings of the 14th International Society for Music Information Retrieval Conference (ISMIR), 2013.

[1] Filip Korzeniowski and Gerhard Widmer, "Feature Learning for Chord Recognition: The Deep Chroma Extractor", Proceedings of the 17th International Society for Music Information Retrieval Conference (ISMIR), 2016.

[1] Filip Korzeniowski and Gerhard Widmer, "A Fully Convolutional Deep Auditory Model for Musical Chord Recognition", Proceedings of IEEE International Workshop on Machine Learning for Signal Processing (MLSP), 2016.

[1] Filip Korzeniowski and Gerhard Widmer, "A Fully Convolutional Deep Auditory Model for Musical Chord Recognition", Proceedings of IEEE International Workshop on Machine Learning for Signal Processing (MLSP), 2016.

[1] Sebastian Böck, Florian Krebs and Gerhard Widmer, "Joint Beat and Downbeat Tracking with Recurrent Neural Networks" Proceedings of the 17th International Society for Music Information Retrieval Conference (ISMIR), 2016.

[1] Sebastian Böck, Florian Krebs and Gerhard Widmer, "Joint Beat and Downbeat Tracking with Recurrent Neural Networks" Proceedings of the 17th International Society for Music Information Retrieval Conference (ISMIR), 2016.

[1] Florian Krebs, Sebastian Böck and Gerhard Widmer, "Rhythmic Pattern Modeling for Beat and Downbeat Tracking in Musical Audio", Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR), 2013.

[2] Florian Krebs, Sebastian Böck and Gerhard Widmer, "An Efficient State Space Model for Joint Tempo and Meter Tracking", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[1] Florian Krebs, Sebastian Böck and Gerhard Widmer, "Downbeat Tracking Using Beat-Synchronous Features and Recurrent Networks", Proceedings of the 17th International Society for Music Information Retrieval Conference (ISMIR), 2016.

[1] Filip Korzeniowski and Gerhard Widmer, "Genre-Agnostic Key Classification with Convolutional Neural Networks", In Proceedings of the 19th International Society for Music Information Retrieval Conference (ISMIR), Paris, France, 2018.

[1] Paul Masri, "Computer Modeling of Sound for Transformation and Synthesis of Musical Signals", PhD thesis, University of Bristol, 1996.

[1] Chris Duxbury, Mark Sandler and Matthew Davis, "A hybrid approach to musical note onset detection", Proceedings of the 5th International Conference on Digital Audio Effects (DAFx), 2002.

[1] Paul Masri, "Computer Modeling of Sound for Transformation and Synthesis of Musical Signals", PhD thesis, University of Bristol, 1996.

[1] Sebastian Böck and Gerhard Widmer, "Maximum Filter Vibrato Suppression for Onset Detection", Proceedings of the 16th International Conference on Digital Audio Effects (DAFx), 2013.

[1] Sebastian Böck and Gerhard Widmer, "Local group delay based vibrato and tremolo suppression for onset detection", Proceedings of the 14th International Society for Music Information Retrieval Conference (ISMIR), 2013.

[1] Paul Brossier, "Automatic Annotation of Musical Audio for Interactive Applications", PhD thesis, Queen Mary University of London, 2006.

[2] Stephen Hainsworth and Malcolm Macleod, "Onset Detection in Musical Audio Signals", Proceedings of the International Computer Music Conference (ICMC), 2003.

[1] Juan Pablo Bello, Chris Duxbury, Matthew Davies and Mark Sandler, "On the use of phase and energy for musical onset detection in the complex domain", IEEE Signal Processing Letters, Volume 11, Number 6, 2004.

[1] Simon Dixon, "Onset Detection Revisited", Proceedings of the 9th International Conference on Digital Audio Effects (DAFx), 2006.

[1] Simon Dixon, "Onset Detection Revisited", Proceedings of the 9th International Conference on Digital Audio Effects (DAFx), 2006.

[1] Juan Pablo Bello, Chris Duxbury, Matthew Davies and Mark Sandler, "On the use of phase and energy for musical onset detection in the complex domain", IEEE Signal Processing Letters, Volume 11, Number 6, 2004.

[1] Simon Dixon, "Onset Detection Revisited", Proceedings of the 9th International Conference on Digital Audio Effects (DAFx), 2006.

[1] Paul Masri, "Computer Modeling of Sound for Transformation and Synthesis of Musical Signals", PhD thesis, University of Bristol, 1996.

[2] Sebastian Böck and Gerhard Widmer, "Maximum Filter Vibrato Suppression for Onset Detection", Proceedings of the 16th International Conference on Digital Audio Effects (DAFx), 2013.

[1] "Universal Onset Detection with bidirectional Long Short-Term Memory Neural Networks" Florian Eyben, Sebastian Böck, Björn Schuller and Alex Graves. Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR), 2010.

[2] "Online Real-time Onset Detection with Recurrent Neural Networks" Sebastian Böck, Andreas Arzt, Florian Krebs and Markus Schedl. Proceedings of the 15th International Conference on Digital Audio Effects (DAFx), 2012.

[1] "Musical Onset Detection with Convolutional Neural Networks" Jan Schlüter and Sebastian Böck. Proceedings of the 6th International Workshop on Machine Learning and Music, 2013.

[1] Sebastian Böck, Florian Krebs and Markus Schedl, "Evaluating the Online Capabilities of Onset Detection Methods", Proceedings of the 13th International Society for Music Information Retrieval Conference (ISMIR), 2012.

[1] Sebastian Böck, Florian Krebs and Markus Schedl, "Evaluating the Online Capabilities of Onset Detection Methods", Proceedings of the 13th International Society for Music Information Retrieval Conference (ISMIR), 2012.

[1] Sebastian Böck and Markus Schedl, "Enhanced Beat Tracking with Context-Aware Neural Networks", Proceedings of the 14th International Conference on Digital Audio Effects (DAFx), 2011.

[1] Sebastian Böck, Florian Krebs and Gerhard Widmer, "Accurate Tempo Estimation based on Recurrent Neural Networks and Resonating Comb Filters", Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[1] "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" Sergey Ioffe and Christian Szegedy. http://arxiv.org/abs/1502.03167, 2015.

[1] Kyunghyun Cho, Bart Van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio, "On the properties of neural machine translation: Encoder-decoder approaches", http://arxiv.org/abs/1409.1259, 2014.

[1] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio, "On the properties of neural machine translation: Encoder-decoder approaches", http://arxiv.org/abs/1409.1259, 2014.

[1] Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter (2015): Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), http://arxiv.org/abs/1511.07289

[R84778d1bb8cd-1] Matthew E. P. Davies, Norberto Degara, and Mark D. Plumbley, "Evaluation Methods for Musical Audio Beat Tracking Algorithms", Technical Report C4DM-TR-09-06, Centre for Digital Music, Queen Mary University of London, 2009.

[1] M. McKinney, D. Moelants, M. Davies and A. Klapuri, "Evaluation of audio beat tracking and music tempo extraction algorithms", Journal of New Music Research, vol. 36, no. 1, 2007.

[1] A.T. Cemgil, B. Kappen, P. Desain, and H. Honing, "On tempo tracking: Tempogram representation and Kalman filtering", Journal Of New Music Research, vol. 28, no. 4, 2001.

[1] M. Goto and Y. Muraoka, "Issues in evaluating beat tracking systems", Working Notes of the IJCAI-97 Workshop on Issues in AI and Music - Evaluation and Assessment, 1997.

[2] Matthew E. P. Davies, Norberto Degara, and Mark D. Plumbley, "Evaluation Methods for Musical Audio Beat Tracking Algorithms", Technical Report C4DM-TR-09-06, Centre for Digital Music, Queen Mary University of London, 2009.

[1] S. Hainsworth, "Techniques for the automated analysis of musical audio", PhD. dissertation, Department of Engineering, Cambridge University, 2004.

[2] A.P. Klapuri, A. Eronen, and J. Astola, "Analysis of the meter of acoustic musical signals", IEEE Transactions on Audio, Speech and Language Processing, vol. 14, no. 1, 2006.

[1] M. E.P. Davies, N. Degara and M. D. Plumbley, “Measuring the performance of beat tracking algorithms algorithms using a beat error histogram”, IEEE Signal Processing Letters, vol. 18, vo. 3, 2011.

[Raff97c8dd6dc-1] Christopher Harte, “Towards Automatic Extraction of Harmony Information from Music Signals.” Dissertation, Department for Electronic Engineering, Queen Mary University of London, 2010.

[Raff97c8dd6dc-2] Johan Pauwels and Geoffroy Peeters. “Evaluating Automatically Estimated Chord Sequences.” In Proceedings of ICASSP 2013, Vancouver, Canada, 2013.

[1] Johan Pauwels and Geoffroy Peeters. “Evaluating Automatically Estimated Chord Sequences.” In Proceedings of ICASSP 2013, Vancouver, Canada, 2013.

[1] Johan Pauwels and Geoffroy Peeters. “Evaluating Automatically Estimated Chord Sequences.” In Proceedings of ICASSP 2013, Vancouver, Canada, 2013.

[1] Christopher Harte, “Towards Automatic Extraction of Harmony Information from Music Signals.” Dissertation, Department for Electronic Engineering, Queen Mary University of London, 2010.

[Re366ebbe117c-1] Sebastian Böck, Florian Krebs and Markus Schedl, “Evaluating the Online Capabilities of Onset Detection Methods”, Proceedings of the 13th International Society for Music Information Retrieval Conference (ISMIR), 2012.

[1] M. McKinney, D. Moelants, M. Davies and A. Klapuri, “Evaluation of audio beat tracking and music tempo extraction algorithms”, Journal of New Music Research, vol. 36, no. 1, 2007.

Python Module Index

m

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W