COS30018 - Intelligent Systems

TASK C.3 - DATA PROCESSING 2

RAHUL RAJU

Student ID: 105143065

Table of Contents

Introduction

The transition from Task 2 of the stock prediction project represents a significant evolution in analytical capability. While the previous version successfully established a machine learning pipeline for price prediction, its visualisation was limited to a basic line graph. This report details the implementation of advanced financial visualisatioin techniques, specifically candlestick and boxplot charts, to provide a more nuanced and professional understanding of market data. These enhancements transform the project from a simple predicitive model into a more comprehensive tool for financial market analysis, allowing users to glean insights into price volatility, trading ranges, and market sentiment that are not apparent in a simple line plot.

Key Enhancements

Candlestick Chart Functionality

The *plot_candlestick_chart* function was implemented using the *mplfinance* library, a specialist tool for financial visualisation. This function moves beyond plotting a single price point (like closing price) to visualising four critical data points for each period: Open, High, Low, and Close (OHLC). This provides an immediate visual representation of market sentiment and price action within a given timeframe.

A key feature of this implementation is the ability to aggregate data into multi-day periods. While a daily candlestick is standard, viewing weekly (*n_days=5*) or monthly data can reveal longer-term trends. The core logic for this aggregation is handled in the following code segment:

```python
# If n_days > 1, we need to resample the data to create candles for n-day periods
if n_days > 1:
    # Make a copy to avoid modifying original data
    data = data.copy()

    # Create a numeric index for grouping (0, 1, 2, 3, ...)
    numeric_index = np.arange(len(data))

    # Create a new column to group every n_days rows using the numeric index
    data['group'] = numeric_index // n_days  # Integer division to create groups

    # Aggregate data for each group to form new candles
    # Open = first open price in the group
    # High = maximum high price in the group
    # Low = minimum low price in the group
    # Close = last close price in the group
    # Volume = sum of volumes in the group (if volume column exists)

    agg_dict = {
        'Open': 'first',
        'High': 'max',
        'Low': 'min',
        'Close': 'last'
    }

    # Add Volume to aggregation if it exists in the data
    if 'Volume' in data.columns:
        agg_dict['Volume'] = 'sum'

    # Group by the 'group' column and aggregate
    resampled_data = data.groupby('group').agg(agg_dict)

    # Create new datetime index - use the last date in each group as the candle date
    # Fixed: Removed include_groups parameter for compatibility with older pandas versions
    new_index = data.groupby('group').apply(lambda x: x.index[-1])
    resampled_data.index = new_index

    # Use the resampled data for plotting
    plot_data = resampled_data
    period_text = f"({n_days}-Day Periods)"
else:
    # Use original data for daily candles
    plot_data = data
    period_text = "(Daily)"
```

Explanation:

- **N Day Aggregation**: The function creates a sequential numeric index because mathematical operations like integer division cannot be performed directly on a *DatetimeIndex*. This numeric index is divided by *n_days* to create distinct groups of consecutive rows.
- **Financial Aggregation Logic:** A dictionary defines how to aggregate each financial column within a group, adhering to market conventions. The 'Open' is the first value in the group, the 'High' is the maximum, the 'Low' is the minimum, and the 'Close' is the last value. Volume is summed to show total activity over the period.

- **Index Reconstruction:** The aggregated data is assigned a new datetime index using the last date from each group, logically marking the end of the period the candle represents.

The actual plotting is then efficiently handled by *mplfinance*, which abstracts the complexity of drawing individual candles:

```
# Create the candlestick chart using mplfinance
mpf.plot(plot_data,                    # The DataFrame with OHLC data
         type='candle',                # Chart type: 'candle' for candlestick
         style=chart_style,            # Visual style of the chart
         title=f'{company} Stock Price {period_text} {title_suffix}',  # Chart title
         ylabel='Price ($)',           # Y-axis label
         ylabel_lower='Volume',        # Y-axis label for volume subplot (if shown)
         volume='Volume' in plot_data.columns,  # Show volume subplot if Volume column exists
         figsize=figsize,              # Size of the figure
         show_nontrading=False)        # Don't show gaps for non-trading days

# Display the chart
plt.show()
```

This function call is highly configurable, allowing for different visual styles and the optional inclusion of a volume subplot, which is critical for confirming price trends.

Boxplot Chart for Volatility Analysis

The *plot_boxplot_chart* function provides a statistical perspective on price movements, complementing the time series view of the candlestick chart. It displays the distribution of prices over consecutive, non overlapping windows of trading days, making it exceptionally useful for analysing volatility and identifying outlier events.

The function prepares the data by grouping it into windows, similar to the candlestick function, but for the purpose of statistical summary:

```
# Make a copy to avoid modifying original data
data_copy = data.copy()

# Create a numeric index for grouping (0, 1, 2, 3, ...)
numeric_index = np.arange(len(data_copy))

# Create a column to identify which window each row belongs to
# We'll create groups of 'window_size' consecutive days using numeric index
data_copy['window_group'] = numeric_index // window_size

# Create a list to store data for each window
window_data = []
window_labels = []

# Group the data by window_group and create boxplot data for each window
grouped = data_copy.groupby('window_group')

for name, group in grouped:
    # Extract the values for the specified column
    values = group[column].values
    window_data.append(values)

    # Create a label for this window (use the last date in the window)
    last_date = group.index[-1].strftime('%Y-%m-%d')
    window_labels.append(last_date)
```

Explanation:

- **Data Preparation for Boxplot:** The data is grouped into windows of *window_size* days. For each windows, all the values for the selected column (e.g. 'Close') are collected into a list of arrays (*window_data*). Each array in this list will be used to generate one boxplot.
- **Statistical Visualisation:** The *plt.boxplot(window_data)* call then generates a boxplot for each window. Each plot visually summarises the data's median, quartiles, and potential outliers within that period. A tight box indicates low volatility and consolidation, while a tall box with long whiskers signifies high volatility and large price swings.

Customisation features enhances clarity, especially with many windows:

```
# Create boxplot: each box represents the distribution of prices in one window
box_plot = plt.boxplot(window_data,
                       labels=window_labels,  # X-axis labels
                       patch_artist=True,     # Fill boxes with color
                       showfliers=True)       # Show outliers

# Customize the appearance
plt.title(f'{company} {column} Price - {window_size}-Day Moving Window {chart_title_suffix}')
plt.xlabel('Window End Date')
plt.ylabel(f'{column} Price ($)')
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels for better readability
plt.grid(True, alpha=0.3)  # Add light grid

# Color the boxes
colors = ['lightblue', 'lightgreen', 'lightcoral', 'lightyellow', 'lightpink']
for i, box in enumerate(box_plot['boxes']):
    box.set_facecolor(colors[i % len(colors)])
```

The use of colour (*patch_artist=True*) and rotated x axis labels are practical touches that prevent the chart from becoming cluttered and unreadable.

Enhanced Data Cleaning and Robustness

A significant challenge addressed in v0.3 was ensuring data quality and compatibility with the visualisatioin libraries. Real world data from *yfinance* can contain MultiIndex columns, *NaN* values, or non numeric types, which cause plotting functions to fail. The following proactive data cleaning was implemented for this task based on these variables:

```
# Handle MultiIndex columns (common with yfinance)
# If columns are MultiIndex, convert to regular index
if isinstance(candlestick_data.columns, pd.MultiIndex):
    # Flatten the MultiIndex columns
    candlestick_data.columns = ['_'.join(col).strip() for col in candlestick_data.columns.values]
    # If there's only one ticker, remove the ticker suffix
    candlestick_data.columns = [col.split('_')[0] if '_' in col else col for col in candlestick_data.columns]

# Forward fill to replace NaN values with previous values
candlestick_data = candlestick_data.ffill()

# Backward fill to handle any remaining NaN values at the beginning
candlestick_data = candlestick_data.bfill()

# Ensure all required columns are numeric
required_columns = ['Open', 'High', 'Low', 'Close']
for col in required_columns:
    if col in candlestick_data.columns:
        # Make sure we're working with a Series
        if isinstance(candlestick_data[col], pd.Series):
            candlestick_data[col] = pd.to_numeric(candlestick_data[col], errors='coerce')
        else:
            # If it's not a Series, convert to Series first
            candlestick_data[col] = pd.to_numeric(pd.Series(candlestick_data[col]), errors='coerce')
```

This variable handling approach ensures the visualisation functions operate reliably, handling common data inconsistencies gracefully.

Challenges and Solutions

The development process presented several learning opportunities that required strategic problem solving.

- **Challenge 1: DatetimeIndex Incompatibility with Grouping.**
  - The initial attempt to group data using *data.index // n_days* resulted in a *TypeError* because a *DatetimeIndex* does not support integer division.
  - **Solution:** The problem was solved by creating a new, plain integer index using *np.arange(len(data))* to facilitate the grouping logic. This taught a valuable lessing in choosing the right tool for the job; a simple numeric index was more practical than a complex datetime index for this specific task.
- **Challenge 2: Data Type and Format Errors.**
  - The *mplfinance* library is strict about input data, throwing errors like *ValueError: Data for column 'Open' must be ALL float or int* when it encountered *NaN* values or string objects.
  - **Solution:** A comprehensive data cleaning pipeline was implemented. This included flattening MultiIndex columns, forward and backward filling *NaN* values, and explicitly converting data types to numeric. This undrescored the critical importance of data preprocessing, which often constitutes a majority of the effort in a data science project.
- **Challenge 3: Backwards Compatibility with Saved Data.**
  - After modifying the *load_process_data* function's return structure, the code failed with a *KeyError: 'feature_scalers'* when trying to load an old, cached *.pkl* file.
  - **Solution:** The old *.pkl* file was manually deleted, forcing the program to regenerate the data with the new structure. This highlighted a key consideration for software maintenance: changes to data structures can break backwards compatibility, and version control strategies should be considered for serialised data.

Figure 1

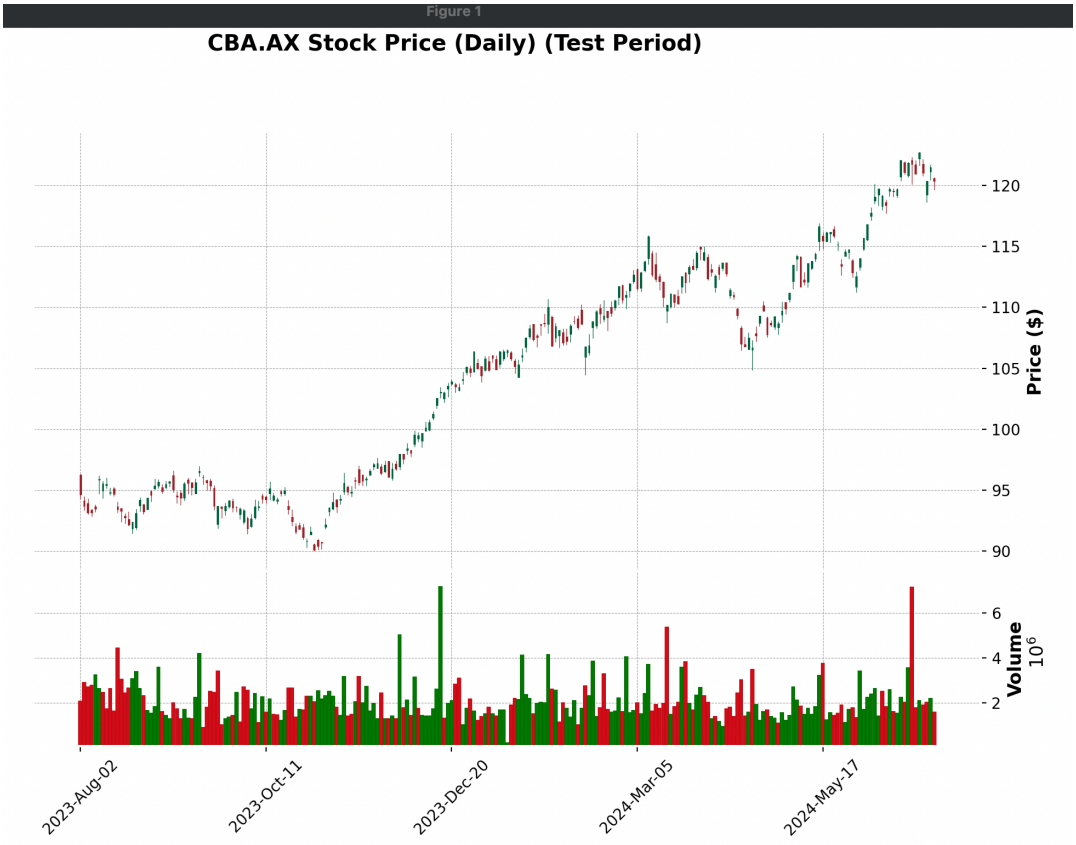
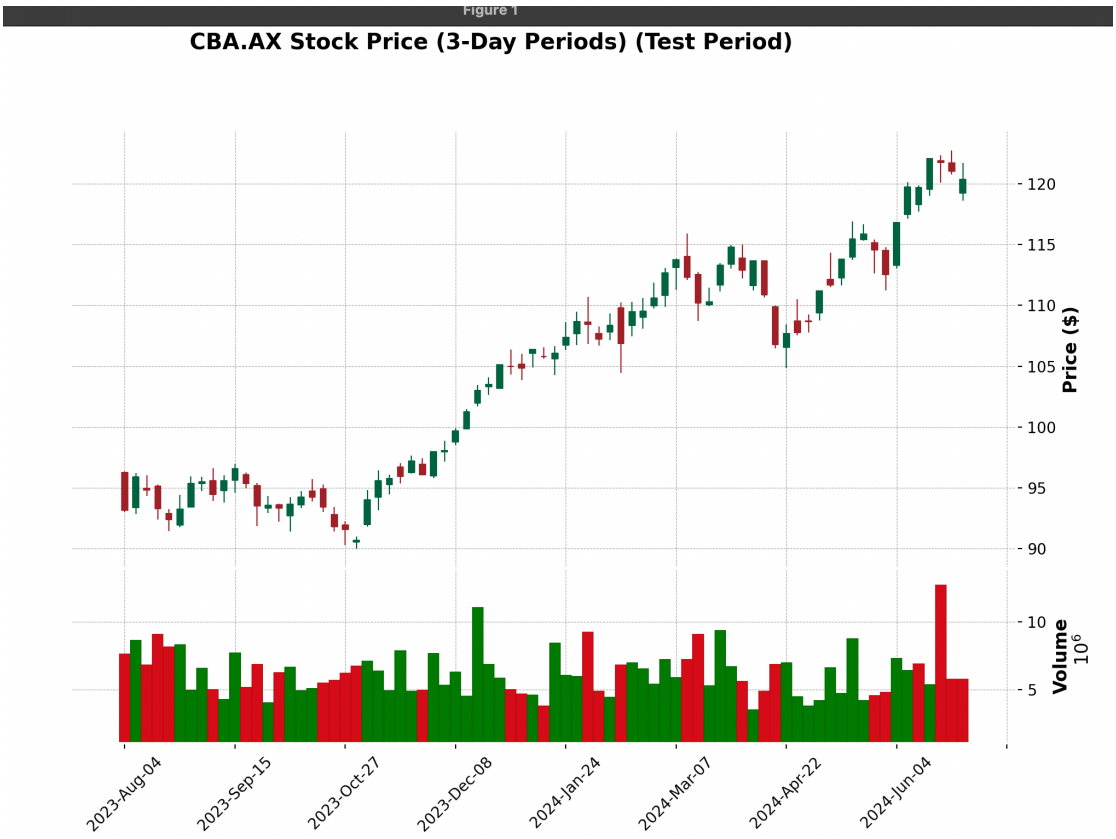


Figure 1



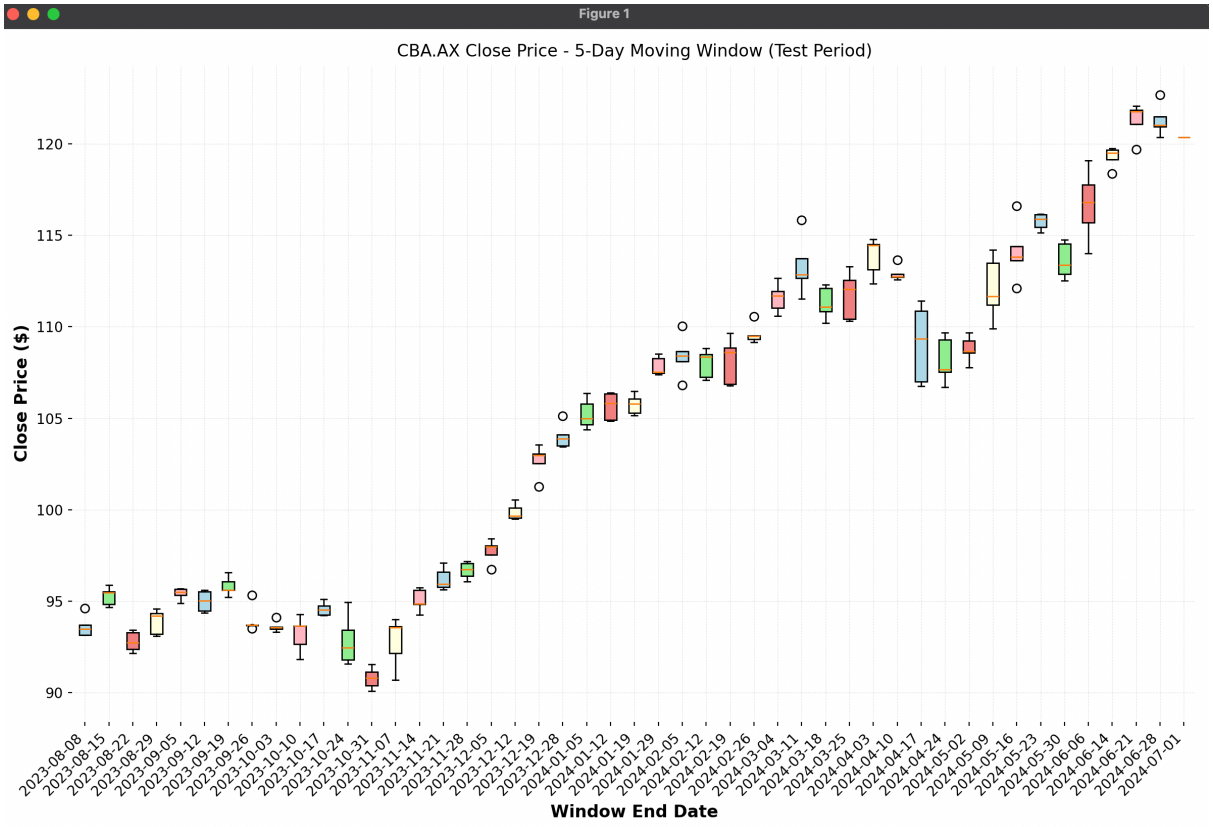


Figure 1
CBA.AX Close Price - 5-Day Moving Window (Test Period)
Close Price ($)
Window End Date

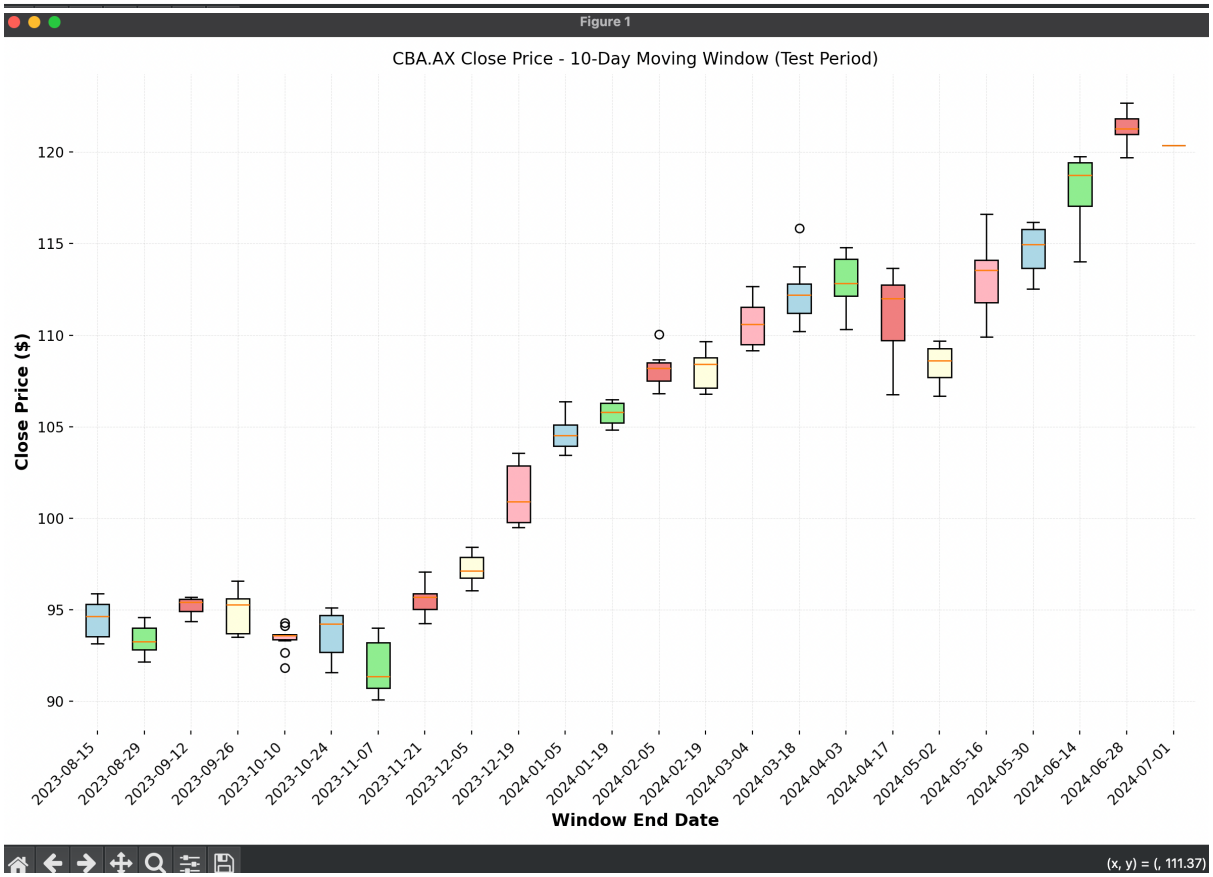


Figure 1
CBA.AX Close Price - 10-Day Moving Window (Test Period)
Close Price ($)
Window End Date
(x, y) = (, 111.37)

Reflection and Learning Outcomes

The implementation of v0.3 was a profound learning experience that extended far beyond simply adding new charts. It provided a practical immersion into the realities of financial data science.

- **Visualisation as a Narrative Tool:** I learned that effective visualisation is a vital part of data processing. A candlestick chart shows the constant battle between buyers and sellers within a period, while a boxplot shows stability or turmoil through statistical distribution. These provide context that a simple prediction line cannot.
- **The Primacy of Data Preprocessing:** This task reinforced that building models and charts is often the final step. The bulk of the work, and the key to a robust application, lies in the meticulous cleaning, validating, and structuring of the raw data.
- **Methodical Debugging as a Core Skill:** Each error encounted was initially a setback, but by systematically analysing the error messages, researching the libraries' expected inputs, and testing incremental fixes, these challenges were overcome. This process build significant confidence in troubleshooting complex code.

Conclusion

Version 0.3 of the stock prediction project successfully elevates the application's analytical and presentational standards. The introduction of professinally implemented candlestick and boxplot charts, backed by robust data preprocessing, provides users with powerful tools to interpret market behaviour. These visualisatoins offer insights into price volatility, trading ranges, and statistical distributions that are essential for informed decision making. The challenges overcome during development have resulted in a more mature, reliable, and insightful codebase, providing a strong foundation for future enhancements such as the integration of technical indicators or the development of a interactive web based dashboard.