madmom: a new Python Audio and Music Signal Processing Library

Sebastian Böck†, Filip Korzeniowski†, Jan Schlüter‡, Florian Krebs†, Gerhard Widmer†‡
† Department of Computational Perception, Johannes Kepler University Linz, Austria
‡ Austrian Research Institute for Artificial Intelligence (OFAI), Vienna, Austria

ABSTRACT

In this paper, we present *madmom*, an open-source audio processing and music information retrieval (MIR) library written in Python. *madmom* features a concise, *NumPy*-compatible, object oriented design with simple calling conventions and sensible default values for all parameters, which facilitates fast prototyping of MIR applications. Prototypes can be seamlessly converted into callable processing pipelines through *madmom*'s concept of *Processors*, callable objects that run transparently on multiple cores. *Processors* can also be serialised, saved, and re-run to allow results to be easily reproduced anywhere.

Apart from low-level audio processing, *madmom* puts emphasis on musically meaningful high-level features. Many of these incorporate machine learning techniques and *madmom* provides a module that implements some methods commonly used in MIR such as hidden Markov models and neural networks. Additionally, *madmom* comes with several state-of-the-art MIR algorithms for onset detection, beat, downbeat and meter tracking, tempo estimation, and chord recognition. These can easily be incorporated into bigger MIR systems or run as stand-alone programs.

Keywords

Music Information Retrieval, Audio Analysis, Signal Processing, Machine Learning, Python, Open Source

1. INTRODUCTION

Music information retrieval (MIR) has become an emerging research area over the last 15 years. Especially audio-based MIR has become more and more important, since the amount of available audio data in the last years exploded beyond being manageable manually.

Most state-of-the-art audio-based MIR algorithms consist of two components: first, low-level features are extracted from the audio signal (*feature extraction* stage), and then the features are analysed (*feature analysis* stage) to retrieve the desired information. Most current MIR systems incorporate machine learning algorithms in the feature analysis stage, with neural networks currently being the most popular and successful ones [2, 3, 11, 19, 20].

MM '16, October 15 - 19, 2016, Amsterdam, Netherlands
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3603-1/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2964284.2973795

Numerous software libraries have been proposed over the years to facilitate research and development of applications in MIR. Some libraries concentrate on low-level feature extraction from audio signals, such as *Marsyas* [21], *YAAFE* [16] and *openSMILE* [9]. Others also include higher level feature extraction such as onset and beat detection as for example in the *MIRtoolbox* [15], *Essentia* [6] and *LibROSA* [17]. However, to our knowledge, there exist no library that also includes machine learning components (except *Marsyas* [21], which contains two classifiers), although machine learning components are crucial in current MIR applications.

Therefore, we propose *madmom*, a library that incorporates low-level feature extraction *and* high-level feature analysis based on machine learning methods. This allows the construction of the full processing chain within a single software framework, making it possible to build standalone programs without any dependency on other machine learning frameworks. Moreover, *madmom* comes with several state-of-the-art systems including their trained models, for example for onset detection [8, 19, 20], tempo estimation [3], beat [2, 11] and downbeat tracking [4, 14], and chord recognition [12, 13].

madmom is written in Python, which has become the language of choice for scientific computing for many people due to its free availability and its simplicity of use. The code is released under BSD license and pre-trained models are released under the CC BY-NC-SA 4.0 license.

1.1 Design and Functionality

1.1.1 Object-oriented programming

madmom follows an object-oriented programming (OOP) approach. We encapsulate everything in objects that are often designed as subclasses of *NumPy*'s *ndarray*, offering all array handling routines inherited from *NumPy* [22] with additional functionality. This compactly bundles data and meta-data (e.g. a *Spectrogram* and its *frame rate*) and simplifies meta-data handling for the user.

1.1.2 Rapid prototyping

madmom aims at minimising the turnaround time from a research idea to a software prototype. Thus, object instantiation is made as simple as possible: e.g., a *Spectrogram* object can be instantiated with a single line of code by only providing the path to an audio file. *madmom* automatically creates all objects in between using sensible default values.

1.1.3 Simple conversion into runnable programs

Once an audio processing algorithm is prototyped, the complete workflow should be easily transformable into a runnable standalone program with a consistent calling interface. This is implemented using *madmom*'s concept of *Processors*.

1.1.4 Machine learning integration

We aim at a seamless integration of machine learning methods without the need of any third party modules. We limit ourselves to testing capabilities (applying pre-trained models), since it is impossible to keep up with newly emerging training methods in the various machine learning domains. Models that have been trained in an external library should be easily be convertible to an internal *madmom* model format.

1.1.5 State-of-the-art features

Many existing libraries provide a huge variety of low-level features, but few musically meaningful high-level features. *madmom* tries to close this gap by offering high-quality state-of-the-art feature extractors for onsets, beats, downbeats, chords, tempo, etc.

1.1.6 Reproducible research

In order to foster reproducible research, we want to be able to save and load the specific settings used to obtain the results for a certain experiment. In *madmom* this is implemented using Python's own *pickle* functionality which allows to save an entire processing chain (including all settings) to a file.

1.1.7 Few dependencies

madmom is built on top of three excellent and wide-spread libraries: *NumPy* [22] provides all the array handling subroutines for *madmom*'s *data classes*. *SciPy* [10] provides optimised routines for the fast Fourier transform (FFT), linear algebra operations and sparse matrix representations. Finally, *Cython* [1] is used to speed up time critical parts of the library by automatically generating C code from a Python-like syntax and then compiling and linking it into extensions which can be transparently used from within Python. These libraries are the only installation and runtime dependencies of *madmom* besides the *Python* standard library itself, supported in version 2.7 as well as 3.3 and newer.

1.1.8 Multi-core capability

We designed *madmom* to be able to exploit the multi-core capabilities of modern computer architectures, by providing functionality to run several programs or *Processors* in parallel.

1.1.9 Extensive documentation

All source code files contain thorough documentation following the *NumPy* format. The complete API reference, instruction on how to build and install the library, as well as interactive *Jupyter* [18] notebooks can be found online at `http://madmom.readthedocs.io`. The documentation is built automatically with *Sphinx*[1].

1.1.10 Open development process

We follow an open development process and the source code and documentation of our project is publicly available on GitHub: `http://github.com/CPJKU/madmom`. To maintain high code quality, we use continuous integration testing via TravisCI[2], code quality tests via QuantifiedCode[3], and test coverage via Coveralls[4].

[1]`http://www.sphinx-doc.org`

2. LIBRARY DESCRIPTION

In this section, we will describe the overall architecture of *madmom*, its packages as well as the provided standalone programs.

madmom's main API is composed of classes, but much of the functionality is implemented as functions (in turn used internally by the classes). This way, *madmom* offers the 'best of both worlds': concise interfaces exposed through classes, and detailed access to functionality through functions. In general, the classes can be split in two different types: the so called *data classes* and *processor classes*.

Data classes represent data entities such as audio signals or spectrograms. They are implemented as subclasses of *NumPy*'s *ndarray*, and thus offer all array handling routines inherited directly from *NumPy* (e.g., transposing or saving the data to file in either binary or human readable format). These classes are enriched by additional attributes and expose additional functionality via methods.

Processor classes exclusively store information on how to process data, i.e. how to transform one data class into another (e.g., from an (audio-)*Signal* into a *Spectrogram*). In order to build chains of transformations, each data class has its corresponding processor class, which implements this transformation. This enables a simple and fast conversion of algorithm prototypes to callable processing pipelines.

2.1 Packages

The library is split into several packages, grouped by functionality. For a detailed description including examples of usage please refer to the library's documentation.

2.1.1 madmom.audio

The *madmom.audio* package includes basic audio signal processing and "low-level" functionality.

The *Signal* and *FramedSignal* classes are used to load an audio signal and split it into (overlapping) frames. Following *madmom*'s automatic instantiation approach, both classes can be instantiated from any object up the instantiation hierarchy – including a simple file name. *madmom* supports almost a wide range of audio and video formats – provided *ffmpeg*[5] is installed – and transparently converts sample rates and number of channels if needed.

Signal is a subclass of *ndarray* with additional attributes like *sample rate* or *number of channels*. *FramedSignal* supports float hop sizes, making it possible to build systems with an arbitrary frame rate – independently of the signal's sample rate – and ensures that all frames are temporally aligned, even if computed with different frame sizes.

The *ShortTimeFourierTransform* and *Spectrogram* classes represent the complex valued STFT and magnitudes respectively. They are the key classes for spectral audio analysis and provide windowing, automatic circular shifting (for correct phase) and zero-padding. Both are independent of the

[2]`http://www.travis-ci.org`
[3]`http://www.quantifiedcode.com`
[4]`http://www.coveralls.io`
[5]`http://www.ffmpeg.org`

data type (integer or float) of the underlying *Signal*, resulting in spectrograms of the same value range. A *Spectrogram* can be filtered with a *Filterbank* (e.g., Mel, Bark, logarithmic), which in turn can be parametrised to reduce the dimensionality or transform the spectrogram into a logarithmically spaced pitch representation closely following the auditory model of the human ear. *madmom* also provides standard *MFCC* and *Chroma* features.

Listing 1 shows an example of how a standard spectral flux onset detection function can be prototyped with *madmom* in a few lines of code.

```
from madmom.audio import (Spectrogram,
                          SpectrogramDifference)

# compute spectral flux
spec = Spectrogram('sample.wav')
diff = SpectrogramDifference(spec,
                             positive_diffs=True)
sf = np.mean(diff, axis=1)
```

Listing 1: Rapid prototyping of the spectral flux onset detection function using *madmom*'s data classes.

2.1.2 *madmom.processors*

Processors are one of the fundamental building blocks of *madmom*. Each *Processor* accepts a number of processing parameters and must provide a *process* method, which takes the data to be processed as its only argument and defines the processing functionality of the *Processor*. An *OutputProcessor* extends this scheme by accepting a second argument which defines the output and can thus be used to write the output of an algorithm to a file. All *Processors* are callable, making it easy to use them interchangeably with normal functions. Furthermore, the *Processor* class provides methods for saving and loading any *Processor* to a file – including all parameters – using Python's own *pickle* library. This facilitates the reproducibility of an experiment.

Multiple *Processors* can be combined into a processing chain using either a *SequentialProcessor* or a *ParallelProcessor*, which execute the chain sequentially or in parallel, using multiple CPU cores if available.

```
from madmom.audio import (
    SpectrogramProcessor,
    SpectrogramDifferenceProcessor)
from madmom.processors import SequentialProcessor
from functools import partial

# define spectral flux processing chain
spec = SpectrogramProcessor()
diff = SpectrogramDifferenceProcessor(
    positive_diffs=True)
mean = partial(np.mean, axis=1)
# wrap everything in a SequentialProcessor
sf_proc = SequentialProcessor([spec, diff, mean])

# process an audio file by calling the processor
sf = sf_proc('sample.wav')
```

Listing 2: Spectral flux onset detection implemented as a callable *Processor*.

Listing 2 shows the conversion of the prototyped algorithm in Listing 1 into a callable *Processor* by simply replacing the used *data classes* with their respective *processor classes* and wrapping them into a *SequentialProcessor*.

2.1.3 *madmom.features*

The *madmom.features* package includes "high-level" functionality which are related to certain MIR tasks, such as onset detection or beat tracking. *madmom*'s focus is on providing musically meaningful and descriptive features rather than a vast number of low to mid-level features. At the time of writing, *madmom* contains state-of-the-art features for onset detection, beat and downbeat tracking, rhythm pattern analysis, tempo estimation and chord recognition.

All features are implemented as *Processors* without a corresponding *data class*. Users can thus use the provided functionality and build algorithms on top of these features. For most of the features, *madmom* also provides stand-alone programs with a consistent calling interface to process audio files (see Section 2.2).

```
from madmom.features.beats import (
    RNNBeatProcessor, DBNBeatTrackingProcessor)
from madmom.processors import SequentialProcessor

# define beat tracking processor
rnn = RNNBeatProcessor()
dbn = DBNBeatTrackingProcessor(
    min_bpm=50, max_bpm=200)
tracker = SequentialProcessor([rnn, dbn])

# track the beats by calling the processor
beats = tracker('sample.wav')
```

Listing 3: Beat tracking with a recurrent neural network (RNN) and dynamic Bayesian network (DBN) in *madmom* using provided *Processors*.

2.1.4 *madmom.evaluation*

All features come with code for evaluation. The implemented metrics are those commonly found in the literature of the respective field.

2.1.5 *madmom.ml*

Most of today's top performing music analysis algorithms incorporate machine learning, with neural networks being the most universal and successful ones at the moment. *madmom* includes Python implementations of commonly used machine learning techniques, namely *Gaussian Mixture Models*, *Hidden Markov Models*, linear-chain *Conditional Random Fields*, and different types of *neural networks*, including feed forward, convolutional, batch normalisation, and recurrent layers, various activation functions and special purpose *long short-term memory* and gated recurrent units.

madmom provides functionality to use these techniques without any dependencies on third-party modules, but does not contain training algorithms. This decision was made on purpose since the library's main focus is on applying machine learning techniques to MIR, rather than providing an extensive set of learning techniques. However, trained models can be easily converted to be compatible with *madmom*, since neural network layers usually are simply defined as a set of weights, biases and an activation function they apply to the input data. Listing 4 shows the code necessary to convert a simple neural network trained using the *Lasagne* library [7].

```
import lasagne
from madmom import ml

# ... here comes the Lasagne training code
# 'net' is the Lasagne network handle
p = lasagne.layers.get_all_param_values(net)
# create a NeuralNetwork in madmom
nn = ml.nn.NeuralNetwork([
    ml.nn.layers.FeedForwardLayer(
        p[0], p[1], ml.nn.activations.relu),
    ml.nn.layers.FeedForwardLayer(
        p[2], p[3], ml.nn.activations.relu),
    ml.nn.layers.FeedForwardLayer(
        p[4], p[5], ml.nn.activations.relu),
    ml.nn.FeedForwardLayer(
        p[6], p[7], ml.nn.activations.sigmoid)
])
# save the network (can be used as a processor)
nn.dump('neural_net.pkl')
```

Listing 4: Converting a deep neural network with three hidden layers from *Lasagne* to *madmom*.

2.1.6 madmom.models

madmom comes with a set of pre-trained models which are distributed under a Creative Commons attribution non-commercial share-alike license, i.e. they can be freely used for research purposes as long as derivative works are distributed under the same license. *madmom* uses the exact same mechanism to save and load the models it uses for *Processors* to be pickled.

2.2 Standalone Programs

madmom comes with a set of standalone programs, covering many areas of MIR. Table 1 lists selected programs included in the library with the performance achieved at the annual *Music Information Retrieval Evaluation eXchange* (MIREX)[6], where MIR algorithms are compared on test datasets. We aggregated the results of all years (2006-2016), i.e. a rank 1 means that the algorithm is the best performing one of all submissions from 2006 until present. The outstanding results in Table 1 highlight the state-of-the-art features *madmom* provides.

Table 1: Ranks of the programs included in *madmom* for the MIREX evaluations, results aggregated over all years (2006-2016).

Program	Task	Year	Rank
CNNOnsetDetector [19]	onset	2016	1
OnsetDetector [8]	onset	2013	2
BeatTracker [5]	beat MCK	2015	1
DBNBeatTracker [2]	beat SMC	2015	1
CRFBeatDetector [11]	beat MAZ	2015	1
DBNDownBeatTracker [4]	downbeat	2016	1
TempoDetector [3]	tempo	2015	1
CNNChordRecognition [13]	chord	2016	1

[6]http://www.music-ir.org/mirex/wiki/

These programs are simple wrappers around the functionality provided by the *madmom.features* package, and provide a simple and easy to use command line interface. They are implemented as *Processors* and can operate either in *single* or *batch* mode, processing single or multiple input files, respectively. Additionally all programs can be *pickled*, serialising all parameters in a way that the program can be executed later on with the exact same settings.

Listing 5 shows the beat tracking algorithm of Listing 3 converted into a runnable program. The usage of an *IOProcessor* instead of a *SequentialProcessor* enables the program to output the detected beats into a file or to `STDOUT`. `io_arguments` provides all the handling needed for the different operation modes.

```
#!/usr/bin/env python

from madmom.processors import (
    IOProcessor, io_arguments)
from madmom.features.beats import (
    RNNBeatProcessor, DBNBeatTrackingProcessor)
from madmom.utils import write_events
from argparse import ArgumentParser

if __name__ == '__main__':
    # define parser
    p = ArgumentParser()
    # add I/O related arguments and define
    # single/batch/pickle processing mode
    io_arguments(p, output_suffix='.beats.txt')
    # parse arguments
    args = p.parse_args()
    # define beat tracking processor
    rnn = RNNBeatProcessor()
    dbn = DBNBeatTrackingProcessor()
    out = write_events
    # wrap as IOProcessor
    processor = IOProcessor([rnn, dbn], out)
    # call the processor in single/batch/pickle
    # mode defined by `args.func`
    args.func(processor, **vars(args))
```

Listing 5: Beat tracking algorithm of Listing 3 converted into a runnable program.

3. CONCLUSION

This paper gave a short introduction to *madmom*, its design principles and library structure. Up-to-date information on functionality can be found in the project's online documentation at `https://madmom.readthedocs.io` and source code repository at `https://github.com/CPJKU/madmom`.

Future work aims at including a streaming mode, i.e. providing online real-time processing of audio signals in a memory efficient way instead of processing whole audio files at a time. In addition, we will gradually extend the set of features and algorithms, as well as add tools to automatically convert models that have been trained with machine learning libraries.

4. ACKNOWLEDGMENTS

This work is supported by the European Union Seventh Framework Programme FP7 / 2007-2013 through the GiantSteps project (grant agreement no. 610591) and the Austrian Science Fund (FWF) project Z159.

5. REFERENCES

[1] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith. Cython: The Best of Both Worlds. *Computing in Science Engineering*, 13(2), 2011.

[2] S. Böck, F. Krebs, and G. Widmer. A Multi-model Approach to Beat Tracking considering Heterogeneous Music Styles. In *Proc. of the 15th Int. Society for Music Information Retrieval Conf. (ISMIR)*, 2014.

[3] S. Böck, F. Krebs, and G. Widmer. Accurate Tempo Estimation based on Recurrent Neural Networks and Resonating Comb Filters. In *Proc. of the 16th Int. Society for Music Information Retrieval Conf. (ISMIR)*, 2015.

[4] S. Böck, F. Krebs, and G. Widmer. Joint Beat and Downbeat Tracking with Recurrent Neural Networks. In *Proc. of the 17th Int. Society for Music Information Retrieval Conf. (ISMIR)*, 2016.

[5] S. Böck and M. Schedl. Enhanced Beat Tracking with Context-Aware Neural Networks. In *Proc. of the 14th Int. Conf. on Digital Audio Effects (DAFx)*, 2011.

[6] D. Bogdanov, N. Wack, E. Gómez, S. Gulati, P. Herrera, O. Mayor, G. Roma, J. Salamon, J. Zapata, and X. Serra. Essentia: an open source library for sound and music analysis. In *In Proc. of ACM Multimedia*, 2013.

[7] S. Dieleman, J. Schlüter, C. Raffel, E. Olson, S. K. Sønderby, D. Nouri, E. Battenberg, A. van den Oord, et al. Lasagne: First release., 2015.

[8] F. Eyben, S. Böck, B. Schuller, and A. Graves. Universal Onset Detection with Bidirectional Long Short-Term Memory Neural Networks. In *Proc. of the 11th Int. Society for Music Information Retrieval Conf. (ISMIR)*, 2010.

[9] F. Eyben, F. Weninger, F. Gross, and B. Schuller. Recent Developments in openSMILE, the Munich Open-Source Multimedia Feature Extractor. In *In Proc. of ACM Multimedia*, Barcelona, Spain, 2013.

[10] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2016-05-20].

[11] F. Korzeniowski, S. Böck, and G. Widmer. Probabilistic Extraction of Beat Positions from a Beat Activation Function. In *Proc. of the 15th Int. Society for Music Information Retrieval Conf. (ISMIR)*, 2014.

[12] F. Korzeniowski and G. Widmer. Feature learning for chord recognition: The deep chroma extractor. In *Proc. of the 17th Int. Society for Music Information Retrieval Conf. (ISMIR)*, 2016.

[13] F. Korzeniowski and G. Widmer. A fully convolutional deep auditory model for musical chord recognition. In *Proc. of the IEEE Int. Workshop on Machine Learning for Signal Processing (MLSP)*, 2016.

[14] F. Krebs, S. Böck, and G. Widmer. Rhythmic Pattern Modeling for Beat and Downbeat Tracking in Musical Audio. In *Proc. of the 14th Int. Society for Music Information Retrieval Conf. (ISMIR)*, 2013.

[15] O. Lartillot and P. Toiviainen. A Matlab toolbox for musical feature extraction from audio. In *Proc. of the 10th Int. Conf. on Digital Audio Effects (DAFx)*, 2007.

[16] B. Mathieu, S. Essid, T. Fillon, and J. Prado. YAAFE, an easy to use and efficient audio feature extraction software. In *Proc. of the 11th Int. Society for Music Information Retrieval Conf. (ISMIR)*, 2010.

[17] B. McFee, C. Raffel, D. Liang, D. Ellis, M. McVicar, E. Battenberg, and O. Nieto. librosa: Audio and Music Signal Analysis in Python. In *Proc. of the 14th Python in Science Conf. (SCIPY)*, 2015.

[18] F. Pérez and B. E. Granger. IPython: A System for Interactive Scientific Computing. *Computing in Science Engineering*, 9(3), 2007.

[19] J. Schlüter and S. Böck. Musical Onset Detection with Convolutional Neural Networks. In *Proceedings of the 6th International Workshop on Machine Learning and Music*, Prague, Czech Republic, 2013.

[20] J. Schlüter and S. Böck. Improved musical onset detection with convolutional neural networks. In *Proc. of the 39th Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, 2014.

[21] G. Tzanetakis and P. Cook. MARSYAS: a framework for audio analysis. *Organised Sound*, 4, 2000.

[22] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering*, 13(2), 2011.