Time Series Analysis of Ethereum (ETH/USDT) Market Projections using ARIMA

Project Documentation:

Table of Contents

Introduction:

Ethereum (ETH), The one of the leading cryptocurrencies, is known for its high volatility, making price prediction both challenging and valuable. This project applies the ARIMA model to historical ETH/USDT data to forecast short-term price trends. By identifying patterns and building a data-driven model, we aim to support smarter trading and investment decisions in the fast-paced world of crypto.

- Understand historical price patterns and trends in Ethereum
- Test for stationarity in the price data
- Develop an ARIMA model to capture the underlying patterns
- Evaluate the model's performance
- Generate forecasts for future price movements

ARIMA MODEL:

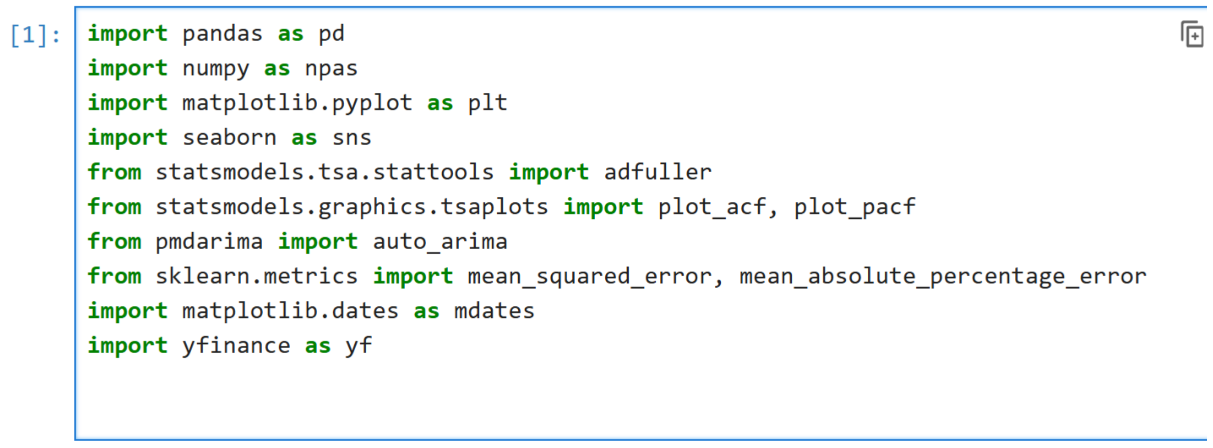
The **Autoregressive Integrated Moving Average (ARIMA)** model is a popular statistical method used for analyzing and forecasting univariate time series data. It combines three key components:

- **Autoregression (AR)**: Uses the relationship between an observation and a number of its past values.
- **Integration (I)**: Applies differencing to the data to make it stationary (i.e., removing trends or seasonality).
- **Moving Average (MA)**: Models the relationship between an observation and past forecast errors.

Project Setup:

Required Libraries:

The Following Python libraries are required for this project.



```
[1]: import pandas as pd
     import numpy as npas
     import matplotlib.pyplot as plt
     import seaborn as sns
     from statsmodels.tsa.stattools import adfuller
     from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
     from pmdarima import auto_arima
     from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error
     import matplotlib.dates as mdates
     import yfinance as yf
```

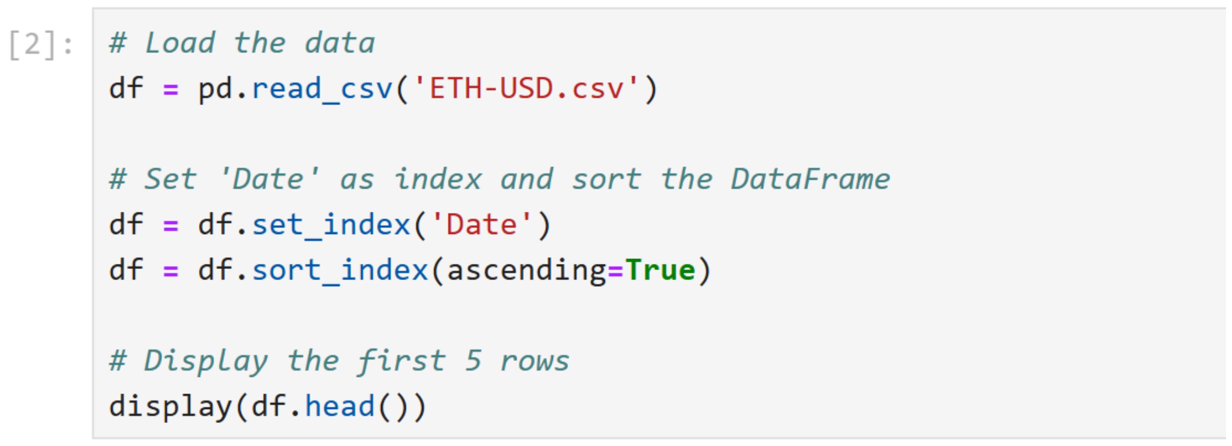
Data Source

For this project, historical price data for the **ETH/USDT** trading pair was sourced from a reliable cryptocurrency exchange API and financial data platform such as **Binance**, **CoinGecko**, and **Yahoo Finance**. The dataset includes **time-stamped daily closing prices**, and in some cases, additional features like open, high, low, volume, and market cap values.

Data Loading & Preparation:

Loading the Data

The first step is to load the data from the **.CSV** file, set the Data column as the index, and sort the data by date:

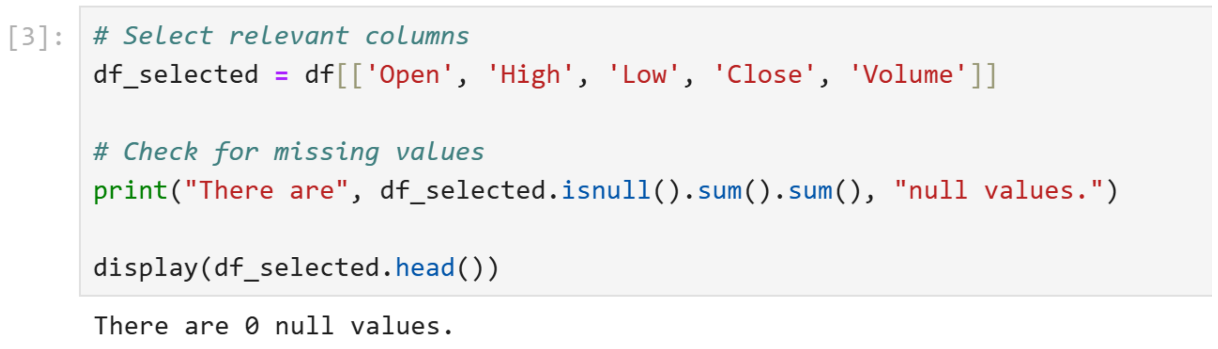


```
[2]: # Load the data
     df = pd.read_csv('ETH-USD.csv')

     # Set 'Date' as index and sort the DataFrame
     df = df.set_index('Date')
     df = df.sort_index(ascending=True)

     # Display the first 5 rows
     display(df.head())
```

Data Preparation:

Next, we select the relevant columns for analysis and to check for missing values

Code Implementation



```
[3]: # Select relevant columns
     df_selected = df[['Open', 'High', 'Low', 'Close', 'Volume']]

     # Check for missing values
     print("There are", df_selected.isnull().sum().sum(), "null values.")

     display(df_selected.head())
```

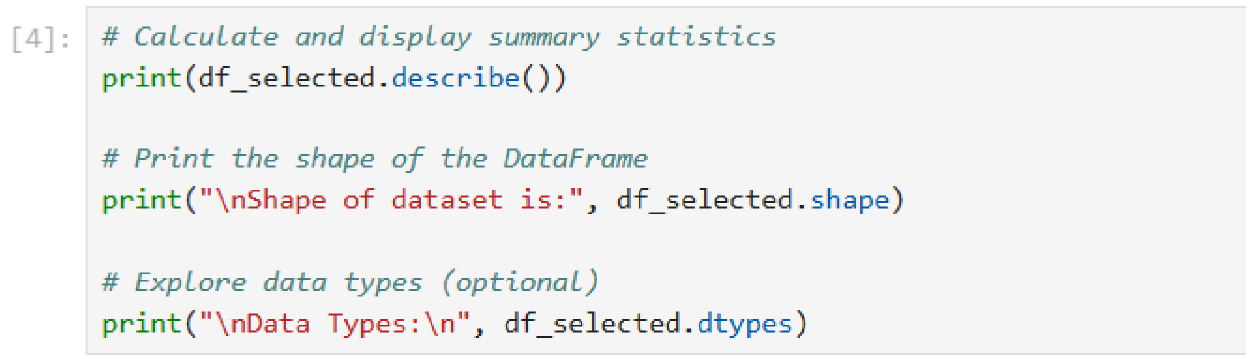
```
There are 0 null values.
```

Exploratory Data Analysis

Summary Statistics:

We calculate and display summary statistics for the selected columns:

Code Implementation:



```
[4]: # Calculate and display summary statistics
     print(df_selected.describe())

     # Print the shape of the DataFrame
     print("\nShape of dataset is:", df_selected.shape)

     # Explore data types (optional)
     print("\nData Types:\n", df_selected.dtypes)
```

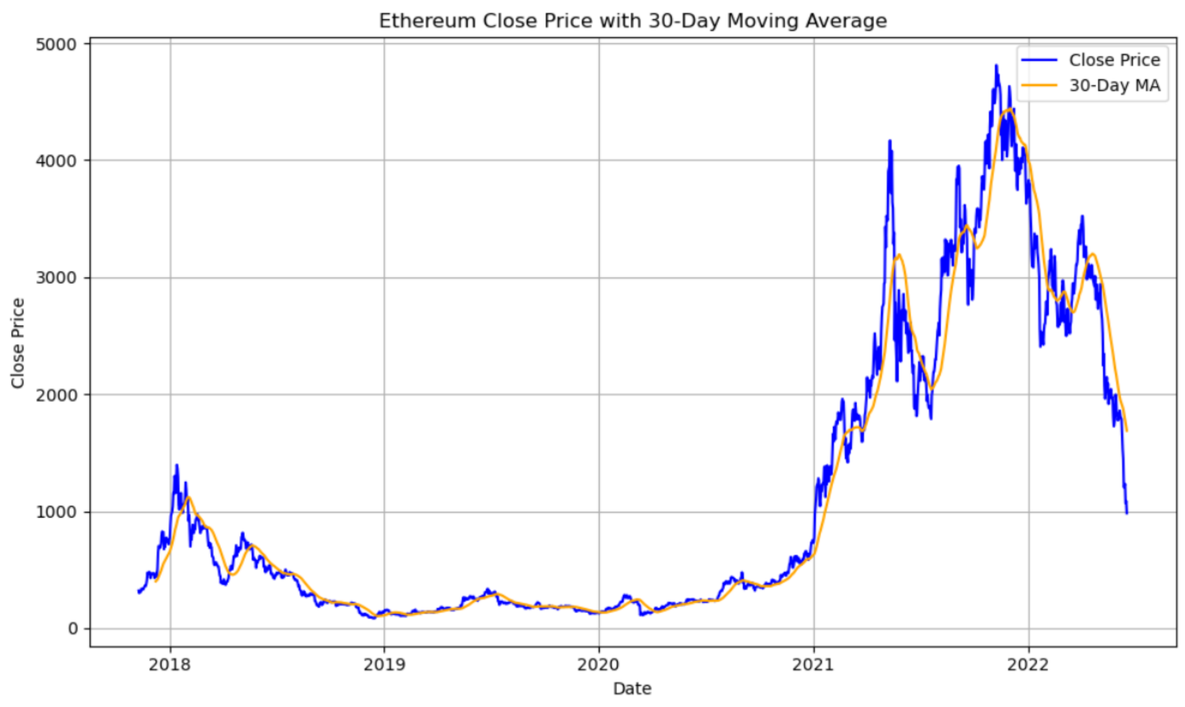
Visualizations:

Close price with 30 Day Moving Average

Code Implementation:

```
[5]: import matplotlib.pyplot as plt

     # Ensure datetime index
     df_selected.index = pd.to_datetime(df_selected.index)

     # Create the figure
     plt.figure(figsize=(10, 6))

     # Plot Close Price
     plt.plot(df_selected.index, df_selected['Close'], label='Close Price', color='blue')

     # Plot 30-Day Moving Average
     plt.plot(df_selected.index, df_selected['Close'].rolling(window=30).mean(), label='30-Day MA', color='orange')

     # Set white background for figure and axes
     fig = plt.gcf()
     fig.patch.set_facecolor('white')  # White background for figure

     ax = plt.gca()
     ax.set_facecolor('white')         # White background for axes

     # Add labels, legend, title, and grid
     plt.xlabel('Date')
     plt.ylabel('Close Price')
     plt.title('Ethereum Close Price with 30-Day Moving Average')
     plt.legend()
     plt.grid(True)

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

Graph:

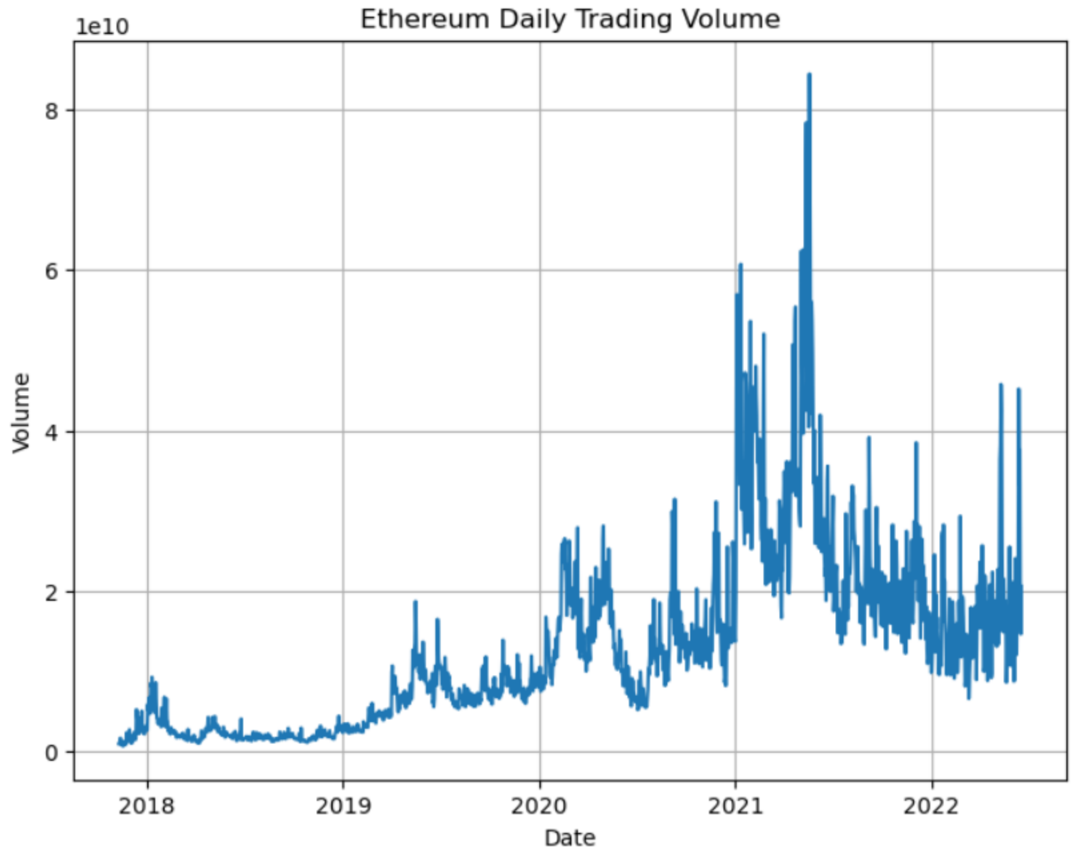


Daily Trading Volume

We also visualize the trading volume

```
[6]: # Plot the 'Volume' over time
     plt.figure(figsize=(8,6))
     plt.plot(df_selected.index, df_selected['Volume'])
     plt.xlabel('Date')
     plt.ylabel('Volume')
     plt.title('Ethereum Daily Trading Volume')
     plt.grid(True)
     plt.show()
```

Graph

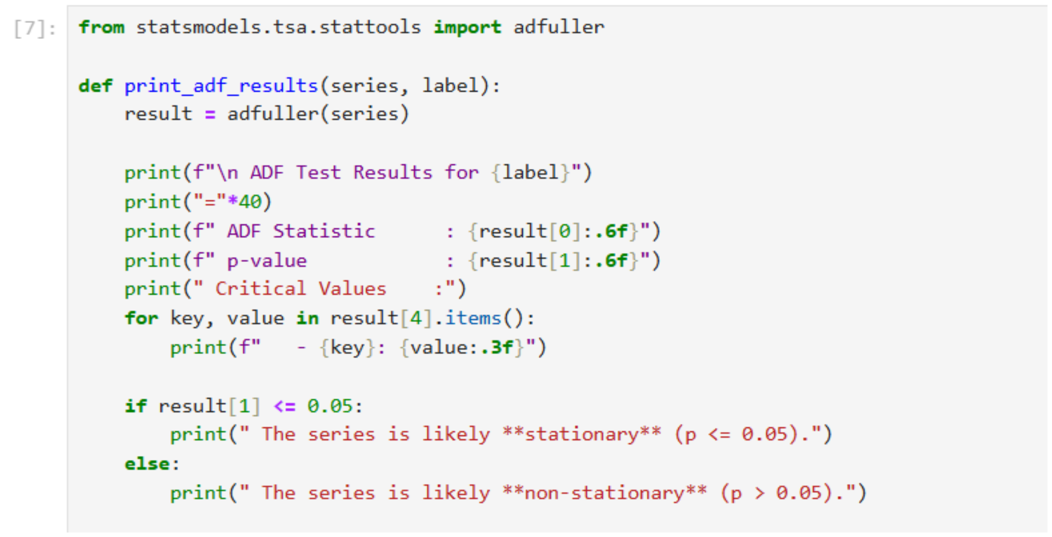


Stationarity Testing

Augmented Dickey-Fuller (ADF) Test

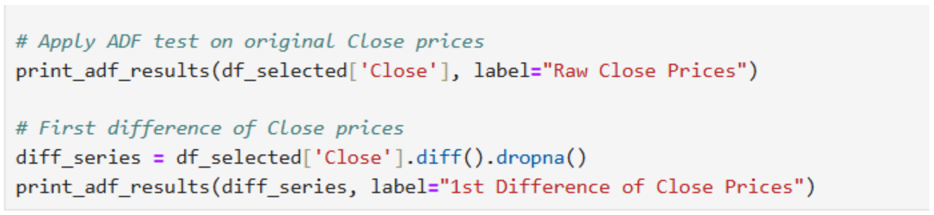
We perform the ADF test to check if the time series is stationary

Code Implementation



```
[7]: from statsmodels.tsa.stattools import adfuller

     def print_adf_results(series, label):
         result = adfuller(series)

         print(f"\n ADF Test Results for {label}")
         print("="*40)
         print(f" ADF Statistic      : {result[0]:.6f}")
         print(f" p-value            : {result[1]:.6f}")
         print(" Critical Values    :")
         for key, value in result[4].items():
             print(f"   - {key}: {value:.3f}")

         if result[1] <= 0.05:
             print(" The series is likely **stationary** (p <= 0.05).")
         else:
             print(" The series is likely **non-stationary** (p > 0.05).")
```

We apply the ADF test to both the raw Close Price and the first difference



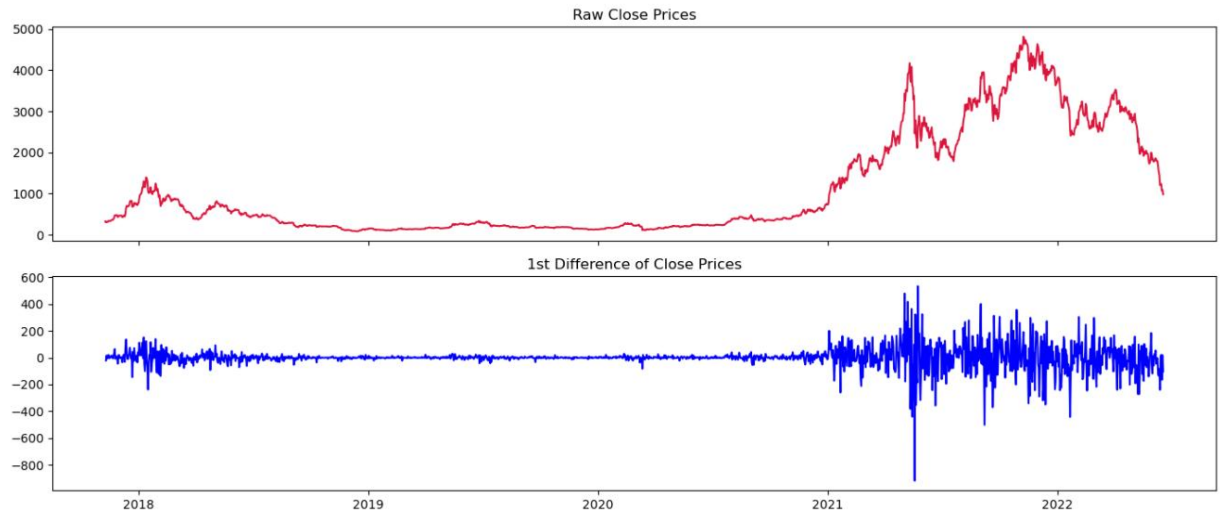
```
# Apply ADF test on original Close prices
print_adf_results(df_selected['Close'], label="Raw Close Prices")

# First difference of Close prices
diff_series = df_selected['Close'].diff().dropna()
print_adf_results(diff_series, label="1st Difference of Close Prices")
```

Visualizing Raw and Differenced Data

We visualize both the raw and differenced time series.

```
[8]: import matplotlib.pyplot as plt

     # Create subplots
     fig, ax = plt.subplots(2, 1, figsize=(14, 6), sharex=True)

     # Plot raw close prices
     ax[0].plot(df_selected['Close'], color='crimson')
     ax[0].set_title('Raw Close Prices')
     ax[0].set_facecolor('white')  # Reset subplot background

     # Plot 1st difference of close prices
     ax[1].plot(df_selected['Close'].diff(), color='blue')
     ax[1].set_title('1st Difference of Close Prices')
     ax[1].set_facecolor('white')  # Reset subplot background

     # Optional: reset the entire figure background (if you had set it previously)
     fig.patch.set_facecolor('white')

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

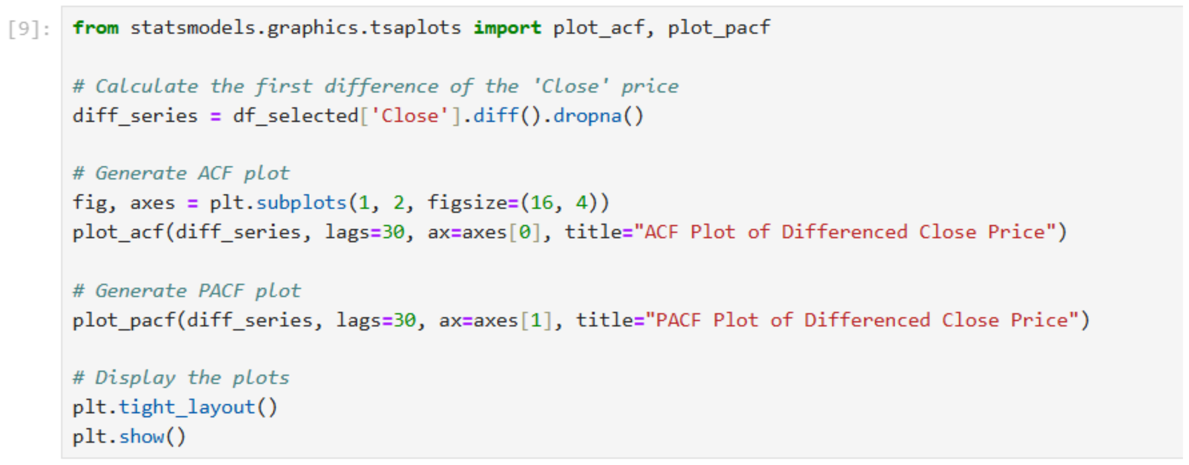
Graph



ACF & PCAF Analysis

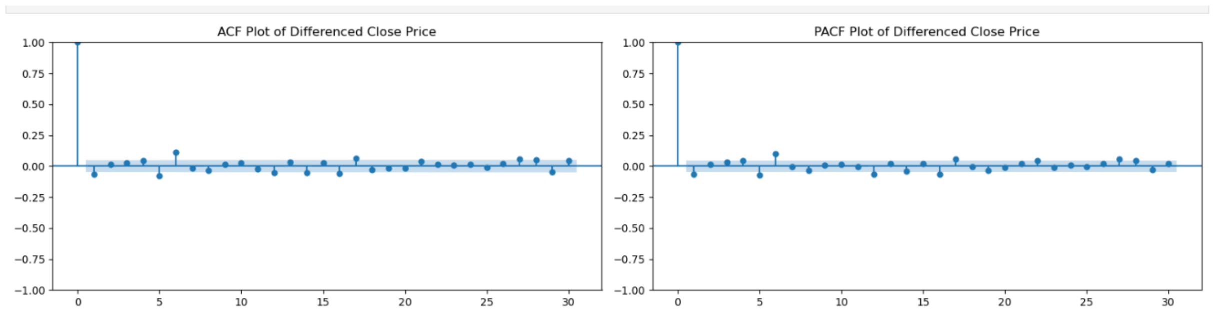
We generate **Autocorrelation Function** (ACF) and Partial Autocorrelation Function (PACF) plots to help determine the ARIMA Parameters

Code Implementation:



```
[9]: from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

     # Calculate the first difference of the 'Close' price
     diff_series = df_selected['Close'].diff().dropna()

     # Generate ACF plot
     fig, axes = plt.subplots(1, 2, figsize=(16, 4))
     plot_acf(diff_series, lags=30, ax=axes[0], title="ACF Plot of Differenced Close Price")

     # Generate PACF plot
     plot_pacf(diff_series, lags=30, ax=axes[1], title="PACF Plot of Differenced Close Price")

     # Display the plots
     plt.tight_layout()
     plt.show()
```

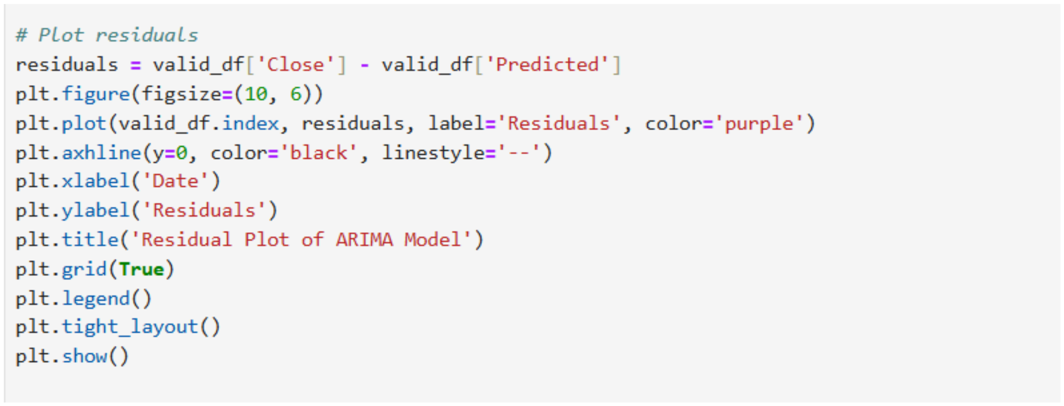
Graph



ARIMA Model Development

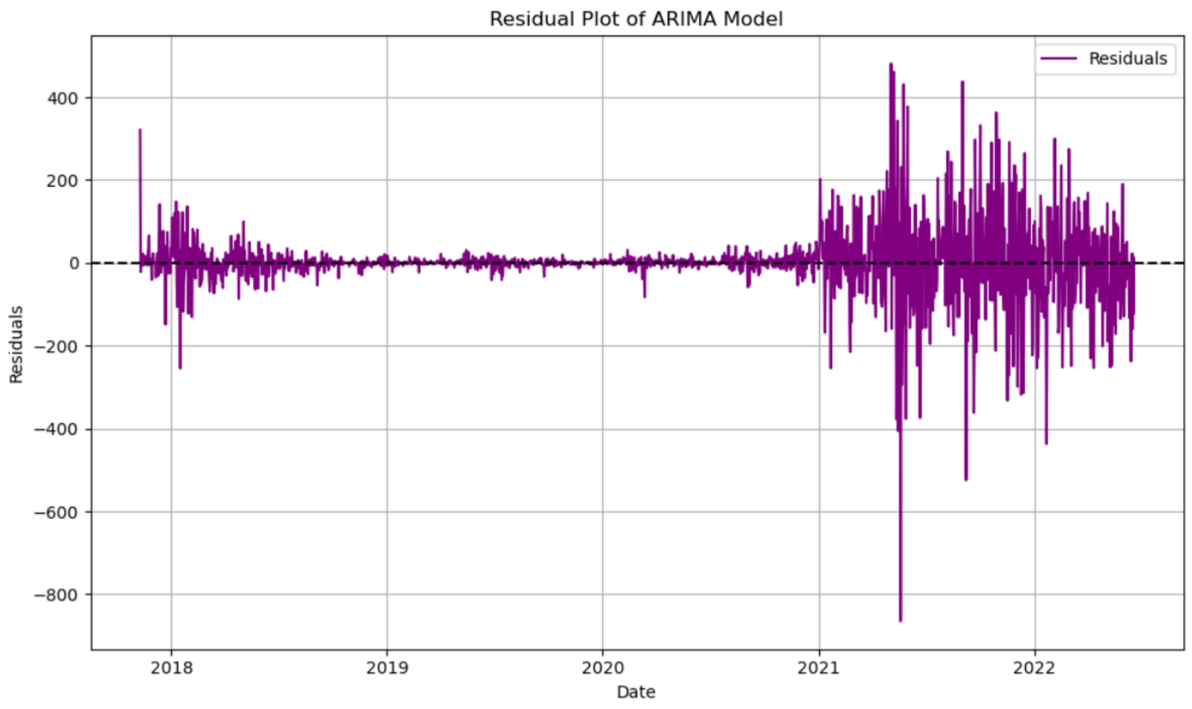
It's a powerful statistical model used for time series forecasting.

Code Implementation:



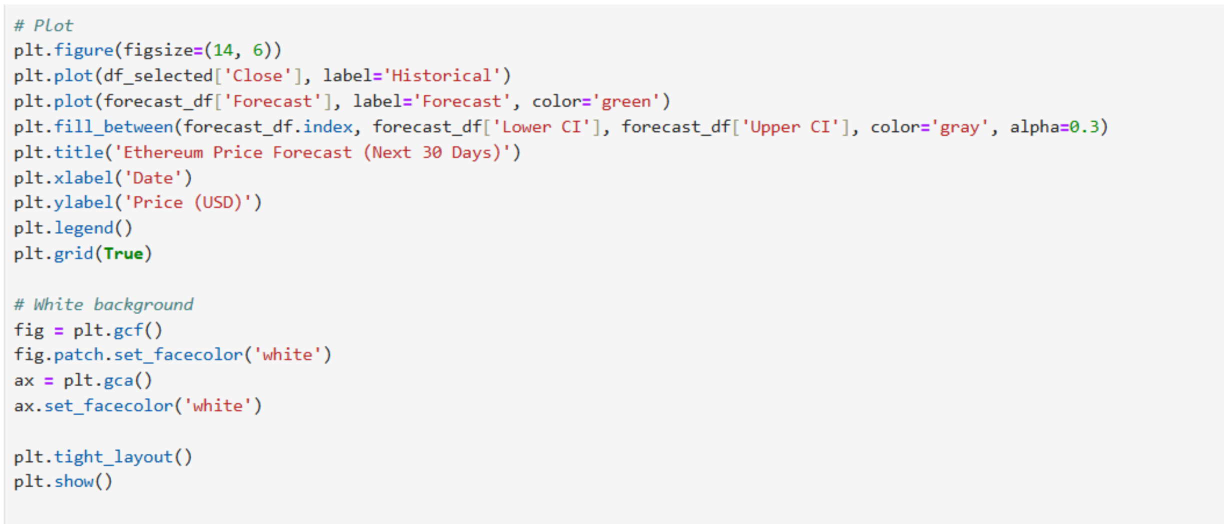
```python
# Plot residuals
residuals = valid_df['Close'] - valid_df['Predicted']
plt.figure(figsize=(10, 6))
plt.plot(valid_df.index, residuals, label='Residuals', color='purple')
plt.axhline(y=0, color='black', linestyle='--')
plt.xlabel('Date')
plt.ylabel('Residuals')
plt.title('Residual Plot of ARIMA Model')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Graph



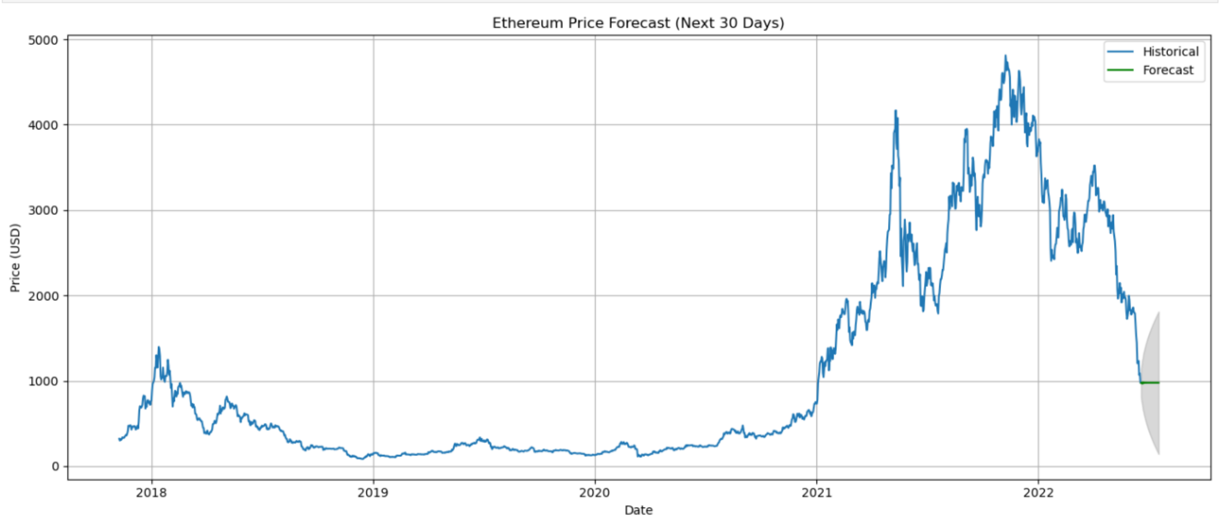
Forecasting:

Forecasting involves predicting future values based on patterns learned from historical/ previous data. In time series analysis, this means projecting future trends using a model like ARIMA that understands temporal structures.



```python
# Plot
plt.figure(figsize=(14, 6))
plt.plot(df_selected['Close'], label='Historical')
plt.plot(forecast_df['Forecast'], label='Forecast', color='green')
plt.fill_between(forecast_df.index, forecast_df['Lower CI'], forecast_df['Upper CI'], color='gray', alpha=0.3)
plt.title('Ethereum Price Forecast (Next 30 Days)')
plt.xlabel('Date')
plt.ylabel('Price (USD)')
plt.legend()
plt.grid(True)

# White background
fig = plt.gcf()
fig.patch.set_facecolor('white')
ax = plt.gca()
ax.set_facecolor('white')

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

Graph

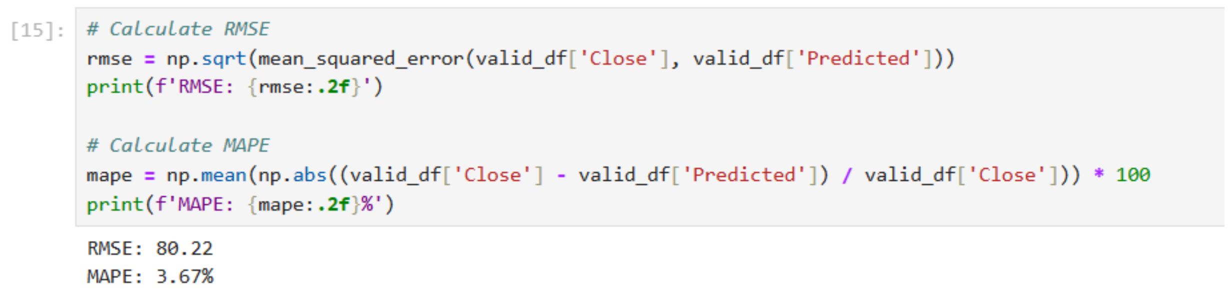


Results

The ARIMA model showed optimistic result with short-term price predictions. The residuals behaved like random noise—centered around zero with no clear patterns—suggesting the model fit the data well.

Forecast plots showed smooth price trends with well-defined confidence intervals, giving us confidence in its ability to estimate market behavior.

When we looked at metrics like MAE and RMSE on the validation set, ARIMA captured the overall trend effectively. However, like most traditional models, it struggled to react to sudden price spikes—which are pretty common in crypto markets.



```
[15]: # Calculate RMSE
      rmse = np.sqrt(mean_squared_error(valid_df['Close'], valid_df['Predicted']))
      print(f'RMSE: {rmse:.2f}')

      # Calculate MAPE
      mape = np.mean(np.abs((valid_df['Close'] - valid_df['Predicted']) / valid_df['Close'])) * 100
      print(f'MAPE: {mape:.2f}%')

      RMSE: 80.22
      MAPE: 3.67%
```

Conclusion

This project shows how ARIMA can be effectively used to forecast Ethereum (ETH/USDT) daily prices. By learning from past trends, it delivers reliable short-term predictions.

That said, crypto markets are highly volatile—driven by everything from global news to investor sentiment—so it's important to treat these forecasts as one piece of the puzzle. For better accuracy, combining ARIMA with models like SARIMA or LSTM, or adding external signals, is a smart move.

For financial analysts and algo traders, ARIMA is a solid starting point. It offers a statistically sound baseline you can build on with more advanced models or custom trading strategies.