COS30018 Intelligent Systems Assignment B

Semester 2, 2023
Assignemnt B Report

Robin Findlay-Marks (103603871) and the Project leader: Mukesh Malani

Self-Assessment Details

The following checklists provide an overview of my self-assessment for this unit.

	Pass (P)	Credit (C)	Distinction (D)	High Distinction (Low HD)	(High HD)
Self-Assessment (please tick)	✓				

Self-assessment Statement

	Included? (tick)
Learning Summary Report	✓
Complete Pass ("core") task work, approved in Canvas	✓

Minimum Pass Checklist

	Included? (tick)
Additional non-core task work (or equivalent) in a private repository and accessible to staff account.	
Spike Extension Report (for spike extensions) in Canvas	
Custom Project plan (for D and/or low HD), and/or High HD Research Plan document in Canvas (optional)	

Credit Checklist, in addition to Pass Checklist

Introduction

This report summarises what I have done for the COS30018 Intelligent Systems Assignment B. For this semester I was given a basic stock prediction file as a starting point (v0.1). Over the following weeks I was given several tasks to incrementally improve the v0.1 code until it is done.

Instructions to run the program

To run the program, first install the requirements listed in the requirements.txt file which are as follows:
sklearn
tensorflow
matplotlib
numpy
pandas
pandas-datareader
yahoo_fin
yfinance
scikit-learn
mplfinance
statsmodels
cython
prophet

These can be install individually with the pip install command, or by running the pip install command on the requirements.txt file.

The next step to run the program is to open the Intellegent-Systems-Assignemnt-B folder in the command prompt.
Finally run the command to run the stock_prediction.py file which is:
py stock_prediction.py

To change the settings of the program, open the parameters.py, which has all the settings and parameters used in the program.

One important setting to note is the 'MODE' parameter. This parameter decides which of several modes to run the program in. They are; 1 – to use the basic RNN models with the data split into train and test by a specified date, 2 - to use the basic RNN models with the data split into train and test by a ratio, 3 – display candlestick chart of the past NDAYS of downloaded data, 4 – display boxplot chart of the past NDAYS of downloaded data, 5 – run the prophet prediction model and any other number, which uses the basic RNN models with the data split into train and test by a random date.

Some additional functionality instructions (assumes parameters are set to default for each instruction)
To test the Multivariate function, set the MODE parameter to 2 and MULTIVARIATE parameter to true
To test the Ensemble with ARIMA function, set the MODE parameter to 2 and ENSEMBLE parameter to true
To test the Ensemble with SARIMA function, set the MODE parameter to 2, ENSEMBLE parameter to true and SARIMA parameter to true
To test the Random Forest function, set the MODE parameter to 2, FOREST parameter to true
To test the hyperparameters, scroll down on the ADVANCED SETTINGS and then Hyperparameters section in parameters.py.
To change the RNN model used, change the hyperparameters's RNN_HYPERPARAM to 1 for LSTM, 2 for SimpleRNN, 3 for GRU

Overall system architecture

The overall program uses just 2 python files. The stock_prediction.py file runs all the functions, while the parameters.py file has all the settings and parameters for the program.

The parameters.py file has various settings such as the ticker symbol, start and end date of the stock data range and how many days into the future to predict are present. Each setting should have a comment to explain what it does and I have already explained what is needed to run the different modes of the program in section of this document detailing how to run the program. As such I will not explain how parameters.py will work.

For the stock_prediction.py file, it starts with the Main() function at the bottom of the file. This function takes the input from the MODE parameter and puts it into a switch to determine what functions and functionality to use.

Main function used:

```python
def Main(): #Main function for deciding what functions to use
    #Make filename for the saved data file
    ticker_data_filename = os.path.join("data", f"{COMPANY}_{TRAIN_START}_{TEST_END}.csv")

    # Switch for checking which mode to run the program in
    match MODE:
        case 1: #Predict with date split
            getDataSplitDate(ticker_data_filename, SPLIT_DATE)
            runTest()

        case 2: #Predict with ratio split
            getDataRatio(ticker_data_filename, RATIO)
            runTest()

        case 3: #Candlestick Chart
            candlestickChart(ticker_data_filename)

        case 4: #Boxplot Chart
            boxplotChart(ticker_data_filename)

        case 5: #Prophet Prediction
            getDataRatio(ticker_data_filename, RATIO)
            runTestProphet()
```

```python
        case _: #Predict with random date split
            #Convert dates to datetime
            dateStart = datetime.strptime(TRAIN_START, '%Y-%m-%d')
            dateEnd = datetime.strptime(TEST_END, '%Y-%m-%d')
            #Get random date inbetween the start and end
            random_date = dateStart + (dateEnd - dateStart) * random.random()
            #convert back to string
            random_date = random_date.strftime('%Y-%m-%d')
            #Use random date as split
            getDataSplitDate(ticker_data_filename, random_date)
            runTest()

Main()
```

Implemented data processing techniques

The program implements several data processing techniques. This starts with the checkFiles function, which can be seen below.

```python
# Function for checking if the data is already downloaded (needs internet connection to download)
# If the data is NOT in a file, it downloads the data and makes a csv file and returns the data
#If the data IS in a file, it reads the data from the file and returns the data
def checkFiles(filename):
    if (os.path.exists(filename)):
        #Read csv file and return the data inside
        data = pd.read_csv(filename)
        # NaN values from pandas are values that are not a number. For example in stocks if the stock data for a
        # specific day was not recorded, i believe it would still have a record for that day, only the values
        # would be NaN or 'Not a Number'

        #what dropna() does is simply remove the missing values from the dataset
        data.dropna(inplace=True)
        return data
    else:
        #Download data from online
        data = yf.download(COMPANY, start=TRAIN_START, end=TEST_END, progress=False)

        # Save data to csv   file
        data.to_csv(filename)
        # For some reason it needs to read it from the file otherwise it won't work
        data = pd.read_csv(filename)

        if (STOREFILE == False):
            os.remove(filename) #Remove stored file
        # remove NaN values from the dataset
        data.dropna(inplace=True)
        return data
```

This function checks if the waned data has already been downloaded. If it has been, it loads the data from the file, otherwise it downloads the data from the internet and then stores it as a file.

The next step is to split the data into training and test datasets, which is run in one of three ways. The first, is to split the data by a specified date, the code can be seen below,

```python
# This function gets the datafile name as well as the split date
# it then runs the file checker to get the dataset, then splits the dataset at the split date
# and sets the trainData and testData
def getDataSplitDate(filename, splitDate):
    df = checkFiles(filename)

    # Make it know that the date colunm is indeed a date
    df['Date'] = pd.to_datetime(df['Date'])
    df = df.set_index(df['Date'])
    df = df.sort_index()

    #Convert input to datetime, add 1 day, then convert back to string
    date = datetime.strptime(splitDate, '%Y-%m-%d')
    testStartDate = date + timedelta(days=1)
    testStartDate = testStartDate.strftime('%Y-%m-%d')

    # Set the fulldata variable
    global fullData
    fullData = df

    # create train/test partition
    global trainData
    trainData = df[TRAIN_START:splitDate]
    trainData = trainData.drop(trainData.columns[[0]], axis=1)
    global testData
    testData = df[testStartDate:TEST_END]
    testData = testData.drop(testData.columns[[0]], axis=1)
    print('Train Dataset:',trainData.shape)
    print('Test Dataset:',testData.shape)
```

the second splits the data by a ratio, the code can be seen below,

```python
# This function gets the datafile name as well as the 'ratio' number
# it then runs the file checker to get the dataset, then splits the dataset at the split date
# and sets the trainData and testData
def getDataRatio(filename, ratio):
    df = checkFiles(filename)

    # Make it know that the date colunm is indeed a date
    df['Date'] = pd.to_datetime(df['Date'])
    df = df.set_index(df['Date'])
    df = df.sort_index()

    # Convert strings to dates
    date1 = datetime.strptime(TRAIN_START, '%Y-%m-%d')
    date2 = datetime.strptime(TEST_END, '%Y-%m-%d')

    # do math to get the date we want
    trainEndDate = date2 + (date1 - date2) / ratio

    #Convert input to datetime, add 1 day
    print("Middle : " + trainEndDate.strftime('%Y-%m-%d'))
    testStartDate = trainEndDate + timedelta(days=1)
    # Convert back to string
    testStartDate = testStartDate.strftime('%Y-%m-%d')
    trainEndDate = trainEndDate.strftime('%Y-%m-%d')

    # Code for testing dates
    print('trainEndDate Dataset:',trainEndDate)
    print('testStartDate:',testStartDate)
```

```python
    # Set the fulldata variable
    global fullData
    fullData = df

    # create train/test partition
    global trainData
    trainData = df[TRAIN_START:trainEndDate]
    trainData = trainData.drop(trainData.columns[[0]], axis=1)
    global testData
    testData = df[testStartDate:TEST_END]
    testData = testData.drop(testData.columns[[0]], axis=1)
```

and the uses a random date to split it, by generating a random date between the start and end date, then inputting that into the split by date function, which can be seen below.

```
        case _: #Predict with random date split
            #Convert dates to datetime
            dateStart = datetime.strptime(TRAIN_START, '%Y-%m-%d')
            dateEnd = datetime.strptime(TEST_END, '%Y-%m-%d')
            #Get random date inbetween the start and end
            random_date = dateStart + (dateEnd - dateStart) * random.random()
            #convert back to string
            random_date = random_date.strftime('%Y-%m-%d')
            #Use random date as split
            getDataSplitDate(ticker_data_filename, random_date)
            runTest()
```

Another data processing method that is present is to display the downloaded data as a candlestick chart (code below)

```
def candlestickChart(filename):
    # Get data
    df = checkFiles(filename)

    # Make it know that the date colunm is indeed a date
    df['Date'] = pd.to_datetime(df['Date'])
    # Set the index of the dataframe to be the date colunm
    df = df.set_index(df['Date'])
    df = df.sort_index()

    # Get the last n days
    actual_prices_small = df[-NDAYS:]

    # Plot the candlestick chart
    mpl.plot(actual_prices_small.set_index("Date"), type="candle", style="charles", title='Candlestick Chart')
    # Uses mplfinance  to make the candlestick chart
    # uses the date colunm as index
    # uses the 'charles' stle to make the decreasing days red and increasing days green
```

and a boxplot chart (code below).

```
def boxplotChart(filename):
    # Get data
    df = checkFiles(filename)
    # Get the last n days
    actual_prices_small = df[-NDAYS:]

    # Plot the boxplot chart
    fig = actual_prices_small[['Open', 'Close', 'Low', 'High']].plot(kind='box', title='Boxplot Chart', grid=True)
    # Uses Matplotlib  to make the boxplot chart
    # uses grid=True to make a grid so the chart is easier to read

    # Add axis labels and display the plot
    plt.xlabel("Stock Data Type")
    plt.ylabel("Price")
    plt.show()
```

I believe that these are all the main data processing techniques used in the program.

Implemented machine learning techniques

The program implements several machine learning techniques. The first of these is a function to function with several inputs and return a deep learning model. The input parameters are the number of layers, the size of each layer and the RNN model subtype. The code for this function can be seen below.

```
def createModelRNN(layer_num, layer_size, layer_name, dropout):
    #Declare some variables so the model knows whats what
    PRICE_VALUE = "Close"

    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(trainData[PRICE_VALUE].values.reshape(-1, 1))

    # To store the training data
    x_train = []
    y_train = []

    scaled_data = scaled_data[:,0] # Turn the 2D array back to a 1D array
    # Prepare the data
    for x in range(LOOKBACK_DAYS, len(scaled_data)):
        x_train.append(scaled_data[x-LOOKBACK_DAYS:x])
        y_train.append(scaled_data[x])

    # Convert them into an array
    x_train, y_train = np.array(x_train), np.array(y_train)
    # Now, x_train is a 2D array(p,q) where p = len(scaled_data) - LOOKBACK_DAYS
    # and q = LOOKBACK_DAYS; while y_train is a 1D array(p)
    x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
    # We now reshape x_train into a 3D array(p, q, 1); Note that x_train
    # is an array of p inputs with each input being a 2D array
```

```python

    model = Sequential() # Basic neural network
    #Add layers to network using for each loop, which takes the layer_num to determine how many layers are added
    for i in range(layer_num):
        if i == 0:
            # first layer
            model.add(layer_name(layer_size, return_sequences=True, input_shape=(x_train.shape[1], 1)))
        elif i == layer_num - 1:
            # last layer
            model.add(layer_name(layer_size, return_sequences=False))
        else:
            # hidden layers
            model.add(layer_name(layer_size, return_sequences=True))
        # add dropout after each layer
        model.add(Dropout(dropout))

    # Prediction of the next closing value of the stock price
    model.add(Dense(units=1))
    # Compile the model
    model.compile(optimizer='adam', loss='mean_squared_error')
    # Now we are going to train this model with our training data
    # (x_train, y_train)
    model.fit(x_train, y_train, epochs=25, batch_size=32)

    # Return completed model to be tested
    return model
```

The input variables that are used for this function are modified from the parameters.py, with various hyperparameter presets in the file. They can be viewed below.

```python
# Default Parameters

#RNN param
LAYER_NUM = 2        # number of layers used
LAYER_SIZE = 50      # size of each layer
LAYER_NAME = LSTM    # which type of RNN model is used
DROPOUT = 0.2        # dropout
```

```python
RNN_HYPERPARAM = 1
ARIMA_HYPERPARAM = 1
SARIMA_HYPERPARAM = 1

match RNN_HYPERPARAM:
    case 1: #LSTM
        LAYER_NUM = 2
        LAYER_SIZE = 50
        LAYER_NAME = LSTM
        DROPOUT = 0.2
    case 2: # RNN
        LAYER_NUM = 2
        LAYER_SIZE = 50
        LAYER_NAME = SimpleRNN
        DROPOUT = 0.2
    case 3: #GRU
        LAYER_NUM = 2
        LAYER_SIZE = 50
        LAYER_NAME = GRU
        DROPOUT = 0.2
    case 4: #P1 settings
        LAYER_NUM = 2
        LAYER_SIZE = 256
        LAYER_NAME = LSTM
        DROPOUT = 0.4
    case 5: #Custom
        LAYER_NUM = 2
        LAYER_SIZE = 50
        LAYER_NAME = GRU
        DROPOUT = 0.1
    case _: #Default settings
        LAYER_NUM = 2
        LAYER_SIZE = 50
        LAYER_NAME = SimpleRNN
        DROPOUT = 0.2
```

The next machine learning technique used is the multistep prediction. This means that the program must be able to predict an amount of days into the future, determined by an input. For this I made the RNN model predict a day in the future, then add that to the model's input data, which is then used to make a prediction another day into the future. This process repeats, until the input (PREDICTION_DAYS) days into the future is predicted. The code for this part of the function is below.

```python
    futurePrice = []


    i = 0

    # for each day in the future to predict
    while i < PREDICTION_DAYS:
        i += 1

        # make it so it does (or redoes) the declaring of real_data based on the last PREDICTION_DAYS amount of days
        real_data = [model_inputs[len(model_inputs) - PREDICTION_DAYS:, 0]]
        real_data = np.array(real_data)
        real_data = np.reshape(real_data, (real_data.shape[0], real_data.shape[1], 1))

        # make prediction of next day
        prediction = model.predict(real_data)
        # unscale and flatten prediction data
        scaledPrdiction = scaler.inverse_transform(prediction)
        # add predicted data to future price
        futurePrice.append(scaledPrdiction.flatten()[0])
        # set prediction to be the flattened but still scaled data
        prediction = prediction.flatten()[0]

        print(futurePrice)

        # set model inputs to be dataframe, add predicted data to it, then make it a numpy array again
        model_inputs = pd.DataFrame(model_inputs)
        model_inputs.loc['0'] = prediction
        model_inputs = model_inputs.to_numpy()
```

The next machine learning technique used is the multivariate prediction. This means that the program must use the data other than just the closing data as inputs to predict the closing price.

The code for this can be viewed below

```
def multivariate_prediction(layer_num, layer_size, layer_name):
    # make a copy of the train and test dataframes
    train_df = trainData.sort_values(by=['Date']).copy()
    test_df = testData.sort_values(by=['Date']).copy()
    # Add dummy column and set dummy values for sacling in the future
    train_df_ext = train_df.copy()
    train_df_ext['Dummy'] = train_df_ext['Close']
    test_df_ext = test_df.copy()
    test_df_ext['Dummy'] = test_df_ext['Close']
    # Get the number of rows in the data
    nrows = train_df.shape[0]
    # Convert the data to numpy values
    np_train_unscaled = np.array(train_df)
    np_test_unscaled = np.array(test_df)
    np_data = np.reshape(np_train_unscaled, (nrows, -1))
    # Transform the data by scaling each feature to a range between 0 and 1
    scaler = MinMaxScaler()
    np_train_scaled = scaler.fit_transform(np_train_unscaled)
    np_test_scaled = scaler.fit_transform(np_test_unscaled)
    # Creating a separate scaler that works on a single column for scaling predictions
    scaler_pred = MinMaxScaler()
    df_Close = pd.DataFrame(train_df_ext['Close'])
    df_Close2 = pd.DataFrame(test_df_ext['Close'])
    np_Close_scaled = scaler_pred.fit_transform(df_Close)
    np_Close_scaled2 = scaler_pred.fit_transform(df_Close2)
    # Set Prediction Index
    index_Close = train_df.columns.get_loc("Close")
    # Create the training and test data
    train_data = np_train_scaled
    test_data = np_test_scaled
    # Here, we create N samples, LOOKBACK_DAYS time steps per sample, and 6 features

    def partition_dataset(LOOKBACK_DAYS, data):
        x, y = [], []
        data_len = data.shape[0]
        for i in range(LOOKBACK_DAYS, data_len):
            x.append(data[i-LOOKBACK_DAYS:i,:]) #contains LOOKBACK_DAYS values 0-LOOKBACK_DAYS * columns
            y.append(data[i, index_Close]) #contains the prediction values for validation,  for single-step prediction
        # Convert x and y to numpy arrays
        x = np.array(x)
        y = np.array(y)
        return x, y
    # Generate training data and test data
    x_train, y_train = partition_dataset(LOOKBACK_DAYS, train_data)
    x_test, y_test = partition_dataset(LOOKBACK_DAYS, test_data)

    # Configure the neural network model
    model = Sequential()

    #Add layers to network using for each loop, which takes the layer_num to determine how many layers are added
    for i in range(layer_num):
        if i == 0:
            # first layer
            model.add(layer_name(layer_size, return_sequences=True, input_shape=(x_train.shape[1], x_train.shape[2])))
        elif i == layer_num - 1:
            # last layer
            model.add(layer_name(layer_size, return_sequences=False))
        else:
            # hidden layers
            model.add(layer_name(layer_size, return_sequences=True))

    # Prediction of the next closing value of the stock price
    model.add(Dense(1))
    # Compile the model
    model.compile(optimizer='adam', loss='mean_squared_error')
    # Training the model
    early_stop = EarlyStopping(monitor='loss', patience=5, verbose=1)
    history = model.fit(x_train, y_train, batch_size=32, epochs=10, validation_data=(x_test, y_test))
    # Get the predicted values
    y_pred_scaled = model.predict(x_test)
    # Unscale the predicted values
    y_pred = scaler_pred.inverse_transform(y_pred_scaled)
    y_test_unscaled = scaler_pred.inverse_transform(y_test.reshape(-1, 1))

    return y_pred
```

The final machine learning technique used that is not part of the extension is an ensemble model of the RNN model and a ARIMA or SARIMA model. An option to use ARIMA or SARIMA is present. The ensemble approach I took, was to get the values for each model and get the average of them to get the ensemble value.

The S/ARIMA code can be viewed below

```
# Function for running ARIMA or SARIMA predictions
def ARIMA_prediction():
    # assign train and test data to variables
    train = trainData['Close'].values
    test1 = testData[1:]
    test = test1['Close'].values
    history = [x for x in train]
    predictions = list()
    # parameters for SARIMA
    my_seasonal_order = (SAUTOREG, SDIFERENCE, SMOVAVG, SEASON)

    # walk-forward validation
    for t in range(len(test)):
        # re-create the ARIMA model after each new observation
        if SARIMA:
            model = ARIMA(history, order=(AUTOREG,DIFERENCE,MOVAVG), seasonal_order=my_seasonal_order)
        else:
            model = ARIMA(history, order=(AUTOREG,DIFERENCE,MOVAVG))
        model_fit = model.fit()
        # make prediction
        output = model_fit.forecast()
        forecast = output[0]
        predictions.append(forecast)
        expected = test[t]
        # keep track of past observations
        history.append(expected)
        print('predicted=%f, expected=%f' % (forecast, expected))
    return predictions
```

The ensembling of the data code can be viewed below

```
    # combine the RNN prediction value and the S/ARIMA prediction value and average them to make an ensemble value
    ensemble_preds = None
    if ENSEMBLE:
        predicted_prices_list = predicted_prices.tolist()
        i = 0
        # for each predicted value from each mode, add the two together and to make the ensemble value
        for value in arima_pred:
            # first parameter is the array that the second parameter is being added to
            ensemble_preds = np.append(ensemble_preds, (arima_pred[i] + predicted_prices_list[i])/2)
            i += 1
```

Details of extension work

For the extension work I decided to add two new machine learning techniques. The first of these is using facebook's prophet to make predictions. The code for this is fairly simple concise, being only one function, however due to it's unique data input method, it has it's own separate MODE from the rest of the code.

The code for this can be viewed below

```python
def runTestProphet():
    #Get pre-split data
    # Only need to extract Close because the date is the index, and as such is automatically transfered over too
    test = fullData['Close']
    # reset index because it turns it back into a normal colunm which is referenced later
    test = test.reset_index()
    # turn date and close colunm into ds and y (necessary for prophet to work)
    test.columns = ['ds', 'y']
    # make sure ds colunm is a datatime
    test['ds']= pd.to_datetime(test['ds'])
    # remove offset days from the training data
    train = test.drop(test.index[-PROPHET_TRAIN_OFFSET:])

    # make the prophet model
    model = Prophet()
    # train the model from the train data
    model.fit(train)
    # setup dataframe from only the date date
    futuredays = list()
    futuredays = test['ds']
    futuredays = pd.DataFrame(futuredays)

    # use the model to make a forecast from the futuredays datetime range
    forecast = model.predict(futuredays)
    # set the actual and predicted values
    actualData = test['y'][-len(forecast):].values
    prophetPrediction = forecast['yhat'].values
    # plot actual vs Predicted Data
    plt.plot(actualData, label='Actual')
    plt.plot(prophetPrediction, label='Predicted')
    plt.legend()
    plt.show()
```

The other machine learning technique I added was random forest. This was more complicated having a main function, and three sub functions. The code for this can be viewed below.

```python
def runTestForest():
    # get test + train data and make local variables
    global testData
    global trainData
    train = trainData["Close"].values
    test = testData["Close"].values

    # turn train and test data into supervised learning
    train = data_to_supervised(train)
    test = data_to_supervised(test)

    # make predictions
    forestPrediction = forest_validation(test, train)

    return forestPrediction
```

```python
    # turn time series data into a supervised learning data
def data_to_supervised(data):
    df = pd.DataFrame(data)
    colunms = list()

    # sliding window technique used to make the new samples for the supervised learning data
    # input sequence (t-n, ... t-1)
    for i in range(1, 0, -1):
        colunms.append(df.shift(i))
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, 1):
        colunms.append(df.shift(-i))
    # put it all together
    newdf = concat(colunms, axis=1)
    # drop NaN values
    newdf.dropna(inplace=True)
    return newdf.values
```

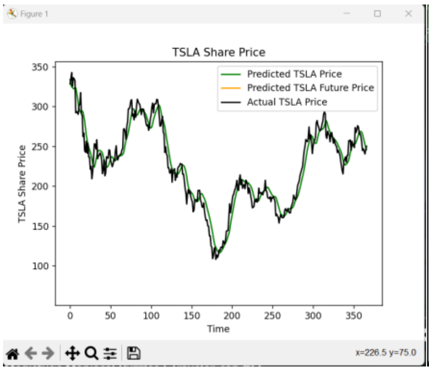
```python
    # loop for running the prediction on a number of days
def forest_loop(test, train):
    predictions = list()
    # seed history with training dataset
    history = [x for x in train]
    # step over each time-step in the test set
    for i in range(len(test)):
        # split test row into input and output columns
        testX = test[i, :-1]
        # fit model on history and make a prediction
        forestPrediction = forest_prediction(history, testX)
        # store forecast in list of predictions
        predictions.append(forestPrediction)
    return predictions
```

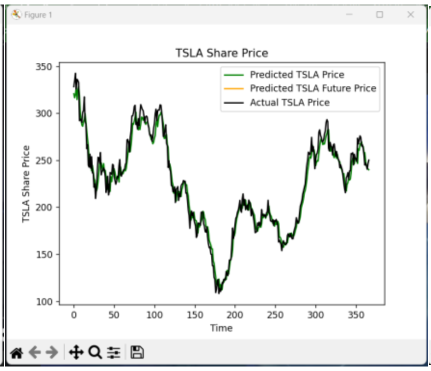
```python
    # fit an random forest model and make a one step prediction
def forest_prediction(train, testX):
    # turn list into an array
    train = asarray(train)
    # split into input and output columns
    trainX, trainy = train[:, :-1], train[:, -1]
    # make model
    model = RandomForestRegressor(n_estimators=FOREST_ESTIMATORS)
    model.fit(trainX, trainy)
    # make a single prediction
    forestPrediction = model.predict([testX])
    return forestPrediction[0]
```

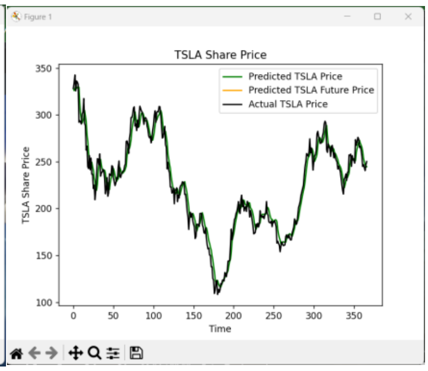
Due to the small page limit for this document, please refer to the task 7 report for more info on how these techniques were implemented

Demonstration of working system

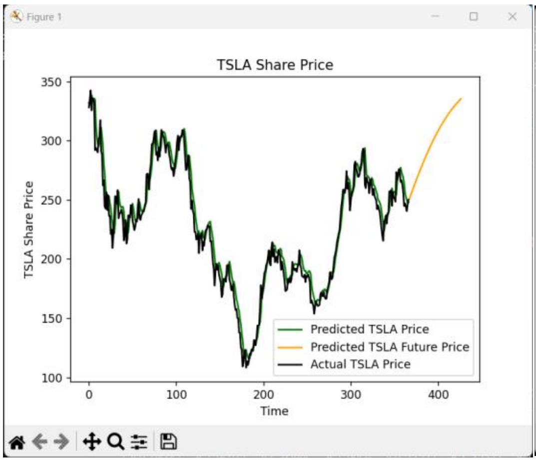
RNN model predictions with LSTM, then SimpleRNN, then GRU

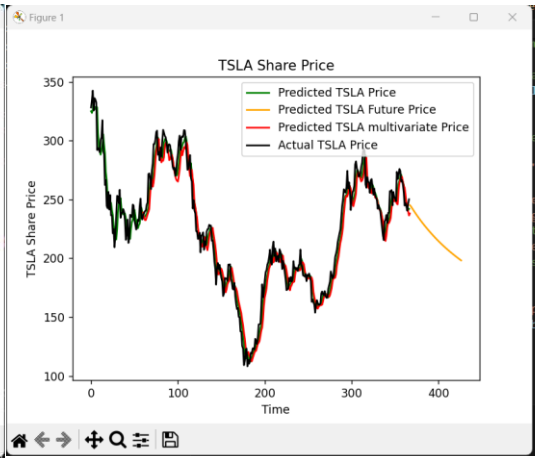




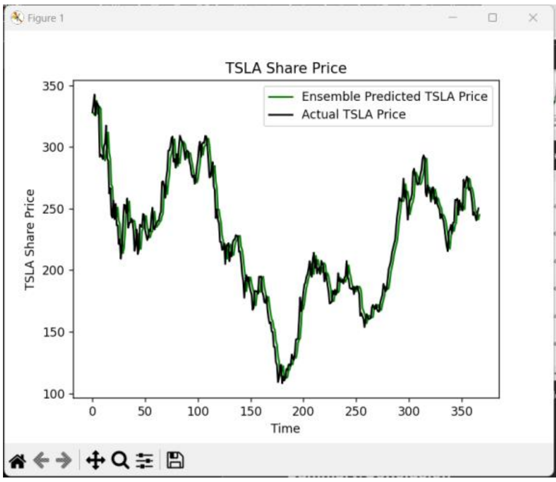


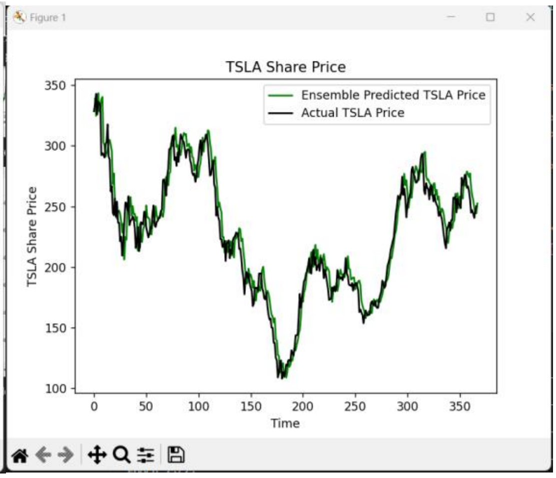
First image is 60 days in the future prediction (multistep), then second is Multivariate prediction



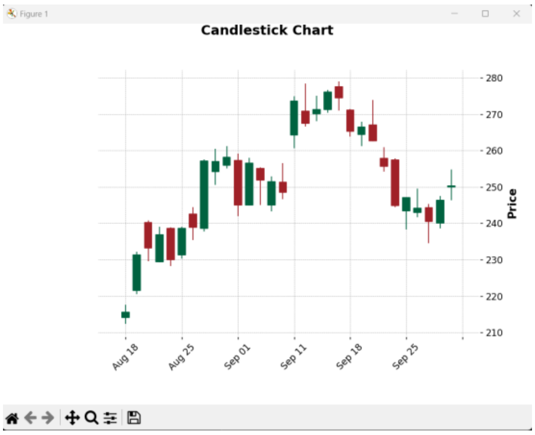


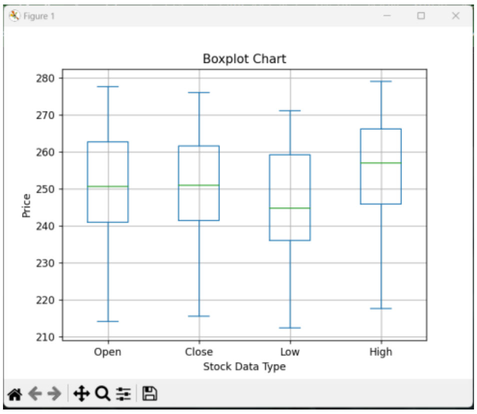
First image is Arima ensemble predictions, then Sarima ensemble predictions



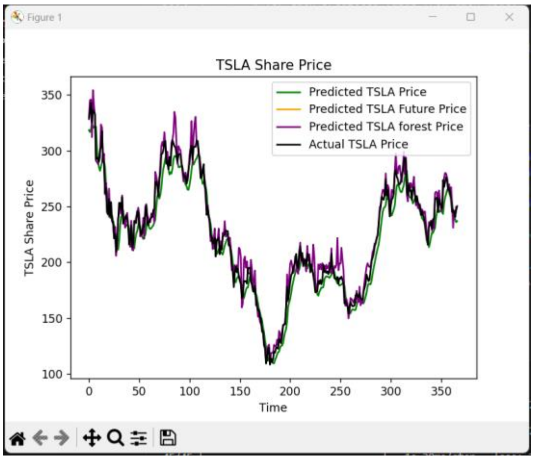


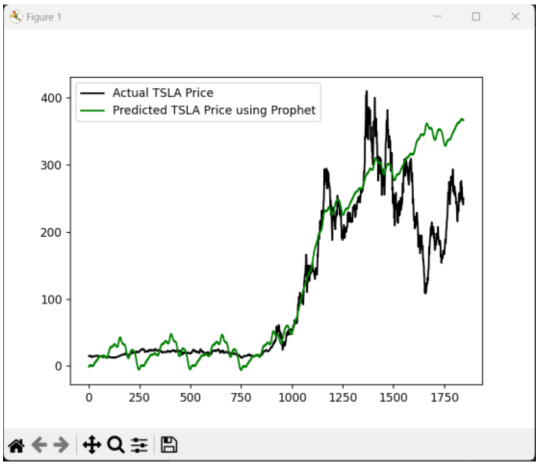
First image is candlestick chart, then boxplot chart





First image is random forest predictions, then the prophet predictions





Reflection on work done

Overall I am happy with the work I did for this assignment. I felt like I learnt a lot about the basics for how to make and run several different machine learning models and what each model is useful for. If in the future I use machine learning models I will have already working code that I made that I can use as a launching pad to get the work done.

I would have liked to be able to get the prophet predictions working so they make more than vaguely similar predictions to the actual data. I am happy with how the random forest code runs, although it clearly needs some tweaking to get that working as well as some of the other models.

My only worry is that I completed this report in the way that was expected of me. I am unsure if I was meant to include this many screenshots of the work I did or if it was meant to be filled with explanations of how the code works. I concluded that it should be screenshots and a summary since reports 1-7 contained more detailed explanations of the work done. Please refer to these reports if you wish for further explanations.

Summary/Conclusion

For this assignment B, I spent a lot of hours working to get all the features of the program working and I believe I succeeded. In addition to this, I also added some extra machine learning functionality in the form of the simple extension.

URL to github:

https://github.com/R0binicus/Intellegent-Systems-Assignemnt-B