COS30049 - Intelligent System

# DATA PROCESSING

TRAN DUC ANH DANG

Student ID: 103995439

## Requirements

### Virtual Environment

- Google Colab
- Jupyter Notebook

### Decencies

- numpy
- matplotlib
- mplfinance
- pandas
- scikit-learn
- pandas-datareader
- yfinance
- pandas_ta
- joblib

## Installation

*Note: Anaconda is required unless Google Collab is being used

### Anaconda/Virtual Environment

1. Download Anaconda: Go to the Anaconda website (https://www.anaconda.com/products/distribution) and download the appropriate version for your operating system.
2. Install Anaconda: Follow the installation instructions for the OS from the Anaconda website.
3. Open Anaconda Navigator: Launch Anaconda Navigator from your installed applications.
4. Create a New Environment (Required): Create a new environment to isolate Jupyter installation on each project. Click on "Environments" in Navigator and then "Create" to make a new environment.
5. Install Jupyter Notebook: In the selected environment, click on the environment name and select "Open Terminal". In the terminal, type: conda install jupyter.

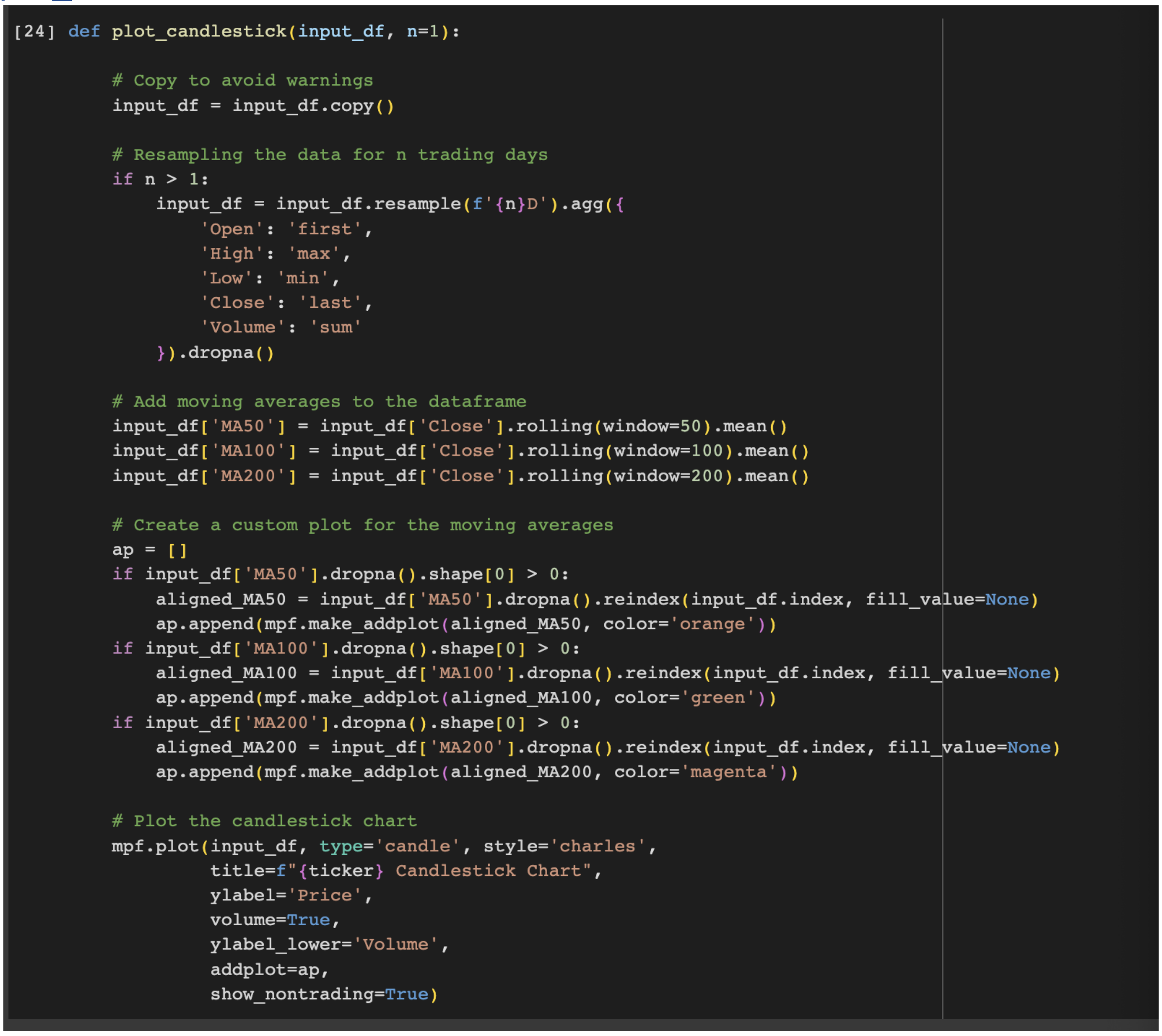
### Dependencies

In Google Colab or Jupyter Notebook, it can directly install the required dependencies using the !pip command in code cells. Here's an example of how to install the dependencies:

**!pip install <package> or !pip install -r <text file>**

## Data Processing 2

### plot_candlestick



```
[24] def plot_candlestick(input_df, n=1):

         # Copy to avoid warnings
         input_df = input_df.copy()

         # Resampling the data for n trading days
         if n > 1:
             input_df = input_df.resample(f'{n}D').agg({
                 'Open': 'first',
                 'High': 'max',
                 'Low': 'min',
                 'Close': 'last',
                 'Volume': 'sum'
             }).dropna()

         # Add moving averages to the dataframe
         input_df['MA50'] = input_df['Close'].rolling(window=50).mean()
         input_df['MA100'] = input_df['Close'].rolling(window=100).mean()
         input_df['MA200'] = input_df['Close'].rolling(window=200).mean()

         # Create a custom plot for the moving averages
         ap = []
         if input_df['MA50'].dropna().shape[0] > 0:
             aligned_MA50 = input_df['MA50'].dropna().reindex(input_df.index, fill_value=None)
             ap.append(mpf.make_addplot(aligned_MA50, color='orange'))
         if input_df['MA100'].dropna().shape[0] > 0:
             aligned_MA100 = input_df['MA100'].dropna().reindex(input_df.index, fill_value=None)
             ap.append(mpf.make_addplot(aligned_MA100, color='green'))
         if input_df['MA200'].dropna().shape[0] > 0:
             aligned_MA200 = input_df['MA200'].dropna().reindex(input_df.index, fill_value=None)
             ap.append(mpf.make_addplot(aligned_MA200, color='magenta'))

         # Plot the candlestick chart
         mpf.plot(input_df, type='candle', style='charles',
                  title=f"{ticker} Candlestick Chart",
                  ylabel='Price',
                  volume=True,
                  ylabel_lower='Volume',
                  addplot=ap,
                  show_nontrading=True)
```

The function `plot_candlestick` takes the following parameters:

- `input_df`: This parameter represents the input DataFrame containing financial data that you want to plot as a candlestick chart.
- `n`: This is an optional parameter (defaulting to 1) that indicates the resampling period in trading days for aggregating the data.

This function has the following features:

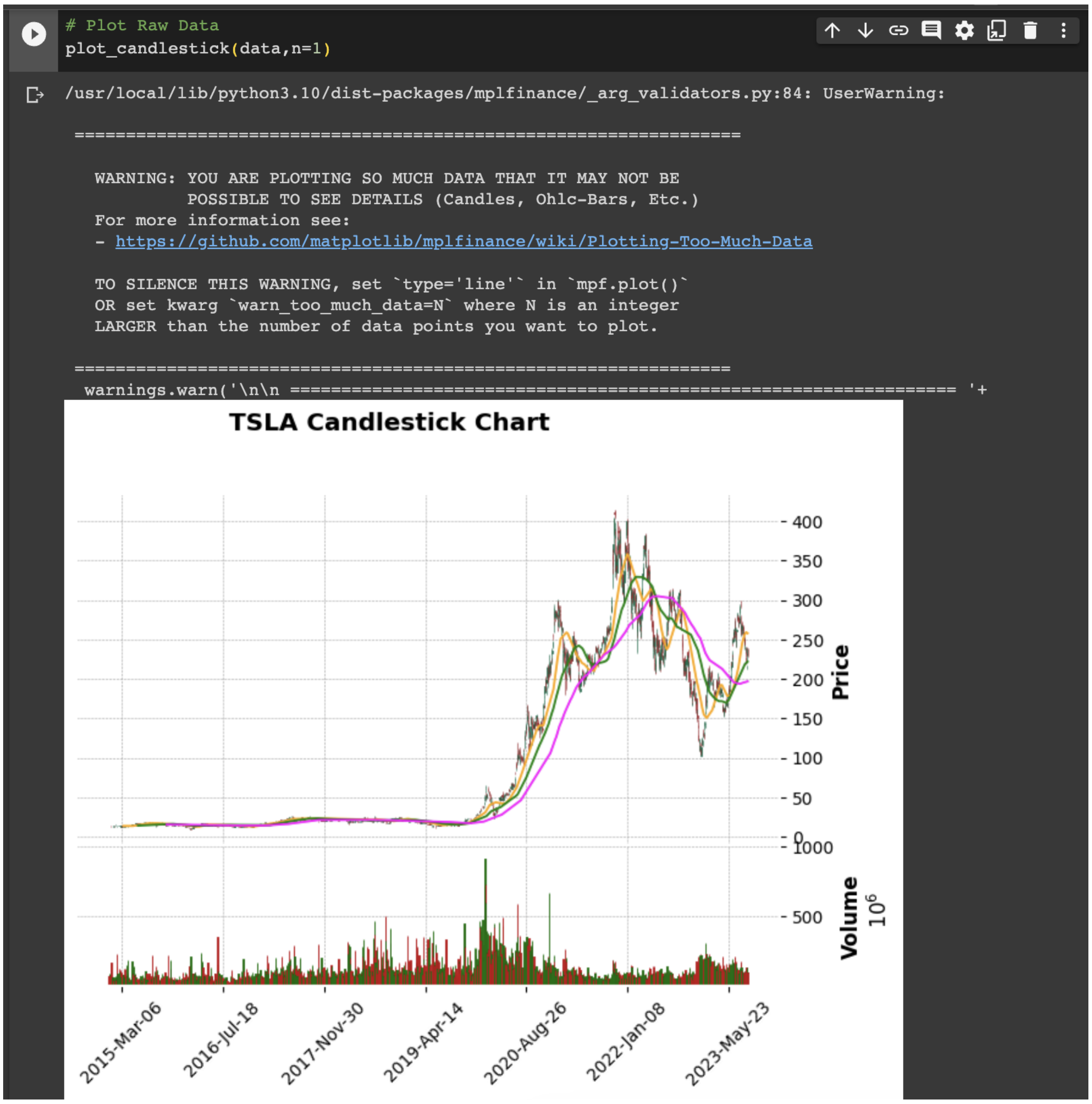
- Copying the DataFrame: The input DataFrame is copied to avoid modifying the original data. The purpose of copying is to prevent any unintended modifications and warnings.
- Resampling Data: If `n` is greater than 1, the function resamples the data based on the specified trading days (`n`). It aggregates the data to find the first open, maximum high, minimum low, last close, and sum of volumes within each resampling period.
- Adding Moving Averages: The function calculates three moving averages (MA) for the 'Close' price: a 50-day moving average (`MA50`), a 100-day moving average

(`MA100`), and a 200-day moving average (`MA200`). These moving averages are added as columns to the DataFrame.

- Creating Additional Plot Elements: The function creates a list of addplot objects using the moving averages (`MA50`, `MA100`, `MA200`) as additional plot elements. These will be displayed alongside the main candlestick chart.
- Plotting the Chart: The function uses the `mpf.plot` function from the mplfinance library to plot the candlestick chart. Various parameters are specified for chart appearance, including chart type, title, labels, volume bars, and the additional plot elements created earlier.

## plot_candlestick output results

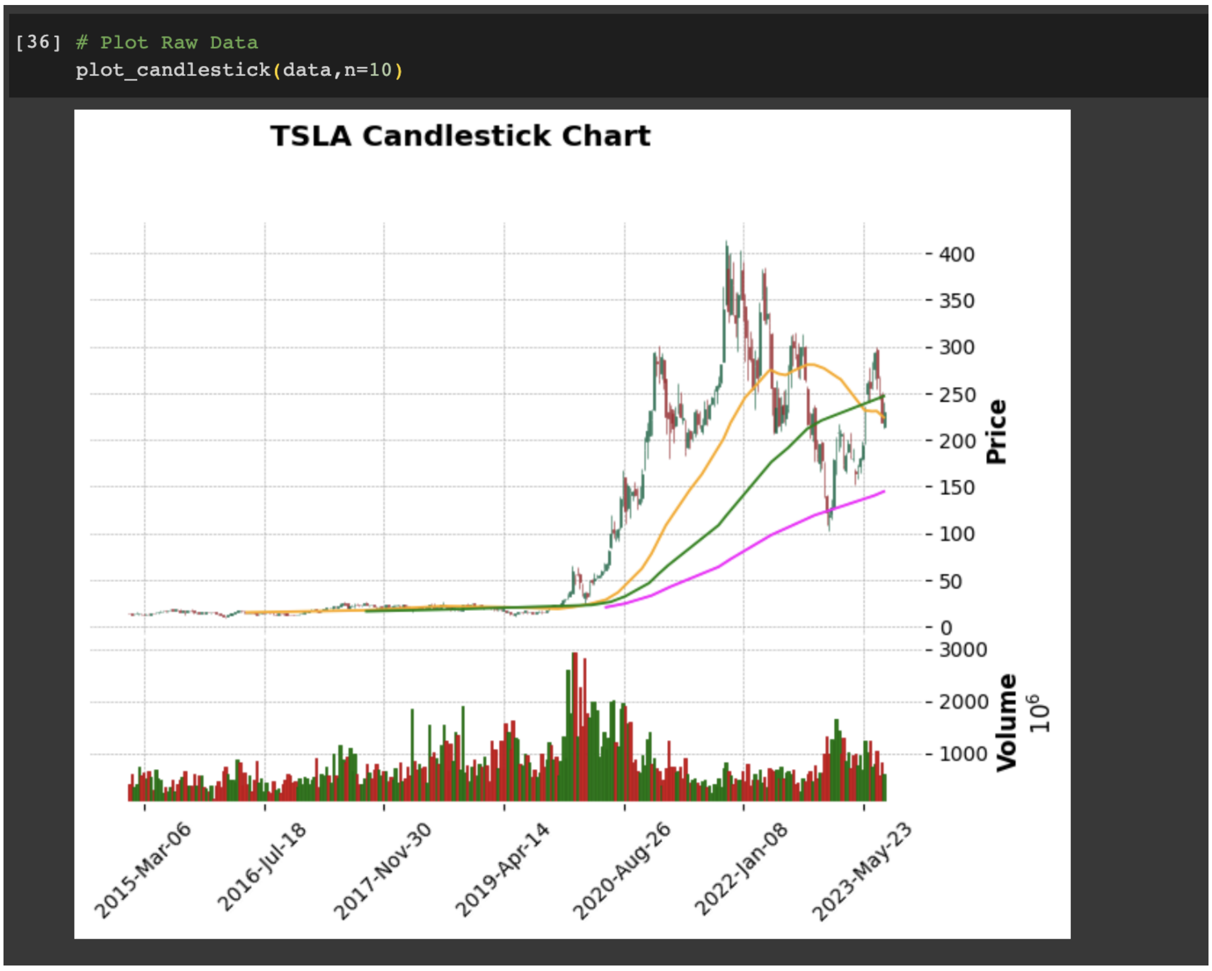
**n = 1**



```
# Plot Raw Data
plot_candlestick(data,n=1)
```

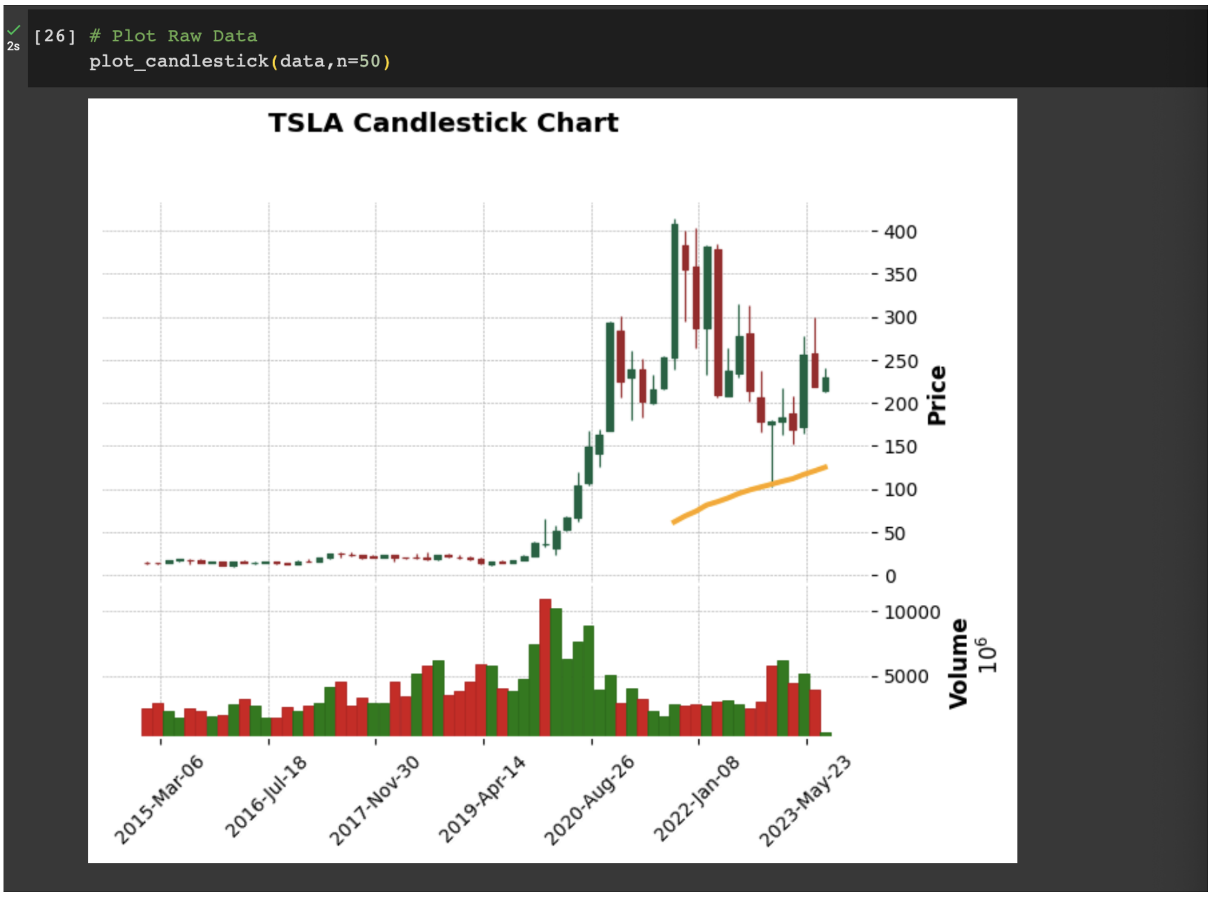
```
/usr/local/lib/python3.10/dist-packages/mplfinance/_arg_validators.py:84: UserWarning:

 ================================================================

   WARNING: YOU ARE PLOTTING SO MUCH DATA THAT IT MAY NOT BE
            POSSIBLE TO SEE DETAILS (Candles, Ohlc-Bars, Etc.)
   For more information see:
   - https://github.com/matplotlib/mplfinance/wiki/Plotting-Too-Much-Data

   TO SILENCE THIS WARNING, set `type='line'` in `mpf.plot()`
   OR set kwarg `warn_too_much_data=N` where N is an integer
   LARGER than the number of data points you want to plot.

 ================================================================
  warnings.warn('\n\n ================================================================ '+
```

**n = 10**



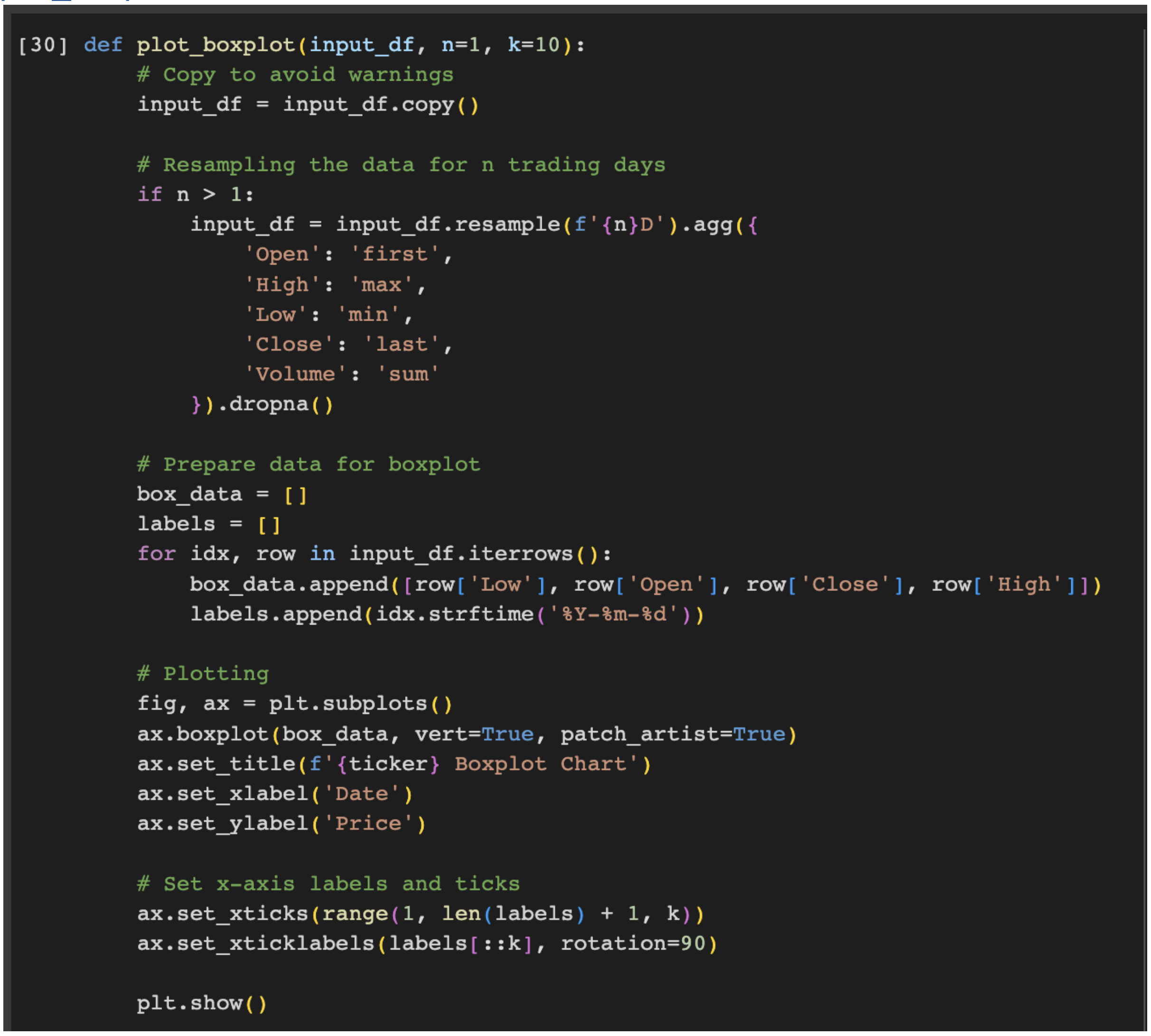
```
[36] # Plot Raw Data
     plot_candlestick(data,n=10)
```

## n = 50



```
[26] # Plot Raw Data
     plot_candlestick(data,n=50)
```

## plot_boxplot



```
[30] def plot_boxplot(input_df, n=1, k=10):
         # Copy to avoid warnings
         input_df = input_df.copy()

         # Resampling the data for n trading days
         if n > 1:
             input_df = input_df.resample(f'{n}D').agg({
                 'Open': 'first',
                 'High': 'max',
                 'Low': 'min',
                 'Close': 'last',
                 'Volume': 'sum'
             }).dropna()

         # Prepare data for boxplot
         box_data = []
         labels = []
         for idx, row in input_df.iterrows():
             box_data.append([row['Low'], row['Open'], row['Close'], row['High']])
             labels.append(idx.strftime('%Y-%m-%d'))

         # Plotting
         fig, ax = plt.subplots()
         ax.boxplot(box_data, vert=True, patch_artist=True)
         ax.set_title(f'{ticker} Boxplot Chart')
         ax.set_xlabel('Date')
         ax.set_ylabel('Price')

         # Set x-axis labels and ticks
         ax.set_xticks(range(1, len(labels) + 1, k))
         ax.set_xticklabels(labels[::k], rotation=90)

         plt.show()
```

The function `plot_boxplot` takes the following parameters:

- `input_df`: This parameter represents the input DataFrame that contains the financial data you want to visualize as a boxplot chart.
- `n`: This is an optional parameter, defaulting to 1, that specifies the resampling period in trading days for aggregating the data.
- `k`: This is an optional parameter, defaulting to 10, that specifies the interval for displaying x-axis labels to avoid clutter.

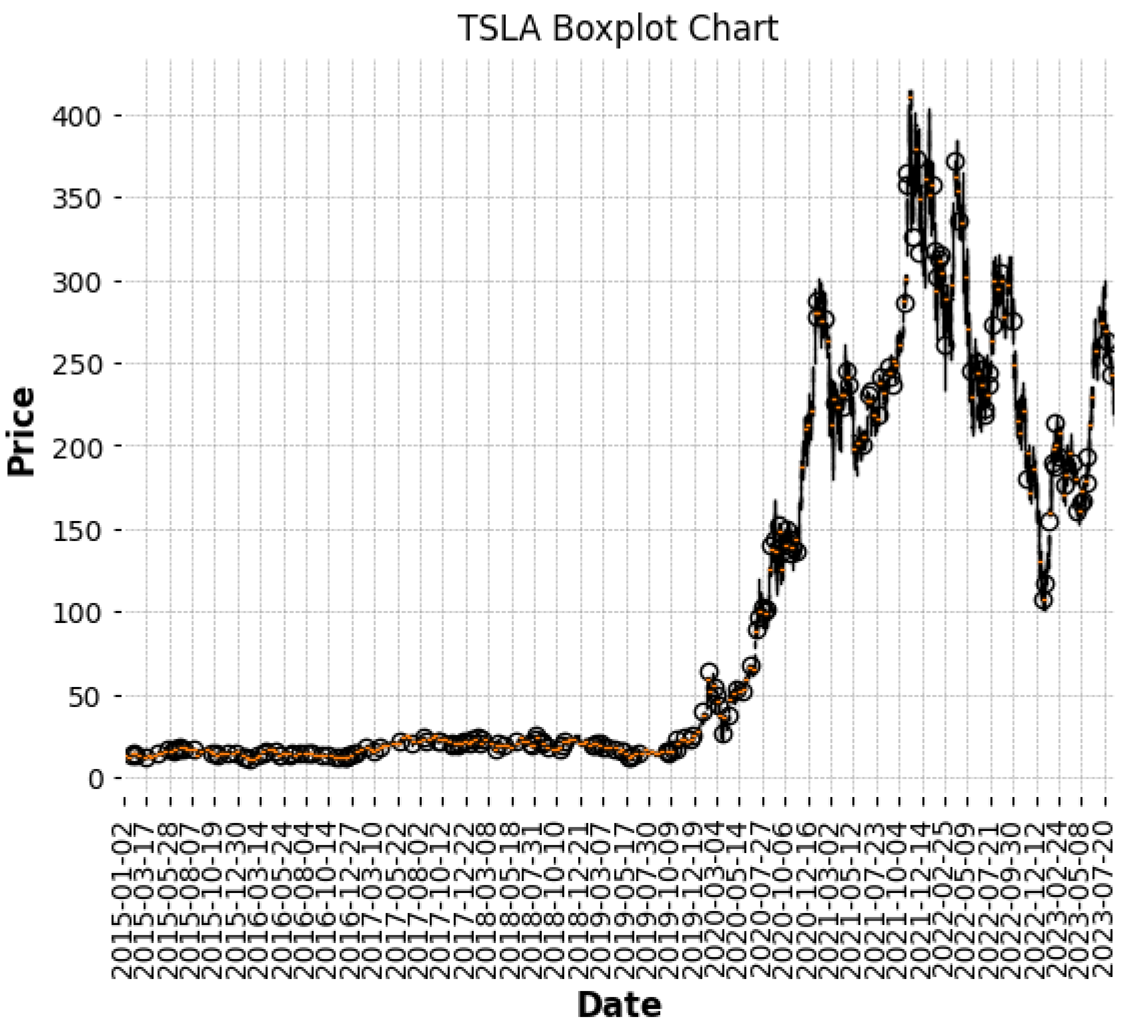
This function has the following features:

- Copying the DataFrame: The input DataFrame is copied to avoid any unintended modification of the original data and to prevent warnings.
- Resampling Data: If `n` is greater than 1, the function resamples the data to aggregate `n` trading days into one box. It finds the first open, maximum high, minimum low, last close, and sum of volumes for each resampling period.
- Preparing Data for Boxplot: A list `box_data` is prepared, which contains lists of the ['Low', 'Open', 'Close', 'High'] prices for each period (resampled if `n > 1`). Additionally, a list `labels` contains the corresponding dates.

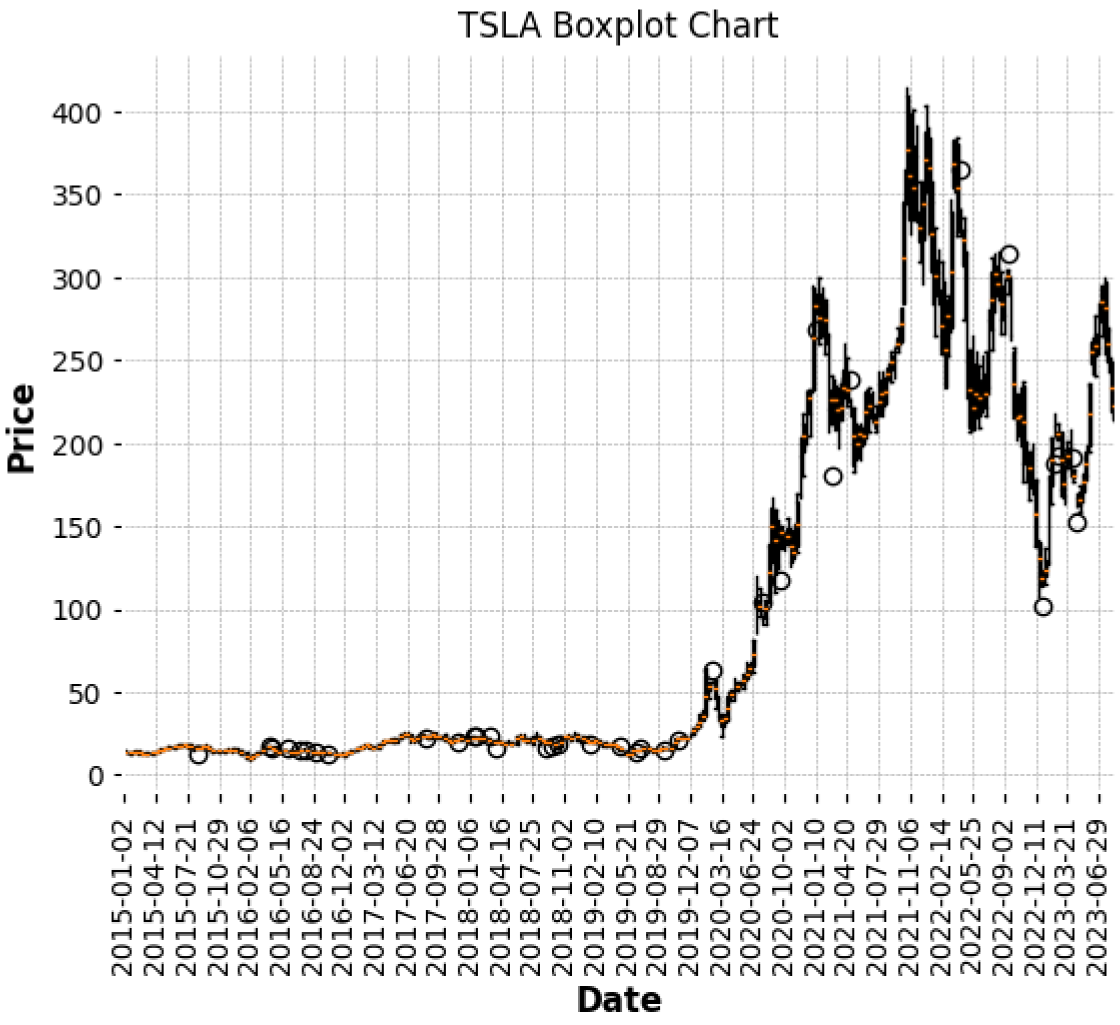
- Plotting the Chart: The function uses matplotlib's `boxplot` function to plot the boxplot chart. Various parameters like orientation (`vert=True`) and patch artist (`patch_artist=True`) are set to control the appearance of the boxplot.
- Setting X-axis Labels: To avoid clutter, x-axis labels are displayed at intervals specified by the parameter `k`. The labels are also rotated for better visibility.
- Setting Titles and Labels: The title of the plot, as well as the x-axis and y-axis labels, are set for better understanding of the visualized data.

## plot_boxplot output results

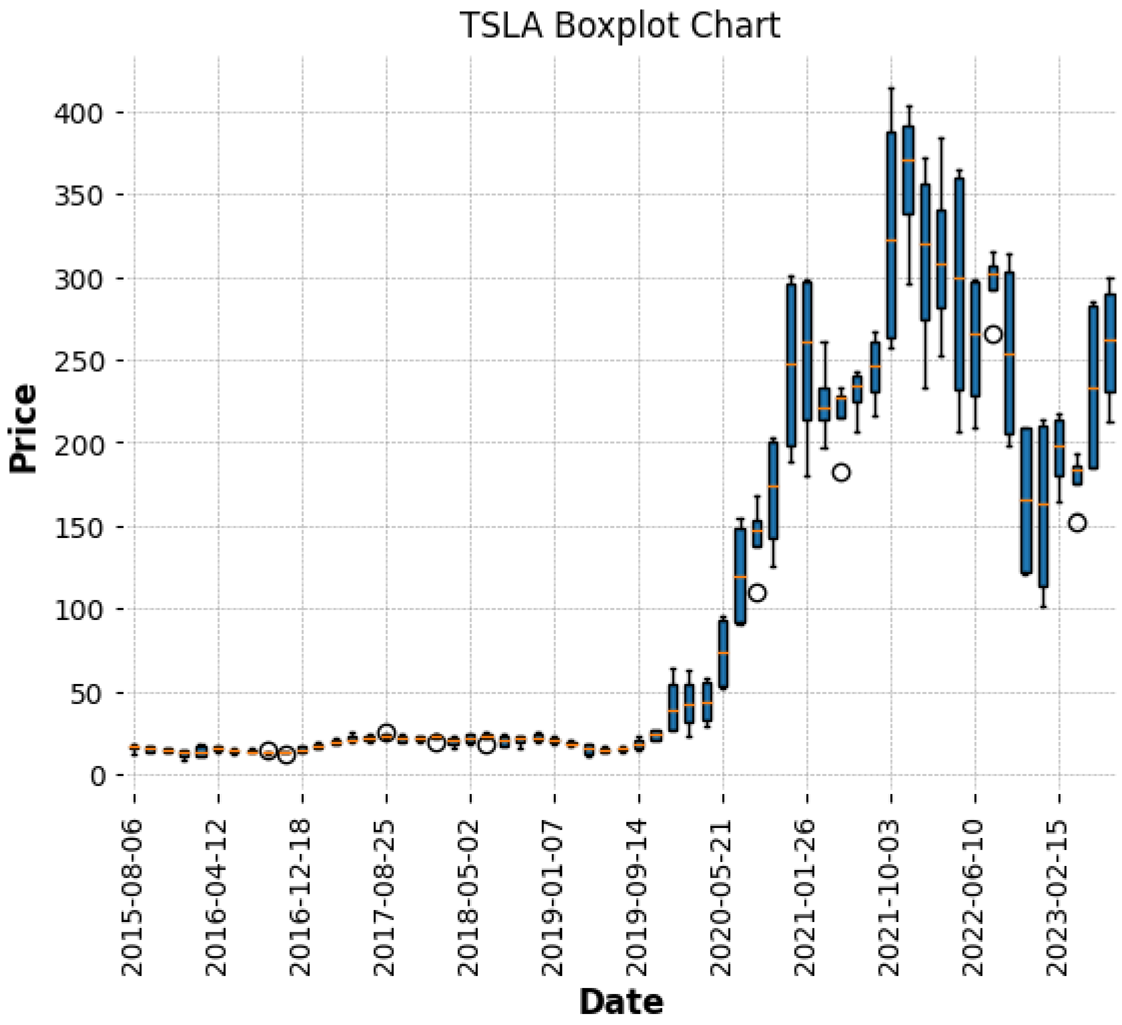
**n = 1, k=50**



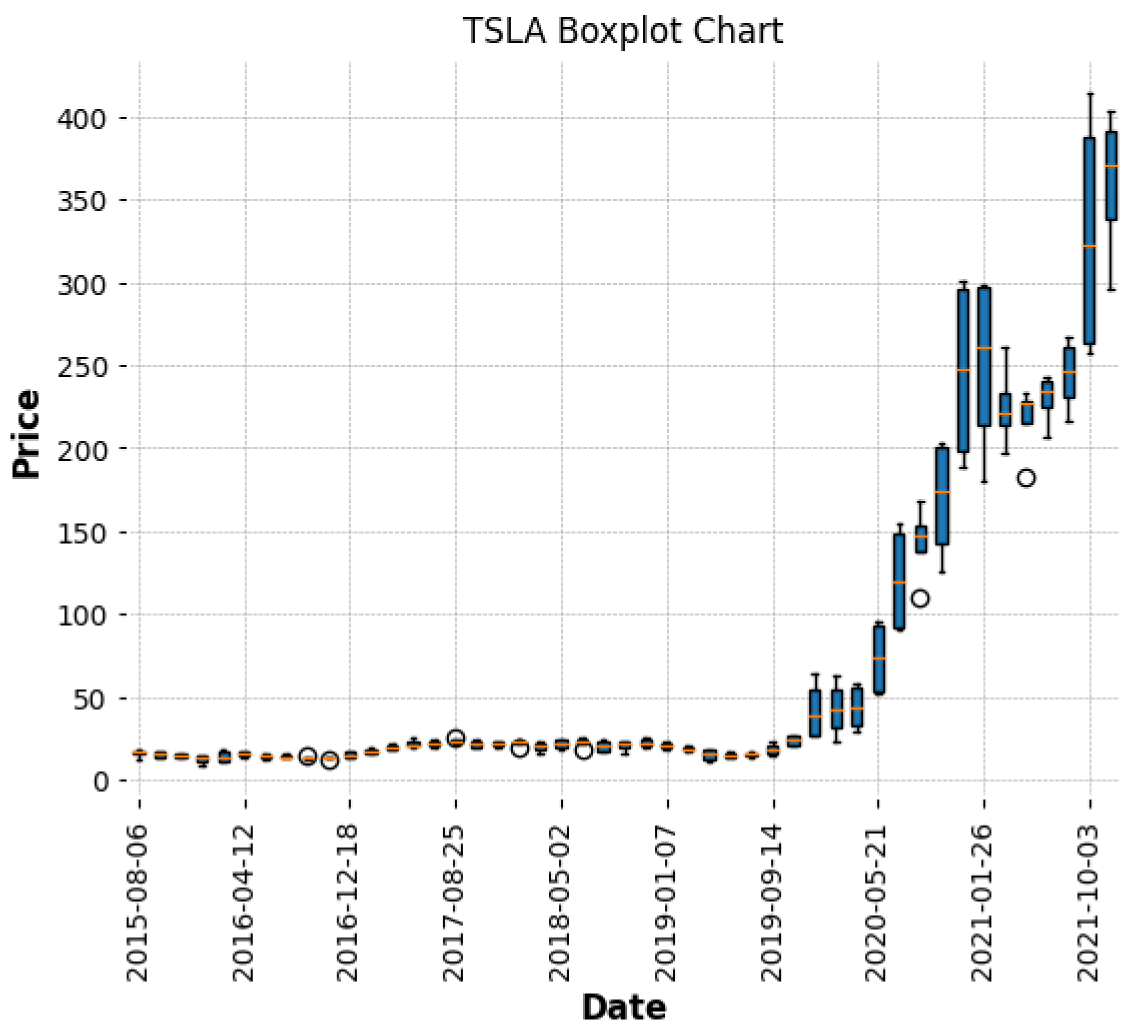
**n = 10, k=10**


TSLA Boxplot Chart
Price
400
350
300
250
200
150
100
50
0
2015-01-02
2015-04-12
2015-07-21
2015-10-29
2016-02-06
2016-05-16
2016-08-24
2016-12-02
2017-03-12
2017-06-20
2017-09-28
2018-01-06
2018-04-16
2018-07-25
2018-11-02
2019-02-10
2019-05-21
2019-08-29
2019-12-07
2020-03-16
2020-06-24
2020-10-02
2021-01-10
2021-04-20
2021-07-29
2021-11-06
2022-02-14
2022-05-25
2022-09-02
2022-12-11
2023-03-21
2023-06-29
Date

**n = 50, k=5**


TSLA Boxplot Chart
Price
400
350
300
250
200
150
100
50
0
2015-08-06
2016-04-12
2016-12-18
2017-08-25
2018-05-02
2019-01-07
2019-09-14
2020-05-21
2021-01-26
2021-10-03
2022-06-10
2023-02-15
Date

**n = 50, k=5**



**Relationship between n and k:**

When n decreases:

- When resampling the data over a smaller window, which will result in more data points (more boxes in the boxplot).
- With more data points, the x-axis will become more crowded.

When k increases:

- Fewer x-axis labels will be displayed, effectively "thinning out" the labels on the x-axis.
- So, if n decreases (resulting in more boxes), it may want to increase k to prevent the x-axis from becoming too crowded with labels.
- Conversely, if n increases (fewer boxes), it could afford to decrease k to show more x-axis labels without overcrowding.

In summary, k needs to increase when n decreases to maintain a legible and useful x-axis in the plot.