In [ ]:

```
!pip install yfinance
!pip install pandas_market_calendars
!pip install empyrical

import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
import empyrical as ep
from google.colab import drive
drive.mount("/content/gdrive", force_remount=True)

file_path = 'SP500_2023.csv'
```

```
Requirement already satisfied: yfinance in /usr/local/lib/python3.10/dist-packages (0.2.3
7)
Requirement already satisfied: pandas>=1.3.0 in /usr/local/lib/python3.10/dist-packages (
from yfinance) (2.0.3)
Requirement already satisfied: numpy>=1.16.5 in /usr/local/lib/python3.10/dist-packages (
from yfinance) (1.25.2)
Requirement already satisfied: requests>=2.31 in /usr/local/lib/python3.10/dist-packages
(from yfinance) (2.31.0)
Requirement already satisfied: multitasking>=0.0.7 in /usr/local/lib/python3.10/dist-pack
ages (from yfinance) (0.0.11)
Requirement already satisfied: lxml>=4.9.1 in /usr/local/lib/python3.10/dist-packages (fr
om yfinance) (4.9.4)
Requirement already satisfied: appdirs>=1.4.4 in /usr/local/lib/python3.10/dist-packages
(from yfinance) (1.4.4)
Requirement already satisfied: pytz>=2022.5 in /usr/local/lib/python3.10/dist-packages (f
rom yfinance) (2023.4)
Requirement already satisfied: frozendict>=2.3.4 in /usr/local/lib/python3.10/dist-packag
es (from yfinance) (2.4.1)
Requirement already satisfied: peewee>=3.16.2 in /usr/local/lib/python3.10/dist-packages
(from yfinance) (3.17.1)
Requirement already satisfied: beautifulsoup4>=4.11.1 in /usr/local/lib/python3.10/dist-p
ackages (from yfinance) (4.12.3)
Requirement already satisfied: html5lib>=1.1 in /usr/local/lib/python3.10/dist-packages (
from yfinance) (1.1)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/dist-packages (
from beautifulsoup4>=4.11.1->yfinance) (2.5)
Requirement already satisfied: six>=1.9 in /usr/local/lib/python3.10/dist-packages (from
html5lib>=1.1->yfinance) (1.16.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/dist-packages (f
rom html5lib>=1.1->yfinance) (0.5.1)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-p
ackages (from pandas>=1.3.0->yfinance) (2.8.2)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages
(from pandas>=1.3.0->yfinance) (2024.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist
-packages (from requests>=2.31->yfinance) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (f
rom requests>=2.31->yfinance) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packa
ges (from requests>=2.31->yfinance) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packa
ges (from requests>=2.31->yfinance) (2024.2.2)
Requirement already satisfied: pandas_market_calendars in /usr/local/lib/python3.10/dist-
packages (4.4.0)
Requirement already satisfied: pandas>=1.1 in /usr/local/lib/python3.10/dist-packages (fr
om pandas_market_calendars) (2.0.3)
Requirement already satisfied: pytz in /usr/local/lib/python3.10/dist-packages (from pand
as_market_calendars) (2023.4)
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.10/dist-packages
```

```
(from pandas_market_calendars) (2.8.2)
Requirement already satisfied: exchange-calendars>=3.3 in /usr/local/lib/python3.10/dist-
packages (from pandas_market_calendars) (4.5.3)
Requirement already satisfied: numpy<2 in /usr/local/lib/python3.10/dist-packages (from e
xchange-calendars>=3.3->pandas_market_calendars) (1.25.2)
Requirement already satisfied: pyluach in /usr/local/lib/python3.10/dist-packages (from e
xchange-calendars>=3.3->pandas_market_calendars) (2.2.0)
Requirement already satisfied: toolz in /usr/local/lib/python3.10/dist-packages (from exc
hange-calendars>=3.3->pandas_market_calendars) (0.12.1)
Requirement already satisfied: tzdata in /usr/local/lib/python3.10/dist-packages (from ex
change-calendars>=3.3->pandas_market_calendars) (2024.1)
Requirement already satisfied: korean-lunar-calendar in /usr/local/lib/python3.10/dist-pa
ckages (from exchange-calendars>=3.3->pandas_market_calendars) (0.3.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from
python-dateutil->pandas_market_calendars) (1.16.0)
Requirement already satisfied: empyrical in /usr/local/lib/python3.10/dist-packages (0.5.
5)
Requirement already satisfied: numpy>=1.9.2 in /usr/local/lib/python3.10/dist-packages (f
rom empyrical) (1.25.2)
Requirement already satisfied: pandas>=0.16.1 in /usr/local/lib/python3.10/dist-packages
(from empyrical) (2.0.3)
Requirement already satisfied: scipy>=0.15.1 in /usr/local/lib/python3.10/dist-packages (
from empyrical) (1.11.4)
Requirement already satisfied: pandas-datareader>=0.2 in /usr/local/lib/python3.10/dist-p
ackages (from empyrical) (0.10.0)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-p
ackages (from pandas>=0.16.1->empyrical) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (f
rom pandas>=0.16.1->empyrical) (2023.4)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages
(from pandas>=0.16.1->empyrical) (2024.1)
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from pand
as-datareader>=0.2->empyrical) (4.9.4)
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.10/dist-package
s (from pandas-datareader>=0.2->empyrical) (2.31.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from
python-dateutil>=2.8.2->pandas>=0.16.1->empyrical) (1.16.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist
-packages (from requests>=2.19.0->pandas-datareader>=0.2->empyrical) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (f
rom requests>=2.19.0->pandas-datareader>=0.2->empyrical) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packa
ges (from requests>=2.19.0->pandas-datareader>=0.2->empyrical) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packa
ges (from requests>=2.19.0->pandas-datareader>=0.2->empyrical) (2024.2.2)
Mounted at /content/gdrive
```

Data Collection

Removing the unwanted data fields like start, ending, name date and name end date.

- **start: The start date of the reporting period, typically indicating the beginning of the month. This date signifies when the data recording for the specific period commences.**
- **ending: The end date of the reporting period, usually the end of the month. This marks the conclusion of data capture for that period.**
- **namedt: The date on which the current name of the security was first recorded. This is useful for tracking changes in company names over time.**
- **nameendt: The end date for the usage of the security's name as recorded. After this date, the name may have changed.**

Converting Date fields to pandas datetime.

In [ ]:

```
monthly_data = pd.read_csv(file_path)
monthly_data.drop(['start', 'ending','namedt','nameendt'], axis=1, inplace=True)
monthly_data['date'] = pd.to_datetime(monthly_data['date'])
```

After checking the data, I found that ***date*** field often represented ***end of the collection period.*** So assuming it as end of month, ***calculating the start date of the month*** , later to be referred as start of the perdiod.

In [ ]:

```
monthly_data['start'] = monthly_data['date'].apply(lambda d: d.replace(day=1))
monthly_data.rename(columns={'date': 'ending'}, inplace=True)
```

In [ ]:

```
monthly_data.head()
```

Out[ ]:

	permno	ending	ret	mcap	comnam	ncusip	shrcd	exchcd	hsiccd	ticker	start
0	11308	2000-01-31	-0.013949	1.418695e+08	COCA COLA CO	19121610	11	1	2086	KO	2000-01-01
1	59408	2000-01-31	-0.034869	8.269172e+07	BANK OF AMERICA CORP	06050510	11	1	6029	BAC	2000-01-01
2	15667	2000-01-31	0.199045	6.149450e+07	AMERICAN HOME PRODUCTS CORP	02660910	11	1	2834	AHP	2000-01-01
3	75857	2000-01-31	-0.236531	2.152104e+07	SOLECTRON CORP	83418210	11	1	3672	SLR	2000-01-01
4	76597	2000-01-31	-0.188235	2.813018e+06	NABISCO GROUP HOLDING CORP	62952P10	11	1	2052	NGH	2000-01-01

Fetching and Aggregating Stock Data

Function: `fetch_daily_data`

This function is designed to ***retrieve daily stock data*** for a specific ticker symbol ***within a given date range***. We adjust the end date by one day to ensure the inclusion of end date's data in our fetch. The data is sourced using the `yfinance` library, which ***pulls historical market data directly from Yahoo Finance.*** For each fetched data frame, we add metadata such as the start and end dates, and the ticker symbol itself, and then ***select relevant columns such as 'High', 'Low', 'Close', 'Volume', and the added metadata columns.***

Function: `aggregate_daily_to_monthly`

After collecting daily stock data, this function ***aggregates the daily figures into monthly summaries.*** It calculates the ***maximum monthly high, minimum monthly low, average daily volume, total volume, and average closing price for the month***. If the daily data frame is empty (indicating no trading data was available), the function returns `None` to avoid errors in further processing.

Data Aggregation Loop

The loop iterates through each row in the `monthly_data` DataFrame, which contains columns specifying the ticker symbol and the start and end dates for each month of interest. ***For each row, it uses the*** *`fetch_daily_data`* ***function to get daily data and then aggregates that data into monthly statistics using*** *`aggregate_daily_to_monthly`*. The results are stored in a list, which is then converted to a DataFrame and saved as a CSV file. This CSV file contains the aggregated monthly data for further analysis or reporting.

This process allows us to systematically analyze stock performance over monthly intervals, providing a structured approach to explore trends and make informed trading decisions.

In [ ]:

```
monthly_data['start'] = pd.to_datetime(monthly_data['start'])
monthly_data['ending'] = pd.to_datetime(monthly_data['ending'])

def fetch_daily_data(ticker, start_date, end_date):
```

```
    adjusted_end_date = end_date + pd.Timedelta(days=1)
    stock = yf.Ticker(ticker)
    daily_data = stock.history(start=start_date, end=adjusted_end_date)
    daily_data['Start_date'] = start_date
    daily_data['End_date'] = end_date
    daily_data['Ticker'] = ticker
    return daily_data[['High', 'Low', 'Close', 'Volume', 'Start_date', 'End_date','Ticke
r']]

def aggregate_daily_to_monthly(daily_data):
    if daily_data.empty:
        return None
    else:
        return {
            'Monthly_High': daily_data['High'].max(),
            'Monthly_Low': daily_data['Low'].min(),
            'Average_Volume': daily_data['Volume'].mean(),
            'Total_Volume': daily_data['Volume'].sum(),
            'Average_Close': daily_data['Close'].mean(),
            'Start_date': daily_data['Start_date'].iloc[0],
            'End_date': daily_data['End_date'].iloc[0],
            'Ticker': daily_data['Ticker'].iloc[0]
        }

results = []
for index, row in monthly_data.iterrows():
    daily_data = fetch_daily_data(row['ticker'], row['start'], row['ending'])
    monthly_aggregate = aggregate_daily_to_monthly(daily_data)
    if monthly_aggregate:
      results.append(monthly_aggregate)

results_df = pd.DataFrame(results)
results_df.to_csv('/content/gdrive/My Drive/Aggregate_data.csv', index=False)
```

Saving the temporary results for later reference

In [ ]:

```
results_df = pd.read_csv('/content/gdrive/My Drive/Aggregate_r2_data.csv')
```

In [ ]:

```
monthly_data.columns
```

Out[ ]:

```
Index(['permno', 'ending', 'ret', 'mcap', 'comnam', 'ncusip', 'shrcd',
       'exchcd', 'hsiccd', 'ticker', 'start'],
      dtype='object')
```

In [ ]:

```
results_df.columns
```

Out[ ]:

```
Index(['Monthly_High', 'Monthly_Low', 'Average_Volume', 'Total_Volume',
       'Average_Close', 'Start_date', 'End_date', 'Ticker'],
      dtype='object')
```

Data Transformation and Merging

Date Conversion and Renaming Columns

Firstly, the `start` and `ending` columns in the `monthly_data` DataFrame are converted from string format to `datetime` objects to facilitate date operations. Simultaneously, we rename these columns to `Start_date` and `End_date` for clarity and consistency. Additionally, the `ticker` column is renamed to `Ticker`, aligning the column names across different DataFrames for easier data manipulation and merging.

Merging DataFrames

The `results_df` DataFrame, which contains aggregated data, also has its date columns (`Start_date` and `End_date`) converted to `datetime`. This ensures that the types match between `monthly_data` and `results_df` for a successful merge. We then perform a left join between `monthly_data` and `results_df` based on the columns `Ticker` , `Start_date` , and `End_date` . This operation integrates our detailed monthly data with the aggregated results, enriching our dataset with both detailed and summary information.

Removing Duplicates and Handling Missing Data

After merging, we remove any duplicate rows that may have occurred due to the join operation, based on the `Ticker` , `Start_date` , and `End_date` columns. This ensures that each row in our DataFrame represents a unique month, ticker, and set of data points. Next, we drop rows containing any NaN values to maintain data integrity, leaving us with a clean dataset ready for further analysis or visualization.

Displaying Data

Finally, we use the `.head()` method to display the first few rows of the cleaned and merged DataFrame. This provides a quick snapshot of our processed data, allowing for a preliminary check of the merging and cleaning operations.

This workflow is essential for ensuring that our data is accurate, consistent, and ready for in-depth analysis or reporting on stock performance over time.

In [ ]:

```
monthly_data['start'] = pd.to_datetime(monthly_data['start'])
monthly_data['ending'] = pd.to_datetime(monthly_data['ending'])
monthly_data.rename(columns={'start': 'Start_date', 'ending': 'End_date', 'ticker': 'Tic
ker'}, inplace=True)

results_df['Start_date'] = pd.to_datetime(results_df['Start_date'])
results_df['End_date'] = pd.to_datetime(results_df['End_date'])

merged_df = pd.merge(monthly_data, results_df, on=['Ticker', 'Start_date', 'End_date'],
how='left')
unique_rows_df = merged_df.drop_duplicates(subset=['Ticker', 'Start_date', 'End_date'])
df = unique_rows_df.dropna()
df.head()
```

Out[ ]:

	permno	End_date	ret	mcap	comnam	ncusip	shrcd	exchcd	hsiccd	Ticker	Start_date	Monthly_l
0	11308	2000-01-31	-0.013949	1.418695e+08	COCA COLA CO	19121610	11	1	2086	KO	2000-01-01	17.122
1	59408	2000-01-31	-0.034869	8.269172e+07	BANK OF AMERICA CORP	06050510	11	1	6029	BAC	2000-01-01	13.514
3	75857	2000-01-31	-0.236531	2.152104e+07	SOLECTRON CORP	83418210	11	1	3672	SLR	2000-01-01	1.250
5	64311	2000-01-31	-0.170732	6.470676e+06	NORFOLK SOUTHERN CORP	65584410	11	1	4731	NSC	2000-01-01	13.357
7	42906	2000-01-31	-0.096859	4.862624e+06	HUNTINGTON BANCSHARES INC	44615010	11	3	6020	HBAN	2000-01-01	9.159

In [ ]:

```
df.shape
```

Out[ ]:

(101907, 16)

Logarithmic Transformation of Volume Data

Adding New Columns for Logarithmic Values

To enhance the analysis and potentially stabilize the variance across the data, we apply a logarithmic transformation to the volume metrics in our dataset. Specifically, we use the `np.log1p` function from the NumPy library, which computes the natural logarithm (`log(1 + x)`) for each data point. This function is particularly useful for financial data like trading volumes, which can vary exponentially across different stocks and over time. The `log1p` function is also helpful as it handles zero values smoothly by computing the logarithm of `(1 + x)`, ensuring no issues with zero or negative inputs.

Columns Transformed

- `Log_Average_Volume`: **This new column is created by applying the logarithmic transformation to the `Average_Volume` column. It represents the average daily trading volume in a logarithmic scale, which might help in reducing skewness of the distribution.**
- `Log_Total_Volume`: **Similarly, this column transforms the `Total_Volume`, which is the sum of all trading volumes in the month, to a logarithmic scale. This transformation might help in highlighting relative changes and trends more clearly than the absolute volume.**

These transformations are critical for subsequent analytical processes, especially in statistical modeling or machine learning, where the normalization of distribution can significantly impact the performance of algorithms.

In [ ]:

```
df['Log_Average_Volume'] = np.log1p(df['Average_Volume'])
df['Log_Total_Volume'] = np.log1p(df['Total_Volume'])
```

```
<ipython-input-25-f57cdd79c2d9>:1: 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/stable/user_g
uide/indexing.html#returning-a-view-versus-a-copy
  df['Log_Average_Volume'] = np.log1p(df['Average_Volume'])
<ipython-input-25-f57cdd79c2d9>: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/stable/user_g
uide/indexing.html#returning-a-view-versus-a-copy
  df['Log_Total_Volume'] = np.log1p(df['Total_Volume'])
```

In [ ]:

```
df.head()
```

Out[ ]:

	permno	End_date	ret	mcap	comnam	ncusip	shrcd	exchcd	hsiccd	Ticker	Start_date	Monthly_l
0	11308	2000-01-31	-0.013949	1.418695e+08	COCA COLA CO	19121610	11	1	2086	KO	2000-01-01	17.122
1	59408	2000-01-31	-0.034869	8.269172e+07	BANK OF AMERICA CORP	06050510	11	1	6029	BAC	2000-01-01	13.514
3	75857	2000-01-31	-0.236531	2.152104e+07	SOLECTRON CORP	83418210	11	1	3672	SLR	2000-01-01	1.250
5	64311	2000-01-	-	6.470676e+06	NORFOLK SOUTHERN	65584410	11	1	4731	NSC	2000-01-	13.357

	permno	End_date	31	0.170732 ret	mcap	comnam	ncusip	shrcd	exchcd	hsiccd	Ticker	Start_date	01	Monthly_l
7	42906	2000-01-31		-0.096859	4.862624e+06	HUNTINGTON BANCSHARES INC	44615010	11	3	6020	HBAN	2000-01-01		9.159

In [ ]:

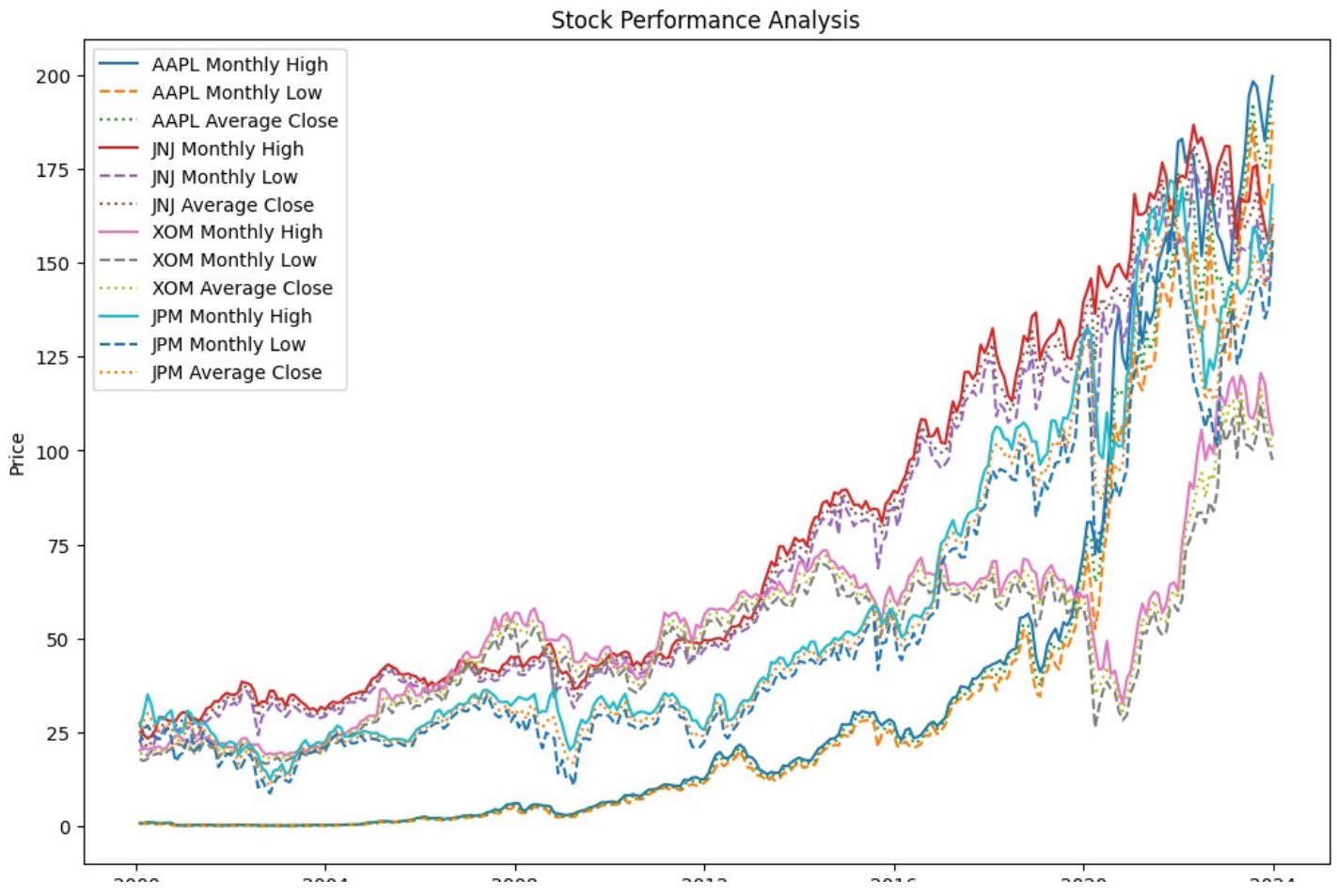
```
selected_tickers = ['AAPL', 'JNJ', 'XOM', 'JPM']
df_selected = df[df['Ticker'].isin(selected_tickers)]
```

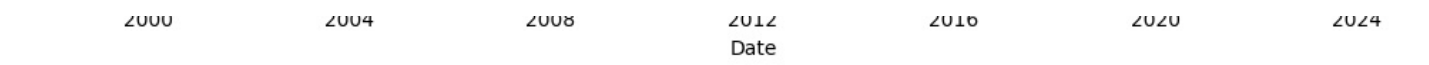
Stock Performance Trends Over Time

- **Price Trends:** The first chart tracks the monthly high, low, and average closing prices for stocks like AAPL, JNJ, XOM, and JPM over a span of two decades. All stocks show significant growth trends with periodic fluctuations reflecting market volatilities and economic cycles. Notably, AAPL shows a sharp upward trend, especially in the later years, indicating robust growth and investor confidence.
- **Comparison of Volatility:** The variability between the monthly highs and lows offers insights into the volatility of each stock. AAPL and JPM exhibit higher volatility as seen by the wider gaps between their monthly high and low prices, suggesting more aggressive price movements within each month.

In [ ]:

```
fig, ax = plt.subplots(figsize=(12, 8))
for ticker in selected_tickers:
    subset = df_selected[df_selected['Ticker'] == ticker]
    ax.plot(subset['End_date'], subset['Monthly_High'], label=f'{ticker} Monthly High')
    ax.plot(subset['End_date'], subset['Monthly_Low'], label=f'{ticker} Monthly Low', li
nestyle='--')
    ax.plot(subset['End_date'], subset['Average_Close'], label=f'{ticker} Average Close'
, linestyle=':')
ax.set_title('Stock Performance Analysis')
ax.set_xlabel('Date')
ax.set_ylabel('Price')
ax.legend()
plt.show()
```



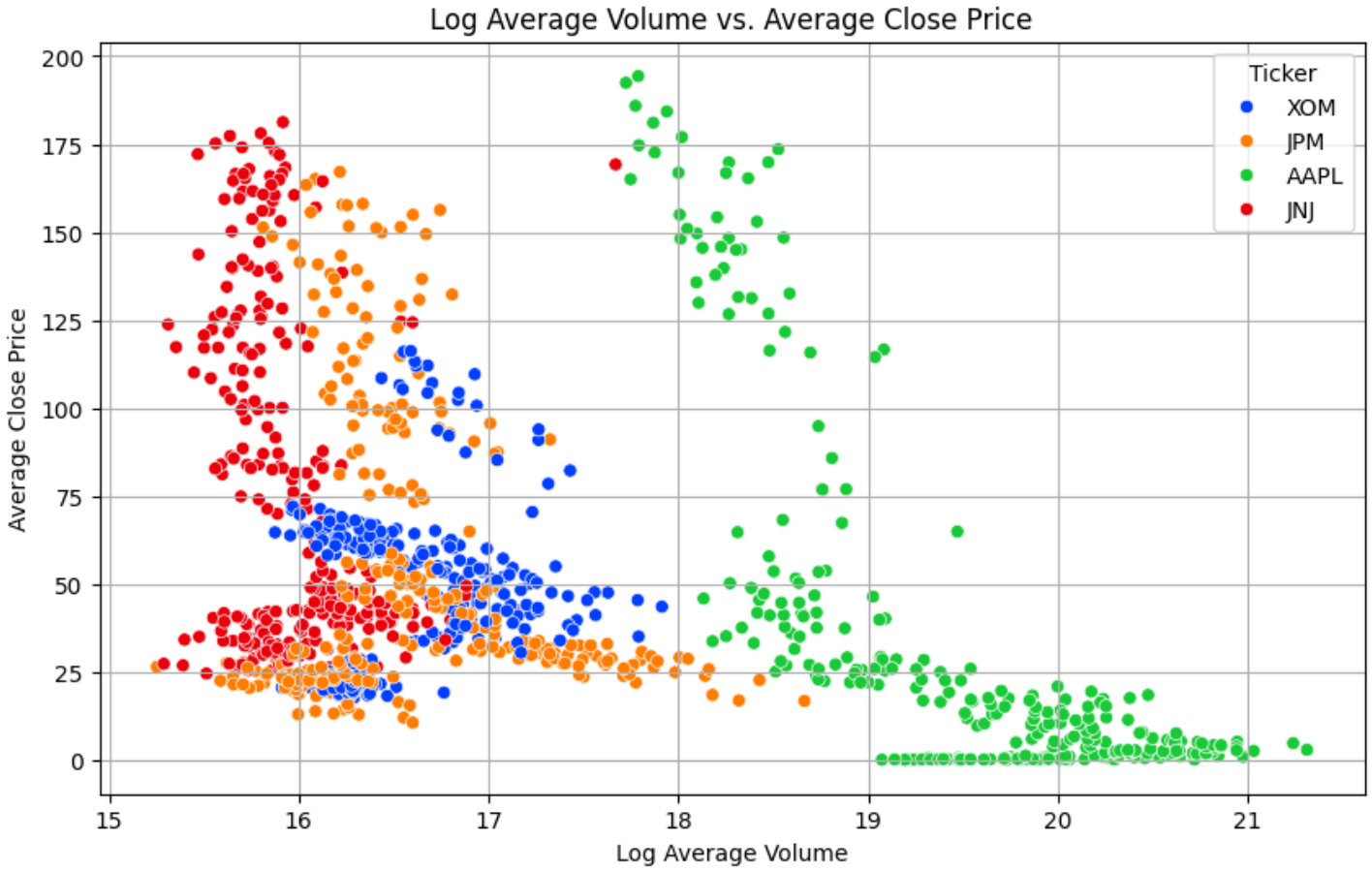


Volume and Market Analysis

- **Volume Trends:** In the line chart for log total volume, AAPL shows a declining trend in recent years despite its increasing stock price, suggesting a possible decrease in trading activity or consolidation of shares. In contrast, companies like JNJ and XOM show more stable volume trends.
- **Volume vs. Price Correlation:** The scatter plot reveals how log average volume correlates with average closing price. For AAPL, higher volumes are not necessarily associated with higher prices, indicating that volume spikes might correspond to both buying and selling pressures.

In [ ]:

```
plt.figure(figsize=(10, 6))
sns.scatterplot(x='Log_Average_Volume', y='Average_Close', hue='Ticker', data=df_selected
, palette='bright')
plt.title('Log Average Volume vs. Average Close Price')
plt.xlabel('Log Average Volume')
plt.ylabel('Average Close Price')
plt.legend(title='Ticker')
plt.grid(True)
plt.show()
```



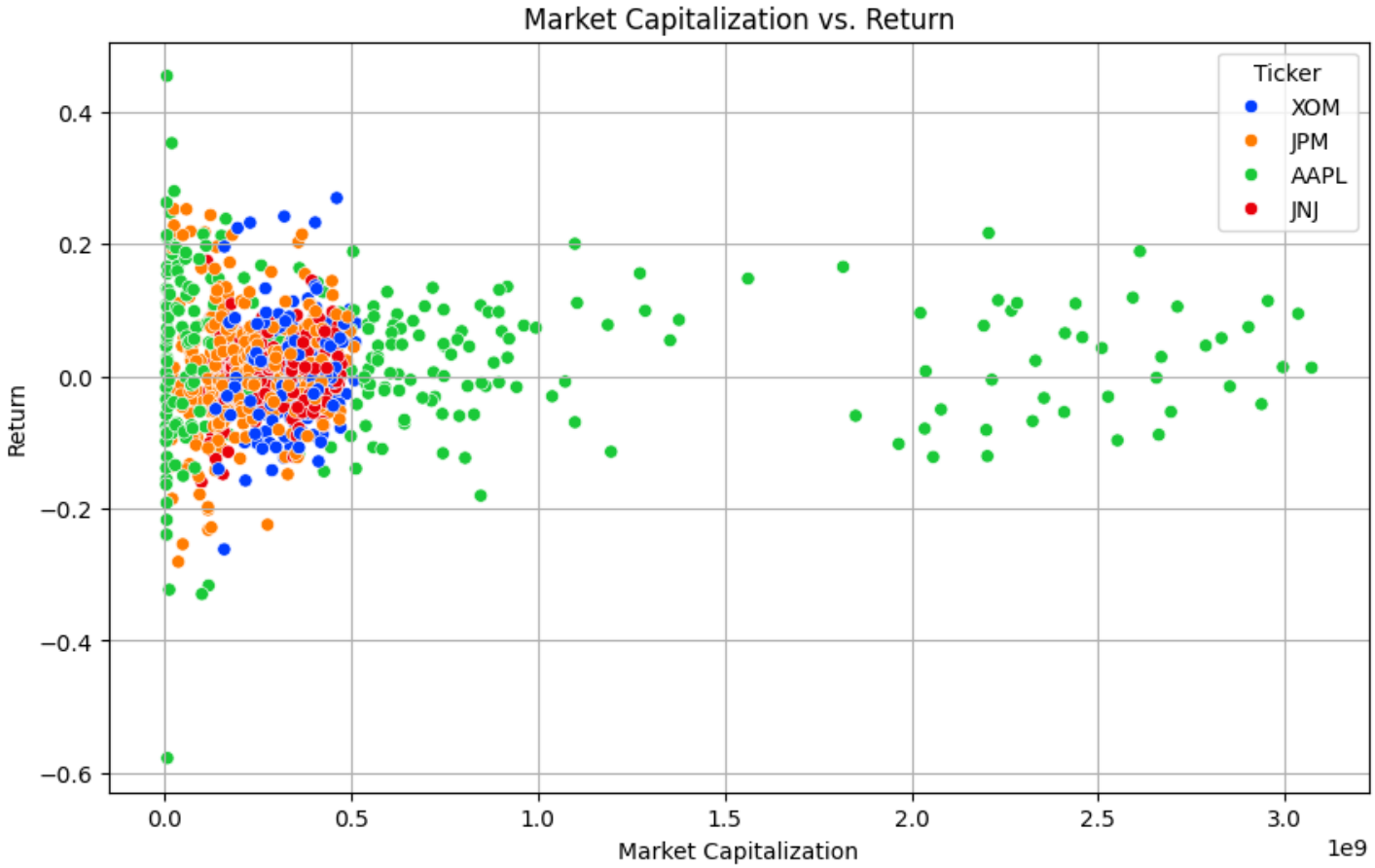
Market Capitalization and Returns

- **Capitalization vs. Return:** The scatter plot shows a weak correlation between market capitalization and returns, especially for larger caps like AAPL and JPM, suggesting that bigger companies do not always promise higher returns, possibly due to their already high valuations.
- **Performance Consistency:** AAPL shows higher returns across a range of market capitalizations, emphasizing its growth and profitability compared to peers like XOM and JNJ which show more scattered distributions of returns across different capitalizations.

In [ ]:

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

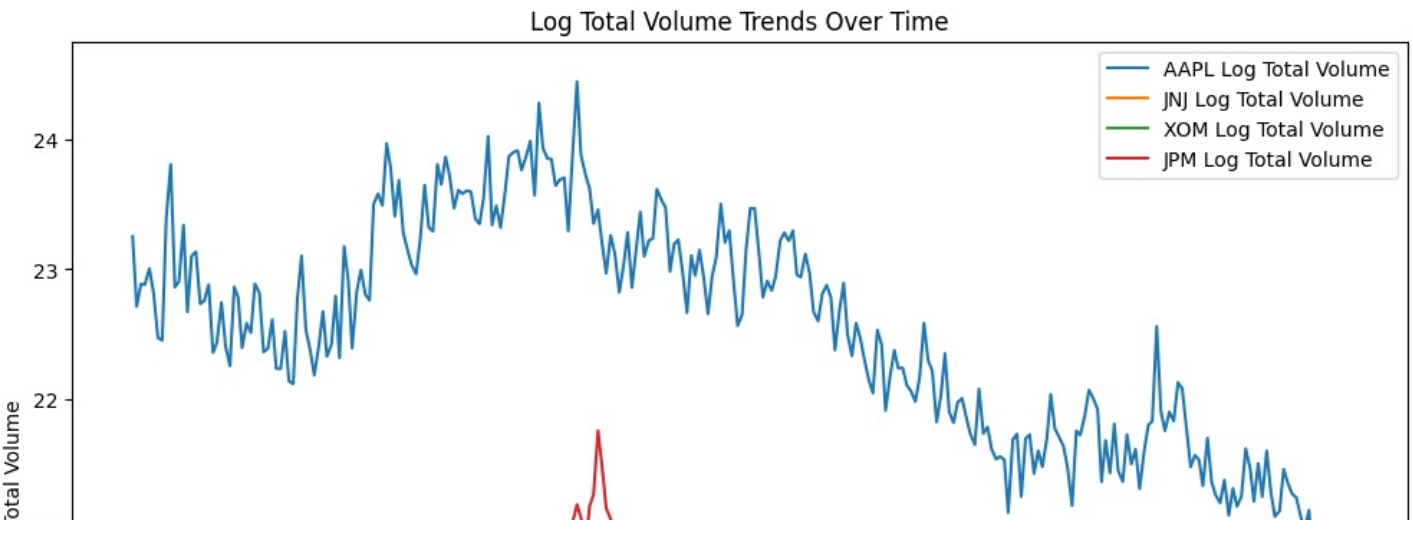
```
plt.figure(figsize=(10, 6))
sns.scatterplot(x='mcap', y='ret', hue='Ticker', data=df_selected, palette='bright')
plt.title('Market Capitalization vs. Return')
plt.xlabel('Market Capitalization')
plt.ylabel('Return')
plt.legend(title='Ticker')
plt.grid(True)
plt.show()
```

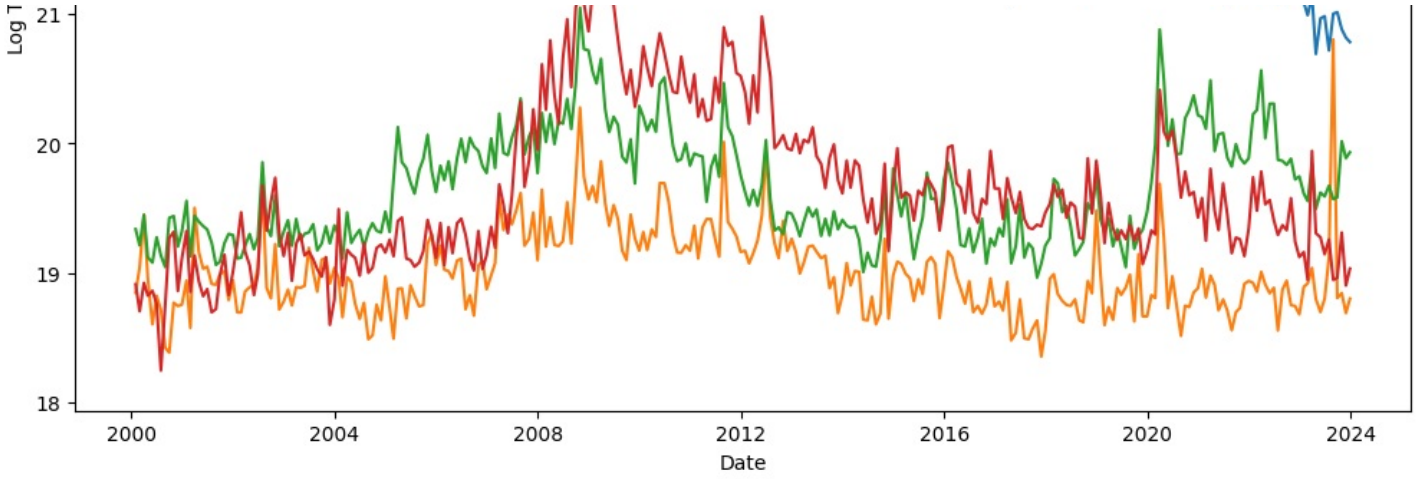


Visualization of Logarithmic Total Volume Trends Over Time

In [ ]:

```
fig, ax = plt.subplots(figsize=(12, 8))
for ticker in selected_tickers:
    subset = df_selected[df_selected['Ticker'] == ticker]
    ax.plot(subset['End_date'], subset['Log_Total_Volume'], label=f'{ticker} Log Total V
olume')
ax.set_title('Log Total Volume Trends Over Time')
ax.set_xlabel('Date')
ax.set_ylabel('Log Total Volume')
ax.legend()
plt.show()
```





Calculate Financial Indicators

`calculate_indicators`

This function is designed to calculate several financial indicators for each group of data, typically grouped by stock ticker or another classification method. These indicators provide insights into price volatility, trading volume, and market capitalization changes over time.

Calculations Performed

- **Price Volatility: Defined as the range between the monthly high and low prices divided by the average closing price for the month. This indicator measures the relative variation in price within a given month, offering insights into the stability or instability of the stock price.**
- **Volume Indicator: Calculated as the ratio of the natural logarithm of the total trading volume to the natural logarithm of the average trading volume. This indicator can help in understanding the consistency of trading activity—higher values may indicate more volatile trading activity within the month.**
- **Logarithmic Market Capitalization (Log_MCap): The natural logarithm of the market capitalization (`mcap`), which helps in normalizing the data and reducing skewness. Logarithmic transformations are often used in financial data to handle wide-ranging values.**
- **Market Capitalization Change (MCap_Change): This is calculated as the difference between the current and previous period's logarithmic market capitalizations. It measures how significantly a company's market size has changed over time, providing a sense of growth or contraction.**

In [ ]:

```
def calculate_indicators(group):
    group['Price_Volatility'] = (group['Monthly_High'] - group['Monthly_Low']) / group['
Average_Close']

    group['Volume_Indicator'] = np.log(group['Total_Volume']) / np.log(group['Average_Vo
lume'])

    group['Log_MCap'] = np.log(group['mcap'])

    group['MCap_Change'] = group['Log_MCap'].diff()

    return group
```

Applying Financial Indicators and Data Preparation

In [ ]:

```
df = df.sort_values(by=['Ticker', 'End_date'])
grouped = df.groupby('Ticker').apply(calculate_indicators)
grouped.reset_index(drop=True, inplace=True)
grouped.dropna(inplace=True)
grouped.head()
```

```
/usr/local/lib/python3.10/dist-packages/pandas/core/arraylike.py:396: RuntimeWarning: div
ide by zero encountered in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
/usr/local/lib/python3.10/dist-packages/pandas/core/arraylike.py:396: RuntimeWarning: div
ide by zero encountered in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

Out[ ]:

	permno	End_date	ret	mcap	comnam	ncusip	shrcd	exchcd	hsiccd	Ticker	...	Monthly_Low	A
1	87432	2000-07-31	-0.447458	1.843008e+07	AGILENT TECHNOLOGIES INC	00846U10	11	1	3826	A	...	24.298402	
2	87432	2000-08-31	0.480061	2.727766e+07	AGILENT TECHNOLOGIES INC	00846U10	11	1	3826	A	...	23.491020	
3	87432	2000-09-29	-0.188601	2.216942e+07	AGILENT TECHNOLOGIES INC	00846U10	11	1	3826	A	...	27.297261	
4	87432	2000-10-31	-0.053640	2.098026e+07	AGILENT TECHNOLOGIES INC	00846U10	11	1	3826	A	...	24.144623	
5	87432	2000-11-30	0.126856	2.364172e+07	AGILENT TECHNOLOGIES INC	00846U10	11	1	3826	A	...	23.414123	

5 rows × 22 columns

In [ ]:

```
grouped.shape
```

Out[ ]:

```
(101266, 22)
```

In [ ]:

```
df = grouped.copy()

columns_to_drop = ['permno', 'comnam', 'ncusip', 'shrcd', 'exchcd', 'hsiccd', 'Start_date
']

df = df.drop(columns=columns_to_drop)

df.to_csv('/content/gdrive/My Drive/r3_data.csv')

df.head()
```

Out[ ]:

	End_date	ret	mcap	Ticker	Monthly_High	Monthly_Low	Average_Volume	Total_Volume	Average_Close	Lo
1	2000-07-31	-0.447458	1.843008e+07	A	49.788660	24.298402	8.047685e+06	160953696.0	37.660608	
2	2000-08-31	0.480061	2.727766e+07	A	39.369580	23.491020	5.554862e+06	127761822.0	30.364647	
3	2000-09-29	-0.188601	2.216942e+07	A	38.562177	27.297261	3.088049e+06	61760983.0	33.189721	
4	2000-10-31	-0.053640	2.098026e+07	A	36.063141	24.144623	4.286745e+06	94308380.0	29.312225	
5	2000-11-30	0.126856	2.364172e+07	A	33.218071	23.414123	3.981011e+06	83601240.0	28.566007	

Integration of Risk-Free Rate Data for Financial Analysis

Loading and Preprocessing Risk-Free Rates

The risk-free rate data is loaded from a CSV file into the DataFrame `risk_free_rates`. Initial steps involve:

- **Inspecting Columns:** Checking the DataFrame's columns to understand the data structure.
- **Removing NA Values:** Dropping rows with any missing values to ensure data integrity.
- **Validating Date Format:** Filtering rows to include only those where the `Date` column matches the expected six-digit year-month format (e.g., 'YYYYMM'). This is crucial for consistent date handling.

Merging DataFrames on Date Key

- **Data Type Alignment:** Before merging, ensure that the `YYYYMM` and `RF` (risk-free rate) columns in `risk_free_rates` are converted to string types to match the formats in the main DataFrame.
- **Merging:** The two DataFrames are merged on the `YYYYMM` column using a left join to attach the corresponding risk-free rate to each row in the main dataset based on the year and month.

In [ ]:

```
risk_free_rates = pd.read_csv('F-F_Research_Data_Factors 2.CSV')
risk_free_rates.columns
risk_free_rates.dropna(inplace=True)
risk_free_rates = risk_free_rates[risk_free_rates['Date'].str.match(r'^\d{6}$')]

risk_free_rates['Date'] = pd.to_datetime(risk_free_rates['Date'], format='%Y%m')

risk_free_rates['YYYYMM'] = risk_free_rates['Date'].dt.strftime('%Y%m')
```

In [ ]:

```
df = pd.read_csv('/content/gdrive/My Drive/r3_data.csv')
df['End_date'] = pd.to_datetime(df['End_date'])
df['YYYYMM'] = df['End_date'].dt.strftime('%Y%m')

risk_free_rates['RF'] = risk_free_rates['RF'].astype(str)
risk_free_rates['YYYYMM'] = risk_free_rates['YYYYMM'].astype(str)

df = pd.merge(df, risk_free_rates[['RF', 'YYYYMM']], on='YYYYMM', how='left')

df['RF'] = pd.to_numeric(df['RF'], errors='coerce')
```

Calculation of Excess Returns

- **Excess Returns:** We calculate the excess return for each entry in the dataset by subtracting the risk-free rate (`RF`) from the stock return (`ret`). This metric, stored in the new column `Excess_Return`, represents the return on the stock over and above the return on a risk-free asset, adjusting for the risk taken by investing in the stock. Computing Excess Returns

In [ ]:

```
df['Excess_Return'] = df['ret'] - df['RF']
df.to_csv('r3_data_with_excess_returns.csv')


df = df.drop(columns=['YYYYMM','ret','Average_Volume','Total_Volume','mcap'])
df.sort_values(by=['Ticker', 'End_date'], inplace=True)
df['Future_Excess_Returns'] = df['Excess_Return'].shift(-1)
df.dropna(inplace=True)
```

In [ ]:

```
data_points_per_stock = df.groupby('Ticker').size()
```

```
data_points_per_stock.describe()
```

Out[ ]:

```
count    626.000000
mean     161.765176
std      101.483196
min        1.000000
25%       70.250000
50%      152.000000
75%      287.000000
max      287.000000
dtype: float64
```

Filtering Data by Minimum Number of Data Points

Setting the Minimum Threshold

To ensure robust statistical analysis and avoid biases due to insufficient data, we set a minimum threshold of data points, specified as `min_data_points = 70`. This threshold is chosen to balance between having enough data for meaningful analysis and retaining a sufficient number of stocks in the dataset.

Filtering the DataFrame

- **Grouping and Filtering: The DataFrame `df` is grouped by the `Ticker` column, which represents different stocks. A filter is then applied to each group to check if the number of rows (data points) exceeds the minimum threshold of 70. This step is crucial for focusing our analysis on stocks with adequate historical data, reducing the potential for skewed results caused by outlier effects or insufficient sample sizes.**
- **Application: The filtering is done using a lambda function within the `filter` method, which checks the length of each grouped subset. Only those groups with more than 70 data points are retained in the new DataFrame `filtered_df`.**

In [ ]:

```
min_data_points = 70

filtered_df = df.groupby('Ticker').filter(lambda x: len(x) > min_data_points)

df.head()
```

Out[ ]:

	Unnamed: 0	End_date	Ticker	Monthly_High	Monthly_Low	Average_Close	Log_Average_Volume	Log_Total_Volume	Price_V
0	1	2000-07-31	A	49.788660	24.298402	37.660608	15.900895	18.896627	0
1	2	2000-08-31	A	39.369580	23.491020	30.364647	15.530184	18.665678	0
2	3	2000-09-29	A	38.562177	27.297261	33.189721	14.943050	17.938782	0
3	4	2000-10-31	A	36.063141	24.144623	29.312225	15.271038	18.362081	0
4	5	2000-11-30	A	33.218071	23.414123	28.566007	15.197047	18.241569	0

Calculating Correlation

The correlation coefficients are computed using the `.corr()` method on the selected features from the DataFrame `df`. This method calculates the Pearson correlation coefficient for each pair of features, a standard measure of the linear dependence between two variables.

Visualizing Correlations with Heatmap

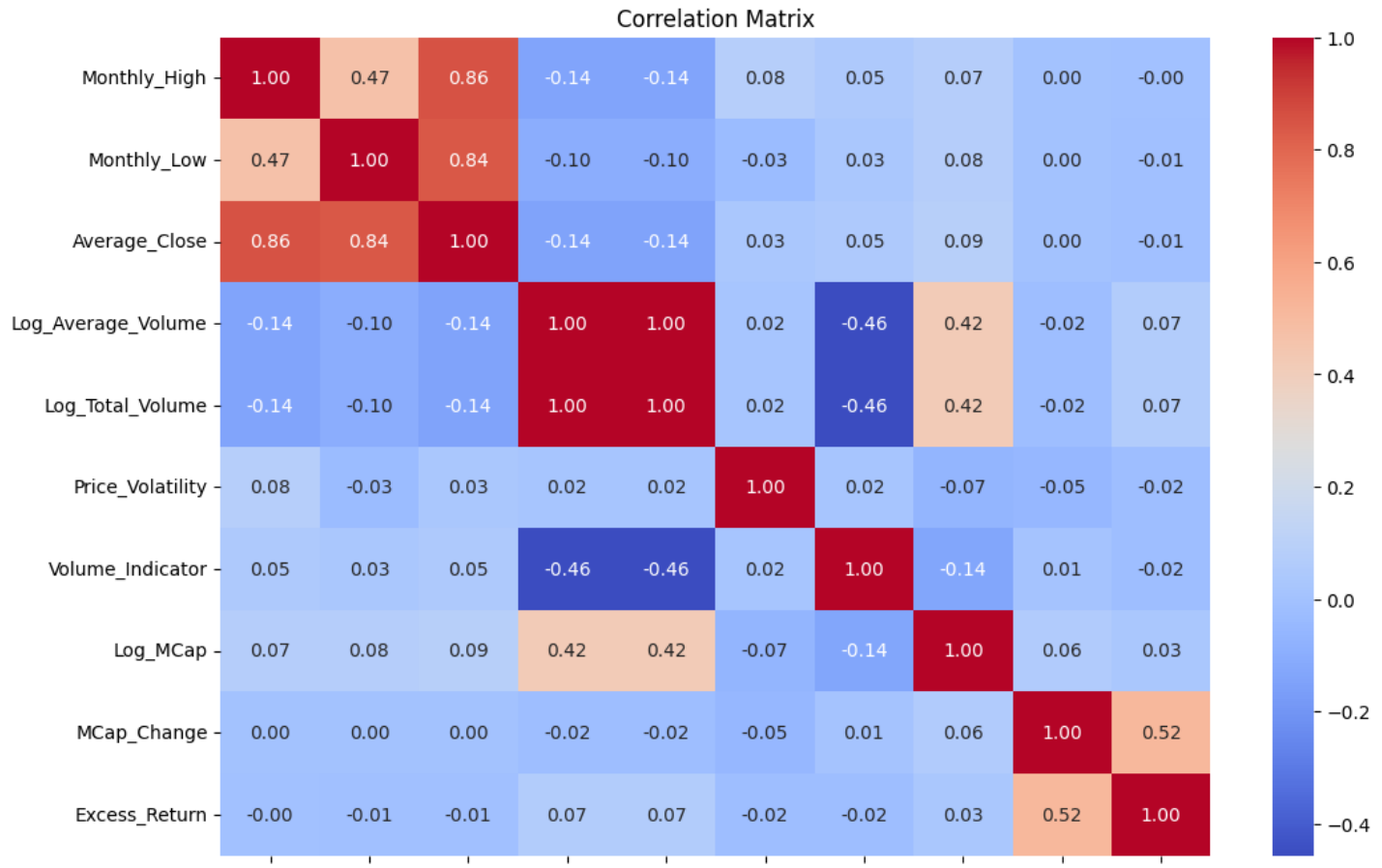
- **Monthly High, Low, and Average Close**: There is a strong positive correlation between `Monthly_High`, `Monthly_Low`, and `Average_Close` (ranging from 0.84 to 0.86). This suggests that when a stock reaches higher highs during a month, it also tends to have higher lows and closes at higher prices, which indicates overall bullish behavior during that period.
- **Log_Average_Volume and Log_Total_Volume**: Both logarithmic volume measures are perfectly correlated (correlation of 1.00), which is expected as they are derived from the same underlying trading volume data but represent different aspects (total vs. average). This perfect correlation confirms the consistency in volume measurement methods.
- **Volume and Price Metrics**: The correlations between both logarithmic volume metrics and the price metrics (`Monthly_High`, `Monthly_Low`, `Average_Close`) are consistently negative but relatively weak (about -0.14). This suggests a slight inverse relationship where higher volumes might not necessarily coincide with higher price levels, possibly indicating higher volumes during price declines.
- **Log_MCap and MCap_Change**: The correlation between `Log_MCap` and the changes in market cap (`MCap_Change`) is very low (0.06), suggesting that large changes in market cap do not happen frequently. This minimal correlation might indicate that market capitalization is relatively stable over the short term for the stocks in the dataset.
- **Excess Return and Market Capitalization**: There is a moderate positive correlation between `Excess_Return` and `MCap_Change` (0.52). This could imply that stocks experiencing significant increases in market cap also tend to yield higher excess returns, potentially reflecting investor confidence and positive market reactions.

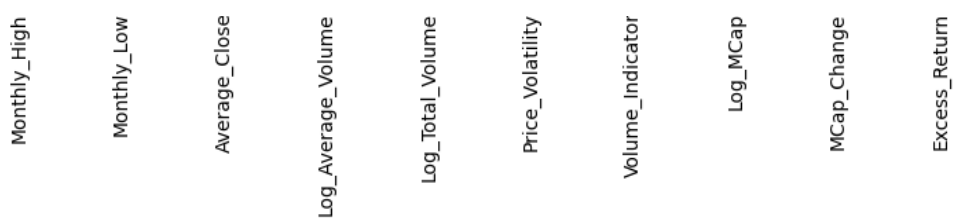
In [ ]:

```
features = ['Monthly_High', 'Monthly_Low','Average_Close', 'Log_Average_Volume', 'Log_Total_Volume', 'Price_Volatility',
            'Volume_Indicator', 'Log_MCap', 'MCap_Change','Excess_Return']

corr_matrix = df[features].corr()

plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```





Average_Close is very well correlated with Monthly_High and Monthly_Low - Applying PCA to combine Monthly_High and Monthly_low data together

In [ ]:

```
features_for_pca = ['Monthly_High', 'Monthly_Low']
pca = PCA(n_components=1)
df['High_Low_PCA'] = pca.fit_transform(df[features_for_pca])
df.drop(columns=features_for_pca + ['Log_Average_Volume'], inplace=True)
df.head()
```

Out[ ]:

	Unnamed: 0	End_date	Ticker	Average_Close	Log_Total_Volume	Price_Volatility	Volume_Indicator	Log_MCap	MCap_Chan
0	1	2000-07-31	A	37.660608	18.896627	0.676841	1.188400	16.729495	-0.4227
1	2	2000-08-31	A	30.364647	18.665678	0.522929	1.201897	17.121578	0.3920
2	3	2000-09-29	A	33.189721	17.938782	0.339410	1.200477	16.914224	-0.2073
3	4	2000-10-31	A	29.312225	18.362081	0.406606	1.202412	16.859092	-0.0551
4	5	2000-11-30	A	28.566007	18.241569	0.343203	1.200336	16.978524	0.1194

In [ ]:

```
features_to_scale = ['Average_Close', 'Log_Total_Volume', 'Price_Volatility',
                     'Volume_Indicator', 'Log_MCap', 'MCap_Change', 'High_Low_PCA']

scaler = StandardScaler()

df[features_to_scale] = scaler.fit_transform(df[features_to_scale])
```

Overview

This script implements a rolling window regression model to predict future excess returns based on selected financial indicators. It uses historical data for each stock ticker to build a predictive model and assesses its accuracy using the coefficient of determination (R^2).

Methodology

1. **Data Preparation:** For each unique stock ticker, the data is filtered and sorted by the end date.
2. **Model Training and Prediction:**
   - A linear regression model is trained on a set of features within a moving window of 40 observations (defined by `window_size`).
   - The model then predicts the excess return for the next time point using current feature values.
3. **Performance Evaluation:**
   - The model's predictions are compared to actual future returns, and the R^2 value is calculated to measure the model's explanatory power.
   - A higher R^2 value indicates a better fit of the model to the data, suggesting that the model can reliably forecast future returns based on historical data.

forecast future returns based on historical data.

Features Used

- `Average_Close`, `Log_Total_Volume`, `Price_Volatility`, `Volume_Indicator`, `Log_MCap`, `MCap_Change`, **and a principal component analysis (PCA) derived feature (** `High_Low_PCA` **).**

Result

In [ ]:

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

feature_names = ['Average_Close', 'Log_Total_Volume', 'Price_Volatility',
                 'Volume_Indicator', 'Log_MCap', 'MCap_Change', 'High_Low_PCA']

model = LinearRegression()

unique_tickers = df['Ticker'].unique()

r_squared_values = []

prediction_details = []

window_size = 40

for ticker in unique_tickers:

    df_stock = df[df['Ticker'] == ticker].sort_values(by='End_date')

    if len(df_stock) > window_size:
        predicted_returns = []
        actual_returns = []

        for i in range(window_size, len(df_stock)):

            X_train = df_stock.iloc[i-window_size:i][feature_names].values
            y_train = df_stock.iloc[i-window_size:i]['Future_Excess_Returns'].values
            model.fit(X_train, y_train)

            X_next = df_stock.iloc[i][feature_names].values.reshape(1, -1)
            y_next = df_stock.iloc[i]['Future_Excess_Returns']

            predicted_return = model.predict(X_next)[0]

            predicted_returns.append(predicted_return)
            actual_returns.append(y_next)

            prediction_details.append({
                'Ticker': ticker,
                'Date': df_stock.iloc[i]['End_date'],
                'Features': X_next,
                'Actual_Return': y_next,
                'Predicted_Return': predicted_return
            })

        r_squared = r2_score(actual_returns, predicted_returns)
        r_squared_values.append((ticker, r_squared))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_regression.py:918: UndefinedMetr
icWarning: R^2 score is not well-defined with less than two samples.
  warnings.warn(msg, UndefinedMetricWarning)
```

In [ ]:

```
r_squared_values_df = pd.DataFrame(r_squared_values, columns=['ticker', 'r_squared'])

r_squared_values_df.dropna()
```

```
total_r_squared_value = r_squared_values_df['r_squared'].sum()

for ticker, r_squared in r_squared_values_df.itertuples(index=False):
    print(f"R-squared value for {ticker}: {r_squared}")
```

```
R-squared value for A: 0.1923856159794467
R-squared value for AA: 0.2231902329984491
R-squared value for AAL: -0.07944890795175308
R-squared value for AAP: -0.2914982438711453
R-squared value for AAPL: 0.0588897472294630 5
R-squared value for ABBV: 0.1789005808236166
R-squared value for ABMD: -0.04324426441345541
R-squared value for ABT: 0.4576841713733799
R-squared value for ACN: 0.010050719299387612
R-squared value for ADBE: 0.30100608981114574
R-squared value for ADI: 0.3285009691504357
R-squared value for ADM: 0.3618659518241718
R-squared value for ADP: 0.47131948509056476
R-squared value for ADSK: 0.1724727553760066
R-squared value for AEE: 0.4491663609170681
R-squared value for AEP: 0.3210026575833812
R-squared value for AES: 0.23322224592278296
R-squared value for AET: 0.4635805090237022
R-squared value for AFL: 0.25457866057515866
R-squared value for AIG: -1.5326098397219523
R-squared value for AIV: -0.337492774819768
R-squared value for AIZ: 0.41356936981976855
R-squared value for AJG: 0.4090970397436584
R-squared value for AKAM: 0.2349266890702263
R-squared value for ALB: 0.07885341486342057
R-squared value for ALGN: -0.3256781417140058
R-squared value for ALK: -0.5018098128477506
R-squared value for ALL: 0.423134750882457
R-squared value for ALLE: -0.18495674565890363
R-squared value for AMAT: 0.2443814902688657
R-squared value for AMCR: -13.836686158661625
R-squared value for AMD: -0.0419897920812915
R-squared value for AME: 0.08078455611031143
R-squared value for AMG: -1.3624281870456008
R-squared value for AMGN: 0.4841901615618864 3
R-squared value for AMP: 0.05309710632533304
R-squared value for AMT: 0.395636045459986
R-squared value for AMZN: -0.11231925458027825
R-squared value for AN: 0.32845029329879327
R-squared value for ANET: 0.15127029468501785
R-squared value for ANF: -0.722864390528398
R-squared value for ANSS: -0.2342886136143163
R-squared value for AON: 0.5164512062703439
R-squared value for AOS: 0.10523128820618977
R-squared value for APA: 0.08279238767172636
R-squared value for APD: 0.28529242374914565
R-squared value for APH: 0.07737476533251042
R-squared value for APTV: -0.3271170114521136
R-squared value for ARE: 0.35496062798847516
R-squared value for ASH: 0.428042613609239
R-squared value for ATI: 0.09599953830672203
R-squared value for ATO: -1.7557908743961992
R-squared value for AVB: 0.0795568208097911
R-squared value for AVGO: -0.0076073205206943495
R-squared value for AVY: 0.3122482097637158
R-squared value for AWK: -0.04890984365824025
R-squared value for AXP: 0.2510045516805356
R-squared value for AZO: 0.5294606577193784
R-squared value for BA: 0.38934808859380554
R-squared value for BAC: 0.220994146241846
R-squared value for BAX: 0.46078572651726357
R-squared value for BBY: 0.2940178580152807
R-squared value for BC: 0.0767105702737454
R-squared value for BDX: 0.46927874744412335
R-squared value for BEN: 0.2398681058013663
R-squared value for BIG: -0.557170109542529
```

```
R-squared value for BIIB: 0.09549867316971727
R-squared value for BIO: -0.8214340304778285
R-squared value for BK: 0.40063048445164307
R-squared value for BKNG: -0.05956543897021205
R-squared value for BKR: -0.3589897681184593
R-squared value for BLK: -0.14094231970783122
R-squared value for BMS: 0.4463823286120434
R-squared value for BMY: 0.47020870299689477
R-squared value for BR: -0.9027369482422025
R-squared value for BSX: 0.33204188620960773
R-squared value for BWA: 0.017324193764555296
R-squared value for BXP: 0.10141323399834745
R-squared value for C: 0.15075948524552119
R-squared value for CAG: 0.37961952050534353
R-squared value for CAH: 0.5365918290539906
R-squared value for CARR: 0.18391227847928415
R-squared value for CAT: 0.2065775060745203
R-squared value for CB: 0.437060394669068
R-squared value for CBOE: 0.4484681083230184
R-squared value for CBRE: -0.6960126555715447
R-squared value for CCI: 0.3238654374874874
R-squared value for CCL: -0.0711753563658879
R-squared value for CCU: -0.08617751233529236
R-squared value for CDNS: 0.2644894667262403
R-squared value for CDW: -6.404650587455197
R-squared value for CE: -2.016061342150418
R-squared value for CF: -0.09333415138809431
R-squared value for CFG: 0.1380520423623649
R-squared value for CHD: -0.27072815742297096
R-squared value for CHRW: 0.3937692004245884
R-squared value for CHTR: 0.5285543233220255
R-squared value for CI: 0.37025551802822143
R-squared value for CIEN: 0.025011385653983087
R-squared value for CINF: 0.4412833993531138
R-squared value for CL: 0.4986317559753697
R-squared value for CLF: -0.1298533069845933
R-squared value for CLX: 0.4613509670567175
R-squared value for CMA: 0.288653829495523
R-squared value for CMCSA: 0.5101350759078565
R-squared value for CME: 0.07052937994502939
R-squared value for CMG: -0.11274686096493869
R-squared value for CMI: 0.05393141613827945
R-squared value for CMS: 0.39176675350999235
R-squared value for CNC: 0.042506208125675626
R-squared value for CNP: 0.3359880335826282
R-squared value for CNX: -1.105433034609792
R-squared value for COF: 0.2499548725015902
R-squared value for COL: -0.18968238684244954
R-squared value for COO: -0.2782619082749693
R-squared value for COP: 0.4161518332594403
R-squared value for COST: 0.378113007403558
R-squared value for COTY: -1.055189051817789
R-squared value for CPB: 0.5409904468096387
R-squared value for CPRT: -0.647405856419748
R-squared value for CPWR: -0.8430064856059731
R-squared value for CRM: 0.06828380462545636
R-squared value for CSCO: 0.2748004027511719
R-squared value for CSX: 0.47311339968405686
R-squared value for CTAS: 0.373787848335673
R-squared value for CTSH: 0.21548044183854131
R-squared value for CTVA: -6.516586505114824
R-squared value for CVG: 0.44345584810027194
R-squared value for CVS: 0.22671646916682575
R-squared value for CVX: 0.5652896399571758
R-squared value for D: 0.5243249301537637
R-squared value for DAL: -0.03155659189783333
R-squared value for DD: 0.24946950105002097
R-squared value for DDS: 0.15431788440493577
R-squared value for DE: 0.3552698530750864
R-squared value for DFS: 0.031234710526006237
R-squared value for DG: 0.09385993830475314
R-squared value for DGX: 0.38131321732571766
```

```
R-squared value for DHI: 0.06128490582800439
R-squared value for DHR: 0.08616803857393629
R-squared value for DIS: 0.33576855715602305
R-squared value for DLR: 0.042014097333821
R-squared value for DLTR: 0.1443460793525102
R-squared value for DLX: -0.5529327801229216
R-squared value for DOV: 0.38686040805624
R-squared value for DOW: -2.056146060924267
R-squared value for DPZ: -1.4581139274384225
R-squared value for DRI: 0.4007396445210093
R-squared value for DTE: 0.3629355875986181
R-squared value for DUK: 0.35234553496643184
R-squared value for DVA: 0.30824909632281094
R-squared value for DVN: -0.05024209460009188
R-squared value for DXC: -0.07736447134813362
R-squared value for DXCM: -0.9558500223262181
R-squared value for EA: 0.27475456628903294
R-squared value for EBAY: 0.3320999429697361
R-squared value for ECL: 0.5072213352059662
R-squared value for ED: 0.5702241592687067
R-squared value for EFX: 0.3822512982422477
R-squared value for EIX: 0.41332081250168773
R-squared value for EL: -0.11374175076146997
R-squared value for EMN: 0.4053869522633624
R-squared value for EMR: 0.3917499477182028
R-squared value for EOG: 0.06931792265085535
R-squared value for EP: 0.4193109490535035
R-squared value for EQIX: 0.30461781877840266
R-squared value for EQR: 0.5412756311378305
R-squared value for EQT: 0.4903770415021239
R-squared value for ES: -0.10422950991170121
R-squared value for ESRX: -0.02252584698514193
R-squared value for ESS: -0.04325880102625668
R-squared value for ETN: 0.3435370394017434
R-squared value for ETR: 0.32090191922700995
R-squared value for EVRG: -0.7341366975231778
R-squared value for EW: 0.1613193339079957
R-squared value for EXC: 0.4253153313727631
R-squared value for EXPD: 0.4077045960428589
R-squared value for EXPE: -0.1577248003619689
R-squared value for EXR: 0.06439972636476998
R-squared value for F: -0.04257424406694432
R-squared value for FANG: -0.8720088062263305
R-squared value for FAST: 0.2040299721000569
R-squared value for FCX: -0.09721741976079112
R-squared value for FDX: 0.37868258869135973
R-squared value for FE: 0.43327011906675383
R-squared value for FFIV: 0.13775394275816177
R-squared value for FHN: -0.3078159769072173
R-squared value for FIS: 0.3559476710365722
R-squared value for FITB: 0.1731804580075167
R-squared value for FLR: 0.3201097486871288
R-squared value for FLS: 0.061903122594329174
R-squared value for FLT: -0.11186948762973015
R-squared value for FMC: 0.048023213242944895
R-squared value for FOSL: 0.11079414837456392
R-squared value for FOX: -3.4595487754876117
R-squared value for FOXA: -2.047197907680981
R-squared value for FRT: 0.16629868194372144
R-squared value for FSLR: -0.07158114003940241
R-squared value for FTI: -0.015519622904429431
R-squared value for FTNT: -1.4058528945630746
R-squared value for FTV: 0.012186548743010728
R-squared value for GD: 0.4188080154817353
R-squared value for GE: 0.3950106012345518
R-squared value for GILD: 0.2691269649970238
R-squared value for GIS: 0.47618795628356136
R-squared value for GL: -6.398932579529932
R-squared value for GLW: 0.28900163301753923
R-squared value for GM: -0.07391273579464386
R-squared value for GME: -0.49779143536592696
R-squared value for GNW: -0.5097554744212207
```

```
R-squared value for GOOG: 0.27399117768333503
R-squared value for GOOGL: -0.16384454468918896
R-squared value for GPC: 0.4661867088242565
R-squared value for GPN: 0.226720729528201
R-squared value for GPS: 0.15686518124430615
R-squared value for GRMN: 0.06980675714709617
R-squared value for GS: 0.37845664107038757
R-squared value for GT: 0.20670672366928688
R-squared value for GWW: 0.557639444762619
R-squared value for HAL: 0.021206597644953695
R-squared value for HAS: 0.43701192886750106
R-squared value for HBAN: 0.27083515667291924
R-squared value for HBI: -0.007060450227738713
R-squared value for HCA: 0.17800477075522292
R-squared value for HD: 0.48417478190107544
R-squared value for HES: 0.17658311221298828
R-squared value for HIG: -0.5698367353602756
R-squared value for HII: 0.0012732605626936122
R-squared value for HLT: 0.04621763080241381
R-squared value for HOG: 0.022109239867867836
R-squared value for HOLX: 0.11385431964379555
R-squared value for HON: 0.2629054183176993
R-squared value for HP: 0.014424967851954462
R-squared value for HPE: 0.10420890989994958
R-squared value for HPQ: 0.370029660473399
R-squared value for HRB: 0.3761539629233229
R-squared value for HRL: 0.21138507614376234
R-squared value for HSIC: 0.03823580894158918
R-squared value for HST: 0.0728183667760528
R-squared value for HSY: 0.5174437659918045
R-squared value for HUM: 0.2616306934951935
R-squared value for HWM: 0.40376485872961376
R-squared value for IBM: 0.43980431818227383
R-squared value for ICE: 0.022132898754653296
R-squared value for IDXX: -0.43045138102459646
R-squared value for IEX: -11.767153533113147
R-squared value for IFF: 0.42607106555165497
R-squared value for IGT: 0.3756751715256902
R-squared value for ILMN: 0.024654659931301337
R-squared value for INCY: 0.002026332203902248
R-squared value for INTC: 0.4173497368134932
R-squared value for INTU: 0.40877326568639394
R-squared value for IP: 0.21461723401278854
R-squared value for IPG: 0.21197258702617428
R-squared value for IPGP: -1.7507598847167416
R-squared value for IQV: -0.7284430734413223
R-squared value for IR: 0.2458092204667729
R-squared value for IRM: 0.16509641688325627
R-squared value for ISRG: -0.009570184242678748
R-squared value for IT: 0.019460405745128084
R-squared value for ITT: 0.42256488763983024
R-squared value for ITW: 0.4545212968424488
R-squared value for IVZ: -0.021282827652721936
R-squared value for J: -5.932626266785204
R-squared value for JBHT: 0.1438511829405118
R-squared value for JBL: 0.2730404138813183
R-squared value for JCI: 0.18163368140594371
R-squared value for JKHY: -2.746115654555723
R-squared value for JNJ: 0.5027657201269367
R-squared value for JNPR: -0.0937522860505191
R-squared value for JPM: 0.4307497193281291
R-squared value for JWN: 0.2264129884949132
R-squared value for K: 0.3927834406692199
R-squared value for KBH: 0.26163108855798267
R-squared value for KEY: 0.4593042901339941
R-squared value for KEYS: -1.1562772731273308
R-squared value for KHC: -0.0024640634404708983
R-squared value for KIM: 0.1184521424743 26
R-squared value for KLAC: 0.3819948111925049
R-squared value for KMB: 0.5328846621856977
R-squared value for KMG: -0.5201782142461999
R-squared value for KMI: 0.034124353654609196
```

```
R-squared value for KMX: -0.024337637532270806
R-squared value for KO: 0.4617965019277256
R-squared value for KR: 0.4512186088063358
R-squared value for KSS: 0.27867149774981503
R-squared value for L: 0.3403803418081247
R-squared value for LDOS: -12.22367685660683
R-squared value for LEG: 0.34158838360961874
R-squared value for LEN: -0.11592426571329817
R-squared value for LH: 0.21359165035077632
R-squared value for LHX: -8.269487061847446
R-squared value for LIN: -1.3316338144417545
R-squared value for LKQ: 0.3220428456162411
R-squared value for LLY: 0.5943587287542103
R-squared value for LMT: 0.5688282805883502
R-squared value for LNC: 0.12116406990860851
R-squared value for LNT: -0.17823009331611694
R-squared value for LOW: 0.5726839489983029
R-squared value for LPX: 0.0290330527418724
R-squared value for LRCX: -0.0007513533821912333
R-squared value for LUV: 0.36175546599234365
R-squared value for LVS: -3.3790793023918706
R-squared value for LW: 0.3517205520041219
R-squared value for LYB: 0.08913537773327107
R-squared value for LYV: -6.792612893365124
R-squared value for M: -0.08842352412274046
R-squared value for MA: 0.36712835559476276
R-squared value for MAA: -0.15418108006313358
R-squared value for MAC: 0.31402577448151414
R-squared value for MAR: 0.35795413442477086
R-squared value for MAS: 0.3886722065012125
R-squared value for MAT: 0.43848145728898136
R-squared value for MBI: -0.3164414733587211
R-squared value for MCD: 0.49890659934067827
R-squared value for MCHP: 0.1994090862852279
R-squared value for MCK: 0.5375335586450785
R-squared value for MCO: 0.43861060099859717
R-squared value for MDLZ: 0.32192025089104614
R-squared value for MDT: 0.5067190662966417
R-squared value for MET: 0.1837669038638311
R-squared value for MGM: 0.15217535474646848
R-squared value for MHK: 0.08485680729175216
R-squared value for MKC: 0.2808942721569132
R-squared value for MKTX: -1.730960524733916
R-squared value for MLM: -0.05966869980262768
R-squared value for MMC: 0.39974390003157045
R-squared value for MMM: 0.4628287727964553
R-squared value for MNST: 0.019650939133305734
R-squared value for MO: 0.36845858926340613
R-squared value for MOS: -0.09868416782271039
R-squared value for MPC: -0.037898343023378356
R-squared value for MRK: 0.45871708782292053
R-squared value for MRO: -0.35567847299417243
R-squared value for MS: 0.22762561093314282
R-squared value for MSCI: -0.864639449166819
R-squared value for MSFT: 0.4590048076619174
R-squared value for MSI: 0.2358885587392643
R-squared value for MTB: 0.41443070971155305
R-squared value for MTD: 0.16713478894129097
R-squared value for MTG: 0.19303061346023143
R-squared value for MU: 0.1576983073176944
R-squared value for MUR: -0.9991842208802928
R-squared value for NAVI: -0.287472152231268
R-squared value for NBR: 0.3087781735889177
R-squared value for NCLH: -0.3027435311332236
R-squared value for NDAQ: -0.021194939576662186
R-squared value for NEE: -0.06148819941642136
R-squared value for NEM: 0.19860915469158724
R-squared value for NFLX: -0.10158904950777004
R-squared value for NFX: -0.1784359977012273
R-squared value for NI: 0.5430305726195075
R-squared value for NKE: 0.2936714629012036
R-squared value for NOC: 0.4870591637709277
```

R-squared value for NOV: -0.7338337125660834
R-squared value for NOW: -2.6206790949650016
R-squared value for NRG: 0.10897164166313755
R-squared value for NSC: 0.49493526668931276
R-squared value for NTAP: 0.15025608153253522
R-squared value for NTRS: 0.34410680370664404
R-squared value for NUE: 0.23535810395860457
R-squared value for NVDA: 0.12695673527209284
R-squared value for NVR: -0.19372035365692142
R-squared value for NWL: 0.18527327171420493
R-squared value for NWS: -0.23680247438158664
R-squared value for NWSA: -0.3720039404256754
R-squared value for NYT: 0.41083380568592787
R-squared value for O: -0.09921042599206409
R-squared value for ODFL: 0.09960778396925252
R-squared value for ODP: 0.16080734511679295
R-squared value for OI: -0.3531719365595114
R-squared value for OKE: 0.15838163296494578
R-squared value for OMC: 0.38339956764435323
R-squared value for ORCL: 0.3540200292798409
R-squared value for ORLY: 0.3220779170452853
R-squared value for OTIS: -1.0361416048359118
R-squared value for OXY: -0.03551302151997637
R-squared value for PAYC: -6.4493916257094135
R-squared value for PAYX: 0.3550197853593259
R-squared value for PBI: 0.5695209972383674
R-squared value for PCAR: 0.42615562818136
R-squared value for PCG: 0.48069605693140904
R-squared value for PDCO: -0.6555021107264314
R-squared value for PEAK: -3.0037045804569678
R-squared value for PEG: 0.4990653537549675
R-squared value for PEP: 0.49345167326494666
R-squared value for PFE: 0.4417119700411509
R-squared value for PFG: 0.04089713233542347
R-squared value for PG: 0.5878875799211367
R-squared value for PGR: 0.5579729314224842
R-squared value for PH: 0.36508908765160153
R-squared value for PHM: -0.009318640591979621
R-squared value for PKG: 0.0031429148572893473
R-squared value for PLD: -1.1160055997004408
R-squared value for PM: 0.20919805029803018
R-squared value for PNC: 0.3984969910826587
R-squared value for PNR: -0.060525198567030536
R-squared value for PNW: 0.5077620307114415
R-squared value for PPG: 0.2961718446122934
R-squared value for PPL: 0.4547216506953544
R-squared value for PRGO: -0.6094959585587323
R-squared value for PRU: 0.24660642784866083
R-squared value for PSA: 0.06536320957297481
R-squared value for PSX: 0.20947217968405107
R-squared value for PVH: -0.40393991396249374
R-squared value for PWR: 0.46425451179943544
R-squared value for PXD: 0.0315156026051826
R-squared value for PYPL: 0.14712804982554994
R-squared value for QCOM: 0.019016946371277266
R-squared value for QRVO: 0.09871185189199094
R-squared value for R: 0.42985729761858604
R-squared value for RCL: -0.34708980721982163
R-squared value for REG: 0.08433747643390499
R-squared value for REGN: 0.24291287549543628
R-squared value for RF: 0.25616684484739927
R-squared value for RHI: 0.34243992942392076
R-squared value for RIG: 0.10341191424197527
R-squared value for RJF: -0.04935911343791499
R-squared value for RL: 0.001145809390326824
R-squared value for RMD: -0.47604998786039787
R-squared value for ROK: 0.26172836445115677
R-squared value for ROL: -0.6634900656218841
R-squared value for ROP: 0.173678932624579
R-squared value for ROST: 0.12073185341513315
R-squared value for RRC: -0.37557657774892683
R-squared value for RSG: 0.1045675955065768

```
R-squared value for RTX: -26.645156072804205
R-squared value for SANM: -0.31351086684011653
R-squared value for SBAC: 0.2437386789043534
R-squared value for SBUX: 0.3185170327477663
R-squared value for SCG: nan
R-squared value for SCHW: 0.029525596150496325
R-squared value for SEE: 0.20023178780163442
R-squared value for SHW: 0.44133095483748763
R-squared value for SJM: 0.29294377769241564
R-squared value for SLB: 0.22514081680940545
R-squared value for SLG: -0.12471867483955479
R-squared value for SLM: 0.23820815921078764
R-squared value for SNA: 0.4242920111880367
R-squared value for SNI: -0.09098482333940838
R-squared value for SNPS: 0.4748409592090348
R-squared value for SNV: 0.47703663107962047
R-squared value for SO: 0.49289562422774413
R-squared value for SPG: 0.18515226861838863
R-squared value for SPGI: 0.3403447039377784
R-squared value for SRCL: 0.15313403936159176
R-squared value for SRE: 0.48566993535645675
R-squared value for STE: -4.038215824404936
R-squared value for STT: 0.1229264316758899
R-squared value for STX: -0.018332181484196353
R-squared value for STZ: 0.18878345757328807
R-squared value for SVU: 0.28212361726146296
R-squared value for SWK: 0.25845851207298487
R-squared value for SWKS: 0.12729423139193285
R-squared value for SWN: -0.7055250036473351
R-squared value for SYF: 0.09647925443340921
R-squared value for SYK: 0.3568196742496549
R-squared value for SYY: 0.2573213376894765
R-squared value for T: 0.5410643005533932
R-squared value for TAP: 0.12905201891742502
R-squared value for TDC: -0.45585905443433394
R-squared value for TDG: -0.008670512649286977
R-squared value for TDY: 0.10981163803649185
R-squared value for TEL: -0.022708595670103637
R-squared value for TER: 0.0919142007228072
R-squared value for TFC: -0.18650596446822276
R-squared value for TFX: -0.40581904463762575
R-squared value for TGT: 0.3661134968421512
R-squared value for THC: -0.15500607300725844
R-squared value for TJX: 0.5173044386789996
R-squared value for TMO: 0.1378652917396691
R-squared value for TMUS: -3.361426733418976
R-squared value for TPR: -0.016562966851519034
R-squared value for TRIP: -0.36053797883140226
R-squared value for TROW: 0.372073956143551
R-squared value for TRV: 0.29554755065559846
R-squared value for TSCO: 0.34571511446153014
R-squared value for TSN: 0.1424536725431529
R-squared value for TT: -4.365363453429569
R-squared value for TTWO: 0.5187641865116405
R-squared value for TUP: -0.1801656302558854
R-squared value for TWX: 0.6648076974836437
R-squared value for TXN: 0.3329890286602928
R-squared value for TXT: 0.05759788789991127
R-squared value for TYL: -3.0284704031219087
R-squared value for UA: -0.19333254374304798
R-squared value for UAA: -0.8539307929771283
R-squared value for UAL: 0.15187949987496951
R-squared value for UDR: 0.48013437785233304
R-squared value for UHS: 0.2467651451392524
R-squared value for UIS: -0.2994225380758695
R-squared value for ULTA: 0.3722795203070758
R-squared value for UNH: 0.46811502888498824
R-squared value for UNM: 0.3186797827503398
R-squared value for UNP: 0.4903263927940342
R-squared value for UPS: 0.252545337679129
R-squared value for URBN: -0.020292303978917126
R-squared value for URI: 0.05549960194956294
```

```
R-squared value for USB: 0.37017323887390186
R-squared value for V: 0.4688128738934355
R-squared value for VFC: 0.45043488097901896
R-squared value for VLO: 0.26779293095545487
R-squared value for VMC: 0.38723924878865645
R-squared value for VNO: -0.24142603334865353
R-squared value for VRSK: 0.34203808028243354
R-squared value for VRSN: 0.22790225341272063
R-squared value for VRTX: 0.2837695829292948
R-squared value for VTR: -0.3475421033665316
R-squared value for VZ: 0.6745034310321225
R-squared value for WAB: -2.7108389407550053
R-squared value for WAT: 0.28050648909659015
R-squared value for WBA: -0.01805920346140022
R-squared value for WDC: -0.004942554728291304
R-squared value for WEC: 0.18742228694946983
R-squared value for WEN: 0.2985805206870803
R-squared value for WFC: 0.49944572258368125
R-squared value for WHR: 0.37195156144681585
R-squared value for WM: 0.16484699322913032
R-squared value for WMB: 0.3783233232969805
R-squared value for WMT: 0.5484649540847872
R-squared value for WOR: -1.8098218703431357
R-squared value for WRB: -3.1709887735579647
R-squared value for WRK: -0.0009010894535343983
R-squared value for WST: -0.37738027079227776
R-squared value for WU: -0.058154185428403116
R-squared value for WY: 0.2945739554988044
R-squared value for WYNN: -0.08367209989473312
R-squared value for X: 0.07494154231324501
R-squared value for XEL: 0.27288853221052733
R-squared value for XOM: 0.5312134196311857
R-squared value for XRAY: 0.2175617730569751
R-squared value for XRX: 0.3506605245317117
R-squared value for XYL: 0.21775675726581933
R-squared value for YUM: 0.31314459034876474
R-squared value for ZBH: -0.39811450276800864
R-squared value for ZBRA: -3.3310534179709883
R-squared value for ZION: 0.2704027606643149
R-squared value for ZTS: -0.1632375804671582
Total R-squared value: -101.13472004603516
```

Model and Data Storage

The following steps ensure that all components of our financial analysis are saved for future use, including reproducibility and further evaluation:

1. **Mount Google Drive: Connects the Google Colab environment to Google Drive to enable file saving and access.**
2. **Save Linear Regression Model:**
    - **The trained linear regression model is serialized and saved to Google Drive using `joblib.dump`. This allows the model to be reloaded and used without retraining.**
3. **Prediction Details:**
    - **The details of each prediction, including ticker, date, features, actual returns, and predicted returns, are compiled into a DataFrame.**
    - **After dropping any rows with missing data, the DataFrame is saved as a CSV file on Google Drive for easy access and analysis.**
4. **R-squared Values:**
    - **A DataFrame of R-squared values, which provides a measure of model performance for each ticker, is transposed and saved as a CSV file. This format facilitates easier viewing and analysis of the data.**
5. **Save Scaler:**
    - **If a scaler is used in data preprocessing (assumed from the context), it is also serialized and saved to ensure consistent data transformation when reloading the model for new predictions.**

In [ ]:

```
import joblib
from google.colab import drive

drive.mount('/content/drive')


model_filename = '/content/drive/My Drive/linear_regression_model_roll.pkl'
joblib.dump(model, model_filename)


prediction_details_df = pd.DataFrame(prediction_details)
prediction_details_df.dropna()
prediction_details_df.to_csv('/content/drive/My Drive/prediction_details_roll.csv', index
=False)
r_squared_values_df.T.to_csv('/content/drive/My Drive/r_squared_values_roll.csv', index=F
alse)
joblib.dump(scaler, '/content/drive/My Drive/scaler.joblib')
```

```
Mounted at /content/drive
```

Out[ ]:

```
['/content/drive/My Drive/scaler.joblib']
```

Overview

This script employs an expanding window linear regression model to predict future excess returns based on financial indicators for different stock tickers. The expanding window approach increases the amount of training data over time, which can improve the model's ability to capture complex patterns as more data becomes available.

Process

1. **Data Preparation:** Each ticker's data is isolated and sorted by the end date.
2. **Model Training and Prediction:**
   - The linear regression model begins with a fixed initial training size and expands the training dataset by incorporating one new data point at a time, retraining the model with each step.
   - This process allows the model to progressively learn from a growing dataset, attempting to improve prediction accuracy over time.
3. **Performance Evaluation:**
   - The model predicts future returns for each time point after the initial training period, and these predictions are compared to the actual future returns to assess model performance.
   - The coefficient of determination (R^2) is calculated for each ticker to evaluate how well the model explains the variability in the observed data.

Features Used

- **Financial metrics such as** `Average_Close`, `Log_Total_Volume`, `Price_Volatility`, `Volume_Indicator`, `Log_MCap`, `MCap_Change`, **and** `High_Low_PCA`.

In [ ]:

```
import numpy as np
from sklearn.linear_model import LinearRegression

feature_names = ['Average_Close', 'Log_Total_Volume', 'Price_Volatility',
                 'Volume_Indicator', 'Log_MCap', 'MCap_Change', 'High_Low_PCA']

model = LinearRegression()

unique_tickers = df['Ticker'].unique()
```

```
prediction_details = []

r_squared_values = []

for ticker in unique_tickers:

    df_stock = df[df['Ticker'] == ticker].sort_values(by='End_date')

    initial_train_size = 30

    if len(df_stock) > initial_train_size:

        predicted_returns = []
        actual_returns_test = []

        for i in range(initial_train_size, len(df_stock)):

            X_train = df_stock.iloc[:i][feature_names].values
            y_train = df_stock.iloc[:i]['Future_Excess_Returns'].values

            model.fit(X_train, y_train)

            X_next = df_stock.iloc[i][feature_names].values
            y_next = df_stock.iloc[i]['Future_Excess_Returns']

            predicted_return = model.predict(X_next.reshape(-1, 7))[0]

            predicted_returns.append(predicted_return)
            actual_returns_test.append(y_next)

            prediction_details.append({
                'Ticker': ticker,
                'Date': df_stock.iloc[i]['End_date'],
                'Features': X_next,
                'Actual_Return': y_next,
                'Predicted_Return': predicted_return
            })

        r_squared_test = r2_score(actual_returns_test, predicted_returns)
        r_squared_values.append((ticker, r_squared_test))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_regression.py:918: UndefinedMetr
icWarning: R^2 score is not well-defined with less than two samples.
  warnings.warn(msg, UndefinedMetricWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_regression.py:918: UndefinedMetr
icWarning: R^2 score is not well-defined with less than two samples.
  warnings.warn(msg, UndefinedMetricWarning)
```

In [ ]:

```
r_squared_values_df = pd.DataFrame(r_squared_values, columns=['ticker', 'r_squared'])

r_squared_values_df.dropna()

total_r_squared_value = r_squared_values_df['r_squared'].sum()

for ticker, r_squared in r_squared_values_df.itertuples(index=False):
    print(f"R-squared value for {ticker}: {r_squared}")

print(f"Total R-squared value: {total_r_squared_value}")
```

```
R-squared value for A: 0.017577716330338666
R-squared value for AA: 0.23177459420490698
R-squared value for AAL: -0.0894520954103375
R-squared value for AAP: -0.004334497391313974
R-squared value for AAPL: -0.2669629818377086
R-squared value for ABBV: 0.12258343629496982
R-squared value for ABMD: -0.04270630188854385
R-squared value for ABT: -0.023555316352775302
R-squared value for ACN: 0.04430208051542728
R-squared value for ADBE: -0.3357407101374106
R-squared value for ADI: -0.1258273918183035
```

R-squared value for ADM: -0.2533262703093486
R-squared value for ADP: -0.2146196837483232
R-squared value for ADSK: -0.37335375583230546
R-squared value for AEE: 0.03619288831592271
R-squared value for AEP: 0.0579830030121371
R-squared value for AES: 0.0808080082334196
R-squared value for AET: -0.28590154119156286
R-squared value for AFL: -0.11101097204760868
R-squared value for AIG: -0.4869196501424198
R-squared value for AIV: -0.12650612058131716
R-squared value for AIZ: -0.0657678576355456
R-squared value for AJG: 0.18172570834660517
R-squared value for AKAM: -0.23111925114837994
R-squared value for ALB: 0.10471433337603797
R-squared value for ALGN: -0.10820148317201306
R-squared value for ALK: -0.3728562657526133
R-squared value for ALL: -0.09599881292750356
R-squared value for ALLE: -0.14934821281749056
R-squared value for AMAT: 0.0066016757938259785
R-squared value for AMCR: -0.7485644819221755
R-squared value for AMD: -0.25714720409412317
R-squared value for AME: 0.033019459065371626
R-squared value for AMG: -0.27146915367560376
R-squared value for AMGN: -0.13349670673340874
R-squared value for AMP: -0.7296344062621063
R-squared value for AMT: 0.023529339275943806
R-squared value for AMZN: -0.5883501690116346
R-squared value for AN: 0.13989964148431966
R-squared value for ANET: 0.40941713499161825
R-squared value for ANF: -0.6646023078509598
R-squared value for ANSS: -0.18104337182817742
R-squared value for AON: 0.45488160331794014
R-squared value for AOS: -0.10837692214394457
R-squared value for APA: -0.07776752060543402
R-squared value for APD: -0.10937223272764784
R-squared value for APH: 0.14469722446102418
R-squared value for APTV: 0.0592017217599462
R-squared value for ARE: 0.39187457957158467
R-squared value for ASH: -0.12364784839605214
R-squared value for ATI: -0.14379785447884075
R-squared value for ATO: 0.4310101541351271
R-squared value for AVB: -0.5412398821143962
R-squared value for AVGO: 0.2052927008566643
R-squared value for AVY: 0.0854749203473315
R-squared value for AWK: -0.006792864736181015
R-squared value for AXP: -0.15359750762521496
R-squared value for AZO: 0.18065231813950877
R-squared value for BA: -0.33092148638092533
R-squared value for BAC: 0.04009369513761252
R-squared value for BAX: -0.19987506838812164
R-squared value for BBY: -0.013717353947070965
R-squared value for BC: -0.5247352945837267
R-squared value for BDX: -0.14872880410127842
R-squared value for BEN: -0.26180664663989583
R-squared value for BIG: -1.2792600580866318
R-squared value for BIIB: -0.16849417869595085
R-squared value for BIO: -2.240306192940467
R-squared value for BK: -0.01298132613116909
R-squared value for BKNG: -0.024726705108285563
R-squared value for BKR: -0.8553093706283021
R-squared value for BLK: 0.00214657328925949
R-squared value for BMS: 0.04840353413049192
R-squared value for BMY: -0.05082043756323951
R-squared value for BR: -0.20909712565126948
R-squared value for BSX: -0.1363906288121548
R-squared value for BWA: 0.15846121423547477
R-squared value for BXP: -0.5914428040575255
R-squared value for C: -0.1281641481722302
R-squared value for CAG: -0.1588494759992909
R-squared value for CAH: 0.07263899652080696
R-squared value for CARR: -3.9594261581571475
R-squared value for CAT: -0.2504085496495816

```
R-squared value for CB: -0.2074373176302895
R-squared value for CBOE: 0.27365579130114637
R-squared value for CBRE: -0.11010991355021327
R-squared value for CCI: 0.022947090515013202
R-squared value for CCL: -0.23647026715557296
R-squared value for CCU: -0.34275992582112114
R-squared value for CDNS: 0.24057233213928164
R-squared value for CDW: -2.2063925128242787
R-squared value for CE: -0.21379923973713866
R-squared value for CF: 0.028507177666944328
R-squared value for CFG: -0.20259410046387583
R-squared value for CHD: -0.1672595230146099
R-squared value for CHRW: -0.10461879234698857
R-squared value for CHTR: 0.401215104827093
R-squared value for CI: -0.4035232105797657
R-squared value for CIEN: 0.08130297742613335
R-squared value for CINF: -0.06451479509387581
R-squared value for CL: -0.08247257037945466
R-squared value for CLF: -0.5461102024361408
R-squared value for CLX: -0.14554865836072484
R-squared value for CMA: -0.009248122861360564
R-squared value for CMCSA: -0.11601388744833496
R-squared value for CME: -0.3704705250293536
R-squared value for CMG: 0.044243993465868825
R-squared value for CMI: -0.09257091344187396
R-squared value for CMS: -0.1983516222851205
R-squared value for CNC: -0.0024390035836197033
R-squared value for CNP: -0.019461904682663622
R-squared value for CNX: -1.4373195441840543
R-squared value for COF: -0.07754138680709821
R-squared value for COL: -1.1281710054888894
R-squared value for COO: -0.2346836975422848
R-squared value for COP: 0.07124099859266575
R-squared value for COST: -0.18168865830783942
R-squared value for COTY: -0.7736895681443694
R-squared value for CPB: -0.07210033849164255
R-squared value for CPRT: 0.06690262306086492
R-squared value for CPWR: -1.7033566196662178
R-squared value for CRL: nan
R-squared value for CRM: 0.10972330555949905
R-squared value for CSCO: 0.07641736488479844
R-squared value for CSX: -0.20532385806710574
R-squared value for CTAS: -0.03034058145722973
R-squared value for CTLT: -0.47783621784680363
R-squared value for CTSH: -0.5070616650545432
R-squared value for CTVA: -0.319613910190345
R-squared value for CVG: 0.4269720865778205
R-squared value for CVS: -0.2305155424030576
R-squared value for CVX: -0.09183384601436417
R-squared value for CZR: -0.012226904250216775
R-squared value for D: -0.24077242392242426
R-squared value for DAL: 0.08705586258380649
R-squared value for DD: -0.08291520093112714
R-squared value for DDS: -0.2468358124566421
R-squared value for DE: -0.2393043914812918
R-squared value for DFS: -0.14482838486317173
R-squared value for DG: -0.17212240251258204
R-squared value for DGX: 0.04025491826939276
R-squared value for DHI: -0.6635693611629851
R-squared value for DHR: -0.3209161333259918
R-squared value for DIS: -0.34662761750459725
R-squared value for DLR: 0.16767379867168886
R-squared value for DLTR: 0.22047501047496176
R-squared value for DLX: -0.7380407251191006
R-squared value for DOV: -0.20524280265615813
R-squared value for DOW: -0.49266169058201803
R-squared value for DPZ: -2.239897981395503
R-squared value for DRI: -0.047204079093946305
R-squared value for DTE: -0.217725335886777
R-squared value for DUK: -0.017332631788821384
R-squared value for DVA: 0.06358354831147339
R-squared value for DVN: -0.27920469183776464
```

```
R-squared value for DXC: -0.36391817708752683
R-squared value for DXCM: -3.9536842334196427
R-squared value for EA: 0.038096668302711856
R-squared value for EBAY: -0.16915769730814323
R-squared value for ECL: 0.0010670175985876273
R-squared value for ED: -0.04141173705563794
R-squared value for EFX: -0.28034075702846417
R-squared value for EIX: -0.13571861980066724
R-squared value for EL: -0.623832287886297
R-squared value for EMN: -0.017706171251542502
R-squared value for EMR: -0.1778452845666325
R-squared value for ENPH: -1.545928553654528
R-squared value for EOG: -0.13109620524922305
R-squared value for EP: 0.05103930852213745
R-squared value for EQIX: -0.03234207468306827
R-squared value for EQR: -0.015161736609864862
R-squared value for EQT: 0.28573340723331053
R-squared value for ES: -0.13816037490826627
R-squared value for ESRX: -0.05324671490818367
R-squared value for ESS: -0.1525525492339197
R-squared value for ETN: -0.07767944993164133
R-squared value for ETR: -0.11967846621468503
R-squared value for ETSY: -0.6541576207637092
R-squared value for EVRG: -0.18682708716234364
R-squared value for EW: 0.0715482385103442
R-squared value for EXC: 0.1462746117553807
R-squared value for EXPD: 0.06499200921288872
R-squared value for EXPE: 0.014719984686448973
R-squared value for EXR: 0.21166996359257728
R-squared value for F: -0.041191219764295184
R-squared value for FANG: 0.19594167418795105
R-squared value for FAST: 0.13988567995916823
R-squared value for FCX: -0.07319056246755662
R-squared value for FDX: -0.12114301690306473
R-squared value for FE: 0.19352541462298034
R-squared value for FFIV: 0.031313020485738696
R-squared value for FHN: 0.1271051365716478
R-squared value for FIS: -0.2604915183436811
R-squared value for FITB: 0.03395771534389991
R-squared value for FL: -0.7077457002541416
R-squared value for FLR: -0.09054885244565503
R-squared value for FLS: -0.09233303151523309
R-squared value for FLT: 0.016831910592724375
R-squared value for FMC: -0.2291651605109406
R-squared value for FOSL: 0.1256490759545722
R-squared value for FOX: -0.4276147471233618
R-squared value for FOXA: -0.32580219162708546
R-squared value for FRT: -0.1927082533361626
R-squared value for FSLR: 0.02905470163265078
R-squared value for FTI: 0.04923142050562568
R-squared value for FTNT: -0.10681017349304267
R-squared value for FTV: 0.0029535189861122912
R-squared value for GD: -0.2129055769866317
R-squared value for GE: 0.19412739966973802
R-squared value for GILD: -0.32165359938352545
R-squared value for GIS: 0.143061362872265556
R-squared value for GL: -0.2335425124995194
R-squared value for GLW: -0.14638208434835276
R-squared value for GM: -0.026036089429952503
R-squared value for GME: -0.2702310570724489
R-squared value for GNRC: -0.30083752826051446
R-squared value for GNW: -1.2890705476845508
R-squared value for GOOG: -0.5757801716681181
R-squared value for GOOGL: -0.07347570971199957
R-squared value for GPC: -0.10228688129159869
R-squared value for GPN: -0.18082297022064542
R-squared value for GPS: -0.0656886951268687
R-squared value for GRMN: -0.038571727074353745
R-squared value for GS: -0.04352915189717055
R-squared value for GT: -0.08408763047055046
R-squared value for GWW: -0.016755826177557775
R-squared value for HAL: -0.16504047134217914
```

R-squared value for HAS: -0.21966337504409283
R-squared value for HBAN: -0.044489392679854856
R-squared value for HBI: -0.015879989762855873
R-squared value for HCA: 0.08061743581322967
R-squared value for HD: -0.2563282522626258
R-squared value for HES: -0.15740584686344272
R-squared value for HIG: -0.10914540281646712
R-squared value for HII: 0.2523829345315959
R-squared value for HLT: -0.011795978200054735
R-squared value for HOG: -1.2106999050011038
R-squared value for HOLX: -0.04235631064269318 5
R-squared value for HON: -0.46477814970048437
R-squared value for HP: -0.14862726294547546
R-squared value for HPE: 0.1926606119259524
R-squared value for HPQ: 0.12372680085895182
R-squared value for HRB: -0.09753149140199713
R-squared value for HRL: 0.0934057804206353
R-squared value for HSIC: -0.018666552717842677
R-squared value for HST: -0.07955572178330805
R-squared value for HSY: -0.24669217568843016
R-squared value for HUM: -0.2425553309239834
R-squared value for HWM: 0.12252364046466235
R-squared value for IBM: -0.3161542422244352
R-squared value for ICE: -0.1768566453850564
R-squared value for IDXX: -0.2756723284920528
R-squared value for IEX: -1.2242399174791654
R-squared value for IFF: -0.21894845211821723
R-squared value for IGT: 0.37699903820390457
R-squared value for ILMN: -0.12323378347652958
R-squared value for INCY: 0.018735944490047673
R-squared value for INTC: -0.07214767476198802
R-squared value for INTU: -0.06488292901685644
R-squared value for IP: -0.1624432395941655
R-squared value for IPG: -0.08860122830285366
R-squared value for IPGP: -1.8894377591362193
R-squared value for IQV: -0.2893329996941949
R-squared value for IR: 0.29972527602570576
R-squared value for IRM: 0.27623440703299584
R-squared value for ISRG: 0.02542596317640533
R-squared value for IT: -0.023553832461571833
R-squared value for ITT: 0.10649119069898483
R-squared value for ITW: -0.2506140646503754
R-squared value for IVZ: -0.04254661320829145
R-squared value for J: -7.67434603318155
R-squared value for JBHT: 0.008037190321245835
R-squared value for JBL: 0.18255337588821 83
R-squared value for JCI: -0.05884150885782491
R-squared value for JKHY: -0.42953820400273 67
R-squared value for JNJ: -0.21722122684140754
R-squared value for JNPR: -0.3126643100308921
R-squared value for JPM: -0.012868203466702033
R-squared value for JWN: -0.2236252737199944
R-squared value for K: -0.16617966580842758
R-squared value for KBH: -0.17326570938897534
R-squared value for KEY: 0.038622540979935116
R-squared value for KEYS: -0.2738751896859579
R-squared value for KHC: -0.0014539385833249874
R-squared value for KIM: -0.43351576230761
R-squared value for KLAC: -0.1952147076328541
R-squared value for KMB: -0.141497048200671
R-squared value for KMG: -2.3192083938883843
R-squared value for KMI: -0.2607610211127376
R-squared value for KMX: -0.21984181871252417
R-squared value for KO: -0.21577429226479694
R-squared value for KR: -0.06411441376376392
R-squared value for KSS: 0.018770995026867077
R-squared value for L: 0.3487720052655755
R-squared value for LDOS: -1.049291300895495
R-squared value for LEG: -0.12096990685171294
R-squared value for LEN: -0.6502363379059004
R-squared value for LH: -0.5816040554286599
R-squared value for LHX: -1.7815734413292188

```
R-squared value for LIN: -0.005459542450665156
R-squared value for LKQ: 0.26297486333089426
R-squared value for LLY: -0.31226905728919907
R-squared value for LMT: -0.18388296019467898
R-squared value for LNC: -0.08885693386175375
R-squared value for LNT: -0.11469130193388088
R-squared value for LOW: -0.1748463105880449
R-squared value for LPX: -0.244786998575095
R-squared value for LRCX: -0.01473579647983203
R-squared value for LUV: -0.06650077660253673
R-squared value for LVS: -1.0668469488744705
R-squared value for LW: 0.6146443721521315
R-squared value for LYB: 0.024237728858734653
R-squared value for LYV: -3.1219067448156146
R-squared value for M: -0.6096748941171921
R-squared value for MA: 0.16623562164908412
R-squared value for MAA: -0.2513286040258995
R-squared value for MAC: 0.44368465733514206
R-squared value for MAR: -0.1863168970219733
R-squared value for MAS: -0.04760446370175053
R-squared value for MAT: -0.02022153101583113
R-squared value for MBI: -0.21635532036380245
R-squared value for MCD: -0.2953457781230635
R-squared value for MCHP: 0.011698110272915785
R-squared value for MCK: -0.09877684072399884
R-squared value for MCO: -0.16613860661970659
R-squared value for MDLZ: 0.18972861700235588
R-squared value for MDT: -0.09523155189105181
R-squared value for MET: -7.354946178028676e-05
R-squared value for MGM: -0.4106944387940019
R-squared value for MHK: 0.058778006768069324
R-squared value for MKC: -0.06311474290906038
R-squared value for MKTX: -0.8098487707749737
R-squared value for MLM: -0.004061342133834689
R-squared value for MMC: -0.2566222731616423
R-squared value for MMM: -0.13941362681127778
R-squared value for MNST: -0.05766054836256229
R-squared value for MO: -0.020600524812312893
R-squared value for MOS: -0.13435691764353042
R-squared value for MPC: 0.27798318596948157
R-squared value for MPWR: -0.7472747448350234
R-squared value for MRK: -0.043051180007240175
R-squared value for MRO: -0.22878421455800435
R-squared value for MS: -0.13775288301415833
R-squared value for MSCI: -0.1386997888799686
R-squared value for MSFT: -0.14729953715221855
R-squared value for MSI: 0.27215193031403206
R-squared value for MTB: -0.02849619113006563
R-squared value for MTD: -0.14370464929140914
R-squared value for MTG: -0.37541618295986945
R-squared value for MU: -0.04820964347511514
R-squared value for MUR: -1.1537310426856595
R-squared value for NAVI: -0.030289049661945233
R-squared value for NBR: 0.14478455474243512
R-squared value for NCLH: -0.15172695698769867
R-squared value for NDAQ: 0.11569782451740374
R-squared value for NEE: 0.08661618299589036
R-squared value for NEM: -0.12561068289493726
R-squared value for NFLX: -0.18078202736892868
R-squared value for NFX: -0.07833429463352037
R-squared value for NI: -0.08581072835928794
R-squared value for NKE: -0.16539432627667416
R-squared value for NOC: -0.27360276863492494
R-squared value for NOV: -0.7028119641126434
R-squared value for NOW: -2.9543651872488863
R-squared value for NRG: 0.18840305682130598
R-squared value for NSC: -0.22567025870066781
R-squared value for NTAP: -0.19917249372093493
R-squared value for NTRS: -0.1587067198881491
R-squared value for NUE: -0.33020494723955563
R-squared value for NVDA: -0.18963619390593056
R-squared value for NVR: 0.06173681381705187
```

```
R-squared value for NWL: 0.07037106485739264
R-squared value for NWS: -0.22205617247874354
R-squared value for NWSA: -0.17121279909084008
R-squared value for NXPI: 0.8603238867146128
R-squared value for NYT: 0.38957345678047084
R-squared value for O: 0.15090354111582194
R-squared value for ODFL: -1.3701167018694327
R-squared value for ODP: -0.17542451513370105
R-squared value for OI: -0.9655042331270736
R-squared value for OKE: 0.1714798738177581
R-squared value for OMC: -0.12867421469646723
R-squared value for ORCL: -0.3119143254261416
R-squared value for ORLY: 0.3104252820832901
R-squared value for OTIS: -4.05145234010005
R-squared value for OXY: -0.3084675717089813
R-squared value for PAYC: -4.876322465839616
R-squared value for PAYX: -0.052290039544376166
R-squared value for PBI: 0.1820040614237045
R-squared value for PCAR: -0.23639109612423814
R-squared value for PCG: 0.09852514727660944
R-squared value for PDCO: -1.2021552886787061
R-squared value for PEAK: -1.0680091504219367
R-squared value for PEG: -0.23219978134384434
R-squared value for PEP: -0.18027315363512986
R-squared value for PFE: -0.12420440005335731
R-squared value for PFG: -0.10164872323206553
R-squared value for PG: -0.20430820967047492
R-squared value for PGR: -0.23613753908010904
R-squared value for PH: -0.18832921919924783
R-squared value for PHM: -0.006060799643627712
R-squared value for PKG: -0.02710952796872257
R-squared value for PLD: -0.5941163972298118
R-squared value for PM: 0.12342113905870111
R-squared value for PNC: -0.0566807338038704
R-squared value for PNR: -0.02077022469545864
R-squared value for PNW: -0.10886249905253176
R-squared value for POOL: -4.5670774910867
R-squared value for PPG: -0.13432585928382612
R-squared value for PPL: 0.09144192978497967
R-squared value for PRGO: -0.114164018563357 23
R-squared value for PRU: 0.05934934198730868
R-squared value for PSA: -0.6611255504175702
R-squared value for PSX: 0.32434751797642003
R-squared value for PTC: -6.906053389410789
R-squared value for PVH: -0.29161051220371004
R-squared value for PWR: 0.35232299376920906
R-squared value for PXD: 0.057849930136872785
R-squared value for PYPL: -0.02036589158923463
R-squared value for QCOM: -0.24906884690672904
R-squared value for QRVO: -0.34605602372302546
R-squared value for R: -0.29635194622925254
R-squared value for RCL: -0.17700929934903176
R-squared value for REG: -0.10335604294398038
R-squared value for REGN: 0.021236994003374865
R-squared value for RF: 0.32797958926092585
R-squared value for RHI: 0.12008641957016875
R-squared value for RIG: -0.12752667374265547
R-squared value for RJF: 0.015526185633496503
R-squared value for RL: -0.40920280981097035
R-squared value for RMD: -0.028569247090991956
R-squared value for ROK: -0.30373738722688715
R-squared value for ROL: -0.17727007123810168
R-squared value for ROP: 0.1855882810958137
R-squared value for ROST: 0.24843879550768821
R-squared value for RRC: -0.3093741655446769
R-squared value for RSG: 0.21885506704451163
R-squared value for RTX: -12.88343841463156
R-squared value for SANM: -0.1362749857449972
R-squared value for SBAC: -0.0547616227001 6016
R-squared value for SBUX: -0.2663898599689267
R-squared value for SCG: -1.5165566081307391
R-squared value for SCHW: -0.664342006047749
```

```
R-squared value for SEE: -0.2269847134595968
R-squared value for SHW: -0.2580030509944664
R-squared value for SIG: nan
R-squared value for SJM: 0.30477667558667687
R-squared value for SLB: -0.3323037581259445
R-squared value for SLG: -0.13129724870038428
R-squared value for SLM: -0.06166706178453207
R-squared value for SNA: -0.1896167835111433
R-squared value for SNI: -0.19080737575087237
R-squared value for SNPS: 0.37563273421851806
R-squared value for SNV: 0.07327430203933927
R-squared value for SO: -0.162312401762696
R-squared value for SPG: -0.05603988699202023
R-squared value for SPGI: 0.33016255529028715
R-squared value for SRCL: 0.07386617564027309
R-squared value for SRE: -0.2810978829280124
R-squared value for STE: -5.075775077647048
R-squared value for STT: -0.15932643452796902
R-squared value for STX: -0.10602432477447032
R-squared value for STZ: -0.8242861699789017
R-squared value for SVU: -0.18257052832695386
R-squared value for SWK: -0.06661559824748897
R-squared value for SWKS: -0.08225668064221425
R-squared value for SWN: -0.33412549600615526
R-squared value for SYF: 0.05326646288908865
R-squared value for SYK: -0.08034161722331667
R-squared value for SYY: -0.08434189015869764
R-squared value for T: -0.18780870190858523
R-squared value for TAP: -0.12277963508345002
R-squared value for TDC: -0.7774386956493387
R-squared value for TDG: -0.1842761601214633
R-squared value for TDY: -2.72281888830435
R-squared value for TEL: -0.17460754674845735
R-squared value for TER: -0.13524071902782508
R-squared value for TFC: -2.417252209258547
R-squared value for TFX: 0.3348456711342904
R-squared value for TGT: -0.11169226489169293
R-squared value for THC: -0.2904235194040621
R-squared value for TJX: 0.14474869932263146
R-squared value for TMO: -0.21757633459753922
R-squared value for TMUS: -0.07174270602558219
R-squared value for TPR: -0.04670877217019997 5
R-squared value for TRIP: -0.42620292221953515
R-squared value for TRMB: -1.965700537382864
R-squared value for TROW: -0.14772460849803148
R-squared value for TRV: -0.0018486313133225796
R-squared value for TSCO: 0.11438710529254792
R-squared value for TSLA: -0.28676882119872293
R-squared value for TSN: -0.35125459482059274
R-squared value for TT: -0.9205041205415108
R-squared value for TTWO: 0.45973766949368045
R-squared value for TUP: -0.28756754488907466
R-squared value for TWX: 0.41251034133145126
R-squared value for TXN: 0.27533117014036057
R-squared value for TXT: -0.1733569524065799
R-squared value for TYL: -3.3390410582405625
R-squared value for UA: -0.009224454590690101
R-squared value for UAA: -0.15168651479267825
R-squared value for UAL: -0.15213547545611217
R-squared value for UDR: 0.32456885868582763
R-squared value for UHS: 0.09447375835586302
R-squared value for UIS: -0.36820807572569314
R-squared value for ULTA: 0.3447432081084669
R-squared value for UNH: -0.16991543194151704
R-squared value for UNM: -0.07179169323191958
R-squared value for UNP: -0.06900836812907762
R-squared value for UPS: -0.08263009610026462
R-squared value for URBN: -0.1954895846761786
R-squared value for URI: -0.01793281564324345
R-squared value for USB: -0.10309940972133114
R-squared value for V: 0.3065001965107156
R-squared value for VFC: -0.0953562604011237 7
```

```
R-squared value for VLO: 0.25503999113001985
R-squared value for VMC: -0.17580304044046247
R-squared value for VNO: -1.2618022188764986
R-squared value for VRSK: 0.020855474422098896
R-squared value for VRSN: -0.46433826237402376
R-squared value for VRTX: 0.2310453739262358
R-squared value for VTR: -0.05178118227417006
R-squared value for VTRS: -3.5079318428898096
R-squared value for VZ: -0.12416031012380158
R-squared value for WAB: 0.11756753101675999
R-squared value for WAT: -0.08937087246483943
R-squared value for WBA: -0.07548077372135076
R-squared value for WDC: -0.04084732153469983
R-squared value for WEC: 0.2045180141078382
R-squared value for WEN: -0.08679919540693515
R-squared value for WFC: 0.1019811684552191
R-squared value for WHR: -0.14863739088332384
R-squared value for WM: -0.12564826006742869
R-squared value for WMB: -0.13948202675922472
R-squared value for WMT: -0.3213006665019693
R-squared value for WOR: -2.5874528035503452
R-squared value for WRB: -2.737149235367387
R-squared value for WRK: -0.09132005921636122
R-squared value for WST: -3.4351219558718213
R-squared value for WU: -0.43928794259586845
R-squared value for WY: 0.017882082165345814
R-squared value for WYNN: -0.37207506101452403
R-squared value for X: 0.13587883809288592
R-squared value for XEL: -0.10750812363310458
R-squared value for XOM: -0.24976254829202738
R-squared value for XRAY: 0.22118810864827598
R-squared value for XRX: 0.27187135773894944
R-squared value for XYL: 0.23112260865180612
R-squared value for YUM: -0.05988221412946704
R-squared value for ZBH: -0.39935911381316247
R-squared value for ZBRA: -5.938984353686698
R-squared value for ZION: 0.28280191203958926
R-squared value for ZTS: -0.03988794448030508
Total R-squared value: -172.77284675219914
```

In [ ]:

```
from sklearn.linear_model import LinearRegression
import joblib
from google.colab import drive

drive.mount('/content/drive')
prediction_details_df = pd.DataFrame(prediction_details)
prediction_details_df.to_csv('/content/drive/My Drive/prediction_details_exp.csv', index=
False)
r_squared_values_df.T.to_csv('/content/drive/My Drive/r_squared_values_exp.csv', index=Fa
lse)

model_filename = '/content/drive/My Drive/linear_regression_model_exp.pkl'
joblib.dump(model, model_filename)
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount
("/content/drive", force_remount=True).
```

Out[ ]:

```
['/content/drive/My Drive/linear_regression_model_exp.pkl']
```

Function: Plot Predicted vs Actual Returns

This function visualizes the comparison between predicted and actual returns for specified stock tickers. It uses a line plot to show both sets of returns over time, enhancing the ability to quickly assess the model's predictive accuracy for each ticker. Key features include different markers and line styles for clarity, a comprehensive legend, and a grid for easy reference. This visualization aids in evaluating model performance and can help in refining prediction strategies.

In [ ]:

```
def plot_predicted_vs_actual_returns(df, ticker_list):
    plt.figure(figsize=(14, 8))

    for ticker in ticker_list:
        df_ticker = df[df['Ticker'] == ticker]

        plt.plot(df_ticker['Date'], df_ticker['Predicted_Return'], label=f'Predicted Ret
urn - {ticker}', marker='o', linestyle='--')
        plt.plot(df_ticker['Date'], df_ticker['Actual_Return'], label=f'Actual Return -
{ticker}', marker='x', linestyle='-')

    plt.title('Predicted vs Actual Returns')
    plt.xlabel('Date')
    plt.ylabel('Returns')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Function: Calculate Portfolio Statistics by Ticker

Overview

This function computes essential portfolio statistics for each ticker within a provided DataFrame. It processes each group of data, segmented by ticker, to determine performance metrics critical for assessing investment strategies.

Process and Metrics Calculated

- **Sorting and Grouping: The data for each ticker is sorted by date to ensure chronological calculations.**
- **Statistics Computed:**
  - **Mean Return: The average return of the strategy for each ticker.**
  - **Standard Deviation: Measures the volatility or risk associated with the ticker's return.**
  - **Cumulative Returns: Product of compounded returns over time, giving a sense of overall strategy effectiveness.**
  - **Maximum Drawdown: The largest peak-to-trough decline in the cumulative returns, an indicator of potential risk from high to low.**
  - **Sharpe Ratio: Calculates risk-adjusted return, which normalizes the returns of an investment compared to its risk.**

In [ ]:

```
def calculate_portfolio_statistics_by_ticker(df):

    stats_list = []

    for ticker, group in df.groupby('Ticker'):
        group = group.sort_values('Date')
        returns = group['Strategy_Returns']

        mean_return = returns.mean()
        std_dev = returns.std()

        cumulative_returns = (1 + returns).cumprod()
        rolling_max = cumulative_returns.cummax()
        max_drawdown = ep.max_drawdown(returns)

        sharpe_ratio = ((mean_return) / std_dev) * (12 ** 0.5) if std_dev != 0 else 0

        stats_list.append({
            'Ticker': ticker,
            'Mean Return': mean_return,
            'Standard Deviation': std_dev,
            'Maximum Drawdown': max_drawdown,
```

```
            'Sharpe Ratio': sharpe_ratio
        })

    return pd.DataFrame(stats_list)
```

Purpose

This function simulates an investment strategy based on the predicted returns of stocks. It allocates a proportion of the total investment to each stock if its predicted return is positive, otherwise it allocates nothing. The function calculates the actual returns based on this strategy.

Calculation Process

1. **Allocation:** Each stock is allocated a portion of the investment depending on whether its predicted return is positive.
2. **Strategy Returns:** Computes the returns from the strategy by multiplying the allocation by the actual returns of the stock.
3. **Buy Signal:** Determines whether to buy the stock (1 if predicted return is positive, otherwise 0).
4. **Adjusted Returns:** Calculates the actual strategy returns by considering only the investments where the buy signal was active.

Output

- **Total Return and ROI:** Calculates the sum of all adjusted returns and the return on investment (ROI), which is the total return divided by the initial investment.

Returns

The function returns the initial investment amount, the total return, and the actual ROI, providing a comprehensive overview of the strategy's performance.

In [ ]:

```
def strategy_proportional_allocation_actual_returns(df, allocation_per_stock=0.002):
    initial_investment = 1000000
    investment_per_stock = initial_investment * allocation_per_stock

    df['Allocation'] = df['Predicted_Return'].apply(
        lambda x: investment_per_stock if x > 0 else 0
    )

    df['Strategy_Returns'] = df['Allocation'] * df['Actual_Return']

    df['Buy'] = df['Predicted_Return'].apply(lambda x: 0 if x < 0 else 1)

    df['Adjusted_Returns'] = df['Buy'] * df['Strategy_Returns']

    total_return = df['Adjusted_Returns'].sum()

    actual_roi = total_return / initial_investment

    return df, initial_investment, total_return, actual_roi
```

Purpose

This function implements an investment strategy focusing on the top-performing stocks based on their actual returns. It allocates an equal portion of the initial investment to each of the top-performing stocks, aiming to capitalize on their historical performance.

Calculation Process

1. **Top Performers Selection:** Calculates the average predicted return for each stock and selects the top `n` performers based on their predicted returns.

2. **Allocation:** Distributes the initial investment equally among the selected top-performing stocks.
3. **Strategy Returns:** Computes the strategy returns by multiplying the allocation by the actual returns of each stock.
4. **Buy Signal:** Determines whether to buy the stock (1 if predicted return is positive, otherwise 0).
5. **Adjusted Returns:** Calculates the adjusted returns considering only the investments where the buy signal was active.

In [ ]:

```
import pandas as pd

def strategy_top_performers_actual_returns(df, top_n=10):
    initial_investment = 100000
    allocation_per_stock = initial_investment / top_n

    avg_predicted_returns = df.groupby('Ticker')['Predicted_Return'].mean().reset_index(
)

    top_stocks = avg_predicted_returns.sort_values(by='Predicted_Return', ascending=Fals
e).head(top_n)

    top_performers = df[df['Ticker'].isin(top_stocks['Ticker'])]

    entries_per_stock = top_performers['Ticker'].value_counts()
    top_performers['Allocation'] = top_performers['Ticker'].apply(lambda x: allocation_p
er_stock / entries_per_stock[x])

    top_performers['Strategy_Returns'] = top_performers['Allocation'] * top_performers['
Actual_Return']

    top_performers['Buy'] = top_performers['Predicted_Return'].apply(lambda x: 0 if x <
0 else 1)
    top_performers['Adjusted_Returns'] = top_performers['Buy'] * top_performers['Strategy
_Returns']
    total_return = top_performers['Adjusted_Returns'].sum()

    actual_roi = total_return / initial_investment

    return top_performers, initial_investment, total_return, actual_roi
```

In [ ]:

```
!pip install empyrical
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
import empyrical as ep
from google.colab import drive
drive.mount("/content/gdrive")


r_squared_values_df = pd.read_csv('r_squared_values_exp.csv')
prediction_details_df = pd.read_csv('prediction_details_exp.csv', encoding='utf-8')

prediction_details_df['Date'] = pd.to_datetime(prediction_details_df['Date'])
```

```
Requirement already satisfied: empyrical in /usr/local/lib/python3.10/dist-packages (0.5.
5)
Requirement already satisfied: numpy>=1.9.2 in /usr/local/lib/python3.10/dist-packages (f
rom empyrical) (1.25.2)
Requirement already satisfied: pandas>=0.16.1 in /usr/local/lib/python3.10/dist-packages
(from empyrical) (2.0.3)
Requirement already satisfied: scipy>=0.15.1 in /usr/local/lib/python3.10/dist-packages (
from empyrical) (1.11.4)
Requirement already satisfied: pandas-datareader>=0.2 in /usr/local/lib/python3.10/dist-p
ackages (from empyrical) (0.10.0)
```

```
ackages (from empyrical) (0.10.0)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.16.1->empyrical) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.16.1->empyrical) (2023.4)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.16.1->empyrical) (2024.1)
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from pandas-datareader>=0.2->empyrical) (4.9.4)
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.10/dist-packages (from pandas-datareader>=0.2->empyrical) (2.31.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas>=0.16.1->empyrical) (1.16.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->pandas-datareader>=0.2->empyrical) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->pandas-datareader>=0.2->empyrical) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->pandas-datareader>=0.2->empyrical) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->pandas-datareader>=0.2->empyrical) (2024.2.2)
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/content/gdrive", force_remount=True).
```

In [ ]:

```
stocks = r_squared_values_df.iloc[0].values
r_squaare_values = r_squared_values_df.iloc[1].values

combined = zip(stocks, r_squaare_values)
r_squared_values_df = pd.DataFrame(combined, columns=['ticker', 'r_squared'])
r_squared_values_df['r_squared'] = pd.to_numeric(r_squared_values_df['r_squared'])
```

1. **Very Good R-squared Values:**
   - **Selects rows from the R-squared DataFrame (`r_squared_values_df`) where the R-squared value is greater than or equal to 0.5.**
2. **Good R-squared Values:**
   - **Selects rows from the R-squared DataFrame where the R-squared value is greater than or equal to 0.2.**
   - **Filters the prediction details DataFrame (`prediction_details_df`) to retain only rows where the ticker is present in the list of tickers with good R-squared values.**
3. **Unique Tickers with Good R-squared Values :**
   - **Extracts the unique tickers from the DataFrame containing tickers with good R-squared values (`good_r_squared_df`).**

In [ ]:

```
very_good_r_squared_df = r_squared_values_df[r_squared_values_df['r_squared'] >= 0.5]
very_good_r_squared_df.shape
```

Out[ ]:

```
(2, 2)
```

In [ ]:

```
good_r_squared_df = r_squared_values_df[r_squared_values_df['r_squared'] >= 0.2]
good_r_squared_df.shape
```

Out[ ]:

```
(48, 2)
```

In [ ]:

```
filtered_df = prediction_details_df[prediction_details_df['Ticker'].isin(good_r_squared_df['ticker'])]
filtered_df.shape
```

Out[ ]:

```
(5130, 5)
```

In [ ]:

```
good_r_squared_df['ticker'].unique()
```

Out[ ]:

```
array(['AA', 'ANET', 'AON', 'ARE', 'ATO', 'AVGO', 'CBOE', 'CDNS', 'CHTR',
       'CVG', 'DLTR', 'EQT', 'EXR', 'HII', 'IGT', 'IR', 'IRM', 'L', 'LKQ',
       'LW', 'MAC', 'MPC', 'MSI', 'NXPI', 'NYT', 'ORLY', 'PSX', 'PWR',
       'RF', 'ROST', 'RSG', 'SJM', 'SNPS', 'SPGI', 'TFX', 'TTWO', 'TWX',
       'TXN', 'UDR', 'ULTA', 'V', 'VLO', 'VRTX', 'WEC', 'XRAY', 'XRX',
       'XYL', 'ZION'], dtype=object)
```

1. **Strategy Evaluation:**
   - **Executes the `strategy_proportional_allocation_actual_returns` function to simulate the investment strategy based on proportional allocation and actual returns.**
   - **Calculates the total invested amount, total return, and return on investment (ROI) for the strategy.**
2. **Portfolio Statistics Calculation:**
   - **Utilizes the `calculate_portfolio_statistics_by_ticker` function to compute essential portfolio statistics for each stock ticker within the DataFrame.**

In [ ]:

```
df = filtered_df.copy()
df_results_1, total_invested_1, total_return_1, roi_1 = strategy_proportional_allocation_
actual_returns(df)
portfolio_stats_1 = calculate_portfolio_statistics_by_ticker(df_results_1)
print(f"Total Invested: ${total_invested_1}, Total Return: ${total_return_1}, ROI: {roi_1
* 100:.2f}%")
```

```
Total Invested: $1000000, Total Return: $543.9679999999971, ROI: 0.05%
```

In [ ]:

```
portfolio_stats_1.describe()
```

Out[ ]:

	Mean Return	Standard Deviation	Maximum Drawdown	Sharpe Ratio
count	48.000000	48.000000	4.800000e+01	48.000000
mean	-1.922075	78.368764	-6.843991e+12	-0.046721
std	9.174430	44.912992	4.553147e+13	0.394813
min	-25.050044	0.000000	-3.155963e+14	-0.940510
25%	-5.822163	51.219354	-3.794371e+08	-0.296006
50%	-1.441895	70.441785	-9.146856e+05	-0.074345
75%	4.281165	110.695189	-4.388535e+02	0.212707
max	19.306161	201.666742	0.000000e+00	0.897995

1. **Strategy Evaluation:**
   - **The `strategy_top_performers_actual_returns` function is executed to simulate the investment strategy, allocating capital equally among the top-performing stocks based on actual returns.**
   - **Total invested amount, total return, and return on investment (ROI) for the strategy are calculated.**
2. **Portfolio Statistics Calculation:**
   - **Portfolio statistics for each stock ticker within the sorted DataFrame (`df_sorted_2`) are computed**

using the `calculate_portfolio_statistics_by_ticker` function.

In [ ]:

```
df = filtered_df.copy()
df_sorted_2, total_invested_2, total_return_2, roi_2 = strategy_top_performers_actual_ret
urns(df)
portfolio_stats_2 = calculate_portfolio_statistics_by_ticker(df_sorted_2)
print(f"Total Invested: ${total_invested_2}, Total Return: ${total_return_2}, ROI: {roi_2
* 100:.2f}%")
```

```
Total Invested: $100000, Total Return: $43.856830091005705, ROI: 0.04%
```

```
<ipython-input-87-5d31d54b1563>:14: 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/stable/user_g
uide/indexing.html#returning-a-view-versus-a-copy
  top_performers['Allocation'] = top_performers['Ticker'].apply(lambda x: allocation_per_
stock / entries_per_stock[x])
<ipython-input-87-5d31d54b1563>:16: 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/stable/user_g
uide/indexing.html#returning-a-view-versus-a-copy
  top_performers['Strategy_Returns'] = top_performers['Allocation'] * top_performers['Act
ual_Return']
<ipython-input-87-5d31d54b1563>:18: 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/stable/user_g
uide/indexing.html#returning-a-view-versus-a-copy
  top_performers['Buy'] = top_performers['Predicted_Return'].apply(lambda x: 0 if x < 0 e
lse 1)
<ipython-input-87-5d31d54b1563>:19: 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/stable/user_g
uide/indexing.html#returning-a-view-versus-a-copy
  top_performers['Adjusted_Returns'] = top_performers['Buy'] * top_performers['Strategy_R
eturns']
```

In [ ]:

```
portfolio_stats_2.describe()
```

Out[ ]:

	Mean Return	Standard Deviation	Maximum Drawdown	Sharpe Ratio
count	10.000000	10.000000	1.000000e+01	10.000000
mean	-6.691581	12.272132	-4.597933e+06	-1.782355
std	5.658960	7.615105	1.451310e+07	0.246375
min	-22.478065	8.284305	-4.590294e+07	-2.348838
25%	-5.039820	9.059194	-1.620825e+04	-1.859682
50%	-4.642497	9.460763	-7.945485e+03	-1.772992
75%	-4.396088	10.287706	-1.768264e+02	-1.626264
max	-4.082546	33.150983	-3.839589e+01	-1.478648

In [ ]:

```
df_sorted_2['Ticker'].unique()
```

Out[ ]:

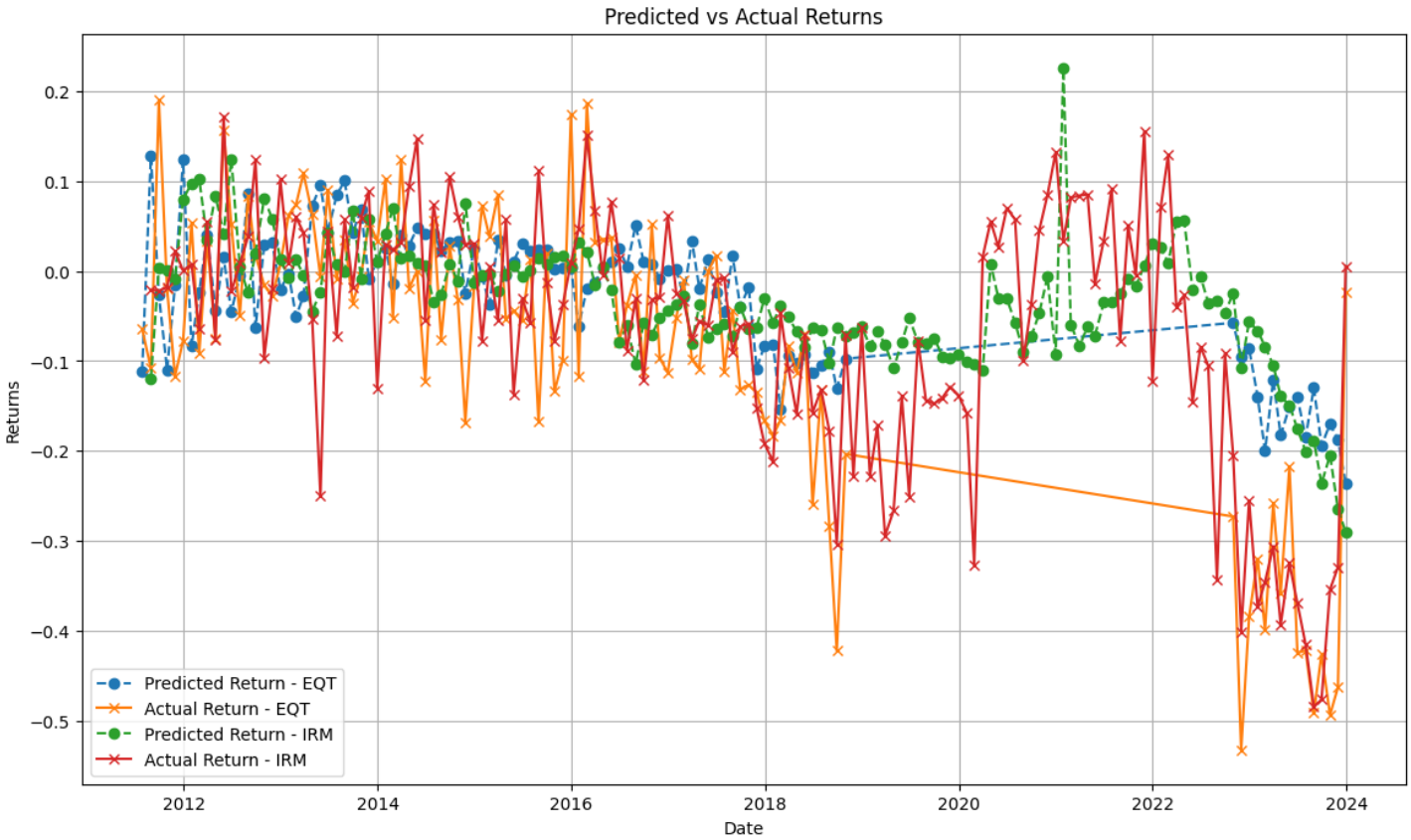
```
array(['EQT', 'IRM', 'L', 'LKQ', 'ORLY', 'PWR', 'RSG', 'SJM', 'WEC',
       'XRAY'], dtype=object)
```

The resulting visualization allows for a direct comparison between the predicted returns from the investment strategy and the actual returns observed for the selected tickers.

As you can see the model is able to learns and follows the patterns well but it also struggles at points especially during time of COVID (2020 - 2023)

In [ ]:

```
selected_tickers = ['EQT', 'IRM']
plot_predicted_vs_actual_returns(df_sorted_2, selected_tickers)
```



Strategy applied using Rolling Window Approach

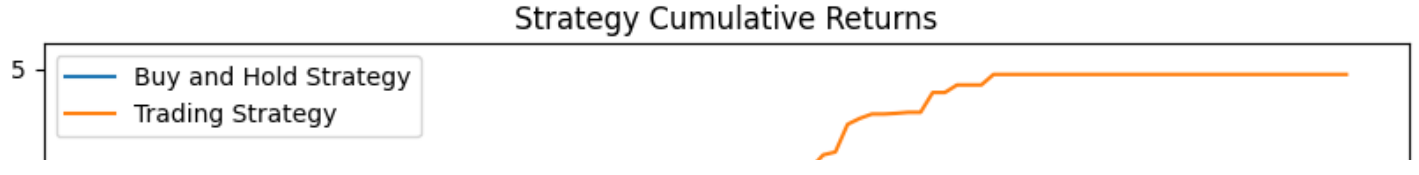
In [ ]:

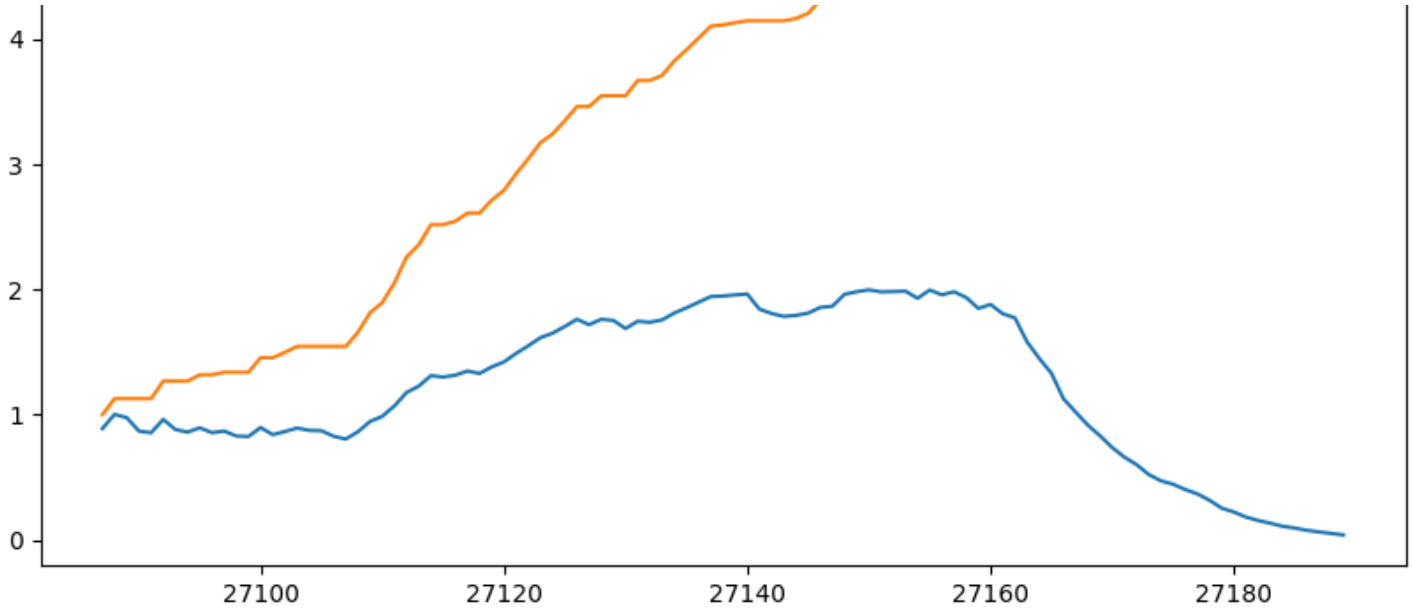
```
selected_tickers = ['EQT']
selected_data = df_sorted_2[df_sorted_2['Ticker'].isin(selected_tickers)]

Buy_and_Hold_Returns = selected_data['Predicted_Return']
Strategy_Returns = selected_data['Predicted_Return'] * selected_data['Buy']

Buy_and_Hold_Returns_cr = (1 + Buy_and_Hold_Returns).cumprod()
Strategy_Returns_cr = (1 + Strategy_Returns).cumprod()

plt.figure(figsize=(10, 5))
plt.plot(Buy_and_Hold_Returns_cr, label='Buy and Hold Strategy')
plt.plot(Strategy_Returns_cr, label='Trading Strategy')
plt.legend()
plt.title('Strategy Cumulative Returns')
plt.show()
```





In [ ]:

```
r_squared_values_df = pd.read_csv('r_squared_values_roll.csv')
prediction_details_df = pd.read_csv('prediction_details_roll.csv')

prediction_details_df['Date'] = pd.to_datetime(prediction_details_df['Date'])

stocks = r_squared_values_df.iloc[0].values
r_squaare_values = r_squared_values_df.iloc[1].values

combined = zip(stocks, r_squaare_values)
r_squared_values_df = pd.DataFrame(combined, columns=['ticker', 'r_squared'])
r_squared_values_df['r_squared'] = pd.to_numeric(r_squared_values_df['r_squared'])
good_r_squared_df = r_squared_values_df[r_squared_values_df['r_squared'] >= 0.2]
good_r_squared_df.shape
```

Out[ ]:

```
(256, 2)
```

In [ ]:

```
filtered_df = prediction_details_df[prediction_details_df['Ticker'].isin(good_r_squared_d
f['ticker'])]
filtered_df.shape
```

Out[ ]:

```
(51599, 5)
```

In [ ]:

```
good_r_squared_df['ticker'].unique()
```

Out[ ]:

```
array(['AA', 'ABT', 'ADBE', 'ADI', 'ADM', 'ADP', 'AEE', 'AEP', 'AES',
       'AET', 'AFL', 'AIZ', 'AJG', 'AKAM', 'ALL', 'AMAT', 'AMGN', 'AMT',
       'AN', 'AON', 'APD', 'ARE', 'ASH', 'AVY', 'AXP', 'AZO', 'BA', 'BAC',
       'BAX', 'BBY', 'BDX', 'BEN', 'BK', 'BMS', 'BMY', 'BSX', 'CAG',
       'CAH', 'CAT', 'CB', 'CBOE', 'CCI', 'CDNS', 'CHRW', 'CHTR', 'CI',
       'CINF', 'CL', 'CLX', 'CMA', 'CMCSA', 'CMS', 'CNP', 'COF', 'COP',
       'COST', 'CPB', 'CSCO', 'CSX', 'CTAS', 'CTSH', 'CVG', 'CVS', 'CVX',
       'D', 'DD', 'DE', 'DGX', 'DIS', 'DOV', 'DRI', 'DTE', 'DUK', 'DVA',
       'EA', 'EBAY', 'ECL', 'ED', 'EFX', 'EIX', 'EMN', 'EMR', 'EP',
       'EQIX', 'EQR', 'EQT', 'ETN', 'ETR', 'EXC', 'EXPD', 'FAST', 'FDX',
       'FE', 'FIS', 'FLR', 'GD', 'GE', 'GILD', 'GIS', 'GLW', 'GOOG',
       'GPC', 'GPN', 'GS', 'GT', 'GWW', 'HAS', 'HBAN', 'HD', 'HON', 'HPQ',
       'HRB', 'HRL', 'HSY', 'HUM', 'HWM', 'IBM', 'IFF', 'IGT', 'INTC',
       'INTU', 'IP', 'IPG', 'IR', 'ITT', 'ITW', 'JBL', 'JNJ', 'JPM',
       'JWN', 'K', 'KBH', 'KEY', 'KLAC', 'KMB', 'KO', 'KR', 'KSS', 'L',
       'LEG', 'LH', 'LKQ', 'LLY', 'LMT', 'LOW', 'LUV', 'LW', 'MA', 'MAC',
```

```
'MAR', 'MAS', 'MAT', 'MCD', 'MCK', 'MCO', 'MDLZ', 'MDT', 'MKC',
'MMC', 'MMM', 'MO', 'MRK', 'MS', 'MSFT', 'MSI', 'MTB', 'NBR', 'NI',
'NKE', 'NOC', 'NSC', 'NTRS', 'NUE', 'NYT', 'OMC', 'ORCL', 'ORLY',
'PAYX', 'PBI', 'PCAR', 'PCG', 'PEG', 'PEP', 'PFE', 'PG', 'PGR',
'PH', 'PM', 'PNC', 'PNW', 'PPG', 'PPL', 'PRU', 'PSX', 'PWR', 'R',
'REGN', 'RF', 'RHI', 'ROK', 'SBAC', 'SBUX', 'SEE', 'SHW', 'SJM',
'SLB', 'SLM', 'SNA', 'SNPS', 'SNV', 'SO', 'SPGI', 'SRE', 'SVU',
'SWK', 'SYK', 'SYY', 'T', 'TGT', 'TJX', 'TROW', 'TRV', 'TSCO',
'TTWO', 'TWX', 'TXN', 'UDR', 'UHS', 'ULTA', 'UNH', 'UNM', 'UNP',
'UPS', 'USB', 'V', 'VFC', 'VLO', 'VMC', 'VRSK', 'VRSN', 'VRTX',
'VZ', 'WAT', 'WEN', 'WFC', 'WHR', 'WMB', 'WMT', 'WY', 'XEL', 'XOM',
'XRAY', 'XRX', 'XYL', 'YUM', 'ZION'], dtype=object)
```

In [ ]:

```
df = filtered_df.copy()
df_results_1, total_invested_1, total_return_1, roi_1 = strategy_proportional_allocation_
actual_returns(df)
portfolio_stats_1 = calculate_portfolio_statistics_by_ticker(df_results_1)
print(f"Total Invested: ${total_invested_1}, Total Return: ${total_return_1}, ROI: {roi_1
* 100:.2f}%")
```

```
Total Invested: $1000000, Total Return: $-104584.52000000005, ROI: -10.46%
```

In [ ]:

```
portfolio_stats_1.describe()
```

Out[ ]:

	Mean Return	Standard Deviation	Maximum Drawdown	Sharpe Ratio
count	256.000000	256.000000	2.560000e+02	256.000000
mean	-2.455721	91.442054	-1.027942e+24	-0.085820
std	7.343579	28.442513	1.020694e+25	0.262191
min	-36.160049	0.000000	-1.211486e+26	-0.931099
25%	-5.731254	75.827452	-6.699882e+12	-0.221587
50%	-2.290960	89.115136	-1.191592e+10	-0.086125
75%	1.923285	103.892141	-3.242635e+07	0.070233
max	18.951143	197.540809	0.000000e+00	0.684417

1. **Strategy Evaluation:**
   - **The `strategy_top_performers_actual_returns` function is executed to simulate the investment strategy, allocating capital equally among the top-performing stocks based on actual returns.**
   - **Total invested amount, total return, and ROI (Return on Investment) for the strategy are calculated.**
2. **Portfolio Statistics Calculation:**
   - **Portfolio statistics for each stock ticker within the sorted DataFrame (`df_sorted_2`) are computed using the `calculate_portfolio_statistics_by_ticker` function.**

In [ ]:

```
df = filtered_df.copy()
df_sorted_2, total_invested_2, total_return_2, roi_2 = strategy_top_performers_actual_ret
urns(df)
portfolio_stats_2 = calculate_portfolio_statistics_by_ticker(df_sorted_2)
print(f"Total Invested: ${total_invested_2}, Total Return: ${total_return_2}, ROI: {roi_2
* 100:.2f}%")
```

```
Total Invested: $100000, Total Return: $-110.26273567268449, ROI: -0.11%
```

```
<ipython-input-87-5d31d54b1563>:14: 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/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  top_performers['Allocation'] = top_performers['Ticker'].apply(lambda x: allocation_per_stock / entries_per_stock[x])
<ipython-input-87-5d31d54b1563>:16: 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/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  top_performers['Strategy_Returns'] = top_performers['Allocation'] * top_performers['Actual_Return']
<ipython-input-87-5d31d54b1563>:18: 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/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  top_performers['Buy'] = top_performers['Predicted_Return'].apply(lambda x: 0 if x < 0 else 1)
<ipython-input-87-5d31d54b1563>:19: 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/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  top_performers['Adjusted_Returns'] = top_performers['Buy'] * top_performers['Strategy_Returns']
```

In [ ]:

```
portfolio_stats_2.describe()
```

Out[ ]:

	Mean Return	Standard Deviation	Maximum Drawdown	Sharpe Ratio
count	10.000000	10.000000	1.000000e+01	10.000000
mean	-4.642358	9.778078	-5.952609e+06	-1.607233
std	1.940211	2.898653	1.837771e+07	0.234520
min	-9.496498	7.752884	-5.824611e+07	-1.940927
25%	-5.223767	8.236072	-6.504428e+04	-1.793326
50%	-4.055333	8.996776	-3.836643e+03	-1.593866
75%	-3.329649	9.887181	-4.648047e+02	-1.391932
max	-2.994205	17.637862	-2.306642e+02	-1.300332

In [ ]:

```
df_sorted_2['Ticker'].unique()
```

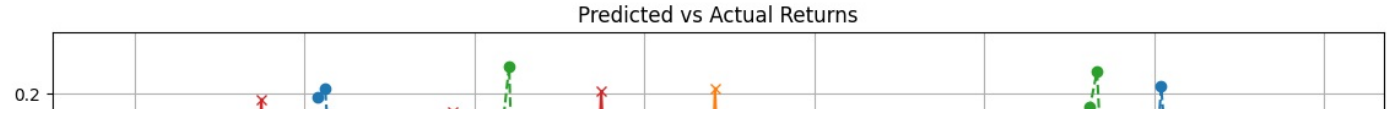
Out[ ]:

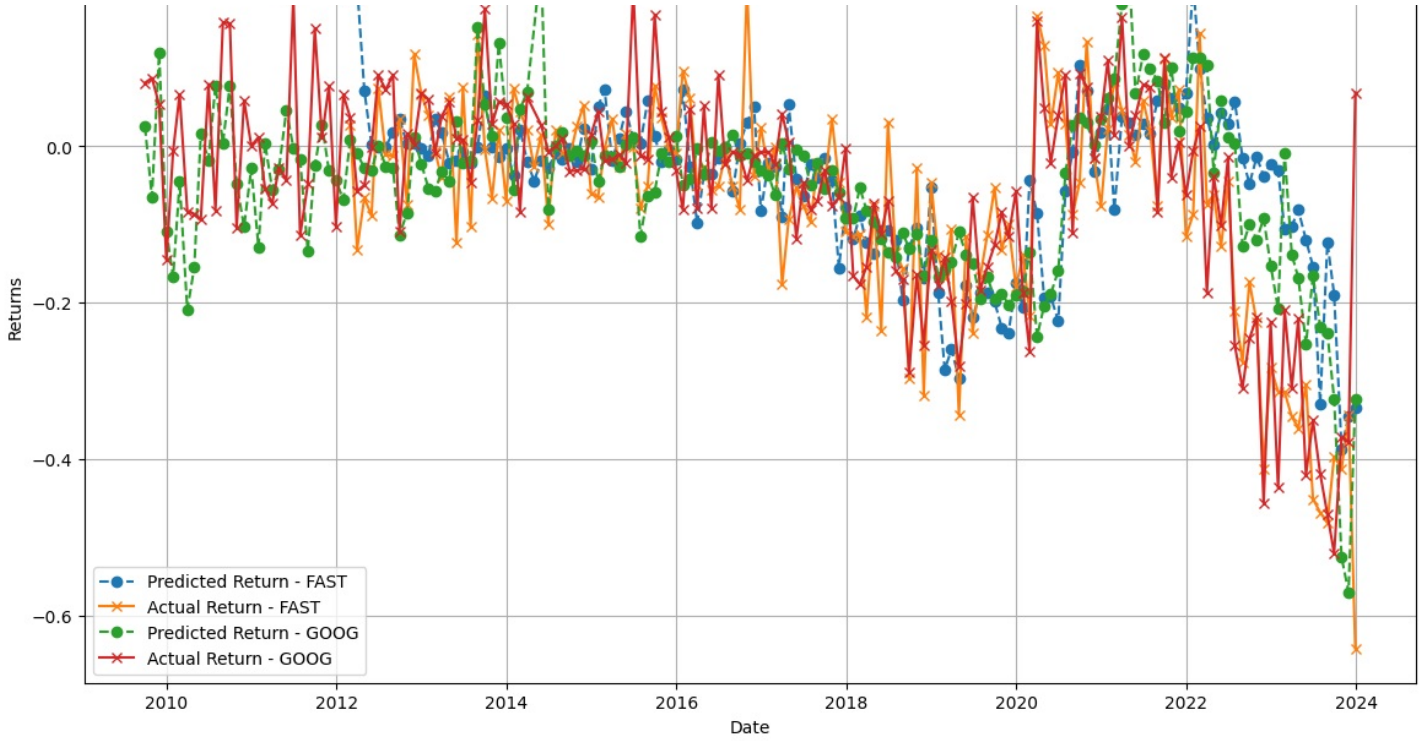
```
array(['AIZ', 'EQT', 'FAST', 'GOOG', 'MA', 'MS', 'PM', 'SJM', 'TRV',
       'VRSN'], dtype=object)
```

Here since,only top performerers are considered the predictions follow the actual closing values better.

In [ ]:

```
selected_tickers = ['FAST', 'GOOG']
plot_predicted_vs_actual_returns(df_sorted_2, selected_tickers)
```





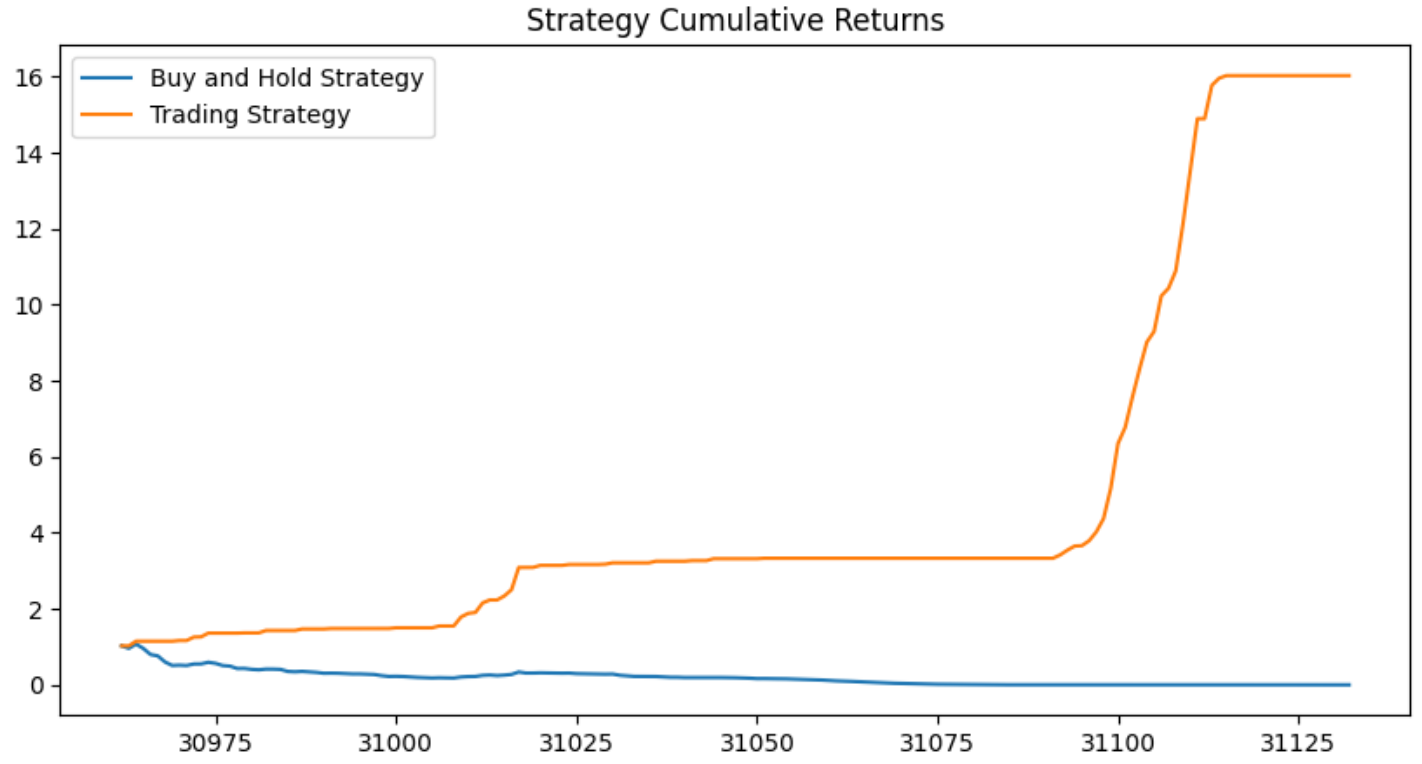
In [ ]:

```
selected_tickers = ['GOOG']
selected_data = df_sorted_2[df_sorted_2['Ticker'].isin(selected_tickers)]

Buy_and_Hold_Returns = selected_data['Predicted_Return']
Strategy_Returns = selected_data['Predicted_Return'] * selected_data['Buy']

Buy_and_Hold_Returns_cr = (1 + Buy_and_Hold_Returns).cumprod()
Strategy_Returns_cr = (1 + Strategy_Returns).cumprod()

plt.figure(figsize=(10, 5))
plt.plot(Buy_and_Hold_Returns_cr, label='Buy and Hold Strategy')
plt.plot(Strategy_Returns_cr, label='Trading Strategy')
plt.legend()
plt.title('Strategy Cumulative Returns')
plt.show()
```



The `get_shares_outstanding` function *retrieves information about the total number of shares outstanding* for a given stock ticker symbol using Yahoo Finance.

for a given stock ticker symbol using Yahoo Finance.

The `fetch_volume_indicator` function *computes a volume indicator based on the logarithm of total volume traded*, providing insights into trading activity trends over monthly intervals.

In [ ]:

```
def get_shares_outstanding(ticker):
    stock = yf.Ticker(ticker)
    info = stock.info
    shares_outstanding = info.get('sharesOutstanding')
    return shares_outstanding
```

In [ ]:

```
def fetch_volume_indicator(data):

    data['Log_Volume'] = np.log(data['Volume'])

    monthly_data = data.resample('M').agg({
        'Log_Volume': ['sum', 'mean']
    })

    monthly_data.columns = ['Total_Log_Volume', 'Avg_Log_Volume']

    monthly_data['Volume_Indicator'] = monthly_data['Total_Log_Volume'] / monthly_data['
Avg_Log_Volume']

    return monthly_data[['Volume_Indicator']]
```

In [ ]:

```
def predict_excess_returns(model, data):

    predicted_excess_returns = model.predict(data.values.reshape(-1,7))

    return predicted_excess_returns[0]
```

In [ ]:

```
import yfinance as yf
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import joblib


model = joblib.load("linear_regression_model_exp.pkl")
scaler = joblib.load('scaler.joblib')
```

```
Means:  [ 7.72845951e+01  1.78876719e+01  1.36686360e-01  1.20757331e+00
  1.68121825e+01  4.11547004e-03 -1.29331197e-14]
Scales (Standard deviations):  [3.42774072e+02 1.31673496e+00 3.31132394e-01 6.70100146e-
02
 1.14810221e+00 9.99968304e-02 4.92293065e+02]
Variances:  [1.17494064e+05 1.73379095e+00 1.09648663e-01 4.49034206e-03
 1.31813868e+00 9.99936610e-03 2.42352461e+05]
```

In [ ]:

```
r_squared_values_df = pd.read_csv('r_squared_values_exp.csv')
stocks = r_squared_values_df.iloc[0].values
r_squaare_values = r_squared_values_df.iloc[1].values

combined = zip(stocks, r_squaare_values)
r_squared_values_df = pd.DataFrame(combined, columns=['ticker', 'r_squared'])
r_squared_values_df['r_squared'] = pd.to_numeric(r_squared_values_df['r_squared'])
good_r_squared_df = r_squared_values_df[r_squared_values_df['r_squared'] >= 0.2]
desired_stocks = good_r_squared_df['ticker'].tolist()
print(desired_stocks)
```

```
['AA', 'ANET', 'AON', 'ARE', 'ATO', 'AVGO', 'CBOE', 'CDNS', 'CHTR', 'CVG', 'DLTR', 'EQT',
'EXR', 'HII', 'IGT', 'IR', 'IRM', 'L', 'LKQ', 'LW', 'MAC', 'MPC', 'MSI', 'NXPI', 'NYT', '
ORLY', 'PSX', 'PWR', 'RF', 'ROST', 'RSG', 'SJM', 'SNPS', 'SPGI', 'TFX', 'TTWO', 'TWX', 'T
XN', 'UDR', 'ULTA', 'V', 'VLO', 'VRTX', 'WEC', 'XRAY', 'XRX', 'XYL', 'ZION']
```

This code fetches historical stock market data for a list of desired stocks, prepares the data by calculating monthly metrics and scaling the features, and then uses a trained machine learning model to predict excess returns for each stock.

1. **Fetching Historical Data:**
   - **Utilizes the Yahoo Finance API to fetch historical stock market data for each desired stock.**
   - **Calculates monthly metrics such as average close price, total volume, average volume, logarithm of total volume, logarithm of average volume, price volatility, volume indicator, market capitalization (MCap), MCap change, and principal component analysis (PCA) of high-low prices.**
2. **Scaling Features:**
   - **Scales the extracted features using a pre-trained scaler.**
3. **Predicting Excess Returns:**
   - **Uses a trained machine learning model to predict excess returns for each desired stock based on the prepared features.**
   - **Stores the predicted excess returns in a dictionary for each stock.**

In [ ]:

```
from datetime import datetime, timedelta

def fetch_and_prepare_data(ticker):
    stock = yf.Ticker(ticker)
    today = datetime.today()

    first_day_current_month = today.replace(day=1)

    last_day_previous_month = first_day_current_month - timedelta(days=1)

    first_day_previous_month = last_day_previous_month.replace(day=1)

    last_day_2_previous_month = first_day_previous_month - timedelta(days=1)
    first_day_2_previous_month = last_day_2_previous_month.replace(day=1)

    start_date = first_day_2_previous_month.strftime('%Y-%m-%d')
    end_date = last_day_previous_month.strftime('%Y-%m-%d')

    stock = yf.Ticker(ticker)
    data = stock.history(start=start_date, end=end_date, interval="1d")

    data.index = pd.to_datetime(data.index)

    monthly_data = data.resample('M').agg({
        'Close': 'mean',
        'Volume': ['sum', 'mean']
    })

    monthly_data.columns = ['Average_Close', 'Total_Volume', 'Average_Volume']


    monthly_data['Log_Total_Volume'] = np.log(monthly_data['Total_Volume'])
    monthly_data['Log_Avg_Volume'] = np.log(monthly_data['Average_Volume'])

    monthly_data['Price_Volatility'] = data['Close'].resample('M').apply(lambda x: np.st
d(x.pct_change()))
    monthly_data['Volume_Indicator'] = monthly_data['Log_Total_Volume']/monthly_data['Lo
g_Avg_Volume']

    shares_outstanding = get_shares_outstanding(ticker)  # Example: 1 billion shares
    if not shares_outstanding:
        return None
```

```
    monthly_data['Log_MCap'] = np.log(monthly_data['Average_Close'] * shares_outstanding
)
    monthly_data['MCap_Change'] = monthly_data['Average_Close'].pct_change()

    high_low = data[['High', 'Low']].resample('M').mean()
    pca = PCA(n_components=1)
    high_low_pca = pca.fit_transform(high_low)
    monthly_data['High_Low_PCA'] = high_low_pca.flatten()

    return monthly_data


features = ['Average_Close', 'Log_Total_Volume', 'Price_Volatility',
                'Volume_Indicator', 'Log_MCap', 'MCap_Change', 'High_Low_PCA']

predicted_returns = dict()
for each_stock in desired_stocks:
    data = fetch_and_prepare_data(each_stock)
    if  data is None or data.empty:
        continue
    data[features] = scaler.transform(data[features])
    final_data = data[features].copy()
    final_data.dropna(inplace=True)
    predicted_return = predict_excess_returns(model, final_data)
    predicted_returns[each_stock] = predicted_return
```

```
ERROR:yfinance:TWX: No price data found, symbol may be delisted (1d 2024-02-01 -> 2024-03
-31)
```

In [ ]:

```
predicted_returns
```

Out[ ]:

```
{'AA': 0.11577623746317946,
 'ANET': -2.2023361068360208,
 'AON': -2.643384345463992,
 'ARE': -0.841126686714483,
 'ATO': -0.7437147233889606,
 'AVGO': -11.01001041976303,
 'CBOE': -1.3106367963984633,
 'CDNS': -2.445848286908556,
 'CHTR': -2.2442308875762773,
 'DLTR': -0.8684948415949354,
 'EQT': 0.045096683329339915,
 'EXR': -0.9751499090235269,
 'HII': -2.313830178699494,
 'IGT': 0.14690922967621184,
 'IR': -0.5138996866817127,
 'IRM': -0.45373994420621283,
 'L': -0.38050714002384456,
 'LKQ': -0.14254416312399543,
 'LW': -0.5975959140206721,
 'MAC': 0.22512277395611663,
 'MPC': -1.4771713167595604,
 'MSI': -2.834522948105522,
 'NXPI': -1.96613616515126,
 'NYT': -0.055641804388430395,
 'ORLY': -9.755211424718356,
 'PSX': -1.1226598981082605,
 'PWR': -2.0398011631801185,
 'RF': 0.18435775251736586,
 'ROST': -1.027495774707542,
 'RSG': -1.362953482231164,
 'SJM': -0.7544898345778027,
 'SNPS': -4.750645135449059,
 'SPGI': -3.5466440813892506,
 'TFX': -1.5676127520411438,
 'TTWO': -0.9511188587635718,
 'TXN': -1.3108572349817469,
 'UDR': 0.010615961616740874,
 'ULTA': -4.384883714605717
```

```
 'V': -2.2990787139598425,
 'VLO': -1.192293304762292,
 'VRTX': -3.3847214215344996,
 'WEC': -0.4094377184503076,
 'XRAY': 0.05903141433542,
 'XRX': 0.26360851674430796,
 'XYL': -0.838397306417761,
 'ZION': 0.009765238693539215}
```

In [ ]:

```
predicted_returns_df = pd.DataFrame(list(predicted_returns.items()), columns=['Ticker',
'Predicted_Return'])
predicted_returns_df.to_csv('final_predicted_returns.csv', index=False)
```

Automated Stock Trading with Interactive Brokers (IBKR) API

The provided code demonstrates *automated stock trading using the Interactive Brokers (IBKR) API for executing buy orders based on predicted returns generated.* The code fetches live prices using Yahoo Finance API, retrieves a list of stocks with positive predicted returns, calculates the investment amount per stock, and places limit orders accordingly through the IBKR API.

The code is *executed in PyCharm due to the live connection requirements with the IBKR API.*

1. **Initialization:**
   - **Establishes a connection with the IBKR API using a custom wrapper class (`IBapi`), which handles order placement and communication with the IBKR servers.**
2. **Order Placement:**
   - **Fetches predicted returns for desired stocks from a CSV file.**
   - **Filters stocks with positive predicted returns and fetches live prices for these stocks using Yahoo Finance API.**
   - **Calculates the investment amount per stock and places limit orders through the IBKR API.**
3. **Execution:**
   - **The code runs in a loop, continuously checking for the next valid order ID from the IBKR API and placing orders accordingly until all orders are placed.**
4. **Termination:**
   - **Disconnects from the IBKR API once all orders are placed and execution is complete.**

- **The code *automates the process of placing buy orders for stocks with positive predicted returns, facilitating algorithmic trading strategies based on machine learning predictions and real-time market data.***

In [ ]:

```
from ibapi.client import EClient
from ibapi.wrapper import EWrapper
from ibapi.contract import Contract
from ibapi.order import *
from ib_insync import *
import pandas as pd
import threading
import time
import yfinance as yf

class IBapi(EWrapper, EClient):
    def __init__(self):
        EClient.__init__(self, self)
        self.data = []
        self.nextorderId = None

    def historicalData(self, reqId, bar):
        print(f'Time: {bar.date} Close: {bar.close}')
        self.data.append([bar.date, bar.close])
```

```
    def nextValidId(self, orderId: int):
        super().nextValidId(orderId)
        self.nextorderId = orderId
        print('The next valid order id is: ', self.nextorderId)

    def orderStatus(self, orderId, status, filled, remaining, avgFullPrice,
                    permId, parentId, lastFillPrice, clientId, whyHeld, mktCapPrice):
        print(f'orderStatus - orderid: {orderId}, status: {status}, filled: {filled}, '
              f'remaining: {remaining}, lastFillPrice: {lastFillPrice}')

    def openOrder(self, orderId, contract, order, orderState):
        print(f'openOrder id: {orderId}, {contract.symbol}, {contract.secType}, @{contrac
t.exchange}: '
              f'{order.action}, {order.orderType}, {order.totalQuantity}, {orderState.st
atus}')

    def execDetails(self, reqId, contract, execution):
        print('Order Executed: ', reqId, contract.symbol, contract.secType, contract.cur
rency,
              execution.execId, execution.orderId, execution.shares, execution.lastLiqui
dity)

    def error(self, reqId, errorCode, errorString):
        print(f"Error - ReqId: {reqId}, ErrorCode: {errorCode}, ErrorString: {errorString
}")
        if errorCode == 202:
            print('Order Canceled - OrderId:', reqId)

def run_loop():
    app.run()

def stock_order(symbol):
    contract = Contract()
    contract.symbol = symbol
    contract.secType = 'STK'
    contract.exchange = 'SMART'
    contract.currency = 'USD'
    return contract

def place_order(app, stock_name, qty, action='BUY', ord_type='LMT', limit_price=None):
    order = Order()
    order.action = action
    order.totalQuantity = qty
    order.orderType = ord_type
    if limit_price:
        order.lmtPrice = limit_price
    order.eTradeOnly = False
    order.firmQuoteOnly = False
    order.outsideRth = True
    app.placeOrder(app.nextorderId, stock_order(stock_name), order)
    app.nextorderId += 1

app = IBapi()
app.connect('127.0.0.1', 7497, 123)

api_thread = threading.Thread(target=run_loop, daemon=True)
api_thread.start()

while True:
    if isinstance(app.nextorderId, int):
        print('Connected and ready to place orders.')
        break
    else:
        print('Waiting for connection...')
        time.sleep(1)


def fetch_live_prices(tickers):
    prices = {}
    for ticker in tickers:
        try:
```

```
            stock = yf.Ticker(ticker)
            ask_price = stock.info['ask']
            prices[ticker] = ask_price
        except Exception as e:
            print(f"Failed to retrieve price for {ticker}: {e}")
            prices[ticker] = None
    return prices

investment_amount = 10000

df = pd.read_csv('final_predicted_returns.csv')
df = df[df['Predicted_Return'] > 0]

tickers = df['Ticker'].tolist()
prices = fetch_live_prices(tickers)
df['Price'] = df['Ticker'].map(prices)

investment_per_stock = investment_amount / len(df)
df['Quantity'] = (investment_per_stock / df['Price']).astype(int)

for index, row in df.iterrows():
    stock = row['Ticker'].upper()
    quantity = row['Quantity']
    price = row['Price']

    place_order(app, stock_name=stock, qty=quantity, action='BUY', ord_type='LMT', limit
_price=price)

    time.sleep(3)

app.disconnect()
```