Part A:

Predict the price of Brent Oil using Linear Regression:

Linear Regression:

Linear regression tries to find a linear relationship between two or more variables. Most of the common machine learning algorithms are Linear regression (Maulud & Abdulazeez, 2020). Here, we use the independent variable y to predict the dependent variable x. There are 2 types of linear regression models: Simple linear regression model with just 1 independent variable. And multiple linear regression model with multiple independent (or explanatory) variables (Seber & Lee, 2003).

We start by importing libraries using the code shown below.

```
[1]  #Loading Libraries
     import math
     import numpy as np
     import pandas as pd
     import matplotlib.dates as mdates
     from matplotlib import pyplot as plt
     from pylab import rcParams
     from sklearn.linear_model import LinearRegression
     from sklearn.model_selection import train_test_split
```

Fig 1. Importing libraries

Then we upload our dataset 'BrentOilPrices.csv' using the code show below.

```
[2]  #Uploading dataset
     df = pd.read_csv('BrentOilPrices.csv', sep = ",")
     df.head()
```

	Date	Price
0	May 20, 1987	18.63
1	May 21, 1987	18.45
2	May 22, 1987	18.55
3	May 25, 1987	18.60
4	May 26, 1987	18.63

Fig 2. Uploading dataset

We also print the dataset to observe the contents of the dataset. We see that it has 8216 rows and 2 columns.

```
[29] print(df)
```

```
                Date  Price
0     May 20, 1987  18.63
1     May 21, 1987  18.45
2     May 22, 1987  18.55
3     May 25, 1987  18.60
4     May 26, 1987  18.63
...            ...    ...
8211  Sep 24, 2019  64.13
8212  Sep 25, 2019  62.41
8213  Sep 26, 2019  62.08
8214  Sep 27, 2019  62.48
8215  Sep 30, 2019  60.99

[8216 rows x 2 columns]
```

Fig 3. Displaying dataset

Since we only need data from the past 20 years, we run the below code. We also print the newly altered dataset. We can see that it now has 5076 rows and 2 columns in the required time frame.

```
[6] #Deleting unrequired data and only keeping data from the last 20 years
    df['date'] = pd.to_datetime(df['date'])

    df = df[~(df['date'] < '1999-10-05')]

    print(df)
```

```
           date  price
3140 1999-10-05  22.64
3141 1999-10-06  23.07
3142 1999-10-07  22.33
3143 1999-10-08  20.78
3144 1999-10-11  20.84
...         ...    ...
8211 2019-09-24  64.13
8212 2019-09-25  62.41
8213 2019-09-26  62.08
8214 2019-09-27  62.48
8215 2019-09-30  60.99

[5076 rows x 2 columns]
```

Fig 4. Displaying dataset after dropping values before the past 20 years

a) Data Visualisation: Define simple line chart to give an idea of the stock price change Brent oil price information for the last 20 years

```
[6] #Line chart for the last 20 years
    rcParams['figure.figsize'] = 13, 8 #width 13, height 8
    ax = df.plot(x='date', y='price')
    ax.legend(['Price'])
    ax.set_xlabel("Date")
    ax.set_ylabel("Price")
```

```
Text(0, 0.5, 'Price')
```

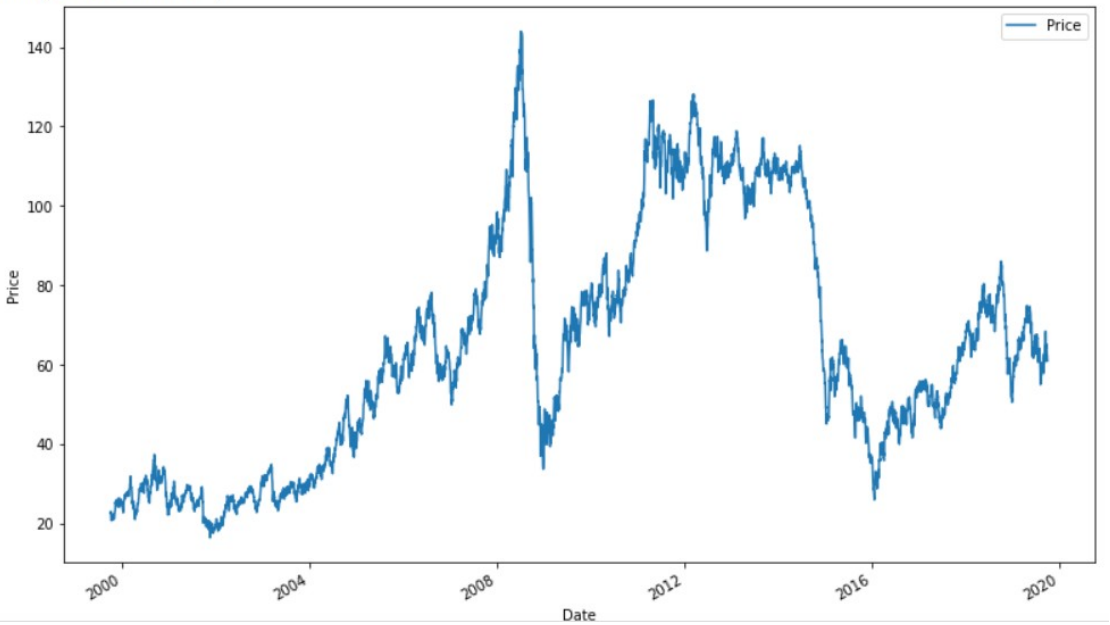

Fig 5. Displaying the dataset for the past 20 years in the form of a line chart

b) Building explanatory variables: The features we are using to predict the price of oil. The variables used here would be moving average 3 days (MA 3) and moving average 9 days (MA 9):

Below we are creating the explanatory variables of moving average 3 and moving average 9 and adding them to the dataset under the column name on 'MA_3' and 'MA_9'.

```
[7] #Creating Moving averages
    df['MA_3']=df.price.rolling(3).mean()
    df['MA_9']=df.price.rolling(9).mean()
```

Fig 6. Creating moving averages.

Plotting the moving averages with the price:

```
[8]  #Plotting moving averages on price chart
     fig, ax = plt.subplots(figsize=(13,4))

     ax.plot(df['price'], label='Price',color='yellow')

     ax.plot(df['MA_9'], label='9-days MA',color='red')
     ax.plot(df['MA_3'], label='3-days MA',color='black')
     plt.title('Stock closing price and moving average')

     ax.legend(loc='upper left')
     ax.set_ylabel('Price')
     ax.set_xlabel("date")
     #ax.xaxis.set_major_formatter(my_year_month_fmt)
```

```
Text(0.5, 0, 'date')
```

Fig 7. Plotting the moving averages with the price

The moving average values have now been added to the dataset and are displayed below. We can see it now has 5076 rows and 4 columns.

```
[9]  #Updated dataset with moving averages
     df
```

	date	price	MA_3	MA_9
3140	1999-10-05	22.64	NaN	NaN
3141	1999-10-06	23.07	NaN	NaN
3142	1999-10-07	22.33	22.680000	NaN
3143	1999-10-08	20.78	22.060000	NaN
3144	1999-10-11	20.84	21.316667	NaN
...	...	...	...	...
8211	2019-09-24	64.13	64.673333	64.286667
8212	2019-09-25	62.41	63.733333	64.470000
8213	2019-09-26	62.08	62.873333	64.562222
8214	2019-09-27	62.48	62.323333	63.902222
8215	2019-09-30	60.99	61.850000	63.391111

5076 rows × 4 columns

As we can see in the displayed dataset, it has null values because of the moving averages. Linear regression doesn't accept null values. The widely accepted method of dealing with this issue is dropping these null values (Shumeiko & Rozora). We used the dropna() function to achieve this as shown below.

```
[10] #Dropping Null values
     df = pd.DataFrame(df).dropna()
     df
```

	date	price	MA_3	MA_9
3148	1999-10-15	21.65	22.063333	21.971111
3149	1999-10-18	22.26	22.063333	21.928889
3150	1999-10-19	21.59	21.833333	21.764444
3151	1999-10-20	21.02	21.623333	21.618889
3152	1999-10-21	21.68	21.430000	21.718889
...	...	...	...	...
8211	2019-09-24	64.13	64.673333	64.286667
8212	2019-09-25	62.41	63.733333	64.470000
8213	2019-09-26	62.08	62.873333	64.562222
8214	2019-09-27	62.48	62.323333	63.902222
8215	2019-09-30	60.99	61.850000	63.391111

5068 rows × 4 columns

Fig 8. Displaying dataset after dropping the rows with NaN(null) values

From the above screenshot, we can see that there are now no null values and hence, the number of rows has been reduced from 5076 to 5068. This dataset is now ready to be tested for linear regression.

c) Preparation of train and test data:

Below we start by first defining both the moving averages as explanatory variables and the 'price' as the dependent variable that has to be predicted.

```
[38] #Defining both the moving averages as explanatory variables and 'price' as the dependent variable
     X = df[['MA_3','MA_9']].values
     y = df['price'].values
```

Fig 9. Defining explanatory variables and dependent variables

Splitting the dataset: This code involves the splitting of the dataset into training and test data. There are two methods of doing this and they are shown below:

Method 1: In this method we use the train_test_split from sklearn. This will randomly split dataset into training and test data. However, since it is random, it is not ideal for reproducibility (Julian, 2022). You need to either provide the amount of training data or the test data. If the data is a float, the value needs to be between 0.1 to 1.0. If the value provided is an int(integer), it will show the total number of training/test data (Real Python, 2021). If neither the training nor the test data is specified, it will assume the training dataset to be 25% by default.

```
[39] #Splitting dataset in train and test data
     X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0, train_size = .75)
```

Fig 10. Splitting dataset in train and test data using sklearn train_test_split

Method 2: The other method shown below uses the length of the dataset to split it into training and test data. The below code splits the data as 80% of dataset as training data. The remaining 20% of the dataset is considered to be test data.

```
[8]  # Setting the training set to 80% of the data
     training = 0.8
     t = int(training*len(df))

     # Training dataset
     X_train = X[:t]
     y_train = y[:t]

     # Testing dataset
     X_test = X[t:]
     y_test = y[t:]
```

Fig 11. Splitting dataset in train and test data using length of dataset

d) Building a linear regression model:

Passing the training dataset through the linear regression model. The X_train contains the explanatory variables and Y_train contains the dependent variables. We use the code shown below to achieve this.

```
[13] model = LinearRegression()

[14] model.fit(X_train, y_train)

     LinearRegression()

[15] y_pred=model.predict(X_test)
```

Fig 12. Building a linear regression model

e) Prediction function and result: Visualising the predicted values versus the actual stock values from the test dataset.

Below we plot the predicted values of the dependent variable (y_pred) against the actual values of the dependent variable (y_test) using a scatterplot. The code and chart are given below.

```
[29] #Plotting predicted value vs actual value of the dependent variable using a scatterplot
     plt.scatter(my_pd['y_test'], my_pd['y_pred'], color='pink')
     plt.plot(range(100), range(100))

     plt.xlabel("Date")
     plt.ylabel("Predicted prices")
     plt.title("Actual  vs Predicted ")

     plt.show()
```

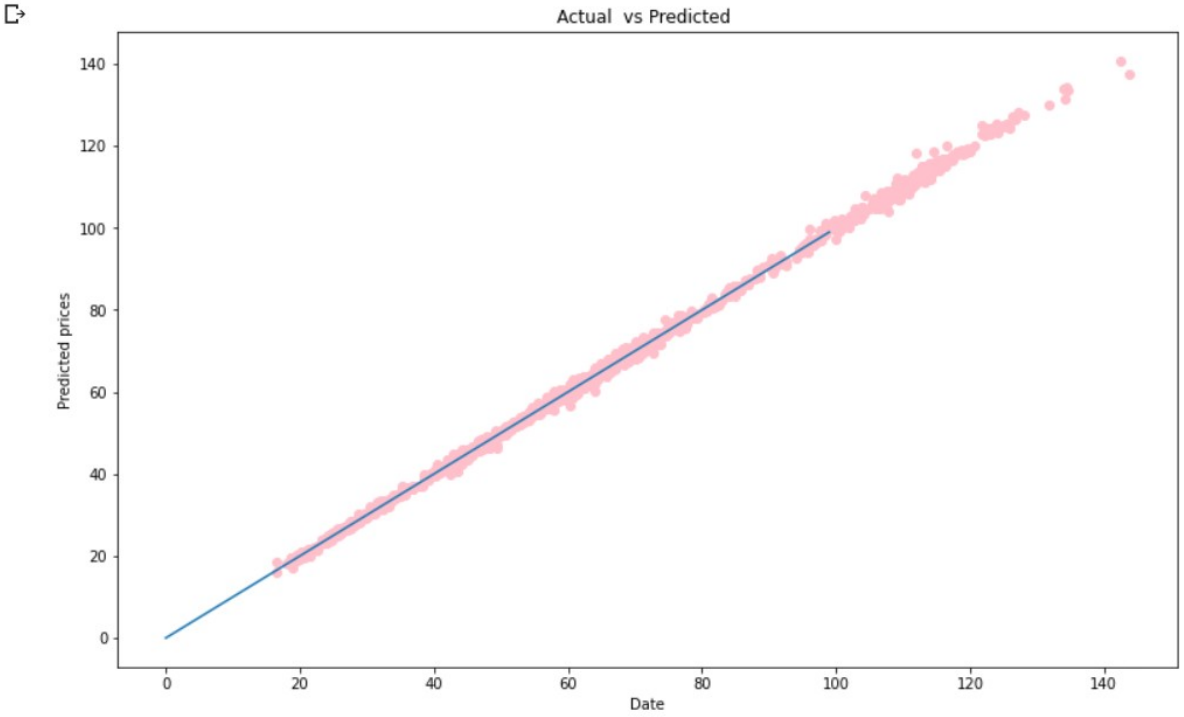

Fig 13. Plotting the actual versus the predicted value as a scatter plot

In the above figure, the pink dots represents the value derived from the linear regression. It is also known as the predicted data. The blue line is the actual data point from the test dataset. The clustering of the pink data points around the blue line shows a very close prediction to the actual value.

We now need to do assessment metrics to understand our model's correctness and to see how well it worked with the given data. By changing the hyperparameters we can improve our model's performance.

Below is the code to evaluate our model's performance:

```
[25] #Model Evaluation
     from sklearn import metrics
     meanAbErr = metrics.mean_absolute_error(y_test, y_pred_model)
     meanSqErr = metrics.mean_squared_error(y_test, y_pred_model)
     rootMeanSqErr = np.sqrt(metrics.mean_squared_error(y_test, y_pred_model))
     print('R squared: {:.2f}'.format(model.score(X,y)*100))
     print('Mean Absolute Error:', meanAbErr)
     print('Mean Square Error:', meanSqErr)
     print('Root Mean Square Error:', rootMeanSqErr)
```

```
R squared: 99.91
Mean Absolute Error: 0.656059707184277
Mean Square Error: 0.8572712769351613
Root Mean Square Error: 0.9258894517895542
```

Fig 14. Evaluating the model's performance

In the above given fig 14, we see that R squared is given as 99.91(which is 0.99). An R squared value close to 1 shows that the model has a high accuracy and has performed well.

f) Calculate the alpha and betas value: Define the linear regression equation using the alpha and betas values

For any given linear equation, the coefficients of the model and the slope of the model have to be calculated. These values are unique.
The linear equation is given by y=Bo + B1x, where Bo is the point at which the line intercepts the y-axis, B1 is the slope of the line, x is the independent variable and y is the dependent variable.

The code for it is given below:

```
[16] print('Intercept: \n', model.intercept_)
     print('Coefficients: \n', model.coef_)
```

```
Intercept:
 0.042238025044120775
Coefficients:
 [ 1.22442904 -0.2250679 ]
```

Fig 15. Printing the intercepts and coefficients of the model with the given dataset

Linear Regression Equation: Here, from the above output, we deduce that Alpha is -0.2250679 and Beta is 1.22442904. In the linear equation Beta is the slope and Alpha is the coefficient.

Conclusion: Thus, considering the fact that we got an R squared value of 99.91, we can safely say that our model performed well with this dataset. We can improve this performance by changing some hyperparameters. Eg. We took 20 years of data into consideration for this model. We can change that by additionally taking a couple of extra years before those taken currently and see the change in the model's accuracy.

Predicting the Brent oil price Stock with LSTM Neural Networks:

LSTM (Long Short Term Memory Network): It is a sophisticated RNN that permits the storage of information. Its wide success and large-scale use by Google, Amazon and various other organisations for speech recognition and translation is credited to its ability to handle the vanishing/exploding gradient problem that the usual RNNs face (Van Houdt et al., 2020).

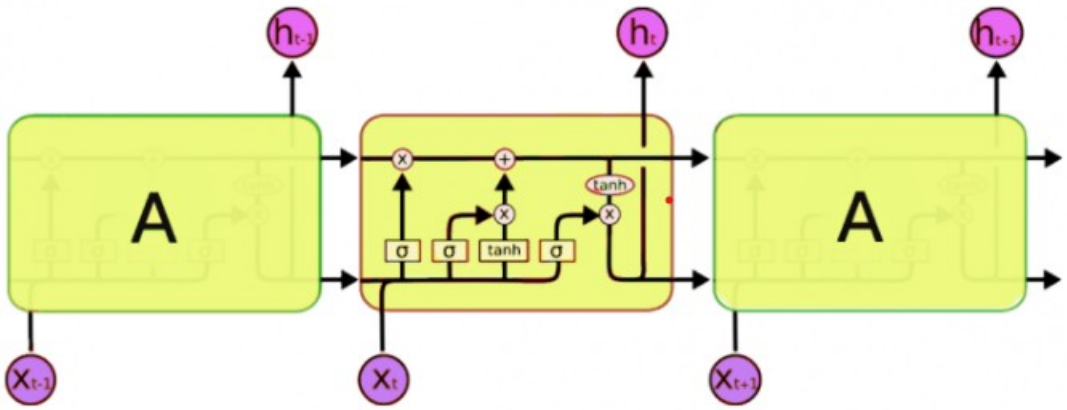

Fig 16. LSTM Architecture

A very simplified explanation of the LSTM architecture can be depicted by its three gates. Namely, the forget gate, the input gate and the output gate. The forget gate as the name suggests is responsible for deciding which of the information isn't important and can be removed by multiplying a filter. It helps in optimising the performance of the LSTM network. The input gate is responsible for adding of information to the cell state. The output gate does the job of selecting important information from the current cell state and showing it as the output (Srivastava, 2020).

Task B:

a) **Define the Train and Test Data: This step covers the preparation of the train data and the test data. Explain the techniques used to generate the train data and the test data for the given Brent oil price time series data set.**

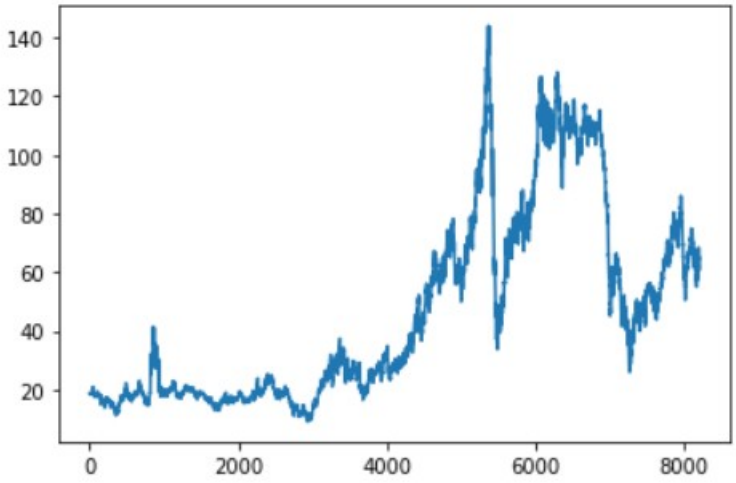

Fig 17. Line chart showing oil prices

```
[2]  dataset
```

	Price
0	18.63
1	18.45
2	18.55
3	18.60
4	18.63
...	...
8211	64.13
8212	62.41
8213	62.08
8214	62.48
8215	60.99

8216 rows × 1 columns

Fig. 18 Displaying the dataset

As we can see in Fig. 18, the dataset has 8216 rows and 1 column. We also display the data as a line chart in Fig. 17. We then import the various libraries to run the model.

We then use the random seed function. Random seed function is fixed at 7 to maintain reproducibility of code whenever it is run again. This is because numpy creates new random values whenever the number isn't given, and the code is run.

```
[4]  # fix random seed for reproducibility
     np.random.seed(7)
```

Fig. 19 random.seed function to ensure reproducibility

```
[6]  # normalize the dataset
     scaler = MinMaxScaler(feature_range=(0, 1))
     dataset1 = scaler.fit_transform(dataset1)
```

Fig. 20 Normalizing the data

The normalizing in Fig. 20 is done to convert all the numeric values in the Price column to a similar scale between 0 to 1 while maintaining as much data as possible. This also helps in improving the model's performance.

```
[7]  # split into train and test sets
     train_size = int(len(dataset1) * 0.67)
     test_size = len(dataset1) - train_size
     train, test = dataset1[0:train_size,:], dataset1[train_size:len(dataset),:]
     print(len(train), len(test))

     5504 2712
```

Fig. 21 Splitting the data

Fig. 21 shows the splitting of data using the length of the dataset. 67% of the dataset is used to train the model. The remaining 33% is used as test data. Hence, 5504 rows are used for training the model and 2712 rows are used in the test dataset.

b) Build the Model: Define the Long Short-Term Memory model (LSTM) and clearly explain the input features as a function of time lag.

```
[8]  # convert an array of values into a dataset matrix
     def create_dataset(dataset1, look_back=1):
       dataX, dataY = [], []
       for i in range(len(dataset1)-look_back-1):
         a = dataset1[i:(i+look_back), 0]
         dataX.append(a)
         dataY.append(dataset1[i + look_back, 0])
       return np.array(dataX), np.array(dataY)
```

Fig. 22 Converting the array of values into a dataset matrix

We start by converting the array of values into a dataset matrix as that is required by the LSTM model. The code is shown in fig. 22.

```
[9]  # reshape into X=t and Y=t+1
     look_back = 1
     trainX, trainY = create_dataset(train, look_back)
     testX, testY = create_dataset(test, look_back)
```

Fig. 23 Create dataset function being called

We called the create dataset function with lookup value as 1. The lookup value are n number of values in the past data that are used to predict the future data (Jose, 2019). The dataset that will be used to train the model is denoted by trainX and trainY. The test dataset is denoted by testX and testY.

```
[10] # reshape input to be [samples, time steps, features]
     trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
     testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

Fig. 24 Reshaping the data using np.reshape

By using the code shown in fig. 24, we reshape the data in the desired format.

```
# create and fit the LSTM network

model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
```

```
Epoch 1/100
5502/5502 - 11s - loss: 0.0023 - 11s/epoch - 2ms/step
Epoch 2/100
5502/5502 - 8s - loss: 6.6361e-05 - 8s/epoch - 2ms/step
Epoch 3/100
5502/5502 - 9s - loss: 6.4523e-05 - 9s/epoch - 2ms/step
Epoch 4/100
5502/5502 - 9s - loss: 6.2863e-05 - 9s/epoch - 2ms/step
Epoch 5/100
5502/5502 - 9s - loss: 6.3531e-05 - 9s/epoch - 2ms/step
Epoch 6/100
5502/5502 - 9s - loss: 6.3299e-05 - 9s/epoch - 2ms/step
Epoch 7/100
5502/5502 - 9s - loss: 6.3863e-05 - 9s/epoch - 2ms/step
Epoch 8/100
5502/5502 - 9s - loss: 6.1730e-05 - 9s/epoch - 2ms/step
Epoch 9/100
5502/5502 - 8s - loss: 6.2410e-05 - 8s/epoch - 1ms/step
Epoch 10/100
5502/5502 - 10s - loss: 6.1537e-05 - 10s/epoch - 2ms/step
Epoch 11/100
```

Fig. 25 Creating and fitting the LSTM model

We use the above code to create and fit the LSTM model for the training data. It has variables such as epoch and batch size.

Epoch is the number of times the dataset is run through the model. It needs to be appropriately defined as it leads to the model over fitting or underfitting the dataset. In our dataset, we take this to be 100.

Batch size is the number of batches the given dataset needs to be divided in while training. It is decided based on the size of dataset. A larger dataset would require multiple batch_size. However, our current dataset isn't too large and the batch_size is set at full batch. Batch_size needs to be as small as possible without expending the entire memory (Brownlee, 2022).

We use the Mean Squared Error loss function and Adam stochastic optimizer. The use of Adam has been increasing consistently (Karpathy, 2017). It can be described as a combination of RMSprop and Stochastic gradient descent with momentum (Bushaev, 2018).

We fit our trainX and trainY using the code shown in fig. 25

```
[30] model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_2 (LSTM)               (None, 4)                 96

 dense_2 (Dense)             (None, 1)                 5

=================================================================
Total params: 101
Trainable params: 101
Non-trainable params: 0
_________________________________________________________________
```

Fig. 26 Model summary

The above model summary shows that this model has 2 layers, one hidden and the other outer later. It also shows that out of all the 101 parameters, all 101 were trainable.

c) Prediction Function and Result: In this step, we are running the model using the test data we defined in step four. Visualise the predicted versus the actual stock values for the specific time period

```
[27] # make predictions
     trainPredict = model.predict(trainX)
     testPredict = model.predict(testX)
     # invert predictions
     trainPredict = scaler.inverse_transform(trainPredict)
     trainY = scaler.inverse_transform([trainY])
     testPredict = scaler.inverse_transform(testPredict)
     testY = scaler.inverse_transform([testY])
     # calculate root mean squared error
     trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
     print('Train Score: %.2f RMSE' % (trainScore))
     testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
     print('Test Score: %.2f RMSE' % (testScore))
```

```
172/172 [==============================] - 1s 2ms/step
85/85 [==============================] - 0s 2ms/step
Train Score: 0.99 RMSE
Test Score: 1.78 RMSE
```

Fig. 27 Making predictions and inverse scaler

The above fig. 27 shows us predicting the values. Following this, our model is complete. However, since our data has been scaled initially, we need to inverse scale it to get our desired values. We do that by using inverse scaler as shown above.

We then calculate the RMSE on the test and train test. They are 0.99 for the train score and 1.78 for the test score.

These results help us conclude that the model is predicting well for the given dataset. However, considering that test RMSE is larger than the train RMSE, we can conclude that the model overfit the dataset. But overall, the prediction has been fine and can be considered to be accurate enough.

Visualisation of actual vs predicted data

```
[45] fig, ax = plt.subplots(figsize=(50,15))
     # shift train predictions for plotting
     trainPredictPlot= np.empty_like(dataset1)
     trainPredictPlot[:, :] = np.nan
     trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
     # shift test predictions for plotting
     testPredictPlot = np.empty_like(dataset1)
     testPredictPlot[:, :] = np.nan
     testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset1)-1, :] = testPredict
     # plot baseline and predictions
     ax.plot(scaler.inverse_transform(dataset1),label='original dataset ')  #blue
     ax.plot(trainPredictPlot, label='predictions for the training dataset')   #orange
     ax.plot(testPredictPlot, label='predictions on the unseen test dataset ')     # green
```

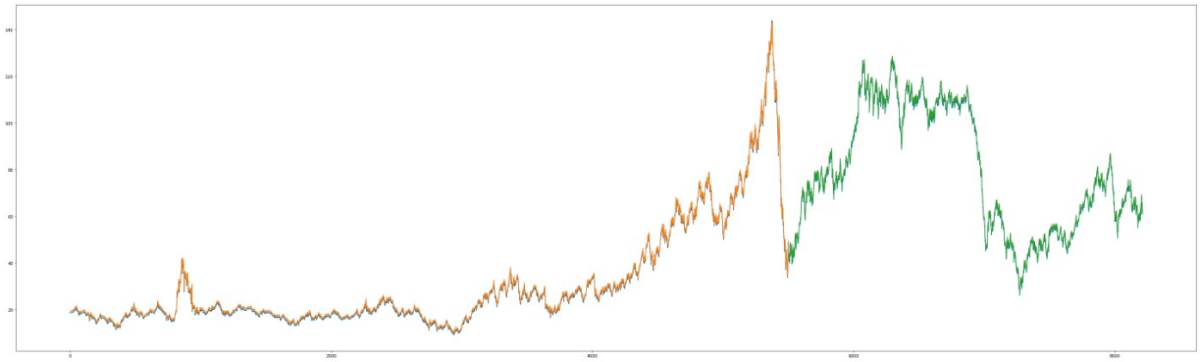

Fig. 28 Plotting of Original dataset, actual test value and predicted value

In the above visualisation, the orange line is the training dataset. The green line is predicted or test dataset. The blue line is the original dataset or can be considered the actual value. We need to compare the green and blue line to understand how well our prediction has been.

Conclusion: Inspite of slight overfitting that has been showcased by the test RMSE being higher than the train RMSE, we can see that the model has predicted well.

Advice for the marketing strategist:

Linear regression is generally used for forecasting sales figures and to assess risk for health insurance companies.

LSTM is generally used for data with continuation. This includes time series data like stock prices. Additionally, it is widely used for speech recognition, text generation and image recognition.

Decision: The decision to use either of these models depends on the type of data available and the type of decision that must be made based on it. As for the dataset above, when we compare the RMSE for both models, we see that the linear regression with RMSE 0.9258 has outperformed the LSTM model with RMSE (test) 1.78.

References:

1. Maulud, D. and Abdulazeez, A.M. (2020) "A review on linear regression comprehensive in machine learning," *Journal of Applied Science and Technology Trends*, 1(4), pp. 140–147. Available at: https://doi.org/10.38094/jastt1457.
2. Seber, G.A. and Lee, A.J. (2003) "Linear regression analysis," *Wiley Series in Probability and Statistics* [Preprint]. Available at: https://doi.org/10.1002/9780471722199.
3. Shumeiko, D. and Rozora, I. (no date) *Handling missing values in machine learning regression problems*. Available at: https://ceur-ws.org/Vol-3106/Short_5.pdf (Accessed: January 4, 2023).
4. Julian (2022) *Reproducible ML: Maybe you shouldn't be using Sklearn's train_test_split*, *Engineering for Data Science* 🛠. Available at: https://engineeringfordatascience.com/posts/ml_repeatable_splitting_using_hashing/ (Accessed: January 4, 2023).
5. Real Python (2021) *Split your dataset with scikit-learn's train_test_split()*, *Real Python*. Real Python. Available at: https://realpython.com/train-test-split-python-data/ (Accessed: January 4, 2023).
6. Van Houdt, G., Mosquera, C. and Nápoles, G. (2020) "A review on the long short-term memory model," *Artificial Intelligence Review*, 53(8), pp. 5929–5955. Available at: https://doi.org/10.1007/s10462-020-09838-1.
7. Srivastava, P. (2020) *Long short term memory: Architecture of LSTM*, *Analytics Vidhya*. Available at: https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/ (Accessed: January 4, 2023).
8. Jose, G.V. (2019) *Time series forecasting with Recurrent Neural Networks*, *Medium*. Towards Data Science. Available at: https://towardsdatascience.com/time-series-forecasting-with-recurrent-neural-networks-74674e289816 (Accessed: January 4, 2023).
9. Brownlee, J. (2022) *Difference between a batch and an epoch in a neural network*, *MachineLearningMastery.com*. Available at: https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/ (Accessed: January 4, 2023).

10. Karpathy, A. (2017) *A peek at trends in machine learning*, *Medium*. Medium. Available at: https://karpathy.medium.com/a-peek-at-trends-in-machine-learning-ab8a1085a106 (Accessed: January 4, 2023).
11. Bushaev, V. (2018) *Adam - latest trends in deep learning optimization.*, *Medium*. Towards Data Science. Available at: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c (Accessed: January 4, 2023).