

---

# QLib Documentation

*Release 0.5.1*

**Microsoft**

**Dec 01, 2020**



---

## Contents

---

<b>1</b>	<b>Document Structure</b>	<b>3</b>
1.1	Qlib: Quantitative Platform . . . . .	3
1.2	Quick Start . . . . .	5
1.3	Installation . . . . .	6
1.4	Qlib Initialization . . . . .	7
1.5	Data Retrieval . . . . .	8
1.6	Custom Model Integration . . . . .	10
1.7	Estimator: Workflow Management . . . . .	13
1.8	Data Layer: Data Framework&Usage . . . . .	24
1.9	Interday Model: Model Training & Prediction . . . . .	32
1.10	Interday Strategy: Portfolio Management . . . . .	35
1.11	Intraday Trading: Model&Strategy Testing . . . . .	38
1.12	Aanalysis: Evaluation & Results Analysis . . . . .	40
1.13	Building Formulaic Alphas . . . . .	52
1.14	Online & Offline mode . . . . .	54
1.15	API Reference . . . . .	54
1.16	Qlib FAQ . . . . .	89
1.17	Changelog . . . . .	90
	<b>Python Module Index</b>	<b>95</b>
	<b>Index</b>	<b>97</b>



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



## 1.1 Qlib: Quantitative Platform

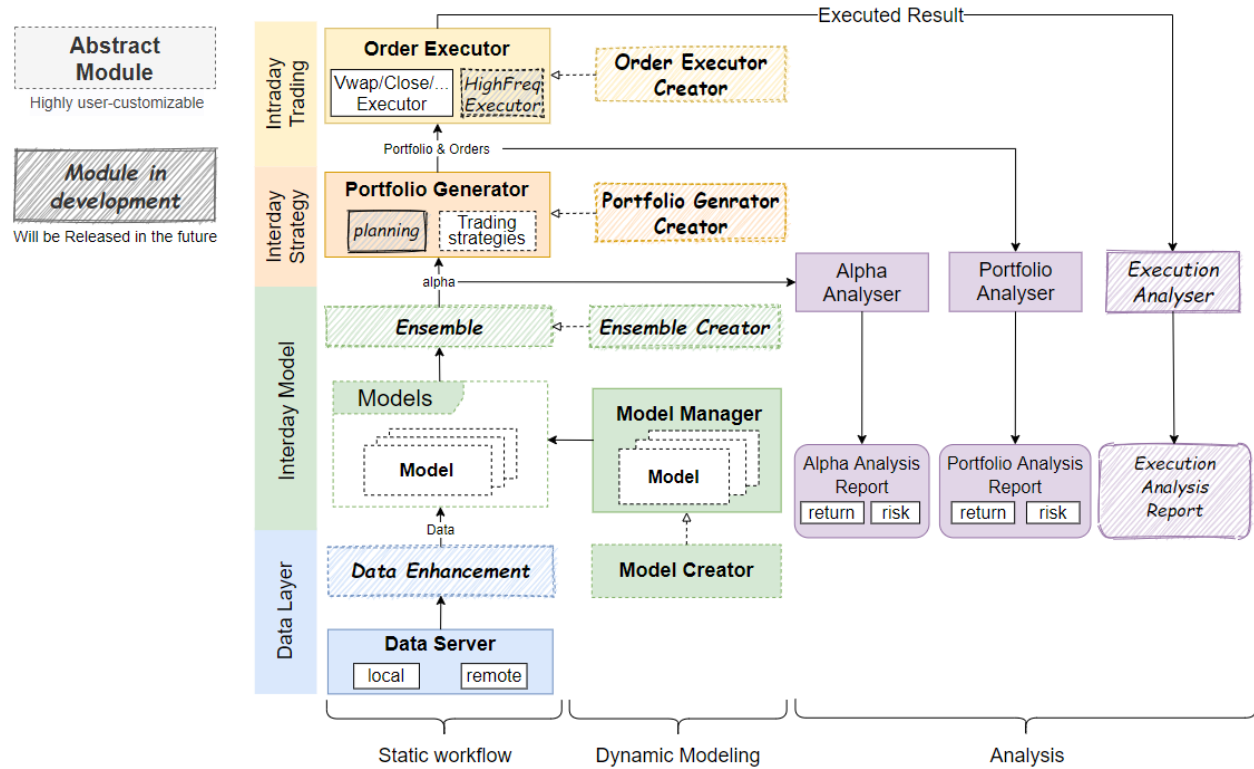
### 1.1.1 Introduction



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

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

## 1.1.2 Framework



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

Name	Description
<i>Data layer</i>	<i>DataServer</i> focuses on providing high-performance infrastructure for users to manage and retrieve raw data. <i>DataEnhancement</i> will preprocess the data and provide the best dataset to be fed into the models.
<i>Inter-day Model</i>	<i>Interday model</i> focuses on producing prediction scores (aka. <i>alpha</i> ). Models are trained by <i>Model Creator</i> and managed by <i>Model Manager</i> . Users could choose one or multiple models for prediction. Multiple models could be combined with <i>Ensemble</i> module.
<i>Inter-day Strategy</i>	<i>Portfolio Generator</i> will take prediction scores as input and output the orders based on the current position to achieve the target portfolio.
<i>Intra-day Trading</i>	<i>Order Executor</i> is responsible for executing orders output by <i>Interday Strategy</i> and returning the executed results.
<i>Analysis</i>	Users could get a detailed analysis report of forecasting signals and portfolios in this part.

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



## 1.2 Quick Start

### 1.2.1 Introduction

This `Quick Start` guide tries to demonstrate

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

### 1.2.2 Installation

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

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

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

### 1.2.3 Prepare Data

Load and prepare data by running the following code:

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

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

### 1.2.4 Auto Quant Research Workflow

`Qlib` provides a tool named `Estimator` 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 `Estimator` with `estimator_config.yaml` as following.**
  - **Estimator result** The result of `Estimator` is as follows, which is also the result of Intraday Trading. 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

(continues on next page)

(continued from previous page)

```
information_ratio 1.187411
max_drawdown      -0.075024
```

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

- **Graphical Reports Analysis:**

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

## 1.2.5 Custom Model Integration

Qlib provides `lightGBM` and `Dnn` model as the baseline of *Interday 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

```
python scripts/get_data.py qlib_data_cn --target_dir ~/.qlib/qlib_data/cn_data
```

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 mode 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.

---

## 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)
['SZ0000651', '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()
```

				\$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						

(continues on next page)

(continued from previous page)

	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 provides `lightGBM` and `Dnn` model as the baseline of `Interday Model`. In addition to the default model, 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.contrib.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.contrib.model.base.Model` and override the methods in it.

- **Override the `__init__` method**
  - QLib passes the initialized parameters to the `__init__` method.
  - The parameter must be consistent with the hyperparameters in the configuration file.
  - Code Example: In the following example, the hyperparameter filed of 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 `x_train`, training label `y_train`, test feature `x_valid`, test label `y_valid` at least.
- The parameters could include some optional parameters with default values, such as train weight `w_train`, test weight `w_valid` and `num_boost_round = 1000`.
- Code Example: In the following example, `num_boost_round = 1000` is an optional parameter.

```
def fit(self, x_train:pd.DataFrame, y_train:pd.DataFrame, x_valid:pd.
↳DataFrame, y_valid:pd.DataFrame,
    w_train:pd.DataFrame = None, w_valid:pd.DataFrame = None, num_boost_round_
↳= 1000, **kwargs):

    # Lightgbm need 1D array as its label
    if y_train.values.ndim == 2 and y_train.values.shape[1] == 1:
        y_train_1d, y_valid_1d = np.squeeze(y_train.values), np.squeeze(y_
↳valid.values)
    else:
        raise ValueError('LightGBM doesn\'t support multi-label training')

    w_train_weight = None if w_train is None else w_train.values
    w_valid_weight = None if w_valid is None else w_valid.values

    dtrain = lgb.Dataset(x_train.values, label=y_train_1d, weight=w_train_
↳weight)
    dvalid = lgb.Dataset(x_valid.values, label=y_valid_1d, weight=w_valid_
↳weight)
    self._model = lgb.train(
        self._params,
        dtrain,
        num_boost_round=num_boost_round,
        valid_sets=[dtrain, dvalid],
        valid_names=['train', 'valid'],
        **kwargs
    )
```

- **Override the *predict* method**

- The parameters include the test features.
- 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 `dnn` to predict the label(such as *preds*) of test data `x_test` and return it.

```
def predict(self, x_test:pd.DataFrame, **kwargs)-> numpy.ndarray:
    if self._model is None:
        raise ValueError('model is not fitted yet!')
    return self._model.predict(x_test.values)
```

- **Override the *save* method & *load* method**

- The *save* method parameter includes the a *filename* that represents an absolute path, user need to save model into the path.
- The *load* method parameter includes the a *buffer* read from the *filename* passed in the *save* method, users need to load model from the *buffer*.
- Code Example:

```
def save(self, filename):
    if self._model is None:
        raise ValueError('model is not fitted yet!')
    self._model.save_model(filename)

def load(self, buffer):
    self._model = lgb.Booster(params={'model_str': buffer.decode('utf-8')})
```

### 1.6.3 Configuration File

The configuration file is described in detail in the [estimator](#) document. In order to integrate the custom model into Qlib, users need to modify the “model” field in the configuration file.

- 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 baseline of the Model in `qlib/examples/estimator/estimator_config.yaml` and `qlib/examples/estimator/estimator_config_dnn.yaml`

### 1.6.4 Model Testing

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

```
cd examples # Avoid running program under the directory contains `qlib`
estimator -c estimator/estimator_config.yaml
```

---

**Note:** `estimator` is a built-in command of Qlib.

---

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

### 1.6.5 Reference

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



## 1.7 Estimator: 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 `Estimator` to automatically run the whole workflow defined by configuration. A concrete execution of the whole workflow is called an *experiment*. With `Estimator`, user can easily run an *experiment*, which includes the following steps:

- **Data**
  - Loading
  - Processing
  - Slicing
- **Model**
  - Training and inference(static or rolling)
  - Saving & loading
- **Evaluation(Back-testing)**

For each *experiment*, `Qlib` will capture the model training details, performance evaluation results and basic information (e.g. names, ids). The captured data will be stored in backend-storage (disk or database).

### 1.7.2 Complete Example

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

```
experiment:
  name: estimator_example
  observer_type: file_storage
  mode: train
model:
  class: LGBModel
  module_path: qlib.contrib.model.gbd
  args:
    loss: mse
    colsample_bytree: 0.8879
    learning_rate: 0.0421
    subsample: 0.8789
    lambda_11: 205.6999
    lambda_12: 580.9768
    max_depth: 8
    num_leaves: 210
    num_threads: 20
data:
  class: Alpha158
  args:
    dropna_label: True
  filter:
    market: csi500
```

(continues on next page)

(continued from previous page)

```
trainer:
  class: StaticTrainer
  args:
    rolling_period: 360
    train_start_date: 2007-01-01
    train_end_date: 2014-12-31
    validate_start_date: 2015-01-01
    validate_end_date: 2016-12-31
    test_start_date: 2017-01-01
    test_end_date: 2020-08-01
strategy:
  class: TopkDropoutStrategy
  args:
    topk: 50
    n_drop: 5
backtest:
  normal_backtest_args:
    verbose: False
    limit_threshold: 0.095
    account: 100000000
    benchmark: SH000905
    deal_price: close
    open_cost: 0.0005
    close_cost: 0.0015
    min_cost: 5
qlib_data:
  # when testing, please modify the following parameters according to the specific_
  ↪environment
  provider_uri: "~/qlib/qlib_data/cn_data"
  region: "cn"
```

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

```
estimator -c configuration.yaml
```

---

**Note:** *estimator* will be placed in your \$PATH directory when installing Qlib.

---

### 1.7.3 Configuration File

Let's get into details of *Estimator* in this section.

Before using *estimator*, users need to prepare a configuration file. The following content shows how to prepare each part of the configuration file.

#### Experiment Section

At first, the configuration file needs to contain a section named *experiment* about the basic information. This section describes how *estimator* tracks and persists current *experiment*. Qlib used *sacred*, a lightweight open-source tool, to configure, organize, generate logs, and manage experiment results. Partial behaviors of *sacred* will base on the *experiment* section.

Following files will be saved by *sacred* after *estimator* finish an *experiment*:

- *model.bin*, model binary file
- *pred.pkl*, model prediction result file
- *analysis.pkl*, backtest performance analysis file
- *positions.pkl*, backtest position records file
- *run*, the experiment information object, usually contains some meta information such as the experiment name, experiment date, etc.

Here is the typical configuration of *experiment* section

```
experiment:
  name: test_experiment
  observer_type: mongo
  mongo_url: mongodb://MONGO_URL
  db_name: public
  finetune: false
  exp_info_path: /home/test_user/exp_info.json
  mode: test
  loader:
    id: 677
```

The meaning of each field is as follows:

- **name** The experiment name, str type, *sacred* <<https://github.com/IDSIA/sacred>>\_ will use this experiment name as an identifier for some important internal processes. Users can find this field in *run* object of *sacred*. The default value is *test\_experiment*.
- **observer\_type** Observer type, str type, there are two choices which include *file\_storage* and *mongo* respectively. If *file\_storage* is selected, all the above-mentioned managed contents will be stored in the *dir* directory, separated by the number of times of experiments as a subfolder. If it is *mongo*, the content will be stored in the database. The default is *file\_storage*.
  - **For *file\_storage* observer.**
    - \* **dir** Directory URL, str type, directory for *file\_storage* observer type, files captured and managed by *sacred* with *file\_storage* observer will be saved to this directory, which is the same directory as *config.json* by default.
  - **For *mongo* observer.**
    - \* **mongo\_url** Database URL, str type, required if the observer type is *mongo*.
    - \* **db\_name** Database name, str type, required if the observer type is *mongo*.
- **finetune** Estimator's behaviors to train models will base on this flag. If you just want to train models from scratch each time instead of based on existing models, please leave *finetune=false*. Otherwise please read the details below.

The following table is the processing logic for different situations.

.	Static		Rolling	
.	finetune:true	finetune:false	finetune:true	finetune:false
Train	<ul style="list-style-type: none"> <li>– Need to provide model (Static or Rolling)</li> <li>– The args in model section will be used for finetuning</li> <li>– Update based on the provided model and parameters</li> </ul>	<ul style="list-style-type: none"> <li>– No need to provide model</li> <li>– The args in model section will be used for training</li> <li>– Train model from scratch</li> </ul>	<ul style="list-style-type: none"> <li>– Need to provide model (Static or Rolling)</li> <li>– The args in model section will be used for finetuning</li> <li>– Update based on the provided model and parameters</li> <li>– <b>Each rolling time slice is based on a model updated from the previous time</b></li> </ul>	<ul style="list-style-type: none"> <li>– Need to provide model (Static or Rolling)</li> <li>– The args in model section will be used for finetuning</li> <li>– Based on the provided model update</li> <li>– Train model from scratch</li> <li>– <b>Train each rolling time slice separately</b></li> </ul>
Test	<ul style="list-style-type: none"> <li>– Model must exist, otherwise an exception will be raised.</li> <li>– For <i>StaticTrainer</i>, users need to train a model and record ‘exp_info’ for ‘Test’.</li> <li>– For <i>RollingTrainer</i>, users need to train a set of models until the latest time, and record ‘exp_info’ for ‘Test’.</li> </ul>			

**Note:**

1. finetune parameters: share model.args parameters.
2. provide model: from *loader.model\_index*, load the index of the model(starting from 0).
3. **If *loader.model\_index* is None:**
  - In ‘Static Finetune=True’, if provide ‘Rolling’, use the last model to update.
  - For *RollingTrainer* with Finetune=True.
    - \* If *StaticTrainer* is used in loader, the model will be used for initialization for finetuning.
    - \* If *RollingTrainer* is used in loader, the existing models will be used without any modification and the new models will be initialized with the model in the last period and finetune one by one.

- ***exp\_info\_path*** save path of experiment info, str type, save the experiment info and model *prediction score* after the experiment is finished. Optional parameter, the default value is `<config_file_dir>/ex_name/exp_info.json`.

- ***mode***

*train or test*, str type.

- *test mode* is designed for inference. Under *test mode*, it will load the model according to the parameters of *loader* and skip model training.
- *train model* is the default value. It will train new models by default and

Please note that when it fails to load model, it will fall back to *fit* model.

---

**Note:** if users choose ‘test mode’, they need to make sure: - The loader of *test\_start\_date* must be less than or equal to the current *test\_start\_date*. - If other parameters of the *loader* model args are different, a warning will appear.

---

- **loader** If you just want to train models from scratch each time instead of based on existing models, please ignore *loader* section. Otherwise please read the details below.

The *loader* section only works when the *mode* is *test* or *finetune* is *true*.

- **model\_index** Model index, int type. The index of the loaded model in *loader\_models* (starting at 0) for the first *finetune*. The default value is *None*.
- **exp\_info\_path** Loader model experiment info path, str type. If the field exists, the following parameters will be parsed from *exp\_info\_path*, and the following parameters will not work. One of this field and *id* must exist at least .
- **id** The experiment id of the model that needs to be loaded, int type. If the *mode* is *test*, this value is required. This field and *exp\_info\_path* must exist one.
- **name** The experiment name of the model that needs to be loaded, str type. The default value is the current experiment *name*.
- **observer\_type** The experiment observer type of the model that needs to be loaded, str type. The default value is the current experiment *observer\_type*.

---

**Note:** The observer type is a concept of the *sacred* module, which determines how files, standard input, and output which are managed by *sacred* are stored.

---

- \* **file\_storage** If *observer\_type* is *file\_storage*, the config may be as follows.

```
experiment:
  name: test_experiment
  dir: <path to a directory> # default is dir of `config.yml`
  observer_type: file_storage
```

- \* **mongo** If *observer\_type* is *mongo*, the config may be as follows.

```
experiment:
  name: test_experiment
  observer_type: mongo
  mongo_url: mongodb://MONGO_URL
  db_name: public
```

Users need to indicate *mongo\_url* and *db\_name* for a mongo observer.

---

**Note:**

**If users choose the mongo observer, they need to make sure:**

- Have an environment with the *mongodb* installed and a mongo database dedicated to storing the results of the experiments.

- The python environment (the version of python and package) to run the experiments and the one to fetch the results are consistent.
- 

## Model Section

Users can use a specified model by configuration with hyper-parameters.

## Custom Models

Qlib supports custom models, but it must be a subclass of the *qlib.contrib.model.Model*, the config for a custom model may be as following.

```
model:
  class: SomeModel
  module_path: /tmp/my_experiment/custom_model.py
  args:
    loss: binary
```

The class *SomeModel* should be in the module *custom\_model*, and Qlib could parse the *module\_path* to load the class.

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

## Data Section

Data Handler can be used to load raw data, prepare features and label columns, preprocess data (standardization, remove NaN, etc.), split training, validation, and test sets. It is a subclass of *qlib.contrib.estimator.handler.BaseDataHandler*.

Users can use the specified data handler by config as follows.

```
data:
  class: Alpha158
  args:
    start_date: 2005-01-01
    end_date: 2018-04-30
    dropna_label: True
  filter:
    market: csi500
    filter_pipeline:
      -
        class: NameDFilter
        module_path: qlib.filter
        args:
          name_rule_re: S(?!\Z3)
          fstart_time: 2018-01-01
          fend_time: 2018-12-11
      -
        class: ExpressionDFilter
        module_path: qlib.filter
        args:
          rule_expression: $open/$factor<=45
          fstart_time: 2018-01-01
          fend_time: 2018-12-11
```

- **class** Data handler class, str type, which should be a subclass of `qlib.contrib.estimator.handler.BaseDataHandler`, and implements 5 important interfaces for loading features, loading raw data, preprocessing raw data, slicing train, validation, and test data. The default value is `ALPHA360`. If users want to write a data handler to retrieve the data in Qlib, `QLibDataHandler` is suggested.
- **module\_path** The module path, str type, absolute url is also supported, indicates the path of the class implementation of the data processor class. The default value is `qlib.contrib.estimator.handler`.
- **args** Parameters used for Data Handler initialization.
  - **train\_start\_date** Training start time, str type, the default value is `2005-01-01`.
  - **start\_date** Data start date, str type.
  - **end\_date** Data end date, str type. the data from start\_date to end\_date decides which part of data will be loaded in `datahandler`, users can only use these data in the following parts.
  - **dropna\_feature (Optional in args)** Drop Nan feature, bool type, the default value is `False`.
  - **dropna\_label (Optional in args)** Drop Nan label, bool type, the default value is `True`. Some multi-label tasks will use this.
  - **normalize\_method (Optional in args)** Normalize data by a given method. str type. Qlib gives two normalizing methods, `MinMax` and `Std`. If users want to build their own method, please override `_process_normalize_feature`.
- **filter** Dynamically filtering the stocks based on the filter pipeline.
  - **market** index name, str type, the default value is `csi500`.
  - **filter\_pipeline** Filter rule list, list type, the default value is `[]`. Can be customized according to users' needs.
    - \* **class** Filter class name, str type.
    - \* **module\_path** The module path, str type.
    - \* **args** The filter class parameters, these parameters are set according to the class, and all the parameters as kwargs to class.

## Custom Data Handler

Qlib support custom data handler, but it must be a subclass of the `qlib.contrib.estimator.handler.BaseDataHandler`, the config for custom data handler may be as follows.

```
data:
  class: SomeDataHandler
  module_path: /tmp/my_experiment/custom_data_handler.py
  args:
    start_date: 2005-01-01
    end_date: 2018-04-30
```

The class `SomeDataHandler` should be in the module `custom_data_handler`, and Qlib could parse the `module_path` to load the class.

If users want to load features and labels by config, they can inherit `qlib.contrib.estimator.handler.ConfigDataHandler`, Qlib also has provided some preprocess methods in this subclass. If users want to use qlib data, `QLibDataHandler` is recommended, from which users can inherit the custom class. `QLibDataHandler` is also a subclass of `ConfigDataHandler`.

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

## Trainer Section

Users can specify the trainer `Trainer` by the config file, which is a subclass of `qlib.contrib.estimator.trainer.BaseTrainer` and implement three important interfaces for training the model, restoring the model, and getting model predictions as follows.

- ***train*** Implement this interface to train the model.
- ***load*** Implement this interface to recover the model from disk.
- ***get\_pred*** Implement this interface to get model prediction results.

Qlib have provided two implemented trainer,

- ***StaticTrainer*** The static trainer will be trained using the training, validation, and test data of the data processor static slicing.
- ***RollingTrainer*** The rolling trainer will use the rolling iterator of the data processor to split data for rolling training.

Users can specify *trainer* with the configuration file:

```
trainer:
  class: StaticTrainer # or RollingTrainer
  args:
    rolling_period: 360
    train_start_date: 2005-01-01
    train_end_date: 2014-12-31
    validate_start_date: 2015-01-01
    validate_end_date: 2016-06-30
    test_start_date: 2016-07-01
    test_end_date: 2017-07-31
```

- ***class*** Trainer class, which should be a subclass of `qlib.contrib.estimator.trainer.BaseTrainer`, and needs to implement three important interfaces, the default value is *StaticTrainer*.
- ***module\_path*** The module path, str type, absolute url is also supported, indicates the path of the trainer class implementation.
- ***args*** Parameters used for `Trainer` initialization.
  - ***rolling\_period*** The rolling period, integer type, indicates how many time steps need rolling when rolling the data. The default value is *60*. Only used in *RollingTrainer*.
  - ***train\_start\_date*** Training start time, str type.
  - ***train\_end\_date*** Training end time, str type.
  - ***validate\_start\_date*** Validation start time, str type.
  - ***validate\_end\_date*** Validation end time, str type.
  - ***test\_start\_date*** Test start time, str type.
  - ***test\_end\_date*** Test end time, str type. If *test\_end\_date* is *-1* or greater than the last date of the data, the last date of the data will be used as *test\_end\_date*.

## Custom Trainer

Qlib supports custom trainer, but it must be a subclass of the `qlib.contrib.estimator.trainer.BaseTrainer`, the config for a custom trainer may be as following:



```

trainer:
  class: SomeTrainer
  module_path: /tmp/my_experiment/custom_trainer.py
  args:
    train_start_date: 2005-01-01
    train_end_date: 2014-12-31
    validate_start_date: 2015-01-01
    validate_end_date: 2016-06-30
    test_start_date: 2016-07-01
    test_end_date: 2017-07-31

```

The class *SomeTrainer* should be in the module *custom\_trainer*, and Qlib could parse the *module\_path* to load the class.

## Strategy Section

Users can specify strategy through a config file, for example:

```

strategy :
  class: TopkDropoutStrategy
  args:
    topk: 50
    n_drop: 5

```

- **class** The strategy class, str type, should be a subclass of *qlib.contrib.strategy.strategy.BaseStrategy*. The default value is *TopkDropoutStrategy*.
- **module\_path** The module location, str type, absolute url is also supported, and absolute path is also supported, indicates the location of the policy class implementation.
- **args** Parameters used for *Trainer* initialization.
  - **topk** The number of stocks in the portfolio
  - **n\_drop** Number of stocks to be replaced in each trading date

Qlib supports custom strategy, but it must be a subclass of the *qlib.contrib.strategy.strategy.BaseStrategy*, the config for custom strategy may be as following:

```

strategy :
  class: SomeStrategy
  module_path: /tmp/my_experiment/custom_strategy.py

```

The class *SomeStrategy* should be in the module *custom\_strategy*, and Qlib could parse the *module\_path* to load the class.

To know more about *Strategy*, please refer to [Strategy](#).

## Backtest Section

Users can specify *backtest* through a config file, for example:

```

backtest :
  normal_backtest_args:
    topk: 50
    benchmark: SH000905
    account: 500000

```

(continues on next page)

(continued from previous page)

```
deal_price: close
min_cost: 5
subscribe_fields:
  - $close
  - $change
  - $factor
```

- **normal\_backtest\_args** Normal backtest parameters. All the parameters in this section will be passed to the `qlib.contrib.evaluate.backtest` function in the form of ***\*\*kwargs***.
  - **benchmark** Stock index symbol, str, or list type, the default value is *None*.

**Note:**

- \* If *benchmark* is *None*, it will use the average change of the day of all stocks in ‘pred’ as the ‘bench’.
- \* If *benchmark* is list, it will use the daily average change of the stock pool in the list as the ‘bench’.
- \* If *benchmark* is str, it will use the daily change as the ‘bench’.

- **account** Backtest initial cash, integer type. The *account* in *strategy* section is deprecated. It only works when *account* is not set in *backtest* section. It will be overridden by *account* in the *backtest* section. The default value is 1e9.
- **deal\_price** Order transaction price field, str type, the default value is *close*.
- **min\_cost** Min transaction cost, float type, the default value is 5.
- **subscribe\_fields** Subscribe quote fields, array type, the default value is [*deal\_price*, *\$close*, *\$change*, *\$factor*].

**Qlib Data Section**

The *qlib\_data* field describes the parameters of qlib initialization.

```
qlib_data:
  # when testing, please modify the following parameters according to the specific_
  ↪environment
  provider_uri: "~/qlib/qlib_data/cn_data"
  region: "cn"
```

- **provider\_uri** The local directory where the data loaded by ‘get\_data.py’ is stored.
- **region**
  - If `region == qlib.config.REG_CN`, ‘qlib’ will be initialized in US-stock mode.
  - If `region == qlib.config.REG_US`, ‘qlib’ will be initialized in china-stock mode.

Please refer to [Initialization](#).

**1.7.4 Experiment Result**

## Form of Experimental Result

The result of the experiment is also the result of the Intraday Trading (Backtest), please refer to [Intraday Trading: Model&Strategy Testing](#).

## Get Experiment Result

### Base Class & Interface

Users can check the experiment results from file storage directly, or check the experiment results from the database, or get the experiment results through two interfaces of a base class *Fetcher* provided by *Qlib*.

The *Fetcher* provides the following interface

- ***get\_experiments(self, exp\_name=None)***: The interface takes one parameters. The *exp\_name* is the experiment name, the default is all experiments. Users can get the returned dictionary with a list of ids and test end date as follows.

```
{
  "ex_a": [
    {
      "id": 1,
      "test_end_date": "2017-01-01"
    }
  ],
  "ex_b": [
    ...
  ]
}
```

- ***get\_experiment(exp\_name, exp\_id, fields=None)*** The interface takes three parameters. The first parameter is the experiment name, the second parameter is the experiment id, and the third parameter is a list of fields. The default value of *fields* is *None*, which means all fields.

---

#### Note:

**Currently supported fields:** ['model', 'analysis', 'positions', 'report\_normal', 'pred', 'task\_config', 'label']

---

Users can get the returned dictionary as follows.

```
{
  'analysis': analysis_df,
  'pred': pred_df,
  'positions': positions_dic,
  'report_normal': report_normal_df,
}
```

## Implemented *Fetcher* s & Examples

*Qlib* provides two implemented *Fetcher* s as follows.

## FileFetcher

The *FileFetcher* is a subclass of *Fetcher*, which could fetch files from *file\_storage* observer. The following is an example: .. code-block:: python

```
>>> from qlib.contrib.estimator.fetcher import FileFetcher
>>> f = FileFetcher(experiments_dir=r'./')
>>> print(f.get_experiments())
{
  'test_experiment': [
    {
      'id': '1',
      'config': ...
    },
    {
      'id': '2',
      'config': ...
    },
    {
      'id': '3',
      'config': ...
    }
  ]
}
>>> print(f.get_experiment('test_experiment', '1'))
               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
```

## MongoFetcher

The *FileFetcher* is a subclass of *Fetcher*, which could fetch files from *mongo* observer. Users should initialize the fetcher with *mongo\_url*. The following is an example:

```
>>> from qlib.contrib.estimator.fetcher import MongoFetcher
>>> f = MongoFetcher(mongo_url=..., db_name=...)
```

# 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 Handler
- 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 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 dataset as follows.

```
python scripts/get_data.py qlib_data_cn --target_dir ~/.qlib/qlib_data/cn_data
```

After running the above command, users can find china-stock data in Qlib format in the `~/.qlib/csv_data/cn_data` directory.

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 data in CSV format into `.bin` files (Qlib format).

Users can download the 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
```

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 --csv_path ~/.qlib/csv_data/my_data --qlib_dir ~/.qlib/qlib_data/my_data --include_fields open,close,high,low,volume,factor
```

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 opening price
- *close* The closing price
- *high* The highest price
- *low* The lowest price
- *volume* The trading volume
- *factor* The Restoration factor

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.

---

## China-Stock Mode & US-Stock Mode

- If users use *Qlib* in china-stock mode, china-stock data is required. Users can use *Qlib* in china-stock mode according to

- Download china-stock in qlib format, please refer to section [Qlib Format Dataset](#).
- **Initialize *Qlib* in china-stock mode** Supposed that users download their Qlib format data in the directory `~/.qlib/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* does not provide a script to download US-stock data

- Prepare data in CSV format
- Convert data from CSV format to Qlib format, please refer to section [Converting CSV Format into Qlib Format](#).
- **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](#).

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 Handler

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

Also, `Data Handler` can be used as an independent module, by which users can easily preprocess data(standardization, remove NaN, etc.) and build datasets. It is a subclass of `qlib.contrib.estimator.handler.BaseDataHandler`, which provides some interfaces as follows.

### Base Class & Interface

Qlib provides a base class `qlib.contrib.estimator.BaseDataHandler`, which provides the following interfaces:

- **setup\_feature** Implement the interface to load the data features.
- **setup\_label** Implement the interface to load the data labels and calculate the users' labels.
- **setup\_processed\_data** Implement the interface for data preprocessing, such as preparing feature columns, discarding blank lines, and so on.

Qlib also provides two functions to help users init the data handler, users can override them for users' needs.

- **\_init\_kwargs** Users can init the kwargs of the data handler in this function, some kwargs may be used when init the raw df. Kwargs are the other attributes in `data.args`, like `dropna_label`, `dropna_feature`
- **\_init\_raw\_df** Users can init the raw df, feature names, and label names of data handler in this function. If the index of feature df and label df are not the same, users need to override this method to merge them (e.g. inner, left, right merge).

If users want to load features and labels by config, users can inherit `qlib.contrib.estimator.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*.

## Usage

Data Handler can be used as a single module, which provides the following methods:

- ***get\_split\_data***
  - According to the start and end dates, return features and labels of the pandas DataFrame type used for the 'Model'
- ***get\_rolling\_data***
  - According to the start and end dates, and *rolling\_period*, an iterator is returned, which can be used to traverse the features and labels used for rolling.

## Example

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

Know more about how to run Data Handler with Estimator, please refer to [Estimator: 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](#).

---

```
from qlib.contrib.estimator.handler import Alpha158
from qlib.contrib.model.gbdt import LGBModel

DATA_HANDLER_CONFIG = {
    "dropna_label": True,
    "start_date": "2007-01-01",
    "end_date": "2020-08-01",
    "market": "csi300",
}

TRAINER_CONFIG = {
    "train_start_date": "2007-01-01",
    "train_end_date": "2014-12-31",
    "validate_start_date": "2015-01-01",
    "validate_end_date": "2016-12-31",
    "test_start_date": "2017-01-01",
    "test_end_date": "2020-08-01",
}

exampleDataHandler = Alpha158(**DATA_HANDLER_CONFIG)

# example of 'get_split_data'
x_train, y_train, x_validate, y_validate, x_test, y_test = exampleDataHandler.get_
    ↪split_data(**TRAINER_CONFIG)
```

(continues on next page)



(continued from previous page)

```
# example of 'get_rolling_data'

for (x_train, y_train, x_validate, y_validate, x_test, y_test) in exampleDataHandler.
    ↪ get_rolling_data(**TRAINER_CONFIG):
        print(x_train, y_train, x_validate, y_validate, x_test, y_test)
```

**Note:** (x\_train, y\_train, x\_validate, y\_validate, x\_test, y\_test) can be used as arguments for the *fit*, *predic*’, and *score* methods of the Interday Model, please refer to [Model](#).

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

## API

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

### 1.8.5 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.
```

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

#### 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.

Override 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.

Override 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.

## 1.8.6 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(instrtmnt, field_expression, freq)]
        - all-time expression -cache data file
        - .meta : an assorted meta file recording the instrument name, field_
↪name, freq, and visit times
      - ...
  - cache/
    - [hash(stockpool_config, field_expression_list, freq)]
      - all-time Dataset-cache data file
      - .meta : an assorted meta file recording the stockpool config, field_
↪names and visit times
      - .index : an assorted index file recording the line index of all_
↪calendars
    - ...
```

## 1.9 Interday Model: Model Training & Prediction

### 1.9.1 Introduction

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

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

### 1.9.2 Base Class & Interface

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

The base class provides the following interfaces:

- `__init__(**kwargs)`
  - Initialization.
  - If users use Estimator to start an *experiment*, the parameter of `__init__` method should be consistent with the hyperparameters in the configuration file.
- `fit(self, x_train, y_train, x_valid, y_valid, w_train=None, w_valid=None, **kwargs)`
  - Train model.
  - **Parameter:**

\* **`x_train`, `pd.DataFrame` type, train feature** The following example explains the value of `x_train`:

		KMID	KLEN	KMID2	KUP
↪ KUP2					
instrument	datetime				
SH600004	2012-01-04	0.000000	0.017685	0.000000	0.
↪ 012862	0.727275				
	2012-01-05	-0.006473	0.025890	-0.250001	0.
↪ 012945	0.499998				
	2012-01-06	0.008117	0.019481	0.416666	0.
↪ 008117	0.416666				
	2012-01-09	0.016051	0.025682	0.624998	0.
↪ 006421	0.250001				
	2012-01-10	0.017323	0.026772	0.647057	0.
↪ 003150	0.117648				
...		...	...	...	...
↪ ...					
SZ300273	2014-12-25	-0.005295	0.038697	-0.136843	0.
↪ 016293	0.421052				
	2014-12-26	-0.022486	0.041701	-0.539215	0.
↪ 002453	0.058824				
	2014-12-29	-0.031526	0.039092	-0.806451	0.
↪ 000000	0.000000				
	2014-12-30	-0.010000	0.032174	-0.310811	0.
↪ 013913	0.432433				
	2014-12-31	0.010917	0.020087	0.543479	0.
↪ 001310	0.065216				

$x\_train$  is a pandas DataFrame, whose index is MultiIndex <instrument(str), date-time(pd.Timestamp)>. Each column of  $x\_train$  corresponds to a feature, and the column name is the feature name.

**Note:** The number and names of the columns are determined by the data handler, please refer to [Data Handler](#) and [Estimator Data Section](#).

\*  **$y\_train$ , pd.DataFrame type, train label** The following example explains the value of  $y\_train$ :

		LABEL
instrument	datetime	
SH600004	2012-01-04	-0.798456
	2012-01-05	-1.366716
	2012-01-06	-0.491026
	2012-01-09	0.296900
	2012-01-10	0.501426
...		...
SZ300273	2014-12-25	-0.465540
	2014-12-26	0.233864
	2014-12-29	0.471368
	2014-12-30	0.411914
	2014-12-31	1.342723

$y\_train$  is a pandas DataFrame, whose index is MultiIndex <instrument(str), date-time(pd.Timestamp)>. The *LABEL* column represents the value of train label.

**Note:** The number and names of the columns are determined by the Data Handler, please refer to [Data Handler](#).

\*  **$x\_valid$ , pd.DataFrame type, validation feature** The format of  $x\_valid$  is same as  $x\_train$

\*  **$y\_valid$ , pd.DataFrame type, validation label** The format of  $y\_valid$  is same as  $y\_train$

\*  **$w\_train$  (Optional args, default is None), pd.DataFrame type, train weight**

$w\_train$  is a pandas DataFrame, whose shape and index is same as  $x\_train$ . The float value in  $w\_train$  represents the weight of the feature at the same position in  $x\_train$ .

\*  **$w\_train$  (Optional args, default is None), pd.DataFrame type, validation weight**

$w\_train$  is a pandas DataFrame, whose shape and index is the same as  $x\_valid$ . The float value in  $w\_train$  represents the weight of the feature at the same position in  $x\_train$ .

•  **$predict(self, x\_test, **kwargs)$**

– Predict test data ‘ $x\_test$ ’

– **Parameter:**

\*  **$x\_test$ , pd.DataFrame type, test features** The form of  $x\_test$  is same as  $x\_train$  in ‘fit’ method.

– **Return:**

\*  **$label$ , np.ndarray type, test label** The label of  $x\_test$  that predicted by model.

•  **$score(self, x\_test, y\_test, w\_test=None, **kwargs)$**

- Evaluate model with test feature/label

- **Parameter:**

- \* **`x_test`, `pd.DataFrame` type, test feature** The format of `x_test` is same as `x_train` in `fit` method.

- \* **`x_test`, `pd.DataFrame` type, test label** The format of `y_test` is same as `y_train` in `fit` method.

- \* **`w_test`, `pd.DataFrame` type, test weight** The format of `w_test` is same as `w_train` in `fit` method.

- Return: float type, evaluation score

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

### 1.9.3 Example

Qlib provides LightGBM and DNN models as the baseline, 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.estimator.handler import Alpha158
from qlib.contrib.model.gbdt import LGBModel

DATA_HANDLER_CONFIG = {
    "dropna_label": True,
    "start_date": "2007-01-01",
    "end_date": "2020-08-01",
    "market": MARKET,
}

TRAINER_CONFIG = {
    "train_start_date": "2007-01-01",
    "train_end_date": "2014-12-31",
    "validate_start_date": "2015-01-01",
    "validate_end_date": "2016-12-31",
    "test_start_date": "2017-01-01",
    "test_end_date": "2020-08-01",
}

x_train, y_train, x_validate, y_validate, x_test, y_test = Alpha158(
    **DATA_HANDLER_CONFIG
).get_split_data(**TRAINER_CONFIG)

MODEL_CONFIG = {
    "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,
```

(continues on next page)

(continued from previous page)

```

    "num_threads": 20,
}
# use default model
model = LGBModel(**MODEL_CONFIG)
model.fit(x_train, y_train, x_validate, y_validate)
_pred = model.predict(x_test)
pred_score = pd.DataFrame(index=_pred.index)
pred_score["score"] = _pred.iloc(axis=1)[0]

```

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

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 Interday Strategy: Portfolio Management

## 1.10.1 Introduction

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

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

Qlib provides several implemented trading 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.
  - 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

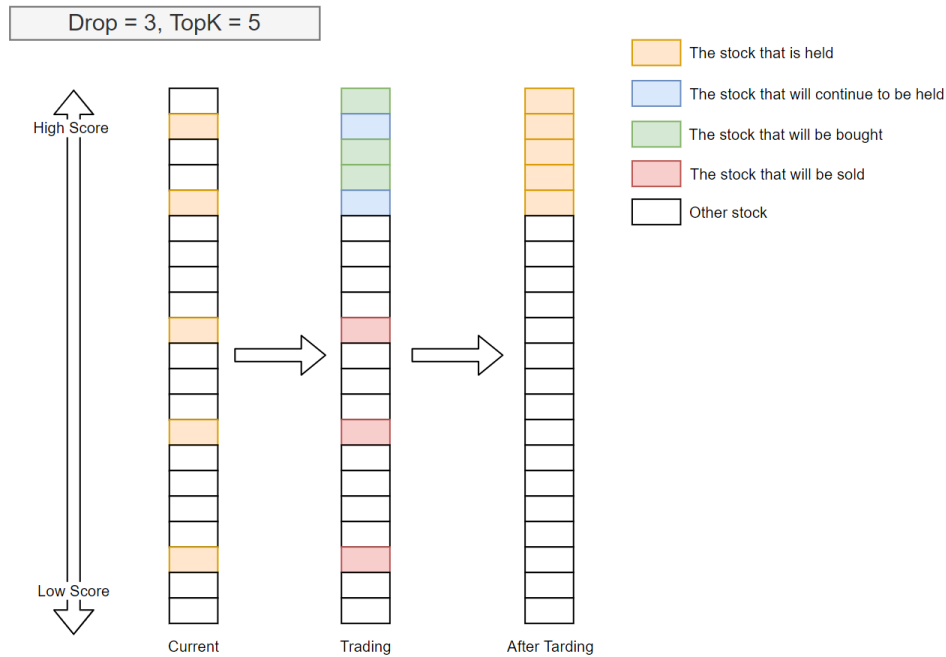
---

**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

Interday 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,
}
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(
```

(continues on next page)

(continued from previous page)

```
)  
    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 Interday Model, please refer to [Interday 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 Interday 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 Interday 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 [Strategy](#).

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

## Prediction Score

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

A prediction sample is shown as follows.

```

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

```

Interday Model module can make predictions, please refer to [Interday 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 Aanalysis: Evaluation & Results Analysis

### 1.12.1 Introduction

Aanalysis 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

### 1.12.2 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)`

---

### 1.12.3 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
```

(continues on next page)

(continued from previous page)

```
# 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**

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

- **show\_notebook** – whether to display graphics in notebook, the default is **True**

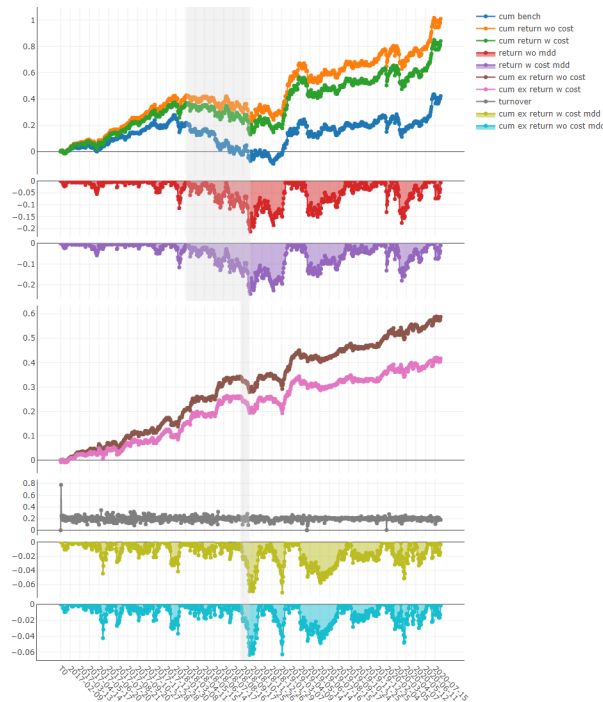
**Returns** if **show\_notebook** is **True**, display in notebook; else return **plotly.graph\_objs.Figure** list

### Graphical Result

#### Note:

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

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



## Usage of `analysis_position.score_ic`

### API

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

score IC

Example:

```
from qlib.data import D
from qlib.contrib.report import analysis_position
pred_df_dates = pred_df.index.get_level_values(level='datetime'
→ ')
features_df = D.features(D.instruments('csi500'), ['Ref($close,
→ -2)/Ref($close, -1)-1'], pred_df_dates.min(), pred_df_dates.
→ max())
```

(continues on next page)

(continued from previous page)

```

features_df.columns = ['label']
pred_label = pd.concat([features_df, pred], axis=1, sort=True).
    ↪reindex(features_df.index)
analysis_position.score_ic_graph(pred_label)

```

**Parameters**

- **pred\_label** – index is **pd.MultiIndex**, index name is **[instrument, datetime]**; columns names is **[score, label]**

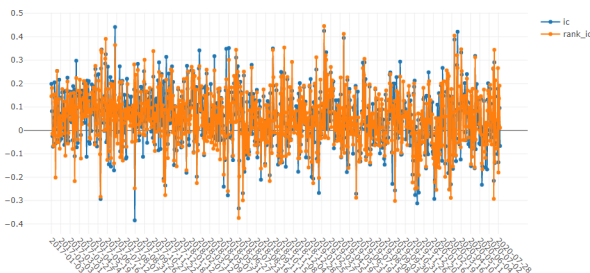
instrument	datetime	score	label
SH600004	2017-12-11	-0.013502	-0.013502
	2017-12-12	-0.072367	-0.072367
	2017-12-13	-0.068605	-0.068605
	2017-12-14	0.012440	0.012440
	2017-12-15	-0.102778	-0.102778

- **show\_notebook** – whether to display graphics in notebook, the default is **True**

**Returns** if show\_notebook is True, display in notebook; else return **plotly.graph\_objs.Figure** list

**Graphical Result****Note:**

- **Axis X:** Trading day
- **Axis Y:**
  - **ic** The *Pearson correlation coefficient* series between *label* and *prediction score*. In the above example, the *label* is formulated as  $Ref(\$close, -1)/\$close - 1$ . Please refer to [Data Feattrue](#) 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._figure.Figure]
```

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**

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

- **report\_long\_short\_df** – **df.index.name** must be **date**, **df.columns** contain **long**, **short**, **long\_short**

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

- **show\_notebook** – Whether to display graphics in a notebook, default **True** If True,

show graph in notebook If False, return graph figure

## Returns

## Graphical Result

### Note:

- general graphics

- *std*

- \* *excess\_return\_without\_cost* The *Standard Deviation* of CAR (cumulative abnormal return) without cost.

- \* *excess\_return\_with\_cost* The *Standard Deviation* of CAR (cumulative abnormal return) with cost.

- *annualized\_return*

- \* *excess\_return\_without\_cost* The *Annualized Rate* of CAR (cumulative abnormal return) without cost.

- \* *excess\_return\_with\_cost* The *Annualized Rate* of CAR (cumulative abnormal return) with cost.

- *information\_ratio*

- \* *excess\_return\_without\_cost* The *Information Ratio* without cost.

- \* *excess\_return\_with\_cost* The *Information Ratio* with cost.

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

- *max\_drawdown*

- \* *excess\_return\_without\_cost* The *Maximum Drawdown* of CAR (cumulative abnormal return) without cost.

- \* *excess\_return\_with\_cost* The *Maximum Drawdown* of CAR (cumulative abnormal return) with cost.



### Note:

- annualized\_return/max\_drawdown/information\_ratio/std graphics

- Axis X: Trading days grouped by month

- Axis Y:

- \* *annualized\_return* graphics

- ***excess\_return\_without\_cost\_annualized\_return*** The *Annualized Rate* series of monthly CAR (cumulative abnormal return) without cost.
- ***excess\_return\_with\_cost\_annualized\_return*** The *Annualized Rate* series of monthly CAR (cumulative abnormal return) with cost.

\* **max\_drawdown** graphics

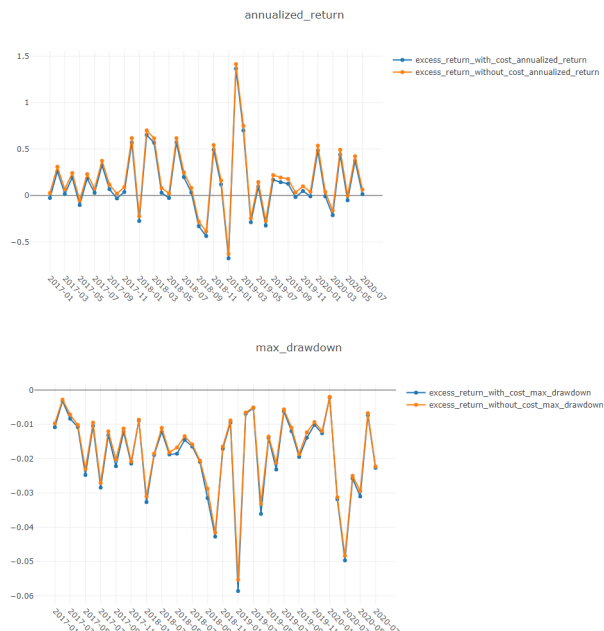
- ***excess\_return\_without\_cost\_max\_drawdown*** The *Maximum Drawdown* series of monthly CAR (cumulative abnormal return) without cost.
- ***excess\_return\_with\_cost\_max\_drawdown*** The *Maximum Drawdown* series of monthly CAR (cumulative abnormal return) with cost.

\* **information\_ratio** graphics

- ***excess\_return\_without\_cost\_information\_ratio*** The *Information Ratio* series of monthly CAR (cumulative abnormal return) without cost.
- ***excess\_return\_with\_cost\_information\_ratio*** The *Information Ratio* series of monthly CAR (cumulative abnormal return) with cost.

\* **std** graphics

- ***excess\_return\_without\_cost\_max\_drawdown*** The *Standard Deviation* series of monthly CAR (cumulative abnormal return) without cost.
- ***excess\_return\_with\_cost\_max\_drawdown*** The *Standard Deviation* series of monthly CAR (cumulative abnormal return) with cost.





## Usage of `analysis_model.analysis_model_performance`

### API

`qlib.contrib.report.analysis_model.analysis_model_performance.ic_figure` (*ic\_df*:

*pan-*  
*das.core.frame.DataFrame,*  
*show\_nature\_day=True,*  
*\*\*kwargs)*

→

`plotly.graph_objs._figure.Fi`

IC figure

#### Parameters

- **ic\_df** – ic DataFrame
- **show\_nature\_day** – whether to display the abscissa of non-trading day

**Returns** `plotly.graph_objs.Figure`

```

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

```

Model performance

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

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

#### Parameters

- **lag** – `pred.groupby(level='instrument')['score'].shift(lag)`. It will be only used in the auto-correlation computing.
- **N** – group number, default 5
- **reverse** – if `True`, `pred['score'] *= -1`
- **rank** – if `True`, calculate rank ic
- **graph\_names** – graph names; default `['cumulative_return', 'pred_ic', 'pred_autocorr', 'pred_turnover']`
- **show\_notebook** – whether to display graphics in notebook, the default is `True`

- **show\_nature\_day** – whether to display the abscissa of non-trading day

**Returns** if show\_notebook is True, display in notebook; else return `plotly.graph_objs.Figure` list

## Graphical Results

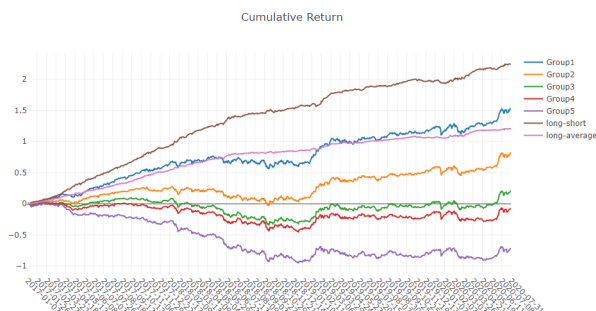
**Note:**

- **cumulative return graphics**

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

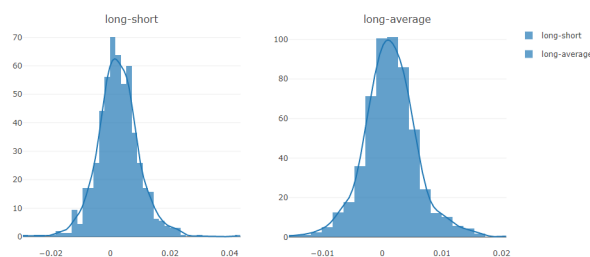
The *ranking ratio* can be formulated as follows.

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



**Note:**

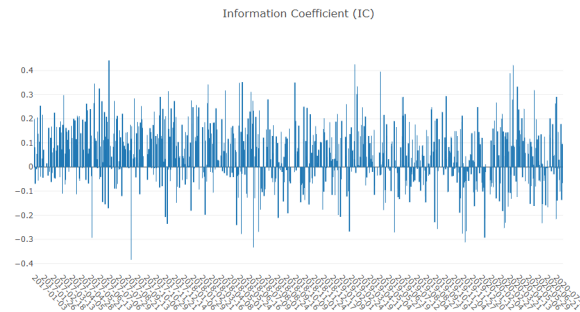
- **long-short/long-average** The distribution of long-short/long-average returns on each trading day



**Note:**

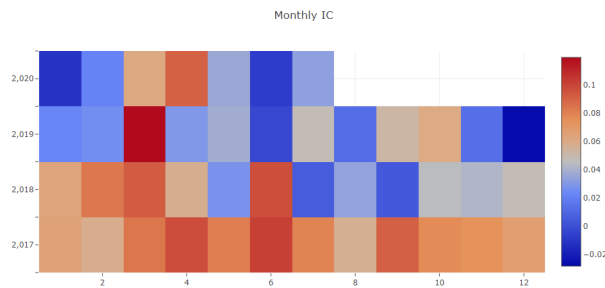
- **Information Coefficient**

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



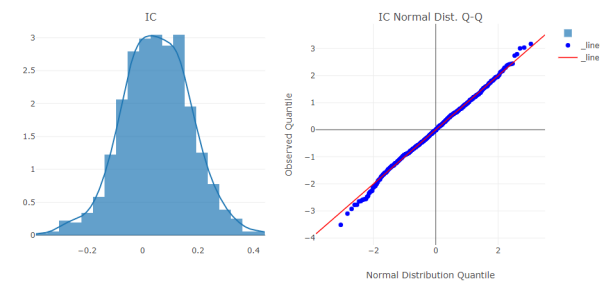
**Note:**

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



**Note:**

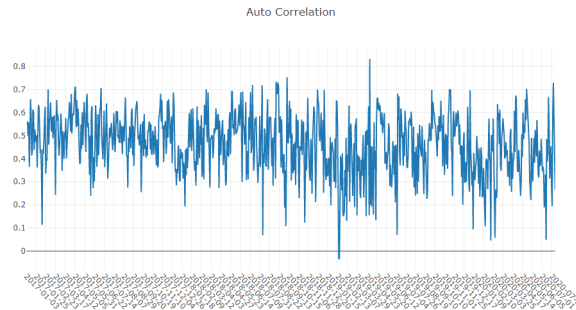
- **IC** The distribution of the *Information Coefficient* on each trading day.
- **IC Normal Dist. Q-Q** The *Quantile-Quantile Plot* is used for the normal distribution of *Information Coefficient* on each trading day.



**Note:**

- **Auto Correlation**

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



## 1.13 Building Formulaic Alphas

### 1.13.1 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.

### 1.13.2 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.contrib.estimator.handler import QLibDataHandler
>> MACD_EXP = '(EMA($close, 12) - EMA($close, 26))/($close - EMA((EMA($close, 12) -
↳EMA($close, 26))/($close, 9)/$close)'
>> fields = [MACD_EXP] # MACD
>> names = ['MACD']
>> labels = ['$close'] # label
>> label_names = ['LABEL']
>> data_handler = QLibDataHandler(start_date='2010-01-01', end_date='2017-12-31',
↳fields=fields, names=names, labels=labels, label_names=label_names)
>> TRAINER_CONFIG = {
..     "train_start_date": "2007-01-01",
..     "train_end_date": "2014-12-31",
..     "validate_start_date": "2015-01-01",
..     "validate_end_date": "2016-12-31",
..     "test_start_date": "2017-01-01",
..     "test_end_date": "2020-08-01",
.. }
>> feature_train, label_train, feature_validate, label_validate, feature_test, label_
↳test = data_handler.get_split_data(**TRAINER_CONFIG)
>> print(feature_train, label_train)
```

		MACD
instrument	datetime	
SH600000	2010-01-04	-0.008625
	2010-01-05	-0.007234
	2010-01-06	-0.007693
	2010-01-07	-0.009633
	2010-01-08	-0.009891
...		...
SZ300251	2014-12-25	0.043072
	2014-12-26	0.041345
	2014-12-29	0.042733
	2014-12-30	0.042066
	2014-12-31	0.036299

[322025 rows x 1 columns]

		LABEL
instrument	datetime	
SH600000	2010-01-04	4.260015
	2010-01-05	4.292182
	2010-01-06	4.207747
	2010-01-07	4.113258
	2010-01-08	4.159496
...		...
SZ300251	2014-12-25	4.343212
	2014-12-26	4.470587
	2014-12-29	4.762474
	2014-12-30	4.369748
	2014-12-31	4.182222

[322025 rows x 1 columns]

### 1.13.3 Reference

To learn more about `Data Handler`, please refer to [Data Handler](#)

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

## 1.14 Online & Offline mode

### 1.14.1 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.

### 1.14.2 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.

### 1.14.3 Reference

If users are interested in Qlib-Server and Online mode, please refer to [Qlib-Server Project](#) and [Qlib-Server Document](#).

## 1.15 API Reference

Here you can find all Qlib interfaces.

### 1.15.1 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 : { 'market': 'csi500',

    'filter\_pipe': [{ 'filter\_type': 'ExpressionDFilter',

        'rule\_expression': '\$open<40', 'filter\_start\_time': None, 'filter\_end\_time': None, 'keep': False},

    { 'filter\_type': 'NameDFilter', 'name\_rule\_re': 'SH[0-9]{4}55', 'filter\_start\_time': None, 'filter\_end\_time': None} ] }

**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.

**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.

**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.

**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.

**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.

**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.

**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.

**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.

**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.

**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.

```
class qlib.data.data.Wrapper
```

Data Provider Wrapper

```
qlib.data.data.register_wrapper(wrapper, cls_or_obj)
```

**Parameters**

- **wrapper** – A wrapper of all kinds of providers
- **cls\_or\_obj** – A class or class name or object instance in data/data.py

```
qlib.data.data.register_all_wrappers()
```

## 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

```
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

```
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.

**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'`

**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

## Feature

### 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 + right_etd]`.

**Returns** lft\_etd, right\_etd

**Return type** (int, int)

**class** qlib.data.base.**Feature**(*name=None*)

Static Expression

This kind of feature will load data from provider

**get\_longest\_back\_rolling**()

Get the longest length of historical data the feature has accessed

This is designed for getting the needed range of the data to calculate the features in specific range at first. However, situations like `Ref(Ref($close, -1), 1)` can not be handled rightly.

So this will only used for detecting the length of historical data needed.

**get\_extended\_window\_size**()

get\_extend\_window\_size

For to calculate this Operator in `range[start_index, end_index]` We have to get the *leaf feature* in `range[start_index - lft_etd, end_index + right_etd]`.

**Returns** lft\_etd, right\_etd

**Return type** (int, int)

**class** qlib.data.base.**ExpressionOps**

Operator Expression

This kind of feature will use operator for feature construction on the fly.

## Operator

**class** qlib.data.ops.**Abs**(*feature*)

Feature Absolute Value

**Parameters** **feature** (*Expression*) – feature instance

**Returns** a feature instance with absolute output

**Return type** *Expression*

**class** qlib.data.ops.**Sign**(*feature*)

Feature Sign

**Parameters** **feature** (*Expression*) – feature instance

**Returns** a feature instance with sign

**Return type** *Expression*

**class** `qlib.data.ops.Log` (*feature*)

Feature Log

**Parameters** **feature** (*Expression*) – feature instance

**Returns** a feature instance with log

**Return type** *Expression*

**class** `qlib.data.ops.Power` (*feature, exponent*)

Feature Power

**Parameters** **feature** (*Expression*) – feature instance

**Returns** a feature instance with power

**Return type** *Expression*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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

**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*

**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`

**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`

**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`

**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`

**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`

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**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*

**class** `qlib.data.ops.EMA` (*feature*, *N*)

Rolling Exponential Mean (EMA)

**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*

**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*

**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*

## Cache

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

**class** `qlib.data.cache.MemCache` (*mem\_cache\_size\_limit=None, limit\_type='length'*)  
Memory cache.

**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.

Override 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.

Override 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.

**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.

Override 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.

**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.

**gen\_dataset\_cache** (*cache\_path, instruments, fields, freq*)

---

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

---

Acquire the lock outside this function

The format the cache contains 3 parts(followed by typical filename).

- **index** [cache/d41366901e25de3ec47297f12e2ba11d.index]
  - The content of the file may be in following format(pandas.Series)

	start	end
1999-11-10 00:00:00	0	1
1999-11-11 00:00:00	1	2
1999-11-12 00:00:00	2	3
...		

---

**Note:** The start is closed. The end is open!!!!

---

- Each line contains two element <timestamp, end\_index>
  - It indicates the *end\_index* 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.

Override 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

## 1.15.2 Contrib

### Data Handler

```
class qlib.contrib.estimator.handler.BaseDataHandler (processors=[], **kwargs)
```

```
split_rolling_periods (train_start_date, train_end_date, validate_start_date, validate_end_date, test_start_date, test_end_date, rolling_period, calendar_freq='day')
```

Calculating the Rolling split periods, the period rolling on market calendar. :param train\_start\_date: :param train\_end\_date: :param validate\_start\_date: :param validate\_end\_date: :param test\_start\_date: :param test\_end\_date: :param rolling\_period: The market period of rolling :param calendar\_freq: The frequency of the market calendar :yield: Rolling split periods

```
get_split_data (train_start_date, train_end_date, validate_start_date, validate_end_date, test_start_date, test_end_date)
```

all return types are DataFrame

```
setup_process_data (df_train, df_valid, df_test)
```

process the train, valid and test data :return: the processed train, valid and test data.

```
get_origin_test_label_with_date (test_start_date, test_end_date, freq='day')
```

Get origin test label

#### Parameters

- **test\_start\_date** – test start date
- **test\_end\_date** – test end date
- **freq** – freq

**Returns** pd.DataFrame

```
setup_feature ()
```

**Implement this method to load raw feature.** the format of the feature is below

return: df\_features

```
setup_label ()
```

**Implement this method to load and calculate label.** the format of the label is below

return: df\_label

```
class qlib.contrib.estimator.handler.QLibDataHandler (start_date, end_date, *args, **kwargs)
```

```
setup_feature ()
```

Load the raw data. return: df\_features

```
setup_label ()
```

Build up labels in df through users' method :return: df\_labels

```
qlib.contrib.estimator.handler.parse_config_to_fields (config)
```

create factors from config

```
config = { 'kbar': {}, # whether to use some hard-code kbar features 'price': { # whether to use raw price features
```

```
    'windows': [0, 1, 2, 3, 4], # use price at n days ago 'feature': ['OPEN', 'HIGH', 'LOW'] # which price field to use
```

```
    }, 'volume': { # whether to use raw volume features
        'windows': [0, 1, 2, 3, 4], # use volume at n days ago
    }, 'rolling': { # whether to use rolling operator based features
        'windows': [5, 10, 20, 30, 60], # rolling windows size
        'include': ['ROC', 'MA', 'STD'], # rolling operator to use
        #if include is None we will use default operators
        'exclude': ['RANK'], # rolling operator not to use
    }
}

class qlib.contrib.estimator.handler.ConfigQLibDataHandler(start_date, end_date,
                                                           processors=None,
                                                           **kwargs)

class qlib.contrib.estimator.handler.ALPHA360(start_date, end_date, processors=None,
                                               **kwargs)

class qlib.contrib.estimator.handler.QLibDataHandlerV1(start_date, end_date, processors=None,
                                                         **kwargs)

def setup_label():
    load the labels df :return: df_labels

class qlib.contrib.estimator.handler.Alpha158(start_date, end_date, processors=None,
                                              **kwargs)
```

## Model

```
class qlib.contrib.model.base.Model
    Model base class

    fit(x_train, y_train, x_valid, y_valid, w_train=None, w_valid=None, **kwargs)
        fit train with cross-validation Fit model when ex_config.finetune is False
```

### Parameters

- **x\_train** (*pd.dataframe*) – train data
- **y\_train** (*pd.dataframe*) – train label
- **x\_valid** (*pd.dataframe*) – valid data
- **y\_valid** (*pd.dataframe*) – valid label
- **w\_train** (*pd.dataframe*) – train weight
- **w\_valid** (*pd.dataframe*) – valid weight

**Returns** trained model

**Return type** *Model*

```
score(x_test, y_test, w_test=None, **kwargs)
    evaluate model with test data/label
```

### Parameters

- **x\_test** (*pd.dataframe*) – test data
- **y\_test** (*pd.dataframe*) – test label
- **w\_test** (*pd.dataframe*) – test weight



**Returns** evaluation score

**Return type** float

**predict** (*x\_test*, *\*\*kwargs*)  
predict given test data

**Parameters** **x\_test** (*pd.dataframe*) – test data

**Returns** test predict label

**Return type** np.ndarray

**save** (*fname*, *\*\*kwargs*)  
save model

**Parameters** **fname** (*str*) – model filename

**load** (*buffer*, *\*\*kwargs*)  
load model

**Parameters** **buffer** (*bytes*) – binary data of model parameters

**Returns** loaded model

**Return type** *Model*

**get\_data\_with\_date** (*date*, *\*\*kwargs*)  
Will be called in online module need to return the data that used to predict the label (score) of stocks at date.

**:param**

**date:** *pd.Timestamp* predict date

**Returns** data: the input data that used to predict the label (score) of stocks at predict date.

**finetune** (*x\_train*, *y\_train*, *x\_valid*, *y\_valid*, *w\_train=None*, *w\_valid=None*, *\*\*kwargs*)  
Finetune model In *RollingTrainer*:

**if loader.model\_index is None:** If provide ‘Static Model’, based on the provided ‘Static’ model update. If provide ‘Rolling Model’, skip the model of load, based on the last ‘provided model’ update.

**if loader.model\_index is not None:** Based on the provided model(loader.model\_index) update.

**In StaticTrainer:**

**If the load is ‘static model’:** Based on the ‘static model’ update

**If the load is ‘rolling model’:** Based on the provided model(loader.model\_index) update. If loader.model\_index is None, use the last model.

**Parameters**

- **x\_train** (*pd.dataframe*) – train data
- **y\_train** (*pd.dataframe*) – train label
- **x\_valid** (*pd.dataframe*) – valid data
- **y\_valid** (*pd.dataframe*) – valid label
- **w\_train** (*pd.dataframe*) – train weight

- **w\_valid** (*pd.dataframe*) – valid weight

**Returns** finetune model

**Return type** *Model*

## 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.

**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 internal state of a strategyw - it can be regard 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*)

**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*)

**generate\_target\_weight\_position** (*score, current, trade\_date*)

Parameter: *score* : pred score for this trade date, *pd.Series*, index is *stock\_id*, contain 'score' column  
*current* : current position, use `Position()` class  
*trade\_exchange* : `Exchange()`  
*trade\_date* : trade date  
generate target position from score for this date and the current position  
The cash is not considered in the position

**generate\_order\_list** (*score\_series, current, trade\_exchange, pred\_date, trade\_date*)

Parameter *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='bottom', risk\_degree=0.95, thresh=1, hold\_thresh=1, \*\*kwargs*)

**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.

#### Parameter

**score\_series** [pd.Seires] 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.get_strategy(strategy=None, topk=50, margin=0.5, n_drop=5, risk_degree=0.95, str_type='amount', ad-just_dates=None)`

#### Parameters

- **strategy** (*Strategy()*) – strategy used in backtest
- **topk** (*int (Default value: 50)*) – top-N stocks to buy.
- **margin** (*int or float (Default value: 0.5)*) –  
 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
- **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

#### Returns

- *class: Strategy*
- *an initialized strategy object*

`qlib.contrib.evaluate.get_exchange(pred, exchange=None, subscribe_fields=[], open_cost=0.0015, close_cost=0.0025, min_cost=5.0, trade_unit=None, limit_threshold=None, deal_price=None, extract_codes=False, shift=1)`

#### Parameters

- **exchange related arguments** (#) –
- **exchange** (*Exchange()*) –
- **subscribe\_fields** (*list*) – subscribe fields

- **open\_cost** (*float*) – open transaction cost
- **close\_cost** (*float*) – close transaction cost
- **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.

### Returns

- *class: Exchange*
- *an initialized Exchange object*

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

This function will help you set a reasonable Exchange and provide default value for strategy :param # backtest workflow related or common arguments: :param pred: predict should has <instrument, datetime> 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.

---

```
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
```

(continues on next page)

(continued from previous page)

```

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**

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

- **show\_notebook** – whether to display graphics in notebook, the default is **True**

**Returns** if **show\_notebook** is **True**, display in notebook; else return **plotly.graph\_objs.Figure** list

**qlib.contrib.report.analysis\_position.score\_ic.score\_ic\_graph** (*pred\_label*: *pandas.core.frame.DataFrame*,  
*show\_notebook*:  
*bool = True*) →  
[<class 'list'>,  
<class 'tuple'>]

score IC

Example:

```

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

```

### Parameters

- **pred\_label** – index is **pd.MultiIndex**, index name is [**instrument**, **datetime**];

columns names is `[score, label]`

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

- **show\_notebook** – whether to display graphics in notebook, the default is **True**

**Returns** if show\_notebook is True, display in notebook; else return **plotly.graph\_objs.Figure** list

`qlib.contrib.report.analysis_position.cumulative_return.cumulative_return_graph` (*position:* dict, *re-report\_normal:* pan-das.core.frame, *label\_data:* pan-das.core.frame, *show\_notebook:* start\_date=None, end\_date=None) → It-er-able[plotly.graph

Backtest buy, sell, and holding cumulative return graph

Example:

```
from qlib.data import D
from qlib.contrib.evaluate import risk_analysis, backtest, ↵
↵long_short_backtest
from qlib.contrib.strategy import TopkDropoutStrategy

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

sparas = {}
sparas['topk'] = 50
sparas['n_drop'] = 5
strategy = TopkDropoutStrategy(**sparas)

report_normal_df, positions = backtest(pred_df, strategy, ↵
↵**bparas)

pred_df_dates = pred_df.index.get_level_values(level='datetime
↵')
features_df = D.features(D.instruments('csi500'), ['Ref($close,
↵ -1)/$close - 1'], pred_df_dates.min(), pred_df_dates.max())
features_df.columns = ['label']
```

(continues on next page)

(continued from previous page)

```
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'])`

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

**Parameters**

- **show\_notebook** – True or False. If True, show graph in notebook, else return figures
- **start\_date** – start date
- **end\_date** – end date



**Returns**

```

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

```

Generate analysis graph and monthly analysis

Example:

```

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'])

```

(continues on next page)

(continued from previous page)

```

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**

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

- **report\_long\_short\_df** – **df.index.name** must be **date**, **df.columns** contain **long**, **short**, **long\_short**

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

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

## Returns

`qlib.contrib.report.analysis_position.rank_label.rank_label_graph` (*position:*  
*dict, label\_data:*  
*panel\_data:*  
*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'])`

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

## Parameters

- **start\_date** – start date
- **end\_date** – end\_date
- **show\_notebook** – **True** or **False**. If True, show graph in notebook, else return figures

#### Returns

`qlib.contrib.report.analysis_model.analysis_model_performance.ic_figure(ic_df:`

`pan-`  
`das.core.frame.DataFrame,`  
`show_nature_day=True,`  
`**kwargs)`  
`→`  
`plotly.graph_objs._figure.Fi`

IC figure

#### Parameters

- **ic\_df** – ic DataFrame
- **show\_nature\_day** – whether to display the abscissa of non-trading day

**Returns** `plotly.graph_objs.Figure`

`qlib.contrib.report.analysis_model.analysis_model_performance.model_performance_graph(pred_label:`

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

Model performance

**Parameters** **pred\_label** – index is `pd.MultiIndex`, index name is `[instrument, datetime]`;  
columns names is `**[score,`

label]\*\*. It is usually same as the label of model training(e.g. “Ref(\$close, -2)/Ref(\$close, -1) - 1”)

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

### Parameters

- **lag** – `pred.groupby(level='instrument')['score'].shift(lag)`. It will be only used in the auto-correlation computing.
- **N** – group number, default 5
- **reverse** – if `True`, `pred['score'] *= -1`
- **rank** – if `True`, calculate rank ic
- **graph\_names** – graph names; default ['cumulative\_return', 'pred\_ic', 'pred\_autocorr', 'pred\_turnover']
- **show\_notebook** – whether to display graphics in notebook, the default is `True`
- **show\_nature\_day** – whether to display the abscissa of non-trading day

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

## 1.16 Qlib FAQ

### 1.16.1 Qlib Frequently Asked Questions

- 1. *RuntimeError: An attempt has been made to start a new process before the current process has finished its bootstrapping phase...*
- 2. *qlib.data.cache.QlibCacheException: It sees the key(...) of the redis lock has existed in your redis db now.*

#### 1. RuntimeError: An attempt has been made to start a new process before the current process has finished its bootstrapping phase...

```
RuntimeError:
    An attempt has been made to start a new process before the
    current process has finished its bootstrapping phase.

    This probably means that you are not using fork to start your
    child processes and you have forgotten to use the proper idiom
    in the main module:

        if __name__ == '__main__':
            freeze_support()
            ...
```

(continues on next page)

(continued from previous page)

```
The "freeze_support()" line can be omitted if the program
is not going to be frozen to produce an executable.
```

This is caused by the limitation of multiprocessing under windows OS. Please refer to [here](#) for more info.

**Solution:** To select a start method you use the `D.features` in the `if __name__ == '__main__':` clause of the main module. For example:

```
import qlib
from qlib.data import D

if __name__ == "__main__":
    qlib.init()
    instruments = ["SH600000"]
    fields = ["$close", "$change"]
    df = D.features(instruments, fields, start_time='2010-01-01', end_time='2012-12-31')
    print(df.head())
```

## 2. `qlib.data.cache.QlibCacheException`: It sees the `key(...)` of the redis lock has existed in your redis db now.

It sees the key of the redis lock has existed in your redis db now. You can use the following command to clear your redis keys and rerun your commands

```
$ redis-cli
> select 1
> flushdb
```

---

**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.

## 1.17 Changelog

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

### 1.17.1 Version 0.1.0

This is the initial release of QLib library.

### 1.17.2 Version 0.1.1

Performance optimize. Add more features and operators.

### 1.17.3 Version 0.1.2

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

### 1.17.4 Version 0.1.3

Bug fix and add instruments filtering mechanism.

### 1.17.5 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.

### 1.17.6 Version 0.2.1

- Support registering user-defined `Provider`.
- Support use operators in string format, e.g. `['Ref($close, 1)']` is valid field format.
- Support dynamic fields in `$some_field` format. And existing fields like `Close()` may be deprecated in the future.

### 1.17.7 Version 0.2.2

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

### 1.17.8 Version 0.2.3

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

### 1.17.9 Version 0.2.4

- Add `profit_attribution` module
- Add `risk_control` and `cost_control` strategies

### 1.17.10 Version 0.3.0

- Add `estimator` module

### 1.17.11 Version 0.3.1

- Add `filter` module

### 1.17.12 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

### 1.17.13 Version 0.3.3

- Fix bug of `filter` module

### 1.17.14 Version 0.3.4

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

### 1.17.15 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`

### 1.17.16 Version 0.4.0

- Add `data` package that holds all data-related codes
- Reform the data provider structure
- Create a server for data centralized management '[qlib-server<https://amc-msra.visualstudio.com/trading-algo/\\_git/qlib-server>](https://amc-msra.visualstudio.com/trading-algo/_git/qlib-server)'
- Add a `ClientProvider` to work with server
- Add a pluggable cache mechanism
- Add a recursive backtracking algorithm to inspect the furthest reference date for an expression



**Note:** The `D.instruments` function does not support `start_time`, `end_time`, and `as_list` parameters, if you want to get the results of previous versions of `D.instruments`, you can do this:

```
>>> from qlib.data import D
>>> instruments = D.instruments(market='csi500')
>>> D.list_instruments(instruments=instruments, start_time='2015-01-01', end_time=
↳ '2016-02-15', as_list=True)
```

### 1.17.17 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`

### 1.17.18 Version 0.4.2

- Refactor `DataHandler`
- Add `ALPHA360 DataHandler`

### 1.17.19 Version 0.4.3

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

### 1.17.20 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.

### 1.17.21 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.

### 1.17.22 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*.

### 1.17.23 Version 0.5.0

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

### 1.17.24 Version greater than Version 0.5.0

Please refer to [Github release Notes](#)

### q

- qlib.contrib.estimator.handler, [75](#)
- qlib.contrib.evaluate, [79](#)
- qlib.contrib.model.base, [76](#)
- qlib.contrib.report.analysis\_model.analysis\_model\_performance,  
    [88](#)
- qlib.contrib.report.analysis\_position.cumulative\_return,  
    [83](#)
- qlib.contrib.report.analysis\_position.rank\_label,  
    [86](#)
- qlib.contrib.report.analysis\_position.report,  
    [81](#)
- qlib.contrib.report.analysis\_position.risk\_analysis,  
    [85](#)
- qlib.contrib.report.analysis\_position.score\_ic,  
    [82](#)
- qlib.contrib.strategy.strategy, [78](#)
- qlib.data.base, [62](#)
- qlib.data.data, [54](#)
- qlib.data.filter, [61](#)
- qlib.data.ops, [63](#)



## A

Abs (class in *qlib.data.ops*), 63  
 Add (class in *qlib.data.ops*), 64  
 AdjustTimer (class in *qlib.contrib.strategy.strategy*), 78  
 Alpha158 (class in *qlib.contrib.estimator.handler*), 76  
 ALPHA360 (class in *qlib.contrib.estimator.handler*), 76  
 And (class in *qlib.data.ops*), 66

## B

backtest() (in module *qlib.contrib.evaluate*), 80  
 BaseDataHandler (class in *qlib.contrib.estimator.handler*), 75  
 BaseDFilter (class in *qlib.data.filter*), 61  
 BaseProvider (class in *qlib.data.data*), 60

## C

cache\_to\_origin\_data() (*qlib.data.cache.DatasetCache* static method), 31, 73  
 cache\_walker() (*qlib.data.data.LocalDatasetProvider* static method), 59  
 calendar() (*qlib.data.data.CalendarProvider* method), 54  
 calendar() (*qlib.data.data.ClientCalendarProvider* method), 59  
 calendar() (*qlib.data.data.LocalCalendarProvider* method), 57  
 CalendarProvider (class in *qlib.data.data*), 54  
 ClientCalendarProvider (class in *qlib.data.data*), 59  
 ClientDatasetProvider (class in *qlib.data.data*), 59  
 ClientInstrumentProvider (class in *qlib.data.data*), 59  
 ClientProvider (class in *qlib.data.data*), 60  
 ConfigQLibDataHandler (class in *qlib.contrib.estimator.handler*), 76  
 Corr (class in *qlib.data.ops*), 71

Count (class in *qlib.data.ops*), 70  
 Cov (class in *qlib.data.ops*), 71  
 cumulative\_return\_graph() (in module *qlib.contrib.report.analysis\_position.cumulative\_return*), 83

## D

dataset() (*qlib.data.cache.DatasetCache* method), 30, 72  
 dataset() (*qlib.data.data.ClientDatasetProvider* method), 59  
 dataset() (*qlib.data.data.DatasetProvider* method), 56  
 dataset() (*qlib.data.data.LocalDatasetProvider* method), 58  
 dataset\_processor() (*qlib.data.data.DatasetProvider* static method), 57  
 DatasetCache (class in *qlib.data.cache*), 30, 72  
 DatasetProvider (class in *qlib.data.data*), 56  
 Delta (class in *qlib.data.ops*), 70  
 DiskDatasetCache (class in *qlib.data.cache*), 73  
 DiskDatasetCache.IndexManager (class in *qlib.data.cache*), 73  
 DiskExpressionCache (class in *qlib.data.cache*), 73  
 Div (class in *qlib.data.ops*), 65

## E

EMA (class in *qlib.data.ops*), 71  
 Eq (class in *qlib.data.ops*), 66  
 Expression (class in *qlib.data.base*), 62  
 expression() (*qlib.data.cache.ExpressionCache* method), 29, 72  
 expression() (*qlib.data.data.ExpressionProvider* method), 56  
 expression() (*qlib.data.data.LocalExpressionProvider* method), 58  
 expression\_calculator() (*qlib.data.data.DatasetProvider* static method), 57

ExpressionCache (class in *qlib.data.cache*), 29, 72  
 ExpressionDFilter (class in *qlib.data.filter*), 62  
 ExpressionOps (class in *qlib.data.base*), 63  
 ExpressionProvider (class in *qlib.data.data*), 56

## F

Feature (class in *qlib.data.base*), 63  
 feature() (*qlib.data.data.FeatureProvider* method), 56  
 feature() (*qlib.data.data.LocalFeatureProvider* method), 58  
 FeatureProvider (class in *qlib.data.data*), 56  
 features() (*qlib.data.data.BaseProvider* method), 60  
 features\_uri() (*qlib.data.data.LocalProvider* method), 60  
 filter\_main() (*qlib.data.filter.SeriesDFilter* method), 61  
 finetune() (*qlib.contrib.model.base.Model* method), 77  
 fit() (*qlib.contrib.model.base.Model* method), 76  
 from\_config() (*qlib.data.filter.BaseDFilter* static method), 61  
 from\_config() (*qlib.data.filter.ExpressionDFilter* method), 62  
 from\_config() (*qlib.data.filter.NameDFilter* static method), 62

## G

Ge (class in *qlib.data.ops*), 65  
 gen\_dataset\_cache() (*qlib.data.cache.DiskDatasetCache* method), 74  
 gen\_expression\_cache() (*qlib.data.cache.DiskExpressionCache* method), 73  
 generate\_order\_list() (*qlib.contrib.strategy.strategy.TopkDropoutStrategy* method), 78  
 generate\_order\_list() (*qlib.contrib.strategy.strategy.WeightStrategyBase* method), 78  
 generate\_target\_weight\_position() (*qlib.contrib.strategy.strategy.WeightStrategyBase* method), 78  
 get\_column\_names() (*qlib.data.data.DatasetProvider* static method), 57  
 get\_data\_with\_date() (*qlib.contrib.model.base.Model* method), 77  
 get\_exchange() (in module *qlib.contrib.evaluate*), 79  
 get\_extended\_window\_size() (*qlib.data.base.Expression* method), 63

get\_extended\_window\_size() (*qlib.data.base.Feature* method), 63  
 get\_extended\_window\_size() (*qlib.data.ops.If* method), 67  
 get\_extended\_window\_size() (*qlib.data.ops.Ref* method), 67  
 get\_instruments\_d() (*qlib.data.data.DatasetProvider* static method), 57  
 get\_longest\_back\_rolling() (*qlib.data.base.Expression* method), 63  
 get\_longest\_back\_rolling() (*qlib.data.base.Feature* method), 63  
 get\_longest\_back\_rolling() (*qlib.data.ops.If* method), 67  
 get\_longest\_back\_rolling() (*qlib.data.ops.Ref* method), 67  
 get\_origin\_test\_label\_with\_date() (*qlib.contrib.estimator.handler.BaseDataHandler* method), 75  
 get\_risk\_degree() (*qlib.contrib.strategy.strategy.TopkDropoutStrategy* method), 78  
 get\_split\_data() (*qlib.contrib.estimator.handler.BaseDataHandler* method), 75  
 get\_strategy() (in module *qlib.contrib.evaluate*), 79  
 Greater (class in *qlib.data.ops*), 65  
 Gt (class in *qlib.data.ops*), 65

## I

ic\_figure() (in module *qlib.contrib.report.analysis\_model.analysis\_model\_performance*) 48, 88  
 IdxMax (class in *qlib.data.ops*), 69  
 IdxMin (class in *qlib.data.ops*), 69  
 If (class in *qlib.data.ops*), 67  
 InstrumentProvider (class in *qlib.data.data*), 55  
 instruments() (*qlib.data.data.InstrumentProvider* static method), 55  
 is\_adjust() (*qlib.contrib.strategy.strategy.AdjustTimer* method), 78  
 is\_adjust() (*qlib.contrib.strategy.strategy.ListAdjustTimer* method), 78

## K

Kurt (class in *qlib.data.ops*), 68

## L

Le (class in *qlib.data.ops*), 66  
 Less (class in *qlib.data.ops*), 65  
 list\_instruments() (*qlib.data.data.ClientInstrumentProvider* method), 59

list\_instruments() (qlib.data.data.InstrumentProvider method), 55

list\_instruments() (qlib.data.data.LocalInstrumentProvider method), 57

ListAdjustTimer (class in qlib.contrib.strategy.strategy), 78

load() (qlib.contrib.model.base.Model method), 77

load() (qlib.data.base.Expression method), 62

LocalCalendarProvider (class in qlib.data.data), 57

LocalDatasetProvider (class in qlib.data.data), 58

LocalExpressionProvider (class in qlib.data.data), 58

LocalFeatureProvider (class in qlib.data.data), 58

LocalInstrumentProvider (class in qlib.data.data), 57

LocalProvider (class in qlib.data.data), 60

locate\_index() (qlib.data.data.CalendarProvider method), 55

Log (class in qlib.data.ops), 64

long\_short\_backtest() (in module qlib.contrib.evaluate), 81

Lt (class in qlib.data.ops), 66

## M

Mad (class in qlib.data.ops), 70

Mask (class in qlib.data.ops), 64

Max (class in qlib.data.ops), 69

Mean (class in qlib.data.ops), 67

Med (class in qlib.data.ops), 69

MemCache (class in qlib.data.cache), 29, 72

MemCacheUnit (class in qlib.data.cache), 29, 72

Min (class in qlib.data.ops), 69

Model (class in qlib.contrib.model.base), 76

model\_performance\_graph() (in module qlib.contrib.report.analysis\_model.analysis\_model\_performance), 48, 88

Mul (class in qlib.data.ops), 65

multi\_cache\_walker() (qlib.data.data.LocalDatasetProvider static method), 59

## N

NameDFilter (class in qlib.data.filter), 61

Ne (class in qlib.data.ops), 66

normalize\_uri\_args() (qlib.data.cache.DatasetCache static method), 31, 73

Not (class in qlib.data.ops), 64

## O

Or (class in qlib.data.ops), 66

## P

parse\_config\_to\_fields() (in module qlib.contrib.estimator.handler), 75

Power (class in qlib.data.ops), 64

predict() (qlib.contrib.model.base.Model method), 77

## Q

qlib.contrib.estimator.handler (module), 75

qlib.contrib.evaluate (module), 79

qlib.contrib.model.base (module), 76

qlib.contrib.report.analysis\_model.analysis\_model\_performance (module), 48, 88

qlib.contrib.report.analysis\_position.cumulative\_return (module), 83

qlib.contrib.report.analysis\_position.rank\_label (module), 86

qlib.contrib.report.analysis\_position.report (module), 40, 81

qlib.contrib.report.analysis\_position.risk\_analysis (module), 44, 85

qlib.contrib.report.analysis\_position.score\_ic (module), 42, 82

qlib.contrib.strategy.strategy (module), 78

qlib.data.base (module), 62

qlib.data.data (module), 54

qlib.data.filter (module), 61

qlib.data.ops (module), 63

QLibDataHandler (class in qlib.contrib.estimator.handler), 75

QLibDataHandlerV1 (class in qlib.contrib.estimator.handler), 76

Quantile (class in qlib.data.ops), 69

## R

Rank (class in qlib.data.ops), 70

rank\_label\_graph() (in module qlib.contrib.report.analysis\_position.rank\_label), 86

read\_data\_from\_cache() (qlib.data.cache.DiskDatasetCache class method), 73

Ref (class in qlib.data.ops), 67

register\_all\_wrappers() (in module qlib.data.data), 61

register\_wrapper() (in module qlib.data.data), 61

report\_graph() (in module qlib.contrib.report.analysis\_position.report), 40, 81

Resi (class in *qlib.data.ops*), 71

risk\_analysis() (in module *qlib.contrib.evaluate*), 79

risk\_analysis\_graph() (in module *qlib.contrib.report.analysis\_position.risk\_analysis*), 44, 85

Rsquare (class in *qlib.data.ops*), 70

## S

save() (*qlib.contrib.model.base.Model* method), 77

score() (*qlib.contrib.model.base.Model* method), 76

score\_ic\_graph() (in module *qlib.contrib.report.analysis\_position.score\_ic*), 42, 82

SeriesDFilter (class in *qlib.data.filter*), 61

setup\_feature() (*qlib.contrib.estimator.handler.BaseDataHandler* method), 75

setup\_feature() (*qlib.contrib.estimator.handler.QLibDataHandler* method), 75

setup\_label() (*qlib.contrib.estimator.handler.BaseDataHandler* method), 75

setup\_label() (*qlib.contrib.estimator.handler.QLibDataHandler* method), 75

setup\_label() (*qlib.contrib.estimator.handler.QLibDataHandlerVI* method), 76

setup\_process\_data() (*qlib.contrib.estimator.handler.BaseDataHandler* method), 75

Sign (class in *qlib.data.ops*), 63

Skew (class in *qlib.data.ops*), 68

Slope (class in *qlib.data.ops*), 70

split\_rolling\_periods() (*qlib.contrib.estimator.handler.BaseDataHandler* method), 75

Std (class in *qlib.data.ops*), 68

StrategyWrapper (class in *qlib.contrib.strategy.strategy*), 78

Sub (class in *qlib.data.ops*), 64

Sum (class in *qlib.data.ops*), 68

## T

to\_config() (*qlib.data.filter.BaseDFilter* method), 61

to\_config() (*qlib.data.filter.ExpressionDFilter* method), 62

to\_config() (*qlib.data.filter.NameDFilter* method), 62

TopkDropoutStrategy (class in *qlib.contrib.strategy.strategy*), 78

## U

update() (*qlib.data.cache.DatasetCache* method), 30, 73

update() (*qlib.data.cache.DiskDatasetCache* method), 74

update() (*qlib.data.cache.DiskExpressionCache* method), 73

update() (*qlib.data.cache.ExpressionCache* method), 30, 72

## V

Var (class in *qlib.data.ops*), 68

## W

WeightStrategyBase (class in *qlib.contrib.strategy.strategy*), 78

WMA (class in *qlib.data.ops*), 71

Wrapper (class in *qlib.data.data*), 61