Main classes

DatasetInfodatasets.DatasetInfo

class datasets.DatasetInfodatasets.DatasetInfohttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/info.py#L92[{"name": "description", "val": ": str = "}, {"name": "citation", "val": ": str = "}, {"name": "homepage", "val": ": str = "}, {"name": "license", "val": ": str = "}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "post_processed", "val": ": typing.Optional[datasets.info.PostProcessedInfo] = None"}, {"name": "supervised_keys", "val": ": typing.Optional[datasets.info.SupervisedKeysData] = None"}, {"name": "builder_name", "val": ": typing.Optional[str] = None"}, {"name": "dataset_name", "val": ": typing.Optional[str] = None"}, {"name": "config_name", "val": ": typing.Optional[str] = None"}, {"name": "version", "val": ": typing.Union[str, datasets.utils.version.Version, NoneType] = None"}, {"name": "splits", "val": ": typing.Optional[dict] = None"}, {"name": "download_checksums", "val": ": typing.Optional[dict] = None"}, {"name": "download_size", "val": ": typing.Optional[int] = None"}, {"name": "post_processing_size", "val": ": typing.Optional[int] = None"}, {"name": "dataset_size", "val": ": typing.Optional[int] = None"}, {"name": "size_in_bytes", "val": ": typing.Optional[int] = None"}]- **description** (`str`) --
A description of the dataset.

- **citation** (`str`) --
  A BibTeX citation of the dataset.
- **homepage** (`str`) --
  A URL to the official homepage for the dataset.
- **license** (`str`) --
  The dataset's license. It can be the name of the license or a paragraph containing the terms of the license.
- **features** (Features, *optional*) --
  The features used to specify the dataset's column types.
- **post_processed** (`PostProcessedInfo` , *optional*) --
  Information regarding the resources of a possible post-processing of a dataset. For example, it can contain the information of an index.
- **supervised_keys** (`SupervisedKeysData` , *optional*) --
  Specifies the input feature and the label for supervised learning if applicable for the

dataset (legacy from TFDS).

- **builder_name** (`str` , *optional*) --
  The name of the `GeneratorBasedBuilder` subclass used to create the dataset. It is also the snake_case version of the dataset builder class name.
- **config_name** (`str` , *optional*) --
  The name of the configuration derived from BuilderConfig.
- **version** (`str` or Version, *optional*) --
  The version of the dataset.
- **splits** (`dict` , *optional*) --
  The mapping between split name and metadata.
- **download_checksums** (`dict` , *optional*) --
  The mapping between the URL to download the dataset's checksums and corresponding metadata.
- **download_size** (`int` , *optional*) --
  The size of the files to download to generate the dataset, in bytes.
- **post_processing_size** (`int` , *optional*) --
  Size of the dataset in bytes after post-processing, if any.
- **dataset_size** (`int` , *optional*) --
  The combined size in bytes of the Arrow tables for all splits.
- **size_in_bytes** (`int` , *optional*) --
  The combined size in bytes of all files associated with the dataset (downloaded files + Arrow files).
- ****config_kwargs** (additional keyword arguments) --
  Keyword arguments to be passed to the BuilderConfig and used in the DatasetBuilder.0
  Information about a dataset.

`DatasetInfo` documents datasets, including its name, version, and features.
See the constructor arguments and properties for a full list.

Not all fields are known on construction and may be updated later.

from_directorydatasets.DatasetInfo.from_directoryhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/info.py#L247[{"name": "dataset_info_dir", "val": ": str"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}]- **dataset_info_dir** (`str`) --
The directory containing the metadata file. This
should be the root directory of a specific dataset version.

- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
  0
  Create DatasetInfo from the JSON file in `dataset_info_dir` .

This function updates all the dynamically generated fields (num_examples, hash, time of creation,...) of the DatasetInfo.

This will overwrite all previous metadata.

Example:

```
>>> from datasets import DatasetInfo
>>> ds_info = DatasetInfo.from_directory("/path/to/directory/")
```

write_to_directorydatasets.DatasetInfo.write_to_directoryhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/info.py#L186[{"name": "dataset_info_dir", "val": ""}, {"name": "pretty_print", "val": " = False"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}]- **dataset_info_dir** (`str`) --
Destination directory.

- **pretty_print** (`bool` , defaults to `False`) --
  If `True` , the JSON will be pretty-printed with the indent level of 4.
- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
  0
  Write `DatasetInfo` and license (if present) as JSON files to `dataset_info_dir` .

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.info.write_to_directory("/path/to/directory/")
```

Datasetdatasets.Dataset

The base class Dataset implements a Dataset backed by an Apache Arrow table.

class datasets.Datasetdatasets.Datasethttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L695[{"name": "arrow_table", "val": ": Table"}, {"name": "info", "val": ": typing.Optional[datasets.info.DatasetInfo] = None"}, {"name": "split", "val": ": typing.Optional[datasets.splits.NamedSplit] = None"}, {"name": "indices_table", "val": ": typing.Optional[datasets.table.Table] = None"}, {"name": "fingerprint", "val": ": typing.Optional[str] = None"}]
A Dataset backed by an Arrow table.

add_columndatasets.Dataset.add_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L6059[{"name": "name", "val": ": str"}, {"name": "column", "val": ": typing.Union[list, numpy.ndarray]"}, {"name": "new_fingerprint", "val": ": str"}, {"name": "feature", "val": ": typing.Union[dict, list, tuple, datasets.features.features.Value, datasets.features.features.ClassLabel, datasets.features.translation.Translation, datasets.features.translation.TranslationVariableLanguages, datasets.features.features.LargeList, datasets.features.features.List, datasets.features.features.Array2D, datasets.features.features.Array3D, datasets.features.features.Array4D, datasets.features.features.Array5D, datasets.features.audio.Audio, datasets.features.image.Image, datasets.features.video.Video, datasets.features.pdf.Pdf, NoneType] = None"}]- **name** (`str`) --
Column name.

- **column** (`list` or `np.array`) --
  Column data to be added.
- **feature** (`FeatureType` or `None` , defaults to `None`) --
  Column datatype.0Dataset
  Add column to Dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> more_text = ds["text"]
>>> ds = ds.add_column(name="text_2", column=more_text)
>>> ds
Dataset({
    features: ['text', 'label', 'text_2'],
    num_rows: 1066
})
```

add_itemdatasets.Dataset.add_itemhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L6317[{"name": "item", "val": ": dict"}, {"name": "new_fingerprint", "val": ": str"}]- **item** (`dict`) --
Item data to be added.0Dataset
Add item to Dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> new_review = {'label': 0, 'text': 'this movie is the absolute worst thing I have ever seen'}
>>> ds = ds.add_item(new_review)
>>> ds[-1]
{'label': 0, 'text': 'this movie is the absolute worst thing I have ever seen'}
```

from_filedatasets.Dataset.from_filehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L778[{"name": "filename", "val": ": str"}, {"name": "info", "val": ": typing.Optional[datasets.info.DatasetInfo] = None"}, {"name": "split", "val": ": typing.Optional[datasets.splits.NamedSplit] = None"}, {"name": "indices_filename", "val": ": typing.Optional[str] = None"}, {"name": "in_memory", "val": ": bool = False"}]- **filename** (`str`) --
File name of the dataset.

- **info** (`DatasetInfo` , *optional*) --
  Dataset information, like description, citation, etc.
- **split** (`NamedSplit` , *optional*) --
  Name of the dataset split.
- **indices_filename** (`str` , *optional*) --

File names of the indices.

- **in_memory** (`bool` , defaults to `False`) --
  Whether to copy the data in-memory.0Dataset
  Instantiate a Dataset backed by an Arrow table at filename.

from_bufferdatasets.Dataset.from_bufferhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L818[{"name": "buffer", "val": ": Buffer"}, {"name": "info", "val": ": typing.Optional[datasets.info.DatasetInfo] = None"}, {"name": "split", "val": ": typing.Optional[datasets.splits.NamedSplit] = None"}, {"name": "indices_buffer", "val": ": typing.Optional[pyarrow.lib.Buffer] = None"}]- **buffer** (`pyarrow.Buffer`) --
Arrow buffer.

- **info** (`DatasetInfo` , *optional*) --
  Dataset information, like description, citation, etc.
- **split** (`NamedSplit` , *optional*) --
  Name of the dataset split.
- **indices_buffer** (`pyarrow.Buffer` , *optional*) --
  Indices Arrow buffer.0Dataset
  Instantiate a Dataset backed by an Arrow buffer.

from_pandasdatasets.Dataset.from_pandashttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L850[{"name": "df", "val": ": DataFrame"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "info", "val": ": typing.Optional[datasets.info.DatasetInfo] = None"}, {"name": "split", "val": ": typing.Optional[datasets.splits.NamedSplit] = None"}, {"name": "preserve_index", "val": ": typing.Optional[bool] = None"}]- **df** (`pandas.DataFrame`) --
Dataframe that contains the dataset.

- **features** (Features, *optional*) --
  Dataset features.
- **info** (`DatasetInfo` , *optional*) --
  Dataset information, like description, citation, etc.
- **split** (`NamedSplit` , *optional*) --
  Name of the dataset split.
- **preserve_index** (`bool` , *optional*) --
  Whether to store the index as an additional column in the resulting Dataset.

The default of `None` will store the index as a column, except for `RangeIndex` which is stored as metadata only.
Use `preserve_index=True` to force it to be stored as a column.0Dataset

Convert `pandas.DataFrame` to a `pyarrow.Table` to create a Dataset.

The column types in the resulting Arrow Table are inferred from the dtypes of the `pandas.Series` in the
DataFrame. In the case of non-object Series, the NumPy dtype is translated to its Arrow equivalent. In the
case of `object`, we need to guess the datatype by looking at the Python objects in this Series.

Be aware that Series of the `object` dtype don't carry enough information to always lead to a meaningful Arrow
type. In the case that we cannot infer a type, e.g. because the DataFrame is of length 0 or the Series only
contains `None/nan` objects, the type is set to `null`. This behavior can be avoided by constructing explicit
features and passing it to this function.

Important: a dataset created with from_pandas() lives in memory
and therefore doesn't have an associated cache directory.
This may change in the future, but in the meantime if you
want to reduce memory usage you should write it back on disk
and reload using e.g. save_to_disk / load_from_disk.

Example:

```
>>> ds = Dataset.from_pandas(df)
```

from_dictdatasets.Dataset.from_dicthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L964[{"name": "mapping", "val": ": dict"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "info", "val": ": typing.Optional[datasets.info.DatasetInfo] = None"}, {"name": "split", "val": ": typing.Optional[datasets.splits.NamedSplit] = None"}]- **mapping** (`Mapping`) --
Mapping of strings to Arrays or Python lists.

- **features** (Features, *optional*) --
  Dataset features.
- **info** (`DatasetInfo` , *optional*) --
  Dataset information, like description, citation, etc.
- **split** (`NamedSplit` , *optional*) --
  Name of the dataset split.0Dataset

Convert `dict` to a `pyarrow.Table` to create a Dataset.

Important: a dataset created with from_dict() lives in memory
and therefore doesn't have an associated cache directory.
This may change in the future, but in the meantime if you
want to reduce memory usage you should write it back on disk
and reload using e.g. save_to_disk / load_from_disk.

from_generatordatasets.Dataset.from_generatorhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1114[{"name": "generator", "val": ": typing.Callable"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "cache_dir", "val": ": str = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "gen_kwargs", "val": ": typing.Optional[dict] = None"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "split", "val": ": NamedSplit = NamedSplit('train')"}, {"name": "**kwargs", "val": ""}]- **generator** (-- `Callable`):
A generator function that `yields` examples.

- **features** (Features, *optional*) --
  Dataset features.
- **cache_dir** (`str` , *optional*, defaults to `"~/.cache/huggingface/datasets"`) --
  Directory to cache data.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Whether to copy the data in-memory.
- **gen_kwargs(`dict` ,** *optional*) --
  Keyword arguments to be passed to the `generator` callable.
  You can define a sharded dataset by passing the list of shards in `gen_kwargs` and setting `num_proc` greater than 1.
- **num_proc** (`int` , *optional*, defaults to `None`) --
  Number of processes when downloading and generating the dataset locally.

This is helpful if the dataset is made of multiple files. Multiprocessing is disabled by default.
If `num_proc` is greater than one, then all list values in `gen_kwargs` must be the same length. These values will be split between calls to the generator. The number of shards will be the minimum of the shortest list in `gen_kwargs` and `num_proc`.

- **split** (NamedSplit, defaults to `Split.TRAIN`) --
  Split name to be assigned to the dataset.
- ****kwargs** (additional keyword arguments) --
  Keyword arguments to be passed to :`GeneratorConfig`.0Dataset
  Create a Dataset from a generator.

Example:

```
>>> def gen():
...     yield {"text": "Good", "label": 0}
...     yield {"text": "Bad", "label": 1}
...
>>> ds = Dataset.from_generator(gen)
```

```
>>> def gen(shards):
...     for shard in shards:
...         with open(shard) as f:
...             for line in f:
...                 yield {"line": line}
...
>>> shards = [f"data{i}.txt" for i in range(32)]
>>> ds = Dataset.from_generator(gen, gen_kwargs={"shards": shards})
```

datadatasets.Dataset.datahttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1808[]
The Apache Arrow table backing the dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.data
MemoryMappedTable
text: string
label: int64
----
text: [["compassionately explores the seemingly irreconcilable situation between conservative chri
label: [[1,1,1,1,1,1,1,1,1,1,...,0,0,0,0,0,0,0,0,0,0]]
```

cache_filesdatasets.Dataset.cache_fileshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1828[]
The cache files containing the Apache Arrow table backing the dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.cache_files
[{'filename': '/root/.cache/huggingface/datasets/rotten_tomatoes_movie_review/default/1.0.0/40d411
```

num_columnsdatasets.Dataset.num_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1846[]
Number of columns in the dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.num_columns
2
```

num_rowsdatasets.Dataset.num_rowshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1861[]
Number of rows in the dataset (same as Dataset.**len**()).

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.num_rows
1066
```

column_namesdatasets.Dataset.column_nameshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1878[]
Names of the columns in the dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.column_names
['text', 'label']
```

shapedatasets.Dataset.shapehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1893[]
Shape of the dataset (number of columns, number of rows).

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.shape
(1066, 2)
```

uniquedatasets.Dataset.uniquehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1910[{"name": "column", "val": ": str"}]- **column** (`str`) --
Column name (list all the column names with column_names).0 `list` List of unique elements in the given column.
Return a list of the unique elements in a column.

This is implemented in the low-level backend and as such, very fast.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.unique('label')
[1, 0]
```

flattendatasets.Dataset.flattenhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2016[{"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}, {"name": "max_depth", "val": " = 16"}]- **new_fingerprint** (`str` , *optional*) --
The new fingerprint of the dataset after transform.
If `None` , the new fingerprint is computed using a hash of the previous fingerprint, and the transform arguments.0DatasetA copy of the dataset with flattened columns.
Flatten the table.
Each column with a struct type is flattened into one column per struct field.
Other columns are left unchanged.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("rajpurkar/squad", split="train")
>>> ds.features
{'id': Value('string'),
 'title': Value('string'),
 'context': Value('string'),
 'question': Value('string'),
 'answers': {'text': List(Value('string')),
 'answer_start': List(Value('int32'))}}
>>> ds = ds.flatten()
>>> ds
Dataset({
    features: ['id', 'title', 'context', 'question', 'answers.text', 'answers.answer_start'],
    num_rows: 87599
})
```

castdatasets.Dataset.casthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2063[{"name": "features", "val": ": Features"}, {"name": "batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "load_from_cache_file", "val": ": typing.Optional[bool] = None"}, {"name": "cache_file_name",

"val": ": typing.Optional[str] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}]- **features** (Features) -- New features to cast the dataset to.
The name of the fields in the features must match the current column names.
The type of the data must also be convertible from one type to the other.
For non-trivial conversion, e.g. `str` <-> `ClassLabel` you should use map() to update the Dataset.

- **batch_size** (`int`, defaults to `1000`) --
  Number of examples per batch provided to cast.
  If `batch_size <= 0` or `batch_size == None` then provide the full dataset as a single batch to cast.
- **keep_in_memory** (`bool`, defaults to `False`) --
  Whether to copy the data in-memory.
- **load_from_cache_file** (`bool`, defaults to `True` if caching is enabled) --
  If a cache file storing the current computation from `function`
  can be identified, use it instead of recomputing.
- **cache_file_name** (`str`, *optional*, defaults to `None`) --
  Provide the name of a path for the cache file. It is used to store the
  results of the computation instead of the automatically generated cache file name.
- **writer_batch_size** (`int`, defaults to `1000`) --
  Number of rows per write operation for the cache file writer.
  This value is a good trade-off between memory usage during the processing, and processing speed.
  Higher value makes the processing do fewer lookups, lower value consume less temporary memory while running map().
- **num_proc** (`int`, *optional*, defaults to `None`) --
  Number of processes for multiprocessing. By default it doesn't
  use multiprocessing.0DatasetA copy of the dataset with casted features.

Cast the dataset to a new set of features.

Example:

```
>>> from datasets import load_dataset, ClassLabel, Value
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.features
{'label': ClassLabel(names=['neg', 'pos']),
 'text': Value('string')}
>>> new_features = ds.features.copy()
>>> new_features['label'] = ClassLabel(names=['bad', 'good'])
>>> new_features['text'] = Value('large_string')
>>> ds = ds.cast(new_features)
>>> ds.features
{'label': ClassLabel(names=['bad', 'good']),
 'text': Value('large_string')}
```

cast_columndatasets.Dataset.cast_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2147[{"name": "column", "val": ": str"}, {"name": "feature", "val": ": typing.Union[dict, list, tuple, datasets.features.features.Value, datasets.features.features.ClassLabel, datasets.features.translation.Translation, datasets.features.translation.TranslationVariableLanguages, datasets.features.features.LargeList, datasets.features.features.List, datasets.features.features.Array2D, datasets.features.features.Array3D, datasets.features.features.Array4D, datasets.features.features.Array5D, datasets.features.audio.Audio, datasets.features.image.Image, datasets.features.video.Video, datasets.features.pdf.Pdf]"}, {"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}]-

column (`str`) --
Column name.

- **feature** (`FeatureType`) --
  Target feature.
- **new_fingerprint** (`str` , *optional*) --
  The new fingerprint of the dataset after transform.
  If `None` , the new fingerprint is computed using a hash of the previous fingerprint, and the transform arguments.0Dataset
  Cast column to feature for decoding.

Example:

```
>>> from datasets import load_dataset, ClassLabel
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.features
{'label': ClassLabel(names=['neg', 'pos']),
 'text': Value('string')}
>>> ds = ds.cast_column('label', ClassLabel(names=['bad', 'good']))
>>> ds.features
{'label': ClassLabel(names=['bad', 'good']),
 'text': Value('string')}
```

remove_columnsdatasets.Dataset.remove_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2190[{"name": "column_names", "val": ": typing.Union[str, list[str]]"}, {"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}]- **column_names** (`Union[str, List[str]]`) --
Name of the column(s) to remove.

- **new_fingerprint** (`str` , *optional*) --
  The new fingerprint of the dataset after transform.
  If `None` , the new fingerprint is computed using a hash of the previous fingerprint, and the transform arguments.0DatasetA copy of the dataset object without the columns to remove.

Remove one or several column(s) in the dataset and the features associated to them.

You can also remove a column using map() with `remove_columns` but the present method doesn't copy the data of the remaining columns and is thus faster.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds = ds.remove_columns('label')
Dataset({
    features: ['text'],
    num_rows: 1066
})
>>> ds = ds.remove_columns(column_names=ds.column_names) # Removing all the columns returns an emp
Dataset({
    features: [],
    num_rows: 0
})
```

rename_columndatasets.Dataset.rename_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2245[{"name": "original_column_name", "val": ": str"}, {"name": "new_column_name", "val": ": str"}, {"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}]- **original_column_name** (`str`) --
Name of the column to rename.

- **new_column_name** (`str`) --
  New name for the column.
- **new_fingerprint** (`str` , *optional*) --
  The new fingerprint of the dataset after transform.
  If `None` , the new fingerprint is computed using a hash of the previous fingerprint, and the transform arguments.0DatasetA copy of the dataset with a renamed column.

Rename a column in the dataset, and move the features associated to the original column under the new column
name.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds = ds.rename_column('label', 'label_new')
Dataset({
    features: ['text', 'label_new'],
    num_rows: 1066
})
```

rename_columnsdatasets.Dataset.rename_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2311[{"name": "column_mapping", "val": ": dict"}, {"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}]- **column_mapping** (`Dict[str, str]`) --
A mapping of columns to rename to their new names

- **new_fingerprint** (`str` , *optional*) --
  The new fingerprint of the dataset after transform.
  If `None` , the new fingerprint is computed using a hash of the previous fingerprint, and the transform arguments.0DatasetA copy of the dataset with renamed columns

Rename several columns in the dataset, and move the features associated to the original columns under
the new column names.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds = ds.rename_columns({'text': 'text_new', 'label': 'label_new'})
Dataset({
    features: ['text_new', 'label_new'],
    num_rows: 1066
})
```

select_columnsdatasets.Dataset.select_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2378[{"name": "column_names", "val": ": typing.Union[str, list[str]]"}, {"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}]- **column_names** (`Union[str, List[str]]`) --

Name of the column(s) to keep.

- **new_fingerprint** (`str` , *optional*) --
  The new fingerprint of the dataset after transform. If `None` ,
  the new fingerprint is computed using a hash of the previous
  fingerprint, and the transform arguments.0DatasetA copy of the dataset object which only consists of
  selected columns.
  Select one or several column(s) in the dataset and the features
  associated to them.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds = ds.select_columns(['text'])
>>> ds
Dataset({
    features: ['text'],
    num_rows: 1066
})
```

class_encode_columndatasets.Dataset.class_encode_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1941[{"name": "column", "val": ": str"}, {"name": "include_nulls", "val": ": bool = False"}]- **column** (`str`) --
The name of the column to cast (list all the column names with column_names)

- **include_nulls** (`bool` , defaults to `False`) --
  Whether to include null values in the class labels. If `True` , the null values will be encoded as the `"None"` class label.
  0
  Casts the given column as ClassLabel and updates the table.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("boolq", split="validation")
>>> ds.features
{'answer': Value('bool'),
 'passage': Value('string'),
 'question': Value('string')}
>>> ds = ds.class_encode_column('answer')
>>> ds.features
{'answer': ClassLabel(num_classes=2, names=['False', 'True']),
 'passage': Value('string'),
 'question': Value('string')}
```

lendatasets.Dataset.**len**https://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2434[]
Number of rows in the dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.__len__
<bound method Dataset.__len__ of Dataset({
    features: ['text', 'label'],
    num_rows: 1066
})>
```

iterdatasets.Dataset.**iter**https://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2451[]
Iterate through the examples.

If a formatting is set with Dataset.set_format() rows will be returned with the selected format.

iterdatasets.Dataset.iterhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2480[{"name": "batch_size", "val": ": int"}, {"name": "drop_last_batch", "val": ": bool = False"}]- **batch_size** (`int`) -- size of each batch to yield.

- **drop_last_batch** (`bool` , default *False*) -- Whether a last batch smaller than the

batch_size should be
dropped0
Iterate through the batches of size *batch_size*.

If a formatting is set with [*~datasets.Dataset.set_format*] rows will be returned with the selected format.

formatted_asdatasets.Dataset.formatted_ashttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2524[{"name": "type", "val": ": typing.Optional[str] = None"}, {"name": "columns", "val": ": typing.Optional[list] = None"}, {"name": "output_all_columns", "val": ": bool = False"}, {"name": "**format_kwargs", "val": ""}]- **type** (`str` , *optional*) --
Either output type selected in `[None, 'numpy', 'torch', 'tensorflow', 'jax', 'arrow', 'pandas', 'polars']` . `None` means `**getitem**`` returns python objects (default).

- **columns** (`List[str]` , *optional*) --
  Columns to format in the output.
  `None` means `__getitem__` returns all columns (default).
- **output_all_columns** (`bool` , defaults to `False`) --
  Keep un-formatted columns as well in the output (as python objects).
- ****format_kwargs** (additional keyword arguments) --
  Keywords arguments passed to the convert function like `np.array` , `torch.tensor` or `tensorflow.ragged.constant` .0
  To be used in a `with` statement. Set `__getitem__` return format (type and columns).

set_formatdatasets.Dataset.set_formathttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2556[{"name": "type", "val": ": typing.Optional[str] = None"}, {"name": "columns", "val": ": typing.Optional[list] = None"}, {"name": "output_all_columns", "val": ": bool = False"}, {"name": "**format_kwargs", "val": ""}]- **type** (`str` , *optional*) --
Either output type selected in `[None, 'numpy', 'torch', 'tensorflow', 'jax', 'arrow', 'pandas', 'polars']` . `None` means `__getitem__` returns python objects (default).

- **columns** (`List[str]` , *optional*) --
  Columns to format in the output.
  `None` means `__getitem__` returns all columns (default).

- **output_all_columns** (`bool` , defaults to `False`) --
  Keep un-formatted columns as well in the output (as python objects).
- ****format_kwargs** (additional keyword arguments) --
  Keywords arguments passed to the convert function like `np.array` , `torch.tensor` or `tensorflow.ragged.constant` .0
  Set `__getitem__` return format (type and columns). The data formatting is applied on-the-fly.
  The format `type` (for example "numpy") is used to format batches when using `__getitem__` .
  It's also possible to use custom transforms for formatting using set_transform().

It is possible to call map() after calling `set_format` . Since `map` may add new columns, then the list of formatted columns

gets updated. In this case, if you apply `map` on a dataset to add a new column, then this column will be formatted as:

```
new formatted columns = (all columns - previously unformatted columns)
```

Example:

```
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
>>> ds = ds.map(lambda x: tokenizer(x['text'], truncation=True, padding=True), batched=True)
>>> ds.set_format(type='numpy', columns=['text', 'label'])
>>> ds.format
{'type': 'numpy',
'format_kwargs': {},
'columns': ['text', 'label'],
'output_all_columns': False}
```

set_transformdatasets.Dataset.set_transformhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2664[{"name": "transform", "val": ": typing.Optional[typing.Callable]"}, {"name": "columns", "val": ": typing.Optional[list] = None"},

{"name": "output_all_columns", "val": ": bool = False"}]- **transform** (`Callable` , *optional*) --
User-defined formatting transform, replaces the format defined by set_format().
A formatting function is a callable that takes a batch (as a `dict`) as input and returns a batch.
This function is applied right before returning the objects in `__getitem__` .

- **columns** (`List[str]` , *optional*) --
  Columns to format in the output.
  If specified, then the input batch of the transform only contains those columns.
- **output_all_columns** (`bool` , defaults to `False`) --
  Keep un-formatted columns as well in the output (as python objects).
  If set to True, then the other un-formatted columns are kept with the output of the transform.0
  Set `__getitem__` return format using this transform. The transform is applied on-the-fly on batches when `__getitem__` is called.
  As set_format(), this can be reset using reset_format().

Example:

```
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
>>> def encode(batch):
...     return tokenizer(batch['text'], padding=True, truncation=True, return_tensors='pt')
>>> ds.set_transform(encode)
>>> ds[0]
{'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1]),
 'input_ids': tensor([  101, 29353,  2135, 15102,  1996,  9428, 20868,  2890,  8663,  6895,
         20470,  2571,  3663,  2090,  4603,  3017,  3008,  1998,  2037, 24211,
         5637,  1998, 11690,  2336,  1012,   102]),
 'token_type_ids': tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0])}
```

reset_formatdatasets.Dataset.reset_formathttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2635[]
Reset `__getitem__` return format to python objects and all columns.

Same as `self.set_format()`

Example:

```
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
>>> ds = ds.map(lambda x: tokenizer(x['text'], truncation=True, padding=True), batched=True)
>>> ds.set_format(type='numpy', columns=['input_ids', 'token_type_ids', 'attention_mask', 'label'])
>>> ds.format
{'columns': ['input_ids', 'token_type_ids', 'attention_mask', 'label'],
 'format_kwargs': {},
 'output_all_columns': False,
 'type': 'numpy'}
>>> ds.reset_format()
>>> ds.format
{'columns': ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
 'format_kwargs': {},
 'output_all_columns': False,
 'type': None}
```

with_formatdatasets.Dataset.with_formathttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2707[{"name": "type", "val": ": typing.Optional[str] = None"}, {"name": "columns", "val": ": typing.Optional[list] = None"}, {"name": "output_all_columns", "val": ": bool = False"}, {"name": "**format_kwargs", "val": ""}]- **type** (`str`, *optional*) -- Either output type selected in `[None, 'numpy', 'torch', 'tensorflow', 'jax', 'arrow', 'pandas', 'polars']`. `None` means `__getitem__` returns python objects (default).

- **columns** (`List[str]`, *optional*) --
  Columns to format in the output.
  `None` means `__getitem__` returns all columns (default).
- **output_all_columns** (`bool`, defaults to `False`) --
  Keep un-formatted columns as well in the output (as python objects).
- ****format_kwargs** (additional keyword arguments) --
  Keywords arguments passed to the convert function like `np.array`, `torch.tensor` or `tensorflow.ragged.constant`.0

Set `__getitem__` return format (type and columns). The data formatting is applied on-the-fly.
The format `type` (for example "numpy") is used to format batches when using `__getitem__`.

It's also possible to use custom transforms for formatting using with_transform().

Contrary to set_format(), `with_format` returns a new Dataset object.

Example:

```
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
>>> ds = ds.map(lambda x: tokenizer(x['text'], truncation=True, padding=True), batched=True)
>>> ds.format
{'columns': ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
 'format_kwargs': {},
 'output_all_columns': False,
 'type': None}
>>> ds = ds.with_format("torch")
>>> ds.format
{'columns': ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
 'format_kwargs': {},
 'output_all_columns': False,
 'type': 'torch'}
>>> ds[0]
{'text': 'compassionately explores the seemingly irreconcilable situation between conservative chr
 'label': tensor(1),
 'input_ids': tensor([  101, 18027, 16310, 16001,  1103,  9321,   178, 11604,  7235,  6617,
         1742,  2165,  2820,  1206,  6588, 22572, 12937,  1811,  2153,  1105,
         1147, 12890, 19587,  6463,  1105, 15026,  1482,   119,   102,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0]),
 'token_type_ids': tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])}
```

with_transformdatasets.Dataset.with_transformhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2778[{"name": "transform", "val": ":

typing.Optional[typing.Callable]"}, {"name": "columns", "val": ": typing.Optional[list] = None"}, {"name": "output_all_columns", "val": ": bool = False"}]- **transform** (`Callable` , `optional`) -- User-defined formatting transform, replaces the format defined by set_format().
A formatting function is a callable that takes a batch (as a `dict`) as input and returns a batch. This function is applied right before returning the objects in `__getitem__` .

- **columns** (`List[str]` , `optional`) --
  Columns to format in the output.
  If specified, then the input batch of the transform only contains those columns.
- **output_all_columns** (`bool` , defaults to `False`) --
  Keep un-formatted columns as well in the output (as python objects).
  If set to `True` , then the other un-formatted columns are kept with the output of the transform.0
  Set `__getitem__` return format using this transform. The transform is applied on-the-fly on batches when `__getitem__` is called.

As set_format(), this can be reset using reset_format().

Contrary to set_transform(), `with_transform` returns a new Dataset object.

Example:

```
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
>>> def encode(example):
...     return tokenizer(example["text"], padding=True, truncation=True, return_tensors='pt')
>>> ds = ds.with_transform(encode)
>>> ds[0]
{'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1]),
 'input_ids': tensor([  101, 18027, 16310, 16001,  1103,  9321,   178, 11604,  7235,  6617,
         1742,  2165,  2820,  1206,  6588, 22572, 12937,  1811,  2153,  1105,
         1147, 12890, 19587,  6463,  1105, 15026,  1482,   119,   102]),
 'token_type_ids': tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0])}
```

getitemdatasets.Dataset.**getitem**https://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2854[{"name": "key", "val": ""}]
Can be used to index columns (by string names) or rows (by integer index or iterable of indices or bools).

cleanup_cache_filesdatasets.Dataset.cleanup_cache_fileshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2867[] `int` Number of removed files.
Clean up all cache files in the dataset cache directory, excepted the currently used cache file if there is
one.

Be careful when running this command that no other process is currently using other cache files.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds.cleanup_cache_files()
10
```

mapdatasets.Dataset.maphttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L2914[{"name": "function", "val": ": typing.Optional[typing.Callable] = None"}, {"name": "with_indices", "val": ": bool = False"}, {"name": "with_rank", "val": ": bool = False"}, {"name": "input_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "batched", "val": ": bool = False"}, {"name": "batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "drop_last_batch", "val": ": bool = False"}, {"name": "remove_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "load_from_cache_file", "val": ": typing.Optional[bool] = None"}, {"name": "cache_file_name", "val": ": typing.Optional[str] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "disable_nullable", "val": ": bool = False"}, {"name": "fn_kwargs", "val": ": typing.Optional[dict] = None"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "suffix_template", "val": ": str = '_{rank:05d}*of*{num_proc:05d}'"}, {"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}, {"name": "desc", "val": ": typing.Optional[str] = None"}, {"name": "try_original_type",

"val": ": typing.Optional[bool] = True"}]- **function** (`Callable`) -- Function with one of the following signatures:

- `function(example: Dict[str, Any]) -> Dict[str, Any]` if `batched=False` and `with_indices=False` and `with_rank=False`
- `function(example: Dict[str, Any], *extra_args) -> Dict[str, Any]` if `batched=False` and `with_indices=True` and/or `with_rank=True` (one extra arg for each)
- `function(batch: Dict[str, List]) -> Dict[str, List]` if `batched=True` and `with_indices=False` and `with_rank=False`
- `function(batch: Dict[str, List], *extra_args) -> Dict[str, List]` if `batched=True` and `with_indices=True` and/or `with_rank=True` (one extra arg for each)

For advanced usage, the function can also return a `pyarrow.Table` .
If the function is asynchronous, then `map` will run your function in parallel.
Moreover if your function returns nothing (`None`), then `map` will run your function and return the dataset unchanged.
If no function is provided, default to identity function: `lambda x: x` .

- **with_indices** (`bool` , defaults to `False`) --
  Provide example indices to `function` . Note that in this case the signature of `function` should be `def function(example, idx[, rank]): ...` .
- **with_rank** (`bool` , defaults to `False`) --
  Provide process rank to `function` . Note that in this case the signature of `function` should be `def function(example[, idx], rank): ...` .
- **input_columns** (`Optional[Union[str, List[str]]]` , defaults to `None`) --
  The columns to be passed into `function`
  as positional arguments. If `None` , a `dict` mapping to all formatted columns is passed as one argument.
- **batched** (`bool` , defaults to `False`) --
  Provide batch of examples to `function` .
- **batch_size** (`int` , *optional*, defaults to `1000`) --
  Number of examples per batch provided to `function` if `batched=True` .
  If `batch_size <= 0` or `batch_size == None` , provide the full dataset as a single batch to `function` .
- **drop_last_batch** (`bool` , defaults to `False`) --
  Whether a last batch smaller than the batch_size should be

dropped instead of being processed by the function.

- **remove_columns** (`Optional[Union[str, List[str]]]` , defaults to `None`) --
  Remove a selection of columns while doing the mapping.
  Columns will be removed before updating the examples with the output of `function` , i.e. if `function` is adding
  columns with names in `remove_columns` , these columns will be kept.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Keep the dataset in memory instead of writing it to a cache file.
- **load_from_cache_file** (`Optional[bool]` , defaults to `True` if caching is enabled) --
  If a cache file storing the current computation from `function`
  can be identified, use it instead of recomputing.
- **cache_file_name** (`str` , *optional*, defaults to `None`) --
  Provide the name of a path for the cache file. It is used to store the
  results of the computation instead of the automatically generated cache file name.
- **writer_batch_size** (`int` , defaults to `1000`) --
  Number of rows per write operation for the cache file writer.
  This value is a good trade-off between memory usage during the processing, and
  processing speed.
  Higher value makes the processing do fewer lookups, lower value consume less
  temporary memory while running `map` .
- **features** (`Optional[datasets.Features]` , defaults to `None`) --
  Use a specific Features to store the cache file
  instead of the automatically generated one.
- **disable_nullable** (`bool` , defaults to `False`) --
  Disallow null values in the table.
- **fn_kwargs** (`Dict` , *optional*, defaults to `None`) --
  Keyword arguments to be passed to `function` .
- **num_proc** (`int` , *optional*, defaults to `None`) --
  The number of processes to use for multiprocessing.
  - If `None` or `0` , no multiprocessing is used and the operation runs in the main process.
  - If greater than `1` , one or multiple worker processes are used to process data in parallel.
    Note: The function passed to `map()` must be picklable for multiprocessing to work correctly
    (i.e., prefer functions defined at the top level of a module, not inside another function

or class).
suffix_template (`str`):
If `cache_file_name` is specified, then this suffix
will be added at the end of the base name of each. Defaults to
`"_{rank:05d}_of_{num_proc:05d}"` . For example, if `cache_file_name` is
"processed.arrow", then for
`rank=1` and `num_proc=4` , the resulting file would be
`"processed_00001_of_00004.arrow"` for the default suffix.

- **new_fingerprint** (`str` , *optional*, defaults to `None`) --
  The new fingerprint of the dataset after transform.
  If `None` , the new fingerprint is computed using a hash of the previous fingerprint, and the transform arguments.
- **desc** (`str` , *optional*, defaults to `None`) --
  Meaningful description to be displayed alongside with the progress bar while mapping examples.
- **try_original_type** (`Optional[bool]` , defaults to `True`) --
  Try to keep the types of the original columns (e.g. int32 -> int32).
  Set to False if you want to always infer new types.0

Apply a function to all the examples in the table (individually or in batches) and update the table.
If your function returns a column that already exists, then it overwrites it.

You can specify whether the function should be batched or not with the `batched` parameter:

- If batched is `False` , then the function takes 1 example in and should return 1 example.
  An example is a dictionary, e.g. `{"text": "Hello there !"}` .
- If batched is `True` and `batch_size` is 1, then the function takes a batch of 1 example as input and can return a batch with 1 or more examples.
  A batch is a dictionary, e.g. a batch of 1 example is `{"text": ["Hello there !"]}` .
- If batched is `True` and `batch_size` is `n > 1` , then the function takes a batch of `n` examples as input and can return a batch with `n` examples, or with an arbitrary number of examples.
  Note that the last batch may have less than `n` examples.
  A batch is a dictionary, e.g. a batch of `n` examples is `{"text": ["Hello there !"] * n}` .

If the function is asynchronous, then `map` will run your function in parallel, with up to one thousand simultaneous calls.
It is recommended to use a `asyncio.Semaphore` in your function if you want to set a maximum number of operations that can run at the same time.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> def add_prefix(example):
...     example["text"] = "Review: " + example["text"]
...     return example
>>> ds = ds.map(add_prefix)
>>> ds[0:3]["text"]
['Review: compassionately explores the seemingly irreconcilable situation between conservative chr
 'Review: the soundtrack alone is worth the price of admission .',
 'Review: rodriguez does a splendid job of racial profiling hollywood style--casting excellent lat

# process a batch of examples
>>> ds = ds.map(lambda example: tokenizer(example["text"]), batched=True)
# set number of processors
>>> ds = ds.map(add_prefix, num_proc=4)
```

filterdatasets.Dataset.filterhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L3792[{"name": "function", "val": ": typing.Optional[typing.Callable] = None"}, {"name": "with_indices", "val": ": bool = False"}, {"name": "with_rank", "val": ": bool = False"}, {"name": "input_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "batched", "val": ": bool = False"}, {"name": "batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "load_from_cache_file", "val": ": typing.Optional[bool] = None"}, {"name": "cache_file_name", "val": ": typing.Optional[str] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "fn_kwargs", "val": ": typing.Optional[dict] = None"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "suffix_template", "val": ": str = '_{rank:05d}*of*{num_proc:05d}'"}, {"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}, {"name": "desc", "val": ": typing.Optional[str] = None"}]- **function** (`Callable`) -- Callable with one of the following signatures:

- `function(example: Dict[str, Any]) -> bool` if `batched=False` and `with_indices=False` and `with_rank=False`
- `function(example: Dict[str, Any], *extra_args) -> bool` if `batched=False` and `with_indices=True` and/or `with_rank=True` (one extra arg for each)
- `function(batch: Dict[str, List]) -> List[bool]` if `batched=True` and `with_indices=False` and `with_rank=False`
- `function(batch: Dict[str, List], *extra_args) -> List[bool]` if `batched=True` and `with_indices=True` and/or `with_rank=True` (one extra arg for each)

If the function is asynchronous, then `filter` will run your function in parallel.
If no function is provided, defaults to an always `True` function: `lambda x: True`.

- **with_indices** (`bool`, defaults to `False`) --
  Provide example indices to `function`. Note that in this case the signature of `function` should be `def function(example, idx[, rank]): ...`.
- **with_rank** (`bool`, defaults to `False`) --
  Provide process rank to `function`. Note that in this case the signature of `function` should be `def function(example[, idx], rank): ...`.
- **input_columns** (`str` or `List[str]`, *optional*) --
  The columns to be passed into `function` as positional arguments. If `None`, a `dict` mapping to all formatted columns is passed as one argument.
- **batched** (`bool`, defaults to `False`) --
  Provide batch of examples to `function`.
- **batch_size** (`int`, *optional*, defaults to `1000`) --
  Number of examples per batch provided to `function` if `batched = True`. If `batched = False`, one example per batch is passed to `function`.
  If `batch_size <= 0` or `batch_size == None`, provide the full dataset as a single batch to `function`.
- **keep_in_memory** (`bool`, defaults to `False`) --
  Keep the dataset in memory instead of writing it to a cache file.
- **load_from_cache_file** (`Optional[bool]`, defaults to `True` if caching is enabled) --
  If a cache file storing the current computation from `function` can be identified, use it instead of recomputing.
- **cache_file_name** (`str`, *optional*) --

Provide the name of a path for the cache file. It is used to store the
results of the computation instead of the automatically generated cache file name.

- **writer_batch_size** (`int` , defaults to `1000`) --
  Number of rows per write operation for the cache file writer.
  This value is a good trade-off between memory usage during the processing, and processing speed.
  Higher value makes the processing do fewer lookups, lower value consume less temporary memory while running `map` .
- **fn_kwargs** (`dict` , *optional*) --
  Keyword arguments to be passed to `function` .
- **num_proc** (`int` , *optional*, defaults to `None`) --
  The number of processes to use for multiprocessing.
  - If `None` or `0` , no multiprocessing is used and the operation runs in the main process.
  - If greater than `1` , one or multiple worker processes are used to process data in parallel.
    Note: The function passed to `map()` must be picklable for multiprocessing to work correctly
    (i.e., prefer functions defined at the top level of a module, not inside another function or class).
- **suffix_template** (`str`) --
  If `cache_file_name` is specified, then this suffix will be added at the end of the base name of each.
  For example, if `cache_file_name` is `"processed.arrow"` , then for `rank = 1` and `num_proc = 4` ,
  the resulting file would be `"processed_00001_of_00004.arrow"` for the default suffix (default `_{rank:05d}_of_{num_proc:05d}`).
- **new_fingerprint** (`str` , *optional*) --
  The new fingerprint of the dataset after transform.
  If `None` , the new fingerprint is computed using a hash of the previous fingerprint, and the transform arguments.
- **desc** (`str` , *optional*, defaults to `None`) --
  Meaningful description to be displayed alongside with the progress bar while filtering examples.0
  Apply a filter function to all the elements in the table in batches
  and update the table so that the dataset only includes examples according to the filter

function.

If the function is asynchronous, then `filter` will run your function in parallel, with up to one thousand simultaneous calls (configurable).
It is recommended to use a `asyncio.Semaphore` in your function if you want to set a maximum number of operations that can run at the same time.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds = ds.filter(lambda x: x["label"] == 1)
>>> ds
Dataset({
    features: ['text', 'label'],
    num_rows: 533
})
```

selectdatasets.Dataset.selecthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L4019[{"name": "indices", "val": ": Iterable"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "indices_cache_file_name", "val": ": typing.Optional[str] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}]- **indices** (`range` , `list` , `iterable` , `ndarray` or `Series`) --
Range, list or 1D-array of integer indices for indexing.
If the indices correspond to a contiguous range, the Arrow table is simply sliced.
However passing a list of indices that are not contiguous creates indices mapping, which is much less efficient,
but still faster than recreating an Arrow table made of the requested rows.

- **keep_in_memory** (`bool` , defaults to `False`) --
  Keep the indices mapping in memory instead of writing it to a cache file.
- **indices_cache_file_name** (`str` , *optional*, defaults to `None`) --
  Provide the name of a path for the cache file. It is used to store the
  indices mapping instead of the automatically generated cache file name.
- **writer_batch_size** (`int` , defaults to `1000`) --
  Number of rows per write operation for the cache file writer.

This value is a good trade-off between memory usage during the processing, and processing speed.
Higher value makes the processing do fewer lookups, lower value consume less temporary memory while running `map`.

- **new_fingerprint** (`str`, *optional*, defaults to `None`) --
  The new fingerprint of the dataset after transform.
  If `None`, the new fingerprint is computed using a hash of the previous fingerprint, and the transform arguments.0
  Create a new dataset with rows selected following the list/array of indices.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds = ds.select(range(4))
>>> ds
Dataset({
    features: ['text', 'label'],
    num_rows: 4
})
```

sortdatasets.Dataset.sorthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L4357[{"name": "column_names", "val": ": typing.Union[str, collections.abc.Sequence[str]]"}, {"name": "reverse", "val": ": typing.Union[bool, collections.abc.Sequence[bool]] = False"}, {"name": "null_placement", "val": ": str = 'at_end'"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "load_from_cache_file", "val": ": typing.Optional[bool] = None"}, {"name": "indices_cache_file_name", "val": ": typing.Optional[str] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}]- **column_names** (`Union[str, Sequence[str]]`) --
Column name(s) to sort by.

- **reverse** (`Union[bool, Sequence[bool]]`, defaults to `False`) --
  If `True`, sort by descending order rather than ascending. If a single bool is provided, the value is applied to the sorting of all column names. Otherwise a list of bools with the same length and order as column_names must be provided.
- **null_placement** (`str`, defaults to `at_end`) --

Put `None` values at the beginning if `at_start` or `first` or at the end if `at_end` or `last`

- **keep_in_memory** (`bool` , defaults to `False`) --
  Keep the sorted indices in memory instead of writing it to a cache file.
- **load_from_cache_file** (`Optional[bool]` , defaults to `True` if caching is enabled) --
  If a cache file storing the sorted indices
  can be identified, use it instead of recomputing.
- **indices_cache_file_name** (`str` , *optional*, defaults to `None`) --
  Provide the name of a path for the cache file. It is used to store the
  sorted indices instead of the automatically generated cache file name.
- **writer_batch_size** (`int` , defaults to `1000`) --
  Number of rows per write operation for the cache file writer.
  Higher value gives smaller cache files, lower value consume less temporary memory.
- **new_fingerprint** (`str` , *optional*, defaults to `None`) --
  The new fingerprint of the dataset after transform.
  If `None` , the new fingerprint is computed using a hash of the previous fingerprint, and the
  transform arguments0
  Create a new dataset sorted according to a single or multiple columns.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset('cornell-movie-review-data/rotten_tomatoes', split='validation')
>>> ds['label'][:10]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> sorted_ds = ds.sort('label')
>>> sorted_ds['label'][:10]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> another_sorted_ds = ds.sort(['label', 'text'], reverse=[True, False])
>>> another_sorted_ds['label'][:10]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

shuffledatasets.Dataset.shufflehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L4485[{"name": "seed", "val": ": typing.Optional[int] = None"}, {"name": "generator", "val": ": typing.Optional[numpy.random._generator.Generator] = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "load_from_cache_file", "val": ": typing.Optional[bool] = None"}, {"name": "indices_cache_file_name", "val": ":

typing.Optional[str] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}]- **seed** (`int` , *optional*) --
A seed to initialize the default BitGenerator if `generator=None` .
If `None` , then fresh, unpredictable entropy will be pulled from the OS.
If an `int` or `array_like[ints]` is passed, then it will be passed to SeedSequence to derive the initial BitGenerator state.

- **generator** (`numpy.random.Generator` , *optional*) --
  Numpy random Generator to use to compute the permutation of the dataset rows.
  If `generator=None` (default), uses `np.random.default_rng` (the default BitGenerator (PCG64) of NumPy).
- **keep_in_memory** (`bool` , default `False`) --
  Keep the shuffled indices in memory instead of writing it to a cache file.
- **load_from_cache_file** (`Optional[bool]` , defaults to `True` if caching is enabled) --
  If a cache file storing the shuffled indices
  can be identified, use it instead of recomputing.
- **indices_cache_file_name** (`str` , *optional*) --
  Provide the name of a path for the cache file. It is used to store the
  shuffled indices instead of the automatically generated cache file name.
- **writer_batch_size** (`int` , defaults to `1000`) --
  Number of rows per write operation for the cache file writer.
  This value is a good trade-off between memory usage during the processing, and processing speed.
  Higher value makes the processing do fewer lookups, lower value consume less temporary memory while running `map` .
- **new_fingerprint** (`str` , *optional*, defaults to `None`) --
  The new fingerprint of the dataset after transform.
  If `None` , the new fingerprint is computed using a hash of the previous fingerprint, and the transform arguments.0
  Create a new Dataset where the rows are shuffled.

Currently shuffling uses numpy random generators.
You can either supply a NumPy BitGenerator to use, or a seed to initiate NumPy's default random generator (PCG64).

Shuffling takes the list of indices `[0:len(my_dataset)]` and shuffles it to create an indices mapping.
However as soon as your Dataset has an indices mapping, the speed can become 10x slower. This is because there is an extra step to get the row index to read using the indices mapping, and most importantly, you aren't reading contiguous chunks of data anymore.
To restore the speed, you'd need to rewrite the entire dataset on your disk again using Dataset.flatten_indices(), which removes the indices mapping.

This may take a lot of time depending of the size of your dataset though:

```
my_dataset[0]  # fast
my_dataset = my_dataset.shuffle(seed=42)
my_dataset[0]  # up to 10x slower
my_dataset = my_dataset.flatten_indices()  # rewrite the shuffled dataset on disk as contiguous ch
my_dataset[0]  # fast again
```

In this case, we recommend switching to an IterableDataset and leveraging its fast approximate shuffling method IterableDataset.shuffle().

It only shuffles the shards order and adds a shuffle buffer to your dataset, which keeps the speed of your dataset optimal:

```
my_iterable_dataset = my_dataset.to_iterable_dataset(num_shards=128)
for example in enumerate(my_iterable_dataset):  # fast
    pass

shuffled_iterable_dataset = my_iterable_dataset.shuffle(seed=42, buffer_size=100)

for example in enumerate(shuffled_iterable_dataset):  # as fast as before
    pass
```

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds['label'][:10]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

# set a seed
>>> shuffled_ds = ds.shuffle(seed=42)
>>> shuffled_ds['label'][:10]
[1, 0, 1, 1, 0, 0, 0, 0, 0, 0]
```

skipdatasets.Dataset.skiphttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L4272[{"name": "n", "val": ": int"}]- **n** (`int`) --
Number of elements to skip.0

Create a new Dataset that skips the first `n` elements.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train")
>>> list(ds.take(3))
[{'label': 1,
 'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
 {'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
 {'label': 1, 'text': 'effective but too-tepid biopic'}]
>>> ds = ds.skip(1)
>>> list(ds.take(3))
[{'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
 {'label': 1, 'text': 'effective but too-tepid biopic'},
 {'label': 1,
 'text': 'if you sometimes like to go to the movies to have fun , wasabi is a good place to start
```

takedatasets.Dataset.takehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L4334[{"name": "n", "val": ": int"}]- **n** (`int`) --
Number of elements to take.0

Create a new Dataset with only the first `n` elements.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train")
>>> small_ds = ds.take(2)
>>> list(small_ds)
[{'label': 1,
 'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
 {'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
```

train_test_splitdatasets.Dataset.train_test_splithttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L4617[{"name": "test_size", "val": ": typing.Union[float, int, NoneType] = None"}, {"name": "train_size", "val": ": typing.Union[float, int, NoneType] = None"}, {"name": "shuffle", "val": ": bool = True"}, {"name": "stratify_by_column", "val": ": typing.Optional[str] = None"}, {"name": "seed", "val": ": typing.Optional[int] = None"}, {"name": "generator", "val": ": typing.Optional[numpy.random._generator.Generator] = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "load_from_cache_file", "val": ": typing.Optional[bool] = None"}, {"name": "train_indices_cache_file_name", "val": ": typing.Optional[str] = None"}, {"name": "test_indices_cache_file_name", "val": ": typing.Optional[str] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "train_new_fingerprint", "val": ": typing.Optional[str] = None"}, {"name": "test_new_fingerprint", "val": ": typing.Optional[str] = None"}]- **test_size**
(`Union[float, int, None]`, *optional*) --
Size of the test split
If `float`, should be between `0.0` and `1.0` and represent the proportion of the dataset to include in the test split.
If `int`, represents the absolute number of test samples.
If `None`, the value is set to the complement of the train size.
If `train_size` is also `None`, it will be set to `0.25`.

- **train_size** (`Union[float, int, None]`, *optional*) --
  Size of the train split
  If `float`, should be between `0.0` and `1.0` and represent the proportion of the dataset to include in the train split.

If `int`, represents the absolute number of train samples.
If `None`, the value is automatically set to the complement of the test size.

- **shuffle** (`bool`, *optional*, defaults to `True`) --
  Whether or not to shuffle the data before splitting.
- **stratify_by_column** (`str`, *optional*, defaults to `None`) --
  The column name of labels to be used to perform stratified split of data.
- **seed** (`int`, *optional*) --
  A seed to initialize the default BitGenerator if `generator=None`.
  If `None`, then fresh, unpredictable entropy will be pulled from the OS.
  If an `int` or `array_like[ints]` is passed, then it will be passed to SeedSequence to derive the initial BitGenerator state.
- **generator** (`numpy.random.Generator`, *optional*) --
  Numpy random Generator to use to compute the permutation of the dataset rows.
  If `generator=None` (default), uses `np.random.default_rng` (the default BitGenerator (PCG64) of NumPy).
- **keep_in_memory** (`bool`, defaults to `False`) --
  Keep the splits indices in memory instead of writing it to a cache file.
- **load_from_cache_file** (`Optional[bool]`, defaults to `True` if caching is enabled) --
  If a cache file storing the splits indices
  can be identified, use it instead of recomputing.
- **train_cache_file_name** (`str`, *optional*) --
  Provide the name of a path for the cache file. It is used to store the
  train split indices instead of the automatically generated cache file name.
- **test_cache_file_name** (`str`, *optional*) --
  Provide the name of a path for the cache file. It is used to store the
  test split indices instead of the automatically generated cache file name.
- **writer_batch_size** (`int`, defaults to `1000`) --
  Number of rows per write operation for the cache file writer.
  This value is a good trade-off between memory usage during the processing, and processing speed.
  Higher value makes the processing do fewer lookups, lower value consume less temporary memory while running `map`.
- **train_new_fingerprint** (`str`, *optional*, defaults to `None`) --
  The new fingerprint of the train set after transform.
  If `None`, the new fingerprint is computed using a hash of the previous fingerprint, and the

transform arguments

- **test_new_fingerprint** (`str` , *optional*, defaults to `None`) --
  The new fingerprint of the test set after transform.
  If `None` , the new fingerprint is computed using a hash of the previous fingerprint, and the transform arguments0
  Return a dictionary (datasets.DatasetDict) with two random train and test subsets (`train` and `test` `Dataset` splits).
  Splits are created from the dataset according to `test_size` , `train_size` and `shuffle` .

This method is similar to scikit-learn `train_test_split` .

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds = ds.train_test_split(test_size=0.2, shuffle=True)
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 852
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 214
    })
})

# set a seed
>>> ds = ds.train_test_split(test_size=0.2, seed=42)

# stratified split
>>> ds = load_dataset("imdb",split="train")
Dataset({
    features: ['text', 'label'],
    num_rows: 25000
})
>>> ds = ds.train_test_split(test_size=0.2, stratify_by_column="label")
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 20000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 5000
    })
})
```

sharddatasets.Dataset.shardhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L4900[{"name": "num_shards", "val": ": int"}, {"name": "index", "val": ": int"}, {"name": "contiguous", "val": ": bool = True"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "indices_cache_file_name", "val": ": typing.Optional[str] = None"}, {"name":

"writer_batch_size", "val": ": typing.Optional[int] = 1000"}]- **num_shards** (`int`) --
How many shards to split the dataset into.

- **index** (`int`) --
  Which shard to select and return.
- **contiguous** -- (`bool` , defaults to `True`):
  Whether to select contiguous blocks of indices for shards.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Keep the dataset in memory instead of writing it to a cache file.
- **indices_cache_file_name** (`str` , *optional*) --
  Provide the name of a path for the cache file. It is used to store the
  indices of each shard instead of the automatically generated cache file name.
- **writer_batch_size** (`int` , defaults to `1000`) --
  This only concerns the indices mapping.
  Number of indices per write operation for the cache file writer.
  This value is a good trade-off between memory usage during the processing, and
  processing speed.
  Higher value makes the processing do fewer lookups, lower value consume less
  temporary memory while running `map` .0
  Return the `index` -nth shard from dataset split into `num_shards` pieces.

This shards deterministically. `dataset.shard(n, i)` splits the dataset into contiguous chunks, so it can be easily concatenated back together after processing. If `len(dataset) % n == l` , then the
first `l` dataset each have length `(len(dataset) // n) + 1` , and the remaining dataset have length `(len(dataset) // n)` .
`datasets.concatenate_datasets([dset.shard(n, i) for i in range(n)])` returns a dataset with the same order as the original.

Note: n should be less or equal to the number of elements in the dataset `len(dataset)` .

On the other hand, `dataset.shard(n, i, contiguous=False)` contains all elements of the dataset whose index mod `n = i` .

Be sure to shard before using any randomizing operator (such as `shuffle`).
It is best if the shard operator is used early in the dataset pipeline.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation")
>>> ds
Dataset({
    features: ['text', 'label'],
    num_rows: 1066
})
>>> ds = ds.shard(num_shards=2, index=0)
>>> ds
Dataset({
    features: ['text', 'label'],
    num_rows: 533
})
```

repeatdatasets.Dataset.repeathttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L4302[{"name": "num_times", "val": ": int"}]- **num_times** (`int`) -- Number of times to repeat the dataset.0

Create a new Dataset that repeats the underlying dataset `num_times` times.

Like itertools.repeat, repeating once just returns the full dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train")
>>> ds = ds.take(2).repeat(2)
>>> list(ds)
[{'label': 1,
 'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
 {'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
 {'label': 1, 'text': 'effective but too-tepid biopic'},
 {'label': 1,
 'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
 {'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
 {'label': 1, 'text': 'effective but too-tepid biopic'}]
```

to_tf_datasetdatasets.Dataset.to_tf_datasethttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L331[{"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "shuffle", "val": ": bool = False"}, {"name": "collate_fn", "val": ": typing.Optional[typing.Callable] = None"}, {"name": "drop_remainder", "val": ": bool = False"}, {"name": "collate_fn_args", "val": ": typing.Optional[dict[str, typing.Any]] = None"}, {"name": "label_cols", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "prefetch", "val": ": bool = True"}, {"name": "num_workers", "val": ": int = 0"}, {"name": "num_test_batches", "val": ": int = 20"}]- **batch_size** (`int`, *optional*) --
Size of batches to load from the dataset. Defaults to `None`, which implies that the dataset won't be
batched, but the returned dataset can be batched later with `tf_dataset.batch(batch_size)`.

- **columns** (`List[str]` or `str`, *optional*) --
  Dataset column(s) to load in the `tf.data.Dataset`.
  Column names that are created by the `collate_fn` and that do not exist in the original dataset can be used.
- **shuffle(`bool`,** defaults to `False`) --
  Shuffle the dataset order when loading. Recommended `True` for training, `False` for validation/evaluation.
- **drop_remainder(`bool`,** defaults to `False`) --
  Drop the last incomplete batch when loading. Ensures

that all batches yielded by the dataset will have the same length on the batch dimension.

- **collate_fn(`Callable`,** *optional*) --
  A function or callable object (such as a `DataCollator`) that will collate
  lists of samples into a batch.
- **collate_fn_args** (`Dict`, *optional*) --
  An optional `dict` of keyword arguments to be passed to the
  `collate_fn`.
- **label_cols** (`List[str]` or `str`, defaults to `None`) --
  Dataset column(s) to load as labels.
  Note that many models compute loss internally rather than letting Keras do it, in which case
  passing the labels here is optional, as long as they're in the input `columns`.
- **prefetch** (`bool`, defaults to `True`) --
  Whether to run the dataloader in a separate thread and maintain
  a small buffer of batches for training. Improves performance by allowing data to be loaded in the
  background while the model is training.
- **num_workers** (`int`, defaults to `0`) --
  Number of workers to use for loading the dataset.
- **num_test_batches** (`int`, defaults to `20`) --
  Number of batches to use to infer the output signature of the dataset.
  The higher this number, the more accurate the signature will be, but the longer it will take to
  create the dataset.0`tf.data.Dataset`
  Create a `tf.data.Dataset` from the underlying Dataset. This `tf.data.Dataset` will load and collate batches from
  the Dataset, and is suitable for passing to methods like `model.fit()` or `model.predict()`.
  The dataset will yield
  `dicts` for both inputs and labels unless the `dict` would contain only a single key, in which case a raw
  `tf.Tensor` is yielded instead.

Example:

```
>>> ds_train = ds["train"].to_tf_dataset(
...     columns=['input_ids', 'token_type_ids', 'attention_mask', 'label'],
...     shuffle=True,
...     batch_size=16,
...     collate_fn=data_collator,
... )
```

push_to_hubdatasets.Dataset.push_to_hubhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L5641[{"name": "repo_id", "val": ": str"}, {"name": "config_name", "val": ": str = 'default'"}, {"name": "set_default", "val": ": typing.Optional[bool] = None"}, {"name": "split", "val": ": typing.Optional[str] = None"}, {"name": "data_dir", "val": ": typing.Optional[str] = None"}, {"name": "commit_message", "val": ": typing.Optional[str] = None"}, {"name": "commit_description", "val": ": typing.Optional[str] = None"}, {"name": "private", "val": ": typing.Optional[bool] = None"}, {"name": "token", "val": ": typing.Optional[str] = None"}, {"name": "revision", "val": ": typing.Optional[str] = None"}, {"name": "create_pr", "val": ": typing.Optional[bool] = False"}, {"name": "max_shard_size", "val": ": typing.Union[str, int, NoneType] = None"}, {"name": "num_shards", "val": ": typing.Optional[int] = None"}, {"name": "embed_external_files", "val": ": bool = True"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}]- **repo_id** (`str`) --
The ID of the repository to push to in the following format: `<user>/<dataset_name>` or `<org>/<dataset_name>`. Also accepts `<dataset_name>`, which will default to the namespace of the logged-in user.

- **config_name** (`str`, defaults to "default") --
  The configuration name (or subset) of a dataset. Defaults to "default".
- **set_default** (`bool`, *optional*) --
  Whether to set this configuration as the default one. Otherwise, the default configuration is the one
  named "default".
- **split** (`str`, *optional*) --
  The name of the split that will be given to that dataset. Defaults to `self.split`.
- **data_dir** (`str`, *optional*) --
  Directory name that will contain the uploaded data files. Defaults to the `config_name` if different
  from "default", else "data".

- **commit_message** (`str` , *optional*) --
  Message to commit while pushing. Will default to `"Upload dataset"` .
- **commit_description** (`str` , *optional*) --
  Description of the commit that will be created.
  Additionally, description of the PR if a PR is created (`create_pr` is True).
- **private** (`bool` , *optional*) --
  Whether to make the repo private. If `None` (default), the repo will be public unless the organization's default is private. This value is ignored if the repo already exists.
- **token** (`str` , *optional*) --
  An optional authentication token for the Hugging Face Hub. If no token is passed, will default
  to the token saved locally when logging in with `huggingface-cli login` . Will raise an error if no token is passed and the user is not logged-in.
- **revision** (`str` , *optional*) --
  Branch to push the uploaded files to. Defaults to the `"main"` branch.
- **create_pr** (`bool` , *optional*, defaults to `False`) --
  Whether to create a PR with the uploaded files or directly commit.
- **max_shard_size** (`int` or `str` , *optional*, defaults to `"500MB"`) --
  The maximum size of the dataset shards to be uploaded to the hub. If expressed as a string, needs to be digits followed by
  a unit (like `"5MB"`).
- **num_shards** (`int` , *optional*) --
  Number of shards to write. By default, the number of shards depends on `max_shard_size` .
- **embed_external_files** (`bool` , defaults to `True`) --
  Whether to embed file bytes in the shards.
  In particular, this will do the following before the push for the fields of type:
  - Audio and Image: remove local path information and embed file content in the Parquet files.
- **num_proc** (`int` , *optional*, defaults to `None`) --
  Number of processes when preparing and uploading the dataset.
  This is helpful if the dataset is made of many samples or media files to embed.
  Multiprocessing is disabled by default.
  0huggingface_hub.CommitInfo
  Pushes the dataset to the hub as a Parquet dataset.
  The dataset is pushed using HTTP requests and does not need to have neither git or git-

lfs installed.

The resulting Parquet files are self-contained by default. If your dataset contains Image, Audio or Video
data, the Parquet files will store the bytes of your images or audio files.
You can disable this by setting `embed_external_files` to `False`.

Example:

```
>>> dataset.push_to_hub("<organization>/<dataset_id>")
>>> dataset_dict.push_to_hub("<organization>/<dataset_id>", private=True)
>>> dataset.push_to_hub("<organization>/<dataset_id>", max_shard_size="1GB")
>>> dataset.push_to_hub("<organization>/<dataset_id>", num_shards=1024)
```

If your dataset has multiple splits (e.g. train/validation/test):

```
>>> train_dataset.push_to_hub("<organization>/<dataset_id>", split="train")
>>> val_dataset.push_to_hub("<organization>/<dataset_id>", split="validation")
>>> # later
>>> dataset = load_dataset("<organization>/<dataset_id>")
>>> train_dataset = dataset["train"]
>>> val_dataset = dataset["validation"]
```

If you want to add a new configuration (or subset) to a dataset (e.g. if the dataset has multiple tasks/versions/languages):

```
>>> english_dataset.push_to_hub("<organization>/<dataset_id>", "en")
>>> french_dataset.push_to_hub("<organization>/<dataset_id>", "fr")
>>> # later
>>> english_dataset = load_dataset("<organization>/<dataset_id>", "en")
>>> french_dataset = load_dataset("<organization>/<dataset_id>", "fr")
```

save_to_diskdatasets.Dataset.save_to_diskhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1493[{"name": "dataset_path", "val": ": typing.Union[str, bytes, os.PathLike]"}, {"name": "max_shard_size", "val": ": typing.Union[str, int, NoneType] = None"}, {"name": "num_shards", "val": ": typing.Optional[int] = None"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "storage_options", "val": ": typing.Optional[dict]

= None"}]- **dataset_path** (`path-like`) --
Path (e.g. `dataset/train`) or remote URI (e.g. `s3://my-bucket/dataset/train`)
of the dataset directory where the dataset will be saved to.

- **max_shard_size** (`int` or `str` , *optional*, defaults to `"500MB"`) --
  The maximum size of the dataset shards to be saved to the filesystem. If expressed as a string, needs to be digits followed by a unit
  (like `"50MB"`).
- **num_shards** (`int` , *optional*) --
  Number of shards to write. By default the number of shards depends on `max_shard_size` and `num_proc` .
- **num_proc** (`int` , *optional*) --
  Number of processes when downloading and generating the dataset locally.
  Multiprocessing is disabled by default.
- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
  0

Saves a dataset to a dataset directory, or in a filesystem using any implementation of `fsspec.spec.AbstractFileSystem` .

For Image, Audio and Video data:

All the Image(), Audio() and Video() data are stored in the arrow files.
If you want to store paths or urls, please use the Value("string") type.

Example:

```
>>> ds.save_to_disk("path/to/dataset/directory")
>>> ds.save_to_disk("path/to/dataset/directory", max_shard_size="1GB")
>>> ds.save_to_disk("path/to/dataset/directory", num_shards=1024)
```

load_from_diskdatasets.Dataset.load_from_diskhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1687[{"name": "dataset_path", "val": ": typing.Union[str, bytes, os.PathLike]"}, {"name": "keep_in_memory", "val": ": typing.Optional[bool] = None"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}]- **dataset_path** (`path-like`) --

Path (e.g. `"dataset/train"`) or remote URI (e.g. `"s3//my-bucket/dataset/train"`)
of the dataset directory where the dataset will be loaded from.

- **keep_in_memory** (`bool`, defaults to `None`) --
  Whether to copy the dataset in-memory. If `None`, the
  dataset will not be copied in-memory unless explicitly enabled by setting
  `datasets.config.IN_MEMORY_MAX_SIZE` to nonzero. See more details in the
  improve performance section.
- **storage_options** (`dict`, *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
  0Dataset or DatasetDict- If `dataset_path` is a path of a dataset directory, the dataset
  requested.
- If `dataset_path` is a path of a dataset dict directory, a `datasets.DatasetDict` with each
  split.

Loads a dataset that was previously saved using `save_to_disk` from a dataset directory, or
from a
filesystem using any implementation of `fsspec.spec.AbstractFileSystem`.

Example:

```
>>> ds = load_from_disk("path/to/dataset/directory")
```

flatten_indicesdatasets.Dataset.flatten_indiceshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L3940[{"name": "keep_in_memory", "val": ": bool = False"}, {"name": "cache_file_name", "val": ": typing.Optional[str] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "disable_nullable", "val": ": bool = False"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "new_fingerprint", "val": ": typing.Optional[str] = None"}]- **keep_in_memory** (`bool`, defaults to `False`) --
Keep the dataset in memory instead of writing it to a cache file.

- **cache_file_name** (`str`, *optional*, default `None`) --
  Provide the name of a path for the cache file. It is used to store the
  results of the computation instead of the automatically generated cache file name.

- **writer_batch_size** (`int` , defaults to `1000`) --
  Number of rows per write operation for the cache file writer.
  This value is a good trade-off between memory usage during the processing, and processing speed.
  Higher value makes the processing do fewer lookups, lower value consume less temporary memory while running `map` .
- **features** (`Optional[datasets.Features]` , defaults to `None`) --
  Use a specific Features to store the cache file
  instead of the automatically generated one.
- **disable_nullable** (`bool` , defaults to `False`) --
  Allow null values in the table.
- **num_proc** (`int` , optional, default `None`) --
  Max number of processes when generating cache. Already cached shards are loaded sequentially
- **new_fingerprint** (`str` , *optional*, defaults to `None`) --
  The new fingerprint of the dataset after transform.
  If `None` , the new fingerprint is computed using a hash of the previous fingerprint, and the transform arguments0
  Create and cache a new Dataset by flattening the indices mapping.

to_csvdatasets.Dataset.to_csvhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L4977[{"name": "path_or_buf", "val": ": typing.Union[str, bytes, os.PathLike, typing.BinaryIO]"}, {"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}, {"name": "**to_csv_kwargs", "val": ""}]- **path_or_buf** (`PathLike` or `FileOrBuffer`) --
Either a path to a file (e.g. `file.csv`), a remote URI (e.g. `hf://datasets/username/my_dataset_name/data.csv`),
or a BinaryIO, where the dataset will be saved to in the specified format.

- **batch_size** (`int` , *optional*) --
  Size of the batch to load in memory and write at once.
  Defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` .
- **num_proc** (`int` , *optional*) --
  Number of processes for multiprocessing. By default it doesn't
  use multiprocessing. `batch_size` in this case defaults to

`datasets.config.DEFAULT_MAX_BATCH_SIZE` but feel free to make it 5x or 10x of the default value if you have sufficient compute power.

- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
- ****to_csv_kwargs** (additional keyword arguments) --
  Parameters to pass to pandas's `pandas.DataFrame.to_csv` .
  Now, `index` defaults to `False` if not specified.

  If you would like to write the index, pass `index=True` and also set a name for the index column by
  passing `index_label` .

  0 `int` The number of characters or bytes written.
  Exports the dataset to csv

Example:

```
>>> ds.to_csv("path/to/dataset/directory")
```

to_pandasdatasets.Dataset.to_pandashttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L5141[{"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "batched", "val": ": bool = False"}]- **batch_size** (`int` , *optional*) --
The size (number of rows) of the batches if `batched` is `True` .
Defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` .

- **batched** (`bool`) --
  Set to `True` to return a generator that yields the dataset as batches
  of `batch_size` rows. Defaults to `False` (returns the whole datasets
  once).0 `pandas.DataFrame` or `Iterator[pandas.DataFrame]`
  Returns the dataset as a `pandas.DataFrame` . Can also return a generator for large datasets.

Example:

```
>>> ds.to_pandas()
```

to_dictdatasets.Dataset.to_dicthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/

arrow_dataset.py#L5036[{"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "batched", "val": ": bool = False"}]- **batch_size** (`int` , *optional*) -- The size (number of rows) of the batches if `batched` is `True` .
Defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` .

- **batched** (`bool`) --
  Set to `True` to return a generator that yields the dataset as batches
  of `batch_size` rows. Defaults to `False` (returns the whole datasets once).0 `dict` or `Iterator[dict]`
  Returns the dataset as a Python dict. Can also return a generator for large datasets.

Example:

```
>>> ds.to_dict()
```

to_jsondatasets.Dataset.to_jsonhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L5079[{"name": "path_or_buf", "val": ": typing.Union[str, bytes, os.PathLike, typing.BinaryIO]"}, {"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}, {"name": "**to_json_kwargs", "val": ""}]- **path_or_buf** (`PathLike` or `FileOrBuffer`) --
Either a path to a file (e.g. `file.json`), a remote URI (e.g. `hf://datasets/username/my_dataset_name/data.json`),
or a BinaryIO, where the dataset will be saved to in the specified format.

- **batch_size** (`int` , *optional*) --
  Size of the batch to load in memory and write at once.
  Defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` .
- **num_proc** (`int` , *optional*) --
  Number of processes for multiprocessing. By default, it doesn't
  use multiprocessing. `batch_size` in this case defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` but feel free to make it 5x or 10x of the default value if you have sufficient compute power.
- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
- ****to_json_kwargs** (additional keyword arguments) --

Parameters to pass to pandas's `pandas.DataFrame.to_json`.
Default arguments are `lines=True` and `orient="records".
The parameter `index` defaults to `False` if `orient` is `"split"` or `"table"`.

If you would like to write the index, pass `index=True`.

0 `int` The number of characters or bytes written.
Export the dataset to JSON Lines or JSON.

The default output format is JSON Lines.
To export to JSON, pass `lines=False` argument and the desired `orient`.

Example:

```
>>> ds.to_json("path/to/dataset/directory/filename.jsonl")
```

to_parquetdatasets.Dataset.to_parquethttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L5240[{"name": "path_or_buf", "val": ": typing.Union[str, bytes, os.PathLike, typing.BinaryIO]"}, {"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}, {"name": "**parquet_writer_kwargs", "val": ""}]- **path_or_buf** (`PathLike` or `FileOrBuffer`) --
Either a path to a file (e.g. `file.parquet`), a remote URI (e.g. `hf://datasets/username/my_dataset_name/data.parquet`),
or a BinaryIO, where the dataset will be saved to in the specified format.

- **batch_size** (`int` , *optional*) --
  Size of the batch to load in memory and write at once.
  By default it aims for row groups with maximum uncompressed byte size of "100MB", defined by `datasets.config.MAX_ROW_GROUP_SIZE`.
- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
- ****parquet_writer_kwargs** (additional keyword arguments) --
  Parameters to pass to PyArrow's `pyarrow.parquet.ParquetWriter`.0 `int` The number of characters or bytes written.
  Exports the dataset to parquet

Example:

```
>>> ds.to_parquet("path/to/dataset/directory")
```

to_sqldatasets.Dataset.to_sqlhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L5280[{"name": "name", "val": ": str"}, {"name": "con", "val": ": typing.Union[str, ForwardRef('sqlalchemy.engine.Connection'), ForwardRef('sqlalchemy.engine.Engine'), ForwardRef('sqlite3.Connection')]"}, {"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "**sql_writer_kwargs", "val": ""}]-

name (`str`) --
Name of SQL table.

- **con** (`str` or `sqlite3.Connection` or `sqlalchemy.engine.Connection` or `sqlalchemy.engine.Connection`) --
  A URI string or a SQLite3/SQLAlchemy connection object used to write to a database.
- **batch_size** (`int` , *optional*) --
  Size of the batch to load in memory and write at once.
  Defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` .
- ****sql_writer_kwargs** (additional keyword arguments) --
  Parameters to pass to pandas's `pandas.DataFrame.to_sql` .
  Now, `index` defaults to `False` if not specified.

  If you would like to write the index, pass `index=True` and also set a name for the index column by
  passing `index_label` .

  0 `int` The number of records written.
  Exports the dataset to a SQL database.

Example:

```
>>> # con provided as a connection URI string
>>> ds.to_sql("data", "sqlite:///my_own_db.sql")
>>> # con provided as a sqlite3 connection object
>>> import sqlite3
>>> con = sqlite3.connect("my_own_db.sql")
>>> with con:
...     ds.to_sql("data", con)
```

to_iterable_datasetdatasets.Dataset.to_iterable_datasethttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L5372[{"name": "num_shards", "val": ": typing.Optional[int] = 1"}]- **num_shards** (`int` , default to `1`) --
Number of shards to define when instantiating the iterable dataset. This is especially useful for big datasets to be able to shuffle properly,
and also to enable fast parallel loading using a PyTorch DataLoader or in distributed setups for example.
Shards are defined using datasets.Dataset.shard(): it simply slices the data without writing anything on disk.0datasets.IterableDataset
Get an datasets.IterableDataset from a map-style datasets.Dataset.
This is equivalent to loading a dataset in streaming mode with datasets.load_dataset(), but much faster since the data is streamed from local files.

Contrary to map-style datasets, iterable datasets are lazy and can only be iterated over (e.g. using a for loop).
Since they are read sequentially in training loops, iterable datasets are much faster than map-style datasets.
All the transformations applied to iterable datasets like filtering or processing are done on-the-fly when you start iterating over the dataset.

Still, it is possible to shuffle an iterable dataset using datasets.IterableDataset.shuffle().
This is a fast approximate shuffling that works best if you have multiple shards and if you specify a buffer size that is big enough.

To get the best speed performance, make sure your dataset doesn't have an indices mapping.
If this is the case, the data are not read contiguously, which can be slow sometimes.
You can use `ds = ds.flatten_indices()` to write your dataset in contiguous chunks of data and have optimal speed before switching to an iterable dataset.

Example:

Basic usage:

```
>>> ids = ds.to_iterable_dataset()
>>> for example in ids:
...     pass
```

With lazy filtering and processing:

```
>>> ids = ds.to_iterable_dataset()
>>> ids = ids.filter(filter_fn).map(process_fn)  # will filter and process on-the-fly when you sta
>>> for example in ids:
...     pass
```

With sharding to enable efficient shuffling:

```
>>> ids = ds.to_iterable_dataset(num_shards=64)  # the dataset is split into 64 shards to be itera
>>> ids = ids.shuffle(buffer_size=10_000)  # will shuffle the shards order and use a shuffle buffe
>>> for example in ids:
...     pass
```

With a PyTorch DataLoader:

```
>>> import torch
>>> ids = ds.to_iterable_dataset(num_shards=64)
>>> ids = ids.filter(filter_fn).map(process_fn)
>>> dataloader = torch.utils.data.DataLoader(ids, num_workers=4)  # will assign 64 / 4 = 16 shards
>>> for example in ids:
...     pass
```

With a PyTorch DataLoader and shuffling:

```
>>> import torch
>>> ids = ds.to_iterable_dataset(num_shards=64)
>>> ids = ids.shuffle(buffer_size=10_000)  # will shuffle the shards order and use a shuffle buffe
>>> dataloader = torch.utils.data.DataLoader(ids, num_workers=4)  # will assign 64 / 4 = 16 shards
>>> for example in ids:
...     pass
```

In a distributed setup like PyTorch DDP with a PyTorch DataLoader and shuffling

```
>>> from datasets.distributed import split_dataset_by_node
>>> ids = ds.to_iterable_dataset(num_shards=512)
>>> ids = ids.shuffle(buffer_size=10_000, seed=42)  # will shuffle the shards order and use a shuf
>>> ids = split_dataset_by_node(ds, world_size=8, rank=0)  # will keep only 512 / 8 = 64 shards fr
>>> dataloader = torch.utils.data.DataLoader(ids, num_workers=4)  # will assign 64 / 4 = 16 shards
>>> for example in ids:
...     pass
```

With shuffling and multiple epochs:

```
>>> ids = ds.to_iterable_dataset(num_shards=64)
>>> ids = ids.shuffle(buffer_size=10_000, seed=42)  # will shuffle the shards order and use a shuf
>>> for epoch in range(n_epochs):
...     ids.set_epoch(epoch)  # will use effective_seed = seed + epoch to shuffle the shards and f
...     for example in ids:
...         pass
```

Feel free to also use `IterableDataset.set_epoch()` when using a PyTorch DataLoader or in distributed setups.
add_faiss_indexdatasets.Dataset.add_faiss_indexhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L6110[{"name": "column", "val": ": str"}, {"name": "index_name", "val": ": typing.Optional[str] = None"}, {"name": "device", "val": ": typing.Optional[int] = None"}, {"name": "string_factory", "val": ": typing.Optional[str] = None"}, {"name": "metric_type", "val": ": typing.Optional[int] = None"}, {"name": "custom_index", "val": ": typing.Optional[ForwardRef('faiss.Index')] = None"}, {"name": "batch_size", "val": ": int = 1000"}, {"name": "train_size", "val": ": typing.Optional[int] = None"}, {"name": "faiss_verbose", "val": ": bool = False"}, {"name": "dtype", "val": " = <class 'numpy.float32'>"}]- **column** (`str`) -- The column of the vectors to add to the index.

- **index_name** (`str`, *optional*) --
  The `index_name`/identifier of the index.
  This is the `index_name` that is used to call get_nearest_examples() or search().
  By default it corresponds to `column`.
- **device** (`Union[int, List[int]]`, *optional*) --
  If positive integer, this is the index of the GPU to use. If negative integer, use all GPUs.
  If a list of positive integers is passed in, run only on those GPUs. By default it uses the

CPU.

- **string_factory** (`str` , *optional*) --
  This is passed to the index factory of Faiss to create the index.
  Default index class is `IndexFlat` .
- **metric_type** (`int` , *optional*) --
  Type of metric. Ex: `faiss.METRIC_INNER_PRODUCT` or `faiss.METRIC_L2` .
- **custom_index** (`faiss.Index` , *optional*) --
  Custom Faiss index that you already have instantiated and configured for your needs.
- **batch_size** (`int`) --
  Size of the batch to use while adding vectors to the `FaissIndex` . Default value is `1000` .

- **train_size** (`int` , *optional*) --
  If the index needs a training step, specifies how many vectors will be used to train the index.
- **faiss_verbose** (`bool` , defaults to `False`) --
  Enable the verbosity of the Faiss index.
- **dtype** (`data-type`) --
  The dtype of the numpy arrays that are indexed.
  Default is `np.float32` .0
  Add a dense index using Faiss for fast retrieval.
  By default the index is done over the vectors of the specified column.
  You can specify `device` if you want to run it on GPU (`device` must be the GPU index).
  You can find more information about Faiss here:
- For string factory

Example:

```
>>> ds = datasets.load_dataset('crime_and_punish', split='train')
>>> ds_with_embeddings = ds.map(lambda example: {'embeddings': embed(example['line'])})
>>> ds_with_embeddings.add_faiss_index(column='embeddings')
>>> # query
>>> scores, retrieved_examples = ds_with_embeddings.get_nearest_examples('embeddings', embed('my n
>>> # save index
>>> ds_with_embeddings.save_faiss_index('embeddings', 'my_index.faiss')

>>> ds = datasets.load_dataset('crime_and_punish', split='train')
>>> # load index
>>> ds.load_faiss_index('embeddings', 'my_index.faiss')
>>> # query
>>> scores, retrieved_examples = ds.get_nearest_examples('embeddings', embed('my new query'), k=10
```

add_faiss_index_from_external_arraysdatasets.Dataset.add_faiss_index_from_external_array shttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L6190[{"name": "external_arrays", "val": ": "}, {"name": "index_name", "val": ": str"}, {"name": "device", "val": ": typing.Optional[int] = None"}, {"name": "string_factory", "val": ": typing.Optional[str] = None"}, {"name": "metric_type", "val": ": typing.Optional[int] = None"}, {"name": "custom_index", "val": ": typing.Optional[ForwardRef('faiss.Index')] = None"}, {"name": "batch_size", "val": ": int = 1000"}, {"name": "train_size", "val": ": typing.Optional[int] = None"}, {"name": "faiss_verbose", "val": ": bool = False"}, {"name": "dtype", "val": " = <class 'numpy.float32'>"}]- **external_arrays** (`np.array`) --
If you want to use arrays from outside the lib for the index, you can set `external_arrays`.
It will use `external_arrays` to create the Faiss index instead of the arrays in the given `column`.

- **index_name** (`str`) --
  The `index_name` /identifier of the index.
  This is the `index_name` that is used to call get_nearest_examples() or search().
- **device** (Optional `Union[int, List[int]]` , *optional*) --
  If positive integer, this is the index of the GPU to use. If negative integer, use all GPUs.
  If a list of positive integers is passed in, run only on those GPUs. By default it uses the CPU.
- **string_factory** (`str` , *optional*) --
  This is passed to the index factory of Faiss to create the index.
  Default index class is `IndexFlat` .

- **metric_type** (`int` , *optional*) --
  Type of metric. Ex: `faiss.faiss.METRIC_INNER_PRODUCT` or `faiss.METRIC_L2` .
- **custom_index** (`faiss.Index` , *optional*) --
  Custom Faiss index that you already have instantiated and configured for your needs.
- **batch_size** (`int` , *optional*) --
  Size of the batch to use while adding vectors to the FaissIndex. Default value is 1000.

- **train_size** (`int` , *optional*) --
  If the index needs a training step, specifies how many vectors will be used to train the index.
- **faiss_verbose** (`bool` , defaults to False) --
  Enable the verbosity of the Faiss index.
- **dtype** (`numpy.dtype`) --
  The dtype of the numpy arrays that are indexed. Default is np.float32.0
  Add a dense index using Faiss for fast retrieval.
  The index is created using the vectors of `external_arrays` .
  You can specify `device` if you want to run it on GPU (`device` must be the GPU index).
  You can find more information about Faiss here:
- For string factory

save_faiss_indexdatasets.Dataset.save_faiss_indexhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/search.py#L535[{"name": "index_name", "val": ": str"}, {"name": "file", "val": ": typing.Union[str, pathlib.PurePath]"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}]- **index_name** (`str`) -- The index_name/identifier of the index. This is the index_name that is used to call `.get_nearest` or `.search` .

- **file** (`str`) -- The path to the serialized faiss index on disk or remote URI (e.g. `"s3://my-bucket/index.faiss"`).
- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
  0
  Save a FaissIndex on disk.

load_faiss_indexdatasets.Dataset.load_faiss_indexhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/search.py#L553[{"name": "index_name", "val": ": str"}, {"name": "file", "val": ": typing.Union[str, pathlib.PurePath]"}, {"name": "device", "val": ": typing.Union[list[int],

int, NoneType] = None"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}]- **index_name** (`str`) -- The index_name/identifier of the index. This is the index_name that is used to
call `.get_nearest` or `.search` .

- **file** (`str`) -- The path to the serialized faiss index on disk or remote URI (e.g. `"s3://my-bucket/index.faiss"`).
- **device** (Optional `Union[int, List[int]]`) -- If positive integer, this is the index of the GPU to use. If negative integer, use all GPUs.
  If a list of positive integers is passed in, run only on those GPUs. By default it uses the CPU.
- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
  0
  Load a FaissIndex from disk.

If you want to do additional configurations, you can have access to the faiss index object by doing
`.get_index(index_name).faiss_index` to make it fit your needs.

add_elasticsearch_indexdatasets.Dataset.add_elasticsearch_indexhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L6249[{"name": "column", "val": ": str"}, {"name": "index_name", "val": ": typing.Optional[str] = None"}, {"name": "host", "val": ": typing.Optional[str] = None"}, {"name": "port", "val": ": typing.Optional[int] = None"}, {"name": "es_client", "val": ": typing.Optional[ForwardRef('elasticsearch.Elasticsearch')] = None"}, {"name": "es_index_name", "val": ": typing.Optional[str] = None"}, {"name": "es_index_config", "val": ": typing.Optional[dict] = None"}]- **column** (`str`) --
The column of the documents to add to the index.

- **index_name** (`str` , *optional*) --
  The `index_name` /identifier of the index.
  This is the index name that is used to call get_nearest_examples() or search().
  By default it corresponds to `column` .
- **host** (`str` , *optional*, defaults to `localhost`) --
  Host of where ElasticSearch is running.
- **port** (`str` , *optional*, defaults to `9200`) --

Port of where ElasticSearch is running.

- **es_client** (`elasticsearch.Elasticsearch` , *optional*) --
  The elasticsearch client used to create the index if host and port are `None` .
- **es_index_name** (`str` , *optional*) --
  The elasticsearch index name used to create the index.
- **es_index_config** (`dict` , *optional*) --
  The configuration of the elasticsearch index.
  Default config is:

```
{
    "settings": {
        "number_of_shards": 1,
        "analysis": {"analyzer": {"stop_standard": {"type": "standard", " stopwords": "_english_"}
    },
    "mappings": {
        "properties": {
            "text": {
                "type": "text",
                "analyzer": "standard",
                "similarity": "BM25"
            },
        }
    },
}
```</paramsdesc><paramgroups>0</paramgroups></docstring>
Add a text index using ElasticSearch for fast retrieval. This is done in-place.

<ExampleCodeBlock anchor="datasets.Dataset.add_elasticsearch_index.example">

Example:

```python
>>> es_client = elasticsearch.Elasticsearch()
>>> ds = datasets.load_dataset('crime_and_punish', split='train')
>>> ds.add_elasticsearch_index(column='line', es_client=es_client, es_index_name="my_es_index")
>>> scores, retrieved_examples = ds.get_nearest_examples('line', 'my new query', k=10)
```

load_elasticsearch_indexdatasets.Dataset.load_elasticsearch_indexhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/search.py#L637[{"name": "index_name", "val": ": str"}, {"name": "es_index_name", "val": ": str"}, {"name": "host", "val": ": typing.Optional[str] = None"}, {"name": "port", "val": ": typing.Optional[int] = None"}, {"name": "es_client", "val": ": typing.Optional[ForwardRef('Elasticsearch')] = None"}, {"name": "es_index_config", "val": ": typing.Optional[dict] = None"}]- **index_name** (`str`) --
The `index_name` /identifier of the index. This is the index name that is used to call `get_nearest` or `search` .

- **es_index_name** (`str`) --
  The name of elasticsearch index to load.
- **host** (`str` , *optional*, defaults to `localhost`) --
  Host of where ElasticSearch is running.
- **port** (`str` , *optional*, defaults to `9200`) --
  Port of where ElasticSearch is running.
- **es_client** (`elasticsearch.Elasticsearch` , *optional*) --
  The elasticsearch client used to create the index if host and port are `None` .
- **es_index_config** (`dict` , *optional*) --
  The configuration of the elasticsearch index.
  Default config is:

```
{
    "settings": {
        "number_of_shards": 1,
        "analysis": {"analyzer": {"stop_standard": {"type": "standard", " stopwords": "_english_"}
    },
    "mappings": {
        "properties": {
            "text": {
                "type": "text",
                "analyzer": "standard",
                "similarity": "BM25"
            },
        }
    },
}
```</paramsdesc><paramgroups>0</paramgroups></docstring>
Load an existing text index using ElasticSearch for fast retrieval.

<docstring><name>list_indexes</name><anchor>datasets.Dataset.list_indexes</anchor><source>https://
List the `colindex_nameumns`/identifiers of all the attached indexes.

<docstring><name>get_index</name><anchor>datasets.Dataset.get_index</anchor><source>https://github
List the `index_name`/identifiers of all the attached indexes.

```
</div>
<div class="docstring border-l-2 border-t-2 pl-4 pt-3.5 border-gray-100 rounded-tl-xl mb-6 mt-8">


<docstring><name>drop_index</name><anchor>datasets.Dataset.drop_index</anchor><source>https://gith
  The `index_name`/identifier of the index.</paramsdesc><paramgroups>0</paramgroups></docstring>
Drop the index with the specified column.




</div>
<div class="docstring border-l-2 border-t-2 pl-4 pt-3.5 border-gray-100 rounded-tl-xl mb-6 mt-8">


<docstring><name>search</name><anchor>datasets.Dataset.search</anchor><source>https://github.com/h
  The name/identifier of the index.
- **query** (`Union[str, np.ndarray]`) --
  The query as a string if `index_name` is a text index or as a numpy array if `index_name` is a v
- **k** (`int`) --
  The number of examples to retrieve.</paramsdesc><paramgroups>0</paramgroups><rettype>`(scores, i
- **scores** (`List[List[float]`): the retrieval scores from either FAISS (`IndexFlatL2` by defaul
- **indices** (`List[List[int]]`): the indices of the retrieved examples</retdesc></docstring>
Find the nearest examples indices in the dataset to the query.









</div>
<div class="docstring border-l-2 border-t-2 pl-4 pt-3.5 border-gray-100 rounded-tl-xl mb-6 mt-8">


<docstring><name>search_batch</name><anchor>datasets.Dataset.search_batch</anchor><source>https://
  The `index_name`/identifier of the index.
- **queries** (`Union[List[str], np.ndarray]`) --
```

```
  The queries as a list of strings if `index_name` is a text index or as a numpy array if `index_n
- **k** (`int`) --
  The number of examples to retrieve per query.</paramsdesc><paramgroups>0</paramgroups><rettype>`
- **total_scores** (`List[List[float]`): the retrieval scores from either FAISS (`IndexFlatL2` by
- **total_indices** (`List[List[int]]`): the indices of the retrieved examples per query</retdesc>
Find the nearest examples indices in the dataset to the query.









</div>
<div class="docstring border-l-2 border-t-2 pl-4 pt-3.5 border-gray-100 rounded-tl-xl mb-6 mt-8">


<docstring><name>get_nearest_examples</name><anchor>datasets.Dataset.get_nearest_examples</anchor>
  The index_name/identifier of the index.
- **query** (`Union[str, np.ndarray]`) --
  The query as a string if `index_name` is a text index or as a numpy array if `index_name` is a v
- **k** (`int`) --
  The number of examples to retrieve.</paramsdesc><paramgroups>0</paramgroups><rettype>`(scores, e
- **scores** (`List[float]`): the retrieval scores from either FAISS (`IndexFlatL2` by default) or
- **examples** (`dict`): the retrieved examples</retdesc></docstring>
Find the nearest examples in the dataset to the query.









</div>
<div class="docstring border-l-2 border-t-2 pl-4 pt-3.5 border-gray-100 rounded-tl-xl mb-6 mt-8">


<docstring><name>get_nearest_examples_batch</name><anchor>datasets.Dataset.get_nearest_examples_ba
```

```
  The `index_name`/identifier of the index.
- **queries** (`Union[List[str], np.ndarray]`) --
  The queries as a list of strings if `index_name` is a text index or as a numpy array if `index_n
- **k** (`int`) --
  The number of examples to retrieve per query.</paramsdesc><paramgroups>0</paramgroups><rettype>`
- **total_scores** (`List[List[float]`): the retrieval scores from either FAISS (`IndexFlatL2` by
- **total_examples** (`List[dict]`): the retrieved examples per query</retdesc></docstring>
Find the nearest examples in the dataset to the query.









</div>
<div class="docstring border-l-2 border-t-2 pl-4 pt-3.5 border-gray-100 rounded-tl-xl mb-6 mt-8">


<docstring><name>info</name><anchor>datasets.Dataset.info</anchor><source>https://github.com/huggi
[DatasetInfo](/docs/datasets/v4.2.0/en/package_reference/main_classes#datasets.DatasetInfo) object

</div>
<div class="docstring border-l-2 border-t-2 pl-4 pt-3.5 border-gray-100 rounded-tl-xl mb-6 mt-8">


<docstring><name>split</name><anchor>datasets.Dataset.split</anchor><source>https://github.com/hug
[NamedSplit](/docs/datasets/v4.2.0/en/package_reference/builder_classes#datasets.NamedSplit) objec

</div>
<div class="docstring border-l-2 border-t-2 pl-4 pt-3.5 border-gray-100 rounded-tl-xl mb-6 mt-8">


<docstring><name>builder_name</name><anchor>datasets.Dataset.builder_name</anchor><source>https://


</div>
<div class="docstring border-l-2 border-t-2 pl-4 pt-3.5 border-gray-100 rounded-tl-xl mb-6 mt-8">
```

<docstring><name>citation</name><anchor>datasets.Dataset.citation</anchor><source>https://github.c

<docstring><name>config_name</name><anchor>datasets.Dataset.config_name</anchor><source>https://gi

<docstring><name>dataset_size</name><anchor>datasets.Dataset.dataset_size</anchor><source>https://

<docstring><name>description</name><anchor>datasets.Dataset.description</anchor><source>https://gi

<docstring><name>download_checksums</name><anchor>datasets.Dataset.download_checksums</anchor><sou

<docstring><name>download_size</name><anchor>datasets.Dataset.download_size</anchor><source>https:

<docstring><name>features</name><anchor>datasets.Dataset.features</anchor><source>https://github.c

<docstring><name>homepage</name><anchor>datasets.Dataset.homepage</anchor><source>https://github.c

<docstring><name>license</name><anchor>datasets.Dataset.license</anchor><source>https://github.com

<docstring><name>size_in_bytes</name><anchor>datasets.Dataset.size_in_bytes</anchor><source>https:

<docstring><name>supervised_keys</name><anchor>datasets.Dataset.supervised_keys</anchor><source>ht

<docstring><name>version</name><anchor>datasets.Dataset.version</anchor><source>https://github.com

```
</div>
<div class="docstring border-l-2 border-t-2 pl-4 pt-3.5 border-gray-100 rounded-tl-xl mb-6 mt-8">


<docstring><name>from_csv</name><anchor>datasets.Dataset.from_csv</anchor><source>https://github.c
  Path(s) of the CSV file(s).
- **split** ([NamedSplit](/docs/datasets/v4.2.0/en/package_reference/builder_classes#datasets.Name
  Split name to be assigned to the dataset.
- **features** ([Features](/docs/datasets/v4.2.0/en/package_reference/main_classes#datasets.Featur
  Dataset features.
- **cache_dir** (`str`, *optional*, defaults to `"~/.cache/huggingface/datasets"`) --
  Directory to cache data.
- **keep_in_memory** (`bool`, defaults to `False`) --
  Whether to copy the data in-memory.
- **num_proc** (`int`, *optional*, defaults to `None`) --
  Number of processes when downloading and generating the dataset locally.
  This is helpful if the dataset is made of multiple files. Multiprocessing is disabled by default

  <Added version="2.8.0"/>
- ****kwargs** (additional keyword arguments) --
  Keyword arguments to be passed to `pandas.read_csv`.</paramsdesc><paramgroups>0</paramgroups><re
Create Dataset from CSV file(s).




<ExampleCodeBlock anchor="datasets.Dataset.from_csv.example">

Example:

```py
>>> ds = Dataset.from_csv('path/to/dataset.csv')
```

from_jsondatasets.Dataset.from_jsonhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1189[{"name": "path_or_paths", "val": ": typing.Union[str, bytes, os.PathLike, list[typing.Union[str, bytes, os.PathLike]]]"}, {"name": "split", "val": ": typing.Optional[datasets.splits.NamedSplit] = None"}, {"name": "features", "val": ":
```

typing.Optional[datasets.features.features.Features] = None"}, {"name": "cache_dir", "val": ": str = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "field", "val": ": typing.Optional[str] = None"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "**kwargs", "val": ""}]- **path_or_paths** (`path-like` or list of `path-like`) -- Path(s) of the JSON or JSON Lines file(s).

- **split** (NamedSplit, *optional*) --
  Split name to be assigned to the dataset.
- **features** (Features, *optional*) --
  Dataset features.
- **cache_dir** (`str` , *optional*, defaults to `"~/.cache/huggingface/datasets"`) --
  Directory to cache data.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Whether to copy the data in-memory.
- **field** (`str` , *optional*) --
  Field name of the JSON file where the dataset is contained in.
- **num_proc** (`int` , *optional* defaults to `None`) --
  Number of processes when downloading and generating the dataset locally.
  This is helpful if the dataset is made of multiple files. Multiprocessing is disabled by default.
- ****kwargs** (additional keyword arguments) --
  Keyword arguments to be passed to `JsonConfig` .0Dataset
  Create Dataset from JSON or JSON Lines file(s).

Example:

```
>>> ds = Dataset.from_json('path/to/dataset.json')
```

from_parquetdatasets.Dataset.from_parquethttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1246[{"name": "path_or_paths", "val": ": typing.Union[str, bytes, os.PathLike, list[typing.Union[str, bytes, os.PathLike]]]"}, {"name": "split", "val": ": typing.Optional[datasets.splits.NamedSplit] = None"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "cache_dir", "val": ": str = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "columns", "val": ": typing.Optional[list[str]] = None"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "**kwargs", "val": ""}]- **path_or_paths** (`path-like` or list of `path-like`) --

Path(s) of the Parquet file(s).

- **split** (`NamedSplit` , *optional*) --
  Split name to be assigned to the dataset.
- **features** (`Features` , *optional*) --
  Dataset features.
- **cache_dir** (`str` , *optional*, defaults to `"~/.cache/huggingface/datasets"`) --
  Directory to cache data.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Whether to copy the data in-memory.
- **columns** (`List[str]` , *optional*) --
  If not `None` , only these columns will be read from the file.
  A column name may be a prefix of a nested field, e.g. 'a' will select
  'a.b', 'a.c', and 'a.d.e'.
- **num_proc** (`int` , *optional*, defaults to `None`) --
  Number of processes when downloading and generating the dataset locally.
  This is helpful if the dataset is made of multiple files. Multiprocessing is disabled by default.
- ****kwargs** (additional keyword arguments) --
  Keyword arguments to be passed to `ParquetConfig` .0Dataset
  Create Dataset from Parquet file(s).

Example:

```
>>> ds = Dataset.from_parquet('path/to/dataset.parquet')
```

from_textdatasets.Dataset.from_texthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1305[{"name": "path_or_paths", "val": ": typing.Union[str, bytes, os.PathLike, list[typing.Union[str, bytes, os.PathLike]]]"}, {"name": "split", "val": ": typing.Optional[datasets.splits.NamedSplit] = None"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "cache_dir", "val": ": str = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "**kwargs", "val": ""}]- **path_or_paths** (`path-like` or list of `path-like`) --
Path(s) of the text file(s).

- **split** (`NamedSplit` , *optional*) --
  Split name to be assigned to the dataset.
- **features** (`Features` , *optional*) --
  Dataset features.
- **cache_dir** (`str` , *optional*, defaults to `"~/.cache/huggingface/datasets"`) --
  Directory to cache data.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Whether to copy the data in-memory.
- **num_proc** (`int` , *optional*, defaults to `None`) --
  Number of processes when downloading and generating the dataset locally.
  This is helpful if the dataset is made of multiple files. Multiprocessing is disabled by default.
- ****kwargs** (additional keyword arguments) --
  Keyword arguments to be passed to `TextConfig` .0Dataset
  Create Dataset from text file(s).

Example:

```
>>> ds = Dataset.from_text('path/to/dataset.txt')
```

from_sqldatasets.Dataset.from_sqlhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L1420[{"name": "sql", "val": ": typing.Union[str, ForwardRef('sqlalchemy.sql.Selectable')]"}, {"name": "con", "val": ": typing.Union[str, ForwardRef('sqlalchemy.engine.Connection'), ForwardRef('sqlalchemy.engine.Engine'), ForwardRef('sqlite3.Connection')]"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "cache_dir", "val": ": str = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "**kwargs", "val": ""}]- **sql** (`str` or `sqlalchemy.sql.Selectable`) --
SQL query to be executed or a table name.

- **con** (`str` or `sqlite3.Connection` or `sqlalchemy.engine.Connection` or `sqlalchemy.engine.Connection`) --
  A URI string used to instantiate a database connection or a SQLite3/SQLAlchemy connection object.
- **features** (Features, *optional*) --
  Dataset features.

- **cache_dir** (`str` , *optional*, defaults to `"~/.cache/huggingface/datasets"`) --
  Directory to cache data.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Whether to copy the data in-memory.
- ****kwargs** (additional keyword arguments) --
  Keyword arguments to be passed to `SqlConfig` .0Dataset
  Create Dataset from SQL query or database table.

Example:

```
>>> # Fetch a database table
>>> ds = Dataset.from_sql("test_data", "postgres:///db_name")
>>> # Execute a SQL query on the table
>>> ds = Dataset.from_sql("SELECT sentence FROM test_data", "postgres:///db_name")
>>> # Use a Selectable object to specify the query
>>> from sqlalchemy import select, text
>>> stmt = select([text("sentence")]).select_from(text("test_data"))
>>> ds = Dataset.from_sql(stmt, "postgres:///db_name")
```

> [!TIP]
> The returned dataset can only be cached if `con` is specified as URI string.

align_labels_with_mappingdatasets.Dataset.align_labels_with_mappinghttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L6371[{"name": "label2id", "val": ": dict"}, {"name": "label_column", "val": ": str"}]- **label2id** (`dict`) --
The label name to ID mapping to align the dataset with.

- **label_column** (`str`) --
  The column name of labels to align on.0
  Align the dataset's label ID and label name mapping to match an input `label2id` mapping. This is useful when you want to ensure that a model's predicted labels are aligned with the dataset.
  The alignment in done using the lowercase label names.

Example:

```
>>> # dataset with mapping {'entailment': 0, 'neutral': 1, 'contradiction': 2}
>>> ds = load_dataset("nyu-mll/glue", "mnli", split="train")
>>> # mapping to align with
>>> label2id = {'CONTRADICTION': 0, 'NEUTRAL': 1, 'ENTAILMENT': 2}
>>> ds_aligned = ds.align_labels_with_mapping(label2id, "label")
```

datasets.concatenate_datasetsdatasets.concatenate_datasetshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/combine.py#L166[{"name": "dsets", "val": ": list"}, {"name": "info", "val": ": typing.Optional[datasets.info.DatasetInfo] = None"}, {"name": "split", "val": ": typing.Optional[datasets.splits.NamedSplit] = None"}, {"name": "axis", "val": ": int = 0"}]- **dsets** (`List[datasets.Dataset]`) --
List of Datasets to concatenate.

- **info** (`DatasetInfo` , *optional*) --
  Dataset information, like description, citation, etc.
- **split** (`NamedSplit` , *optional*) --
  Name of the dataset split.
- **axis** (`{0, 1}` , defaults to `0`) --
  Axis to concatenate over, where `0` means over rows (vertically) and `1` means over columns
  (horizontally).
  0

Converts a list of Dataset with the same schema into a single Dataset.

Example:

```
>>> ds3 = concatenate_datasets([ds1, ds2])
```

datasets.interleave_datasetsdatasets.interleave_datasetshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/combine.py#L18[{"name": "datasets", "val": ": list"}, {"name": "probabilities", "val": ": typing.Optional[list[float]] = None"}, {"name": "seed", "val": ": typing.Optional[int] = None"}, {"name": "info", "val": ": typing.Optional[datasets.info.DatasetInfo] = None"}, {"name": "split", "val": ": typing.Optional[datasets.splits.NamedSplit] = None"}, {"name": "stopping_strategy", "val": ": typing.Literal['first_exhausted', 'all_exhausted', 'all_exhausted_without_replacement'] = 'first_exhausted'"}]- **datasets** (`List[Dataset]` or

`List[IterableDataset]`) --
List of datasets to interleave.

- **probabilities** (`List[float]` , *optional*, defaults to `None`) --
  If specified, the new dataset is constructed by sampling
  examples from one source at a time according to these probabilities.
- **seed** (`int` , *optional*, defaults to `None`) --
  The random seed used to choose a source for each example.
- **info** (DatasetInfo, *optional*) --
  Dataset information, like description, citation, etc.

- **split** (NamedSplit, *optional*) --
  Name of the dataset split.

- **stopping_strategy** (`str` , defaults to `first_exhausted`) --
  Three strategies are proposed right now, `first_exhausted` , `all_exhausted` and
  `all_exhausted_without_replacement` .
  By default, `first_exhausted` is an undersampling strategy, i.e the dataset construction is stopped as soon as one dataset has ran out of samples.
  If the strategy is `all_exhausted` , we use an oversampling strategy, i.e the dataset construction is stopped as soon as every samples of every dataset has been added at least once.
  When strategy is `all_exhausted_without_replacement` we make sure that each sample in each dataset is sampled only once.
  Note that if the strategy is `all_exhausted` , the interleaved dataset size can get enormous:
  - with no probabilities, the resulting dataset will have `max_length_datasets*nb_dataset` samples.
  - with given probabilities, the resulting dataset will have more samples if some datasets have really low probability of visiting.0Dataset or IterableDatasetReturn type depends on the input `datasets`
    parameter. `Dataset` if the input is a list of `Dataset` , `IterableDataset` if the input is a list of
    `IterableDataset` .

Interleave several datasets (sources) into a single dataset.
The new dataset is constructed by alternating between the sources to get the examples.

You can use this function on a list of Dataset objects, or on a list of IterableDataset objects.

- If `probabilities` is `None` (default) the new dataset is constructed by cycling between each source to get the examples.
- If `probabilities` is not `None`, the new dataset is constructed by getting examples from a random source at a time according to the provided probabilities.

The resulting dataset ends when one of the source datasets runs out of examples except when `oversampling` is `True`,
in which case, the resulting dataset ends when all datasets have ran out of examples at least one time.

Note for iterable datasets:

In a distributed setup or in PyTorch DataLoader workers, the stopping strategy is applied per process.
Therefore the "first_exhausted" strategy on an sharded iterable dataset can generate less samples in total (up to 1 missing sample per subdataset per worker).

Example:

For regular datasets (map-style):

```
>>> from datasets import Dataset, interleave_datasets
>>> d1 = Dataset.from_dict({"a": [0, 1, 2]})
>>> d2 = Dataset.from_dict({"a": [10, 11, 12]})
>>> d3 = Dataset.from_dict({"a": [20, 21, 22]})
>>> dataset = interleave_datasets([d1, d2, d3], probabilities=[0.7, 0.2, 0.1], seed=42, stopping_s
>>> dataset["a"]
[10, 0, 11, 1, 2, 20, 12, 10, 0, 1, 2, 21, 0, 11, 1, 2, 0, 1, 12, 2, 10, 0, 22]
>>> dataset = interleave_datasets([d1, d2, d3], probabilities=[0.7, 0.2, 0.1], seed=42)
>>> dataset["a"]
[10, 0, 11, 1, 2]
>>> dataset = interleave_datasets([d1, d2, d3])
>>> dataset["a"]
[0, 10, 20, 1, 11, 21, 2, 12, 22]
>>> dataset = interleave_datasets([d1, d2, d3], stopping_strategy="all_exhausted")
>>> dataset["a"]
[0, 10, 20, 1, 11, 21, 2, 12, 22]
>>> d1 = Dataset.from_dict({"a": [0, 1, 2]})
>>> d2 = Dataset.from_dict({"a": [10, 11, 12, 13]})
>>> d3 = Dataset.from_dict({"a": [20, 21, 22, 23, 24]})
>>> dataset = interleave_datasets([d1, d2, d3])
>>> dataset["a"]
[0, 10, 20, 1, 11, 21, 2, 12, 22]
>>> dataset = interleave_datasets([d1, d2, d3], stopping_strategy="all_exhausted")
>>> dataset["a"]
[0, 10, 20, 1, 11, 21, 2, 12, 22, 0, 13, 23, 1, 10, 24]
>>> dataset = interleave_datasets([d1, d2, d3], probabilities=[0.7, 0.2, 0.1], seed=42)
>>> dataset["a"]
[10, 0, 11, 1, 2]
>>> dataset = interleave_datasets([d1, d2, d3], probabilities=[0.7, 0.2, 0.1], seed=42, stopping_s
>>> dataset["a"]
[10, 0, 11, 1, 2, 20, 12, 13, ..., 0, 1, 2, 0, 24]
For datasets in streaming mode (iterable):

>>> from datasets import interleave_datasets
>>> d1 = load_dataset('allenai/c4', 'es', split='train', streaming=True)
>>> d2 = load_dataset('allenai/c4', 'fr', split='train', streaming=True)
>>> dataset = interleave_datasets([d1, d2])
>>> iterator = iter(dataset)
>>> next(iterator)
```

```
{'text': 'Comprar Zapatillas para niña en chancla con goma por...'}
>>> next(iterator)
{'text': 'Le sacre de philippe ier, 23 mai 1059 - Compte Rendu...'
```

datasets.distributed.split_dataset_by_nodedatasets.distributed.split_dataset_by_nodehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/distributed.py#L10[{"name": "dataset", "val": ": ~DatasetType"}, {"name": "rank", "val": ": int"}, {"name": "world_size", "val": ": int"}]-
dataset (Dataset or IterableDataset) --
The dataset to split by node.

- **rank** (`int`) --
  Rank of the current node.
- **world_size** (`int`) --
  Total number of nodes.0Dataset or IterableDatasetThe dataset to be used on the node at rank `rank` .

Split a dataset for the node at rank `rank` in a pool of nodes of size `world_size` .

For map-style datasets:

Each node is assigned a chunk of data, e.g. rank 0 is given the first chunk of the dataset.
To maximize data loading throughput, chunks are made of contiguous data on disk if possible.

For iterable datasets:

If the dataset has a number of shards that is a factor of `world_size` (i.e. if `dataset.num_shards % world_size == 0`),
then the shards are evenly assigned across the nodes, which is the most optimized.
Otherwise, each node keeps 1 example out of `world_size` , skipping the other examples.

datasets.enable_cachingdatasets.enable_cachinghttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/fingerprint.py#L94[]

When applying transforms on a dataset, the data are stored in cache files.
The caching mechanism allows to reload an existing cache file if it's already been computed.

Reloading a dataset is possible since the cache files are named using the dataset fingerprint, which is updated
after each transform.

If disabled, the library will no longer reload cached datasets files when applying transforms to the datasets.
More precisely, if the caching is disabled:

- cache files are always recreated
- cache files are written to a temporary directory that is deleted when session closes
- cache files are named using a random hash instead of the dataset fingerprint
- use save_to_disk() to save a transformed dataset or it will be deleted when session closes
- caching doesn't affect load_dataset(). If you want to regenerate a dataset from scratch you should use
  the `download_mode` parameter in load_dataset().

datasets.disable_cachingdatasets.disable_cachinghttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/fingerprint.py#L115[]

When applying transforms on a dataset, the data are stored in cache files.
The caching mechanism allows to reload an existing cache file if it's already been computed.

Reloading a dataset is possible since the cache files are named using the dataset fingerprint, which is updated
after each transform.

If disabled, the library will no longer reload cached datasets files when applying transforms to the datasets.
More precisely, if the caching is disabled:

- cache files are always recreated
- cache files are written to a temporary directory that is deleted when session closes
- cache files are named using a random hash instead of the dataset fingerprint
- use save_to_disk() to save a transformed dataset or it will be deleted when session closes
- caching doesn't affect load_dataset(). If you want to regenerate a dataset from scratch you should use
  the `download_mode` parameter in load_dataset().

datasets.is_caching_enableddatasets.is_caching_enabledhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/fingerprint.py#L136[]

When applying transforms on a dataset, the data are stored in cache files.

The caching mechanism allows to reload an existing cache file if it's already been computed.

Reloading a dataset is possible since the cache files are named using the dataset fingerprint, which is updated
after each transform.

If disabled, the library will no longer reload cached datasets files when applying transforms to the datasets.
More precisely, if the caching is disabled:

- cache files are always recreated
- cache files are written to a temporary directory that is deleted when session closes
- cache files are named using a random hash instead of the dataset fingerprint
- use save_to_disk()] to save a transformed dataset or it will be deleted when session closes
- caching doesn't affect load_dataset(). If you want to regenerate a dataset from scratch you should use
  the `download_mode` parameter in load_dataset().

class datasets.Columndatasets.Columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L633[{"name": "source", "val": ": typing.Union[ForwardRef('Dataset'), ForwardRef('Column')]"}, {"name": "column_name", "val": ": str"}]

An iterable for a specific column of a Dataset.

Example:

Iterate on the texts of the "text" column of a dataset:

```
for text in dataset["text"]:
    ...
```

It also works with nested columns:

```
for source in dataset["metadata"]["source"]:
    ...
```

DatasetDictdatasets.DatasetDict

Dictionary with split names as keys ('train', 'test' for example), and `Dataset` objects as values. It also has dataset transform methods like map or filter, to process all the splits at once.

class datasets.DatasetDictdatasets.DatasetDicthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L57""
A dictionary (dict of str: datasets.Dataset) with dataset transforms methods (map, filter, etc.)

datadatasets.DatasetDict.datahttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L98[]
The Apache Arrow tables backing each split.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds.data
```

cache_filesdatasets.DatasetDict.cache_fileshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L113[]
The cache files containing the Apache Arrow table backing each split.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds.cache_files
{'test': [{'filename': '/root/.cache/huggingface/datasets/rotten_tomatoes_movie_review/default/1.0
 'train': [{'filename': '/root/.cache/huggingface/datasets/rotten_tomatoes_movie_review/default/1.
 'validation': [{'filename': '/root/.cache/huggingface/datasets/rotten_tomatoes_movie_review/defau
```

num_columnsdatasets.DatasetDict.num_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L131[]
Number of columns in each split of the dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds.num_columns
{'test': 2, 'train': 2, 'validation': 2}
```

num_rowsdatasets.DatasetDict.num_rowshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L147[]
Number of rows in each split of the dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds.num_rows
{'test': 1066, 'train': 8530, 'validation': 1066}
```

column_namesdatasets.DatasetDict.column_nameshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L163[]
Names of the columns in each split of the dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds.column_names
{'test': ['text', 'label'],
 'train': ['text', 'label'],
 'validation': ['text', 'label']}
```

shapedatasets.DatasetDict.shapehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L181[]
Shape of each split of the dataset (number of rows, number of columns).

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds.shape
{'test': (1066, 2), 'train': (8530, 2), 'validation': (1066, 2)}
```

uniquedatasets.DatasetDict.uniquehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L230[{"name": "column", "val": ": str"}]- **column** (`str`) -- column name (list all the column names with column_names)0Dict[`str` , `list`]Dictionary of unique elements in the given column.
Return a list of the unique elements in a column for each split.

This is implemented in the low-level backend and as such, very fast.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds.unique("label")
{'test': [1, 0], 'train': [1, 0], 'validation': [1, 0]}
```

cleanup_cache_filesdatasets.DatasetDict.cleanup_cache_fileshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L254[] `Dict` with the number of removed files for each split
Clean up all cache files in the dataset cache directory, excepted the currently used cache file if there is one.
Be careful when running this command that no other process is currently using other cache files.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds.cleanup_cache_files()
{'test': 0, 'train': 0, 'validation': 0}
```

mapdatasets.DatasetDict.maphttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L818[{"name": "function", "val": ": typing.Optional[typing.Callable] = None"},

{"name": "with_indices", "val": ": bool = False"}, {"name": "with_rank", "val": ": bool = False"}, {"name": "with_split", "val": ": bool = False"}, {"name": "input_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "batched", "val": ": bool = False"}, {"name": "batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "drop_last_batch", "val": ": bool = False"}, {"name": "remove_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "load_from_cache_file", "val": ": typing.Optional[bool] = None"}, {"name": "cache_file_names", "val": ": typing.Optional[dict[str, typing.Optional[str]]] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "disable_nullable", "val": ": bool = False"}, {"name": "fn_kwargs", "val": ": typing.Optional[dict] = None"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "desc", "val": ": typing.Optional[str] = None"}, {"name": "try_original_type", "val": ": typing.Optional[bool] = True"}]- **function** (`callable`) -- with one of the following signature:

- `function(example: Dict[str, Any]) -> Dict[str, Any]` if `batched=False` and `with_indices=False`
- `function(example: Dict[str, Any], indices: int) -> Dict[str, Any]` if `batched=False` and `with_indices=True`
- `function(batch: Dict[str, list]) -> Dict[str, list]` if `batched=True` and `with_indices=False`
- `function(batch: Dict[str, list], indices: list[int]) -> Dict[str, list]` if `batched=True` and `with_indices=True`

For advanced usage, the function can also return a `pyarrow.Table`.
If the function is asynchronous, then `map` will run your function in parallel.
Moreover if your function returns nothing (`None`), then `map` will run your function and return the dataset unchanged.
If no function is provided, default to identity function: `lambda x: x`.

- **with_indices** (`bool` , defaults to `False`) --
  Provide example indices to `function`. Note that in this case the signature of `function` should be `def function(example, idx): ...`.
- **with_rank** (`bool` , defaults to `False`) --
  Provide process rank to `function`. Note that in this case the signature of `function` should be `def function(example[, idx], rank): ...`.

- **with_split** (`bool` , defaults to `False`) --
  Provide process split to `function` . Note that in this case the
  signature of `function` should be `def function(example[, idx], split): ...` .
- **input_columns** (`[Union[str, list[str]]]` , *optional*, defaults to `None`) --
  The columns to be passed into `function` as
  positional arguments. If `None` , a dict mapping to all formatted columns is passed as one
  argument.
- **batched** (`bool` , defaults to `False`) --
  Provide batch of examples to `function` .
- **batch_size** (`int` , *optional*, defaults to `1000`) --
  Number of examples per batch provided to `function` if `batched=True` ,
  `batch_size <= 0` or `batch_size == None` then provide the full dataset as a single batch to
  `function` .
- **drop_last_batch** (`bool` , defaults to `False`) --
  Whether a last batch smaller than the batch_size should be
  dropped instead of being processed by the function.
- **remove_columns** (`[Union[str, list[str]]]` , *optional*, defaults to `None`) --
  Remove a selection of columns while doing the mapping.
  Columns will be removed before updating the examples with the output of `function` , i.e. if
  `function` is adding
  columns with names in `remove_columns` , these columns will be kept.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Keep the dataset in memory instead of writing it to a cache file.
- **load_from_cache_file** (`Optional[bool]` , defaults to `True` if caching is enabled) --
  If a cache file storing the current computation from `function`
  can be identified, use it instead of recomputing.
- **cache_file_names** (`[Dict[str, str]]` , *optional*, defaults to `None`) --
  Provide the name of a path for the cache file. It is used to store the
  results of the computation instead of the automatically generated cache file name.
  You have to provide one `cache_file_name` per dataset in the dataset dictionary.
- **writer_batch_size** (`int` , default `1000`) --
  Number of rows per write operation for the cache file writer.
  This value is a good trade-off between memory usage during the processing, and
  processing speed.
  Higher value makes the processing do fewer lookups, lower value consume less

temporary memory while running `map`.

- **features** (`[datasets.Features]`, *optional*, defaults to `None`) --
  Use a specific Features to store the cache file
  instead of the automatically generated one.
- **disable_nullable** (`bool`, defaults to `False`) --
  Disallow null values in the table.
- **fn_kwargs** (`Dict`, *optional*, defaults to `None`) --
  Keyword arguments to be passed to `function`
- **num_proc** (`int`, *optional*, defaults to `None`) --
  The number of processes to use for multiprocessing.
  - If `None` or `0`, no multiprocessing is used and the operation runs in the main process.
  - If greater than `1`, one or multiple worker processes are used to process data in parallel.
    Note: The function passed to `map()` must be picklable for multiprocessing to work correctly
    (i.e., prefer functions defined at the top level of a module, not inside another function or class).
- **desc** (`str`, *optional*, defaults to `None`) --
  Meaningful description to be displayed alongside with the progress bar while mapping examples.
- **try_original_type** (`Optional[bool]`, defaults to `True`) --
  Try to keep the types of the original columns (e.g. int32 -> int32).
  Set to False if you want to always infer new types.0

Apply a function to all the examples in the table (individually or in batches) and update the table.
If your function returns a column that already exists, then it overwrites it.
The transformation is applied to all the datasets of the dataset dictionary.

You can specify whether the function should be batched or not with the `batched` parameter:

- If batched is `False`, then the function takes 1 example in and should return 1 example.
  An example is a dictionary, e.g. `{"text": "Hello there !"}`.
- If batched is `True` and `batch_size` is 1, then the function takes a batch of 1 example as input and can return a batch with 1 or more examples.
  A batch is a dictionary, e.g. a batch of 1 example is `{"text": ["Hello there !"]}`.

- If batched is `True` and `batch_size` is `n > 1`, then the function takes a batch of `n` examples as input and can return a batch with `n` examples, or with an arbitrary number of examples.
  Note that the last batch may have less than `n` examples.
  A batch is a dictionary, e.g. a batch of `n` examples is `{"text": ["Hello there !"] * n}`.

If the function is asynchronous, then `map` will run your function in parallel, with up to one thousand simultaneous calls.
It is recommended to use a `asyncio.Semaphore` in your function if you want to set a maximum number of operations that can run at the same time.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> def add_prefix(example):
...     example["text"] = "Review: " + example["text"]
...     return example
>>> ds = ds.map(add_prefix)
>>> ds["train"][0:3]["text"]
['Review: the rock is destined to be the 21st century's new " conan " and that he's going to make
 'Review: the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
 'Review: effective but too-tepid biopic']

# process a batch of examples
>>> ds = ds.map(lambda example: tokenizer(example["text"]), batched=True)
# set number of processors
>>> ds = ds.map(add_prefix, num_proc=4)
```

filterdatasets.DatasetDict.filterhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L979[{"name": "function", "val": ": typing.Optional[typing.Callable] = None"}, {"name": "with_indices", "val": ": bool = False"}, {"name": "with_rank", "val": ": bool = False"}, {"name": "input_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "batched", "val": ": bool = False"}, {"name": "batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "load_from_cache_file", "val": ": typing.Optional[bool] = None"}, {"name": "cache_file_names", "val": ": typing.Optional[dict[str, typing.Optional[str]]] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] =

1000"}, {"name": "fn_kwargs", "val": ": typing.Optional[dict] = None"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "desc", "val": ": typing.Optional[str] = None"}]- **function** (`Callable`) -- Callable with one of the following signatures:

- `function(example: Dict[str, Any]) -> bool` if `batched=False` and `with_indices=False` and `with_rank=False`
- `function(example: Dict[str, Any], *extra_args) -> bool` if `batched=False` and `with_indices=True` and/or `with_rank=True` (one extra arg for each)
- `function(batch: Dict[str, list]) -> list[bool]` if `batched=True` and `with_indices=False` and `with_rank=False`
- `function(batch: Dict[str, list], *extra_args) -> list[bool]` if `batched=True` and `with_indices=True` and/or `with_rank=True` (one extra arg for each)

If no function is provided, defaults to an always `True` function: `lambda x: True`.

- **with_indices** (`bool` , defaults to `False`) --
  Provide example indices to `function` . Note that in this case the signature of `function` should be `def function(example, idx[, rank]): ...` .
- **with_rank** (`bool` , defaults to `False`) --
  Provide process rank to `function` . Note that in this case the signature of `function` should be `def function(example[, idx], rank): ...` .
- **input_columns** (`[Union[str, list[str]]]` , *optional*, defaults to `None`) --
  The columns to be passed into `function` as positional arguments. If `None` , a dict mapping to all formatted columns is passed as one argument.
- **batched** (`bool` , defaults to `False`) --
  Provide batch of examples to `function` .
- **batch_size** (`int` , *optional*, defaults to `1000`) --
  Number of examples per batch provided to `function` if `batched=True` `batch_size <= 0` or `batch_size == None` then provide the full dataset as a single batch to `function` .
- **keep_in_memory** (`bool` , defaults to `False`) --
  Keep the dataset in memory instead of writing it to a cache file.
- **load_from_cache_file** (`Optional[bool]` , defaults to `True` if caching is enabled) --
  If a cache file storing the current computation from `function` can be identified, use it instead of recomputing.

- **cache_file_names** (`[Dict[str, str]]` , *optional*, defaults to `None`) --
  Provide the name of a path for the cache file. It is used to store the
  results of the computation instead of the automatically generated cache file name.
  You have to provide one `cache_file_name` per dataset in the dataset dictionary.
- **writer_batch_size** (`int` , defaults to `1000`) --
  Number of rows per write operation for the cache file writer.
  This value is a good trade-off between memory usage during the processing, and
  processing speed.
  Higher value makes the processing do fewer lookups, lower value consume less
  temporary memory while running `map` .
- **fn_kwargs** (`Dict` , *optional*, defaults to `None`) --
  Keyword arguments to be passed to `function`
- **num_proc** (`int` , *optional*, defaults to `None`) --
  The number of processes to use for multiprocessing.
  - If `None` or `0` , no multiprocessing is used and the operation runs in the main process.
  - If greater than `1` , one or multiple worker processes are used to process data in
    parallel.
    Note: The function passed to `map()` must be picklable for multiprocessing to work
    correctly
    (i.e., prefer functions defined at the top level of a module, not inside another function
    or class).
- **desc** (`str` , *optional*, defaults to `None`) --
  Meaningful description to be displayed alongside with the progress bar while filtering
  examples.0
  Apply a filter function to all the elements in the table in batches
  and update the table so that the dataset only includes examples according to the filter
  function.
  The transformation is applied to all the datasets of the dataset dictionary.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds.filter(lambda x: x["label"] == 1)
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 4265
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 533
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 533
    })
})
```

sortdatasets.DatasetDict.sorthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L1144[{"name": "column_names", "val": ": typing.Union[str, collections.abc.Sequence[str]]"}, {"name": "reverse", "val": ": typing.Union[bool, collections.abc.Sequence[bool]] = False"}, {"name": "null_placement", "val": ": str = 'at_end'"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "load_from_cache_file", "val": ": typing.Optional[bool] = None"}, {"name": "indices_cache_file_names", "val": ": typing.Optional[dict[str, typing.Optional[str]]] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] = 1000"}]- **column_names** (`Union[str, Sequence[str]]`) -- Column name(s) to sort by.

- **reverse** (`Union[bool, Sequence[bool]]` , defaults to `False`) -- If `True` , sort by descending order rather than ascending. If a single bool is provided, the value is applied to the sorting of all column names. Otherwise a list of bools with the same length and order as column_names must be provided.
- **null_placement** (`str` , defaults to `at_end`) -- Put `None` values at the beginning if `at_start` or `first` or at the end if `at_end` or `last`
- **keep_in_memory** (`bool` , defaults to `False`) -- Keep the sorted indices in memory instead of writing it to a cache file.
- **load_from_cache_file** (`Optional[bool]` , defaults to `True` if caching is enabled) --

If a cache file storing the sorted indices
can be identified, use it instead of recomputing.

- **indices_cache_file_names** (`[Dict[str, str]]` , *optional*, defaults to `None`) --
  Provide the name of a path for the cache file. It is used to store the
  indices mapping instead of the automatically generated cache file name.
  You have to provide one `cache_file_name` per dataset in the dataset dictionary.
- **writer_batch_size** (`int` , defaults to `1000`) --
  Number of rows per write operation for the cache file writer.
  Higher value gives smaller cache files, lower value consume less temporary memory.0
  Create a new dataset sorted according to a single or multiple columns.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset('cornell-movie-review-data/rotten_tomatoes')
>>> ds['train']['label'][:10]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> sorted_ds = ds.sort('label')
>>> sorted_ds['train']['label'][:10]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> another_sorted_ds = ds.sort(['label', 'text'], reverse=[True, False])
>>> another_sorted_ds['train']['label'][:10]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

shuffledatasets.DatasetDict.shufflehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L1211[{"name": "seeds", "val": ": typing.Union[int, dict[str, typing.Optional[int]], NoneType] = None"}, {"name": "seed", "val": ": typing.Optional[int] = None"}, {"name": "generators", "val": ": typing.Optional[dict[str, numpy.random._generator.Generator]] = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "load_from_cache_file", "val": ": typing.Optional[bool] = None"}, {"name": "indices_cache_file_names", "val": ": typing.Optional[dict[str, typing.Optional[str]]] = None"}, {"name": "writer_batch_size", "val": ": typing.Optional[int] = 1000"}]- **seeds** (`Dict[str, int]` or `int` , *optional*) --
A seed to initialize the default BitGenerator if `generator=None` .
If `None` , then fresh, unpredictable entropy will be pulled from the OS.
If an `int` or `array_like[ints]` is passed, then it will be passed to SeedSequence to derive

the initial BitGenerator state.
You can provide one `seed` per dataset in the dataset dictionary.

- **seed** (`int` , *optional*) --
  A seed to initialize the default BitGenerator if `generator=None` . Alias for seeds (a `ValueError` is raised if both are provided).
- **generators** (`Dict[str, *optional*, np.random.Generator]`) --
  Numpy random Generator to use to compute the permutation of the dataset rows.
  If `generator=None` (default), uses `np.random.default_rng` (the default BitGenerator (PCG64) of NumPy).
  You have to provide one `generator` per dataset in the dataset dictionary.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Keep the dataset in memory instead of writing it to a cache file.
- **load_from_cache_file** (`Optional[bool]` , defaults to `True` if caching is enabled) --
  If a cache file storing the current computation from `function`
  can be identified, use it instead of recomputing.
- **indices_cache_file_names** (`Dict[str, str]` , *optional*) --
  Provide the name of a path for the cache file. It is used to store the
  indices mappings instead of the automatically generated cache file name.
  You have to provide one `cache_file_name` per dataset in the dataset dictionary.
- **writer_batch_size** (`int` , defaults to `1000`) --
  Number of rows per write operation for the cache file writer.
  This value is a good trade-off between memory usage during the processing, and processing speed.
  Higher value makes the processing do fewer lookups, lower value consume less temporary memory while running `map` .0
  Create a new Dataset where the rows are shuffled.

The transformation is applied to all the datasets of the dataset dictionary.

Currently shuffling uses numpy random generators.
You can either supply a NumPy BitGenerator to use, or a seed to initiate NumPy's default random generator (PCG64).

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds["train"]["label"][:10]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

# set a seed
>>> shuffled_ds = ds.shuffle(seed=42)
>>> shuffled_ds["train"]["label"][:10]
[0, 1, 0, 1, 0, 0, 0, 0, 0, 0]
```

set_formatdatasets.DatasetDict.set_formathttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L575[{"name": "type", "val": ": typing.Optional[str] = None"}, {"name": "columns", "val": ": typing.Optional[list] = None"}, {"name": "output_all_columns", "val": ": bool = False"}, {"name": "**format_kwargs", "val": ""}]- **type** (`str` , *optional*) -- Either output type selected in `[None, 'numpy', 'torch', 'tensorflow', 'jax', 'arrow', 'pandas', 'polars']` . `None` means `__getitem__` returns python objects (default).

- **columns** (`list[str]` , *optional*) --
  Columns to format in the output.
  `None` means `__getitem__` returns all columns (default).
- **output_all_columns** (`bool` , defaults to False) --
  Keep un-formatted columns as well in the output (as python objects),
- ****format_kwargs** (additional keyword arguments) --
  Keywords arguments passed to the convert function like `np.array` , `torch.tensor` or `tensorflow.ragged.constant` .0
  Set `__getitem__` return format (type and columns).
  The format is set for every dataset in the dataset dictionary.

It is possible to call `map` after calling `set_format` . Since `map` may add new columns, then the list of formatted columns
gets updated. In this case, if you apply `map` on a dataset to add a new column, then this column will be formatted:

`new formatted columns = (all columns - previously unformatted columns)`

Example:

```
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer
>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
>>> ds = ds.map(lambda x: tokenizer(x["text"], truncation=True, padding=True), batched=True)
>>> ds.set_format(type="numpy", columns=['input_ids', 'token_type_ids', 'attention_mask', 'label']
>>> ds["train"].format
{'columns': ['input_ids', 'token_type_ids', 'attention_mask', 'label'],
 'format_kwargs': {},
 'output_all_columns': False,
 'type': 'numpy'}
```

reset_formatdatasets.DatasetDict.reset_formathttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L626[]

Reset `__getitem__` return format to python objects and all columns.

The transformation is applied to all the datasets of the dataset dictionary.

Same as `self.set_format()`

Example:

```
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
>>> ds = ds.map(lambda x: tokenizer(x["text"], truncation=True, padding=True), batched=True)
>>> ds.set_format(type="numpy", columns=['input_ids', 'token_type_ids', 'attention_mask', 'label']
>>> ds["train"].format
{'columns': ['input_ids', 'token_type_ids', 'attention_mask', 'label'],
 'format_kwargs': {},
 'output_all_columns': False,
 'type': 'numpy'}
>>> ds.reset_format()
>>> ds["train"].format
{'columns': ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
 'format_kwargs': {},
 'output_all_columns': False,
 'type': None}
```

formatted_asdatasets.DatasetDict.formatted_ashttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L535[{"name": "type", "val": ": typing.Optional[str] = None"}, {"name": "columns", "val": ": typing.Optional[list] = None"}, {"name": "output_all_columns", "val": ": bool = False"}, {"name": "**format_kwargs", "val": ""}]- **type** (`str`, *optional*) -- Either output type selected in `[None, 'numpy', 'torch', 'tensorflow', 'jax', 'arrow', 'pandas', 'polars']`. `None` means `__getitem__` returns python objects (default).

- **columns** (`list[str]`, *optional*) --
  Columns to format in the output.
  `None` means `__getitem__` returns all columns (default).
- **output_all_columns** (`bool`, defaults to False) --
  Keep un-formatted columns as well in the output (as python objects).
- ****format_kwargs** (additional keyword arguments) --
  Keywords arguments passed to the convert function like `np.array`, `torch.tensor` or `tensorflow.ragged.constant`.0
  To be used in a `with` statement. Set `__getitem__` return format (type and columns).
  The transformation is applied to all the datasets of the dataset dictionary.

with_formatdatasets.DatasetDict.with_formathttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L687[{"name": "type", "val": ": typing.Optional[str] = None"}, {"name": "columns", "val": ": typing.Optional[list] = None"}, {"name": "output_all_columns", "val": ": bool = False"}, {"name": "**format_kwargs", "val": ""}]- **type** (`str`, *optional*) -- Either output type selected in `[None, 'numpy', 'torch', 'tensorflow', 'jax', 'arrow', 'pandas', 'polars']`. `None` means `__getitem__` returns python objects (default).

- **columns** (`list[str]`, *optional*) --
  Columns to format in the output.
  `None` means `__getitem__` returns all columns (default).
- **output_all_columns** (`bool`, defaults to `False`) --
  Keep un-formatted columns as well in the output (as python objects).
- ****format_kwargs** (additional keyword arguments) --
  Keywords arguments passed to the convert function like `np.array`, `torch.tensor` or `tensorflow.ragged.constant`.0
  Set `__getitem__` return format (type and columns). The data formatting is applied on-the-

fly.
The format `type` (for example "numpy") is used to format batches when using `__getitem__`.
The format is set for every dataset in the dataset dictionary.

It's also possible to use custom transforms for formatting using with_transform().

Contrary to set_format(), `with_format` returns a new DatasetDict object with new Dataset objects.

Example:

```
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
>>> ds = ds.map(lambda x: tokenizer(x['text'], truncation=True, padding=True), batched=True)
>>> ds["train"].format
{'columns': ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
 'format_kwargs': {},
 'output_all_columns': False,
 'type': None}
>>> ds = ds.with_format("torch")
>>> ds["train"].format
{'columns': ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
 'format_kwargs': {},
 'output_all_columns': False,
 'type': 'torch'}
>>> ds["train"][0]
{'text': 'compassionately explores the seemingly irreconcilable situation between conservative chr
 'label': tensor(1),
 'input_ids': tensor([  101, 18027, 16310, 16001,  1103,  9321,   178, 11604,  7235,  6617,
        1742,  2165,  2820,  1206,  6588, 22572, 12937,  1811,  2153,  1105,
        1147, 12890, 19587,  6463,  1105, 15026,  1482,   119,   102,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0]),
 'token_type_ids': tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])}
```

with_transformdatasets.DatasetDict.with_transformhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L764[{"name": "transform", "val": ":

typing.Optional[typing.Callable]"}, {"name": "columns", "val": ": typing.Optional[list] = None"}, {"name": "output_all_columns", "val": ": bool = False"}]- **transform** (`Callable`, *optional*) -- User-defined formatting transform, replaces the format defined by set_format().
A formatting function is a callable that takes a batch (as a dict) as input and returns a batch. This function is applied right before returning the objects in `__getitem__`.

- **columns** (`list[str]`, *optional*) --
  Columns to format in the output.
  If specified, then the input batch of the transform only contains those columns.
- **output_all_columns** (`bool`, defaults to False) --
  Keep un-formatted columns as well in the output (as python objects).
  If set to `True`, then the other un-formatted columns are kept with the output of the transform.0
  Set `__getitem__` return format using this transform. The transform is applied on-the-fly on batches when `__getitem__` is called.
  The transform is set for every dataset in the dataset dictionary

As set_format(), this can be reset using reset_format().

Contrary to `set_transform()`, `with_transform` returns a new DatasetDict object with new Dataset objects.

Example:

```
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
>>> def encode(example):
...     return tokenizer(example['text'], truncation=True, padding=True, return_tensors="pt")
>>> ds = ds.with_transform(encode)
>>> ds["train"][0]
{'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1]),
 'input_ids': tensor([  101,  1103,  2067,  1110, 17348,  1106,  1129,  1103,  6880,  1432,
        112,   188,  1207,   107, 14255,  1389,   107,  1105,  1115,  1119,
        112,   188,  1280,  1106,  1294,   170, 24194,  1256,  3407,  1190,
        170, 11791,  5253,   188,  1732,  7200, 10947, 12606,  2895,   117,
        179,  7766,   118,   172, 15554,  1181,  3498,  6961,  3263,  1137,
        188,  1566,  7912, 14516,  6997,   119,   102]),
 'token_type_ids': tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0])}
```

flattendatasets.DatasetDict.flattenhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L197[{"name": "max_depth", "val": " = 16"}]
Flatten the Apache Arrow Table of each split (nested features are flatten).
Each column with a struct type is flattened into one column per struct field.
Other columns are left unchanged.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("rajpurkar/squad")
>>> ds["train"].features
{'id': Value('string'),
 'title': Value('string'),
 'context': Value('string'),
 'question': Value('string'),
 'answers.text': List(Value('string')),
 'answers.answer_start': List(Value('int32'))}
>>> ds.flatten()
DatasetDict({
    train: Dataset({
        features: ['id', 'title', 'context', 'question', 'answers.text', 'answers.answer_start'],
        num_rows: 87599
    })
    validation: Dataset({
        features: ['id', 'title', 'context', 'question', 'answers.text', 'answers.answer_start'],
        num_rows: 10570
    })
})
```

castdatasets.DatasetDict.casthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L278[{"name": "features", "val": ": Features"}]- **features** (Features) --
New features to cast the dataset to.
The name and order of the fields in the features must match the current column names.
The type of the data must also be convertible from one type to the other.
For non-trivial conversion, e.g. `string` <-> `ClassLabel` you should use map() to update the dataset.0

Cast the dataset to a new set of features.
The transformation is applied to all the datasets of the dataset dictionary.

Example:

```
>>> from datasets import load_dataset, ClassLabel, Value
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds["train"].features
{'label': ClassLabel(names=['neg', 'pos']),
 'text': Value('string')}
>>> new_features = ds["train"].features.copy()
>>> new_features['label'] = ClassLabel(names=['bad', 'good'])
>>> new_features['text'] = Value('large_string')
>>> ds = ds.cast(new_features)
>>> ds["train"].features
{'label': ClassLabel(names=['bad', 'good']),
 'text': Value('large_string')}
```

cast_columndatasets.DatasetDict.cast_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L310[{"name": "column", "val": ": str"}, {"name": "feature", "val": ""}]- **column** (`str`) --
Column name.

- **feature** (`Feature`) --
  Target feature.0DatasetDict
  Cast column to feature for decoding.

Example:

```
>>> from datasets import load_dataset, ClassLabel
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds["train"].features
{'label': ClassLabel(names=['neg', 'pos']),
 'text': Value('string')}
>>> ds = ds.cast_column('label', ClassLabel(names=['bad', 'good']))
>>> ds["train"].features
{'label': ClassLabel(names=['bad', 'good']),
 'text': Value('string')}
```

remove_columnsdatasets.DatasetDict.remove_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L339[{"name": "column_names", "val": ": typing.Union[str, list[str]]"}]- **column_names** (`Union[str, list[str]]`) --
Name of the column(s) to remove.0DatasetDictA copy of the dataset object without the

columns to remove.

Remove one or several column(s) from each split in the dataset
and the features associated to the column(s).

The transformation is applied to all the splits of the dataset dictionary.

You can also remove a column using map() with `remove_columns` but the present method doesn't copy the data of the remaining columns and is thus faster.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds = ds.remove_columns("label")
DatasetDict({
    train: Dataset({
        features: ['text'],
        num_rows: 8530
    })
    validation: Dataset({
        features: ['text'],
        num_rows: 1066
    })
    test: Dataset({
        features: ['text'],
        num_rows: 1066
    })
})
```

rename_columndatasets.DatasetDict.rename_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L381[{"name": "original_column_name", "val": ": str"}, {"name": "new_column_name", "val": ": str"}]- **original_column_name** (`str`) --
Name of the column to rename.

- **new_column_name** (`str`) --
  New name for the column.0

Rename a column in the dataset and move the features associated to the original column

under the new column name.
The transformation is applied to all the datasets of the dataset dictionary.

You can also rename a column using map() with `remove_columns` but the present method:

- takes care of moving the original features under the new column name.
- doesn't copy the data to a new dataset and is thus much faster.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds = ds.rename_column("label", "label_new")
DatasetDict({
    train: Dataset({
        features: ['text', 'label_new'],
        num_rows: 8530
    })
    validation: Dataset({
        features: ['text', 'label_new'],
        num_rows: 1066
    })
    test: Dataset({
        features: ['text', 'label_new'],
        num_rows: 1066
    })
})
```

rename_columnsdatasets.DatasetDict.rename_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L429[{"name": "column_mapping", "val": ": dict"}]- **column_mapping** (`Dict[str, str]`) --
A mapping of columns to rename to their new names.0DatasetDictA copy of the dataset with renamed columns.

Rename several columns in the dataset, and move the features associated to the original columns under
the new column names.
The transformation is applied to all the datasets of the dataset dictionary.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds.rename_columns({'text': 'text_new', 'label': 'label_new'})
DatasetDict({
    train: Dataset({
        features: ['text_new', 'label_new'],
        num_rows: 8530
    })
    validation: Dataset({
        features: ['text_new', 'label_new'],
        num_rows: 1066
    })
    test: Dataset({
        features: ['text_new', 'label_new'],
        num_rows: 1066
    })
})
```

select_columnsdatasets.DatasetDict.select_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L467[{"name": "column_names", "val": ": typing.Union[str, list[str]]"}]- **column_names** (`Union[str, list[str]]`) -- Name of the column(s) to keep.0
Select one or several column(s) from each split in the dataset and the features associated to the column(s).

The transformation is applied to all the splits of the dataset dictionary.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes")
>>> ds.select_columns("text")
DatasetDict({
    train: Dataset({
        features: ['text'],
        num_rows: 8530
    })
    validation: Dataset({
        features: ['text'],
        num_rows: 1066
    })
    test: Dataset({
        features: ['text'],
        num_rows: 1066
    })
})
```

class_encode_columndatasets.DatasetDict.class_encode_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L503[{"name": "column", "val": ": str"}, {"name": "include_nulls", "val": ": bool = False"}]- **column** (`str`) --
The name of the column to cast.

- **include_nulls** (`bool` , defaults to `False`) --
  Whether to include null values in the class labels. If `True` , the null values will be encoded as the `"None"` class label.
  0
  Casts the given column as ClassLabel and updates the tables.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("boolq")
>>> ds["train"].features
{'answer': Value('bool'),
 'passage': Value('string'),
 'question': Value('string')}
>>> ds = ds.class_encode_column("answer")
>>> ds["train"].features
{'answer': ClassLabel(num_classes=2, names=['False', 'True']),
 'passage': Value('string'),
 'question': Value('string')}
```

push_to_hubdatasets.DatasetDict.push_to_hubhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L1616[{"name": "repo_id", "val": ""}, {"name": "config_name", "val": ": str = 'default'"}, {"name": "set_default", "val": ": typing.Optional[bool] = None"}, {"name": "data_dir", "val": ": typing.Optional[str] = None"}, {"name": "commit_message", "val": ": typing.Optional[str] = None"}, {"name": "commit_description", "val": ": typing.Optional[str] = None"}, {"name": "private", "val": ": typing.Optional[bool] = None"}, {"name": "token", "val": ": typing.Optional[str] = None"}, {"name": "revision", "val": ": typing.Optional[str] = None"}, {"name": "create_pr", "val": ": typing.Optional[bool] = False"}, {"name": "max_shard_size", "val": ": typing.Union[str, int, NoneType] = None"}, {"name": "num_shards", "val": ": typing.Optional[dict[str, int]] = None"}, {"name": "embed_external_files", "val": ": bool = True"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}]- **repo_id** (`str`) --
The ID of the repository to push to in the following format: `<user>/<dataset_name>` or `<org>/<dataset_name>`. Also accepts `<dataset_name>`, which will default to the namespace of the logged-in user.

- **config_name** (`str`) --
  Configuration name of a dataset. Defaults to "default".
- **set_default** (`bool` , *optional*) --
  Whether to set this configuration as the default one. Otherwise, the default configuration is the one
  named "default".
- **data_dir** (`str` , *optional*) --
  Directory name that will contain the uploaded data files. Defaults to the `config_name` if

different
from "default", else "data".

- **commit_message** (`str` , *optional*) --
  Message to commit while pushing. Will default to `"Upload dataset"` .
- **commit_description** (`str` , *optional*) --
  Description of the commit that will be created.
  Additionally, description of the PR if a PR is created (`create_pr` is True).
- **private** (`bool` , *optional*) --
  Whether to make the repo private. If `None` (default), the repo will be public unless the organization's default is private. This value is ignored if the repo already exists.
- **token** (`str` , *optional*) --
  An optional authentication token for the Hugging Face Hub. If no token is passed, will default
  to the token saved locally when logging in with `huggingface-cli login` . Will raise an error if no token is passed and the user is not logged-in.
- **revision** (`str` , *optional*) --
  Branch to push the uploaded files to. Defaults to the `"main"` branch.
- **create_pr** (`bool` , *optional*, defaults to `False`) --
  Whether to create a PR with the uploaded files or directly commit.
- **max_shard_size** (`int` or `str` , *optional*, defaults to `"500MB"`) --
  The maximum size of the dataset shards to be uploaded to the hub. If expressed as a string, needs to be digits followed by a unit
  (like `"500MB"` or `"1GB"`).
- **num_shards** (`Dict[str, int]` , *optional*) --
  Number of shards to write. By default, the number of shards depends on `max_shard_size` .
  Use a dictionary to define a different num_shards for each split.
- **embed_external_files** (`bool` , defaults to `True`) --
  Whether to embed file bytes in the shards.
  In particular, this will do the following before the push for the fields of type:
  - Audio and Image removes local path information and embed file content in the Parquet files.
- **num_proc** (`int` , *optional*, defaults to `None`) --
  Number of processes when preparing and uploading the dataset.
  This is helpful if the dataset is made of many samples or media files to embed.
  Multiprocessing is disabled by default.

0huggingface_hub.CommitInfo

Pushes the DatasetDict to the hub as a Parquet dataset.

The DatasetDict is pushed using HTTP requests and does not need to have neither git or git-lfs installed.

Each dataset split will be pushed independently. The pushed dataset will keep the original split names.

The resulting Parquet files are self-contained by default: if your dataset contains Image or Audio
data, the Parquet files will store the bytes of your images or audio files.
You can disable this by setting `embed_external_files` to False.

Example:

```
>>> dataset_dict.push_to_hub("<organization>/<dataset_id>")
>>> dataset_dict.push_to_hub("<organization>/<dataset_id>", private=True)
>>> dataset_dict.push_to_hub("<organization>/<dataset_id>", max_shard_size="1GB")
>>> dataset_dict.push_to_hub("<organization>/<dataset_id>", num_shards={"train": 1024, "test": 8})
```

If you want to add a new configuration (or subset) to a dataset (e.g. if the dataset has multiple tasks/versions/languages):

```
>>> english_dataset.push_to_hub("<organization>/<dataset_id>", "en")
>>> french_dataset.push_to_hub("<organization>/<dataset_id>", "fr")
>>> # later
>>> english_dataset = load_dataset("<organization>/<dataset_id>", "en")
>>> french_dataset = load_dataset("<organization>/<dataset_id>", "fr")
```

save_to_diskdatasets.DatasetDict.save_to_diskhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L1294[{"name": "dataset_dict_path", "val": ": typing.Union[str, bytes, os.PathLike]"}, {"name": "max_shard_size", "val": ": typing.Union[str, int, NoneType] = None"}, {"name": "num_shards", "val": ": typing.Optional[dict[str, int]] = None"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}]- **dataset_dict_path** (`path-like`) -- Path (e.g. `dataset/train`) or remote URI (e.g. `s3://my-bucket/dataset/train`)
of the dataset dict directory where the dataset dict will be saved to.

- **max_shard_size** (`int` or `str` , *optional*, defaults to `"500MB"`) --
  The maximum size of the dataset shards to be saved to the filesystem. If expressed as a string, needs to be digits followed by a unit
  (like `"50MB"`).
- **num_shards** (`Dict[str, int]` , *optional*) --
  Number of shards to write. By default the number of shards depends on `max_shard_size` and `num_proc` .
  You need to provide the number of shards for each dataset in the dataset dictionary.
  Use a dictionary to define a different num_shards for each split.
- **num_proc** (`int` , *optional*, default `None`) --
  Number of processes when downloading and generating the dataset locally.
  Multiprocessing is disabled by default.
- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
  0

Saves a dataset dict to a filesystem using `fsspec.spec.AbstractFileSystem` .

For Image, Audio and Video data:

All the Image(), Audio() and Video() data are stored in the arrow files.
If you want to store paths or urls, please use the Value("string") type.

Example:

```
>>> dataset_dict.save_to_disk("path/to/dataset/directory")
>>> dataset_dict.save_to_disk("path/to/dataset/directory", max_shard_size="1GB")
>>> dataset_dict.save_to_disk("path/to/dataset/directory", num_shards={"train": 1024, "test": 8})
```

load_from_diskdatasets.DatasetDict.load_from_diskhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L1368[{"name": "dataset_dict_path", "val": ": typing.Union[str, bytes, os.PathLike]"}, {"name": "keep_in_memory", "val": ": typing.Optional[bool] = None"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}]- **dataset_dict_path** (`path-like`) --
Path (e.g. `"dataset/train"`) or remote URI (e.g. `"s3//my-bucket/dataset/train"`)
of the dataset dict directory where the dataset dict will be loaded from.

- **keep_in_memory** (`bool` , defaults to `None`) --
  Whether to copy the dataset in-memory. If `None` , the
  dataset will not be copied in-memory unless explicitly enabled by setting `datasets.config.IN_MEMORY_MAX_SIZE` to nonzero. See more details in the
  improve performance section.
- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
  0DatasetDict

Load a dataset that was previously saved using `save_to_disk` from a filesystem using `fsspec.spec.AbstractFileSystem` .

Example:

```
>>> ds = load_from_disk('path/to/dataset/directory')
```

from_csvdatasets.DatasetDict.from_csvhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L1428[{"name": "path_or_paths", "val": ": dict"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "cache_dir", "val": ": str = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "**kwargs", "val": ""}]- **path_or_paths** (`dict` of path-like) --
Path(s) of the CSV file(s).

- **features** (Features, *optional*) --
  Dataset features.
- **cache_dir** (str, *optional*, defaults to `"~/.cache/huggingface/datasets"`) --
  Directory to cache data.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Whether to copy the data in-memory.
- ****kwargs** (additional keyword arguments) --
  Keyword arguments to be passed to `pandas.read_csv` .0DatasetDict
  Create DatasetDict from CSV file(s).

Example:

```
>>> from datasets import DatasetDict
>>> ds = DatasetDict.from_csv({'train': 'path/to/dataset.csv'})
```

from_jsondatasets.DatasetDict.from_jsonhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L1471[{"name": "path_or_paths", "val": ": dict"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "cache_dir", "val": ": str = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "**kwargs", "val": ""}]- **path_or_paths** (`path-like` or list of `path-like`) --
Path(s) of the JSON Lines file(s).

- **features** (Features, *optional*) --
  Dataset features.
- **cache_dir** (str, *optional*, defaults to `"~/.cache/huggingface/datasets"`) --
  Directory to cache data.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Whether to copy the data in-memory.
- ****kwargs** (additional keyword arguments) --
  Keyword arguments to be passed to `JsonConfig` .0DatasetDict
  Create DatasetDict from JSON Lines file(s).

Example:

```
>>> from datasets import DatasetDict
>>> ds = DatasetDict.from_json({'train': 'path/to/dataset.json'})
```

from_parquetdatasets.DatasetDict.from_parquethttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L1514[{"name": "path_or_paths", "val": ": dict"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "cache_dir", "val": ": str = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "columns", "val": ": typing.Optional[list[str]] = None"}, {"name": "**kwargs", "val": ""}]- **path_or_paths** (`dict` of path-like) --
Path(s) of the CSV file(s).

- **features** (Features, *optional*) --
  Dataset features.
- **cache_dir** (`str` , *optional*, defaults to `"~/.cache/huggingface/datasets"`) --

Directory to cache data.

- **keep_in_memory** (`bool` , defaults to `False`) --
  Whether to copy the data in-memory.
- **columns** (`list[str]` , *optional*) --
  If not `None` , only these columns will be read from the file.
  A column name may be a prefix of a nested field, e.g. 'a' will select
  'a.b', 'a.c', and 'a.d.e'.
- ****kwargs** (additional keyword arguments) --
  Keyword arguments to be passed to `ParquetConfig` .0DatasetDict
  Create DatasetDict from Parquet file(s).

Example:

```
>>> from datasets import DatasetDict
>>> ds = DatasetDict.from_parquet({'train': 'path/to/dataset/parquet'})
```

from_textdatasets.DatasetDict.from_texthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L1563[{"name": "path_or_paths", "val": ": dict"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "cache_dir", "val": ": str = None"}, {"name": "keep_in_memory", "val": ": bool = False"}, {"name": "**kwargs", "val": ""}]- **path_or_paths** (`dict` of path-like) --
Path(s) of the text file(s).

- **features** (Features, *optional*) --
  Dataset features.
- **cache_dir** (`str` , *optional*, defaults to `"~/.cache/huggingface/datasets"`) --
  Directory to cache data.
- **keep_in_memory** (`bool` , defaults to `False`) --
  Whether to copy the data in-memory.
- ****kwargs** (additional keyword arguments) --
  Keyword arguments to be passed to `TextConfig` .0DatasetDict
  Create DatasetDict from text file(s).

Example:

```
>>> from datasets import DatasetDict
>>> ds = DatasetDict.from_text({'train': 'path/to/dataset.txt'})
```

IterableDatasetdatasets.IterableDataset

The base class IterableDataset implements an iterable Dataset backed by python generators.

class datasets.IterableDatasetdatasets.IterableDatasethttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L2125[{"name": "ex_iterable", "val": ": _BaseExamplesIterable"}, {"name": "info", "val": ": typing.Optional[datasets.info.DatasetInfo] = None"}, {"name": "split", "val": ": typing.Optional[datasets.splits.NamedSplit] = None"}, {"name": "formatting", "val": ": typing.Optional[datasets.iterable_dataset.FormattingConfig] = None"}, {"name": "shuffling", "val": ": typing.Optional[datasets.iterable_dataset.ShufflingConfig] = None"}, {"name": "distributed", "val": ": typing.Optional[datasets.iterable_dataset.DistributedConfig] = None"}, {"name": "token_per_repo_id", "val": ": typing.Optional[dict[str, typing.Union[str, bool, NoneType]]] = None"}]
A Dataset backed by an iterable.

from_generatordatasets.IterableDataset.from_generatorhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L2522[{"name": "generator", "val": ": typing.Callable"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "gen_kwargs", "val": ": typing.Optional[dict] = None"}, {"name": "split", "val": ": NamedSplit = NamedSplit('train')"}]- **generator** (`Callable`) --
A generator function that `yields` examples.

- **features** (`Features` , *optional*) --
  Dataset features.
- **gen_kwargs(** `dict` **,** *optional*) --
  Keyword arguments to be passed to the `generator` callable.
  You can define a sharded iterable dataset by passing the list of shards in `gen_kwargs` .
  This can be used to improve shuffling and when iterating over the dataset with multiple workers.
- **split** (NamedSplit, defaults to `Split.TRAIN`) --

Split name to be assigned to the dataset.

0 `IterableDataset`

Create an Iterable Dataset from a generator.

Example:

```
>>> def gen():
...     yield {"text": "Good", "label": 0}
...     yield {"text": "Bad", "label": 1}
...
>>> ds = IterableDataset.from_generator(gen)
```

```
>>> def gen(shards):
...     for shard in shards:
...         with open(shard) as f:
...             for line in f:
...                 yield {"line": line}
...
>>> shards = [f"data{i}.txt" for i in range(32)]
>>> ds = IterableDataset.from_generator(gen, gen_kwargs={"shards": shards})
>>> ds = ds.shuffle(seed=42, buffer_size=10_000)  # shuffles the shards order + uses a shuffle buf
>>> from torch.utils.data import DataLoader
>>> dataloader = DataLoader(ds.with_format("torch"), num_workers=4)  # give each worker a subset o
```

remove_columnsdatasets.IterableDataset.remove_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3254[{"name": "column_names", "val": ": typing.Union[str, list[str]]"}]- **column_names** (`Union[str, List[str]]`) --

Name of the column(s) to remove.0 `IterableDataset` A copy of the dataset object without the columns to remove.

Remove one or several column(s) in the dataset and the features associated to them.
The removal is done on-the-fly on the examples when iterating over the dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train", streaming=True)
>>> next(iter(ds))
{'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
>>> ds = ds.remove_columns("label")
>>> next(iter(ds))
{'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
```

select_columnsdatasets.IterableDataset.select_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3289[{"name": "column_names", "val": ": typing.Union[str, list[str]]"}]- **column_names** (`Union[str, List[str]]`) --
Name of the column(s) to select.0 `IterableDataset` A copy of the dataset object with selected columns.
Select one or several column(s) in the dataset and the features
associated to them. The selection is done on-the-fly on the examples
when iterating over the dataset.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train", streaming=True)
>>> next(iter(ds))
{'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
>>> ds = ds.select_columns("text")
>>> next(iter(ds))
{'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
```

cast_columndatasets.IterableDataset.cast_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3340[{"name": "column", "val": ": str"}, {"name": "feature", "val": ": typing.Union[dict, list, tuple, datasets.features.features.Value, datasets.features.features.ClassLabel, datasets.features.translation.Translation, datasets.features.translation.TranslationVariableLanguages, datasets.features.features.LargeList, datasets.features.features.List, datasets.features.features.Array2D, datasets.features.features.Array3D, datasets.features.features.Array4D, datasets.features.features.Array5D, datasets.features.audio.Audio, datasets.features.image.Image, datasets.features.video.Video,

datasets.features.pdf.Pdf]"}]- **column** (`str`) --
Column name.

- **feature** (`Feature`) --
  Target feature.0 `IterableDataset`
  Cast column to feature for decoding.

Example:

```
>>> from datasets import load_dataset, Audio
>>> ds = load_dataset("PolyAI/minds14", name="en-US", split="train", streaming=True)
>>> ds.features
{'audio': Audio(sampling_rate=8000, mono=True, decode=True, id=None),
 'english_transcription': Value('string'),
 'intent_class': ClassLabel(num_classes=14, names=['abroad', 'address', 'app_error', 'atm_limit',
 'lang_id': ClassLabel(num_classes=14, names=['cs-CZ', 'de-DE', 'en-AU', 'en-GB', 'en-US', 'es-ES'
 'path': Value('string'),
 'transcription': Value('string')}
>>> ds = ds.cast_column("audio", Audio(sampling_rate=16000))
>>> ds.features
{'audio': Audio(sampling_rate=16000, mono=True, decode=True, id=None),
 'english_transcription': Value('string'),
 'intent_class': ClassLabel(num_classes=14, names=['abroad', 'address', 'app_error', 'atm_limit',
 'lang_id': ClassLabel(num_classes=14, names=['cs-CZ', 'de-DE', 'en-AU', 'en-GB', 'en-US', 'es-ES'
 'path': Value('string'),
 'transcription': Value('string')}
```

castdatasets.IterableDataset.casthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3387[{"name": "features", "val": ": Features"}]- **features** (Features) --
New features to cast the dataset to.
The name of the fields in the features must match the current column names.
The type of the data must also be convertible from one type to the other.
For non-trivial conversion, e.g. `string` <-> `ClassLabel` you should use map() to update the Dataset.0 `IterableDataset` A copy of the dataset with casted features.

Cast the dataset to a new set of features.

Example:

```
>>> from datasets import load_dataset, ClassLabel, Value
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train", streaming=True)
>>> ds.features
{'label': ClassLabel(names=['neg', 'pos']),
 'text': Value('string')}
>>> new_features = ds.features.copy()
>>> new_features["label"] = ClassLabel(names=["bad", "good"])
>>> new_features["text"] = Value("large_string")
>>> ds = ds.cast(new_features)
>>> ds.features
{'label': ClassLabel(names=['bad', 'good']),
 'text': Value('large_string')}
```

decodedatasets.IterableDataset.decodehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3434[{"name": "enable", "val": ": bool = True"}, {"name": "num_threads", "val": ": int = 0"}]- **enable** (`bool` , defaults to `True`) --
Enable or disable features decoding.

- **num_threads** (`int` , defaults to `0`) --
  Enable multithreading for features decoding.0 `IterableDataset` A copy of the dataset with casted features.

Enable or disable the dataset features decoding for audio, image, video.

When enabled (default), media types are decoded:

- audio -> dict of "array" and "sampling_rate" and "path"
- image -> PIL.Image
- video -> torchvision.io.VideoReader

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.

Disabling decoding is useful if you want to iterate on the paths or bytes of the media files without actually decoding their content. To disable decoding you can use `.decode(False)` , which

is equivalent to calling `.cast()` or `.cast_column()` with all the Audio, Image and Video types set to `decode=False`.

Examples:

Disable decoding:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("sshh12/planet-textures", split="train", streaming=True)
>>> next(iter(ds))
{'image': <PIL.PngImagePlugin.PngImageFile image mode=RGB size=2048x1024>,
'text': 'A distant celestial object with an icy crust, displaying a light blue shade, covered with
>>> ds = ds.decode(False)
>>> ds.features
{'image': Image(mode=None, decode=False, id=None),
'text': Value('string')}
>>> next(iter(ds))
{
  'image': {
    'path': 'hf://datasets/sshh12/planet-textures@69dc4cef7a5c4b2cfe387727ec8ea73d4bff7302/train/t
    'bytes': None
  },
  'text': 'A distant celestial object with an icy crust, displaying a light blue shade, covered wi
}
```

Speed up streaming with multithreading:

```
>>> import os
>>> from datasets import load_dataset
>>> from tqdm import tqdm
>>> ds = load_dataset("sshh12/planet-textures", split="train", streaming=True)
>>> num_threads = min(32, (os.cpu_count() or 1) + 4)
>>> ds = ds.decode(num_threads=num_threads)
>>> for _ in tqdm(ds):  # 20 times faster !
...     ...
```

iterdatasets.IterableDataset.**iter**https://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L2459[]

iterdatasets.IterableDataset.iterhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L2484[{"name": "batch_size", "val": ": int"}, {"name": "drop_last_batch", "val": ": bool = False"}]- **batch_size** (`int`) -- size of each batch to yield.

- **drop_last_batch** (`bool` , default *False*) -- Whether a last batch smaller than the batch_size should be
  dropped0
  Iterate through the batches of size *batch_size*.

mapdatasets.IterableDataset.maphttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L2694[{"name": "function", "val": ": typing.Optional[typing.Callable] = None"}, {"name": "with_indices", "val": ": bool = False"}, {"name": "input_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "batched", "val": ": bool = False"}, {"name": "batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "drop_last_batch", "val": ": bool = False"}, {"name": "remove_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "features", "val": ": typing.Optional[datasets.features.features.Features] = None"}, {"name": "fn_kwargs", "val": ": typing.Optional[dict] = None"}]- **function** (`Callable` , *optional*, defaults to `None`) --
Function applied on-the-fly on the examples when you iterate on the dataset.
It must have one of the following signatures:

- `function(example: Dict[str, Any]) -> Dict[str, Any]` if `batched=False` and `with_indices=False`
- `function(example: Dict[str, Any], idx: int) -> Dict[str, Any]` if `batched=False` and `with_indices=True`
- `function(batch: Dict[str, List]) -> Dict[str, List]` if `batched=True` and `with_indices=False`
- `function(batch: Dict[str, List], indices: List[int]) -> Dict[str, List]` if `batched=True` and `with_indices=True`

For advanced usage, the function can also return a `pyarrow.Table` .
If the function is asynchronous, then `map` will run your function in parallel.
Moreover if your function returns nothing (`None`), then `map` will run your function and return the dataset unchanged.
If no function is provided, default to identity function: `lambda x: x` .

- **with_indices** (`bool` , defaults to `False`) --

Provide example indices to `function`. Note that in this case the signature of `function` should be `def function(example, idx[, rank]): ...`.

- **input_columns** (`Optional[Union[str, List[str]]]`, defaults to `None`) --
  The columns to be passed into `function`
  as positional arguments. If `None`, a dict mapping to all formatted columns is passed as one argument.
- **batched** (`bool`, defaults to `False`) --
  Provide batch of examples to `function`.
- **batch_size** (`int`, *optional*, defaults to `1000`) --
  Number of examples per batch provided to `function` if `batched=True`.
  `batch_size <= 0` or `batch_size == None` then provide the full dataset as a single batch to `function`.
- **drop_last_batch** (`bool`, defaults to `False`) --
  Whether a last batch smaller than the batch_size should be
  dropped instead of being processed by the function.
- **remove_columns** (`[List[str]]`, *optional*, defaults to `None`) --
  Remove a selection of columns while doing the mapping.
  Columns will be removed before updating the examples with the output of `function`, i.e. if `function` is adding
  columns with names in `remove_columns`, these columns will be kept.
- **features** (`[Features]`, *optional*, defaults to `None`) --
  Feature types of the resulting dataset.
- **fn_kwargs** (`Dict`, *optional*, default `None`) --
  Keyword arguments to be passed to `function`.0

Apply a function to all the examples in the iterable dataset (individually or in batches) and update them.
If your function returns a column that already exists, then it overwrites it.
The function is applied on-the-fly on the examples when iterating over the dataset.

You can specify whether the function should be batched or not with the `batched` parameter:

- If batched is `False`, then the function takes 1 example in and should return 1 example.
  An example is a dictionary, e.g. `{"text": "Hello there !"}`.
- If batched is `True` and `batch_size` is 1, then the function takes a batch of 1 example as input and can return a batch with 1 or more examples.

A batch is a dictionary, e.g. a batch of 1 example is {"text": ["Hello there !"]}.

- If batched is `True` and `batch_size` is `n` > 1, then the function takes a batch of `n` examples as input and can return a batch with `n` examples, or with an arbitrary number of examples.
  Note that the last batch may have less than `n` examples.
  A batch is a dictionary, e.g. a batch of `n` examples is `{"text": ["Hello there !"] * n}`.

If the function is asynchronous, then `map` will run your function in parallel, with up to one thousand simulatenous calls.
It is recommended to use a `asyncio.Semaphore` in your function if you want to set a maximum number of operations that can run at the same time.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train", streaming=True)
>>> def add_prefix(example):
...     example["text"] = "Review: " + example["text"]
...     return example
>>> ds = ds.map(add_prefix)
>>> list(ds.take(3))
[{'label': 1,
 'text': 'Review: the rock is destined to be the 21st century's new " conan " and that he's going
 {'label': 1,
 'text': 'Review: the gorgeously elaborate continuation of " the lord of the rings " trilogy is so
 {'label': 1, 'text': 'Review: effective but too-tepid biopic'}]
```

rename_columndatasets.IterableDataset.rename_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3199[{"name": "original_column_name", "val": ": str"}, {"name": "new_column_name", "val": ": str"}]- **original_column_name** (`str`) -- Name of the column to rename.

- **new_column_name** (`str`) --
  New name for the column.0 `IterableDataset` A copy of the dataset with a renamed column.

Rename a column in the dataset, and move the features associated to the original column under the new column

name.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train", streaming=True)
>>> next(iter(ds))
{'label': 1,
 'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
>>> ds = ds.rename_column("text", "movie_review")
>>> next(iter(ds))
{'label': 1,
 'movie_review': 'the rock is destined to be the 21st century's new " conan " and that he's going
```

filterdatasets.IterableDataset.filterhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L2846[{"name": "function", "val": ": typing.Optional[typing.Callable] = None"}, {"name": "with_indices", "val": " = False"}, {"name": "input_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "batched", "val": ": bool = False"}, {"name": "batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "fn_kwargs", "val": ": typing.Optional[dict] = None"}]- **function** (`Callable`) --
Callable with one of the following signatures:

- `function(example: Dict[str, Any]) -> bool` if `with_indices=False, batched=False`
- `function(example: Dict[str, Any], indices: int) -> bool` if `with_indices=True, batched=False`
- `function(example: Dict[str, List]) -> List[bool]` if `with_indices=False, batched=True`
- `function(example: Dict[str, List], indices: List[int]) -> List[bool]` if `with_indices=True, batched=True`

If the function is asynchronous, then `filter` will run your function in parallel.
If no function is provided, defaults to an always True function: `lambda x: True`.

- **with_indices** (`bool` , defaults to `False`) --
  Provide example indices to `function` . Note that in this case the signature of `function` should be `def function(example, idx): ...` .
- **input_columns** (`str` or `List[str]` , *optional*) --
  The columns to be passed into `function` as

positional arguments. If `None`, a dict mapping to all formatted columns is passed as one argument.

- **batched** (`bool`, defaults to `False`) --
  Provide batch of examples to `function`.
- **batch_size** (`int`, *optional*, default `1000`) --
  Number of examples per batch provided to `function` if `batched=True`.
- **fn_kwargs** (`Dict`, *optional*, default `None`) --
  Keyword arguments to be passed to `function`.0
  Apply a filter function to all the elements so that the dataset only includes examples according to the filter function.
  The filtering is done on-the-fly when iterating over the dataset.

If the function is asynchronous, then `filter` will run your function in parallel, with up to one thousand simulatenous calls (configurable).
It is recommended to use a `asyncio.Semaphore` in your function if you want to set a maximum number of operations that can run at the same time.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train", streaming=True)
>>> ds = ds.filter(lambda x: x["label"] == 0)
>>> list(ds.take(3))
[{'label': 0, 'movie_review': 'simplistic , silly and tedious .'},
 {'label': 0,
 'movie_review': "it's so laddish and juvenile , only teenage boys could possibly find it funny ."
 {'label': 0,
 'movie_review': 'exploitative and largely devoid of the depth or sophistication that would make w
```

shuffledatasets.IterableDataset.shufflehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L2932[{"name": "seed", "val": " = None"}, {"name": "generator", "val": ": typing.Optional[numpy.random._generator.Generator] = None"}, {"name": "buffer_size", "val": ": int = 1000"}]- **seed** (`int`, *optional*, defaults to `None`) --
Random seed that will be used to shuffle the dataset.
It is used to sample from the shuffle buffer and also to shuffle the data shards.

- **generator** (`numpy.random.Generator`, *optional*) --

Numpy random Generator to use to compute the permutation of the dataset rows.
If `generator=None` (default), uses `np.random.default_rng` (the default BitGenerator (PCG64) of NumPy).

- **buffer_size** (`int` , defaults to `1000`) --
  Size of the buffer.0

Randomly shuffles the elements of this dataset.

This dataset fills a buffer with `buffer_size` elements, then randomly samples elements from this buffer,
replacing the selected elements with new elements. For perfect shuffling, a buffer size greater than or
equal to the full size of the dataset is required.

For instance, if your dataset contains 10,000 elements but `buffer_size` is set to 1000, then `shuffle` will
initially select a random element from only the first 1000 elements in the buffer. Once an element is
selected, its space in the buffer is replaced by the next (i.e. 1,001-st) element,
maintaining the 1000 element buffer.

If the dataset is made of several shards, it also does shuffle the order of the shards.
However if the order has been fixed by using skip() or take()
then the order of the shards is kept unchanged.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train", streaming=True)
>>> list(ds.take(3))
[{'label': 1,
 'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
 {'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
 {'label': 1, 'text': 'effective but too-tepid biopic'}]
>>> shuffled_ds = ds.shuffle(seed=42)
>>> list(shuffled_ds.take(3))
[{'label': 1,
 'text': "a sports movie with action that's exciting on the field and a story you care about off i
 {'label': 1,
 'text': 'at its best , the good girl is a refreshingly adult take on adultery . . .'},
 {'label': 1,
 'text': "sam jones became a very lucky filmmaker the day wilco got dropped from their record labe
```

batchdatasets.IterableDataset.batchhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3558[{"name": "batch_size", "val": ": int"}, {"name": "drop_last_batch", "val": ": bool = False"}]- **batch_size** (`int`) -- The number of samples in each batch.

- **drop_last_batch** (`bool` , defaults to `False`) -- Whether to drop the last incomplete batch.0

Group samples from the dataset into batches.

Example:

```
>>> ds = load_dataset("some_dataset", streaming=True)
>>> batched_ds = ds.batch(batch_size=32)
```

skipdatasets.IterableDataset.skiphttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3006[{"name": "n", "val": ": int"}]- **n** (`int`) -- Number of elements to skip.0

Create a new IterableDataset that skips the first `n` elements.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train", streaming=True)
>>> list(ds.take(3))
[{'label': 1,
 'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
 {'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
 {'label': 1, 'text': 'effective but too-tepid biopic'}]
>>> ds = ds.skip(1)
>>> list(ds.take(3))
[{'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
 {'label': 1, 'text': 'effective but too-tepid biopic'},
 {'label': 1,
 'text': 'if you sometimes like to go to the movies to have fun , wasabi is a good place to start
```

takedatasets.IterableDataset.takehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3093[{"name": "n", "val": ": int"}]- **n** (`int`) -- Number of elements to take.0

Create a new IterableDataset with only the first `n` elements.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train", streaming=True)
>>> small_ds = ds.take(2)
>>> list(small_ds)
[{'label': 1,
 'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
 {'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
```

sharddatasets.IterableDataset.shardhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3130[{"name": "num_shards", "val": ": int"}, {"name": "index", "val": ": int"}, {"name": "contiguous", "val": ": bool = True"}]- **num_shards** (`int`) --

How many shards to split the dataset into.

- **index** (`int`) --
  Which shard to select and return.
- **contiguous** -- (`bool` , defaults to `True`):
  Whether to select contiguous blocks of indices for shards.0
  Return the `index` -nth shard from dataset split into `num_shards` pieces.

This shards deterministically. `dataset.shard(n, i)` splits the dataset into contiguous chunks, so it can be easily concatenated back together after processing. If `dataset.num_shards % n == l` , then the first `l` datasets each have `(dataset.num_shards // n) + 1` shards, and the remaining datasets have `(dataset.num_shards // n)` shards. `datasets.concatenate_datasets([dset.shard(n, i) for i in range(n)])` returns a dataset with the same order as the original.
In particular, `dataset.shard(dataset.num_shards, i)` returns a dataset with 1 shard.

Note: n should be less or equal to the number of shards in the dataset `dataset.num_shards` .

On the other hand, `dataset.shard(n, i, contiguous=False)` contains all the shards of the dataset whose index mod `n = i` .

Be sure to shard before using any randomizing operator (such as `shuffle`).
It is best if the shard operator is used early in the dataset pipeline.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("amazon_polarity", split="train", streaming=True)
>>> ds
Dataset({
    features: ['label', 'title', 'content'],
    num_shards: 4
})
>>> ds.shard(num_shards=2, index=0)
Dataset({
    features: ['label', 'title', 'content'],
    num_shards: 2
})
```

repeatdatasets.IterableDataset.repeathttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3050[{"name": "num_times", "val": ": typing.Optional[int]"}]-
num_times (`int`) or (`None`) --
Number of times to repeat the dataset. If `None`, the dataset will be repeated indefinitely.0

Create a new IterableDataset that repeats the underlying dataset `num_times` times.

N.B. The effect of calling shuffle after repeat depends significantly on buffer size.
With buffer_size 1, duplicate data is never seen in the same iteration, even after shuffling:
ds.repeat(n).shuffle(seed=42, buffer_size=1) is equivalent to ds.shuffle(seed=42, buffer_size=1).repeat(n),
and only shuffles shard orders within each iteration.
With buffer size >= (num samples in the dataset * num_times), we get full shuffling of the repeated data, i.e. we can observe duplicates in
the same iteration.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train")
>>> ds = ds.take(2).repeat(2)
>>> list(ds)
[{'label': 1,
 'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
 {'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
 {'label': 1, 'text': 'effective but too-tepid biopic'},
 {'label': 1,
 'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
 {'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
 {'label': 1, 'text': 'effective but too-tepid biopic'}]
```

to_csvdatasets.IterableDataset.to_csvhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3688[{"name": "path_or_buf", "val": ": typing.Union[str, bytes, os.PathLike, typing.BinaryIO]"}, {"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}, {"name": "**to_csv_kwargs", "val": ""}]- **path_or_buf** (`PathLike` or `FileOrBuffer`) --
Either a path to a file (e.g. `file.csv`), a remote URI (e.g. `hf://datasets/username/my_dataset_name/data.csv`),
or a BinaryIO, where the dataset will be saved to in the specified format.

- **batch_size** (`int` , *optional*) --
  Size of the batch to load in memory and write at once.
  Defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` .
- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
- ****to_csv_kwargs** (additional keyword arguments) --
  Parameters to pass to pandas's `pandas.DataFrame.to_csv` .
  The parameter `index` defaults to `False` if not specified.
  If you would like to write the index, pass `index=True` and also set a name for the index column by
  passing `index_label` .0 `int` The number of characters or bytes written.
  Exports the dataset to csv.

This iterates on the dataset and loads it completely in memory before writing it.

Example:

```
>>> ds.to_csv("path/to/dataset/directory")
```

to_pandasdatasets.IterableDataset.to_pandashttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3622[{"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "batched", "val": ": bool = False"}]- **batch_size** (`int` , *optional*) --
The size (number of rows) of the batches if `batched` is `True` .
Defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` .

- **batched** (`bool`) --
  Set to `True` to return a generator that yields the dataset as batches of `batch_size` rows. Defaults to `False` (returns the whole datasets once).0`pandas.DataFrame` or `Iterator[pandas.DataFrame]`
  Returns the dataset as a `pandas.DataFrame` . Can also return a generator for large datasets.

Example:

```
>>> ds.to_pandas()
```

to_dictdatasets.IterableDataset.to_dicthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3584[{"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "batched", "val": ": bool = False"}]- **batch_size** (`int` , *optional*) -- The size (number of rows) of the batches if `batched` is `True` .
Defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` .0`dict` or `Iterator[dict]`
Returns the dataset as a Python dict. Can also return a generator for large datasets.

Example:

```
>>> ds.to_dict()
```

to_jsondatasets.IterableDataset.to_jsonhttps://github.com/huggingface/datasets/blob/4.2.0/src/

datasets/iterable_dataset.py#L3731[{"name": "path_or_buf", "val": ": typing.Union[str, bytes, os.PathLike, typing.BinaryIO]"}, {"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}, {"name": "**to_json_kwargs", "val": ""}]- **path_or_buf** (`PathLike` or `FileOrBuffer`) --
Either a path to a file (e.g. `file.json`), a remote URI (e.g. `hf://datasets/username/my_dataset_name/data.json`),
or a BinaryIO, where the dataset will be saved to in the specified format.

- **batch_size** (`int` , *optional*) --
  Size of the batch to load in memory and write at once.
  Defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` .
- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
- ****to_json_kwargs** (additional keyword arguments) --
  Parameters to pass to pandas's `pandas.DataFrame.to_json` .
  Default arguments are `lines=True` and `orient="records". The parameter` index `defaults to` False `if` orient `is` "split" `or` "table " `. If you would like to write the index, pass` index=Tru e `.</paramsdesc><paramgroups>0</paramgroups><rettype>` int`The number of characters or bytes written.
  Export the dataset to JSON Lines or JSON.

This iterates on the dataset and loads it completely in memory before writing it.

The default output format is JSON Lines.
To export to JSON, pass `lines=False` argument and the desired `orient` .

Example:

```
>>> ds.to_json("path/to/dataset/directory/filename.jsonl")
```

```
>>> num_shards = dataset.num_shards
>>> for index in range(num_shards):
...     shard = dataset.shard(index, num_shards)
...     shard.to_json(f"path/of/my/dataset/data-{index:05d}.jsonl")
```

to_parquetdatasets.IterableDataset.to_parquethttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3827[{"name": "path_or_buf", "val": ": typing.Union[str, bytes, os.PathLike, typing.BinaryIO]"}, {"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "storage_options", "val": ": typing.Optional[dict] = None"}, {"name": "**parquet_writer_kwargs", "val": ""}]- **path_or_buf** (`PathLike` or `FileOrBuffer`) --
Either a path to a file (e.g. `file.parquet`), a remote URI (e.g. `hf://datasets/username/my_dataset_name/data.parquet`),
or a BinaryIO, where the dataset will be saved to in the specified format.

- **batch_size** (`int` , *optional*) --
  Size of the batch to load in memory and write at once.
  Defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` .
- **storage_options** (`dict` , *optional*) --
  Key/value pairs to be passed on to the file-system backend, if any.
- ****parquet_writer_kwargs** (additional keyword arguments) --
  Parameters to pass to PyArrow's `pyarrow.parquet.ParquetWriter` .0 `int` The number of characters or bytes written.
  Exports the dataset to parquet

Example:

```
>>> ds.to_parquet("path/to/dataset/directory")
```

```
>>> num_shards = dataset.num_shards
>>> for index in range(num_shards):
...     shard = dataset.shard(index, num_shards)
...     shard.to_parquet(f"path/of/my/dataset/data-{index:05d}.parquet")
```

to_sqldatasets.IterableDataset.to_sqlhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L3785[{"name": "name", "val": ": str"}, {"name": "con", "val": ": typing.Union[str, ForwardRef('sqlalchemy.engine.Connection'), ForwardRef('sqlalchemy.engine.Engine'), ForwardRef('sqlite3.Connection')]"}, {"name": "batch_size", "val": ": typing.Optional[int] = None"}, {"name": "**sql_writer_kwargs", "val": ""}]- **name** (`str`) --
Name of SQL table.

- **con** (`str` or `sqlite3.Connection` or `sqlalchemy.engine.Connection` or `sqlalchemy.engine.Connection`) --
  A URI string or a SQLite3/SQLAlchemy connection object used to write to a database.
- **batch_size** (`int` , *optional*) --
  Size of the batch to load in memory and write at once.
  Defaults to `datasets.config.DEFAULT_MAX_BATCH_SIZE` .
- ****sql_writer_kwargs** (additional keyword arguments) --
  Parameters to pass to pandas's `pandas.DataFrame.to_sql` .
  The parameter `index` defaults to `False` if not specified.
  If you would like to write the index, pass `index=True` and also set a name for the index column by
  passing `index_label` .0 `int` The number of records written.
  Exports the dataset to a SQL database.

Example:

```
>>> # con provided as a connection URI string
>>> ds.to_sql("data", "sqlite:///my_own_db.sql")
>>> # con provided as a sqlite3 connection object
>>> import sqlite3
>>> con = sqlite3.connect("my_own_db.sql")
>>> with con:
...     ds.to_sql("data", con)
```

push_to_hubdatasets.IterableDataset.push_to_hubhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L4032[{"name": "repo_id", "val": ": str"}, {"name": "config_name", "val": ": str = 'default'"}, {"name": "set_default", "val": ": typing.Optional[bool] = None"}, {"name": "split", "val": ": typing.Optional[str] = None"}, {"name": "data_dir", "val": ": typing.Optional[str] = None"}, {"name": "commit_message", "val": ": typing.Optional[str] = None"}, {"name": "commit_description", "val": ": typing.Optional[str] = None"}, {"name": "private", "val": ": typing.Optional[bool] = None"}, {"name": "token", "val": ": typing.Optional[str] = None"}, {"name": "revision", "val": ": typing.Optional[str] = None"}, {"name": "create_pr", "val": ": typing.Optional[bool] = False"}, {"name": "num_shards", "val": ": typing.Optional[int] = None"}, {"name": "embed_external_files", "val": ": bool = True"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}]- **repo_id** (`str`) --
The ID of the repository to push to in the following format: `<user>/<dataset_name>` or

`<org>/<dataset_name>`. Also accepts `<dataset_name>`, which will default to the namespace of the logged-in user.

- **config_name** (`str`, defaults to "default") --
  The configuration name (or subset) of a dataset. Defaults to "default".
- **set_default** (`bool`, *optional*) --
  Whether to set this configuration as the default one. Otherwise, the default configuration is the one
  named "default".
- **split** (`str`, *optional*) --
  The name of the split that will be given to that dataset. Defaults to `self.split`.
- **data_dir** (`str`, *optional*) --
  Directory name that will contain the uploaded data files. Defaults to the `config_name` if different
  from "default", else "data".
- **commit_message** (`str`, *optional*) --
  Message to commit while pushing. Will default to `"Upload dataset"`.
- **commit_description** (`str`, *optional*) --
  Description of the commit that will be created.
  Additionally, description of the PR if a PR is created (`create_pr` is True).
- **private** (`bool`, *optional*) --
  Whether to make the repo private. If `None` (default), the repo will be public unless the organization's default is private. This value is ignored if the repo already exists.
- **token** (`str`, *optional*) --
  An optional authentication token for the Hugging Face Hub. If no token is passed, will default
  to the token saved locally when logging in with `huggingface-cli login`. Will raise an error if no token is passed and the user is not logged-in.
- **revision** (`str`, *optional*) --
  Branch to push the uploaded files to. Defaults to the `"main"` branch.
- **create_pr** (`bool`, *optional*, defaults to `False`) --
  Whether to create a PR with the uploaded files or directly commit.
- **num_shards** (`int`, *optional*) --
  Number of shards to write. Equals to this dataset's `.num_shards` by default.
- **embed_external_files** (`bool`, defaults to `True`) --

Whether to embed file bytes in the shards.
In particular, this will do the following before the push for the fields of type:
- Audio and Image: remove local path information and embed file content in the Parquet files.

- **num_proc** (`int` , *optional*, defaults to `None`) --
Number of processes when preparing and uploading the dataset.
This is helpful if the dataset is made of many samples and transformations.
Multiprocessing is disabled by default.0huggingface_hub.CommitInfo
Pushes the dataset to the hub as a Parquet dataset.
The dataset is pushed using HTTP requests and does not need to have neither git or git-lfs installed.

The resulting Parquet files are self-contained by default. If your dataset contains Image, Audio or Video
data, the Parquet files will store the bytes of your images or audio files.
You can disable this by setting `embed_external_files` to `False`.

Example:

```
>>> dataset.push_to_hub("<organization>/<dataset_id>")
>>> dataset_dict.push_to_hub("<organization>/<dataset_id>", private=True)
>>> dataset.push_to_hub("<organization>/<dataset_id>", num_shards=1024)
```

If your dataset has multiple splits (e.g. train/validation/test):

```
>>> train_dataset.push_to_hub("<organization>/<dataset_id>", split="train")
>>> val_dataset.push_to_hub("<organization>/<dataset_id>", split="validation")
>>> # later
>>> dataset = load_dataset("<organization>/<dataset_id>")
>>> train_dataset = dataset["train"]
>>> val_dataset = dataset["validation"]
```

If you want to add a new configuration (or subset) to a dataset (e.g. if the dataset has multiple tasks/versions/languages):

```
>>> english_dataset.push_to_hub("<organization>/<dataset_id>", "en")
>>> french_dataset.push_to_hub("<organization>/<dataset_id>", "fr")
>>> # later
>>> english_dataset = load_dataset("<organization>/<dataset_id>", "en")
>>> french_dataset = load_dataset("<organization>/<dataset_id>", "fr")
```

load_state_dictdatasets.IterableDataset.load_state_dicthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L2242[{"name": "state_dict", "val": ": dict"}]
Load the state_dict of the dataset.
The iteration will restart at the next example from when the state was saved.

Resuming returns exactly where the checkpoint was saved except in two cases:

1. examples from shuffle buffers are lost when resuming and the buffers are refilled with new data
2. combinations of `.with_format(arrow)` and batched `.map()` may skip one batch.

Example:

```
>>> from datasets import Dataset, concatenate_datasets
>>> ds = Dataset.from_dict({"a": range(6)}).to_iterable_dataset(num_shards=3)
>>> for idx, example in enumerate(ds):
...     print(example)
...     if idx == 2:
...         state_dict = ds.state_dict()
...         print("checkpoint")
...         break
>>> ds.load_state_dict(state_dict)
>>> print(f"restart from checkpoint")
>>> for example in ds:
...     print(example)
```

which returns:

```
{'a': 0}
{'a': 1}
{'a': 2}
checkpoint
restart from checkpoint
{'a': 3}
{'a': 4}
{'a': 5}
```

```
>>> from torchdata.stateful_dataloader import StatefulDataLoader
>>> ds = load_dataset("deepmind/code_contests", streaming=True, split="train")
>>> dataloader = StatefulDataLoader(ds, batch_size=32, num_workers=4)
>>> # checkpoint
>>> state_dict = dataloader.state_dict()  # uses ds.state_dict() under the hood
>>> # resume from checkpoint
>>> dataloader.load_state_dict(state_dict)  # uses ds.load_state_dict() under the hood
```

state_dictdatasets.IterableDataset.state_dicthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L2189[] `dict`

Get the current state_dict of the dataset.

It corresponds to the state at the latest example it yielded.

Resuming returns exactly where the checkpoint was saved except in two cases:

1. examples from shuffle buffers are lost when resuming and the buffers are refilled with new data
2. combinations of `.with_format(arrow)` and batched `.map()` may skip one batch.

Example:

```
>>> from datasets import Dataset, concatenate_datasets
>>> ds = Dataset.from_dict({"a": range(6)}).to_iterable_dataset(num_shards=3)
>>> for idx, example in enumerate(ds):
...     print(example)
...     if idx == 2:
...         state_dict = ds.state_dict()
...         print("checkpoint")
...         break
>>> ds.load_state_dict(state_dict)
>>> print(f"restart from checkpoint")
>>> for example in ds:
...     print(example)
```

which returns:

```
{'a': 0}
{'a': 1}
{'a': 2}
checkpoint
restart from checkpoint
{'a': 3}
{'a': 4}
{'a': 5}
```

```
>>> from torchdata.stateful_dataloader import StatefulDataLoader
>>> ds = load_dataset("deepmind/code_contests", streaming=True, split="train")
>>> dataloader = StatefulDataLoader(ds, batch_size=32, num_workers=4)
>>> # checkpoint
>>> state_dict = dataloader.state_dict()  # uses ds.state_dict() under the hood
>>> # resume from checkpoint
>>> dataloader.load_state_dict(state_dict)  # uses ds.load_state_dict() under the hood
```

infodatasets.IterableDataset.infohttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L167[]
DatasetInfo object containing all the metadata in the dataset.

splitdatasets.IterableDataset.splithttps://github.com/huggingface/datasets/blob/4.2.0/src/

datasets/arrow_dataset.py#L172[]
NamedSplit object corresponding to a named dataset split.

builder_namedatasets.IterableDataset.builder_namehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L177[]

citationdatasets.IterableDataset.citationhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L181[]

config_namedatasets.IterableDataset.config_namehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L185[]

dataset_sizedatasets.IterableDataset.dataset_sizehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L189[]

descriptiondatasets.IterableDataset.descriptionhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L193[]

download_checksumsdatasets.IterableDataset.download_checksumshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L197[]

download_sizedatasets.IterableDataset.download_sizehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L201[]

featuresdatasets.IterableDataset.featureshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L205[]

homepagedatasets.IterableDataset.homepagehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L209[]

licensedatasets.IterableDataset.licensehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L213[]

size_in_bytesdatasets.IterableDataset.size_in_byteshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L217[]

supervised_keysdatasets.IterableDataset.supervised_keyshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/arrow_dataset.py#L221[]

versiondatasets.IterableDataset.versionhttps://github.com/huggingface/datasets/blob/4.2.0/src/

datasets/arrow_dataset.py#L225[]

class datasets.IterableColumndatasets.IterableColumnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/iterable_dataset.py#L2092[{"name": "source", "val": ": typing.Union[ForwardRef('IterableDataset'), ForwardRef('IterableColumn')]"}, {"name": "column_name", "val": ": str"}]

An iterable for a specific column of an IterableDataset.

Example:

Iterate on the texts of the "text" column of a dataset:

```
for text in dataset["text"]:
    ...
```

It also works with nested columns:

```
for source in dataset["metadata"]["source"]:
    ...
```

IterableDatasetDictdatasets.IterableDatasetDict

Dictionary with split names as keys ('train', 'test' for example), and `IterableDataset` objects as values.

class datasets.IterableDatasetDictdatasets.IterableDatasetDicthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L1986""

mapdatasets.IterableDatasetDict.maphttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L2087[{"name": "function", "val": ": typing.Optional[typing.Callable] = None"}, {"name": "with_indices", "val": ": bool = False"}, {"name": "with_split", "val": ": bool = False"}, {"name": "input_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "batched", "val": ": bool = False"}, {"name": "batch_size", "val": ": int = 1000"}, {"name": "drop_last_batch", "val": ": bool = False"}, {"name": "remove_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "fn_kwargs", "val": ": typing.Optional[dict] = None"}]- **function** (`Callable` , *optional*, defaults to `None`) --

Function applied on-the-fly on the examples when you iterate on the dataset.
It must have one of the following signatures:

- `function(example: Dict[str, Any]) -> Dict[str, Any]` if `batched=False` and `with_indices=False`
- `function(example: Dict[str, Any], idx: int) -> Dict[str, Any]` if `batched=False` and `with_indices=True`
- `function(batch: Dict[str, list]) -> Dict[str, list]` if `batched=True` and `with_indices=False`
- `function(batch: Dict[str, list], indices: list[int]) -> Dict[str, list]` if `batched=True` and `with_indices=True`

For advanced usage, the function can also return a `pyarrow.Table`.
If the function is asynchronous, then `map` will run your function in parallel.
Moreover if your function returns nothing (`None`), then `map` will run your function and return the dataset unchanged.
If no function is provided, default to identity function: `lambda x: x`.

- **with_indices** (`bool`, defaults to `False`) --
  Provide example indices to `function`. Note that in this case the signature of `function` should be `def function(example, idx[, rank]): ...`.
- **input_columns** (`[Union[str, list[str]]]`, *optional*, defaults to `None`) --
  The columns to be passed into `function`
  as positional arguments. If `None`, a dict mapping to all formatted columns is passed as one argument.
- **batched** (`bool`, defaults to `False`) --
  Provide batch of examples to `function`.
- **batch_size** (`int`, *optional*, defaults to `1000`) --
  Number of examples per batch provided to `function` if `batched=True`.
- **drop_last_batch** (`bool`, defaults to `False`) --
  Whether a last batch smaller than the `batch_size` should be
  dropped instead of being processed by the function.
- **remove_columns** (`[list[str]]`, *optional*, defaults to `None`) --
  Remove a selection of columns while doing the mapping.
  Columns will be removed before updating the examples with the output of `function`, i.e. if `function` is adding

columns with names in `remove_columns`, these columns will be kept.

- **fn_kwargs** (`Dict`, *optional*, defaults to `None`) --
  Keyword arguments to be passed to `function` 0

Apply a function to all the examples in the iterable dataset (individually or in batches) and update them.
If your function returns a column that already exists, then it overwrites it.
The function is applied on-the-fly on the examples when iterating over the dataset.
The transformation is applied to all the datasets of the dataset dictionary.

You can specify whether the function should be batched or not with the `batched` parameter:

- If batched is `False`, then the function takes 1 example in and should return 1 example.
  An example is a dictionary, e.g. `{"text": "Hello there !"}`.
- If batched is `True` and `batch_size` is 1, then the function takes a batch of 1 example as input and can return a batch with 1 or more examples.
  A batch is a dictionary, e.g. a batch of 1 example is `{"text": ["Hello there !"]}`.
- If batched is `True` and `batch_size` is `n` > 1, then the function takes a batch of `n` examples as input and can return a batch with `n` examples, or with an arbitrary number of examples.
  Note that the last batch may have less than `n` examples.
  A batch is a dictionary, e.g. a batch of `n` examples is `{"text": ["Hello there !"] * n}`.

If the function is asynchronous, then `map` will run your function in parallel, with up to one thousand simultaneous calls.
It is recommended to use a `asyncio.Semaphore` in your function if you want to set a maximum number of operations that can run at the same time.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", streaming=True)
>>> def add_prefix(example):
...     example["text"] = "Review: " + example["text"]
...     return example
>>> ds = ds.map(add_prefix)
>>> next(iter(ds["train"]))
{'label': 1,
 'text': 'Review: the rock is destined to be the 21st century's new " conan " and that he's going
```

filterdatasets.IterableDatasetDict.filterhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L2187[{"name": "function", "val": ": typing.Optional[typing.Callable] = None"}, {"name": "with_indices", "val": " = False"}, {"name": "input_columns", "val": ": typing.Union[str, list[str], NoneType] = None"}, {"name": "batched", "val": ": bool = False"}, {"name": "batch_size", "val": ": typing.Optional[int] = 1000"}, {"name": "fn_kwargs", "val": ": typing.Optional[dict] = None"}]- **function** (`Callable`) --
Callable with one of the following signatures:

- `function(example: Dict[str, Any]) -> bool` if `with_indices=False, batched=False`
- `function(example: Dict[str, Any], indices: int) -> bool` if `with_indices=True, batched=False`
- `function(example: Dict[str, list]) -> list[bool]` if `with_indices=False, batched=True`
- `function(example: Dict[str, list], indices: list[int]) -> list[bool]` if `with_indices=True, batched=True`

If no function is provided, defaults to an always True function: `lambda x: True`.

- **with_indices** (`bool` , defaults to `False`) --
  Provide example indices to `function` . Note that in this case the signature of `function` should be `def function(example, idx): ...` .
- **input_columns** (`str` or `list[str]` , *optional*) --
  The columns to be passed into `function` as
  positional arguments. If `None` , a dict mapping to all formatted columns is passed as one argument.
- **batched** (`bool` , defaults to `False`) --
  Provide batch of examples to `function`
- **batch_size** (`int` , *optional*, defaults to `1000`) --

Number of examples per batch provided to `function` if `batched=True`.

- **fn_kwargs** (`Dict` , *optional*, defaults to `None`) --
  Keyword arguments to be passed to `function` 0
  Apply a filter function to all the elements so that the dataset only includes examples according to the filter function.
  The filtering is done on-the-fly when iterating over the dataset.
  The filtering is applied to all the datasets of the dataset dictionary.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", streaming=True)
>>> ds = ds.filter(lambda x: x["label"] == 0)
>>> list(ds["train"].take(3))
[{'label': 0, 'text': 'Review: simplistic , silly and tedious .'},
 {'label': 0,
 'text': "Review: it's so laddish and juvenile , only teenage boys could possibly find it funny ."
 {'label': 0,
 'text': 'Review: exploitative and largely devoid of the depth or sophistication that would make w
```

shuffledatasets.IterableDatasetDict.shufflehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L2250[{"name": "seed", "val": " = None"}, {"name": "generator", "val": ": typing.Optional[numpy.random._generator.Generator] = None"}, {"name": "buffer_size", "val": ": int = 1000"}]- **seed** (`int` , *optional*, defaults to `None`) --
Random seed that will be used to shuffle the dataset.
It is used to sample from the shuffle buffer and also to shuffle the data shards.

- **generator** (`numpy.random.Generator` , *optional*) --
  Numpy random Generator to use to compute the permutation of the dataset rows.
  If `generator=None` (default), uses `np.random.default_rng` (the default BitGenerator (PCG64) of NumPy).
- **buffer_size** (`int` , defaults to `1000`) --
  Size of the buffer.0

Randomly shuffles the elements of this dataset.
The shuffling is applied to all the datasets of the dataset dictionary.

This dataset fills a buffer with buffer_size elements, then randomly samples elements from this buffer,
replacing the selected elements with new elements. For perfect shuffling, a buffer size greater than or
equal to the full size of the dataset is required.

For instance, if your dataset contains 10,000 elements but `buffer_size` is set to 1000, then `shuffle` will
initially select a random element from only the first 1000 elements in the buffer. Once an element is
selected, its space in the buffer is replaced by the next (i.e. 1,001-st) element,
maintaining the 1000 element buffer.

If the dataset is made of several shards, it also does `shuffle` the order of the shards.
However if the order has been fixed by using skip() or take()
then the order of the shards is kept unchanged.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", streaming=True)
>>> list(ds["train"].take(3))
[{'label': 1,
 'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
 {'label': 1,
 'text': 'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge th
 {'label': 1, 'text': 'effective but too-tepid biopic'}]
>>> ds = ds.shuffle(seed=42)
>>> list(ds["train"].take(3))
[{'label': 1,
 'text': "a sports movie with action that's exciting on the field and a story you care about off i
 {'label': 1,
 'text': 'at its best , the good girl is a refreshingly adult take on adultery . . .'},
 {'label': 1,
 'text': "sam jones became a very lucky filmmaker the day wilco got dropped from their record labe
```

with_formatdatasets.IterableDatasetDict.with_formathttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L2041[{"name": "type", "val": ": typing.Optional[str] =

None"}]- **type** (`str` , *optional*) --
Either output type selected in
`[None, 'numpy', 'torch', 'tensorflow', 'jax', 'arrow', 'pandas', 'polars']`.
`None` means it returns python objects (default).0

Return a dataset with the specified format.

Example:

```
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="validation", streaming=T
>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
>>> ds = ds.map(lambda x: tokenizer(x['text'], truncation=True, padding=True), batched=True)
>>> ds = ds.with_format("torch")
>>> next(iter(ds))
{'text': 'compassionately explores the seemingly irreconcilable situation between conservative chr
 'label': tensor(1),
 'input_ids': tensor([  101, 18027, 16310, 16001,  1103,  9321,   178, 11604,  7235,  6617,
        1742,  2165,  2820,  1206,  6588, 22572, 12937,  1811,  2153,  1105,
        1147, 12890, 19587,  6463,  1105, 15026,  1482,   119,   102,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0]),
 'token_type_ids': tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])}
```

castdatasets.IterableDatasetDict.casthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L2458[{"name": "features", "val": ": Features"}]- **features** (`Features`)
--

New features to cast the dataset to.
The name of the fields in the features must match the current column names.
The type of the data must also be convertible from one type to the other.
For non-trivial conversion, e.g. `string` <-> `ClassLabel` you should use `map` to update the Dataset.0IterableDatasetDictA copy of the dataset with casted features.

Cast the dataset to a new set of features.
The type casting is applied to all the datasets of the dataset dictionary.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", streaming=True)
>>> ds["train"].features
{'label': ClassLabel(names=['neg', 'pos']),
 'text': Value('string')}
>>> new_features = ds["train"].features.copy()
>>> new_features['label'] = ClassLabel(names=['bad', 'good'])
>>> new_features['text'] = Value('large_string')
>>> ds = ds.cast(new_features)
>>> ds["train"].features
{'label': ClassLabel(names=['bad', 'good']),
 'text': Value('large_string')}
```

cast_columndatasets.IterableDatasetDict.cast_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L2427[{"name": "column", "val": ": str"}, {"name": "feature", "val": ": typing.Union[dict, list, tuple, datasets.features.features.Value, datasets.features.features.ClassLabel, datasets.features.translation.Translation, datasets.features.translation.TranslationVariableLanguages, datasets.features.features.LargeList, datasets.features.features.List, datasets.features.features.Array2D, datasets.features.features.Array3D, datasets.features.features.Array4D, datasets.features.features.Array5D, datasets.features.audio.Audio, datasets.features.image.Image, datasets.features.video.Video, datasets.features.pdf.Pdf]"}]- **column** (`str`) --
Column name.

- **feature** (`Feature`) --

Target feature.0IterableDatasetDict

Cast column to feature for decoding.

The type casting is applied to all the datasets of the dataset dictionary.

Example:

```
>>> from datasets import load_dataset, ClassLabel
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", streaming=True)
>>> ds["train"].features
{'label': ClassLabel(names=['neg', 'pos']),
 'text': Value('string')}
>>> ds = ds.cast_column('label', ClassLabel(names=['bad', 'good']))
>>> ds["train"].features
{'label': ClassLabel(names=['bad', 'good']),
 'text': Value('string')}
```

remove_columnsdatasets.IterableDatasetDict.remove_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L2375[{"name": "column_names", "val": ": typing.Union[str, list[str]]"}]- **column_names** (`Union[str, list[str]]`) --
Name of the column(s) to remove.0IterableDatasetDictA copy of the dataset object without the columns to remove.

Remove one or several column(s) in the dataset and the features associated to them.
The removal is done on-the-fly on the examples when iterating over the dataset.
The removal is applied to all the datasets of the dataset dictionary.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", streaming=True)
>>> ds = ds.remove_columns("label")
>>> next(iter(ds["train"]))
{'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
```

rename_columndatasets.IterableDatasetDict.rename_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L2311[{"name": "original_column_name", "val": ": str"}, {"name": "new_column_name", "val": ": str"}]- **original_column_name** (`str`) --

Name of the column to rename.

- **new_column_name** (`str`) --
  New name for the column.0IterableDatasetDictA copy of the dataset with a renamed column.

Rename a column in the dataset, and move the features associated to the original column under the new column
name.
The renaming is applied to all the datasets of the dataset dictionary.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", streaming=True)
>>> ds = ds.rename_column("text", "movie_review")
>>> next(iter(ds["train"]))
{'label': 1,
 'movie_review': 'the rock is destined to be the 21st century's new " conan " and that he's going
```

rename_columnsdatasets.IterableDatasetDict.rename_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L2347[{"name": "column_mapping", "val": ": dict"}]- **column_mapping** (`Dict[str, str]`) --
A mapping of columns to rename to their new names.0IterableDatasetDictA copy of the dataset with renamed columns

Rename several columns in the dataset, and move the features associated to the original columns under
the new column names.
The renaming is applied to all the datasets of the dataset dictionary.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", streaming=True)
>>> ds = ds.rename_columns({"text": "movie_review", "label": "rating"})
>>> next(iter(ds["train"]))
{'movie_review': 'the rock is destined to be the 21st century's new " conan " and that he's going
 'rating': 1}
```

select_columnsdatasets.IterableDatasetDict.select_columnshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L2401[{"name": "column_names", "val": ": typing.Union[str, list[str]]"}]- **column_names** (`Union[str, list[str]]`) -- Name of the column(s) to keep.0IterableDatasetDictA copy of the dataset object with only selected columns.

Select one or several column(s) in the dataset and the features associated to them. The selection is done on-the-fly on the examples when iterating over the dataset. The selection is applied to all the datasets of the dataset dictionary.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", streaming=True)
>>> ds = ds.select("text")
>>> next(iter(ds["train"]))
{'text': 'the rock is destined to be the 21st century's new " conan " and that he's going to make
```

push_to_hubdatasets.IterableDatasetDict.push_to_hubhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/dataset_dict.py#L2495[{"name": "repo_id", "val": ""}, {"name": "config_name", "val": ": str = 'default'"}, {"name": "set_default", "val": ": typing.Optional[bool] = None"}, {"name": "data_dir", "val": ": typing.Optional[str] = None"}, {"name": "commit_message", "val": ": typing.Optional[str] = None"}, {"name": "commit_description", "val": ": typing.Optional[str] = None"}, {"name": "private", "val": ": typing.Optional[bool] = None"}, {"name": "token", "val": ": typing.Optional[str] = None"}, {"name": "revision", "val": ": typing.Optional[str] = None"}, {"name": "create_pr", "val": ": typing.Optional[bool] = False"}, {"name": "num_shards", "val": ": typing.Optional[dict[str, int]] = None"}, {"name": "embed_external_files", "val": ": bool = True"}, {"name": "num_proc", "val": ": typing.Optional[int] = None"}]- **repo_id** (`str`) --

The ID of the repository to push to in the following format: `<user>/<dataset_name>` or `<org>/<dataset_name>`. Also accepts `<dataset_name>`, which will default to the namespace of the logged-in user.

- **config_name** (`str`) --
  Configuration name of a dataset. Defaults to "default".
- **set_default** (`bool` , *optional*) --
  Whether to set this configuration as the default one. Otherwise, the default configuration is the one
  named "default".
- **data_dir** (`str` , *optional*) --
  Directory name that will contain the uploaded data files. Defaults to the `config_name` if different
  from "default", else "data".
- **commit_message** (`str` , *optional*) --
  Message to commit while pushing. Will default to `"Upload dataset"` .
- **commit_description** (`str` , *optional*) --
  Description of the commit that will be created.
  Additionally, description of the PR if a PR is created (`create_pr` is True).
- **private** (`bool` , *optional*) --
  Whether to make the repo private. If `None` (default), the repo will be public unless the organization's default is private. This value is ignored if the repo already exists.
- **token** (`str` , *optional*) --
  An optional authentication token for the Hugging Face Hub. If no token is passed, will default
  to the token saved locally when logging in with `huggingface-cli login` . Will raise an error if no token is passed and the user is not logged-in.
- **revision** (`str` , *optional*) --
  Branch to push the uploaded files to. Defaults to the `"main"` branch.
- **create_pr** (`bool` , *optional*, defaults to `False`) --
  Whether to create a PR with the uploaded files or directly commit.
- **num_shards** (`Dict[str, int]` , *optional*) --
  Number of shards to write. Equals to this dataset's `.num_shards` by default.
  Use a dictionary to define a different num_shards for each split.
- **embed_external_files** (`bool` , defaults to `True`) --

Whether to embed file bytes in the shards.
In particular, this will do the following before the push for the fields of type:
- Audio and Image removes local path information and embed file content in the Parquet files.

- **num_proc** (`int` , *optional*, defaults to `None`) --
Number of processes when preparing and uploading the dataset.
This is helpful if the dataset is made of many samples or media files to embed.
Multiprocessing is disabled by default.
0huggingface_hub.CommitInfo
Pushes the DatasetDict to the hub as a Parquet dataset.
The DatasetDict is pushed using HTTP requests and does not need to have neither git or git-lfs installed.

Each dataset split will be pushed independently. The pushed dataset will keep the original split names.

The resulting Parquet files are self-contained by default: if your dataset contains Image or Audio
data, the Parquet files will store the bytes of your images or audio files.
You can disable this by setting `embed_external_files` to False.

Example:

```
>>> dataset_dict.push_to_hub("<organization>/<dataset_id>")
>>> dataset_dict.push_to_hub("<organization>/<dataset_id>", private=True)
>>> dataset_dict.push_to_hub("<organization>/<dataset_id>", num_shards={"train": 1024, "test": 8})
```

If you want to add a new configuration (or subset) to a dataset (e.g. if the dataset has multiple tasks/versions/languages):

```
>>> english_dataset.push_to_hub("<organization>/<dataset_id>", "en")
>>> french_dataset.push_to_hub("<organization>/<dataset_id>", "fr")
>>> # later
>>> english_dataset = load_dataset("<organization>/<dataset_id>", "en")
>>> french_dataset = load_dataset("<organization>/<dataset_id>", "fr")
```

Featuresdatasets.Features

class datasets.Featuresdatasets.Featureshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L1734[{"name": "*args", "val": ""}, {"name": "**kwargs", "val": ""}]
A special dictionary that defines the internal structure of a dataset.

Instantiated with a dictionary of type `dict[str, FieldType]`, where keys are the desired column names,
and values are the type of that column.

`FieldType` can be one of the following:

- Value feature specifies a single data type value, e.g. `int64` or `string`.
- ClassLabel feature specifies a predefined set of classes which can have labels associated to them and
  will be stored as integers in the dataset.
- Python `dict` specifies a composite feature containing a mapping of sub-fields to sub-features.
  It's possible to have nested fields of nested fields in an arbitrary manner.
- List or LargeList specifies a composite feature containing a sequence of
  sub-features, all of the same feature type.
- Array2D, Array3D, Array4D or Array5D feature for multidimensional arrays.
- Audio feature to store the absolute path to an audio file or a dictionary with the relative path
  to an audio file ("path" key) and its bytes content ("bytes" key).
  This feature loads the audio lazily with a decoder.
- Image feature to store the absolute path to an image file, an `np.ndarray` object, a `PIL.Image.Image` object
  or a dictionary with the relative path to an image file ("path" key) and its bytes content ("bytes" key).
  This feature extracts the image data.
- Video feature to store the absolute path to a video file, a `torchcodec.decoders.VideoDecoder` object
  or a dictionary with the relative path to a video file ("path" key) and its bytes content ("bytes" key).

This feature loads the video lazily with a decoder.

- Pdf feature to store the absolute path to a PDF file, a `pdfplumber.pdf.PDF` object or a dictionary with the relative path to a PDF file ("path" key) and its bytes content ("bytes" key).
  This feature loads the PDF lazily with a PDF reader.
- Translation or TranslationVariableLanguages feature specific to Machine Translation.

copydatasets.Features.copyhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L2157[]Features

Make a deep copy of Features.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train")
>>> copy_of_features = ds.features.copy()
>>> copy_of_features
{'label': ClassLabel(names=['neg', 'pos']),
 'text': Value('string')}
```

decode_batchdatasets.Features.decode_batchhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L2130[{"name": "batch", "val": ": dict"}, {"name": "token_per_repo_id", "val": ": typing.Optional[dict[str, typing.Union[str, bool, NoneType]]] = None"}]- **batch** (`dict[str, list[Any]]`) --
Dataset batch data.

- **token_per_repo_id** (`dict` , *optional*) --
  To access and decode audio or image files from private repositories on the Hub, you can pass
  a dictionary repo_id (str) -> token (bool or str)0 `dict[str, list[Any]]`
  Decode batch with custom feature decoding.

decode_columndatasets.Features.decode_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L2105[{"name": "column", "val": ": list"}, {"name": "column_name", "val": ": str"}, {"name": "token_per_repo_id", "val": ": typing.Optional[dict[str, typing.Union[str, bool, NoneType]]] = None"}]- **column** (`list[Any]`) --

Dataset column data.

- **column_name** (`str`) --
  Dataset column name.0 `list[Any]`
  Decode column with custom feature decoding.

decode_exampledatasets.Features.decode_examplehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L2082[{"name": "example", "val": ": dict"}, {"name": "token_per_repo_id", "val": ": typing.Optional[dict[str, typing.Union[str, bool, NoneType]]] = None"}]- **example** (`dict[str, Any]`) --
Dataset row data.

- **token_per_repo_id** (`dict` , *optional*) --
  To access and decode audio or image files from private repositories on the Hub, you can pass
  a dictionary `repo_id (str) -> token (bool or str)` .0 `dict[str, Any]`
  Decode example with custom feature decoding.

encode_batchdatasets.Features.encode_batchhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L2063[{"name": "batch", "val": ""}]- **batch** (`dict[str, list[Any]]`) --
Data in a Dataset batch.0 `dict[str, list[Any]]`

Encode batch into a format for Arrow.

encode_columndatasets.Features.encode_columnhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L2047[{"name": "column", "val": ""}, {"name": "column_name", "val": ": str"}]- **column** (`list[Any]`) --
Data in a Dataset column.

- **column_name** (`str`) --
  Dataset column name.0 `list[Any]`

Encode column into a format for Arrow.

encode_exampledatasets.Features.encode_examplehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L2033[{"name": "example", "val": ""}]- **example** (`dict[str, Any]`) --

Data in a Dataset row.0 `dict[str, Any]`

Encode example into a format for Arrow.

flattendatasets.Features.flattenhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L2230[{"name": "max_depth", "val": " = 16"}]FeaturesThe flattened features.
Flatten the features. Every dictionary column is removed and is replaced by
all the subfields it contains. The new fields are named by concatenating the
name of the original column and the subfield name like this: `<original>.<subfield>`.

If a column contains nested dictionaries, then all the lower-level subfields names are
also concatenated to form new columns: `<original>.<subfield>.<subsubfield>`, etc.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("rajpurkar/squad", split="train")
>>> ds.features.flatten()
{'answers.answer_start': List(Value('int32'), id=None),
 'answers.text': List(Value('string'), id=None),
 'context': Value('string'),
 'id': Value('string'),
 'question': Value('string'),
 'title': Value('string')}
```

from_arrow_schemadatasets.Features.from_arrow_schemahttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L1816[{"name": "pa_schema", "val": ": Schema"}]- **pa_schema** (`pyarrow.Schema`) --
Arrow Schema.0Features

Construct Features from Arrow Schema.
It also checks the schema metadata for Hugging Face Datasets features.
Non-nullable fields are not supported and set to nullable.

Also, pa.dictionary is not supported and it uses its underlying type instead.
Therefore datasets convert DictionaryArray objects to their actual values.

from_dictdatasets.Features.from_dicthttps://github.com/huggingface/datasets/blob/4.2.0/src/

datasets/features/features.py#L1850[{"name": "dic", "val": ""}]- **dic** (*dict[str, Any]*) --
Python dictionary.0*Features*

Construct [*Features*] from dict.

Regenerate the nested feature object from a deserialized dict.
We use the *_type* key to infer the dataclass name of the feature *FieldType*.

It allows for a convenient constructor syntax
to define features from deserialized JSON dictionaries. This function is used in particular when deserializing
a [*DatasetInfo*] that was dumped to a JSON object. This acts as an analogue to
[*Features.from_arrow_schema*] and handles the recursive field-by-field instantiation, but doesn't require
any mapping to/from pyarrow, except for the fact that it takes advantage of the mapping of pyarrow primitive
dtypes that [*Value*] automatically performs.

Example:

```
>>> Features.from_dict(&amp;lcub;'_type': &amp;lcub;'dtype': 'string', 'id': None, '_type': 'Value
&amp;lcub;'_type': Value('string')}
```

reorder_fields_asdatasets.Features.reorder_fields_ashttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L2177[{"name": "other", "val": ": Features"}]- **other** ([*Features*]) --
The other [*Features*] to align with.0[*Features*]

Reorder Features fields to match the field order of other [*Features*].

The order of the fields is important since it matters for the underlying arrow data.
Re-ordering the fields allows to make the underlying arrow data type match.

Example:

```
>>> from datasets import Features, List, Value
>>> # let's say we have two features with a different order of nested fields (for a and b for exam
>>> f1 = Features(&amp;lcub;"root": &amp;lcub;"a": Value("string"), "b": Value("string")}})
>>> f2 = Features(&amp;lcub;"root": &amp;lcub;"b": Value("string"), "a": Value("string")}})
>>> assert f1.type != f2.type
>>> # re-ordering keeps the base structure (here List is defined at the root level), but makes the
>>> f1.reorder_fields_as(f2)
&amp;lcub;'root': List(&amp;lcub;'b': Value('string'), 'a': Value('string')})}
>>> assert f1.reorder_fields_as(f2).type == f2.type
```

Scalardatasets.Value

class datasets.Valuedatasets.Valuehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L482[{"name": "dtype", "val": ": str"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **dtype** (`str`) --
Name of the data type.0

Scalar feature value of a particular data type.

The possible dtypes of `Value` are as follows:

- `null`
- `bool`
- `int8`
- `int16`
- `int32`
- `int64`
- `uint8`
- `uint16`
- `uint32`
- `uint64`
- `float16`
- `float32` (alias float)
- `float64` (alias double)
- `time32[(s|ms)]`
- `time64[(us|ns)]`

- `timestamp[(s|ms|us|ns)]`
- `timestamp[(s|ms|us|ns), tz=(tzstring)]`
- `date32`
- `date64`
- `duration[(s|ms|us|ns)]`
- `decimal128(precision, scale)`
- `decimal256(precision, scale)`
- `binary`
- `large_binary`
- `string`
- `large_string`
- `string_view`

Example:

```
>>> from datasets import Features
>>> features = Features({'stars': Value('int32')})
>>> features
{'stars': Value('int32')}
```

class datasets.ClassLabeldatasets.ClassLabelhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L973[{"name": "num_classes", "val": ": dataclasses.InitVar[typing.Optional[int]] = None"}, {"name": "names", "val": ": list = None"}, {"name": "names_file", "val": ": dataclasses.InitVar[typing.Optional[str]] = None"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **num_classes** (`int` , *optional*) --
Number of classes. All labels must be < `num_classes` .

- **names** (`list` of `str` , *optional*) --
  String names for the integer classes.
  The order in which the names are provided is kept.
- **names_file** (`str` , *optional*) --
  Path to a file with names for the integer classes, one per line.0
  Feature type for integer class labels.

There are 3 ways to define a `ClassLabel` , which correspond to the 3 arguments:

- `num_classes` : Create 0 to (num_classes-1) labels.

- `names`: List of label strings.
- `names_file`: File containing the list of labels.

Under the hood the labels are stored as integers.
You can use negative integers to represent unknown/missing labels.

Example:

```
>>> from datasets import Features, ClassLabel
>>> features = Features({'label': ClassLabel(num_classes=3, names=['bad', 'ok', 'good'])})
>>> features
{'label': ClassLabel(names=['bad', 'ok', 'good'])}
```

cast_storagedatasets.ClassLabel.cast_storagehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L1138[{"name": "storage", "val": ": typing.Union[pyarrow.lib.StringArray, pyarrow.lib.IntegerArray]"}]- **storage** (`Union[pa.StringArray, pa.IntegerArray]`) --
PyArrow array to cast.0 `pa.Int64Array` Array in the `ClassLabel` arrow storage type.
Cast an Arrow array to the `ClassLabel` arrow storage type.
The Arrow types that can be converted to the `ClassLabel` pyarrow storage type are:

- `pa.string()`
- `pa.int()`

int2strdatasets.ClassLabel.int2strhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L1092[{"name": "values", "val": ": typing.Union[int, collections.abc.Iterable]"}]
Conversion `integer` => class name `string`.

Regarding unknown/missing labels: passing negative integers raises `ValueError`.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train")
>>> ds.features["label"].int2str(0)
'neg'
```

str2intdatasets.ClassLabel.str2inthttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L1047[{"name": "values", "val": ": typing.Union[str, collections.abc.Iterable]"}]
Conversion class name `string` => `integer`.

Example:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train")
>>> ds.features["label"].str2int('neg')
0
```

Compositedatasets.LargeList

class datasets.LargeListdatasets.LargeListhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L1232[{"name": "feature", "val": ": typing.Any"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **feature** (`FeatureType`) --
Child feature data type of each item within the large list.0
Feature type for large list data composed of child feature data type.

It is backed by `pyarrow.LargeListType`, which is like `pyarrow.ListType` but with 64-bit rather than 32-bit offsets.

class datasets.Listdatasets.Listhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L1204[{"name": "feature", "val": ": typing.Any"}, {"name": "length", "val": ": int = -1"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **feature** (`FeatureType`) --
Child feature data type of each item within the large list.

- **length** (optional `int`, default to -1) --
  Length of the list if it is fixed.
  Defaults to -1 which means an arbitrary length.0
  Feature type for large list data composed of child feature data type.

It is backed by `pyarrow.ListType`, which uses 32-bit offsets or a fixed length.

class datasets.Sequencedatasets.Sequencehttps://github.com/huggingface/datasets/blob/

4.2.0/src/datasets/features/features.py#L1170[{"name": "feature", "val": " = None"}, {"name": "length", "val": " = -1"}, {"name": "**kwargs", "val": ""}]- **feature** (`FeatureType`) --
Child feature data type of each item within the large list.

- **length** (optional `int` , default to -1) --
  Length of the list if it is fixed.
  Defaults to -1 which means an arbitrary length.0List of the specified feature, except `dict`
  of sub-features
  which are converted to `dict` of lists of sub-features for compatibility with TFDS.

A `Sequence` is a utility that automatically converts internal dictionary feature into a dictionary of lists. This behavior is implemented to have a compatibility layer with the TensorFlow Datasets library but may be
un-wanted in some cases. If you don't want this behavior, you can use a List or a LargeList instead of the Sequence.

Translationdatasets.Translation

class datasets.Translationdatasets.Translationhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/translation.py#L12[{"name": "languages", "val": ": list"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **languages** (`dict`) --
A dictionary for each example mapping string language codes to string translations.0
`Feature` for translations with fixed languages per example.
Here for compatibility with tfds.

Example:

```
>>> # At construction time:
>>> datasets.features.Translation(languages=['en', 'fr', 'de'])
>>> # During data generation:
>>> yield {
...         'en': 'the cat',
...         'fr': 'le chat',
...         'de': 'die katze'
... }
```

flattendatasets.Translation.flattenhttps://github.com/huggingface/datasets/blob/4.2.0/src/

datasets/features/translation.py#L44[]
Flatten the Translation feature into a dictionary.

class datasets.TranslationVariableLanguagesdatasets.TranslationVariableLanguageshttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/translation.py#L52[{"name": "languages", "val": ": typing.Optional[list] = None"}, {"name": "num_languages", "val": ": typing.Optional[int] = None"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **languages** (`dict`) --
A dictionary for each example mapping string language codes to one or more string translations.
The languages present may vary from example to example.0- `language` or `translation` (variable-length 1D `tf.Tensor` of `tf.string`)Language codes sorted in ascending order or plain text translations, sorted to align with language codes.
`Feature` for translations with variable languages per example.
Here for compatibility with tfds.

Example:

```
>>> # At construction time:
>>> datasets.features.TranslationVariableLanguages(languages=['en', 'fr', 'de'])
>>> # During data generation:
>>> yield {
...         'en': 'the cat',
...         'fr': ['le chat', 'la chatte,']
...         'de': 'die katze'
... }
>>> # Tensor returned :
>>> {
...         'language': ['en', 'de', 'fr', 'fr'],
...         'translation': ['the cat', 'die katze', 'la chatte', 'le chat'],
... }
```

flattendatasets.TranslationVariableLanguages.flattenhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/translation.py#L122[]
Flatten the TranslationVariableLanguages feature into a dictionary.

Arraysdatasets.Array2D

class datasets.Array2Ddatasets.Array2Dhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L578[{"name": "shape", "val": ": tuple"}, {"name": "dtype", "val": ": str"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **shape** (`tuple`) --
Size of each dimension.

- **dtype** (`str`) --
  Name of the data type.0
  Create a two-dimensional array.

Example:

```
>>> from datasets import Features
>>> features = Features({'x': Array2D(shape=(1, 3), dtype='int32')})
```

class datasets.Array3Ddatasets.Array3Dhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L603[{"name": "shape", "val": ": tuple"}, {"name": "dtype", "val": ": str"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **shape** (`tuple`) --
Size of each dimension.

- **dtype** (`str`) --
  Name of the data type.0
  Create a three-dimensional array.

Example:

```
>>> from datasets import Features
>>> features = Features({'x': Array3D(shape=(1, 2, 3), dtype='int32')})
```

class datasets.Array4Ddatasets.Array4Dhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L628[{"name": "shape", "val": ": tuple"}, {"name": "dtype", "val": ": str"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **shape** (`tuple`) --
Size of each dimension.

- **dtype** (`str`) --
  Name of the data type.0

Create a four-dimensional array.

Example:

```
>>> from datasets import Features
>>> features = Features({'x': Array4D(shape=(1, 2, 2, 3), dtype='int32')})
```

class datasets.Array5Ddatasets.Array5Dhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/features.py#L653[{"name": "shape", "val": ": tuple"}, {"name": "dtype", "val": ": str"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **shape** (`tuple`) --
Size of each dimension.

- **dtype** (`str`) --
  Name of the data type.0
  Create a five-dimensional array.

Example:

```
>>> from datasets import Features
>>> features = Features({'x': Array5D(shape=(1, 2, 2, 3, 3), dtype='int32')})
```

Audiodatasets.Audio

class datasets.Audiodatasets.Audiohttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/audio.py#L24[{"name": "sampling_rate", "val": ": typing.Optional[int] = None"}, {"name": "decode", "val": ": bool = True"}, {"name": "stream_index", "val": ": typing.Optional[int] = None"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **sampling_rate** (`int` , *optional*) --
Target sampling rate. If `None` , the native sampling rate is used.

- **mono** (`bool` , defaults to `True`) --
  Whether to convert the audio signal to mono by averaging samples across channels.
- **decode** (`bool` , defaults to `True`) --
  Whether to decode the audio data. If `False` ,
  returns the underlying dictionary in the format

`{"path": audio_path, "bytes": audio_bytes}`.

- **stream_index** (`int` , *optional*) --
  The streaming index to use from the file. If `None` defaults to the "best" index.0
  Audio `Feature` to extract audio data from an audio file.

Input: The Audio feature accepts as input:

- A `str` : Absolute path to the audio file (i.e. random access is allowed).
- A `pathlib.Path` : path to the audio file (i.e. random access is allowed).
- A `dict` with the keys:
  - `path` : String with relative path of the audio file to the archive file.
  - `bytes` : Bytes content of the audio file.

  This is useful for parquet or webdataset files which embed audio files.
- A `dict` with the keys:
  - `array` : Array containing the audio sample
  - `sampling_rate` : Integer corresponding to the sampling rate of the audio sample.
- A `torchcodec.decoders.AudioDecoder` : torchcodec audio decoder object.

Output: The Audio features output data as `torchcodec.decoders.AudioDecoder` objects, with additional keys:

- `array` : Array containing the audio sample
- `sampling_rate` : Integer corresponding to the sampling rate of the audio sample.

Example:

```
>>> from datasets import load_dataset, Audio
>>> ds = load_dataset("PolyAI/minds14", name="en-US", split="train")
>>> ds = ds.cast_column("audio", Audio(sampling_rate=44100))
>>> ds[0]["audio"]
<datasets.features._torchcodec.AudioDecoder object at 0x11642b6a0>
>>> audio = ds[0]["audio"]
>>> audio.get_samples_played_in_range(0, 10)
AudioSamples:
    data (shape): torch.Size([2, 110592])
    pts_seconds: 0.0
    duration_seconds: 2.507755102040816
    sample_rate: 44100
```

cast_storagedatasets.Audio.cast_storagehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/audio.py#L223[{"name": "storage", "val": ": typing.Union[pyarrow.lib.StringArray, pyarrow.lib.StructArray]"}]- **storage** (`Union[pa.StringArray, pa.StructArray]`) --
PyArrow array to cast.0 `pa.StructArray` Array in the Audio arrow storage type, that is `pa.struct({"bytes": pa.binary(), "path": pa.string()})`
Cast an Arrow array to the Audio arrow storage type.
The Arrow types that can be converted to the Audio pyarrow storage type are:

- `pa.string()` - it must contain the "path" data
- `pa.binary()` - it must contain the audio bytes
- `pa.struct({"bytes": pa.binary()})`
- `pa.struct({"path": pa.string()})`
- `pa.struct({"bytes": pa.binary(), "path": pa.string()})` - order doesn't matter

decode_exampledatasets.Audio.decode_examplehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/audio.py#L159[{"name": "value", "val": ": dict"}, {"name": "token_per_repo_id", "val": ": typing.Optional[dict[str, typing.Union[str, bool, NoneType]]] = None"}]- **value** (`dict`) --
A dictionary with keys:

- `path` : String with relative audio file path.
- `bytes` : Bytes of the audio file.
- **token_per_repo_id** (`dict` , *optional*) --
  To access and decode
  audio files from private repositories on the Hub, you can pass
  a dictionary repo_id (`str`) -> token (`bool` or `str`)0 `torchcodec.decoders.AudioDecoder`
  Decode example audio file into audio data.

embed_storagedatasets.Audio.embed_storagehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/audio.py#L263[{"name": "storage", "val": ": StructArray"}, {"name": "token_per_repo_id", "val": " = None"}]- **storage** (`pa.StructArray`) --
PyArrow array to embed.0 `pa.StructArray` Array in the Audio arrow storage type, that is `pa.struct({"bytes": pa.binary(), "path": pa.string()})` .
Embed audio files into the Arrow array.

encode_exampledatasets.Audio.encode_examplehttps://github.com/huggingface/datasets/

blob/4.2.0/src/datasets/features/audio.py#L91[{"name": "value", "val": ": typing.Union[str, bytes, bytearray, dict, ForwardRef('AudioDecoder')]"}]- **value** (`str` , `bytes` , `bytearray` , `dict` , `AudioDecoder`) --
Data passed as input to Audio feature.0 `dict`
Encode example into a format for Arrow.

flattendatasets.Audio.flattenhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/audio.py#L212[]
If in the decodable state, raise an error, otherwise flatten the feature into a dictionary.

Imagedatasets.Image

class datasets.Imagedatasets.Imagehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/image.py#L47[{"name": "mode", "val": ": typing.Optional[str] = None"}, {"name": "decode", "val": ": bool = True"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **mode** (`str` , *optional*) --
The mode to convert the image to. If `None` , the native mode of the image is used.

- **decode** (`bool` , defaults to `True`) --
  Whether to decode the image data. If `False` ,
  returns the underlying dictionary in the format
  `{"path": image_path, "bytes": image_bytes}` .0
  Image `Feature` to read image data from an image file.

Input: The Image feature accepts as input:

- A `str` : Absolute path to the image file (i.e. random access is allowed).
- A `pathlib.Path` : path to the image file (i.e. random access is allowed).
- A `dict` with the keys:
  - `path` : String with relative path of the image file to the archive file.
  - `bytes` : Bytes of the image file.

  This is useful for parquet or webdataset files which embed image files.
- An `np.ndarray` : NumPy array representing an image.
- A `PIL.Image.Image` : PIL image object.

Output: The Image features output data as `PIL.Image.Image` objects.

Examples:

```
>>> from datasets import load_dataset, Image
>>> ds = load_dataset("AI-Lab-Makerere/beans", split="train")
>>> ds.features["image"]
Image(decode=True, id=None)
>>> ds[0]["image"]
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=500x500 at 0x15E52E7F0>
>>> ds = ds.cast_column('image', Image(decode=False))
{'bytes': None,
 'path': '/root/.cache/huggingface/datasets/downloads/extracted/b0a21163f78769a2cf11f58dfc767fb458
```

cast_storagedatasets.Image.cast_storagehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/image.py#L213[{"name": "storage", "val": ": typing.Union[pyarrow.lib.StringArray, pyarrow.lib.StructArray, pyarrow.lib.ListArray]"}]- **storage** (`Union[pa.StringArray, pa.StructArray, pa.ListArray]`) --
PyArrow array to cast.0 `pa.StructArray` Array in the Image arrow storage type, that is `pa.struct({"bytes": pa.binary(), "path": pa.string()})` .
Cast an Arrow array to the Image arrow storage type.
The Arrow types that can be converted to the Image pyarrow storage type are:

- `pa.string()` - it must contain the "path" data
- `pa.binary()` - it must contain the image bytes
- `pa.struct({"bytes": pa.binary()})`
- `pa.struct({"path": pa.string()})`
- `pa.struct({"bytes": pa.binary(), "path": pa.string()})` - order doesn't matter
- `pa.list(*)` - it must contain the image array data

decode_exampledatasets.Image.decode_examplehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/image.py#L139[{"name": "value", "val": ": dict"}, {"name": "token_per_repo_id", "val": " = None"}]- **value** (`str` or `dict`) --
A string with the absolute image file path, a dictionary with
keys:

- `path` : String with absolute or relative image file path.
- `bytes` : The bytes of the image file.
- **token_per_repo_id** (`dict` , *optional*) --

To access and decode
image files from private repositories on the Hub, you can pass
a dictionary repo_id (`str`) -> token (`bool` or `str`).0`PIL.Image.Image`
Decode example image file into image data.

embed_storagedatasets.Image.embed_storagehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/image.py#L259[{"name": "storage", "val": ": StructArray"}, {"name": "token_per_repo_id", "val": " = None"}]- **storage** (`pa.StructArray`) --
PyArrow array to embed.0`pa.StructArray`Array in the Image arrow storage type, that is `pa.struct({"bytes": pa.binary(), "path": pa.string()})`.
Embed image files into the Arrow array.

encode_exampledatasets.Image.encode_examplehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/image.py#L98[{"name": "value", "val": ": typing.Union[str, bytes, bytearray, dict, numpy.ndarray, ForwardRef('PIL.Image.Image')]"}]- **value** (`str`, `np.ndarray`, `PIL.Image.Image` or `dict`) --
Data passed as input to Image feature.0`dict` with "path" and "bytes" fields
Encode example into a format for Arrow.

flattendatasets.Image.flattenhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/image.py#L200[]
If in the decodable state, return the feature itself, otherwise flatten the feature into a dictionary.

Videodatasets.Video

class datasets.Videodatasets.Videohttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/video.py#L29[{"name": "decode", "val": ": bool = True"}, {"name": "stream_index", "val": ": typing.Optional[int] = None"}, {"name": "dimension_order", "val": ": typing.Literal['NCHW', 'NHWC'] = 'NCHW'"}, {"name": "num_ffmpeg_threads", "val": ": int = 1"}, {"name": "device", "val": ": typing.Union[str, ForwardRef('torch.device'), NoneType] = 'cpu'"}, {"name": "seek_mode", "val": ": typing.Literal['exact', 'approximate'] = 'exact'"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **mode** (`str`, *optional*) --
The mode to convert the video to. If `None`, the native mode of the video is used.

- **decode** (`bool`, defaults to `True`) --
  Whether to decode the video data. If `False`,
  returns the underlying dictionary in the format

`{"path": video_path, "bytes": video_bytes}`.

- **stream_index** (`int`, *optional*) --
  The streaming index to use from the file. If `None` defaults to the "best" index.
- **dimension_order** (`str`, defaults to `NCHW`) --
  The dimension order of the decoded frames.
  where N is the batch size, C is the number of channels,
  H is the height, and W is the width of the frames.
- **num_ffmpeg_threads** (`int`, defaults to `1`) --
  The number of threads to use for decoding the video. (Recommended to keep this at 1)
- **device** (`str` or `torch.device`, defaults to `cpu`) --
  The device to use for decoding the video.
- **seek_mode** (`str`, defaults to `exact`) --
  Determines if frame access will be “exact” or “approximate”.
  Exact guarantees that requesting frame i will always return frame i, but doing so requires an initial scan of the file.
  Approximate is faster as it avoids scanning the file, but less accurate as it uses the file's metadata to calculate where i probably is.
  read more here0

Video `Feature` to read video data from a video file.

Input: The Video feature accepts as input:

- A `str`: Absolute path to the video file (i.e. random access is allowed).
- A `pathlib.Path`: path to the video file (i.e. random access is allowed).
- A `dict` with the keys:
  - `path`: String with relative path of the video file in a dataset repository.
  - `bytes`: Bytes of the video file.
  This is useful for parquet or webdataset files which embed video files.
- A `torchcodec.decoders.VideoDecoder`: torchcodec video decoder object.

Output: The Video features output data as `torchcodec.decoders.VideoDecoder` objects.

Examples:

```
>>> from datasets import Dataset, Video
>>> ds = Dataset.from_dict({"video":["path/to/Screen Recording.mov"]}).cast_column("video", Video(
>>> ds.features["video"]
Video(decode=True, id=None)
>>> ds[0]["video"]
<torchcodec.decoders._video_decoder.VideoDecoder object at 0x14a61e080>
>>> video = ds[0]["video"]
>>> video.get_frames_in_range(0, 10)
FrameBatch:
data (shape): torch.Size([10, 3, 50, 66])
pts_seconds: tensor([0.4333, 0.4333, 0.4333, 0.4333, 0.4333, 0.4333, 0.4333, 0.4333, 0.4333,
        0.4333], dtype=torch.float64)
duration_seconds: tensor([0.0167, 0.0167, 0.0167, 0.0167, 0.0167, 0.0167, 0.0167, 0.0167, 0.0167,
        0.0167], dtype=torch.float64)
>>> ds.cast_column('video', Video(decode=False))[0]["video]
{'bytes': None,
 'path': 'path/to/Screen Recording.mov'}
```

cast_storagedatasets.Video.cast_storagehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/video.py#L241[{"name": "storage", "val": ": typing.Union[pyarrow.lib.StringArray, pyarrow.lib.StructArray, pyarrow.lib.ListArray]"}]- **storage** (`Union[pa.StringArray, pa.StructArray, pa.ListArray]`) -- PyArrow array to cast.0 `pa.StructArray` Array in the Video arrow storage type, that is `pa.struct({"bytes": pa.binary(), "path": pa.string()})` .
Cast an Arrow array to the Video arrow storage type.
The Arrow types that can be converted to the Video pyarrow storage type are:

- `pa.string()` - it must contain the "path" data
- `pa.binary()` - it must contain the video bytes
- `pa.struct({"bytes": pa.binary()})`
- `pa.struct({"path": pa.string()})`
- `pa.struct({"bytes": pa.binary(), "path": pa.string()})` - order doesn't matter
- `pa.list(*)` - it must contain the video array data

decode_exampledatasets.Video.decode_examplehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/video.py#L155[{"name": "value", "val": ": typing.Union[str, datasets.features.video.Example]"}, {"name": "token_per_repo_id", "val": ":

typing.Optional[dict[str, typing.Union[bool, str]]] = None"}]- **value** (`str` or `dict`) --
A string with the absolute video file path, a dictionary with
keys:

- `path` : String with absolute or relative video file path.
- `bytes` : The bytes of the video file.
- **token_per_repo_id** (`dict` , *optional*) --
  To access and decode
  video files from private repositories on the Hub, you can pass
  a dictionary repo_id (`str`) -> token (`bool` or `str`).0 `torchcodec.decoders.VideoDecoder`
  Decode example video file into video data.

encode_exampledatasets.Video.encode_examplehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/video.py#L107[{"name": "value", "val": ": typing.Union[str, bytes, bytearray, datasets.features.video.Example, numpy.ndarray, ForwardRef('VideoDecoder')]"}]- **value** (`str` , `np.ndarray` , `bytes` , `bytearray` , `VideoDecoder` or `dict`) --
Data passed as input to Video feature.0 `dict` with "path" and "bytes" fields
Encode example into a format for Arrow.

flattendatasets.Video.flattenhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/video.py#L228[]
If in the decodable state, return the feature itself, otherwise flatten the feature into a dictionary.

Pdfdatasets.Pdf

class datasets.Pdfdatasets.Pdfhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/pdf.py#L31[{"name": "decode", "val": ": bool = True"}, {"name": "id", "val": ": typing.Optional[str] = None"}]- **mode** (`str` , *optional*) --
The mode to convert the pdf to. If `None` , the native mode of the pdf is used.

- **decode** (`bool` , defaults to `True`) --
  Whether to decode the pdf data. If `False` ,
  returns the underlying dictionary in the format `{"path": pdf_path, "bytes": pdf_bytes}` .0

Experimental.
Pdf `Feature` to read pdf documents from a pdf file.

Input: The Pdf feature accepts as input:

- A `str` : Absolute path to the pdf file (i.e. random access is allowed).
- A `pathlib.Path` : path to the pdf file (i.e. random access is allowed).
- A `dict` with the keys:
  - `path` : String with relative path of the pdf file in a dataset repository.
  - `bytes` : Bytes of the pdf file.
    This is useful for archived files with sequential access.
- A `pdfplumber.pdf.PDF` : pdfplumber pdf object.

Examples:

```
>>> from datasets import Dataset, Pdf
>>> ds = Dataset.from_dict({"pdf": ["path/to/pdf/file.pdf"]}).cast_column("pdf", Pdf())
>>> ds.features["pdf"]
Pdf(decode=True, id=None)
>>> ds[0]["pdf"]
<pdfplumber.pdf.PDF object at 0x7f8a1c2d8f40>
>>> ds = ds.cast_column("pdf", Pdf(decode=False))
>>> ds[0]["pdf"]
{'bytes': None,
'path': 'path/to/pdf/file.pdf'}
```

cast_storagedatasets.Pdf.cast_storagehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/pdf.py#L186[{"name": "storage", "val": ": typing.Union[pyarrow.lib.StringArray, pyarrow.lib.StructArray, pyarrow.lib.ListArray]"}]- **storage**
(`Union[pa.StringArray, pa.StructArray, pa.ListArray]`) --
PyArrow array to cast.0 `pa.StructArray` Array in the Pdf arrow storage type, that is `pa.struct({"bytes": pa.binary(), "path": pa.string()})` .
Cast an Arrow array to the Pdf arrow storage type.
The Arrow types that can be converted to the Pdf pyarrow storage type are:

- `pa.string()` - it must contain the "path" data
- `pa.binary()` - it must contain the image bytes
- `pa.struct({"bytes": pa.binary()})`
- `pa.struct({"path": pa.string()})`
- `pa.struct({"bytes": pa.binary(), "path": pa.string()})` - order doesn't matter

- `pa.list(*)` - it must contain the pdf array data

decode_exampledatasets.Pdf.decode_examplehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/pdf.py#L115[{"name": "value", "val": ": dict"}, {"name": "token_per_repo_id", "val": " = None"}]- **value** (`str` or `dict`) --
A string with the absolute pdf file path, a dictionary with
keys:

- `path` : String with absolute or relative pdf file path.
- `bytes` : The bytes of the pdf file.
- **token_per_repo_id** (`dict` , *optional*) --
  To access and decode pdf files from private repositories on
  the Hub, you can pass a dictionary
  repo_id (`str`) -> token (`bool` or `str`).0 `pdfplumber.pdf.PDF`
  Decode example pdf file into pdf data.

embed_storagedatasets.Pdf.embed_storagehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/pdf.py#L223[{"name": "storage", "val": ": StructArray"}, {"name": "token_per_repo_id", "val": " = None"}]- **storage** (`pa.StructArray`) --
PyArrow array to embed.0 `pa.StructArray` Array in the PDF arrow storage type, that is `pa.struct({"bytes": pa.binary(), "path": pa.string()})` .
Embed PDF files into the Arrow array.

encode_exampledatasets.Pdf.encode_examplehttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/pdf.py#L80[{"name": "value", "val": ": typing.Union[str, bytes, bytearray, dict, ForwardRef('pdfplumber.pdf.PDF')]"}]- **value** (`str` , `bytes` , `pdfplumber.pdf.PDF` or `dict`) --
Data passed as input to Pdf feature.0 `dict` with "path" and "bytes" fields
Encode example into a format for Arrow.

flattendatasets.Pdf.flattenhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/features/pdf.py#L173[]
If in the decodable state, return the feature itself, otherwise flatten the feature into a dictionary.

Filesystemsdatasets.filesystems.is_remote_filesystem

datasets.filesystems.is_remote_filesystemdatasets.filesystems.is_remote_filesystemhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/filesystems/__init__.py#L28[{"name": "fs", "val": ": AbstractFileSystem"}]- **fs** (`fsspec.spec.AbstractFileSystem`) --
An abstract super-class for pythonic file-systems, e.g. `fsspec.filesystem('file')` or `s3fs.S3FileSystem`.0

Checks if `fs` is a remote filesystem.

Fingerprintdatasets.fingerprint.Hasher

class datasets.fingerprint.Hasherdatasets.fingerprint.Hasherhttps://github.com/huggingface/datasets/blob/4.2.0/src/datasets/fingerprint.py#L170[]
Hasher that accepts python objects as inputs.