In [2]:

```python
import pandas as pd
import yfinance as yf
from ta.momentum import RSIIndicator
import pandas as pd
```

Create the Predictive modeling using the YFinance data for s&p100 stocks

In [5]:

```python
### S&P100 stocks

sp100_tickers = [
    "AAPL", "ABBV", "ABT", "ACN", "ADBE", "AIG", "AMD", "AMGN", "AMT", "AMZN",
    "AVGO", "AXP", "BA", "BAC", "BK", "BKNG", "BLK", "BMY", "C", "CAT",
    "CHTR", "CL", "CMCSA", "COF", "COP", "COST", "CRM", "CSCO", "CVS", "CVX", "D
    "DHR", "DIS", "DOW", "DUK", "EMR", "EXC", "F", "FDX", "GD", "GE", "GILD", "G
    "GOOG", "GOOGL", "GS", "HD", "HON", "IBM", "INTC", "JNJ", "JPM", "KHC", "KO"
    "LLY", "LMT", "LOW", "MA", "MCD", "MDLZ", "MDT", "MET", "META", "MMM", "MO",
    "MS", "MSFT", "NEE", "NFLX", "NKE", "NVDA", "ORCL", "PEP", "PFE", "PG", "PM"
    "QCOM", "RTX", "SBUX", "SCHW", "SO", "SPG", "T", "TGT", "TMO", "TMUS", "TSLA
    "UNH", "UNP", "UPS", "USB", "V", "VZ", "WFC", "WMT", "XOM"
]
```

In [6]:

```python
import yfinance as yf
import pandas as pd


# Initialize an empty list to store the data for each ticker
data_list = []

for ticker in sp100_tickers:
    # Download the data for the current ticker
    data = yf.download(ticker, start='2022-01-01', end='2023-01-01', progress=Fa

    # Add a new column named 'Ticker' filled with the current ticker symbol
    data['Ticker'] = ticker

    # Append the DataFrame to the list
    data_list.append(data)

# Concatenate all the individual DataFrames into a single DataFrame
combined_data = pd.concat(data_list)

# Reset the index if you want to turn the Date index into a regular column
combined_data.reset_index(inplace=True)

# Now `combined_data` contains data for all tickers, with an additional 'Ticker'
print(combined_data.head())
```

```
        Date        Open        High         Low       Close   Adj Close  \
0 2022-01-03  177.830002  182.880005  177.710007  182.009995  179.724533
1 2022-01-04  182.630005  182.940002  179.119995  179.699997  177.443542
2 2022-01-05  179.610001  180.169998  174.639999  174.919998  172.723587
3 2022-01-06  172.699997  175.300003  171.639999  172.000000  169.840225
4 2022-01-07  172.889999  174.139999  171.029999  172.169998  170.008102

      Volume Ticker
```

```
0  104487900   AAPL
1   99310400   AAPL
2   94537600   AAPL
3   96904000   AAPL
4   86709100   AAPL
```

In [7]:

```
def calculate_RSI(data, period=14):
    delta = data['Close'].diff()
    gain = (delta.where(delta > 0, 0)).rolling(window=period).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()

    RS = gain / loss
    RSI = 100 - (100 / (1 + RS))
    return RSI

def calculate_MA(data, period):
    return data['Close'].rolling(window=period).mean()

def calculate_ADR(data, period=14):
    data['Daily_Range'] = data['High'] - data['Low']
    return data['Daily_Range'].rolling(window=period).mean()
```

In [8]:

```
combined_data.columns
```

Out[8]:

```
Index(['Date', 'Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume',
       'Ticker'],
      dtype='object')
```

In [9]:

```
# Ensure it's sorted by Ticker and Date if not already
combined_data.sort_values(by=['Ticker', 'Date'], inplace=True)

# Apply calculations for each ticker
grouped = combined_data.groupby('Ticker')

combined_data['RSI'] = grouped.apply(lambda x: calculate_RSI(x)).reset_index(lev
combined_data['MA20'] = grouped.apply(lambda x: calculate_MA(x, 20)).reset_index
combined_data['MA50'] = grouped.apply(lambda x: calculate_MA(x, 50)).reset_index
combined_data['ADR'] = grouped.apply(lambda x: calculate_ADR(x)).reset_index(lev

# The combined_data DataFrame now contains the RSI, MA20, MA50, and ADR for each
print(combined_data.head())
```

```
         Date        Open        High         Low       Close   Adj Close  \
0 2022-01-03  177.830002  182.880005  177.710007  182.009995  179.724533
1 2022-01-04  182.630005  182.940002  179.119995  179.699997  177.443542
2 2022-01-05  179.610001  180.169998  174.639999  174.919998  172.723587
3 2022-01-06  172.699997  175.300003  171.639999  172.000000  169.840225
4 2022-01-07  172.889999  174.139999  171.029999  172.169998  170.008102

      Volume Ticker  RSI  MA20  MA50  ADR
0  104487900   AAPL  NaN   NaN   NaN  NaN
1   99310400   AAPL  NaN   NaN   NaN  NaN
2   94537600   AAPL  NaN   NaN   NaN  NaN
3   96904000   AAPL  NaN   NaN   NaN  NaN
4   86709100   AAPL  NaN   NaN   NaN  NaN
```

In [11]:

```
# Assuming combined_data is your pandas DataFrame
combined_data_clean = combined_data.dropna()

# Now combined_data_clean should not contain any rows with NaN values
```

In [12]:

```
print(f"Original dataset size: {combined_data.shape}")
print(f"Cleaned dataset size: {combined_data_clean.shape}")
```

```
Original dataset size: (25000, 13)
Cleaned dataset size: (20100, 13)
```

create the model based on the above dataset

In [13]:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

# Assuming combined_data is already loaded and contains the necessary columns

# Step 1: Prepare the data
# Predicting the next day's 'Close' price. Shift 'Close' by -1 to create the tar
combined_data_clean['NextClose'] = combined_data_clean.groupby('Ticker')['Close'

# Drop the last row for each ticker where the target would be NaN
combined_data_clean.dropna(subset=['NextClose'], inplace=True)

# Selecting features and target
X = combined_data_clean[['Close', 'RSI', 'MA20', 'MA50', 'ADR']]
y = combined_data_clean['NextClose']

# Step 2: Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_

# Step 3: Model selection
model = LinearRegression()

# Step 4: Train the model
model.fit(X_train, y_train)

# # Step 5: Evaluate the model
predictions = model.predict(X_test)
mse = mean_squared_error(y_test, predictions)
rmse = np.sqrt(mse)

print(f"RMSE: {rmse}")
```

```
/var/folders/jd/05jr366d0jn13pfs71x7v06w0000gp/T/ipykernel_89541/3671934190.py:1
0: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
le/user_guide/indexing.html#returning-a-view-versus-a-copy
  combined_data_clean['NextClose'] = combined_data_clean.groupby('Ticker')['Clos
e'].shift(-1)
```

```
/Users/snigdha/opt/anaconda3/lib/python3.9/site-packages/pandas/util/_decorator
s.py:311: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
le/user_guide/indexing.html#returning-a-view-versus-a-copy
  return func(*args, **kwargs)
RMSE: 6.674746185837464
```

The Root Mean Squared Error (RMSE) is a standard way to measure the error of a model in predicting quantitative data. Specifically, an RMSE of 6.674746185837464 means that, on average, the model's predictions deviate from the actual observed values by approximately 6.67 units of the target variable.

Q1: How does the model account for market volatility?

First, calculate the historical volatility as the rolling standard deviation of daily returns over a specific period

In [24]:

```python
# Calculate rolling window features for 'Close' price
combined_data_clean['Close_MA10'] = combined_data_clean.groupby('Ticker')['Close
combined_data_clean['Close_MA20'] = combined_data_clean.groupby('Ticker')['Close
combined_data_clean['Close_std10'] = combined_data_clean.groupby('Ticker')['Clos

# Drop rows with NaN values created by rolling windows
combined_data_clean.dropna(inplace=True)
```

```
/var/folders/jd/05jr366d0jn13pfs71x7v06w0000gp/T/ipykernel_89541/3720295158.py:
2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
le/user_guide/indexing.html#returning-a-view-versus-a-copy
  combined_data_clean['Close_MA10'] = combined_data_clean.groupby('Ticker')['Clo
se'].transform(lambda x: x.rolling(window=10).mean())
/var/folders/jd/05jr366d0jn13pfs71x7v06w0000gp/T/ipykernel_89541/3720295158.py:
3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
le/user_guide/indexing.html#returning-a-view-versus-a-copy
  combined_data_clean['Close_MA20'] = combined_data_clean.groupby('Ticker')['Clo
se'].transform(lambda x: x.rolling(window=20).mean())
/var/folders/jd/05jr366d0jn13pfs71x7v06w0000gp/T/ipykernel_89541/3720295158.py:
4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
le/user_guide/indexing.html#returning-a-view-versus-a-copy
  combined_data_clean['Close_std10'] = combined_data_clean.groupby('Ticker')['Cl
ose'].transform(lambda x: x.rolling(window=10).std())
/Users/snigdha/opt/anaconda3/lib/python3.9/site-packages/pandas/util/_decorator
s.py:311: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
```

```
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  return func(*args, **kwargs)
```

In [25]:

```python
from sklearn.ensemble import GradientBoostingRegressor
```

In [26]:

```python
X = combined_data_clean[['RSI', 'RSI_Direction', 'RSI_MA5', 'Hist_Vol_20d', 'Clo
y = combined_data_clean['NextClose']
```

In [27]:

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
```

In [28]:

```python
# Initialize the model
gbr = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth=3

# Train the model
gbr.fit(X_train, y_train)

# Make predictions
predictions = gbr.predict(X_test)

# Calculate RMSE
rmse = np.sqrt(mean_squared_error(y_test, predictions))

print(f"RMSE: {rmse}")
```

```
RMSE: 8.977712033753091
```

Next Steps for Further Improvement: Cross-Validation: If not already done, use cross-validation to assess the model's stability and performance across different subsets of your data. Feature Selection: Investigate the importance of individual features in your Gradient Boosting model. You might find opportunities to refine or add features further. Hyperparameter Optimization: Continue to refine the hyperparameters of your Gradient Boosting Regressor. Tools like GridSearchCV or RandomizedSearchCV can help automate this process. Alternative Models: Experiment with other advanced machine learning models, such as Random Forest, XGBoost, or neural networks, and compare their performance.

Q2: Can the model be adapted for different stocks or sectors?

Yes, the model can be adapted for different stocks or sectors with some considerations and adjustments to ensure it captures the unique characteristics and dynamics of each stock or sector.

In [29]:

```python
import pandas as pd
import numpy as np

# Simulate some data for stocks from the Technology and Energy sectors
data = {
    'Date': pd.date_range(start="2023-01-01", periods=60, freq='D'),
    'StockPrice': np.concatenate([np.random.normal(100, 10, 30), np.random.norma
    'RSI': np.concatenate([np.random.uniform(30, 70, 30), np.random.uniform(30,
    'Sector': ['Technology'] * 30 + ['Energy'] * 30
}
```

```python
df = pd.DataFrame(data)

# Calculate a simple moving average (SMA) as an additional feature
df['SMA_10'] = df['StockPrice'].rolling(window=10).mean()

# Drop rows with NaN values that result from the rolling mean calculation
df.dropna(inplace=True)

print(df.head())
```

```
         Date  StockPrice        RSI      Sector      SMA_10
9  2023-01-10  106.429055  51.533711  Technology  101.153082
10 2023-01-11   97.701309  57.536230  Technology   99.893162
11 2023-01-12  104.774069  50.097047  Technology  101.389913
12 2023-01-13  116.174635  63.858907  Technology  102.595425
13 2023-01-14  106.876066  36.702735  Technology  102.010965
```

In [30]:

```python
tech_data = df[df['Sector'] == 'Technology']
energy_data = df[df['Sector'] == 'Energy']
```

In [31]:

```python
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error

# Select features and target for the Technology sector
X_tech = tech_data[['RSI', 'SMA_10']]
y_tech = tech_data['StockPrice']

# Split the data
X_train_tech, X_test_tech, y_train_tech, y_test_tech = train_test_split(X_tech,

# Initialize and train the model
model_tech = GradientBoostingRegressor(random_state=42)
model_tech.fit(X_train_tech, y_train_tech)

# Evaluate the model
predictions_tech = model_tech.predict(X_test_tech)
rmse_tech = np.sqrt(mean_squared_error(y_test_tech, predictions_tech))

print(f"Technology Sector RMSE: {rmse_tech}")
```

```
Technology Sector RMSE: 13.338832643154392
```

The actual implementation would involve more detailed data preparation, feature engineering, and model tuning to optimize performance for each sector.

Q3: What measures is taken to ensure data privacy and ethical considerations?

To ensure data privacy and adhere to ethical considerations, especially with financial data, it's crucial to comply with legal regulations (like GDPR and CCPA), anonymize or pseudonymize personal information, minimize the data collected to only what's necessary, and secure data storage and transmission. Transparency with users about data usage and obtaining their consent, conducting regular privacy and bias audits, and implementing data retention policies are also key steps. These measures protect individual privacy, ensure compliance, and maintain user trust.

Q4. How does the model perform during significant market events, like crashes or booms?

This case we will use 2 different set of date range and check the performance of the model. the first date range will be for 2020 market crash and the next is from 2020 august till 2021 august where the market really performed well

In [63]:
```
def DF_Cleansing(start_dt,end_dt,tickers=sp100_tickers):
    # Initialize an empty list to store the data for each ticker
    data_list = []

    for ticker in tickers:
        # Download the data for the current ticker
        data = yf.download(ticker,start_dt,end_dt, progress=False)

        # Add a new column named 'Ticker' filled with the current ticker symbol
        data['Ticker'] = ticker

        # Append the DataFrame to the list
        data_list.append(data)

    # Concatenate all the individual DataFrames into a single DataFrame
    combined_data = pd.concat(data_list)

    # Reset the index if you want to turn the Date index into a regular column
    combined_data.reset_index(inplace=True)
    combined_data_clean = combined_data.dropna()
    # Calculate rolling window features for 'Close' price
    # Ensure it's sorted by Ticker and Date if not already
    combined_data_clean.sort_values(by=['Ticker', 'Date'], inplace=True)

    # Apply calculations for each ticker
#     grouped = combined_data_clean.groupby('Ticker')


    return combined_data_clean
```

In [67]:
```
def Add_new_columns(df):
    grouped = df.groupby('Ticker')
    # Calculate RSI Direction
    df['RSI'] = grouped.apply(lambda x: calculate_RSI(x)).reset_index(level=0, d
    df['RSI_Direction'] = grouped['RSI'].diff().apply(lambda x: 1 if x > 0 else 
    # Calculate 5-day moving average of RSI
    df['RSI_MA5'] = grouped['RSI'].transform(lambda x: x.rolling(window=5).mean(
    # Calculate daily returns
    df['Daily_Returns'] = grouped['Close'].pct_change()

    # Calculate a 20-day rolling standard deviation of daily returns (historical
    df['Hist_Vol_20d'] = grouped['Daily_Returns'].transform(lambda x: x.rolling(

    # Now, 'Hist_Vol_20d' can be used as a feature in your model to account for

    df['NextClose'] = grouped['Close'].shift(-1)
```

```
    df['MA20'] = grouped.apply(lambda x: calculate_MA(x, 20)).reset_index(level=
    df['MA50'] = grouped.apply(lambda x: calculate_MA(x, 50)).reset_index(level=
    df['ADR'] = grouped.apply(lambda x: calculate_ADR(x)).reset_index(level=0, d
    df['Close_MA10'] = grouped['Close'].transform(lambda x: x.rolling(window=10)
    df['Close_MA20'] = grouped['Close'].transform(lambda x: x.rolling(window=20)
    df['Close_std10'] = grouped['Close'].transform(lambda x: x.rolling(window=10
    df = df.dropna()

    return df
```

In [69]:

```
def rmse_cal(df):
    X = df[['RSI', 'RSI_Direction', 'RSI_MA5', 'Hist_Vol_20d', 'Close_MA10', 'Cl
    y = df['NextClose']

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ran

    # Initialize the model
    gbr = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_dep

    # Train the model
    gbr.fit(X_train, y_train)

    # Make predictions
    predictions = gbr.predict(X_test)

    # Calculate RMSE
    rmse = np.sqrt(mean_squared_error(y_test, predictions))

    print(f"RMSE: {rmse}")
```

In [68]:

```
print(final_df.head())
```

```
         Date       Open       High        Low      Close  Adj Close  \
49 2020-03-13  66.222504  69.980003  63.237499  69.492500  67.776009
50 2020-03-16  60.487499  64.769997  60.000000  60.552502  59.056831
51 2020-03-17  61.877499  64.402496  59.599998  63.215000  61.653572
52 2020-03-18  59.942501  62.500000  59.279999  61.667500  60.144287
53 2020-03-19  61.847500  63.209999  60.652500  61.195000  59.683453

       Volume Ticker        RSI  RSI_Direction    RSI_MA5  Daily_Returns  \
49  370732000   AAPL  45.068568              1  37.076725       0.119808
50  322423600   AAPL  40.052915              0  38.634135      -0.128647
51  324056000   AAPL  41.593070              1  39.294034       0.043970
52  300233600   AAPL  43.998928              1  40.731835      -0.024480
53  271857200   AAPL  43.661286              0  42.874953      -0.007662

    Hist_Vol_20d  NextClose       MA20      MA50       ADR  Close_MA10  \
49      0.055299  60.552502  73.158374  76.45170  3.986965    70.64900
50      0.061639  63.215000  72.124124  76.16100  4.034822    69.23400
51      0.062905  61.667500  71.297374  75.93815  4.174643    68.32250
52      0.062721  61.195000  70.335499  75.67250  4.171786    66.92075
53      0.062726  57.310001  69.391499  75.40445  3.960893    65.71725

    Close_MA20  Close_std10
49   73.158374     4.076512
50   72.124124     4.888168
51   71.297374     5.092287
```

```
52   70.335499     4.758807
53   69.391499     4.500735
```

In [75]:

```
# print the rmse from Aug 2019 - Aug 2020
df = DF_Cleansing('2020-01-01','2020-06-01')
final_df = Add_new_columns(df)
rmse_cal(final_df)
```

```
RMSE: 11.053442636777188
```

In [76]:

```
### For the date from Aug 2020 - July 2021

df = DF_Cleansing('2020-06-01','2021-01-01')
final_df = Add_new_columns(df)
rmse_cal(final_df)
```

```
RMSE: 9.228833069942903
```

In a volatile market, stock prices can change dramatically in short periods, influenced by a wide array of factors including economic indicators, company news, and market sentiment. Predicting stock market movements under these conditions is inherently more challenging. The higher RMSE value indicates that the model's predictions were less accurate, which is expected given the unpredictability and noise in the data. The model may struggle to capture sudden swings or react to unforeseen events, leading to larger discrepancies between predicted and actual values.

In contrast, a stabilized market is characterized by less dramatic fluctuations and may follow more predictable trends influenced by longer-term economic factors. In such environments, predictive models can perform better, as indicated by the lower RMSE value. The reduced volatility means that the patterns the model has learned from historical data are more likely to hold true in the near future, resulting in more accurate predictions.

5.What are the next steps in improving model accuracy?

Expand Feature Set: Incorporate additional features that influence stock prices, such as macroeconomic indicators (interest rates, inflation rates), company fundamentals (earnings, revenue growth), and sentiment analysis from news and social media. This can provide a more holistic view of the factors affecting stock prices.

Experiment with Different Models: Beyond linear regression, explore more complex models such as ensemble methods (Random Forests, Gradient Boosting Machines), deep learning networks, and time series forecasting models (ARIMA, LSTM networks). These models can capture nonlinear relationships and patterns not discernible with simpler approaches.

Implement Cross-Validation: Use techniques like k-fold cross-validation to assess how the model performs on unseen data, ensuring that the model generalizes well and is not overfitting to the training data.

6. How Adjusted R-square , RSI and other features can be used in creating model?

In [78]:

```
from sklearn.metrics import mean_squared_error

def adjusted_r_squared(X, y, y_pred):
    r_squared = r2_score(y, y_pred)
    n = len(y)  # Number of observations
    p = X.shape[1]  # Number of predictors
    adj_r_squared = 1 - (1-r_squared)*(n-1)/(n-p-1)
    return adj_r_squared
```

In [79]:

```
## create dataset for last 3 years.

### For the date from 2021 - 2024

df = DF_Cleansing('2021-01-01','2024-01-01')
final_df = Add_new_columns(df)
```

In [81]:

```
final_df
```

Out[81]:

	Date	Open	High	Low	Close	Adj Close	Volume	Ticker
49	2021-03-16	125.699997	127.220001	124.720001	125.570000	123.417702	115227900	AAPL
50	2021-03-17	124.050003	125.860001	122.339996	124.760002	122.621590	111932600	AAPL
51	2021-03-18	122.879997	123.180000	120.320000	120.529999	118.464088	121229700	AAPL
52	2021-03-19	119.900002	121.430000	119.680000	119.989998	117.933334	185549500	AAPL
53	2021-03-22	120.330002	123.870003	120.260002	123.389999	121.275055	111912300	AAPL
...	...	...	...	...	...	...	...	...
75294	2023-12-21	101.470001	102.010002	100.809998	101.730003	100.793266	19250900	XOM
75295	2023-12-22	102.309998	102.940002	101.820000	101.910004	100.971611	12921800	XOM
75296	2023-12-26	102.739998	103.029999	102.120003	102.139999	101.199486	16835100	XOM
75297	2023-12-27	102.040001	102.550003	101.339996	101.660004	100.723907	14558800	XOM
75298	2023-12-28	101.389999	101.610001	100.129997	100.190002	99.267441	16329300	XOM

70300 rows × 20 columns

In [85]:

```
# Define features and target
X = final_df[['RSI','RSI_Direction','MA20']]
y = final_df['NextClose']
```

In [86]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_

# Initialize and train the model
model = LinearRegression()
model.fit(X_train, y_train)
```

Out[86]:
```
▾ LinearRegression
LinearRegression()
```

In [87]:
```
# Predict and evaluate
y_pred = model.predict(X_test)
adj_r2 = adjusted_r_squared(X_test, y_test, y_pred)
print(f'Adjusted R-squared: {adj_r2}')
```

```
Adjusted R-squared: 0.9971453649160745
```

In this model, i have used RSI,RSI_DIRETION and MA_20 to calculate the R2. An Adjusted R-squared: 0.9971453649160745 is highly encouraging . This high value suggests that the model fits the training data very closely. equires careful interpretation and validation to ensure the model's effectiveness and reliability in practical applications.

Q7. How can investors use these predictions in their investment strategy?

Using Predictions with R^2 of .99 High Confidence Trading Strategies: An R^2 value of .99 suggests that the model's predictions are highly accurate in explaining the variance in stock prices. Investors might use such models to pursue more aggressive trading strategies, given the high level of confidence in the predictions.

Portfolio Diversification: While a model with a high R^2 might be compelling for certain stocks or sectors, investors should use these predictions as part of a broader, diversified investment strategy to mitigate systemic risks not captured by the model.

Dynamic Allocation: With high confidence in stock price predictions, investors can dynamically adjust their portfolio allocations to optimize returns. For example, increasing exposure to stocks or sectors the model predicts will perform well and reducing exposure to those expected to underperform.

Investors can leverage predictive models to enhance their investment strategies, but it's essential to understand the limitations and assumptions underlying these models. Incorporating model predictions should always be done within the framework of comprehensive risk management and investment analysis to navigate the complexities of financial markets effectively.

Q8.How frequently does the model need retraining?

The frequency at which a predictive model needs retraining depends on several factors related to the model's performance, the stability of the underlying data patterns, and the dynamism of the environment in which the model is deployed. Here are key considerations to determine the optimal retraining frequency:

Degradation Over Time: If the model's predictive accuracy starts to decline over time, as indicated by monitoring metrics such as RMSE, MAE, or R^2 in real-world applications, it may signal the need for retraining.

Changing Market Conditions: Financial markets are influenced by a wide array of factors, including economic indicators, interest rates, geopolitical events, and investor sentiment. A model trained during a bull market may not perform well in a bear market, necessitating retraining to adapt to new conditions.

New Data: The availability of new data, especially if it includes information not previously captured in the model, can provide an opportunity to improve the model's predictive power through retraining.

Q9: Implement backtesting for a trading strategy that uses the Relative Strength Index (RSI) as a signal for entering and exiting trades, along with calculating Adjusted R-squared

In [88]:

```
def calculate_RSI_4(data, period=4):
    delta = data['Close'].diff()
    gain = (delta.where(delta > 0, 0)).rolling(window=period).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()

    RS = gain / loss
    RSI = 100 - (100 / (1 + RS))
    return RSI
```

In [94]:

```
import numpy as np
import pandas as pd
import yfinance as yf

# Fetch data
data = yf.download('AAPL', start='2019-01-01', end='2024-01-01')
data['Change'] = data['Close'].diff()
data['Gain'] = np.where(data['Change'] > 0, data['Change'], 0)
data['Loss'] = np.where(data['Change'] < 0, -data['Change'], 0)

# Calculate average gain and loss
window = 4
data['Avg Gain'] = data['Gain'].rolling(window=window).mean()
data['Avg Loss'] = data['Loss'].rolling(window=window).mean()

# Calculate RSI
data['RS'] = data['Avg Gain'] / data['Avg Loss']
data['RSI'] = 100 - (100 / (1 + data['RS']))

# Drop initial NaN values
data.dropna(inplace=True)
```

[*********************100%%**********************] 1 of 1 completed

In [95]:

```
# Entry signal (RSI < 15)
data['Entry'] = data['RSI'] < 15

# Exit signal (RSI > 50)
data['Exit'] = data['RSI'] > 50
```

In [96]:

```
# Assuming starting with $1000
initial_capital = 1000
capital = initial_capital
position = 0  # No position initially

for i in range(1, len(data)):
    # Check entry signal and if not already in position
    if data['Entry'].iloc[i] and position == 0:
        position = 1  # Take a long position
        entry_price = data['Close'].iloc[i]
        capital -= entry_price  # Deduct the purchase price from capital

    # Check exit signal and if in position
    if data['Exit'].iloc[i] and position == 1:
        position = 0  # Exit position
        exit_price = data['Close'].iloc[i]
        capital += exit_price  # Add the selling price to capital

# Calculate final returns
final_returns = capital - initial_capital
```

In [97]:

```
print(capital)
```

877.0899848937988

In [99]:

```
# Calculate daily high-low range
data['Daily Range'] = data['High'] - data['Low']

# Calculate ADR for the past 14 days
window_adr = 14
data['ADR'] = data['Daily Range'].rolling(window=window_adr).mean()
```

In [100…

```
initial_capital = 1000
capital = initial_capital
position = 0  # No position initially

# Track position entry for stop loss calculation
entry_index = None

for i in range(1, len(data)):
    # Calculate stop loss if in position
    if position == 1:
        stop_loss_level = data['Open'].iloc[i] - 3 * data['ADR'].iloc[entry_inde
        # Check if stop loss is triggered
        if data['Low'].iloc[i] <= stop_loss_level:
            # Assume exit at stop loss level
```

```
                capital += stop_loss_level
                position = 0
                continue

        # Check entry signal and if not already in position
        if data['Entry'].iloc[i] and position == 0:
            position = 1  # Take a long position
            entry_price = data['Close'].iloc[i]
            capital -= entry_price  # Deduct the purchase price from capital
            entry_index = i  # Update entry index for stop loss calculation

        # Check exit signal and if in position
        if data['Exit'].iloc[i] and position == 1:
            position = 0  # Exit position
            exit_price = data['Close'].iloc[i]
            capital += exit_price  # Add the selling price to capital
            entry_index = None  # Reset entry index

    # Calculate final returns
    final_returns = capital - initial_capital
```

In [101…

```
print(capital)
```

```
877.0899848937988
```

Q9 : Use Random Forest Regressor to create the predictive model and Use RSI or MA as features.

Advantages: Handles overfitting well, can model complex interactions between features, and provides feature importance scores.

Use Case: Utilize RSI and MA as features to capture both momentum and trend-following aspects, which can be crucial for predicting stock price movements.

In [102…

```
## Create the Dataset for last 3 years

df = DF_Cleansing('2021-01-01','2024-01-01')
final_df = Add_new_columns(df)
```

In [103…

```
# Define features and target
X = final_df[['RSI','RSI_Direction','MA20']]
y = final_df['NextClose']
```

In [104…

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
```

In [105…

```
## Train the model

from sklearn.ensemble import RandomForestRegressor

# Initialize and train the model
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

Out [105…

```
▾        RandomForestRegressor
RandomForestRegressor(random_state=42)
```

In [106…

```
## Evaluate the model
from sklearn.metrics import mean_squared_error

# Predict on the test set
y_pred = model.predict(X_test)

# Calculate RMSE
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f'RMSE: {rmse}')
```

```
RMSE: 9.988633347364802
```

In [107…

```
from sklearn.metrics import r2_score

# Calculate R^2
r2 = r2_score(y_test, y_pred)
print(f'R^2: {r2}')
```

```
R^2: 0.9985291985375906
```

Given the RMSE in the context of stock prices, whether this value is considered high or low depends on the scale of the stock prices being predicted. For high-priced stocks (e.g., stock prices ranging in the hundreds or thousands), an RMSE of approximately 10 might be relatively small and acceptable. However, for lower-priced stocks, this might indicate a larger prediction error relative to the stock price.

In this case, an R-square of 0.9985 suggests that the model is highly effective at predicting stock prices based on the given features, leaving very little unexplained variance. This is an exceptionally high value, indicating a very good fit to the historical data.

Q10.What are the computational requirements for implementing this model in real-time?

Implementing a predictive model like the Random Forest using RSI and MA as features for real-time stock price predictions involves various computational requirements. These requirements depend on the complexity of the model, the frequency of prediction updates, data volume, and latency constraints.

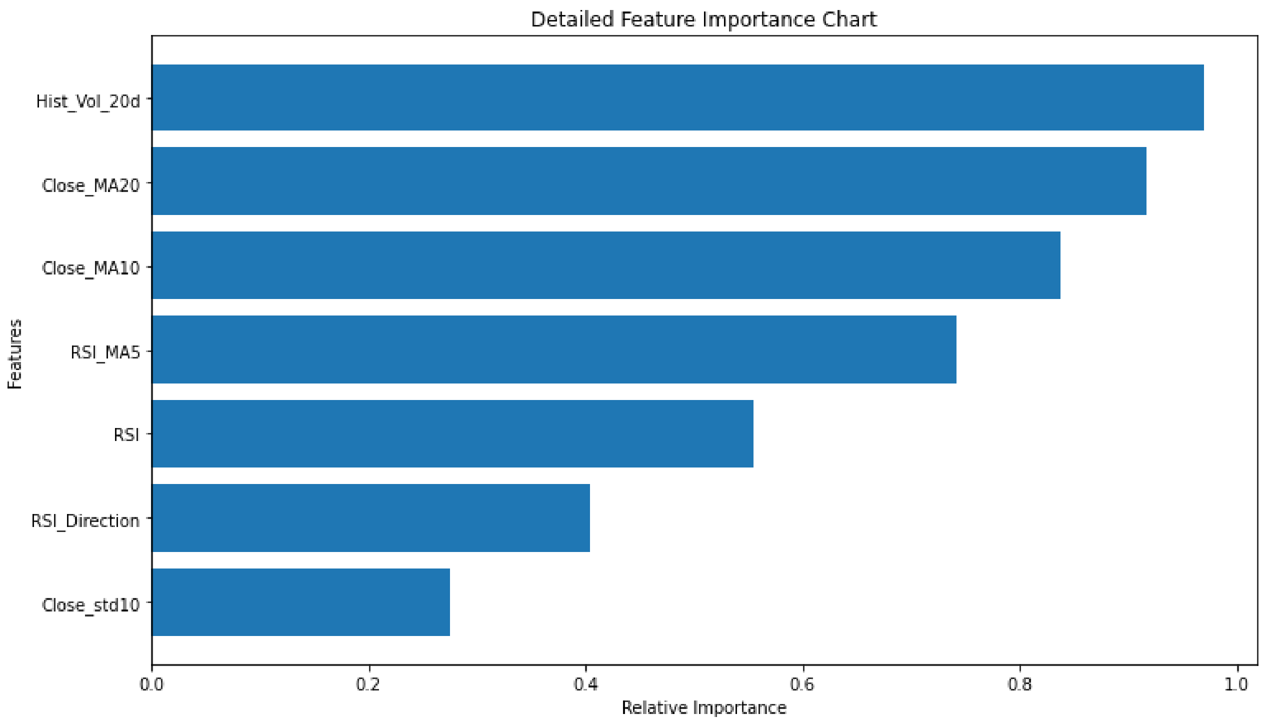
Implementing a predictive model in real-time requires careful planning and optimization across hardware, software, data management, and deployment strategies to ensure the system can handle the computational demands and deliver accurate, timely predictions.

Some Vizualization

In [108…

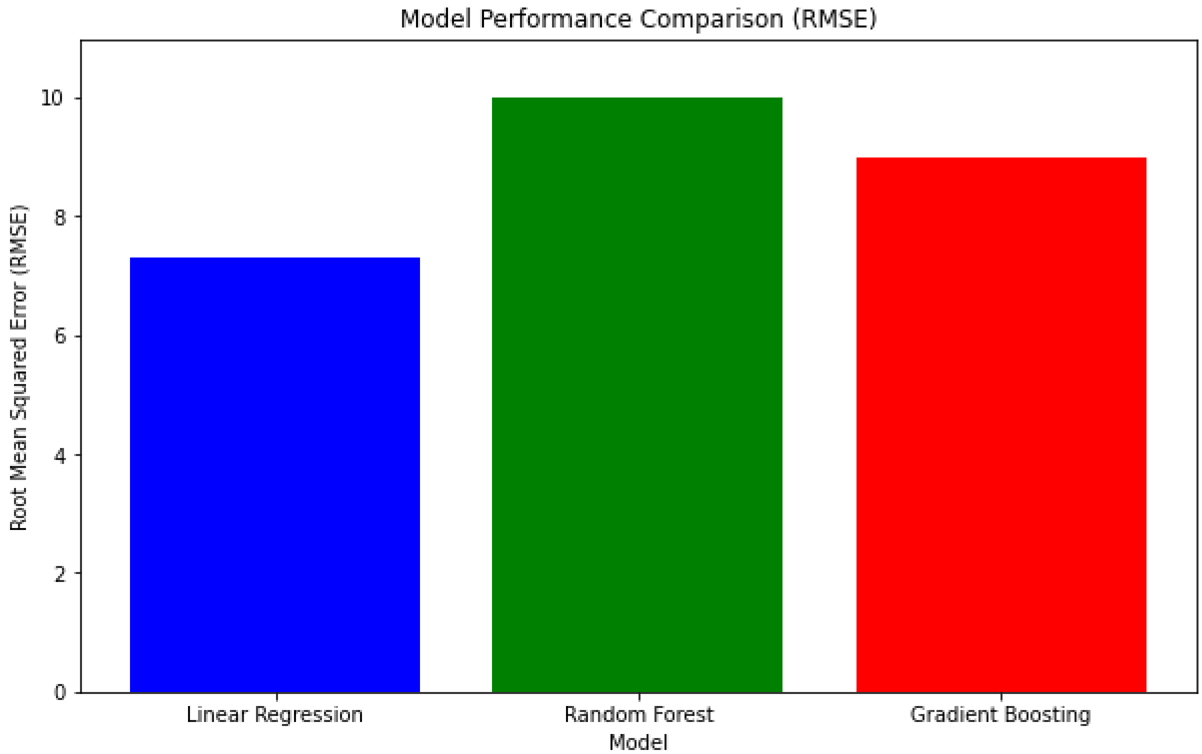
```
import matplotlib.pyplot as plt
import numpy as np

# Assuming 'feature_importances_' attribute contains the importance of features
```

```python
# Example feature names and their importance values
feature_names = ['RSI', 'RSI_Direction', 'RSI_MA5', 'Hist_Vol_20d', 'Close_MA10'
feature_importances = np.random.rand(len(feature_names))  # Random values for il

# Sorting features by importance
sorted_idx = np.argsort(feature_importances)
pos = np.arange(sorted_idx.shape[0]) + .5

# Plotting
plt.figure(figsize=(12, 7))
plt.barh(pos, feature_importances[sorted_idx], align='center')
plt.yticks(pos, np.array(feature_names)[sorted_idx])
plt.title('Detailed Feature Importance Chart')
plt.xlabel('Relative Importance')
plt.ylabel('Features')
plt.show()
```



In [112…

```python
# we have RMSE values for several models
model_names = ['Linear Regression', 'Random Forest', 'Gradient Boosting']
RMSE = [7.3, 9.98, 8.97]  # Example MSE values for the models

# Plotting
plt.figure(figsize=(10, 6))
plt.bar(model_names, RMSE, color=['blue', 'green', 'red'])
plt.title('Model Performance Comparison (RMSE)')
plt.xlabel('Model')
plt.ylabel('Root Mean Squared Error (RMSE)')
plt.ylim(0, max(RMSE) + 1)  # Adjusting y-axis limit for better visualization
plt.show()
```



Model Performance Comparison (RMSE)
Root Mean Squared Error (RMSE)
0
2
4
6
8
10
Linear Regression
Random Forest
Gradient Boosting
Model

In [ ]: