Task 3 - Data Processing 2

Kelvin Dang - 104776732 - Swinburne University of Technology

Overview

In this week's task, I had to implement a few data visualization features for the **v0.2** version of the project, by writing functions having the ability to display stock market financial data using two useful kinds of chart, candlestick and boxplot. Furthermore, an option allowing each candlestick to represent the data of multiple trading days also needs to be included.

Candlestick Visualization

Candlestick chart is the most commonly used type of chart in financial markets to describe price movements and fluctuations of a security. While looking similar to a bar chart, it captures all four important price details (open, close, high, and low) in one single candlestick, with each one covering a specific time interval. The figure below from Wikipedia shows how information are represented in a candlestick.

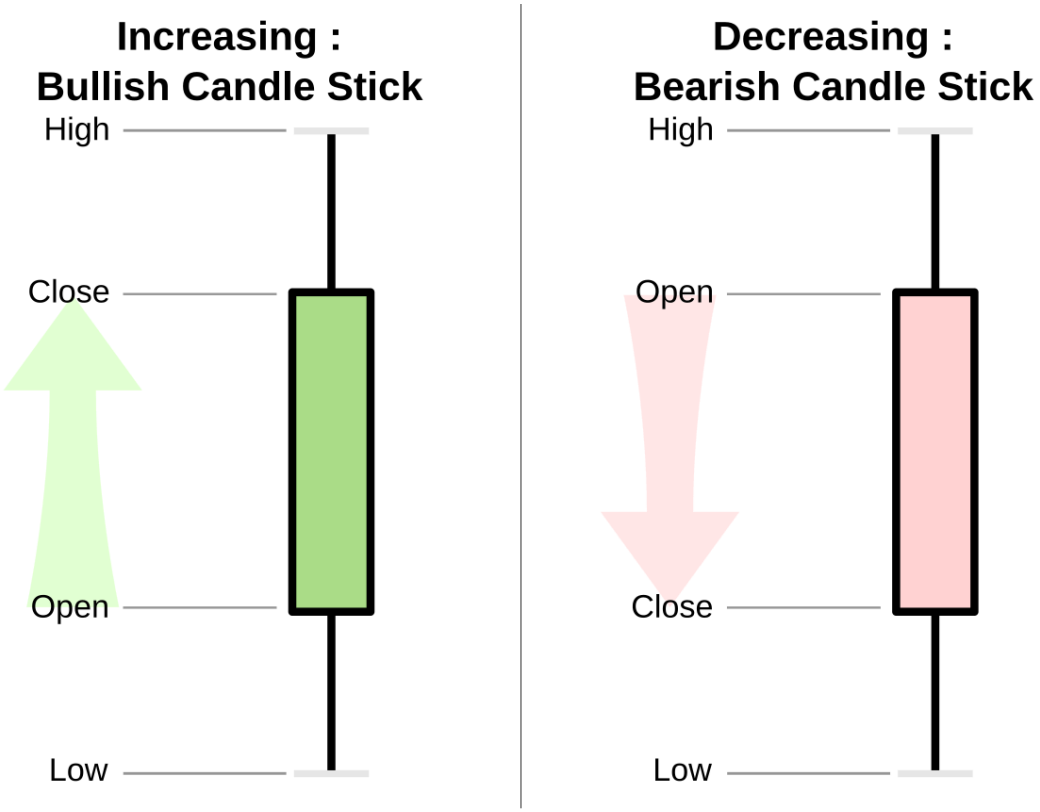

Figure 1: How a candlestick represents a particular trading pattern in a time interval.

In my attempt to implement this feature, I used the `mplfinance` library, which is a library built on top of `matplotlib` and can be used to generate static candlestick charts. With a new library to be used, I updated the `requirements.txt` file to include it, and used VSCode's Command Palette to create a new virtual environment with all required packages installed, ready for planned implementations.

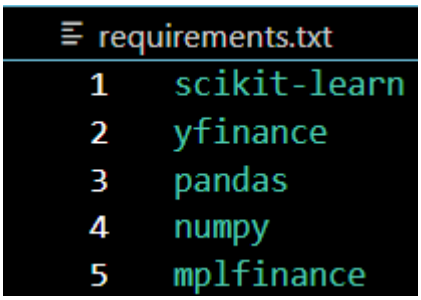

Figure 2: The updated `requirements.txt` file, including the newly added `mplfinance` library.

I also create a new Python module named `visualizations.py`, which will contain the two required visualization functions for this week's task. The candlestick visualization function would have the name of `candlestick_visualization()`, with its declaration and details about itself and the included arguments shown in the figure below. The feature allowing each candlestick to represent multiple

trading days was also planned to be implemented, with the `trading_days` argument specifying how many trading days to be exact.

```
def candlestick_visualization(data, trading_days=1, feature_columns=["Close", "High", "Low", "Open", "Volume"]):
    """
    Visualizes the stock data using a candlestick chart, with intervals defined by the trading_days parameter.
    Arguments:
        data (pd.DataFrame): The stock data DataFrame.
        trading_days (int): The number of trading days to group together for each candlestick (default is 1).
        feature_columns (list): The list of feature columns to use from the DataFrame (default includes common stock features).
    """
```

Figure 3: A brief look into the candlestick visualization function.

To begin with, the function would check two important things, whether the input DataFrame is in the right format as downloaded from Yahoo Finance, and whether the `trading_days` parameter value is an integer with a value not smaller than 1, as required by the task handout. This is how I designed the function to do these checks.

```
# Ensure the data is in the correct format.
for col in feature_columns:
    assert col in data.columns, f"Column \"{col}\" not found in data."
# Ensure the trading_days parameter is an integer not less than 1.
assert isinstance(trading_days, int) and trading_days >= 1, "\"trading_days\" must be an integer not less than 1."
```

Figure 4: Input data and parameter validation.

By checking that all common stock features as listed in the default value of the `feature_columns` argument are present in the input DataFrame, I could ensure that it is in the right format as having been retrieved from Yahoo Finance, ready to be used for visualizations. At the same time, by checking the `trading_days` parameter value against the constraints mentioned above, it can be confirmed to be ready for the upcoming processes, where I would have to do some tweaks to the DataFrame using it in order to fulfill the advanced visualization requirements.

As we knew, the stock data from Yahoo Trading is listed on a daily basis. Therefore, for each candlestick to be able to obtain information from multiple consecutive days of trading, I would have to group them up, and aggregate their data values together. The code block below would effectively complete those tasks.

```python
# Group the data into windows of "trading_days" days.
group = (pd.Series(range(len(data)), index=data.index) // trading_days)
data_modified = data.groupby(group).agg({
    "Close": "last",
    "High": "max",
    "Low": "min",
    "Open": "first",
    "Volume": "sum"
})
# Set the index to the last date in each group.
data_modified.index = data.groupby(group).apply(lambda x: x.index[-1])
```

Figure 5: Data aggregation by multiple consecutive trading days.

After having been gathered together in groups of several trading days (the number is predetermined by the respective parameter value), aggregation functions would be used to decide the common stock values of each interval.

- The “Close” value of each interval will be that of the last included trading day.
- The “High” value of each interval will be that belonging to one of the included trading days having the maximum value.
- The “Low” value of each interval will be that belonging to one of the included trading days having the minimum value.
- The “Open” value of each interval will be that of the first included trading day.
- The “Volume” value will be computed by adding up those coming from all the included trading days.

After having all feature values decided, the index of each group would be set to the last date in each one, instead of auto-decided index values. A short lambda function returning the last index value of each group is used to accomplish this.

In the process of put this into action, I encountered a KeyError regarding the above `agg()` call, telling me that none of the feature columns were detected.

```
KeyError: "Column(s) ['Close', 'High', 'Low', 'Open', 'Volume'] do not exist"
```

Figure 6: KeyError due to unavailability of all feature columns.

The reason is the input data had a multi-level index (”Price”, “Ticker”) in place. Due to the “Ticker” value being similar among all entries, I eliminated it and

hence reduced the index to being single-level by adding the `multi_level_index` parameter into the `yf.download()` function call in the `stock_prediction.py` module (mentioned in Task 2), with the value of `False` indicating the desired single-level index format.

```
  df = yf.download(ticker, start=start_date, end=end_date)
[**********************100%**********************]  1 of 1 completed
Price            Close       High        Low       Open   Volume
Ticker         CBA.AX     CBA.AX     CBA.AX     CBA.AX   CBA.AX
Date
2020-01-02  63.946331  64.218514  63.554073  63.874286  1416232
2020-01-03  64.290543  64.995013  64.242513  64.818895  1622784
2020-01-06  63.858269  63.946328  63.425988  63.826253  2129260
2020-01-07  65.003029  65.003029  64.186491  64.698830  2417468
2020-01-08  64.762863  65.043047  64.066400  65.019032  1719114
```

Figure 7: The previous version of the downloaded stock data, having the multi-level index in place.

```
# Download stock data from Yahoo Finance.
df = yf.download(ticker, start=start_date, end=end_date, multi_level_index=False)
```

Figure 8: The modified `yf.download()` call, with the `multi_level_index` parameter set to `False`.

```
                 Close        High         Low        Open   Volume
Date
2020-01-02   63.946316   64.218499   63.554058   63.874271  1416232
2020-01-03   64.290550   64.995020   64.242520   64.818903  1622784
2020-01-06   63.858250   63.946308   63.425969   63.826234  2129260
2020-01-07   65.003014   65.003014   64.186476   64.698815  2417468
2020-01-08   64.762863   65.043047   64.066400   65.019032  1719114
...                ...         ...         ...         ...      ...
2025-07-25  170.242325  170.971082  168.371211  170.764267  1626092
2025-07-28  172.241470  172.241470  168.991643  169.877957  1249203
2025-07-29  171.640747  171.985433  169.907505  171.236991  1313893
2025-07-30  174.299713  174.900442  170.616557  171.325614  1817808
2025-07-31  175.205719  175.481462  173.433077  174.417876  1918660

[1412 rows x 5 columns]
```

Figure 9: The single-level index format of the downloaded stock data, ready for visualizations.

With this manipulation completed, the data grouping and aggregation process ran successfully, returning the below DataFrame.

	Close	High	Low	Open	Volume
2020-01-10	66.043701	66.043701	63.425977	63.874256	15194506
2020-01-21	66.828232	68.013002	65.427320	65.723512	15541981
2020-01-31	68.253159	68.821534	66.836236	66.844243	22642366
2020-02-11	67.820885	68.045034	66.668121	67.364578	18550888
2020-02-20	71.920166	72.888231	69.005662	69.005662	36609128
...	...	...	...	...	...
2025-06-25	188.490662	189.081547	175.314052	176.279151	17572194
2025-07-04	175.294357	188.963386	173.758072	187.594515	16931536
2025-07-15	177.027603	178.829783	174.151987	174.870896	9578439
2025-07-24	170.833206	180.681212	168.075774	175.294352	15603000
2025-07-31	175.205719	175.481462	168.371211	170.764267	7925656

Figure 10: The grouped and aggregated data, with `trading_days` set at 7.

At the moment when all the preparations are done, I was ready to plot the candlestick chart using `mplfinance`, by including a single `mpf.plot()` call, as shown below.

```
# Plot the candlestick chart.
mpf.plot(
    data_modified,
    type="candle",
    style="binance",
    title=f"{TICKER} - Candlestick Chart ({trading_days}-Day Intervals)",
    ylabel="Price",
    volume=True,
    ylabel_lower="Shares\nTraded",
)
```

Figure 11: The `mpf.plot()` call to plot the candlestick chart.

The type of the chart is specified as “candle” equivalent to candlestick chart, along with a semantic title and a section expressing volumes which is added by setting the “volume” parameter to `True`. This is how the chart turned out.

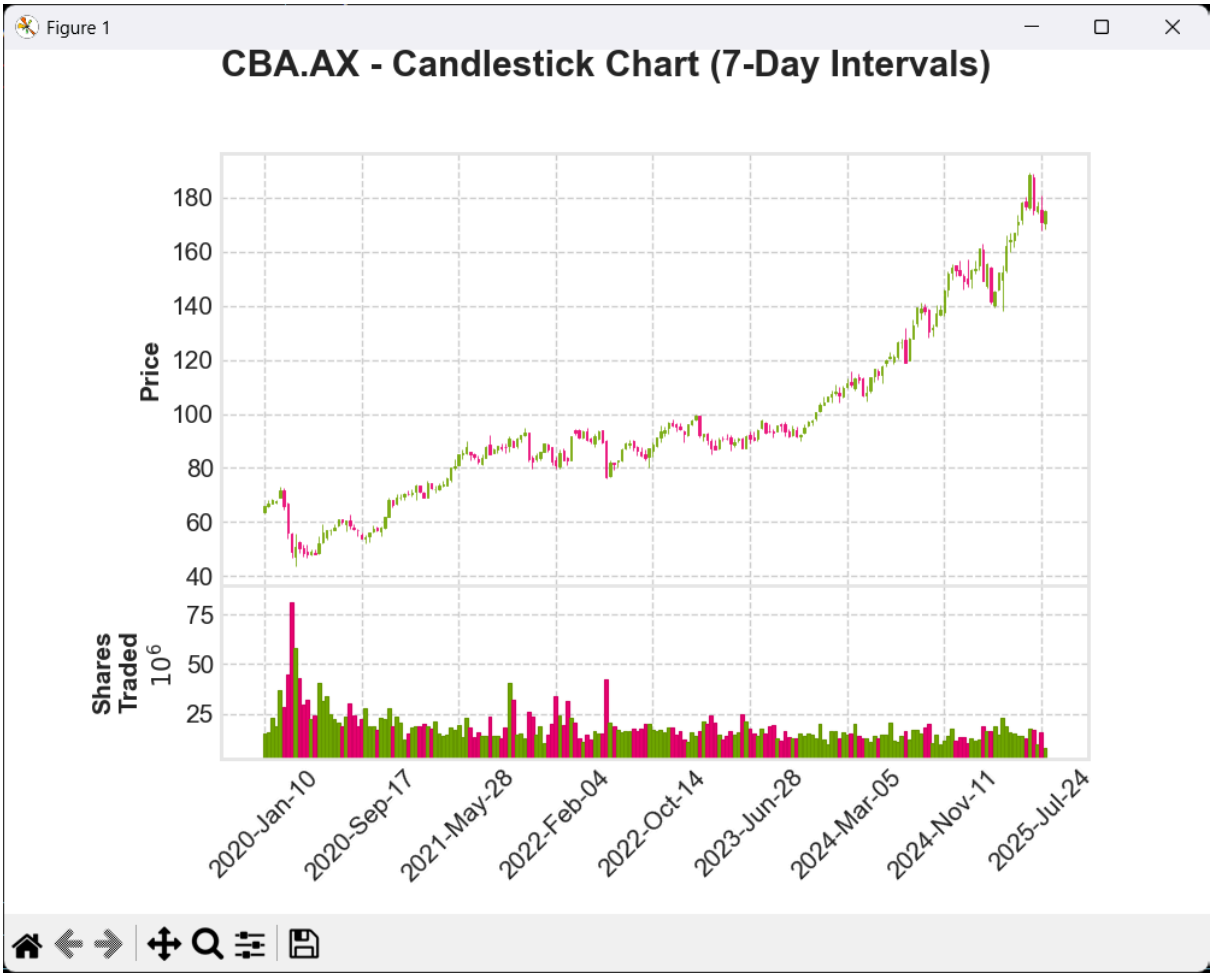

Figure 12: The output candlestick chart, with each candlestick expressing the aggregate data of 7 consecutive trading days.

This successful output concludes the candlestick visualization process, with the additional requirement of multiple trading days represented in each single candlestick fulfilled.

Boxplot Visualization

Boxplot chart is a descriptive and expressive type of chart whose information about the respective data includes locality, spread, and skewness, graphically demonstrated through their quartiles. As stated in the task brief, this is particularly useful for displaying data for a moving range of multiple consecutive trading days.

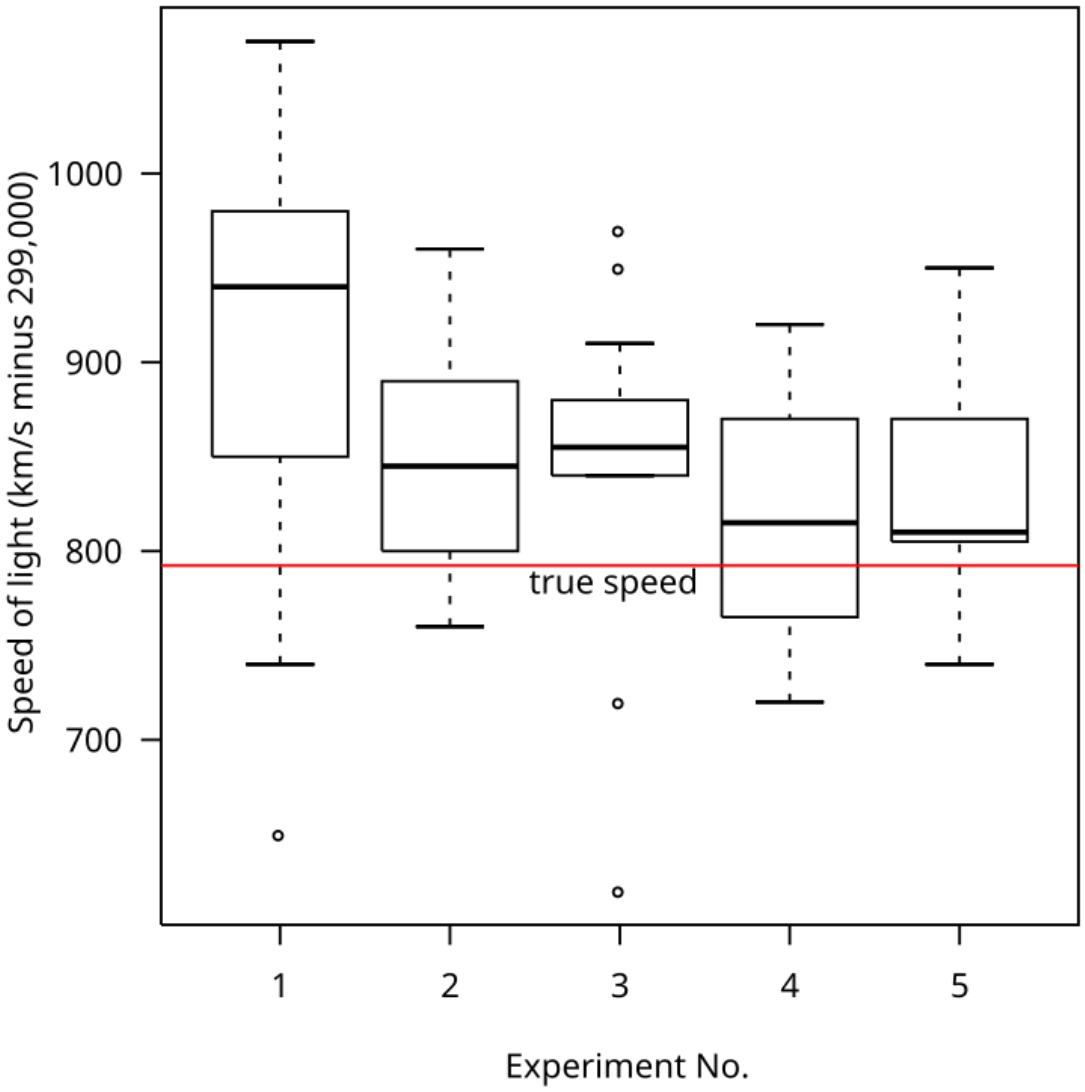

Figure 13: An example boxplot chart.

For implementing this function, I make use of a new library called `seaborn`, a statistical data visualization library which is also based on `matplotlib` underneath. In order to kick off implementations, I had to update the `requirements.txt` file once more to include the new library, and then used VSCode's Command Palette to create a new virtual environment with all required packages installed and ready.

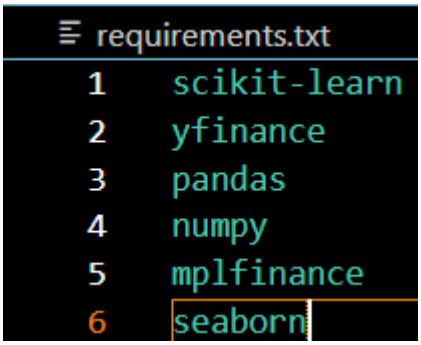

Figure 14: The updated `requirements.txt`, including the newly added `seaborn` library.

Another function was defined in the `visualizations.py` module for the boxplot visualization subtask, named `boxplot_visualization()`, with its declaration and details about itself and its arguments, along with its full functionality implementation shown in the figure below.

```
def boxplot_visualization(data, price_columns=["Close", "High", "Low", "Open"]):
    """
    Visualizes the stock data using a boxplot for the specified price columns.
    Arguments:
        data (pd.DataFrame): The stock data DataFrame.
        price_columns (list): The list of price columns to visualize (default includes common price features).
    """
    # Ensure the data is in the correct format.
    for col in FEATURE_COLUMNS:
        assert col in data.columns, f"Column \"{col}\" not found in data."
    # Plot the boxplot.
    sns.boxplot(data=data[price_columns], orient="h")
    # Show the plot.
    plt.show()
```

Figure 15: The `boxplot_visualization()` function.

Accompanied by the input data, an argument called `price_columns` is added, denoting which category of price values would be included in the boxplot. By default, all four common stock price values would be visualized.

After completing the data validation check, which is the same as the one included in the candlestick visualization function, a single `sns.boxplot()` call with the “orient” parameter set to “h” meaning horizontal showing, followed by a `plt.show()` call to show the plot, would complete the subtask of boxplot visualization, thus successfully achieve completion for all of this week’s tasks.

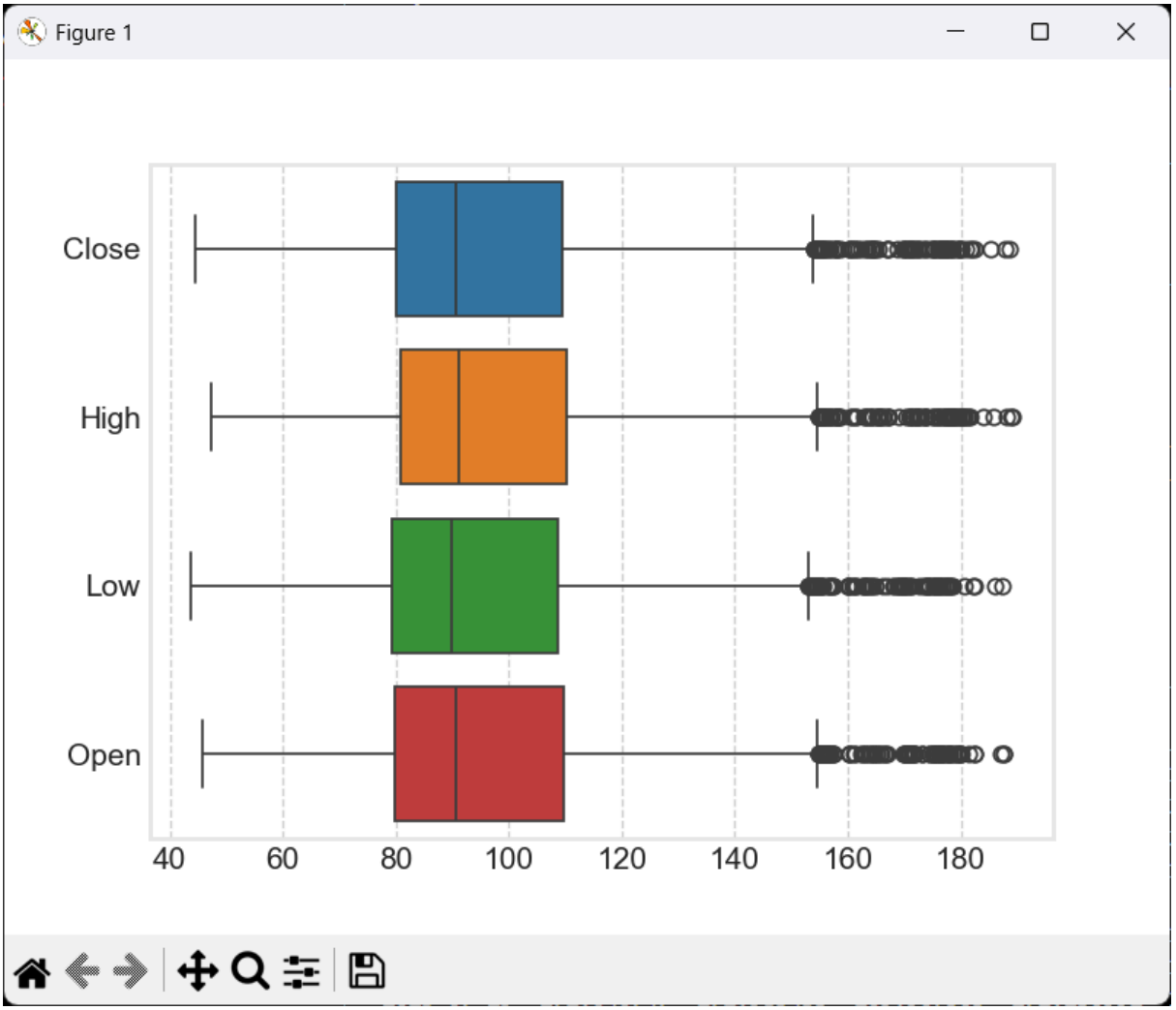

Figure 16: The output boxplot chart, with 4 distinct price categories as shown.

References

- Candlestick visualization tutorial on Python using `mplfinance`.

  https://coderzcolumn.com/tutorials/data-science/candlestick-chart-in-python-mplfinance-plotly-bokeh

- Candlestick chart on Wikipedia.

  https://en.wikipedia.org/wiki/Candlestick_chart

- Boxplot chart on Wikipedia.

  https://en.wikipedia.org/wiki/Box_plot