# DTSA 5509 Final Project: Stock Market Prediction

## 1. Project Introduction

While the stock market offers vast investment opportunities, it also carries enormous risk due to its inherent unpredictability. Investment decisions often rely on educated guesses, which can sometimes lead to costly mistakes. This unpredictability of stock price movements has always posed a tough challenge for investors. Several factors contribute to this uncertainty. First, stock prices are highly sensitive to a wide range of information, such as company performance. For example, on April 12, 2025, news of the Department of Justice's investigation into UnitedHealth Group for potential Medicare fraud led to an over 13% drop in their stock price overnight, illustrating how quickly negative news can impact market value. Second, the market is influenced by human behavior, which is often driven by irrational emotions like fear and greed. For instance, the 2008 financial crisis was triggered by the subprime mortgage crisis, but it was worsened by widespread investor panic selling, which crashed the market. Third, unforeseen events such as natural disasters and political instability can cause sudden shifts in the market, making it less predictable. For example, the COVID-19 pandemic in 2020 led to lockdowns, business closures, and a global economic downturn, which resulted in a steep decline in stock markets worldwide. In the face of such unpredictability, machine learning offers a potential solution. By applying supervised learning techniques, models can be trained on historical market data can potentially predict future stock price changes with certain level of accuracy.

## 2. Project Goal

The primary goal of this project is to determine whether technical indicators alone are effective predictors of short-term (1-day and 5-day) stock returns using supervised machine learning. Specifically, the project aims to build and evaluate models that forecast the S&P 500 (`SPY`)'s daily and weekly returns based on indicators like the Relative Strength Index (`RSI`), Simple Moving Average (`SMA`), and rolling volatility. By comparing model performance and analyzing prediction accuracy, the project will assess the effectiveness of these technical indicators for building predictive models.

## 3. Project Data

This project will use 30 years of `SPY` historical data, which is crucial for training reliable models for predicting overall U.S. stock market performance. `SPY` is the ticker symbol for an Exchange Traded Fund (ETF) that aims to mirror the S&P 500 index, which tracks the performance of 500 large-cap U.S. companies and is widely considered a benchmark of overall U.S. stock market performance. This 30-year timeframe provides not only a large dataset but also captures a variety of market conditions, including economic expansions and recessions, as well as major market crashes like the dot-com bubble in 2000, the financial crisis in 2008, and the COVID-19 pandemic in 2021. This diverse dataset helps the model make more accurate predictions across different scenarios. On the other hand, a shorter timeframe might lead to overfitting and inaccurate predictions when conditions change.

The `SPY` data is downloaded from **Yahoo Finance** using the `yfinance` Python API and saved as a raw file `SPY_data_raw.csv` before cleaning (Yahoo Finance, n.d.). This raw file `SPY_data_raw.csv` serves as a backup and allows the data to be reproduced for different models without downloading from the Yahoo API again to prevent API download restriction and blocking.

In [1]:
```
import pandas as pd
import numpy as np

import yfinance as yf
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.model_selection import TimeSeriesSplit, cross_val_score
from sklearn.linear_model import LinearRegression, Ridge, Lasso, LogisticRegression
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.feature_selection import RFE


ticker = 'SPY'
start_date = '1995-05-01'
end_date = '2025-04-30'

"""
################################################################################
# This section was commented out as the data was downloaded from Yahoo and saved as
################################################################################
download_data = yf.download(ticker, start=start_date, end=end_date, progress=False)
# print(spy_data.head())
# Save the raw data as a file so don't have to download the data again from Yahoo
download_data.to_csv('SPY_data_raw.csv')
"""
# To ensure parsing is consistent, specify index_col=0, parse_dates=True.
spy_data = pd.read_csv('SPY_data_raw.csv', index_col=0, parse_dates=True, date_form
print(spy_data.columns)
print(spy_data.index)
print(spy_data.head())
```

```python
print(spy_data.info())
#print(spy_data.describe())
print("Check missing values:\n", spy_data.isna().sum())
```

```
Index(['Close', 'High', 'Low', 'Open', 'Volume'], dtype='object')
Index(['Ticker', 'Date', '1995-05-01', '1995-05-02', '1995-05-03',
       '1995-05-04', '1995-05-05', '1995-05-08', '1995-05-09', '1995-05-10',
       ...
       '2025-04-15', '2025-04-16', '2025-04-17', '2025-04-21', '2025-04-22',
       '2025-04-23', '2025-04-24', '2025-04-25', '2025-04-28', '2025-04-29'],
      dtype='object', name='Price', length=7552)
                         Close               High                Low  \
Price
Ticker                     SPY                SPY                SPY
Date                       NaN                NaN                NaN
1995-05-01  30.301225662231445  30.420847871040742   30.30122566223145
1995-05-02   30.36564826965332  30.411656827637643   30.26442944208781
1995-05-03  30.788917541503906   30.78891754150391  30.466857734584412

                          Open  Volume
Price
Ticker                     SPY     SPY
Date                       NaN     NaN
1995-05-01  30.356435912451122  518700
1995-05-02   30.32884142326586  228400
1995-05-03  30.466857734584412  724700
<class 'pandas.core.frame.DataFrame'>
Index: 7552 entries, Ticker to 2025-04-29
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Close   7551 non-null   object
 1   High    7551 non-null   object
 2   Low     7551 non-null   object
 3   Open    7551 non-null   object
 4   Volume  7551 non-null   object
dtypes: object(5)
memory usage: 354.0+ KB
None
Check missing values:
 Close     1
High      1
Low       1
Open      1
Volume    1
dtype: int64
```

# 4. Raw Data Description

- **Rows**: 7552
- **Columns**: 5 columns — `Close`, `High`, `Low`, `Open`, `Volume`
- **Data Type**: All numeric except Date
- **Index**: Labelled by strings such as "Ticker", "Date
- **Form**: Single-table format

# 5. Data Cleaning Before Cleaning

## 5.1 Prelimary Findings:

- The dataset appears to have misaligned header rows. Specifically, the first three rows are not data but were incorrectly parsed as part of the dataset, causing `Ticker` and `Date` to be mixed into column labels.
- As a result, null values are present in the third row, and columns have incorrect data types.
- The date, price, and volume columns are stored as `object` rather than their correct types ( `datetime` , `float` , `int` respectively).

## 5.2 Cleaning Steps

**5.2.1 Remove Incorrect Header Rows**:

- The first three rows were not actual data. These rows were dropped.
- Rows containing null values resulting from this misparsing were also removed.

**5.2.2 Convert to Numeric**:

- All price columns ( `Open` , `High` , `Low` , `Close` ) and `Volume` were converted to numeric using `pd.to_numeric()` .
- The `errors='coerce'` option was used to convert any non-numeric values to `NaN` , allowing for easier detection and handling of issues.

**5.2.3 Fix Data Types**:

- Price columns were converted to `float` .
- Volume was cast to `int` .
- The `Date` column was converted to `datetime` using `pd.to_datetime()` .

In [2]:

```
# 5.2.1 Drop non-numeric index rows 'Ticker' and 'Date'
spy_data = spy_data.iloc[2:]

# 5.2.2 Convert all price and volumn columns to numeric
spy_data = spy_data.apply(pd.to_numeric, errors='coerce')

# 5.2.3 Check all the columns data type, and the index again
print(spy_data.columns)
print(spy_data.index)
print(spy_data.head())
print(spy_data.info())
print(spy_data.describe())
print("Check missing values:\n", spy_data.isna().sum())
```

```
Index(['Close', 'High', 'Low', 'Open', 'Volume'], dtype='object')
Index(['1995-05-01', '1995-05-02', '1995-05-03', '1995-05-04', '1995-05-05',
       '1995-05-08', '1995-05-09', '1995-05-10', '1995-05-11', '1995-05-12',
       ...
       '2025-04-15', '2025-04-16', '2025-04-17', '2025-04-21', '2025-04-22',
       '2025-04-23', '2025-04-24', '2025-04-25', '2025-04-28', '2025-04-29'],
      dtype='object', name='Price', length=7550)
                Close       High        Low       Open  Volume
Price
1995-05-01  30.301226  30.420848  30.301226  30.356436  518700
1995-05-02  30.365648  30.411657  30.264429  30.328841  228400
1995-05-03  30.788918  30.788918  30.466858  30.466858  724700
1995-05-04  30.770533  31.028181  30.696919  30.825743  311400
1995-05-05  30.733740  30.899371  30.669328  30.899371  314900
<class 'pandas.core.frame.DataFrame'>
Index: 7550 entries, 1995-05-01 to 2025-04-29
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Close   7550 non-null   float64
 1   High    7550 non-null   float64
 2   Low     7550 non-null   float64
 3   Open    7550 non-null   float64
 4   Volume  7550 non-null   int64
dtypes: float64(4), int64(1)
memory usage: 353.9+ KB
None
             Close         High          Low         Open        Volume
count  7550.000000  7550.000000  7550.000000  7550.000000  7.550000e+03
mean    166.179609   167.142211   165.093478   166.165986  8.974138e+07
std     135.106507   135.789142   134.303457   135.080516  9.106100e+07
min      30.301226    30.411657    30.264429    30.328841  9.500000e+03
25%      76.321274    76.726579    75.764894    76.324964  2.986325e+07
50%      99.673542   100.203836    99.084458    99.677424  6.731610e+07
75%     225.237198   226.319019   223.334802   225.166298  1.191772e+08
max     611.091675   611.390763   607.731787   609.705872  8.710263e+08
Check missing values:
 Close     0
High      0
Low       0
Open      0
Volume    0
dtype: int64
```

**5.2.4 Set and Sort Index**:

- The index column was originally misnamed as `'Price'` and stored as string. It was renamed to `'Date'` and converted to datetime.
- The data was sorted by date to prepare it for time-series modeling and ensure chronological order.

In [3]:
```python
# 5.2.4 Convert the index to datetime
spy_data.index = pd.to_datetime(spy_data.index)

# Rename the index to 'Date' for clarity
```

```
spy_data.index.name = 'Date'

# Sort index to ensure date is order
spy_data = spy_data.sort_index()

# Check that Date is index now
print(spy_data.columns)
print(spy_data.index)
print(spy_data.head())
print(spy_data.info())
print(spy_data.describe())
print("Check missing values:\n", spy_data.isna().sum())
```

```
Index(['Close', 'High', 'Low', 'Open', 'Volume'], dtype='object')
DatetimeIndex(['1995-05-01', '1995-05-02', '1995-05-03', '1995-05-04',
               '1995-05-05', '1995-05-08', '1995-05-09', '1995-05-10',
               '1995-05-11', '1995-05-12',
               ...
               '2025-04-15', '2025-04-16', '2025-04-17', '2025-04-21',
               '2025-04-22', '2025-04-23', '2025-04-24', '2025-04-25',
               '2025-04-28', '2025-04-29'],
              dtype='datetime64[ns]', name='Date', length=7550, freq=None)
                Close       High        Low       Open  Volume
Date
1995-05-01  30.301226  30.420848  30.301226  30.356436  518700
1995-05-02  30.365648  30.411657  30.264429  30.328841  228400
1995-05-03  30.788918  30.788918  30.466858  30.466858  724700
1995-05-04  30.770533  31.028181  30.696919  30.825743  311400
1995-05-05  30.733740  30.899371  30.669328  30.899371  314900
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 7550 entries, 1995-05-01 to 2025-04-29
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Close   7550 non-null   float64
 1   High    7550 non-null   float64
 2   Low     7550 non-null   float64
 3   Open    7550 non-null   float64
 4   Volume  7550 non-null   int64
dtypes: float64(4), int64(1)
memory usage: 353.9 KB
None
             Close         High          Low         Open        Volume
count  7550.000000  7550.000000  7550.000000  7550.000000  7.550000e+03
mean    166.179609   167.142211   165.093478   166.165986  8.974138e+07
std     135.106507   135.789142   134.303457   135.080516  9.106100e+07
min      30.301226    30.411657    30.264429    30.328841  9.500000e+03
25%      76.321274    76.726579    75.764894    76.324964  2.986325e+07
50%      99.673542   100.203836    99.084458    99.677424  6.731610e+07
75%     225.237198   226.319019   223.334802   225.166298  1.191772e+08
max     611.091675   611.390763   607.731787   609.705872  8.710263e+08
Check missing values:
 Close     0
High      0
Low       0
Open      0
Volume    0
dtype: int64
```

**5.2.5 Check and Handle Missing Values**:

- Missing values in price and volume likely reflect non-trading days (e.g. weekends or holidays).
- Although no missing rows were found after cleanup, forward-fill ( `ffill()` ) was applied to ensure continuity.
    - Since S&P 500 (SPY) index has grown from $\sim 30 in 1995 to$ 500+ in 2025, using a mean imputation would introduce unrealistic values.

- Whereas forward-fill ensure continuity by using the most recent valid value.

In [4]:
```
# 5.2.5 Check and Handle Missing Values
print("Check missing values before Forward fill:\n", spy_data.isna().sum())

# Forward fill for consistency
spy_data = spy_data.ffill()

print("Check missing values after Forward Fill:\n", spy_data.isna().sum())
```

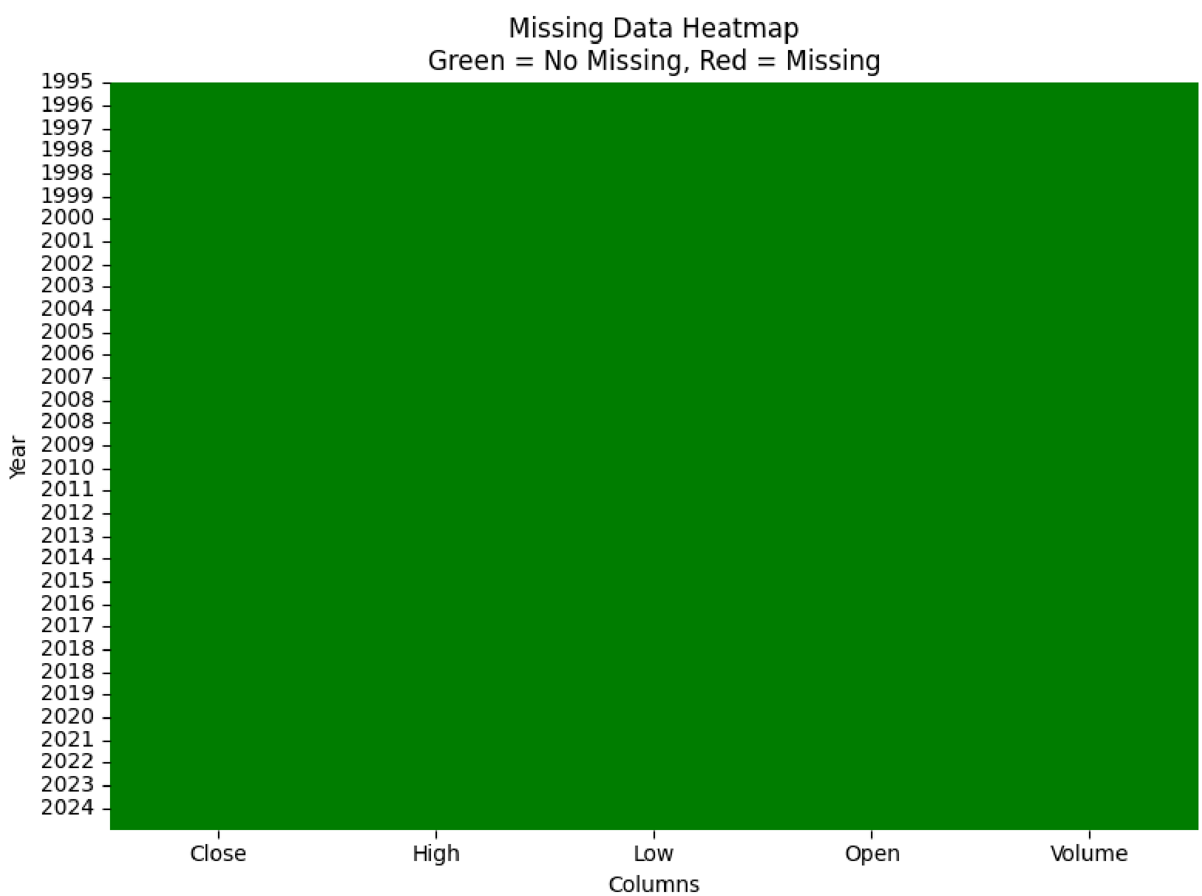
```
Check missing values before Forward fill:
 Close     0
High      0
Low       0
Open      0
Volume    0
dtype: int64
Check missing values after Forward Fill:
 Close     0
High      0
Low       0
Open      0
Volume    0
dtype: int64
```

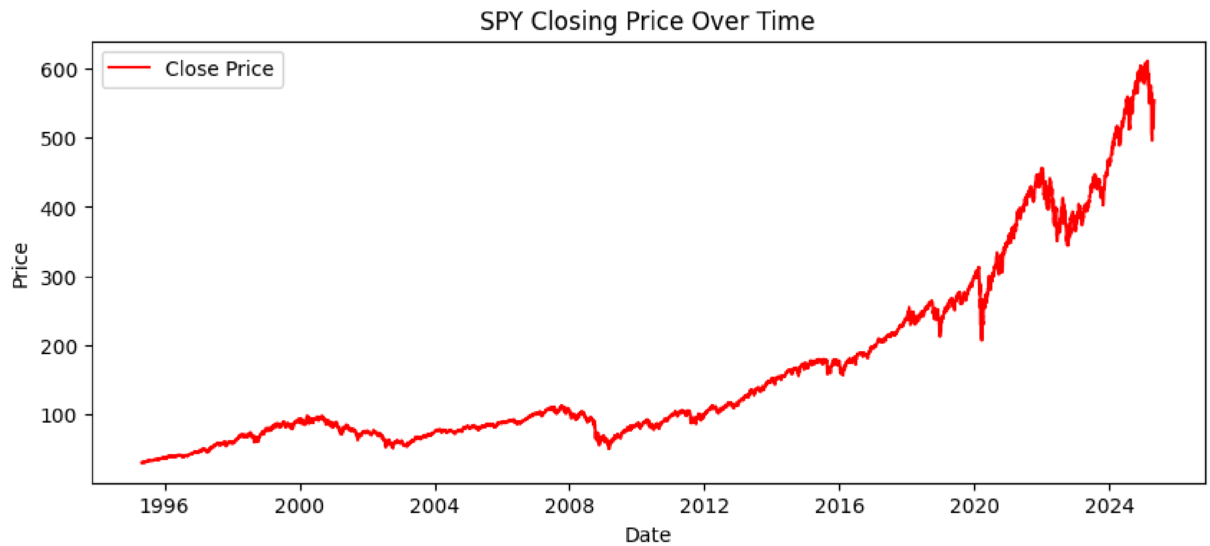
**5.2.6 Visual Diagnostics**:

In [5]:
```
# 5.2.6 Sanity Check After Cleaning
print(spy_data.info())
print("Check missing values:\n", spy_data.isna().sum())

##############################
# Plot missing values as a heatmap
# Create custom colormap: green for no missing (False), Red for missing (True)
cmap = ListedColormap(['green', 'red'])

plt.figure(figsize=(8, 6))
ax = sns.heatmap(spy_data.isna(), cmap=cmap, cbar=False)

# Customize y-axis to show only years
tick_locs = ax.get_yticks()
tick_labels = spy_data.index[tick_locs.astype(int)].year
ax.set_yticklabels(tick_labels)
tick_labels

plt.title('Missing Data Heatmap\nGreen = No Missing, Red = Missing', fontsize=12)
plt.xlabel('Columns')
plt.ylabel('Year')
plt.tight_layout()
plt.show()

##############################

# Line plot of SPY Price over Time
plt.figure(figsize=(10, 4))
```

```python
plt.plot(spy_data['Close'], label='Close Price', color='red')
plt.title("SPY Closing Price Over Time")
plt.xlabel("Date")
plt.ylabel("Price")
plt.legend()
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 7550 entries, 1995-05-01 to 2025-04-29
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Close   7550 non-null   float64
 1   High    7550 non-null   float64
 2   Low     7550 non-null   float64
 3   Open    7550 non-null   float64
 4   Volume  7550 non-null   int64
dtypes: float64(4), int64(1)
memory usage: 353.9 KB
None
Check missing values:
 Close     0
High      0
Low       0
Open      0
Volume    0
dtype: int64
```






Visually confirmed the data description and integrity:

- Tabulated summary shows correct row/column counts and expected data types.
  - **Rows**: 7,550
  - **Columns**: 5 — `Close`, `High`, `Low`, `Open`, `Volume`
  - **Data Types**: All numeric ( `float` / `int` ), `Date` as `datetime`
  - **Index**: Date
  - **Format**: Single-table format
- A `sns.heatmap()` of `.isna()` highlights missing values:
  - **Red** → Missing data ( `True` )
  - **Green** → No missing data ( `False` )
- Time-series plots of SPY price trends over the period closely resemble charts from Yahoo Finance (https://finance.yahoo.com/quote/SPY/).

**5.2.7 Export Cleaned Data**:

- The cleaned dataset was exported to `CSV` as `SPY_data_cleaned.csv` for use in further analysis and modeling.

In [6]:
```
# 5.2.7 Export Cleaned Data
spy_data.to_csv('SPY_data_cleaned.csv', index=True)
```

In [7]:
```
# open the cleaned dataset
spy_data = pd.read_csv('SPY_data_cleaned.csv', index_col=0, parse_dates=True)
print(spy_data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 7550 entries, 1995-05-01 to 2025-04-29
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Close   7550 non-null   float64
 1   High    7550 non-null   float64
 2   Low     7550 non-null   float64
 3   Open    7550 non-null   float64
 4   Volume  7550 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 353.9 KB
None
```

### 5.3 Data Cleaning Summary

- No null values remain. The index is correctly labeled as `Date`, set to datetime type, and sorted in chronological order—ensuring the dataset is ready for time-series modeling.
- A simple but effective data cleaning strategy includes using functions like `.info()` and `.isna().sum()` to perform quick sanity checks for data type consistency and missing values.

## 6. Predictor Features

The features listed below are commonly used in technical analysis and serve as a strong starting point for forecasting stock prices over time. These indicators are derived from historical price data and offer insights into market trends, momentum, and volatility. Further explanation on why these derived features are used instead of raw SPY prices will be provided in EDA Section 7.1.


## 6.1 Target Variables

A prediction target is required to train supervised ML models. For this project, the goal is to forecast S&P 500 (SPY) returns over 1-day and 5-day horizons. The target variables are:

- `Return_Forward_1D` : the return from today's close to tomorrow's close.
- `Return_Forward_5D` : the return from today's close to 5 trading days ahead.

Example: during training for 1-day price return, the model learns from technical indicators at time `t` and their corresponding `Return_Forward_1D` values (returns from `t` to `t+1` ).

When tested, the model receives only the input features at time `t` , and its predicted answer is then compared to the actual target ( `Return_Forward_1D` ).

**Formula:**

$$\text{Return}_{Forward1D} = \frac{\text{Close}_{t+1} - \text{Close}_t}{\text{Close}_t}$$

In [8]:
```
spy_data['Return_Forward_1D'] = spy_data['Close'].pct_change().shift(-1)
spy_data['Return_Forward_5D'] = spy_data['Close'].pct_change(periods=5).shift(-5)
```

## 6.2 Simple Moving Average (SMA)

**Definition:** SMA is the average closing price over a specified period which helps smooth out price fluctuations and identify trends.

**Common SMAs:**

- `SMA15` : 15 days SMA for short-term analysis.
- `SMA200` : 200 days SMA for long-term trend. When SMA15 < SMA200, it's considered downward trend.

**Formula:**

$$\text{SMA}_{n,t} = \frac{1}{n} \sum_{i=0}^{n-1} \text{Close}_{t-i}$$

In [9]:
```
spy_data['SMA15'] = spy_data['Close'].rolling(window=15).mean()
spy_data['SMA200'] = spy_data['Close'].rolling(window=200).mean()
```

## 6.3 Relative Strength Index (RSI)

**Definition:** RSI developed by Welles Wilder in 1978 to measures the magnitude of recent price changes to evaluate overbought or oversold conditions.

**Formula (14-day RSI):**

$$RS = \frac{\text{Avg Gain}}{\text{Avg Loss}}$$

$$\text{RSI}_{14} = 100 - \left(\frac{100}{1+RS}\right)$$

In [10]:
```
########## Calculate RSI ###############
# Calculate the daily closing price difference
delta = spy_data['Close'].diff()

# Group gains and losses:
# use clip to filter off -ve delta for the +ve gain and vice versa
# gain = Positive delta only
# loss = Negative delta only
gain = delta.clip(lower=0)
loss = -delta.clip(upper=0)

# Calculate the average gain and average loss over 14 days
avg_gain = gain.rolling(window=14).mean()
avg_loss = loss.rolling(window=14).mean()

# Calculate the Relative Strength (RS)
rs = avg_gain / avg_loss

# Calculate the Relative Strength Index (RSI)
spy_data['RSI_14'] = 100 - (100 / (1 + rs))
```

## 6.4 Volatility

**Definition:** Volatility is the rolling Std Dev of Returns. The degree of variation in returns interpreted as risk.

**Formula (rolling 10-day):** $\text{Volatility}_t = \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} (r_{t-i} - \bar{r})^2}$

In [11]:
```
spy_data['Return_1D'] = spy_data['Close'].pct_change()
spy_data['Volatility'] = spy_data['Return_1D'].rolling(window=10).std()
```

## 6.5 Dropped Missing Value

Missing values (NaNs) were expected after creating features such as returns, SMA, RSI, and volatility. These NaNs arise due to the rolling window calculations required by these indicators.

To ensure clean inputs for analysis and modeling, rows containing any missing values were dropped before proceeding to EDA and modeling stages.

In [12]:
```
# Check missing values after creating the features
print("\n\nCheck missing values after creating features:\n", spy_data.isna().sum())

# Drop rows with NA after computing the features
spy_data.dropna(inplace=True)

# Sanity Check missing values again after dropping NA
print("\n\nCheck missing values after dorpping:\n", spy_data.isna().sum())
```

```


Check missing values after creating features:
 Close                  0
High                  0
Low                   0
Open                  0
Volume                0
Return_Forward_1D     1
Return_Forward_5D     5
SMA15                14
SMA200              199
RSI_14               14
Return_1D             1
Volatility           10
dtype: int64


Check missing values after dorpping:
 Close                0
High                0
Low                 0
Open                0
Volume              0
Return_Forward_1D   0
Return_Forward_5D   0
SMA15               0
SMA200              0
RSI_14              0
Return_1D           0
Volatility          0
dtype: int64
```

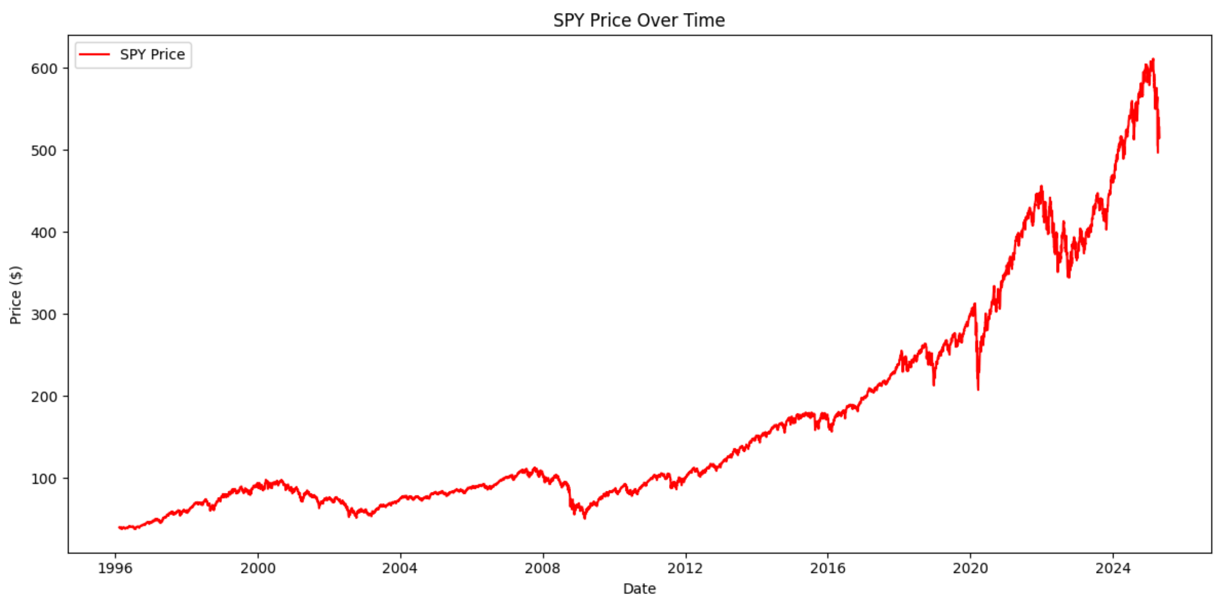
# 7. Exploratory Data Analysis (EDA)

This section explores the dataset structure, highlights key trends and potential outliers, and examines relationships between features. The goal is to ensure that inputs for machine learning models are meaningful, relevant, and free from major data quality issues.

## 7.1 Price Trend Overview

In [13]:

```
# Plot SPY Price Trend Over Time
plt.figure(figsize=(12, 6))
plt.plot(spy_data['Close'], label='SPY Price', color='Red')
plt.title('SPY Price Over Time')
plt.xlabel('Date')
plt.ylabel('Price ($)')
plt.legend()
plt.tight_layout()
plt.show()
```

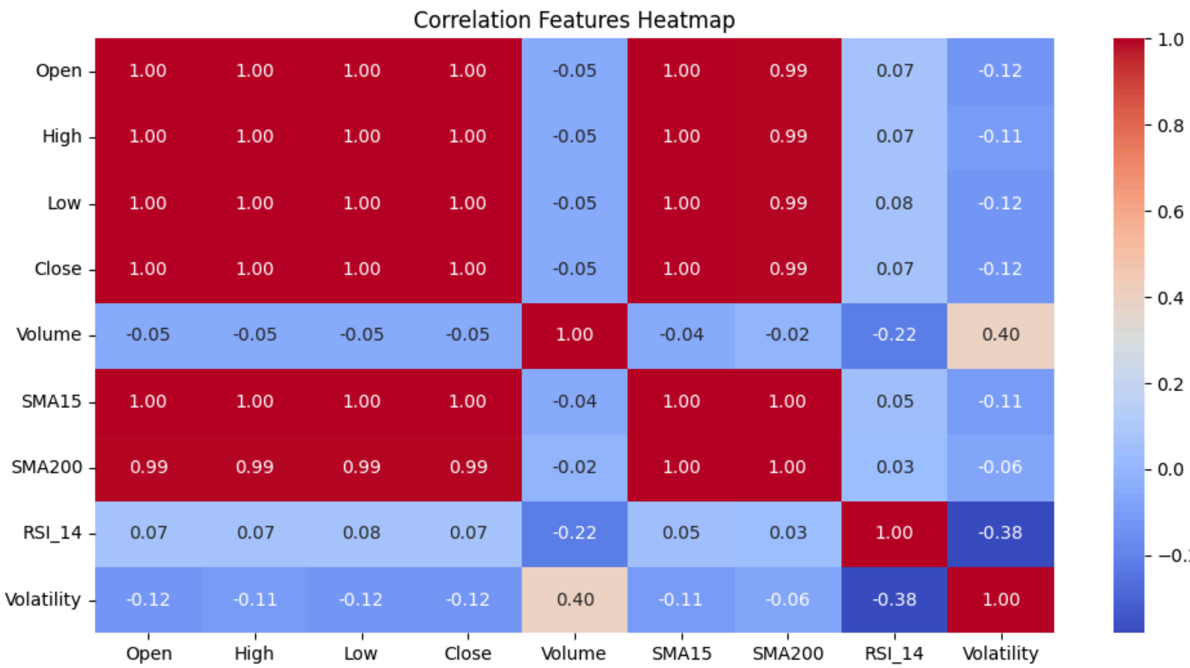



The plot shows that SPY prices exhibit a strong long-term upward trend, especially after 2008. This ever-increasing behavior suggests that raw daily prices are not ideal features for machine learning, as most ML models assume stationarity or rely on meaningful variation. The persistent upward bias can lead to poor predictive performance and overfitting. A better approach is to use derived metrics such as `Daily Return`, `SMA`, or `RSI` technical indicators for predictive modeling.

## 7.2 Correlation Features Heatmap

A feature correlation matrix heatmap was created to understand the relationships between the selected features. This helps identify redundancy and highly correlated features, which can impact linear model predictions.

In [14]:
```
#Plot Heatmap Feature Correlation Matrix
plt.figure(figsize=(12, 6))
selected_features = ['Open', 'High', 'Low', 'Close', 'Volume', 'SMA15', 'SMA200', '
sns.heatmap(spy_data[selected_features].corr(), annot=True, fmt=".2f", cmap='coolwa
plt.title("Correlation Features Heatmap")
plt.show()
```



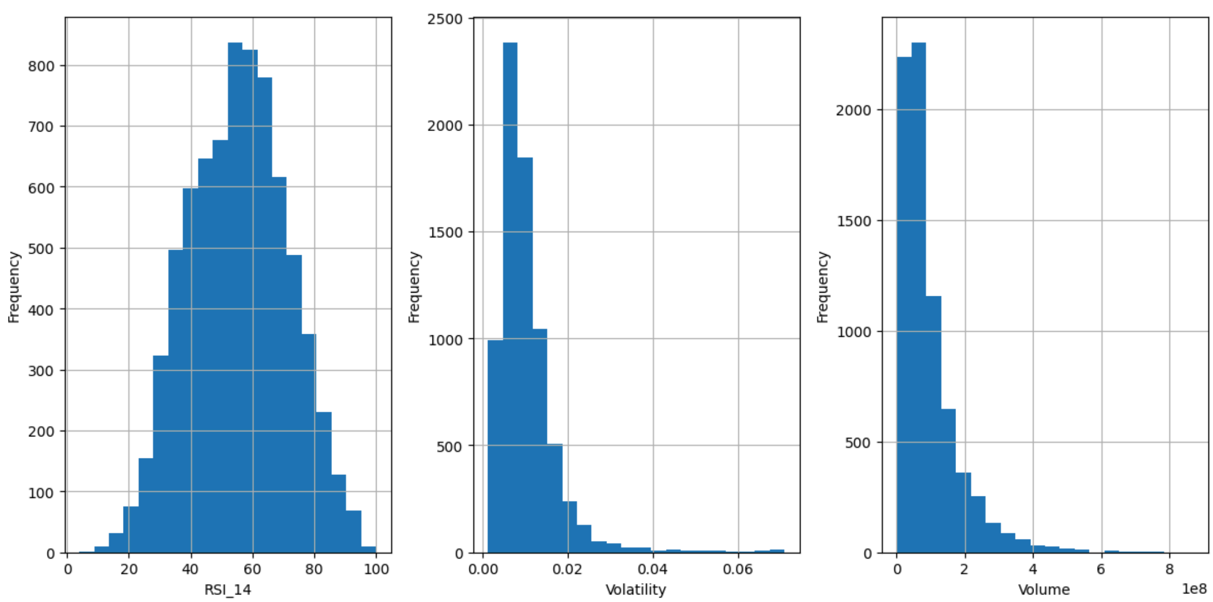

**Findings:**

- Features like `Open`, `High`, `Low`, `Close`, `SMA15`, and `SMA200` are almost perfectly correlated (correlation ≈ 1.00), indicating strong redundancy. These features could lead to multicollinearity and are not meaningful for modeling.
- Indicators such as `RSI_14` and `Volatility` show much weaker correlation with price-based features which suggests they provide more independent signals and could improve model generalization.
- `RSI_14` and `Volatility` show much weaker correlation with price-based features (like Open, Close, etc.)which suggests `RSI_14` and `Volatility` don't move in the same way as the basic price features. This means `RSI_14` and `Volatility` are not just repeating the same information which can help ML model see new patterns that price data might miss. This can make the model more accurate when making predictions on new data.
- Simliarly, `Volume` has slightly negative correlation with most price-related features. This implies `Volume` may also contribute unique information to the model and should be considered in feature selection.

## 7.3 Feature Distributions

This section examines the distribution of selected features to identify skewness and potential data transformation needs.

In [15]:
```
# Plot distribution of the selected feature
features = ['RSI_14', 'Volatility', 'Volume']
plt.figure(figsize=(12, 6))

for i, feature in enumerate(features):
    plt.subplot(1, len(features), i + 1)
    plt.hist(spy_data[feature], bins=20)
    plt.xlabel(feature)
    plt.ylabel('Frequency')
    plt.grid(True)

# prevent chart overlapping
plt.tight_layout()

plt.show()
```




**Findings:**

- `RSI_14` appears approximately bell-shaped and symmetrical.
  - The RSI values generally fall within a normal distribution range, fluctuating around the mid-zone.
  - Only a few instances occur at the extreme ends, which correspond to oversold or overbought conditions in the market.
- `Volatility` has a noticeable right skew with fat tails as expected low volatility most days and occasional high spikes.
  - This suggests that SPY tends to be stable most of the time.
  - The long tail on the right highlights rare but significant spikes in volatility, often tied to market stress.

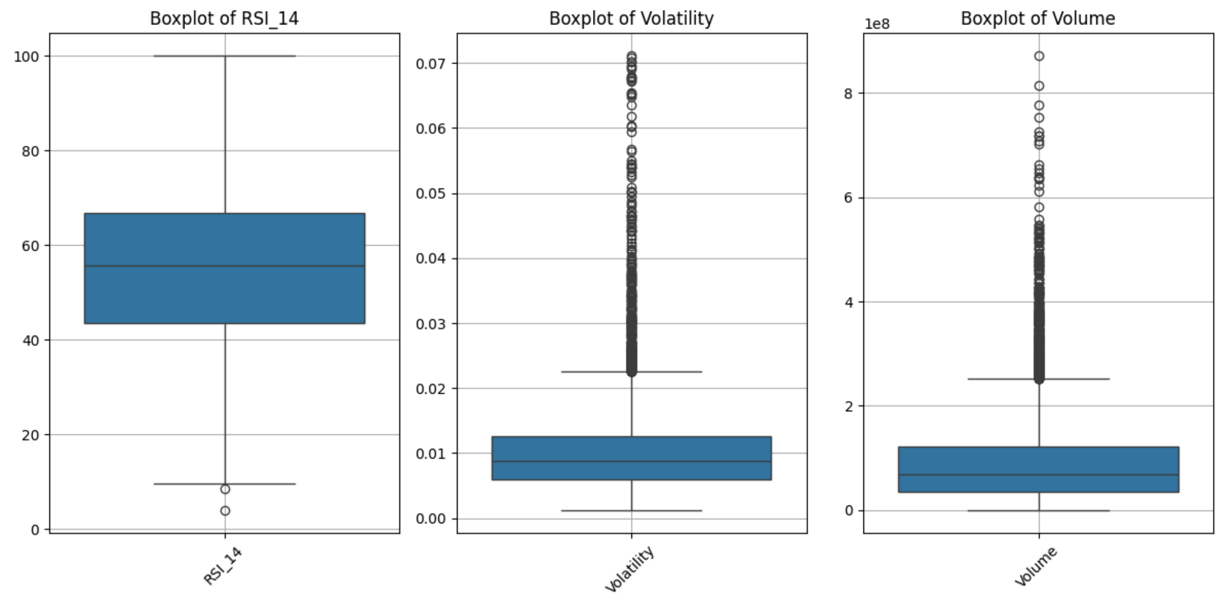
    - The distribution is already close to normal for most values so log transformation will have minimum effrct on Volatility.
- `Volume` is heavily right-skewed with fat tails.
    - Most trading days show relatively low trading volume.
    - However, there are occasional days with exceptionally high volume, which which will required a Log Transformation so not to distort model.

## 7.4 Boxplots for Outlier Detection

This section uses boxplots to identify outliers in selected features that could potentially impact modeling performance.

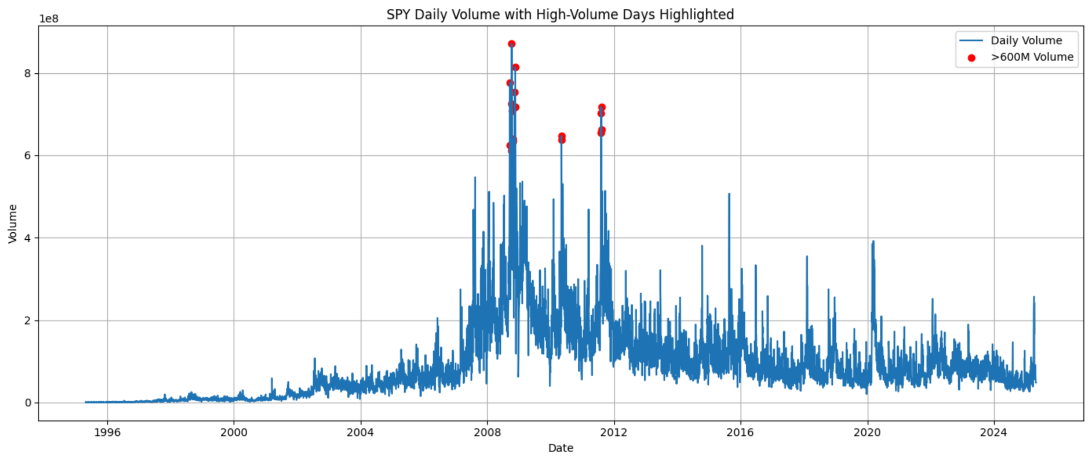
In [16]:

```
# Create Boxplot with subplots in 1 row and 3 columns
fig, axes = plt.subplots(1, 3, figsize=(12, 6))

# Boxplot for RSI_14
sns.boxplot(data=spy_data[['RSI_14']], ax=axes[0])
axes[0].set_title("Boxplot of RSI_14")
axes[0].tick_params(axis='x', rotation=45)
axes[0].grid(True)

# Boxplot for Volatility
sns.boxplot(data=spy_data[['Volatility']], ax=axes[1])
axes[1].set_title("Boxplot of Volatility")
axes[1].tick_params(axis='x', rotation=45)
axes[1].grid(True)

# Boxplot for Volume
sns.boxplot(data=spy_data[['Volume']], ax=axes[2])
axes[2].set_title("Boxplot of Volume")
axes[2].tick_params(axis='x', rotation=45)
axes[2].grid(True)

# prevent chart overlapping
plt.tight_layout()

plt.show()
```




**Findings**:

- `RSI_14` shows a relatively narrow range with a few lower-end outliers.
    - The box spans approximately from 45 to 70, with a median around 55.
    - Whiskers extend roughly from 30 to 100, capturing most typical values.
    - A few outliers fall below 30, indicating rare oversold conditions.
- `Volatility` demonstrates that SPY price is usually stable but does experience rare high-volatility events.
    - The box is tightly compressed between 0.005 and 0.015, with a median near 0.01.
    - Many outliers appear beyond the upper whisker at around 0.02, suggesting occasional sudden volatility spikes caused by turbulent market periods.
    - These outliers are not noise but important signal to insight about market risk under stress.
- `Volume` exhibiting low trading volume but with a few has extreme extremely high volume selling or buying of SPY
    - The box lies between approximately 0.5×e8 and 1.5×e8, with the median around 1×e8.
    - A significant number of outliers exceed 6×e8, may correspond to exceptional market events such as the 2008 financial crisis, characterized by panic-driven buying or selling.




## 7.5 EDA Summary

- The SPY price trend shows long-term growth so it would be appropreiate to use derived indicators such as `RSI` instead of raw prices in order for ML to make accurate prediction.
- Correlation analysis highlighted redundancy among prices related features and justified `RSI`, `Volatility` and `Volume` feature selection before modeling.
- `RSI`, `Volatility` and `Volume` will require treatment during preprocessing model.

# 8. Modeling

This section builds simple models to predict SPY's 1-day future returns using technical indicators and price-based features. The target is the daily return, calculated as the percentage change from today's close to the next day's close. Several machine learning methods — including MLR, Ridge, Lasso, and Random Forest — are used to evaluate and compare predictive performance.

## 8.1 Feature Preprocessing

Before modeling, the selected features have to be transformed to improve learning performance and mitigate skew and scale issues. The features `RSI_14`, `Volatility`, and `Volume` were retained based on prior multicollinearity checks.

### 8.1.1 Log Transformation

`Volume` showed heavy right skew and extreme outliers. A logarithmic transformation was applied using `log1p` to compress the long tail while retaining scale integrity and improves distribution symmetry:

$$\text{Volume}_{\text{log}} = \log(\text{Volume} + 1)$$

In [17]:
```
# For sanity check, print out the volumne value before and after Log Transform
print("Before Log Transform:\n ", spy_data['Volume'].describe())

spy_data['Volume_log'] = np.log1p(spy_data['Volume'])

print("\n\nAfter Log Transform:\n ", spy_data['Volume_log'].describe())
```

```
Before Log Transform:
  count    7.346000e+03
mean     9.218074e+07
std      9.110129e+07
min      1.844000e+05
25%      3.460730e+07
50%      6.918480e+07
75%      1.214108e+08
max      8.710263e+08
Name: Volume, dtype: float64


After Log Transform:
  count    7346.000000
mean       17.711817
std         1.411765
min        12.124868
25%        17.359575
50%        18.052292
75%        18.614690
max        20.585183
Name: Volume_log, dtype: float64
```

### 8.1.2 Feature Standardization

Although `RSI_14` and `Volatility` showed only mild skew, all features were standardized to ensure they contributed equally to model learning.

The formula for standardization is:

$$X_{\text{scaled}} = \frac{X-\mu}{\sigma}$$

Where:

- μ is the mean of the feature
- σ is the standard deviation of the feature

This process scales the data to have a mean of 0 and a standard deviation of 1. The `StandardScaler()` library will be used for this.

In [18]:
```
# Feature Standardization
features_to_scale = ['RSI_14', 'Volatility', 'Volume_log']

# Provides descriptive statistics of the data before and after standardization
# to compare the transformation and verify its effect.
print("Before Standarization:\n ", spy_data[features_to_scale].describe())
```

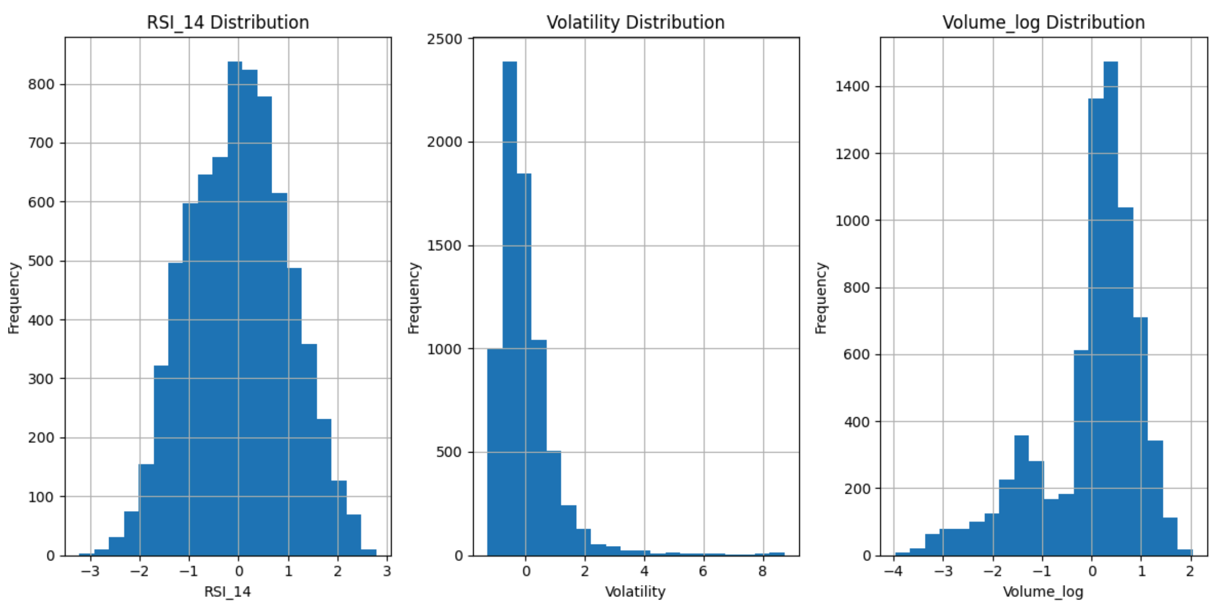
```
# Initialize the scaler
scaler = StandardScaler()

# Make a copy of the dataset to be transformed
spy_data_scaled = spy_data.copy()

# Transform selected features
spy_data_scaled[features_to_scale] = scaler.fit_transform(spy_data_scaled[features_

# Sanity check on result
print("\n\nAfter Standarization:\n ", spy_data_scaled[features_to_scale].describe()
print("\n\nCheck missing values for spy_data_scaled:\n", spy_data_scaled.isna().sum
```

```
Before Standarization:
              RSI_14   Volatility   Volume_log
count  7346.000000  7346.000000  7346.000000
mean     55.492419     0.010282    17.711817
std      15.997105     0.006963     1.411765
min       3.989674     0.001265    12.124868
25%      43.552353     0.005895    17.359575
50%      55.802970     0.008745    18.052292
75%      66.831720     0.012546    18.614690
max     100.000000     0.071055    20.585183


After Standarization:
               RSI_14    Volatility    Volume_log
count  7.346000e+03  7.346000e+03  7.346000e+03
mean   6.190408e-17 -6.190408e-17 -1.826170e-15
std    1.000068e+00  1.000068e+00  1.000068e+00
min   -3.219723e+00 -1.295201e+00 -3.957690e+00
25%   -7.464400e-01 -6.301376e-01 -2.495214e-01
50%    1.941421e-02 -2.208699e-01  2.411860e-01
75%    7.088828e-01  3.251492e-01  6.395785e-01
max    2.782417e+00  8.728431e+00  2.035438e+00


Check missing values for spy_data_scaled:
 Close                 0
High                 0
Low                  0
Open                 0
Volume               0
Return_Forward_1D    0
Return_Forward_5D    0
SMA15                0
SMA200               0
RSI_14               0
Return_1D            0
Volatility           0
Volume_log           0
dtype: int64
```

Visualization of the distribution of the features and boxchart after transformation and standardization:

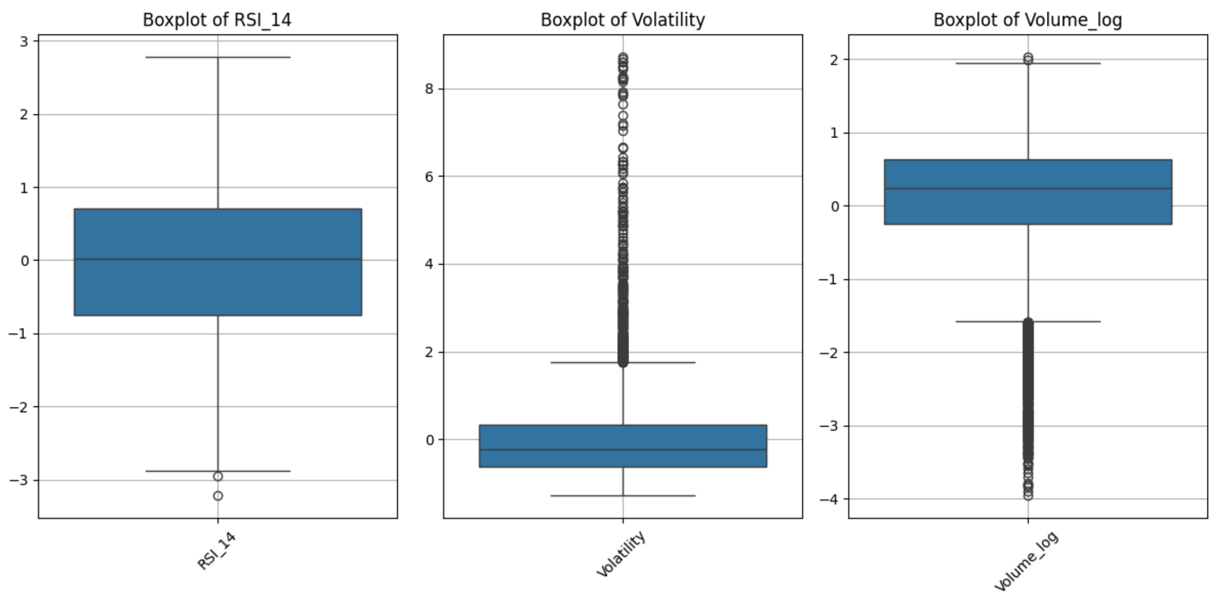
- Log-transfromed has effectively reduced Volume skewness and made the distribution more symmetrical.
- Log-transforming Volatility didn't help much because most of its values are already small and close to zero. The transformation didn't significantly fix the skew or make the data more normal, so preserve the original feature.
- From the boxchart, the features data is centered around 0 after standardization.

In [19]:
```
# Plot distribution of the selected feature
features_to_scale = ['RSI_14', 'Volatility', 'Volume_log']
plt.figure(figsize=(12, 6))

for i, feature in enumerate(features_to_scale):
    plt.subplot(1, len(features_to_scale), i + 1)
    plt.hist(spy_data_scaled[feature], bins=20)
    plt.xlabel(feature)
    plt.ylabel('Frequency')
    plt.title(f'{feature} Distribution')
    plt.grid(True)

# prevent chart overlapping
plt.tight_layout()

plt.show()
```




In [20]:
```
# Create Boxplot with subplots in 1 row and 3 columns
fig, axes = plt.subplots(1, 3, figsize=(12, 6))

# Boxplot for RSI_14
sns.boxplot(data=spy_data_scaled[['RSI_14']], ax=axes[0])
axes[0].set_title("Boxplot of RSI_14")
axes[0].tick_params(axis='x', rotation=45)
axes[0].grid(True)

# Boxplot for Volatility
sns.boxplot(data=spy_data_scaled[['Volatility']], ax=axes[1])
```

```python
axes[1].set_title("Boxplot of Volatility")
axes[1].tick_params(axis='x', rotation=45)
axes[1].grid(True)

# Boxplot for Volume
sns.boxplot(data=spy_data_scaled[['Volume_log']], ax=axes[2])
axes[2].set_title("Boxplot of Volume_log")
axes[2].tick_params(axis='x', rotation=45)
axes[2].grid(True)

# prevent chart overlapping
plt.tight_layout()

plt.show()
```

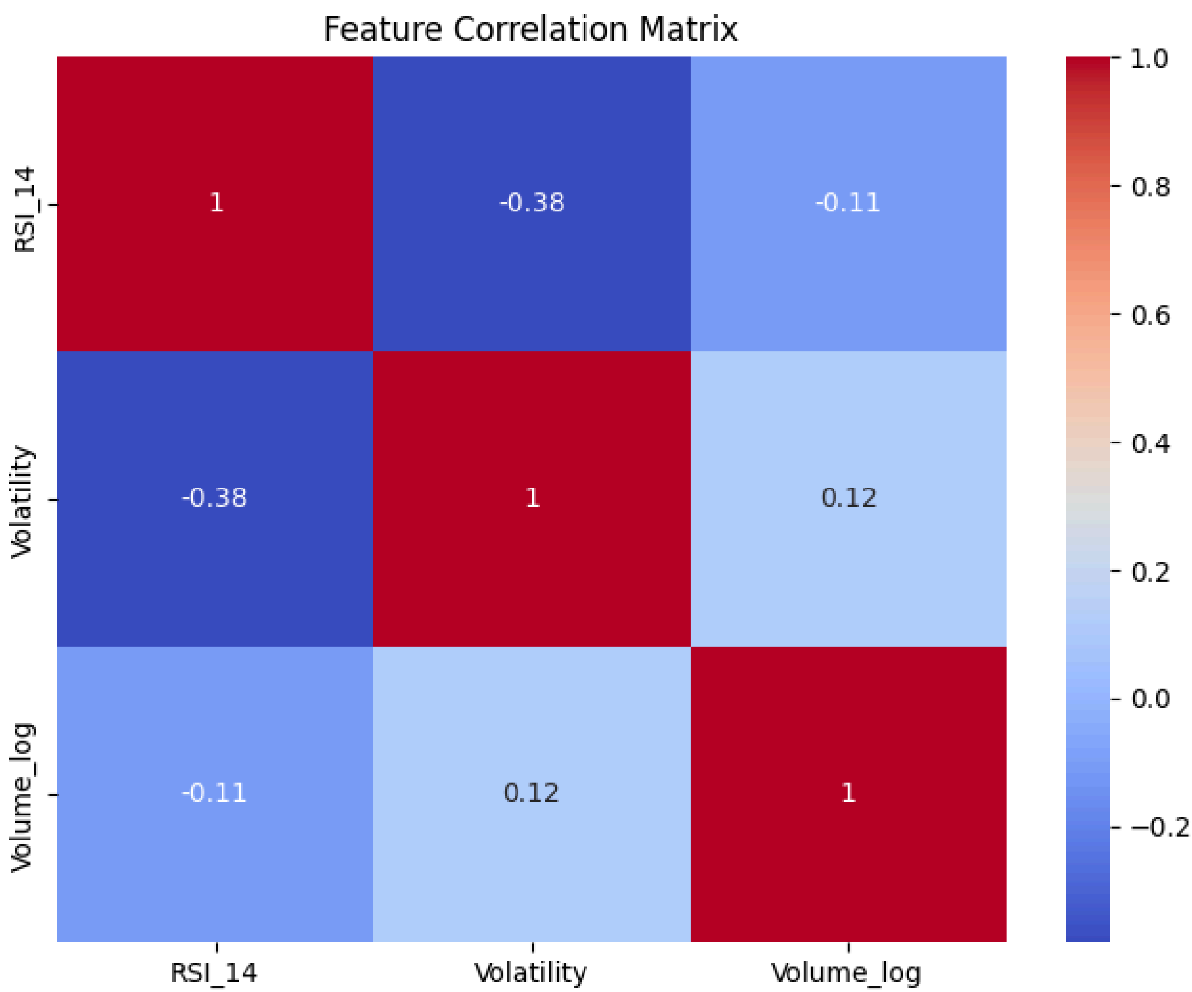



## 8.2 Multicollinearity Check

### 8.2.1 Correlation Matrix

A feature correlation matrix heatmap was created to understand the relationships between the selected features. This helps identify highly correlated features, which can impact linear model predictions.

In [21]:
```python
# Plot Heatmap Feature Correlation Matrix
X = spy_data_scaled[features_to_scale]
y = spy_data_scaled['Return_Forward_1D']
plt.figure(figsize=(8, 6))
sns.heatmap(X.corr(), annot=True, cmap='coolwarm')
plt.title('Feature Correlation Matrix')
plt.show()
```




### 8.2.2 Variance Inflation Factor (VIF)

Multicollinearity was assessed using the VIF.

In [22]:
```
# High VIF (>10) suggests multicollinearity = drop or regularize those features.
X = spy_data_scaled[features_to_scale]
y = spy_data_scaled['Return_Forward_1D']
vif_data = pd.DataFrame()
vif_data['feature'] = X.columns
vif_data['VIF'] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1]
print(vif_data)
```

```
      feature       VIF
0      RSI_14  1.174803
1  Volatility  1.177849
2  Volume_log  1.018413
```

Features with VIF < 10 confirmed the final selecttion:

| Feature | VIF |
|---|---|
| RSI_14 | 1.174374 |
| Volatility | 1.177444 |

| Feature | VIF |
|---|---|
| Volume_log | 1.018445 |

## 8.3 Recursive Feature Elimination (RFE)

While RFE is not primarily for multicollinearity check, it does helps understand which features are most important for prediction.

In [23]:
```
# Recursive Feature Elimination (RFE)
rfe = RFE(estimator=LinearRegression(), n_features_to_select=3)
rfe.fit(X, y)
print("Selected Features:", X.columns[rfe.support_])

# Selected Features: Index(['RSI_14', 'Volatility', 'Volume_log'], dtype='object')
```

```
Selected Features: Index(['RSI_14', 'Volatility', 'Volume_log'], dtype='object')
```

RFE confirmed that `RSI_14`, `Volatility`, and `Volume_log` are not redundant and should not be eliminated.

## 8.4 Multiple Linear Regression (MLR)

MLR is a natural first choice for modeling because it is easy to interpret, computationally efficient, and provides a baseline for comparison with more complex models. MLR assumes a linear relationship between the target variable (1-day past return) and the independent features `RSI_14`, `Volatility`, and `Volume_log`.

sklearn's `LinearRegression` is used. For cross-validation with this time series data, `TimeSeriesSplit` from sklearn is essential (Jones, 2025; KoshurAI, 2023, Scikit-learn, n.d.). It maintains chronological order by splitting the dataset into 5 sequential folds (`n_splits=5`). This ensures training data always precedes test data, with the model learning from past data (up to a point) and predicting the immediate future

> *Note: Sections 8.4 to 8.7 focus on model screening using R-Squared scores from cross-validation. These steps help identify candidate models for more detailed evaluation in the next phase (Section 9).*

In [24]:
```
# 8.4 Multiple Linear Regression (MLR)
model = LinearRegression()

# TimeSeriesSplit split the dataset into 5 sequential folds
# tscv variable will be used for MLR, Ridge & Lasso
tscv = TimeSeriesSplit(n_splits=5)

# Cross-validation with the TimeSeriesSplit provides the R-Squared score
r2_scores = cross_val_score(model, X, y, cv=tscv, scoring='r2')

# Print only 5 decimal places
```

```
np.set_printoptions(precision=5)
print("LinearRegression Cross-validated R-Squared Score: ", r2_scores, "\n")

# LinearRegression Cross-validated R-Squared Score:  [-0.00405 -0.00704  0.00128 -0
```

```
LinearRegression Cross-validated R-Squared Score:  [-0.00405 -0.00704  0.00128 -0.00
143  0.00269]
```

LinearRegression Cross-validated R-Squared Score: `[-0.00405 -0.00704  0.00128 -0.00143  0.00269]`

- The low and negative R-squared scores indicate underfitting and the model has failed to learn the underlying relationship in the data.
- R-squared values close to zero mean implied that MLR's predictions are not much better than simply using the mean of the target variable.
- The scores are consistent, but also consistently poor, so the model is stable but not useful.

In [25]:
```
# reduce to only one feature
features_reduced = ['RSI_14']
X_reduced = spy_data_scaled[features_reduced]
#y = spy_data_scaled['Return_Forward_1D']
r2_scores_reduced = cross_val_score(model, X_reduced, y, cv=tscv, scoring='r2')

print("LinearRegression Cross-validated R-Squared Score after reducing the features

#LinearRegression Cross-validated R-Squared Score after reducing the features:  [-0
```

```
LinearRegression Cross-validated R-Squared Score after reducing the features:  [-0.0
0723  0.00068  0.00191 -0.00123  0.00241]
```

LinearRegression Cross-validated R-Squared Score after reducing the features: `[-0.00723  0.00068  0.00191 -0.00123  0.00241]` Even after trying to reduce to one feature `RSI_14`, the LinearRegression Cross-validated R-Squared Score did not improve.

## 8.5 Ridge Regression

Ridge Regression is useful when overfitting is a concern. Ridge helps stabilize the model without removing features, which is good for noisy data like financial data.

In [26]:
```
 # Using default alpha for initial Ridge model.
model = Ridge(alpha=1.0)

# Cross-validation with the TimeSeriesSplit provides the R-Squared score
r2_scores = cross_val_score(model, X, y, cv=tscv, scoring='r2')

# Output R-Squared score
print("Ridge Cross-validated R-Squared Score: ", r2_scores, "\n")
# Ridge Cross-validated R-Squared Score:  [-0.00405 -0.00704  0.00128 -0.00143  0.0
```

```
Ridge Cross-validated R-Squared Score:  [-0.00405 -0.00704  0.00128 -0.00143  0.0026
9]
```

Ridge Cross-validated R-Squared Score: `[-0.00405 -0.00704  0.00128 -0.00143  0.00269]`

- The Ridge cross-validated R-squared scores are similar to MLR's, meaning that Ridge offers no real advantage over MLR.
- Ridge regression is suppose to address multicollinearity and reduce overfitting so this implied that multicollinearity and overfitting weren't major problems.
- It also demostrated that regularization had little effect.

## 8.6 Lasso Regression

Unlike Ridge, Lasso Regression can shrink some coefficients to zero. This is useful when not all features are good predictors, which often happens in financial modeling.

In [27]:
```python
# Lasso model with a small regularization strength
model = Lasso(alpha=0.01)

# Cross-validation with the TimeSeriesSplit provides the R-Squared score
r2_scores = cross_val_score(model, X, y, cv=tscv, scoring='r2')

# Output R-Squared score
print("Lasso Cross-validated R-Squared Score: ", r2_scores, "\n")

# Lasso Cross-validated R-Squared Score:  [-2.62410e-03 -2.35783e-04 -1.28475e-03 -
```

```
Lasso Cross-validated R-Squared Score:  [-2.62410e-03 -2.35783e-04 -1.28475e-03 -8.0
4740e-05 -1.43023e-04]
```

Lasso Cross-validated R-Squared Score: `[-2.62410e-03 -2.35783e-04 -1.28475e-03 -8.04740e-05 -1.43023e-04]`

- The Lasso cross-validated R-squared scores are all negative and still poor at predicting the average return, similar to Ridge.
- The features weren't very helpful in making predictions likely explains why Ridge and Lasso didn't help.

## 8.7 Random Forest Regressor

Random Forest is a powerful method that finds hidden patterns in data. It handles complex relationships and is good for noisy financial data because it resists overfitting and doesn't need features to be perfectly clear.

In [28]:
```python
# Random Forest Regressor
from sklearn.ensemble import RandomForestRegressor
```

```python
rf = RandomForestRegressor(n_estimators=100, random_state=5)
rf_scores = cross_val_score(rf, X, y, cv=tscv, scoring='r2')
print("Random Forest Cross-validated R-Squared Score:", rf_scores)

# Random Forest Cross-validated R-Squared Score: [-0.83506 -0.05636 -0.21322 -0.189
```

```
Random Forest Cross-validated R-Squared Score: [-0.83506 -0.05636 -0.21322 -0.18905
-0.0433 ]
```

Random Forest Cross-validated R-Squared Score: `[-0.83506 -0.05636 -0.21322 -0.18905 -0.0433 ]`

- The Random Forest cross-validated R-squared scores are mostly negative which is worse than the previous models.
- This implies that Random Forest model performs significantly worse in predicting the average return.
- Since a powerful model like Random Forest couldn't even find a useful pattern for forecasting in this data, it is unlikely a model issue but a features issue.

## 8.8 Logistic Regression

Given the poor performance of the linear regression models, a Logistic Regression model was also tested. This was to explore predicting stock direction (up/down) instead of return magnitude, aiming to improve prediction outcomes. The target variable was converted to 'True' if the 1-day forward return was positive ( `Return_Forward_1D > 0` )

However, the Logistic Regression model returned an average accuracy of 54.5%, which is only marginally better than chance. Given this limited performance, Logistic Regression was not selected for further evaluation in Section 9.

In [29]:
```python
### Logistic Regression model

# Convert Return_Forward_1D to a binary format.
# True if greater than 0 else False.
y_1D_binary = (spy_data['Return_Forward_1D'] > 0).astype(int)

model = LogisticRegression()
scores = cross_val_score(model, X, y_1D_binary, cv=tscv, scoring='accuracy')
print("Accuracy Score:", scores.mean())
```

```
Accuracy Score: 0.5446078431372549
```

## 8.9 Underfitting Discussion

- Across all models — including linear models (MLR, Ridge, Lasso), a tree-based model (Random Forest), and the classification-based Logistic Regression — the results indicate underfitting.

- Regression models produced low or even negative R-squared scores, suggesting they failed to learn any meaningful relationship between the features and target returns.
- Logistic Regression also showed poor classification performance, with cross-validated accuracy barely above chance (~54%), further supporting the conclusion that the features lack predictive power.
- This consistent underperformance across both simple and complex models suggests the issue lies not with model capacity, but with the **weak signal in the input features**.
- Overfitting is not a concern here, as none of the models demonstrated strong learning even on training folds — confirming the absence of a meaningful pattern to exploit.

# 9. Results and Analysis

Although the models exhibited low R-squared values in initial testing, this section proceeds to evaluate their performance in predicting SPY returns (future next-day return). This analysis aims to understand the specific limitations of the models in this context.

**Methodology:**

- Use MLR and Ridge Regression model for prediction
- Evaluate key metrics `R-Squared` (coefficient of determination), `RMSE` (Root Mean Squared Error), and `MAE` (Mean Absolute Error) on both model
- Compare both models results
- Visualize predicted vs. actual and residuals

## 9.1 Set Target Variables

- First is to set future returns as target variable (y) in model training.
- `Return_Forward_1D` is the return from today's close to tomorrow's close.
- `Return_Forward_5D` is used to predict returns 5 days ahead.

In [30]:

```
# Define VIF-cleaned features, y and targets
X = spy_data_scaled[features_to_scale]

# unscaled target
y_1D = spy_data['Return_Forward_1D']
y_5D = spy_data['Return_Forward_5D']
```

## 9.2 Train-Test Split for Time Series

Simliarly to `Section 8.4`, `TimeSeriesSplit()` uses the option `n_splits=5` split into 5 sequential folds, with the training data precedes test data.

In [31]:

```
# TimeSeriesSplit
tscv = TimeSeriesSplit(n_splits=5)

# Store metrics
```

```
results = {
    "Model": [],
    "Target": [],
    "R2": [],
    "RMSE": [],
    "MAE": []
}
```

## 9.3 Evaluate Models

In this section, Multiple Linear Regression (MLR) and Ridge Regression will be evaluated on both 1-day and 5-day forward return targets.

In order to be consistent while evaluating the models, each model is trained and tested using TimeSeriesSplit cross-validation as set previously.

In [32]:
```
# --- Evaluate MLR Model ---
model = LinearRegression()
# 1-Day predictions
MLR_R2_scores_1D = cross_val_score(model, X, y_1D, cv=tscv, scoring='r2')
MLR_RMSE_scores_1D = -cross_val_score(model, X, y_1D, cv=tscv, scoring='neg_root_me
MLR_MAE_scores_1D = -cross_val_score(model, X, y_1D, cv=tscv, scoring='neg_mean_abs

# 5-Day predictions
MLR_R2_scores_5D = cross_val_score(model, X, y_5D, cv=tscv, scoring='r2')
MLR_RMSE_scores_5D = -cross_val_score(model, X, y_5D, cv=tscv, scoring='neg_root_me
MLR_MAE_scores_5D = -cross_val_score(model, X, y_5D, cv=tscv, scoring='neg_mean_abs


# --- Evaluate Ridge Model ---
model = Ridge(alpha=1.0)
# 1-Day predictions
Ridge_R2_scores_1D = cross_val_score(model, X, y_1D, cv=tscv, scoring='r2')
Ridge_RMSE_scores_1D = -cross_val_score(model, X, y_1D, cv=tscv, scoring='neg_root_
Ridge_MAE_scores_1D = -cross_val_score(model, X, y_1D, cv=tscv, scoring='neg_mean_a

# 5-Day predictions
Ridge_R2_scores_5D = cross_val_score(model, X, y_5D, cv=tscv, scoring='r2')
Ridge_RMSE_scores_5D = -cross_val_score(model, X, y_5D, cv=tscv, scoring='neg_root_
Ridge_MAE_scores_5D = -cross_val_score(model, X, y_5D, cv=tscv, scoring='neg_mean_a

# Consolidate results as summary_data
summary_data = {
    "Model": ["MLR", "MLR", "Ridge", "Ridge"],
    "Target": ["1D", "5D", "1D", "5D"],
    "R2": [
        np.mean(MLR_R2_scores_1D),
        np.mean(MLR_R2_scores_5D),
        np.mean(Ridge_R2_scores_1D),
        np.mean(Ridge_R2_scores_5D)
    ],
    "RMSE": [
        np.mean(MLR_RMSE_scores_1D),
        np.mean(MLR_RMSE_scores_5D),
```

```
        np.mean(Ridge_RMSE_scores_1D),
        np.mean(Ridge_RMSE_scores_5D)
    ],
    "MAE": [
        np.mean(MLR_MAE_scores_1D),
        np.mean(MLR_MAE_scores_5D),
        np.mean(Ridge_MAE_scores_1D),
        np.mean(Ridge_MAE_scores_5D)
    ]
}


# Post-evaluation MLR prediction with MAE
# The results will be used in 9.4 results table
# and 9.5 visualization plots
model_1D = LinearRegression().fit(X, y_1D)
model_5D = LinearRegression().fit(X, y_5D)

pred_1D = model_1D.predict(X)
pred_5D = model_5D.predict(X)

mae_1D = mean_absolute_error(y_1D, pred_1D)
mae_5D = mean_absolute_error(y_5D, pred_5D)
```

## 9.4 Results Table

The final results are `R-Squared`, `RMSE`, and `MAE` which will determine the following:

- Model effectiveness between features and future returns
- Sensitivity to prediction for `1-Day` vs. `5-Day` future average returns
- The strength or weakness of each model in comparison

In [33]:
```
print("--- Result Table ---")
# Display result table with R2, RMSE, MAE
summary_df = pd.DataFrame(summary_data)
print(summary_df)

# Comparing MAE to Returns in same scale
print("\n\n---- Comparing MAE to Returns in same scale ---")
print("\n--- 1-Day Analysis ---")
print(f"MAE (1-Day): {mae_1D:.5f}")
print(f"Typical 1-Day Return Range: [{y_1D.min():.5f}, {y_1D.max():.5f}]")
print(f"Mean Absolute 1-Day Return: {np.mean(np.abs(y_1D)):.5f}")

print("\n--- 5-Day Analysis ---")
print(f"MAE (5-Day): {mae_5D:.5f}")
print(f"Typical 5-Day Return Range: [{y_5D.min():.5f}, {y_5D.max():.5f}]")
print(f"Mean Absolute 5-Day Return: {np.mean(np.abs(y_5D)):.5f}")

# Percentage MAE to Returns
print("\n--- Percentage MAE ---")
print(f"Percentage MAE (1-Day): { (mae_1D / np.mean(np.abs(y_1D))) * 100:.2f}%")
print(f"Percentage MAE (5-Day): { (mae_5D / np.mean(np.abs(y_5D))) * 100:.2f}%")
```

```
--- Result Table ---
   Model Target        R2      RMSE       MAE
0    MLR     1D -0.001710  0.012030  0.007995
1    MLR     5D -0.013481  0.024665  0.017365
2  Ridge     1D -0.001708  0.012030  0.007995
3  Ridge     5D -0.013473  0.024665  0.017365


---- Comparing MAE to Returns in same scale ---

--- 1-Day Analysis ---
MAE (1-Day): 0.00819
Typical 1-Day Return Range: [-0.10942, 0.14520]
Mean Absolute 1-Day Return: 0.00822

--- 5-Day Analysis ---
MAE (5-Day): 0.01762
Typical 5-Day Return Range: [-0.19793, 0.19404]
Mean Absolute 5-Day Return: 0.01786

--- Percentage MAE ---
Percentage MAE (1-Day): 99.60%
Percentage MAE (5-Day): 98.62%
```

### Summary of the Result Table

Based on the results, neither the MLR nor the Ridge model predicted the returns well:

- Ridge regression performed similarly to MLR, likely because the features weren't strongly related to the SPY price returns.
- The negative R-squared indicates that using features like RSI, volatility, and volume was actually worse than simply guessing the average future return.
- Even though the MAE values look small, they are very large (over 98%) when compared to the typical size of the actual returns.
- Predicting returns over 5 days was more difficult than predicting daily returns, as shown by the higher RMSE and MAE for the 5-day target.

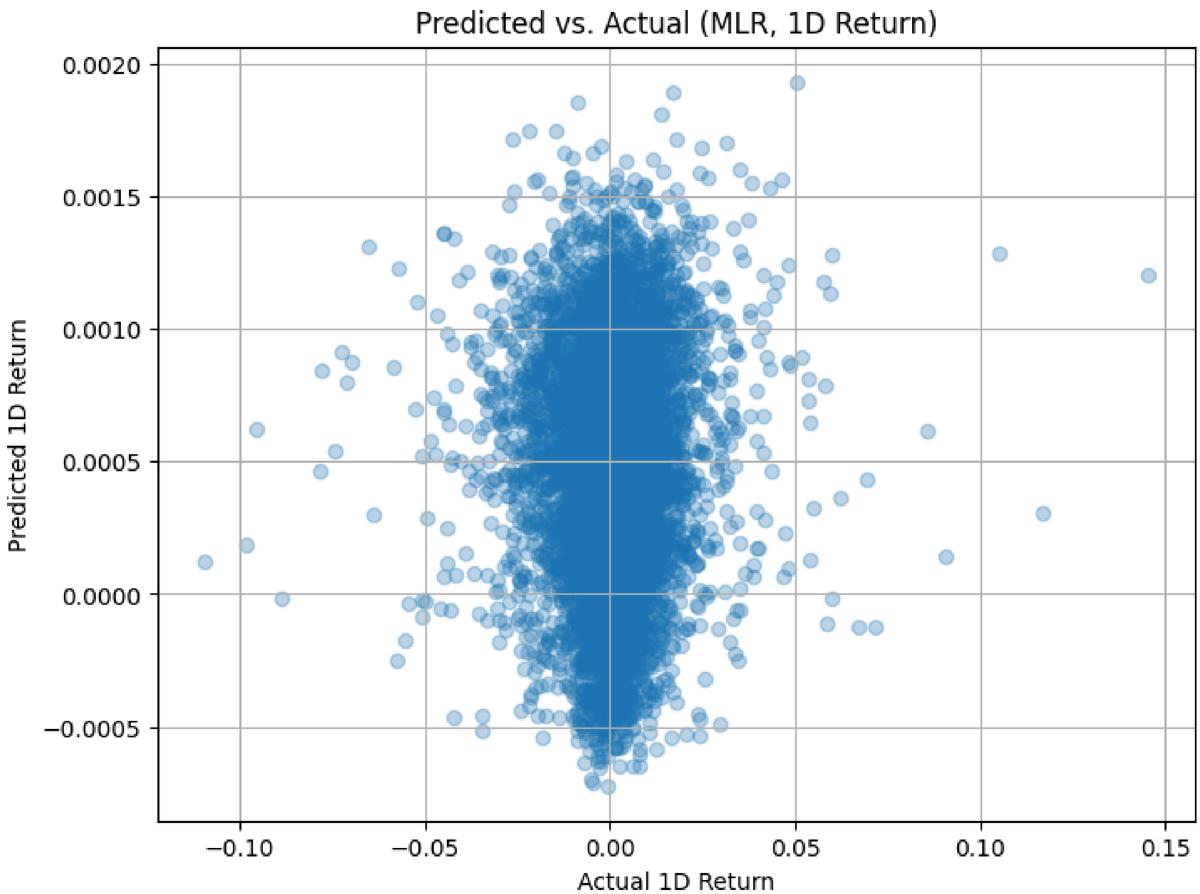
## 9.5 Visualizations

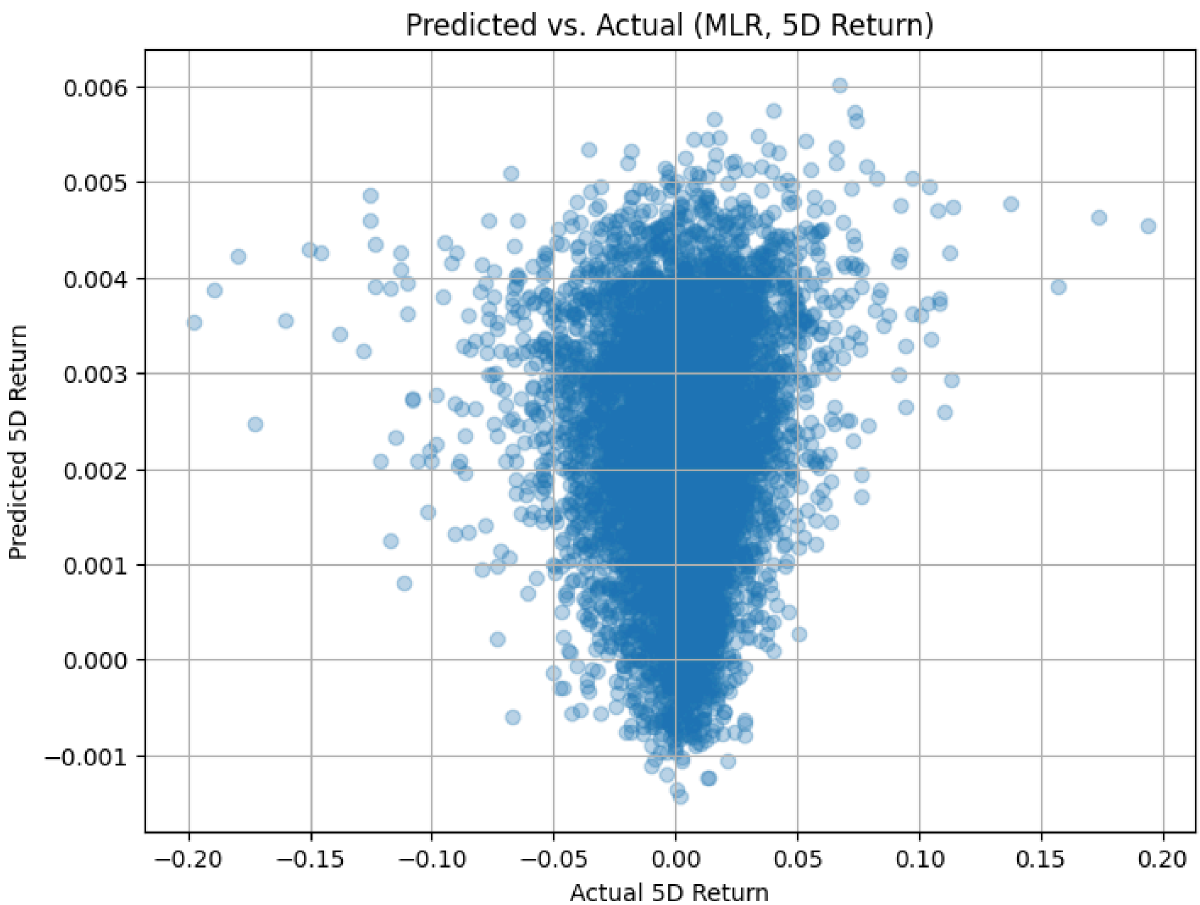
### 9.5.1 Predicted vs Actual Plot

Predicted vs Actual plot is to shows how closely predicted values match actual outcomes. Ideally, the points should lie diagonally, which implied perfect prediction.

In [34]:
```
# MLR Predicted vs Actual Plot with 1-Day return
plt.figure(figsize=(8, 6))
plt.scatter(y_1D, pred_1D, alpha=0.3)
plt.xlabel("Actual 1D Return")
plt.ylabel("Predicted 1D Return")
plt.title("Predicted vs. Actual (MLR, 1D Return)")
plt.grid(True)
```

```python
plt.show()

# MLR Predicted vs Actual Plot with 5-Day return
plt.figure(figsize=(8, 6))
plt.scatter(y_5D, pred_5D, alpha=0.3)
plt.xlabel("Actual 5D Return")
plt.ylabel("Predicted 5D Return")
plt.title("Predicted vs. Actual (MLR, 5D Return)")
plt.grid(True)
plt.show()
```





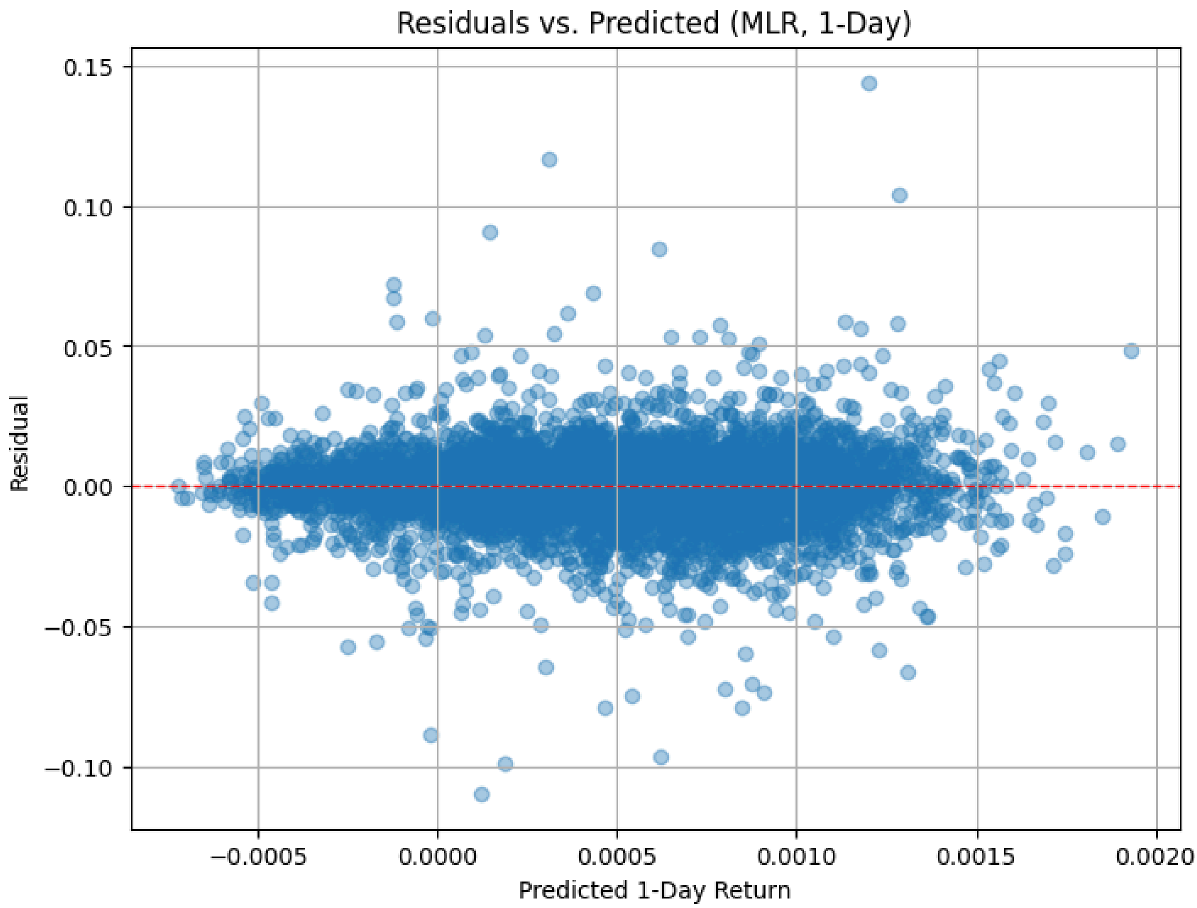

The points are clumped horizontally around the actual zero line on the x-axis, forming a "bee hive" shape. For the 1-day return, the predicted values has a narrow range between -0.0005 and 0.0015 implies the model only predicts values close to zero — only small variation are captured. The narrow spread of predicted values reflects limited predictive power. Predicted values show little variation, even though actual values vary more, which is a sign of underfitting. The lack of a diagonal slope implies weak correlation between predicted and actual returns.

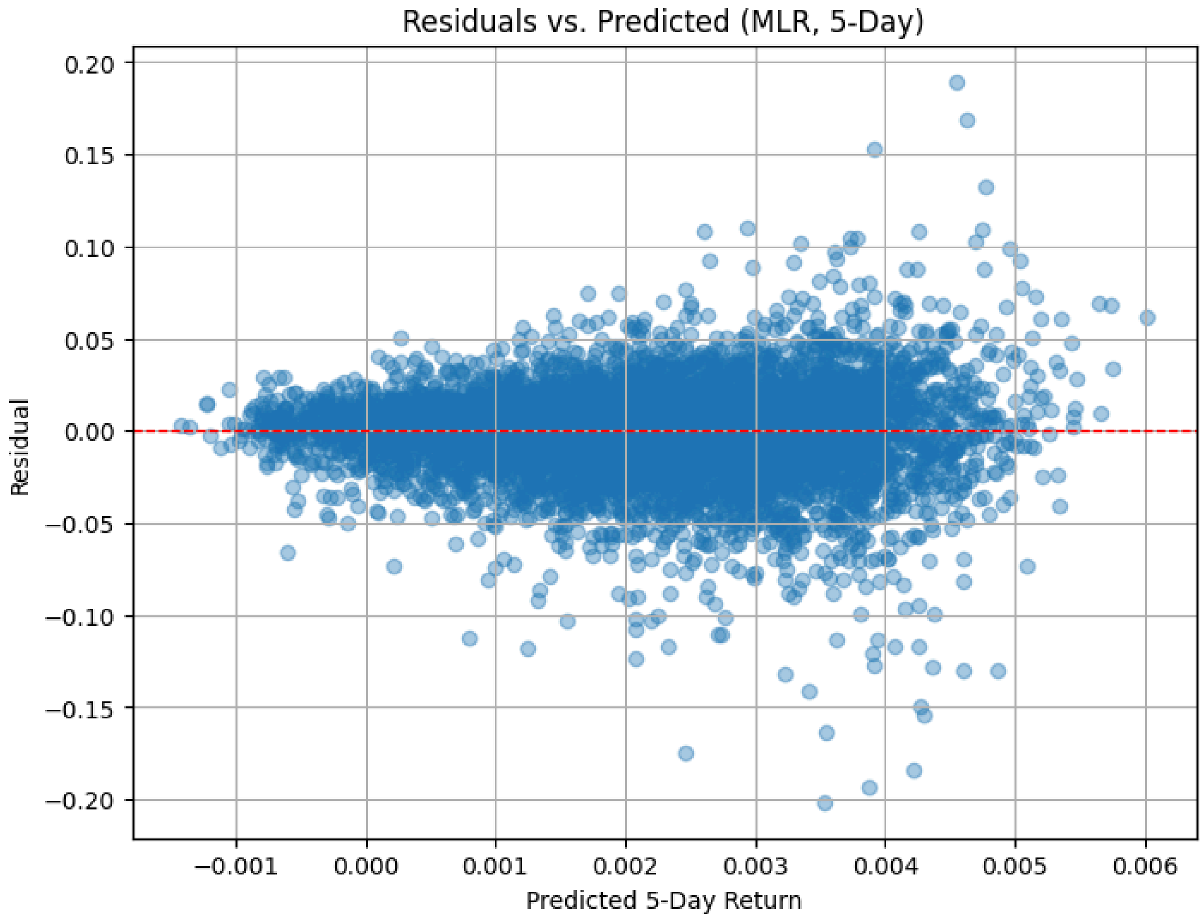
### 9.5.2 Residuals Plot

A Residuals vs. Predicted plot evaluates model errors and checks assumptions. The goal is to check whether the residuals are randomly scattered around zero with no pattern, which indicates that the model has no systematic bias.

In [35]:
```
# Residuals Plot for MLR 1-Day return
residuals_1D = y_1D - pred_1D
plt.figure(figsize=(8, 6))
plt.scatter(pred_1D, residuals_1D, alpha=0.4)
plt.axhline(0, color='red', linestyle='--', linewidth=1)
plt.xlabel("Predicted 1-Day Return")
plt.ylabel("Residual")
plt.title("Residuals vs. Predicted (MLR, 1-Day)")
plt.grid(True)
plt.show()
```

```python
# Residuals Plot for MLR 5-Day return
residuals_5D = y_5D - pred_5D
plt.figure(figsize=(8, 6))
plt.scatter(pred_5D, residuals_5D, alpha=0.4)
plt.axhline(0, color='red', linestyle='--', linewidth=1)
plt.xlabel("Predicted 5-Day Return")
plt.ylabel("Residual")
plt.title("Residuals vs. Predicted (MLR, 5-Day)")
plt.grid(True)
plt.show()
```






The residuals for the 1-day and 5-day return plots are evenly distributed vertically, but tightly clustered around zero with no visible pattern (Qualtrics, n.d.). This suggests the model does not suffer from major bias. However, the spread indicates low predictive power, consistent with the earlier negative R-Squared metrics and MAE near 100% . For the 1-day return, the predicted values has a narrow range between -0.0005 and 0.0015 implies the model only predicts values close to zero — only small variation are captured. Furthermore, the residuals range from about -0.10 to +0.15, which is large relative to the predicted values, i.e. the error is larger than the prediction.

## 9.6 Analysis

- MLR and Ridge models are not overfitting, but their features simply don't explain future returns well.
- Even complex non-linear models like Random Forest struggled, suggesting the problem isn't the model type.
- Both Ridge and Lasso were included for completeness and to show awareness of regularization techniques.
- Overall, short-term SPY returns are likely influenced by unpredictable noise, market mood, or external economic data not used here.

# 10. Discussion and Conclusion

This project aimed to determine if technical indicators alone could predict short-term SPY stock returns using supervised machine learning. After extensive testing with various models, including linear methods like MLR, Ridge, and Lasso, and more complex algorithms such as Random Forest and Logistic Regression, the answer is definitively no. None of the models achieved meaningful predictive performance, consistently showing low R-squared values and classification accuracy barely above chance.

Despite the models' poor predictive power, the project's setup had several strengths. Data was thoroughly cleaned and prepared, with features like Volume appropriately transformed. The modeling setup adhered to best practices, employing cross-validation, regularization, and robust feature selection methods. Additionally, clear visuals were effectively used to illustrate the models' significant limitations, revealing the ineffectiveness of the chosen features.

The key takeaway is that traditional technical indicators, on their own, are insufficient for accurate short-term forecasting. Financial markets are complex and noisy, influenced by many factors beyond historical price patterns. To improve stock market predictions, future efforts should focus on incorporating external data, such as market sentiment from social media or wider economic indicators (e.g., unemployment rate, CPI, GDP). Exploring weekly, monthly, or yearly SPY price data, rather than daily, could reduce noise and potentially reveal longer-term trends. Instead of predicting returns, considering price direction with a classification model like Logistic Regression is another avenue. Lastly, models like Long Short-Term Memory (LSTM) might be better at finding patterns in financial data due to their ability to remember long sequences (Mohammed, S., 2024; Wikipedia contributors, n.d.).

# References


Jones, K. (Jan 17, 2025). *Cross-validation for time series data*. Medium. https://medium.com/@kylejones_47003/cross-validation-for-time-series-data-51fd11c38e2b

KoshurAI. (Dec 19, 2023). *Understanding TimeSeriesSplit cross-validation for time series data*. Medium. https://koshurai.medium.com/understanding-timeseriessplit-cross-validation-for-time-series-data-4c232cc4f844

Mohammed, S. (Mar 13, 2024). *Understanding LSTM networks and financial forecasting: Decoding market trends with deep learning*. Medium. https://medium.com/@iamshahzu/understanding-lstm-networks-and-financial-forecasting-decoding-market-trends-with-deep-learning-6eb78a981be8

Qualtrics. (n.d.). *Interpreting residual plots to improve your regression*. https://www.qualtrics.com/support/stats-iq/analyses/regression-guides/interpreting-residual-plots-improve-regression/

Scikit-learn. (n.d.). *cross_val_score*. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

Scikit-learn. (n.d.). *TimeSeriesSplit*. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html

Wikipedia contributors. (n.d.). *Long short-term memory*. Wikipedia. https://en.wikipedia.org/wiki/Long_short-term_memory

Yahoo Finance. (n.d.). *SPDR S&P 500 ETF Trust (SPY) stock historical prices & data*. https://finance.yahoo.com/quote/SPY/history

## GitHub Repository Link

https://github.com/peculiardatabits/DTSA5509_Supervised_ML