Prophet Documentation

Release 0.1.0

Michael Su

May 11, 2018

Contents

Prophet is a Python microframework for financial markets. Prophet strives to let the programmer focus on modeling financial strategies, portfolio management, and analyzing backtests. It achieves this by having few functions to learn to hit the ground running, yet being flexible enough to accomodate sophistication.

Go to *Quickstart* to get started or see the *Tutorial* for a more thorough introduction.

Prophet's data generators, order generators, and analyzers are all executed sequentially which is conducive to allowing individuals to build off of others work. If you have a package you would like to share, please open an issue on the github repository to register it in the *Registry*.

Join the mailing list here or join by email.

CHAPTER 1

Features

- Flexible market backtester
- Convenient order generator
- See *Roadmap* for upcoming features

CHAPTER 2

User Guide

Quickstart

To install Prophet, run:

```
pip install prophet
```

Here's a quick simulation with a OrderGenerator that uses avoids leverage and buys 100 shares of AAPL stock a day as long as it has enough cash.

```
import datetime as dt

from prophet import Prophet
from prophet.data import YahooCloseData
from prophet.analyze import default_analyzers
from prophet.orders import Orders


class OrderGenerator(object):

    def run(self, prices, timestamp, cash, **kwargs):
        # Lets buy lots of Apple!
        symbol = "AAPL"
        orders = Orders()
        if (prices.loc[timestamp, symbol] * 100) < cash:
            orders.add_order(symbol, 100)

        return orders


prophet = Prophet()
prophet.set_universe(["AAPL", "XOM"])
prophet.register_data_generators(YahooCloseData())
prophet.set_order_generator(OrderGenerator())
prophet.register_portfolio_analyzers(default_analyzers)

backtest = prophet.run_backtest(start=dt.datetime(2010, 1, 1))
analysis = prophet.analyze_backtest(backtest)
print analysis
# +--------------------------------------+
# | sharpe            |    1.09754359611 |
# | average_return    | 0.00105478425027 |
# | cumulative_return |         2.168833 |
# | volatility        |  0.0152560508189 |
# +--------------------------------------+

# Generate orders for you to execute today
# Using Nov, 10 2014 as the date because there might be no data for today's
# date (Market might not be open) and we don't want examples to fail.
today = dt.datetime(2014, 11, 10)
print prophet.generate_orders(today)
# Orders[Order(symbol='AAPL', shares=100)]
```

Tutorial

Introduction

In this tutorial we will:

1. **Implement Bollinger bands as an indicator using a 20 day look back. The upper band should represent the mean plus two**

    - Today's moving average breaks below the upper band.
    - Yesterday's moving average was above the lower band.
    - The market's moving average was 1.2 standard devations above the average.

Note: To learn more about what a bollinger band is, please see this article.

2. Create an event analyzer that will output a series of trades based on events. For simplicity, we will put a 1 for each timestamp and stock symbol pair where we want to execute a "buy" order.
3. Feed that data into the simulator and write an order generator that will create "buy" orders in blocks of 100 shares for each signal in the event study from step 2. The order generator will automatically sell the shares either 5 trading days later or on the last day of the simulation.
4. Print the performance of the strategy in terms of total return, average daily return, standard deviation of daily return, and Sharpe Ratio for the time period.

You can get the full source code of the tutorial here

The tutorial is based off of the last homework in QSTK. Since the portfolio is analyzed from the start date, the returned metrics will be different even if you use the same stock universe as the homework.

Data Generation

First you need to initialize the object and setup the stock universe:

```
prophet = Prophet()
prophet.set_universe(["AAPL", "XOM"])
```

Then you register any data generators.

```
# Registering data generators
prophet.register_data_generators(YahooCloseData(),
                                 BollingerData(),
                                 BollingerEventStudy())
```

Note: Please see the source code of `prophet.data` for an example of a data generator. Data generators don't have to just pull raw data though like `prophet.data.YahooCloseData` does. For instance, you can generate correlation data based off the price data. Prophet encourages you to logically separate out different steps in your analysis.

The `name` attribute of each of the generators is the key on the `data` object at which the generated data is stored. This data object is passed into each of the data generators. For example, since the `YahooCloseData` object has the name "prices", we can use the price data in the `BollingerData` that we execute right after.

```
import pandas as pd
from prophet.data import DataGenerator


class BollingerData(DataGenerator):
    name = "bollinger"

    def run(self, data, symbols, lookback, **kwargs):
        prices = data['prices'].copy()

        rolling_std = pd.rolling_std(prices, lookback)
        rolling_mean = pd.rolling_mean(prices, lookback)

        bollinger_values = (prices - rolling_mean) / (rolling_std)

        for s_key in symbols:
            prices[s_key] = prices[s_key].fillna(method='ffill')
            prices[s_key] = prices[s_key].fillna(method='bfill')
            prices[s_key] = prices[s_key].fillna(1.0)

        return bollinger_values
```

See how the `BollingerData.run()` method uses the price data to generate a rolling standard deviation and rolling mean. The fillna method is used here to fill in missing data. Realistically, only the `bfill()` method is uses in this example because the first 20 days won't have 20 prior days of price data to generate the rolling mean and standard deviation.

Note: `prices` is also passed into the run function of all `DataGenerator` objects for convenience but we want to emphasize that the `data` object is where most data from data generators is stored.

The line below normalizes the bollinger data relative to the the rolling standard devation. This gives us the number of standard devations as an integer value. This means a value of 2 would be the upper band and a value of -2 would be the lower band.

```
bollinger_values = (prices - rolling_mean) / (rolling_std)
```

At this point we need one more generator. We will call this one BollingerEventStudy. Essentially, all it will do is run through the bollinger data and see if our conditions to issue a buy order are met.

```
class BollingerEventStudy(DataGenerator):
    name = "events"

    def run(self, data, symbols, start, end, lookback, **kwargs):
        bollinger_data = data['bollinger']

        # Add an extra timestamp before close_data.index to be able
        # to retrieve the prior day's data for the first day
        start_index = bollinger_data.index.get_loc(start) - 1
        timestamps = bollinger_data.index[start_index:]

        # Find events that occur when the market is up more then 2%
        bollinger_spy = bollinger_data['SPX'] >= 1.2  # Series
        bollinger_today = bollinger_data.loc[timestamps[1:]] <= -2.0
        bollinger_yesterday = bollinger_data.loc[timestamps[:-1]] >= -2.0
        # When we look up a date in bollinger_yesterday,
        # we want the data from the day before our input
        bollinger_yesterday.index = bollinger_today.index
        events = (bollinger_today & bollinger_yesterday).mul(
            bollinger_spy, axis=0)

        return events.fillna(0)
```

Note: Notice how all the data generators use the *pandas* library as much as possible instead of python for loops. This is key to keeping your simulations fast. In general, try to keep as much code as possible running in C using libraries like *numpy* and *pandas*.

Order Generation

Now we need to create an order generator. One thing we need to do is keep track of sell orders which we want to execute 5 days after the "buy" order. To do that, when we call run the first time, we run the `setup()` method.

```
class OrderGenerator(object):

    def setup(self, events):
        sell_orders = pd.DataFrame(index=events.index, columns=events.columns)
        sell_orders = sell_orders.fillna(0)
        self.sell_orders = sell_orders

    def run(self, prices, timestamp, cash, data, **kwargs):
        """ Takes bollinger event data and generates orders """
        events = data['events']
        if not hasattr(self, 'sell_orders'):
            self.setup(events)
```

Note: The order generator API may change slightly in future version to allow for less hacky setup functions.

The rest of the `run()` function will find all buy signals from the event study, find all sell orders from the sell orders Dataframe, and create orders from both sources. When creating an buy order, it will also add a sell order to the `sell_orders` Dataframe.

```
# def run(...):
#     ...

    orders = Orders()
    # Find buy events for this timestamp
    timestamps = prices.index
    daily_data = events.loc[timestamp]
    order_series = daily_data[daily_data > 0]
    # Sell 5 market days after bought
    index = timestamps.get_loc(timestamp)
    if index + 5 >= len(timestamps):
        sell_datetime = timestamps[-1]
    else:
        sell_datetime = timestamps[index + 5]

    symbols = order_series.index
    self.sell_orders.loc[sell_datetime, symbols] -= 100
    daily_sell_data = self.sell_orders.loc[timestamp]
    daily_sell_orders = daily_sell_data[daily_sell_data != 0]

    # Buy and sell in increments of 100
    for symbol in daily_sell_orders.index:
        orders.add_order(symbol, -100)

    daily_event_data = events.loc[timestamp]
    daily_buy_orders = daily_event_data[daily_event_data != 0]

    # Buy and sell in increments of 100
    for symbol in daily_buy_orders.index:
        orders.add_order(symbol, 100)

    return orders
```

Now we register the order generator and execute the backtest.

```
prophet.set_order_generator(OrderGenerator())
backtest = prophet.run_backtest(start=dt.datetime(2008, 1, 1),
                                end=dt.datetime(2009, 12, 31), lookback=20)
```

Portfolio Analysis

The last step is to analyze the portfolio:

```
prophet.register_portfolio_analyzers(default_analyzers)
analysis = prophet.analyze_backtest(backtest)
print(analysis)
```

`default_analyzers` is a list of the four types of analysis we want. Much like the BollingerData generator, the Sharpe ratio analyzer uses the data returned by the volatility and average return analyzers to generate a Sharpe ratio.

Advanced

Slippage & Commissions

The `run_backtest()` method on the `Prophet` object contains a commission and slippage option for you to make the backtest more realistic. Slippage is how much of the price increases (when buying) or decreases (when selling) from the price data. Commission represents the fees you pay per trade.

Please open an issue if those parameters aren't sufficent for your needs. See the *API* for more details.

Best Practices

Try to keep as much of your code as possible in the pandas (or numpy) space. Lots of smart folk have spent a considerable amount of time optimizing those libraries. Since most of the code in pandas and numpy is executed in C, it will be much more performant.

For the data generators, please pass around pandas Dataframes as much as possible. (Which then means that your order generator will have to operate on pandas Dataframes)

Registry

There are currently no packages in the registry

If you have a package you would like to share, please open an issue on the github repository to register it in the *Registry*.

CHAPTER 3

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

API

This part of the documentation covers all the interfaces of Prophet. For parts where Flask depends on external libraries, we document the most important right here and provide links to the canonical documentation.

Prophet Object

class `prophet.`**`Prophet`**

The application object. Serves as the primary interface for using the Prophet library.

`config`

dict

Dictionary of settings to make available to other functions. Useful for things like `RISK_FREE_RATE`.

`analyze_backtest`(*backtest*)

Analyzes a backtest with the registered portfolio analyzers.

Parameters **backtest** (*prophet.backtest.BackTest*) – a backtest object

Returns prophet.backtest.BackTest

`generate_orders`(*target_datetime*, *lookback=0*, *cash=1000000*, *buffer_days=0*, *portfolio=Portfolio()*)

Generates orders for a given day. Useful for generating trade orders for a your personal account.

Parameters

- **target_datetime** (*datetime*) – The datetime you want to generate orders for.
- **lookback** (*int*) – Number of trading days you want data for before the (target_datetime - buffer_days)
- **cash** (*int*) – Amount of starting cash
- **buffer_days** (*int*) – number of trading days you want extra data generated for. Acts as a data start date.
- **portfolio** (*prophet.portfolio.Portfolio*) – Starting portfolio

register_data_generators(**functions*)
Registers functions that generate data to be assessed in the order generator.

Parameters **functions** (*list*) – List of functions.

register_portfolio_analyzers(*functions*)
Registers a list of functions that are sequentially executed to generate data. This list is appended to list of existing data generators.

Parameters **functions** (*list of function*) – Each function in the list of args is executed in sequential order.

run_backtest(*start*, *end=None*, *lookback=0*, *slippage=0.0*, *commission=0.0*, *cash=1000000*, *initial_portfolio=Portfolio()*)
Runs a backtest over a given time period.

Parameters

- **start** (*datetime*) – The start of the backtest window
- **end** (*datetime*) – The end of the backtest windows
- **lookback** (*int*) – Number of trading days you want data for before the start date
- **slippage** (*float*) – Percent price slippage when executing order
- **commission** (*float*) – Amount of commission paid per order
- **cash** (*int*) – Amount of starting cash
- **portfolio** (*prophet.portfolio.Portfolio*) – Starting portfolio

Returns prophet.backtest.BackTest

set_order_generator(*order_generator*)
Sets the order generator for backtests.

Parameters **order_generator** – Instance of class with a run method that generates

set_universe(*symbols*)
Sets the list of all tickers symbols that will be used.

Parameters **symbols** (*list of str*) – Ticker symbols to be used by Prophet.

Order Objects

class prophet.orders.**Order**
Order(symbol, shares)

class prophet.orders.**Orders**(**args*)
Orders object that an OrderGenerator should return.

add_order(*symbol*, *shares*)
Add an order to the orders list.

Parameters

- **symbol** (*str*) – Stock symbol to purchase
- **shares** (*int*) – Number of shares to purchase. Can be negative.

Backtest Object

`prophet.backtest.`**`BackTest`**

Portfolio Objects

class `prophet.portfolio.`**`Portfolio`**

Portfolio object where keys are stock symbols and values are share counts. You can pass these into a backtest to start with an initial basket of stocks.

Note: Subclasses dict in v0.1

Analyzer Objects

class `prophet.analyze.`**`Analyzer`**

class `prophet.analyze.`**`Volatility`**

class `prophet.analyze.`**`AverageReturn`**

class `prophet.analyze.`**`Sharpe`**

class `prophet.analyze.`**`CumulativeReturn`**

`prophet.analyze.`**`default_analyzers`** **= [volatility, average_return, sharpe, cumulative_return, maximum_drawdown,**

list() -> new empty list list(iterable) -> new list initialized from iterable's items

Data Objects

CHAPTER 4

Contributor Guide

Getting Started

Setup your dev environment with the following commands.

```
git clone git@github.com:Emsu/prophet.git
cd prophet
virtualenv env
. env/bin/activate
pip install -r dev-requirements.txt
python setup.py develop
```

If you want to help with longer term development, please open an issue here

CHAPTER 5

Additional Notes

Roadmap

v0.2

- Ability to handle more frequent timeseries data
- Stress Tester and scenario builder
- More data sources

Changelog

Version 0.1.1

- Added Python 3 support

Version 0.1

- First public release

License

Python Module Index