Fintech545 Project Week4

Kaiye Chen

September 2022

1 Question1

1.1 Expectations and Simulations

$$\begin{aligned} E[P_t|P_{t-1}] &= E[P_{t-1} + r_t|P_{t-1}] \\ &= P_{t-1} + E[r_t|P_{t-1}] \\ &= P_{t-1} \end{aligned}$$

$$\begin{aligned} Std[P_t|P_{t-1}] &= Std[P_{t-1} + r_t|P_{t-1}] \\ &= Std[r_t|P_{t-1}] \\ &= \sigma \end{aligned}$$

$$\begin{aligned} E[P_t|P_{t-1}] &= E[P_{t-1}r_t + P_{t-1}|P_{t-1}] \\ &= P_{t-1}E[1 + r_t|P_{t-1}] \\ &= P_{t-1} \end{aligned}$$

$$\begin{aligned} Std[P_t|P_{t-1}] &= Std[P_{t-1}r_t + P_{t-1}|P_{t-1}] \\ &= P_{t-1}Std[r_t|P_{t-1}] \\ &= P_{t-1}\sigma \end{aligned}$$

$$\begin{aligned} Std[P_t|P_{t-1}] &= Std[ln(P_{t-1}) + ln(e^{r_t})|P_{t-1}] \\ &= Std[r_t|P_{t-1}] \\ &= \sigma \end{aligned}$$

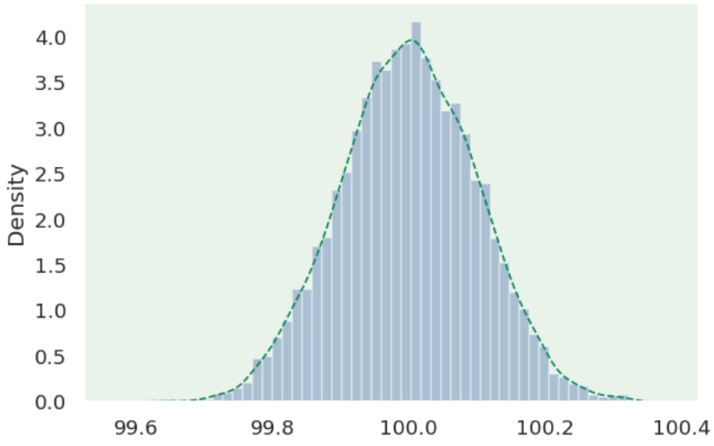
```python
def simulate_path(method, start_price, sigma, sample_size):
    return0 = scipy.stats.norm(0, sigma).rvs(sample_size)
    price_arr = []
    if method == "Classical":
        for i in range(0, sample_size):
            price = start_price + return0[i]
            price_arr.append(price)
        expected_mean = start_price
        expected_variance = sigma
    elif method == "Arithemetic":
        for i in range(0, sample_size):
            price = start_price * (1 + return0[i])
            price_arr.append(price)
        expected_mean = start_price
        expected_variance = start_price * sigma
    elif method == "Geometric":
        for i in range(0, sample_size):
            price = start_price * np.exp(return0[i])
            price_arr.append(np.log(price))
        expected_mean = np.log(start_price)
        expected_variance = sigma
    else:
        print("Wrong method!")
    return price_arr, expected_mean, expected_variance
```

In this part, I first calculate the statistical expectations of all 3 methods. Then, I defined a function to simulate the paths, as well as calculate the expectations.

1.2 Results and Comparisons

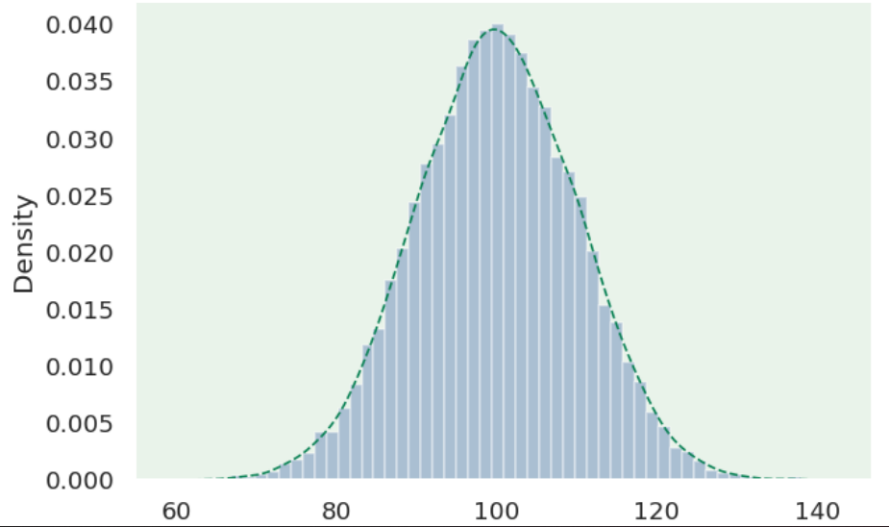
```python
show_simulation_results("Classical", 100, 0.1, 10000)
```

```
Classical
        simulate_mean = 100.00, expected_mean = 100.00
        simulate_standard_deviation = 0.10, expected_sigma = 0.10
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is
  warnings.warn(msg, FutureWarning)
```



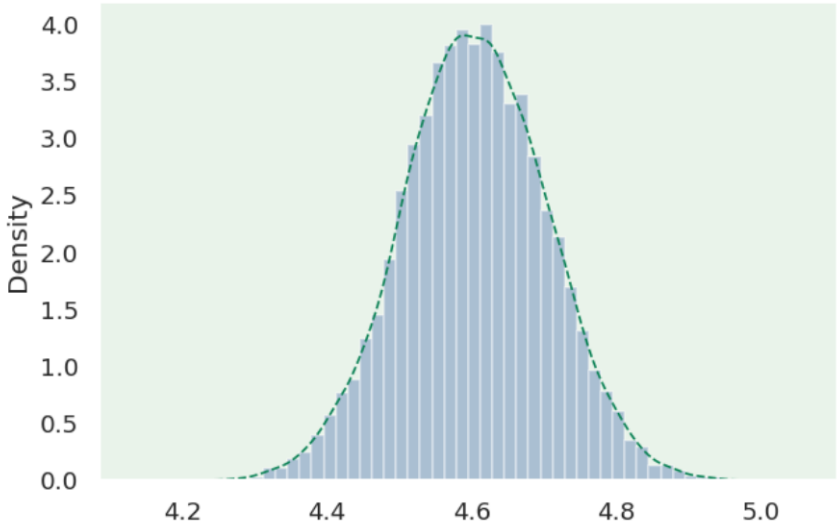
```
show_simulation_results("Arithemetic",  100,  0.1,  10000)
```

```
Arithemetic
        simulate_mean = 99.98, expected_mean = 100.00
        simulate_standard_deviation = 10.04, expected_sigma = 10.00
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` i
  warnings.warn(msg, FutureWarning)
```




```
show_simulation_results("Geometric",  100,  0.1,  10000)
```

```
Geometric
        simulate_mean = 4.60, expected_mean = 4.61
        simulate_standard_deviation = 0.10, expected_sigma = 0.10
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplo
  warnings.warn(msg, FutureWarning)
```



In this part, I use density plot and KDE to smooth all the results. The results shows that the simulation mean and deviation is very closed to the expectation. Meanwhile, I choose 10000 as my sample size, as this number become larger, the distance between simulation and expectation will become smaller.

2 Question2

2.1 Defined Simulation Methods

```
def simulate_normal(df, alpha, sample_size):
    sigma = df.std()
    normal = np.random.normal(0,sigma,size=sample_size)
    VaR = -np.percentile(normal, alpha)
    print(f"Normal distribution VaR: {round(VaR*100,2)}%")
    return VaR, normal
```

```
def simulate_EW(df, lambda0, alpha, sample_size):
    temp = return_set.reset_index()
    adj_weights = cal_adj_weights(lambda0, temp)
    sigma = np.sqrt(cal_cov(df.values, df.values, adj_weights))
    Weighted_normal = np.random.normal(0, sigma, 10000)
    VaR = -np.percentile(Weighted_normal,5)
    print(f"Exponentially Weighted Normal VaR: {round(VaR*100,2)}%")
    return VaR, Weighted_normal
```

```
def MLE_t(array):
    def t_log_lik(par_vec, x):
        lik = -np.log(t(df=par_vec[0], loc=par_vec[1], scale=par_vec[2]).pdf(x)).sum()
        return lik
    cons = ({'type': 'ineq', 'fun': lambda x: x[0] - 2},{'type': 'ineq', 'fun': lambda x: x[2]})
    df, mean, scale = optimize.minimize(fun=t_log_lik, x0=[2, np.array(array).mean(), np.array(array).std()
    return df, mean, scale

def simulate_MLE_t(df, alpha, sample_size):
    t_params = MLE_t(return_set)
    simulation_t = t.rvs(df = t_params[0], loc = t_params[1], scale = t_params[2], size = sample_si
    VaR = -np.percentile(simulation_t,alpha)
    print(f"T distribution VaR: {round(VaR*100,2)}%")
    return VaR, simulation_t
```

```
def simulate_history(df, alpha):
    VaR = -np.percentile(df,5)
    print(f"Historic Simulation VaR: {round(VaR*100,2)}%")
    return VaR
```

In this part, I defined several functions to simulate the distribution with different methods.

2.2 Introduced out of sample data

```
def show_simulation_results1(df, df1, lambda0, alpha, sample_size):
    Normal_VaR, Normal_set = simulate_normal(df, alpha, sample_size)
    Weighted_Normal_VaR, Weighted_set = simulate_EW(df, lambda0, alpha, sample_size)
    MLE_t_VaR, t_set = simulate_MLE_t(df, alpha, sample_size)
    History_VaR = simulate_history(df, alpha)

    Normal_VaR1, Normal_set1 = simulate_normal(df1, alpha, sample_size)
    Weighted_Normal_VaR1, Weighted_set1 = simulate_EW(df1, lambda0, alpha, sample_size)
    MLE_t_VaR1, t_set1 = simulate_MLE_t(df1, alpha, sample_size)
    History_VaR1 = simulate_history(df1, alpha)

    fig, axes = plt.subplots(1, 2, figsize=(18,5))
    sns.histplot(df, binrange = (-.2,.2),binwidth = 0.01,stat='density',ax=axes[0], label='Historical', color='b', alpha=0.5)
    sns.distplot(Normal_set, ax=axes[0], hist=False, label='Normal')
    sns.distplot(Weighted_set, ax=axes[0], hist=False, label='Weighted Normal')
    sns.distplot(t_set, ax=axes[0], hist=False, label='T')
    axes[0].legend()

    sns.histplot(df1, binrange = (-.2,.2),binwidth = 0.01,stat='density',ax=axes[1], label='OOS', color='black', alpha=0.5)
    sns.distplot(Normal_set1, ax=axes[1], hist=False, label='Normal')
    sns.distplot(Weighted_set1, ax=axes[1], hist=False, label='Weighted Normal')
    sns.distplot(t_set1, ax=axes[1], hist=False, label='T')
    axes[1].legend()
```

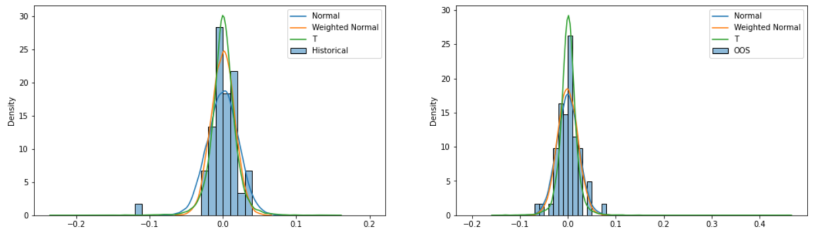
In this part, I introduced new data from yahoo finance, range from 1/15 to 4/15. Then, I used to data sets to run all 4 simulation process individually.

2.3 Simulation Results

```
show_simulation_results1(return_set, return_oos, 0.94, 5, 10000)
```

```
Normal distribution VaR: 3.46%
Exponentially Weighted Normal VaR: 2.61%
T distribution VaR: 2.63%
Historic Simulation VaR: 2.07%
Normal distribution VaR: 3.74%
Exponentially Weighted Normal VaR: 3.49%
T distribution VaR: 2.65%
Historic Simulation VaR: 2.86%
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: distplot is a deprecated function and will be removed in a future
  warnings.warn(msg, FutureWarning)
```



In this part, I get the results from two data set. In each data set, the normal simulation produced the largest VaR. However, for other 3 methods, there is no obvious ranking relationship. As a result, I can't say that there is any kind of concrete relationship between each simulation method. Comparing two data set, we can see that the VaRs are very closed, which indicates that in these two time period, the risk profile of selected stocks under VaR metric. Also in the plot, we can also see that the distribution of all simulation methods are pretty closed.

3 Question3

3.1 Calculation methods

```
def Historical_VaR(portfolio, price, alpha):
    portfolio = portfolio.set_index('Stock')
    current_price = pd.DataFrame({"price":price.iloc[-1]})
    portfolio = portfolio.join(current_price.loc[portfolio.index])
    history_values = price[portfolio.index] @ portfolio['Holding']
    return0 = history_values.diff().dropna()
    return -np.percentile(return0,alpha*100)
```

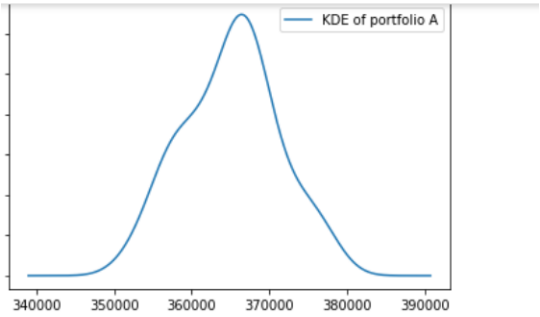
```
def DeltaNormal_VaR(portfolio, price, alpha):
    portfolio = portfolio.set_index('Stock')
    current_price = pd.DataFrame({"price":price.iloc[-1]})
    portfolio = portfolio.join(current_price.loc[portfolio.index])
    portfolio["Value"] = portfolio['Holding'] * portfolio['price']
    PV = portfolio["Value"].sum()
    portfolio["Weight"] = portfolio["Value"] / PV
    weight = portfolio["Weight"]

    return_all = price.pct_change().dropna()
    port_sigma = np.sqrt(weight.T @ return_all[portfolio.index].cov() @ weight)

    return -PV*port_sigma*norm.ppf(alpha)
```

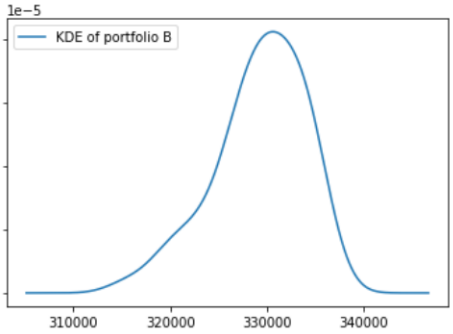
In this part, I defined two possible methods to calculate the VaR, delta normal and historical. For the delta normal method, since the return is linear, the gradient in portfolio sigma can be simplified as weight.

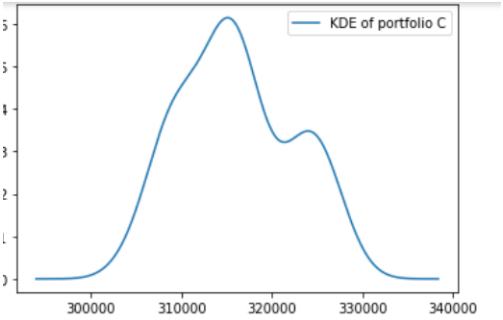
3.2 Method chosen

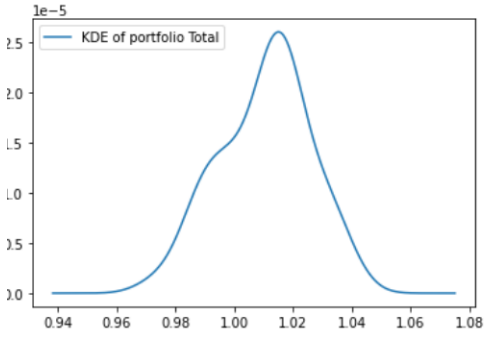
```
def test_normality(df, alpha):
    test_stat, p = scipy.stats.shapiro(df)
    if p >= alpha:
        return True
    else:
        return False
```

```
Normality: True
Normality: False
Normality: False
Normality: True
1e−5
```









In this part, I defined a function to test the normality of the distribution. It is a traditional sapiro test. The results shows that not all the distribution

are normal, so it didn't fit the assumption of delta normal, so I will choose the historical method. The kdf plot here also shows that portfolio B and C are not normally distributed.

3.3 Results

```
results
```

	A	B	C	Total
Historical_VaR	5610.072337	5664.361566	3163.626064	12697.538762

In this final part, it shows the result of VaR calculation under historical method. The result here is subadditive since A+B+C ¡ Total.