Perfana

Daniel

Dec 16, 2019

CONTENTS

Performance Analytics (Perfana) is a toolbox to calculate various analytics and statistics for financial engineering. It also contains shorthands for plotting common charts.

CHAPTER

ONE

PERFANA PACKAGE

1.1 Getting Started

1.1.1 Python version support

Only Python 3.6 and 3.7.

1.1.2 Installing Perfana

`Perfana` can be installed via pip from PyPI

```
pip install perfana
```

Alternatively, you can install it via `conda` with

```
conda install -c danielbok perfana
```

1.2 API Reference

The exact API of all functions and classes, as given by the docstrings. The API documents expected types and allowed features for all functions, and all parameters available for the algorithms.

1.2.1 Core

Core API applies to a series of methods that apply to a pandas `DataFrame` or `Series` or iterable TimeSeries like object.

Core API

Core API applies to a series of methods that apply to a pandas DataFrame or Series or iterable TimeSeries like object.

Relative

Relative API contains a series of functions that apply to a pandas DataFrame or Series or iterable TimeSeries like object to calculate various forms of relative comparison measures.

Correlation Measure

perfana.core.relative.**correlation_measure**(*portfolio*, *benchmark*, *duration='monthly'*, ***, *is_returns=False*, *date_as_index=True*)

Computes the correlation measure through time. The data is assumed to be daily. If the benchmark is a single series, a single TimeSeriesData will be returned. Otherwise, a dictionary of TimeSeries will be returned where the keys are each individual benchmark

Parameters

- **portfolio** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The portfolio values vector or matrix
- **benchmark** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The benchmark values vector or matrix
- **duration** (Union[str, int]) – Duration to calculate the relative price index with. Either a string or positive integer value can be specified. Supported string values are 'day', 'week', 'month', 'quarter', 'semi-annual' and 'year'
- **is_returns** – Set this to true if the portfolio and benchmark values are in "returns" instead of raw values (i.e. prices or raw index value)
- **date_as_index** – If true, returns the date as the dataframe's index. Otherwise, the date is placed as a column in the dataframe

Returns A DataFrame of the correlation measure between the assets in the portfolio against the benchmark If multiple series are included in the benchmark, returns a dictionary where the keys are the benchmarks' name and the values are the correlation measure of the portfolio against that particular benchmark

Return type TimeSeriesData or dict of TimeSeriesData

Examples

```
>>> from perfana.datasets import load_etf
>>> from perfana.core import correlation_measure
>>> etf = load_etf().dropna()
>>> returns = etf.iloc[:, 1:]
>>> benchmark = etf.iloc[:, 0]
>>> correlation_measure(returns, benchmark, 'monthly').head()
                BND       VTI       VWO
Date
2007-05-10 -0.384576  0.890783  0.846000
2007-05-11 -0.525299  0.911693  0.857288
2007-05-14 -0.482180  0.912002  0.855114
2007-05-15 -0.439073  0.913992  0.842561
2007-05-16 -0.487110  0.899859  0.837781
```

Relative Price Index

perfana.core.relative.**relative_price_index**(*portfolio*, *benchmark*, *duration='monthly'*, ***, *is_returns=False*, *date_as_index=True*)

Computes the relative price index through time. The data is assumed to be daily. If the benchmark is a single series, a single TimeSeriesData will be returned. Otherwise, a dictionary of TimeSeries will be returned where the keys are each individual benchmark

Notes

The relative price index at a particular time t for an asset a against its benchmark b is given by

$$RP_{a,t} = r_{a,t-d} - r_{b,t-d}$$

where d is the duration. For example, if the duration is 'monthly', d will be 22 days.

Parameters

- **portfolio** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The portfolio values vector or matrix
- **benchmark** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The benchmark values vector or matrix
- **duration** (Union[str, int]) – Duration to calculate the relative price index with. Either a string or positive integer value can be specified. Supported string values are 'day', 'week', 'month', 'quarter', 'semi-annual' and 'year'
- **is_returns** – Set this to true if the portfolio and benchmark values are in "returns" instead of raw values (i.e. prices or raw index value)
- **date_as_index** – If true, returns the date as the dataframe's index. Otherwise, the date is placed as a column in the dataframe

Returns A DataFrame of the relative price index between the assets in the portfolio against the benchmark If multiple series are included in the benchmark, returns a dictionary where the keys are the benchmarks' name and the values are the relative price index of the portfolio against that particular benchmark

Return type TimeSeriesData or dict of TimeSeriesData

Examples

```
>>> from perfana.datasets import load_etf
>>> from perfana.core import relative_price_index
>>> etf = load_etf().dropna()
>>> returns = etf.iloc[:, 1:]
>>> benchmark = etf.iloc[:, 0]
>>> relative_price_index(returns, benchmark, 'monthly').head()
                BND       VTI       VWO
Date
2007-05-10 -0.016000  0.009433  0.000458
2007-05-11 -0.031772  0.008626  0.013009
2007-05-14 -0.016945  0.014056  0.008658
2007-05-15 -0.002772  0.020824  0.018758
2007-05-16  0.002791  0.025402  0.028448
```

Returns

Returns API contains a series of functions that apply to a pandas DataFrame or Series or iterable TimeSeries like object to calculate various forms of returns.

Active Premium

perfana.core.returns.**active_premium**(*ra*, *rb*, *freq=None*, *geometric=True*, *prefixes=('PRT', 'BMK')*)

The return on an investment's annualized return minus the benchmark's annualized return.

Parameters

- **ra** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The assets returns vector or matrix
- **rb** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The benchmark returns
- **freq** (Optional[str]) – Frequency of the data. Use one of monthly, quarterly, semi-annually, yearly
- **geometric** – If True, calculates the geometric returns. Otherwise, calculates the arithmetic returns
- **prefixes** – Prefix to apply to overlapping column names in the left and right side, respectively. This is also applied when the column name is an integer (i.e. 0 -> PRT_0). It is the default name of the Series data if there are no name to the Series

Returns Active premium of each strategy against benchmark

Return type TimeSeriesData

Examples

```
>>> from perfana.datasets import load_etf
>>> from perfana.core import active_premium
# Get returns starting from the date where all etf has data
>>> etf = load_etf().dropna().pa.to_returns().dropna()
>>> active_premium(etf, etf)
          VBK       BND       VTI       VWO
VBK  0.000000 -0.055385 -0.010407 -0.063939
BND  0.055385  0.000000  0.044979 -0.008554
VTI  0.010407 -0.044979  0.000000 -0.053532
VWO  0.063939  0.008554  0.053532  0.000000
>>> active_premium(etf.VBK, etf.BND)
          VBK
BND  0.055385
```

Annualized Returns

perfana.core.returns.**annualized_returns**(*r*, *freq=None*, *geometric=True*)

Calculates the annualized returns from the data

The formula for annualized geometric returns is formulated by raising the compound return to the number of periods in a year, and taking the root to the number of total observations:

$$\prod_{i}^{N}(1+r_i)^{\frac{s}{N}}-1$$

where s is the number of observations in a year, and N is the total number of observations.

For simple returns (geometric=FALSE), the formula is:

$$\frac{s}{N}\sum_{i}^{N} r_i$$

Parameters

- **r** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – Numeric returns series or data frame
- **freq** (Optional[str]) – Frequency of the data. Use one of daily, weekly, monthly, quarterly, semi-annually, yearly
- **geometric** – If True, calculates the geometric returns. Otherwise, calculates the arithmetic returns

Returns Annualized returns

Return type float or Series

Examples

```
>>> from perfana.datasets import load_etf
>>> from perfana.core import annualized_returns
# Get returns starting from the date where all etf has data
>>> etf = load_etf().dropna().pa.to_returns().dropna()
VBK    0.091609
BND    0.036224
VTI    0.081203
VWO    0.027670
dtype: float64
>>> annualized_returns(etf.VWO)
0.02767037698144148
```

Excess Returns

perfana.core.returns.**excess_returns**(*ra*, *rb*, *freq=None*, *geometric=True*)

An average annualized excess return is convenient for comparing excess returns

Excess returns is calculated by first annualizing the asset returns and benchmark returns stream. See the docs for *annualized_returns()* for more details. The geometric returns formula is:

$$r_g = \frac{r_a - r_b}{1 + r_b}$$

The arithmetic excess returns formula is:

$$r_g = r_a - r_b$$

Returns calculation will be truncated by the one with the shorter length. Also, annualized returns are calculated by the geometric annualized returns in both cases

Parameters

- **ra** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The assets returns vector or matrix

- **rb** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The benchmark returns. If this is a vector and the asset returns is a matrix, then all assets returns (columns) will be compared against this single benchmark. Otherwise, if this is a matrix, then assets will be compared to each individual benchmark (i.e. column for column)
- **freq** (Optional[str]) – Frequency of the data. Use one of [daily, weekly, monthly, quarterly, semi-annually, yearly]
- **geometric** – If True, calculates the geometric excess returns. Otherwise, calculates the arithmetic excess returns

Returns Excess returns of each strategy against benchmark

Return type TimeSeriesData

Examples

```
>>> from perfana.datasets import load_etf
>>> from perfana.core import excess_returns
# Get returns starting from the date where all etf has data
>>> etf = load_etf().dropna().pa.to_returns().dropna()
>>> excess_returns(etf, etf.VBK)
VBK    0.000000
BND   -0.050737
VTI   -0.009533
VWO   -0.058573
dtype: float64
```

Relative Returns

perfana.core.returns.**relative_returns**(*ra*, *rb*, *prefixes=('PRT', 'BMK')*)

Calculates the ratio of the cumulative performance for two assets through time

Parameters

- **ra** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The assets returns vector or matrix
- **rb** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The benchmark returns
- **prefixes** – Prefix to apply to overlapping column names in the left and right side, respectively. This is also applied when the column name is an integer (i.e. 0 -> PRT_0). It is the default name of the Series data if there are no name to the Series

Returns Returns a DataFrame of the cumulative returns ratio between 2 asset classes. Returns a Series if there is only 2 compared classes.

Return type TimeSeriesData

Examples

```
>>> from perfana.datasets import load_etf
>>> from perfana.core import relative_returns
# Get returns starting from the date where all etf has data
>>> etf = load_etf().dropna().pa.to_returns().dropna()
```

(continues on next page)

(continued from previous page)

```
>>> relative_returns(etf.tail(), etf.VBK.tail())
            VBK/VBK   BND/VBK   VTI/VBK   VWO/VBK
Date
2019-02-25      1.0  0.996027  0.997856  1.009737
2019-02-26      1.0  1.004013  1.002591  1.013318
2019-02-27      1.0  0.997005  0.997934  1.000389
2019-02-28      1.0  1.001492  1.001461  0.998348
2019-03-01      1.0  0.987385  0.997042  0.988521
```

Risk

Risk API contains a series of functions that apply to a pandas DataFrame or Series or iterable TimeSeries like object to calculate various forms of risk.

Drawdown

perfana.core.risk.**drawdown**(*data*, *weights=None*, *geometric=True*, *rebalance=True*)

Calculates the drawdown at each time instance.

If data is DataFrame-like, weights must be specified. If data is Series-like, weights can be left empty.

Parameters

- **data** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The assets returns vector or matrix
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series, None]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **geometric** – If True, calculates the geometric mean, otherwise, calculates the arithmetic mean.
- **rebalance** – If True, portfolio is assumed to be rebalanced at every step.

Returns Drawdown at each time instance

Return type Series

Examples

```
>>> from perfana.datasets import load_hist
>>> from perfana.core import drawdown
>>> hist = load_hist().iloc[:, :7]
>>> weights = [0.25, 0.18, 0.24, 0.05, 0.04, 0.13, 0.11]
>>> drawdown(hist, weights).min()
-0.4007984968456346
>>> drawdown(hist.iloc[:, 0]).min()
-0.5491340502573534
```

```
import matplotlib.pyplot as plt

from perfana.core import drawdown
from perfana.datasets import load_hist
```

(continues on next page)

(continued from previous page)

```
data = load_hist().iloc[:, :7]
weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]

dd = drawdown(data, weights)
plt.plot(dd.index, dd)
plt.title("Drawdown")
plt.xlabel("Time")
plt.ylabel("Drawdown Depth")
plt.show()
```

Drawdown Summary

perfana.core.risk.**drawdown_summary**(*data*, *weights=None*, *geometric=True*, *rebalance=True*, ***, *top=5*)

A summary of each drawdown instance. Output is ranked by depth of the drawdown.

If data is DataFrame-like, weights must be specified. If data is Series-like, weights can be left empty.

Parameters

- **data** (Union[DataFrame, Iterable[Union[int, float]], ndarray, Series]) – The assets returns vector or matrix
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series, None]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **geometric** – If True, calculates the geometric mean, otherwise, calculates the arithmetic mean.
- **rebalance** – If True, portfolio is assumed to be rebalanced at every step.
- **top** (Optional[int]) – If None, returns all episodes. If specified, returns the top *n* episodes ranked by the depth of drawdown.

Returns A data frame summarizing each drawdown episode

Return type DataFrame

Examples

```
>>> from perfana.datasets import load_hist
>>> from perfana.core import drawdown_summary
>>> hist = load_hist().iloc[:, :7]
>>> weights = [0.25, 0.18, 0.24, 0.05, 0.04, 0.13, 0.11]
>>> drawdown_summary(hist, weights)
       Start     Trough        End  Drawdown  Length  ToTrough  Recovery
0 2007-11-30 2009-02-28 2014-02-28 -0.400798      76        16        60
1 2000-04-30 2003-03-31 2004-02-29 -0.203652      47        36        11
2 1990-01-31 1990-11-30 1991-05-31 -0.150328      17        11         6
3 1998-04-30 1998-10-31 1999-06-30 -0.149830      15         7         8
4 1994-02-28 1995-03-31 1996-01-31 -0.132766      24        14        10
>>> drawdown_summary(hist.iloc[:, 0])
       Start     Trough        End  Drawdown  Length  ToTrough  Recovery
0 2007-11-30 2009-02-28 2014-05-31 -0.549134      79        16        63
```

(continues on next page)

(continued from previous page)

1	2000-04-30	2003-03-31	2006-12-31	-0.474198	81	36	45
2	1990-01-31	1990-09-30	1994-01-31	-0.286489	49	9	40
3	1998-08-31	1998-09-30	1999-01-31	-0.148913	6	2	4
4	2018-10-31	2018-12-31	2019-03-31	-0.130014	6	3	3

1.2.2 Monte Carlo

Monte Carlo API applies to a series of methods that apply to a 3 dimensional data cube where the dimensions represent the time, trials and assets respectively. For example, if the simulated cube projects 10 years of monthly data for 8 assets for 10000 trials (that is 10000 simulations), the cube will be a numpy array with shape (120, 10000, 8).

Monte Carlo API

Monte Carlo API applies to a series of methods that apply to a 3 dimensional data cube where the dimensions represent the time, trials and assets respectively. For example, if the simulated cube projects 10 years of monthly data for 8 assets for 10000 trials (that is 10000 simulations), the cube will be a numpy array with shape (120, 10000, 8).

Returns

Annualized Returns

perfana.monte_carlo.returns.**annualized_returns_m**(*data*, *weights*, *freq*, *geometric=True*, *rebalance=True*)

Calculates the annualized returns from the Monte Carlo simulation

The formula for annualized geometric returns is formulated by raising the compound return to the number of periods in a year, and taking the root to the number of total observations. For the rebalance, geometric returns, the annualized returns is derived by:

$$y = M/s$$

$$\frac{1}{N}\sum_{i}^{N}\left[\prod_{j}^{T}\left(1+\sum_{k}^{A}(r_{ijk}\cdot w_k)\right)\right]^{\frac{1}{y}} - 1$$

where *s* is the number of observations in a year, and *M* is the total number of observations, *N* is the number of trials in the simulation, *T* is the number of trials in the simulation and *A* is the number of assets in the simulation.

For simple returns (geometric=FALSE), the formula for the rebalanced case is:

$$\frac{s}{NM}\left[\sum_{i}^{N}\sum_{j}^{T}\sum_{k}^{A}(r_{ijk}\cdot w_k)\right]$$

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.

- **freq** (Union[str, int]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.
- **geometric** (bool) – If True, calculates the geometric mean, otherwise, calculates the arithmetic mean.
- **rebalance** (bool) – If True, portfolio is assumed to be rebalanced at every step.

Returns Annualized returns of the portfolio

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import annualized_returns_m
>>> cube = load_cube()[..., :7]
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> annualized_returns_m(cube, weights, 'month')
0.02111728739277985
```

Annualized Returns against Benchmark

perfana.monte_carlo.returns.**annualized_bmk_returns_m**(*data*, *weights*, *bmk_weights*, *freq*, *geometric=True*, *rebalance=True*)

Calculates the returns of the portfolio relative to a benchmark portfolio.

The benchmark components must be placed after the portfolio components in the simulated returns cube.

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **bmk_weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the benchmark portfolio.
- **freq** (Union[str, int]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.
- **geometric** (bool) – If True, calculates the geometric mean, otherwise, calculates the arithmetic mean.
- **rebalance** (bool) – If True, portfolio is assumed to be rebalanced at every step.

Returns The portfolio returns relative to the benchmark

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import annualized_bmk_returns_m
>>> cube = load_cube()
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> bmk_weights = [0.65, 0.35]
>>> freq = "quarterly"
>>> annualized_bmk_returns_m(cube, weights, bmk_weights, freq)
-0.006819613944426206
```

Annualized Quantile Returns

perfana.monte_carlo.returns.**annualized_quantile_returns_m**(*data*, *weights*, *quantile*, *freq*, *geometric=True*, *rebalance=True*, *interpolation='midpoint'*)

Compute the q-th quantile of the returns in the simulated data cube.

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **quantile** (Union[float, Iterable[float]]) – Quantile or sequence of quantiles to compute, which must be between 0 and 1 inclusive
- **freq** (Union[str, int]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.
- **geometric** (bool) – If True, calculates the geometric mean, otherwise, calculates the arithmetic mean.
- **rebalance** (bool) – If True, portfolio is assumed to be rebalanced at every step.
- **interpolation** – This optional parameter specifies the interpolation method to use when the desired quantile lies between two data points `i < j`:
  - linear: `i + (j - i) * fraction`, where `fraction` is the fractional part of the index surrounded by `i` and `j`.
  - lower: `i`.
  - higher: `j`.
  - nearest: `i` or `j`, whichever is nearest.
  - midpoint: `(i + j) / 2`.

Returns The returns of the portfolio relative to the benchmark at the specified quantile

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import annualized_quantile_returns_m
>>> cube = load_cube()[..., :7]
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> freq = "quarterly"
>>> q = 0.25
>>> annualized_quantile_returns_m(cube, weights, q, freq)
0.005468353416130167
>>> q = [0.25, 0.75]
>>> annualized_quantile_returns_m(cube, weights, q, freq)
array([0.00546835, 0.03845033])
```

Annualized Quantile Returns against Benchmark

perfana.monte_carlo.returns.**annualized_bmk_quantile_returns_m**(*data*, *weights*, *bmk_weights*, *quantile*, *freq*, *geometric=True*, *rebalance=True*, *interpolation='midpoint'*)

Compares the annualized returns against a benchmark at the specified quantiles.

The benchmark components must be placed after the portfolio components in the simulated returns cube.

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **bmk_weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the benchmark portfolio.
- **quantile** (Union[float, Iterable[float]]) – Quantile or sequence of quantiles to compute, which must be between 0 and 1 inclusive
- **freq** (Union[str, int]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.
- **geometric** (bool) – If True, calculates the geometric mean, otherwise, calculates the arithmetic mean.
- **rebalance** (bool) – If True, portfolio is assumed to be rebalanced at every step.
- **interpolation** – This optional parameter specifies the interpolation method to use when the desired quantile lies between two data points `i < j`:
  - linear: `i + (j - i) * fraction`, where `fraction` is the fractional part of the index surrounded by `i` and `j`.
  - lower: `i`.
  - higher: `j`.

- nearest: `i` or `j`, whichever is nearest.
- midpoint: `(i + j) / 2`.

Returns The returns of the portfolio over the benchmark at the specified quantiles

Return type float or array_like of floats

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import annualized_bmk_quantile_returns_m
>>> cube = load_cube()
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> bmk_weights = [0.65, 0.35]
>>> freq = "quarterly"
>>> q = 0.25
>>> annualized_bmk_quantile_returns_m(cube, weights, bmk_weights, q, freq)
-0.010792419409674459
>>> q = [0.25, 0.75]
>>> annualized_bmk_quantile_returns_m(cube, weights, bmk_weights, q, freq)
array([-0.01079242, -0.0025487 ])
```

Returns Attribution

perfana.monte_carlo.returns.**returns_attr**(*data*, *weights*, *freq*, *geometric=True*, *rebalance=True*)

Derives the returns attribution given a data cube and weights.

Notes

The return values are defined as follows:

- **marginal** The absolute marginal contribution of the asset class towards the portfolio returns. It is essentially the percentage attribution multiplied by the portfolio returns.
- **percentage** The percentage contribution of the asset class towards the portfolio returns. This number though named in percentage is actually in decimals. Thus 0.01 represents a 1% contribution.

Parameters

- **data** (`ndarray`) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (`Union[Iterable[Union[int, float]], ndarray, Series]`) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **freq** (`Union[str, int]`) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.
- **geometric** (`bool`) – If True, calculates the geometric mean, otherwise, calculates the arithmetic mean.
- **rebalance** (`bool`) – If True, portfolio is assumed to be rebalanced at every step.

Returns A named tuple of marginal and percentage returns attribution respectively. The marginal attribution is the returns of the simulated data over time multiplied by the percentage attribution.

Return type Attribution

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import returns_attr
>>> cube = load_cube()[..., :3]
>>> weights = [0.33, 0.34, 0.33]
>>> freq = "quarterly"
>>> attr = returns_attr(cube, weights, freq)
>>> attr.marginal
array([0.00996204, 0.00733369, 0.00963802])
>>> attr.percentage
array([0.36987203, 0.27228623, 0.35784174])
>>> attr.marginal is attr[0]
True
>>> attr.percentage is attr[1]
True
```

Returns Distribution

perfana.monte_carlo.returns.**returns_distribution**(*data*, *weights*, *freq=None*, *annualize=True*, *geometric=True*, *rebalance=True*)

Calculates the returns distribution of the simulation cube

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **freq** (Union[str, int, None]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1. If annualize is False, freq can be ignored.
- **annualize** – If true, the returns distribution values are annualized
- **geometric** (bool) – If True, calculates the geometric mean, otherwise, calculates the arithmetic mean.
- **rebalance** (bool) – If True, portfolio is assumed to be rebalanced at every step.

Returns A vector of the distribution of returns

Return type Array

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import returns_distribution
>>> cube = load_cube()[..., :3]
>>> weights = [0.33, 0.34, 0.33]
>>> freq = "quarterly"
>>> returns_distribution(cube, weights, freq).shape
(1000,)
```

Returns Path

perfana.monte_carlo.returns.**returns_path**(*data*, *weights*, *rebalance=True*, *quantile=None*)

Returns a matrix of the returns path of the portfolio.

The first axis represents the time and the second axis represents the trials.

If the `quantile` argument is specified, the specific quantile for each time period will be returned. Thus, if the 0.75 quantile is specified, it is the 75th quantile for each time period and not the path the 75th quantile in the terminal period took.

Parameters

- **data** (`ndarray`) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (`Union[Iterable[Union[int, float]], ndarray, Series]`) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **rebalance** (`bool`) – If True, portfolio is assumed to be rebalanced at every step.
- **quantile** (`Union[int, float, Iterable[Union[int, float]], ndarray, Series, None]`) – Quantile or sequence of quantiles to compute, which must be between 0 and 1 inclusive

Returns The returns path for the portfolio

Return type ndarray

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import returns_path
>>> cube = load_cube()[..., :3]
>>> weights = [0.33, 0.34, 0.33]
>>> returns_path(cube, weights).shape
(1000, 81)
>>> returns_path(cube, weights, quantile=0.75).shape  # 75th quantile
(81, )
>>> returns_path(cube, weights, quantile=[0.25, 0.5, 0.75]).shape  # 25th, 50th␣
↪and 75th quantile
(3, 81)
```

```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from perfana.datasets import load_cube
from perfana.monte_carlo import returns_path

data = load_cube()[..., :7]
weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
quantile = [0.05, 0.25, 0.5, 0.75, 0.95]
color = np.array([
    (200, 200, 200),
    (143, 143, 143),
    (196, 8, 8),
    (143, 143, 143),
    (200, 200, 200),
]) / 255

paths = returns_path(data, weights, quantile=quantile)
n, t = paths.shape
x = np.arange(t)

fig = plt.figure(figsize=(8, 6))
sp = fig.add_subplot(111)
for i in reversed(range(n)):
    label = f"{int(quantile[i] * 100)}"
    sp.plot(x, paths[i], figure=fig, label=label, color=color[i])

sp.fill_between(x, paths[0], paths[-1], color=color[0])
sp.fill_between(x, paths[1], paths[-2], color=color[1])
sp.set_title("Cumulative returns path")
sp.set_xlabel("Time Period")
sp.set_ylabel("Returns")
sp.grid()
sp.legend(title="Percentile")
fig.show()
```

Risk

Portfolio Beta against Asset

perfana.monte_carlo.risk.**beta_m**(*cov_or_data*, *weights*, *freq=None*, *aid=0*)

Derives the portfolio beta with respect to the specified asset class

Notes

The asset is identified by its index (aid) on the covariance matrix / simulated returns cube / weight vector. If a simulated returns data cube is given, the frequency of the data must be specified. In this case, the empirical covariance matrix would be used to derive the volatility.

Parameters

- **cov_or_data** (ndarray) – Covariance matrix or simulated returns data cube.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the covariance matrix shape or the simulated data's last axis.

- **freq** (Union[str, int, None]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.
- **aid** – Asset index

Returns Portfolio beta with respect to asset class.

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import portfolio_cov, beta_m
>>> data = load_cube()[..., :3]  # first 3 asset classes only
>>> weights = [0.33, 0.34, 0.33]
>>> freq = 'quarterly'
>>> dm_eq_id = 0  # calculate correlation with respect to developing markets␣
↪equity
>>> beta_m(data, weights, freq, dm_eq_id)
1.3047194776321622
```

Portfolio Correlation against Asset

perfana.monte_carlo.risk.**correlation_m**(*cov_or_data*, *weights*, *freq=None*, *aid=0*)

Derives the portfolio correlation with respect to the specified asset class

Notes

The asset is identified by its index (aid) on the covariance matrix / simulated returns cube / weight vector. If a simulated returns data cube is given, the frequency of the data must be specified. In this case, the empirical covariance matrix would be used to derive the volatility.

Parameters

- **cov_or_data** (ndarray) – Covariance matrix or simulated returns data cube.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the covariance matrix shape or the simulated data's last axis.
- **freq** (Union[str, int, None]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.
- **aid** – Asset index

Returns Portfolio correlation with respect to asset class

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import portfolio_cov, correlation_m
>>> data = load_cube()[..., :3]  # first 3 asset classes only
>>> weights = [0.33, 0.34, 0.33]
>>> freq = 'quarterly'
>>> dm_eq_id = 0  # calculate correlation with respect to developing markets␣
↪equity
>>> correlation_m(data, weights, freq, dm_eq_id)
0.9642297301278216
```

CVaR Attribution

perfana.monte_carlo.risk.**cvar_attr**(*data*, *weights*, *alpha=0.95*, *rebalance=True*, *invert=True*)

Calculates the CVaR (Expected Shortfall) attribution for each asset class in the portfolio.

Notes

From a mathematical point of view, the alpha value (confidence level for calculation) should be taken at the negative extreme of the distribution. However, the default is set to ease the practitioner.

The return values are defined as follows:

- **marginal** The absolute marginal contribution of the asset class towards the portfolio CVaR. It is essentially the percentage attribution multiplied by the portfolio CVaR.
- **percentage** The percentage contribution of the asset class towards the portfolio CVaR. This number though named in percentage is actually in decimals. Thus 0.01 represents a 1% contribution.

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **alpha** – Confidence level for calculation.
- **rebalance** – If True, portfolio is assumed to be rebalanced at every step.
- **invert** – Whether to invert the confidence interval level. See Notes.

Returns A named tuple of relative and absolute CVaR (expected shortfall) attribution respectively. The absolute attribution is the CVaR of the simulated data over time multiplied by the percentage attribution.

Return type Attribution

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import cvar_attr
>>> cube = load_cube()[..., :3]
>>> weights = [0.33, 0.34, 0.33]
```

(continues on next page)

(continued from previous page)

```
>>> attr = cvar_attr(cube, weights, alpha=0.95)
>>> attr.marginal
array([-0.186001  , -0.35758411, -0.20281477])
>>> attr.percentage
array([0.24919752, 0.47907847, 0.27172401])
>>> attr.marginal is attr[0]
True
>>> attr.percentage is attr[1]
True
```

CVaR Diversification Ratio

perfana.monte_carlo.risk.**cvar_div_ratio**(*data*, *weights*, *alpha=0.95*, *rebalance=True*, *invert=True*)

Calculates the CVaR (Expected Shortfall) tail diversification ratio of the portfolio

Notes

From a mathematical point of view, the alpha value (confidence level for calculation) should be taken at the negative extreme of the distribution. However, the default is set to ease the practitioner.

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **alpha** – Confidence level for calculation.
- **rebalance** – If True, portfolio is assumed to be rebalanced at every step.
- **invert** – Whether to invert the confidence interval level. See Notes.

Returns CVaR (Expected Shortfall) tail diversification ratio

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import cvar_div_ratio
>>> cube = load_cube()[..., :3]
>>> weights = [0.33, 0.34, 0.33]
>>> cvar_div_ratio(cube, weights)
0.8965390850633622
```

Portfolio CVaR

perfana.monte_carlo.risk.**cvar_m**(*data*, *weights*, *alpha=0.95*, *rebalance=True*, *invert=True*)

Calculates the Conditional Value at Risk (Expected Shortfall) of the portfolio.

Notes

From a mathematical point of view, the alpha value (confidence level for calculation) should be taken at the negative extreme of the distribution. However, the default is set to ease the practitioner.

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **alpha** – Confidence level for calculation.
- **rebalance** – If True, portfolio is assumed to be rebalanced at every step.
- **invert** – Whether to invert the confidence interval level. See Notes.

Returns CVaR (Expected Shortfall) of the portfolio

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import cvar_m
>>> cube = load_cube()[..., :3]
>>> weights = [0.33, 0.34, 0.33]
>>> cvar_m(cube, weights)
-0.7463998716846179
```

Diversification Ratio

perfana.monte_carlo.risk.**diversification_m**(*cov_or_data*, *weights*, *freq*)

Derives the diversification ratio of the portfolio

Parameters

- **cov_or_data** (ndarray) – Covariance matrix or simulated returns data cube.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **freq** (Union[str, int]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.

Returns Tracking error of the portfolio

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import portfolio_cov, diversification_m
>>> data = load_cube()[..., :7]  # first 7 asset classes
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> freq = 'quarterly'
>>> diversification_m(data, weights, freq)
```

Drawdown Statistics

perfana.monte_carlo.risk.**drawdown_m**(*data*, *weights*, *geometric=True*, *rebalance=True*)

Calculates the drawdown statistics

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **geometric** – If True, calculates the geometric mean, otherwise, calculates the arithmetic mean.
- **rebalance** – If True, portfolio is assumed to be rebalanced at every step.

Returns A named tuple containing the average maximum drawdown and the drawdown path for each simulation instance.

Return type Drawdown

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import drawdown_m
>>> data = load_cube()[..., :7]
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> dd = drawdown_m(data, weights)
>>> dd.average
-0.3198714473717889
>>> dd.paths.shape
(80, 1000)
```

Portfolio Empirical Covariance Matrix

perfana.monte_carlo.risk.**portfolio_cov**(*data*, *freq*)

Forms the empirical portfolio covariance matrix from the simulation data cube

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **freq** (Union[str, int]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.

Returns Empirical portfolio covariance matrix

Return type array_like of float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import portfolio_cov
>>> data = load_cube()[..., :3]  # first 3 asset classes only
>>> portfolio_cov(data, 'quarterly').round(4)
array([[0.0195, 0.0356, 0.021 ],
       [0.0356, 0.0808, 0.0407],
       [0.021 , 0.0407, 0.0239]])
```

Risk Performance Benchmark

perfana.monte_carlo.risk.**prob_loss**(*data*, *weights*, *rebalance=True*, *terminal=False*)

Calculates the probability of the portfolio suffering a loss

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **rebalance** – If True, portfolio is assumed to be rebalanced at every step.
- **terminal** – If True, this only compares the probability of a loss at the last stage. If False (default), the calculation will take into account if the portfolio was "ruined" and count it as a loss even though the terminal value is positive.

Returns A named tuple containing the probability of underperformance and loss

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import prob_loss
>>> data = load_cube()
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> prob_loss(data, weights)
0.198
```

Risk Performance Benchmark

perfana.monte_carlo.risk.**prob_under_perf**(*data*, *weights*, *bmk_weights*, *rebalance=True*, *terminal=False*)

Calculates the probability of the portfolio underperforming the benchmark at the terminal state

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **bmk_weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the benchmark portfolio.
- **rebalance** – If True, portfolio is assumed to be rebalanced at every step.
- **terminal** – If True, this only compares the probability of underperformance at the last stage. If False (default), the calculation will take into account if the portfolio was "ruined" and count it as an underperformance against the benchmark even though the terminal value is higher than the benchmark. If both portfolios are "ruined", then it underperforms if it is ruined earlier.

Returns A named tuple containing the probability of underperformance and loss

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import prob_under_perf
>>> data = load_cube()
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> bmk_weights = [0.65, 0.35]
>>> prob_under_perf(data, weights, bmk_weights)
0.863
```

Tail Loss Statistics

perfana.monte_carlo.risk.**tail_loss**(*data*, *weights*, *threshold=-0.3*, *rebalance=True*)

Calculates the probability and expectation of a tail loss beyond a threshold

Threshold by default is set at -0.3, which means find the probability that the portfolio loses more than 30% of its value and the expected loss.

Notes

The return values are defined as follows:

- **prob** Probability of having a tail loss exceeding the threshold
- **expected_loss** Value of the expected loss for the portfolio at the threshold

Parameters

- **data** (ndarray) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.

- **threshold** – Portfolio loss threshold.
- **rebalance** – If True, portfolio is assumed to be rebalanced at every step.

Returns A named tuple containing the probability and expected loss of the portfolio exceeding the threshold.

Return type TailLoss

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import tail_loss
>>> data = load_cube()[..., :3]  # first 3 asset classes only
>>> weights = [0.33, 0.34, 0.33]
>>> loss = tail_loss(data, weights, -0.3)
>>> loss.prob
0.241
>>> loss.expected_loss
-0.3978210273894446
```

Tracking Error

perfana.monte_carlo.risk.**tracking_error_m**(*cov_or_data*, *weights*, *bmk_weights*, *freq*)

Calculates the tracking error with respect to the benchmark.

If a covariance matrix is used as the data, the benchmark components must be placed after the portfolio components. If a simulated returns cube is used as the data, the benchmark components must be placed after the portfolio components.

Parameters

- **cov_or_data** (ndarray) – Covariance matrix or simulated returns data cube.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **bmk_weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the benchmark portfolio.
- **freq** (Union[str, int]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.

Returns Tracking error of the portfolio

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import portfolio_cov, tracking_error_m
>>> data = load_cube()
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> bmk_weights = [0.65, 0.35]
```

(continues on next page)

(continued from previous page)

```
>>> freq = 'quarterly'
>>> tracking_error_m(data, weights, bmk_weights, freq)
0.031183281273726802
```

Volatility Attribution

perfana.monte_carlo.risk.**vol_attr**(*cov_or_data*, *weights*, *freq*)

Derives the volatility attribution given a data cube and weights.

Notes

The return values are defined as follows:

marginal The absolute marginal contribution of the asset class towards the portfolio volatility. It is essentially the percentage attribution multiplied by the portfolio volatility.

percentage The percentage contribution of the asset class towards the portfolio volatility. This number though named in percentage is actually in decimals. Thus 0.01 represents a 1% contribution.

Parameters

- **cov_or_data** (ndarray) – Covariance matrix or simulated returns data cube.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **freq** (Union[str, int]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.

Returns A named tuple of relative and absolute volatility attribution respectively. The absolute attribution is the volatility of the simulated data over time multiplied by the percentage attribution.

Return type Attribution

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import vol_attr
>>> data = load_cube()[..., :3]  # first 3 asset classes only
>>> weights = [0.33, 0.34, 0.33]
>>> freq = 'quarterly'
>>> attr = vol_attr(data, weights, freq)
>>> attr.marginal.round(4)
array([0.2352, 0.5006, 0.2643])
>>> attr.percentage.round(4)
array([0.0445, 0.0947, 0.05  ])
>>> attr.marginal is attr[0]
True
>>> attr.percentage is attr[1]
True
```

Portfolio Volatility

perfana.monte_carlo.risk.**volatility_m**(*cov_or_data*, *weights*, *freq=None*)

Calculates the portfolio volatility given a simulated returns cube or a covariance matrix

Notes

If a simulated returns data cube is given, the frequency of the data must be specified. In this case, the empirical covariance matrix would be used to derive the volatility.

Parameters

- **cov_or_data** (ndarray) – Covariance matrix or simulated returns data cube.
- **weights** (Union[Iterable[Union[int, float]], ndarray, Series]) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the covariance matrix shape or the simulated data's last axis.
- **freq** (Union[str, int, None]) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.

Returns Portfolio volatility.

Return type float

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import portfolio_cov, volatility_m
>>> data = load_cube()[..., :3]  # first 3 asset classes only
>>> weights = [0.33, 0.34, 0.33]
>>> freq = 'quarterly'
>>> cov_mat = portfolio_cov(data, freq).round(4)  # empirical covariance matrix
>>> # Using covariance matrix
>>> volatility_m(cov_mat, weights)
0.1891091219375734
>>> # Using the simulated returns data cube
>>> volatility_m(data, weights, freq)
0.1891091219375734
```

Sensitivity

Sensitivity of Portfolio to Shocks

perfana.monte_carlo.sensitivity.**sensitivity_m**(*data*, *weights*, *freq*, *shock=0.05*, *geometric=True*, *rebalance=True*, *cov=None*, *cvar_cutoff=3*, *cvar_data=None*, *alpha=0.95*, *invert=True*, *names=None*, *leveraged=False*, *distribute=True*)

Calculates the sensitivity of adding and removing from the asset class on the portfolio.

This is a wrapper function for the 3 sensitivity calculations. For more granular usages, use the base functions instead.

Notes

When given a positive shock and a "proportionate" distribution strategy, each asset class is given an additional amount by removing from the other asset classes proportionately. For example, given a portfolio with weights `[0.1, 0.2, 0.3, 0.4]`, a shock of 5% to the first asset in the portfolio will result in weights `[0.15, 0.19, 0.28, 0.38]`. A negative shock works by removing from the asset class and adding to the other asset classes proportionately.

If the distribution strategy is set to `False`, the asset class' weight is increased without removing from the other asset classes. Thus the sum of the portfolio weights will not equal 1.

By default, the portfolio is **not** leveraged. This means that the asset class be shorted (negative shock) to go below 0 and levered (positive shock) to go above 1. The asset class weight is thus capped between 0 and 1 by default. If the `leverage` option is set to `True`, then this value is no longer capped.

Parameters

- **data** (*ndarray*) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (*array_like*) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **freq** (*Frequency*) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.
- **shock** (*float*) – The amount to shock each asset class by. A positive number represents adding to the asset class by proportionately removing from the other asset class. A negative number represents removing from the asset class and adding to the other asset class proportionately.
- **geometric** (*bool*) – If True, calculates the geometric mean, otherwise, calculates the arithmetic mean.
- **cov** (*ndarray*) – Asset covariance matrix
- **cvar_cutoff** (*int*) – Number of years to trim the data cube by for cvar calculation.
- **cvar_data** (*np.ndarray*) – If specified, will use this data cube instead of the main data cube for cvar calculations.
- **alpha** (*float*) – Confidence level for calculation.
- **invert** (*bool*) – Whether to invert the confidence interval level
- **rebalance** (*bool*) – If True, portfolio is assumed to be rebalanced at every step.
- **names** (*list of str*) – Asset class names
- **leveraged** (*bool*) – If True, asset weights are allowed to go below 0 and above 1. This represents that the asset class can be shorted or levered.
- **distribute** (*bool*) – If True, asset value changes are distributed proportionately to all other asset classes. See Notes for more information.

Returns A dataframe with the asset names as the indices and with columns (ret, vol, cvar) representing returns, volatility and CVaR respectively.

Return type DataFrame

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import sensitivity_m
>>> data = load_cube()[..., :7]
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> freq = 'quarterly'
>>> shock = 0.05  # 5% absolute shock
>>> sensitivity_m(data, weights, freq, shock)
              ret       vol      cvar
Asset_1  0.022403  0.113284 -0.485220
Asset_2  0.020484  0.121786 -0.542988
Asset_3  0.022046  0.113964 -0.492411
Asset_4  0.020854  0.109301 -0.478581
Asset_5  0.020190  0.104626 -0.459786
Asset_6  0.020335  0.106652 -0.467798
Asset_7  0.020220  0.106140 -0.468692
```

Sensitivity of Portfolio's Annualized Returns to Shocks

perfana.monte_carlo.sensitivity.**sensitivity_returns_m**(*data*, *weights*, *freq*, *shock=0.05*, *geometric=True*, *rebalance=True*, *names=None*, *leveraged=False*, *distribute=True*)

Calculates the sensitivity of a shock to the annualized returns of the portfolio

Notes

When given a positive shock and a "proportionate" distribution strategy, each asset class is given an additional amount by removing from the other asset classes proportionately. For example, given a portfolio with weights `[0.1, 0.2, 0.3, 0.4]`, a shock of 5% to the first asset in the portfolio will result in weights `[0.15, 0.19, 0.28, 0.38]`. A negative shock works by removing from the asset class and adding to the other asset classes proportionately.

If the distribution strategy is set to `False`, the asset class' weight is increased without removing from the other asset classes. Thus the sum of the portfolio weights will not equal 1.

By default, the portfolio is **not** leveraged. This means that the asset class be shorted (negative shock) to go below 0 and levered (positive shock) to go above 1. The asset class weight is thus capped between 0 and 1 by default. If the `leverage` option is set to `True`, then this value is no longer capped.

Parameters

- **data** (*ndarray*) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (*array_like*) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **shock** (*float*) – The amount to shock each asset class by. A positive number represents adding to the asset class by proportionately removing from the other asset class. A negative number represents removing from the asset class and adding to the other asset class proportionately.

- **freq** (*Frequency*) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.
- **geometric** (*bool*) – If True, calculates the geometric mean, otherwise, calculates the arithmetic mean.
- **rebalance** (*bool*) – If True, portfolio is assumed to be rebalanced at every step.
- **names** (*list of str*) – Asset class names
- **leveraged** (*bool*) – If True, asset weights are allowed to go below 0 and above 1. This represents that the asset class can be shorted or levered.
- **distribute** (*bool*) – If True, asset value changes are distributed proportionately to all other asset classes. See Notes for more information.

Returns A series with asset names as the index and annualized returns as its value

Return type Series

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import sensitivity_returns_m
>>> data = load_cube()[..., :7]
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> freq = 'quarterly'
>>> shock = 0.05  # 5% absolute shock
>>> sensitivity_returns_m(data, weights, freq, shock)
Asset_1    0.022403
Asset_2    0.020484
Asset_3    0.022046
Asset_4    0.020854
Asset_5    0.020190
Asset_6    0.020335
Asset_7    0.020220
Name: ret, dtype: float64
```

Sensitivity of Portfolio's Annualized Volatility to Shocks

perfana.monte_carlo.sensitivity.**sensitivity_vol_m**(*cov_or_data*, *weights*, *freq=None*, *shock=0.05*, *names=None*, *leveraged=False*, *distribute=True*)

Calculates the sensitivity of a shock to the annualized volatility of the portfolio

Parameters

- **cov_or_data** (ndarray) – Monte carlo simulation data or covariance matrix. If simulation cube, this must be 3 dimensional with the axis representing time, trial and asset respectively and frequency will also need to be specified.
- **weights** (*array_like*) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **freq** (*Frequency*) – Frequency of the data. Can either be a string ('week', 'month', 'quarter', 'semi-annual', 'year') or an integer specifying the number of units per year. Week: 52, Month: 12, Quarter: 4, Semi-annual: 6, Year: 1.

- **shock** (*float*) – The amount to shock each asset class by. A positive number represents adding to the asset class by proportionately removing from the other asset class. A negative number represents removing from the asset class and adding to the other asset class proportionately.
- **names** (*list of str*) – Asset class names
- **leveraged** (*bool*) – If True, asset weights are allowed to go below 0 and above 1. This represents that the asset class can be shorted or levered.
- **distribute** (*bool*) – If True, asset value changes are distributed proportionately to all other asset classes. See Notes for more information.

Returns A series with asset names as the index and annualized volatility as its value

Return type Series

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import sensitivity_vol_m
>>> data = load_cube()[..., :7]
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> freq = 'quarterly'
>>> shock = 0.05  # 5% absolute shock
>>> sensitivity_vol_m(data, weights, freq, shock)
Asset_1    0.113284
Asset_2    0.121786
Asset_3    0.113964
Asset_4    0.109301
Asset_5    0.104626
Asset_6    0.106652
Asset_7    0.106140
Name: vol, dtype: float64
```

Sensitivity of Portfolio's CVaR to Shocks

perfana.monte_carlo.sensitivity.**sensitivity_cvar_m**(*data*, *weights*, *shock=0.05*, *alpha=0.95*, *rebalance=True*, *invert=True*, *names=None*, *leveraged=False*, *distribute=True*)

Calculates the sensitivity of a shock to the CVaR of the portfolio

Notes

When given a positive shock and a "proportionate" distribution strategy, each asset class is given an additional amount by removing from the other asset classes proportionately. For example, given a portfolio with weights `[0.1, 0.2, 0.3, 0.4]`, a shock of 5% to the first asset in the portfolio will result in weights `[0.15, 0.19, 0.28, 0.38]`. A negative shock works by removing from the asset class and adding to the other asset classes proportionately.

If the distribution strategy is set to `False`, the asset class' weight is increased without removing from the other asset classes. Thus the sum of the portfolio weights will not equal 1.

By default, the portfolio is **not** leveraged. This means that the asset class be shorted (negative shock) to go below 0 and levered (positive shock) to go above 1. The asset class weight is thus capped between 0 and 1 by default. If the `leverage` option is set to `True`, then this value is no longer capped.

Parameters

- **data** (*ndarray*) – Monte carlo simulation data. This must be 3 dimensional with the axis representing time, trial and asset respectively.
- **weights** (*array_like*) – Weights of the portfolio. This must be 1 dimensional and must match the dimension of the data's last axis.
- **shock** (*float*) – The amount to shock each asset class by. A positive number represents adding to the asset class by proportionately removing from the other asset class. A negative number represents removing from the asset class and adding to the other asset class proportionately.
- **alpha** (*float*) – Confidence level for calculation.
- **invert** (*bool*) – Whether to invert the confidence interval level
- **rebalance** (*bool*) – If True, portfolio is assumed to be rebalanced at every step.
- **names** (*list of str*) – Asset class names
- **leveraged** (*bool*) – If True, asset weights are allowed to go below 0 and above 1. This represents that the asset class can be shorted or levered.
- **distribute** (*bool*) – If True, asset value changes are distributed proportionately to all other asset classes. See Notes for more information.

Returns A series with asset names as the index and CVaR as its value

Return type Series

Examples

```
>>> from perfana.datasets import load_cube
>>> from perfana.monte_carlo import sensitivity_cvar_m
>>> data = load_cube()[..., :7]
>>> weights = [0.25, 0.18, 0.13, 0.11, 0.24, 0.05, 0.04]
>>> freq = 'quarterly'
>>> shock = 0.05  # 5% absolute shock
>>> sensitivity_cvar_m(data, weights, shock)
Asset_1   -0.485220
Asset_2   -0.542988
Asset_3   -0.492411
Asset_4   -0.478581
Asset_5   -0.459786
Asset_6   -0.467798
Asset_7   -0.468692
Name: cvar, dtype: float64
```

1.2.3 Datasets

Data sets contain a series of python data objects to help get the user started on the package.

Datasets API

Data sets contain a series of python data objects to help get the user started on the package.

Load Sample Simulation Data Cube

perfana.datasets.base.**load_cube**(*, *download=False*)

Loads a sample Monte Carlo simulation of 9 asset classes.

The dimension of the cube is 80 * 1000 * 9. The first axis represents the time, the second represents the number of trials (simulations) and the third represents each asset class.

Parameters **download** (*bool*) – If True, forces the data to be downloaded again from the repository. Otherwise, loads the data from the stash folder

Returns A data cube of simulated returns

Return type ndarray

Load ETF Data

perfana.datasets.base.**load_etf**(*, *date_as_index=True*, *download=False*)

Dataset contains prices of 4 ETF ranging from 2001-06-15 to 2019-03-01.

Parameters

- **date_as_index** (bool) – If True, sets the first column as the index of the DataFrame
- **download** (*bool*) – If True, forces the data to be downloaded again from the repository. Otherwise, loads the data from the stash folder

Returns A data frame containing the prices of 4 ETF

Return type DataFrame

Load Swiss Market Index

perfana.datasets.base.**load_smi**(*, *as_returns=False*, *download=False*)

Dataset contains the close prices of all 20 constituents of the Swiss Market Index (SMI) from 2011-09-09 to 2012-03-28.

Parameters

- **as_returns** (*bool*) – If true, transforms the price data to returns data
- **download** (*bool*) – If True, forces the data to be downloaded again from the repository. Otherwise, loads the data from the stash folder

Returns A data frame of the closing prices of all 20 constituents of the Swiss Market Index

Return type DataFrame

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

V