Tutorial

This section covers the fundamentals of developing with *librosa*, including a package overview, basic and advanced usage, and integration with the *scikit-learn* package. We will assume basic familiarity with Python and NumPy/SciPy.

Overview

The *librosa* package is structured as collection of submodules:

- librosa
  - **librosa.beat**

    Functions for estimating tempo and detecting beat events.

  - **librosa.core**

    Core functionality includes functions to load audio from disk, compute various spectrogram representations, and a variety of commonly used tools for music analysis. For convenience, all functionality in this submodule is directly accessible from the top-level `librosa.*` namespace.

  - **librosa.decompose**

    Functions for harmonic-percussive source separation (HPSS) and generic spectrogram decomposition using matrix decomposition methods implemented in *scikit-learn*.

  - **librosa.display**

    Visualization and display routines using `matplotlib`.

  - **librosa.effects**

    Time-domain audio processing, such as pitch shifting and time stretching. This submodule also provides time-domain wrappers for the *decompose*submodule.

  - **librosa.feature**

    Feature extraction and manipulation. This includes low-level feature extraction, such as chromagrams, pseudo-constant-Q (log-frequency) transforms, Mel spectrogram, MFCC, and tuning estimation. Also provided are feature manipulation methods, such as delta features, memory embedding, and event-synchronous feature alignment.

  - **librosa.filters**

    Filter-bank generation (chroma, pseudo-CQT, CQT, etc.). These are primarily internal functions used by other parts of *librosa*.

  - **librosa.onset**

    Onset detection and onset strength computation.

  - **librosa.output**

    Text- and wav-file output. *(Deprecated)*

- **librosa.segment**

  Functions useful for structural segmentation, such as recurrence matrix construction, time-lag representation, and sequentially constrained clustering.

- **librosa.sequence**

  Functions for sequential modeling. Various forms of Viterbi decoding, and helper functions for constructing transition matrices.

- **librosa.util**

  Helper utilities (normalization, padding, centering, etc.)

Quickstart

Before diving into the details, we'll walk through a brief example program

```
# Beat tracking example
from __future__ import print_function
import librosa

# 1. Get the file path to the included audio example
filename = librosa.util.example_audio_file()

# 2. Load the audio as a waveform `y`
#    Store the sampling rate as `sr`
y, sr = librosa.load(filename)

# 3. Run the default beat tracker
tempo, beat_frames = librosa.beat.beat_track(y=y, sr=sr)

print('Estimated tempo: {:.2f} beats per minute'.format(tempo))

# 4. Convert the frame indices of beat events into timestamps
beat_times = librosa.frames_to_time(beat_frames, sr=sr)
```

The first step of the program:

```
filename = librosa.util.example_audio_file()
```

gets the path to the audio example file included with *librosa*. After this step, `filename` will be a string variable containing the path to the example audio file. The example is encoded in OGG Vorbis format, so you will need the appropriate codec installed for audioread.

The second step:

```
y, sr = librosa.load(filename)
```

loads and decodes the audio as a time series `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 `librosa.load()`.

Next, we run the beat tracker:

```
tempo, beat_frames = librosa.beat.beat_track(y=y, sr=sr)
```

The output of the beat tracker is an estimate of the tempo (in beats per minute), and an array of frame numbers corresponding to detected beat events.

Frames here correspond to short windows of the signal (`y`), each separated by `hop_length = 512` samples. Since v0.3, *librosa* uses centered frames, so that the *k*th frame is centered around sample `k * hop_length`.

The next operation converts the frame numbers `beat_frames` into timings:

```
beat_times = librosa.frames_to_time(beat_frames, sr=sr)
```

Now, `beat_times` will be an array of timestamps (in seconds) corresponding to detected beat events.

The contents of `beat_times` should look something like this:

```
7.43
8.29
9.218
10.124
...
```

Advanced usage

Here we'll cover a more advanced example, integrating harmonic-percussive separation, multiple spectral features, and beat-synchronous feature aggregation.

```python
# Feature extraction example
import numpy as np
import librosa

# Load the example clip
y, sr = librosa.load(librosa.util.example_audio_file())

# Set the hop length; at 22050 Hz, 512 samples ~= 23ms
hop_length = 512

# Separate harmonics and percussives into two waveforms
y_harmonic, y_percussive = librosa.effects.hpss(y)

# Beat track on the percussive signal
tempo, beat_frames = librosa.beat.beat_track(y=y_percussive,
                                             sr=sr)

# Compute MFCC features from the raw signal
mfcc = librosa.feature.mfcc(y=y, sr=sr, hop_length=hop_length, n_mfcc=13)

# And the first-order differences (delta features)
mfcc_delta = librosa.feature.delta(mfcc)

# Stack and synchronize between beat events
# This time, we'll use the mean value (default) instead of median
beat_mfcc_delta = librosa.util.sync(np.vstack([mfcc, mfcc_delta]),
                                    beat_frames)

# Compute chroma features from the harmonic signal
chromagram = librosa.feature.chroma_cqt(y=y_harmonic,
                                        sr=sr)

# Aggregate chroma features between beat events
# We'll use the median value of each feature between beat frames
beat_chroma = librosa.util.sync(chromagram,
                                beat_frames,
                                aggregate=np.median)

# Finally, stack all beat-synchronous features together
beat_features = np.vstack([beat_chroma, beat_mfcc_delta])
```

This example builds on tools we've already covered in the quickstart example, so here we'll focus just on the new parts.

The first difference is the use of the effects module for time-series harmonic-percussive separation:

```
y_harmonic, y_percussive = librosa.effects.hpss(y)
```

The result of this line is that the time series `y` has been separated into two time series, containing the harmonic (tonal) and percussive (transient) portions of the signal. Each of `y_harmonic` and `y_percussive` have the same shape and duration as `y`.

The motivation for this kind of operation is two-fold: first, percussive elements tend to be stronger indicators of rhythmic content, and can help provide more stable beat tracking results; second, percussive elements can pollute tonal feature representations (such as chroma) by contributing energy across all frequency bands, so we'd be better off without them.

Next, we introduce the feature module and extract the Mel-frequency cepstral coefficients from the raw signal `y`:

```
mfcc = librosa.feature.mfcc(y=y, sr=sr, hop_length=hop_length, n_mfcc=13)
```

The output of this function is the matrix `mfcc`, which is an *numpy.ndarray* of size `(n_mfcc, T)` (where `T` denotes the track duration in frames). Note that we use the same `hop_length` here as in the beat tracker, so the detected `beat_frames` values correspond to columns of `mfcc`.

The first type of feature manipulation we introduce is `delta`, which computes (smoothed) first-order differences among columns of its input:

```
mfcc_delta = librosa.feature.delta(mfcc)
```

The resulting matrix `mfcc_delta` has the same shape as the input `mfcc`.

The second type of feature manipulation is `sync`, which aggregates columns of its input between sample indices (e.g., beat frames):

```
beat_mfcc_delta = librosa.util.sync(np.vstack([mfcc, mfcc_delta]),
                                    beat_frames)
```

Here, we've vertically stacked the `mfcc` and `mfcc_delta` matrices together. The result of this operation is a matrix `beat_mfcc_delta` with the same number of rows as its input, but the number of columns depends on `beat_frames`. Each column `beat_mfcc_delta[:, k]` will be the *average* of input columns between `beat_frames[k]` and `beat_frames[k+1]`. (`beat_frames` will be expanded to span the full range `[0, T]` so that all data is accounted for.)

Next, we compute a chromagram using just the harmonic component:

```
chromagram = librosa.feature.chroma_cqt(y=y_harmonic,
                                        sr=sr)
```

After this line, `chromagram` will be a *numpy.ndarray* of size `(12, T)`, and each row corresponds to a pitch class (e.g., *C*, *C#*, etc.). Each column of `chromagram` is normalized by its peak value, though this behavior can be overridden by setting the `norm` parameter.

Once we have the chromagram and list of beat frames, we again synchronize the chroma between beat events:

```
beat_chroma = librosa.util.sync(chromagram,
                                beat_frames,
                                aggregate=np.median)
```

This time, we've replaced the default aggregate operation (*average*, as used above for MFCCs) with the *median*. In general, any statistical summarization function can be supplied here, including *np.max()*, *np.min()*, *np.std()*, etc.

Finally, the all features are vertically stacked again:

```
beat_features = np.vstack([beat_chroma, beat_mfcc_delta])
```

resulting in a feature matrix `beat_features` of dimension `(12 + 13 + 13, # beat intervals)`.

More examples

More example scripts are provided in the advanced examples section.