

Problem 1

Question:

Use the stock returns in DailyReturn.csv for this problem. DailyReturn.csv contains returns for 100 large US stocks and as well as the ETF, SPY, which tracks the S&P500.

Create a routine for calculating an exponentially weighted covariance matrix. If you have a package that calculates it for you, verify that it calculates the values you expect. This means you still have to implement it.

Use PCA and plot the cumulative variance explained by each eigenvalue for λ each chosen.

Solution:

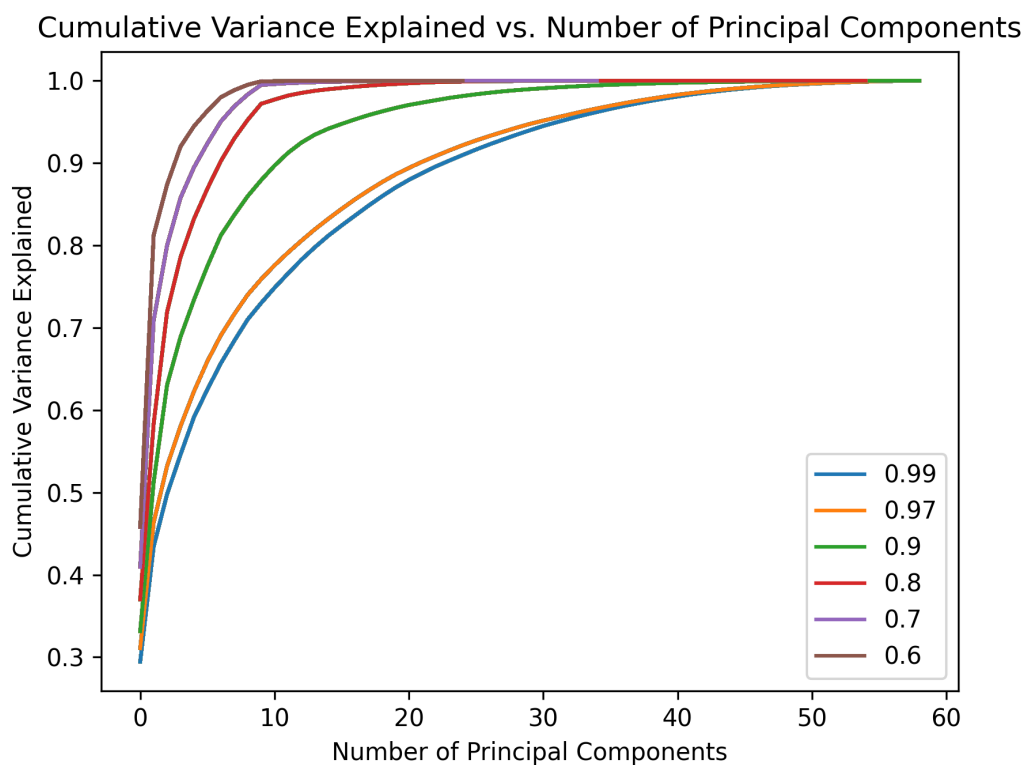
$$w_{t-i} = (1 - \lambda) \lambda^{i-1}$$

$$\widehat{w}_{t-i} = \frac{w_{t-i}}{\sum_{j=1}^n w_{t-j}}$$

To solve this problem, I first used the function ‘calculate_weight’ to calculate the weight in an infinite horizon. Since we don’t have an infinite horizon, I normalized the weights to 1 (adjust the sum of weights to 1) and got the adjusted weights. To calculate exponentially weighted covariance, I calculated the mean X and mean Y and got the weight covariance matrix by the function below.

$$\widehat{cov}(x, y) = \sum_{i=1}^n w_{t-i} (x_{t-i} - \bar{x}) (y_{t-i} - \bar{y})$$

In python, we can use ‘np.linalg.eig’ to get the eigenvalues and eigenvectors. I used simply PCA and removed all negative eigenvalues. I choose seven different lambdas to see the explained information through time. The results are in the following plot. We can notice that when lambda is smaller, the first eigenvalue is bigger, and the explained percent is smaller. This is because higher weight is added to the recent data.



Problem 2

Question:

Copy the `chol_psd()`, and `near_psd()` functions from the course repository –implement in your programming language of choice. These are core functions you will need throughout the remainder of the class. Implement Higham’s 2002 nearest psd correlation function.

Generate a non-psd correlation matrix that is 500x500. Use `near_psd()` and Higham’s method to fix the matrix. Confirm the matrix is now PSD.

Compare the results of both using the Frobenius Norm. Compare the run time between the two. How does the run time of each function compare as N increases? Based on the above, discuss the pros and cons of each method and when you would use each.

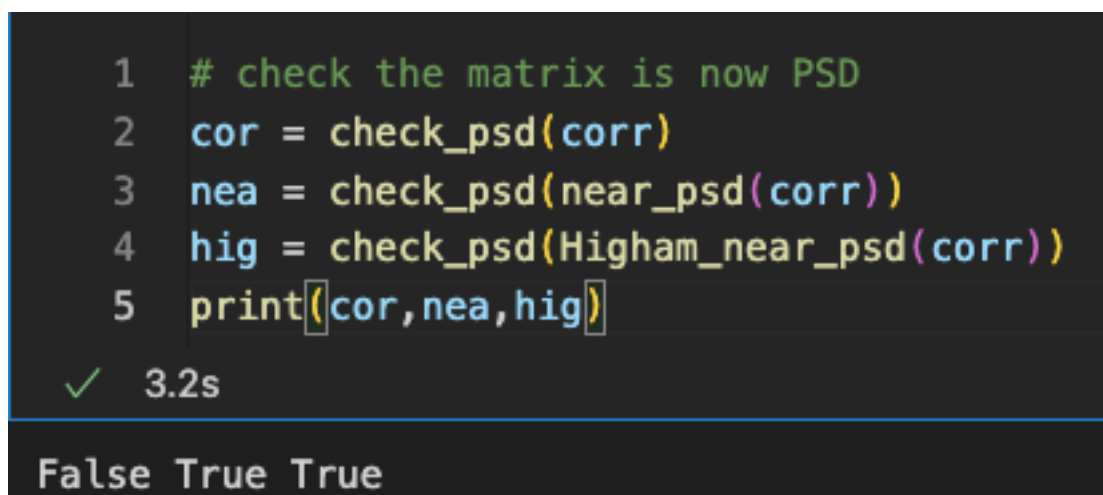
There is no wrong answer here, I want you to think through this and tell me what you think.

Solution:

First, I translated the `chol_psd()` and `near_psd()` functions in Python. Cholesky Factorization can be applied to positive definite matrices, and we should process our non-positive definite matrix. From Rebonato and Jackel, we can use `near_psd` to convert the non-PSD matrix into a near-PSD matrix. Higham is another to achieve it. Then, I wrote a `generate_nonpsd_matrix` function to generate a matrix by giving N. I set the $N=500$, $\text{corr}[0,1]=0.7357$, $\text{corr}[1,0]=0.7357$ and I got `corr` below.

```
array([[1.      , 0.7357, 0.9      , ..., 0.9      , 0.9      , 0.9      ],
       [0.7357, 1.      , 0.9      , ..., 0.9      , 0.9      , 0.9      ],
       [0.9      , 0.9      , 1.      , ..., 0.9      , 0.9      , 0.9      ],
       ...,
       [0.9      , 0.9      , 0.9      , ..., 1.      , 0.9      , 0.9      ],
       [0.9      , 0.9      , 0.9      , ..., 0.9      , 1.      , 0.9      ],
       [0.9      , 0.9      , 0.9      , ..., 0.9      , 0.9      , 1.      ]])
```

To ensure that the `near_psd()` function and `Higham_near_psd` truly convert non-PSD matrix into near PSD matrix, I wrote the function `check_psd(matrix)` to check it, and the results showed below. These two ways convert non-PSD matrix successfully.



```
1 # check the matrix is now PSD
2 cor = check_psd(corr)
3 nea = check_psd(near_psd(corr))
4 hig = check_psd(Higham_near_psd(corr))
5 print(cor,nea,hig)
```

✓ 3.2s

False True True

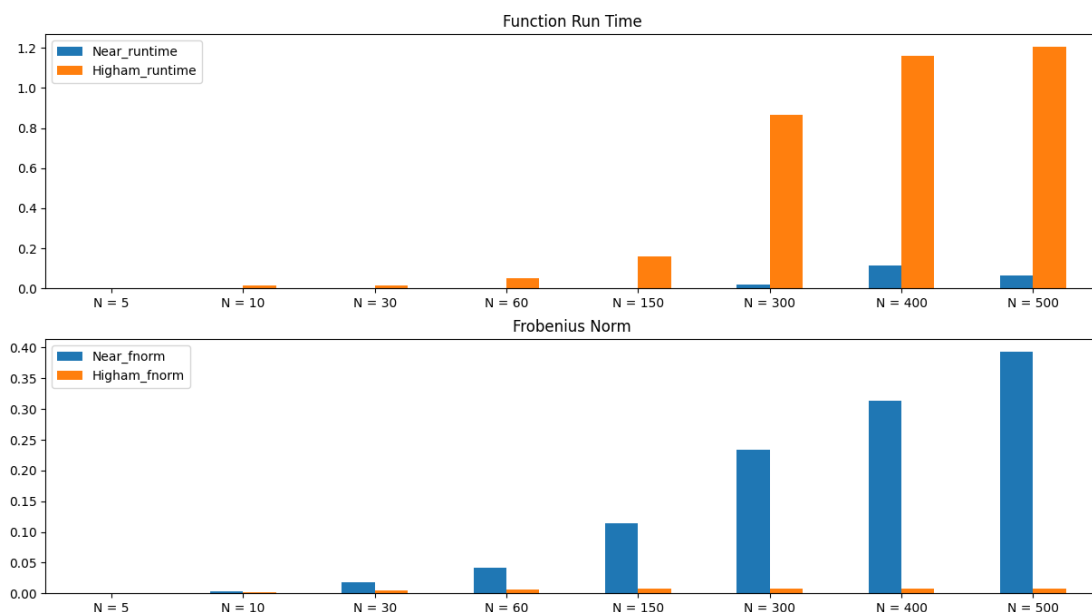
To compare the time spent in two different ways, I wrote a `neasure_runtime` function to calculate the time. The run time and Frobenius norm results of two ways

are stored in `runtime_data` and `norm_data`, shown below. The `runtime_data` and `norm_data` were also plotted using matplotlib in a single plot with two subplots.

The runtime data shows that `near_psd()` function runs fast in all N ranges. When N is small; these two ways run at a quick speed. `Higham_near_psd()` runs much slower than `near_psd()` when N is a significant number. When N increased, the runtime of the `Higham_near_psd()` function increased dramatically.

The `_fnorm` data shows how calculating the Frobenius norm by two different methods with different N . We noticed that when N is in a small number, the norms of `Near` and `Higham` are both at a low level and have less information loss. When N increases, `Near_fnorm` increases while `Higham_fnorm` keeps in a low range, we learn that `Higham` can be more accurate in the whole range of N , and `Near` loses more information when N gets more extensive.

As for me, if N is small, these two ways are almost the same; I can choose one of them randomly. But when N is significant, I prefer to use the `Higham` method. Though this method runs slower, accuracy is the most important thing.



Problem 3

Question:

Implement a multivariate normal simulation that allows for simulation directly from a

covariance matrix or using PCA with an optional parameter for % variance explained.

If you have a library

that can do these, you still need to implement it yourself for this homework and prove that it functions as expected.

Generate a correlation matrix and variance vector 2 ways:

1. Standard Pearson correlation/variance (you do not need to reimplement the `cor()` and `var()` functions).

2. Exponentially weighted $\lambda=0.97$

Combine these to form 4 different covariance matrices.

(Pearson correlation + `var()`), Pearson correlation + EW variance, etc.)

Simulate 25,000 draws from each covariance matrix using:

1. Direct Simulation
2. PCA with 100% explained.
3. PCA with 75% explained.
4. PCA with 50% explained.

Calculate the covariance of the simulated values. Compare the simulated covariance to it's input matrix using the Frobenius Norm (L2 norm, sum of the square of the difference between the matrices). Compare the run times for each simulation.

What can we say about the trade-offs between time to run and accuracy.

Solution:

To solve this problem, I first calculated the covariance of the simulated values. I did something similar to the second question to compare tradeoffs between time to run and accuracy. I calculate their run-time and Frobenius Norm. The results shown below.

Run time data shows that the direct method is slow, it uses the longest time compared to other methods. PCA has many advantages like reducing noisy in data and improve algorithm runtime. We can see from the plot that the more dimensions PCA reduces, the faster it will be.

Frobenius Norm shows that the direct method has almost no loss of information. While PCA reduces more dimensions, the more information it will lose, the less accurate it will be. As the opinion in the second

question, information accuracy is more critical, and it is worth it for us to spend a longer time.

