3 - Optimal Portfolios and Matrices"

Espen Sirnes

2024-09-18

Table of contents

This lecture explores the strategic behavior of an investor in the stock market, particularly under the assumption of risk aversion, as discussed in the previous note on utility theory. Risk lovers generally prefer the most risky assets, while risk-neutral investors opt for assets with the highest returns. In contrast, a risk-averse investor seeks to maximize returns without disproportionately increasing volatility, typically measured as variance.

1 Matrices

To calculate optimal portfolios for any number of assets, a basic understanding of matrix algebra is essential. Matrix algebra simplifies the resolution of several equations simultaneously, a process that becomes increasingly complex with the addition of variables. Using matrix functions in software like Excel and various statistical packages allows us to solve systems of equations efficiently without manually computing each one.

Matrices not only streamline the computation but also simplify notation, making the formulation of equations for optimal portfolios more manageable.

A matrix is a structured array of numbers arranged in rows and columns, essentially a set of vectors. Here's an example of a vector:

```
import numpy as np
np.random.randint(0,100,3)
```

```
array([23,  6, 52])
```

Combining several vectors side-by-side forms a matrix:

```
np.random.randint(0,100,(2,3))
```

```
array([[81, 20, 94],
       [54, 12, 38]])
```

This format is sometimes denoted as $\mathrm{X}_{N\times K}$ to indicate the number of rows (N) and columns (K).

2 Algebra with Matrices

Matrix algebra operates under similar principles to ordinary algebra—allowing addition, subtraction, multiplication, and division (through inversion)—but it also requires adherence to specific rules.

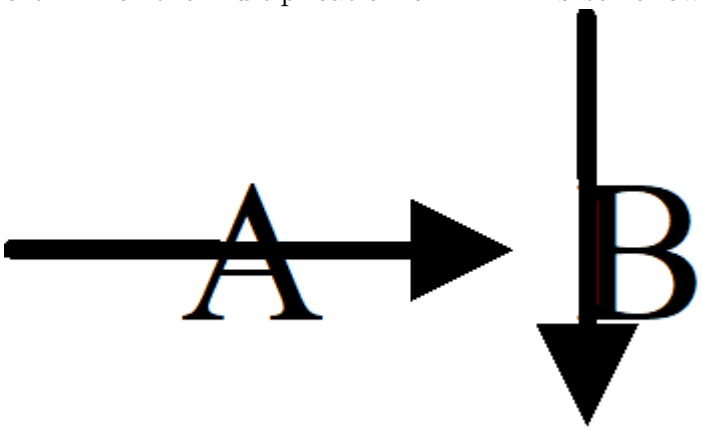
2.1 Matrix Multiplication

The core operation in matrix algebra is matrix multiplication, which combines elements from the rows of the first matrix with the columns of the second. For example, multiplying a 2×3 matrix by a 3×2 matrix yields:

```
X = np.random.randint(0,5,(2,3))
Y = np.random.randint(0,5,(3,2))
result = np.dot(X, Y)
print(X)
print(Y)
print(result)
```

```
[[4 0 4]
 [1 4 4]]
[[4 3]
 [1 4]
 [1 1]]
[[20 16]
 [12 23]]
```

What happens is that we sum the product of the elements in each row of the first matrix and each column of the second. You can for example check that element [0,0] of the result is the sum of the product of the first row of the first matrix, and the first column of the second. An easy way to remember this is to think of the multiplication of $A \times B$ is to follow the lines of the



letters:

Due to the rules for matrix multiplication, it requires the number of columns in the first matrix to match the number of rows in the second.

The matrix multiplication is different from the normal multiplication in Python. Normal multiplicaiton can be done with the normal multiplication operator `*`. It will then multiply each element in X with the corresponding element of Y, and both matrices must be of the same size:

```
X = np.random.randint(0,5,(2,3))
Y = np.random.randint(0,5,(2,3))
result = X*Y
print(X)
print(Y)
print(result)
```

```
[[2 3 2]
 [4 0 1]]
[[4 4 4]
 [2 0 2]]
[[ 8 12  8]
 [ 8  0  2]]
```

The reason for using the former method, is that the former is required for solving sets of equations.

2.2 Adding and Subtracting Matrices

Adding or subtracting matrices is straightforward; simply add or subtract corresponding elements. In Python, the multiplication requires numpy function, but if the matrices are numpy variables, subtraction and addition can be done with the normal operators.

```
import numpy as np

X = np.random.randint(0,100,(2,2))
Y = np.random.randint(0,100,(2,2))

# Addition of matrices
result_add = X + Y
print(X)
print(Y)
print(result_add)
```

```
[[ 2  1]
 [ 9 65]]
[[69 19]
 [87 28]]
[[71 20]
```

```
 [96 93]]
```

2.3 Dividing with a Matrix

While direct division isn't defined in matrix operations, we can achieve a similar result by multiplying by the inverse of a matrix. The inverse of a matrix X, denoted X^{-1}, satisfies:

$$X \times X^{-1} = I = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

where I is the identity matrix. Multiplying any matrix by I results in the original matrix, akin to multiplying any number by 1.

In practice, while the concept is straightforward, the actual calculation of a matrix inverse can become complex for larger matrices and is typically handled by computers. We will not go through the method of obtaining the inverse in this course, we will in stead just utilize the numpy funciton for calculating the inverse. Specifically, we use `np.linalg.inv(X)`. We can check that it actually complies with the definition like this:

```
X = np.random.randint(0,10,(3,3))
# Calculating inverse of X
X_inv = np.linalg.inv(X)

# Testing
np.round(np.dot(X_inv, X),1)
```

```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [-0.,  0.,  1.]])
```

2.4 Solving Equations with Matrix Algebra

The foundation we've established for matrix algebra now allows us to efficiently solve systems of equations. Consider solving the following pair of simultaneous equations:

$$x_{11}a_1 + x_{12}a_2 = b_1 x_{21}a_1 + x_{22}a_2 = b_2$$

Here, we know the values of x and b but need to find the values of a. These equations can be succinctly expressed using matrix notation:

$$\mathrm{X} \times \mathrm{a} = \mathrm{b}$$

where a and b are column vectors. Let us define the right hand side vector `b` and the coeficient matrix `X` randomly in python as

```
b =  np.random.randint(0,100,(2,1))
# Define matrix X
X =  np.random.randint(0,100,(2,2))
print(X)
print(b)
```

```
[[72 23]
 [33 81]]
[[14]
 [97]]
```

To solve for a, we use the inverse of X, provided it exists, and multiplies it with the left and right hand sides of the equation, just as we would divide with X on both sides to solve for a single equation:

$$\mathrm{X}^{-1} \times \mathrm{X} \times \mathrm{a} = \mathrm{X}^{-1}\mathrm{b}$$

Since we know that X^{-1} is the solution to $\mathrm{X}^{-1} \times \mathrm{X} = \mathrm{I}$, premultiplying with X^{-1} yields:

$$\mathrm{a} = \mathrm{X}^{-1}\mathrm{b}$$

Hence, we have found an easy way to solve any linear equation. We can test that it works in python. Let us first find `a` using this approach:

```
a = np.dot(np.linalg.inv(X), b)
a
```

```
array([[-0.21624285],
       [ 1.2856298 ]])
```

If you get a "Singular matrix" error its because we are generating `X` with a few random integers, which sometimes creates unsolvable systems, so just generate `X` and `b` again.

Now we can test, if the solution for a actually works, by applying it on the original equation $\mathrm{X} \times \mathrm{a} = \mathrm{b}$. This should yield the right hand side of th equation, `b`:

```
np.dot(X, a)
```

```
array([[14.],
       [97.]])
```

Compare this with the actual `b`:

```
np.dot(X, a)
```

```
array([[14.],
       [97.]])
```

Thus, we have identified an effective method to solve any system of equations, provided that X is invertible. If X cannot be inverted, it indicates that two or more equations are essentially identical, leading to an "underdetermined" system. In such cases, some equations are redundant, and there are not enough independent equations to determine the values of all variables. Remember the fundamental rule: we need an equal number of equations and unknowns to uniquely solve for each variable.

2.5 Transposing

Transposing a matrix involves swapping its rows and columns. For example, a 2×3 matrix:

$$\mathrm{X}_{2\times 3} = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix}$$

transposes to:

$$\mathrm{X}'_{2\times 3} = \begin{pmatrix} x_{11} & x_{21} \\ x_{12} & x_{22} \\ x_{13} & x_{23} \end{pmatrix}$$

where $'$ denotes the transposed matrix. For a column vector a, transposing and then multiplying by itself, a$'$a, calculates the sum of squares of its components.

```
# Example of matrix transposition
X_2x3 = np.random.randint(0,100,(2,3))
X_transposed = X_2x3.T
X_transposed
```

```
array([[81, 69],
       [91,  4],
       [83, 10]])
```

Transposition is often used to conform to the requirements of matrix multiplication, where the number of columns in the first matrix must match the number of rows in the second. If this is not the case, one might transpose the first matrix to facilitate multiplication.

3 Calculus and matrices

Deriving matrices follows similar principles to deriving polynomials. For instance:

$$\frac{d\left(a^2\sigma^2\right)}{da} = 2a\sigma^2$$

applies to scalar variables, and for a matrix and a column vector a, we have:

$$\frac{d\left(\mathrm{a}'\ \mathrm{a}\right)}{d\mathrm{a}'} = 2\ \mathrm{a}$$

assuming is symmetric. In practical terms, the derivative with respect to `a` here, given some values for `a`, is

```
# Derivation with matrix and vector
a = np.random.randint(0,100,(2,1))
Sigma = np.random.randint(0,100,(2,2))

# Derivative of a' Σ a with respect to a
derivative = 2 * np.dot(Sigma, a)
derivative
```

```
array([[ 4620],
       [10632]])
```

We can rewrite the matrix formulation in scalar form, to check that the rule is correct. The scalar form of a′ a is

$$\mathrm{a}' \, \mathrm{a} = \sum_{j=0}^{N} a_j \left(\sum_{i=0}^{N} a_i \sigma_{ij} \right)$$

You can verify that

$$\frac{d(\mathrm{a}' \, \mathrm{a})}{d\mathrm{a}} = 2[\sum_{i=0}^{N} a_i \sigma_{i0}, ..., \sum_{i=0}^{N} a_i \sigma_{iN}]$$

4 Optimal portfolios with more than one asset

We remember from above the previous chapter that with one asset, the optimal portfolio was

$$a = \frac{(mu - r)}{\lambda \sigma^2}$$

From this we concluded that:

1. The more risk-averse the person is, the less they should invest.
2. The larger the expected return of the asset, the more should be invested.
3. The greater the risk associated with the asset, represented by σ^2, the less should be invested.

Now, let us consider the optimal investments if we have more than one asset.

4.1 Optimal Portfolios with Any Number of Assets

Let us now assume that the investor in the previous section has a portfolio of K assets, not just one. Their wealth next period, assuming the entire amount is borrowed, is then expressed in matrix notation as:

$$W_1 = \mathrm{a}'\mathrm{x} - 1r$$

where a represents the portfolio weights, x represents the returns, and 1 is a column vector of ones, such that $1r$ is a column vector of the risk-free interest rate r. Recall from earlier that the investor aims to maximize the difference between expected return and variance:

$$\max_{\mathrm{a}} Z = \mathbb{E}W_1 - \pi\frac{1}{2}\operatorname{var}(W_1)$$

x now is a column vector of many normally distributed variables with different variances and expectations. We denote the expected returns by μ_i for asset i, and the associated vector of these returns by . Given a portfolio a, the expected return on the portfolio then becomes:

$$\mathbb{E}W_1 = \mathrm{a}'(\mathbb{E}\mathrm{x} - 1r) = \mathrm{a}\ - 1r$$

For the variance, the risk free return r is not relevant, since means are subtracted anyway. We define the covariance matrix, all the combinations of variance and covariance between the stocks as

$$\operatorname{var} W_1 = \ = \begin{bmatrix} \sigma_0 & \sigma_1 & \cdots & \sigma_N \\ \sigma_1 & \sigma_2 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_N & \cdots & \cdots & \sigma_{NN} \end{bmatrix}$$

Where σ_{ij} is the covariance between i and j, and σ_i^2 is the variance of asset i. This is the covariance matrix, denoted by the capital sigma, .

When a vector is normally distributed we write it as $\mathrm{x} \sim N(\ ,\ )$.

We have now derived expressions for $\mathbb{E}(W_1)$ and $\operatorname{var}(W_1)$ using matrix notation. Building on the concepts from the previous lecture, we can now formulate our portfolio optimization problem as:

$$\max_{\mathrm{a}} Z = \mathrm{a}(\ -1r) - \lambda \frac{1}{2} \mathrm{a}'\ \mathrm{a}$$

Taking the derivative with respect to a' yields the K first order conditions:

$$\frac{dZ}{d\mathrm{a}} = (\ -1r) - \lambda\ \mathrm{a} = 0$$

Hence, in optimum:

$$\mathrm{a} = \frac{1}{\lambda}(\ -1r)$$

By premultiplying with the inverse of , we obtain the optimal portfolio:

$$\mathrm{a} = \frac{1}{\lambda}^{-1}\ -1r)$$

Note that this formula looks very similar to the formula for an optimal portfolio with only one asset:

$$a = \frac{(mu - r)}{\pi\sigma^2}$$

In general, we may draw the same conclusions as in the case of one asset:

1. The more risk-averse the person is (large π), the less they should invest.
2. The larger the expected return the asset has, the more should be invested.
3. The more risk is associated with the asset, the less should be invested.

5 Empirical example

Vi bruker scriptmuligheten i Titlon for å hente data

```
import pandas as pd
#Query script for Microsoft SQL Server (MSSQL) client
import pymssql
con = pymssql.connect(host='titlon.uit.no',
```

```
                    user="esi000@uit.no",
                    password  = "39oQ!Fjzpuh$OZ2FpewRp",
                    database='OSE')
crsr=con.cursor()
crsr.execute("""
    SELECT  * FROM [OSE].[dbo].[equity]
    WHERE year([Date]) >= 2016
    ORDER BY [Name],[Date]
""")
r=crsr.fetchall()
df=pd.DataFrame(list(r), columns=[i[0] for i in crsr.description])
pd.to_pickle(df,'output/stocks.df')
print(df)
df = None


#YOU NEED TO BE CONNECTED TO YOUR INSTITUTION VIA VPN, OR BE AT THE
```

```
         Date  Internal code  SecurityId  CcompanyId Symbol       ISIN \
0     2019-07-12    2014128.0   1304857.0     12720.0   2020  BMG9156K1018
1     2019-07-15    2014128.0   1304857.0     12720.0   2020  BMG9156K1018
2     2019-07-16    2014128.0   1304857.0     12720.0   2020  BMG9156K1018
3     2019-07-17    2014128.0   1304857.0     12720.0   2020  BMG9156K1018
4     2019-07-18    2014128.0   1304857.0     12720.0   2020  BMG9156K1018
...          ...          ...         ...         ...    ...          ...
546507  2024-03-27    2014095.0         NaN         NaN   AASB  NO0010672181
546508  2024-03-28    2014095.0         NaN         NaN   AASB  NO0010672181
546509  2024-04-02    2014095.0         NaN         NaN   AASB  NO0010672181
546510  2024-04-03    2014095.0         NaN         NaN   AASB  NO0010672181
546511  2024-04-04    2014095.0         NaN         NaN   AASB  NO0010672181

                Name  BestBidPrice  BestAskPrice   Open  ... \
0         2020 Bulkers         82.05         83.00  87.20  ...
1         2020 Bulkers         80.05         81.00  83.00  ...
2         2020 Bulkers         80.55         81.00  81.00  ...
3         2020 Bulkers         80.40         81.00  81.00  ...
4         2020 Bulkers         77.40         80.00  80.01  ...
...                ...           ...           ...    ...  ...
546507  AASEN SPAREBANK        121.00        118.00 121.00  ...
```

```
546508  AASEN SPAREBANK        123.00     118.00   0.00  ...
546509  AASEN SPAREBANK        119.00     118.30  121.00  ...
546510  AASEN SPAREBANK        123.00     119.00  119.00  ...
546511  AASEN SPAREBANK        121.00     119.22   0.00  ...

      NOWA_DayLnrate  bills_3month_Lnrate     Sector  IN_OSEBX  Equity
0          0.000056             0.000056  Industrials       0    NaN
1          0.000055             0.000056  Industrials       0    NaN
2          0.000055             0.000056  Industrials       0    NaN
3          0.000055             0.000056  Industrials       0    NaN
4          0.000056             0.000057  Industrials       0    NaN
...             ...                  ...         ...     ...    ...
546507     0.000196             0.000191  Financials        0    NaN
546508     0.000196             0.000191  Financials        0    NaN
546509     0.000196             0.000192  Financials        0    NaN
546510     0.000196             0.000192  Financials        0    NaN
546511     0.000196             0.000192  Financials        0    NaN

        Debt  Earnings  debt_ratio  PE       ID
0        NaN       NaN         NaN NaN  1402537
1        NaN       NaN         NaN NaN  1402664
2        NaN       NaN         NaN NaN  1402784
3        NaN       NaN         NaN NaN  1413127
4        NaN       NaN         NaN NaN  1413224
...      ...       ...         ...  ..      ...
546507   NaN       NaN         NaN NaN  1767510
546508   NaN       NaN         NaN NaN  1774640
546509   NaN       NaN         NaN NaN  1774810
546510   NaN       NaN         NaN NaN  1767992
546511   NaN       NaN         NaN NaN  1768158

[546512 rows x 48 columns]
```

5.1 The portfolio front

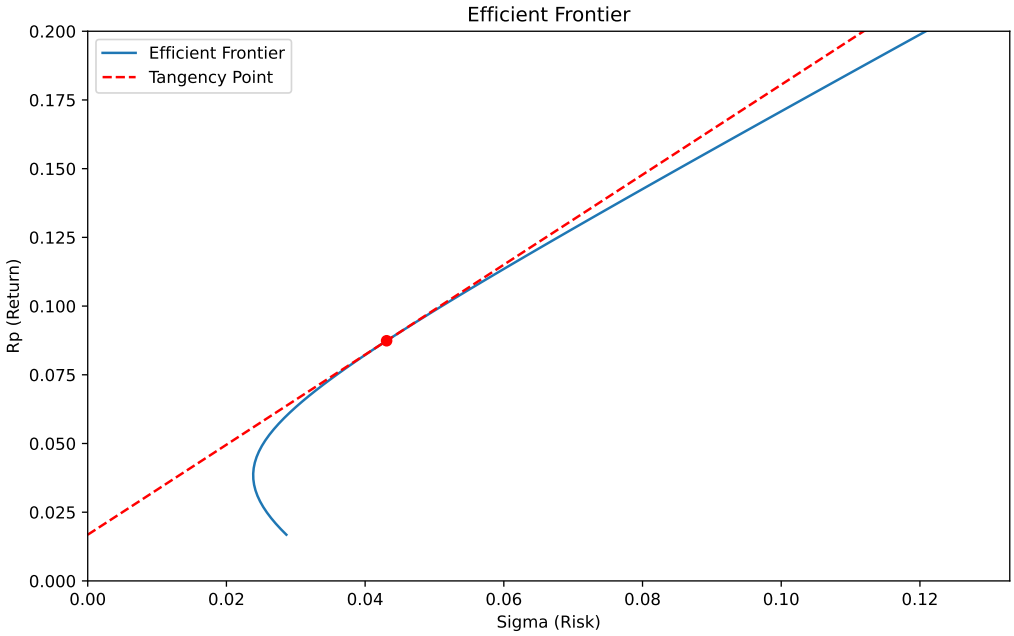
When the numnber of stocks becomes large, the chances that a few of them ar essentially identical risk-wise increases. Therefore, in this example, the dimension is reduced by the function `get_independent_portfolios`, in a way that we will come back to. In a very simplified way, a matrix R is

computed, that is multiplied with both the mean vector and the covariance matrix.

When that is done, we can calulate the optimal portfolio (C) and the portfolio front. The portfolio front is the minum variance given any level of return.

```
import functions
import pandas
import decomposition
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

MAX_AXIS = 0.2

df = pd.read_pickle('output/stocks.df')
# Defining risk free, cov matrix and mean vector
rf = np.exp(0.2/12)-1#df['NOWA_DayLnrate'].mean()


cov_matrix, means, df_month = functions.calc_moments(df)
df = None

# There are so many stocks, that many will be highly correlated
# This creates a singular matrix (too few genuine equations)
# It is therefore neccessary to do a little trick to reduce the
# Number of dimensions. Multiplying the matrix and means with
# R, reduces the dimension
R = decomposition.get_independent_portfolios(cov_matrix, 0.00001)
cov_matrix = R.T @ cov_matrix @ R
means = R.T @ means

# Create a vector of ones with the same length as the number of colu
ones = np.ones(cov_matrix.shape[0])

# Defining the sum of the covariance elements:
A = np.dot(ones.T, np.dot(np.linalg.inv(cov_matrix), ones))

# Defining the sum of the optimal portfolio, in order to
```

```
# normalize the portfolio size to one
B = np.dot(ones.T, np.dot(np.linalg.inv(cov_matrix), means-rf))

# Calculating the un-normalized optimal portfolio:
C = np.dot(means.T-rf, np.dot(np.linalg.inv(cov_matrix), means-rf))

#Creating plot
fig, ax = plt.subplots(figsize=(10, 6))

# Setting the range of rp values and sigma values
rp_values = np.linspace(0, MAX_AXIS, 100)


# Calculate and plot sigma values for each rp
sigma_values = 1/A + ((rp_values - abs(B)/A)**2) / (C - B**2/A)
ax.plot(sigma_values**0.5, rp_values+rf, label='Efficient Frontier')

# Calculate the tangency point of the normalized optimal
# portfolio and plotting it
tangency_rp = C/abs(B)
tangency_sigma =  1/A + ((tangency_rp - abs(B)/A)**2) / (C - B**2/A)

ax.plot(tangency_sigma**0.5, tangency_rp + rf, 'ro')

# Plotting the portfolio front
sigma_range = np.linspace(0, np.max(sigma_values**0.5), 100)
ax.plot(sigma_range, rf + sigma_range*tangency_rp/tangency_sigma**0.

# Setting the axis ranges, lables, title and legend:
ax.set_xlim([0, np.max(sigma_values**0.5)])
ax.set_ylim([0, np.max(rp_values)])
ax.set_xlabel('Sigma (Risk)')
ax.set_ylabel('Rp (Return)')
ax.set_title('Efficient Frontier')
ax.legend()
```


Efficient Frontier
Efficient Frontier
Tangency Point
Rp (Return)
Sigma (Risk)
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0.00
0.02
0.04
0.06
0.08
0.10
0.12