Task 3 Report: Stock Market Visualization Functions

Student Name: Tommy Tran
Student ID: 104939946

Task 3 Implementation Overview

Task 3 required developing two specialized visualization functions:

1. **Candlestick Chart Function** - Display OHLC data with configurable n-day aggregation
2. **Boxplot Chart Function** - Display rolling window statistical analysis of price movements

I tested both functions extensively using Commonwealth Bank of Australia (CBA.AX) data from Yahoo Finance, covering the period 2021-2023, which provided a good mix of trending and volatile market conditions.

1. Candlestick Chart Implementation

1.1 My Approach and Initial Mistakes

When I first started implementing the candlestick chart, I made several assumptions that turned out to be wrong. Initially, I thought I should use a rolling window approach for n-day aggregation - taking overlapping periods like a moving average. After researching financial charting conventions and testing with real data, I realized this wasn't standard practice. Financial analysts use non-overlapping periods, so a 5-day chart shows Monday-Friday as one candle, Tuesday-Saturday as the next, and so on.

1.2 Critical Code Analysis and Learning Process

1.2.1 Understanding OHLC Aggregation Rules

The most challenging part was figuring out how to properly aggregate multiple days into a single candle:

```
aggregated_row = {
    'Open': chunk['Open'].iloc[0],        # First day's open
    'High': chunk['High'].max(),          # Highest high
    'Low': chunk['Low'].min(),            # Lowest low
    'Close': chunk['Close'].iloc[-1],     # Last day's close
}
```

At first, I misunderstood how this should work. I initially thought about averaging the opens and closes, but after checking Investopedia and several finance forums, I confirmed that the standard convention is: open = first day's opening price, close = last day's closing price, high = maximum across all days, low = minimum across all days. This makes sense because it represents what actually happened during that period.

1.2.2 Volume Aggregation Confusion

Volume aggregation was another area where I initially got confused:

```
if 'Volume' in chunk.columns:
    aggregated_row['Volume'] = chunk['Volume'].sum()  # Sum of volumes
```

My first instinct was to average the volume, but after testing different methods with CBA data, I confirmed that summing is the correct approach. This represents the total trading activity during the aggregated period, which is what traders actually care about.

1.2.3 A Critical Bug I Fixed

I encountered a frustrating bug where my candlestick charts were displaying incorrectly - the dates seemed misaligned and some candles appeared in the wrong order. After hours of debugging, I discovered the issue was with pandas indexing:

```
# This was causing the bug:
data = pd.DataFrame(grouped_data)
# Missing: data.set_index('Date', inplace=True)

# Fixed version:
data = pd.DataFrame(grouped_data)
data.set_index('Date', inplace=True)
```

The problem was that after creating the aggregated DataFrame, I wasn't properly setting the Date column as the index. This caused mplfinance to interpret the data incorrectly. Fixing this taught me how crucial proper pandas indexing is, especially when working with time series data.

1.3 Testing Results and Observations

When testing with one year of Apple (AAPL) data from Yahoo Finance, I noticed that weekly aggregation (n_days=5) showed much clearer trend patterns compared to daily data. The daily charts were quite noisy, but the weekly view revealed underlying trends that were harder to spot in the daily noise. This gave me a better understanding of why financial analysts use different timeframes for different types of analysis.

2. Boxplot Rolling Window Implementation

2.1 My Implementation Strategy

For the boxplot function, I took a different approach based on what I learned from the candlestick implementation:

```
def plot_boxplots_moving_window(df, price_column="Close", window=20, stride=1, ...):
```

This time, I researched the statistical concepts first before coding. I wanted to understand what information boxplots could provide to stock analysts.

2.2 Understanding the Rolling Window Logic

The core sliding window algorithm was more straightforward once I understood the mathematical concept:

```
for end in range(window, len(prices) + 1, stride):
    start = end - window
    w = prices.iloc[start:end].values
    data_windows.append(w)
```

I initially struggled with the indexing here. My first version had off-by-one errors that took me a while to debug. The key insight was understanding that end represents the position *after* the last element we want to include, which is standard Python slicing behavior.

2.3 Practical Testing Insights

When testing the boxplot function with CBA data using 20-day windows, I discovered something interesting about market volatility. During stable periods, the boxes were relatively small and consistent. But during volatile periods (like market corrections), some windows showed much larger boxes with more outliers. This helped me understand how boxplots can reveal changing market conditions over time.

4. Main Challenges I Faced

4.1 Technical Learning Curve

The biggest challenge was using the mplfinance library. Unlike matplotlib, which I was familiar with, mplfinance has its own conventions and styling system. Understanding the make_marketcolors() and make_mpf_style() functions required reading through multiple examples and experimenting with different parameter combinations.

4.2 Domain Knowledge Gap

Initially, I underestimated how much financial domain knowledge I needed. Simple questions like "How do you aggregate volume?" or "What time period should each candle represent?" required research into trading conventions and market practices.

4.3 Data Quality Issues

Working with real market data introduced challenges I hadn't anticipated. Missing data points, market holidays, and irregular trading schedules all affected my functions. While I implemented basic error handling, I realized there's much more complexity in real-world financial data processing.

5. Limitations and Future Improvements

5.1 Current Limitations

My current implementation has several limitations I'm aware of:

- The boxplot function assumes equal time spacing between data points, but markets are closed on weekends and holidays
- The candlestick aggregation doesn't handle partial periods at the end of datasets elegantly
- Error messages could be more user-friendly for non-technical users

5.2 Future Enhancement Ideas

If I were to extend this project, I'd focus on:

1. **Smart date handling**: Adjusting for market holidays and irregular trading days
2. **Interactive features**: Adding zoom, pan, and hover capabilities using plotly
3. **Technical indicators**: Overlaying moving averages, RSI, or other common indicators
4. **Performance optimization**: Handling much larger datasets more efficiently

References

1. mplfinance Documentation. "Plotting Functions and Parameters." https://github.com/matplotlib/mplfinance

2. Stack Overflow. "mplfinance volume colors and styling." https://stackoverflow.com/questions/tagged/mplfinance

3. Pandas Documentation. "Time Series / Date functionality." https://pandas.pydata.org/docs/user_guide/timeseries.html

4. Yahoo Finance. Historical stock data used for testing. https://finance.yahoo.com

5. Matplotlib Documentation. "matplotlib.pyplot.boxplot parameters." https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.boxplot.html