# QLib Documentation

***Release 0.6.2***

**Microsoft**

**Feb 02, 2021**

# Contents

`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.

CHAPTER 1

# Document Structure

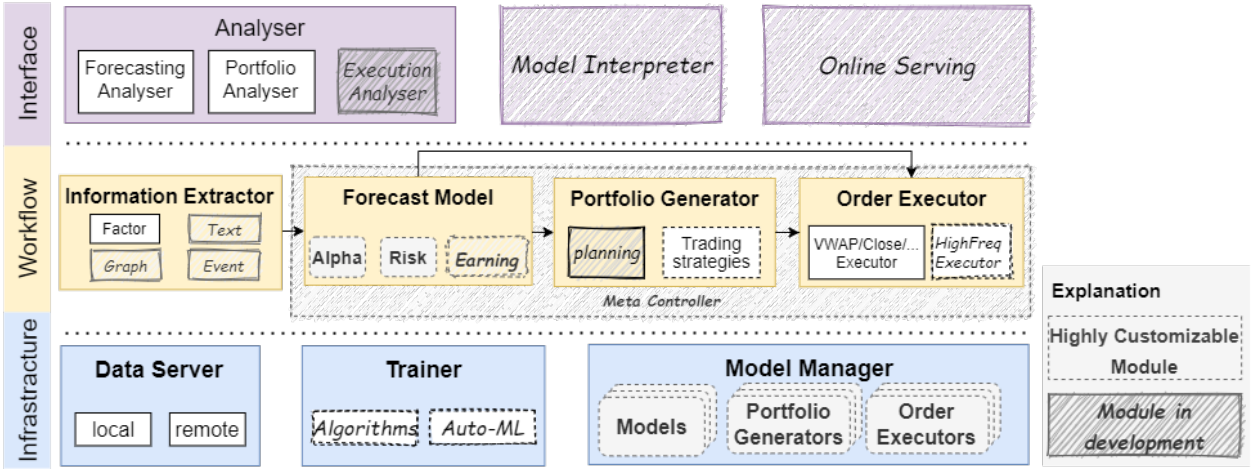
## 1.1 Qlib: Quantitative Platform

### 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 |
|---|---|
| *Infrastructure* layer | *Infrastructure* layer provides underlying support for Quant research. *DataServer* provides high-performance infrastructure for users to manage and retrieve raw data. *Trainer* provides flexible interface to control the training process of models which enable algorithms controlling the training process. |
| *Workflow* layer | *Workflow* layer covers the whole workflow of quantitative investment. *Information Extractor* extracts data for models. *Forecast Model* focuses on producing all kinds of forecast signals (e.g. _alpha_, risk) for other modules. With these signals *Portfolio Generator* will generate the target portfolio and produce orders to be executed by *Order Executor*. |
| *Interface* layer | *Interface* layer tries to present a user-friendly interface for the underlying system. *Analyser* 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 intsall `Qlib` according to the following steps:

- Before installing `Qlib` from source, users need to install some dependencies:

- Clone the repository and install `Qlib`

To kown 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 kown 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.config 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*:

- ***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.config.REG_CN*).** Currently: `qlib.config.REG_US` ('us') and `qlib.config.REG_CN` ('cn') is supported. Different value of *region* will result in different stock market mode. - `qlib.config.REG_US`: US stock market. - `qlib.config.REG_CN`: China stock market.

  Different modes will result in different trading limitations and costs.
- ***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.expm",
    "kwargs": {
        "uri": "python_execution_path/mlruns",
        "default_exp_name": "Experiment",
    }
})
```

## 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
   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])
>> 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()
```

(continues on next page)

(continued from previous page)

```
                            $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.

### 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:
        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 userd 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, and when users one 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"],
        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 hyper-parameter 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.gbdt
    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`.

```yaml
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
    backtest:
        verbose: False
        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.gbdt
        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]
```

(continues on next page)

(continued from previous page)

```
    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.

#### 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.gbdt
    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 preprossing 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
    backtest:
        verbose: False
        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

## 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 dataset, which can be accessed through this link:

| Dataset | US Market | China Market |
|---|---|---|
| Alpha360 | | |
| Alpha158 | | |

### 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.

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

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/csv_data/cn_data` directory and `~/.qlib/csv_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.

### 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.

Users can download the demo china-stock data in CSV format as follows for reference to the CSV format.

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

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, `factor = adjusted_price / 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.

---

#### 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 t**
  - 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/csv_data/cn_data`. Users only need to initialize `Qlib` as follows.

```
from qlib.config 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 china-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/csv_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)
```

### 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'

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
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 leanable `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: Tuple[dict, str, qlib.data.dataset.loader.DataLoader] = None*, *infer_processors=[]*, *learn_processors=[]*, *process_type='append'*, *drop_raw=False*, ***kwargs*)

DataHandler with **(L)earnable (P)rocessor**

**__init__**(*instruments=None*, *start_time=None*, *end_time=None*, *data_loader: Tuple[dict, str, qlib.data.dataset.loader.DataLoader] = None*, *infer_processors=[]*, *learn_processors=[]*, *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 processed by infer_processors
  - self._learn will be processed by learn_processors

  PTYPE_A = 'append'
  - self._infer will 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_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

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

**init**(*init_type: str = 'fit_seq'*, *enable_cache: bool = False*)

Initialize the data of Qlib

**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'*) → 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_*.

**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

If users want to load features and labels by config, users can inherit `qlib.data.dataset.handler.ConfigDataHandler`, `Qlib` also provides some preprocess method in this subclass.

If users want to use qlib data, *QLibDataHandler* is recommended. Users can inherit their custom class from *QLibDataHandler*, which is also a subclass of *ConfigDataHandler*.

### 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"))
```

### 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, qlib.data.dataset.handler.DataHandler], segments: dict*)

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, qlib.data.dataset.handler.DataHandler], segments: dict*)

**Parameters**

- **handler** (*Union[dict, DataHandler]*) – handler will be passed into setup_data.
- **segments** (*dict*) – handler will be passed into setup_data.

**init**(***kwargs*)

Initialize the DatasetH, Only parameters belonging to handler.init will be passed in

**setup_data**(*handler: Union[dict, qlib.data.dataset.handler.DataHandler], segments: dict*)

Setup the underlying data.

**Parameters**

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

**prepare**(*segments: Union[List[str], Tuple[str], str, slice], 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[str], Tuple[str], str, 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.
- **data_key** (*str*) – The data to fetch: DK_* Default is DK_I, which indicate fetching data for **inference**.

**Returns**

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

**Raises** NotImplementedError:

#### API

To know more about `Dataset`, please refer to **'Dataset API <../reference/api.html#module-qlib.data.dataset.__init__>'_**.

### 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*)
Update expression cache to latest calendar.

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

**Parameters** **cache_uri** (*str*) – the complete uri of expression cache file (include dir path).

**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*)
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*)
Update dataset cache to latest calendar.

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

**Parameters** **cache_uri** (*str*) – the complete uri of dataset cache file (include dir path).

**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
        - 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
```

(continues on next page)

(continued from previous page)

```
            - .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*)
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*) → object
give prediction given Dataset

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

**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

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.gbdt",
        "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",
```

(continues on next page)

(continued from previous page)

```
        "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()
    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`.

### 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 needs.

## 1.10.2 Base Class & Interface

### BaseStrategy

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

- ***get_risk_degree*** Return the proportion of your total value you will use in investment. Dynamically risk_degree will result in Market timing.
- ***generate_order_list*** Return the order list.

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

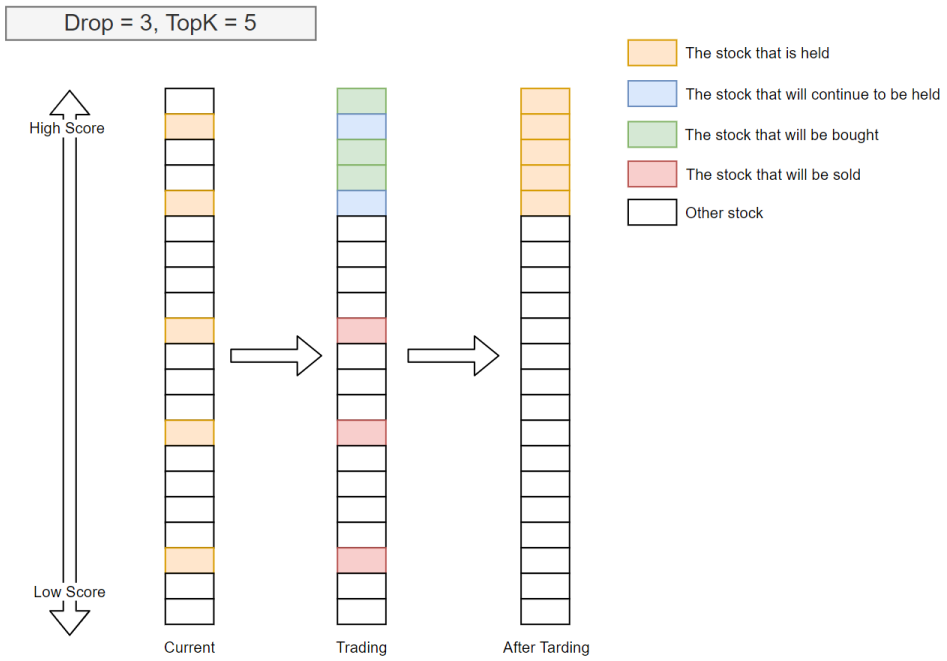
*TopkDropoutStrategy* is a subclass of *BaseStrategy* and implement the interface *generate_order_list* whose process is as follows.

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

  **Note:** `Topk-Drop` algorithm

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

  Currently, the number of held stocks is *Topk*. On each trading day, the *Drop* number of held stocks with the worst *prediction score* will be sold, and the same number of unheld stocks with the best *prediction score* will be bought.




  `TopkDrop` algorithm sells *Drop* stocks every trading day, which guarantees a fixed turnover rate.

- Generate the order list from the target amount

## 1.10.4 Usage & Example

`Portfolio Strategy` can be specified in the `Intraday Trading(Backtest)`, the example is as follows.

```
from qlib.contrib.strategy.strategy import TopkDropoutStrategy
from qlib.contrib.evaluate import backtest
STRATEGY_CONFIG = {
    "topk": 50,
    "n_drop": 5,
}
```

(continues on next page)

(continued from previous page)

```
BACKTEST_CONFIG = {
    "verbose": False,
    "limit_threshold": 0.095,
    "account": 100000000,
    "benchmark": BENCHMARK,
    "deal_price": "close",
    "open_cost": 0.0005,
    "close_cost": 0.0015,
    "min_cost": 5,

}
# use default strategy
strategy = TopkDropoutStrategy(**STRATEGY_CONFIG)

# pred_score is the `prediction score` output by Model
report_normal, positions_normal = backtest(
    pred_score, strategy=strategy, **BACKTEST_CONFIG
)
```

Also, the above example has been given in `examples/train_backtest_analyze.ipynb`.

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

To know more about `Intraday Trading`, please refer to Intraday Trading: Model&Strategy Testing.

### 1.10.5 Reference

To know more about `Portfolio Strategy`, please refer to Strategy API.

## 1.11 Intraday Trading: Model&Strategy Testing

### 1.11.1 Introduction

`Intraday Trading` is designed to test models and strategies, which help users to check the performance of a custom model/strategy.

---

**Note:** `Intraday Trading` uses `Order Executor` to trade and execute orders output by `Portfolio Strategy`. `Order Executor` is a component in Qlib Framework, which can execute orders. `VWAP Executor` and `Close Executor` is supported by `Qlib` now. In the future, `Qlib` will support `HighFreq Executor` also.

---

### 1.11.2 Example

Users need to generate a *prediction score'(a pandas DataFrame) with MultiIndex<instrument, datetime> and a 'score* column. And users need to assign a strategy used in backtest, if strategy is not assigned, a *TopkDropoutStrategy* strategy with *(topk=50, n_drop=5, risk_degree=0.95, limit_threshold=0.0095)* will be used. If `Strategy` module is not users' interested part, *TopkDropoutStrategy* is enough.

The simple example of the default strategy is as follows.

```
from qlib.contrib.evaluate import backtest
# pred_score is the prediction score
report, positions = backtest(pred_score, topk=50, n_drop=0.5, verbose=False, limit_
↪threshold=0.0095)
```

To know more about backtesting with a specific `Strategy`, please refer to Portfolio Strategy.

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

### 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.

### Backtest 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).

### 1.11.3 Reference

To know more about `Intraday Trading`, please refer to Intraday Trading.

## 1.12 Qlib Recorder: Experiment Management

### 1.12.1 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, pleaes refer to the related documents here.

### 1.12.2 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*)
A global system that helps to manage the experiments.

**__init__** (*exp_manager*)
Initialize self. See help(type(self)) for accurate signature.

**start** (*experiment_name=None*, *recorder_name=None*)
Method to start an experiment. This method can only be called within a Python's *with* statement. Here is the example code:

```
with R.start('test', 'recorder_1'):
    model.fit(dataset)
    R.log...
    ... # further operations
```

**Parameters**

- **experiment_name** (*str*) – name of the experiment one wants to start.
- **recorder_name** (*str*) – name of the recorder under the experiment one wants to start.

**start_exp** (*experiment_name=None*, *recorder_name=None*, *uri=None*)
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_name** (*str*) – the name of the experiment to be started
- **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.

**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*) → 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, and the experiment is set to be active.
  - 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, and 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.

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('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.

**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.

**get_recorder**(*recorder_id=None*, *recorder_name=None*, *experiment_name=None*)
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('test'):
    recorder = R.get_recorder()

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

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

# Case 4
```

(continues on next page)

(continued from previous page)

```
recorder = R.get_recorder(recorder_id='2e7a4efd66574fa49039e00ffaefa99d') ->␣
↪Error

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

**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).

- 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('test'):
    pred = model.predict(dataset)
    R.save_objects(**{"pred.pkl": pred}, artifact_path='prediction')

# Case 2
with R.start('test'):
    R.save_objects(local_path='results/pred.pkl')
```

**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.

**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, ...

### 1.12.3 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*, *default_exp_name*)

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*, *default_exp_name*)

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

**start_exp**(*experiment_name=None*, *recorder_name=None*, *uri=None*, ***kwargs*)

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

**Parameters**

- **experiment_name** (*str*) – name of the active experiment.
- **recorder_name** (*str*) – name of the recorder to be started.
- **uri** (*str*) – the current tracking URI.

**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=None*)

Create an experiment.

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

**Returns**

**Return type** An experiment object.

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

Get a pandas DataFrame of records that fit the search criteria of the experiment. Inputs are the search critera 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*)

Retrieve an experiment. This method includes getting an active experiment, and get_or_create a specific experiment. The returned experiment will be active.

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, and 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, and 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 it if hasn't been created before.

**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.

**get_uri**()
Get the default tracking URI or current URI.

**Returns**

**Return type** The tracking URI string.

**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.

### 1.12.4 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_name=None*)
Start the experiment and set it to be active. This method will also start a new recorder.

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

**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 critera 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*)
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, and the recorder shoud 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, and the recorder shoud 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.

**Returns**

**Return type** A recorder object.

**list_recorders**()
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*.

**Returns**

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

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*'.

### 1.12.5 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).

**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.

### 1.12.6 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.
- `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.

For more information about the APIs, please refer to Record Template API.

## 1.13 Analysis: Evaluation & Results Analysis

### 1.13.1 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

### 1.13.2 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)`

### 1.13.3 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:

```
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)

report_normal_df, _ = backtest(pred_df, strategy, **bparas)

qcr.report_graph(report_normal_df)
```

**Parameters**

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

| date | return | cost | bench | turnover |
|---|---|---|---|---|
| 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**.

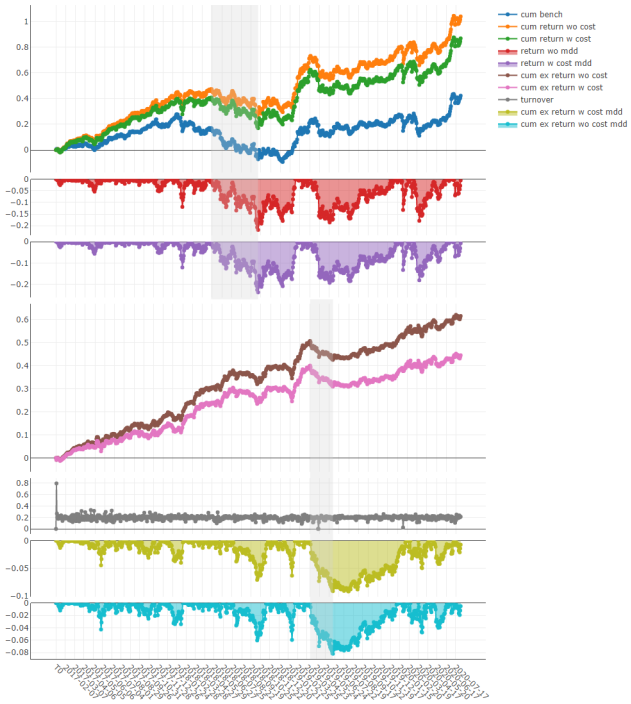
**Returns** if show_notebook is True, display in notebook; else return **plotly.graph_objs.Figure** list.

### 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']
```

(continues on next page)

(continued from previous page)

```
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.

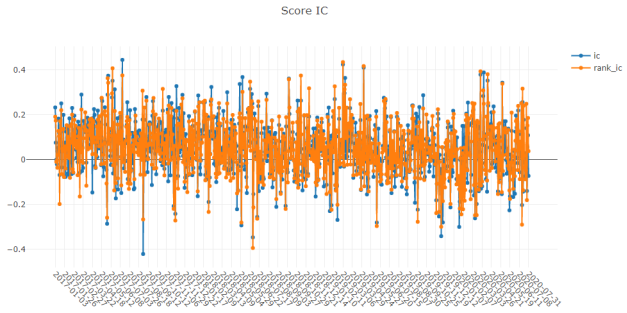
### Graphical Result

---

**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 Featrue 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*, *report_normal_df: pandas.core.frame.DataFrame = None*, *report_long_short_df: pandas.core.frame.DataFrame = None*, *show_notebook: bool = True*) → Iterable[plotly.graph_objs._figu

Generate analysis graph and monthly analysis

Example:

```
from qlib.contrib.evaluate import risk_analysis, backtest, long_
↪short_backtest
from qlib.contrib.strategy import TopkDropoutStrategy
from qlib.contrib.report import analysis_position

# backtest parameters
bparas = {}
bparas['limit_threshold'] = 0.095
bparas['account'] = 1000000000

sparas = {}
sparas['topk'] = 50
sparas['n_drop'] = 230
strategy = TopkDropoutStrategy(**sparas)

report_normal_df, positions = backtest(pred_df, strategy,␣
↪**bparas)
# long_short_map = long_short_backtest(pred_df)
# report_long_short_df = pd.DataFrame(long_short_map)

analysis = dict()
# analysis['pred_long'] = risk_analysis(report_long_short_df[
↪'long'])
# analysis['pred_short'] = risk_analysis(report_long_short_df[
↪'short'])
```

(continues on next page)

(continued from previous page)

```
# analysis['pred_long_short'] = risk_analysis(report_long_short_
↪df['long_short'])
analysis['excess_return_without_cost'] = risk_analysis(report_
↪normal_df['return'] - report_normal_df['bench'])
analysis['excess_return_with_cost'] = risk_analysis(report_
↪normal_df['return'] - report_normal_df['bench'] - report_
↪normal_df['cost'])
analysis_df = pd.concat(analysis)

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 |
| | 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 **return**, **turnover**, **cost**, **bench**.

| date | return | cost | bench | turnover |
|---|---|---|---|---|
| 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**.

| date | long | short | long_short |
|---|---|---|---|
| 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

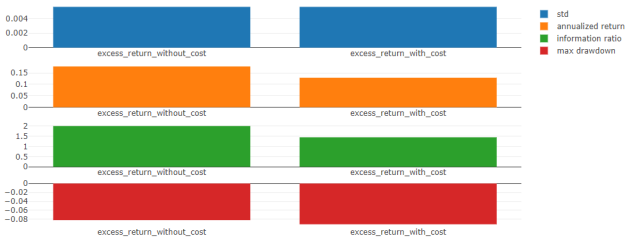
---

**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.

---

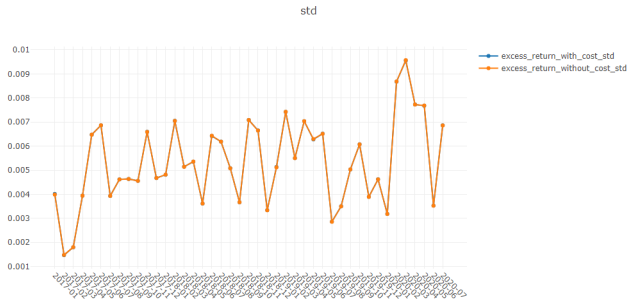



---

**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.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_l 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.

### Graphical Results

---

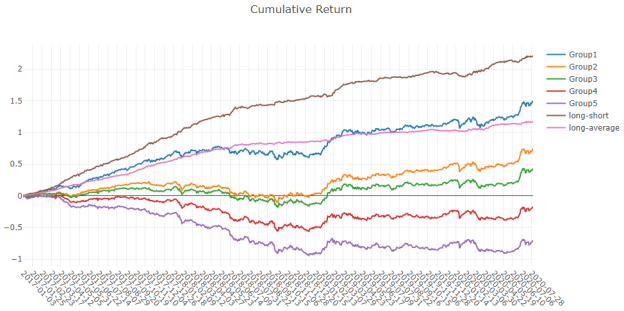
**Note:**

- **cumulative return graphics**
  - ***Group1*:** The *Cumulative Return* series of stocks group with (*ranking ratio* of label <= 20%)
  - ***Group2*:** The *Cumulative Return* series of stocks group with (20% < *ranking ratio* of label <= 40%)
  - ***Group3*:** The *Cumulative Return* series of stocks group with (40% < *ranking ratio* of label <= 60%)
  - ***Group4*:** The *Cumulative Return* series of stocks group with (60% < *ranking ratio* of label <= 80%)
  - ***Group5*:** The *Cumulative Return* series of stocks group with (80% < *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.**

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

---

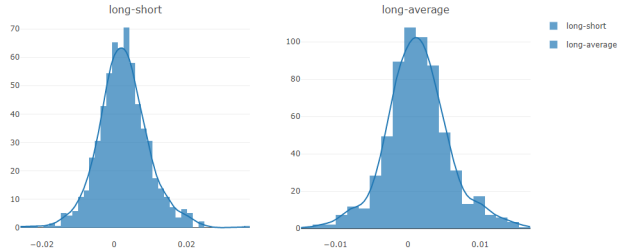



---

**Note:**

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

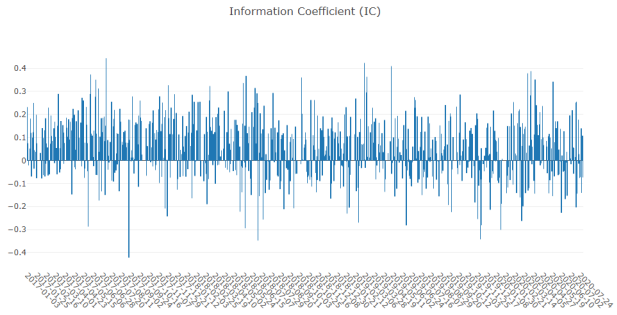
---




---

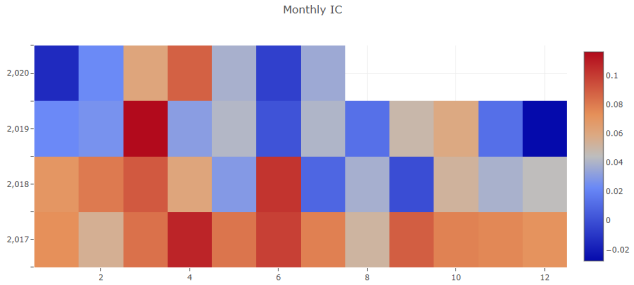
**Note:**

- **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*.



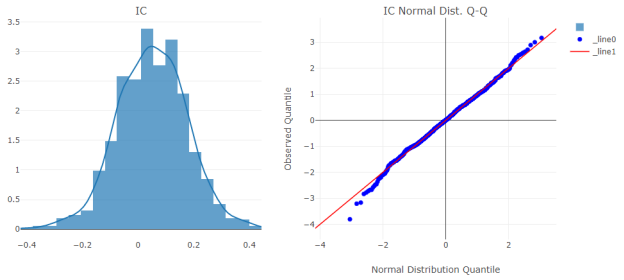

**Note:**

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




**Note:**

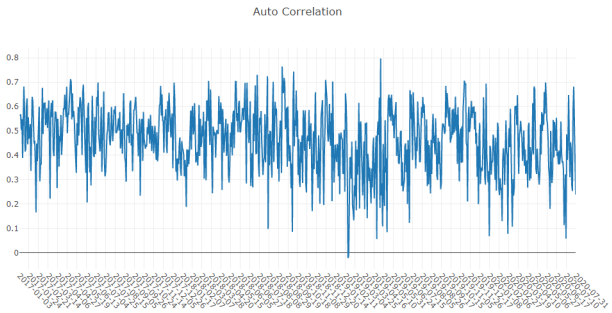
- **IC** The distribution of the *Information Coefficient* on each trading day.
- **IC Normal Dist. Q-Q** The *Quantile-Quantile Plot* is used for the normal distribution of *Information Coefficient* on each trading day.




**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.




## 1.14 Building Formulaic Alphas

### 1.14.1 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.

### 1.14.2 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),
..     "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
```

### 1.14.3 Reference

To learn more about `Data Loader`, please refer to Data Loader

To learn more about `Data API`, please refer to Data API

## 1.15 `Online` & `Offline` mode

### 1.15.1 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.

### 1.15.2 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.

### 1.15.3 Reference

If users are interested in `Qlib-Server` and `Online` mode, please refer to Qlib-Server Project and Qlib-Server Document.

## 1.16 API Reference

Here you can find all `Qlib` interfaces.

### 1.16.1 Data

#### Provider

**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*)

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.

**class** qlib.data.data.**InstrumentProvider**

Instrument provider base class

Provide instrument data.

**static instruments**(*market='all'*, *filter_pipe=None*)

Get the general config dictionary for a base market adding several dynamic filters.

**Parameters**

- **market** (*str*) – market/industry/index shortname, e.g. all/sse/szse/sse50/csi300/csi500.
- **filter_pipe** (*list*) – the list of dynamic filters.

**Returns**

dict of stockpool config. {'market'=>base market name, 'filter_pipe'=>list of filters}

example :

**Return type** dict

**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.**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'*)

Get Expression 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 expression

**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'*)

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.

**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*)

Load and process the data, return the data set. - default using multi-kernel method.

**static expression_calculator**(*inst*, *start_time*, *end_time*, *freq*, *column_names*, *spans=None*, *g_config=None*)

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**(***kwargs*)
Local calendar data provider class

Provide calendar data from local data source.

**__init__**(***kwargs*)
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.

**Returns** list of timestamps

**Return type** list

**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.**LocalInstrumentProvider**
Local instrument data provider class

Provide instrument data from local data source.

**__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.**LocalFeatureProvider**(***kwargs*)
Local feature data provider class

Provide feature data from local data source.

**__init__**(***kwargs*)
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.**LocalExpressionProvider**
Local expression data provider class

Provide expression data from local data source.

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

**expression**(*instrument*, *field*, *start_time=None*, *end_time=None*, *freq='day'*)
Get Expression 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 expression

**Return type** pd.Series

**class** qlib.data.data.**LocalDatasetProvider**
Local dataset data provider class

Provide dataset data from local data source.

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

**dataset**(*instruments*, *fields*, *start_time=None*, *end_time=None*, *freq='day'*)
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.

**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*)
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.

**Returns** a pandas dataframe with <instrument, datetime> index.

**Return type** pd.DataFrame

**class** qlib.data.data.**BaseProvider**
Local provider class

To keep compatible with old qlib provider.

**features**(*instruments*, *fields*, *start_time=None*, *end_time=None*, *freq='day'*, *disk_cache=None*)

**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 reponse 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.**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*)
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*)

**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.

**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.

**keep: bool** whether to keep the instruments of which features don't exist in the filter time span.

**from_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

**load**(*instrument*, *start_index*, *end_index*, *freq*)
load feature

**Parameters**

- **instrument** (*str*) – instrument code.
- **start_index** (*str*) – feature start index [in calendar].
- **end_index** (*str*) – feature end index [in calendar].
- **freq** (*str*) – feature frequency.

**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 + rght_etd].

**Returns** lft_etd, rght_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 + rght_etd].

**Returns** lft_etd, rght_etd

**Return type** (int, int)

**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 + rght_etd].

**Returns** lft_etd, rght_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.**Power**(*feature*, *exponent*)
Feature Power
**Parameters** **feature** (Expression) – feature instance

**Returns** a feature instance with power

**Return type** *Expression*

**__init__**(*feature*, *exponent*)
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 + rght_etd].

**Returns** lft_etd, rght_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.**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 + rght_etd].

**Returns** lft_etd, rght_etd

**Return type** (int, int)

**class** qlib.data.ops.**Rolling**(*feature*, *N*, *func*)
Rolling Operator

**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 + rght_etd].

**Returns** lft_etd, rght_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 + rght_etd].

**Returns** lft_etd, rght_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
**Parameters**
- **feature** (Expression) – feature instance
- **N** (*int*) – rolling window size

**Returns** a feature instance with 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 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.**OpsWrapper**
Ops Wrapper

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

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*)
Update expression cache to latest calendar.

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

**Parameters** **cache_uri** (*str*) – the complete uri of expression cache file (include dir path).

**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*)
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*)
Update dataset cache to latest calendar.

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

**Parameters** **cache_uri** (*str*) – the complete uri of dataset cache file (include dir path).

**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*)
Update expression cache to latest calendar.

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

**Parameters** **cache_uri** (*str*) – the complete uri of expression cache file (include dir path).

**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*, *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*)

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*)

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

**gen_dataset_cache**(*cache_path*, *instruments*, *fields*, *freq*)

---

**Note:** This function does not consider the cache read write lock. Please

---

Aquire 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.

:return type pd.DataFrame; The fields of the returned DataFrame are consistent with the parameters of the function.

**update**(*cache_uri*)

Update dataset cache to latest calendar.

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

**Parameters** **cache_uri** (*str*) – the complete uri of dataset cache file (include dir path).

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

**Return type** int

### Dataset

### Dataset Class

**class** qlib.data.dataset.__init__.**Dataset**(**args*, ***kwargs*)

Preparing data for model training and inferencing.

**__init__**(**args*, ***kwargs*)

init is designed to finish following steps:

- **setup data**
  - The data related attributes' names should start with '_' so that it will not be saved on disk when serializing.
- **initialize the state of the dataset(info to prepare the data)**
  - The name of essential state for preparing data should not start with '_' so that it could be serialized on disk when serializing.

The data could specify the info to caculate the essential data for preparation

**setup_data**(**args*, ***kwargs*)

Setup the data.

We split the setup_data function for following situation:

- User have a Dataset object with learned status on disk.
- User load the Dataset object from the disk(Note the init function is skiped).
- User call *setup_data* to load new data.
- User prepare data for model based on previous status.

**prepare**(**args*, ***kwargs*) → object
The type of dataset depends on the model. (It could be pd.DataFrame, pytorch.DataLoader, etc.) The parameters should specify the scope for the prepared data The method should: - process the data

- return the processed data

**Returns** return the object

**Return type** object

**class** qlib.data.dataset.__init__.**DatasetH**(*handler: Union[dict, qlib.data.dataset.handler.DataHandler], segments: dict*)
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, qlib.data.dataset.handler.DataHandler], segments: dict*)

**Parameters**

- **handler** (*Union[dict, DataHandler]*) – handler will be passed into setup_data.
- **segments** (*dict*) – handler will be passed into setup_data.

**init**(***kwargs*)
Initialize the DatasetH, Only parameters belonging to handler.init will be passed in

**setup_data**(*handler: Union[dict, qlib.data.dataset.handler.DataHandler], segments: dict*)
Setup the underlying data.

**Parameters**

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

**prepare**(*segments: Union[List[str], Tuple[str], str, slice], 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[str], Tuple[str], str, 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.
- **data_key** (*str*) – The data to fetch: DK_* Default is DK_I, which indicate fetching data for **inference**.

**Returns**

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

**Raises** NotImplementedError:

**class** qlib.data.dataset.__init__.**TSDataSampler**(*data: pandas.core.frame.DataFrame*, *start*, *end*, *step_len: int*, *fillna_type: str = 'none'*)

(T)ime-(S)eries DataSampler This is the result of TSDatasetH

It works like *torch.data.utils.Dataset*, it provides a very convient interface for constructing time-series dataset based on tabular data.

If user have further requirements for processing data, user could process them based on *TSDataSampler* or create more powerful subclasses.

Known Issues: - For performance issues, this Sampler will convert dataframe into arrays for better performance. This could result
in a different data type

**__init__**(*data: pandas.core.frame.DataFrame*, *start*, *end*, *step_len: int*, *fillna_type: str = 'none'*)

Build a dataset which looks like torch.data.utils.Dataset.

**Parameters**

- **data** (*pd.DataFrame*) – The raw tabular data
- **start** – The indexable start time
- **end** – The indexable end time
- **step_len** (*int*) – The length of the time-series step
- **fillna_type** (*int*) – How will qlib handle the sample if there is on sample in a specific date. none:

  fill with np.nan

  **ffill:** ffill with previous sample

  **ffill+bfill:** ffill with previous samples first and fill with later samples second

**get_index**()

Get the pandas index of the data, it will be useful in following scenarios - Special sampler will be used (e.g. user want to sample day by day)

**static build_index**(*data: pandas.core.frame.DataFrame*) → dict

The relation of the data

**Parameters** **data** (*pd.DataFrame*) – The dataframe with <datetime, DataFrame>

**Returns** {<index>: <prev_index or None>} # get the previous index of a line given index

**Return type** dict

**class** qlib.data.dataset.__init__.**TSDatasetH**(*step_len=30*, **args*, ***kwargs*)

(T)ime-(S)eries Dataset (H)andler

Covnert the tabular data to Time-Series data

Requirements analysis

The typical workflow of a user to get time-series data for an sample - process features - slice proper data from data handler: dimension of sample <feature, > - Build relation of samples by <time, instrument> index

- Be able to sample times series of data <timestep, feature>
- It will be better if the interface is like "torch.utils.data.Dataset"

- **User could build customized batch based on the data**
  - The dimension of a batch of data <batch_idx, feature, timestep>

**__init__**(*step_len=30*, **args*, ***kwargs*)

**Parameters**

- **handler** (*Union[dict, DataHandler]*) – handler will be passed into setup_data.
- **segments** (*dict*) – handler will be passed into setup_data.

**setup_data**(**args*, ***kwargs*)

Setup the underlying data.

**Parameters**

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

### 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: Tuple[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: Tuple[list, tuple, dict]*)

**Parameters** **config** (*Tuple[list, tuple, dict]*) – Config will be used to describe the fields and column names

**load_group_df**(*instruments*, *exprs: list*, *names: list*, *start_time=None*, *end_time=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(
↪$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.**QlibDataLoader**(*config: Tuple[list, tuple, dict]*, *filter_pipe=None*, *swap_level=True*, *freq='day'*)

Same as QlibDataLoader. The fields can be define by config

**__init__**(*config: Tuple[list, tuple, dict]*, *filter_pipe=None*, *swap_level=True*, *freq='day'*)

**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

**load_group_df**(*instruments*, *exprs: list*, *names: list*, *start_time=None*, *end_time=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: dict*, *join='outer'*)

DataLoader that supports loading data from file or as provided.

**__init__**(*config: dict*, *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(
↪$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

### Data Handler

**class** qlib.data.dataset.handler.**DataHandler**(*instruments=None*, *start_time=None*, *end_time=None*, *data_loader: Tuple[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 suported(The order will 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
```

**__init__**(*instruments=None*, *start_time=None*, *end_time=None*, *data_loader: Tuple[dict, str, qlib.data.dataset.loader.DataLoader] = None*, *init_data=True*, *fetch_orig=True*)

**Parameters**

- **instruments** – The stock list to retrive.
- **start_time** – start_time of the original data.
- **end_time** – end_time of the original data.
- **data_loader** (*Tuple[dict, str, DataLoader]*) – data loader to load the data.
- **init_data** – intialize the original data in the constructor.
- **fetch_orig** (*bool*) – Return the original data instead of copy if possible.

**conf_data**(***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.

**init**(*enable_cache: bool = False*)

initialize the data. In case of running intialization for multiple time, it will do nothing for the second 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.tslibs.timestamps.Timestamp, slice, str] = slice(None, None, None), level: Union[str, int] = 'datetime', col_set: Union[str, List[str]] = '__all', squeeze: bool = False*) → 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** (*Union[str, List[str]]*) –
  - if isinstance(col_set, str):

    select a set of meaningful columns.(e.g. features, columns)

    **if cal_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
- **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: Tuple[dict, str, qlib.data.dataset.loader.DataLoader] = None*, *infer_processors=[]*, *learn_processors=[]*, *process_type='append'*, *drop_raw=False*, ***kwargs*)

DataHandler with **(L)earnable (P)rocessor**

**__init__**(*instruments=None*, *start_time=None*, *end_time=None*, *data_loader: Tuple[dict, str, qlib.data.dataset.loader.DataLoader] = None*, *infer_processors=[]*, *learn_processors=[]*, *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 processed by infer_processors
  - self._learn will be processed by learn_processors

  PTYPE_A = 'append'
  - self._infer will 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_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

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

**init**(*init_type: str = 'fit_seq'*, *enable_cache: bool = False*)

Initialize the data of Qlib

**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'*) → 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_*.

**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

### Processor

qlib.data.dataset.processor.**get_group_columns**(*df: pandas.core.frame.DataFrame, group: 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 reiles 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 infenrece.

**Return type** bool

**class** qlib.data.dataset.processor.**DropnaProcessor**(*fields_group=None*)

**__init__** (*fields_group=None*)
Initialize self. See help(type(self)) for accurate signature.

**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.

**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.

**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*)
learn data processing parameters

**Parameters** **df** (*pd.DataFrame*) – When we fit and process data with processor one by one. The fit function reiles 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*)
learn data processing parameters

**Parameters** **df** (*pd.DataFrame*) – When we fit and process data with processor one by one. The fit function reiles 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:** mean(x) = median(x) std(x) = 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*)
learn data processing parameters

**Parameters** **df** (*pd.DataFrame*) – When we fit and process data with processor one by one. The fit function reiles on the output of previous processor, i.e. *df*.

**class** qlib.data.dataset.processor.**CSZScoreNorm**(*fields_group=None*)
Cross Sectional ZScore Normalization

**__init__**(*fields_group=None*)
Initialize self. See help(type(self)) for accurate signature.

**class** qlib.data.dataset.processor.**CSRankNorm**(*fields_group=None*)
Cross Sectional Rank Normalization

**__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.

### 1.16.2 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*)
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
```

(continues on next page)

(continued from previous page)

```
)
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*) → object
give prediction given Dataset

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

**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(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

**class** qlib.contrib.strategy.strategy.**StrategyWrapper**(*inner_strategy*)
StrategyWrapper is a wrapper of another strategy. By overriding some methods to make some changes on the basic strategy Cost control and risk control will base on this class.

**__init__**(*inner_strategy*)

**Parameters** **inner_strategy** – set the inner strategy.

**class** qlib.contrib.strategy.strategy.**AdjustTimer**

Responsible for timing of position adjusting

This is designed as multiple inheritance mechanism due to: - the is_adjust may need access to the internel state of a strategy.

- it can be reguard as a enhancement to the existing strategy.

**is_adjust**(*trade_date*)

Return if the strategy can adjust positions on *trade_date* Will normally be used in strategy do trading with trade frequency

**class** qlib.contrib.strategy.strategy.**ListAdjustTimer**(*adjust_dates=None*)

**__init__**(*adjust_dates=None*)

**Parameters** **adjust_dates** – an iterable object, it will return a timelist for trading dates

**is_adjust**(*trade_date*)

Return if the strategy can adjust positions on *trade_date* Will normally be used in strategy do trading with trade frequency

**class** qlib.contrib.strategy.strategy.**WeightStrategyBase**(*order_generator_cls_or_obj=<class 'qlib.contrib.strategy.order_generator.OrderGenWI*
**args*, ***kwargs*)

**__init__**(*order_generator_cls_or_obj=<class 'qlib.contrib.strategy.order_generator.OrderGenWInteract'>*, **args*, ***kwargs*)

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

**generate_target_weight_position**(*score*, *current*, *trade_date*)

Generate target position from score for this date and the current position.The cash is not considered in the position

**Parameters**

- **score** (*pd.Series*) – pred score for this trade date, index is stock_id, contain 'score' column.
- **current** (*Position()*) – current position.
- **trade_exchange** (*Exchange()*) –
- **trade_date** (*pd.Timestamp*) – trade date.

**generate_order_list**(*score_series*, *current*, *trade_exchange*, *pred_date*, *trade_date*)

**Parameters**

- **score_series** (*pd.Seires*) – stock_id , score.
- **current** (*Position()*) – current of account.
- **trade_exchange** (*Exchange()*) – exchange.
- **trade_date** (*pd.Timestamp*) – date.

**class** qlib.contrib.strategy.strategy.**TopkDropoutStrategy**(*topk*, *n_drop*, *method_sell='bottom'*, *method_buy='top'*, *risk_degree=0.95*, *thresh=1*, *hold_thresh=1*, *only_tradable=False*, ***kwargs*)

**__init__**(*topk*, *n_drop*, *method_sell='bottom'*, *method_buy='top'*, *risk_degree=0.95*, *thresh=1*, *hold_thresh=1*, *only_tradable=False*, ***kwargs*)

**Parameters**

- **topk** (*int*) – the number of stocks in the portfolio.
- **n_drop** (*int*) – number of stocks to be replaced in each trading date.
- **method_sell** (*str*) – dropout method_sell, random/bottom.
- **method_buy** (*str*) – dropout method_buy, random/top.
- **risk_degree** (*float*) – position percentage of total value.
- **thresh** (*int*) – minimun holding days since last buy singal of the stock.
- **hold_thresh** (*int*) – minimum holding days before sell stock , will check current.get_stock_count(order.stock_id) >= self.thresh.
- **only_tradable** (*bool*) – will the strategy only consider the tradable stock when buying and selling. if only_tradable:

  strategy will make buy sell decision without checking the tradable state of the stock.

  **else:** strategy will make decision with the tradable state of the stock info and avoid buy and sell them.

**get_risk_degree**(*date*)

Return the proportion of your total value you will used in investment. Dynamically risk_degree will result in Market timing.

**generate_order_list**(*score_series*, *current*, *trade_exchange*, *pred_date*, *trade_date*)

Gnererate order list according to score_series at trade_date, will not change current.

**Parameters**

- **score_series** (*pd.Series*) – stock_id , score.
- **current** (*Position()*) – current of account.
- **trade_exchange** (*Exchange()*) – exchange.
- **pred_date** (*pd.Timestamp*) – predict date.
- **trade_date** (*pd.Timestamp*) – trade date.

### Evaluate

qlib.contrib.evaluate.**risk_analysis**(*r*, *N=252*)

Risk Analysis

**Parameters**

- **r** (*pandas.Series*) – daily return series.

- **N** (*int*) – scaler for annualizing information_ratio (day: 250, week: 50, month: 12).

qlib.contrib.evaluate.**backtest**(*pred*, *account=1000000000.0*, *shift=1*, *benchmark='SH000905'*, *verbose=True*, ***kwargs*)

This function will help you set a reasonable Exchange and provide default value for strategy :param - **backtest workflow related or commmon arguments**: :param pred: predict should has <datetime, instrument> index and one *score* column. :type pred: pandas.DataFrame :param account: init account value. :type account: float :param shift: whether to shift prediction by one day. :type shift: int :param benchmark: benchmark code, default is SH000905 CSI 500. :type benchmark: str :param verbose: whether to print log. :type verbose: bool :param - **strategy related arguments**: :param strategy: strategy used in backtest. :type strategy: Strategy() :param topk: top-N stocks to buy. :type topk: int (Default value: 50) :param margin:

- if isinstance(margin, int):

  sell_limit = margin

- else:

  sell_limit = pred_in_a_day.count() * margin

buffer margin, in single score_mode, continue holding stock if it is in nlargest(sell_limit). sell_limit should be no less than topk.

**Parameters**

- **n_drop** (*int*) – number of stocks to be replaced in each trading date.
- **risk_degree** (*float*) – 0-1, 0.95 for example, use 95% money to trade.
- **str_type** (*'amount', 'weight' or 'dropout'*) – strategy type: TopkAmountStrategy ,TopkWeightStrategy or TopkDropoutStrategy.
- **exchange related arguments** (–) –
- **exchange** (*Exchange()*) – pass the exchange for speeding up.
- **subscribe_fields** (*list*) – subscribe fields.
- **open_cost** (*float*) – open transaction cost. The default value is 0.002(0.2%).
- **close_cost** (*float*) – close transaction cost. The default value is 0.002(0.2%).
- **min_cost** (*float*) – min transaction cost.
- **trade_unit** (*int*) – 100 for China A.
- **deal_price** (*str*) – dealing price type: 'close', 'open', 'vwap'.
- **limit_threshold** (*float*) – limit move 0.1 (10%) for example, long and short with same limit.
- **extract_codes** (*bool*) – will we pass the codes extracted from the pred to the exchange.

  **Note:** This will be faster with offline qlib.

- **executor related arguments** (–) –
- **executor** (*BaseExecutor()*) – executor used in backtest.
- **verbose** (*bool*) – whether to print log.

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:

```
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)
```

(continues on next page)

(continued from previous page)

```
report_normal_df, _ = backtest(pred_df, strategy, **bparas)

qcr.report_graph(report_normal_df)
```

**Parameters**

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

| date | return | cost | bench | turnover |
|---|---|---|---|---|
| 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*, *report_normal: pandas.core.frame.*, *label_data: pandas.core.frame.*, *show_notebook*, *start_date=Non*, *end_date=None*) → Iterable[plotly.grap

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 = ['label']

qcr.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'])*

| | | label |
|---|---|---|
| instrument | datetime | |
| 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*, *report_normal_df: pandas.core.frame.DataFrame = None*, *report_long_short_df: pandas.core.frame.DataFrame = None*, *show_notebook: bool = True*) → Iterable[plotly.graph_objs._figu

Generate analysis graph and monthly analysis

Example:

```
from qlib.contrib.evaluate import risk_analysis, backtest,␣
↪long_short_backtest
from qlib.contrib.strategy import TopkDropoutStrategy
from qlib.contrib.report import analysis_position

# backtest parameters
bparas = {}
bparas['limit_threshold'] = 0.095
bparas['account'] = 1000000000

sparas = {}
sparas['topk'] = 50
sparas['n_drop'] = 230
strategy = TopkDropoutStrategy(**sparas)

report_normal_df, positions = backtest(pred_df, strategy,␣
↪**bparas)
# long_short_map = long_short_backtest(pred_df)
# report_long_short_df = pd.DataFrame(long_short_map)

analysis = dict()
# analysis['pred_long'] = risk_analysis(report_long_short_df[
↪'long'])
# analysis['pred_short'] = risk_analysis(report_long_short_df[
↪'short'])
# analysis['pred_long_short'] = risk_analysis(report_long_
↪short_df['long_short'])
analysis['excess_return_without_cost'] = risk_analysis(report_
↪normal_df['return'] - report_normal_df['bench'])
```

(continues on next page)

(continued from previous page)

```
analysis['excess_return_with_cost'] = risk_analysis(report_
↪normal_df['return'] - report_normal_df['bench'] - report_
↪normal_df['cost'])
analysis_df = pd.concat(analysis)

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 |
| | 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 **return**, **turnover**, **cost**, **bench**.

| date | return | cost | bench | turnover |
|---|---|---|---|---|
| 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**.

| date | long | short | long_short |
|---|---|---|---|
| 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: pandas.core.frame.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())
features_df.columns = ['label']

qcr.rank_label_graph(positions, features_df, pred_df_dates.
↪min(), pred_df_dates.max())
```

**Parameters**

- **position** – position data; **qlib.contrib.backtest.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']).*

```
                                label
instrument  datetime
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: pandas.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_l pandas.co lag: int = 1, N: int = 5, reverse= rank= graph_ list = ['grou 'pred_ 'pred_ show_ bool = True, show_* → [<class 'list'>, <class 'tuple'>]

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.

### 1.16.3 Workflow

#### Experiment Manager

**class** qlib.workflow.expm.**ExpManager**(*uri*, *default_exp_name*)

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*, *default_exp_name*)

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

**start_exp**(*experiment_name=None*, *recorder_name=None*, *uri=None*, ***kwargs*)

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

**Parameters**

- **experiment_name** (*str*) – name of the active experiment.
- **recorder_name** (*str*) – name of the recorder to be started.
- **uri** (*str*) – the current tracking URI.

**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=None*)
Create an experiment.

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

**Returns**

**Return type** An experiment object.

**search_records**(*experiment_ids=None*, ***kwargs*)
Get a pandas DataFrame of records that fit the search criteria of the experiment. Inputs are the search critera 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*)
Retrieve an experiment. This method includes getting an active experiment, and get_or_create a specific experiment. The returned experiment will be active.

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, and 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, and 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 it if hasn't been created before.

**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.

**get_uri**()
Get the default tracking URI or current URI.

**Returns**

**Return type** The tracking URI string.

**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_name=None*)
Start the experiment and set it to be active. This method will also start a new recorder.

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

**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 critera 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*)
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, and the recorder shoud 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, and the recorder shoud 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.

**Returns**

**Return type** A recorder object.

**list_recorders**()
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*.

**Returns**

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

### 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).

**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.

**__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*)
Load the stored records. Due to the fact that some problems occured when we tried to balancing a clean API with the Python's inheritance. This method has to be used in a rather ugly way, and we will try to fix them in the future:

```
sar = SigAnaRecord(recorder)
ic = sar.load(sar.get_path("ic.pkl"))
```

**Parameters** **name** (*str*) – the name for the file to be load.

**Returns**

**Return type** The stored records.

**list**()
List the stored records.

**Returns**

**Return type** A list of all the stored records.

**check**(*parent=False*)
Check if the records is properly generated and saved.

FileExistsError: whether the records are stored properly.

**class** qlib.workflow.record_temp.**SignalRecord**(*model=None*, *dataset=None*, *recorder=None*, ***kwargs*)
This is the Signal Record class that generates the signal prediction. This class inherits the RecordTemp class.

**__init__**(*model=None*, *dataset=None*, *recorder=None*, ***kwargs*)
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 stored records.

**Returns**

**Return type** A list of all the stored records.

**load**(*name='pred.pkl'*)
Load the stored records. Due to the fact that some problems occured when we tried to balancing a clean API with the Python's inheritance. This method has to be used in a rather ugly way, and we will try to fix them in the future:

```
sar = SigAnaRecord(recorder)
ic = sar.load(sar.get_path("ic.pkl"))
```

**Parameters** **name** (*str*) – the name for the file to be load.

**Returns**

**Return type** The stored records.

**class** qlib.workflow.record_temp.**SigAnaRecord**(*recorder*, *ana_long_short=False*, *ann_scaler=252*, ***kwargs*)
This is the Signal Analysis Record class that generates the analysis results such as IC and IR. This class inherits the RecordTemp class.

**__init__**(*recorder*, *ana_long_short=False*, *ann_scaler=252*, ***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 stored records.

**Returns**

**Return type** A list of all the stored records.

**class** qlib.workflow.record_temp.**PortAnaRecord**(*recorder*, *config*, ***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*

**__init__**(*recorder*, *config*, ***kwargs*)

**config["strategy"]** [dict] define the strategy class as well as the kwargs.

**config["backtest"]** [dict] define the backtest kwargs.

**generate**(***kwargs*)
Generate certain records such as IC, backtest etc., and save them.

**Parameters** **kwargs** –

**list**()
List the stored records.

**Returns**

**Return type** A list of all the stored records.

## 1.17 Qlib FAQ

### 1.17.1 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...

```
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()
                ...

        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.

## 1.18 Changelog

Here you can see the full list of changes between each QLib release.

### 1.18.1 Version 0.1.0

This is the initial release of QLib library.

### 1.18.2 Version 0.1.1

Performance optimize. Add more features and operators.

### 1.18.3 Version 0.1.2

- Support operator syntax. Now `High() - Low()` is equivalent to `Sub(High(), Low())`.
- Add more technical indicators.

### 1.18.4 Version 0.1.3

Bug fix and add instruments filtering mechanism.

### 1.18.5 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.

### 1.18.6 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 exising fields like `Close()` may be deprecated in the future.

### 1.18.7 Version 0.2.2

- Add `disk_cache` for reusing features (enabled by default).
- Add `qlib.contrib` for experimental model construction and evaluation.

### 1.18.8 Version 0.2.3

- Add `backtest` module
- Decoupling the Strategy, Account, Position, Exchange from the backtest module

### 1.18.9 Version 0.2.4

- Add `profit attribution` module
- Add `rick_control` and `cost_control` strategies

### 1.18.10 Version 0.3.0

- Add `estimator` module

### 1.18.11 Version 0.3.1

- Add `filter` module

### 1.18.12 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

### 1.18.13 Version 0.3.3

- Fix bug of `filter` module

### 1.18.14 Version 0.3.4

- Support for `finetune model`
- Refactor `fetcher` code

### 1.18.15 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 calender now, not on normal date
- Move some date config from `handler` to `trainer`

### 1.18.16 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>'_**
- 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)
```

---

### 1.18.17 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**

### 1.18.18 Version 0.4.2

- Refactor DataHandler
- Add `Alpha360` DataHandler

### 1.18.19 Version 0.4.3

- Implementing Online Inference and Trading Framework
- Refactoring The interfaces of backtest and strategy module.

### 1.18.20 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.

### 1.18.21 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.

### 1.18.22 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*.

### 1.18.23 Version 0.5.0

- **First opensource version**
  - Refine the docs, code
  - Add baselines
  - public data crawler

### 1.18.24 Version greater than Version 0.5.0

Please refer to Github release Notes

# Python Module Index

## q

# Index

## Symbols

## A



## B



## C

## D



## E



## F

## G

## I



## K



## L

## M



## N



## O



## P



## Q

## R



## S



## T



## U

## V



## W



## Z