Problem 1

Use the data in problem1.csv. Fit a Normal Distribution and a Generalized T distribution to this data. Calculate the VaR and ES for both fitted distributions.

Overlay the graphs the distribution PDFs, VaR, and ES values. What do you notice? Explain the differences.

I first defined a function called "cal_ES" which implements the process to calculate VaR and ES for a given set of data as included in "week05.jl". Then, I read given csv file, and fitted data into a normal distribution and a t distribution respectively. With fitted parameters, I derived VaRs and ran simulations for both distributions; eventually, I called the defined function to calculate ES for both normal distribution and t distribution. Results are below (note that I also called functions "RiskFunctions.VaR_Hist" and "RiskFunctions.CVaR_Hist" defined in the package riskfolio to directly calculate VaR and ES for given data to check my answers above):

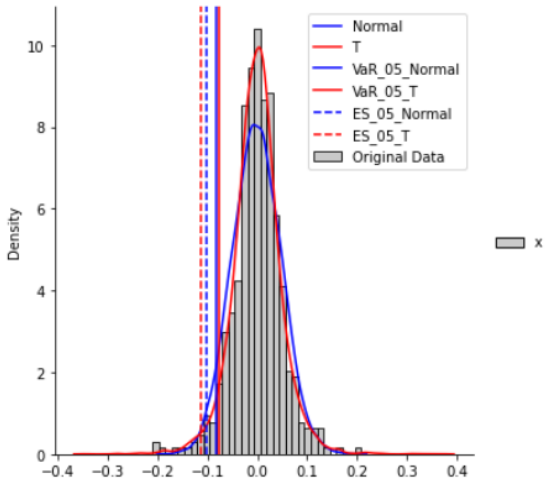
```
# print out results
print("VaR_05_norm {:.4f} vs. VaR_05_t {:.4f} vs. VaR_05_pack {:.4f}".format(VaR_05_norm, VaR_05_t, VaR_05_pack))
print("ES_05_norm  {:.4f} vs. ES_05_t {:.4f} vs. ES_05_pack {:.4f}".format(ES_05_norm, ES_05_t, ES_05_pack))
```

```
VaR_05_norm 0.0813 vs. VaR_05_t 0.0765 vs. VaR_05_pack 0.0782
ES_05_norm  0.1011 vs. ES_05_t 0.1120 vs. ES_05_pack 0.1168
```

As shown above, VaR and ES derived under normal distribution and t distribution are quite close to each other. Then, I plotted original data, both distributions, and values of VaR and ES under two different distributions into one graph:

```
# plot distributions and ES and VaR
plt.figure()
# plot original data
sns.displot(data, stat='density', palette=('Greys'), label='Original Data')
# plot simulation
sns.kdeplot(sim_norm, color="b", label='Normal')
sns.kdeplot(sim_t, color="r", label='T')
# plot ES and VaR onto the graph
plt.axvline(x=-VaR_05_norm, color='b', label='VaR_05_Normal')
plt.axvline(x=-VaR_05_t, color='r', label='VaR_05_T')
plt.axvline(x=-ES_05_norm, color='b', label='ES_05_Normal', linestyle="dashed")
plt.axvline(x=-ES_05_t, color='r', label='ES_05_T', linestyle="dashed")
plt.legend()
plt.show()
```

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



As shown, since the number of data has been greater than 30, in accordance with the

central limit theorem, t distribution has been like the shape of normal distribution, but, in this case, t distribution fits original data better than normal distribution. In the graph, VaR_05_Normal is on the left of VaR_05_T, indicating that on a 5% bad day, loss under normal distribution would be greater than loss under t distribution. This could be due to nuances between shapes/distributions of normal distribution and t distribution – the critical value of 5% of the normal distribution is smaller than the critical value of 5% of the t distribution in this case. However, as shown in the graph, ES_05_Normal is on the right of ES_05_T, demonstrating that expected loss of given the loss beyond VaR is higher under t distribution. As mentioned in class, expected shortfall, numerically, is the mean of all values less than or equal to VaR. Despite nuances between values of VaRs under two distributions, t distribution has fatter tails than normal distributions, and more extremely small values exist in t distribution, leading the mean of all values, the expected shortfall therefore, to be smaller. Consequently, ES_05_T is smaller than/on the left of ES_05_Normal.

Problem 2

In your main repository, create a Library for risk management. Create modules, classes, packages, etc as you see fit. Include all the functionality we have discussed so far in class. Make sure it includes

1. Covariance estimation techniques.
2. Non PSD fixes for correlation matrices
3. Simulation Methods
4. VaR calculation methods (all discussed)
5. ES calculation

Create a test suite and show that each function performs as expected.

I wrote a python file called "functionlib.py" which contains almost every important function that has been wrote before and would be used for risk management in the future. For this problem, I wrote test cases for each function to test if they are working appropriately or not.

For covariance estimation techniques, I wrote two functions – one is for calculation of exponential weights, and the other is for calculation of exponentially weighted covariance matrix:

```
import functionlib
import pandas as pd

# tests for written library

# read data
data = pd.read_csv("DailyPrices.csv")
dates = data.iloc[:,0]

# test for covariance matrix
# set up lists of weights, cumulative weights, and set up lamda equal to 0.5
weights = []
cum_weights = []
lamda = 0.5

# call both defined functions to get the result
functionlib.populateWeights(data.iloc[:,0],weights, cum_weights, lamda)
print("Length of exponential weights for lamda being 0.5:", len(weights))

# reverse w so that it corresponds to the ascending order of dates used in the DailyReturn.csv
rev_weights = weights[::-1]
covariance_matrix = functionlib.exwCovMat(data, dates, rev_weights)
print("Shape of exponentially weighted covariance matrix is:", covariance_matrix.shape)
```

```
Length of exponential weights for lamda being 0.5: 249
Shape of exponentially weighted covariance matrix is: (100, 100)
```

As shown above, shapes of exponential weights vector and exponentially weighted covariance are the same as expected, and these two functions used for calculation of covariance matrices are tested, therefore.

For fixations of non-psd matrices, I wrote several functions, including algorithms for Cholesky factorization for both positive semi-definite and positive definite matrices, near_PSD method, and Higham method. I use [[1, 0.7357, 0.9], [0.7357, 1, 0.9], [0.9, 0.9, 1]] as an example of positive semi-definite matrix and use [[2, 1], [1, 2]] as an example of positive definite to test my functions:

```
# test for fixations of matrices
# set up positive semi-definite matrix, positive definite matrix, and root matrix
# generate a N * N non-psd correlation matrix
N = 3
psd = np.full((N, N), 0.9)
for i in range(N):
    psd[i][i] = 1.0
psd[0][1] = 0.7357
psd[1][0] = 0.7357
pd = np.array([[2, 1], [1, 2]])
root = np.array([[0, 0], [0, 0]])

# call defined functions and print results
print(functionlib.chol_psd_forpsd(root,psd))
print(functionlib.chol_psd_forpd(root,pd))
print(functionlib.near_PSD(psd, epsilon=0.0))
print(functionlib.Higham(psd, tolerance=1e-8))
```

```
[[1.         0.         0.        ]
 [0.7357     1.         0.        ]
 [0.9        0.9        0.43588989]]
[[1.41421356 0.        ]
 [0.70710678 1.41421356]]
[[1.     0.7357 0.9   ]
 [0.7357 1.     0.9   ]
 [0.9    0.9    1.    ]]
[[1.     0.7357 0.9   ]
 [0.7357 1.     0.9   ]
 [0.9    0.9    1.    ]]
```

Testing whether matrices resulting from near_PSD and Higham are positive definite or not (through checking their eigenvalues):

```
# check eigenvalues of resulting matrices from near_PSD and Higham
mat1 = functionlib.near_PSD(psd, epsilon=0.0)
val1, vec1 = eigh(mat1)
print(val1)
mat2 = functionlib.Higham(psd, tolerance=1e-8)
val2, vec2 = eigh(mat2)
print(val2)
```

```
[0.04296751 0.2643     2.69273249]
[0.04296751 0.2643     2.69273249]
```

As shown above, eigenvalues of resulting matrices from near_PSD and Higham are all positive, proving that these two methods are working well to fix matrices.

For simulation methods, I wrote two functions – one is using PCA to simulate, and the other one uses direct simulation:

```
# test for simulation methods
# use the covariance matrix derived previously to test PCA simulation and direct simulation
result_PCA = functionlib.simulate_PCA(covariance_matrix, 10000, percent_explained=1)
result_direct = functionlib.direct_simulate(covariance_matrix, 10000)
print(result_PCA.shape)
print(result_direct.shape)
```

```
(10000, 100)
(10000, 100)
```

As shown above in the screenshot, both simulations generate simulated matrices with shapes we want, proving that they are performing as what they are expected.

The next part of functions in my function library are for calculation of VaR. The first function in this part is to compute VaR with assumption of certain distribution of given data:

```
# test for VaR calculations
# calculate return based on given prices first
log_returns = functionlib.return_cal(data,method="log",datecol="Date")
# demean log_returns
log_returns_aapl = log_returns.loc[:,"AAPL"]
log_returns_aapl = log_returns_aapl - log_returns_aapl.mean()
VaR_05_norm = functionlib.VaR_bas_dist(log_returns_aapl, alpha=0.05, dist="normal", n=10000)
VaR_05_t = functionlib.VaR_bas_dist(log_returns_aapl, alpha=0.05, dist="t", n=10000)
VaR_05_ar1 = functionlib.VaR_bas_dist(log_returns_aapl, alpha=0.05, dist="ar1", n=10000)
print("VaR_05_norm: {:.4f}; VaR_05_t: {:.4f}; VaR_05_ar1: {:.4f}".format(VaR_05_norm, VaR_05_t, VaR_05_ar1))
```

```
/home/jovyan/work/week5/functionlib.py:425: PerformanceWarning: DataFrame is highly fragmented.  This is usuall
which has poor performance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-
  out[vars[i]] = returns.iloc[:, i]
VaR_05_norm: 0.0368; VaR_05_t: 0.0364; VaR_05_ar1: 0.0369
```

As shown above, VaR assuming normal distribution, t distribution, and AR(1) model could be calculated from calling this function. Then, I am going to test functions for calculations of VaR for portfolios, and I assume that now I have a portfolio which includes one share for each stock included in the "DailyPrices.csv":

```
# set up portfolio holdings
hold_val = np.empty(len(data.columns[1:]))
hold_val.fill(1)
portfolio = {"Stock": data.columns[1:], "Holdings": hold_val}
portfolio = pd.DataFrame(portfolio)
current_p = data.iloc[-1,1:]
# call defined functions to test
del_norm = functionlib.del_norm_VaR(current_p, portfolio, log_returns, lamda=0.94, alpha=0.05)
his_VaR = functionlib.hist_VaR(current_p, portfolio, log_returns, alpha=0.05)
MC_VaR = functionlib.MC_VaR(current_p, portfolio, log_returns, n=1000, alpha=0.05)
print("Delta Normal VaR: {:.4f}; historic VaR: {:.4f}; MC VaR: {:.4f}".format(del_norm, his_VaR, MC_VaR))
```

```
Delta Normal VaR: 385.7948; historic VaR: 607.3003; MC VaR: 245.7245
```

In addition to VaR, calculation of ES is included in the "functionlib", and it is tested with "problem1.csv":

```
# test for ES calculation
problem1 = pd.read_csv("problem1.csv")
ES = functionlib.cal_ES(problem1,alpha=0.05)
print("ES: {:.4f}".format(ES))
```

```
ES: 0.0121
```

Eventually, I have an additional function defined in my function library, used to compute Frobenius norm of matrices. I set up two matrices and compute Frobenius norm of the difference of those matrices; as definition of Frobenius norm goes, the answer should be the square root of the sum of square of each item in a matrix:

```
# test for F_norm
mat1 = np.array([[1,2],[3,4]])
mat2 = np.array([[5,6],[7,8]])
diff = mat2 - mat1
functionlib.F_Norm(diff)
```

```
8.0
```

Since the difference matrix is [[4, 4], [4, 4]], the answer being 8 is correct. As above, tests for my function library have all been conducted (there is one last function defined in "functionlib" used to calculate returns of given prices – this function has been used and tested previously when testing functions for VaR).

Problem 3

Use your repository from #2.

Using Portfolio.csv and DailyPrices.csv. Assume the expected return on all stocks is 0.

This file contains the stock holdings of 3 portfolios. You own each of these portfolios.

Fit a Generalized T model to each stock and calculate the VaR and ES of each portfolio as well as your total VaR and ES. Compare the results from this to your VaR form Problem 3 from Week 4.

In this problem, I first read data from given files, and extracted them in accordance with different portfolios, just like I did in problem 3 in last week's project – this process includes extracting current prices of each stock in each portfolio, calculating portfolio values for each portfolio, getting arithmetic returns of each stock in each portfolio. After dealing with these data, I started to build a Gaussian copula for later simulations: 1. I fitted returns of each stock in each portfolio into a t distribution, and stored fitted parameters and distribution into a dictionary; 2. then, I computed U matrices and corresponding spearman correlation matrices; 3. with correlation matrices, I started to simulate from multivariate normal distributions with means being 0 and standard deviation being correlation matrices (in my case, I only simulate 500 times, since when I tried to run 1000 simulations, the system failed); 4. I then used simulated values to calculated updated U matrices with using cdf of standard normal distribution; 5. eventually, I converted those values back to values in t distributions and generated a new pandas dataframe. I took the generated dataframe as the simulated returns and used it to update simulated current prices. With updated prices, I updated portfolios' values, and followingly used them to calculate 5% VaR and 5% ES (using a modified version of the function defined in "functionlib", my function library); results are below:

```
# call defined fucntion to calculate ES
VaR_05_tot, ES_05_tot = cal_ES(port_tot_val, PV_tot)
VaR_05_a, ES_05_a = cal_ES(port_a_val, PV_a)
VaR_05_b, ES_05_b = cal_ES(port_b_val, PV_b)
VaR_05_c, ES_05_c = cal_ES(port_c_val, PV_c)
print("VaR: Total Portfolio: ${:.4f}, Portfolio A: ${:.4f}, Portfolio B: ${:.4f}, Portfolio C: ${:.4f}".format(VaR_05_tot, VaR_05_a, VaR_05_b, VaR_05_c))
print("ES: Total Portfolio: ${:.4f}, Portfolio A: ${:.4f}, Portfolio B: ${:.4f}, Portfolio C: ${:.4f}".format(ES_05_tot, ES_05_a, ES_05_b, ES_05_c))
```

```
VaR: Total Portfolio: $19594.5984, Portfolio A: $7625.9297, Portfolio B: $7078.2180, Portfolio C: $5919.8441
ES: Total Portfolio: $25983.4424, Portfolio A: $10518.5333, Portfolio B: $8438.9668, Portfolio C: $7831.4188
```

For your information, in last week's project, when I used arithmetic and delta normal VaR method to calculate VaR, results are:

```
# calculation of Delta Normal VaR of the combination of three portfolios
w = []
cw = []
populateWeights(port_tot_re,w,cw, 0.94)
w = w[::-1]
tot_cov = exwCovMat(port_tot_re, w)
tot_fac = math.sqrt(np.transpose(delta_tot) @ tot_cov @ delta_tot)
VaR_05_tot = -PV_tot * stats.norm.ppf(0.05, loc=0, scale=1) * tot_fac
print("5% VaR of the combination of three portfolios is ${:.4f}".format(VaR_05_tot))
print("5% VaR of the combination of three portfolios could also be expressed as {:.4f}%".format(VaR_05_tot * 100 / PV_tot))
```

```
5% VaR of the combination of three portfolios is $13577.0754
5% VaR of the combination of three portfolios could also be expressed as 1.5707%
```

```
# calculation of Delta Normal VaR of portfolio a
w = []
cw = []
populateWeights(port_a_re,w,cw, 0.94)
w = w[::-1]
a_cov = exwCovMat(port_a_re, w)
a_fac = math.sqrt(np.transpose(delta_a) @ a_cov @ delta_a)
VaR_05_a = -PV_a * stats.norm.ppf(0.05, loc=0, scale=1) * a_fac
print("5% VaR of portfolio a is ${:.4f}".format(VaR_05_a))
print("5% VaR of portfolio a could also be expressed as {:.4f}%".format(VaR_05_a * 100 / PV_a))
```

```
5% VaR of portfolio a is $5670.2029
5% VaR of portfolio a could also be expressed as 1.8904%
```

```
# calculation of Delta Normal VaR of portfolio b
w = []
cw = []
populateWeights(port_b_re,w,cw, 0.94)
w = w[::-1]
b_cov = exwCovMat(port_b_re, w)
b_fac = math.sqrt(np.transpose(delta_b) @ b_cov @ delta_b)
VaR_05_b = -PV_b * stats.norm.ppf(0.05, loc=0, scale=1) * b_fac
print("5% VaR of portfolio b is ${:.4f}".format(VaR_05_b))
print("5% VaR of portfolio b could also be expressed as {:.4f}%".format(VaR_05_b * 100 / PV_b))
```

```
5% VaR of portfolio b is $4494.5984
5% VaR of portfolio b could also be expressed as 1.5268%
```

```
# calculation of Delta Normal VaR of portfolio c
w = []
cw = []
populateWeights(port_c_re,w,cw, 0.94)
w = w[::-1]
c_cov = exwCovMat(port_c_re, w)
c_fac = math.sqrt(np.transpose(delta_c) @ c_cov @ delta_c)
VaR_05_c = -PV_c * stats.norm.ppf(0.05, loc=0, scale=1) * c_fac
print("5% VaR of portfolio c is ${:.4f}".format(VaR_05_c))
print("5% VaR of portfolio c could also be expressed as {:.4f}%".format(VaR_05_c * 100 / PV_c))
```

```
5% VaR of portfolio c is $3786.5890
5% VaR of portfolio c could also be expressed as 1.4022%
```

When I was using log returns and historic VaR method, results are:

```
# calculate VaR with Historic VaR of combined portfolio
sim_prices_tot = (np.exp(port_tot_logre)) * np.transpose(current_prices_tot)
port_values_tot = np.dot(sim_prices_tot, portfolios.Holding)
port_values_tot_sorted = np.sort(port_values_tot)
index_tot = np.floor(0.05*len(port_tot_logre))
VaR_05_tot_logH = PV_tot - port_values_tot_sorted[int(index_tot-1)]
print("5% VaR of combined portfolio using Historic VaR is ${:.4f}".format(VaR_05_tot_logH))
```

```
5% VaR of combined portfolio using Historic VaR is $21237.2184
```

```
# calculate VaR with Historic VaR of portfolio A
sim_prices_a = (np.exp(port_a_logre)) * np.transpose(current_prices_a)
port_values_a = np.dot(sim_prices_a, portfolio_a.Holding)
port_values_a_sorted = np.sort(port_values_a)
index_a = np.floor(0.05*len(port_a_logre))
VaR_05_a_logH = PV_a - port_values_a_sorted[int(index_a-1)]
print("5% VaR of portfolio A using Historic VaR is ${:.4f}".format(VaR_05_a_logH))
```

```
5% VaR of portfolio A using Historic VaR is $9138.8743
```

```
# calculate VaR with Historic VaR of portfolio B
sim_prices_b = (np.exp(port_b_logre)) * np.transpose(current_prices_b)
port_values_b = np.dot(sim_prices_b, portfolio_b.Holding)
port_values_b_sorted = np.sort(port_values_b)
index_b = np.floor(0.05*len(port_b_logre))
VaR_05_b_logH = PV_b - port_values_b_sorted[int(index_b-1)]
print("5% VaR of portfolio B using Historic VaR is ${:.4f}".format(VaR_05_b_logH))
```

```
5% VaR of portfolio B using Historic VaR is $7273.7692
```

```
# calculate VaR with Historic VaR of portfolio C
sim_prices_c = (np.exp(port_c_logre)) * np.transpose(current_prices_c)
port_values_c = np.dot(sim_prices_c, portfolio_c.Holding)
port_values_c_sorted = np.sort(port_values_c)
index_c = np.floor(0.05*len(port_c_logre))
VaR_05_c_logH = PV_c - port_values_c_sorted[int(index_c-1)]
print("5% VaR of portfolio C using Historic VaR is ${:.4f}".format(VaR_05_c_logH))
```

```
5% VaR of portfolio C using Historic VaR is $5773.4722
```

As shown above, my resulting VaRs are but significantly deviated from results using arithmetic returns and delta normal VaR method – one possible reason could be different assumptions of underlying distributions of returns, and we may notice that VaRs derived assuming returns following t distributions are generally larger than VaRs derived assuming normally distributed returns, and this could be a result of fatter tails of t distributions, comparing to normal distributions. In addition, my results in this week's project are similar to results using log returns and historic VaR method in problem 3 in last week's project. This may be due to that my resulting VaRs are derived assuming that returns are not following a normal distribution, and last week's results using historic VaR method also breaks the normality assumption of returns.