INSTITUTE FOR ADVANCED COMPUTING

AND SOFTWARE DEVELOPMENT AKURDI,

PUNE

Build a model to predict stock market performance using LSTM

PG-DBDA SEP 2023

Submitted By:

Group No: 05

Roll No.	**Name**
239540	**Shreyas Dande**
239548	**Viraj Kawade**

Mrs. Priyanka Bhor
Project Guide

Mr. Rohit Puranik
Centre Coordinator

Abstract

The stock market is a complex system influenced by various factors, making it challenging to predict accurately. In this project, we aim to develop a predictive model for stock market performance using Long Short-Term Memory (LSTM) networks, a type of recurrent neural network (RNN) known for its ability to capture sequential patterns in data. We preprocess the data, train the LSTM model, and evaluate its performance using historical stock price data. Our goal is to provide insights into the potential of deep learning techniques for stock market prediction.

Acknowledgement

We acknowledge the support and guidance received during the development of this project. We extend our sincere gratitude to all individuals and organizations that contributed to our learning journey .

First and foremost, we would like to thank our project guide **Mrs. Priyanka Bhor** Madam for their constant guidance and support throughout the project. We extend our sincere thanks to our respected centre coordinator, **Mr. Rohit Puranik** Sir for allowing us to use the facilities available.

We would also like to express our appreciation to the faculty members of our department for their constructive feedback and encouragement. Their insights and suggestions have helped us to refine our ideas and announce the quality of our work.

Furthermore, we would like to thank our families and friends for their unwavering support and encouragement throughout our academic journey, their love and support have been a constant source of motivation and inspiration for us.

Thank you for all your valuable contributions to our project

Viraj Kawade (239548)

Shreyas Dande (239540)

Table of Contents

Introduction

The stock market plays a crucial role in the global economy, with investors seeking to make informed decisions based on market trends and stock price movements. Predicting stock market performance is a challenging task due to the dynamic nature of financial markets and the influence of various external factors. Traditional statistical methods often struggle to capture the complex patterns present in stock market data. In recent years, machine learning techniques, particularly deep learning models like LSTM networks, have shown promise in predicting stock prices.

Project Objective

The primary objective of this project is to develop a predictive model for stock market performance using LSTM networks. By leveraging historical stock price data and deep learning techniques, we aim to build a model capable of forecasting future stock prices with a high degree of accuracy. The project seeks to explore the potential of LSTM networks in capturing the temporal dependencies and patterns inherent in stock market data.

Project Overview

- Data Preparation:
  We preprocess the historical stock price data, including cleaning, normalization, and feature engineering, to prepare it for model training.

- LSTM Model Training:
  We design and train an LSTM neural network using the preprocessed data. The model architecture consists of multiple LSTM layers followed by dense layers for prediction.

- Model Evaluation:
  We evaluate the trained LSTM model's performance using test data and metrics such as mean squared error (MSE) and coefficient of determination (R-squared).

- Visualization:
  We visualize the predicted stock prices and compare them with actual prices to assess the model's accuracy and effectiveness.

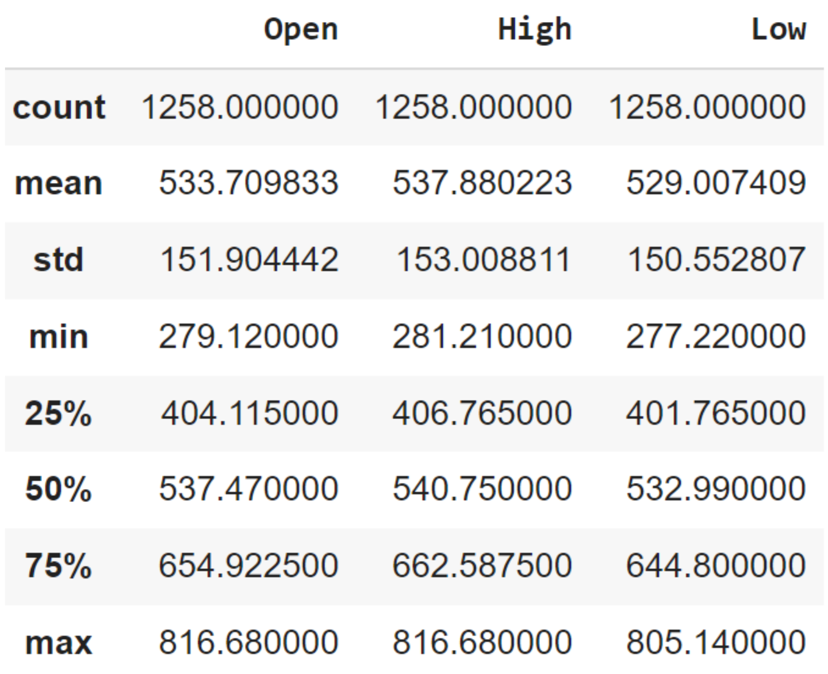
Project Scope

This project focuses on predicting stock market performance for a specific dataset (Google stock prices). While the LSTM model demonstrates its effectiveness in this context, further research and experimentation may be required to generalize the approach to other stocks and financial markets. Additionally, the project's scope includes exploring different LSTM architectures, hyperparameters, and training strategies to optimize model performance.

Project Details

Generating descriptive statistics of the DataFrame 'data'.This will be useful for getting a quick statistical overview of the dataset and understanding the distribution and variability of the data within each column.

data.describe()



	Open	High	Low
count	1258.000000	1258.000000	1258.000000
mean	533.709833	537.880223	529.007409
std	151.904442	153.008811	150.552807
min	279.120000	281.210000	277.220000
25%	404.115000	406.765000	401.765000
50%	537.470000	540.750000	532.990000
75%	654.922500	662.587500	644.800000
max	816.680000	816.680000	805.140000

Replacing the ',' with an empty character for the whole data['Close'] column, this will be helpful in typecasting the field as float from object

data['Close'] = data['Close'].apply(lambda x: float(x.replace(',', '')))

This line of code applies a lambda function to each element in the 'Close' column.

- Within the lambda function, **x.replace(',', '')** removes commas from the string representation of the number.
- The **float()** function then converts the resulting string (with commas removed) into a floating-point number.
- The converted values are assigned back to the 'Close' column.

# Converting the data['Close'] as type float.

data['Close']=data['Close'].astype('float')

Typecasting to Float:

data['Close'] = data['Close'].astype('float')

- After removing commas and converting the 'Close' column elements to float using the **apply** method, this line explicitly converts the 'Close' column to type float using the **astype()** method.

The purpose of these operations is to ensure that the 'Close' column contains numerical (float) values without commas, making it suitable for numerical operations and analysis. This preprocessing step is common when dealing with datasets where numeric values are represented as strings with formatting symbols like commas.

# Replacing the ',' with an empty character for the whole data['Volume'] column, this will be helpful in typecasting the field as float from object

data['Volume'] = data['Volume'].apply(lambda x: float(x.replace(',', '')))

- This line of code applies a lambda function to each element in the 'Volume' column.
- Within the lambda function, **x.replace(',', '')** removes commas from the string representation of the number.
- The **float()** function then converts the resulting string (with commas removed) into a floating-point number.
- The converted values are assigned back to the 'Volume' column.

# Converting the data['Volume'] as type int.

data['Volume']= data['Volume'].astype('int')

Typecasting to Integer:

After removing commas and converting the 'Volume' column elements to float using the **apply** method, this line explicitly converts the 'Volume' column to type integer using the **astype()** method.

The purpose of these operations is to ensure that the 'Volume' column contains numerical (integer) values without commas, making it suitable for numerical operations and analysis. This preprocessing step is common when dealing with datasets where numeric values are represented as strings with formatting symbols like commas. However, note that converting volume to an integer may result in loss of precision, especially if the volume values are large. Depending on your requirements, you may choose to keep the volume values as floating-point numbers.

Extracting and converting the 'Open' column from the DataFrame 'data' to a NumPy array:

open_stock_price_df = data[['Open']]

open_stock_price_df = np.array(open_stock_price_df)

The Above code snippet extracts the 'Open' column from the DataFrame **data** and converts it into a NumPy array.

1. **Extracting the 'Open' Column:**

open_stock_price_df = data[['Open']]

- This line of code selects only the 'Open' column from the DataFrame **data** and creates a new DataFrame called **open_stock_price_df**.
- The double square brackets (**[['Open']]**) are used to select a single column as a DataFrame, preserving its DataFrame structure.

2. **Converting to NumPy Array:**

open_stock_price_df = np.array(open_stock_price_df)

- After selecting the 'Open' column as a DataFrame, this line of code converts the DataFrame **open_stock_price_df** into a NumPy array using the **np.array()** function.
- Now, **open_stock_price_df** is a NumPy array containing the values from the 'Open' column of the DataFrame **data**.

After executing these lines of code, you will have a NumPy array (**open_stock_price_df**) containing the values from the 'Open' column of the DataFrame **data**, which you can use for further analysis, processing, or feeding into machine learning models.

#Scaling the 'open_stock_price_df' array using the MinMaxScaler:

scaler = MinMaxScaler(feature_range=(0,1))

open_stock_price_scaled = scaler.fit_transform(open_stock_price_df)

open_stock_price_scaled

1. **Create a MinMaxScaler Object:**

scaler = MinMaxScaler(feature_range=(0,1))

- This line creates a MinMaxScaler object with the desired feature range, which is set to (0,1) in this case.

2. **Reshape the Array (if Necessary):**

pythonCopy code

open_stock_price_df = open_stock_price_df.reshape(-1, 1)

- Before scaling the data, you may need to reshape the array to ensure that it has the correct shape for the MinMaxScaler. MinMaxScaler expects a 2D array where each column represents a feature. Here, we reshape it to (-1, 1) to indicate that there is only one feature.

3. **Fit and Transform the Data:**

pythonCopy code

open_stock_price_scaled = scaler.fit_transform(open_stock_price_df)

- This line fits the MinMaxScaler to the data and transforms it. The **fit_transform()** method computes the minimum and maximum values of the data and then scales the data to the specified range (0 to 1).

After executing these lines of code, **open_stock_price_scaled** will contain the scaled values of the 'Open' column, which are normalized to the range between 0 and 1. These scaled values are often more suitable for use in machine learning algorithms and other analyses.

```
x_train = []
y_train = []

for i in range(memory_state, data.shape[0]):
    x_train.append(open_stock_price_scaled[i-memory_state:i, 0])
    y_train.append(open_stock_price_scaled[i, 0])
x_train=np.array(x_train)
y_train=np.array(y_train)
```

The Above code snippet is used to prepare training data for a time series forecasting model, likely for an LSTM network.

1. **Initialization of Lists:**
    - Two empty lists **x_train** and **y_train** are initialized to store the input sequences and corresponding target values, respectively.
2. **Loop for Creating Training Sequences:**
    - The loop iterates over the range from **memory_state** to the number of rows in the dataset (**data.shape[0]**).
    - It generates training sequences and their corresponding target values.
    - For each iteration:
        - **x_train.append(open_stock_price_scaled[i-memory_state:i, 0])**: It appends a subsequence of length **memory_state** from **open_stock_price_scaled** to **x_train**. This subsequence represents the input sequence at time **i**.
        - **y_train.append(open_stock_price_scaled[i, 0])**: It appends the target value (the next data point after the input sequence) to **y_train**.
3. **Conversion to NumPy Arrays:**
    - After the loop, the lists **x_train** and **y_train** are converted to NumPy arrays using **np.array()**.

The resulting **x_train** array will contain sequences of length **memory_state**, where each sequence represents historical data used for prediction. The corresponding **y_train** array contains the target values, which are the next data points following the input sequences.

This code prepares the training data in the appropriate format for training a time series forecasting model, such as an LSTM network. The **memory_state** variable determines the length of the input sequences, which is a crucial parameter in defining the model's memory or context for making predictions.

Reshaping the 'x_train' array for compatibility with LSTM network input requirements:

x_train=np.reshape(x_train,(x_train.shape[0],x_train.shape[1],1))

The Above code snippet reshapes the **x_train** array to comply with the input requirements of an LSTM network

x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))

Reshaping the Array:

- The **np.reshape()** function is used to reshape the **x_train** array.
- The new shape passed as the second argument is **(x_train.shape[0], x_train.shape[1], 1)**.
- The first dimension **x_train.shape[0]** remains unchanged and represents the number of samples or sequences.
- The second dimension **x_train.shape[1]** represents the length of each sequence, which is the number of timesteps or memory states.
- The third dimension **1** indicates the number of features per timestep. In this case, there's only one feature (the 'Open' price), so it is set to 1.

By reshaping the array in this way, it conforms to the input shape required by LSTM networks, which expect input data in a 3D format **(batch_size, timesteps, input_dim)**. In this context:

- **batch_size**: Represents the number of samples or sequences in the training data.
- **timesteps**: Represents the length of each sequence, i.e., the number of time steps or memory states.
- **input_dim**: Represents the number of features per timestep.

The reshaped **x_train** array is now compatible with the input requirements of LSTM networks and can be used for training the model. Each sample in **x_train** represents a sequence of historical 'Open' prices, with each sequence containing a specified number of timesteps.

```
 # Initializing a new Sequential model and assigning it to the variable
'lstm_model':
lstm_model=Sequential()

# Adding first layers to the LSTM neural network model 'lstm_model':
lstm_model.add(LSTM(units=128,return_sequences=True,
input_shape=(x_train.shape[1],1)))
lstm_model.add(Dropout(0.2))

# Adding Second layers to the LSTM neural network model 'lstm_model':

lstm_model.add(LSTM(units=128,return_sequences=True))
lstm_model.add(Dropout(0.2))

# Adding third layers to the LSTM neural network model 'lstm_model':
lstm_model.add(LSTM(units=128))
lstm_model.add(Dropout(0.2))

# Adding a Dense layer to the LSTM model 'lstm_model':
lstm_model.add(Dense(units=1))
```

```
def r_squared(y_true, y_pred):
 SS_res = K.sum(K.square(y_true - y_pred))
 SS_tot = K.sum(K.square(y_true - K.mean(y_true)))
 return (1 - SS_res / (SS_tot + K.epsilon()))
```

The Above code computes the coefficient of determination (R-squared) between the true target values (**y_true**) and the predicted target values (**y_pred**).

1. **Importing Keras Backend:**
2. **Keras.backend** provides functions that operate on tensors, which are used to perform computations in the backend (e.g., TensorFlow or Theano).
3. **Defining the r_squared Function:**
    - The function **r_squared(y_true, y_pred)** takes two arguments:
        - **y_true**: The true target values.
        - **y_pred**: The predicted target values.
4. **Calculating Residual Sum of Squares (SS_res):**
    - **SS_res = K.sum(K.square(y_true - y_pred))**: Computes the sum of squared differences between the true target values and the predicted target values.
5. **Calculating Total Sum of Squares (SS_tot):**
    - **SS_tot = K.sum(K.square(y_true - K.mean(y_true)))**: Computes the total sum of squares, which is the sum of squared differences between the true target values and their mean.
6. **Computing R-squared:**
    - The R-squared value is calculated using the formula:

      R2 = 1 – (RSS/TSS)

      where RSS is the residual sum of squares

      TSS is the total sum of squares
7. **Returning R-squared Value:**
    - The function returns the computed R-squared value.

R-squared is a statistical measure that represents the proportion of the variance in the dependent variable that is predictable from the independent variables. It ranges from 0 to 1, where 1 indicates that the model perfectly predicts the target variable, and 0 indicates that the model does not explain any of the variability of the target variable.

```
# Compiling the LSTM model 'lstm_model'

lstm_model.compile(optimizer='nadam', loss='mean_squared_error',
metrics=[r_squared])
```

The **lstm_model.compile()** function in Keras is used to configure the learning process of the model. It specifies the optimizer, loss function, and evaluation metrics to be used during training .

1. **Optimizer (optimizer='nadam'):**
    - The **'nadam'** optimizer is specified. Nadam stands for Nesterov Adam optimizer, which is an extension of the Adam optimizer. It combines the benefits of both Adam and Nesterov momentum methods.
2. **Loss Function (loss='mean_squared_error'):**
    - The mean squared error (MSE) loss function is specified. MSE is a common choice for regression problems, including time series forecasting, where the goal is to minimize the squared differences between the predicted and true values.
3. **Metrics (metrics=[r_squared]):**
    - The evaluation metric used during training is specified as the R-squared function (**r_squared**) that you defined earlier.
    - During training, the model will compute the R-squared value in addition to the loss, which helps monitor the model's performance and how well it fits the training data.

With this configuration, the model will be trained using the Nadam optimizer to minimize the mean squared error loss function, and the R-squared metric will be computed and reported during training to evaluate the model's performance on the training data. This setup is suitable for regression tasks where the goal is to predict continuous values, such as stock prices.

# Training the LSTM model 'lstm_model' with the training data in 200 epochs:

```
history = lstm_model.fit(x_train,y_train,epochs=200,
batch_size=32,validation_split=0.3)
```

1. **Training Data (x_train, y_train):**
    - **x_train**: Input sequences (features) used for training.
    - **y_train**: Target values corresponding to the input sequences, used for training the model to predict the next value in the sequence.
2. **Epochs (epochs=200):**
    - The number of epochs specifies how many times the entire training dataset will be passed forward and backward through the neural network.
    - In this case, the model will be trained for 200 epochs.
3. **Batch Size (batch_size=32):**
    - The batch size determines the number of samples that will be used to update the model's weights in each iteration.
    - A batch size of 32 means that 32 samples will be processed at once before updating the weights.
    - Using mini-batches (as opposed to processing the entire dataset at once) helps to speed up the training process and can also help to regularize the model.
4. **Validation Split (validation_split=0.3):**
    - The validation split parameter specifies the fraction of the training data to be used for validation during training.

- In this case, 30% of the training data (**x_train** and **y_train**) will be reserved for validation.
- The validation data is used to monitor the model's performance on data that it hasn't seen during training and can help detect overfitting.

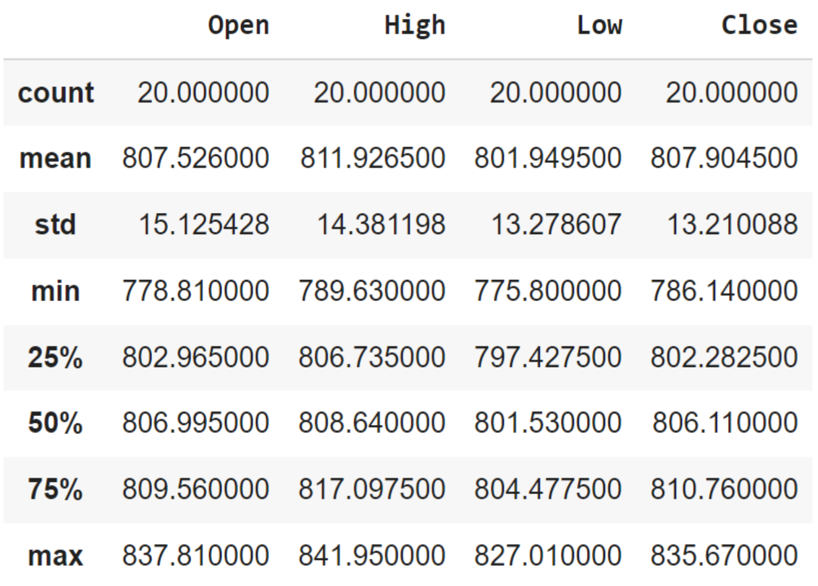
5. **History Object:**
    - The **history** object contains information about the training process, including the loss and metric values recorded during training and validation.

After training is complete, the **history** object will contain the training history, which can be used to visualize the training and validation loss (and any other metrics) over the epochs. This information can help in evaluating the model's performance and diagnosing potential issues such as overfitting.

# Describing the test data

test_data.describe()

When you execute this code, pandas will compute and display statistics such as count, mean, standard deviation, minimum, quartiles, and maximum values for each numerical column in the **test_data** DataFrame.



	Open	High	Low	Close
count	20.000000	20.000000	20.000000	20.000000
mean	807.526000	811.926500	801.949500	807.904500
std	15.125428	14.381198	13.278607	13.210088
min	778.810000	789.630000	775.800000	786.140000
25%	802.965000	806.735000	797.427500	802.282500
50%	806.995000	808.640000	801.530000	806.110000
75%	809.560000	817.097500	804.477500	810.760000
max	837.810000	841.950000	827.010000	835.670000

```
test_data['Volume'] = test_data['Volume'].apply(lambda x: float(x.replace(',', '')))
test_data['Volume'] = test_data['Volume'].astype('int')
```

1. **Replacing Commas and Typecasting to Float:**
   - This line of code applies a lambda function to each element in the 'Volume' column.
   - Within the lambda function, **x.replace(',', '')** removes commas from the string representation of the number.
   - The **float()** function then converts the resulting string (with commas removed) into a floating-point number.
   - The converted values are assigned back to the 'Volume' column.
2. **Typecasting to Integer:**
   - After removing commas and converting the 'Volume' column elements to float using the **apply** method, this line explicitly converts the 'Volume' column to type integer using the **astype()** method.

The purpose of these operations is to ensure that the 'Volume' column contains numerical (integer) values without commas, making it suitable for numerical operations and analysis. This preprocessing step is common when dealing with datasets where numeric values are represented as strings with formatting symbols like commas. However, note that converting volume to an integer may result in loss of precision, especially if the volume values are large. Depending on your requirements, you may choose to keep the volume values as floating-point numbers.

test_open_stock_price_df=test_data.iloc[:,1:2].values

The Above code you provided selects the second column of the **test_data** DataFrame and converts it into a NumPy array.

- **test_data.iloc[:, 1:2]**: This selects all rows (**:**) and the second column (**1:2**) of the **test_data** DataFrame using integer-based indexing.
- **.values**: This converts the selected portion of the DataFrame into a NumPy array.

After executing this line of code, **test_open_stock_price_df** will be a NumPy array containing the values from the second column (index 1) of the **test_data** DataFrame. This is often used to extract the target variable or response variable (in this case, likely the stock prices) from the DataFrame for further analysis or model evaluation.

# Extracting and converting the 'Open' column from the DataFrame 'test_data' to a NumPy array:

test_open_stock_price_df=test_data[['Open']]

test_open_stock_price_df=np.array(test_open_stock_price_df

1. **Extracting the 'Open' Column:**
    - This line of code selects only the 'Open' column from the DataFrame **test_data** and creates a new DataFrame called **test_open_stock_price_df**.
    - The double square brackets (**[['Open']]**) are used to select a single column as a DataFrame, preserving its DataFrame structure.
2. **Converting to NumPy Array:**

test_open_stock_price_df = np.array(test_open_stock_price_df)

- After selecting the 'Open' column as a DataFrame, this line of code converts the DataFrame **test_open_stock_price_df** into a NumPy array using the **np.array()** function.
- Now, **test_open_stock_price_df** is a NumPy array containing the values from the 'Open' column of the DataFrame **test_data**.

After executing these lines of code, you will have a NumPy array (**test_open_stock_price_df**) containing the values from the 'Open' column of the DataFrame **test_data**, which you can use for further analysis, processing, or feeding into machine learning models.

```
# Preparing input data for the LSTM model prediction on the test dataset:

dataset=pd.concat((data['Open'],test_data['Open']),axis=0)
inputs=dataset[len(dataset)-len(test_data)-memory_state:].values
inputs.shape
```

1. **Concatenating Datasets (pd.concat):**

```
dataset = pd.concat((data['Open'], test_data['Open']), axis=0)
```

- This line concatenates the 'Open' columns from the training dataset (**data**) and the test dataset (**test_data**) along the vertical axis (axis=0). The resulting **dataset** contains both the training and test data.

2. **Selecting Input Data (inputs):**

```
inputs = dataset[len(dataset) - len(test_data) - memory_state:].values
```

- This line selects a portion of the combined dataset (**dataset**) for preparing input data.
- The selected portion starts from the index **len(dataset) - len(test_data) - memory_state** to the end of the dataset.
- The **.values** attribute converts the selected portion to a NumPy array, and it represents the input data for the LSTM model prediction on the test dataset.

The variable **memory_state** seems to be the length of the input sequences used in training the LSTM model. The shape of the **inputs** array will give you information about the size of the input data that will be fed into the model for prediction on the test dataset.

```
# Reshaping the 'inputs' array for model prediction:

inputs=inputs.reshape(-1,1)
```

The Above code snippet reshapes the **inputs** array to prepare it for model prediction.

- The **reshape()** function is used to change the shape of the array **inputs**.
- The new shape passed as the argument is **(-1, 1)**.
- The **-1** in the reshape function means that NumPy will automatically calculate the size of the first dimension (the number of rows) based on the size of the second dimension (the number of columns) and the total number of elements.
- The **1** in the second dimension indicates that each row will have only one element.

This reshaping operation effectively converts the **inputs** array into a 2D array where each row contains a single element. This format is often required for feeding data into machine learning models, including LSTM models, where each sample is represented as a row with one or more features.

```
# Scaling the reshaped 'inputs' using the previously defined MinMaxScaler
'scaler':


inputs = scaler.transform(inputs)
inputs
```

The Above code snippet scales the reshaped **inputs** array using the previously defined MinMaxScaler named **scaler**.

The **transform()** method of the MinMaxScaler object **scaler** is called on the **inputs** array.

- This method scales the values in the **inputs** array based on the minimum and maximum values learned from the training data used to fit the scaler.
- The scaled values are assigned back to the **inputs** array.

By scaling the inputs using the same scaler that was used to scale the training data, you ensure that the input data for prediction is transformed in the same way as the training data. This consistency in scaling is important for maintaining the integrity of the data and ensuring that the model makes predictions based on comparable scales.

```
# Preparing the test dataset for the LSTM model:
x_test=[]
for i in range(memory_state,inputs.shape[0]):
   x_test.append(inputs[i-memory_state:i,0])
x_test=np.array(x_test)
```

The Above code prepares the test dataset for the LSTM model by creating sequences of input data.

1. **Initialization of x_test List:**
   - An empty list named **x_test** is initialized to store sequences of input data for testing.
2. **Creating Test Sequences:**
   - The loop iterates over the range from **memory_state** to the number of rows in the **inputs** array (**inputs.shape[0]**).
   - For each iteration:
     - **inputs[i - memory_state:i, 0]**: It selects a subsequence of length **memory_state** from the **inputs** array starting from index **i - memory_state** up to index **i**. The **0** index indicates that we are selecting the first (and only) feature from the input data.
     - The selected subsequence represents the input data at time **i**, and it is appended to the **x_test** list.
3. **Converting to NumPy Array:**
   - After the loop, the list of input sequences **x_test** is converted to a NumPy array using **np.array()**.

After executing this code, **x_test** will contain sequences of input data formatted in a way suitable for testing with the LSTM model. Each sequence in **x_test** represents historical input data, and the model will make predictions based on these sequences.

Reshaping the test dataset 'x_test' for compatibility with LSTM model input requirements:

```
x_test=np.reshape(x_test,(x_test.shape[0],x_test.shape[1],1))
x_test.shape
```

The Above code reshapes the test dataset **x_test** to comply with the input requirements of an LSTM model. Here's how the reshaping is done:

- The **np.reshape()** function is used to reshape the **x_test** array.
- The new shape passed as the second argument is **(x_test.shape[0], x_test.shape[1], 1)**.
- The first dimension **x_test.shape[0]** represents the number of samples or sequences in the test dataset.
- The second dimension **x_test.shape[1]** represents the length of each sequence, i.e., the number of time steps or memory states.
- The third dimension **1** indicates the number of features per time step. In this case, there's only one feature (the 'Open' price), so it is set to 1.

By reshaping the array in this way, it conforms to the input shape required by LSTM networks, which expect input data in a 3D format **(batch_size, timesteps, input_dim)**. In this context:

- **batch_size**: Represents the number of samples or sequences in the test dataset.
- **timesteps**: Represents the length of each sequence, i.e., the number of time steps or memory states.
- **input_dim**: Represents the number of features per time step.

The reshaped **x_test** array is now compatible with the input requirements of LSTM networks and can be used for making predictions using the trained model. Each sample in **x_test** represents a sequence of historical 'Open' prices, with each sequence containing a specified number of timesteps.

```
# Generating predictions for the test dataset using the trained LSTM model:

predicted_stock_price=lstm_model.predict(x_test)
```

The Above code you provided generates predictions for the test dataset using the trained LSTM model.

- The **predict()** method of the LSTM model (**lstm_model**) is called on the test dataset (**x_test**).
- This method generates predictions for the input sequences in **x_test** using the trained LSTM model.
- The predictions are stored in the variable **predicted_stock_price**.

After executing this code, **predicted_stock_price** will contain the model's predictions for the test dataset. Each element in **predicted_stock_price** corresponds to a predicted value for the next time step based on the input sequences provided in **x_test**. These predictions can be compared with the actual values from the test dataset to evaluate the model's performance.

```
# Converting the scaled predictions back to their original scale:
predicted_stock_price=scaler.inverse_transform(predicted_stock_price)


#Displaying the Predicted Stock Price
predicted_stock_price
```

The Above code converts the scaled predictions back to their original scale using the **inverse_transform()** method of the **scaler** object.

- The **inverse_transform()** method of the **scaler** object is called on the **predicted_stock_price** array.
- This method reverses the scaling transformation applied to the predictions during preprocessing.
- It restores the predictions to their original scale, making them interpretable in the same units as the original data.

After executing this code, **predicted_stock_price** will contain the predicted stock prices in their original scale. These prices can be directly compared with the actual stock prices to assess the accuracy of the LSTM model's predictions.

Visualization Plot

# Visualizing the Actual vs Predicted Google Stock Prices for January 2017 using matplotlib:

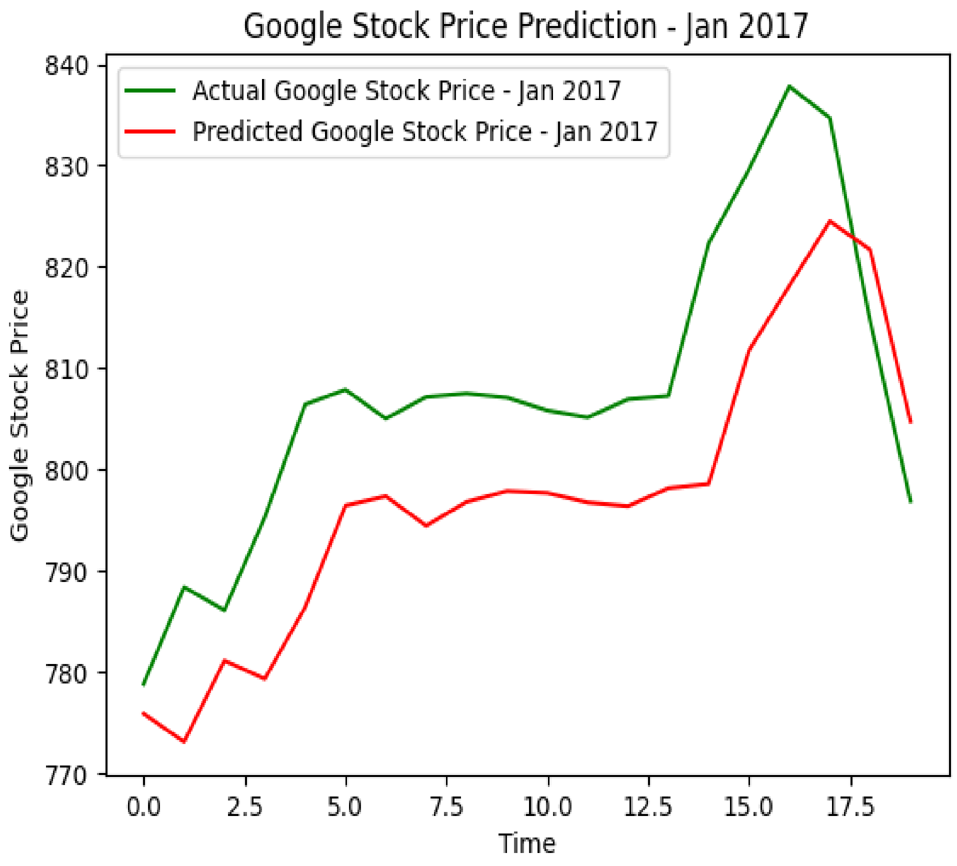
```
plt.plot(test_open_stock_price_df,color='green', label="Actual Google Stock Price - Jan 2017")
plt.plot(predicted_stock_price,color='red', label="Predicted Google Stock Price - Jan 2017")
plt.title('Google Stock Price Prediction - Jan 2017')
plt.xlabel('Time')
plt.ylabel("Google Stock Price")
plt.legend()
plt.show()
```

The Above code uses Matplotlib to plot the actual and predicted Google stock prices for January 2017.

plt.plot() is used to plot the actual Google stock prices (**test_open_stock_price_df**) in green and the predicted Google stock prices (**predicted_stock_price**) in red.

- **plt.title()** sets the title of the plot.
- **plt.xlabel()** sets the label for the x-axis.
- **plt.ylabel()** sets the label for the y-axis.
- **plt.legend()** displays the legend to differentiate between the actual and predicted stock prices.
- **plt.show()** displays the plot.

This code will create a line plot showing the actual and predicted Google stock prices for January 2017. It allows for visual comparison between the actual stock prices and the model's predictions, helping to assess the performance of the LSTM model in predicting stock prices.



R-squared is a statistical measure that represents the proportion of the variance in the dependent variable that is predictable from the independent variables. . The R-squared value is a measure of how well the predictions made by a model fit the actual data .It ranges from 0 to 1, where 1 indicates that the model perfectly predicts the target variable, and 0 indicates that the model does not explain any of the variability of the target variable .

```
    """
Calculate the coefficient of determination, R-squared, using NumPy.
Parameters:
    y_true (numpy.ndarray): Actual values.
    y_pred (numpy.ndarray): Predicted values.
Returns:
    float: R-squared value.
    """


def numpy_r_squared(y_true, y_pred):
    residual_sum_of_squares = np.sum((y_true - y_pred) ** 2)
    total_sum_of_squares = np.sum((y_true - np.mean(y_true)) ** 2)
    r_squared = 1 - (residual_sum_of_squares / total_sum_of_squares)
    return r_squared
```

The **numpy_r_squared** function calculates the coefficient of determination (R-squared) using NumPy. The function takes two NumPy arrays as input: **y_true**, containing the actual values, and **y_pred**, containing the predicted values.

- It computes the residual sum of squares (RSS) using the formula:
- $RSS=\sum(y\text{true}-y\text{pred})^2$.

- It computes the total sum of squares (TSS) using the formula:
- $TSS=\sum(y\text{true}-\text{mean}(y\text{true}))^2$.
- It calculates the R-squared value using the formula: $R^2 = 1 - (RSS/TSS)$
- Finally, it returns the computed R-squared value.

The R-squared value obtained, $R^2 =0.2423$, indicates that approximately 24.23% of the variance in the actual stock prices is explained by the predicted stock prices.

There isn't a specific "ideal" value of R-squared (R^2) for stock market prediction using LSTM models or any other predictive modeling approach. The suitability of an R-squared value depends on various factors including the nature of the data, the complexity of the problem, and the specific requirements of the application.

However, in general, a higher R-squared value indicates a better fit of the model to the data, suggesting that a larger proportion of the variance in the dependent variable (e.g., stock prices) is explained by the independent variables (e.g., features used for prediction).

In the context of stock market prediction, achieving a high R-squared value can be challenging due to the inherent volatility and unpredictability of stock prices. Stock prices are influenced by numerous factors including economic conditions, market sentiment, geopolitical events, and company-specific news, among others. These factors can lead to high levels of noise and uncertainty in the data, making it difficult for models to accurately predict stock prices.

While it's desirable to achieve as high an R-squared value as possible, it's also important to consider the limitations of the data and the model, as well as the inherent uncertainty associated with stock market predictions.

Conclusion

In conclusion, this project demonstrates the application of LSTM networks in predicting stock market performance. By leveraging deep learning techniques and historical stock price data, we have developed a predictive model capable of forecasting future stock prices. While the model's accuracy may vary depending on the dataset and market conditions, the results indicate the potential of LSTM networks in stock market prediction. Further research and refinement of the model could lead to improved performance and broader applications in financial markets.

This report provides insights into the process of building a stock market prediction model using LSTM networks and serves as a foundation for future research in this area.

References

- TensorFlow Documentation
- Keras Documentation
- Scikit-learn Documentation
- Pandas Documentation
- Matplotlib Documentation