FinQuant Documentation

Release 0.2.2

Frank Milthaler

May 03, 2020

Contents

FinQuant is a program for financial portfolio management, analysis and optimisation. It is designed to generate an object that holds your data, e.g. stock prices of different stocks, which automatically computes the most common quantities, such as *Expected annual Return*, *Volatility* and *Sharpe Ratio*. Moreover, it provides a library for computing different kinds of *Returns* and visualising *Moving Averages* and *Bollinger Bands*. Finally, given a set of stocks, it also allows for finding optimised portfolios.

FinQuant is made to be easily extended. I hope it proves itself useful for hobby investors, students, geeks, and the intellectual curious.

Caution: While *FinQuant* has tests in place that are run automatically by Travis CI, it cannot guarantee to be bug free, nor that the analysis or optimised portfolio yield to wealth. Please use at your own discretion and refer to the *License*.

CHAPTER 1

Installation

As it is common for open-source projects, there are several ways to get hold of the code. Choose whichever suits you and your purposes best.

1.1 Dependencies

FinQuant depends on the following Python packages:

- `python>=3.5.0`
- `numpy>=1.15`
- `pandas>=0.24`
- `matplotlib>=1.5.1`
- `quandl>=3.4.5`
- `yfinance>=0.1.43`
- `scipy>=1.2.0`
- `pytest>=2.8.7`

1.2 From PyPI

FinQuant can be obtained from PyPI:

```
pip install FinQuant
```

1.3 From GitHub

Get the code from GitHub:

```
git clone https://github.com/fmilthaler/FinQuant.git
```

Then inside `FinQuant` run:

```
python setup.py install
```

Alternatively, if you do not wish to install *FinQuant*, you can also download/clone it as stated above, and then make sure to add it to your `PYTHONPATH`.

CHAPTER 2

Table of Contents

2.1 Quick Start

This section covers some quick examples of *FinQuant*'s features. For a full overview please continue with the documentation, and/or have a look at *Examples*.

2.1.1 Building a Portfolio

Getting an object of `Portfolio` that holds stock prices of four different stocks, as well as its properties and interfaces to optimisation methods is as simple as:

```
from finquant.portfolio import build_portfolio
names = ['GOOG', 'AMZN', 'MCD', 'DIS']
pf = build_portfolio(names=names)
```

The above uses *Quandl* in the background to download the requested data. For more information on *Quandl*, please refer to quandl.

If preferred, *FinQuant* also allows to fetch stock price data from Yahoo Finance. The code snippet below is the equivalent to the above, but using yfinance instead (default value for `data_api` is `"quandl"`):

```
from finquant.portfolio import build_portfolio
names = ['GOOG', 'AMZN', 'MCD', 'DIS']
pf = build_portfolio(names=names, data_api="yfinance")
```

Alternatively, if you already are in possession of stock prices you want to analyse/optimise, you can do the following.

```
import pathlib
from finquant.portfolio import build_portfolio
df_data_path = pathlib.Path() / 'data' / 'ex1-stockdata.csv'
df_data = pd.read_csv(df_data_path, index_col='Date', parse_dates=True)
# building a portfolio by providing stock data
pf = build_portfolio(data=df_data)
```

For this to work, the data is required to be a `pandas.DataFrame` with stock prices as columns.

2.1.2 Properties of the Portfolio

The portfolio's properties are automatically computed as it is being built. One can have a look at them with

```
pf.properties()
```

which shows

```
------------------------------------------------------------------------
Stocks: GOOG, AMZN, MCD, DIS
Time window/frequency: 252
Risk free rate: 0.005
Portfolio Expected Return: 0.266
Portfolio Volatility: 0.156
Portfolio Sharpe Ratio: 1.674

Skewness:
    GOOG      AMZN      MCD       DIS
0  0.124184  0.087516  0.58698  0.040569

Kurtosis:
    GOOG      AMZN       MCD       DIS
0 -0.751818 -0.856101 -0.602008 -0.892666

Information:
   Allocation  Name
0        0.25  GOOG
1        0.25  AMZN
2        0.25   MCD
3        0.25   DIS
------------------------------------------------------------------------
```

2.1.3 Moving Averages

Moving Averages and *Bollinger Bands* can be computed and visualised with the help of the module `finquant.moving_average`.

Note: When computing/visualising a *band* of Moving Averages, `compute_ma` automatically finds the buy/sell signals based on the minimum/maximum *Moving Average* that were computed and highlights those with arrow up/down markers.

```
from finquant.moving_average import compute_ma, ema
# get stock data for Disney
dis = pf.get_stock("DIS").data.copy(deep=True)
spans = [10, 50, 100, 150, 200]
# computing and visualising a band of moving averages
ma = compute_ma(dis, ema, spans, plot=True)
print(ma.tail())
```

which results in

```
               DIS         10d         50d         100d        150d        200d
Date
2017-12-22  108.67  109.093968  104.810423  103.771618  103.716741  103.640858
2017-12-26  108.12  108.916883  104.940210  103.857724  103.775063  103.685426
2017-12-27  107.64  108.684722  105.046085  103.932621  103.826254  103.724775
2017-12-28  107.77  108.518409  105.152905  104.008608  103.878489  103.765026
2017-12-29  107.51  108.335062  105.245340  104.077943  103.926588  103.802290
```

2.1.4 Portfolio Optimisation

FinQuant allows the optimisation of financial portfolios along the *Efficient Frontier* by minimising a cost/objective function. *FinQuant* uses the Python package `scipy` for the minimisation. Alternatively, a *Monte Carlo* approach is implemented as well. The below demonstrates how *FinQuant* performs such an optimisation and visualisation of the results.

```
# Monte Carlo optimisation
opt_w, opt_res = pf.mc_optimisation(num_trials=5000)
pf.mc_plot_results()
# minimisation to compute efficient frontier and optimal portfolios along it
pf.ef_plot_efrontier()
pf.ef.plot_optimal_portfolios()
# plotting individual stocks
pf.plot_stocks()
```

2.2 Portfolio Management

As mentioned above, *FinQuant* is a program for financial portfolio management, among others. The module `finquant.portfolio` does exactly that.

Note: The impatient reader who simply wants to jump in and start using *FinQuant* is advised to jump to *build_portfolio* and have a look at and play around with the *Examples*.

This module is the **core** of *FinQuant*. It provides

- a public class `Stock` that holds and calculates quantities of a single stock,
- a public class `Portfolio` that holds and calculates quantities of a financial portfolio, which is a collection of Stock instances.
- a public function `build_portfolio()` that automatically constructs and returns an instance of `Portfolio` and instances of `Stock`. The relevant stock data is either retrieved through *quandl/yfinance* or provided by the user as a `pandas.DataFrame` (after loading it manually from disk/reading from file). For an example on how to use it, please read the corresponding docstring, or have a look at the examples in the sub-directory `example`.

The classes `Stock` and `Portfolio` are designed to easily manage your financial portfolio, and make the most common quantitative calculations, such as:

- cumulative returns of the portfolio's stocks
- daily returns of the portfolio's stocks (daily percentage change),

- daily log returns of the portfolio's stocks,
- Expected (annualised) Return,
- Volatility,
- Sharpe Ratio,
- skewness of the portfolio's stocks,
- Kurtosis of the portfolio's stocks,
- the portfolio's covariance matrix.

Furthermore, the constructed portfolio can be optimised for

- minimum Volatility,
- maximum Sharpe Ratio
- minimum Volatility for a given Expected Return
- maximum Sharpe Ratio for a given target Volatility

by either performing a numerical computation to solve a minimisation problem, or by performing a Monte Carlo simulation of *n* trials. The former should be the preferred method for reasons of computational effort and accuracy. The latter is only included for the sake of completeness.

Finally, functions are implemented to generated the following plots:

- Monte Carlo run to find optimal portfolio(s)
- Efficient Frontier
- Portfolio with the minimum Volatility based a numerical optimisation
- Portfolio with the maximum Sharpe Ratio based on a numerical optimisation
- Portfolio with the minimum Volatility for a given Expected Return based on a numerical optimisation
- Portfolio with the maximum Sharpe Ratio for a given target Volatility based on a numerical optimisation
- Individual stocks of the portfolio (Expected Return over Volatility)

2.2.1 Stock

class `finquant.portfolio.`**`Stock`**(*investmentinfo*, *data*)

Object that contains information about a stock/fund. To initialise the object, it requires a name, information about the stock/fund given as one of the following data structures:

- `pandas.Series`
- `pandas.DataFrame`

The investment information can contain as little information as its name, and the amount invested in it, the column labels must be `Name` and `Allocation` respectively, but it can also contain more information, such as

- Year
- Strategy
- CCY
- etc.

It also requires either data, e.g. daily closing prices as a `pandas.DataFrame` or `pandas.Series`. `data` must be given as a `pandas.DataFrame`, and at least one data column is required to containing the closing price, hence it is required to contain one column label `<stock_name> - Adj. Close` which is used to compute the return of investment. However, `data` can contain more data in additional columns.

__init__(*investmentinfo*, *data*)

Input

investmentinfo `pandas.DataFrame` of investment information

data `pandas.DataFrame` of stock price

comp_daily_returns()

Computes the daily returns (percentage change). See `finquant.returns.daily_returns`.

comp_expected_return(*freq=252*)

Computes the Expected Return of the stock.

Input

freq `int` (default: `252`), number of trading days, default value corresponds to trading days in a year

Output

expected_return Expected Return of stock.

comp_volatility(*freq=252*)

Computes the Volatility of the stock.

Input

freq `int` (default: `252`), number of trading days, default value corresponds to trading days in a year

Output

volatility Volatility of stock.

properties()

Nicely prints out the properties of the stock: Expected Return, Volatility, Skewness, Kurtosis as well as the `Allocation` (and other information provided in investmentinfo.)

2.2.2 Portfolio

class finquant.portfolio.**Portfolio**

Object that contains information about a investment portfolio. To initialise the object, it does not require any input. To fill the portfolio with investment information, the function `add_stock(stock)` should be used, in which `stock` is an object of `Stock`.

__init__()

Initiates `Portfolio`.

add_stock(*stock*)

Adds a stock of type `Stock` to the portfolio. Each time `add_stock` is called, the following instance variables are updated:

- `portfolio`: `pandas.DataFrame`, adds a column with information from `stock`
- `stocks`: `dictionary`, adds an entry for `stock`
- `data`: `pandas.DataFrame`, adds a column of stock prices from `stock`

Also, the following instance variables are (re-)computed:

- `expected_return`: Expected Return of the portfolio
- `volatility`: Volatility of the portfolio
- `sharpe`: Sharpe Ratio of the portfolio
- `skew`: Skewness of the portfolio's stocks
- `kurtosis`: Kurtosis of the portfolio's stocks

Input

stock an object of `Stock`

comp_cov()
Compute and return a `pandas.DataFrame` of the covariance matrix of the portfolio.

Output

cov a `pandas.DataFrame` of the covariance matrix of the portfolio.

comp_cumulative_returns()
Computes the cumulative returns of all stocks in the portfolio. See `finquant.returns.cumulative_returns`.

Output

ret a `pandas.DataFrame` of cumulative returns of given stock prices.

comp_daily_log_returns()
Computes the daily log returns of all stocks in the portfolio. See `finquant.returns.daily_log_returns`.

Output

ret a `pandas.DataFrame` of log Returns

comp_daily_returns()
Computes the daily returns (percentage change) of all stocks in the portfolio. See `finquant.returns.daily_returns`.

Output

ret a `pandas.DataFrame` of daily percentage change of Returns of given stock prices.

comp_expected_return (*freq=252*)
Computes the Expected Return of the portfolio.

Input

freq `int` (default: `252`), number of trading days, default value corresponds to trading days in a year.

Output

expected_return `float` the Expected Return of the portfolio.

comp_mean_returns (*freq=252*)
Computes the mean returns based on historical stock price data. See `finquant.returns.historical_mean_return`.

Input

freq `int` (default: `252`), number of trading days, default value corresponds to trading days in a year.

Output

ret a `pandas.DataFrame` of historical mean Returns.

comp_sharpe()
Compute and return the Sharpe Ratio of the portfolio.

Output

sharpe `float`, the Sharpe Ratio of the portfolio

comp_stock_volatility(*freq=252*)
Computes the Volatilities of all the stocks individually

Input

freq `int` (default: `252`), number of trading days, default value corresponds to trading days in a year.

Output

volatilies `pandas.DataFrame` with the individual Volatilities of all stocks of the portfolio.

comp_volatility(*freq=252*)
Computes the Volatility of the given portfolio.

Input

freq `int` (default: `252`), number of trading days, default value corresponds to trading days in a year.

Output

volatility `float` the Volatility of the portfolio.

comp_weights()
Computes and returns a `pandas.Series` of the weights/allocation of the stocks of the portfolio.

Output

weights a `pandas.Series` with weights/allocation of all stocks within the portfolio.

ef_efficient_frontier(*targets=None*)
Interface to `finquant.efficient_frontier.EfficientFrontier.efficient_frontier`.

Gets portfolios for a range of given target Returns. If no targets were provided, the algorithm will find the minimum and maximum Returns of the portfolio's individual stocks, and set the target range according to those values. Results in the Efficient Frontier.

Input

targets `list`/`numpy.ndarray` (default: `None`) of `floats`, range of target Returns.

Output

efrontier `numpy.ndarray` of (Volatility, Return) values.

ef_efficient_return(*target*, *verbose=False*)
Interface to `finquant.efficient_frontier.EfficientFrontier.efficient_return`.

Finds the portfolio with the minimum Volatility for a given target return.

Input

target `float`, the target return of the optimised portfolio.

verbose `boolean` (default= `False`), whether to print out properties or not.

Output

df_weights a `pandas.DataFrame` of weights/allocation of stocks within the optimised portfolio.

ef_efficient_volatility(*target*, *verbose=False*)

Interface to `finquant.efficient_frontier.EfficientFrontier.efficient_volatility`.

Finds the portfolio with the maximum Sharpe Ratio for a given target Volatility.

Input

target `float`, the target Volatility of the optimised portfolio.

verbose `boolean` (default= `False`), whether to print out properties or not.

Output

df_weights a `pandas.DataFrame` of weights/allocation of stocks within the optimised portfolio.

ef_maximum_sharpe_ratio(*verbose=False*)

Interface to `finquant.efficient_frontier.EfficientFrontier.maximum_sharpe_ratio`.

Finds the portfolio with the maximum Sharpe Ratio, also called the tangency portfolio.

Input

verbose `boolean` (default= `False`), whether to print out properties or not.

Output

df_weights a `pandas.DataFrame` of weights/allocation of stocks within the optimised portfolio.

ef_minimum_volatility(*verbose=False*)

Interface to `finquant.efficient_frontier.EfficientFrontier.minimum_volatility`.

Finds the portfolio with the minimum Volatility.

Input

verbose `boolean` (default= `False`), whether to print out properties or not.

Output

df_weights a `pandas.DataFrame` of weights/allocation of stocks within the optimised portfolio.

ef_plot_efrontier()

Interface to `finquant.efficient_frontier.EfficientFrontier.plot_efrontier`.

Plots the Efficient Frontier.

ef_plot_optimal_portfolios()

Interface to `finquant.efficient_frontier.EfficientFrontier.plot_optimal_portfolios`.

Plots markers of the optimised portfolios for

- minimum Volatility, and
- maximum Sharpe Ratio.

get_stock(*name*)
Returns the instance of `Stock` with name `name`.

Input

name `string` of the name of the stock that is returned. Must match one of the labels in the dictionary `self.stocks`.

Output

stock instance of `Stock`.

mc_optimisation(*num_trials=1000*)
Interface to `finquant.monte_carlo.MonteCarloOpt.optimisation`.

Optimisation of the portfolio by performing a Monte Carlo simulation.

Input

num_trials `int` (default: `1000`), number of portfolios to be computed, each with a random distribution of weights/allocation in each stock.

Output

opt_w `pandas.DataFrame` with optimised investment strategies for maximum Sharpe Ratio and minimum Volatility.

opt_res `pandas.DataFrame` with Expected Return, Volatility and Sharpe Ratio for portfolios with minimum Volatility and maximum Sharpe Ratio.

mc_plot_results()
Plots the results of the Monte Carlo run, with all of the randomly generated weights/portfolios, as well as markers for the portfolios with the

- minimum Volatility, and
- maximum Sharpe Ratio.

mc_properties()
Calculates and prints out Expected annualised Return, Volatility and Sharpe Ratio of optimised portfolio.

plot_stocks(*freq=252*)
Plots the Expected annual Returns over annual Volatility of the stocks of the portfolio.

Input

freq `int` (default: `252`), number of trading days, default value corresponds to trading days in a year.

properties()
Nicely prints out the properties of the portfolio:

- Expected Return,
- Volatility,
- Sharpe Ratio,
- skewness,
- Kurtosis

as well as the allocation of the stocks across the portfolio.

2.2.3 build_portfolio

`portfolio.`**`build_portfolio`**`()`

This function builds and returns an instance of `Portfolio` given a set of input arguments.

Input

pf_allocation (optional) `pandas.DataFrame` with the required data column labels `Name` and `Allocation` of the stocks. If not given, it is automatically generated with an equal weights for all stocks in the resulting portfolio.

names (optional) A `string` or `list` of `strings`, containing the names of the stocks, e.g. "GOOG" for Google.

start_date (optional) `string`/`datetime` start date of stock data to be requested through *quandl*/*yfinance* (default: `None`).

end_date (optional) `string`/`datetime` end date of stock data to be requested through *quandl*/*yfinance* (default: `None`).

data (optional) A `pandas.DataFrame` which contains quantities of the stocks listed in `pf_allocation`.

data_api (optional) A `string` (default: `quandl`) which determines how to obtain stock prices, if data is not provided by the user. Valid values:

- `quandl` (Python package/API to *Quandl*)
- `yfinance` (Python package formerly known as `fix-yahoo-finance`)

Output

pf Instance of `Portfolio` which contains all the information requested by the user.

Note: Only the following combinations of inputs are allowed:

- `names`, `pf_allocation` (optional), `start_date` (optional), `end_date` (optional), data_api (optional)
- `data`, `pf_allocation` (optional)

The two different ways this function can be used are useful for:

1. building a portfolio by pulling data from *quandl*/*yfinance*,
2. building a portfolio by providing stock data which was obtained otherwise, e.g. from data files.

If used in an unsupported way, the function (or subsequently called function) raises appropriate Exceptions with useful information what went wrong.

2.3 Expected Return, Volatility, Sharpe Ratio

The *Expected Return*, *Volatility* and *Sharpe Ratio* of a portfolio are computed with the module `finquant.quants`.

The module provides functions to compute quantities relevant to financial portfolios, e.g. a weighted average, which is the expected value/return, a weighted standard deviation (volatility), and the Sharpe ratio.

finquant.quants.**annualised_portfolio_quantities**(*weights*, *means*, *cov_matrix*, *risk_free_rate=0.005*, *freq=252*)

Computes and returns the expected annualised return, volatility and Sharpe Ratio of a portfolio.

Input

weights numpy.ndarray/pd.Series of weights

means numpy.ndarray/pd.Series of mean/average values

cov_matrix numpy.ndarray/pandas.DataFrame, covariance matrix

risk_free_rate float (default= 0.005), risk free rate

freq int (default= 252), number of trading days, default value corresponds to trading days in a year

Output

(Expected Return, Volatility, Sharpe Ratio) tuple of those three quantities

finquant.quants.**sharpe_ratio**(*exp_return*, *volatility*, *risk_free_rate=0.005*)

Computes the Sharpe Ratio

Input

exp_return int/float, Expected Return of a portfolio

volatility int/float, Volatility of a portfolio

risk_free_rate int/float (default= 0.005), risk free rate

Output

sharpe ratio float (exp_return - risk_free_rate)/float(volatility)

finquant.quants.**weighted_mean**(*means*, *weights*)

Computes the weighted mean/average, or in the case of a financial portfolio, it can be used for the Expected Return of said portfolio.

Input

means numpy.ndarray/pd.Series of mean/average values

weights numpy.ndarray/pd.Series of weights

Output

weighted mu numpy.ndarray: (np.sum(means*weights))

finquant.quants.**weighted_std**(*cov_matrix*, *weights*)

Computes the weighted standard deviation, or Volatility of a portfolio, which contains several stocks.

Input

cov_matrix numpy.ndarray/pandas.DataFrame, covariance matrix

weights numpy.ndarray/pd.Series of weights

Output

weighted sigma numpy.ndarray: np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))

2.4 Daily and Historical Returns

Returns are implemented in `finquant.returns`.

The module provides functions to compute different kinds of returns of stocks.

`finquant.returns.`**`cumulative_returns`**(*data*, *dividend=0*)

Returns DataFrame with cumulative returns

$$R = \frac{\text{price}_{t_i} - \text{price}_{t_0} + \text{dividend}}{\text{price}_{t_0}}$$

Input

data `pandas.DataFrame` with daily stock prices

dividend `float` (default= `0`), paid dividend

Output

ret a `pandas.DataFrame` of cumulative Returns of given stock prices.

`finquant.returns.`**`daily_log_returns`**(*data*)

Returns DataFrame with daily log returns

$$R_{\log} = \log\left(1 + \frac{\text{price}_{t_i} - \text{price}_{t_{i-1}}}{\text{price}_{t_{i-1}}}\right)$$

Input

data `pandas.DataFrame` with daily stock prices

Output

ret a `pandas.DataFrame` of log(1 + daily percentage change of Returns)

`finquant.returns.`**`daily_returns`**(*data*)

Returns DataFrame with daily returns (percentage change)

$$R = \frac{\text{price}_{t_i} - \text{price}_{t_{i-1}}}{\text{price}_{t_{i-1}}}$$

Input

data `pandas.DataFrame` with daily stock prices

Output

ret a `pandas.DataFrame` of daily percentage change of Returns of given stock prices.

`finquant.returns.`**`historical_mean_return`**(*data*, *freq=252*)

Returns the mean return based on historical stock price data.

Input

data `pandas.DataFrame` with daily stock prices

freq `int` (default= `252`), number of trading days, default value corresponds to trading days in a year

Output

ret a `pandas.DataFrame` of historical mean Returns.

2.5 Moving Average

Moving Averages are implemented in `finquant.moving_average`.

The module provides functions to compute and visualise:

- *Simple Moving Averages*,
- *Exponential Moving Averages*,
- a *band* of *Moving Averages* (simple or exponential), and
- *Bollinger Bands*.

`finquant.moving_average.`**`compute_ma`**(*data*, *fun*, *spans*, *plot=True*)

Computes a band of moving averages (sma or ema, depends on the input argument *fun*) for a number of different time windows. If *plot* is *True*, it also computes and sets markers for buy/sell signals based on crossovers of the Moving Averages with the shortest/longest spans.

Input

data pandas.DataFrame with stock prices, only one column is expected.

fun function that computes a moving average, e.g. sma (simple) or ema (exponential).

spans list of integers, time windows to compute the Moving Average on.

plot boolean (default: True), whether to plot the moving averages and buy/sell signales based on crossovers of shortest and longest moving average.

Output

ma pandas.DataFrame with moving averages of given data.

`finquant.moving_average.`**`ema`**(*data*, *span=100*)

Computes and returns the exponential moving average.

Note: the moving average is computed on all columns.

Input

data pandas.DataFrame with stock prices in columns

span int (defaul: 100), number of days/values over which the average is computed

Output

ema pandas.DataFrame of exponential moving average

`finquant.moving_average.`**`ema_std`**(*data*, *span=100*)

Computes and returns the standard deviation of the exponential moving average.

Input

data pandas.DataFrame with stock prices in columns

span int (defaul: 100), number of days/values over which the average is computed

Output

ema_std pandas.DataFrame of standard deviation of exponential moving average

`finquant.moving_average.`**`plot_bollinger_band`**(*data*, *fun*, *span*)

Computes and visualises a Bolling Band.

Input

data pandas.DataFrame with stock prices in columns

fun function that computes a moving average, e.g. sma (simple) or ema (exponential).

span int (defaul: 100), number of days/values over which the average is computed

finquant.moving_average.**sma**(*data*, *span=100*)

Computes and returns the simple moving average.

Note: the moving average is computed on all columns.

Input

data pandas.DataFrame with stock prices in columns

span int (defaul: 100), number of days/values over which the average is computed

Output

sma pandas.DataFrame of simple moving average

finquant.moving_average.**sma_std**(*data*, *span=100*)

Computes and returns the standard deviation of the simple moving average.

Input

data pandas.DataFrame with stock prices in columns

span int (defaul: 100), number of days/values over which the average is computed

Output

sma_std pandas.DataFrame of standard deviation of simple moving average

2.6 Efficient Frontier

FinQuant allows to optimise a given portfolio by minimising a cost/objective function. The module `finquant.efficient_frontier` contains a class `EfficientFrontier` that provides public functions to compute and visualise optimised portfolios.

class finquant.efficient_frontier.**EfficientFrontier**(*mean_returns*, *cov_matrix*, *risk_free_rate=0.005*, *freq=252*, *method='SLSQP'*)

An object designed to perform optimisations based on minimising a cost/objective function. It can find parameters for portfolios with

- minimum volatility
- maximum Sharpe ratio
- minimum volatility for a given target return
- maximum Sharpe ratio for a given target volatility

It also provides functions to compute the Efficient Frontier between a range of Returns, plot the Efficient Frontier, plot the optimal portfolios (minimum Volatility and maximum Sharpe Ratio).

__init__(*mean_returns*, *cov_matrix*, *risk_free_rate=0.005*, *freq=252*, *method='SLSQP'*)

Input

mean_returns `pandas.Series`, individual expected returns for all stocks in the portfolio

cov_matrix `pandas.DataFrame`, covariance matrix of returns

risk_free_rate `int/float` (default= `0.005`), risk free rate

freq `int` (default= `252`), number of trading days, default value corresponds to trading days in a year

method `string` (default= `"SLSQP"`), type of solver method to use, must be one of:

- 'Nelder-Mead'
- 'Powell'
- 'CG'
- 'BFGS'
- 'Newton-CG'
- 'L-BFGS-B'
- 'TNC'
- 'COBYLA'
- 'SLSQP'
- 'trust-constr'
- 'dogleg'
- 'trust-ncg'
- 'trust-exact'
- 'trust-krylov'

all of which are officially supported by scipy.optimize.minimize

efficient_frontier(*targets=None*)

Gets portfolios for a range of given target returns. If no targets were provided, the algorithm will find the minimum and maximum returns of the portfolio's individual stocks, and set the target range according to those values. Results in the Efficient Frontier.

Input

targets `list/numpy.ndarray` (default= `None`) of `floats`, range of target returns.

Output

efrontier `numpy.ndarray` of (volatility, return) values

efficient_return(*target*, *save_weights=True*)

Finds the portfolio with the minimum volatility for a given target return.

Input

target `float`, the target return of the optimised portfolio.

save_weights `boolean` (default= `True`), for internal use only. Whether to save the optimised weights in the instance variable `weights` (and `df_weights`). Useful for the case of computing the efficient frontier after doing an optimisation, else the optimal weights would be overwritten by the efficient frontier computations. Best to ignore this argument.

Output

df_weights

- if "save_weights" is True: a `pandas.DataFrame` of weights/allocation of stocks within the optimised portfolio.

weights

- if "save_weights" is False: a `numpy.ndarray` of weights/allocation of stocks within the optimised portfolio.

`efficient_volatility`(*target*)

Finds the portfolio with the maximum Sharpe ratio for a given target volatility.

Input

target `float`, the target volatility of the optimised portfolio.

Output

df_weights a `pandas.DataFrame` of weights/allocation of stocks within the optimised portfolio.

`maximum_sharpe_ratio`(*save_weights=True*)

Finds the portfolio with the maximum Sharpe Ratio, also called the tangency portfolio.

Input

save_weights `boolean` (default= `True`), for internal use only. Whether to save the optimised weights in the instance variable `weights` (and `df_weights`). Useful for the case of computing the efficient frontier after doing an optimisation, else the optimal weights would be overwritten by the efficient frontier computations. Best to ignore this argument.

Output

df_weights

- if "save_weights" is True: a `pandas.DataFrame` of weights/allocation of stocks within the optimised portfolio.

weights

- if "save_weights" is False: a `numpy.ndarray` of weights/allocation of stocks within the optimised portfolio.

`minimum_volatility`(*save_weights=True*)

Finds the portfolio with the minimum volatility.

Input

save_weights `boolean` (default= `True`), for internal use only. Whether to save the optimised weights in the instance variable `weights` (and `df_weights`). Useful for the case of computing the efficient frontier after doing an optimisation, else the optimal weights would be overwritten by the efficient frontier computations. Best to ignore this argument.

Output

df_weights

- if "save_weights" is True: a `pandas.DataFrame` of weights/allocation of stocks within the optimised portfolio.

weights

- if "save_weights" is False: a `numpy.ndarray` of weights/allocation of stocks within the optimised portfolio.

plot_efrontier()
Plots the Efficient Frontier.

plot_optimal_portfolios()
Plots markers of the optimised portfolios for

- minimum Volatility, and
- maximum Sharpe Ratio.

properties(*verbose=False*)
Calculates and prints out Expected annualised Return, Volatility and Sharpe Ratio of optimised portfolio.

Input

verbose `boolean` (default= `False`), whether to print out properties or not

2.7 Monte Carlo

The *Monte Carlo* method is implemented in `finquant.monte_carlo`.

The module provides a class `MonteCarlo` which is an implementation of the Monte Carlo method and a class `MonteCarloOpt` which allows the user to perform a Monte Carlo run to find optimised financial portfolios, given an intial portfolio.

class finquant.monte_carlo.**MonteCarlo**(*num_trials=1000*)
An object to perform a Monte Carlo run/simulation.
__init__(*num_trials=1000*)

Input

num_trials `int` (default: `1000`), number of iterations of the Monte Carlo run/simulation.

run(*fun*, ***kwargs*)

Input

fun Function to call at each iteration of the Monte Carlo run.

kwargs (optional) Additional arguments that are passed to *fun*.

Output

result List of quantities returned from *fun* at each iteration.

class finquant.monte_carlo.**MonteCarloOpt**(*returns*, *num_trials=1000*, *risk_free_rate=0.005*, *freq=252*, *initial_weights=None*)
An object to perform a Monte Carlo run/simulation for finding optimised financial portfolios.

Inherits from *MonteCarlo*.
__init__(*returns*, *num_trials=1000*, *risk_free_rate=0.005*, *freq=252*, *initial_weights=None*)

Input

returns A `pandas.DataFrame` which contains the returns of stocks. Note: If applicable, the given returns should be computed with the same risk free rate and time window/frequency (arguments `risk_free_rate` and `freq` as passed down here.

num_trials `int` (default: `1000`), number of portfolios to be computed, each with a random distribution of weights/allocation in each stock

risk_free_rate `float` (default: `0.005`), the risk free rate as required for the Sharpe Ratio

freq `int` (default: `252`), number of trading days, default value corresponds to trading days in a year

initial_weights `list`/`numpy.ndarray` (default: `None`), weights of initial/given portfolio, only used to plot a marker for the initial portfolio in the optimisation plot.

Output

opt `pandas.DataFrame` with optimised investment strategies for maximum Sharpe Ratio and minimum volatility.

optimisation()
Optimisation of the portfolio by performing a Monte Carlo simulation.

Output

opt_w `pandas.DataFrame` with optimised investment strategies for maximum Sharpe Ratio and minimum volatility.

opt_res `pandas.DataFrame` with Expected Return, Volatility and Sharpe Ratio for portfolios with minimum Volatility and maximum Sharpe Ratio.

plot_results()
Plots the results of the Monte Carlo run, with all of the randomly generated weights/portfolios, as well as markers for the portfolios with the minimum Volatility and maximum Sharpe Ratio.

properties()
Prints out the properties of the Monte Carlo optimisation.

2.8 Examples

For more information about the project and details on how to use it, please look at the examples discussed below.

Note: In the below examples, `pf` refers to an instance of `finquant.portfolio.Portfolio`, the object that holds all stock prices and computes its most common quantities automatically. To make *FinQuant* a user-friendly program, that combines data analysis, visualisation and optimisation, the object also provides interfaces to the main features that are provided in the modules in `./finquant/` and are discussed throughout this documentation.

2.8.1 Building a portfolio with data from web *quandl*/*yfinance*

This example shows how to use *FinQuant* to build a financial portfolio by downloading stock price data by using the Python package *quandl*/*yfinance*.

Note: This example refers to `example/Example-Build-Portfolio-from-web.py` of the GitHub repository. It can be downloaded with jupyter notebook cell information: `download Example-Build-Portfolio-from-web.py`

```
# # Building a portfolio with data from `quandl`/`yfinance`
# ## Building a portfolio with `build_portfolio()` by downloading relevant data␣
↪through `quandl`/`yfinance` with stock names, start and end date and column labels
# This example only focuses on how to use `build_portfolio()` to get an instance of␣
↪`Portfolio` by providing minimal information that is passed on to `quandl`/
↪`yfinance`. For a more exhaustive description of this package and example, please␣
↪try `Example-Analysis` and `Example-Optimisation`.

import pandas as pd
import datetime

# importing some custom functions/objects
from finquant.portfolio import build_portfolio

# ## Get data from `quandl`/`yfinance` and build portfolio
# First we need to build a pandas.DataFrame that holds relevant data for our␣
↪portfolio. The minimal information needed are stock names and the amount of money␣
↪to be invested in them, e.g. Allocation.

# To play around yourself with different stocks, here is a short list of companies␣
↪and their tickers
# d = {0: {'Name':'WIKI/GOOG', 'Allocation':20},  # Google
#      1: {'Name':'WIKI/AMZN', 'Allocation':33},  # Amazon
#      2: {'Name':'WIKI/MSFT', 'Allocation':18},  # Microsoft
#      3: {'Name':'WIKI/AAPL', 'Allocation':10},  # Apple
#      4: {'Name':'WIKI/KO', 'Allocation':15},    # Coca-Cola
#      5: {'Name':'WIKI/XOM', 'Allocation':11},   # Exxon Mobil
#      6: {'Name':'WIKI/JPM', 'Allocation':21},   # JP Morgan
#      7: {'Name':'WIKI/DIS', 'Allocation':9},    # Disney
#      8: {'Name':'WIKI/MCD', 'Allocation':23},   # McDonald's
#      9: {'Name':'WIKI/WMT', 'Allocation':3},    # Walmart
#     10: {'Name':'WIKI/YHOO', 'Allocation':7},   # Yahoo
#     11: {'Name':'WIKI/GS', 'Allocation':9},     # Goldman Sachs
#     }

d = {
    0: {"Name": "WIKI/GOOG", "Allocation": 20},
    1: {"Name": "WIKI/AMZN", "Allocation": 10},
    2: {"Name": "WIKI/MCD", "Allocation": 15},
    3: {"Name": "WIKI/DIS", "Allocation": 18},
}
# If you wish to use Yahoo Finance as source, you must remove "WIKI/" from the stock␣
↪names/tickers

pf_allocation = pd.DataFrame.from_dict(d, orient="index")

# ### User friendly interface to quandl/yfinance
# As mentioned above, in this example `build_portfolio()` is used to build a␣
↪portfolio by performing a query to `quandl`/`yfinance`.
#
# To download Google's stock data, `quandl` requires the string `"WIKI/GOOG"`. For␣
↪simplicity, `FinQuant` facilitates a set of functions under the hood to sort out␣
↪lots of specific commands/required input for `quandl`/`yfinance`. When using␣
↪`FinQuant`, the user simply needs to provide a list of stock names/tickers.
# For example, if using `quandl` as a data source (default), a list of names/tickers␣
↪as shown below is a valid input for `FinQuant`'s function `build_
↪portfolio(names=names)`:
```

(continues on next page)

(continued from previous page)

```
#  * `names = ["WIKI/GOOG", "WIKI/AMZN"]`
#
# If using `yfinance` as a data source, `FinQuant`'s function `build_
↪portfolio(names=names)` expects the stock names to be without any leading/trailing␣
↪string (check Yahoo Finance for correct stock names):
#  * `names = ["GOOG", "AMZN"]`
#
# By default, `FinQuant` uses `quandl` to obtain stock price data. The function␣
↪`build_portfolio()` can be called with the optional argument `data_api` to use␣
↪`yfinance` instead:
#  * `build_portfolio(names=names, data_api="yfinance")`
#
# In the below example we are using the default option, `quandl`.

# here we set the list of names based on the names in
# the DataFrame pf_allocation
names = pf_allocation["Name"].values.tolist()

# dates can be set as datetime or string, as shown below:
start_date = datetime.datetime(2015, 1, 1)
end_date = "2017-12-31"

# While quandl/yfinance will download lots of different prices for each stock,
# e.g. high, low, close, etc, FinQuant will extract the column "Adj. Close" ("Adj␣
↪Close" if using yfinance).

pf = build_portfolio(
    names=names, pf_allocation=pf_allocation, start_date=start_date, end_date=end_date
)

# ## Portfolio is successfully built
# Getting data from the portfolio

# the portfolio information DataFrame
print(pf.portfolio)

# the portfolio stock data, prices DataFrame
print(pf.data.head(3))

# print out information and quantities of given portfolio
print(pf)
pf.properties()

# ## Please continue with `Example-Build-Portfolio-from-file.py`.
# As mentioned above, this example only shows how to use `build_portfolio()` to get␣
↪an instance of `Portfolio` by downloading data through `quandl`/`yfinance`.
```

2.8.2 Building a portfolio with preset data

This example shows how to use *FinQuant* to build a financial portfolio by providing stock price data yourself, e.g. by reading data from disk/file.

Note: This example refers to `example/Example-Build-Portfolio-from-file.py` of the GitHub repository. It can be downloaded with jupyter notebook cell information: `download`

Example-Build-Portfolio-from-file.py

```
# # Building a portfolio with data from disk
# ## Building a portfolio with `build_portfolio()` with data obtained from data files.
# Note: The stock data is provided in two data files. The stock data was previously pulled from quandl.

import pathlib
import pandas as pd
import datetime

# importing FinQuant's function to automatically build the portfolio
from finquant.portfolio import build_portfolio

# ### Get data from disk/file
# Here we use `pandas.read_cvs()` method to read in the data.

# stock data was previously pulled from quandl and stored in ex1-stockdata.csv
# commands used to save data:
# pf.portfolio.to_csv("ex1-portfolio.csv", encoding='utf-8', index=False, header=True)
# pf.data.to_csv("ex1-stockdata.csv", encoding='utf-8', index=True, index_label="Date")
# read data from files:
df_pf_path = pathlib.Path.cwd() / ".." / "data" / "ex1-portfolio.csv"
df_data_path = pathlib.Path.cwd() / ".." / "data" / "ex1-stockdata.csv"
df_pf = pd.read_csv(df_pf_path)
df_data = pd.read_csv(df_data_path, index_col="Date", parse_dates=True)

# ### Examining the DataFrames

print(df_pf)

print(df_data.head(3))

# ## Building a portfolio with `build_portfolio()`
# `build_portfolio()` is an interface that can be used in different ways. Two of which is shown below. For more information the docstring is shown below as well.
# In this example `build_portfolio()` is being passed `df_data`, which was read in from file above.

print(build_portfolio.__doc__)

# ## Building a portfolio with data only
# Below is an example of only passing a `DataFrame` containing data (e.g. stock prices) to `build_portfolio()` in order to build an instance of `Portfolio`. In this case, the allocation of stocks is automatically generated by equally distributing the weights across all stocks.

# building a portfolio by providing stock data
pf = build_portfolio(data=df_data)

# ### Portfolio is successfully built
# Below it is shown how the allocation of the stocks and the data (e.g. prices) of the stocks can be obtained from the object `pf`.

# the portfolio information DataFrame
print(pf.portfolio.name)
```

(continues on next page)

(continued from previous page)

```
print(pf.portfolio)

# the portfolio stock data, prices DataFrame
print(pf.data.head(3))

# ## Building a portfolio with data and desired allocation
# If a specific allocation of stocks in the portfolio is desired, a `DataFrame` such as `df_pf` (which was read from file above) can be passed to `build_portfolio()` as shown below.

# building a portfolio by providing stock data
# and a desired allocation
pf2 = build_portfolio(data=df_data, pf_allocation=df_pf)

# the portfolio information DataFrame
print(pf2.portfolio.name)
print(pf2.portfolio)

# the portfolio stock data, prices DataFrame
print(pf2.data.head(3))
```

2.8.3 Analysis of a portfolio

This example shows how to use an instance of `finquant.portfolio.Portfolio`, get the portfolio's quantities, such as

- Expected Returns,
- Volatility,
- Sharpe Ratio.

It also shows how to extract individual stocks from the given portfolio. Moreover it shows how to compute and visualise:

- the different Returns provided by the module `finquant.returns`,
- *Moving Averages*, a band of *Moving Averages*, and a *Bollinger Band*.

Note: This example refers to `example/Example-Analysis.py` of the GitHub repository. It can be downloaded with jupyter notebook cell information: `download Example-Analysis.py`

```
# # Example:
# ## Building a portfolio with `build_portfolio()` with data obtained from data files.
# Note: The stock data is provided in two data files. The stock data was previously pulled from quandl.

import pathlib
import matplotlib.pyplot as plt
import pandas as pd
import datetime

# importing FinQuant's function to automatically build the portfolio
from finquant.portfolio import build_portfolio

```

(continues on next page)

(continued from previous page)

```
# ## Building a portfolio with `build_portfolio()`
# As in previous example, using `build_portfolio()` to generate an object of `Portfolio`.

# read data from files:
df_data_path = pathlib.Path.cwd() / ".." / "data" / "ex1-stockdata.csv"
df_data = pd.read_csv(df_data_path, index_col="Date", parse_dates=True)
# building a portfolio by providing stock data
pf = build_portfolio(data=df_data)

# ## Expected Return, Volatility and Sharpe Ratio of Portfolio
# The annualised expected return and volatility as well as the Sharpe Ratio are automatically computed. They are obtained as shown below.
# The expected return and volatility are based on 252 trading days by default. The Sharpe Ratio is computed with a risk free rate of 0.005 by default.

# expected (annualised) return
print(pf.expected_return)

# volatility
print(pf.volatility)

# Sharpe ratio (computed with a risk free rate of 0.005 by default)
print(pf.sharpe)

# ## Getting Skewness and Kurtosis of the stocks

print(pf.skew)

print(pf.kurtosis)

# ## Nicely printing out portfolio quantities
# To print the expected annualised return, volatility, Sharpe ratio, skewness and Kurtosis of the portfolio and its stocks, one can simply do `pf.properties()`.

print(pf)
pf.properties()

# ## Daily returns and log returns
# `FinQuant` provides functions to compute daily returns and annualised mean returns of a given DataFrame in various ways.

# annualised mean returns
print(pf.comp_mean_returns())

# daily returns (percentage change)
print(pf.comp_cumulative_returns().head(3))

print(pf.comp_daily_log_returns().head(3))

# plotting stock data of portfolio
pf.data.plot()
plt.show()

# The stock prices of Google and Amazon are much higher than those for McDonald's and Disney. Hence the fluctuations of the latter ones are barely seen in the above plot.
# One can use `pandas.plot()` method to create a secondary y axis.
```

(continues on next page)

(continued from previous page)

```

pf.data.plot(secondary_y=["WIKI/MCD", "WIKI/DIS"], grid=True)
plt.show()

# plotting cumulative returns (price_{t} - price_{t=0}) / price_{t=0}
pf.comp_cumulative_returns().plot().axhline(y=0, color="black", lw=3)
plt.show()

# plotting daily percentage changes of returns
pf.comp_daily_returns().plot().axhline(y=0, color="black")
plt.show()

# plotting daily log returns
pf.comp_daily_log_returns().plot().axhline(y=0, color="black")
plt.show()

# cumulative log returns
pf.comp_daily_log_returns().cumsum().plot().axhline(y=0, color="black")
plt.show()

# ## Moving Averages
# `FinQuant` provides a module `finquant.moving_average` to compute moving averages. See below.

from finquant.moving_average import sma

# simple moving average
ax = pf.data.plot(secondary_y=["WIKI/MCD", "WIKI/DIS"], grid=True)
# computing simple moving average over a span of 50 (trading) days
# and plotting it
sma(pf.data, span=50).plot(ax=ax, secondary_y=["WIKI/MCD", "WIKI/DIS"], grid=True)
plt.show()

from finquant.moving_average import ema

# exponential moving average
ax = pf.data.plot(secondary_y=["WIKI/MCD", "WIKI/DIS"], grid=True)
# computing exponential moving average and plotting it
ema(pf.data).plot(ax=ax, secondary_y=["WIKI/MCD", "WIKI/DIS"])
plt.show()

# ## Band of moving averages and Buy/Sell signals
# `FinQuant` also provides a method `finquant.moving_average.compute_ma` that automatically computes and plots several moving averages. It also **finds buy/sell signals based on crossovers** of the shortest and longest moving average.
# To learn more about it and its input arguments, read its docstring and see the example below.

from finquant.moving_average import compute_ma

print(compute_ma.__doc__)

# get stock data for disney
dis = pf.get_stock("WIKI/DIS").data.copy(deep=True)
# we want moving averages of 10, 50, 100, and 200 days.
spans = [10, 50, 100, 150, 200]
# compute and plot moving averages
```

(continues on next page)

(continued from previous page)

```
dis_ma = compute_ma(dis, ema, spans, plot=True)
plt.show()

# ## Plot the Bollinger Band of one stock
# The Bollinger Band can be automatically computed and plotted with the method `finquant.moving_average.plot_bollinger_band`. See below for an example.

# plot the bollinger band of the disney stock prices
from finquant.moving_average import plot_bollinger_band

# get stock data for disney
dis = pf.get_stock("WIKI/DIS").data.copy(deep=True)
span = 20
# for simple moving average:
plot_bollinger_band(dis, sma, span)
plt.show()
# for exponential moving average:
plot_bollinger_band(dis, ema, span)
plt.show()

# ## Recomputing expected return, volatility and Sharpe ratio
# **Note**: When doing so, the instance variables for
#  - Expected return
#  - Volatility
#  - Sharpe Ratio
# are automatically recomputed.

# If the return, volatility and Sharpe ratio need to be computed based
# on a different time window and/or risk free rate, one can recompute
# those values as shown below
# 1. set the new value(s)
pf.freq = 100
pf.risk_free_rate = 0.02

# 2.a compute and get new values based on new freq/risk_free_rate
exret = pf.comp_expected_return(freq=100)
vol = pf.comp_volatility(freq=100)
sharpe = pf.comp_sharpe()
print(
    "For {} trading days and a risk free rate of {}:".format(pf.freq, pf.risk_free_rate)
)
print("Expected return: {:0.3f}".format(exret))
print("Volatility: {:0.3f}".format(vol))
print("Sharpe Ratio: {:0.3f}".format(sharpe))

# 2.b print out properties of portfolio (which is based on new freq/risk_free_rate)
pf.properties()

# ## Extracting data of stocks individually
# Each stock (its information and data) of the portfolio is stored as a `Stock` data structure. If needed, one can of course extract the relevant data from the portfolio DataFrame, or access the `Stock` instance. The commands are very similar to the once for `Portfolio`. See below how it can be used.

# getting Stock object from portfolio, for Google's stock
goog = pf.get_stock("WIKI/GOOG")
```

(continues on next page)

(continued from previous page)

```
# getting the stock prices
goog_prices = goog.data
print(goog_prices.head(3))

print(goog.comp_daily_returns().head(3))

print(goog.expected_return)

print(goog.volatility)

print(goog.skew)

print(goog.kurtosis)

print(goog)
goog.properties()

# ## Extracting stock data by date
# Since quandl provides a DataFrame with an index of dates, it is easy to extract data from the portfolio for a given time frame. Three examples are shown below.

print(pf.data.loc[str(datetime.datetime(2015, 1, 2))])

print(pf.data.loc[pf.data.index > datetime.datetime(2016, 1, 2)].head(3))

print(pf.data.loc[pf.data.index.year == 2017].head(3))
```

2.8.4 Optimisation of a portfolio

This example focusses on the optimisation of a portfolio. To achieve this, the example shows the usage of `finquant.efficient_frontier.EfficientFrontier` for numerically optimising the portfolio, for the

- Minimum Volatility
- Maximum Sharpe Ratio
- Minimum Volatility for a given target Return
- Maximum Sharpe Ratio for a given target Volatility.

Furthermore, it is also shown how the entire *Efficient Frontier* and the optimal portfolios can be computed and visualised. If needed, it also gives an example of plotting the individual stocks of the given portfolio within the computed *Efficient Frontier*.

Also, the optimisation of a portfolio and its visualisation based on a *Monte Carlo* is shown.

Finally, *FinQuant*'s visualisation methods allow for overlays, if this is desired. Thus, with only the following few lines of code, one can create an overlay of the *Monte Carlo* run, the *Efficient Frontier*, its optimised portfolios for *Minimum Volatility* and *Maximum Sharpe Ratio*, as well as the portfolio's individual stocks.

Note: This example refers to `example/Example-Optimisation.py` of the GitHub repository. It can be downloaded with jupyter notebook cell information: `download Example-Optimisation.py`

```
# # Example: Portfolio optimisation
# This example shows how `FinQuant` can be used to optimise a portfolio.
```

(continues on next page)

(continued from previous page)

```
# Two different approaches are implemented in `FinQuant`:
#  1. Efficient Frontier
#  2. Monte Carlo run
# With the *Efficient Frontier* approach, the portfolio can be optimised for
#  - minimum volatility,
#  - maximum Sharpe ratio
#  - minimum volatility for a given expected return
#  - maximum Sharpe ratio for a given target volatility
# by performing a numerical solve to minimise/maximise an objective function.
# Alternatively a *Monte Carlo* run of `n` trials can be performed to find the optimal portfolios for
#  - minimum volatility,
#  - maximum Sharpe ratio
# The approach branded as *Efficient Frontier* should be the preferred method for reasons of computational effort and accuracy. The latter approach is only included for the sake of completeness, and creation of beautiful plots.
#
# ## Visualisation
# Not only does `FinQuant` allow for the optimisation of a portfolio with the above mentioned methods and objectives, `FinQuant` also allows for the computation and visualisation of an *Efficient Frontier* and *Monte Carlo* run.
# Let `pf` be an instance of `Portfolio`. The *Efficient Frontier* can be computed and visualised with `pf.ef_plot_efrontier()`. The optimal portfolios for *minimum volatility* and *maximum Sharpe ratio* can be visualised with `pf.ef_plot_optimal_portfolios()`. And if required, the individual stocks of the portfolio can be visualised with `pf.plot_stocks(show=False)`. An overlay of these three commands is shown below.
# Finally, the entire result of a *Monte Carlo* run can also be visualised automatically by `FinQuant`. An example is shown below.

import pathlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import datetime

# importing FinQuant's function to automatically build the portfolio
from finquant.portfolio import build_portfolio

# ### Get data from disk/file
# Here we use `pandas.read_cvs()` method to read in the data.

# stock data was previously pulled from quandl and stored in ex1-stockdata.csv
# read data from files:
df_data_path = pathlib.Path.cwd() / ".." / "data" / "ex1-stockdata.csv"
df_data = pd.read_csv(df_data_path, index_col="Date", parse_dates=True)
# building a portfolio by providing stock data
pf = build_portfolio(data=df_data)
print(pf)
pf.properties()

# # Portfolio optimisation
# ## Efficient Frontier
# Based on the **Efficient Frontier**, the portfolio can be optimised for
#  - minimum volatility
#  - maximum Sharpe ratio
#  - minimum volatility for a given target return
```

(continues on next page)

(continued from previous page)

```
#  - maximum Sharpe ratio for a given target volatility
# See below for an example for each optimisation.

# if needed, change risk free rate and frequency/time window of the portfolio
print("pf.risk_free_rate = {}".format(pf.risk_free_rate))
print("pf.freq = {}".format(pf.freq))

pf.ef_minimum_volatility(verbose=True)

# optimisation for maximum Sharpe ratio
pf.ef_maximum_sharpe_ratio(verbose=True)

# minimum volatility for a given target return of 0.26
pf.ef_efficient_return(0.26, verbose=True)

# maximum Sharpe ratio for a given target volatility of 0.22
pf.ef_efficient_volatility(0.22, verbose=True)

# ### Manually creating an instance of EfficientFrontier
# If required, or preferred, the below code shows how the same is achieved by manually creating an instance of EfficientFrontier, passing it the mean returns and covariance matrix of the previously assembled portfolio.

from finquant.efficient_frontier import EfficientFrontier

# creating an instance of EfficientFrontier
ef = EfficientFrontier(pf.comp_mean_returns(freq=1), pf.comp_cov())
# optimisation for minimum volatility
print(ef.minimum_volatility())

# printing out relevant quantities of the optimised portfolio
(expected_return, volatility, sharpe) = ef.properties(verbose=True)

# ### Computing and visualising the Efficient Frontier
# `FinQuant` offers several ways to compute the *Efficient Frontier*.
#  1. Through the opject `pf`
#    - with automatically setting limits of the *Efficient Frontier*
#  2. By manually creating an instance of `EfficientFrontier` and providing the data from the portfolio
#    - with automatically setting limits of the *Efficient Frontier*
#    - by providing a range of target expected return values
# As before, let `pf` and be an instance of `Portfolio`. The following code snippets compute and plot an *Efficient Frontier* of a portfolio, its optimised portfolios and individual stocks within the portfolio.
#  - `pf.ef_plot_efrontier()`
#  - `pf.ef_plot_optimal_portfolios()`
#  - `pf.plot_stocks()`

# #### Efficient Frontier of `pf`

# computing and plotting efficient frontier of pf
pf.ef_plot_efrontier()
# adding markers to optimal solutions
pf.ef_plot_optimal_portfolios()
# and adding the individual stocks to the plot
pf.plot_stocks()
plt.show()
```

(continues on next page)

(continued from previous page)

```
101
102 # #### Manually creating an Efficient Frontier with target return values
103
104 targets = np.linspace(0.12, 0.45, 50)
105 # computing efficient frontier
106 efficient_frontier = ef.efficient_frontier(targets)
107 # plotting efficient frontier
108 ef.plot_efrontier()
109 # adding markers to optimal solutions
110 pf.ef.plot_optimal_portfolios()
111 # and adding the individual stocks to the plot
112 pf.plot_stocks()
113 plt.show()
114
115 # ## Monte Carlo
116 # Perform a Monte Carlo run to find the portfolio with the minimum volatility and␣
    ↪maximum Sharpe Ratio.
117
118 opt_w, opt_res = pf.mc_optimisation(num_trials=5000)
119 pf.mc_properties()
120 pf.mc_plot_results()
121 # again, the individual stocks can be added to the plot
122 pf.plot_stocks()
123 plt.show()
124
125 print(opt_res)
126 print()
127 print(opt_w)
128
129 # # Optimisation overlay
130 # ## Overlay of Monte Carlo portfolios and Efficient Frontier solutions
131
132 opt_w, opt_res = pf.mc_optimisation(num_trials=5000)
133 pf.mc_plot_results()
134 pf.ef_plot_efrontier()
135 pf.ef.plot_optimal_portfolios()
136 pf.plot_stocks()
137 plt.show()
```

2.9 License

2.10 About

I was inspired to develop this program by recent application procedures in the finance sector. Being tasked with a typical exercise for a *Quant* position at an investment firm, I first started doing some reading about finance and financial portfolios in particular. Successfully completing the exercise felt good, but I had many more ideas on how to improve and extend my solution. The result of which is *FinQuant*.

My academical background is quite diverse, as I studied Mechanical Engineering & Computer Science, Systems & Control Engineering for my undergraduate and postgraduate courses respectively, followed by a doctorate in Computational Physics at Imperial College London. With that skillset I decided to go rogue and enter the field of Data Science/Engineering.

I enjoy challenging myself and learning new things, which is probably the reason for me working on this and other projects.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

Symbols

A

B

C

D

E