
QLib Documentation

Release 0.8.4.99

Microsoft

Apr 24, 2022

Contents

1	Document Structure	3
1.1	Qlib: Quantitative Platform	3
1.2	Quick Start	4
1.3	Installation	5
1.4	Qlib Initialization	6
1.5	Data Retrieval	8
1.6	Custom Model Integration	11
1.7	Workflow: Workflow Management	14
1.8	Data Layer: Data Framework & Usage	19
1.9	Forecast Model: Model Training & Prediction	34
1.10	Portfolio Strategy: Portfolio Management	37
1.11	Design of Nested Decision Execution Framework for High-Frequency Trading	43
1.12	Meta Controller: Meta-Task & Meta-Dataset & Meta-Model	44
1.13	Qlib Recorder: Experiment Management	47
1.14	Analysis: Evaluation & Results Analysis	62
1.15	Online Serving	75
1.16	Building Formulaic Alphas	86
1.17	Online & Offline mode	87
1.18	Serialization	88
1.19	Task Management	89
1.20	(P)oint-(I)n-(T)ime Database	95
1.21	API Reference	97
1.22	Qlib FAQ	191
1.23	Changelog	194
	Python Module Index	199
	Index	201

Qlib is an AI-oriented quantitative investment platform, which aims to realize the potential, empower the research, and create the value of AI technologies in quantitative investment.

1.1 Qlib: Quantitative Platform

1.1.1 Introduction



Qlib is an AI-oriented quantitative investment platform, which aims to realize the potential, empower the research, and create the value of AI technologies in quantitative investment.

With Qlib, users can easily try their ideas to create better Quant investment strategies.

1.1.2 Framework

At the module level, QLib is a platform that consists of above components. The components are designed as loose-coupled modules and each component could be used stand-alone.

Name	Description
<i>In- fras- truc- ture layer</i>	<i>Infrastructure</i> layer provides underlying support for Quant research. <i>DataServer</i> provides high-performance infrastructure for users to manage and retrieve raw data. <i>Trainer</i> provides flexible interface to control the training process of models which enable algorithms controlling the training process.
<i>Work- flow layer</i>	<i>Workflow</i> layer covers the whole workflow of quantitative investment. <i>Information Extractor</i> extracts data for models. <i>Forecast Model</i> focuses on producing all kinds of forecast signals (e.g. <i>alpha</i> , risk) for other modules. With these signals <i>Decision Generator</i> will generate the target trading decisions(i.e. portfolio, orders) to be executed by <i>Execution Env</i> (i.e. the trading market). There may be multiple levels of <i>Trading Agent</i> and <i>Execution Env</i> (e.g. an <i>order executor trading agent and intraday order execution environment</i> could behave like an interday trading environment and nested in <i>daily portfolio management trading agent and interday trading environment</i>)
<i>In- ter- face layer</i>	<i>Interface</i> layer tries to present a user-friendly interface for the underlying system. <i>Analyser</i> module will provide users detailed analysis reports of forecasting signals, portfolios and execution results

- The modules with hand-drawn style are under development and will be released in the future.
- The modules with dashed borders are highly user-customizable and extendible.

1.2 Quick Start

1.2.1 Introduction

This `Quick Start` guide tries to demonstrate

- It's very easy to build a complete Quant research workflow and try users' ideas with `QLib`.
- Though with public data and simple models, machine learning technologies work very well in practical Quant investment.

1.2.2 Installation

Users can easily install `QLib` according to the following steps:

- Before installing `QLib` from source, users need to install some dependencies:
- Clone the repository and install `QLib`

To know more about *installation*, please refer to [QLib Installation](#).

1.2.3 Prepare Data

Load and prepare data by running the following code:

This dataset is created by public data collected by crawler scripts in `scripts/data_collector/`, which have been released in the same repository. Users could create the same dataset with it.

To know more about *prepare data*, please refer to [Data Preparation](#).

1.2.4 Auto Quant Research Workflow

Qlib provides a tool named `qrun` to run the whole workflow automatically (including building dataset, training models, backtest and evaluation). Users can start an auto quant research workflow and have a graphical reports analysis according to the following steps:

- **Quant Research Workflow:**

- Run `qrun` with a config file of the LightGBM model `workflow_config_lightgbm.yaml` as following.
- **Workflow result** The result of `qrun` is as follows, which is also the typical result of Forecast model (alpha). Please refer to [Intraday Trading](#). for more details about the result.

		risk
excess_return_without_cost	mean	0.000605
	std	0.005481
	annualized_return	0.152373
	information_ratio	1.751319
	max_drawdown	-0.059055
excess_return_with_cost	mean	0.000410
	std	0.005478
	annualized_return	0.103265
	information_ratio	1.187411
	max_drawdown	-0.075024

To know more about *workflow* and *qrun*, please refer to [Workflow: Workflow Management](#).

- **Graphical Reports Analysis:**

- **Run `examples/workflow_by_code.ipynb` with jupyter notebook** Users can have portfolio analysis or prediction score (model prediction) analysis by run `examples/workflow_by_code.ipynb`.
- **Graphical Reports** Users can get graphical reports about the analysis, please refer to [Analysis: Evaluation & Results Analysis](#) for more details.

1.2.5 Custom Model Integration

Qlib provides a batch of models (such as `lightGBM` and `MLP` models) as examples of Forecast Model. In addition to the default model, users can integrate their own custom models into Qlib. If users are interested in the custom model, please refer to [Custom Model Integration](#).

1.3 Installation

1.3.1 Qlib Installation

Note: *Qlib* supports both *Windows* and *Linux*. It's recommended to use *Qlib* in *Linux*. *Qlib* supports Python3, which is up to Python3.8.

Users can easily install *Qlib* by pip according to the following command:

```
pip install pyqlib
```

Also, Users can install *Qlib* by the source code according to the following steps:

- Enter the root directory of *Qlib*, in which the file `setup.py` exists.
- Then, please execute the following command to install the environment dependencies and install *Qlib*:

```
$ pip install numpy
$ pip install --upgrade cython
$ git clone https://github.com/microsoft/qlib.git && cd qlib
$ python setup.py install
```

Note: It's recommended to use *anaconda/miniconda* to setup the environment. *Qlib* needs *lightgbm* and *pytorch* packages, use pip to install them.

Use the following code to make sure the installation successful:

```
>>> import qlib
>>> qlib.__version__
<LATEST VERSION>
```

1.4 Qlib Initialization

1.4.1 Initialization

Please follow the steps below to initialize *Qlib*.

Download and prepare the Data: execute the following command to download stock data. Please pay *attention* that the data is collected from [Yahoo Finance](#) and the data might not be perfect. We recommend users to prepare their own data if they have high-quality datasets. Please refer to [Data](#) for more information about customized dataset.

```
python scripts/get_data.py qlib_data --target_dir ~/.qlib/qlib_data/cn_data -
↪-region cn
```

Please refer to [Data Preparation](#) for more information about `get_data.py`,

Initialize *Qlib* before calling other APIs: run following code in python.

```
import qlib
# region in [REG_CN, REG_US]
from qlib.constant import REG_CN
provider_uri = "~/.qlib/qlib_data/cn_data" # target_dir
qlib.init(provider_uri=provider_uri, region=REG_CN)
```

Note: Do not import qlib package in the repository directory of Qlib, otherwise, errors may occur.

Parameters

Besides *provider_uri* and *region*, *qlib.init* has other parameters. The following are several important parameters of *qlib.init* (Qlib has a lot of config. Only part of parameters are limited here. More detailed setting can be found [here](#)):

- ***provider_uri*** Type: str. The URI of the Qlib data. For example, it could be the location where the data loaded by `get_data.py` are stored.

- ***region***

Type: str, optional parameter(default: *qlib.constant.REG_CN*). Currently: `qlib.constant.REG_US` ('us') and `qlib.constant.REG_CN` ('cn') is supported. Different value of *region* will result in different stock market mode. - `qlib.constant.REG_US`: US stock market. - `qlib.constant.REG_CN`: China stock market.

Different modes will result in different trading limitations and costs. The region is just [shortcuts for defining a batch of configurations](#), which include minimal trading order unit (`trade_unit`), trading limitation (`limit_threshold`), etc. It is not a necessary part and users can set the key configurations manually if the existing region setting can't meet their requirements.

- ***redis_host***

Type: str, optional parameter(default: "127.0.0.1"), host of *redis* The lock and cache mechanism relies on redis.

- ***redis_port*** Type: int, optional parameter(default: 6379), port of *redis*

Note: The value of *region* should be aligned with the data stored in *provider_uri*. Currently, `scripts/get_data.py` only provides China stock market data. If users want to use the US stock market data, they should prepare their own US-stock data in *provider_uri* and switch to US-stock mode.

Note: If Qlib fails to connect redis via *redis_host* and *redis_port*, cache mechanism will not be used! Please refer to [Cache](#) for details.

- ***exp_manager*** Type: dict, optional parameter, the setting of *experiment manager* to be used in qlib. Users can specify an experiment manager class, as well as the tracking URI for all the experiments. However, please be aware that we only support input of a dictionary in the following style for *exp_manager*. For more information about *exp_manager*, users can refer to [Recorder: Experiment Management](#).

```
# For example, if you want to set your tracking_uri to a <specific folder>, ↵
↵you can initialize qlib below
qlib.init(provider_uri=provider_uri, region=REG_CN, exp_manager= {
    "class": "MLflowExpManager",
    "module_path": "qlib.workflow.exp_m",
    "kwargs": {
        "uri": "python_execution_path/mlruns",
        "default_exp_name": "Experiment",
    }
})
```

- **mongo** Type: dict, optional parameter, the setting of **MongoDB** which will be used in some features such as **Task Management**, with high performance and clustered processing. Users need to follow the steps in [installation](#) to install MongoDB firstly and then access it via a URI. Users can access mongoddb with credential by setting “task_url” to a string like “*mongodb://%s:%s@%s*” % (*user, pwd, host* + “:” + *port*).

```
# For example, you can initialize qlib below
qlib.init(provider_uri=provider_uri, region=REG_CN, mongo={
    "task_url": "mongodb://localhost:27017/", # your mongo url
    "task_db_name": "rolling_db", # the database name of Task Management
})
```

- **logging_level** The logging level for the system.
- **kernels** The number of processes used when calculating features in Qlib’s expression engine. It is very helpful to set it to 1 when you are debuggin an expression calculating exception

1.5 Data Retrieval

1.5.1 Introduction

Users can get stock data with Qlib. The following examples demonstrate the basic user interface.

1.5.2 Examples

QLib Initialization:

Note: In order to get the data, users need to initialize Qlib with *qlib.init* first. Please refer to [initialization](#).

If users followed steps in [initialization](#) and downloaded the data, they should use the following code to initialize qlib

```
>> import qlib
>> qlib.init(provider_uri='~/qlib/qlib_data/cn_data')
```

Load trading calendar with given time range and frequency:

```
>> from qlib.data import D
>> D.calendar(start_time='2010-01-01', end_time='2017-12-31', freq='day')[:2]
[Timestamp('2010-01-04 00:00:00'), Timestamp('2010-01-05 00:00:00')]
```

Parse a given market name into a stock pool config:

```
>> from qlib.data import D
>> D.instruments(market='all')
{'market': 'all', 'filter_pipe': []}
```

Load instruments of certain stock pool in the given time range:

```
>> from qlib.data import D
>> instruments = D.instruments(market='csi300')
>> D.list_instruments(instruments=instruments, start_time='2010-01-01', end_time=
→ '2017-12-31', as_list=True)[:6]
['SH600036', 'SH600110', 'SH600087', 'SH600900', 'SH600089', 'SZ000912']
```

Load dynamic instruments from a base market according to a name filter

```
>> from qlib.data import D
>> from qlib.data.filter import NameDFilter
>> nameDFilter = NameDFilter(name_rule_re='SH[0-9]{4}55')
>> instruments = D.instruments(market='csi300', filter_pipe=[nameDFilter])
>> D.list_instruments(instruments=instruments, start_time='2015-01-01', end_time=
↳ '2016-02-15', as_list=True)
['SH600655', 'SH601555']
```

Load dynamic instruments from a base market according to an expression filter

```
>> from qlib.data import D
>> from qlib.data.filter import ExpressionDFilter
>> expressionDFilter = ExpressionDFilter(rule_expression='$close>2000')
>> instruments = D.instruments(market='csi300', filter_pipe=[expressionDFilter])
>> D.list_instruments(instruments=instruments, start_time='2015-01-01', end_time=
↳ '2016-02-15', as_list=True)
['SZ000651', 'SZ000002', 'SH600655', 'SH600570']
```

For more details about filter, please refer [Filter API](#).

Load features of certain instruments in a given time range:

```
>> from qlib.data import D
>> instruments = ['SH600000']
>> fields = ['$close', '$volume', 'Ref($close, 1)', 'Mean($close, 3)', '$high-$low']
>> D.features(instruments, fields, start_time='2010-01-01', end_time='2017-12-31',
↳ freq='day').head()
```

			\$close	\$volume	Ref(\$close, 1)	Mean(\$close, 3)	\$high-\$low
↳ \$low							
	instrument	datetime					
	SH600000	2010-01-04	86.778313	16162960.0	88.825928	88.061483	↳
↳ 2.907631							
		2010-01-05	87.433578	28117442.0	86.778313	87.679273	↳
↳ 3.235252							
		2010-01-06	85.713585	23632884.0	87.433578	86.641825	↳
↳ 1.720009							
		2010-01-07	83.788803	20813402.0	85.713585	85.645322	↳
↳ 3.030487							
		2010-01-08	84.730675	16044853.0	83.788803	84.744354	↳
↳ 2.047623							

Load features of certain stock pool in a given time range:

Note: With cache enabled, the qlib data server will cache data all the time for the requested stock pool and fields, it may take longer to process the request for the first time than that without cache. But after the first time, requests with the same stock pool and fields will hit the cache and be processed faster even the requested time period changes.

```
>> from qlib.data import D
>> from qlib.data.filter import NameDFilter, ExpressionDFilter
>> nameDFilter = NameDFilter(name_rule_re='SH[0-9]{4}55')
>> expressionDFilter = ExpressionDFilter(rule_expression='$close>Ref($close,1)')
>> instruments = D.instruments(market='csi300', filter_pipe=[nameDFilter,
↳ expressionDFilter])
```

(continues on next page)

(continued from previous page)

```
>> fields = ['$close', '$volume', 'Ref($close, 1)', 'Mean($close, 3)', '$high-$low']
>> D.features(instruments, fields, start_time='2010-01-01', end_time='2017-12-31',
↪ freq='day').head()
```

			\$close	\$volume	Ref(\$close, 1)	Mean(\$close, 3)
↪ \$high-\$low						
instrument	datetime					
SH600655	2010-01-04	2699.567383	158193.328125	2619.070312	2626.	
↪ 097738	124.580566					
	2010-01-08	2612.359619	77501.406250	2584.567627	2623.	
↪ 220133	83.373047					
	2010-01-11	2712.982422	160852.390625	2612.359619	2636.	
↪ 636556	146.621582					
	2010-01-12	2788.688232	164587.937500	2712.982422	2704.	
↪ 676758	128.413818					
	2010-01-13	2790.604004	145460.453125	2788.688232	2764.	
↪ 091553	128.413818					

For more details about features, please refer [Feature API](#).

Note: When calling `D.features()` at the client, use parameter `disk_cache=0` to skip dataset cache, use `disk_cache=1` to generate and use dataset cache. In addition, when calling at the server, users can use `disk_cache=2` to update the dataset cache.

When you are building complicated expressions, implementing all the expressions in a single string may not be easy. For example, it looks quite long and complicated:

```
>> from qlib.data import D
>> data = D.features(["sh600519"], ["(($high / $close) + ($open / $close)) * (($high /
↪ $close) + ($open / $close)) / (($high / $close) + ($open / $close))"], start_time=
↪ "20200101")
```

But using string is not the only way to implement the expression. You can also implement expression by code. Here is an example which does the same thing as above examples.

```
>> from qlib.data.ops import *
>> f1 = Feature("high") / Feature("close")
>> f2 = Feature("open") / Feature("close")
>> f3 = f1 + f2
>> f4 = f3 * f3 / f3

>> data = D.features(["sh600519"], [f4], start_time="20200101")
>> data.head()
```

1.5.3 API

To know more about how to use the Data, go to API Reference: [Data API](#)

1.6 Custom Model Integration

1.6.1 Introduction

Qlib's *Model Zoo* includes models such as LightGBM, MLP, LSTM, etc.. These models are examples of Forecast Model. In addition to the default models Qlib provide, users can integrate their own custom models into Qlib.

Users can integrate their own custom models according to the following steps.

- Define a custom model class, which should be a subclass of the `qlib.model.base.Model`.
- Write a configuration file that describes the path and parameters of the custom model.
- Test the custom model.

1.6.2 Custom Model Class

The Custom models need to inherit `qlib.model.base.Model` and override the methods in it.

- **Override the `__init__` method**

- Qlib passes the initialized parameters to the `__init__` method.
- The hyperparameters of model in the configuration must be consistent with those defined in the `__init__` method.
- Code Example: In the following example, the hyperparameters of model in the configuration file should contain parameters such as `loss:mse`.

```
def __init__(self, loss='mse', **kwargs):
    if loss not in {'mse', 'binary'}:
        raise NotImplementedError
    self._scorer = mean_squared_error if loss == 'mse' else roc_auc_score
    self._params.update(objective=loss, **kwargs)
    self._model = None
```

- **Override the `fit` method**

- Qlib calls the fit method to train the model.
- The parameters must include training feature *dataset*, which is designed in the interface.
- The parameters could include some *optional* parameters with default values, such as `num_boost_round = 1000` for *GBDT*.
- Code Example: In the following example, `num_boost_round = 1000` is an optional parameter.

```
def fit(self, dataset: DatasetH, num_boost_round = 1000, **kwargs):

    # prepare dataset for lgb training and evaluation
    df_train, df_valid = dataset.prepare(
        ["train", "valid"], col_set=["feature", "label"], data_
↪key=DataHandlerLP.DK_L
    )
    x_train, y_train = df_train["feature"], df_train["label"]
    x_valid, y_valid = df_valid["feature"], df_valid["label"]

    # Lightgbm need 1D array as its label
    if y_train.values.ndim == 2 and y_train.values.shape[1] == 1:
```

(continues on next page)

(continued from previous page)

```

        y_train, y_valid = np.squeeze(y_train.values), np.squeeze(y_valid.
↪values)
    else:
        raise ValueError("LightGBM doesn't support multi-label training")

    dtrain = lgb.Dataset(x_train.values, label=y_train)
    dvalid = lgb.Dataset(x_valid.values, label=y_valid)

    # fit the model
    self.model = lgb.train(
        self.params,
        dtrain,
        num_boost_round=num_boost_round,
        valid_sets=[dtrain, dvalid],
        valid_names=["train", "valid"],
        early_stopping_rounds=early_stopping_rounds,
        verbose_eval=verbose_eval,
        evals_result=evals_result,
        **kwargs
    )

```

- **Override the *predict* method**

- The parameters must include the parameter *dataset*, which will be used to get the test dataset.
- Return the *prediction score*.
- Please refer to [Model API](#) for the parameter types of the fit method.
- Code Example: In the following example, users need to use *LightGBM* to predict the label(such as *preds*) of test data *x_test* and return it.

```

def predict(self, dataset: DatasetH, **kwargs) -> pandas.Series:
    if self.model is None:
        raise ValueError("model is not fitted yet!")
    x_test = dataset.prepare("test", col_set="feature", data_
↪key=DataHandlerLP.DK_I)
    return pd.Series(self.model.predict(x_test.values), index=x_test.index)

```

- **Override the *finetune* method (Optional)**

- This method is optional to the users. When users want to use this method on their own models, they should inherit the *ModelFT* base class, which includes the interface of *finetune*.
- The parameters must include the parameter *dataset*.
- Code Example: In the following example, users will use *LightGBM* as the model and finetune it.

```

def finetune(self, dataset: DatasetH, num_boost_round=10, verbose_eval=20):
    # Based on existing model and finetune by train more rounds
    dtrain, _ = self._prepare_data(dataset)
    self.model = lgb.train(
        self.params,
        dtrain,
        num_boost_round=num_boost_round,
        init_model=self.model,
        valid_sets=[dtrain],
        valid_names=["train"],

```

(continues on next page)

(continued from previous page)

```
        verbose_eval=verbose_eval,
    )
```

1.6.3 Configuration File

The configuration file is described in detail in the [Workflow](#) document. In order to integrate the custom model into QLib, users need to modify the “model” field in the configuration file. The configuration describes which models to use and how we can initialize it.

- Example: The following example describes the *model* field of configuration file about the custom lightgbm model mentioned above, where *module_path* is the module path, *class* is the class name, and *args* is the hyperparameter passed into the `__init__` method. All parameters in the field is passed to *self._params* by ***kwargs* in `__init__` except *loss = mse*.

```
model:
  class: LGBModel
  module_path: qlib.contrib.model.gbd
  args:
    loss: mse
    colsample_bytree: 0.8879
    learning_rate: 0.0421
    subsample: 0.8789
    lambda_l1: 205.6999
    lambda_l2: 580.9768
    max_depth: 8
    num_leaves: 210
    num_threads: 20
```

Users could find configuration file of the baselines of the Model in `examples/benchmarks`. All the configurations of different models are listed under the corresponding model folder.

1.6.4 Model Testing

Assuming that the configuration file is `examples/benchmarks/LightGBM/workflow_config_lightgbm.yaml`, users can run the following command to test the custom model:

```
cd examples # Avoid running program under the directory contains `qlib`
qrun benchmarks/LightGBM/workflow_config_lightgbm.yaml
```

Note: `qrun` is a built-in command of QLib.

Also, Model can also be tested as a single module. An example has been given in `examples/workflow_by_code.ipynb`.

1.6.5 Reference

To know more about Forecast Model, please refer to [Forecast Model: Model Training & Prediction](#) and [Model API](#).

1.7 Workflow: Workflow Management

1.7.1 Introduction

The components in [QLib Framework](#) are designed in a loosely-coupled way. Users could build their own Quant research workflow with these components like [Example](#).

Besides, [QLib](#) provides more user-friendly interfaces named `qrun` to automatically run the whole workflow defined by configuration. Running the whole workflow is called an *execution*. With `qrun`, user can easily start an *execution*, which includes the following steps:

- **Data**
 - Loading
 - Processing
 - Slicing
- **Model**
 - Training and inference
 - Saving & loading
- **Evaluation**
 - Forecast signal analysis
 - Backtest

For each *execution*, [QLib](#) has a complete system to tracking all the information as well as artifacts generated during training, inference and evaluation phase. For more information about how [QLib](#) handles this, please refer to the related document: [Recorder: Experiment Management](#).

1.7.2 Complete Example

Before getting into details, here is a complete example of `qrun`, which defines the workflow in typical Quant research. Below is a typical config file of `qrun`.

```
qlib_init:
  provider_uri: "~/qlib/qlib_data/cn_data"
  region: cn
market: &market csi300
benchmark: &benchmark SH000300
data_handler_config: &data_handler_config
  start_time: 2008-01-01
  end_time: 2020-08-01
  fit_start_time: 2008-01-01
  fit_end_time: 2014-12-31
  instruments: *market
port_analysis_config: &port_analysis_config
  strategy:
    class: TopkDropoutStrategy
    module_path: qlib.contrib.strategy.strategy
    kwargs:
      topk: 50
      n_drop: 5
      signal:
```

(continues on next page)

(continued from previous page)

```

        - <MODEL>
        - <DATASET>

    backtest:
        limit_threshold: 0.095
        account: 100000000
        benchmark: *benchmark
        deal_price: close
        open_cost: 0.0005
        close_cost: 0.0015
        min_cost: 5

    task:
        model:
            class: LGBModel
            module_path: qlib.contrib.model.gbd
            kwargs:
                loss: mse
                colsample_bytree: 0.8879
                learning_rate: 0.0421
                subsample: 0.8789
                lambda_l1: 205.6999
                lambda_l2: 580.9768
                max_depth: 8
                num_leaves: 210
                num_threads: 20
        dataset:
            class: DatasetH
            module_path: qlib.data.dataset
            kwargs:
                handler:
                    class: Alpha158
                    module_path: qlib.contrib.data.handler
                    kwargs: *data_handler_config
                segments:
                    train: [2008-01-01, 2014-12-31]
                    valid: [2015-01-01, 2016-12-31]
                    test: [2017-01-01, 2020-08-01]
        record:
            - class: SignalRecord
              module_path: qlib.workflow.record_temp
              kwargs: {}
            - class: PortAnaRecord
              module_path: qlib.workflow.record_temp
              kwargs:
                  config: *port_analysis_config

```

After saving the config into *configuration.yaml*, users could start the workflow and test their ideas with a single command below.

```
qrun configuration.yaml
```

If users want to use *qrun* under debug mode, please use the following command:

```
python -m pdb qlib/workflow/cli.py examples/benchmarks/LightGBM/workflow_config_
↪lightgbm_Alpha158.yaml
```

Note: `qrun` will be placed in your `$PATH` directory when installing Qlib.

Note: The symbol `&` in `yaml` file stands for an anchor of a field, which is useful when another fields include this parameter as part of the value. Taking the configuration file above as an example, users can directly change the value of `market` and `benchmark` without traversing the entire configuration file.

1.7.3 Configuration File

Let's get into details of `qrun` in this section. Before using `qrun`, users need to prepare a configuration file. The following content shows how to prepare each part of the configuration file.

The design logic of the configuration file is very simple. It predefines fixed workflows and provide this `yaml` interface to users to define how to initialize each component. It follow the design of `init_instance_by_config`. It defines the initialization of each component of Qlib, which typically include the class and the initialization arguments.

For example, the following `yaml` and code are equivalent.

```
model:
  class: LGBModel
  module_path: qlib.contrib.model.gbd
  kwargs:
    loss: mse
    colsample_bytree: 0.8879
    learning_rate: 0.0421
    subsample: 0.8789
    lambda_l1: 205.6999
    lambda_l2: 580.9768
    max_depth: 8
    num_leaves: 210
    num_threads: 20
```

```
from qlib.contrib.model.gbd import LGBModel
kwargs = {
    "loss": "mse",
    "colsample_bytree": 0.8879,
    "learning_rate": 0.0421,
    "subsample": 0.8789,
    "lambda_l1": 205.6999,
    "lambda_l2": 580.9768,
    "max_depth": 8,
    "num_leaves": 210,
    "num_threads": 20,
}
LGBModel(kwargs)
```

Qlib Init Section

At first, the configuration file needs to contain several basic parameters which will be used for `qlib` initialization.

```
provider_uri: ~/.qlib/qlib_data/cn_data
region: cn
```

The meaning of each field is as follows:

- **provider_uri** Type: str. The URI of the Qlib data. For example, it could be the location where the data loaded by `get_data.py` are stored.
- **region**
 - If `region == "us"`, Qlib will be initialized in US-stock mode.
 - If `region == "cn"`, Qlib will be initialized in China-stock mode.

Note: The value of `region` should be aligned with the data stored in `provider_uri`.

Task Section

The `task` field in the configuration corresponds to a `task`, which contains the parameters of three different subsections: `Model`, `Dataset` and `Record`.

Model Section

In the `task` field, the `model` section describes the parameters of the model to be used for training and inference. For more information about the base `Model` class, please refer to [Qlib Model](#).

```
model:
  class: LGBModel
  module_path: qlib.contrib.model.gbd
  kwargs:
    loss: mse
    colsample_bytree: 0.8879
    learning_rate: 0.0421
    subsample: 0.8789
    lambda_l1: 205.6999
    lambda_l2: 580.9768
    max_depth: 8
    num_leaves: 210
    num_threads: 20
```

The meaning of each field is as follows:

- **class** Type: str. The name for the model class.
- **module_path** Type: str. The path for the model in qlib.
- **kwargs** The keywords arguments for the model. Please refer to the specific model implementation for more information: [models](#).

Note: Qlib provides a util named: `init_instance_by_config` to initialize any class inside Qlib with the configuration includes the fields: `class`, `module_path` and `kwargs`.

Dataset Section

The `dataset` field describes the parameters for the `Dataset` module in Qlib as well those for the module `DataHandler`. For more information about the `Dataset` module, please refer to [Qlib Model](#).

The keywords arguments configuration of the `DataHandler` is as follows:

```
data_handler_config: &data_handler_config
  start_time: 2008-01-01
  end_time: 2020-08-01
  fit_start_time: 2008-01-01
  fit_end_time: 2014-12-31
  instruments: *market
```

Users can refer to the document of `DataHandler` for more information about the meaning of each field in the configuration.

Here is the configuration for the `Dataset` module which will take care of data preprocessing and slicing during the training and testing phase.

```
dataset:
  class: DatasetH
  module_path: qlib.data.dataset
  kwargs:
    handler:
      class: Alpha158
      module_path: qlib.contrib.data.handler
      kwargs: *data_handler_config
    segments:
      train: [2008-01-01, 2014-12-31]
      valid: [2015-01-01, 2016-12-31]
      test: [2017-01-01, 2020-08-01]
```

Record Section

The *record* field is about the parameters the `Record` module in `Qlib`. `Record` is responsible for tracking training process and results such as *information Coefficient (IC)* and *backtest* in a standard format.

The following script is the configuration of *backtest* and the *strategy* used in *backtest*:

```
port_analysis_config: &port_analysis_config
  strategy:
    class: TopkDropoutStrategy
    module_path: qlib.contrib.strategy.strategy
    kwargs:
      topk: 50
      n_drop: 5
      signal:
        - <MODEL>
        - <DATASET>
  backtest:
    limit_threshold: 0.095
    account: 100000000
    benchmark: *benchmark
    deal_price: close
    open_cost: 0.0005
    close_cost: 0.0015
    min_cost: 5
```

For more information about the meaning of each field in configuration of *strategy* and *backtest*, users can look up the documents: `Strategy` and `Backtest`.

Here is the configuration details of different *Record Template* such as `SignalRecord` and `PortAnaRecord`:

```
record:
- class: SignalRecord
  module_path: qlib.workflow.record_temp
  kwargs: {}
- class: PortAnaRecord
  module_path: qlib.workflow.record_temp
  kwargs:
    config: *port_analysis_config
```

For more information about the `Record` module in `Qlib`, user can refer to the related document: [Record](#).

1.8 Data Layer: Data Framework & Usage

1.8.1 Introduction

`Data Layer` provides user-friendly APIs to manage and retrieve data. It provides high-performance data infrastructure.

It is designed for quantitative investment. For example, users could build formulaic alphas with `Data Layer` easily. Please refer to [Building Formulaic Alphas](#) for more details.

The introduction of `Data Layer` includes the following parts.

- Data Preparation
- Data API
- Data Loader
- Data Handler
- Dataset
- Cache
- Data and Cache File Structure

Here is a typical example of `Qlib` data workflow

- Users download data and converting data into `Qlib` format(with filename suffix `.bin`). In this step, typically only some basic data are stored on disk(such as OHLCV).
- Creating some basic features based on `Qlib`'s expression Engine(e.g. “`Ref($close, 60) / $close`”, the return of last 60 trading days). Supported operators in the expression engine can be found [here](#). This step is typically implemented in `Qlib`'s [Data Loader](#) which is a component of [Data Handler](#).
- If users require more complicated data processing (e.g. data normalization), [Data Handler](#) support user-customized processors to process data(some predefined processors can be found [here](#)). The processors are different from operators in expression engine. It is designed for some complicated data processing methods which is hard to supported in operators in expression engine.
- At last, [Dataset](#) is responsible to prepare model-specific dataset from the processed data of `Data Handler`

1.8.2 Data Preparation

Qlib Format Data

We've specially designed a data structure to manage financial data, please refer to the [File storage design](#) section in [Qlib paper](#) for detailed information. Such data will be stored with filename suffix *.bin* (We'll call them *.bin* file, *.bin* format, or qlib format). *.bin* file is designed for scientific computing on finance data.

Qlib provides two different off-the-shelf datasets, which can be accessed through this [link](#):

Dataset	US Market	China Market
Alpha360		
Alpha158		

Also, Qlib provides a high-frequency dataset. Users can run a high-frequency dataset example through this [link](#).

Qlib Format Dataset

Qlib has provided an off-the-shelf dataset in *.bin* format, users could use the script `scripts/get_data.py` to download the China-Stock dataset as follows. The price volume data look different from the actual dealing price because of they are **adjusted** ([adjusted price](#)). And then you may find that the adjusted price may be different from different data sources. This is because different data sources may vary in the way of adjusting prices. Qlib normalize the price on first trading day of each stock to 1 when adjusting them. Users can leverage *\$factor* to get the original trading price (e.g. *\$close / \$factor* to get the original close price).

```
# download 1d
python scripts/get_data.py qlib_data --target_dir ~/.qlib/qlib_data/cn_data --region_
↪cn

# download 1min
python scripts/get_data.py qlib_data --target_dir ~/.qlib/qlib_data/qlib_cn_1min --
↪region cn --interval 1min
```

In addition to China-Stock data, Qlib also includes a US-Stock dataset, which can be downloaded with the following command:

```
python scripts/get_data.py qlib_data --target_dir ~/.qlib/qlib_data/us_data --region_
↪us
```

After running the above command, users can find china-stock and us-stock data in Qlib format in the `~/.qlib/qlib_data/cn_data` directory and `~/.qlib/qlib_data/us_data` directory respectively.

Qlib also provides the scripts in `scripts/data_collector` to help users crawl the latest data on the Internet and convert it to qlib format.

When Qlib is initialized with this dataset, users could build and evaluate their own models with it. Please refer to [Initialization](#) for more details.

Automatic update of daily frequency data

It is recommended that users update the data manually once (`--trading_date 2021-05-25`) and then set it to update automatically.

For more information refer to: [yahoo collector](#)

- Automatic update of data to the “qlib” directory each trading day(Linux)
 - use *crontab*: `crontab -e`

- set up timed tasks:

```
* * * * 1-5 python <script path> update_data_to_bin --qlib_data_ld_
↪dir <user data dir>
```

* **script path:** *scripts/data_collector/yahoo/collector.py*

- Manual update of data

```
python scripts/data_collector/yahoo/collector.py update_data_to_
↪bin --qlib_data_ld_dir <user data dir> --trading_date <start_
↪date> --end_date <end date>
```

- *trading_date*: start of trading day
- *end_date*: end of trading day(not included)

Converting CSV Format into Qlib Format

Qlib has provided the script `scripts/dump_bin.py` to convert **any** data in CSV format into *.bin* files (Qlib format) as long as they are in the correct format.

Besides downloading the prepared demo data, users could download demo data directly from the Collector as follows for reference to the CSV format. Here are some example:

for daily data:

```
python scripts/get_data.py csv_data_cn --target_dir ~/.qlib/csv_data/cn_data
```

for 1min data:

```
python scripts/data_collector/yahoo/collector.py download_data --source_dir ~/.
↪qlib/stock_data/source/cn_1min --region CN --start 2021-05-20 --end 2021-05-23 -
↪delay 0.1 --interval 1min --limit_nums 10
```

Users can also provide their own data in CSV format. However, the CSV data **must satisfies** following criterions:

- CSV file is named after a specific stock *or* the CSV file includes a column of the stock name
 - Name the CSV file after a stock: *SH600000.csv*, *AAPL.csv* (not case sensitive).
 - CSV file includes a column of the stock name. User **must** specify the column name when dumping the data. Here is an example:

```
python scripts/dump_bin.py dump_all ... --symbol_field_name symbol
```

where the data are in the following format:

- CSV file **must** includes a column for the date, and when dumping the data, user must specify the date column name. Here is an example:

```
python scripts/dump_bin.py dump_all ... --date_field_name date
```

where the data are in the following format:

Supposed that users prepare their CSV format data in the directory `~/.qlib/csv_data/my_data`, they can run the following command to start the conversion.

```
python scripts/dump_bin.py dump_all --csv_path ~/.qlib/csv_data/my_data --qlib_dir ~/
↪.qlib/qlib_data/my_data --include_fields open,close,high,low,volume,factor
```

For other supported parameters when dumping the data into *.bin* file, users can refer to the information by running the following commands:

```
python dump_bin.py dump_all --help
```

After conversion, users can find their Qlib format data in the directory *~/.qlib/qlib_data/my_data*.

Note: The arguments of *--include_fields* should correspond with the column names of CSV files. The columns names of dataset provided by Qlib should include open, close, high, low, volume and factor at least.

- **open** The adjusted opening price
- **close** The adjusted closing price
- **high** The adjusted highest price
- **low** The adjusted lowest price
- **volume** The adjusted trading volume
- **factor** The Restoration factor. Normally, $\text{factor} = \text{adjusted_price} / \text{original_price}$, *adjusted price* reference: [split adjusted](#)

In the convention of Qlib data processing, *open*, *close*, *high*, *low*, *volume*, *money* and *factor* will be set to NaN if the stock is suspended. If you want to use your own alpha-factor which can't be calculate by OCHLV, like PE, EPS and so on, you could add it to the CSV files with OHCLV together and then dump it to the Qlib format data.

Stock Pool (Market)

Qlib defines [stock pool](#) as stock list and their date ranges. Predefined stock pools (e.g. csi300) may be imported as follows.

```
python collector.py --index_name CSI300 --qlib_dir <user qlib data dir> --method_
↪ parse_instruments
```

Multiple Stock Modes

Qlib now provides two different stock modes for users: China-Stock Mode & US-Stock Mode. Here are some different settings of these two modes:

Region	Trade Unit	Limit Threshold
China	100	0.099
US	1	None

The *trade unit* defines the unit number of stocks can be used in a trade, and the *limit threshold* defines the bound set to the percentage of ups and downs of a stock.

- **If users use Qlib in china-stock mode, china-stock data is required.** Users can use Qlib in china-stock mode according to
 - Download china-stock in qlib format, please refer to section [Qlib Format Dataset](#).
 - **Initialize Qlib in china-stock mode** Supposed that users download their Qlib format data in the directory *~/.qlib/qlib_data/cn_data*. Users only need to initialize Qlib as follows.

```
from qlib.constant import REG_CN
qlib.init(provider_uri='~/qlib/qlib_data/cn_data', region=REG_CN)
```

- If users use Qlib in US-stock mode, US-stock data is required. Qlib also provides a script to download US-stock data. U

- Download us-stock in qlib format, please refer to section *Qlib Format Dataset*.
- **Initialize Qlib in US-stock mode** Supposed that users prepare their Qlib format data in the directory ~/qlib/qlib_data/us_data. Users only need to initialize Qlib as follows.

```
from qlib.config import REG_US
qlib.init(provider_uri='~/qlib/qlib_data/us_data', region=REG_US)
```

Note: PRs for new data source are highly welcome! Users could commit the code to crawl data as a PR like [the examples here](#). And then we will use the code to create data cache on our server which other users could use directly.

1.8.3 Data API

Data Retrieval

Users can use APIs in `qlib.data` to retrieve data, please refer to [Data Retrieval](#).

Feature

Qlib provides *Feature* and *ExpressionOps* to fetch the features according to users' needs.

- **Feature** Load data from the data provider. User can get the features like *\$high*, *\$low*, *\$open*, *\$close*, etc, which should correspond with the arguments of *–include_fields*, please refer to section *Converting CSV Format into Qlib Format*.
- **ExpressionOps** *ExpressionOps* will use operator for feature construction. To know more about Operator, please refer to [Operator API](#). Also, Qlib supports users to define their own custom Operator, an example has been given in `tests/test_register_ops.py`.

To know more about Feature, please refer to [Feature API](#).

Filter

Qlib provides *NameDFilter* and *ExpressionDFilter* to filter the instruments according to users' needs.

- **NameDFilter** Name dynamic instrument filter. Filter the instruments based on a regulated name format. A name rule regular expression is required.
- **ExpressionDFilter** Expression dynamic instrument filter. Filter the instruments based on a certain expression. An expression rule indicating a certain feature field is required.
 - *basic features filter*: `rule_expression = '$close/$open>5'`
 - *cross-sectional features filter*: `rule_expression = '$rank($close)<10'`
 - *time-sequence features filter*: `rule_expression = '$Ref($close, 3)>100'`

Here is a simple example showing how to use filter in a basic Qlib workflow configuration file:

```
filter: &filter
  filter_type: ExpressionDFilter
  rule_expression: "Ref($close, -2) / Ref($close, -1) > 1"
  filter_start_time: 2010-01-01
  filter_end_time: 2010-01-07
  keep: False

data_handler_config: &data_handler_config
  start_time: 2010-01-01
  end_time: 2021-01-22
  fit_start_time: 2010-01-01
  fit_end_time: 2015-12-31
  instruments: *market
  filter_pipe: [*filter]
```

To know more about `Filter`, please refer to [Filter API](#).

Reference

To know more about `Data` API, please refer to [Data API](#).

1.8.4 Data Loader

`Data Loader` in `Qlib` is designed to load raw data from the original data source. It will be loaded and used in the `Data Handler` module.

QlibDataLoader

The `QlibDataLoader` class in `Qlib` is such an interface that allows users to load raw data from the `Qlib` data source.

StaticDataLoader

The `StaticDataLoader` class in `Qlib` is such an interface that allows users to load raw data from file or as provided.

Interface

Here are some interfaces of the `QlibDataLoader` class:

class `qlib.data.dataset.loader.DataLoader`

`DataLoader` is designed for loading raw data from original data source.

load (`instruments`, `start_time=None`, `end_time=None`) → `pandas.core.frame.DataFrame`
load the data as `pd.DataFrame`.

Example of the data (The multi-index of the columns is optional.):

	feature	label			
	\$close	\$volume	Ref(\$close, 1)	Mean(
↩️\$close, 3)	\$high-\$low	LABEL0			

(continues on next page)

(continued from previous page)

datetime	instrument				
2010-01-04	SH600000	81.807068	17145150.0	83.737389	↪
↪83.016739	2.741058	0.0032			
	SH600004	13.313329	11800983.0	13.313329	↪
↪13.317701	0.183632	0.0042			
	SH600005	37.796539	12231662.0	38.258602	↪
↪37.919757	0.970325	0.0289			

Parameters

- **instruments** (*str* or *dict*) – it can either be the market name or the config file of instruments generated by InstrumentProvider.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.

Returns data load from the under layer source**Return type** pd.DataFrame**API**

To know more about `Data Loader`, please refer to [Data Loader API](#).

1.8.5 Data Handler

The `Data Handler` module in `Qlib` is designed to handler those common data processing methods which will be used by most of the models.

Users can use `Data Handler` in an automatic workflow by `qrun`, refer to [Workflow: Workflow Management](#) for more details.

DataHandlerLP

In addition to use `Data Handler` in an automatic workflow with `qrun`, `Data Handler` can be used as an independent module, by which users can easily preprocess data (standardization, remove NaN, etc.) and build datasets.

In order to achieve so, `Qlib` provides a base class `qlib.data.dataset.DataHandlerLP`. The core idea of this class is that: we will have some learnable `Processors` which can learn the parameters of data processing(e.g., parameters for zscore normalization). When new data comes in, these *trained Processors* can then process the new data and thus processing real-time data in an efficient way becomes possible. More information about `Processors` will be listed in the next subsection.

Interface

Here are some important interfaces that `DataHandlerLP` provides:

```
class qlib.data.dataset.handler.DataHandlerLP (instruments=None,
                                              start_time=None, end_time=None,
                                              data_loader: Union[dict, str,
qlib.data.dataset.loader.DataLoader]
                                              = None, infer_processors: List[T] =
[], learn_processors: List[T] = [],
                                              shared_processors: List[T] = [], process_type='append',
                                              drop_raw=False,
                                              **kwargs)
```

DataHandler with (L)earnable (P)rocessor

This handler will produce three pieces of data in `pd.DataFrame` format. - `DK_R / self._data`: the raw data loaded from the loader - `DK_I / self._infer`: the data processed for inference - `DK_L / self._learn`: the data processed for learning model.

The motivation of using different processor workflows for learning and inference Here are some examples. - The instrument universe for learning and inference may be different. - The processing of some samples may rely on label (for example, some samples hit the limit may need extra processing or be dropped).

These processors only apply to the learning phase.

Tips to improve the performance of data handler - To reduce the memory cost

- `drop_raw=True`: this will modify the data inplace on raw data;

```
__init__(instruments=None, start_time=None, end_time=None, data_loader: Union[dict,
str, qlib.data.dataset.loader.DataLoader] = None, infer_processors: List[T] = [],
learn_processors: List[T] = [], shared_processors: List[T] = [], process_type='append',
drop_raw=False, **kwargs)
```

Parameters

- **infer_processors** (*list*) –
 - list of <description info> of processors to generate data for inference
 - example of <description info>:
- **learn_processors** (*list*) – similar to `infer_processors`, but for generating data for learning models
- **process_type** (*str*) – `PTYPE_I` = 'independent'
 - `self._infer` will be processed by `infer_processors`
 - `self._learn` will be processed by `learn_processors`

`PTYPE_A` = 'append'

 - `self._infer` will be processed by `infer_processors`
 - `self._learn` will be processed by `infer_processors` + `learn_processors`
 - * (e.g. `self._infer` processed by `learn_processors`)
- **drop_raw** (*bool*) – Whether to drop the raw data

fit ()
fit data without processing the data

fit_process_data ()
fit and process data

The input of the *fit* will be the output of the previous processor

process_data (*with_fit*: *bool* = *False*)

process_data data. Fun *processor.fit* if necessary

Notation: (data) [processor]

data processing flow of self.process_type == DataHandlerLP.PTYPE_I (self._data)-
[shared_processors]-(_shared_df)-[learn_processors]-(_learn_df)

-[infer_processors]-(_infer_df)

data processing flow of self.process_type == DataHandlerLP.PTYPE_A (self._data)-
[shared_processors]-(_shared_df)-[infer_processors]-(_infer_df)-[learn_processors]-(_learn_df)

Parameters with_fit (*bool*) – The input of the *fit* will be the output of the previous processor

config (*processor_kwargs*: *dict* = *None*, ***kwargs*)

configuration of data. # what data to be loaded from data source

This method will be used when loading pickled handler from dataset. The data will be initialized with different time range.

setup_data (*init_type*: *str* = '*fit_seq*', ***kwargs*)

Set up the data in case of running initialization for multiple time

Parameters

- **init_type** (*str*) – The type *IT_** listed above.
- **enable_cache** (*bool*) – default value is false:
– if *enable_cache* == True:

the processed data will be saved on disk, and handler will load the cached data from the disk directly when we call *init* next time

fetch (*selector*: *Union*[*pandas._libs.tslibs.timestamps.Timestamp*, *slice*, *str*] = *slice(None, None, None)*, *level*: *Union*[*str*, *int*] = '*datetime*', *col_set*='*__all*', *data_key*: *str* = '*infer*', *squeeze*: *bool* = *False*, *proc_func*: *Callable* = *None*) → *pandas.core.frame.DataFrame*
fetch data from underlying data source

Parameters

- **selector** (*Union*[*pd.Timestamp*, *slice*, *str*]) – describe how to select data by index.
- **level** (*Union*[*str*, *int*]) – which index level to select the data.
- **col_set** (*str*) – select a set of meaningful columns.(e.g. features, columns).
- **data_key** (*str*) – the data to fetch: *DK_**.
- **proc_func** (*Callable*) – please refer to the doc of *DataHandler.fetch*

Returns

Return type *pd.DataFrame*

get_cols (*col_set*='*__all*', *data_key*: *str* = '*infer*') → *list*
get the column names

Parameters

- **col_set** (*str*) – select a set of meaningful columns.(e.g. features, columns).
- **data_key** (*str*) – the data to fetch: *DK_**.

Returns *list* of column names

Return type list

classmethod **cast** (*handler*: *qlib.data.dataset.handler.DataHandlerLP*) → *qlib.data.dataset.handler.DataHandlerLP*

Motivation - A user create a datahandler in his customized package. Then he want to share the processed handler to other users without introduce the package dependency and complicated data processing logic.
- This class make it possible by casting the class to `DataHandlerLP` and only keep the processed data

Parameters **handler** (`DataHandlerLP`) – A subclass of `DataHandlerLP`

Returns the converted processed data

Return type `DataHandlerLP`

If users want to load features and labels by config, users can define a new handler and call the static method `parse_config_to_fields` of `qlib.contrib.data.handler.Alpha158`.

Also, users can pass `qlib.contrib.data.processor.ConfigSectionProcessor` that provides some preprocess methods for features defined by config into the new handler.

Processor

The `Processor` module in `Qlib` is designed to be learnable and it is responsible for handling data processing such as *normalization* and *drop none/nan features/labels*.

`Qlib` provides the following `Processors`:

- `DropnaProcessor`: *processor* that drops N/A features.
- `DropnaLabel`: *processor* that drops N/A labels.
- `TanhProcess`: *processor* that uses *tanh* to process noise data.
- `ProcessInf`: *processor* that handles infinity values, it will be replaces by the mean of the column.
- `Fillna`: *processor* that handles N/A values, which will fill the N/A value by 0 or other given number.
- `MinMaxNorm`: *processor* that applies min-max normalization.
- `ZscoreNorm`: *processor* that applies z-score normalization.
- `RobustZScoreNorm`: *processor* that applies robust z-score normalization.
- `CSZScoreNorm`: *processor* that applies cross sectional z-score normalization.
- `CSRankNorm`: *processor* that applies cross sectional rank normalization.
- `CSZFillna`: *processor* that fills N/A values in a cross sectional way by the mean of the column.

Users can also create their own *processor* by inheriting the base class of `Processor`. Please refer to the implementation of all the processors for more information ([Processor Link](#)).

To know more about `Processor`, please refer to [Processor API](#).

Example

`Data Handler` can be run with `qrun` by modifying the configuration file, and can also be used as a single module.

Know more about how to run `Data Handler` with `qrun`, please refer to [Workflow: Workflow Management](#)

`Qlib` provides implemented data handler *Alpha158*. The following example shows how to run *Alpha158* as a single module.

Note: Users need to initialize Qlib with `qlib.init` first, please refer to [initialization](#).

```
import qlib
from qlib.contrib.data.handler import Alpha158

data_handler_config = {
    "start_time": "2008-01-01",
    "end_time": "2020-08-01",
    "fit_start_time": "2008-01-01",
    "fit_end_time": "2014-12-31",
    "instruments": "csi300",
}

if __name__ == "__main__":
    qlib.init()
    h = Alpha158(**data_handler_config)

    # get all the columns of the data
    print(h.get_cols())

    # fetch all the labels
    print(h.fetch(col_set="label"))

    # fetch all the features
    print(h.fetch(col_set="feature"))
```

Note: In the Alpha158, Qlib uses the label $Ref(\$close, -2)/Ref(\$close, -1) - 1$ that means the change from T+1 to T+2, rather than $Ref(\$close, -1)/\$close - 1$, of which the reason is that when getting the T day close price of a china stock, the stock can be bought on T+1 day and sold on T+2 day.

API

To know more about Data Handler, please refer to [Data Handler API](#).

1.8.6 Dataset

The Dataset module in Qlib aims to prepare data for model training and inferencing.

The motivation of this module is that we want to maximize the flexibility of of different models to handle data that are suitable for themselves. This module gives the model the flexibility to process their data in an unique way. For instance, models such as GBDT may work well on data that contains *nan* or *None* value, while neural networks such as MLP will break down on such data.

If user's model need process its data in a different way, user could implement his own Dataset class. If the model's data processing is not special, DatasetH can be used directly.

The DatasetH class is the *dataset* with *Data Handler*. Here is the most important interface of the class:

```
class qlib.data.dataset.__init__.DatasetH(handler: Union[Dict[KT, VT],
    qlib.data.dataset.handler.DataHandler], segments: Dict[str, Tuple], fetch_kwargs: Dict[KT, VT] = {}, **kwargs)

    Dataset with Data(H)andler
```

User should try to put the data preprocessing functions into handler. Only following data processing functions should be placed in Dataset:

- The processing is related to specific model.
- The processing is related to data split.

__init__ (*handler: Union[Dict[KT, VT], qlib.data.dataset.handler.DataHandler]*, *segments: Dict[str, Tuple]*, *fetch_kwargs: Dict[KT, VT] = {}*, ***kwargs*)
Setup the underlying data.

Parameters

- **handler** (*Union[dict, DataHandler]*) – handler could be:
 - instance of *DataHandler*
 - config of *DataHandler*. Please refer to *DataHandler*
- **segments** (*dict*) – Describe the options to segment the data. Here are some examples:

config (*handler_kwargs: dict = None*, ***kwargs*)
Initialize the DatasetH

Parameters

- **handler_kwargs** (*dict*) – Config of DataHandler, which could include the following arguments:
 - arguments of *DataHandler.conf_data*, such as ‘instruments’, ‘start_time’ and ‘end_time’.
- **kwargs** (*dict*) – Config of DatasetH, such as
 - **segments** [*dict*] Config of segments which is same as ‘segments’ in self.__init__

setup_data (*handler_kwargs: dict = None*, ***kwargs*)
Setup the Data

Parameters handler_kwargs (*dict*) – init arguments of DataHandler, which could include the following arguments:

- **init_type** : Init Type of Handler
- **enable_cache** : whether to enable cache

prepare (*segments: Union[List[str], Tuple[str], str, slice, pandas.core.indexes.base.Index]*, *col_set='__all'*, *data_key='infer'*, ***kwargs*) → Union[List[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame]
Prepare the data for learning and inference.

Parameters

- **segments** (*Union[List[Text], Tuple[Text], Text, slice]*) – Describe the scope of the data to be prepared Here are some examples:
 - ‘train’
 - [‘train’, ‘valid’]
- **col_set** (*str*) – The col_set will be passed to self.handler when fetching data. TODO: make it automatic:
 - select DK_I for test data
 - select DK_L for training data.

- **data_key** (*str*) – The data to fetch: DK_* Default is DK_I, which indicate fetching data for **inference**.
- **kwargs** –

The parameters that kwargs may contain:

flt_col [*str*] It only exists in TSDatasetH, can be used to add a column of data(True or False) to filter data. This parameter is only supported when it is an instance of TSDatasetH.

Returns

Return type Union[List[pd.DataFrame], pd.DataFrame]

Raises NotImplementedError:

API

To know more about `Dataset`, please refer to [Dataset API](#).

1.8.7 Cache

Cache is an optional module that helps accelerate providing data by saving some frequently-used data as cache file. QLib provides a *Memcache* class to cache the most-frequently-used data in memory, an inheritable *ExpressionCache* class, and an inheritable *DatasetCache* class.

Global Memory Cache

Memcache is a global memory cache mechanism that composes of three *MemCacheUnit* instances to cache **Calendar**, **Instruments**, and **Features**. The *MemCache* is defined globally in *cache.py* as *H*. Users can use *H['c']*, *H['i']*, *H['f']* to get/set *memcache*.

```
class qlib.data.cache.MemCacheUnit(*args, **kwargs)
    Memory Cache Unit.

    __init__(*args, **kwargs)
        Initialize self. See help(type(self)) for accurate signature.

    limited
        whether memory cache is limited

class qlib.data.cache.MemCache(mem_cache_size_limit=None, limit_type='length')
    Memory cache.

    __init__(mem_cache_size_limit=None, limit_type='length')
```

Parameters

- **mem_cache_size_limit** (*cache max size.*) –
- **limit_type** (*length or sizeof; length(call fun: len), size(call fun: sys.getsizeof)*) –

ExpressionCache

ExpressionCache is a cache mechanism that saves expressions such as **Mean(\$close, 5)**. Users can inherit this base class to define their own cache mechanism that saves expressions according to the following steps.

- Override *self._uri* method to define how the cache file path is generated
- Override *self._expression* method to define what data will be cached and how to cache it.

The following shows the details about the interfaces:

class `qlib.data.cache.ExpressionCache(provider)`

Expression cache mechanism base class.

This class is used to wrap expression provider with self-defined expression cache mechanism.

Note: Override the *_uri* and *_expression* method to create your own expression cache mechanism.

expression (*instrument, field, start_time, end_time, freq*)

Get expression data.

Note: Same interface as *expression* method in expression provider

update (*cache_uri: Union[str, pathlib.Path], freq: str = 'day'*)

Update expression cache to latest calendar.

Override this method to define how to update expression cache corresponding to users' own cache mechanism.

Parameters

- **cache_uri** (*str or Path*) – the complete uri of expression cache file (include dir path).
- **freq** (*str*) –

Returns 0(successful update)/ 1(no need to update)/ 2(update failure).

Return type int

Qlib has currently provided implemented disk cache *DiskExpressionCache* which inherits from *ExpressionCache*. The expressions data will be stored in the disk.

DatasetCache

DatasetCache is a cache mechanism that saves datasets. A certain dataset is regulated by a stock pool configuration (or a series of instruments, though not recommended), a list of expressions or static feature fields, the start time, and end time for the collected features and the frequency. Users can inherit this base class to define their own cache mechanism that saves datasets according to the following steps.

- Override *self._uri* method to define how their cache file path is generated
- Override *self._expression* method to define what data will be cached and how to cache it.

The following shows the details about the interfaces:

class `qlib.data.cache.DatasetCache(provider)`

Dataset cache mechanism base class.

This class is used to wrap dataset provider with self-defined dataset cache mechanism.

Note: Override the `_uri` and `_dataset` method to create your own dataset cache mechanism.

dataset (*instruments*, *fields*, *start_time=None*, *end_time=None*, *freq='day'*, *disk_cache=1*, *inst_processors=[]*)
Get feature dataset.

Note: Same interface as *dataset* method in dataset provider

Note: The server use `redis_lock` to make sure read-write conflicts will not be triggered but client readers are not considered.

update (*cache_uri: Union[str, pathlib.Path]*, *freq: str = 'day'*)
Update dataset cache to latest calendar.

Override this method to define how to update dataset cache corresponding to users' own cache mechanism.

Parameters

- **cache_uri** (*str or Path*) – the complete uri of dataset cache file (include dir path).
- **freq** (*str*) –

Returns 0(successful update)/ 1(no need to update)/ 2(update failure)

Return type int

static cache_to_origin_data (*data*, *fields*)
cache data to origin data

Parameters

- **data** – `pd.DataFrame`, cache data.
- **fields** – feature fields.

Returns `pd.DataFrame`.

static normalize_uri_args (*instruments*, *fields*, *freq*)
normalize uri args

QLib has currently provided implemented disk cache *DiskDatasetCache* which inherits from *DatasetCache* . The datasets' data will be stored in the disk.

1.8.8 Data and Cache File Structure

We've specially designed a file structure to manage data and cache, please refer to the [File storage design](#) section in [QLib paper](#) for detailed information. The file structure of data and cache is listed as follows.

```
- data/
  [raw data] updated by data providers
  - calendars/
    - day.txt
  - instruments/
    - all.txt
```

(continues on next page)

(continued from previous page)

```
- csi500.txt
- ...
- features/
  - sh600000/
    - open.day.bin
    - close.day.bin
    - ...
  - ...
[cached data] updated when raw data is updated
- calculated features/
  - sh600000/
    - [hash(instrtument, field_expression, freq)]
      - all-time expression -cache data file
      - .meta : an assorted meta file recording the instrument name, field_
↪name, freq, and visit times
    - ...
  - cache/
    - [hash(stockpool_config, field_expression_list, freq)]
      - all-time Dataset-cache data file
      - .meta : an assorted meta file recording the stockpool config, field_
↪names and visit times
    - .index : an assorted index file recording the line index of all_
↪calendars
    - ...
```

1.9 Forecast Model: Model Training & Prediction

1.9.1 Introduction

Forecast Model is designed to make the *prediction score* about stocks. Users can use the Forecast Model in an automatic workflow by qrun, please refer to [Workflow: Workflow Management](#).

Because the components in Qlib are designed in a loosely-coupled way, Forecast Model can be used as an independent module also.

1.9.2 Base Class & Interface

Qlib provides a base class `qlib.model.base.Model` from which all models should inherit.

The base class provides the following interfaces:

class `qlib.model.base.Model`

Learnable Models

fit (*dataset: qlib.data.dataset.Dataset, reweighter: qlib.data.dataset.weight.Reweighter*)

Learn model from the base model

Note: The attribute names of learned model should *not* start with ‘_’. So that the model could be dumped to disk.

The following code example shows how to retrieve *x_train*, *y_train* and *w_train* from the *dataset*:

```

# get features and labels
df_train, df_valid = dataset.prepare(
    ["train", "valid"], col_set=["feature", "label"], data_
    ↪key=DataHandlerLP.DK_L
)
x_train, y_train = df_train["feature"], df_train["label"]
x_valid, y_valid = df_valid["feature"], df_valid["label"]

# get weights
try:
    wdf_train, wdf_valid = dataset.prepare(["train", "valid"], col_
    ↪set=["weight"],
                                         data_key=DataHandlerLP.DK_
    ↪L)
    w_train, w_valid = wdf_train["weight"], wdf_valid["weight"]
except KeyError as e:
    w_train = pd.DataFrame(np.ones_like(y_train.values), index=y_
    ↪train.index)
    w_valid = pd.DataFrame(np.ones_like(y_valid.values), index=y_
    ↪valid.index)

```

Parameters **dataset** (*Dataset*) – dataset will generate the processed data from model training.

predict (*dataset: qlib.data.dataset.Dataset, segment: Union[str, slice] = 'test'*) → object
give prediction given Dataset

Parameters

- **dataset** (*Dataset*) – dataset will generate the processed dataset from model training.
- **segment** (*Text or slice*) – dataset will use this segment to prepare data. (default=test)

Returns

Return type Prediction results with certain type such as *pandas.Series*.

Qlib also provides a base class `qlib.model.base.ModelFT`, which includes the method for finetuning the model.

For other interfaces such as *finetune*, please refer to [Model API](#).

1.9.3 Example

Qlib's *Model Zoo* includes models such as LightGBM, MLP, LSTM, etc.. These models are treated as the baselines of Forecast Model. The following steps show how to run “LightGBM” as an independent module.

- Initialize Qlib with *qlib.init* first, please refer to [Initialization](#).
- Run the following code to get the *prediction score pred_score*

```

from qlib.contrib.model.gbdt import LGBModel
from qlib.contrib.data.handler import Alpha158
from qlib.utils import init_instance_by_config, flatten_dict
from qlib.workflow import R
from qlib.workflow.record_temp import SignalRecord, PortAnaRecord

```

(continues on next page)

(continued from previous page)

```

market = "csi300"
benchmark = "SH000300"

data_handler_config = {
    "start_time": "2008-01-01",
    "end_time": "2020-08-01",
    "fit_start_time": "2008-01-01",
    "fit_end_time": "2014-12-31",
    "instruments": market,
}

task = {
    "model": {
        "class": "LGBModel",
        "module_path": "qlib.contrib.model.gbd",
        "kwargs": {
            "loss": "mse",
            "colsample_bytree": 0.8879,
            "learning_rate": 0.0421,
            "subsample": 0.8789,
            "lambda_1": 205.6999,
            "lambda_2": 580.9768,
            "max_depth": 8,
            "num_leaves": 210,
            "num_threads": 20,
        },
    },
    "dataset": {
        "class": "DatasetH",
        "module_path": "qlib.data.dataset",
        "kwargs": {
            "handler": {
                "class": "Alpha158",
                "module_path": "qlib.contrib.data.handler",
                "kwargs": data_handler_config,
            },
            "segments": {
                "train": ("2008-01-01", "2014-12-31"),
                "valid": ("2015-01-01", "2016-12-31"),
                "test": ("2017-01-01", "2020-08-01"),
            },
        },
    },
}

# model initiaiton
model = init_instance_by_config(task["model"])
dataset = init_instance_by_config(task["dataset"])

# start exp
with R.start(experiment_name="workflow"):
    # train
    R.log_params(**flatten_dict(task))
    model.fit(dataset)

    # prediction
    recorder = R.get_recorder()

```

(continues on next page)

(continued from previous page)

```
sr = SignalRecord(model, dataset, recorder)
sr.generate()
```

Note: *Alpha158* is the data handler provided by Qlib, please refer to [Data Handler](#). *SignalRecord* is the *Record Template* in Qlib, please refer to [Workflow](#).

Also, the above example has been given in `examples/train_backtest_analyze.ipynb`. Technically, the meaning of the model prediction depends on the label setting designed by user. By default, the meaning of the score is normally the rating of the instruments by the forecasting model. The higher the score, the more profit the instruments.

1.9.4 Custom Model

Qlib supports custom models. If users are interested in customizing their own models and integrating the models into Qlib, please refer to [Custom Model Integration](#).

1.9.5 API

Please refer to [Model API](#).

1.10 Portfolio Strategy: Portfolio Management

1.10.1 Introduction

`Portfolio Strategy` is designed to adopt different portfolio strategies, which means that users can adopt different algorithms to generate investment portfolios based on the prediction scores of the `Forecast Model`. Users can use the `Portfolio Strategy` in an automatic workflow by `Workflow` module, please refer to [Workflow: Workflow Management](#).

Because the components in Qlib are designed in a loosely-coupled way, `Portfolio Strategy` can be used as an independent module also.

Qlib provides several implemented portfolio strategies. Also, Qlib supports custom strategy, users can customize strategies according to their own requirements.

After users specifying the models(forecasting signals) and strategies, running backtest will help users to check the performance of a custom model(forecasting signals)/strategy.

1.10.2 Base Class & Interface

BaseStrategy

Qlib provides a base class `qlib.strategy.base.BaseStrategy`. All strategy classes need to inherit the base class and implement its interface.

- **`generate_trade_decision`** `generate_trade_decision` is a key interface that generates trade decisions in each trading bar. The frequency to call this method depends on the executor frequency(“time_per_step”=“day” by default). But the trading frequency can be decided by users’ implementation. For example, if the user wants to trading in weekly while the `time_per_step` is “day” in executor, user can return non-empty `TradeDecision` weekly(otherwise return empty like [this](#)).

Users can inherit *BaseStrategy* to customize their strategy class.

WeightStrategyBase

QLib also provides a class `qlib.contrib.strategy.WeightStrategyBase` that is a subclass of *BaseStrategy*.

WeightStrategyBase only focuses on the target positions, and automatically generates an order list based on positions. It provides the *generate_target_weight_position* interface.

- ***generate_target_weight_position***
 - According to the current position and trading date to generate the target position. The cash is not considered in the output weight distribution.
 - Return the target position.

Note: Here the *target position* means the target percentage of total assets.

WeightStrategyBase implements the interface *generate_order_list*, whose processions is as follows.

- Call *generate_target_weight_position* method to generate the target position.
- Generate the target amount of stocks from the target position.
- Generate the order list from the target amount

Users can inherit *WeightStrategyBase* and implement the interface *generate_target_weight_position* to customize their strategy class, which only focuses on the target positions.

1.10.3 Implemented Strategy

QLib provides a implemented strategy classes named *TopkDropoutStrategy*.

TopkDropoutStrategy

TopkDropoutStrategy is a subclass of *BaseStrategy* and implement the interface *generate_order_list* whose process is as follows.

- Adopt the Topk-Drop algorithm to calculate the target amount of each stock

Note: There are two parameters for the Topk-Drop algorithm

- *Topk*: The number of stocks held
- *Drop*: The number of stocks sold on each trading day

In general, the number of stocks currently held is *Topk*, with the exception of being zero at the beginning period of trading. For each trading day, let d be the number of the instruments currently held and with a rank K when ranked by the prediction scores from high to low. Then d number of stocks currently held with the worst *prediction score* will be sold, and the same number of unheld stocks with the best *prediction score* will be bought.

In general, $d = \text{'Drop'}$, especially when the pool of the candidate instruments is large, K is large, and *Drop* is small.

In most cases, `TopkDrop` algorithm sells and buys *Drop* stocks every trading day, which yields a turnover rate of $2 \times \text{'Drop'}/\$K\$$.

The following images illustrate a typical scenario. .. image:: ../_static/img/topk_drop.png

alt Topk-Drop

- Generate the order list from the target amount

EnhancedIndexingStrategy

EnhancedIndexingStrategy Enhanced indexing combines the arts of active management and passive management, with the aim of outperforming a benchmark index (e.g., S&P 500) in terms of portfolio return while controlling the risk exposure (a.k.a. tracking error).

For more information, please refer to `qlib.contrib.strategy.signal_strategy.EnhancedIndexingStrategy` and `qlib.contrib.strategy.optimizer.enhanced_indexing.EnhancedIndexingOptimizer`.

1.10.4 Usage & Example

First, user can create a model to get trading signals(the variable name is `pred_score` in following cases).

Prediction Score

The *prediction score* is a pandas DataFrame. Its index is `<datetime(pd.Timestamp), instrument(str)>` and it must contains a *score* column.

A prediction sample is shown as follows.

datetime	instrument	score
2019-01-04	SH600000	-0.505488
2019-01-04	SZ002531	-0.320391
2019-01-04	SZ000999	0.583808
2019-01-04	SZ300569	0.819628
2019-01-04	SZ001696	-0.137140
...		...
2019-04-30	SZ000996	-1.027618
2019-04-30	SH603127	0.225677
2019-04-30	SH603126	0.462443
2019-04-30	SH603133	-0.302460
2019-04-30	SZ300760	-0.126383

`Forecast Model` module can make predictions, please refer to [Forecast Model: Model Training & Prediction](#).

Normally, the prediction score is the output of the models. But some models are learned from a label with a different scale. So the scale of the prediction score may be different from your expectation(e.g. the return of instruments).

QLib didn't add a step to scale the prediction score to a unified scale due to the following reasons. - Because not every trading strategy cares about the scale(e.g. `TopkDropoutStrategy` only cares about the order). So the strategy is responsible for rescaling the prediction score(e.g. some portfolio-optimization-based strategies may require a meaningful scale). - The model has the flexibility to define the target, loss, and data processing. So we don't think there is a silver bullet to rescale it back directly barely based on the model's outputs. If you want to scale it back to some meaningful values(e.g. stock returns.), an intuitive solution is to create a regression model for the model's recent outputs and your recent target values.

Running backtest

- In most cases, users could backtest their portfolio management strategy with `backtest_daily`.

```
from pprint import pprint

import qlib
import pandas as pd
from qlib.utils.time import Freq
from qlib.utils import flatten_dict
from qlib.contrib.evaluate import backtest_daily
from qlib.contrib.evaluate import risk_analysis
from qlib.contrib.strategy import TopkDropoutStrategy

# init qlib
qlib.init(provider_uri=<qlib data dir>)

CSI300_BENCH = "SH000300"
STRATEGY_CONFIG = {
    "topk": 50,
    "n_drop": 5,
    # pred_score, pd.Series
    "signal": pred_score,
}

strategy_obj = TopkDropoutStrategy(**STRATEGY_CONFIG)
report_normal, positions_normal = backtest_daily(
    start_time="2017-01-01", end_time="2020-08-01", strategy=strategy_obj
)
analysis = dict()
# default frequency will be daily (i.e. "day")
analysis["excess_return_without_cost"] = risk_analysis(report_normal[
    ↪ "return"] - report_normal["bench"])
analysis["excess_return_with_cost"] = risk_analysis(report_normal["return
    ↪ "] - report_normal["bench"] - report_normal["cost"])

analysis_df = pd.concat(analysis) # type: pd.DataFrame
pprint(analysis_df)
```

- If users would like to control their strategies in a more detailed(e.g. users have a more advanced version of executor), user could follow this example.

```
from pprint import pprint

import qlib
import pandas as pd
from qlib.utils.time import Freq
from qlib.utils import flatten_dict
from qlib.backtest import backtest, executor
from qlib.contrib.evaluate import risk_analysis
from qlib.contrib.strategy import TopkDropoutStrategy

# init qlib
qlib.init(provider_uri=<qlib data dir>)

CSI300_BENCH = "SH000300"
# Benchmark is for calculating the excess return of your strategy.
```

(continues on next page)

(continued from previous page)

```

# Its data format will be like **ONE normal instrument**.
# For example, you can query its data with the code below
# `D.features(["SH000300"], ["$close"], start_time='2010-01-01', end_
↪time='2017-12-31', freq='day')`
# It is different from the argument `market`, which indicates a universe_
↪of stocks (e.g. **A SET** of stocks like csi300)
# For example, you can query all data from a stock market with the code_
↪below.
# `D.features(D.instruments(market='csi300'), ["$close"], start_time=
↪'2010-01-01', end_time='2017-12-31', freq='day')`

FREQ = "day"
STRATEGY_CONFIG = {
    "topk": 50,
    "n_drop": 5,
    # pred_score, pd.Series
    "signal": pred_score,
}

EXECUTOR_CONFIG = {
    "time_per_step": "day",
    "generate_portfolio_metrics": True,
}

backtest_config = {
    "start_time": "2017-01-01",
    "end_time": "2020-08-01",
    "account": 100000000,
    "benchmark": CSI300_BENCH,
    "exchange_kwargs": {
        "freq": FREQ,
        "limit_threshold": 0.095,
        "deal_price": "close",
        "open_cost": 0.0005,
        "close_cost": 0.0015,
        "min_cost": 5,
    },
}

# strategy object
strategy_obj = TopkDropoutStrategy(**STRATEGY_CONFIG)
# executor object
executor_obj = executor.SimulatorExecutor(**EXECUTOR_CONFIG)
# backtest
portfolio_metric_dict, indicator_dict = backtest(executor=executor_obj,
↪strategy=strategy_obj, **backtest_config)
analysis_freq = "{0}{1}".format(*Freq.parse(FREQ))
# backtest info
report_normal, positions_normal = portfolio_metric_dict.get(analysis_
↪freq)

# analysis
analysis = dict()
analysis["excess_return_without_cost"] = risk_analysis(
    report_normal["return"] - report_normal["bench"], freq=analysis_freq
)
analysis["excess_return_with_cost"] = risk_analysis(

```

(continues on next page)

(continued from previous page)

```

    report_normal["return"] - report_normal["bench"] - report_normal[
↪ "cost"], freq=analysis_freq
)

analysis_df = pd.concat(analysis) # type: pd.DataFrame
# log metrics
analysis_dict = flatten_dict(analysis_df["risk"].unstack().T.to_dict())
# print out results
pprint(f"The following are analysis results of benchmark return(
↪ {analysis_freq}).")
pprint(risk_analysis(report_normal["bench"], freq=analysis_freq))
pprint(f"The following are analysis results of the excess return without_
↪ cost ({analysis_freq}).")
pprint(analysis["excess_return_without_cost"])
pprint(f"The following are analysis results of the excess return with_
↪ cost ({analysis_freq}).")
pprint(analysis["excess_return_with_cost"])

```

Result

The backtest results are in the following form:

		risk
excess_return_without_cost	mean	0.000605
	std	0.005481
	annualized_return	0.152373
	information_ratio	1.751319
	max_drawdown	-0.059055
excess_return_with_cost	mean	0.000410
	std	0.005478
	annualized_return	0.103265
	information_ratio	1.187411
	max_drawdown	-0.075024

- ***excess_return_without_cost***

- ***mean*** Mean value of the *CAR* (cumulative abnormal return) without cost
- ***std*** The *Standard Deviation* of *CAR* (cumulative abnormal return) without cost.
- ***annualized_return*** The *Annualized Rate* of *CAR* (cumulative abnormal return) without cost.
- ***information_ratio*** The *Information Ratio* without cost. please refer to [Information Ratio – IR](#).
- ***max_drawdown*** The *Maximum Drawdown* of *CAR* (cumulative abnormal return) without cost, please refer to [Maximum Drawdown \(MDD\)](#).

- ***excess_return_with_cost***

- ***mean*** Mean value of the *CAR* (cumulative abnormal return) series with cost
- ***std*** The *Standard Deviation* of *CAR* (cumulative abnormal return) series with cost.
- ***annualized_return*** The *Annualized Rate* of *CAR* (cumulative abnormal return) with cost.
- ***information_ratio*** The *Information Ratio* with cost. please refer to [Information Ratio – IR](#).
- ***max_drawdown*** The *Maximum Drawdown* of *CAR* (cumulative abnormal return) with cost, please refer to [Maximum Drawdown \(MDD\)](#).

1.10.5 Reference

To know more about the *prediction score* `pred_score` output by `Forecast Model`, please refer to [Forecast Model: Model Training & Prediction](#).

1.11 Design of Nested Decision Execution Framework for High-Frequency Trading

1.11.1 Introduction

Daily trading (e.g. portfolio management) and intraday trading (e.g. orders execution) are two hot topics in Quant investment and usually studied separately.

To get the joint trading performance of daily and intraday trading, they must interact with each other and run backtest jointly. In order to support the joint backtest strategies in multiple levels, a corresponding framework is required. None of the publicly available high-frequency trading frameworks considers multi-level joint trading, which make the backtesting aforementioned inaccurate.

Besides backtesting, the optimization of strategies from different levels is not standalone and can be affected by each other. For example, the best portfolio management strategy may change with the performance of order executions(e.g. a portfolio with higher turnover may becomes a better choice when we improve the order execution strategies). To achieve the overall good performance , it is necessary to consider the interaction of strategies in different level.

Therefore, building a new framework for trading in multiple levels becomes necessary to solve the various problems mentioned above, for which we designed a nested decision execution framework that consider the interaction of strategies.

The design of the framework is shown in the yellow part in the middle of the figure above. Each level consists of `Trading Agent` and `Execution Env`. `Trading Agent` has its own data processing module (`Information Extractor`), forecasting module (`Forecast Model`) and decision generator (`Decision Generator`). The trading algorithm generates the decisions by the `Decision Generator` based on the forecast signals output by the `Forecast Module`, and the decisions generated by the trading algorithm are passed to the `Execution Env`, which returns the execution results.

The frequency of trading algorithm, decision content and execution environment can be customized by users (e.g. intraday trading, daily-frequency trading, weekly-frequency trading), and the execution environment can be nested with finer-grained trading algorithm and execution environment inside (i.e. sub-workflow in the figure, e.g. daily-frequency orders can be turned into finer-grained decisions by splitting orders within the day). The flexibility of nested decision execution framework makes it easy for users to explore the effects of combining different levels of trading strategies and break down the optimization barriers between different levels of trading algorithm.

1.11.2 Example

An example of nested decision execution framework for high-frequency can be found [here](#).

Besides, the above examples, here are some other related work about high-frequency trading in QLib.

- [Prediction with high-frequency data](#)
- [Examples](#) to extract features form high-frequency data without fixed frequency.
- [A paper](#) for high-frequency trading.

1.12 Meta Controller: Meta-Task & Meta-Dataset & Meta-Model

1.12.1 Introduction

`Meta Controller` provides guidance to `Forecast Model`, which aims to learn regular patterns among a series of forecasting tasks and use learned patterns to guide forthcoming forecasting tasks. Users can implement their own meta-model instance based on `Meta Controller` module.

1.12.2 Meta Task

A *Meta Task* instance is the basic element in the meta-learning framework. It saves the data that can be used for the *Meta Model*. Multiple *Meta Task* instances may share the same *Data Handler*, controlled by *Meta Dataset*. Users should use `prepare_task_data()` to obtain the data that can be directly fed into the *Meta Model*.

class `qlib.model.meta.task.MetaTask` (*task*: dict, *meta_info*: object, *mode*: str = 'full')

A single meta-task, a meta-dataset contains a list of them. It serves as a component as in `MetaDatasetDS`

The data processing is different - the processed input may be different between training and testing

- **When training, the X, y, X_test, y_test in training tasks are necessary (# PROC_MODE_FULL #)**
but not necessary in test tasks. (# PROC_MODE_TEST #)
- When the meta model can be transferred into other dataset, only `meta_info` is necessary (# PROC_MODE_TRANSFER #)

__init__ (*task*: dict, *meta_info*: object, *mode*: str = 'full')

The `__init__` func is responsible for - store the task - store the origin input data for - process the input data for meta data

Parameters

- **task** (*dict*) – the task to be enhanced by meta model
- **meta_info** (*object*) – the input for meta model

get_meta_input () → object

Return the **processed** `meta_info`

1.12.3 Meta Dataset

Meta Dataset controls the meta-information generating process. It is on the duty of providing data for training the *Meta Model*. Users should use `prepare_tasks` to retrieve a list of *Meta Task* instances.

class `qlib.model.meta.dataset.MetaTaskDataset` (*segments*: Union[Dict[str, Tuple], float],
*args, **kwargs)

A dataset fetching the data in a meta-level.

A Meta Dataset is responsible for - input tasks(e.g. Qlib tasks) and prepare meta tasks

- meta task contains more information than normal tasks (e.g. input data for meta model)

The learnt pattern could transfer to other meta dataset. The following cases should be supported - A meta-model trained on meta-dataset A and then applied to meta-dataset B

- Some pattern are shared between meta-dataset A and B, so meta-input on meta-dataset A are used when meta model are applied on meta-dataset-B

__init__ (*segments: Union[Dict[str, Tuple], float], *args, **kwargs*)
 The meta-dataset maintains a list of meta-tasks when it is initialized.

The segments indicates the way to divide the data

The duty of the `__init__` function of `MetaTaskDataset` - initialize the tasks

prepare_tasks (*segments: Union[List[str], str], *args, **kwargs*) →
 List[`qlib.model.meta.task.MetaTask`]
 Prepare the data in each meta-task and ready for training.

The following code example shows how to retrieve a list of meta-tasks from the *meta_dataset*:

```
# get the train segment and the test segment, both of them are lists
train_meta_tasks, test_meta_tasks = meta_dataset.prepare_tasks([
    ↪ "train", "test"])
```

Parameters *segments* (`Union[List[Text], Tuple[Text], Text]`) – the info to select data

Returns A list of the prepared data of each meta-task for training the meta-model. For multiple segments [seg1, seg2, ... , segN], the returned list will be [[tasks in seg1], [tasks in seg2], ... , [tasks in segN]]. Each task is a meta task

Return type list

1.12.4 Meta Model

General Meta Model

Meta Model instance is the part that controls the workflow. The usage of the *Meta Model* includes: 1. Users train their *Meta Model* with the *fit* function. 2. The *Meta Model* instance guides the workflow by giving useful information via the *inference* function.

class `qlib.model.meta.model.MetaModel`

The meta-model guiding the model learning.

The word *Guiding* can be categorized into two types based on the stage of model learning - The definition of learning tasks: Please refer to docs of *MetaTaskModel* - Controlling the learning process of models: Please refer to the docs of *MetaGuideModel*

fit (**args, **kwargs*)
 The training process of the meta-model.

inference (**args, **kwargs*) → object
 The inference process of the meta-model.

Returns Some information to guide the model learning

Return type object

Meta Task Model

This type of meta-model may interact with task definitions directly. Then, the *Meta Task Model* is the class for them to inherit from. They guide the base tasks by modifying the base task definitions. The function *prepare_tasks* can be used to obtain the modified base task definitions.

class `qlib.model.meta.model.MetaTaskModel`

This type of meta-model deals with base task definitions. The meta-model creates tasks for training new base forecasting models after it is trained. *prepare_tasks* directly modifies the task definitions.

fit (*meta_dataset: qlib.model.meta.dataset.MetaTaskDataset*)

The MetaTaskModel is expected to get prepared MetaTask from meta_dataset. And then it will learn knowledge from the meta tasks

inference (*meta_dataset: qlib.model.meta.dataset.MetaTaskDataset*) → List[dict]

MetaTaskModel will make inference on the meta_dataset The MetaTaskModel is expected to get prepared MetaTask from meta_dataset. Then it will create modified task with Qlib format which can be executed by Qlib trainer.

Returns A list of modified task definitions.

Return type List[dict]

Meta Guide Model

This type of meta-model participates in the training process of the base forecasting model. The meta-model may guide the base forecasting models during their training to improve their performances.

class `qlib.model.meta.model.MetaGuideModel`

This type of meta-model aims to guide the training process of the base model. The meta-model interacts with the base forecasting models during their training process.

fit (**args, **kwargs*)

The training process of the meta-model.

inference (**args, **kwargs*)

The inference process of the meta-model.

Returns Some information to guide the model learning

Return type object

1.12.5 Example

Qlib provides an implementation of Meta Model module, DDG-DA, which adapts to the market dynamics.

DDG-DA includes four steps:

1. Calculate meta-information and encapsulate it into Meta Task instances. All the meta-tasks form a Meta Dataset instance.
2. Train DDG-DA based on the training data of the meta-dataset.
3. Do the inference of the DDG-DA to get guide information.
4. Apply guide information to the forecasting models to improve their performances.

The [above example](#) can be found in `examples/benchmarks_dynamic/DDG-DA/workflow.py`.

1.13 Qlib Recorder: Experiment Management

1.13.1 Introduction

Qlib contains an experiment management system named `QlibRecorder`, which is designed to help users handle experiment and analyse results in an efficient way.

There are three components of the system:

- ***ExperimentManager*** a class that manages experiments.
- ***Experiment*** a class of experiment, and each instance of it is responsible for a single experiment.
- ***Recorder*** a class of recorder, and each instance of it is responsible for a single run.

Here is a general view of the structure of the system:

This experiment management system defines a set of interface and provided a concrete implementation `MLflowExpManager`, which is based on the machine learning platform: `MLFlow` ([link](#)).

If users set the implementation of `ExpManager` to be `MLflowExpManager`, they can use the command `mlflow ui` to visualize and check the experiment results. For more information, please refer to the related documents [here](#).

1.13.2 Qlib Recorder

`QlibRecorder` provides a high level API for users to use the experiment management system. The interfaces are wrapped in the variable `R` in `Qlib`, and users can directly use `R` to interact with the system. The following command shows how to import `R` in Python:

```
from qlib.workflow import R
```

`QlibRecorder` includes several common API for managing *experiments* and *recorders* within a workflow. For more available APIs, please refer to the following section about *Experiment Manager*, *Experiment* and *Recorder*.

Here are the available interfaces of `QlibRecorder`:

class `qlib.workflow.__init__.QlibRecorder` (*exp_manager: qlib.workflow.expm.ExpManager*)

A global system that helps to manage the experiments.

__init__ (*exp_manager: qlib.workflow.expm.ExpManager*)

Initialize self. See `help(type(self))` for accurate signature.

start (*, *experiment_id: Optional[str] = None, experiment_name: Optional[str] = None, recorder_id: Optional[str] = None, recorder_name: Optional[str] = None, uri: Optional[str] = None, resume: bool = False*)

Method to start an experiment. This method can only be called within a Python's *with* statement. Here is the example code:

```
# start new experiment and recorder
with R.start(experiment_name='test', recorder_name='recorder_1'):
    model.fit(dataset)
    R.log...
    ... # further operations

# resume previous experiment and recorder
with R.start(experiment_name='test', recorder_name='recorder_1',
    ↳ resume=True): # if users want to resume recorder, they have to specify the
    ↳ exact same name for experiment and recorder.
    ... # further operations
```

Parameters

- **experiment_id** (*str*) – id of the experiment one wants to start.
- **experiment_name** (*str*) – name of the experiment one wants to start.
- **recorder_id** (*str*) – id of the recorder under the experiment one wants to start.
- **recorder_name** (*str*) – name of the recorder under the experiment one wants to start.
- **uri** (*str*) – The tracking uri of the experiment, where all the artifacts/metrics etc. will be stored. The default uri is set in the qlib.config. Note that this uri argument will not change the one defined in the config file. Therefore, the next time when users call this function in the same experiment, they have to also specify this argument with the same value. Otherwise, inconsistent uri may occur.
- **resume** (*bool*) – whether to resume the specific recorder with given name under the given experiment.

start_exp (*, *experiment_id=None, experiment_name=None, recorder_id=None, recorder_name=None, uri=None, resume=False*)

Lower level method for starting an experiment. When use this method, one should end the experiment manually and the status of the recorder may not be handled properly. Here is the example code:

```
R.start_exp(experiment_name='test', recorder_name='recorder_1')
... # further operations
R.end_exp('FINISHED') or R.end_exp(Recorder.STATUS_S)
```

Parameters

- **experiment_id** (*str*) – id of the experiment one wants to start.
- **experiment_name** (*str*) – the name of the experiment to be started
- **recorder_id** (*str*) – id of the recorder under the experiment one wants to start.
- **recorder_name** (*str*) – name of the recorder under the experiment one wants to start.
- **uri** (*str*) – the tracking uri of the experiment, where all the artifacts/metrics etc. will be stored. The default uri are set in the qlib.config.
- **resume** (*bool*) – whether to resume the specific recorder with given name under the given experiment.

Returns

Return type An experiment instance being started.

end_exp (*recorder_status='FINISHED'*)

Method for ending an experiment manually. It will end the current active experiment, as well as its active recorder with the specified *status* type. Here is the example code of the method:

```
R.start_exp(experiment_name='test')
... # further operations
R.end_exp('FINISHED') or R.end_exp(Recorder.STATUS_S)
```

Parameters status (*str*) – The status of a recorder, which can be SCHEDULED, RUNNING, FINISHED, FAILED.

search_records (*experiment_ids*, ***kwargs*)

Get a pandas DataFrame of records that fit the search criteria.

The arguments of this function are not set to be rigid, and they will be different with different implementation of ExpManager in QLib. QLib now provides an implementation of ExpManager with mlflow, and here is the example code of the this method with the MLflowExpManager:

```
R.log_metrics(m=2.50, step=0)
records = R.search_runs([experiment_id], order_by=["metrics.m DESC"])
```

Parameters

- **experiment_ids** (*list*) – list of experiment IDs.
- **filter_string** (*str*) – filter query string, defaults to searching all runs.
- **run_view_type** (*int*) – one of enum values ACTIVE_ONLY, DELETED_ONLY, or ALL (e.g. in mlflow.entities.ViewType).
- **max_results** (*int*) – the maximum number of runs to put in the dataframe.
- **order_by** (*list*) – list of columns to order by (e.g., “metrics.rmse”).

Returns

- A *pandas.DataFrame* of records, where each metric, parameter, and tag
- are expanded into their own columns named *metrics.*, *params.**, and *tags.**
- respectively. For records that don’t have a particular metric, parameter, or tag, their
- value will be (NumPy) *Nan*, *None*, or *None* respectively.

list_experiments ()

Method for listing all the existing experiments (except for those being deleted.)

```
exps = R.list_experiments()
```

Returns

Return type A dictionary (name -> experiment) of experiments information that being stored.

list_recorders (*experiment_id=None*, *experiment_name=None*)

Method for listing all the recorders of experiment with given id or name.

If user doesn’t provide the id or name of the experiment, this method will try to retrieve the default experiment and list all the recorders of the default experiment. If the default experiment doesn’t exist, the method will first create the default experiment, and then create a new recorder under it. (More information about the default experiment can be found [here](#)).

Here is the example code:

```
recorders = R.list_recorders(experiment_name='test')
```

Parameters

- **experiment_id** (*str*) – id of the experiment.
- **experiment_name** (*str*) – name of the experiment.

Returns

Return type A dictionary (id -> recorder) of recorder information that being stored.

get_exp (*, experiment_id=None, experiment_name=None, create: bool = True, start: bool = False)
→ qlib.workflow.exp.Experiment

Method for retrieving an experiment with given id or name. Once the *create* argument is set to True, if no valid experiment is found, this method will create one for you. Otherwise, it will only retrieve a specific experiment or raise an Error.

- If ‘create’ is True:
 - If *active experiment* exists:
 - * no id or name specified, return the active experiment.
 - * if id or name is specified, return the specified experiment. If no such exp found, create a new experiment with given id or name.
 - If *active experiment* not exists:
 - * no id or name specified, create a default experiment, and the experiment is set to be active.
 - * if id or name is specified, return the specified experiment. If no such exp found, create a new experiment with given name or the default experiment.
- Else If ‘create’ is False:
 - If ‘active experiment’ exists:
 - * no id or name specified, return the active experiment.
 - * if id or name is specified, return the specified experiment. If no such exp found, raise Error.
 - If *active experiment* not exists:
 - * no id or name specified. If the default experiment exists, return it, otherwise, raise Error.
 - * if id or name is specified, return the specified experiment. If no such exp found, raise Error.

Here are some use cases:

```
# Case 1
with R.start('test'):
    exp = R.get_exp()
    recorders = exp.list_recorders()

# Case 2
with R.start('test'):
    exp = R.get_exp(experiment_name='test1')

# Case 3
exp = R.get_exp() -> a default experiment.

# Case 4
exp = R.get_exp(experiment_name='test')

# Case 5
exp = R.get_exp(create=False) -> the default experiment if exists.
```

Parameters

- **experiment_id** (*str*) – id of the experiment.
- **experiment_name** (*str*) – name of the experiment.

- **create** (*boolean*) – an argument determines whether the method will automatically create a new experiment according to user’s specification if the experiment hasn’t been created before.

- **start** (*bool*) –

when start is True, if the experiment has not started(not activated), it will start

It is designed for R.log_params to auto start experiments

Returns

Return type An experiment instance with given id or name.

delete_exp (*experiment_id=None, experiment_name=None*)

Method for deleting the experiment with given id or name. At least one of id or name must be given, otherwise, error will occur.

Here is the example code:

```
R.delete_exp(experiment_name='test')
```

Parameters

- **experiment_id** (*str*) – id of the experiment.
- **experiment_name** (*str*) – name of the experiment.

get_uri ()

Method for retrieving the uri of current experiment manager.

Here is the example code:

```
uri = R.get_uri()
```

Returns

Return type The uri of current experiment manager.

set_uri (*uri: Optional[str]*)

Method to reset the current uri of current experiment manager.

NOTE: - When the uri is refer to a file path, please using the absolute path instead of strings like “~/ml-runs/”

The backend don’t support strings like this.

uri_context (*uri: str*)

Temporarily set the exp_manager’s **default_uri** to uri

NOTE: - Please refer to the NOTE in the *set_uri*

Parameters **uri** (*Text*) – the temporal uri

get_recorder (*, *recorder_id=None, recorder_name=None, experiment_id=None, experiment_name=None*) → `qlib.workflow.recorder.Recorder`

Method for retrieving a recorder.

- If *active recorder* exists:
 - no id or name specified, return the active recorder.
 - if id or name is specified, return the specified recorder.

- If *active recorder* not exists:
 - no id or name specified, raise Error.
 - if id or name is specified, and the corresponding *experiment_name* must be given, return the specified recorder. Otherwise, raise Error.

The recorder can be used for further process such as *save_object*, *load_object*, *log_params*, *log_metrics*, etc.

Here are some use cases:

```
# Case 1
with R.start(experiment_name='test'):
    recorder = R.get_recorder()

# Case 2
with R.start(experiment_name='test'):
    recorder = R.get_recorder(recorder_id='2e7a4efd66574fa49039e00ffaefa99d')

# Case 3
recorder = R.get_recorder() -> Error

# Case 4
recorder = R.get_recorder(recorder_id='2e7a4efd66574fa49039e00ffaefa99d') ->
↳Error

# Case 5
recorder = R.get_recorder(recorder_id='2e7a4efd66574fa49039e00ffaefa99d',
↳experiment_name='test')
```

Here are some things users may concern - Q: What recorder will it return if multiple recorder meets the query (e.g. query with *experiment_name*) - A: If mlflow backend is used, then the recorder with the latest *start_time* will be returned. Because MLflow's *search_runs* function guarantee it

Parameters

- **recorder_id** (*str*) – id of the recorder.
- **recorder_name** (*str*) – name of the recorder.
- **experiment_name** (*str*) – name of the experiment.

Returns

Return type A recorder instance.

delete_recorder (*recorder_id=None, recorder_name=None*)

Method for deleting the recorders with given id or name. At least one of id or name must be given, otherwise, error will occur.

Here is the example code:

```
R.delete_recorder(recorder_id='2e7a4efd66574fa49039e00ffaefa99d')
```

Parameters

- **recorder_id** (*str*) – id of the experiment.
- **recorder_name** (*str*) – name of the experiment.

save_objects (*local_path=None, artifact_path=None, **kwargs*)

Method for saving objects as artifacts in the experiment to the uri. It supports either saving from a local file/directory, or directly saving objects. User can use valid python's keywords arguments to specify the object to be saved as well as its name (name: value).

In summary, this API is designs for saving **objects to the experiments management backend path**, 1. Qlib provide two methods to specify **objects** - Passing in the object directly by passing with ****kwargs** (e.g. `R.save_objects(trained_model=model)`) - Passing in the local path to the object, i.e. *local_path* parameter. 2. *artifact_path* represents the **the experiments management backend path**

- If *active recorder* exists: it will save the objects through the active recorder.
- If *active recorder* not exists: the system will create a default experiment, and a new recorder and save objects under it.

Note: If one wants to save objects with a specific recorder. It is recommended to first get the specific recorder through *get_recorder* API and use the recorder the save objects. The supported arguments are the same as this method.

Here are some use cases:

```
# Case 1
with R.start(experiment_name='test'):
    pred = model.predict(dataset)
    R.save_objects(**{"pred.pkl": pred}, artifact_path='prediction')
    rid = R.get_recorder().id
...
R.get_recorder(recorder_id=rid).load_object("prediction/pred.pkl") # after_
→saving objects, you can load the previous object with this api

# Case 2
with R.start(experiment_name='test'):
    R.save_objects(local_path='results/pred.pkl', artifact_path="prediction")
    rid = R.get_recorder().id
...
R.get_recorder(recorder_id=rid).load_object("prediction/pred.pkl") # after_
→saving objects, you can load the previous object with this api
```

Parameters

- **local_path** (*str*) – if provided, them save the file or directory to the artifact URI.
- **artifact_path** (*str*) – the relative path for the artifact to be stored in the URI.
- ****kwargs** (*Dict[Text, Any]*) – the object to be saved. For example, `{"pred.pkl": pred}`

load_object (*name: str*)

Method for loading an object from artifacts in the experiment in the uri.

log_params (***kwargs*)

Method for logging parameters during an experiment. In addition to using *R*, one can also log to a specific recorder after getting it with *get_recorder* API.

- If *active recorder* exists: it will log parameters through the active recorder.
- If *active recorder* not exists: the system will create a default experiment as well as a new recorder, and log parameters under it.

Here are some use cases:

```
# Case 1
with R.start('test'):
    R.log_params(learning_rate=0.01)

# Case 2
R.log_params(learning_rate=0.01)
```

Parameters *argument* (*keyword*) – name1=value1, name2=value2, ...

log_metrics (*step=None, **kwargs*)

Method for logging metrics during an experiment. In addition to using R, one can also log to a specific recorder after getting it with *get_recorder* API.

- If *active recorder* exists: it will log metrics through the active recorder.
- If *active recorder* not exists: the system will create a default experiment as well as a new recorder, and log metrics under it.

Here are some use cases:

```
# Case 1
with R.start('test'):
    R.log_metrics(train_loss=0.33, step=1)

# Case 2
R.log_metrics(train_loss=0.33, step=1)
```

Parameters *argument* (*keyword*) – name1=value1, name2=value2, ...

set_tags (***kwargs*)

Method for setting tags for a recorder. In addition to using R, one can also set the tag to a specific recorder after getting it with *get_recorder* API.

- If *active recorder* exists: it will set tags through the active recorder.
- If *active recorder* not exists: the system will create a default experiment as well as a new recorder, and set the tags under it.

Here are some use cases:

```
# Case 1
with R.start('test'):
    R.set_tags(release_version="2.2.0")

# Case 2
R.set_tags(release_version="2.2.0")
```

Parameters *argument* (*keyword*) – name1=value1, name2=value2, ...

1.13.3 Experiment Manager

The *ExpManager* module in Qlib is responsible for managing different experiments. Most of the APIs of *ExpManager* are similar to *QlibRecorder*, and the most important API will be the *get_exp* method. User can directly refer to the documents above for some detailed information about how to use the *get_exp* method.

class qlib.workflow.expm.**ExpManager** (*uri: str, default_exp_name: Optional[str]*)

This is the *ExpManager* class for managing experiments. The API is designed similar to mlflow. (The link: https://mlflow.org/docs/latest/python_api/mlflow.html)

__init__ (*uri: str, default_exp_name: Optional[str]*)

Initialize self. See help(type(self)) for accurate signature.

start_exp (*, *experiment_id: Optional[str] = None, experiment_name: Optional[str] = None, recorder_id: Optional[str] = None, recorder_name: Optional[str] = None, uri: Optional[str] = None, resume: bool = False, **kwargs*)

Start an experiment. This method includes first get_or_create an experiment, and then set it to be active.

Parameters

- **experiment_id** (*str*) – id of the active experiment.
- **experiment_name** (*str*) – name of the active experiment.
- **recorder_id** (*str*) – id of the recorder to be started.
- **recorder_name** (*str*) – name of the recorder to be started.
- **uri** (*str*) – the current tracking URI.
- **resume** (*boolean*) – whether to resume the experiment and recorder.

Returns

Return type An active experiment.

end_exp (*recorder_status: str = 'SCHEDULED', **kwargs*)

End an active experiment.

Parameters

- **experiment_name** (*str*) – name of the active experiment.
- **recorder_status** (*str*) – the status of the active recorder of the experiment.

create_exp (*experiment_name: Optional[str] = None*)

Create an experiment.

Parameters **experiment_name** (*str*) – the experiment name, which must be unique.

Returns

- *An experiment object.*
- *Raise*
- *—*
- *ExpAlreadyExistError*

search_records (*experiment_ids=None, **kwargs*)

Get a pandas DataFrame of records that fit the search criteria of the experiment. Inputs are the search criteria user want to apply.

Returns

- *A pandas.DataFrame of records, where each metric, parameter, and tag*
- *are expanded into their own columns named metrics., params.*, and tags.***
- *respectively. For records that don't have a particular metric, parameter, or tag, their*
- *value will be (NumPy) Nan, None, or None respectively.*

get_exp (*, *experiment_id=None, experiment_name=None, create: bool = True, start: bool = False*)

Retrieve an experiment. This method includes getting an active experiment, and get_or_create a specific experiment.

When user specify experiment id and name, the method will try to return the specific experiment. When user does not provide recorder id or name, the method will try to return the current active experiment. The *create* argument determines whether the method will automatically create a new experiment according to user's specification if the experiment hasn't been created before.

- If *create* is True:
 - If *active experiment* exists:
 - * no id or name specified, return the active experiment.
 - * if id or name is specified, return the specified experiment. If no such exp found, create a new experiment with given id or name. If *start* is set to be True, the experiment is set to be active.
 - If *active experiment* not exists:
 - * no id or name specified, create a default experiment.
 - * if id or name is specified, return the specified experiment. If no such exp found, create a new experiment with given id or name. If *start* is set to be True, the experiment is set to be active.
- Else If *create* is False:
 - If *active experiment* exists:
 - * no id or name specified, return the active experiment.
 - * if id or name is specified, return the specified experiment. If no such exp found, raise Error.
 - If *active experiment* not exists:
 - * no id or name specified. If the default experiment exists, return it, otherwise, raise Error.
 - * if id or name is specified, return the specified experiment. If no such exp found, raise Error.

Parameters

- **experiment_id** (*str*) – id of the experiment to return.
- **experiment_name** (*str*) – name of the experiment to return.
- **create** (*boolean*) – create the experiment if it hasn't been created before.
- **start** (*boolean*) – start the new experiment if one is created.

Returns

Return type An experiment object.

delete_exp (*experiment_id=None, experiment_name=None*)

Delete an experiment.

Parameters

- **experiment_id** (*str*) – the experiment id.
- **experiment_name** (*str*) – the experiment name.

default_uri

Get the default tracking URI from qlib.config.C

uri

Get the default tracking URI or current URI.

Returns

Return type The tracking URI string.

set_uri (*uri: Optional[str] = None*)

Set the current tracking URI and the corresponding variables.

Parameters **uri** (*str*) –

list_experiments ()

List all the existing experiments.

Returns

Return type A dictionary (name -> experiment) of experiments information that being stored.

For other interfaces such as *create_exp*, *delete_exp*, please refer to [Experiment Manager API](#).

1.13.4 Experiment

The `Experiment` class is solely responsible for a single experiment, and it will handle any operations that are related to an experiment. Basic methods such as *start*, *end* an experiment are included. Besides, methods related to *recorders* are also available: such methods include *get_recorder* and *list_recorders*.

class `qlib.workflow.exp.Experiment` (*id, name*)

This is the *Experiment* class for each experiment being run. The API is designed similar to mlflow. (The link: https://mlflow.org/docs/latest/python_api/mlflow.html)

__init__ (*id, name*)

Initialize self. See `help(type(self))` for accurate signature.

start (*, *recorder_id=None, recorder_name=None, resume=False*)

Start the experiment and set it to be active. This method will also start a new recorder.

Parameters

- **recorder_id** (*str*) – the id of the recorder to be created.
- **recorder_name** (*str*) – the name of the recorder to be created.
- **resume** (*bool*) – whether to resume the first recorder

Returns

Return type An active recorder.

end (*recorder_status='SCHEDULED'*)

End the experiment.

Parameters **recorder_status** (*str*) – the status the recorder to be set with when ending (SCHEDULED, RUNNING, FINISHED, FAILED).

create_recorder (*recorder_name=None*)

Create a recorder for each experiment.

Parameters **recorder_name** (*str*) – the name of the recorder to be created.

Returns

Return type A recorder object.

search_records (***kwargs*)

Get a pandas DataFrame of records that fit the search criteria of the experiment. Inputs are the search criteria user want to apply.

Returns

- A *pandas.DataFrame* of records, where each metric, parameter, and tag
- are expanded into their own columns named *metrics.*, *params.**, and *tags.**
- respectively. For records that don't have a particular metric, parameter, or tag, their
- value will be (NumPy) *Nan*, *None*, or *None* respectively.

delete_recorder (*recorder_id*)

Create a recorder for each experiment.

Parameters **recorder_id** (*str*) – the id of the recorder to be deleted.

get_recorder (*recorder_id=None*, *recorder_name=None*, *create: bool = True*, *start: bool = False*)

Retrieve a Recorder for user. When user specify recorder id and name, the method will try to return the specific recorder. When user does not provide recorder id or name, the method will try to return the current active recorder. The *create* argument determines whether the method will automatically create a new recorder according to user's specification if the recorder hasn't been created before.

- If *create* is True:
 - If *active recorder* exists:
 - * no id or name specified, return the active recorder.
 - * if id or name is specified, return the specified recorder. If no such exp found, create a new recorder with given id or name. If *start* is set to be True, the recorder is set to be active.
 - If *active recorder* not exists:
 - * no id or name specified, create a new recorder.
 - * if id or name is specified, return the specified experiment. If no such exp found, create a new recorder with given id or name. If *start* is set to be True, the recorder is set to be active.
- Else If *create* is False:
 - If *active recorder* exists:
 - * no id or name specified, return the active recorder.
 - * if id or name is specified, return the specified recorder. If no such exp found, raise Error.
 - If *active recorder* not exists:
 - * no id or name specified, raise Error.
 - * if id or name is specified, return the specified recorder. If no such exp found, raise Error.

Parameters

- **recorder_id** (*str*) – the id of the recorder to be deleted.
- **recorder_name** (*str*) – the name of the recorder to be deleted.
- **create** (*boolean*) – create the recorder if it hasn't been created before.
- **start** (*boolean*) – start the new recorder if one is **created**.

Returns

Return type A recorder object.

list_recorders (*rtype: str = 'dict', **flt_kwargs*) → Union[List[qlib.workflow.recorder.Recorder], Dict[str, qlib.workflow.recorder.Recorder]]

List all the existing recorders of this experiment. Please first get the experiment instance before calling this method. If user want to use the method *R.list_recorders()*, please refer to the related API document in *QlibRecorder*.

flt_kwargs [dict] filter recorders by conditions e.g. list_recorders(status=Recorder.STATUS_FI)

Returns

if rtype == "dict": A dictionary (id -> recorder) of recorder information that being stored.

elif rtype == "list": A list of Recorder.

Return type The return type depent on *rtype*

For other interfaces such as *search_records*, *delete_recorder*, please refer to [Experiment API](#).

Qlib also provides a default *Experiment*, which will be created and used under certain situations when users use the APIs such as *log_metrics* or *get_exp*. If the default *Experiment* is used, there will be related logged information when running Qlib. Users are able to change the name of the default *Experiment* in the config file of Qlib or during Qlib's [initialization](#), which is set to be '*Experiment*'.

1.13.5 Recorder

The *Recorder* class is responsible for a single recorder. It will handle some detailed operations such as *log_metrics*, *log_params* of a single run. It is designed to help user to easily track results and things being generated during a run.

Here are some important APIs that are not included in the *QlibRecorder*:

class qlib.workflow.recorder.**Recorder** (*experiment_id, name*)

This is the *Recorder* class for logging the experiments. The API is designed similar to mlflow. (The link: https://mlflow.org/docs/latest/python_api/mlflow.html)

The status of the recorder can be SCHEDULED, RUNNING, FINISHED, FAILED.

__init__ (*experiment_id, name*)

Initialize self. See help(type(self)) for accurate signature.

save_objects (*local_path=None, artifact_path=None, **kwargs*)

Save objects such as prediction file or model checkpoints to the artifact URI. User can save object through keywords arguments (name:value).

Please refer to the docs of qlib.workflow:R.save_objects

Parameters

- **local_path** (*str*) – if provided, them save the file or directory to the artifact URI.
- **artifact_path=None** (*str*) – the relative path for the artifact to be stored in the URI.

load_object (*name*)

Load objects such as prediction file or model checkpoints.

Parameters **name** (*str*) – name of the file to be loaded.

Returns

Return type The saved object.

start_run()

Start running or resuming the Recorder. The return value can be used as a context manager within a *with* block; otherwise, you must call `end_run()` to terminate the current run. (See *ActiveRun* class in *mlflow*)

Returns

Return type An active running object (e.g. *mlflow.ActiveRun* object)

end_run()

End an active Recorder.

log_params(kwargs)**

Log a batch of params for the current run.

Parameters arguments (*keyword*) – key, value pair to be logged as parameters.

log_metrics(step=None, **kwargs)

Log multiple metrics for the current run.

Parameters arguments (*keyword*) – key, value pair to be logged as metrics.

set_tags(kwargs)**

Log a batch of tags for the current run.

Parameters arguments (*keyword*) – key, value pair to be logged as tags.

delete_tags(*keys)

Delete some tags from a run.

Parameters keys (*series of strs of the keys*) – all the name of the tag to be deleted.

list_artifacts(artifact_path: str = None)

List all the artifacts of a recorder.

Parameters artifact_path (*str*) – the relative path for the artifact to be stored in the URI.

Returns

Return type A list of artifacts information (name, path, etc.) that being stored.

list_metrics()

List all the metrics of a recorder.

Returns

Return type A dictionary of metrics that being stored.

list_params()

List all the params of a recorder.

Returns

Return type A dictionary of params that being stored.

list_tags()

List all the tags of a recorder.

Returns

Return type A dictionary of tags that being stored.

For other interfaces such as *save_objects*, *load_object*, please refer to [Recorder API](#).

1.13.6 Record Template

The `RecordTemp` class is a class that enables generate experiment results such as IC and backtest in a certain format. We have provided three different *Record Template* class:

- `SignalRecord`: This class generates the *prediction* results of the model.
- `SigAnaRecord`: This class generates the *IC*, *ICIR*, *Rank IC* and *Rank ICIR* of the model.

Here is a simple example of what is done in `SigAnaRecord`, which users can refer to if they want to calculate IC, Rank IC, Long-Short Return with their own prediction and label.

```
from qlib.contrib.eva.alpha import calc_ic, calc_long_short_return

ic, ric = calc_ic(pred.iloc[:, 0], label.iloc[:, 0])
long_short_r, long_avg_r = calc_long_short_return(pred.iloc[:, 0], label.iloc[:, 0])
```

- `PortAnaRecord`: This class generates the results of *backtest*. The detailed information about *backtest* as well as the available *strategy*, users can refer to [Strategy](#) and [Backtest](#).

Here is a simple example of what is done in `PortAnaRecord`, which users can refer to if they want to do backtest based on their own prediction and label.

```
from qlib.contrib.strategy.strategy import TopkDropoutStrategy
from qlib.contrib.evaluate import (
    backtest as normal_backtest,
    risk_analysis,
)

# backtest
STRATEGY_CONFIG = {
    "topk": 50,
    "n_drop": 5,
}
BACKTEST_CONFIG = {
    "limit_threshold": 0.095,
    "account": 100000000,
    "benchmark": BENCHMARK,
    "deal_price": "close",
    "open_cost": 0.0005,
    "close_cost": 0.0015,
    "min_cost": 5,
}

strategy = TopkDropoutStrategy(**STRATEGY_CONFIG)
report_normal, positions_normal = normal_backtest(pred_score, strategy=strategy,
↪ **BACKTEST_CONFIG)

# analysis
analysis = dict()
analysis["excess_return_without_cost"] = risk_analysis(report_normal["return"] -
↪ report_normal["bench"])
analysis["excess_return_with_cost"] = risk_analysis(report_normal["return"] - report_
↪ normal["bench"] - report_normal["cost"])
analysis_df = pd.concat(analysis) # type: pd.DataFrame
print(analysis_df)
```

For more information about the APIs, please refer to [Record Template API](#).

1.13.7 Known Limitations

- The Python objects are saved based on pickle, which may results in issues when the environment dumping objects and loading objects are different.

1.14 Analysis: Evaluation & Results Analysis

1.14.1 Introduction

Analysis is designed to show the graphical reports of Intraday Trading , which helps users to evaluate and analyse investment portfolios visually. The following are some graphics to view:

- **analysis_position**
 - report_graph
 - score_ic_graph
 - cumulative_return_graph
 - risk_analysis_graph
 - rank_label_graph
- **analysis_model**
 - model_performance_graph

All of the accumulated profit metrics(e.g. return, max drawdown) in QLib are calculated by summation. This avoids the metrics or the plots being skewed exponentially over time.

1.14.2 Graphical Reports

Users can run the following code to get all supported reports.

```
>> import qlib.contrib.report as qcr
>> print(qcr.GRAPH_NAME_LIST)
['analysis_position.report_graph', 'analysis_position.score_ic_graph', 'analysis_
↪ position.cumulative_return_graph', 'analysis_position.risk_analysis_graph',
↪ 'analysis_position.rank_label_graph', 'analysis_model.model_performance_graph']
```

Note: For more details, please refer to the function document: `similar to help(qcr.analysis_position.report_graph)`

1.14.3 Usage & Example

Usage of *analysis_position.report*

API

`qlib.contrib.report.analysis_position.report.report_graph`(*report_df*: *pandas.core.frame.DataFrame*,
show_notebook: *bool = True*) → [*<class 'list'>*,
<class 'tuple'>]

display backtest report

Example:

```
import qlib
import pandas as pd
from qlib.utils.time import Freq
from qlib.utils import flatten_dict
from qlib.backtest import backtest, executor
from qlib.contrib.evaluate import risk_analysis
from qlib.contrib.strategy import TopkDropoutStrategy

# init qlib
qlib.init(provider_uri=<qlib data dir>)

CSI300_BENCH = "SH000300"
FREQ = "day"
STRATEGY_CONFIG = {
    "topk": 50,
    "n_drop": 5,
    # pred_score, pd.Series
    "signal": pred_score,
}

EXECUTOR_CONFIG = {
    "time_per_step": "day",
    "generate_portfolio_metrics": True,
}

backtest_config = {
    "start_time": "2017-01-01",
    "end_time": "2020-08-01",
    "account": 100000000,
    "benchmark": CSI300_BENCH,
    "exchange_kwargs": {
        "freq": FREQ,
        "limit_threshold": 0.095,
        "deal_price": "close",
        "open_cost": 0.0005,
        "close_cost": 0.0015,
        "min_cost": 5,
    },
}

# strategy object
strategy_obj = TopkDropoutStrategy(**STRATEGY_CONFIG)
# executor object
executor_obj = executor.SimulatorExecutor(**EXECUTOR_CONFIG)
# backtest
portfolio_metric_dict, indicator_dict = _
↳ backtest(executor=executor_obj, strategy=strategy_obj, _
↳ **backtest_config) (continues on next page)
```

(continued from previous page)

```
analysis_freq = "{0}{1}".format(*Freq.parse(FREQ))
# backtest info
report_normal_df, positions_normal = portfolio_metric_dict.
    ↳get(analysis_freq)

qcr.analysis_position.report_graph(report_normal_df)
```

Parameters

- **report_df** – **df.index.name** must be **date**, **df.columns** must contain **return**, **turnover**, **cost**, **bench**.

	return	cost	bench	turnover
date				
2017-01-04	0.003421	0.000864	0.011693	0.576325
2017-01-05	0.000508	0.000447	0.000721	0.227882
2017-01-06	-0.003321	0.000212	-0.004322	0.102765
2017-01-09	0.006753	0.000212	0.006874	0.105864
2017-01-10	-0.000416	0.000440	-0.003350	0.208396

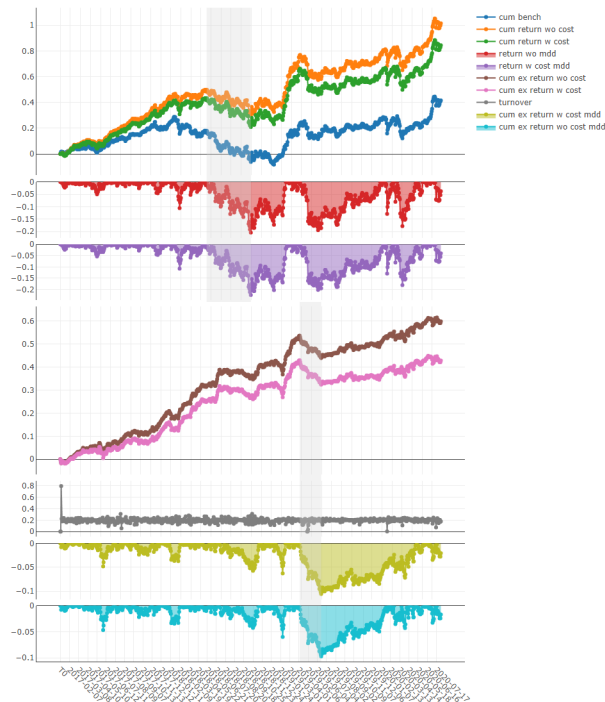
- **show_notebook** – whether to display graphics in notebook, the default is **True**.

Returns if **show_notebook** is **True**, display in notebook; else return **plotly.graph_objs.Figure** list.

Graphical Result

Note:

- Axis X: Trading day
- Axis Y:
 - **cum bench** Cumulative returns series of benchmark
 - **cum return wo cost** Cumulative returns series of portfolio without cost
 - **cum return w cost** Cumulative returns series of portfolio with cost
 - **return wo mdd** Maximum drawdown series of cumulative return without cost
 - **return w cost mdd:** Maximum drawdown series of cumulative return with cost
 - **cum ex return wo cost** The *CAR* (cumulative abnormal return) series of the portfolio compared to the benchmark without cost.
 - **cum ex return w cost** The *CAR* (cumulative abnormal return) series of the portfolio compared to the benchmark with cost.
 - **turnover** Turnover rate series
 - **cum ex return wo cost mdd** Drawdown series of *CAR* (cumulative abnormal return) without cost
 - **cum ex return w cost mdd** Drawdown series of *CAR* (cumulative abnormal return) with cost
- The shaded part above: Maximum drawdown corresponding to *cum return wo cost*
- The shaded part below: Maximum drawdown corresponding to *cum ex return wo cost*



Usage of `analysis_position.score_ic`

API

`qlib.contrib.report.analysis_position.score_ic.score_ic_graph`(*pred_label*: *pandas.core.frame.DataFrame*,
show_notebook: *bool = True*) →
 [<class 'list'>,
 <class 'tuple'>]

score IC

Example:

```
from qlib.data import D
from qlib.contrib.report import analysis_position
pred_df_dates = pred_df.index.get_level_values(level='datetime')
features_df = D.features(D.instruments('csi500'), ['Ref($close, -2)/Ref($close, -1)-1'], pred_df_dates.min(), pred_df_dates.
    ↪max())
features_df.columns = ['label']
pred_label = pd.concat([features_df, pred], axis=1, sort=True).
    ↪reindex(features_df.index)
analysis_position.score_ic_graph(pred_label)
```

Parameters

- **pred_label** – index is `pd.MultiIndex`, index name is `[instrument, datetime]`; columns names is `[score, label]`.

instrument	datetime	score	label
SH600004	2017-12-11	-0.013502	-0.013502
	2017-12-12	-0.072367	-0.072367
	2017-12-13	-0.068605	-0.068605
	2017-12-14	0.012440	0.012440
	2017-12-15	-0.102778	-0.102778

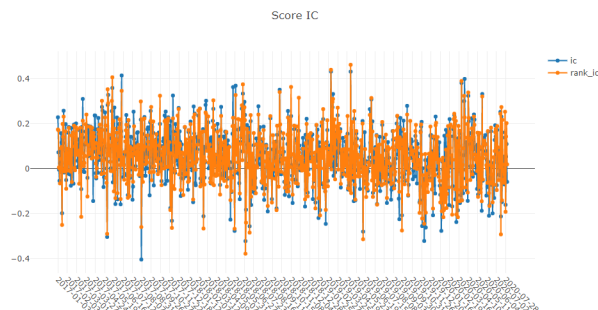
- **show_notebook** – whether to display graphics in notebook, the default is **True**.

Returns if show_notebook is True, display in notebook; else return **plotly.graph_objs.Figure** list.

Graphical Result

Note:

- **Axis X:** Trading day
- **Axis Y:**
 - **ic** The *Pearson correlation coefficient* series between *label* and *prediction score*. In the above example, the *label* is formulated as $Ref(\$close, -1)/\$close - 1$. Please refer to [Data Feature](#) for more details.
 - **rank_ic** The *Spearman's rank correlation coefficient* series between *label* and *prediction score*.



Usage of `analysis_position.risk_analysis`

API

```

qlib.contrib.report.analysis_position.risk_analysis.risk_analysis_graph(analysis_df:
    pandas.core.frame.DataFrame
    =
    None,
    re-
    port_normal_df:
    pandas.core.frame.DataFrame
    =
    None,
    re-
    port_long_short_df:
    pandas.core.frame.DataFrame
    =
    None,
    show_notebook:
    bool
    =
    True)
→
Iter-
able[plotly.graph_objs._figu

```

Generate analysis graph and monthly analysis

Example:

```

import qlib
import pandas as pd
from qlib.utils.time import Freq
from qlib.utils import flatten_dict
from qlib.backtest import backtest, executor
from qlib.contrib.evaluate import risk_analysis
from qlib.contrib.strategy import TopkDropoutStrategy

# init qlib
qlib.init(provider_uri=<qlib data dir>)

CSI300_BENCH = "SH000300"
FREQ = "day"
STRATEGY_CONFIG = {
    "topk": 50,
    "n_drop": 5,
    # pred_score, pd.Series
    "signal": pred_score,
}

EXECUTOR_CONFIG = {
    "time_per_step": "day",
    "generate_portfolio_metrics": True,
}

```

(continues on next page)

(continued from previous page)

```

backtest_config = {
    "start_time": "2017-01-01",
    "end_time": "2020-08-01",
    "account": 100000000,
    "benchmark": CSI300_BENCH,
    "exchange_kwargs": {
        "freq": FREQ,
        "limit_threshold": 0.095,
        "deal_price": "close",
        "open_cost": 0.0005,
        "close_cost": 0.0015,
        "min_cost": 5,
    },
}

# strategy object
strategy_obj = TopkDropoutStrategy(**STRATEGY_CONFIG)
# executor object
executor_obj = executor.SimulatorExecutor(**EXECUTOR_CONFIG)
# backtest
portfolio_metric_dict, indicator_dict = _
↳backtest(executor=executor_obj, strategy=strategy_obj, _
↳**backtest_config)
analysis_freq = "{0}{1}".format(*Freq.parse(FREQ))
# backtest info
report_normal_df, positions_normal = portfolio_metric_dict.
↳get(analysis_freq)
analysis = dict()
analysis["excess_return_without_cost"] = risk_analysis(
    report_normal_df["return"] - report_normal_df["bench"], _
↳freq=analysis_freq
)
analysis["excess_return_with_cost"] = risk_analysis(
    report_normal_df["return"] - report_normal_df["bench"] - _
↳report_normal_df["cost"], freq=analysis_freq
)

analysis_df = pd.concat(analysis) # type: pd.DataFrame
analysis_position.risk_analysis_graph(analysis_df, report_
↳normal_df)

```

Parameters

- **analysis_df** – analysis data, index is **pd.MultiIndex**; columns names is **[risk]**.

		risk
excess_return_without_cost	mean	0.000692
	std	0.005374
	annualized_return	0.174495
	information_ratio	2.045576
excess_return_with_cost	max_drawdown	-0.079103
	mean	0.000499
	std	0.005372
	annualized_return	0.125625
	information_ratio	1.473152
	max_drawdown	-0.088263

- **report_normal_df** – **df.index.name** must be **date**, **df.columns** must contain **return**, **turnover**, **cost**, **bench**.

	return	cost	bench	turnover
date				
2017-01-04	0.003421	0.000864	0.011693	0.576325
2017-01-05	0.000508	0.000447	0.000721	0.227882
2017-01-06	-0.003321	0.000212	-0.004322	0.102765
2017-01-09	0.006753	0.000212	0.006874	0.105864
2017-01-10	-0.000416	0.000440	-0.003350	0.208396

- **report_long_short_df** – **df.index.name** must be **date**, **df.columns** contain **long**, **short**, **long_short**.

	long	short	long_short
date			
2017-01-04	-0.001360	0.001394	0.000034
2017-01-05	0.002456	0.000058	0.002514
2017-01-06	0.000120	0.002739	0.002859
2017-01-09	0.001436	0.001838	0.003273
2017-01-10	0.000824	-0.001944	-0.001120

- **show_notebook** – Whether to display graphics in a notebook, default **True**. If **True**, show graph in notebook. If **False**, return graph figure

Returns

Graphical Result

Note:

- **general graphics**

– *std*

- * **excess_return_without_cost** The *Standard Deviation* of *CAR* (cumulative abnormal return) without cost.
- * **excess_return_with_cost** The *Standard Deviation* of *CAR* (cumulative abnormal return) with cost.

– *annualized_return*

- * **excess_return_without_cost** The *Annualized Rate* of *CAR* (cumulative abnormal return) without cost.
- * **excess_return_with_cost** The *Annualized Rate* of *CAR* (cumulative abnormal return) with cost.

– *information_ratio*

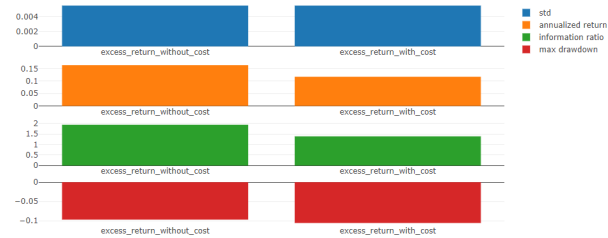
- * **excess_return_without_cost** The *Information Ratio* without cost.
- * **excess_return_with_cost** The *Information Ratio* with cost.

To know more about *Information Ratio*, please refer to [Information Ratio – IR](#).

– *max_drawdown*

- * **excess_return_without_cost** The *Maximum Drawdown* of *CAR* (cumulative abnormal return) without cost.

* ***excess_return_with_cost*** The *Maximum Drawdown* of *CAR* (cumulative abnormal return) with cost.



Note:

- **annualized_return/max_drawdown/information_ratio/std graphics**

- Axis X: Trading days grouped by month
- Axis Y:

- * **annualized_return graphics**

- ***excess_return_without_cost_annualized_return*** The *Annualized Rate* series of monthly *CAR* (cumulative abnormal return) without cost.
 - ***excess_return_with_cost_annualized_return*** The *Annualized Rate* series of monthly *CAR* (cumulative abnormal return) with cost.

- * **max_drawdown graphics**

- ***excess_return_without_cost_max_drawdown*** The *Maximum Drawdown* series of monthly *CAR* (cumulative abnormal return) without cost.
 - ***excess_return_with_cost_max_drawdown*** The *Maximum Drawdown* series of monthly *CAR* (cumulative abnormal return) with cost.

- * **information_ratio graphics**

- ***excess_return_without_cost_information_ratio*** The *Information Ratio* series of monthly *CAR* (cumulative abnormal return) without cost.
 - ***excess_return_with_cost_information_ratio*** The *Information Ratio* series of monthly *CAR* (cumulative abnormal return) with cost.

- * **std graphics**

- ***excess_return_without_cost_max_drawdown*** The *Standard Deviation* series of monthly *CAR* (cumulative abnormal return) without cost.
 - ***excess_return_with_cost_max_drawdown*** The *Standard Deviation* series of monthly *CAR* (cumulative abnormal return) with cost.
-



Usage of `analysis_model.analysis_model_performance`

API

```
qlib.contrib.report.analysis_model.analysis_model_performance.ic_figure(ic_df:
    pandas.core.frame.DataFrame,
    show_nature_day=True,
    **kwargs)
→
plotly.graph_objs._figure.Figure
```

IC figure

Parameters

- **ic_df** – ic DataFrame
- **show_nature_day** – whether to display the abscissa of non-trading day

Returns plotly.graph_objs.Figure

```
qlib.contrib.report.analysis_model.analysis_model_performance.model_performance_graph(pred_label, pan-  
das.co  
lag:  
int  
=  
1,  
N:  
int  
=  
5,  
re-  
verse=  
rank=  
graph_  
list  
=  
['grou  
'pred_  
'pred_  
show_  
bool  
=  
True,  
show_  
→  
[<class  
'list'>,  
<class  
'tu-  
ple'>]
```

Model performance

Parameters **pred_label** – index is **pd.MultiIndex**, index name is **[instrument, datetime]**;
columns names is ****[score,**
label]**. It is usually same as the label of model training(e.g. “Ref(\$close, -2)/Ref(\$close, -1) - 1”).

instrument	datetime	score	label
SH600004	2017-12-11	-0.013502	-0.013502
	2017-12-12	-0.072367	-0.072367
	2017-12-13	-0.068605	-0.068605
	2017-12-14	0.012440	0.012440
	2017-12-15	-0.102778	-0.102778

Parameters

- **lag** – `pred.groupby(level='instrument')['score'].shift(lag)`. It will be only used in the auto-correlation computing.
- **N** – group number, default 5.

- **reverse** – if *True*, $\text{pred}[\text{'score'}] \neq -1$.
- **rank** – if *True*, calculate rank ic.
- **graph_names** – graph names; default ['cumulative_return', 'pred_ic', 'pred_autocorr', 'pred_turnover'].
- **show_notebook** – whether to display graphics in notebook, the default is *True*.
- **show_nature_day** – whether to display the abscissa of non-trading day.

Returns if *show_notebook* is *True*, display in notebook; else return *plotly.graph_objs.Figure* list.

Graphical Results

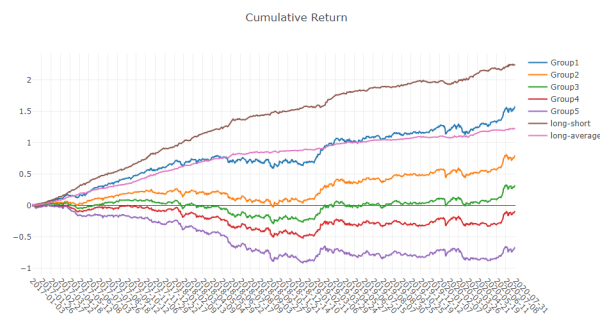
Note:

- **cumulative return graphics**

- **Group1:** The *Cumulative Return* series of stocks group with (*ranking ratio* of label $\leq 20\%$)
- **Group2:** The *Cumulative Return* series of stocks group with ($20\% < \text{ranking ratio}$ of label $\leq 40\%$)
- **Group3:** The *Cumulative Return* series of stocks group with ($40\% < \text{ranking ratio}$ of label $\leq 60\%$)
- **Group4:** The *Cumulative Return* series of stocks group with ($60\% < \text{ranking ratio}$ of label $\leq 80\%$)
- **Group5:** The *Cumulative Return* series of stocks group with ($80\% < \text{ranking ratio}$ of label)
- **long-short:** The Difference series between *Cumulative Return* of *Group1* and of *Group5*
- **long-average** The Difference series between *Cumulative Return* of *Group1* and average *Cumulative Return* for all stocks.

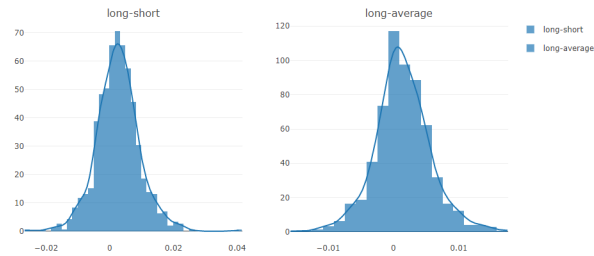
The *ranking ratio* can be formulated as follows.

$$\text{ranking ratio} = \frac{\text{Ascending Ranking of label}}{\text{Number of Stocks in the Portfolio}}$$



Note:

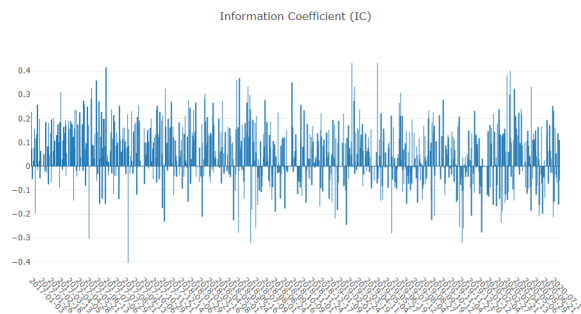
- **long-short/long-average** The distribution of long-short/long-average returns on each trading day



Note:

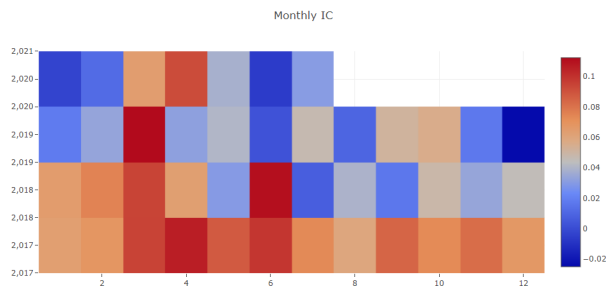
- **Information Coefficient**

- The *Pearson correlation coefficient* series between *labels* and *prediction scores* of stocks in portfolio.
- The graphics reports can be used to evaluate the *prediction scores*.



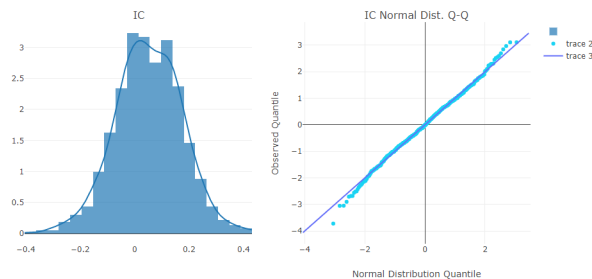
Note:

- **Monthly IC** Monthly average of the *Information Coefficient*



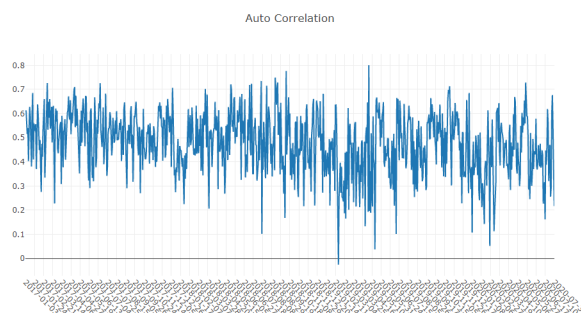
Note:

- **IC** The distribution of the *Information Coefficient* on each trading day.
- **IC Normal Dist. Q-Q** The *Quantile-Quantile Plot* is used for the normal distribution of *Information Coefficient* on each trading day.

**Note:**

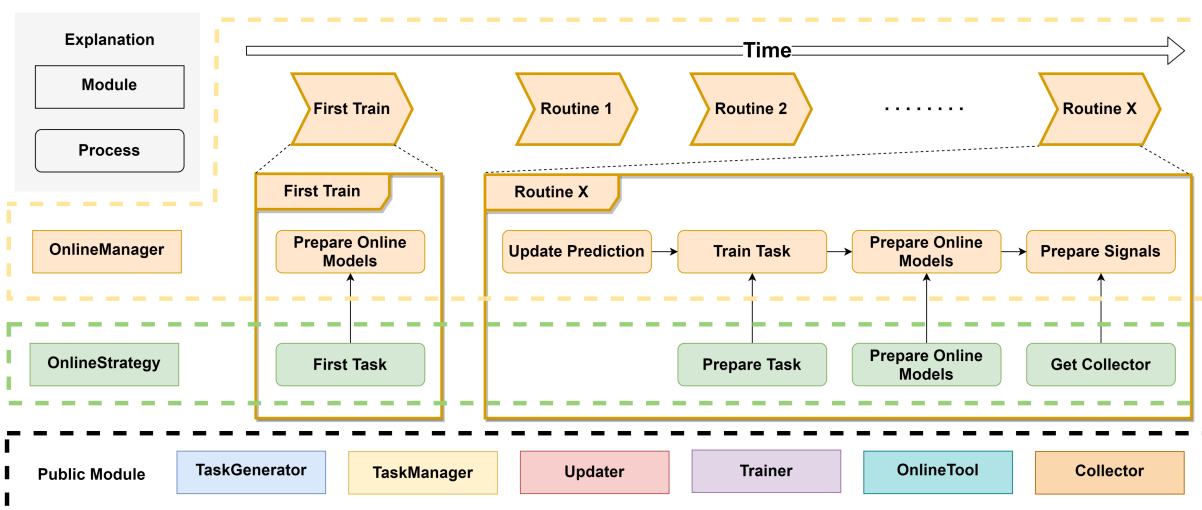
- **Auto Correlation**

- The *Pearson correlation coefficient* series between the latest *prediction scores* and the *prediction scores lag days* ago of stocks in portfolio on each trading day.
- The graphics reports can be used to estimate the turnover rate.



1.15 Online Serving

1.15.1 Introduction



In addition to backtesting, one way to test a model is effective is to make predictions in real market conditions or even do real trading based on those predictions. *Online Serving* is a set of modules for online models using the latest data, which including *Online Manager*, *Online Strategy*, *Online Tool*, *Updater*.

Here are several examples for reference, which demonstrate different features of *Online Serving*. If you have many models or *task* needs to be managed, please consider *Task Management*. The examples are based on some components in *Task Management* such as *TrainerRM* or *Collector*.

NOTE: User should keep his data source updated to support online serving. For example, Qlib provides a batch of scripts to help users update Yahoo daily data.

Known limitations currently - Currently, the daily updating prediction for the next trading day is supported. But generating orders for the next trading day is not supported due to the *limitations of public data* <<https://github.com/microsoft/qlib/issues/215#issuecomment-766293563>>_

1.15.2 Online Manager

OnlineManager can manage a set of *Online Strategy* and run them dynamically.

With the change of time, the decisive models will be also changed. In this module, we call those contributing models *online* models. In every routine(such as every day or every minute), the *online* models may be changed and the prediction of them needs to be updated. So this module provides a series of methods to control this process.

This module also provides a method to simulate *Online Strategy* in history. Which means you can verify your strategy or find a better one.

There are 4 total situations for using different trainers in different situations:

Situations	Description
Online + Trainer	When you want to do a REAL routine, the Trainer will help you train the models. It will train models task by task and strategy by strategy.
Online + Delay-Trainer	DelayTrainer will skip concrete training until all tasks have been prepared by different strategies. It makes users can parallelly train all tasks at the end of <i>routine</i> or <i>first_train</i> . Otherwise, these functions will get stuck when each strategy prepare tasks.
Simulation + Trainer	It will behave in the same way as <i>Online + Trainer</i> . The only difference is that it is for simulation/backtesting instead of online trading
Simulation + Delay-Trainer	When your models don't have any temporal dependence, you can use DelayTrainer for the ability to multitasking. It means all tasks in all routines can be REAL trained at the end of simulating. The signals will be prepared well at different time segments (based on whether or not any new model is online).

Here is some pseudo code the demonstrate the workflow of each situation

For simplicity

- Only one strategy is used in the strategy
- `update_online_pred` is only called in the online mode and is ignored

1) Online + Trainer

```
tasks = first_train()
models = trainer.train(tasks)
trainer.end_train(models)
for day in online_trading_days:
```

(continues on next page)

(continued from previous page)

```
# OnlineManager.routine
models = trainer.train(strategy.prepare_tasks()) # for each strategy
strategy.prepare_online_models(models) # for each strategy

trainer.end_train(models)
prepare_signals() # prepare trading signals daily
```

Online + DelayTrainer: the workflow is the same as *Online + Trainer*.

2) Simulation + DelayTrainer

```
# simulate
tasks = first_train()
models = trainer.train(tasks)
for day in historical_calendars:
    # OnlineManager.routine
    models = trainer.train(strategy.prepare_tasks()) # for each strategy
    strategy.prepare_online_models(models) # for each strategy
# delay_prepare()
# FIXME: Currently the delay_prepare is not implemented in a proper way.
trainer.end_train(<for all previous models>)
prepare_signals()
```

Can we simplify current workflow? - Can reduce the number of state of tasks?

- For each task, we have three phases (i.e. task, partly trained task, final trained task)

```
class qlib.workflow.online.manager.OnlineManager (strategies:
                                                    Union[qlib.workflow.online.strategy.OnlineStrategy,
List[qlib.workflow.online.strategy.OnlineStrategy]],
trainer: qlib.model.trainer.Trainer =
None, begin_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp]
= None, freq='day')
```

`OnlineManager` can manage online models with *Online Strategy*. It also provides a history recording of which models are online at what time.

```
__init__ (strategies: Union[qlib.workflow.online.strategy.OnlineStrategy,
List[qlib.workflow.online.strategy.OnlineStrategy]], trainer: qlib.model.trainer.Trainer
= None, begin_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp] = None,
freq='day')
```

Init `OnlineManager`. One `OnlineManager` must have at least one `OnlineStrategy`.

Parameters

- **strategies** (`Union[OnlineStrategy, List[OnlineStrategy]]`) – an instance of `OnlineStrategy` or a list of `OnlineStrategy`
- **begin_time** (`Union[str, pd.Timestamp]`, optional) – the `OnlineManager` will begin at this time. Defaults to `None` for using the latest date.
- **trainer** (`Trainer`) – the trainer to train task. `None` for using `TrainerR`.
- **freq** (`str`, optional) – data frequency. Defaults to “day”.

```
first_train (strategies: List[qlib.workflow.online.strategy.OnlineStrategy] = None, model_kwargs:
dict = {})
```

Get tasks from every strategy’s `first_tasks` method and train them. If using `DelayTrainer`, it can finish training all together after every strategy’s `first_tasks`.

Parameters

- **strategies** (*List[OnlineStrategy]*) – the strategies list (need this param when adding strategies). None for use default strategies.
- **model_kwargs** (*dict*) – the params for *prepare_online_models*

routine (*cur_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp] = None, task_kwargs: dict = {}, model_kwargs: dict = {}, signal_kwargs: dict = {}*)

Typical update process for every strategy and record the online history.

The typical update process after a routine, such as day by day or month by month. The process is: Update predictions -> Prepare tasks -> Prepare online models -> Prepare signals.

If using DelayTrainer, it can finish training all together after every strategy's prepare_tasks.

Parameters

- **cur_time** (*Union[str, pd.Timestamp], optional*) – run routine method in this time. Defaults to None.
- **task_kwargs** (*dict*) – the params for *prepare_tasks*
- **model_kwargs** (*dict*) – the params for *prepare_online_models*
- **signal_kwargs** (*dict*) – the params for *prepare_signals*

get_collector (***kwargs*) → *qlib.workflow.task.collect.MergeCollector*

Get the instance of *Collector* to collect results from every strategy. This collector can be a basis as the signals preparation.

Parameters ***kwargs* – the params for *get_collector*.

Returns the collector to merge other collectors.

Return type *MergeCollector*

add_strategy (*strategies: Union[qlib.workflow.online.strategy.OnlineStrategy, List[qlib.workflow.online.strategy.OnlineStrategy]]*)

Add some new strategies to OnlineManager.

Parameters **strategy** (*Union[OnlineStrategy, List[OnlineStrategy]]*)
– a list of OnlineStrategy

prepare_signals (*prepare_func: Callable = <qlib.model.ens.ensemble.AverageEnsemble object>, over_write=False*)

After preparing the data of the last routine (a box in box-plot) which means the end of the routine, we can prepare trading signals for the next routine.

NOTE: Given a set prediction, all signals before these prediction end times will be prepared well.

Even if the latest signal already exists, the latest calculation result will be overwritten.

Note: Given a prediction of a certain time, all signals before this time will be prepared well.

Parameters

- **prepare_func** (*Callable, optional*) – Get signals from a dict after collecting. Defaults to *AverageEnsemble()*, the results collected by *MergeCollector* must be {xxx:pred}.
- **over_write** (*bool, optional*) – If True, the new signals will overwrite. If False, the new signals will append to the end of signals. Defaults to False.

Returns the signals.

Return type `pd.DataFrame`

get_signals () → Union[pandas.core.series.Series, pandas.core.frame.DataFrame]

Get prepared online signals.

Returns `pd.Series` for only one signals every datetime. `pd.DataFrame` for multiple signals, for example, buy and sell operations use different trading signals.

Return type Union[`pd.Series`, `pd.DataFrame`]

simulate (*end_time=None, frequency='day', task_kwargs={}, model_kwargs={}, signal_kwargs={}*)
→ Union[pandas.core.series.Series, pandas.core.frame.DataFrame]

Starting from the current time, this method will simulate every routine in `OnlineManager` until the end time.

Considering the parallel training, the models and signals can be prepared after all routine simulating.

The delay training way can be `DelayTrainer` and the delay preparing signals way can be `delay_prepare`.

Parameters

- **end_time** – the time the simulation will end
- **frequency** – the calendar frequency
- **task_kwargs** (*dict*) – the params for *prepare_tasks*
- **model_kwargs** (*dict*) – the params for *prepare_online_models*
- **signal_kwargs** (*dict*) – the params for *prepare_signals*

Returns `pd.Series` for only one signals every datetime. `pd.DataFrame` for multiple signals, for example, buy and sell operations use different trading signals.

Return type Union[`pd.Series`, `pd.DataFrame`]

delay_prepare (*model_kwargs={}, signal_kwargs={}*)

Prepare all models and signals if something is waiting for preparation.

Parameters

- **model_kwargs** – the params for *end_train*
- **signal_kwargs** – the params for *prepare_signals*

1.15.3 Online Strategy

`OnlineStrategy` module is an element of online serving.

class `qlib.workflow.online.strategy.OnlineStrategy` (*name_id: str*)

`OnlineStrategy` is working with *Online Manager*, responding to how the tasks are generated, the models are updated and signals are prepared.

__init__ (*name_id: str*)

Init `OnlineStrategy`. This module **MUST** use `Trainer` to finishing model training.

Parameters

- **name_id** (*str*) – a unique name or id.
- **trainer** (`Trainer`, *optional*) – a instance of `Trainer`. Defaults to `None`.

prepare_tasks (*cur_time*, ***kwargs*) → List[dict]

After the end of a routine, check whether we need to prepare and train some new tasks based on *cur_time* (None for latest).. Return the new tasks waiting for training.

You can find the last online models by `OnlineTool.online_models`.

prepare_online_models (*trained_models*, *cur_time=None*) → List[object]

Select some models from trained models and set them to online models. This is a typical implementation to online all trained models, you can override it to implement the complex method. You can find the last online models by `OnlineTool.online_models` if you still need them.

NOTE: Reset all online models to trained models. If there are no trained models, then do nothing.

NOTE: Current implementation is very naive. Here is a more complex situation which is more closer to the practical scenarios. 1. Train new models at the day before *test_start* (at time stamp *T*) 2. Switch models at the *test_start* (at time timestamp *T + 1* typically)

Parameters

- **models** (*list*) – a list of models.
- **cur_time** (*pd.DataFrame*) – current time from OnlineManger. None for the latest.

Returns a list of online models.

Return type List[object]

first_tasks () → List[dict]

Generate a series of tasks firstly and return them.

get_collector () → `qlib.workflow.task.collect.Collector`

Get the instance of `Collector` to collect different results of this strategy.

For example:

- 1) collect predictions in Recorder
- 2) collect signals in a txt file

Returns Collector

```
class qlib.workflow.online.strategy.RollingStrategy (name_id: str,
                                                    task_template: Union[dict,
                                                                    List[dict]],
                                                    rolling_gen: qlib.workflow.task.gen.RollingGen)
```

This example strategy always uses the latest rolling model sas online models.

```
__init__ (name_id: str, task_template: Union[dict, List[dict]], rolling_gen:
          qlib.workflow.task.gen.RollingGen)
Init RollingStrategy.
```

Assumption: the str of *name_id*, the experiment name, and the trainer's experiment name are the same.

Parameters

- **name_id** (*str*) – a unique name or id. Will be also the name of the Experiment.
- **task_template** (*Union[dict, List[dict]]*) – a list of task_template or a single template, which will be used to generate many tasks using *rolling_gen*.
- **rolling_gen** (`RollingGen`) – an instance of `RollingGen`

get_collector (*process_list*=[<*qlib.model.ens.group.RollingGroup* object>], *rec_key_func*=None, *rec_filter_func*=None, *artifacts_key*=None)

Get the instance of **Collector** to collect results. The returned collector must distinguish results in different models.

Assumption: the models can be distinguished based on the model name and rolling test segments. If you do not want this assumption, please implement your method or use another *rec_key_func*.

Parameters

- **rec_key_func** (*Callable*) – a function to get the key of a recorder. If None, use recorder id.
- **rec_filter_func** (*Callable*, *optional*) – filter the recorder by return True or False. Defaults to None.
- **artifacts_key** (*List[str]*, *optional*) – the artifacts key you want to get. If None, get all artifacts.

first_tasks () → *List[dict]*

Use *rolling_gen* to generate different tasks based on *task_template*.

Returns a list of tasks

Return type *List[dict]*

prepare_tasks (*cur_time*) → *List[dict]*

Prepare new tasks based on *cur_time* (None for the latest).

You can find the last online models by *OnlineToolR.online_models*.

Returns a list of new tasks.

Return type *List[dict]*

1.15.4 Online Tool

OnlineTool is a module to set and unset a series of *online* models. The *online* models are some decisive models in some time points, which can be changed with the change of time. This allows us to use efficient submodels as the market-style changing.

class *qlib.workflow.online.utils.OnlineTool*

OnlineTool will manage *online* models in an experiment that includes the model recorders.

__init__ ()

Init *OnlineTool*.

set_online_tag (*tag*, *recorder*: *Union[list, object]*)

Set *tag* to the model to sign whether online.

Parameters

- **tag** (*str*) – the tags in *ONLINE_TAG*, *OFFLINE_TAG*
- **recorder** (*Union[list, object]*) – the model's recorder

get_online_tag (*recorder*: *object*) → *str*

Given a model recorder and return its online tag.

Parameters **recorder** (*Object*) – the model's recorder

Returns the online tag

Return type *str*

reset_online_tag (*recorder: Union[list, object]*)

Offline all models and set the recorders to ‘online’.

Parameters **recorder** (*Union[list, object]*) – the recorder you want to reset to ‘online’.

online_models () → list

Get current *online* models

Returns a list of *online* models.

Return type list

update_online_pred (*to_date=None*)

Update the predictions of *online* models to *to_date*.

Parameters **to_date** (*pd.Timestamp*) – the pred before this date will be updated. None for updating to the latest.

class qlib.workflow.online.utils.**OnlineToolR** (*default_exp_name: str = None*)

The implementation of OnlineTool based on (R)ecorder.

__init__ (*default_exp_name: str = None*)

Init OnlineToolR.

Parameters **default_exp_name** (*str*) – the default experiment name.

set_online_tag (*tag, recorder: Union[qlib.workflow.recorder.Recorder, List[T]]*)

Set *tag* to the model’s recorder to sign whether online.

Parameters

- **tag** (*str*) – the tags in *ONLINE_TAG*, *NEXT_ONLINE_TAG*, *OFFLINE_TAG*
- **recorder** (*Union[Recorder, List]*) – a list of Recorder or an instance of Recorder

get_online_tag (*recorder: qlib.workflow.recorder.Recorder*) → str

Given a model recorder and return its online tag.

Parameters **recorder** (*Recorder*) – an instance of recorder

Returns the online tag

Return type str

reset_online_tag (*recorder: Union[qlib.workflow.recorder.Recorder, List[T]], exp_name: str = None*)

Offline all models and set the recorders to ‘online’.

Parameters

- **recorder** (*Union[Recorder, List]*) – the recorder you want to reset to ‘online’.
- **exp_name** (*str*) – the experiment name. If None, then use *default_exp_name*.

online_models (*exp_name: str = None*) → list

Get current *online* models

Parameters **exp_name** (*str*) – the experiment name. If None, then use *default_exp_name*.

Returns a list of *online* models.

Return type list

update_online_pred (*to_date=None, from_date=None, exp_name: str = None*)

Update the predictions of online models to *to_date*.

Parameters

- **to_date** (*pd.Timestamp*) – the pred before this date will be updated. None for updating to latest time in Calendar.
- **exp_name** (*str*) – the experiment name. If None, then use *default_exp_name*.

1.15.5 Updater

Updater is a module to update artifacts such as predictions when the stock data is updating.

class `qlib.workflow.online.update.RMDLoader` (*rec: qlib.workflow.recorder.Recorder*)

Recorder Model Dataset Loader

__init__ (*rec: qlib.workflow.recorder.Recorder*)

Initialize self. See `help(type(self))` for accurate signature.

get_dataset (*start_time, end_time, segments=None, unprepared_dataset: Optional[qlib.data.dataset.DatasetH] = None*) → `qlib.data.dataset.DatasetH` *Op-*

Load, config and setup dataset.

This dataset is for inference.

Parameters

- **start_time** – the *start_time* of underlying data
- **end_time** – the *end_time* of underlying data
- **segments** – dict the segments config for dataset Due to the time series dataset (TSDatasetH), the test segments maybe different from *start_time* and *end_time*
- **unprepared_dataset** – Optional[DatasetH] if user don't want to load dataset from recorder, please specify user's dataset

Returns the instance of DatasetH

Return type `DatasetH`

class `qlib.workflow.online.update.RecordUpdater` (*record:*

qlib.workflow.recorder.Recorder,
**args, **kwargs*)

Update a specific recorders

__init__ (*record: qlib.workflow.recorder.Recorder, *args, **kwargs*)

Initialize self. See `help(type(self))` for accurate signature.

update (**args, **kwargs*)

Update info for specific recorder

class `qlib.workflow.online.update.DSBasedUpdater` (*record:*

qlib.workflow.recorder.Recorder,
to_date=None, from_date=None,
hist_ref: Optional[int] = None,
freq='day', fname='pred.pkl',
loader_cls: type = <class
'qlib.workflow.online.update.RMDLoader'>)

Dataset-Based Updater - Providing updating feature for Updating data based on Qlib Dataset

Assumption - Based on Qlib dataset - The data to be updated is a multi-level index `pd.DataFrame`. For example label , prediction.

LABEL0

datetime instrument 2021-05-10 SH600000 0.006965

SH600004 0.003407

... .. 2021-05-28 SZ300498 0.015748

SZ300676 -0.001321

```
__init__(record: qlib.workflow.recorder.Recorder, to_date=None, from_date=None, hist_ref:
Optional[int] = None, freq='day', fname='pred.pkl', loader_cls: type = <class
'qlib.workflow.online.update.RMDLoader'>)
```

Init PredUpdater.

Expected behavior in following cases: - if `to_date` is greater than the max date in the calendar, the data will be updated to the latest date - if there are data before `from_date` or after `to_date`, only the data between `from_date` and `to_date` are affected.

Parameters

- **record** – Recorder
- **to_date** – update to prediction to the `to_date` if `to_date` is None:
data will updated to the latest date.
- **from_date** – the update will start from `from_date` if `from_date` is None:
the updating will occur on the next tick after the latest data in historical data
- **hist_ref** – int Sometimes, the dataset will have historical depends. Leave the problem to users to set the length of historical dependency If user doesn't specify this parameter, Updater will try to load dataset to automatically determine the `hist_ref`

Note: the `start_time` is not included in the `hist_ref`

- **loader_cls** – type the class to load the model and dataset

```
prepare_data(unprepared_dataset: Optional[qlib.data.dataset.DatasetH] = None) →
qlib.data.dataset.DatasetH
```

Load dataset - if `unprepared_dataset` is specified, then prepare the dataset directly - Otherwise,

Separating this function will make it easier to reuse the dataset

Returns the instance of `DatasetH`

Return type `DatasetH`

```
update(dataset: qlib.data.dataset.DatasetH = None, write: bool = True, ret_new: bool = False) →
Optional[object]
```

Parameters

- **dataset** (`DatasetH`) – `DatasetH`: the instance of `DatasetH`. None for prepare it again.
- **write** (`bool`) – will the the write action be executed
- **ret_new** (`bool`) – will the updated data be returned

Returns the updated dataset

Return type Optional[object]

get_update_data (*dataset: qlib.data.dataset.Dataset*) → pandas.core.frame.DataFrame
return the updated data based on the given dataset

The difference between *get_update_data* and *update* - *update_date* only include some data specific feature
- *update* include some general routine steps(e.g. prepare dataset, checking)

```
class qlib.workflow.online.update.PredUpdater (record: qlib.workflow.recorder.Recorder,
                                              to_date=None,          from_date=None,
                                              hist_ref: Optional[int] = None,
                                              freq='day',          fname='pred.pkl',
                                              loader_cls: type = <class
                                              'qlib.workflow.online.update.RMDLoader'>)
```

Update the prediction in the Recorder

get_update_data (*dataset: qlib.data.dataset.Dataset*) → pandas.core.frame.DataFrame
return the updated data based on the given dataset

The difference between *get_update_data* and *update* - *update_date* only include some data specific feature
- *update* include some general routine steps(e.g. prepare dataset, checking)

```
class qlib.workflow.online.update.LabelUpdater (record: qlib.workflow.recorder.Recorder,
                                              to_date=None, **kwargs)
```

Update the label in the recorder

Assumption - The label is generated from record_temp.SignalRecord.

__init__ (*record: qlib.workflow.recorder.Recorder, to_date=None, **kwargs*)
Init PredUpdater.

Expected behavior in following cases: - if *to_date* is greater than the max date in the calendar, the data will be updated to the latest date - if there are data before *from_date* or after *to_date*, only the data between *from_date* and *to_date* are affected.

Parameters

- **record** – Recorder
- **to_date** – update to prediction to the *to_date* if *to_date* is None:
data will updated to the latest date.
- **from_date** – the update will start from *from_date* if *from_date* is None:
the updating will occur on the next tick after the latest data in historical data
- **hist_ref** – int Sometimes, the dataset will have historical depends. Leave the problem to users to set the length of historical dependency If user doesn't specify this parameter, Updater will try to load dataset to automatically determine the *hist_ref*

Note: the start_time is not included in the *hist_ref*

- **loader_cls** – type the class to load the model and dataset

get_update_data (*dataset: qlib.data.dataset.Dataset*) → pandas.core.frame.DataFrame
return the updated data based on the given dataset

The difference between `get_update_data` and `update - update_date` only include some data specific feature - `update` include some general routine steps(e.g. prepare dataset, checking)

1.16 Building Formulaic Alphas

1.16.1 Introduction

In quantitative trading practice, designing novel factors that can explain and predict future asset returns are of vital importance to the profitability of a strategy. Such factors are usually called alpha factors, or alphas in short.

A formulaic alpha, as the name suggests, is a kind of alpha that can be presented as a formula or a mathematical expression.

1.16.2 Building Formulaic Alphas in qlib

In Qlib, users can easily build formulaic alphas.

Example

MACD, short for moving average convergence/divergence, is a formulaic alpha used in technical analysis of stock prices. It is designed to reveal changes in the strength, direction, momentum, and duration of a trend in a stock's price.

MACD can be presented as the following formula:

$$MACD = 2 \times (DIF - DEA)$$

Note: *DIF* means Differential value, which is 12-period EMA minus 26-period EMA.

$$DIF = \frac{EMA(CLOSE, 12) - EMA(CLOSE, 26)}{CLOSE}$$

‘DEA’ means a 9-period EMA of the DIF.

$$DEA = \frac{EMA(DIF, 9)}{CLOSE}$$

Users can use `Data Handler` to build formulaic alphas *MACD* in qlib:

Note: Users need to initialize Qlib with `qlib.init` first. Please refer to [initialization](#).

```
>> from qlib.data.dataset.loader import QlibDataLoader
>> MACD_EXP = '(EMA($close, 12) - EMA($close, 26))/ $close - EMA((EMA($close, 12) -
↪EMA($close, 26))/ $close, 9)/ $close'
>> fields = [MACD_EXP] # MACD
>> names = ['MACD']
>> labels = ['Ref($close, -2)/Ref($close, -1) - 1'] # label
>> label_names = ['LABEL']
>> data_loader_config = {
..     "feature": (fields, names),
```

(continues on next page)

(continued from previous page)

```

..     "label": (labels, label_names)
.. }
>> data_loader = QlibDataLoader(config=data_loader_config)
>> df = data_loader.load(instruments='csi300', start_time='2010-01-01', end_time=
↪ '2017-12-31')
>> print(df)

```

		feature	label
		MACD	LABEL
datetime	instrument		
2010-01-04	SH600000	-0.011547	-0.019672
	SH600004	0.002745	-0.014721
	SH600006	0.010133	0.002911
	SH600008	-0.001113	0.009818
	SH600009	0.025878	-0.017758
...	
2017-12-29	SZ300124	0.007306	-0.005074
	SZ300136	-0.013492	0.056352
	SZ300144	-0.000966	0.011853
	SZ300251	0.004383	0.021739
	SZ300315	-0.030557	0.012455

1.16.3 Reference

To learn more about Data Loader, please refer to [Data Loader](#)

To learn more about Data API, please refer to [Data API](#)

1.17 Online & Offline mode

1.17.1 Introduction

Qlib supports Online mode and Offline mode. Only the Offline mode is introduced in this document.

The Online mode is designed to solve the following problems:

- Manage the data in a centralized way. Users don't have to manage data of different versions.
- Reduce the amount of cache to be generated.
- Make the data can be accessed in a remote way.

1.17.2 Qlib-Server

Qlib-Server is the assorted server system for Qlib, which utilizes Qlib for basic calculations and provides extensive server system and cache mechanism. With QlibServer, the data provided for Qlib can be managed in a centralized manner. With Qlib-Server, users can use Qlib in Online mode.

1.17.3 Reference

If users are interested in Qlib-Server and Online mode, please refer to [Qlib-Server Project](#) and [Qlib-Server Document](#).

1.18 Serialization

1.18.1 Introduction

Qlib supports dumping the state of `DataHandler`, `DataSet`, `Processor` and `Model`, etc. into a disk and reloading them.

1.18.2 Serializable Class

Qlib provides a base class `qlib.utils.serial.Serializable`, whose state can be dumped into or loaded from disk in *pickle* format. When users dump the state of a `Serializable` instance, the attributes of the instance whose name **does not** start with `_` will be saved on the disk. However, users can use `config` method or override `default_dump_all` attribute to prevent this feature.

Users can also override `pickle_backend` attribute to choose a pickle backend. The supported value is “pickle” (default and common) and “dill” (dump more things such as function, more information in [here](#)).

1.18.3 Example

Qlib’s serializable class includes `DataHandler`, `DataSet`, `Processor` and `Model`, etc., which are subclass of `qlib.utils.serial.Serializable`. Specifically, `qlib.data.dataset.DatasetH` is one of them. Users can serialize `DatasetH` as follows.

```
#####dump dataset#####
dataset.to_pickle(path="dataset.pkl") # dataset is an instance of qlib.data.dataset.
↳DatasetH

#####reload dataset#####
with open("dataset.pkl", "rb") as file_dataset:
    dataset = pickle.load(file_dataset)
```

Note: Only state of `DatasetH` should be saved on the disk, such as some *mean* and *variance* used for data normalization, etc.

After reloading the `DatasetH`, users need to reinitialize it. It means that users can reset some states of `DatasetH` or `QlibDataHandler` such as *instruments*, *start_time*, *end_time* and *segments*, etc., and generate new data according to the states (data is not state and should not be saved on the disk).

A more detailed example is in this [link](#).

1.18.4 API

Please refer to [Serializable API](#).

1.19 Task Management

1.19.1 Introduction

The [Workflow](#) part introduces how to run research workflow in a loosely-coupled way. But it can only execute one task when you use `qrun`. To automatically generate and execute different tasks, Task Management provides a whole process including *Task Generating*, *Task Storing*, *Task Training* and *Task Collecting*. With this module, users can run their task automatically at different periods, in different losses, or even by different models. The processes of task generation, model training and combine and collect data are shown in the following figure.

This whole process can be used in [Online Serving](#).

An example of the entire process is shown [here](#).

1.19.2 Task Generating

A task consists of *Model*, *Dataset*, *Record*, or anything added by users. The specific task template can be viewed in [Task Section](#). Even though the task template is fixed, users can customize their TaskGen to generate different task by task template.

Here is the base class of TaskGen:

```
class qlib.workflow.task.gen.TaskGen
```

The base class for generating different tasks

Example 1:

input: a specific task template and rolling steps

output: rolling version of the tasks

Example 2:

input: a specific task template and losses list

output: a set of tasks with different losses

generate (*task: dict*) → List[dict]

Generate different tasks based on a task template

Parameters **task** (*dict*) – a task template

Returns A list of tasks

Return type typing.List[dict]

QLib provides a class [RollingGen](#) to generate a list of task of the dataset in different date segments. This class allows users to verify the effect of data from different periods on the model in one experiment. More information is [here](#).

1.19.3 Task Storing

To achieve higher efficiency and the possibility of cluster operation, Task Manager will store all tasks in [MongoDB](#). TaskManager can fetch undone tasks automatically and manage the lifecycle of a set of tasks with error handling. Users **MUST** finish the configuration of [MongoDB](#) when using this module.

Users need to provide the MongoDB URL and database name for using TaskManager in [initialization](#) or make a statement like this.

```
from qlib.config import C
C["mongo"] = {
    "task_url" : "mongodb://localhost:27017/", # your MongoDB url
    "task_db_name" : "rolling_db" # database name
}
```

class qlib.workflow.task.manage.**TaskManager** (*task_pool: str*)

Here is what will a task looks like when it created by TaskManager

```
{
    'def': pickle serialized task definition. using pickle will make it easier
    'filter': json-like data. This is for filtering the tasks.
    'status': 'waiting' | 'running' | 'done'
    'res': pickle serialized task result,
}
```

The tasks manager assumes that you will only update the tasks you fetched. The mongo fetch one and update will make it date updating secure.

This class can be used as a tool from commandline. Here are several examples. You can view the help of manage module with the following commands: `python -m qlib.workflow.task.manage -h` # show manual of manage module CLI `python -m qlib.workflow.task.manage wait -h` # show manual of the wait command of manage

```
python -m qlib.workflow.task.manage -t <pool_name> wait
python -m qlib.workflow.task.manage -t <pool_name> task_stat
```

Note: Assumption: the data in MongoDB was encoded and the data out of MongoDB was decoded

Here are four status which are:

STATUS_WAITING: waiting for training

STATUS_RUNNING: training

STATUS_PART_DONE: finished some step and waiting for next step

STATUS_DONE: all work done

__init__ (*task_pool: str*)

Init Task Manager, remember to make the statement of MongoDB url and database name firstly. A TaskManager instance serves a specific task pool. The static method of this module serves the whole MongoDB.

Parameters **task_pool** (*str*) – the name of Collection in MongoDB

static list () → list

List the all collection(task_pool) of the db.

Returns list

replace_task (*task, new_task*)

Use a new task to replace a old one

Parameters

- **task** – old task
- **new_task** – new task

insert_task (*task*)

Insert a task.

Parameters *task* – the task waiting for insert

Returns pymongo.results.InsertOneResult

insert_task_def (*task_def*)

Insert a task to task_pool

Parameters *task_def* (*dict*) – the task definition

Returns

Return type pymongo.results.InsertOneResult

create_task (*task_def_l*, *dry_run=False*, *print_nt=False*) → List[str]

If the tasks in *task_def_l* are new, then insert new tasks into the *task_pool*, and record *inserted_id*. If a task is not new, then just query its *_id*.

Parameters

- **task_def_l** (*list*) – a list of task
- **dry_run** (*bool*) – if insert those new tasks to task pool
- **print_nt** (*bool*) – if print new task

Returns a list of the *_id* of *task_def_l*

Return type List[str]

fetch_task (*query={}*, *status='waiting'*) → dict

Use query to fetch tasks.

Parameters

- **query** (*dict*, *optional*) – query dict. Defaults to {}.
- **status** (*str*, *optional*) – [description]. Defaults to STATUS_WAITING.

Returns a task(document in collection) after decoding

Return type dict

safe_fetch_task (*query={}*, *status='waiting'*)

Fetch task from *task_pool* using query with contextmanager

Parameters *query* (*dict*) – the dict of query

Returns dict

Return type a task(document in collection) after decoding

query (*query={}*, *decode=True*)

Query task in collection. This function may raise exception *pymongo.errors.CursorNotFound: cursor id not found* if it takes too long to iterate the generator

```
python -m qlib.workflow.task.manage -t <your task pool> query '{"_id":
"615498be837d0053acbc5d58"}'
```

Parameters

- **query** (*dict*) – the dict of query
- **decode** (*bool*) –

Returns dict

Return type a task(document in collection) after decoding

re_query (*_id*) → dict
Use *_id* to query task.

Parameters *_id* (*str*) – *_id* of a document

Returns a task(document in collection) after decoding

Return type dict

commit_task_res (*task*, *res*, *status*='done')
Commit the result to task['res'].

Parameters

- **task** (*[type]*) – [description]
- **res** (*object*) – the result you want to save
- **status** (*str*, *optional*) – STATUS_WAITING, STATUS_RUNNING, STATUS_DONE, STATUS_PART_DONE. Defaults to STATUS_DONE.

return_task (*task*, *status*='waiting')
Return a task to status. Always using in error handling.

Parameters

- **task** (*[type]*) – [description]
- **status** (*str*, *optional*) – STATUS_WAITING, STATUS_RUNNING, STATUS_DONE, STATUS_PART_DONE. Defaults to STATUS_WAITING.

remove (*query*={})
Remove the task using query

Parameters **query** (*dict*) – the dict of query

task_stat (*query*={}) → dict
Count the tasks in every status.

Parameters **query** (*dict*, *optional*) – the query dict. Defaults to {}.

Returns dict

reset_waiting (*query*={})
Reset all running task into waiting status. Can be used when some running task exit unexpected.

Parameters **query** (*dict*, *optional*) – the query dict. Defaults to {}.

prioritize (*task*, *priority*: *int*)
Set priority for task

Parameters

- **task** (*dict*) – The task query from the database
- **priority** (*int*) – the target priority

wait (*query*={})
When multiprocessing, the main progress may fetch nothing from TaskManager because there are still some running tasks. So main progress should wait until all tasks are trained well by other progress or machines.

Parameters **query** (*dict*, *optional*) – the query dict. Defaults to {}.

More information of Task Manager can be found in [here](#).

1.19.4 Task Training

After generating and storing those `task`, it's time to run the `task` which is in the `WAITING` status. QLib provides a method called `run_task` to run those `task` in task pool, however, users can also customize how tasks are executed. An easy way to get the `task_func` is using `qlib.model.trainer.task_train` directly. It will run the whole workflow defined by `task`, which includes *Model*, *Dataset*, *Record*.

```
qlib.workflow.task.manage.run_task(task_func: Callable, task_pool: str, query: dict = {},
                                   force_release: bool = False, before_status: str = 'waiting',
                                   after_status: str = 'done', **kwargs)
```

While the task pool is not empty (has `WAITING` tasks), use `task_func` to fetch and run tasks in task_pool

After running this method, here are 4 situations (before_status -> after_status):

`STATUS_WAITING` -> `STATUS_DONE`: use `task["def"]` as `task_func` param, it means that the task has not been started

`STATUS_WAITING` -> `STATUS_PART_DONE`: use `task["def"]` as `task_func` param

`STATUS_PART_DONE` -> `STATUS_PART_DONE`: use `task["res"]` as `task_func` param, it means that the task has been started but not completed

`STATUS_PART_DONE` -> `STATUS_DONE`: use `task["res"]` as `task_func` param

Parameters

- **task_func** (*Callable*) –
 def (task_def, **kwargs) -> <res which will be committed> the function to run the task
- **task_pool** (*str*) – the name of the task pool (Collection in MongoDB)
- **query** (*dict*) – will use this dict to query task_pool when fetching task
- **force_release** (*bool*) – will the program force to release the resource
- **before_status** (*str*;) – the tasks in before_status will be fetched and trained. Can be `STATUS_WAITING`, `STATUS_PART_DONE`.
- **after_status** (*str*;) – the tasks after trained will become after_status. Can be `STATUS_WAITING`, `STATUS_PART_DONE`.
- **kwargs** – the params for `task_func`

Meanwhile, QLib provides a module called `Trainer`.

```
class qlib.model.trainer.Trainer
```

The trainer can train a list of models. There are `Trainer` and `DelayTrainer`, which can be distinguished by when it will finish real training.

```
__init__()
```

Initialize self. See `help(type(self))` for accurate signature.

```
train(tasks: list, *args, **kwargs) -> list
```

Given a list of task definitions, begin training, and return the models.

For `Trainer`, it finishes real training in this method. For `DelayTrainer`, it only does some preparation in this method.

Parameters `tasks` – a list of tasks

Returns a list of models

Return type list

end_train (*models: list, *args, **kwargs*) → list

Given a list of models, finished something at the end of training if you need. The models may be Recorder, txt file, database, and so on.

For Trainer, it does some finishing touches in this method. For DelayTrainer, it finishes real training in this method.

Parameters *models* – a list of models

Returns a list of models

Return type list

is_delay () → bool

If Trainer will delay finishing *end_train*.

Returns if DelayTrainer

Return type bool

has_worker () → bool

Some trainer has backend worker to support parallel training This method can tell if the worker is enabled.

Returns if the worker is enabled

Return type bool

worker ()

start the worker

Raises NotImplementedError: – If the worker is not supported

Trainer will train a list of tasks and return a list of model recorders. Qlib offer two kinds of Trainer, TrainerR is the simplest way and TrainerRM is based on TaskManager to help manager tasks lifecycle automatically. If you do not want to use Task Manager to manage tasks, then use TrainerR to train a list of tasks generated by TaskGen is enough. [Here](#) are the details about different Trainer.

1.19.5 Task Collecting

Before collecting model training results, you need to use the `qlib.init` to specify the path of mlruns.

To collect the results of task after training, Qlib provides [Collector](#), [Group](#) and [Ensemble](#) to collect the results in a readable, expandable and loosely-coupled way.

[Collector](#) can collect objects from everywhere and process them such as merging, grouping, averaging and so on. It has 2 step action including `collect` (collect anything in a dict) and `process_collect` (process collected dict).

[Group](#) also has 2 steps including `group` (can group a set of object based on *group_func* and change them to a dict) and `reduce` (can make a dict become an ensemble based on some rule). For example: `{(A,B,C1): object, (A,B,C2): object} —group—> {(A,B): {C1: object, C2: object}} —reduce—> {(A,B): object}`

[Ensemble](#) can merge the objects in an ensemble. For example: `{C1: object, C2: object} —Ensemble—> object`. You can set the ensembles you want in the `Collector`'s `process_list`. Common ensembles include `AverageEnsemble` and `RollingEnsemble`. Average ensemble is used to ensemble the results of different models in the same time period. Rollingensemble is used to ensemble the results of different models in the same time period

So the hierarchy is `Collector`'s second step corresponds to `Group`. And `Group`'s second step correspond to `Ensemble`.

For more information, please see [Collector](#), [Group](#) and [Ensemble](#), or the [example](#).

1.20 (P)oint-(I)n-(T)ime Database

1.20.1 Introduction

Point-in-time data is a very important consideration when performing any sort of historical market analysis.

For example, let's say we are backtesting a trading strategy and we are using the past five years of historical data as our input. Our model is assumed to trade once a day, at the market close, and we'll say we are calculating the trading signal for 1 January 2020 in our backtest. At that point, we should only have data for 1 January 2020, 31 December 2019, 30 December 2019 etc.

In financial data (especially financial reports), the same piece of data may be amended for multiple times overtime. If we only use the latest version for historical backtesting, data leakage will happen. Point-in-time database is designed for solving this problem to make sure user get the right version of data at any historical timestamp. It will keep the performance of online trading and historical backtesting the same.

1.20.2 Data Preparation

Qlib provides a crawler to help users to download financial data and then a converter to dump the data in Qlib format. Please follow [scripts/data_collector/pit/README.md](#) to download and convert data. Besides, you can find some additional usage examples there.

1.20.3 File-based design for PIT data

Qlib provides a file-based storage for PIT data.

For each feature, it contains 4 columns, i.e. date, period, value, _next. Each row corresponds to a statement.

The meaning of each feature with filename like *XXX_a.data*:

- *date*: the statement's date of publication.
- *period*: the period of the statement. (e.g. it will be quarterly frequency in most of the markets)
 - If it is an annual period, it will be an integer corresponding to the year
 - If it is an quarterly periods, it will be an integer like *<year><index of quarter>*. The last two decimal digits represents the index of quarter. Others represent the year.
- *value*: the described value
- *_next*: the byte index of the next occurrence of the field.

Besides the feature data, an index *XXX_a.index* is included to speed up the querying performance

The statements are sorted by the *date* in ascending order from the beginning of the file.

```
# the data format from XXXX.data
array([(20070428, 200701, 0.090219 , 4294967295),
      (20070817, 200702, 0.13933 , 4294967295),
      (20071023, 200703, 0.24586301, 4294967295),
      (20080301, 200704, 0.3479 , 80),
      (20080313, 200704, 0.395989 , 4294967295),
      (20080422, 200801, 0.100724 , 4294967295),
      (20080828, 200802, 0.24996801, 4294967295),
      (20081027, 200803, 0.33412001, 4294967295),
      (20090325, 200804, 0.39011699, 4294967295),
      (20090421, 200901, 0.102675 , 4294967295),
```

(continues on next page)

(continued from previous page)

```

(20090807, 200902, 0.230712 , 4294967295),
(20091024, 200903, 0.30072999, 4294967295),
(20100402, 200904, 0.33546099, 4294967295),
(20100426, 201001, 0.083825 , 4294967295),
(20100812, 201002, 0.200545 , 4294967295),
(20101029, 201003, 0.260986 , 4294967295),
(20110321, 201004, 0.30739301, 4294967295),
(20110423, 201101, 0.097411 , 4294967295),
(20110831, 201102, 0.24825101, 4294967295),
(20111018, 201103, 0.318919 , 4294967295),
(20120323, 201104, 0.4039 , 420),
(20120411, 201104, 0.403925 , 4294967295),
(20120426, 201201, 0.112148 , 4294967295),
(20120810, 201202, 0.26484701, 4294967295),
(20121026, 201203, 0.370487 , 4294967295),
(20130329, 201204, 0.45004699, 4294967295),
(20130418, 201301, 0.099958 , 4294967295),
(20130831, 201302, 0.21044201, 4294967295),
(20131016, 201303, 0.30454299, 4294967295),
(20140325, 201304, 0.394328 , 4294967295),
(20140425, 201401, 0.083217 , 4294967295),
(20140829, 201402, 0.16450299, 4294967295),
(20141030, 201403, 0.23408499, 4294967295),
(20150421, 201404, 0.319612 , 4294967295),
(20150421, 201501, 0.078494 , 4294967295),
(20150828, 201502, 0.137504 , 4294967295),
(20151023, 201503, 0.201709 , 4294967295),
(20160324, 201504, 0.26420501, 4294967295),
(20160421, 201601, 0.073664 , 4294967295),
(20160827, 201602, 0.136576 , 4294967295),
(20161029, 201603, 0.188062 , 4294967295),
(20170415, 201604, 0.244385 , 4294967295),
(20170425, 201701, 0.080614 , 4294967295),
(20170728, 201702, 0.15151 , 4294967295),
(20171026, 201703, 0.25416601, 4294967295),
(20180328, 201704, 0.32954201, 4294967295),
(20180428, 201801, 0.088887 , 4294967295),
(20180802, 201802, 0.170563 , 4294967295),
(20181029, 201803, 0.25522 , 4294967295),
(20190329, 201804, 0.34464401, 4294967295),
(20190425, 201901, 0.094737 , 4294967295),
(20190713, 201902, 0. , 1040),
(20190718, 201902, 0.175322 , 4294967295),
(20191016, 201903, 0.25581899, 4294967295)],
dtype=[('date', '<u4'), ('period', '<u4'), ('value', '<f8'), ('_next', '<u4')])
# - each row contains 20 byte

# The data format from XXXX.index. It consists of two parts
# 1) the start index of the data. So the first part of the info will be like
2007
# 2) the remain index data will be like information below
# - The data indicate the **byte index** of first data update of a period.
# - e.g. Because the info at both byte 80 and 100 corresponds to 200704. The byte_
→index of first occurance (i.e. 100) is recorded in the data.
array([
    0,      20,      40,      60,      100,
   120,     140,     160,     180,     200,

```

(continues on next page)

(continued from previous page)

```

220,      240,      260,      280,      300,
320,      340,      360,      380,      400,
440,      460,      480,      500,      520,
540,      560,      580,      600,      620,
640,      660,      680,      700,      720,
740,      760,      780,      800,      820,
840,      860,      880,      900,      920,
940,      960,      980,     1000,     1020,
1060, 4294967295], dtype=uint32)

```

Known limitations:

- Currently, the PIT database is designed for quarterly or annually factors, which can handle fundamental data of financial reports in most markets.
- Qlib leverage the file name to identify the type of the data. File with name like *XXX_q.data* corresponds to quarterly data. File with name like *XXX_a.data* corresponds to annual data.
- The calculation of PIT is not performed in the optimal way. There is great potential to boost the performance of PIT data calculation.

1.21 API Reference

Here you can find all Qlib interfaces.

1.21.1 Data

Provider

class `qlib.data.data.ProviderBackendMixin`

This helper class tries to make the provider based on storage backend more convenient. It is not necessary to inherit this class if that provider doesn't rely on the backend storage.

class `qlib.data.data.CalendarProvider`

Calendar provider base class

Provide calendar data.

calendar (*start_time=None, end_time=None, freq='day', future=False*)

Get calendar of certain market in given time range.

Parameters

- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **freq** (*str*) – time frequency, available: year/quarter/month/week/day.
- **future** (*bool*) – whether including future trading day.

Returns calendar list

Return type list

locate_index (*start_time, end_time, freq, future=False*)

Locate the start time index and end time index in a calendar under certain frequency.

Parameters

- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **freq** (*str*) – time frequency, available: year/quarter/month/week/day.
- **future** (*bool*) – whether including future trading day.

Returns

- *pd.Timestamp* – the real start time.
- *pd.Timestamp* – the real end time.
- *int* – the index of start time.
- *int* – the index of end time.

load_calendar (*freq, future*)

Load original calendar timestamp from file.

Parameters

- **freq** (*str*) – frequency of read calendar file.
- **future** (*bool*) –

Returns list of timestamps

Return type list

class qlib.data.data.**InstrumentProvider**

Instrument provider base class

Provide instrument data.

static instruments (*market: Union[List[T], str] = 'all', filter_pipe: Optional[List[T]] = None*)

Get the general config dictionary for a base market adding several dynamic filters.

Parameters

- **market** (*Union[List, str]*) –
str: market/industry/index shortname, e.g. all/sse/szse/sse50/csi300/csi500.
list: ["ID1", "ID2"]. A list of stocks
- **filter_pipe** (*list*) – the list of dynamic filters.

Returns

- **dict** (*if isinstance(market, str)*) – dict of stockpool config. { 'market'=>base market name, 'filter_pipe'=>list of filters }
 example :
- **list** (*if isinstance(market, list)*) – just return the original list directly. NOTE: this will make the instruments compatible with more cases. The user code will be simpler.

list_instruments (*instruments, start_time=None, end_time=None, freq='day', as_list=False*)

List the instruments based on a certain stockpool config.

Parameters

- **instruments** (*dict*) – stockpool config.

- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **as_list** (*bool*) – return instruments as list or dict.

Returns instruments list or dictionary with time spans

Return type dict or list

class qlib.data.data.**FeatureProvider**

Feature provider class

Provide feature data.

feature (*instrument, field, start_time, end_time, freq*)

Get feature data.

Parameters

- **instrument** (*str*) – a certain instrument.
- **field** (*str*) – a certain field of feature.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **freq** (*str*) – time frequency, available: year/quarter/month/week/day.

Returns data of a certain feature

Return type pd.Series

class qlib.data.data.**PITProvider**

period_feature (*instrument, field, start_index: int, end_index: int, cur_time: pandas._libs.tslibs.timestamps.Timestamp, period: Optional[int] = None*) → pandas.core.series.Series

get the historical periods data series between *start_index* and *end_index*

Parameters

- **start_index** (*int*) – start_index is a relative index to the latest period to cur_time
- **end_index** (*int*) – end_index is a relative index to the latest period to cur_time in most cases, the start_index and end_index will be a non-positive values For example, start_index == -3 end_index == 0 and current period index is cur_idx, then the data between [start_index + cur_idx, end_index + cur_idx] will be retrieved.
- **period** (*int*) – This is used for query specific period. The period is represented with int in Qlib. (e.g. 202001 may represent the first quarter in 2020) NOTE: *period* will override *start_index* and *end_index*

Returns The index will be integers to indicate the periods of the data An typical examples will be TODO

Return type pd.Series

Raises FileNotFoundError – This exception will be raised if the queried data do not exist.

class qlib.data.data.**ExpressionProvider**

Expression provider class

Provide Expression data.

__init__ ()

Initialize self. See help(type(self)) for accurate signature.

expression (*instrument*, *field*, *start_time=None*, *end_time=None*, *freq='day'*) → pandas.core.series.Series
Get Expression data.

The responsibility of *expression* - parse the *field* and load the according data. - When loading the data, it should handle the time dependency of the data. *get_expression_instance* is commonly used in this method

Parameters

- **instrument** (*str*) – a certain instrument.
- **field** (*str*) – a certain field of feature.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **freq** (*str*) – time frequency, available: year/quarter/month/week/day.

Returns

data of a certain expression

The data has two types of format 1) expression with datetime index 2) expression with integer index - because the datetime is not as good as

Return type pd.Series

class qlib.data.data.DatasetProvider

Dataset provider class

Provide Dataset data.

dataset (*instruments*, *fields*, *start_time=None*, *end_time=None*, *freq='day'*, *inst_processors=[]*)
Get dataset data.

Parameters

- **instruments** (*list or dict*) – list/dict of instruments or dict of stockpool config.
- **fields** (*list*) – list of feature instances.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **freq** (*str*) – time frequency.
- **inst_processors** (*Iterable[Union[dict, InstProcessor]]*) – the operations performed on each instrument

Returns a pandas dataframe with <instrument, datetime> index.

Return type pd.DataFrame

static get_instruments_d (*instruments*, *freq*)

Parse different types of input instruments to output instruments_d Wrong format of input instruments will lead to exception.

static get_column_names (*fields*)

Get column names from input fields

static dataset_processor (*instruments_d*, *column_names*, *start_time*, *end_time*, *freq*,
inst_processors=[])

Load and process the data, return the data set. - default using multi-kernel method.

static inst_calculator (*inst*, *start_time*, *end_time*, *freq*, *column_names*, *spans*=None,
g_config=None, *inst_processors*=[])

Calculate the expressions for **one** instrument, return a df result. If the expression has been calculated before, load from cache.

return value: A data frame with index 'datetime' and other data columns.

class qlib.data.data.LocalCalendarProvider (*remote*=False, *backend*={})

Local calendar data provider class

Provide calendar data from local data source.

__init__ (*remote*=False, *backend*={})

Initialize self. See help(type(self)) for accurate signature.

load_calendar (*freq*, *future*)

Load original calendar timestamp from file.

Parameters

- **freq** (*str*) – frequency of read calendar file.
- **future** (*bool*) –

Returns list of timestamps

Return type list

class qlib.data.data.LocalInstrumentProvider (*backend*={})

Local instrument data provider class

Provide instrument data from local data source.

__init__ (*backend*={}) → None

Initialize self. See help(type(self)) for accurate signature.

list_instruments (*instruments*, *start_time*=None, *end_time*=None, *freq*='day', *as_list*=False)

List the instruments based on a certain stockpool config.

Parameters

- **instruments** (*dict*) – stockpool config.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **as_list** (*bool*) – return instruments as list or dict.

Returns instruments list or dictionary with time spans

Return type dict or list

class qlib.data.data.LocalFeatureProvider (*remote*=False, *backend*={})

Local feature data provider class

Provide feature data from local data source.

__init__ (*remote*=False, *backend*={})

Initialize self. See help(type(self)) for accurate signature.

feature (*instrument*, *field*, *start_index*, *end_index*, *freq*)

Get feature data.

Parameters

- **instrument** (*str*) – a certain instrument.
- **field** (*str*) – a certain field of feature.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **freq** (*str*) – time frequency, available: year/quarter/month/week/day.

Returns data of a certain feature

Return type pd.Series

class qlib.data.data.LocalPITProvider

period_feature (*instrument, field, start_index, end_index, cur_time, period=None*)

get the historical periods data series between *start_index* and *end_index*

Parameters

- **start_index** (*int*) – start_index is a relative index to the latest period to cur_time
- **end_index** (*int*) – end_index is a relative index to the latest period to cur_time in most cases, the start_index and end_index will be a non-positive values For example, start_index == -3 end_index == 0 and current period index is cur_idx, then the data between [start_index + cur_idx, end_index + cur_idx] will be retrieved.
- **period** (*int*) – This is used for query specific period. The period is represented with int in Qlib. (e.g. 202001 may represent the first quarter in 2020) NOTE: *period* will override *start_index* and *end_index*

Returns The index will be integers to indicate the periods of the data An typical examples will be TODO

Return type pd.Series

Raises FileNotFoundError – This exception will be raised if the queried data do not exist.

class qlib.data.data.LocalExpressionProvider (*time2idx=True*)

Local expression data provider class

Provide expression data from local data source.

__init__ (*time2idx=True*)

Initialize self. See help(type(self)) for accurate signature.

expression (*instrument, field, start_time=None, end_time=None, freq='day'*)

Get Expression data.

The responsibility of *expression* - parse the *field* and load the according data. - When loading the data, it should handle the time dependency of the data. *get_expression_instance* is commonly used in this method

Parameters

- **instrument** (*str*) – a certain instrument.
- **field** (*str*) – a certain field of feature.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.

- **freq** (*str*) – time frequency, available: year/quarter/month/week/day.

Returns

data of a certain expression

The data has two types of format 1) expression with datetime index 2) expression with integer index - because the datetime is not as good as

Return type `pd.Series`

class `qlib.data.data.LocalDatasetProvider` (*align_time: bool = True*)

Local dataset data provider class

Provide dataset data from local data source.

__init__ (*align_time: bool = True*)

Parameters **align_time** (*bool*) – Will we align the time to calendar the frequency is flexible in some dataset and can't be aligned. For the data with fixed frequency with a shared calendar, the align data to the calendar will provides following benefits - Align queries to the same parameters, so the cache can be shared.

dataset (*instruments, fields, start_time=None, end_time=None, freq='day', inst_processors=[]*)

Get dataset data.

Parameters

- **instruments** (*list or dict*) – list/dict of instruments or dict of stockpool config.
- **fields** (*list*) – list of feature instances.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **freq** (*str*) – time frequency.
- **inst_processors** (*Iterable[Union[dict, InstProcessor]]*) – the operations performed on each instrument

Returns a pandas dataframe with <instrument, datetime> index.

Return type `pd.DataFrame`

static **multi_cache_walker** (*instruments, fields, start_time=None, end_time=None, freq='day'*)

This method is used to prepare the expression cache for the client. Then the client will load the data from expression cache by itself.

static **cache_walker** (*inst, start_time, end_time, freq, column_names*)

If the expressions of one instrument haven't been calculated before, calculate it and write it into expression cache.

class `qlib.data.data.ClientCalendarProvider`

Client calendar data provider class

Provide calendar data by requesting data from server as a client.

__init__ ()

Initialize self. See help(type(self)) for accurate signature.

calendar (*start_time=None, end_time=None, freq='day', future=False*)

Get calendar of certain market in given time range.

Parameters

- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **freq** (*str*) – time frequency, available: year/quarter/month/week/day.
- **future** (*bool*) – whether including future trading day.

Returns calendar list

Return type list

class qlib.data.data.**ClientInstrumentProvider**

Client instrument data provider class

Provide instrument data by requesting data from server as a client.

__init__ ()

Initialize self. See help(type(self)) for accurate signature.

list_instruments (*instruments*, *start_time=None*, *end_time=None*, *freq='day'*, *as_list=False*)

List the instruments based on a certain stockpool config.

Parameters

- **instruments** (*dict*) – stockpool config.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **as_list** (*bool*) – return instruments as list or dict.

Returns instruments list or dictionary with time spans

Return type dict or list

class qlib.data.data.**ClientDatasetProvider**

Client dataset data provider class

Provide dataset data by requesting data from server as a client.

__init__ ()

Initialize self. See help(type(self)) for accurate signature.

dataset (*instruments*, *fields*, *start_time=None*, *end_time=None*, *freq='day'*, *disk_cache=0*, *return_uri=False*, *inst_processors=[]*)

Get dataset data.

Parameters

- **instruments** (*list or dict*) – list/dict of instruments or dict of stockpool config.
- **fields** (*list*) – list of feature instances.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.
- **freq** (*str*) – time frequency.
- **inst_processors** (*Iterable[Union[dict, InstProcessor]]*) – the operations performed on each instrument

Returns a pandas dataframe with <instrument, datetime> index.

Return type pd.DataFrame

class qlib.data.data.BaseProvider

Local provider class It is a set of interface that allow users to access data. Because PITD is not exposed publicly to users, so it is not included in the interface.

To keep compatible with old qlib provider.

features (*instruments, fields, start_time=None, end_time=None, freq='day', disk_cache=None, inst_processors=[]*)

disk_cache [int] whether to skip(0)/use(1)/replace(2) disk_cache

This function will try to use cache method which has a keyword *disk_cache*, and will use provider method if a type error is raised because the DatasetD instance is a provider class.

class qlib.data.data.LocalProvider

features_uri (*instruments, fields, start_time, end_time, freq, disk_cache=1*)

Return the uri of the generated cache of features/dataset

Parameters

- **disk_cache** –
- **instruments** –
- **fields** –
- **start_time** –
- **end_time** –
- **freq** –

class qlib.data.data.ClientProvider

Client Provider

Requesting data from server as a client. Can propose requests:

- Calendar : Directly respond a list of calendars
- Instruments (without filter): Directly respond a list/dict of instruments
- Instruments (with filters): Respond a list/dict of instruments
- Features : Respond a cache uri

The general workflow is described as follows: When the user use client provider to propose a request, the client provider will connect the server and send the request. The client will start to wait for the response. The response will be made instantly indicating whether the cache is available. The waiting procedure will terminate only when the client get the response saying *feature_available* is true. *BUG* : Everytime we make request for certain data we need to connect to the server, wait for the response and disconnect from it. We can't make a sequence of requests within one connection. You can refer to <https://python-socketio.readthedocs.io/en/latest/client.html> for documentation of python-socketIO client.

__init__ ()

Initialize self. See help(type(self)) for accurate signature.

qlib.data.data.CalendarProviderWrapper

alias of *qlib.data.data.CalendarProvider*

qlib.data.data.InstrumentProviderWrapper

alias of *qlib.data.data.InstrumentProvider*

qlib.data.data.FeatureProviderWrapper

alias of *qlib.data.data.FeatureProvider*

```
qlib.data.data.PITProviderWrapper
    alias of qlib.data.data.PITProvider

qlib.data.data.ExpressionProviderWrapper
    alias of qlib.data.data.ExpressionProvider

qlib.data.data.DatasetProviderWrapper
    alias of qlib.data.data.DatasetProvider

qlib.data.data.BaseProviderWrapper
    alias of qlib.data.data.BaseProvider

qlib.data.data.register_all_wrappers(C)
```

Filter

```
class qlib.data.filter.BaseDFilter
    Dynamic Instruments Filter Abstract class

    Users can override this class to construct their own filter

    Override __init__ to input filter regulations

    Override filter_main to use the regulations to filter instruments

    __init__()
        Initialize self. See help(type(self)) for accurate signature.

    static from_config(config)
        Construct an instance from config dict.

        Parameters config (dict) – dict of config parameters.

    to_config()
        Construct an instance from config dict.

        Returns return the dict of config parameters.

        Return type dict

class qlib.data.filter.SeriesDFilter (fstart_time=None, fend_time=None, keep=False)
    Dynamic Instruments Filter Abstract class to filter a series of certain features

    Filters should provide parameters:
        • filter start time
        • filter end time
        • filter rule

    Override __init__ to assign a certain rule to filter the series.

    Override _getFilterSeries to use the rule to filter the series and get a dict of {inst => series}, or override filter_main for more advanced series filter rule

    __init__(fstart_time=None, fend_time=None, keep=False)

        Init function for filter base class. Filter a set of instruments based on a certain rule within a certain period assigned by fstart_time and fend_time.

        Parameters

            • fstart_time (str) – the time for the filter rule to start filter the instruments.

            • fend_time (str) – the time for the filter rule to stop filter the instruments.
```

- **keep** (*bool*) – whether to keep the instruments of which features don't exist in the filter time span.

filter_main (*instruments*, *start_time=None*, *end_time=None*)

Implement this method to filter the instruments.

Parameters

- **instruments** (*dict*) – input instruments to be filtered.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.

Returns filtered instruments, same structure as input instruments.

Return type dict

class qlib.data.filter.**NamedFilter** (*name_rule_re*, *fstart_time=None*, *fend_time=None*)

Name dynamic instrument filter

Filter the instruments based on a regulated name format.

A name rule regular expression is required.

__init__ (*name_rule_re*, *fstart_time=None*, *fend_time=None*)

Init function for name filter class

name_rule_re: **str** regular expression for the name rule.

static from_config (*config*)

Construct an instance from config dict.

Parameters **config** (*dict*) – dict of config parameters.

to_config ()

Construct an instance from config dict.

Returns return the dict of config parameters.

Return type dict

class qlib.data.filter.**ExpressionDFilter** (*rule_expression*, *fstart_time=None*, *fend_time=None*, *keep=False*)

Expression dynamic instrument filter

Filter the instruments based on a certain expression.

An expression rule indicating a certain feature field is required.

Examples

- *basic features filter* : `rule_expression = '$close/$open>5'`
- *cross-sectional features filter* : `rule_expression = '$rank($close)<10'`
- *time-sequence features filter* : `rule_expression = '$Ref($close, 3)>100'`

__init__ (*rule_expression*, *fstart_time=None*, *fend_time=None*, *keep=False*)

Init function for expression filter class

fstart_time: **str** filter the feature starting from this time.

fend_time: **str** filter the feature ending by this time.

rule_expression: **str** an input expression for the rule.

static from_config (*config*)

Construct an instance from config dict.

Parameters **config** (*dict*) – dict of config parameters.

to_config ()

Construct an instance from config dict.

Returns return the dict of config parameters.

Return type dict

Class

class qlib.data.base.**Expression**

Expression base class

Expression is designed to handle the calculation of data with the format below data with two dimension for each instrument, - feature - time: it could be observation time or period time.

- period time is designed for Point-in-time database. For example, the period time maybe 2014Q4, its value can observed for multiple times(different value may be observed at different time due to amendment).

load (*instrument, start_index, end_index, *args*)

load feature This function is responsible for loading feature/expression based on the expression engine.

The concrete implementation will be separated into two parts: 1) caching data, handle errors.

- This part is shared by all the expressions and implemented in Expression
- 2) **processing and calculating data based on the specific expression.**
- This part is different in each expression and implemented in each expression

Expression Engine is shared by different data. Different data will have different extra information for *args*.

Parameters

- **instrument** (*str*) – instrument code.
 - **start_index** (*str*) – feature start index [in calendar].
 - **end_index** (*str*) – feature end index [in calendar].
 - **may contain following information** (**args*) –
 - **if it is used in basic expression engine data, it contains following arguments** (*1*) –
- freq:** str feature frequency.
- **if is used in PIT data, it contains following arguments** (*2*) –
- cur_pit:** it is designed for the point-in-time data.
- period:** int This is used for query specific period. The period is represented with int in Qlib. (e.g. 202001 may represent the first quarter in 2020)

Returns feature series: The index of the series is the calendar index

Return type pd.Series

get_longest_back_rolling()

Get the longest length of historical data the feature has accessed

This is designed for getting the needed range of the data to calculate the features in specific range at first. However, situations like `Ref(Ref($close, -1), 1)` can not be handled rightly.

So this will only used for detecting the length of historical data needed.

get_extended_window_size()

get_extend_window_size

For to calculate this Operator in `range[start_index, end_index]` We have to get the *leaf feature* in `range[start_index - lft_etd, end_index + right_etd]`.

Returns lft_etd, right_etd

Return type (int, int)

class qlib.data.base.**Feature** (*name=None*)

Static Expression

This kind of feature will load data from provider

__init__ (*name=None*)

Initialize self. See `help(type(self))` for accurate signature.

get_longest_back_rolling()

Get the longest length of historical data the feature has accessed

This is designed for getting the needed range of the data to calculate the features in specific range at first. However, situations like `Ref(Ref($close, -1), 1)` can not be handled rightly.

So this will only used for detecting the length of historical data needed.

get_extended_window_size()

get_extend_window_size

For to calculate this Operator in `range[start_index, end_index]` We have to get the *leaf feature* in `range[start_index - lft_etd, end_index + right_etd]`.

Returns lft_etd, right_etd

Return type (int, int)

class qlib.data.base.**PFeature** (*name=None*)

class qlib.data.base.**ExpressionOps**

Operator Expression

This kind of feature will use operator for feature construction on the fly.

Operator

class qlib.data.ops.**ElemOperator** (*feature*)

Element-wise Operator

Parameters *feature* (*Expression*) – feature instance

Returns feature operation output

Return type *Expression*

__init__ (*feature*)

Initialize self. See `help(type(self))` for accurate signature.

get_longest_back_rolling()

Get the longest length of historical data the feature has accessed

This is designed for getting the needed range of the data to calculate the features in specific range at first. However, situations like `Ref(Ref($close, -1), 1)` can not be handled rightly.

So this will only used for detecting the length of historical data needed.

get_extended_window_size()

get_extend_window_size

For to calculate this Operator in `range[start_index, end_index]` We have to get the *leaf feature* in `range[start_index - lft_etd, end_index + right_etd]`.

Returns lft_etd, right_etd

Return type (int, int)

class `qlib.data.ops.NpElemOperator` (*feature, func*)

Numpy Element-wise Operator

Parameters

- **feature** (*Expression*) – feature instance
- **func** (*str*) – numpy feature operation method

Returns feature operation output

Return type *Expression*

__init__ (*feature, func*)

Initialize self. See `help(type(self))` for accurate signature.

class `qlib.data.ops.Abs` (*feature*)

Feature Absolute Value

Parameters **feature** (*Expression*) – feature instance

Returns a feature instance with absolute output

Return type *Expression*

__init__ (*feature*)

Initialize self. See `help(type(self))` for accurate signature.

class `qlib.data.ops.Sign` (*feature*)

Feature Sign

Parameters **feature** (*Expression*) – feature instance

Returns a feature instance with sign

Return type *Expression*

__init__ (*feature*)

Initialize self. See `help(type(self))` for accurate signature.

class `qlib.data.ops.Log` (*feature*)

Feature Log

Parameters **feature** (*Expression*) – feature instance

Returns a feature instance with log

Return type *Expression*

__init__ (*feature*)

Initialize self. See `help(type(self))` for accurate signature.

class `qlib.data.ops.Mask` (*feature, instrument*)

Feature Mask

Parameters

- **feature** (*Expression*) – feature instance
- **instrument** (*str*) – instrument mask

Returns a feature instance with masked instrument

Return type *Expression*

`__init__` (*feature, instrument*)

Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Not` (*feature*)

Not Operator

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance

Returns feature elementwise not output

Return type *Feature*

`__init__` (*feature*)

Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.PairOperator` (*feature_left, feature_right*)

Pair-wise operator

Parameters

- **feature_left** (*Expression*) – feature instance or numeric value
- **feature_right** (*Expression*) – feature instance or numeric value
- **func** (*str*) – operator function

Returns two features' operation output

Return type *Feature*

`__init__` (*feature_left, feature_right*)

Initialize self. See help(type(self)) for accurate signature.

get_longest_back_rolling ()

Get the longest length of historical data the feature has accessed

This is designed for getting the needed range of the data to calculate the features in specific range at first. However, situations like `Ref(Ref($close, -1), 1)` can not be handled rightly.

So this will only used for detecting the length of historical data needed.

get_extended_window_size ()

get_extend_window_size

For to calculate this Operator in `range[start_index, end_index]` We have to get the *leaf feature* in `range[start_index - lft_etd, end_index + right_etd]`.

Returns `lft_etd, right_etd`

Return type (int, int)

class `qlib.data.ops.NpPairOperator` (*feature_left, feature_right, func*)

Numpy Pair-wise operator

Parameters

- **feature_left** (*Expression*) – feature instance or numeric value

- **feature_right** (*Expression*) – feature instance or numeric value
- **func** (*str*) – operator function

Returns two features’ operation output

Return type *Feature*

__init__ (*feature_left, feature_right, func*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Power** (*feature_left, feature_right*)

Power Operator

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance

Returns The bases in feature_left raised to the exponents in feature_right

Return type *Feature*

__init__ (*feature_left, feature_right*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Add** (*feature_left, feature_right*)

Add Operator

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance

Returns two features’ sum

Return type *Feature*

__init__ (*feature_left, feature_right*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Sub** (*feature_left, feature_right*)

Subtract Operator

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance

Returns two features’ subtraction

Return type *Feature*

__init__ (*feature_left, feature_right*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Mul** (*feature_left, feature_right*)

Multiply Operator

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance

Returns two features’ product

Return type *Feature*

__init__ (*feature_left, feature_right*)

Initialize self. See help(type(self)) for accurate signature.

```

class qlib.data.ops.Div (feature_left, feature_right)
    Division Operator
    Parameters
        • feature_left (Expression) – feature instance
        • feature_right (Expression) – feature instance
    Returns two features' division
    Return type Feature
    __init__ (feature_left, feature_right)
        Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.Greater (feature_left, feature_right)
    Greater Operator
    Parameters
        • feature_left (Expression) – feature instance
        • feature_right (Expression) – feature instance
    Returns greater elements taken from the input two features
    Return type Feature
    __init__ (feature_left, feature_right)
        Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.Less (feature_left, feature_right)
    Less Operator
    Parameters
        • feature_left (Expression) – feature instance
        • feature_right (Expression) – feature instance
    Returns smaller elements taken from the input two features
    Return type Feature
    __init__ (feature_left, feature_right)
        Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.Gt (feature_left, feature_right)
    Greater Than Operator
    Parameters
        • feature_left (Expression) – feature instance
        • feature_right (Expression) – feature instance
    Returns bool series indicate left > right
    Return type Feature
    __init__ (feature_left, feature_right)
        Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.Ge (feature_left, feature_right)
    Greater Equal Than Operator
    Parameters
        • feature_left (Expression) – feature instance
        • feature_right (Expression) – feature instance
    Returns bool series indicate left >= right

```

Return type *Feature*

`__init__` (*feature_left*, *feature_right*)

Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Lt` (*feature_left*, *feature_right*)

Less Than Operator

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance

Returns bool series indicate *left < right*

Return type *Feature*

`__init__` (*feature_left*, *feature_right*)

Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Le` (*feature_left*, *feature_right*)

Less Equal Than Operator

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance

Returns bool series indicate *left <= right*

Return type *Feature*

`__init__` (*feature_left*, *feature_right*)

Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Eq` (*feature_left*, *feature_right*)

Equal Operator

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance

Returns bool series indicate *left == right*

Return type *Feature*

`__init__` (*feature_left*, *feature_right*)

Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Ne` (*feature_left*, *feature_right*)

Not Equal Operator

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance

Returns bool series indicate *left != right*

Return type *Feature*

`__init__` (*feature_left*, *feature_right*)

Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.And` (*feature_left*, *feature_right*)

And Operator

Parameters

- **feature_left** (*Expression*) – feature instance

- **feature_right** (*Expression*) – feature instance

Returns two features' row by row & output

Return type *Feature*

__init__ (*feature_left, feature_right*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Or** (*feature_left, feature_right*)

Or Operator

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance

Returns two features' row by row | outputs

Return type *Feature*

__init__ (*feature_left, feature_right*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**If** (*condition, feature_left, feature_right*)

If Operator

Parameters

- **condition** (*Expression*) – feature instance with bool values as condition
- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance

__init__ (*condition, feature_left, feature_right*)

Initialize self. See help(type(self)) for accurate signature.

get_longest_back_rolling ()

Get the longest length of historical data the feature has accessed

This is designed for getting the needed range of the data to calculate the features in specific range at first. However, situations like Ref(Ref(\$close, -1), 1) can not be handled rightly.

So this will only used for detecting the length of historical data needed.

get_extended_window_size ()

get_extend_window_size

For to calculate this Operator in range[start_index, end_index] We have to get the *leaf feature* in range[start_index - lft_etd, end_index + right_etd].

Returns lft_etd, right_etd

Return type (int, int)

class qlib.data.ops.**Rolling** (*feature, N, func*)

Rolling Operator The meaning of rolling and expanding is the same in pandas. When the window is set to 0, the behaviour of the operator should follow *expanding* Otherwise, it follows *rolling*

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size
- **func** (*str*) – rolling method

Returns rolling outputs

Return type *Expression*

`__init__(feature, N, func)`

Initialize self. See help(type(self)) for accurate signature.

`get_longest_back_rolling()`

Get the longest length of historical data the feature has accessed

This is designed for getting the needed range of the data to calculate the features in specific range at first. However, situations like `Ref(Ref($close, -1), 1)` can not be handled rightly.

So this will only used for detecting the length of historical data needed.

`get_extended_window_size()`

`get_extend_window_size`

For to calculate this Operator in `range[start_index, end_index]` We have to get the *leaf feature* in `range[start_index - lft_etd, end_index + right_etd]`.

Returns `lft_etd, right_etd`

Return type `(int, int)`

class `qlib.data.ops.Ref(feature, N)`

Feature Reference

Parameters

- **feature** (`Expression`) – feature instance
- **N** (`int`) – `N = 0`, retrieve the first data; `N > 0`, retrieve data of `N` periods ago; `N < 0`, future data

Returns a feature instance with target reference

Return type `Expression`

`__init__(feature, N)`

Initialize self. See help(type(self)) for accurate signature.

`get_longest_back_rolling()`

Get the longest length of historical data the feature has accessed

This is designed for getting the needed range of the data to calculate the features in specific range at first. However, situations like `Ref(Ref($close, -1), 1)` can not be handled rightly.

So this will only used for detecting the length of historical data needed.

`get_extended_window_size()`

`get_extend_window_size`

For to calculate this Operator in `range[start_index, end_index]` We have to get the *leaf feature* in `range[start_index - lft_etd, end_index + right_etd]`.

Returns `lft_etd, right_etd`

Return type `(int, int)`

class `qlib.data.ops.Mean(feature, N)`

Rolling Mean (MA)

Parameters

- **feature** (`Expression`) – feature instance
- **N** (`int`) – rolling window size

Returns a feature instance with rolling average

Return type `Expression`

`__init__(feature, N)`
 Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Sum(feature, N)`

Rolling Sum

Parameters

- **feature** (`Expression`) – feature instance
- **N** (`int`) – rolling window size

Returns a feature instance with rolling sum

Return type `Expression`

`__init__(feature, N)`
 Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Std(feature, N)`

Rolling Std

Parameters

- **feature** (`Expression`) – feature instance
- **N** (`int`) – rolling window size

Returns a feature instance with rolling std

Return type `Expression`

`__init__(feature, N)`
 Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Var(feature, N)`

Rolling Variance

Parameters

- **feature** (`Expression`) – feature instance
- **N** (`int`) – rolling window size

Returns a feature instance with rolling variance

Return type `Expression`

`__init__(feature, N)`
 Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Skew(feature, N)`

Rolling Skewness

Parameters

- **feature** (`Expression`) – feature instance
- **N** (`int`) – rolling window size

Returns a feature instance with rolling skewness

Return type `Expression`

`__init__(feature, N)`
 Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Kurt(feature, N)`

Rolling Kurtosis

Parameters

- **feature** (`Expression`) – feature instance

- **N** (*int*) – rolling window size

Returns a feature instance with rolling kurtosis

Return type *Expression*

__init__ (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Max** (*feature*, *N*)

Rolling Max

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling max

Return type *Expression*

__init__ (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**IdxMax** (*feature*, *N*)

Rolling Max Index

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling max index

Return type *Expression*

__init__ (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Min** (*feature*, *N*)

Rolling Min

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling min

Return type *Expression*

__init__ (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**IdxMin** (*feature*, *N*)

Rolling Min Index

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling min index

Return type *Expression*

__init__ (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Quantile** (*feature*, *N*, *qscore*)

Rolling Quantile

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling quantile

Return type *Expression*

__init__ (*feature*, *N*, *qscore*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Med** (*feature*, *N*)

Rolling Median

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling median

Return type *Expression*

__init__ (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Mad** (*feature*, *N*)

Rolling Mean Absolute Deviation

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling mean absolute deviation

Return type *Expression*

__init__ (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Rank** (*feature*, *N*)

Rolling Rank (Percentile)

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling rank

Return type *Expression*

__init__ (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**Count** (*feature*, *N*)

Rolling Count

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling count of number of non-NaN elements

Return type *Expression*

`__init__` (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Delta` (*feature*, *N*)

Rolling Delta

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with end minus start in rolling window

Return type *Expression*

`__init__` (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Slope` (*feature*, *N*)

Rolling Slope This operator calculate the slope between *idx* and *feature*. (e.g. [*<feature_t1>*, *<feature_t2>*, *<feature_t3>*] and [1, 2, 3])

Usage Example: - “Slope(*\$close*, %d)/*\$close*”

TODO: # Some users may want pair-wise rolling like *Slope*(*A*, *B*, *N*)

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with linear regression slope of given window

Return type *Expression*

`__init__` (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Rsquare` (*feature*, *N*)

Rolling R-value Square

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with linear regression r-value square of given window

Return type *Expression*

`__init__` (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class `qlib.data.ops.Resi` (*feature*, *N*)

Rolling Regression Residuals

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with regression residuals of given window

Return type *Expression*

`__init__` (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**WMA** (*feature*, *N*)

Rolling WMA

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with weighted moving average output

Return type *Expression*

__init__ (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**EMA** (*feature*, *N*)

Rolling Exponential Mean (EMA)

Parameters

- **feature** (*Expression*) – feature instance
- **N** (*int*, *float*) – rolling window size

Returns a feature instance with regression r-value square of given window

Return type *Expression*

__init__ (*feature*, *N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.**PairRolling** (*feature_left*, *feature_right*, *N*, *func*)

Pair Rolling Operator

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling output of two input features

Return type *Expression*

__init__ (*feature_left*, *feature_right*, *N*, *func*)

Initialize self. See help(type(self)) for accurate signature.

get_longest_back_rolling ()

Get the longest length of historical data the feature has accessed

This is designed for getting the needed range of the data to calculate the features in specific range at first. However, situations like Ref(Ref(\$close, -1), 1) can not be handled rightly.

So this will only used for detecting the length of historical data needed.

get_extended_window_size ()

get_extend_window_size

For to calculate this Operator in range[start_index, end_index] We have to get the *leaf feature* in range[start_index - lft_etd, end_index + rght_etd].

Returns lft_etd, rght_etd

Return type (int, int)

class qlib.data.ops.**Corr** (*feature_left*, *feature_right*, *N*)

Rolling Correlation

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling correlation of two input features

Return type *Expression*

__init__ (*feature_left, feature_right, N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.Cov (*feature_left, feature_right, N*)

Rolling Covariance

Parameters

- **feature_left** (*Expression*) – feature instance
- **feature_right** (*Expression*) – feature instance
- **N** (*int*) – rolling window size

Returns a feature instance with rolling max of two input features

Return type *Expression*

__init__ (*feature_left, feature_right, N*)

Initialize self. See help(type(self)) for accurate signature.

class qlib.data.ops.TResample (*feature, freq, func*)

__init__ (*feature, freq, func*)

Resampling the data to target frequency. The resample function of pandas is used. - the timestamp will be at the start of the time span after resample.

Parameters

- **feature** (*Expression*) – An expression for calculating the feature
- **freq** (*str*) – It will be passed into the resample method for resampling based on given frequency
- **func** (*method*) – The method to get the resampled values Some expression are high frequently used

class qlib.data.ops.OpsWrapper

Ops Wrapper

__init__ ()

Initialize self. See help(type(self)) for accurate signature.

register (*ops_list: List[Union[Type[qlib.data.base.ExpressionOps], dict]]*)

register operator

Parameters **ops_list** (*List[Union[Type[ExpressionOps], dict]]*) –

- if type(*ops_list*) is *List[Type[ExpressionOps]]*, each element of *ops_list* represents the operator class, which should be the subclass of *ExpressionOps*.
- if type(*ops_list*) is *List[dict]*, each element of *ops_list* represents the config of operator, which has the

{ “class”: class_name, “module_path”: path,

} Note: *class* should be the class name of operator, *module_path* should be a python module or path of file.

`qlib.data.ops.register_all_ops(C)`
 register all operator

Cache

class `qlib.data.cache.MemCacheUnit(*args, **kwargs)`
 Memory Cache Unit.

__init__(*args, **kwargs)
 Initialize self. See help(type(self)) for accurate signature.

limited
 whether memory cache is limited

class `qlib.data.cache.MemCache(mem_cache_size_limit=None, limit_type='length')`
 Memory cache.

__init__(mem_cache_size_limit=None, limit_type='length')

Parameters

- **mem_cache_size_limit** (cache max size.) –
- **limit_type** (length or sizeof; length(call fun: len), size(call fun: sys.getsizeof)) –

class `qlib.data.cache.ExpressionCache(provider)`
 Expression cache mechanism base class.

This class is used to wrap expression provider with self-defined expression cache mechanism.

Note: Override the `_uri` and `_expression` method to create your own expression cache mechanism.

expression (instrument, field, start_time, end_time, freq)
 Get expression data.

Note: Same interface as `expression` method in expression provider

update (cache_uri: Union[str, pathlib.Path], freq: str = 'day')
 Update expression cache to latest calendar.

Override this method to define how to update expression cache corresponding to users' own cache mechanism.

Parameters

- **cache_uri** (str or Path) – the complete uri of expression cache file (include dir path).
- **freq** (str) –

Returns 0(successful update)/ 1(no need to update)/ 2(update failure).

Return type int

class `qlib.data.cache.DatasetCache(provider)`
 Dataset cache mechanism base class.

This class is used to wrap dataset provider with self-defined dataset cache mechanism.

Note: Override the `_uri` and `_dataset` method to create your own dataset cache mechanism.

dataset (*instruments, fields, start_time=None, end_time=None, freq='day', disk_cache=1, inst_processors=[]*)
 Get feature dataset.

Note: Same interface as *dataset* method in dataset provider

Note: The server use `redis_lock` to make sure read-write conflicts will not be triggered but client readers are not considered.

update (*cache_uri: Union[str, pathlib.Path], freq: str = 'day'*)
 Update dataset cache to latest calendar.

Override this method to define how to update dataset cache corresponding to users' own cache mechanism.

Parameters

- **cache_uri** (*str or Path*) – the complete uri of dataset cache file (include dir path).
- **freq** (*str*) –

Returns 0(successful update)/ 1(no need to update)/ 2(update failure)

Return type int

static cache_to_origin_data (*data, fields*)
 cache data to origin data

Parameters

- **data** – `pd.DataFrame`, cache data.
- **fields** – feature fields.

Returns `pd.DataFrame`.

static normalize_uri_args (*instruments, fields, freq*)
 normalize uri args

class `qlib.data.cache.DiskExpressionCache` (*provider, **kwargs*)
 Prepared cache mechanism for server.

__init__ (*provider, **kwargs*)
 Initialize self. See `help(type(self))` for accurate signature.

gen_expression_cache (*expression_data, cache_path, instrument, field, freq, last_update*)
 use bin file to save like feature-data.

update (*sid, cache_uri, freq: str = 'day'*)
 Update expression cache to latest calendar.

Override this method to define how to update expression cache corresponding to users' own cache mechanism.

Parameters

- **cache_uri** (*str or Path*) – the complete uri of expression cache file (include dir path).
- **freq** (*str*) –

Returns 0(successful update)/ 1(no need to update)/ 2(update failure).

Return type int

class qlib.data.cache.DiskDatasetCache (*provider, **kwargs*)

Prepared cache mechanism for server.

__init__ (*provider, **kwargs*)

Initialize self. See help(type(self)) for accurate signature.

classmethod read_data_from_cache (*cache_path: Union[str, pathlib.Path], start_time, end_time, fields*)

read_cache_from

This function can read data from the disk cache dataset

Parameters

- **cache_path** –
- **start_time** –
- **end_time** –
- **fields** – The fields order of the dataset cache is sorted. So rearrange the columns to make it consistent.

Returns

class IndexManager (*cache_path: Union[str, pathlib.Path]*)

The lock is not considered in the class. Please consider the lock outside the code. This class is the proxy of the disk data.

__init__ (*cache_path: Union[str, pathlib.Path]*)

Initialize self. See help(type(self)) for accurate signature.

gen_dataset_cache (*cache_path: Union[str, pathlib.Path], instruments, fields, freq, inst_processors=[]*)

Note: This function does not consider the cache read write lock. Please

Acquire the lock outside this function

The format the cache contains 3 parts(followed by typical filename).

- index : cache/d41366901e25de3ec47297f12e2ba11d.index
 - The content of the file may be in following format(pandas.Series)

	start	end
1999-11-10 00:00:00	0	1
1999-11-11 00:00:00	1	2
1999-11-12 00:00:00	2	3
...		

Note: The start is closed. The end is open!!!!

- Each line contains two element <start_index, end_index> with a timestamp as its index.
 - It indicates the *start_index* (included) and *end_index* (excluded) of the data for *timestamp*
- meta data: cache/d41366901e25de3ec47297f12e2ba11d.meta
 - data : cache/d41366901e25de3ec47297f12e2ba11d
 - This is a hdf file sorted by datetime

Parameters

- **cache_path** – The path to store the cache.
- **instruments** – The instruments to store the cache.
- **fields** – The fields to store the cache.
- **freq** – The freq to store the cache.
- **inst_processors** – Instrument processors.

:return type pd.DataFrame; The fields of the returned DataFrame are consistent with the parameters of the function.

update (*cache_uri*, *freq*: str = 'day')

Update dataset cache to latest calendar.

Override this method to define how to update dataset cache corresponding to users' own cache mechanism.

Parameters

- **cache_uri** (*str* or *Path*) – the complete uri of dataset cache file (include dir path).
- **freq** (*str*) –

Returns 0(successful update)/ 1(no need to update)/ 2(update failure)

Return type int

Storage

class qlib.data.storage.storage.**BaseStorage**

class qlib.data.storage.storage.**CalendarStorage** (*freq*: str, *future*: bool, ***kwargs*)

The behavior of CalendarStorage's methods and List's methods of the same name remain consistent

__init__ (*freq*: str, *future*: bool, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

data

get all data

Raises ValueError – If the data(storage) does not exist, raise ValueError

index (*value*: str) → int

Raises ValueError – If the data(storage) does not exist, raise ValueError

class qlib.data.storage.storage.**InstrumentStorage** (*market*: str, *freq*: str, ***kwargs*)

__init__ (*market: str, freq: str, **kwargs*)
Initialize self. See help(type(self)) for accurate signature.

data
get all data

Raises `ValueError` – If the data(storage) does not exist, raise `ValueError`

update (*[E], **F*) → `None`. Update D from mapping/iterable E and F.

Notes

If E present and has a `.keys()` method, does: for k in E: `D[k] = E[k]`

If E present and lacks `.keys()` method, does: for (k, v) in E: `D[k] = v`

In either case, this is followed by: for k, v in F.items(): `D[k] = v`

class `qlib.data.storage.storage.FeatureStorage` (*instrument: str, field: str, freq: str, **kwargs*)

__init__ (*instrument: str, field: str, freq: str, **kwargs*)
Initialize self. See help(type(self)) for accurate signature.

data
get all data

Notes

if data(storage) does not exist, return empty `pd.Series`: `return pd.Series(dtype=np.float32)`

start_index
get FeatureStorage start index

Notes

If the data(storage) does not exist, return `None`

end_index
get FeatureStorage end index

Notes

The right index of the data range (both sides are closed)

The next data appending point will be `end_index + 1`

If the data(storage) does not exist, return `None`

write (*data_array: Union[List[T], numpy.ndarray, Tuple], index: int = None*)
Write data_array to FeatureStorage starting from index.

Notes

If index is None, append data_array to feature.

If len(data_array) == 0; return

If (index - self.end_index) >= 1, self[end_index+1: index] will be filled with np.nan

Examples

rebase (*start_index: int = None, end_index: int = None*)

Rebase the start_index and end_index of the FeatureStorage.

start_index and end_index are closed intervals: [start_index, end_index]

Examples

rewrite (*data: Union[List[T], numpy.ndarray, Tuple], index: int*)

overwrite all data in FeatureStorage with data

Parameters

- **data** (*Union[List, np.ndarray, Tuple]*) – data
- **index** (*int*) – data start index

class qlib.data.storage.file_storage.**FileStorageMixin**

FileStorageMixin, applicable to FileXXXStorage Subclasses need to have provider_uri, freq, storage_name, file_name attributes

check ()

check self.uri

Raises ValueError

class qlib.data.storage.file_storage.**FileCalendarStorage** (*freq: str, future: bool, provider_uri: dict = None, **kwargs*)

__init__ (*freq: str, future: bool, provider_uri: dict = None, **kwargs*)

Initialize self. See help(type(self)) for accurate signature.

data

get all data

Raises ValueError – If the data(storage) does not exist, raise ValueError

index (*value: str*) → int

Raises ValueError – If the data(storage) does not exist, raise ValueError

class qlib.data.storage.file_storage.**FileInstrumentStorage** (*market: str, freq: str, provider_uri: dict = None, **kwargs*)

__init__ (*market: str, freq: str, provider_uri: dict = None, **kwargs*)

Initialize self. See help(type(self)) for accurate signature.

data

get all data

Raises `ValueError` – If the data(storage) does not exist, raise `ValueError`

update (`[E]`, `**F`) \rightarrow `None`. Update `D` from mapping/iterable `E` and `F`.

Notes

If `E` present and has a `.keys()` method, does: for `k` in `E`: `D[k] = E[k]`

If `E` present and lacks `.keys()` method, does: for `(k, v)` in `E`: `D[k] = v`

In either case, this is followed by: for `k, v` in `F.items()`: `D[k] = v`

```
class qlib.data.storage.file_storage.FileFeatureStorage (instrument: str, field: str,
                                                         freq: str, provider_uri: dict
                                                         = None, **kwargs)
```

```
__init__ (instrument: str, field: str, freq: str, provider_uri: dict = None, **kwargs)
    Initialize self. See help(type(self)) for accurate signature.
```

```
data
    get all data
```

Notes

if data(storage) does not exist, return empty `pd.Series`: `return pd.Series(dtype=np.float32)`

```
write (data_array: Union[List[T], numpy.ndarray], index: int = None)  $\rightarrow$  None
    Write data_array to FeatureStorage starting from index.
```

Notes

If `index` is `None`, append `data_array` to feature.

If `len(data_array) == 0`; return

If `(index - self.end_index) >= 1`, `self[end_index+1: index]` will be filled with `np.nan`

Examples

```
start_index
    get FeatureStorage start index
```

Notes

If the data(storage) does not exist, return `None`

```
end_index
    get FeatureStorage end index
```

Notes

The right index of the data range (both sides are closed)

The next data appending point will be `end_index + 1`

If the data(storage) does not exist, return `None`

Dataset

Dataset Class

Data Loader

class qlib.data.dataset.loader.**DataLoader**

DataLoader is designed for loading raw data from original data source.

load (*instruments*, *start_time=None*, *end_time=None*) → pandas.core.frame.DataFrame
load the data as pd.DataFrame.

Example of the data (The multi-index of the columns is optional.):

		feature				
		label				
		\$close	\$volume	Ref(\$close, 1)	Mean (
		\$close, 3)	\$high-\$low		LABEL0	
datetime	instrument					
2010-01-04	SH600000	81.807068	17145150.0	83.737389		
		83.016739	2.741058	0.0032		
	SH600004	13.313329	11800983.0	13.313329		
		13.317701	0.183632	0.0042		
	SH600005	37.796539	12231662.0	38.258602		
		37.919757	0.970325	0.0289		

Parameters

- **instruments** (*str* or *dict*) – it can either be the market name or the config file of instruments generated by InstrumentProvider.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.

Returns data load from the under layer source

Return type pd.DataFrame

class qlib.data.dataset.loader.**DLWParser** (*config: Union[list, tuple, dict]*)

(D)ata(L)oader (W)ith (P)arser for features and names

Extracting this class so that QlibDataLoader and other dataloaders(such as QdbDataLoader) can share the fields.

__init__ (*config: Union[list, tuple, dict]*)

Parameters **config** (*Union[list, tuple, dict]*) – Config will be used to describe the fields and column names

load_group_df (*instruments*, *exprs: list*, *names: list*, *start_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp] = None*, *end_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp] = None*, *gp_name: str = None*) → pandas.core.frame.DataFrame
load the dataframe for specific group

Parameters

- **instruments** – the instruments.
- **exprs** (*list*) – the expressions to describe the content of the data.

- **names** (*list*) – the name of the data.

Returns the queried dataframe.

Return type `pd.DataFrame`

load (*instruments=None, start_time=None, end_time=None*) → `pandas.core.frame.DataFrame`

load the data as `pd.DataFrame`.

Example of the data (The multi-index of the columns is optional.):

		feature			
		label			
		\$close	\$volume	Ref(\$close, 1)	Mean (
		\$high-\$low			
		LABEL0			
	datetime	instrument			
	2010-01-04	SH600000	81.807068	17145150.0	83.737389
	83.016739	2.741058	0.0032		
		SH600004	13.313329	11800983.0	13.313329
	13.317701	0.183632	0.0042		
		SH600005	37.796539	12231662.0	38.258602
	37.919757	0.970325	0.0289		

Parameters

- **instruments** (*str or dict*) – it can either be the market name or the config file of instruments generated by `InstrumentProvider`.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.

Returns data load from the under layer source

Return type `pd.DataFrame`

```
class qlib.data.dataset.loader.QlibDataLoader (config: Tuple[list, tuple, dict], filter_pipe:
List[T] = None, swap_level: bool =
True, freq: Union[str, dict] = 'day',
inst_processor: dict = None)
```

Same as `QlibDataLoader`. The fields can be define by config

```
__init__ (config: Tuple[list, tuple, dict], filter_pipe: List[T] = None, swap_level: bool = True, freq:
Union[str, dict] = 'day', inst_processor: dict = None)
```

Parameters

- **config** (*Tuple[list, tuple, dict]*) – Please refer to the doc of `DLW-Parser`
- **filter_pipe** – Filter pipe for the instruments
- **swap_level** – Whether to swap level of `MultiIndex`
- **freq** (*dict or str*) – If `type(config) == dict` and `type(freq) == str`, load config data using `freq`. If `type(config) == dict` and `type(freq) == dict`, load config[<group_name>] data using `freq[<group_name>]`
- **inst_processor** (*dict*) – If `inst_processor` is not `None` and `type(config) == dict`, load config[<group_name>] data using `inst_processor[<group_name>]`

```
load_group_df(instruments, exprs: list, names: list, start_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp] = None, end_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp] = None, gp_name: str = None) → pandas.core.frame.DataFrame
```

load the dataframe for specific group

Parameters

- **instruments** – the instruments.
- **exprs** (*list*) – the expressions to describe the content of the data.
- **names** (*list*) – the name of the data.

Returns the queried dataframe.

Return type `pd.DataFrame`

```
class qlib.data.dataset.loader.StaticDataLoader(config: Union[dict, str, pandas.core.frame.DataFrame], join='outer')
```

DataLoader that supports loading data from file or as provided.

```
__init__(config: Union[dict, str, pandas.core.frame.DataFrame], join='outer')
```

Parameters

- **config** (*dict*) – {fields_group: <path or object>}
- **join** (*str*) – How to align different dataframes

load (*instruments=None, start_time=None, end_time=None*) → `pandas.core.frame.DataFrame`
load the data as `pd.DataFrame`.

Example of the data (The multi-index of the columns is optional.):

		feature			
		label			
		\$close	\$volume	Ref(\$close, 1)	Mean(
		\$high-\$low			
		LABEL0			
datetime	instrument				
2010-01-04	SH600000	81.807068	17145150.0	83.737389	
↪ 83.016739		2.741058	0.0032		
	SH600004	13.313329	11800983.0	13.313329	
↪ 13.317701		0.183632	0.0042		
	SH600005	37.796539	12231662.0	38.258602	
↪ 37.919757		0.970325	0.0289		

Parameters

- **instruments** (*str or dict*) – it can either be the market name or the config file of instruments generated by `InstrumentProvider`.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.

Returns data load from the under layer source

Return type `pd.DataFrame`


```
class qlib.data.dataset.loader.DataLoaderDH (handler_config: dict, fetch_kwargs: dict =  

{}, is_group=False)
```

DataLoader based on (D)ata (H)andler It is designed to load multiple data from data handler - If you just want to load data from single datahandler, you can write them in single data handler

TODO: What make this module not that easy to use. - For online scenario

- The underlayer data handler should be configured. But data loader doesn't provide such interface & hook.

```
__init__ (handler_config: dict, fetch_kwargs: dict = {}, is_group=False)
```

Parameters

- **handler_config** (*dict*) – handler_config will be used to describe the handlers
- **fetch_kwargs** (*dict*) – fetch_kwargs will be used to describe the different arguments of fetch method, such as col_set, squeeze, data_key, etc.
- **is_group** (*bool*) – is_group will be used to describe whether the key of handler_config is group

load (*instruments=None, start_time=None, end_time=None*) → pandas.core.frame.DataFrame
load the data as pd.DataFrame.

Example of the data (The multi-index of the columns is optional.):

		feature			
		label			
		\$close	\$volume	Ref(\$close, 1)	Mean(
		\$high-\$low	LABEL0		
datetime	instrument				
2010-01-04	SH600000	81.807068	17145150.0	83.737389	
→ 83.016739	2.741058	0.0032			
	SH600004	13.313329	11800983.0	13.313329	
→ 13.317701	0.183632	0.0042			
	SH600005	37.796539	12231662.0	38.258602	
→ 37.919757	0.970325	0.0289			

Parameters

- **instruments** (*str or dict*) – it can either be the market name or the config file of instruments generated by InstrumentProvider.
- **start_time** (*str*) – start of the time range.
- **end_time** (*str*) – end of the time range.

Returns data load from the under layer source

Return type pd.DataFrame

Data Handler

```
class qlib.data.dataset.handler.DataHandler (instruments=None, start_time=None,  

end_time=None, data_loader: Union[dict,  

str, qlib.data.dataset.loader.DataLoader] =  

None, init_data=True, fetch_orig=True)
```

The steps to using a handler 1. initialized data handler (call by *init*). 2. use the data.

The data handler try to maintain a handler with 2 level. *datetime & instruments*.

Any order of the index level can be supported (The order will be implied in the data). The order `<datetime, instruments>` will be used when the dataframe index name is missed.

Example of the data: The multi-index of the columns is optional.

		feature			
↪	label				
		\$close	\$volume	Ref(\$close, 1)	Mean(\$close, 3)
↪	\$high-\$low LABEL0				
	datetime instrument				
	2010-01-04 SH600000	81.807068	17145150.0	83.737389	83.016739
↪	2.741058 0.0032				
	SH600004	13.313329	11800983.0	13.313329	13.317701
↪	0.183632 0.0042				
	SH600005	37.796539	12231662.0	38.258602	37.919757
↪	0.970325 0.0289				

Tips for improving the performance of datahandler - Fetching data with `col_set=CS_RAW` will return the raw data and may avoid pandas from copying the data when calling `loc`

`__init__` (*instruments=None, start_time=None, end_time=None, data_loader: Union[dict, str, qlib.data.dataset.loader.DataLoader] = None, init_data=True, fetch_orig=True*)

Parameters

- **instruments** – The stock list to retrieve.
- **start_time** – start_time of the original data.
- **end_time** – end_time of the original data.
- **data_loader** (*Union[dict, str, DataLoader]*) – data loader to load the data.
- **init_data** – initialize the original data in the constructor.
- **fetch_orig** (*bool*) – Return the original data instead of copy if possible.

`config` (***kwargs*)

configuration of data. # what data to be loaded from data source

This method will be used when loading pickled handler from dataset. The data will be initialized with different time range.

`setup_data` (*enable_cache: bool = False*)

Set Up the data in case of running initialization for multiple time

It is responsible for maintaining following variable 1) `self._data`

Parameters `enable_cache` (*bool*) – default value is false:

- if `enable_cache == True`:
the processed data will be saved on disk, and handler will load the cached data from the disk directly when we call `init` next time

`fetch` (*selector: Union[pandas._libs.timestamps.Timestamp, slice, str, pandas.core.indexes.base.Index] = slice(None, None, None), level: Union[str, int] = 'datetime', col_set: Union[str, List[str]] = '__all__', squeeze: bool = False, proc_func: Callable = None*)
→ `pandas.core.frame.DataFrame`
fetch data from underlying data source

Parameters

- **selector** (*Union[pd.Timestamp, slice, str]*) – describe how to select data by index It can be categories as following - fetch single index - fetch a range of index

- a slice range
- pd.Index for specific indexes

Following conflictions may occurs - Does [20200101”, “20210101”] mean selecting this slice or these two days?

- slice have higher priorities

- **level** (*Union[str, int]*) – which index level to select the data

- **col_set** (*Union[str, List[str]]*) –

- if isinstance(col_set, str):
 - select a set of meaningful, pd.Index columns.(e.g. features, columns)
 - if col_set == CS_RAW:** the raw dataset will be returned.
- if isinstance(col_set, List[str]):
 - select several sets of meaningful columns, the returned data has multiple levels

- **proc_func** (*Callable*) –

- Give a hook for processing data before fetching
- **An example to explain the necessity of the hook:**
 - * A Dataset learned some processors to process data which is related to data segmentation
 - * It will apply them every time when preparing data.
 - * The learned processor require the dataframe remains the same format when fitting and applying
 - * However the data format will change according to the parameters.
 - * So the processors should be applied to the underlayer data.

- **squeeze** (*bool*) – whether squeeze columns and index

Returns

Return type pd.DataFrame.

get_cols (*col_set='__all'*) → list
get the column names

Parameters **col_set** (*str*) – select a set of meaningful columns.(e.g. features, columns)

Returns list of column names

Return type list

get_range_selector (*cur_date: Union[pandas._libs.tslibs.timestamps.Timestamp, str], periods: int*) → slice
get range selector by number of periods

Parameters

- **cur_date** (*pd.Timestamp or str*) – current date

- **periods** (*int*) – number of periods

get_range_iterator (*periods: int, min_periods: Optional[int] = None, **kwargs*)
 → Iterator[Tuple[pandas._libs.tslibs.timestamps.Timestamp, pandas.core.frame.DataFrame]]
 get a iterator of sliced data with given periods

Parameters

- **periods** (*int*) – number of periods.
- **min_periods** (*int*) – minimum periods for sliced dataframe.
- **kwargs** (*dict*) – will be passed to *self.fetch*.

```
class qlib.data.dataset.handler.DataHandlerLP (instruments=None,  

                                             start_time=None, end_time=None,  

                                             data_loader: Union[dict, str,  

                                             qlib.data.dataset.loader.DataLoader]  

                                             = None, infer_processors: List[T] =  

                                             [], learn_processors: List[T] = [],  

                                             shared_processors: List[T] = [], process_type='append',  

                                             drop_raw=False,  

                                             **kwargs)
```

DataHandler with (L)earnable (P)rocessor

This handler will produce three pieces of data in `pd.DataFrame` format. - `DK_R / self._data`: the raw data loaded from the loader - `DK_I / self._infer`: the data processed for inference - `DK_L / self._learn`: the data processed for learning model.

The motivation of using different processor workflows for learning and inference Here are some examples. - The instrument universe for learning and inference may be different. - The processing of some samples may rely on label (for example, some samples hit the limit may need extra processing or be dropped).

These processors only apply to the learning phase.

Tips to improve the performance of data handler - To reduce the memory cost

- `drop_raw=True`: this will modify the data inplace on raw data;

```
__init__ (instruments=None, start_time=None, end_time=None, data_loader: Union[dict,  

          str, qlib.data.dataset.loader.DataLoader] = None, infer_processors: List[T] = [],  

          learn_processors: List[T] = [], shared_processors: List[T] = [], process_type='append',  

          drop_raw=False, **kwargs)
```

Parameters

- **infer_processors** (*list*) –
 - list of <description info> of processors to generate data for inference
 - example of <description info>:
 - **learn_processors** (*list*) – similar to `infer_processors`, but for generating data for learning models
 - **process_type** (*str*) – `PTYPE_I` = 'independent'
 - `self._infer` will be processed by `infer_processors`
 - `self._learn` will be processed by `learn_processors`
- `PTYPE_A` = 'append'
- `self._infer` will be processed by `infer_processors`
 - `self._learn` will be processed by `infer_processors` + `learn_processors`

* (e.g. `self._infer` processed by `learn_processors`)

- **drop_raw** (*bool*) – Whether to drop the raw data

fit ()
fit data without processing the data

fit_process_data ()
fit and process data

The input of the *fit* will be the output of the previous processor

process_data (*with_fit: bool = False*)
process_data data. Fun *processor.fit* if necessary

Notation: (data) [processor]

```
# data processing flow of self.process_type == DataHandlerLP.PTYPE_I (self._data)-
[shared_processors]-(_shared_df)-[learn_processors]-(_learn_df)
-[infer_processors]-(_infer_df)

# data processing flow of self.process_type == DataHandlerLP.PTYPE_A (self._data)-
[shared_processors]-(_shared_df)-[infer_processors]-(_infer_df)-[learn_processors]-(_learn_df)
```

Parameters with_fit (*bool*) – The input of the *fit* will be the output of the previous processor

config (*processor_kwargs: dict = None, **kwargs*)
configuration of data. # what data to be loaded from data source

This method will be used when loading pickled handler from dataset. The data will be initialized with different time range.

setup_data (*init_type: str = 'fit_seq', **kwargs*)
Set up the data in case of running initialization for multiple time

Parameters

- **init_type** (*str*) – The type *IT_** listed above.
- **enable_cache** (*bool*) – default value is false:
 - if *enable_cache* == True:

the processed data will be saved on disk, and handler will load the cached data from the disk directly when we call *init* next time

fetch (*selector: Union[pandas._libs.tslibs.timestamps.Timestamp, slice, str] = slice(None, None, None), level: Union[str, int] = 'datetime', col_set='__all__', data_key: str = 'infer', squeeze: bool = False, proc_func: Callable = None*) → `pandas.core.frame.DataFrame`
fetch data from underlying data source

Parameters

- **selector** (*Union[pd.Timestamp, slice, str]*) – describe how to select data by index.
- **level** (*Union[str, int]*) – which index level to select the data.
- **col_set** (*str*) – select a set of meaningful columns.(e.g. features, columns).
- **data_key** (*str*) – the data to fetch: *DK_**.
- **proc_func** (*Callable*) – please refer to the doc of `DataHandler.fetch`

Returns

Return type `pd.DataFrame`

get_cols (*col_set*='__all__', *data_key*: *str* = 'infer') → list
get the column names

Parameters

- **col_set** (*str*) – select a set of meaningful columns.(e.g. features, columns).
- **data_key** (*str*) – the data to fetch: DK_*.

Returns list of column names

Return type list

classmethod cast (*handler*: *qlib.data.dataset.handler.DataHandlerLP*) → *qlib.data.dataset.handler.DataHandlerLP*

Motivation - A user create a datahandler in his customized package. Then he want to share the processed handler to other users without introduce the package dependency and complicated data processing logic.
- This class make it possible by casting the class to `DataHandlerLP` and only keep the processed data

Parameters handler (`DataHandlerLP`) – A subclass of `DataHandlerLP`

Returns the converted processed data

Return type `DataHandlerLP`

Processor

`qlib.data.dataset.processor.get_group_columns` (*df*: *pandas.core.frame.DataFrame*, *group*: *Optional[str]*)

get a group of columns from multi-index columns DataFrame

Parameters

- **df** (*pd.DataFrame*) – with multi of columns.
- **group** (*str*) – the name of the feature group, i.e. the first level value of the group index.

class `qlib.data.dataset.processor.Processor`

fit (*df*: *pandas.core.frame.DataFrame* = *None*)

learn data processing parameters

Parameters df (*pd.DataFrame*) – When we fit and process data with processor one by one. The fit function relies on the output of previous processor, i.e. *df*.

is_for_infer () → bool

Is this processor usable for inference Some processors are not usable for inference.

Returns if it is usable for inference.

Return type bool

readonly () → bool

Does the processor treat the input data readonly (i.e. does not write the input data) when processing

Knowing the readonly information is helpful to the Handler to avoid unnecessary copy

config (***kwargs*)

configure the serializable object

Parameters

- **may include following keys** (*kwargs*) –
 - dump_all** [bool] will the object dump all object
 - exclude** [list] What attribute will not be dumped
 - include** [list] What attribute will be dumped
- **recursive** (*bool*) – will the configuration be recursive

```
class qlib.data.dataset.processor.DropnaProcessor (fields_group=None)
```

```
    __init__ (fields_group=None)
```

Initialize self. See help(type(self)) for accurate signature.

```
    readonly ()
```

Does the processor treat the input data readonly (i.e. does not write the input data) when processing

Knowing the readonly information is helpful to the Handler to avoid unnecessary copy

```
class qlib.data.dataset.processor.DropnaLabel (fields_group='label')
```

```
    __init__ (fields_group='label')
```

Initialize self. See help(type(self)) for accurate signature.

```
    is_for_infer () → bool
```

The samples are dropped according to label. So it is not usable for inference

```
class qlib.data.dataset.processor.DropCol (col_list=[])
```

```
    __init__ (col_list=[])
```

Initialize self. See help(type(self)) for accurate signature.

```
    readonly ()
```

Does the processor treat the input data readonly (i.e. does not write the input data) when processing

Knowing the readonly information is helpful to the Handler to avoid unnecessary copy

```
class qlib.data.dataset.processor.FilterCol (fields_group='feature', col_list=[])
```

```
    __init__ (fields_group='feature', col_list=[])
```

Initialize self. See help(type(self)) for accurate signature.

```
    readonly ()
```

Does the processor treat the input data readonly (i.e. does not write the input data) when processing

Knowing the readonly information is helpful to the Handler to avoid unnecessary copy

```
class qlib.data.dataset.processor.TanhProcess
```

Use tanh to process noise data

```
class qlib.data.dataset.processor.ProcessInf
```

Process infinity

```
class qlib.data.dataset.processor.Fillna (fields_group=None, fill_value=0)
```

Process NaN

```
    __init__ (fields_group=None, fill_value=0)
```

Initialize self. See help(type(self)) for accurate signature.

```
class qlib.data.dataset.processor.MinMaxNorm (fit_start_time, fit_end_time,  
                                              fields_group=None)
```

`__init__` (*fit_start_time, fit_end_time, fields_group=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*df: pandas.core.frame.DataFrame = None*)
learn data processing parameters

Parameters **df** (*pd.DataFrame*) – When we fit and process data with processor one by one. The fit function relies on the output of previous processor, i.e. *df*.

class qlib.data.dataset.processor.**ZScoreNorm** (*fit_start_time, fit_end_time, fields_group=None*)
ZScore Normalization

`__init__` (*fit_start_time, fit_end_time, fields_group=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*df: pandas.core.frame.DataFrame = None*)
learn data processing parameters

Parameters **df** (*pd.DataFrame*) – When we fit and process data with processor one by one. The fit function relies on the output of previous processor, i.e. *df*.

class qlib.data.dataset.processor.**RobustZScoreNorm** (*fit_start_time, fit_end_time, fields_group=None, clip_outlier=True*)
Robust ZScore Normalization

Use robust statistics for Z-Score normalization: $\text{mean}(x) = \text{median}(x)$ $\text{std}(x) = \text{MAD}(x) * 1.4826$

Reference: https://en.wikipedia.org/wiki/Median_absolute_deviation.

`__init__` (*fit_start_time, fit_end_time, fields_group=None, clip_outlier=True*)
Initialize self. See help(type(self)) for accurate signature.

fit (*df: pandas.core.frame.DataFrame = None*)
learn data processing parameters

Parameters **df** (*pd.DataFrame*) – When we fit and process data with processor one by one. The fit function relies on the output of previous processor, i.e. *df*.

class qlib.data.dataset.processor.**CSZScoreNorm** (*fields_group=None, method='zscore'*)
Cross Sectional ZScore Normalization

`__init__` (*fields_group=None, method='zscore'*)
Initialize self. See help(type(self)) for accurate signature.

class qlib.data.dataset.processor.**CSRankNorm** (*fields_group=None*)
Cross Sectional Rank Normalization. “Cross Sectional” is often used to describe data operations. The operations across different stocks are often called Cross Sectional Operation.

For example, CSRankNorm is an operation that grouping the data by each day and rank *across* all the stocks in each day.

Explanation about 3.46 & 0.5

```
import numpy as np
import pandas as pd
x = np.random.random(10000) # for any variable
x_rank = pd.Series(x).rank(pct=True) # if it is converted to rank, it will be a
↳ uniform distributed
x_rank_norm = (x_rank - x_rank.mean()) / x_rank.std() # Normally, we will
↳ normalize it to make it like normal distribution

x_rank.mean() # accounts for 0.5
1 / x_rank.std() # accounts for 3.46
```



```
__init__(fields_group=None)
    Initialize self. See help(type(self)) for accurate signature.
```

```
class qlib.data.dataset.processor.CSZFillna(fields_group=None)
    Cross Sectional Fill Nan
```

```
__init__(fields_group=None)
    Initialize self. See help(type(self)) for accurate signature.
```

```
class qlib.data.dataset.processor.HashStockFormat
    Process the storage of from df into hasing stock format
```

1.21.2 Contrib

Model

```
class qlib.model.base.BaseModel
    Modeling things

    predict(*args, **kwargs) → object
        Make predictions after modeling things
```

```
class qlib.model.base.Model
    Learnable Models

    fit(dataset: qlib.data.dataset.Dataset, reweighter: qlib.data.dataset.weight.Reweighter)
        Learn model from the base model
```

Note: The attribute names of learned model should *not* start with ‘_’. So that the model could be dumped to disk.

The following code example shows how to retrieve *x_train*, *y_train* and *w_train* from the *dataset*:

```
# get features and labels
df_train, df_valid = dataset.prepare(
    ["train", "valid"], col_set=["feature", "label"], data_
    ↪key=DataHandlerLP.DK_L
)
x_train, y_train = df_train["feature"], df_train["label"]
x_valid, y_valid = df_valid["feature"], df_valid["label"]

# get weights
try:
    wdf_train, wdf_valid = dataset.prepare(["train", "valid"], col_
    ↪set=["weight"],
                                         data_key=DataHandlerLP.
    ↪DK_L)
    w_train, w_valid = wdf_train["weight"], wdf_valid["weight"]
except KeyError as e:
    w_train = pd.DataFrame(np.ones_like(y_train.values), index=y_
    ↪train.index)
    w_valid = pd.DataFrame(np.ones_like(y_valid.values), index=y_
    ↪valid.index)
```

Parameters dataset (*Dataset*) – dataset will generate the processed data from model training.

predict (*dataset: qlib.data.dataset.Dataset, segment: Union[str, slice] = 'test'*) → object
 give prediction given Dataset

Parameters

- **dataset** (*Dataset*) – dataset will generate the processed dataset from model training.
- **segment** (*Text or slice*) – dataset will use this segment to prepare data. (default=test)

Returns

Return type Prediction results with certain type such as *pandas.Series*.

class qlib.model.base.**ModelFT**

Model (F)ine(t)unable

finetune (*dataset: qlib.data.dataset.Dataset*)

finetune model based given dataset

A typical use case of finetuning model with qlib.workflow.R

```
# start exp to train init model
with R.start(experiment_name="init models"):
    model.fit(dataset)
    R.save_objects(init_model=model)
    rid = R.get_recorder().id

# Finetune model based on previous trained model
with R.start(experiment_name="finetune model"):
    recorder = R.get_recorder(recorder_id=rid, experiment_name="init models")
    model = recorder.load_object("init_model")
    model.finetune(dataset, num_boost_round=10)
```

Parameters dataset (*Dataset*) – dataset will generate the processed dataset from model training.

Strategy

Evaluate

qlib.contrib.evaluate.**risk_analysis** (*r, N: int = None, freq: str = 'day'*)

Risk Analysis

Parameters

- **r** (*pandas.Series*) – daily return series.
- **N** (*int*) – scaler for annualizing information_ratio (day: 252, week: 50, month: 12), at least one of *N* and *freq* should exist
- **freq** (*str*) – analysis frequency used for calculating the scaler, at least one of *N* and *freq* should exist

qlib.contrib.evaluate.**indicator_analysis** (*df, method='mean'*)

analyze statistical time-series indicators of trading

Parameters

- **df** (*pandas.DataFrame*) –
 columns: like ['pa', 'pos', 'ffr', 'deal_amount', 'value'].

Necessary fields:

- 'pa' is the price advantage in trade indicators
- 'pos' is the positive rate in trade indicators
- 'ffr' is the fulfill rate in trade indicators

Optional fields:

- 'deal_amount' is the total deal deal_amount, only necessary when method is 'amount_weighted'
- 'value' is the total trade value, only necessary when method is 'value_weighted'

index: Index(datetime)

- **method** (*str*, *optional*) – statistics method of pa/ffr, by default “mean” - if method is ‘mean’, count the mean statistical value of each trade indicator - if method is ‘amount_weighted’, count the deal_amount weighted mean statistical value of each trade indicator - if method is ‘value_weighted’, count the value weighted mean statistical value of each trade indicator Note: statistics method of pos is always “mean”

Returns statistical value of each trade indicators

Return type pd.DataFrame

```
qlib.contrib.evaluate.backtest_daily(start_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp], end_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp], strategy: Union[str, dict, qlib.strategy.base.BaseStrategy], executor: Union[str, dict, qlib.backtest.executor.BaseExecutor] = None, account: Union[float, int, qlib.backtest.position.Position] = 100000000.0, benchmark: str = 'SH000300', exchange_kwargs: dict = None, pos_type: str = 'Position')
```

initialize the strategy and executor, then executor the backtest of daily frequency

Parameters

- **start_time** (*Union[str, pd.Timestamp]*) – closed start time for backtest
NOTE: This will be applied to the outmost executor’s calendar.
- **end_time** (*Union[str, pd.Timestamp]*) – closed end time for backtest
NOTE: This will be applied to the outmost executor’s calendar. E.g. Executor[day](Executor[1min]), setting *end_time* == 20XX0301 will include all the minutes on 20XX0301
- **strategy** (*Union[str, dict, BaseStrategy]*) – for initializing outermost portfolio strategy. Please refer to the docs of `init_instance_by_config` for more information.

E.g.

executor [Union[str, dict, BaseExecutor]] for initializing the outermost executor.

benchmark: **str** the benchmark for reporting.

account [Union[float, int, Position]] information for describing how to creating the account For *float* or *int*:

Using Account with only initial cash

For Position: Using Account with a Position

exchange_kwargs [dict] the kwargs for initializing Exchange E.g.

```
exchange_kwargs = {
    "freq": freq,
    "limit_threshold": None, # limit_threshold is None, using C.limit_
    ↪threshold
    "deal_price": None, # deal_price is None, using C.deal_price
    "open_cost": 0.0005,
    "close_cost": 0.0015,
    "min_cost": 5,
}
```

pos_type [str] the type of Position.

Returns

- **report_normal** (*pd.DataFrame*) – backtest report
- **positions_normal** (*pd.DataFrame*) – backtest positions

`qlib.contrib.evaluate.long_short_backtest` (*pred*, *topk=50*, *deal_price=None*, *shift=1*, *open_cost=0*, *close_cost=0*, *trade_unit=None*, *limit_threshold=None*, *min_cost=5*, *subscribe_fields=[]*, *extract_codes=False*)

A backtest for long-short strategy

Parameters

- **pred** – The trading signal produced on day *T*.
- **topk** – The short topk securities and long topk securities.
- **deal_price** – The price to deal the trading.
- **shift** – Whether to shift prediction by one day. The trading day will be T+1 if *shift==1*.
- **open_cost** – open transaction cost.
- **close_cost** – close transaction cost.
- **trade_unit** – 100 for China A.
- **limit_threshold** – limit move 0.1 (10%) for example, long and short with same limit.
- **min_cost** – min transaction cost.
- **subscribe_fields** – subscribe fields.
- **extract_codes** – bool. will we pass the codes extracted from the pred to the exchange. NOTE: This will be faster with offline qlib.

Returns

The result of backtest, it is represented by a dict. { “long”: long_returns(excess), “short”: short_returns(excess), “long_short”: long_short_returns }

Report

`qlib.contrib.report.analysis_position.report.report_graph` (*report_df*: *pandas.core.frame.DataFrame*, *show_notebook*: *bool = True*) → [*<class 'list'>*, *<class 'tuple'>*]

display backtest report

Example:

```
import qlib
import pandas as pd
from qlib.utils.time import Freq
from qlib.utils import flatten_dict
from qlib.backtest import backtest, executor
from qlib.contrib.evaluate import risk_analysis
from qlib.contrib.strategy import TopkDropoutStrategy

# init qlib
qlib.init(provider_uri=<qlib data dir>)

CSI300_BENCH = "SH000300"
FREQ = "day"
STRATEGY_CONFIG = {
    "topk": 50,
    "n_drop": 5,
    # pred_score, pd.Series
    "signal": pred_score,
}

EXECUTOR_CONFIG = {
    "time_per_step": "day",
    "generate_portfolio_metrics": True,
}

backtest_config = {
    "start_time": "2017-01-01",
    "end_time": "2020-08-01",
    "account": 100000000,
    "benchmark": CSI300_BENCH,
    "exchange_kwargs": {
        "freq": FREQ,
        "limit_threshold": 0.095,
        "deal_price": "close",
        "open_cost": 0.0005,
        "close_cost": 0.0015,
        "min_cost": 5,
    },
}

# strategy object
strategy_obj = TopkDropoutStrategy(**STRATEGY_CONFIG)
# executor object
executor_obj = executor.SimulatorExecutor(**EXECUTOR_CONFIG)
# backtest
portfolio_metric_dict, indicator_dict = ↵
↵backtest(executor=executor_obj, strategy=strategy_obj, ↵
↵**backtest_config)
```

(continues on next page)

(continued from previous page)

```

analysis_freq = "{0}{1}".format(*Freq.parse(FREQ))
# backtest info
report_normal_df, positions_normal = portfolio_metric_dict.
    ↳ get(analysis_freq)

qcr.analysis_position.report_graph(report_normal_df)

```

Parameters

- **report_df** – **df.index.name** must be **date**, **df.columns** must contain **return**, **turnover**, **cost**, **bench**.

	return	cost	bench	turnover
date				
2017-01-04	0.003421	0.000864	0.011693	0.576325
2017-01-05	0.000508	0.000447	0.000721	0.227882
2017-01-06	-0.003321	0.000212	-0.004322	0.102765
2017-01-09	0.006753	0.000212	0.006874	0.105864
2017-01-10	-0.000416	0.000440	-0.003350	0.208396

- **show_notebook** – whether to display graphics in notebook, the default is **True**.

Returns if **show_notebook** is **True**, display in notebook; else return **plotly.graph_objs.Figure** list.

```

qlib.contrib.report.analysis_position.score_ic.score_ic_graph(pred_label: pandas.core.frame.DataFrame,
    show_notebook:
    bool = True) →
    [<class 'list'>,
     <class 'tuple'>]

```

score IC

Example:

```

from qlib.data import D
from qlib.contrib.report import analysis_position
pred_df_dates = pred_df.index.get_level_values(level='datetime'
    ↳ ')
features_df = D.features(D.instruments('csi500'), ['Ref($close,
    ↳ -2)/Ref($close, -1)-1'], pred_df_dates.min(), pred_df_dates.
    ↳ max())
features_df.columns = ['label']
pred_label = pd.concat([features_df, pred], axis=1, sort=True).
    ↳ reindex(features_df.index)
analysis_position.score_ic_graph(pred_label)

```

Parameters

- **pred_label** – index is **pd.MultiIndex**, index name is **[instrument, datetime]**; columns names is **[score, label]**.

instrument	datetime	score	label
SH600004	2017-12-11	-0.013502	-0.013502
	2017-12-12	-0.072367	-0.072367
	2017-12-13	-0.068605	-0.068605
	2017-12-14	0.012440	0.012440
	2017-12-15	-0.102778	-0.102778

- **show_notebook** – whether to display graphics in notebook, the default is **True**.

Returns if show_notebook is True, display in notebook; else return `plotly.graph_objs.Figure` list.

`qlib.contrib.report.analysis_position.cumulative_return.cumulative_return_graph` (*position:* dict, re-port_normal: pan-das.core.frame. la-bel_data: pan-das.core.frame. show_notebook start_date=None end_date=None → It-er-able[plotly.graph

Backtest buy, sell, and holding cumulative return graph

Example:

```
from qlib.data import D
from qlib.contrib.evaluate import risk_analysis, backtest,
    ↳ long_short_backtest
from qlib.contrib.strategy import TopkDropoutStrategy

# backtest parameters
bparas = {}
bparas['limit_threshold'] = 0.095
bparas['account'] = 1000000000

sparas = {}
sparas['topk'] = 50
sparas['n_drop'] = 5
strategy = TopkDropoutStrategy(**sparas)

report_normal_df, positions = backtest(pred_df, strategy,
    ↳ **bparas)

pred_df_dates = pred_df.index.get_level_values(level='datetime
    ↳ ')
features_df = D.features(D.instruments('csi500'), ['Ref($close,
    ↳ -1)/$close - 1'], pred_df_dates.min(), pred_df_dates.max())
features_df.columns = ['label1']

qcr.analysis_position.cumulative_return_graph(positions,
    ↳ report_normal_df, features_df)
```

Graph desc:

- Axis X: Trading day.
- Axis Y:
- Above axis Y: $((Ref(\$close, -1)/\$close - 1) * weight).sum() / weight.sum()).cumsum()$.
- Below axis Y: Daily weight sum.

- In the **sell** graph, $y < 0$ stands for profit; in other cases, $y > 0$ stands for profit.
- In the **buy_minus_sell** graph, the **y** value of the **weight** graph at the bottom is $buy_weight + sell_weight$.
- In each graph, the **red line** in the histogram on the right represents the average.

Parameters

- **position** – position data
- **report_normal** –

	return	cost	bench	turnover
date				
2017-01-04	0.003421	0.000864	0.011693	0.576325
2017-01-05	0.000508	0.000447	0.000721	0.227882
2017-01-06	-0.003321	0.000212	-0.004322	0.102765
2017-01-09	0.006753	0.000212	0.006874	0.105864
2017-01-10	-0.000416	0.000440	-0.003350	0.208396

- **label_data** – *D.features* result; index is *pd.MultiIndex*, index name is [*instrument*, *datetime*]; columns names is [*label*].

The label **T** is the change from **T** to **T+1**, it is recommended to use `close`, example:
`D.features(D.instruments('csi500'), ['Ref($close, -1)/$close-1'])`

instrument	datetime	label
SH600004	2017-12-11	-0.013502
	2017-12-12	-0.072367
	2017-12-13	-0.068605
	2017-12-14	0.012440
	2017-12-15	-0.102778

Parameters

- **show_notebook** – True or False. If True, show graph in notebook, else return figures
- **start_date** – start date
- **end_date** – end date

Returns


```

qlib.contrib.report.analysis_position.risk_analysis.risk_analysis_graph(analysis_df:
    pandas.core.frame.DataFrame
    =
    None,
    re-
    port_normal_df:
    pandas.core.frame.DataFrame
    =
    None,
    re-
    port_long_short_df:
    pandas.core.frame.DataFrame
    =
    None,
    show_notebook:
    bool
    =
    True)
→
It-
er-
able[plotly.graph_objs._fig

```

Generate analysis graph and monthly analysis

Example:

```

import qlib
import pandas as pd
from qlib.utils.time import Freq
from qlib.utils import flatten_dict
from qlib.backtest import backtest, executor
from qlib.contrib.evaluate import risk_analysis
from qlib.contrib.strategy import TopkDropoutStrategy

# init qlib
qlib.init(provider_uri=<qlib data dir>)

CSI300_BENCH = "SH000300"
FREQ = "day"
STRATEGY_CONFIG = {
    "topk": 50,
    "n_drop": 5,
    # pred_score, pd.Series
    "signal": pred_score,
}

EXECUTOR_CONFIG = {
    "time_per_step": "day",
    "generate_portfolio_metrics": True,
}

backtest_config = {
    "start_time": "2017-01-01",
    "end_time": "2020-08-01",

```

(continues on next page)

(continued from previous page)

```

"account": 100000000,
"benchmark": CSI300_BENCH,
"exchange_kwargs": {
    "freq": FREQ,
    "limit_threshold": 0.095,
    "deal_price": "close",
    "open_cost": 0.0005,
    "close_cost": 0.0015,
    "min_cost": 5,
},
}

# strategy object
strategy_obj = TopkDropoutStrategy(**STRATEGY_CONFIG)
# executor object
executor_obj = executor.SimulatorExecutor(**EXECUTOR_CONFIG)
# backtest
portfolio_metric_dict, indicator_dict = \
    ↳backtest(executor=executor_obj, strategy=strategy_obj, \
    ↳**backtest_config)
analysis_freq = "{0}{1}".format(*Freq.parse(FREQ))
# backtest info
report_normal_df, positions_normal = portfolio_metric_dict.
    ↳get(analysis_freq)
analysis = dict()
analysis["excess_return_without_cost"] = risk_analysis(
    report_normal_df["return"] - report_normal_df["bench"], \
    ↳freq=analysis_freq
)
analysis["excess_return_with_cost"] = risk_analysis(
    report_normal_df["return"] - report_normal_df["bench"] - \
    ↳report_normal_df["cost"], freq=analysis_freq
)

analysis_df = pd.concat(analysis) # type: pd.DataFrame
analysis_position.risk_analysis_graph(analysis_df, report_
    ↳normal_df)

```

Parameters

- **analysis_df** – analysis data, index is **pd.MultiIndex**; columns names is **[risk]**.

excess_return_without_cost	mean	0.000692
	std	0.005374
	annualized_return	0.174495
	information_ratio	2.045576
	max_drawdown	-0.079103
excess_return_with_cost	mean	0.000499
	std	0.005372
	annualized_return	0.125625
	information_ratio	1.473152
	max_drawdown	-0.088263

- **report_normal_df** – **df.index.name** must be **date**, **df.columns** must contain **re-**
turn, turnover, cost, bench.

	return	cost	bench	turnover
date				
2017-01-04	0.003421	0.000864	0.011693	0.576325
2017-01-05	0.000508	0.000447	0.000721	0.227882
2017-01-06	-0.003321	0.000212	-0.004322	0.102765
2017-01-09	0.006753	0.000212	0.006874	0.105864
2017-01-10	-0.000416	0.000440	-0.003350	0.208396

- **report_long_short_df** – **df.index.name** must be **date**, **df.columns** contain **long**, **short**, **long_short**.

	long	short	long_short
date			
2017-01-04	-0.001360	0.001394	0.000034
2017-01-05	0.002456	0.000058	0.002514
2017-01-06	0.000120	0.002739	0.002859
2017-01-09	0.001436	0.001838	0.003273
2017-01-10	0.000824	-0.001944	-0.001120

- **show_notebook** – Whether to display graphics in a notebook, default **True**. If **True**, show graph in notebook If **False**, return graph figure

Returns

`qlib.contrib.report.analysis_position.rank_label.rank_label_graph` (*position:*
dict, label_data:
panel_data:
panel_data.DataFrame,
start_date=None,
end_date=None,
show_notebook=True)
 → *Iterable[plotly.graph_objs._figure.Figure]*

Ranking percentage of stocks buy, sell, and holding on the trading day. Average rank-ratio(similar to `sell_df['label'].rank(ascending=False) / len(sell_df)`) of daily trading

Example:

```
from qlib.data import D
from qlib.contrib.evaluate import backtest
from qlib.contrib.strategy import TopkDropoutStrategy

# backtest parameters
bparas = {}
bparas['limit_threshold'] = 0.095
bparas['account'] = 1000000000

sparas = {}
sparas['topk'] = 50
sparas['n_drop'] = 230
strategy = TopkDropoutStrategy(**sparas)

_, positions = backtest(pred_df, strategy, **bparas)

pred_df_dates = pred_df.index.get_level_values(level='datetime')
features_df = D.features(D.instruments('csi500'), ['Ref($close,
→ 1)/$close - 1'], pred_df_dates.min(), pred_df_dates.max())
```

(continues on next page)

(continued from previous page)

```
features_df.columns = ['label']

qcr.analysis_position.rank_label_graph(positions, features_df,
    pred_df_dates.min(), pred_df_dates.max())
```

Parameters

- **position** – position data; **qlib.backtest.backtest** result.
- **label_data** – **D.features** result; index is **pd.MultiIndex**, index name is **[instrument, datetime]**; columns names is **[label]**.

The label **T** is the change from **T** to **T+1**, it is recommended to use `close`, example: `D.features(D.instruments('csi500'), ['Ref($close, -1)/$close-1'])`.

instrument	datetime	label
SH600004	2017-12-11	-0.013502
	2017-12-12	-0.072367
	2017-12-13	-0.068605
	2017-12-14	0.012440
	2017-12-15	-0.102778

Parameters

- **start_date** – start date
- **end_date** – end_date
- **show_notebook** – **True** or **False**. If **True**, show graph in notebook, else return figures.

Returns

`qlib.contrib.report.analysis_model.analysis_model_performance.ic_figure(ic_df:`

`pan-`
`das.core.frame.DataFrame,`
`show_nature_day=True,`
`**kwargs)`
`→`
`plotly.graph_objs._figure.Fi`

IC figure

Parameters

- **ic_df** – ic DataFrame
- **show_nature_day** – whether to display the abscissa of non-trading day

Returns `plotly.graph_objs.Figure`

```

qlib.contrib.report.analysis_model.analysis_model_performance.model_performance_graph(pred_label,
pan-
das.co
lag:
int
=
1,
N:
int
=
5,
re-
verse=
rank=
graph_
list
=
['grou
'pred_
'pred_
show_
bool
=
True,
show_
→
[<class
'list'>,
<class
'tu-
ple'>]

```

Model performance

Parameters `pred_label` – index is `pd.MultiIndex`, index name is `[instrument, datetime]`; columns names is `**[score, label]**`. It is usually same as the label of model training(e.g. “`Ref($close, -2)/Ref($close, -1) - 1`”).

instrument	datetime	score	label
SH600004	2017-12-11	-0.013502	-0.013502
	2017-12-12	-0.072367	-0.072367
	2017-12-13	-0.068605	-0.068605
	2017-12-14	0.012440	0.012440
	2017-12-15	-0.102778	-0.102778

Parameters

- **lag** – `pred.groupby(level='instrument')['score'].shift(lag)`. It will be only used in the auto-correlation computing.
- **N** – group number, default 5.
- **reverse** – if `True`, `pred['score'] *= -1`.
- **rank** – if `True`, calculate rank ic.
- **graph_names** – graph names; default `['cumulative_return', 'pred_ic', 'pred_autocorr', 'pred_turnover']`.
- **show_notebook** – whether to display graphics in notebook, the default is `True`.

- **show_nature_day** – whether to display the abscissa of non-trading day.

Returns if show_notebook is True, display in notebook; else return *plotly.graph_objs.Figure* list.

1.21.3 Workflow

Experiment Manager

class qlib.workflow.expm.**ExpManager** (*uri: str, default_exp_name: Optional[str]*)

This is the *ExpManager* class for managing experiments. The API is designed similar to mlflow. (The link: https://mlflow.org/docs/latest/python_api/mlflow.html)

__init__ (*uri: str, default_exp_name: Optional[str]*)

Initialize self. See help(type(self)) for accurate signature.

start_exp (*, *experiment_id: Optional[str] = None, experiment_name: Optional[str] = None, recorder_id: Optional[str] = None, recorder_name: Optional[str] = None, uri: Optional[str] = None, resume: bool = False, **kwargs*)

Start an experiment. This method includes first get_or_create an experiment, and then set it to be active.

Parameters

- **experiment_id** (*str*) – id of the active experiment.
- **experiment_name** (*str*) – name of the active experiment.
- **recorder_id** (*str*) – id of the recorder to be started.
- **recorder_name** (*str*) – name of the recorder to be started.
- **uri** (*str*) – the current tracking URI.
- **resume** (*boolean*) – whether to resume the experiment and recorder.

Returns

Return type An active experiment.

end_exp (*recorder_status: str = 'SCHEDULED', **kwargs*)

End an active experiment.

Parameters

- **experiment_name** (*str*) – name of the active experiment.
- **recorder_status** (*str*) – the status of the active recorder of the experiment.

create_exp (*experiment_name: Optional[str] = None*)

Create an experiment.

Parameters **experiment_name** (*str*) – the experiment name, which must be unique.

Returns

- An experiment object.
- Raise
- —
- *ExpAlreadyExistError*

search_records (*experiment_ids=None, **kwargs*)

Get a pandas DataFrame of records that fit the search criteria of the experiment. Inputs are the search criteria user want to apply.

Returns

- A *pandas.DataFrame* of records, where each metric, parameter, and tag
- are expanded into their own columns named *metrics.*, *params.*, and *tags.*
- respectively. For records that don't have a particular metric, parameter, or tag, their
- value will be (NumPy) *Nan*, *None*, or *None* respectively.

get_exp (*, *experiment_id=None*, *experiment_name=None*, *create: bool = True*, *start: bool = False*)

Retrieve an experiment. This method includes getting an active experiment, and *get_or_create* a specific experiment.

When user specify experiment id and name, the method will try to return the specific experiment. When user does not provide recorder id or name, the method will try to return the current active experiment. The *create* argument determines whether the method will automatically create a new experiment according to user's specification if the experiment hasn't been created before.

- If *create* is *True*:
 - If *active experiment* exists:
 - * no id or name specified, return the active experiment.
 - * if id or name is specified, return the specified experiment. If no such exp found, create a new experiment with given id or name. If *start* is set to be *True*, the experiment is set to be active.
 - If *active experiment* not exists:
 - * no id or name specified, create a default experiment.
 - * if id or name is specified, return the specified experiment. If no such exp found, create a new experiment with given id or name. If *start* is set to be *True*, the experiment is set to be active.
- Else If *create* is *False*:
 - If *active experiment* exists:
 - * no id or name specified, return the active experiment.
 - * if id or name is specified, return the specified experiment. If no such exp found, raise *Error*.
 - If *active experiment* not exists:
 - * no id or name specified. If the default experiment exists, return it, otherwise, raise *Error*.
 - * if id or name is specified, return the specified experiment. If no such exp found, raise *Error*.

Parameters

- **experiment_id** (*str*) – id of the experiment to return.
- **experiment_name** (*str*) – name of the experiment to return.
- **create** (*boolean*) – create the experiment if it hasn't been created before.
- **start** (*boolean*) – start the new experiment if one is created.

Returns

Return type An experiment object.

delete_exp (*experiment_id=None, experiment_name=None*)

Delete an experiment.

Parameters

- **experiment_id** (*str*) – the experiment id.
- **experiment_name** (*str*) – the experiment name.

default_uri

Get the default tracking URI from qlib.config.C

uri

Get the default tracking URI or current URI.

Returns

Return type The tracking URI string.

set_uri (*uri: Optional[str] = None*)

Set the current tracking URI and the corresponding variables.

Parameters **uri** (*str*) –

list_experiments ()

List all the existing experiments.

Returns

Return type A dictionary (name -> experiment) of experiments information that being stored.

Experiment

class qlib.workflow.exp.**Experiment** (*id, name*)

This is the *Experiment* class for each experiment being run. The API is designed similar to mlflow. (The link: https://mlflow.org/docs/latest/python_api/mlflow.html)

__init__ (*id, name*)

Initialize self. See help(type(self)) for accurate signature.

start (*, *recorder_id=None, recorder_name=None, resume=False*)

Start the experiment and set it to be active. This method will also start a new recorder.

Parameters

- **recorder_id** (*str*) – the id of the recorder to be created.
- **recorder_name** (*str*) – the name of the recorder to be created.
- **resume** (*bool*) – whether to resume the first recorder

Returns

Return type An active recorder.

end (*recorder_status='SCHEDULED'*)

End the experiment.

Parameters **recorder_status** (*str*) – the status the recorder to be set with when ending (SCHEDULED, RUNNING, FINISHED, FAILED).

create_recorder (*recorder_name=None*)

Create a recorder for each experiment.

Parameters `recorder_name` (*str*) – the name of the recorder to be created.

Returns

Return type A recorder object.

search_records (***kwargs*)

Get a pandas DataFrame of records that fit the search criteria of the experiment. Inputs are the search criteria user want to apply.

Returns

- A *pandas.DataFrame* of records, where each metric, parameter, and tag
- are expanded into their own columns named *metrics.*, *params.**, and *tags.**
- respectively. For records that don't have a particular metric, parameter, or tag, their
- value will be (NumPy) *Nan*, *None*, or *None* respectively.

delete_recorder (*recorder_id*)

Create a recorder for each experiment.

Parameters `recorder_id` (*str*) – the id of the recorder to be deleted.

get_recorder (*recorder_id=None, recorder_name=None, create: bool = True, start: bool = False*)

Retrieve a Recorder for user. When user specify recorder id and name, the method will try to return the specific recorder. When user does not provide recorder id or name, the method will try to return the current active recorder. The *create* argument determines whether the method will automatically create a new recorder according to user's specification if the recorder hasn't been created before.

- If *create* is True:
 - If *active recorder* exists:
 - * no id or name specified, return the active recorder.
 - * if id or name is specified, return the specified recorder. If no such exp found, create a new recorder with given id or name. If *start* is set to be True, the recorder is set to be active.
 - If *active recorder* not exists:
 - * no id or name specified, create a new recorder.
 - * if id or name is specified, return the specified experiment. If no such exp found, create a new recorder with given id or name. If *start* is set to be True, the recorder is set to be active.
- Else If *create* is False:
 - If *active recorder* exists:
 - * no id or name specified, return the active recorder.
 - * if id or name is specified, return the specified recorder. If no such exp found, raise Error.
 - If *active recorder* not exists:
 - * no id or name specified, raise Error.
 - * if id or name is specified, return the specified recorder. If no such exp found, raise Error.

Parameters

- **recorder_id** (*str*) – the id of the recorder to be deleted.

- **recorder_name** (*str*) – the name of the recorder to be deleted.
- **create** (*boolean*) – create the recorder if it hasn't been created before.
- **start** (*boolean*) – start the new recorder if one is **created**.

Returns

Return type A recorder object.

list_recorders (*rtype: str = 'dict', **flt_kwargs*) → Union[List[qlib.workflow.recorder.Recorder], Dict[str, qlib.workflow.recorder.Recorder]]

List all the existing recorders of this experiment. Please first get the experiment instance before calling this method. If user want to use the method *R.list_recorders()*, please refer to the related API document in *QlibRecorder*.

flt_kwargs [dict] filter recorders by conditions e.g. `list_recorders(status=Recorder.STATUS_FI)`

Returns

if rtype == "dict": A dictionary (id -> recorder) of recorder information that being stored.

elif rtype == "list": A list of Recorder.

Return type The return type depend on *rtype*

Recorder

class `qlib.workflow.recorder.Recorder` (*experiment_id, name*)

This is the *Recorder* class for logging the experiments. The API is designed similar to mlflow. (The link: https://mlflow.org/docs/latest/python_api/mlflow.html)

The status of the recorder can be SCHEDULED, RUNNING, FINISHED, FAILED.

__init__ (*experiment_id, name*)

Initialize self. See `help(type(self))` for accurate signature.

save_objects (*local_path=None, artifact_path=None, **kwargs*)

Save objects such as prediction file or model checkpoints to the artifact URI. User can save object through keywords arguments (name:value).

Please refer to the docs of `qlib.workflow:R.save_objects`

Parameters

- **local_path** (*str*) – if provided, them save the file or directory to the artifact URI.
- **artifact_path=None** (*str*) – the relative path for the artifact to be stored in the URI.

load_object (*name*)

Load objects such as prediction file or model checkpoints.

Parameters **name** (*str*) – name of the file to be loaded.

Returns

Return type The saved object.

start_run()

Start running or resuming the Recorder. The return value can be used as a context manager within a *with* block; otherwise, you must call `end_run()` to terminate the current run. (See *ActiveRun* class in `mlflow`)

Returns

Return type An active running object (e.g. `mlflow.ActiveRun` object)

end_run()

End an active Recorder.

log_params(kwargs)**

Log a batch of params for the current run.

Parameters arguments (*keyword*) – key, value pair to be logged as parameters.

log_metrics(step=None, **kwargs)

Log multiple metrics for the current run.

Parameters arguments (*keyword*) – key, value pair to be logged as metrics.

set_tags(kwargs)**

Log a batch of tags for the current run.

Parameters arguments (*keyword*) – key, value pair to be logged as tags.

delete_tags(*keys)

Delete some tags from a run.

Parameters keys (*series of strs of the keys*) – all the name of the tag to be deleted.

list_artifacts(artifact_path: str = None)

List all the artifacts of a recorder.

Parameters artifact_path (*str*) – the relative path for the artifact to be stored in the URI.

Returns

Return type A list of artifacts information (name, path, etc.) that being stored.

list_metrics()

List all the metrics of a recorder.

Returns

Return type A dictionary of metrics that being stored.

list_params()

List all the params of a recorder.

Returns

Return type A dictionary of params that being stored.

list_tags()

List all the tags of a recorder.

Returns

Return type A dictionary of tags that being stored.

Record Template

class qlib.workflow.record_temp.**RecordTemp** (*recorder*)

This is the Records Template class that enables user to generate experiment results such as IC and backtest in a certain format.

save (***kwargs*)

It behaves the same as self.recorder.save_objects. But it is an easier interface because users don't have to care about *get_path* and *artifact_path*

__init__ (*recorder*)

Initialize self. See help(type(self)) for accurate signature.

generate (***kwargs*)

Generate certain records such as IC, backtest etc., and save them.

Parameters *kwargs* –

load (*name: str, parents: bool = True*)

It behaves the same as self.recorder.load_object. But it is an easier interface because users don't have to care about *get_path* and *artifact_path*

Parameters

- **name** (*str*) – the name for the file to be load.
- **parents** (*bool*) – Each recorder has different *artifact_path*. So parents recursively find the path in parents Sub classes has higher priority

Returns

Return type The stored records.

list ()

List the supported artifacts. Users don't have to consider self.get_path

Returns

Return type A list of all the supported artifacts.

check (*include_self: bool = False, parents: bool = True*)

Check if the records is properly generated and saved. It is useful in following examples - checking if the depended files complete before generating new things. - checking if the final files is completed

Parameters

- **include_self** (*bool*) – is the file generated by self included
- **parents** (*bool*) – will we check parents
- **Raise** –
- **-----** –
- **FileNotFoundError** –

:param : :type : whether the records are stored properly.

class qlib.workflow.record_temp.**SignalRecord** (*model=None, dataset=None, recorder=None*)

This is the Signal Record class that generates the signal prediction. This class inherits the **RecordTemp** class.

__init__ (*model=None, dataset=None, recorder=None*)

Initialize self. See help(type(self)) for accurate signature.

generate (***kwargs*)

Generate certain records such as IC, backtest etc., and save them.

Parameters *kwargs* –

list ()

List the supported artifacts. Users don't have to consider self.get_path

Returns

Return type A list of all the supported artifacts.

class qlib.workflow.record_temp.**ACRecordTemp** (*recorder, skip_existing=False*)

Automatically checking record template

__init__ (*recorder, skip_existing=False*)

Initialize self. See help(type(self)) for accurate signature.

generate (**args, **kwargs*)

automatically checking the files and then run the concrete generating task

class qlib.workflow.record_temp.**HFSignalRecord** (*recorder, **kwargs*)

This is the Signal Analysis Record class that generates the analysis results such as IC and IR. This class inherits the RecordTemp class.

depend_cls

alias of *SignalRecord*

__init__ (*recorder, **kwargs*)

Initialize self. See help(type(self)) for accurate signature.

generate ()

Generate certain records such as IC, backtest etc., and save them.

Parameters *kwargs* –

list ()

List the supported artifacts. Users don't have to consider self.get_path

Returns

Return type A list of all the supported artifacts.

class qlib.workflow.record_temp.**SigAnaRecord** (*recorder, ana_long_short=False, ann_scaler=252, label_col=0, skip_existing=False*)

This is the Signal Analysis Record class that generates the analysis results such as IC and IR. This class inherits the RecordTemp class.

depend_cls

alias of *SignalRecord*

__init__ (*recorder, ana_long_short=False, ann_scaler=252, label_col=0, skip_existing=False*)

Initialize self. See help(type(self)) for accurate signature.

list ()

List the supported artifacts. Users don't have to consider self.get_path

Returns

Return type A list of all the supported artifacts.

```
class qlib.workflow.record_temp.PortAnaRecord(recorder, config=None,
                                              risk_analysis_freq: Union[List[T],
                                                                    str] = None,
                                              indicator_analysis_freq: Union[List[T],
                                                                    str] = None,
                                              indicator_analysis_method=None,
                                              skip_existing=False, **kwargs)
```

This is the Portfolio Analysis Record class that generates the analysis results such as those of backtest. This class inherits the RecordTemp class.

The following files will be stored in recorder - report_normal.pkl & positions_normal.pkl:

- The return report and detailed positions of the backtest, returned by *qlib/contrib/evaluate.py:backtest*
- port_analysis.pkl : The risk analysis of your portfolio, returned by *qlib/contrib/evaluate.py:risk_analysis*

depend_cls

alias of *SignalRecord*

```
__init__(recorder, config=None, risk_analysis_freq: Union[List[T], str] = None,
          indicator_analysis_freq: Union[List[T], str] = None,
          indicator_analysis_method=None, skip_existing=False, **kwargs)
```

config["strategy"] [dict] define the strategy class as well as the kwargs.

config["executor"] [dict] define the executor class as well as the kwargs.

config["backtest"] [dict] define the backtest kwargs.

risk_analysis_freq [str|List[str]] risk analysis freq of report

indicator_analysis_freq [str|List[str]] indicator analysis freq of report

indicator_analysis_method [str, optional, default by None] the candidated values include 'mean', 'amount_weighted', 'value_weighted'

list()

List the supported artifacts. Users don't have to consider self.get_path

Returns

Return type A list of all the supported artifacts.

1.21.4 Task Management

TaskGen

TaskGenerator module can generate many tasks based on TaskGen and some task templates.

```
qlib.workflow.task.gen.task_generator(tasks, generators) → list
```

Use a list of TaskGen and a list of task templates to generate different tasks.

For examples:

There are 3 task templates a,b,c and 2 TaskGen A,B. A will generates 2 tasks from a template and B will generates 3 tasks from a template. task_generator([a, b, c], [A, B]) will finally generate $3 \times 2 \times 3 = 18$ tasks.

Parameters

- **tasks** (*List[dict]* or *dict*) – a list of task templates or a single task
- **generators** (*List[TaskGen]* or *TaskGen*) – a list of TaskGen or a single TaskGen

Returns a list of tasks

Return type list

class `qlib.workflow.task.gen.TaskGen`

The base class for generating different tasks

Example 1:

input: a specific task template and rolling steps

output: rolling version of the tasks

Example 2:

input: a specific task template and losses list

output: a set of tasks with different losses

generate (*task: dict*) → List[dict]

Generate different tasks based on a task template

Parameters *task* (*dict*) – a task template

Returns A list of tasks

Return type typing.List[dict]

`qlib.workflow.task.gen.handler_mod` (*task: dict, rolling_gen*)

Help to modify the handler end time when using RollingGen. It tries to handle the following case - Handler's data end_time is earlier than dataset's test_data's segments.

- To handle this, handler's data's end_time is extended.

If the handler's end_time is None, then it is not necessary to change its end time.

Parameters

- *task* (*dict*) – a task template
- *rg* (*RollingGen*) – an instance of RollingGen

`qlib.workflow.task.gen.trunc_segments` (*ta: qlib.workflow.task.utils.TimeAdjuster, segments: Dict[str, pandas._libs.tslibs.timestamps.Timestamp], days, test_key='test'*)

To avoid the leakage of future information, the segments should be truncated according to the test start_time

Note: This function will change segments **inplace**

class `qlib.workflow.task.gen.RollingGen` (*step: int = 40, rtype: str = 'expanding', ds_extra_mod_func: Union[None, Callable] = <function handler_mod>, test_key='test', train_key='train', trunc_days: int = None, task_copy_func: Callable = <function deepcopy>)*

__init__ (*step: int = 40, rtype: str = 'expanding', ds_extra_mod_func: Union[None, Callable] = <function handler_mod>, test_key='test', train_key='train', trunc_days: int = None, task_copy_func: Callable = <function deepcopy>)*

Generate tasks for rolling

Parameters

- *step* (*int*) – step to rolling
- *rtype* (*str*) – rolling type (expanding, sliding)
- *ds_extra_mod_func* (*Callable*) – A method like: handler_mod(task: dict, rg: RollingGen). Do some extra action after generating a task. For example, use handler_mod to modify the end time of the handler of a dataset.

- **trunc_days** (*int*) – trunc some data to avoid future information leakage
- **task_copy_func** (*Callable*) – the function to copy entire task. This is very useful when user want to share something between tasks

gen_following_tasks (*task*: *dict*, *test_end*: *pandas._libs.tslibs.timestamps.Timestamp*) → *List[dict]*
generating following rolling tasks for *task* until *test_end*

Parameters

- **task** (*dict*) – Qlib task format
- **test_end** (*pd.Timestamp*) – the latest rolling task includes *test_end*

Returns the following tasks of *task* ('*task* itself is excluded)

Return type *List[dict]*

generate (*task*: *dict*) → *List[dict]*

Converting the task into a rolling task.

Parameters **task** (*dict*) – A dict describing a task. For example.

```
DEFAULT_TASK = {
    "model": {
        "class": "LGBModel",
        "module_path": "qlib.contrib.model.gbdt",
    },
    "dataset": {
        "class": "DatasetH",
        "module_path": "qlib.data.dataset",
        "kwargs": {
            "handler": {
                "class": "Alpha158",
                "module_path": "qlib.contrib.data.handler",
                "kwargs": {
                    "start_time": "2008-01-01",
                    "end_time": "2020-08-01",
                    "fit_start_time": "2008-01-01",
                    "fit_end_time": "2014-12-31",
                    "instruments": "csi100",
                },
            },
        },
    },
    "segments": {
        "train": ("2008-01-01", "2014-12-31"),
        "valid": ("2015-01-01", "2016-12-20"), #_
        "test": ("2017-01-01", "2020-08-01"),
    },
    "record": [
        {
            "class": "SignalRecord",
            "module_path": "qlib.workflow.record_temp",
        },
    ],
}
```

→ Please avoid leaking the future test data into validation

Returns *List[dict]*

Return type a list of tasks

```
class qlib.workflow.task.gen.MultiHorizonGenBase (horizon: List[int] = [5], label_leak_n=2)
```

```
__init__ (horizon: List[int] = [5], label_leak_n=2)
```

This task generator tries to generate tasks for different horizons based on an existing task

Parameters

- **horizon** (*List [int]*) – the possible horizons of the tasks
- **label_leak_n** (*int*) – How many future days it will take to get complete label after the day making prediction For example: - User make prediction on day *T* (after getting the close price on *T*) - The label is the return of buying stock on *T + 1* and selling it on *T + 2* - the *label_leak_n* will be 2 (e.g. two days of information is leaked to leverage this sample)

```
set_horizon (task: dict, hr: int)
```

This method is designed to change the task **in place**

Parameters

- **task** (*dict*) – Qlib's task
- **hr** (*int*) – the horizon of task

```
generate (task: dict)
```

Generate different tasks based on a task template

Parameters **task** (*dict*) – a task template

Returns A list of tasks

Return type typing.List[dict]

TaskManager

TaskManager can fetch unused tasks automatically and manage the lifecycle of a set of tasks with error handling. These features can run tasks concurrently and ensure every task will be used only once. Task Manager will store all tasks in [MongoDB](#). Users **MUST** finished the configuration of [MongoDB](#) when using this module.

A task in TaskManager consists of 3 parts - tasks description: the desc will define the task - tasks status: the status of the task - tasks result: A user can get the task with the task description and task result.

```
class qlib.workflow.task.manage.TaskManager (task_pool: str)
```

Here is what will a task looks like when it created by TaskManager

```
{
    'def': pickle serialized task definition. using pickle will make it easier
    'filter': json-like data. This is for filtering the tasks.
    'status': 'waiting' | 'running' | 'done'
    'res': pickle serialized task result,
}
```

The tasks manager assumes that you will only update the tasks you fetched. The mongo fetch one and update will make it date updating secure.

This class can be used as a tool from commandline. Here are several examples. You can view the help of manage module with the following commands: `python -m qlib.workflow.task.manage -h` # show manual of manage module CLI `python -m qlib.workflow.task.manage wait -h` # show manual of the wait command of manage

```
python -m qlib.workflow.task.manage -t <pool_name> wait
python -m qlib.workflow.task.manage -t <pool_name> task_stat
```

Note: Assumption: the data in MongoDB was encoded and the data out of MongoDB was decoded

Here are four status which are:

STATUS_WAITING: waiting for training

STATUS_RUNNING: training

STATUS_PART_DONE: finished some step and waiting for next step

STATUS_DONE: all work done

__init__ (*task_pool: str*)

Init Task Manager, remember to make the statement of MongoDB url and database name firstly. A TaskManager instance serves a specific task pool. The static method of this module serves the whole MongoDB.

Parameters **task_pool** (*str*) – the name of Collection in MongoDB

static list () → list

List the all collection(task_pool) of the db.

Returns list

replace_task (*task, new_task*)

Use a new task to replace a old one

Parameters

- **task** – old task
- **new_task** – new task

insert_task (*task*)

Insert a task.

Parameters **task** – the task waiting for insert

Returns pymongo.results.InsertOneResult

insert_task_def (*task_def*)

Insert a task to task_pool

Parameters **task_def** (*dict*) – the task definition

Returns

Return type pymongo.results.InsertOneResult

create_task (*task_def_l, dry_run=False, print_nt=False*) → List[str]

If the tasks in task_def_l are new, then insert new tasks into the task_pool, and record inserted_id. If a task is not new, then just query its _id.

Parameters

- **task_def_l** (*list*) – a list of task
- **dry_run** (*bool*) – if insert those new tasks to task pool
- **print_nt** (*bool*) – if print new task

Returns a list of the _id of task_def_l

Return type List[str]

fetch_task (*query*={}, *status*='waiting') → dict
Use query to fetch tasks.

Parameters

- **query** (*dict*, *optional*) – query dict. Defaults to {}.
- **status** (*str*, *optional*) – [description]. Defaults to STATUS_WAITING.

Returns a task(document in collection) after decoding

Return type dict

safe_fetch_task (*query*={}, *status*='waiting')
Fetch task from task_pool using query with contextmanager

Parameters **query** (*dict*) – the dict of query

Returns dict

Return type a task(document in collection) after decoding

query (*query*={}, *decode*=True)

Query task in collection. This function may raise exception *pymongo.errors.CursorNotFound: cursor id not found* if it takes too long to iterate the generator

```
python -m qlib.workflow.task.manage -t <your task pool> query '{"_id":
"615498be837d0053acbc5d58"}'
```

Parameters

- **query** (*dict*) – the dict of query
- **decode** (*bool*) –

Returns dict

Return type a task(document in collection) after decoding

re_query (*_id*) → dict
Use _id to query task.

Parameters **_id** (*str*) – _id of a document

Returns a task(document in collection) after decoding

Return type dict

commit_task_res (*task*, *res*, *status*='done')
Commit the result to task['res'].

Parameters

- **task** (*[type]*) – [description]
- **res** (*object*) – the result you want to save
- **status** (*str*, *optional*) – STATUS_WAITING, STATUS_RUNNING, STATUS_DONE, STATUS_PART_DONE. Defaults to STATUS_DONE.

return_task (*task*, *status*='waiting')
Return a task to status. Always using in error handling.

Parameters

- **task** (*[type]*) – [description]

- **status** (*str*, *optional*) – STATUS_WAITING, STATUS_RUNNING, STATUS_DONE, STATUS_PART_DONE. Defaults to STATUS_WAITING.

remove (*query={}*)

Remove the task using query

Parameters **query** (*dict*) – the dict of query

task_stat (*query={}*) → dict

Count the tasks in every status.

Parameters **query** (*dict*, *optional*) – the query dict. Defaults to {}.

Returns dict

reset_waiting (*query={}*)

Reset all running task into waiting status. Can be used when some running task exit unexpected.

Parameters **query** (*dict*, *optional*) – the query dict. Defaults to {}.

prioritize (*task*, *priority: int*)

Set priority for task

Parameters

- **task** (*dict*) – The task query from the database
- **priority** (*int*) – the target priority

wait (*query={}*)

When multiprocessing, the main progress may fetch nothing from TaskManager because there are still some running tasks. So main progress should wait until all tasks are trained well by other progress or machines.

Parameters **query** (*dict*, *optional*) – the query dict. Defaults to {}.

`qlib.workflow.task.manage.run_task` (*task_func: Callable*, *task_pool: str*, *query: dict = {}*,
force_release: bool = False, *before_status: str = 'waiting'*,
after_status: str = 'done', ***kwargs*)

While the task pool is not empty (has WAITING tasks), use task_func to fetch and run tasks in task_pool

After running this method, here are 4 situations (before_status -> after_status):

STATUS_WAITING -> STATUS_DONE: use task["def"] as *task_func* param, it means that the task has not been started

STATUS_WAITING -> STATUS_PART_DONE: use task["def"] as *task_func* param

STATUS_PART_DONE -> STATUS_PART_DONE: use task["res"] as *task_func* param, it means that the task has been started but not completed

STATUS_PART_DONE -> STATUS_DONE: use task["res"] as *task_func* param

Parameters

- **task_func** (*Callable*) –
def (*task_def*, ***kwargs*) -> <res which will be committed> the function to run the task
- **task_pool** (*str*) – the name of the task pool (Collection in MongoDB)
- **query** (*dict*) – will use this dict to query task_pool when fetching task
- **force_release** (*bool*) – will the program force to release the resource
- **before_status** (*str:*) – the tasks in before_status will be fetched and trained.
Can be STATUS_WAITING, STATUS_PART_DONE.

- **after_status** (*str*:) – the tasks after trained will become after_status. Can be STATUS_WAITING, STATUS_PART_DONE.
- **kwargs** – the params for *task_func*

Trainer

The Trainer will train a list of tasks and return a list of model recorders. There are two steps in each Trainer including “train”(make model recorder) and “end_train”(modify model recorder).

This is a concept called DelayTrainer, which can be used in online simulating for parallel training. In DelayTrainer, the first step is only to save some necessary info to model recorders, and the second step which will be finished in the end can do some concurrent and time-consuming operations such as model fitting.

QLib offer two kinds of Trainer, TrainerR is the simplest way and TrainerRM is based on TaskManager to help manager tasks lifecycle automatically.

`qlib.model.trainer.begin_task_train(task_config: dict, experiment_name: str, recorder_name: str = None) → qlib.workflow.recorder.Recorder`

Begin task training to start a recorder and save the task config.

Parameters

- **task_config** (*dict*) – the config of a task
- **experiment_name** (*str*) – the name of experiment
- **recorder_name** (*str*) – the given name will be the recorder name. None for using rid.

Returns the model recorder

Return type *Recorder*

`qlib.model.trainer.end_task_train(rec: qlib.workflow.recorder.Recorder, experiment_name: str) → qlib.workflow.recorder.Recorder`

Finish task training with real model fitting and saving.

Parameters

- **rec** (*Recorder*) – the recorder will be resumed
- **experiment_name** (*str*) – the name of experiment

Returns the model recorder

Return type *Recorder*

`qlib.model.trainer.task_train(task_config: dict, experiment_name: str, recorder_name: str = None) → qlib.workflow.recorder.Recorder`

Task based training, will be divided into two steps.

Parameters

- **task_config** (*dict*) – The config of a task.
- **experiment_name** (*str*) – The name of experiment
- **recorder_name** (*str*) – The name of recorder

Returns *Recorder*

Return type The instance of the recorder

class `qlib.model.trainer.Trainer`

The trainer can train a list of models. There are Trainer and DelayTrainer, which can be distinguished by when it will finish real training.

`__init__()`

Initialize self. See `help(type(self))` for accurate signature.

`train(tasks: list, *args, **kwargs) → list`

Given a list of task definitions, begin training, and return the models.

For Trainer, it finishes real training in this method. For DelayTrainer, it only does some preparation in this method.

Parameters `tasks` – a list of tasks

Returns a list of models

Return type list

`end_train(models: list, *args, **kwargs) → list`

Given a list of models, finished something at the end of training if you need. The models may be Recorder, txt file, database, and so on.

For Trainer, it does some finishing touches in this method. For DelayTrainer, it finishes real training in this method.

Parameters `models` – a list of models

Returns a list of models

Return type list

`is_delay() → bool`

If Trainer will delay finishing `end_train`.

Returns if DelayTrainer

Return type bool

`has_worker() → bool`

Some trainer has backend worker to support parallel training This method can tell if the worker is enabled.

Returns if the worker is enabled

Return type bool

`worker()`

start the worker

Raises `NotImplementedError`: – If the worker is not supported

`class qlib.model.trainer.TrainerR(experiment_name: str = None, train_func: Callable = <function task_train>)`

Trainer based on (R)ecorder. It will train a list of tasks and return a list of model recorders in a linear way.

Assumption: models were defined by `task` and the results will be saved to `Recorder`.

`__init__(experiment_name: str = None, train_func: Callable = <function task_train>)`

Init TrainerR.

Parameters

- **experiment_name** (`str`, *optional*) – the default name of experiment.
- **train_func** (`Callable`, *optional*) – default training method. Defaults to `task_train`.

`train(tasks: list, train_func: Callable = None, experiment_name: str = None, **kwargs) →`

List[`qlib.workflow.recorder.Recorder`]

Given a list of ‘task’s and return a list of trained Recorder. The order can be guaranteed.

Parameters

- **tasks** (*list*) – a list of definitions based on *task* dict
- **train_func** (*Callable*) – the training method which needs at least *tasks* and *experiment_name*. None for the default training method.
- **experiment_name** (*str*) – the experiment name, None for use default name.
- **kwargs** – the params for train_func.

Returns a list of Recorders

Return type List[Recorder]

end_train (*models: list, **kwargs*) → List[qlib.workflow.recorder.Recorder]
Set STATUS_END tag to the recorders.

Parameters **models** (*list*) – a list of trained recorders.

Returns the same list as the param.

Return type List[Recorder]

```
class qlib.model.trainer.DelayTrainerR(experiment_name: str = None,
                                       train_func=<function begin_task_train>,
                                       end_train_func=<function end_task_train>)
```

A delayed implementation based on TrainerR, which means *train* method may only do some preparation and *end_train* method can do the real model fitting.

```
__init__(experiment_name: str = None, train_func=<function begin_task_train>,
         end_train_func=<function end_task_train>)
```

Init TrainerRM.

Parameters

- **experiment_name** (*str*) – the default name of experiment.
- **train_func** (*Callable, optional*) – default train method. Defaults to *begin_task_train*.
- **end_train_func** (*Callable, optional*) – default end_train method. Defaults to *end_task_train*.

end_train (*models, end_train_func=None, experiment_name: str = None, **kwargs*) → List[qlib.workflow.recorder.Recorder]

Given a list of Recorder and return a list of trained Recorder. This class will finish real data loading and model fitting.

Parameters

- **models** (*list*) – a list of Recorder, the tasks have been saved to them
- **end_train_func** (*Callable, optional*) – the end_train method which needs at least *recorder's* and *experiment_name*. Defaults to None for using self.end_train_func.
- **experiment_name** (*str*) – the experiment name, None for use default name.
- **kwargs** – the params for end_train_func.

Returns a list of Recorders

Return type List[Recorder]

```
class qlib.model.trainer.TrainerRM(experiment_name: str = None, task_pool: str = None,  
                                  train_func=<function task_train>, skip_run_task: bool =  
                                  False)
```

Trainer based on (R)ecorder and Task(M)anager. It can train a list of tasks and return a list of model recorders in a multiprocessing way.

Assumption: *task* will be saved to TaskManager and *task* will be fetched and trained from TaskManager

```
__init__(experiment_name: str = None, task_pool: str = None, train_func=<function task_train>,  
         skip_run_task: bool = False)  
    Init TrainerR.
```

Parameters

- **experiment_name** (*str*) – the default name of experiment.
- **task_pool** (*str*) – task pool name in TaskManager. None for use same name as *experiment_name*.
- **train_func** (*Callable, optional*) – default training method. Defaults to *task_train*.
- **skip_run_task** (*bool*) – If *skip_run_task* == True: Only run *task* in the worker. Otherwise skip *run_task*.

```
train(tasks: list, train_func: Callable = None, experiment_name: str = None, before_status: str =  
      'waiting', after_status: str = 'done', **kwargs) → List[qlib.workflow.recorder.Recorder]  
    Given a list of task's and return a list of trained Recorder. The order can be guaranteed.
```

This method defaults to a single process, but TaskManager offered a great way to parallel training. Users can customize their *train_func* to realize multiple processes or even multiple machines.

Parameters

- **tasks** (*list*) – a list of definitions based on *task* dict
- **train_func** (*Callable*) – the training method which needs at least *task*'s and *experiment_name*. None for the default training method.
- **experiment_name** (*str*) – the experiment name, None for use default name.
- **before_status** (*str*) – the tasks in *before_status* will be fetched and trained. Can be STATUS_WAITING, STATUS_PART_DONE.
- **after_status** (*str*) – the tasks after trained will become *after_status*. Can be STATUS_WAITING, STATUS_PART_DONE.
- **kwargs** – the params for *train_func*.

Returns a list of Recorders

Return type List[Recorder]

```
end_train(recs: list, **kwargs) → List[qlib.workflow.recorder.Recorder]  
    Set STATUS_END tag to the recorders.
```

Parameters **recs** (*list*) – a list of trained recorders.

Returns the same list as the param.

Return type List[Recorder]

```
worker(train_func: Callable = None, experiment_name: str = None)
```

The multiprocessing method for *train*. It can share a same *task_pool* with *train* and can run in other progress or other machines.

Parameters

- **train_func** (*Callable*) – the training method which needs at least *task*'s and *experiment_name*. None for the default training method.
- **experiment_name** (*str*) – the experiment name, None for use default name.

has_worker () → bool

Some trainer has backend worker to support parallel training This method can tell if the worker is enabled.

Returns if the worker is enabled

Return type bool

```
class qlib.model.trainer.DelayTrainerRM(experiment_name: str = None, task_pool: str =
                                         None, train_func=<function begin_task_train>,
                                         end_train_func=<function end_task_train>,
                                         skip_run_task: bool = False)
```

A delayed implementation based on TrainerRM, which means *train* method may only do some preparation and *end_train* method can do the real model fitting.

```
__init__(experiment_name: str = None, task_pool: str = None, train_func=<function begin_task_train>, end_train_func=<function end_task_train>, skip_run_task: bool = False)
```

Init DelayTrainerRM.

Parameters

- **experiment_name** (*str*) – the default name of experiment.
- **task_pool** (*str*) – task pool name in TaskManager. None for use same name as *experiment_name*.
- **train_func** (*Callable*, *optional*) – default train method. Defaults to *begin_task_train*.
- **end_train_func** (*Callable*, *optional*) – default end_train method. Defaults to *end_task_train*.
- **skip_run_task** (*bool*) – If *skip_run_task* == True: Only run_task in the worker. Otherwise skip run_task. E.g. Starting trainer on a CPU VM and then waiting tasks to be finished on GPU VMs.

```
train(tasks: list, train_func=None, experiment_name: str = None, **kwargs) →
List[qlib.workflow.recorder.Recorder]
```

Same as *train* of TrainerRM, after_status will be STATUS_PART_DONE.

Parameters

- **tasks** (*list*) – a list of definition based on *task* dict
- **train_func** (*Callable*) – the train method which need at least *task*'s and *experiment_name*. Defaults to None for using self.train_func.
- **experiment_name** (*str*) – the experiment name, None for use default name.

Returns a list of Recorders

Return type List[Recorder]

```
end_train(recs, end_train_func=None, experiment_name: str = None, **kwargs) →
List[qlib.workflow.recorder.Recorder]
```

Given a list of Recorder and return a list of trained Recorder. This class will finish real data loading and model fitting.

Parameters

- **recs** (*list*) – a list of Recorder, the tasks have been saved to them.
- **end_train_func** (*Callable, optional*) – the end_train method which need at least *recorder's* and *'experiment_name'*. Defaults to None for using self.end_train_func.
- **experiment_name** (*str*) – the experiment name, None for use default name.
- **kwargs** – the params for end_train_func.

Returns a list of Recorders

Return type List[[Recorder](#)]

worker (*end_train_func=None, experiment_name: str = None*)

The multiprocessing method for *end_train*. It can share a same task_pool with *end_train* and can run in other progress or other machines.

Parameters

- **end_train_func** (*Callable, optional*) – the end_train method which need at least *recorder's* and *'experiment_name'*. Defaults to None for using self.end_train_func.
- **experiment_name** (*str*) – the experiment name, None for use default name.

has_worker () → bool

Some trainer has backend worker to support parallel training This method can tell if the worker is enabled.

Returns if the worker is enabled

Return type bool

Collector

Collector module can collect objects from everywhere and process them such as merging, grouping, averaging and so on.

class qlib.workflow.task.collect.**Collector** (*process_list=[]*)

The collector to collect different results

__init__ (*process_list=[]*)

Init Collector.

Parameters **process_list** (*list or Callable*) – the list of processors or the instance of a processor to process dict.

collect () → dict

Collect the results and return a dict like {key: things}

Returns

the dict after collecting.

For example:

```
{“prediction”: pd.Series}
```

```
{“IC”: {“Xgboost”: pd.Series, “LSTM”: pd.Series}}
```

Return type dict

static process_collect (*collected_dict, process_list=[], *args, **kwargs*) → dict

Do a series of processing to the dict returned by collect and return a dict like {key: things} For example, you can group and ensemble.

Parameters

- **collected_dict** (*dict*) – the dict return by *collect*
- **process_list** (*list or Callable*) – the list of processors or the instance of a processor to process dict.
- **processor order is the same as the list order.** (*The*) – For example: [Group1(..., Ensemble1()), Group2(..., Ensemble2())]

Returns the dict after processing.

Return type dict

```
class qlib.workflow.task.collect.MergeCollector (collector_dict: Dict[str,
qlib.workflow.task.collect.Collector],
process_list: List[Callable] = [],
merge_func=None)
```

A collector to collect the results of other Collectors

For example:

We have 2 collector, which named A and B. A can collect {"prediction": pd.Series} and B can collect {"IC": {"Xgboost": pd.Series, "LSTM": pd.Series}}. Then after this class's collect, we can collect {"A_prediction": pd.Series, "B_IC": {"Xgboost": pd.Series, "LSTM": pd.Series}}

```
__init__(collector_dict: Dict[str, qlib.workflow.task.collect.Collector], process_list: List[Callable]
= [], merge_func=None)
Init MergeCollector.
```

Parameters

- **collector_dict** (*Dict[str, Collector]*) – the dict like {collector_key, Collector}
- **process_list** (*List[Callable]*) – the list of processors or the instance of processor to process dict.
- **merge_func** (*Callable*) – a method to generate outermost key. The given params are *collector_key* from *collector_dict* and *key* from every collector after collecting. None for using tuple to connect them, such as "ABC"+"a","b") -> ("ABC", ("a","b")).

collect () → dict

Collect all results of *collector_dict* and change the outermost key to a recombination key.

Returns the dict after collecting.

Return type dict

```
class qlib.workflow.task.collect.RecorderCollector (experiment, process_list=[],
rec_key_func=None,
rec_filter_func=None,
artifacts_path={'pred':
'pred.pkl'}, artifacts_key=None,
list_kwargs={}, status: Iterable[T_co] = {'FINISHED'})
```

```
__init__(experiment, process_list=[], rec_key_func=None, rec_filter_func=None, arti-
facts_path={'pred': 'pred.pkl'}, artifacts_key=None, list_kwargs={}, status: Iter-
able[T_co] = {'FINISHED'})
Init RecorderCollector.
```

Parameters

- **experiment** – (Experiment or str): an instance of an Experiment or the name of an Experiment (Callable): an callable function, which returns a list of experiments
- **process_list** (*list or Callable*) – the list of processors or the instance of a processor to process dict.
- **rec_key_func** (*Callable*) – a function to get the key of a recorder. If None, use recorder id.
- **rec_filter_func** (*Callable, optional*) – filter the recorder by return True or False. Defaults to None.
- **artifacts_path** (*dict, optional*) – The artifacts name and its path in Recorder. Defaults to {"pred": "pred.pkl", "IC": "sig_analysis/ic.pkl"}.
- **artifacts_key** (*str or List, optional*) – the artifacts key you want to get. If None, get all artifacts.
- **list_kwargs** (*str*) – arguments for list_recorders function.
- **status** (*Iterable*) – only collect recorders with specific status. None indicating collecting all the recorders

collect (*artifacts_key=None, rec_filter_func=None, only_exist=True*) → dict
Collect different artifacts based on recorder after filtering.

Parameters

- **artifacts_key** (*str or List, optional*) – the artifacts key you want to get. If None, use the default.
- **rec_filter_func** (*Callable, optional*) – filter the recorder by return True or False. If None, use the default.
- **only_exist** (*bool, optional*) – if only collect the artifacts when a recorder really has. If True, the recorder with exception when loading will not be collected. But if False, it will raise the exception.

Returns the dict after collected like {artifact: {rec_key: object}}

Return type dict

get_exp_name () → str
Get experiment name

Returns experiment name

Return type str

Group

Group can group a set of objects based on *group_func* and change them to a dict. After group, we provide a method to reduce them.

For example:

group: {(A,B,C1): object, (A,B,C2): object} -> {(A,B): {C1: object, C2: object}} reduce: {(A,B): {C1: object, C2: object}} -> {(A,B): object}

class qlib.model.ens.group.**Group** (*group_func=None, ens: qlib.model.ens.ensemble.Ensemble = None*)

Group the objects based on dict

`__init__` (*group_func=None, ens: qlib.model.ens.ensemble.Ensemble = None*)
Init Group.

Parameters

- **group_func** (*Callable, optional*) – Given a dict and return the group key and one of the group elements.
For example: {(A,B,C1): object, (A,B,C2): object} -> {(A,B): {C1: object, C2: object}}
- **to None.** (*Defaults*) –
- **ens** (*Ensemble, optional*) – If not None, do ensemble for grouped value after grouping.

group (**args, **kwargs*) → dict
Group a set of objects and change them to a dict.

For example: {(A,B,C1): object, (A,B,C2): object} -> {(A,B): {C1: object, C2: object}}

Returns grouped dict

Return type dict

reduce (**args, **kwargs*) → dict
Reduce grouped dict.

For example: {(A,B): {C1: object, C2: object}} -> {(A,B): object}

Returns reduced dict

Return type dict

class qlib.model.ens.group.**RollingGroup** (*ens=<qlib.model.ens.ensemble.RollingEnsemble object>*)

Group the rolling dict

group (*rolling_dict: dict*) → dict
Given an rolling dict likes {(A,B,R): things}, return the grouped dict likes {(A,B): {R:things}}

NOTE: There is an assumption which is the rolling key is at the end of the key tuple, because the rolling results always need to be ensemble firstly.

Parameters **rolling_dict** (*dict*) – an rolling dict. If the key is not a tuple, then do nothing.

Returns grouped dict

Return type dict

`__init__` (*ens=<qlib.model.ens.ensemble.RollingEnsemble object>*)
Init Group.

Parameters

- **group_func** (*Callable, optional*) – Given a dict and return the group key and one of the group elements.
For example: {(A,B,C1): object, (A,B,C2): object} -> {(A,B): {C1: object, C2: object}}
- **to None.** (*Defaults*) –
- **ens** (*Ensemble, optional*) – If not None, do ensemble for grouped value after grouping.

Ensemble

Ensemble module can merge the objects in an Ensemble. For example, if there are many submodels predictions, we may need to merge them into an ensemble prediction.

class `qlib.model.ens.ensemble.Ensemble`

Merge the ensemble_dict into an ensemble object.

For example: {Rollinga_b: object, Rollingb_c: object} -> object

When calling this class:

Args: ensemble_dict (dict): the ensemble dict like {name: things} waiting for merging

Returns: object: the ensemble object

class `qlib.model.ens.ensemble.SingleKeyEnsemble`

Extract the object if there is only one key and value in the dict. Make the result more readable. {Only key: Only value} -> Only value

If there is more than 1 key or less than 1 key, then do nothing. Even you can run this recursively to make dict more readable.

NOTE: Default runs recursively.

When calling this class:

Args: ensemble_dict (dict): the dict. The key of the dict will be ignored.

Returns: dict: the readable dict.

class `qlib.model.ens.ensemble.RollingEnsemble`

Merge a dict of rolling dataframe like *prediction* or *IC* into an ensemble.

NOTE: The values of dict must be pd.DataFrame, and have the index "datetime".

When calling this class:

Args: ensemble_dict (dict): a dict like {"A": pd.DataFrame, "B": pd.DataFrame}. The key of the dict will be ignored.

Returns: pd.DataFrame: the complete result of rolling.

class `qlib.model.ens.ensemble.AverageEnsemble`

Average and standardize a dict of same shape dataframe like *prediction* or *IC* into an ensemble.

NOTE: The values of dict must be pd.DataFrame, and have the index "datetime". If it is a nested dict, then flat it.

When calling this class:

Args: ensemble_dict (dict): a dict like {"A": pd.DataFrame, "B": pd.DataFrame}. The key of the dict will be ignored.

Returns: pd.DataFrame: the complete result of averaging and standardizing.

Utils

Some tools for task management.

`qlib.workflow.task.utils.get_mongodb()` → pymongo.database.Database

Get database in MongoDB, which means you need to declare the address and the name of a database at first.

For example:

Using `qlib.init()`:

```

    mongo_conf = { "task_url": task_url, # your MongoDB url "task_db_name":
                  task_db_name, # database name
    }
    qlib.init(..., mongo=mongo_conf)

```

After `qlib.init()`:

```

C["mongo"] = { "task_url" : "mongodb://localhost:27017", "task_db_name" :
              "rolling_db"
            }

```

Returns the Database instance

Return type Database

`qlib.workflow.task.utils.list_recorders(experiment, rec_filter_func=None)`

List all recorders which can pass the filter in an experiment.

Parameters

- **experiment** (*str* or *Experiment*) – the name of an Experiment or an instance
- **rec_filter_func** (*Callable*, *optional*) – return True to retain the given recorder. Defaults to None.

Returns a dict {rid: recorder} after filtering.

Return type dict

class `qlib.workflow.task.utils.TimeAdjuster(future=True, end_time=None)`

Find appropriate date and adjust date.

__init__ (*future=True*, *end_time=None*)

Initialize self. See `help(type(self))` for accurate signature.

set_end_time (*end_time=None*)

Set end time. None for use calendar's end time.

Parameters **end_time** –

get (*idx: int*)

Get datetime by index.

Parameters **idx** (*int*) – index of the calendar

max () → `pandas._libs.tslibs.timestamps.Timestamp`

Return the max calendar datetime

align_idx (*time_point*, *tp_type='start'*) → *int*

Align the index of *time_point* in the calendar.

Parameters

- **time_point** –
- **tp_type** (*str*) –

Returns **index**

Return type *int*

cal_interval (*time_point_A*, *time_point_B*) → *int*

Calculate the trading day interval (*time_point_A* - *time_point_B*)

Parameters

- **time_point_A** – *time_point_A*

- **time_point_B** – time_point_B (is the past of time_point_A)

Returns the interval between A and B

Return type int

align_time (*time_point*, *tp_type*='start') → pandas._libs.tslibs.timestamps.Timestamp

Align time_point to trade date of calendar

Parameters

- **time_point** – Time point
- **tp_type** – str time point type (“start”, “end”)

Returns pd.Timestamp

align_seg (*segment*: Union[dict, tuple]) → Union[dict, tuple]

Align the given date to the trade date

for example:

```
input: {'train': ('2008-01-01', '2014-12-31'), 'valid': ('2015-01-01
→', '2016-12-31'), 'test': ('2017-01-01', '2020-08-01')}

output: {'train': (Timestamp('2008-01-02 00:00:00'), Timestamp('
→'2014-12-31 00:00:00')),
        'valid': (Timestamp('2015-01-05 00:00:00'), Timestamp('2016-
→12-30 00:00:00')),
        'test': (Timestamp('2017-01-03 00:00:00'), Timestamp('2020-
→07-31 00:00:00'))}
```

Parameters **segment** –

Returns Union[dict, tuple]

Return type the start and end trade date (pd.Timestamp) between the given start and end date.

truncate (*segment*: tuple, *test_start*, *days*: int) → tuple

Truncate the segment based on the test_start date

Parameters

- **segment** (*tuple*) – time segment
- **test_start** –
- **days** (*int*) – The trading days to be truncated the data in this segment may need ‘days’ data *days* are based on the *test_start*. For example, if the label contains the information of 2 days in the near future, the prediction horizon 1 day. (e.g. the prediction target is $Ref(\$close, -2)/Ref(\$close, -1) - 1$) the days should be $2 + 1 == 3$ days.

Returns tuple

Return type new segment

shift (*seg*: tuple, *step*: int, *rtype*='sliding') → tuple

Shift the datetime of segment

If there are None (which indicates unbounded index) in the segment, this method will return None.

Parameters

- **seg** – datetime segment
- **step** (*int*) – rolling step
- **rtype** (*str*) – rolling type (“sliding” or “expanding”)

Returns tuple

Return type new segment

Raises KeyError: – shift will raise error if the index(both start and end) is out of self.cal

1.21.5 Online Serving

Online Manager

OnlineManager can manage a set of *Online Strategy* and run them dynamically.

With the change of time, the decisive models will be also changed. In this module, we call those contributing models *online* models. In every routine(such as every day or every minute), the *online* models may be changed and the prediction of them needs to be updated. So this module provides a series of methods to control this process.

This module also provides a method to simulate *Online Strategy* in history. Which means you can verify your strategy or find a better one.

There are 4 total situations for using different trainers in different situations:

Situations	Description
Online + Trainer	When you want to do a REAL routine, the Trainer will help you train the models. It will train models task by task and strategy by strategy.
Online + Delay-Trainer	DelayTrainer will skip concrete training until all tasks have been prepared by different strategies. It makes users can parallelly train all tasks at the end of <i>routine</i> or <i>first_train</i> . Otherwise, these functions will get stuck when each strategy prepare tasks.
Simulation + Trainer	It will behave in the same way as <i>Online + Trainer</i> . The only difference is that it is for simulation/backtesting instead of online trading
Simulation + Delay-Trainer	When your models don't have any temporal dependence, you can use DelayTrainer for the ability to multitasking. It means all tasks in all routines can be REAL trained at the end of simulating. The signals will be prepared well at different time segments (based on whether or not any new model is online).

Here is some pseudo code the demonstrate the workflow of each situation

For simplicity

- Only one strategy is used in the strategy
- *update_online_pred* is only called in the online mode and is ignored

1) Online + Trainer

```
tasks = first_train()
models = trainer.train(tasks)
trainer.end_train(models)
for day in online_trading_days:
    # OnlineManager.routine
    models = trainer.train(strategy.prepare_tasks()) # for each strategy
```

(continues on next page)

(continued from previous page)

```

strategy.prepare_online_models(models)  # for each strategy

trainer.end_train(models)
prepare_signals()  # prepare trading signals daily

```

Online + DelayTrainer: the workflow is the same as *Online + Trainer*.

2) Simulation + DelayTrainer

```

# simulate
tasks = first_train()
models = trainer.train(tasks)
for day in historical_calendars:
    # OnlineManager.routine
    models = trainer.train(strategy.prepare_tasks())  # for each strategy
    strategy.prepare_online_models(models)  # for each strategy
# delay_prepare()
# FIXME: Currently the delay_prepare is not implemented in a proper way.
trainer.end_train(<for all previous models>)
prepare_signals()

```

Can we simplify current workflow? - Can reduce the number of state of tasks?

- For each task, we have three phases (i.e. task, partly trained task, final trained task)

```

class qlib.workflow.online.manager.OnlineManager (strategies:
                                                    Union[qlib.workflow.online.strategy.OnlineStrategy,
List[qlib.workflow.online.strategy.OnlineStrategy]],
trainer: qlib.model.trainer.Trainer =
None, begin_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp]
= None, freq='day')

```

OnlineManager can manage online models with *Online Strategy*. It also provides a history recording of which models are online at what time.

```

__init__(strategies: Union[qlib.workflow.online.strategy.OnlineStrategy,
List[qlib.workflow.online.strategy.OnlineStrategy]], trainer: qlib.model.trainer.Trainer
= None, begin_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp] = None,
freq='day')

```

Init OnlineManager. One OnlineManager must have at least one OnlineStrategy.

Parameters

- **strategies** (*Union[OnlineStrategy, List[OnlineStrategy]]*) – an instance of OnlineStrategy or a list of OnlineStrategy
- **begin_time** (*Union[str, pd.Timestamp], optional*) – the OnlineManager will begin at this time. Defaults to None for using the latest date.
- **trainer** (*Trainer*) – the trainer to train task. None for using TrainerR.
- **freq** (*str, optional*) – data frequency. Defaults to “day”.

```

first_train (strategies: List[qlib.workflow.online.strategy.OnlineStrategy] = None, model_kwargs:
dict = {})

```

Get tasks from every strategy’s `first_tasks` method and train them. If using DelayTrainer, it can finish training all together after every strategy’s `first_tasks`.

Parameters

- **strategies** (*List[OnlineStrategy]*) – the strategies list (need this param when adding strategies). None for use default strategies.
- **model_kwargs** (*dict*) – the params for *prepare_online_models*

routine (*cur_time: Union[str, pandas._libs.tslibs.timestamps.Timestamp] = None, task_kwargs: dict = {}, model_kwargs: dict = {}, signal_kwargs: dict = {}*)

Typical update process for every strategy and record the online history.

The typical update process after a routine, such as day by day or month by month. The process is: Update predictions -> Prepare tasks -> Prepare online models -> Prepare signals.

If using DelayTrainer, it can finish training all together after every strategy's prepare_tasks.

Parameters

- **cur_time** (*Union[str, pd.Timestamp], optional*) – run routine method in this time. Defaults to None.
- **task_kwargs** (*dict*) – the params for *prepare_tasks*
- **model_kwargs** (*dict*) – the params for *prepare_online_models*
- **signal_kwargs** (*dict*) – the params for *prepare_signals*

get_collector (***kwargs*) → *qlib.workflow.task.collect.MergeCollector*

Get the instance of *Collector* to collect results from every strategy. This collector can be a basis as the signals preparation.

Parameters ***kwargs* – the params for *get_collector*.

Returns the collector to merge other collectors.

Return type *MergeCollector*

add_strategy (*strategies: Union[qlib.workflow.online.strategy.OnlineStrategy, List[qlib.workflow.online.strategy.OnlineStrategy]]*)

Add some new strategies to OnlineManager.

Parameters **strategy** (*Union[OnlineStrategy, List[OnlineStrategy]]*)
– a list of OnlineStrategy

prepare_signals (*prepare_func: Callable = <qlib.model.ens.ensemble.AverageEnsemble object>, over_write=False*)

After preparing the data of the last routine (a box in box-plot) which means the end of the routine, we can prepare trading signals for the next routine.

NOTE: Given a set prediction, all signals before these prediction end times will be prepared well.

Even if the latest signal already exists, the latest calculation result will be overwritten.

Note: Given a prediction of a certain time, all signals before this time will be prepared well.

Parameters

- **prepare_func** (*Callable, optional*) – Get signals from a dict after collecting. Defaults to *AverageEnsemble()*, the results collected by *MergeCollector* must be {xxx:pred}.
- **over_write** (*bool, optional*) – If True, the new signals will overwrite. If False, the new signals will append to the end of signals. Defaults to False.

Returns the signals.

Return type `pd.DataFrame`

get_signals () → Union[pandas.core.series.Series, pandas.core.frame.DataFrame]
Get prepared online signals.

Returns `pd.Series` for only one signals every datetime. `pd.DataFrame` for multiple signals, for example, buy and sell operations use different trading signals.

Return type Union[pd.Series, pd.DataFrame]

simulate (*end_time=None, frequency='day', task_kwargs={}, model_kwargs={}, signal_kwargs={}*)
→ Union[pandas.core.series.Series, pandas.core.frame.DataFrame]

Starting from the current time, this method will simulate every routine in `OnlineManager` until the end time.

Considering the parallel training, the models and signals can be prepared after all routine simulating.

The delay training way can be `DelayTrainer` and the delay preparing signals way can be `delay_prepare`.

Parameters

- **end_time** – the time the simulation will end
- **frequency** – the calendar frequency
- **task_kwargs** (*dict*) – the params for *prepare_tasks*
- **model_kwargs** (*dict*) – the params for *prepare_online_models*
- **signal_kwargs** (*dict*) – the params for *prepare_signals*

Returns `pd.Series` for only one signals every datetime. `pd.DataFrame` for multiple signals, for example, buy and sell operations use different trading signals.

Return type Union[pd.Series, pd.DataFrame]

delay_prepare (*model_kwargs={}, signal_kwargs={}*)

Prepare all models and signals if something is waiting for preparation.

Parameters

- **model_kwargs** – the params for *end_train*
- **signal_kwargs** – the params for *prepare_signals*

Online Strategy

`OnlineStrategy` module is an element of online serving.

class `qlib.workflow.online.strategy.OnlineStrategy` (*name_id: str*)

`OnlineStrategy` is working with *Online Manager*, responding to how the tasks are generated, the models are updated and signals are prepared.

__init__ (*name_id: str*)

Init `OnlineStrategy`. This module **MUST** use `Trainer` to finishing model training.

Parameters

- **name_id** (*str*) – a unique name or id.
- **trainer** (`Trainer`, *optional*) – a instance of `Trainer`. Defaults to `None`.

prepare_tasks (*cur_time*, ***kwargs*) → List[dict]

After the end of a routine, check whether we need to prepare and train some new tasks based on *cur_time* (None for latest).. Return the new tasks waiting for training.

You can find the last online models by `OnlineTool.online_models`.

prepare_online_models (*trained_models*, *cur_time=None*) → List[object]

Select some models from trained models and set them to online models. This is a typical implementation to online all trained models, you can override it to implement the complex method. You can find the last online models by `OnlineTool.online_models` if you still need them.

NOTE: Reset all online models to trained models. If there are no trained models, then do nothing.

NOTE: Current implementation is very naive. Here is a more complex situation which is more closer to the practical scenarios. 1. Train new models at the day before *test_start* (at time stamp *T*) 2. Switch models at the *test_start* (at time timestamp *T + 1* typically)

Parameters

- **models** (*list*) – a list of models.
- **cur_time** (*pd.DataFrame*) – current time from OnlineManger. None for the latest.

Returns a list of online models.

Return type List[object]

first_tasks () → List[dict]

Generate a series of tasks firstly and return them.

get_collector () → `qlib.workflow.task.collect.Collector`

Get the instance of `Collector` to collect different results of this strategy.

For example:

- 1) collect predictions in Recorder
- 2) collect signals in a txt file

Returns Collector

```
class qlib.workflow.online.strategy.RollingStrategy (name_id: str,
                                                    task_template: Union[dict,
                                                                    List[dict]],
                                                    rolling_gen: qlib.workflow.task.gen.RollingGen)
```

This example strategy always uses the latest rolling model sas online models.

```
__init__ (name_id: str, task_template: Union[dict, List[dict]], rolling_gen:
          qlib.workflow.task.gen.RollingGen)
Init RollingStrategy.
```

Assumption: the str of *name_id*, the experiment name, and the trainer's experiment name are the same.

Parameters

- **name_id** (*str*) – a unique name or id. Will be also the name of the Experiment.
- **task_template** (*Union[dict, List[dict]]*) – a list of task_template or a single template, which will be used to generate many tasks using *rolling_gen*.
- **rolling_gen** (`RollingGen`) – an instance of `RollingGen`

get_collector (*process_list*=[<*qlib.model.ens.group.RollingGroup* object>], *rec_key_func*=None, *rec_filter_func*=None, *artifacts_key*=None)

Get the instance of **Collector** to collect results. The returned collector must distinguish results in different models.

Assumption: the models can be distinguished based on the model name and rolling test segments. If you do not want this assumption, please implement your method or use another *rec_key_func*.

Parameters

- **rec_key_func** (*Callable*) – a function to get the key of a recorder. If None, use recorder id.
- **rec_filter_func** (*Callable*, *optional*) – filter the recorder by return True or False. Defaults to None.
- **artifacts_key** (*List[str]*, *optional*) – the artifacts key you want to get. If None, get all artifacts.

first_tasks () → *List[dict]*

Use *rolling_gen* to generate different tasks based on *task_template*.

Returns a list of tasks

Return type *List[dict]*

prepare_tasks (*cur_time*) → *List[dict]*

Prepare new tasks based on *cur_time* (None for the latest).

You can find the last online models by *OnlineToolR.online_models*.

Returns a list of new tasks.

Return type *List[dict]*

Online Tool

OnlineTool is a module to set and unset a series of *online* models. The *online* models are some decisive models in some time points, which can be changed with the change of time. This allows us to use efficient submodels as the market-style changing.

class *qlib.workflow.online.utils.OnlineTool*

OnlineTool will manage *online* models in an experiment that includes the model recorders.

__init__ ()

Init OnlineTool.

set_online_tag (*tag*, *recorder*: *Union[list, object]*)

Set *tag* to the model to sign whether online.

Parameters

- **tag** (*str*) – the tags in *ONLINE_TAG*, *OFFLINE_TAG*
- **recorder** (*Union[list, object]*) – the model's recorder

get_online_tag (*recorder*: *object*) → *str*

Given a model recorder and return its online tag.

Parameters **recorder** (*Object*) – the model's recorder

Returns the online tag

Return type *str*

reset_online_tag (*recorder: Union[list, object]*)

Offline all models and set the recorders to ‘online’.

Parameters **recorder** (*Union[list, object]*) – the recorder you want to reset to ‘online’.

online_models () → list

Get current *online* models

Returns a list of *online* models.

Return type list

update_online_pred (*to_date=None*)

Update the predictions of *online* models to *to_date*.

Parameters **to_date** (*pd.Timestamp*) – the pred before this date will be updated. None for updating to the latest.

class qlib.workflow.online.utils.**OnlineToolR** (*default_exp_name: str = None*)

The implementation of OnlineTool based on (R)ecorder.

__init__ (*default_exp_name: str = None*)

Init OnlineToolR.

Parameters **default_exp_name** (*str*) – the default experiment name.

set_online_tag (*tag, recorder: Union[qlib.workflow.recorder.Recorder, List[T]]*)

Set *tag* to the model’s recorder to sign whether online.

Parameters

- **tag** (*str*) – the tags in *ONLINE_TAG*, *NEXT_ONLINE_TAG*, *OFFLINE_TAG*
- **recorder** (*Union[Recorder, List]*) – a list of Recorder or an instance of Recorder

get_online_tag (*recorder: qlib.workflow.recorder.Recorder*) → str

Given a model recorder and return its online tag.

Parameters **recorder** (*Recorder*) – an instance of recorder

Returns the online tag

Return type str

reset_online_tag (*recorder: Union[qlib.workflow.recorder.Recorder, List[T]], exp_name: str = None*)

Offline all models and set the recorders to ‘online’.

Parameters

- **recorder** (*Union[Recorder, List]*) – the recorder you want to reset to ‘online’.
- **exp_name** (*str*) – the experiment name. If None, then use *default_exp_name*.

online_models (*exp_name: str = None*) → list

Get current *online* models

Parameters **exp_name** (*str*) – the experiment name. If None, then use *default_exp_name*.

Returns a list of *online* models.

Return type list

update_online_pred (*to_date=None, from_date=None, exp_name: str = None*)

Update the predictions of online models to *to_date*.

Parameters

- **to_date** (*pd.Timestamp*) – the pred before this date will be updated. None for updating to latest time in Calendar.
- **exp_name** (*str*) – the experiment name. If None, then use *default_exp_name*.

RecordUpdater

Updater is a module to update artifacts such as predictions when the stock data is updating.

class `qlib.workflow.online.update.RMDLoader` (*rec: qlib.workflow.recorder.Recorder*)
Recorder Model Dataset Loader

__init__ (*rec: qlib.workflow.recorder.Recorder*)
Initialize self. See help(type(self)) for accurate signature.

get_dataset (*start_time, end_time, segments=None, unprepared_dataset: Optional[qlib.data.dataset.DatasetH] = None*) → `qlib.data.dataset.DatasetH` *Optional*
Load, config and setup dataset.

This dataset is for inference.

Parameters

- **start_time** – the *start_time* of underlying data
- **end_time** – the *end_time* of underlying data
- **segments** – dict the segments config for dataset Due to the time series dataset (TSDatasetH), the test segments maybe different from *start_time* and *end_time*
- **unprepared_dataset** – Optional[DatasetH] if user don't want to load dataset from recorder, please specify user's dataset

Returns the instance of DatasetH

Return type *DatasetH*

class `qlib.workflow.online.update.RecordUpdater` (*record: qlib.workflow.recorder.Recorder, *args, **kwargs*)

Update a specific recorders

__init__ (*record: qlib.workflow.recorder.Recorder, *args, **kwargs*)
Initialize self. See help(type(self)) for accurate signature.

update (**args, **kwargs*)
Update info for specific recorder

class `qlib.workflow.online.update.DSBasedUpdater` (*record: qlib.workflow.recorder.Recorder, to_date=None, from_date=None, hist_ref: Optional[int] = None, freq='day', fname='pred.pkl', loader_cls: type = <class 'qlib.workflow.online.update.RMDLoader'>*)

Dataset-Based Updater - Providing updating feature for Updating data based on Qlib Dataset

Assumption - Based on Qlib dataset - The data to be updated is a multi-level index pd.DataFrame. For example label , prediction.

LABEL0

datetime instrument 2021-05-10 SH600000 0.006965

SH600004 0.003407

... ... 2021-05-28 SZ300498 0.015748

SZ300676 -0.001321

```
__init__(record: qlib.workflow.recorder.Recorder, to_date=None, from_date=None, hist_ref:
Optional[int] = None, freq='day', fname='pred.pkl', loader_cls: type = <class
'qlib.workflow.online.update.RMDLoader'>)
```

Init PredUpdater.

Expected behavior in following cases: - if *to_date* is greater than the max date in the calendar, the data will be updated to the latest date - if there are data before *from_date* or after *to_date*, only the data between *from_date* and *to_date* are affected.

Parameters

- **record** – Recorder
- **to_date** – update to prediction to the *to_date* if *to_date* is None:
data will updated to the latest date.
- **from_date** – the update will start from *from_date* if *from_date* is None:
the updating will occur on the next tick after the latest data in historical data
- **hist_ref** – int Sometimes, the dataset will have historical depends. Leave the problem to users to set the length of historical dependency If user doesn't specify this parameter, Updater will try to load dataset to automatically determine the *hist_ref*

Note: the start_time is not included in the *hist_ref*

- **loader_cls** – type the class to load the model and dataset

```
prepare_data (unprepared_dataset: Optional[qlib.data.dataset.DatasetH] = None) →
qlib.data.dataset.DatasetH
```

Load dataset - if *unprepared_dataset* is specified, then prepare the dataset directly - Otherwise,

Separating this function will make it easier to reuse the dataset

Returns the instance of DatasetH

Return type *DatasetH*

```
update (dataset: qlib.data.dataset.DatasetH = None, write: bool = True, ret_new: bool = False) →
Optional[object]
```

Parameters

- **dataset** (*DatasetH*) – DatasetH: the instance of DatasetH. None for prepare it again.
- **write** (*bool*) – will the the write action be executed
- **ret_new** (*bool*) – will the updated data be returned

Returns the updated dataset

Return type Optional[object]

get_update_data (*dataset: qlib.data.dataset.Dataset*) → pandas.core.frame.DataFrame
return the updated data based on the given dataset

The difference between *get_update_data* and *update* - *update_date* only include some data specific feature
- *update* include some general routine steps(e.g. prepare dataset, checking)

```
class qlib.workflow.online.update.PredUpdater (record: qlib.workflow.recorder.Recorder,
                                              to_date=None,          from_date=None,
                                              hist_ref: Optional[int] = None,
                                              freq='day',          fname='pred.pkl',
                                              loader_cls: type = <class
                                              'qlib.workflow.online.update.RMDLoader'>)
```

Update the prediction in the Recorder

get_update_data (*dataset: qlib.data.dataset.Dataset*) → pandas.core.frame.DataFrame
return the updated data based on the given dataset

The difference between *get_update_data* and *update* - *update_date* only include some data specific feature
- *update* include some general routine steps(e.g. prepare dataset, checking)

```
class qlib.workflow.online.update.LabelUpdater (record: qlib.workflow.recorder.Recorder,
                                              to_date=None, **kwargs)
```

Update the label in the recorder

Assumption - The label is generated from record_temp.SignalRecord.

__init__ (*record: qlib.workflow.recorder.Recorder, to_date=None, **kwargs*)
Init PredUpdater.

Expected behavior in following cases: - if *to_date* is greater than the max date in the calendar, the data will be updated to the latest date - if there are data before *from_date* or after *to_date*, only the data between *from_date* and *to_date* are affected.

Parameters

- **record** – Recorder
- **to_date** – update to prediction to the *to_date* if *to_date* is None:
data will updated to the latest date.
- **from_date** – the update will start from *from_date* if *from_date* is None:
the updating will occur on the next tick after the latest data in historical data
- **hist_ref** – int Sometimes, the dataset will have historical depends. Leave the problem to users to set the length of historical dependency If user doesn't specify this parameter, Updater will try to load dataset to automatically determine the *hist_ref*

Note: the start_time is not included in the *hist_ref*

- **loader_cls** – type the class to load the model and dataset

get_update_data (*dataset: qlib.data.dataset.Dataset*) → pandas.core.frame.DataFrame
return the updated data based on the given dataset

The difference between `get_update_data` and `update - update_date` only include some data specific feature
 - `update` include some general routine steps(e.g. prepare dataset, checking)

1.21.6 Utils

Serializable

Serializable will change the behaviors of pickle.

The rule to tell if a attribute will be kept or dropped when dumping. The rule with higher priorities is on the top - in the config attribute list -> always dropped - in the include attribute list -> always kept - in the exclude attribute list -> always dropped - name not starts with `_` -> kept - name starts with `_` -> kept if `dump_all` is true else dropped

It provides a syntactic sugar for distinguish the attributes which user doesn't want. - For examples, a learnable Data-handler just wants to save the parameters without data when dumping to disk

```
qlib.utils.serial.Serializable.dump_all
will the object dump all object
```

1.22 Qlib FAQ

1.22.1 Qlib Frequently Asked Questions

- 1. *RuntimeError: An attempt has been made to start a new process before the current process has finished its bootstrapping phase...*
- 2. *qlib.data.cache.QlibCacheException: It sees the key(...) of the redis lock has existed in your redis db now.*
- 3. *ModuleNotFoundError: No module named 'qlib.data._libs.rolling'*
- 4. *BadNamespaceError: / is not a connected namespace*
- 5. *TypeError: send() got an unexpected keyword argument 'binary'*

1. RuntimeError: An attempt has been made to start a new process before the current process has finished its bootstrapping phase...

```
RuntimeError:
    An attempt has been made to start a new process before the
    current process has finished its bootstrapping phase.

    This probably means that you are not using fork to start your
    child processes and you have forgotten to use the proper idiom
    in the main module:

        if __name__ == '__main__':
            freeze_support()
            ...
```

(continues on next page)

(continued from previous page)

```
The "freeze_support()" line can be omitted if the program
is not going to be frozen to produce an executable.
```

This is caused by the limitation of multiprocessing under windows OS. Please refer to [here](#) for more info.

Solution: To select a start method you use the `D.features` in the `if __name__ == '__main__':` clause of the main module. For example:

```
import qlib
from qlib.data import D

if __name__ == "__main__":
    qlib.init()
    instruments = ["SH600000"]
    fields = ["$close", "$change"]
    df = D.features(instruments, fields, start_time='2010-01-01', end_time='2012-12-31')
    print(df.head())
```

2. `qlib.data.cache.QlibCacheException`: It sees the key(...) of the redis lock has existed in your redis db now.

It sees the key of the redis lock has existed in your redis db now. You can use the following command to clear your redis keys and rerun your commands

```
$ redis-cli
> select 1
> flushdb
```

If the issue is not resolved, use `keys *` to find if multiple keys exist. If so, try using `flushall` to clear all the keys.

Note: `qlib.config.redis_task_db` defaults is 1, users can use `qlib.init(redis_task_db=<other_db>)` settings.

Also, feel free to post a new issue in our GitHub repository. We always check each issue carefully and try our best to solve them.

3. `ModuleNotFoundError`: No module named '`qlib.data._libs.rolling`'

```
#### Do not import qlib package in the repository directory in case of importing qlib_
↳from . without compiling #####
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "qlib/qlib/__init__.py", line 19, in init
    from .data.cache import H
File "qlib/qlib/data/__init__.py", line 8, in <module>
    from .data import (
File "qlib/qlib/data/data.py", line 20, in <module>
    from .cache import H
File "qlib/qlib/data/cache.py", line 36, in <module>
    from .ops import Operators
```

(continues on next page)

(continued from previous page)

```
File "qlib/qlib/data/ops.py", line 19, in <module>
    from ._libs.rolling import rolling_slope, rolling_rsquare, rolling_resi
ModuleNotFoundError: No module named 'qlib.data._libs.rolling'
```

- If the error occurs when importing qlib package with PyCharm IDE, users can execute the following command in the project root folder to compile Cython files and generate executable files:

```
python setup.py build_ext --inplace
```

- If the error occurs when importing qlib package with command `python`, users need to change the running directory to ensure that the script does not run in the project directory.

4. BadNamespaceError: / is not a connected namespace

```
File "qlib_online.py", line 35, in <module>
    cal = D.calendar()
File "e:\code\python\microsoft\qlib_latest\qlib\qlib\data\data.py", line 973, in _
    ↪calendar
    return Cal.calendar(start_time, end_time, freq, future=future)
File "e:\code\python\microsoft\qlib_latest\qlib\qlib\data\data.py", line 798, in _
    ↪calendar
    self.conn.send_request(
File "e:\code\python\microsoft\qlib_latest\qlib\qlib\data\client.py", line 101, in _
    ↪send_request
    self.sio.emit(request_type + "_request", request_content)
File "G:\apps\miniconda\envs\qlib\lib\site-packages\python_socketio-5.3.0-py3.8.
    ↪egg\socketio\client.py", line 369, in emit
    raise exceptions.BadNamespaceError(
BadNamespaceError: / is not a connected namespace.
```

- The version of python-socketio in qlib needs to be the same as the version of python-socketio in qlib-server:

```
pip install -U python-socketio==<qlib-server python-socketio version>
```

5. TypeError: send() got an unexpected keyword argument 'binary'

```
File "qlib_online.py", line 35, in <module>
    cal = D.calendar()
File "e:\code\python\microsoft\qlib_latest\qlib\qlib\data\data.py", line 973, in _
    ↪calendar
    return Cal.calendar(start_time, end_time, freq, future=future)
File "e:\code\python\microsoft\qlib_latest\qlib\qlib\data\data.py", line 798, in _
    ↪calendar
    self.conn.send_request(
File "e:\code\python\microsoft\qlib_latest\qlib\qlib\data\client.py", line 101, in _
    ↪send_request
    self.sio.emit(request_type + "_request", request_content)
File "G:\apps\miniconda\envs\qlib\lib\site-packages\socketio\client.py", line 263, in _
    ↪emit
    self._send_packet(packet.Packet(packet.EVENT, namespace=namespace,
File "G:\apps\miniconda\envs\qlib\lib\site-packages\socketio\client.py", line 339, in _
    ↪_send_packet
```

(continues on next page)

(continued from previous page)

```
self.eio.send(ep, binary=binary)
TypeError: send() got an unexpected keyword argument 'binary'
```

- The `python-engineio` version needs to be compatible with the `python-socketio` version, reference: <https://github.com/miguelgrinberg/python-socketio#version-compatibility>

```
pip install -U python-engineio==<compatible python-socketio version>
# or
pip install -U python-socketio==3.1.2 python-engineio==3.13.2
```

1.23 Changelog

Here you can see the full list of changes between each QLib release.

1.23.1 Version 0.1.0

This is the initial release of QLib library.

1.23.2 Version 0.1.1

Performance optimize. Add more features and operators.

1.23.3 Version 0.1.2

- Support operator syntax. Now `High() - Low()` is equivalent to `Sub(High(), Low())`.
- Add more technical indicators.

1.23.4 Version 0.1.3

Bug fix and add instruments filtering mechanism.

1.23.5 Version 0.2.0

- Redesign `LocalProvider` database format for performance improvement.
- Support load features as string fields.
- Add scripts for database construction.
- More operators and technical indicators.

1.23.6 Version 0.2.1

- Support registering user-defined `Provider`.
- Support use operators in string format, e.g. `['Ref($close, 1)']` is valid field format.

- Support dynamic fields in `$some_field` format. And existing fields like `Close()` may be deprecated in the future.

1.23.7 Version 0.2.2

- Add `disk_cache` for reusing features (enabled by default).
- Add `qlib.contrib` for experimental model construction and evaluation.

1.23.8 Version 0.2.3

- Add `backtest` module
- Decoupling the `Strategy`, `Account`, `Position`, `Exchange` from the `backtest` module

1.23.9 Version 0.2.4

- Add `profit_attribution` module
- Add `risk_control` and `cost_control` strategies

1.23.10 Version 0.3.0

- Add `estimator` module

1.23.11 Version 0.3.1

- Add `filter` module

1.23.12 Version 0.3.2

- Add real price trading, if the `factor` field in the data set is incomplete, use `adj_price` trading
- Refactor `handler` `launcher` `trainer` code
- Support `backtest` configuration parameters in the configuration file
- Fix bug in position amount is 0
- Fix bug of `filter` module

1.23.13 Version 0.3.3

- Fix bug of `filter` module

1.23.14 Version 0.3.4

- Support for `finetune` model
- Refactor `fetcher` code

1.23.15 Version 0.3.5

- Support multi-label training, you can provide multiple label in handler. (But LightGBM doesn't support due to the algorithm itself)
- Refactor handler code, dataset.py is no longer used, and you can deploy your own labels and features in feature_label_config
- Handler only offer DataFrame. Also, trainer and model.py only receive DataFrame
- Change split_rolling_data, we roll the data on market calendar now, not on normal date
- Move some date config from handler to trainer

1.23.16 Version 0.4.0

- Add data package that holds all data-related codes
- Reform the data provider structure
- Create a server for data centralized management '[qlib-server<https://amc-msra.visualstudio.com/trading-algo/_git/qlib-server>](https://amc-msra.visualstudio.com/trading-algo/_git/qlib-server)'
- Add a ClientProvider to work with server
- Add a pluggable cache mechanism
- Add a recursive backtracking algorithm to inspect the furthest reference date for an expression

Note: The `D.instruments` function does not support `start_time`, `end_time`, and `as_list` parameters, if you want to get the results of previous versions of `D.instruments`, you can do this:

```
>>> from qlib.data import D
>>> instruments = D.instruments(market='csi500')
>>> D.list_instruments(instruments=instruments, start_time='2015-01-01', end_time=
↪ '2016-02-15', as_list=True)
```

1.23.17 Version 0.4.1

- Add support Windows
- Fix instruments type bug
- Fix features is empty bug(It will cause failure in updating)
- Fix cache lock and update bug
- Fix use the same cache for the same field (the original space will add a new cache)
- Change “logger handler” from config
- Change model load support 0.4.0 later
- The default value of the `method` parameter of `risk_analysis` function is changed from `ci` to `si`

1.23.18 Version 0.4.2

- Refactor DataHandler
- Add Alpha360 DataHandler

1.23.19 Version 0.4.3

- Implementing Online Inference and Trading Framework
- Refactoring The interfaces of backtest and strategy module.

1.23.20 Version 0.4.4

- Optimize cache generation performance
- Add report module
- Fix bug when using ServerDatasetCache offline.
- In the previous version of `long_short_backtest`, there is a case of `np.nan` in `long_short`. The current version 0.4.4 has been fixed, so `long_short_backtest` will be different from the previous version.
- In the 0.4.2 version of `risk_analysis` function, `N` is 250, and `N` is 252 from 0.4.3, so 0.4.2 is 0.002122 smaller than the 0.4.3 the backtest result is slightly different between 0.4.2 and 0.4.3.
- **refactor the argument of backtest function.**
 - **NOTE:** - The default arguments of topk margin strategy is changed. Please pass the arguments explicitly if you want to get the same backtest result as previous version. - The `TopkWeightStrategy` is changed slightly. It will try to sell the stocks more than `topk`. (The backtest result of `TopkAmountStrategy` remains the same)
- The margin ratio mechanism is supported in the Topk Margin strategies.

1.23.21 Version 0.4.5

- **Add multi-kernel implementation for both client and server.**
 - Support a new way to load data from client which skips dataset cache.
 - Change the default dataset method from single kernel implementation to multi kernel implementation.
- Accelerate the high frequency data reading by optimizing the relative modules.
- Support a new method to write config file by using dict.

1.23.22 Version 0.4.6

- **Some bugs are fixed**
 - The default config in *Version 0.4.5* is not friendly to daily frequency data.
 - Backtest error in `TopkWeightStrategy` when `WithInteract=True`.

1.23.23 Version 0.5.0

- **First opensource version**
 - Refine the docs, code
 - Add baselines
 - public data crawler

1.23.24 Version 0.8.0

- **The backtest is greatly refactored.**
 - Nested decision execution framework is supported
 - **There are lots of changes for daily trading, it is hard to list all of them. But a few important changes could be listed.**
 - * **The trading limitation is more accurate;**
 - In [previous version](#), longing and shorting actions share the same action.
 - In [current version](#), the trading limitation is different between logging and shorting action.
 - * **The constant is different when calculating annualized metrics.**
 - [Current version](#) uses more accurate constant than [previous version](#)
 - * [A new version](#) of data is released. Due to the unstability of Yahoo data source, the data may be different after downloading data again.
 - * Users could check out the backtesting results between [Current version](#) and [previous version](#)

1.23.25 Other Versions

Please refer to [Github release Notes](#)

q

- `qlib.contrib.evaluate`, 142
- `qlib.contrib.report.analysis_model.analysis_model_performance`, 152
- `qlib.contrib.report.analysis_position.cumulative_return`, 147
- `qlib.contrib.report.analysis_position.rank_label`, 151
- `qlib.contrib.report.analysis_position.report`, 145
- `qlib.contrib.report.analysis_position.risk_analysis`, 148
- `qlib.contrib.report.analysis_position.score_ic`, 146
- `qlib.data.base`, 108
- `qlib.data.data`, 97
- `qlib.data.dataset.__init__`, 130
- `qlib.data.dataset.handler`, 133
- `qlib.data.dataset.loader`, 130
- `qlib.data.dataset.processor`, 138
- `qlib.data.filter`, 106
- `qlib.data.ops`, 109
- `qlib.model.base`, 141
- `qlib.model.ens.ensemble`, 178
- `qlib.model.ens.group`, 176
- `qlib.model.trainer`, 169
- `qlib.utils.serial.Serializable`, 191
- `qlib.workflow.online.manager`, 181
- `qlib.workflow.online.strategy`, 184
- `qlib.workflow.online.update`, 188
- `qlib.workflow.online.utils`, 186
- `qlib.workflow.record_temp`, 160
- `qlib.workflow.task.collect`, 174
- `qlib.workflow.task.gen`, 162
- `qlib.workflow.task.manage`, 165
- `qlib.workflow.task.utils`, 178

Symbols

`__init__()` (*qlib.data.base.Feature* method), 109
`__init__()` (*qlib.data.cache.DiskDatasetCache* method), 125
`__init__()` (*qlib.data.cache.DiskDatasetCache.IndexManager* method), 125
`__init__()` (*qlib.data.cache.DiskExpressionCache* method), 124
`__init__()` (*qlib.data.cache.MemCache* method), 31, 123
`__init__()` (*qlib.data.cache.MemCacheUnit* method), 31, 123
`__init__()` (*qlib.data.data.ClientCalendarProvider* method), 103
`__init__()` (*qlib.data.data.ClientDatasetProvider* method), 104
`__init__()` (*qlib.data.data.ClientInstrumentProvider* method), 104
`__init__()` (*qlib.data.data.ClientProvider* method), 105
`__init__()` (*qlib.data.data.ExpressionProvider* method), 100
`__init__()` (*qlib.data.data.LocalCalendarProvider* method), 101
`__init__()` (*qlib.data.data.LocalDatasetProvider* method), 103
`__init__()` (*qlib.data.data.LocalExpressionProvider* method), 102
`__init__()` (*qlib.data.data.LocalFeatureProvider* method), 101
`__init__()` (*qlib.data.data.LocalInstrumentProvider* method), 101
`__init__()` (*qlib.data.dataset.__init__.DatasetH* method), 30
`__init__()` (*qlib.data.dataset.handler.DataHandler* method), 134
`__init__()` (*qlib.data.dataset.handler.DataHandlerLP* method), 26, 136
`__init__()` (*qlib.data.dataset.loader.DLWParser* method), 130
`__init__()` (*qlib.data.dataset.loader.DataLoaderDH* method), 133
`__init__()` (*qlib.data.dataset.loader.QlibDataLoader* method), 131
`__init__()` (*qlib.data.dataset.loader.StaticDataLoader* method), 132
`__init__()` (*qlib.data.dataset.processor.CSRankNorm* method), 140
`__init__()` (*qlib.data.dataset.processor.CSZFillna* method), 141
`__init__()` (*qlib.data.dataset.processor.CSZScoreNorm* method), 140
`__init__()` (*qlib.data.dataset.processor.DropCol* method), 139
`__init__()` (*qlib.data.dataset.processor.DropnaLabel* method), 139
`__init__()` (*qlib.data.dataset.processor.DropnaProcessor* method), 139
`__init__()` (*qlib.data.dataset.processor.Fillna* method), 139
`__init__()` (*qlib.data.dataset.processor.FilterCol* method), 139
`__init__()` (*qlib.data.dataset.processor.MinMaxNorm* method), 139
`__init__()` (*qlib.data.dataset.processor.RobustZScoreNorm* method), 140
`__init__()` (*qlib.data.dataset.processor.ZScoreNorm* method), 140
`__init__()` (*qlib.data.filter.BaseDFilter* method), 106
`__init__()` (*qlib.data.filter.ExpressionDFilter* method), 107
`__init__()` (*qlib.data.filter.NameDFilter* method), 107
`__init__()` (*qlib.data.filter.SeriesDFilter* method), 106
`__init__()` (*qlib.data.ops.Abs* method), 110
`__init__()` (*qlib.data.ops.Add* method), 112
`__init__()` (*qlib.data.ops.And* method), 115
`__init__()` (*qlib.data.ops.Corr* method), 122

`__init__()` (*qlib.data.ops.Count method*), 120
`__init__()` (*qlib.data.ops.Cov method*), 122
`__init__()` (*qlib.data.ops.Delta method*), 120
`__init__()` (*qlib.data.ops.Div method*), 113
`__init__()` (*qlib.data.ops.EMA method*), 121
`__init__()` (*qlib.data.ops.ElemOperator method*), 109
`__init__()` (*qlib.data.ops.Eq method*), 114
`__init__()` (*qlib.data.ops.Ge method*), 114
`__init__()` (*qlib.data.ops.Greater method*), 113
`__init__()` (*qlib.data.ops.Gt method*), 113
`__init__()` (*qlib.data.ops.IdxMax method*), 118
`__init__()` (*qlib.data.ops.IdxMin method*), 118
`__init__()` (*qlib.data.ops.If method*), 115
`__init__()` (*qlib.data.ops.Kurt method*), 118
`__init__()` (*qlib.data.ops.Le method*), 114
`__init__()` (*qlib.data.ops.Less method*), 113
`__init__()` (*qlib.data.ops.Log method*), 110
`__init__()` (*qlib.data.ops.Lt method*), 114
`__init__()` (*qlib.data.ops.Mad method*), 119
`__init__()` (*qlib.data.ops.Mask method*), 111
`__init__()` (*qlib.data.ops.Max method*), 118
`__init__()` (*qlib.data.ops.Mean method*), 117
`__init__()` (*qlib.data.ops.Med method*), 119
`__init__()` (*qlib.data.ops.Min method*), 118
`__init__()` (*qlib.data.ops.Mul method*), 112
`__init__()` (*qlib.data.ops.Ne method*), 114
`__init__()` (*qlib.data.ops.Not method*), 111
`__init__()` (*qlib.data.ops.NpElemOperator method*), 110
`__init__()` (*qlib.data.ops.NpPairOperator method*), 112
`__init__()` (*qlib.data.ops.OpsWrapper method*), 122
`__init__()` (*qlib.data.ops.Or method*), 115
`__init__()` (*qlib.data.ops.PairOperator method*), 111
`__init__()` (*qlib.data.ops.PairRolling method*), 121
`__init__()` (*qlib.data.ops.Power method*), 112
`__init__()` (*qlib.data.ops.Quantile method*), 119
`__init__()` (*qlib.data.ops.Rank method*), 119
`__init__()` (*qlib.data.ops.Ref method*), 116
`__init__()` (*qlib.data.ops.Resi method*), 120
`__init__()` (*qlib.data.ops.Rolling method*), 116
`__init__()` (*qlib.data.ops.Rsquare method*), 120
`__init__()` (*qlib.data.ops.Sign method*), 110
`__init__()` (*qlib.data.ops.Skew method*), 117
`__init__()` (*qlib.data.ops.Slope method*), 120
`__init__()` (*qlib.data.ops.Std method*), 117
`__init__()` (*qlib.data.ops.Sub method*), 112
`__init__()` (*qlib.data.ops.Sum method*), 117
`__init__()` (*qlib.data.ops.TResample method*), 122
`__init__()` (*qlib.data.ops.Var method*), 117
`__init__()` (*qlib.data.ops.WMA method*), 121
`__init__()` (*qlib.data.storage.file_storage.FileCalendarStorage method*), 128
`__init__()` (*qlib.data.storage.file_storage.FileFeatureStorage method*), 129
`__init__()` (*qlib.data.storage.file_storage.FileInstrumentStorage method*), 128
`__init__()` (*qlib.data.storage.storage.CalendarStorage method*), 126
`__init__()` (*qlib.data.storage.storage.FeatureStorage method*), 127
`__init__()` (*qlib.data.storage.storage.InstrumentStorage method*), 126
`__init__()` (*qlib.model.ens.group.Group method*), 176
`__init__()` (*qlib.model.ens.group.RollingGroup method*), 177
`__init__()` (*qlib.model.meta.dataset.MetaTaskDataset method*), 44
`__init__()` (*qlib.model.meta.task.MetaTask method*), 44
`__init__()` (*qlib.model.trainer.DelayTrainerR method*), 171
`__init__()` (*qlib.model.trainer.DelayTrainerRM method*), 173
`__init__()` (*qlib.model.trainer.Trainer method*), 93, 169
`__init__()` (*qlib.model.trainer.TrainerR method*), 170
`__init__()` (*qlib.model.trainer.TrainerRM method*), 172
`__init__()` (*qlib.workflow.__init__.QlibRecorder method*), 47
`__init__()` (*qlib.workflow.exp.Experiment method*), 57, 156
`__init__()` (*qlib.workflow.expm.ExpManager method*), 55, 154
`__init__()` (*qlib.workflow.online.manager.OnlineManager method*), 77, 182
`__init__()` (*qlib.workflow.online.strategy.OnlineStrategy method*), 79, 184
`__init__()` (*qlib.workflow.online.strategy.RollingStrategy method*), 80, 185
`__init__()` (*qlib.workflow.online.update.DSBasedUpdater method*), 84, 189
`__init__()` (*qlib.workflow.online.update.LabelUpdater method*), 85, 190
`__init__()` (*qlib.workflow.online.update.RMDLoader method*), 83, 188
`__init__()` (*qlib.workflow.online.update.RecordUpdater method*), 83, 188
`__init__()` (*qlib.workflow.online.utils.OnlineTool method*), 81, 186
`__init__()` (*qlib.workflow.online.utils.OnlineToolR method*), 82, 187
`__init__()` (*qlib.workflow.record_temp.ACRecordTemp method*), 161
`__init__()` (*qlib.workflow.record_temp.HFSignalRecord*

method), 161
 __init__() (*qlib.workflow.record_temp.PortAnaRecord*
method), 162
 __init__() (*qlib.workflow.record_temp.RecordTemp*
method), 160
 __init__() (*qlib.workflow.record_temp.SigAnaRecord*
method), 161
 __init__() (*qlib.workflow.record_temp.SignalRecord*
method), 160
 __init__() (*qlib.workflow.recorder.Recorder*
method), 59, 158
 __init__() (*qlib.workflow.task.collect.Collector*
method), 174
 __init__() (*qlib.workflow.task.collect.MergeCollector*
method), 175
 __init__() (*qlib.workflow.task.collect.RecorderCollector*
method), 175
 __init__() (*qlib.workflow.task.gen.MultiHorizonGenBase*
method), 165
 __init__() (*qlib.workflow.task.gen.RollingGen*
method), 163
 __init__() (*qlib.workflow.task.manage.TaskManager*
method), 90, 166
 __init__() (*qlib.workflow.task.utils.TimeAdjuster*
method), 179

A

Abs (*class in qlib.data.ops*), 110
 ACRecordTemp (*class in qlib.workflow.record_temp*),
 161
 Add (*class in qlib.data.ops*), 112
 add_strategy() (*qlib.workflow.online.manager.OnlineManager*
method), 78, 183
 align_idx() (*qlib.workflow.task.utils.TimeAdjuster*
method), 179
 align_seg() (*qlib.workflow.task.utils.TimeAdjuster*
method), 180
 align_time() (*qlib.workflow.task.utils.TimeAdjuster*
method), 180
 And (*class in qlib.data.ops*), 114
 AverageEnsemble (*class in qlib.model.ens.ensemble*), 178

B

backtest_daily() (*in module qlib.contrib.evaluate*), 143
 BaseDFilter (*class in qlib.data.filter*), 106
 BaseModel (*class in qlib.model.base*), 141
 BaseProvider (*class in qlib.data.data*), 104
 BaseProviderWrapper (*in module qlib.data.data*),
 106
 BaseStorage (*class in qlib.data.storage.storage*), 126
 begin_task_train() (*in module qlib.model.trainer*), 169

C

cache_to_origin_data() (*qlib.data.cache.DatasetCache* *static method*),
 33, 124
 cache_walker() (*qlib.data.data.LocalDatasetProvider*
static method), 103
 cal_interval() (*qlib.workflow.task.utils.TimeAdjuster*
method), 179
 calendar() (*qlib.data.data.CalendarProvider*
method), 97
 calendar() (*qlib.data.data.ClientCalendarProvider*
method), 103
 CalendarProvider (*class in qlib.data.data*), 97
 CalendarProviderWrapper (*in module qlib.data.data*), 105
 CalendarStorage (*class in qlib.data.storage.storage*), 126
 Cast() (*qlib.data.dataset.handler.DataHandlerLP*
class method), 28, 138
 check() (*qlib.data.storage.file_storage.FileStorageMixin*
method), 128
 check() (*qlib.workflow.record_temp.RecordTemp*
method), 160
 ClientCalendarProvider (*class in qlib.data.data*), 103
 ClientDatasetProvider (*class in qlib.data.data*),
 104
 ClientInstrumentProvider (*class in qlib.data.data*), 104
 ClientProvider (*class in qlib.data.data*), 105
 collect() (*qlib.workflow.task.collect.Collector*
method), 174
 collect() (*qlib.workflow.task.collect.MergeCollector*
method), 175
 collect() (*qlib.workflow.task.collect.RecorderCollector*
method), 176
 Collector (*class in qlib.workflow.task.collect*), 174
 commit_task_res() (*qlib.workflow.task.manage.TaskManager*
method), 92, 167
 config() (*qlib.data.dataset.__init__.DatasetH*
method), 30
 config() (*qlib.data.dataset.handler.DataHandler*
method), 134
 config() (*qlib.data.dataset.handler.DataHandlerLP*
method), 27, 137
 config() (*qlib.data.dataset.processor.Processor*
method), 138
 Corr (*class in qlib.data.ops*), 121
 Count (*class in qlib.data.ops*), 119
 Cov (*class in qlib.data.ops*), 122
 create_exp() (*qlib.workflow.expm.ExpManager*
method), 55, 154
 create_recorder() (*qlib.workflow.exp.Experiment*

method), 57, 156
 create_task() (*qlib.workflow.task.manage.TaskManager* method), 91, 166
 CSRankNorm (class in *qlib.data.dataset.processor*), 140
 CSZFillna (class in *qlib.data.dataset.processor*), 141
 CSZScoreNorm (class in *qlib.data.dataset.processor*), 140
 cumulative_return_graph() (in module *qlib.contrib.report.analysis_position.cumulative_return*), 147
D
 data (*qlib.data.storage.file_storage.FileCalendarStorage* attribute), 128
 data (*qlib.data.storage.file_storage.FileFeatureStorage* attribute), 129
 data (*qlib.data.storage.file_storage.FileInstrumentStorage* attribute), 128
 data (*qlib.data.storage.storage.CalendarStorage* attribute), 126
 data (*qlib.data.storage.storage.FeatureStorage* attribute), 127
 data (*qlib.data.storage.storage.InstrumentStorage* attribute), 127
 DataHandler (class in *qlib.data.dataset.handler*), 133
 DataHandlerLP (class in *qlib.data.dataset.handler*), 25, 136
 DataLoader (class in *qlib.data.dataset.loader*), 24, 130
 DataLoaderDH (class in *qlib.data.dataset.loader*), 132
 dataset() (*qlib.data.cache.DatasetCache* method), 33, 124
 dataset() (*qlib.data.data.ClientDatasetProvider* method), 104
 dataset() (*qlib.data.data.DatasetProvider* method), 100
 dataset() (*qlib.data.data.LocalDatasetProvider* method), 103
 dataset_processor() (*qlib.data.data.DatasetProvider* static method), 100
 DatasetCache (class in *qlib.data.cache*), 32, 123
 DatasetH (class in *qlib.data.dataset.__init__*), 29
 DatasetProvider (class in *qlib.data.data*), 100
 DatasetProviderWrapper (in module *qlib.data.data*), 106
 default_uri (*qlib.workflow.expm.ExpManager* attribute), 56, 156
 delay_prepare() (*qlib.workflow.online.manager.OnlineManager* method), 79, 184
 DelayTrainerR (class in *qlib.model.trainer*), 171
 DelayTrainerRM (class in *qlib.model.trainer*), 173
 delete_exp() (*qlib.workflow.__init__.QlibRecorder* method), 51

delete_exp() (*qlib.workflow.expm.ExpManager* method), 56, 156
 delete_recorder() (*qlib.workflow.__init__.QlibRecorder* method), 52
 delete_recorder() (*qlib.workflow.exp.Experiment* method), 58, 157
 delete_tags() (*qlib.workflow.recorder.Recorder* method), 60, 159
 Delta (class in *qlib.data.ops*), 120
 depend_cls (*qlib.workflow.record_temp.HFSignalRecord* attribute), 161
 depend_cls (*qlib.workflow.record_temp.PortAnaRecord* attribute), 162
 depend_cls (*qlib.workflow.record_temp.SigAnaRecord* attribute), 161
 DiskDatasetCache (class in *qlib.data.cache*), 125
 DiskDatasetCache.IndexManager (class in *qlib.data.cache*), 125
 DiskExpressionCache (class in *qlib.data.cache*), 124
 Div (class in *qlib.data.ops*), 112
 DLWParser (class in *qlib.data.dataset.loader*), 130
 DropCol (class in *qlib.data.dataset.processor*), 139
 DropnaLabel (class in *qlib.data.dataset.processor*), 139
 DropnaProcessor (class in *qlib.data.dataset.processor*), 139
 DSBasedUpdater (class in *qlib.workflow.online.update*), 83, 188
 dump_all (in module *qlib.utils.serial.Serializable*), 191
E
 ElemOperator (class in *qlib.data.ops*), 109
 EMA (class in *qlib.data.ops*), 121
 end() (*qlib.workflow.exp.Experiment* method), 57, 156
 end_exp() (*qlib.workflow.__init__.QlibRecorder* method), 48
 end_exp() (*qlib.workflow.expm.ExpManager* method), 55, 154
 end_index (*qlib.data.storage.file_storage.FileFeatureStorage* attribute), 129
 end_index (*qlib.data.storage.storage.FeatureStorage* attribute), 127
 end_run() (*qlib.workflow.recorder.Recorder* method), 60, 159
 end_task_train() (in module *qlib.model.trainer*), 171
 end_train() (*qlib.model.trainer.DelayTrainerR* method), 171
 end_train() (*qlib.model.trainer.DelayTrainerRM* method), 173
 end_train() (*qlib.model.trainer.Trainer* method), 93, 170

`end_train()` (*qlib.model.trainer.TrainerR method*), 171
`end_train()` (*qlib.model.trainer.TrainerRM method*), 172
`Ensemble` (*class in qlib.model.ens.ensemble*), 178
`Eq` (*class in qlib.data.ops*), 114
`Experiment` (*class in qlib.workflow.exp*), 57, 156
`ExpManager` (*class in qlib.workflow.expm*), 54, 154
`Expression` (*class in qlib.data.base*), 108
`expression()` (*qlib.data.cache.ExpressionCache method*), 32, 123
`expression()` (*qlib.data.data.ExpressionProvider method*), 100
`expression()` (*qlib.data.data.LocalExpressionProvider method*), 102
`ExpressionCache` (*class in qlib.data.cache*), 32, 123
`ExpressionDFilter` (*class in qlib.data.filter*), 107
`ExpressionOps` (*class in qlib.data.base*), 109
`ExpressionProvider` (*class in qlib.data.data*), 99
`ExpressionProviderWrapper` (*in module qlib.data.data*), 106

F

`Feature` (*class in qlib.data.base*), 109
`feature()` (*qlib.data.data.FeatureProvider method*), 99
`feature()` (*qlib.data.data.LocalFeatureProvider method*), 101
`FeatureProvider` (*class in qlib.data.data*), 99
`FeatureProviderWrapper` (*in module qlib.data.data*), 105
`features()` (*qlib.data.data.BaseProvider method*), 105
`features_uri()` (*qlib.data.data.LocalProvider method*), 105
`FeatureStorage` (*class in qlib.data.storage.storage*), 127
`fetch()` (*qlib.data.dataset.handler.DataHandler method*), 134
`fetch()` (*qlib.data.dataset.handler.DataHandlerLP method*), 27, 137
`fetch_task()` (*qlib.workflow.task.manage.TaskManager method*), 91, 167
`FileCalendarStorage` (*class in qlib.data.storage.file_storage*), 128
`FileFeatureStorage` (*class in qlib.data.storage.file_storage*), 129
`FileInstrumentStorage` (*class in qlib.data.storage.file_storage*), 128
`FileStorageMixin` (*class in qlib.data.storage.file_storage*), 128
`Fillna` (*class in qlib.data.dataset.processor*), 139
`filter_main()` (*qlib.data.filter.SeriesDFilter method*), 107
`FilterCol` (*class in qlib.data.dataset.processor*), 139
`finetune()` (*qlib.model.base.ModelFT method*), 142
`first_tasks()` (*qlib.workflow.online.strategy.OnlineStrategy method*), 80, 185
`first_tasks()` (*qlib.workflow.online.strategy.RollingStrategy method*), 81, 186
`first_train()` (*qlib.workflow.online.manager.OnlineManager method*), 77, 182
`fit()` (*qlib.data.dataset.handler.DataHandlerLP method*), 26, 137
`fit()` (*qlib.data.dataset.processor.MinMaxNorm method*), 140
`fit()` (*qlib.data.dataset.processor.Processor method*), 138
`fit()` (*qlib.data.dataset.processor.RobustZScoreNorm method*), 140
`fit()` (*qlib.data.dataset.processor.ZScoreNorm method*), 140
`fit()` (*qlib.model.base.Model method*), 34, 141
`fit()` (*qlib.model.meta.model.MetaGuideModel method*), 46
`fit()` (*qlib.model.meta.model.MetaModel method*), 45
`fit()` (*qlib.model.meta.model.MetaTaskModel method*), 46
`fit_process_data()` (*qlib.data.dataset.handler.DataHandlerLP method*), 26, 137
`from_config()` (*qlib.data.filter.BaseDFilter static method*), 106
`from_config()` (*qlib.data.filter.ExpressionDFilter static method*), 107
`from_config()` (*qlib.data.filter.NameDFilter static method*), 107

G

`Ge` (*class in qlib.data.ops*), 113
`gen_dataset_cache()` (*qlib.data.cache.DiskDatasetCache method*), 125
`gen_expression_cache()` (*qlib.data.cache.DiskExpressionCache method*), 124
`gen_following_tasks()` (*qlib.workflow.task.gen.RollingGen method*), 164
`generate()` (*qlib.workflow.record_temp.ACRecordTemp method*), 161
`generate()` (*qlib.workflow.record_temp.HFSignalRecord method*), 161
`generate()` (*qlib.workflow.record_temp.RecordTemp method*), 160
`generate()` (*qlib.workflow.record_temp.SignalRecord method*), 160

`generate()` (*qlib.workflow.task.gen.MultiHorizonGenBase* method), 165
`generate()` (*qlib.workflow.task.gen.RollingGen* method), 164
`generate()` (*qlib.workflow.task.gen.TaskGen* method), 89, 163
`get()` (*qlib.workflow.task.utils.TimeAdjuster* method), 179
`get_collector()` (*qlib.workflow.online.manager.OnlineManager* method), 78, 183
`get_collector()` (*qlib.workflow.online.strategy.OnlineStrategy* method), 80, 185
`get_collector()` (*qlib.workflow.online.strategy.RollingStrategy* method), 80, 185
`get_cols()` (*qlib.data.dataset.handler.DataHandler* method), 135
`get_cols()` (*qlib.data.dataset.handler.DataHandlerLP* method), 27, 138
`get_column_names()` (*qlib.data.data.DatasetProvider* static method), 100
`get_dataset()` (*qlib.workflow.online.update.RMDLoader* method), 83, 188
`get_exp()` (*qlib.workflow.__init__.QlibRecorder* method), 50
`get_exp()` (*qlib.workflow.expm.ExpManager* method), 55, 155
`get_exp_name()` (*qlib.workflow.task.collect.RecorderCollector* method), 176
`get_extended_window_size()` (*qlib.data.base.Expression* method), 109
`get_extended_window_size()` (*qlib.data.base.Feature* method), 109
`get_extended_window_size()` (*qlib.data.ops.ElemOperator* method), 110
`get_extended_window_size()` (*qlib.data.ops.If* method), 115
`get_extended_window_size()` (*qlib.data.ops.PairOperator* method), 111
`get_extended_window_size()` (*qlib.data.ops.PairRolling* method), 121
`get_extended_window_size()` (*qlib.data.ops.Ref* method), 116
`get_extended_window_size()` (*qlib.data.ops.Rolling* method), 116
`get_group_columns()` (in module *qlib.data.dataset.processor*), 138
`get_instruments_d()` (*qlib.data.data.DatasetProvider* static method), 100
`get_longest_back_rolling()` (*qlib.data.base.Expression* method), 108
`get_longest_back_rolling()` (*qlib.data.base.Feature* method), 109
`get_longest_back_rolling()` (*qlib.data.ops.ElemOperator* method), 109
`get_longest_back_rolling()` (*qlib.data.ops.If* method), 115
`get_longest_back_rolling()` (*qlib.data.ops.PairOperator* method), 111
`get_longest_back_rolling()` (*qlib.data.ops.PairRolling* method), 121
`get_longest_back_rolling()` (*qlib.data.ops.Ref* method), 116
`get_longest_back_rolling()` (*qlib.data.ops.Rolling* method), 116
`get_meta_input()` (*qlib.model.meta.task.MetaTask* method), 44
`get_mongodb()` (in module *qlib.workflow.task.utils*), 178
`get_online_tag()` (*qlib.workflow.online.utils.OnlineTool* method), 81, 186
`get_online_tag()` (*qlib.workflow.online.utils.OnlineToolR* method), 82, 187
`get_range_iterator()` (*qlib.data.dataset.handler.DataHandler* method), 136
`get_range_selector()` (*qlib.data.dataset.handler.DataHandler* method), 135
`get_recorder()` (*qlib.workflow.__init__.QlibRecorder* method), 51
`get_recorder()` (*qlib.workflow.exp.Experiment* method), 58, 157
`get_signals()` (*qlib.workflow.online.manager.OnlineManager* method), 79, 184
`get_update_data()` (*qlib.workflow.online.update.DSBasedUpdater* method), 85, 190
`get_update_data()` (*qlib.workflow.online.update.LabelUpdater* method), 85, 190
`get_update_data()` (*qlib.workflow.online.update.PredUpdater* method), 85, 190
`get_uri()` (*qlib.workflow.__init__.QlibRecorder* method), 51
`Greater` (class in *qlib.data.ops*), 113
`Group` (class in *qlib.model.ens.group*), 176
`group()` (*qlib.model.ens.group.Group* method), 177
`group()` (*qlib.model.ens.group.RollingGroup* method), 177
`Gt` (class in *qlib.data.ops*), 113

H

`handler_mod()` (in module *qlib.workflow.task.gen*), 163

has_worker() (*qlib.model.trainer.DelayTrainerRM method*), 174

has_worker() (*qlib.model.trainer.Trainer method*), 94, 170

has_worker() (*qlib.model.trainer.TrainerRM method*), 173

HashStockFormat (class in *qlib.data.dataset.processor*), 141

HFSignalRecord (class in *qlib.workflow.record_temp*), 161

I

ic_figure() (in module *qlib.contrib.report.analysis_model.analysis_model_performance*), 71, 152

IdxMax (class in *qlib.data.ops*), 118

IdxMin (class in *qlib.data.ops*), 118

If (class in *qlib.data.ops*), 115

index() (*qlib.data.storage.file_storage.FileCalendarStorage method*), 128

index() (*qlib.data.storage.storage.CalendarStorage method*), 126

indicator_analysis() (in module *qlib.contrib.evaluate*), 142

inference() (*qlib.model.meta.model.MetaGuideModel method*), 46

inference() (*qlib.model.meta.model.MetaModel method*), 45

inference() (*qlib.model.meta.model.MetaTaskModel method*), 46

insert_task() (*qlib.workflow.task.manage.TaskManager method*), 90, 166

insert_task_def() (*qlib.workflow.task.manage.TaskManager method*), 91, 166

inst_calculator() (*qlib.data.data.DatasetProvider static method*), 101

InstrumentProvider (class in *qlib.data.data*), 98

InstrumentProviderWrapper (in module *qlib.data.data*), 105

instruments() (*qlib.data.data.InstrumentProvider static method*), 98

InstrumentStorage (class in *qlib.data.storage.storage*), 126

is_delay() (*qlib.model.trainer.Trainer method*), 94, 170

is_for_infer() (*qlib.data.dataset.processor.DropnaLabel method*), 139

is_for_infer() (*qlib.data.dataset.processor.Processor method*), 138

K

Kurt (class in *qlib.data.ops*), 117

L

LabelUpdater (class in *qlib.workflow.online.update*), 85, 190

Le (class in *qlib.data.ops*), 114

Less (class in *qlib.data.ops*), 113

limited (*qlib.data.cache.MemCacheUnit attribute*), 31, 123

list() (*qlib.workflow.record_temp.HFSignalRecord method*), 161

list() (*qlib.workflow.record_temp.PortAnaRecord method*), 162

list() (*qlib.workflow.record_temp.RecordTemp method*), 160

list() (*qlib.workflow.record_temp.SigAnaRecord method*), 161

list() (*qlib.workflow.record_temp.SignalRecord method*), 161

list() (*qlib.workflow.task.manage.TaskManager static method*), 90, 166

list_artifacts() (*qlib.workflow.recorder.Recorder method*), 60, 159

list_experiments() (*qlib.workflow.__init__.QlibRecorder method*), 49

list_experiments() (*qlib.workflow.expm.ExpManager method*), 57, 156

list_instruments() (*qlib.data.data.ClientInstrumentProvider method*), 104

list_instruments() (*qlib.data.data.InstrumentProvider method*), 98

list_instruments() (*qlib.data.data.LocalInstrumentProvider method*), 101

list_metrics() (*qlib.workflow.recorder.Recorder method*), 60, 159

list_params() (*qlib.workflow.recorder.Recorder method*), 60, 159

list_recorders() (in module *qlib.workflow.task.utils*), 179

list_recorders() (*qlib.workflow.__init__.QlibRecorder method*), 49

list_recorders() (*qlib.workflow.exp.Experiment method*), 59, 158

list_tags() (*qlib.workflow.recorder.Recorder method*), 60, 159

load() (*qlib.data.base.Expression method*), 108

load() (*qlib.data.dataset.loader.DataLoader method*), 24, 130

load() (*qlib.data.dataset.loader.DataLoaderDH method*), 133

load() (*qlib.data.dataset.loader.DLWParser method*),

131
load() (*qlib.data.dataset.loader.StaticDataLoader*
method), 132
load() (*qlib.workflow.record_temp.RecordTemp*
method), 160
load_calendar() (*qlib.data.data.CalendarProvider*
method), 98
load_calendar() (*qlib.data.data.LocalCalendarProvider*
method), 101
load_group_df() (*qlib.data.dataset.loader.DLWParser*
method), 130
load_group_df() (*qlib.data.dataset.loader.QlibDataLoader*
method), 131
load_object() (*qlib.workflow.__init__.QlibRecorder*
method), 53
load_object() (*qlib.workflow.recorder.Recorder*
method), 59, 158
LocalCalendarProvider (*class in qlib.data.data*),
101
LocalDatasetProvider (*class in qlib.data.data*),
103
LocalExpressionProvider (*class in*
qlib.data.data), 102
LocalFeatureProvider (*class in qlib.data.data*),
101
LocalInstrumentProvider (*class in*
qlib.data.data), 101
LocalPITProvider (*class in qlib.data.data*), 102
LocalProvider (*class in qlib.data.data*), 105
locate_index() (*qlib.data.data.CalendarProvider*
method), 97
Log (*class in qlib.data.ops*), 110
log_metrics() (*qlib.workflow.__init__.QlibRecorder*
method), 54
log_metrics() (*qlib.workflow.recorder.Recorder*
method), 60, 159
log_params() (*qlib.workflow.__init__.QlibRecorder*
method), 53
log_params() (*qlib.workflow.recorder.Recorder*
method), 60, 159
long_short_backtest() (*in module*
qlib.contrib.evaluate), 144
Lt (*class in qlib.data.ops*), 114

M

Mad (*class in qlib.data.ops*), 119
Mask (*class in qlib.data.ops*), 110
Max (*class in qlib.data.ops*), 118
max() (*qlib.workflow.task.utils.TimeAdjuster* *method*),
179
Mean (*class in qlib.data.ops*), 116
Med (*class in qlib.data.ops*), 119
MemCache (*class in qlib.data.cache*), 31, 123
MemCacheUnit (*class in qlib.data.cache*), 31, 123

MergeCollector (*class in qlib.workflow.task.collect*),
175
MetaGuideModel (*class in qlib.model.meta.model*), 46
MetaModel (*class in qlib.model.meta.model*), 45
MetaTask (*class in qlib.model.meta.task*), 44
MetaTaskDataset (*class in qlib.model.meta.dataset*),
44
MetaTaskModel (*class in qlib.model.meta.model*), 45
Min (*class in qlib.data.ops*), 118
MinMaxNorm (*class in qlib.data.dataset.processor*), 139
Model (*class in qlib.model.base*), 34, 141
model_performance_graph() (*in module*
qlib.contrib.report.analysis_model.analysis_model_performance)
72, 152
ModelFT (*class in qlib.model.base*), 142
Mul (*class in qlib.data.ops*), 112
multi_cache_walker()
(*qlib.data.data.LocalDatasetProvider* *static*
method), 103
MultiHorizonGenBase (*class in*
qlib.workflow.task.gen), 165

N

NamedFilter (*class in qlib.data.filter*), 107
Ne (*class in qlib.data.ops*), 114
normalize_uri_args()
(*qlib.data.cache.DatasetCache* *static method*),
33, 124
Not (*class in qlib.data.ops*), 111
NpElemOperator (*class in qlib.data.ops*), 110
NpPairOperator (*class in qlib.data.ops*), 111

O

online_models() (*qlib.workflow.online.utils.OnlineTool*
method), 82, 187
online_models() (*qlib.workflow.online.utils.OnlineToolR*
method), 82, 187
OnlineManager (*class in*
qlib.workflow.online.manager), 77, 182
OnlineStrategy (*class in*
qlib.workflow.online.strategy), 79, 184
OnlineTool (*class in qlib.workflow.online.utils*), 81,
186
OnlineToolR (*class in qlib.workflow.online.utils*), 82,
187
OpsWrapper (*class in qlib.data.ops*), 122
Or (*class in qlib.data.ops*), 115

P

PairOperator (*class in qlib.data.ops*), 111
PairRolling (*class in qlib.data.ops*), 121
period_feature() (*qlib.data.data.LocalPITProvider*
method), 102

period_feature() (*qlib.data.data.PITProvider* method), 99
 PFeature (*class in qlib.data.base*), 109
 PITProvider (*class in qlib.data.data*), 99
 PITProviderWrapper (*in module qlib.data.data*), 105
 PortAnaRecord (*class in qlib.workflow.record_temp*), 161
 Power (*class in qlib.data.ops*), 112
 predict() (*qlib.model.base.BaseModel* method), 141
 predict() (*qlib.model.base.Model* method), 35, 141
 PredUpdater (*class in qlib.workflow.online.update*), 85, 190
 prepare() (*qlib.data.dataset.__init__.DatasetH* method), 30
 prepare_data() (*qlib.workflow.online.update.DSBasedUpdater* method), 84, 189
 prepare_online_models() (*qlib.workflow.online.strategy.OnlineStrategy* method), 80, 185
 prepare_signals() (*qlib.workflow.online.manager.OnlineManager* method), 78, 183
 prepare_tasks() (*qlib.model.meta.dataset.MetaTaskDataset* method), 45
 prepare_tasks() (*qlib.workflow.online.strategy.OnlineStrategy* method), 79, 184
 prepare_tasks() (*qlib.workflow.online.strategy.RollingStrategy* method), 81, 186
 prioritize() (*qlib.workflow.task.manage.TaskManager* method), 92, 168
 process_collect() (*qlib.workflow.task.collect.Collector* static method), 174
 process_data() (*qlib.data.dataset.handler.DataHandlerLP* method), 26, 137
 ProcessInf (*class in qlib.data.dataset.processor*), 139
 Processor (*class in qlib.data.dataset.processor*), 138
 ProviderBackendMixin (*class in qlib.data.data*), 97

Q

qlib.contrib.evaluate (*module*), 142
 qlib.contrib.report.analysis_model.analysis_model (*module*), 71, 152
 qlib.contrib.report.analysis_position.cumulative (*module*), 147
 qlib.contrib.report.analysis_position.rank_label (*module*), 151
 qlib.contrib.report.analysis_position.report (*module*), 63, 145
 qlib.contrib.report.analysis_position.risk_analysis (*module*), 67, 148
 qlib.contrib.report.analysis_position.score_ic (*module*), 65, 146
 qlib.data.base (*module*), 108
 qlib.data.data (*module*), 97
 qlib.data.dataset.__init__ (*module*), 130
 qlib.data.dataset.handler (*module*), 133
 qlib.data.dataset.loader (*module*), 130
 qlib.data.dataset.processor (*module*), 138
 qlib.data.filter (*module*), 106
 qlib.data.ops (*module*), 109
 qlib.model.base (*module*), 141
 qlib.model.ens.ensemble (*module*), 178
 qlib.model.ens.group (*module*), 176
 qlib.model.trainer (*module*), 169
 qlib.utils.serial.Serializable (*module*), 181
 qlib.workflow.online.manager (*module*), 76, 181
 qlib.workflow.online.strategy (*module*), 79, 184
 qlib.workflow.online.update (*module*), 83, 188
 qlib.workflow.online.utils (*module*), 81, 186
 qlib.workflow.record_temp (*module*), 160
 qlib.workflow.task.collect (*module*), 174
 qlib.workflow.task.gen (*module*), 162
 qlib.workflow.task.manage (*module*), 165
 qlib.workflow.task.utils (*module*), 178
 QlibDataLoader (*class in qlib.data.dataset.loader*), 131
 QlibRecorder (*class in qlib.workflow.__init__*), 47
 Quantile (*class in qlib.data.ops*), 118
 query() (*qlib.workflow.task.manage.TaskManager* method), 91, 167

R

Rank (*class in qlib.data.ops*), 119
 rank_label_graph() (*in module qlib.contrib.report.analysis_position.rank_label*), 151
 re_query() (*qlib.workflow.task.manage.TaskManager* method), 92, 167
 read_data_from_cache() (*qlib.data.cache.DiskDatasetCache* class method), 125
 read_data_from_cache() (*qlib.data.cache.DiskDatasetCache* class method), 125
 read_data_from_cache() (*qlib.data.dataset.processor.DropColnaProcessor* method), 139
 read_data_from_cache() (*qlib.data.dataset.processor.DropnaProcessor* method), 139
 read_data_from_cache() (*qlib.data.dataset.processor.FilterColnaProcessor* method), 139
 read_data_from_cache() (*qlib.data.dataset.processor.Processor* method), 138

rebase() (*qlib.data.storage.storage.FeatureStorage* method), 128

Recorder (class in *qlib.workflow.recorder*), 59, 158

RecorderCollector (class in *qlib.workflow.task.collect*), 175

RecordTemp (class in *qlib.workflow.record_temp*), 160

RecordUpdater (class in *qlib.workflow.online.update*), 83, 188

reduce() (*qlib.model.ens.group.Group* method), 177

Ref (class in *qlib.data.ops*), 116

register() (*qlib.data.ops.OpsWrapper* method), 122

register_all_ops() (in module *qlib.data.ops*), 122

register_all_wrappers() (in module *qlib.data.data*), 106

remove() (*qlib.workflow.task.manage.TaskManager* method), 92, 168

replace_task() (*qlib.workflow.task.manage.TaskManager* method), 90, 166

report_graph() (in module *qlib.contrib.report.analysis_position.report*), 63, 145

reset_online_tag() (*qlib.workflow.online.utils.OnlineTool* method), 81, 186

reset_online_tag() (*qlib.workflow.online.utils.OnlineToolR* method), 82, 187

reset_waiting() (*qlib.workflow.task.manage.TaskManager* method), 92, 168

Resi (class in *qlib.data.ops*), 120

return_task() (*qlib.workflow.task.manage.TaskManager* method), 92, 167

rewrite() (*qlib.data.storage.storage.FeatureStorage* method), 128

risk_analysis() (in module *qlib.contrib.evaluate*), 142

risk_analysis_graph() (in module *qlib.contrib.report.analysis_position.risk_analysis*), 67, 148

RMDLoader (class in *qlib.workflow.online.update*), 83, 188

RobustZScoreNorm (class in *qlib.data.dataset.processor*), 140

Rolling (class in *qlib.data.ops*), 115

RollingEnsemble (class in *qlib.model.ens.ensemble*), 178

RollingGen (class in *qlib.workflow.task.gen*), 163

RollingGroup (class in *qlib.model.ens.group*), 177

RollingStrategy (class in *qlib.workflow.online.strategy*), 80, 185

routine() (*qlib.workflow.online.manager.OnlineManager* method), 78, 183

Rsquare (class in *qlib.data.ops*), 120

run_task() (in module *qlib.workflow.task.manage*), 93, 168

S

safe_fetch_task() (*qlib.workflow.task.manage.TaskManager* method), 91, 167

save() (*qlib.workflow.record_temp.RecordTemp* method), 160

save_objects() (*qlib.workflow.__init__.QlibRecorder* method), 52

save_objects() (*qlib.workflow.recorder.Recorder* method), 59, 158

score_ic_graph() (in module *qlib.contrib.report.analysis_position.score_ic*), 65, 146

search_records() (*qlib.workflow.__init__.QlibRecorder* method), 48

search_records() (*qlib.workflow.exp.Experiment* method), 57, 157

search_records() (*qlib.workflow.expm.ExpManager* method), 55, 154

SeriesDFilter (class in *qlib.data.filter*), 106

set_end_time() (*qlib.workflow.task.utils.TimeAdjuster* method), 179

set_horizon() (*qlib.workflow.task.gen.MultiHorizonGenBase* method), 165

set_online_tag() (*qlib.workflow.online.utils.OnlineTool* method), 81, 186

set_online_tag() (*qlib.workflow.online.utils.OnlineToolR* method), 82, 187

set_tags() (*qlib.workflow.__init__.QlibRecorder* method), 54

set_tags() (*qlib.workflow.recorder.Recorder* method), 60, 159

set_uri() (*qlib.workflow.__init__.QlibRecorder* method), 51

set_uri() (*qlib.workflow.expm.ExpManager* method), 57, 156

setup_data() (*qlib.data.dataset.__init__.DatasetH* method), 30

setup_data() (*qlib.data.dataset.handler.DataHandler* method), 134

setup_data() (*qlib.data.dataset.handler.DataHandlerLP* method), 27, 137

shift() (*qlib.workflow.task.utils.TimeAdjuster* method), 180

SigAnaRecord (class in *qlib.workflow.record_temp*), 161

Sign (class in *qlib.data.ops*), 110

SignalRecord (class in *qlib.workflow.record_temp*), 160

simulate() (*qlib.workflow.online.manager.OnlineManager* method), 79, 184

SingleKeyEnsemble (class in *qlib.model.ens.ensemble*), 178

Skew (class in *qlib.data.ops*), 117

Slope (class in *qlib.data.ops*), 120

start() (*qlib.workflow.__init__.QlibRecorder* method), 47

start() (*qlib.workflow.exp.Experiment* method), 57, 156

start_exp() (*qlib.workflow.__init__.QlibRecorder* method), 48

start_exp() (*qlib.workflow.expm.ExpManager* method), 55, 154

start_index(*qlib.data.storage.file_storage.FileFeatureStorage* attribute), 129

start_index(*qlib.data.storage.storage.FeatureStorage* attribute), 127

start_run() (*qlib.workflow.recorder.Recorder* method), 59, 158

StaticDataLoader (class in *qlib.data.dataset.loader*), 132

Std (class in *qlib.data.ops*), 117

Sub (class in *qlib.data.ops*), 112

Sum (class in *qlib.data.ops*), 117

T

TanhProcess (class in *qlib.data.dataset.processor*), 139

task_generator() (in module *qlib.workflow.task.gen*), 162

task_stat() (*qlib.workflow.task.manage.TaskManager* method), 92, 168

task_train() (in module *qlib.model.trainer*), 169

TaskGen (class in *qlib.workflow.task.gen*), 89, 163

TaskManager (class in *qlib.workflow.task.manage*), 90, 165

TimeAdjuster (class in *qlib.workflow.task.utils*), 179

to_config() (*qlib.data.filter.BaseDFilter* method), 106

to_config() (*qlib.data.filter.ExpressionDFilter* method), 108

to_config() (*qlib.data.filter.NameDFilter* method), 107

train() (*qlib.model.trainer.DelayTrainerRM* method), 173

train() (*qlib.model.trainer.Trainer* method), 93, 170

train() (*qlib.model.trainer.TrainerR* method), 170

train() (*qlib.model.trainer.TrainerRM* method), 172

Trainer (class in *qlib.model.trainer*), 93, 169

TrainerR (class in *qlib.model.trainer*), 170

TrainerRM (class in *qlib.model.trainer*), 171

TResample (class in *qlib.data.ops*), 122

trunc_segments() (in module *qlib.workflow.task.gen*), 163

truncate() (*qlib.workflow.task.utils.TimeAdjuster* method), 180

U

update() (*qlib.data.cache.DatasetCache* method), 33, 124

update() (*qlib.data.cache.DiskDatasetCache* method), 126

update() (*qlib.data.cache.DiskExpressionCache* method), 124

update() (*qlib.data.cache.ExpressionCache* method), 32, 123

update() (*qlib.data.storage.file_storage.FileInstrumentStorage* method), 129

update() (*qlib.data.storage.storage.InstrumentStorage* method), 127

update() (*qlib.workflow.online.update.DSBasedUpdater* method), 84, 189

update() (*qlib.workflow.online.update.RecordUpdater* method), 83, 188

update_online_pred() (*qlib.workflow.online.utils.OnlineTool* method), 82, 187

update_online_pred() (*qlib.workflow.online.utils.OnlineToolR* method), 82, 187

uri (*qlib.workflow.expm.ExpManager* attribute), 56, 156

uri_context() (*qlib.workflow.__init__.QlibRecorder* method), 51

V

Var (class in *qlib.data.ops*), 117

W

wait() (*qlib.workflow.task.manage.TaskManager* method), 92, 168

WMA (class in *qlib.data.ops*), 120

worker() (*qlib.model.trainer.DelayTrainerRM* method), 174

worker() (*qlib.model.trainer.Trainer* method), 94, 170

worker() (*qlib.model.trainer.TrainerRM* method), 172

write() (*qlib.data.storage.file_storage.FileFeatureStorage* method), 129

write() (*qlib.data.storage.storage.FeatureStorage* method), 127

Z

ZScoreNorm (class in *qlib.data.dataset.processor*), 140