ETF Forecasting

The following data set has been provided by BORIS MARJANOVIC on Kaggle. He has allowed free use of the data for personal use. The link to complete data set : https://www.kaggle.com/datasets/borismarjanovic/price-volume-data-for-all-us-stocks-etfs/data

We will be using several machine learning algorithms along with statistical methods to train and test out model. We begin by importing all the necessary libraries for this project.

In [1]:
```
import os
import pandas as pd
import seaborn as sns
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error, accuracy_score
```

For this project, we will be creating an **ETF** class that encapsulates all the necessary methods to analyze, modify, and train data models on ETF (Exchange-Traded Fund) data. The class provides functionalities for reading the data, cleaning it, generating descriptive statistics, plotting various financial metrics, and building a linear regression model to predict the closing prices.

Note that we are creating a class because all the datasets have same strucutre and heence would have same preprocessing and analysis. In case of a dataset with different structure, we would have to manually clean and preprocess that data

Overview of the ETF Class

- **Initialization**:
  - The class is initialized with the name of the ETF, which is used to load the corresponding data file.
- **Methods**:
  - **read_etfs**: Reads the ETF data from a CSV file and loads it into a pandas DataFrame.
  - **describe**: Provides detailed information about the DataFrame, including its structure, statistics, and any missing values.
  - **clean**: Cleans the data by converting the date column, dropping unnecessary columns, and adjusting the volume data for better readability.
  - **plot_folder**: Creates a directory structure to save plots for the ETF.
  - **plot_open_price**, **plot_close_price**, **plot_highs**, **plot_lows**, **plot_volume**: These methods individually handle the creation and saving of line and bar plots for different financial metrics (open price, close price, highs, lows, and volume) over time.
  - **plot_series** and **plot_bar**: Handle the creation of line and bar plots for different financial metrics over time.
  - **ml_preprocess**: Prepares the data for machine learning by adding new features such as day, month, and year extracted from the date.
  - **linearModel**: Builds and trains a linear regression model to predict the closing price of the ETF based on several features, and calculates the Mean Squared Error (MSE) and R^2 score for model evaluation.

With this class, we can efficiently perform a wide range of operations on ETF data, from basic exploratory data analysis to more advanced predictive modeling.

In [2]:
```
class ETF:
    def __init__(self, name):
        self.name = name
        self.df = self.read_etfs()
        self.cleaned_df = None
        self.r2Score = None
        self.mse = None
        self.trainX = None
        self.trainY  = None
        self.testX = None
        self.testY = None
        self.model = None
        self.features = None
        self.target = None

    def read_etfs(self) -> pd.DataFrame:
        df = pd.read_csv(f"./Data/ETFs/{self.name}.us.txt")
```

```python
        return df

    def describe(self, df: pd.DataFrame):
        print("DataFrame Information:")
        df.info()

        print("\nDescriptive Statistics:")
        print(df.describe())

        print("\nFirst 5 Rows:")
        print(df.head())

        print("\nLast 5 Rows:")
        print(df.tail())

        print("\nDataFrame Shape (rows, columns):")
        print(df.shape)

        print("\nRandom Sample of 5 Rows:")
        print(df.sample(5))

        print("\nMissing Values Count per Column:")
        print(df.isnull().sum())
        print()

    def clean(self) -> pd.DataFrame:
        df = self.df.copy()
        df["Date"] = pd.to_datetime(df["Date"])
        df = df.drop("OpenInt", axis=1)
        df["Volume"] = df["Volume"] / 10**7
        df = df.rename(columns={"Volume": "Volume in Millions"})
        self.cleaned_df = df
        return df

    def plot_folder(self):
        if not os.path.exists(f"./Plots/ETFs/{self.name}"):
            os.makedirs(f"./Plots/ETFs/{self.name}")
        return

    def plot_open_price(self):
        if self.cleaned_df is None:
            raise ValueError("Data is not cleaned before plotting.")
        self.plot_folder()
        self.plot_series(self.cleaned_df, 'Open', 'Open Price')

    def plot_close_price(self):
        if self.cleaned_df is None:
            raise ValueError("Data is not cleaned before plotting.")
        self.plot_folder()
        self.plot_series(self.cleaned_df, 'Close', 'Close Price')

    def plot_highs(self):
        if self.cleaned_df is None:
            raise ValueError("Data is not cleaned before plotting.")
        self.plot_folder()
        self.plot_series(self.cleaned_df, 'High', 'Highs')

    def plot_lows(self):
        if self.cleaned_df is None:
            raise ValueError("Data is not cleaned before plotting.")
        self.plot_folder()
        self.plot_series(self.cleaned_df, 'Low', 'Lows')

    def plot_volume(self):
        if self.cleaned_df is None:
            raise ValueError("Data is not cleaned before plotting.")
        self.plot_folder()
        self.plot_bar(self.cleaned_df, 'Volume in Millions', 'Volume')

    def plot_series(self, df, column, title):
        plt.figure(figsize=(8, 5))
        plt.plot(df["Date"], df[column], color='royalblue', linestyle='-', linewidth=1, label=title)
        plt.title(f'ETF {title}', fontsize=16)
        plt.xlabel('Date', fontsize=14)
        plt.ylabel(title, fontsize=14)
        plt.grid(True, linestyle='--', alpha=0.7)
        plt.tight_layout()
```

```python
        plt.savefig(f"./Plots/ETFs/{self.name}/{title}.png")

    def plot_bar(self, df, column, title):
        plt.figure(figsize=(8, 5))
        plt.bar(df["Date"], df[column], color='royalblue', label=title)
        plt.title(f'ETF {title}', fontsize=16)
        plt.xlabel('Date', fontsize=14)
        plt.ylabel(title, fontsize=14)
        plt.grid(True, linestyle='--', alpha=0.7)
        plt.tight_layout()
        plt.savefig(f"./Plots/ETFs/{self.name}/{title}.png")

    def ml_preprocess(self, df: pd.DataFrame) -> pd.DataFrame:
        df["Date"] = pd.to_datetime(df["Date"])
        df["Day"] = df["Date"].dt.dayofweek
        df["Month"] = df["Date"].dt.month
        df["Year"] = df["Date"].dt.year
        df = df.drop('Date',axis=1)
        df.dropna(inplace=True)
        return df

    def linearModel(self, df, features, target):
        X = df[features]
        Y = df[target]
        trainX, testX, trainY, testY = train_test_split(X, Y, random_state = 0)
        model = LinearRegression()
        model.fit(trainX, trainY)
        predY = model.predict(testX)
        self.mse = mean_squared_error(testY, predY)
        self.r2Score = r2_score(testY, predY)
        self.trainX = trainX
        self.trainY = trainY
        self.testX = testX
        self.testY = testY
        self.model = model
        self.features = features
        self.target = target

    def dtr(self, df, features, target, mln = None):
        X = df[features]
        Y = df[target]
        trainX, testX, trainY, testY = train_test_split(X, Y, random_state = 0)
        model = DecisionTreeRegressor(random_state = 0, max_leaf_nodes = mln)
        model.fit(trainX, trainY)
        predY = model.predict(testX)
        self.mse = mean_squared_error(testY, predY)
        self.r2Score = r2_score(testY, predY)
        self.trainX = trainX
        self.trainY = trainY
        self.testX = testX
        self.testY = testY
        self.model = model
        self.features = features
        self.target = target

    def dtrCheck(self):
        def get_mae(maxLeaf, trainX, testX, trainY, testY):
            model = DecisionTreeRegressor(max_leaf_nodes = maxLeaf, random_state = 0)
            model.fit(trainX, trainY)
            preds_val = model.predict(testX)
            mae = mean_absolute_error(testY, preds_val)
            return(mae)

        for maxLeaf in [5, 50, 500, 5000]:
            my_mae = get_mae(maxLeaf, self.trainX, self.testX, self.trainY, self.testY)
            print(f"Max leaf nodes: {maxLeaf}  \t\t Mean Absolute Error:  {my_mae}")

    def rfr(self, df, features, target):
        X = df[features]
        Y = df[target]
        trainX, testX, trainY, testY = train_test_split(X, Y, random_state = 0)
        model = RandomForestRegressor(random_state = 0)
        model.fit(trainX, trainY)
        predY = model.predict(testX)
        self.mse = mean_squared_error(testY, predY)
        self.r2Score = r2_score(testY, predY)
```

```
        self.trainX = trainX
        self.trainY = trainY
        self.testX = testX
        self.testY = testY
        self.model = model
        self.features = features
        self.target = target

    def plotPredictions(self):
        if self.testX is None or self.testY is None:
            raise ValueError("Model has not been trained or data is not prepared.")
        model = self.model
        predY = model.predict(self.testX)
        plt.figure(figsize=(10, 6))
        plt.scatter(self.testY, predY, color='royalblue', alpha=0.7, edgecolors='w', linewidth=0.5)
        plt.plot([min(self.testY), max(self.testY)], [min(self.testY), max(self.testY)], color='red',
        plt.title('Predictions vs Actuals', fontsize=16)
        plt.xlabel('Actual Values', fontsize=14)
        plt.ylabel('Predicted Values', fontsize=14)
        plt.grid(True, linestyle='--', alpha=0.7)
        plt.tight_layout()
        plt.show()
```

Analysis

Now the class is ready to be called. We can see the various results that the analysis will give us. For the purpose of this project we will be using the historic data of **QQQ (Invesco QQQ Trust, Series 1)**. We begin by assigning the ticker as the default value. For analysis of other datasets, the ticker's value can be changed accordingly.

In [3]:
```
ticker = "qqq"
etf = ETF(ticker)
```

Now we use the **describe** method for the ETF class to describe the data set we have.

In [4]:
```
etf.describe(etf.df)
```

```
DataFrame Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4701 entries, 0 to 4700
Data columns (total 7 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   Date     4701 non-null   object 
 1   Open     4701 non-null   float64
 2   High     4701 non-null   float64
 3   Low      4701 non-null   float64
 4   Close    4701 non-null   float64
 5   Volume   4701 non-null   int64  
 6   OpenInt  4701 non-null   int64  
dtypes: float64(4), int64(2), object(1)
memory usage: 257.2+ KB

Descriptive Statistics:
              Open         High          Low        Close        Volume  \
count  4701.000000  4701.000000  4701.000000  4701.000000  4.701000e+03   
mean     58.398648    58.888507    57.837278    58.386467  8.054378e+07   
std      31.211635    31.316778    31.071677    31.220362  5.903922e+07   
min      17.830000    18.361000    17.665000    17.938000  5.828392e+06   
25%      34.904000    35.173000    34.559000    34.876000  3.447708e+07   
50%      45.743000    46.112000    45.279000    45.656000  7.083852e+07   
75%      79.321000    80.286000    78.248000    79.160000  1.074447e+08   
max     153.810000   154.540000   153.620000   154.510000  6.755370e+08   

       OpenInt  
count   4701.0  
mean       0.0  
std        0.0  
min        0.0  
25%        0.0  
50%        0.0  
75%        0.0  
max        0.0  

First 5 Rows:
         Date    Open    High     Low   Close    Volume  OpenInt
0  1999-03-10  45.722  45.750  44.967  45.665  11700414        0
1  1999-03-11  45.994  46.260  44.988  45.880  21670048        0
2  1999-03-12  45.721  45.749  44.406  44.770  19553768        0
3  1999-03-15  45.101  46.103  44.625  46.052  14245348        0
4  1999-03-16  46.253  46.643  45.749  46.447  10971066        0

Last 5 Rows:
            Date    Open     High     Low   Close    Volume  OpenInt
4696  2017-11-06  153.13  153.850  153.10  153.75  28685854        0
4697  2017-11-07  153.67  154.082  153.34  153.87  21285469        0
4698  2017-11-08  153.81  154.540  153.62  154.51  17326500        0
4699  2017-11-09  153.26  153.770  152.11  153.69  40554952        0
4700  2017-11-10  153.36  153.800  153.06  153.68  20138114        0

DataFrame Shape (rows, columns):
(4701, 7)

Random Sample of 5 Rows:
            Date    Open    High     Low   Close     Volume  OpenInt
2697  2009-11-27  39.467  40.260  39.402  39.966   66662018        0
2242  2008-02-08  39.391  39.912  39.156  39.756  190578044        0
3941  2014-11-06  98.138  98.526  97.769  98.497   25462581        0
1099  2003-07-25  27.834  28.467  27.511  28.431   92503804        0
3263  2012-02-28  59.920  60.482  59.856  60.482   46771305        0

Missing Values Count per Column:
Date       0
Open       0
High       0
Low        0
Close      0
Volume     0
OpenInt    0
dtype: int64
```

We can see that there are 7 columns in the dataset with Date beign reffered as an object data type. There are total of 4701 records and the last cloumn **OpenInt** has all the records set to zero. There are no missing values in any column, which is extremely good. Now we will clean and preprocess the data for plotting some basic charts, to gain more insight.

Data Cleaning

In [5]:
```
etf.clean()
etf.describe(etf.cleaned_df)
```

```
DataFrame Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4701 entries, 0 to 4700
Data columns (total 6 columns):
 #   Column             Non-Null Count  Dtype         
---  ------             --------------  -----         
 0   Date               4701 non-null   datetime64[ns]
 1   Open               4701 non-null   float64       
 2   High               4701 non-null   float64       
 3   Low                4701 non-null   float64       
 4   Close              4701 non-null   float64       
 5   Volume in Millions  4701 non-null   float64       
dtypes: datetime64[ns](1), float64(5)
memory usage: 220.5 KB

Descriptive Statistics:
                                Date         Open         High          Low  \
count                           4701  4701.000000  4701.000000  4701.000000   
mean   2008-07-12 15:48:21.595405568    58.398648    58.888507    57.837278   
min              1999-03-10 00:00:00    17.830000    18.361000    17.665000   
25%              2003-11-11 00:00:00    34.904000    35.173000    34.559000   
50%              2008-07-15 00:00:00    45.743000    46.112000    45.279000   
75%              2013-03-15 00:00:00    79.321000    80.286000    78.248000   
max              2017-11-10 00:00:00   153.810000   154.540000   153.620000   
std                              NaN    31.211635    31.316778    31.071677   

             Close  Volume in Millions  
count  4701.000000         4701.000000  
mean     58.386467            8.054378  
min      17.938000            0.582839  
25%      34.876000            3.447708  
50%      45.656000            7.083852  
75%      79.160000           10.744472  
max     154.510000           67.553702  
std      31.220362            5.903922  

First 5 Rows:
        Date    Open    High     Low   Close  Volume in Millions
0 1999-03-10  45.722  45.750  44.967  45.665            1.170041
1 1999-03-11  45.994  46.260  44.988  45.880            2.167005
2 1999-03-12  45.721  45.749  44.406  44.770            1.955377
3 1999-03-15  45.101  46.103  44.625  46.052            1.424535
4 1999-03-16  46.253  46.643  45.749  46.447            1.097107

Last 5 Rows:
           Date    Open     High     Low   Close  Volume in Millions
4696 2017-11-06  153.13  153.850  153.10  153.75            2.868585
4697 2017-11-07  153.67  154.082  153.34  153.87            2.128547
4698 2017-11-08  153.81  154.540  153.62  154.51            1.732650
4699 2017-11-09  153.26  153.770  152.11  153.69            4.055495
4700 2017-11-10  153.36  153.800  153.06  153.68            2.013811

DataFrame Shape (rows, columns):
(4701, 6)

Random Sample of 5 Rows:
           Date     Open     High      Low    Close  Volume in Millions
3992 2015-01-22   99.690  101.250   98.816  101.150            3.549803
4489 2017-01-11  121.780  122.070  121.170  122.070            1.821072
3728 2014-01-03   83.569   83.644   82.945   82.964            3.730892
1674 2005-11-03   35.920   36.235   35.865   36.103           15.162473
435  2000-11-28   61.645   62.540   58.398   58.524            5.458921

Missing Values Count per Column:
Date                  0
Open                  0
High                  0
Low                   0
Close                 0
Volume in Millions    0
dtype: int64
```

As we can see, the **Date** column has been converted to *datetime64* object. For the sake of simplicity in plotting we have converted **Volume** column to **Volume in Millions**. This will help us plot the data on graph closer to measurable scale. Now we can move on to plot different plot and analyse them accordingly.

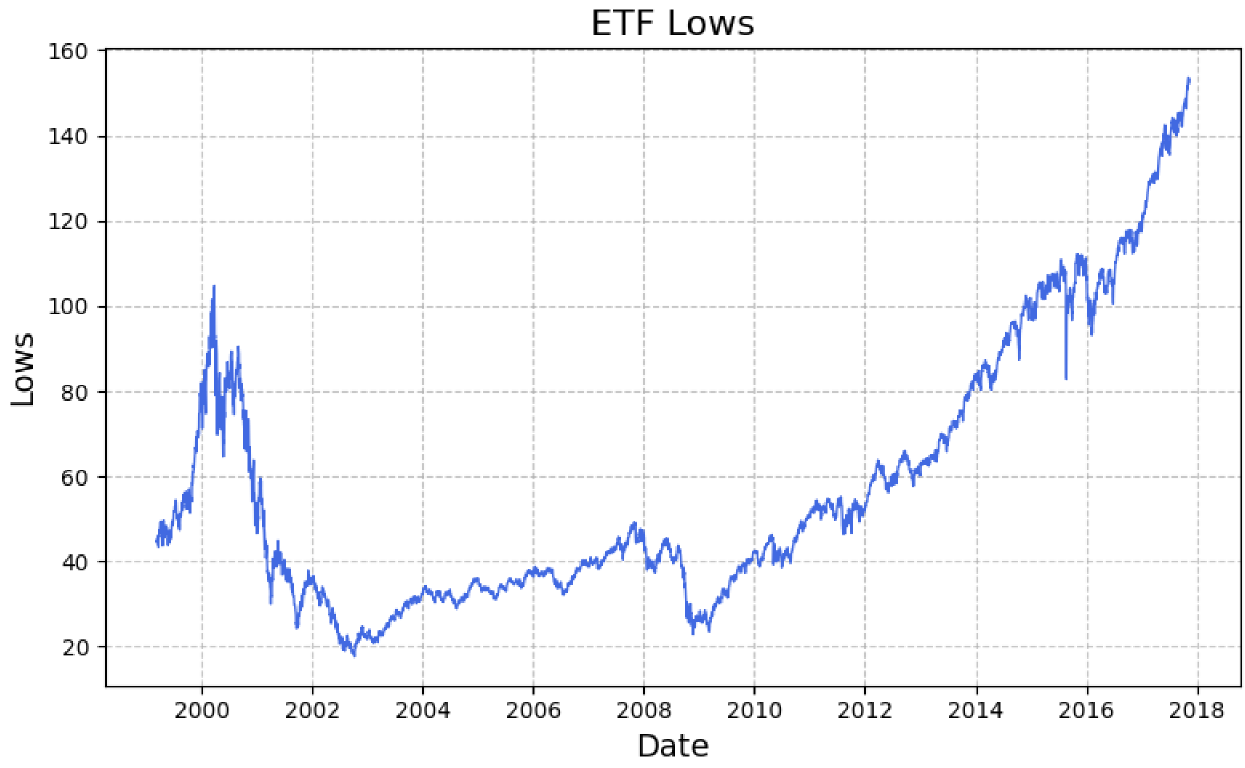
Data Visualization

In [6]: `etf.plot_open_price()`

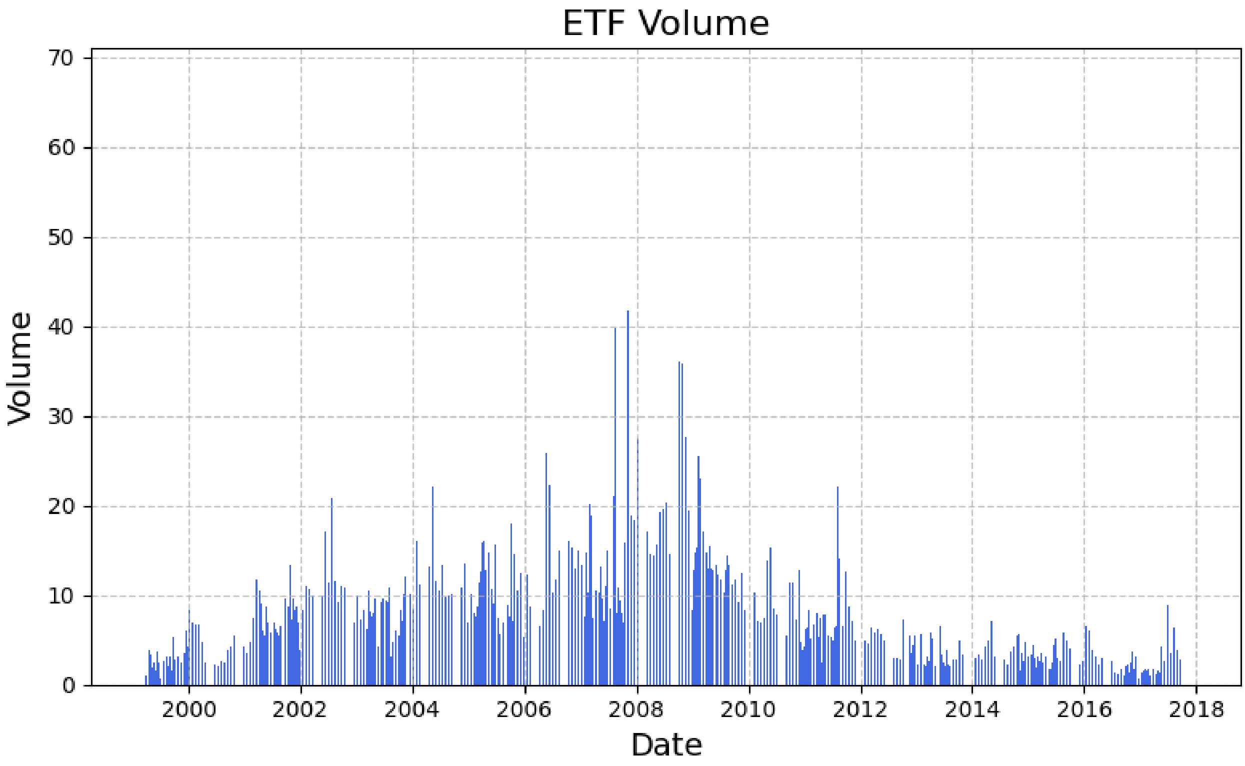
In [7]: `etf.plot_close_price()`

In [8]: `etf.plot_highs()`

In [9]: `etf.plot_lows()`



In [10]: `etf.plot_volume()`


ETF Volume
Volume
Date
0
10
20
30
40
50
60
70
2000
2002
2004
2006
2008
2010
2012
2014
2016
2018

Data Expansion

In financial analysis, enhancing a dataset with additional columns can provide deeper insights into the behavior and trends of a financial instrument, such as the QQQ ETF. Below are the added columns and their significance:

1. **Daily Return**:
    - **Purpose**: This column represents the percentage change between the opening and closing prices of the ETF each day. It helps in understanding the day-to-day price movement and volatility.
    - **Importance**: Daily returns are crucial for calculating various performance metrics, such as average returns, and are also used in risk management to assess the volatility of an asset.
2. **20-Day, 50-Day, and 200-Day Moving Averages**:
    - **Purpose**: Moving averages smooth out price data to help identify the direction of the trend over specific periods (short-term, medium-term, and long-term).
    - **Importance**: These indicators are widely used in technical analysis to spot trends and potential reversal points. For example, a 50-day MA crossing above the 200-day MA (known as a "Golden Cross") is often seen as a bullish signal.
3. **30-Day Volatility**:
    - **Purpose**: Volatility is calculated as the rolling standard deviation of daily returns over the past 30 days.
    - **Importance**: Understanding volatility is key to assessing the risk associated with the ETF. High volatility indicates large price swings, which could mean higher risk and potential return.
4. **Relative Strength Index (RSI)**:
    - **Purpose**: RSI is a momentum oscillator that measures the speed and change of price movements, typically used to identify overbought or oversold conditions.
    - **Importance**: RSI helps traders make informed decisions by indicating whether an asset is potentially overvalued or undervalued, which could signal a trend reversal.
5. **Bollinger Bands**:
    - **Components**: Middle Band (20-day MA), Upper Band (Middle Band + 2 standard deviations), Lower Band (Middle Band - 2 standard deviations).
    - **Purpose**: Bollinger Bands provide a visual representation of volatility and relative price levels.
    - **Importance**: These bands help in identifying overbought and oversold conditions. Prices tend to bounce between the upper and lower bands, making this a useful tool for spotting potential buy and sell points.
6. **Cumulative Return**:
    - **Purpose**: This column shows the total return of the ETF relative to its initial closing price.
    - **Importance**: Cumulative return is vital for understanding the overall performance of the ETF over time. It gives investors a clear picture of how their investment has grown or shrunk.
7. **Volume Weighted Average Price (VWAP)**:
    - **Purpose**: VWAP is the average price at which the ETF has traded throughout the day, weighted by volume.
    - **Importance**: VWAP is often used as a benchmark by traders to determine the quality of their trade executions. It provides insight into the price levels at which most trading occurred, which can help in identifying support and resistance levels.

Each of these columns provides valuable insights into different aspects of the ETF's behavior, enabling more informed decision-making and better risk management.

In [11]:
```
def expand(df):
        df["Daily Return"] = (df["Close"] - df["Open"]) / df["Open"] * 100

        df["20-Day MA"] = df["Close"].rolling(window=20).mean()
        df["50-Day MA"] = df["Close"].rolling(window=50).mean()
        df["200-Day MA"] = df["Close"].rolling(window=200).mean()

        df["30-Day Volatility"] = df["Daily Return"].rolling(window=30).std()

        delta = df["Close"].diff()
        gain = (delta.where(delta > 0, 0)).rolling(window=14).mean()
        loss = (-delta.where(delta < 0, 0)).rolling(window=14).mean()
        rs = gain / loss
        df["RSI"] = 100 - (100 / (1 + rs))

        df["Middle Band"] = df["20-Day MA"]
        df["Upper Band"] = df["Middle Band"] + (2 * df["Close"].rolling(window=20).std())
        df["Lower Band"] = df["Middle Band"] - (2 * df["Close"].rolling(window=20).std())

        df["Cumulative Return"] = (df["Close"] / df["Close"].iloc[0]) - 1

        df["VWAP"] = (df["Volume in Millions"] * df["Close"]).cumsum() / df["Volume in Millions"].cu

expand(etf.cleaned_df)
etf.describe(etf.cleaned_df)
```

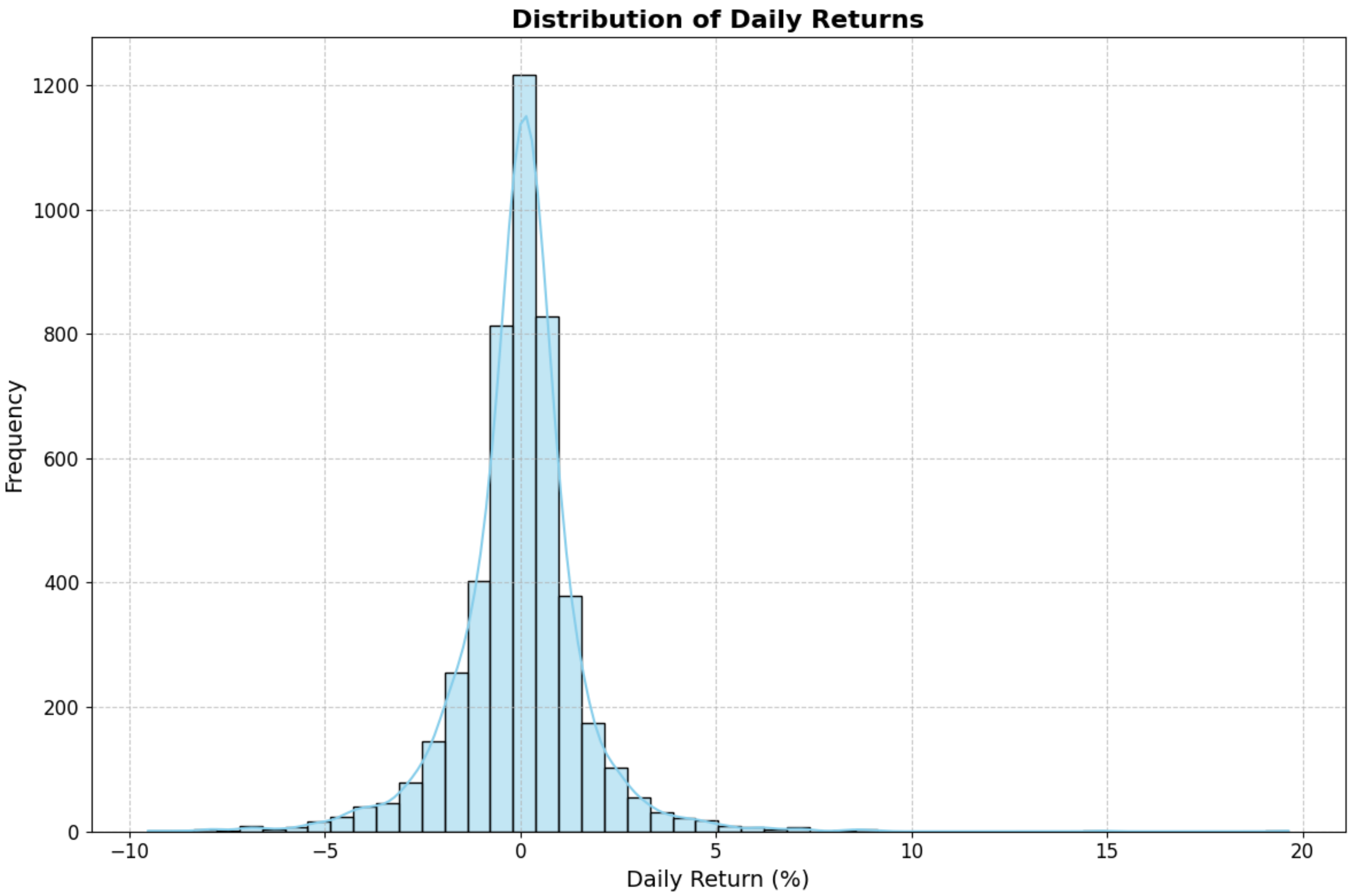
```
DataFrame Information:
```

```
DataFrame Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4701 entries, 0 to 4700
Data columns (total 17 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Date                4701 non-null   datetime64[ns]
 1   Open                4701 non-null   float64
 2   High                4701 non-null   float64
 3   Low                 4701 non-null   float64
 4   Close               4701 non-null   float64
 5   Volume in Millions  4701 non-null   float64
 6   Daily Return        4701 non-null   float64
 7   20-Day MA           4682 non-null   float64
 8   50-Day MA           4652 non-null   float64
 9   200-Day MA          4502 non-null   float64
 10  30-Day Volatility   4672 non-null   float64
 11  RSI                 4688 non-null   float64
 12  Middle Band         4682 non-null   float64
 13  Upper Band          4682 non-null   float64
 14  Lower Band          4682 non-null   float64
 15  Cumulative Return   4701 non-null   float64
 16  VWAP                4701 non-null   float64
dtypes: datetime64[ns](1), float64(16)
memory usage: 624.5 KB

Descriptive Statistics:
                                Date         Open         High          Low  \
count                           4701  4701.000000  4701.000000  4701.000000
mean   2008-07-12 15:48:21.595405568    58.398648    58.888507    57.837278
min              1999-03-10 00:00:00    17.830000    18.361000    17.665000
25%              2003-11-11 00:00:00    34.904000    35.173000    34.559000
50%              2008-07-15 00:00:00    45.743000    46.112000    45.279000
75%              2013-03-15 00:00:00    79.321000    80.286000    78.248000
max              2017-11-10 00:00:00   153.810000   154.540000   153.620000
std                              NaN    31.211635    31.316778    31.071677

             Close  Volume in Millions  Daily Return    20-Day MA  \
count  4701.000000         4701.000000   4701.000000  4682.000000
mean     58.386467            8.054378     -0.014040    58.221481
min      17.938000            0.582839     -9.524849    19.021850
25%      34.876000            3.447708     -0.633859    34.846650
50%      45.656000            7.083852      0.056777    45.631775
75%      79.160000           10.744472      0.661173    79.047137
max     154.510000           67.553702     19.639184   150.777000
std      31.220362            5.903922      1.566960    30.949979

         50-Day MA   200-Day MA  30-Day Volatility          RSI  Middle Band  \
count  4652.000000  4502.000000        4672.000000  4688.000000  4682.000000
mean     57.966895    56.701543           1.300707    54.554532    58.221481
min      20.300720    22.002960           0.227525     5.335683    19.021850
25%      34.714480    34.392620           0.723227    42.216829    34.846650
50%      44.660040    44.061610           0.960163    53.743091    45.631775
75%      79.454435    74.404424           1.575454    66.205129    79.047137
max     147.710600   138.945150           5.589283   100.000000   150.777000
std      30.561356    28.688344           0.887698    16.660681    30.949979

        Upper Band   Lower Band  Cumulative Return         VWAP
count  4682.000000  4682.000000        4701.000000  4701.000000
mean     60.830193    55.612769           0.278582    42.645885
min      20.337688    16.993830          -0.607183    36.248469
25%      36.275861    33.231894          -0.236264    37.289467
50%      48.151883    43.468702          -0.000197    39.018451
75%      84.742485    73.137090           0.733494    43.510343
max     155.897794   145.656206           2.383554    72.384166
std      31.735176    30.280653           0.683683     8.850508

First 5 Rows:
        Date    Open    High     Low   Close  Volume in Millions  \
0 1999-03-10  45.722  45.750  44.967  45.665            1.170041
1 1999-03-11  45.994  46.260  44.988  45.880            2.167005
2 1999-03-12  45.721  45.749  44.406  44.770            1.955377
3 1999-03-15  45.101  46.103  44.625  46.052            1.424535
4 1999-03-16  46.253  46.643  45.749  46.447            1.097107

   Daily Return  20-Day MA  50-Day MA  200-Day MA  30-Day Volatility  RSI  \
0     -0.124666        NaN        NaN         NaN                NaN  NaN
1      0.247858        NaN        NaN         NaN                NaN  NaN
```

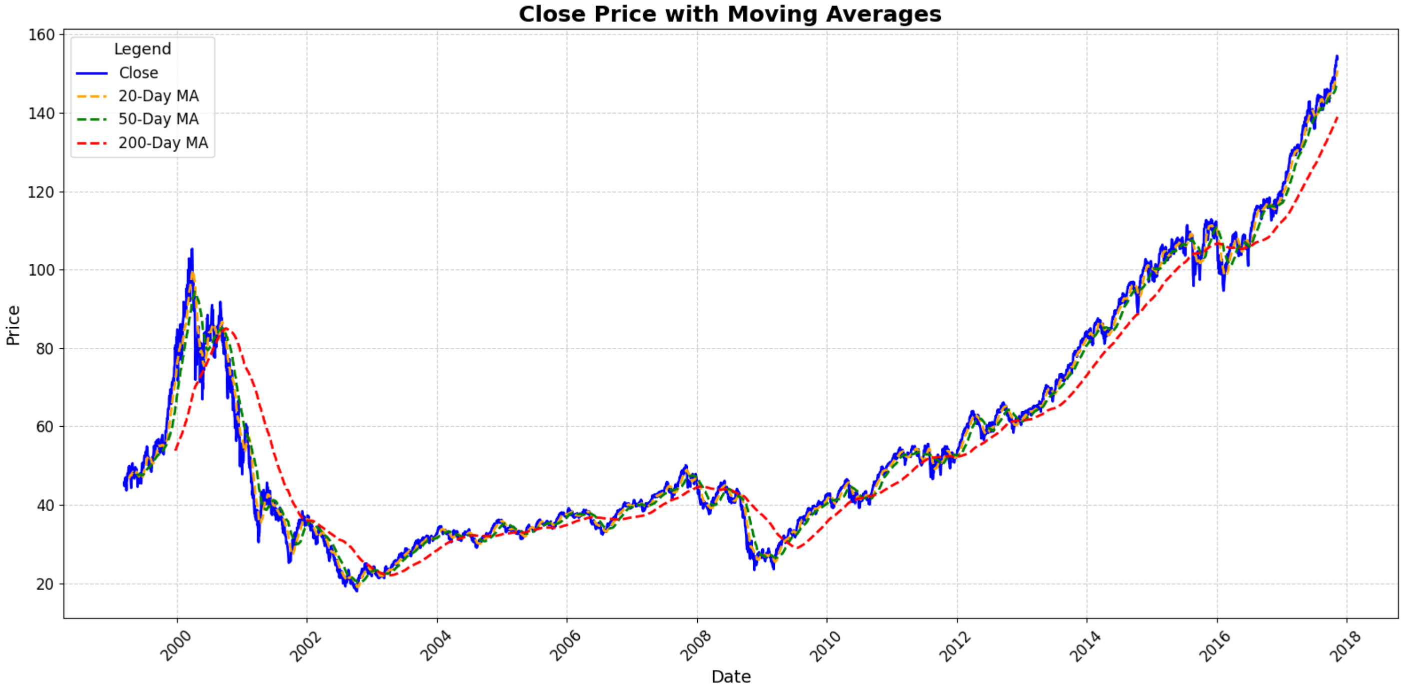
```
1     -0.247858        NaN        NaN          NaN               NaN  NaN
2     -2.080007        NaN        NaN          NaN               NaN  NaN
3      2.108601        NaN        NaN          NaN               NaN  NaN
4      0.419432        NaN        NaN          NaN               NaN  NaN

   Middle Band  Upper Band  Lower Band  Cumulative Return       VWAP
0          NaN         NaN         NaN           0.000000  45.665000
1          NaN         NaN         NaN           0.004708  45.804616
2          NaN         NaN         NaN          -0.019599  45.422359
3          NaN         NaN         NaN           0.008475  45.555894
4          NaN         NaN         NaN           0.017125  45.681007

Last 5 Rows:
           Date    Open     High     Low   Close  Volume in Millions  \
4696 2017-11-06  153.13  153.850  153.10  153.75            2.868585
4697 2017-11-07  153.67  154.082  153.34  153.87            2.128547
4698 2017-11-08  153.81  154.540  153.62  154.51            1.732650
4699 2017-11-09  153.26  153.770  152.11  153.69            4.055495
4700 2017-11-10  153.36  153.800  153.06  153.68            2.013811

      Daily Return  20-Day MA  50-Day MA  200-Day MA  30-Day Volatility  \
4696      0.404885   149.5770   146.8950   138.35380           0.396822
4697      0.130149   149.8905   147.1304   138.50660           0.391770
4698      0.455107   150.2140   147.3674   138.65640           0.394058
4699      0.280569   150.5100   147.5546   138.80140           0.395071
4700      0.208659   150.7770   147.7106   138.94515           0.388193

            RSI  Middle Band  Upper Band  Lower Band  Cumulative Return  \
4696  71.705069     149.5770  153.878961  145.275039           2.366911
4697  73.215941     149.8905  154.489431  145.291569           2.369539
4698  78.492647     150.2140  155.161848  145.266152           2.383554
4699  72.035398     150.5100  155.549691  145.470309           2.365597
4700  78.723404     150.7770  155.897794  145.656206           2.365378

           VWAP
4696  45.190377
4697  45.196487
4698  45.201490
4699  45.213111
4700  45.218880

DataFrame Shape (rows, columns):
(4701, 17)

Random Sample of 5 Rows:
           Date     Open     High      Low    Close  Volume in Millions  \
4258 2016-02-11   93.742   95.577   93.486   94.823            6.965338
4283 2016-03-18  105.840  106.020  105.200  105.770            3.825582
4005 2015-02-10  100.510  101.640  100.350  101.500            2.435834
2636 2009-09-01   36.605   37.246   35.935   36.047           17.758259
4428 2016-10-13  115.480  116.150  114.800  115.850            2.189971

      Daily Return  20-Day MA  50-Day MA  200-Day MA  30-Day Volatility  \
4258      1.153165   99.07550  104.97840  106.055885           1.348504
4283     -0.066138  103.17300  101.01946  105.537150           1.070074
4005      0.984977   99.47675  100.10394   94.499145           0.971024
2636     -1.524382   36.49555   35.08356   30.118755           0.928515
4428      0.320402  116.91800  115.99960  108.057215           0.752173

            RSI  Middle Band  Upper Band  Lower Band  Cumulative Return  \
4258  29.298174     99.07550  104.325032   93.825968           1.076492
4283  78.396437    103.17300  106.732663   99.613337           1.316216
4005  58.956805     99.47675  102.114559   96.838941           1.222709
2636  44.247039     36.49555   37.467433   35.523667          -0.210621
4428  41.955446    116.91800  118.400368  115.435632           1.536954

           VWAP
4258  42.862088
4283  42.983217
4005  41.351993
2636  36.956930
4428  43.506917

Missing Values Count per Column:
Date                    0
Open                    0
High                    0
Low                     0
```

```
Low                       0
Close                     0
Volume in Millions        0
Daily Return              0
20-Day MA                19
50-Day MA                49
200-Day MA              199
30-Day Volatility        29
RSI                      13
Middle Band              19
Upper Band               19
Lower Band               19
Cumulative Return         0
VWAP                      0
dtype: int64
```

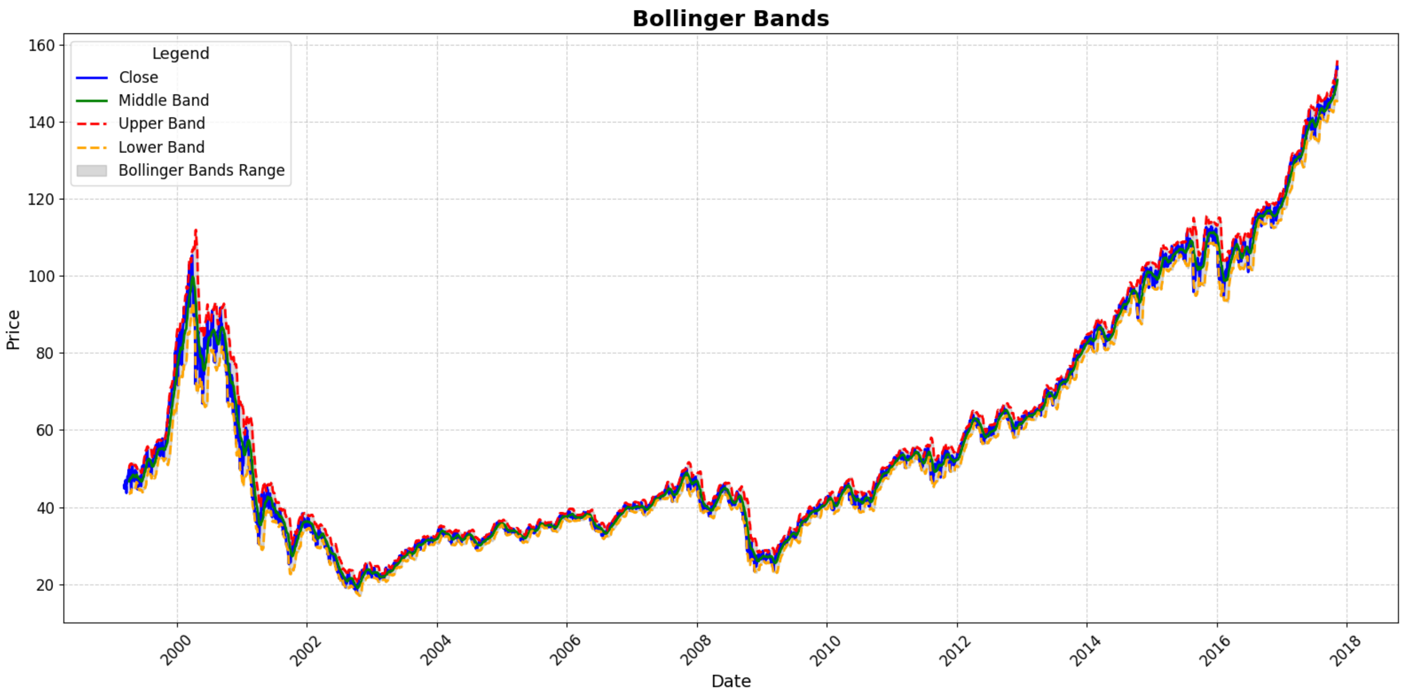
In [12]:
```
plt.figure(figsize=(12, 8))
sns.histplot(etf.cleaned_df['Daily Return'].dropna(), kde=True, bins=50, color='skyblue', edgecolor=
plt.title('Distribution of Daily Returns', fontsize=16, fontweight='bold')
plt.xlabel('Daily Return (%)', fontsize=14)
plt.ylabel('Frequency', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.savefig(f"./Plots/ETFs/{etf.name}/Daily Returns.png")
```



In [13]:
```
plt.figure(figsize=(16, 8))
sns.lineplot(x='Date', y='Close', data=etf.cleaned_df, label='Close', color='blue', linewidth=2)
sns.lineplot(x='Date', y='20-Day MA', data=etf.cleaned_df, label='20-Day MA', color='orange', linest
sns.lineplot(x='Date', y='50-Day MA', data=etf.cleaned_df, label='50-Day MA', color='green', linesty
sns.lineplot(x='Date', y='200-Day MA', data=etf.cleaned_df, label='200-Day MA', color='red', linesty
plt.title('Close Price with Moving Averages', fontsize=18, fontweight='bold')
plt.xlabel('Date', fontsize=14)
plt.ylabel('Price', fontsize=14)
plt.legend(title='Legend', title_fontsize='13', fontsize='12')
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.savefig(f"./Plots/ETFs/{etf.name}/Moving Average.png")
```

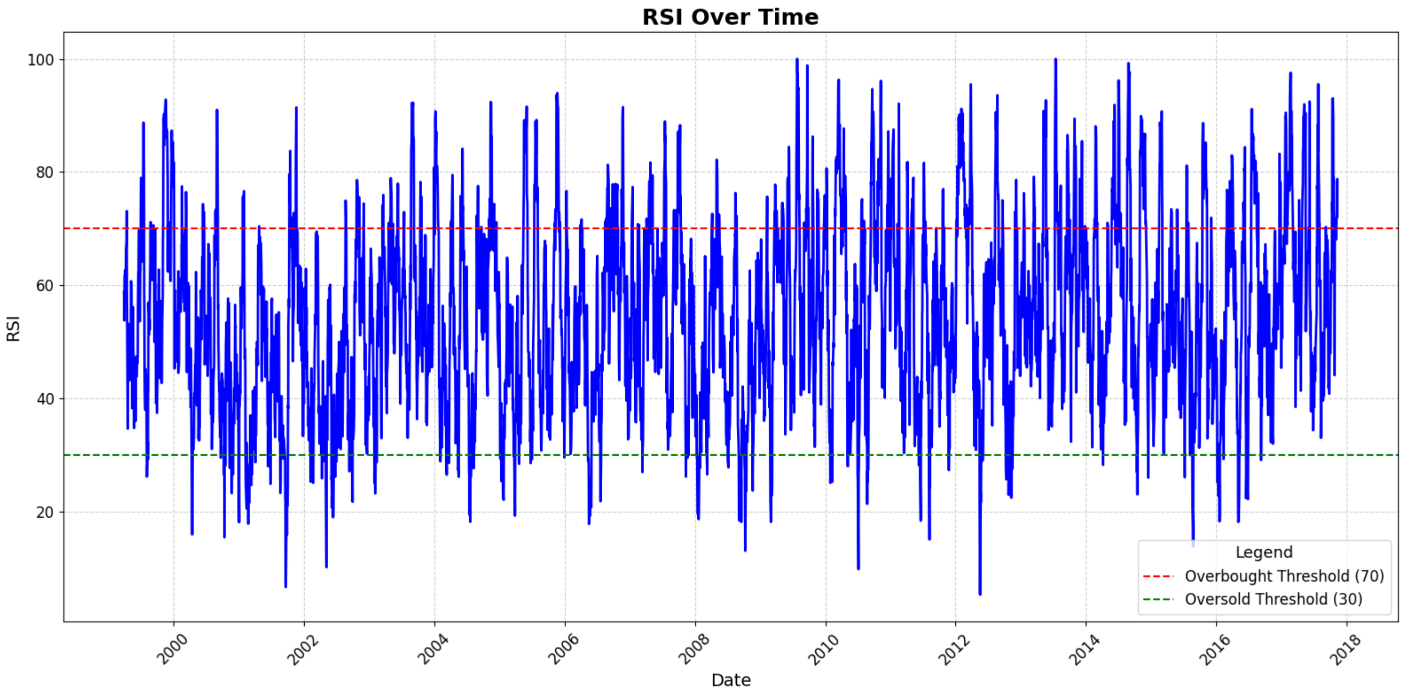


In [14]:
```
plt.figure(figsize=(16, 8))
sns.lineplot(x='Date', y='Close', data=etf.cleaned_df, label='Close', color='blue', linewidth=2)
sns.lineplot(x='Date', y='Middle Band', data=etf.cleaned_df, label='Middle Band', color='green', line
sns.lineplot(x='Date', y='Upper Band', data=etf.cleaned_df, label='Upper Band', color='red', linesty
sns.lineplot(x='Date', y='Lower Band', data=etf.cleaned_df, label='Lower Band', color='orange', line
plt.fill_between(etf.cleaned_df['Date'], etf.cleaned_df['Lower Band'], etf.cleaned_df['Upper Band'],
plt.title('Bollinger Bands', fontsize=18, fontweight='bold')
plt.xlabel('Date', fontsize=14)
plt.ylabel('Price', fontsize=14)
plt.legend(title='Legend', title_fontsize='13', fontsize='12')
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.savefig(f"./Plots/ETFs/{etf.name}/Bollinger Bands.png")
```

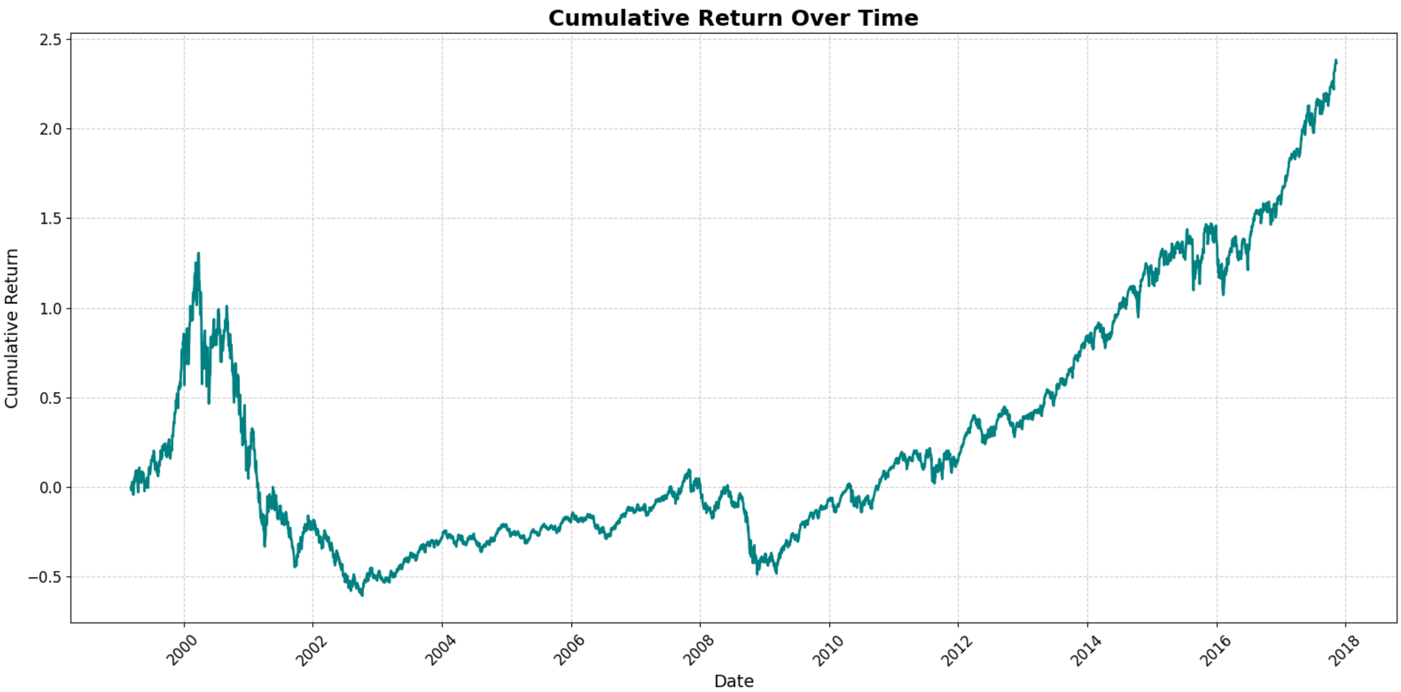


In [15]:
```
plt.figure(figsize=(16, 8))
sns.lineplot(x='Date', y='RSI', data=etf.cleaned_df, color='blue', linewidth=2)
plt.axhline(70, linestyle='--', color='red', linewidth=1.5, label='Overbought Threshold (70)')
plt.axhline(30, linestyle='--', color='green', linewidth=1.5, label='Oversold Threshold (30)')
plt.title('RSI Over Time', fontsize=18, fontweight='bold')
plt.xlabel('Date', fontsize=14)
plt.ylabel('RSI', fontsize=14)
plt.legend(title='Legend', title_fontsize='13', fontsize='12')
```

```
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.savefig(f"./Plots/ETFs/{etf.name}/RSI.png")
```

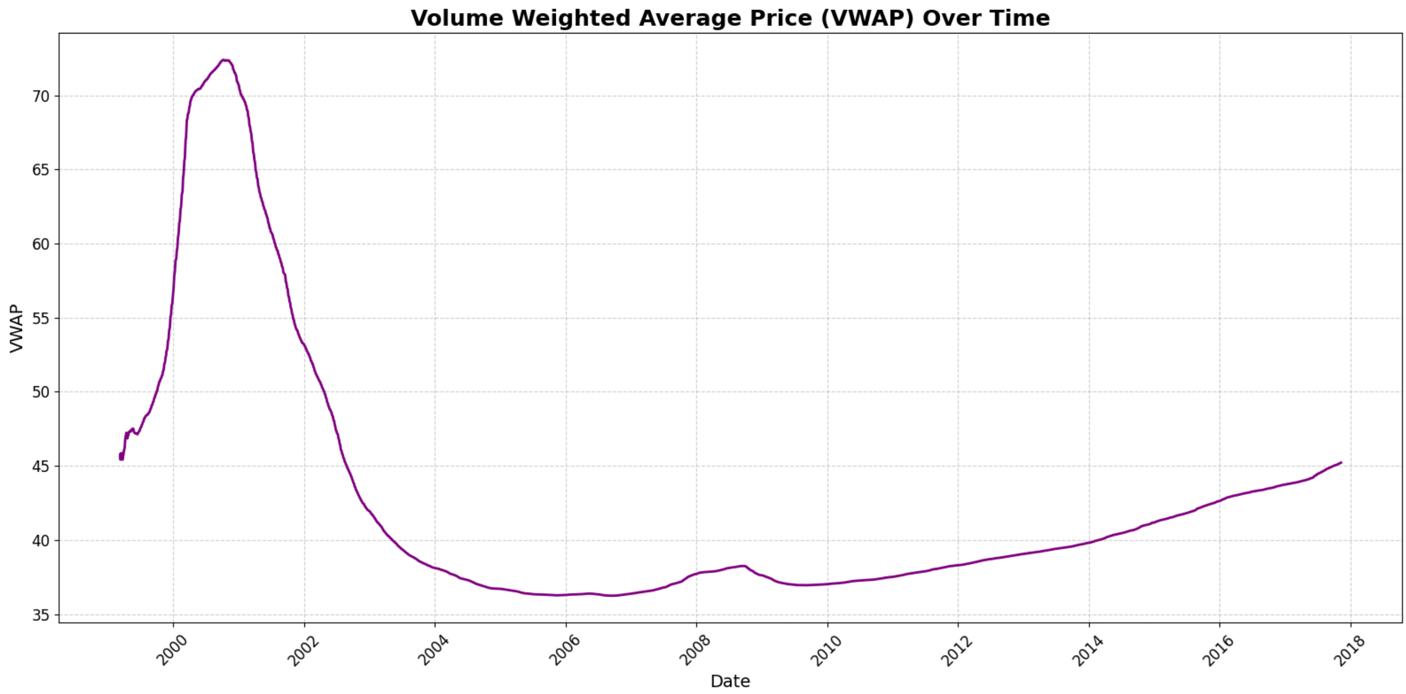


In [16]:
```
plt.figure(figsize=(16, 8))
sns.lineplot(x='Date', y='Cumulative Return', data=etf.cleaned_df, color='teal', linewidth=2)
plt.title('Cumulative Return Over Time', fontsize=18, fontweight='bold')
plt.xlabel('Date', fontsize=14)
plt.ylabel('Cumulative Return', fontsize=14)
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.savefig(f"./Plots/ETFs/{etf.name}/Cumulative Returns.png")
```

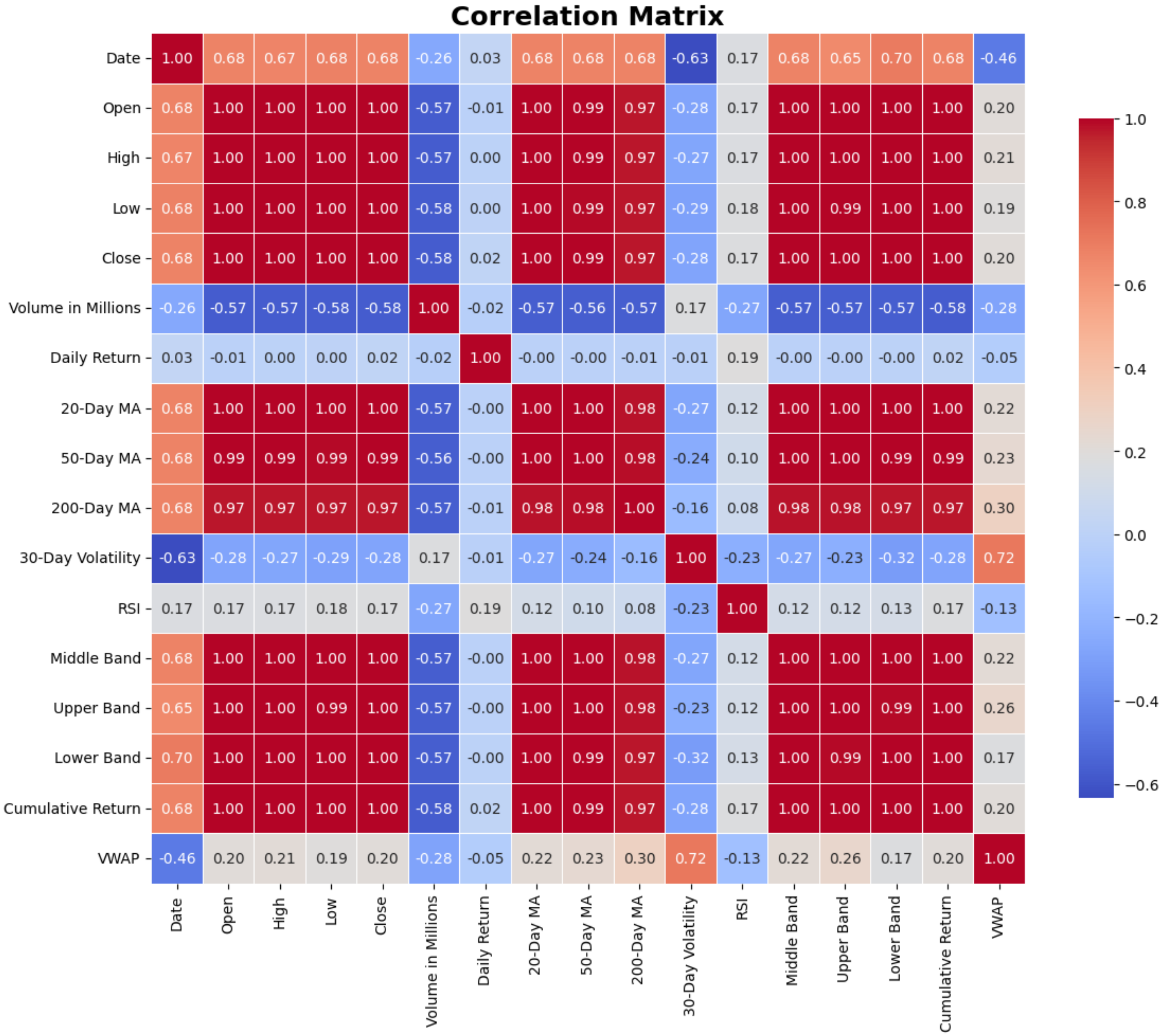


In [17]:
```
plt.figure(figsize=(16, 8))
sns.lineplot(x='Date', y='VWAP', data=etf.cleaned_df, color='purple', linewidth=2)
plt.title('Volume Weighted Average Price (VWAP) Over Time', fontsize=18, fontweight='bold')
plt.xlabel('Date', fontsize=14)
plt.ylabel('VWAP', fontsize=14)
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.grid(True, linestyle='--', alpha=0.6)
```

```
plt.tight_layout()
plt.savefig(f"./Plots/ETFs/{etf.name}/VWAP.png")
```

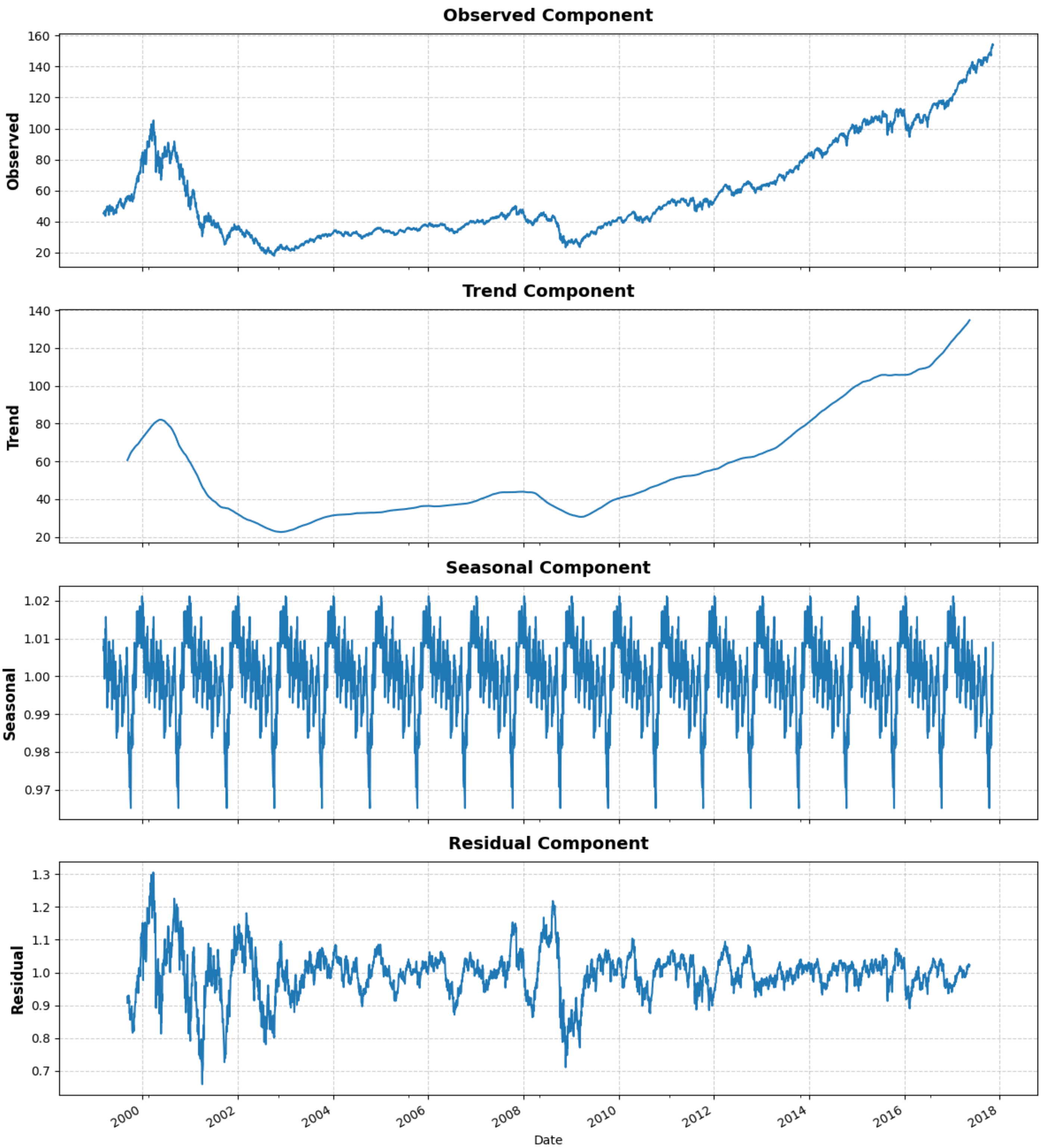


In [18]:
```
plt.figure(figsize=(12, 10))
corr_matrix = etf.cleaned_df.corr()
sns.heatmap(corr_matrix,annot=True, fmt=".2f", cmap="coolwarm",linewidths=0.5,linecolor='white',cbar
plt.title(f'Correlation Matrix', fontsize=18, fontweight='bold')
plt.tight_layout()
plt.savefig(f"./Plots/ETFs/{etf.name}/CorrelationMatrix.png", dpi=300)
```

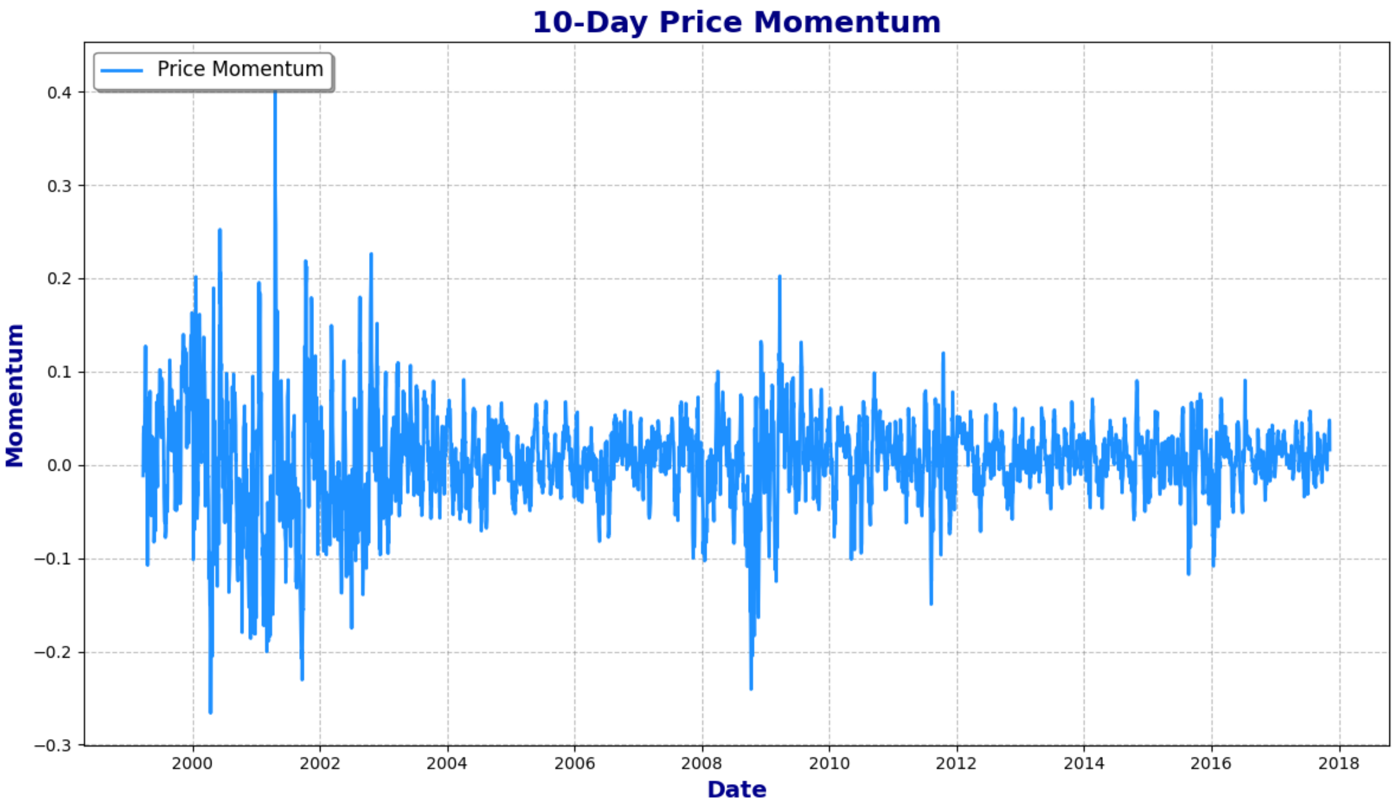


In [19]:
```
df = etf.cleaned_df.set_index('Date')
decomposition = seasonal_decompose(df['Close'], model="multiplicative", period=252)
# Assuming 252 trading days in a year
fig, axes = plt.subplots(4, 1, figsize=(12, 14), sharex=True)
components = {"Observed": decomposition.observed,"Trend": decomposition.trend,"Seasonal": decomposit
for ax, (label, data) in zip(axes, components.items()):
    data.plot(ax=ax, color='tab:blue')
    ax.set_ylabel(label, fontsize=12, fontweight='bold')
    ax.grid(True, linestyle='--', alpha=0.6)
    ax.set_title(f'{label} Component', fontsize=14, fontweight='bold', pad=10)
plt.suptitle(f'Seasonal Decomposition', fontsize=18, fontweight='bold', y=1.02)
plt.tight_layout(rect=[0, 0, 1, 0.97])
plt.savefig(f"./Plots/ETFs/{etf.name}/SeasonalDecomposition.png", dpi=300)
```

Seasonal Decomposition



In [20]:
```
df = etf.cleaned_df.copy()
df['Price Momentum'] = df['Close'].pct_change(periods=10)
plt.figure(figsize=(12, 7))
plt.plot(df["Date"], df["Price Momentum"], color='dodgerblue', label="Price Momentum", linewidth=2)
plt.title(f'10-Day Price Momentum', fontsize=18, fontweight='bold', color='navy')
plt.xlabel('Date', fontsize=14, fontweight='bold', color='darkblue')
plt.ylabel('Momentum', fontsize=14, fontweight='bold', color='darkblue')
plt.grid(True, linestyle='--', alpha=0.5, color='gray')
plt.legend(loc='upper left', fontsize=12, frameon=True, shadow=True, facecolor='white', edgecolor='g
plt.tight_layout()
plt.savefig(f"./Plots/ETFs/{etf.name}/Momentum10Days.png", dpi=300)
```



Preprocessing

With the dataset now cleaned, visualized, and expanded, we are set to implement machine learning models. Our first step involves preprocessing the dataset by splitting the **Date** feature into Day, Month, and Year. Subsequently, we will partition the dataset into training and testing subsets.

Next, we will employ a **Linear Regression** model on the training set and evaluate its performance on the testing set. To quantify the model's accuracy, we will compute the Mean Squared Error (MSE) and the R-squared (R^2) score. These metrics will provide insights into the model's prediction error and its explanatory power, respectively.

We will also drop **NaN Values** as Linear Regression Model can not handel those.

The procedure will be as follows:

1. **Feature Engineering:** Decompose the **Date** column into Day, Month, and Year components.
2. **Data Splitting:** Divide the dataset into training and testing sets.
3. **Model Training:** Fit a Linear Regression model to the training data.
4. **Model Evaluation:** Assess the model using the testing data by calculating MSE and R^2 score.

In [21]:
```
etf.cleaned_df = etf.ml_preprocess(etf.cleaned_df)
etf.describe(etf.cleaned_df)
```

```
DataFrame Information:
<class 'pandas.core.frame.DataFrame'>
Index: 4502 entries, 199 to 4700
Data columns (total 19 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   Open                 4502 non-null   float64
 1   High                 4502 non-null   float64
 2   Low                  4502 non-null   float64
 3   Close                4502 non-null   float64
 4   Volume in Millions   4502 non-null   float64
 5   Daily Return         4502 non-null   float64
 6   20-Day MA            4502 non-null   float64
 7   50-Day MA            4502 non-null   float64
 8   200-Day MA           4502 non-null   float64
 9   30-Day Volatility    4502 non-null   float64
 10  RSI                  4502 non-null   float64
 11  Middle Band          4502 non-null   float64
 12  Upper Band           4502 non-null   float64
 13  Lower Band           4502 non-null   float64
 14  Cumulative Return    4502 non-null   float64
 15  VWAP                 4502 non-null   float64
 16  Day                  4502 non-null   int32  
 17  Month                4502 non-null   int32  
```

```
 18  Year                4502 non-null   int32
dtypes: float64(16), int32(3)
memory usage: 650.7 KB

Descriptive Statistics:
              Open         High          Low        Close  Volume in Millions  \
count  4502.000000  4502.000000  4502.000000  4502.000000         4502.000000
mean     58.606703    59.089688    58.056943    58.594391            8.275376
std      31.839785    31.947222    31.694678    31.847566            5.929846
min      17.830000    18.361000    17.665000    17.938000            0.582839
25%      34.510000    34.876750    34.256750    34.513250            3.632026
50%      44.361500    44.720500    43.934500    44.369500            7.368226
75%      81.487500    82.583000    80.205000    81.407250           10.926713
max     153.810000   154.540000   153.620000   154.510000           67.553702

       Daily Return    20-Day MA    50-Day MA   200-Day MA  30-Day Volatility  \
count   4502.000000  4502.000000  4502.000000  4502.000000        4502.000000
mean      -0.013509    58.427293    58.151989    56.701543           1.285818
std        1.557934    31.522646    31.039578    28.688344           0.898399
min       -9.524849    19.021850    20.300720    22.002960           0.227525
25%       -0.612941    34.492638    34.456145    34.392620           0.714585
50%        0.057823    44.349575    43.911110    44.061610           0.939432
75%        0.637668    81.167200    80.710345    74.404424           1.534756
max       19.639184   150.777000   147.710600   138.945150           5.589283

               RSI  Middle Band   Upper Band   Lower Band  Cumulative Return  \
count  4502.000000  4502.000000  4502.000000  4502.000000        4502.000000
mean     54.397285    58.427293    60.996196    55.858390           0.283136
std      16.706092    31.522646    32.322261    30.837410           0.697417
min       5.335683    19.021850    20.337688    16.993830          -0.607183
25%      42.022807    34.492638    36.001637    32.851924          -0.244208
50%      53.638027    44.349575    46.265796    42.226362          -0.028370
75%      66.110475    81.167200    85.801051    75.838628           0.782706
max     100.000000   150.777000   155.897794   145.656206           2.383554

              VWAP          Day        Month         Year
count  4502.000000  4502.000000  4502.000000  4502.000000
mean     42.368591     2.022434     6.514660  2008.426921
std       8.929648     1.399427     3.415418     5.163134
min      36.248469     0.000000     1.000000  1999.000000
25%      37.243369     1.000000     4.000000  2004.000000
50%      38.797308     2.000000     7.000000  2008.000000
75%      43.019616     3.000000     9.000000  2013.000000
max      72.384166     4.000000    12.000000  2017.000000

First 5 Rows:
       Open    High     Low   Close  Volume in Millions  Daily Return  \
199  79.762  79.875  78.196  79.731            4.990187     -0.038866
200  80.366  81.155  79.591  80.684            3.923473      0.395690
201  80.762  80.789  78.605  80.203            4.312556     -0.692157
202  80.264  80.932  79.203  79.702            3.432112     -0.700189
203  80.481  82.718  80.371  82.718            3.693078      2.779538

     20-Day MA  50-Day MA  200-Day MA  30-Day Volatility        RSI  \
199   71.82440   64.41102   53.812815           1.872958  84.112577
200   72.39580   64.89468   53.987910           1.845120  83.558589
201   72.93240   65.40214   54.159525           1.844401  80.080662
202   73.50145   65.89276   54.334185           1.849869  77.428949
203   74.35075   66.47628   54.517515           1.906968  85.248843

     Middle Band  Upper Band  Lower Band  Cumulative Return       VWAP  Day  \
199     71.82440   79.174644   64.474156           0.745998  55.630711    2
200     72.39580   80.629106   64.162494           0.766867  55.790074    3
201     72.93240   81.741832   64.122968           0.756334  55.959579    0
202     73.50145   82.524404   64.478496           0.745363  56.090052    1
203     74.35075   83.491397   65.210103           0.811409  56.246583    2

     Month  Year
199     12  1999
200     12  1999
201     12  1999
202     12  1999
203     12  1999

Last 5 Rows:
        Open     High     Low   Close  Volume in Millions  Daily Return  \
4696  153.13  153.850  153.10  153.75            2.868585      0.404885
```

```
4697  153.67  154.082  153.34  153.87           2.128547      0.130149
4698  153.81  154.540  153.62  154.51           1.732650      0.455107
4699  153.26  153.770  152.11  153.69           4.055495      0.280569
4700  153.36  153.800  153.06  153.68           2.013811      0.208659

      20-Day MA  50-Day MA  200-Day MA  30-Day Volatility        RSI  \
4696   149.5770   146.8950   138.35380           0.396822  71.705069
4697   149.8905   147.1304   138.50660           0.391770  73.215941
4698   150.2140   147.3674   138.65640           0.394058  78.492647
4699   150.5100   147.5546   138.80140           0.395071  72.035398
4700   150.7770   147.7106   138.94515           0.388193  78.723404

      Middle Band  Upper Band  Lower Band  Cumulative Return       VWAP  Day  \
4696     149.5770  153.878961  145.275039           2.366911  45.190377    0
4697     149.8905  154.489431  145.291569           2.369539  45.196487    1
4698     150.2140  155.161848  145.266152           2.383554  45.201490    2
4699     150.5100  155.549691  145.470309           2.365597  45.213111    3
4700     150.7770  155.897794  145.656206           2.365378  45.218880    4

      Month  Year
4696     11  2017
4697     11  2017
4698     11  2017
4699     11  2017
4700     11  2017

DataFrame Shape (rows, columns):
(4502, 19)

Random Sample of 5 Rows:
        Open    High     Low   Close  Volume in Millions  Daily Return  \
2582  33.037  33.176  32.560  32.587           12.059929     -1.362109
3848  90.207  90.216  89.503  90.197            2.335206     -0.011086
1112  27.703  27.784  27.408  27.603            6.482848     -0.360972
359   82.105  82.481  80.037  80.315            1.714920     -2.180135
3341  59.971  60.546  59.925  60.292            3.886112      0.535259

      20-Day MA  50-Day MA  200-Day MA  30-Day Volatility        RSI  \
2582   32.70185   31.46836   29.654585           1.406337  61.876607
3848   89.15315   86.44210   82.764670           0.439482  63.142958
1112   27.80340   27.62958   24.434055           1.340076  46.309771
359    83.80705   83.97742   81.618805           2.416412  37.753897
3341   58.36705   59.96374   56.395275           1.112892  60.966458

      Middle Band  Upper Band  Lower Band  Cumulative Return       VWAP  Day  \
2582     32.70185   34.805396   30.598304          -0.286390  37.008902    1
3848     89.15315   90.548720   87.757580           0.975189  40.447288    3
1112     27.80340   28.851293   26.755507          -0.395533  38.975717    2
359      83.80705   92.173720   75.440380           0.758787  71.610083    3
3341     58.36705   60.049871   56.684229           0.320311  38.695665    1

      Month  Year
2582      6  2009
3848      6  2014
1112      8  2003
359       8  2000
3341      6  2012

Missing Values Count per Column:
Open                  0
High                  0
Low                   0
Close                 0
Volume in Millions    0
Daily Return          0
20-Day MA             0
50-Day MA             0
200-Day MA            0
30-Day Volatility     0
RSI                   0
Middle Band           0
Upper Band            0
Lower Band            0
Cumulative Return     0
VWAP                  0
Day                   0
Month                 0
```

```
Year                  0
dtype: int64
```

Linear Regression

It is a fundamental machine learning algorithm used for predicting a continuous target variable based on one or more input features. It assumes a linear relationship between the dependent variable (target) and the independent variables (features). The goal of Linear Regression is to fit a straight line (or hyperplane in the case of multiple features) that best captures this relationship.

For a simple Linear Regression model with one feature, the relationship between the target variable $ y $ and the feature $ x $ can be expressed as:

$$ y = \beta_0 + \beta_1 x + \epsilon $$
Where:

- $ y $ is the target variable.
- $ x $ is the feature.
- $ \beta_0 $ is the intercept of the regression line (the value of $ y $ when $ x = 0 $).
- $ \beta_1 $ is the slope of the regression line (how much $ y $ changes for a unit change in $ x $).
- $ \epsilon $ represents the error term or residuals (the difference between the actual and predicted values of $ y $).

In the case of multiple features, the model generalizes to:

$$ y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n + \epsilon $$
Where $ n $ is the number of features.

How Does Linear Regression Work?

1. **Model Fitting:**
   - The model is trained by finding the values of $ \beta_0 $, $ \beta_1 $, ..., $ \beta_n $ that minimize the error term $ \epsilon $.
   - This is typically done using the **Ordinary Least Squares (OLS)** method, which minimizes the sum of squared residuals: $$ \text{SSE} = \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 $$ Where $ y_i $ is the actual value and $ \hat{y}_i $ is the predicted value.
2. **Prediction:**
   - Once the model is trained, it can predict the target variable $ y $ for new data points by plugging the feature values into the learned linear equation.
3. **Evaluation:**
   - The performance of the model is typically evaluated using metrics like **Mean Squared Error (MSE)** and **R-squared (R^2)**.
   - **MSE** measures the average squared difference between actual and predicted values: $$ \text{MSE} = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 $$
   - **R^2 Score** represents the proportion of the variance in the dependent variable that is predictable from the independent variables. It is calculated as: $$ R^2 = 1 - \frac{\text{SS}_{\text{res}}}{\text{SS}_{\text{tot}}} $$ Where $ \text{SS}_{\text{res}} $ is the sum of squares of residuals and $ \text{SS}_{\text{tot}} $ is the total sum of squares (variance of the target).

Key Assumptions

For Linear Regression to provide reliable results, several assumptions need to be met:

1. **Linearity:** The relationship between the independent and dependent variables is linear.
2. **Independence:** The residuals (errors) are independent.
3. **Homoscedasticity:** The residuals have constant variance at every level of the independent variable.
4. **Normality:** The residuals of the model are normally distributed.

Limitations

- **Sensitivity to Outliers:** Linear Regression is highly sensitive to outliers, which can significantly affect the model's performance.
- **Limited to Linear Relationships:** It cannot capture non-linear relationships unless features are transformed or additional polynomial terms are added.
- **Assumptions Need to be Met:** The model relies on the assumptions mentioned earlier, and if these are violated, the results may be unreliable.

In [22]:
```
features = list(etf.cleaned_df.columns)
target = features.pop(features.index("Close"))
etf.linearModel(etf.cleaned_df, features, target)
```

Mean Squared Error (MSE)

- **MSE** helps us understand the average squared error between the predicted and actual values, with lower values indicating better model performance.
- **Range:** MSE is a non-negative value that can range from 0 to infinity.
- **Interpretation:**
  - **0:** Indicates a perfect model with no error; the predicted values exactly match the actual values.
  - **>0:** The model has some error. The larger the MSE, the worse the model's performance.
  - **Infinity:** Represents a model with very poor performance, where the predictions are far from the actual values.

In [23]:
```
print(etf.mse)
```

```
6.108681034034072e-27
```

R-squared (R²)

- **R²** indicates the proportion of variance in the target variable that is explained by the model, with higher values indicating better model performance.
- **Range:** R² ranges from -∞ to 1.
- **Interpretation:**
  - **1:** Indicates a perfect fit; the model explains 100% of the variance in the data.
  - **0:** The model does not explain any of the variance in the data; it performs as well as a model that always predicts the mean value.
  - **< 0:** The model performs worse than simply predicting the mean of the target variable. This could happen if the predictions are completely off, leading to a higher error than the variance of the actual data.

In [24]:
```
print(etf.r2Score)
```

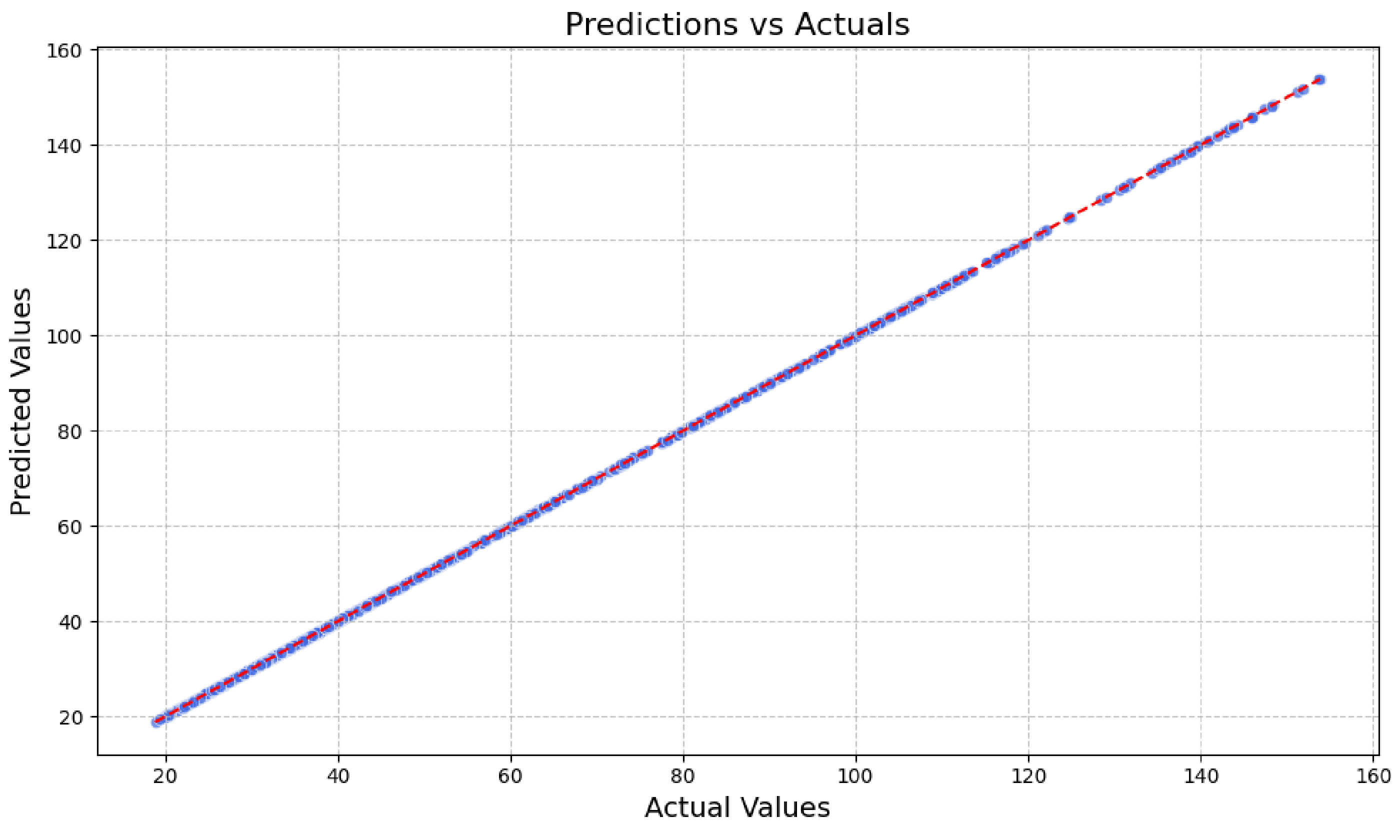
```
1.0
```

Interpreting the Linear Regression Results

- **Mean Squared Error (MSE): 1.6097371305360009e-26**
  - The MSE value is extremely close to 0, which indicates that the model's predictions are almost identical to the actual values. In linear regression, a lower MSE indicates better performance, and in this case, the MSE is so small that it suggests near-perfect predictions.
- **R² (R-squared): 1.0**
  - An R² value of 1.0 signifies that the model explains 100% of the variance in the target variable. This means that the model has perfectly fitted the data, capturing all the variability in the target variable.

These results suggest that the linear regression model has achieved an ideal fit to the data. However, such a perfect fit is uncommon in practice and might indicate overfitting, where the model is too closely tailored to the training data and might not perform as well on unseen data. It's important to validate the model's performance on a separate test dataset to ensure it generalizes well.

In [25]:
```
etf.plotPredictions()
```


Predictions vs Actuals
Predicted Values
Actual Values
160
140
120
100
80
60
40
20
20
40
60
80
100
120
140
160

Decision Tree Regression

Decision Tree Regression is a non-parametric supervised learning algorithm used for predicting a continuous target variable. It models the relationship between input features and the target variable by recursively splitting the data into subsets based on feature values. The goal is to create regions that are as homogeneous as possible in terms of the target variable.

How Does Decision Tree Regression Work?

1. **Splitting the Data:**
    - The dataset is split into smaller subsets based on feature values. The best split is determined by minimizing a loss function, usually the Mean Squared Error (MSE): $$ \text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 $$ Where $ y_i $ is the actual value, $ \hat{y}_i $ is the predicted value, and $ n $ is the number of data points in the subset.
2. **Building the Tree:**
    - The process of splitting continues recursively, forming a tree structure. Each internal node represents a decision based on a feature, each branch represents the outcome of the decision, and each leaf node represents the predicted value (the mean of the target values in that region).
3. **Prediction:**
    - For a new data point, the model traverses the tree from the root to a leaf node by making decisions at each node. The prediction is the value at the leaf node where the data point lands.

Evaluation:

- The performance of the model can be evaluated using metrics like **Mean Squared Error (MSE)**, **Mean Absolute Error (MAE)**, and **R-squared (R^2)**:
    - **MSE:** Measures the average squared difference between actual and predicted values.
    - **MAE:** Measures the average absolute difference between actual and predicted values: $$ \text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| $$
    - **R^2 Score:** Represents the proportion of the variance in the dependent variable that is predictable from the independent variables.

Advantages:

- **Interpretability:** The resulting tree model is easy to understand and interpret, with clear decision rules.
- **Non-linearity:** Decision trees can capture non-linear relationships between features and the target variable.
- **No Need for Feature Scaling:** Decision trees do not require normalization or standardization of features.

Limitations:

- **Overfitting:** Decision trees can easily overfit the training data, capturing noise instead of the underlying pattern.
- **Instability:** Small changes in the data can lead to significantly different tree structures.
- **Bias:** The model tends to favor features with more levels, which can introduce bias.

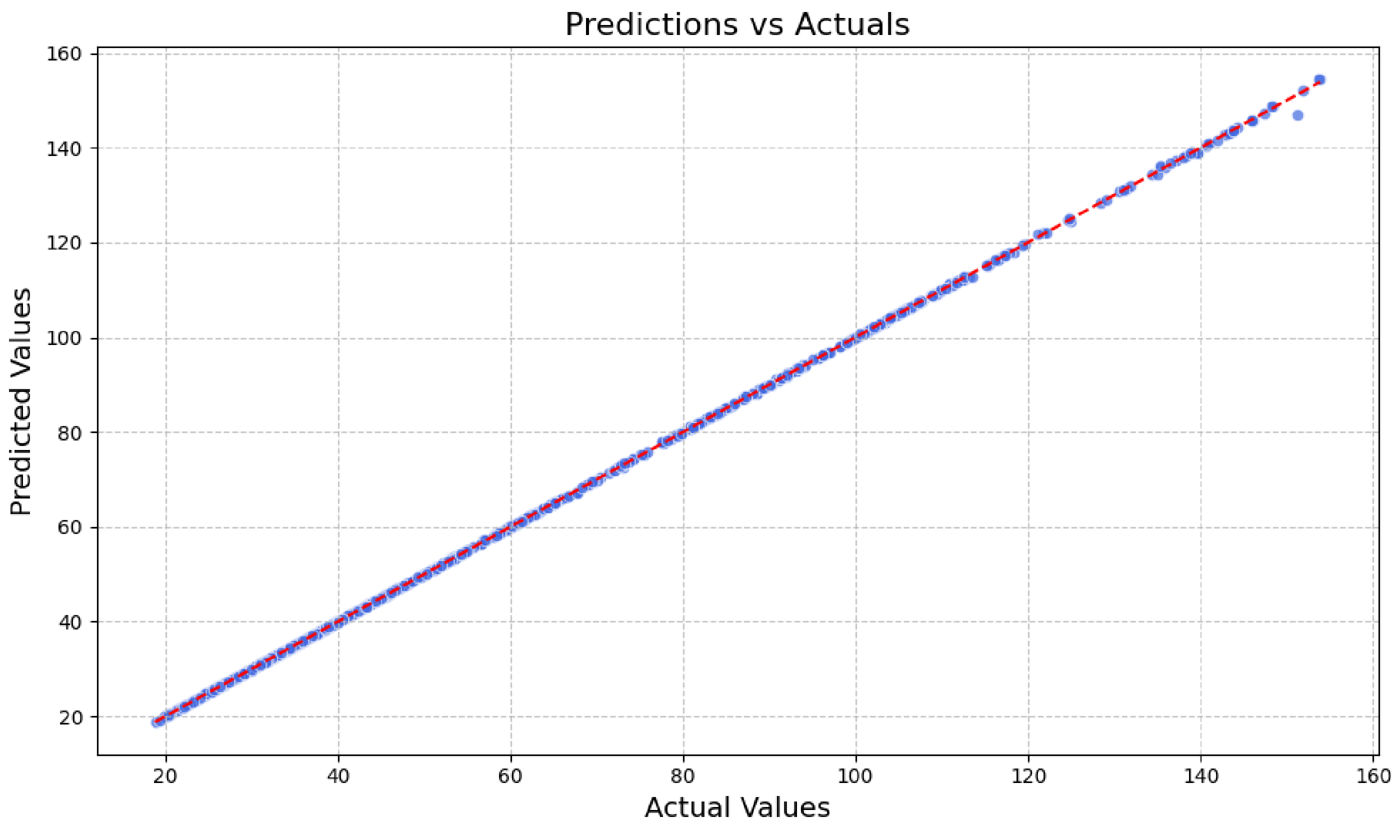
Pruning:

To address overfitting, **pruning** techniques can be applied to simplify the tree by removing branches that have little importance. Pruning can be done by setting a minimum number of samples required to split a node or by setting a maximum depth for the tree.

In [26]:
```
etf.dtr(etf.cleaned_df, features, target)
print(etf.mse)
print(etf.r2Score)
```

```
0.028423741563055063
0.9999708177352533
```

In [27]:
```
etf.plotPredictions()
```



Interpreting the Decision Tree Regression Results

- **Mean Squared Error (MSE): 0.028423741563055063**
  - The MSE value is very low, indicating that the model's predictions are highly accurate, with minimal difference between the predicted and actual values. In Decision Tree Regression, a lower MSE signifies better model performance, and this result suggests that the model has captured the underlying patterns in the data very well.
- **R^2 (R-squared): 0.9999708177352533**
  - An R^2 value of 0.99997 means that the model explains nearly 100% of the variance in the target variable. This indicates that the Decision Tree model has an excellent fit, capturing almost all the variability in the data.

These results demonstrate that the Decision Tree Regression model has achieved a nearly perfect fit to the data. While these metrics are highly favorable, it's essential to check for potential overfitting, where the model might perform exceptionally well on training data but less so on unseen data. Validating the model on a test dataset is crucial to ensure it generalizes effectively. Hence we check for potential underfitting or overfitting.

In [28]:
```
etf.dtrCheck()
```

```
Max leaf nodes: 5      Mean Absolute Error:  5.666741805353695
Max leaf nodes: 50      Mean Absolute Error:  0.5457968225965555
Max leaf nodes: 500      Mean Absolute Error:  0.0781685873918358
Max leaf nodes: 5000      Mean Absolute Error:  0.05973001776198934
```

Given the following results it would be best if we fit out model with 50 tree nodes.

In [29]:
```
etf.dtr(etf.cleaned_df, features, target, 50)
print(etf.mse)
print(etf.r2Score)
```

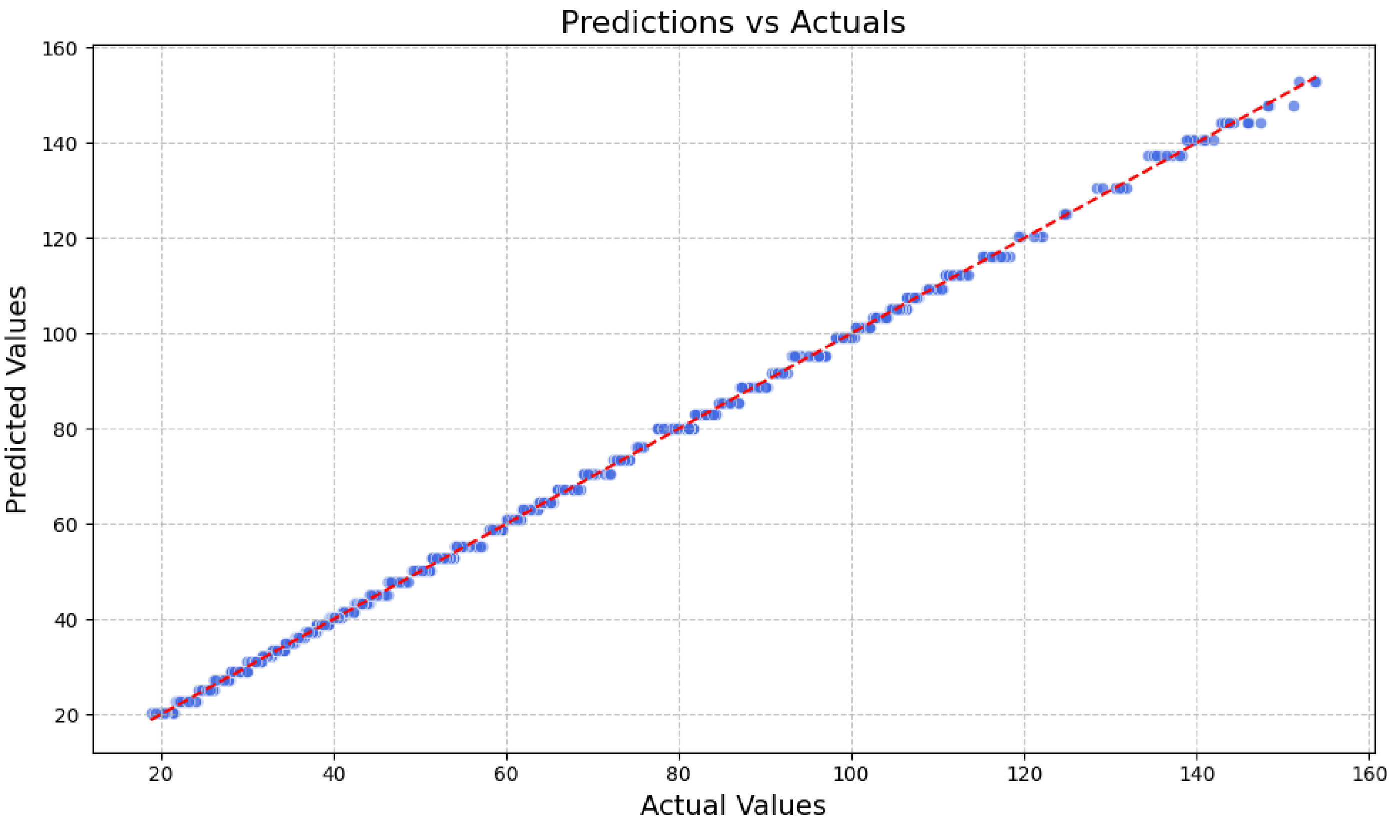
```
0.4867922285467166
0.9995002171104537
```

Interpreting the Updated Decision Tree Regression Results

- **Mean Squared Error (MSE): 0.4867922285467166**
  - The MSE value indicates the average squared difference between the predicted and actual values. Although higher than previous results, it still suggests that the model's predictions are relatively accurate. In Decision Tree Regression, a lower MSE is preferable, and this result shows that while the model performs well, there is a noticeable increase in error compared to the earlier evaluation.
- **R^2 (R-squared): 0.9995002171104537**
  - An R^2 value of 0.9995 means that the model explains approximately 99.95% of the variance in the target variable. This reflects that the model has a very good fit and captures most of the variability in the data. However, it is slightly lower than before, indicating a slight reduction in the model's ability to explain the variance.

These results suggest that the Decision Tree Regression model with 50 max leaf nodes provides a very good fit to the data but shows a slight increase in error compared to previous configurations. While the R^2 value remains high, reflecting strong performance, the higher MSE suggests that there might be some trade-off between model complexity and accuracy.

In [30]:
```
etf.plotPredictions()
```



Random Forest

Random Forest is a versatile ensemble learning method used for both classification and regression tasks. It constructs multiple decision trees during training and combines their predictions to produce a more accurate and robust model.

How Does Random Forest Work?

1. **Bootstrap Aggregation (Bagging)**:
   - Random Forest utilizes bagging by creating multiple subsets of the training data through sampling with replacement. Each subset is used to train an individual decision tree.
2. **Decision Trees**:
   - Each tree is built to its maximum depth without pruning. During training, each node is split based on the best feature chosen from a random subset of features, ensuring diversity among the trees.
3. **Aggregation**:
   - For regression tasks, the final prediction is the average of the predictions from all trees. For classification tasks, it is determined by the majority vote from all trees.

Benefits of Using Random Forest

1. **Handling Non-Linearity**:
   - Unlike linear regression, which assumes a linear relationship between features and the target, Random Forest can capture complex, non-linear relationships. This is particularly useful for financial data, where relationships are often non-linear.
2. **Robustness**:
   - Random Forest is less prone to overfitting compared to a single decision tree due to its ensemble approach. This enhances model robustness and generalization to new data.
3. **Feature Importance**:
   - Random Forest can assess feature importance, helping identify which variables (e.g., trading volume, economic indicators) significantly influence predictions, such as ETF prices.
4. **Enhanced Performance**:
   - Even with an excellent MSE of *1.6097371305360009e-26* and an R^2 of *1.0* using linear regression, Random Forest can potentially improve prediction accuracy and robustness by uncovering complex patterns that linear models might miss.

Key Assumptions

1. **Independence of Trees**:
   - Assumes that the decision trees within the forest are independent. By averaging their predictions, the model reduces variance and minimizes overfitting.
2. **Random Feature Selection**:
   - Each tree split is based on a random subset of features, promoting diversity among trees and capturing varied aspects of the data.
3. **Bagging Technique**:
   - Relies on the bagging approach, which assumes that combining predictions from multiple models trained on different data subsets will enhance overall model performance.

Limitations

1. **Complexity**:
   - The model's complexity increases with the number of trees and their depth, which can make it harder to interpret compared to simpler models like linear regression.
2. **Computational Cost**:
   - Training can be resource-intensive and time-consuming, especially with a large number of trees and features.
3. **Risk of Overfitting**:
   - While generally robust, using too many trees can still lead to overfitting, particularly on smaller datasets.
4. **Feature Importance Bias**:
   - The assessment of feature importance can be skewed towards features with more levels or continuous variables, which may not always represent their true significance.
5. **High-Dimensional Data**:
   - Performance may decline with very high-dimensional data if features are highly correlated or not informative.

In [31]:
```
etf.rfr(etf.cleaned_df, features, target)
```

```
print(etf.mse)
print(etf.r2Score)
```

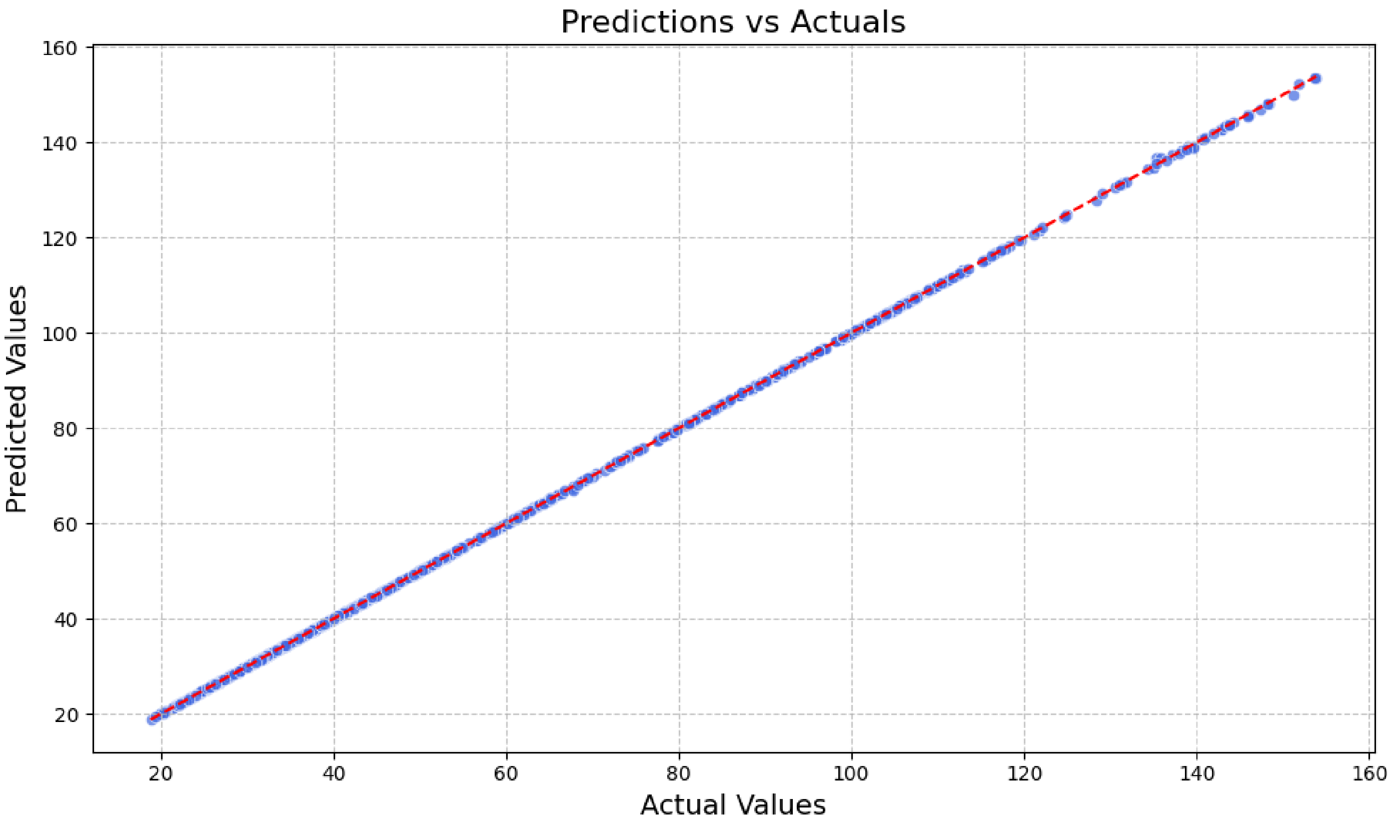
```
0.008947722943783822
0.9999908134958501
```

Interpreting the Random Forest Regression Results

- **Mean Squared Error (MSE): 0.008947722943783822**
  - The MSE value is very low, indicating that the model's predictions are very close to the actual values. In Random Forest Regression, a lower MSE signifies better model performance, and this result suggests that the model has captured the underlying patterns in the data with high accuracy.
- **R^2 (R-squared): 0.9999908134958501**
  - An R^2 value of 0.99999 means that the model explains approximately 99.999% of the variance in the target variable. This indicates an exceptional fit, where the model captures nearly all of the variability in the data.

These results demonstrate that the Random Forest Regression model has achieved an excellent fit to the data, with minimal error and nearly perfect explanatory power. Such high performance suggests that the model generalizes very well to the data.

In [32]:
```
etf.plotPredictions()
```



Neural Network

A Neural Network is a computational model inspired by the human brain, designed to recognize patterns and learn complex relationships between input features and a target variable. It consists of layers of interconnected neurons, where each neuron processes input data and passes the output to the next layer.

How Does a Neural Network Work?

1. **Structure:**
   - **Input Layer:** The input layer receives the raw data (features) and passes it to the hidden layers.
   - **Hidden Layers:** These layers process the inputs using weights and biases. The output of each neuron is determined by applying an activation function to the weighted sum of its inputs.
   - **Output Layer:** The output layer produces the final predictions based on the processed information from the hidden layers.
2. **Forward Propagation:**
   - During forward propagation, input data passes through the network layer by layer. Each neuron in the hidden layers computes a weighted sum of its inputs, applies an activation function, and sends the output to the next layer.
   - The final output layer produces predictions, which can be continuous (regression) or categorical (classification).
3. **Loss Function:**
   - The loss function quantifies the difference between the predicted output and the actual target values. Common loss functions include:
     - **Mean Squared Error (MSE)** for regression: $$ \text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 $$
     - **Cross-Entropy Loss** for classification: $$ \text{L} = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)\right] $$
4. **Backpropagation:**
   - Backpropagation is the process of minimizing the loss function by adjusting the weights and biases in the network. It calculates the gradient of the loss function with respect to each weight using the chain rule and updates the weights using an optimization algorithm like **Stochastic Gradient Descent (SGD)**.
5. **Training:**
   - The network undergoes multiple epochs (iterations) of forward propagation and backpropagation to minimize the loss function. Over time, the model learns to make more accurate predictions.

Evaluation:

- The performance of a neural network can be evaluated using metrics like **Mean Squared Error (MSE)** for regression or **Accuracy, Precision, Recall,** and **F1-Score** for classification:
  - **Accuracy:** The proportion of correctly predicted instances out of the total instances.
  - **Precision:** The ratio of true positive predictions to the total positive predictions.
  - **Recall:** The ratio of true positive predictions to the actual positive instances.
  - **F1-Score:** The harmonic mean of precision and recall.

Advantages:

- **Ability to Capture Complex Patterns:** Neural networks can model complex relationships in data, especially with non-linear activation functions.
- **Adaptability:** They can be applied to a wide range of problems, including image recognition, natural language processing, and time-series forecasting.
- **Scalability:** Neural networks can handle large amounts of data and benefit from parallel computation.

Limitations:

- **Computational Complexity:** Training neural networks requires significant computational resources, especially for deep networks with many layers.
- **Overfitting:** Neural networks can overfit to the training data, especially with small datasets.
- **Interpretability:** Neural networks are often considered "black boxes," making it difficult to interpret how the model makes decisions.

Regularization Techniques:

To prevent overfitting, regularization techniques like **Dropout** and **L2 Regularization** can be applied:

- **Dropout:** Randomly dropping a fraction of the neurons during training to prevent the network from becoming too reliant on specific neurons.
- **L2 Regularization:** Adding a penalty to the loss function proportional to the sum of the squared weights, encouraging smaller weights and reducing overfitting.

In [33]:
```
class Neural(nn.Module):
    def __init__(self, input_size):
        super(Neural, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
```

```
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

def neuralNetwork(df, features, target, epochs=100, lr=0.001):
    X = torch.tensor(df[features].values, dtype=torch.float32)
    Y = torch.tensor(df[target].values, dtype=torch.float32).unsqueeze(1)
    trainX, testX, trainY, testY = train_test_split(X, Y, random_state=0)
    model = Neural(trainX.shape[1])
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=1e-5)
    for epoch in range(epochs):
        model.train()
        optimizer.zero_grad()
        output = model(trainX)
        loss = criterion(output, trainY)
        loss.backward()
        optimizer.step()
        if (epoch+1) % 10 == 0:
            print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
    model.eval()
    with torch.no_grad():
        predictions = model(testX)
        mse = criterion(predictions, testY).item()
        print(f'MSE on test set: {mse:.4f}')

    return model, mse, trainX, trainY, testX, testY
```

In [34]:

```
count = 0
while True:
    count += 1
    model, mse, trainX, trainY, testX, testY = neuralNetwork(etf.cleaned_df, features, target)
    if mse < 100 or count == 50:
        break
```

```
Epoch [10/100], Loss: 1427.4164
Epoch [20/100], Loss: 1092.4950
Epoch [30/100], Loss: 954.4913
Epoch [40/100], Loss: 837.5951
Epoch [50/100], Loss: 673.4206
Epoch [60/100], Loss: 567.7768
Epoch [70/100], Loss: 442.8987
Epoch [80/100], Loss: 341.1231
Epoch [90/100], Loss: 320.9391
Epoch [100/100], Loss: 288.5771
MSE on test set: 705.5704
Epoch [10/100], Loss: 2499.2046
Epoch [20/100], Loss: 1655.8646
Epoch [30/100], Loss: 1167.8164
Epoch [40/100], Loss: 912.6354
Epoch [50/100], Loss: 776.8035
Epoch [60/100], Loss: 641.1437
Epoch [70/100], Loss: 520.6839
Epoch [80/100], Loss: 468.1089
Epoch [90/100], Loss: 400.3139
Epoch [100/100], Loss: 380.3502
MSE on test set: 749.4872
Epoch [10/100], Loss: 2479.0867
Epoch [20/100], Loss: 1499.9940
Epoch [30/100], Loss: 1199.8109
Epoch [40/100], Loss: 1019.5978
Epoch [50/100], Loss: 879.3671
Epoch [60/100], Loss: 754.0897
Epoch [70/100], Loss: 636.6065
Epoch [80/100], Loss: 512.4489
Epoch [90/100], Loss: 392.3849
Epoch [100/100], Loss: 355.5101
MSE on test set: 178.0190
Epoch [10/100], Loss: 2396.4697
Epoch [20/100], Loss: 2695.6389
Epoch [30/100], Loss: 1487.5640
Epoch [40/100], Loss: 1209.4747
Epoch [50/100], Loss: 1014.4758
Epoch [60/100], Loss: 781.6338
Epoch [70/100], Loss: 652.7566
Epoch [80/100], Loss: 559.3065
Epoch [90/100], Loss: 464.4584
Epoch [100/100], Loss: 404.8523
MSE on test set: 60.4610
```