risk_hw4

April 30, 2024

[10]:
```
import os
import yfinance as yf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from math import *
from arch import arch_model
from scipy.stats import norm
os.chdir('/Users/yiyujie/Desktop/program/Financial Risk Management')
```

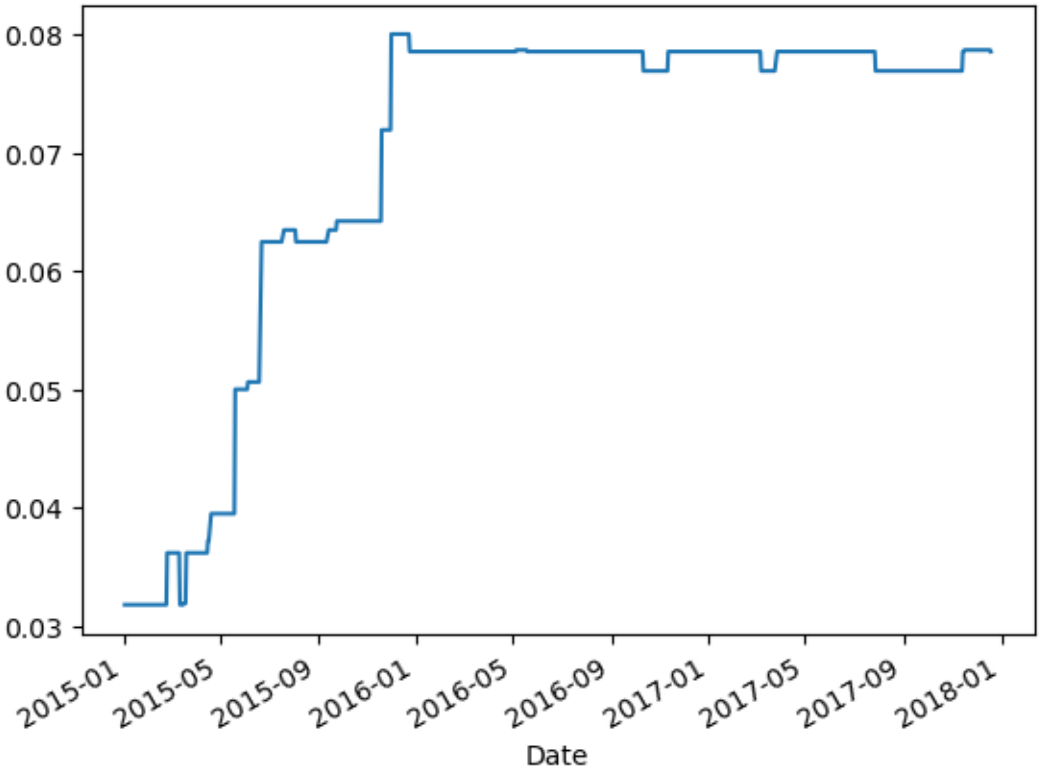
0.1 1. Choosing a VaR technique

Download the excel file which contains the time series of gains for a strategy from 1/2/2014 to 12/19/2017. 1. Historical method * For each day in 2015-2017, compute historical VaR and exponential weighted 1-day 99%-VaR (with = 0.995).

[2]:
```
returns4 = pd.read_csv('hw4_returns.csv')
returns4['Date'] = pd.to_datetime(returns4['Date'])
returns4 = returns4.set_index('Date')
returns_2015 = returns4['2015':]
```

[3]:
```
def historical_var(returns, confidence_level=0.99):
    # Sort returns from smallest to largest
    sorted_returns = returns.sort_values()

    # Calculate the index corresponding to the 1% quantile
    var_index = int((1 - confidence_level) * len(sorted_returns))

    # VaR is the return at that index
    var = - sorted_returns.iloc[var_index]

    return var

historical_vars = returns4['Return'].expanding().apply(lambda x:␣
 ↪historical_var(pd.Series(x)), raw=True)
historical_vars = historical_vars['2015':]
historical_vars.plot()
```

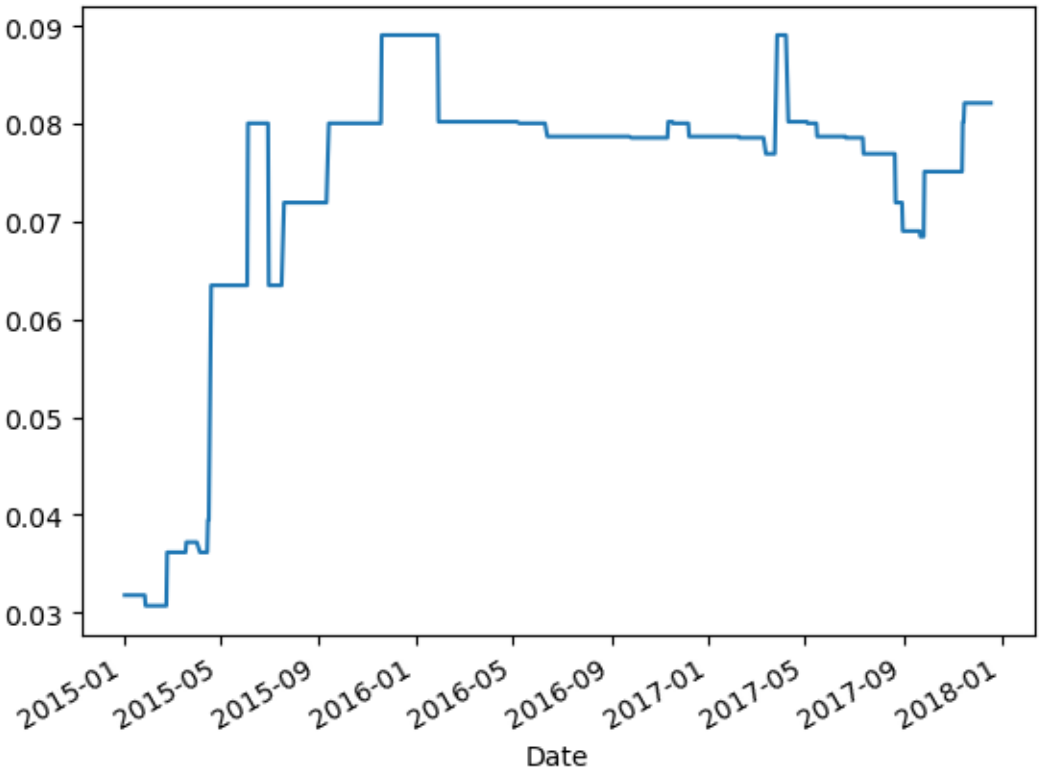
[3]: <Axes: xlabel='Date'>



[4]:
```
def exponential_historical_var(returns, lam, confidence_level=0.99):
    returns_var = pd.DataFrame(returns.copy().rename('Return'))
    # multiply weights
    n = len(returns_var)
    returns_var['weight'] = (1-lam)/(1-lam**n)*lam**(np.linspace(n-1, 0, n))

    # Sort returns from smallest to largest
    sorted_returns = returns_var.sort_values(by = 'Return')

    # cumsum weight
    sorted_returns['cumsum_weight'] = np.cumsum(sorted_returns['weight'])

    # exponential var
    var = -min(sorted_returns.loc[sorted_returns['cumsum_weight'] >=␣
 ↪1-confidence_level, 'Return'])

    return var

historical_vars_exp = returns4['Return'].expanding().apply(lambda x:␣
 ↪exponential_historical_var(pd.Series(x), lam=0.995), raw=True)
```

```
historical_vars_exp = historical_vars_exp['2015':]
historical_vars_exp.plot()
```

[4]: <Axes: xlabel='Date'>

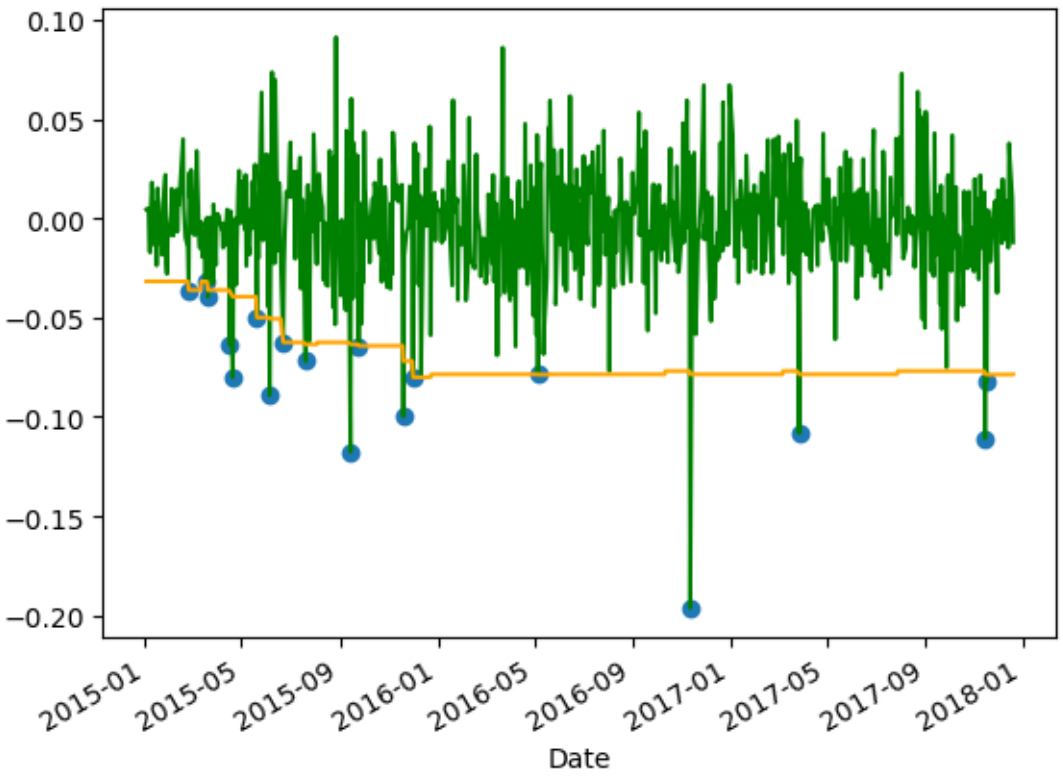


- Backtest the measures for VaR you obtained in question 1. How many exceptions did the two measures produce? What do you conclude?

[5]:
```
returns_2015['Return'].plot(c='green')
(-historical_vars).plot(c='orange')
breaches = returns_2015[returns_2015['Return']<=-historical_vars]
print("The number of exceptions are ", len(breaches))
plt.scatter(breaches.index, breaches['Return'])
```

```
The number of exceptions are  18
```

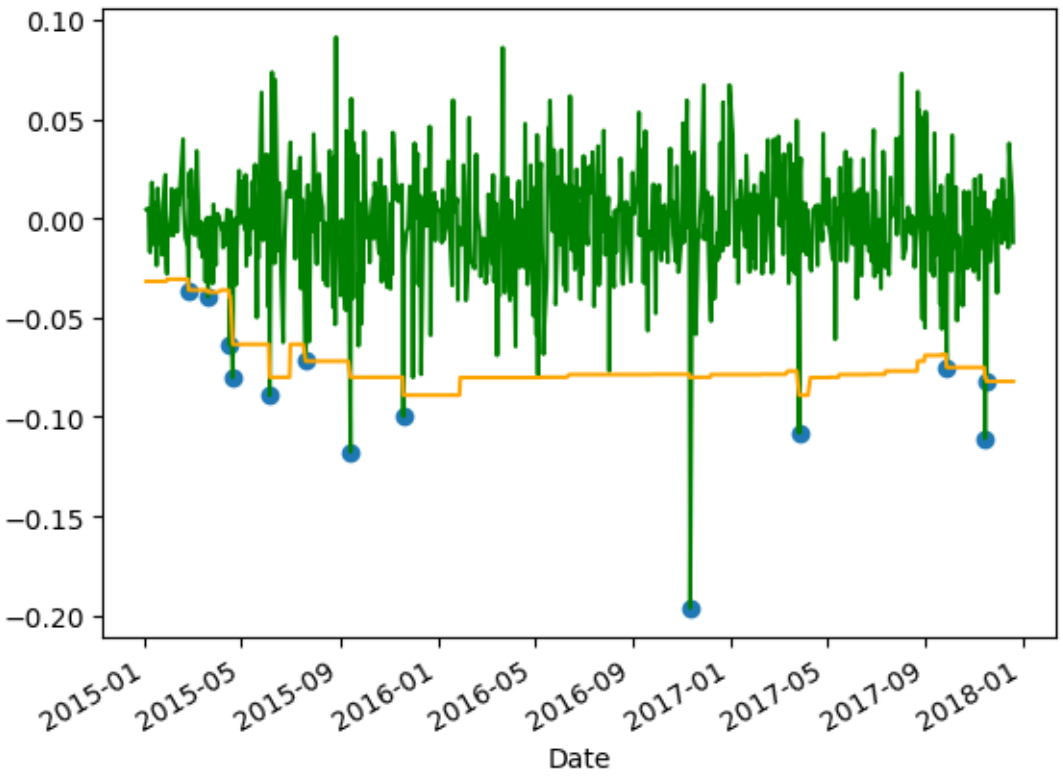
[5]: <matplotlib.collections.PathCollection at 0x14d1d6890>



[6]:
```
returns_2015['Return'].plot(c='green')
(-historical_vars_exp).plot(c='orange')
breaches = returns_2015[returns_2015['Return']<=-historical_vars_exp]
print("The number of exceptions are ", len(breaches))
plt.scatter(breaches.index, breaches['Return'])
```

```
The number of exceptions are  13
```

[6]:
```
<matplotlib.collections.PathCollection at 0x14d04d7d0>
```



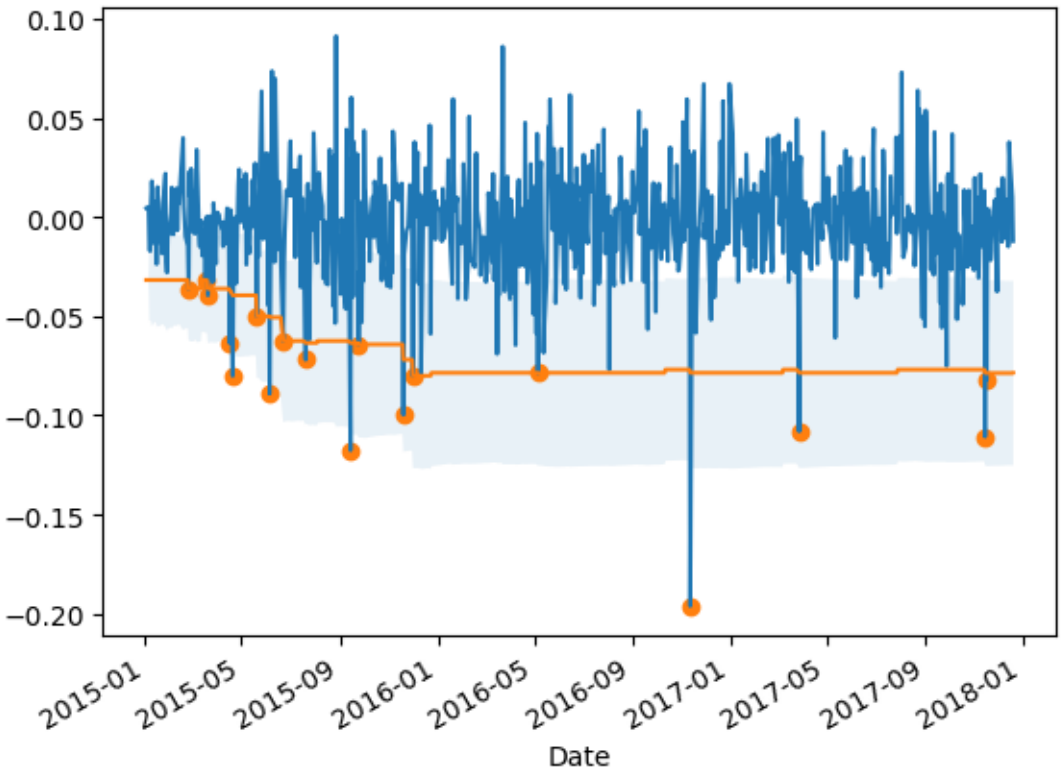
- For each day in the sample, compute the 95% confidence intervals of the historical VaR and the exponential weighted VaR you obtained in question 1, using both parametric (for the historical VaR) and boot-strap methods (for the two measures). For the parametric method, assume the gains are normally distributed.

[9]:
```
def parametric_confidence_interval(returns, var_estimate, confidence_level=0.
↪95):
    std_dev = returns.std()
    z_score = norm.ppf(confidence_level)
    return (var_estimate - z_score * std_dev, var_estimate + z_score * std_dev)
```

[51]:
```
# historical VaR + parametric_confidence_interval
returns_var = pd.merge(returns_2015, historical_vars.rename('var'),␣
↪left_index=True, right_index=True)

# Initialize columns for the lower and upper bounds
returns_var['lower_ci'] = np.nan
returns_var['upper_ci'] = np.nan

# Loop through the DataFrame using expanding windows
for i in range(len(returns_var)):
```

```
    # Calculate the confidence intervals based on the current VaR estimate and␣
 ↪expanding std dev
    lower_ci, upper_ci = parametric_confidence_interval(returns_var['Return'][:
 ↪i+1], returns_var['var'].iloc[i])
    # Assign the calculated intervals to the DataFrame
    returns_var.loc[returns_var.index[i], 'lower_ci'] = lower_ci
    returns_var.loc[returns_var.index[i], 'upper_ci'] = upper_ci

returns_2015['Return'].plot(label='return')
(-historical_vars).plot()
plt.fill_between(returns_var.index, -returns_var['lower_ci'],␣
 ↪-returns_var['upper_ci'], alpha=0.1, label='95% Confidence Interval')
breaches = returns_2015[returns_2015['Return']<=-historical_vars]
plt.scatter(breaches.index, breaches['Return'])
```

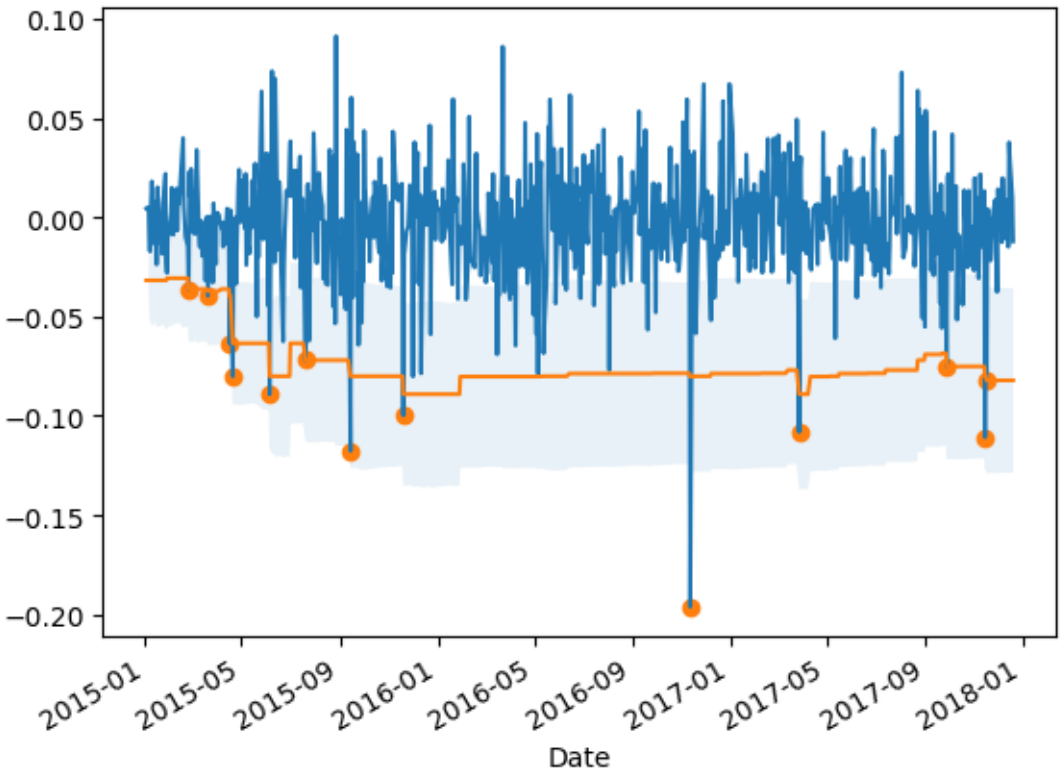
[51]: <matplotlib.collections.PathCollection at 0x1570eebd0>



[52]:
```
# exponential weighted VaR + parametric_confidence_interval
returns_var_exp = pd.merge(returns_2015, historical_vars_exp.rename('var'),␣
 ↪left_index=True, right_index=True)
```

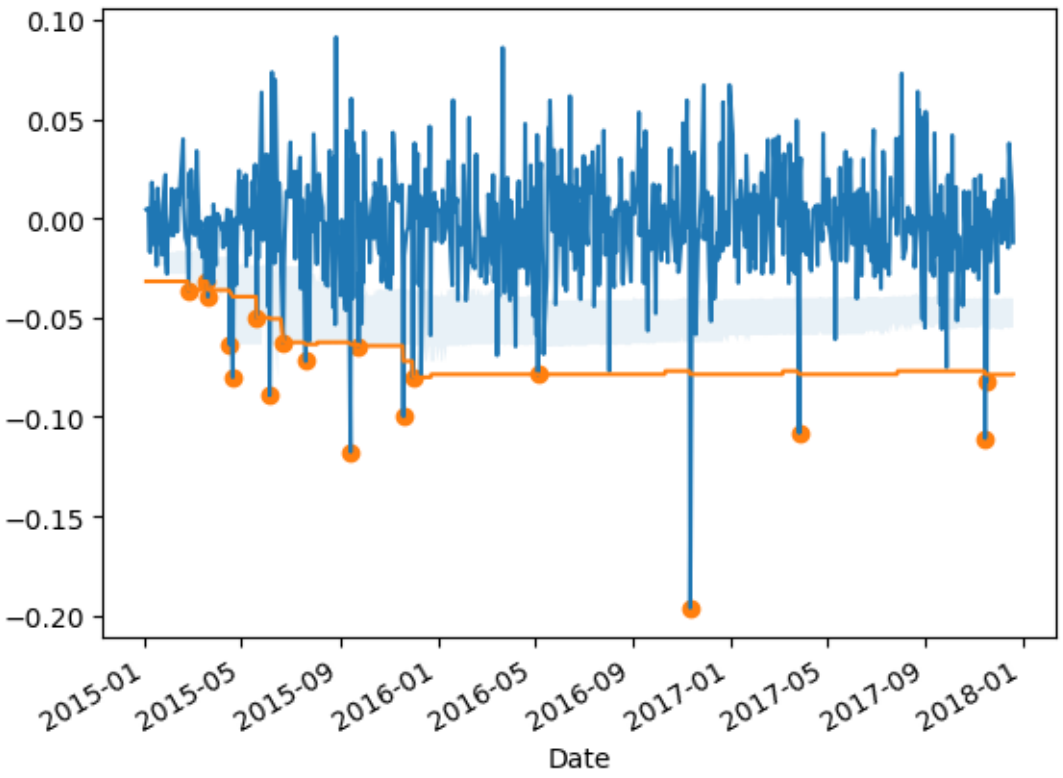
```
# Initialize columns for the lower and upper bounds
returns_var_exp['exp_lower_ci'] = np.nan
returns_var_exp['exp_upper_ci'] = np.nan

# Loop through the DataFrame using expanding windows
for i in range(len(returns_var_exp)):
    # Calculate the confidence intervals based on the current VaR estimate and␣
 ↪expanding std dev
    lower_ci, upper_ci =␣
 ↪parametric_confidence_interval(returns_var_exp['Return'][:i+1],␣
 ↪returns_var_exp['var'].iloc[i])
    # Assign the calculated intervals to the DataFrame
    returns_var_exp.loc[returns_var_exp.index[i], 'exp_lower_ci'] = lower_ci
    returns_var_exp.loc[returns_var_exp.index[i], 'exp_upper_ci'] = upper_ci

returns_2015['Return'].plot()
(-historical_vars_exp).plot()
plt.fill_between(returns_var.index, -returns_var_exp['exp_lower_ci'],␣
 ↪-returns_var_exp['exp_upper_ci'], alpha=0.1, label='95% Confidence Interval')
breaches = returns_2015[returns_2015['Return']<=-historical_vars_exp]
plt.scatter(breaches.index, breaches['Return'])
```

[52]: <matplotlib.collections.PathCollection at 0x157142010>



[130]:
```
# historical VaR + boot-strap methods
confidence_level = 0.95
iterations = 1000
np.random.seed(42)

# Initialize columns for the lower and upper bounds
returns_2015.loc[:,'bt_lower_ci'] = np.nan
returns_2015.loc[:,'bt_upper_ci'] = np.nan

# Loop through the DataFrame using expanding windows
for i in range(len(returns_2015)):
    VaRs = np.zeros(iterations)
    returns = returns_2015.iloc[:i+1]
    # bootstrap
    bootstrap_samples = np.random.choice(returns['Return'],␣
 ↪(iterations,len(returns)), replace=True)
    for k,data in zip(range(iterations),bootstrap_samples):
        data = pd.Series(data)
        VaRs[k] = historical_var(data, confidence_level = confidence_level)

    # Calculate the confidence intervals based on the current VaR estimate and␣
 ↪expanding std dev
    lower_ci, upper_ci = np.quantile(VaRs, [0.025, 0.975])
    # Assign the calculated intervals to the DataFrame
    returns_2015.loc[returns_2015.index[i], 'bt_lower_ci'] = lower_ci
    returns_2015.loc[returns_2015.index[i], 'bt_upper_ci'] = upper_ci

plt.fill_between(returns_2015.index, -returns_2015['bt_lower_ci'],␣
 ↪-returns_2015['bt_upper_ci'], alpha=0.1, label='95% Confidence Interval')
returns_2015['Return'].plot(label='return')
(-historical_vars).plot()
breaches = returns_2015[returns_2015['Return']<=-historical_vars]
plt.scatter(breaches.index, breaches['Return'])
```

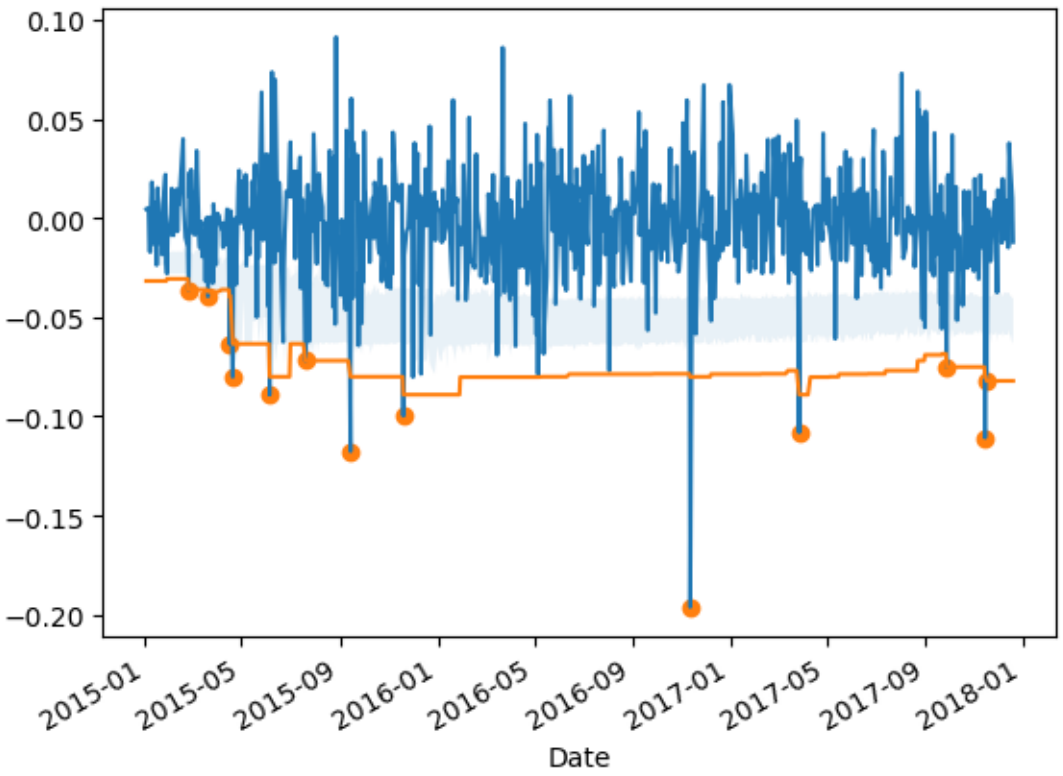
[130]:
```
<matplotlib.collections.PathCollection at 0x15b306bd0>
```



[86]:
```
# exponential weighted VaR + boot-strap methods
confidence_level = 0.95
iterations = 100
lam = 0.995

# Initialize columns for the lower and upper bounds
returns_2015.loc[:,'bt_exp_lower_ci'] = np.nan
returns_2015.loc[:,'bt_exp_upper_ci'] = np.nan

# Loop through the DataFrame using expanding windows
for i in range(len(returns_2015)):
    VaRs = np.zeros(iterations)
    returns = returns_2015.iloc[:i+1]
    # bootstrap
    bootstrap_samples = np.random.choice(returns['Return'],␣
 ↪(iterations,len(returns)), replace=True)
    for k,data in zip(range(iterations),bootstrap_samples):
        data = pd.Series(data)
        VaRs[k] = exponential_historical_var(data, lam=lam,␣
 ↪confidence_level=confidence_level)
```

```
    # Calculate the confidence intervals based on the current VaR estimate and␣
 ↪expanding std dev
    lower_ci, upper_ci = np.quantile(VaRs, [0.025, 0.975])
    # Assign the calculated intervals to the DataFrame
    returns_2015.loc[returns_2015.index[i], 'bt_exp_lower_ci'] = lower_ci
    returns_2015.loc[returns_2015.index[i], 'bt_exp_upper_ci'] = upper_ci

returns_2015['Return'].plot()
(-historical_vars_exp).plot()
plt.fill_between(returns_var.index, -returns_2015['bt_exp_lower_ci'],␣
 ↪-returns_2015['bt_exp_upper_ci'], alpha=0.1, label='95% Confidence Interval')
breaches = returns_2015[returns_2015['Return']<=-historical_vars_exp]
plt.scatter(breaches.index, breaches['Return'])
```

[86]: <matplotlib.collections.PathCollection at 0x15847a210>

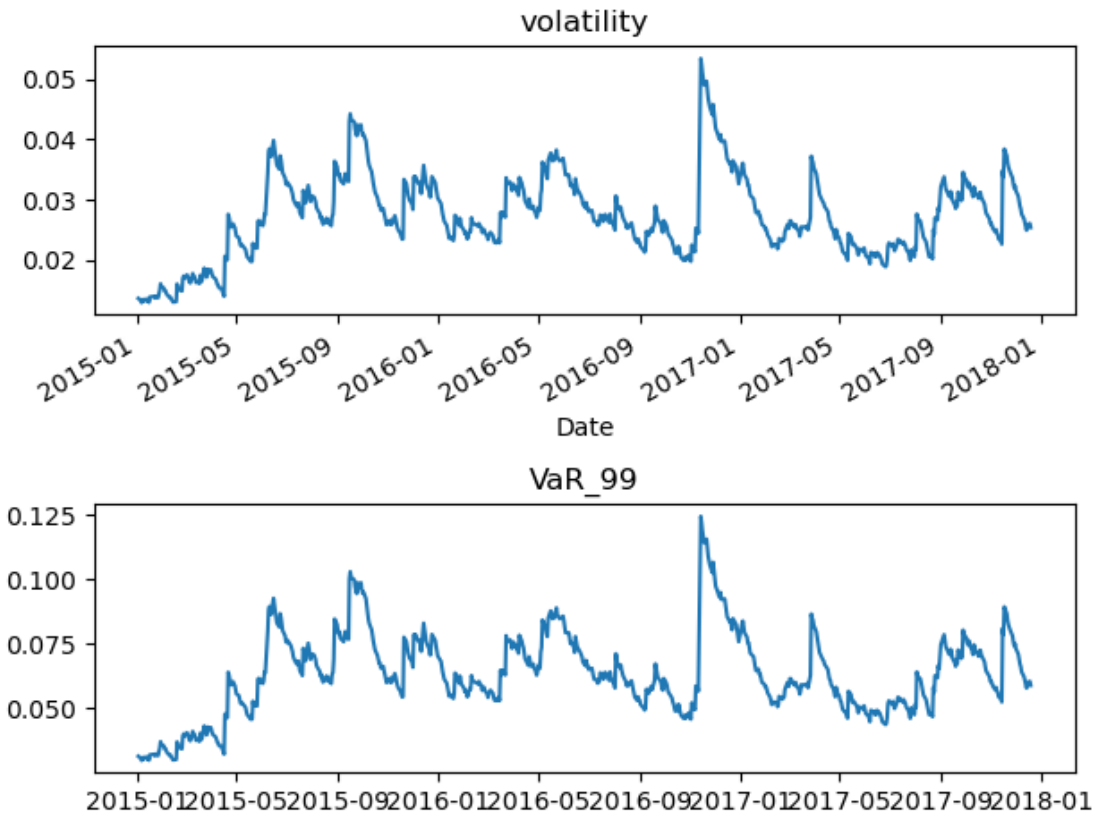


2. Model-building approach

- Compute volatility using the EWMA with = 0.94. Compute the corresponding measure of VaR.

[13]:
```
import warnings
warnings.filterwarnings('ignore')
```

[14]:
```
confidence_level = 0.99

returns4['Return2'] = returns4['Return']**2
return_2 = returns4.loc[returns4.index>='2015']
Lambda = 0.94
sigma0 = sum(np.array(returns4.loc[returns4.index<'2015','Return2']) *␣
 ↪(1-Lambda)*Lambda**(np.linspace(len(returns4.loc[returns4.index<'2015'].
 ↪index), 1, len(returns4.loc[returns4.index<'2015'].index))))
return_2.loc[return_2.index[0], 'sigma2'] = sigma0 * Lambda + return_2.
 ↪loc[return_2.index[0], 'Return2'] * (1-Lambda)

for j in range(1, len(return_2)):
    return_2.loc[return_2.index[j], 'sigma2'] = return_2.loc[return_2.
 ↪index[j-1], 'sigma2'] * Lambda + return_2.loc[return_2.index[j-1],␣
 ↪'Return2'] * (1-Lambda)

return_2.loc[:,'sigma'] = np.sqrt(return_2['sigma2'])

# visualize
plt.subplot(2, 1, 1)
return_2['sigma'].plot()
plt.title('volatility')
plt.subplot(2, 1, 2)
plt.plot(-return_2['sigma']*norm.ppf(0.01))
plt.title('VaR_99')
plt.tight_layout()
plt.show()
```

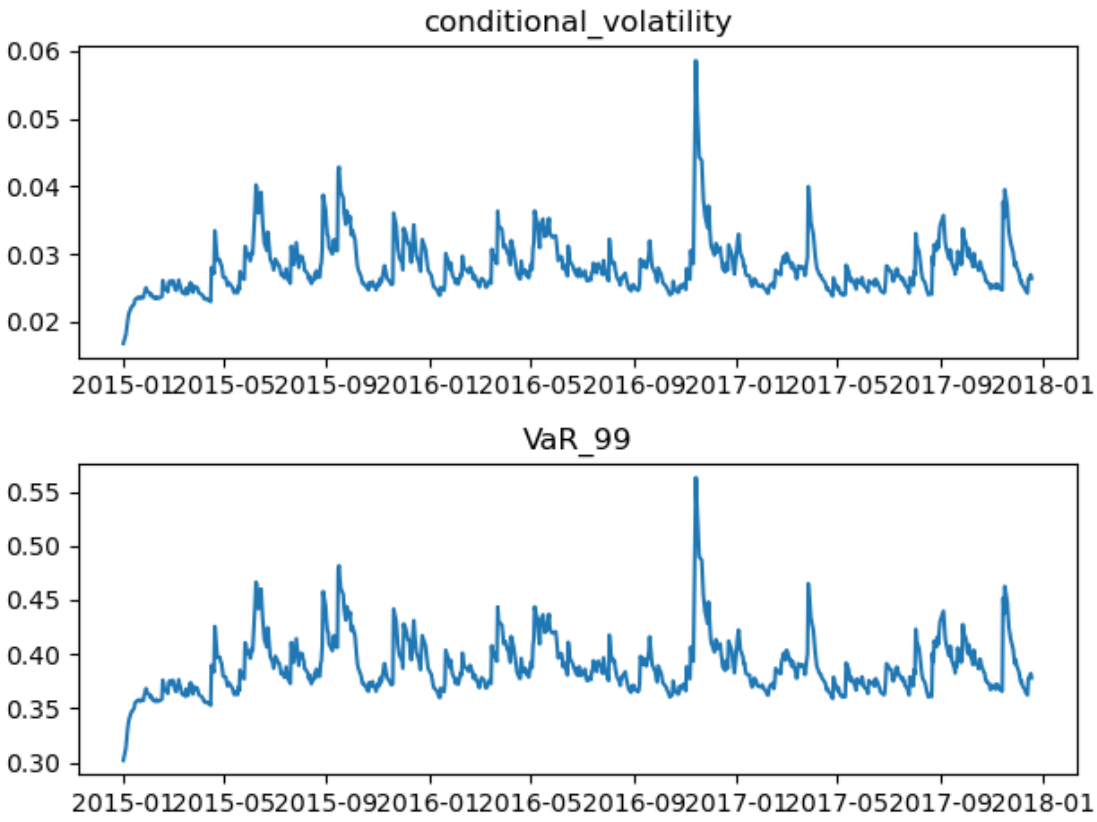


- Use maximum likelihood estimation to estimate a GARCH model for volatility. Compute the corresponding measure of VaR.

[15]:
```
confidence_level = 0.99

# Convert returns to percentage if they're not already
returns = np.array(100 * return_2['Return'])

# Fit the GARCH(1,1) model
model = arch_model(returns, mean='constant', vol='GARCH', p=1, q=1)
model_fit = model.fit(disp='off')  # disp='off' turns off the convergence␣
 ↪messages

# Display the summary of the model fit
print(model_fit.summary())

# Calculate the conditional volatility
conditional_volatility = model_fit.conditional_volatility
plt.subplot(2, 1, 1)
plt.plot(return_2.index,conditional_volatility/100)
plt.title('conditional_volatility')
```

```python
# VaR is the quantile multiplied by the conditional volatility
VaR_99 = -norm.ppf(0.01) * np.sqrt(conditional_volatility/100)

plt.subplot(2, 1, 2)
plt.plot(return_2.index,VaR_99)
plt.title('VaR_99')
plt.tight_layout()
plt.show()
```

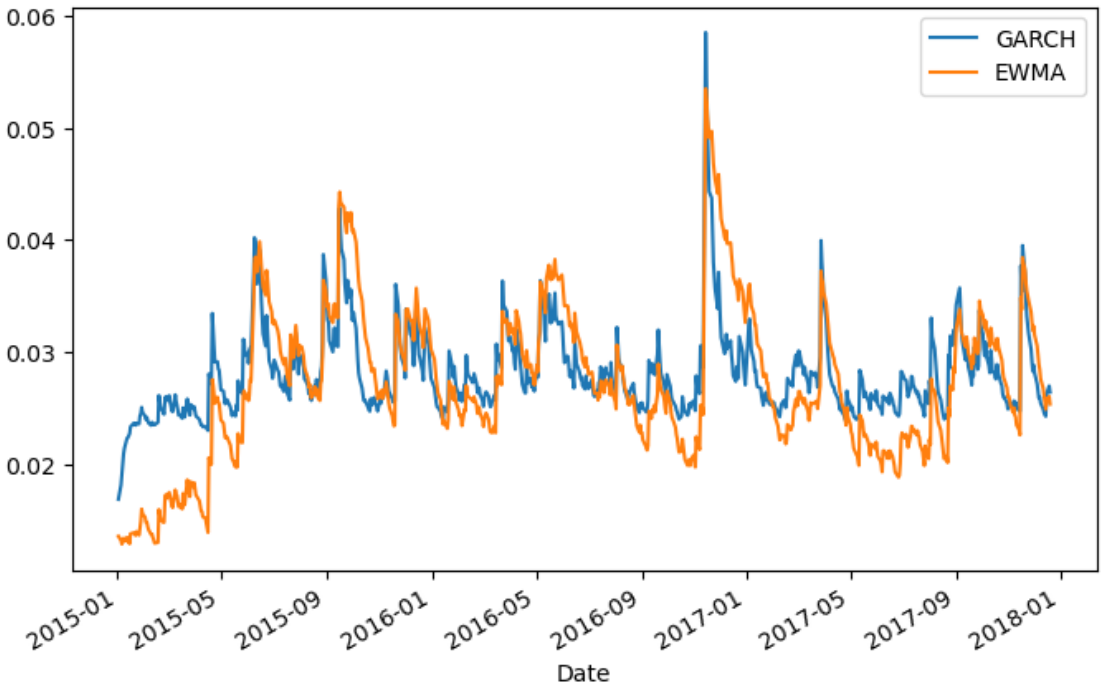
```
                     Constant Mean - GARCH Model Results
==============================================================================
Dep. Variable:                          y   R-squared:                       0.000
Mean Model:                Constant Mean   Adj. R-squared:                  0.000
Vol Model:                         GARCH   Log-Likelihood:               -1826.91
Distribution:                     Normal   AIC:                           3661.82
Method:            Maximum Likelihood      BIC:                           3680.29
                                           No. Observations:                  748
Date:                  Tue, Apr 30 2024    Df Residuals:                      747
Time:                          15:17:46    Df Model:                            1
                                Mean Model
==============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
------------------------------------------------------------------------------
mu            -0.2884      0.118     -2.450  1.429e-02 [ -0.519,-5.767e-02]
                            Volatility Model
==============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
------------------------------------------------------------------------------
omega          0.9345      0.352      2.652  7.996e-03    [  0.244,  1.625]
alpha[1]       0.0715  4.697e-02      1.523      0.128 [-2.052e-02,  0.164]
beta[1]        0.8142  6.415e-02     12.693  6.506e-37    [  0.689,  0.940]
==============================================================================

Covariance estimator: robust
```



- Compare the results from the two approaches.

[18]:
```
plt.figure(figsize=(8,5))
plt.plot(return_2.index,conditional_volatility/100,label='GARCH')
return_2['sigma'].plot(label='EWMA')
plt.legend()
```

[18]:
```
<matplotlib.legend.Legend at 0x14e363690>
```



- **Volatility Dynamics**
  - EWMA: It assigns a greater weight to more recent observations using a decay factor (, lambda). The decay factor in your EWMA model is 0.94, which means that recent observations have significantly more influence on the volatility estimate than older ones. This approach is relatively simple and reactive to recent market movements but does not capture long-term trends as effectively.
  - GARCH: On the other hand, estimates volatility based on a longer-term perspective by considering not only recent squared returns (alpha parameter) but also the persistence of volatility over time (beta parameter). Your GARCH model, as shown in the summary, has an alpha of 0.0715 and a significant beta of 0.8142, suggesting that past variances are quite influential in predicting future variance. The model captures volatility clustering, which is common in financial time series.
- **Modeling Approach**
  - EWMA: It is a form of a simple moving average with exponential weighting and does not inherently model the distribution of returns.
  - GARCH: Uses MLE (Maximum Likelihood Estimation), a statistical method to find the parameters that make the observed data most probable. This method allows for statistical testing of the significance of each parameter, as evident from the t-statistics and p-values in the summary. The negative mean (mu) suggests that the returns are, on average, below zero, but the confidence interval includes zero, indicating this result may not be statistically significant.
- **Volatility Patterns**

  - The plotted volatility from EWMA in Figure 1 shows a less smooth volatility path compared to Figure 2, which may reflect the reactive nature of EWMA to recent returns. GARCH, given its parameters, seems to produce a smoother volatility path that slowly reacts to spikes in volatility.

- **Statistical Significance**
  - GARCH's parameter significance can be directly assessed through t-statistics and p-values. The beta parameter is highly significant, which confirms that past volatility (volatility clustering) is crucial for forecasting future volatility in this time series. On the contrary, the significance of the alpha parameter is not confirmed (p-value > 0.05), suggesting that the short-term shocks have less impact than the long-term volatility persistence.

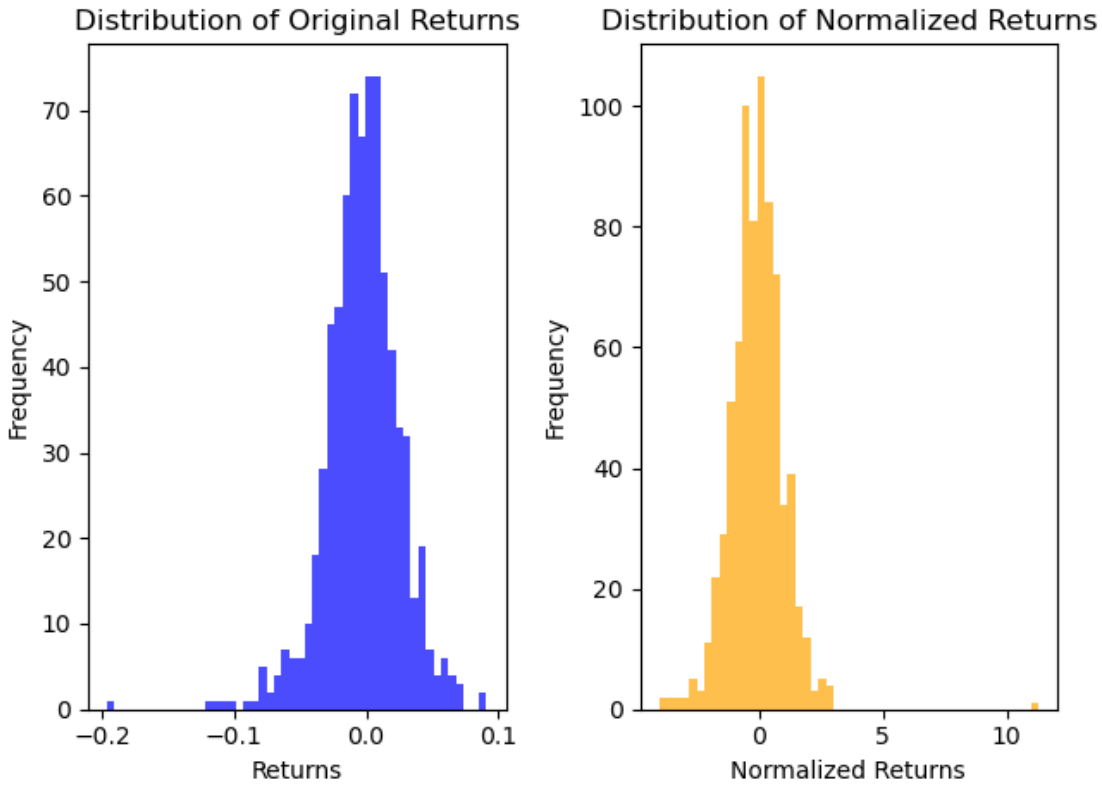
3. A mixed approach

- For each day in the sample, compute the volatility of the portfolio in the previous month. Normalize gains with estimated volatility. Compare the distribution of the normalized gain with the original ones.

[93]:
```
return_2['monthly_vol'] = return_2['Return'].rolling(window='30D').std()
return_2['normalized_returns'] = return_2['Return'] / return_2['monthly_vol']
```

[94]:
```
# Histogram of original returns
plt.subplot(1, 2, 1)
plt.hist(return_2['Return'].dropna(), bins=50, alpha=0.7, color='blue')
plt.title('Distribution of Original Returns')
plt.xlabel('Returns')
plt.ylabel('Frequency')

# Histogram of normalized returns
plt.subplot(1, 2, 2)
plt.hist(return_2['normalized_returns'].dropna(), bins=50, alpha=0.7,␣
 ↪color='orange')
plt.title('Distribution of Normalized Returns')
plt.xlabel('Normalized Returns')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()

# You can also calculate and compare statistical properties such as mean,␣
 ↪variance, skewness, and kurtosis.
from scipy.stats import skew, kurtosis

original_skewness = skew(return_2['Return'].dropna())
original_kurtosis = kurtosis(return_2['Return'].dropna())
normalized_skewness = skew(return_2['normalized_returns'].dropna())
normalized_kurtosis = kurtosis(return_2['normalized_returns'].dropna())
```

```
print(f'Original Skewness: {original_skewness}, Normalized Skewness:␣
 ↪{normalized_skewness}')
print(f'Original Kurtosis: {original_kurtosis}, Normalized Kurtosis:␣
 ↪{normalized_kurtosis}')
```




```
Original Skewness: -0.6954357681727689, Normalized Skewness: 1.257977245402529
Original Kurtosis: 3.891286430262606, Normalized Kurtosis: 14.935442048255343
```

- Develop an approach to measure VaR which takes advantage of your response to the previous question. Implement it and compare its exceptions with the previous approaches. Optional: You can use the approach of extreme value theory.

[97]:
```
# Calculate historical VaR on normalized returns
historical_normalized_var = -np.percentile(return_2['normalized_returns'].
 ↪dropna(), (1-confidence_level) * 100)

# Convert normalized VaR back to the original scale using the most recent␣
 ↪volatility
most_recent_volatility = return_2['monthly_vol'].iloc[-1]
historical_var = historical_normalized_var * most_recent_volatility
```

```
print(f"Historical Normalized VaR (95%): {historical_normalized_var}")
print(f"Historical VaR (95%): {historical_var}")
```

```
Historical Normalized VaR (95%): 1.7046794532972092
Historical VaR (95%): 0.03027561277401434
```

4. Combining your answers to the previous questions, write a proposal to the head of trading for measuring the risk of this trade in real time, justifying your choices.

Proposal for Implementing a Real-Time Risk Measurement Framework

Objective The objective of this proposal is to outline a robust framework for measuring and managing the risk associated with our trading activities in real-time, leveraging both traditional and advanced risk measurement methodologies.

Background and Analysis Our recent study explored various risk measurement methods, including Historical VaR, Exponential Weighted Moving Average (EWMA) models, and GARCH models. Here are key findings and how they can be incorporated into our risk management framework:

1. **Historical VaR and EWMA**
   - **Performance**: Historical VaR, calculated using both a straightforward historical method and an exponential weighted approach, showed good responsiveness to market changes.
   - **Application**: Continue utilizing these measures but enhance them with real-time data updates to reflect the latest market conditions more accurately.
2. **GARCH Model**
   - **Insights**: The GARCH model proved effective in capturing the clustering of volatility, crucial for predicting future market volatility based on both recent and older data.
   - **Implementation**: Use GARCH models to estimate future volatility and adjust our risk thresholds accordingly. This approach will help us manage expectations during periods of high market volatility and stabilize risk estimates when the market is calm.
3. **Normalization of Gains**
   - **Technique**: By normalizing gains using the volatility estimates from our models, we were able to compare the distribution of normalized gains with original returns effectively.
   - **Utility**: Implement this normalization process in our real-time risk assessments to standardize risk measures across different trading conditions and ensure consistent risk perception.
4. **Confidence Interval Calculation**
   - **Methodology**: We applied parametric and bootstrap methods to compute confidence intervals for our VaR estimates, providing a range within which we expect our true VaR to lie.
   - **Real-Time Application**: Regularly update and calculate these confidence intervals in real-time to maintain an accurate measure of risk exposure.

Proposed Real-Time Risk Management Framework

1. **Real-Time Data Processing**: Leverage high-frequency data feeds to update risk metrics dynamically.
2. **Integrated Risk Dashboard**: Develop a dashboard that displays real-time analytics, including VaR metrics, volatility estimates, and normalized gains.

3. **Alert System**: Establish automated alerts for risk breaches based on updated VaR and confidence intervals. Adaptive Risk Models: Utilize adaptive algorithms that recalibrate risk models based on incoming data, enhancing prediction accuracy.

Justification

- **Comprehensive Risk Overview**: The integration of these models provides a multi-faceted view of risk that encompasses both the typical market conditions and the outliers, ensuring that no aspect of risk is overlooked.
- **Adaptability to Market Conditions**: By continuously updating our risk metrics based on the latest data, we can adapt more quickly to changing market conditions, potentially reducing losses and improving response strategies.
- **Regulatory Compliance**: This approach aligns with global regulatory requirements for risk management, potentially reducing regulatory concerns and enhancing our market reputation.

[ ]: