**PROBLEM STATEMENT:**

In today's fast-paced financial markets, investors and analysts require efficient and user-friendly tools to perform comprehensive stock market analysis and make informed investment decisions. Traditional methods often involve complex software and significant manual effort, lacking the real-time interactivity and integration capabilities necessary for dynamic market conditions. There is a need for an intuitive, accessible platform that can seamlessly integrate with financial data sources, perform advanced data analytics, and provide interactive visualizations and predictive insights. This project aims to develop a web-based stock market analytics application using Streamlit and Python, which addresses these needs by offering a comprehensive, real-time, and user-friendly solution for stock market analysis.

**ABSTRACT:**

This paper presents the development of an interactive web application for comprehensive stock market analytics, leveraging Streamlit, a prominent Python library. The application integrates financial data retrieval, technical analysis, and machine learning to offer insightful visualizations and predictive analytics for stock market data. Historical stock data is fetched using the Yahoo Finance API, allowing users to input stock symbols and date ranges. Key technical indicators such as Volume Weighted Average Price (VWAP), Moving Average Convergence Divergence (MACD), Relative Strength Index (RSI), Bollinger Bands, On-Balance Volume (OBV), Simple Moving Average (SMA), Exponential Moving Average (EMA), Average True Range (ATR), Stochastic Oscillator, and Chaikin Money Flow (CMF) are calculated and visualized.

The application also facilitates comparative analysis of multiple stocks and includes advanced machine learning capabilities. A Random Forest Regressor model is employed to predict future stock closing prices based on selected features, with the model's performance evaluated using Mean Squared Error (MSE) and R-squared (R²) metrics. Data preprocessing functionalities such as binning, normalization, and both random and stratified sampling are implemented to enhance the analysis process.

Dynamic and animated charts are created using Plotly, enhancing user engagement and comprehension of complex financial data. The application aims to serve both novice and experienced traders by providing a robust tool for analyzing stock performance, predicting prices, and making informed trading decisions. The user-friendly interface and interactive visualizations make this application a valuable asset for financial analysis.

**STREAMLIT:**

The purpose of Streamlit in this project is to provide a straightforward and efficient platform for developing an interactive web application that allows users to perform comprehensive stock market analysis. It facilitates the creation of real-time, dynamic visualizations and user interfaces without requiring extensive front-end development knowledge. Streamlit integrates seamlessly with Python libraries, enabling rapid prototyping, easy deployment, and immediate user feedback, which enhances the overall user experience and accessibility of the stock market analytics tool.

**PYTHON:**

Python is preferred for this project due to its simplicity, readability, and extensive ecosystem of libraries that cater specifically to data analysis, machine learning, and web development. Libraries such as Pandas, NumPy, Scikit-learn, and Plotly provide robust tools for data manipulation, statistical analysis, model building, and interactive visualization. Additionally, Python’s integration with Streamlit enables the rapid development of user-friendly web applications. Its widespread use in the finance and data science communities ensures ample resources, community support, and continuous improvements, making it an ideal choice for building a comprehensive stock market analytics platform.

**OBJECTIVES:**

**1. Data Retrieval:**

* Implement a system to fetch historical stock market data for specified symbols and date ranges using the Yahoo Finance API.
* Ensure the data retrieval process is efficient, reliable, and can handle potential errors or data unavailability gracefully.

**2. Technical Analysis:**

* Calculate and visualize key technical indicators such as Volume Weighted Average Price (VWAP), Moving Average Convergence Divergence (MACD), Relative Strength Index (RSI), Bollinger Bands, On-Balance Volume (OBV), Simple Moving Average (SMA), Exponential Moving Average (EMA), Average True Range (ATR), Stochastic Oscillator, and Chaikin Money Flow (CMF).
* Provide clear and informative visualizations for these indicators to assist users in understanding stock performance and trends.

**3. Comparative Analysis:**

* Enable users to compare the closing prices of multiple stocks over a specified date range.
* Provide visual tools to highlight differences and similarities in stock performance.

**4. Predictive Analytics:**

* Develop a machine learning model, specifically a Random Forest Regressor, to predict future stock closing prices based on user-selected features.
* Evaluate the model's performance using appropriate metrics such as Mean Squared Error (MSE) and R-squared (R²) scores.
* Create an interface for users to input feature values and obtain predicted stock prices.

**5. Data Preprocessing:**

* Implement binning techniques to categorize and analyze the distribution of selected features.
* Provide options for data normalization to ensure consistent scaling of features.
* Incorporate both random and stratified sampling techniques to generate representative data subsets for analysis and model training.

**6. Interactive Visualizations:**

* Utilize Plotly to create dynamic and interactive charts for various time series data and technical indicators.
* Develop animations for time series plots to enhance user engagement and facilitate a better understanding of data trends over time.

**7. User Experience:**

* Design a user-friendly interface using Streamlit that simplifies the interaction with complex financial data and analytical tools.
* Ensure the application is accessible to both novice and experienced traders, providing intuitive controls and informative outputs.

**8. Robustness and Reliability:**

* Ensure the application handles edge cases and potential errors gracefully, providing meaningful feedback to users in case of issues.
* Optimize the performance and responsiveness of the application to handle large datasets and complex computations efficient.

**OVERVIEW OF THE DATASET:**

This dataset encompasses the price history and trading volumes of fifty stocks listed in the NIFTY 50 index from the National Stock Exchange (NSE) of India. Spanning from January 1, 2000, to April 30, 2021, it provides a detailed daily record of pricing and trading values for each stock. The dataset covers a diverse range of companies, including ADANIPORTS, ASIANPAINT, AXISBANK, BAJAJ-AUTO, BAJAJFINSV, BAJFINANCE, BHARTIARTL, BPCL, BRITANNIA, CIPLA, COALINDIA, DRREDDY, EICHERMOT, GAIL, GRASIM, HCLTECH, HDFC, HDFCBANK, HEROMOTOCO, HINDALCO, HINDUNILVR, ICICIBANK, INDUSINDBK, INFRATEL, INFY, IOC, ITC, JSWSTEEL, KOTAKBANK, LT, MARUTI, MM, NESTLEIND, NTPC, ONGC, POWERGRID, RELIANCE, SBIN, SHREECEM, SUNPHARMA, TATAMOTORS, TATASTEEL, TCS, TECHM, TITAN, ULTRACEMCO, UPL, VEDL, WIPRO, and ZEEL.

**1. Date:** The date of the trading day, providing a chronological timeline of stock market activity.

**2. Symbol:** The unique symbol or ticker assigned to each stock, facilitating identification and trading on the exchange.

**3. Series:** The categorization of the stock based on its type, such as equity shares or preference shares, which may impact trading regulations and investor rights.

**4. Prev Close:** The closing price of the stock on the previous trading day, serving as a reference point for analyzing price movements.

**5. Open:** The opening price of the stock at the beginning of the trading day, indicating the initial valuation at which trading commenced.

**6. High:** The highest price reached by the stock during the trading day, reflecting the peak value achieved by the stock.

**7. Low:** The lowest price reached by the stock during the trading day, representing the minimum value recorded for the stock.

**8. Last:** The last traded price of the stock during the trading day, providing the most recent valuation before the market closes.

**9. Close:** The closing price of the stock on the trading day, indicating the final valuation at the end of the trading session.

**10. VWAP:** Volume Weighted Average Price, calculated as the average price of the stock weighted by trading volume, providing insight into the average transaction price.

**11. Volume:** The total number of shares traded for the stock on the trading day, reflecting the level of market activity and liquidity.

**12. Turnover:** The total value of stocks traded for the stock on the trading day, representing the overall monetary value exchanged.

**13. Trades:** The total number of trades executed for the stock on the trading day, indicating the level of trading activity and investor participation.

**14. Deliverable Volume:** The volume of shares that are delivered (settled) on the trading day, providing insight into the actual transactions executed.

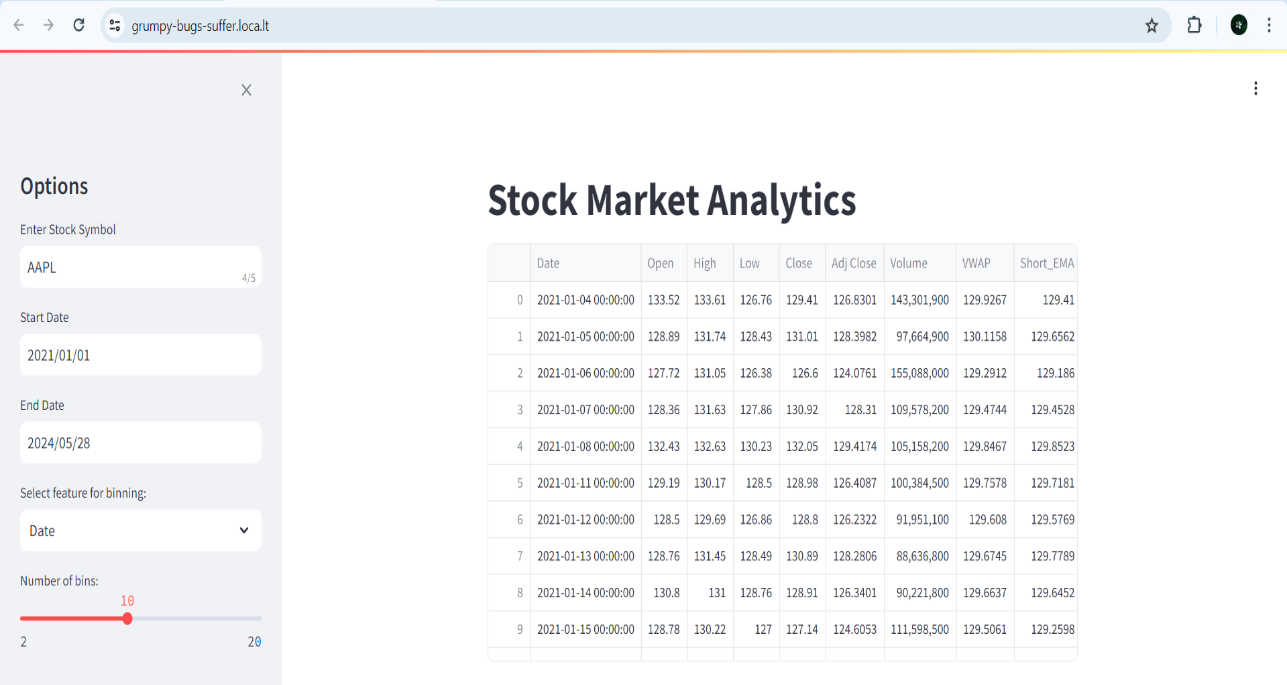
**15. %Deliverable:** The percentage of deliverable volume to total volume traded, indicating the proportion of shares that result in actual delivery.

**16. Company Name:** The name of the company associated with the stock, providing context and identification for the listed entity.

**17. Industry:** The industry or sector to which the company belongs, offering insight into the business operations and market positioning.

**18. ISIN Code:** The International Securities Identification Number assigned to the stock, providing a unique identifier for securities traded globally.

**DASHBOARD:**



The dashboard presents a comprehensive analysis of stock market data, offering users interactive features to explore the price history and trading volumes of various stocks listed in the NIFTY 50 index from the National Stock Exchange (NSE) of India. Users can input parameters like stock symbol, date range, and select analysis options such as feature selection, binning, normalization, and sampling techniques. Visualizations include dynamic charts illustrating closing prices, volume traded, MACD indicator, RSI, Bollinger Bands, and more, providing insights into stock performance over time. Additionally, users can utilize machine learning models to predict stock prices and evaluate model accuracy. Animated time series plots enable dynamic observation of trends, enhancing the dashboard's usability for data analysis and decision-making.

**CLOSING PRICE OVER TIME:**

**CODE:**

st.subheader("Closing Price Over Time")

fig\_close = go.Figure(data=go.Scatter(x=data['Date'], y=data['Close'], mode='lines'))

fig\_close.update\_layout(title=f"Closing Price of {symbol}")

st.plotly\_chart(fig\_close, use\_container\_width=True)

**DESCRIPTION:**

It’s a subplot in the Streamlit dashboard titled "Closing Price Over Time". It utilizes Plotly to generate a line plot (`go.Scatter`) representing the closing prices of a selected stock (`data['Close']`) over time (`data['Date']`). The title of the plot is dynamically set to "Closing Price of [selected stock symbol]". Finally, the plot is rendered using `st.plotly\_chart`, ensuring it fits within the width of the container.

****

**INFERENCE:**

The inference from the output of this code segment would be the trend of the closing prices of the selected stock over time. By visualizing the closing prices as a line plot, we can observe whether the stock's prices have been increasing, decreasing, or remaining relatively stable over the specified time period. This visualization helps in understanding the historical performance of the stock and identifying any patterns or trends that may exist.

**VOLUME TRADED OVER TIME:**

**CODE:**

st.subheader("Volume Traded Over Time")

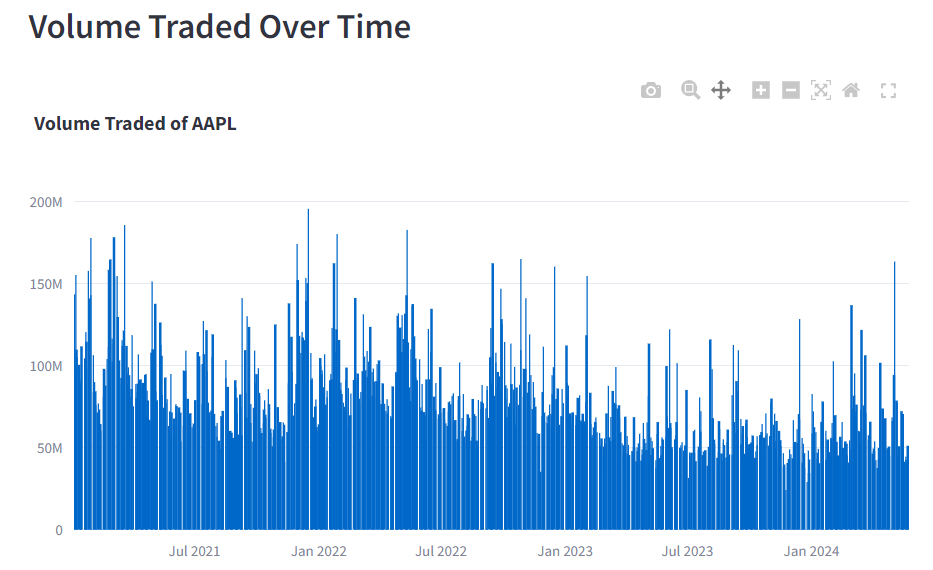
fig\_volume = go.Figure(data=go.Bar(x=data['Date'], y=data['Volume']))

fig\_volume.update\_layout(title=f"Volume Traded of {symbol}")

st.plotly\_chart(fig\_volume, use\_container\_width=True)

**DESCRIPTION:**

This plot focuses on visualizing the volume traded over time for a selected stock. It creates a subplot titled "Volume Traded Over Time" and uses Plotly to generate a bar chart (go.Bar) with the x-axis representing dates (data['Date']) and the y-axis representing the volume traded (data['Volume']). The title of the plot dynamically includes the selected stock symbol. Finally, the plot is rendered using st.plotly\_chart, ensuring it fits within the width of the container.

****

**INFERENCE:**

The inference from this code segment is the visualization of the volume traded over time for the selected stock symbol. The bar chart shows the volume of trades on the y-axis and the corresponding dates on the x-axis. By observing this chart, one can analyze the trading activity of the stock over the specified time period. Increases or decreases in trading volume can indicate shifts in market sentiment or increased investor interest, providing valuable insights into the stock's liquidity and trading patterns.

**OPENING PRIE VS CLOSING PRICE OVER TIME:**

**CODE:**

st.subheader("Opening vs Closing Prices Over Time")

fig\_open\_close = go.Figure()

fig\_open\_close.add\_trace(go.Scatter(x=data['Date'], y=data['Open'], mode='lines', name='Open'))

fig\_open\_close.add\_trace(go.Scatter(x=data['Date'], y=data['Close'], mode='lines', name='Close'))

fig\_open\_close.update\_layout(title='Opening vs Closing Prices Over Time')

st.plotly\_chart(fig\_open\_close, use\_container\_width=True)

**DESCRIPTION:**

It’s a subplot titled "Opening vs Closing Prices Over Time" in the Streamlit dashboard. It initializes a Plotly figure (`fig\_open\_close`) and adds two line plots using `go.Scatter`. The first plot represents the opening prices (`data['Open']`) over time (`data['Date']`), labeled as 'Open'. The second plot represents the closing prices (`data['Close']`) over the same time period, labeled as 'Close'. The subplot is then given a title, "Opening vs Closing Prices Over Time", and rendered using `st.plotly\_chart`, ensuring it fits within the width of the container.

****

**INFERENCE:**

This code segment is the comparison between the opening and closing prices of the selected stock symbol over time. The line chart displays the trend of both opening and closing prices on the y-axis against the corresponding dates on the x-axis. By analyzing this chart, one can gain insights into the price movements of the stock throughout the specified time period. Discrepancies or similarities between opening and closing prices can indicate intraday volatility, market sentiment, and potential trading opportunities.

**OHLC CHART:**

**CODE:**

st.subheader("OHLC Chart")

fig\_candlestick = go.Figure(data=[go.Candlestick(x=data['Date'],

open=data['Open'],

high=data['High'],

low=data['Low'],

close=data['Close'])])

fig\_candlestick.update\_layout(title='OHLC Chart')

st.plotly\_chart(fig\_candlestick, use\_container\_width=True)

**DESCRIPTION:**

The correct term is "OHLC" chart, which stands for Open-High-Low-Close chart. It is a type of financial chart used to illustrate the movements of a security's price over a specific time period. Each candlestick in an OHLC chart represents the open, high, low, and close prices of a security for a particular period, typically a day. The body of the candlestick shows the opening and closing prices, while the upper and lower "wicks" or "shadows" represent the highest and lowest prices reached during the period, respectively. These charts are commonly used in technical analysis to analyze price trends and make trading decisions.

****

**INFERENCE:**

It generates an OHLC (Open-High-Low-Close) chart, is a visualization of the price movement of the selected stock symbol over time. The OHLC chart displays the opening, high, low, and closing prices of the stock for each trading day. By examining the OHLC chart, one can observe the price range (from the lowest to the highest), as well as the opening and closing prices, providing insights into price volatility, market sentiment, and potential trading patterns such as trends, reversals, or consolidation periods.

**VWAP:**

**CODE:**

st.subheader("Volume Weighted Average Price (VWAP) Over Time")

fig\_vwap = go.Figure()

fig\_vwap.add\_trace(go.Scatter(x=data['Date'], y=data['VWAP'], mode='lines', name='VWAP'))

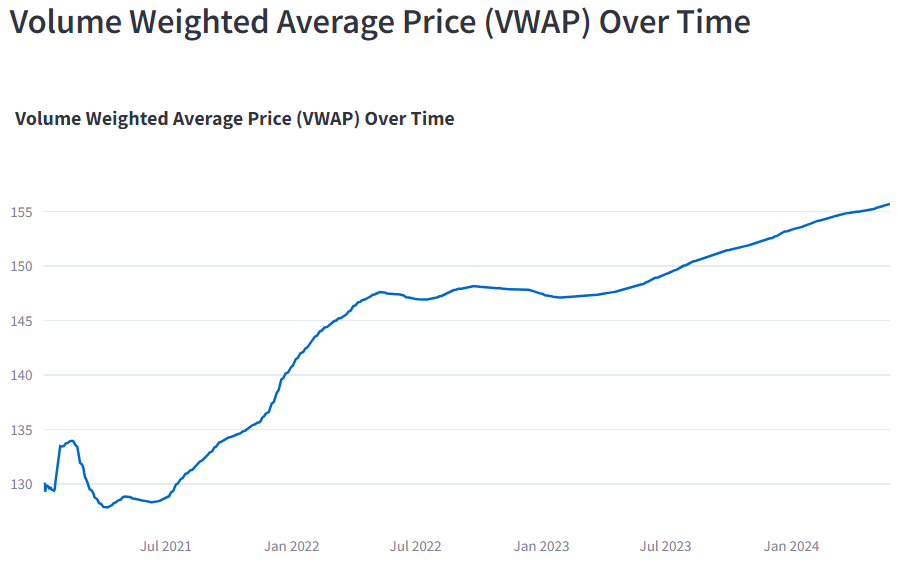
fig\_vwap.update\_layout(title='Volume Weighted Average Price (VWAP) Over Time')

st.plotly\_chart(fig\_vwap, use\_container\_width=True)

**DESCRIPTION:**

This code segment creates a plot of the Volume Weighted Average Price (VWAP) over time using Plotly. It begins by initializing a Plotly figure object (`fig\_vwap`). Next, a scatter plot is added to the figure using the `go.Scatter` function, where the x-axis represents the dates (`data['Date']`) and the y-axis represents the VWAP (`data['VWAP']`). The mode is set to 'lines' to connect the data points with lines.

The layout of the figure is updated to set the title to "Volume Weighted Average Price (VWAP) Over Time" using `fig\_vwap.update\_layout`. Finally, the plot is rendered in the Streamlit app using `st.plotly\_chart(fig\_vwap, use\_container\_width=True)`, which displays the plot with the specified layout settings within the Streamlit interface, utilizing the available container width.



**INFERENCE:**

The code segment generates a plot showing the Volume Weighted Average Price (VWAP) over time. VWAP is a trading benchmark used by traders to assess the average price at which a security has traded throughout the day, based on both volume and price. This visualization provides insights into the average price levels at which significant trading activity has occurred over the specified time period. Analyzing the VWAP over time can help traders understand price trends, support and resistance levels, as well as potential buying or selling opportunities based on deviations from the VWAP.

**TRADE ANALYSIS:**

**CODE:**

st.subheader("Number of Trades Over Time")

fig\_trades = go.Figure()

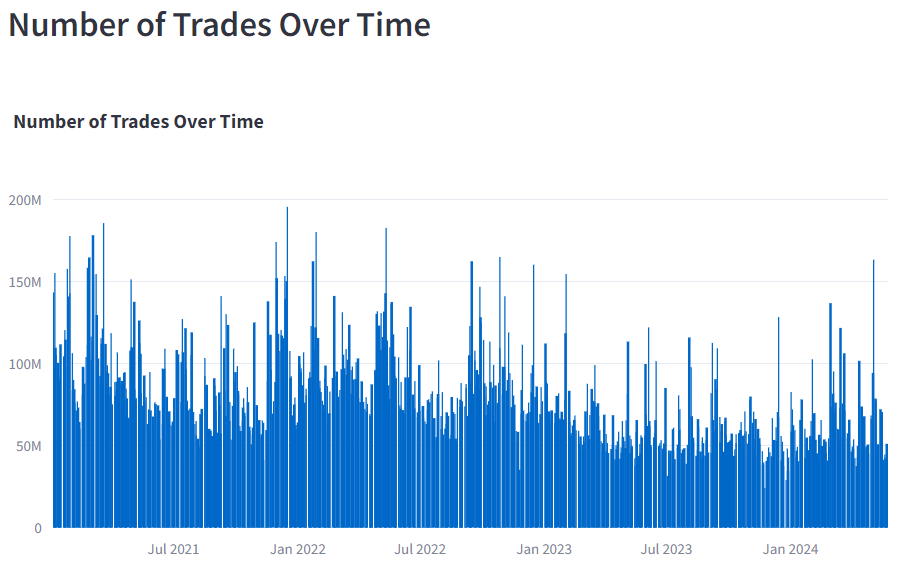
fig\_trades.add\_trace(go.Bar(x=data['Date'], y=data['Volume'], name='Trades'))

fig\_trades.update\_layout(title='Number of Trades Over Time')

st.plotly\_chart(fig\_trades, use\_container\_width=True)

**DESCRIPTION:**

It begins by creating a Plotly figure object (fig\_trades). Then, a bar plot is added to the figure using the go.Bar function, where the x-axis represents the dates (data['Date']) and the y-axis represents the number of trades (data['Volume']). The name of the bar is set to 'Trades'. The layout of the figure is updated to set the title to "Number of Trades Over Time" using fig\_trades.update\_layout. Finally, the plot is rendered in the Streamlit app using st.plotly\_chart(fig\_trades, use\_container\_width=True), which displays the bar chart within the Streamlit interface, utilizing the available container width.



**INFERENCE:**

Analyzing trades involves calculating metrics like daily returns, which show the percentage change in the stock price from one day to the next. This can help in understanding the stock's volatility and in developing trading strategies.

**COMPARATIVE ANALYSIS:**

**CODE:** symbols\_list = ['AAPL', 'MSFT', 'GOOGL'] # Add more symbols as needed

st.subheader("Comparative Closing Prices")

fig\_compare = go.Figure()

for symbol in symbols\_list:

compare\_data = load\_data(symbol, start\_date, end\_date)

fig\_compare.add\_trace(go.Scatter(x=compare\_data['Date'], y=compare\_data['Close'], mode='lines', name=symbol))

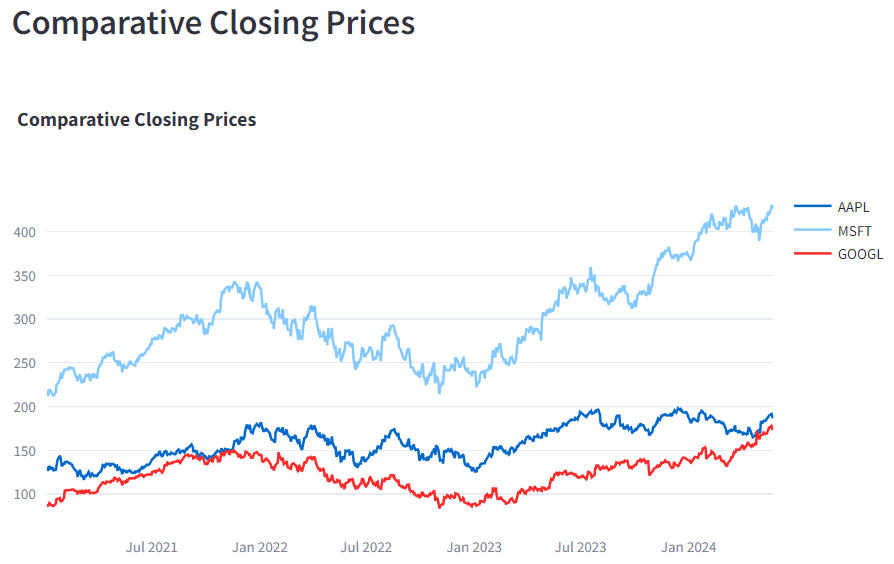
fig\_compare.update\_layout(title='Comparative Closing Prices')

st.plotly\_chart(fig\_compare, use\_container\_width=True)

**DESCRIPTION:**

It creates a Plotly figure object (fig\_compare) to visualize the comparative closing prices. Within a loop iterating over each symbol in the symbols\_list, it loads the data for each symbol using a function load\_data() with specified start and end dates. For each symbol, a line plot is added to the figure, where the x-axis represents the dates (compare\_data['Date']) and the y-axis represents the closing prices (compare\_data['Close']). The name of each line plot is set to the corresponding symbol.

After adding all the traces, the layout of the figure is updated to set the title to "Comparative Closing Prices". Finally, the plot is rendered in the Streamlit app using st.plotly\_chart(fig\_compare, use\_container\_width=True), which displays the comparative line chart within the Streamlit interface, utilizing the available container width.

****

**INFERENCE:**

Comparing the stock's performance against a benchmark, such as the S&P 500, helps in evaluating how well the stock is doing relative to the broader market. This can provide context to the stock's movements and aid in portfolio performance assessment.

**MACD INDICATORS:**

**CODE:** def calculate\_macd(data, short\_window=12, long\_window=26, signal\_window=9):

data['Short\_EMA'] = data['Close'].ewm(span=short\_window, adjust=False).mean()

data['Long\_EMA'] = data['Close'].ewm(span=long\_window, adjust=False).mean()

data['MACD'] = data['Short\_EMA'] - data['Long\_EMA']

data['Signal\_Line'] = data['MACD'].ewm(span=signal\_window, adjust=False).mean()

return data

data = calculate\_macd(data)

st.subheader("MACD Indicator")

fig\_macd = go.Figure()

fig\_macd.add\_trace(go.Scatter(x=data['Date'], y=data['MACD'], mode='lines', name='MACD'))

fig\_macd.add\_trace(go.Scatter(x=data['Date'], y=data['Signal\_Line'], mode='lines', name='Signal Line'))

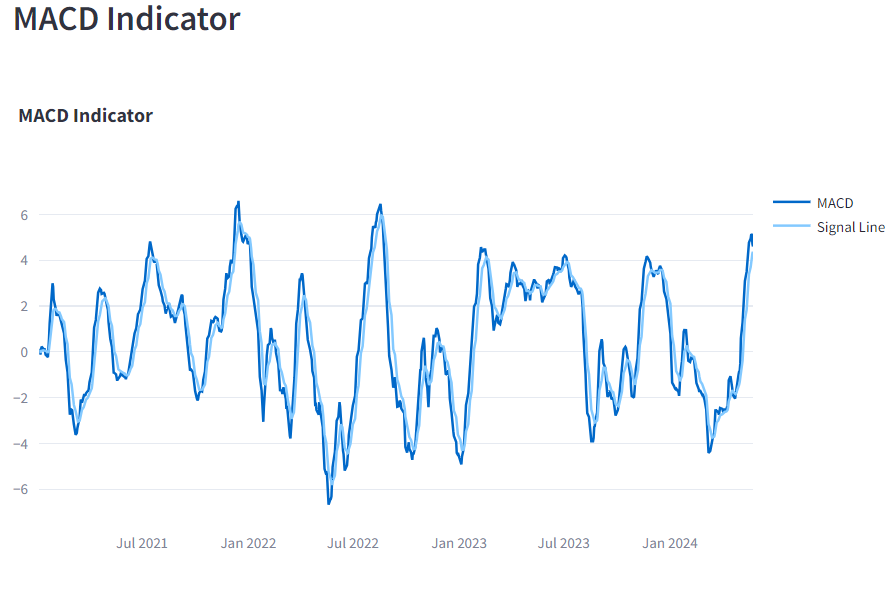
fig\_macd.update\_layout(title='MACD Indicator')

st.plotly\_chart(fig\_macd, use\_container\_width=True)

**DESCRIPTION:**

The calculate\_macd function takes the stock data as input along with optional parameters for the short window, long window, and signal window (default values are 12, 26, and 9, respectively). Within the function, it computes the Short EMA (Exponential Moving Average) and Long EMA for the closing prices based on the specified window sizes. Then, it calculates the MACD line by taking the difference between the Short EMA and Long EMA. Finally, it computes the Signal Line by applying an EMA to the MACD line.

After calculating the MACD indicator for the data, it plots the MACD line and Signal Line over time using a Plotly figure (fig\_macd). The x-axis represents the dates (data['Date']), and the y-axis represents the MACD values (data['MACD'] and data['Signal\_Line']).

****

**INFERENCE:**

**MACD Line**: The difference between the 12-day EMA and the 26-day EMA.

**Signal Line:** A 9-day EMA of the MACD Line.

A bullish signal occurs when the MACD crosses above the Signal Line, and a bearish signal occurs when it crosses below.

**RSI CHARTS:**

**CODE:**

def calculate\_rsi(data, window=14):

delta = data['Close'].diff(1)

gain = (delta.where(delta > 0, 0)).fillna(0)

loss = (-delta.where(delta < 0, 0)).fillna(0)

avg\_gain = gain.rolling(window=window).mean()

avg\_loss = loss.rolling(window=window).mean()

rs = avg\_gain / avg\_loss

data['RSI'] = 100 - (100 / (1 + rs))

return data

data = calculate\_rsi(data)

st.subheader("Relative Strength Index (RSI)")

fig\_rsi = go.Figure()

fig\_rsi.add\_trace(go.Scatter(x=data['Date'], y=data['RSI'], mode='lines', name='RSI'))

fig\_rsi.update\_layout(title='Relative Strength Index (RSI)', yaxis=dict(range=[0, 100]))

st.plotly\_chart(fig\_rsi, use\_container\_width=True)

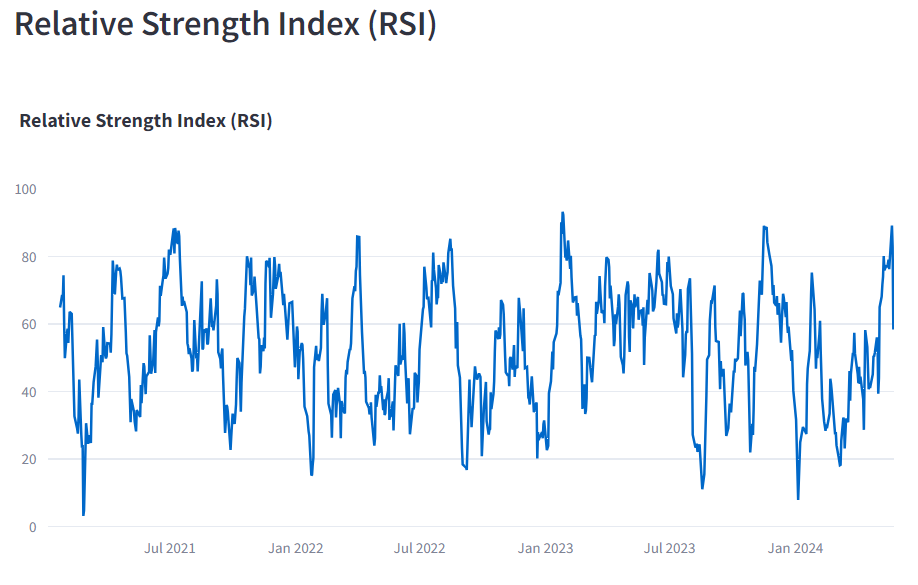
**DESCRIPTION:**

This code segment calculates the Relative Strength Index (RSI) for the given stock data. The calculate\_rsi function takes the stock data as input along with an optional parameter for the window size (default value is 14).

Within the function, it calculates the price differences (delta) between consecutive closing prices. It separates the gains and losses from the price differences and computes the average gain (avg\_gain) and average loss (avg\_loss) over the specified window size using the rolling mean. Then, it calculates the Relative Strength (RS) as the ratio of average gain to average loss.

**formula:**

RSI = 100 - (100 / (1 + RS)).

****

**INFERENCE:**

**Overbought:** RSI above 70 suggests the stock may be overbought.

**Oversold:** RSI below 30 suggests the stock may be oversold.

**BOLLINGER BANDS:**

**CODE:** def calculate\_bollinger\_bands(data, window=20):

data['MA20'] = data['Close'].rolling(window=window).mean()

data['STD20'] = data['Close'].rolling(window=window).std()

data['Upper\_Band'] = data['MA20'] + (data['STD20'] \* 2)

data['Lower\_Band'] = data['MA20'] - (data['STD20'] \* 2)

return data

data = calculate\_bollinger\_bands(data)

st.subheader("Bollinger Bands")

fig\_bollinger = go.Figure()

fig\_bollinger.add\_trace(go.Scatter(x=data['Date'], y=data['Close'], mode='lines', name='Close'))

fig\_bollinger.add\_trace(go.Scatter(x=data['Date'], y=data['Upper\_Band'], mode='lines', name='Upper Band'))

fig\_bollinger.add\_trace(go.Scatter(x=data['Date'], y=data['Lower\_Band'], mode='lines', name='Lower Band'))

fig\_bollinger.update\_layout(title='Bollinger Bands')

st.plotly\_chart(fig\_bollinger, use\_container\_width=True)

**DESCRIPTION:**

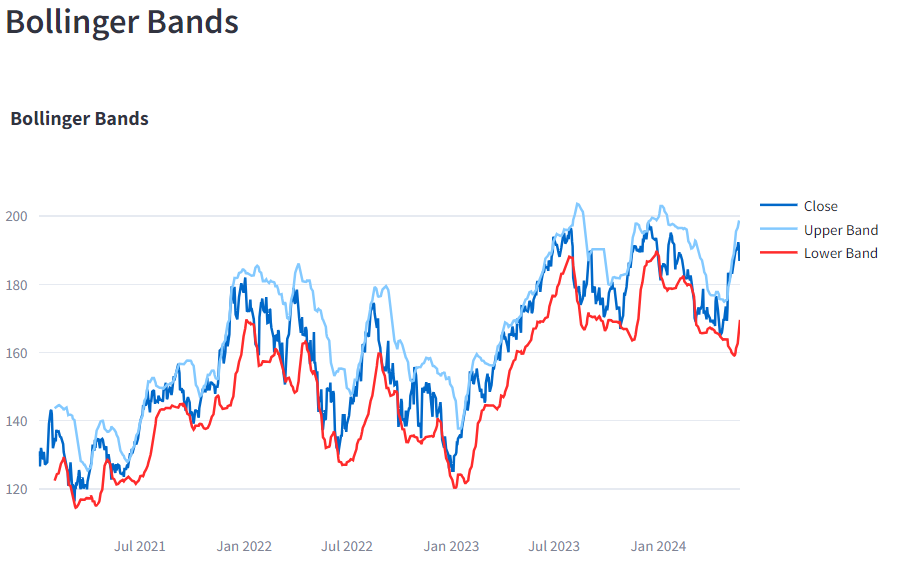
This code segment calculates the Bollinger Bands for the given stock data. The calculate\_bollinger\_bands function takes the stock data as input along with an optional parameter for the window size (default value is 20).

Within the function, it calculates the rolling mean (MA20) and rolling standard deviation (STD20) of the closing prices over the specified window size using the rolling method. Then, it computes the Upper and Lower Bollinger Bands using the formulas:

Upper Band = MA20 + (2 \* STD20)

Lower Band = MA20 - (2 \* STD20)

After calculating the Bollinger Bands for the data, it plots the closing prices, Upper Band, and Lower Band over time using a Plotly figure (fig\_bollinger). The x-axis represents the dates (data['Date']), and the y-axis represents the respective values. The closing prices are plotted as a line (mode='lines'), and the Upper and Lower Bands are also plotted as lines.

****

**INFERENCE:**

Price touching the upper band: Indicates the stock may be overbought.

Price touching the lower band: Indicates the stock may be oversold.

**TIME-SERIES DECOMPOSITION:**

**CODE:**

st.subheader("Time Series Decomposition")

decomposition = seasonal\_decompose(data['Close'].dropna(), model='multiplicative', period=30)

fig\_decomp = go.Figure()

fig\_decomp.add\_trace(go.Scatter(x=data['Date'], y=decomposition.trend, mode='lines', name='Trend'))

fig\_decomp.add\_trace(go.Scatter(x=data['Date'], y=decomposition.seasonal, mode='lines', name='Seasonal'))

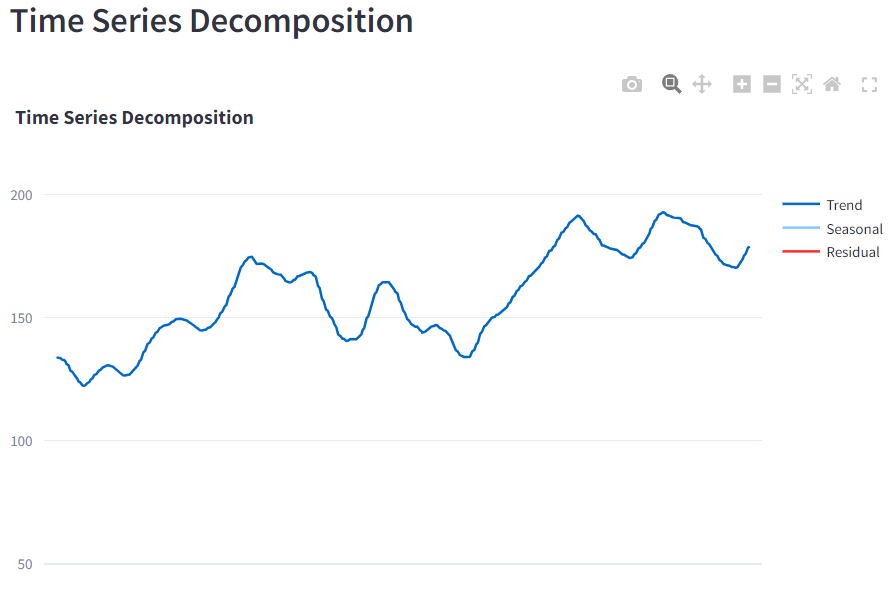
fig\_decomp.add\_trace(go.Scatter(x=data['Date'], y=decomposition.resid, mode='lines', name='Residual'))

fig\_decomp.update\_layout(title='Time Series Decomposition', height=600)

st.plotly\_chart(fig\_decomp, use\_container\_width=True)

**DESCRIPTION:**

The code snippet performs time series decomposition on the 'Close' price data within a Streamlit app. It uses a multiplicative model to break down the series into trend, seasonal, and residual components, assuming a periodicity of 30 days. The results are plotted using Plotly, displaying each component as a separate line trace in a figure titled "Time Series Decomposition." The figure is adjusted to fit the container width for clear visualization.



**INFERENCE:**

The time series decomposition of the 'Close' price data reveals distinct components: the trend shows a general upward movement with some fluctuations, the seasonal component captures regular patterns or cycles, and the residual component reflects random noise and irregular variations. This breakdown helps identify underlying patterns in the data, separating long-term trends from seasonal effects and noise, thereby providing insights into the price movements.

**ON-BALANCE -VOLUME:**

**CODE:**

def calculate\_obv(data):

data['OBV'] = (np.sign(data['Close'].diff()) \* data['Volume']).fillna(0).cumsum()

return data

data = calculate\_obv(data)

st.subheader("On-Balance Volume (OBV)")

fig\_obv = go.Figure()

fig\_obv.add\_trace(go.Scatter(x=data['Date'], y=data['OBV'], mode='lines', name='OBV'))

fig\_obv.update\_layout(title='On-Balance Volume (OBV)')

st.plotly\_chart(fig\_obv, use\_container\_width=True)

**DESCRIPTION:**

This code segment calculates the On-Balance Volume (OBV) for the given stock data. The `calculate\_obv` function takes the stock data as input.

Within the function, it computes the OBV values using the following steps:

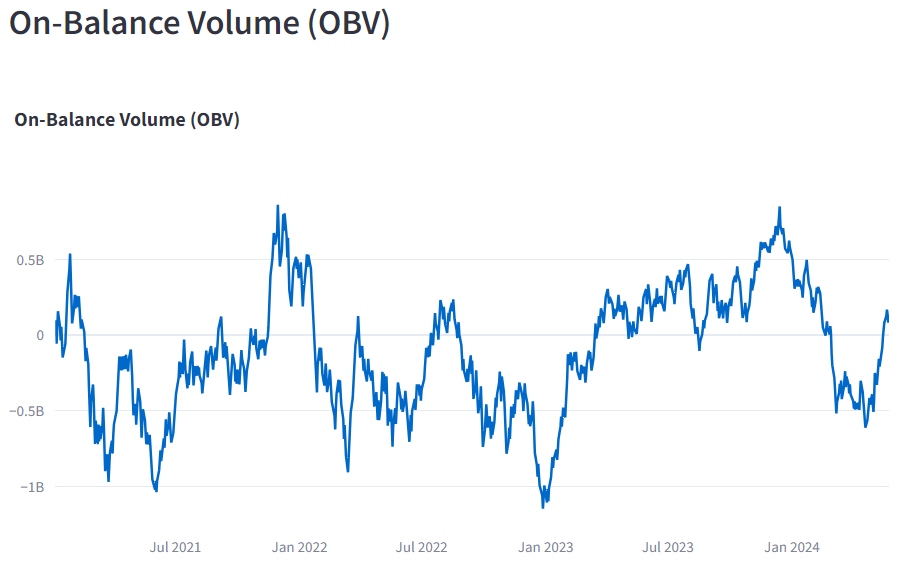
1. Calculates the difference in closing prices (`data['Close'].diff()`) and applies the `np.sign` function to get the sign of each difference. This indicates whether the closing price increased (positive value) or decreased (negative value) compared to the previous day.

2. Multiplies the sign of the price difference by the volume traded (`data['Volume']`) for each day to determine whether the volume should be added or subtracted based on the price movement.

3. Fills any NaN values resulting from the first calculation with 0 and then computes the cumulative sum of the OBV values over time using `cumsum`.

After calculating the OBV for the data, it plots the OBV values over time using a Plotly figure (`fig\_obv`). The x-axis represents the dates (`data['Date']`), and the y-axis represents the OBV values. The OBV is plotted as a line (`mode='lines'`).

The layout of the figure is updated to set the title to "On-Balance Volume (OBV)". Finally, the OBV chart is rendered in the Streamlit app using `st.plotly\_chart(fig\_obv, use\_container\_width=True)`, which displays the chart within the Streamlit interface, utilizing the available container width.



**INFERENCE:**

OBV is a momentum indicator that uses volume flow to predict changes in stock price. It adds volume on up days and subtracts volume on down days. The layout of the figure is updated to set the title to "On-Balance Volume (OBV)". Finally, the OBV chart is rendered in the Streamlit app using st.plotly\_chart(fig\_obv, use\_container\_width=True), which displays the chart within the Streamlit interface, utilizing the available container width.

**Rising OBV:** Indicates buying pressure.

**Falling OBV:** Indicates selling pressure.

**SIMPLE MOVING AVERAGE:**

**CODE:** def calculate\_sma(data, window):

data[f'SMA\_{window}'] = data['Close'].rolling(window=window).mean()

return data

data = calculate\_sma(data, window=50)

data = calculate\_sma(data, window=200)

st.subheader("Simple Moving Average (SMA)")

fig\_sma = go.Figure()

fig\_sma.add\_trace(go.Scatter(x=data['Date'], y=data['Close'], mode='lines', name='Close'))

fig\_sma.add\_trace(go.Scatter(x=data['Date'], y=data['SMA\_50'], mode='lines', name='SMA 50'))

fig\_sma.add\_trace(go.Scatter(x=data['Date'], y=data['SMA\_200'], mode='lines', name='SMA 200'))

fig\_sma.update\_layout(title='Simple Moving Average (SMA)')

st.plotly\_chart(fig\_sma, use\_container\_width=True)

**DESCRIPTION:**

This code calculates the Simple Moving Average (SMA) for a given stock dataset. The `calculate\_sma` function computes the SMA with a specified window size.

Here's how it works:

1. The `calculate\_sma` function takes two parameters: the `data` dataframe and the `window` size for the SMA.

2. Within the function, it computes the SMA for the closing prices (`data['Close']`) using the `rolling` method with the specified window size.

3. The calculated SMA values are stored in new columns named based on the window size (e.g., `SMA\_50`, `SMA\_200`).

4. After calculating the SMA for both the 50-day and 200-day windows, the function returns the updated dataframe.

The SMA is calculated twice in this case, first with a 50-day window and then with a 200-day window. This is a common practice in technical analysis for comparing short-term and long-term trends.

After calculating the SMAs, the code plots the closing prices along with the SMAs on a Plotly figure (`fig\_sma`). It adds three traces to the plot:

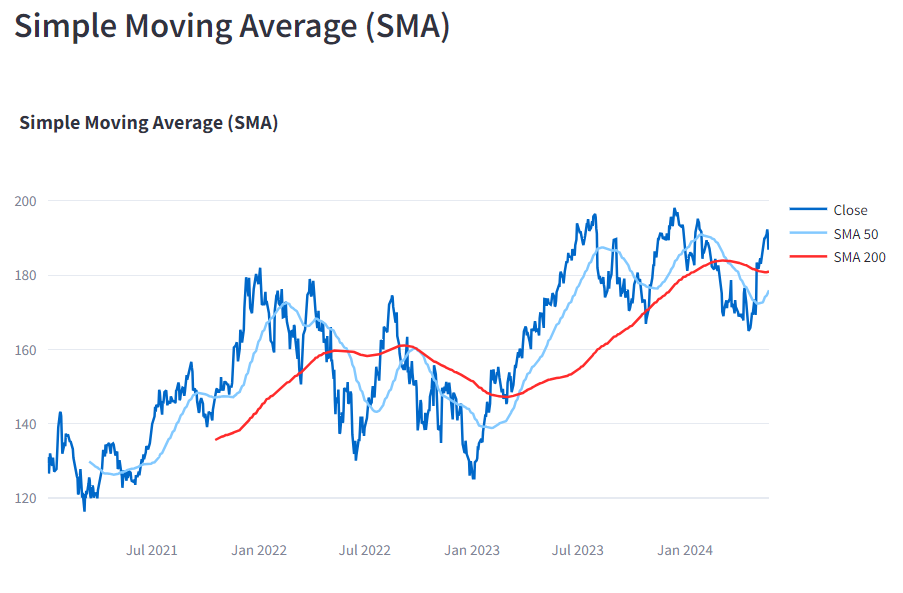
- The closing prices as a line (`mode='lines'`, name='Close').

- The 50-day SMA as a line (`mode='lines'`, name='SMA 50').

- The 200-day SMA as a line (`mode='lines'`, name='SMA 200').

The layout of the figure is updated to set the title to "Simple Moving Average (SMA)".

Finally, the SMA chart is rendered in the Streamlit app using `st.plotly\_chart(fig\_sma, use\_container\_width=True)`, which displays the chart within the Streamlit interface, utilizing the available container width.

****

**INFERENCE:**

SMA is the average stock price over a specified number of periods. It smooths out price data to identify trends.

**Short-term SMA:** More sensitive to recent price changes.

**Long-term SMA:** More stable and less sensitive to short-term fluctuations.

**CORRELATION HEATMAP:**

**CODE:**

st.subheader("Correlation Heatmap")

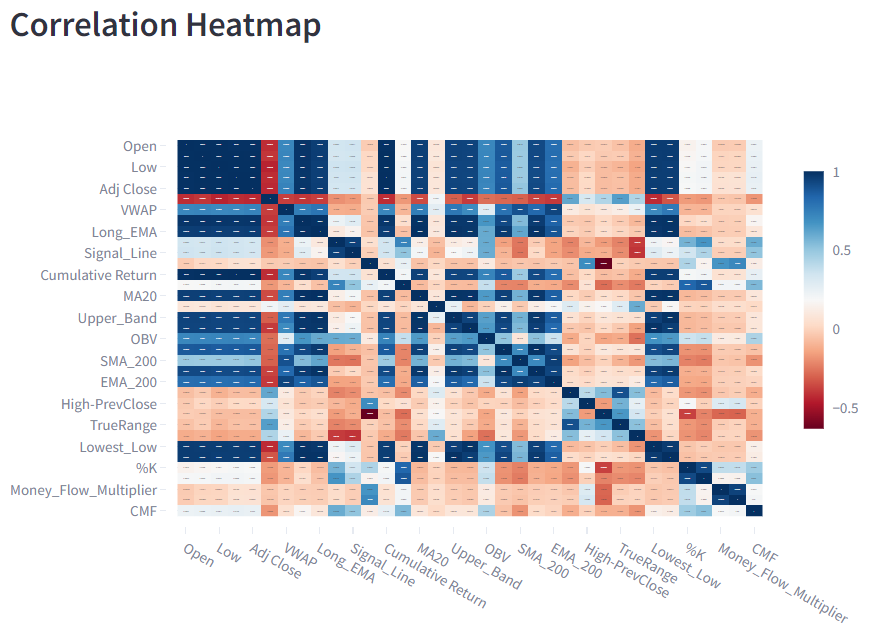
corr\_matrix = data[numeric\_data].corr()

fig\_heatmap = px.imshow(corr\_matrix, text\_auto=True, aspect="auto", color\_continuous\_scale='RdBu')

st.plotly\_chart(fig\_heatmap, use\_container\_width=True)

**DESCRIPTION**:

The code snippet creates and displays a correlation heatmap for numerical data within a Streamlit app. It calculates the correlation matrix for the numerical columns in the dataset, then uses Plotly Express to generate a heatmap with the 'RdBu' color scale. The heatmap includes auto-generated text annotations and is adjusted to fit the container width for better visualization.



**INFERENCE:**

The correlation heatmap shows the relationships between different numerical features in the dataset. High positive correlations (closer to 1, shown in dark blue) are observed between some moving averages and related indicators, indicating they move together. For example, 'MA20', 'SMA\_200', and 'EMA\_200' have strong positive correlations with each other. Conversely, high negative correlations (closer to -1, shown in dark red) indicate inverse relationships, such as between 'Signal\_Line' and 'CMF'. Features like 'TrueRange' and 'Low' exhibit moderate positive correlations with 'Open' and 'High-PrevClose'. The heatmap reveals these relationships, highlighting which features are closely related and which are inversely related, aiding in understanding the interdependencies within the data.

**PAIR PLOT:**

**CODE:**

st.subheader("Pair Plot")

# Select fewer important features to avoid congestion

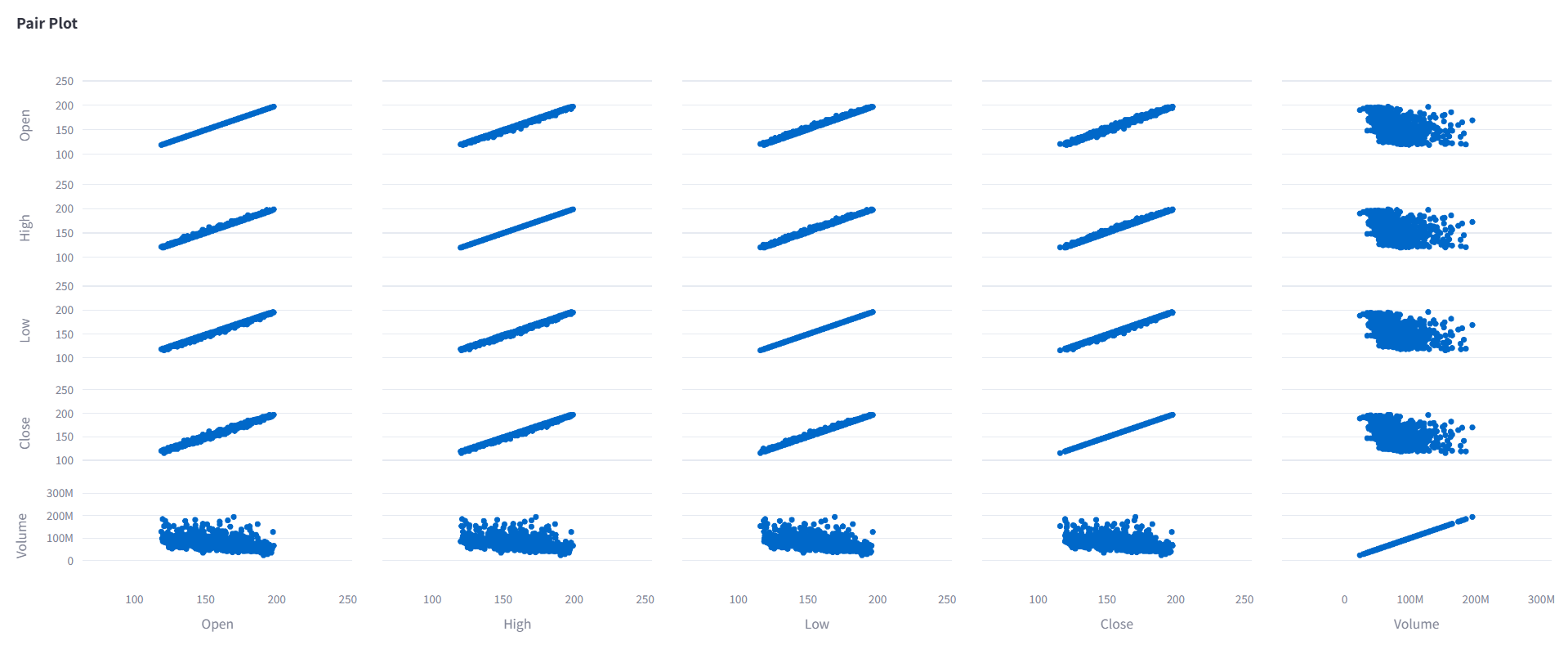
selected\_columns = numeric\_d[['Open', 'High', 'Low', 'Close', 'Volume']]

fig\_pairplot = px.scatter\_matrix(selected\_columns, dimensions=selected\_columns.columns, title="Pair Plot", height=800, width=800)

st.plotly\_chart(fig\_pairplot, use\_container\_width=True)

**DESCRIPTION:**

The code snippet generates a pair plot to visualize relationships between selected numerical features ('Open', 'High', 'Low', 'Close', 'Volume') in the dataset. It uses Plotly Express to create a scatter matrix, showing pairwise scatter plots and distributions for these features. The plot is configured to have a title "Pair Plot" and dimensions of 800x800 pixels, and it is displayed in the Streamlit app with container width adjustment for optimal viewing.



**EXPONENTIAL MOVING AVERAGE(EMA):**

**CODE:**

def calculate\_ema(data, window):

data[f'EMA\_{window}'] = data['Close'].ewm(span=window, adjust=False).mean()

return data

data = calculate\_ema(data, window=50)

data = calculate\_ema(data, window=200)

st.subheader("Exponential Moving Average (EMA)")

fig\_ema = go.Figure()

fig\_ema.add\_trace(go.Scatter(x=data['Date'], y=data['Close'], mode='lines', name='Close'))

fig\_ema.add\_trace(go.Scatter(x=data['Date'], y=data['EMA\_50'], mode='lines', name='EMA 50'))

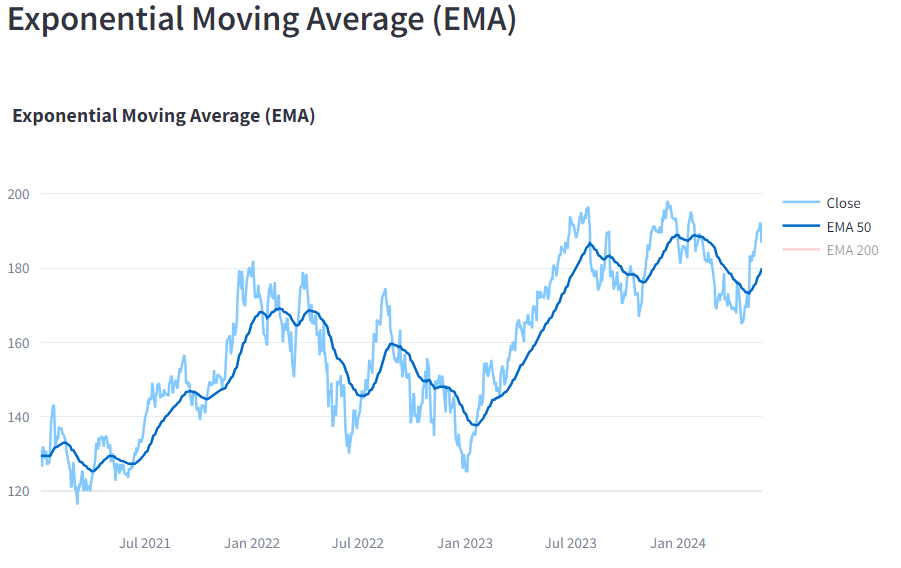
fig\_ema.add\_trace(go.Scatter(x=data['Date'], y=data['EMA\_200'], mode='lines', name='EMA 200'))

fig\_ema.update\_layout(title='Exponential Moving Average (EMA)')

st.plotly\_chart(fig\_ema, use\_container\_width=True)

**DESCRIPTION:**

In this segment, the Exponential Moving Average (EMA) is calculated and visually represented for a selected stock's closing prices within a Streamlit dashboard. Initially, the calculate\_ema function is defined to compute the EMA using the provided data and a specified window size. Specifically, EMAs are calculated for both a 50-day and a 200-day window. Subsequently, a subplot titled "Exponential Moving Average (EMA)" is generated using Plotly within the Streamlit dashboard. This subplot comprises three lines: one showcasing the stock's closing prices over time, alongside two others exhibiting the computed EMAs for the 50-day and 200-day windows, respectively. To enhance clarity, the plot is furnished with a descriptive title. Finally, the plot is displayed using st.plotly\_chart, ensuring its optimal fit within the Streamlit app's container width. Such visualization offers valuable insights into the underlying trend and smoothing patterns applied to the stock's price data through the EMA computation.



**INFERENCE:**

Based on the graph, the 50-day and 200-day exponential moving averages (EMAs) are both increasing over time. This suggests that the price of the asset is in an uptrend.

However, it is important to note that EMAs are lagging indicators, meaning that they react to price changes with a delay. So, while the EMAs in the image suggest that the price trend is up, they do not guarantee that the uptrend will continue in the future.

Here are some additional things to consider:

* The slope of the EMAs. A steeper slope suggests a stronger uptrend.
* Volatility. The EMAs will be more volatile if the price of the asset is volatile.
* Other technical indicators. EMAs are often used in conjunction with other technical indicators to confirm signals.

Overall, the exponential moving averages in the image suggest that the price of the asset is in an uptrend. However, it is important to consider other factors before making any investment decisions.

**AVERAGE TRUE RANGE:**

**CODE:**

def calculate\_atr(data, window=14):

data['High-Low'] = data['High'] - data['Low']

data['High-PrevClose'] = np.abs(data['High'] - data['Close'].shift(1))

data['Low-PrevClose'] = np.abs(data['Low'] - data['Close'].shift(1))

data['TrueRange'] = data[['High-Low', 'High-PrevClose', 'Low-PrevClose']].max(axis=1)

data['ATR'] = data['TrueRange'].rolling(window=window).mean()

return data

data = calculate\_atr(data)

st.subheader("Average True Range (ATR)")

fig\_atr = go.Figure()

fig\_atr.add\_trace(go.Scatter(x=data['Date'], y=data['ATR'], mode='lines', name='ATR'))

fig\_atr.update\_layout(title='Average True Range (ATR)')

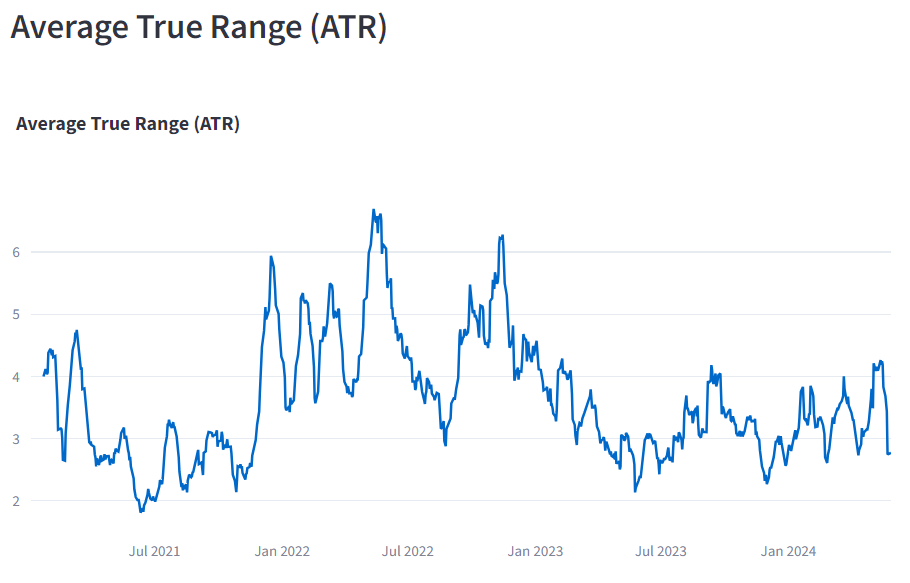
st.plotly\_chart(fig\_atr, use\_container\_width=True)

**DESCRIPTION:**

The provided code effectively calculates the Average True Range (ATR), a crucial technical analysis indicator that quantifies market volatility. By decomposing the range of price movement, the ATR provides traders and analysts with insights into price fluctuations over a specified period. The core of the calculation lies in the `calculate\_atr` function. This function processes a DataFrame containing stock price data, specifically columns for 'High', 'Low', and 'Close' prices. It computes the True Range (TR), which is the greatest of the following: the current high minus the current low, the absolute value of the current high minus the previous close, and the absolute value of the current low minus the previous close. This computation is repeated for each period in the dataset, and the rolling mean of these True Range values over a defined window (defaulting to 14 periods) yields the ATR.

Upon calculating the ATR, the code proceeds to visualize it using Plotly within a Streamlit application. A subheader titled "Average True Range (ATR)" is introduced to provide context. The visualization is achieved by creating a scatter plot where the x-axis represents the date and the y-axis denotes the ATR values. This graphical representation is dynamically rendered in the Streamlit app using the `st.plotly\_chart` function, which ensures an interactive and responsive display.

This implementation not only demonstrates the ATR's utility in assessing market volatility but also showcases how modern tools like Plotly and Streamlit can be harnessed to create insightful and user-friendly analytical dashboards. The ability to visualize ATR trends over time enables traders to identify periods of high and low volatility, aiding in the development of more informed trading strategies.



**INFERENCE:**

The Average True Range (ATR) chart indicates fluctuating market volatility from mid-2021 to early 2024, with periods of high volatility peaking around mid-2021 and early 2022, followed by reduced volatility mid-2022 to early 2023, and a resurgence in mid-2023. The recent trend towards early 2024 shows increasing volatility. These cyclical patterns suggest traders should be prepared for varying market conditions, with higher ATR values implying greater price movements and risk, necessitating cautious and adaptable trading strategies.

**STOCHASTIC OSCILLATOR:**

**CODE:**

def calculate\_stochastic\_oscillator(data, window=14):

data['Lowest\_Low'] = data['Low'].rolling(window=window).min()

data['Highest\_High'] = data['High'].rolling(window=window).max()

data['%K'] = 100 \* ((data['Close'] - data['Lowest\_Low']) / (data['Highest\_High'] - data['Lowest\_Low']))

data['%D'] = data['%K'].rolling(window=3).mean()

return data

data = calculate\_stochastic\_oscillator(data)

st.subheader("Stochastic Oscillator")

fig\_stochastic = go.Figure()

fig\_stochastic.add\_trace(go.Scatter(x=data['Date'], y=data['%K'], mode='lines', name='%K'))

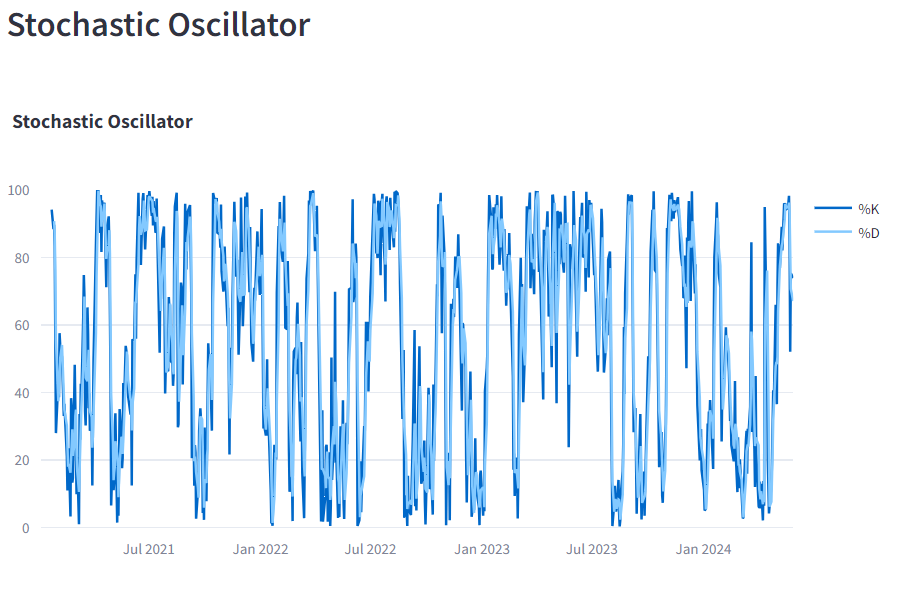
fig\_stochastic.add\_trace(go.Scatter(x=data['Date'], y=data['%D'], mode='lines', name='%D'))

fig\_stochastic.update\_layout(title='Stochastic Oscillator', yaxis=dict(range=[0, 100]))

st.plotly\_chart(fig\_stochastic, use\_container\_width=True)

**DESCRIPTION:**

Focusing on the Stochastic Oscillator computation and visualization within a Streamlit dashboard, this code segment commences with the definition of the calculate\_stochastic\_oscillator function. This function is pivotal for deriving the Stochastic Oscillator values based on the provided stock data. Utilizing rolling windows, it computes the lowest low and highest high prices over a specified period, typically set to 14 days. Subsequently, employing the formula %K = 100 \* ((Close - Lowest\_Low) / (Highest\_High - Lowest\_Low)), it calculates the %K value. Additionally, %D value is determined by taking the rolling mean of %K over a window of 3 periods. Once these Stochastic Oscillator values are computed, the code visualizes them within the Streamlit dashboard using Plotly. The resulting plot, labeled "Stochastic Oscillator," portrays both %K and %D values plotted against corresponding dates. For clarity, the y-axis range is constrained between 0 and 100. This visualization proves invaluable for evaluating momentum and identifying potential trend reversals in the selected stock's price movements over time.



**INFERENCE:**

The Stochastic Oscillator chart from mid-2021 to early 2024 shows frequent oscillations of the %K and %D lines between 0 and 100, indicating rapid changes in market momentum. The asset frequently enters overbought (>80) and oversold (<20) conditions, suggesting multiple potential buying and selling opportunities. The frequent crossings of %K and %D lines reflect a highly volatile market with numerous short-term fluctuations, providing traders with many signals for potential entries and exits based on momentum shifts.

**CHAIKIN MONEY FLOW:**

**CODE:**

def calculate\_cmf(data, window=20):

data['Money\_Flow\_Multiplier'] = ((data['Close'] - data['Low']) - (data['High'] - data['Close'])) / (data['High'] - data['Low'])

data['Money\_Flow\_Volume'] = data['Money\_Flow\_Multiplier'] \* data['Volume']

data['CMF'] = data['Money\_Flow\_Volume'].rolling(window=window).sum() / data['Volume'].rolling(window=window).sum()

return data

data = calculate\_cmf(data)

st.subheader("Chaikin Money Flow (CMF)")

fig\_cmf = go.Figure()

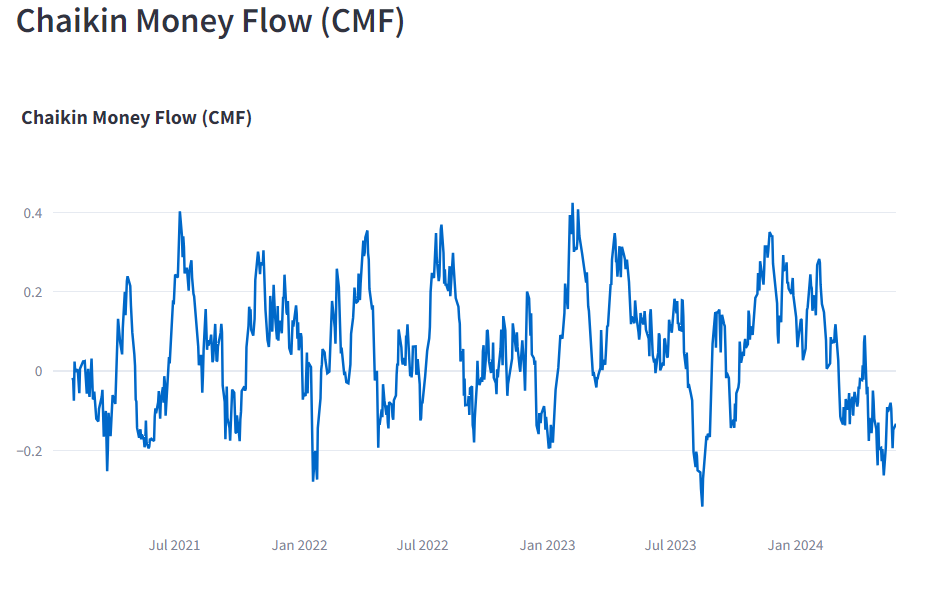
fig\_cmf.add\_trace(go.Scatter(x=data['Date'], y=data['CMF'], mode='lines', name='CMF'))

fig\_cmf.update\_layout(title='Chaikin Money Flow (CMF)')

st.plotly\_chart(fig\_cmf, use\_container\_width=True)

**DESCRIPTION:**

Focusing on the Chaikin Money Flow (CMF) computation and visualization within a Streamlit dashboard, this code segment initiates with the definition of the calculate\_cmf function. This function is pivotal for deriving the CMF values based on the provided stock data. It computes the Money Flow Multiplier, which represents the proportion of the day's trading activity occurring at the top or bottom of the day's price range. Subsequently, it calculates the Money Flow Volume, which is the Money Flow Multiplier multiplied by the day's volume. Utilizing rolling windows, the CMF is then determined by summing the Money Flow Volume over a specified period and dividing it by the sum of volumes over the same period. Once these CMF values are computed, the code visualizes them within the Streamlit dashboard using Plotly. The resulting plot, labeled "Chaikin Money Flow (CMF)," depicts the CMF values plotted against corresponding dates. For clarity, the plot's title is set to "Chaikin Money Flow (CMF)." This visualization is instrumental in assessing the flow of money into or out of the selected stock, aiding in identifying potential trends and market sentiment shifts over time.



**INFERENCE:**

The Chaikin Money Flow (CMF) chart from mid-2021 to early 2024 indicates fluctuating levels of buying and selling pressure, with values oscillating between 0.4 and -0.2. Positive CMF values suggest periods of accumulation (buying pressure), while negative values indicate distribution (selling pressure). The chart shows several peaks and troughs, reflecting shifts in market sentiment. Notably, there are more frequent and deeper dips into negative territory in recent months, suggesting increasing selling pressure and potential bearish sentiment in early 2024. Overall, the CMF highlights significant volatility and changes in market dynamics over the observed period.

**BINNING:**

**CODE:**

bin\_feature = st.sidebar.selectbox("Select feature for binning:", data.columns)

num\_bins = st.sidebar.slider("Number of bins:", min\_value=2, max\_value=20, value=10)

binned\_data, bin\_edges = np.histogram(data[bin\_feature].astype(float), bins=num\_bins)

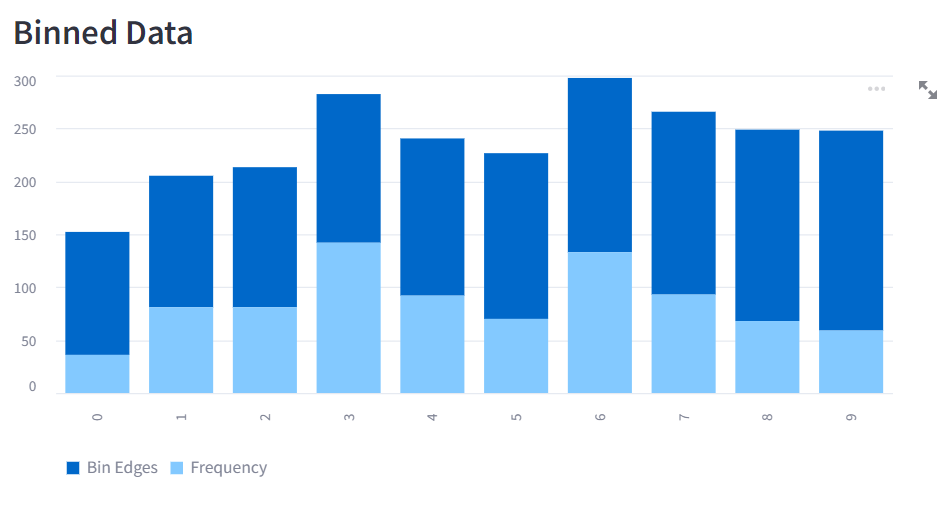
**DESCRIPTION:**

The given snippet integrates interactive elements into a Streamlit application, allowing users to perform binning on a selected feature from a dataset. Through Streamlit's sidebar widgets, users can dynamically choose a feature for binning and specify the number of bins to segment the data.

First, the `st.sidebar.selectbox` function creates a dropdown menu in the sidebar, populated with the column names of the `data` DataFrame. This dropdown allows users to select which feature (column) they want to bin. Next, a slider is implemented using `st.sidebar.slider`, enabling users to set the desired number of bins for the histogram, with a range from 2 to 20 and a default value of 10.

Once the user selects the feature and the number of bins, the code employs NumPy's `np.histogram` function to perform the binning. This function takes the selected feature (converted to float) and the specified number of bins, returning two outputs: `binned\_data`, which contains the counts of data points in each bin, and `bin\_edges`, which represents the edges of the bins.

This interactive approach allows users to customize their analysis directly within the Streamlit app, facilitating an intuitive exploration of data distributions and enabling more tailored insights.



**NORMALIZATION:**

**CODE:**

normalize\_data = st.sidebar.checkbox("Normalize data")

if normalize\_data:

scaler = MinMaxScaler()

data\_normalized = pd.DataFrame(scaler.fit\_transform(data), columns=data.columns)

else:

data\_normalized = data

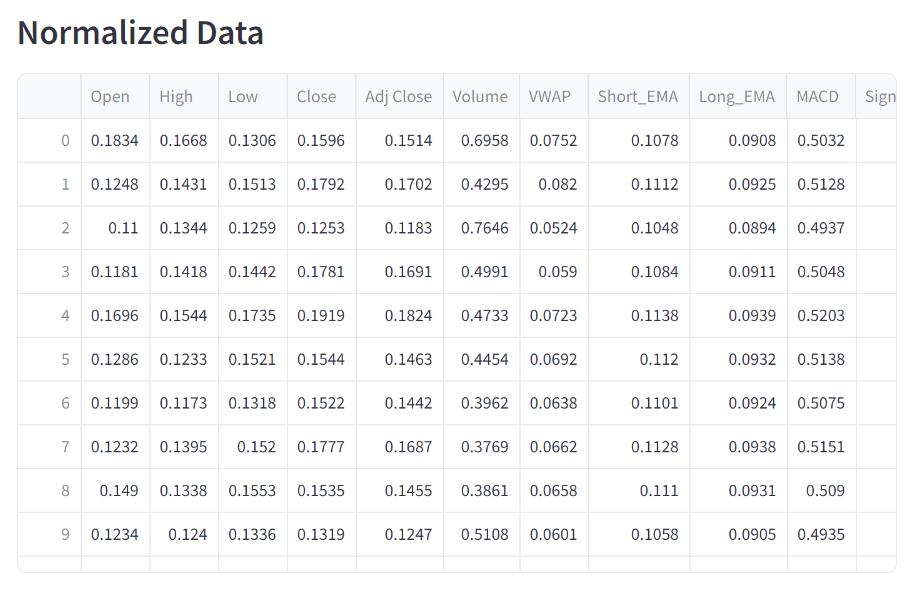
**DESCRIPTION:**

The given snippet integrates interactive elements into a Streamlit application, allowing users to perform binning on a selected feature from a dataset. Through Streamlit's sidebar widgets, users can dynamically choose a feature for binning and specify the number of bins to segment the data.

First, the `st.sidebar.selectbox` function creates a dropdown menu in the sidebar, populated with the column names of the `data` DataFrame. This dropdown allows users to select which feature (column) they want to bin. Next, a slider is implemented using `st.sidebar.slider`, enabling users to set the desired number of bins for the histogram, with a range from 2 to 20 and a default value of 10.

Once the user selects the feature and the number of bins, the code employs NumPy's `np.histogram` function to perform the binning. This function takes the selected feature (converted to float) and the specified number of bins, returning two outputs: `binned\_data`, which contains the counts of data points in each bin, and `bin\_edges`, which represents the edges of the bins.

This interactive approach allows users to customize their analysis directly within the Streamlit app, facilitating an intuitive exploration of data distributions and enabling more tailored insights.



**SAMPLING:**

**CODE:**

sampling\_technique = st.sidebar.selectbox("Sampling technique:", ["None", "Random Sampling", "Stratified Sampling"])

if sampling\_technique == "Random Sampling":

sample\_size = st.sidebar.slider("Sample size:", min\_value=1, max\_value=len(data), value=len(data)//2)

sampled\_data = data.sample(sample\_size)

elif sampling\_technique == "Stratified Sampling":

stratify\_by = st.sidebar.selectbox("Select feature for stratification:", data.columns)

if stratify\_by not in data.columns:

st.error(f"Selected column '{stratify\_by}' does not exist in the dataset.")

sampled\_data = data

else:

if sample\_size > len(data):

st.error("Sample size cannot be larger than the dataset size.")

sample\_size = len(data)

sampled\_data = data.groupby(stratify\_by, group\_keys=False).apply(lambda x: x.sample(min(len(x), sample\_size)))

else:

sampled\_data = data

**DESCRIPTION:**

The provided code snippet incorporates interactive elements in a Streamlit application, enabling users to select and apply different sampling techniques on a dataset. Using Streamlit's sidebar widgets, users can choose between "None", "Random Sampling", and "Stratified Sampling" to sample the data, adjusting parameters as needed.

The `st.sidebar.selectbox` function creates a dropdown menu in the sidebar, offering a selection of sampling techniques. If the user chooses "Random Sampling", a slider appears via `st.sidebar.slider`, allowing the user to specify the sample size. The `data.sample` method then randomly selects the specified number of rows from the dataset.

For "Stratified Sampling", another dropdown menu allows the user to choose a feature for stratification from the dataset's columns. If the selected column doesn't exist, an error message is displayed using `st.error`, and the original dataset is used. If the column is valid and the sample size does not exceed the dataset size, the dataset is grouped by the chosen feature, and the `sample` method is applied within each group to maintain proportional representation.

If "None" is selected, no sampling is applied, and the entire dataset is used as is. This interactive setup allows users to dynamically apply sampling techniques within the Streamlit app, facilitating flexible and tailored data analysis.

