Problem 1

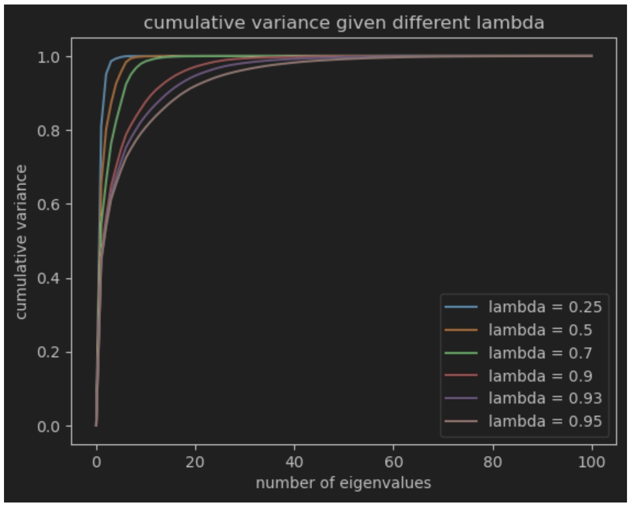
Use the stock returns in DailyReturn.csv for this problem. DailyReturn.csv contains returns for 100 large US stocks and as well as the ETF, SPY which tracks the S&P500.
Create a routine for calculating an exponentially weighted covariance matrix. If you have a package that calculates it for you, verify that it calculates the values you expect. This means you still have to implement it.
Vary $\lambda \in (0, 1)$. Use PCA and plot the cumulative variance explained by each eigenvalue for each λ chosen.
What does this tell us about values of λ and the effect it has on the covariance matrix?

Answer:

As obtained from the notes in class, the exponentially weight covariance between any two stocks is:

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

Thus, I performed expo_weighted_cov() on the given stock return, with lambda used to assign different weight to stock returns at different time spots. And then we performed PCA to calculate the cumulative variance explained by eigenvalues. And the graph is shown below, the higher cumulative variance, the higher percentage explained by current given number of eigenvalues, and each line represents a condition of a chosen lambda.



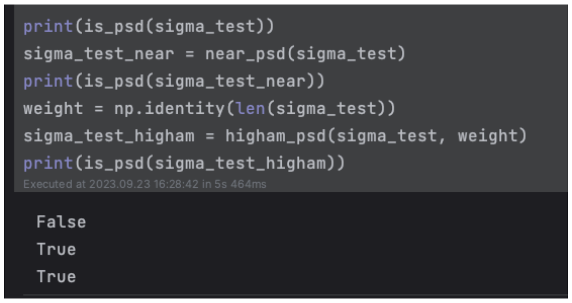
As can be obtained from the graph, all lines increase from 0 to 1, the difference lies in the increasing rate. When lambda is smaller, a sharper increase from 0 to 1 can be witnessed; and as lambda grows, the increase would be more gradual. This means when lambda is small, more importance (weight) will be attached to the most recent stock returns, and therefore, earlier data are less significant; and when lambda grows, the weights are more likely to be evenly distributed. This indicates smaller lambda allows more recent data to contain more information concentrating in one place, and thus, we will need fewer principal components to capture the variance of original data.

Problem 2

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.

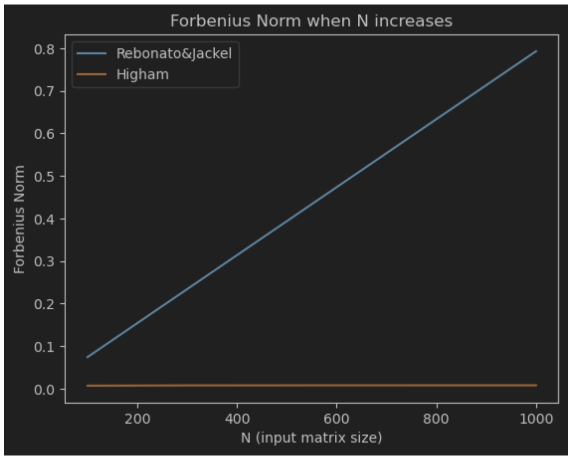
Answer:
For chol_psd(), near_psd(), and higham_psd() implementation and test, we created a non-psd correlation matrix that is 500*500 as instructed. The results are shown below:



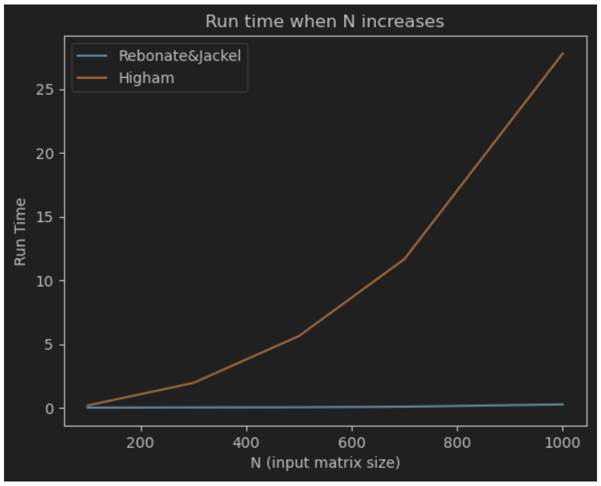
```
print(is_psd(sigma_test))
sigma_test_near = near_psd(sigma_test)
print(is_psd(sigma_test_near))
weight = np.identity(len(sigma_test))
sigma_test_higham = higham_psd(sigma_test, weight)
print(is_psd(sigma_test_higham))
Executed at 2023.09.23 16:28:42 in 5s 464ms

False
True
True
```

sigma_test is the input non-psd correlation matrix, and our is_psd() has successfully identified its non-psd characteristic. As for sigma_test_near obtained through near_psd(), and sigma_test_higham obtained through higham_psd(), our is_psd() also shows they have been successfully transformed into psd matrix. As for the comparison of Forbenius Norm of near_psd() (Rebonato&Jackel's method) and higham_psd() (Higham's method), we decided to increase the input non-psd matrix size to see how they perform along such increase.



As can be obtained from the graph, the Forbenius Norm for R&J is always higher than that of Higham, and as the size of input matrix increases, such difference widens. This means Higham's error is always and much smaller than that of R&J, thus indicating a higher accuracy for Higham. This is rather sensible, as when we implement higham_psd(), minimizing Forbenius Norm is indeed a part of that.
Also, for the comparison of run time of near_psd() and higham_psd(), we would adopt similar approach.



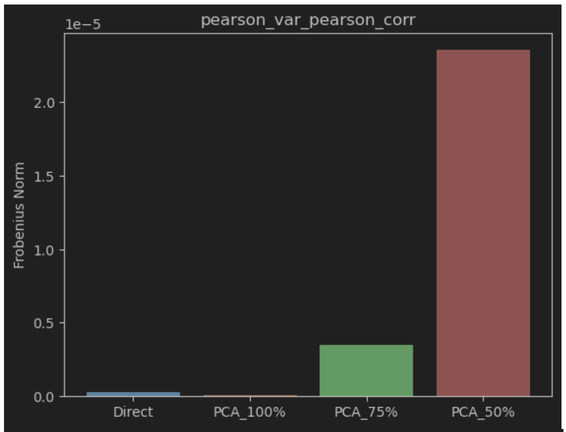
As can be obtained from the graph, run time for R&J method is always shorter than that of Higham's method, and as we increase the size of the input matrix, such difference widens. This is also sensible, as in Higham's method we adopted an iterative approach to gradually eliminate elements that make matrix non-psd, and in R&J's method, negative elements are just simply set to 0.
Therefore, the advantage of near_psd (R&J) is that it takes less time, but it leads to higher error; the advantage of higham_psd (Higham) is that it leads to smaller error and more accurate, but it takes more time than near_psd, and the larger the input matrix size, the slower it will be. In a practical situation, if the time is limited, or input matrix size is very large, or we don't care too much about the most similar matrix (smallest error possible), near_psd() is a better option; while when we have enough time, or input matrix is relatively small, or the similarity with the input matrix is crucial (smallest error possible), we should use higham_psd().

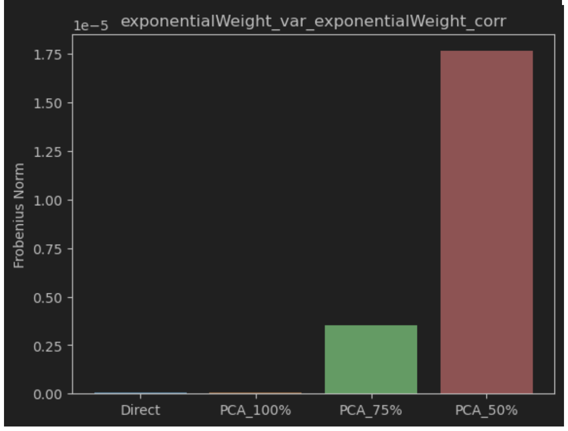
Problem 3

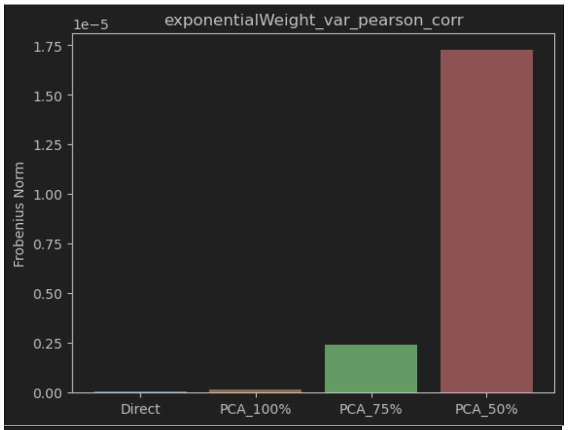
Using DailyReturn.csv.
Implement a multivariate normal simulation that allows for simulation directly from a covariance matrix or using PCA with an optional parameter for % variance explained. If you have a library that can do these, you still need to implement it yourself for this homework and prove 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.

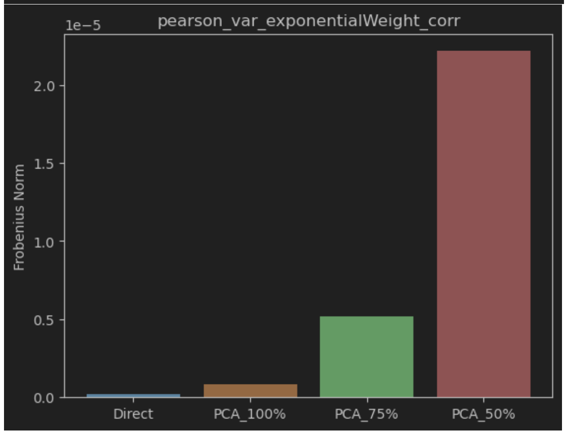
Answer:

First, we constructed two functions for multivariate normal simulation, sim_from_cov() for simulation using Cholesky factorization directly, and sim_from_pca() for simulation using PCA.
Next, we implemented 4 covariance matrix, of which ew_var_ew_cor is the one using exponentially weighted method to get variance vector and exponentially weighted method to get correlation matrix, ps_var_ps_cor is the one using Pearson method to get variance vector and Pearson method to get correlation matrix, ew_vars_ps_cor is the one using exponentially weighted method to get variance vector and Pearson method to get correlation matrix, ps_var_ew_cor is the one using Pearson method to get variance vector and exponentially weighted method to get correlation matrix. For exponentially weighted methods, lambda was set to be 0.97.
The simulation was performed for direct simulation, PCA with 100% explained, PCA with 75% explained, PCA with 50% explained.
The comparisons of Forbenius Norms for 4 methods with 4 covariance matrices are shown below. As can be obtained from the graphs, for all covariance matrices, direct and PCA with 100% explained held the lowest error and highest accuracy, and as the percent explained decrease, PCA is becoming less accurate. This is rather sensible, as percent explained decreases, less features are incorporated into the PCA model, thus less accurate.

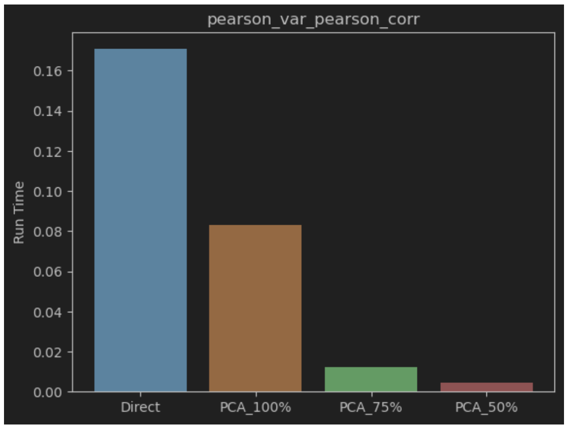


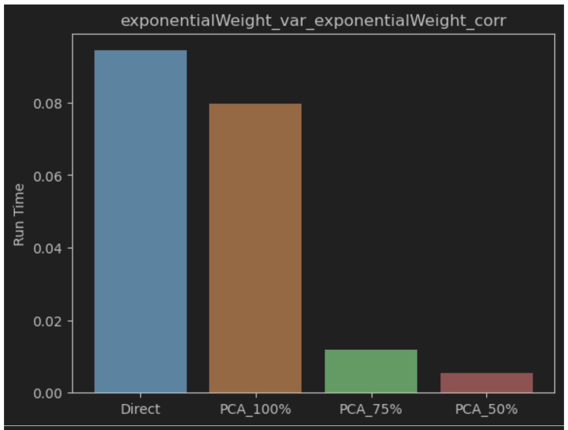


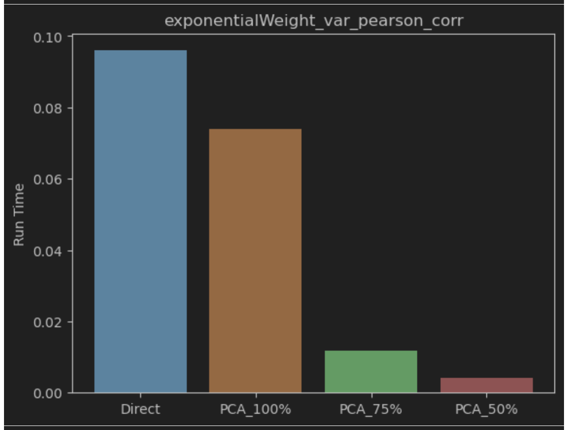


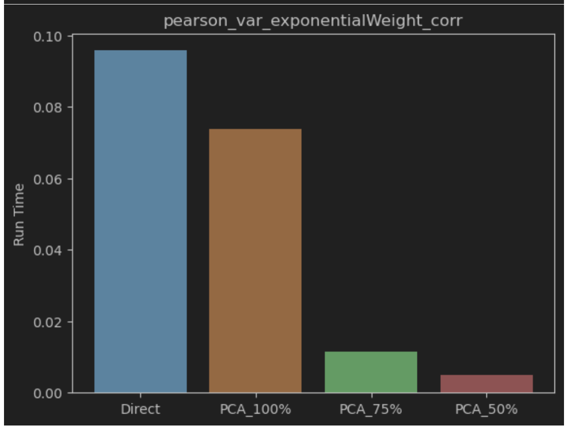


The comparisons of run time for 4 methods with 4 covariance matrices are shown below. As can be obtained from the graphs, for all covariance matrices, direct method is the slowest one, and PCA with 100% explained is on average slightly better than that. As percent explained decreases, run time drops significantly, with PCA with 50% explained being the fastest in every case.









Thus, we would conclude that for PCA with 100% explained and direct simulation, PCA with 100% explained is a better option, as both methods have very low error and are similarly accurate, while PCA with 100% explained is faster. However, we should be more careful with the tradeoff when we choose between PCA with 100% explained, 75% explained, and 50% explained, as when we decrease the percent explained, we would get the result faster, but at the same time the result itself is more inaccurate.