Task Report Cos30018 Option B

B.3: Data Processing 2

Name: Le Bao Nguyen

Student Id: 104169837

I. Importing libraries (From the task B.2):

```
import numpy as np
import pandas as pd
import yfinance as yf
from sklearn.preprocessing import MinMaxScaler
import plotly.graph_objects as go
import plotly.express as px
from IPython.display import display, HTML
from plotly.subplots import make_subplots
from google.colab import drive

# Mount Google Drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Figure 1: Importing libraries to run the code.

- The script import several libraries:

- "numpy": Supports numerical operations as well as arrays.
- "pandas": Utilized for analysis and data modification.
- "yfinance": Allows downloading historical market data from Yahoo Finance.
- "MinMaxScaler" from "sklearn.preprocessing": Applying a part of scikit-learn, data is scaled to a range (0, 1).
- "plotly.graph_objects" and "plotly.express": Creating interactive plots
- "IPython.display": Displaying data frames as HTML in Jupyter notebooks.
- "make_subplots" from "plotly.subplots": This function allows the creation of multi-panel figures with multiple subplots. It is

particularly useful for arranging different types of plots, such as boxplots and volume charts, in a cohesive layout.

- "Drive" from "google.collab": Import drive from google collab to connect and save file to google drive.

II. Data loading and processing (From the task B.2):

```
[ ]  # Download data from Yahoo Finance
     data = yf.download(ticker, start=start_date, end=end_date)

     # Ensure the index is a DateTimeIndex
     data.index = pd.to_datetime(data.index)

     # Fill NaN values with previous values
     data.fillna(method='ffill', inplace=True)

     # Sanity check: Ensure high is not less than low
     if (data['High'] < data['Low']).any():
         raise ValueError("Inconsistent data: High value is less than Low value for some periods.")

     # Default to scaling the 'Close' column if no columns are specified
     if columns_to_scale is None or not columns_to_scale:
         columns_to_scale = ['Close']

     # Create a DataFrame for scaled data
     scaled_data = data.copy()
     scalers = {}

     # Scale specified columns
     for column in columns_to_scale:
         scaler = MinMaxScaler(feature_range=(0, 1))
         scaled_column = scaler.fit_transform(data[column].values.reshape(-1, 1))
         scaled_data[f'Scaled_{column}'] = scaled_column
         scalers[column] = scaler

     # Extract close prices and scale them
     close_prices = data['Close'].values.reshape(-1, 1)
     scaler = MinMaxScaler(feature_range=(0, 1))
     scaled_close_prices = scaler.fit_transform(close_prices)

     # Determine split date based on split_ratio or split_by_date
     if split_by_date:
         split_date = pd.Timestamp(split_ratio)
     else:
         split_date = pd.to_datetime(start_date) + (pd.to_datetime(end_date) - pd.to_datetime(start_date)) * split_ratio
```

Figure 2: Loading and processing data (1).

```
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_close_prices = scaler.fit_transform(close_prices)

    # Determine split date based on split_ratio or split_by_date
    if split_by_date:
        split_date = pd.Timestamp(split_ratio)
    else:
        split_date = pd.to_datetime(start_date) + (pd.to_datetime(end_date) - pd.to_datetime(start_date)) * split_ratio

    # Split data into train and test sets
    if split_by_date:
        train_data = scaled_close_prices[data.index < split_date]
        test_data = scaled_close_prices[data.index >= split_date]
    else:
        train_data = scaled_close_prices[:int(len(scaled_close_prices) * split_ratio)]
        test_data = scaled_close_prices[int(len(scaled_close_prices) * split_ratio):]

    # Save data to a local file, replacing any existing file
    if local_file:
        file_path = f"/content/drive/My Drive/Cos30018/{local_file}"  # Change to your desired path in Google Drive
        data.to_csv(file_path)

    return train_data, test_data, scalers, data, scaled_data
```

Figure 3: Loading and processing data (2).

- Using yfinance, the "load_and_processdata" function retrieves stock data for a specified ticker symbol and date range. The 'Close' prices are extracted and scaled using MinMaxScaler to a range between 0 and 1, after any missing values in the data are filled in using forward fill.
- Additionally, the function divides the data by a given ratio or date to create training and testing sets. The function saves the downloaded data as a CSV file if a local file name is given. The location that the file saved is in the gg drive.

III. Displaying data in a custom table (From the task B.2):

```
[ ]  def display_custom_table(df, num_rows=5):
         """
         Display the first few and last few rows of the DataFrame with ellipses in between.

         Parameters:
         - df: DataFrame to display.
         - num_rows: Number of rows to display from the start and end of the DataFrame.
         """
         if len(df) <= 2 * num_rows:
             # Display the entire DataFrame if it's small enough
             display(df)
         else:
             # Display the first few and last few rows with ellipses in between
             head = df.head(num_rows)
             tail = df.tail(num_rows)
             ellipsis_row = pd.DataFrame([['...'] * len(df.columns)], columns=df.columns, index=['...'])
             df_display = pd.concat([head, ellipsis_row, tail])
             display(HTML(df_display.to_html(index=True)))
```

Figure 4: Displaying the data from csv file.

- If a data frame is too big to display completely, the "display_custom_table" method shows it in a truncated format. The data frame's beginning and last few rows are displayed, with an ellipsis row between them to represent the middle rows that were left out.
- This method guarantees that users won't be able to overflow the display when viewing a representative sample of huge data frame. The number of rows to display at the top and bottom of the data frame, as well as the data frame to be displayed, are the two arguments passed to the function.

IV. Displaying the candlestick chart:

```python
def candlestick_chart(data, title='Candlestick chart', n_days=1, candle_width=0.8):
    # Resample data if n_days > 1, aggregating over n trading days
    if n_days > 1:
        data_resampled = data.resample(f'{n_days}D').agg({
            'Open': 'first',
            'High': 'max',
            'Low': 'min',
            'Close': 'last',
            'Volume': 'sum'
        }).dropna()
    else:
        data_resampled = data

    # Create candlestick chart using Plotly
    fig = go.Figure(data=[
        go.Candlestick(
            x=data_resampled.index,
            open=data_resampled['Open'],
            high=data_resampled['High'],
            low=data_resampled['Low'],
            close=data_resampled['Close'],
            name=title,
            increasing_line_width=2,
            decreasing_line_width=2,
            increasing=dict(line=dict(width=candle_width)),
            decreasing=dict(line=dict(width=candle_width))
        ),
        go.Bar(
            x=data_resampled.index,
            y=data_resampled['Volume'],
            name='Volume',
            marker_color='blue',
            yaxis='y2',
            opacity=0.3
        )
    ])

    # Set y-axis range with some padding
    price_range = data_resampled['High'].max() - data_resampled['Low'].min()
    fig.update_layout(
        title=title,
```

Figure 5: Code to print the candle stick chart (1).

```
])

# Set y-axis range with some padding
price_range = data_resampled['High'].max() - data_resampled['Low'].min()
fig.update_layout(
    title=title,
    xaxis_title='Date',
    yaxis_title='Price',
    yaxis=dict(
        range=[
            data_resampled['Low'].min() - price_range * 0.05,
            data_resampled['High'].max() + price_range * 0.05
        ]
    ),
    yaxis2=dict(
        title='Volume',
        overlaying='y',
        side='right',
        showgrid=False
    ),
    xaxis_rangeslider_visible=True
)

# Add range selector buttons
fig.update_xaxes(
    rangeslider_visible=True,
    rangeselector=dict(
        buttons=list([
            dict(count=1, label="1m", step="month", stepmode="backward"),
            dict(count=6, label="6m", step="month", stepmode="backward"),
            dict(count=1, label="YTD", step="year", stepmode="todate"),
            dict(count=1, label="1y", step="year", stepmode="backward"),
            dict(step="all")
        ])
    )
)

fig.show()
```

Figure 6: Code to print the candle stick chart (2).

- Using Plotly, the "candlestick_chart" function generates a candlestick chart using stock data. Resampling the data to aggregate over several trading days is an optional feature. The 'Open', 'High', 'Low', and 'Close' prices as well as the 'Volume' for the designated time are calculated.
- After that, the function creates the candlestick chart, which has selector buttons for various time periods and a range slider that allows you to choose the candle width. The function arguments consist of the width of the candlestick lines, the number of trading

days each candlestick represents, the chart title, and the stock data DataFrame.

- "Data": Data frame containing stock data.
- "title": Title of the candlestick chart.
- "n_days": Number of trading days each candlestick represents.
- "candle_width": Width of the candlestick lines.

V. Displaying the box plot chart:

```python
def boxplot_chart(data, title='Boxplot chart', window_size=20):
    """
    Function to display stock market financial data using a boxplot chart for a moving window.

    Parameters:
    - data: DataFrame containing stock market data with columns 'Open', 'High', 'Low', 'Close', 'Adj Close', and 'Volume'.
    - title: Title of the boxplot chart.
    - window_size: Size of the moving window for calculating rolling statistics.

    This function displays a boxplot for the specified columns over a moving window and a volume chart below it.
    """
    # Ensure data index is a datetime index for proper resampling
    data.index = pd.to_datetime(data.index)

    # Create subplots with 2 rows: one for box plots, one for volume
    fig = make_subplots(
        rows=2, cols=1,
        shared_xaxes=True,
        vertical_spacing=0.1,
        row_heights=[0.7, 0.3],
        subplot_titles=(title, "Volume")
    )

    # Define the columns to be plotted in the boxplot
    columns = ['Open', 'High', 'Low', 'Close', 'Adj Close']

    # Calculate rolling mean for the specified window size
    rolling_data = data[columns].rolling(window=window_size).mean().dropna()

    # Create boxplots for the specified columns in the rolling window data
    for col in columns:
        fig.add_trace(go.Box(
            y=rolling_data[col],
            name=f"{col} ({window_size} Days Moving Average)",
            boxmean=True,  # Include the mean in the boxplot
            hovertemplate=(
                f"<b>{col}</b><br>"
                "Max: %{y}<br>"
                "Q3: %{upperfence}<br>"
                "Median: %{med}<br>"
                "Q1: %{lowerfence}<br>"
                "Min: %{min}<extra></extra>"
            )
        ), row=1, col=1)
```

Figure 7: Code to print the box plot chart (1).

```
# Create boxplots for the specified columns in the rolling window data
for col in columns:
    fig.add_trace(go.Box(
        y=rolling_data[col],
        name=f"{col} ({window_size} Days Moving Average)",
        boxmean=True,  # Include the mean in the boxplot
        hovertemplate=(
            f"<b>{col}</b><br>"
            "Max: %{y}<br>"
            "Q3: %{upperfence}<br>"
            "Median: %{med}<br>"
            "Q1: %{lowerfence}<br>"
            "Min: %{min}<extra></extra>"
        )
    ), row=1, col=1)

# Add volume as a bar chart below the box plots
fig.add_trace(go.Bar(
    x=data.index,
    y=data['Volume'],
    name='Volume',
    marker_color='blue',
    opacity=0.3
), row=2, col=1)

# Update layout for the boxplot and volume chart
fig.update_layout(
    title=title,
    yaxis_title='Price',
    yaxis2_title='Volume',
    xaxis_title='',
    showlegend=False
)

# Show the figure
fig.show()
```

Figure 8: Code to print the box plot chart (2).

- In order to visualize the distribution of stock prices throughout the rolling periods, the "boxplot_chart" function creates boxplot charts for specified rolling windows of stock data.
- Plotting these values as boxplots, it calculates the rolling mean for each price column ('Open,' High,' Low,' Close,' Adj Close') over the specified window widths.
- This function uses Plotly to create interactive visualizations and accepts the stock data DataFrame, the chart title, and a rolling window size as arguments.

- "data": DataFrame containing stock data.
- "title": Title of the boxplot chart.

- "window_size": Sizes of the rolling windows for the boxplots.
- A bar chart is added to show the trade volume for each day below the boxplots. Given that changes in volume might reveal changes in investor views and possible price volatility, the volume data gives the price movements context.

VI. Main script run:

```
[35] # Example usage
if __name__ == "__main__":
    ticker = 'AMZN'
    start_date = '2016-01-01'
    end_date = '2024-03-20'
    local_file = 'amzn_data.csv'

    # Specify columns to scale
    columns_to_scale = ["Volume"]

    # Load and process data
    train_data, test_data, scalers, original_data, scaled_data = load_and_process_data(
        ticker=ticker,
        start_date=start_date,
        end_date=end_date,
        local_file=local_file,
        split_ratio=0.7,
        columns_to_scale=columns_to_scale
    )

    # Display the original and scaled data tables
    display_custom_table(scaled_data, num_rows=5)

    total_data = np.concatenate((train_data, test_data), axis=0)

    # Plot candlestick chart
    candlestick_chart(original_data[(original_data.index >= start_date) & (original_data.index <= end_date)], title=f'{ticker} Candlestick Chart', n_days=7, candle_width=5)  # Use larger n_days to make each candlestick larger

    # Display the boxplot chart with volume
    boxplot_chart(original_data, title=f'{ticker} Boxplot and Volume Chart', window_size=20)
```

Figure 9: The script to run the code and change the date.

- The defined functions are used as an example in the main block. It loads and analyzes Amazon.inc (AMZN) stock data, divides it based on dates and ratios, creates a personalized table, and generates boxplot and candlestick charts.

Figure 10: The AMZN candle stick chart.

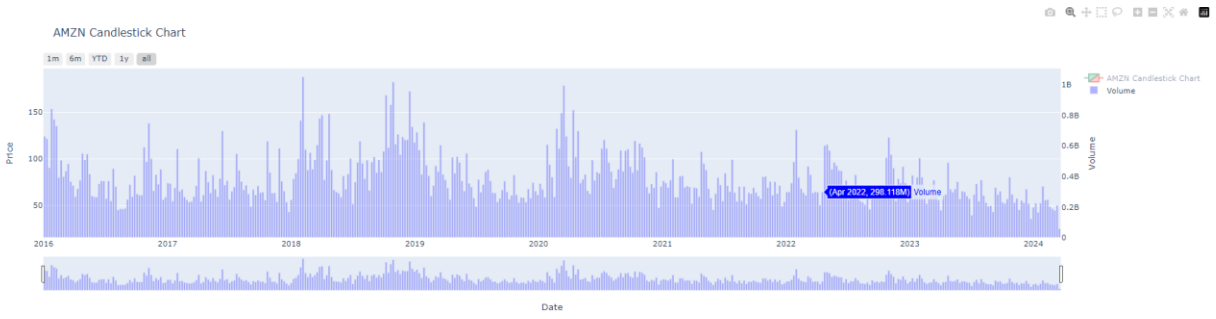

Figure 11: The AMZN volume chart.

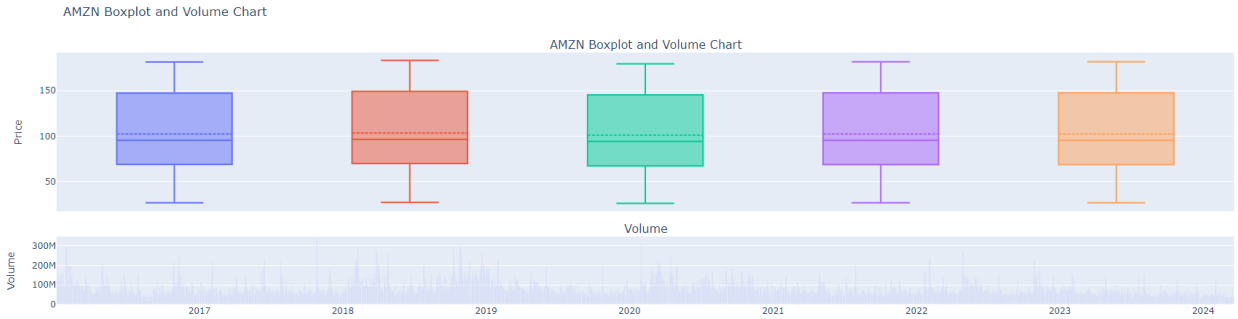

Figure 12: The AMZN box plot chart.

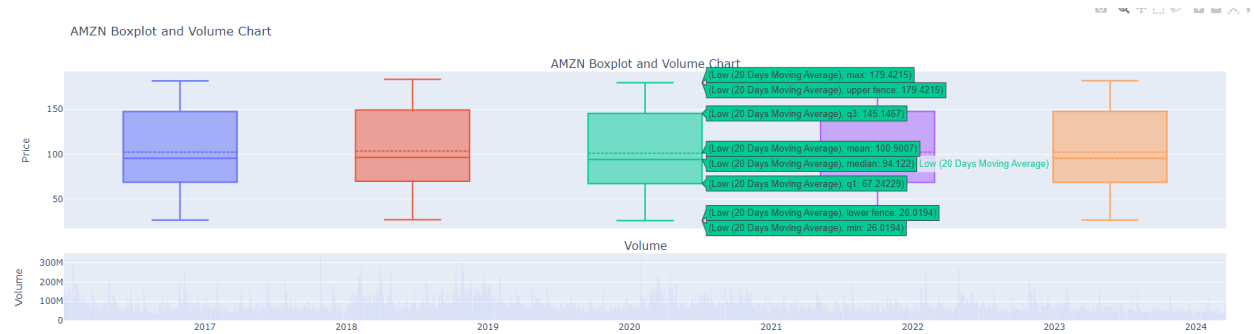

Figure 13: The AMZN box plot chart when hovering.

Figure 14: The AMZN volume bar of box plot chart when hovering.

VII. Challenge faced:

- One of the main difficulties was making sure that the data was resampled accurately for the candlestick chart, which included knowing how to use the pandas aggregation algorithms appropriately. Additionally, a thorough understanding of pandas' rolling operations was required in order to set up the rolling windows for the boxplot display. Thorough investigation and testing were also necessary to properly configure Plotly's display parameters, such as range sliders and selector buttons. Also, the hover at the box plot still being duplicated at the end of median information.