mlfinlab

Release 0.4.1

Hudson & Thames

Sep 04, 2019

GETTING STARTED

mlfinlab is an open source package based on the research of Dr Marcos Lopez de Prado in his new book Advances in Financial Machine Learning. This implementation started out as a spring board for a research project in the Masters in Financial Engineering programme at WorldQuant University and has grown into a mini research group called Hudson and Thames (not affiliated with the university).

CHAPTER

ONE

NOTES

mlfinlab is a living, breathing project, and new functionalities are consistently being added.

The implementations that will be added in the future as well as the implementations that are currently supported can be seen below:

- **Part 4: Useful Financial Features**
- Working on Chapter 19: Microstructural Features (Maksim)

- **Part 3: Backtesting**
- Done Chapter 16: ML Asset Allocation
- Done Chapter 10: Bet Sizing

- **Part 2: Modelling**
- Done Chapter 8: Feature Importance
- Done Chapter 7: Cross-Validation
- Done Chapter 6: Ensemble Methods
- Done Sequential Bootstrap Ensemble

- **Part 1: Data Analysis**
- Done Chapter 5: Fractionally Differentiated Features
- Done Chapter 4: Sample Weights
- Done Chapter 3: Labeling
- Done Chapter 2: Data Structures
- Purchased high quality raw tick data.
- Email us if you would like a sample of the standard bars.

CHAPTER

TWO

BUILT WITH

- Github - Development platform and repo
- Travis CI - Continuous integration, test and deploy

CHAPTER

THREE

GETTING STARTED

3.1 Installation

3.1.1 Recommended Versions

- Anaconda 3
- Python 3.6

3.1.2 Installation for Users

The package can be installed from the PyPi index via the console: Launch the terminal and run:

```
pip install mlfinlab
```

3.1.3 Installation for Developers

Clone the package repo to your local machine then follow the steps below.

Mac OS X and Ubuntu Linux

1. Make sure you install the latest version of the Anaconda 3 distribution. To do this you can follow the install and update instructions found on this link
2. Launch a terminal
3. Create a New Conda Environment. From terminal:

```
conda create -n <env name> python=3.6 anaconda
```

Accept all the requests to install.

4. Now activate the environment with:

```
source activate <env name>
```

5. From Terminal: go to the directory where you have saved the file, example:

```
cd Desktop/mlfinlab
```

6. Install Python requirements, by running the command:

```
pip install -r requirements.txt
```

Windows

1. Download and install the latest version of Anaconda 3
2. Launch Anaconda Navigator
3. Click Environments, choose an environment name, select Python 3.6, and click Create
4. Click Home, browse to your new environment, and click Install under Jupyter Notebook
5. Launch Anaconda Prompt and activate the environment:

```
conda activate <env name>
```

6. From Anaconda Prompt: go to the directory where you have saved the file, example:

```
cd Desktop/mlfinlab
```

7. Install Python requirements, by running the command:

```
pip install -r requirements.txt
```

3.2 Barriers to Entry

As most of you know, getting through the first 3 chapters of the book is challenging as it relies on HFT data to create the new financial data structures. Sourcing the HFT data is very difficult and thus we have resorted to purchasing the full history of S&P500 Emini futures tick data from TickData LLC.

We are not affiliated with TickData in any way but would like to recommend others to make use of their service. The full history cost us about $750 and is worth every penny. They have really done a great job at cleaning the data and providing it in a user friendly manner.

3.2.1 Sample Data

TickData does offer about 20 days worth of raw tick data which can be sourced from their website link.

For those of you interested in working with a two years of sample tick, volume, and dollar bars, it is provided for in the research repo.

You should be able to work on a few implementations of the code with this set.

3.2.2 Additional Sources

Searching for free tick data can be a challenging task. The following three sources may help:

1. Dukascopy. Offers free historical tick data for some futures, though you do have to register.
2. Most crypto exchanges offer tick data but not historical (see Binance API). So you'd have to run a script for a few days.
3. Blog Post: How and why I got 75Gb of free foreign exchange "Tick" data.

3.3 Requirements

- codecov==2.0.15
- coverage==4.5.2
- pandas==0.24.2
- pylint==2.3.0
- numpy==1.16.4
- xmlrunner==1.7.7
- numba==0.43.0
- scikit-learn==0.21
- *Installation*
- *Barriers to Entry*
- *Requirements*

CHAPTER

FOUR

IMPLEMENTATIONS

4.1 Data Structures

When analyzing financial data, unstructured datasets are commonly transformed into a structured format referred to as bars, where a bar represents a row in a table. mlfinlab implements tick, volume, and dollar bars using traditional standard bar methods as well as the less common information driven bars.

4.1.1 Data Preparation

Read in our data:

```
# Required Imports
import numpy as np
import pandas as pd


data = pd.read_csv('FILE_PATH')
```

Data Formatting

In order to utilize the bar sampling methods presented below, our data must first be formatted properly. Many data vendors will let you choose the format of your raw tick data files. We want to only focus on the following 3 columns: date_time, price, volume. The reason for this is to minimise the size of the csv files and the amount of time when reading in the files.

Our data is sourced from TickData LLC which provide TickWrite 7, to aid in the formatting of saved files. This allows us to save csv files in the format date_time, price, volume.

For this tutorial we will assume that you need to first do some preprocessing and then save your data to a csv file:

```
# Format the Data

# Don't convert to datetime here, it will take forever to convert
date_time = data['Date'] + ' ' + data['Time']
new_data = pd.concat([date_time, data['Price'], data['Volume']], axis=1)
new_data.columns = ['date', 'price', 'volume']
```

Initially, your instinct may be to pass mlfinlab package an in-memory DataFrame object but the truth is when you're running the function in production, your raw tick data csv files will be way too large to hold in memory. We used the subset 2011 to 2019 and it was more than 25 gigs. It is for this reason that the mlfinlab package requires a file path to read the raw data files from disk:

```
# Save to csv
new_data.to_csv('FILE_PATH', index=False)
```

4.1.2 Standard Bars

The three standard bar methods implemented share a similiar underlying idea in that we want to sample a bar after a certain threshold is reached.

1. For tick bars, we sample a bar after a certain number of ticks.
2. For volume bars, we sample a bar after a certain volume amount is traded.
3. For dollar bars, we sample a bar after a certain dollar amount is traded.

These bars are used throughout the text book (Advances in Financial Machine Learning, By Marcos Lopez de Prado, 2018, pg 25) to build the more interesting features for predicting financial time series data.

Tick Bars

get_tick_bars(*file_path*, *threshold=2800*, *batch_size=20000000*, *verbose=True*, *to_csv=False*, *output_path=None*)

Creates the tick bars: date_time, open, high, low, close.

Parameters

- **file_path** – File path pointing to csv data.
- **threshold** – A cumulative number of ticks above this threshold triggers a sample to be taken.
- **batch_size** – The number of rows per batch. Less RAM = smaller batch size.
- **verbose** – Print out batch numbers (True or False)
- **to_csv** – Save bars to csv after every batch run (True or False)
- **output_path** – Path to csv file, if to_csv is True

Returns Dataframe of tick bars

```
from mlfinlab.data_structures import standard_data_structures


# Tick Bars
tick = standard_data_structures.get_tick_bars('FILE_PATH', threshold=5500,
batch_size=1000000, verbose=False)
```

Volume Bars

get_volume_bars(*file_path*, *threshold=28224*, *batch_size=20000000*, *verbose=True*, *to_csv=False*, *output_path=None*)

Creates the volume bars: date_time, open, high, low, close.

Parameters

- **file_path** – File path pointing to csv data.
- **threshold** – A cumulative number of contracts traded above this threshold triggers a sample to be taken.

- **batch_size** – The number of rows per batch. Less RAM = smaller batch size.
- **verbose** – Print out batch numbers (True or False)
- **to_csv** – Save bars to csv after every batch run (True or False)
- **output_path** – Path to csv file, if to_csv is True

Returns Dataframe of volume bars

```
from mlfinlab.data_structures import standard_data_structures


# Volume Bars
volume = standard_data_structures.get_volume_bars('FILE_PATH', threshold=28000,
batch_size=1000000, verbose=False)
```

Dollar Bars

get_dollar_bars(*file_path*, *threshold=70000000*, *batch_size=20000000*, *verbose=True*, *to_csv=False*, *output_path=None*)

Creates the dollar bars: date_time, open, high, low, close.

Parameters

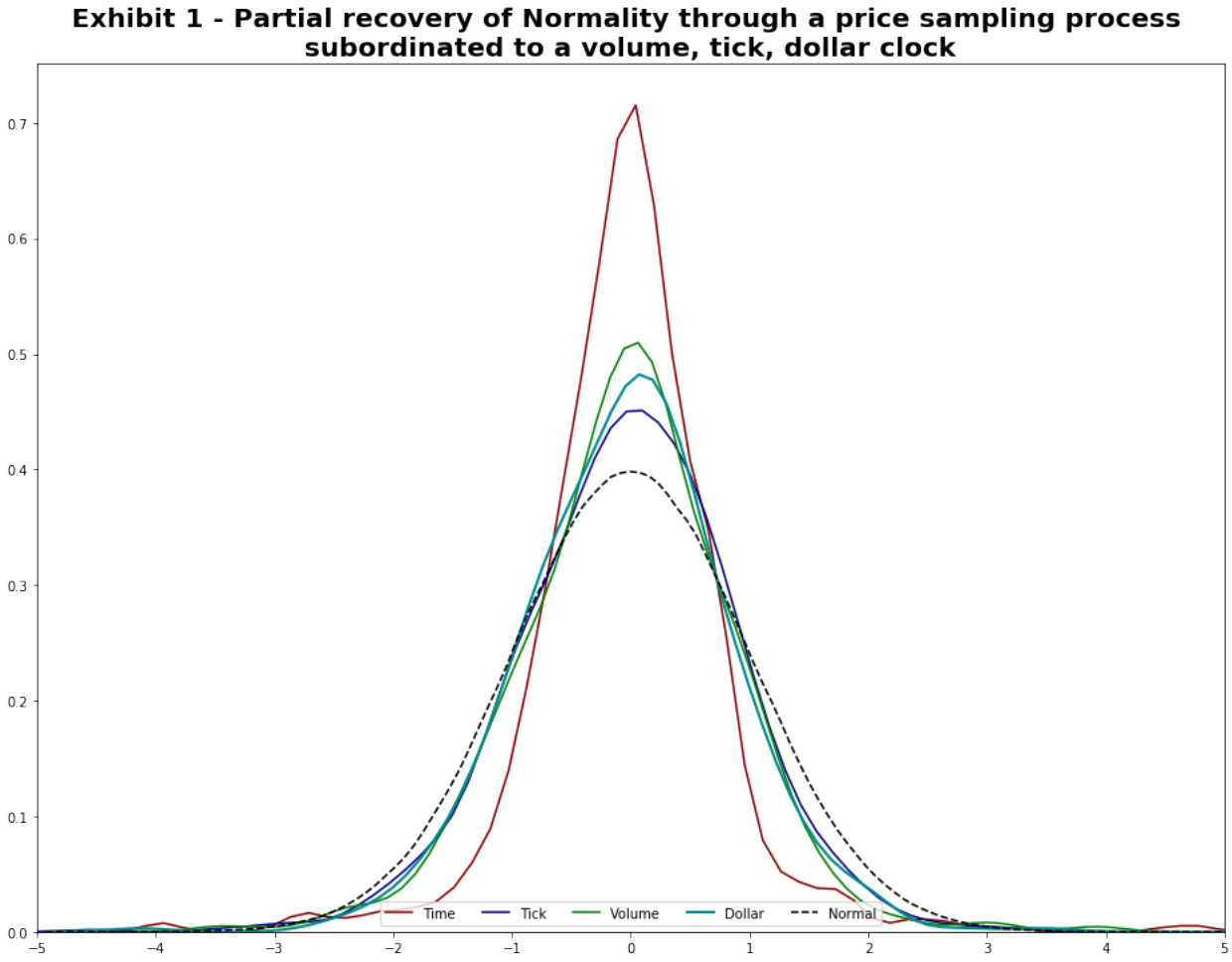
- **file_path** – File path pointing to csv data.
- **threshold** – A cumulative dollar value above this threshold triggers a sample to be taken.
- **batch_size** – The number of rows per batch. Less RAM = smaller batch size.
- **verbose** – Print out batch numbers (True or False)
- **to_csv** – Save bars to csv after every batch run (True or False)
- **output_path** – Path to csv file, if to_csv is True

Returns Dataframe of dollar bars

```
from mlfinlab.data_structures import standard_data_structures


# Dollar Bars
dollar = standard_data_structures.get_dollar_bars('FILE_PATH', threshold=70000000,
batch_size=1000000, verbose=True)
```

Statistical Properties

It can be seen below that tick, volume, and dollar bars all exhibit a distribution significantly closer to normal versus standard time bars:



4.1.3 Information-Driven Bars

Information-driven bars are based on the notion of sampling a bar when new information arrives to the market. The two types of information-driven bars implemented are imbalance bars and run bars. For each type, tick, volume, and dollar bars are included.

Imbalance Bars

Imbalance Bars Generation Algorithm

Let's discuss imbalance bars generation on example of volume imbalance bars. As it is described in Advances in Financial Machine Learning book:

First let's define what is the tick rule:

$$b_t = \begin{cases} b_{t-1}, & \Delta p_t{=}0 \\ |\Delta p_t|/\Delta p_t, & \Delta p_t \neq 0 \end{cases}$$

For any given t, where p_t is the price associated with t and v_t is volume, the tick rule b_t is defined as:

Tick rule is used as a proxy of trade direction, however, some data providers already provide customers with tick direction, in this case we don't need to calculate tick rule, just use the provided tick direction instead.

Cumulative volume imbalance from 1 to T is defined as:

$$\theta_t = \sum_{t=1}^{T} b_t * v_t$$

T is the time when the bar is sampled.

Next we need to define $E_0[T]$ as expected number of ticks, the book suggests to use EWMA of expected number of ticks from previously generated bars. Let's introduce the first hyperparameter for imbalance bars generation: **num_prev_bars** which corresponds to window used for EWMA calculation.

Here we face the problem of first bar generation, because we don't know expected number of ticks with no bars generated. To solve this we introduce the second hyperparameter: expected_num_ticks_init which corresponds to initial guess for **expected number of ticks** before the first imbalance bar is generated.

Bar is sampled when:

$$|\theta_t| \geq E_0[T] * [2v^+ - E_0[v_t]]$$

To estimate (expected imbalance) we simply calculate EWMA of volume imbalance from previous bars, that is why we need to store volume imbalances in imbalance array, the window for estimation is either **expected_num_ticks_init** before the first bar is sampled, or expected number of ticks($E_0[T]$) * **num_prev_bars** when the first bar is generated.

Note that when we have at least one imbalance bar generated we update $2v^+ - E_0[v_t]$ only when the next bar is sampled not on every trade observed

Algorithm Logic

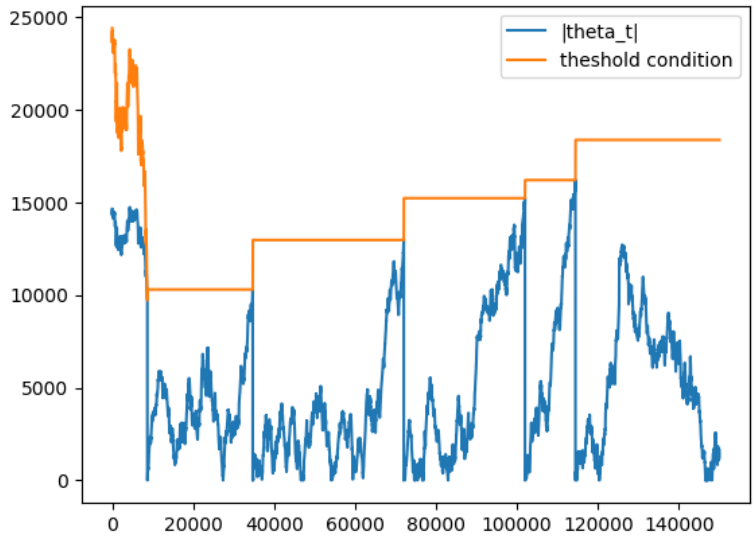
Now we have understood the logic of imbalance bar generation, let's understand how the process looks in details:

```
num_prev_bars = 3
expected_num_ticks_init = 100000
expected_num_ticks = expected_num_ticks_init
cum_theta = 0
num_ticks = 0
imbalance_array = []
imbalance_bars = []
bar_length_array = []
for row in data.rows:
    #track high,low,close, volume info
    num_ticks += 1
    tick_rule = get_tick_rule(price, prev_price)
    volume_imbalance = tick_rule * row['volume']
    imbalance_array.append(volume_imbalance)
    cum_theta += volume_imbalance
    if len(imbalance_bars) == 0 and len(imbalance_array) >= expected_num_ticks_init:
        expected_imbalance = ewma(imbalance_array, window=expected_num_ticks_init)

    if abs(cum_theta) >= expected_num_ticks * abs(expected_imbalance):
        bar = form_bar(open, high, low, close, volume)
        imbalance_bars.append(bar)
        bar_length_array.append(num_ticks)
        cum_theta, num_ticks = 0, 0
        expected_num_ticks = ewma(bar_lenght_array, window=num_prev_bars)
        expected_imbalance = ewma(imbalance_array, window = num_prev_bars * expected_
↪num_ticks)
```

Note that in algorithm pseudo-code we reset θ_t when bar is formed, in our case the formula for θ_t is:

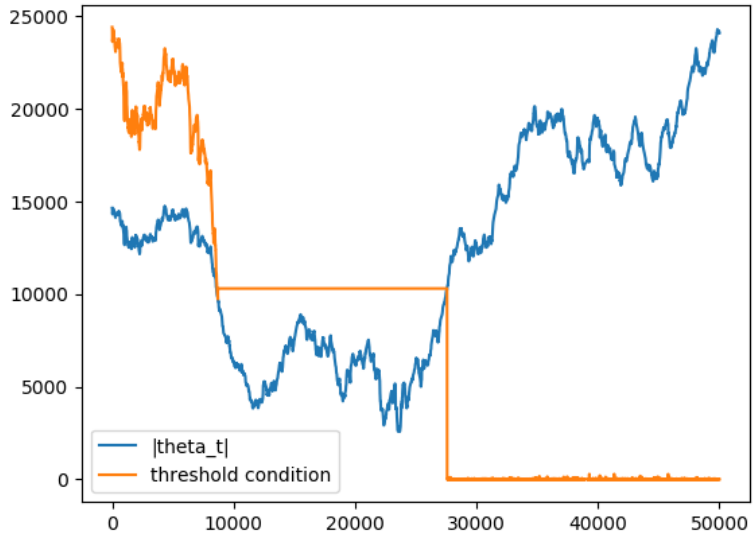
$$\theta_t = \sum_{t=t^*}^{T} b_t * v_t$$

Let's look at dynamics of $|\theta_t|$ and $E_0[T] * |2v^+ - E_0[v_t]|$ to understand why we decided to reset θ_t when bar is formed. The dynamics when theta value is reset:



Note that on the first ticks, threshold condition is not stable. Remember, before the first bar is generated, expected imbalance is calculated on every tick with window = expected_num_ticks_init, that is why it changes with every tick. After the first bar was generated both expected number of ticks ($E_0[T]$) and exptected volume imbalance ($2v^+ - E_0[v_t]$) are updated only when the next bar is generated

When theta is not reset:



The reason for that is due to the fact that theta is accumulated when several bars are generated theta value is not reset $\Rightarrow$ condition is met on small number of ticks $\Rightarrow$ length of the next bar converges to 1 $\Rightarrow$ bar is sampled on the next consecutive tick.

The logic described above is implemented in the **mlfinlab** package under ImbalanceBars

Examples

get_tick_imbalance_bars(*file_path*, *num_prev_bars*, *exp_num_ticks_init=100000*, *batch_size=2e7*, *verbose=True*, *to_csv=False*, *output_path=None*)

Creates the tick imbalance bars: date_time, open, high, low, close.

Parameters

- **file_path** – File path pointing to csv data.
- **num_prev_bars** – Number of previous bars used for EWMA window expected # of ticks
- **exp_num_ticks_init** – initial expected number of ticks per bar
- **batch_size** – The number of rows per batch. Less RAM = smaller batch size.
- **verbose** – Print out batch numbers (True or False)
- **to_csv** – Save bars to csv after every batch run (True or False)
- **output_path** – Path to csv file, if to_csv is True

Returns DataFrame of tick bars

```
from mlfinlab.data_structures import imbalance_data_structures


# Tick Imbalance Bars
tick_imbalance = imbalance_data_structures.get_tick_imbalance_bars('FILE_PATH',
num_prev_bars=3, exp_num_ticks_init=100000)
```

get_volume_imbalance_bars(*file_path*, *num_prev_bars*, *exp_num_ticks_init=100000*, *batch_size=2e7*, *verbose=True*, *to_csv=False*, *output_path=None*)

Creates the volume imbalance bars: date_time, open, high, low, close.

Parameters

- **file_path** – File path pointing to csv data.
- **num_prev_bars** – Number of previous bars used for EWMA window expected # of ticks
- **exp_num_ticks_init** – initial expected number of ticks per bar
- **batch_size** – The number of rows per batch. Less RAM = smaller batch size.
- **verbose** – Print out batch numbers (True or False)
- **to_csv** – Save bars to csv after every batch run (True or False)
- **output_path** – Path to csv file, if to_csv is True

Returns DataFrame of volume bars

```
from mlfinlab.data_structures import imbalance_data_structures


# Volume Imbalance Bars
volume_imbalance = imbalance_data_structures.get_volume_imbalance_bars('FILE_PATH',
num_prev_bars=3, exp_num_ticks_init=100000)
```

get_dollar_imbalance_bars (*file_path*, *num_prev_bars*, *exp_num_ticks_init=100000*, *batch_size=2e7*, *verbose=True*, *to_csv=False*, *output_path=None*)

Creates the dollar imbalance bars: date_time, open, high, low, close.

Parameters

- **file_path** – File path pointing to csv data.
- **num_prev_bars** – Number of previous bars used for EWMA window expected # of ticks
- **exp_num_ticks_init** – initial expected number of ticks per bar
- **batch_size** – The number of rows per batch. Less RAM = smaller batch size.
- **verbose** – Print out batch numbers (True or False)
- **to_csv** – Save bars to csv after every batch run (True or False)
- **output_path** – Path to csv file, if to_csv is True

Returns DataFrame of dollar bars

```
from mlfinlab.data_structures import imbalance_data_structures


# Dollar Imbalance Bars
dollar = imbalance_data_structures.get_dollar_bars('FILE_PATH',
num_prev_bars=3, exp_num_ticks_init=100000)
```

Run Bars

Run bars share the same mathematical structure as imblance bars, however, instead of looking at each individual trade, we are looking at sequences of trades in the same direction. The idea is that we are trying to detect order flow imbalance caused by actions such as large traders sweeping the order book or iceberg orders.

Examples of implementations of run bars can be seen below:

get_tick_run_bars (*file_path*, *num_prev_bars*, *exp_num_ticks_init=100000*, *batch_size=2e7*, *verbose=True*, *to_csv=False*, *output_path=None*)

Creates the tick run bars: date_time, open, high, low, close.

Parameters

- **file_path** – File path pointing to csv data.
- **num_prev_bars** – Number of previous bars used for EWMA window expected # of ticks
- **exp_num_ticks_init** – initial expected number of ticks per bar
- **batch_size** – The number of rows per batch. Less RAM = smaller batch size.
- **verbose** – Print out batch numbers (True or False)
- **to_csv** – Save bars to csv after every batch run (True or False)
- **output_path** – Path to csv file, if to_csv is True

Returns DataFrame of tick bars

```
from mlfinlab.data_structures import run_data_structures


# Tick Run Bars
```

(continues on next page)

(continued from previous page)

```
tick_run = run_data_structures.get_tick_run_bars('FILE_PATH',
num_prev_bars=3, exp_num_ticks_init=100000)
```

get_volume_run_bars (*file_path*, *num_prev_bars*, *exp_num_ticks_init=100000*, *batch_size=2e7*, *verbose=True*, *to_csv=False*, *output_path=None*)

Creates the volume run bars: date_time, open, high, low, close.

Parameters

- **file_path** – File path pointing to csv data.
- **num_prev_bars** – Number of previous bars used for EWMA window expected # of ticks
- **exp_num_ticks_init** – initial expected number of ticks per bar
- **batch_size** – The number of rows per batch. Less RAM = smaller batch size.
- **verbose** – Print out batch numbers (True or False)
- **to_csv** – Save bars to csv after every batch run (True or False)
- **output_path** – Path to csv file, if to_csv is True

Returns DataFrame of volume bars

```
from mlfinlab.data_structures import run_data_structures


# Volume Run Bars
volume_run = run_data_structures.get_volume_run_bars('FILE_PATH',
num_prev_bars=3, exp_num_ticks_init=100000)
```

get_dollar_run_bars (*file_path*, *num_prev_bars*, *exp_num_ticks_init=100000*, *batch_size=2e7*, *verbose=True*, *to_csv=False*, *output_path=None*)

Creates the dollar run bars: date_time, open, high, low, close.

Parameters

- **file_path** – File path pointing to csv data.
- **num_prev_bars** – Number of previous bars used for EWMA window expected # of ticks
- **exp_num_ticks_init** – initial expected number of ticks per bar
- **batch_size** – The number of rows per batch. Less RAM = smaller batch size.
- **verbose** – Print out batch numbers (True or False)
- **to_csv** – Save bars to csv after every batch run (True or False)
- **output_path** – Path to csv file, if to_csv is True

Returns DataFrame of dollar bars

```
from mlfinlab.data_structures import run_data_structures


# Dollar Run Bars
dollar_run = run_data_structures.get_dollar_run_bars('FILE_PATH',
num_prev_bars=3, exp_num_ticks_init=100000)
```

4.1.4 Research Notebooks

The following research notebooks can be used to better understand the previously discussed data structures

Standard Bars

- Getting Started
- Sample Techniques

Imbalance Bars

- Imbalance Bars

4.2 Filters

Filters are used to filter events based on some kind of trigger. For example a structural break filter can be used to filter events where a structural break occurs. This event is then used to measure the return from the event to some event horizon, say a day.

4.2.1 CUSUM Filter

Snippet 2.4, page 39, The Symmetric CUSUM Filter.

The CUSUM filter is a quality-control method, designed to detect a shift in the mean value of a measured quantity away from a target value.

The filter is set up to identify a sequence of upside or downside divergences from any reset level zero.

We sample a bar t if and only if S_t >= threshold, at which point S_t is reset to 0.

One practical aspect that makes CUSUM filters appealing is that multiple events are not triggered by raw_time_series hovering around a threshold level, which is a flaw suffered by popular market signals such as Bollinger Bands.

It will require a full run of length threshold for raw_time_series to trigger an event. Once we have obtained this subset of event-driven bars, we will let the ML algorithm determine whether the occurrence of such events constitutes actionable intelligence.

Threshold can be either fixed (float value) or dynamic (pd.Series).

Below is an implementation of the Symmetric CUSUM filter. Note: As per the book this filter is applied to closing prices but we extended it to also work on other time series such as volatility.

cusum_filter(*raw_time_series*, *threshold*, *time_stamps=True*)

Parameters

- **raw_time_series** – (series) of close prices (or other time series, e.g. volatility).
- **threshold** – (float or pd.Series) when the abs(change) is larger than the threshold, the function captures it as an event, can be dynamic if threshold is pd.Series
- **time_stamps** – (bool) default is to return a DateTimeIndex, change to false to have it return a list.

Returns (datetime index vector) vector of datetimes when the events occurred. This is used later to sample.

An example showing how the CUSUM filter can be used to downsample a time series of close prices can be seen below:

```
from mlfinlab.filters import cusum_filter

cusum_events = cusum_filter(data['close'], threshold=0.05)
```

4.2.2 Z-score Filter

Z-score filter is used to define explosive/peak points in time series. https://stackoverflow.com/questions/22583391/peak-signal-detection-in-realtime-timeseries-data

It uses rolling simple moving average, rolling simple moving standard deviation and z_score(threshold). When current time series value exceeds (rolling average + z_score * rolling std) event is sampled

z_score_filter(*raw_time_series*, *mean_window*, *std_window*, *z_score=3*, *time_stamps=True*)

Parameters

- **raw_time_series** – (series) of close prices (or other time series, e.g. volatility).
- **mean_window** – (int): rolling mean window
- **std_window** – (int): rolling std window
- **z_score** – (float): number of standard deviations to trigger the event
- **time_stamps** – (bool) default is to return a DateTimeIndex, change to false to have it return a list.

Returns (datetime index vector) vector of datetimes when the events occurred. This is used later to sample.

An example of how the Z-score filter can be used to downsample a time series:

```
from mlfinlb.filters import z_score_filter

z_score_events = z_score_filter(data['close'], mean_window=100, std_window=100, z_
↪score=3)
```

4.3 Labeling

The primary labeling method used in financial academia is the fixed-time horizon method. While ubiquitous, this method has many faults which are remedied by the triple-barrier method discussed below. The triple-barrier method can be extended to incorporate meta-labeling which will also be demonstrated and discussed below.

4.3.1 Triple-Barrier Method

The idea behind the triple-barrier method is that we have three barriers: an upper barrier, a lower barrier, and a vertical barrier. The upper barrier represents the threshold an observation's return needs to reach in order to be considered a buying opportunty (a label of 1), the lower barrier represents the threshold an observation's return needs to reach in order to be considered a selling opportunity (a label of -1), and the vertical barrier represents the amount of time an observation has to reach its given return in either direction before it is given a label of 0. This concept can be better understood visually and is shown in the figure below taken from Advances in Financial Machine Learning (reference):

One of the major faults with the fixed-time horizon method is that observations are given a label with respect to a certain threshold after a fixed interval regardless of their respective volatilities. In other words, the expected returns of every observation are treated equally regardless of the associated risk. The triple-barrier method tackles this issue by dynamically setting the upper and lower barriers for each observation based on their given volatilities.

4.3.2 Meta-Labeling

Advances in Financial Machine Learning, Chapter 3, page 50. Reads:

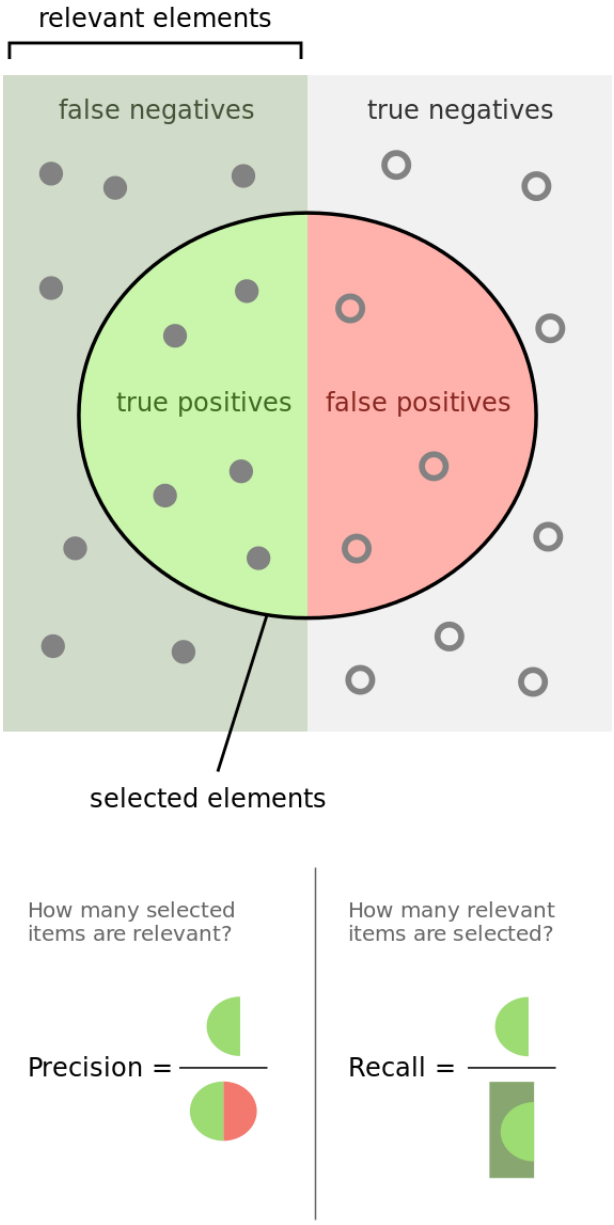
"Suppose that you have a model for setting the side of the bet (long or short). You just need to learn the size of that bet, which includes the possibility of no bet at all (zero size). This is a situation that practitioners face regularly. We often know whether we want to buy or sell a product, and the only remaining question is how much money we should risk in such a bet. We do not want the ML algorithm to learn the side, just to tell us what is the appropriate size. At this point, it probably does not surprise you to hear that no book or paper has so far discussed this common problem. Thankfully, that misery ends here."“

I call this problem meta-labeling because we want to build a secondary ML model that learns how to use a primary exogenous model.

The ML algorithm will be trained to decide whether to take the bet or pass, a purely binary prediction. When the predicted label is 1, we can use the probability of this secondary prediction to derive the size of the bet, where the side (sign) of the position has been set by the primary model.

How to use Meta-Labeling

Binary classification problems present a trade-off between type-I errors (false positives) and type-II errors (false negatives). In general, increasing the true positive rate of a binary classifier will tend to increase its false positive rate. The receiver operating characteristic (ROC) curve of a binary classifier measures the cost of increasing the true positive rate, in terms of accepting higher false positive rates.



The image illustrates the so-called "confusion matrix." On a set of observations, there are items that exhibit a condition (positives, left rectangle), and items that do not exhibit a condition (negative, right rectangle). A binary classifier predicts that some items exhibit the condition (ellipse), where the TP area contains the true positives and the TN area contains the true negatives. This leads to two kinds of errors: false positives (FP) and false negatives (FN). "Precision" is the ratio between the TP area and the area in the ellipse. "Recall" is the ratio between the TP area and the area in the left rectangle. This notion of recall (aka true positive rate) is in the context of classification problems, the analogous

to "power" in the context of hypothesis testing. "Accuracy" is the sum of the TP and TN areas divided by the overall set of items (square). In general, decreasing the FP area comes at a cost of increasing the FN area, because higher precision typically means fewer calls, hence lower recall. Still, there is some combination of precision and recall that maximizes the overall efficiency of the classifier. The F1-score measures the efficiency of a classifier as the harmonic average between precision and recall.

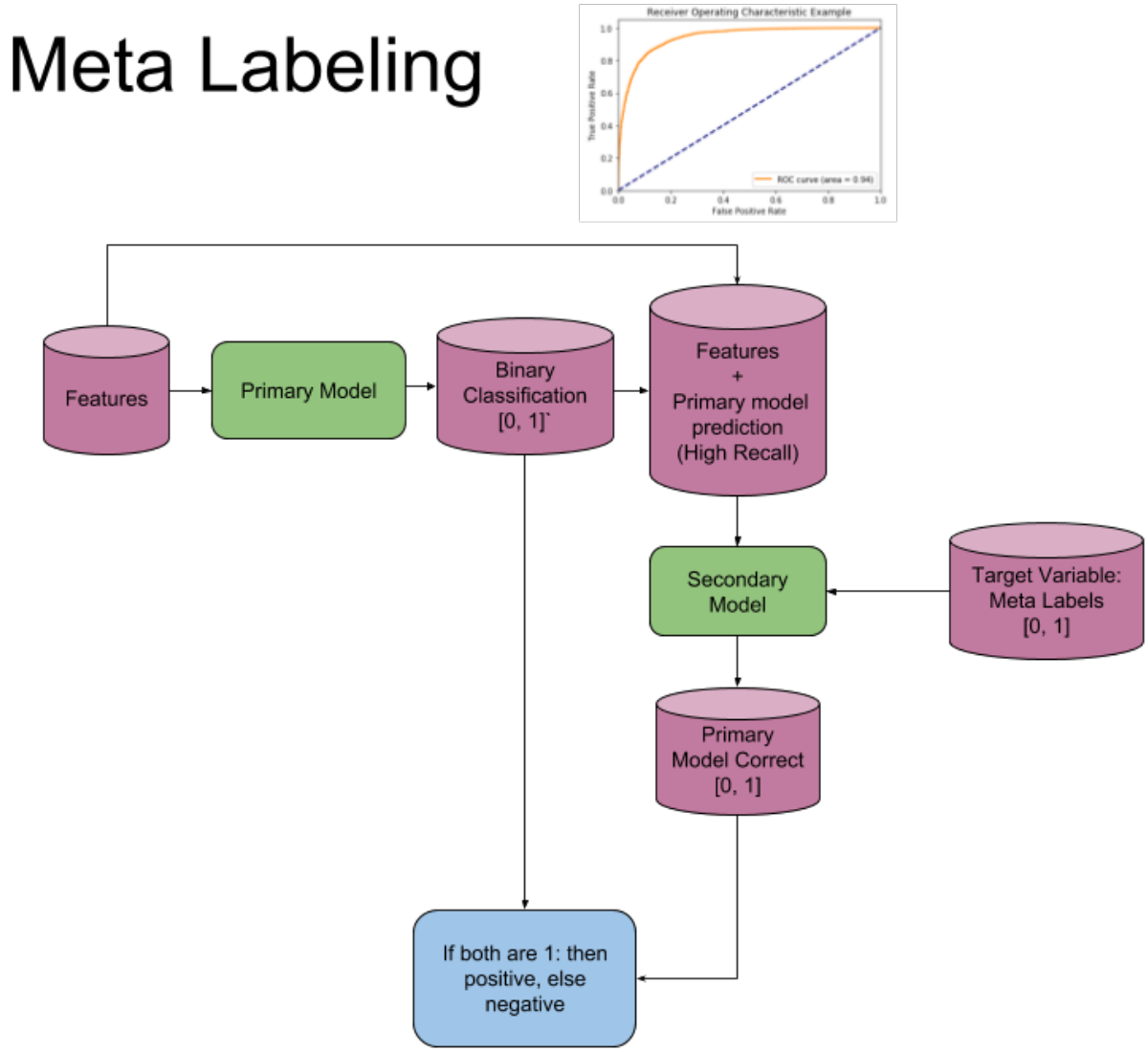
Meta-labeling is particularly helpful when you want to achieve higher F1-scores. First, we build a model that achieves high recall, even if the precision is not particularly high. Second, we correct for the low precision by applying meta-labeling to the positives predicted by the primary model.

Meta-labeling will increase your F1-score by filtering out the false positives, where the majority of positives have already been identified by the primary model. Stated differently, the role of the secondary ML algorithm is to determine whether a positive from the primary (exogenous) model is true or false. It is not its purpose to come up with a betting opportunity. Its purpose is to determine whether we should act or pass on the opportunity that has been presented.

Meta-labeling is a very powerful tool to have in your arsenal, for four additional reasons. **First**, ML algorithms are often criticized as black boxes. Meta-labeling allows you to build an ML system on top of a white box (like a fundamental model founded on economic theory). This ability to transform a fundamental model into an ML model should make meta-labeling particularly useful to "quantamental" firms. **Second**, the effects of overfitting are limited when you apply metalabeling, because ML will not decide the side of your bet, only the size. **Third**, by decoupling the side prediction from the size prediction, meta-labeling enables sophisticated strategy structures. For instance, consider that the features driving a rally may differ from the features driving a sell-off. In that case, you may want to develop an ML strategy exclusively for long positions, based on the buy recommendations of a primary model, and an ML strategy exclusively for short positions, based on the sell recommendations of an entirely different primary model. **Fourth**, achieving high accuracy on small bets and low accuracy on large bets will ruin you. As important as identifying good opportunities is to size them properly, so it makes sense to develop an ML algorithm solely focused on getting that critical decision (sizing) right. We will retake this fourth point in Chapter 10. In my experience, meta-labeling ML models can deliver more robust and reliable outcomes than standard labeling models.

Model Architecture

The following image explains the model architecture. The **first** step is to train a primary model (binary classification). **Second** a threshold level is determined at which the primary model has a high recall, in the coded example you will find that 0.30 is a good threshold, ROC curves could be used to help determine a good level. **Third** the features from the first model are concatenated with the predictions from the first model, into a new feature set for the secondary model. Meta Labels are used as the target variable in the second model. Now fit the second model. **Fourth** the prediction from the secondary model is combined with the prediction from the primary model and only where both are true, is your final prediction true. I.e. if your primary model predicts a 3 and your secondary model says you have a high probability of the primary model being correct, is your final prediction a 3, else not 3.



4.3.3 Implementation

The following functions are used for the triple-barrier method which works in tandem with meta-labeling.

get_daily_vol (*close*, *lookback=100*)

Snippet 3.1 computes the daily volatility at intraday estimation points, applying a span of lookback days to an exponentially weighted moving standard deviation.

See the pandas documentation for details on the pandas.Series.ewm function.

Note: This function is used to compute dynamic thresholds for profit taking and stop loss limits.

Parameters

- **close** – Closing prices
- **lookback** – lookback period to compute volatility

Returns series of daily volatility value

add_vertical_barrier(*t_events*, *close*, *num_days=0*, *num_hours=0*, *num_minutes=0*, *num_seconds=0*)

Snippet 3.4 page 49, Adding a Vertical Barrier

For each index in t_events, it finds the timestamp of the next price bar at or immediately after a number of days num_days. This vertical barrier can be passed as an optional argument t1 in get_events.

This function creates a series that has all the timestamps of when the vertical barrier would be reached.

Parameters

- **t_events** – (series) series of events (symmetric CUSUM filter)
- **close** – (series) close prices
- **num_days** – (int) number of days to add for vertical barrier
- **num_hours** – (int) number of hours to add for vertical barrier
- **num_minutes** – (int) number of minutes to add for vertical barrier
- **num_seconds** – (int) number of seconds to add for vertical barrier

Returns (series) timestamps of vertical barriers

get_events(*close*, *t_events*, *pt_sl*, *target*, *min_ret*, *num_threads*, *vertical_barrier_times=False*, *side_prediction=None*)

Snippet 3.6 page 50, Getting the Time of the First Touch, with Meta Labels

This function is orchestrator to meta-label the data, in conjunction with the Triple Barrier Method.

Parameters

- **close** – (series) Close prices
- **t_events** – (series) of t_events. These are timestamps that will seed every triple barrier. These are the timestamps selected by the sampling procedures discussed in Chapter 2, Section 2.5. Eg: CUSUM Filter
- **pt_sl** – (2 element array) element 0, indicates the profit taking level; element 1 is stop loss level. A non-negative float that sets the width of the two barriers. A 0 value means that the respective horizontal barrier (profit taking and/or stop loss) will be disabled.
- **target** – (series) of values that are used (in conjunction with pt_sl) to determine the width of the barrier. In this program this is daily volatility series.
- **min_ret** – (float) The minimum target return required for running a triple barrier search.
- **num_threads** – (int) The number of threads concurrently used by the function.
- **vertical_barrier_times** – (series) A pandas series with the timestamps of the vertical barriers. We pass a False when we want to disable vertical barriers.
- **side_prediction** – (series) Side of the bet (long/short) as decided by the primary model

Returns

(data frame) of events events.index is event's starttime

events['t1'] is event's endtime

events['trgt'] is event's target

events['side'] (optional) implies the algo's position side

get_bins (*triple_barrier_events*, *close*)
Snippet 3.7, page 51, Labeling for Side & Size with Meta Labels

Compute event's outcome (including side information, if provided). events is a DataFrame where:

Now the possible values for labels in out['bin'] are {0,1}, as opposed to whether to take the bet or pass, a purely binary prediction. When the predicted label the previous feasible values {1,0,1}. The ML algorithm will be trained to decide is 1, we can use the probability of this secondary prediction to derive the size of the bet, where the side (sign) of the position has been set by the primary model.

Parameters

- **triple_barrier_events** – (data frame)

  events.index is event's starttime

  events['t1'] is event's endtime

  events['trgt'] is event's target

  events['side'] (optional) implies the algo's position side

  Case 1: ('side' not in events): bin in (-1,1) <-label by price action

  Case 2: ('side' in events): bin in (0,1) <-label by pnl (meta-labeling)

- **close** – (series) close prices

Returns (data frame) of meta-labeled events

drop_labels (*events*, *min_pct=.05*)
Snippet 3.8 page 54 This function recursively eliminates rare observations.

Parameters

- **events** – (data frame) events
- **min_pct** – (float) a fraction used to decide if the observation occurs less than that fraction

Returns (data frame) of event

4.3.4 Example

Suppose we use a mean reverting strategy as our primary model, giving each observation a label of 1 or -1. We can then use meta-labeling to act as a filter for the bets of our primary model.

```
import mlfinlab as ml
import numpy as np
import pandas as pd
```

```
# Read in data
data = pd.read_csv('FILE_PATH')
```

Assuming we have a pandas series with the timestamps of our observations and their respective labels given by the primary model, the process to generate meta-labels goes as follows.

```
# Compute daily volatility
daily_vol = ml.util.get_daily_vol(close=data['close'], lookback=50)

# Apply Symmetric CUSUM Filter and get timestamps for events
# Note: Only the CUSUM filter needs a point estimate for volatility
cusum_events = ml.filters.cusum_filter(data['close'],
```

(continues on next page)

(continued from previous page)

```
threshold=daily_vol['2011-09-01':'2018-01-01'].mean()*0.5)

# Compute vertical barrier
vertical_barriers = ml.labeling.add_vertical_barrier(t_events=cusum_events,
close=data['close'], num_days=1)
```

Once we have computed our daily volatility along with our vertical time barriers and have downsampled our series using the CUSUM filter, we can use the triple-barrier method to compute our meta-labels by passing in the side predicted by the primary model.

```
pt_sl = [1, 2]
min_ret = 0.005
triple_barrier_events = ml.labeling.get_events(close=data['close'],
                                               t_events=cusum_events,
                                               pt_sl=pt_sl,
                                               target=daily_vol,
                                               min_ret=min_ret,
                                               num_threads=3,
                                               vertical_barrier_times=vertical_barriers,
                                               side_prediction=data['side'])
```

As can be seen above, we have scaled our lower barrier and set our minimum return to 0.005.

Meta-labels can then be computed using the time that each observation touched its respective barrier

```
meta_labels = ml.labeling.get_bins(triple_barrier_events, data['close'])
```

This example ends with creating the meta-labels. To see a further explanation of using these labels in a secondary model to help filter out false positives, see the research notebooks below.

4.3.5 Research Notebooks

The following research notebooks can be used to better understand the triple-barrier method and meta-labeling

Triple-Barrier Method

- Trend Follow Question
- Bollinger band Question

Meta-Labeling

- Meta Labeling MNIST

4.4 Sampling

In financial machine learning, samples are not independent. The most part of traditional machine learning algorithms assume that samples are IID, in case of financial machine learning samples are neither identically distributed not independent. In this section we will tackle the problem of samples dependency. As you remember, we mostly label our data sets using the triple-barrier method. Each label in triple-barrier event has label index and label end time (t1) which corresponds to time when one of barriers was touched.

4.4.1 Sample Uniqueness

Let's look at example of 3 samples: A, B, C.

Imagine that:

- A was generated at t_1 and triggered on t_8
- B was generated at t_3 and triggered on t_6
- C was generated on t_7 and triggered on t_9

In this case we see that A used information about returns on $[t_1, t_8]$ to generate label-endtime which overlaps with $[t_3, t_6]$ which was used by B, however C didn't use any returns information which was used by to label other samples. Here we would like to introduce the concept of concurrency.

We say that labels y_i and y_j are concurrent at t if they are a function of at least one common return at $r_{t-1,t}$

In terms of concurrency label C is the most 'pure' as it doesn't use any piece of information from other labels, while A is the 'dirtiest' as it uses information from both B and C. By understanding average label uniqueness you can measure how 'pure' your dataset is based on concurrency of labels. We can measure average label uniqueness using get_av_uniqueness_from_triple_barrier function from the mlfinlab package.

This function is the orchestrator to derive average sample uniqueness from a dateset labeled by the triple barrier method.

get_av_uniqueness_from_triple_barrier(*triple_barrier_events*, *close_series*, *num_threads*)

Parameters

- **triple_barrier_events** – (data frame) of events from labeling.get_events()
- **close_series** – (pd.Series) close prices.
- **num_threads** – (int) The number of threads concurrently used by the function.

Returns (pd.Series) average uniqueness over event's lifespan for each index in triple_barrier_events

An example of calculating average uniqueness given that we have already found our barrier events can be seen below:

```
import pandas as pd
import numpy as np
from mlfinlab.sampling.concurrent import get_av_uniqueness_from_triple_barrier


barrier_events = pd.read_csv('FILE_PATH', index_col=0, parse_dates=[0,2])
close_prices = pd.read_csv('FILE_PATH', index_col=0, parse_dates=[0,2])


av_unique = get_av_uniqueness_from_triple_barrier(barrier_events, close_prices.close,
 num_threads=3)
```

We would like to build our model in such a way that it takes into account labels concurrency. In order to do that we need to look at the bootstrapping algorithm of Random Forest.

4.4.2 Sequential Bootstrapping

The key power of ensemble learning techniques is bagging (which is bootstrapping with replacement). The key idea behind bagging is to randomly choose samples for each decision tree. In this case trees become diverse and by averaging predictions of diverse tress built on randomly selected samples and random subset of features data scientists make the algorithm much less prone to overfit.

However, in our case we would not only like to randomly choose samples but also choose samples which are unique and non-concurrent. But how can we solve this problem? Here comes Sequential Bootstrapping algorithm.

The key idea behind Sequential Bootstrapping is to select samples in such a way that on each iteration we maximize average uniqueness of selected subsamples.

Implementation

The core functions behind Sequential Bootstrapping are implemented in mlfinlab and can be seen below:

get_ind_matrix(*triple_barrier_events*, *price_bars*)
Snippet 4.3, page 65, Build an Indicator Matrix

Get indicator matrix. The book implementation uses bar_index as input, however there is no explanation how to form it. We decided that using triple_barrier_events and price bars by analogy with concurrency is the best option.

Parameters

- **samples_info_sets** – (pd.Series): triple barrier events.t1 from labeling.get_events
- **price_bars** – (pd.DataFrame): price bars which were used to form triple barrier events

Returns (np.array) indicator binary matrix indicating what (price) bars influence the label for each observation

get_ind_mat_average_uniqueness(*ind_mat*)
Snippet 4.4. page 65, Compute Average Uniqueness Average uniqueness from indicator matrix

Parameters **ind_mat** – (np.matrix) indicator binary matrix

Returns (float) average uniqueness

get_ind_mat_label_uniqueness(*ind_mat*)
An adaption of Snippet 4.4. page 65, which returns the indicator matrix element uniqueness.

Parameters **ind_mat** – (np.matrix) indicator binary matrix

Returns (np.matrix) element uniqueness

seq_bootstrap(*ind_mat*, *sample_length=None*, *warmup_samples=None*, *compare=False*, *verbose=False*, *random_state=np.random.RandomState()*)
Snippet 4.5, Snippet 4.6, page 65, Return Sample from Sequential Bootstrap Generate a sample via sequential bootstrap.

Note: Moved from pd.DataFrame to np.matrix for performance increase

Parameters

- **ind_mat** – (data frame) indicator matrix from triple barrier events
- **sample_length** – (int) Length of bootstrapped sample
- **warmup_samples** – (list) list of previously drawn samples
- **compare** – (boolean) flag to print standard bootstrap uniqueness vs sequential bootstrap uniqueness
- **verbose** – (boolean) flag to print updated probabilities on each step
- **random_state** – (np.random.RandomState) random state

Returns (array) of bootstrapped samples indexes

Example

An example of Sequential Bootstrap using a a toy example from the book can be seen below.

Consider a set of labels $\{y_i\}_{i=0,1,2}$ where:

- label y_0 is a function of return $r_{0,2}$
- label y_1 is a function of return $r_{2,3}$
- label y_2 is a function of return $r_{4,5}$

The first thing we need to do is to build and indicator matrix. Columns of this matrix correspond to samples and rows correspond to price returns timestamps which were used during samples labelling. In our case indicator matrix is:

```
ind_mat = pd.DataFrame(index = range(0,6), columns=range(0,3))
```

```
ind_mat.loc[:, 0] = [1, 1, 1, 0, 0, 0]
ind_mat.loc[:, 1] = [0, 0, 1, 1, 0, 0]
ind_mat.loc[:, 2] = [0, 0, 0, 0, 1, 1]
```

One can use get_ind_matrix method from mlfinlab to build indicator matrix from triple-barrier events.

```
triple_barrier_ind_mat = get_ind_matrix(barrier_events)
```

We can get average label uniqueness on indicator matrix using get_ind_mat_average_uniqueness function from mlfinlab.

```
ind_mat_uniqueness = get_ind_mat_average_uniqueness(triple_barrier_ind_mat)
```

Let's get the first sample average uniqueness (we need to filter out zeros to get unbiased result).

```
first_sample = ind_mat_uniqueness[0]
first_sample[first_sample > 0].mean()
>> 0.26886446886446885
```

```
av_unique.iloc[0]
>> tW    0.238776
```

As you can see it is quite close to values generated by **get_av_uniqueness_from_triple_barrier** function call.

Let's move back to our example. In Sequential Bootstrapping algorithm we start with an empty array of samples (ϕ) and loop through all samples to get the probability of chosing the sample based on average uniqueness of reduced indicator matrix constructed from [previously chosen columns] + sample

```
phi = []
while length(phi) < number of samples to bootstrap:
    average_uniqueness_array = []
    for sample in samples:
        previous_columns  = phi
        ind_mat_reduced = ind_mat[previous_columns + i]
        average_uniqueness_array[sample] = get_ind_mat_average_uniqueness(ind_mat_
↪reduced)
    // normalise so that probabilities sum up to 1
    probability_array = average_uniqueness_array / sum(average_uniqueness_array)
    chosen_sample = random_choice(samples, probability = probability_array)
    phi.append(chosen_sample)
```

For performance increase we optimized and parallesied for-loop using numba, which corresponds to bootstrap_loop_run function.

Not let's finish the example:

To be as close to the mlfinlab implementation let's convert ind_mat to numpy matrix

```
ind_mat = ind_mat.values
```

1st iteration:

On the first step all labels will have equal probalities as average uniqueness of matrix with 1 column is 1. Say we have chosen 1 on the first step

2nd iteration:

```
phi = [1] # Sample chosen from the 2st step
uniqueness_array = np.array([None, None, None])
for i in range(0, 3):
    ind_mat_reduced = ind_mat[:, phi + [i]]
    label_uniqueness = get_ind_mat_average_uniqueness(ind_mat_reduced)[-1]
    # The last value corresponds to appended i
    uniqueness_array[i] = (label_uniqueness[label_uniqueness > 0].mean())
prob_array = uniqueness_array / sum(uniqueness_array)
```

```
  prob_array
  >> array([0.35714285714285715, 0.21428571428571427, 0.42857142857142855],
dtype=object)
```

Probably the second chosen feature will be 2 (prob_array[2] = 0.42857 which is the largest probability). As you can see up till now the algorithm has chosen two the least concurrent labels (1 and 2)

3rd iteration:

```
phi = [1,2]
uniqueness_array = np.array([None, None, None])
for i in range(0, 3):
    ind_mat_reduced = ind_mat[:, phi + [i]]
    label_uniqueness = get_ind_mat_average_uniqueness(ind_mat_reduced)[-1]
    uniqueness_array[i] = (label_uniqueness[label_uniqueness > 0].mean())
prob_array = uniqueness_array / sum(uniqueness_array)
```

```
  prob_array
  >> array([0.45454545454545453, 0.2727272727272727, 0.2727272727272727],
dtype=object)
```

Sequential Bootstrapping tries to minimise the probability of repeated samples so as you can see the most probable sample would be 0 with 1 and 2 already selected.

4th iteration:

```
phi = [1, 2, 0]
uniqueness_array = np.array([None, None, None])
for i in range(0, 3):
    ind_mat_reduced = ind_mat[:, phi + [i]]
    label_uniqueness = get_ind_mat_average_uniqueness(ind_mat_reduced)[-1]
    uniqueness_array[i] = (label_uniqueness[label_uniqueness > 0].mean())
prob_array = uniqueness_array / sum(uniqueness_array)
```

```
  prob_array
  >> array([0.32653061224489793, 0.3061224489795918, 0.36734693877551017],
dtype=object)
```

The most probable sample would be 2 in this case

After 4 steps of sequential bootstrapping our drawn samples are [1, 2, 0, 2]

Let's see how this example is solved by the mlfinlab implementation. To reproduce that:

1) we need to set warmup to [1], which corresponds to phi = [1] on the first step
2) verbose = True to print updated probabilities

```
samples = seq_bootstrap(ind_mat, sample_length=4, warmup_samples=[1], verbose=True)
>> [0.33333333 0.33333333 0.33333333]
>> [0.35714286 0.21428571 0.42857143]
>> [0.45454545 0.27272727 0.27272727]
>> [0.32653061 0.30612245 0.36734694]
```

```
samples
>> [1, 2, 0, 2]
```

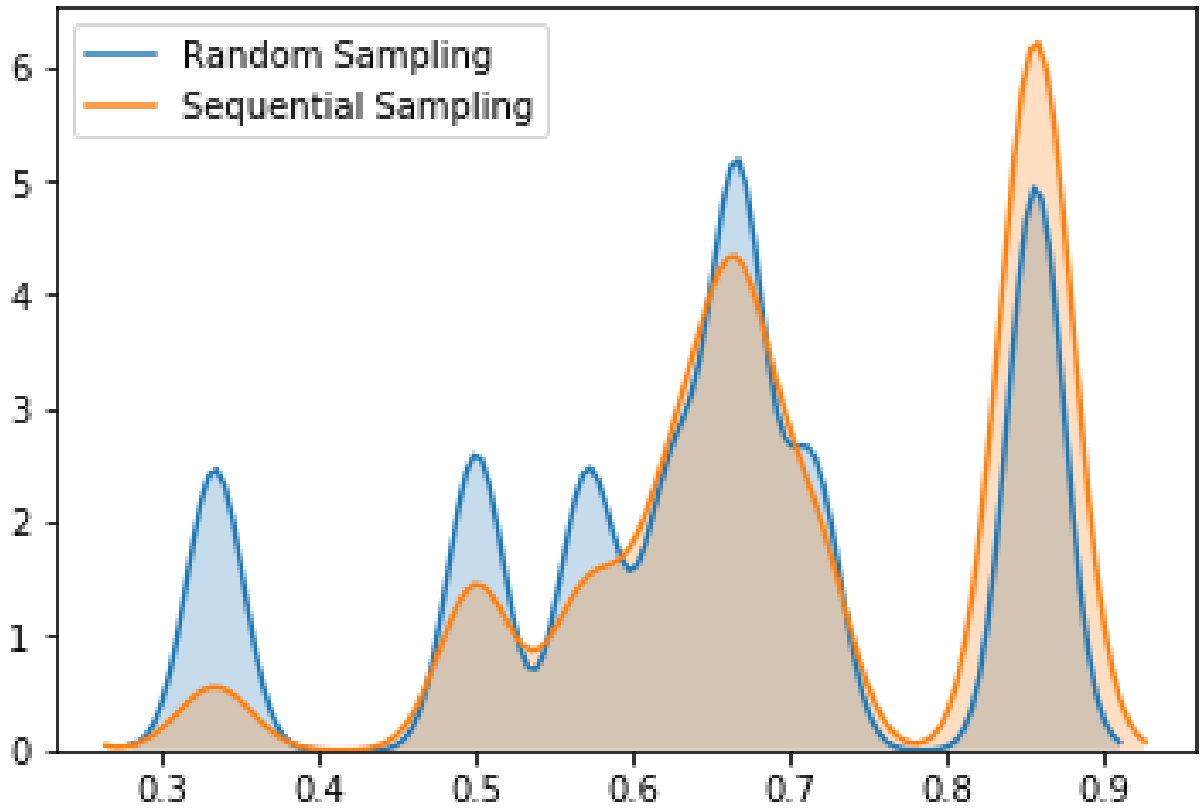
As you can see the first 2 iterations of algorithm yield the same probabilities, however sometimes the algorithm randomly chooses not the 2 sample on 2nd iteration that is why further probabilities are different from the example above. However, if you repeat the process several times you'll see that on average drawn sample equal to the one from the example

Monte-Carlo Experiment

Let's see how sequential bootstrapping increases average label uniqueness on this example by generating 3 samples using sequential bootstrapping and 3 samples using standard random choise, repeat the experiment 10000 times and record corresponding label uniqueness in each experiment

```
standard_unq_array = np.zeros(10000) * np.nan # Array of random sampling uniqueness
seq_unq_array = np.zeros(10000) * np.nan # Array of Sequential Bootstapping uniqueness
for i in range(0, 10000):
    bootstrapped_samples = seq_bootstrap(ind_mat, sample_length=3)
    random_samples = np.random.choice(ind_mat.shape[1], size=3)

    random_unq = get_ind_mat_average_uniqueness(ind_mat[:, random_samples])
    random_unq_mean = random_unq[random_unq > 0].mean()

    sequential_unq = get_ind_mat_average_uniqueness(ind_mat[:, bootstrapped_samples])
    sequential_unq_mean = sequential_unq[sequential_unq > 0].mean()

    standard_unq_array[i] = random_unq_mean
    seq_unq_array[i] = sequential_unq_mean
```

KDE plots of label uniqueness support the fact that sequential bootstrapping gives higher average label uniqueness



We can compare average label uniqueness using sequential bootstrap vs label uniqueness using standard random sampling by setting compare parameter to True. We have massively increased the performance of Sequential Bootstrapping which was described in the book. For comparison generating 50 samples from 8000 barrier-events would take 3 days, we have reduced time to 10-12 seconds which decreases by increasing number of CPUs.

Let's apply sequential bootstrapping to our full data set and draw 50 samples:

```
Standard uniqueness: 0.9465875370919882
Sequential uniqueness: 0.9913169319826338
```

Sometimes you would see that standard bootstrapping gives higher uniqueness, however as it was shown in Monte-Carlo example, on average Sequential Bootstrapping algorithm has higher average uniqueness.

4.4.3 Sample Weights

mlfinlab supports two methods of applying sample weights. The first is weighting an observation based on its given return as well as average uniqueness. The second is weighting an observation based on a time decay.

By Returns and Average Uniqueness

The following function utilizes a samples average uniqueness and its return to compute sample weights:

get_weights_by_return (*triple_barrier_events*, *close_series*, *num_threads=5*)

Parameters

- **triple_barrier_events** – (data frame) of events from labeling.get_events()

- **close_series** – (pd.Series) close prices
- **num_threads** – (int) the number of threads concurrently used by the function.

Returns (pd.Series) of sample weights based on number return and concurrency

This function can be utilized as shown below assuming we have already found our barrier events

```
import pandas as pd
import numpy as np
from mlfinlab.sampling.attribution import get_weights_by_return


barrier_events = pd.read_csv('FILE_PATH', index_col=0, parse_dates=[0,2])
close_prices = pd.read_csv('FILE_PATH', index_col=0, parse_dates=[0,2])


sample_weights = get_weights_by_return(barrier_events, close_prices.close,
 num_threads=3)
```

By Time Decay

The following function assigns sample weights using a time decay factor

get_weights_by_time_decay (*triple_barrier_events*, *close_series*, *num_threads=5*, *decay=1*)

Parameters

- **triple_barrier_events** – (data frame) of events from labeling.get_events()
- **close_series** – (pd.Series) close prices
- **num_threads** – (int) the number of threads concurrently used by the function.
- **decay** – (int) decay factor - decay = 1 means there is no time decay - 0 < decay < 1 means that weights decay linearly over time, but every observation still receives a strictly positive weight, regadless of how old - decay = 0 means that weights converge linearly to zero, as they become older - decay < 0 means that the oldes portion c of the observations receive zero weight (i.e they are erased from memory)

This function can be utilized as shown below assuming we have already found our barrier events

```
import pandas as pd
import numpy as np
from mlfinlab.sampling.attribution import get_weights_by_time_decay


barrier_events = pd.read_csv('FILE_PATH', index_col=0, parse_dates=[0,2])
close_prices = pd.read_csv('FILE_PATH', index_col=0, parse_dates=[0,2])


sample_weights =  get_weights_by_time_decay(barrier_events, close_prices.close,
num_threads=3, decay=0.4)
```

4.4.4 Research Notebooks

The following research notebooks can be used to better understand the previously discussed sampling methods

Sample Uniqueness and Weights

- Sample Uniqueness and Weights

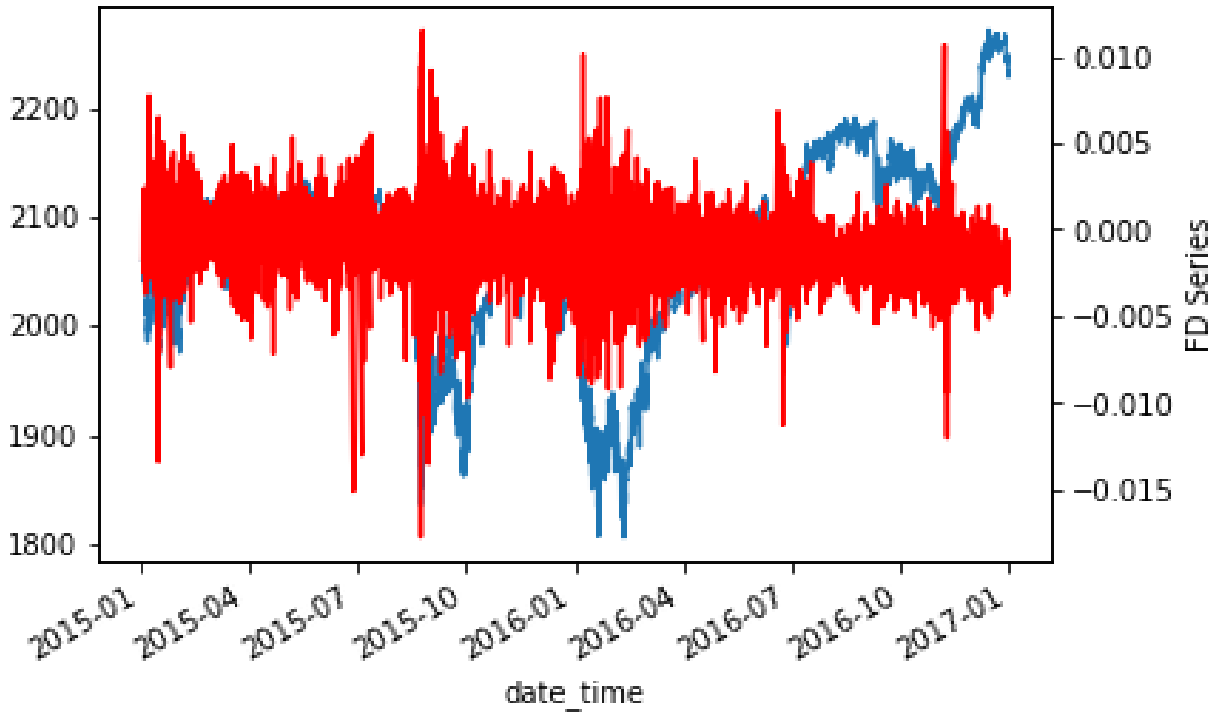
Sequential Bootstrapping

- Sequential Bootstrapping

4.5 Fractionally Differentiated Features

One of the challenges of quantitative analysis in finance is that price time series have trends or non-constant mean. This makes the time series non-stationary. Non-stationary time series are hard to work with when we want to do inferential analysis such as average and variance of returns, or probability of loss. Stationary series also help in supervised learning methods. Specifically, in supervised learning one needs to map hitherto unseen observations to a set of labeled examples and determine the label of the new observation. As Marcos Lopez de Prado (MLdP) says in Chapter 5, "if the features are not stationary we cannot map the new observation to a large number of known examples". However, to make a time series (or a feature) stationary often requires data transformations like computing changes (change in price, yields or volatility). These transformations also leave the time series bereft of any memory and thereby reducing or eliminating its predictive capability. Fractionally differentiated features tackle this problem by deriving features through fractionally differentiating a time series to the point where the series is stationary, but not over differencing such that we lose all predictive power.

The following graph shows a fractionally differenced series plotted over the original closing price series:



4.5.1 Implementation

The following function implemented in mlfinlab can be used to derive fractionally differentiated features.

frac_diff_ffd(*series*, *diff_amt*, *thresh=1e-5)*)
Source: Chapter 5, AFML (section 5.5, page 83);

The steps are as follows:

- Compute weights (this is a one-time exercise)
- Iteratively apply the weights to the price series and generate output points

This is the expanding window variant of the fracDiff algorithm

1. Note: For thresh-1, nothing is skipped
2. Note: diff_amt can be any positive fractional, not necessarility bounded [0, 1]

Parameters

- **series** – (pd.Series) a time series that needs to be differenced
- **diff_amt** – (float) Differencing amount
- **thresh** – (float) threshold or epsilon

Returns (pd.DataFrame) data frame of differenced series

Given that we know the amount we want to difference our price series, fractionally differentiated features can be derived as follows:

```
import numpy as np
import pandas as pd
from mlfinlab.features.fracdiff import frac_diff_ffd

data = pd.read_csv('FILE_PATH')
frac_diff_series = frac_diff_ffd(data['close'], 0.5)
```

4.5.2 Research Notebook

The following research notebook can be used to better understand fractionally differentiated features.

Fractionally Differentiated Features

- Fractionally Differentiated Features

4.6 Cross Validation

This implementation is based on Chapter 7 of the book. The purpose of performing cross validation is to reduce the probability of over-fitting and the book recommends it as the main tool of research. There are two innovations compared to the classical K-Fold Cross Validation implemented in sklearn.

1. The first one is a process called **purging** which removes from the *training* set those samples that are build with information that overlaps samples in the *testing* set. More details on this in section 7.4.1, page 105.

2. The second innovation is a process called **embargo** which removes a number of observations from the *end* of the test set. This further prevents leakage where the purging process is not enough. More details on this in section 7.4.2, page 107.

Implements the book chapter 7 on Cross Validation for financial data.

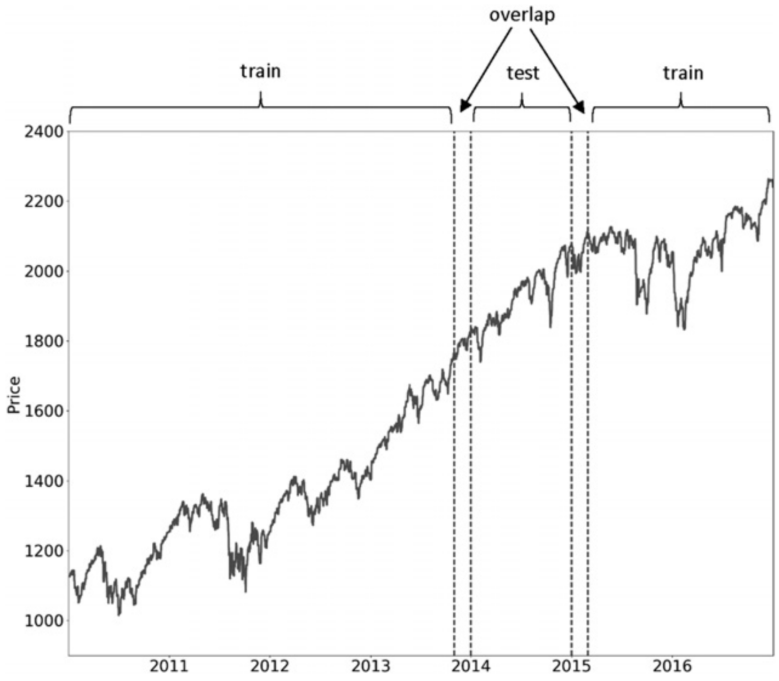


Fig. 1: Image showing the process of **purging**. Figure taken from page 107 of the book.

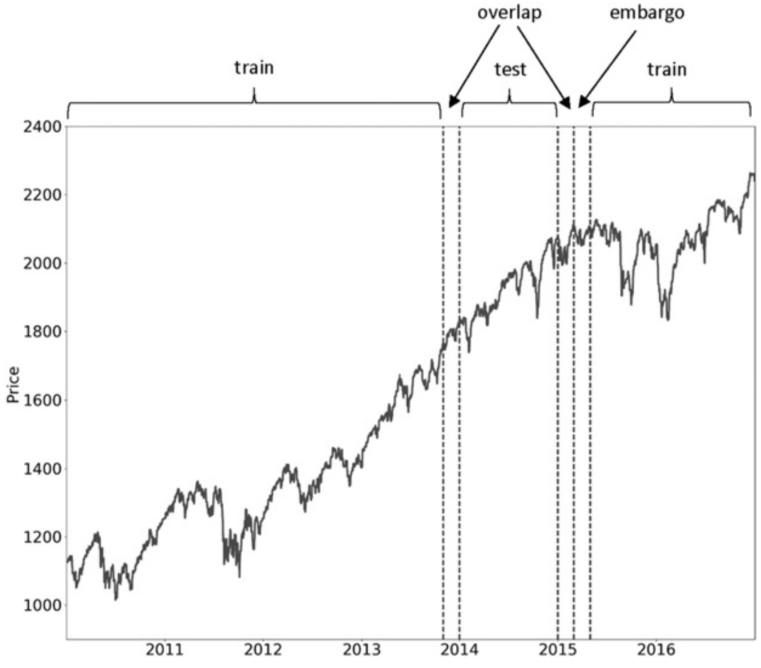


Fig. 2: Image showing the process of **embargo**. Figure taken from page 108 of the book.

class PurgedKFold(*n_splits: int = 3*, *samples_info_sets: pandas.core.series.Series = None*, *pct_embargo: float = 0.0*)

Extend KFold class to work with labels that span intervals The train is purged of observations overlapping test-label intervals Test set is assumed contiguous (shuffle=False), w/o training samples in between

Parameters

- **n_splits** – The number of splits. Default to 3
- **samples_info_sets** – The information range on which each record is constructed from *samples_info_sets.index*: Time when the information extraction started. *samples_info_sets.value*: Time when the information extraction ended.
- **pct_embargo** – Percent that determines the embargo size.

split(*X: pandas.core.frame.DataFrame*, *y: pandas.core.series.Series = None*, *groups=None*)

The main method to call for the PurgedKFold class

Parameters

- **X** – The pd.DataFrame samples dataset that is to be split
- **y** – The pd.Series sample labels series
- **groups** – array-like, with shape (n_samples,), optional Group labels for the samples used while splitting the dataset into train/test set.

Returns This method yields uples of (train, test) where train and test are lists of sample indices

ml_cross_val_score(*classifier: sklearn.base.ClassifierMixin*, *X: pandas.core.frame.DataFrame*, *y: pandas.core.series.Series*, *cv_gen: sklearn.model_selection._split.BaseCrossValidator*, *sample_weight: numpy.ndarray = None*, *scoring: str = 'neg_log_loss'*)

Snippet 7.4, page 110, Using the PurgedKFold Class. Function to run a cross-validation evaluation of the using sample weights and a custom CV generator.

Note: This function is different to the book in that it requires the user to pass through a CV object. The book will accept a None value as a default and then resort to using PurgedCV, this also meant that extra arguments had to be passed to the function. To correct this we have removed the default and require the user to pass a CV object to the function.

Example:

```
cv_gen = PurgedKFold(n_splits=n_splits, samples_info_sets=samples_info_sets, pct_
↪embargo=pct_embargo)
scores_array = ml_cross_val_score(classifier, X, y, cv_gen, sample_weight=None,␣
↪scoring='neg_log_loss')
```

Parameters

- **classifier** – A sk-learn Classifier object instance.
- **X** – The dataset of records to evaluate.
- **y** – The labels corresponding to the X dataset.
- **cv_gen** – Cross Validation generator object instance.
- **sample_weight** – A numpy array of weights for each record in the dataset.
- **scoring** – A metric name to use for scoring; currently supports *neg_log_loss*, *accuracy*, *f1*, *precision*, *recall*, and *roc_auc*.

Returns The computed score as a numpy array.

ml_get_train_times(*samples_info_sets: pandas.core.series.Series*, *test_times: pandas.core.series.Series*) → pandas.core.series.Series

Snippet 7.1, page 106, Purging observations in the training set

This function find the training set indexes given the information on which each record is based and the range for the test set. Given test_times, find the times of the training observations.

Parameters

- **samples_info_sets** – The information range on which each record is constructed from *samples_info_sets.index*: Time when the information extraction started. *samples_info_sets.value*: Time when the information extraction ended.
- **test_times** – Times for the test dataset.

4.7 Sequentially Bootstrapped Bagging Classifier/Regressor

In sampling section we have shown that sampling should be done by Sequential Bootstrapping. SequentiallyBootstrappedBaggingClassifier and SequentiallyBootstrappedBaggingRegressor extend sklearn's BaggingClassifier/Regressor by using Sequential Bootstrapping instead of random sampling.

In order to build indicator matrix we need Triple Barrier Events (samples_info_sets) and price bars used to label training data set. That is why samples_info_sets and price bars are input parameters for classifier/regressor.

Implementation of Sequentially Bootstrapped Bagging Classifier using sklearn's library as base class

class SequentiallyBootstrappedBaggingClassifier(*samples_info_sets*, *price_bars*, *base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap_features=False*, *oob_score=False*, *warm_start=False*, *n_jobs=None*, *random_state=None*, *verbose=0*)

A Sequentially Bootstrapped Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset generated using Sequential Bootstrapping sampling procedure and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

Parameters

- **samples_info_sets** – pd.Series, The information range on which each record is constructed from *samples_info_sets.index*: Time when the information extraction started. *samples_info_sets.value*: Time when the information extraction ended.
- **price_bars** – pd.DataFrame Price bars used in samples_info_sets generation
- **base_estimator** – object or None, optional (default=None) The base estimator to fit on random subsets of the dataset. If None, then the base estimator is a decision tree.
- **n_estimators** – int, optional (default=10) The number of base estimators in the ensemble.
- **max_samples** – int or float, optional (default=1.0) The number of samples to draw from X to train each base estimator. If int, then draw *max_samples* samples. If float, then draw *max_samples * X.shape[0]* samples.

- **max_features** – int or float, optional (default=1.0) The number of features to draw from X to train each base estimator. If int, then draw *max_features* features. If float, then draw *max_features * X.shape[1]* features.
- **bootstrap_features** – boolean, optional (default=False) Whether features are drawn with replacement.
- **oob_score** – bool, optional (default=False) Whether to use out-of-bag samples to estimate the generalization error.
- **warm_start** – bool, optional (default=False) When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new ensemble.
- **n_jobs** – int or None, optional (default=None) The number of jobs to run in parallel for both *fit* and *predict*. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.
- **random_state** – int, RandomState instance or None, optional (default=None) If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.
- **verbose** – int, optional (default=0) Controls the verbosity when fitting and predicting.

Variables

- **base_estimator** – estimator The base estimator from which the ensemble is grown.
- **estimators** – list of estimators The collection of fitted base estimators.
- **estimators_samples** – list of arrays The subset of drawn samples (i.e., the in-bag samples) for each base estimator. Each subset is defined by an array of the indices selected.
- **estimators_features** – list of arrays The subset of drawn features for each base estimator.
- **classes** – array of shape = [n_classes] The classes labels.
- **n_classes** – int or list The number of classes.
- **oob_score** – float Score of the training dataset obtained using an out-of-bag estimate.
- **oob_decision_function** – array of shape = [n_samples, n_classes] Decision function computed with out-of-bag estimate on the training set. If n_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, *oob_decision_function_* might contain NaN.

class SequentiallyBootstrappedBaggingRegressor(*samples_info_sets*, *price_bars*, *base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap_features=False*, *oob_score=False*, *warm_start=False*, *n_jobs=None*, *random_state=None*, *verbose=0*)

A Sequentially Bootstrapped Bagging regressor is an ensemble meta-estimator that fits base regressors each on random subsets of the original dataset using Sequential Bootstrapping and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

Parameters

- **samples_info_sets** – pd.Series, The information range on which each record is constructed from *samples_info_sets.index*: Time when the information extraction started. *samples_info_sets.value*: Time when the information extraction ended.
- **price_bars** – pd.DataFrame Price bars used in samples_info_sets generation
- **base_estimator** – object or None, optional (default=None) The base estimator to fit on random subsets of the dataset. If None, then the base estimator is a decision tree.
- **n_estimators** – int, optional (default=10) The number of base estimators in the ensemble.
- **max_samples** – int or float, optional (default=1.0) The number of samples to draw from X to train each base estimator. If int, then draw *max_samples* samples. If float, then draw *max_samples * X.shape[0]* samples.
- **max_features** – int or float, optional (default=1.0) The number of features to draw from X to train each base estimator. If int, then draw *max_features* features. If float, then draw *max_features * X.shape[1]* features.
- **bootstrap_features** – boolean, optional (default=False) Whether features are drawn with replacement.
- **oob_score** – bool Whether to use out-of-bag samples to estimate the generalization error.
- **warm_start** – bool, optional (default=False) When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new ensemble.
- **n_jobs** – int or None, optional (default=None) The number of jobs to run in parallel for both *fit* and *predict*. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.
- **random_state** – int, RandomState instance or None, optional (default=None) If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.
- **verbose** – int, optional (default=0) Controls the verbosity when fitting and predicting.

Variables

- **estimators** – list of estimators The collection of fitted sub-estimators.
- **estimators_samples** – list of arrays The subset of drawn samples (i.e., the in-bag samples) for each base estimator. Each subset is defined by an array of the indices selected.
- **estimators_features** – list of arrays The subset of drawn features for each base estimator.
- **oob_score** – float Score of the training dataset obtained using an out-of-bag estimate.
- **oob_prediction** – array of shape = [n_samples] Prediction computed with out-of-bag estimate on the training set. If n_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, *oob_prediction_* might contain NaN.

An example of using SequentiallyBootstrappedBaggingClassifier

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from mlfinlab.ensemble import SequentiallyBootstrappedBaggingClassifier

X = pd.read_csv('X_FILE_PATH', index_col=0, parse_dates = [0])
```

(continues on next page)

(continued from previous page)

```
y = pd.read_csv('y_FILE_PATH', index_col=0, parse_dates = [0])
triple_barrier_events = pd.read_csv('BARRIER_FILE_PATH', index_col=0, parse_dates =␣
↪[0, 2])
price_bars = pd.read_csv('PRICE_BARS_FILE_PATH', index_col=0, parse_dates = [0, 2])

triple_barrier_events = triple_barrier_events.loc[X.index, :] # take only train part
price_events = price_events[(price_events.index >= X.index.min()) & (price_events.
↪index <= X.index.max())]

base_est = RandomForestClassifier(n_estimators=1, criterion='entropy',␣
↪bootstrap=False,
                                  class_weight='balanced_subsample')
clf = SequentiallyBootstrappedBaggingClassifier(base_estimator=base_est, samples_info_
↪sets=triple_barrier_events.t1,
                                                price_bars=price_bars, oob_score=True)
clf.fit(X, y)
```

4.8 Feature Importance

One of the key research principles of Advances in Financial Machine learning is:

> **Backtesting is not a research tool. Feature importance is.**

There are three ways to get feature importance scores:

1) Mean Decrease Impurity (MDI). This score can be obtained from tree-based classifiers and corresponds to sklearn's feature_importances attribute. MDI uses in-sample (IS) performance to estimate feature importance.
2) Mean Decrease Accuracy (MDA). This method can be applied to any tree-based classifier, not only tree based. MDA uses out-of-sample (OOS) performance in order to estimate feature importance.
3) Single Feature Importance (SFI). MDA and MDI feature suffer from substitution effects: if two features are highly correlated, one of them will be considered as important while the other one will be redundant. SFI is OOS feature importance estimator which doesn't suffer from substitution effect because it estimates each feature importance separately.

4.8.1 MDI, MDA, SFI feature importance

Module which implements feature importance algorithms described in Chapter 8

feature_importance_mean_decrease_accuracy(*clf*, *X*, *y*, *cv_gen*, *sample_weight=None*, *scoring='neg_log_loss'*)

Snippet 8.3, page 116-117. MDA Feature Importance

This function uses OOS performance to generate Mean Decrease Accuracy importance

Parameters

- **clf** – (sklearn.ClassifierMixin): any sklearn classifier
- **X** – (pd.DataFrame): train set features
- **y** – (pd.DataFrame, np.array): train set labels
- **cv_gen** – (cross_validation.PurgedKFold): cross-validation object
- **sample_weight** – (np.array): sample weights, if None equal to ones

- **scoring** – (str): scoring function used to determine importance, either 'neg_log_loss' or 'accuracy'

Returns (pd.DataFrame): mean and std feature importance

feature_importance_mean_decrease_impurity(*clf*, *feature_names*)
Snippet 8.2, page 115. MDI Feature importance

This function generates feature importance from classifiers estimators importance using Mean Impurity Reduction (MDI) algorithm

Parameters

- **clf** – (BaggingClassifier, RandomForest or any ensemble sklearn object): trained classifier
- **feature_names** – (list): array of feature names

Returns (pd.DataFrame): individual MDI feature importance

feature_importance_sfi(*clf*, *X*, *y*, *cv_gen*, *sample_weight=None*, *scoring='neg_log_loss'*)
Snippet 8.4, page 118. Implementation of SFI

This function generates Single Feature Importance based on OOS score (using cross-validation object)

Parameters

- **clf** – (sklearn.ClassifierMixin): any sklearn classifier
- **X** – (pd.DataFrame): train set features
- **y** – (pd.DataFrame, np.array): train set labels
- **cv_gen** – (cross_validation.PurgedKFold): cross-validation object
- **sample_weight** – (np.array): sample weights, if None equal to ones
- **scoring** – (str): scoring function used to determine importance

Returns (pd.DataFrame): mean and std feature importance

plot_feature_importance(*imp*, *oob_score*, *oos_score*, *savefig=False*, *output_path=None*)
Snippet 8.10, page 124. Feature importance plotting function

Plot feature importance function :param imp: (pd.DataFrame): mean and std feature importance :param oob_score: (float): out-of-bag score :param oos_score: (float): out-of-sample (or cross-validation) score :param savefig: (bool): boolean flag to save figure to a file :param output_path: (str): if savefig is True, path where figure should be saved :return: None

An example showing how to use various feature importance functions:

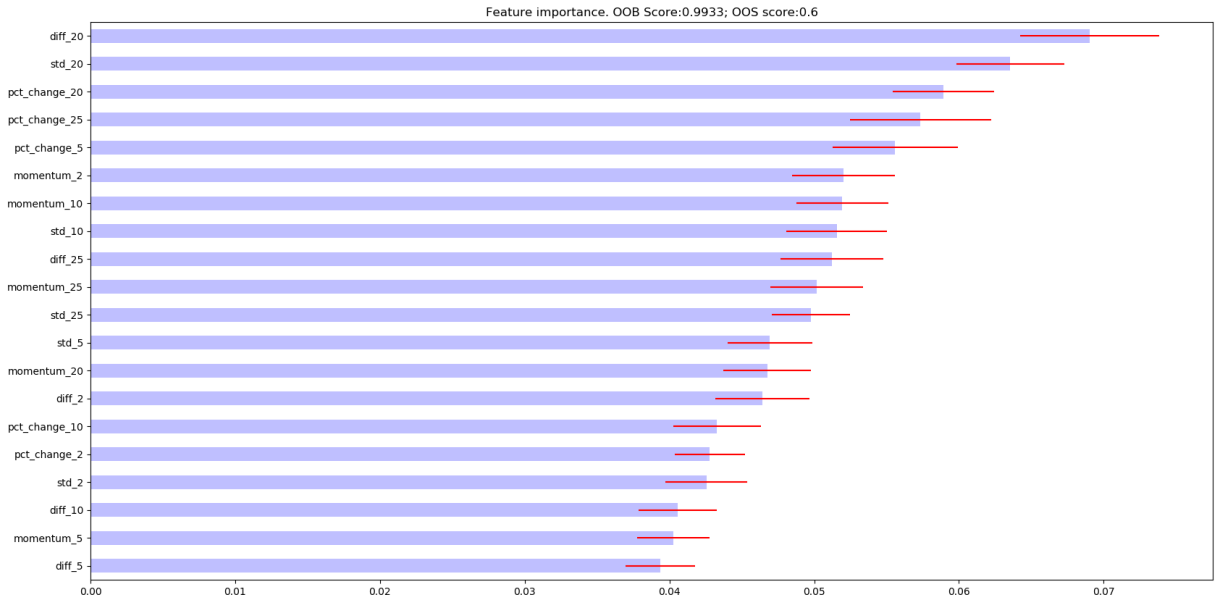
```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from mlfinlab.ensemble import SequentiallyBootstrappedBaggingClassifier
      from mlfinlab.feature_importance import feature_importance_mean_imp_reduction,␣
↪feature_importance_mean_decrease_accuracy, feature_importance_sfi, plot_feature_
↪importance
from mlfinlab.cross_validation import PurgedKFold, ml_cross_val_score
from mlfinlab.ensemble import SequentiallyBootstrappedBaggingClassifier


X_train = pd.read_csv('X_FILE_PATH', index_col=0, parse_dates = [0])
y_train = pd.read_csv('y_FILE_PATH', index_col=0, parse_dates = [0])
triple_barrier_events = pd.read_csv('BARRIER_FILE_PATH', index_col=0, parse_dates =␣
↪[0, 2])
price_bars = pd.read_csv('PRICE_BARS_FILE_PATH', index_col=0, parse_dates = [0, 2])
```

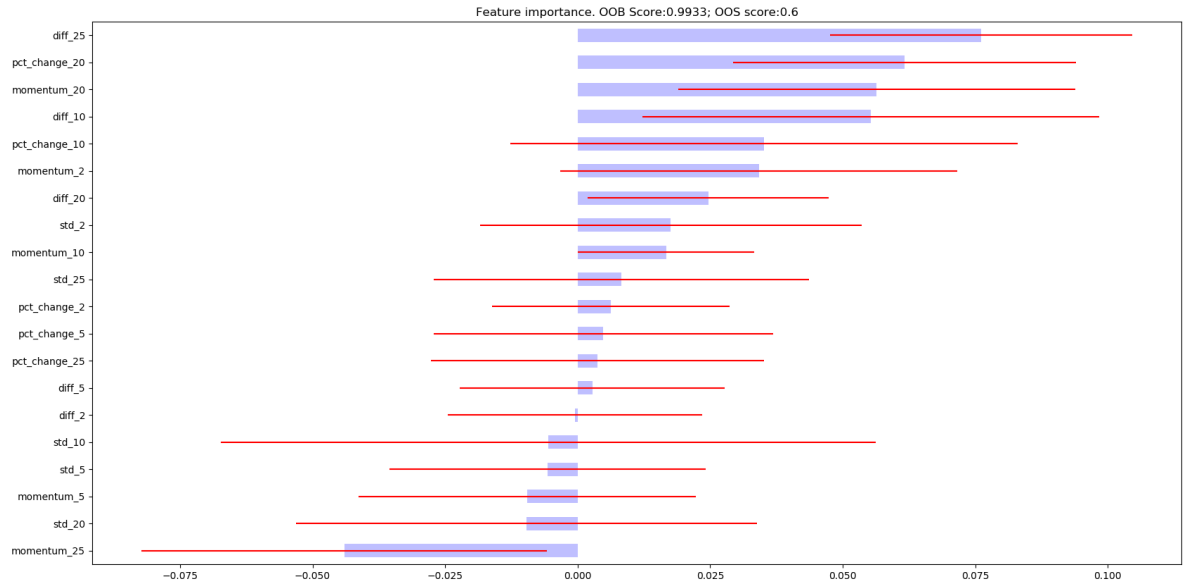
(continues on next page)

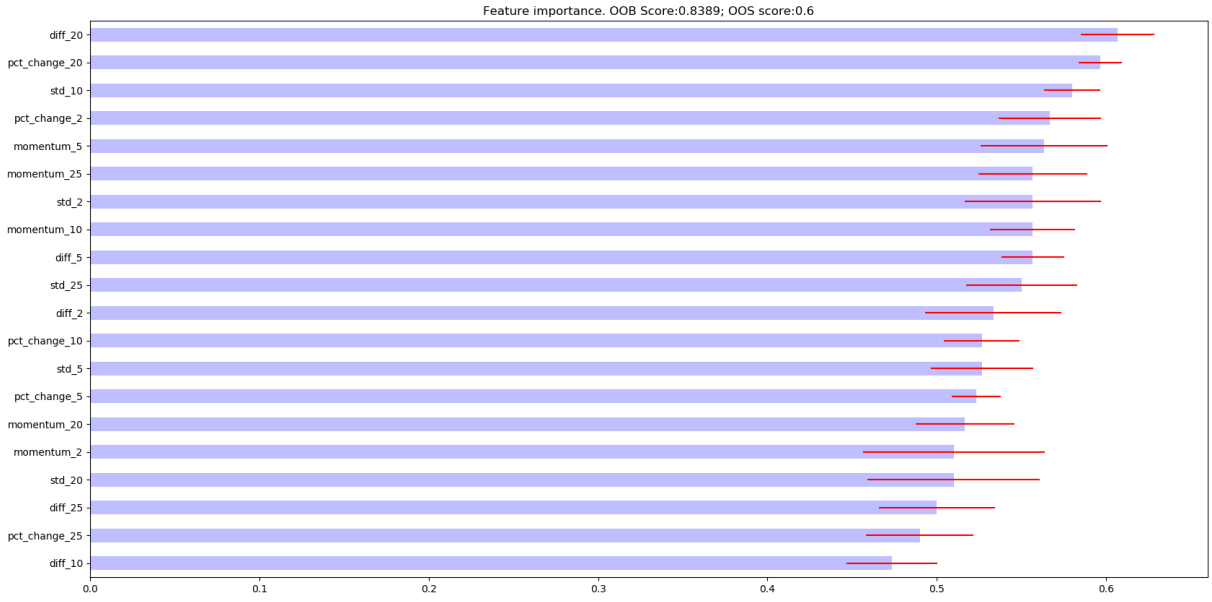
(continued from previous page)

```
triple_barrier_events = triple_barrier_events.loc[X.index, :] # take only train part
price_events = price_events[(price_events.index >= X.index.min()) & (price_events.
↪index <= X.index.max())]

cv_gen = PurgedKFold(n_splits=4, samples_info_sets=triple_barrier_events.t1)

base_est = RandomForestClassifier(n_estimators=1, criterion='entropy',␣
↪bootstrap=False,
                                  class_weight='balanced_subsample')
clf = SequentiallyBootstrappedBaggingClassifier(base_estimator=base_est, samples_info_
↪sets=triple_barrier_events.t1,
                                                price_bars=price_bars, oob_score=True)
clf.fit(X_train, y_train)

oos_score = ml_cross_val_score(sclf, X_train, y_train, cv_gen=cv_gen, sample_
↪weight=None,
                                    scoring='accuracy').mean()

mdi_feature_imp = feature_importance_mean_imp_reduction(clf, X_train.columns)
mda_feature_imp = feature_importance_mean_decrease_accuracy(clf, X_train, y_train, cv_
↪gen, scoring='neg_log_loss')
sfi_feature_imp = feature_importance_sfi(clf, X_train, y_train, cv_gen, scoring=
↪'accuracy')

plot_feature_importance(mdi_feat_imp, oob_score=clf.oob_score_, oos_score=oos_score,
                            savefig=True, output_path='mdi_feat_imp.png')
plot_feature_importance(mda_feat_imp, oob_score=clf.oob_score_, oos_score=oos_score,
                            savefig=True, output_path='mda_feat_imp.png')
plot_feature_importance(sfi_feat_imp, oob_score=clf.oob_score_, oos_score=oos_score,
                            savefig=True, output_path='mdi_feat_imp.png')
```

Resulting images for MDI, MDA, SFI feature importances respectively:

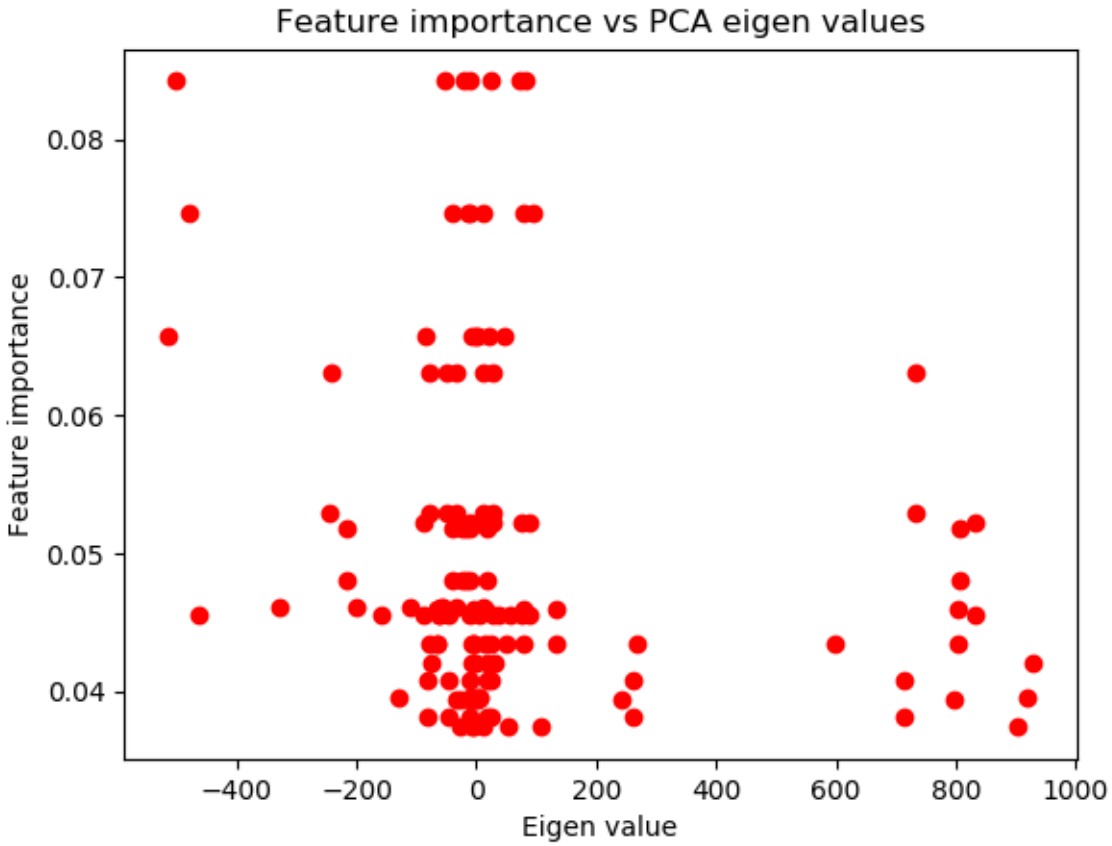






4.8.2 PCA features and analysis

Partial solution to solve substitution effects is to orthogonalize features - apply PCA to them. However, PCA can be used not only to reduce the dimension of your data set, but also to understand whether the patterns detected by feature importance are valid. Suppose, that you derive orthogonal features using PCA. Your PCA analysis has determined that some features are more 'principal' than others, without any knowledge of the labels (unsupervised learning). That is, PCA has ranked features without any possible overfitting in a classification sense. When your MDI, MDA, SFI analysis selects as most important (using label information) the same features that PCA chose as principal (ignoring label information), this constitutes confirmatory evidence that the pattern identified by the ML algorithm is not entirely overfit. Here is the example plot of MDI feature importance vs PCA eigen values:



Module which implements feature PCA compression and PCA analysis of feature importance

get_orthogonal_features (*feature_df*, *variance_thresh=0.95*)
Snippet 8.5, page 119. Computation of Orthogonal Features.

Get PCA orthogonal features

Parameters

- **feature_df** – (pd.DataFrame): with features
- **variance_thresh** – (float): % of overall variance which compressed vectors should explain

Returns (pd.DataFrame): compressed PCA features which explain %variance_thresh of variance

get_pca_rank_weighted_kendall_tau (*feature_imp*, *pca_rank*)
Snippet 8.6, page 121. Computation of Weighted Kendall's Tau Between Feature Importance and Inverse PCA Ranking

Parameters

- **feature_imp** – (np.array): with feature mean importance
- **pca_rank** – (np.array): PCA based feature importance rank

Returns (float): weighted Kendall tau of feature importance and inverse PCA rank with p_value

feature_pca_analysis (*feature_df*, *feature_importance*, *variance_thresh=0.95*)
Perform correlation analysis between feature importance (MDI for example, supervised) and PCA eigen values

(unsupervised). High correlation means that probably the pattern identified by the ML algorithm is not entirely overfit.

Parameters

- **feature_df** – (pd.DataFrame): with features
- **feature_importance** – (pd.DataFrame): individual MDI feature importance
- **variance_thresh** – (float): % of overall variance which compressed vectors should explain in PCA compression

Returns (dict): with kendall, spearman, pearson and weighted_kendall correlations and p_values

Let's see how PCA feature extraction is analysis are done using mlfinlab functions:

```
import pandas as pd
from mlfinlab.feature_importance.orthogonal import get_orthogonal_features, feature_
↪pca_analysis

X_train = pd.read_csv('X_FILE_PATH', index_col=0, parse_dates = [0])
feat_imp = pd.read_csv('FEATURE_IMP_PATH')

pca_features = get_orthogonal_features(X_train)
correlation_dict = feature_pca_analysis(X_train, feat_imp)
```

4.9 Bet Sizing

"There are fascinating parallels between strategy games and investing. Some of the best portfolio managers I have worked with are excellent poker players, perhaps more so than chess players. One reason is bet sizing, for which Texas Hold'em provides a great analogue and training ground. Your ML algorithm can achieve high accuracy, but if you do not size your bets properly, your investment strategy will inevitably lose money. In this chapter we will review a few approaches to size bets from ML predictions." Advances in Financial Machine Learning, Chapter 10: Bet Sizing, pg 141.

The code in this directory falls under 3 submodules:

1. Bet Sizing: We have extended the code from the book in an easy to use format for practitioners to use going forward.
2. EF3M: An implementation of the EF3M algorithm.
3. Chapter10_Snippets: Documented and adjusted snippets from the book for users to experiment with.

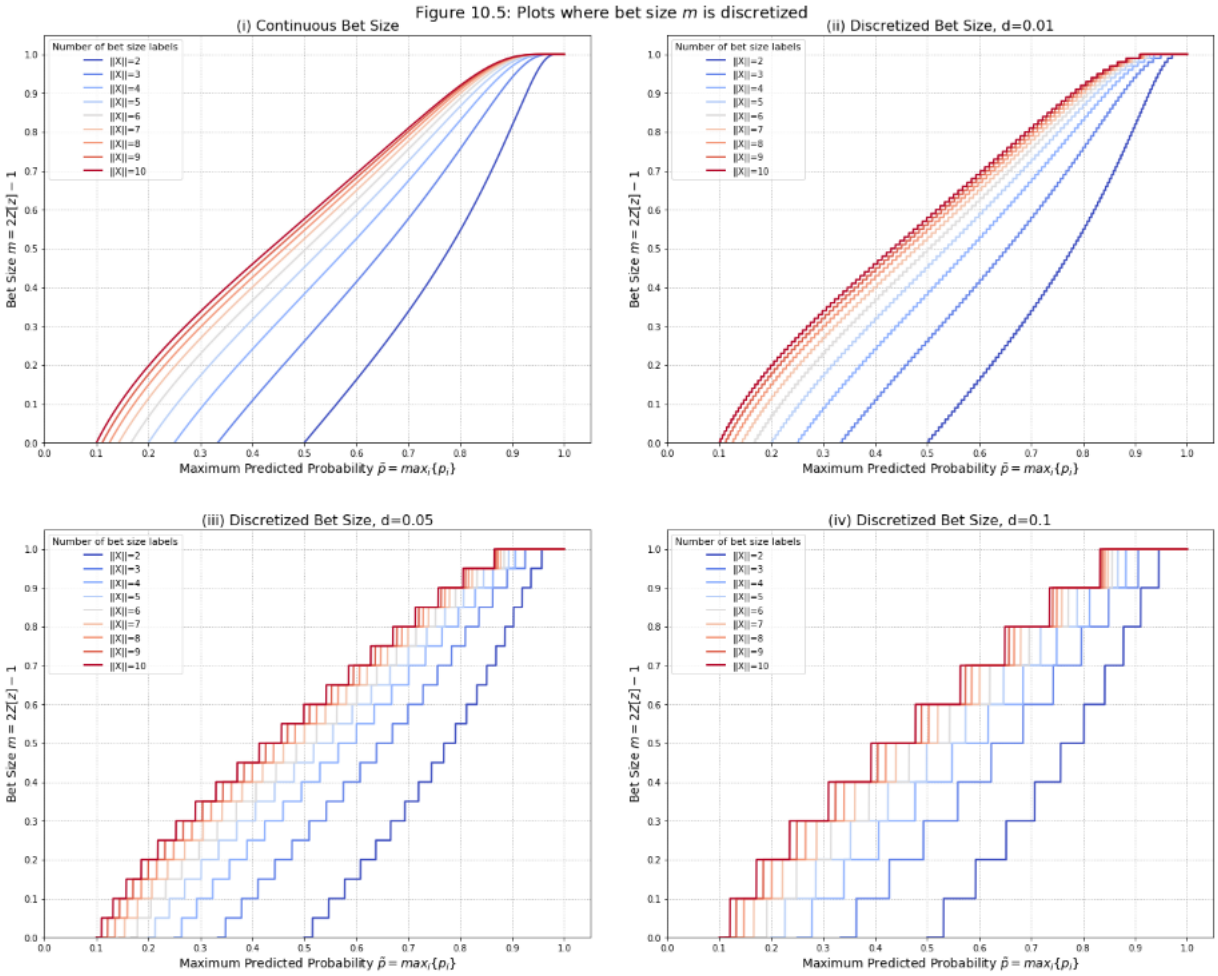


Figure 10.5: Plots where bet size m is discretized

4.9.1 Bet Sizing Methods

Functions for bet sizing are implemented based on the approaches described in chapter 10.

Bet Sizing From Predicted Probability

Assuming a machine learning algorithm has predicted a series of investment positions, one can use the probabilities of each of these predictions to derive the size of that specific bet.

bet_size_probability(*events*, *prob*, *num_classes*, *pred=None*, *step_size=0.0*, *average_active=False*, *num_threads=1*)

Calculates the bet size using the predicted probability. Note that if 'average_active' is True, the returned pandas.Series will be twice the length of the original since the average is calculated at each bet's open and close.

Parameters

- **events** – (pandas.DataFrame) Contains at least the column 't1', the expiry datetime of the product, with a datetime index, the datetime the position was taken.
- **prob** – (pandas.Series) The predicted probability.
- **num_classes** – (int) The number of predicted bet sides.

- **pred** – (pd.Series) The predicted bet side. Default value is None which will return a relative bet size (i.e. without multiplying by the side).
- **step_size** – (float) The step size at which the bet size is discretized, default is 0.0 which imposes no discretization.
- **average_active** – (bool) Option to average the size of active bets, default value is False.
- **num_threads** – (int) The number of processing threads to utilize for multiprocessing, default value is 1.

Returns (pandas.Series) The bet size, with the time index.

Dynamic Bet Sizes

Assuming one has a series of forecasted prices for a given investment product, that forecast and the current market price and position can be used to dynamically calculate the bet size.

bet_size_dynamic(*current_pos*, *max_pos*, *market_price*, *forecast_price*, *cal_divergence=10*, *cal_bet_size=0.95*, *func='sigmoid'*)

Calculates the bet sizes, target position, and limit price as the market price and forecast price fluctuate. The current position, maximum position, market price, and forecast price can be passed as separate pandas.Series (with a common index), as individual numbers, or a combination thereof. If any one of the aforementioned arguments is a pandas.Series, the other arguments will be broadcast to a pandas.Series of the same length and index.

Parameters

- **current_pos** – (pandas.Series, int) Current position.
- **max_pos** – (pandas.Series, int) Maximum position
- **market_price** – (pandas.Series, float) Market price.
- **forecast_price** – (pandas.Series, float) Forecast price.
- **cal_divergence** – (float) The divergence to use in calibration.
- **cal_bet_size** – (float) The bet size to use in calibration.
- **func** – (string) Function to use for dynamic calculation. Valid options are: 'sigmoid', 'power'.

Returns (pandas.DataFrame) Bet size (bet_size), target position (t_pos), and limit price (l_p).

Strategy-Independent Bet Sizing Approaches

These approaches consider the number of concurrent active bets and their sides, and sets the bet size is such a way that reserves some cash for the possibility that the trading signal strengthens before it weakens.

bet_size_budget(*events_t1*, *sides*)

Calculates a bet size from the bet sides and start and end times. These sequences are used to determine the number of concurrent long and short bets, and the resulting strategy-independent bet sizes are the difference between the average long and short bets at any given time. This strategy is based on the section 10.2 in "Advances in Financial Machine Learning". This creates a linear bet sizing scheme that is aligned to the expected number of concurrent bets in the dataset.

Parameters

- **events_t1** – (pandas.Series) The end datetime of the position with the start datetime as the index.

- **sides** – (pandas.Series) The side of the bet with the start datetime as index. Index must match the 'events_t1' argument exactly. Bet sides less than zero are interpretted as short, bet sides greater than zero are interpretted as long.

Returns (pandas.DataFrame) The 'events_t1' and 'sides' arguments as columns, with the number of concurrent active long and short bets, as well as the bet size, in additional columns.

bet_size_reserve(*events_t1*, *sides*, *fit_runs=100*, *epsilon=1e-05*, *factor=5*, *variant=2*, *max_iter=10000*, *num_workers=1*, *return_parameters=False*)

Calculates the bet size from bet sides and start and end times. These sequences are used to determine the number of concurrent long and short bets, and the difference between the two at each time step, c_t. A mixture of two Gaussian distributions is fit to the distribution of c_t, which is then used to determine the bet size. This strategy results in a sigmoid-shaped bet sizing response aligned to the expected number of concurrent long and short bets in the dataset.

Note that this function creates a <mlfinlab.bet_sizing.ef3m.M2N> object and makes use of the parallel fitting functionality. As such, this function accepts and passes fitting parameters to the mlfinlab.bet_sizing.ef3m.M2N.mp_fit() method.

Parameters

- **events_t1** – (pandas.Series) The end datetime of the position with the start datetime as the index.
- **sides** – (pandas.Series) The side of the bet with the start datetime as index. Index must match the 'events_t1' argument exactly. Bet sides less than zero are interpretted as short, bet sides greater than zero are interpretted as long.
- **fit_runs** – (int) Number of runs to execute when trying to fit the distribution.
- **epsilon** – (float) Error tolerance.
- **factor** – (float) Lambda factor from equations.
- **variant** – (int) Which algorithm variant to use, 1 or 2.
- **max_iter** – (int) Maximum number of iterations after which to terminate loop.
- **num_workers** – (int) Number of CPU cores to use for multiprocessing execution, set to -1 to use all CPU cores. Default is 1.
- **return_parameters** – (bool) If True, function also returns a dictionary of the fited mixture parameters.

Returns (pandas.DataFrame) The 'events_t1' and 'sides' arguments as columns, with the number of concurrent active long, short bets, the difference between long and short, and the bet size in additional columns. Also returns the mixture parameters if 'return_parameters' is set to True.

Additional Utility Functions For Bet Sizing

confirm_and_cast_to_df(*d_vars*)

Accepts either pandas.Series (with a common index) or integer/float values, casts all non-pandas.Series values to Series, and returns a pandas.DataFrame for further calculations. This is a helper function to the 'bet_size_dynamic' function.

Parameters **d_vars** – (dict) A dictionary where the values are either pandas.Series or single int/float values. All pandas.Series passed are assumed to have the same index. The keys of the dictionary will be used for column names in the returned pandas.DataFrame.

Returns (pandas.DataFrame) The values from the input dictionary in pandas.DataFrame format, with dictionary keys as column names.

get_concurrent_sides(*events_t1*, *sides*)

Given the side of the position along with its start and end timestamps, this function returns two pandas.Series indicating the number of concurrent long and short bets at each timestamp.

Parameters

- **events_t1** – (pandas.Series) The end datetime of the position with the start datetime as the index.
- **sides** – (pandas.Series) The side of the bet with the start datetime as index. Index must match the 'events_t1' argument exactly. Bet sides less than zero are interpretted as short, bet sides greater than zero are interpretted as long.

Returns (pandas.DataFrame) The 'events_t1' and 'sides' arguments as columns, with two additional columns indicating the number of concurrent active long and active short bets at each timestamp.

cdf_mixture(*x_val*, *parameters*)

The cumulative distribution function of a mixture of 2 normal distributions, evaluated at x_val.

Parameters

- **x_val** – (float) Value at which to evaluate the CDF.
- **parameters** – (list) The parameters of the mixture, [mu_1, mu_2, sigma_1, sigma_2, p_1]

Returns (float) CDF of the mixture.

single_bet_size_mixed(*c_t*, *parameters*)

Returns the single bet size based on the description provided in question 10.4(c), provided the difference in concurrent long and short positions, c_t, and the fitted parameters of the mixture of two Gaussain distributions.

Parameters

- **c_t** – (int) The difference in the number of concurrent long bets minus short bets.
- **parameters** – (list) The parameters of the mixture, [mu_1, mu_2, sigma_1, sigma_2, p_1]

Returns (float) Bet size.

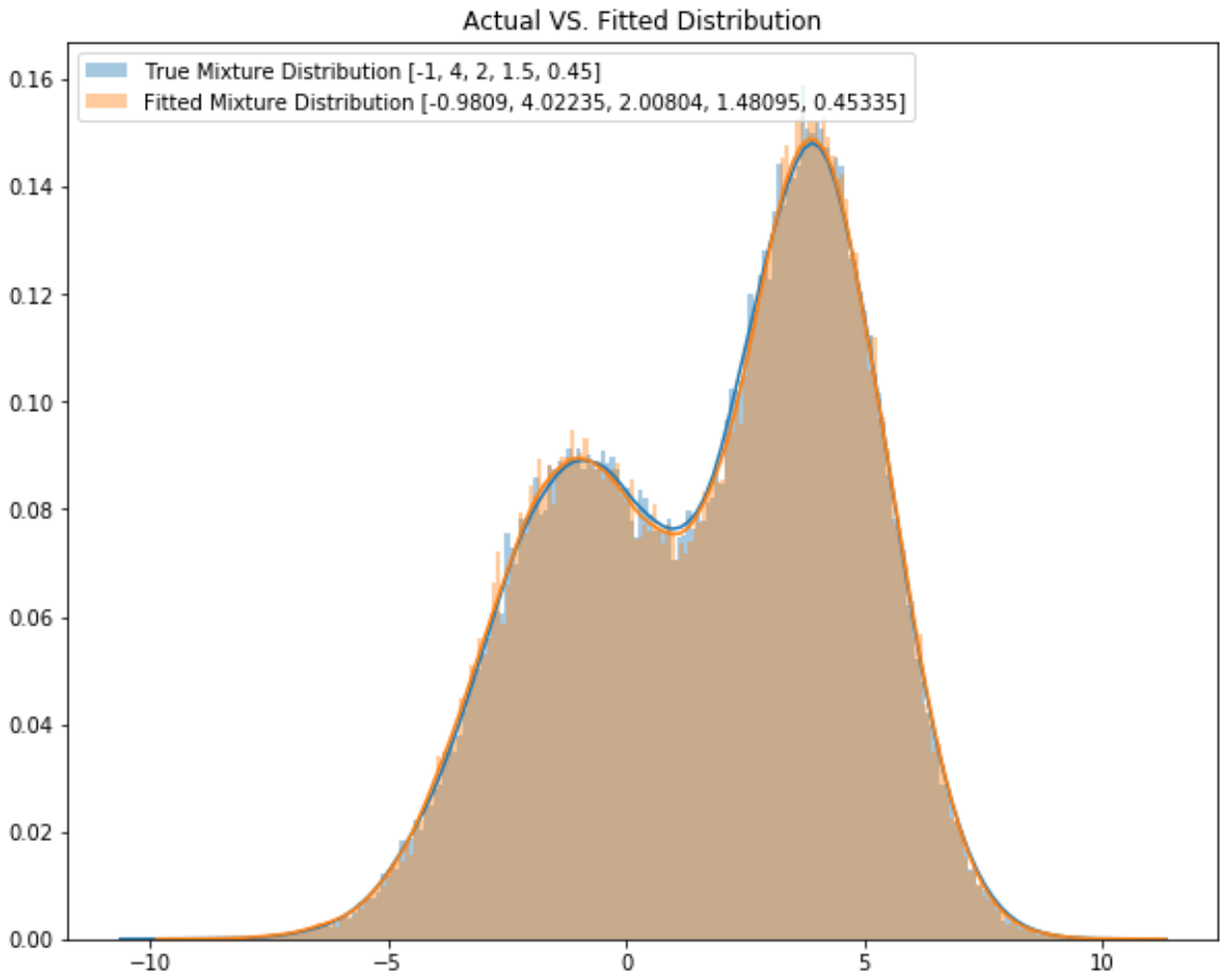
4.9.2 EF3M - Exact Fit using the first 3 Moments

The EF3M algorithm was introduced in a paper by Marcos Lopez de Prado and Matthew D. Foreman, titled "A mixture of Gaussians approach to mathematical portfolio oversight: the EF3M algorithm".

The abstract reads: "An analogue can be made between: (a) the slow pace at which species adapt to an environment, which often results in the emergence of a new distinct species out of a once homogeneous genetic pool, and (b) the slow changes that take place over time within a fund, mutating its investment style. A fund's track record provides a sort of genetic marker, which we can use to identify mutations. This has motivated our use of a biometric procedure to detect the emergence of a new investment style within a fund's track record. In doing so, we answer the question: "What is the probability that a particular PM's performance is departing from the reference distribution used to allocate her capital?" The EF3M approach, inspired by evolutionary biology, may help detect early stages of an evolutionary divergence in an investment style, and trigger a decision to review a fund's capital allocation."

The Exact Fit of the first 3 Moments (EF3M) algorithm allows the parameters of a mixture of Gaussian distributions to be estimated given the first 5 moments of the mixture distribution, as well as the assumption that the mixture distribution is composed of a number of Gaussian distributions.

A more thorough investigation into the algorithm can be found within our Research repository



M2N

A class for determining the means, standard deviations, and mixture proportion of a given distribution from it's first four or five statistical moments.

class M2N (*moments*, *epsilon=1e-05*, *factor=5*, *n_runs=1*, *variant=1*, *max_iter=100000*, *num_workers=-1*)

M2N - A Mixture of 2 Normal distributions This class is used to contain parameters and equations for the EF3M algorithm, when fitting parameters to a mixture of 2 Gaussian distributions.

Parameters

- **moments** – (list) The first five (1… 5) raw moments of the mixture distribution.
- **epsilon** – (float) Fitting tolerance
- **factor** – (float) Lambda factor from equations
- **n_runs** – (int) Number of times to execute 'singleLoop'
- **variant** – (int) The EF3M variant to execute, options are 1: EF3M using first 4 moments, 2: EF3M using first 5 moments
- **max_iter** – (int) Maximum number of iterations to perform in the 'fit' method
- **num_workers** – (int) Number of CPU cores to use for multiprocessing execution. Default is -1 which sets num_workers to all cores.

fit (*mu_2*)
Fits and the parameters that describe the mixture of the 2 Normal distributions for a given set of initial parameter guesses.

Parameters **mu_2** – (float) An initial estimate for the mean of the second distribution.

get_moments (*parameters*, *return_result=False*)
Calculates and returns the first five (1...5) raw moments corresponding to the newly estimated parameters.

Parameters

- **parameters** – (list) List of parameters if the specific order [mu_1, mu_2, sigma_1, sigma_2, p_1]
- **return_result** – (bool) If True, method returns a result instead of setting the 'self.new_moments' attribute.

Returns (list) List of the first five moments

iter_4 (*mu_2*, *p_1*)
Evaluation of the set of equations that make up variant #1 of the EF3M algorithm (fitting using the first four moments).

Parameters

- **mu_2** – (float) Initial parameter value for mu_2
- **p_1** – (float) Probability defining the mixture; p_1, 1 - p_1

Returns (list) List of estimated parameter if no invalid values are encountered (e.g. complex values, divide-by-zero), otherwise an empty list is returned.

iter_5 (*mu_2*, *p_1*)
Evaluation of the set of equations that make up variant #2 of the EF3M algorithm (fitting using the first five moments).

Parameters

- **mu_2** – (float) Initial parameter value for mu_2
- **p_1** – (float) Probability defining the mixture; p_1, 1-p_1

Returns (list) List of estimated parameter if no invalid values are encountered (e.g. complex values, divide-by-zero), otherwise an empty list is returned.

mp_fit ()
Parallelized implementation of the 'single_fit_loop' method. Makes use of dask.delayed to execute multiple calls of 'single_fit_loop' in parallel.

Returns (pd.DataFrame) Fitted parameters and error

single_fit_loop (*epsilon=0*)
A single scan through the list of mu_2 values, cataloging the successful fittings in a DataFrame.

Parameters **epsilon** – (float) Fitting tolerance.

Returns (pd.DataFrame) Fitted parameters and error

Utility Functions For Fitting Of Distribution Mixtures

centered_moment (*moments*, *order*)
Compute a single moment of a specific order about the mean (centered) given moments about the origin (raw).

Parameters

- **moments** – (list) First 'order' raw moments
- **order** – (int) The order of the moment to calculate

Returns (float) The central moment of specified order.

raw_moment (*central_moments*, *dist_mean*)
Calculates a list of raw moments given a list of central moments.

Parameters

- **central_moments** – (list) The first n (1...n) central moments as a list.
- **dist_mean** – (float) The mean of the distribution.

Returns (list) The first n+1 (0...n) raw moments.

most_likely_parameters (*data*, *ignore_columns='error'*, *res=10000*)
Determines the most likely parameter estimate using a KDE from the DataFrame of the results of the fit from the M2N object.

Parameters

- **data** – (pandas.DataFrame) Contains parameter estimates from all runs.
- **ignore_columns** – (string, list) Column or columns to exclude from analysis.
- **res** – (int) Resolution of the kernel density estimate.

Returns (dict) Labels and most likely estimates for parameters.

4.9.3 Chapter 10 Code Snippets

Chapter 10 of "Advances in Financial Machine Learning" contains a number of Python code snippets, many of which are used to create the top level bet sizing functions. These functions can be found in `mlfinlab.bet_sizing.ch10_snippets.py`.

Snippets For Bet Sizing From Probabilities

get_signal (*prob*, *num_classes*, *pred=None*)
SNIPPET 10.1 - FROM PROBABILITIES TO BET SIZE Calculates the given size of the bet given the side and the probability (i.e. confidence) of the prediction. In this representation, the probability will always be between 1/num_classes and 1.0.

Parameters

- **prob** – (pd.Series) The probability of the predicted bet side.
- **num_classes** – (int) The number of predicted bet sides.
- **pred** – (pd.Series) The predicted bet side. Default value is None which will return a relative bet size (i.e. without multiplying by the side).

Returns (pd.Series) The bet size.

avg_active_signals (*signals*, *num_threads=1*)
SNIPPET 10.2 - BETS ARE AVERAGED AS LONG AS THEY ARE STILL ACTIVE Function averages the bet sizes of all concurrently active bets. This function makes use of multiprocessing.

Parameters

- **signals** – (pandas.DataFrame) Contains at least the following columns: 'signal' - the bet size 't1' - the closing time of the bet And the index must be datetime format.

- **num_threads** – (int) Number of threads to use in multiprocessing, default value is 1.

Returns (pandas.Series) The averaged bet sizes.

mp_avg_active_signals(*signals*, *molecule*)
Part of SNIPPET 10.2 A function to be passed to the 'mp_pandas_obj' function to allow the bet sizes to be averaged using multiprocessing.

At time loc, average signal among those still active. Signal is active if (a) it is issued before or at loc, and (b) loc is before the signal's end time, or end time is still unknown (NaT).

Parameters

- **signals** – (pandas.DataFrame) Contains at least the following columns: 'signal' (the bet size) and 't1' (the closing time of the bet).
- **molecule** – (list) Indivisible tasks to be passed to 'mp_pandas_obj', in this case a list of datetimes.

Returns (pandas.Series) The averaged bet size sub-series.

discrete_signal(*signal0*, *step_size*)
SNIPPET 10.3 - SIZE DISCRETIZATION TO PREVENT OVERTRADING Discretizes the bet size signal based on the step size given.

Parameters

- **signal0** – (pandas.Series) The signal to discretize.
- **step_size** – (float) Step size.

Returns (pandas.Series) The discretized signal.

Snippets for Dynamic Bet Sizing

bet_size_sigmoid(*w_param*, *price_div*)
Part of SNIPPET 10.4 Calculates the bet size from the price divergence and a regulating coefficient. Based on a sigmoid function for a bet size algorithm.

Parameters

- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **price_div** – (float) Price divergence, forecast price - market price.

Returns (float) The bet size.

get_target_pos_sigmoid(*w_param*, *forecast_price*, *market_price*, *max_pos*)
Part of SNIPPET 10.4 Calculates the target position given the forecast price, market price, maximum position size, and a regulating coefficient. Based on a sigmoid function for a bet size algorithm.

Parameters

- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **forecast_price** – (float) Forecast price.
- **market_price** – (float) Market price.
- **max_pos** – (int) Maximum absolute position size.

Returns (int) Target position.

inv_price_sigmoid(*forecast_price*, *w_param*, *m_bet_size*)
Part of SNIPPET 10.4 Calculates the inverse of the bet size with respect to the market price. Based on a sigmoid function for a bet size algorithm.

Parameters

- **forecast_price** – (float) Forecast price.
- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **m_bet_size** – (float) Bet size.

Returns (float) Inverse of bet size with respect to market price.

limit_price_sigmoid(*target_pos*, *pos*, *forecast_price*, *w_param*, *max_pos*)
Part of SNIPPET 10.4 Calculates the limit price. Based on a sigmoid function for a bet size algorithm.

Parameters

- **target_pos** – (int) Target position.
- **pos** – (int) Current position.
- **forecast_price** – (float) Forecast price.
- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **max_pos** – (int) Maximum absolute position size.

Returns (float) Limit price.

get_w_sigmoid(*price_div*, *m_bet_size*)
Part of SNIPPET 10.4 Calculates the inverse of the bet size with respect to the regulating coefficient 'w'. Based on a sigmoid function for a bet size algorithm.

Parameters

- **price_div** – (float) Price divergence, forecast price - market price.
- **m_bet_size** – (float) Bet size.

Returns (float) Inverse of bet size with respect to the regulating coefficient.

bet_size_power(*w_param*, *price_div*)
Derived from SNIPPET 10.4 Calculates the bet size from the price divergence and a regulating coefficient. Based on a power function for a bet size algorithm.

Parameters

- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **price_div** – (float) Price divergence, f - market_price, must be between -1 and 1, inclusive.

Returns (float) The bet size.

get_target_pos_power(*w_param*, *forecast_price*, *market_price*, *max_pos*)
Derived from SNIPPET 10.4 Calculates the target position given the forecast price, market price, maximum position size, and a regulating coefficient. Based on a power function for a bet size algorithm.

Parameters

- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **forecast_price** – (float) Forecast price.
- **market_price** – (float) Market price.

- **max_pos** – (float) Maximum absolute position size.

Returns (float) Target position.

inv_price_power(*forecast_price*, *w_param*, *m_bet_size*)
Derived from SNIPPET 10.4 Calculates the inverse of the bet size with respect to the market price. Based on a power function for a bet size algorithm.

Parameters

- **forecast_price** – (float) Forecast price.
- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **m_bet_size** – (float) Bet size.

Returns (float) Inverse of bet size with respect to market price.

limit_price_power(*target_pos*, *pos*, *forecast_price*, *w_param*, *max_pos*)
Derived from SNIPPET 10.4 Calculates the limit price. Based on a power function for a bet size algorithm.

Parameters

- **target_pos** – (float) Target position.
- **pos** – (float) Current position.
- **forecast_price** – (float) Forecast price.
- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **max_pos** – (float) Maximum absolute position size.

Returns (float) Limit price.

get_w_power(*price_div*, *m_bet_size*)
Derived from SNIPPET 10.4 Calculates the inverse of the bet size with respect to the regulating coefficient 'w'. The 'w' coefficient must be greater than or equal to zero. Based on a power function for a bet size algorithm.

Parameters

- **price_div** – (float) Price divergence, forecast price - market price.
- **m_bet_size** – (float) Bet size.

Returns (float) Inverse of bet size with respect to the regulating coefficient.

bet_size(*w_param*, *price_div*, *func*)
Derived from SNIPPET 10.4 Calculates the bet size from the price divergence and a regulating coefficient. The 'func' argument allows the user to choose between bet sizing functions.

Parameters

- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **price_div** – (float) Price divergence, f - market_price
- **func** – (string) Function to use for dynamic calculation. Valid options are: 'sigmoid', 'power'.

Returns (float) The bet size.

get_target_pos(*w_param*, *forecast_price*, *market_price*, *max_pos*, *func*)
Derived from SNIPPET 10.4 Calculates the target position given the forecast price, market price, maximum position size, and a regulating coefficient. The 'func' argument allows the user to choose between bet sizing functions.

Parameters

- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **forecast_price** – (float) Forecast price.
- **market_price** – (float) Market price.
- **max_pos** – (int) Maximum absolute position size.
- **func** – (string) Function to use for dynamic calculation. Valid options are: 'sigmoid', 'power'.

Returns (int) Target position.

inv_price (*forecast_price*, *w_param*, *m_bet_size*, *func*)
Derived from SNIPPET 10.4 Calculates the inverse of the bet size with respect to the market price. The 'func' argument allows the user to choose between bet sizing functions.

Parameters

- **forecast_price** – (float) Forecast price.
- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **m_bet_size** – (float) Bet size.

Returns (float) Inverse of bet size with respect to market price.

limit_price (*target_pos*, *pos*, *forecast_price*, *w_param*, *max_pos*, *func*)
Derived from SNIPPET 10.4 Calculates the limit price. The 'func' argument allows the user to choose between bet sizing functions.

Parameters

- **target_pos** – (int) Target position.
- **pos** – (int) Current position.
- **forecast_price** – (float) Forecast price.
- **w_param** – (float) Coefficient regulating the width of the bet size function.
- **max_pos** – (int) Maximum absolute position size.
- **func** – (string) Function to use for dynamic calculation. Valid options are: 'sigmoid', 'power'.

Returns (float) Limit price.

get_w (*price_div*, *m_bet_size*, *func*)
Derived from SNIPPET 10.4 Calculates the inverse of the bet size with respect to the regulating coefficient 'w'. The 'func' argument allows the user to choose between bet sizing functions.

Parameters

- **price_div** – (float) Price divergence, forecast price - market price.
- **m_bet_size** – (float) Bet size.
- **func** – (string) Function to use for dynamic calculation. Valid options are: 'sigmoid', 'power'.

Returns (float) Inverse of bet size with respect to the regulating coefficient.

4.9.4 Research Notebooks

The following research notebooks can be used to better understand bet sizing.

Exercises From Chapter 10

- Chapter 10 Exercise Notebook
- EF3M Algorithm Test Cases

4.10 Portfolio Optimisation

The portfolio optimisation module contains some classic algorithms that are used for asset allocation and optimising strategies. We will discuss these algorithms in detail below.

4.10.1 Hierarchical Risk Parity (HRP)

Hierarchical Risk Parity is a novel portfolio optimisation method developed by Marcos Lopez de Prado. The working of the algorithm can be broken down into 3 steps:

1. Based on the expected returns of the assets, they are segregated into clusters via hierarchical tree clustering.
2. Based on these clusters, the covariance matrix of the returns is diagonalised in a quasi manner such that assets within the same cluster are regrouped together.
3. Finally, using an iterative approach, weights are assigned to each cluster recursively. At each node, the weight breaks down into the sub-cluster until all the individual assets are assigned a unique weight.

Although, it is a simple algorithm, HRP has been found to be a very stable algorithm as compared to its older counterparts. This is because, HRP does not involve taking inverse of the covariance matrix matrix which makes it robust to small changes in the covariances of the asset returns.

Implementation

This module implements the HRP algorithm mentioned in the following paper: López de Prado, Marcos, Building Diversified Portfolios that Outperform Out-of-Sample (May 23, 2016). Journal of Portfolio Management, 2016; The code is reproduced with modification from his book: Advances in Financial Machine Learning, Chp-16

class HierarchicalRiskParity

The HRP algorithm is a robust algorithm which tries to overcome the limitations of the CLA algorithm. It has three important steps - hierarchical tree clustering, quasi diagnalisation and recursive bisection. Non-inversion of covariance matrix makes HRP a very stable algorithm and insensitive to small changes in covariances.

__init__()

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

allocate(*asset_prices*, *resample_by='B'*, *use_shrinkage=False*)

Calculate asset allocations using HRP algorithm

Parameters

- **asset_prices** – (pd.Dataframe) a dataframe of historical asset prices (daily close) indexed by date
- **resample_by** – (str) specifies how to resample the prices - weekly, daily, monthly etc.. Defaults to 'B' meaning daily business days which is equivalent to no resampling
- **use_shrinkage** – (Boolean) specifies whether to shrink the covariances

plot_clusters(*assets*)

Plot a dendrogram of the hierarchical clusters

Parameters **assets** – (list) list of asset names in the portfolio

4.10.2 The Critical Line Algorithm (CLA)

This is a robust alternative to the quadratic optimisation used to find mean-variance optimal portfolios. The major difference between classic Mean-Variance and CLA is the type of optimisation problem solved. A typical mean-variance optimisation problem looks something like this:

$$\underset{w}{\text{minimise}} \ \left\{w^T \Sigma w\right\}$$

where, $\sum_i w_i = 1$ and $0 <= w <= 1$. CLA also solves the same problem but with some added constraints - each weight of an asset in the portfolio can have different lower and upper bounds. The optimisation objective still remains the same but the second constraint changes to - $l_i <= w_i <= u_i$. Each weight in the allocation has an upper and a lower bound, which increases the number of constraints to be solved.

The current CLA implementation in the package supports the following solutions:

1. CLA Turning Points
2. Maximum Sharpe Portfolio
3. Minimum Variance Portfolio
4. Efficient Frontier Solution

Implementation

This module implements the famous Critical Line Algorithm for mean-variance portfolio optimisation. It is reproduced with modification from the following paper: D.H. Bailey and M.L. Prado "An Open-Source Implementation of the Critical- Line Algorithm for Portfolio Optimization",Algorithms, 6 (2013), 169-196.

class CLA(*weight_bounds=(0, 1)*, *calculate_returns='mean'*)

CLA is a famous portfolio optimisation algorithm used for calculating the optimal allocation weights for a given portfolio. It solves the optimisation problem with constraints on each weight - lower and upper bounds on the weight value. This class can compute multiple types of solutions - the normal cla solution, minimum variance solution, maximum sharpe solution and finally the solution to the efficient frontier.

__init__(*weight_bounds=(0, 1)*, *calculate_returns='mean'*)

Initialise the storage arrays and some preprocessing.

Parameters

- **weight_bounds** – (tuple) a tuple specifying the lower and upper bound ranges for the portfolio weights
- **calculate_returns** – (str) the method to use for calculation of expected returns. Currently supports "mean" and "exponential"

allocate(*asset_prices*, *solution='cla_turning_points'*, *resample_by='B'*)

Calculate the portfolio asset allocations using the method specified.

Parameters

- **asset_prices** – (pd.Dataframe) a dataframe of historical asset prices (adj closed)
- **solution** – (str) specify the type of solution to compute. Options are: cla_turning_points, max_sharpe, min_volatility, efficient_frontier
- **resample_by** – (str) specifies how to resample the prices - weekly, daily, monthly etc.. Defaults to 'B' meaning daily business days which is equivalent to no resampling

4.10.3 Mean-Variance Optimisation

This class contains the classic Mean-Variance optimisation techniques which use quadratic optimisation to get solutions to the portfolio allocation problem. Currently, it only supports the basic inverse-variance allocation strategy (IVP) but we aim to add more functions to tackle different optimisation objectives like maximum sharpe, minimum volatility, targeted-risk return maximisation and much more.

Implementation

This module implements the classic mean-variance optimisation techniques for calculating the efficient frontier. It uses typical quadratic optimisers to generate optimal portfolios for different objective functions.

class MeanVarianceOptimisation

This class contains a variety of methods dealing with different solutions to the mean variance optimisation problem.

__init__()

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

allocate(*asset_prices*, *solution='inverse_variance'*, *resample_by='B'*)

Calculate the portfolio asset allocations using the method specified.

Parameters

- **asset_prices** – (pd.Dataframe) a dataframe of historical asset prices (daily close)
- **solution** – (str) the type of solution/algorithm to use to calculate the weights
- **resample_by** – (str) specifies how to resample the prices - weekly, daily, monthly etc.. Defaults to 'B' meaning daily business days which is equivalent to no resampling

4.10.4 Examples

Lets see how to import and use the 3 portfolio optimisation classes

```
from mlfinlab.portfolio_optimization.cla import CLA
from mlfinlab.portfolio_optimization.hrp import HierarchicalRiskParity
from mlfinlab.portfolio_optimization.mean_variance import MeanVarianceOptimisation
import numpy as np
import pandas as pd
```

```
# Read in data
stock_prices = pd.read_csv('FILE_PATH', parse_dates=True, index_col='Date') # The␣
↪date column may be named differently for your input.
```

One important thing to remember here - all the 3 classes require that the stock prices dataframe be indexed by date because internally this will be used to calculate the expected returns.

```
# Compute HRP weights
hrp = HierarchicalRiskParity()
hrp.allocate(asset_prices=stock_prices, resample_by='B')
hrp_weights = hrp.weights.sort_values(by=0, ascending=False, axis=1)

# Compute IVP weights
mvo = MeanVarianceOptimisation()
mvo.allocate(asset_prices=stock_prices, solution='inverse_variance', resample_by='B
ivp_weights = mvo.weights.sort_values(by=0, ascending=False, axis=1)
```

For HRP and IVP, you can access the computed weights as shown above. They are in the form of a dataframe and we can sort them in descending order of their weights.

```
# Compute different solutions using CLA
cla = CLA()

# Turning Points
cla.allocate(asset_prices=stock_prices, resample_by='W', solution='cla_turning_points
↪')
cla_weights = cla.weights.sort_values(by=0, ascending=False, axis=1) # Gives a␣
↪dataframe with each row as a solution (turning_points)

# Maximum Sharpe Solution
cla.allocate(asset_prices=stock_prices, resample_by='W', solution='max_sharpe')
cla_weights = cla.weights.sort_values(by=0, ascending=False, axis=1) # Single set of␣
↪weights for the max-sharpe portfolio
max_sharpe_value = cla.max_sharpe # Accessing the max sharpe value

# Minimum Variance Solution
cla.allocate(asset_prices=stock_prices, resample_by='W', solution='min_volatility')
cla_weights = cla.weights.sort_values(by=0, ascending=False, axis=1) # Single set of␣
↪weights for the min-variance portfolio
min_variance_value = cla.min_var # Accessing the min-variance value

# Efficient Frontier Solution
cla.allocate(asset_prices=stock_prices, resample_by='W', solution='efficient_frontier
↪')
cla_weights = cla.weights
means, sigma = cla.efficient_frontier_means, cla.efficient_frontier_sigma
```

4.10.5 Research Notebooks

The following research notebooks can be used to better understand how the algorithms within this module can be used on real stock data.

- Chapter 16 Exercise Notebook
- *Data Structures*
- *Filters*
- *Labeling*
- *Sampling*
- *Fractionally Differentiated Features*
- *Cross Validation*
- *Sequentially Bootstrapped Bagging Classifier/Regressor*
- *Feature Importance*
- *Bet Sizing*
- *Portfolio Optimisation*

CHAPTER

FIVE

ADDITIONAL INFORMATION

5.1 Contact

At the moment the project is still rather small and thus I would recommend getting in touch with us over email so that we can further discuss the areas of contribution that interest you the most. We have a slack channel where we all communicate.

For now you can get hold us at: research@hudsonthames.org

Looking forward to hearing from you!

5.2 Contributing

5.2.1 Areas of Contribution

Currently we have a live project board that follows the principles of Agile Project Management. This board is available to the public and lets everyone know what the maintainers are currently working on. The board has an ice bucket filled with new features and documentation that have priority.

Needless to say, if you are interested in working on something not on the project board, just drop us an email. We would be very excited to discuss this further.

The typical contributions are:

- Answering the questions at the back of a chapter in a Jupyter Notebook. Research repo
- Adding new features and core functionality to the mlfinlab package.

5.2.2 Templates

We have created templates to help aid in creating issues and PRs:

- Bug report
- Feature request
- Custom issue template
- Pull Request Template

5.3 License

module mlfinlab

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

F

G

H

I

L

M

P

R

S

Z