STOCK ANALYSIS PACKAGE

GROUP 8:

SUBHAM MODA, SHUBHAM NARKHEDE, YANYUN WANG, XINRAN LIU, ZIXIN ZHOU

Overview

The Stock Analysis Package is a powerful and user-friendly Python-based tool meticulously crafted to empower investors and financial analysts with comprehensive insights for informed decision-making in the dynamic world of stock markets. Leveraging the capabilities of the yfinance library, specifically the yf.Ticker module, this package facilitates seamless data fetching, ensuring users have access to real-time market data. With a keen focus on user empowerment, the package encompasses a diverse set of features ranging from intuitive candlestick visualizations to sophisticated technical analysis tools like moving averages, MACD analysis, Bollinger Bands, RSI and fetching latest news. This holistic approach aims to provide users with a robust toolkit for delving into historical stock trends, allowing them to make informed investment decisions. Additionally, the package goes beyond technical indicators by incorporating the latest news relevant to the selected stocks, further enhancing its utility in the fast-paced world of financial markets. In essence, the Stock Analysis Package serves as a comprehensive and indispensable resource for individuals seeking a deeper understanding of market dynamics and trends, ultimately guiding them towards more informed and strategic investment choices.

Implementation

The entirety of the code for this module is encapsulated within the StockAnalysisPackage. Users can effortlessly import this package and access its diverse functionalities. Subsequently, we will comprehensively elucidate each functionality embedded in the code.

Import

```
import pandas as pd
import numpy as np
import yfinance as yf
import matplotlib.pyplot as plt
from datetime import datetime, date
import plotly.graph_objects as go
```

We import the above python packages which help us in achieving the desired outcomes.
Pandas - we use pandas to store the data that we retrieve and perform further analysis and store the results of the same.

Numpy - helps us in performing various mathematical tasks quickly and easily.
Yfinance – It is yahoo's package that helps us in fetching the real-time as well as historic data for stocks.
Mathplotlib – We initially were going to use this package to visualize all our graphs, but later updated it with Plotly.
Plotly – We use this package to visualize all our graphs, as it provides interactive graphs where user can interact with the graph or just hover over a certain point in the graph to see the instant's values.

StockAnalysisPackage

```
class StockAnalysisPackage:
    def __init__(self, symbol, start_date, end_date):
      self.symbol = symbol
      self.start_date = start_date
      self.end_date = end_date
      self.stock_data = None
```

Parameter: symbol, start_date, end_date
Output: Object

In the initialization of our package class, three essential input parameters—namely, 'symbol,' 'start_date,' and 'end_date'—are required from the user upon creating an object of this class. The 'symbol' parameter expects a string value representing the ticker or stock name in the stock market. The 'start_date' parameter takes a string value in the format 'yyyy-mm-dd,' determining the commencement date from which the data is to be fetched. Similarly, the 'end_date' parameter accepts a string value in the 'yyyy-mm-dd' format, indicating the termination date until which the data should be fetched. These user-provided values are stored within the class and are subsequently utilized by various functions for their respective tasks. The class also initializes the 'stock_data' variable, which serves as a storage container for the fetched data in the form of a dataframe, to be utilized by subsequent functions after the 'fetch_stock_data' function is invoked.

fetch_stock_data

```
def fetch_stock_data(self):

      self.ticker = yf.Ticker(self.symbol)
      info = None
      today_date = date.today()

      try:
        info = self.ticker.info
        quoteType = info['quoteType']
```

```
        if quoteType != 'EQUITY':    # Check if the ticker mentioned is a
stock ticker or not
          raise Exception("Symbol Error: Ticker does not belong to
stock/equity")
        else:
          list_date =
datetime.utcfromtimestamp(info['firstTradeDateEpochUtc']).strftime('%Y-%m-
%d')
          if self.start_date < list_date:    # Check if the start date is
before the list date of stock
            raise Exception("Date Error: Start Date before stock List
Date")
          elif self.end_date > str(today_date):    # Check if the end date
is after today's date
            raise Exception("Date Error: End Date after today's date")
          else:
            self.stock_data = yf.download(self.symbol,
start=self.start_date, end=self.end_date)
            self.stock_data["Date"] = self.stock_data.index
            self.stock_data = self.stock_data[["Date", "Open",
"High","Low", "Close", "Adj Close", "Volume"]]
            self.stock_data.reset_index(drop=True, inplace=True)
            print("Stock data fetched successfully.")
            print(self.stock_data.info())
      except Exception as e:
        if "404 Client Error" in str(e):
          print("Ticker {} does not exist.".format(self.symbol))
        elif "Date" in str(e):
          print(e)
        elif "Symbol" in str(e):
          print(e)
```

Parameter: None
Output: Object

Upon calling the 'fetch_stock_data' function, a thorough examination of the parameters provided by the user during the instantiation of the class object ensues. The initial step involves validating that the symbol submitted by the user corresponds to a legitimate equity or stock ticker, excluding other forms of tickers like mutual funds. If the provided ticker is confirmed as valid for equity, the process proceeds; otherwise, an exception for symbol error is raised. Subsequently, the 'start_date' undergoes scrutiny to ensure its validity and ascertain that it does not precede the stock's listing date. Should the date precede the listing, an exception for date error is raised. Simultaneously, a check is implemented to ensure that the 'end_date' is not set to a future date; if it is, a date error exception is raised. Once these checks are successfully navigated, the system proceeds to examine for potential connection errors. If, for any reason, the connection is

disrupted, or data retrieval from Yahoo's servers encounters an issue, a client error exception is raised. In optimal conditions, when all user parameters are deemed correct and the system operates smoothly, the 'fetch_stock_data' function utilizes 'yf.download' to retrieve the data, storing it in the 'stock_data' variable as a dataframe with columns Date, Open, High, Low, Close, Adj Close, and Volume.

visualize_candlestick

```
def visualize_candlestick(self, plot_start = None, plot_end = None):

      plot_start = plot_start or self.start_date
      plot_end = plot_end or self.end_date

      try:
        if plot_start < self.start_date:
          raise Exception("Date Error: Plot Start date provided is before
the Data Fetch start date")
        elif plot_end > self.end_date:
          raise Exception("Date Error: Plot End date provided is after the
Data Fetch end date")
        else:
          plot_data = self.stock_data[(self.stock_data['Date'] >=
plot_start) & (self.stock_data['Date'] <= plot_end)]

          fig =
go.Figure(data=[go.Candlestick(x=plot_data['Date'],open=plot_data['Open'],
\

high=plot_data['High'],low=plot_data['Low'], close=plot_data['Close'])])
          fig.update_layout(xaxis_rangeslider_visible=False)
          fig.show()
      except Exception as e:
        print(e)
```

Parameter: plot_start (Optional), plot_end (Optional)
Output: Graph

This function offers users the capability to visualize the entire stock data using candlesticks, a widely employed visualization method in financial markets for price representation. The function accommodates two optional parameters, namely 'plot_start' and 'plot_end,' allowing users to specify the timeframe for visualizing the data. It is imperative that these dates fall within the range established by the 'start_date' and 'end_date' provided during the object's initialization. Checks are implemented to ensure compliance, and if the specified dates extend beyond this range, a date error exception is raised. In instances where users opt not to provide optional

parameters, the function defaults to visualizing the complete dataset. To ensure seamless functionality of the plot, the presence of any issues with the Plotly package is also addressed. Checks have been integrated to handle potential package-related errors, with specific exceptions raised and displayed accordingly.

calculate_moving_average

```
def calculate_moving_average(self, window = 21):
      self.window = window

      try:
          # Calculate moving average
          self.stock_data['MA'] =
self.stock_data['Close'].rolling(window=window).mean()
          print(f"Moving average (window={window}) calculated
successfully.")
      except Exception as e:
          print(f"Error calculating moving average: {e}")
```

Parameter: window (Default = 12, Optional)
Output: Object

This function is designed to compute the moving average over the stock data. It accepts an optional integer parameter, referred to as 'window,' with a default value set at 21. Users have the flexibility to calculate the moving average over a different number of periods by specifying their preferred 'window' value. If any errors arise during the moving average calculation, the function raises an exception. The resulting value is then computed and stored as a new column in the 'stock_data' dataframe, named 'MA.'

visualize_moving_average

```
def visualize_moving_average(self, plot_start = None, plot_end = None):

      plot_start = plot_start or self.start_date
      plot_end = plot_end or self.end_date

      try:
        if plot_start < self.start_date:
          raise Exception("Date Error: Plot Start date provided is before
the Data Fetch start date")
        elif plot_end > self.end_date:
          raise Exception("Date Error: Plot End date provided is after the
Data Fetch end date")
        else:
```

```
          plot_data = self.stock_data[(self.stock_data['Date'] >=
plot_start) & (self.stock_data['Date'] <= plot_end)]

          candlestick_trace = go.Candlestick(x=plot_data['Date'],
                                   open=plot_data['Open'],
                                   high=plot_data['High'],
                                   low=plot_data['Low'],
                                   close=plot_data['Close'],
                                   name='Value')
          ma_trace = go.Scatter(x=plot_data['Date'],
                                y=plot_data['MA'],
                                mode='lines',
                                name=f'Moving Average ({self.window}-
day)',
                                line=dict(color="#0000ff"))

          layout = go.Layout(title='Candlestick Chart with Moving
Average',
                             xaxis=dict(title='Date'),
                             yaxis=dict(title='Price'))

          fig = go.Figure(data=[candlestick_trace, ma_trace],
layout=layout)
          fig.update_layout(xaxis_rangeslider_visible=False)

          fig.show()
      except Exception as e:
          print(e)
```

Parameter: plot_start (Optional), plot_end (Optional)
Output: Graph

This function serves the purpose of visualizing the data in conjunction with the moving average, presenting the price data as candlesticks and the moving average line in blue. Leveraging the 'MA' column generated in the preceding function, this visualization function also incorporates optional parameters, namely 'plot_start' and 'plot_end.' Similar to 'visualize_candlestick,' these parameters undergo conditional checks to ensure their alignment within the 'start_date' and 'end_date' range. If users choose not to provide these parameters, the graph is generated for the entire dataset. Additionally, the function incorporates error-handling mechanisms to catch and display any potential errors arising from other packages during the visualization process.

perform_macd

```
def perform_macd(self, short_window=13, long_window=33, signal_window=9):
      try:
```

```
            # Calculate short-term and long-term exponential moving averages
            short_ema = self.stock_data['Close'].ewm(span=short_window,
adjust=False).mean()
            long_ema = self.stock_data['Close'].ewm(span=long_window,
adjust=False).mean()

            # Calculate MACD and signal line
            self.stock_data['MACD'] = short_ema - long_ema
            self.stock_data['Signal_Line'] =
self.stock_data['MACD'].ewm(span=signal_window,\

adjust=False).mean()

            print("MACD analysis performed successfully.")
        except Exception as e:
            print(f"Error performing MACD analysis: {e}")
```

Parameter: short_window (Default = 13, Optional), long_window (Default = 33, Optional), signal_window (Default = 9, Optional)
Output: Object

The 'perform_macd' function is employed to compute Moving Average Convergence/Divergence (MACD) values based on specified window configurations. This function features three optional parameters: 'short_window' (default value: 13), 'long_window' (default value: 33), and 'signal_window' (default value: 9). Users have the flexibility to calculate MACD values over different window configurations by providing their desired values in the corresponding order when invoking the function. The function ensures the accurate execution of calculations, and in the event of any errors, it captures and displays the exceptions for user awareness. The values of macd and signal lines are stored as additional columns to the stock_data dataframe, with column names as 'MACD' and 'Signal_Line'.

visualize_macd

```
def visualize_macd(self, plot_start = None, plot_end = None):
        plot_start = plot_start or self.start_date
        plot_end = plot_end or self.end_date
        try:
          if plot_start < self.start_date:
            raise Exception("Date Error: Plot Start date provided is before
the Data Fetch start date")
          elif plot_end > self.end_date:
            raise Exception("Date Error: Plot End date provided is after the
Data Fetch end date")
          else:
```

```
            plot_data = self.stock_data[(self.stock_data['Date'] >=
plot_start) & (self.stock_data['Date'] <= plot_end)]

            macd_trace = go.Scatter(x=plot_data['Date'],
y=plot_data['MACD'], mode='lines', name='MACD')
            signal_trace = go.Scatter(x=plot_data['Date'],
y=plot_data['Signal_Line'], mode='lines', name='Signal Line')

            layout = go.Layout(title='MACD Indicator',
                               xaxis=dict(title='Date'),
                               yaxis=dict(title='MACD Value'))
            fig = go.Figure(data=[macd_trace, signal_trace], layout=layout)
            fig.show()
        except Exception as e:
            print(e)
```

Parameter: plot_start (Optional), plot_end (Optional)
Output: Graph

This visualization function is dedicated to presenting MACD values alongside the signal line. As observed in our preceding visualization functions, this one also grants users the flexibility to specify the 'plot_start' and 'plot_end' parameters, offering a graph within the 'start_date' and 'end_date' range if provided. In the absence of these parameters, the function visualizes the graph for the entire dataset. The date validation process aligns with the methodology applied in prior functions. The resulting plot displays MACD values as a blue line and signal line values as a red line.

visualize_macd_histogram

```
def visualize_macd_histogram(self, plot_start=None, plot_end=None):
        try:
            # Calculate MACD Histogram
            self.stock_data['MACD_Histogram'] = self.stock_data['MACD'] -
self.stock_data['Signal_Line']

            # Visualize MACD Histogram
            plot_data = self.stock_data[(self.stock_data['Date'] >=
plot_start) & (self.stock_data['Date'] <= plot_end)]
            histogram_trace = go.Bar(x=plot_data['Date'],
y=plot_data['MACD_Histogram'], name='MACD Histogram')

            layout = go.Layout(title='MACD Histogram',
                               xaxis=dict(title='Date'),
                               yaxis=dict(title='MACD Histogram'))
```

```
            fig = go.Figure(data=[histogram_trace], layout=layout)
            fig.show()

            print("MACD Histogram visualization performed successfully.")
        except Exception as e:
            print(f"Error visualizing MACD Histogram: {e}")
```

Parameter: plot_start (Optional), plot_end (Optional)
Output: Graph

This function provides an alternative visualization of MACD values through histograms. It adheres to the same optional parameters and validation methods observed in all preceding visualization functions. Similar to those functions, users can specify 'plot_start' and 'plot_end' parameters to view a graph within the 'start_date' and 'end_date' range. In the absence of these parameters, the function generates histograms of MACD values for the complete dataset.

calculate_bollinger_bands

```
def calculate_bollinger_bands(self, window=20, num_std=2):
        try:
            # Calculate moving average
            self.stock_data['MA'] =
self.stock_data['Close'].rolling(window=window).mean()

            # Calculate standard deviation
            self.stock_data['STD'] =
self.stock_data['Close'].rolling(window=window).std()

            # Calculate upper and lower Bollinger Bands
            self.stock_data['Upper_Band'] = self.stock_data['MA'] + (num_std
* self.stock_data['STD'])
            self.stock_data['Lower_Band'] = self.stock_data['MA'] - (num_std
* self.stock_data['STD'])

            print(f"Bollinger Bands (window={window}, num_std={num_std})
calculated successfully.")
        except Exception as e:
            print(f"Error calculating Bollinger Bands: {e}")
```

Parameter: window (Default = 20, Optional), num_std (Default = 2, Optional)
Output: Object

The 'calculate_bollinger_bands' function is employed to compute the upper and lower bands for Bollinger Bands. This function accommodates two optional parameters: 'window' with a default

value of 20 and 'num_std' with a default value of 2. The calculated values are stored in the 'stock_data' dataframe under the columns 'Upper_Band' and 'Lower_Band' respectively. The function includes checks to handle calculation errors, and if any such errors arise, they are captured and displayed through exceptions.

visualize_bollinger_bands

```
def visualize_bollinger_bands(self, plot_start=None, plot_end=None):
      plot_start = plot_start or self.start_date
      plot_end = plot_end or self.end_date
      try:
          if plot_start < self.start_date:
              raise Exception("Date Error: Plot Start date provided is
before the Data Fetch start date")
          elif plot_end > self.end_date:
              raise Exception("Date Error: Plot End date provided is after
the Data Fetch end date")
          else:
              plot_data = self.stock_data[(self.stock_data['Date'] >=
plot_start) & (self.stock_data['Date'] <= plot_end)]
              candlestick_trace = go.Candlestick(x=plot_data['Date'],
                                                 open=plot_data['Open'],
                                                 high=plot_data['High'],
                                                 low=plot_data['Low'],
                                                 close=plot_data['Close'],
                                                 name='Candlestick')
              upper_band_trace = go.Scatter(x=plot_data['Date'],
                                            y=plot_data['Upper_Band'],
                                            mode='lines',
                                            name='Upper Bollinger Band',
                                            line=dict(color="#FF5733"))
              lower_band_trace = go.Scatter(x=plot_data['Date'],
                                            y=plot_data['Lower_Band'],
                                            mode='lines',
                                            name='Lower Bollinger Band',
                                            line=dict(color="#33FF57"))
              layout = go.Layout(title='Bollinger Bands',
                                 xaxis=dict(title='Date'),
                                 yaxis=dict(title='Price'))

              fig = go.Figure(data=[candlestick_trace, upper_band_trace,
lower_band_trace], layout=layout)
              fig.update_layout(xaxis_rangeslider_visible=False)
              fig.show()
      except Exception as e:
```

```
            print(e)
```

Parameter: plot_start (Optional), plot_end (Optional)
Output: Graph

This visualization function follows the established parameters and data checks, similar to other visualization functions. Its primary purpose is to generate a plot featuring price data alongside Bollinger Bands. The price data is visualized in the form of candlesticks, while the upper Bollinger Band is presented in orange and the lower Bollinger Band in green. In the event of any errors occurring during the graph plotting process, the function captures and displays the specific error.

calculate_rsi

```
def calculate_rsi(self, window=14):
      try:
          # Calculate daily price changes
          daily_changes = self.stock_data['Close'].diff()
          # Calculate average gains and losses over the specified window
          avg_gain = daily_changes.where(daily_changes > 0,
0).rolling(window=window).mean()
          avg_loss = -daily_changes.where(daily_changes < 0,
0).rolling(window=window).mean()
          # Calculate Relative Strength (RS)
          relative_strength = avg_gain / avg_loss
          # Calculate Relative Strength Index (RSI)
          self.stock_data['RSI'] = 100 - (100 / (1 + relative_strength))
          print(f"RSI (window={window}) calculated successfully.")
      except Exception as e:
          print(f"Error calculating RSI: {e}")
```

Parameter: window (Default = 14, Optional)
Output: Object

The purpose of this function is to calculate the relative strength index (RSI) of the stock, featuring one optional parameter, 'window,' with a default value of 14. Users have the flexibility to specify an alternative value for 'window' as per their preference to calculate RSI accordingly. The calculated RSI values are stored in the 'stock_data' dataframe under the column 'RSI.' The function incorporates checks to capture and display any errors that may arise during the calculation process.

visualize_rsi

```
def visualize_rsi(self, plot_start=None, plot_end=None):
      plot_start = plot_start or self.start_date
```

```
      plot_end = plot_end or self.end_date
      try:
          if plot_start < self.start_date:
              raise Exception("Date Error: Plot Start date provided is
before the Data Fetch start date")
          elif plot_end > self.end_date:
              raise Exception("Date Error: Plot End date provided is after
the Data Fetch end date")
          else:
              plot_data = self.stock_data[(self.stock_data['Date'] >=
plot_start) & (self.stock_data['Date'] <= plot_end)]

              rsi_trace = go.Scatter(x=plot_data['Date'],
y=plot_data['RSI'], mode='lines', name='RSI', line=dict(color="#FF5733"))

              # Add horizontal lines for overbought and oversold levels
(e.g., 70 and 30)
              overbought_trace = go.Scatter(x=plot_data['Date'], y=[70] *
len(plot_data), mode='lines', name='Overbought',
line=dict(color="#FF0000"))
              oversold_trace = go.Scatter(x=plot_data['Date'], y=[30] *
len(plot_data), mode='lines', name='Oversold', line=dict(color="#00FF00"))
              layout = go.Layout(title='Relative Strength Index (RSI)',
                                 xaxis=dict(title='Date'),
                                 yaxis=dict(title='RSI'))
              fig = go.Figure(data=[rsi_trace, overbought_trace,
oversold_trace], layout=layout)
              fig.update_layout(xaxis_rangeslider_visible=False)
              fig.show()
      except Exception as e:
          print(e)
```

Parameter: plot_start (Optional), plot_end (Optional)
Output: Graph

This function serves as the final visualization function within our package, maintaining alignment with the parameters and checks applied in all preceding visualization functions. Its purpose is to generate a plot illustrating RSI values alongside overbought and oversold traces. The graph displays three lines, designating overbought at 70%, oversold at 30%, and the RSI value line for the stock. Exception handling is incorporated to manage any errors that may arise during the graph plotting process.

get_latest_news

```
def get_latest_news(self):
```

```
    try:
      if not self.ticker.news:
        raise Exception("No News: Cannot fetch latest news.")
      else:
        for news in self.ticker.news:
          print("Title: ", news['title'])
          print("Link: ", news['link'])
          print("Publisher: ", news['publisher'],"\n\n")
    except Exception as e:
      print(e)
```

Parameter: None
Output: Title, Link, Publisher

In addition to the technical aspects, market news significantly influences stocks. This concluding function in our package offers users the option to retrieve and read the latest news relevant to the stock they are analyzing. The function initially checks for the presence of any recent news; if none is found, it raises an exception for no news. In the event of available news, the function provides three key details: the news title, a link to the website for accessing the complete article, and the name of the publisher.

Functioning

```
stock_package = StockAnalysisPackage(symbol='AMXN', start_date='2022-01-01', end_date='2023-01-01')
stock_package.fetch_stock_data()
```

```
Symbol Error: Ticker does not belong to stock/equity
```

```
stock_package = StockAnalysisPackage(symbol='FAKE', start_date='2022-01-01', end_date='2023-01-01')
stock_package.fetch_stock_data()
```

```
Ticker FAKE does not exist.
```

```
stock_package = StockAnalysisPackage(symbol='LXEO', start_date='2022-01-01', end_date='2023-01-01')
stock_package.fetch_stock_data()
```

```
Date Error: Start Date before stock List Date
```

```
stock_package = StockAnalysisPackage(symbol='AAPL', start_date='2022-01-01', end_date='2024-01-01')
stock_package.fetch_stock_data()
```

```
Date Error: End Date after today's date
```

```
stock_package = StockAnalysisPackage(symbol='AAPL', start_date='2022-01-01', end_date='2023-01-01')
stock_package.fetch_stock_data()
```

```
[**********100%%**********]  1 of 1 completed
Stock data fetched successfully.
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 251 entries, 0 to 250
Data columns (total 7 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       251 non-null    datetime64[ns]
 1   Open       251 non-null    float64
 2   High       251 non-null    float64
 3   Low        251 non-null    float64
 4   Close      251 non-null    float64
 5   Adj Close  251 non-null    float64
 6   Volume     251 non-null    int64
dtypes: datetime64[ns](1), float64(5), int64(1)
memory usage: 13.9 KB
```

```
stock_package.visualize_candlestick()
```


```
stock_package.visualize_candlestick('2022-06-01','2023-01-01')
```

```
stock_package.visualize_candlestick('2021-06-01','2023-01-01')
```

```
Date Error: Plot Start date provided is before the Data Fetch start date
```

```
stock_package.visualize_candlestick('2022-06-01','2024-01-01')
```

```
Date Error: Plot End date provided is after the Data Fetch end date
```

```
stock_package.calculate_moving_average(20)
```

```
Moving average (window=20) calculated successfully.
```

```
stock_package.visualize_moving_average()
```


```
stock_package.visualize_moving_average('2022-06-01','2023-01-01')
```

```
stock_package.visualize_moving_average('2021-06-01','2023-01-01')
```

```
Date Error: Plot Start date provided is before the Data Fetch start date
```

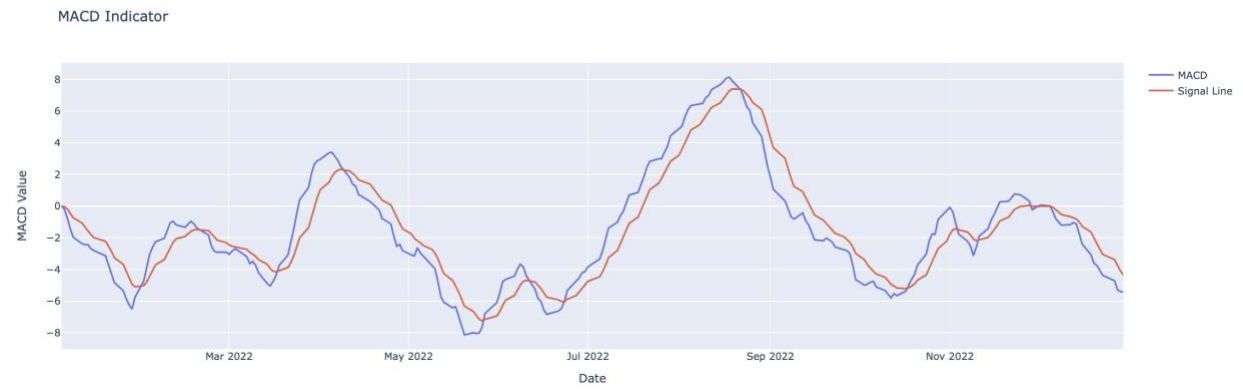
```
stock_package.visualize_moving_average('2022-06-01','2024-01-01')
```

```
Date Error: Plot End date provided is after the Data Fetch end date
```

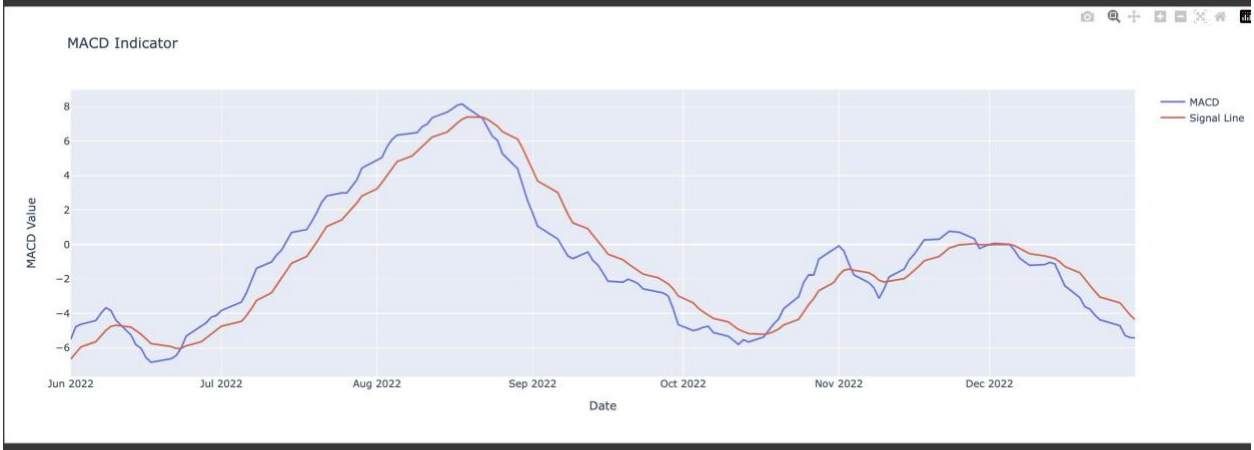
```
stock_package.perform_macd()
```

```
MACD analysis performed successfully.
```

```
stock_package.visualize_macd()
```




```
stock_package.visualize_macd('2022-06-01','2023-01-01')
```




```
stock_package.visualize_macd('2021-06-01','2023-01-01')
```

```
Date Error: Plot Start date provided is before the Data Fetch start date
```

```
stock_package.visualize_macd('2022-06-01','2024-01-01')
```

```
Date Error: Plot End date provided is after the Data Fetch end date
```

```
stock_package.get_latest_news()
```

```
Title:  Apple's iPhone 15 Pro unleashes 3D memories on Vision Pro
Link:  https://finance.yahoo.com/m/fd04bb74-39bb-3f14-8f39-335df1e6f2e2/apple%27s-iphone-15-pro.html
Publisher:  AppleInsider


Title:  Best Dow Jones Stocks To Buy And Watch In December 2023: Microsoft Sells Off
Link:  https://finance.yahoo.com/m/65b53896-faf4-3a06-9d0d-a63cf3c83192/best-dow-jones-stocks-to-buy.html
Publisher:  Investor's Business Daily


Title:  Best Monitor for MacBook Pro
Link:  https://finance.yahoo.com/m/855780c3-1e16-354d-9c9c-17bd8acba556/best-monitor-for-macbook-pro.html
Publisher:  AppleInsider


Title:  12 Best Growth Stocks to Buy Today According to Billionaire Ken Fisher
Link:  https://finance.yahoo.com/news/12-best-growth-stocks-buy-182331189.html
Publisher:  Insider Monkey


Title:  11 Best Robinhood Stocks According to Billionaires
Link:  https://finance.yahoo.com/news/11-best-robinhood-stocks-according-175515660.html
Publisher:  Insider Monkey


Title:  PayPal Is a Huge Value Opportunity With Growth Risks
Link:  https://finance.yahoo.com/news/paypal-huge-value-opportunity-growth-171850392.html
Publisher:  GuruFocus.com


Title:  Tight-Lipped Magnificent Seven Stock Breaks Out Amid Rumors
Link:  https://finance.yahoo.com/m/ebce070f-6fdf-3286-909c-2da9de763841/tight-lipped-magnificent.html
Publisher:  Investor's Business Daily


Title:  Amazon Wins Top EU Court Clash Over €250 Million Tax Bill
Link:  https://finance.yahoo.com/news/amazon-wins-top-eu-court-084914109.html
Publisher:  Bloomberg
```

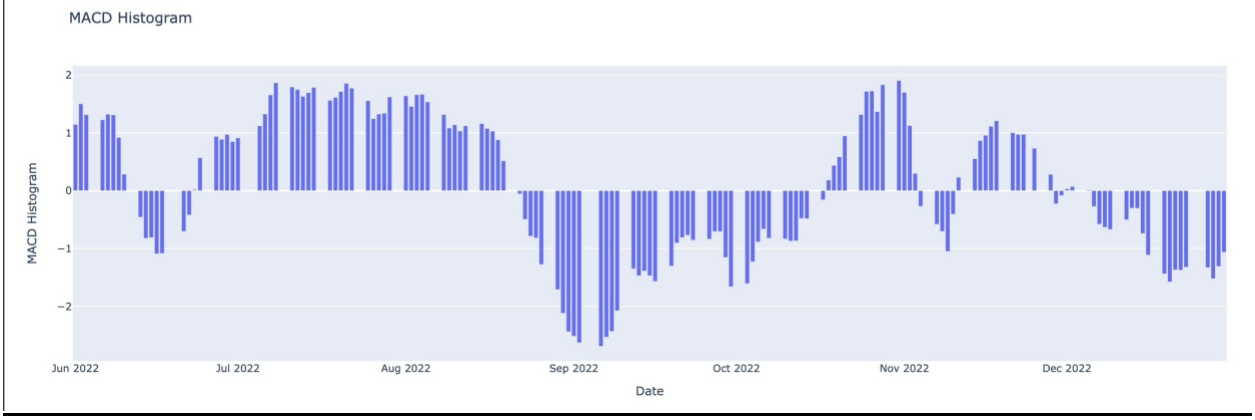
```
stock_package.calculate_bollinger_bands(window=20, num_std=2)
stock_package.visualize_bollinger_bands('2022-06-01', '2023-01-01')
```

```
# Call the calculate_rsi method
stock_package.calculate_rsi(window=14)

# Call the visualize_rsi method
stock_package.visualize_rsi('2022-06-01', '2023-01-01')
```

```
RSI (window=14) calculated successfully.
```


```
# Perform MACD analysis
stock_package.perform_macd()

# Call the visualize_macd_histogram method
stock_package.visualize_macd_histogram('2022-06-01', '2023-01-01')
```

```
MACD analysis performed successfully.
```



Contribution

Shubham Narkhede -

- Package setup, adding required packages.
- Main class initialization function and calls of class functions.
- Data fetching function.
- Implementation of exception and error handling in all sections of the package.
- MACD Histogram visualization function (`visualize_macd_histogram`).

Subham Moda –

- Candlestick visualization function (`visualize_candlestick`).
- Moving average calculation function (`calculate_moving_average`).

- Moving average visualization function (`visualize_moving_average`).
- Error handling as required.
- Project Report

Xinran Liu -

- MACD calculation function (`perform_macd`).
- MACD visualization function (`visualize_macd`).

Zixin Zhou -

- Bollinger Bands calculation function (`calculate_bollinger_bands`).
- Bollinger Bands visualization function (`visualize_bollinger_bands`).

Yanyun Wang-

- Relative Strength Index (RSI) calculation function (`calculate_rsi`).
- RSI visualization function (`visualize_rsi`).
- Latest news fetching function (`get_latest_news`).
- Function calls of all the feature of the Package