scaper Documentation

Release 1.6.5

Justin Salamon & Duncan MacConnell

Apr 23, 2021

Contents

Scaper is a library for soundscape synthesis and augmentation.

For a quick introduction to using scaper, please refer to the *Scaper tutorial*. For a detailed description of scaper and its applications check out the scaper-paper:

Scaper: A library for soundscape synthesis and augmentation J. Salamon, D. MacConnell, M. Cartwright, P. Li, and J. P. Bello In IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA), New Paltz, NY, USA, Oct. 2017.

CHAPTER 1

Getting started

1.1 Installation instructions

1.1.1 Non-python dependencies

Scaper has one non-python dependency: - FFmpeg: https://ffmpeg.org/

If you are installing Scaper on Windows, you will also need: - SoX: http://sox.sourceforge.net/

On Linux/macOS SoX is replaced by [SoxBindings](https://github.com/pseeth/soxbindings) which is significantly faster, giving better runtime performance in Scaper. On these platforms SoxBindings is installed automatically when calling *pip install scaper* (see below).

On macOS ffmpeg can be installed using homebrew:

```
>>> brew install ffmpeg
```

On linux you can use your distribution's package manager, e.g. on Ubuntu (15.04 "Vivid Vervet" or newer):

```
>>> sudo apt-get install ffmpeg
```

NOTE: on earlier versions of Ubuntu ffmpeg may point to a Libav binary which is not the correct binary. If you are using anaconda, you can install the correct version by calling:

```
>>> conda install -c conda-forge ffmpeg
```

Otherwise, you can obtain a static binary from the ffmpeg website.

On Windows you can use the provided installation binaries:

- SoX: https://sourceforge.net/projects/sox/files/sox/
- FFmpeg: https://ffmpeg.org/download.html#build-windows

1.1.2 Installing Scaper

The simplest way to install `scaper` is by using `pip`, which will also install the required dependencies if needed. To install `scaper` using `pip`, simply run

```
>>> pip install scaper
```

To install the latest version of scaper from source:

1. Clone or pull the lastest version:

```
>>> git clone git@github.com:justinsalamon/scaper.git
```

2. Install using pip to handle python dependencies:

```
>>> cd scaper
>>> pip install -e .
```

1.2 Scaper tutorial

1.2.1 Introduction

Welcome to the scaper tutorial! In this tutorial, we'll explain how scaper works and show how to use scaper to synthesize soundscapes and generate corresponding annotation files.

1.2.2 Organize your audio files (source material)

Scaper creates new soundscapes by combining and transforming a set of existing audio files, which we'll refer to as the *source material*. By combining/transforming the source material in different ways, scaper can create an infinite number of different soundscapes from the same source material. The source material is comprised of two groups of files: *background* files and *foreground* files:

- **Background files**: are used to create the background of the soundscape, and should contain audio material that is perceived as a single holistic sound which is more distant, ambiguous, and texture-like (e.g. the "hum" or "drone" of an urban environment, or "wind and rain" sounds in a natural environment). Importantly, background files should not contain salient sound events.
- **Foreground files**: are used to create sound events. Each foreground audio file should contain a single sound event (short or long) such as a car honk, an animal vocalization, continuous speech, a siren or an idling engine. Foreground files should be as clean as possible with no background noise and no silence before/after the sound event.

The source material must be organized as follows: at the top level, you need a `background` folder and a `foreground` folder. Within each of these, scaper expects a folder for each category (label) of sounds. Example categories of background sounds include "street", "forest" or "park". Example categories of foreground sounds include "speech", "car_honk", "siren", "dog_bark", "bird_song", "idling_engine", etc. Within each category folder, scaper expects WAV files of that category: background files should contain a single ambient recording, and foreground files should contain a single sound event. The filename of each audio file is not important as long as it ends with `.wav`.

Here's an example of a valid folder structure for the source material:

```
- foreground
    - siren
        - siren1.wav
```

(continues on next page)

(continued from previous page)

```
        - siren2.wav
        - some_siren_sound.wav
    - car_honk
        - honk.wav
        - beep.wav
    - human_voice
        - hello_world.wav
        - shouting.wav
- background
    - park
        - central_park.wav
    - street
        - quiet_street.wav
        - loud_street.wav
```

EXAMPLE SOURCE MATERIAL can be obtained by downloading the scaper repository (approx. 50mb). The audio can be found under `scaper-master/tests/data/audio` (`audio` contains two subfolders, `background` and `foreground`). For the remainder of this tutorial, we'll assume you've downloaded this material and copied the `audio` folder to your home directory. If you copy it somewhere else (or use different source material), be sure to change the paths to the `foreground_folder` and `background_folder` in the example code below.

1.2.3 Create a Scaper object

The first step is to create a `Scaper` object:

```
import scaper
import os

path_to_audio = os.path.expanduser('~/audio')

soundscape_duration = 10.0
seed = 123
foreground_folder = os.path.join(path_to_audio, 'foreground')
background_folder = os.path.join(path_to_audio, 'background')
sc = scaper.Scaper(soundscape_duration, foreground_folder, background_folder)
sc.ref_db = -20
```

We need to supply three arguments to create a `Scaper` object:

1. The desired duration: all soundscapes generated by this Scaper object will have this duration.
2. The path to the foreground folder.
3. The path to the background folder.

If you're not sure what the foreground and background folders are, please see *Organize your audio files (source material)*.

Finally, we set the reference level `sc.ref_db`, i.e. the loudnes of the background, measured in LUFS. Later when we add foreground events, we'll have to specify an `snr` (signal-to-noise ratio) value, i.e. by how many decibels (dB) should the foreground event be louder (or softer) with respect to the background level specified by `sc.ref_db`.

Seeding the Scaper object for reproducibility

A further argument can be specified to the `Scaper` object:

- The random state: this can be either a numpy.random.RandomState object or an integer. In the latter case, a random state will be constructed. The random state is what will be used for drawing from any distributions. If the audio kept in all of the folders is exactly the same and the random state is fixed between runs, the same soundscape will be generated both times. If you don't define any random state or set seed to None, runs will be random and not reproducible. You can use np.random.get_state() to reproduce the run after the fact by recording the seed that was used somewhere.

This can be specified like so (e.g. for a random seed of 123):

```
seed = 123
sc = scaper.Scaper(soundscape_duration, foreground_folder, background_folder,
                   random_state=seed)
sc.ref_db = -20
```

If the random state is not specified, it defaults to the old behavior which just uses the RandomState used by np.random. You can also set the random state after creation via `Scaper.set_random_state`. Alternatively, you can set the random state directly:

```
import numpy as np
seed = np.random.RandomState(123)
sc = scaper.Scaper(soundscape_duration, foreground_folder, background_folder,
                   random_state=seed)
sc.ref_db = -20
```

1.2.4 Adding a background and foreground sound events

Adding a background

Next, we can optionally add a background track to our soundscape:

```
sc.add_background(label=('const', 'park'),
                  source_file=('choose', []),
                  source_time=('const', 0))
```

To add a background we have to specify:

- `label`: the label (category) of background, which has to match the name of one of the subfolders in our background folder (in our example "park" or "street").
- `source_file`: the path to the specific audio file to be used.
- `source_time`: the time in the source file from which to start the background.

Note how in the example above we do not specify these values directly by providing strings or floats, but rather we provide each arugment with a tuple. These tuples are called **distribution tuples** and are used in scaper for specifying all sound event parameters. Let's explain:

Distribution tuples

One of the powerful things about scaper is that it allows you to define a soundscape in a probabilistic way. That is, rather than specifying constant (hard coded) values for each sound event, you can specify a distribution of values to sample from. Later on, when we call `sc.generate()`, a soundscape will be "instantiated" by sampling a value for each distribution tuple in each sound event (foreground and background). Every time we call `sc.generate()`, a new value will be sampled for each distribution tuple, resulting in a different soundscape.

The distribution tuples currently supported by scaper are:

- `('const', value)`: a constant, given by `value`.
- `('choose', list)`: uniformly sample from a finite set of values given by `list`.
- `('uniform', min, max)`: sample from a uniform distribution between `min` and `max`.
- `('normal', mean, std)`: sample from a normal distribution with mean `mean` and standard deviation `std`.
- `('truncnorm', mean, std, min, max)`: sample from a truncated normal distribution with mean `mean` and standard deviation `std`, limited to values between `min` and `max`.

Special cases: the `label` and `source_file` parameters in `sc.add_background()` (and as we'll see later `sc.add_event()` as well) must be specified using either the `const` or `choose` distribution tuples. When using `choose`, these two parameters (and only these) can also accept a special version of the `choose` tuple in the form `('choose', [])`, i.e. with an empty list. In this case, scaper will use the file structure in the foreground and background folders to automatically populate the list with all valid labels (in the case of the `label` parameter) and all valid filenames (in the case of the `source_file` parameter).

Adding a foreground sound event

Next, we can add foreground sound events. Let's add one to start with:

```
sc.add_event(label=('const', 'siren'),
             source_file=('choose', []),
             source_time=('const', 0),
             event_time=('uniform', 0, 9),
             event_duration=('truncnorm', 3, 1, 0.5, 5),
             snr=('normal', 10, 3),
             pitch_shift=('uniform', -2, 2),
             time_stretch=('uniform', 0.8, 1.2))
```

A foreground sound event requires several additional parameters compared to a background event. The full set of parameters is:

- `label`: the label (category) of foreground event, which has to match the name of one of the subfolders in our foreground folder (in our example "siren", "car_honk" or "human_voice").
- `source_file`: the path to the specific audio file to be used.
- `source_time`: the time in the source file from which to start the event.
- `event_time`: the start time of the event in the synthesized soundscape.
- `event_duration`: the duration of the event in the synthesized soundscape.
- `snr`: the signal-to-noise ratio (in LUFS) compared to the background. In other words, how many dB above or below the background should this sound event be percieved.

Scaper also supports on-the-fly augmentation of sound events, that is, applying audio transformations to the sound events in order to increase the variability of the resulting soundscape. Currently, the supported transformations include pitch shifting and time stretching:

- `pitch_shift`: the number of semitones (can be fractional) by which to shift the sound up or down.
- `time_stretch`: the factor by which to stretch the sound event. Factors <1 will make the event shorter, and factors >1 will make it longer.

If you do not wish to apply any transformations, these latter two parameters (and only these) also accept `None` instead of a distribution tuple.

So, going back to the example code above, we're adding a siren sound event, the specific audio file to use will be chosen randomly from all available siren audio files in the `foreground/siren` subfolder, the event will start at time 0 of the source file, and be "pasted" into the synthesized soundscape anywhere between times 0 and 9 chosen uniformly. The event duration will be randomly chosen from a truncated normal distribution with a mean of 3 seconds, standard deviation of 1 second, and min/max values of 0.5 and 5 seconds respectively. The loudness with respect to the background will be chosen from a normal distribution with mean 10 dB and standard deviation of 3 dB. Finally, the pitch of the sound event will be shifted by a value between -2 and 2 semitones chosen uniformly within that range, and will be stretched (or condensed) by a factor chosen uniformly between 0.8 and 1.2.

Let's add a couple more events:

```
for _ in range(2):
    sc.add_event(label=('choose', []),
                 source_file=('choose', []),
                 source_time=('const', 0),
                 event_time=('uniform', 0, 9),
                 event_duration=('truncnorm', 3, 1, 0.5, 5),
                 snr=('normal', 10, 3),
                 pitch_shift=None,
                 time_stretch=None)
```

Here we use a for loop to quickly add two sound events. The specific label and source file for each event will be determined when we call `sc.generate()` (coming up), and will change with each call to this function.

1.2.5 Synthesizing soundscapes

Up to this point, we have created a `Scaper` object and added a background and three foreground sound events, whose parameters are specified using distribution tuples. Internally, this creates an *event specification*, i.e. a probabilistically-defined list of sound events. To synthesize a soundscape, we call the `generate()` function:

```
audiofile = 'soundscape.wav'
jamsfile = 'soundscape.jams'
txtfile = 'soundscape.txt'
sc.generate(audiofile, jamsfile,
            allow_repeated_label=True,
            allow_repeated_source=True,
            reverb=0.1,
            disable_sox_warnings=True,
            no_audio=False,
            txt_path=txtfile)
```

This will instantiate the event specification by sampling specific parameter values for every sound event from the distribution tuples stored in the specification. Once all parameter values have been sampled, they are used by scaper's audio processing engine to compose the soundscape and save the resulting audio to `audiofile`.

But that's not where it ends! Scaper will also generate an annotation file in JAMS format which serves as the reference annotation (also referred to as "ground truth") for the generated soundscape. Due to the flexibility of the JAMS format scaper will store in the JAMS file, in addition to the actual sound events, the probabilistic event specification (one for background events and one for foreground events). The `value` field of each observation in the JAMS file will contain a dictionary with all instantiated parameter values. This allows us to fully reconstruct the audio of a scaper soundscape from its JAMS annotation using the `scaper.generate_from_jams()` function (not discussed in this tutorial).

We can optionally provide `generate()` a path to a text file with the `txt_path` parameter. If provided, scaper will also save a simplified annotation of the soundscape in a tab-separated text file with three columns for the start time, end time, and label of every foreground sound event (note that the background is not stored in the simplified annotation!). The default separator is a tab, for compatibility with the Audacity label file format. The separator can be changed via `generate()`'s `txt_sep` parameter.

1.2.6 Synthesizing isolated events alongside the soundscape

We can also output the isolated foreground events and backgrounds alongside the soundscape. This is especially useful for generating datasets that can be used to train and evaluate source separation algorithms or models. To enable this, two additional arguments can be given to `generate()` and `generate_from_jams()`:

- `save_isolated_events`: whether or not to save the audio corresponding to the to the isolated foreground events and backgrounds within the synthesized soundscape. In our example, there are three components - the background and the two foreground events.
- `isolated_events_path`: the path where the audio corresponding to the isolated foreground events and backgrounds will be saved. If None (default) and save_isolated_events = True, the events are saved to <parentdir>/<audiofilename>_events/, where <parentdir> is the parent folder of the soundscape audio file provided in the audiofile parameter in the example below:

```
audiofile = '~/scaper_output/mysoundscape.wav'
jamsfile = '~/scaper_output/mysoundscape.jams'
txtfile = '~/scaper_output/mysoundscape.txt'
sc.generate(audiofile, jamsfile,
            allow_repeated_label=True,
            allow_repeated_source=True,
            reverb=None,
            disable_sox_warnings=True,
            no_audio=False,
            txt_path=txtfile,
            save_isolated_events=True)
```

The code above will produce the following directory structure:

```
~/scaper_output/mysoundscape.wav
~/scaper_output/mysoundscape.jams
~/scaper_output/mysoundscape.txt
~/scaper_output/mysoundscape_events/
  background0_<label>.wav
  foreground0_<label>.wav
  foreground1_<label>.wav
```

The labels for each isolated event are determined after `generate` is called. If `isolated_events_path` were specified, then it would produce:

```
~/scaper_output/mysoundscape.wav
~/scaper_output/mysoundscape.jams
~/scaper_output/mysoundscape.txt
<isolated_events_path>/
  background0_<label>.wav
  foreground0_<label>.wav
  foreground1_<label>.wav
```

The audio of the isolated events is guaranteed to sum up to the soundscape audio if and only if `reverb` is `None`! The audio of the isolated events as well as the audio of the soundscape can be accessed directly via the jams file as follows:

```
import soundfile as sf

jam = jams.load(jams_file)
ann = jam.annotations.search(namespace='scaper')[0]

soundscape_audio, sr = sf.read(ann.sandbox.scaper.soundscape_audio_path)
isolated_event_audio_paths = ann.sandbox.scaper.isolated_events_audio_path
```

(continues on next page)

(continued from previous page)

```
isolated_audio = []

for event_spec, event_audio in zip(ann, isolated_event_audio_paths):
    # event_spec contains the event description, label, etc
    # event_audio contains the path to the actual audio
    # make sure the path matches the event description
    assert event_spec.value['role'] in event_audio_path
    assert event_spec.value['label'] in event_audio_path

    isolated_audio.append(sf.read(event_audio)[0])

# the sum of the isolated audio should sum to the soundscape
assert sum(isolated_audio) == soundscape_audio
```

That's it! For a more detailed example of automatically synthesizing 1000 soundscapes using a single `Scaper` object, please see the *Example: synthesizing 1000 soundscapes in one go*.

CHAPTER 2

Examples

2.1 Example: synthesizing 1000 soundscapes in one go

```
import scaper
import numpy as np

# OUTPUT FOLDER
outfolder = 'audio/soundscapes/'

# SCAPER SETTINGS
fg_folder = 'audio/soundbank/foreground/'
bg_folder = 'audio/soundbank/background/'

n_soundscapes = 1000
ref_db = -50
duration = 10.0

min_events = 1
max_events = 9

event_time_dist = 'truncnorm'
event_time_mean = 5.0
event_time_std = 2.0
event_time_min = 0.0
event_time_max = 10.0

source_time_dist = 'const'
source_time = 0.0

event_duration_dist = 'uniform'
event_duration_min = 0.5
event_duration_max = 4.0

snr_dist = 'uniform'
```

(continues on next page)

(continued from previous page)

```
snr_min = 6
snr_max = 30

pitch_dist = 'uniform'
pitch_min = -3.0
pitch_max = 3.0

time_stretch_dist = 'uniform'
time_stretch_min = 0.8
time_stretch_max = 1.2

# generate a random seed for this Scaper object
seed = 123

# create a scaper that will be used below
sc = scaper.Scaper(duration, fg_folder, bg_folder, random_state=seed)
sc.protected_labels = []
sc.ref_db = ref_db

# Generate 1000 soundscapes using a truncated normal distribution of start times

for n in range(n_soundscapes):

    print('Generating soundscape: {:d}/{:d}'.format(n+1, n_soundscapes))

    # reset the event specifications for foreground and background at the
    # beginning of each loop to clear all previously added events
    sc.reset_bg_spec()
    sc.reset_fg_spec()

    # add background
    sc.add_background(label=('const', 'noise'),
                      source_file=('choose', []),
                      source_time=('const', 0))

    # add random number of foreground events
    n_events = np.random.randint(min_events, max_events+1)
    for _ in range(n_events):
        sc.add_event(label=('choose', []),
                     source_file=('choose', []),
                     source_time=(source_time_dist, source_time),
                     event_time=(event_time_dist, event_time_mean, event_time_std,␣
↪event_time_min, event_time_max),
                     event_duration=(event_duration_dist, event_duration_min, event_
↪duration_max),
                     snr=(snr_dist, snr_min, snr_max),
                     pitch_shift=(pitch_dist, pitch_min, pitch_max),
                     time_stretch=(time_stretch_dist, time_stretch_min, time_stretch_
↪max))

    # generate
    audiofile = os.path.join(outfolder, "soundscape_unimodal{:d}.wav".format(n))
    jamsfile = os.path.join(outfolder, "soundscape_unimodal{:d}.jams".format(n))
    txtfile = os.path.join(outfolder, "soundscape_unimodal{:d}.txt".format(n))

    sc.generate(audiofile, jamsfile,
                allow_repeated_label=True,
```

(continues on next page)

(continued from previous page)

```
                allow_repeated_source=False,
                reverb=0.1,
                disable_sox_warnings=True,
                no_audio=False,
                txt_path=txtfile)
```

CHAPTER 3

API Reference

3.1 Core functionality

CHAPTER 4

Contribute

- Issue tracker
- Source code

CHAPTER 5

Changes

5.1 Changelog

5.1.1 v1.6.5.rc0

- Added a new distirbution tuple: `("choose_weighted", list_of_options, probabilities)`, which supports weighted sampling: `list_of_options[i]` is chosen with probability `probabilities[i]`.

5.1.2 v1.6.4

- **Scaper.generate now accepts a new argument for controlling trade-off between speed and quality in pitch shifting and tim**
  - quick_pitch_time: if True, both time stretching and pitch shifting will be applied in quick mode, which is much faster but has lower quality.

5.1.3 v1.6.3

- **Scaper.generate now accepts two new optional arguments for controlling audio clipping and normalization:**
  - fix_clipping: if True and the soundscape audio is clipping, it will be peak normalized and all isolated events will be scaled accordingly.
  - peak_normalization: if True, sounscape audio will be peak normalized regardless of whether it's clipping or not and all isolated events will be scaled accordingly.
- All generate arguments are now documented in the scaper sandbox inside the JAMS annotation.
- Furthermore, we also document in the JAMS: the scale factor used for peak normalization, the change in ref_db, and the actual ref_db of the generated audio.

5.1.4 v1.6.2

- Switching from FFMpeg LUFS calculation to pyloudnorm for better performance: runtime is reduced by approximately 30%
- The loudness calculation between FFMpeg LUFS and pyloudnorm is slightly different so this version will generate marginally different audio data compared to previous versions: the change is not perceptible, but np.allclose() tests on audio from previous versions of Scaper may fail.
- This change updates the regression data for Scaper's regression tests.
- This release used soxbindings 1.2.2 and pyloudnorm 0.1.0.

5.1.5 v1.6.1

- Trimming now happens on read, rather than after read. This prevents the entire file from being loaded into memory. This is helpful for long source audio files.
- Since the audio processing pipeline has changed, this version will generate marginally different audio data compared to previous versions: the change is not perceptible, but np.allclose() tests on audio from previous versions of Scaper may fail.
- This change updates the regression data for Scaper's regression tests

5.1.6 v1.6.0

- Uses soxbindings when installing on Linux or MacOS, which results in better performance.
- Adds explicit support for Python 3.7 and 3.8. Drops support for Python 2.7 and 3.4.

5.1.7 v1.5.1

- Fixes a bug with fade in and out lengths are set to 0.
- This is the last version to support Python 2.7 and 3.4.

5.1.8 v1.5.0

- Scaper now returns the generated audio and annotations directly in memory, allowing you to skip any/all file I/O!
- Saving the audio and annotations to disk is still supported, but is now optional.
- While this update modifies the API of several functions, it should still be backwards compatible.

5.1.9 v1.4.0

- Operations on all files happen in-memory now, via new PySox features (build_array) and numpy operations for applying fades.
- Scaper is faster now due to the in-memory changes.

5.1.10 v1.3.9

- Fixed a bug where trim before generating soundscapes from a JAMS file with saving of isolated events resulted in incorrect soundscape audio.

5.1.11 v1.3.8

- Fixed a bug where _sample_trunc_norm returned an array in Scipy 1.5.1, but returns a scalar in Scipy 1.4.0.

5.1.12 v1.3.7

- Fixed a bug where time stretched events could have a negative start time if they were longer than the soundscape duration.

5.1.13 v1.3.6

- Use sox flag -s for time stretching (speech mode), gives better sounding results.

5.1.14 v1.3.5

- Fixed a bug where short backgrounds did not concatenate to fill the entire soundscape.

5.1.15 v1.3.4

- Fixed a bug where the soundscapes were off by one sample when generated. Fixes bug where generating from jams using a trimmed jams file was using the trimmed soundscape duration instead of the original duration.
- Added a field to the sandbox that keeps track of the original duration of the soundscape before any trimming is applied.

5.1.16 v1.3.3

- Fixed a bug with the format and subtype of audio files not being maintained in match_sample_length.

5.1.17 v1.3.2

- Fixed a bug with generating the file names when saving the isolated events. The idx for background and foreground events now increment separately.

5.1.18 v1.3.1

- Fixed a bug with generating docs on ReadTheDocs.

5.1.19 v1.3.0

- Source separation support! Add option to save isolated foreground events and background audio files.
- Makes pysoundfile a formal dependency.
- Seeding tests more robust.

5.1.20 v1.2.0

- Added a random_state parameter to Scaper object, which allows all runs to be perfectly reproducible given the same audio and the same random seed.
- Switched from numpydoc to napoleon for generating the documentation. Also switched Sphinx to the most recent version.
- Added functions to Scaper object that allow one to reset the foreground and background event specifications independently. This allows users to reuse the same Scaper object and generate multiple soundscapes.
- Added a function to Scaper that allows the user to set the random state after creation.

5.1.21 v1.1.0

- Added functionality which modifies a source_time distribution tuple according to the duration of the source and the duration of the event.
- This release alters behavior of Scaper compared to earlier versions.

5.1.22 v1.0.3

- Fix bug where temp files might not be closed if an error is raised

5.1.23 v1.0.2

- Store sample rate in output JAMS inside the scaper sandbox

5.1.24 v1.0.1

- Fix bug where estimated duration of time stretched event is different to actual duration leading to incorrect silence padding and sometimes incorrect soundscape duration (in audio samples).

5.1.25 v1.0.0

- Major revision
- Support jams>=0.3
- Switch from the sound_event to the scaper namespace.
- While the API remains compatible with previous versions, the change of underlying namespace breaks compatibility with jams files created using scaper for versions <1.0.0.

5.1.26 v0.2.1

- Fix bug related to creating temp files on Windows.

5.1.27 v0.2.0

- #28: Improve LUFS calculation:
  - Compute LUFS *after* initial processing (e.g. trimming, augmentation) of foreground and background events
  - Self-concatenate short events (< 500 ms) to avoid ffmpeg constant of -70.0 LUFS

5.1.28 v0.1.2

- Fix markdown display on PyPi

5.1.29 v0.1.1

- Increases minimum version of pysox to 1.3.3 to prevent crashing on Windows

5.1.30 v0.1.0

- First release.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

S