12.2.OMIMS-PortOpt

December 10, 2019

1 Modern Portfolio Theory - Efficient Frontier in Python

1.1 Disclaimer

All the analyses provided below has been developed for illustrating some concepts about Modern Portfolio Theory. They have no value from an investment perspective.

1.2 Modern Portfolio Theory

Modern portfolio theory (MPT), or *mean-variance analysis*, is a mathematical framework for assembling a portfolio of assets based on its expected return and its level of risk.

It was introduced in an essay in 1952 by the economist Harry Markowitz, for which he was later awarded a Nobel Prize in economics.

Quadratic utility implies mean-variance preferences.

An investor will choose his optimal portfolio by determining a portfolio that maximizes:

$$\mu_p - \frac{g}{2}\sigma_p^2$$

where μ_p is he expected portfolio return, σ_p^2 its variance, and g is a risk-aversion parameter.

1.3 Warm Up: Mean-Variance Portfolio with Two Risky Assets

We consider a portfolio invested in 2 risky assets A and B

Let w_A be the weight in A and w_B be the weight in B.

The expected return on a given portfolio is

$$E[r_p] = w_A r_A + w_B r_B$$

where $E[r_p]$ is the expectation of the portfolio return and $\mathbf{r}^T = (r_A r_B)$ is the vector of expected returns for each risky asset.

The variance of this portfolio is given by:

$$\sigma_p^2 = w_A^2\sigma_A^2 + w_B^2\sigma_B^2 + 2w_A w_B \sigma_A \sigma_B \rho_{A,B}$$

This can be rewritten as

$$\mathbf{w}^T\mathbf{V}\mathbf{w},$$

where $\mathbf{w}^T = (w_A \; w_B)$ and $\mathbf{V} = Cov(\mathbf{r}, \mathbf{r})$ is the variance-covariance matrix between the returns.

With two assets:

$$\mathbf{V} = \begin{pmatrix} \sigma_A^2 & \sigma_{A,B} \\ \sigma_{A,B} & \sigma_B^2 \end{pmatrix} = \begin{pmatrix} \sigma_A^2 & \rho_{A,B}\sigma_A\sigma_B \\ \rho_{A,B}\sigma_A\sigma_B & \sigma_B^2 \end{pmatrix}$$

1.4 Diversification

Importantly, the portfolio variance is a function of the correlation coefficient between the assets in the portfolio, but the expected return is not.

Now, assume that $\rho_{A,B} = 1$, then:

$$\sigma_p^2 = (w_A\sigma_A + w_B\sigma_B)^2$$

Consequently:

$$\sigma_p = w_A\sigma_A + w_B\sigma_B$$

When $\rho_{A,B} = 1$, the risk on a portfolio (measured by its standard deviation) is the weighted-average of the risk of the individual assets in the portfolio. However, in practice $\rho_{A,B} < 1$ and so the risk on a portfolio is less than the weighted-average of the risk of the individual assets in the portfolio. This is the benefit of *diversification*

1.5 Efficient Frontier

The *efficient frontier* is the set of *optimal* portfolios that offer the *highest* expected return for a defined level of risk or the *lowest* risk for a given level of expected return

1.6 Effect of Correlation on Diversification: Two Assets

The graph below shows how the efficient frontier looks based on different values of correlation.

x-axis: portfolio standard deviation σ_p, **y-axis**: return $E(r_p)$

The stocks selected for this analysis are BCV (BCVN), Nestlé (NESN), Swisscom (SCMN), Roche (ROG), Zürich Insurance (ZURN), Schindler (SCHN) and Lindt (LISN).

```
In [1]: import pandas as pd
        import datetime
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        import scipy.optimize

        %matplotlib inline

        # Import of data
        # Price corresponds to the adjusted closing price
        # An adjusted closing price is a stock's closing price on any given day
        # of trading that has been amended to include any distributions and corpora
        # actions that occurred at any time before the next day's open. The adjuste
        # closing price is often used when examining historical returns or performi
        # a detailed analysis of historical returns.

        mydateparser = lambda x: pd.datetime.strptime(x, '%d/%m/%Y')

        stocks = pd.read_csv("Data/Stocks.csv", parse_dates=['Date'], date_parser=m

        # the data frame is indexed by the Date column
```

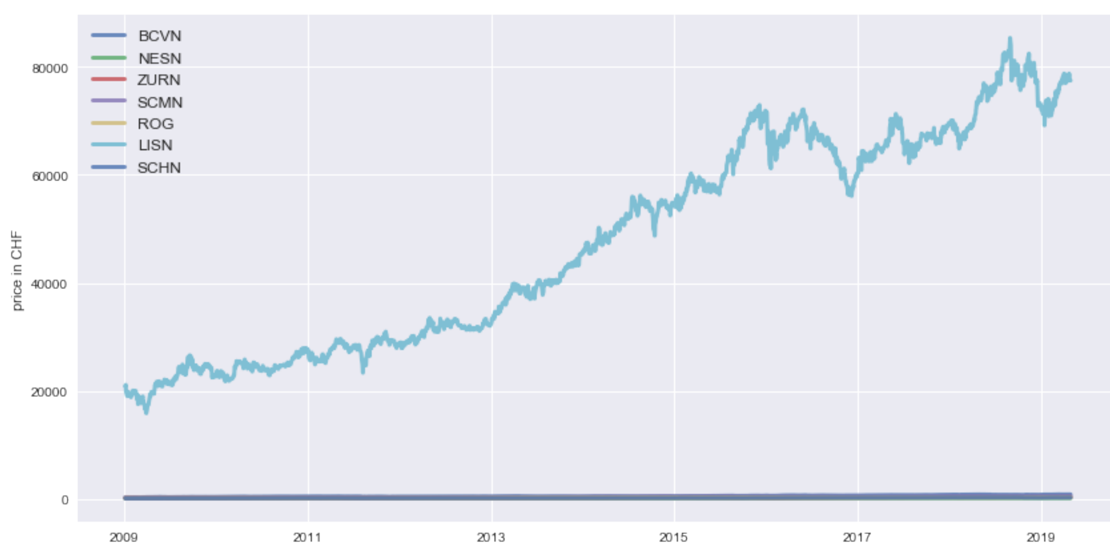
```
stocks = stocks.set_index("Date", drop = True)

type(stocks)

stocks.head()
```

```
Out[1]:                    BCVN       NESN        ZURN        SCMN        ROG  \
        Date
        2009-01-05  249.210556  29.899191  121.601677  206.359268  114.629456
        2009-01-06  249.989304  29.899191  124.758858  208.380981  115.852516
        2009-01-07  258.166504  29.927401  122.212769  205.059586  115.444824
        2009-01-08  255.440796  29.659437  120.226791  204.626404  117.211487
        2009-01-09  255.440796  28.474752  120.888771  205.203979  116.803772

                           LISN       SCHN
        Date
        2009-01-05  20837.46875  39.564487
        2009-01-06  20925.02148  42.496677
        2009-01-07  21100.12695  42.135178
        2009-01-08  19773.70898  42.215508
        2009-01-09  19708.04102  41.572838
```

Plot of the daily adjusted closing prices of each stock from 01/01/2009 to 29/04/2019.

```
In [2]: plt.figure(figsize=(14, 7))
        for c in stocks.columns.values:
            plt.plot(stocks.index, stocks[c], lw=3, alpha=0.8,label=c)
        plt.legend(loc='upper left', fontsize=12)
        plt.ylabel('price in CHF')
```

```
Out[2]: <matplotlib.text.Text at 0x1912cc5a080>
```



```
In [3]: stocks.plot(secondary_y = ["LISN"], grid = True, figsize = (14,10))
```

```
Out[3]: <matplotlib.axes._subplots.AxesSubplot at 0x1912ced1748>
```

Daily log-return data frame

```
In [4]: # shift moves dates back by 1. With the shift (+1), P(t-1) is now at the sa
        returns = stocks.apply(lambda x: np.log(x) - np.log(x.shift(1)))
        returns.head()
```

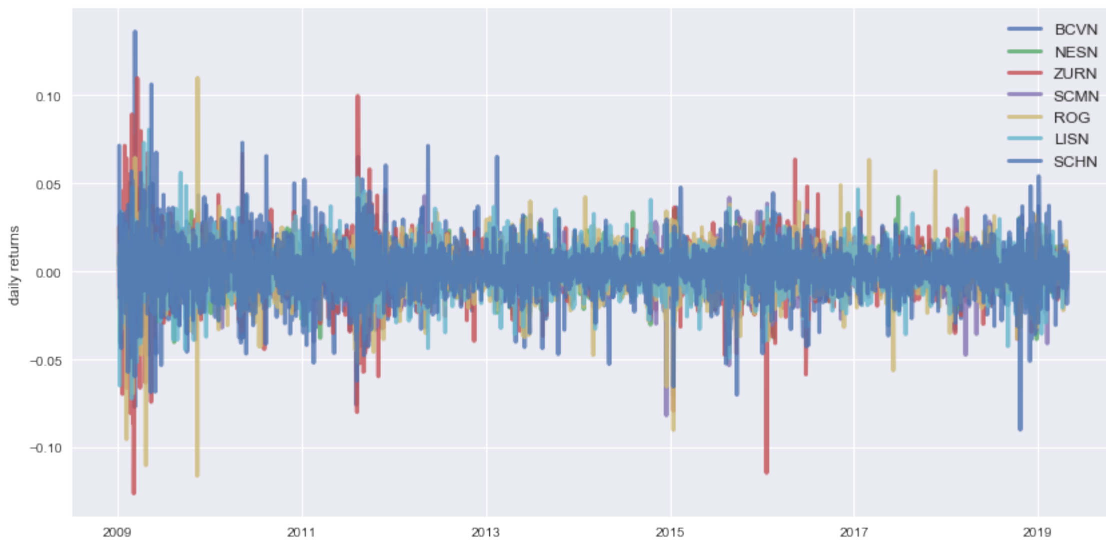
```
Out[4]:                 BCVN      NESN      ZURN      SCMN       ROG      LISN  \
        Date
        2009-01-05       NaN       NaN       NaN       NaN       NaN       NaN
        2009-01-06  0.003120  0.000000  0.025632  0.009749  0.010613  0.004193
        2009-01-07  0.032187  0.000943 -0.020619 -0.016067 -0.003525  0.008333
        2009-01-08 -0.010614 -0.008994 -0.016384 -0.002115  0.015187 -0.064926
        2009-01-09  0.000000 -0.040763  0.005491  0.002819 -0.003485 -0.003326

                        SCHN
        Date
        2009-01-05       NaN
        2009-01-06  0.071494
        2009-01-07 -0.008543
```

```
2009-01-08  0.001905
2009-01-09 -0.015341
```

Plot of the daily log-returns

```
In [5]: # returns = stocks.pct_change()

        plt.figure(figsize=(14, 7))
        for c in returns.columns.values:
            plt.plot(returns.index, returns[c], lw=3, alpha=0.8,label=c)
        plt.legend(loc='upper right', fontsize=12)
        plt.ylabel('daily returns')
```

```
Out[5]: <matplotlib.text.Text at 0x1912d3b7e10>
```



Knowing the mean daily returns over the period under consideration, we annualize it by multiplying it by 252 (252 open days in a year and log-returns are additive). Same approach for the volatility but with the square-root-of-time rule (1-year vol = $\sqrt{252}$ 1-day vol).

Daily volatility is given by :

$$\sqrt{\mathbf{w}^T \mathbf{V} \mathbf{w}},$$

The function below takes as arguments the portfolio weights , the mean expected return vector, and the return covariance matrix. Then it returns the portfolio standard deviation and its expected return

```
In [6]: def portfolio_vol_ret(weights, mean_returns, cov_matrix):
            exp_ret = np.sum(mean_returns*weights ) *252
            std = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights))) * np.sqrt
            return std, exp_ret
```

The function below generates random portfolios. It takes as arguments the number of portfolios to generate, the expected mean return vector, the return covariance matrix, and the risk-free rate. It returns two data frames. The first one contains portfolio standard deviations, expected returns, and their sharpe ratios. The second data frame store their weights.

```
In [7]: def random_portfolios(num_portfolios, mean_returns, cov_matrix, risk_free_r
            results = np.zeros((3,num_portfolios))
            weights_record = []
            for i in range(num_portfolios):
                weights = np.random.random(len(mean_returns))
                weights /= np.sum(weights)
                weights_record.append(weights)
                portfolio_std_dev, portfolio_return = portfolio_vol_ret(weights, me
                results[0,i] = portfolio_std_dev
                results[1,i] = portfolio_return
                results[2,i] = (portfolio_return - risk_free_rate) / portfolio_std_
            return results, weights_record
```

Here are the arguments that will be passed to the the *random_portfolios* function. We will generate 25'000 random portfolios !

```
In [8]: mean_returns = returns.mean()
        cov_matrix = returns.cov()
        num_portfolios = 25000
        risk_free_rate = -0.0075
```

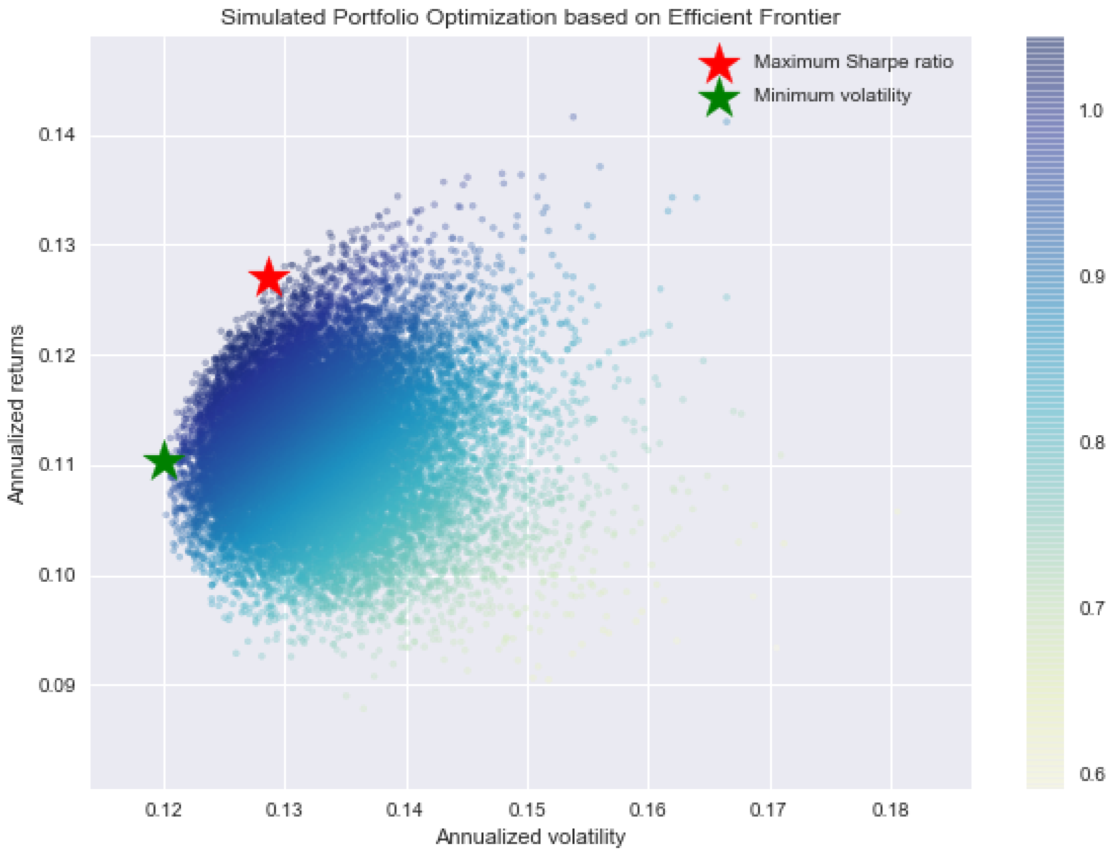
The function below will display the result in a nice way...

```
In [9]: def display_simulated_ef_with_random(mean_returns, cov_matrix, num_portfoli
            results, weights = random_portfolios(num_portfolios,mean_returns, cov_m

            max_sharpe_idx = np.argmax(results[2])
            sdp, rp = results[0,max_sharpe_idx], results[1,max_sharpe_idx]
            max_sharpe_allocation = pd.DataFrame(weights[max_sharpe_idx],index=stoc
            max_sharpe_allocation.allocation = [round(i*100,2)for i in max_sharpe_a
            max_sharpe_allocation = max_sharpe_allocation.T

            min_vol_idx = np.argmin(results[0])
            sdp_min, rp_min = results[0,min_vol_idx], results[1,min_vol_idx]
            min_vol_allocation = pd.DataFrame(weights[min_vol_idx],index=stocks.col
            min_vol_allocation.allocation = [round(i*100,2)for i in min_vol_allocat
            min_vol_allocation = min_vol_allocation.T

            print('-'*80)
            print("Maximum Sharpe Ratio Portfolio Allocation\n")
            print("Annualized Return:", round(rp,2))
            print("Annualized Volatility:", round(sdp,2))
            print("\n")
            print(max_sharpe_allocation)
```

```
            print('-'*80)
            print("Minimum Volatility Portfolio Allocation\n")
            print("Annualized Return:", round(rp_min,2))
            print("Annualized Volatility:", round(sdp_min,2))
            print("\n")
            print(min_vol_allocation)

            plt.figure(figsize=(10, 7))
            plt.scatter(results[0,:],results[1,:],c=results[2,:],cmap='YlGnBu', mar
            plt.colorbar()
            plt.scatter(sdp,rp,marker='*',color='r',s=500, label='Maximum Sharpe ra
            plt.scatter(sdp_min,rp_min,marker='*',color='g',s=500, label='Minimum v
            plt.title('Simulated Portfolio Optimization based on Efficient Frontier
            plt.xlabel('Annualized volatility')
            plt.ylabel('Annualized returns')
            plt.legend(labelspacing=0.8)
```

In [10]:
```
display_simulated_ef_with_random(mean_returns, cov_matrix, num_portfolios,
```

```
--------------------------------------------------------------------------------
Maximum Sharpe Ratio Portfolio Allocation

Annualized Return: 0.13
Annualized Volatility: 0.13


             BCVN   NESN  ZURN  SCMN   ROG   LISN   SCHN
allocation  16.06  29.38  0.48  7.24  1.13  20.98  24.73
--------------------------------------------------------------------------------
Minimum Volatility Portfolio Allocation

Annualized Return: 0.11
Annualized Volatility: 0.12


             BCVN  NESN  ZURN   SCMN   ROG   LISN  SCHN
allocation  19.44  25.7   1.2  27.91  2.02  16.48  7.25
```



Until now, we have randomly generated some portfolios and identified among them the one with the lowest volatility and the one with the highest Sharpe ratio.

But it is possible to determine them analytically.

In *scipy.optimize*, only a minimize function is available meaning that a maximization problem should be reformulated as a minimization problem. We remind that $max\ f(x)$ is equivalent to -min -$f(x)$.

Maximizing the Sharpe ratio is equivalent to minus minimizing minus the Sharpe ratio

```
In [11]: def neg_sharpe_ratio(weights, mean_returns, cov_matrix, risk_free_rate):
             vol, exp_ret = portfolio_vol_ret(weights, mean_returns, cov_matrix)
             return -(exp_ret - risk_free_rate) / vol
```

We will use a Sequential Quadratic Programming algorithm (SLSQP) to solve this constraint problem (maximizing the Sharpe ratio). Sequential quadratic programming (SQP) is an iterative method for constrained nonlinear optimization. SQP methods are used on mathematical problems for which the objective function and the constraints are twice continuously differentiable.

SQP methods solve a sequence of optimization subproblems, each of which optimizes a quadratic model of the objective subject to a linearization of the constraints. If the problem is unconstrained, then the method reduces to Newton's method for finding a point where the gradient of the objective vanishes. If the problem has only equality constraints, then the method is equivalent to applying Newton's method to the first-order optimality conditions of the problem.

```
In [12]: # initial weights for the optimization = equal weighting

         num_assets = len(mean_returns)
         print('Nbr of assets :', num_assets)

         init_weights =num_assets*[1./num_assets,]

         print('Initial weights (1/7) : ', init_weights)


         # box constraints
         # each weight is between 0 and 1
         bound = (0.0,1.0)

         bounds = tuple(bound for asset in range(num_assets))

         print('Box constraints for the weights :', bounds)
```

```
Nbr of assets : 7
Initial weights (1/7) :  [0.14285714285714285, 0.14285714285714285, 0.1428571428571
Box constraints for the weights : ((0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0),
```

Constraint: sum of the weights = 1 is equivalent to sum of the weights - 1 = 0
constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1})
We define now the function that will return the portfolio with the highest Sharpe ratio

```
In [13]: def max_sharpe_ratio(mean_returns, cov_matrix, risk_free_rate):
             num_assets = len(mean_returns)
             args = (mean_returns, cov_matrix, risk_free_rate)
             constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1})
             bound = (0.0,1.0)
             bounds = tuple(bound for asset in range(num_assets))
             result = scipy.optimize.minimize(neg_sharpe_ratio, num_assets*[1./num_
                                   method='SLSQP', bounds=bounds, constraints=constra
             return result
```

We also compute the minimum variance portfolio.

```
In [14]: def portfolio_volatility(weights, mean_returns, cov_matrix):
             return portfolio_vol_ret(weights, mean_returns, cov_matrix)[0]

         def min_variance(mean_returns, cov_matrix):
             num_assets = len(mean_returns)
             args = (mean_returns, cov_matrix)
             constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1})
             bound = (0.0,1.0)
             bounds = tuple(bound for asset in range(num_assets))
```

```
    result = scipy.optimize.minimize(portfolio_volatility, num_assets*[1./
                         method='SLSQP', bounds=bounds, constraints=constra

    return result
```

The first function *efficient_return* computes the **efficient** portfolio for a *given target return*, and the second function *efficient_frontier* will take a **range of target returns** and compute efficient portfolio for each return level.

```
In [15]: def efficient_return(mean_returns, cov_matrix, target):
             num_assets = len(mean_returns)
             args = (mean_returns, cov_matrix)

             def portfolio_return(weights):
                 return portfolio_vol_ret(weights, mean_returns, cov_matrix)[1]

             constraints = ({'type': 'eq', 'fun': lambda x: portfolio_return(x) - t
                            {'type': 'eq', 'fun': lambda x: np.sum(x) - 1})
             bounds = tuple((0,1) for asset in range(num_assets))
             result = scipy.optimize.minimize(portfolio_volatility, num_assets*[1./
             return result


         def efficient_frontier(mean_returns, cov_matrix, returns_range):
             efficients = []
             for ret in returns_range:
                 efficients.append(efficient_return(mean_returns, cov_matrix, ret))
             return efficients
```

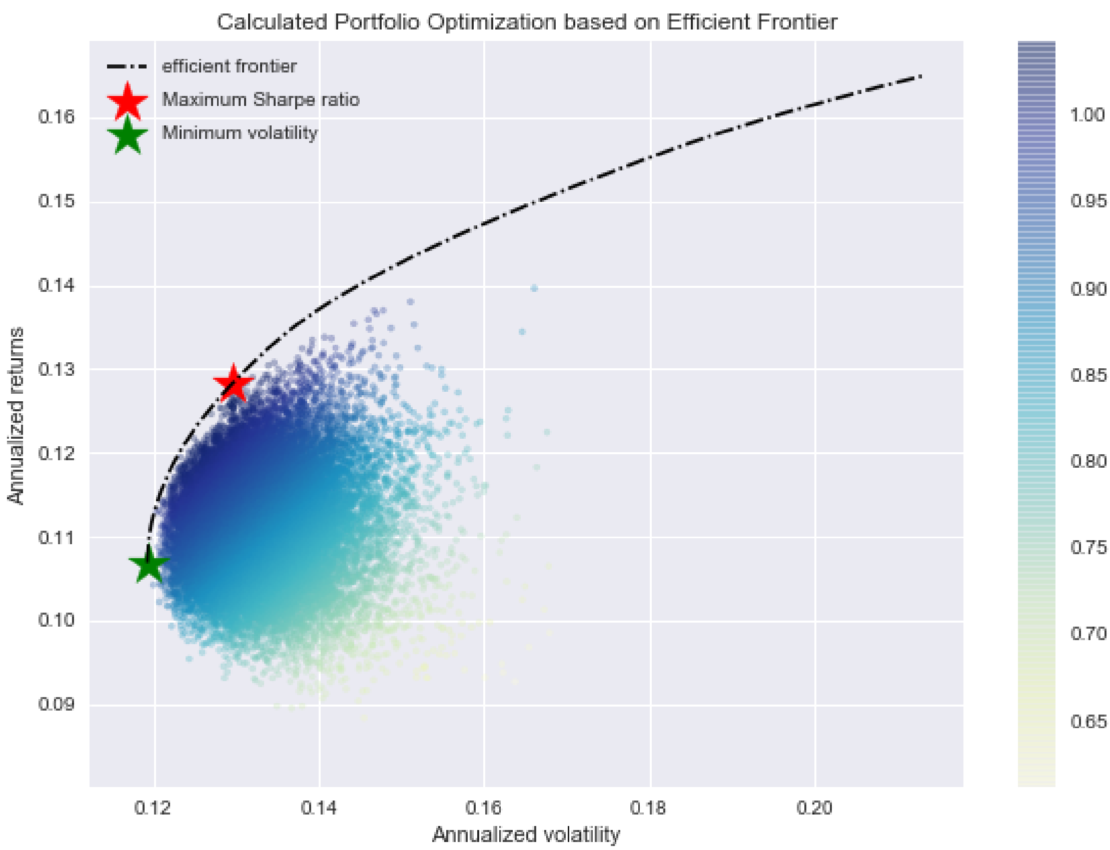
We plot the portfolios with the maximal Sharpe ratio, the minimum volatility and all the randomly generated portfolios. We also plot the efficient frontier line.

```
In [16]: def display_ef_with_random_portfolios(mean_returns, cov_matrix, num_portfo
             results, _ = random_portfolios(num_portfolios,mean_returns, cov_matrix

             max_sharpe = max_sharpe_ratio(mean_returns, cov_matrix, risk_free_rate
             sdp, rp = portfolio_vol_ret(max_sharpe['x'], mean_returns, cov_matrix)
             max_sharpe_allocation = pd.DataFrame(max_sharpe.x,index=stocks.columns
             max_sharpe_allocation.allocation = [round(i*100,2)for i in max_sharpe_
             max_sharpe_allocation = max_sharpe_allocation.T
             max_sharpe_allocation

             min_vol = min_variance(mean_returns, cov_matrix)
             sdp_min, rp_min = portfolio_vol_ret(min_vol['x'], mean_returns, cov_ma
             min_vol_allocation = pd.DataFrame(min_vol.x,index=stocks.columns,colum
             min_vol_allocation.allocation = [round(i*100,2)for i in min_vol_alloca
             min_vol_allocation = min_vol_allocation.T

             print("-"*80)
```

```
    print("Maximum Sharpe Ratio Portfolio Allocation\n")
    print("Annualized Return:", round(rp,2))
    print("Annualized Volatility:", round(sdp,2))
    print("\n")
    print(max_sharpe_allocation)
    print("-"*80)
    print("Minimum Volatility Portfolio Allocation\n")
    print("Annualized Return:", round(rp_min,2))
    print("Annualized Volatility:", round(sdp_min,2))
    print("\n")
    print(min_vol_allocation)

    plt.figure(figsize=(10, 7))
    plt.scatter(results[0,:],results[1,:],c=results[2,:],cmap='YlGnBu', ma
    plt.colorbar()
    plt.scatter(sdp,rp,marker='*',color='r',s=500, label='Maximum Sharpe r
    plt.scatter(sdp_min,rp_min,marker='*',color='g',s=500, label='Minimum

    target = np.linspace(rp_min, 0.165, 50)
    efficient_portfolios = efficient_frontier(mean_returns, cov_matrix, ta
    plt.plot([p['fun'] for p in efficient_portfolios], target, linestyle='
    plt.title('Calculated Portfolio Optimization based on Efficient Fronti
    plt.xlabel('Annualized volatility')
    plt.ylabel('Annualized returns')
    plt.legend(labelspacing=0.8)
```

```
In [17]: display_ef_with_random_portfolios(mean_returns, cov_matrix, num_portfolios
```

```
--------------------------------------------------------------------------------
Maximum Sharpe Ratio Portfolio Allocation

Annualized Return: 0.13
Annualized Volatility: 0.13


             BCVN   NESN  ZURN  SCMN  ROG   LISN   SCHN
allocation  16.39  29.36   0.0  7.12  0.0  20.91  26.21
--------------------------------------------------------------------------------
Minimum Volatility Portfolio Allocation

Annualized Return: 0.11
Annualized Volatility: 0.12


             BCVN  NESN  ZURN   SCMN   ROG   LISN  SCHN
allocation  15.74  23.3   0.0  33.46  5.85  15.42  6.24
```



We now plot each individual stocks in the graph. We can see how diversification is lowering the risk by optimizing the allocation.

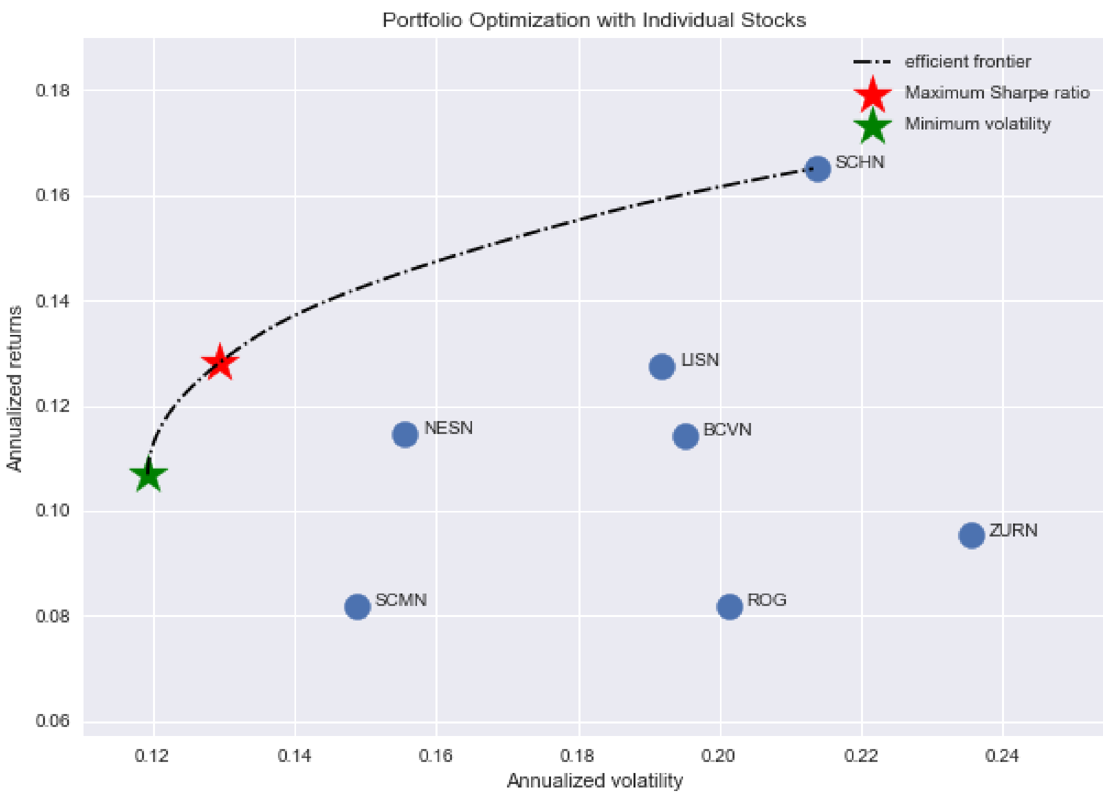
```
In [18]: def display_ef_with_selected_stocks(mean_returns, cov_matrix, risk_free_ra
             max_sharpe = max_sharpe_ratio(mean_returns, cov_matrix, risk_free_rate
             sdp, rp = portfolio_vol_ret(max_sharpe['x'], mean_returns, cov_matrix)
             max_sharpe_allocation = pd.DataFrame(max_sharpe.x,index=stocks.columns
             max_sharpe_allocation.allocation = [round(i*100,2)for i in max_sharpe_
             max_sharpe_allocation = max_sharpe_allocation.T
             max_sharpe_allocation

             min_vol = min_variance(mean_returns, cov_matrix)
             sdp_min, rp_min = portfolio_vol_ret(min_vol['x'], mean_returns, cov_ma
             min_vol_allocation = pd.DataFrame(min_vol.x,index=stocks.columns,colum
             min_vol_allocation.allocation = [round(i*100,2)for i in min_vol_alloca
             min_vol_allocation = min_vol_allocation.T

             an_vol = np.std(returns) * np.sqrt(252)
             an_rt = mean_returns * 252

             print("-"*80)
```

```
    print("Maximum Sharpe Ratio Portfolio Allocation\n")
    print("Annualized Return:", round(rp,2))
    print("Annualized Volatility:", round(sdp,2))
    print("\n")
    print("max_sharpe_allocation")
    print("-"*80)
    print("Minimum Volatility Portfolio Allocation\n")
    print("Annualized Return:", round(rp_min,2))
    print("Annualized Volatility:", round(sdp_min,2))
    print("\n")
    print("min_vol_allocation")
    print("-"*80)
    print("Individual Stock Returns and Volatility\n")
    for i, txt in enumerate(stocks.columns):
        print(txt,":","Annualized return",round(an_rt[i],2),", Annualized
    print("-"*80)

    fig, ax = plt.subplots(figsize=(10, 7))
    ax.scatter(an_vol,an_rt,marker='o',s=200)

    for i, txt in enumerate(stocks.columns):
        ax.annotate(txt, (an_vol[i],an_rt[i]), xytext=(10,0), textcoords='
    ax.scatter(sdp,rp,marker='*',color='r',s=500, label='Maximum Sharpe ra
    ax.scatter(sdp_min,rp_min,marker='*',color='g',s=500, label='Minimum v

    target = np.linspace(rp_min, 0.165, 50)
    efficient_portfolios = efficient_frontier(mean_returns, cov_matrix, ta
    ax.plot([p['fun'] for p in efficient_portfolios], target, linestyle='-
    ax.set_title('Portfolio Optimization with Individual Stocks')
    ax.set_xlabel('Annualized volatility')
    ax.set_ylabel('Annualized returns')
    ax.legend(labelspacing=0.8)
```

```
In [19]: display_ef_with_selected_stocks(mean_returns, cov_matrix, risk_free_rate)
```

```
--------------------------------------------------------------------------------
Maximum Sharpe Ratio Portfolio Allocation

Annualized Return: 0.13
Annualized Volatility: 0.13


max_sharpe_allocation
--------------------------------------------------------------------------------
Minimum Volatility Portfolio Allocation

Annualized Return: 0.11
Annualized Volatility: 0.12
```

```
min_vol_allocation
--------------------------------------------------------------------------------
Individual Stock Returns and Volatility

BCVN : Annualized return 0.11 , Annualized volatility: 0.2
NESN : Annualized return 0.11 , Annualized volatility: 0.16
ZURN : Annualized return 0.1 , Annualized volatility: 0.24
SCMN : Annualized return 0.08 , Annualized volatility: 0.15
ROG : Annualized return 0.08 , Annualized volatility: 0.2
LISN : Annualized return 0.13 , Annualized volatility: 0.19
SCHN : Annualized return 0.17 , Annualized volatility: 0.21
--------------------------------------------------------------------------------
```




```
In [ ]:
```