Portfolio Construction

[1]:
```
import os
import time
import datetime
import numpy as np
import pandas as pd
import scipy.stats as scs
from pylab import plt, mpl

plt.style.use('seaborn-v0_8')
mpl.rcParams['savefig.dpi'] = 300
mpl.rcParams['font.family'] = 'serif'
pd.set_option('mode.chained_assignment', None)
pd.set_option('display.float_format', '{:.4f}'.format)
np.set_printoptions(suppress=True, precision=4)
os.environ['PYTHONHASHSEED'] = '0'
```

Portfolio Optimization

The modern or mean-variance portfolio theory is a fundamental pillar of financial theory. The theory is built upon the assumption of normally distributed returns, which enables us to focus on mean and variance as the primary statistics for describing the distribution of end-of-period wealth.

The process of portfolio optimization involves solving a mathematical problem to find the set of portfolio weights that minimizes the portfolio variance for a given level of expected return or maximizes the expected return for a given level of portfolio risk. This empowers investors to construct portfolios that align precisely with their risk preferences and investment objectives.

Data

We will analyze four historical financial time series, two for technology stocks and two for exchange traded funds (ETFs).

[2]:
```
dataFrame = pd.read_csv('sp500_asset_selection.csv', # sp500_asset_selection,␣
 ↪dow_asset_selection
                        index_col=0, parse_dates=True).dropna()

columns = ['AAPL', 'UNH', 'HD', 'V', 'MSFT', 'GLD']
```

```python
noOfAssets = len(columns)

dFrame = dataFrame[columns].dropna()
dFrame.info()# display dataset information
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3395 entries, 2010-01-04 to 2023-06-29
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   AAPL    3395 non-null   float64
 1   UNH     3395 non-null   float64
 2   HD      3395 non-null   float64
 3   V       3395 non-null   float64
 4   MSFT    3395 non-null   float64
 5   GLD     3395 non-null   float64
dtypes: float64(6)
memory usage: 185.7 KB
```

[3]:
```python
dFrame.head()
```

[3]:

Date	AAPL	UNH	HD	V	MSFT	GLD
2010-01-04	7.6432	31.5300	28.6700	22.0350	30.9500	109.8000
2010-01-05	7.6564	31.4800	28.8800	21.7825	30.9600	109.7000
2010-01-06	7.5346	31.7900	28.7800	21.4900	30.7700	111.5100
2010-01-07	7.5207	33.0100	29.1200	21.6900	30.4500	110.8200
2010-01-08	7.5707	32.7000	28.9800	21.7500	30.6600	111.3700

Log returns of the financial instruments

[4]:
```python
log_returns = np.log(dFrame / dFrame.shift(1))
log_returns.dropna(inplace=True)
log_returns.head()
```

[4]:

Date	AAPL	UNH	HD	V	MSFT	GLD
2010-01-05	0.0017	-0.0016	0.0073	-0.0115	0.0003	-0.0009
2010-01-06	-0.0160	0.0098	-0.0035	-0.0135	-0.0062	0.0164
2010-01-07	-0.0019	0.0377	0.0117	0.0093	-0.0105	-0.0062
2010-01-08	0.0066	-0.0094	-0.0048	0.0028	0.0069	0.0050
2010-01-11	-0.0089	0.0067	-0.0287	-0.0029	-0.0128	0.0132

Frequency distribution of the log returns

Figure 1-1. shows the frequency distribution of the log returns for the financial instruments

[5]:
```
log_returns.hist(bins=50, figsize=(10, 8));
plt.figtext(0.5, 0.0001, 'Fig 1-1. Histograms of log returns for financial␣
  ↪instruments', style='italic', ha='center')
plt.show()
```

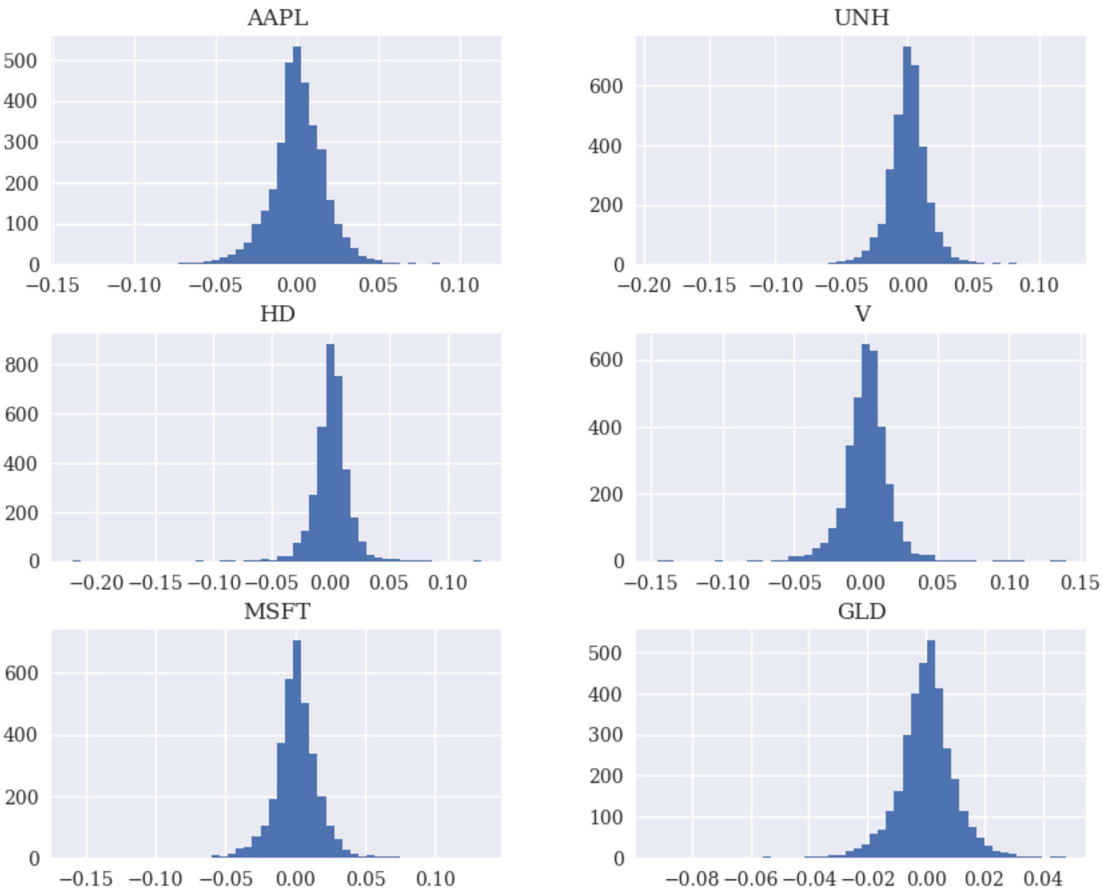

Fig 1-1. Histograms of log returns for financial instruments

The Basic Theory

In this scenario, we will assume that the investor is not allowed to set up short positions in a financial instrument. Only long positions are permitted, meaning that the investor's entire wealth must be divided among the available instruments. The goal is to ensure that all positions are long (positive) and that the total sum of positions adds up to 100%.

To achieve this, let's consider four instruments. One approach could be to invest an equal amount in each instrument, allocating 25% of the available wealth to each instrument. This ensures a balanced distribution. Alternatively, we can use Monte Carlo simulation to generate random portfolio weight vectors on a larger scale, this allows us to simulate various portfolio allocations and record the resulting expected portfolio return and variance. To enhance code readability and simplify the implementation, we have defined two functions: port_returns() and port_volatility(). These functions provide a clear and concise way to calculate the expected portfolio return and variance, respectively.

[6]:
```
def port_returns(weights):
    return np.sum(log_returns.mean() * weights) * 252

def port_volatility(weights):
    return np.sqrt(np.dot(weights.T, np.dot(log_returns.cov() * 252, weights)))

portReturns    = []
portVolatility = []

for p in range (2000):
    weights = np.random.random(noOfAssets)
    weights /= np.sum(weights)

    portReturns.append(port_returns(weights))
    portVolatility.append(port_volatility(weights))

portReturns = np.array(portReturns)
portVolatility = np.array(portVolatility)
```

Figure 1-2. shows the results of the Monte Carlo simulation. In addition, it provides results for the Sharpe ratio.

[7]:
```
plt.figure(figsize=(10, 6))
plt.scatter(portVolatility, portReturns, c = portReturns / portVolatility,
            marker='o', cmap='coolwarm')
plt.xlabel('expected volatility')
plt.ylabel('expected return')
plt.colorbar(label='Sharpe ratio');
plt.figtext(0.5, 0.0001, 'Fig 1-2. Expected return and volatility for random␣
 ↪portfolio weights', style='italic', ha='center')
plt.show()
```

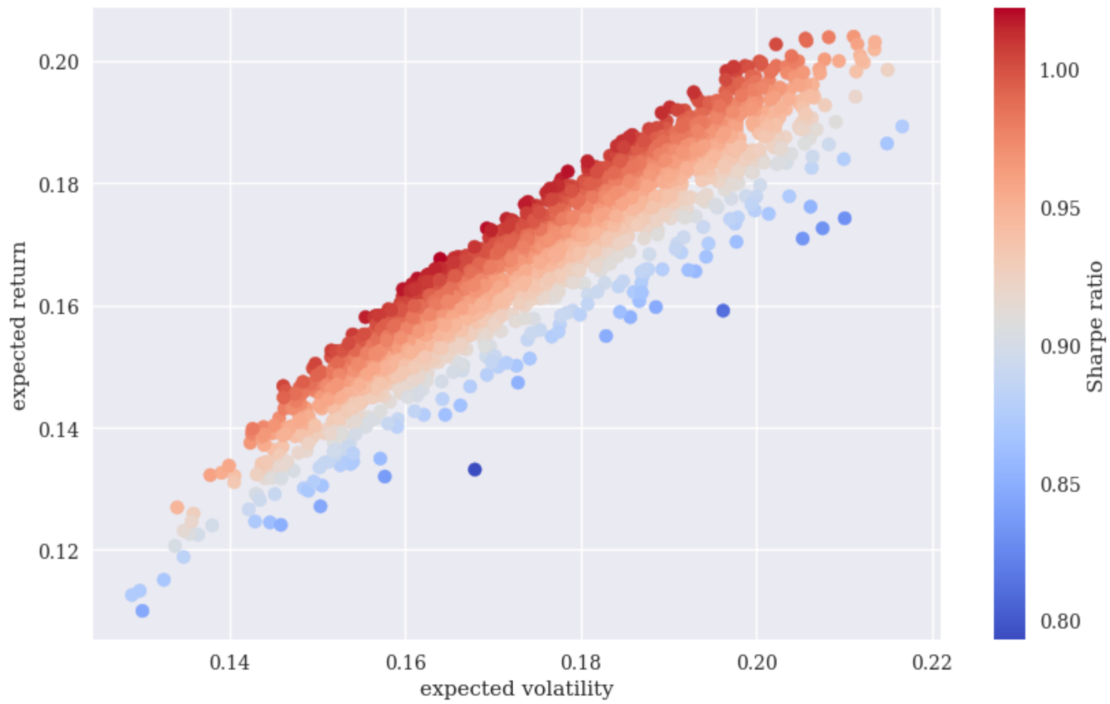

Fig 1-2. Expected return and volatility for random portfolio weights

It is evident upon examining Figure 1-2 that not all weight distributions exhibit satisfactory performance when evaluated based on mean and volatility. For instance, when considering a fixed risk level of, let's say, 14%, there exist multiple portfolios that yield varying returns. As an investor, one is typically interested in maximizing returns while maintaining a fixed risk level or minimizing risk while expecting a fixed return. This collection of portfolios is commonly referred to as the *efficient frontier*, which will be further elaborated upon in the subsequent section.

Optimal Portfolios

To optimize the portfolio composition, we can use a minimization function that is quite versatile. This function allows for the inclusion of equality constraints, inequality constraints, and numerical bounds for the parameters.

First, let's focus on maximizing the Sharpe ratio. In formal terms, we aim to minimize the negative value of the Sharpe ratio to derive the maximum value and achieve the optimal portfolio composition. To ensure that all parameters (weights) add up to 1, we can set a constraint. This constraint can be formulated using the conventions of the minimize() function. Additionally, we need to ensure that the parameter values (weights) fall within the range of 0 and 1. These values can be provided to the minimization function as a tuple of tuples.

The only missing input for the optimization function is a starting parameter list, which serves as an initial guess for the weights vector. In this case, an equal distribution of weights will suffice.

[8]:
```python
import scipy.optimize as sco

def min_func_sharpe(weights):    # function to be minimized
    return -port_returns(weights) / port_volatility(weights)

constraints_ = ({'type': 'eq', 'fun': lambda x:  np.sum(x) - 1})  #equality␣
 ↪constraint

bounds_ = tuple((0, 1) for x in range(noOfAssets))   # bounds for the␣
 ↪parameters.

equal_weights = np.array(noOfAssets * [1. / noOfAssets,]) # equal weights vector

equal_weights
```

[8]:
```
array([0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667])
```

[9]:
```python
min_func_sharpe(equal_weights)
```

[9]:
```
-0.9884742405123803
```

When we call the function, it not only returns the optimal parameter values but also stores the results in an object called "optimals." The primary focus is on obtaining the composition of the optimal portfolio. To achieve this, we can access the results object by specifying the key of interest, which in this case is "x."

[10]:
```python
optimals = sco.minimize(min_func_sharpe, equal_weights,
                        method='SLSQP', bounds=bounds_,
                        constraints=constraints_)
```

The results from the optimization i.e the optimal portfolio weights

[11]:
```python
optimals['x'].round(3)
```

[11]:
```
array([0.272, 0.272, 0.171, 0.085, 0.   , 0.2  ])
```

The resulting portfolio return.

[12]:
```python
port_returns(optimals['x']).round(3)
```

[12]:
```
0.172
```

The resulting portfolio volatility.

[13]:
```python
port_volatility(optimals['x']).round(3)
```

[13]:
```
0.168
```

The maximum Sharpe ratio

[14]:
```
port_returns(optimals['x']) / port_volatility(optimals['x'])
```

[14]:
```
1.0242913520009258
```

0.0.1 The minimization of the portfolio volatility

[15]:
```
optimalsVolatility = sco.minimize(port_volatility, equal_weights,
                                  method='SLSQP', bounds=bounds_,
                                  constraints=constraints_)
optimalsVolatility
```

[15]:
```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: 0.12379909591404203
       x: [ 1.910e-02  1.038e-01  1.250e-01  9.193e-02  4.475e-02
            6.155e-01]
     nit: 7
     jac: [ 1.234e-01  1.236e-01  1.236e-01  1.233e-01  1.231e-01
            1.240e-01]
    nfev: 49
    njev: 7
```

[16]:
```
optimalsVolatility['x'].round(3)
```

[16]:
```
array([0.019, 0.104, 0.125, 0.092, 0.045, 0.615])
```

The resulting portfolio return

[17]:
```
port_returns(optimalsVolatility['x']).round(3)
```

[17]:
```
0.093
```

The resulting portfolio volatility

[18]:
```
port_volatility(optimalsVolatility['x']).round(3)
```

[18]:
```
0.124
```

The maximum Sharpe ratio

[19]:
```
port_returns(optimalsVolatility['x']) / port_volatility(optimalsVolatility['x'])
```

[19]:
```
0.7544340233448462
```

In this particular scenario, the portfolio consists of just two financial instruments. This unique combination of assets results in what is commonly referred to as the minimum volatility or minimum variance portfolio. By carefully selecting these specific instruments, we can achieve a portfolio mix that minimizes fluctuations and optimizes stability.

Efficient Frontier

To derive all optimal portfolios, which are portfolios with minimum volatility for a given target return level or maximum return for a given risk level, a similar optimization approach is followed as before. The only difference is that we need to iterate over multiple starting conditions.

The approach we take is to fix a target return level and derive the portfolio weights that lead to the minimum volatility value for each target return level. This optimization process involves two conditions: one for the target return level, target_return, and one for the sum of the portfolio weights, as we have done previously. The boundary values for each parameter remain the same. However, when iterating over different target return levels (target_return_levels), one condition for the minimization changes. Therefore, the constraints dictionary is updated during every loop.

The two binding constraints for the efficient frontier

[20]:
```
cons = ({'type': 'eq', 'fun': lambda x:  port_returns(x) - target_return},
        {'type': 'eq', 'fun': lambda x:  np.sum(x) - 1})

bnds = tuple((0, 1) for x in weights)
```

The minimization of portfolio volatility for different target returns

[21]:
```
target_return_levels = np.linspace (0.09, 0.215, 100)  # initial: (0.05, 0.2,␣
↪50); (0.06, 0.15, 50)
tvols = []
for target_return in target_return_levels:
    res = sco.minimize(port_volatility, equal_weights, method='SLSQP',
                       bounds=bnds, constraints=cons)
    tvols.append(res['fun'])
tvols = np.array(tvols)
```

Figure 1-3 presents the optimization results. The crosses represent the optimal portfolios for a specific target return, while the dots represent the random portfolios, as mentioned earlier. Additionally, the figure showcases two larger stars: The red one, denotes the minimum volatility/variance portfolio (located on the leftmost side), and the yellow one represents the portfolio with the maximum Sharpe ratio.

[22]:
```
plt.figure(figsize=(10, 6))
plt.scatter(portVolatility, portReturns, c = portReturns / portVolatility,
```

```
            marker='o', alpha=0.8, cmap='coolwarm')
plt.plot(tvols, target_return_levels, 'b', lw=4.0)
plt.plot(port_volatility(optimals['x']), port_returns(optimals['x']),
         'y*', markersize=15.0)
plt.plot(port_volatility(optimalsVolatility['x']),␣
 ↪port_returns(optimalsVolatility['x']),
         'r*', markersize=15.0)
plt.xlabel('expected volatility')
plt.ylabel('expected return')
plt.colorbar(label='Sharpe ratio')
plt.figtext(0.5, 0.0001, 'Fig 1-3. Minimum risk portfolios for given return␣
 ↪levels (efficient frontier)', style='italic', ha='center')
plt.show()
```

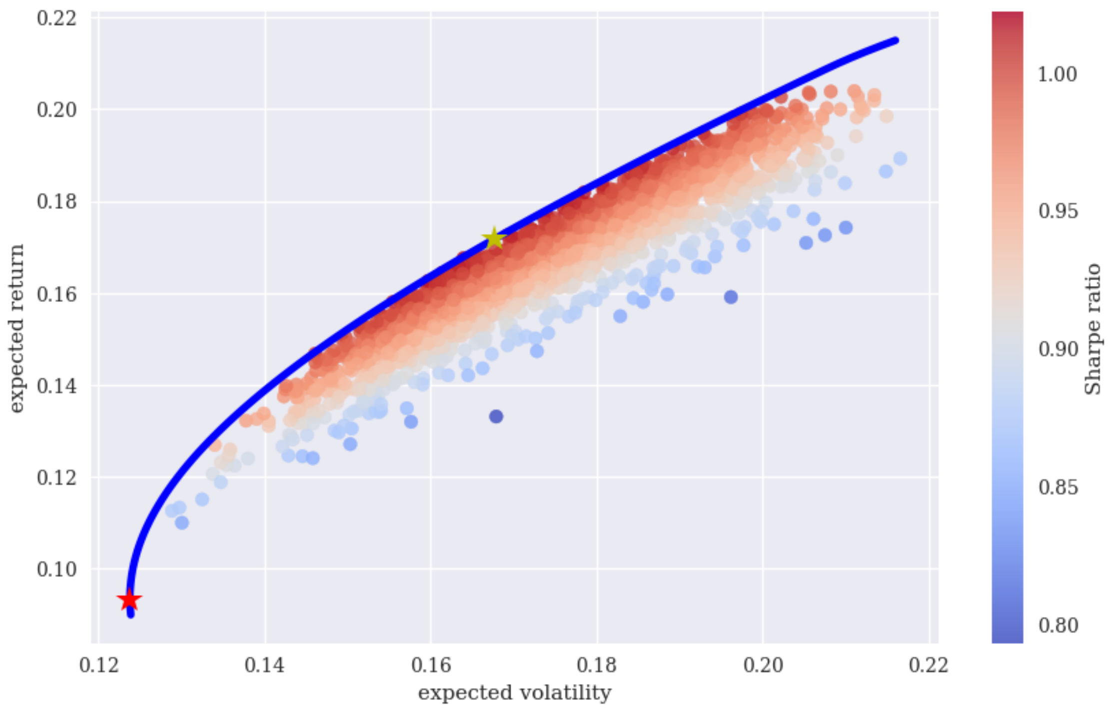

Fig 1-3. Minimum risk portfolios for given return levels (efficient frontier)

The *efficient frontier* consists of all optimal portfolios that offer a higher return than the absolute minimum variance portfolio. These portfolios outperform all others in terms of expected returns at a given risk level. In other words, they strike the perfect balance between risk and reward, maximizing returns while minimizing risk.

Capital Market Line

The Capital Market Line (CML) introduces the concept of a riskless investment opportunity in addition to risky financial instruments like stocks or commodities. While cash accounts with large banks are considered riskless, they typically yield only a small return. However, including this riskless asset in an investment portfolio expands the range of efficient investment opportunities for investors.

The CML suggests that investors should first determine an efficient portfolio of risky assets and then add the riskless asset to the mix. By adjusting the proportion of their wealth invested in the riskless asset, investors can achieve any desired risk-return profile along the straight line connecting the riskless asset and the efficient portfolio.

To identify the optimal efficient portfolio for investment, investors should select the portfolio where the tangent line of the efficient frontier intersects with the risk-return point of the riskless portfolio. This ensures the best balance between risk and return in the investment strategy.

To compute this optimal portfolio, the function sco.fsolve() from the scipy.optimize library can be used. It requires an initial parameterization, which should be carefully chosen through a combination of educated guesses and trial and error. It's important to note that the success or failure of the optimization process may depend on the initial parameterization.

In the calculations for the efficient frontier, a functional approximation and the first derivative are utilized. To achieve this, cubic splines interpolation is employed, which allows for a differentiable functional approximation. The spline interpolation specifically focuses on portfolios from the efficient frontier. By adopting this numerical approach, it becomes possible to define a continuously differentiable function f(x) for the efficient frontier, as well as its corresponding first derivative function df(x). This methodology ensures a smooth and continuous representation of the efficient frontier, enabling further analysis and optimization.

[23]:
```
import scipy.interpolate as sci

# Index position of minimum volatility portfolio
ind = np.argmin(tvols)

# Relevant portfolio volatility and return values.
evols = tvols[ind:]
erets = target_return_levels[ind:]

# Cubic splines interpolation
tck = sci.splrep(evols, erets)

def f(x):
    ''' Efficient frontier function (splines approximation).
    '''
    return sci.splev(x, tck, der=0)

def df(x):
    ''' First derivative of efficient frontier function.
    '''
    return sci.splev(x, tck, der=1)
```

[24]:
```
def equations(p, rf=0.04):
```

```
    # The equations describing the capital market line
    eq1 = rf - p[0]
    eq2 = rf + p[1] * p[2] - f(p[2])
    eq3 = p[1] - df(p[2])
    return eq1, eq2, eq3

# The Initial values, Solved by a combination of educated guesses with trial␣
 ↪and error
# Solving these equations for given initial values.
#opt = sco.fsolve(equations, [0.03, 0.5, 0.15]) #0.5 # educated guesses with␣
 ↪trial and error
opt = sco.fsolve(equations, [0.04, 0.5, 0.15]) # The optimal parameter values.
```

[25]:
```
np.round(equations(opt), 10)  # the equation values are all zero
```

[25]:
```
array([ 0., -0.,  0.])
```

In Figure 1-4, we can observe the graphical representation of the results. The star symbolizes the optimal portfolio, which lies on the efficient frontier and has a tangent line passing through the riskless asset point. This visualization provides valuable insights into the portfolio's performance and risk characteristics.

[26]:
```
plt.figure(figsize=(10, 6))
plt.scatter(portVolatility, portReturns, c = (portReturns - 0.04) /␣
 ↪portVolatility,
            marker='o', cmap='coolwarm')
plt.plot(evols, erets, 'b', lw=4.0)
cx = np.linspace(0.00, 0.25)
plt.plot(cx, opt[0] + opt[1] * cx, 'r', lw=1.5)
plt.plot(opt[2], f(opt[2]), 'y*', markersize=15.0)
plt.grid(True)
plt.axhline(0, color='k', ls='--', lw=2.0)
plt.axvline(0, color='k', ls='--', lw=2.0)
plt.xlabel('expected volatility')
plt.ylabel('expected return')
plt.colorbar(label='Sharpe ratio')
plt.figtext(0.5, 0.0001, 'Fig 1-4. Capital market line and tangency portfolio␣
 ↪(star) for risk-free rate of 1%', style='italic', ha='center')
plt.show()
```

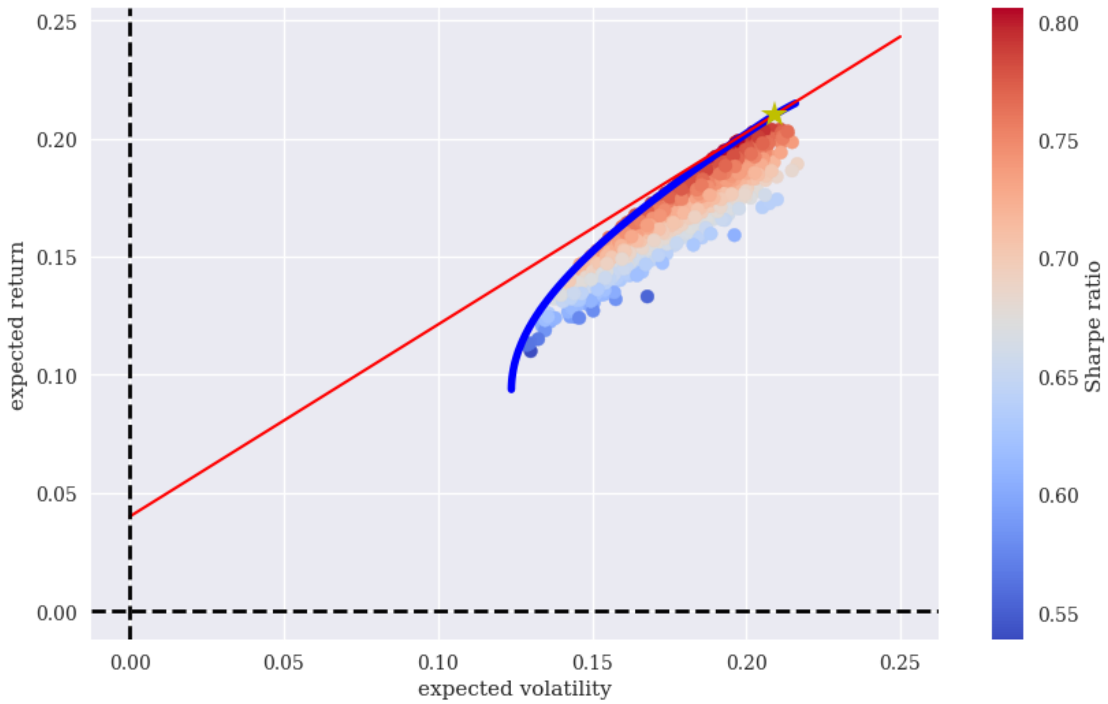

Fig 1-4. Capital market line and tangency portfolio (star) for risk-free rate of 1%

The portfolio weights of the optimal (tangent) portfolio consist of three out of the four assets in the mix. Here are the specific weightings for each asset:

[27]:
```
cons = ({'type': 'eq', 'fun': lambda x:  port_returns(x) - f(opt[2])},
        {'type': 'eq', 'fun': lambda x:  np.sum(x) - 1})

# Binding constraints for the tangent portfolio (gold star in Figure 1-4).
res = sco.minimize(port_volatility, equal_weights, method='SLSQP',
                       bounds=bnds, constraints=cons)

# The portfolio weights for this particular portfolio
res['x'].round(3)
```

[27]:
```
array([0.401, 0.355, 0.178, 0.066, 0.    , 0.    ])
```

The resulting portfolio return

[28]:
```
port_returns(res['x'])
```

[28]:
```
0.2101350546617612
```

The resulting portfolio volatility

[29]:
```
port_volatility(res['x'])
```

[29]: 0.2092832612477399

The maximum Sharpe ratio

[30]:
```
port_returns(res['x']) / port_volatility(res['x'])
```

[30]: 1.0040700503659152

[ ]:
```

```

[ ]:
```

```