mirdata

Release 0.3.2

The mirdata development team

Feb 02, 2021

CONTENTS

`mirdata` is an open-source Python library that provides tools for working with common Music Information Retrieval (MIR) datasets, including tools for:

- downloading datasets to a common location and format
- validating that the files for a dataset are all present
- loading annotation files to a common format, consistent with `mir_eval`
- parsing track level metadata for detailed evaluations.

```
pip install mirdata
```

For more details on how to use the library see the *Tutorial*.

CHAPTER

ONE

CITING MIRDATA

If you are using the library for your work, please cite the version you used as indexed at Zenodo:

If you refer to mirdata's design principles, motivation etc., please cite the following paper[1]:

When working with datasets, please cite the version of `mirdata` that you are using (given by the `DOI` above) **AND** include the reference of the dataset, which can be found in the respective dataset loader using the `cite()` method.

[1] Rachel M. Bittner, Magdalena Fuentes, David Rubinstein, Andreas Jansson, Keunwoo Choi, and Thor Kell. "mirdata: Software for Reproducible Usage of Datasets." In Proceedings of the 20th International Society for Music Information Retrieval (ISMIR) Conference, 2019.:

CHAPTER

TWO

CONTRIBUTING TO MIRDATA

We welcome contributions to this library, especially new datasets. Please see *Contributing* for guidelines.

- Issue Tracker
- Source Code

2.1 Overview

```
pip install mirdata
```

`mirdata` is a library which aims to standardize how audio datasets are accessed in Python, removing the need for writing custom loaders in every project, and improving reproducibility. Working with datasets usually requires an often cumbersome step of downloading data and writing load functions that load related files (for example, audio and annotations) into a standard format to be used for experimenting or evaluating. `mirdata` does all of this for you:

```
import mirdata

print(mirdata.list_datasets())

tinysol = mirdata.initialize('tinysol')
tinysol.download()

# get annotations and audio for a random track
example_track = tinysol.choice_track()
instrument = example_track.instrument_full
pitch = example_track.pitch
y, sr = example_track.audio
```

`mirdata` loaders contain methods to:

- `download()`: download (or give instructions to download) a dataset
- `load_*()`: load a dataset's files (audio, metadata, annotations, etc.) into standard formats, so you don't have to write them yourself which are compatible with `mir_eval` and `jams`.
- `validate()`: validate that a dataset is complete and correct
- `cite()`: quickly print a dataset's relevant citation
- access `track` and `multitrack` objects for grouping multiple annotations for a particular track/multitrack
- and more

See the *Tutorial* for a detailed explanation of how to get started using this library.

2.1.1 mirdata design principles

Ease of use and contribution

We designed `mirdata` to be easy to use and easy to contribute to. `mirdata` simplifies the research pipeline considerably, facilitating research in a wider diversity of tasks and musical datasets. We provide detailed examples on how to interact with the library in the *Tutorial*, as well as detail explanation on how to contribute in *Contributing*. Additionally, we have a repository of Jupyter notebooks with usage examples of the different datasets.

Reproducibility

We aim for `mirdata` to aid in increasing research reproducibility by providing a common framework for MIR researchers to compare and validate their data. If mistakes are found in annotations or audio versions change, using `mirdata`, the community can fix mistakes while still being able to compare methods moving forward.

canonical versions

The `dataset loaders` in `mirdata` are written for what we call the `canonical version` of a dataset. Whenever possible, this should be the official release of the dataset as published by the dataset creator/s. When this is not possible, (e.g. for data that is no longer available), the procedure we follow is to find as many copies of the data as possible from different researchers (at least 4), and use the most common one. To make this process transparent, when there are doubts about the data consistency we open an issue and leave it to the community to discuss what to use.

Standardization

Different datasets have different annotations, metadata, etc. We try to respect the idiosyncracies of each dataset as much as we can. For this reason, `tracks` in each `Dataset` in `mirdata` have different attributes, e.g. some may have `artist` information and some may not. However there are some elements that are common in most datasets, and in these cases we standarize them to increase the usability of the library. Some examples of this are the annotations in `mirdata`, e.g. `BeatData`.

2.1.2 indexes

Indexes in *mirdata* are manifests of the files in a dataset and their corresponding md5 checksums. Specifically, an index is a json file with the mandatory top-level key `version` and at least one of the optional top-level keys `metadata`, `tracks`, `multitracks` or `records`. An index might look like:

Example Index

```
{   "version": "1.0.0",
    "metadata": {
        "metadata_file_1": [
                // the relative path for metadata_file_1
                "path_to_metadata/metadata_file_1.csv",
                // metadata_file_1 md5 checksum
                "bb8b0ca866fc2423edde01325d6e34f7"
            ],
        "metadata_file_2": [
                // the relative path for metadata_file_2
                "path_to_metadata/metadata_file_2.csv",
```

(continues on next page)

(continued from previous page)

```
                // metadata_file_2 md5 checksum
                "6cce186ce77a06541cdb9f0a671afb46"
            ]
        }
    "tracks": {
        "track1": {
            'audio': ["audio_files/track1.wav", "6c77777ce77a06541cdb9f0a671afb46"],
            'beats': ["annotations/track1.beats.csv",
↪"ab8b0ca866fc2423edde01325d6e34f7"],
            'sections': ["annotations/track1.sections.txt",
↪"05abeca866fc2423edde01325d6e34f7"],
        }
        "track2": {
            'audio': ["audio_files/track2.wav", "6c77777ce77a06542cdb9f0a672afb46"],
            'beats': ["annotations/track2.beats.csv",
↪"ab8b0ca866fc2423edde02325d6e34f7"],
            'sections': ["annotations/track2.sections.txt",
↪"05abeca866fc2423edde02325d6e34f7"],
        }
        ...
        }
}
```

The optional top-level keys (*tracks*, *multitracks* and *records*) relate to different organizations of music datasets. *tracks* are used when a dataset is organized as a collection of individual tracks, namely mono or multi-channel audio, spectrograms only, and their respective annotations. *multitracks* are used in when a dataset comprises of multitracks - different groups of tracks which are directly related to each other. Finally, *records* are used when a dataset consits of groups of tables (e.g. relational databases), as many recommendation datasets do.

See the contributing docs *1. Create an index* for more information about mirdata indexes.

2.1.3 annotations

mirdata provdes `Annotation` classes of various kinds which provide a standard interface to different annotation formats. These classes are compatible with the `mir_eval` library's expected format, as well as with the jams format. The format can be easily extended to other formats, if requested.

2.1.4 metadata

When available, we provide extensive and easy-to-access `metadata` to facilitate track metadata-specific analysis. `metadata` is available as attroibutes at the `track` level, e.g. `track.artist`.

2.2 Supported Datasets and Annotations

2.2.1 Dataset Quick Reference

This table is provided as a guide for users to select appropriate datasets. The list of annotations omits some metadata for brevity, and we document the dataset's primary annotations only. The number of tracks indicates the number of unique "tracks" in a dataset, but it may not reflect the actual size or diversity of a dataset, as tracks can vary greatly in length (from a few seconds to a few minutes), and may be homogeneous.

"Downloadable" possible values:

- : Freely downloadable
- : Available upon request
- : Youtube Links only
- : Not available

Find the API documentation for each of the below datasets in *Initializing*.

Dataset	Downloadable?	Annotation Types	Tracks	License
AcousticBrainz Genre	• audio: • annotations: • features:	• *Genre*	>4M	• BY-SA • Custom
Beatles	• audio: • annotations:	• *Beats* • *Chords* • *Sections* • *Vocal Activity*	180	
Beatport EDM key	• audio: • annotations:	• global *Key*	1486	BY-SA
cante100	• audio: • annotations:	*F0*	100	Custom
DALI	• audio: • annotations:	• *Lyrics* • Vocal *Notes*	5358	BY-SA
Giantsteps key	• audio: • annotations:	global *Key*	500	BY-SA
Giantsteps tempo	• audio: : • annotations:	• global *Genre* • global *Tempo*	664	BY-SA
Groove MIDI	• audio: • midi:	• *Beats* • *Tempo* • *Drums*	1150	BY-SA
Gtzan-Genre	• audio: : • annotations:	global *Genre*	1000	
Guitarset	• audio: • midi:	• *Beats* • *Chords* • *Key* • *Tempo* • *Notes* • *F0*	360	
Ikala	• audio: • annotations:	• Vocal *F0* • *Lyrics*	252	Custom
IRMAS	• audio: • annotations:	• *Instruments* • *Genre*	9579	BY-NC-SA
MAESTRO	• audio: • annotations:	Piano *Notes*	1282	BY-NC-SA

2.2.2 Annotation Types

The table above provides annotation types as a guide for choosing appropriate datasets, but it is difficult to generically categorize annotation types, as they depend on varying definitions and their meaning can change depending on the type of music they correspond to. Here we provide a rough guide to the types in this table, but we **strongly recommend** reading the dataset specific documentation to ensure the data is as you expect. To see how these annotation types are implemented in `mirdata` see *Annotations*.

Beats

Musical beats, typically encoded as sequence of timestamps and corresponding beat positions. This implicitly includes *downbeat* information (the beginning of a musical measure).

Chords

Musical chords, e.g. as might be played on a guitar. Typically encoded as a sequence of labeled events, where each event has a start time, end time, and a label. The label taxonomy varies per dataset, but typically encode a chord's root and its quality, e.g. A:m7 for "A minor 7".

Drums

Transcription of the drums, typically encoded as a sequence of labeled events, where the labels indicate which drum instrument (e.g. cymbal, snare drum) is played. These events often overlap with one another, as multiple drums can be played at the same time.

F0

Musical pitch contours, typically encoded as time series indidcating the musical pitch over time. The time series typically have evenly spaced timestamps, each with a correspoinding pitch value which may be encoded in a number of formats/granularities, including midi note numbers and Hertz.

Genre

A typically global "tag", indicating the genre of a recording. Note that the concept of genre is highly subjective and we refer those new to this task to this article.

Instruments

Labels indicating which instrument is present in a musical recording. This may refer to recordings of solo instruments, or to recordings with multiple instruments. The labels may be global to a recording, or they may vary over time, indicating the presence/absence of a particular instrument as a time series.

Key

Musical key. This can be defined globally for an audio file or as a sequence of events.

Lyrics

Lyrics corresponding to the singing voice of the audio. These may be raw text with no time information, or they may be time-aligned events. They may have varying levels of granularity (paragraph, line, word, phoneme, character) depending on the dataset.

Melody

The musical melody of a song. Melody has no universal definition and is typically defined per dataset. It is typically enocoded as *F0* or as *Notes*. Other types of annotations such as Vocal F0 or Vocal Notes can often be considered as melody annotations as well.

Notes

Musical note events, typically encoded as sequences of start time, end time, label. The label typically indicates a musical pitch, which may be in a number of formats/granularities, including midi note numbers, Hertz, or pitch class.

Phrases

Musical phrase events, typically encoded by a sequence of timestamps indicating the boundary times and defined by solfège symbols. This annotations are not intended to describe the complete melody but the musical phrases present in the track.

Sections

Musical sections, which may be "flat" or "hierarchical", typically encoded by a sequence of timestamps indicating musical section boundary times. Section annotations sometimes also include labels for sections, which may indicate repetitions and/or the section type (e.g. Chorus, Verse).

Technique

The playing technique used by a particular instrument, for example "Pizzicato". This label may be global for a given recording or encoded as a sequence of labeled events.

Tempo

The tempo of a song, typical in units of beats-per-minute (bpm). This is often indicated globally per track, but in practice tracks may have tempos that change, and some datasets encode tempo as time-varying quantity. Additionally, there may be multiple reasonable tempos at any given time (for example, often 2x or 0.5x a tempo value will also be "correct"). For this reason, some datasets provide two or more different tempo values.

Vocal Activity

A time series or sequence of events indicating when singing voice is present in a recording. This type of annotation is implicitly available when Vocal *F0* or Vocal *Notes* annotations are available.

Stroke Name

An open "tag" to identify an instrument stroke name or type. Used for instruments that have specific stroke labels.

Tonic

The absolute tonic of a track. It may refer to the tonic a single stroke, or the tonal center of a track.

2.3 Tutorial

2.3.1 Installation

To install `mirdata`:

```
pip install mirdata
```

2.3.2 Usage

`mirdata` is easily imported into your Python code by:

```
import mirdata
```

Initializing a dataset

Print a list of all available dataset loaders by calling:

```
import mirdata
print(mirdata.list_datasets())
```

To use a loader, (for example, 'orchset') you need to initialize it by calling:

```
import mirdata
orchset = mirdata.initialize('orchset')
```

Now `orchset` is a `Dataset` object containing common methods, described below.

Downloading a dataset

All dataset loaders in `mirdata` have a `download()` function that allows the user to download the canonical version of the dataset (when available). When initializing a dataset it is important to set up correctly the directory where the dataset is going to be stored and retrieved.

Downloading a dataset into the default folder: In this first example, `data_home` is not specified. Thus, ORCHSET will be downloaded and retrieved from `mir_datasets` folder created at user root folder:

```
import mirdata
orchset = mirdata.initialize('orchset')
orchset.download()  # Dataset is downloaded at user root folder
```

Downloading a dataset into a specified folder: Now `data_home` is specified and so ORCHSET will be downloaded and retrieved from it:

```
orchset = mirdata.initialize('orchset', data_home='Users/johnsmith/Desktop')
orchset.download()  # Dataset is downloaded at John Smith's desktop
```

Partially downloading a dataset

The `download()` functions allows to partially download a dataset. In other words, if applicable, the user can select which elements of the dataset they want to download. Each dataset has a `REMOTES` dictionary were all the available elements are listed.

`cante100` has different elements as seen in the `REMOTES` dictionary. Thus, we can specify which of these elements are downloaded, by passing to the `download()` function the list of keys in `REMOTES` that we are interested in. This list is passed to the `download()` function through the `partial_download` variable.

Example REMOTES

```
REMOTES = {
    "spectrogram": download_utils.RemoteFileMetadata(
        filename="cante100_spectrum.zip",
        url="https://zenodo.org/record/1322542/files/cante100_spectrum.zip?download=1
↪",
        checksum="0b81fe0fd7ab2c1adc1ad789edb12981",  # the md5 checksum
        destination_dir="cante100_spectrum",  # relative path for where to unzip the␣
↪data, or None
    ),
    "melody": download_utils.RemoteFileMetadata(
        filename="cante100midi_f0.zip",
        url="https://zenodo.org/record/1322542/files/cante100midi_f0.zip?download=1",
        checksum="cce543b5125eda5a984347b55fdcd5e8",  # the md5 checksum
        destination_dir="cante100midi_f0",  # relative path for where to unzip the␣
↪data, or None
    ),
    "notes": download_utils.RemoteFileMetadata(
        filename="cante100_automaticTranscription.zip",
        url="https://zenodo.org/record/1322542/files/cante100_automaticTranscription.
↪zip?download=1",
        checksum="47fea64c744f9fe678ae5642a8f0ee8e",  # the md5 checksum
        destination_dir="cante100_automaticTranscription",  # relative path for where␣
↪to unzip the data, or None
    ),
```

(continues on next page)

(continued from previous page)

```
    "metadata": download_utils.RemoteFileMetadata(
        filename="cante100Meta.xml",
        url="https://zenodo.org/record/1322542/files/cante100Meta.xml?download=1",
        checksum="6cce186ce77a06541cdb9f0a671afb46",  # the md5 checksum
    ),
    "README": download_utils.RemoteFileMetadata(
        filename="cante100_README.txt",
        url="https://zenodo.org/record/1322542/files/cante100_README.txt?download=1",
        checksum="184209b7e7d816fa603f0c7f481c0aae",  # the md5 checksum
    ),
}
```

An partial download example for `cante100` dataset could be:

```
cante100.download(partial_download=['spectrogram', 'melody', 'metadata'])
```

Validating a dataset

Using the method `validate()` we can check if the files in the local version are the same than the available canical version, and the files were downloaded correctly (none of them are corrupted).

For big datasets: In future `mirdata` versions, a random validation will be included. This improvement will reduce validation time for very big datasets.

Accessing annotations

We can choose a random track from a dataset with the `choice_track()` method.

Example Index

```
random_track = orchset.choice_track()
print(random_track)
>>> Track(
       alternating_melody=True,
       audio_path_mono="user/mir_datasets/orchset/audio/mono/Beethoven-S3-I-ex1.wav",
       audio_path_stereo="user/mir_datasets/orchset/audio/stereo/Beethoven-S3-I-ex1.
↪wav",
       composer="Beethoven",
       contains_brass=False,
       contains_strings=True,
       contains_winds=True,
       excerpt="1",
       melody_path="user/mir_datasets/orchset/GT/Beethoven-S3-I-ex1.mel",
       only_brass=False,
       only_strings=False,
       only_winds=False,
       predominant_melodic_instruments=['strings', 'winds'],
       track_id="Beethoven-S3-I-ex1",
       work="S3-I",
       audio_mono: (np.ndarray, float),
       audio_stereo: (np.ndarray, float),
       melody: F0Data,
    )
```

We can also access specific tracks by id. The available track ids can be acessed via the *.track_ids* attribute. In the next example we take the first track id, and then we retrieve the melody annotation.

```
orchset_ids = orchset.track_ids  # the list of orchset's track ids
orchset_data = orchset.load_tracks()  # Load all tracks in the dataset
example_track = orchset_data[orchset_ids[0]]  # Get the first track

# Accessing the track's melody annotation
example_melody = example_track.melody
```

Alternatively, we don't need to load the whole dataset to get a single track.

```
orchset_ids = orchset.track_ids  # the list of orchset's track ids
example_track = orchset.track(orchset_ids[0])  # load this particular track
example_melody = example_track.melody  # Get the melody from first track
```

Accessing data remotely

Annotations can also be accessed through `load_*()` methods which may be useful, for instance, when your data isn't available locally. If you specify the annotation's path, you can use the module's loading functions directly. Let's see an example.

Accessing annotations remotely example

```
# Load list of track ids of the dataset
orchset_ids = orchset.track_ids

# Load a single track, specifying the remote location
example_track = orchset.track(orchset_ids[0], data_home='user/my_custom/remote_path')
melody_path = example_track.melody_path

print(melody_path)
>>> user/my_custom/remote_path/GT/Beethoven-S3-I-ex1.mel
print(os.path.exists(melody_path))
>>> False

# Write code here to locally download your path e.g. to a temporary file.
def my_downloader(remote_path):
    # the contents of this function will depend on where your data lives, and how␣
↪permanently you want the files to remain on the machine. We point you to libraries␣
↪handling common use cases below.
    # for data you would download via scp, you could use the [scp](https://pypi.org/
↪project/scp/) library
    # for data on google drive, use [pydrive](https://pythonhosted.org/PyDrive/)
    # for data on google cloud storage use [google-cloud-storage](https://pypi.org/
↪project/google-cloud-storage/)
    return local_path_to_downloaded_data

# Get path where youe data lives
temp_path = my_downloader(melody_path)

# Accessing to track melody annotation
example_melody = orchset.load_melody(temp_path)
```

(continues on next page)

(continued from previous page)

```
print(example_melody.frequencies)
>>> array([  0.   ,   0.   ,   0.   , ..., 391.995, 391.995, 391.995])
print(example_melody.times)
>>> array([0.000e+00, 1.000e-02, 2.000e-02, ..., 1.244e+01, 1.245e+01, 1.246e+01])
```

Annotation classes

`mirdata` defines annotation-specific data classes. These data classes are meant to standarize the format for all loaders, and are compatibly with JAMS and mir_eval.

The list and descriptions of available annotation classes can be found in *Annotations*.

Note: These classes may be extended in the case that a loader requires it.

Iterating over datasets and annotations

In general, most datasets are a collection of tracks, and in most cases each track has an audio file along with annotations.

With the `load_tracks()` method, all tracks are loaded as a dictionary with the ids as keys and track objects (which include their respective audio and annotations, which are lazy-loaded on access) as values.

```
orchset = mirdata.initialize('orchset')
for key, track in orchset.load_tracks().items():
    print(key, track.audio_path)
```

Alternatively, we can loop over the `track_ids` list to directly access each track in the dataset.

```
orchset = mirdata.initialize('orchset')
for track_id in orchset.track_ids:

    print(track_id, orchset.track(track_id).audio_path)
```

Basic example: including mirdata in your pipeline

If we wanted to use `orchset` to evaluate the performance of a melody extraction algorithm (in our case, `very_bad_melody_extractor`), and then split the scores based on the metadata, we could do the following:

mirdata usage example

```
import mir_eval
import mirdata
import numpy as np
import sox

def very_bad_melody_extractor(audio_path):
    duration = sox.file_info.duration(audio_path)
    time_stamps = np.arange(0, duration, 0.01)
    melody_f0 = np.random.uniform(low=80.0, high=800.0, size=time_stamps.shape)
```

(continues on next page)

(continued from previous page)

```
    return time_stamps, melody_f0

# Evaluate on the full dataset
orchset = mirdata.initialize("orchset")
orchset_scores = {}
orchset_data = orchset.load_tracks()
for track_id, track_data in orchset_data.items():
    est_times, est_freqs = very_bad_melody_extractor(track_data.audio_path_mono)

    ref_melody_data = track_data.melody
    ref_times = ref_melody_data.times
    ref_freqs = ref_melody_data.frequencies

    score = mir_eval.melody.evaluate(ref_times, ref_freqs, est_times, est_freqs)
    orchset_scores[track_id] = score

# Split the results by composer and by instrumentation
composer_scores = {}
strings_no_strings_scores = {True: {}, False: {}}
for track_id, track_data in orchset_data.items():
    if track_data.composer not in composer_scores.keys():
        composer_scores[track_data.composer] = {}

    composer_scores[track_data.composer][track_id] = orchset_scores[track_id]
    strings_no_strings_scores[track_data.contains_strings][track_id] = \
        orchset_scores[track_id]
```

This is the result of the example above.

Example result

```
print(strings_no_strings_scores)
>>> {True: {
        'Beethoven-S3-I-ex1':OrderedDict([
               ('Voicing Recall', 1.0),
               ('Voicing False Alarm', 1.0),
               ('Raw Pitch Accuracy', 0.029798422436459245),
               ('Raw Chroma Accuracy', 0.08063102541630149),
               ('Overall Accuracy', 0.0272654370489174)
               ]),
        'Beethoven-S3-I-ex2': OrderedDict([
               ('Voicing Recall', 1.0),
               ('Voicing False Alarm', 1.0),
               ('Raw Pitch Accuracy', 0.009221311475409836),
               ('Raw Chroma Accuracy', 0.07377049180327869),
               ('Overall Accuracy', 0.008754863813229572)]),
        ...

        'Wagner-Tannhauser-Act2-ex2': OrderedDict([
               ('Voicing Recall', 1.0),
               ('Voicing False Alarm', 1.0),
               ('Raw Pitch Accuracy', 0.03685636856368564),
               ('Raw Chroma Accuracy', 0.08997289972899729),
               ('Overall Accuracy', 0.036657681940700806)])
        }}
```

You can see that `very_bad_melody_extractor` performs very badly!

Using mirdata with tensorflow

The following is a simple example of a generator that can be used to create a tensorflow Dataset.

mirdata with tf.data.Dataset example

```
import mirdata
import numpy as np
import tensorflow as tf

def orchset_generator():
    # using the default data_home
    orchset = mirdata.initialize("orchset")
    track_ids = orchset.track_ids()
    for track_id in track_ids:
        track = orchset.track(track_id)
        audio_signal, sample_rate = track.audio_mono
        yield {
            "audio": audio_signal.astype(np.float32),
            "sample_rate": sample_rate,
            "annotation": {
                "times": track.melody.times.astype(np.float32),
                "freqs": track.melody.frequencies.astype(np.float32),
            },
            "metadata": {"track_id": track.track_id}
        }

dataset = tf.data.Dataset.from_generator(
    orchset_generator,
    {
        "audio": tf.float32,
        "sample_rate": tf.float32,
        "annotation": {"times": tf.float32, "freqs": tf.float32},
        "metadata": {'track_id': tf.string}
    }
)
```

In future `mirdata` versions, generators for Tensorflow and Pytorch will be included.

2.4 Initializing

`mirdata.initialize`(*dataset_name*, *data_home=None*)

Load a mirdata dataset by name

Example

```
orchset = mirdata.initialize('orchset')  # get the orchset dataset
orchset.download()  # download orchset
```

(continues on next page)

(continued from previous page)

```
orchset.validate()  # validate orchset
track = orchset.choice_track()  # load a random track
print(track)  # see what data a track contains
orchset.track_ids()  # load all track ids
```

Parameters

- **dataset_name** (*str*) – the dataset's name see mirdata.DATASETS for a complete list of possibilities
- **data_home** (*str or None*) – path where the data lives. If None uses the default location.

Returns *Dataset* – a mirdata.core.Dataset object

`mirdata.list_datasets()`

Get a list of all mirdata dataset names

Returns *list* – list of dataset names as strings

2.5 Dataset Loaders

2.5.1 acousticbrainz_genre

Acoustic Brainz Genre dataset

Dataset Info

The AcousticBrainz Genre Dataset consists of four datasets of genre annotations and music features extracted from audio suited for evaluation of hierarchical multi-label genre classification systems.

Description about the music features can be found here: https://essentia.upf.edu/streaming_extractor_music.html

The datasets are used within the MediaEval AcousticBrainz Genre Task. The task is focused on content-based music genre recognition using genre annotations from multiple sources and large-scale music features data available in the AcousticBrainz database. The goal of our task is to explore how the same music pieces can be annotated differently by different communities following different genre taxonomies, and how this should be addressed by content-based genre r ecognition systems.

We provide four datasets containing genre and subgenre annotations extracted from four different online metadata sources:

- AllMusic and Discogs are based on editorial metadata databases maintained by music experts and enthusiasts. These sources contain explicit genre/subgenre annotations of music releases (albums) following a predefined genre namespace and taxonomy. We propagated release-level annotations to recordings (tracks) in AcousticBrainz to build the datasets.
- Lastfm and Tagtraum are based on collaborative music tagging platforms with large amounts of genre labels provided by their users for music recordings (tracks). We have automatically inferred a genre/subgenre taxonomy and annotations from these labels.

For details on format and contents, please refer to the data webpage.

Note, that the AllMusic ground-truth annotations are distributed separately at https://zenodo.org/record/2554044.

If you use the MediaEval AcousticBrainz Genre dataset or part of it, please cite our ISMIR 2019 overview paper:

```
Bogdanov, D., Porter A., Schreiber H., Urbano J., & Oramas S. (2019).
The AcousticBrainz Genre Dataset: Multi-Source, Multi-Level, Multi-Label, and Large-
↪Scale.
20th International Society for Music Information Retrieval Conference (ISMIR 2019).
```

This work is partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 688382 AudioCommons.

---

class mirdata.datasets.acousticbrainz_genre.**Dataset**(*data_home=None*)

The acousticbrainz genre dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

filter_index(*search_key*)

Load from AcousticBrainz genre dataset the indexes that match with search_key.

Parameters **search_key** (*str*) – regex to match with folds, mbid or genres

Returns *dict* – {*track_id*: track data}

license()

Print the license

load_all_train()
Load from AcousticBrainz genre dataset the tracks that are used for training across the four different datasets.

Returns *dict* – {*track_id*: track data}

load_all_validation()
Load from AcousticBrainz genre dataset the tracks that are used for validating across the four different datasets.

Returns *dict* – {*track_id*: track data}

load_allmusic_train()
Load from AcousticBrainz genre dataset the tracks that are used for validation in allmusic dataset.

Returns *dict* – {*track_id*: track data}

load_allmusic_validation()
Load from AcousticBrainz genre dataset the tracks that are used for validation in allmusic dataset.

Returns *dict* – {*track_id*: track data}

load_discogs_train()
Load from AcousticBrainz genre dataset the tracks that are used for training in discogs dataset.

Returns *dict* – {*track_id*: track data}

load_discogs_validation()
Load from AcousticBrainz genre dataset the tracks that are used for validation in tagtraum dataset.

Returns *dict* – {*track_id*: track data}

load_extractor(**args*, ***kwargs*)
Load a AcousticBrainz Dataset json file with all the features and metadata.

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to a json file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_lastfm_train()
Load from AcousticBrainz genre dataset the tracks that are used for training in lastfm dataset.

Returns *dict* – {*track_id*: track data}

load_lastfm_validation()
Load from AcousticBrainz genre dataset the tracks that are used for validation in lastfm dataset.

Returns *dict* – {*track_id*: track data}

load_tagtraum_train()
Load from AcousticBrainz genre dataset the tracks that are used for training in tagtraum dataset.

Returns *dict* – {*track_id*: track data}

load_tagtraum_validation()
Load from AcousticBrainz genre dataset the tracks that are used for validating in tagtraum dataset.

Returns *dict* – {*track_id*: track data}

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids

Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.acousticbrainz_genre.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

AcousticBrainz Genre Dataset track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **track_id** (*str*) – track id
- **genre** (*list*) – human-labeled genre and subgenres list
- **mbid** (*str*) – musicbrainz id
- **mbid_group** (*str*) – musicbrainz id group
- ***artist*** (*list*) – the track's artist/s
- ***title*** (*list*) – the track's title
- ***date*** (*list*) – the track's release date/s
- **filename** (*str*) – the track's filename
- ***album*** (*list*) – the track's album/s
- **track_number** (*list*) – the track number/s
- ***tonal*** (*dict*) – dictionary of acousticbrainz tonal features
- ***low_level*** (*dict*) – dictionary of acousticbrainz low-level features
- ***rhythm*** (*dict*) – dictionary of acousticbrainz rhythm features

Other Parameters **acousticbrainz_metadata** (*dict*) – dictionary of metadata provided by AcousticBrainz

property album

metadata album annotation

Returns *list* – album

property artist

metadata artist annotation

Returns *list* – artist

property date
metadata date annotation

Returns *list* – date

property file_name
metadata file_name annotation

Returns *str* – file name

property low_level
low_level track descriptors.

Returns

dict –

- 'average_loudness': dynamic range descriptor. It rescales average loudness, computed on 2sec windows with 1 sec overlap, into the [0,1] interval. The value of 0 corresponds to signals with large dynamic range, 1 corresponds to signal with little dynamic range. Algorithms: Loudness
- 'dynamic_complexity': dynamic complexity computed on 2sec windows with 1sec overlap. Algorithms: DynamicComplexity
- 'silence_rate_20dB', 'silence_rate_30dB', 'silence_rate_60dB': rate of silent frames in a signal for thresholds of 20, 30, and 60 dBs. Algorithms: SilenceRate
- 'spectral_rms': spectral RMS. Algorithms: RMS
- 'spectral_flux': spectral flux of a signal computed using L2-norm. Algorithms: Flux
- 'spectral_centroid', 'spectral_kurtosis', 'spectral_spread', 'spectral_skewness': centroid and central moments statistics describing the spectral shape. Algorithms: Centroid, CentralMoments
- 'spectral_rolloff': the roll-off frequency of a spectrum. Algorithms: RollOff
- 'spectral_decrease': spectral decrease. Algorithms: Decrease
- 'hfc': high frequency content descriptor as proposed by Masri. Algorithms: HFC
- 'zerocrossingrate' zero-crossing rate. Algorithms: ZeroCrossingRate
- 'spectral_energy': spectral energy. Algorithms: Energy
- 'spectral_energyband_low', 'spectral_energyband_middle_low', 'spectral_energyband_middle_high',
- 'spectral_energyband_high': spectral energy in frequency bands [20Hz, 150Hz], [150Hz, 800Hz], [800Hz, 4kHz], and [4kHz, 20kHz]. Algorithms EnergyBand
- 'barkbands': spectral energy in 27 Bark bands. Algorithms: BarkBands
- 'melbands': spectral energy in 40 mel bands. Algorithms: MFCC
- 'erbbands': spectral energy in 40 ERB bands. Algorithms: ERBBands
- 'mfcc': the first 13 mel frequency cepstrum coefficients. See algorithm: MFCC
- 'gfcc': the first 13 gammatone feature cepstrum coefficients. Algorithms: GFCC
- 'barkbands_crest', 'barkbands_flatness_db': crest and flatness computed over energies in Bark bands. Algorithms: Crest, FlatnessDB
- 'barkbands_kurtosis', 'barkbands_skewness', 'barkbands_spread': central moments statistics over energies in Bark bands. Algorithms: CentralMoments

- 'melbands_crest', 'melbands_flatness_db': crest and flatness computed over energies in mel bands. Algorithms: Crest, FlatnessDB
- 'melbands_kurtosis', 'melbands_skewness', 'melbands_spread': central moments statistics over energies in mel bands. Algorithms: CentralMoments
- 'erbbands_crest', 'erbbands_flatness_db': crest and flatness computed over energies in ERB bands. Algorithms: Crest, FlatnessDB
- 'erbbands_kurtosis', 'erbbands_skewness', 'erbbands_spread': central moments statistics over energies in ERB bands. Algorithms: CentralMoments
- 'dissonance': sensory dissonance of a spectrum. Algorithms: Dissonance
- 'spectral_entropy': Shannon entropy of a spectrum. Algorithms: Entropy
- 'pitch_salience': pitch salience of a spectrum. Algorithms: PitchSalience
- 'spectral_complexity': spectral complexity. Algorithms: SpectralComplexity
- 'spectral_contrast_coeffs', 'spectral_contrast_valleys': spectral contrast features. Algorithms: SpectralContrast

property rhythm

rhythm essentia extractor descriptors

Returns

dict –

- 'beats_position': time positions [sec] of detected beats using beat tracking algorithm by Degara et al., 2012. Algorithms: RhythmExtractor2013, BeatTrackerDegara
- 'beats_count': number of detected beats
- 'bpm': BPM value according to detected beats
- 'bpm_histogram_first_peak_bpm', 'bpm_histogram_first_peak_spread', 'bpm_histogram_first_peak_weight',
- 'bpm_histogram_second_peak_bpm', 'bpm_histogram_second_peak_spread', 'bpm_histogram_second_peak_weight': descriptors characterizing highest and second highest peak of the BPM histogram. Algorithms: BpmHistogramDescriptors
- 'beats_loudness', 'beats_loudness_band_ratio': spectral energy computed on beats segments of audio across the whole spectrum, and ratios of energy in 6 frequency bands. Algorithms: BeatsLoudness, SingleBeatLoudness
- 'onset_rate': number of detected onsets per second. Algorithms: OnsetRate
- 'danceability': danceability estimate. Algorithms: Danceability

property title

metadata title annotation

Returns *list* – title

to_jams()

the track's data in jams format

Returns *jams.JAMS* – return track data in jam format

property tonal

tonal features

Returns

dict –

- 'tuning_frequency': estimated tuning frequency [Hz]. Algorithms: TuningFrequency
- 'tuning_nontempered_energy_ratio' and 'tuning_equal_tempered_deviation'
- 'hpcp', 'thpcp': 32-dimensional harmonic pitch class profile (HPCP) and its transposed version. Algorithms: HPCP
- 'hpcp_entropy': Shannon entropy of a HPCP vector. Algorithms: Entropy
- 'key_key', 'key_scale': Global key feature. Algorithms: Key
- 'chords_key', 'chords_scale': Global key extracted from chords detection.
- 'chords_strength', 'chords_histogram': : strength of estimated chords and normalized histogram of their progression; Algorithms: ChordsDetection, ChordsDescriptors
- 'chords_changes_rate', 'chords_number_rate': chords change rate in the progression; ratio of different chords from the total number of chords in the progression; Algorithms: ChordsDetection, ChordsDescriptors

property tracknumber

metadata tracknumber annotation

Returns *list* – tracknumber

`mirdata.datasets.acousticbrainz_genre.`**`load_extractor`**(*fhandle*)

Load a AcousticBrainz Dataset json file with all the features and metadata.

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to a json file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.2 beatles

Beatles Dataset Loader

Dataset Info

The Beatles Dataset includes beat and metric position, chord, key, and segmentation annotations for 179 Beatles songs. Details can be found in http://matthiasmauch.net/_pdf/mauch_omp_2009.pdf and http://isophonics.net/content/reference-annotations-beatles.

---

class `mirdata.datasets.beatles.`**`Dataset`**(*data_home=None*)

The beatles dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset

- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()
Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()
Print the reference

property default_path
Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a Beatles audio file.

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_beats(**args*, ***kwargs*)
Load Beatles format beat data from a file

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to a beat annotation file

Returns *BeatData* – loaded beat data

load_chords(**args*, ***kwargs*)
Load Beatles format chord data from a file

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to a chord annotation file

Returns *ChordData* – loaded chord data

load_sections(**args*, ***kwargs*)
Load Beatles format section data from a file

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to a section annotation file

Returns *SectionData* – loaded section data

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.beatles.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
Beatles track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – path where the data lives

Variables

- **audio_path** (*str*) – track audio path
- **beats_path** (*str*) – beat annotation path
- **chords_path** (*str*) – chord annotation path
- **keys_path** (*str*) – key annotation path
- **sections_path** (*str*) – sections annotation path
- ***title*** (*str*) – title of the track
- **track_id** (*str*) – track id

Other Parameters

- **beats** (*BeatData*) – human-labeled beat annotations
- **chords** (*ChordData*) – human-labeled chord annotations
- **key** (*KeyData*) – local key annotations
- **sections** (*SectionData*) – section annotations

property audio
The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()
the track's data in jams format

Returns *jams.JAMS* – return track data in jam format

`mirdata.datasets.beatles.load_audio` (*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]
Load a Beatles audio file.

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.beatles.load_beats` (*fhandle: TextIO*) → *mirdata.annotations.BeatData*
Load Beatles format beat data from a file

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to a beat annotation file

Returns *BeatData* – loaded beat data

`mirdata.datasets.beatles.load_chords` (*fhandle: TextIO*) → *mirdata.annotations.ChordData*
Load Beatles format chord data from a file

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to a chord annotation file

Returns *ChordData* – loaded chord data

`mirdata.datasets.beatles.load_key` (*fhandle: TextIO*) → *mirdata.annotations.KeyData*
Load Beatles format key data from a file

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to a key annotation file

Returns *KeyData* – loaded key data

`mirdata.datasets.beatles.load_sections` (*fhandle: TextIO*) → *mirdata.annotations.SectionData*
Load Beatles format section data from a file

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to a section annotation file

Returns *SectionData* – loaded section data

2.5.3 beatport_key

beatport_key Dataset Loader

Dataset Info

The Beatport EDM Key Dataset includes 1486 two-minute sound excerpts from various EDM subgenres, annotated with single-key labels, comments and confidence levels generously provided by Eduard Mas Marín, and thoroughly revised and expanded by Ángel Faraldo.

The original audio samples belong to online audio snippets from Beatport, an online music store for DJ's and Electronic Dance Music Producers (<http:www.beatport.com>). If this dataset were used in further research, we would appreciate the citation of the current DOI (10.5281/zenodo.1101082) and the following doctoral dissertation, where a detailed description of the properties of this dataset can be found:

```
Ángel Faraldo (2017). Tonality Estimation in Electronic Dance Music: A Computational
↪and Musically Informed
Examination. PhD Thesis. Universitat Pompeu Fabra, Barcelona.
```

This dataset is mainly intended to assess the performance of computational key estimation algorithms in electronic dance music subgenres.

Data License: Creative Commons Attribution Share Alike 4.0 International

class mirdata.datasets.beatport_key.**Dataset**(*data_home=None*)

The beatport_key dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download the dataset

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()

Print the license

load_artist(**args*, ***kwargs*)

Load beatport_key tempo data from a file

Parameters **metadata_path** (*str*) – path to metadata annotation file

Returns *list* – list of artists involved in the track.

load_audio(**args*, ***kwargs*)

Load a beatport_key audio file.

Parameters **audio_path** (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_genre(**args*, ***kwargs*)

Load beatport_key genre data from a file

Parameters **metadata_path** (*str*) – path to metadata annotation file

Returns *dict* – with the list with genres ['genres'] and list with sub-genres ['sub_genres']

load_key(**args*, ***kwargs*)

Load beatport_key format key data from a file

Parameters **keys_path** (*str*) – path to key annotation file

Returns *list* – list of annotated keys

load_tempo(**args*, ***kwargs*)

Load beatport_key tempo data from a file

Parameters **metadata_path** (*str*) – path to metadata annotation file

Returns *str* – tempo in beats per minute

load_tracks()

Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids

Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.beatport_key.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

beatport_key track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored.

Variables

- **audio_path** (*str*) – track audio path
- **keys_path** (*str*) – key annotation path
- **metadata_path** (*str*) – sections annotation path
- ***title*** (*str*) – title of the track

- **track_id** (*str*) – track id

Other Parameters

- **key** (*list*) – list of annotated musical keys
- **artists** (*list*) – artists involved in the track
- **genre** (*dict*) – genres and subgenres
- **tempo** (*int*) – tempo in beats per minute

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.beatport_key.**load_artist**(*metadata_path*)

Load beatport_key tempo data from a file

Parameters **metadata_path** (*str*) – path to metadata annotation file

Returns *list* – list of artists involved in the track.

mirdata.datasets.beatport_key.**load_audio**(*audio_path*)

Load a beatport_key audio file.

Parameters **audio_path** (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.beatport_key.**load_genre**(*metadata_path*)

Load beatport_key genre data from a file

Parameters **metadata_path** (*str*) – path to metadata annotation file

Returns *dict* – with the list with genres ['genres'] and list with sub-genres ['sub_genres']

mirdata.datasets.beatport_key.**load_key**(*keys_path*)

Load beatport_key format key data from a file

Parameters **keys_path** (*str*) – path to key annotation file

Returns *list* – list of annotated keys

mirdata.datasets.beatport_key.**load_tempo**(*metadata_path*)

Load beatport_key tempo data from a file

Parameters **metadata_path** (*str*) – path to metadata annotation file

Returns *str* – tempo in beats per minute

2.5.4 cante100

cante100 Loader

Dataset Info

The cante100 dataset contains 100 tracks taken from the COFLA corpus. We defined 10 style families of which 10 tracks each are included. Apart from the style family, we manually annotated the sections of the track in which the vocals are present. In addition, we provide a number of low-level descriptors and the fundamental frequency corresponding to the predominant melody for each track. The meta-information includes editoral meta-data and the musicBrainz ID.

Total tracks: 100

cante100 audio is only available upon request. To download the audio request access in this link: https://zenodo.org/record/1324183. Then unzip the audio into the cante100 general dataset folder for the rest of annotations and files.

Audio specifications:

- Sampling frequency: 44.1 kHz
- Bit-depth: 16 bit
- Audio format: .mp3

cante100 dataset has spectrogram available, in csv format. spectrogram is available to download without request needed, so at first instance, cante100 loader uses the spectrogram of the tracks.

The available annotations are:

- F0 (predominant melody)
- Automatic transcription of notes (of singing voice)

CANTE100 LICENSE (COPIED FROM ZENODO PAGE)

```
The provided datasets are offered free of charge for internal non-commercial use.
We do not grant any rights for redistribution or modification. All data collections␣
↪were gathered
by the COFLA team.
© COFLA 2015. All rights reserved.
```

For more details, please visit: http://www.cofla-project.com/?page_id=134

---

class mirdata.datasets.cante100.**Dataset**(*data_home=None*)

The cante100 dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()
Print the reference

property default_path
Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a cante100 audio file.

Parameters **fhandle** (*str*) – path to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_melody(**args*, ***kwargs*)
Load cante100 f0 annotations

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to melody annotation file

Returns *F0Data* – predominant melody

load_notes(**args*, ***kwargs*)
Load note data from the annotation files

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to a notes annotation file

Returns *NoteData* – note annotations

load_spectrogram(**args*, ***kwargs*)
Load a cante100 dataset spectrogram file.

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to an audio file

Returns *np.ndarray* – spectrogram

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.cante100.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
cante100 track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets/cante100*

Variables

- **track_id** (*str*) – track id
- **identifier** (*str*) – musicbrainz id of the track
- ***artist*** (*str*) – performing artists
- ***title*** (*str*) – title of the track song
- **release** (*str*) – release where the track can be found
- **duration** (*str*) – duration in seconds of the track

Other Parameters

- **melody** (*F0Data*) – annotated melody
- **notes** (*NoteData*) – annotated notes

property audio
The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

property spectrogram
spectrogram of The track's audio

Returns *np.ndarray* – spectrogram

to_jams()
Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.cante100.**load_audio**(*fhandle: str*) → Tuple[numpy.ndarray, float]
Load a cante100 audio file.

Parameters **fhandle** (*str*) – path to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.cante100.load_melody`(*fhandle: TextIO*) → Optional[*mirdata.annotations.F0Data*]

Load cante100 f0 annotations

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to melody annotation file

Returns *F0Data* – predominant melody

`mirdata.datasets.cante100.load_notes`(*fhandle: TextIO*) → *mirdata.annotations.NoteData*

Load note data from the annotation files

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to a notes annotation file

Returns *NoteData* – note annotations

`mirdata.datasets.cante100.load_spectrogram`(*fhandle: TextIO*) → numpy.ndarray

Load a cante100 dataset spectrogram file.

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to an audio file

Returns *np.ndarray* – spectrogram

2.5.5 dali

DALI Dataset Loader

Dataset Info

DALI contains 5358 audio files with their time-aligned vocal melody. It also contains time-aligned lyrics at four levels of granularity: notes, words, lines, and paragraphs.

For each song, DALI also provides additional metadata: genre, language, musician, album covers, or links to video clips.

For more details, please visit: https://github.com/gabolsgabs/DALI

---

class `mirdata.datasets.dali.Dataset`(*data_home=None*)

The dali dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()
Print the reference

property default_path
Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_annotations_class(**args*, ***kwargs*)
Load full annotations into the DALI class object

Parameters **annotations_path** (*str*) – path to a DALI annotation file

Returns *DALI.annotations* – DALI annotations object

load_annotations_granularity(**args*, ***kwargs*)
Load annotations at the specified level of granularity

Parameters

- **annotations_path** (*str*) – path to a DALI annotation file
- **granularity** (*str*) – one of 'notes', 'words', 'lines', 'paragraphs'

Returns NoteData for granularity='notes' or LyricData otherwise

load_audio(**args*, ***kwargs*)
Load a DALI audio file.

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate (*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.dali.**Track** (*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
DALI melody Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- ***album*** (*str*) – the track's album
- **annotation_path** (*str*) – path to the track's annotation file
- ***artist*** (*str*) – the track's artist
- **audio_path** (*str*) – path to the track's audio file
- **audio_url** (*str*) – youtube ID
- **dataset_version** (*int*) – dataset annotation version
- **ground_truth** (*bool*) – True if the annotation is verified
- **language** (*str*) – sung language
- **release_date** (*str*) – year the track was released
- **scores_manual** (*int*) – manual score annotations
- **scores_ncc** (*float*) – ncc score annotations
- ***title*** (*str*) – the track's title
- **track_id** (*str*) – the unique track id
- **url_working** (*bool*) – True if the youtube url was valid

Other Parameters

- **notes** (*NoteData*) – vocal notes
- **words** (*LyricData*) – word-level lyrics
- **lines** (*LyricData*) – line-level lyrics
- **paragraphs** (*LyricData*) – paragraph-level lyrics
- **annotation-object** (*DALI.Annotations*) – DALI annotation object

property audio
The track's audio

Returns

- np.ndarray - audio signal

- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.dali.**load_annotations_class**(*annotations_path*)

Load full annotations into the DALI class object

Parameters **annotations_path** (*str*) – path to a DALI annotation file

Returns *DALI.annotations* – DALI annotations object

mirdata.datasets.dali.**load_annotations_granularity**(*annotations_path*, *granularity*)

Load annotations at the specified level of granularity

Parameters

- **annotations_path** (*str*) – path to a DALI annotation file
- **granularity** (*str*) – one of 'notes', 'words', 'lines', 'paragraphs'

Returns NoteData for granularity='notes' or LyricData otherwise

mirdata.datasets.dali.**load_audio**(*fhandle: BinaryIO*) → Optional[Tuple[numpy.ndarray, float]]

Load a DALI audio file.

Parameters **fhandle** (*str or file-like*) – path or file-like object pointing to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.6 giantsteps_key

giantsteps_key Dataset Loader

Dataset Info

The GiantSteps+ EDM Key Dataset includes 600 two-minute sound excerpts from various EDM subgenres, annotated with single-key labels, comments and confidence levels by Daniel G. Camhi, and thoroughly revised and expanded by Ángel Faraldo at MTG UPF. Additionally, 500 tracks have been thoroughly analysed, containing pitch-class set descriptions, key changes, and additional modal changes. This dataset is a revision of the original GiantSteps Key Dataset, available in Github (<https://github.com/GiantSteps/giantsteps-key-dataset>) and initially described in:

```
Knees, P., Faraldo, Á., Herrera, P., Vogl, R., Böck, S., Hörschläger, F., Le Goff, M.
↪(2015).
Two Datasets for Tempo Estimation and Key Detection in Electronic Dance Music
↪Annotated from User Corrections.
In Proceedings of the 16th International Society for Music Information Retrieval
↪Conference, 364-370. Málaga, Spain.
```

The original audio samples belong to online audio snippets from Beatport, an online music store for DJ's and Electronic Dance Music Producers (<http:www.beatport.com>). If this dataset were used in further research, we would appreciate the citation of the current DOI (10.5281/zenodo.1101082) and the following doctoral dissertation, where a detailed description of the properties of this dataset can be found:

```
Ángel Faraldo (2017). Tonality Estimation in Electronic Dance Music: A Computational␣
↪and Musically Informed Examination.
PhD Thesis. Universitat Pompeu Fabra, Barcelona.
```

This dataset is mainly intended to assess the performance of computational key estimation algorithms in electronic dance music subgenres.

All the data of this dataset is licensed with Creative Commons Attribution Share Alike 4.0 International.

---

class `mirdata.datasets.giantsteps_key.`**`Dataset`**(*data_home=None*)

The giantsteps_key dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()

Print the license

load_artist(**args*, ***kwargs*)

Load giantsteps_key tempo data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path pointing to metadata annotation file

Returns *list* – list of artists involved in the track.

load_audio(**args*, ***kwargs*)
Load a giantsteps_key audio file.

Parameters **fhandle** (*str or file-like*) – path pointing to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_genre(**args*, ***kwargs*)
Load giantsteps_key genre data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path pointing to metadata annotation file

Returns *dict – {'genres': [...], 'subgenres': [...]}*

load_key(**args*, ***kwargs*)
Load giantsteps_key format key data from a file

Parameters **fhandle** (*str or file-like*) – File like object or string pointing to key annotation file

Returns *str* – loaded key data

load_tempo(**args*, ***kwargs*)
Load giantsteps_key tempo data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or string pointing to metadata annotation file

Returns *str* – loaded tempo data

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.giantsteps_key.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
giantsteps_key track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **audio_path** (*str*) – track audio path

- **keys_path** (*str*) – key annotation path
- **metadata_path** (*str*) – sections annotation path
- **title** (*str*) – title of the track
- **track_id** (*str*) – track id

Other Parameters

- **key** (*str*) – musical key annotation
- **artists** (*list*) – list of artists involved
- **genres** (*dict*) – genres and subgenres
- **tempo** (*int*) – crowdsourced tempo annotations in beats per minute

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.giantsteps_key.**load_artist**(*fhandle: TextIO*) → List[str]

Load giantsteps_key tempo data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path pointing to metadata annotation file

Returns *list* – list of artists involved in the track.

mirdata.datasets.giantsteps_key.**load_audio**(*fhandle: str*) → Tuple[numpy.ndarray, float]

Load a giantsteps_key audio file.

Parameters **fhandle** (*str or file-like*) – path pointing to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.giantsteps_key.**load_genre**(*fhandle: TextIO*) → Dict[str, List[str]]

Load giantsteps_key genre data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path pointing to metadata annotation file

Returns *dict – {'genres': [...], 'subgenres': [...]}*

mirdata.datasets.giantsteps_key.**load_key**(*fhandle: TextIO*) → str

Load giantsteps_key format key data from a file

Parameters **fhandle** (*str or file-like*) – File like object or string pointing to key annotation file

Returns *str* – loaded key data

mirdata.datasets.giantsteps_key.**load_tempo**(*fhandle: TextIO*) → str

Load giantsteps_key tempo data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or string pointing to metadata annotation file

Returns *str* – loaded tempo data

2.5.7 giantsteps_tempo

giantsteps_tempo Dataset Loader

Dataset Info

GiantSteps tempo + genre is a collection of annotations for 664 2min(1) audio previews from www.beatport.com, created by Richard Vogl <richard.vogl@tuwien.ac.at> and Peter Knees <peter.knees@tuwien.ac.at>

references:

The audio files (664 files, size ~1gb) can be downloaded from http://www.beatport.com/ using the bash script:

https://github.com/GiantSteps/giantsteps-tempo-dataset/blob/master/audio_dl.sh

To download the files manually use links of the following form: http://geo-samples.beatport.com/lofi/<name of mp3 file> e.g.: http://geo-samples.beatport.com/lofi/5377710.LOFI.mp3

To convert the audio files to .wav use the script found at https://github.com/GiantSteps/giantsteps-tempo-dataset/blob/master/convert_audio.sh and run:

```
./convert_audio.sh
```

To retrieve the genre information, the JSON contained within the website was parsed. The tempo annotation was extracted from forum entries of people correcting the bpm values (i.e. manual annotation of tempo). For more information please refer to the publication [giantsteps_tempo_cit_1].

[giantsteps_tempo_cit_2] found some files without tempo. There are:

```
3041381.LOFI.mp3
3041383.LOFI.mp3
1327052.LOFI.mp3
```

Their v2 tempo is denoted as 0.0 in tempo and mirex and has no annotation in the JAMS format.

Most of the audio files are 120 seconds long. Exceptions are:

```
name              length (sec)
906760.LOFI.mp3   62
1327052.LOFI.mp3  70
4416506.LOFI.mp3  80
1855660.LOFI.mp3  119
3419452.LOFI.mp3  119
3577631.LOFI.mp3  119
```

---

class mirdata.datasets.giantsteps_tempo.**Dataset**(*data_home=None*)

The giantsteps_tempo dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded

- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()
Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()
Print the reference

property default_path
Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a giantsteps_tempo audio file.

Parameters **fhandle** (*str or file-like*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_genre(**args*, ***kwargs*)
Load genre data from a file

Parameters **path** (*str*) – path to metadata annotation file

Returns *str* – loaded genre data

load_tempo(**args*, ***kwargs*)
Load giantsteps_tempo tempo data from a file ordered by confidence

Parameters **fhandle** (*str or file-like*) – File-like object or path to tempo annotation file

Returns *annotations.TempoData* – Tempo data

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids

Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.giantsteps_tempo.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

giantsteps_tempo track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **audio_path** (*str*) – track audio path
- ***title*** (*str*) – title of the track
- **track_id** (*str*) – track id
- **annotation_v1_path** (*str*) – track annotation v1 path
- **annotation_v2_path** (*str*) – track annotation v2 path

Other Parameters

- **genre** (*dict*) – Human-labeled metadata annotation
- **tempo** (*list*) – List of annotations.TempoData, ordered by confidence
- **tempo_v2** (*list*) – List of annotations.TempoData for version 2, ordered by confidence

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

to_jams_v2()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.giantsteps_tempo.**load_audio**(*fhandle: str*) → Tuple[numpy.ndarray, float]

Load a giantsteps_tempo audio file.

Parameters **fhandle** (*str or file-like*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.giantsteps_tempo.load_genre`(*fhandle: TextIO*) → str

Load genre data from a file

Parameters **path** (*str*) – path to metadata annotation file

Returns *str* – loaded genre data

`mirdata.datasets.giantsteps_tempo.load_tempo`(*fhandle: TextIO*) → *mirdata.annotations.TempoData*

Load giantsteps_tempo tempo data from a file ordered by confidence

Parameters **fhandle** (*str or file-like*) – File-like object or path to tempo annotation file

Returns *annotations.TempoData* – Tempo data

2.5.8 groove_midi

Groove MIDI Loader

Dataset Info

The Groove MIDI Dataset (GMD) is composed of 13.6 hours of aligned MIDI and synthesized audio of human-performed, tempo-aligned expressive drumming. The dataset contains 1,150 MIDI files and over 22,000 measures of drumming.

To enable a wide range of experiments and encourage comparisons between methods on the same data, Gillick et al. created a new dataset of drum performances recorded in MIDI format. They hired professional drummers and asked them to perform in multiple styles to a click track on a Roland TD-11 electronic drum kit. They also recorded the aligned, high-quality synthesized audio from the TD-11 and include it in the release.

The Groove MIDI Dataset (GMD), has several attributes that distinguish it from existing ones:

- The dataset contains about 13.6 hours, 1,150 MIDI files, and over 22,000 measures of drumming.
- Each performance was played along with a metronome set at a specific tempo by the drummer.
- The data includes performances by a total of 10 drummers, with more than 80% of duration coming from hired professionals. The professionals were able to improvise in a wide range of styles, resulting in a diverse dataset.
- The drummers were instructed to play a mix of long sequences (several minutes of continuous playing) and short beats and fills.
- Each performance is annotated with a genre (provided by the drummer), tempo, and anonymized drummer ID.
- Most of the performances are in 4/4 time, with a few examples from other time signatures.
- Four drummers were asked to record the same set of 10 beats in their own style. These are included in the test set split, labeled eval-session/groove1-10.
- In addition to the MIDI recordings that are the primary source of data for the experiments in this work, the authors captured the synthesized audio outputs of the drum set and aligned them to within 2ms of the corresponding MIDI files.

A train/validation/test split configuration is provided for easier comparison of model accuracy on various tasks.

The dataset is made available by Google LLC under a Creative Commons Attribution 4.0 International (CC BY 4.0) License.

For more details, please visit: http://magenta.tensorflow.org/datasets/groove

class mirdata.datasets.groove_midi.**Dataset**(*data_home=None*)

The groove_midi dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()

Print the license

load_audio(**args*, ***kwargs*)

Load a Groove MIDI audio file.

Parameters **path** – path to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_beats(**args*, ***kwargs*)

Load beat data from the midi file.

Parameters

- **midi_path** (*str*) – path to midi file
- **midi** (*pretty_midi.PrettyMIDI*) – pre-loaded midi object or None if None, the midi object is loaded using midi_path

Returns *annotations.BeatData* – machine generated beat data

load_drum_events(**args*, ***kwargs*)

Load drum events from the midi file.

Parameters

- **midi_path** (*str*) – path to midi file
- **midi** (*pretty_midi.PrettyMIDI*) – pre-loaded midi object or None if None, the midi object is loaded using midi_path

Returns *annotations.EventData* – drum event data

load_midi(**args*, ***kwargs*)

Load a Groove MIDI midi file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to midi file

Returns *midi_data (pretty_midi.PrettyMIDI)* – pretty_midi object

load_tracks()

Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids

Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.groove_midi.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Groove MIDI Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **drummer** (*str*) – Drummer id of the track (ex. 'drummer1')
- **session** (*str*) – Type of session (ex. 'session1', 'eval_session')
- **track_id** (*str*) – track id of the track (ex. 'drummer1/eval_session/1')
- **style** (*str*) – Style (genre, groove type) of the track (ex. 'funk/groove1')
- **tempo** (*int*) – track tempo in beats per minute (ex. 138)

- **beat_type** (*str*) – Whether the track is a beat or a fill (ex. 'beat')
- **time_signature** (*str*) – Time signature of the track (ex. '4-4', '6-8')
- **midi_path** (*str*) – Path to the midi file
- **audio_path** (*str*) – Path to the audio file
- **duration** (*float*) – Duration of the midi file in seconds
- **split** (*str*) – Whether the track is for a train/valid/test set. One of 'train', 'valid' or 'test'.

Other Parameters

- **beats** (*BeatData*) – Machine-generated beat annotations
- **drum_events** (*EventData*) – Annotated drum kit events
- **midi** (*pretty_midi.PrettyMIDI*) – object containing MIDI information

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.groove_midi.**load_audio**(*path: str*) → Tuple[Optional[numpy.ndarray], Optional[float]]

Load a Groove MIDI audio file.

Parameters **path** – path to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.groove_midi.**load_beats**(*midi_path*, *midi=None*)

Load beat data from the midi file.

Parameters

- **midi_path** (*str*) – path to midi file
- **midi** (*pretty_midi.PrettyMIDI*) – pre-loaded midi object or None if None, the midi object is loaded using midi_path

Returns *annotations.BeatData* – machine generated beat data

mirdata.datasets.groove_midi.**load_drum_events**(*midi_path*, *midi=None*)

Load drum events from the midi file.

Parameters

- **midi_path** (*str*) – path to midi file
- **midi** (*pretty_midi.PrettyMIDI*) – pre-loaded midi object or None if None, the midi object is loaded using midi_path

Returns *annotations.EventData* – drum event data

`mirdata.datasets.groove_midi.load_midi`(*fhandle: BinaryIO*) → Optional[pretty_midi.PrettyMIDI]

Load a Groove MIDI midi file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to midi file

Returns *midi_data (pretty_midi.PrettyMIDI)* – pretty_midi object

2.5.9 gtzan_genre

GTZAN-Genre Dataset Loader

Dataset Info

This dataset was used for the well known genre classification paper:

```
"Musical genre classification of audio signals " by G. Tzanetakis and
P. Cook in IEEE Transactions on Audio and Speech Processing 2002.
```

The dataset consists of 1000 audio tracks each 30 seconds long. It contains 10 genres, each represented by 100 tracks. The tracks are all 22050 Hz mono 16-bit audio files in .wav format.

---

class `mirdata.datasets.gtzan_genre.Dataset`(*data_home=None*)

The gtzan_genre dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a GTZAN audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.gtzan_genre.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
gtzan_genre Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path to the audio file
- **genre** (*str*) – annotated genre
- **track_id** (*str*) – track id

property audio
The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()
Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

`mirdata.datasets.gtzan_genre.load_audio`(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a GTZAN audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.10 guitarset

GuitarSet Loader

Dataset Info

GuitarSet provides audio recordings of a variety of musical excerpts played on an acoustic guitar, along with time-aligned annotations including pitch contours, string and fret positions, chords, beats, downbeats, and keys.

GuitarSet contains 360 excerpts that are close to 30 seconds in length. The 360 excerpts are the result of the following combinations:

- 6 players
- 2 versions: comping (harmonic accompaniment) and soloing (melodic improvisation)
- 5 styles: Rock, Singer-Songwriter, Bossa Nova, Jazz, and Funk
- 3 Progressions: 12 Bar Blues, Autumn Leaves, and Pachelbel Canon.
- 2 Tempi: slow and fast.

The tonality (key) of each excerpt is sampled uniformly at random.

GuitarSet was recorded with the help of a hexaphonic pickup, which outputs signals for each string separately, allowing automated note-level annotation. Excerpts are recorded with both the hexaphonic pickup and a Neumann U-87 condenser microphone as reference. 3 audio recordings are provided with each excerpt with the following suffix:

- hex: original 6 channel wave file from hexaphonic pickup
- hex_cln: hex wave files with interference removal applied
- mic: monophonic recording from reference microphone
- mix: monophonic mixture of original 6 channel file

Each of the 360 excerpts has an accompanying JAMS file which stores 16 annotations. Pitch:

- 6 pitch_contour annotations (1 per string)
- 6 midi_note annotations (1 per string)

Beat and Tempo:

- 1 beat_position annotation
- 1 tempo annotation

Chords:

- 2 chord annotations: instructed and performed. The instructed chord annotation is a digital version of the lead sheet that's provided to the player, and the performed chord annotations are inferred from note annotations, using segmentation and root from the digital lead sheet annotation.

For more details, please visit: http://github.com/marl/guitarset/

---

class `mirdata.datasets.guitarset.`**`Dataset`**(*data_home=None*)

The guitarset dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()

Print the license

load_audio(**args*, ***kwargs*)

Load a Guitarset audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_beats(**args*, ***kwargs*)
Load a Guitarset beats annotation.

Parameters **fhandle** (*str or file-like*) – File-like object or path of the jams annotation file

Returns *BeatData* – Beat data

load_chords(**args*, ***kwargs*)
Load a guitarset chord annotation.

Parameters

- **jams_path** (*str*) – Path of the jams annotation file
- **leadsheet_version** (*Bool*) – Whether or not to load the leadsheet version of the chord annotation If False, load the infered version.

Returns *ChordData* – Chord data

load_key_mode(**args*, ***kwargs*)
Load a Guitarset key-mode annotation.

Parameters **fhandle** (*str or file-like*) – File-like object or path of the jams annotation file

Returns *KeyData* – Key data

load_multitrack_audio(**args*, ***kwargs*)
Load a Guitarset multitrack audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_notes(**args*, ***kwargs*)
Load a guitarset note annotation for a given string

Parameters

- **jams_path** (*str*) – Path of the jams annotation file
- **string_num** (*int), in range(6*) – Which string to load. 0 is the Low E string, 5 is the high e string.

Returns *NoteData* – Note data for the given string

load_pitch_contour(**args*, ***kwargs*)
Load a guitarset pitch contour annotation for a given string

Parameters

- **jams_path** (*str*) – Path of the jams annotation file
- **string_num** (*int), in range(6*) – Which string to load. 0 is the Low E string, 5 is the high e string.

Returns *F0Data* – Pitch contour data for the given string

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate (*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.guitarset.**Track** (*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
guitarset Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **audio_hex_cln_path** (*str*) – path to the debleeded hex wave file
- **audio_hex_path** (*str*) – path to the original hex wave file
- **audio_mic_path** (*str*) – path to the mono wave via microphone
- **audio_mix_path** (*str*) – path to the mono wave via downmixing hex pickup
- **jams_path** (*str*) – path to the jams file
- **mode** (*str*) – one of ['solo', 'comp'] For each excerpt, players are asked to first play in 'comp' mode and later play a 'solo' version on top of the already recorded comp.
- **player_id** (*str*) – ID of the different players. one of ['00', '01', . . . , '05']
- **style** (*str*) – one of ['Jazz', 'Bossa Nova', 'Rock', 'Singer-Songwriter', 'Funk']
- **tempo** (*float*) – BPM of the track
- **track_id** (*str*) – track id

Other Parameters

- **beats** (*BeatData*) – beat positions
- **leadsheet_chords** (*ChordData*) – chords as written in the leadsheet
- **inferred_chords** (*ChordData*) – chords inferred from played transcription
- **key_mode** (*KeyData*) – key and mode
- **pitch_contours** (*dict*) – Pitch contours per string - 'E': F0Data(. . .) - 'A': F0Data(. . .) - 'D': F0Data(. . .) - 'G': F0Data(. . .) - 'B': F0Data(. . .) - 'e': F0Data(. . .)
- **notes** (*dict*) – Notes per string - 'E': NoteData(. . .) - 'A': NoteData(. . .) - 'D': NoteData(. . .) - 'G': NoteData(. . .) - 'B': NoteData(. . .) - 'e': NoteData(. . .)

property audio_hex
Hexaphonic audio (6-channels) with one channel per string

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_hex_cln

Hexaphonic audio (6-channels) with one channel per string after bleed removal

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_mic

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_mix

Mixture audio (mono)

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.guitarset.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Guitarset audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.guitarset.**load_beats**(*fhandle: TextIO*) → *mirdata.annotations.BeatData*

Load a Guitarset beats annotation.

Parameters **fhandle** (*str or file-like*) – File-like object or path of the jams annotation file

Returns *BeatData* – Beat data

mirdata.datasets.guitarset.**load_chords**(*jams_path*, *leadsheet_version=True*)

Load a guitarset chord annotation.

Parameters

- **jams_path** (*str*) – Path of the jams annotation file
- **leadsheet_version** (*Bool*) – Whether or not to load the leadsheet version of the chord annotation If False, load the infered version.

Returns *ChordData* – Chord data

mirdata.datasets.guitarset.**load_key_mode**(*fhandle: TextIO*) → *mirdata.annotations.KeyData*

Load a Guitarset key-mode annotation.

Parameters **fhandle** (*str or file-like*) – File-like object or path of the jams annotation file

Returns *KeyData* – Key data

`mirdata.datasets.guitarset.load_multitrack_audio(fhandle: BinaryIO) → Tuple[numpy.ndarray, float]`

Load a Guitarset multitrack audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.guitarset.load_notes(jams_path, string_num)`

Load a guitarset note annotation for a given string

Parameters

- **jams_path** (*str*) – Path of the jams annotation file
- **string_num** (*int), in range(6*) – Which string to load. 0 is the Low E string, 5 is the high e string.

Returns *NoteData* – Note data for the given string

`mirdata.datasets.guitarset.load_pitch_contour(jams_path, string_num)`

Load a guitarset pitch contour annotation for a given string

Parameters

- **jams_path** (*str*) – Path of the jams annotation file
- **string_num** (*int), in range(6*) – Which string to load. 0 is the Low E string, 5 is the high e string.

Returns *F0Data* – Pitch contour data for the given string

2.5.11 ikala

iKala Dataset Loader

Dataset Info

The iKala dataset is comprised of 252 30-second excerpts sampled from 206 iKala songs (plus 100 hidden excerpts reserved for MIREX). The music accompaniment and the singing voice are recorded at the left and right channels respectively and can be found under the Wavfile directory. In addition, the human-labeled pitch contours and timestamped lyrics can be found under PitchLabel and Lyrics respectively.

For more details, please visit: http://mac.citi.sinica.edu.tw/ikala/

---

class `mirdata.datasets.ikala.Dataset(data_home=None)`

The ikala dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded

- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()
Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()
Print the reference

property default_path
Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_f0(**args*, ***kwargs*)
Load an ikala f0 annotation

Parameters **fhandle** (*str or file-like*) – File-like object or path to f0 annotation file

Raises **IOError** – If f0_path does not exist

Returns *F0Data* – the f0 annotation data

load_instrumental_audio(**args*, ***kwargs*)
Load ikala instrumental audio

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - audio signal
- float - sample rate

load_lyrics(**args*, ***kwargs*)
Load an ikala lyrics annotation

Parameters **fhandle** (*str or file-like*) – File-like object or path to lyric annotation file

Raises **IOError** – if lyrics_path does not exist

Returns *LyricData* – lyric annotation data

load_mix_audio(**args*, ***kwargs*)
Load an ikala mix.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - audio signal
- float - sample rate

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

load_vocal_audio(**args*, ***kwargs*)
Load ikala vocal audio

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - audio signal
- float - sample rate

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.ikala.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
ikala Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path to the track's audio file
- **f0_path** (*str*) – path to the track's f0 annotation file
- **lyrics_path** (*str*) – path to the track's lyric annotation file
- **section** (*str*) – section. Either 'verse' or 'chorus'
- **singer_id** (*str*) – singer id
- **song_id** (*str*) – song id of the track
- **track_id** (*str*) – track id

Other Parameters

- **f0** (*F0Data*) – human-annotated singing voice pitch

- **lyrics** (*LyricsData*) – human-annotated lyrics

property instrumental_audio
instrumental audio (mono)

Returns

- np.ndarray - audio signal
- float - sample rate

property mix_audio
mixture audio (mono)

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()
Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

property vocal_audio
solo vocal audio (mono)

Returns

- np.ndarray - audio signal
- float - sample rate

mirdata.datasets.ikala.**load_f0**(*fhandle: TextIO*) → *mirdata.annotations.F0Data*
Load an ikala f0 annotation

Parameters **fhandle** (*str or file-like*) – File-like object or path to f0 annotation file

Raises **IOError** – If f0_path does not exist

Returns *F0Data* – the f0 annotation data

mirdata.datasets.ikala.**load_instrumental_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]
Load ikala instrumental audio

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - audio signal
- float - sample rate

mirdata.datasets.ikala.**load_lyrics**(*fhandle: TextIO*) → *mirdata.annotations.LyricData*
Load an ikala lyrics annotation

Parameters **fhandle** (*str or file-like*) – File-like object or path to lyric annotation file

Raises **IOError** – if lyrics_path does not exist

Returns *LyricData* – lyric annotation data

mirdata.datasets.ikala.**load_mix_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]
Load an ikala mix.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - audio signal
- float - sample rate

`mirdata.datasets.ikala.load_vocal_audio` (*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load ikala vocal audio

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - audio signal
- float - sample rate

2.5.12 irmas

IRMAS Loader

Dataset Info

IRMAS: a dataset for instrument recognition in musical audio signals

This dataset includes musical audio excerpts with annotations of the predominant instrument(s) present. It was used for the evaluation in the following article:

```
Bosch, J. J., Janer, J., Fuhrmann, F., & Herrera, P. "A Comparison of Sound␣
↪Segregation Techniques for
Predominant Instrument Recognition in Musical Audio Signals", in Proc. ISMIR (pp. 559-
↪564), 2012.
```

IRMAS is intended to be used for training and testing methods for the automatic recognition of predominant instruments in musical audio. The instruments considered are: cello, clarinet, flute, acoustic guitar, electric guitar, organ, piano, saxophone, trumpet, violin, and human singing voice. This dataset is derived from the one compiled by Ferdinand Fuhrmann in his PhD thesis, with the difference that we provide audio data in stereo format, the annotations in the testing dataset are limited to specific pitched instruments, and there is a different amount and lenght of excerpts from the original dataset.

The dataset is split into training and test data.

Training data

Total audio samples: 6705 They are excerpts of 3 seconds from more than 2000 distinct recordings.

Audio specifications

- Sampling frequency: 44.1 kHz
- Bit-depth: 16 bit
- Audio format: .wav

IRMAS Dataset trainig samples are annotated by storing the information of each track in their filenames.

- Predominant instrument:
  - The annotation of the predominant instrument of each excerpt is both in the name of the containing folder, and in the file name: cello (cel), clarinet (cla), flute (flu), acoustic guitar (gac), electric guitar (gel), organ (org), piano (pia), saxophone (sax), trumpet (tru), violin (vio), and human singing voice (voi).

  - The number of files per instrument are: cel(388), cla(505), flu(451), gac(637), gel(760), org(682), pia(721), sax(626), tru(577), vio(580), voi(778).
- Drum presence
  - Additionally, some of the files have annotation in the filename regarding the presence ([dru]) or non presence([nod]) of drums.
- The annotation of the musical genre:
  - country-folk ([cou_fol])
  - classical ([cla]),
  - pop-rock ([pop_roc])
  - latin-soul ([lat_sou])
  - jazz-blues ([jaz_blu]).

Testing data

Total audio samples: 2874

Audio specifications

- Sampling frequency: 44.1 kHz
- Bit-depth: 16 bit
- Audio format: .wav

IRMAS Dataset testing samples are annotated by the following basis:

- Predominant instrument:

  The annotations for an excerpt named: "excerptName.wav" are given in "excerptName.txt". More than one instrument may be annotated in each excerpt, one label per line. This part of the dataset contains excerpts from a diversity of western musical genres, with varied instrumentations, and it is derived from the original testing dataset from Fuhrmann (http://www.dtic.upf.edu/~ffuhrmann/PhD/). Instrument nomenclatures are the same as the training dataset.

Dataset compiled by Juan J. Bosch, Ferdinand Fuhrmann, Perfecto Herrera, Music Technology Group - Universitat Pompeu Fabra (Barcelona).

For more details, please visit: https://www.upf.edu/web/mtg/irmas

---

class `mirdata.datasets.irmas.`**`Dataset`**(*data_home=None*)

The irmas dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()
Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()
Print the reference

property default_path
Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a IRMAS dataset audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_pred_inst(**args*, ***kwargs*)
Load predominant instrument of track

Parameters **fhandle** (*str or file-like*) – File-like object or path where the test annotations are stored.

Returns *list(str)* – test track predominant instrument(s) annotations

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class `mirdata.datasets.irmas.`**`Track`**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

IRMAS track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets/Mridangam-Stroke*

Variables

- **`track_id`** (`str`) – track id
- **`predominant_instrument`** (`list`) – Training tracks predominant instrument
- **`train`** (`bool`) – flag to identify if the track is from the training of the testing dataset
- **`genre`** (`str`) – string containing the namecode of the genre of the track.
- **`drum`** (`bool`) – flag to identify if the track contains drums or not.

Other Parameters **instrument** (*list*) – list of predominant instruments as str

`property audio`

The track's audio signal

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`to_jams`()

the track's data in jams format

Returns *jams.JAMS* – return track data in jam format

`mirdata.datasets.irmas.`**`load_audio`**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a IRMAS dataset audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.irmas.`**`load_pred_inst`**(*fhandle: TextIO*) → List[str]

Load predominant instrument of track

Parameters **fhandle** (*str or file-like*) – File-like object or path where the test annotations are stored.

Returns *list(str)* – test track predominant instrument(s) annotations

2.5.13 maestro

MAESTRO Dataset Loader

Dataset Info

MAESTRO (MIDI and Audio Edited for Synchronous TRacks and Organization) is a dataset composed of over 200 hours of virtuosic piano performances captured with fine alignment (~3 ms) between note labels and audio waveforms.

The dataset is created and released by Google's Magenta team.

The dataset contains over 200 hours of paired audio and MIDI recordings from ten years of International Piano-e-Competition. The MIDI data includes key strike velocities and sustain/sostenuto/una corda pedal positions. Audio and MIDI files are aligned with 3 ms accuracy and sliced to individual musical pieces, which are annotated with composer, title, and year of performance. Uncompressed audio is of CD quality or higher (44.1–48 kHz 16-bit PCM stereo).

A train/validation/test split configuration is also proposed, so that the same composition, even if performed by multiple contestants, does not appear in multiple subsets. Repertoire is mostly classical, including composers from the 17th to early 20th century.

The dataset is made available by Google LLC under a Creative Commons Attribution Non-Commercial Share-Alike 4.0 (CC BY-NC-SA 4.0) license.

This loader supports MAESTRO version 2.

For more details, please visit: https://magenta.tensorflow.org/datasets/maestro

---

class `mirdata.datasets.maestro.`**`Dataset`**(*data_home=None*)

The maestro dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download the dataset

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded

- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()

Print the license

load_audio(**args*, ***kwargs*)

Load a MAESTRO audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_midi(**args*, ***kwargs*)

Load a MAESTRO midi file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to midi file

Returns *pretty_midi.PrettyMIDI* – pretty_midi object

load_notes(**args*, ***kwargs*)

Load note data from the midi file.

Parameters

- **midi_path** (*str*) – path to midi file
- **midi** (*pretty_midi.PrettyMIDI*) – pre-loaded midi object or None if None, the midi object is loaded using midi_path

Returns *NoteData* – note annotations

load_tracks()

Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids

Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.maestro.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

MAESTRO Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **audio_path** (*str*) – Path to the track's audio file
- **canonical_composer** (*str*) – Composer of the piece, standardized on a single spelling for a given name.
- **canonical_title** (*str*) – Title of the piece. Not guaranteed to be standardized to a single representation.
- **duration** (*float*) – Duration in seconds, based on the MIDI file.
- **midi_path** (*str*) – Path to the track's MIDI file
- **split** (*str*) – Suggested train/validation/test split.
- **track_id** (*str*) – track id
- **year** (*int*) – Year of performance.

Cached Property: midi (pretty_midi.PrettyMIDI): object containing MIDI annotations notes (NoteData): annotated piano notes

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.maestro.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a MAESTRO audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.maestro.**load_midi**(*fhandle: BinaryIO*) → pretty_midi.PrettyMIDI

Load a MAESTRO midi file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to midi file

Returns *pretty_midi.PrettyMIDI* – pretty_midi object

mirdata.datasets.maestro.**load_notes**(*midi_path*, *midi=None*)

Load note data from the midi file.

Parameters

- **midi_path** (*str*) – path to midi file
- **midi** (*pretty_midi.PrettyMIDI*) – pre-loaded midi object or None if None, the midi object is loaded using midi_path

Returns *NoteData* – note annotations

2.5.14 medley_solos_db

Medley-solos-DB Dataset Loader.

Dataset Info

Medley-solos-DB is a cross-collection dataset for automatic musical instrument recognition in solo recordings. It consists of a training set of 3-second audio clips, which are extracted from the MedleyDB dataset (Bittner et al., ISMIR 2014) as well as a test set of 3-second clips, which are extracted from the solosDB dataset (Essid et al., IEEE TASLP 2009).

Each of these clips contains a single instrument among a taxonomy of eight:

0. clarinet,
1. distorted electric guitar,
2. female singer,
3. flute,
4. piano,
5. tenor saxophone,
6. trumpet, and
7. violin.

The Medley-solos-DB dataset is the dataset that is used in the benchmarks of musical instrument recognition in the publications of Lostanlen and Cella (ISMIR 2016) and Andén et al. (IEEE TSP 2019).

---

class `mirdata.datasets.medley_solos_db.`**`Dataset`**(*data_home=None*)

The medley_solos_db dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a Medley Solos DB audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.medley_solos_db.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
medley_solos_db Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path to the track's audio file
- **instrument** (*str*) – instrument encoded by its English name
- **instrument_id** (*int*) – instrument encoded as an integer

- **song_id** (*int*) – song encoded as an integer
- **subset** (*str*) – either equal to 'train', 'validation', or 'test'
- **track_id** (*str*) – track id

property audio
The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()
Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.medley_solos_db.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Medley Solos DB audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.15 medleydb_melody

MedleyDB melody Dataset Loader

Dataset Info

MedleyDB melody is a subset of the MedleyDB dataset containing only the mixtures and melody annotations.

MedleyDB is a dataset of annotated, royalty-free multitrack recordings. MedleyDB was curated primarily to support research on melody extraction, addressing important shortcomings of existing collections. For each song we provide melody f0 annotations as well as instrument activations for evaluating automatic instrument recognition.

For more details, please visit: https://medleydb.weebly.com

---

class mirdata.datasets.medleydb_melody.**Dataset**(*data_home=None*)
The medleydb_melody dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()
Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()
Print the reference

property default_path
Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a MedleyDB audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_melody(**args*, ***kwargs*)
Load a MedleyDB melody1 or melody2 annotation file

Parameters **fhandle** (*str or file-like*) – File-like object or path to a melody annotation file

Raises **IOError** – if melody_path does not exist

Returns *F0Data* – melody data

load_melody3(**args*, ***kwargs*)
Load a MedleyDB melody3 annotation file

Parameters **fhandle** (*str or file-like*) – File-like object or melody 3 melody annotation path

Raises **IOError** – if melody_path does not exist

Returns *MultiF0Data* – melody 3 annotation data

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids

Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.medleydb_melody.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

medleydb_melody Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- ***artist*** (*str*) – artist
- **audio_path** (*str*) – path to the audio file
- **genre** (*str*) – genre
- **is_excerpt** (*bool*) – True if the track is an excerpt
- **is_instrumental** (*bool*) – True of the track does not contain vocals
- **melody1_path** (*str*) – path to the melody1 annotation file
- **melody2_path** (*str*) – path to the melody2 annotation file
- **melody3_path** (*str*) – path to the melody3 annotation file
- **n_sources** (*int*) – Number of instruments in the track
- ***title*** (*str*) – title
- **track_id** (*str*) – track id

Other Parameters

- **melody1** (*F0Data*) – the pitch of the single most predominant source (often the voice)
- **melody2** (*F0Data*) – the pitch of the predominant source for each point in time
- **melody3** (*MultiF0Data*) – the pitch of any melodic source. Allows for more than one f0 value at a time

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

`mirdata.datasets.medleydb_melody.load_audio` (*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a MedleyDB audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.medleydb_melody.load_melody` (*fhandle: TextIO*) → *mirdata.annotations.F0Data*

Load a MedleyDB melody1 or melody2 annotation file

Parameters **fhandle** (*str or file-like*) – File-like object or path to a melody annotation file

Raises **IOError** – if melody_path does not exist

Returns *F0Data* – melody data

`mirdata.datasets.medleydb_melody.load_melody3` (*fhandle: TextIO*) → *mirdata.annotations.MultiF0Data*

Load a MedleyDB melody3 annotation file

Parameters **fhandle** (*str or file-like*) – File-like object or melody 3 melody annotation path

Raises **IOError** – if melody_path does not exist

Returns *MultiF0Data* – melody 3 annotation data

2.5.16 medleydb_pitch

MedleyDB pitch Dataset Loader

Dataset Info

MedleyDB Pitch is a pitch-tracking subset of the MedleyDB dataset containing only f0-annotated, monophonic stems.

MedleyDB is a dataset of annotated, royalty-free multitrack recordings. MedleyDB was curated primarily to support research on melody extraction, addressing important shortcomings of existing collections. For each song we provide melody f0 annotations as well as instrument activations for evaluating automatic instrument recognition.

For more details, please visit: https://medleydb.weebly.com

---

class `mirdata.datasets.medleydb_pitch.Dataset` (*data_home=None*)

The medleydb_pitch dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()
Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()
Print the reference

property default_path
Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a MedleyDB audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_pitch(**args*, ***kwargs*)
load a MedleyDB pitch annotation file

Parameters **pitch_path** (*str*) – path to pitch annotation file

Raises **IOError** – if pitch_path doesn't exist

Returns *F0Data* – pitch annotation

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class `mirdata.datasets.medleydb_pitch.`**`Track`**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

medleydb_pitch Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **`artist`** (*str*) – artist
- **`audio_path`** (*str*) – path to the audio file
- **`genre`** (*str*) – genre
- **`instrument`** (*str*) – instrument of the track
- **`pitch_path`** (*str*) – path to the pitch annotation file
- **`title`** (*str*) – title
- **`track_id`** (*str*) – track id

Other Parameters **pitch** (*F0Data*) – human annotated pitch

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

`mirdata.datasets.medleydb_pitch.`**`load_audio`**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a MedleyDB audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.medleydb_pitch.`**`load_pitch`**(*fhandle: TextIO*) → *mirdata.annotations.F0Data*

load a MedleyDB pitch annotation file

Parameters **pitch_path** (*str*) – path to pitch annotation file

Raises **`IOError`** – if pitch_path doesn't exist

Returns *F0Data* – pitch annotation

2.5.17 mridangam_stroke

Mridangam Stroke Dataset Loader

Dataset Info

The Mridangam Stroke dataset is a collection of individual strokes of the Mridangam in various tonics. The dataset comprises of 10 different strokes played on Mridangams with 6 different tonic values. The audio examples were recorded from a professional Carnatic percussionist in a semi-anechoic studio conditions by Akshay Anantapadmanabhan.

Total audio samples: 6977

Used microphones:

- SM-58 microphones
- H4n ZOOM recorder.

Audio specifications:

- Sampling frequency: 44.1 kHz
- Bit-depth: 16 bit
- Audio format: .wav

The dataset can be used for training models for each Mridangam stroke. The presentation of the dataset took place on the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2013) on May 2013. You can read the full publication here: https://repositori.upf.edu/handle/10230/25756

Mridangam Dataset is annotated by storing the informat of each track in their filenames. The structure of the filename is:

```
<TrackID>__<AuthorName>__<StrokeName>-<Tonic>-<InstanceNum>.wav
```

The dataset is made available by CompMusic under a Creative Commons Attribution 3.0 Unported (CC BY 3.0) License.

For more details, please visit: https://compmusic.upf.edu/mridangam-stroke-dataset

---

class `mirdata.datasets.mridangam_stroke.`**`Dataset`**(*data_home=None*)

The mridangam_stroke dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()
Print the reference

property default_path
Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a Mridangam Stroke Dataset audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.mridangam_stroke.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
Mridangam Stroke track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored.

Variables

- **track_id** (*str*) – track id
- **audio_path** (*str*) – audio path
- **stroke_name** (*str*) – name of the Mridangam stroke present in Track
- **tonic** (*str*) – tonic of the stroke in the Track

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.mridangam_stroke.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Mridangam Stroke Dataset audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.18 orchset

ORCHSET Dataset Loader

Dataset Info

Orchset is intended to be used as a dataset for the development and evaluation of melody extraction algorithms. This collection contains 64 audio excerpts focused on symphonic music with their corresponding annotation of the melody.

For more details, please visit: https://zenodo.org/record/1289786#.XREpzaeZPx6

---

class mirdata.datasets.orchset.**Dataset**(*data_home=None*)

The orchset dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset

- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()
Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()
Print the reference

property default_path
Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio_mono(**args*, ***kwargs*)
Load an Orchset audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_audio_stereo(**args*, ***kwargs*)
Load an Orchset audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the stereo audio signal
- float - The sample rate of the audio file

load_melody(**args*, ***kwargs*)
Load an Orchset melody annotation file

Parameters **fhandle** (*str or file-like*) – File-like object or path to melody annotation file

Raises **IOError** – if melody_path doesn't exist

Returns *F0Data* – melody annotation data

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.orchset.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
orchset Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **alternating_melody** (*bool*) – True if the melody alternates between instruments
- **audio_path_mono** (*str*) – path to the mono audio file
- **audio_path_stereo** (*str*) – path to the stereo audio file
- **composer** (*str*) – the work's composer
- **contains_brass** (*bool*) – True if the track contains any brass instrument
- **contains_strings** (*bool*) – True if the track contains any string instrument
- **contains_winds** (*bool*) – True if the track contains any wind instrument
- **excerpt** (*str*) – True if the track is an excerpt
- **melody_path** (*str*) – path to the melody annotation file
- **only_brass** (*bool*) – True if the track contains brass instruments only
- **only_strings** (*bool*) – True if the track contains string instruments only
- **only_winds** (*bool*) – True if the track contains wind instruments only
- **predominant_melodic_instruments** (*list*) – List of instruments which play the melody
- **track_id** (*str*) – track id
- **work** (*str*) – The musical work

Other Parameters **melody** (*F0Data*) – melody annotation

property audio_mono
the track's audio (mono)

Returns

- np.ndarray - the mono audio signal

- float - The sample rate of the audio file

property audio_stereo
the track's audio (stereo)

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

to_jams()
Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.orchset.**load_audio_mono**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load an Orchset audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.orchset.**load_audio_stereo**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load an Orchset audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the stereo audio signal
- float - The sample rate of the audio file

mirdata.datasets.orchset.**load_melody**(*fhandle: TextIO*) → *mirdata.annotations.F0Data*

Load an Orchset melody annotation file

Parameters **fhandle** (*str or file-like*) – File-like object or path to melody annotation file

Raises **IOError** – if melody_path doesn't exist

Returns *F0Data* – melody annotation data

2.5.19 rwc_classical

RWC Classical Dataset Loader

Dataset Info

The Classical Music Database consists of 50 pieces

- Symphonies: 4 pieces
- Concerti: 2 pieces
- Orchestral music: 4 pieces
- Chamber music: 10 pieces
- Solo performances: 24 pieces

- Vocal performances: 6 pieces

A note about the Beat annotations:

- 48 corresponds to the duration of a quarter note (crotchet)
- 24 corresponds to the duration of an eighth note (quaver)
- 384 corresponds to the position of a downbeat

In 4/4 time signature, they correspond as follows:

```
384: 1st beat in a measure (i.e., downbeat position)
48: 2nd beat
96: 3rd beat
144 4th beat
```

In 3/4 time signature, they correspond as follows:

```
384: 1st beat in a measure (i.e., downbeat position)
48: 2nd beat
96: 3rd beat
```

In 6/8 time signature, they correspond as follows:

```
384: 1st beat in a measure (i.e., downbeat position)
24: 2nd beat
48: 3rd beat
72: 4th beat
96: 5th beat
120: 6th beat
```

For more details, please visit: https://staff.aist.go.jp/m.goto/RWC-MDB/rwc-mdb-c.html

---

class `mirdata.datasets.rwc_classical.`**`Dataset`**(*data_home=None*)

The rwc_classical dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a RWC audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_beats(**args*, ***kwargs*)
Load rwc beat data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to beats annotation file

Returns *BeatData* – beat data

load_sections(**args*, ***kwargs*)
Load rwc section data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to sections annotation file

Returns *SectionData* – section data

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.rwc_classical.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

rwc_classical Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **artist** (*str*) – the track's artist
- **audio_path** (*str*) – path of the audio file
- **beats_path** (*str*) – path of the beat annotation file
- **category** (*str*) – One of 'Symphony', 'Concerto', 'Orchestral', 'Solo', 'Chamber', 'Vocal', or blank.
- **composer** (*str*) – Composer of this Track.
- **duration** (*float*) – Duration of the track in seconds
- **piece_number** (*str*) – Piece number of this Track, [1-50]
- **sections_path** (*str*) – path of the section annotation file
- **suffix** (*str*) – string within M01-M06
- **title** (*str*) – Title of The track.
- **track_id** (*str*) – track id
- **track_number** (*str*) – CD track number of this Track

Other Parameters

- **sections** (*SectionData*) – human-labeled section annotations
- **beats** (*BeatData*) – human-labeled beat annotations

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.rwc_classical.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a RWC audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.rwc_classical.**load_beats**(*fhandle: TextIO*) → *mirdata.annotations.BeatData*

Load rwc beat data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to beats annotation file

Returns *BeatData* – beat data

`mirdata.datasets.rwc_classical.load_sections`(*fhandle: TextIO*) → Optional[*mirdata.annotations.SectionData*]

Load rwc section data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to sections annotation file

Returns *SectionData* – section data

2.5.20 rwc_jazz

RWC Jazz Dataset Loader.

Dataset Info

The Jazz Music Database consists of 50 pieces:

- **Instrumentation variations:** 35 pieces (5 pieces × 7 instrumentations).

  The instrumentation-variation pieces were recorded to obtain different versions of the same piece; i.e., different arrangements performed by different player instrumentations. Five standard-style jazz pieces were originally composed and then performed in modern-jazz style using the following seven instrumentations:

  1. Piano solo
  2. Guitar solo
  3. Duo: Vibraphone + Piano, Flute + Piano, and Piano + Bass
  4. Piano trio: Piano + Bass + Drums
  5. Piano trio + Trumpet or Tenor saxophone
  6. Octet: Piano trio + Guitar + Alto saxophone + Baritone saxophone + Tenor saxophone × 2
  7. Piano trio + Vibraphone or Flute

- **Style variations:** 9 pieces

  The style-variation pieces were recorded to represent various styles of jazz. They include four well-known public-domain pieces and consist of

  1. Vocal jazz: 2 pieces (including "Aura Lee")
  2. Big band jazz: 2 pieces (including "The Entertainer")
  3. Modal jazz: 2 pieces
  4. Funky jazz: 2 pieces (including "Silent Night")
  5. Free jazz: 1 piece (including "Joyful, Joyful, We Adore Thee")

- **Fusion (crossover):** 6 pieces

  The fusion pieces were recorded to obtain music that combines elements of jazz with other styles such as popular, rock, and latin. They include music with an eighth-note feel, music with a sixteenth-note feel, and Latin jazz music.

For more details, please visit: https://staff.aist.go.jp/m.goto/RWC-MDB/rwc-mdb-j.html

---

class `mirdata.datasets.rwc_jazz.Dataset`(*data_home=None*)

The rwc_jazz dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()

Print the license

load_audio(**args*, ***kwargs*)

Load a RWC audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_beats(**args*, ***kwargs*)

Load rwc beat data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to beats annotation file

Returns *BeatData* – beat data

load_sections(**args*, ***kwargs*)

Load rwc section data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to sections annotation file

Returns *SectionData* – section data

load_tracks()

Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids

Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.rwc_jazz.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

rwc_jazz Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- ***artist*** (*str*) – Artist name
- **audio_path** (*str*) – path of the audio file
- **beats_path** (*str*) – path of the beat annotation file
- **duration** (*float*) – Duration of the track in seconds
- **instruments** (*str*) – list of used instruments.
- **piece_number** (*str*) – Piece number of this Track, [1-50]
- **sections_path** (*str*) – path of the section annotation file
- **suffix** (*str*) – M01-M04
- ***title*** (*str*) – Title of The track.
- **track_id** (*str*) – track id
- **track_number** (*str*) – CD track number of this Track
- **variation** (*str*) – style variations

Other Parameters

- **sections** (*SectionData*) – human-labeled section data
- **beats** (*BeatData*) – human-labeled beat data

property audio

The track's audio

Returns

- np.ndarray - audio signal

- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

2.5.21 rwc_popular

RWC Popular Dataset Loader

Dataset Info

The Popular Music Database consists of 100 songs — 20 songs with English lyrics performed in the style of popular music typical of songs on the American hit charts in the 1980s, and 80 songs with Japanese lyrics performed in the style of modern Japanese popular music typical of songs on the Japanese hit charts in the 1990s.

For more details, please visit: https://staff.aist.go.jp/m.goto/RWC-MDB/rwc-mdb-p.html

---

class mirdata.datasets.rwc_popular.**Dataset**(*data_home=None*)

The rwc_popular dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download

- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a RWC audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_beats(**args*, ***kwargs*)
Load rwc beat data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to beats annotation file

Returns *BeatData* – beat data

load_chords(**args*, ***kwargs*)
Load rwc chord data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to chord annotation file

Returns *ChordData* – chord data

load_sections(**args*, ***kwargs*)
Load rwc section data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to sections annotation file

Returns *SectionData* – section data

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

load_vocal_activity(**args*, ***kwargs*)
Load rwc vocal activity data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to vocal activity annotation file

Returns *EventData* – vocal activity data

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.rwc_popular.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

rwc_popular Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **artist** (*str*) – artist
- **audio_path** (*str*) – path of the audio file
- **beats_path** (*str*) – path of the beat annotation file
- **chords_path** (*str*) – path of the chord annotation file
- **drum_information** (*str*) – If the drum is 'Drum sequences', 'Live drums', or 'Drum loops'
- **duration** (*float*) – Duration of the track in seconds
- **instruments** (*str*) – List of used instruments
- **piece_number** (*str*) – Piece number, [1-50]
- **sections_path** (*str*) – path of the section annotation file
- **singer_information** (*str*) – could be male, female or vocal group
- **suffix** (*str*) – M01-M04
- **tempo** (*str*) – Tempo of the track in BPM
- **title** (*str*) – title
- **track_id** (*str*) – track id
- **track_number** (*str*) – CD track number
- **voca_inst_path** (*str*) – path of the vocal/instrumental annotation file

Other Parameters

- **sections** (*SectionData*) – human-labeled section annotation
- **beats** (*BeatData*) – human-labeled beat annotation
- **chords** (*ChordData*) – human-labeled chord annotation
- **vocal_instrument_activity** (*EventData*) – human-labeled vocal/instrument activity

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.rwc_popular.**load_chords**(*fhandle: TextIO*) → *mirdata.annotations.ChordData*

Load rwc chord data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to chord annotation file

Returns *ChordData* – chord data

`mirdata.datasets.rwc_popular.load_vocal_activity`(*fhandle: TextIO*) → *mirdata.annotations.EventData*

Load rwc vocal activity data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to vocal activity annotation file

Returns *EventData* – vocal activity data

2.5.22 salami

SALAMI Dataset Loader

Dataset Info

The SALAMI dataset contains Structural Annotations of a Large Amount of Music Information: the public portion contains over 2200 annotations of over 1300 unique tracks.

NB: mirdata relies on the **corrected** version of the 2.0 annotations: Details can be found at https://github.com/bmcfee/salami-data-public/tree/hierarchy-corrections and https://github.com/DDMAL/salami-data-public/pull/15.

For more details, please visit: https://github.com/DDMAL/salami-data-public

---

class `mirdata.datasets.salami.Dataset`(*data_home=None*)

The salami dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.

- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a Salami audio file.

Parameters **fhandle** (*str or file-like*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_sections(**args*, ***kwargs*)
Load salami sections data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to sectin annotation file

Returns *SectionData* – section data

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.salami.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
salami Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **annotator_1_id** (*str*) – number that identifies annotator 1
- **annotator_1_time** (*str*) – time that the annotator 1 took to complete the annotation
- **annotator_2_id** (*str*) – number that identifies annotator 1
- **annotator_2_time** (*str*) – time that the annotator 1 took to complete the annotation
- ***artist*** (*str*) – song artist

- **audio_path** (*str*) – path to the audio file
- **broad_genre** (*str*) – broad genre of the song
- **duration** (*float*) – duration of song in seconds
- **genre** (*str*) – genre of the song
- **sections_annotator1_lowercase_path** (*str*) – path to annotations in hierarchy level 1 from annotator 1
- **sections_annotator1_uppercase_path** (*str*) – path to annotations in hierarchy level 0 from annotator 1
- **sections_annotator2_lowercase_path** (*str*) – path to annotations in hierarchy level 1 from annotator 2
- **sections_annotator2_uppercase_path** (*str*) – path to annotations in hierarchy level 0 from annotator 2
- **source** (*str*) – dataset or source of song
- ***title*** (*str*) – title of the song

Other Parameters

- **sections_annotator_1_uppercase** (*SectionData*) – annotations in hierarchy level 0 from annotator 1
- **sections_annotator_1_lowercase** (*SectionData*) – annotations in hierarchy level 1 from annotator 1
- **sections_annotator_2_uppercase** (*SectionData*) – annotations in hierarchy level 0 from annotator 2
- **sections_annotator_2_lowercase** (*SectionData*) – annotations in hierarchy level 1 from annotator 2

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.salami.**load_audio**(*fhandle: str*) → Tuple[numpy.ndarray, float]

Load a Salami audio file.

Parameters **fhandle** (*str or file-like*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.salami.**load_sections**(*fhandle: TextIO*) → *mirdata.annotations.SectionData*

Load salami sections data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to sectin annotation file

Returns *SectionData* – section data

2.5.23 saraga_carnatic

Saraga Dataset Loader

Dataset Info

This dataset contains time aligned melody, rhythm and structural annotations of Carnatic Music tracks, extracted from the large open Indian Art Music corpora of CompMusic.

The dataset contains the following manual annotations referring to audio files:

- Section and tempo annotations stored as start and end timestamps together with the name of the section and tempo during the section (in a separate file)
- Sama annotations referring to rhythmic cycle boundaries stored as timestamps.
- Phrase annotations stored as timestamps and transcription of the phrases using solfège symbols ({S, r, R, g, G, m, M, P, d, D, n, N}).
- Audio features automatically extracted and stored: pitch and tonic.
- The annotations are stored in text files, named as the audio filename but with the respective extension at the end, for instance: "Bhuvini Dasudane.tempo-manual.txt".

The dataset contains a total of 249 tracks. A total of 168 tracks have multitrack audio.

The files of this dataset are shared with the following license: Creative Commons Attribution Non Commercial Share Alike 4.0 International

Dataset compiled by: Bozkurt, B.; Srinivasamurthy, A.; Gulati, S. and Serra, X.

For more information about the dataset as well as IAM and annotations, please refer to: https://mtg.github.io/saraga/, where a really detailed explanation of the data and annotations is published.

---

class `mirdata.datasets.saraga_carnatic.`**`Dataset`**(*data_home=None*)

The saraga_carnatic dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a Saraga Carnatic audio file.

Parameters **audio_path** (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_metadata(**args*, ***kwargs*)
Load a Saraga Carnatic metadata file

Parameters **metadata_path** (*str*) – path to metadata json file

Returns

dict –

metadata with the following fields

- title (str): Title of the piece in the track
- mbid (str): MusicBrainz ID of the track
- album_artists (list, dicts): list of dicts containing the album artists present in the track and its mbid
- artists (list, dicts): list of dicts containing information of the featuring artists in the track
- raaga (list, dict): list of dicts containing information about the raagas present in the track
- form (list, dict): list of dicts containing information about the forms present in the track
- work (list, dicts): list of dicts containing the work present in the piece, and its mbid
- taala (list, dicts): list of dicts containing the talas present in the track and its uuid
- concert (list, dicts): list of dicts containing the concert where the track is present and its mbid

load_phrases(**args*, ***kwargs*)
Load phrases

Parameters **phrases_path** (*str*) – Local path where the phrase annotation is stored. If *None*, returns None.

Returns *EventData* – phrases annotation for track

load_pitch(**args*, ***kwargs*)
Load pitch

Parameters **pitch path** (*str*) – Local path where the pitch annotation is stored. If *None*, returns None.

Returns *F0Data* – pitch annotation

load_sama(**args*, ***kwargs*)
Load sama

Parameters **sama_path** (*str*) – Local path where the sama annotation is stored. If *None*, returns None.

Returns *BeatData* – sama annotations

load_sections(**args*, ***kwargs*)
Load sections from carnatic collection

Parameters **sections_path** (*str*) – Local path where the section annotation is stored.

Returns *SectionData* – section annotations for track

load_tempo(**args*, ***kwargs*)
Load tempo from carnatic collection

Parameters **tempo_path** (*str*) – Local path where the tempo annotation is stored.

Returns

dict –

Dictionary of tempo information with the following keys:

- tempo_apm: tempo in aksharas per minute (APM)
- tempo_bpm: tempo in beats per minute (BPM)
- sama_interval: median duration (in seconds) of one tāla cycle
- beats_per_cycle: number of beats in one cycle of the tāla
- subdivisions: number of aksharas per beat of the tāla

load_tonic(**args*, ***kwargs*)
Load track absolute tonic

Parameters **tonic_path** (*str*) – Local path where the tonic path is stored. If *None*, returns None.

Returns *int* – Tonic annotation in Hz

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.saraga_carnatic.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Saraga Track Carnatic class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. default=None If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **audio_path** (*str*) – path to audio file
- **audio_ghatam_path** (*str*) – path to ghatam audio file
- **audio_mridangam_left_path** (*str*) – path to mridangam left audio file
- **audio_mridangam_right_path** (*str*) – path to mridangam right audio file
- **audio_violin_path** (*str*) – path to violin audio file
- **audio_vocal_s_path** (*str*) – path to vocal s audio file
- **audio_vocal_pat** (*str*) – path to vocal pat audio file
- **ctonic_path** (*srt*) – path to ctonic annotation file
- **pitch_path** (*srt*) – path to pitch annotation file
- **pitch_vocal_path** (*srt*) – path to vocal pitch annotation file
- **tempo_path** (*srt*) – path to tempo annotation file
- **sama_path** (*srt*) – path to sama annotation file
- **sections_path** (*srt*) – path to sections annotation file
- **phrases_path** (*srt*) – path to phrases annotation file
- **metadata_path** (*srt*) – path to metadata file

Other Parameters

- **tonic** (*float*) – tonic annotation
- **pitch** (*F0Data*) – pitch annotation
- **pitch_vocal** (*F0Data*) – vocal pitch annotation
- **tempo** (*dict*) – tempo annotations
- **sama** (*BeatData*) – sama section annotations
- **sections** (*SectionData*) – track section annotations
- **phrases** (*SectionData*) – phrase annotations
- **metadata** (*dict*) – track metadata with the following fields:

- title (str): Title of the piece in the track
- mbid (str): MusicBrainz ID of the track
- album_artists (list, dicts): list of dicts containing the album artists present in the track and its mbid
- artists (list, dicts): list of dicts containing information of the featuring artists in the track
- raaga (list, dict): list of dicts containing information about the raagas present in the track
- form (list, dict): list of dicts containing information about the forms present in the track
- work (list, dicts): list of dicts containing the work present in the piece, and its mbid
- taala (list, dicts): list of dicts containing the talas present in the track and its uuid
- concert (list, dicts): list of dicts containing the concert where the track is present and its mbid

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.saraga_carnatic.**load_audio**(*audio_path*)

Load a Saraga Carnatic audio file.

Parameters **audio_path** (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.saraga_carnatic.**load_metadata**(*metadata_path*)

Load a Saraga Carnatic metadata file

Parameters **metadata_path** (*str*) – path to metadata json file

Returns

dict –

metadata with the following fields

- title (str): Title of the piece in the track
- mbid (str): MusicBrainz ID of the track
- album_artists (list, dicts): list of dicts containing the album artists present in the track and its mbid
- artists (list, dicts): list of dicts containing information of the featuring artists in the track
- raaga (list, dict): list of dicts containing information about the raagas present in the track
- form (list, dict): list of dicts containing information about the forms present in the track

- work (list, dicts): list of dicts containing the work present in the piece, and its mbid
- taala (list, dicts): list of dicts containing the talas present in the track and its uuid
- concert (list, dicts): list of dicts containing the concert where the track is present and its mbid

`mirdata.datasets.saraga_carnatic.`**`load_phrases`**(*phrases_path*)

Load phrases

Parameters **phrases_path** (*str*) – Local path where the phrase annotation is stored. If *None*, returns None.

Returns *EventData* – phrases annotation for track

`mirdata.datasets.saraga_carnatic.`**`load_pitch`**(*pitch_path*)

Load pitch

Parameters **pitch path** (*str*) – Local path where the pitch annotation is stored. If *None*, returns None.

Returns *F0Data* – pitch annotation

`mirdata.datasets.saraga_carnatic.`**`load_sama`**(*sama_path*)

Load sama

Parameters **sama_path** (*str*) – Local path where the sama annotation is stored. If *None*, returns None.

Returns *BeatData* – sama annotations

`mirdata.datasets.saraga_carnatic.`**`load_sections`**(*sections_path*)

Load sections from carnatic collection

Parameters **sections_path** (*str*) – Local path where the section annotation is stored.

Returns *SectionData* – section annotations for track

`mirdata.datasets.saraga_carnatic.`**`load_tempo`**(*tempo_path*)

Load tempo from carnatic collection

Parameters **tempo_path** (*str*) – Local path where the tempo annotation is stored.

Returns

dict –

Dictionary of tempo information with the following keys:

- tempo_apm: tempo in aksharas per minute (APM)
- tempo_bpm: tempo in beats per minute (BPM)
- sama_interval: median duration (in seconds) of one tāla cycle
- beats_per_cycle: number of beats in one cycle of the tāla
- subdivisions: number of aksharas per beat of the tāla

`mirdata.datasets.saraga_carnatic.`**`load_tonic`**(*tonic_path*)

Load track absolute tonic

Parameters **tonic_path** (*str*) – Local path where the tonic path is stored. If *None*, returns None.

Returns *int* – Tonic annotation in Hz

2.5.24 saraga_hindustani

Saraga Dataset Loader

Dataset Info

This dataset contains time aligned melody, rhythm and structural annotations of Hindustani Music tracks, extracted from the large open Indian Art Music corpora of CompMusic.

The dataset contains the following manual annotations referring to audio files:

- Section and tempo annotations stored as start and end timestamps together with the name of the section and tempo during the section (in a separate file)
- Sama annotations referring to rhythmic cycle boundaries stored as timestamps
- Phrase annotations stored as timestamps and transcription of the phrases using solfège symbols ({S, r, R, g, G, m, M, P, d, D, n, N})
- Audio features automatically extracted and stored: pitch and tonic.
- The annotations are stored in text files, named as the audio filename but with the respective extension at the end, for instance: "Bhuvini Dasudane.tempo-manual.txt".

The dataset contains a total of 108 tracks.

The files of this dataset are shared with the following license: Creative Commons Attribution Non Commercial Share Alike 4.0 International

Dataset compiled by: Bozkurt, B.; Srinivasamurthy, A.; Gulati, S. and Serra, X.

For more information about the dataset as well as IAM and annotations, please refer to: https://mtg.github.io/saraga/, where a really detailed explanation of the data and annotations is published.

---

class `mirdata.datasets.saraga_hindustani.`**`Dataset`**(*data_home=None*)

The saraga_hindustani dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a Saraga Hindustani audio file.

Parameters **audio_path** (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_phrases(**args*, ***kwargs*)
Load phrases

Parameters **phrases_path** (*str*) – Local path where the phrase annotation is stored. If *None*, returns None.

Returns *EventData* – phrases annotation for track

load_pitch(**args*, ***kwargs*)
Load automatic extracted pitch or melody

Parameters **pitch path** (*str*) – Local path where the pitch annotation is stored. If *None*, returns None.

Returns *F0Data* – pitch annotation

load_sama(**args*, ***kwargs*)
Load sama

Parameters **sama_path** (*str*) – Local path where the sama annotation is stored. If *None*, returns None.

Returns *SectionData* – sama annotations

load_sections(**args*, ***kwargs*)
Load tracks sections

Parameters **sections_path** (*str*) – Local path where the section annotation is stored.

Returns *SectionData* – section annotations for track

load_tempo(**args*, ***kwargs*)
Load tempo from hindustani collection

Parameters **tempo_path** (*str*) – Local path where the tempo annotation is stored.

Returns

dict – Dictionary of tempo information with the following keys:

- tempo: median tempo for the section in mātrās per minute (MPM)
- matra_interval: tempo expressed as the duration of the mātra (essentially dividing 60 by tempo, expressed in seconds)
- sama_interval: median duration of one tāl cycle in the section
- matras_per_cycle: indicator of the structure of the tāl, showing the number of mātrā in a cycle of the tāl of the recording
- start_time: start time of the section
- duration: duration of the section

load_tonic(**args*, ***kwargs*)

Load track absolute tonic

Parameters **tonic_path** (*str*) – Local path where the tonic path is stored. If *None*, returns None.

Returns *int* – Tonic annotation in Hz

load_tracks()

Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids

Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.saraga_hindustani.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Saraga Hindustani Track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. default=None If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **audio_path** (*str*) – path to audio file
- **ctonic_path** (*str*) – path to ctonic annotation file
- **pitch_path** (*str*) – path to pitch annotation file
- **tempo_path** (*str*) – path to tempo annotation file

- **sama_path** (*str*) – path to sama annotation file
- **sections_path** (*str*) – path to sections annotation file
- **phrases_path** (*str*) – path to phrases annotation file
- **metadata_path** (*str*) – path to metadata annotation file

Other Parameters

- **tonic** (*float*) – tonic annotation
- **pitch** (*F0Data*) – pitch annotation
- **tempo** (*dict*) – tempo annotations
- **sama** (*BeatData*) – Sama section annotations
- **sections** (*SectionData*) – track section annotations
- **phrases** (*EventData*) – phrase annotations
- **metadata** (*dict*) – track metadata with the following fields
  - title (str): Title of the piece in the track
  - mbid (str): MusicBrainz ID of the track
  - album_artists (list, dicts): list of dicts containing the album artists present in the track and its mbid
  - artists (list, dicts): list of dicts containing information of the featuring artists in the track
  - raags (list, dict): list of dicts containing information about the raags present in the track
  - forms (list, dict): list of dicts containing information about the forms present in the track
  - release (list, dicts): list of dicts containing information of the release where the track is found
  - works (list, dicts): list of dicts containing the work present in the piece, and its mbid
  - taals (list, dicts): list of dicts containing the taals present in the track and its uuid
  - layas (list, dicts): list of dicts containing the layas present in the track and its uuid

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.saraga_hindustani.**load_audio**(*audio_path*)

Load a Saraga Hindustani audio file.

Parameters **audio_path** (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.saraga_hindustani.load_metadata(metadata_path)`

Load a Saraga Hindustani metadata file

Parameters **metadata_path** (*str*) – path to metadata json file

Returns

dict –

metadata with the following fields

- title (str): Title of the piece in the track
- mbid (str): MusicBrainz ID of the track
- album_artists (list, dicts): list of dicts containing the album artists present in the track and its mbid
- artists (list, dicts): list of dicts containing information of the featuring artists in the track
- raags (list, dict): list of dicts containing information about the raags present in the track
- forms (list, dict): list of dicts containing information about the forms present in the track
- release (list, dicts): list of dicts containing information of the release where the track is found
- works (list, dicts): list of dicts containing the work present in the piece, and its mbid
- taals (list, dicts): list of dicts containing the taals present in the track and its uuid
- layas (list, dicts): list of dicts containing the layas present in the track and its uuid

`mirdata.datasets.saraga_hindustani.load_phrases(phrases_path)`

Load phrases

Parameters **phrases_path** (*str*) – Local path where the phrase annotation is stored. If *None*, returns None.

Returns *EventData* – phrases annotation for track

`mirdata.datasets.saraga_hindustani.load_pitch(pitch_path)`

Load automatic extracted pitch or melody

Parameters **pitch path** (*str*) – Local path where the pitch annotation is stored. If *None*, returns None.

Returns *F0Data* – pitch annotation

`mirdata.datasets.saraga_hindustani.load_sama(sama_path)`

Load sama

Parameters **sama_path** (*str*) – Local path where the sama annotation is stored. If *None*, returns None.

Returns *SectionData* – sama annotations

`mirdata.datasets.saraga_hindustani.load_sections(sections_path)`

Load tracks sections

Parameters **sections_path** (*str*) – Local path where the section annotation is stored.

Returns *SectionData* – section annotations for track

`mirdata.datasets.saraga_hindustani.load_tempo(tempo_path)`

Load tempo from hindustani collection

Parameters **tempo_path** (*str*) – Local path where the tempo annotation is stored.

Returns

dict – Dictionary of tempo information with the following keys:

- tempo: median tempo for the section in mātrās per minute (MPM)
- matra_interval: tempo expressed as the duration of the mātra (essentially dividing 60 by tempo, expressed in seconds)
- sama_interval: median duration of one tāl cycle in the section
- matras_per_cycle: indicator of the structure of the tāl, showing the number of mātrā in a cycle of the tāl of the recording
- start_time: start time of the section
- duration: duration of the section

`mirdata.datasets.saraga_hindustani.load_tonic(tonic_path)`

Load track absolute tonic

Parameters **tonic_path** (*str*) – Local path where the tonic path is stored. If *None*, returns None.

Returns *int* – Tonic annotation in Hz

2.5.25 tinysol

TinySOL Dataset Loader.

Dataset Info

TinySOL is a dataset of 2913 samples, each containing a single musical note from one of 14 different instruments:

- Bass Tuba
- French Horn
- Trombone
- Trumpet in C
- Accordion
- Contrabass
- Violin
- Viola
- Violoncello
- Bassoon
- Clarinet in B-flat
- Flute
- Oboe
- Alto Saxophone

These sounds were originally recorded at Ircam in Paris (France) between 1996 and 1999, as part of a larger project named Studio On Line (SOL). Although SOL contains many combinations of mutes and extended playing techniques, TinySOL purely consists of sounds played in the so-called "ordinary" style, and in absence of mute.

TinySOL can be used for education and research purposes. In particular, it can be employed as a dataset for training and/or evaluating music information retrieval (MIR) systems, for tasks such as instrument recognition or fundamental frequency estimation. For this purpose, we provide an official 5-fold split of TinySOL as a metadata attribute. This split has been carefully balanced in terms of instrumentation, pitch range, and dynamics. For the sake of research reproducibility, we encourage users of TinySOL to adopt this split and report their results in terms of average performance across folds.

We encourage TinySOL users to subscribe to the Ircam Forum so that they can have access to larger versions of SOL.

For more details, please visit: https://www.orch-idea.org/

---

class `mirdata.datasets.tinysol.`**`Dataset`**(*data_home=None*)

The tinysol dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()

Print the license

load_audio(**args*, ***kwargs*)

Load a TinySOL audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_tracks()

Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids

Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.tinysol.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

tinysol Track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path of the audio file
- **dynamics** (*str*) – dynamics abbreviation. Ex: pp, mf, ff, etc.
- **dynamics_id** (*int*) – pp=0, p=1, mf=2, f=3, ff=4
- **family** (*str*) – instrument family encoded by its English name
- **instance_id** (*int*) – instance ID. Either equal to 0, 1, 2, or 3.
- **instrument_abbr** (*str*) – instrument abbreviation
- **instrument_full** (*str*) – instrument encoded by its English name
- **is_resampled** (*bool*) – True if this sample was pitch-shifted from a neighbor; False if it was genuinely recorded.
- **pitch** (*str*) – string containing English pitch class and octave number
- **pitch_id** (*int*) – MIDI note index, where middle C ("C4") corresponds to 60
- **string_id** (*NoneType*) – string ID. By musical convention, the first string is the highest. On wind instruments, this is replaced by *None*.
- **technique_abbr** (*str*) – playing technique abbreviation
- **technique_full** (*str*) – playing technique encoded by its English name
- **track_id** (*str*) – track id

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.tinysol.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a TinySOL audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.26 tonality_classicaldb

Tonality classicalDB Dataset Loader

Dataset Info

The Tonality classicalDB Dataset includes 881 classical musical pieces across different styles from s.XVII to s.XX annotated with single-key labels.

Tonality classicalDB Dataset was created as part of:

```
Gómez, E. (2006). PhD Thesis. Tonal description of music audio signals.
Department of Information and Communication Technologies.
```

This dataset is mainly intended to assess the performance of computational key estimation algorithms in classical music.

2020 note: The audio is privates. If you don't have the original audio collection, you could create it from your private collection because most of the recordings are well known. To this end, we provide musicbrainz metadata. Moreover, we have added the spectrum and HPCP chromagram of each audio.

This dataset can be used with mirdata library: https://github.com/mir-dataset-loaders/mirdata

Spectrum features have been computed as is shown here: https://github.com/mir-dataset-loaders/mirdata-notebooks/blob/master/Tonality_classicalDB/ClassicalDB_spectrum_features.ipynb

HPCP chromagram has been computed as is shown here: https://github.com/mir-dataset-loaders/mirdata-notebooks/blob/master/Tonality_classicalDB/ClassicalDB_HPCP_features.ipynb

Musicbrainz metadata has been computed as is shown here: https://github.com/mir-dataset-loaders/mirdata-notebooks/blob/master/Tonality_classicalDB/ClassicalDB_musicbrainz_metadata.ipynb

---

class mirdata.datasets.tonality_classicaldb.**Dataset**(*data_home=None*)

The tonality_classicaldb dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset

- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

choice_track()
Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()
Print the reference

property default_path
Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)
Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()
Print the license

load_audio(**args*, ***kwargs*)
Load a Tonality classicalDB audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

load_hpcp(**args*, ***kwargs*)
Load Tonality classicalDB HPCP feature from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to HPCP file

Returns *np.ndarray* – loaded HPCP data

load_key(**args*, ***kwargs*)
Load Tonality classicalDB format key data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to key annotation file

Returns *str* – musical key data

load_musicbrainz(**args*, ***kwargs*)
Load Tonality classicalDB musicbraiz metadata from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to musicbrainz metadata file

Returns *dict* – musicbrainz metadata

load_spectrum(**args*, ***kwargs*)
Load Tonality classicalDB spectrum data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to spectrum file

Returns *np.ndarray* – spectrum data

load_tracks()
Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.tonality_classicaldb.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
tonality_classicaldb track class

Parameters **track_id** (*str*) – track id of the track

Variables

- **audio_path** (*str*) – track audio path
- **key_path** (*str*) – key annotation path
- ***title*** (*str*) – title of the track
- **track_id** (*str*) – track id

Other Parameters

- **key** (*str*) – key annotation
- **spectrum** (*np.array*) – computed audio spectrum
- **hpcp** (*np.array*) – computed hpcp
- **musicbrainz_metadata** (*dict*) – MusicBrainz metadata

property audio
The track's audio

Returns

- np.ndarray - audio signal

- float - sample rate

to_jams()

Get the track's data in jams format

Returns *jams.JAMS* – the track's data in jams format

mirdata.datasets.tonality_classicaldb.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Tonality classicalDB audio file.

Parameters **fhandle** (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.tonality_classicaldb.**load_hpcp**(*fhandle: TextIO*) → numpy.ndarray

Load Tonality classicalDB HPCP feature from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to HPCP file

Returns *np.ndarray* – loaded HPCP data

mirdata.datasets.tonality_classicaldb.**load_key**(*fhandle: TextIO*) → str

Load Tonality classicalDB format key data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to key annotation file

Returns *str* – musical key data

mirdata.datasets.tonality_classicaldb.**load_musicbrainz**(*fhandle: TextIO*) → Dict[Any, Any]

Load Tonality classicalDB musicbraiz metadata from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to musicbrainz metadata file

Returns *dict* – musicbrainz metadata

mirdata.datasets.tonality_classicaldb.**load_spectrum**(*fhandle: TextIO*) → numpy.ndarray

Load Tonality classicalDB spectrum data from a file

Parameters **fhandle** (*str or file-like*) – File-like object or path to spectrum file

Returns *np.ndarray* – spectrum data

2.6 Core

Core mirdata classes

class mirdata.core.**Dataset**(*data_home=None, name=None, track_class=None, bibtex=None, remotes=None, download_info=None, license_info=None, custom_index_path=None*)

mirdata Dataset class

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format

- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

__init__(*data_home=None*, *name=None*, *track_class=None*, *bibtex=None*, *remotes=None*, *download_info=None*, *license_info=None*, *custom_index_path=None*)

Dataset init method

Parameters

- **data_home** (*str or None*) – path where mirdata will look for the dataset
- **name** (*str or None*) – the identifier of the dataset
- **track_class** (*mirdata.core.Track or None*) – a Track class
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **download_info** (*str or None*) – download instructions or caveats
- **license_info** (*str or None*) – license of the dataset
- **custom_index_path** (*str or None*) – overwrites the default index path for remote indexes

choice_track()

Choose a random track

Returns *Track* – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns *str* – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

license()

Print the license

load_tracks()

Load all tracks in the dataset

Returns *dict* – {*track_id*: track data}

Raises **NotImplementedError** – If the dataset does not support Tracks

track_ids
Return track ids

Returns *list* – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters **verbose** (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.core.**MultiTrack**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata=None*)
MultiTrack class.

A multitrack class is a collection of track objects and their associated audio that can be mixed together. A multitrack is iteslf a Track, and can have its own associated audio (such as a mastered mix), its own metadata and its own annotations.

get_mix()
Create a linear mixture given a subset of tracks.

Parameters **track_keys** (*list*) – list of track keys to mix together

Returns *np.ndarray* – mixture audio with shape (n_samples, n_channels)

get_random_target(*n_tracks=None*, *min_weight=0.3*, *max_weight=1.0*)
Get a random target by combining a random selection of tracks with random weights

Parameters

- **n_tracks** (*int or None*) – number of tracks to randomly mix. If None, uses all tracks
- **min_weight** (*float*) – minimum possible weight when mixing
- **max_weight** (*float*) – maximum possible weight when mixing

Returns

- np.ndarray - mixture audio with shape (n_samples, n_channels)
- list - list of keys of included tracks
- list - list of weights used to mix tracks

get_target(*track_keys*, *weights=None*, *average=True*, *enforce_length=True*)
Get target which is a linear mixture of tracks

Parameters

- **track_keys** (*list*) – list of track keys to mix together
- **weights** (*list or None*) – list of positive scalars to be used in the average
- **average** (*bool*) – if True, computes a weighted average of the tracks if False, computes a weighted sum of the tracks
- **enforce_length** (*bool*) – If True, raises ValueError if the tracks are not the same length. If False, pads audio with zeros to match the length of the longest track

Returns *np.ndarray* – target audio with shape (n_channels, n_samples)

Raises **ValueError** – if sample rates of the tracks are not equal if enforce_length=True and lengths are not equal

class mirdata.core.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata=None*)

Track base class

See the docs for each dataset loader's Track class for details

__init__(*track_id*, *data_home*, *dataset_name*, *index*, *metadata=None*)

Track init method. Sets boilerplate attributes, including:

- `track_id`
- `_dataset_name`
- `_data_home`
- `_track_paths`
- `_track_metadata`

Parameters

- **track_id** (*str*) – track id
- **data_home** (*str*) – path where mirdata will look for the dataset
- **dataset_name** (*str*) – the identifier of the dataset
- **index** (*dict*) – the dataset's file index
- **metadata** (*dict or None*) – a dictionary of metadata or None

class mirdata.core.**cached_property**(*func*)

Cached propery decorator

A property that is only computed once per instance and then replaces itself with an ordinary attribute. Deleting the attribute resets the property. Source: https://github.com/bottlepy/bottle/commit/fa7733e075da0d790d809aa3d2f53071897e6f76

mirdata.core.**copy_docs**(*original*)

Decorator function to copy docs from one function to another

mirdata.core.**docstring_inherit**(*parent*)

Decorator function to inherit docstrings from the parent class.

Adds documented Attributes from the parent to the child docs.

mirdata.core.**none_path_join**(*partial_path_list*)

Join a list of partial paths. If any part of the path is None, returns None.

Parameters **partial_path_list** (*list*) – List of partial paths

Returns *str or None* – joined path string or None

2.7 Annotations

mirdata annotation data types

class mirdata.annotations.**Annotation**

Annotation base class

class mirdata.annotations.**BeatData**(*times*, *positions=None*)

BeatData class

Variables

- **times** (*np.ndarray*) – array of time stamps (as floats) in seconds with positive, strictly increasing values
- **positions** (*np.ndarray or None*) – array of beat positions (as ints) e.g. 1, 2, 3, 4

class mirdata.annotations.**ChordData**(*intervals*, *labels*, *confidence=None*)

ChordData class

Variables

- **intervals** (*np.ndarray or None*) – (n x 2) array of intervals (as floats) in seconds in the form [start_time, end_time] with positive time stamps and end_time >= start_time.
- **labels** (*list*) – list chord labels (as strings)
- **confidence** (*np.ndarray or None*) – array of confidence values between 0 and 1

class mirdata.annotations.**EventData**(*intervals*, *events*)

TempoData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals (as floats) in seconds in the form [start_time, end_time] with positive time stamps and end_time >= start_time.
- **events** (*list*) – list of event labels (as strings)

class mirdata.annotations.**F0Data**(*times*, *frequencies*, *confidence=None*)

F0Data class

Variables

- **times** (*np.ndarray*) – array of time stamps (as floats) in seconds with positive, strictly increasing values
- **frequencies** (*np.ndarray*) – array of frequency values (as floats) in Hz
- **confidence** (*np.ndarray or None*) – array of confidence values between 0 and 1

class mirdata.annotations.**KeyData**(*intervals*, *keys*)

KeyData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals (as floats) in seconds in the form [start_time, end_time] with positive time stamps and end_time >= start_time.
- **keys** (*list*) – list key labels (as strings)

class mirdata.annotations.**LyricData**(*intervals*, *lyrics*, *pronunciations=None*)

LyricData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals (as floats) in seconds in the form [start_time, end_time] with positive time stamps and end_time >= start_time.
- **lyrics** (*list*) – list of lyrics (as strings)
- **pronunciations** (*list or None*) – list of pronunciations (as strings)

class mirdata.annotations.**MultiF0Data**(*times*, *frequency_list*, *confidence_list=None*)

MultiF0Data class

Variables

- **times** (*np.ndarray*) – array of time stamps (as floats) in seconds with positive, strictly increasing values
- **frequency_list** (*list*) – list of lists of frequency values (as floats) in Hz
- **confidence_list** (*list or None*) – list of lists of confidence values between 0 and 1

class mirdata.annotations.**NoteData**(*intervals*, *notes*, *confidence=None*)

NoteData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals (as floats) in seconds in the form [start_time, end_time] with positive time stamps and end_time >= start_time.
- **notes** (*np.ndarray*) – array of notes (as floats) in Hz
- **confidence** (*np.ndarray or None*) – array of confidence values between 0 and 1

class mirdata.annotations.**SectionData**(*intervals*, *labels=None*)

SectionData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals (as floats) in seconds in the form [start_time, end_time] times should be positive and intervals should have non-negative duration
- **labels** (*list or None*) – list of labels (as strings)

class mirdata.annotations.**TempoData**(*intervals*, *value*, *confidence=None*)

TempoData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals (as floats) in seconds in the form [start_time, end_time] with positive time stamps and end_time >= start_time.
- **value** (*list*) – array of tempo values (as floats)
- **confidence** (*np.ndarray or None*) – array of confidence values between 0 and 1

mirdata.annotations.**validate_array_like**(*array_like*, *expected_type*, *expected_dtype*, *none_allowed=False*)

Validate that array-like object is well formed

If array_like is None, validation passes automatically.

Parameters

- **array_like** (*array-like*) – object to validate
- **expected_type** (*type*) – expected type, either list or np.ndarray
- **expected_dtype** (*type*) – expected dtype
- **none_allowed** (*bool*) – if True, allows array to be None

Raises

- **TypeError** – if type/dtype does not match expected_type/expected_dtype
- **ValueError** – if array

`mirdata.annotations.validate_confidence(confidence)`
Validate if confidence is well-formed.

If confidence is None, validation passes automatically

Parameters **confidence** (*np.ndarray*) – an array of confidence values

Raises **ValueError** – if confidence are not between 0 and 1

`mirdata.annotations.validate_intervals(intervals)`
Validate if intervals are well-formed.

If intervals is None, validation passes automatically

Parameters **intervals** (*np.ndarray*) – (n x 2) array

Raises

- **ValueError** – if intervals have an invalid shape, have negative values
- **or if end times are smaller than start times.** –

`mirdata.annotations.validate_lengths_equal(array_list)`
Validate that arrays in list are equal in length

Some arrays may be None, and the validation for these are skipped.

Parameters **array_list** (*list*) – list of array-like objects

Raises **ValueError** – if arrays are not equal in length

`mirdata.annotations.validate_times(times)`
Validate if times are well-formed.

If times is None, validation passes automatically

Parameters **times** (*np.ndarray*) – an array of time stamps

Raises **ValueError** – if times have negative values or are non-increasing

2.8 Advanced

2.8.1 mirdata.validate

Utility functions for mirdata

`mirdata.validate.log_message(message, verbose=True)`
Helper function to log message

Parameters

- **message** (*str*) – message to log
- **verbose** (*bool*) – if false, the message is not logged

`mirdata.validate.md5(file_path)`
Get md5 hash of a file.

Parameters **file_path** (*str*) – File path

Returns *str* – md5 hash of data in file_path

`mirdata.validate.validate(local_path, checksum)`
Validate that a file exists and has the correct checksum

Parameters

- **local_path** (*str*) – file path
- **checksum** (*str*) – md5 checksum

Returns

- bool - True if file exists
- bool - True if checksum matches

`mirdata.validate.`**`validate_files`**(*file_dict*, *data_home*, *verbose*)

Validate files

Parameters

- **file_dict** (*dict*) – dictionary of file information
- **data_home** (*str*) – path where the data lives
- **verbose** (*bool*) – if True, show progress

Returns

- dict - missing files
- dict - files with invalid checksums

`mirdata.validate.`**`validate_index`**(*dataset_index*, *data_home*, *verbose=True*)

Validate files in a dataset's index

Parameters

- **dataset_index** (*list*) – dataset indices
- **data_home** (*str*) – Local home path that the dataset is being stored
- **verbose** (*bool*) – if true, prints validation status while running

Returns

- dict - file paths that are in the index but missing locally
- dict - file paths with differing checksums

`mirdata.validate.`**`validate_metadata`**(*file_dict*, *data_home*, *verbose*)

Validate files

Parameters

- **file_dict** (*dict*) – dictionary of file information
- **data_home** (*str*) – path where the data lives
- **verbose** (*bool*) – if True, show progress

Returns

- dict - missing files
- dict - files with invalid checksums

`mirdata.validate.`**`validator`**(*dataset_index*, *data_home*, *verbose=True*)

Checks the existence and validity of files stored locally with respect to the paths and file checksums stored in the reference index. Logs invalid checksums and missing files.

Parameters

- **dataset_index** (*list*) – dataset indices

- **data_home** (*str*) – Local home path that the dataset is being stored
- **verbose** (*bool*) – if True (default), prints missing and invalid files to stdout. Otherwise, this function is equivalent to validate_index.

Returns

missing_files (list) –

List of file paths that are in the dataset index but missing locally.

invalid_checksums (list): List of file paths that file exists in the dataset index but has a different checksum compare to the reference checksum.

2.8.2 mirdata.download_utils

Utilities for downloading from the web.

class mirdata.download_utils.**DownloadProgressBar**(*_, **__)

Wrap *tqdm* to show download progress

class mirdata.download_utils.**RemoteFileMetadata**(*filename*, *url*, *checksum*, *destination_dir=None*, *unpack_directories=None*)

The metadata for a remote file

Variables

- **filename** (*str*) – the remote file's basename
- **url** (*str*) – the remote file's url
- **checksum** (*str*) – the remote file's md5 checksum
- **destination_dir** (*str or None*) – the relative path for where to save the file
- **unpack_directories** (*list or None*) – list of relative directories. For each directory the contents will be moved to destination_dir (or data_home if not provieds)

mirdata.download_utils.**download_from_remote**(*remote*, *save_dir*, *force_overwrite*)

Download a remote dataset into path Fetch a dataset pointed by remote's url, save into path using remote's filename and ensure its integrity based on the MD5 Checksum of the downloaded file.

Adapted from scikit-learn's sklearn.datasets.base._fetch_remote.

Parameters

- **remote** (*RemoteFileMetadata*) – Named tuple containing remote dataset meta information: url, filename and checksum
- **save_dir** (*str*) – Directory to save the file to. Usually *data_home*
- **force_overwrite** (*bool*) – If True, overwrite existing file with the downloaded file. If False, does not overwrite, but checks that checksum is consistent.

Returns *str* – Full path of the created file.

mirdata.download_utils.**download_tar_file**(*tar_remote*, *save_dir*, *force_overwrite*, *cleanup*)

Download and untar a tar file.

Parameters

- **tar_remote** (*RemoteFileMetadata*) – Object containing download information
- **save_dir** (*str*) – Path to save downloaded file

- **force_overwrite** (*bool*) – If True, overwrites existing files
- **cleanup** (*bool*) – If True, remove tarfile after untarring

`mirdata.download_utils.`**`download_zip_file`**(*zip_remote*, *save_dir*, *force_overwrite*, *cleanup*)

Download and unzip a zip file.

Parameters

- **zip_remote** (*RemoteFileMetadata*) – Object containing download information
- **save_dir** (*str*) – Path to save downloaded file
- **force_overwrite** (*bool*) – If True, overwrites existing files
- **cleanup** (*bool*) – If True, remove zipfile after unziping

`mirdata.download_utils.`**`downloader`**(*save_dir*, *remotes=None*, *partial_download=None*, *info_message=None*, *force_overwrite=False*, *cleanup=False*)

Download data to *save_dir* and optionally log a message.

Parameters

- **save_dir** (*str*) – The directory to download the data
- **remotes** (*dict or None*) – A dictionary of RemoteFileMetadata tuples of data in zip format. If None, there is no data to download
- **partial_download** (*list or None*) – A list of keys to partially download the remote objects of the download dict. If None, all data is downloaded
- **info_message** (*str or None*) – A string of info to log when this function is called. If None, no string is logged.
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete the zip/tar file after extracting.

`mirdata.download_utils.`**`extractall_unicode`**(*zfile*, *out_dir*)

Extract all files inside a zip archive to a output directory.

In comparison to the zipfile, it checks for correct file name encoding

Parameters

- **zfile** (*obj*) – Zip file object created with zipfile.ZipFile
- **out_dir** (*str*) – Output folder

`mirdata.download_utils.`**`move_directory_contents`**(*source_dir*, *target_dir*)

Move the contents of source_dir into target_dir, and delete source_dir

Parameters

- **source_dir** (*str*) – path to source directory
- **target_dir** (*str*) – path to target directory

`mirdata.download_utils.`**`untar`**(*tar_path*, *cleanup*)

Untar a tar file inside it's current directory.

Parameters

- **tar_path** (*str*) – Path to tar file
- **cleanup** (*bool*) – If True, remove tarfile after untarring

`mirdata.download_utils.unzip`(*zip_path*, *cleanup*)
Unzip a zip file inside it's current directory.

Parameters

- **zip_path** (*str*) – Path to zip file
- **cleanup** (*bool*) – If True, remove zipfile after unzipping

2.8.3 mirdata.jams_utils

Utilities for converting mirdata Annotation classes to jams format.

`mirdata.jams_utils.beats_to_jams`(*beat_data*, *description=None*)
Convert beat annotations into jams format.

Parameters

- **beat_data** (*annotations.BeatData*) – beat data object
- **description** (*str*) – annotation description

Returns *jams.Annotation* – jams annotation object.

`mirdata.jams_utils.chords_to_jams`(*chord_data*, *description=None*)
Convert chord annotations into jams format.

Parameters

- **chord_data** (*annotations.ChordData*) – chord data object
- **description** (*str*) – annotation description

Returns *jams.Annotation* – jams annotation object.

`mirdata.jams_utils.events_to_jams`(*event_data*, *description=None*)
Convert events annotations into jams format.

Parameters

- **event_data** (*annotations.EventData*) – event data object
- **description** (*str*) – annotation description

Returns *jams.Annotation* – jams annotation object.

`mirdata.jams_utils.f0s_to_jams`(*f0_data*, *description=None*)
Convert f0 annotations into jams format.

Parameters

- **f0_data** (*annotations.F0Data*) – f0 annotation object
- **description** (*str*) – annotation descriptoin

Returns *jams.Annotation* – jams annotation object.

`mirdata.jams_utils.jams_converter`(*audio_path=None*, *spectrogram_path=None*, *beat_data=None*, *chord_data=None*, *note_data=None*, *f0_data=None*, *section_data=None*, *multi_section_data=None*, *tempo_data=None*, *event_data=None*, *key_data=None*, *lyrics_data=None*, *tags_gtzan_data=None*, *tags_open_data=None*, *metadata=None*)
Convert annotations from a track to JAMS format.

Parameters

- **audio_path** (*str or None*) – A path to the corresponding audio file, or None. If provided, the audio file will be read to compute the duration. If None, 'duration' must be a field in the metadata dictionary, or the resulting jam object will not validate.
- **spectrum_cante100_path** (*str or None*) – A path to the corresponding spectrum file, or None.
- **beat_data** (*list or None*) – A list of tuples of (annotations.BeatData, str), where str describes the annotation (e.g. 'beats_1').
- **chord_data** (*list or None*) – A list of tuples of (annotations.ChordData, str), where str describes the annotation.
- **note_data** (*list or None*) – A list of tuples of (annotations.NoteData, str), where str describes the annotation.
- **f0_data** (*list or None*) – A list of tuples of (annotations.F0Data, str), where str describes the annotation.
- **section_data** (*list or None*) – A list of tuples of (annotations.SectionData, str), where str describes the annotation.
- **multi_section_data** (*list or None*) – A list of tuples. Tuples in multi_section_data should contain another list of tuples, indicating annotations in the different levels e.g. ([(segments0, level0), '(segments1, level1)], annotator) and a str indicating the annotator
- **tempo_data** (*list or None*) – A list of tuples of (float, str), where float gives the tempo in bpm and str describes the annotation.
- **event_data** (*list or None*) – A list of tuples of (annotations.EventData, str), where str describes the annotation.
- **key_data** (*list or None*) – A list of tuples of (annotations.KeyData, str), where str describes the annotation.
- **lyrics_data** (*list or None*) – A list of tuples of (annotations.LyricData, str), where str describes the annotation.
- **tags_gtzan_data** (*list or None*) – A list of tuples of (str, str), where the first srt is the tag and the second is a descriptor of the annotation.
- **tags_open_data** (*list or None*) – A list of tuples of (str, str), where the first srt is the tag and the second is a descriptor of the annotation.
- **metadata** (*dict or None*) – A dictionary containing the track metadata.

Returns *jams.JAMS* – A JAMS object containing the annotations.

`mirdata.jams_utils.`**`keys_to_jams`**(*key_data*, *description*)

Convert key annotations into jams format.

Parameters

- **key_data** (*annotations.KeyData*) – key data object
- **description** (*str*) – annotation description

Returns *jams.Annotation* – jams annotation object.

`mirdata.jams_utils.`**`lyrics_to_jams`**(*lyric_data*, *description=None*)

Convert lyric annotations into jams format.

Parameters

- **lyric_data** (*annotations.LyricData*) – lyric annotation object
- **description** (*str*) – annotation descriptoin

Returns *jams.Annotation* – jams annotation object.

`mirdata.jams_utils.`**`multi_sections_to_jams`**(*multisection_data*, *description*)

Convert multi-section annotations into jams format.

Parameters

- **multisection_data** (*list*) – list of tuples of the form [(SectionData, int)]
- **description** (*str*) – annotation description

Returns *jams.Annotation* – jams annotation object.

`mirdata.jams_utils.`**`notes_to_jams`**(*note_data*, *description*)

Convert note annotations into jams format.

Parameters

- **note_data** (*annotations.NoteData*) – note data object
- **description** (*str*) – annotation description

Returns *jams.Annotation* – jams annotation object.

`mirdata.jams_utils.`**`sections_to_jams`**(*section_data*, *description=None*)

Convert section annotations into jams format.

Parameters

- **section_data** (*annotations.SectionData*) – section data object
- **description** (*str*) – annotation description

Returns *jams.Annotation* – jams annotation object.

`mirdata.jams_utils.`**`tag_to_jams`**(*tag_data*, *namespace='tag_open'*, *description=None*)

Convert lyric annotations into jams format.

Parameters

- **lyric_data** (*annotations.LyricData*) – lyric annotation object
- **namespace** (*str*) – the jams-compatible tag namespace
- **description** (*str*) – annotation descriptoin

Returns *jams.Annotation* – jams annotation object.

`mirdata.jams_utils.`**`tempos_to_jams`**(*tempo_data*, *description=None*)

Convert tempo annotations into jams format.

Parameters

- **tempo_data** (*annotations.TempoData*) – tempo data object
- **description** (*str*) – annotation description

Returns *jams.Annotation* – jams annotation object.

2.9 Contributing

We encourage contributions to mirdata, especially new dataset loaders. To contribute a new loader, follow the steps indicated below and create a Pull Request (PR) to the github repository. For any doubt or comment about your contribution, you can always submit an issue or open a discussion in the repository.

- Issue Tracker
- Source Code

2.9.1 Installing mirdata for development purposes

To install `mirdata` for development purposes:

- First run:

```
git clone https://github.com/mir-dataset-loaders/mirdata.git
```

- Then, after opening source data library you have to install the dependencies for updating the documentation and running tests:

```
pip install .
pip install .[tests]
pip install .[docs]
pip install .[dali]
```

We recommend to install pyenv to manage your Python versions and install all `mirdata` requirements. You will want to install the latest versions of Python 3.6 and 3.7. Once `pyenv` and the Python versions are configured, install `pytest`. Make sure you installed all the pytest plugins to automatically test your code successfully. Finally, run:

```
pytest tests/ --local
```

All tests should pass!

2.9.2 Writing a new dataset loader

The steps to add a new dataset loader to `mirdata` are:

1. *Create an index*
2. *Create a module*
3. *Add tests*
4. *Submit your loader*

Before starting, if your dataset **is not fully downloadable** you should:

1. Contact the mirdata team by opening an issue or PR so we can discuss how to proceed with the closed dataset.
2. Show that the version used to create the checksum is the "canonical" one, either by getting the version from the dataset creator, or by verifying equivalence with several other copies of the dataset.

To reduce friction, we will make commits on top of contributors PRs by default unless the `please-do-not-edit` flag is used.

1. Create an index

`mirdata`'s structure relies on *indexes*. Indexes are dictionaries contain information about the structure of the dataset which is necessary for the loading and validating functionalities of `mirdata`. In particular, indexes contain information about the files included in the dataset, their location and checksums. The necessary steps are:

1. To create an index, first cereate a script in `scripts/`, as `make_dataset_index.py`, which generates an index file.
2. Then run the script on the *canonical versions* of the dataset and save the index in `mirdata/datasets/indexes/` as `dataset_index.json`.

Here there is an example of an index to use as guideline:

Example Make Index Script

```
import argparse
import glob
import json
import os
from mirdata.validate import md5

DATASET_INDEX_PATH = "../mirdata/datasets/indexes/dataset_index.json"


def make_dataset_index(dataset_data_path):
    annotation_dir = os.path.join(dataset_data_path, "annotation")
    annotation_files = glob.glob(os.path.join(annotation_dir, "*.lab"))
    track_ids = sorted([os.path.basename(f).split(".")[0] for f in annotation_files])

    # top-key level metadata
    metadata_checksum = md5(os.path.join(dataset_data_path, "id_mapping.txt"))
    index_metadata = {"metadata": {"id_mapping": ("id_mapping.txt", metadata_
↪checksum)}}

    # top-key level tracks
    index_tracks = {}
    for track_id in track_ids:
        audio_checksum = md5(
            os.path.join(dataset_data_path, "Wavfile/{}.wav".format(track_id))
        )
        annotation_checksum = md5(
            os.path.join(dataset_data_path, "annotation/{}.lab".format(track_id))
        )

        index_tracks[track_id] = {
            "audio": ("Wavfile/{}.wav".format(track_id), audio_checksum),
            "annotation": ("annotation/{}.lab".format(track_id), annotation_checksum),
        }

    # top-key level version
    dataset_index = {"version": None}

    # combine all in dataset index
    dataset_index.update(index_metadata)
    dataset_index.update({"tracks": index_tracks})

    with open(DATASET_INDEX_PATH, "w") as fhandle:
```

(continues on next page)

(continued from previous page)

```
        json.dump(dataset_index, fhandle, indent=2)


def main(args):
    make_dataset_index(args.dataset_data_path)


if __name__ == "__main__":
    PARSER = argparse.ArgumentParser(description="Make dataset index file.")
    PARSER.add_argument(
        "dataset_data_path", type=str, help="Path to dataset data folder."
    )

    main(PARSER.parse_args())
```

More examples of scripts used to create dataset indexes can be found in the scripts folder.

tracks

Most MIR datasets are organized as a collection of tracks and annotations. In such case, the index should make use of the `tracks` top-level key. A dictionary should be stored under the `tracks` top-level key where the keys are the unique track ids of the dataset. The values are a dictionary of files associated with a track id, along with their checksums. These files can be for instance audio files or annotations related to the track id. File paths are relative to the top level directory of a dataset.

Index Examples - Tracks

If the version *1.0* of a given dataset has the structure:

```
> Example_Dataset/
    > audio/
        track1.wav
        track2.wav
        track3.wav
    > annotations/
        track1.csv
        Track2.csv
        track3.csv
    > metadata/
        metadata_file.csv
```

The top level directory is `Example_Dataset` and the relative path for `track1.wav` would be `audio/track1.wav`. Any unavailable fields are indicated with *null*. A possible index file for this example would be:

```
{   "version": "1.0",
    "tracks":
        "track1": {
            "audio": [
                "audio/track1.wav",  // the relative path for track1's audio file
                "912ec803b2ce49e4a541068d495ab570"  // track1.wav's md5 checksum
            ],
            "annotation": [
                "annotations/track1.csv",  // the relative path for track1's␣
↪annotation
```

(continues on next page)

(continued from previous page)

```
                "2cf33591c3b28b382668952e236cccd5"  // track1.csv's md5 checksum
            ]
        },
        "track2": {
            "audio": [
                "audio/track2.wav",
                "65d671ec9787b32cfb7e33188be32ff7"
            ],
            "annotation": [
                "annotations/Track2.csv",
                "e1964798cfe86e914af895f8d0291812"
            ]
        },
        "track3": {
            "audio": [
                "audio/track3.wav",
                "60edeb51dc4041c47c031c4bfb456b76"
            ],
            "annotation": [
                "annotations/track3.csv",
                "06cb006cc7b61de6be6361ff904654b3"
            ]
        },
    }
"metadata": {
        "metadata_file": [
            "metadata/metadata_file.csv",
            "7a41b280c7b74e2ddac5184708f9525b"
        ]
}
}
```

Note: In this example there is a (purposeful) mismatch between the name of the audio file `track2.wav` and its corresponding annotation file, `Track2.csv`, compared with the other pairs. This mismatch should be included in the index. This type of slight difference in filenames happens often in publicly available datasets, making pairing audio and annotation files more difficult. We use a fixed, version-controlled index to account for this kind of mismatch, rather than relying on string parsing on load.

multitracks

Index Examples - Multitracks

Coming soon

records

Index Examples - Records

Coming soon

2. Create a module

Once the index is created you can create the loader. For that, we suggest you use the following template and adjust it for your dataset. To quickstart a new module:

1. Copy the example below and save it to `mirdata/datasets/<your_dataset_name>.py`
2. Find & Replace `Example` with the <your_dataset_name>.
3. Remove any lines beginning with # – which are there as guidelines.

Example Module

```
"""Example Dataset Loader

.. admonition:: Dataset Info
    :class: dropdown

    Please include the following information at the top level docstring for the␣
↪dataset's module `dataset.py`:

    1. Describe annotations included in the dataset
    2. Indicate the size of the datasets (e.g. number files and duration, hours)
    3. Mention the origin of the dataset (e.g. creator, institution)
    4. Describe the type of music included in the dataset
    5. Indicate any relevant papers related to the dataset
    6. Include a description about how the data can be accessed and the license it␣
↪uses (if applicable)

"""
import csv
import logging
import json
import os

import librosa
import numpy as np
# -- import whatever you need here and remove
# -- example imports you won't use

from mirdata import download_utils
from mirdata import jams_utils
from mirdata import core, annotations

# -- Add any relevant citations here
BIBTEX = """
@article{article-minimal,
  author = "L[eslie] B. Lamport",
  title = "The Gnats and Gnus Document Preparation System",
```

(continues on next page)

(continued from previous page)

```
  journal = "G-Animal's Journal",
  year = "1986"
}
"""

# -- REMOTES is a dictionary containing all files that need to be downloaded.
# -- The keys should be descriptive (e.g. 'annotations', 'audio').
# -- When having data that can be partially downloaded, remember to set up
# -- correctly destination_dir to download the files following the correct structure.
REMOTES = {
    'remote_data': download_utils.RemoteFileMetadata(
        filename='a_zip_file.zip',
        url='http://website/hosting/the/zipfile.zip',
        checksum='00000000000000000000000000000000',  # -- the md5 checksum
        destination_dir='path/to/unzip' # -- relative path for where to unzip the␣
↪data, or None
    ),
}

# -- Include any information that should be printed when downloading
# -- remove this variable if you don't need to print anything during download
DOWNLOAD_INFO = """
Include any information you want to be printed when dataset.download() is called.
These can be instructions for how to download the dataset (e.g. request access on␣
↪zenodo),
caveats about the download, etc
"""

# -- Include the dataset's license information
LICENSE_INFO = """
The dataset's license information goes here.
"""


class Track(core.Track):
    """Example track class
    # -- YOU CAN AUTOMATICALLY GENERATE THIS DOCSTRING BY CALLING THE SCRIPT:
    # -- `scripts/print_track_docstring.py my_dataset`
    # -- note that you'll first need to have a test track (see "Adding tests to your␣
↪dataset" below)

    Args:
        track_id (str): track id of the track

    Attributes:
        track_id (str): track id
        # -- Add any of the dataset specific attributes here

    """
    def __init__(self, track_id, data_home, dataset_name, index, metadata):

        # -- this sets the following attributes:
        # -- * track_id
        # -- * _dataset_name
        # -- * _data_home
        # -- * _track_paths
        # -- * _track_metadata
```

(continues on next page)

(continued from previous page)

```
        super().__init__(
            track_id,
            data_home,
            dataset_name=dataset_name,
            index=index,
            metadata=metadata,
        )

        # -- add any dataset specific attributes here
        self.audio_path = os.path.join(
            self._data_home, self._track_paths['audio'][0])
        self.annotation_path = os.path.join(
            self._data_home, self._track_paths['annotation'][0])

    # -- `annotation` will behave like an attribute, but it will only be loaded
    # -- and saved when someone accesses it. Useful when loading slightly
    # -- bigger files or for bigger datasets. By default, we make any time
    # -- series data loaded from a file a cached property
    @core.cached_property
    def annotation(self):
        """output type: description of output"""
        return load_annotation(self.annotation_path)

    # -- `audio` will behave like an attribute, but it will only be loaded
    # -- when someone accesses it and it won't be stored. By default, we make
    # -- any memory heavy information (like audio) properties
    @property
    def audio(self):
        """(np.ndarray, float): DESCRIPTION audio signal, sample rate"""
        return load_audio(self.audio_path)

    # -- we use the to_jams function to convert all the annotations in the JAMS␣
↪format.
    # -- The converter takes as input all the annotations in the proper format (e.g.␣
↪beats
    # -- will be fed as beat_data=[(self.beats, None)], see jams_utils), and returns␣
↪a jams
    # -- object with the annotations.
    def to_jams(self):
        """Jams: the track's data in jams format"""
        return jams_utils.jams_converter(
            audio_path=self.audio_path,
            annotation_data=[(self.annotation, None)],
            metadata=self._metadata,
        )
        # -- see the documentation for `jams_utils.jams_converter for all fields


# -- if the dataset contains multitracks, you can define a MultiTrack similar to a␣
↪Track
# -- you can delete the block of code below if the dataset has no multitracks
class MultiTrack(core.MultiTrack):
    """Example multitrack class

    Args:
        mtrack_id (str): multitrack id
        data_home (str): Local path where the dataset is stored.
```

(continues on next page)

(continued from previous page)

```
            If `None`, looks for the data in the default directory, `~/mir_datasets/
↪Example`

    Attributes:
        mtrack_id (str): track id
        tracks (dict): {track_id: Track}
        track_audio_attribute (str): the name of the attribute of Track which
            returns the audio to be mixed
        # -- Add any of the dataset specific attributes here

    """
    def __init__(self, mtrack_id, data_home):
        self.mtrack_id = mtrack_id
        self._data_home = data_home
        # these three attributes below must have exactly these names
        self.track_ids = [...] # define which track_ids should be part of the␣
↪multitrack
        self.tracks = {t: Track(t, self._data_home) for t in self.track_ids}
        self.track_audio_property = "audio" # the property of Track which returns the␣
↪relevant audio file for mixing

        # -- optionally add any multitrack specific attributes here
        self.mix_path = ...  # this can be called whatever makes sense for the␣
↪datasets
        self.annotation_path = ...

    # -- multitracks can optionally have mix-level cached properties and properties
    @core.cached_property
    def annotation(self):
        """output type: description of output"""
        return load_annotation(self.annotation_path)

    @property
    def audio(self):
        """(np.ndarray, float): DESCRIPTION audio signal, sample rate"""
        return load_audio(self.audio_path)

    # -- multitrack classes are themselves Tracks, and also need a to_jams method
    # -- for any mixture-level annotations
    def to_jams(self):
        """Jams: the track's data in jams format"""
        return jams_utils.jams_converter(
            audio_path=self.mix_path,
            annotation_data=[(self.annotation, None)],
            ...
        )
        # -- see the documentation for `jams_utils.jams_converter for all fields


@io.coerce_to_bytes_io
def load_audio(fhandle):
    """Load a Example audio file.

    Args:
        fhandle (str or file-like): path or file-like object pointing to an audio file

    Returns:
```

(continues on next page)

(continued from previous page)

```
        * np.ndarray - the audio signal
        * float - The sample rate of the audio file

    """
    # -- for example, the code below. This should be dataset specific!
    # -- By default we load to mono
    # -- change this if it doesn't make sense for your dataset.
    return librosa.load(audio_path, sr=None, mono=True)


# -- Write any necessary loader functions for loading the dataset's data
@io.coerce_to_string_io
def load_annotation(fhandle):

    # -- if there are some file paths for this annotation type in this dataset's
    # -- index that are None/null, uncomment the lines below.
    # if annotation_path is None:
    #     return None

    reader = csv.reader(fhandle, delimiter=' ')
    intervals = []
    annotation = []
    for line in reader:
        intervals.append([float(line[0]), float(line[1])])
        annotation.append(line[2])

    annotation_data = annotations.EventData(
        np.array(intervals), np.array(annotation)
    )
    return annotation_data

# -- use this decorator so the docs are complete
@core.docstring_inherit(core.Dataset)
class Dataset(core.Dataset):
    """The Example dataset
    """

    def __init__(self, data_home=None):
        super().__init__(
            data_home,
            name=NAME,
            track_class=Track,
            bibtex=BIBTEX,
            remotes=REMOTES,
            download_info=DOWNLOAD_INFO,
            license_info=LICENSE_INFO,
        )

    # -- Copy any loaders you wrote that should be part of the Dataset class
    # -- use this core.copy_docs decorator to copy the docs from the original
    # -- load_ function
    @core.copy_docs(load_audio)
    def load_audio(self, *args, **kwargs):
        return load_audio(*args, **kwargs)

    @core.copy_docs(load_annotation)
    def load_annotation(self, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        return load_annotation(*args, **kwargs)

    # -- if your dataset has a top-level metadata file, write a loader for it here
    # -- you do not have to include this function if there is no metadata
    @core.cached_property
    def _metadata(self):
        metadata_path = os.path.join(self.data_home, 'example_metadta.csv')

        # load metadata however makes sense for your dataset
        metadata_path = os.path.join(data_home, 'example_metadata.json')
        with open(metadata_path, 'r') as fhandle:
            metadata = json.load(fhandle)

        return metadata

    # -- if your dataset needs to overwrite the default download logic, do it here.
    # -- this function is usually not necessary unless you need very custom download␣
↪logic
    def download(
        self, partial_download=None, force_overwrite=False, cleanup=False
    ):
        """Download the dataset

        Args:
            partial_download (list or None):
                A list of keys of remotes to partially download.
                If None, all data is downloaded
            force_overwrite (bool):
                If True, existing files are overwritten by the downloaded files.
            cleanup (bool):
                Whether to delete any zip/tar files after extracting.

        Raises:
            ValueError: if invalid keys are passed to partial_download
            IOError: if a downloaded file's checksum is different from expected

        """
        # see download_utils.downloader for basic usage - if you only need to call␣
↪downloader
        # once, you do not need this function at all.
        # only write a custom function if you need it!
```

You may find these examples useful as references:

- A simple, fully downloadable dataset
- A dataset which is partially downloadable
- A dataset with restricted access data
- A dataset which uses dataset-level metadata
- A dataset which does not use dataset-level metadata
- A dataset with a custom download function
- A dataset with a remote index

For many more examples, see the datasets folder.

3. Add tests

To finish your contribution, include tests that check the integrity of your loader. For this, follow these steps:

1. Make a toy version of the dataset in the tests folder `tests/resources/mir_datasets/my_dataset/`, so you can test against little data. For example:
   - Include all audio and annotation files for one track of the dataset
   - For each audio/annotation file, reduce the audio length to 1-2 seconds and remove all but a few of the annotations.
   - If the dataset has a metadata file, reduce the length to a few lines.
2. Test all of the dataset specific code, e.g. the public attributes of the Track class, the load functions and any other custom functions you wrote. See the tests folder for reference. If your loader has a custom download function, add tests similar to this loader.
3. Locally run `pytest -s tests/test_full_dataset.py --local --dataset my_dataset` before submitting your loader to make sure everything is working.

Note: We have written automated tests for all loader's `cite`, `download`, `validate`, `load`, `track_ids` functions, as well as some basic edge cases of the `Track` class, so you don't need to write tests for these!

Example Test File

```
import numpy as np

from mirdata import annotations
from mirdata.datasets import example
from tests.test_utils import run_track_tests


def test_track():
    default_trackid = "some_id"
    data_home = "tests/resources/mir_datasets/dataset"
    dataset = example.Dataset(data_home)
    track = dataset.track(default_trackid)

    expected_attributes = {
        "track_id": "some_id",
        "audio_path": "tests/resources/mir_datasets/example/" + "Wavfile/some_id.wav",
        "song_id": "some_id",
        "annotation_path": "tests/resources/mir_datasets/example/annotation/some_id.pv
↪",
    }

    expected_property_types = {"annotation": annotations.XData}

    assert track._track_paths == {
        "audio": ["Wavfile/some_id.wav", "278ae003cb0d323e99b9a643c0f2eeda"],
        "annotation": ["Annotation/some_id.pv", "0d93a011a9e668fd80673049089bbb14"],
    }
```

(continues on next page)

(continued from previous page)

```
    run_track_tests(track, expected_attributes, expected_property_types)

    # test audio loading functions
    audio, sr = track.audio
    assert sr == 44100
    assert audio.shape == (44100 * 2,)


def test_to_jams():

    default_trackid = "some_id"
    data_home = "tests/resources/mir_datasets/dataset"
    dataset = example.Dataset(data_home)
    track = dataset.track(default_trackid)
    jam = track.to_jams()

    annotations = jam.search(namespace="annotation")[0]["data"]
    assert [annotation.time for annotation in annotations] == [0.027, 0.232]
    assert [annotation.duration for annotation in annotations] == [
        0.20500000000000002,
        0.736,
    ]
    # ... etc


def test_load_annotation():
    # load a file which exists
    annotation_path = "tests/resources/mir_datasets/dataset/Annotation/some_id.pv"
    annotation_data = example.load_annotation(annotation_path)

    # check types
    assert type(annotation_data) == annotations.XData
    assert type(annotation_data.times) is np.ndarray
    # ... etc

    # check values
    assert np.array_equal(annotation_data.times, np.array([0.016, 0.048]))
    # ... etc


def test_metadata():
    data_home = "tests/resources/mir_datasets/dataset"
    dataset = example.Dataset(data_home)
    metadata = dataset._metadata
    assert metadata["some_id"] == "something"
```

Running your tests locally

Before creating a PR, you should run all the tests locally like this:

```
pytest tests/ --local
```

The *–local* flag skips tests that are built to run only on the remote testing environment.

To run one specific test file:

```
pytest tests/test_ikala.py
```

Finally, there is one local test you should run, which we can't easily run in our testing environment.

```
pytest -s tests/test_full_dataset.py --local --dataset dataset
```

Where `dataset` is the name of the module of the dataset you added. The `-s` tells pytest not to skip print statments, which is useful here for seeing the download progress bar when testing the download function.

This tests that your dataset downloads, validates, and loads properly for every track. This test takes a long time for some datasets, but it's important to ensure the integrity of the library.

We've added one extra convenience flag for this test, for getting the tests running when the download is very slow:

```
pytest -s tests/test_full_dataset.py --local --dataset my_dataset --skip-download
```

which will skip the downloading step. Note that this is just for convenience during debugging - the tests should eventually all pass without this flag.

Working with big datasets

In the development of large datasets, it is advisable to create an index as small as possible to optimize the implementation process of the dataset loader and pass the tests.

Working with remote indexes

For the end-user there is no difference between the remote and local indexes. However, indexes can get large when there are a lot of tracks in the dataset. In these cases, storing and accessing an index remotely can be convenient. Large indexes can be added to REMOTES, and will be downloaded with the rest of the dataset. For example:

```
"index": download_utils.RemoteFileMetadata(
    filename="acousticbrainz_genre_index.json.zip",
    url="https://zenodo.org/record/4298580/files/acousticbrainz_genre_index.json.zip?
↪download=1",
    checksum="810f1c003f53cbe58002ba96e6d4d138",
)
```

Unlike local indexes, the remote indexes will live in the `data_home` directory. When creating the `Dataset` object, specify the `custom_index_path` to where the index will be downloaded (as a relative path to `data_home`).

Reducing the testing space usage

We are trying to keep the test resources folder size as small as possible, because it can get really heavy as new loaders are added. We kindly ask the contributors to reduce the size of the testing data if possible (e.g. trimming the audio tracks, keeping just two rows for csv files).

4. Submit your loader

Before you submit your loader make sure to:

1. Add your module to `docs/source/mirdata.rst` following an alphabetical order
2. Add your module to `docs/source/table.rst` following an alphabetical order as follows:

```
* - Dataset
  - Downloadable?
  - Annotation Types
  - Tracks
  - License
```

An example of this for the `Beatport EDM key` dataset:

```
* - Beatport EDM key
  - - audio:
    - annotations:
  - - global :ref:`key`
  - 1486
  - .. image:: https://licensebuttons.net/l/by-sa/3.0/88x31.png
        :target: https://creativecommons.org/licenses/by-sa/4.0
```

(you can check that this was done correctly by clicking on the readthedocs check when you open a PR). You can find license badges images and links here.

Pull Request template

When starting your PR please use the new_loader.md template, it will simplify the reviewing process and also help you make a complete PR. You can do that by adding `&template=new_loader.md` at the end of the url when you are creating the PR :

`...mir-dataset-loaders/mirdata/compare?expand=1` will become `...mir-dataset-loaders/mirdata/compare?expand=1&template=new_loader.md`.

Docs

Staged docs for every new PR are built, and you can look at them by clicking on the "readthedocs" test in a PR. To quickly troubleshoot any issues, you can build the docs locally by nagivating to the `docs` folder, and running `make html` (note, you must have `sphinx` installed). Then open the generated `_build/source/index.html` file in your web browser to view.

Troubleshooting

If github shows a red `X` next to your latest commit, it means one of our checks is not passing. This could mean:

1. running `black` has failed – this means that your code is not formatted according to `black`'s code-style. To fix this, simply run the following from inside the top level folder of the repository:

```
black --target-version py38 mirdata/ tests/
```

2. the test coverage is too low – this means that there are too many new lines of code introduced that are not tested.
3. the docs build has failed – this means that one of the changes you made to the documentation has caused the build to fail. Check the formatting in your changes and make sure they are consistent.
4. the tests have failed – this means at least one of the tests is failing. Run the tests locally to make sure they are passing. If they are passing locally but failing in the check, open an *issue* and we can help debug.

2.9.3 Documentation

This documentation is in rst format. It is built using Sphinx and hosted on readthedocs. The API documentation is built using autodoc, which autogenerates documentation from the code's docstrings. We use the napoleon plugin for building docs in Google docstring style. See the next section for docstring conventions.

mirdata uses Google's Docstring formatting style. Here are some common examples.

Note: The small formatting details in these examples are important. Differences in new lines, indentation, and spacing make a difference in how the documentation is rendered. For example writing `Returns:` will render correctly, but `Returns` or `Returns :` will not.

Functions:

```
def add_to_list(list_of_numbers, scalar):
    """Add a scalar to every element of a list.
    You can write a continuation of the function description here on the next line.

    You can optionally write more about the function here. If you want to add an␣
↪example
    of how this function can be used, you can do it like below.

    Example:
        .. code-block:: python

        foo = add_to_list([1, 2, 3], 2)

    Args:
        list_of_numbers (list): A short description that fits on one line.
        scalar (float):
            Description of the second parameter. If there is a lot to say you can
            overflow to a second line.

    Returns:
        list: Description of the return. The type here is not in parentheses

    """
    return [x + scalar for x in list_of_numbers]
```

Functions with more than one return value:

```
def multiple_returns():
    """This function has no arguments, but more than one return value. Autodoc with␣
↪napoleon doesn't handle this well,
    and we use this formatting as a workaround.

    Returns:
        * int - the first return value
        * bool - the second return value

    """
    return 42, True
```

One-line docstrings

```
def some_function():
    """
    One line docstrings must be on their own separate line, or autodoc does not build␣
↪them properly
    """
    ...
```

Objects

```
"""Description of the class
overflowing to a second line if it's long

Some more details here

Args:
    foo (str): First argument to the __init__ method
    bar (int): Second argument to the __init__ method

Attributes:
    foobar (str): First track attribute
    barfoo (bool): Second track attribute

Cached Properties:
    foofoo (list): Cached properties are special mirdata attributes
    barbar (None): They are lazy loaded properties.
    barf (bool): Document them with this special header.

"""
```

2.9.4 Conventions

Loading from files

We use the following libraries for loading data from files:

Format	library
audio (wav, mp3, . . .)	librosa
midi	pretty_midi
json	json
csv	csv
jams	jams

Track Attributes

Custom track attributes should be global, track-level data. For some datasets, there is a separate, dataset-level metadata file with track-level metadata, e.g. as a csv. When a single file is needed for more than one track, we recommend using writing a `_metadata` cached property (which returns a dictionary, either keyed by track_id or freeform) in the Dataset class (see the dataset module example code above). When this is specified, it will populate a track's hidden `_track_metadata` field, which can be accessed from the Track class.

For example, if `_metadata` returns a dictionary of the form:

```
{
    'track1': {
        'artist': 'A',
        'genre': 'Z'
    },
    'track2': {
        'artist': 'B',
        'genre': 'Y'
    }
}
```

the `_track metadata` for `track_id=track2` will be:

```
{
    'artist': 'B',
    'genre': 'Y'
}
```

Load methods vs Track properties

Track properties and cached properties should be trivial, and directly call a `load_*` method. There should be no additional logic in a track property/cached property, and instead all logic should be done in the load method. We separate these because the track properties are only usable when data is available locally - when data is remote, the load methods are used instead.

Missing Data

If a Track has a property, for example a type of annotation, that is present for some tracks and not others, the property should be set to *None* when it isn't available.

The index should only contain key-values for files that exist.

2.9.5 Custom Decorators

cached_property

This is used primarily for Track classes.

This decorator causes an Object's function to behave like an attribute (aka, like the `@property` decorator), but caches the value in memory after it is first accessed. This is used for data which is relatively large and loaded from files.

docstring_inherit

This decorator is used for children of the Dataset class, and copies the Attributes from the parent class to the docstring of the child. This gives us clear and complete docs without a lot of copy-paste.

copy_docs

This decorator is used mainly for a dataset's `load_` functions, which are attached to a loader's Dataset class. The attached function is identical, and this decorator simply copies the docstring from another function.

coerce_to_bytes_io/coerce_to_string_io

These are two decorators used to simplify the loading of various *Track* members in addition to giving users the ability to use file streams instead of paths in case the data is in a remote location e.g. GCS. The decorators modify the function to:

- Return *None* if *None* if passed in.
- Open a file if a string path is passed in either *'w'* mode for *string_io* or *wb* for *bytes_io* and pass the file handle to the decorated function.
- Pass the file handle to the decorated function if a file-like object is passed.

This cannot be used if the function to be decorated takes multiple arguments. *coerce_to_bytes_io* should not be used if trying to load an mp3 with librosa as libsndfile does not support *mp3* yet and *audioread* expects a path.

2.10 FAQ

2.10.1 How do I add a new loader?

Take a look at our *Contributing* docs!

2.10.2 How do I get access to a dataset if the download function says it's not available?

We don't distribute data ourselves, so unfortunately it is up to you to find the data yourself. We strongly encourage you to favor datasets which are currently available.

2.10.3 Can you send me the data for a dataset which is not available?

No, we do not host or distribute datasets.

2.10.4 How do I request a new dataset?

Open an issue and tag it with the "New Loader" label.

2.10.5 What do I do if my data fails validation?

Very often, data fails vaildation because of how the files are named or how the folder is structured. If this is the case, try renaming/reorganizing your data to match what mirdata expects. If your data fails validation because of the checksums, this means that you are using data which is different from what most people are using, and you should try to get the more common dataset version, for example by using the data loader's download function.

2.10.6 How do you choose the data that is used to create the checksums?

Whenever possible, the data downloaded using `.download()` is the same data used to create the checksums. If this isn't possible, we did our best to get the data from the original source (the dataset creator) in order to create the checksum. If this is again not possible, we found as many versions of the data as we could from different users of the dataset, computed checksums on all of them and used the version which was the most common amongst them.

2.10.7 Does mirdata provide data loaders for pytorch/Tensorflow?

For now, no. Music datasets are very widely varied in their annotation types and supported tasks. To make a data loader, there would need to be "standard" ways to encode the desired inputs/outputs - unfortunately this is not universal for most datasets and usages. Still, this library provides the necessary first step for building data loaders and it is easy to build data loaders on top of this. For more information, see *Using mirdata with tensorflow*.

2.10.8 A download link is broken for a loader's `.download()` function. What do I do?

Please open an issue and tag it with the "broken link" label.

2.10.9 Why the name, mirdata?

mirdata = mir + data. MIR is an acronym for Music Information Retrieval, and the library was built for working with data.

2.10.10 If I find a mistake in an annotation, should I fix it in the loader?

No. All datasets have "mistakes", and we do not want to create another version of each dataset ourselves. The loaders should load the data as released. After that, it's up to the user what they want to do with it.

2.10.11 Does mirdata support data which lives off-disk?

Yes. While the simple useage of mirdata assumes that data lives on-disk, it can be used for off-disk data as well. See *Accessing data remotely* for details.

BIBLIOGRAPHY

[giantsteps_tempo_cit_1] Peter Knees, Ángel Faraldo, Perfecto Herrera, Richard Vogl, Sebastian Böck, Florian Hörschläger, Mickael Le Goff: "Two data sets for tempo estimation and key detection in electronic dance music annotated from user corrections", Proc. of the 16th Conference of the International Society for Music Information Retrieval (ISMIR'15), Oct. 2015, Malaga, Spain.

[giantsteps_tempo_cit_2] Hendrik Schreiber, Meinard Müller: "A Crowdsourced Experiment for Tempo Estimation of Electronic Dance Music", Proc. of the 19th Conference of the International Society for Music Information Retrieval (ISMIR'18), Sept. 2018, Paris, France.

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

R

S

T

U

V