Operator Insights

Data-Juicer Operators (OPs) provide a variety of basic capabilities of data processing, including modification, cleaning, filtering and deduplication.

For now, there are 5 types of OPs: Formatter, Mapper, Filter, Deduplicator, Selector.

Formatters are used to discover, load, and canonicalize source data to Data-Juicer DJDataset or its subclass NestedDataset, which support several input data formats, such as csv/tsv, json/jsonl, parquet, doc, pdf, and plain-text format (e.g. txt, tex, code).

The other 4 types of OPs are mainly for data processing.

In this notebook, we focus on these 4 types of OPs. And in the following sections, we will run several Operators to gain a deeper understanding of OPs, and inspect their results.

Mappers

A Mapper is primarily used for editing, modifying, enhancing, or synthesizing data sample functionality.

The basic class of Mapper is like:

```
class Mapper(OP):
    ...
    def process(self, sample: Dict) -> Dict:
        # process a single sample
        ...

    def run(self, dataset: DJDataset) -> DJDataset:
        return dataset.map(self.process, **args)
    ...
```

It needs a `process` method to process a single sample. In the unified `run` method, Mapper just processes each sample in the dataset with the `map` method of DJDataset.

For example, we can use `CleanIpMapper` to clean up IP addresses in text.

First, we import this OP from Data-Juicer.

In [ ]:
```
from data_juicer.ops.mapper.clean_ip_mapper import CleanIpMapper
op = CleanIpMapper()
```

Then, we can apply this OP in two ways.

- Invoke OP's process directly

In [2]:
```
sample = {'text': 'test of ip 11.22.33.44'}
out_sample = op.process(sample)

print(out_sample)
```

```
{'text': 'test of ip '}
```

- Invoke OP's process with NestedDataset

In [3]:
```
from data_juicer.core import NestedDataset

samples = [{'text': 'test of ip 11.22.33.44'}]
ds = NestedDataset.from_list(samples)
out_ds = ds.map(op.process)

for sample in out_ds:
    print(sample)
```

```
Map: 100%|██████████| 1/1 [00:00<00:00, 555.17 examples/s]
{'text': 'test of ip '}
```

You can also invoke `run` method directly on the dataset.

In [5]:
```
out_ds = op.run(ds)

for sample in out_ds:
    print(sample)
```

```
num_proc must be <= 1. Reducing num_proc to 1 for dataset of size 1.
clean_ip_mapper_process: 100%|██████████| 1/1 [00:00<00:00, 743.80 example
s/s]
{'text': 'test of ip '}
```

Filters

A Filter is mainly used to filter out unexpected samples, such as low-quality, NSFW samples, or filter according to some statistics (we call it "stats").

The basic class of Filter is like:

```
class Filter(OP):
    ...
    def compute_stats(self, sample: Dict) -> Dict:
        # compute stats of a single sample
        ...

    def process(self, sample: Dict) -> bool:
        # decide whether to keep this single sample
        ...

    def run(self, dataset: DJDataset) -> DJDataset:
        return dataset.map(self.compute_stats, **args)
                      .filter(self.process, **args)
    ...
```

It needs a `compute_stats` method to compute the statistics we are interested in, such as the number of words, image-text similarity, video motion scores, etc. The `process` method is also required. But different from Mappers, this method here is used to decide whether to keep a single sample according to their computed stats and predefined thresholds. As a result, the implementation of the `run` method of Filters is to compute stats for each sample in the dataset and then to filter out those unexpected samples.

Here we show you how to use `WordsNumFilter` to filter out samples whose number of words is not within the range of [3, 10], which means, to discard samples with less than 3 or more than 10 words.

In [6]:
```
from data_juicer.core import NestedDataset

samples = [
    {'text': 'Data Juicer'},
    {'text': 'Welcome to Data Juicer Playground'}
]
ds = NestedDataset.from_list(samples)

# Add a new column to the dataset to store the stats of the Filter operat
from data_juicer.utils.constant import Fields
ds = ds.add_column(name=Fields.stats, column=[{}] * ds.num_rows)

print(ds)
```

```
Dataset({
    features: ['text', '__dj__stats__'],
    num_rows: 2
})
```

Then we initialize this Filter.

In [7]:
```
from data_juicer.ops.filter.words_num_filter import WordsNumFilter
op = WordsNumFilter(
    min_num=3,
    max_num=10
)
```

Finally, we just need to compute stats for each sample and then

In [8]:
```
ds_stats = ds.map(op.compute_stats)

for sample in ds_stats:
    print(sample)

out_ds = ds_stats.filter(op.process)

print(out_ds)
print(f'Number of samples of output dataset : {len(out_ds)}')
for sample in out_ds:
    print(sample)
```

```
Map: 100%|██████████| 2/2 [00:00<00:00, 700.74 examples/s]
```

```
{'text': 'Data Juicer', '__dj__stats__': {'num_words': 2}}
{'text': 'Welcome to Data Juicer Playground', '__dj__stats__': {'num_word
s': 5}}
Filter: 100%|██████████| 2/2 [00:00<00:00, 1072.85 examples/s]
Dataset({
    features: ['text', '__dj__stats__'],
    num_rows: 1
})
Number of samples of output dataset : 1
{'text': 'Welcome to Data Juicer Playground', '__dj__stats__': {'num_word
s': 5}}
```

Same as Mappers, you can also invoke the `run` method directly.

In [9]:
```python
out_ds = op.run(ds)

print(out_ds)
print(f'Number of samples of output dataset : {len(out_ds)}')
for sample in out_ds:
    print(sample)
```

```
num_proc must be <= 2. Reducing num_proc to 2 for dataset of size 2.
words_num_filter_compute_stats (num_proc=2): 100%|██████████| 2/2 [00:00<0
0:00, 20.13 examples/s]
num_proc must be <= 2. Reducing num_proc to 2 for dataset of size 2.
words_num_filter_process (num_proc=2): 100%|██████████| 2/2 [00:00<00:00,
25.24 examples/s]
Dataset({
    features: ['text', '__dj__stats__'],
    num_rows: 1
})
Number of samples of output dataset : 1
{'text': 'Welcome to Data Juicer Playground', '__dj__stats__': {'num_word
s': 5}}
```

Deduplicators

A Deduplicator is mainly used to detect and remove duplicate or near-duplicate samples from datasets.

The basic class of Deduplicator is like:

```python
class Deduplicator(OP):
    ...
    def compute_hash(self, sample: Dict) -> Dict:
        # compute the hash of a single sample
        ...

    def process(self, dataset: DJDataset) -> DJDataset:
        # deduplicate for the dataset
        ...

    def run(self, dataset: DJDataset) -> DJDataset:
        dataset = dataset.map(self.compute_hash, **args)
        return self.process(dataset)
    ...
```

Similar to Filter, it needs to compute the hash values for each sample in `compute_hash` method and then apply the deduplication process for the whole dataset.

Here is a case-insensitive demo to deduplicate samples using exact matching (md5 hash) from `DocumentDeduplicator` OP.

In [10]:
```
from data_juicer.core import NestedDataset

samples = [
    {'text': 'welcome to data juicer playground'},
    {'text': 'Welcome to Data Juicer Playground'}
]
ds = NestedDataset.from_list(samples)

print(ds)
print(f'Number of samples of input dataset : {len(ds)}')
```

```
Dataset({
    features: ['text'],
    num_rows: 2
})
Number of samples of input dataset : 2
```

We initialize a Deduplicator first.

In [11]:
```
from data_juicer.ops.deduplicator.document_deduplicator import DocumentDe
op = DocumentDeduplicator(lowercase=True)
```

Then we run these two methods for the dataset.

In [12]:
```
ds_with_hash = ds.map(op.compute_hash)
out_ds, dup_pairs= op.process(ds_with_hash, show_num=1)

print(out_ds)
print(f'Number of samples of output dataset : {len(out_ds)}')
for sample in out_ds:
    print(sample)
```

```
Map: 100%|██████████| 2/2 [00:00<00:00, 1023.38 examples/s]
Filter: 100%|██████████| 2/2 [00:00<00:00, 1433.46 examples/s]
Dataset({
    features: ['text', '__dj__hash'],
    num_rows: 1
})
Number of samples of output dataset : 1
{'text': 'welcome to data juicer playground', '__dj__hash': 'f2fea8ead9b37
a9f9085037a8d97d497'}
```

Here `dup_pairs` is the duplicate pairs obtained from the Tracer tool, which will be introduced later. We can print it to find out the duplicate pairs to check if it works well.

In [13]:
```
for key, dup_pair in dup_pairs.items():
    print(f'Deduplicate hash value : {key}')
```

```
    for sample in dup_pair:
        print(sample)
```

```
Deduplicate hash value : f2fea8ead9b37a9f9085037a8d97d497
{'text': 'welcome to data juicer playground', '__dj__hash': 'f2fea8ead9b37
a9f9085037a8d97d497'}
{'text': 'Welcome to Data Juicer Playground', '__dj__hash': 'f2fea8ead9b37
a9f9085037a8d97d497'}
```

Selectors

A Selector is mainly used to select top samples based on ranking. It is primarily used to perform statistical analysis on a specific field of a dataset. For instance, it can select the top k samples with the highest frequency, or to choose a portion of samples with the highest proportion.

The basic class of Selector is like:

```
class Selector(OP):
    ...
    def process(self, sample: Dict) -> Dict:
        # select from the whole dataset
        ...

    def run(self, dataset: DJDataset) -> DJDataset:
        return self.process(dataset)
    ...
```

It's quite similar to Mapper but selects target samples from the whole dataset.

Here is an example. First we construct a dataset with 5 samples, and use Selector operator to select the top 2 samples based on `meta.count`. It's worth noticing that Data-Juicer support '.' operator to access the nested fields in the dataset.

In [14]:
```
from data_juicer.core import NestedDataset

samples = [{
            'text': 'Today is Sun',
            'meta': {'count': 5 }
        }, {
            'text': 'a v s e c s f e f g a a a  ',
            'meta': {'count': 23 }
        }, {
            'text': ', 。、„”“«»1」「《》´:：？！ ',
            'meta': { 'count': 48 }
        }, {
            'text': '他的英文名字叫Harry Potter',
            'meta': { 'count': 78 }
        }, {
            'text': '这是一个测试',
            'meta': { 'count': 3 }
        }]
ds = NestedDataset.from_list(samples)

print(ds)
print(f'Number of samples of input dataset : {len(ds)}')
```

```
Dataset({
    features: ['text', 'meta'],
    num_rows: 5
})
Number of samples of input dataset : 5
```

Initialize a `TopkSpecifiedFieldSelector` and specify the target key.

In [15]:
```
from data_juicer.ops.selector.topk_specified_field_selector import TopkSp
op = TopkSpecifiedFieldSelector(field_key='meta.count', topk=2)
```

Select top-k samples from the whole dataset.

In [16]:
```
out_ds = op.process(ds)

print(out_ds)
print(f'Number of samples of output dataset : {len(out_ds)}')
for sample in out_ds:
    print(sample)
```

```
Dataset({
    features: ['text', 'meta'],
    num_rows: 2
})
Number of samples of output dataset : 2
{'text': '他的英文名字叫Harry Potter', 'meta': {'count': 78}}
{'text': ', 。、„”“«»1」「《》´∶？！', 'meta': {'count': 48}}
```

Multimodel Demos

Data-Juicer now supplies a lot of operators to process **multimodal** data. Here we provide a few demos to show you how to process multimodal datasets.

Note

The input dataset must adhere to the Data-Juicer format, characterized by a text-centric, multi-chunk structure interspersed with special tokens.

Additionally, we offer a suite of multimodal tools designed to facilitate conversion between other formats and the Data-Juicer format. We will dive into the detail of this format and corresponding conversion tools in the next notebook.

We use the test dataset from Data-Juicer in `{data-juicer_root}/tests/ops/data` to show you how to construct your dataset and process multimodel data.

Image-Text Dataset

Generally speaking, most multimodal datasets consist of at least two modalities, such as image-text pairs.

We construct an image-text dataset first.

In [17]:
```
import os
from data_juicer.core import NestedDataset
from data_juicer.utils.mm_utils import SpecialTokens

data_juicer_root = '../'  # change this path to your data-juicer root

data_path = f'{data_juicer_root}/tests/ops/data'
cat_path = os.path.join(data_path, 'cat.jpg')

# we need a spacial token to indicate the position of multimodal data in
samples = [{
            'text': f'{SpecialTokens.image}a photo of a cat',  # 0.245700
            'images': [cat_path]
        }, {
            'text': f'{SpecialTokens.image}a photo of a dog',  # 0.193049
            'images': [cat_path]
        }]
ds = NestedDataset.from_list(samples)

# add a new column to the dataset to store the statistical values of the
from data_juicer.utils.constant import Fields
ds = ds.add_column(name=Fields.stats, column=[{}] * ds.num_rows)

print(ds)
print(f'Number of samples of input dataset : {len(ds)}')
```

```
Dataset({
    features: ['text', 'images', '__dj__stats__'],
    num_rows: 2
})
Number of samples of input dataset : 2
```

Here we use `ImageTextSimilarityFilter` to filter out the samples whose image-text pair is not aligned well.

During the initialization of `ImageTextSimilarityFilter` operator, we need to load the `openai/clip-vit-base-patch32` model. You can also replace this HuggingFace Hub model path with a local model path.

In [18]:
```
from data_juicer.ops.filter.image_text_similarity_filter import ImageText
op = ImageTextSimilarityFilter(
    hf_clip = 'openai/clip-vit-base-patch32',  # you can replace it with
    min_score=0.2,
    max_score=0.9
)
```

```
/usr/local/python310/lib/python3.10/site-packages/huggingface_hub/file_dow
nload.py:1132: FutureWarning: `resume_download` is deprecated and will be
removed in version 1.0.0. Downloads always resume when possible. If you wa
nt to force a new download, use `force_download=True`.
  warnings.warn(
/usr/local/python310/lib/python3.10/site-packages/torch/_utils.py:831: Use
rWarning: TypedStorage is deprecated. It will be removed in the future and
UntypedStorage will be the only storage class. This should only matter to
you if you are using storages directly.  To access UntypedStorage directl
y, use tensor.untyped_storage() instead of tensor.storage()
  return self.fget.__get__(instance, owner)()
```

In [19]:
```
ds_with_stats = ds.map(op.compute_stats)

for sample in ds_with_stats:
    print(sample)

out_ds = ds_with_stats.filter(op.process)

print(out_ds)
print(f'Number of samples of output dataset : {len(out_ds)}')
for sample in out_ds:
    print(sample)
```

```
Map: 100%|██████████| 2/2 [00:00<00:00,  4.26 examples/s]
{'text': '<__dj__image>a photo of a cat', 'images': ['../data-juicer/test
s/ops/data/cat.jpg'], '__dj__stats__': {'image_text_similarity': [0.245696
91717624664]}}
{'text': '<__dj__image>a photo of a dog', 'images': ['../data-juicer/test
s/ops/data/cat.jpg'], '__dj__stats__': {'image_text_similarity': [0.193045
43733596802]}}
Filter: 100%|██████████| 2/2 [00:00<00:00, 1240.00 examples/s]
Dataset({
    features: ['text', 'images', '__dj__stats__'],
    num_rows: 1
})
Number of samples of output dataset : 1
{'text': '<__dj__image>a photo of a cat', 'images': ['../data-juicer/test
s/ops/data/cat.jpg'], '__dj__stats__': {'image_text_similarity': [0.245696
91717624664]}}
```

Image-Only Dataset

Sometimes we may want to process datasets based on a single modality or work with single-modal datasets.

Here we take `ImageShapeFilter` as example to process image-only dataset.

In [20]:
```
import os
from datasets import Dataset

data_juicer_root = '../'  # change this path to your data-juicer root

data_path = f'{data_juicer_root}/tests/ops/data'
img1_path = os.path.join(data_path, 'img1.png') # 336*336
img2_path = os.path.join(data_path, 'img2.jpg') # 640*480
img3_path = os.path.join(data_path, 'img3.jpg') # 342*500

samples = [{
            'images': [img1_path]
        }, {
            'images': [img2_path]
        }, {
            'images': [img3_path]
        }]
ds = Dataset.from_list(samples)

# add a new column to the dataset to store the statistical values of the
from data_juicer.utils.constant import Fields
```

```
ds = ds.add_column(name=Fields.stats, column=[{}] * ds.num_rows)

print(ds)
print(f'Number of samples of input dataset : {len(ds)}')
```

```
Dataset({
    features: ['images', '__dj__stats__'],
    num_rows: 3
})
Number of samples of input dataset : 3
```

In [21]:
```
from data_juicer.ops.filter.image_shape_filter import ImageShapeFilter
op = ImageShapeFilter(
    min_width=400,
    min_height=400
)
```

In [22]:
```
ds = ds.map(op.compute_stats)

for sample in ds:
    print(sample)

out_ds = ds.filter(op.process)

print(out_ds)
print(f'Number of samples of output dataset : {len(out_ds)}')
for sample in out_ds:
    print(sample)
```

```
Map: 100%|██████████| 3/3 [00:00<00:00, 352.96 examples/s]
{'images': ['../data-juicer/tests/ops/data/img1.png'], '__dj__stats__':
{'image_height': [336], 'image_width': [336]}}
{'images': ['../data-juicer/tests/ops/data/img2.jpg'], '__dj__stats__':
{'image_height': [480], 'image_width': [640]}}
{'images': ['../data-juicer/tests/ops/data/img3.jpg'], '__dj__stats__':
{'image_height': [500], 'image_width': [342]}}
Filter: 100%|██████████| 3/3 [00:00<00:00, 1489.28 examples/s]
Dataset({
    features: ['images', '__dj__stats__'],
    num_rows: 1
})
Number of samples of output dataset : 1
{'images': ['../data-juicer/tests/ops/data/img2.jpg'], '__dj__stats__':
{'image_height': [480], 'image_width': [640]}}
```

Conclusion

In this notebook, we take a look at the basic 5 types of OPs in Data-Juicer. For each type of OP, we take an OP example to show how to process dataset with them. Finally, we provide two demos to show the processing capabilities on multimodal datasets.