Sid Bhatia - FE630 ~ Midterm (PDF)

March 31, 2024

0.0.1 FE630 - Midterm Project

Author: Sid Bhatia

Date: March 25th, 2023

Pledge: I pledge my honor that I have abided by the Stevens Honor System.

Professor: Papa Momar Ndiaye

Question 1. (15 pts) The supplied `data.zip` file contains 30 space-delimited text files that contain price and volume data for 30 companies. Each row of each file contains date, opening price, closing price, high price, low price, volume, and adjusted closing price (last column). You will need that data for question 1.

Write a program `processdata` to:

1. Read all daily price files;
2. Create a price matrix `P` by aligning the data's dates and placing the adjusted closing prices side-by-side in columns;
3. From the `P matrix`, create a matrix of simple (not logarithmic) daily returns `R`;
4. Compute the vector of average daily returns mu for the companies using the `mean` function (do not use loops);
5. Compute the covariance matrix `Q` from the return matrix using the `cov` function; and
6. Save the return vector `mu` and the covariance matrix `Q` in the native format for your programming language in a file called `inputs.ext`, where `ext` is the appropriate extension for a binary file in your language.

[ ]:
```
import pandas as pd
import numpy as np
import os
from typing import List

def processdata(data_dir: str = 'data') -> None:
    """
    Processes stock price data to compute and save the average daily return␣
 ↪vector and covariance matrix.

    This function reads stock price data from text files, each containing data␣
 ↪for a company, then:
    1. Creates a price matrix with adjusted close prices,
    2. Calculates the daily return matrix,
    3. Computes the vector of average daily returns for each company,
    4. Computes the covariance matrix of the return matrix,
    5. Saves the average daily returns vector and the covariance matrix to␣
↪binary files.

    Parameters:
    - data_dir (str): The directory containing the stock price files. Default␣
↪is 'data'.

    Returns:
    - None. The function saves two files: 'inputs_mu.pkl' and 'inputs_Q.pkl'␣
↪with the results.
    """
    # List to store the adjusted close price data for each company.
    price_data: List[pd.Series] = []

    # Loop through each file in the specified directory.
    for file in os.listdir(data_dir):
        if file.endswith('.txt'):
            filepath = os.path.join(data_dir, file)
            # Read data, assuming space-separated values without an explicit␣
↪header.
            df = pd.read_csv(filepath, sep=' ', header=None,
                             names=['Date', 'Open', 'Close', 'High', 'Low',␣
↪'Volume', 'Adj Close'])
            # Set date as the index for easy alignment later.
            df.set_index('Date', inplace=True)
            # Append the adjusted close price series to the list.
            price_data.append(df['Adj Close'])

    # Concatenate all the adjusted close prices side-by-side, aligning by date.
    P = pd.concat(price_data, axis=1)
    P.sort_index(inplace=True)  # Ensure the dates are in order.

    # Calculate daily returns by comparing each price with the previous day's␣
↪price.
    R = P.pct_change().dropna()  # Drop the first row since its percentage␣
↪change is undefined.

    # Calculate the vector of average daily returns for each company.
    mu = R.mean(axis=0)

    print(mu.head())
    # Calculate the covariance matrix of the daily returns.
    Q = R.cov()

    print(Q.head())

    # Save the vector of average daily returns and the covariance matrix as␣
 ↪binary files.
    mu.to_pickle('inputs_mu.pkl')
    Q.to_pickle('inputs_Q.pkl')

processdata()
```

```
Adj Close    0.000225
Adj Close    0.000473
Adj Close    0.000945
Adj Close    0.000540
Adj Close   -0.000340
dtype: float64
           Adj Close  Adj Close  Adj Close  Adj Close  Adj Close  Adj Close  \
Adj Close   0.000342   0.000079   0.000080   0.000097   0.000104   0.000069
Adj Close   0.000079   0.000144   0.000066   0.000088   0.000062   0.000051
Adj Close   0.000080   0.000066   0.000169   0.000076   0.000058   0.000056
Adj Close   0.000097   0.000088   0.000076   0.000215   0.000075   0.000073
Adj Close   0.000104   0.000062   0.000058   0.000075   0.000166   0.000059

           Adj Close  Adj Close  Adj Close  Adj Close  …  Adj Close  \
Adj Close   0.000100   0.000084   0.000075   0.000076  …   0.000076
Adj Close   0.000056   0.000060   0.000071   0.000063  …   0.000063
Adj Close   0.000050   0.000058   0.000066   0.000062  …   0.000056
Adj Close   0.000074   0.000073   0.000080   0.000074  …   0.000071
Adj Close   0.000077   0.000061   0.000055   0.000077  …   0.000071

           Adj Close  Adj Close  Adj Close  Adj Close  Adj Close  Adj Close  \
Adj Close   0.000055   0.000037   0.000052   0.000054   0.000072   0.000073
Adj Close   0.000055   0.000043   0.000043   0.000055   0.000052   0.000068
Adj Close   0.000046   0.000037   0.000034   0.000049   0.000062   0.000078
Adj Close   0.000055   0.000039   0.000040   0.000069   0.000079   0.000072
Adj Close   0.000043   0.000030   0.000043   0.000047   0.000053   0.000068

           Adj Close  Adj Close  Adj Close
Adj Close   0.000050   0.000044   0.000089
Adj Close   0.000052   0.000041   0.000053
Adj Close   0.000041   0.000039   0.000049
Adj Close   0.000040   0.000041   0.000066
Adj Close   0.000040   0.000031   0.000072

[5 rows x 30 columns]
```

Question 2. (15 pts) Write a function called `port` that uses standard quadratic programming libraries that will:

- Take the set of input parameters `mu` (mean vector μ), `Q` (covariance matrix Q), and `tau` (risk tolerance τ), and return vector h that maximizes the following utility function U defined by:

  $$U(h) = -\frac{1}{2}h^T Q h + \tau h^T \mu$$

  subject to the constraints

  $$0 \leq h_i \leq 0.1 \text{ for all } i, \text{ and}$$

  $$\sum_{i=1}^{n} h_i = h^T e = 1$$

  where n is the number of securities in the portfolio.

[ ]:
```python
import cvxpy as cp

def port(mu: np.ndarray, Q: np.ndarray, tau: float) -> np.ndarray:
    """
    Solves the portfolio optimization problem to maximize the utility function
    under given constraints using quadratic programming.

    Parameters:
    - mu (np.ndarray): The mean return vector for the securities.
    - Q (np.ndarray): The covariance matrix of the securities' returns.
    - tau (float): The risk tolerance parameter of the utility function.

    Returns:
    - h (np.ndarray): The optimized portfolio weights vector.
    """

    # Number of securities
    n = mu.shape[0]

    ## print(n)

    # Portfolio weights variable
    h = cp.Variable(n)

    ## print(h)

    # Define utility function to maximize.
    utility = -0.5 * cp.quad_form(h, Q) + tau * cp.matmul(h.T, mu)

    ## print(utility)

    # Constraints: 0 <= h_i <= 0.1 for all i, and sum(h_i) == 1
    constraints = [0 <= h, h <= 0.1, cp.sum(h) == 1]
    ## print(constraints)

    # Problem definition
    problem = cp.Problem(cp.Maximize(utility), constraints)

    # Solve the problem.
    problem.solve()

    # Return the optimized portfolio weights.
    return h.value
```

Question 3. (15 pts) Write a program called `frontier` that will:

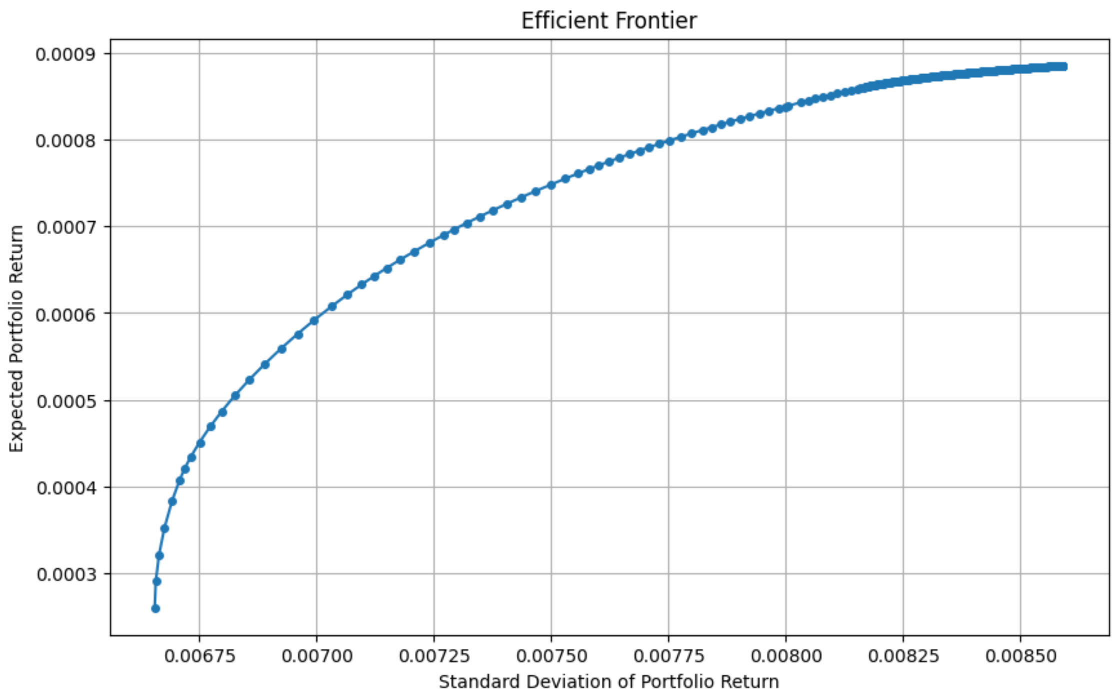
1. Load in your programming environment the data stored in the `inputs.ext`;
2. Create a sequence `TAU` containing numbers from zero to 0.5 in steps of 0.001;
3. Run through a loop for each value of your `TAU` sequence to

- Find the optimum portfolio with the given `m`, `Q`, and `tau` selected from `TAU`;
- Compute the optimum portfolio's expected return and standard deviation of return;
- Store the portfolio return and standard deviation.

4. After completing the loop, plot the efficient frontier.

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

def load_inputs(mu_path: str = 'inputs_mu.pkl', Q_path: str = 'inputs_Q.pkl')␣
 ↪-> (np.ndarray, np.ndarray): # type: ignore
    """
    Loads the mean returns vector and covariance matrix from the specified␣
 ↪files.

    Parameters:
    - mu_path (str): Path to the file containing the mean returns vector.
    - Q_path (str): Path to the file containing the covariance matrix.

    Returns:
    - tuple: A tuple containing the mean returns vector and the covariance␣
 ↪matrix.
    """
    mu = pd.read_pickle(mu_path).values
    Q = pd.read_pickle(Q_path).values
    return mu, Q

def compute_metrics(mu: np.ndarray, Q: np.ndarray, h: np.ndarray) -> (float,␣
 ↪float): # type: ignore
    """
    Computes the expected return and standard deviation of the portfolio.
    Parameters:
    - mu (np.ndarray): The mean return vector for the securities.
    - Q (np.ndarray): The covariance matrix of the securities' returns.
    - h (np.ndarray): The portfolio weights.

    Returns:
    - tuple: A tuple containing the expected return and standard deviation of␣
 ↪the portfolio.
    """
    expected_return = np.dot(mu.T, h)
    std_dev = np.sqrt(np.dot(h.T, np.dot(Q, h)))
    return expected_return, std_dev

def frontier():
    """
    Plots the efficient frontier by optimizing portfolios over a range of risk␣
 ↪tolerance values using the `port` function.
    Loads mean returns vector and covariance matrix, then calculates and plots␣
 ↪the efficient frontier.
    """
    # Load the mean returns vector and covariance matrix from saved files.
    mu, Q = load_inputs()
    TAU = np.arange(0, 0.501, 0.001)  # Range of risk tolerance values
    returns = []  # List to store expected returns of optimized portfolios
    std_devs = []  # List to store standard deviations of optimized portfolios

    for tau in TAU:
        # Use the previously defined port() function for portfolio optimization.
        h_opt = port(mu, Q, tau)
        # Compute expected return and standard deviation for the optimized␣
 ↪portfolio.
        ret, std = compute_metrics(mu, Q, h_opt)
        # Store the results.
        returns.append(ret)
        std_devs.append(std)

    # Plot the efficient frontier.
    plt.figure(figsize=(10, 6))
    plt.plot(std_devs, returns, 'o-', markersize=4)
    plt.title('Efficient Frontier')
    plt.xlabel('Standard Deviation of Portfolio Return')
    plt.ylabel('Expected Portfolio Return')
    plt.grid(True)
    plt.show()
```

[ ]:
```
frontier()
```



The supplied `Midtermdata.zip` file contains an **R** data file `data.rda` and a tab-delimited text file `data.tsv` containing the Dow-Jones Industrial Index and the closing prices for the 30 companies in that index for 250 trading days. **You will use that data for questions 4, 5 and 6.**

Question 4. (15 pts) Write a program to: 1. read in this Dow-Jones data, 2. convert the matrix of daily prices to daily simple returns (not logarithmic returns), 3. annualize the returns by multiplying them by 252 (the typical number of trading days in a year), 4. move the index column out of the matrix and into a separate vector, 5. compute a covariance matrix `Qts` based on the time-series of returns, and 6. print out the first five rows and five columns of the covariance matrix.

Clearly describe all steps in your program with comments. List your program in your answer document. No points will be awarded unless the steps associated with each part of the question are clearly distinguished. Also, submit the source code file.

Step 1.

```
[ ]: def read_data(file_path: str) -> pd.DataFrame:
         """
         Reads the data from a CSV file into a pandas DataFrame, setting the correct␣
       ↪DateTime index
         and using the first row as column names.

         Parameters:
         - file_path (str): The path to the data file.

         Returns:
```

```
    - pd.DataFrame: A DataFrame with the correct dates as index and the stock␣
↪tickers as column names.
    """
    # Read the file.
    data = pd.read_csv(file_path)

    # Rename the first value to be 'Date'.
    data.iloc[0, 0] = 'Date'

    # Set the first row as the header.
    data.columns = data.iloc[0]

    # Drop the first row.
    data = data.drop(data.index[0])

    # Reset the index.
    data = data.reset_index(drop=True)

    # Convert numerical dates to string.
    data['Date'] = data['Date'].astype(str)

    # Remove the '.0' part (if it exists).
    data['Date'] = data['Date'].str.replace(r'\.0$', '', regex=True)

    # Convert string dates to DateTime objects.
    data['Date'] = pd.to_datetime(data['Date'], format='%Y%m%d')

    # Set the date column as the index.
    data.set_index('Date', inplace=True)

    # Ensure the data is numeric.
    for col in data.columns[0:]:
        data[col] = pd.to_numeric(data[col], errors='coerce')

    return data
```

Step 2.

```
[ ]: def calculate_simple_returns(prices: pd.DataFrame) -> pd.DataFrame:
         """
         Converts a matrix of daily prices to daily simple returns.

         Parameters:
         - prices (pd.DataFrame): A DataFrame with daily prices.

         Returns:
         - pd.DataFrame: A DataFrame with daily simple returns.
         """
```

```
    # Calculate daily simple returns.
    returns = prices.pct_change().dropna()  # Drop the first row as the return␣
↪for the first day is not defined.
    return returns
```

Step 3.

```
[ ]: def annualize_returns(returns: pd.DataFrame) -> pd.DataFrame:
         """
         Annualizes the daily simple returns.

         Parameters:
         - returns (pd.DataFrame): A DataFrame with daily simple returns.

         Returns:
         - pd.DataFrame: A DataFrame with annualized returns.
         """
         # Multiply by the number of trading days to annualize.
         annual_returns = returns * 252
         return annual_returns
```

Step 4.

```
[ ]: from typing import Tuple

     def extract_index(data: pd.DataFrame) -> Tuple[pd.DataFrame, pd.Series]:
         """
         Moves the index column out of the DataFrame into a separate vector.

         Parameters:
         - data (pd.DataFrame): A DataFrame with the index as one of the columns.

         Returns:
         - Tuple[pd.DataFrame, pd.Series]: A tuple of the DataFrame without the␣
     ↪index column and the index column as a Series.
         """
         # Assuming the index is the first column, we separate it from the data.
         index_column = data.iloc[:, 0]
         data_without_index = data.iloc[:, 1:]
         return data_without_index, index_column
```

Step 5.

```
[ ]: def compute_covariance_matrix(returns: pd.DataFrame) -> pd.DataFrame:
         """
         Computes the covariance matrix based on the time-series of returns.

         Parameters:
         - returns (pd.DataFrame): A DataFrame with annualized returns.
```

```
    Returns:
    - pd.DataFrame: The covariance matrix of the returns.
    """
    # Compute the covariance matrix.
    Qts = returns.cov()
    return Qts
```

Step 6.

[ ]:
```
# file_path = 'Midtermdata/data.tsv'
file_path = 'Midtermdata/data.csv'

# Step 1: Read in the data.
data = read_data(file_path)

# print(data.head())

# Step 2 and 3: Convert to simple returns and annualize.
daily_returns = calculate_simple_returns(data)
annual_returns = annualize_returns(daily_returns)

# print(annual_returns)

# Step 4: Move the index column out of the matrix.
annual_returns_modified = annual_returns.iloc[1:, 1:]

# Step 5: Compute the covariance matrix.
Qts = compute_covariance_matrix(annual_returns_modified)

# Step 6: Print the first five rows and columns of the covariance matrix.
print(Qts.iloc[:5, :5])
```

```
0          AAPL        AXP         BA        CAT       CSCO
0
AAPL  17.255424   5.189919   8.155240   7.535200   7.789729
AXP    5.189919  11.842255   3.717910   5.042402   3.480634
BA     8.155240   3.717910  11.618264   5.935137   6.465403
CAT    7.535200   5.042402   5.935137  15.649550   6.247492
CSCO   7.789729   3.480634   6.465403   6.247492  12.651489
```

Implementation in R The following section implements the same concepts as above but in R when it was difficult to parse the data in Python.

However, **please use the Python results** as I was able to fix/debug the issues.

```
library(tidyverse)
library(lubridate)
```

```r
# Function to read the data and set the first column as a DateTime index
read_data <- function(file_path) {
  # Read the data, assuming the first row is the header
  data <- read.csv(file_path, header = TRUE)

  # Convert the first column to Date format assuming it's in 'YYYYMMDD' format
  # Assuming the first column is named 'Date' or similar; adjust accordingly
  data[,1] <- ymd(data[,1])

  # Set the first column as row names if needed or keep as a separate Date column
  # Optionally, you can make the date column as row names (not always recommended for time ser
  # rownames(data) <- data[,1]
  # data <- data[,-1]

  return(data)
}

# Function to calculate simple returns
calculate_simple_returns <- function(prices) {
  returns <- prices / lag(prices) - 1
  returns <- na.omit(returns)  # Remove NA values resulted from lagging
  return(returns)
}

# Function to annualize returns
annualize_returns <- function(returns) {
  annual_returns <- returns * 252
  return(annual_returns)
}

# Function to compute the covariance matrix
compute_covariance_matrix <- function(returns) {
  Qts <- cov(returns)
  return(Qts)
}
```

Pre-Processing

```r
# Step 1: Read in the data
file_path <- "Midtermdata/data.csv"

# Read the CSV file, making sure the first row is used as column names.
data <- read.csv(file_path, header = TRUE, check.names = FALSE)

# Set the value in the first row, first column to "Date"
data[1, 1] <- "Date"

# Now, use the first row to set column names and remove the first row from the data
```

```r
colnames(data) <- as.character(unlist(data[1, ]))
data <- data[-1, ]

# Assuming 'data' is your dataframe and 'Date' is the column with dates in YYYYMMDD format
data$Date <- as.Date(as.character(data$Date), format="%Y%m%d")

# Assuming the first column is the date, and the rest are the prices
data[,-1] <- sapply(data[,-1], as.numeric)

head(data)

# dates <- data[[1]] # Extract the first column as dates
# prices <- data[-1] # Remove the first column from the data
#
# # Convert the first column to dates if it's not already
# dates <- as.Date(as.character(dates), format="%Y%m%d")

# Step 2: Calculate daily returns.
daily_returns <- data[, -1] / lag(data[, -1]) - 1
daily_returns <- daily_returns[-1, ]  # Remove the first row which will be NA.

print(head(daily_returns))

# Step 3: Annualize the daily returns.
annual_returns <- apply(daily_returns, 2, mean, na.rm = TRUE) * 252

print(annual_returns)

# Step 4: Extract the index column.
## Already did in pre-processing.

# Step 5: Compute the covariance matrix of daily returns.
Qts <- cov(daily_returns, use = "pairwise.complete.obs")

# Step 6: Print out the first five rows and columns of the covariance matrix.
print(Qts[1:5, 1:5])
```

Question 5. (20 pts) Given n securities $S_i, (i = 1, \dots, 30)$, the single index model for their securities' returns is given by

$$r_i = \alpha_i + \beta_i r_M + \epsilon_i$$

where r_M is the return of the Index (here the Dow Jones), ϵ_i is a random variable specific to Security S_i satisfies

$$\mathbb{E}[\epsilon_i] = 0, \; cov(\epsilon_i, r_M) = 0, \text{ if } i \neq j.$$

Under that model, we have the securities Expected Returns $\mu_i = \alpha_i + \beta_i \mu_M$ where μ_M is the Expected Return of the Index. The securities variance and covariances are specified by

$$\sigma_i^2 = \beta_i^2\sigma_M^2 + \sigma_{R_i}^2 \text{ and } \sigma_{ij} = \beta_i\beta_j\sigma_M^2 \text{ when } i \neq j$$

where σ_M^2 is the variance of the Index, and $\sigma_{R_i}^2$ is the variance of the residual ϵ_i for security S_i.

Write a program utilizing the Dow-Jones data that: 1. uses a loop to regress each company's returns onto the index return, 2. prints a table of intercepts, slopes (β_i), and idiosyncratic standard deviations σ_{R_i} (standard deviation of the residuals) for all companies $i = 1, \ldots, 30$, 3. computes and prints the variance of the index's return, 4. computes the single-index approximation to the covariance matrix, `Qsi` using your computed σ_M^2, β_i, and $\sigma_{R_i}^2$, for all i, and 5. prints the first five rows and columns of this covariance matrix.

Clearly describe all steps in your program with comments. List your program in your answer document. No points will be awarded unless the steps associated with each part of the question are clearly distinguished. Also, submit the source code file.

```
[ ]: import statsmodels.api as sm

     def perform_regression(index_returns: pd.Series, security_returns: pd.Series)␣
      ↪-> Tuple[float, float, float]:
         """
         Performs linear regression of security returns on index returns.

         Parameters:
         - index_returns (pd.Series): Daily returns of the index.
         - security_returns (pd.Series): Daily returns of a security.

         Returns:
         - Tuple[float, float, float]: Intercept (alpha), slope (beta), and standard␣
      ↪deviation of residuals (sigma_r).
         """
         X = sm.add_constant(index_returns)  # Add a constant to include intercept␣
      ↪in the model.
         model = sm.OLS(security_returns, X).fit()
         return model.params[0], model.params[1], model.resid.std()

     def analyze_data(file_path: str) -> None:
         """
         Analyzes Dow-Jones data to compute and print statistical measures.

         Performs regressions of each company's returns onto the index return,
         prints intercepts, slopes, idiosyncratic standard deviations, variance of␣
      ↪the index,
         and part of the covariance matrix.

         Parameters:
         - file_path (str): Path to the CSV file containing the Dow-Jones data.
         """
```

```
    data = read_data(file_path)
    # print(data.head())

    daily_returns = calculate_simple_returns(data)
    # print(daily_returns.head())

    index_returns = daily_returns['^DJI']
    # print(index_returns)

    results = pd.DataFrame(columns=['Intercept', 'Beta', 'Sigma_R'])

    # Loop through securities and regress each against the index.
    for column in daily_returns.columns[1:]:  # Skip Dow Jones column
        alpha, beta, sigma_r = perform_regression(index_returns,␣
↪daily_returns[column])
        results.loc[column] = [alpha, beta, sigma_r]

    print("Intercepts, Betas, and Idiosyncratic Standard Deviations:")
    print(results)

    sigma_M_squared = np.var(index_returns)
    print(f"\nVariance of the Index's Return: {sigma_M_squared}")

    ## Compute single-index approximation to the covariance matrix:

    # Create an empty DataFrame to store the covariance matrix Qsi.
    Qsi = pd.DataFrame(index=daily_returns.columns[1:], columns=daily_returns.
↪columns[1:])

    for i in range(len(Qsi)):
        for j in range(len(Qsi.columns)):
            if i == j:
                Qsi.iloc[i, j] = results['Beta'].iloc[i] ** 2 * sigma_M_squared␣
↪+ results['Sigma_R'].iloc[i] ** 2
            else:
                Qsi.iloc[i, j] = results['Beta'].iloc[i] * results['Beta'].
↪iloc[j] * sigma_M_squared

    print("\nSingle-Index Approximation Covariance Matrix (First 5 Rows and␣
↪Columns):")
    print(Qsi.iloc[:5, :5])

analyze_data(file_path)
```

```
Intercepts, Betas, and Idiosyncratic Standard Deviations:
      Intercept      Beta   Sigma_R
AAPL   0.000268  1.138782  0.012475
```

```
AXP   -0.000719  0.853526  0.011009
BA     0.000697  1.019695  0.009490
CAT   -0.001290  1.063530  0.012045
CSCO   0.000750  1.060068  0.009936
CVX   -0.001031  1.148306  0.012780
DD    -0.000181  1.020580  0.011313
DIS    0.000940  0.917125  0.010339
GE     0.000601  1.065219  0.009995
GS    -0.000071  1.165276  0.006910
HD     0.001003  1.015542  0.007833
IBM   -0.000517  1.050286  0.008685
INTC   0.000126  0.999549  0.012832
JNJ   -0.000219  0.854040  0.006560
JPM    0.000263  1.204165  0.007180
KO     0.000197  0.662824  0.006618
MCD    0.000802  0.930011  0.009012
MMM    0.000116  0.963949  0.006070
MRK   -0.000208  0.953009  0.010229
MSFT   0.000630  1.243621  0.013432
NKE    0.001375  0.982035  0.009613
PFE    0.000620  0.885675  0.008155
PG    -0.000416  0.766293  0.006506
TRV    0.000483  0.873620  0.006558
UNH    0.000886  1.031037  0.011808
UTX   -0.000240  0.977089  0.008357
V      0.001430  1.118423  0.010289
VZ    -0.000194  0.748745  0.007216
WMT   -0.001070  0.775067  0.010807
XOM   -0.000593  1.051269  0.009551

Variance of the Index's Return: 8.83749785920134e-05

Single-Index Approximation Covariance Matrix (First 5 Rows and Columns):
0         AAPL       AXP        BA       CAT      CSCO
0
AAPL   0.00027  0.000086  0.000103  0.000107  0.000107
AXP   0.000086  0.000186  0.000077   0.00008   0.00008
BA    0.000103  0.000077  0.000182  0.000096  0.000096
CAT   0.000107   0.00008  0.000096  0.000245    0.0001
CSCO  0.000107   0.00008  0.000096    0.0001  0.000198
```

Implementation in R Below is the implementation in R, which I advise you to **not consider** as that was before I could fix the Python bugs. **Please use the Python results.**

```r
# 0. Pre-process the Data: Already done, moving to regression

# Step 1. Conduct the regressional analysis
results <- list() # To store regression results
```

```
for(i in 3:ncol(data)){
  # Using backticks to handle special characters in column names
  formula <- as.formula(paste("`", colnames(data)[i], "` ~ `", colnames(data)[2], "`", sep = "
  reg <- lm(formula, data = data)
  alpha_i <- coef(reg)[1] # Intercept
  beta_i <- coef(reg)[2] # Slope
  sigma_R_i <- sd(residuals(reg)) # Standard deviation of residuals
  results[[colnames(data)[i]]] <- c(alpha_i, beta_i, sigma_R_i)
}

# Step 2. Creating a data frame from the results list for better visualization
results_df <- do.call(rbind, results)
colnames(results_df) <- c("Intercept", "Beta", "Sigma_R")
rownames(results_df) <- names(results)

print(results_df)

# Convert results_df into a data frame.
results_df <- as.data.frame(results_df)

# Step 2. Printing the results table
print(results_df)

# Step 3: Compute and print the variance of the index's return
sigma_M_squared <- var(data[,2]) # Assuming the 2nd column is the DJI
print(sigma_M_squared)

# Step 4. Compute the covariance matrix: Using formula sigma^2_M * Beta_i * Beta_j + delta_ij
## Here, delta_ij is the Kronecker delta, which is 1 if i = j and 0 otherwise
Qsi <- matrix(nrow=30, ncol=30)

for(i in 1:nrow(results_df)){
  for(j in 1:nrow(results_df)){
    if(i == j){
      # Diagonal elements represent the variance for each security
      Qsi[i,j] <- sigma_M_squared * (results_df$Beta[i]^2) + (results_df$Sigma_R[i]^2)
    } else {
      # Off-diagonal elements represent the covariances between securities
      Qsi[i,j] <- sigma_M_squared * results_df$Beta[i] * results_df$Beta[j]
    }
  }
}

# 5. Print the first five rows and columns of the covariance matrix.
print(Qsi[1:5, 1:5])
```

Question 6. (20 pts) With your function `port` from the Question 2:

1. Use the set of input parameters `mu` (mean vector μ), `Qts` (covariance matrix Q computed with a time series approximation), and `tau` (risk tolerance τ) to compute the efficient frontier corresponding to the maximization of utility function U defined by

$$U(h) = -\frac{1}{2}h^T Q h + \tau h^T \mu$$

subject to the constraints

$$0 \leq h_i \leq 0.1 \text{ for all } i, \text{ and}$$

$$\sum_{i=1}^{n} h^T e = 1$$

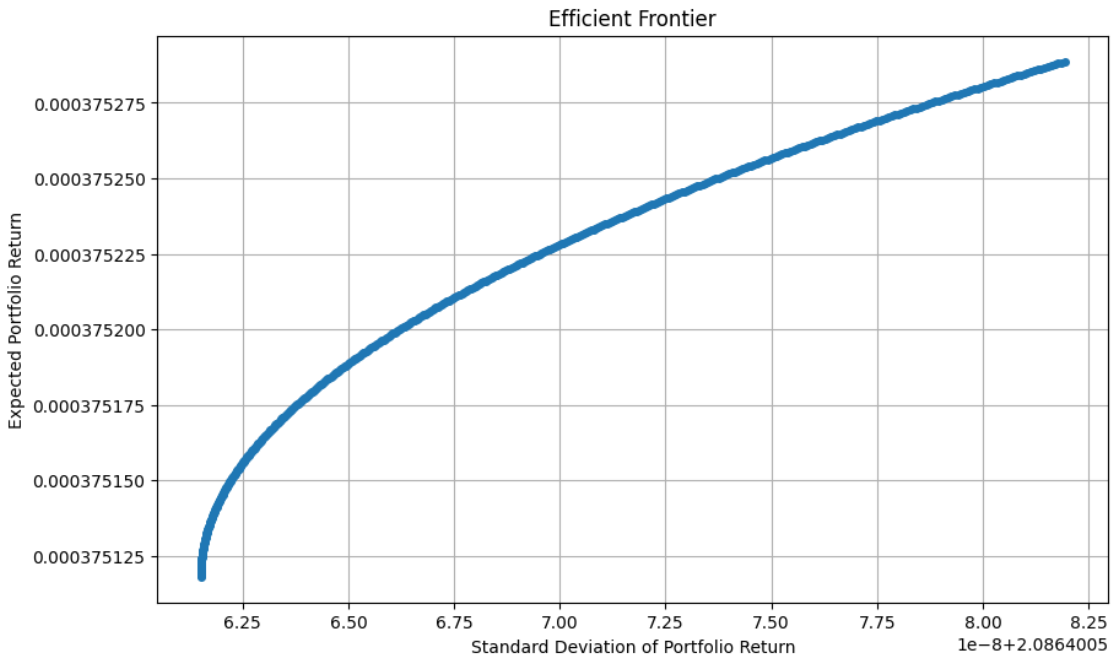
where n is the number of securities in the portfolio. Be sure to include all constraints; both equality and inequality constraints.
2. Repeat the computation of the efficient frontier but now use the approximate covariance matrix `Qsi` from the single index model instead of the time-series model `Qts`.
3. Plot the efficient frontier using the time-series approximation to `Qts` in blue, and superimpose the efficient frontier computed with the single index covariance model `Qsi` in red. Do you think the results are similar enough that the single-index covariance model is valid?

Clearly describe all steps in your program with comments. List your program in your answer document. No points will be awarded unless the steps associated with each part of the question are clearly distinguished. Also, submit the source code file.

Step 1.

```
[ ]: def frontier_qts():
         """
         Plots the efficient frontier by optimizing portfolios over a range of risk␣
     ↪tolerance values using the `port` function.
         Loads mean returns vector and covariance matrix, then calculates and plots␣
     ↪the efficient frontier.
         """
         # Load the mean returns vector and covariance matrix from saved files.
         mu, Q = load_inputs()
         TAU = np.arange(0, 0.501, 0.001)  # Range of risk tolerance values
         returns = []  # List to store expected returns of optimized portfolios
         std_devs = []  # List to store standard deviations of optimized portfolios

         for tau in TAU:
             # Use the previously defined port() function for portfolio optimization.
             h_opt = port(mu, Qts, tau)
             # Compute expected return and standard deviation for the optimized␣
     ↪portfolio.
             ret, std = compute_metrics(mu, Qts, h_opt)
             # Store the results.
             returns.append(ret)
             std_devs.append(std)
```

```
    # Plot the efficient frontier.
    plt.figure(figsize=(10, 6))
    plt.plot(std_devs, returns, 'o-', markersize=4)
    plt.title('Efficient Frontier')
    plt.xlabel('Standard Deviation of Portfolio Return')
    plt.ylabel('Expected Portfolio Return')
    plt.grid(True)
    plt.show()

frontier_qts()
```


Efficient Frontier
Expected Portfolio Return
Standard Deviation of Portfolio Return
1e−8+2.0864005