lhotse

Release 0.1

Lhotse development team

Mar 25, 2021

CONTENTS:

CHAPTER

ONE

GETTING STARTED

Lhotse is a Python library aiming to make speech and audio data preparation flexible and accessible to a wider community. Alongside k2, it is a part of the next generation Kaldi speech processing library.

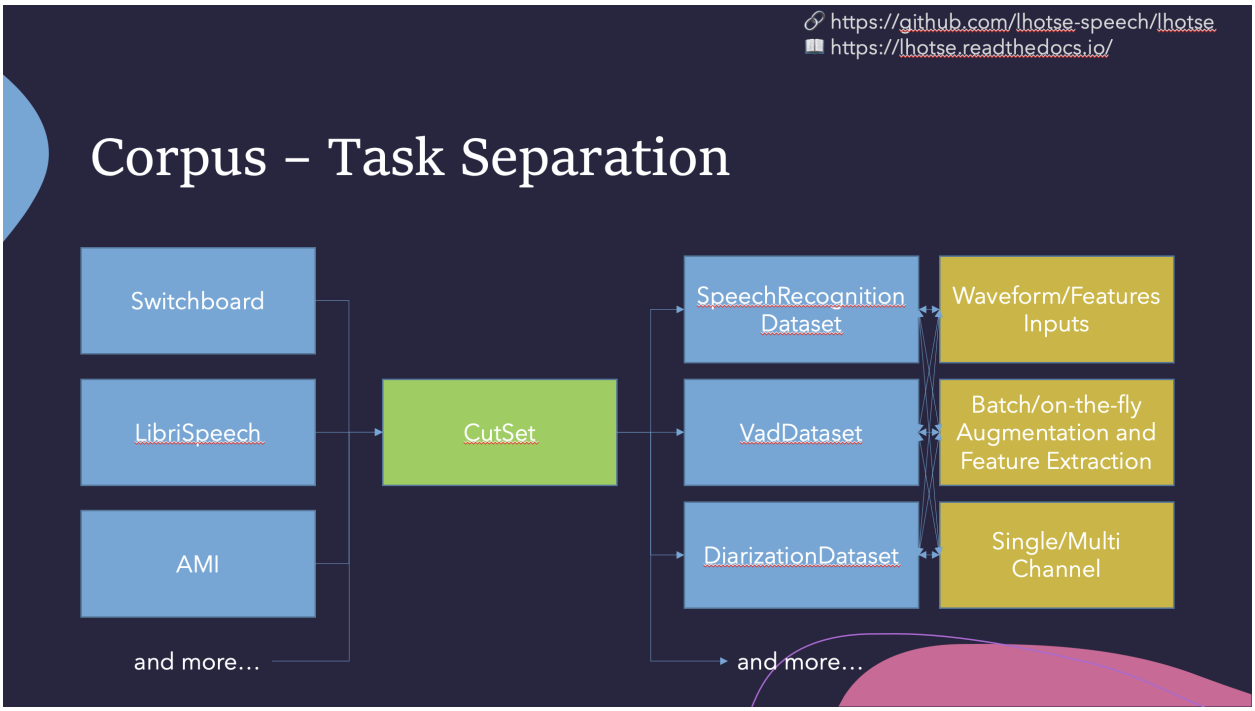
1.1 About

1.1.1 Main goals

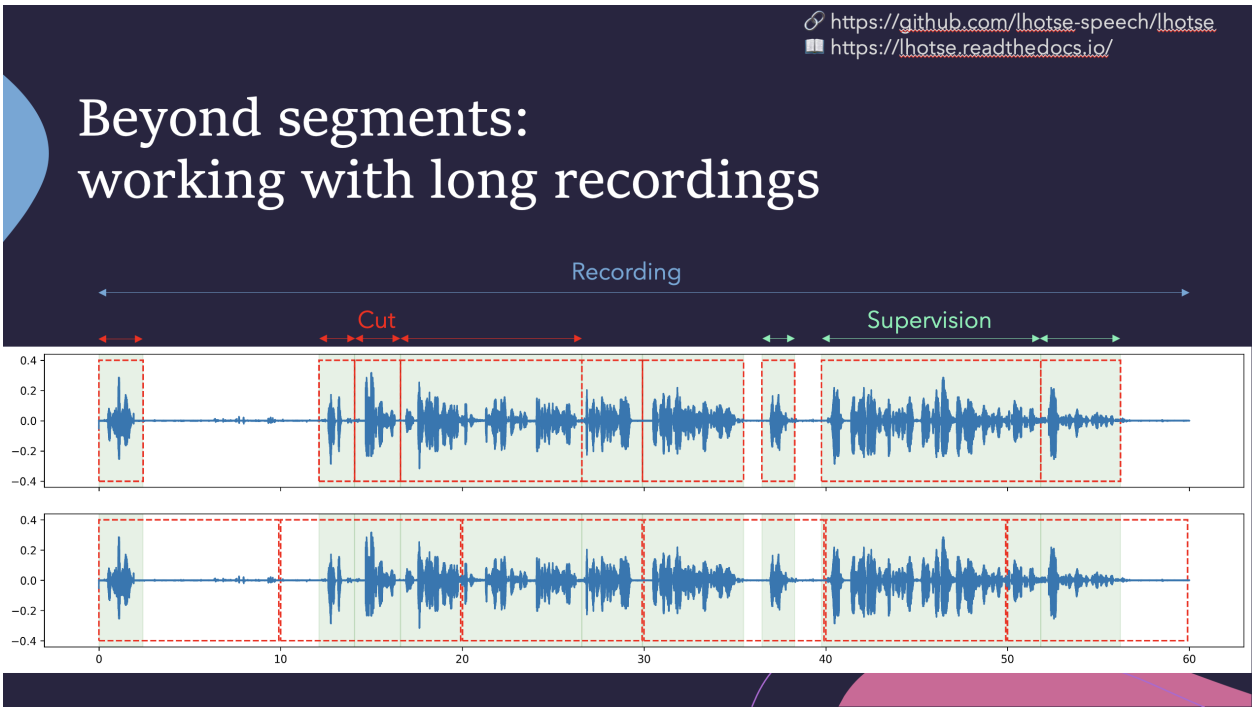
- Attract a wider community to speech processing tasks with a **Python-centric design**.
- Accommodate experienced Kaldi users with an **expressive command-line interface**.
- Provide **standard data preparation recipes** for commonly used corpora.
- Provide **PyTorch Dataset classes** for speech and audio related tasks.
- Flexible data preparation for model training with the notion of **audio cuts**.
- **Efficiency**, especially in terms of I/O bandwidth and storage capacity.

1.1.2 Main ideas

Like Kaldi, Lhotse provides standard data preparation recipes, but extends that with a seamless PyTorch integration through task-specific Dataset classes. The data and meta-data are represented in human-readable text manifests and exposed to the user through convenient Python classes.



Lhotse introduces the notion of audio cuts, designed to ease the training data construction with operations such as mixing, truncation and padding that are performed on-the-fly to minimize the amount of storage required. Data augmentation and feature extraction are supported both in pre-computed mode, with highly-compressed feature matrices stored on disk, and on-the-fly mode that computes the transformations upon request. Additionally, Lhotse introduces feature-space cut mixing to make the best of both worlds.



1.2 Installation

Lhotse supports Python version 3.6 and later.

1.2.1 Pip

Lhotse is available on PyPI:

```
pip install lhotse
```

To install the latest, unreleased version, do:

```
pip install git+https://github.com/lhotse-speech/lhotse
```

1.2.2 Development installation

For development installation, you can fork/clone the GitHub repo and install with pip:

```
git clone https://github.com/lhotse-speech/lhotse
cd lhotse
pip install -e '.[dev]'

# Running unit tests
pytest test
```

This is an editable installation (`-e` option), meaning that your changes to the source code are automatically reflected when importing lhotse (no re-install needed). The `[dev]` part means you're installing extra dependencies that are used to run tests, build documentation or launch jupyter notebooks.

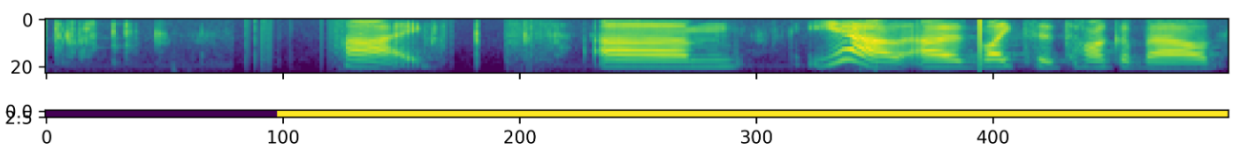
1.3 Examples

We have example recipes showing how to prepare data and load it in Python as a PyTorch `Dataset`. They are located in the `examples` directory.

A short snippet to show how Lhotse can make audio data prepartion quick and easy:

```
from torch.utils.data import DataLoader
from lhotse import CutSet, Fbank
from lhotse.dataset import VadDataset, SingleCutSampler
from lhotse.recipes import prepare_switchboard

# Prepare data manifests from a raw corpus distribution.
# The RecordingSet describes the metadata about audio recordings;
# the sampling rate, number of channels, duration, etc.
# The SupervisionSet describes metadata about supervision segments:
# the transcript, speaker, language, and so on.
swbd = prepare_switchboard('/export/corpora3/LDC/LDC97S62')

# CutSet is the workhorse of Lhotse, allowing for flexible data manipulation.
# We create 5-second cuts by traversing SWBD recordings in windows.
# No audio data is actually loaded into memory or stored to disk at this point.
cuts = CutSet.from_manifests(
    recordings=swbd['recordings'],
    supervisions=swbd['supervisions']
).cut_into_windows(duration=5)

# We compute the log-Mel filter energies and store them on disk;
# Then, we pad the cuts to 5 seconds to ensure all cuts are of equal length,
# as the last window in each recording might have a shorter duration.
# The padding will be performed once the features are loaded into memory.
cuts = cuts.compute_and_store_features(
    extractor=Fbank(),
    storage_path='feats',
    num_jobs=8
).pad(duration=5.0)

# Construct a Pytorch Dataset class for Voice Activity Detection task:
dataset = VadDataset(cuts)
sampler = SingleCutSampler(cuts)
dataloader = DataLoader(dataset, sampler=sampler, batch_size=None)
batch = next(iter(dataloader))
```

The `VadDataset` will yield a batch with pairs of feature and supervision tensors such as the following - the speech starts roughly at the first second (100 frames):



CHAPTER

TWO

REPRESENTING A CORPUS

In Lhotse, we represent the data using YAML (more readable) or JSON (faster) manifests. For most audio corpora, we will need two types of manifests to fully describe them: a recording manifest and a supervision manifest.

Caution: We show all the examples in YAML format for improved readability. However, when processing medium/large datasets, we recommend to use JSON, which is much quicker to load and save.

2.1 Recording manifest

The recording manifest describes the recordings in a given corpus. It only contains information about the recording itself - this manifest does not specify any segmentation information or supervision such as the transcript or the speaker. It means that when a recording is a 1 hour long file, it is a single item in this manifest.

When coming from Kaldi, think of it as *wav.scp* on steroids, that also contains *reco2dur*, *reco2num_samples* and some extra information.

This is a YAML manifest for a corpus with two recordings:

```
---
- id: 'recording-1'
  sampling_rate: 8000
  num_samples: 4000
  duration: 0.5
  sources:
    - type: file
      channels: [0]
      source: 'test/fixtures/mono_c0.wav'
    - type: file
      channels: [1]
      source: 'test/fixtures/mono_c1.wav'
- id: 'recording-2'
  sampling_rate: 8000
  num_samples: 8000
  duration: 1.0
  sources:
    - type: file
      channels: [0, 1]
      source: 'test/fixtures/stereo.wav'
```

Each recording is described by:

- a unique id,

- its sampling rate,
- the number of samples,
- the duration in seconds,
- a list of audio sources.

Audio source is a useful abstraction for cases when the user has an audio format not supported by the library, or wants to use shell tools such as SoX to perform some additional preprocessing. An audio source has the following properties:

- type: either *file* or *command*
- channel_ids: a list of integer identifiers for each channel in the recording
- source: in case of a *file*, it's a path; in case of a *command*, its a shell command that will be expected to write a WAVE file to stdout.

2.1.1 Python

In Python, the recording manifest is represented by classes `RecordingSet`, `Recording`, and `AudioSource`. Example usage:

```
recordings = RecordingSet.from_yaml('audio.yml')
for recording in recordings:
    # Note: all time units in Lhotse are seconds
    if recording.duration >= 7.5:
        samples = recording.load_audio(
            channels=0,
            offset=2.5,
            duration=5.0
        )
        # Further sample processing
```

2.2 Supervision manifest

The supervision manifest contains the supervision information that we have about the recordings. In particular, it involves the segmentation - there might be a single segment for a single utterance recording, and multiple segments for a recording of a converstion.

When coming from Kaldi, think of it as a *segments* file on steroids, that also contains *utt2spk*, *utt2gender*, *utt2dur*, etc.

This is a YAML supervision manifest:

```
---
- id: 'segment-1'
  recording_id: 'recording-2'
  channel: 0
  start: 0.1
  duration: 0.3
  text: 'transcript of the first segment'
  language: 'english'
  speaker: 'Norman Dyhrentfurth'

- id: 'segment-2'
  recording_id: 'recording-2'
```

(continues on next page)

(continued from previous page)

```
  start: 0.5
  duration: 0.4
```

Each segment is characterized by the following attributes:

- a unique id,
- a corresponding recording id,
- start time in seconds, relative to the beginning of the recording,
- the duration in seconds

Each segment may be assigned optional supervision information. In this example, the first segment contains the transcription text, the language of the utterance and a speaker name. The second segment contains only the minimal amount of information, which should be interpreted as: "this is some area of interest in the recording that we know nothing else about."

2.2.1 Python

In Python, the supervision manifest is represented by classes `SupervisionSet` and `SupervisionSegment`. Example usage:

```
supervisions = SupervisionSet.from_segments([
    SupervisionSegment(
        id='segment-1',
        recording_id='recording-1',
        start=0.5,
        duration=10.7,
        text='quite a long utterance'
    )
])
print(f'There is {len(supervisions)} supervision in the set.')
```

2.3 Standard data preparation recipes

We provide a number of standard data preparation recipes. By that, we mean a collection of a Python function + a CLI tool that create the manifests given a corpus directory.

Currently supported corpora:

- Aishell `lhotse.recipes.prepare_aishell()`
- AMI `lhotse.recipes.prepare_ami()`
- BABEL `lhotse.recipes.prepare_single_babel_language()`
- English Broadcast News 1997 `lhotse.recipes.prepare_broadcast_news()`
- Heroico `lhotse.recipes.prepare_heroico()`
- MiniLibriMix `lhotse.recipes.prepare_librimix()`
- MUSAN `lhotse.recipes.prepare_musan()`
- LibriSpeech (including "mini") `lhotse.recipes.prepare_librispeech()`
- LJ Speech `lhotse.recipes.prepare_ljspeech()`

- MobvoiHotWord `lhotse.recipes.prepare_mobvoihotwords()`
- National Speech Corpus (Singaporean English) `lhotse.recipes.prepare_nsc()`
- Switchboard `lhotse.recipes.prepare_switchboard()`
- TED-LIUM v3 `lhotse.recipes.prepare_tedlium()`

2.4 Adding new corpora

General pointers:

- Each corpus has a dedicated Python file in `lhotse/recipes`.
- For publicly available corpora that can be freely downloaded, we usually define a function called `download`, `download_and_untar`, etc.
- Each data preparation recipe should expose a single function called `prepare_X`, with X being the name of the corpus, that produces dicts like: `{'recordings': <RecordingSet>, 'supervisions': <SupervisionSet>}` for the data in that corpus.
- When a corpus defines standard split (e.g. train/dev/test), we return a dict with the following structure: `{'train': {'recordings': <RecordingSet>, 'supervisions': <SupervisionSet>}, 'dev': ...}`
- Some corpora (like LibriSpeech) come with pre-segmented recordings. In these cases, the `SupervisionSegment` will exactly match the `Recording` duration (and there will likely be exactly one segment corresponding to any recording).
- Corpora with longer recordings (e.g. conversational, like Switchboard) should have exactly one `Recording` object corresponding to a single conversation/session, that spans its whole duration. Each speech segment in that recording should be represented as a `SupervisionSegment` with the same `recording_id` value.
- Corpora with multiple channels for each session (e.g. AMI) should have a single `Recording` with multiple `AudioSource` objects - each corresponding to a separate channel.

CHAPTER

THREE

CUTS

3.1 Overview

Audio cuts are one of the main Lhotse features. Cut is a part of a recording, but it can be longer than a supervision segment, or even span multiple segments. The regions without a supervision are just audio that we don't assume we know anything about - there may be silence, noise, non-transcribed speech, etc. Task-specific datasets can leverage this information to generate masks for such regions.

Currently, cuts are created after the feature extraction step (we might still change that). It means that every cut also represents the extracted features for the part of recording it represents.

Cuts can be modified using three basic operations: truncation, mixing and appending. These operations are not immediately performed on the audio or features. Instead, we create new `Cut` objects, possibly of different types, that represent a cut after modification. We only modify the actual audio and feature matrices once the user calls `load_features()` or `load_audio()`.

This design allows for quick on-the-fly data augmentation. In each training epoch, we may mix the cuts with different noises, SNRs, etc. We also do not need to re-compute and store the features for different mixes, as the mixing process consists of element-wise addition of the spectral energies (possibly with additional exp and log operations for log-energies). As of now, we only support this dynamic mix on log Mel energy (_fbank_) features. We anticipate to add support for other types of features as well.

The common attributes for all cut objects are the following:

- id
- duration
- supervisions
- num_frames
- num_features
- load_features()
- truncate()
- mix()
- append()
- from_dict()

3.2 Types of cuts

There are three cut classes:

- `Cut`, also referred to as "simple cut", can be traced back to a single particular recording (and channel).
- `PaddingCut` is an "artificial" recording used for padding other Cuts through mixing to achieve uniform duration.
- `MixedCut` is a collection of `Cut` and `PaddingCut` objects, together with mix parameters: offset and desired sound-to-noise ratio (SNR) for each track. Both the offset and the SNR are relative to the first cut in the mix.

Each of these types has additional attributes that are not common - e.g., it makes sense to specify *start* for `Cut` to locate it in the source recording, but it is undefined for `MixedCut` and `PaddingCut`.

3.3 Cut manifests

All cut types can be stored in the YAML manifests. An example manifest with simple cuts might look like:

```
- duration: 10.0
  features:
    channels: 0
    duration: 16.04
    num_features: 23
    num_frames: 1604
    recording_id: recording-1
    start: 0.0
    storage_path: test/fixtures/libri/storage/dc2e0952-f2f8-423c-9b8c-f5481652ee1d.llc
    storage_type: lilcom
    type: fbank
  id: 849e13d8-61a2-4d09-a542-dac1aee1b544
  start: 0.0
  supervisions: []
  type: Cut
```

Notice that the cut type is specified in YAML. The supervisions list might be empty - some tasks do not need them, e.g. unsupervised training, source separation, or speech enhancement.

Mixed cuts look differently in the manifest:

```
- id: mixed-cut-id
  tracks:
    - cut:
        duration: 7.78
        features:
          channels: 0
          duration: 7.78
          type: fbank
          num_frames: 778
          num_features: 23
          recording_id: 7850-286674-0014
          start: 0.0
          storage_path: test/fixtures/mix_cut_test/feats/storage/9dc645db-cbe4-4529-
↪85e4-b6ed4f59c340.llc
          storage_type: lilcom
        id: 0c5fdf79-efe7-4d45-b612-3d90d9af8c4e
```

(continues on next page)

(continued from previous page)

```
        start: 0.0
        supervisions:
          - channel: 0
            duration: 7.78
            gender: f
            id: 7850-286674-0014
            language: null
            recording_id: 7850-286674-0014
            speaker: 7850-286674
            start: 0.0
            text: SURE ENOUGH THERE HE CAME THROUGH THE SHALLOW WATER HIS WET BACK␣
↪SHELL PARTLY
              OUT OF IT AND SHINING IN THE SUNLIGHT
      offset: 0.0
    - cut:
        duration: 9.705
        features:
          channels: 0
          duration: 9.705
          type: fbank
          num_frames: 970
          num_features: 23
          recording_id: 2412-153948-0014
          start: 0.0
          storage_path: test/fixtures/mix_cut_test/feats/storage/5078e7eb-57a6-4000-
↪b0f2-fa4bf9c52090.llc
          storage_type: lilcom
        id: 78bef88d-e62e-4cfa-9946-a1311442c6f7
        start: 0.0
        supervisions:
          - channel: 0
            duration: 9.705
            gender: f
            id: 2412-153948-0014
            language: null
            recording_id: 2412-153948-0014
            speaker: 2412-153948
            start: 0.0
            text: THERE WAS NO ONE IN THE WHOLE WORLD WHO HAD THE SMALLEST IDEA SAVE␣
↪THOSE
              WHO WERE THEMSELVES ON THE OTHER SIDE OF IT IF INDEED THERE WAS ANY ONE␣
↪AT ALL
              COULD I HOPE TO CROSS IT
      offset: 3.89
      snr: 20.0
  type: MixedCut
```

Mixed cuts literally consist of simple cuts, their feature descriptions, and their supervisions. These are combined together when a user queries `MixedCut` for supervisions, features, or duration. Note that the first simple cut is missing an SNR field - it is optional (i.e. *None*). That is because the semantics of 0 SNR are: re-scale one of the signals, so that the SNR between two signals is zero. We denote no re-scaling by not specifying the SNR at all.

The amount of text in these manifests can be considerable in larger datasets, but they are highly compressible. We support their automated (de-)compression with gzip - it's sufficient to add ".gz" at the end of filename when writing or reading, both in Python classes and the CLI tools.

3.4 Python

Some examples of how cuts can be manipulated to create a desired dataset for model training.

```
cuts = CutSet.from_yaml('cuts.yml')
# Reject too short segments
cuts = cuts.filter(lambda cut: cut.duration >= 3.0)
# Pad short segments with silence to 5 seconds.
cuts = cuts.pad(desired_duration=5.0)
# Truncate longer segments to 5 seconds.
cuts = cuts.truncate(max_duration=5.0, offset_type='random')
# Save cuts
cuts.to_yaml('cuts-5s.yml')
```

3.5 CLI

Analogous examples of how to perform the same operations in the terminal:

```
# Reject short segments
lhotse yaml filter duration>=3.0 cuts.yml cuts-3s.yml
# Pad short segments to 5 seconds.
lhotse cut pad --duration 5.0 cuts-3s.yml cuts-5s-pad.yml
# Truncate longer segments to 5 seconds.
lhotse cut truncate --max-duration 5.0 --offset-type random cuts-5s-pad.yml cuts-5s.
↪yml
```

CHAPTER

FOUR

FEATURE EXTRACTION

Feature extraction in Lhotse is currently based exclusively on the Torchaudio library. We support spectrograms, log-Mel energies (*fbank*) and MFCCs. *Fbank* are the default features. We also support custom defined feature extractors via a Python API (which won't be available in the CLI, unless there is a popular demand for that).

We are striving for a simple relation between the audio duration, the number of frames, and the frame shift. You only need to know two of those values to compute the third one, regardless of the frame length. This is equivalent of having Kaldi's `snip_edges` parameter set to False.

4.1 Storing features

Features in Lhotse are stored as numpy matrices with shape `(num_frames, num_features)`. By default, we use lilcom for lossy compression and reduce the size on the disk by about 3x. The lilcom compression method uses a fixed precision that doesn't depend on the magnitude of the thing being compressed, so it's better suited to log-energy features than energy features. We currently support two kinds of storage:

- HDF5 files with multiple feature matrices
- directory with feature matrix per file

We retrieve the arrays by loading the whole feature matrix from disk and selecting the relevant region (e.g. specified by a cut). Therefore it makes sense to cut the recordings first, and then extract the features for them to avoid loading unnecessary data from disk (especially for very long recordings).

There are two types of manifests:

- one describing the feature extractor;
- one describing the extracted feature matrices.

The feature extractor manifest is mapped to a Python configuration dataclass. An example for *spectrogram*:

```
dither: 0.0
energy_floor: 1e-10
frame_length: 0.025
frame_shift: 0.01
min_duration: 0.0
preemphasis_coefficient: 0.97
raw_energy: true
remove_dc_offset: true
round_to_power_of_two: true
window_type: povey
type: spectrogram
```

And the corresponding configuration class:

class lhotse.features.**SpectrogramConfig**(*dither: float = 0.0*, *window_type: str = 'povey'*, *frame_length: float = 0.025*, *frame_shift: float = 0.01*, *remove_dc_offset: bool = True*, *round_to_power_of_two: bool = True*, *energy_floor: float = 1e-10*, *min_duration: float = 0.0*, *preemphasis_coefficient: float = 0.97*, *raw_energy: bool = True*)

dither: float = 0.0

window_type: str = 'povey'

frame_length: float = 0.025

frame_shift: float = 0.01

remove_dc_offset: bool = True

round_to_power_of_two: bool = True

energy_floor: float = 1e-10

min_duration: float = 0.0

preemphasis_coefficient: float = 0.97

raw_energy: bool = True

__init__(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*)

Initialize self. See help(type(self)) for accurate signature.

The feature matrices manifest is a list of documents. These documents contain the information necessary to tie the features to a particular recording: `start`, `duration`, `channel` and `recording_id`. They currently do not have their own IDs. They also provide some useful information, such as the type of features, number of frames and feature dimension. Finally, they specify how the feature matrix is stored with `storage_type` (currently `numpy` or `lilcom`), and where to find it with the `storage_path`. In the future there might be more storage types.

```
- channels: 0
  duration: 16.04
  num_features: 23
  num_frames: 1604
  recording_id: recording-1
  start: 0.0
  storage_path: test/fixtures/libri/storage/dc2e0952-f2f8-423c-9b8c-f5481652ee1d.llc
  storage_type: lilcom
  type: fbank
```

4.2 Creating custom feature extractor

There are two components needed to implement a custom feature extractor: a configuration and the extractor itself. We expect the configuration class to be a dataclass, so that it can be automatically mapped to dict and serialized. The feature extractor should inherit from `FeatureExtractor`, and implement a small number of methods/properties. The base class takes care of initialization (you need to pass a config object), serialization to YAML, etc. A minimal, complete example of adding a new feature extractor:

```
from scipy.signal import stft

@dataclass
class ExampleFeatureExtractorConfig:
    frame_len: Seconds = 0.025
    frame_shift: Seconds = 0.01


class ExampleFeatureExtractor(FeatureExtractor):
    """
    A minimal class example, showing how to implement a custom feature extractor in␣
↪Lhotse.
    """
    name = 'example-feature-extractor'
    config_type = ExampleFeatureExtractorConfig

    def extract(self, samples: np.ndarray, sampling_rate: int) -> np.ndarray:
        f, t, Zxx = stft(
            samples,
            sampling_rate,
            nperseg=round(self.config.frame_len * sampling_rate),
            noverlap=round(self.frame_shift * sampling_rate)
        )
        # Note: returning a magnitude of the STFT might interact badly with lilcom␣
↪compression,
        # as it performs quantization of the float values and works best with log-
↪scale quantities.
        # It's advised to turn lilcom compression off, or use log-scale, in such␣
↪cases.
        return np.abs(Zxx)

    @property
    def frame_shift(self) -> Seconds:
        return self.config.frame_shift

    def feature_dim(self, sampling_rate: int) -> int:
        return (sampling_rate * self.config.frame_len) / 2 + 1
```

The overridden members include:

- `name` for easy debuggability/automatic re-creation of an extractor;
- `config_type` which specifies the complementary configuration class type;
- `extract()` where the actual computation takes place;
- `frame_shift` property, which is key to know the relationship between the duration and the number of frames.
- `feature_dim()` method, which accepts the `sampling_rate` as its argument, as some types of features (e.g. spectrogram) will depend on that.

Additionally, there are two extra methods than when overridden, allow to perform dynamic feature-space mixing (see Cuts):

```
@staticmethod
def mix(features_a: np.ndarray, features_b: np.ndarray, gain_b: float) -> np.ndarray:
    raise ValueError(f'The feature extractor\'s "mix" operation is undefined.')

@staticmethod
```

(continues on next page)

(continued from previous page)

```
def compute_energy(features: np.ndarray) -> float:
    raise ValueError(f'The feature extractor\'s "compute_energy" is undefined.')
```

They are:

- `mix()` which specifies how to mix two feature matrices to obtain a new feature matrix representing the sum of signals;
- `compute_energy()` which specifies how to obtain a total energy of the feature matrix, which is needed to mix two signals with a specified SNR. E.g. for a power spectrogram, this could be the sum of every time-frequency bin. It is expected to never return a zero.

During the feature-domain mix with a specified signal-to-noise ratio (SNR), we assume that one of the signals is a reference signal - it is used to initialize the `FeatureMixer` class. We compute the energy of both signals and scale the non-reference signal, so that its energy satisfies the requested SNR. The scaling factor (gain) is computed using the following formula:

```
        """
        assert offset >= 0.0, "Negative offset in mixing is not supported."

        reference_feats = self.tracks[0]
        num_frames_offset = compute_num_frames(duration=offset, frame_shift=self.
↪frame_shift,
                                               sampling_rate=sampling_rate)
        current_num_frames = reference_feats.shape[0]
        incoming_num_frames = feats.shape[0] + num_frames_offset
        mix_num_frames = max(current_num_frames, incoming_num_frames)
```

Note that we interpret the energy and the SNR in a power quantity context (as opposed to root-power/field quantities).

4.3 Feature normalization

We will briefly discuss how to perform mean and variance normalization (a.k.a. CMVN) in Lhotse effectively. We compute and store unnormalized features, and it is up to the user to normalize them if they want to do so. There are three common ways to perform feature normalization:

- **Global normalization**: we compute the means and variances using the whole data (`FeatureSet` or `CutSet`), and apply the same transform on every sample. The global statistics can be computed efficiently with `FeatureSet.compute_global_stats()` or `CutSet.compute_global_feature_stats()`. They use an iterative algorithm that does not require loading the whole dataset into memory.
- **Per-instance normalization**: we compute the means and variances separately for each data sample (i.e. a single feature matrix). Each feature matrix undergoes a different transform. This approach seems to be common in computer vision modelling.
- **Sliding window ("online") normalization**: we compute the means and variances using a slice of the feature matrix with a specified duration, e.g. 3 seconds (a standard value in Kaldi). This is useful when we expect the model to work on incomplete inputs, e.g. streaming speech recognition. We currently recommend using Torchaudio CMVN for that.

4.4 Storage backend details

Lhotse can be extended with additional storage backends via two abstractions: `FeaturesWriter` and `FeaturesReader`. We currently implement the following writers (and their corresponding readers):

- `lhotse.features.io.LilcomFilesWriter`
- `lhotse.features.io.NumpyFilesWriter`
- `lhotse.features.io.LilcomHdf5Writer`
- `lhotse.features.io.NumpyHdf5Writer`

The `FeaturesWriter` and `FeaturesReader` API is as follows:

class lhotse.features.io.**FeaturesWriter**

`FeaturesWriter` defines the interface of how to store numpy arrays in a particular storage backend. This backend could either be:

- separate files on a local filesystem;
- a single file with multiple arrays;
- cloud storage;
- etc.

Each class inheriting from `FeaturesWriter` must define:

- **the `write()` method, which defines the storing operation** (accepts a `key` used to place the `value` array in the storage);
- **the `storage_path()` property, which is either a common directory for the files,** the name of the file storing multiple arrays, name of the cloud bucket, etc.
- **the `name()` property that is unique to this particular storage mechanism -** it is stored in the features manifests (metadata) and used to automatically deduce the backend when loading the features.

Each `FeaturesWriter` can also be used as a context manager, as some implementations might need to free a resource after the writing is finalized. By default nothing happens in the context manager functions, and this can be modified by the inheriting subclasses.

Example:

with MyWriter('some/path') as storage: extractor.extract_from_recording_and_store(recording, storage)

The features loading must be defined separately in a class inheriting from `FeaturesReader`.

abstract property name

Return type `str`

abstract property storage_path

Return type `str`

abstract write(*key*, *value*)

Return type `str`

class lhotse.features.io.**FeaturesReader**

`FeaturesReader` defines the interface of how to load numpy arrays from a particular storage backend. This backend could either be:

- separate files on a local filesystem;

- a single file with multiple arrays;
- cloud storage;
- etc.

Each class inheriting from `FeaturesReader` must define:

- **the `read()` method, which defines the loading operation** (accepts the `key` to locate the array in the storage and return it). The read method should support selecting only a subset of the feature matrix, with the bounds expressed as arguments `left_offset_frames` and `right_offset_frames`. It's up to the Reader implementation to load only the required part or trim it to that range only after loading. It is assumed that the time dimension is always the first one.
- **the `name()` property that is unique to this particular storage mechanism -** it is stored in the features manifests (metadata) and used to automatically deduce the backend when loading the features.

The features writing must be defined separately in a class inheriting from `FeaturesWriter`.

abstract property name

Return type `str`

abstract read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type `ndarray`

4.5 Python usage

The feature manifest is represented by a `FeatureSet` object. Feature extractors have a class that represents both the extract and its configuration, named `FeatureExtractor`. We provide a utility called `FeatureSetBuilder` that can process a `RecordingSet` in parallel, store the feature matrices on disk and generate a feature manifest.

For example:

```
from lhotse import RecordingSet, Fbank, LilcomFilesWriter

# Read a RecordingSet from disk
recording_set = RecordingSet.from_yaml('audio.yml')
# Create a log Mel energy filter bank feature extractor with default settings
feature_extractor = Fbank()
# Create a feature set builder that uses this extractor and stores the results in a␣
↪directory called 'features'
with LilcomFilesWriter('features') as storage:
    builder = FeatureSetBuilder(feature_extractor=feature_extractor, storage=storage)
    # Extract the features using 8 parallel processes, compress, and store them on in
↪'features/storage/' directory.
    # Then, return the feature manifest object, which is also compressed and
    # stored in 'features/feature_manifest.json.gz'
    feature_set = builder.process_and_store_recordings(
        recordings=recording_set,
        num_jobs=8
    )
```

It is also possible to extract the features directly from *CutSet* - see below:

lhotse.cut.CutSet.**compute_and_store_features**(*self*, *extractor*, *storage_path*, *num_jobs=None*, *augment_fn=None*, *storage_type=<class 'lhotse.features.io.LilcomFilesWriter'>*, *executor=None*, *mix_eagerly=True*, *progress_bar=True*)

Extract features for all cuts, possibly in parallel, and store them using the specified storage object.

Examples:

Extract fbank features on one machine using 8 processes, store each array in a separate file with lilcom compression:

```
>>> cuts = CutSet(...)
... cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='feats',
...     num_jobs=8
... )
```

Extract fbank features on one machine using 8 processes, store arrays partitioned in 8 HDF5 files with lilcom compression:

```
>>> cuts = CutSet(...)
... cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='feats',
...     num_jobs=8,
...     storage_type=LilcomHdf5Writer
... )
```

Extract fbank features on multiple machines using a Dask cluster with 80 jobs, store arrays partitioned in 80 HDF5 files with lilcom compression:

```
>>> from distributed import Client
... cuts = CutSet(...)
... cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='feats',
...     num_jobs=80,
...     storage_type=LilcomHdf5Writer,
...     executor=Client(...)
... )
```

Parameters

- **extractor** (*FeatureExtractor*) – A `FeatureExtractor` instance (either Lhotse's built-in or a custom implementation).
- **storage_path** (`Union[Path, str]`) – The path to location where we will store the features. The exact type and layout of stored files will be dictated by the `storage_type` argument.
- **num_jobs** (`Optional[int]`) – The number of parallel processes used to extract the features. We will internally split the CutSet into this many chunks and process each chunk in parallel.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional callable used for audio augmentation. Be careful with the types of augmentations used:

if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use `CutSet.perturb_speed()` instead.

- **storage_type** (`Type`[~FW]) – a `FeaturesWriter` subclass type. It determines how the featurs are stored to disk, e.g. separate file per array, HDF5 files with multiple arrays, etc.
- **executor** (`Optional`[`Executor`]) – when provided, will be used to parallelize the feature extraction process. By default, we will instantiate a ProcessPoolExecutor. Learn more about the `Executor` API at https://lhotse.readthedocs.io/en/latest/parallelism.html
- **mix_eagerly** (`bool`) – Related to how the features are extracted for `MixedCut` instances, if any are present. When False, extract and store the features for each track separately, and mix them dynamically when loading the features. When True, mix the audio first and store the mixed features, returning a new `Cut` instance with the same ID. The returned `Cut` will not have a `Recording` attached.
- **progress_bar** (`bool`) – Should a progress bar be displayed (automatically turned off for parallel computation).

Return type *CutSet*

Returns Returns a new `CutSet` with `Features` manifests attached to the cuts.

4.6 CLI usage

An equivalent example using the terminal:

```
lhotse write-default-feature-config feat-config.yml
lhotse make-feats -j 8 --storage-type lilcom_files -f feat-config.yml audio.yml␣
↪features/
```

4.7 Kaldi compatibility caveats

We are relying on Torchaudio Kaldi compatibility module, so most of the spectrogram/fbank/mfcc parameters are the same as in Kaldi. However, we are not fully compatible - Kaldi computes energies from a signal scaled between -32,768 to 32,767, while Torchaudio scales the signal between -1.0 and 1.0. It results in Kaldi energies being significantly greater than in Lhotse. By default, we turn off dithering for deterministic feature extraction.

CHAPTER

FIVE

EXECUTING TASKS IN PARALLEL

In this section we will explain how Lhotse uses a generic interface called `Executor` to parallelize some tasks (mostly feature extraction).

There are multiple ways we can parallelize execution of a Python method:

- using multi-threading (single node, single process);
- using multi-processing (single node, multiplce processes);
- using distributed processing (multiple nodes, multiple processes).

The `Executor` API, introduced in Python's standard library in `concurrent.futures` module, allows us to use any of these methods, while writing the code independently of how it is going to be parallelized. This module defines two types of *executors*, i.e. `concurrent.futures.ProcessPoolExecutor` and `concurrent.futures.ThreadPoolExecutor`. We refer the reader to the official documentation of concurrent.futures for details. On a high level, these executors accepts tasks in the form of a Python function and an iterable of arguments, and then distribute the tasks among workers, automatically balancing the load (no manual splitting into chunks/batches is necessary).

Some methods in Lhotse (notably: `lhotse.CutSet.compute_and_store_features()`) have a parameter called `executor`, which is set to `None` by default. It means that by default, they are going to run everything in a single thread and process. The user can pass an object satisfying the `Executor` API instead, and these methods will automatically parallelize the underlying tasks.

Multi-processing: This is the recommended way to parallelize the execution for most users. An example of use to extract features on a `lhotse.CutSet`:

```
from concurrent.futures import ProcessPoolExecutor
from lhotse import CutSet, Fbank, LilcomFilesWriter
num_jobs = 8
with ProcessPoolExecutor(num_jobs) as ex:
    cuts: CutSet = cuts.compute_and_store_features(
        extractor=Fbank(),
        storage=LilcomFilesWriter('feats'),
        executor=ex
    )
```

Distributed processing: This is the recommended way for more advanced users that have the access and desire to leverage high-performance clusters (e.g. at universities). A library called Dask offers multiple powerful Python interfaces for distributed execution. One of them is called Dask.distributed. It implements the `Executor` API via class `distributed.Client`, that can be connected to an existing Dask cluster. The setup of Dask clusters is beyond the scope of this documentation, however you can find a working implementation for the CLSP Sun Grid Engine system here.

Caution: Dask is an optional dependency for Lhotse and has to be installed separately. You can install it with `pip install dask distributed`.

Multi-threading: We discourage using multi-threading with Python. Python is well known for its issues with multi-threading due to global interpreter lock (GIL), which prohibits most "typical" multi-threaded code from running in parallel. Therefore, usually concurrent tasks have to be executed in separate processes (each with its own interpreter), or use threading at the native (C or C++) level. Lhotse currently does not implement any native components, so we rely on Python-level parallelism.

If you are sure that you want to use multi-threading, you can use `concurrent.futures.ThreadPoolExecutor`. We use it sometimes in Lhotse when we expect the operations to be I/O bound rather than CPU bound (like scanning the filesystem for multiple files).

CHAPTER

SIX

AUGMENTATION

We support time-domain data augmentation via WavAugment and torchaudio libraries. They both leverage libsox to provide about 50 different audio effects like reverb, speed perturbation, pitch, etc.

Since `WavAugment` depends on libsox, it is an optional depedency for Lhotse, which can be installed using `tools/install_wavaugment.sh` (for convenience, the script will also compile libsox from source - note that the `WavAugment` authors warn their library is untested on Mac).

Torchaudio also depends on libsox, but seems to provide it when installed via anaconda. This functionality is only available with PyTorch 1.7+ and torchaudio 0.7+.

Using Lhotse's Python API, you can compose an arbitrary effect chain. On the other hand, for the CLI we provide a small number of predefined effect chains, such as `pitch` (pitch shifting), `reverb` (reverberation), and `pitch_reverb_tdrop` (pitch shift + reverberation + time dropout of a 50ms chunk).

6.1 Python usage

Warning: When using WavAugment or torchaudio data augmentation together with a multiprocessing executor (i.e. `ProcessPoolExecutor`), it is necessary to start it using the "spawn" context. Otherwise the process may hang (or terminate) on some systems due to libsox internals not handling forking well. Use: `ProcessPoolExecutor(..., mp_context=multiprocessing.get_context("spawn"))`.

Lhotse's `FeatureExtractor` and `Cut` offer convenience functions for feature extraction with data augmentation performed before that. These functions expose an optional parameter called `augment_fn` that has a signature like:

```
def augment_fn(audio: Union[np.ndarray, torch.Tensor], sampling_rate: int) -> np.
↪ndarray: ...
```

For `torchaudio` we define a `SoxEffectTransform` class:

class lhotse.augmentation.**SoxEffectTransform**(*effects*)

Class-style wrapper for torchaudio SoX effect chains. It should be initialized with a config-like list of items that define SoX effect to be applied. It supports sampling randomized values for effect parameters through the `RandomValue` wrapper.

Example:

```
>>> audio = np.random.rand(16000)
>>> augment_fn = SoxEffectTransform(effects=[
>>>     ['reverb', 50, 50, RandomValue(0, 100)],
>>>     ['speed', RandomValue(0.9, 1.1)],
```

(continues on next page)

(continued from previous page)

```
>>>     ['rate', 16000],
>>> ])
>>> augmented = augment_fn(audio, 16000)
```

See SoX manual or `torchaudio.sox_effects.effect_names()` for the list of possible effects. The parameters and the meaning of the values are explained in SoX manual/help.

__init__(*effects*)
Initialize self. See help(type(self)) for accurate signature.

sample_effects()
Resolve a list of effects, replacing random distributions with samples from them. It converts every number to string to match the expectations of torchaudio.

Return type `List[List[str]]`

We define a `WavAugmenter` class that is a thin wrapper over `WavAugment`. It can either be created with a predefined, or a user-supplied effect chain.

class lhotse.augmentation.**WavAugmenter**(*effect_chain*)
A wrapper class for WavAugment's effect chain. You should construct the `augment.EffectChain` beforehand and pass it on to this class.

This class is only available when WavAugment is installed, as it is an optional dependency for Lhotse. It can be installed using the script in "<main-repo-directory>/tools/install_wavaugment.sh"

For more details on how to augment, see https://github.com/facebookresearch/WavAugment

__init__(*effect_chain*)
Initialize self. See help(type(self)) for accurate signature.

static create_predefined(*name*, *sampling_rate*, ***kwargs*)
Create a WavAugmenter class with one of the predefined augmentation setups available in Lhotse. Some examples are: "pitch", "reverb", "pitch_reverb_tdrop".

Parameters

- **name** (`str`) – the name of the augmentation setup.
- **sampling_rate** (`int`) – expected sampling rate of the input audio.

Return type `WavAugmenter`

6.2 CLI usage

To extract features from augmented audio, you can pass an extra `--augmentation` argument to `lhotse feat extract`.

```
lhotse feat extract -a pitch ...
```

You can create a dataset with both clean and augmented features by combining different variants of extracted features, e.g.:

```
lhotse feat extract audio.yml clean_feats/
lhotse feat extract -a pitch audio.yml pitch_feats/
lhotse feat extract -a reverb audio.yml reverb_feats/
lhotse yaml combine {clean,pitch,reverb}_feats/feature_manifest.yml.gz combined_feats.
↪yml
```

CHAPTER

SEVEN

PYTORCH DATASETS

Lhotse supports PyTorch's dataset API, providing implementations for the `Dataset` and `Sampler` concepts. They can be used together with the standard `DataLoader` class for efficient mini-batch collection with multiple parallel readers and pre-fetching.

7.1 A quick re-cap of PyTorch's data API

PyTorch defines the Dataset class that is responsible for reading the data from disk/memory/Internet/database/etc., and converting it to tensors that can be used for network training or inference. These `Dataset`'s are typically „map-style" datasets which are given an index (or a list of indices) and return the corresponding data samples.

The selection of indices is performed by the `Sampler` class. `Sampler`, knowing the length (number of items) in a `Dataset`, can use various strategies to determine the order of elements to read (e.g. sequential reads, or random reads).

More details about the data pipeline API in PyTorch can be found here.

7.2 About Lhotse's Datasets and Samplers

Lhotse provides a number of utilities that make it simpler to define `Dataset`'s for speech processing tasks. *`CutSet`* is the base data structure that is used to initialize the `Dataset` class. This makes it possible to manipulate the speech data in convenient ways - pad, mix, concatenate, augment, compute features, look up the supervision information, etc.

Lhotse's `Dataset`'s will perform batching by themselves, because auto-collation in `DataLoader` is too limiting for speech data handling. These `Dataset`'s expect to be handed lists of element indices, so that they can collate the data *before* it is passed to the `DataLoader` (which must use `batch_size=None`). It allows for interesting collation methods - e.g. **padding the speech with noise recordings, or actual acoustic context**, rather than artificial zeroes; or **dynamic batch sizes**.

The items for mini-batch creation are selected by the `Sampler`. Lhotse defines `Sampler` classes that are initialized with *`CutSet`*'s, so that they can look up specific properties of an utterance to stratify the sampling. For example, *`SingleCutSampler`* has a defined `max_frames` attribute, and it will keep sampling cuts for a batch until they do not exceed the specified number of frames. Another strategy — used in *`BucketingSampler`* — will first group the cuts of similar durations into buckets, and then randomly select a bucket to draw the whole batch from.

For tasks where both input and output of the model are speech utterances, we can use the *`CutPairsSampler`*, which accepts two *`CutSet`*'s and will match the cuts in them by their IDs.

A typical Lhotse's dataset API usage might look like this:

```
from torch.utils.data import DataLoader
from lhotse.dataset import SpeechRecognitionDataset, SingleCutSampler

cuts = CutSet(...)
dset = SpeechRecognitionDataset(cuts)
sampler = SingleCutSampler(cuts, max_frames=50000)
# Dataset performs batching by itself, so we have to indicate that
# to the DataLoader with batch_size=None
dloader = DataLoader(dset, sampler=sampler, batch_size=None, num_workers=1)
for batch in dloader:
    ...  # process data
```

7.3 Dataset's list

class lhotse.dataset.diarization.**DiarizationDataset**(*cuts*, *min_speaker_dim=None*, *global_speaker_ids=False*)

A PyTorch Dataset for the speaker diarization task. Our assumptions about speaker diarization are the following:

- **we assume a single channel input (for now), which could be either a true mono signal** or a beamforming result from a microphone array.
- **we assume that the supervision used for model training is a speech activity matrix, with one** row dedicated to each speaker (either in the current cut or the whole dataset, depending on the settings). The columns correspond to feature frames. Each row is effectively a Voice Activity Detection supervision for a single speaker. This setup is somewhat inspired by the TS-VAD paper: https://arxiv.org/abs/2005.07272

Each item in this dataset is a dict of:

```
{
    'features': (B x T x F) tensor
    'speaker_activity': (B x num_speaker x T) tensor
}
```

Constructor arguments:

Parameters

- **cuts** (*CutSet*) – a `CutSet` used to create the dataset object.
- **min_speaker_dim** (`Optional[int]`) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **global_speaker_ids** (`bool`) – a bool, indicates whether the same speaker should always retain the same row index in the speaker activity matrix (useful for speaker-dependent systems)
- **root_dir** – a prefix path to be attached to the feature files paths.

__init__(*cuts*, *min_speaker_dim=None*, *global_speaker_ids=False*)

Initialize self. See help(type(self)) for accurate signature.

class lhotse.dataset.unsupervised.**UnsupervisedDataset**(*cuts*)

Dataset that contains no supervision - it only provides the features extracted from recordings. The returned features are a `torch.Tensor` of shape `(T x F)`, where T is the number of frames, and F is the feature dimension.

__init__(*cuts*)
Initialize self. See help(type(self)) for accurate signature.

class lhotse.dataset.unsupervised.**UnsupervisedWaveformDataset**(*cuts*)
A variant of UnsupervisedDataset that provides waveform samples instead of features. The output is a tensor of shape (C, T), with C being the number of channels and T the number of audio samples. In this implemenation, there will always be a single channel.

class lhotse.dataset.unsupervised.**DynamicUnsupervisedDataset**(*feature_extractor*, *cuts*, *augment_fn=None*)
An example dataset that shows how to use on-the-fly feature extraction in Lhotse. It accepts two additional inputs - a FeatureExtractor and an optional WavAugmenter for time-domain data augmentation.. The output is approximately the same as that of the `UnsupervisedDataset` - there might be slight differences for `MixedCut``s, because this dataset mixes them in the time domain, and ``UnsupervisedDataset` does that in the feature domain. Cuts that are not mixed will yield identical results in both dataset classes.

__init__(*feature_extractor*, *cuts*, *augment_fn=None*)
Initialize self. See help(type(self)) for accurate signature.

class lhotse.dataset.speech_recognition.**K2SpeechRecognitionDataset**(*cuts*, *return_cuts=False*, *cut_transforms=None*)
The PyTorch Dataset for the speech recognition task using K2 library.

This dataset expects to be queried with lists of cut IDs, for which it loads features and automatically collates/batches them.

To use it with a PyTorch DataLoader, set `batch_size=None` and provide a `SingleCutSampler` sampler.

Each item in this dataset is a dict of:

```
{
    'features': float tensor of shape (B, T, F)
    'supervisions': [
        {
            'sequence_idx': Tensor[int] of shape (S,)
            'text': List[str] of len S
            'start_frame': Tensor[int] of shape (S,)
            'num_frames': Tensor[int] of shape (S,)
            # Optionally, when return_cuts=True
            'cut': List[AnyCut] of len S
        }
    ]
}
```

Dimension symbols legend: * `B` - batch size (number of Cuts) * `S` - number of supervision segments (greater or equal to B, as each Cut may have multiple supervisions) * `T` - number of frames of the longest Cut * `F` - number of features

The 'sequence_idx' field is the index of the Cut used to create the example in the Dataset.

__init__(*cuts*, *return_cuts=False*, *cut_transforms=None*)
K2 ASR IterableDataset constructor.

Parameters

- **cuts** (*CutSet*) – the `CutSet` to sample data from.
- **return_cuts** (`bool`) – When `True`, will additionally return a "cut" field in each batch with the Cut objects used to create that batch.

- **cut_transforms** (Optional[List[Callable[[*CutSet*], *CutSet*]]]) – A list of transforms to be applied on each sampled batch (e.g. cut concatenation, noise cuts mixing, etc.).

lhotse.dataset.**speech_synthesis**
alias of *lhotse.dataset.speech_synthesis*

class lhotse.dataset.source_separation.**DynamicallyMixedSourceSeparationDataset**(*sources_set*, *mixtures_set*, *nonsources_set=Non*

A PyTorch Dataset for the source separation task. It's created from a number of CutSets:

- `sources_set`: provides the audio cuts for the sources that (the targets of source separation),
- `mixtures_set`: provides the audio cuts for the signal mix (the input of source separation),
- `nonsources_set`: *(optional)* provides the audio cuts for other signals that are in the mix, but are not the targets of source separation. Useful for adding noise.

When queried for data samples, it returns a dict of:

```
{
    'sources': (N x T x F) tensor,
    'mixture': (T x F) tensor,
    'real_mask': (N x T x F) tensor,
    'binary_mask': (T x F) tensor
}
```

This Dataset performs on-the-fly feature-domain mixing of the sources. It expects the mixtures_set to contain MixedCuts, so that it knows which Cuts should be mixed together.

__init__(*sources_set*, *mixtures_set*, *nonsources_set=None*)
Initialize self. See help(type(self)) for accurate signature.

validate()

class lhotse.dataset.source_separation.**PreMixedSourceSeparationDataset**(*sources_set*, *mixtures_set*)

A PyTorch Dataset for the source separation task. It's created from two CutSets - one provides the audio cuts for the sources, and the other one the audio cuts for the signal mix. When queried for data samples, it returns a dict of:

```
{
    'sources': (N x T x F) tensor,
    'mixture': (T x F) tensor,
    'real_mask': (N x T x F) tensor,
    'binary_mask': (T x F) tensor
}
```

It expects both CutSets to return regular Cuts, meaning that the signals were mixed in the time domain. In contrast to DynamicallyMixedSourceSeparationDataset, no on-the-fly feature-domain-mixing is performed.

__init__(*sources_set*, *mixtures_set*)
Initialize self. See help(type(self)) for accurate signature.

class lhotse.dataset.vad.**VadDataset**(*cuts*)
The PyTorch Dataset for the voice activity detection task. Each item in this dataset is a dict of:

```
{
    'features': (T x F) tensor
    'is_voice': (T x 1) tensor
    'cut': List[Cut]
}
```

__init__(*cuts*)
Initialize self. See help(type(self)) for accurate signature.

7.4 Sampler's list

class lhotse.dataset.sampling.**CutSampler**(*cut_ids*, *shuffle=False*, *world_size=None*, *rank=None*, *seed=0*)

CutSampler is responsible for collecting batches of cuts, given specified criteria. It implements correct handling of distributed sampling in DataLoader, so that the cuts are not duplicated across workers.

Sampling in a CutSampler is intended to be very quick - it only uses the metadata in `CutSet` manifest to select the cuts, and is not intended to perform any I/O.

CutSampler works similarly to PyTorch's DistributedSampler - when `shuffle=True`, you should call `sampler.set_epoch(epoch)` at each new epoch to have a different ordering of returned elements.

Example usage:

```
>>> dataset = K2SpeechRecognitionDataset(cuts)
>>> sampler = SingleCutSampler(cuts, shuffle=True)
>>> loader = DataLoader(dataset, sampler=sampler, batch_size=None)
>>> for epoch in range(start_epoch, n_epochs):
...     sampler.set_epoch(epoch)
...     train(loader)
```

Note: For implementers of new samplers: Subclasses of CutSampler are expected to implement `__next__()` to introduce specific sampling logic (e.g. based on filters such as max number of frames/tokens/etc.). CutSampler defines `__iter__()`, which optionally shuffles the cut IDs, and resets `self.current_idx` to zero (to be used and incremented inside of `__next__()`.

__init__(*cut_ids*, *shuffle=False*, *world_size=None*, *rank=None*, *seed=0*)

Parameters

- **cut_ids** (`Iterable[str]`) – An iterable of cut IDs for the full dataset. CutSampler will take care of partitioning that into distributed workers (if needed).
- **shuffle** (`bool`) – When `True`, the cuts will be shuffled at the start of iteration. Convenient when mini-batch loop is inside an outer epoch-level loop, e.g.: *for epoch in range(10): for batch in dataset: ...* as every epoch will see a different cuts order.
- **world_size** (`Optional[int]`) – Total number of distributed nodes. We will try to infer it by default.
- **rank** (`Optional[int]`) – Index of distributed node. We will try to infer it by default.
- **seed** (`int`) – Random seed used to consistently shuffle the dataset across different processes.

set_epoch(*epoch*)
Sets the epoch for this sampler. When `shuffle=True`, this ensures all replicas use a different random ordering for each epoch. Otherwise, the next iteration of this sampler will yield the same ordering.

Parameters **epoch** (`int`) – Epoch number.

Return type `None`

class lhotse.dataset.sampling.**SingleCutSampler**(*cuts*, *max_frames=26000*, *max_cuts=None*, ***kwargs*)
Samples cuts from a CutSet to satisfy the criteria of max_frames and max_cuts. It behaves like an iterable that yields lists of strings (cut IDs).

__init__(*cuts*, *max_frames=26000*, *max_cuts=None*, ***kwargs*)
SingleCutSampler's constructor.

Parameters

- **cuts** (*CutSet*) – the `CutSet` to sample data from.
- **max_frames** (`int`) – The maximum number of feature frames from `cuts` that we're going to put in a single batch. The padding introduced during collation does not contribute to that limit.
- **max_cuts** (`Optional[int]`) – The maximum number of cuts sampled to form a mini-batch. By default, this constraint is off.
- **kwargs** – Arguments to be passed into `CutSampler`.

class lhotse.dataset.sampling.**CutPairsSampler**(*source_cuts*, *target_cuts*, *max_source_frames=26000*, *max_target_frames=26000*, *max_cuts=None*, ***kwargs*)
Samples pairs of cuts from a "source" and "target" CutSet. It expects that both CutSet's strictly consist of Cuts with corresponding IDs. It will try to satisfy the criteria of max_source_frames, max_target_frames, and max_cuts. It behaves like an iterable that yields lists of strings (cut IDs).

__init__(*source_cuts*, *target_cuts*, *max_source_frames=26000*, *max_target_frames=26000*, *max_cuts=None*, ***kwargs*)
CutPairsSampler's constructor.

Parameters

- **source_cuts** (*CutSet*) – the first `CutSet` to sample data from.
- **target_cuts** (*CutSet*) – the second `CutSet` to sample data from.
- **max_source_frames** (`int`) – The maximum number of feature frames from `source_cuts` that we're going to put in a single batch. The padding introduced during collation does not contribute to that limit.
- **max_source_frames** – The maximum number of feature frames from `target_cuts` that we're going to put in a single batch. The padding introduced during collation does not contribute to that limit.
- **max_cuts** (`Optional[int]`) – The maximum number of cuts sampled to form a mini-batch. By default, this constraint is off.

class lhotse.dataset.sampling.**BucketingSampler**(**cuts*, *sampler_type=<class 'lhotse.dataset.sampling.SingleCutSampler'>*, *num_buckets=10*, ***kwargs*)
Sorts the cuts in a `CutSet` by their duration and puts them into similar duration buckets. For each bucket, it instantiates a simpler sampler instance, e.g. *SingleCutSampler*.

It behaves like an iterable that yields lists of strings (cut IDs). During iteration, it randomly selects one of the buckets to yield the batch from, until all the underlying samplers are depleted (which means it's the end of an epoch).

Examples:

Bucketing sampler with 20 buckets, sampling single cuts:

```
>>> sampler = BucketingSampler(
...     cuts,
...     # BucketingSampler specific args
...     sampler_type=SingleCutSampler, num_buckets=20,
...     # Args passed into SingleCutSampler
...     max_frames=20000
... )
```

Bucketing sampler with 20 buckets, sampling pairs of source-target cuts:

```
>>> sampler = BucketingSampler(
...     cuts, target_cuts,
...     # BucketingSampler specific args
...     sampler_type=CutPairsSampler, num_buckets=20,
...     # Args passed into CutPairsSampler
...     max_source_frames=20000, max_target_frames=15000
... )
```

__init__(**cuts*, *sampler_type=<class 'lhotse.dataset.sampling.SingleCutSampler'>*, *num_buckets=10*, ***kwargs*)

BucketingSampler's constructor.

Parameters

- **cuts** (*CutSet*) – one or more `CutSet` objects. The first one will be used to determine the buckets for all of them. Then, all of them will be used to instantiate the per-bucket samplers.
- **sampler_type** (`Type`) – a sampler type that will be created for each underlying bucket.
- **num_buckets** (`int`) – how many buckets to create.
- **kwargs** – Arguments used to create the underlying sampler for each bucket.

set_epoch(*epoch*)

Sets the epoch for this sampler. When `shuffle=True`, this ensures all replicas use a different random ordering for each epoch. Otherwise, the next iteration of this sampler will yield the same ordering.

Parameters **epoch** (`int`) – Epoch number.

Return type `None`

property is_depleted

Return type `bool`

lhotse.dataset.sampling.**partition_cut_ids**(*data_source*, *world_size=1*, *rank=0*)

Returns a list of cut IDs to be used by a single dataloading process. For multiple dataloader workers or `DistributedDataParallel` training, that list will be a subset of `sampler.full_data_source`.

Parameters

- **data_source** (`List[str]`) – a list of Cut IDs, representing the full dataset.
- **world_size** (`int`) – Total number of distributed nodes. Set only when using `DistributedDataParallel`.

- **rank** (int) – Index of distributed node. Set only when using DistributedDataParallel.

Return type List[str]

7.5 Collation utilities for building custom Datasets

lhotse.dataset.collation.**collate_features**(*cuts*)
Load features for all the cuts and return them as a batch in a torch tensor. The output shape is (batch, time, features). The cuts will be padded with silence if necessary.

Return type Tensor

lhotse.dataset.collation.**collate_audio**(*cuts*)
Load audio samples for all the cuts and return them as a batch in a torch tensor. The output shape is (batch, time). The cuts will be padded with silence if necessary.

Return type Tensor

lhotse.dataset.collation.**collate_multi_channel_features**(*cuts*)
Load features for all the cuts and return them as a batch in a torch tensor. The cuts have to be of type MixedCut and their tracks will be interpreted as individual channels. The output shape is (batch, channel, time, features). The cuts will be padded with silence if necessary.

Return type Tensor

lhotse.dataset.collation.**collate_multi_channel_audio**(*cuts*)
Load audio samples for all the cuts and return them as a batch in a torch tensor. The cuts have to be of type MixedCut and their tracks will be interpreted as individual channels. The output shape is (batch, channel, time). The cuts will be padded with silence if necessary.

Return type Tensor

lhotse.dataset.collation.**collate_vectors**(*tensors*, *padding_value=-100*, *matching_shapes=False*)
Convert an iterable of 1-D tensors (of possibly various lengths) into a single stacked tensor.

Parameters

- **tensors** (Iterable[Union[Tensor, ndarray]]) – an iterable of 1-D tensors.
- **padding_value** (Union[int, float]) – the padding value inserted to make all tensors have the same length.
- **matching_shapes** (bool) – when True, will fail when input tensors have different shapes.

Return type Tensor

Returns a tensor with shape (B, L) where B is the number of input tensors and L is the number of items in the longest tensor.

lhotse.dataset.collation.**collate_matrices**(*tensors*, *padding_value=0*, *matching_shapes=False*)
Convert an iterable of 2-D tensors (of possibly various first dimension, but consistent second dimension) into a single stacked tensor.

Parameters

- **tensors** (Iterable[Union[Tensor, ndarray]]) – an iterable of 2-D tensors.

- **padding_value** (Union[int, float]) – the padding value inserted to make all tensors have the same length.
- **matching_shapes** (bool) – when True, will fail when input tensors have different shapes.

Return type Tensor

Returns a tensor with shape (B, L, F) where B is the number of input tensors, L is the largest found shape[0], and F is equal to shape[1].

lhotse.dataset.collation.**maybe_pad**(*cuts*)

Check if all cuts' durations are equal and pad them to match the longest cut otherwise.

Return type *CutSet*

CHAPTER

EIGHT

KALDI INTEROPERABILITY

We support importing Kaldi data directories that contain at least the `wav.scp` file, required to create the `RecordingSet`. Other files, such as `segments`, `utt2spk`, etc. are used to create the `SupervisionSet`.

We currently do not support the following (but may start doing so in the future):

- Importing Kaldi's extracted features (`feats.scp` is ignored)
- Exporting Lhotse manifests as Kaldi data directories.

8.1 Python

Python methods related to Kaldi support:

`lhotse.kaldi.`**`load_kaldi_data_dir`**(*path*, *sampling_rate*)
Load a Kaldi data directory and convert it to a Lhotse RecordingSet and SupervisionSet manifests. For this to work, at least the wav.scp file must exist. SupervisionSet is created only when a segments file exists. All the other files (text, utt2spk, etc.) are optional, and some of them might not be handled yet. In particular, feats.scp files are ignored.

Return type `Tuple[`*RecordingSet*`, Optional[`*SupervisionSet*`]]`

`lhotse.kaldi.`**`export_to_kaldi`**(*recordings*, *supervisions*, *output_dir*)
Export a pair of `RecordingSet` and `SupervisionSet` to a Kaldi data directory. Currently, it only supports single-channel recordings that have a single `AudioSource`.

The `RecordingSet` and `SupervisionSet` must be compatible, i.e. it must be possible to create a `CutSet` out of them.

Parameters

- **`recordings`** (*RecordingSet*) – a `RecordingSet` manifest.
- **`supervisions`** (*SupervisionSet*) – a `SupervisionSet` manifest.
- **`output_dir`** (`Union[Path, str]`) – path where the Kaldi-style data directory will be created.

`lhotse.kaldi.`**`load_kaldi_text_mapping`**(*path*, *must_exist=False*)
Load Kaldi files such as utt2spk, spk2gender, text, etc. as a dict.

Return type `Dict[str, Optional[str]]`

`lhotse.kaldi.`**`save_kaldi_text_mapping`**(*data*, *path*)
Save flat dicts to Kaldi files such as utt2spk, spk2gender, text, etc.

8.2 CLI

Converting Kaldi data directory called `data/train`, with 16kHz sampling rate recordings, to a directory with Lhotse manifests called `train_manifests`:

```
lhotse convert-kaldi data/train 16000 train_manifests
```

CHAPTER

NINE

COMMAND-LINE INTERFACE

9.1 lhotse

The shell entry point to Lhotse, a tool and a library for audio data manipulation in high altitudes.

```
lhotse [OPTIONS] COMMAND [ARGS]...
```

Options

-s, --seed <seed>
Random seed.

9.1.1 combine

Load MANIFESTS, combine them into a single one, and write it to OUTPUT_MANIFEST.

```
lhotse combine [OPTIONS] [MANIFESTS]... OUTPUT_MANIFEST
```

Arguments

MANIFESTS
Optional argument(s)

OUTPUT_MANIFEST
Required argument

9.1.2 cut

Group of commands used to create CutSets.

```
lhotse cut [OPTIONS] COMMAND [ARGS]...
```

append

Create a new CutSet by appending the cuts in CUT_MANIFESTS. CUT_MANIFESTS are iterated position-wise (the cuts on i'th position in each manfiest are appended to each other). The cuts are appended in the order in which they appear in the input argument list. If CUT_MANIFESTS have different lengths, the script stops once the shortest CutSet is depleted.

```
lhotse cut append [OPTIONS] [CUT_MANIFESTS]... OUTPUT_CUT_MANIFEST
```

Arguments

CUT_MANIFESTS
Optional argument(s)

OUTPUT_CUT_MANIFEST
Required argument

mix-by-recording-id

Create a CutSet stored in OUTPUT_CUT_MANIFEST by matching the Cuts from CUT_MANIFESTS by their recording IDs and mixing them together.

```
lhotse cut mix-by-recording-id [OPTIONS] [CUT_MANIFESTS]...
                               OUTPUT_CUT_MANIFEST
```

Arguments

CUT_MANIFESTS
Optional argument(s)

OUTPUT_CUT_MANIFEST
Required argument

mix-sequential

Create a CutSet stored in OUTPUT_CUT_MANIFEST by iterating jointly over CUT_MANIFESTS and mixing the Cuts on the same positions. E.g. the first output cut is created from the first cuts in each input manifest. The mix is performed by summing the features from all Cuts. If the CUT_MANIFESTS have different number of Cuts, the mixing ends when the shorter manifest is depleted.

```
lhotse cut mix-sequential [OPTIONS] [CUT_MANIFESTS]... OUTPUT_CUT_MANIFEST
```

Arguments

CUT_MANIFESTS
Optional argument(s)

OUTPUT_CUT_MANIFEST
Required argument

pad

Create a new CutSet by padding the cuts in CUT_MANIFEST. The cuts will be right-padded, i.e. the padding is placed after the signal ends.

```
lhotse cut pad [OPTIONS] CUT_MANIFEST OUTPUT_CUT_MANIFEST
```

Options

-d, --duration <duration>
Desired duration of cuts after padding. Cuts longer than this won't be affected. By default, pad to the longest cut duration found in CUT_MANIFEST.

Arguments

CUT_MANIFEST
Required argument

OUTPUT_CUT_MANIFEST
Required argument

random-mixed

Create a CutSet stored in OUTPUT_CUT_MANIFEST that contains supervision regions from SUPERVISION_MANIFEST and features supplied by FEATURE_MANIFEST. It first creates a trivial CutSet, splits it into two equal, randomized parts and mixes their features. The parameters of the mix are controlled via SNR_RANGE and OFFSET_RANGE.

```
lhotse cut random-mixed [OPTIONS] SUPERVISION_MANIFEST FEATURE_MANIFEST
                        OUTPUT_CUT_MANIFEST
```

Options

-s, --snr-range <snr_range>
Range of SNR values (in dB) that will be uniformly sampled in order to mix the signals.

-o, --offset-range <offset_range>
Range of relative offset values (0 - 1), which will offset the "right" signal by this many times the duration of the "left" signal. It is uniformly sampled for each mix operation.

Arguments

SUPERVISION_MANIFEST
Required argument

FEATURE_MANIFEST
Required argument

OUTPUT_CUT_MANIFEST
Required argument

simple

Create a CutSet stored in OUTPUT_CUT_MANIFEST. Depending on the provided options, it may contain any combination of recording, feature and supervision manifests. Either RECORDING_MANIFEST or FEATURE_MANIFEST has to be provided. When SUPERVISION_MANIFEST is provided, the cuts time span will correspond to that of the supervision segments. Otherwise, that time span corresponds to the one found in features, if available, otherwise recordings.

```
lhotse cut simple [OPTIONS] OUTPUT_CUT_MANIFEST
```

Options

-r, --recording-manifest <recording_manifest>
Optional recording manifest - will be used to attach the recordings to the cuts.

-f, --feature-manifest <feature_manifest>
Optional feature manifest - will be used to attach the features to the cuts.

-s, --supervision_manifest <supervision_manifest>
Optional supervision manifest - will be used to attach the supervisions to the cuts.

Arguments

OUTPUT_CUT_MANIFEST
Required argument

truncate

Truncate the cuts in the CUT_MANIFEST and write them to OUTPUT_CUT_MANIFEST. Cuts shorter than MAX_DURATION will not be modified.

```
lhotse cut truncate [OPTIONS] CUT_MANIFEST OUTPUT_CUT_MANIFEST
```

Options

--preserve-id
Should the cuts preserve IDs (by default, they will get new, random IDs)

-d, --max-duration <max_duration>
The maximum duration in seconds of a cut in the resulting manifest. [required]

-o, --offset-type <offset_type>
Where should the truncated cut start: "start" - at the start of the original cut, "end" - MAX_DURATION before the end of the original cut, "random" - randomly choose somewhere between "start" and "end" options.

Options start|end|random

--keep-overflowing-supervisions, --discard-overflowing-supervisions
When a cut is truncated in the middle of a supervision segment, should the supervision be kept.

Arguments

CUT_MANIFEST
Required argument

OUTPUT_CUT_MANIFEST
Required argument

windowed

Create a CutSet stored in OUTPUT_CUT_MANIFEST from feature regions in FEATURE_MANIFEST. The feature matrices are traversed in windows with CUT_SHIFT increments, creating cuts of constant CUT_DURATION.

```
lhotse cut windowed [OPTIONS] FEATURE_MANIFEST OUTPUT_CUT_MANIFEST
```

Options

-d, --cut-duration <cut_duration>
How long should the cuts be in seconds.

-s, --cut-shift <cut_shift>
How much to shift the cutting window in seconds (by default the shift is equal to CUT_DURATION).

--keep-shorter-windows, --discard-shorter-windows
When true, the last window will be used to create a Cut even if its duration is shorter than CUT_DURATION.

Arguments

FEATURE_MANIFEST
Required argument

OUTPUT_CUT_MANIFEST
Required argument

9.1.3 feat

Feature extraction related commands.

```
lhotse feat [OPTIONS] COMMAND [ARGS]...
```

extract

Extract features for recordings in a given AUDIO_MANIFEST. The features are stored in OUTPUT_DIR, with one file per recording (or segment).

```
lhotse feat extract [OPTIONS] RECORDING_MANIFEST OUTPUT_DIR
```

Options

-a, --augmentation <augmentation>
Optional time-domain data augmentation effect chain to apply.

Options pitch|speed|reverb|pitch_reverb_tdrop

-f, --feature-manifest <feature_manifest>
Optional manifest specifying feature extractor configuration.

--storage-type <storage_type>
Select a storage backend for the feature matrices.

Options lilcom_files|lilcom_hdf5|numpy_files|numpy_hdf5

-t, --lilcom-tick-power <lilcom_tick_power>
Determines the compression accuracy; the input will be compressed to integer multiples of 2^tick_power

-r, --root-dir <root_dir>
Root directory - all paths in the manifest will use this as prefix.

-j, --num-jobs <num_jobs>
Number of parallel processes.

Arguments

RECORDING_MANIFEST
Required argument

OUTPUT_DIR
Required argument

write-default-config

Save a default feature extraction config to OUTPUT_CONFIG.

```
lhotse feat write-default-config [OPTIONS] OUTPUT_CONFIG
```

Options

-f, --feature-type <feature_type>
Which feature extractor type to use.

Options fbank|mfcc|spectrogram

Arguments

OUTPUT_CONFIG
Required argument

9.1.4 filter

Filter a MANIFEST according to the rule specified in PREDICATE, and save the result to OUTPUT_MANIFEST. It is intended to work generically with most manifest types - it supports RecordingSet, SupervisionSet and CutSet.

The PREDICATE specifies which attribute is used for item selection. Some examples:
lhotse filter 'duration>4.5' supervision.json output.json
lhotse filter 'num_frames<600' cuts.json output.json
lhotse filter 'start=0' cuts.json output.json
lhotse filter 'channel!=0' audio.json output.json

It currently only supports comparison of numerical manifest item attributes, such as: start, duration, end, channel, num_frames, num_features, etc.

```
lhotse filter [OPTIONS] PREDICATE MANIFEST OUTPUT_MANIFEST
```

Arguments

PREDICATE
Required argument

MANIFEST
Required argument

OUTPUT_MANIFEST
Required argument

9.1.5 kaldi

Kaldi import/export related commands.

```
lhotse kaldi [OPTIONS] COMMAND [ARGS]...
```

export

Convert a pair of `RecordingSet` and `SupervisionSet` manifests into a Kaldi-style data directory.

```
lhotse kaldi export [OPTIONS] RECORDINGS SUPERVISIONS OUTPUT_DIR
```

Arguments

RECORDINGS
Required argument

SUPERVISIONS
Required argument

OUTPUT_DIR
Required argument

import

Convert a Kaldi data dir DATA_DIR into a directory MANIFEST_DIR of lhotse manifests. Ignores feats.scp. The SAMPLING_RATE has to be explicitly specified as it is not available to read from DATA_DIR.

```
lhotse kaldi import [OPTIONS] DATA_DIR SAMPLING_RATE MANIFEST_DIR
```

Arguments

DATA_DIR
Required argument

SAMPLING_RATE
Required argument

MANIFEST_DIR
Required argument

9.1.6 obtain

Command group for download and extract data.

```
lhotse obtain [OPTIONS] COMMAND [ARGS]...
```

aishell

Aishell download.

```
lhotse obtain aishell [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

ami

AMI download.

```
lhotse obtain ami [OPTIONS] TARGET_DIR
```

Options

-mic <mic>
AMI microphone setting.

Options ihm|ihm-mix|sdm|mdm

-url <url>
AMI data downloading URL.

-force-download <force_download>
If True, download even if file is present.

Arguments

TARGET_DIR
Required argument

heroico

heroico download.

```
lhotse obtain heroico [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

librimix

Mini LibriMix download.

```
lhotse obtain librimix [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

librispeech

Mini Librispeech download.

```
lhotse obtain librispeech [OPTIONS] TARGET_DIR
```

Options

--full, --mini
Download Librispeech [default] or mini Librispeech.

Arguments

TARGET_DIR
Required argument

musan

MUSAN download.

```
lhotse obtain musan [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

tedlium

TED-LIUM v3 download (approx. 11GB).

```
lhotse obtain tedlium [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

9.1.7 prepare

Command group with data preparation recipes.

```
lhotse prepare [OPTIONS] COMMAND [ARGS]...
```

aishell

Aishell ASR data preparation.

```
lhotse prepare aishell [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

ami

AMI data preparation.

```
lhotse prepare ami [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-mic <mic>
AMI microphone setting.

Options ihm|ihm-mix|sdm|mdm

-partition <partition>
Data partition to use (see http://groups.inf.ed.ac.uk/ami/corpus/datasets.shtml).

Options scenario-only|full-corpus|full-corpus-asr

-max-pause <max_pause>
Max pause allowed between word segments to combine segments.

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

babel

This is a data preparation recipe for the IARPA BABEL corpus (see: https://www.iarpa.gov/index.php/research-programs/babel). It should support all of the languages available in BABEL. It will prepare the data from the "conversational" part of BABEL.

This script should be invoked separately for each language you want to prepare, e.g.: $ lhotse prepare babel /export/corpora5/Babel/IARPA_BABEL_BP_101 data/cantonese $ lhotse prepare babel /export/corpora5/Babel/BABEL_OP1_103 data/bengali

```
lhotse prepare babel [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

broadcast-news

English Broadcast News 1997 data preparation. It will output three manifests: for recordings, topic sections, and speech segments. It supports the following LDC distributions:

* 1997 English Broadcast News Train (HUB4)
Speech LDC98S71
Transcripts LDC98T28

This data is not available for free - your institution needs to have an LDC subscription.

```
lhotse prepare broadcast-news [OPTIONS] AUDIO_DIR TRANSCRIPT_DIR OUTPUT_DIR
```

Arguments

AUDIO_DIR
Required argument

TRANSCRIPT_DIR
Required argument

OUTPUT_DIR
Required argument

heroico

heroico Answers ASR data preparation.

```
lhotse prepare heroico [OPTIONS] SPEECH_DIR TRANSCRIPT_DIR OUTPUT_DIR
```

Arguments

SPEECH_DIR
Required argument

TRANSCRIPT_DIR
Required argument

OUTPUT_DIR
Required argument

librimix

LibrMix source separation data preparation.

```
lhotse prepare librimix [OPTIONS] LIBRIMIX_CSV OUTPUT_DIR
```

Options

--sampling-rate <sampling_rate>
Sampling rate to set in the RecordingSet manifest.

--min-segment-seconds <min_segment_seconds>
Remove segments shorter than MIN_SEGMENT_SECONDS.

--with-precomputed-mixtures, --no-precomputed-mixtures
Optionally create an RecordingSet manifest including the precomputed LibriMix mixtures.

Arguments

LIBRIMIX_CSV
Required argument

OUTPUT_DIR
Required argument

librispeech

Mini Librispeech ASR data preparation.

```
lhotse prepare librispeech [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-j, --num-jobs <num_jobs>
How many threads to use (can give good speed-ups with slow disks).

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

musan

MUSAN data preparation.

```
lhotse prepare musan [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--use-vocals, --no-vocals
Whether to include vocal music in "music" part.

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

nsc

This is a data preparation recipe for the National Corpus of Speech in Singaporean English.

```
lhotse prepare nsc [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-p, --dataset-part <dataset_part>
Which part of NSC should be prepared

Options PART3_SameCloseMic|PART3_SeparateIVR

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

switchboard

The Switchboard corpus preparation.

This is conversational telephone speech collected as 2-channel, 8kHz-sampled
data. We are using just the Switchboard-1 Phase 1 training data.
The catalog number LDC97S62 (Switchboard-1 Release 2) corresponds, we believe,
to what we have. We also use the Mississippi State transcriptions, which
we download separately from
http://www.isip.piconepress.com/projects/switchboard/releases/switchboard_word_alignments.tar.gz

This data is not available for free - your institution needs to have an LDC subscription.

```
lhotse prepare switchboard [OPTIONS] AUDIO_DIR OUTPUT_DIR
```

Options

--transcript-dir <transcript_dir>

--sentiment-dir <sentiment_dir>
Optional path to LDC2020T14 package with sentiment annotations for SWBD.

--omit-silence, --retain-silence
Should the [silence] segments be kept.

Arguments

AUDIO_DIR
Required argument

OUTPUT_DIR
Required argument

tedlium

TED-LIUM v3 recording and supervision manifest preparation.

```
lhotse prepare tedlium [OPTIONS] TEDLIUM_DIR OUTPUT_DIR
```

Arguments

TEDLIUM_DIR
Required argument

OUTPUT_DIR
Required argument

9.1.8 split

Load MANIFEST, split it into NUM_SPLITS equal parts and save as separate manifests in OUTPUT_DIR.

```
lhotse split [OPTIONS] NUM_SPLITS MANIFEST OUTPUT_DIR
```

Options

-s, --shuffle
Optionally shuffle the sequence before splitting.

Arguments

NUM_SPLITS
Required argument

MANIFEST
Required argument

OUTPUT_DIR
Required argument

9.1.9 subset

Load MANIFEST, select the FIRST or LAST number of items and store it in OUTPUT_MANIFEST.

```
lhotse subset [OPTIONS] MANIFEST OUTPUT_MANIFEST
```

Options

--first <first>

--last <last>

Arguments

MANIFEST
Required argument

OUTPUT_MANIFEST
Required argument

9.1.10 validate

Validate a Lhotse manifest file.

```
lhotse validate [OPTIONS] MANIFEST
```

Options

--read-data, --dont-read-data
Should the audio/features data be read from disk to perform additional checks (could be extremely slow for large manifests).

Arguments

MANIFEST
Required argument

CHAPTER

TEN

API REFERENCE

This page contains a comprehensive list of all classes and functions within *lhotse*.

10.1 Recording manifests

Data structures used for describing audio recordings in a dataset.

class lhotse.audio.**AudioSource**(*type: str*, *channels: List[int]*, *source: str*)

AudioSource represents audio data that can be retrieved from somewhere. Supported sources of audio are currently: - 'file' (formats supported by librosa, possibly multi-channel) - 'command' [unix pipe] (must be WAVE, possibly multi-channel)

type: str

channels: List[int]

source: str

load_audio(*offset_seconds=0.0*, *duration_seconds=None*)

Load the AudioSource (both files and commands) with librosa, accounting for many audio formats and multi-channel inputs. Returns numpy array with shapes: (n_samples) for single-channel, (n_channels, n_samples) for multi-channel.

Return type ndarray

with_path_prefix(*path*)

Return type *AudioSource*

static from_dict(*data*)

Return type *AudioSource*

__init__(*type*, *channels*, *source*)

Initialize self. See help(type(self)) for accurate signature.

lhotse.audio.**read_audio**(*path*, *offset*, *duration*)

Return type Tuple[ndarray, int]

class lhotse.audio.**Recording**(*id: str*, *sources: List[*lhotse.audio.AudioSource*]*, *sampling_rate: int*, *num_samples: int*, *duration: float*, *transforms: Optional[List[Dict]] = None*)

Recording represents an AudioSource along with some metadata.

id: str

sources: List[AudioSource]

sampling_rate: int

num_samples: int

duration: Seconds

transforms: Optional[List[Dict]] = None

static from_sphere(*sph_path*, *recording_id=None*, *relative_path_depth=None*)
Read a SPHERE file's header and create the corresponding `Recording`.

Parameters

- **sph_path** (`Union[Path, str]`) – Path to the sphere (.sph) file.
- **recording_id** (`Optional[str]`) – recording id, when not specified ream the filename's stem ("x.wav" -> "x").
- **relative_path_depth** (`Optional[int]`) – optional int specifying how many last parts of the file path should be retained in the `AudioSource`. By default writes the path as is.

Return type *Recording*

Returns a new `Recording` instance pointing to the sphere file.

static from_wav(*path*, *recording_id=None*)
Read a WAVE file's header and create the corresponding `Recording`.

Parameters

- **path** (`Union[Path, str]`) – Path to the WAVE (.wav) file.
- **recording_id** (`Optional[str]`) – recording id, when not specified ream the filename's stem ("x.wav" -> "x").

Return type *Recording*

Returns a new `Recording` instance pointing to the audio file.

static from_file(*path*, *recording_id=None*)
Read an audio file's header and create the corresponding `Recording`. Suitable to use when each physical file represents a separate recording session.

If a recording session consists of multiple files (e.g. one per channel), it is advisable to create the `Recording` object manually, with each file represented as a separate `AudioSource` object.

Parameters

- **path** (`Union[Path, str]`) – Path to an audio file supported by libsoundfile (pysoundfile).
- **recording_id** (`Optional[str]`) – recording id, when not specified ream the filename's stem ("x.wav" -> "x").

Returns a new `Recording` instance pointing to the audio file.

property num_channels

property channel_ids

load_audio(*channels=None*, *offset_seconds=0.0*, *duration_seconds=None*)

Return type `ndarray`

with_path_prefix(*path*)

Return type *Recording*

perturb_speed(*factor*, *affix_id=True*)
Return a new `Recording` that will lazily perturb the speed while loading audio. The `num_samples` and `duration` fields are updated to reflect the shrinking/extending effect of speed.

Parameters

- **factor** (`float`) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (`bool`) – When true, we will modify the `Recording.id` field by affixing it with "_sp{factor}".

Return type *Recording*

Returns a modified copy of the current `Recording`.

resample(*sampling_rate*)
Return a new `Recording` that will be lazily resampled while loading audio. :type sampling_rate: `int` :param sampling_rate: The new sampling rate. :rtype: *Recording* :return: A resampled `Recording`.

static from_dict(*data*)

Return type *Recording*

__init__(*id*, *sources*, *sampling_rate*, *num_samples*, *duration*, *transforms=None*)
Initialize self. See help(type(self)) for accurate signature.

class lhotse.audio.**RecordingSet**(**args*, ***kwds*)
RecordingSet represents a dataset of recordings. It does not contain any annotation - just the information needed to retrieve a recording (possibly multi-channel, from files or from shell commands and pipes) and some metadata for each of them.

It also supports (de)serialization to/from YAML and takes care of mapping between rich Python classes and YAML primitives during conversion.

recordings: Dict[str, Recording]

static from_recordings(*recordings*)

Return type *RecordingSet*

static from_dicts(*data*)

Return type *RecordingSet*

to_dicts()

Return type `List[dict]`

filter(*predicate*)
Return a new RecordingSet with the Recordings that satisfy the *predicate*.

Parameters **predicate** (`Callable[[Recording], bool]`) – a function that takes a recording as an argument and returns bool.

Return type *RecordingSet*

Returns a filtered RecordingSet.

split(*num_splits*, *shuffle=False*)
Split the `RecordingSet` into `num_splits` pieces of equal size.

Parameters

- **num_splits** (`int`) – Requested number of splits.
- **shuffle** (`bool`) – Optionally shuffle the recordings order first.

Return type List[*RecordingSet*]

Returns A list of RecordingSet pieces.

subset(*first=None*, *last=None*)
Return a new RecordingSet according to the selected subset criterion. Only a single argument to subset is supported at this time.

Parameters

- **first** (Optional[int]) – int, the number of first recordings to keep.
- **last** (Optional[int]) – int, the number of last recordings to keep.

Return type *RecordingSet*

Returns a new RecordingSet with the subset results.

load_audio(*recording_id*, *channels=None*, *offset_seconds=0.0*, *duration_seconds=None*)

Return type ndarray

with_path_prefix(*path*)

Return type *RecordingSet*

num_channels(*recording_id*)

Return type int

sampling_rate(*recording_id*)

Return type int

num_samples(*recording_id*)

Return type int

duration(*recording_id*)

Return type float

perturb_speed(*factor*, *affix_id=True*)
Return a new RecordingSet that will lazily perturb the speed while loading audio. The num_samples and duration fields are updated to reflect the shrinking/extending effect of speed.

Parameters

- **factor** (float) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (bool) – When true, we will modify the Recording.id field by affixing it with "_sp{factor}".

Return type *RecordingSet*

Returns a RecordingSet containing the perturbed Recording objects.

resample(*sampling_rate*)
Apply resampling to all recordings in the RecordingSet and return a new RecordingSet. :type sampling_rate: int :param sampling_rate: The new sampling rate. :rtype: *RecordingSet* :return: a new RecordingSet with lazily resampled Recording objects.

__init__(*recordings*)
Initialize self. See help(type(self)) for accurate signature.

class lhotse.audio.**AudioMixer**(*base_audio*, *sampling_rate*)

Utility class to mix multiple waveforms into a single one. It should be instantiated separately for each mixing session (i.e. each `MixedCut` will create a separate `AudioMixer` to mix its tracks). It is initialized with a numpy array of audio samples (typically float32 in [-1, 1] range) that represents the "reference" signal for the mix. Other signals can be mixed to it with different time offsets and SNRs using the `add_to_mix` method. The time offset is relative to the start of the reference signal (only positive values are supported). The SNR is relative to the energy of the signal used to initialize the `AudioMixer`.

__init__(*base_audio*, *sampling_rate*)

Parameters

- **base_audio** (`ndarray`) – A numpy array with the audio samples for the base signal (all the other signals will be mixed to it).
- **sampling_rate** (`int`) – Sampling rate of the audio.

property unmixed_audio

Return a numpy ndarray with the shape (num_tracks, num_samples), where each track is zero padded and scaled adequately to the offsets and SNR used in `add_to_mix` call.

Return type `ndarray`

property mixed_audio

Return a numpy ndarray with the shape (1, num_samples) - a mono mix of the tracks supplied with `add_to_mix` calls.

Return type `ndarray`

add_to_mix(*audio*, *snr=None*, *offset=0.0*)

Add audio (only support mono-channel) of a new track into the mix. :type audio: `ndarray` :param audio: An array of audio samples to be mixed in. :type snr: `Optional[float]` :param snr: Signal-to-noise ratio, assuming *audio* represents noise (positive SNR - lower *audio* energy, negative SNR - higher *audio* energy) :type offset: `float` :param offset: How many seconds to shift *audio* in time. For mixing, the signal will be padded before the start with low energy values. :return:

lhotse.audio.**audio_energy**(*audio*)

Return type `float`

10.2 Supervision manifests

Data structures used for describing supervisions in a dataset.

class lhotse.supervision.**SupervisionSegment**(*id: str*, *recording_id: str*, *start: float*, *duration: float*, *channel: int = 0*, *text: Union[str, NoneType] = None*, *language: Union[str, NoneType] = None*, *speaker: Union[str, NoneType] = None*, *gender: Union[str, NoneType] = None*, *custom: Union[Dict[str, Any], NoneType] = None*)

id: str

recording_id: str

start: Seconds

duration: Seconds

channel: int = 0

text: Optional[str] = None

language: Optional[str] = None

speaker: Optional[str] = None

gender: Optional[str] = None

custom: Optional[Dict[str, Any]] = None

property end

Return type float

with_offset(*offset*)
Return an identical SupervisionSegment, but with the offset added to the start field.

Return type *SupervisionSegment*

perturb_speed(*factor*, *sampling_rate*, *affix_id=True*)
Return a SupervisionSegment that has time boundaries matching the recording/cut perturbed with the same factor.

Parameters

- **factor** (float) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **sampling_rate** (int) – The sampling rate is necessary to accurately perturb the start and duration (going through the sample counts).
- **affix_id** (bool) – When true, we will modify the id and recording_id fields by affixing it with "_sp{factor}".

Return type *SupervisionSegment*

Returns a modified copy of the current Recording.

trim(*end*)
Return an identical SupervisionSegment, but ensure that self.start is not negative (in which case it's set to 0) and self.end does not exceed the end parameter.

This method is useful for ensuring that the supervision does not exceed a cut's bounds, in which case pass cut.duration as the end argument, since supervision times are relative to the cut.

Return type *SupervisionSegment*

map(*transform_fn*)
Return a copy of the current segment, transformed with transform_fn.

Parameters **transform_fn** (Callable[[*SupervisionSegment*], *SupervisionSegment*]) – a function that takes a segment as input, transforms it and returns a new segment.

Return type *SupervisionSegment*

Returns a modified SupervisionSegment.

transform_text(*transform_fn*)
Return a copy of the current segment with transformed text field. Useful for text normalization, phonetic transcription, etc.

Parameters **transform_fn** (Callable[[str], str]) – a function that accepts a string and returns a string.

Return type *SupervisionSegment*

Returns a SupervisionSegment with adjusted text.

static from_dict(*data*)

Return type *SupervisionSegment*

__init__(*id*, *recording_id*, *start*, *duration*, *channel=0*, *text=None*, *language=None*, *speaker=None*, *gender=None*, *custom=None*)
Initialize self. See help(type(self)) for accurate signature.

class lhotse.supervision.**SupervisionSet**(**args*, ***kwds*)
SupervisionSet represents a collection of segments containing some supervision information. The only required fields are the ID of the segment, ID of the corresponding recording, and the start and duration of the segment in seconds. All other fields, such as text, language or speaker, are deliberately optional to support a wide range of tasks, as well as adding more supervision types in the future, while retaining backwards compatibility.

segments: Dict[str, SupervisionSegment]

static from_segments(*segments*)

Return type *SupervisionSet*

static from_dicts(*data*)

Return type *SupervisionSet*

to_dicts()

Return type List[dict]

split(*num_splits*, *shuffle=False*)
Split the SupervisionSet into num_splits pieces of equal size.

Parameters

- **num_splits** (int) – Requested number of splits.
- **shuffle** (bool) – Optionally shuffle the supervisions order first.

Return type List[*SupervisionSet*]

Returns A list of SupervisionSet pieces.

subset(*first=None*, *last=None*)
Return a new SupervisionSet according to the selected subset criterion. Only a single argument to subset is supported at this time.

Parameters

- **first** (Optional[int]) – int, the number of first supervisions to keep.
- **last** (Optional[int]) – int, the number of last supervisions to keep.

Return type *SupervisionSet*

Returns a new SupervisionSet with the subset results.

filter(*predicate*)
Return a new SupervisionSet with the SupervisionSegments that satisfy the *predicate*.

Parameters **predicate** (Callable[[*SupervisionSegment*], bool]) – a function that takes a supervision as an argument and returns bool.

Return type *SupervisionSet*

Returns a filtered SupervisionSet.

map(*transform_fn*)
Map a `transform_fn` to the SupervisionSegments and return a new `SupervisionSet`.

Parameters **transform_fn** (`Callable[[SupervisionSegment], SupervisionSegment]`) – a function that modifies a supervision as an argument.

Return type *SupervisionSet*

Returns a new `SupervisionSet` with modified segments.

transform_text(*transform_fn*)
Return a copy of the current `SupervisionSet` with the segments having a transformed `text` field. Useful for text normalization, phonetic transcription, etc.

Parameters **transform_fn** (`Callable[[str], str]`) – a function that accepts a string and returns a string.

Return type *SupervisionSet*

Returns a `SupervisionSet` with adjusted text.

find(*recording_id*, *channel=None*, *start_after=0*, *end_before=None*, *adjust_offset=False*)
Return an iterable of segments that match the provided `recording_id`.

Parameters

- **recording_id** (`str`) – Desired recording ID.
- **channel** (`Optional[int]`) – When specified, return supervisions in that channel - otherwise, in all channels.
- **start_after** (`float`) – When specified, return segments that start after the given value.
- **end_before** (`Optional[float]`) – When specified, return segments that end before the given value.
- **adjust_offset** (`bool`) – When true, return segments as if the recordings had started at `start_after`. This is useful for creating Cuts. Fom a user perspective, when dealing with a Cut, it is no longer helpful to know when the supervisions starts in a recording - instead, it's useful to know when the supervision starts relative to the start of the Cut. In the anticipated use-case, `start_after` and `end_before` would be the beginning and end of a cut; this option converts the times to be relative to the start of the cut.

Return type `Iterable[SupervisionSegment]`

Returns An iterator over supervision segments satisfying all criteria.

__init__(*segments*, *_segments_by_recording_id=None*)
Initialize self. See help(type(self)) for accurate signature.

10.3 Feature extraction and manifests

Data structures and tools used for feature extraction and description.

10.3.1 Features API - extractor and manifests

class lhotse.features.base.**FeatureExtractor**(*config=None*)

The base class for all feature extractors in Lhotse. It is initialized with a config object, specific to a particular feature extraction method. The config is expected to be a dataclass so that it can be easily serialized.

All derived feature extractors must implement at least the following:

- a `name` class attribute (how are these features called, e.g. 'mfcc')
- a `config_type` class attribute that points to the configuration dataclass type
- the `extract` method,
- the `frame_shift` property.

Feature extractors that support feature-domain mixing should additionally specify two static methods:

- `compute_energy`, and
- `mix`.

By itself, the `FeatureExtractor` offers the following high-level methods that are not intended for overriding:

- `extract_from_samples_and_store`
- `extract_from_recording_and_store`

These methods run a larger feature extraction pipeline that involves data augmentation and disk storage.

name = None

config_type = None

__init__(*config=None*)

Initialize self. See help(type(self)) for accurate signature.

abstract extract(*samples*, *sampling_rate*)

Defines how to extract features using a numpy ndarray of audio samples and the sampling rate.

Return type `ndarray`

Returns a numpy ndarray representing the feature matrix.

abstract property frame_shift

Return type `float`

abstract feature_dim(*sampling_rate*)

Return type `int`

static mix(*features_a*, *features_b*, *energy_scaling_factor_b*)

Perform feature-domain mix of two singals, `a` and `b`, and return the mixed signal.

Parameters

- **features_a** (`ndarray`) – Left-hand side (reference) signal.
- **features_b** (`ndarray`) – Right-hand side (mixed-in) signal.
- **energy_scaling_factor_b** (`float`) – A scaling factor for `features_b` energy. It is used to achieve a specific SNR. E.g. to mix with an SNR of 10dB when both `features_a` and `features_b` energies are 100, the `features_b` signal energy needs to be scaled by 0.1. Since different features (e.g. spectrogram, fbank, MFCC) require different combination of transformations (e.g. exp, log, sqrt, pow) to allow mixing

of two signals, the exact place where to apply `energy_scaling_factor_b` to the signal is determined by the implementer.

Return type `ndarray`

Returns A mixed feature matrix.

static compute_energy(*features*)

Compute the total energy of a feature matrix. How the energy is computed depends on a particular type of features. It is expected that when implemented, `compute_energy` will never return zero.

Parameters **features** (`ndarray`) – A feature matrix.

Return type `float`

Returns A positive float value of the signal energy.

extract_from_samples_and_store(*samples*, *storage*, *sampling_rate*, *offset=0*, *channel=None*, *augment_fn=None*)

Extract the features from an array of audio samples in a full pipeline:

- optional audio augmentation;
- extract the features;
- save them to disk in a specified directory;
- return a `Features` object with a description of the extracted features.

Note, unlike in `extract_from_recording_and_store`, the returned `Features` object might not be suitable to store in a `FeatureSet`, as it does not reference any particular `Recording`. Instead, this method is useful when extracting features from cuts - especially `MixedCut` instances, which may be created from multiple recordings and channels.

Parameters

- **samples** (`ndarray`) – a numpy ndarray with the audio samples.
- **sampling_rate** (`int`) – integer sampling rate of `samples`.
- **storage** (*FeaturesWriter*) – a `FeaturesWriter` object that will handle storing the feature matrices.
- **offset** (`float`) – an offset in seconds for where to start reading the recording - when used for `Cut` feature extraction, must be equal to `Cut.start`.
- **channel** (`Optional[int]`) – an optional channel number to insert into `Features` manifest.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional `WavAugmenter` instance to modify the waveform before feature extraction.

Return type *Features*

Returns a `Features` manifest item for the extracted feature matrix (it is not written to disk).

extract_from_recording_and_store(*recording*, *storage*, *offset=0*, *duration=None*, *channels=None*, *augment_fn=None*)

Extract the features from a `Recording` in a full pipeline:

- load audio from disk;
- optionally, perform audio augmentation;
- extract the features;
- save them to disk in a specified directory;

- return a `Features` object with a description of the extracted features and the source data used.

  Parameters

  - **recording** (*Recording*) – a `Recording` that specifies what's the input audio.
  - **storage** (*FeaturesWriter*) – a `FeaturesWriter` object that will handle storing the feature matrices.
  - **offset** (`float`) – an optional offset in seconds for where to start reading the recording.
  - **duration** (`Optional[float]`) – an optional duration specifying how much audio to load from the recording.
  - **channels** (`Union[int, List[int], None]`) – an optional int or list of ints, specifying the channels; by default, all channels will be used.
  - **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional `WavAugmenter` instance to modify the waveform before feature extraction.

  Return type *Features*

  Returns a `Features` manifest item for the extracted feature matrix.

classmethod from_dict(*data*)

Return type *FeatureExtractor*

classmethod from_yaml(*path*)

Return type *FeatureExtractor*

to_yaml(*path*)

lhotse.features.base.**get_extractor_type**(*name*)
Return the feature extractor type corresponding to the given name.

Parameters **name** (`str`) – specifies which feature extractor should be used.

Return type `Type`

Returns A feature extractors type.

lhotse.features.base.**create_default_feature_extractor**(*name*)
Create a feature extractor object with a default configuration.

Parameters **name** (`str`) – specifies which feature extractor should be used.

Return type `Optional[FeatureExtractor]`

Returns A new feature extractor instance.

lhotse.features.base.**register_extractor**(*cls*)
This decorator is used to register feature extractor classes in Lhotse so they can be easily created just by knowing their name.

An example of usage:

@register_extractor class MyFeatureExtractor: ...

Parameters **cls** – A type (class) that is being registered.

Returns Registered type.

class lhotse.features.base.**TorchaudioFeatureExtractor**(*config=None*)
Common abstract base class for all torchaudio based feature extractors.

feature_fn = None

extract(*samples*, *sampling_rate*)

Defines how to extract features using a numpy ndarray of audio samples and the sampling rate.

Return type ndarray

Returns a numpy ndarray representing the feature matrix.

property frame_shift

Return type float

class lhotse.features.base.**Features**(*type: str*, *num_frames: int*, *num_features: int*, *frame_shift: float*, *sampling_rate: int*, *start: float*, *duration: float*, *storage_type: str*, *storage_path: str*, *storage_key: str*, *recording_id: Optional[str] = None*, *channels: Optional[Union[int, List[int]]] = None*)

Represents features extracted for some particular time range in a given recording and channel. It contains metadata about how it's stored: storage_type describes "how to read it", for now it supports numpy arrays serialized with np.save, as well as arrays compressed with lilcom; storage_path is the path to the file on the local filesystem.

type: str

num_frames: int

num_features: int

frame_shift: Seconds

sampling_rate: int

start: Seconds

duration: Seconds

storage_type: str

storage_path: str

storage_key: str

recording_id: Optional[str] = None

channels: Optional[Union[int, List[int]]] = None

property end

Return type float

load(*start=None*, *duration=None*)

Return type ndarray

with_path_prefix(*path*)

Return type *Features*

static from_dict(*data*)

Return type *Features*

__init__(*type*, *num_frames*, *num_features*, *frame_shift*, *sampling_rate*, *start*, *duration*, *storage_type*, *storage_path*, *storage_key*, *recording_id=None*, *channels=None*)

Initialize self. See help(type(self)) for accurate signature.

class lhotse.features.base.**FeatureSet**(**args*, ***kwds*)

Represents a feature manifest, and allows to read features for given recordings within particular channels and time ranges. It also keeps information about the feature extractor parameters used to obtain this set. When a given recording/time-range/channel is unavailable, raises a KeyError.

features: List[Features]

static from_features(*features*)

Return type *FeatureSet*

static from_dicts(*data*)

Return type *FeatureSet*

to_dicts()

Return type List[dict]

with_path_prefix(*path*)

Return type *FeatureSet*

split(*num_splits*, *shuffle=False*)

Split the FeatureSet into num_splits pieces of equal size.

Parameters

- **num_splits** (int) – Requested number of splits.
- **shuffle** (bool) – Optionally shuffle the features order first.

Return type List[*FeatureSet*]

Returns A list of FeatureSet pieces.

subset(*first=None*, *last=None*)

Return a new FeatureSet according to the selected subset criterion. Only a single argument to subset is supported at this time.

Parameters

- **first** (Optional[int]) – int, the number of first supervisions to keep.
- **last** (Optional[int]) – int, the number of last supervisions to keep.

Return type *FeatureSet*

Returns a new FeatureSet with the subset results.

find(*recording_id*, *channel_id=0*, *start=0.0*, *duration=None*, *leeway=0.05*)

Find and return a Features object that best satisfies the search criteria. Raise a KeyError when no such object is available.

Parameters

- **recording_id** (str) – str, requested recording ID.
- **channel_id** (int) – int, requested channel.
- **start** (float) – float, requested start time in seconds for the feature chunk.
- **duration** (Optional[float]) – optional float, requested duration in seconds for the feature chunk. By default, return everything from the start.
- **leeway** (float) – float, controls how strictly we have to match the requested start and duration criteria. It is necessary to keep a small positive value here (default 0.05s), as there might be differneces between the duration of recording/supervision segment, and the

duration of features. The latter one is constrained to be a multiple of frame_shift, while the former can be arbitrary.

Return type *Features*

Returns a Features object satisfying the search criteria.

load(*recording_id*, *channel_id=0*, *start=0.0*, *duration=None*)
Find a Features object that best satisfies the search criteria and load the features as a numpy ndarray. Raise a KeyError when no such object is available.

Return type ndarray

compute_global_stats(*storage_path=None*)
Compute the global means and standard deviations for each feature bin in the manifest. It follows the implementation in scikit-learn: https://github.com/scikit-learn/scikit-learn/blob/0fb307bf39bbdacd6ed713c00724f8f871d60370/sklearn/utils/extmath.py#L715 which follows the paper: "Algorithms for computing the sample variance: analysis and recommendations", by Chan, Golub, and LeVeque.

Parameters **storage_path** (Union[Path, str, None]) – an optional path to a file where the stats will be stored with pickle.

Return a dict of ``{'norm_means'``{'norm_means' np.ndarray, 'norm_stds': np.ndarray}`` with the shape of the arrays equal to the number of feature bins in this manifest.

Return type Dict[str, ndarray]

__init__(*features=<factory>*, *_features_by_recording_id=None*)
Initialize self. See help(type(self)) for accurate signature.

class lhotse.features.base.**FeatureSetBuilder**(*feature_extractor*, *storage*, *augment_fn=None*)
An extended constructor for the FeatureSet. Think of it as a class wrapper for a feature extraction script. It consumes an iterable of Recordings, extracts the features specified by the FeatureExtractor config, and saves stores them on the disk.

Eventually, we plan to extend it with the capability to extract only the features in specified regions of recordings and to perform some time-domain data augmentation.

__init__(*feature_extractor*, *storage*, *augment_fn=None*)
Initialize self. See help(type(self)) for accurate signature.

process_and_store_recordings(*recordings*, *output_manifest=None*, *num_jobs=1*)

Return type *FeatureSet*

lhotse.features.base.**store_feature_array**(*feats*, *storage*)
Store feats array on disk, using lilcom compression by default.

Parameters

- **feats** (ndarray) – a numpy ndarray containing features.
- **storage** (*FeaturesWriter*) – a FeaturesWriter object to use for array storage.

Return type str

Returns a path to the file containing the stored array.

lhotse.features.base.**compute_global_stats**(*feature_manifests*, *storage_path=None*)
Compute the global means and standard deviations for each feature bin in the manifest. It performs only a single pass over the data and iteratively updates the estimate of the means and variances.

We follow the implementation in scikit-learn: https://github.com/scikit-learn/scikit-learn/blob/0fb307bf39bbdacd6ed713c00724f8f871d60370/sklearn/utils/extmath.py#L715 which follows the paper: "Algorithms for computing the sample variance: analysis and recommendations", by Chan, Golub, and LeVeque.

Parameters

- **feature_manifests** (Iterable[*Features*]) – an iterable of Features objects.
- **storage_path** (Union[Path, str, None]) – an optional path to a file where the stats will be stored with pickle.

Return a dict of ``{'norm_means'``{'norm_means' np.ndarray, 'norm_stds': np.ndarray}`` with the shape of the arrays equal to the number of feature bins in this manifest.

Return type Dict[str, ndarray]

10.3.2 Torchaudio feature extractors

class lhotse.features.fbank.**FbankConfig**(*dither: float = 0.0, window_type: str = 'povey', frame_length: float = 0.025, frame_shift: float = 0.01, remove_dc_offset: bool = True, round_to_power_of_two: bool = True, energy_floor: float = 1e-10, min_duration: float = 0.0, preemphasis_coefficient: float = 0.97, raw_energy: bool = True, low_freq: float = 20.0, high_freq: float = - 400.0, num_mel_bins: int = 40, use_energy: bool = False, vtln_low: float = 100.0, vtln_high: float = - 500.0, vtln_warp: float = 1.0*)

dither: float = 0.0

window_type: str = 'povey'

frame_length: float = 0.025

frame_shift: float = 0.01

remove_dc_offset: bool = True

round_to_power_of_two: bool = True

energy_floor: float = 1e-10

min_duration: float = 0.0

preemphasis_coefficient: float = 0.97

raw_energy: bool = True

low_freq: float = 20.0

high_freq: float = -400.0

num_mel_bins: int = 40

use_energy: bool = False

vtln_low: float = 100.0

vtln_high: float = -500.0

vtln_warp: float = 1.0

__init__(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*, *low_freq=20.0*, *high_freq=- 400.0*, *num_mel_bins=40*, *use_energy=False*, *vtln_low=100.0*, *vtln_high=-500.0*, *vtln_warp=1.0*)
Initialize self. See help(type(self)) accurate signature.

class lhotse.features.fbank.**Fbank**(*config=None*)
Log Mel energy filter bank feature extractor based on `torchaudio.compliance.kaldi.fbank` function.

name = 'fbank'

config_type
alias of *FbankConfig*

feature_dim(*sampling_rate*)

Return type int

static mix(*features_a*, *features_b*, *energy_scaling_factor_b*)
Perform feature-domain mix of two singals, `a` and `b`, and return the mixed signal.

Parameters

- **features_a** (ndarray) – Left-hand side (reference) signal.
- **features_b** (ndarray) – Right-hand side (mixed-in) signal.
- **energy_scaling_factor_b** (float) – A scaling factor for `features_b` energy. It is used to achieve a specific SNR. E.g. to mix with an SNR of 10dB when both `features_a` and `features_b` energies are 100, the `features_b` signal energy needs to be scaled by 0.1. Since different features (e.g. spectrogram, fbank, MFCC) require different combination of transformations (e.g. exp, log, sqrt, pow) to allow mixing of two signals, the exact place where to apply `energy_scaling_factor_b` to the signal is determined by the implementer.

Return type ndarray

Returns A mixed feature matrix.

static compute_energy(*features*)
Compute the total energy of a feature matrix. How the energy is computed depends on a particular type of features. It is expected that when implemented, `compute_energy` will never return zero.

Parameters **features** (ndarray) – A feature matrix.

Return type float

Returns A positive float value of the signal energy.

class lhotse.features.mfcc.**MfccConfig**(*dither: float = 0.0*, *window_type: str = 'povey'*, *frame_length: float = 0.025*, *frame_shift: float = 0.01*, *remove_dc_offset: bool = True*, *round_to_power_of_two: bool = True*, *energy_floor: float = 1e-10*, *min_duration: float = 0.0*, *preemphasis_coefficient: float = 0.97*, *raw_energy: bool = True*, *low_freq: float = 20.0*, *high_freq: float = 0.0*, *num_mel_bins: int = 23*, *use_energy: bool = False*, *vtln_low: float = 100.0*, *vtln_high: float = - 500.0*, *vtln_warp: float = 1.0*, *cepstral_lifter: float = 22.0*, *num_ceps: int = 13*)

dither: float = 0.0

window_type: str = 'povey'

frame_length: float = 0.025

frame_shift: float = 0.01

remove_dc_offset: bool = True

round_to_power_of_two: bool = True

energy_floor: float = 1e-10

min_duration: float = 0.0

preemphasis_coefficient: float = 0.97

raw_energy: bool = True

low_freq: float = 20.0

high_freq: float = 0.0

num_mel_bins: int = 23

use_energy: bool = False

vtln_low: float = 100.0

vtln_high: float = -500.0

vtln_warp: float = 1.0

cepstral_lifter: float = 22.0

num_ceps: int = 13

__init__(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*, *low_freq=20.0*, *high_freq=0.0*, *num_mel_bins=23*, *use_energy=False*, *vtln_low=100.0*, *vtln_high=- 500.0*, *vtln_warp=1.0*, *cepstral_lifter=22.0*, *num_ceps=13*)

Initialize self. See help(type(self)) for accurate signature.

class lhotse.features.mfcc.**Mfcc**(*config=None*)

MFCC feature extractor based on `torchaudio.compliance.kaldi.mfcc` function.

name = 'mfcc'

config_type

alias of *MfccConfig*

feature_dim(*sampling_rate*)

Return type int

class lhotse.features.spectrogram.**SpectrogramConfig**(*dither: float = 0.0*, *window_type: str = 'povey'*, *frame_length: float = 0.025*, *frame_shift: float = 0.01*, *remove_dc_offset: bool = True*, *round_to_power_of_two: bool = True*, *energy_floor: float = 1e-10*, *min_duration: float = 0.0*, *preemphasis_coefficient: float = 0.97*, *raw_energy: bool = True*)

dither: float = 0.0

window_type: str = 'povey'

frame_length: float = 0.025

frame_shift: float = 0.01

remove_dc_offset: bool = True

round_to_power_of_two: bool = True

energy_floor: float = 1e-10

min_duration: float = 0.0

preemphasis_coefficient: float = 0.97

raw_energy: bool = True

__init__(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*)

Initialize self. See help(type(self)) for accurate signature.

class lhotse.features.spectrogram.**Spectrogram**(*config=None*)

Log spectrogram feature extractor based on `torchaudio.compliance.kaldi.spectrogram` function.

name = 'spectrogram'

config_type

alias of *SpectrogramConfig*

feature_dim(*sampling_rate*)

Return type int

static mix(*features_a*, *features_b*, *energy_scaling_factor_b*)

Perform feature-domain mix of two singals, `a` and `b`, and return the mixed signal.

Parameters

- **features_a** (ndarray) – Left-hand side (reference) signal.
- **features_b** (ndarray) – Right-hand side (mixed-in) signal.
- **energy_scaling_factor_b** (float) – A scaling factor for `features_b` energy. It is used to achieve a specific SNR. E.g. to mix with an SNR of 10dB when both `features_a` and `features_b` energies are 100, the `features_b` signal energy needs to be scaled by 0.1. Since different features (e.g. spectrogram, fbank, MFCC) require different combination of transformations (e.g. exp, log, sqrt, pow) to allow mixing of two signals, the exact place where to apply `energy_scaling_factor_b` to the signal is determined by the implementer.

Return type ndarray

Returns A mixed feature matrix.

static compute_energy(*features*)

Compute the total energy of a feature matrix. How the energy is computed depends on a particular type of features. It is expected that when implemented, `compute_energy` will never return zero.

Parameters **features** (ndarray) – A feature matrix.

Return type `float`

Returns A positive float value of the signal energy.

10.3.3 Feature storage

class `lhotse.features.io.`**FeaturesWriter**

`FeaturesWriter` defines the interface of how to store numpy arrays in a particular storage backend. This backend could either be:

- separate files on a local filesystem;
- a single file with multiple arrays;
- cloud storage;
- etc.

Each class inheriting from `FeaturesWriter` must define:

- **the `write()` method, which defines the storing operation** (accepts a `key` used to place the `value` array in the storage);
- **the `storage_path()` property, which is either a common directory for the files,** the name of the file storing multiple arrays, name of the cloud bucket, etc.
- **the `name()` property that is unique to this particular storage mechanism -** it is stored in the features manifests (metadata) and used to automatically deduce the backend when loading the features.

Each `FeaturesWriter` can also be used as a context manager, as some implementations might need to free a resource after the writing is finalized. By default nothing happens in the context manager functions, and this can be modified by the inheriting subclasses.

Example:

with MyWriter('some/path') as storage: extractor.extract_from_recording_and_store(recording, storage)

The features loading must be defined separately in a class inheriting from `FeaturesReader`.

abstract property name

Return type `str`

abstract property storage_path

Return type `str`

abstract write(*key*, *value*)

Return type `str`

class `lhotse.features.io.`**FeaturesReader**

`FeaturesReader` defines the interface of how to load numpy arrays from a particular storage backend. This backend could either be:

- separate files on a local filesystem;
- a single file with multiple arrays;
- cloud storage;
- etc.

Each class inheriting from `FeaturesReader` must define:

- **the `read()` method, which defines the loading operation** (accepts the `key` to locate the array in the storage and return it). The read method should support selecting only a subset of the feature matrix, with the bounds expressed as arguments `left_offset_frames` and `right_offset_frames`. It's up to the Reader implementation to load only the required part or trim it to that range only after loading. It is assumed that the time dimension is always the first one.
- **the `name()` property that is unique to this particular storage mechanism -** it is stored in the features manifests (metadata) and used to automatically deduce the backend when loading the features.

The features writing must be defined separately in a class inheriting from `FeaturesWriter`.

abstract property name

Return type `str`

abstract read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type `ndarray`

lhotse.features.io.**available_storage_backends**()

Return type `List[str]`

lhotse.features.io.**register_reader**(*cls*)

Decorator used to add a new `FeaturesReader` to Lhotse's registry.

Example:

@register_reader class MyFeatureReader(FeatureReader):

...

lhotse.features.io.**register_writer**(*cls*)

Decorator used to add a new `FeaturesWriter` to Lhotse's registry.

Example:

@register_writer class MyFeatureWriter(FeatureWriter):

...

lhotse.features.io.**get_reader**(*name*)

Find a `FeaturesReader` sub-class that corresponds to the provided `name` and return its type.

Example:

reader_type = get_reader("lilcom_files") reader = reader_type("/storage/features/")

Return type `Type[FeaturesReader]`

lhotse.features.io.**get_writer**(*name*)

Find a `FeaturesWriter` sub-class that corresponds to the provided `name` and return its type.

Example:

writer_type = get_writer("lilcom_files") writer = writer_type("/storage/features/")

Return type `Type[FeaturesWriter]`

class lhotse.features.io.**LilcomFilesReader**(*storage_path*, **args*, ***kwargs*)

Reads Lilcom-compressed files from a directory on the local filesystem. `storage_path` corresponds to the directory path; `storage_key` for each utterance is the name of the file in that directory.

name = 'lilcom_files'

__init__(*storage_path*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**LilcomFilesWriter**(*storage_path*, *tick_power=-5*, **args*, ***kwargs*)
Writes Lilcom-compressed files to a directory on the local filesystem. `storage_path` corresponds to the directory path; `storage_key` for each utterance is the name of the file in that directory.

name = 'lilcom_files'

__init__(*storage_path*, *tick_power=-5*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

property storage_path

Return type str

write(*key*, *value*)

Return type str

class lhotse.features.io.**NumpyFilesReader**(*storage_path*, **args*, ***kwargs*)
Reads non-compressed numpy arrays from files in a directory on the local filesystem. `storage_path` corresponds to the directory path; `storage_key` for each utterance is the name of the file in that directory.

name = 'numpy_files'

__init__(*storage_path*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**NumpyFilesWriter**(*storage_path*, **args*, ***kwargs*)
Writes non-compressed numpy arrays to files in a directory on the local filesystem. `storage_path` corresponds to the directory path; `storage_key` for each utterance is the name of the file in that directory.

name = 'numpy_files'

__init__(*storage_path*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

property storage_path

Return type str

write(*key*, *value*)

Return type str

lhotse.features.io.**lookup_cache_or_open**(*storage_path*)
Helper internal function used in HDF5 readers. It opens the HDF files and keeps their handles open in a global program cache to avoid excessive amount of syscalls when the *Reader class is instantiated and destroyed in a loop repeatedly (frequent use-case).

The file handles can be freed at any time by calling `close_cached_file_handles()`.

lhotse.features.io.**close_cached_file_handles**()
Closes the cached file handles in `lookup_cache_or_open` (see its docs for more details).

Return type None

class lhotse.features.io.**NumpyHdf5Reader**(*storage_path*, **args*, ***kwargs*)

Reads non-compressed numpy arrays from a HDF5 file with a "flat" layout. Each array is stored as a separate HDF Dataset because their shapes (numbers of frames) may vary. storage_path corresponds to the HDF5 file path; storage_key for each utterance is the key corresponding to the array (i.e. HDF5 "Group" name).

name = 'numpy_hdf5'

__init__(*storage_path*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**NumpyHdf5Writer**(*storage_path*, **args*, ***kwargs*)

Writes non-compressed numpy arrays to a HDF5 file with a "flat" layout. Each array is stored as a separate HDF Dataset because their shapes (numbers of frames) may vary. storage_path corresponds to the HDF5 file path; storage_key for each utterance is the key corresponding to the array (i.e. HDF5 "Group" name).

Internally, this class opens the file lazily so that this object can be passed between processes without issues. This simplifies the parallel feature extraction code.

name = 'numpy_hdf5'

__init__(*storage_path*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

property storage_path

Return type str

write(*key*, *value*)

Return type str

close()

Return type None

class lhotse.features.io.**LilcomHdf5Reader**(*storage_path*, **args*, ***kwargs*)

Reads lilcom-compressed numpy arrays from a HDF5 file with a "flat" layout. Each array is stored as a separate HDF Dataset because their shapes (numbers of frames) may vary. storage_path corresponds to the HDF5 file path; storage_key for each utterance is the key corresponding to the array (i.e. HDF5 "Group" name).

name = 'lilcom_hdf5'

__init__(*storage_path*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**LilcomHdf5Writer**(*storage_path*, *tick_power=-5*, **args*, ***kwargs*)

Writes lilcom-compressed numpy arrays to a HDF5 file with a "flat" layout. Each array is stored as a separate HDF Dataset because their shapes (numbers of frames) may vary. storage_path corresponds to the HDF5 file path; storage_key for each utterance is the key corresponding to the array (i.e. HDF5 "Group" name).

name = 'lilcom_hdf5'

__init__ (*storage_path*, *tick_power=- 5*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

property storage_path

Return type str

write (*key*, *value*)

Return type str

close ()

Return type None

10.3.4 Feature-domain mixing

class lhotse.features.mixer.**FeatureMixer** (*feature_extractor*, *base_feats*, *frame_shift*, *padding_value=- 1000.0*)
Utility class to mix multiple feature matrices into a single one. It should be instantiated separately for each mixing session (i.e. each MixedCut will create a separate FeatureMixer to mix its tracks). It is initialized with a numpy array of features (typically float32) that represents the "reference" signal for the mix. Other signals can be mixed to it with different time offsets and SNRs using the add_to_mix method. The time offset is relative to the start of the reference signal (only positive values are supported). The SNR is relative to the energy of the signal used to initialize the FeatureMixer.

It relies on the FeatureExtractor to have defined mix and compute_energy methods, so that the FeatureMixer knows how to scale and add two feature matrices together.

__init__ (*feature_extractor*, *base_feats*, *frame_shift*, *padding_value=- 1000.0*)

Parameters

- **feature_extractor** (*FeatureExtractor*) – The FeatureExtractor instance that specifies how to mix the features.
- **base_feats** (ndarray) – The features used to initialize the FeatureMixer are a point of reference in terms of energy and offset for all features mixed into them.
- **frame_shift** (float) – Required to correctly compute offset and padding during the mix.
- **padding_value** (float) – The value used to pad the shorter features during the mix. This value is adequate only for log space features. For non-log space features, e.g. energies, use either 0 or a small positive value like 1e-5.

property num_features

property unmixed_feats
Return a numpy ndarray with the shape (num_tracks, num_frames, num_features), where each track's feature matrix is padded and scaled adequately to the offsets and SNR used in add_to_mix call.

Return type ndarray

property mixed_feats
Return a numpy ndarray with the shape (num_frames, num_features) - a mono mixed feature matrix of the tracks supplied with add_to_mix calls.

Return type ndarray

add_to_mix (*feats*, *sampling_rate*, *snr=None*, *offset=0.0*)
Add feature matrix of a new track into the mix. :type feats: ndarray :param feats: A 2D feature matrix to be mixed in. :type sampling_rate: int :param sampling_rate: The sampling rate of feats :type snr:

Optional[float] :param snr: Signal-to-noise ratio, assuming feats represents noise (positive SNR - lower feats energy, negative SNR - higher feats energy) :type offset: float :param offset: How many seconds to shift feats in time. For mixing, the signal will be padded before the start with low energy values.

exception lhotse.features.mixer.**NonPositiveEnergyError**

10.4 Augmentation

10.5 Cuts

Data structures and tools used to create training/testing examples.

class lhotse.cut.**CutUtilsMixin**

A mixin class for cuts which contains all the methods that share common implementations.

Note: Ideally, this would've been an abstract base class specifying the common interface, but ABC's do not mix well with dataclasses in Python. It is possible we'll ditch the dataclass for cuts in the future and make this an ABC instead.

property trimmed_supervisions

Return the supervisions in this Cut that have modified time boundaries so as not to exceed the Cut's start or end.

Note that when cut.supervisions is called, the supervisions may have negative start values that indicate the supervision actually begins before the cut, or end values that exceed the Cut's duration (it means the supervision continued in the original recording after the Cut's ending).

Return type List[*SupervisionSegment*]

mix(*other*, *offset_other_by=0.0*, *snr=None*)

Refer to mix() documentation.

Return type *MixedCut*

append(*other*, *snr=None*)

Append the other Cut after the current Cut. Conceptually the same as mix but with an offset matching the current cuts length. Optionally scale down (positive SNR) or scale up (negative SNR) the other cut. Returns a MixedCut, which only keeps the information about the mix; actual mixing is performed during the call to load_features.

Return type *MixedCut*

compute_features(*extractor*, *augment_fn=None*)

Compute the features from this cut. This cut has to be able to load audio.

Parameters

- **extractor** (*FeatureExtractor*) – a FeatureExtractor instance used to compute the features.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – optional WavAugmenter instance for audio augmentation.

Return type ndarray

Returns a numpy ndarray with the computed features.

plot_audio()

Display a plot of the waveform. Requires matplotlib to be installed.

play_audio()
Display a Jupyter widget that allows to listen to the waveform. Works only in Jupyter notebook/lab or similar (e.g. Colab).

plot_features()
Display the feature matrix as an image. Requires matplotlib to be installed.

speakers_feature_mask(*min_speaker_dim=None*, *speaker_to_idx_map=None*)
Return a matrix of per-speaker activity in a cut. The matrix shape is (num_speakers, num_frames), and its values are 0 for nonspeech **frames** and 1 for speech **frames** for each respective speaker.

This is somewhat inspired by the TS-VAD setup: https://arxiv.org/abs/2005.07272

Parameters

- **min_speaker_dim** (Optional[int]) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **speaker_to_idx_map** (Optional[Dict[str, int]]) – optional dict mapping speaker names (strings) to their global indices (ints). Useful when you want to preserve the order of the speakers (e.g. speaker XYZ is always mapped to index 2)

Return type ndarray

speakers_audio_mask(*min_speaker_dim=None*, *speaker_to_idx_map=None*)
Return a matrix of per-speaker activity in a cut. The matrix shape is (num_speakers, num_samples), and its values are 0 for nonspeech **samples** and 1 for speech **samples** for each respective speaker.

This is somewhat inspired by the TS-VAD setup: https://arxiv.org/abs/2005.07272

Parameters

- **min_speaker_dim** (Optional[int]) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **speaker_to_idx_map** (Optional[Dict[str, int]]) – optional dict mapping speaker names (strings) to their global indices (ints). Useful when you want to preserve the order of the speakers (e.g. speaker XYZ is always mapped to index 2)

Return type ndarray

supervisions_feature_mask()
Return a 1D numpy array with value 1 for **frames** covered by at least one supervision, and 0 for **frames** not covered by any supervision.

Return type ndarray

supervisions_audio_mask()
Return a 1D numpy array with value 1 for **samples** covered by at least one supervision, and 0 for **samples** not covered by any supervision.

Return type ndarray

with_id(*id_*)
Return a copy of the Cut with a new ID.

Return type Union[*Cut*, *MixedCut*, *PaddingCut*]

class lhotse.cut.**Cut**(*id: str*, *start: float*, *duration: float*, *channel: int*, *supervisions: List[lhotse.supervision.SupervisionSegment] = <factory>*, *features: Optional[lhotse.features.base.Features] = None*, *recording: Optional[lhotse.audio.Recording] = None*)

A Cut is a single "segment" that we'll train on. It contains the features corresponding to a piece of a recording, with zero or more SupervisionSegments.

The SupervisionSegments indicate which time spans of the Cut contain some kind of supervision information: e.g. transcript, speaker, language, etc. The regions without a corresponding SupervisionSegment may contain anything - usually we assume it's either silence or some kind of noise.

Note: The SupervisionSegment time boundaries are relative to the beginning of the cut. E.g. if the underlying Recording starts at 0s (always true), the Cut starts at 100s, and the SupervisionSegment starts at 3s, it means that in the Recording the supervision actually started at 103s. In some cases, the supervision might have a negative start, or a duration exceeding the duration of the Cut; this means that the supervision in the recording extends beyond the Cut.

id: str

start: Seconds

duration: Seconds

channel: int

supervisions: List[SupervisionSegment]

features: Optional[*lhotse.features.base.Features*] = None

recording: Optional[*lhotse.audio.Recording*] = None

property recording_id

Return type str

property end

Return type float

property has_features

Return type bool

property has_recording

Return type bool

property frame_shift

Return type Optional[float]

property num_frames

Return type Optional[int]

property num_samples

Return type Optional[int]

property num_features

Return type Optional[int]

property features_type

Return type Optional[str]

property sampling_rate

Return type int

load_features()
Load the features from the underlying storage and cut them to the relevant [begin, duration] region of the current Cut.

Return type Optional[ndarray]

load_audio()
Load the audio by locating the appropriate recording in the supplied RecordingSet. The audio is trimmed to the [begin, end] range specified by the Cut.

Return type Optional[ndarray]

Returns a numpy ndarray with audio samples, with shape (1 <channel>, N <samples>)

compute_and_store_features(*extractor*, *storage*, *augment_fn=None*, **args*, ***kwargs*)
Compute the features from this cut, store them on disk, and attach a feature manifest to this cut. This cut has to be able to load audio.

Parameters

- **extractor** (*FeatureExtractor*) – a FeatureExtractor instance used to compute the features.
- **output_dir** – the directory where the computed features will be stored.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation.

Return type Union[*Cut*, *MixedCut*, *PaddingCut*]

Returns a new Cut instance with a Features manifest attached to it.

truncate(***, *offset=0.0*, *duration=None*, *keep_excessive_supervisions=True*, *preserve_id=False*, *_supervisions_index=None*)
Returns a new Cut that is a sub-region of the current Cut.

Note that no operation is done on the actual features - it's only during the call to load_features() when the actual changes happen (a subset of features is loaded).

Parameters

- **offset** (float) – float (seconds), controls the start of the new cut relative to the current Cut's start. E.g., if the current Cut starts at 10.0, and offset is 2.0, the new start is 12.0.
- **duration** (Optional[float]) – optional float (seconds), controls the duration of the resulting Cut. By default, the duration is (end of the cut before truncation) - (offset).
- **keep_excessive_supervisions** (bool) – bool. Since trimming may happen inside a SupervisionSegment, the caller has an option to either keep or discard such supervisions.
- **preserve_id** (bool) – bool. Should the truncated cut keep the same ID or get a new, random one.
- **_supervisions_index** (Optional[Dict[str, IntervalTree]]) – when passed, allows to speed up processing of Cuts with a very large number of supervisions. Intended as an internal parameter.

Return type *Cut*

Returns a new Cut instance. If the current Cut is shorter than the duration, return None.

pad(*duration*, *pad_feat_value=- 23.025850929940457*)
Return a new MixedCut, padded to duration seconds with zeros in the recording, and low-energy values in each feature bin.

Parameters

- **duration** (float) – The cut's minimal duration after padding.
- **pad_feat_value** (float) – A float value that's used for padding the features. By default we assume a log-energy floor of approx. -23 (1e-10 after exp).

Return type Union[*Cut*, *MixedCut*, *PaddingCut*]

Returns a padded MixedCut if duration is greater than this cut's duration, otherwise self.

perturb_speed(*factor*, *affix_id=True*)

Return a new Cut that will lazily perturb the speed while loading audio. The num_samples, start and duration fields are updated to reflect the shrinking/extending effect of speed. We are also updating the time markers of the underlying Recording and the supervisions.

Parameters

- **factor** (float) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (bool) – When true, we will modify the Cut.id field by affixing it with "_sp{factor}".

Return type *Cut*

Returns a modified copy of the current Cut.

map_supervisions(*transform_fn*)

Modify the SupervisionSegments by *transform_fn* of this Cut.

Parameters **transform_fn** (Callable[[*SupervisionSegment*], *SupervisionSegment*]) – a function that modifies a supervision as an argument.

Return type Union[*Cut*, *MixedCut*, *PaddingCut*]

Returns a modified Cut.

filter_supervisions(*predicate*)

Modify cut to store only supervisions accepted by *predicate*

Example:

```
>>> cut = cut.filter_supervisions(lambda s: s.id in supervision_ids)
>>> cut = cut.filter_supervisions(lambda s: s.duration < 5.0)
>>> cut = cut.filter_supervisions(lambda s: s.text is not None)
```

Parameters **predicate** (Callable[[*SupervisionSegment*], bool]) – A callable that accepts *SupervisionSegment* and returns bool

Return type Union[*Cut*, *MixedCut*, *PaddingCut*]

Returns a modified Cut

static from_dict(*data*)

Return type *Cut*

with_features_path_prefix(*path*)

Return type *Cut*

with_recording_path_prefix(*path*)

Return type *Cut*

__init__(*id*, *start*, *duration*, *channel*, *supervisions=<factory>*, *features=None*, *recording=None*)
Initialize self. See help(type(self)) for accurate signature.

class lhotse.cut.**PaddingCut**(*id: str*, *duration: float*, *sampling_rate: int*, *feat_value: float*, *num_frames: Optional[int] = None*, *num_features: Optional[int] = None*, *frame_shift: Optional[float] = None*, *num_samples: Optional[int] = None*)

Represents a cut filled with zeroes in the time domain, or some specified value in the frequency domain. It's used to make training samples evenly sized (same duration/number of frames).

id: str

duration: Seconds

sampling_rate: int

feat_value: float

num_frames: Optional[int] = None

num_features: Optional[int] = None

frame_shift: Optional[float] = None

num_samples: Optional[int] = None

property start

Return type float

property end

Return type float

property supervisions

property has_features

Return type bool

property has_recording

Return type bool

load_features(**args*, ***kwargs*)

Return type Optional[ndarray]

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

Return type Optional[ndarray]

truncate(***, *offset=0.0*, *duration=None*, *keep_excessive_supervisions=True*, *preserve_id=False*, ***kwargs*)

Return type *PaddingCut*

pad(*duration*)
Create a new PaddingCut with duration when its longer than this Cuts duration. Helper function used in batch cut padding.

Parameters **duration** (float) – The cuts minimal duration after padding.

Return type *PaddingCut*

Returns self or a new PaddingCut, depending on duration.

perturb_speed(*factor*, *affix_id=True*)
Return a new `PaddingCut` that will "mimic" the effect of speed perturbation on `duration` and `num_samples`.

Parameters

- **factor** (`float`) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (`bool`) – When true, we will modify the `PaddingCut.id` field by affixing it with "_sp{factor}".

Return type *PaddingCut*

Returns a modified copy of the current `PaddingCut`.

compute_and_store_features(*extractor*, **args*, ***kwargs*)
Returns a new PaddingCut with updates information about the feature dimension and number of feature frames, depending on the `extractor` properties.

Return type `Union[`*Cut*, *MixedCut*, *PaddingCut*`]`

map_supervisions(*transform_fn*)
Just for consistency with *Cut* and *MixedCut*.

Parameters **transform_fn** (`Callable[[Any], Any]`) – a dummy function that would be never called actually.

Return type `Union[`*Cut*, *MixedCut*, *PaddingCut*`]`

Returns the PaddingCut itself.

filter_supervisions(*predicate*)
Just for consistency with *Cut* and *MixedCut*.

Parameters **predicate** (`Callable[[`*SupervisionSegment*`], bool]`) – A callable that accepts *SupervisionSegment* and returns bool

Return type `Union[`*Cut*, *MixedCut*, *PaddingCut*`]`

Returns a modified Cut

static from_dict(*data*)

Return type *PaddingCut*

with_features_path_prefix(*path*)

Return type *PaddingCut*

with_recording_path_prefix(*path*)

Return type *PaddingCut*

__init__(*id*, *duration*, *sampling_rate*, *feat_value*, *num_frames=None*, *num_features=None*, *frame_shift=None*, *num_samples=None*)
Initialize self. See help(type(self)) for accurate signature.

class lhotse.cut.**MixTrack**(*cut: Union[*lhotse.cut.Cut, lhotse.cut.PaddingCut*]*, *offset: float = 0.0*, *snr: Optional[float] = None*)
Represents a single track in a mix of Cuts. Points to a specific Cut and holds information on how to mix it with other Cuts, relative to the first track in a mix.

cut: Union[Cut, PaddingCut]

offset: float = 0.0

snr: Optional[float] = None

static from_dict(*data*)

__init__(*cut*, *offset=0.0*, *snr=None*)
Initialize self. See help(type(self)) for accurate signature.

class lhotse.cut.**MixedCut**(*id: str*, *tracks: List[*lhotse.cut.MixTrack*]*)
Represents a Cut that's created from other Cuts via mix or append operations. The actual mixing operations are performed upon loading the features into memory. In order to load the features, it needs to access the CutSet object that holds the "ingredient" cuts, as it only holds their IDs ("pointers"). The SNR and offset of all the tracks are specified relative to the first track.

id: str

tracks: List[MixTrack]

property supervisions
Lists the supervisions of the underlying source cuts. Each segment start time will be adjusted by the track offset.

Return type List[*SupervisionSegment*]

property start

Return type float

property end

Return type float

property duration

Return type float

property has_features

Return type bool

property has_recording

Return type bool

property num_frames

Return type Optional[int]

property frame_shift

Return type Optional[float]

property sampling_rate

Return type Optional[int]

property num_samples

Return type Optional[int]

property num_features

Return type Optional[int]

property features_type

Return type Optional[str]

truncate(*, *offset=0.0*, *duration=None*, *keep_excessive_supervisions=True*, *preserve_id=False*, *_supervisions_index=None*)

Returns a new MixedCut that is a sub-region of the current MixedCut. This method truncates the underlying Cuts and modifies their offsets in the mix, as needed. Tracks that do not fit in the truncated cut are removed.

Note that no operation is done on the actual features - it's only during the call to load_features() when the actual changes happen (a subset of features is loaded).

Parameters

- **offset** (`float`) – float (seconds), controls the start of the new cut relative to the current MixedCut's start.
- **duration** (`Optional[float]`) – optional float (seconds), controls the duration of the resulting MixedCut. By default, the duration is (end of the cut before truncation) - (offset).
- **keep_excessive_supervisions** (`bool`) – bool. Since trimming may happen inside a SupervisionSegment, the caller has an option to either keep or discard such supervisions.
- **preserve_id** (`bool`) – bool. Should the truncated cut keep the same ID or get a new, random one.

Return type `Union[`*Cut*, *MixedCut*, *PaddingCut*`]`

Returns a new MixedCut instance.

pad(*duration*, *pad_feat_value=-23.025850929940457*)

Return a new MixedCut, padded to `duration` seconds with zeros in the recording, and `pad_feat_value` in each feature bin.

Parameters

- **duration** (`float`) – The cut's minimal duration after padding.
- **pad_feat_value** (`float`) – A float value that's used for padding the features. By default we assume a log-energy floor of approx. -23 (1e-10 after exp).

Return type `Union[`*Cut*, *MixedCut*, *PaddingCut*`]`

Returns a padded MixedCut if duration is greater than this cut's duration, otherwise `self`.

perturb_speed(*factor*, *affix_id=True*)

Return a new `MixedCut` that will lazily perturb the speed while loading audio. The `num_samples`, `start` and `duration` fields of the underlying Cuts (and their Recordings and SupervisionSegments) are updated to reflect the shrinking/extending effect of speed. We are also updating the offsets of all underlying tracks.

Parameters

- **factor** (`float`) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (`bool`) – When true, we will modify the `MixedCut.id` field by affixing it with "_sp{factor}".

Return type *MixedCut*

Returns a modified copy of the current `MixedCut`.

load_features(*mixed=True*)

Loads the features of the source cuts and mixes them on-the-fly.

Parameters **mixed** (bool) – when True (default), returns a 2D array of features mixed in the feature domain. Otherwise returns a 3D array with the first dimension equal to the number of tracks.

Return type Optional[ndarray]

Returns A numpy ndarray with features and with shape (num_frames, num_features), or (num_tracks, num_frames, num_features)

load_audio(*mixed=True*)

Loads the audios of the source cuts and mix them on-the-fly.

Parameters **mixed** (bool) – When True (default), returns a mono mix of the underlying tracks. Otherwise returns a numpy array with the number of channels equal to the number of tracks.

Return type Optional[ndarray]

Returns A numpy ndarray with audio samples and with shape (num_channels, num_samples)

plot_tracks_features()

Display the feature matrix as an image. Requires matplotlib to be installed.

plot_tracks_audio()

Display plots of the individual tracks' waveforms. Requires matplotlib to be installed.

compute_and_store_features(*extractor*, *storage*, *augment_fn=None*, *mix_eagerly=True*)

Compute the features from this cut, store them on disk, and create a new *Cut* object with the feature manifest attached. This cut has to be able to load audio.

Parameters

- **extractor** (*FeatureExtractor*) – a FeatureExtractor instance used to compute the features.
- **storage** (*FeaturesWriter*) – a FeaturesWriter instance used to store the features.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation.
- **mix_eagerly** (bool) – when False, extract and store the features for each track separately, and mix them dynamically when loading the features. When True, mix the audio first and store the mixed features, returning a new Cut instance with the same ID. The returned Cut will not have a Recording attached.

Return type Union[*Cut*, *MixedCut*, *PaddingCut*]

Returns a new Cut instance if mix_eagerly is True, or returns self with each of the tracks containing the Features manifests.

map_supervisions(*transform_fn*)

Modify the SupervisionSegments by *transform_fn* of this MixedCut.

Parameters **transform_fn** (Callable[[*SupervisionSegment*], *SupervisionSegment*]) – a function that modifies a supervision as an argument.

Return type Union[*Cut*, *MixedCut*, *PaddingCut*]

Returns a modified MixedCut.

filter_supervisions(*predicate*)

Modify cut to store only supervisions accepted by *predicate*

Example:

```
>>> cut = cut.filter_supervisions(lambda s: s.id in supervision_ids)
>>> cut = cut.filter_supervisions(lambda s: s.duration < 5.0)
>>> cut = cut.filter_supervisions(lambda s: s.text is not None)
```

Parameters **predicate** (Callable[[*SupervisionSegment*], bool]) – A callable that accepts *SupervisionSegment* and returns bool

Return type Union[*Cut*, *MixedCut*, *PaddingCut*]

Returns a modified Cut

static from_dict(*data*)

Return type *MixedCut*

with_features_path_prefix(*path*)

Return type *MixedCut*

with_recording_path_prefix(*path*)

Return type *MixedCut*

__init__(*id*, *tracks*)
Initialize self. See help(type(self)) for accurate signature.

class lhotse.cut.**CutSet**(**args*, ***kwds*)
CutSet combines features with their corresponding supervisions. It may have wider span than the actual supervisions, provided the features for the whole span exist. It is the basic building block of PyTorch-style Datasets for speech/audio processing tasks.

cuts: Dict[str, AnyCut]

property mixed_cuts

Return type Dict[str, *MixedCut*]

property simple_cuts

Return type Dict[str, *Cut*]

property ids

Return type Iterable[str]

property speakers

Return type FrozenSet[str]

static from_cuts(*cuts*)

Return type *CutSet*

static from_manifests(*recordings=None*, *supervisions=None*, *features=None*, *random_ids=False*)
Create a CutSet from any combination of supervision, feature and recording manifests. At least one of recording_set or feature_set is required. The Cut boundaries correspond to those found in the feature_set, when available, otherwise to those found in the recording_set When a supervision_set is provided, we'll attach to the Cut all supervisions that have a matching recording ID and are fully contained in the Cut's boundaries.

Parameters

- **recordings** (Optional[*RecordingSet*]) – a RecordingSet manifest.

- **supervisions** (Optional[*SupervisionSet*]) – a SupervisionSet manifest.
- **features** (Optional[*FeatureSet*]) – a FeatureSet manifest.
- **random_ids** (bool) – boolean, should the cut IDs be randomized. By default, use the recording ID with a loop index and a channel idx, i.e. "{recording_id}-{idx}-{channel}")

Return type *CutSet*

Returns a new CutSet instance.

static from_dicts(*data*)

Return type *CutSet*

to_dicts()

Return type List[dict]

describe()

Print a message describing details about the CutSet - the number of cuts and the duration statistics, including the total duration and the percentage of speech segments.

Example output: Cuts count: 547 Total duration (hours): 326.4 Speech duration (hours): 79.6 (24.4%) **** Duration statistics (seconds): mean 2148.0 std 870.9 min 477.0 25% 1523.0 50% 2157.0 75% 2423.0 max 5415.0 dtype: float64

Return type None

split(*num_splits*, *shuffle=False*)

Split the CutSet into num_splits pieces of equal size.

Parameters

- **num_splits** (int) – Requested number of splits.
- **shuffle** (bool) – Optionally shuffle the cuts order first.

Return type List[*CutSet*]

Returns A list of CutSet pieces.

subset(*, *supervision_ids=None*, *cut_ids=None*, *first=None*, *last=None*)

Return a new CutSet according to the selected subset criterion. Only a single argument to subset is supported at this time.

Example:

```
>>> cuts = CutSet.from_yaml('path/to/cuts')
>>> train_set = cuts.subset(supervision_ids=train_ids)
>>> test_set = cuts.subset(supervision_ids=test_ids)
```

Parameters

- **supervision_ids** (Optional[Iterable[str]]) – List of supervision IDs to keep.
- **cut_ids** (Optional[Iterable[str]]) – List of cut IDs to keep.
- **first** (Optional[int]) – int, the number of first cuts to keep.
- **last** (Optional[int]) – int, the number of last cuts to keep.

Return type *CutSet*

Returns a new CutSet with the subset results.

filter_supervisions(*predicate*)
Return a new CutSet with Cuts containing only *SupervisionSegments* satisfying *predicate*

Cuts without supervisions are preserved

Example:

```
>>> cuts = CutSet.from_yaml('path/to/cuts')
>>> at_least_five_second_supervisions = cuts.filter_supervisions(lambda
↪s: s.duration >= 5)
```

Parameters **predicate** (Callable[[*SupervisionSegment*], bool]) – A callable that accepts *SupervisionSegment* and returns bool

Return type *CutSet*

Returns a CutSet with filtered supervisions

filter(*predicate*)
Return a new CutSet with the Cuts that satisfy the *predicate*.

Parameters **predicate** (Callable[[Union[*Cut*, *MixedCut*, *PaddingCut*]], bool]) – a function that takes a cut as an argument and returns bool.

Return type *CutSet*

Returns a filtered CutSet.

trim_to_supervisions()
Return a new CutSet with Cuts that have identical spans as their supervisions.

Return type *CutSet*

Returns a CutSet.

trim_to_unsupervised_segments()
Return a new CutSet with Cuts created from segments that have no supervisions (likely silence or noise).

Return type *CutSet*

Returns a CutSet.

mix_same_recording_channels()
Find cuts that come from the same recording and have matching start and end times, but represent different channels. Then, mix them together (in matching groups) and return a new CutSet that contains their mixes. This is useful for processing microphone array recordings.

It is intended to be used as the first operation after creating a new CutSet (but might also work in other circumstances, e.g. if it was cut to windows first).

Example:

```
>>> ami = prepare_ami('path/to/ami')
>>> cut_set = CutSet.from_manifests(recordings=ami['train']['recordings'])
>>> multi_channel_cut_set = cut_set.mix_same_recording_channels()
```

In the AMI example, the multi_channel_cut_set will yield MixedCuts that hold all single-channel Cuts together.

Return type *CutSet*

sort_by_duration(*ascending=False*)
Sort the CutSet according to cuts duration and return the result. Descending by default.

Return type *CutSet*

sort_like(*other*)
Sort the CutSet according to the order of cut IDs in `other` and return the result.

Return type *CutSet*

index_supervisions(*index_mixed_tracks=False*)
Create a two-level index of supervision segments. It is a mapping from a Cut's ID to an interval tree that contains the supervisions of that Cut.

The interval tree can be efficiently queried for overlapping and/or enveloping segments. It helps speed up some operations on Cuts of very long recordings (1h+) that contain many supervisions.

Parameters **index_mixed_tracks** (`bool`) – Should the tracks of MixedCut's be indexed as additional, separate entries.

Return type `Dict[str, IntervalTree]`

Returns a mapping from Cut ID to an interval tree of SupervisionSegments.

pad(*duration=None*, *pad_feat_value=-23.025850929940457*)
Return a new CutSet with Cuts padded to `duration` in seconds. Cuts longer than `duration` will not be affected. Cuts will be padded to the right (i.e. after the signal).

Parameters

- **duration** (`Optional[float]`) – The cuts minimal duration after padding. When not specified, we'll choose the duration of the longest cut in the CutSet.
- **pad_feat_value** (`float`) – A float value that's used for padding the features. By default we assume a log-energy floor of approx. -23 (1e-10 after exp).

Return type *CutSet*

Returns A padded CutSet.

truncate(*max_duration*, *offset_type*, *keep_excessive_supervisions=True*, *preserve_id=False*)
Return a new CutSet with the Cuts truncated so that their durations are at most *max_duration*. Cuts shorter than *max_duration* will not be changed. :type max_duration: `float` :param max_duration: float, the maximum duration in seconds of a cut in the resulting manifest. :type offset_type: `str` :param offset_type: str, can be: - 'start' => cuts are truncated from their start; - 'end' => cuts are truncated from their end minus max_duration; - 'random' => cuts are truncated randomly between their start and their end minus max_duration :type keep_excessive_supervisions: `bool` :param keep_excessive_supervisions: bool. When a cut is truncated in the middle of a supervision segment, should the supervision be kept. :type preserve_id: `bool` :param preserve_id: bool. Should the truncated cut keep the same ID or get a new, random one. :rtype: *CutSet* :return: a new CutSet instance with truncated cuts.

cut_into_windows(*duration*, *keep_excessive_supervisions=True*)
Return a new `CutSet`, made by traversing each `Cut` in windows of `duration` seconds and creating new `Cut` out of them.

The last window might have a shorter duration if there was not enough audio, so you might want to use either `.filter()` or `.pad()` afterwards to obtain a uniform duration `CutSet`.

Parameters

- **duration** (`float`) – Desired duration of the new cuts in seconds.
- **keep_excessive_supervisions** (`bool`) – bool. When a cut is truncated in the middle of a supervision segment, should the supervision be kept.

Return type *CutSet*

Returns a new CutSet with cuts made from shorter duration windows.

sample(*n_cuts=1*)
Randomly sample this `CutSet` and return `n_cuts` cuts. When `n_cuts` is 1, will return a single cut instance; otherwise will return a `CutSet`.

Return type `Union[Cut, MixedCut, PaddingCut, CutSet]`

perturb_speed(*factor*, *affix_id=True*)

Return type *CutSet*

mix(*cuts*, *duration=None*, *snr=20*, *mix_prob=1.0*)
Mix cuts in this `CutSet` with randomly sampled cuts from another `CutSet`. A typical application would be data augmentation with noise, music, babble, etc.

Parameters

- **cuts** (*CutSet*) – a `CutSet` containing cuts to be mixed into this `CutSet`.
- **duration** (`Optional[float]`) – an optional float in seconds. When `None`, we will preserve the duration of the cuts in `self` (i.e. we'll truncate the mix if it exceeded the original duration). Otherwise, we will keep sampling cuts to mix in until we reach the specified `duration` (and truncate to that value, should it be exceeded).
- **snr** (`Union[float, Sequence[float], None]`) – an optional float, or pair (range) of floats, in decibels. When it's a single float, we will mix all cuts with this SNR level (where cuts in `self` are treated as signals, and cuts in `cuts` are treated as noise). When it's a pair of floats, we will uniformly sample SNR values from that range. When `None`, we will mix the cuts without any level adjustment (could be too noisy for data augmentation).
- **mix_prob** (`float`) – an optional float in range [0, 1]. Specifies the probability of performing a mix. Values lower than 1.0 mean that some cuts in the output will be unchanged.

Return type *CutSet*

Returns a new `CutSet` with mixed cuts.

compute_and_store_features(*extractor*, *storage_path*, *num_jobs=None*, *augment_fn=None*, *storage_type=<class 'lhotse.features.io.LilcomFilesWriter'>*, *executor=None*, *mix_eagerly=True*, *progress_bar=True*)
Extract features for all cuts, possibly in parallel, and store them using the specified storage object.

Examples:

Extract fbank features on one machine using 8 processes, store each array in a separate file with lilcom compression:

```
>>> cuts = CutSet(...)
... cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='feats',
...     num_jobs=8
... )
```

Extract fbank features on one machine using 8 processes, store arrays partitioned in 8 HDF5 files with lilcom compression:

```
>>> cuts = CutSet(...)
... cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='feats',
```

(continues on next page)

(continued from previous page)

```
...     num_jobs=8,
...     storage_type=LilcomHdf5Writer
... )
```

Extract fbank features on multiple machines using a Dask cluster with 80 jobs, store arrays partitioned in 80 HDF5 files with lilcom compression:

```
>>> from distributed import Client
... cuts = CutSet(...)
... cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='feats',
...     num_jobs=80,
...     storage_type=LilcomHdf5Writer,
...     executor=Client(...)
... )
```

Parameters

- **extractor** (*FeatureExtractor*) – A `FeatureExtractor` instance (either Lhotse's built-in or a custom implementation).
- **storage_path** (`Union[Path, str]`) – The path to location where we will store the features. The exact type and layout of stored files will be dictated by the `storage_type` argument.
- **num_jobs** (`Optional[int]`) – The number of parallel processes used to extract the features. We will internally split the CutSet into this many chunks and process each chunk in parallel.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use `CutSet.perturb_speed()` instead.
- **storage_type** (`Type[~FW]`) – a `FeaturesWriter` subclass type. It determines how the featurs are stored to disk, e.g. separate file per array, HDF5 files with multiple arrays, etc.
- **executor** (`Optional[Executor]`) – when provided, will be used to parallelize the feature extraction process. By default, we will instantiate a ProcessPoolExecutor. Learn more about the `Executor` API at https://lhotse.readthedocs.io/en/latest/parallelism.html
- **mix_eagerly** (`bool`) – Related to how the features are extracted for `MixedCut` instances, if any are present. When False, extract and store the features for each track separately, and mix them dynamically when loading the features. When True, mix the audio first and store the mixed features, returning a new `Cut` instance with the same ID. The returned `Cut` will not have a `Recording` attached.
- **progress_bar** (`bool`) – Should a progress bar be displayed (automatically turned off for parallel computation).

Return type *CutSet*

Returns Returns a new `CutSet` with `Features` manifests attached to the cuts.

compute_global_feature_stats(*storage_path=None*, *max_cuts=None*)
Compute the global means and standard deviations for each feature bin in the manifest. It follows the implementation in scikit-learn: https://github.com/scikit-learn/scikit-learn/blob/0fb307bf39bbdacd6ed713c00724f8f871d60370/sklearn/utils/extmath.py#L715 which follows the paper: "Algorithms for computing the sample variance: analysis and recommendations", by Chan, Golub, and LeVeque.

Parameters

- **storage_path** (Union[Path, str, None]) – an optional path to a file where the stats will be stored with pickle.
- **max_cuts** (Optional[int]) – optionally, limit the number of cuts used for stats estimation. The cuts will be selected randomly in that case.

Return a dict of ``{'norm_means'``{'norm_means' np.ndarray, 'norm_stds': np.ndarray}`` with the shape of the arrays equal to the number of feature bins in this manifest.

Return type Dict[str, ndarray]

with_features_path_prefix(*path*)

Return type *CutSet*

with_recording_path_prefix(*path*)

Return type *CutSet*

map(*transform_fn*)
Modify the cuts in this CutSet and return a new CutSet.

Parameters **transform_fn** (Callable[[Union[*Cut*, *MixedCut*, *PaddingCut*]], Union[*Cut*, *MixedCut*, *PaddingCut*]]) – A callable (function) that accepts a single cut instance and returns a single cut instance.

Return type *CutSet*

Returns a new CutSet with modified cuts.

modify_ids(*transform_fn*)
Modify the IDs of cuts in this CutSet. Useful when combining multiple ``CutSet``s that were created from a single source, but contain features with different data augmentations techniques.

Parameters **transform_fn** (Callable[[str], str]) – A callable (function) that accepts a string (cut ID) and returns

a new string (new cut ID). :rtype: *CutSet* :return: a new CutSet with cuts with modified IDs.

map_supervisions(*transform_fn*)
Modify the SupervisionSegments by *transform_fn* in this CutSet.

Parameters **transform_fn** (Callable[[*SupervisionSegment*], *SupervisionSegment*]) – a function that modifies a supervision as an argument.

Return type *CutSet*

Returns a new, modified CutSet.

transform_text(*transform_fn*)
Return a copy of this CutSet with all SupervisionSegments text transformed with transform_fn. Useful for text normalization, phonetic transcription, etc.

Parameters **transform_fn** (Callable[[str], str]) – a function that accepts a string and returns a string.

Return type *CutSet*

Returns a new, modified CutSet.

__init__(*cuts*)
Initialize self. See help(type(self)) for accurate signature.

lhotse.cut.**make_windowed_cuts_from_features**(*feature_set*, *cut_duration*, *cut_shift=None*, *keep_shorter_windows=False*)
Converts a FeatureSet to a CutSet by traversing each Features object in - possibly overlapping - windows, and creating a Cut out of that area. By default, the last window in traversal will be discarded if it cannot satisfy the *cut_duration* requirement.

Parameters

- **feature_set** (*FeatureSet*) – a FeatureSet object.
- **cut_duration** (`float`) – float, duration of created Cuts in seconds.
- **cut_shift** (`Optional[float]`) – optional float, specifies how many seconds are in between the starts of consecutive windows. Equals *cut_duration* by default.
- **keep_shorter_windows** (`bool`) – bool, when True, the last window will be used to create a Cut even if its duration is shorter than *cut_duration*.

Return type *CutSet*

Returns a CutSet object.

lhotse.cut.**mix**(*reference_cut*, *mixed_in_cut*, *offset=0*, *snr=None*)
Overlay, or mix, two cuts. Optionally the *mixed_in_cut* may be shifted by *offset* seconds and scaled down (positive SNR) or scaled up (negative SNR). Returns a MixedCut, which contains both cuts and the mix information. The actual feature mixing is performed during the call to `MixedCut.load_features()`.

Parameters

- **reference_cut** (`Union[Cut, MixedCut, PaddingCut]`) – The reference cut for the mix - offset and snr are specified w.r.t this cut.
- **mixed_in_cut** (`Union[Cut, MixedCut, PaddingCut]`) – The mixed-in cut - it will be offset and rescaled to match the offset and snr parameters.
- **offset** (`float`) – How many seconds to shift the `mixed_in_cut` w.r.t. the `reference_cut`.
- **snr** (`Optional[float]`) – Desired SNR of the *right_cut* w.r.t. the *left_cut* in the mix.

Return type *MixedCut*

Returns A MixedCut instance.

lhotse.cut.**append**(*left_cut*, *right_cut*, *snr=None*)
Helper method for functional-style appending of Cuts.

Return type *MixedCut*

lhotse.cut.**mix_cuts**(*cuts*)
Return a MixedCut that consists of the input Cuts mixed with each other as-is.

Return type *MixedCut*

lhotse.cut.**append_cuts**(*cuts*)
Return a MixedCut that consists of the input Cuts appended to each other as-is.

Return type `Union[Cut, MixedCut, PaddingCut]`

10.6 Recipes

Convenience methods used to prepare recording and supervision manifests for standard corpora.

10.7 Kaldi conversion

Convenience methods used to interact with Kaldi data directories.

lhotse.kaldi.**load_kaldi_data_dir**(*path*, *sampling_rate*)
Load a Kaldi data directory and convert it to a Lhotse RecordingSet and SupervisionSet manifests. For this to work, at least the wav.scp file must exist. SupervisionSet is created only when a segments file exists. All the other files (text, utt2spk, etc.) are optional, and some of them might not be handled yet. In particular, feats.scp files are ignored.

Return type Tuple[*RecordingSet*, Optional[*SupervisionSet*]]

lhotse.kaldi.**export_to_kaldi**(*recordings*, *supervisions*, *output_dir*)
Export a pair of RecordingSet and SupervisionSet to a Kaldi data directory. Currently, it only supports single-channel recordings that have a single AudioSource.

The RecordingSet and SupervisionSet must be compatible, i.e. it must be possible to create a CutSet out of them.

Parameters

- **recordings** (*RecordingSet*) – a RecordingSet manifest.
- **supervisions** (*SupervisionSet*) – a SupervisionSet manifest.
- **output_dir** (Union[Path, str]) – path where the Kaldi-style data directory will be created.

lhotse.kaldi.**load_kaldi_text_mapping**(*path*, *must_exist=False*)
Load Kaldi files such as utt2spk, spk2gender, text, etc. as a dict.

Return type Dict[str, Optional[str]]

lhotse.kaldi.**save_kaldi_text_mapping**(*data*, *path*)
Save flat dicts to Kaldi files such as utt2spk, spk2gender, text, etc.

10.8 Others

Helper methods used throughout the codebase.

lhotse.manipulation.**combine**(**manifests*)
Combine multiple manifests of the same type into one.

Examples:

```
>>> # Pass several arguments
>>> combine(recording_set1, recording_set2, recording_set3)
>>> # Or pass a single list/tuple of manifests
>>> combine([supervision_set1, supervision_set2])
```

Return type ~Manifest

lhotse.manipulation.**to_manifest**(*items*)

Take an iterable of data types in Lhotse such as Recording, SupervisonSegment or Cut, and create the manifest of the corresponding type. When the iterable is empty, returns None.

Return type Optional[~Manifest]

lhotse.manipulation.**load_manifest**(*path*)

Generic utility for reading an arbitrary manifest.

Return type ~Manifest

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W