REPORT ON

PREDICTION OF STOCK PRICE

By:

Divya peram(522137)

Arkala chandhra shekar vinay(521486)

WHY WE CHOSE:

It's a realistic topic which is a very relevant case for traders which will help them interpret the market data and help them predict upcoming market trends or patterns and as traders like ourselves this might be useful for us to understand and learn more about the market and help us have useful insights

INTRODUCTION:

Stock price prediction is a crucial task for investors and traders. This report explores the application of Recurrent Neural Networks (RNNs) to predict stock prices using historical data. The dataset used for this analysis is sourced from online specifically the 'IndianBankNifty (2).csv' file.

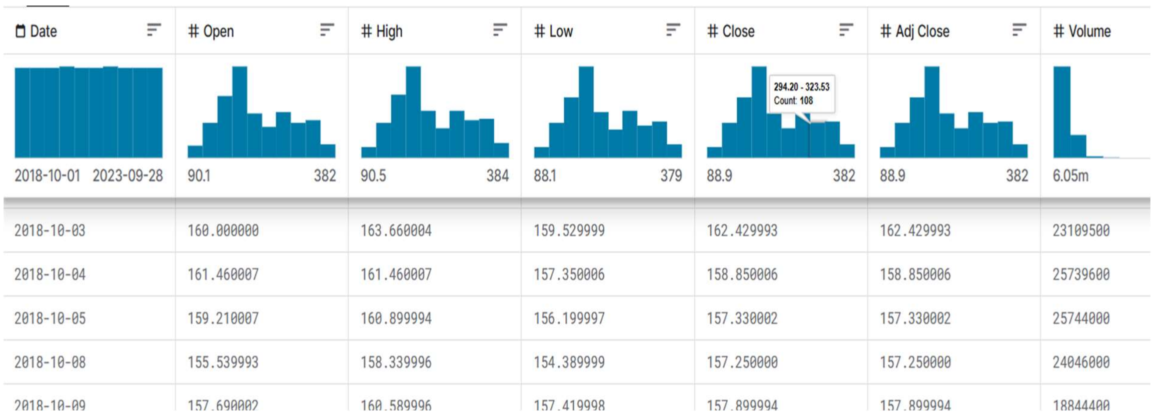
We have used the concept of neural networks to classify different stock premium:

Neural networks are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network. Neural networks rely on training data to learn and improve their accuracy over time. However, once these learning algorithms are fine-tuned for accuracy, they are powerful tools in computer science and artificial intelligence, allowing us to classify and cluster data at a high velocity.

DATASET OVERVIEW:

'The IndianBankNifty (2).csv' is a time series sequential model data and it's a historical data of a stock which contains their 'open' and 'close' price and 'high' and 'low' of the day and also volume and other important attributes of our data. These help us predict the unknown or upcoming data price or trends for our data . The 'Date' column is in datetime format, and the 'Open' price represents the opening stock price on a given day.

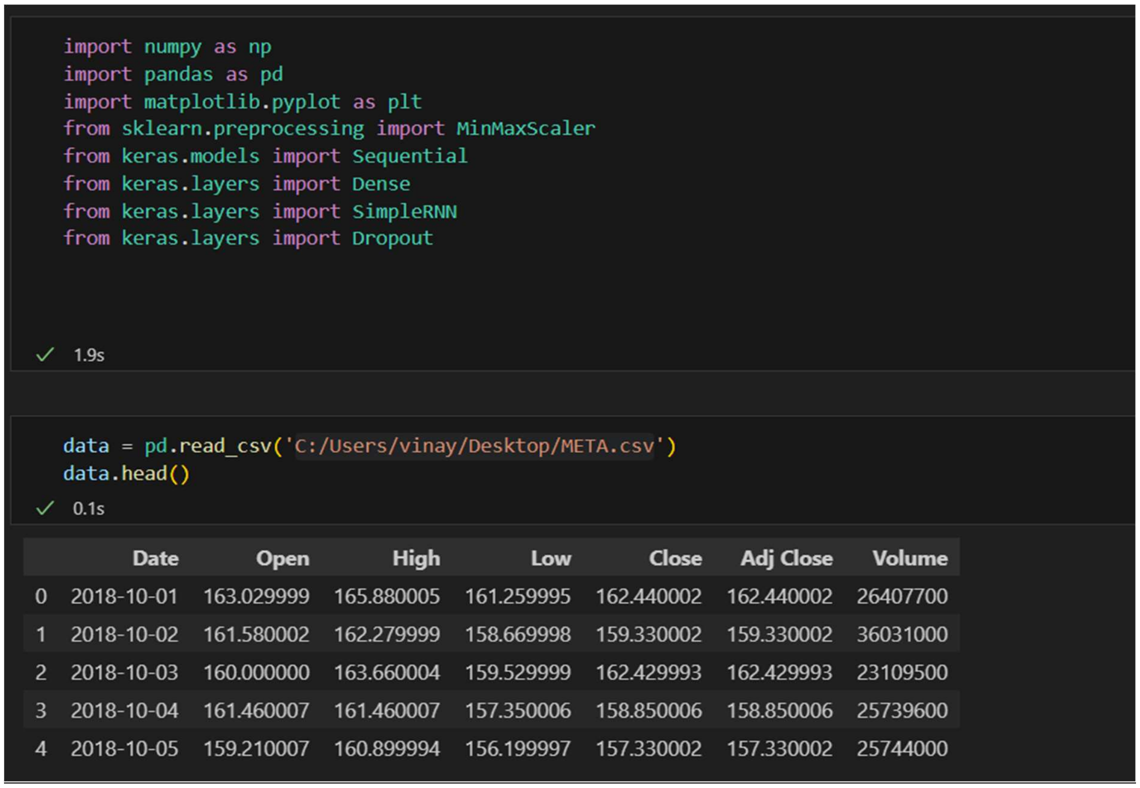
- Dataset Size: The dataset consists of a certain number of records, capturing daily stock information over a specified period.
- Columns:
- Date: Represents the date of the stock information.
- Open: Indicates the opening stock price on a given day.



Date	# Open	# High	# Low	# Close	# Adj Close	# Volume
2018-10-01 2023-09-28	90.1 382	90.5 384	88.1 379	88.9 382	88.9 382	6.05m
2018-10-03	160.000000	163.660004	159.529999	162.429993	162.429993	23109500
2018-10-04	161.460007	161.460007	157.350006	158.850006	158.850006	25739600
2018-10-05	159.210007	160.899994	156.199997	157.330002	157.330002	25744000
2018-10-08	155.539993	158.339996	154.389999	157.250000	157.250000	24046000
2018-10-09	157.690002	160.589996	157.419998	157.899994	157.899994	18844400

IMPORTING LIBRARIES:

We started by importing essential libraries and loading the stock data into a Pandas Data Frame. Key libraries include **NumPy** for numerical operations, **Pandas** for data manipulation, **Matplotlib** for data visualization, and **scikit-learn** for data preprocessing. we also imported **sequential** from keras models for stacking of neural networks and **dense** from keras layers is used as output and as hidden layer also **simple rnn** to capture sequential info since our data is a sequence time series and finally **dropout** is imported for its ability to prevent overfitting.



```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import SimpleRNN
from keras.layers import Dropout
```

✓ 1.9s

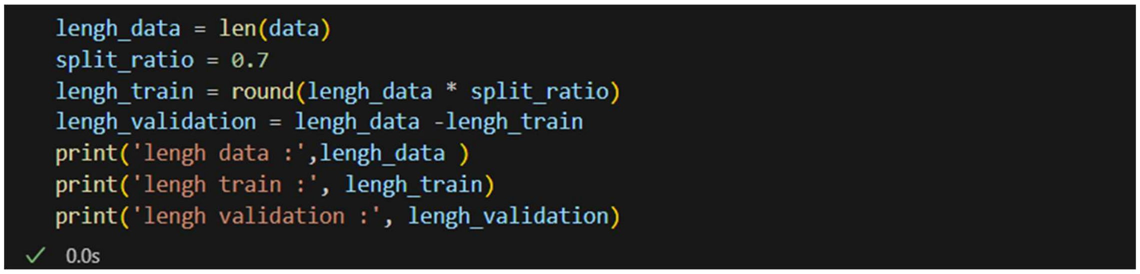
```
data = pd.read_csv('C:/Users/vinay/Desktop/META.csv')
data.head()
```

✓ 0.1s

```
         Date        Open        High         Low       Close   Adj Close    Volume
0  2018-10-01  163.029999  165.880005  161.259995  162.440002  162.440002  26407700
1  2018-10-02  161.580002  162.279999  158.669998  159.330002  159.330002  36031000
2  2018-10-03  160.000000  163.660004  159.529999  162.429993  162.429993  23109500
3  2018-10-04  161.460007  161.460007  157.350006  158.850006  158.850006  25739600
4  2018-10-05  159.210007  160.899994  156.199997  157.330002  157.330002  25744000
```

Keras is a high-level neural networks API developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research. Keras has the following key features: Allows the same code to run on CPU or on GPU, seamlessly. User-friendly API which makes it easy to quickly prototype deep learning models.Built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both. Supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, etc. This means that Keras is appropriate for building essentially any deep learning model, from a memory network to a neural Turing machine.

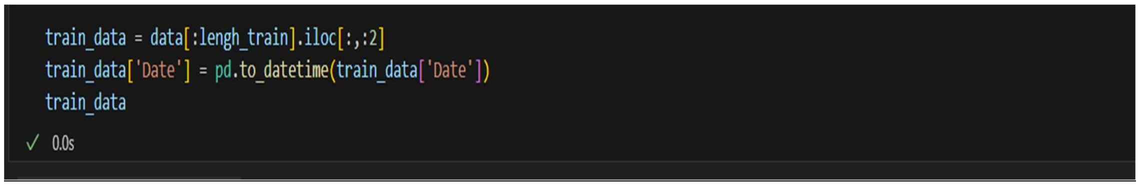
PREPROCESSING:



```
lengh_data = len(data)
split_ratio = 0.7
lengh_train = round(lengh_data * split_ratio)
lengh_validation = lengh_data -lengh_train
print('lengh data :',lengh_data )
print('lengh train :', lengh_train)
print('lengh validation :', lengh_validation)
✓ 0.0s
```

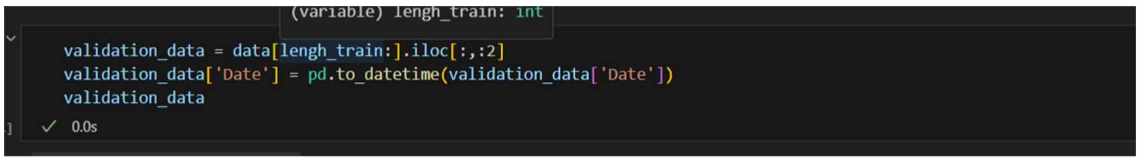
DATA SPLIT:

We then divided the dataset into training and testing (validation)sets with a 70:30 ratio based on a specified split ratio of 0.7.in which 70% is training or input data and 30% is testing or predicted data. The length of the training set is determined by this ratio. our training data is termed as **train data** and testing is termed as **validation data.** For training and testing we used first two columns of the data open and date and changed date into date time for easy time based calculations.



```
train_data = data[:lengh_train].iloc[:,:2]
train_data['Date'] = pd.to_datetime(train_data['Date'])
train_data
✓ 0.0s
```

VALIDATION: a validation set is created (validation_data) to evaluate the model's performance on unseen data. The 'Date' column in the validation set is converted to datetime for time-based analysis. During training, the model is assessed on both the training and validation sets to ensure generalization. This process enhances the model's robustness and helps identify potential overfitting. Including a validation set is crucial for unbiased model evaluation and reliable stock price predictions on new, unseen data.



```
(variable) lengh_train: int
validation_data = data[lengh_train:].iloc[:,:2]
validation_data['Date'] = pd.to_datetime(validation_data['Date'])
validation_data
✓ 0.0s
```

RESHAPING:

To ensure data is in a format compatible with expected input shape reshaping is used to transform the input data into a format that is compatible with the input requirements of the neural network mode.

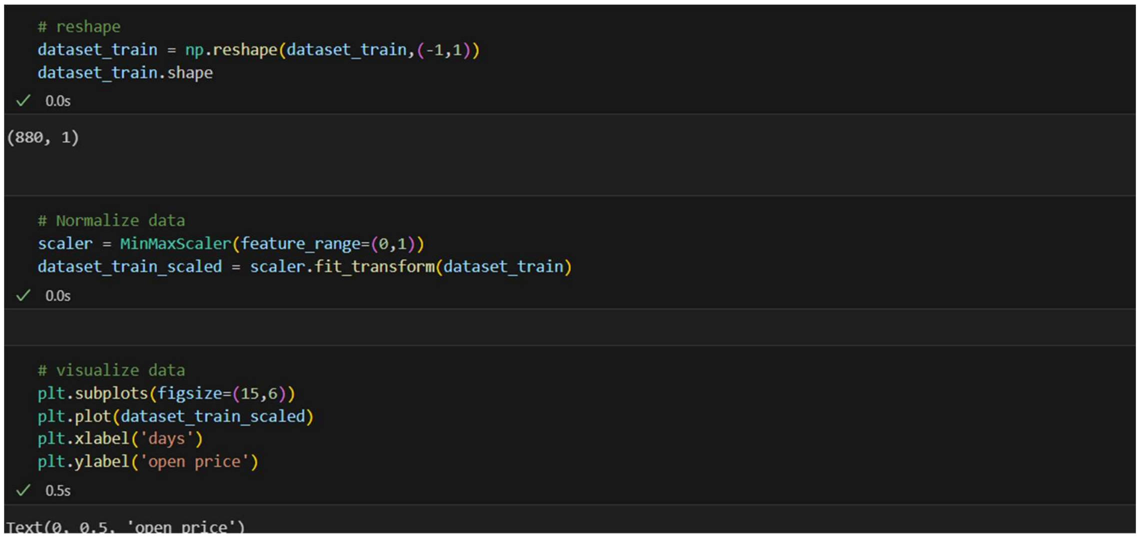
The 'Open' prices (dataset train) are initially a 1D array. The np.reshape() function is used to convert it into a 2D array with one column. The -1 in the reshaping operation is a placeholder that automatically calculates the size of one dimension based on the size of the other, ensuring that the total number of elements remains constant.

DATA NORMALIZATION:

Normalization is used to rescale the data so that all values fall within the specific range, min max scaler feature is use do to set data into a range of [0,1] ensures all features have similar scale .

Fit transform:

it’s a method to compute the scaling parameters based on training data and transform ‘dataset train’ array to scaled. The transformed data is now within the range of [0,1]



```
# reshape
dataset_train = np.reshape(dataset_train,(-1,1))
dataset_train.shape
```

```
✓ 0.0s
(880, 1)
```

```
# Normalize data
scaler = MinMaxScaler(feature_range=(0,1))
dataset_train_scaled = scaler.fit_transform(dataset_train)
```

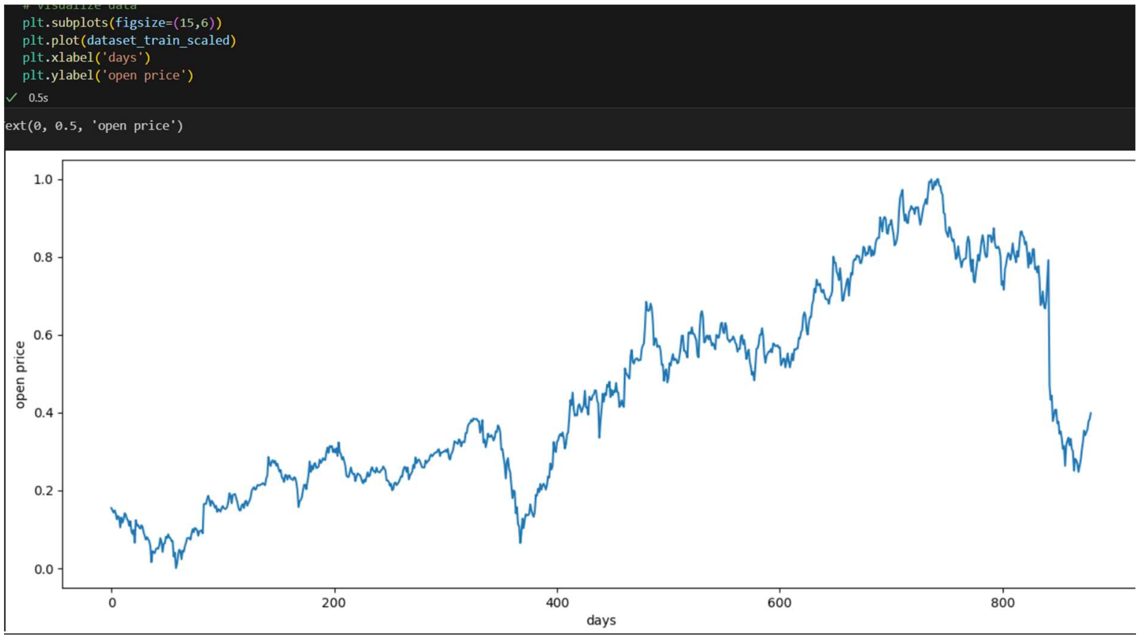
```
✓ 0.0s
```

```
# visualize data
plt.subplots(figsize=(15,6))
plt.plot(dataset_train_scaled)
plt.xlabel('days')
plt.ylabel('open price')
```

```
✓ 0.5s
Text(0, 0.5, 'open price')
```

visualization:

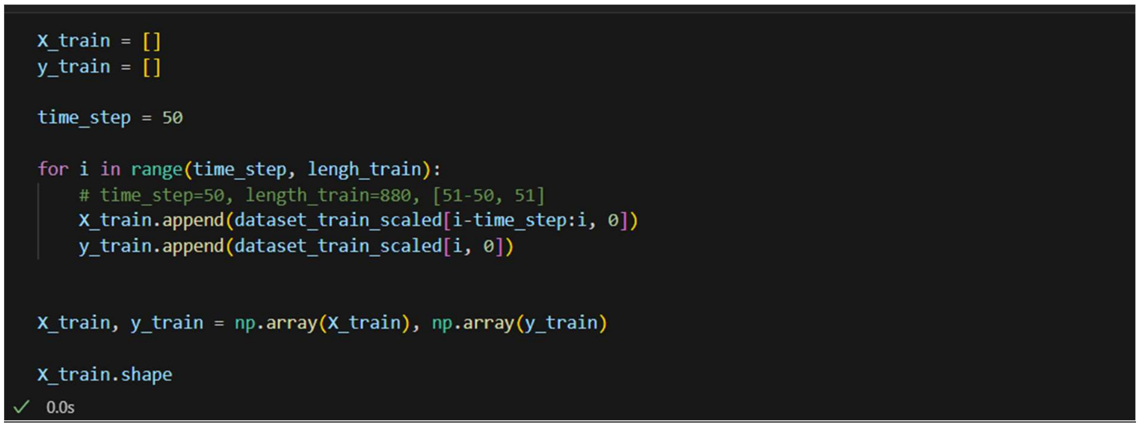
The normalized data is the visualized using Matplotlib. Visualizing the normalized data provides insights into the trends.



```
plt.subplots(figsize=(15,6))
plt.plot(dataset_train_scaled)
plt.xlabel('days')
plt.ylabel('open price')
```

```
ext(0, 0.5, 'open price')
```

Train data:



```
X_train = []
y_train = []

time_step = 50

for i in range(time_step, lengh_train):
    # time_step=50, length_train=880, [51-50, 51]
    X_train.append(dataset_train_scaled[i-time_step:i, 0])
    y_train.append(dataset_train_scaled[i, 0])


X_train, y_train = np.array(X_train), np.array(y_train)

X_train.shape
```

the training set is represented by the variable X_train. here X_train is a list that contains sequences of time_step consecutive data points from the normalized training data (dataset_train_scaled). Each sequence in X_train serves as an input sample for training the neural network.

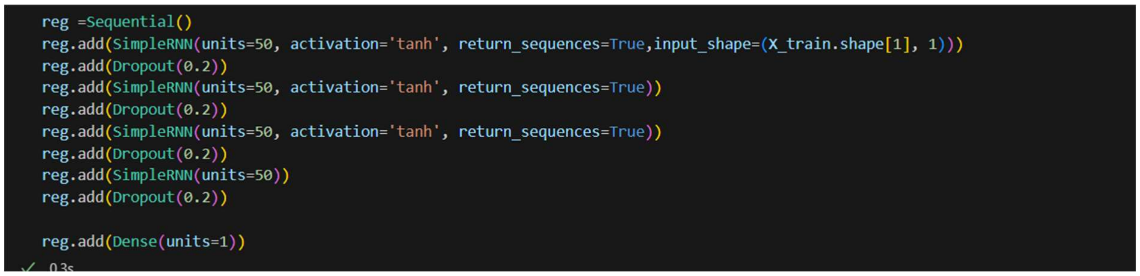
The for loop iterates over the range from time_step to lengh_train, and for each iteration, it appends a sequence of time_step data points to X_train. These sequences are created by taking a sliding window of size time_step over the normalized training data.

The fit method is used to train the Simple RNN model (reg).

The input data (X_train) and target labels (y_train) are provided.

<u>MODEL ARCHITECTURE:</u>

The model architecture consists of a Simple Recurrent Neural Network (RNN) and a Long Short-Term Memory (LSTM) network. Below is an overview of the model architecture for both the Simple RNN and the LSTM:



```
reg =Sequential()
reg.add(SimpleRNN(units=50, activation='tanh', return_sequences=True,input_shape=(X_train.shape[1], 1)))
reg.add(Dropout(0.2))
reg.add(SimpleRNN(units=50, activation='tanh', return_sequences=True))
reg.add(Dropout(0.2))
reg.add(SimpleRNN(units=50, activation='tanh', return_sequences=True))
reg.add(Dropout(0.2))
reg.add(SimpleRNN(units=50))
reg.add(Dropout(0.2))

reg.add(Dense(units=1))
✓ 0.3s
```

Simple RNN Model: The model starts with a Simple RNN layer with 50 units, using the hyperbolic tangent (tanh) activation function, and returns sequences for subsequent layers.

Tanh activation function

The hyperbolic tangent function, often denoted as tanh, is a common activation function used in neural networks

<u>*Range:*</u> The tanh function maps its input to the range [-1, 1], which makes it convenient for situations where you want to emphasize strong negative or positive values. It's centered around 0, making it zero-centered, which can help the learning dynamics of the model.

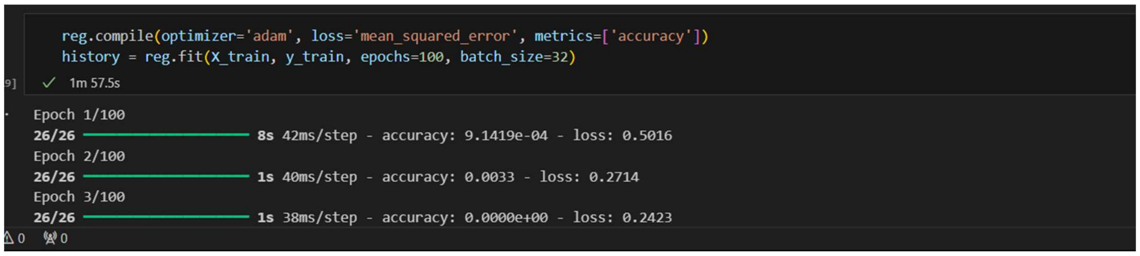
<u>*Non-linearity*</u>: Tanh introduces non-linearity into the network, allowing it to learn complex patterns and relationships in the data.

In neural networks, the tanh activation function is often used in hidden layers, especially in recurrent neural networks (RNNs) and long short-term memory networks (LSTMs). However, in recent years, the Rectified Linear Unit (ReLU) and its variants have gained popularity due to their simplicity and better performance in many situations.

Dropout layers are added after each RNN layer to prevent overfitting.

A Sequential model with multiple SimpleRNN layers is constructed. Dropout layers are included to prevent overfitting.

The final layer is a Dense layer with one unit, which is the output layer for regression tasks.



```
reg.compile(optimizer='adam', loss='mean_squared_error', metrics=['accuracy'])
history = reg.fit(X_train, y_train, epochs=100, batch_size=32)
✓ 1m 57.5s

Epoch 1/100
26/26 ━━━━━━━━━━━━━━━━━━━━ 8s 42ms/step - accuracy: 9.1419e-04 - loss: 0.5016
Epoch 2/100
26/26 ━━━━━━━━━━━━━━━━━━━━ 1s 40ms/step - accuracy: 0.0033 - loss: 0.2714
Epoch 3/100
26/26 ━━━━━━━━━━━━━━━━━━━━ 1s 38ms/step - accuracy: 0.0000e+00 - loss: 0.2423
```

Model Compilation:

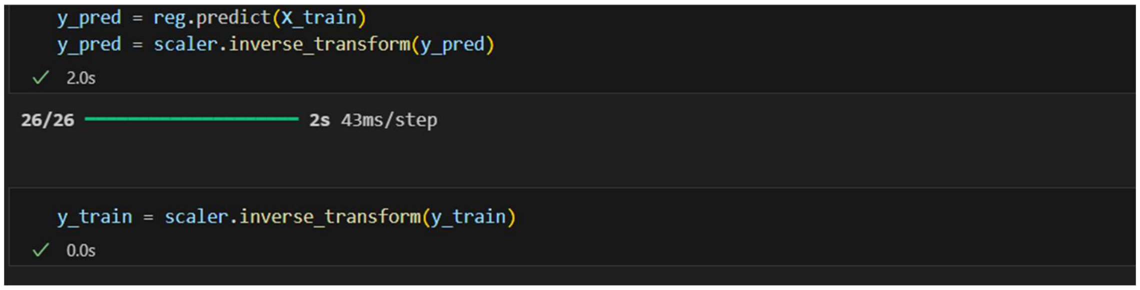
The compilation involves specifying the optimizer, loss function, and evaluation metric.

These models are trained using historical stock prices and validated using a separate validation set. The choice of model architecture, dropout layers, and hyperparameters may be further optimized based on performance metrics and experimentation.

Optimizer ADAM

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data. Adam was presented by Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto in their 2015 ICLR paper (poster) titled “Adam: A Method for Stochastic Optimization“. I will quote liberally from their paper in this post, unless stated otherwise. The algorithm is che name Adam is derived from adaptive moment estimation.When introducing the algorithm, the authors list the attractive benefits of using Adam on non-convex optimization problems, as follows:alled Adam. It is not an acronym and is not written as “ADAM”.

Straightforward to implement. Computationally efficient Little memory requirements. Invariant to diagonal rescale of the gradients. Well suited for problems that are large in terms of data and/or parameters. Appropriate for non-stationary objectives. Appropriate for problems with very noisy/or sparse gradients. Hyper-parameters have intuitive interpretation and typically require little tuning



```
y_pred = reg.predict(X_train)
y_pred = scaler.inverse_transform(y_pred)
✓ 2.0s

26/26 ━━━━━━━━━━━━━━━━━━━━ 2s 43ms/step

y_train = scaler.inverse_transform(y_train)
✓ 0.0s
```

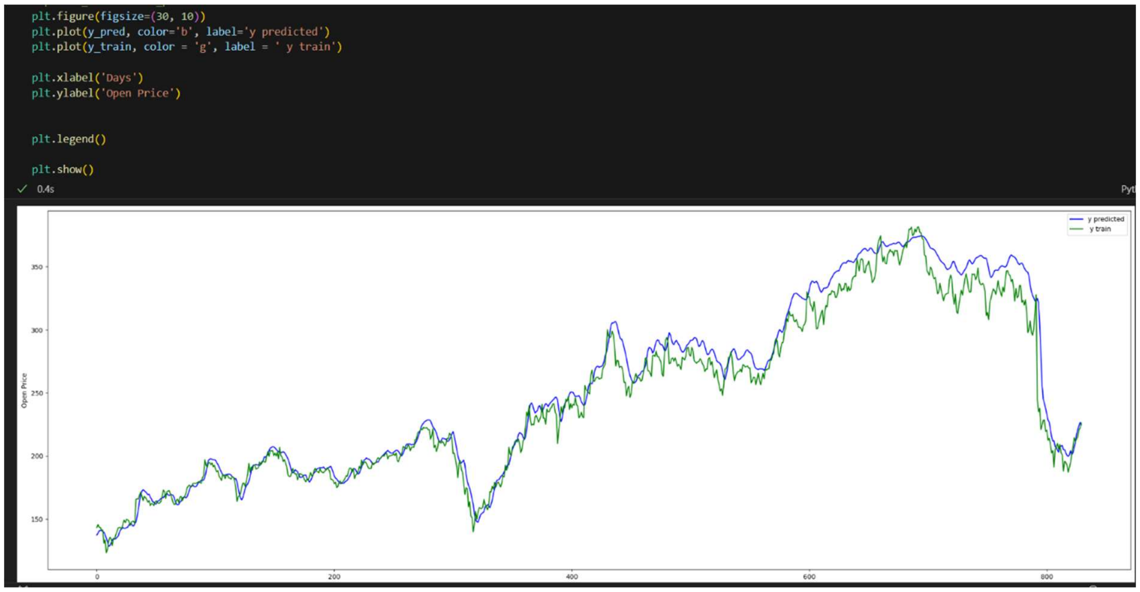
reg.predict(X_train): This line predicts the target variable using the regression model reg on the features X_train. The predict() method takes the features X_train as input and returns the predicted values of the target variable.

y_pred = scaler.inverse_transform(y_pred): This line inversely transforms the predicted values y_pred using the inverse_transform() method of the scaler scaler. This step is important if the original target variable was transformed or scaled before training the model. It brings the predicted values back to their original scale or form.

scaler.inverse_transform(y_train): This line uses the inverse_transform() method of the scaler object (scaler) to revert the scaled or transformed values of the target variable y_train back to their original scale or form.

After this operation, y_train should contain the original values of the target variable, which were scaled or transformed during the preprocessing stage to make them suitable for model training.

It's important to ensure that the scaler object scaler used for inverse transformation has been fitted to the training data and that its parameters are appropriate for the transformation and inverse transformation operations. This helps maintain consistency in the scaling or transformation process across the training and testing datasets.



```
plt.figure(figsize=(30, 10))
plt.plot(y_pred, color='b', label='y predicted')
plt.plot(y_train, color = 'g', label = ' y train')

plt.xlabel('Days')
plt.ylabel('Open Price')


plt.legend()

plt.show()
```

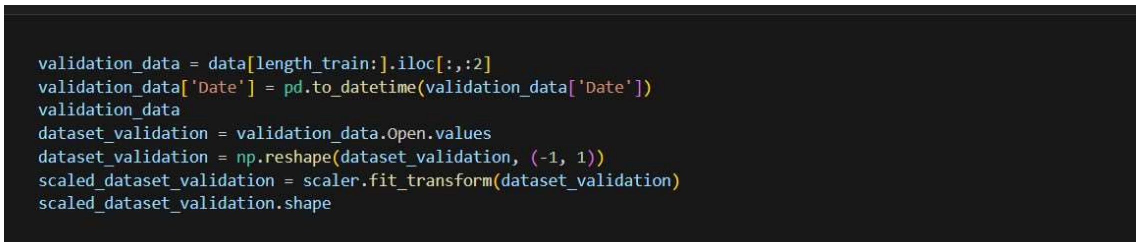
```
✓ 0.4s
```

plt.plot(y_pred, color='b', label='y predicted'): This line plots the predicted values (y_pred) using a blue line ('b') and adds a label 'y predicted'.

plt.plot(y_train, color='g', label='y train'): This line plots the actual target values (y_train) using a green line ('g') and adds a label 'y train'.

plt.xlabel('Days') and plt.ylabel('Open Price'): These lines set labels for the x-axis and y-axis, respectively.

Test data(validation):

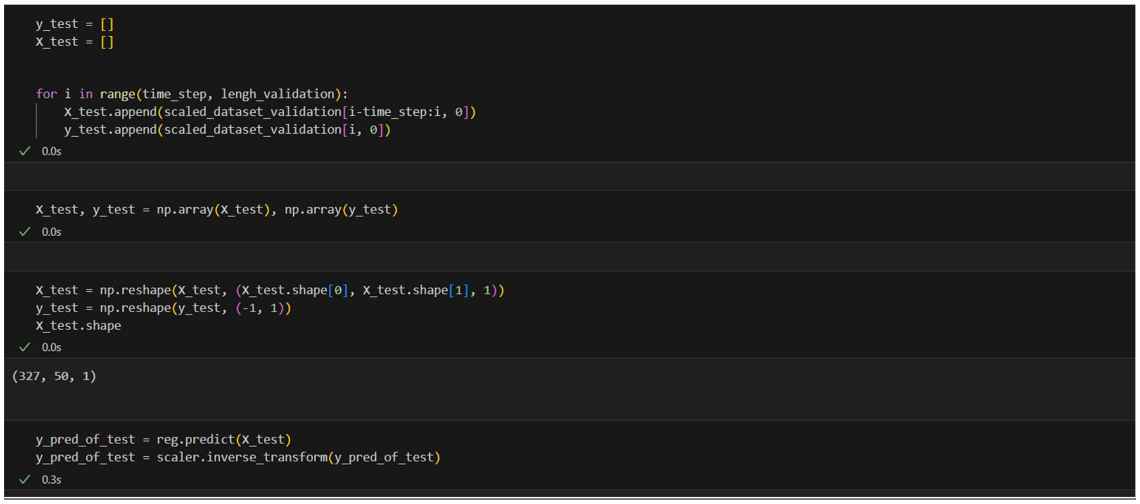


```
validation_data = data[length_train:].iloc[:,:2]
validation_data['Date'] = pd.to_datetime(validation_data['Date'])
validation_data
dataset_validation = validation_data.Open.values
dataset_validation = np.reshape(dataset_validation, (-1, 1))
scaled_dataset_validation = scaler.fit_transform(dataset_validation)
scaled_dataset_validation.shape
```

the 'Open' column from the validation_data, converts it to a NumPy array, and then reshapes it into a column vector using np.reshape(). This reshaping is commonly done to ensure that the array has the correct shape for further processing.

the dataset_validation using the fit_transform() method of the scaler scaler. It scales the data based on the parameters learned from the validation set. Finally, it prints the shape of the scaled dataset.

The shape of scaled_dataset_validation would reflect the number of rows and columns after scaling, typically preserving the number of rows but modifying the number of columns based on the transformation applied by the scaler.



```
y_test = []
X_test = []


for i in range(time_step, lengh_validation):
    X_test.append(scaled_dataset_validation[i-time_step:i, 0])
    y_test.append(scaled_dataset_validation[i, 0])
✓ 0.0s

X_test, y_test = np.array(X_test), np.array(y_test)
✓ 0.0s

X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
y_test = np.reshape(y_test, (-1, 1))
X_test.shape
✓ 0.0s
(327, 50, 1)

y_pred_of_test = reg.predict(X_test)
y_pred_of_test = scaler.inverse_transform(y_pred_of_test)
✓ 0.3s
```

This loop generates input-output pairs (X_test and "y_test*) for the validation data, similar to what was done for the training data.

For each iteration, it extracts a sequence of time_step* consecutive values from the scaled validation data (X_test). The corresponding next value is appended to 'y_test'.

The steps ensure that the validation data undergoes the same preprocessing steps (reshaping and normalization) as the training data for consistency. Generating sequences of input-output pairs prepares the validation data for evaluation, allowing you to assess the model's performance on data it has not seen during training.

np.array(X_test) and np.array(y_test): Converts the lists 'X_test and y test to NumPy arrays.

np-reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))*:

Reshapes the input sequences (X_test) to have the required shape for feeding into the neural network. The added dimension of size 1 represents the feature dimension.

Reshaping is crucial to match the expected input and output shapes of the neural network model. It ensures that the data has the correct dimensions for prediction.

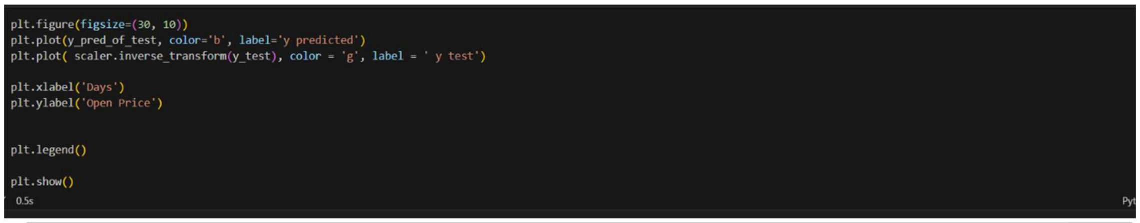
reg-predict(X_ test) : Uses the trained neural network ('reg*) to make predictions on the validation data ('X_test*).

This step produces the model's predictions for the validation set, allowing you to assess how well the model generalizes to unseen data.

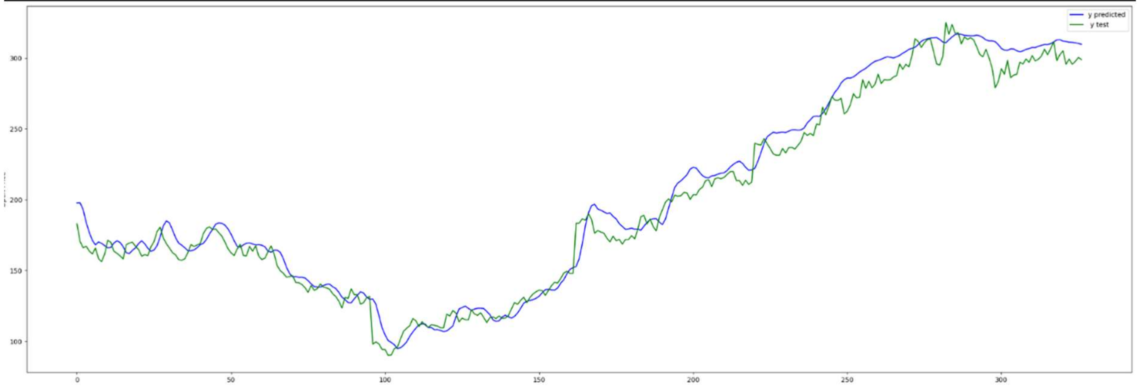
scaler-inverse_transformy_pred_of_test)*: Inverts the earlier normalization transformation on the predicted values (y_pred_of_test'). It converts the scaled predictions back to the original

scale of the data.

Visualizing testing and prediction data :

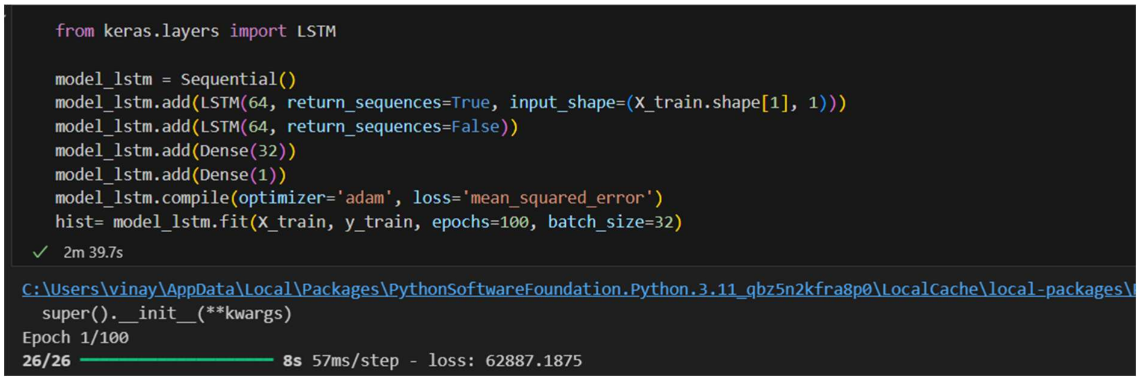


```
plt.figure(figsize=(30, 10))
plt.plot(y_pred_of_test, color='b', label='y predicted')
plt.plot( scaler.inverse_transform(y_test), color = 'g', label = ' y test')

plt.xlabel('Days')
plt.ylabel('Open Price')


plt.legend()

plt.show()
 0.5s
```



This plot visually compares the predicted open prices (Predicted Open Price (Validation) in blue) generated by the Simple RNN model with the actual open prices from the validation data (Actual Open Price (Validation) in green)

LSTM Model:



```
from keras.layers import LSTM

model_lstm = Sequential()
model_lstm.add(LSTM(64, return_sequences=True, input_shape=(X_train.shape[1], 1)))
model_lstm.add(LSTM(64, return_sequences=False))
model_lstm.add(Dense(32))
model_lstm.add(Dense(1))
model_lstm.compile(optimizer='adam', loss='mean_squared_error')
hist= model_lstm.fit(X_train, y_train, epochs=100, batch_size=32)
✓ 2m 39.7s

C:\Users\vinay\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\
  super().__init__(**kwargs)
Epoch 1/100
26/26 ━━━━━━━━━━━━━━━━━━━━ 8s 57ms/step - loss: 62887.1875
```

LSTMs are a type of recurrent neural network (RNN) designed to handle sequences of data

The LSTM layers, along with dense layers, define the architecture of the model. The choice of hyperparametegs such as the number of units in the LSTM layers and dense layers is based on experimentation and problem requirements.

model_lstm = Sequential ()*: Initializes a sequential model.

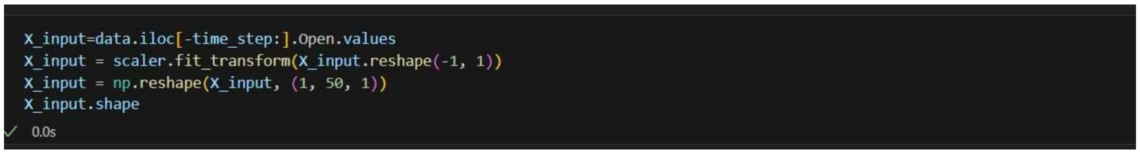
model lstm-add (LSTM64, return_sequences-True, input_ shape= (X_train-shape[1], 1)))*: Adds the first LSTM layer to the model with 64 units, returns sequences for subsequent layers, and specifies the input shape. Adds a second LSTM layer with 64 units, but this layer doesn't return sequences (it returns the output

This step evaluates how well the LSTM model can make predictions on data that it hasn't encountered during training

model, lstm-add (Dense (32)) *: Adds a dense (fully connected) layer with 32 units.

model, lstm. add (Dense (1)) *: Adds the final dense layer with 1 unit, representing the output of the model.

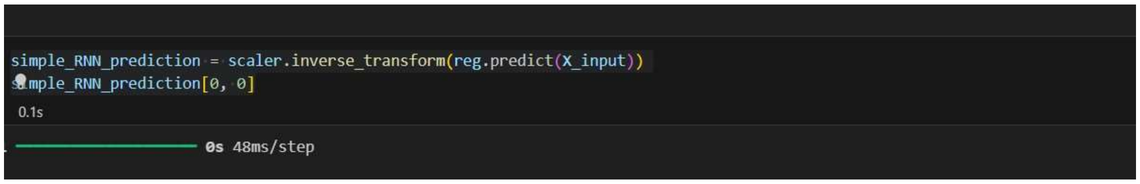
The 'hist* variable stores the training history, including loss values at each epoch. This information can be used for analysing the model's performance over time



```
X_input=data.iloc[-time_step:].Open.values
X_input = scaler.fit_transform(X_input.reshape(-1, 1))
X_input = np.reshape(X_input, (1, 50, 1))
X_input.shape
✓ 0.0s
```

Extracting input data: Extracts the most recent 'time step' number of 'Open' values from the 'data' Data Frame. This step selects a portion of the time series data to be used as input for making predictions. The length of this sequence is determined by the time step variable.

Scaling input data: The MinMaxScaler to normalize the input data. The 'fit transform' method both fits the scaler to the data and transforms the data to the scaled version.



```
simple_RNN_prediction = scaler.inverse_transform(reg.predict(X_input))
simple_RNN_prediction[0, 0]
0.1s
━━━━━━━━━━━━━━━━━━━━ 0s 48ms/step
```

reg-predict(X_input) : Uses the trained neural network (reg) to make predictions on the prepared input data (X_input).scaler. inverse transform(): Applies the inverse transformation to convert the scaled predictions back to the original scale of the data.

simple_RNN_prediction[0, 0]: Accesses the first element of the first row in the 'simple_RNN_pediction' array

This retrieves the predicted Open value for the next day after performing the inverse transformation.

CONCLUSION :

Simple RNN and LSTM models for stock price prediction, with a focus on visual evaluation. The models undergo training and testing phases, emphasizing qualitative assessments through plotted comparisons of predicted and actual prices.

In conclusion, the implemented code showcases the application of recurrent neural network (RNN) models, specifically a Simple RNN and a Long Short-Term Memory (LSTM) network, for predicting stock prices. The code efficiently loads and preprocesses historical stock price data, focusing on the 'Open' prices and normalizing them for model training. Both models are trained and evaluated with an emphasis on visual inspection of predicted and actual stock prices.