Task 3 Report

NACA HITCHMAN - 103072170

Utility Functions

Firstly, as these functions will be plotting original downloaded finance data-frames, and not predicted data like the previous task, some code needs to be moved from the processData function from task 2 into their own separate functions. Specifically, the finance data downloading and file saving, as well as the data NANs processing.

By moving both into their own separate functions, we can re-use them for this task and still also call them in the original processData function. We can see below the data download code was moved into a new downloadData function, where we pass the ticker, dates, and whether or save file Boolean. Further, the NAN processing code was moved into a processNANs function, still retaining the ability to specify if NANs are to be dropped or filled. Both of these functions return dataframes so they can be used in other areas.

```python
def downloadData(ticker, start_date, end_date, save_file=False):
    #create data folder in working directory if it doesnt already exist
    data_dir = os.path.join(os.getcwd(), 'data')
    if not os.path.exists(data_dir):
        os.makedirs(data_dir)
    data = None
    #if ticker is a string, load it from yfinance library
    if isinstance(ticker, str):
        # Check if data file exists based on ticker, start_date and end_date
        file_path = os.path.join(data_dir, f"{ticker}_{start_date}_{end_date}.csv")
        if os.path.exists(file_path):
            # Load data from file
            data = pd.read_csv(file_path)
        else:
            # Download data using yfinance
            data = yf.download(ticker, start=start_date, end=end_date, progress=False)
            # Save data to file if boolean save_file is True
            if save_file:
                data.to_csv(file_path)
    #if passed in ticker is a dataframe, use it directly
    elif isinstance(ticker, pd.DataFrame):
        # already loaded, use it directly
        data = ticker
    else:
        # raise error if ticker is neither a string nor a dataframe
        raise TypeError("ticker can be either a str or a `pd.DataFrame` instances")

    # return the dataframe
    return data

def processNANs(df, fillna_method):
    # Deal with potential NaN values in the data
    # Drop NaN values
    if fillna_method == 'drop':
        df.dropna(inplace=True)
    #use forward fill method, fill NaN values with the previous value
    elif fillna_method == 'ffill':
        df.fillna(method='ffill', inplace=True)
    #use backward fill method, fill NaN values with the next value
    elif fillna_method == 'bfill':
        df.fillna(method='bfill', inplace=True)
    #use mean method, fill NaN values with the mean of the column
    elif fillna_method == 'mean':
        df.fillna(data.mean(), inplace=True)

    return df
```

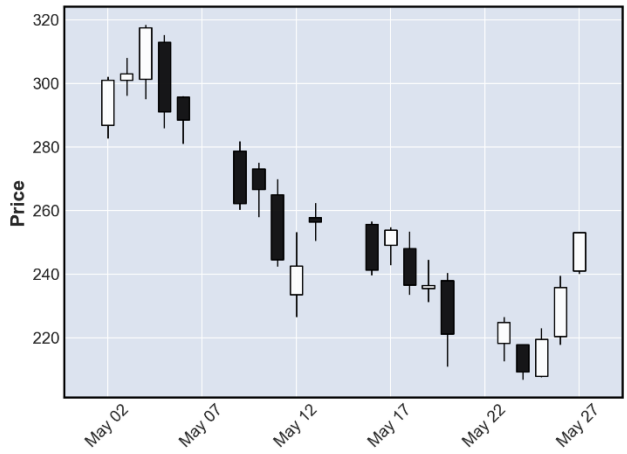
Candlestick Chart Function

Creating the candlestick chart was extremely straightforward. The task specifies we need to be able to pass n as the number of trading days represented by each stick, so for the parameters we pass both a data frame, and an n integer with a default value of 1 if the user chooses not to pass a value.

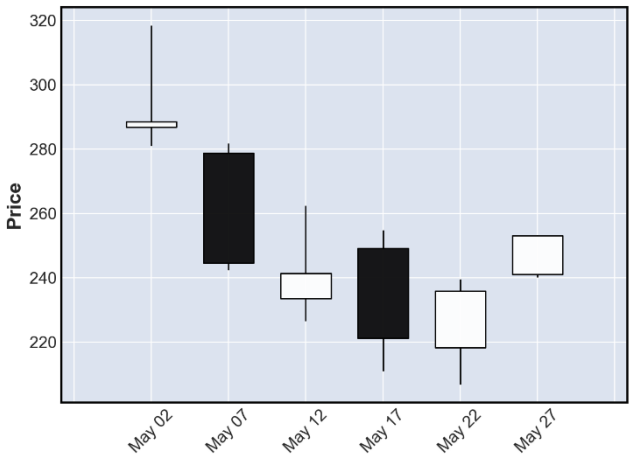
```
def plot_candlestick(df, n=1):
    # Resample the data to have one row per n trading days
    df = df.resample(f'{n}D').agg({'Open': 'first',
                                   'High': 'max',
                                   'Low': 'min',
                                   'Close': 'last'})

    # Create the candlestick chart
    mpf.plot(df, type='candle')
```

The resample method is then run on the dataframe to adjust it to fit the n trading days parameter. “f’{n}D’” specifies to interpres n as days, as agg helps us ensure the various columns are aggegated correctly according to their candlechart usage.vThen, the mpf plot method can be used, passing the ‘candle’ type to show a candlestick chart using the data frame.

n = 1 :



n = 5 :



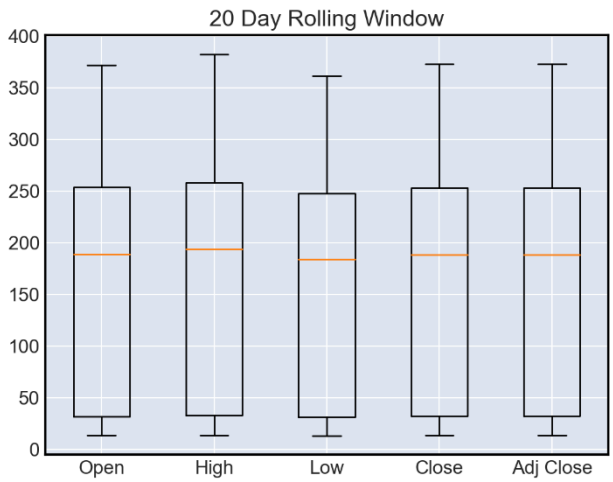
Boxplot Chart Function

Creating the boxplot was also relatively straight forward. The specifications asked for being able to pass n as the size of a rolling window average, so our parameters include a dataframe, n for the window size, and the list of columns to be displayed.

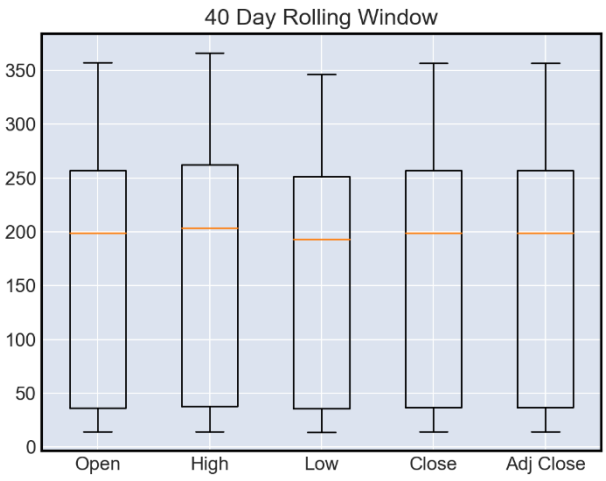
```
def plot_boxplot(df, n, columns):
    # Calculate the rolling window data for each column
    rolling_data = [df[column].rolling(n).mean() for column in columns]

    # Create the box plot
    fig, ax = plt.subplots()
    ax.boxplot([data.dropna() for data in rolling_data], labels=columns)
    ax.set_title(f'{n} Day Rolling Window')

    # Show the plot
    plt.show()
```

Firstly, the rolling data is calculated for the passed data-frame. Each column of the data is loop through, and the built in rolling() method is used, passing n, to get the data we need. The pyplot subplots() method is used to give us an ax object, which then allows us to generate a boxplot, passing a modified rolling_data list with the nans dropped. The title of the plot is then set, before slowing the plot using pyplot again.

n = 20 :



n = 40 :



I found that since the volume column values are significantly larger than the other columns, they can't be displayed without the other columns appearing invisible due to how much smaller they are. This is why I had to include being able to pass which columns to display, so that the other columns could be visible by not passing volume.