daskperiment Documentation

sinhrks

Apr 24, 2019

Contents

daskperiment is a tool to perform reproducible machine learning experiment. It enables users to define and manage the history of trials (given parameters, results and execution environment).

This package is built on *Dask*, a package for parallel computing with task scheduling. Each experiment trial is internally expressed as *Dask* computation graph, and can be executed in parallel.

It can be used both on Jupyter and command line (and also on standard Python interpreter). The benefits of daskperiemnt are:

- Compatibility with standard Python/Jupyter environment (and optionally with standard KVS).
  - No need to set up server applications
  - No need to registrate on any cloud services
  - Run on standard / customized Python shells
- Intuitive user interface
  - Few modifications on existing codes are needed
  - Trial histories are logged automatically (no need to write additional codes for logging)
  - *Dask* compatible API
  - Easily accessible experiments history (with *pandas* basic operations)
  - Less managiment works on Git (no need to make branch per trials)
  - (Experimental) Web dashboard to manage trial history
- Traceability of experiment related information
  - Trial result and its (hyper) parameters.
  - Code contexts
  - Environment information
    - Device information
    - OS information
    - Python version
    - Installed Python packages and its version
    - Git information
- Reproducibility
  - Check function purity (each step should return the same output for the same inputs)
  - Automatic random seeding
- Auto saving and loading of previous experiment history
- Parallel execution of experiment steps
- Experiment sharing
  - Redis backend
  - MongoDB backend

Contents:

CHAPTER 1

What's new

1.1 v0.5.0

1.1.1 Enhancement

- MongoDB backend support
- Redesign web dashboard

1.1.2 Bug fix

- Trial result may be incorrect if it is ran from multiple threads

1.2 v0.4.0

1.2.1 Enhancement

- Added *verbose* option to *Experiment.get_history* (default *False*)
- Environment now collects following info:
  - Detailed CPU info with *py-cpuinfo*
  - *conda info*
  - *numpy.show_config()*
  - *scipy.show_config()*
  - *pandas.show_versions()*
- Dashboard now supports to show summary, code and environment info.

1.2.2 Bug fix

- Log output of "number of installed python packages" is incorrect
- Dashboard can't switch display metric name

1.3 v0.3.0

1.3.1 Enhancement

- Function purity check
- Random seed handling
- Parameter now supports default value
- Capture Git info
- (Experimental) Minimal dashboard

1.3.2 Bug fix

- *Experiment.get_history* with no parameters results in empty *DataFrame*.
- Unable to change log level and its handler one time.

1.4 v0.2.0

1.4.1 Enhancement

- Redis backend support
- "Result Type" column is added to a *DataFrame* which *Experiment.get_history* returns (as *pandas* may change column dtype)

1.5 v0.1.1

1.5.1 Bug fix

- *str* input for experiment function may be incorrectly regarded as a parameter if the same parameter exists.
- Experiment.get_histoy raises *TypeError* in *pandas* 0.22 or earlier
- Experiment may raise *AttributeError* depending on *pip* version

1.6 v0.1.0

- Initial release

CHAPTER 2

Quickstart

This section describes a minimal example. First, create *daskperiment.Experiment* instance. This instance controlls an experiment, a chain of functions to output value and a collection of input variables.

Note: Unrelated logs are omitted in following examples.

```
>>> import numpy as np
>>> import daskperiment

>>> ex = daskperiment.Experiment(id='quickstart_pj')
...
>>> ex
Experiment(id: quickstart_pj, trial_id: 0, backend: LocalBackend('daskperiment_cache/
↪quickstart_pj'))
```

Then, use *Experiment.parameter* method to define parameters (input variables for the experiment). The actual value of each parameter can be changed in every trial.

```
>>> a = ex.parameter('a')
>>> b = ex.parameter('b')
>>> a
Parameter(a: Undefined)
```

Next, define each experiment step (function) by decorating with *Experiment* instance (*@ex*).

Note that the function to output the final result (mostly objective value to be minimized or maximized) must be decorated with *Experiment.result*. The chain of these functions are expressed as *Dask.Delayed* instance.

```
>>> @ex
>>> def prepare_data(a, b):
>>>     return a + b

>>> @ex.result
```

(continues on next page)

(continued from previous page)

```
>>> def calculate_score(s):
>>>     return 10 / s

>>> d = prepare_data(a, b)
>>> s = calculate_score(d)
>>> s
Delayed('calculate_score-ebe2d261-8903-45e1-b224-72b4c886e4c5')
```

Thus, you can visualize computation graph via *.visualize* method.

```
>>> s.visualize()
```

Use *Experiment.set_parameters* method to set parameters for a trial. After setting parameters, *Parameter* variable and experiment result can be computable.

Parameters are recommended to be a scalar (or lightweight value) because these are stored as history (for example, passing filename as a parameter is preffered rather than passing *DataFrame*).

```
>>> ex.set_parameters(a=1, b=2)
...
```

```
>>> s.compute()
... [INFO] Started Experiment (trial id=1)
...
... [INFO] Finished Experiment (trial id=1)
...
3.3333333333333335
```

You can update any parameters for next trial. Every trials can be distinguished by trial id.

```
>>> ex.set_parameters(b=3)
>>> s.compute()
...
... [INFO] Started Experiment (trial id=2)
...
... [INFO] Finished Experiment (trial id=2)
...
2.5
```

After some trials, you can retrieve parameter values specifying trial id.

```
>>> ex.get_parameters(trial_id=1)
{'a': 1, 'b': 2}

>>> ex.get_parameters(trial_id=2)
{'a': 1, 'b': 3}
```

Experiment.get_history returns a *DataFrame* which stores the history of trial parameters and its results. You can select desirable trial using *pandas* basic operation.

```
>>> ex.get_history()
   a  b    Result      Result Type  Success                   Finished  \
1  1  2  3.333333  <class 'float'>     True 2019-02-03 XX:XX:XX.XXXXXX
2  1  3  2.500000  <class 'float'>     True 2019-02-03 XX:XX:XX.XXXXXX

     Process Time  Description
```

(continues on next page)

(continued from previous page)

```
1 00:00:00.014183          NaN
2 00:00:00.012354          NaN
```

When any error occurs during the trial, Experiment instance stores the log as failed trial. The "Description" column contains the error detail.

```
>>> ex.set_parameters(a=1, b=-1)
>>> s.compute()
...
ZeroDivisionError: division by zero

>>> ex.get_history()
   a  b    Result      Result Type  Success                   Finished  \
1  1  2  3.333333  <class 'float'>     True 2019-02-03 XX:XX:XX.XXXXXX
2  1  3  2.500000  <class 'float'>     True 2019-02-03 XX:XX:XX.XXXXXX
3  1 -1       NaN             None    False 2019-02-03 XX:XX:XX.XXXXXX

     Process Time                          Description
1 00:00:00.014183                                  NaN
2 00:00:00.012354                                  NaN
3 00:00:00.015954  ZeroDivisionError(division by zero)
```

2.1 Handling Intermediate Result

Next example shows how to retrieve an intermediate result of the chain.

The only difference is using *Experiment.persist* decorator. It makes *Experiment* instance to keep the decorated function's intermediate result. After definition, rebuilt the same workflow using the persisted function.

Note that an intermediate result is saved as a pickle file named with its function name which must be unique in the experiment.

```
>>> @ex.persist
>>> def prepare_data(a, b):
>>>     return a + b

>>> d = prepare_data(a, b)
>>> s = calculate_score(d)
... [WARNING] Code context has been changed: prepare_data
... [WARNING] @@ -1,3 +1,3 @@
... [WARNING] -@ex
... [WARNING] +@ex.persist
... [WARNING]  def prepare_data(a, b):
... [WARNING]      return a + b

...
```

Note: If you execute the code above, *daskperiment* outputs some "WARNING" indicating code contexts has been changed. It's because *daskperiment* automatically tracks code context to guarantee reproducibility.

Let's perform some trials.

```
>>> ex.set_parameters(a=1, b=2)
>>> s.compute()
...
... [INFO] Finished Experiment (trial id=4)
...
3.3333333333333335

>>> ex.set_parameters(a=3, b=2)
>>> s.compute()
...
... [INFO] Finished Experiment (trial id=5)
...
2.0
```

You can retrieve intermediate results via *Experiment.get_persisted* method by specifying function name and trial id.

```
>>> ex.get_persisted('prepare_data', trial_id=4)
...
3

>>> ex.get_persisted('prepare_data', trial_id=5)
...
5
```

2.2 Monitoring Metrics

You may need to monitor transition of some metrics during each trial. In each experiment function, you can call *Experiment.save_metric* to save metric with its key (name) and epoch.

```
>>> @ex.result
>>> def calculate_score(s):
>>>     for i in range(100):
>>>         ex.save_metric('dummy_score', epoch=i, value=100 - np.random.random() * i)
>>>     return 10 / s

>>> d = prepare_data(a, b)
>>> s = calculate_score(d)
...

>>> ex.set_parameters(a=1, b=2)
>>> s.compute()
...
... [INFO] Finished Experiment (trial id=6)
...
3.3333333333333335
```

After a trial, you can load saved metric using *Experiment.load_metric* specifying its name and trial_id. As it returns metrics as a *DataFrame*, you can easily investigate them.

```
>>> dummy_score = ex.load_metric('dummy_score', trial_id=6)
>>> dummy_score.head()
Trial ID           6
Epoch
0         100.000000
```

(continues on next page)

(continued from previous page)

```
1           99.925724
2           99.616405
3           98.527259
4           97.086730
```

Perform another trial.

```
>>> ex.set_parameters(a=3, b=4)
>>> s.compute()
...
... [INFO] Finished Experiment (trial id=7)
...
1.4285714285714286
```

To compare metrics between trials, pass multiple trial ids to *Experiment.load_metric*.

```
>>> ex.load_metric('dummy_score', trial_id=[6, 7]).head()
Trial ID           6           7
Epoch
0          100.000000  100.000000
1           99.925724   99.497605
2           99.616405   99.459706
3           98.527259   98.027079
4           97.086730   99.517617
```

2.3 Check Code Context

During the trials, *daskperiment* tracks code contexts decorated with *Experiment* decorators.

To check the tracked code contexts, use *Experiment.get_code* specifying trial id (if is not specified, it returns current code).

```
>>> ex.get_code()
@ex.persist
def prepare_data(a, b):
    return a + b


@ex.result
def calculate_score(s):
    for i in range(100):
        ex.save_metric('dummy_score', epoch=i, value=100 - np.random.random() * i)

    return 10 / s

>>> ex.get_code(trial_id=1)
@ex
def prepare_data(a, b):
    return a + b


@ex.result
def calculate_score(s):
    return 10 / s
```

Each code context is also saved as a text file per trial id. So it is easy to handle by diff tools and Git.

2.4 Function Purity And Handling Randomness

To make the experiment reproducible, all the experiment step should be "pure" function (it always returns the same outputs if the inputs to it are the same). In other words, the function should not have internal state nor randomness.

daskperiment checks whether each experiment step is pure. It internally stores the hash of inputs and output, and issues a warning if its output is changed even though the inputs are unchanged.

To illustrate this, add randomness to the example code.

```
>>> @ex.result
>>> def calculate_score(s):
>>>     for i in range(100):
>>>         ex.save_metric('dummy_score', epoch=i, value=100 - np.random.random() * i)

>>>     return 10 / s + np.random.random()

>>> d = prepare_data(a, b)
>>> s = calculate_score(d)
```

Because of the code change, it outputs the different results even though its inputs (parameters) are the same. In this case, *daskperiment* issuess the warning.

```
>>> s.compute()
...
... [INFO] Random seed is not provided, initialized with generated seed: 1336143935
...
... [WARNING] Experiment step result is changed with the same input: (step: calculate_
↪score, args: (7,), kwargs: {})
... [INFO] Finished Experiment (trial id=8)
2.1481070929378823
```

This function outputs different result in every trial because of the randomness. To make the function reproducible, random seed should be provided.

To do this, pass *seed* argument to *compute* method. Note that this trial issue the warning because its result is different to the previous result (no seed).

```
>>> s.compute(seed=1)
...
... [INFO] Random seed is initialized with given seed: 1
...
... [WARNING] Experiment step result is changed with the same input: (step: calculate_
↪score, args: (7,), kwargs: {})
... [INFO] Finished Experiment (trial id=9)
1.7552163303435249
```

Another trial with the same seed doesn't issue the warning, because the result is unchanged.

```
>>> s.compute(seed=1)
...
... [INFO] Random seed is initialized with given seed: 1
...
... [INFO] Finished Experiment (trial id=9)
1.7552163303435249
```

2.5 Save Experiment Status

daskperiment automatically saves its internal state when a experiment result is computed (when *.compute* is called). Also, *Experiment* instance recovers previous state when it is instanciated.

Following example instanciates *Experiment* instance using the same id as above. Thus, the created *Experiment* recovers its previous trial history.

```
>>> ex_new = daskperiment.Experiment(id='quickstart_pj')
```

Calling *.get_history* returns information of previous trials.

```
>>> ex_new.get_history()
...
```

Also, *Experiment* instance automatically detects the environment change from its previous trial. Following is a sample log when package update is detected (*pandas* 0.23.4 -> 0.24.0).

```
... [INFO] Loaded Experiment(id: quickstart_pj, trial_id: 14) from path=daskperiment_
↪cache/quickstart_pj/quickstart_pj.pkl
... [WARNING] Installed Python packages have been changed
... [WARNING] @@ -142 +142 @@
... [WARNING] -pandas 0.23.4 (/Users/sinhrks/anaconda/lib/python3.6/site-packages)
... [WARNING] +pandas 0.24.0 (/Users/sinhrks/anaconda/lib/python3.6/site-packages)
```

CHAPTER 3

Dashboard

daskperiment supports a web dashboard to check experiment histories.

3.1 Launch From Script

To launch the dashboard from script, use *Experiment.start_dashboard*. It should be non-blocking when called from interactive shell like Jupyter, and be blocking when executed as a file.

```
>>> ex = daskperiment.Experiment('your_experiment_id')
>>> ex.start_dashboard()
```

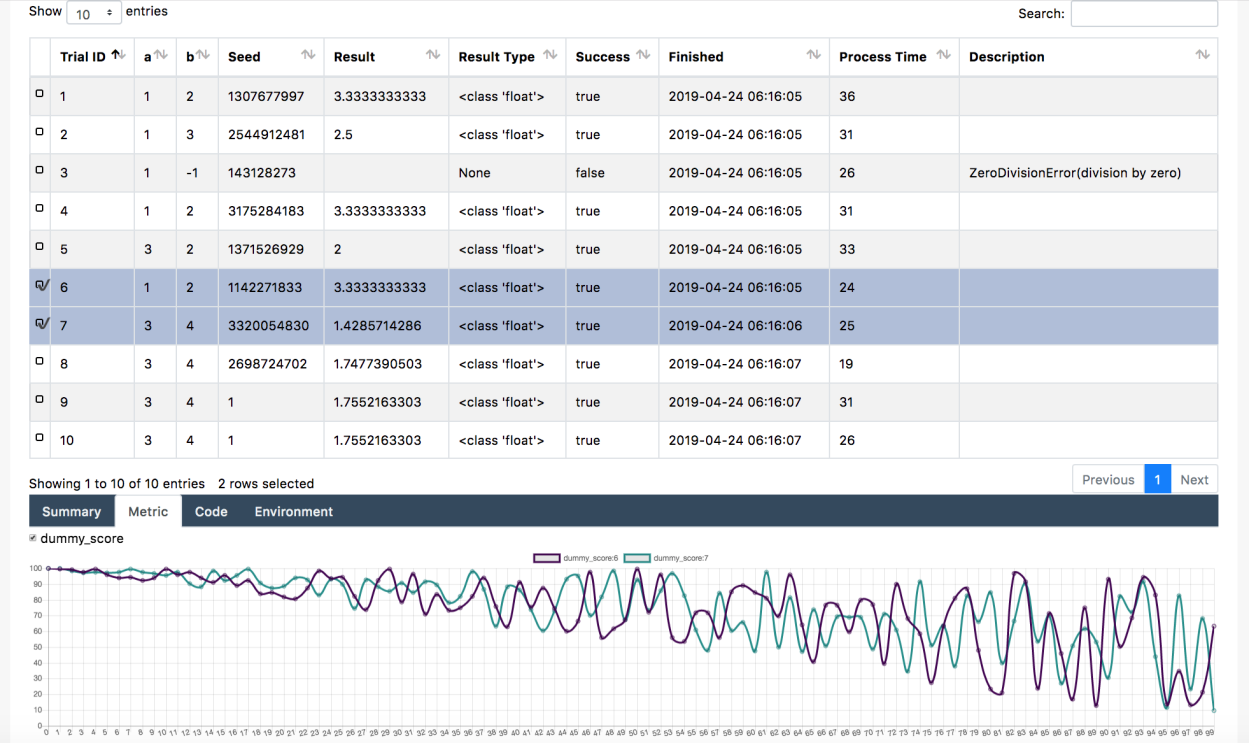
3.2 Launch From Terminal

To launch from the terminal, use *daskperimentboard* command providing your experiment id.

```
daskperimentboard your_experiment_id
```

3.3 Access To The Dashboard

After launching the dashboard, open http://localhost:5000/ .


Show 10 entries
Search:
Trial ID | a | b | Seed | Result | Result Type | Success | Finished | Process Time | Description
1 | 1 | 2 | 1307677997 | 3.3333333333 | <class 'float'> | true | 2019-04-24 06:16:05 | 36 |
2 | 1 | 3 | 2544912481 | 2.5 | <class 'float'> | true | 2019-04-24 06:16:05 | 31 |
3 | 1 | -1 | 143128273 | | None | false | 2019-04-24 06:16:05 | 26 | ZeroDivisionError(division by zero)
4 | 1 | 2 | 3175284183 | 3.3333333333 | <class 'float'> | true | 2019-04-24 06:16:05 | 31 |
5 | 3 | 2 | 1371526929 | 2 | <class 'float'> | true | 2019-04-24 06:16:05 | 33 |
6 | 1 | 2 | 1142271833 | 3.3333333333 | <class 'float'> | true | 2019-04-24 06:16:05 | 24 |
7 | 3 | 4 | 3320054830 | 1.4285714286 | <class 'float'> | true | 2019-04-24 06:16:06 | 25 |
8 | 3 | 4 | 2698724702 | 1.7477390503 | <class 'float'> | true | 2019-04-24 06:16:07 | 19 |
9 | 3 | 4 | 1 | 1.7552163303 | <class 'float'> | true | 2019-04-24 06:16:07 | 31 |
10 | 3 | 4 | 1 | 1.7552163303 | <class 'float'> | true | 2019-04-24 06:16:07 | 26 |
Showing 1 to 10 of 10 entries 2 rows selected
Previous 1 Next
Summary Metric Code Environment
dummy_score
dummy_score:6
dummy_score:7

CHAPTER 4

Command Line Interface

daskperiment also supports execution from command line.

First, prepare a *Python* script to define experiment. The usage of *Experiment* class is all the same as Jupyter example. *daskperiment* regards the result of a function decorated with *Experiment.result* (*calculate_score* function in below case) as experiment output.

The below is a prepared script example named "simple_experiment.py".

```
import daskperiment

ex = daskperiment.Experiment(id='simple_experiment_pj')

a = ex.parameter('a')
b = ex.parameter('b')

@ex
def prepare_data(a, b):
    return a + b

@ex.result
def calculate_score(s):
    return s + 1

d = prepare_data(a, b)
calculate_score(d)
```

You can provide parameter values from command line options using *key*=*value* format. *daskperinemt* automatically parse parameters and perform computation.

```
python simple_experiment.py a=1 b=2

... [INFO] Initialized new experiment: Experiment(id: simple_experiment_pj, trial_id:␣
↪0, backend: LocalBackend('daskperiment_cache/simple_experiment_pj'))
...
```

(continues on next page)

(continued from previous page)

```
... [INFO] Finished Experiment (trial id=1)
...
```

Let's perform another trial using different parameters. daskperiment automatically saves trial history as the same as Jupyter example (see trial id is incremented).

```
python ../scripts/simple_experiment.py a=3 b=2

... [INFO] Loading Experiment from file: daskperiment_cache/simple_experiment_pj/
↪simple_experiment_pj.pkl
...
... [INFO] Finished Experiment (trial id=2)
...
```

To confirm the experiment results, instanciate *Experiment* specifying the id of the script and use *Experiment.get_history*.

```
>>> import daskperiment

>>> ex = daskperiment.Experiment(id='simple_experiment_pj')
>>> ex.get_history()
   a  b  Result    Result Type  Success                  Finished  \
1  1  2       4  <class 'int'>     True 2019-02-03 XX:XX:XX.XXXXXX
2  3  2       6  <class 'int'>     True 2019-02-03 XX:XX:XX.XXXXXX

     Process Time  Description
1 00:00:00.009560          NaN
2 00:00:00.006512          NaN
```

4.1 Command Line Options

Use *–seed* option to provide random seed from terminal,

```
python random_experiment.py --seed 1
```

CHAPTER 5

Tips

5.1 Monitor Trial Related Information

You may want to log trial related information such as user who performs the trial, etc. The easiest way is to include these information in parameters.

```
>>> ex.parameter('user_name')
>>> ex.set_parameters(..., user_name='my_name')
```

5.2 Track Data Version

Most of experiments relies on external data source such as CSV, relational DB, etc. Sometimes experiment result is unexpectedly changed because of external data changes. In such cases, even though *daskperiment* checks function purity and issues a warning, users may not determine the exact reason.

It is recommended that data loading function is defined as a separate step, because if you receive a warning from that loading function, you can understand that the external data has changed.

```
>>> @ex
>>> def load_user_data(filename):
>>>     return pd.read_csv(filename)
```

5.3 Check Trial ID During Execution

You may want to know the current trial id during the trial. Then use *Experiment.current_trial_id* to check current trial id.

```
>>> @ex
>>> def experiment_step(a, b):
>>>     print('debug_information', ex.current_trial_id)
>>>     ...
```

You cannnot refer to *current_trial_id* outside of the experiment step because the id is generated when the trial is performed.

```
>>> ex.current_trial_id
daskperiment.core.errors.TrialIDNotFoundError: Current Trial ID only exists during a␣
↪trial execution
```

To check the last trial id of the experiment outside of the experiment step, use *.trial_id* property.

```
>>> ex.trial_id
3
```

The next trial should be numbered as *.trial_id + 1*, if no other trial is triggered until your execution. Note that *.trial_id* cannot be referred during a trial execution to avoid confusion between *.trial_id* and *.current_trial_id*.

CHAPTER 6

Backend

daskperiment uses *Backend* classes to define how and where experiment results are saved. Currently, following backends are supported.

- *LocalBackend*: Information is stored in local files. This is for personal usage with single PC. Use this backend when you don't need to share information with others or to move file(s) to another PC.
- *RedisBackend*: Information is stored in Redis and can be shared in a small team or among several PCs.
- *MongoBackend*: Information is stored in MongoDB and can be shared in a team or among PCs.

You can specify required *Backend* via *backend* keyword in *Experiment* instanciation.

6.1 LocalBackend

LocalBackend saves information as local files. When you create *Experiment* instance without *backend* argument, the *Experiment* uses *LocalBackend* to save its information as default.

Note: Unrelated logs are omited in following examples.

```
>>> import daskperiment
>>> daskperiment.Experiment('local_default_backend')
... [INFO] Creating new cache directory: /Users/sinhrks/Git/daskperiment/daskperiment_
↪cache/local_default_backend
... [INFO] Initialized new experiment: Experiment(id: local_default_backend, trial_
↪id: 0, backend: LocalBackend('daskperiment_cache/local_default_backend'))
...
Experiment(id: local_default_backend, trial_id: 0, backend: LocalBackend(
↪'daskperiment_cache/local_default_backend'))
```

To change the directory location, specify the path as *pathlib.Path* instance to *backend* argument.

```
>>> import pathlib
>>> daskperiment.Experiment('local_custom_backend', backend=pathlib.Path('my_dir'))
... [INFO] Creating new cache directory: /Users/sinhrks/Git/daskperiment/my_dir
... [INFO] Initialized new experiment: Experiment(id: local_custom_backend, trial_id:␣
↪0, backend: LocalBackend('my_dir'))
...
Experiment(id: local_custom_backend, trial_id: 0, backend: LocalBackend('my_dir'))
```

The following table shows information and saved location under specified cache directory.

Information	Format	Path
Experiment status (internal state)	Pickle	<experiment id>.pkl
Experiment history	Pickle	<experiment id>.pkl
Persisted results	Pickle	persist/<experiment id>_<function name>_<trial id>.pkl
Metrics	Pickle	<experiment id>.pkl
Function input & output hash	Pickle	<experiment id>.pkl
Code contexts	Text	code/<experiment id>_<trial id>.py
Platform information	Text(JSON)	environmemt/device_<experiment id>_<trial id>.json
CPU information	Text(JSON)	environmemt/cpu_<experiment id>_<trial id>.json
Python information	Text(JSON)	environmemt/python_<experiment id>_<trial id>.json
NumPy information (*numpy.show_config*)	Text	environmemt/numpy_<experiment id>_<trial id>.txt
SciPy information (*scipy.show_config*)	Text	environmemt/scipy_<experiment id>_<trial id>.txt
pandas information (*pandas.show_versions*)	Text	environmemt/pandas_<experiment id>_<trial id>.txt
conda information (*conda info*)	Text	environmemt/conda_<experiment id>_<trial id>.txt
Git information	Text(JSON)	environmemt/git_<experiment id>_<trial id>.json
Python package information	Text	environmemt/requirements_<experiment id>_<trial id>.txt

6.2 RedisBackend

RedisBackend saves information using *Redis*. To use *RedisBackend*, one of the simple ways is specifying Redis URI as *backend* argument.

```
>>> daskperiment.Experiment('redis_uri_backend', backend='redis://localhost:6379/0')
... [INFO] Initialized new experiment: Experiment(id: redis_uri_backend, trial_id: 0,␣
↪backend: RedisBackend('redis://localhost:6379/0'))
...
Experiment(id: redis_uri_backend, trial_id: 0, backend: RedisBackend('redis://
↪localhost:6379/0'))
```

Or, you can use *redis.ConnectionPool*.

```
>>> import redis
>>> pool = redis.ConnectionPool.from_uri('redis://localhost:6379/0')
>>> daskperiment.Experiment('redis_pool_backend', backend=pool)
... [INFO] Initialized new experiment: Experiment(id: redis_pool_backend, trial_id: 0,
↪ backend: RedisBackend('redis://localhost:6379/0'))
...
Experiment(id: redis_pool_backend, trial_id: 0, backend: RedisBackend('redis://
↪localhost:6379/0'))
```

(continues on next page)

(continued from previous page)

The following table shows information and saved location under specified Redis database.

Information	Format	Key
Experiment status (internal state)	Text	<experiment id>:trial_id
Experiment history (parameters)	Pickle	<experiment id>:parameter:<trial id>
Experiment history (results)	Pickle	<experiment id>:history:<trial id>
Persisted results	Pickle	<experiment id>:persist:<function name>:<trial id>
Metrics	Pickle	<experiment id>:metric:<metric name>:<trial id>
Function input & output hash	Text	<experiment id>:step_hash:<function name>-<input hash>
Code contexts	Text	<experiment id>:code:<trial id>
Platform information	Text(JSON)	<experiment id>:device:<trial id>
CPU information	Text(JSON)	<experiment id>:cpu:<trial id>
Python information	Text(JSON)	<experiment id>:python:<trial id>
NumPy information (*numpy.show_config*)	Text	<experiment id>:numpy:<trial id>
SciPy information (*scipy.show_config*)	Text	<experiment id>:scipy:<trial id>
pandas information (*pandas.show_versions*)	Text	<experiment id>:pandas:<trial id>
conda information (*conda info*)	Text	<experiment id>:conda:<trial id>
Git information	Text(JSON)	<experiment id>:git:<trial id>
Python package information	Text	<experiment id>:requirements:<trial id>

6.3 MongoBackend

MongoBackend saves information using MongoDB. To use *MongoBackend*, one of the simple ways is specifying MongoDB URI as *backend* argument.

```
>>> daskperiment.Experiment('mongo_uri_backend', backend='mongodb://localhost:27017/
↪test_db')
... [INFO] Initialized new experiment: Experiment(id: redis_uri_backend, trial_id: 0,␣
↪backend: MongoBackend('mongodb://localhost:27017/test_db'))
...
Experiment(id: mongo_uri_backend, trial_id: 0, backend: MongoBackend('mongodb://
↪localhost:27017/test_db'))
```

Or, you can use *pymogo.database.Database*. Note that you cannot pass *MongoClient* as backend because it doesn't specify the backend database.

```
>>> import pymongo
>>> client = pymongo.MongoClient('mongodb://localhost:27017/')
>>> db = client.test_db
>>> db
Database(MongoClient(host=['localhost:27017'], document_class=dict, tz_aware=False,␣
↪connect=True), 'test_db')
>>> daskperiment.Experiment('mongo_db_backend', backend=db)
... [INFO] Initialized new experiment: Experiment(id: mongo_db_backend, trial_id: 0,␣
↪backend: MongoBackend('mongodb://localhost:27017/test_db'))
...
Experiment(id: mongo_db_backend, trial_id: 0, backend: MongoBackend('mongodb://
↪localhost:27017/test_db'))
```

(continues on next page)

(continued from previous page)

The *MongoBackend* creates a document collection named experiment id under the database specified in *MongoBackend*. The following table shows document created under the collection.

Information	Format	Document
Experiment status (internal state)	Text	*{'experiment_id': <experiment id>, 'category': 'trial_id'}*
Experiment history (parameters)	Pickle	*{'experiment_id': <experiment id>, 'category': 'trial', 'trial_id': <trial id>, 'parameter'=<parameters>, 'history'=<history>}*
Experiment history (results)	Pickle	(same document as parameters)
Persisted results	Pickle	*{'experiment_id': <experiment id>, 'category': 'persist', 'step': <function name>, 'trial_id': <trial id>}*
Metrics	Pickle	*{'experiment_id': <experiment id>, 'category': 'metric', 'metric_key': <metric name>, 'trial_id': <trial id>}*
Function input & output hash	Text	*{'experiment_id': <experiment id>, 'category': 'step_hash', 'input_hash': <function name>-<input hash>}*
Code contexts	Text	*{'experiment_id': <experiment id>, 'category': 'code', 'trial_id': <trial id>}*
Platform information	Text(JSON)	*{'experiment_id': <experiment id>, 'category': 'environment', 'trial_id': <trial id>, 'device': <device>, . . . }*
CPU information	Text(JSON)	(same document as device)
Python information	Text(JSON)	(same document as device)
NumPy information (*numpy.show_config*)	Text	(same document as device)
SciPy information (*scipy.show_config*)	Text	(same document as device)
pandas information (*pandas.show_versions*)	Text	(same document as device)
conda information (*conda info*)	Text	(same document as device)
Git information	Text(JSON)	(same document as device)
Python package information	Text	(same document as device)