CUHK Business School
The Chinese University of Hong Kong

MSC IN FINANCE
PRE-TERM COURSE WEEK4:
Linear Algebra
And Statistical Tools in Python

Tutor: Yuan Lu
CUHK Department of Finance
yuan.lu@link.cuhk.edu.hk

Aug 8, 2023

1 Agenda

[...]

- Review of Linear Algebra
- Linear Regressions
  - Statsmodels
  - Scikit-learn
- Portfolio Optimization
  - Monte-Carlo Simulation
  - Scipy Optimize Module

2 Review of Linear Algebra

Linear algebra is essential for machine learning. This Jupyter notebook reviews some of the basic concepts and operations in lienar algebra. In addition, we also provide the Python codes for various operations in linear algebra.

2.1 Import Related Libraries

In [1]:

```
import pandas as pd
import numpy as np
```

2.2 Scalars, Vectors and Matrices

1. scalars are constants.

2. row vector is a row of scalars, column vector is a column of scalars
3. matrix is a collection of many row vectors (or column vectors)
4. scalar is a special case of vector, vector is a special case of matrix

In [2]:

```
# This is a scalar
a = 2
```

In [3]:

```
# This is a row vector
# It is important to do the double brakets
b = np.array([[1,2,5]])
display(b.shape,b)
```

```
(1, 3)

array([[1, 2, 5]])
```

In [4]:

```
# This is a column vector
c = np.array([[1],[2],[5]])
display(c.shape,c)
```

```
(3, 1)

array([[1],
       [2],
       [5]])
```

In [5]:

```
# It is important to note that b and c are 2-dimensional arrays in NumPy.
# If you want to create one-dimensional arrays, do this
b1 = np.array([1,2,5])
display(b1.shape,b1)
```

```
(3,)

array([1, 2, 5])
```

In [6]:

```
# You can convert 1-dimensional array to column vector or row vector
# using reshape
b2 = b1.reshape([1,3])
display(b2.shape,b2)

c2 = b1.reshape([3,1])
display(c2.shape,c2)
```

```
(1, 3)

array([[1, 2, 5]])

(3, 1)

array([[1],
       [2],
       [5]])
```

In [7]:

```
# You can also convert column or row vector to 1-dimensional array
b3 = b2.flatten()
display(b3.shape,b3)
c3 = c2.flatten()
display(c3.shape,c3)
```

```
(3,)

array([1, 2, 5])

(3,)

array([1, 2, 5])
```

In [8]:

```
# This is a 3x4 matrix
A = np.array([[1,2,5,-1],[2,0,4,6],[3,5,2,1]])
display(A.shape,A)
```

```
(3, 4)

array([[ 1,  2,  5, -1],
       [ 2,  0,  4,  6],
       [ 3,  5,  2,  1]])
```

2.3 Transpose

1. Transpose of a row vector is a column vector, transpose of a column vector is a row vector
2. Transpose of a matrix is a new matrix whose rows are the columns of the original matrix

$b = [1, 2, 5]$

In [9]:

```
c = b.T
display(c.shape,c)
```

(3, 1)

```
array([[1],
       [2],
       [5]])
```

$$A = \begin{bmatrix} 1 & 2 & 5 & -1 \\ 2 & 0 & 4 & 6 \\ 3 & 5 & 2 & 1 \end{bmatrix}$$

In [10]:

```
A1 = A.T
display(A1.shape,A1)
```

(4, 3)

```
array([[ 1,  2,  3],
       [ 2,  0,  5],
       [ 5,  4,  2],
       [-1,  6,  1]])
```

2.4 Some Special Matrices

- 0_n is an n-vector of zeros

In [11]:

```
zero_vec = np.zeros([4,1])
print(zero_vec)
```

```
[[0.]
 [0.]
 [0.]
 [0.]]
```

- 1_n is an n-vector of ones

In [12]:

```
one_vec = np.ones([4,1])
print(one_vec)
```

```
[[1.]
 [1.]
 [1.]
 [1.]]
```

- I_n is an $n \times n$ identity matrix (only ones on the diagonal and zeros elsewhere)

In [13]:

```
idtymat = np.eye(5)
print(idtymat)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

2.5 Additions and Subtractions

1. Additions and subtractions of two vectors and matrices are done on an element-by-element basis
2. It means the two vectors (or two matrices) must be of same dimension
3. However, numpy allows broadcasting, which allows you to add a column (or row) vector to a matrix.

In [14]:

```
b1 = np.array([[2,3,1]])
b2 = np.array([[4,2,0]])
b3 = b1+b2
print(b3)
```

```
[[6 5 1]]
```

In [15]:

```
A1 = np.array([[2,3,5],[1,2,3],[7,6,4],[5,2,1]])
A2 = np.array([[1,2,3],[7,8,9],[4,5,6],[2,4,2]])
A3 = A1+A2
print(A3)
```

```
[[ 3  5  8]
 [ 8 10 12]
 [11 11 10]
 [ 7  6  3]]
```

In [16]:

```
# Here is an example of array broadcasting
display(b1,A1,A1+b1)
```

```
array([[2, 3, 1]])
```

```
array([[2, 3, 5],
       [1, 2, 3],
       [7, 6, 4],
       [5, 2, 1]])
```

```
array([[4, 6, 6],
       [3, 5, 4],
       [9, 9, 5],
       [7, 5, 2]])
```

2.6 Dot Product and Matrix Multiplication

1. We can multiply two vectors together with dot product $a'b = \sum_{i=1}^{n} a_i b_i$.
2. We can multiply a matrix with a vector If A is $m \times n$ and b is $n \times 1$, then $c = Ab$ is an $m \times 1$ vector, with $c_i = \sum_{j=1}^{n} a_{i,j} b_j, \quad i = 1, \ldots, m.$
3. We can multiply two matrices together If A is $m \times n$ and B is $n \times p$, then $C = AB$ is an $m \times p$ matrix, with $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}, \quad i = 1, \ldots, m, \ j = 1, \ldots, p.$
4. Matrix multiplication can only be performed if the number of columns of the left matrix is the same as the number of rows of the right matrix.

In [17]:

```
a = np.array([[3],[2],[1]])
b = np.array([[4],[5],[6]])
c = np.dot(a.T,b)
display(a.T,b,c)
```

```
array([[3, 2, 1]])

array([[4],
       [5],
       [6]])

array([[28]])
```

In [18]:

```
A = np.array([[2,3,1],[3,2,7]])
c = A.dot(b)
display(A,b,c)
```

```
array([[2, 3, 1],
       [3, 2, 7]])

array([[4],
       [5],
       [6]])

array([[29],
       [64]])
```

In [19]:

```
# An alternative way of doing the above
A = np.array([[2,3,1],[3,2,7]])
c = np.matmul(A,b)
display(A,b,c)
```

```
array([[2, 3, 1],
       [3, 2, 7]])

array([[4],
       [5],
       [6]])

array([[29],
       [64]])
```

In [20]:

```
B = np.array([[3,2,1,4],[4,3,2,7],[3,0,-1,2]])
C = np.matmul(A,B)
display(A,B,C)
```

```
array([[2, 3, 1],
       [3, 2, 7]])

array([[ 3,  2,  1,  4],
       [ 4,  3,  2,  7],
       [ 3,  0, -1,  2]])

array([[21, 13,  7, 31],
       [38, 12,  0, 40]])
```

2.7 Matrix Inverse

1. Given an $n \times n$ square matrix A, we find a matrix B such that $AB = I_n$, we call $B = A^{-1}$.
2. Note that $AA^{-1} = I_n$ but $A^{-1}A = I_n$ also holds.

In [21]:

```
A = np.array([[3,2,1,4],[4,3,2,7],[3,0,-1,2],[2,1,5,4]])
B = np.linalg.inv(A)
display(A,B)
```

```
array([[ 3,  2,  1,  4],
       [ 4,  3,  2,  7],
       [ 3,  0, -1,  2],
       [ 2,  1,  5,  4]])

array([[ 0.91176471, -0.64705882,  0.20588235,  0.11764706],
       [ 1.38235294, -0.52941176, -0.55882353, -0.17647059],
       [ 0.32352941, -0.29411765, -0.08823529,  0.23529412],
       [-1.20588235,  0.82352941,  0.14705882, -0.05882353]])
```

In [22]:

```
np.round(B,4)
```

Out[22]:

```
array([[ 0.9118, -0.6471,  0.2059,  0.1176],
       [ 1.3824, -0.5294, -0.5588, -0.1765],
       [ 0.3235, -0.2941, -0.0882,  0.2353],
       [-1.2059,  0.8235,  0.1471, -0.0588]])
```

$AB = I_n$

In [23]:

```
C = np.matmul(A,B)
np.round(C,4)
```

Out[23]:

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

$$BA = I_n$$

In [24]:

```
C1 = np.matmul(B,A)
np.round(C1,4)
```

Out[24]:

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

2.8 Solving a Linear System of Equations

1. Matrix inverse allows us to solve a system of linear equations.
2. If we want to solve:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 &= b_1, \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 &= b_2, \\ a_{3,1}x_2 + a_{3,2}x_2 + a_{3,3}x_3 &= b_3, \end{aligned}$$

we can simply do $x = A^{-1}b$ if A is invertible.

In [25]:

```
A = np.array([[2,3,4],[5,-1,2],[2,3,6]])
b = np.array([[2],[4],[6]])
x = np.matmul(np.linalg.inv(A),b)
display(A,b,x)
```

```
array([[ 2,  3,  4],
       [ 5, -1,  2],
       [ 2,  3,  6]])
```

```
array([[2],
       [4],
       [6]])
```

```
array([[-0.35294118],
       [-1.76470588],
       [ 2.        ]])
```

$$Ax = b$$

In [26]:

```
np.matmul(A,x)
```

Out[26]:

```
array([[2.],
       [4.],
       [6.]])
```

2.9 Trace and Determinant

1. Trace of a matrix is the sum of its diagonal elements.
2. Determinant of a 2×2 matrix
$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$
is $|A| = ad - bc$.
3. For general $n \times n$ matrix, see Wikipedia (https://en.wikipedia.org/wiki/Determinant) for definition of its determinant

In [27]:

```
t = np.trace(A)
d = np.linalg.det(A)
display(A,t,d)
```

```
array([[ 2,  3,  4],
       [ 5, -1,  2],
       [ 2,  3,  6]])
```

```
7
```

```
-34.00000000000001
```

2.10 Positive Definite Matrix

1. A symmetric $n \times n$ matrix A is positive definite if $x' Ax > 0$ for any non-zero n-vector x.
2. An example of positive definite matrix is covariance matrix (or correlation matrix).
3. For a positive definite matrix, its determinant is positive.

In [28]:

```
A = np.array([[3,2,1],[2,4,-1],[1,-1,5]])
A
```

Out[28]:

```
array([[ 3,  2,  1],
       [ 2,  4, -1],
       [ 1, -1,  5]])
```

In [29]:

```
# Use a vector x, and compute x'Ax, check if it is positive
x = np.array([[2],[3],[-7]])
np.dot(x.T,A.dot(x)).item()
```

Out[29]:

```
331
```

In [30]:

```
np.linalg.det(A)
```

Out[30]:

```
29.000000000000018
```

3 Linear Regressions

Based on Python library: Statsmodels and Sklearn.

Examples from the website: https://blog.quantinsti.com/linear-regression-market-data-python-r/ (https://blog.quantinsti.com/linear-regression-market-data-python-r/)

In [31]:

```
### Import the required libraries

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

## To use statsmodels for linear regression
import statsmodels.formula.api as smf

## To use sklearn for linear regression
from sklearn.linear_model import LinearRegression
```

We work with the historical returns of Coca-Cola (NYSE: KO), its competitor PepsiCo (NASDAQ: PEP), the US Dollar index (ICE: DX) and the SPDR S&P 500 ETF (NYSEARCA: SPY).

In [32]:

```
# Read data from csv file, the file contains the log returns.
df=pd.read_csv("data_linear_reg.csv")
df
```

Out[32]:

	Date	spy	ko	pep	usdx
0	2020-08-04	0.003855	0.008388	0.005617	-0.001284
1	2020-08-05	0.006192	0.011288	-0.008914	-0.005907
2	2020-08-06	0.006662	0.005491	-0.002866	-0.000431
3	2020-08-07	0.000718	0.006717	0.006456	0.006981
4	2020-08-10	0.002984	-0.001675	-0.005574	0.001497
...	...	...	...	...	...
747	2023-07-25	0.002726	-0.003368	0.002302	0.000000
748	2023-07-26	0.000154	0.012770	0.001253	-0.004549
749	2023-07-27	-0.006652	-0.009722	-0.016153	0.008685
750	2023-07-28	0.009743	0.000640	0.009397	-0.001475
751	2023-07-31	0.001902	-0.008842	-0.015089	0.002359

752 rows × 5 columns

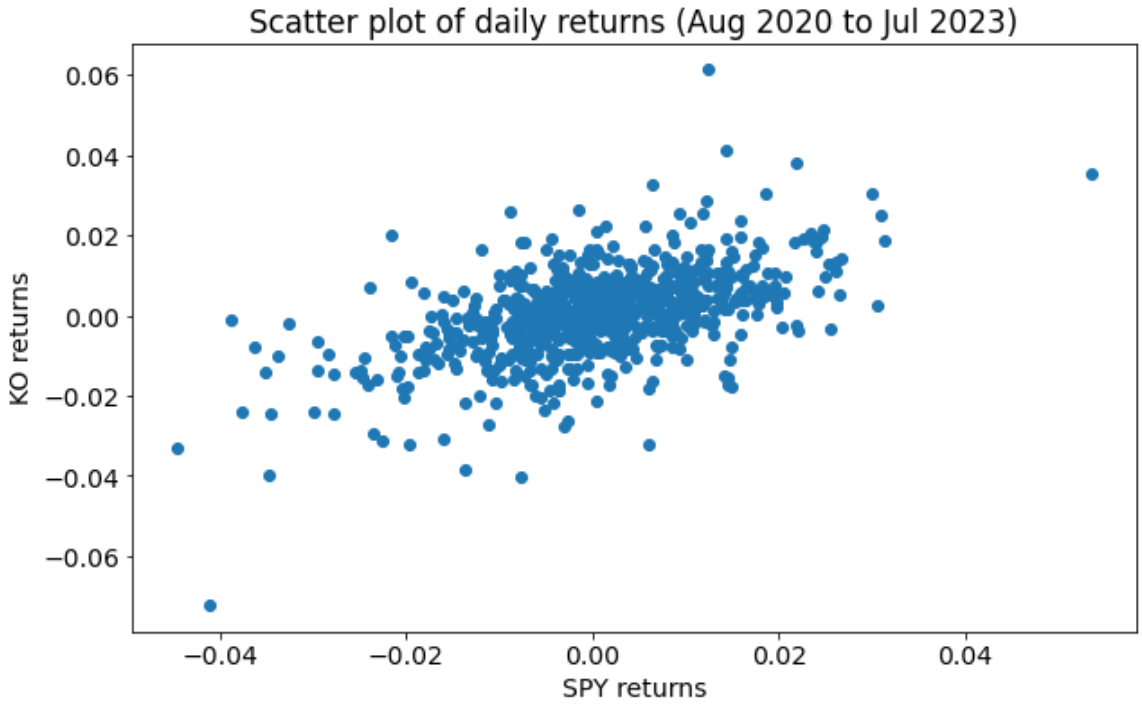
In [33]:

```
# # Online Fetch data and calculate log returns. If you've read data from csv file,
please skip the part.

# import yfinance as yf

# ########################################################
# ## Fetch data from yfinance
# ## 3-year daily data for Coca-Cola, SPY, Pepsi, and USD index

# end1 = datetime.date(2023, 8, 1)
# start1 = end1 - pd.Timedelta(days = 365 * 3)

# ko_df = yf.download("KO", start = start1, end = end1, progress = False)
# spy_df = yf.download("SPY", start = start1, end = end1, progress = False)
# pep_df = yf.download("PEP", start = start1, end = end1, progress = False)
# usdx_df = yf.download("DX-Y.NYB", start = start1, end = end1, progress = False)

# ########################################################
# ## Calculate log returns for the period based on Adj Close prices

# ko_df['ko'] = np.log(ko_df['Adj Close'] / ko_df['Adj Close'].shift(1))
# spy_df['spy'] = np.log(spy_df['Adj Close'] / spy_df['Adj Close'].shift(1))
# pep_df['pep'] = np.log(pep_df['Adj Close'] / pep_df['Adj Close'].shift(1))
# usdx_df['usdx'] = np.log(usdx_df['Adj Close'] / usdx_df['Adj Close'].shift(1))

# ########################################################
# ## Create a dataframe with X's (spy, pep, usdx) and Y (ko)

# df = pd.concat([spy_df['spy'], ko_df['ko'],
#                 pep_df['pep'], usdx_df['usdx']], axis = 1).dropna()

# # ## Save the csv file. Good practice to save data files after initial processing
# # df.to_csv("data_linear_reg.csv")

# ########################################################
```

We first create a scatter plot of the SPY and KO returns to better understand how they are related.

In [34]:

```
## A scatter plot of X (spy) and Y (ko) to examine the nature of their relationship visually

plt.figure(figsize = (10, 6))
plt.rcParams.update({'font.size': 14})
plt.xlabel("SPY returns")
plt.ylabel("KO returns")
plt.title("Scatter plot of daily returns (Aug 2020 to Jul 2023)")
plt.scatter(df['spy'], df['ko'])
plt.show()
```



We also calculate correlations between different variables to analyze the strength of the linear relationships here.

In [35]:

```
### Calculate correlation between Xs and Y

df.corr()
```

Out[35]:

	spy	ko	pep	usdx
spy	1.000000	0.539683	0.543284	-0.376342
ko	0.539683	1.000000	0.743223	-0.230901
pep	0.543284	0.743223	1.000000	-0.175518
usdx	-0.376342	-0.230901	-0.175518	1.000000

3.1 Statsmodels

In [36]:

```
### Fit a simple linear regression model to the data using statsmodels

### Create an instance of the class OLS
slr_sm_model = smf.ols('ko ~ spy', data=df)

### Fit the model (statsmodels calculates beta_0 and beta_1 here)
slr_sm_model_ko = slr_sm_model.fit()

### Summarize the model

print(slr_sm_model_ko.summary())

param_slr = slr_sm_model_ko.params
```

OLS Regression Results

Dep. Variable:	ko	R-squared:	0.291
Model:	OLS	Adj. R-squared:	0.290
Method:	Least Squares	F-statistic:	308.2
Date:	Tue, 08 Aug 2023	Prob (F-statistic):	4.61e-58
Time:	23:14:48	Log-Likelihood:	2458.4
No. Observations:	752	AIC:	-4913.
Df Residuals:	750	BIC:	-4904.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	0.0003	0.000	0.745	0.457	-0.000	0.001
spy	0.5152	0.029	17.556	0.000	0.458	0.573

Omnibus:	72.813	Durbin-Watson:	1.892
Prob(Omnibus):	0.000	Jarque-Bera (JB):	455.258
Skew:	-0.066	Prob(JB):	1.39e-99
Kurtosis:	6.810	Cond. No.	87.3

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [37]:

```
### Print the parameter estimates of the simple linear regression model

print("========================================================================")
print("The intercept in the statsmodels regression model is", \
      np.round(param_slr.Intercept, 4))
print("The slope in the statsmodels regression model is", \
      np.round(param_slr.spy, 4))
print("========================================================================")
```

```
========================================================================
The intercept in the statsmodels regression model is 0.0003
The slope in the statsmodels regression model is 0.5152
========================================================================
```

As the results above, the model for KO's returns can be expressed as

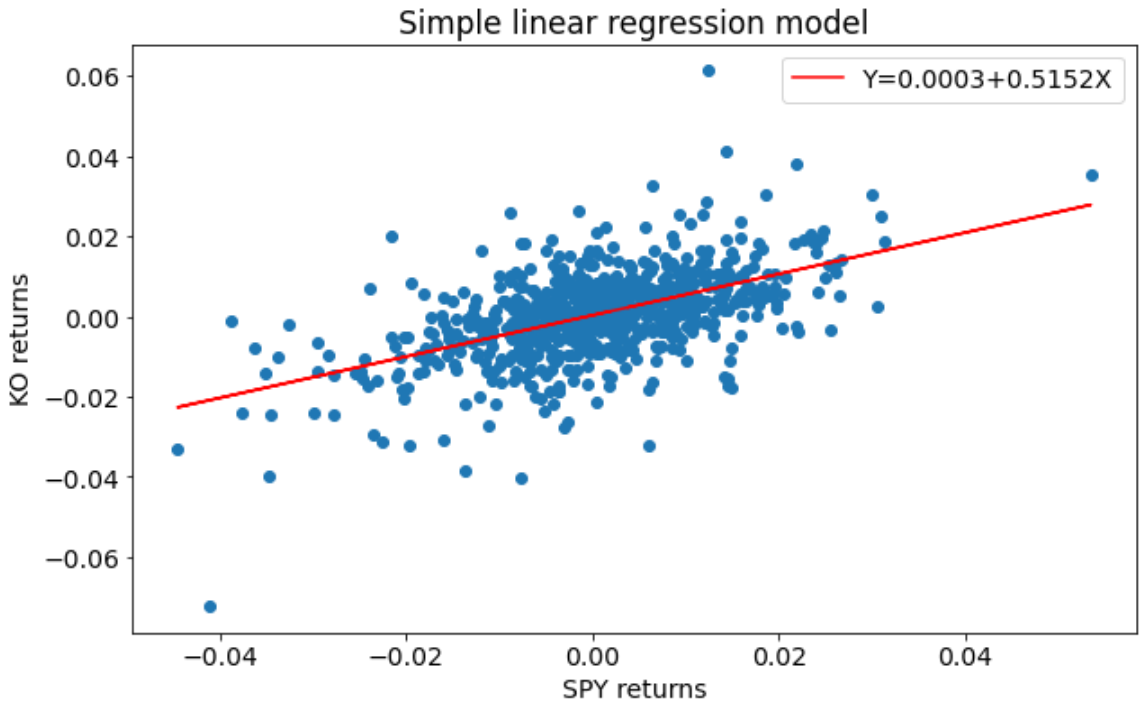
$$\hat{Y}_i = 0.0003 + 0.5152 X_i$$

The caret mark or ^ above the Y_i indicates that it is the fitted (or predicted) value of KO's returns as opposed to the observed returns. We obtain it by computing the RHS of the above equation.

We plot the best fit line (i.e. the regression line) for the data set as shown below.

In [38]:

```
### Linear regression plot of X (spy) and Y (ko)
plt.figure(figsize = (10, 6))
plt.rcParams.update({'font.size': 14})
plt.xlabel("SPY returns")
plt.ylabel("KO returns")
plt.title("Simple linear regression model")
plt.scatter(df['spy'],df['ko'])
plt.plot(df['spy'], param_slr.Intercept+param_slr.spy * df['spy'],
         label='Y={:.4f}+{:.4f}X'.format(param_slr.Intercept, param_slr.spy),
         color='red')
plt.legend()
plt.show()
```



For multiple linear regression, we use SPY, PEP and DX returns as explanatory variables to explain KO returns.

In [39]:

```
### Fit a multiple linear regression model to the data using statsmodels

### Create an instance of the class OLS
mlr_sm_model = smf.ols('ko ~ spy + pep + usdx', data=df)

### Fit the model (statsmodels calculates beta_0, beta_1, beta_2, beta_3 here)
mlr_sm_model_ko = mlr_sm_model.fit()

### Summarize the model

print(mlr_sm_model_ko.summary())
```

OLS Regression Results

Dep. Variable:	ko	R-squared:	0.581
Model:	OLS	Adj. R-squared:	0.579
Method:	Least Squares	F-statistic:	345.9
Date:	Tue, 08 Aug 2023	Prob (F-statistic):	7.91e-141
Time:	23:14:48	Log-Likelihood:	2656.1
No. Observations:	752	AIC:	-5304.
Df Residuals:	748	BIC:	-5286.
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	8.604e-05	0.000	0.332	0.740	-0.000	0.001
spy	0.1635	0.029	5.715	0.000	0.107	0.220
pep	0.6692	0.029	22.713	0.000	0.611	0.727
usdx	-0.1285	0.061	-2.112	0.035	-0.248	-0.009

Omnibus:	196.858	Durbin-Watson:	1.970
Prob(Omnibus):	0.000	Jarque-Bera (JB):	1903.019
Skew:	0.884	Prob(JB):	0.00
Kurtosis:	10.590	Cond. No.	239.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [40]:

```
### Print the parameter estimates of the muliple linear regression model

param_mlr = mlr_sm_model_ko.params

print("========================================================================")
print("The intercept and slopes in the statsmodels regression model are")
print("\n")
print(param_mlr)
print("========================================================================")
```

```
========================================================================
The intercept and slopes in the statsmodels regression model are


Intercept    0.000086
spy          0.163520
pep          0.669178
usdx        -0.128472
dtype: float64
========================================================================
```

As per the results above, the model for KO's returns can be expressed as

$$\hat{Y}_i = 0.0001 + 0.1635X_{1,i} + 0.6692X_{2,i} - 0.1285X_{3,i}$$

The caret mark above the Y_i indicates that it is the fitted (or predicted) value of KO's returns based on the historical returns on SPY, PEP, and USDX.

3.2 Scikit-Learn

We can also use the scikit-learn library for regressional analysis. Unlike statsmodels, here, we don't have the option of a full summary table with a detailed output.

Notice also how the features and output specified are done so in the .fit() routine (with features first and then the output variable). In statsmodels, we specified the parameters (in reverse order) when we instantiated the OLS class.

The coefficients obtained as you see here and in multiple regression, are the same with both libraries.

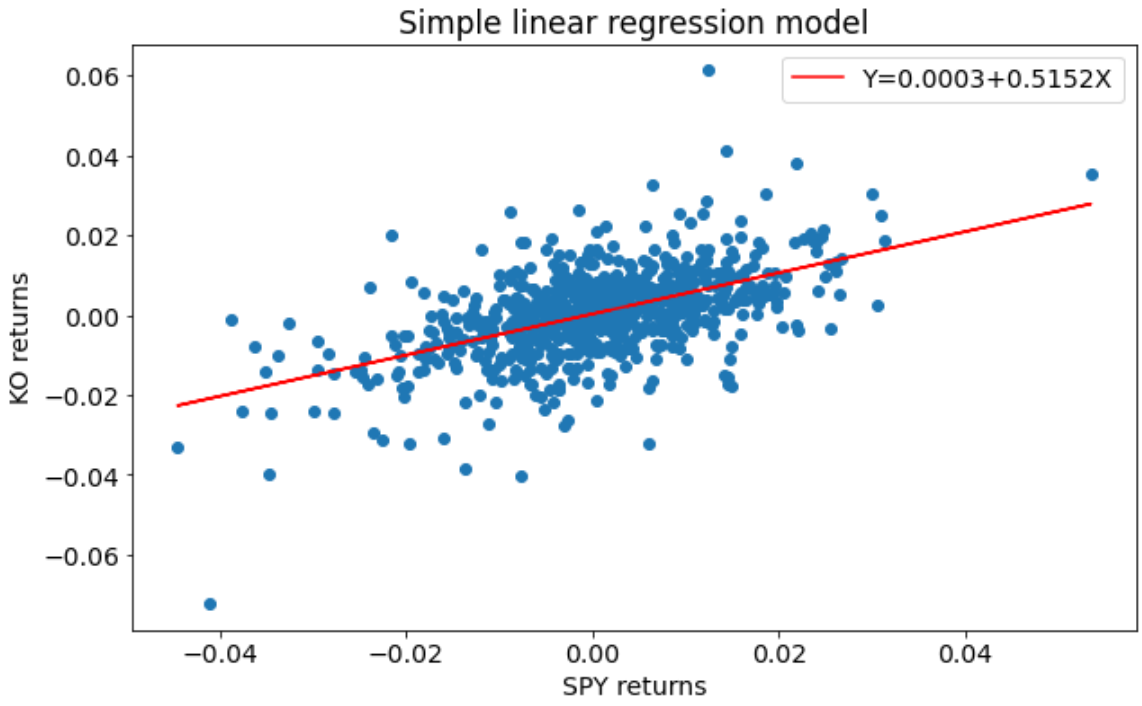
In [41]:

```
### Create an instance of the class LinearRegression()
slr_skl_model = LinearRegression()

### Fit the model (sklearn calculates beta_0 and beta_1 here)

X = df['spy'].values.reshape(-1, 1)
slr_skl_model_ko = slr_skl_model.fit(X, df['ko'])

print("The intercept in the sklearn regression result is",
np.round(slr_skl_model_ko.intercept_, 4))
print("The slope in the sklearn regression model is",
np.round(slr_skl_model_ko.coef_[0], 4))
```

```
The intercept in the sklearn regression result is 0.0003
The slope in the sklearn regression model is 0.5152
```

In [42]:

```
### Linear regression plot of X (spy) and Y (ko)

plt.figure(figsize = (10, 6))
plt.rcParams.update({'font.size': 14})
plt.xlabel("SPY returns")
plt.ylabel("KO returns")
plt.title("Simple linear regression model")
plt.scatter(df['spy'], df['ko'])
plt.plot(X, slr_skl_model.predict(X), label='Y={:.4f}+
{:.4f}X'.format(slr_skl_model_ko.intercept_, slr_skl_model_ko.coef_[0]), color='red')
plt.legend()
plt.show()
```



In [43]:

```
### Print the parameter estimates of the simple linear regression model

print("======================================================================")
print("The intercept in the sklearn regression result is", \
      np.round(slr_skl_model_ko.intercept_, 4))
print("The slope in the sklearn regression model is", \
      np.round(slr_skl_model_ko.coef_[0], 4))
print("======================================================================")
```

```
======================================================================
The intercept in the sklearn regression result is 0.0003
The slope in the sklearn regression model is 0.5152
======================================================================
```

As the results above, the model for KO's returns can be expressed as

$$\hat{Y}_i = 0.0003 + 0.5152 X_i$$

The regression estimations in sklearn are the same as those in statsmodels.

In [44]:

```
### Fit a multiple linear regression model to the data using sklearn

### Create an instance of the class LinearRegression()
mlr_skl_model = LinearRegression()

### Fit the model (sklearn calculates beta_0 and beta_1 here)

X = df[['spy', 'pep', 'usdx']]
y = df['ko']
mlr_skl_model_ko = mlr_skl_model.fit(X, y)
```

In [45]:

```
print("The intercept in the sklearn regression result is", \
      mlr_skl_model_ko.intercept_)
print("The slope in the sklearn regression model is", \
      mlr_skl_model_ko.coef_)
```

```
The intercept in the sklearn regression result is 8.603960244215307e-05
The slope in the sklearn regression model is [ 0.16352043  0.66917759 -0.1
2847178]
```

As per the results above, the model for KO's returns can be expressed as

$$\hat{Y}_i = 0.0001 + 0.1635 X_{1,i} + 0.6692 X_{2,i} - 0.1285 X_{3,i}$$

The regression estimations in sklearn are the same as those in statsmodels.

4 Portfolio Optimization

- **Modern Portfolio Theory**

- Harry M. Markowitz (August 24, 1927 – June 22, 2023), Nobel-Winning Pioneer of Modern Portfolio Theory.

- Investors can optimize their portfolio by selecting a combination of assets that maximizes expected return for a given level of risk or minimizes risk for a given level of expected return.
- Diversify the portfolio across different assets and sectors, which helps to reduce the overall volatility and risk of the portfolio.

- Two key components: **expected return** and **risk** of a portfolio. By using Python libraries such as SciPy Optimize and the Monte Carlo Method, we can create a more efficient and accurate optimization process compared to traditional methods.
- The **SciPy Optimize**: a library that provides optimization algorithms for a wide range of optimization problems, including linear and nonlinear programming, global and local optimization, and optimization of differential equations.
- The **Monte Carlo Method**: a statistical method used to simulate the behavior of a system by generating a large number of random scenarios.

4.1 Import Packages and Read Data

In [46]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
```

In [47]:

```
# Read Data from csv file. Specify the first column as the index for the DataFrame
adj_close=pd.read_csv('adj_close.csv',index_col=0)

symbols = adj_close.columns.tolist()  # ['AAPL','MSFT','AMZN','GOOGL'] #our sample stocks
num_assets = len(symbols)  #the number of assets

adj_close
```

Out[47]:

Date	AAPL	AMZN	GOOGL	MSFT
2015-01-02	24.531769	15.426000	26.477501	40.620655
2015-01-05	23.840658	15.109500	25.973000	40.247131
2015-01-06	23.842915	14.764500	25.332001	39.656406
2015-01-07	24.177242	14.921000	25.257500	40.160248
2015-01-08	25.106186	15.023000	25.345501	41.341682
...	...	...	...	...
2019-12-24	69.421059	89.460503	67.221497	152.286896
2019-12-26	70.798401	93.438499	68.123497	153.535156
2019-12-27	70.771538	93.489998	67.732002	153.815781
2019-12-30	71.191574	92.344498	66.985497	152.490082
2019-12-31	71.711739	92.391998	66.969498	152.596542

1258 rows × 4 columns

In [48]:

```
# # Fetch Data online. Skip the part if you've read data from csv file
# from pandas_datareader import data as pdr
# import yfinance as yf

# yf.pdr_override()

# # symbols = ['AAPL','GOOG','TSLA','MSFT','META','AMZN']
# symbols = ['AAPL','MSFT','AMZN','GOOGL']
# num_assets = len(symbols)  #the number of assets

# data = pdr.get_data_yahoo(symbols,start="2015-01-01", end="2020-01-01")
# adj_close=data['Adj Close']
# adj_close
```

4.2 Calculate Mean and Variance of returns

- **Expected return** is the average return that an investor can expect to earn on an investment over a given period of time.
- **Arithmatic returns**

$$R_{i,t} = \frac{(adjClose_{i,t} - adjClose_{i,t-1})}{adjClose_{i,t-1}}$$

- **Log returns**

$$R_{i,t} = log(adjClose_{i,t}) - log(adjClose_{i,t-1})$$

In [49]:

```
# Calculate average returns for stocks as expected returns
def calc_returns(price_data, ret_type="arithmatic"):
    """
    Parameters
        price_data: price timeseries pd.DataFrame object.

        ret_type:   return calculation type. \"arithmatic\" or \"log\"

    Returns:
        returns timeseries pd.DataFrame object
    """

    if ret_type=="arithmatic":
        ret = price_data.pct_change().dropna()
    elif ret_type=="log":
        ret = np.log(price_data/price_data.shift()).dropna()
    else:
        raise ValueError("ret_type: return calculation type is not valid. use
\"arithmatic\" or \"log\"")

    return(ret)
```

In [50]:

```
daily_ret = calc_returns(adj_close,  ret_type="log")
daily_ret
```

Out[50]:

Date	AAPL	AMZN	GOOGL	MSFT
2015-01-05	-0.028577	-0.020731	-0.019238	-0.009238
2015-01-06	0.000095	-0.023098	-0.024989	-0.014786
2015-01-07	0.013925	0.010544	-0.002945	0.012625
2015-01-08	0.037702	0.006813	0.003478	0.028994
2015-01-09	0.001072	-0.011818	-0.012286	-0.008441
...	...	...	...	...
2019-12-24	0.000950	-0.002116	-0.004601	-0.000191
2019-12-26	0.019646	0.043506	0.013329	0.008163
2019-12-27	-0.000380	0.000551	-0.005763	0.001826
2019-12-30	0.005918	-0.012328	-0.011083	-0.008656
2019-12-31	0.007280	0.000514	-0.000239	0.000698

1257 rows × 4 columns

After calculate the stock returns, we then comput the mean μ and covariance matrics.

In [51]:

```
def calc_returns_stats(returns):
    """
    Parameters
        returns: returns timeseries pd.DataFrame object

    Returns:
        mean_returns: Avereage of returns
        cov_matrix: returns Covariance matrix
    """
    mean_returns = returns.mean()
    cov_matrix = returns.cov()
    return (mean_returns, cov_matrix)
```

In [52]:

```
mean_returns, cov_matrix = calc_returns_stats(daily_ret)
display(mean_returns, cov_matrix)
```

```
AAPL     0.000853
AMZN     0.001424
GOOGL    0.000738
MSFT     0.001053
dtype: float64
```

	AAPL	AMZN	GOOGL	MSFT
AAPL	0.000246	0.000142	0.000122	0.000132
AMZN	0.000142	0.000333	0.000176	0.000167
GOOGL	0.000122	0.000176	0.000221	0.000144
MSFT	0.000132	0.000167	0.000144	0.000215

4.2.1 Portfolio Mean and Variance

1. Suppose R_t is an n-vector of returns of n assets at time t.
2. Let $\mu = E[R_t]$ and $\Sigma = \mathrm{Var}[R_t]$.
3. Let w be an n-vector of weights in the n assets.
4. **Portfolio mean** μ_P :

$$\mu_P = w_0\mu_0 + w_1\mu_1 + w_2\mu_2 + \ldots + w_n\mu_n = w^T\mu = \sum_{i=1}^{n} w_i\mu_i$$

5. **Portfolio variance** σ_P^2 :

$$\sigma_P^2 = w^T\Sigma w = \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j \sigma_{ij}$$

In [53]:

```
def portfolio(weights, mean_returns, cov_matrix):

    portfolio_return = np.dot(weights.reshape(1,-1),
mean_returns.values.reshape(-1,1))
    portfolio_var = np.dot(np.dot(weights.reshape(1,-1), cov_matrix.values),
weights.reshape(-1,1))

    return (portfolio_return.item()*252,portfolio_var.item()*252)
```

Scenario 1 : If we hold each asset equally-weighted. `weights=[0.25,0.25,0.25,0.25]`

In [54]:

```
weights=np.array([1/len(symbols) for i in range(len(symbols))])
returns,variances=portfolio(weights, mean_returns, cov_matrix)

rf=0.01 # assume annualized risk-free interest rate is 1%
SharpeRatio=(returns-rf)/np.sqrt(variances)

print(f'The Expected Return of the Portfolio is {np.round(returns*100,2)}% and
Variance is {np.round(variances*100,2)}%.\n When risk-free interest rate is
{np.round(rf*100,2)}%, the Sharpe Ratio of the Portfolio is
{np.round(SharpeRatio,2)}')
```

```
The Expected Return of the Portfolio is 25.63% and Variance is 4.38%.
 When risk-free interest rate is 1.0%, the Sharpe Ratio of the Portfolio i
s 1.18
```

Scenario 2 : If we hold each asset by `weights=[0.1,0.2,0.3,0.4]`

In [55]:

```
weights=np.array([0.1,0.2,0.3,0.3])
returns,variances=portfolio(weights, mean_returns, cov_matrix)

rf=0.01 # assume annualized risk-free interest rate is 1%
SharpeRatio=(returns-rf)/np.sqrt(variances)

print(f'The Expected Return of the Portfolio is {np.round(returns*100,2)}% and
Variance is {np.round(variances*100,2)}%.\n When risk-free interest rate is
{np.round(rf*100,2)}%, the Sharpe Ratio of the Portfolio is
{np.round(SharpeRatio,2)}')
```

```
The Expected Return of the Portfolio is 22.87% and Variance is 3.61%.
 When risk-free interest rate is 1.0%, the Sharpe Ratio of the Portfolio i
s 1.15
```

Scenario n : If we hold each asset by $weights = [w_1, w_2, w_3, w_4]$

- How to determine the **optimal weights**?
  - Maximize expected return while obtaining a certain level of risk.
  - Minimize risk (variance) while obtaining a certain level of return.

1. Monte Carlo Simulations

- Generate random portfolios by assigning random weights to each asset.
- Repeat multiple times and obtain the optimal weights for portfolio with maximum return and minimum risk

2. Scipy.Optimize Module

- Utilize the minimize method in the module to "minimum variance portfolio" and "maximum Sharpe ratio" optimization problem.

4.3 Caluculate Optimal Weights

4.3.1 Monte Carlo Method

- The **Monte Carlo Method** is based on the idea of simulating the behavior of a system by generating a large number of random scenarios. The Monte Carlo Method is used in MPT to generate random portfolios and calculate their expected return and risk.

- Steps for implementing the Monte Carlo Method in MPT:
  - Generate random portfolios by assigning weights to each asset.
  - Calculate expected return and risk for each portfolio.
  - Identify the portfolio with maximum return and minimum risk.
  - Repeat the process multiple times for robustness.

- **Advantages** of using the Monte Carlo Method in MPT: Handles non-linear problems and portfolios with many assets. Can estimate optimal solutions with uncertain or unknown return and risk.
- **Considerations** : The Monte Carlo Method is computationally expensive and time-consuming, requiring a large number of simulations. It does not guarantee global optima but can provide a good approximation.

In [56]:

```
rf = 0.01 # assume annualized risk-free rate is 1%

# list containers to store random portfolios
p_ret = []
p_vol = []
p_weights = []


# Simulation times
num_portfolios = 50000
```

In [57]:

```
for i in range(num_portfolios):

    # Form random portfolio: Assign random weights to assets
    weights = np.random.random(num_assets) # Random floats in the half-open interval
[0.0, 1.0): Long-only portfolio
    weights = weights/np.sum(weights)

    # Calculate portfolio returns and variance
    porfolio_ret, porfolio_var = portfolio(weights, mean_returns, cov_matrix)

    # Append the random portfolio into our simulation list
    p_weights.append(weights)
    p_ret.append(porfolio_ret)
    p_vol.append(porfolio_var)


# Store the simulation results into DataFrame
data = {'Returns':p_ret, 'Variances':p_vol}
for counter, symbol in enumerate(symbols):
    data[symbol+' weight'] = [w[counter] for w in p_weights]

portfolios = pd.DataFrame(data)

portfolios['Sharpe Ratio']=(portfolios['Returns']-rf)/np.sqrt(portfolios['Variances'])

portfolios
```

Out[57]:

	Returns	Variances	AAPL weight	AMZN weight	GOOGL weight	MSFT weight	Sharpe Ratio
0	0.289203	0.049129	0.290216	0.422830	0.013620	0.273335	1.259651
1	0.256413	0.043608	0.263461	0.240209	0.228710	0.267620	1.179994
2	0.245196	0.044346	0.084576	0.172209	0.403393	0.339821	1.116873
3	0.260270	0.043647	0.287198	0.228152	0.150799	0.333851	1.197927
4	0.254863	0.045065	0.243938	0.290316	0.319721	0.146025	1.153464
...	...	...	...	...	...	...	...
49995	0.260770	0.047249	0.275043	0.360426	0.308185	0.056346	1.153669
49996	0.236477	0.043348	0.103897	0.098397	0.414046	0.383659	1.087782
49997	0.224826	0.041875	0.280452	0.072030	0.417912	0.229605	1.049805
49998	0.252514	0.042908	0.310522	0.192859	0.192199	0.304420	1.170760
49999	0.301319	0.052275	0.126759	0.494064	0.048487	0.330690	1.274152

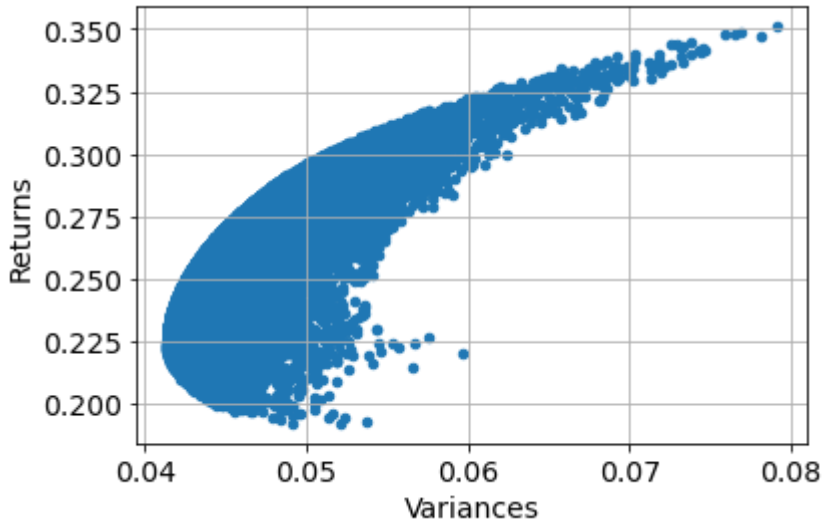
50000 rows × 7 columns

In [58]:

```
portfolios.plot.scatter(x='Variances', y='Returns', grid=True)
```

Out[58]:

```
<AxesSubplot:xlabel='Variances', ylabel='Returns'>
```



4.3.1.1 Global Minimum Variance Portfolio

In [59]:

```
portfolios[portfolios['Variances']==portfolios['Variances'].min()]
```

Out[59]:

	Returns	Variances	AAPL weight	AMZN weight	GOOGL weight	MSFT weight	Sharpe Ratio
2677	0.223526	0.041358	0.303129	0.007392	0.343711	0.345769	1.049952

In [60]:

```
min_var_port = portfolios.iloc[portfolios['Variances'].idxmin()]
min_var_port
```

Out[60]:

```
Returns         0.223526
Variances       0.041358
AAPL weight     0.303129
AMZN weight     0.007392
GOOGL weight    0.343711
MSFT weight     0.345769
Sharpe Ratio    1.049952
Name: 2677, dtype: float64
```

In [61]:

```
# Output Global-Minimum Variance portfolio
print("\nPortfolio with Minimum-Variance:\n")

print(f"Annual Sharpe Ratio: {round(min_var_port['Sharpe Ratio'],3)} | Annual Return:
% {round(min_var_port['Returns']*100,2)} | Annual Volatility: %
{round(min_var_port['Variances']*100,3)}\n")
for index,symbol in enumerate(symbols):
    print(f'{symbol}:\t% {round(min_var_port[index+2]*100,2)}')
```

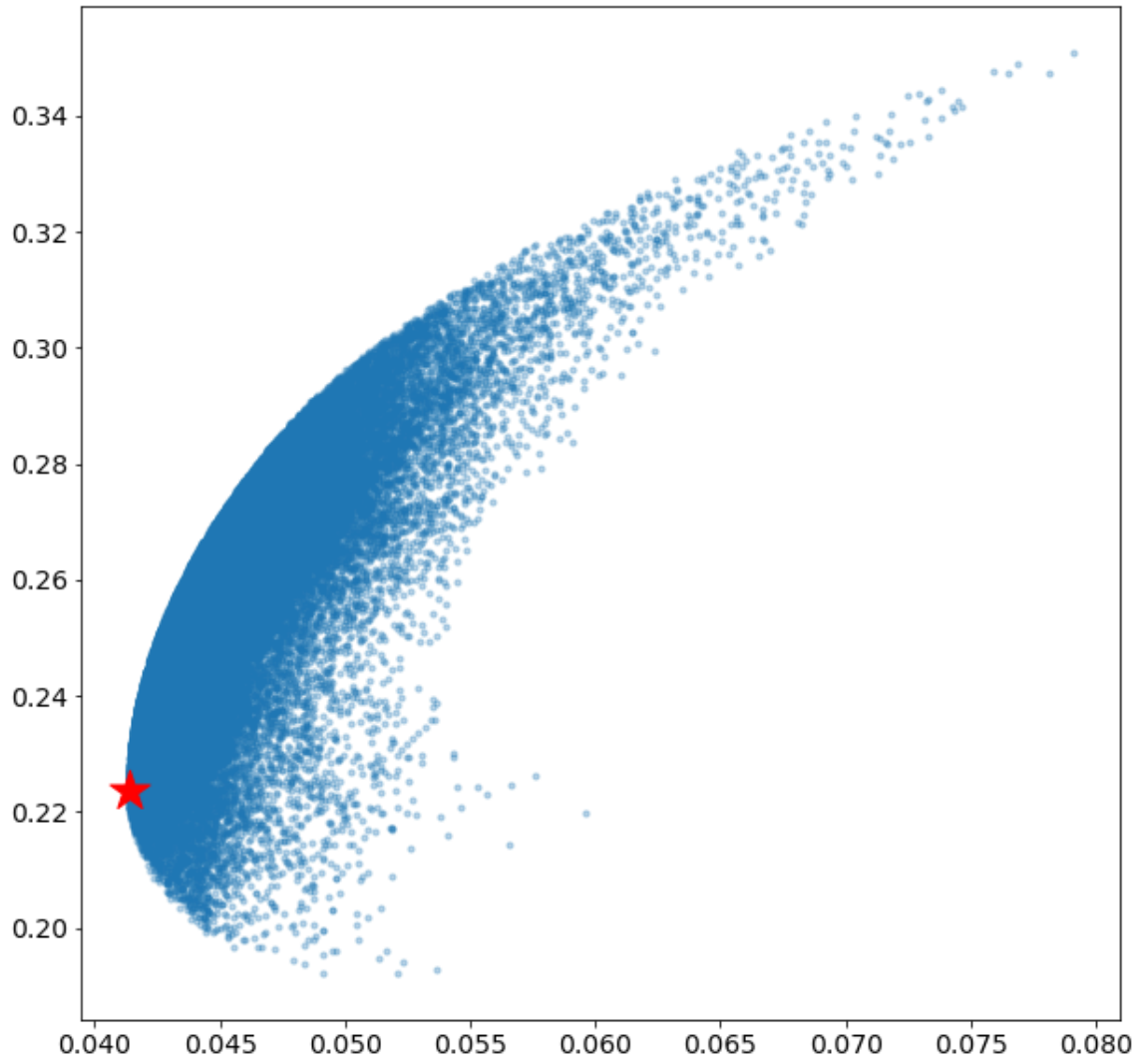
```
Portfolio with Minimum-Variance:

Annual Sharpe Ratio: 1.05 | Annual Return: % 22.35 | Annual Volatility: %
4.136

AAPL:   % 30.31
AMZN:   % 0.74
GOOGL:  % 34.37
MSFT:   % 34.58
```

In [62]:

```
# plotting the minimum volatility portfolio
plt.subplots(figsize=[10,10])
plt.scatter(portfolios['Variances'], portfolios['Returns'],marker='o', s=10,
alpha=0.3)
plt.scatter(min_var_port[1], min_var_port[0], color='r', marker='*', s=500)
```

Out[62]:

```
<matplotlib.collections.PathCollection at 0x1b255720cd0>
```



4.3.1.2 Optimal Risky Portfolio (Maximum-Sharpe Ratio (Tangency) portfolio)

In [63]:

```
# Finding the optimal portfolio

optimal_risky_port = portfolios.iloc[((portfolios['Returns']-
rf)/np.sqrt(portfolios['Variances'])).idxmax()]

optimal_risky_port
```

Out[63]:

```
Returns         0.307865
Variances       0.053733
AAPL weight     0.072716
AMZN weight     0.495079
GOOGL weight    0.001405
MSFT weight     0.430800
Sharpe Ratio    1.284993
Name: 11873, dtype: float64
```

In [64]:

```
# Output Maximum-Sharpe Ratio (Tangency) portfolio
print("\nPortfolio with Maximum-Sharpe Ratio:\n")

print(f"Annual Sharpe Ratio: {round(optimal_risky_port['Sharpe Ratio'],3)} | Annual
Return: % {round(optimal_risky_port['Returns']*100,2)} | Annual Volatility: %
{round(optimal_risky_port['Variances']*100,3)}\n")
for index,symbol in enumerate(symbols):
    print(f'{symbol}:\t% {round(optimal_risky_port[index+2]*100,2)}')
```

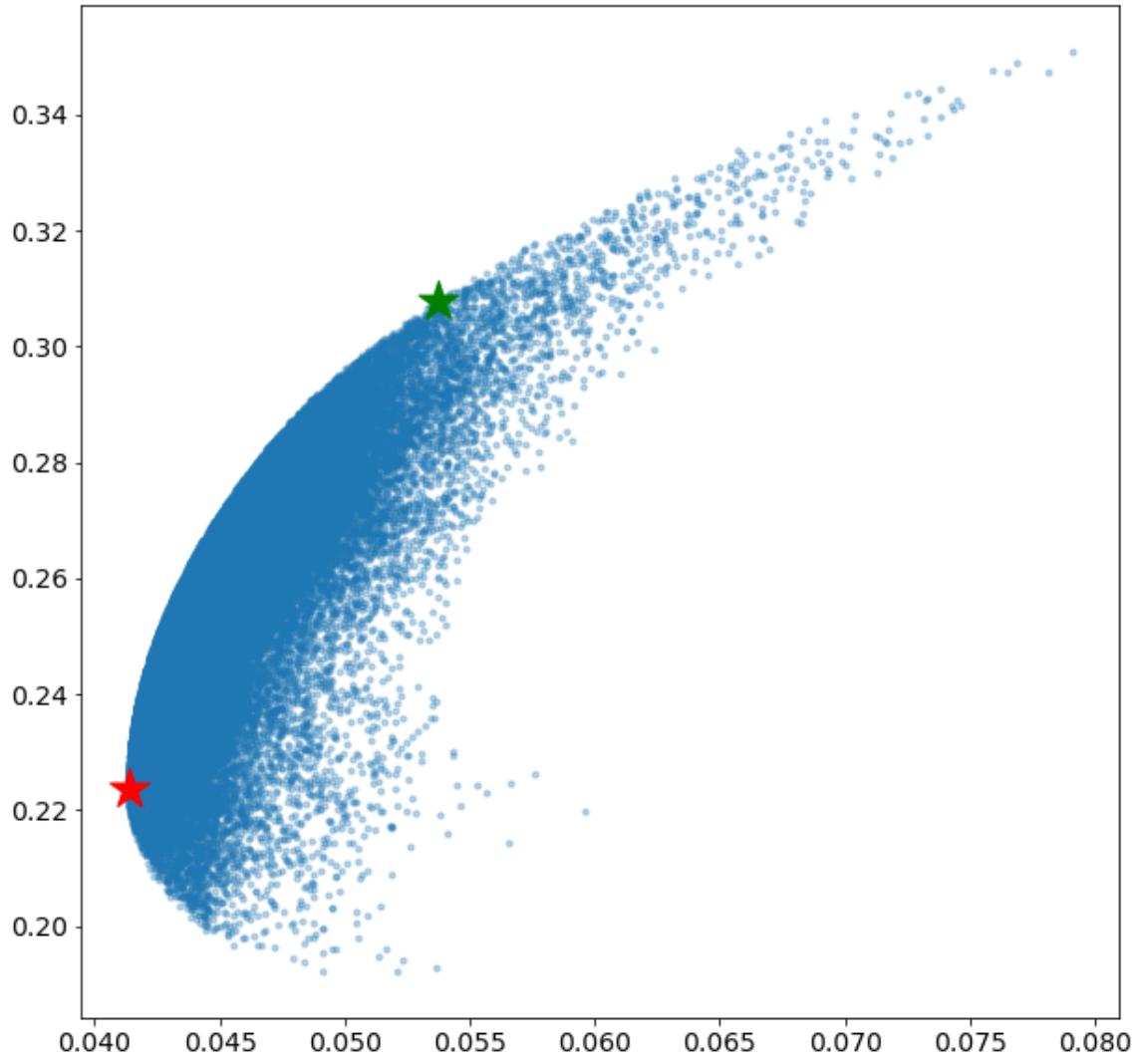
```
Portfolio with Maximum-Sharpe Ratio:

Annual Sharpe Ratio: 1.285 | Annual Return: % 30.79 | Annual Volatility: %
5.373

AAPL:   % 7.27
AMZN:   % 49.51
GOOGL:  % 0.14
MSFT:   % 43.08
```

In [65]:

```
# Plotting optimal portfolio
plt.subplots(figsize=(10, 10))
plt.scatter(portfolios['Variances'], portfolios['Returns'],marker='o', s=10,
alpha=0.3)
plt.scatter(min_var_port[1], min_var_port[0], color='r', marker='*', s=500)
plt.scatter(optimal_risky_port[1], optimal_risky_port[0], color='g', marker='*',
s=500)
```

Out[65]:

```
<matplotlib.collections.PathCollection at 0x1b252dc6670>
```



4.3.2 Scipy Optimize Module

In [66]:

```
import scipy.optimize as opt
```

4.3.2.1 Minimize Portfolio Variance

In [67]:

```
# Objective Function: Minimize portfolio variance
def portfolio_variance(weights, mean_returns, cov_matrix):
    portfolio_return, portfolio_var = portfolio(weights, mean_returns, cov_matrix)
    return(portfolio_var)


def minimize_portfolio_variance(mean_returns, cov_matrix, risk_free_rate=0, w_bounds=
(0,1)):
    "This function finds the portfolio weights which minimize the portfolio
volatility(variance)"

    init_guess = np.array([1/len(mean_returns) for i in range(len(mean_returns))])
    args = (mean_returns, cov_matrix)
    constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1})
    result = opt.minimize(fun=portfolio_variance,
                          x0=init_guess,
                          args=args,
                          method='SLSQP',
                          bounds=tuple(w_bounds for i in range(len(mean_returns))),
                          constraints=constraints,
                          )

    if result['success']:
        print(result['message'])
        min_var = result['fun']
        min_var_weights = result['x']
        min_var_return, min_var_variance= portfolio(min_var_weights, mean_returns,
cov_matrix)

        min_var_sharpe = ((min_var_return-risk_free_rate)/np.sqrt(min_var_variance))
        return(min_var_sharpe, min_var_weights, min_var_return, min_var_variance)
    else:
        print("Optimization operation was not succesfull!")
        print(result['message'])
        return(None)
```

In [68]:

```
# Call minimize function
min_var_sharpe, min_var_weights, min_var_return, min_var_variance =
minimize_portfolio_variance(mean_returns, cov_matrix,risk_free_rate=0.01, w_bounds=
(0,1))


# Output minimum volatility porfolio
print("\nPortfolio with Minimum-Variance:\n")

print(f"Annual Sharpe Ratio: {round(min_var_sharpe,3)} | Annual Return: %
{round(min_var_return*100,2)} | Annual Volatility: %
{round(min_var_variance*100,3)}\n")
for index,symbol in enumerate(symbols):
    print(f'{symbol}:\t% {round(min_var_weights[index]*100,2)}')
```

```
Optimization terminated successfully

Portfolio with Minimum-Variance:

Annual Sharpe Ratio: 1.049 | Annual Return: % 22.34 | Annual Volatility: %
4.135

AAPL:   % 31.05
AMZN:   % 0.95
GOOGL:  % 34.34
MSFT:   % 33.67
```

4.3.2.2 Maximize Portfolio Sharpe Ratio

minimize negative portfolio Sharpe ratio

In [69]:

```
# Objective Function: Maximize portfolio Sharpe Ratio
def neg_sharpe_ratio(weights, mean_returns, cov_matrix, risk_free_rate=0):
    portfolio_return, portfolio_var = portfolio(weights, mean_returns, cov_matrix)
    sr = ((portfolio_return - risk_free_rate)/np.sqrt(portfolio_var))
    return(-sr)


def optimize_sharpe_ratio(mean_returns, cov_matrix, risk_free_rate=0, w_bounds=(0,1)):
    "This function finds the portfolio weights which minimize the negative sharpe
ratio"

    init_guess = np.array([1/len(mean_returns) for i in range(len(mean_returns))])

    args = (mean_returns, cov_matrix, risk_free_rate)
    constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1})
    result = opt.minimize(fun=neg_sharpe_ratio,
                          x0=init_guess,
                          args=args,
                          method='SLSQP',
                          bounds=tuple(w_bounds for i in range(len(mean_returns)) ),
                          constraints=constraints,
                          )

    if result['success']:
        print(result['message'])
        opt_sharpe = - result['fun']
        opt_weights = result['x']
        opt_return, opt_variance = portfolio(opt_weights, mean_returns, cov_matrix)
        return(opt_sharpe, opt_weights, opt_return, opt_variance)
    else:
        print("Optimization operation was not succesfull!")
        print(result['message'])
        return(None)
```

In [70]:

```
# Call minimize function to get max sharpe portfolio
opt_sharpe, opt_weights, opt_return, opt_variance =
optimize_sharpe_ratio(mean_returns, cov_matrix,risk_free_rate=0.01, w_bounds=(0,1))

# Output maximum sharpe ratio porfolio
print("\nPortfolio with maximum sharpe ratio:\n")
print(f"Annual Sharpe Ratio: {round(opt_sharpe,3)} | Annual Return: %
{round(opt_return*100,3)} | Annual Volatility: % {round(opt_variance*100,3)}\n")

for index,symbol in enumerate(symbols):
    print(f'{symbol}:\t% {round(opt_weights[index]*100,3)}')
```

```
Optimization terminated successfully

Portfolio with maximum sharpe ratio:

Annual Sharpe Ratio: 1.286 | Annual Return: % 30.892 | Annual Volatility:
% 5.406

AAPL:   % 8.746
AMZN:   % 51.314
GOOGL:  % 0.0
MSFT:   % 39.94
```

5 This Week

- Review of Linear Algebra
- Linear Regressions
  - Statsmodels
  - Scikit-learn
- Portfolio Optimization
  - Monte-Carlo Simulation
  - Scipy Optimize Module

6 Next Week

- Python Project: Web Scraping
- Course Review
- Q&A