**Oliver Jamieson 21443945 Assessment 2 AI Conversation list**

Sure, here's a Python code snippet using the `pandas` library to investigate volatility in stock price history:

```python

import pandas as pd

import numpy as np

# Load stock price history from a CSV file

stock\_data = pd.read\_csv('stock\_price\_history.csv')

# Convert the 'Date' column to datetime format

stock\_data['Date'] = pd.to\_datetime(stock\_data['Date'])

# Calculate daily returns

stock\_data['Daily Return'] = stock\_data['Close'].pct\_change()

# Calculate volatility (standard deviation of daily returns)

volatility = stock\_data['Daily Return'].std()

print("Volatility (standard deviation of daily returns):", volatility)

```

Make sure to replace `'stock\_price\_history.csv'` with the filename containing your stock price data. This code will load the data, calculate the daily returns, and then compute the standard deviation of these returns, which represents the volatility of the stock.

Write a code to analyze how different levels of volatility impact the stability of stock prices, you can categorize the data into different volatility levels and then calculate statistics for each level. Here's a Python code snippet to do that:

```python

import pandas as pd

import numpy as np

# Load stock price history from a CSV file

stock\_data = pd.read\_csv('stock\_price\_history.csv')

# Convert the 'Date' column to datetime format

stock\_data['Date'] = pd.to\_datetime(stock\_data['Date'])

# Calculate daily returns

stock\_data['Daily Return'] = stock\_data['Close'].pct\_change()

# Define volatility levels

low\_volatility\_threshold = 0.01

high\_volatility\_threshold = 0.03

# Categorize volatility levels

stock\_data['Volatility Level'] = np.where(stock\_data['Daily Return'].abs() < low\_volatility\_threshold, 'Low',

np.where(stock\_data['Daily Return'].abs() < high\_volatility\_threshold, 'Medium', 'High'))

# Calculate stability statistics for each volatility level

stability\_stats = stock\_data.groupby('Volatility Level')['Close'].agg(['mean', 'std'])

print("Stability Statistics for Different Volatility Levels:")

print(stability\_stats)

```

In this code:

- We define two thresholds to categorize volatility into low, medium, and high levels.

- We categorize each day's volatility based on the daily return.

- Then, we group the data by volatility level and calculate statistics like mean and standard deviation of closing prices for each level.

Adjust the thresholds according to your definition of low, medium, and high volatility, and make sure to replace `'stock\_price\_history.csv'` with the filename containing your stock price data.

Certainly! We can simulate significant market events affecting stock prices on specific days and adjust the impact of these events to observe varying outcomes. Here's how you can do it:

```python

import pandas as pd

import numpy as np

# Load stock price history from a CSV file

stock\_data = pd.read\_csv('stock\_price\_history.csv')

# Convert the 'Date' column to datetime format

stock\_data['Date'] = pd.to\_datetime(stock\_data['Date'])

# Function to simulate market events

def simulate\_market\_events(stock\_data, event\_dates, impact\_factors):

for date, impact\_factor in zip(event\_dates, impact\_factors):

idx = stock\_data.index[stock\_data['Date'] == date][0]

stock\_data.at[idx, 'Close'] \*= impact\_factor

# Define market event dates and impact factors

event\_dates = ['2024-01-15', '2024-03-10', '2024-05-01'] # Example event dates

impact\_factors = [0.95, 1.1, 0.85] # Example impact factors (0.95 represents a 5% decrease, 1.1 represents a 10% increase, and 0.85 represents a 15% decrease)

# Simulate market events

simulate\_market\_events(stock\_data, event\_dates, impact\_factors)

# Calculate daily returns

stock\_data['Daily Return'] = stock\_data['Close'].pct\_change()

# Analyze the impact on volatility

volatility\_before\_event = stock\_data.loc[stock\_data['Date'] < event\_dates[0], 'Daily Return'].std()

volatility\_after\_event = stock\_data.loc[stock\_data['Date'] >= event\_dates[-1], 'Daily Return'].std()

print("Volatility before the first event:", volatility\_before\_event)

print("Volatility after the last event:", volatility\_after\_event)

```

In this code:

- We define `simulate\_market\_events()` function to apply the impact factors on stock prices for specified event dates.

- We specify the event dates and their corresponding impact factors.

- We simulate the market events by calling `simulate\_market\_events()`.

- Then, we calculate daily returns and analyze the impact on volatility before and after the events.

Adjust the event dates, impact factors, and any other parameters as needed for your analysis. This simulation allows you to observe how different market events with varying impacts affect the stock's volatility.

Certainly! Let's implement three basic trading strategies: "buy and hold", "moving average crossover", and "momentum-based" strategies. We'll then simulate these strategies and test their effectiveness over time.

1. \*\*Buy and Hold Strategy\*\*: Buy the stock at the beginning of the simulation and hold it throughout the entire period.

2. \*\*Moving Average Crossover Strategy\*\*: Buy the stock when the short-term moving average crosses above the long-term moving average, and sell when the short-term moving average crosses below the long-term moving average.

3. \*\*Momentum-Based Strategy\*\*: Buy the stock when its price momentum is positive (e.g., when the stock's return over the last N days is positive), and sell when momentum turns negative.

Let's implement these strategies in Python:

```python

import pandas as pd

import numpy as np

# Load stock price history from a CSV file

stock\_data = pd.read\_csv('stock\_price\_history.csv')

# Convert the 'Date' column to datetime format

stock\_data['Date'] = pd.to\_datetime(stock\_data['Date'])

# Function to simulate "buy and hold" strategy

def buy\_and\_hold\_strategy(stock\_data):

initial\_price = stock\_data.iloc[0]['Close']

final\_price = stock\_data.iloc[-1]['Close']

return (final\_price - initial\_price) / initial\_price

# Function to simulate moving average crossover strategy

def moving\_average\_crossover\_strategy(stock\_data, short\_window=50, long\_window=200):

signals = pd.DataFrame(index=stock\_data.index)

signals['Signal'] = 0.0

# Compute short and long-term moving averages

signals['Short\_MA'] = stock\_data['Close'].rolling(window=short\_window, min\_periods=1, center=False).mean()

signals['Long\_MA'] = stock\_data['Close'].rolling(window=long\_window, min\_periods=1, center=False).mean()

# Generate signals

signals['Signal'][short\_window:] = np.where(signals['Short\_MA'][short\_window:] > signals['Long\_MA'][short\_window:], 1.0, 0.0)

# Calculate returns based on signals

signals['Position'] = signals['Signal'].diff()

return (stock\_data['Close'][-1] - stock\_data['Close'][0]) / stock\_data['Close'][0]

# Function to simulate momentum-based strategy

def momentum\_based\_strategy(stock\_data, window=50):

signals = pd.DataFrame(index=stock\_data.index)

signals['Signal'] = 0.0

# Calculate price momentum

signals['Momentum'] = stock\_data['Close'].pct\_change(window=window)

# Generate signals

signals['Signal'][window:] = np.where(signals['Momentum'][window:] > 0, 1.0, 0.0)

# Calculate returns based on signals

signals['Position'] = signals['Signal'].diff()

return (stock\_data['Close'][-1] - stock\_data['Close'][0]) / stock\_data['Close'][0]

# Simulate and test the strategies

buy\_and\_hold\_return = buy\_and\_hold\_strategy(stock\_data)

print("Buy and Hold Strategy Return:", buy\_and\_hold\_return)

moving\_average\_crossover\_return = moving\_average\_crossover\_strategy(stock\_data)

print("Moving Average Crossover Strategy Return:", moving\_average\_crossover\_return)

momentum\_based\_return = momentum\_based\_strategy(stock\_data)

print("Momentum-Based Strategy Return:", momentum\_based\_return)

```

In this code:

- We define functions to simulate each trading strategy.

- For the moving average crossover and momentum-based strategies, we calculate signals based on specified conditions and compute returns accordingly.

- We simulate and test each strategy using historical stock price data.

You can adjust the parameters of each strategy (e.g., moving average window, momentum window) and analyze their effectiveness over different time periods or stocks.

To analyze how different levels of volatility impact the stability of stock prices and the potential for investment gains or losses using the provided stock market simulation, we can run multiple simulations with varying volatility levels and observe the outcomes. We can measure stability by looking at the variation in stock prices over time and potential gains or losses by comparing the final price with the initial price. Here's how you can do it:

```python

from simulacra.stock\_market\_simulation import StockMarketSimulation

import matplotlib.pyplot as plt

# Write a helper function

def plot\_prices(prices, event\_day=None):

"""

Plots the stock prices from a simulation with an optional vertical line marking a major market event.

Parameters:

prices (List[float]): A list of stock prices to be plotted.

event\_day (Optional[int]): The day on which the major market event occurs (defaults to None).

Returns:

None

"""

plt.figure(figsize=(10, 6))

plt.plot(prices, label='Stock Price')

if event\_day is not None:

plt.axvline(x=event\_day, color='red', linestyle='--', label='Major Market Event')

plt.xlabel('Days')

plt.ylabel('Price ($)')

plt.title('Stock Market Simulation')

plt.legend()

plt.show()

# Function to run simulation with different volatility levels

def run\_simulation\_with\_varying\_volatility(start\_price, days, volatility\_list, event\_day, event\_impact):

results = []

for volatility in volatility\_list:

sim = StockMarketSimulation(start\_price=start\_price, days=days, volatility=volatility,

drift=-0.001, event\_day=event\_day, event\_impact=event\_impact)

prices = sim.run\_simulation()

results.append(prices)

return results

# Function to analyze stability and potential gains or losses

def analyze\_results(results):

for i, prices in enumerate(results):

initial\_price = prices[0]

final\_price = prices[-1]

stability = np.std(prices)

gain\_or\_loss = (final\_price - initial\_price) / initial\_price

print(f"Volatility Level {i+1}:")

print(f"Stability (Standard Deviation of Prices): {stability}")

print(f"Potential Gain or Loss: {gain\_or\_loss}\n")

# Define parameters

start\_price = 100

days = 365

volatility\_list = [0.01, 0.03, 0.05] # Example volatility levels to test

event\_day = 100

event\_impact = -0.2

# Run simulation with different volatility levels

results = run\_simulation\_with\_varying\_volatility(start\_price, days, volatility\_list, event\_day, event\_impact)

# Analyze the results

analyze\_results(results)

# Visualize the results

for i, prices in enumerate(results):

plot\_prices(prices, event\_day)

```

In this code:

- We define a function `run\_simulation\_with\_varying\_volatility()` to run simulations with different volatility levels.

- We calculate stability by computing the standard deviation of stock prices and potential gains or losses by comparing initial and final prices.

- We then print and visualize the results for each volatility level.

Adjust the parameters as needed, such as the number of days, event day, event impact, and volatility levels, to further explore their effects on stability and potential gains or losses.s