Author: Yashraj Singh

Date: September 19, 2024

Topic: Price Forecasting using Time Series Model(ARIMA)

LinkedIn: www.linkedin.com/in/yashrajm320

The Importance of Time Series Analysis in Quantitative Finance

Introduction

In quantitative finance, time series analysis plays a pivotal role in understanding and forecasting financial data over time. Financial markets are inherently dynamic, with asset prices, interest rates, and volatility fluctuating continuously. Time series analysis enables analysts and traders to detect patterns, estimate future prices, and develop strategies based on the historical behavior of financial instruments.

By modeling the temporal dependence in financial data, time series analysis helps to capture trends, cycles, and volatility structures, which are crucial for pricing, risk management, and trading decisions. Common techniques used in time series analysis include **ARIMA (AutoRegressive Integrated Moving Average)**, **GARCH (Generalized Autoregressive Conditional Heteroskedasticity)**, and **Exponential Smoothing**.

This article explores the key concepts and mathematical foundations behind these techniques and provides a real-world example of predicting stock prices to inform trading decisions.

---

1. Key Techniques in Time Series Analysis

A. ARIMA (AutoRegressive Integrated Moving Average)

ARIMA is one of the most popular techniques for time series forecasting, especially for data that shows temporal autocorrelation. ARIMA models can capture different patterns, including trends, seasonalities, and noise, making it a versatile tool in finance.

The ARIMA model consists of three key components:

- **AR (AutoRegressive)**: This part models the dependency between an observation and a certain number of lagged observations (previous values of the time series).
- **I (Integrated)**: The "integrated" part refers to differencing the data to make the time series stationary (i.e., removing trends and seasonality).
- **MA (Moving Average)**: This part models the dependency between an observation and the residual errors from previous time steps.

The notation for ARIMA is typically expressed as **ARIMA(p, d, q)**, where:

- p: The number of lag terms (autoregressive component).

- d: The number of differencing steps needed to make the series stationary.
- q: The number of lagged forecast errors (moving average component).

Mathematical Intuition:

An ARIMA model can be written as:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_q \epsilon_{t-q} + \epsilon_t$$

Where:

- y_t is the value of the time series at time t.
- $(\phi_1, \phi_2, \ldots, \phi_p)$ are the autoregressive coefficients.
- $(\theta_1, \theta_2, \ldots, \theta_q)$ are the moving average coefficients.
- (ϵ_t) is the white noise (random error) at time t.

Finance Analogy:

Consider a stock price that shows a trend over time with small deviations due to random market factors. The **AR** component explains how past prices influence the current price, while the **MA** component captures how past errors (unexpected price movements) affect the forecast. The **I** component helps remove the trend, allowing us to forecast future stock prices by focusing on deviations from the historical pattern.

Example in Trading:

In stock trading, ARIMA can be used to predict short-term price movements. By training an ARIMA model on historical prices, traders can generate future price forecasts and develop strategies, such as entering buy/sell positions based on predicted price trends.

B. GARCH (Generalized Autoregressive Conditional Heteroskedasticity)

GARCH models are used to predict volatility in financial time series. Volatility is a crucial factor in options pricing, risk management, and trading strategies. In financial markets, volatility often shows **clustering**, meaning that high-volatility periods are followed by more volatile periods, and low-volatility periods are followed by more stable periods.

The GARCH model addresses this by modeling the variance (volatility) of the errors in a time series. It builds on the ARCH (Autoregressive Conditional Heteroskedasticity) model by introducing an autoregressive term for past variances.

A **GARCH(p, q)** model can be expressed as:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{p} \alpha_i \epsilon_{t-i}^2 + \sum_{j=1}^{q} \beta_j \sigma_{t-j}^2$$

Where:

- σ_t^2 is the conditional variance at time t.

- ϵ_{t-i}^2 represents the squared residuals from previous time steps (past shocks).
- α_0 is a constant term.
- α_i represents the impact of past squared shocks (ARCH effect).
- β_j represents the impact of past variances (GARCH effect).

Mathematical Intuition:

In GARCH, the **conditional variance** σ_t^2 is modeled as a function of past variances and past squared errors (shocks). The model captures the tendency for volatility clustering, meaning that large movements in stock prices (both up and down) are likely to be followed by more large movements.

Finance Analogy:

Think of the stock market as a calm or stormy sea. During volatile periods (e.g., during a financial crisis), the sea is rough, and large waves follow each other (high volatility clustering). During calm periods, the waves are small and consistent (low volatility). GARCH models this relationship and helps forecast when the market might turn rough again (increase in volatility).

Example in Risk Management:

GARCH is frequently used in risk management to forecast future volatility, which is a key component in Value at Risk (VaR) calculations. Traders can adjust their portfolios by increasing or decreasing exposure based on expected volatility forecasts from GARCH models.

C. Exponential Smoothing (ETS)

Exponential Smoothing is another important technique in time series forecasting, especially for data with a trend or seasonality. The idea is to assign exponentially decreasing weights to past observations, so recent data points have more influence on the forecast.

There are three types of Exponential Smoothing:

1. **Simple Exponential Smoothing (SES)**: Suitable for data without trends or seasonality.
2. **Holt's Linear Trend Model**: Used for data with a trend.
3. **Holt-Winters Seasonal Model**: Used for data with both a trend and seasonality.

The general formula for Exponential Smoothing is:

$$S_t = \alpha\, y_t + (1-\alpha)\, S_{t-1}$$

Where:

- S_t is the smoothed value at time t.
- y_t is the actual value at time t.
- α is the smoothing factor, $(0<\alpha<1)$.

Mathematical Intuition:

Exponential Smoothing takes a weighted average of past observations, with more recent data points receiving higher weights. The smoothing factor α determines how much weight is given to recent observations versus older ones.

Finance Analogy:

Imagine you're trying to forecast the demand for a product by looking at past sales. In the financial context, **Exponential Smoothing** can be used to forecast stock prices by giving more importance to recent price changes while still accounting for past prices, albeit with lower importance. This method is particularly effective for trend-following strategies.

Example in Forecasting:

Traders can use Exponential Smoothing to predict future stock prices, taking into account recent price trends while smoothing out short-term fluctuations. This is especially useful for short-term trading strategies where the recent price action is more informative than older data.

---

2. Example: Predicting Stock Prices Using ARIMA

Let's now walk through a practical example of using **ARIMA** to predict stock prices and inform trading decisions.

Building a Time Series Model for Nifty Prices

Notebook Overview

This document serves as a guide to developing a robust time series model for forecasting Nifty prices. It is designed to walk you through each step, from data collection to performance evaluation, integrating theoretical knowledge with practical application.

1. Reading Price Data

- **Data Acquisition**: Fetch Nifty price data using the `yfinance` API, renowned for its accessibility and reliability.
- **Resampling**: Obtain weekly frequency data to focus on longer-term trends and reduce noise.

2. Data Sanity Check

- **Quality Assessment**: Perform checks for missing values and outliers to ensure data integrity.
- **Data Cleaning**: Implement strategies to handle anomalies, such as interpolation for missing values or removal of erroneous data points, ensuring robustness in the modeling process.

3. Selecting the Right Model and Forecasting Prices

- **Preliminary Analysis**: Examine the stationarity of the time series, applying differencing if necessary to meet model assumptions.
- **Model Determination**:
  - Assess the Autoregressive (AR) order to gauge the influence of past values on current predictions.
  - Evaluate the Moving Average (MA) order to understand the impact of past forecast errors on future values.
- **Model Selection**: Use the Akaike Information Criterion (AIC) to identify the most efficient model in terms of informational balance and complexity.
- **Implementation**: Develop a function for model selection and forecasting, employing a sliding window technique to update predictions dynamically.

4. Trading Strategy Using Predicted Prices

- **Strategy Design**: Formulate a trading strategy based on model predictions, detailing specific buy and sell signals.
- **Execution Plan**: Outline the operational aspects of executing trades based on predictive insights, considering transaction costs and timing.

5. Performance Analysis

- **Model Evaluation**: Analyze the accuracy of the model's predictions by comparing them against actual market movements.
- **Strategy Testing**: Assess the strategy's performance using backtesting, comparing it against a benchmark 'buy and hold' strategy to evaluate relative effectiveness.

Summary

This notebook is structured to facilitate a deep understanding of time series analysis applied to financial markets, with a specific focus on Nifty prices. By the end, you should be able to not only develop and test a forecasting model but also understand how to translate these forecasts into actionable trading strategies.

Import modules

```
!pip install pmdarima
```

```
Requirement already satisfied: pmdarima in
/usr/local/lib/python3.10/dist-packages (2.0.4)
Requirement already satisfied: joblib>=0.11 in
/usr/local/lib/python3.10/dist-packages (from pmdarima) (1.4.2)
Requirement already satisfied: Cython!=0.29.18,!=0.29.31,>=0.29 in
/usr/local/lib/python3.10/dist-packages (from pmdarima) (3.0.11)
Requirement already satisfied: numpy>=1.21.2 in
/usr/local/lib/python3.10/dist-packages (from pmdarima) (1.26.4)
Requirement already satisfied: pandas>=0.19 in
/usr/local/lib/python3.10/dist-packages (from pmdarima) (2.1.4)
Requirement already satisfied: scikit-learn>=0.22 in
```

```
/usr/local/lib/python3.10/dist-packages (from pmdarima) (1.3.2)
Requirement already satisfied: scipy>=1.3.2 in
/usr/local/lib/python3.10/dist-packages (from pmdarima) (1.13.1)
Requirement already satisfied: statsmodels>=0.13.2 in
/usr/local/lib/python3.10/dist-packages (from pmdarima) (0.14.3)
Requirement already satisfied: urllib3 in
/usr/local/lib/python3.10/dist-packages (from pmdarima) (2.0.7)
Requirement already satisfied: setuptools!=50.0.0,>=38.6.0 in
/usr/local/lib/python3.10/dist-packages (from pmdarima) (71.0.4)
Requirement already satisfied: packaging>=17.1 in
/usr/local/lib/python3.10/dist-packages (from pmdarima) (24.1)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.10/dist-packages (from pandas>=0.19->pmdarima)
(2.8.2)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.10/dist-packages (from pandas>=0.19->pmdarima)
(2024.2)
Requirement already satisfied: tzdata>=2022.1 in
/usr/local/lib/python3.10/dist-packages (from pandas>=0.19->pmdarima)
(2024.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.22-
>pmdarima) (3.5.0)
Requirement already satisfied: patsy>=0.5.6 in
/usr/local/lib/python3.10/dist-packages (from statsmodels>=0.13.2-
>pmdarima) (0.5.6)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-
packages (from patsy>=0.5.6->statsmodels>=0.13.2->pmdarima) (1.16.0)
```

```
import sys
sys.path.append("..")

# Ignore warnings
import warnings
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('seaborn-darkgrid')

# For data manipulation
import pandas as pd
import numpy as np
import itertools
from tqdm import tqdm

# Import adfuller
from statsmodels.tsa.stattools import adfuller

# For statistical analysis
```

```
from statsmodels.tsa.arima.model import ARIMA

#for price data
import yfinance as yf
import pandas_datareader.data as web
```

Read price data

```
# Define the symbol, start, and end dates for the Nifty index
symbol = '^NSEI'
start = '2019-01-01'
end = '2023-12-31'

# Download weekly closing price data for Nifty from Yahoo Finance
data = yf.download(symbol, start=start, end=end, interval='1d')[['Adj Close']]
# Display the first few rows of the data
print(data.head())
```

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

                Adj Close
Date
2019-01-02  10792.500000
2019-01-03  10672.250000
2019-01-04  10727.349609
2019-01-07  10771.799805
2019-01-08  10802.150391
```

Data Sanity Check

In this section, we define a method to ensure the integrity of our dataset before proceeding with further analysis. This method will address key aspects:

- **Missing Values**: Identification and imputation of missing data points using forward-fill to preserve continuity in time series data.
- **Outliers**: Detection and examination of extreme values based on statistical thresholds to prevent skewed analysis. Outliers will be assessed against a rolling median over a 52-week period to determine their significance.
- **Data Continuity**: Visual and statistical checks for discontinuities or abrupt changes in the data that could indicate recording errors or genuine market anomalies.

This proactive approach helps ensure that our model's inputs are reliable and that the ensuing analysis is robust.

```
import pandas as pd

def sanity_check(data):
    """
    Check for and handle data issues such as missing values and
extreme outliers.

    Parameters:
    - data (DataFrame): The DataFrame containing the 'Adj Close'
prices of the Nifty index.

    Returns:
    - DataFrame: The cleaned and potentially modified DataFrame.
    """
    # Check for missing values
    if data.isnull().values.any():
        print("Missing values found.")
        # Filling missing values with the previous day's prices
(forward fill)
        data.fillna(method='ffill', inplace=True)
        print("Missing values have been forward-filled.")
    else:
        print("No missing values in the data.")

    return data

data = sanity_check(data)
```

```
No missing values in the data.
```

```
# Check for invalid/NaN data
print(data.isnull().sum())
```

```
Adj Close    0
dtype: int64
```

This data has no NaN values.

Checking for Stationarity

- **ADF Test**: We employ the Augmented Dickey-Fuller (ADF) test, a popular statistical test used to check for stationarity. The test aims to determine whether a unit root is present in the series, which is indicative of non-stationarity.
- **Interpretation**: If the ADF test shows that the data does not contain a unit root, the series can be considered stationary, and models like ARMA (Autoregressive Moving Average) are suitable.
- **Differencing**: If the test indicates non-stationarity, we proceed by differencing the data. This involves subtracting the previous observation from the current observation. We then reapply the ADF test to the differenced data to check for stationarity again.

- **Model Application**: If the differenced data is stationary, we use an ARIMA (Autoregressive Integrated Moving Average) model, which integrates the differencing step into the modeling process.

This approach ensures that the time series model is built on a solid foundation of data that meets the necessary statistical properties, enhancing the reliability and accuracy of the forecasts.

```
def check_stationarity(df):
    adf_result = adfuller(df)

    if(adf_result[1] < 0.05):
        print('Data is stationary')
    else:
        print('Data is not stationary')

# Checking the Adj Close for stationarity
check_stationarity(data['Adj Close'])
```

```
Data is not stationary
```

The asset is not stationary. This means we can not use the ARMA model. Next, we will check whether we can apply the ARIMA model.

```
# Check the differenced asset 1 for stationarity
check_stationarity(data['Adj Close'].diff().dropna())
```

```
Data is stationary
```

Understanding ACF and PACF Plots

In the process of building ARIMA models for time series forecasting, the Autocorrelation Function (ACF) and the Partial Autocorrelation Function (PACF) are crucial tools for identifying the order of the AR (AutoRegressive) and MA (Moving Average) components.

Autocorrelation Function (ACF)

- **What is ACF?** The ACF measures the correlation between time series observations at different lags. In simpler terms, it expresses how well the current value of the series is related to its past values.
- **Significance:** ACF is used to identify the moving average component of an ARIMA model, denoted as (q). It shows the extent of correlation between a variable and its lag across successive time intervals. If the ACF shows a gradual decline, it suggests a moving average process might be suitable.

Partial Autocorrelation Function (PACF)

- **What is PACF?** The PACF measures the correlation between observations at two points in time, controlling for the values at all shorter lags. It isolates the effect of intervening time points.

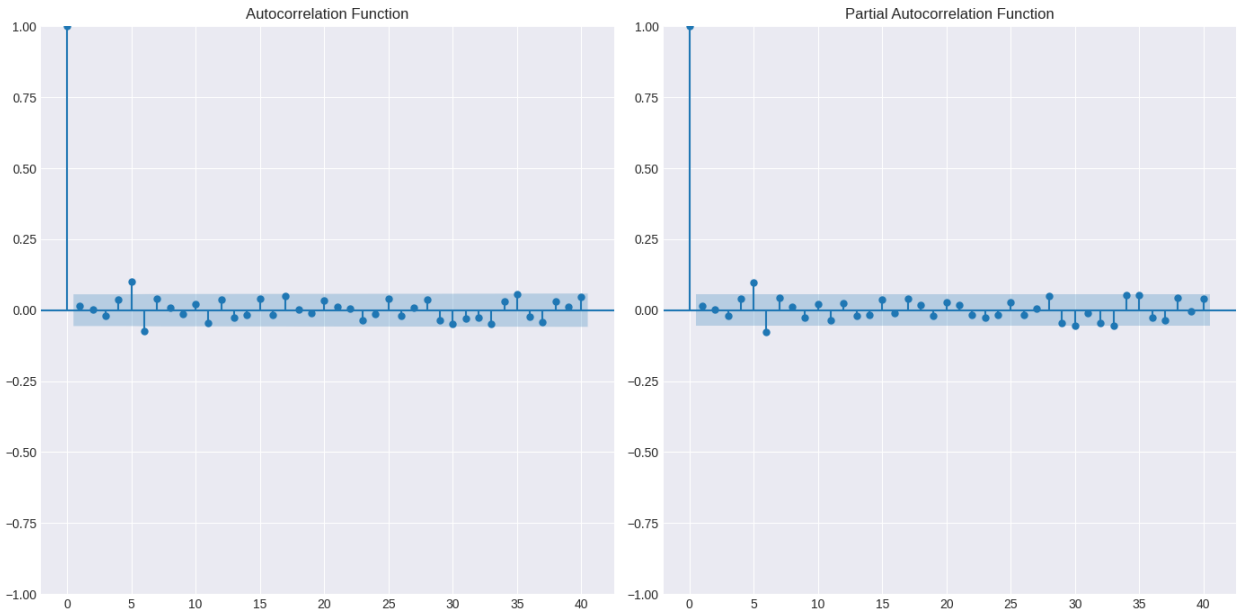
- **Significance:** PACF is primarily used to determine the order of the autoregressive component of an ARIMA model, denoted as (p). A sharp cut-off in the PACF after a certain number of lags suggests the order of the AR process.

Application in Time Series Analysis

- **Differencing:** Before plotting ACF and PACF, the series should be made stationary. This often involves differencing the series, where each value is subtracted from its previous value, to remove trends and seasonality.
- **Model Identification:** By examining the patterns in the ACF and PACF plots, you can make informed decisions about the likely parameters for the ARIMA model. For example, if the PACF plot shows a significant spike at lag 2, followed by non-significant spikes, this suggests an AR(2) model might be appropriate.

These plots provide a visual insight into the data which can guide the specification of your time series model, improving the model's accuracy and predictive power.

```
# Importing neccessary module from statsmodels library
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# To accurately map the acf and pacf, we need a stationary series
diff_data = data['Adj Close'].diff().dropna()

# Plotting ACF
plt.figure(figsize=(14, 7))
plt.subplot(121)  # 1 row, 2 columns, 1st subplot
plot_acf(diff_data, ax=plt.gca(), lags=40)
plt.title('Autocorrelation Function')

# Plotting PACF
plt.subplot(122)  # 1 row, 2 columns, 2nd subplot
plot_pacf(diff_data, ax=plt.gca(), lags=40)
plt.title('Partial Autocorrelation Function')

plt.tight_layout()
plt.show()
```



The data is stationary after differencing. This means we can use the ARIMA model. Let us now find the p,q, and d order of the ARIMA model.

We can compare the AIC score of the models with different p,q and d values. The best fit model would be the one with the lowest AIC score.

Modeling time series using ARIMA models

The **ARIMA (AutoRegressive Integrated Moving Average)** class of models is a popular statistical technique in time series forecasting. It exploits different standard temporal structures seen in time series processes.

> *Exponential smoothing and ARIMA models are the two most widely used approaches to time series forecasting, and provide complementary approaches to the problem. While exponential smoothing models are based on a description of the trend and seasonality in the data, ARIMA models aim to describe the autocorrelations in the data.*

Overview on ARIMA

We will now take a brief look at its key features by taking apart the acronym.

1. **Auto Regressive (AR)**:

- Regression of a time series process onto itself (its past versions)
- A time series process is AR if its present value depends on a linear combination of past observations.
- In financial time series, an AR model attempts to explain the mean reversion and trending behaviours that we observe in asset prices.

1. **Integrated (I)**:

For a time series process X_t recorded at regular intervals, the difference operation is defined as

$$\nabla X_t = X_t - X_{t-1}$$

The difference operator (denoted by ∇) can be applied repeatedly. For example,

$$\nabla^2 X_t = \nabla X_t - \nabla X_{t-1}$$

- A time series process is integrated of order d (denoted by $I(d)$), if differencing the observations d times, makes the process stationary.

1. **Moving Average (MA)**:

- A time series process is MA if its present value can be written as a linear combination of past error terms.
- MA models try to capture the idiosyncratic shocks observed in financial markets. We can think of events like terrorist attacks, earnings surprises, sudden political changes, etc. as the random shocks affecting the asset price movements.

When we use the ARIMA class to model a time series process, each of the above components are specified in the model as parameters (with the notations p, d, and q respectively).

That is, the classification $ARIMA(p,d,q)$ process can be thought of as

$$AR(p)I(d)MA(q)$$

Here,

1. p: The number of past observations (we usually call them *lagged terms*) of the process included in the model.
2. d: The number of times we difference the original process to make it stationary.
3. q: The number of past error terms (we usually call them *lagged error terms* or *lagged residuals*) of the process included in the model.

When we model our time series process with the ARIMA class, we implicitly assume that the underlying data generating process (and by extension the observations we record) is an ARIMA process.

We should validate our assumptions (especially the implicit ones which slip under the radar) and recognize the limitations of our models. A well-known deficiency of ARIMA applications on financial time series is its failure to capture the phenomenon of volatility clustering. However, despite their inaccurate point estimates, they give rise to informative confidence intervals.

All of the below models would have good explanatory and predictive power only if the process is stationary.

$$AR(1): X_t = \phi X_{t-1} + \epsilon_t$$

$$MA(1): X_t = \epsilon_t + \theta \epsilon_{t-1}$$

$$AR(p): X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \ldots + \phi_p X_{t-p} + \epsilon_t$$

$$MA(q): X_t = \epsilon_t + \theta_1 \epsilon_{t-1} + \ldots + \theta_q \epsilon_{t-q}$$

$$ARMA(p,q): X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \ldots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \ldots + \theta_q \epsilon_{t-q}$$

```
# Splitting the data into train and test that we can use the test data to evaluate our chosen model
split = int(len(data)*0.8)
data_train = data[:split]
data_test = data[split:]

print(data_train.shape[0], data_test.shape[0])
```

```
985 247
```

```
print(data_test.tail())
```

```
                Adj Close
Date
2023-12-22  21349.400391
2023-12-26  21441.349609
2023-12-27  21654.750000
2023-12-28  21778.699219
2023-12-29  21731.400391
```

```
import warnings
import pmdarima as pm

def find_best_arima_parameters(data_series, seasonal=False, m=1):
    """
    Finds the best ARIMA parameters for the given time series data using auto_arima from pmdarima.

    Parameters:
    - data_series (pd.Series): Pandas Series containing the time series data.
    - seasonal (bool): Whether to consider seasonal differencing.
    - m (int): The number of periods in each season (relevant if seasonal is True).

    Returns:
    - tuple: The best (p, d, q) parameters for the ARIMA model.
    """
    # Ignoring warnings
    warnings.filterwarnings('ignore')

    # Performing the auto ARIMA process
    model = pm.auto_arima(data_series, seasonal=seasonal, m=m,
                          trace=True, error_action='ignore', suppress_warnings=True)

    # Extracting the best ARIMA order
    best_order = model.order  # This gives (p, d, q)

    # Optionally, print the summary of the best model
    print(model.summary())
```

```
    return best_order

# Assuming 'data' is your DataFrame and 'Adj Close' is a column in
that DataFrame
best_param = find_best_arima_parameters(data_train['Adj Close'])
print("Best ARIMA parameters (p, d, q):", best_param)

Performing stepwise search to minimize aic
 ARIMA(2,1,2)(0,0,0)[0] intercept   : AIC=12798.213, Time=0.84 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=12805.026, Time=0.03 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=12806.984, Time=0.05 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=12806.989, Time=0.05 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=12805.113, Time=0.02 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : AIC=12810.971, Time=0.17 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=12810.981, Time=0.09 sec
 ARIMA(3,1,2)(0,0,0)[0] intercept   : AIC=12798.938, Time=0.67 sec
 ARIMA(2,1,3)(0,0,0)[0] intercept   : AIC=12799.055, Time=0.84 sec
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=12808.989, Time=0.07 sec
 ARIMA(1,1,3)(0,0,0)[0] intercept   : AIC=12812.291, Time=0.34 sec
 ARIMA(3,1,1)(0,0,0)[0] intercept   : AIC=12812.400, Time=0.23 sec
 ARIMA(3,1,3)(0,0,0)[0] intercept   : AIC=12802.674, Time=1.43 sec
 ARIMA(2,1,2)(0,0,0)[0]             : AIC=12811.477, Time=0.34 sec

Best model:  ARIMA(2,1,2)(0,0,0)[0] intercept
Total fit time: 5.171 seconds
                               SARIMAX Results

==============================================================================
========
Dep. Variable:                           y   No. Observations:
985
Model:                 SARIMAX(2, 1, 2)   Log Likelihood                -
6393.107
Date:                  Thu, 19 Sep 2024   AIC
12798.213
Time:                          09:24:06   BIC
12827.563
Sample:                               0   HQIC
12809.377
                                  - 985

Covariance Type:                    opg

==============================================================================
========
                 coef    std err          z      P>|z|      [0.025
0.975]
------------------------------------------------------------------------------
--------
```

```
intercept      7.1539     14.145      0.506      0.613     -20.570
34.878
ar.L1         -0.6018      0.019    -32.133      0.000      -0.639
-0.565
ar.L2         -0.9424      0.018    -53.080      0.000      -0.977
-0.908
ma.L1          0.6298      0.012     53.217      0.000       0.607
0.653
ma.L2          0.9785      0.013     74.881      0.000       0.953
1.004
sigma2      2.586e+04    670.942     38.544      0.000    2.45e+04
2.72e+04
===================================================================
=============
Ljung-Box (L1) (Q):                   0.75   Jarque-Bera (JB):
1267.92
Prob(Q):                              0.39   Prob(JB):
0.00
Heteroskedasticity (H):               1.16   Skew:
-0.76
Prob(H) (two-sided):                  0.19   Kurtosis:
8.35
===================================================================
=============

Warnings:
[1] Covariance matrix calculated using the outer product of gradients
(complex-step).
Best ARIMA parameters (p, d, q): (2, 1, 2)
```

We find that ARIMA(2,1,2) is the best model

Forecast Prices

We will now create a function to apply the ARIMA model with the selected parameters on the close price data.

```
def get_predicted_prices(close_prices, best_param):

    # Defining the best model
    best_model = ARIMA(close_prices.values, order=best_param)
    best_model_fit = best_model.fit(method_kwargs={"warn_convergence":
False})

    # Making forecast
    predictions = best_model_fit.forecast(steps=1)[0]
    return predictions
```

```
# Assuming best_param has already been determined using the training
data
predictions = []
for i in range(len(data_test['Adj Close'])):
    # Incrementally adding test data to simulate a rolling forecast
    current_data = pd.concat([data_train['Adj Close'], data_test['Adj
Close'].iloc[:i]])
    next_pred = get_predicted_prices(current_data, best_param)
    predictions.append(next_pred)

# Converting the predictions list to a DataFrame
predictions_df = pd.DataFrame(predictions,
columns=['predicted_price'])
predictions_df.index = data_test.index

print(predictions_df)
```

```
            predicted_price
Date
2022-12-29     18124.805379
2022-12-30     18190.704801
2023-01-02     18105.668161
2023-01-03     18197.011369
2023-01-04     18232.363400
...                     ...
2023-12-22     21254.774560
2023-12-26     21357.397053
2023-12-27     21443.287619
2023-12-28     21658.757517
2023-12-29     21782.182140

[247 rows x 1 columns]
```

```
#concatenating the predictions_df with data_test
data_test = pd.concat([data_test, predictions_df], axis=1)
print(data_test)
```

```
                Adj Close  predicted_price
Date
2022-12-29  18191.000000     18124.805379
2022-12-30  18105.300781     18190.704801
2023-01-02  18197.449219     18105.668161
2023-01-03  18232.550781     18197.011369
2023-01-04  18042.949219     18232.363400
...                  ...              ...
2023-12-22  21349.400391     21254.774560
2023-12-26  21441.349609     21357.397053
2023-12-27  21654.750000     21443.287619
2023-12-28  21778.699219     21658.757517
2023-12-29  21731.400391     21782.182140
```

```
[247 rows x 2 columns]
```

```
data_test['predicted_returns'] =
data_test['predicted_price'].pct_change()
data_test['actual_returns'] = data_test['Adj Close'].pct_change()

data_test.dropna(inplace=True)

#signal generation
data_test['signal'] = np.where(data_test['predicted_returns'] >=0, 1,
-1)

#we may approach differently here and assume that we have learnt the
predicted prices in advance and will be able to position ourselves
data_test['strategy_returns'] = data_test['signal'] *
data_test['actual_returns']
data_test['cumulative_returns'] =
np.cumprod(data_test['strategy_returns']+1)
print(data_test)
```

```
                Adj Close  predicted_price  predicted_returns
actual_returns  \
Date

2023-01-02  18197.449219     18105.668161          -0.004675
0.005090
2023-01-03  18232.550781     18197.011369           0.005045
0.001929
2023-01-04  18042.949219     18232.363400           0.001943       -
0.010399
2023-01-05  17992.150391     18037.771994          -0.010673       -
0.002815
2023-01-06  17859.449219     17991.807100          -0.002548       -
0.007376
...                  ...              ...                ...
...
2023-12-22  21349.400391     21254.774560           0.005168
0.004439
2023-12-26  21441.349609     21357.397053           0.004828
0.004307
2023-12-27  21654.750000     21443.287619           0.004022
0.009953
2023-12-28  21778.699219     21658.757517           0.010048
0.005724
2023-12-29  21731.400391     21782.182140           0.005699       -
0.002172

            signal  strategy_returns  cumulative_returns
Date
```

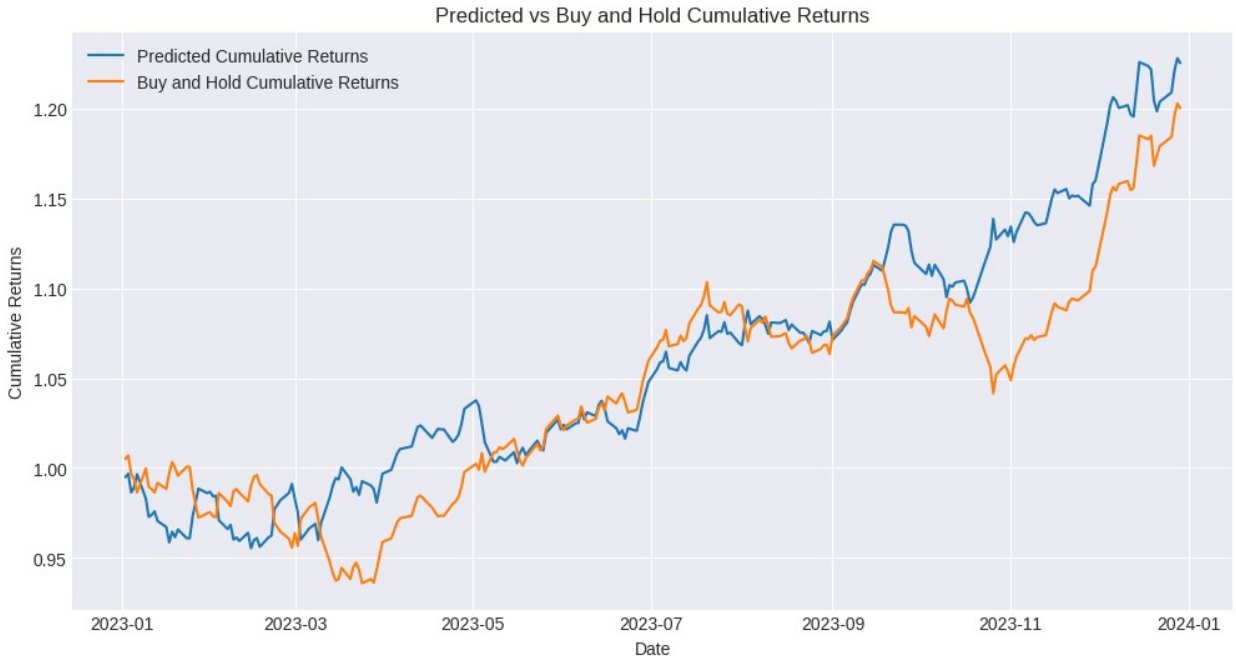
```
2023-01-02      -1         -0.005090             0.994910
2023-01-03       1          0.001929             0.996830
2023-01-04       1         -0.010399             0.986463
2023-01-05      -1          0.002815             0.989241
2023-01-06      -1          0.007376             0.996537
...            ...               ...                  ...
2023-12-22       1          0.004439             1.203870
2023-12-26       1          0.004307             1.209055
2023-12-27       1          0.009953             1.221089
2023-12-28       1          0.005724             1.228078
2023-12-29       1         -0.002172             1.225411

[245 rows x 7 columns]
```

```
#buy and hold returns
buy_and_hold_returns = (1+data_test['actual_returns']).cumprod()
print(buy_and_hold_returns)
```

```
Date
2023-01-02    1.005090
2023-01-03    1.007028
2023-01-04    0.996556
2023-01-05    0.993750
2023-01-06    0.986421
                ...
2023-12-22    1.179180
2023-12-26    1.184258
2023-12-27    1.196045
2023-12-28    1.202891
2023-12-29    1.200278
Name: actual_returns, Length: 245, dtype: float64
```

Plot the predicted and actual cumulative(buy and hold returns)

```
plt.figure(figsize=(12, 6))
plt.plot(data_test['cumulative_returns'], label='Predicted Cumulative
Returns')
plt.plot(buy_and_hold_returns, label='Buy and Hold Cumulative
Returns')
plt.xlabel('Date')
plt.ylabel('Cumulative Returns')
plt.title('Predicted vs Buy and Hold Cumulative Returns')
plt.legend()
plt.show()
```



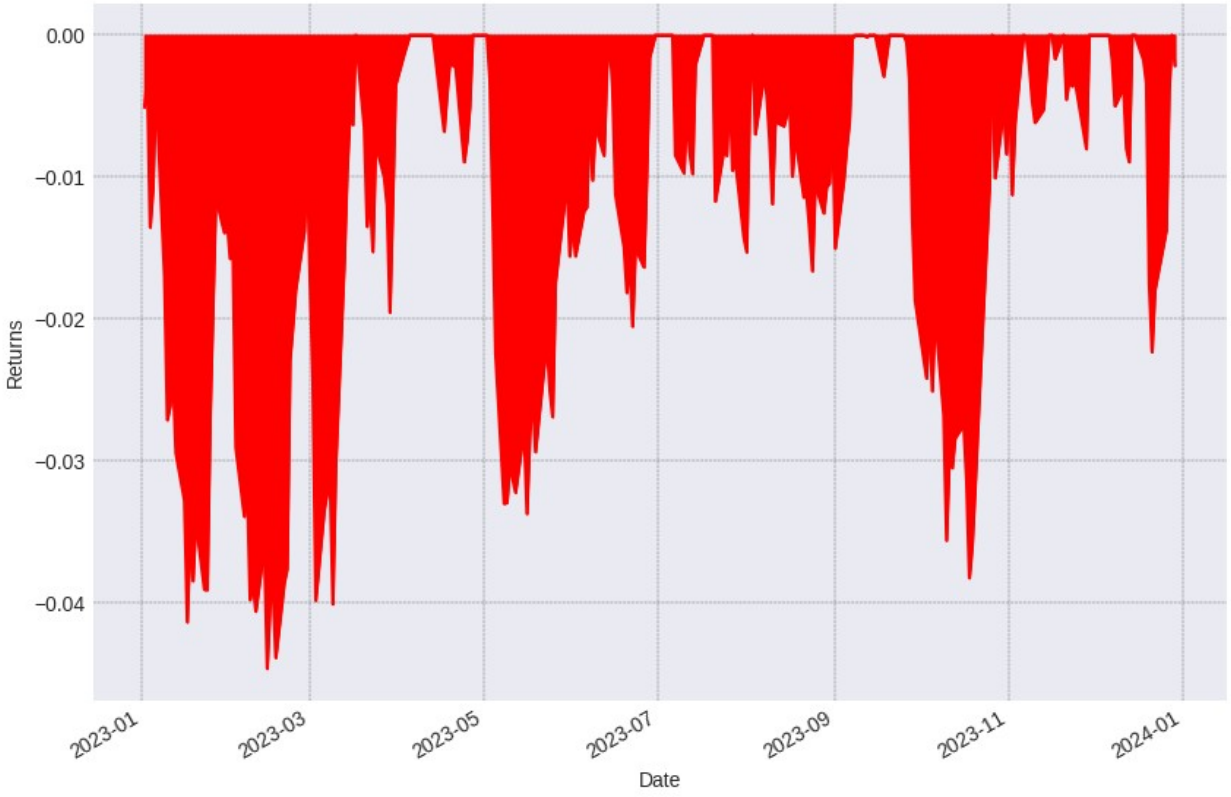
WE CAN SEE HOW OUR STRATEGY OUTPERFORMED THE BUY AND HOLD RETURNS(MARKET RETURNS)

WE WILL COMPARE THE STRATGY ON TWO MORE METRICS

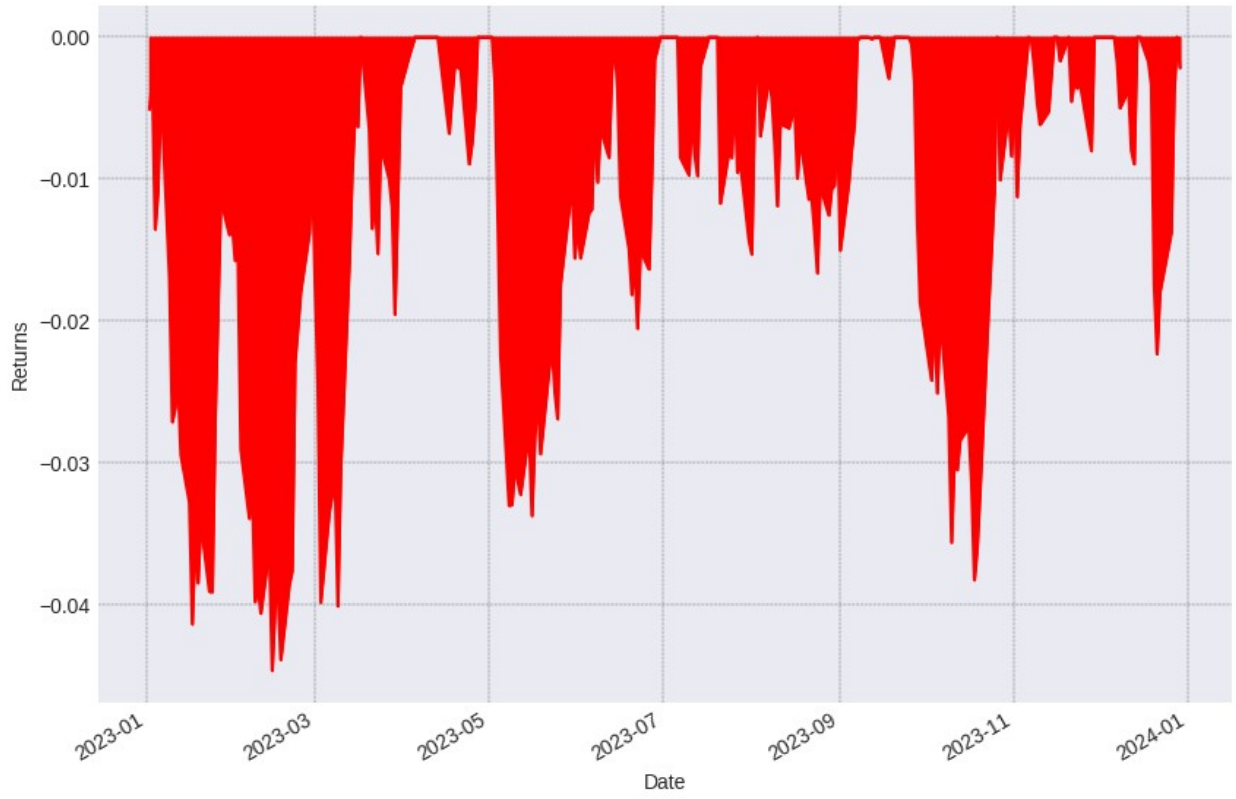
1. SHARPE_RATIO
2. DRAWDOWN

```
#DRAWDOWN ANALYSIS
def calc_drawdown(cum_rets):
    # Calculate the running maximum
    running_max = np.maximum.accumulate(cum_rets.dropna())
    # Ensure the value never drops below 1
    running_max[running_max < 1] = 1
    # Calculate the percentage drawdown
    drawdown = (cum_rets)/running_max - 1
    return drawdown

def plot_drawdown(drawdown):
    fig = plt.figure(figsize=(10, 7))
    # Plot
    drawdown.plot(color='r')
    plt.ylabel('Returns')
    plt.fill_between(drawdown.index, drawdown, color='red')
    plt.grid(which="major", color='k', linestyle='-.', linewidth=0.2)
    plt.show()

drawdown_strategy = calc_drawdown(data_test['cumulative_returns'])
print("The maximum drawdown of the strategy is %.2f" %
```

```
(drawdown_strategy.min()*100))
plot_drawdown(drawdown_strategy)

The maximum drawdown of the strategy is -4.46
```




```
#buy and hold max drawdown

drawdown_buy_n_hold = calc_drawdown(buy_and_hold_returns)
print("The maximum drawdown of the bnh strategy is %.2f" %
(drawdown_buy_n_hold.min()*100))
plot_drawdown(drawdown_strategy)

The maximum drawdown of the bnh strategy is -7.06
```




```
#let's check the sharpe ratio of the strategy
sharpe_ratio =
(data_test['strategy_returns'].mean()*252)/(data_test['strategy_return
s'].std()*np.sqrt(252))
print("The sharpe ratio of the strategy is %.2f" % sharpe_ratio)
```

```
The sharpe ratio of the strategy is 2.18
```

Conclusion

Time series analysis is fundamental in quantitative finance for making informed predictions about market behavior. Techniques such as **ARIMA**, **GARCH**, and **Exponential Smoothing** provide powerful tools for forecasting asset prices, volatility, and trends. These models help traders and portfolio managers anticipate market movements, manage risks, and optimize trading strategies.

By combining time series techniques with financial intuition, analysts can gain deeper insights into how assets will perform in the future, leading to more robust decision-making processes.