Orca Documentation

Release 1.0.0

Li, Wang

November 17, 2014

Contents:

CHAPTER

ONE

INTRODUCTION

Orca is a Python library specifically designed for backtesting **statistical arbitrage** strategies(*"alpha"* henceforth) on China A-Shares market.

This library is integrated with a delibrately structured **MongoDB** as its data source. It depends heavily on the high-performance Python library Pandas, thus it is recommended to get familiar with its basic data structures(Series, DataFrame, Panel).

To experiment with Orca, it is advised that one creates a virtual Python environment with **virtualenv**:

```
$virtualenv /your/local/directory
```

and install the following dependency libraries properly:

- Pandas
- PyMongo
- PyTables
- TA-Lib

Orca has 8 main components:

- MongoDB interface
- Data cache
- Universes
- Alphas
- Performance
- Utilities
- Updaters
- Alpha DB interface

For backtesting purpose, one only have to focus on the first 6 components which are explained in detail in this documentation.

CHAPTER

TWO

TUTORIAL

2.1 Play with data

To fetch data from the underlying MongoDB, the first thing is to find out the fetcher class associated with this data. (These associations should be detailed in a Wiki page.) For example::

```
>>> from orca.mongo import QuoteFetcher
>>> fetcher = QuoteFetcher()
>>> fetcher.dnames
['open', 'high', 'low', 'close', 'prevclose', 'volume', 'amount', 'returns']
>>> fetcher.fetch('close', 20140101, 20140131)
```

Each fetcher class provides 4 methods tailored for slightly different situations:

fetch (*self*, *dname*, *startdate*, *enddate=None*, *backdays=0*)
: This is most used when one wants to fetch data between two dates(i.e. `startdate`, `enddate`), both included. When `enddate` is None, it defaults to the maximal date available in the MongoDB. `backdays` is useful in fetching data with enough history for simulation. For example, if the simulation starts at 20090101 and on each day, the alpha takes the last 20 days' closing price as input, then one can just set `startdate=20090101` and `backdays=20`, the fetched closing price will contain data earlier than the first simulation date.

fetch_window (*self*, *dname*, *window*)
: This is most used when one happens to have a consecutive list of trading dates(i.e. `window`). It is the actual workhorse behind the scene, of which the other 3 methods are wrappers.

fetch_history (*self*, *dname*, *date*, *backdays*)
: This is intended to be a convenient data fetching method used in production alphas. With `delay` properly set in the fetcher instance, this method returns data up to `date` with a length of `backdays`, i.e. history data with respect to `date`, hence the name.

fetch_daily (*self*, *dname*, *date*, *offset=0*)
: This is also intended to be a convenient data fetching method used in production alphas. `offset` is usually a positive integer, meaning the returned data is `offset`-days ago with respect to `date`.

A fetcher instance has other attributes that also affect returned data in these fetching methods:

delay
: This integer affects method `fetch_history()`. Usually it is a non-negative integer and measures the distance between the most recent history data date and `date` in the method; in particular, when `delay` is 1, the history data does not contain data from `date` (if `date` itself is a trading day). One is highly recommended to set this attribute **explicitly** in the intialization. Though it is always possible to override this attribute by supplying a keyword argument like::

  ```
  >>> quote.fetch_history('close', '20140111', 5, delay=2)
  ```

datetime_index, reindex
: These boolean attributes affects the returned data formats(i.e. index/columns for Series/DataFrame). When `datetime` is True, date index will be transformed into DatetimeIndex. When `reindex` is True, columns will be reindexed to use a common list of stocks ids. The former is useful in data resampling while the latter is useful to aligh different DataFrames.

2.2 Build cache

For a research topic, datas employed is usually a very tiny portion of those stored in MongoDB. To reduce database pressure and save time, one is advised to build a cache to save fetched datas. For example::

```
>>> from orca.data import CSVSaver
>>> saver = CSVSaver('./.cache')
>>> saver['close.csv'] = close
```

Later on, the saved data can be read from the cache::

```
>>> from orca.data import CSVLoader
>>> loader = CSVLoader('./cache', postfix='.csv')
>>> loaded_close = loader['close']
>>> from orca.utils.testing import frames_equal
>>> frames_equal(close, loaded_close)
False
>>> loader = CSVLoader('./cache', postfix='.csv')
>>> loader.configure(parse_date=True)
>>> loaded_close = loader['close']
>>> frames_equal(close, loaded_close)
True
```

For performance consideration, one is advised to use HDF5 format to save data if this does not bring much anxiety.

2.3 Write alphas

After datas are ready, either fetched from MongoDB or loaded from a temporary cache, one can now write alphas. A dummy example::

```
>>> from orca.alpha import BacktestingAlpha
>>> class MyAlpha(BacktestingAlpha):
...     def generate(self, date):
...         self.alphas[date] = close.ix[date]
>>> alpha = MyAlpha()
>>> alpha.run(startdate=20140101, enddate=20140131)
>>> alpha.get_alphas()
```

The method `get_alphas` will **always** return a DataFrame with DatetimeIndex and columns from the common stock id list. This dummy alpha is equivalent to::

```
>>> class MyAlpha2(BacktestingAlpha):
...     def generate(self, date):
...         self.alphas[date] = quote.fetch_daily('close', date)
>>> alpha2 = MyAlpha()
>>> alpha2.run(startdate=20140101, enddate=20140131)
>>> frames_equal(alpha.get_alphas(), alpha2.get_alphas())
True
```

A complete Python file for this dummy alpha::

```
from orca.mongo import QuoteFetcher
from orca.alpha import BacktestingAlpha

quote = QuoteFetcher()
close = quote.fetch(20140101, 20140130)

class MyAlpha:
    def generate(self, date):
        self.alphas[date] = close.ix[date]

if __name__ == '__main__':

    alpha = MyAlpha()
    alpha.run(20140101, 20140130)
    alpha.get_alphas().to_csv('alpha.csv')
```

Save this file as *dummy.py* and then run::

```
$ python dummy.py
```

The generated alpha is dumped in file *alpha.csv*.

2.4 Choose universes

Before jumping to performance analysis after the method `run` is called, it is important to associate it with a universe to make the analysis results more robust.

Example::

```
>>> from orca.universe.common import create_backtesting_topliquid_filter
>>> univ = create_backtesting_topliquid_filter(70, 70)
>>> univ = univ.filter(20140101, 20140130)
>>> from orca.universe.common import Liq70
>>> univ2 = Liq70.filter(20140101, 20140130)
```

Unless the attributes `datetime_index` or `reindex` overridden in method `filter` or set to False during initialization, the returned DataFrame is **always** properly formatted. Now use this universe to filter out alphas::

```
>>> from orca.operation.api import intersect
>>> alphadf = intersect(alpha.get_alphas(), univ)
>>> alphadf2 = intersect(alpha.get_alphas(), univ2)
```

2.5 Performance analysis

To analyse an alpha's performance(on certain universe), Orca introduces a two-step process.

Performance

> The alpha(more exactly, the DataFrame returned by method `get_alphas`) is passed into a Performance class. To further analyse this alpha from different perspectives, the Performance instance provides different methods to return special Analyser instances to calculate performance metrics.

Analyser

All performance metrics attached to a single alpha is actually performed in this class. It accepts a DataFrame and provides metric-calculation methods.

This is better illustrated with an example::

```
>>> from orca.performance import Performance
>>> perf = Performance(alphadf)
>>> longshort = Performance.get_longshort()
>>> long = Performance.get_qtop(0.3, index='HS300')
>>> qtail = Performance.get_qtail(0.3)
>>> quantiles = Performance.get_quantiles(10)
>>> longshort.get_ir()
[...]
>>> qtail.get_returns(cost=0.001)
[...]
>>> long.get_returns(cost=0.001, index=True)
[...]
>>> for i, q in enumerate(quantiles):
...     print i, q.get_returns()
[...]
```

If one forgets to filter out alpha before plugging it into Performance class, this class also provides a method to make up for this::

```
>>> perf = Performance(alpha)
>>> perf1 = perf.get_universe(univ)
>>> perf2 = perf.get_universe(univ2)
```

The **second** way is actually preferred. Another convenient method is that Performance can restrict an alpha on HS300, CS500 and other(the rest) to see its performance on the roughly so-called 'big', 'medium' and 'small' universes::

```
>>> big, mid, sml = perf.get_bms()
>>> big.get_original().get_ir()
```

2.6 Parallel

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

D