Dataset features

Features defines the internal structure of a dataset. It is used to specify the underlying serialization format. What's more interesting to you though is that Features contains high-level information about everything from the column names and types, to the ClassLabel. You can think of Features as the backbone of a dataset.

The Features format is simple: `dict[column_name, column_type]`. It is a dictionary of column name and column type pairs. The column type provides a wide range of options for describing the type of data you have.

Let's have a look at the features of the MRPC dataset from the GLUE benchmark:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset('nyu-mll/glue', 'mrpc', split='train')
>>> dataset.features
{'idx': Value('int32'),
 'label': ClassLabel(names=['not_equivalent', 'equivalent']),
 'sentence1': Value('string'),
 'sentence2': Value('string'),
}
```

The Value feature tells 🤗 Datasets:

- The `idx` data type is `int32`.
- The `sentence1` and `sentence2` data types are `string`.

🤗 Datasets supports many other data types such as `bool`, `float32` and `binary` to name just a few.

> [!TIP]
> Refer to Value for a full list of supported data types.

The ClassLabel feature informs 🤗 Datasets the `label` column contains two classes. The classes are labeled `not_equivalent` and `equivalent`. Labels are stored as integers in the dataset. When you retrieve the labels, ClassLabel.int2str() and ClassLabel.str2int() carries out the conversion from integer value to label name, and vice versa.

If your data type contains a list of objects, then you want to use the List feature. Remember the SQuAD dataset?

```
>>> from datasets import load_dataset
>>> dataset = load_dataset('rajpurkar/squad', split='train')
>>> dataset.features
{'id': Value('string'),
 'title': Value('string'),
 'context': Value('string'),
 'question': Value('string'),
 'answers': {'text': List(Value('string')),
  'answer_start': List(Value('int32'))}}
```

The `answers` field is constructed using the dict of features because and contains two subfields, `text` and `answer_start`, which are lists of `string` and `int32`, respectively.

> [!TIP]
> See the flatten section to learn how you can extract the nested subfields as their own independent columns.

The array feature type is useful for creating arrays of various sizes. You can create arrays with two dimensions using Array2D, and even arrays with five dimensions using Array5D.

```
>>> features = Features({'a': Array2D(shape=(1, 3), dtype='int32')})
```

The array type also allows the first dimension of the array to be dynamic. This is useful for handling sequences with variable lengths such as sentences, without having to pad or truncate the input to a uniform shape.

```
>>> features = Features({'a': Array3D(shape=(None, 5, 2), dtype='int32')})
```

Audio feature

Audio datasets have a column with type Audio, which contains three important fields:

- `array`: the decoded audio data represented as a 1-dimensional array.

- `path`: the path to the downloaded audio file.
- `sampling_rate`: the sampling rate of the audio data.

When you load an audio dataset and call the audio column, the Audio feature automatically decodes and resamples the audio file:

```
>>> from datasets import load_dataset, Audio

>>> dataset = load_dataset("PolyAI/minds14", "en-US", split="train")
>>> dataset[0]["audio"]
<datasets.features._torchcodec.AudioDecoder object at 0x11642b6a0>
```

> [!WARNING]
> Index into an audio dataset using the row index first and then the `audio` column - `dataset[0]["audio"]` - to avoid decoding and resampling all the audio files in the dataset. Otherwise, this can be a slow and time-consuming process if you have a large dataset.

With `decode=False`, the Audio type simply gives you the path or the bytes of the audio file, without decoding it into an torchcodec `AudioDecoder` object,

```
>>> dataset = load_dataset("PolyAI/minds14", "en-US", split="train").cast_column("audio", Audio(de
>>> dataset[0]
{'audio': {'bytes': None,
  'path': '/root/.cache/huggingface/datasets/downloads/extracted/f14948e0e84be638dd7943ac36518a4cf
 'english_transcription': 'I would like to set up a joint account with my partner',
 'intent_class': 11,
 'lang_id': 4,
 'path': '/root/.cache/huggingface/datasets/downloads/extracted/f14948e0e84be638dd7943ac36518a4cf3
 'transcription': 'I would like to set up a joint account with my partner'}
```

Image feature

Image datasets have a column with type Image, which loads `PIL.Image` objects from images stored as bytes:

When you load an image dataset and call the image column, the Image feature automatically decodes the image file:

```
>>> from datasets import load_dataset, Image

>>> dataset = load_dataset("AI-Lab-Makerere/beans", split="train")
>>> dataset[0]["image"]
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=500x500 at 0x125506CF8>
```

> [!WARNING]
> Index into an image dataset using the row index first and then the `image` column - `dataset[0]["image"]` - to avoid decoding all the image files in the dataset. Otherwise, this can be a slow and time-consuming process if you have a large dataset.

With `decode=False`, the Image type simply gives you the path or the bytes of the image file, without decoding it into an `PIL.Image`,

```
>>> dataset = load_dataset("AI-Lab-Makerere/beans", split="train").cast_column("image", Image(deco
>>> dataset[0]["image"]
{'bytes': None,
 'path': '/Users/username/.cache/huggingface/datasets/downloads/extracted/772e7c1fba622cff102b85dd
```

Depending on the dataset, you may get the path to the local downloaded image, or the content of the image as bytes if the dataset is not made of individual files.

You can also define a dataset of images from numpy arrays:

```
>>> ds = Dataset.from_dict({"i": [np.zeros(shape=(16, 16, 3), dtype=np.uint8)]}, features=Features
```

And in this case the numpy arrays are encoded into PNG (or TIFF if the pixels values precision is important).

For multi-channels arrays like RGB or RGBA, only uint8 is supported. If you use a larger precision, you get a warning and the array is downcasted to uint8.
For gray-scale images you can use the integer or float precision you want as long as it is compatible with `Pillow`. A warning is shown if your image integer or float precision is too high, and in this case the array is downcated: an int64 array is downcasted to int32, and a float64 array is downcasted to float32.