Ahmed Ibrahim

Week 03 solutions pdf

Codes to all the problems are included in the python (ipynb) file in github

Problem 1

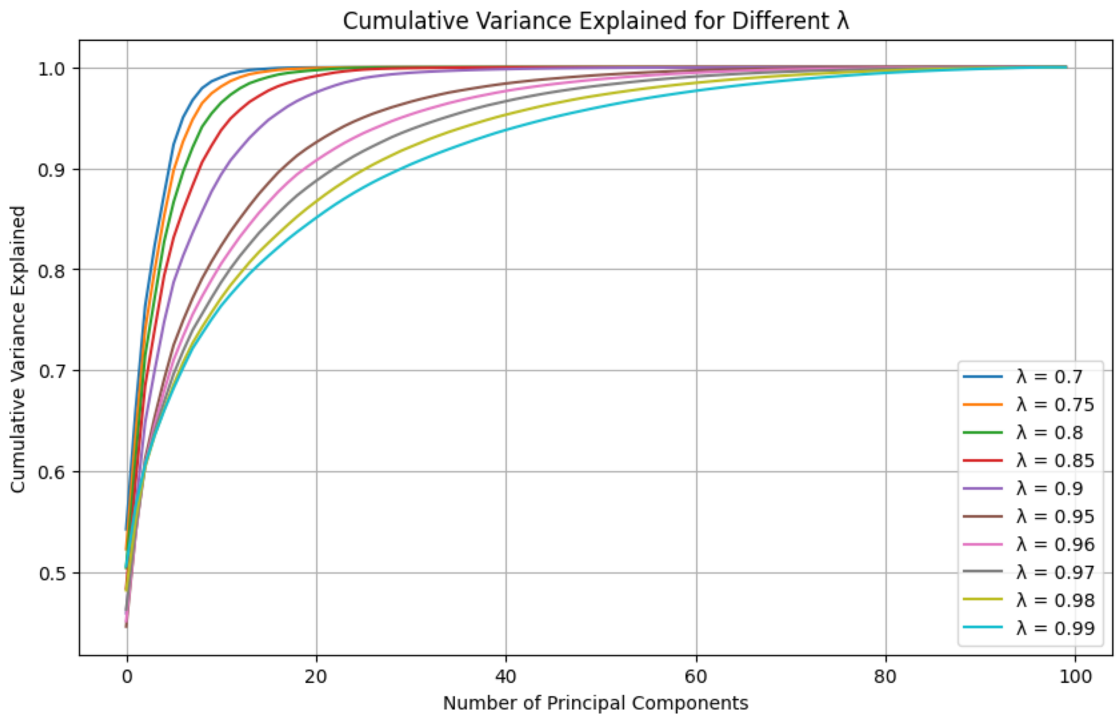
Use the stock returns in DailyReturn.csv for this problem. DailyReturn.csv contains returns for 100 large US stocks and as well as the ETF, SPY which tracks the S&P500.

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 λchosen.

What does this tell us about values of λ and the effect it has on the covariance matrix?

I approached this problem by initializing a zero covariance matrix, and iterated through each asset to calculate series of equal weights, mean of the securities and their weighted covariance.Then, I calculated PCA on those weights, and calculated the percentage explained by each eigen values. I repeated the process for a variety of 10 lambda values ranging from 0.7 to 0.99. The plots below shown are for various lambdas.



From the graph, we can see that for smaller lambdas, each eigenvalue explains as little variance compared to larger lambda values. As a result, when lambda is small, the covariance is explained by a small number of eigen values, and therefore, the cumulative variance explained reaches to 1.0 within a few values. However, for large values, the covariance gets explained by more eigenvalues and the model would need more number of Principal components to explain the cumulative variance.

The covariance matrix is used to calculate eigenvalues and its corresponding eigenvectors. By varying the lambda, we are changing the number of eigenvalues and vectors needed to explain the data, which changes the shape of the graph.

In the context of exponentially weighted covariance matrices, changing the value of λ corresponds to adjusting the level of importance given to recent observations versus older ones. When λ is close to 1, it means that the model is assigning a high weight to recent observations, effectively considering them more important. On the other hand, when λ is close to 0, older observations are given relatively higher importance.

1. λ close to 1 (High Weight on Recent Observations):
   - In this case, the model is giving more importance to recent observations. This means that the covariance matrix is more sensitive to recent changes in the data.
   - As a result, the first few eigenvalues will tend to capture a larger portion of the total variance in the data. This is reflected in the plot by a steeper increase in cumulative variance explained for smaller eigenvalues
2. λ close to 0 (More Uniform Weighting of Observations):
   - When λ is very close to 0, the model gives almost equal importance to all observations, regardless of how old they are.
   - As a result, the covariance matrix is less sensitive to short-term fluctuations and is more stable over time.

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. You can use the code I used in class:

Use near_psd() and Higham's method to fix the matrix. Confirm the matrix is now PSD.

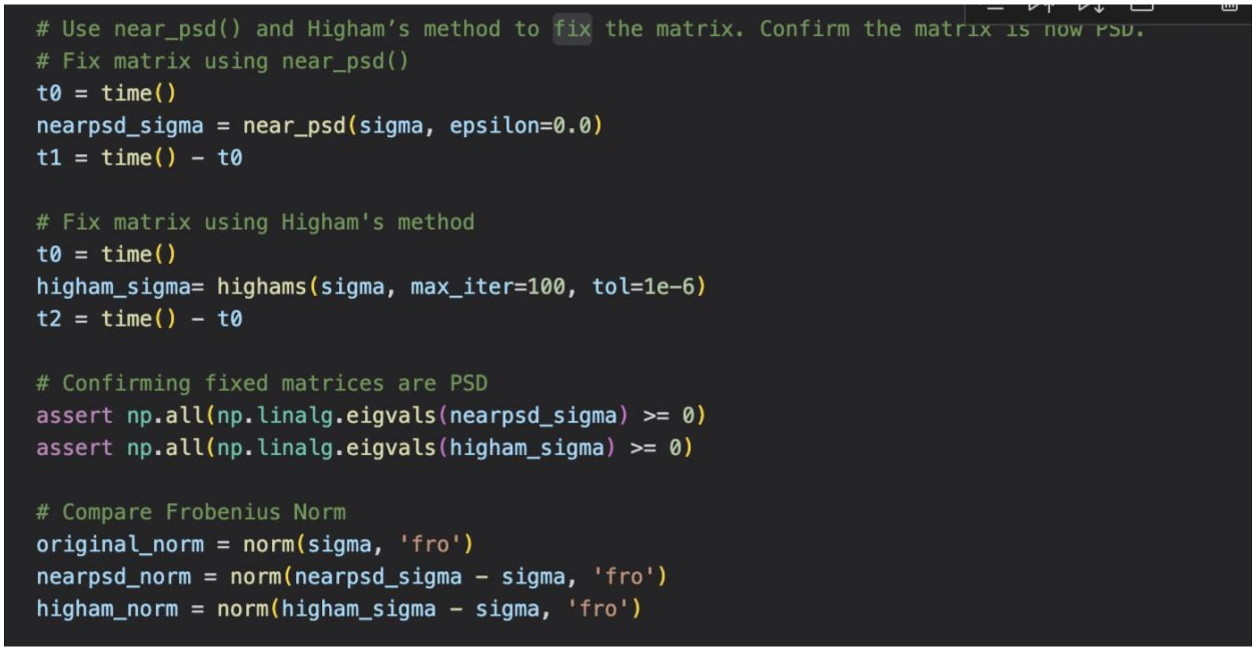
Compare the results of both using the Frobenius Norm. Compare the run time between the two. How does the run time of each function compare as N increases?

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.

N	Accuracy(Norm)		Runtime	
50	Near PSD	Higham	Near PSD	Higham
50	0.1857	0.0578	0.0043s	0.1203s
100	0.2728	0.0610	0.0292s	1.2482s
500	0.6275	0.0636	0.1169s	10.2910s
1000	0.8905	0.0640	0.3709s	35.8140s

For problem 2, I created functions for Cholesky, near PSD and Higham. I created a non psd correlation matrix and used used near_psd and higham method to fix the matrix as shown below. Checked the eigenvalues to make sure they are non negative.
Afterwards, calculated the Frobenius norm to check the difference between near PSD and higham and compared the result with the original norms. We have also calculated the runtime of both the functions shown in the table above.

From the table above, we can see that the run time for both functions increase significantly as N increases. The run time definitely increases a lot more for the Higham function compared to the Near PSD suggesting that the near PSD function is more efficient. However, by looking at the norm values, the Higham is more accurate and the values remain in closer proximity as N increases. This suggests that Higham is more stable than Near PSD.



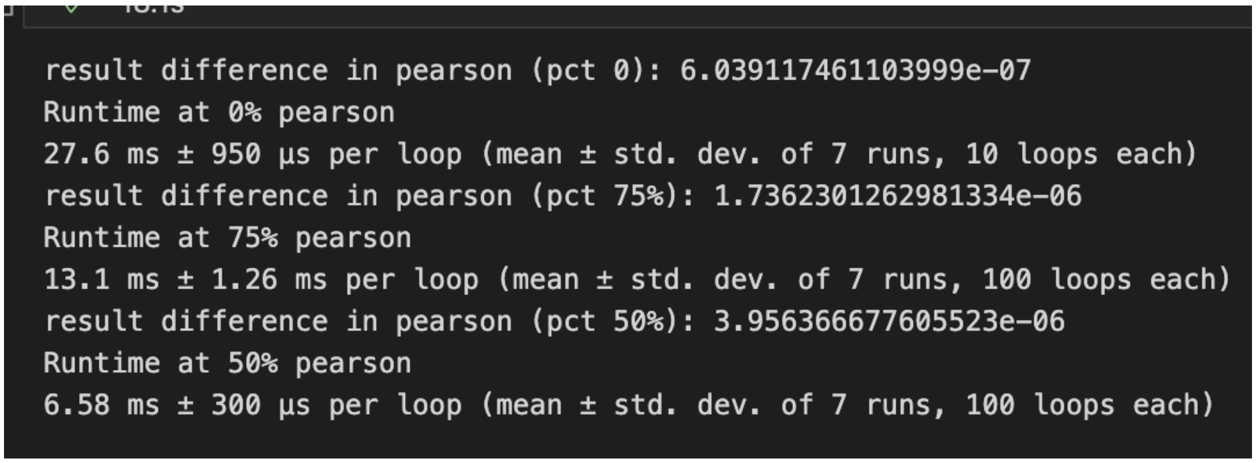
```
# Use near_psd() and Higham’s method to fix the matrix. Confirm the matrix is now PSD.
# Fix matrix using near_psd()
t0 = time()
nearpsd_sigma = near_psd(sigma, epsilon=0.0)
t1 = time() - t0

# Fix matrix using Higham's method
t0 = time()
higham_sigma= highams(sigma, max_iter=100, tol=1e-6)
t2 = time() - t0

# Confirming fixed matrices are PSD
assert np.all(np.linalg.eigvals(nearpsd_sigma) >= 0)
assert np.all(np.linalg.eigvals(higham_sigma) >= 0)

# Compare Frobenius Norm
original_norm = norm(sigma, 'fro')
nearpsd_norm = norm(nearpsd_sigma - sigma, 'fro')
higham_norm = norm(higham_sigma - sigma, 'fro')
```

Problem 3

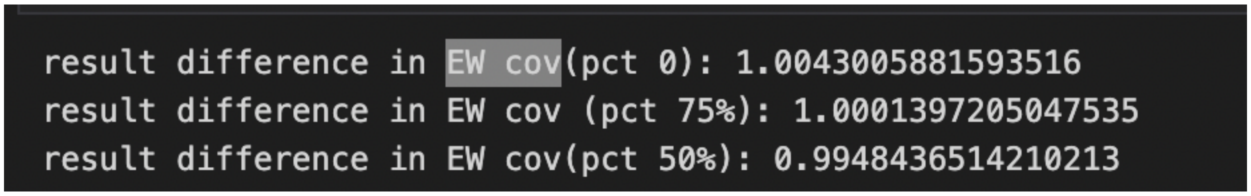
For problem 3, I implemented four covariances:

1. Pearson Covariance
2. (Pearson Correlation * EW Variance)
3. Pear + EW Cov
4. EW var + EW Corr

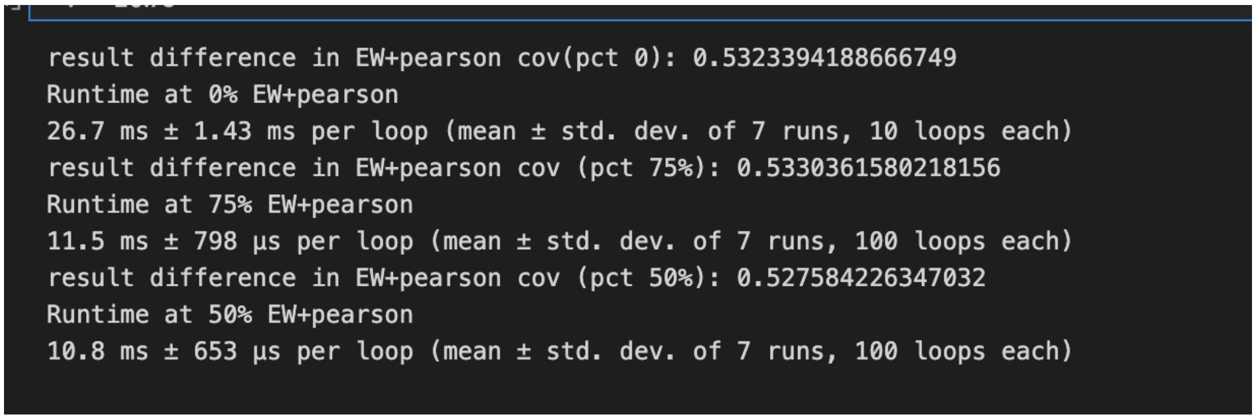
Below are the runtime and result difference for percent simulation 0, 75% and 50%. The codes are attached for detailed analysis.



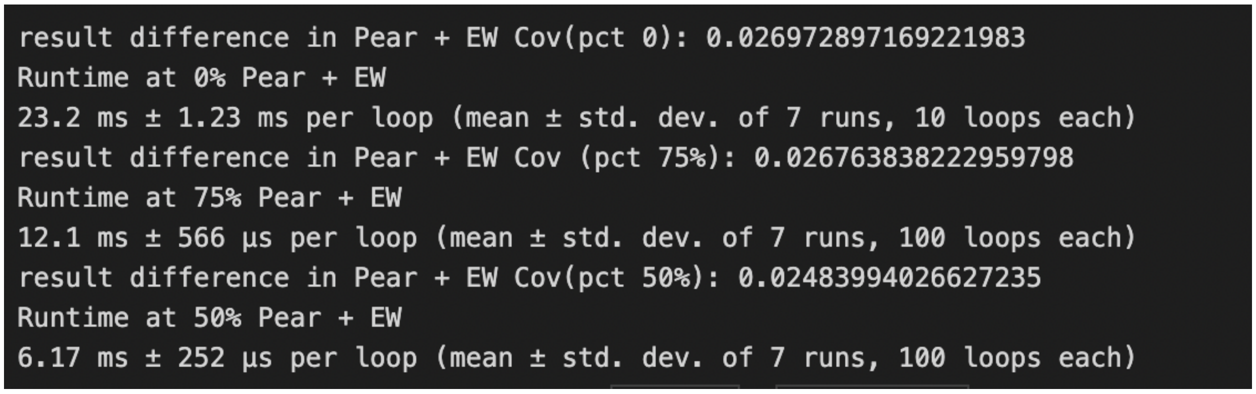
```
result difference in pearson (pct 0): 6.039117461103999e-07
Runtime at 0% pearson
27.6 ms ± 950 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
result difference in pearson (pct 75%): 1.7362301262981334e-06
Runtime at 75% pearson
13.1 ms ± 1.26 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
result difference in pearson (pct 50%): 3.956366677605523e-06
Runtime at 50% pearson
6.58 ms ± 300 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```



```
result difference in EW cov(pct 0): 1.0043005881593516
result difference in EW cov (pct 75%): 1.0001397205047535
result difference in EW cov(pct 50%): 0.9948436514210213
```



```
result difference in EW+pearson cov(pct 0): 0.5323394188666749
Runtime at 0% EW+pearson
26.7 ms ± 1.43 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
result difference in EW+pearson cov (pct 75%): 0.5330361580218156
Runtime at 75% EW+pearson
11.5 ms ± 798 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
result difference in EW+pearson cov (pct 50%): 0.527584226347032
Runtime at 50% EW+pearson
10.8 ms ± 653 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```



```
result difference in Pear + EW Cov(pct 0): 0.026972897169221983
Runtime at 0% Pear + EW
23.2 ms ± 1.23 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
result difference in Pear + EW Cov (pct 75%): 0.026763838222959798
Runtime at 75% Pear + EW
12.1 ms ± 566 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
result difference in Pear + EW Cov(pct 50%): 0.02483994026627235
Runtime at 50% Pear + EW
6.17 ms ± 252 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

We noticed that as PCT explains drops, the run time shortens and when PCT increases, the run time also increase. The result difference show that at higher PCT values, the result differences decrease, suggesting that higher PCT values provide more accurate results. However, it is worth noting that the result difference does not significant drop with PCT explains suggesting it might not always be worth it to go at high PCT level given it requires more computing speed.