[Mineure Finances] DOLPHIN 2017 project

November 27, 2017

Team #4

Alaa BAKHTI (SCIA 2018)
Joseph DIOMANDE (SIGL 2018)
Xavier YVONNE (SCIA 2018)

1 Introduction

1.1 Overview

This project is a robot-advisor, namely an application optimizing a portfolio with given assets. The portfolio is optimized with respect to the Sharpe ratio (see Section 1).
Our program is entirely written in Python 3. To run it, just open the notebook (e.g. type `jupyter notebook` in a shell) named `portfolio_optimizer.ipynb`, and run all cells therein. An Internet connection is required to fetch the asset values through the given API and submit the optimized portfolio.
Everything concerning the API is available at https://dolphin.jump-technology.com:3389/api/v1/ and in the given documentation.

1.2 Goal

The assets to build the portfolio from, are quoted from 2012-01-01 (start date) to 2017-06-30 (end date). There are 528 of them, but to avoid currency conversion issues with rates varying from day to day, we only restrict ourselves to the 283 asserts quoted in EUR (€) - see Section 2.2 for further details about this. For every asset, we have at our disposal its daily pricing, global return (that is, the return between start date and end date), annual return, volatility, value at risk, and Sharpe. The Sharpe ratio S for an asset (and similarly, for a portfolio) is defined as

$$S = (\rho_a - r_f) / \sigma,$$

where

- ρ_a is the annual return,
- r_f is the return of the "risk-free" asset, $r_f = 0.05\%$;
- σ is the volatility, that is the standard deviation of the return divided by the square root of the considered period expressed in years. If we take this period equal to one year, then the volatility is just the standard deviation.

The goal of this project is to select the parts (weights) of all assets to make a portfolio having the best Sharpe, with the following constraints:

- **select exactly 20 different assets ;**
- **each selected asset must have a weight between 1% and 10%.**

Actually, for the submission, we have to post on the server not directly the weight of each selected asset, but its quantity or position, that is, the number of copies of this asset in the portfolio. For an asset, the position Q is a nonnegative integer related to the weight w by

$$w = Q.V / V_{tot},$$

where

- V_{tot} is the value of the whole portfolio, arbitrarily set to (approximately) V_{tot} = 10 M€ ;
- V is the pricing of the asset at date D, D being the end of the day before submission of the portfolio. According to this, D should be 2017-11-26 ; but since the API does not provide quotations up to this date, we take for D the latest available date viz. D = 2017-06-30.

Note that our submission is a portfolio <u>for this only date</u>.

2 API managing

To achieve our goal, we must

1. set up the server,
2. get the necessary data though the API,
3. optimize the portfolio itself,
4. compute its Sharpe,
5. post the portfolio on the server.

2.1 Server setup

We define a `Server` class with the following methods:

- __init__ : the constructor, setting all the attributes. Its mandatory arguments, whose names are self-explicit, are: `server_url`, `user`, and `pwd.`
- `get, post, put:` methods to `GET`, `POST` and `PUT` `http` requests, respectively. The transferred objects are JSON objects. Our methods just call the ones from the `requests` and `json` Python packages with the suitable arguments. For more details about the request bodies and the structure of the JSON objects transferred, please refer to the given documentation.

To start a server, just type `server = Server(server_url, user, pwd)` with the suitable server URL, user and password ids.

2.2 Fetch server data

Once the server is set up, we need to get the financial data to optimize our portfolio. Most of this job is made with our `ApiManager` class. We explain here roughly how to instance this class and what it can do; for a detailed description of its attributes and methods, please refer to the docstring of this class provided in our notebook.

An object of this class must be instanced with:

- the server `server` described in the previous section;
- the `ratio_list` consisting of the list of ratios of interest, namely ratios #15 (Beta), #17 (annual return), #18 (volatility), #19 (Pearson correlation), #20 (Sharpe), #21 (global return), #22 (Value at Risk) and #29 (exposition action);
- `ratio_identifiers_dict`, a dictionary whose keys are the ratio names (strings) and values are the corresponding ratio ids given above;

- `RISK_FREE`, the constant $r_f = 0.05\%$ (see Section 1.2);
- `START_DATE`, the constant string corresponding to the start date '2012-01-01';
- `END_DATE`, the constant string corresponding to the end date '2017-06-30';
- `PORTFOLIO_VALUE`, the desired portfolio value in €, namely V_{tot} = 10 M€.

Warning: instancing an `ApiManager` object takes a long time (several minutes depending on the Internet connection), because there are many data to fetch on the server. During the defense, we shall bring a laptop with an object called `api_manager` already instanced.

Recall that we work only with the 283 asserts quoted in EUR (€). To do this, we set the flag `only_euro=True` when calling the `__get_assets` method of our class (setting this flag to `False` would enable us to work with all the assets).

Given a list `ratio_ids` of ratio ids described above (15, 17, 18 and so on), our `__get_ratio` method provides, for all 283 asserts and dates between the start date and the end date, the list of the corresponding ratio values. In particular, the snippet code

```
self.assets_ratio_list = self.__get_assets_ratio_list(self.START_DATE,
self.END_DATE)
ratio_ids = [ratio_identifiers_dict[x] for x in ['global_return', 'sharpe',
'volatility']]
self.global_returns, self.sharpes, self.volatilities = [x for x in
self.__get_ratio(ratio_ids)]
```

returns 3 lists: `global_returns`, `sharpes` and `volatilities` for all assets, the 3 names being self-explicit.

The daily returns are computed by the `__get_assets_daily_returns` method. The trouble is, the assets are not quoted every day, so the API does not provide all the quotations between the start date and the end date. We discuss how we fixed this in Section 4.2.

Finally, the portfolio we want to optimize is fetched through the `__get_portfolio` method.

2.3 Portfolio optimization (Part I)

Once all the needed data are collected through the API (see the previous section), we are ready to optimize our portfolio. Our `PortfolioOptimizer` class does this. An object of this class must be instanced with only one argument: the `ApiManager` object.

Recall the formula for the Sharpe in Section 1.2. To optimize our portfolio, we need to compute the vector $R = (r_i)$ of the annual returns of the 283 assets, along with the covariance matrix Σ of the returns.

The annual return value r_i for the ith asset could easily be retrieved from the API, just like we did for the global value, but with replacing the ratio id 21 (global return) by 17 (annual return). However, we prefer computing the annual return from the global return with the formula

$$r_i = (1 + \text{global return})^{(365 / \text{period})} - 1;$$

here, period is the number of days between start date and end date — in our case, period = 1642.

Recall that thanks to the API, the daily returns for all assets are known (see the previous section). We can view them as an MxN matrix, with M the number of quotation days (M=1642 if we fill the missing values in a suitable way) and N the number of assets (N=283). We compute the covariance matrix Σ of the returns from this matrix using the `numpy.cov` method.
We now have all the keys to compute the Sharpe of a portfolio with given weights. This is done with our

`evaluate_portfolio` method from the `PortfolioOptimizer` class. Actually, two methods for computing the Sharpe are implemented: the orthodox one described above, and a quicker one that applies the same formula, but replaces the vector $R = (r_i)$ of the annual returns by the vector of the *global* returns. Call the `evaluate_portfolio` method with the flag `is_annuel=True` (default) to compute the Sharpe using the first method, and with `is_annuel=False` to compute the Sharpe using the second.

Of course, the hardest thing still has to be done: compute the weights of the optimized portfolio. The best method we found uses Markowitz theory, but is too long to be described in this section: we postpone its complete description, along with the description of more naive optimization methods, to Section 3. Just mention now that we call the `optimize_portfolio_composition` method, which returns the following:

- `db_ids` : the id list of the 20 assets selected in our portfolio,
- `weights` : the corresponding weights,
- `portfolio_sharpe` : the Sharpe of our portfolio.

Once the weights for our optimized portfolio are found, we can compute the corresponding quantities using the `_get_asset_quantities` method from the `ApiManager` class, as explained in Section 1.2.

2.4 Post the optimized portfolio

In the previous section, we explained how to find the composition (quantities of all assets) of our optimized portfolio. To post it on the server and complete our submission, Just call the `post_portfolio` method from the `ApiManager` class with the following arguments:

- `asset_db_ids` : the id list of the 20 assets selected in our portfolio,
- asset_weights : the corresponding weights,
- `start_date`,
- `end_date`.

Recall from Section 1.2 that due to technical reasons, the date D used to compute quantities from weights for the submitted portfolio is identical to the end date viz. 2017-06-30, so we do not need to pass D as an extra argument.

3 Portfolio optimization (Part II)

3.1 Global approach

Our work relies to a large extent on **Markowitz theory**. Recall that $R = (r_i)$, the vector of the annual returns of all the 283 assets, and $\Sigma = (v_{i,j})$, the covariance matrix of the returns, are known thanks the API (see Sections 2.2 and 2.3).

Let $W= (w_i)$ be the vector of the weights of the 283 assets in the portfolio. Recall that the portfolio return ρ and its volatility σ can be computed by

$$\rho = \Sigma_i w_i r_i = R^T.W, \ \sigma = \Sigma_{i,j} w_i \ w_j v_{i,j} = W^T.\Sigma.W,$$

whence the Sharpe S of the portfolio is given by

$$S = (\rho - r_f) / \sigma = (R - r_f \mathbf{1})^T.W \ / \ W^T.\Sigma.W.$$

(here, **1** is the vector filled only with ones, and $r_f = 0.05\%$). We see that the Sharpe S is a function of the weights W, so our problem becomes:

$$\text{find argmax } S(W) = \text{argmax } (R - r_f\mathbf{1})^T.W \;/\; W^T.\Sigma.W$$

subject to

$$0 \leq W,\ \mathbf{1}^T.W = 1,$$
$$w_i = 0 \text{ or } 1\% \leq w_i \leq 10\%,$$
$$\#\text{Supp}(W) = \#\{ i \mid w_i \neq 0 \} = 20.$$

Supp(W) represents the indexes of the assets that actually contribute to the portfolio, and is called **support** of the portfolio.

Note that maximizing the Sharpe amounts to both minimizing the volatility σ and maximizing the return ρ, which in turn amounts to minimizing

$$f(W) = \sigma - \alpha\rho = W^T.\Sigma.W - \alpha\, R^T.W,$$

subject to the same constraints. Here, α is a positive extra parameter, called `risk_tolerance` in our implementation, that we can fix ourselves. In the sequel, we take $\alpha = 1$. Other values for α have been tested, but do not lead to significant changes.

Finding argmin f(W) is a quadratic problem that can be solved by standard means if the support Supp(W) is known, or if the constraint #Supp(W) = 20 is dropped. This leads us to split the problem in half:

1) determine the support Supp(W), see Section 3.2;
2) minimize f(W) with this support and the other constraints $0 \leq W$, $\mathbf{1}^T.W = 1$, $w_i = 0$ or $1\% \leq w_i \leq 10\%$, see Section 3.3.

Roughly speaking, minimizing f(W) with a given support is made with our `optimizer` method from the `PortfolioOptimizer`class, whereas determining the whole portfolio is made with our `optimize_portfolio_composition` method. Sections 3.2 and 3.3 should make it clear which arguments these methods have to be called with.

3.2 Support selection

Determining the support is a hard problem, as far as either mathematics or computer science are concerned. We could not find any definitive solution for it, so we derived three independent methods, the two first being pretty naive.

1. Choose the support at random. Do a large number of attempts to increase the best portfolio Sharpe found.

1. Gather in the support the 20 indexes corresponding to the assets having the best Sharpe. This is easy to do, because the Sharpe of any single asset can be retrieved directly from the API: indeed, once the `api_manager` object is instanced, just type

```
np.argsort(api_manager.sharpes)[-20:][::-1]
```

2. Find the support by solving the following quadratic problem. Let I be the set of all the 283 possible asset indexes, and $0\% \leq w_{min} \leq w_{max} \leq 100\%$ be real numbers to be fixed (we explain how at the end of this section). The function f we want to minimize (with $\alpha = 1$) can be written as

$$f(W) = ½\, W^T.P.W + q^T.W,$$

where $P = 2\,\Sigma$ is twice the covariance matrix, and $q = -R$. The problem we solve here is: find argmin f(W) subject to the usual constraints $0 \leq W$, $\mathbf{1}^T.W = 1$, together with $w_{min} \leq w_i \leq w_{max}$ (i in

I). This is the same problem as in Section 3.1, except that we dropped the support constraint, because the support is what we want to determine. Our present constraints can be rewritten matricially as

$$G.W \leq h,$$
$$A.W = b,$$

where W is the vector of the unknown variables, and the matrices G, h, A, b are not hard to write down explicitly (see also the `optimizer` method from our `PotfolioOptimizer` class). Now, solving this quadratic problem is immediate thanks to Python `cvxopt` library: just write `solvers.qp(P, q, G, h, A, b)['x']` to be done.
Now, to determine the support, **we simply take the indexes of the 20 assets having the highest weights found.**
Of course, depending on w_{min} and w_{max}, the portfolio we get at this stage has little chance to meet the required constraint $1\% \leq w_i \leq 10\%$, but remember we currently are only concerned with finding the portfolio support and not its exact composition – that this is fixed in Section 3.3.
A final word for choosing w_{min} and w_{max}. The last argument shows that we could even take $w_{min} = 0\%$ and $w_{max} = 100\%$, but this would lead to poor results (see the discussion in Section 4.4 for more details). We found it best to take e.g. $w_{min} = 0.1\%$ and $w_{max} = 10\%$. Taking other values (with w_{min} not too small and w_{max} not too large) does not change much to the corresponding support or the final portfolio Sharpe value (after the job in Section 3.3 is done).

3.3 Weight optimization

Thanks to the previous section, we now may assume that the support J of the portfolio is known. We again have several independent methods to determine the best portfolio composition, namely the weights w_i for i in J (of course, we have $w_i = 0$ if i is not in J). The two first following methods are naive.

a. Choose the weights indexed by the support at random, provided all the constraints are met. We did not actually experiment this option.

b. Take all weights indexed by the support (of cardinality 20) with uniform distribution:

$$w_i = 1/20 = 5\% \text{ for all i in J.}$$

c. Solve the quadratic problem discussed in Section 3. 2. 3, in which we replace the set I by J (the support), and set $w_{min} = 1\%$ and $w_{max} = 10\%$ to meet the constraints.

3.4 Results

Recall that we work with the parameters:

- `only_euro=True` (we are interested only in assets quoted in €, see Section 2.2),
- α = `risk_tolerance` = 1 (see Section 3.1),
- `is_annuel=True` (the returns for the Sharpe are computed on an annual basis, see Section 2.3).

Section 3.2 gives 3 different methods (1, 2 and 3) to determine the support of the portfolio, and Section 3.3 gives at least 2 different methods (b and c) to determine the weights once the support is known. We can combine these methods in 6 different ways, giving rise to 6 different methods and 6 different Sharpe. We sum up our results into a chart (see next page).

Portfolio Sharpe	Uniform weights	Optimized weights
Random support (1000 attempts)	26.9	44.5
Support with 20 **best Sharpes**	20.3	30.8
Optimized support (w_{min} = 0.1%, w_{max}= 10%)	45.2	**44.8**

4 Difficulties met

4.1 API Usage

It took us quite a long time to understand (and code) how to get the ratios from the documentation and post the portfolio in the right format.

4.2 Computation of the covariance

We were unable to retrieve all the ratios given in the documentation, especially the Beta (id 15) and the Pearson correlation (id 19). This made it impossible for us to compute the covariance of the returns directly, whence our approach (see Section 2.3). Please note also that not all the daily returns were available, so we made a `__none_values_manager` method inside the `ApiManager` class to address this. Basically, when a daily return is missing, we replace it by the half-sum of the returns of the days before and after, or by 0 if one of the latter is also unavailable.

4.3 Make the Sharpe optimization into a standard quadratic problem

Classic Markowitz theory optimizes the volatility *for a given return,* so how to optimize both the volatility and the return to get the best Sharpe? Since the portfolio return is comprised between R_{min} = return of the worst asset and R_{max} = return of the best asset, Xavier thought at first about sampling the interval [R_{min}; R_{max}], then solve the problem with the return equal to any fixed value in the sampled interval, and finally keep the return value corresponding to the best solution found ; of course, this would be a lengthy algorithm due to the large size of the sample. Discussions with comrades led us to think at the difference (volatility – return) as an objective function, whose optimization is equivalent to the Sharpe optimization, which is a way better approach because it still leads to a quadratic problem and avoids sampling.

4.4 Deal with the support

Recall that *exactly 20 assets are required in the portfolio.* We saw in Section 3 that without this constraint, our problem results to a quadratic problem that can be easily solved thanks to Python `cvxopt` library. However, adding this constraint makes the problem much harder, and `cvxopt` can no longer help us. This is the very reason we had to split the problem in half: first, determine the support, and then solve the problem with this support with `cvxopt`. We also could find no definitive solution for determining the support, so we derived several methods for it (see Section 3.2). When determining the support using optimization, we had in particular to beware of the lower and higher bounds for the weights put in the solver, because otherwise the solver gave only one dominant asset – that is, one asset with weight $w \approx 100\%$ (or the specified upper bound w_{max}), and almost all others with weights $w \approx 0\%$ (or the specified lower bound w_{min}). We found this weird, because we know since Markowitz that a good portfolio contains several assets (diversification strategy). We could fix it by choosing appropriate bounds for weights in the support research.

5 Conclusion

The naive algorithm, consisting of selecting the 20 assets having the best Sharpe and taking a uniform weight $w = 5\%$, gives a Sharpe

$$S \approx 20.3.$$

In contrast, the best method we found uses Markowitz theory, which amounts to solving a quadratic problem. To determine the portfolio support, we first run our solver with all 283 assets and weights between 0.1% and 10% ; from the solver output, we take the 20 assets having the highest weights. Once the support is known, we relaunch our solver on the same problem, but only with assets in the support and weights between 1% and 10% to meet the constraints. This yields a Sharpe

$$S \approx 44.8,$$

which is more than twice better! The detailed composition of our best portfolio can be found in our Python notebook after the title "Get Portfolio composition in the API".

We were very happy to make this project. We have learned a lot of things: how to deal with a REST API using Python, basic finance knowledge, and we found the optimization problem itself very challenging. It was a good opportunity for us to read further literature about finance and Markowitz theory (LAM), e.g. http://www.actuaries.org/AFIR/Colloquia/Rome2/Cesarone_Scozzari_Tardella.pdf .