1 FINANCIAL DATA CORRELATIONS

In this chapter, we introduce the idea of financial data correlations. This is the fundamental relationship driving most financial investment and trading models. If every financial data series such as stock returns is probabilistically independent of another, then there are no systematic risk factors affecting all stocks – each stock moves by its own independent unique risk. The latter is not reflective of actual stock market and corporate sector linkages. Listed companies do share common risks such as when they are in the same industry that is affected by an industry change or when they are all affected by the cost of funds when market interest rates change. When stock returns co-move or have non-zero correlations, they can be combined in a portfolio in an optimal way to maximize the investor's welfare. For example, if an investor wants to keep the risk proxied by portfolio return volatility to the minimum, then the portfolio can be chosen by selecting weights on stocks forming the portfolio such that the portfolio has minimum return volatility (square root of variance). There are many other ways to optimize a portfolio (selecting optimal weights of composition stocks) depending on what is the investor's objective function. We will discuss these concepts and computations using machine learning approach in this chapter.

Before that we do a quick review of 2 of the most popular and widely used packages in Python – Pandas and Numpy. Both also work well with the plotting package Matplotlib. Numpy is a library for mathematical computations. Numpy looks at data mainly via arrays (Numpy arrays). A 1-dimensional (1D) array (or a flat array) of [1,2,3,4,5] can be created as "ar1=np.array([1,2,3,4,5])" after installing numpy. The open and close quotation marks " " are not part of the command statement. "np." is important so the code will call up numpy to execute the array command. It's dimension is 1 or a rank of 1. When ar1.shape is run, the output is (5,) indicating only one-dimension or a list of 5 numbers. If instead, "ar2=np.array([[1,2,3,4,5]])" is entered, the output ar2 is a 2D or rank 2 matrix with output (1,5) when ar2.shape is run. Note the second set of brackets makes the entry in () a list of one list, so it becomes 2D matrix with one row and 5 entries in that row. If instead, "ar3=np.array([[[1,2,3,4,5]]])" is entered, the output ar3 is a 3D or rank 3 matrix with output (1,1,5) when ar3.shape is run. Output of "np.array([[[1,2,3,4,5],[6,7,8,9,10]]]).shape" is (1,2,5). If "ar4 = np.array ([[[1,2,3,4,5],[6,7,8,9,10]]])", then ar4.shape[0] outputs the number of rows as

1, then ar4.shape[1] outputs the number of columns as 2, and output of ar4.shape[2] is 5. Output of "np.array ([[1], [2], [3], [4], [5]]).shape" is (5,1), and so on. Matrix multiplication or .dot can be used on numpy arrays with conformable dimensions.

Pandas is a library for data manipulation and is highly compatible with Numpy. It operates on tables such as in an Excel file. A numpy 2D or higher dimension array can be directly passed onto Pandas as a dataframe object, e.g., "dfar2=pd.DataFrame(ar2)" and printing the dataframe "Print(dfar2)" gives

```
"   0 1 2 3 4
 0 1 2 3 4 5  "
```

with an additional index variable 0 on the leftmost column and named columns 0, 1, 2 ,3 4, respectively on the first row. The leftmost index basically counts the number of rows but it starts at the number 0 and goes up to the length of the dataframe minus 1. The columns can also be renamed, e.g., "dfar2.columns=['new1','new2','new3','new4','new5']" will produce

```
"    new1 new2 new3 new4 new5
0    1       2      3      4      5  "
```

if printed. Some commands are Pandas-based, such as dfar2.info() and dfar2. describe() and work only with dataframe inputs and not with numpy arrays.

1.1 Portfolio Diversification

Markowitz efficient portfolio frontier represents the boundary of portfolios of risky stocks that that have the minimum return variance given the expected portfolio return above the minimum variance portfolio return. With a continuous efficient frontier, it is also the maximum expected return given corresponding variance or else volatility (square root of variance) of the portfolio. There is an exact analytical solution if there are no short-sale constraints, i.e., some stocks could have a negative portfolio weight. However, when there is a short-sale constraint, all stocks must be held in positive or zero quantities. With short-sale constraint and the capital constraint that all portfolio weights must sum to one (no idle money), the solution to obtain minimum portfolio return volatility, given expected portfolio return above the minimum variance portfolio return, is typically found via numerical optimization methods. In this chapter, we use the Scipy package/library in Python for such optimizations.

Let there be N stocks and a feasible portfolio is formed with weight w_i on stock i. It is feasible based on the constraints $\sum_{i=1}^{N} w_i = 1$, and $w_i \geq 0$ for every i. Let the weight vector be $w = (w_1, w_2, w_3, \ldots, w_N)^T$. Let the expected return[1] of stock i be $E(r_{it}^*)$ at time t. The expected return vector is $\mu = (\mu_1, \mu_2, \mu_3, \ldots, \mu_N)^T$ where $\mu_i = E(r_{it}^*)$. If r_{it}^* is stationary, then its unconditional expectation is constant for every t. The covariance matrix of $(r_{1t}^*, r_{2t}^*, r_{3t}^*, \ldots, r_{Nt}^*)^T$ for each t is given as $\Sigma_{N\times N}$. A minimum variance portfolio (portfolio with the minimum return variance) can be found as follows.

$$\min_{w} \; w^T \Sigma w \quad \text{subject to } \sum_{i=1}^{N} w_i = 1 \text{ and } w_i \geq 0 \text{ for every i} \qquad (1.1)$$

The superscript T refers where appropriate to transpose. The objective function under the solution w_{MVP} is the minimum return variance, ${w_{MVP}}^T \Sigma w_{MVP}$, associated with the minimum variance portfolio (MVP).

Another type of optimal portfolio is that of maximizing the Sharpe ratio defined as $w^T\mu / \sqrt{w^T \Sigma w}$ which is expected portfolio return per unit of risk or per unit of portfolio return standard deviation. This is solved as:

$$\min_{w} \sqrt{w^T \Sigma w} \, / \, w^T \mu$$

$$\text{subject to } \sum_{i=1}^{N} w_i = 1 \text{ and } w_i \geq 0 \text{ for every i} \qquad (1.2)$$

Finally, the minimum variance of a portfolio for a given expected return k greater or equal to the return variance of the minimum variance portfolio can be found by solving:

$$\min_{w} \; w^T \Sigma w$$

$$\text{subject to } \sum_{i=1}^{N} w_i = 1 \text{ and } w_i \geq 0 \text{ for every i, and } w^T\mu = k \qquad (1.3)$$

In model (1.3), maximum k for $\sum_{i=1}^{N} w_i = 1$ and $w_i \geq 0$ occurs when $\mu_m = \max(\mu_1, \mu_2, \mu_3, \ldots, \mu_N)$ and $w_m = 1$ while $w_{i\neq m} = 0$. Then max $k = \mu_m$. For any k, ${w_{MVP}}^T \mu \leq k \leq \mu_m$, we can find solution w(k) such that the efficient portfolio frontier occurs at expected return k and portfolio return volatility $\sqrt{w(k)^T \Sigma \, w(k)}$.

---

[1] * or asterisk to r_{it} denotes that the return is adjusted for dividends.

1.2 Worked Example – Data

Stock price data are obtained from public source Yahoo Finance. In this first part of the data work, 8 of the largest ecommerce listed companies, viz. Alibaba, Amazon, EBay, Rakuten, Suning, Wayfair, Zalando, and JD.Com are examined. The program merges the 8 individual stock price data sets in code line [25], and then computes their continuously compounded return rates in code line [28]. Portfolio optimizations based on the stock return series are carried out in code lines [36] to [38]. See demonstration file Chapter1-1portecom.ipynb.

```
In [1]: ### This module computes returns and mean-var frontier given each "expected" return based on averaged realized return
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
In [2]: ### We can import the data set. The dataframe name is now df
### By default, date columns are represented as object when loading data from a CSV file
### We can read in the Date column
df = pd.read_csv('###Alibaba_USD.csv', parse_dates=True)
### If True and parse_dates is enabled, pandas will attempt to infer the format of the datetime strings in the columns
### See date grouping codes in https://stackoverflow.com/questions/11391969/how-to-group-pandas-dataframe-entries-by-
### date-in-a-non-unique-column
### If reading a .txt file, other options about in-between spaces may need to be specified, e.g. ...header=None,
###    delimiter=r"\s+"...
### Columns can be renamed viz. ...pandas.read_csv('...',parse_dates=True,names=['col1','col2','col3','col4','col5','col6'])
### Note the parsed Date column cannot be renamed
```

```
In [3]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1965 entries, 0 to 1964
Data columns (total 7 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       1965 non-null   object
 1   Open       1965 non-null   float64
 2   High       1965 non-null   float64
 3   Low        1965 non-null   float64
 4   Close      1965 non-null   float64
 5   Adj Close  1965 non-null   float64
 6   Volume     1965 non-null   int64
dtypes: float64(5), int64(1), object(1)
memory usage: 107.6+ KB
```

```
In [4]: df.head()
```

Out[4]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	19/9/2014	92.699997	99.699997	89.949997	93.889999	93.889999	271879400
1	22/9/2014	92.699997	92.949997	89.500000	89.889999	89.889999	66657800
2	23/9/2014	88.940002	90.480003	86.620003	87.169998	87.169998	39009800
3	24/9/2014	88.470001	90.570000	87.220001	90.570000	90.570000	32088000
4	25/9/2014	91.089996	91.500000	88.500000	88.919998	88.919998	28598000

Code lines [3] and [4] show information about the data type in the various columns read into data frame df. Data type "float64" or 64 bit (double precision) floating point number which is a real number with a decimal point – the decimal point can "float" to a correct position. Floating point numbers can be represented in computer programs as a base 2 (binary) fraction that is convenient for computations. An integer can typically be converted to a floating number for computations. df.head() prints the first 5 lines of the data frame df.

```
In [5]: df[["day", "month", "year"]] = df["Date"].str.split("/", expand = True) ### Splits the Date entry into 3 separate columns
        print("\nNew DataFrame:")
        print(df)
        ### Note this split command does not work if ...parse_dates=['date']... is entered into the pd.read_csv(..)
```

```
New DataFrame:
            Date        Open        High         Low       Close   Adj Close  \
0      19/9/2014   92.699997   99.699997   89.949997   93.889999   93.889999
1      22/9/2014   92.699997   92.949997   89.500000   89.889999   89.889999
2      23/9/2014   88.940002   90.480003   86.620003   87.169998   87.169998
3      24/9/2014   88.470001   90.570000   87.220001   90.570000   90.570000
4      25/9/2014   91.089996   91.500000   88.500000   88.919998   88.919998
...          ...         ...         ...         ...         ...         ...
1960    5/7/2022  114.510002  120.529999  112.139999  120.129997  120.129997
1961    6/7/2022  118.930000  120.000000  115.510002  119.120003  119.120003
1962    7/7/2022  120.629997  125.000000  120.629997  122.389999  122.389999
1963    8/7/2022  122.260002  125.839996  120.699997  120.900002  120.900002
1964   11/7/2022  115.459999  115.580002  109.330002  109.570000  109.570000

         Volume day month  year
0     271879400  19     9  2014
1      66657800  22     9  2014
2      39009800  23     9  2014
3      32088000  24     9  2014
4      28598000  25     9  2014
...         ...  ..   ...   ...
1960   20989300   5     7  2022
1961   20222700   6     7  2022
1962   24202000   7     7  2022
1963   27201200   8     7  2022
1964   31249900  11     7  2022

[1965 rows x 10 columns]
```

In code line [5], the object of Date can sometimes be usefully separated into day, month, and year. These are then converted to floating numbers in code line [8] should there be a need to use them as numbers for some computations.

```
In [6]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1965 entries, 0 to 1964
Data columns (total 10 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       1965 non-null   object
 1   Open       1965 non-null   float64
 2   High       1965 non-null   float64
 3   Low        1965 non-null   float64
 4   Close      1965 non-null   float64
 5   Adj Close  1965 non-null   float64
 6   Volume     1965 non-null   int64
 7   day        1965 non-null   object
 8   month      1965 non-null   object
 9   year       1965 non-null   object
dtypes: float64(5), int64(1), object(4)
memory usage: 153.6+ KB
```

```
In [7]: df.head()
```

Out[7]:

	Date	Open	High	Low	Close	Adj Close	Volume	day	month	year
0	19/9/2014	92.699997	99.699997	89.949997	93.889999	93.889999	271879400	19	9	2014
1	22/9/2014	92.699997	92.949997	89.500000	89.889999	89.889999	66657800	22	9	2014
2	23/9/2014	88.940002	90.480003	86.620003	87.169998	87.169998	39009800	23	9	2014
3	24/9/2014	88.470001	90.570000	87.220001	90.570000	90.570000	32088000	24	9	2014
4	25/9/2014	91.089996	91.500000	88.500000	88.919998	88.919998	28598000	25	9	2014

```
In [8]: cols = df.select_dtypes(exclude=['float']).columns
        df[cols] = df[cols].apply(pd.to_numeric, downcast='float', errors='coerce')
        ### Above converts the Date, day. mnth, year from objects to float#
        ### errors = 'coerce' sets invalid parsing as NaN.
        df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1965 entries, 0 to 1964
Data columns (total 10 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       0 non-null      float32
 1   Open       1965 non-null   float64
 2   High       1965 non-null   float64
 3   Low        1965 non-null   float64
 4   Close      1965 non-null   float64
 5   Adj Close  1965 non-null   float64
 6   Volume     1965 non-null   float32
 7   day        1965 non-null   float32
 8   month      1965 non-null   float32
 9   year       1965 non-null   float32
dtypes: float32(5), float64(5)
memory usage: 115.3 KB
```

Code line [9] selects only rows in the data set df that have years (one of the features or columns) between 2017 and 2021 inclusive. This sub data set is now in a data frame called dfAlibaba. The original Date column is dropped. The data frame is also redefined to include only 4 columns of “Adj Close”, “day”, “month”, and “year”. Using “Adj Close” or adjusted closing price of the stock is preferable for computing daily return rates. The adjustment is for dividend issue and other stock distribution such as stock dividends and stock splits. This is explained as follows.

In U.S., listed or public firm typically pays dividends on a quarterly basis. Suppose on date Y, a firm announces that some dividends are planned to be paid (payment of cash) on date Y+20 days. There are two other important dates in-between, e.g., Y+10 is the ex-dividend date and Y+11 is the record date. Any investor J who buys the stock from investor I on or after the ex-dividend date of Y+10 will not be paid the announced dividends. Investor I who had purchased the share before ex-dividend date and had not sold by Y+10 will receive the dividends even if he/she sold by Y+20. As stock buy/sell transaction will typically take more than a day for settlement, a sell by I at Y+10 would not appear on the record date so the investor I still receives the dividend at Y+20 according to the record at Y+11. On the other hand, investor J who purchased the stock on Y+10 and later would not appear on the record date, so investor J would not receive the dividend at Y+20. The records for dividend distributions are updated with each new forthcoming dividend issue.

The price of the stock typically falls on the ex-dividend date by an amount equal to the dividend. Thus, it is important to consider adding the dividend so that the computed return rate would not appear to drop due to ex-dividend. There are two ways to compute cum-dividend return rates. The obvious one is to include expected future dividend payout D_{Y+20} on ex-dividend date, i.e., at Y+10, the return rate is $(P_{Y+10} + D_{Y+20})/P_{Y+9} - 1$. Or the continuously compounded return rate, that has the advantage of a support or potential range of $(-\infty, +\infty)$, i.e., $r_{Y+10} = \ln (P_{Y+10} + D_{Y+20})/P_{Y+9}$ could be computed. Note that the dividend is not exactly collected at Y+10. Another way is to adjust the time series of all daily closing prices as follows.

Subtract D_{Y+20} from the price the day prior to ex-dividend date Y+9. Call this the adjusted closing price at Y+9, $P_{Y+9}^* = P_{Y+9} - D_{Y+20}$. Then the continuously compounded return rate at Y+10 is computed as $r_{Y+10}^* = \ln P_{Y+10}/P_{Y+9}^*$. For daily return, this is approximately the same as r_{Y+10} . Moreover, all closing prices prior to Y+10 are adjusted by the ratio P_{Y+9}^*/ P_{Y+9} , i.e., $P_{Y+8}^* = P_{Y+8} \times (P_{Y+9}^*/ P_{Y+9})$, $P_{Y+7}^* = P_{Y+7} \times (P_{Y+9}^*/ P_{Y+9})$, and so on. So, return rate at Y+9 calculated using the adjusted closing prices is $\ln P_{Y+9}^*/P_{Y+8}^* = \ln P_{Y+9}/P_{Y+8}$. Earlier return rates are $r_{Y+8}^* = \ln P_{Y+8}^*/P_{Y+7}^* = \ln P_{Y+8}/P_{Y+7}$ and so on.

If a firm announces a 1:1 stock dividend or else a 2:1 stock split, the effect is the same – the investor gets an additional share for every share he/she owns. At effective date of distribution, the closing price of the share will typically drop to half. All prices prior would be adjusted to $P_t^*=P_t/2$.

```
In [9]: ### Select only rows where year >= 2017 and <=2021, i.e. 5 years
        dfAlibaba = df[(df['year']>=2017)&(df['year']<=2021)]
        dfAlibaba = dfAlibaba.drop('Date',axis=1) ### 1 is the axis number (0 for rows and 1 for columns)
        ### If we do not drop the Date column, Date will be shown as NaN
        ### Includes only rows where years are between 2017 and 2021 inclusive
        ### Next remove all columns except keeping Adj Close (closing price including dividends and split effects) and day, month, year
        dfAlibaba = dfAlibaba[['Adj Close','day','month','year']]
        print(dfAlibaba)

               Adj Close   day  month    year
        576    88.599998   3.0    1.0  2017.0
        577    90.510002   4.0    1.0  2017.0
        578    94.370003   5.0    1.0  2017.0
        579    93.889999   6.0    1.0  2017.0
        580    94.720001   9.0    1.0  2017.0
        ...          ...   ...    ...     ...
        1830  116.589996  27.0   12.0  2021.0
        1831  114.800003  28.0   12.0  2021.0
        1832  112.089996  29.0   12.0  2021.0
        1833  122.989998  30.0   12.0  2021.0
        1834  118.790001  31.0   12.0  2021.0

        [1259 rows x 4 columns]

In [10]: ### Next we process all the other 7 stocks in ECOMMERCE sector

In [11]: dfAmazon = pd.read_csv('###Amazon_USD.csv',parse_dates=True)
         dfAmazon[["day", "month", "year"]] = dfAmazon["Date"].str.split("/", expand = True) ### Splits the Date entry into 3 separate col
         cols = dfAmazon.select_dtypes(exclude=['float']).columns
         dfAmazon[cols] = dfAmazon[cols].apply(pd.to_numeric, downcast='float', errors='coerce')
         dfAmazon = dfAmazon[(dfAmazon['year']>=2017)&(dfAmazon['year']<=2021)]
         dfAmazon = dfAmazon.drop('Date',axis=1)
         dfAmazon = dfAmazon[['Adj Close','day','month','year']]
```

In [12]:
```
print(dfAmazon)
### Check to make sure this firm data also has same rows
```

```
       Adj Close   day  month    year
4941   37.683498   3.0    1.0  2017.0
4942   37.859001   4.0    1.0  2017.0
4943   39.022499   5.0    1.0  2017.0
4944   39.799500   6.0    1.0  2017.0
4945   39.846001   9.0    1.0  2017.0
...          ...   ...    ...     ...
6195  169.669495  27.0   12.0  2021.0
6196  170.660995  28.0   12.0  2021.0
6197  169.201004  29.0   12.0  2021.0
6198  168.644501  30.0   12.0  2021.0
6199  166.716995  31.0   12.0  2021.0

[1259 rows x 4 columns]
```

In [13]:
```
dfEbay = pd.read_csv('###Ebay_USD.csv',parse_dates=True)
dfEbay[["day", "month", "year"]] = dfEbay["Date"].str.split("/", expand = True) ### Splits the Date entry into 3 separate columns
cols = dfEbay.select_dtypes(exclude=['float']).columns
dfEbay[cols] = dfEbay[cols].apply(pd.to_numeric, downcast='float', errors='coerce')
dfEbay = dfEbay[(dfEbay['year']>=2017)&(dfEbay['year']<=2021)]
dfEbay = dfEbay.drop('Date',axis=1)
dfEbay = dfEbay[['Adj Close','day','month','year']]
```

In [14]:
```
print(dfEbay)
```

```
      Adj Close   day  month    year
4598  28.413393   3.0    1.0  2017.0
4599  28.337217   4.0    1.0  2017.0
4600  28.575266   5.0    1.0  2017.0
4601  29.565544   6.0    1.0  2017.0
4602  29.279884   9.0    1.0  2017.0
...         ...   ...    ...     ...
5852  65.094376  27.0   12.0  2021.0
5853  65.510750  28.0   12.0  2021.0
5854  65.887474  29.0   12.0  2021.0
5855  66.204720  30.0   12.0  2021.0
5856  65.927132  31.0   12.0  2021.0

[1259 rows x 4 columns]
```

In [15]:
```
dfRak= pd.read_csv('###Rakuten_USD.csv',parse_dates=True)
dfRak[["day", "month", "year"]] = dfRak["Date"].str.split("/", expand = True) ### Splits the Date entry into 3 separate columns
cols = dfRak.select_dtypes(exclude=['float']).columns
dfRak[cols] = dfRak[cols].apply(pd.to_numeric, downcast='float', errors='coerce')
dfRak= dfRak[(dfRak['year']>=2017)&(dfRak['year']<=2021)]
dfRak = dfRak.drop('Date',axis=1)
dfRak = dfRak[['Adj Close','day','month','year']]
```

In [16]:
```
print(dfRak)
```

```
      Adj Close   day  month    year
740       9.825   3.0    1.0  2017.0
741      10.100   4.0    1.0  2017.0
742      10.250   5.0    1.0  2017.0
743      10.520   6.0    1.0  2017.0
744      10.590   9.0    1.0  2017.0
...         ...   ...    ...     ...
1994      9.905  27.0   12.0  2021.0
1995      9.980  28.0   12.0  2021.0
1996     10.130  29.0   12.0  2021.0
1997     10.000  30.0   12.0  2021.0
1998     10.000  31.0   12.0  2021.0

[1259 rows x 4 columns]
```

In [17]:
```
dfSun= pd.read_csv('###Suning_CNY.csv',parse_dates=True)
dfSun[["day", "month", "year"]] = dfSun["Date"].str.split("/", expand = True) ### Splits the Date entry into 3 separate columns
cols = dfSun.select_dtypes(exclude=['float']).columns
dfSun[cols] = dfSun[cols].apply(pd.to_numeric, downcast='float', errors='coerce')
dfSun= dfSun[(dfSun['year']>=2017)&(dfSun['year']<=2021)]
dfSun = dfSun.drop('Date',axis=1)
dfSun = dfSun[['Adj Close','day','month','year']]
```

In [18]:
```
print(dfSun)
```

```
      Adj Close   day  month    year
3054  11.436833   3.0    1.0  2017.0
3055  11.446534   4.0    1.0  2017.0
3056  11.407731   5.0    1.0  2017.0
3057  11.233124   6.0    1.0  2017.0
3058  11.213723   9.0    1.0  2017.0
...         ...   ...    ...     ...
4264   4.140000  27.0   12.0  2021.0
4265   4.130000  28.0   12.0  2021.0
4266   4.080000  29.0   12.0  2021.0
4267   4.110000  30.0   12.0  2021.0
4268   4.120000  31.0   12.0  2021.0

[1215 rows x 4 columns]
```

In [19]:
```
dfWay= pd.read_csv('###Wayfair_USD.csv',parse_dates=True)
dfWay[["day", "month", "year"]] = dfWay["Date"].str.split("/", expand = True) ### Splits the Date entry into 3 separate columns
cols = dfWay.select_dtypes(exclude=['float']).columns
dfWay[cols] = dfWay[cols].apply(pd.to_numeric, downcast='float', errors='coerce')
dfWay= dfWay[(dfWay['year']>=2017)&(dfWay['year']<=2021)]
dfWay = dfWay.drop('Date',axis=1)
dfWay = dfWay[['Adj Close','day','month','year']]
```

In [20]:
```
print(dfWay)
```

```
       Adj Close   day  month    year
567    35.180000   3.0    1.0  2017.0
568    36.480000   4.0    1.0  2017.0
569    37.169998   5.0    1.0  2017.0
570    37.360001   6.0    1.0  2017.0
571    38.049999   9.0    1.0  2017.0
...          ...   ...    ...     ...
1821  198.880005  27.0   12.0  2021.0
1822  192.860001  28.0   12.0  2021.0
1823  191.720001  29.0   12.0  2021.0
1824  192.809998  30.0   12.0  2021.0
1825  189.970001  31.0   12.0  2021.0

[1259 rows x 4 columns]
```

In [21]:
```
dfZal= pd.read_csv('###Zalando_EUR.csv',parse_dates=True)
dfZal[["day", "month", "year"]] = dfZal["Date"].str.split("/", expand = True) ### Splits the Date entry into 3 separate columns
cols = dfZal.select_dtypes(exclude=['float']).columns
dfZal[cols] = dfZal[cols].apply(pd.to_numeric, downcast='float', errors='coerce')
dfZal= dfZal[(dfZal['year']>=2017)&(dfZal['year']<=2021)]
dfZal = dfZal.drop('Date',axis=1)
dfZal = dfZal[['Adj Close','day','month','year']]
```

In [22]:
```
print(dfZal)
```

```
      Adj Close   day  month    year
570   36.904999   2.0    1.0  2017.0
571   37.314999   3.0    1.0  2017.0
572   35.919998   4.0    1.0  2017.0
573   36.764999   5.0    1.0  2017.0
574   37.349998   6.0    1.0  2017.0
...         ...   ...    ...     ...
1832  70.519997  23.0   12.0  2021.0
1833  70.459999  27.0   12.0  2021.0
1834  70.800003  28.0   12.0  2021.0
1835  70.699997  29.0   12.0  2021.0
1836  71.139999  30.0   12.0  2021.0

[1267 rows x 4 columns]
```

In [23]:
```
dfJD= pd.read_csv('###Jingdong_USD.csv',parse_dates=True)
dfJD[["day", "month", "year"]] = dfJD["Date"].str.split("/", expand = True) ### Splits the Date entry into 3 separate columns
cols = dfJD.select_dtypes(exclude=['float']).columns
dfJD[cols] = dfJD[cols].apply(pd.to_numeric, downcast='float', errors='coerce')
dfJD= dfJD[(dfJD['year']>=2017)&(dfJD['year']<=2021)]
dfJD = dfJD.drop('Date',axis=1)
dfJD = dfJD[['Adj Close','day','month','year']]
```

In [24]:
```
print(dfJD)
```

```
      Adj Close   day  month    year
659   25.184586   3.0    1.0  2017.0
660   25.213848   4.0    1.0  2017.0
661   25.652773   5.0    1.0  2017.0
662   25.623512   6.0    1.0  2017.0
663   25.613758   9.0    1.0  2017.0
...         ...   ...    ...     ...
1913  66.043701  27.0   12.0  2021.0
1914  64.248985  28.0   12.0  2021.0
1915  64.014885  29.0   12.0  2021.0
1916  68.667503  30.0   12.0  2021.0
1917  68.345619  31.0   12.0  2021.0

[1259 rows x 4 columns]
```

In code lines [25] to [27], the adjusted closing price data sets of the 8 stocks are merged so that only all stock prices with the same day, month, and year are included. A few missing day gaps are ignored.

```
In [25]: ### merging all 8 data files by same day, month, year -- those that do not overlap are eliminated
dfmerge = pd.merge(dfAlibaba,dfAmazon,on = ['day', 'month','year'])
dfmerge.columns=['AlibabaP','day','month','year','AmazonP'] ### Rename the columns
dfmerge = pd.merge(dfmerge,dfEbay,on = ['day', 'month','year'])
dfmerge.columns=['AlibabaP','day','month','year','AmazonP','EbayP']
dfmerge = pd.merge(dfmerge,dfRak,on = ['day', 'month','year'])
dfmerge.columns=['AlibabaP','day','month','year','AmazonP','EbayP','RakutenP']
dfmerge = pd.merge(dfmerge,dfSun,on = ['day', 'month','year'])
dfmerge.columns=['AlibabaP','day','month','year','AmazonP','EbayP','RakutenP','SuningP']
dfmerge = pd.merge(dfmerge,dfWay,on = ['day', 'month','year'])
dfmerge.columns=['AlibabaP','day','month','year','AmazonP','EbayP','RakutenP','SuningP','WayfairP']
dfmerge = pd.merge(dfmerge,dfZal,on = ['day', 'month','year'])
dfmerge.columns=['AlibabaP','day','month','year','AmazonP','EbayP','RakutenP','SuningP','WayfairP','ZalandoP']
dfmerge = pd.merge(dfmerge,dfJD,on = ['day', 'month','year'])
dfmerge.columns=['AlibabaP','day','month','year','AmazonP','EbayP','RakutenP','SuningP','WayfairP','ZalandoP','JDcomP']
```

```
In [26]: print(dfmerge)
```

```
        AlibabaP   day  month    year     AmazonP      EbayP  RakutenP  \
0      88.599998   3.0    1.0  2017.0   37.683498  28.413393     9.825
1      90.510002   4.0    1.0  2017.0   37.859001  28.337217    10.100
2      94.370003   5.0    1.0  2017.0   39.022499  28.575266    10.250
3      93.889999   6.0    1.0  2017.0   39.799500  29.565544    10.520
4      94.720001   9.0    1.0  2017.0   39.846001  29.279884    10.590
...          ...   ...    ...     ...         ...        ...       ...
1156  118.660004  23.0   12.0  2021.0  171.068497  64.331001     9.770
1157  116.589996  27.0   12.0  2021.0  169.669495  65.094376     9.905
1158  114.800003  28.0   12.0  2021.0  170.660995  65.510750     9.980
1159  112.089996  29.0   12.0  2021.0  169.201004  65.887474    10.130
1160  122.989998  30.0   12.0  2021.0  168.644501  66.204720    10.000

        SuningP    WayfairP   ZalandoP     JDcomP
0     11.436833   35.180000  37.314999  25.184586
1     11.446534   36.480000  35.919998  25.213848
2     11.407731   37.169998  36.764999  25.652773
3     11.233124   37.360001  37.349998  25.623512
4     11.213723   38.049999  37.404999  25.613758
...         ...         ...        ...        ...
1156   4.110000  204.369995  70.519997  66.960564
1157   4.140000  198.880005  70.459999  66.043701
1158   4.130000  192.860001  70.800003  64.248985
1159   4.080000  191.720001  70.699997  64.014885
1160   4.110000  192.809998  71.139999  68.667503

[1161 rows x 11 columns]
```

```
In [27]: ### Now delete columns day, month, year; but before that keep day, month, year separately for later concatenation
dy=dfmerge["day"]
mth=dfmerge["month"]
yr=dfmerge["year"]
del(dfmerge['day'],dfmerge['month'],dfmerge['year'])
print(dfmerge)
```

```
        AlibabaP     AmazonP      EbayP  RakutenP    SuningP    WayfairP  \
0      88.599998   37.683498  28.413393     9.825  11.436833   35.180000
1      90.510002   37.859001  28.337217    10.100  11.446534   36.480000
2      94.370003   39.022499  28.575266    10.250  11.407731   37.169998
3      93.889999   39.799500  29.565544    10.520  11.233124   37.360001
4      94.720001   39.846001  29.279884    10.590  11.213723   38.049999
...          ...         ...        ...       ...        ...         ...
1156  118.660004  171.068497  64.331001     9.770   4.110000  204.369995
1157  116.589996  169.669495  65.094376     9.905   4.140000  198.880005
1158  114.800003  170.660995  65.510750     9.980   4.130000  192.860001
1159  112.089996  169.201004  65.887474    10.130   4.080000  191.720001
1160  122.989998  168.644501  66.204720    10.000   4.110000  192.809998

       ZalandoP     JDcomP
0     37.314999  25.184586
1     35.919998  25.213848
2     36.764999  25.652773
3     37.349998  25.623512
4     37.404999  25.613758
...         ...        ...
1156  70.519997  66.960564
1157  70.459999  66.043701
1158  70.800003  64.248985
1159  70.699997  64.014885
1160  71.139999  68.667503

[1161 rows x 8 columns]
```

Code line [28] computes the continuously compounded return rate that can also be expressed as ln $[1+ (P_{t+1}^*/P_t^* - 1)]$.

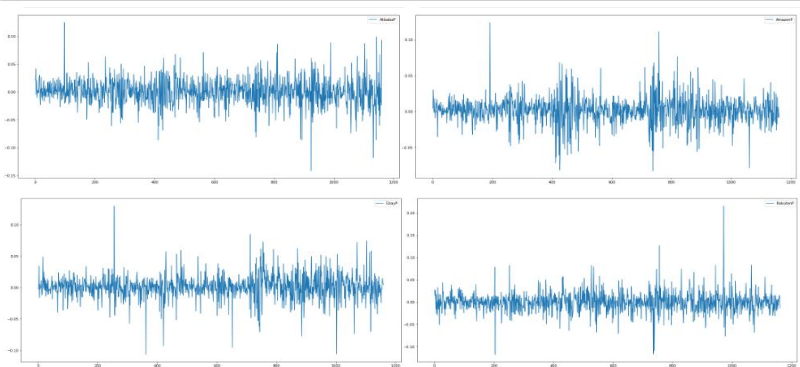
In [28]:
```
dfret=dfmerge.pct_change().apply(lambda x: np.log(1+x))
### Lambda is an alternative way of defining function inline using a single line of python code. percent change = x.
### Pandas -- Computes the percentage change from the immediately previous row by default
### Note the time series is in ascending order of time (ie later rows are later in time)
dfret=dfret.iloc[1:len(dfret.index),:] ### Remove NaN values at the first row. Note iloc[0:..,:] indicates first row
dfret.columns=['AlibabaP','AmazonP','EbayP','RakutenP','SuningP','WayfairP','ZalandoP','JDcomP'] ### Rename the columns
```

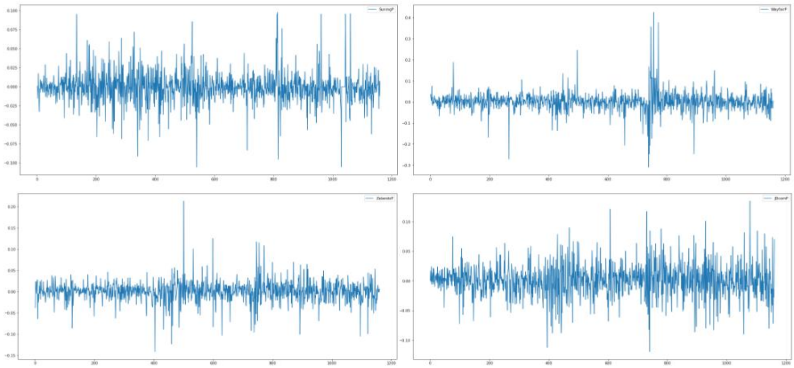
The time series of the 8 columns of stock returns are shown in code line [29]. Code line [30] uses seaborn package to print the time series graphs of the individual stock returns in 2 columns and 4 rows. Code lines [31] and [32] show the covariance and the correlation matrices of the return rates.

In [29]:
```
print(dfret)
```

```
      AlibabaP   AmazonP     EbayP  RakutenP   SuningP  WayfairP  ZalandoP  \
1     0.021329  0.004646 -0.002685  0.027605  0.000848  0.036286 -0.038101
2     0.041763  0.030270  0.008365  0.014742 -0.003396  0.018738  0.023252
3    -0.005099  0.019716  0.034068  0.026001 -0.015424  0.005099  0.015787
4     0.008801  0.001168 -0.009709  0.006632 -0.001729  0.018300  0.001472
5     0.021205 -0.001281 -0.016394 -0.016183  0.017153  0.075646 -0.004421
...        ...       ...       ...       ...       ...       ...       ...
1156  0.007189  0.000184  0.014748  0.002049 -0.028779  0.002842  0.007973
1157 -0.017599 -0.008212  0.011797  0.013723  0.007273 -0.027230 -0.000851
1158 -0.015472  0.005827  0.006376  0.007543 -0.002418 -0.030737  0.004814
1159 -0.023889 -0.008592  0.005734  0.014918 -0.012180 -0.005929 -0.001414
1160  0.092801 -0.003294  0.004803 -0.012916  0.007326  0.005669  0.006204

        JDcomP
1     0.001161
2     0.017258
3    -0.001141
4    -0.000381
5     0.024079
...        ...
1156 -0.071660
1157 -0.013787
1158 -0.027551
1159 -0.003650
1160  0.070160

[1160 rows x 8 columns]
```

In [30]:
```
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import stats

fig, axs = plt.subplots(ncols=2, nrows=4, figsize=(30, 30))
index = 0
axs = axs.flatten()
for k in dfret.items():
    sns.lineplot(data = k,ax=axs[index])
    index += 1
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=5.0)
```





```
In [31]: # Finding the simple covariance matrix from a series of returns
         covar = dfret.cov()
         print(covar)
```

```
          AlibabaP   AmazonP     EbayP  RakutenP   SuningP  WayfairP  \
AlibabaP  0.000568  0.000214  0.000141  0.000105  0.000073  0.000308
AmazonP   0.000214  0.000366  0.000133  0.000080  0.000027  0.000307
EbayP     0.000141  0.000133  0.000378  0.000080  0.000028  0.000289
RakutenP  0.000105  0.000080  0.000080  0.000520  0.000043  0.000202
SuningP   0.000073  0.000027  0.000028  0.000043  0.000423  0.000103
WayfairP  0.000308  0.000307  0.000289  0.000202  0.000103  0.002131
ZalandoP  0.000120  0.000115  0.000113  0.000091  0.000055  0.000335
JDcomP    0.000430  0.000243  0.000164  0.000116  0.000064  0.000355

          ZalandoP    JDcomP
AlibabaP  0.000120  0.000430
AmazonP   0.000115  0.000243
EbayP     0.000113  0.000164
RakutenP  0.000091  0.000116
SuningP   0.000055  0.000064
WayfairP  0.000335  0.000355
ZalandoP  0.000610  0.000169
JDcomP    0.000169  0.000767
```

```
In [32]: # Finding the simple correlation matrix from a series of returns
         corr_matrix = dfret.corr()
         print(corr_matrix)
```

```
          AlibabaP   AmazonP     EbayP  RakutenP   SuningP  WayfairP  \
AlibabaP  1.000000  0.469633  0.304793  0.192598  0.148897  0.280226
AmazonP   0.469633  1.000000  0.356632  0.183431  0.067568  0.348132
EbayP     0.304793  0.356632  1.000000  0.181259  0.070084  0.322172
RakutenP  0.192598  0.183431  0.181259  1.000000  0.091049  0.191544
SuningP   0.148897  0.067568  0.070084  0.091049  1.000000  0.108223
WayfairP  0.280226  0.348132  0.322172  0.191544  0.108223  1.000000
ZalandoP  0.203696  0.242445  0.236042  0.161165  0.108273  0.293582
JDcomP    0.651556  0.458592  0.304356  0.184363  0.113146  0.277591

          ZalandoP    JDcomP
AlibabaP  0.203696  0.651556
AmazonP   0.242445  0.458592
EbayP     0.236042  0.304356
RakutenP  0.161165  0.184363
SuningP   0.108273  0.113146
WayfairP  0.293582  0.277591
ZalandoP  1.000000  0.247294
JDcomP    0.247294  1.000000
```

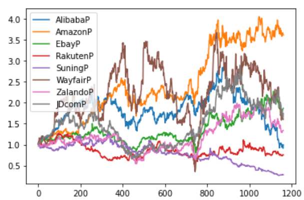
Sometimes we may need to check the data to ensure that there are no values that are infinite that could then lead to computational problems in what follows. In code line [33] we do this check. We could also check for how many missing values using np.isnull(dfret).values.sum().

```
In [33]: count = np.isinf(dfret).values.sum() ### Checking for infinite values using isinf() and displaying the count
         print(count)

         0
```

The cumulative return time series of the stocks from 2017 till 2021are shown in code line [34] and plotted in [35].

```
In [34]: ### Calculating daily cumulative return over 5 years
         dfcumuret=(dfret+1).cumprod()
         ### cumprod() returns a series of the same length as the original input series, containing the cumulative product
         print(dfcumuret)

               AlibabaP   AmazonP     EbayP  RakutenP   SuningP  WayfairP  ZalandoP  \
         1     1.021329  1.004646  0.997315  1.027605  1.000848  1.036286  0.961899
         2     1.063982  1.035057  1.005658  1.042755  0.997449  1.055704  0.984265
         3     1.058557  1.055464  1.039919  1.069867  0.982064  1.061087  0.999803
         4     1.067873  1.056696  1.029823  1.076962  0.980367  1.080505  1.001274
         5     1.090518  1.055343  1.012940  1.059533  0.997183  1.162241  0.996848
         ...        ...       ...       ...       ...       ...       ...       ...
         1156  0.967080  3.671289  1.816915  0.737815  0.281117  1.703261  1.327109
         1157  0.950061  3.641141  1.838348  0.747940  0.283162  1.656880  1.325980
         1158  0.935361  3.662357  1.850070  0.753582  0.282477  1.605952  1.332363
         1159  0.913016  3.630891  1.860678  0.764824  0.279036  1.596431  1.330479
         1160  0.997745  3.618929  1.869616  0.754945  0.281080  1.605482  1.338734


                 JDcomP
         1     1.001161
         2     1.018440
         3     1.017277
         4     1.016890
         5     1.041376
         ...        ...
         1156  1.708827
         1157  1.685267
         1158  1.638836
         1159  1.632854
         1160  1.747416

         [1160 rows x 8 columns]
```

```
In [35]: fig = plt.figure()
         (dfret + 1).cumprod().plot()
         plt.show()

         <Figure size 432x288 with 0 Axes>
```



In the following [36], the optimal weights on the 8 stocks are found to form the optimal portfolio. The weights are solutions to a minimization problem whereby the objective function is the annualized portfolio return standard error. This follows model (1.1) in subsection 1.1. Throughout, the key measurements of mean return and return standard deviation are in annualized terms. These are computed as follows.

For a particular stock i , its annualized return is $252 \times r_{it}^*$ where r_{it}^* is the daily return rate and is assumed to be unconditionally stationary (having same mean and variance for each time). Thus, the mean annual return is $252 \times E(r_{it}^*)$

where E(.) is the expectation operator, and $E(r_{it}^*)$ is estimated using $\frac{1}{T}\sum_{t=1}^{T} r_{it}^*$ with daily returns r_{it}^* , t=1,2,…..,T. The mean annualized return is estimated as 252 $\frac{1}{T}\sum_{t=1}^{T} r_{it}^*$ where 252 is approximately the number of trading days in U.S. For all stocks, the vector of mean annualized return is r = np.mean(dfret,axis=0)*252. Thus, r is μ in (1.1). A portfolio with weights vector w would have portfolio mean annualized return as $w^T r$.

For stocks i=1,2,…,N, with return vector r_t^* at day t, the variance of the daily portfolio return is $var(w^T r_t^*)$. Variance of portfolio annual return is var($\sum_{t=1}^{252} w^T r_t^*$) that is approximately $\sum_{t=1}^{252} \mathrm{var}(w^T r_t^*)$ or $252 \times var(w^T r_t^*)$ when intertemporal correlations are assumed to be approximately zero. $var(w^T r_t^*)$ is estimated using $w^T var(r_t^*) w$ for a given w. Each ijth element of $var(r_t^*)$ or covar is estimated as

$$\frac{1}{T-k}\sum_{t=1}^{T}(r_{it}^* - \mu_i)(r_{jt}^* - \mu_j)$$

where k =1 for i=j, and k=2 for i≠j. Annualized portfolio return volatility is $\sqrt{252 \times (w^T \mathrm{var}(r_t^*)\, w)}$ or np.sqrt(np.dot(w,np.dot(w,covar))*252).

In [36]:
```
### Ref: https://www.kaggle.com/code/trangthvu/efficient-frontier-optimization/notebook
import scipy
### All weights, of course, must be between 0 and 1. Thus we set 0 and 1 as the boundaries.
from scipy.optimize import Bounds
bounds = Bounds(0, 1)

### The second boundary is the sum of weights.
from scipy.optimize import LinearConstraint
linear_constraint = LinearConstraint(np.ones((dfret.shape[1],), dtype=int),1,1)
  ### 1,1 in argument of LinearConstraint refers to Lb, Up in Lb <= A.dot(w) <= Ub; if Lb=Ub, it implies equality constraint
  ### df.shape[0] refers to no. rows, .shape[1] refers to no. cols; np.ones fill up with ones
  ### Above -- np.ones((dfret.shape[1],), dtype=int) is A, i.e. 1 x 8 elements of ones since dfret.shape[1] gives dim of cols
  ### Then A'w = 1 is the constraint, i.e. sum of wts must equal to one

covar = dfret.cov()
r = np.mean(dfret,axis=0)*252
### axis=0 means to apply calculation “column-wise”, axis=1 means to:apply calculation “row-wise”,
### r is annualized vector mean return
### Here. mean is calculated for each XYZ stock return time series (column)

def ret(r,w):
    return r.dot(w)                                 ### Note ret(r,w) is defined here. r.dot(w) is matrix multiplication of r and w
def vol(w,covar):                                   ### Risk level or volatility
    return np.sqrt(np.dot(w,np.dot(w,covar))*252)### same as sqrt of w^T \Sigma w *252 -- annualized return volatility
def sharpe (ret,vol):
    return ret/vol
```

```
### Find a portfolio with the minimum risk.
from scipy.optimize import minimize
### Create x0, the first guess at the values of each stock's weight.
weights = np.ones(dfret.shape[1])
x0 = weights/np.sum(weights)

### Define a function to calculate volatility
portfstderr = lambda w: np.sqrt(np.dot(w,np.dot(w,covar))*252)  ### w is input to function lambda that outputs portfvola
res1 = minimize(portfstderr,x0,method='trust-constr',constraints = linear_constraint,bounds = bounds)
   ### constraint means unit vector .dot(w) = 1; minmize chooses wts w

### Objective function is portfstderr
### 'trust-constr' is to minimize a scalar function subject to constraints -- algorithm updates x0 till obj fn portfvola is min
### minimize(..) function returns optimal weight w_min
### These are the weights of the stocks in the portfolio with the lowest level of risk possible.
w_min = res1.x
### optimization full output.x gives the solution array
```

```
np.set_printoptions(suppress = True, precision=3)
### Suppress=True means always printing floating point numbers using fixed point notation, for nos. close to zero, i.e. 0.

print(w_min)
print('return: % .4f'% (ret(r,w_min)), 'risk: % .4f'% vol(w_min,covar))  ### this is min var portfolio
### "print" treats the % as a special character you need to add, so it can know, that when you type "f"
###   the number (result) that will be printed will be a floating point type, and the ".4" tells your "print"
###   to print only the first 4 digits after the point.
```

```
[0.036 0.21  0.209 0.165 0.278 0.    0.102 0.   ]
return:  0.0622 risk:  0.1908
```

For the minimum variance portfolio solution in [36], the optimal weights indicate 3.6% of total investment wealth allocated to Alibaba, 21% to Amazon, 20.9% to Ebay, 16.5% to Rakuten, 27.8% to Suning, and 10.2% to Zalando. Close to zero % are put into Wayfair and JD.Com. The weights are all positive and sum to one. In [37], the objective function to be minimized is the inverse of the Sharpe ratio – the solution is the same as maximizing Sharpe ratio. This follows model (1.2) in subsection 1.1.

```
In [37]: ### Define 1/Sharpe_ratio as invSharpe
invSharpe  = lambda w: np.sqrt(np.dot(w,np.dot(w,covar))*252)/r.dot(w)
res2 = minimize(invSharpe,x0,method='trust-constr',constraints = linear_constraint,bounds = bounds)
### Objective function is invSharpe -- inverse of Sharpe ratio
#These are the weights of the stocks in the portfolio with the highest Sharpe ratio - call the weight vector w_Sharpe

w_Sharpe = res2.x
   ### constraint means unit vector .dot(w) = 1; minmize chooses wts w
   ### optimization full output.x gives the solution array of optimal weights in min inverse Sharpe ratio or max Sharpe ratio
```

```
print(w_Sharpe)
print('return: % .4f'% (ret(r,w_Sharpe)), 'risk: % .4f'% vol(w_Sharpe,covar))  ### this is max Sharpe ratio portfolio
### "print" treats the % as a special character you need to add, so it can know, that when you type "f"
###   the number (result) that will be printed will be a floating point type, and the ".4" tells your "print"
###   to print only the first 4 digits after the point.

print( 1/(  np.sqrt(np.dot(w_Sharpe,np.dot(w_Sharpe,covar))*252)/r.dot(w_Sharpe) ) )
### Above is optimized objective function -- the max Sharpe ratio.
### It can also be found using print(sharpe(ret(r,w_Sharpe),vol(w_Sharpe,covar)))
```

```
[0.    0.768 0.171 0.    0.    0.03  0.031 0.   ]
return:  0.2969 risk:  0.2697
1.1006526861512465
```

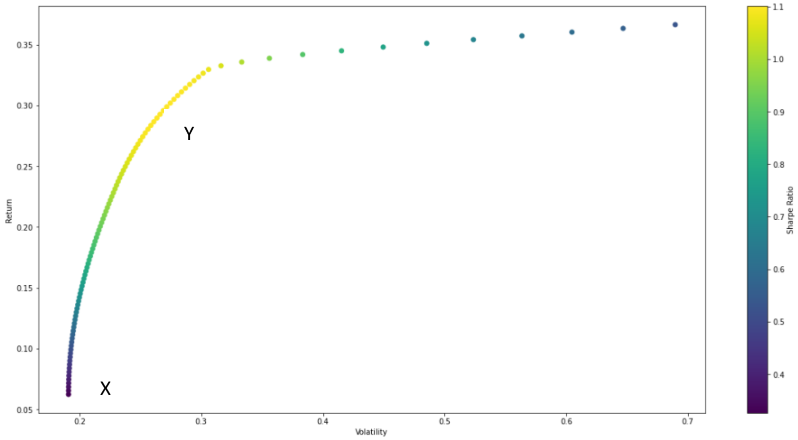
The solution to [37] shows the maximized Sharpe ratio is 1.1007 whereas the portfolio return, and standard error are respectively 29.69% and 26.97%. This latter result appears to be superior to that of minimum variance portfolio in (1.1) with a portfolio return of a much lower 6.22% and volatility of 19.08%.

In code line [38], the Markowitz mean-variance efficient portfolio frontier is drawn under the constraints of positive weights as in model (1.3). Different required returns give rise to different minimized volatility portfolios. These are plotted.

In [38]:
```
w = w_min ### w is now optimal portfolio weights, sum to 1
num_ports = 100
gap = (np.amax(r) - ret(r,w_min))/num_ports
### np.amax in numpy returns max in the array -- since weights sum to 1 and are bounded in (0,1). max portf ret is amax(r)
### The aboove range given by gap starts at ret given by Min Var Portf to Max of all mean returns -- maximum possibe

all_weights = np.zeros((num_ports, len(dfret.columns))) ### all_weights is 2D 100 x 8 zero matrix
### Note: len(dfret.columns) is 8 -- there are 8 stocks here
### print(np.shape(all_weights)) gives (100,8) -- same as print(all_weights.shape) that gives (100,8)

print("*******************************************************************")
####################################################################################################################
### First note that in Python, a Tuple is a grouping of unnamed, ordered values that can be of different types; an
###   array is a collection where elements' values can be changed, and are of a single type
###   all_weights is a 2D array of 100 rows each with 8 cols. all_weight[0] below is the first 1D sub-array that is 1st row
### Note: all_weights[0] is [0. 0. 0. 0. 0. 0. 0. 0.], i.e. first row of 100 x 8 all_weight
### Note: all_weights[1] is [0. 0. 0. 0. 0. 0. 0. 0.], i.e. second row of 100 x 8 all_weight
### .........
### Note: all_weights[99 is [0. 0. 0. 0. 0. 0. 0. 0.], i.e. 100th row of 100 x 8 all_weight
### print(all_weights[0].shape) gives (8,) -- a 1-tuple with 8 elements
### print(all_weights.shape[0]) gives 100, i.e. dimension of the rows
### print(all_weights.shape[1]) gives 8, i.e. dim of the cols
####################################################################################################################
ret_arr = np.zeros(num_ports) ### this is a 1-tuple of 100 zeros
### Note: print(ret_arr.shape) gives (100,) -- this is a 1D tuple
###       print(ret_arr.shape[0]) gives 100, the number of elements in 1D tuple
###       if we print(ret_arr.shape[1]) --- this gives "tuple index out of range" as there is no other dimension
### If we use instead ret_arr = np.zeros((num_ports,1)), then print(ret_arr.shape) gives (100,1), a dataframe with one column

vol_arr = np.zeros(num_ports)

for i in range(num_ports): ### this means looping from i=0 to 1,2,3,4,....,99 (100 loops in total)
    port_ret = ret(r,w) + i*gap
    double_constraint = LinearConstraint([np.ones(dfret.shape[1]),r],[1,port_ret],[1,port_ret])

    ### Create x0: initial guesses for weights.
    x0 = w_min
    ### Define a function for annualized portfolio volatility.
    portfvola = lambda w: np.sqrt(np.dot(w,np.dot(w,covar))*252)
    res = minimize(portfvola,x0,method='trust-constr',constraints = double_constraint,bounds = bounds)
       ### Above double constraints mean unit vector .dot(w) = 1; r .dot(w) = port_ret; minimize chooses wts w
    all_weights[i,:]=res.x ### i row x 8 optimal wts (at row i)
    ret_arr[i]=port_ret
    vol_arr[i]=vol(res.x,covar)
```

```
### Indented paras after "for i..." form the loop

sharpe_arr = ret_arr/vol_arr  ### sharpe_arr is 100 x 1 array since it is ret_arr[100]/vol_arr[100] element by element

plt.figure(figsize=(20,10))
plt.scatter(vol_arr, ret_arr, c=sharpe_arr, cmap='viridis')
### in plt.scatter, c is a scalar or sequence of n numbers to be mapped to colors using cmap
### in plt, for sequential plots, 'viridis' gives colors across the 3D representation of vol_arr, ret_arr, sharpe_arr
### c= in front of third dimension sharpe_arr gives the colors in that dimension, otherwise dots will be all blue
plt.colorbar(label='Sharpe Ratio')
plt.xlabel('Volatility')
plt.ylabel('Return')
plt.show()
```

```
*******************************************************************
```



The above scatter plot shows the minimum variance portfolio X (left lowest point) with a return of 6.22% and volatility of 19.08%. The maximum Sharpe ratio portfolio Y has a return of 29.69%, volatility of 26.97%, and a Sharpe ratio of is 1.1007. There are no short sales in the portfolios.

The following codes concatenates the day, month, year and forms a new data set of the 8 portfolio returns with 11 columns that is saved in ret_portecom.csv, i.e., using dfret1.to_csv

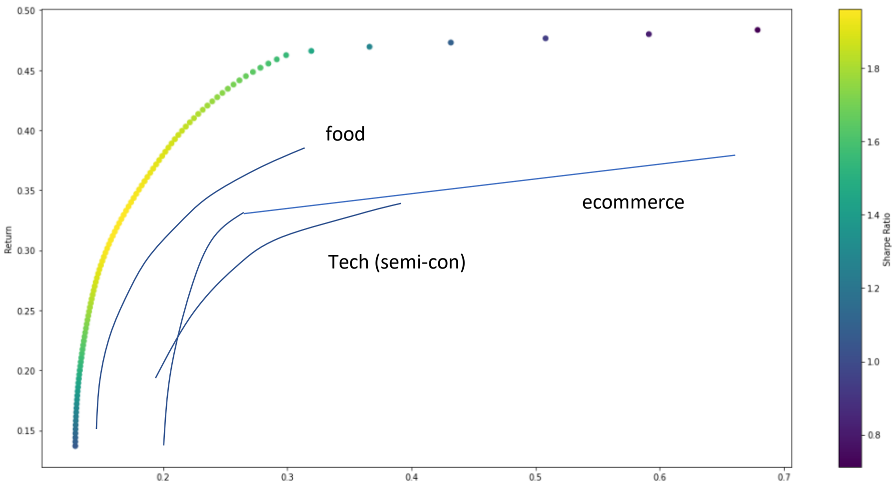
```
In [39]: dfret.shape
Out[39]: (1160, 8)

In [40]: t1=pd.concat([dy,mth,yr],axis=1) ### Axis=1 is important -- aligning columns
         t1=t1.iloc[1:,:]
         print(t1)

       day  month    year
1      4.0    1.0  2017.0
2      5.0    1.0  2017.0
3      6.0    1.0  2017.0
4      9.0    1.0  2017.0
5     10.0    1.0  2017.0
...    ...    ...     ...
1156  23.0   12.0  2021.0
1157  27.0   12.0  2021.0
1158  28.0   12.0  2021.0
1159  29.0   12.0  2021.0
1160  30.0   12.0  2021.0

[1160 rows x 3 columns]

In [41]: t1.shape
Out[41]: (1160, 3)

In [42]: dfret1=pd.concat([dfret, t1],axis=1)

In [43]: print(dfret1)

       AlibabaP   AmazonP     EbayP  RakutenP  SuningP  WayfairP  ZalandoP  \
1      0.021329  0.004646 -0.002685  0.027605  0.000848  0.036286 -0.038101
2      0.041763  0.030270  0.008365  0.014742 -0.003396  0.018738  0.023252
3     -0.005099  0.019716  0.034068  0.026001 -0.015424  0.005099  0.015787
4      0.008801  0.001168 -0.009709  0.006632 -0.001729  0.018300  0.001472
5      0.021205 -0.001281 -0.016394 -0.016183  0.017153  0.075646 -0.004421
...         ...       ...       ...       ...       ...       ...       ...
1156   0.007189  0.000184  0.014748  0.002049 -0.028779  0.002842  0.007973
1157  -0.017599 -0.008212  0.011797  0.013723  0.007273 -0.027230 -0.000851
1158  -0.015472  0.005827  0.006376  0.007543 -0.002418 -0.030737  0.004814
1159  -0.023889 -0.008592  0.005734  0.014918 -0.012180 -0.005929 -0.001414
1160   0.092801 -0.003294  0.004803 -0.012916  0.007326  0.005669  0.006204

        JDcomP   day  month    year
1     0.001161   4.0    1.0  2017.0
2     0.017258   5.0    1.0  2017.0
3    -0.001141   6.0    1.0  2017.0
4    -0.000381   9.0    1.0  2017.0
5     0.024079  10.0    1.0  2017.0
...        ...   ...    ...     ...
1156 -0.071660  23.0   12.0  2021.0
1157 -0.013787  27.0   12.0  2021.0
1158 -0.027551  28.0   12.0  2021.0
1159 -0.003650  29.0   12.0  2021.0
1160  0.070160  30.0   12.0  2021.0

[1160 rows x 11 columns]

In [44]: import pandas as pd
         dfret1.to_csv('ret_portecom.csv')
```

1.3 Forming Optimal Portfolios

Two other industry portfolios – technology (semi-conductor firms) and food – are similarly formed, each with 8 stocks. The tech stocks include Broadcomm, Intel, Microchip, Micron, Qualcomm, Samsung, SK Hynix, and SMIC. The food stocks include Tyson, Pepsico, Nestle, Mondelez, Kweichow, Diageo, Danone, and Anheuser-Busch.

The 3 industry portfolios in data sets ret_portecom.csv, ret_porttech.csv, and ret_portfood.csv with a total of 24 stocks are merged into a larger portfolio for similar optimizations. See demonstration file Chapter1-2.ipynb. The data set is further split into two parts: a training set using data from 2017 till 2020, and a test set of data in 2021. But we use only the first 112 data points at beginning of 2021 to about June 2021 – this is to ensure recency when we apply the optimal weights calculated from the training set and apply them to the new return data immediately after the training period.

The outcome of using the training set data to estimate the portfolio mean returns and portfolio return covariance and to form the efficient portfolio all at once is shown below. The efficient frontiers of the 3 separate industry portfolios, though using a slightly longer data set including 2021 are also shown. Clearly the larger combined industry set of stocks produce a better performing efficient frontier – for any given expected return, the portfolio volatility is smaller. This is evidence of the benefit of portfolio diversification when more stock returns are added which have lower variances and low correlations with the others.



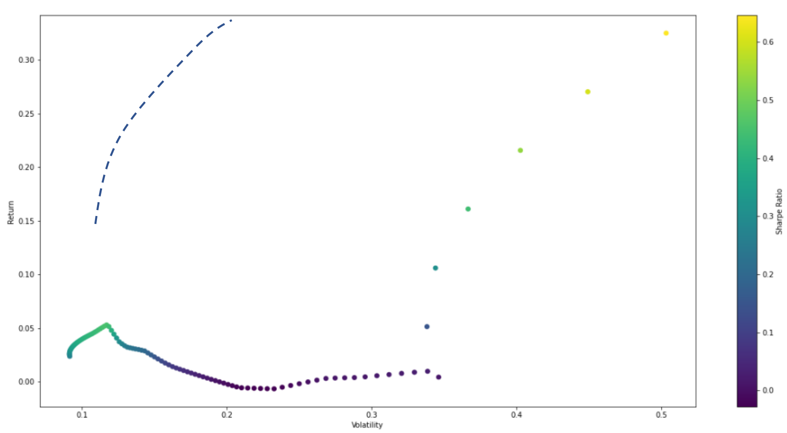
However, the above results on expected portfolio return versus portfolio return volatility is based on finding weights within a given sample and evaluating the frontier based on those weights. After the optimal weights for every k, as in (1.3), are computed, suppose we apply these weights to fresh data that is out-of-sample in the test data set. Would we obtain the out-of-sample or test set predicted portfolio returns and volatilities that are similar to that mapped in the training set? We use the test set to construct the out-of-sample mean and covariance matrix.

The results are shown below.

```
### Below the test set data is used with the optimal wts computed with training set to form the eff portf frontier
testcovar=testset.cov()
testr = np.mean(testset,axis=0)*252

### initialize
testport_ret = np.zeros(num_ports)
testport_vol = np.zeros(num_ports)

for i in range(num_ports):
    testport_ret[i] = ret(testr,all_weights[i,:])
    testport_vol[i]= vol(all_weights[i,:],testcovar)

testport_sharpe = testport_ret/testport_vol
```

```
plt.figure(figsize=(20,10))
plt.scatter(testport_vol, testport_ret, c=testport_sharpe, cmap='viridis')
### in plt.scatter, c is a scalar or sequence of n numbers to be mapped to colors using cmap
### in plt, for sequential plots, 'viridis' gives colors across the 3D representation of vol_arr, ret_arr, sharpe_arr
### c= in front of third dimension sharpe_arr gives the colors in that dimension, otherwise dots will be all blue
plt.colorbar(label='Sharpe Ratio')
plt.xlabel('Volatility')
plt.ylabel('Return')
plt.show()
```

Clearly, the training set mean return μ and portfolio return volatility based on training set Σ would be different for the test set mean return and covariance matrix Σ. The resulting portfolio performances based on the fresh test returns data show substandard performance as compared with the in-sample dotted efficient frontier curve.



There are at least two possible reasons why using out-of-sample or test set data would produce substandard performances. In this case, the stocks with high weights computed from the training data set has generally lower returns in the test data set than in the training data set. Thus, the new average portfolio return is substandard. The performance errors are created due to the randomness of returns that could produce large deviations from the expectations in μ. Another possible reason is that the estimation of the covariance matrix Σ using the training set data could contain random errors and similarly this would affect the out-of-sample portfolio return volatility based on the fresh returns.

Both mean return and portfolio return volatility errors could be reduced if we have a model to better explain the stock returns such as in a multi-factor linear model. In this case, if the factors are correctly found and anticipated, the errors remain only in the residuals of the linear model, and these errors would be smaller. We do not go into multi-factor models here but suggest another method popular in machine learning approach to attempt to reduce the error in the use of in-sample estimate of Σ. In a later section, we show how a more accurate forward prediction of future expected return could be used to improve on training set optimal portfolio weights for future investing results.

1.4 Denoising the Correlation Matrix

In finance theory, one often thinks of the stock return covariance and corresponding correlation matrices as constant matrices. Their sample estimates are random due to small sampling errors – but with fixed N number of stocks and a time series T that approaches $+\infty$, i.e. $N/T \downarrow 0$, the sample estimate of the correlation matrix $\hat{\Sigma}$ would converge to the population constant of $\Sigma_{N\times N}$ given stationary time series of the stock returns.

However, suppose N, T are increasingly large, so that N/T does not converge to zero, but instead converge as $N, T \rightarrow +\infty$, to some finite positive number, i.e., $0 < N/T < 1$. In small sample, it is also the case that $0 < N/T < 1$. Then every element in the (small) sample correlation matrix is a random variable. We may treat the sample correlation matrix in this context as a random matrix which is in general a matrix-valued random variable.

A square matrix A is positive definite if for any non-zero conformable vector w, $w^T A w > 0$. If $A_{N\times N}\, w_{iN\times 1} = \lambda_i\, w_{iN\times 1}$ for scalar λ_i, then λ_i is called an eigenvalue of A while w_i is the corresponding eigenvector. There are N number of eigenvalues and eigenvectors. The eigenvalues are found by solving the determinant, $|A - \lambda_i I|$. Some of these may be repeated. In general, these eigenvectors are non-unique, so additional constraints to make them normalized and orthogonal to each other are imposed. In other words, for any i,j eigenvectors, w_i , w_j: $w_i^T w_i = 1$, $w_j^T w_j = 1$, and $w_i^T w_j = 0$ for $i \neq j$.

A positive definite symmetric matrix has strictly positive eigenvalues. A matrix is positive definite if it is symmetric, and all its eigenvalues are strictly positive. Hence sample or else population covariance and correlation matrices of stock returns, that are both symmetric and positive definite, have positive eigenvalues.

Suppose a vector of random variables change over time due to independent and identically distributed noise (with mean 0 and variance of 1) and not signals. Noises, unlike signals, are not systematic factors in the market. Then the Marcenko-Pastur Theorem states that as N, T $\to +\infty$, and N/T converge to a finite positive number in (0,1), then the eigenvalues λ_i of the sample correlation matrix converge to a probability density function (pdf) as follows (and not a degenerate single value distribution of one).

The important characterization of the random matrix is that the eigenvalues are random variables.

$$\text{pdf}\ (\lambda_i) = \frac{\sqrt{(U-\lambda_i)(\lambda_i-L)}}{2\pi\lambda_i\sigma^2 N/T}$$

where $U = \sigma^2\left(1+\sqrt{N/T}\right)^2$ and $L = \sigma^2\left(1-\sqrt{N/T}\right)^2$, for $\lambda_i \in [L, U]$. Outside of $[L, U]$, pdf $(\lambda_i) = 0$.

The above result is demonstrated in the following codes in demonstration file Chapter1-3.ipynb whereby T × N (T=50,000 and N=900) random normal numbers with mean 0 and variance 1 are generated as noises to find estimates of the eigenvalues of the sample correlation matrix of the simulated N number of random numbers.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
```

```
In [2]: ### To obtain the eigenvalues and eigenvectors (normalized and orthogonal) from a matrix, sorting eigenvalues in ascending order
        def getPCA(matrix):
            eVal,eVec=np.linalg.eigh(matrix) ### ..eigh(x) -- x is  real symmetric or complex Hermitian (conjugate symmetric) array.
               ### Returns two objects, a 1-D array eVal =eigenvalues of matrix - eigenvalues with multiplicity, may not be ordered.
               ### and a 2-D square matrix, column v[:, i] is the normalized eigenvector corresponding to the eigenvalue w[i].
            indices=eVal.argsort() ### Returns the indices that would sort an array.
            eVal,eVec=eVal[indices],eVec[:,indices] ### ensures eVal and eVec elements are increasing in order of eVal or eigenvalues
            eVal=np.diagflat(eVal) ### Create a two-dimensional array with the flattened (changing to 1dim) input eVal as a diagonal.
            return eVal,eVec
```

```
In [3]: x = np.random.normal(0, 1, size=(50000,900)) ### first argument mean, second std dev. Random matrix dim in size (T,N)
```

```
In [4]: eVal0,eVec0=getPCA(np.corrcoef(x,rowvar=False))
          ### matrix in getPCA argument is correlation matrix from x.
          ### If rowvar is True (default), then each row represents a variable, with observations in the columns.
          ### Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.
          ### np.corrcoef(x,rowvar=False).shape = (900,900); len(eVal0) = 900; eVec0.shape is (900,900)
```

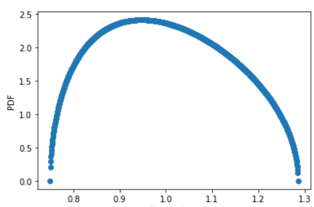
Code line [2] produces a user-defined function getPCA to derive the 900 × 1 eigenvalues and corresponding 900 × 900 eigenvectors (900 columns of eigenvectors with each eigenvector a dimension of 900 × 1) in [4] based on random correlation matrix of x. (In this case, it is also the covariance matrix of x.) We can think of sample data $x_{50,000 \times 900}$ in [3] as deriving from a random vector of $X_{900 \times 1}$ where the i^{th} column of x represents the time series of the i^{th} row random variable in X. Note that X has zero mean and a standard deviation of one for all random variables. Taking the i^{th} eigenvector w_i, $w_i^T X$ is the i^{th} principal component of X. It is a linear combination of X with a variance equal to its eigenvalue λ_i. The sum of all the eigenvalues or $\sum_{i=1}^{900} \lambda_i = 900$ which is also the total variance of the correlation matrix Σ, i.e., its trace or sum of the diagonal elements. Hence, the largest principal components explain most of the variances present in the random X.

Code lines [5], [6] compute the approximate theoretical Marcenko-Pastur pdf given T=50,000 and N=900. This is plotted in code line [7].

```
In [5]: def randommatpdf(var,q,pts):  ###
            eMin, eMax = var*(1-(1./q)**.5)**2,var*(1+(1./q)**.5)**2
            eVal=np.linspace(eMin,eMax,pts) ### returns pts (no.) equally spaced vector of nos. starting at eMin to eMax
            pdf=q/(2*np.pi*var*eVal)*((eMax-eVal)*(eVal-eMin))**.5
            pdf=pd.Series(pdf,index=eVal) ### Pandas uses Series (similar to 1-dim array in numpy, but essential difference
                                          ### is the presence of the index in pandas that can be defined,
                                          ### whereas Numpy Array has an implicitly defined integer index
            return pdf
```

```
In [6]: pdf0=randommatpdf(1.,q=x.shape[0]/float(x.shape[1]),pts=900) ### print(x.shape[0]) = 50,000; print(x.shape[1])=900
        ### Here variance var is entered as 1. pts no. must match simulated N. float() needs not be used unless item is to be divided
        ### and number needs be a floating type, i.e. with decimal
```

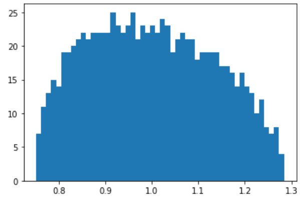
```
In [7]: plt.scatter(pdf0.index, pdf0) ### This is the theoretical Marcenko-Pastur pdf of the eigenvalues
        plt.xlabel('Eigenvalues')
        plt.ylabel('PDF')

Out[7]: Text(0, 0.5, 'PDF')
```



Code lines [8] and [9] evaluate the eigenvalues and plot it as a histogram.

```
In [8]: ### "print(np.diagonal(eVal0).shape)" gives (900,)
        ### "eVal0.shape" gives (900,900)-- 2D numpy array
        eigenvalues=np.diagonal(eVal0) ### this changes the diagonal matrix eVal0 back to single column with eigenvalues.shape as (900,)
        ### "print(eigenvalues)" can be used to check program line is correct and delivers the required output
```

```
In [9]: import matplotlib.pyplot as plt

        plt.hist(eigenvalues, bins = 50)
        plt.show()
        ### This empirical pdf seems to be explained well by the theoretical pdf
```



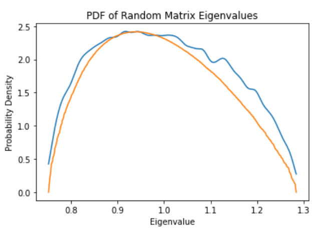
Code lines [10], [11] evaluate the empirical pdf of the eigenvalues generated by the simulated i.i.d. random variables. The kernel refers to the integrand function whereby the integral produces the area under the curve of one. The kernel density estimation provides the empirical probability density function. Different kernel function specification may produce slightly different smoothness in the empirical pdf.

```
In [10]: ### kernel density estimation (KDE) is a nonparametric smoothing method to estimate a probability density estimation
         ### The y-axis or count of the corresponding Histogram is calibrated to pdf measures such that area under pdf = 1
         ### Valid kernels are ['gaussian'|'tophat'|'epanechnikov'|'exponential'|'linear'|'cosine'] Default is 'gaussian'.
         from sklearn.neighbors import KernelDensity
         def fitKDE(obs,bWidth=.25,kernel='gaussian',x=None):
             if len(obs.shape)==1: obs=obs.reshape(-1,1)
             kde=KernelDensity(kernel=kernel,bandwidth=bWidth).fit(obs)
             if x is None:x=np.unique(obs).reshape(-1,1)
             if len(x.shape)==1: x=x.reshape(-1,1)
             logProb=kde.score_samples(x)
             pdf=pd.Series(np.exp(logProb),index=x.flatten())
             return pdf
```

```
In [11]: pdf1=fitKDE(eigenvalues,bWidth=.01)   ### pdf1 is an object wrapping together eigenvalue and its pdf
```

```
In [12]: import seaborn as sns
         fig, ax = plt.subplots()
         ax = sns.lineplot(x=eigenvalues, y=pdf1, ax=ax) ### prints empirical pdf
         ax = sns.lineplot(x=eigenvalues, y=pdf0, ax=ax) ### prints theoretical pdf
         ax.set_title('PDF of Random Matrix Eigenvalues')
         ax.set_xlabel('Eigenvalue')
         ax.set_ylabel('Probability Density')
         plt.show()
```

Code line [12] plots the empirical pdf of the computed eigenvalues side-by-side the approximate theoretical pdf of the 900 eigenvalues.

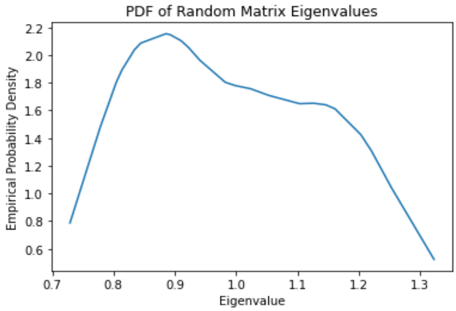



```
In [13]: print(np.min(eigenvalues),np.max(eigenvalues))

         0.7514753569204135 1.2841886690523217
```

It is seen that for the large T=50,000 and N=900, the empirical pdf looks similar to the theoretical pdf (area under the pdf curve is one).

However, when we use only N=24, T=889 (size of the training set data) similar random normal numbers with mean zero and variance one, the empirical pdf is as follows. See demonstration file Chapter1-4.ipynb. This empirical pdf does not approximate the theoretical pdf as closely since N, T are small. Nevertheless, it still shows that when the randomness is just independently identically distributed noises, eigenvalues can still range from 0.7 to 1.3. The idea is to look for significantly larger values of eigenvalues that would signal the presence of signals and not just noises.

```
import seaborn as sns
fig, ax = plt.subplots()
ax = sns.lineplot(x=eigenvalues, y=pdf1, ax=ax) ### prints empirical pdf
ax.set_title('PDF of Random Matrix Eigenvalues')
ax.set_xlabel('Eigenvalue')
ax.set_ylabel('Empirical Probability Density')
plt.show()
```



Suppose we use the actual stock return data of the N=24 stocks over time series T=889 in the training data set to form the correlation matrix $\Sigma_{24\times24}$, 'corr_matrix'. In the program, this is computed as corr_matrix = trainingset.corr(). The eigenvalues and eigenvectors of $\Sigma_{24\times24}$ are found using the following. 'eigenvalues' show the list of 24 computed eigenvalues. See demonstration file Chapter1-5.ipynb.

```
### Create a principal component analysis (PCA) plot for the first two dimensions.
def getPCA(matrix):
    eVal,eVec=np.linalg.eigh(matrix) ### ..eigh(x) -- x is  real symmetric or complex Hermitian (conjugate symmetric) array.
       ### Returns two objects, a 1-D array eVal =eigenvalues of matrix - eigenvalues with multiplicity, may not be ordered.
       ### and a 2-D square matrix, column v[:, i] is the normalized eigenvector corresponding to the eigenvalue w[i].
    indices=eVal.argsort() ### Returns the indices that would sort an array. -1 refers to the last axis
    eVal,eVec=eVal[indices],eVec[:,indices]
    eVal=np.diagflat(eVal) ### Create a two-dimensional array with the flattened (changing to 1dim) input eVal as a diagonal.
    return eVal,eVec
```

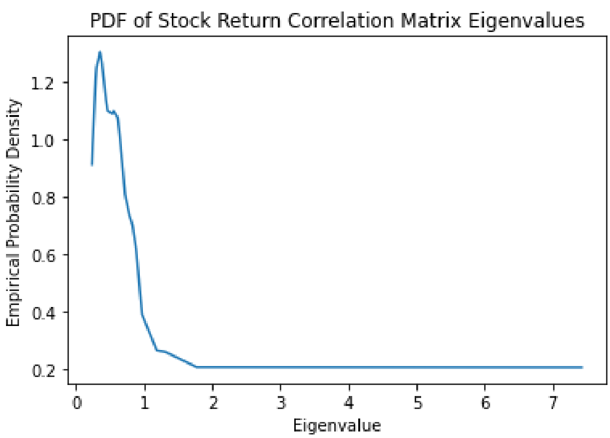
```
eVal0,eVec0=getPCA(corr_matrix)
```

```
eigenvalues=np.diagonal(eVal0) ### creates a list of the diagonal elements in original
print(eigenvalues)
```

```
[0.23777545 0.24830976 0.29134883 0.30236206 0.35499693 0.36322877
 0.38419966 0.44332669 0.46659641 0.535817   0.55524774 0.61387326
 0.61922829 0.64157765 0.72200588 0.78494843 0.83230546 0.88442046
 0.97049917 1.18529588 1.317256   1.77494235 2.04324288 7.42719501]
```

Note that the correlation matrix 'corr_matrix' is the sample correlation matrix of the test set returns, which is also the covariance matrix of the standardized test set returns. The latter is also the correlation matrix of the standardized test set returns. Hence 'corr_matrix' is correlation matrix of the standardized test set returns. It may be different from the simulated random normal numbers with mean zero and variance one in that the off-diagonal correlation numbers are theoretically not necessarily zeros in the returns sample.

The computed empirical pdf of the eigenvalues is shown as follows.



It is seen that most of the eigenvalues falls below 1.3 and these variances of the principal components could be due to white noise. Only those eigenvalues > 1.3 represent combinations of the standardized stock return random variables (with mean 0 and variance 1) that produce bigger variances that could be due to true market signals and not white noise.

Denoising the empirical correlation matrix is one way to produce a more accurate correlation matrix for portfolio optimization. We show one method of denoising – the constant residual eigenvalue method. This approach is to set a constant eigenvalue for all random eigenvalues that are smaller than or equal to U which we can approximate using 1.3 as seen in the pdf of the random matrix eigenvalues with N=24 and T=889. This constant is the average of all the eigenvalues below 1.3. Thus the total variance is preserved.

Computation of the denoised correlation matrix ‘corr2’ and the reconstituted denoised covariance matrix ‘cov2’ are shown in the code lines below.

```
### The 21st to 24th elements on the eigenvalue list are above 1.3
### The constant residual eigenvalue method is applied to average as constant all eigenvalues below 1.3
### -- or even nullify as they could be just from noises

small=20 ### eigenvalues elements up to order 20 are all below 1.3
neweig=eigenvalues.copy()   ### rather similar without .copy()
neweig[:small]=eigenvalues[:small].sum()/small
eigenvalues1=np.diagflat(neweig)
corr1=np.dot(eVec0,eigenvalues1).dot(eVec0.T) ### eVec.T is transpose of eVec
dd=np.diag(corr1)
corr2=corr1/dd ### divides ith column of corr1 by ith element in list dd, same effect as corr1 .dot(np.diagflat(1/dd))
```

```
### Now we use corr2 as denoised correlation matrix of the 24 stock returns
```

```
### corr2 is transformed back to corresponding covariance matrix using original variances
var = trainingset.var()
sd = np.sqrt(var)  ### here output comes from pandas -- dataframe, so some numpy commands may not work as they are not arrays
                   ### such as np.diagflat()
```

```
sd1=sd.to_numpy()  ### now sd1 is array
```

```
sd2=np.diagflat(sd1) ### sd2 is 24 x 24 diagonal matrix with diagonal as std devs
```

```
cov2=(sd2 .dot(corr2)) .dot (sd2.T)
```

Finally, we use the denoised covariance matrix to re-compute the optimal weights, ‘all_weights’, of portfolio based on the training data set with the same portfolio return means but using the denoised covariance matrix, ‘decovar’ in the following code line. See demonstration file Chapter1-6.ipynb.

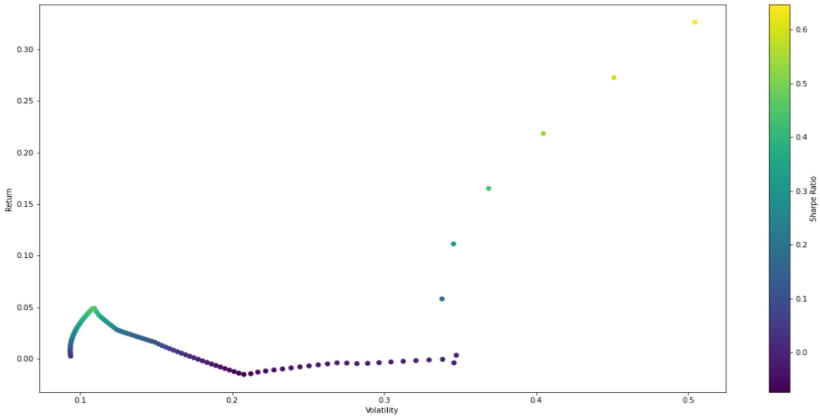
```
w = w_min   ### w is now optimal portfolio weights, sum to 1
num_ports = 100
gap = (np.amax(r) - ret(r,w_min))/num_ports
### np.amax in numpy returns max in the array -- since weights sum to 1 and are bounded in (0,1). max portf ret is amax(r)
### The aboove range given by gap starts at ret given by Min Var Portf to Max of all mean returns -- maximum possibe

all_weights = np.zeros((num_ports, len(trainingset.columns)))   ### all_weights is 2D 100 x 24 zero matrix
### Note: len(trainingset.columns) is 24 -- there are 24 stocks here
### print(np.shape(all_weights)) gives (100,24) -- same as print(all_weights.shape) that gives (100,24)

print("*****************************************************************")
##########################################################################################################################
```

```
ret_arr = np.zeros(num_ports) ### this is a 1-tuple of 100 zeros
vol_arr = np.zeros(num_ports)

for i in range(num_ports): ### this means looping from i=0 to 1,2,3,4,....,99 (100 loops in total)
    port_ret = ret(r,w) + i*gap
    double_constraint = LinearConstraint([np.ones(trainingset.shape[1]),r],[1,port_ret],[1,port_ret])
      ### Above, objective minimization is doubly constrained to have wts sum to one and Sum wts x rets sum to port_ret

    ### Create x0: initial guesses for weights.
    x0 = w_min
    ### Define a function for portfolio volatility.
    portfstderr = lambda w1: np.sqrt(np.dot(w1,np.dot(w1,decovar))*252)
    optweight = minimize(portfstderr,x0,method='trust-constr',constraints = double_constraint,bounds = bounds)

    all_weights[i,:]=optweight.x   ### 24 x 1 optimal wts at row i
    ret_arr[i]=port_ret
    vol_arr[i]=vol(optweight.x,decovar)
```

```
### Indented paras after "for i..." form the loop

sharpe_arr = ret_arr/vol_arr  ### sharpe_arr is 100 x 1 array since it is ret_arr[100]/vol_arr[100] element by element

plt.figure(figsize=(20,10))
plt.scatter(vol_arr, ret_arr, c=sharpe_arr, cmap='viridis')
### in plt.scatter, c is a scalar or sequence of n numbers to be mapped to colors using cmap
### in plt, for sequential plots, 'viridis' gives colors across the 3D representation of vol_arr, ret_arr, sharpe_arr
### c= in front of third dimension sharpe_arr gives the colors in that dimension, otherwise dots will be all blue
plt.colorbar(label='Sharpe Ratio')
plt.xlabel('Volatility')
plt.ylabel('Return')
plt.show()
```

****

After that we employ the test data with the optimal weights computed as above to find the new portfolio results. This is shown as follows.



However, in this case, there is no significant change from the case when the returns covariance matrix is not denoised. This could be due to larger errors in the portfolio expected returns than in the portfolio return variances. Such methods may be more effective in other problems where the means do not change as much.

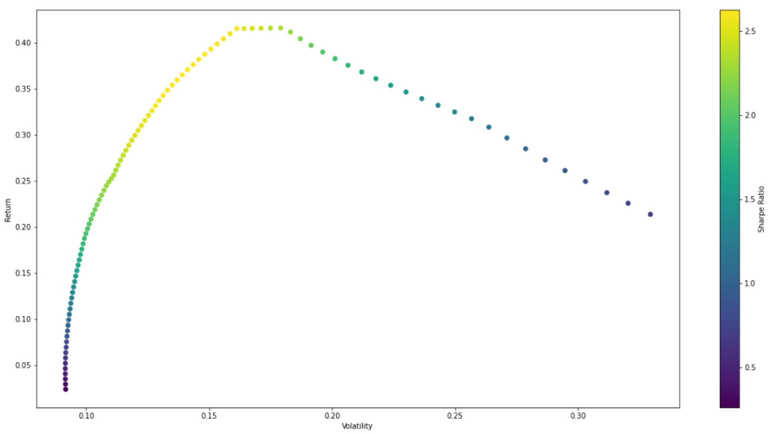
1.5 Using a More Accurate Forward Predictor

As shown earlier, diversification is limited in improving results if the future returns have means that are not similar to the means based on historical returns used to compute the optimal portfolio weights. In fact, most of the times, accurate predictions of future means count more toward better portfolio performance than diversification itself. This is why prediction of future (expected) returns or similarly prediction of future stock prices is such an important business in financial data science. In this section, we show how a more accurate forward prediction of future expected return could be used to improve on training set optimal portfolio weights for future investing results.

See demonstration file Chapter1-7.ipynb. We employ the same training set return data of the 24 stocks. This is used as before to compute the covariance matrix. For the stock mean returns, however, we assume there is a good prediction model – for illustration, suppose these predicted returns are identical to the mean returns based on the 224 returns observations per stock in 2021. (This is of course forward looking – but is used as an illustration here.) This mean vector is then used together with the historical covariance to compute the optimal portfolio weights.

The returns in the first 111 observations of 2021 form the test data set which is a subset of the 224 observations. The test data set returns are therefore randomly distributed across its means (based on 111 observations) that are close to that based on the 224 observations.

After the optimal weights for every (required return) k are computed using the training set, suppose we apply these weights to fresh data that is out-of-sample in the test data set of 111 observations. This test set is used to construct the out-of-sample mean and covariance matrix of the portfolio returns. The results are shown below.



Clearly, for whichever set of optimal weight chosen (depending on the target required returns), the performance is close to the ex-ante frontier (for volatility < 0.16.

1.6 Summary

Portfolio optimization is a ubiquitous part of buy-side investments. It achieves good in-sample risk-return performances by diversifying risks and attempting to attain the highest expected portfolio return for a given level of estimated portfolio return volatility or risk or to attain the minimum risk for a given level of expected portfolio return. However, there is difficulty of attaining good ex-post portfolio performance results when the ex-post returns or ex-post covariances, particularly the former, differ in a significant way from those used in the training or in-sample.

Theoretically, minimizing volatility subject to required return may also not be the most preferred outcome. Some investors prefer to maximize the return per unit volatility – Sharpe ratio. It is also possible that portfolio return positive skewness is desired and increasing weights to increase portfolio return skewness may attain better performances although typically the volatility would be higher.

The machine learning approach could attempt to reduce the ex-portfolio performance problem by attempting to form the expected returns and estimated covariances based on predictions rather than based simply on historical data. We can use factor models to try to find better estimates of future returns. We can also use Neural Networks or other ML prediction methods to predict next period expected stock returns and use these for constructing optimal weights with the training set data.

For avoiding noise in estimating training covariance matrix, we show how to use the constant residual eigenvalue method to denoise and form a more accurate covariance matrix for the portfolio weight optimization.

1.7 Other References

https://towardsdatascience.com/the-data-science-trilogy-numpy-pandas-and-matplotlib-basics-42192b89e26
https://builtin.com/data-science/portfolio-optimization-python
Marchenko, V. A.; Pastur, L. A. (1967). "Распределение собственных значений в некоторых ансамблях случайных матриц" [Distribution of eigenvalues for some sets of random matrices]. *Mat.Sb.* N.S. (in Russian). **72** (114:4): 507–536. doi:10.1070/SM1967v001n04ABEH001994
Markowitz, Harry, (1952), "Portfolio Selection", The Journal of Finance, Vol. 7, No. 1., pp. 77-91