Load audio data

You can load an audio dataset using the Audio feature that automatically decodes and resamples the audio files when you access the examples.
Audio decoding is based on the `torchcodec` python package, which uses the `FFmpeg` C library under the hood.

Installation

To work with audio datasets, you need to have the `audio` dependencies installed.
Check out the installation guide to learn how to install it.

Local files

You can load your own dataset using the paths to your audio files. Use the cast_column() function to take a column of audio file paths, and cast it to the Audio feature:

```
>>> audio_dataset = Dataset.from_dict({"audio": ["path/to/audio_1", "path/to/audio_2", ..., "path/
>>> audio_dataset[0]["audio"]
<datasets.features._torchcodec.AudioDecoder object at 0x11642b6a0>
```

AudioFolder

You can also load a dataset with an `AudioFolder` dataset builder. It does not require writing a custom dataloader, making it useful for quickly creating and loading audio datasets with several thousand audio files.

AudioFolder with metadata

To link your audio files with metadata information, make sure your dataset has a `metadata.csv` file. Your dataset structure might look like:

```
folder/train/metadata.csv
folder/train/first_audio_file.mp3
folder/train/second_audio_file.mp3
folder/train/third_audio_file.mp3
```

Your `metadata.csv` file must have a `file_name` column which links audio files with their metadata. An example `metadata.csv` file might look like:

```
file_name,transcription
first_audio_file.mp3,znowu się duch z ciałem zrośnie w młodocianej wstaniesz wiosnie i możesz skut
second_audio_file.mp3,już u źwierzyńca podwojów król zasiada przy nim książęta i panowie rada a gd
third_audio_file.mp3,pewnie kędyś w obłędzie ubite minęły szlaki zaczekajmy dzień jaki poślemy szu
```

`AudioFolder` will load audio data and create a `transcription` column containing texts from `metadata.csv`:

```
>>> from datasets import load_dataset

>>> dataset = load_dataset("username/dataset_name")
>>> # OR locally:
>>> dataset = load_dataset("/path/to/folder")
```

For local datasets, this is equivalent to passing `audiofolder` manually in load_dataset() and the directory in `data_dir`:

```
>>> dataset = load_dataset("audiofolder", data_dir="/path/to/folder")
```

Metadata can also be specified as JSON Lines, in which case use `metadata.jsonl` as the name of the metadata file. This format is helpful in scenarios when one of the columns is complex, e.g. a list of floats, to avoid parsing errors or reading the complex values as strings.

To ignore the information in the metadata file, set `drop_metadata=True` in load_dataset():

```
>>> from datasets import load_dataset

>>> dataset = load_dataset("username/dataset_with_metadata", drop_metadata=True)
```

If you don't have a metadata file, `AudioFolder` automatically infers the label name from the directory name.
If you want to drop automatically created labels, set `drop_labels=True`.
In this case, your dataset will only contain an audio column:

```
>>> from datasets import load_dataset

>>> dataset = load_dataset("username/dataset_without_metadata", drop_labels=True)
```

Finally the `filters` argument lets you load only a subset of the dataset, based on a condition on the label or the metadata. This is especially useful if the metadata is in Parquet format, since this format enables fast filtering. It is also recommended to use this argument with `streaming=True`, because by default the dataset is fully downloaded before filtering.

```
>>> filters = [("label", "=", 0)]
>>> dataset = load_dataset("username/dataset_name", streaming=True, filters=filters)
```

> [!TIP]
> For more information about creating your own `AudioFolder` dataset, take a look at the Create an audio dataset guide.

For a guide on how to load any type of dataset, take a look at the general loading guide.

Audio decoding

By default, audio files are decoded sequentially as torchcodec `AudioDecoder` objects when you iterate on a dataset.
However it is possible to speed up the dataset significantly using multithreaded decoding:

```
>>> import os
>>> num_threads = num_threads = min(32, (os.cpu_count() or 1) + 4)
>>> dataset = dataset.decode(num_threads=num_threads)
>>> for example in dataset:  # up to 20 times faster !
...     ...
```

You can enable multithreading using `num_threads`. This is especially useful to speed up

remote data streaming.
However it can be slower than `num_threads=0` for local data on fast disks.

If you are not interested in the images decoded as NumPy arrays and would like to access the path/bytes instead, you can disable decoding:

```
>>> dataset = dataset.decode(False)
```

Note: IterableDataset.decode() is only available for streaming datasets at the moment.