Process

🤗 Datasets provides many tools for modifying the structure and content of a dataset. These tools are important for tidying up a dataset, creating additional columns, converting between features and formats, and much more.

This guide will show you how to:

- Reorder rows and split the dataset.
- Rename and remove columns, and other common column operations.
- Apply processing functions to each example in a dataset.
- Concatenate datasets.
- Apply a custom formatting transform.
- Save and export processed datasets.

For more details specific to processing other dataset modalities, take a look at the process audio dataset guide, the process image dataset guide, or the process text dataset guide.

The examples in this guide use the MRPC dataset, but feel free to load any dataset of your choice and follow along!

```
>>> from datasets import load_dataset
>>> dataset = load_dataset("nyu-mll/glue", "mrpc", split="train")
```

> [!WARNING]
> All processing methods in this guide return a new Dataset object. Modification is not done in-place. Be careful about overriding your previous dataset!

Sort, shuffle, select, split, and shard

There are several functions for rearranging the structure of a dataset.
These functions are useful for selecting only the rows you want, creating train and test splits, and sharding very large datasets into smaller chunks.

Sort

Use sort() to sort column values according to their numerical values. The provided column must be NumPy compatible.

```
>>> dataset["label"][:10]
[1, 0, 1, 0, 1, 1, 0, 1, 0, 0]
>>> sorted_dataset = dataset.sort("label")
>>> sorted_dataset["label"][:10]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> sorted_dataset["label"][-10:]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Under the hood, this creates a list of indices that is sorted according to values of the column. This indices mapping is then used to access the right rows in the underlying Arrow table.

Shuffle

The shuffle() function randomly rearranges the column values. You can specify the `generator` parameter in this function to use a different `numpy.random.Generator` if you want more control over the algorithm used to shuffle the dataset.

```
>>> shuffled_dataset = sorted_dataset.shuffle(seed=42)
>>> shuffled_dataset["label"][:10]
[1, 1, 1, 0, 1, 1, 1, 1, 1, 0]
```

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.
Alternatively, you can switch to an IterableDataset and leverage its fast approximate shuffling IterableDataset.shuffle():

```
>>> iterable_dataset = dataset.to_iterable_dataset(num_shards=128)
>>> shuffled_iterable_dataset = iterable_dataset.shuffle(seed=42, buffer_size=1000)
```

Select and Filter

There are two options for filtering rows in a dataset: select() and filter().

- select() returns rows according to a list of indices:

```
>>> small_dataset = dataset.select([0, 10, 20, 30, 40, 50])
>>> len(small_dataset)
6
```

- filter() returns rows that match a specified condition:

```
>>> start_with_ar = dataset.filter(lambda example: example["sentence1"].startswith("Ar"))
>>> len(start_with_ar)
6
>>> start_with_ar["sentence1"]
['Around 0335 GMT , Tab shares were up 19 cents , or 4.4 % , at A $ 4.56 , having earlier set a re
'Arison said Mann may have been one of the pioneers of the world music movement and he had a deep
'Arts helped coach the youth on an eighth-grade football team at Lombardi Middle School in Green B
'Around 9 : 00 a.m. EDT ( 1300 GMT ) , the euro was at $ 1.1566 against the dollar , up 0.07 perce
"Arguing that the case was an isolated example , Canada has threatened a trade backlash if Tokyo '
'Artists are worried the plan would harm those who need help most - performers who have a difficul
]
```

filter() can also filter by indices if you set `with_indices=True`:

```
>>> even_dataset = dataset.filter(lambda example, idx: idx % 2 == 0, with_indices=True)
>>> len(even_dataset)
1834
>>> len(dataset) / 2
1834.0
```

Unless the list of indices to keep is contiguous, those methods also create an indices mapping under the hood.

Split

The train_test_split() function creates train and test splits if your dataset doesn't already have them. This allows you to adjust the relative proportions or an absolute number of samples in each split. In the example below, use the `test_size` parameter to create a test split that is 10% of the original dataset:

```
>>> dataset.train_test_split(test_size=0.1)
{'train': Dataset(schema: {'sentence1': 'string', 'sentence2': 'string', 'label': 'int64', 'idx':
'test': Dataset(schema: {'sentence1': 'string', 'sentence2': 'string', 'label': 'int64', 'idx': 'i
>>> 0.1 * len(dataset)
366.8
```

The splits are shuffled by default, but you can set `shuffle=False` to prevent shuffling.

Shard

🤗 Datasets supports sharding to divide a very large dataset into a predefined number of chunks. Specify the `num_shards` parameter in shard() to determine the number of shards to split the dataset into. You'll also need to provide the shard you want to return with the `index` parameter.

For example, the stanfordnlp/imdb dataset has 25000 examples:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset("stanfordnlp/imdb", split="train")
>>> print(dataset)
Dataset({
    features: ['text', 'label'],
    num_rows: 25000
})
```

After sharding the dataset into four chunks, the first shard will only have 6250 examples:

```
>>> dataset.shard(num_shards=4, index=0)
Dataset({
    features: ['text', 'label'],
    num_rows: 6250
})
>>> print(25000/4)
6250.0
```

Rename, remove, cast, and flatten

The following functions allow you to modify the columns of a dataset. These functions are useful for renaming or removing columns, changing columns to a new set of features, and flattening nested column structures.

Rename

Use rename_column() when you need to rename a column in your dataset. Features associated with the original column are actually moved under the new column name, instead of just replacing the original column in-place.

Provide rename_column() with the name of the original column, and the new column name:

```
>>> dataset
Dataset({
    features: ['sentence1', 'sentence2', 'label', 'idx'],
    num_rows: 3668
})
>>> dataset = dataset.rename_column("sentence1", "sentenceA")
>>> dataset = dataset.rename_column("sentence2", "sentenceB")
>>> dataset
Dataset({
    features: ['sentenceA', 'sentenceB', 'label', 'idx'],
    num_rows: 3668
})
```

Remove

When you need to remove one or more columns, provide the column name to remove to the remove_columns() function. Remove more than one column by providing a list of column names:

```
>>> dataset = dataset.remove_columns("label")
>>> dataset
Dataset({
    features: ['sentence1', 'sentence2', 'idx'],
    num_rows: 3668
})
>>> dataset = dataset.remove_columns(["sentence1", "sentence2"])
>>> dataset
Dataset({
    features: ['idx'],
    num_rows: 3668
})
```

Conversely, select_columns() selects one or more columns to keep and removes the rest. This function takes either one or a list of column names:

```
>>> dataset
Dataset({
    features: ['sentence1', 'sentence2', 'label', 'idx'],
    num_rows: 3668
})
>>> dataset = dataset.select_columns(['sentence1', 'sentence2', 'idx'])
>>> dataset
Dataset({
    features: ['sentence1', 'sentence2', 'idx'],
    num_rows: 3668
})
>>> dataset = dataset.select_columns('idx')
>>> dataset
Dataset({
    features: ['idx'],
    num_rows: 3668
})
```

Cast

The cast() function transforms the feature type of one or more columns. This function accepts your new Features as its argument. The example below demonstrates how to change the ClassLabel and Value features:

```
>>> dataset.features
{'sentence1': Value('string'),
'sentence2': Value('string'),
'label': ClassLabel(names=['not_equivalent', 'equivalent']),
'idx': Value('int32')}

>>> from datasets import ClassLabel, Value
>>> new_features = dataset.features.copy()
>>> new_features["label"] = ClassLabel(names=["negative", "positive"])
>>> new_features["idx"] = Value("int64")
>>> dataset = dataset.cast(new_features)
>>> dataset.features
{'sentence1': Value('string'),
'sentence2': Value('string'),
'label': ClassLabel(names=['negative', 'positive']),
'idx': Value('int64')}
```

> [!TIP]
> Casting only works if the original feature type and new feature type are compatible. For example, you can cast a column with the feature type `Value("int32")` to `Value("bool")` if the original column only contains ones and zeros.

Use the cast_column() function to change the feature type of a single column. Pass the column name and its new feature type as arguments:

```
>>> dataset.features
{'audio': Audio(sampling_rate=44100, mono=True)}

>>> dataset = dataset.cast_column("audio", Audio(sampling_rate=16000))
>>> dataset.features
{'audio': Audio(sampling_rate=16000, mono=True)}
```

Flatten

Sometimes a column can be a nested structure of several types. Take a look at the nested structure below from the SQuAD dataset:

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

The `answers` field contains two subfields: `text` and `answer_start`. Use the flatten() function to extract the subfields into their own separate columns:

```
>>> flat_dataset = dataset.flatten()
>>> flat_dataset
Dataset({
    features: ['id', 'title', 'context', 'question', 'answers.text', 'answers.answer_start'],
 num_rows: 87599
})
```

Notice how the subfields are now their own independent columns: `answers.text` and `answers.answer_start`.

Map

Some of the more powerful applications of 🤗 Datasets come from using the map() function. The primary purpose of map() is to speed up processing functions. It allows you to apply a processing function to each example in a dataset, independently or in batches. This function can even create new rows and columns.

In the following example, prefix each `sentence1` value in the dataset with `'My sentence: '`.

Start by creating a function that adds `'My sentence: '` to the beginning of each sentence. The function needs to accept and output a `dict`:

```
>>> def add_prefix(example):
...     example["sentence1"] = 'My sentence: ' + example["sentence1"]
...     return example
```

Now use map() to apply the `add_prefix` function to the entire dataset:

```
>>> updated_dataset = small_dataset.map(add_prefix)
>>> updated_dataset["sentence1"][:5]
['My sentence: Amrozi accused his brother , whom he called " the witness " , of deliberately disto
"My sentence: Yucaipa owned Dominick 's before selling the chain to Safeway in 1998 for $ 2.5 bill
'My sentence: They had published an advertisement on the Internet on June 10 , offering the cargo
'My sentence: Around 0335 GMT , Tab shares were up 19 cents , or 4.4 % , at A $ 4.56 , having earl
]
```

Let's take a look at another example, except this time, you'll remove a column with map(). When you remove a column, it is only removed after the example has been provided to the mapped function. This allows the mapped function to use the content of the columns before they are removed.

Specify the column to remove with the `remove_columns` parameter in map():

```
>>> updated_dataset = dataset.map(lambda example: {"new_sentence": example["sentence1"]}, remove_c
>>> updated_dataset.column_names
['sentence2', 'label', 'idx', 'new_sentence']
```

> [!TIP]
> Datasets also has a remove_columns() function which is faster because it doesn't copy the data of the remaining columns.

You can also use map() with indices if you set `with_indices=True`. The example below adds the index to the beginning of each sentence:

```
>>> updated_dataset = dataset.map(lambda example, idx: {"sentence2": f"{idx}: " + example["sentenc
>>> updated_dataset["sentence2"][:5]
['0: Referring to him as only " the witness " , Amrozi accused his brother of deliberately distort
 "1: Yucaipa bought Dominick 's in 1995 for $ 693 million and sold it to Safeway for $ 1.8 billion
 "2: On June 10 , the ship 's owners had published an advertisement on the Internet , offering the
 '3: Tab shares jumped 20 cents , or 4.6 % , to set a record closing high at A $ 4.57 .',
 '4: PG & E Corp. shares jumped $ 1.63 or 8 percent to $ 21.03 on the New York Stock Exchange on F
]
```

Multiprocessing

Multiprocessing significantly speeds up processing by parallelizing processes on the CPU. Set the `num_proc` parameter in map() to set the number of processes to use:

```
>>> updated_dataset = dataset.map(lambda example, idx: {"sentence2": f"{idx}: " + example["sentenc
```

The map() also works with the rank of the process if you set `with_rank=True`. This is analogous to the `with_indices` parameter. The `with_rank` parameter in the mapped function goes after the `index` one if it is already present.

```
>>> import torch
>>> from multiprocess import set_start_method
>>> from transformers import AutoTokenizer, AutoModelForCausalLM
>>> from datasets import load_dataset
>>>
>>> # Get an example dataset
>>> dataset = load_dataset("fka/awesome-chatgpt-prompts", split="train")
>>>
>>> # Get an example model and its tokenizer
>>> model = AutoModelForCausalLM.from_pretrained("Qwen/Qwen1.5-0.5B-Chat").eval()
>>> tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen1.5-0.5B-Chat")
>>>
>>> def gpu_computation(batch, rank):
...     # Move the model on the right GPU if it's not there already
...     device = f"cuda:{(rank or 0) % torch.cuda.device_count()}"
...     model.to(device)
...
...     # Your big GPU call goes here, for example:
...     chats = [[
...         {"role": "system", "content": "You are a helpful assistant."},
...         {"role": "user", "content": prompt}
...     ] for prompt in batch["prompt"]]
...     texts = [tokenizer.apply_chat_template(
...         chat,
...         tokenize=False,
...         add_generation_prompt=True
...     ) for chat in chats]
...     model_inputs = tokenizer(texts, padding=True, return_tensors="pt").to(device)
...     with torch.no_grad():
...         outputs = model.generate(**model_inputs, max_new_tokens=512)
...     batch["output"] = tokenizer.batch_decode(outputs, skip_special_tokens=True)
...     return batch
>>>
>>> if __name__ == "__main__":
...     set_start_method("spawn")
...     updated_dataset = dataset.map(
...         gpu_computation,
...         batched=True,
...         batch_size=16,
```

```
...         with_rank=True,
...         num_proc=torch.cuda.device_count(),  # one process per GPU
...     )
```

The main use-case for rank is to parallelize computation across several GPUs. This requires setting `multiprocess.set_start_method("spawn")`. If you don't you'll receive the following CUDA error:

```
RuntimeError: Cannot re-initialize CUDA in forked subprocess. To use CUDA with multiprocessing, yo
```

Batch processing

The map() function supports working with batches of examples. Operate on batches by setting `batched=True`. The default batch size is 1000, but you can adjust it with the `batch_size` parameter. Batch processing enables interesting applications such as splitting long sentences into shorter chunks and data augmentation.

Split long examples

When examples are too long, you may want to split them into several smaller chunks. Begin by creating a function that:

1. Splits the `sentence1` field into chunks of 50 characters.
2. Stacks all the chunks together to create the new dataset.

```
>>> def chunk_examples(examples):
...     chunks = []
...     for sentence in examples["sentence1"]:
...         chunks += [sentence[i:i + 50] for i in range(0, len(sentence), 50)]
...     return {"chunks": chunks}
```

Apply the function with map():

```
>>> chunked_dataset = dataset.map(chunk_examples, batched=True, remove_columns=dataset.column_name
>>> chunked_dataset[:10]
{'chunks': ['Amrozi accused his brother , whom he called " the ',
            'witness " , of deliberately distorting his evidenc',
            'e .',
            "Yucaipa owned Dominick 's before selling the chain",
            ' to Safeway in 1998 for $ 2.5 billion .',
            'They had published an advertisement on the Interne',
            't on June 10 , offering the cargo for sale , he ad',
            'ded .',
            'Around 0335 GMT , Tab shares were up 19 cents , or',
            ' 4.4 % , at A $ 4.56 , having earlier set a record']}
```

Notice how the sentences are split into shorter chunks now, and there are more rows in the dataset.

```
>>> dataset
Dataset({
 features: ['sentence1', 'sentence2', 'label', 'idx'],
 num_rows: 3668
})
>>> chunked_dataset
Dataset({
    features: ['chunks'],
    num_rows: 10470
})
```

Data augmentation

The map() function could also be used for data augmentation. The following example generates additional words for a masked token in a sentence.

Load and use the RoBERTA model in 🤗 Transformers' FillMaskPipeline:

```
>>> from random import randint
>>> from transformers import pipeline

>>> fillmask = pipeline("fill-mask", model="roberta-base")
>>> mask_token = fillmask.tokenizer.mask_token
>>> smaller_dataset = dataset.filter(lambda e, i: i<100, with_indices=True)
```

Create a function to randomly select a word to mask in the sentence. The function should also return the original sentence and the top two replacements generated by RoBERTA.

```
>>> def augment_data(examples):
...     outputs = []
...     for sentence in examples["sentence1"]:
...         words = sentence.split(' ')
...         K = randint(1, len(words)-1)
...         masked_sentence = " ".join(words[:K]  + [mask_token] + words[K+1:])
...         predictions = fillmask(masked_sentence)
...         augmented_sequences = [predictions[i]["sequence"] for i in range(3)]
...         outputs += [sentence] + augmented_sequences
...
...     return {"data": outputs}
```

Use map() to apply the function over the whole dataset:

```
>>> augmented_dataset = smaller_dataset.map(augment_data, batched=True, remove_columns=dataset.col
>>> augmented_dataset[:9]["data"]
['Amrozi accused his brother , whom he called " the witness " , of deliberately distorting his evi
 'Amrozi accused his brother, whom he called " the witness ", of deliberately withholding his evid
 'Amrozi accused his brother, whom he called " the witness ", of deliberately suppressing his evid
 'Amrozi accused his brother, whom he called " the witness ", of deliberately destroying his evide
 "Yucaipa owned Dominick 's before selling the chain to Safeway in 1998 for $ 2.5 billion .",
 'Yucaipa owned Dominick Stores before selling the chain to Safeway in 1998 for $ 2.5 billion.',
 "Yucaipa owned Dominick's before selling the chain to Safeway in 1998 for $ 2.5 billion.",
 'Yucaipa owned Dominick Pizza before selling the chain to Safeway in 1998 for $ 2.5 billion.'
]
```

For each original sentence, RoBERTA augmented a random word with three alternatives. The

original word `distorting` is supplemented by `withholding`, `suppressing`, and `destroying`.

Asynchronous processing

Asynchronous functions are useful to call API endpoints in parallel, for example to download content like images or call a model endpoint.

You can define an asynchronous function using the `async` and `await` keywords, here is an example function to call a chat model from Hugging Face:

```
>>> import aiohttp
>>> import asyncio
>>> from huggingface_hub import get_token
>>> sem = asyncio.Semaphore(20)  # max number of simultaneous queries
>>> async def query_model(model, prompt):
...     api_url = f"https://api-inference.huggingface.co/models/{model}/v1/chat/completions"
...     headers = {"Authorization": f"Bearer {get_token()}", "Content-Type": "application/json"}
...     json = {"messages": [{"role": "user", "content": prompt}], "max_tokens": 20, "seed": 42}
...     async with sem, aiohttp.ClientSession() as session, session.post(api_url, headers=headers,
...         output = await response.json()
...         return {"Output": output["choices"][0]["message"]["content"]}
```

Asynchronous functions run in parallel, which accelerates the process a lot. The same code takes a lot more time if it's run sequentially, because it does nothing while waiting for the model response. It is generally recommended to use `async` / `await` when you function has to wait for a response from an API for example, or if it downloads data and it can take some time.

Note the presence of a `Semaphore`: it sets the maximum number of queries that can run in parallel. It is recommended to use a `Semaphore` when calling APIs to avoid rate limit errors.

Let's use it to call the microsoft/Phi-3-mini-4k-instruct model and ask it to return the main topic of each math problem in the Maxwell-Jia/AIME_2024 dataset:

```
>>> from datasets import load_dataset
>>> ds = load_dataset("Maxwell-Jia/AIME_2024", split="train")
>>> model = "microsoft/Phi-3-mini-4k-instruct"
>>> prompt = 'What is this text mainly about ? Here is the text:\n\n```\n{Problem}\n```\n\nReply u
>>> async def get_topic(example):
...     return await query_model(model, prompt.format(Problem=example['Problem']))
>>> ds = ds.map(get_topic)
>>> ds[0]
{'ID': '2024-II-4',
 'Problem': 'Let $x,y$ and $z$ be positive real numbers that...',
 'Solution': 'Denote $\\log_2(x) = a$, $\\log_2(y) = b$, and...,
 'Answer': 33,
 'Output': 'The main topic is Logarithms.'}
```

Here, Dataset.map() runs many `get_topic` function asynchronously so it doesn't have to wait for every single model response which would take a lot of time to do sequentially.

By default, Dataset.map() runs up to one thousand map functions in parallel, so don't forget to set the maximum number of API calls that can run in parallel with a `Semaphore`, otherwise the model could return rate limit errors or overload. For advanced use cases, you can change the maximum number of queries in parallel in `datasets.config`.

Process multiple splits

Many datasets have splits that can be processed simultaneously with DatasetDict.map(). For example, tokenize the `sentence1` field in the train and test split by:

```
>>> from datasets import load_dataset

# load all the splits
>>> dataset = load_dataset('nyu-mll/glue', 'mrpc')
>>> encoded_dataset = dataset.map(lambda examples: tokenizer(examples["sentence1"]), batched=True)
>>> encoded_dataset["train"][0]
{'sentence1': 'Amrozi accused his brother , whom he called " the witness " , of deliberately disto
'sentence2': 'Referring to him as only " the witness " , Amrozi accused his brother of deliberatel
'label': 1,
'idx': 0,
'input_ids': [  101,  7277,  2180,  5303,  4806,  1117,  1711,   117,  2292, 1119,  1270,   107,
'token_type_ids': [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': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
}
```

Distributed usage

When you use map() in a distributed setting, you should also use torch.distributed.barrier. This ensures the main process performs the mapping, while the other processes load the results, thereby avoiding duplicate work.

The following example shows how you can use `torch.distributed.barrier` to synchronize the processes:

```
>>> from datasets import Dataset
>>> import torch.distributed

>>> dataset1 = Dataset.from_dict({"a": [0, 1, 2]})

>>> if training_args.local_rank > 0:
...     print("Waiting for main process to perform the mapping")
...     torch.distributed.barrier()

>>> dataset2 = dataset1.map(lambda x: {"a": x["a"] + 1})

>>> if training_args.local_rank == 0:
...     print("Loading results from main process")
...     torch.distributed.barrier()
```

Batch

The `batch()` method allows you to group samples from the dataset into batches. This is particularly useful when you want to create batches of data for training or evaluation, especially when working with deep learning models.

Here's an example of how to use the `batch()` method:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train")
>>> batched_dataset = dataset.batch(batch_size=4)
>>> batched_dataset[0]
{'text': ['the rock is destined to be the 21st century\'s new " conan " and that he\'s going to ma
        'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge tha
        'effective but too-tepid biopic',
        'if you sometimes like to go to the movies to have fun , wasabi is a good place to start .
'label': [1, 1, 1, 1]}
```

The `batch()` method accepts the following parameters:

- `batch_size` (`int`): The number of samples in each batch.
- `drop_last_batch` (`bool`, defaults to `False`): Whether to drop the last incomplete batch if the dataset size is not divisible by the batch size.
- `num_proc` (`int`, optional, defaults to `None`): The number of processes to use for multiprocessing. If None, no multiprocessing is used. This can significantly speed up batching for large datasets.

Note that `Dataset.batch()` returns a new Dataset where each item is a batch of multiple samples from the original dataset. If you want to process data in batches, you should use a batched map() directly, which applies a function to batches but the output dataset is unbatched.

Concatenate

Separate datasets can be concatenated if they share the same column types. Concatenate datasets with concatenate_datasets():

```
>>> from datasets import concatenate_datasets, load_dataset

>>> stories = load_dataset("ajibawa-2023/General-Stories-Collection", split="train")
>>> stories = stories.remove_columns([col for col in stories.column_names if col != "text"])  # on
>>> wiki = load_dataset("wikimedia/wikipedia", "20220301.en", split="train")
>>> wiki = wiki.remove_columns([col for col in wiki.column_names if col != "text"])  # only keep t

>>> assert stories.features.type == wiki.features.type
>>> bert_dataset = concatenate_datasets([stories, wiki])
```

You can also concatenate two datasets horizontally by setting `axis=1` as long as the datasets have the same number of rows:

```
>>> from datasets import Dataset
>>> stories_ids = Dataset.from_dict({"ids": list(range(len(stories)))})
>>> stories_with_ids = concatenate_datasets([stories, stories_ids], axis=1)
```

Interleave

You can also mix several datasets together by taking alternating examples from each one to create a new dataset. This is known as *interleaving*, which is enabled by the interleave_datasets() function. Both interleave_datasets() and concatenate_datasets() work with regular Dataset and IterableDataset objects.
Refer to the Stream guide for an example of how to interleave IterableDataset objects.

You can define sampling probabilities for each of the original datasets to specify how to interleave the datasets.
In this case, the new dataset is constructed by getting examples one by one from a random dataset until one of the datasets runs out of samples.

```
>>> from datasets import Dataset, interleave_datasets
>>> seed = 42
>>> probabilities = [0.3, 0.5, 0.2]
>>> 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]})
>>> dataset = interleave_datasets([d1, d2, d3], probabilities=probabilities, seed=seed)
>>> dataset["a"]
[10, 11, 20, 12, 0, 21, 13]
```

You can also specify the `stopping_strategy`. The default strategy, `first_exhausted`, is a subsampling strategy, i.e the dataset construction is stopped as soon one of the dataset runs out of samples.
You can specify `stopping_strategy=all_exhausted` to execute an oversampling strategy. In this case, the dataset construction is stopped as soon as every samples in every dataset has been added at least once. In practice, it means that if a dataset is exhausted, it will return to the beginning of this dataset until the stop criterion has been reached.
Note that if no sampling probabilities are specified, the new dataset will have `max_length_datasets*nb_dataset samples`.
There is also `stopping_strategy=all_exhausted_without_replacement` to ensure that every sample is seen exactly once.

```
>>> 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]})
>>> dataset = interleave_datasets([d1, d2, d3], stopping_strategy="all_exhausted")
>>> dataset["a"]
[0, 10, 20, 1, 11, 21, 2, 12, 22, 0, 13, 20]
```

Format

The with_format() function changes the format of a column to be compatible with some common data formats. Specify the output you'd like in the `type` parameter. You can also choose which the columns you want to format using `columns=`. Formatting is applied on-the-fly.

For example, create PyTorch tensors by setting `type="torch"`:

```
>>> dataset = dataset.with_format(type="torch")
```

The set_format() function also changes the format of a column, except it runs in-place:

```
>>> dataset.set_format(type="torch")
```

If you need to reset the dataset to its original format, set the format to `None` (or use reset_format()):

```
>>> dataset.format
{'type': 'torch', 'format_kwargs': {}, 'columns': [...], 'output_all_columns': False}
>>> dataset = dataset.with_format(None)
>>> dataset.format
{'type': None, 'format_kwargs': {}, 'columns': [...], 'output_all_columns': False}
```

Tensors formats

Several tensors or arrays formats are supported. It is generally recommended to use these formats instead of converting outputs of a dataset to tensors or arrays manually to avoid unnecessary data copies and accelerate data loading.

Here is the list of supported tensors or arrays formats:

- NumPy: format name is "numpy", for more information see Using Datasets with NumPy
- PyTorch: format name is "torch", for more information see Using Datasets with PyTorch
- TensorFlow: format name is "tensorflow", for more information see Using Datasets with TensorFlow
- JAX: format name is "jax", for more information see Using Datasets with JAX

> [!TIP]
> Check out the Using Datasets with TensorFlow guide for more details on how to efficiently create a TensorFlow dataset.

When a dataset is formatted in a tensor or array format, all the data are formatted as tensors or arrays (except unsupported types like strings for example for PyTorch):

```
>>> ds = Dataset.from_dict({"text": ["foo", "bar"], "tokens": [[0, 1, 2], [3, 4, 5]]})
>>> ds = ds.with_format("torch")
>>> ds[0]
{'text': 'foo', 'tokens': tensor([0, 1, 2])}
>>> ds[:2]
{'text': ['foo', 'bar'],
 'tokens': tensor([[0, 1, 2],
         [3, 4, 5]])}
```

Tabular formats

You can use a dataframes or tables format to optimize data loading and data processing, since they generally offer zero-copy operations and transforms written in low-level languages.

Here is the list of supported dataframes or tables formats:

- Pandas: format name is "pandas", for more information see Using Datasets with Pandas
- Polars: format name is "polars", for more information see Using Datasets with Polars
- PyArrow: format name is "arrow", for more information see Using Datasets with PyArrow

When a dataset is formatted in a dataframe or table format, every dataset row or batches of rows is formatted as a dataframe or table, and dataset colums are formatted as a series or array:

```
>>> ds = Dataset.from_dict({"text": ["foo", "bar"], "label": [0, 1]})
>>> ds = ds.with_format("pandas")
>>> ds[:2]
  text  label
0  foo      0
1  bar      1
```

Those formats make it possible to iterate on the data faster by avoiding data copies, and also enable faster data processing in map() or filter():

```
>>> ds = ds.map(lambda df: df.assign(upper_text=df.text.str.upper()), batched=True)
>>> ds[:2]
  text  label upper_text
0  foo      0        FOO
1  bar      1        BAR
```

Custom format transform

The with_transform() function applies a custom formatting transform on-the-fly. This function replaces any previously specified format. For example, you can use this function to tokenize and pad tokens on-the-fly. Tokenization is only applied when examples are accessed:

```
>>> from transformers import AutoTokenizer

>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
>>> def encode(batch):
...     return tokenizer(batch["sentence1"], batch["sentence2"], padding="longest", truncation=Tru
>>> dataset = dataset.with_transform(encode)
>>> dataset.format
{'type': 'custom', 'format_kwargs': {'transform': <function __main__.encode(batch)>}, 'columns': [
```

There is also set_transform() which does the same but runs in-place.

You can also use the with_transform() function for custom decoding on Features.

The example below uses the `pydub` package as an alternative to `torchcodec` decoding:

```
>>> import numpy as np
>>> from pydub import AudioSegment

>>> audio_dataset_amr = Dataset.from_dict({"audio": ["audio_samples/audio.amr"]})

>>> def decode_audio_with_pydub(batch, sampling_rate=16_000):
...     def pydub_decode_file(audio_path):
...         sound = AudioSegment.from_file(audio_path)
...         if sound.frame_rate != sampling_rate:
...             sound = sound.set_frame_rate(sampling_rate)
...         channel_sounds = sound.split_to_mono()
...         samples = [s.get_array_of_samples() for s in channel_sounds]
...         fp_arr = np.array(samples).T.astype(np.float32)
...         fp_arr /= np.iinfo(samples[0].typecode).max
...         return fp_arr
...
...     batch["audio"] = [pydub_decode_file(audio_path) for audio_path in batch["audio"]]
...     return batch

>>> audio_dataset_amr.set_transform(decode_audio_with_pydub)
```

Save

Once your dataset is ready, you can save it as a Hugging Face Dataset in Parquet format and reuse it later with load_dataset().

Save your dataset by providing the name of the dataset repository on Hugging Face you wish to save it to to push_to_hub():

```
encoded_dataset.push_to_hub("username/my_dataset")
```

You can use multiple processes to upload it in parallel. This is especially useful if you want to speed up the process:

```
dataset.push_to_hub("username/my_dataset", num_proc=8)
```

Use the load_dataset() function to reload the dataset (in streaming mode or not):

```
from datasets import load_dataset
reloaded_dataset = load_dataset("username/my_dataset", streaming=True)
```

Alternatively, you can save it locally in Arrow format on disk. Compared to Parquet, Arrow is uncompressed which makes it much faster to reload which is great for local use on disk and ephemeral caching. But since it's larger and with less metadata, it is slower to upload/download/query than Parquet and less suited for long term storage.

Use the save_to_disk() and load_from_disk() function to reload the dataset from your disk:

```
>>> encoded_dataset.save_to_disk("path/of/my/dataset/directory")
>>> # later
>>> from datasets import load_from_disk
>>> reloaded_dataset = load_from_disk("path/of/my/dataset/directory")
```

Export

🤗 Datasets supports exporting as well so you can work with your dataset in other applications. The following table shows currently supported file formats you can export to:

File type	Export method
CSV	Dataset.to_csv()
JSON	Dataset.to_json()
Parquet	Dataset.to_parquet()
SQL	Dataset.to_sql()
In-memory Python object	Dataset.to_pandas(), `Dataset.to_polars()` or Dataset.to_dict()

For example, export your dataset to a CSV file like this:

```
>>> encoded_dataset.to_csv("path/of/my/dataset.csv")
```