carat

Release 0.1.0

Jul 03, 2019

Contents

carat documentation

```
 _  _ __  _ _|_
(_ (_| | (_| |_   computer-aided rhythm analysis toolbox
```

For a quick introduction to using `carat`, please refer to the *Tutorial*.

CHAPTER 1

Getting started

1.1 Installation

1.1.1 pypi

The simplest way to install *carat* is through the Python Package Index (PyPI). This will ensure that all required dependencies are fulfilled. This can be achieved by executing the following command:

```
pip install carat
```

or:

```
sudo pip install carat
```

to install system-wide, or:

```
pip install -u carat
```

to install just for your own user.

1.1.2 Source

If you've downloaded the archive manually from the releases page, you can install using the `setuptools` script:

```
tar xzf carat-VERSION.tar.gz
cd carat-VERSION/
python setup.py install
```

If you intend to develop librosa or make changes to the source code, you can install with *pip install -e* to link to your actively developed source tree:

```
tar xzf carat-VERSION.tar.gz
cd carat-VERSION/
pip install -e .
```

Alternately, the latest development version can be installed via pip:

```
pip install git+https://github.com/mrocamora/carat
```

1.1.3 ffmpeg

To fuel `audioread` with more audio-decoding power, you can install *ffmpeg* which ships with many audio decoders. Note that conda users on Linux and OSX will have this installed by default; Windows users must install ffmpeg separately.

OSX users can use *homebrew* to install ffmpeg by calling *brew install ffmpeg* or get a binary version from their website https://www.ffmpeg.org.

1.2 Tutorial

This section covers some fundamentals of using *carat*, including a package overview and basic usage. We assume basic familiarity with Python and NumPy/SciPy.

1.2.1 Overview

The *carat* package is structured as a collection of submodules:

- carat
  - ***carat.annotations*** Functions for loading annotations files, such as beat annotations.
  - ***carat.display*** Visualization and display routines using `matplotlib`.
  - ***carat.clustering*** Functions for clustering and low-dimensional embedding.
  - ***carat.features*** Feature extraction and manipulation.
  - ***carat.util*** Helper utilities.

1.2.2 Quickstart

The following is a brief example program for rhythmic patterns analysis using carat.

It is based on the rhythmic patterns analysis proposed in [CIM2014]

```
'''
 _  _  __ _ _|_
(_ (_| | (_| |_   computer-aided rhythm analysis toolbox

Rhythmic patterns analysis example

'''
import carat

```

(continues on next page)

(continued from previous page)

```python
# 1. Get the file path to an included audio example
#    This is a recording of an ensemble of candombe drums
audio_path = carat.util.example_audio_file(num_file=1)

# 2. Get the file path the annotations for the example audio file
annotations_path = carat.util.example_beats_file(num_file=1)

# 3. Load the audio waveform `y` and its sampling rate as `sr`
y, sr = carat.audio.load(audio_path, sr=None)

# 4. Load beats/downbeats time instants and beat/downbeats labels
beats, beat_labs = carat.annotations.load_beats(annotations_path)
downbeats, downbeat_labs = carat.annotations.load_downbeats(annotations_path)

# 5. Compute an accentuation feature indicating when a note has been articulated
#    We focus on the low frequency band (20 to 200 Hz) to get low sounding drum events
acce, times, _ = carat.features.accentuation_feature(y, sr, minfreq=20, maxfreq=200)

# 6. Compute a feature map of the rhythmic patterns
# number of beats per bar
n_beats = int(round(beats.size/downbeats.size))
# you have to provide the number of tatums (subdivisions) per beat
n_tatums = 4
# compute the feature map from the feature signal and the beat/dowbeat annotations
map_acce, _, _, _ = carat.features.feature_map(acce, times, beats, downbeats,
                                               n_beats=n_beats, n_tatums=n_tatums)


# 7. Group rhythmic patterns into clusters
# set the number of clusters to look for
n_clusters = 4
# clustering of rhythmic patterns
cluster_labs, centroids, _ = carat.clustering.rhythmic_patterns(map_acce, n_
↪clusters=n_clusters)
```

The first step of the program:

```python
audio_path = carat.util.example_audio_file(num_file=1)
```

gets the path to an audio example file included with *carat*. After this step, `audio_path` will be a string variable containing the path to the example audio file.

Similarly, the following line:

```python
annotations_path = carat.util.example_beats_file(num_file=1)
```

gets the path to the annotations file for the same example.

The second step:

```python
y, sr = carat.audio.load(audio_path)
```

loads and decodes the audio as a `y`, represented as a one-dimensional NumPy floating point array. The variable `sr` contains the sampling rate of `y`, that is, the number of samples per second of audio. By default, all audio is mixed to mono and resampled to 22050 Hz at load time. This behavior can be overridden by supplying additional arguments to `carat.audio.load()`.

Next, we load the annotations:

```
beats, beat_labs = carat.annotations.load_beats(annotations_path)
downbeats, downbeat_labs = carat.annotations.load_downbeats(annotations_path)
```

The `beats` are a one-dimensional Numpy array representing the time location of beats, and `beat_labs` is a list of `string` elements that correspond to the labels given for each beat. This is the same for `downbeats` and `downbeat_labs`, except that they correspond to downbeats.

Then, we compute an accentuation feature from the audio waveform:

```
acce, times, _ = carat.features.accentuation_feature(y, sr, minfreq=20, maxfreq=200)
```

This is based on the Spectral flux, that consists in seizing the changes in the spectral magnitude of the audio signal along different frequency bands. In principle, the feature value is high when a note has been articulated and close to zero otherwise. Note that this example is tailored towards the rhythmic patterns of the lowest sounding of the three drum types taking part in the recording, so the analysis focuses on the low frequencies (20 to 200 Hz).

The feature values are stored in the one-dimensional Numpy array `acce`, and the time instants corresponding to each feature value are given in `times`, which is also a one-dimensional Numpy array.

Next, we compute the feature map from the feature signal and the beat/downbeat annotations:

```
n_beats = int(round(beats.size/downbeats.size))
n_tatums = 4
map_acce, _, _, _ = carat.features.feature_map(acce, times, beats, downbeats,
                                               n_beats=n_beats, n_tatums=n_tatums)
```

Note that we have to provide the beats and the downbeats, which were loaded from the annotations. Besides, the number of beats per bar and the number of of tatums (subdivisions) per beat has to be provided.

In this step the accentuation feature is organized into a feature map. First, the feature signal is time-quantized to the rhythm metric structure by considering a grid of tatum pulses equally distributed within the annotated beats. The corresponding feature value is taken as the maximum within window centered at the frame closest to each tatum instant. This yields feature vectors whose coordinates correspond to the tatum pulses of the rhythm cycle (or bar). Finally, a feature map of the cycle-length rhythmic patterns of the audio file is obtained by building a matrix whose columns are consecutive feature vectors, and stored in `map_acce` as a Numpy array matrix.

Finally, the rhythmic patterns of the feature map are grouped into clusters:

```
n_clusters = 4
cluster_labs, centroids, _ = carat.clustering.rhythmic_patterns(map_acce, n_
↪clusters=n_clusters)
```

Note that the number of clusters `n_clusters` has to be specified as an input parameter. The clustering is done using the classical K-means method with Euclidean distance (but other clustering methods and distance measures can be used too).

The result of the clustering is a set of cluster numbers given in `cluster_labs`, that indicate to which cluster belongs each rhythmic pattern. Besides, the centroid of each cluster is given in `centroids` as a representative rhythmic pattern of the group. In this way, they represent the different types of rhythmic patterns found in the recording.

…

1.2.3 More examples

More example scripts are provided in the *Examples* section.

orphan

1.3 Examples

Note: Click *here* to download the full example code

1.3.1 Plot audio and beats

This example shows how to load/plot an audio file and the corresponding beat annotations file.

```
# Code source: Martín Rocamora
# License: MIT
```

Imports

- matplotlib for visualization

```
from __future__ import print_function
import matplotlib.pyplot as plt
import carat
```

First, we'll load one of the audio files included in `carat`. We get the path to the audio file example number 1, and load 10 seconds of the file.

Note 1: By default, `carat` will resample the signal to 22050Hz, but this can disabled by saying *sr=None* (`carat` uses librosa for loading audio files, so it inherits all its functionality and behaviour).

```
audio_path = carat.util.example_audio_file(num_file=1)

y, sr = carat.audio.load(audio_path, duration=10.0)
```

Next, we'll load the annotations provided for the example audio file. We get the path to the annotations file corresponding to example number 1, and then we load beats and downbeats, along with their labels.

```
annotations_path = carat.util.example_beats_file(num_file=1)

beats, beat_labs = carat.annotations.load_beats(annotations_path)
downbeats, downbeat_labs = carat.annotations.load_downbeats(annotations_path)
```

Note 2: It is assumed that the beat annotations are provided as a text file (csv). Apart from the time data (mandatory) a label can be given for each beat (optional). The time data is assumed to be given in seconds. The labels may indicate the beat number within the rhythm cycle (e.g. 1.1, 1.2, or 1, 2).

Note 3: The same annotations file is used for both beats and downbeats. This is based on annotation labels that provide a particular string to identify the downbeats. In this case, this string is .1, and is the one used by default. You can specify the string to look for in the labels data to select downbeats by setting the *downbeat_label* parameter value. For instance, *downbeat_label='1'* is used for loading annotations of the samba files included.

Note 4: By default the columns are assumed to be separated by a comma, but you can specify another separating string by setting the *delimiter* parameter value. For instance, a blank space *delimiter=' '* is used for loading annotations of the samba files included.

Let's print the first 10 beat and the first 3 downbeats, with their corresponding labels.

```
print(beats[:10])
print(beat_labs[:10])
```

(continues on next page)

(continued from previous page)

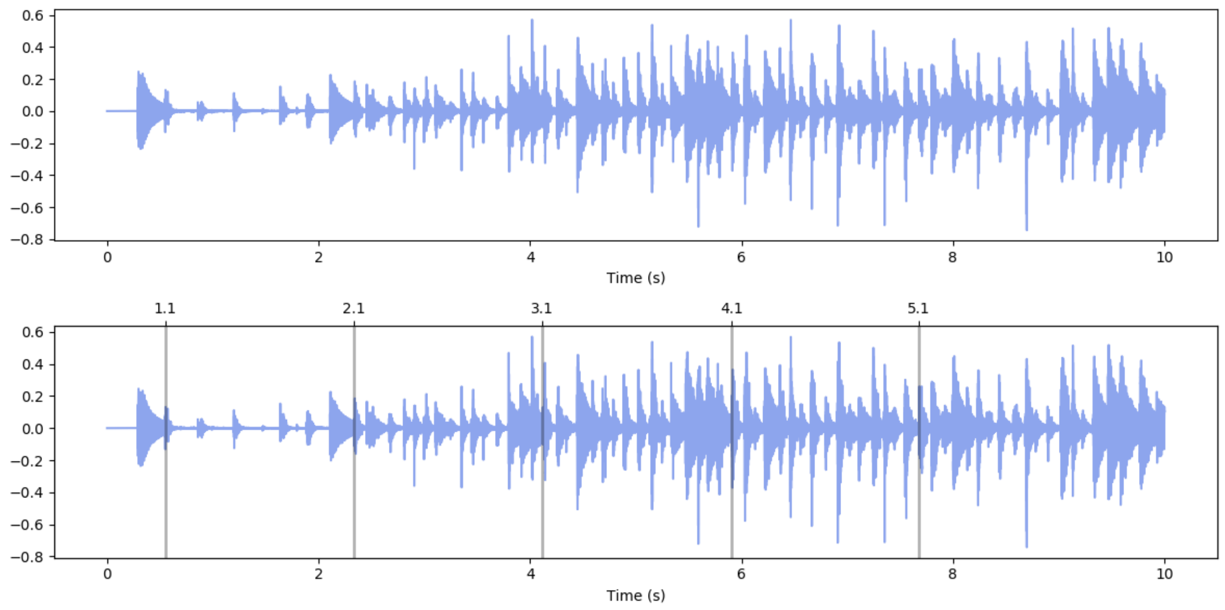
```
print(downbeats[:3])
print(downbeat_labs[:3])
```

Out:

```
[0.54857143 0.99387755 1.46140589 1.8953288  2.33265306 2.80902494
 3.25365079 3.68412698 4.11530612 4.5815873 ]
['1.1', '1.2', '1.3', '1.4', '2.1', '2.2', '2.3', '2.4', '3.1', '3.2']
[0.54857143 2.33265306 4.11530612]
['1.1', '2.1', '3.1']
```

Finally we plot the audio waveform and the beat annotations

```
plt.figure(figsize=(12, 6))
ax1 = plt.subplot(2, 1, 1)
carat.display.wave_plot(y, sr, ax=ax1)
ax2 = plt.subplot(2, 1, 2, sharex=ax1)
carat.display.wave_plot(y, sr, ax=ax2, beats=downbeats, beat_labs=downbeat_labs)
plt.tight_layout()

plt.show()
```



Total running time of the script: (0 minutes 2.159 seconds)

Note: Click *here* to download the full example code

1.3.2 Plot feature map

This example shows how to compute a feature map from de audio waveform.

This type of feature map for rhythmic patterns analysis was first proposed in [CIM2014].

```
# Code source: Martín Rocamora
# License: MIT
```

Imports

- matplotlib for visualization

```
from __future__ import print_function
import matplotlib.pyplot as plt
import carat
```

The accentuation feature is organized into a feature map. First, the feature signal is time-quantized to the rhythm metric structure by considering a grid of tatum pulses equally distributed within the annotated beats. The corresponding feature value is taken as the maximum within window centered at the frame closest to each tatum instant. This yields feature vectors whose coordinates correspond to the tatum pulses of the rhythm cycle (or bar). Finally, a feature map of the cycle-length rhythmic patterns of the audio file is obtained by building a matrix whose columns are consecutive feature vectors.

First, we'll load one of the audio files included in `carat`.

```
audio_path = carat.util.example_audio_file(num_file=1)

y, sr = carat.audio.load(audio_path)
```

Next, we'll load the annotations provided for the example audio file.

```
annotations_path = carat.util.example_beats_file(num_file=1)

beats, beat_labs = carat.annotations.load_beats(annotations_path)
downbeats, downbeat_labs = carat.annotations.load_downbeats(annotations_path)
```

Then, we'll compute the accentuation feature.

Note: This example is tailored towards the rhythmic patterns of the lowest sounding of the three drum types taking part in the recording, so the analysis focuses on the low frequencies (20 to 200 Hz).

```
acce, times, _ = carat.features.accentuation_feature(y, sr, minfreq=20, maxfreq=200)
```

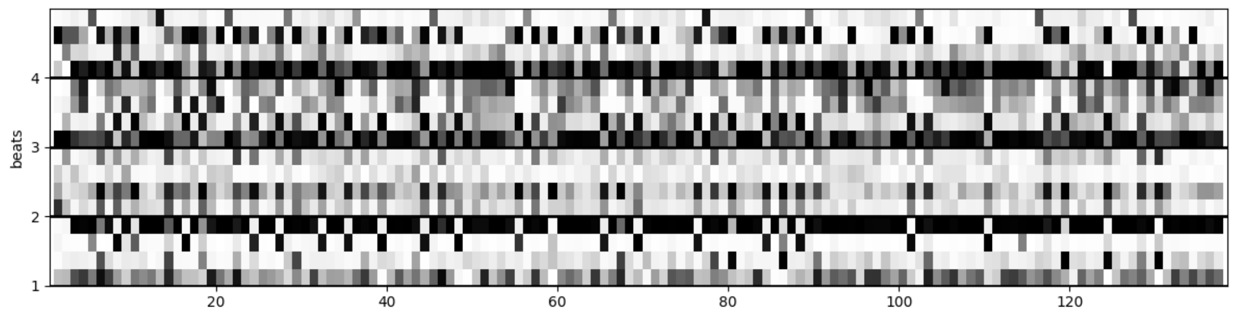
Next, we'll compute the feature map. Note that we have to provide the beats, the downbeats, which were loaded from the annotations. Besides, the number of beats per bar and the number of of tatums (subdivisions) per beat has to be provided.

```
n_beats = int(round(beats.size/downbeats.size))
n_tatums = 4

map_acce, _, _, _ = carat.features.feature_map(acce, times, beats, downbeats, n_
↪beats=n_beats,
                                               n_tatums=n_tatums)
```

Finally we plot the feature map for the low frequencies of the audio file.

Note: This feature map representation enables the inspection of the patterns evolution over time, as well as their similarities and differences, in a very informative way. Note that if a certain tatum pulse is articulated for several consecutive bars, it will be shown as a dark horizontal line in the map. Conversely, changes in repetitive patterns are readily distinguishable as variations in the distribution of feature values.

```
plt.figure(figsize=(12, 6))
ax1 = plt.subplot(211)
carat.display.map_show(map_acce, ax=ax1, n_tatums=n_tatums)
plt.tight_layout()

plt.show()
```



Total running time of the script: (0 minutes 21.352 seconds)

Note: Click *here* to download the full example code

1.3.3 Plot accentuation feature

This example shows how to compute an accentuation feature from de audio waveform.

```
# Code source: Martín Rocamora
# License: MIT
```

Imports

- matplotlib for visualization

```
from __future__ import print_function
import matplotlib.pyplot as plt
import carat
```

The accentuation feature is based on the Spectral flux, that consists in seizing the changes in the spectral magnitude of the audio signal along different frequency bands. In principle, the feature value is high when a note has been articulated and close to zero otherwise.

First, we'll load one of the audio files included in `carat`. We get the path to the audio file example number 1, and load 10 seconds of the file.

```
audio_path = carat.util.example_audio_file(num_file=1)

y, sr = carat.audio.load(audio_path, duration=10.0)
```

Next, we'll load the annotations provided for the example audio file. We get the path to the annotations file corresponding to example number 1, and then we load beats and downbeats, along with their labels.

```
annotations_path = carat.util.example_beats_file(num_file=1)

beats, beat_labs = carat.annotations.load_beats(annotations_path)
downbeats, downbeat_labs = carat.annotations.load_downbeats(annotations_path)
```

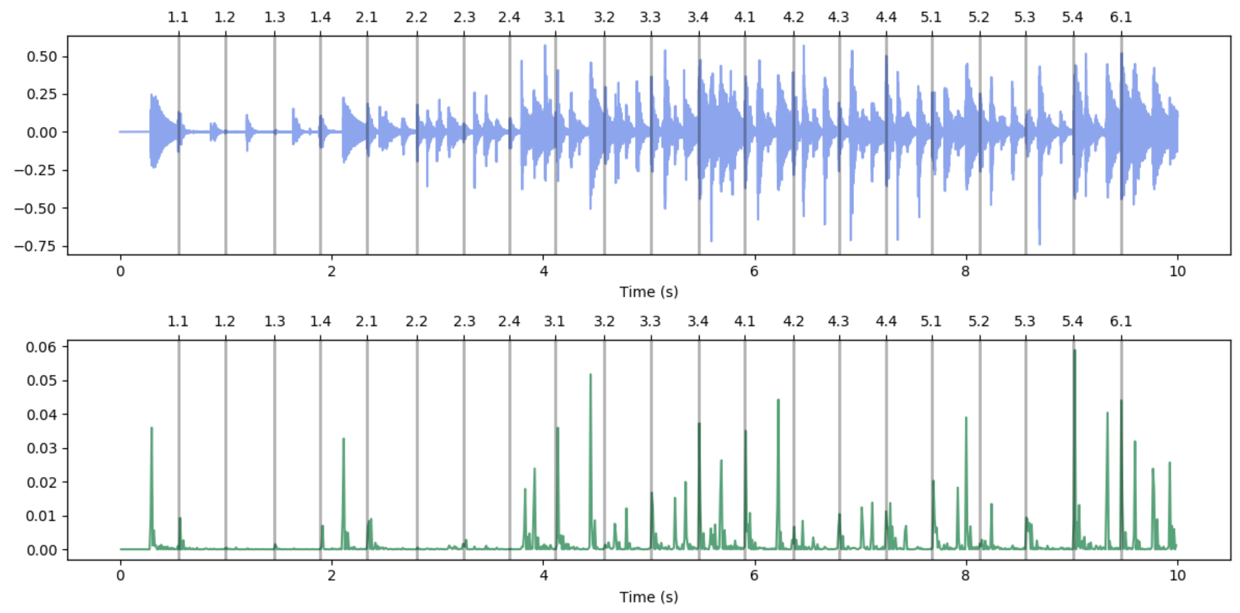
Then, we'll compute the accentuation feature.

Note: This example is tailored towards the rhythmic patterns of the lowest sounding of the three drum types taking part in the recording, so the analysis focuses on the low frequencies (20 to 200 Hz).

```
acce, times, _ = carat.features.accentuation_feature(y, sr, minfreq=20, maxfreq=200)
```

Finally we plot the audio waveform, the beat annotations and the accentuation feature values.

```
# plot waveform and accentuation feature
plt.figure(figsize=(12, 6))
# plot waveform
ax1 = plt.subplot(2, 1, 1)
carat.display.wave_plot(y, sr, ax=ax1, beats=beats, beat_labs=beat_labs)
# plot accentuation feature
ax2 = plt.subplot(2, 1, 2, sharex=ax1)
carat.display.feature_plot(acce, times, ax=ax2, beats=beats, beat_labs=beat_labs)
plt.tight_layout()

plt.show()
```



Total running time of the script: (0 minutes 2.605 seconds)

Note: Click *here* to download the full example code

1.3.4 Plot cluster centroids

This example shows how to plot centroids of the clusters of rhythmic patterns.

```
# Code source: Martín Rocamora
# License: MIT
```

Imports

- matplotlib for visualization

```
from __future__ import print_function
import matplotlib.pyplot as plt
import carat
```

We group rhythmic patterns into clusters and plot their centroids.

First, we'll load one of the audio files included in `carat`.

```
audio_path = carat.util.example_audio_file(num_file=1)

y, sr = carat.audio.load(audio_path)
```

Next, we'll load the annotations provided for the example audio file.

```
annotations_path = carat.util.example_beats_file(num_file=1)

beats, beat_labs = carat.annotations.load_beats(annotations_path)
downbeats, downbeat_labs = carat.annotations.load_downbeats(annotations_path)
```

Then, we'll compute the accentuation feature.

Note: This example is tailored towards the rhythmic patterns of the lowest sounding of the three drum types taking part in the recording, so the analysis focuses on the low frequencies (20 to 200 Hz).

```
acce, times, _ = carat.features.accentuation_feature(y, sr, minfreq=20, maxfreq=200)
```

Next, we'll compute the feature map.

```
n_beats = int(round(beats.size/downbeats.size))
n_tatums = 4

map_acce, _, _, _ = carat.features.feature_map(acce, times, beats, downbeats, n_
↪beats=n_beats,
                                               n_tatums=n_tatums)
```

Then, we'll group rhythmic patterns into clusters. This is done using the classical K-means method with Euclidean distance (but other clustering methods and distance measures can be used too).

Note: The number of clusters n_clusters has to be specified as an input parameter.

```
n_clusters = 4

cluster_labs, centroids, _ = carat.clustering.rhythmic_patterns(map_acce, n_
↪clusters=n_clusters)
```

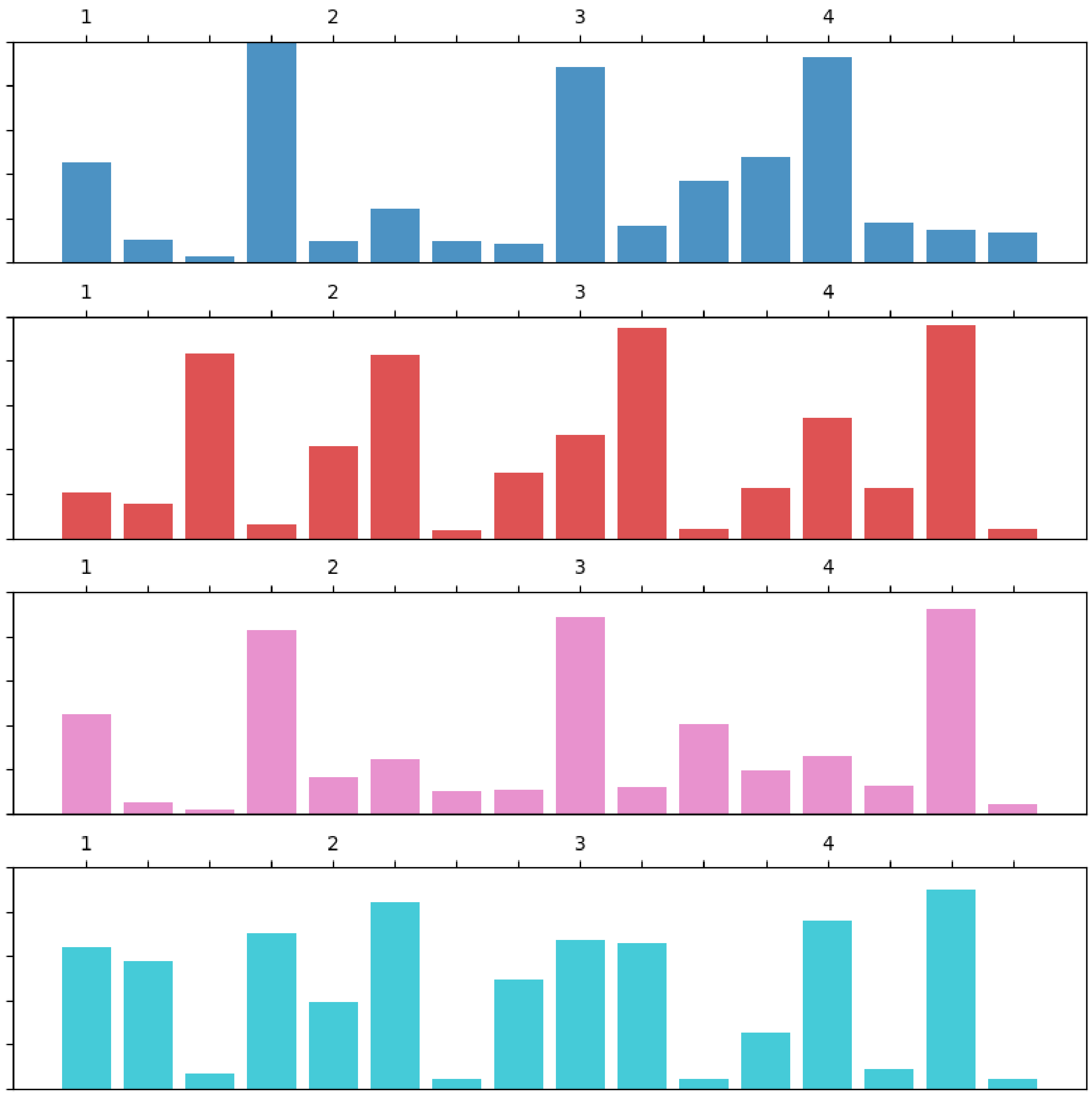
(continues on next page)

(continued from previous page)

```
```

Finally we plot the centroids of the clusters of rhythmic patterns.

```
fig = plt.figure(figsize=(8, 8))
carat.display.centroids_plot(centroids, n_tatums=n_tatums)

plt.tight_layout()

plt.show()
```



Total running time of the script: (0 minutes 22.871 seconds)

Note: Click *here* to download the full example code

1.3.5 Plot feature map clusters

This example shows how to cluster rhythmic patterns from a feature map.

This is based on the rhythmic patterns analysis proposed in [CIM2014].

```
# Code source: Martín Rocamora
# License: MIT
```

Imports

- matplotlib for visualization

```
from __future__ import print_function
import matplotlib.pyplot as plt
import carat
```

We group rhythmic patterns into clusters to aid the analysis of their differences and similarities.

First, we'll load one of the audio files included in `carat`.

```
audio_path = carat.util.example_audio_file(num_file=1)

y, sr = carat.audio.load(audio_path)
```

Next, we'll load the annotations provided for the example audio file.

```
annotations_path = carat.util.example_beats_file(num_file=1)

beats, beat_labs = carat.annotations.load_beats(annotations_path)
downbeats, downbeat_labs = carat.annotations.load_downbeats(annotations_path)
```

Then, we'll compute the accentuation feature.

Note: This example is tailored towards the rhythmic patterns of the lowest sounding of the three drum types taking part in the recording, so the analysis focuses on the low frequencies (20 to 200 Hz).

```
acce, times, _ = carat.features.accentuation_feature(y, sr, minfreq=20, maxfreq=200)
```

Next, we'll compute the feature map.

```
n_beats = int(round(beats.size/downbeats.size))
n_tatums = 4

map_acce, _, _, _ = carat.features.feature_map(acce, times, beats, downbeats, n_
↪beats=n_beats,
                                               n_tatums=n_tatums)
```

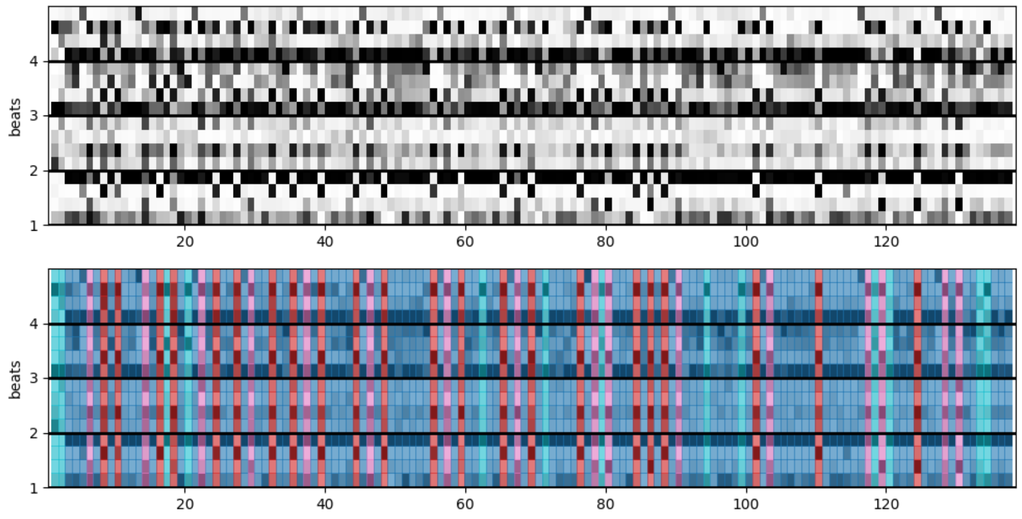
Then, we'll group rhythmic patterns into clusters. This is done using the classical K-means method with Euclidean distance (but other clustering methods and distance measures can be used too).

Note: The number of clusters n_clusters has to be specified as an input parameter.

```
n_clusters = 4

cluster_labs, centroids, _ = carat.clustering.rhythmic_patterns(map_acce, n_
↪clusters=n_clusters)
```

Finally we plot the feature map and the obtained clusters.

```
plt.figure(figsize=(12, 6))
# plot feature map
ax1 = plt.subplot(211)
carat.display.map_show(map_acce, ax=ax1, n_tatums=n_tatums)
# plot feature map with clusters in colors
ax2 = plt.subplot(212)
carat.display.map_show(map_acce, ax=ax2, n_tatums=n_tatums, clusters=cluster_labs)

plt.show()
```



Total running time of the script: (0 minutes 20.391 seconds)

Note: Click *here* to download the full example code

1.3.6 Plot low-dimensional embedding

This example shows how to plot a low-dimensional embedding of the rhythmic patterns.

This is based on the rhythmic patterns analysis proposed in [CIM2014].

```
# Code source: Martín Rocamora
# License: MIT
```

Imports

- matplotlib for visualization

- Axes3D from mpl_toolkits.mplot3d for 3D plots

```
from __future__ import print_function
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import carat
```

We compute the feature map of rhythmic patterns and we learn a manifold in a low–dimensional space. The patterns are they shown in the low–dimensional space before and after being grouped into clusters.

First, we'll load one of the audio files included in `carat`.

```
audio_path = carat.util.example_audio_file(num_file=1)

y, sr = carat.audio.load(audio_path)
```

Next, we'll load the annotations provided for the example audio file.

```
annotations_path = carat.util.example_beats_file(num_file=1)

beats, beat_labs = carat.annotations.load_beats(annotations_path)
downbeats, downbeat_labs = carat.annotations.load_downbeats(annotations_path)
```

Then, we'll compute the accentuation feature.

Note: This example is tailored towards the rhythmic patterns of the lowest sounding of the three drum types taking part in the recording, so the analysis focuses on the low frequencies (20 to 200 Hz).

```
acce, times, _ = carat.features.accentuation_feature(y, sr, minfreq=20, maxfreq=200)
```

Next, we'll compute the feature map.

```
n_beats = int(round(beats.size/downbeats.size))
n_tatums = 4

map_acce, _, _, _ = carat.features.feature_map(acce, times, beats, downbeats, n_
↪beats=n_beats,
                                               n_tatums=n_tatums)
```

Then, we'll group rhythmic patterns into clusters. This is done using the classical K-means method with Euclidean distance (but other clustering methods and distance measures can be used too).

Note: The number of clusters n_clusters has to be specified as an input parameter.

```
n_clusters = 4

cluster_labs, centroids, _ = carat.clustering.rhythmic_patterns(map_acce, n_
↪clusters=n_clusters)
```

Next, we compute a low-dimensional embedding of the rhythmic pattern. This is mainly done for visualization purposes. This representation can be useful to select the number of clusters, or to spot outliers. There are several approaches for dimensionality reduction among which isometric mapping, Isomap, was selected (other embedding methods can be also applied). Isomap is preferred since it is capable of keeping the levels of similarity among the original patterns after being mapped to the lower dimensional space. Besides, it allows the projection of new patterns onto the low-dimensional space.

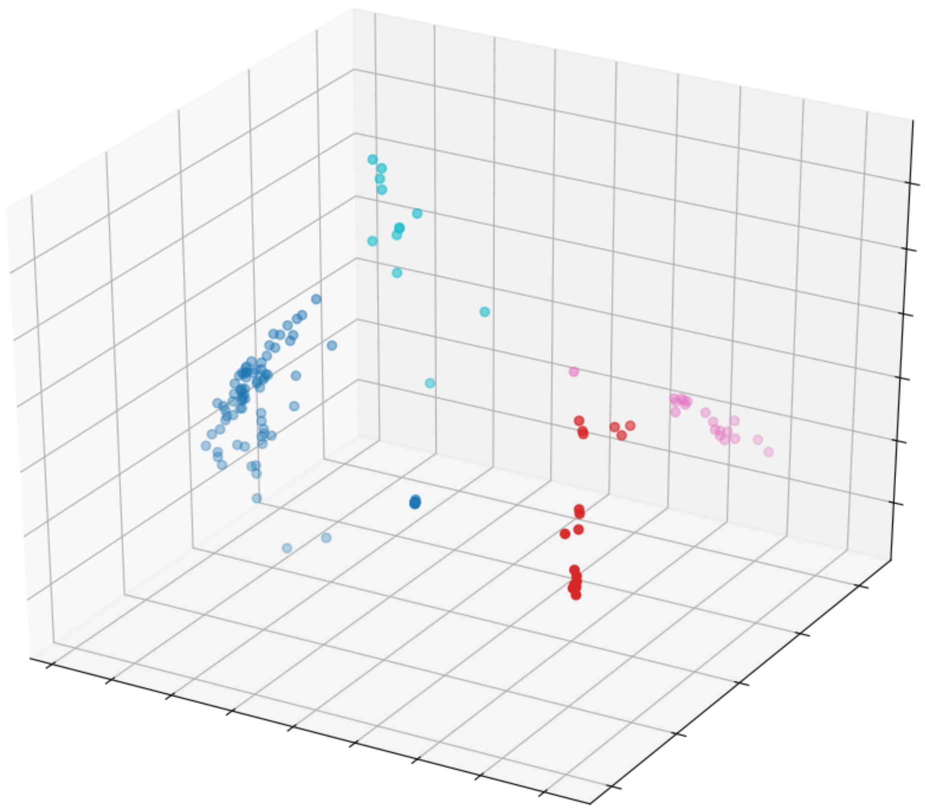
Note 1: You have to provide the number of dimensions to map on. Although any number of dimensions can be used to compute the embedding, only 2- and 3-dimensions plots are available (for obvious reasons).

Note 2: 3D plots need Axes3D from mpl_toolkits.mplot3d

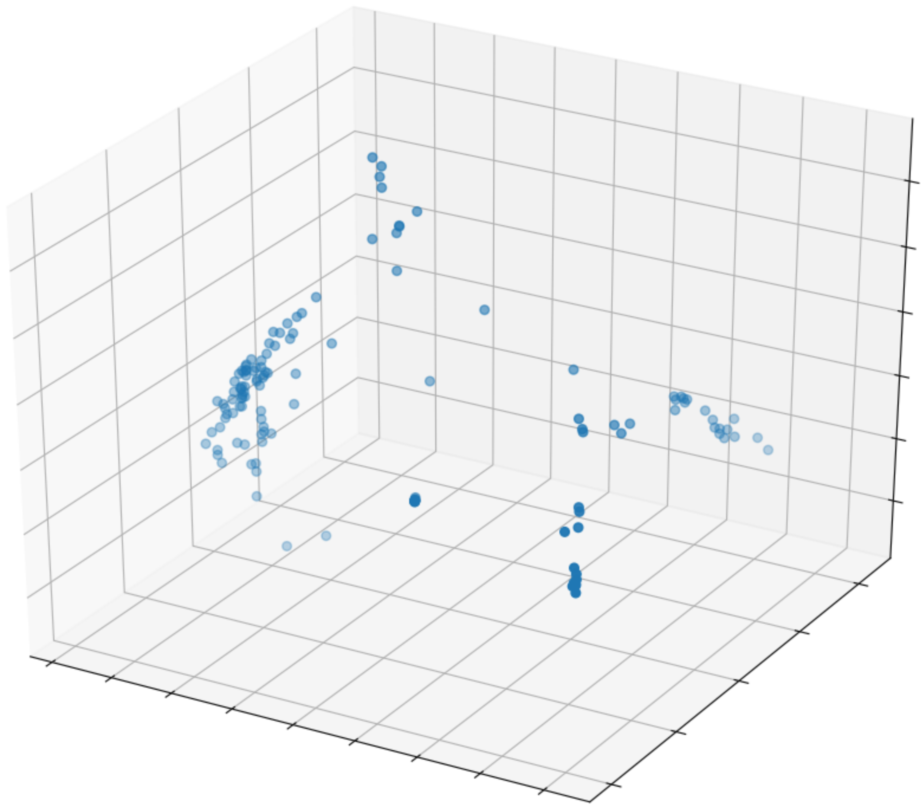
```
n_dims = 3
map_emb = carat.clustering.manifold_learning(map_acce, method='isomap', n_
↪components=n_dims)
```

Finally we plot the low-dimensional embedding of the rhythmic patterns and the clusters obtained.

```
fig1 = plt.figure(figsize=(10, 8))
ax1 = fig1.add_subplot(111, projection='3d')
carat.display.embedding_plot(map_emb, ax=ax1, clusters=cluster_labs, s=30)
plt.tight_layout()

fig2 = plt.figure(figsize=(10, 8))
ax2 = fig2.add_subplot(111, projection='3d')
carat.display.embedding_plot(map_emb, ax=ax2, s=30)
plt.tight_layout()

plt.show()
```



•



-

Total running time of the script: (0 minutes 20.776 seconds)

CHAPTER 2

API documentation

2.1 Annotations

2.1.1 Reading and writing annotations

load_beats(labels_file[, delimiter, . . .])	Load annotated beats from text (csv) file.
load_downbeats(labels_file[, delimiter, . . .])	Load annotated downbeats from text (csv) file.

carat.annotations.load_beats

carat.annotations.**load_beats**(*labels_file*, *delimiter=', '*, *times_col=0*, *labels_col=1*)

Load annotated beats from text (csv) file.

Parameters

labels_file [str] name (including path) of the input file

delimiter [str] string used as delimiter in the input file

times_col [int] column index of the time data

labels_col [int] column index of the label data

Returns

beat_times [np.ndarray] time instants of the beats

beat_labels [list] labels at the beats (e.g. 1.1, 1.2, etc)

Notes

It is assumed that the beat annotations are provided as a text file (csv). Apart from the time data (mandatory) a label can be given for each beat (optional). The time data is assumed to be given in seconds. The labels may indicate the beat number within the rhythm cycle (e.g. 1.1, 1.2, or 1, 2).

Examples

Load an included example file from the candombe dataset. http://www.eumus.edu.uy/candombe/datasets/ISMIR2015/

```
>>> annotations_file = carat.util.example_beats_file(num_file=1)
>>> beats, beat_labs = annotations.load_beats(annotations_file)
>>> beats[0]
0.548571428
>>> beat_labs[0]
'1.1'
```

Load an included example file from the samba dataset. http://www.smt.ufrj.br/~starel/datasets/brid.html

```
>>> annotations_file = carat.util.example_beats_file(num_file=2)
>>> beats, beat_labs = annotations.load_beats(annotations_file, delimiter=' ')
>>> beats
array([ 2.088,  2.559,  3.012,   3.48,  3.933,   4.41,  4.867,   5.32,
        5.771,  6.229,   6.69,  7.167,  7.633,  8.092,  8.545,   9.01,
         9.48,  9.943, 10.404, 10.865, 11.322, 11.79 , 12.251, 12.714,
       13.167, 13.624, 14.094, 14.559, 15.014, 15.473, 15.931,   16.4,
       16.865, 17.331, 17.788, 18.249, 18.706, 19.167, 19.643, 20.096,
       20.557, 21.018, 21.494, 21.945, 22.408, 22.869, 23.31 , 23.773,
       24.235, 24.692, 25.151, 25.608, 26.063, 26.52 ])
```

```
>>> beat_labs
['1', '2', '1', '2', '1', '2', '1', '2', '1', '2', '1', '2', '1', '2',
 '1', '2', '1', '2', '1', '2', '1', '2', '1', '2', '1', '2', '1', '2',
 '1', '2', '1', '2', '1', '2', '1', '2', '1', '2', '1', '2', '1', '2',
 '1', '2', '1', '2', '1', '2', '1', '2', '1', '2', '1', '2']
```

carat.annotations.load_downbeats

carat.annotations.**load_downbeats**(*labels_file*, *delimiter=', '*, *times_col=0*, *labels_col=1*, *downbeat_label='.1'*)

Load annotated downbeats from text (csv) file.

Parameters

labels_file [str] name (including path) of the input file

delimiter [str] string used as delimiter in the input file

times_col [int] column index of the time data

labels_col [int] column index of the label data

downbeat_label [str] string to look for in the label data to select downbeats

Returns

downbeat_times [np.ndarray] time instants of the downbeats

downbeat_labels [list] abels at the downbeats

Notes

It is assumed that the annotations are provided as a text file (csv). Apart from the time data (mandatory) a label can be given for each downbeat (optional). The time data is assumed to be given in seconds.

If a single file contains both beats and downbeats then the downbeat_label is used to select downbeats. The downbeats are those beats whose label has the given downbeat_label string. For instance the beat labels can be numbers, e.g. '1', '2'. Then, the downbeat_label is just '1'. This is the case for the BRID samba dataset. In the case of the candombe dataset, the beat labels indicate bar number and beat number. For instance, '1.1', '1.2', '1.3' and '1.4' are the four beats of the first bar. Hence, the string needed to indetify the downbeats is '.1'.

Examples

Load an included example file from the candombe dataset. http://www.eumus.edu.uy/candombe/datasets/ISMIR2015/

```
>>> annotations_file = carat.util.example_beats_file(num_file=1)
>>> downbeats, downbeat_labs = carat.annotations.load_downbeats(annotations_file)
>>> downbeats[:3]
array([0.54857143, 2.33265306, 4.11530612])
>>> downbeat_labs[:3]
['1.1', '2.1', '3.1']
```

Load an included example file from the samba dataset. http://www.smt.ufrj.br/~starel/datasets/brid.html

```
>>> annotations_file = carat.util.example_beats_file(num_file=2)
>>> downbeats, downbeat_labs = annotations.load_downbeats(annotations_file,
                                                          delimiter=' ', downbeat_
↪label='1')
>>> downbeats
array([ 2.088,  3.012,  3.933,  4.867,  5.771,  6.69 ,  7.633,  8.545,
        9.48 , 10.404, 11.322, 12.251, 13.167, 14.094, 15.014, 15.931,
       16.865, 17.788, 18.706, 19.643, 20.557, 21.494, 22.408,  23.31,
       24.235, 25.151, 26.063])
>>> downbeat_labs
['1', '1', '1', '1', '1', '1', '1', '1', '1', '1', '1', '1', '1', '1',
 '1', '1', '1', '1', '1', '1', '1', '1', '1', '1', '1', '1', '1']
```

2.2 Features

2.2.1 Accentuation features

`accentuation_feature`(signal, fs[, sum_flag, ...])	Compute accentuation feature from audio signal.
`feature_normalization`(feature, time, beats)	Local amplitude normalization of the feature signal.
`feature_time_quantize`(feature, time, tatums)	Time quantization of the feature signal to a tatum grid.

carat.features.accentuation_feature

carat.features.**accentuation_feature**(*signal, fs, sum_flag=True, log_flag=False, mel_flag=True, alpha=1000, maxfilt_flag=False, maxbins=3, **kwargs*)

Compute accentuation feature from audio signal.

Based on the log-power Mel spectrogram [1].

[1] Böck, Sebastian, and Gerhard Widmer.

"Maximum filter vibrato suppression for onset detection." 16th International Conference on Digital Audio Effects, Maynooth, Ireland. 2013.

This performs the following calculations to the input signal:

input->STFT->(Mel scale)->(Log)->(Max filtering)->Diff->HWR->(Sum)

Parenthesis denote optional steps.

Args:

- input: signal
- fs: sampling rate
- sum_flag (bool): true if the features are to be summed for each frame.
- log_flag (bool): true if the features energy are to be converted to dB.
- mel_flag (bool): true if the features are to be mapped in the Mel scale.
- alpha (int): compression parameter for dB conversion - log10(alpha*abs(S)+1).
- maxfilt_flag (bool): true if a maximum filtering is applied to the feature.
- maxbins (int): number of frequency bins for maximum filter size
- `**kw` : these keyword arguments are passed down to each of the functions used

Returns:

- feature (numpy array): feature values
- time (numpy array): time values

carat.features.feature_normalization

`carat.features.`**`feature_normalization`**(*feature*, *time*, *beats*, *n_tatums=4*, *pnorm=8*)

Local amplitude normalization of the feature signal.

Based on the feature map introduced in [1] and detailed in [2].

[1] Rocamora, Jure, Biscainho "Tools for detection and classification of piano drum patterns from candombe recordings." 9th Conference on Interdisciplinary Musicology (CIM), Berlin, Germany. 2014.

[2] Rocamora, Cancela, Biscainho "Information theory concepts applied to the analysis of rhythm in recorded music with recurrent rhythmic patterns." Journal of the AES, 67(4), 2019.

A local amplitude normalization is carried out to preserve intensity variations of the rhythmic patterns while discarding long-term fluctuations in dynamics. A p-norm within a local window is applied. The window width is proportional to the beat period.

Args:

- feature (numpy array): feature signal values
- time (numpy array): time instants of the feature values
- beats (numpy array): time instants of the tactus beats
- n_tatums (int): number of tatums per tactus beat
- pnorm (int): p-norm order for normalization

Returns:

- :

Raises:

- norm_feature (numpy array): normalized feature signal values

carat.features.feature_time_quantize

`carat.features.`**`feature_time_quantize`**(*feature*, *time*, *tatums*, *window=0.1*)

Time quantization of the feature signal to a tatum grid.

The feature signal is time-quantized to the rhythm metric structure by considering a grid of tatum pulses equally distributed within the tactus beats. The feature value assigned to each tatum time instant is obtained as the maximum value of the feature signal within a certain window centered at the tatum time instant. Default value for the total window lenght is 100 ms.

Args:

- feature (numpy array): feature signal values
- time (numpy array): time instants of the feature values
- tatums (numpy array): time instants of the tatum grid

Returns:

- :

Raises:

- quantized_feature (numpy array): time quantized feature signal values

2.2.2 Feature maps

`feature_map`(feature, time, beats, downbeats)	Compute feature map from accentuation feature signal.

carat.features.feature_map

`carat.features.`**`feature_map`**(*feature*, *time*, *beats*, *downbeats*, *n_beats=4*, *n_tatums=4*, *norm_flag=True*, *pnorm=8*, *window=0.1*)

Compute feature map from accentuation feature signal.

Based on the feature map introduced in [1].

[1] Rocamora, Jure, Biscainho "Tools for detection and classification of piano drum patterns from candombe recordings." 9th Conference on Interdisciplinary Musicology (CIM), Berlin, Germany. 2014.

The accentuation feature is organized into a feature map. First, the feature signal is time-quantized to the rhythm metric structure by considering a grid of tatum pulses equally distributed within the annotated beats. The corresponding feature value is taken as the maximum within window centered at the frame closest to each tatum instant. This yields feature vectors whose coordinates correspond to the tatum pulses of the rhythm cycle (or bar). Finally, a feature map of the cycle-length rhythmic patterns of the audio file is obtained by building a matrix whose columns are consecutive feature vectors.

Args:

- feature (numpy array): feature signal
- `**kw`: these keyword arguments are passed down to each of the functions used

Returns:

- :

Raises:

-

2.2.3 Time-frequency

`spectrogram`(signal, fs[, window_length, ...])	Calculates the Short-Time Fourier Transform a signal.
`melSpectrogram`(in_spec, in_time, in_freq[, ...])	This function converts a Spectrogram with linearly spaced frequency components to the Mel scale.

carat.features.spectrogram

`carat.features.`**`spectrogram`**(*signal*, *fs*, *window_length=0.02*, *hop=0.01*, *windowing_function=<function hanning at 0x7ff4ea50e048>*, *dft_length=None*, *zp_flag=False*)

Calculates the Short-Time Fourier Transform a signal.

Given an input signal, it calculates the DFT of frames of the signal and stores them in bi-dimensional Scipy array.

Args:

- window_len (float):length of the window in seconds (must be positive).
- window (callable): a callable object that receives the window length in samples and returns a numpy array containing the windowing function samples.
- hop (float): frame hop between adjacent frames in seconds.
- zp_flag (bool): a flag indicating if the *Zero-Phase Windowing* should be performed.

Returns:

- spec (numpy array): spectrogram data
- time (numpy array): time in seconds of each frame
- frequnecy (numpy array): frequency grid

carat.features.melSpectrogram

`carat.features.`**`melSpectrogram`**(*in_spec*, *in_time*, *in_freq*, *nfilts=40*, *minfreq=20*, *maxfreq=None*)

This function converts a Spectrogram with linearly spaced frequency components to the Mel scale.

> Given an input signal, it calculates the DFT of frames of the signal and stores them in bi-dimensional Scipy array.

Args:

- window_len (float): length of the window in seconds (must be positive).
- window (callable): a callable object that receives the window length in samples and returns a numpy array containing the windowing function samples.

Returns:

- spec (numpy array): mel-spectrogram data

- time (numpy array): time in seconds of each frame
- frequnecy (numpy array): frequency grid

2.2.4 Miscellaneous

generate_tatum_grid(beats, downbeats, n_tatums)	Generate tatum temporal grid from time instants of the tactus beats.
halfWaveRectification(in_signal)	Half-wave rectifies features.
calculateDelta(in_signal[, delta_filter_length])	This function calculates the delta coefficients of a given feature.
sumFeatures(in_signal)	This function sums all features along frames.

carat.features.generate_tatum_grid

`carat.features.`**`generate_tatum_grid`**(*beats*, *downbeats*, *n_tatums*)

Generate tatum temporal grid from time instants of the tactus beats.

A grid of tatum pulses is generated equally distributed within the given tactus beats. The grid is used to time quantize the feature signal to the rhythmic metric structure.

Parameters

labels_time (np.ndarray) [time instants of the tactus beats]

labels (list) [labels at the tactus beats (e.g. 1.1, 1.2, etc)]

Returns

tatum_time (np.ndarray) [time instants of the tatum beats]

carat.features.halfWaveRectification

`carat.features.`**`halfWaveRectification`**(*in_signal*)

Half-wave rectifies features.

All feature values below zero are assigned to zero.

Args:

- input: feature object
- delta_filter_length (int): length of the filter used to calculate the Delta coefficients. Must be an odd number.

Returns:

- output: numpy array

Raises:

- ValueError when the input features are complex.

carat.features.calculateDelta

`carat.features.`**`calculateDelta`**(*in_signal*, *delta_filter_length=3*)

This function calculates the delta coefficients of a given feature.

Args:

- input: input feature signal
- delta_filter_length (int): length of the filter used to calculate the Delta coefficients. Must be an odd number.

Returns:

- output: output feature signal

carat.features.sumFeatures

`carat.features.`**`sumFeatures`**(*in_signal*)

This function sums all features along frames.

Args:

- input: input feature signal

Returns:

- output: output feature signal

2.3 Clustering

2.3.1 Clustering and manifold learning

`rhythmic_patterns`(data[, n_clusters, method])	Clustering of rhythmic patterns from feature map.
`manifold_learning`(data[, method, …])	Manifold learning for dimensionality reduction of rhythmic patterns data.

carat.clustering.rhythmic_patterns

`carat.clustering.`**`rhythmic_patterns`**(*data*, *n_clusters=4*, *method='kmeans'*)

Clustering of rhythmic patterns from feature map.

Based on the feature map clustering analysis introduced in [1].

[1] Rocamora, Jure, Biscainho "Tools for detection and classification of piano drum patterns from candombe recordings." 9th Conference on Interdisciplinary Musicology (CIM), Berlin, Germany. 2014.

Parameters

data [np.ndarray] feature map

n_clusters [int] number of clusters

method [str] clustering method

Returns

c_labs [np.ndarray] cluster labels for each data point

c_centroids [np.ndarray] cluster centroids

c_method [sklearn.cluster] sklearn cluster method object

See also:

`sklearn.cluster.KMeans`

carat.clustering.manifold_learning

`carat.clustering.`**`manifold_learning`**(*data*, *method='isomap'*, *n_neighbors=7*, *n_components=3*)

Manifold learning for dimensionality reduction of rhythmic patterns data.

Based on the dimensionality reduction for rhythmic patterns introduced in [1].

[1] Rocamora, Jure, Biscainho "Tools for detection and classification of piano drum patterns from candombe recordings." 9th Conference on Interdisciplinary Musicology (CIM), Berlin, Germany. 2014.

Args:

- data (numpy array): feature map
- n_neighbors (int): number of neighbors for each dat point
- n_components (int): number of coordinates for the manifold

Returns:

- embedding(numpy array): lower-dimensional embedding of the data

Raises:

-

2.4 Display

`wave_plot`(y[, sr, x_axis, beats, beat_labs, ax])	Plot an audio waveform and beat labels (optinal).
`map_show`(data[, x_coords, y_coords, ax, . . .])	Display a feature map.
`feature_plot`(feature, time[, x_axis, beats, . . .])	Plot an audio waveform and beat labels (optinal).
`embedding_plot`(data[, clusters, ax])	Display an 2D or 3D embedding of the rhythmic patterns data.
`centroids_plot`(centroids[, n_tatums, ax_list])	Plot centroids of rhythmic patterns clusters.
`plot_centroid`(centroid[, n_tatums, ax])	Plot centroid of a rhythmic patterns cluster.

2.4.1 carat.display.wave_plot

`carat.display.`**`wave_plot`**(*y*, *sr=22050*, *x_axis='time'*, *beats=None*, *beat_labs=None*, *ax=None*, ***kwargs*)

Plot an audio waveform and beat labels (optinal).

Parameters

y [np.ndarray] audio time series

sr [number > 0 [scalar]] sampling rate of *y*

x_axis [str {'time', 'off', 'none'} or None] If 'time', the x-axis is given time tick-marks.

ax [matplotlib.axes.Axes or None] Axes to plot on instead of the default *plt.gca()*.

kwargs Additional keyword arguments to *matplotlib*.

2.4.2 carat.display.map_show

carat.display.**map_show**(*data*, *x_coords=None*, *y_coords=None*, *ax=None*, *n_tatums=4*, *clusters=None*, ***kwargs*)

Display a feature map.

Parameters

data [np.ndarray] Feature map to display

x_coords [np.ndarray [shape=data.shape[1]+1]]

y_coords [np.ndarray [shape=data.shape[0]+1]] Optional positioning coordinates of the input data.

ax [matplotlib.axes.Axes or None] Axes to plot on instead of the default *plt.gca()*.

n_tatums [int] Number of tatums (subdivisions) per tactus beat

clusters [np.ndarray] Array indicating cluster number for each pattern of the input data. If provided (not None) the clusters area displayed with colors.

kwargs [additional keyword arguments] Arguments passed through to `matplotlib.pyplot.pcolormesh`.

By default, the following options are set:

- *cmap=gray_r*
- *rasterized=True*
- *edgecolors='None'*
- *shading='flat'*

Returns

axes The axis handle for the figure.

See also:

`matplotlib.pyplot.pcolormesh`

2.4.3 carat.display.feature_plot

carat.display.**feature_plot**(*feature*, *time*, *x_axis='time'*, *beats=None*, *beat_labs=None*, *ax=None*, ***kwargs*)

Plot an audio waveform and beat labels (optinal).

Parameters

feature [np.ndarray] feature time series

time [np.ndarray] time instant of the feature values

x_axis [str {'time', 'off', 'none'} or None] If 'time', the x-axis is given time tick-marks.

ax [matplotlib.axes.Axes or None] Axes to plot on instead of the default *plt.gca()*.

kwargs Additional keyword arguments to *matplotlib*.

2.4.4 carat.display.embedding_plot

`carat.display.`**`embedding_plot`**(*data*, *clusters=None*, *ax=None*, ***kwargs*)

Display an 2D or 3D embedding of the rhythmic patterns data.

Parameters

data [np.ndarray] Low-embedding data points

ax [matplotlib.axes.Axes or None] Axes to plot on instead of the default *plt.gca()*.

clusters [np.ndarray] Array indicating cluster number for each point of the input data. If provided (not None) the clusters area displayed with colors.

kwargs [additional keyword arguments] Arguments passed through to `matplotlib.pyplot.pcolormesh`.

Returns

axes The axis handle for the figure.

See also:

`matplotlib.pyplot.pcolormesh`

2.4.5 carat.display.centroids_plot

`carat.display.`**`centroids_plot`**(*centroids*, *n_tatums=4*, *ax_list=None*, ***kwargs*)

Plot centroids of rhythmic patterns clusters.

Parameters

centroids: np.ndarray centroids of the rhythmic patterns clusters

n_tatums [int] Number of tatums (subdivisions) per tactus beat

ax_list [list of matplotlib.axes.Axes or None, one element per centroid] Axes to plot on instead of the default *plt.gca()*.

kwargs Additional keyword arguments to *matplotlib*.

Returns

ax [list of matplotlib.axes.Axes]

2.4.6 carat.display.plot_centroid

`carat.display.`**`plot_centroid`**(*centroid*, *n_tatums=4*, *ax=None*, ***kwargs*)

Plot centroid of a rhythmic patterns cluster.

Parameters

centroid [np.ndarray] centroid feature values

n_tatums [int] Number of tatums (subdivisions) per tactus beat

ax [matplotlib.axes.Axes or None] Axes to plot on instead of the default *plt.gca()*.

kwargs Additional keyword arguments to *matplotlib*.

2.5 Util

2.5.1 Signal segmentation

`segmentSignal`(signal, window_len, hop)	Segmentation of an array-like input:
`beat2signal`(y, time, beats, ind_beat)	Get the signal fragment corresponding to a beat given by index ind_beat.
`get_time_segment`(y, time, time_ini, time_end)	Get a segment of an array, given by initial and ending indexes.

carat.util.segmentSignal

`carat.util.`**`segmentSignal`**(*signal*, *window_len*, *hop*)

Segmentation of an array-like input:

Given an array-like, this function calculates the DFT of frames of the signal and stores them in bi-dimensional Scipy array.

Args: signal (array-like): object to be windowed. Must be a one-dimensional array-like object. window_len (int): window size in samples. hop (int): frame hop between adjacent frames in seconds.

Returns: A 2-D numpy array containing the windowed signal. Each element of this array X can be defined as:

X[m,n] = x[n+Hm]

where, H is the HOP in samples, 0<=n<=N, N = window_len, and 0<m<floor(((len(x)-N)/H)+1).

Raises: AttributeError if signal is not one-dimensional. ValueError if window_len or hop are not strictly positives.

carat.util.beat2signal

`carat.util.`**`beat2signal`**(*y*, *time*, *beats*, *ind_beat*)

Get the signal fragment corresponding to a beat given by index ind_beat. If instead of beats, downbeats are used, then a bar is returned.

Args: y (numpy array): signal array. Must be a one-dimensional array. time (numpy array): corresponding time. Must be a one-dimensional array. beats (numpy array): time instants of the beats. ind_beat (int): index of the desired beat.

Returns: beat_segment (numpy array): segment of the signal corresponding to the beat.

Raises: AttributeError if y or time is not a one-dimensional numpy array. ValueError if ind_beat fall outside the beats bounds.

carat.util.get_time_segment

`carat.util.`**`get_time_segment`**(*y*, *time*, *time_ini*, *time_end*)

Get a segment of an array, given by initial and ending indexes.

Args: y (numpy array): signal array. Must be a one-dimensional array. time (numpy array): corresponding time. Must be a one-dimensional array. time_ini (int): initial time value. time_end (int): ending time value.

Returns: segment (numpy array): segment of the signal.

Raises: AttributeError if y or time is not a one-dimensional numpy array. ValueError if idx_ini or idx_end fall outside the signal bounds. ValueError if idx_ini >= idx_end.

2.5.2 Time-frequency

`STFT`(x, window_length, hop[, ...])	Calculates the Short-Time Fourier Transform a signal.
`fft2mel`(freq, nfilts, minfreq, maxfreq)	This method returns a 2-D Numpy array of weights that map a linearly spaced spectrogram to the Mel scale.
`hz2mel`(f_hz)	Converts a given frequency in Hz to the Mel scale.
`mel2hz`(z_mel)	Converts a given frequency in the Mel scale to Hz scale.

carat.util.STFT

`carat.util.`**`STFT`**(*x*, *window_length*, *hop*, *windowing_function=<function hanning at 0x7ff4ea50e048>*, *dft_length=None*, *zp_flag=False*)

Calculates the Short-Time Fourier Transform a signal.

Given an input signal, it calculates the DFT of frames of the signal and stores them in bi-dimensional Scipy array.

Args: window_len (float): length of the window in seconds (must be positive). window (callable): a callable object that receives the window length in samples and returns a numpy array containing the windowing function samples. hop (float): frame hop between adjacent frames in seconds. final_time (positive integer): time (in seconds) up to which the spectrogram is calculated. zp_flag (bool): a flag indicating if the *Zero-Phase Windowing* should be performed.

Returns: spec: numpy array time: numpy array frequency: numpy array

Raises:

carat.util.fft2mel

`carat.util.`**`fft2mel`**(*freq*, *nfilts*, *minfreq*, *maxfreq*)

This method returns a 2-D Numpy array of weights that map a linearly spaced spectrogram to the Mel scale.

Args: freq (1-D Numpy array): frequency of the components of the DFT. nfilts (): number of output bands. minfreq (): frequency of the first MEL coefficient. maxfreq (): frequency of the last MEL coefficient.

Returns: The center frequencies in Hz of the Mel bands.

carat.util.hz2mel

`carat.util.`**`hz2mel`**(*f_hz*)

Converts a given frequency in Hz to the Mel scale.

Args: f_hz (Numpy array): Array containing the frequencies in HZ that should be converted.

Returns: A Numpy array (of same shape as f_zh) containing the converted frequencies.

carat.util.mel2hz

`carat.util.`**`mel2hz`**(*z_mel*)

Converts a given frequency in the Mel scale to Hz scale.

Args: z_mel (Numpy array): Array of frequencies in the Mel scale that should be converted.

Returns: A Numpy array (of same shape as z_mel) containing the converted frequencies.

2.5.3 Miscellaneous

`example_audio_file`([num_file])	Get the path to an included audio example file.
`example_beats_file`([num_file])	Get the path to an included example file of beats annotations.
`find_nearest`(array, value)	Find index of the nearest value of an array to a given value
`deltas`(x[, w])	this function estimates the derivative of x

carat.util.example_audio_file

`carat.util.`**`example_audio_file`**(*num_file=None*)

Get the path to an included audio example file.

Parameters

num_file [int] Number to select among the example files available.

Returns

filename [str] Path to the audio example file included with `carat`.

Examples

```
>>> # Load the waveform from the default example track
>>> y, sr = carat.audio.load(carat.util.example_audio_file())
```

```
>>> # Load 10 seconds of the waveform from the example track number 1
>>> y, sr = carat.audio.load(carat.util.example_audio_file(num_file=1),␣
↪duration=10.0))
```

```
>>> # Load the waveform from the example track number 2
>>> y, sr = carat.audio.load(carat.util.example_audio_file(num_file=2))
```

carat.util.example_beats_file

`carat.util.`**`example_beats_file`**(*num_file=None*)

Get the path to an included example file of beats annotations.

Parameters

num_file [int] Number to select among the example files available.

Returns

filename [str] Path to the beats annotations example file included with `carat`.

Examples

```
>>> # Load beats and downbeats from the example audio file number 1
>>> beats, b_labs = carat.load_beats(carat.util.example_beats_file(num_file=1))
>>> downbeats, d_labs = carat.load_downbeats(carat.util.example_beats_file(num_
↪file=1))
```

carat.util.find_nearest

`carat.util.`**`find_nearest`**(*array*, *value*)

Find index of the nearest value of an array to a given value

Parameters

array (numpy.ndarray) [array]

value (float) [value]

Returns

idx (int) [index of nearest value in the array]

carat.util.deltas

`carat.util.`**`deltas`**(*x*, *w=3*)

this function estimates the derivative of x

Bibliography

[CIM2014] *Tools for detection and classification of piano drum patterns from candombe recordings.* Rocamora, Jure, Biscainho. 9th Conference on Interdisciplinary Musicology (CIM), Berlin, Germany. 2014.

Python Module Index

Index