STATS 315 Final Report
Author: Xinyang Zhou

1. Introduction

As the world's leading investment bank, the Goldman Sachs Group is famous for its wide range of services provided in the financial sector. Starting from May 4, 1999, its stock became public, with an initial price of 76 USD per share. Today, its stock is selling at 359 USD per share.

We observed a big rise in the stock price and would like to use deep learning models to predict the stock price. To be more precise, we will use the closing price as our input, since the closing price is more informative than the opening price. In this project, we will use the first 70% of the data as our training data to help build the models, and the rest 30% of the data to test our model. We want to select a best model to predict the Goldman Sachs closing stock price with the minimum loss and the maximum efficiency.

2. Dataset

This dataset is obtained from the U.S. Securities and Exchange Commission (SEC), which provides historical data of The Goldman Sachs Group, Inc. (GS). The data is available at a daily level. Currency used is US Dollar. The range of prices collected is from May 4, 1999 to recently: March 24, 2022, with a final closing price of 336.23 USD.

In this dataset, we have seven columns: "Date", "Open", "High", "Low", "Close", "Adj Close", and "Volume". "Open" and "Close" indicates the price from the first and the last transaction of a trading day. "High" and "Low" represent the maximum and minimum prices on a trading day. "Adj Close" represents the closing price adjusted to reflect the value after accounting for any corporate actions. "Volume" represents the number of units traded in a day.

We first want to see the actual dataset, with some example rows and columns (Table 1, Appendix). Then we want to get an overall impression of the trends (Figure 1, Appendix). We observe the stock price reaches a local maximum at time 2000. We also observe a tremendous drop right after this boom, as depicted around time 2500. Then, we see the stock price have ups and downs, reaching the peak at time around 5500.

We have 5762 rows in total, which means we have 5762 closing values. If we divide this number by 365 days per day, we arrive at around 15.78, which is less than 16 years. Since the beginning date in the dataset is in the year 1999, and the ending date is in the year 2022, we conclude that some of the entries are missing due to unforeseen problems like the financial crisis and the stock market crash.

We want to explore the opening stock prices as well to make sure there is no obvious difference in patterns compared to the closing prices. We generate the opening price plot and conclude that the general patterns matched (Figure 2, Appendix).

3. Method

First of all, we need to do transformations on the closing prices so that we can manipulate them. In this project, we will use the first 70% of the dataset as our training data and the rest as our testing data. Therefore, we separate the closing prices into two parts accordingly.

We will then start off with the first model. We will use LSTM as our main layer, as it is designed for sequential input data with some underlying dependencies over a long period of time. The model starts with an LSTM layer with 50 units of input, set return_sequences to

“True” as we want to explore some inner connections, and specify the input shape appropriately. Then we add the same layer with the same number of units two more times to maximize its performance. In the end, we add a Dense layer with unit 1 to get a single output. We compile the model using the mean squared error loss function and the “adam” optimizer first. Then we modified the model a little by changing the loss function to binary cross-entropy. Then we can evaluate the performance of these two models by finding their accuracy. We want to pick a better-performed model and visualize our stock price prediction.

We then decrease the number of units in each of the LSTM layers to see the effects on the overall accuracy. Now we set the number of units to be 10 and we evaluate its performance.

Besides using LSTM layers, we want to check the effectiveness of the SimpleRNN layer. We add a single SimpleRNN layer with 50 units of input and a ‘relu’ activation function. Similarly, we want to use the “mean_squared_error” loss function and the “adam” optimizer. Then we evaluate the model and print out the accuracy. We want to modify this model again by reducing significantly on the units. This time, we will only have 5 units.

After dealing with LSTM and SimpleRNN, the two “appropriate” layer types for a stock prediction case, we want to see how other layers perform. We first build our model using Dense layers. We will have 50 units in the first Dense layer, 10 units in the next two Dense layers, and a final Dense layer with 1 unit as output.

Then we want to implement a non-sequential model. For our first non-sequential model, we add a SimpleRNN with 64 units of input at the beginning followed by two Dense layers with 20 units each. Under the call function, we create this model as a binomial shape, meaning that we have two routes for each call. In the end, the function will return the sum of two weighted “x” as the final input to the last Dense layer. We also do a small change to the model by replacing the SimpleRNN layer with a 1D convolution layer. We evaluate the accuracy of these two models and do comparisons.

In the end, we want to fit a linear regression model to the closing price as our baseline approach and observe interesting features. We will use “High”, “Low”, “Open”, and “Volume” as our predictors, and “Close” as our response. By applying the appropriate function, we should be able to find the coefficients of each predictor and the predicted stock price for all testing data points. We calculate the percentage of the predicted stock price that is within ± 1 USD range compared to the real stock price at a given time. In the end, we want to visualize the performance of our linear regression predicted values.

4. Experiments

For our first model using LSTM layers only, our result is convincing (Figure 3, Appendix). We observe that the stock price predictions (red curve) based on the train data (yellow curve) are aligned closely with the real data points and trends (blue curve). We observe the only notable difference happens when the stock price reaches the global maximum at the time around 5500. The real stock price is much higher than our predicted price at that point. The maximum difference can be as large as 20 USD per share. But in general, this model predicts the stock price well and we consider it a good model.

We also fit a similar model with a different loss function. If we use “binary_crossentropy” instead of “mean_squared_error”, we arrive at a much larger loss and hence less accurate.

Based on our experiments, the loss for “mean_squared_error” is 0.0008869, whereas the loss for “binary_crossentropy” is 0.6187, which is much bigger.

Then we try to fit a model with decreased number of units in each LSTM layer and the “mean_squared_error” loss function. We set the unit to be 10. We fit and evaluate the model. The loss for this model is 0.0009782. We observe that it is slightly higher than the model with 50 units in each layer. We conclude that the decrease in the number of units can potentially affect the performance of a given model negatively. In our case, the negative impact is minuscule and can be ignored. We believe this is also a good model for our dataset.

Now we want to fit a model using different layers. We want to try SimpleRNN at this time. We add a single SimpleRNN layer with 50 units, a “mean_squared_error” loss function, and an “adam” optimizer to fit the model. After evaluation, we get a loss of 0.0001764. This is the simplest model, but it also has the smallest loss so far.

We want to confirm our previous finding by decreasing the number of units tremendously. Instead of 10, this time we will use only 5 units. We fit the SimpleRNN model once again with this modification only. We evaluate the model and arrive at a loss of 0.02219. Compared to the 50 units case, this loss function is much larger. Hence, we want to use a relatively larger number of units in the layers, if time allowed.

We visualize the SimpleRNN model with 50 units (Figure 4, Appendix). This time, we observe that almost all the predicted stock prices (red curve) and the underlying real stock prices (blue curve) match, including the interval around the global maximum. This is a better prediction compared to the very first model we have.

We move forward to use some less intuitive layers, the Dense layer. This is the most common layer in Deep Learning, and we want to see the performance of our Dense layer in the stock price prediction scenarios. The structure of this model is described in the previous section. After fitting, we evaluate the model and get the loss of 0.0003098. By comparison, we observe that this model is better performed than the LSTM model, getting the second place under the SimpleRNN model. We visualize the prediction (Figure 5, Appendix). We conclude that this is also a well-fitted model because there is no obvious mismatch compared to the underlying real stock price (blue curve).

We also want to explore how well the non-sequential models work for our scenario. After fitting the model, we get a loss of 0.0004539. This relatively low-level loss is anticipated, as the first layer we use is SimpleRNN. We now replace it with a 1D convolutional layer. After fitting the model, we get a loss of 0.0007455, which is higher. We therefore gather all the information for our neural network models and create the table:

Model	**Loss**
LSTM layers; 50 units; loss = 'mean_squared_error'	0.0008869
LSTM layers; 50 units; loss = 'binary_crossentropy'	0.6187
LSTM layers; 10 units; loss = 'mean_squared_error'	0.0009782
SimpleRNN; 50 units; loss = 'mean_squared_error'	0.0001764
SimpleRNN; 5 units; loss = 'mean_squared_error'	0.02219
Dense; 50, 10, 10 units; loss = 'mean_squared_error'	0.0003098
Non-Sequential with SimpleRNN(64)	0.0004539

Non-Sequential with conv1D(64)	0.0007455

Table 1

Our final approach is linear regression. The coefficients we get for “High”, “Low”, “Open”, and “Volume” are 7.95954956e-01, 7.92577066e-01, -5.88396122e-01, and 1.16758272e-08 respectively. The intercept we get is -0.06228. We can also make a table to see the difference between actual and predicted stock prices (Table 2, Appendix). We also calculate the percentage of close predictions. We observe that only 21% of the stock prices are within the real stock price$\pm$1 USD range. We also visualize our predicted stock prices (Figure 6, Appendix), where the blue line represents our prediction and the red line is the underlying real stock prices. We observe that the prediction stock prices go ups and downs continuously with unsteadiness. Therefore, we conclude that linear regression is not a good approach to our goal.

5. Conclusion / Future Work

From table 1, we conclude that “mean_squared_error” is the best loss function to use in the stock prediction cases compared to “binary_crossentropy”. With a significant drop in units, the performance of our models will be negatively affected. LSTM, SimpleRNN layer, Dense layers, and non-sequential model all performed well in the experiment, under the proper loss function and number of units. A SimpleRNN layer with 50 units is the model with the lowest loss. It can also be executed quickly. In addition, the code for this model is the simplest, which can save a lot of time comparing to other models. Therefore, we conclude that the model using a SimpleRNN layer with loss function “mean_squared_error” and optimizer “adam” is the best choice for our Goldman Sachs closing stock price prediction purpose.

By doing this project, I learned that the complicated model is not always the best. On the contrary, the best model we selected in the project is the simplest – only one SimpleRNN layer and a Dense layer. The shortcoming of my work is that I didn’t have enough time to apply more different loss functions to my models and monitor the effects. If I have more resources, I want to try some extreme cases, for example, if the number of units becomes very large, what will happen to the loss? Is it always better to have a smaller loss? These questions can be addressed through rigorous testing procedures and I look forward to completing it as well in the near future.

Appendix:

```
import numpy as np
from numpy import array
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
import os
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, SimpleRNN
from tensorflow.keras.layers import LSTM, Conv2D, Flatten

import math
from sklearn.metrics import mean_squared_error
```

```
from google.colab import drive
drive.mount('/content/drive')
gs = pd.read_csv('/content/drive/MyDrive/STATS 315 project/Goldman Sachs Stock.csv')
gs
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dr
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	1999-05-04	76.000000	77.250000	70.000000	70.375000	53.576797	22320900
1	1999-05-05	69.875000	69.875000	66.250000	69.125000	52.625153	7565700
2	1999-05-06	68.000000	69.375000	67.062500	67.937500	51.721100	2905700
3	1999-05-07	67.937500	74.875000	66.750000	74.125000	56.431648	4862300
4	1999-05-10	73.375000	73.500000	70.250000	70.687500	53.814709	2589400
...	...	...	...	...	...	...	...
5757	2022-03-18	338.869995	346.769989	337.299988	345.380005	345.380005	5861100
5758	2022-03-21	345.260010	346.299988	337.149994	339.000000	339.000000	3401200
5759	2022-03-22	342.200012	346.239990	340.119995	343.010010	343.010010	2840200
5760	2022-03-23	340.000000	340.829987	335.130005	335.609985	335.609985	2196800
5761	2022-03-24	336.440002	337.500000	334.299988	336.230011	336.230011	1943600

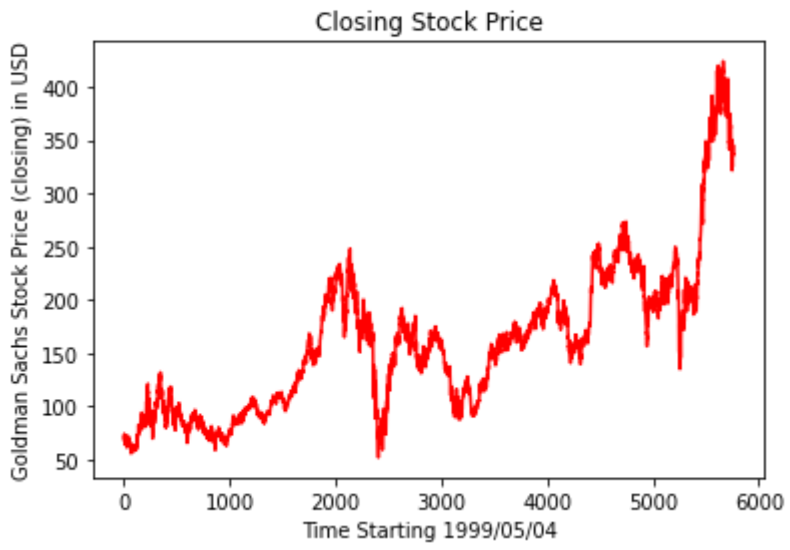
5762 rows × 7 columns

* Table 1 *

```
closing = gs.reset_index()['Close']
```

```
closing
plt.plot(closing, color = "red")
plt.xlabel("Time Starting 1999/05/04")
plt.ylabel("Goldman Sachs Stock Price (closing) in USD")
plt.title("Closing Stock Price")
figure(figsize=(8,6), dpi = 100)
```

```
<Figure size 800x600 with 0 Axes>
```



```
<Figure size 800x600 with 0 Axes>
```

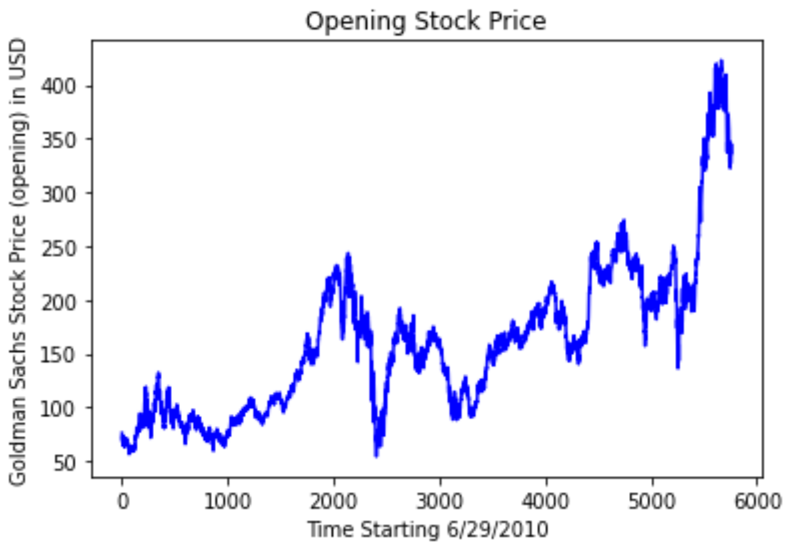
* Figure 1 *

```
gs.shape
print(closing)
```

```
0        70.375000
1        69.125000
2        67.937500
3        74.125000
4        70.687500
           ...
5757    345.380005
5758    339.000000
5759    343.010010
5760    335.609985
5761    336.230011
Name: Close, Length: 5762, dtype: float64
```

```
opening = gs.reset_index()['Open']
plt.plot(opening, color = "blue")
plt.xlabel("Time Starting 1999/05/04")
plt.ylabel("Goldman Sachs Stock Price (opening) in USD")
plt.title("Opening Stock Price")
```

```
Text(0.5, 1.0, 'Opening Stock Price')
```



*** Figure 2 ***

Now let's fit some model!

But first - I need to rearrange the data so that it will be easily accessible for the future split of train/test datasets.

```
scaler = MinMaxScaler(feature_range=(0,1))
closing = scaler.fit_transform(np.array(closing).reshape(-1,1))
```

First, let's use 70% of the data as our train data, and the rest as for testing.

```
size_train = int(len(closing)*0.7)
size_test = len(closing)-size_train
```

Now we specify the train data and test data.

```
train_data,test_data=closing[0:size_train,:],closing[size_train:len(closing),:1]
```

```
size_train, size_test
```

```
(4033, 1729)
```

```
def create_dataset(dataset, cons=1):
  temp1, temp2 = [], []
  for i in range(len(dataset)-cons-1):
    a = dataset[i:(i+cons), 0]
    temp1.append(a)
    temp2.append(dataset[i + cons, 0])
  return np.array(temp1), np.array(temp2)
```

```
past = 100
X_train, y_train = create_dataset(train_data, past)
X_test, y_test = create_dataset(test_data, past)
```

```
print(X_train.shape), print(y_train.shape)
print(X_test.shape), print(y_test.shape)
```

```
(3932, 100)
(3932,)
(1628, 100)
(1628,)
(None, None)
```

```
X_train =X_train.reshape(X_train.shape[0],X_train.shape[1] , 1)
X_test = X_test.reshape(X_test.shape[0],X_test.shape[1] , 1)
```

```
print(X_train.shape), print(y_train.shape)
print(X_test.shape), print(y_test.shape)
```

```
(3932, 100, 1)
(3932,)
(1628, 100, 1)
(1628,)
(None, None)
```

Building up layers and models!

```
model=Sequential()
model.add(LSTM(50,return_sequences=True,input_shape=(100,1)))
model.add(LSTM(50,return_sequences=True))
model.add(LSTM(50))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='adam')
```

```
model.fit(X_train,y_train,validation_data = (X_test, y_test), epochs=50,batch_size=32,verbose=1)
```

```
123/123 [==============================] - 12s 95ms/step - loss: 7.1904e-05 - va
Epoch 23/50
123/123 [==============================] - 12s 95ms/step - loss: 7.1472e-05 - va
Epoch 24/50
123/123 [==============================] - 12s 95ms/step - loss: 7.2487e-05 - va
Epoch 25/50
123/123 [==============================] - 12s 96ms/step - loss: 6.9928e-05 - va
Epoch 26/50
123/123 [==============================] - 12s 97ms/step - loss: 7.4105e-05 - va
```

```
    Epoch 27/50
    123/123 [==============================] - 12s 96ms/step - loss: 7.1358e-05 - va
    Epoch 28/50
    123/123 [==============================] - 12s 95ms/step - loss: 7.3227e-05 - va
    Epoch 29/50
    123/123 [==============================] - 12s 96ms/step - loss: 7.8638e-05 - va
    Epoch 30/50
    123/123 [==============================] - 12s 97ms/step - loss: 7.5930e-05 - va
    Epoch 31/50
    123/123 [==============================] - 12s 98ms/step - loss: 7.3705e-05 - va
    Epoch 32/50

    123/123 [==============================] - 12s 99ms/step - loss: 7.1570e-05 - va
    Epoch 33/50
    123/123 [==============================] - 12s 99ms/step - loss: 7.4442e-05 - va
    Epoch 34/50
    123/123 [==============================] - 13s 105ms/step - loss: 7.0915e-05 - v
    Epoch 35/50
    123/123 [==============================] - 14s 113ms/step - loss: 7.8238e-05 - v
    Epoch 36/50
    123/123 [==============================] - 12s 96ms/step - loss: 7.2504e-05 - va
    Epoch 37/50
    123/123 [==============================] - 12s 96ms/step - loss: 7.3975e-05 - va
    Epoch 38/50
    123/123 [==============================] - 12s 95ms/step - loss: 7.1845e-05 - va
    Epoch 39/50
    123/123 [==============================] - 12s 96ms/step - loss: 7.3740e-05 - va
    Epoch 40/50
    123/123 [==============================] - 12s 96ms/step - loss: 7.2714e-05 - va
    Epoch 41/50
    123/123 [==============================] - 12s 95ms/step - loss: 7.9249e-05 - va
    Epoch 42/50
    123/123 [==============================] - 12s 95ms/step - loss: 7.1620e-05 - va
    Epoch 43/50
    123/123 [==============================] - 12s 95ms/step - loss: 7.1356e-05 - va
    Epoch 44/50
    123/123 [==============================] - 12s 96ms/step - loss: 7.6494e-05 - va
    Epoch 45/50
    123/123 [==============================] - 12s 95ms/step - loss: 6.8863e-05 - va
    Epoch 46/50
    123/123 [==============================] - 12s 96ms/step - loss: 6.8071e-05 - va
    Epoch 47/50
    123/123 [==============================] - 12s 96ms/step - loss: 7.0388e-05 - va
    Epoch 48/50
    123/123 [==============================] - 12s 96ms/step - loss: 6.9949e-05 - va
    Epoch 49/50
    123/123 [==============================] - 12s 96ms/step - loss: 7.0039e-05 - va
    Epoch 50/50
    123/123 [==============================] - 12s 95ms/step - loss: 6.6783e-05 - va
    <keras.callbacks.History at 0x7f85be9375e0>
```

```
model_modified =Sequential()
model_modified.add(LSTM(50,return_sequences=True,input_shape=(100,1)))
model_modified.add(LSTM(50,return_sequences=True))
model_modified.add(LSTM(50))
model_modified.add(Dense(1))
model_modified.compile(loss='binary_crossentropy',optimizer='adam')
model_modified.fit(X_train,y_train,validation_split = 0.3, epochs=50,batch_size=32,verbose=1)
```

```
    86/86 [==============================] - 7s 82ms/step - loss: 0.4408 - val_loss:
    Epoch 23/50
    86/86 [==============================] - 7s 82ms/step - loss: 0.4407 - val_loss:
    Epoch 24/50
    86/86 [==============================] - 7s 82ms/step - loss: 0.4407 - val_loss:
    Epoch 25/50
    86/86 [==============================] - 7s 82ms/step - loss: 0.4407 - val_loss:
    Epoch 26/50
    86/86 [==============================] - 7s 81ms/step - loss: 0.4407 - val_loss:
    Epoch 27/50

    86/86 [==============================] - 7s 83ms/step - loss: 0.4406 - val_loss:
    Epoch 28/50
    86/86 [==============================] - 7s 82ms/step - loss: 0.4406 - val_loss:
    Epoch 29/50
    86/86 [==============================] - 7s 84ms/step - loss: 0.4406 - val_loss:
    Epoch 30/50
    86/86 [==============================] - 7s 83ms/step - loss: 0.4406 - val_loss:
    Epoch 31/50
    86/86 [==============================] - 7s 84ms/step - loss: 0.4406 - val_loss:
    Epoch 32/50
    86/86 [==============================] - 7s 83ms/step - loss: 0.4406 - val_loss:
    Epoch 33/50
    86/86 [==============================] - 7s 81ms/step - loss: 0.4406 - val_loss:
    Epoch 34/50
    86/86 [==============================] - 7s 82ms/step - loss: 0.4407 - val_loss:
    Epoch 35/50
    86/86 [==============================] - 7s 82ms/step - loss: 0.4406 - val_loss:
    Epoch 36/50
    86/86 [==============================] - 7s 82ms/step - loss: 0.4406 - val_loss:
    Epoch 37/50
    86/86 [==============================] - 7s 84ms/step - loss: 0.4407 - val_loss:
    Epoch 38/50
    86/86 [==============================] - 7s 83ms/step - loss: 0.4406 - val_loss:
    Epoch 39/50
    86/86 [==============================] - 7s 82ms/step - loss: 0.4406 - val_loss:
    Epoch 40/50
    86/86 [==============================] - 7s 81ms/step - loss: 0.4406 - val_loss:
    Epoch 41/50
    86/86 [==============================] - 7s 83ms/step - loss: 0.4406 - val_loss:
    Epoch 42/50
    86/86 [==============================] - 7s 83ms/step - loss: 0.4405 - val_loss:
    Epoch 43/50
    86/86 [==============================] - 9s 104ms/step - loss: 0.4405 - val_loss
    Epoch 44/50
    86/86 [==============================] - 10s 120ms/step - loss: 0.4406 - val_los
    Epoch 45/50
    86/86 [==============================] - 9s 102ms/step - loss: 0.4405 - val_loss
    Epoch 46/50
    86/86 [==============================] - 7s 83ms/step - loss: 0.4406 - val_loss:
    Epoch 47/50
    86/86 [==============================] - 7s 84ms/step - loss: 0.4405 - val_loss:
    Epoch 48/50
    86/86 [==============================] - 7s 84ms/step - loss: 0.4405 - val_loss:
    Epoch 49/50
```

```
86/86 [==============================] - 7s 85ms/step - loss: 0.4405 - val_loss:
Epoch 50/50
86/86 [==============================] - 7s 84ms/step - loss: 0.4407 - val_loss:
<keras.callbacks.History at 0x7f85b9b32b20>
```

```
acc = model.evaluate(X_test, y_test, batch_size=3)
print('Accuracy for model:', acc)

acc_mo = model_modified.evaluate(X_test, y_test, batch_size=3)
print('Accuracy for model_modified:', acc_mo)
```

```
543/543 [==============================] - 6s 12ms/step - loss: 8.8691e-04
Accuracy for model: 0.0008869105949997902
543/543 [==============================] - 6s 11ms/step - loss: 0.6187
Accuracy for model_modified: 0.6187497973442078
```

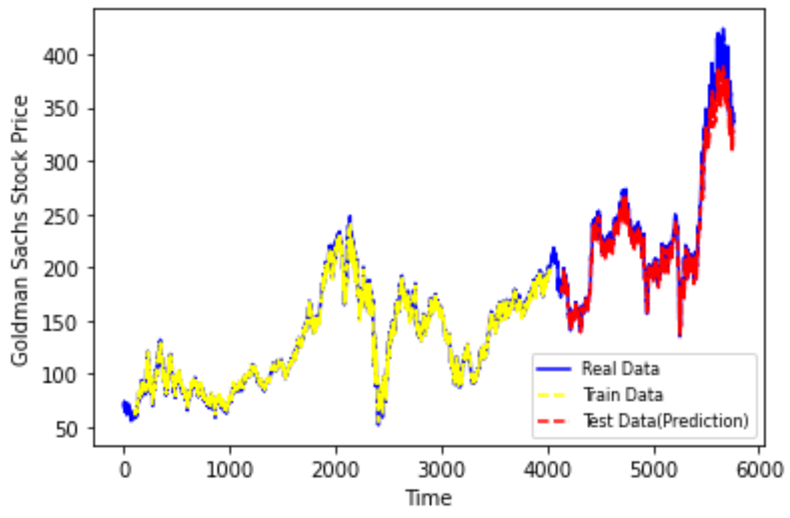
```
train_predict=model.predict(X_train)
test_predict=model.predict(X_test)

train_predict=scaler.inverse_transform(train_predict)
test_predict=scaler.inverse_transform(test_predict)
```

```
123/123 [==============================] - 3s 22ms/step
51/51 [==============================] - 1s 22ms/step
```

```
past=100
trainPredictPlot = np.empty_like(closing)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[past:len(train_predict)+past, :] = train_predict
# shift test predictions for plotting
testPredictPlot = np.empty_like(closing)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(train_predict)+(past*2)+1:len(closing)-1, :] = test_predict

# plot baseline and predictions
plt.plot(scaler.inverse_transform(closing), color = 'blue', linestyle = 'solid')
plt.plot(trainPredictPlot, color = 'yellow', linestyle = 'dashed')
plt.plot(testPredictPlot, color = 'red', linestyle = 'dashed')
plt.xlabel("Time")
plt.ylabel("Goldman Sachs Stock Price")
plt.legend(["Real Data", "Train Data", "Test Data(Prediction)"], loc = 'lower right', prop = {'size'
figure(figsize=(20,10))
plt.show()
```



```
<Figure size 1440x720 with 0 Axes>
```

*** Figure 3 ***

As shown above, it is a very good prediction!

Now we reduce the number of units in each LSTM layers to see the effects on overall accuracy

```
model2=Sequential()
model2.add(LSTM(10,return_sequences=True,input_shape=(100,1)))
model2.add(LSTM(10,return_sequences=True))
model2.add(LSTM(10))
model2.add(Dense(1))
model2.compile(loss='mean_squared_error',optimizer='adam')
```

```
model2.fit(X_train,y_train,validation_data = (X_test, y_test),epochs=50,batch_size=32,verbose=1)
```

```
    123/123 [==============================] - 8s 68ms/step - loss: 1.6439e-04 - val_
    Epoch 23/50
    123/123 [==============================] - 8s 64ms/step - loss: 1.5965e-04 - val_
    Epoch 24/50
    123/123 [==============================] - 9s 69ms/step - loss: 1.5947e-04 - val_
    Epoch 25/50
    123/123 [==============================] - 9s 70ms/step - loss: 1.5454e-04 - val_
    Epoch 26/50
    123/123 [==============================] - 8s 64ms/step - loss: 1.5799e-04 - val_
    Epoch 27/50
    123/123 [==============================] - 8s 63ms/step - loss: 1.4689e-04 - val_
    Epoch 28/50
    123/123 [==============================] - 8s 65ms/step - loss: 1.4426e-04 - val_
    Epoch 29/50
    123/123 [==============================] - 8s 64ms/step - loss: 1.4045e-04 - val_
    Epoch 30/50
    123/123 [==============================] - 8s 65ms/step - loss: 1.3104e-04 - val_
    Epoch 31/50
    123/123 [==============================] - 8s 66ms/step - loss: 1.1998e-04 - val_
```

```
Epoch 32/50
123/123 [==============================] - 8s 67ms/step - loss: 1.1982e-04 - val
Epoch 33/50
123/123 [==============================] - 8s 66ms/step - loss: 1.1947e-04 - val
Epoch 34/50
123/123 [==============================] - 8s 65ms/step - loss: 1.1404e-04 - val
Epoch 35/50
123/123 [==============================] - 8s 66ms/step - loss: 1.1389e-04 - val
Epoch 36/50
123/123 [==============================] - 8s 64ms/step - loss: 1.0876e-04 - val
Epoch 37/50
123/123 [==============================] - 8s 66ms/step - loss: 1.0388e-04 - val
Epoch 38/50
123/123 [==============================] - 8s 65ms/step - loss: 1.0527e-04 - val
Epoch 39/50
123/123 [==============================] - 8s 65ms/step - loss: 1.0110e-04 - val
Epoch 40/50
123/123 [==============================] - 8s 64ms/step - loss: 1.0168e-04 - val
Epoch 41/50
123/123 [==============================] - 8s 65ms/step - loss: 9.5542e-05 - val
Epoch 42/50
123/123 [==============================] - 8s 63ms/step - loss: 9.3957e-05 - val
Epoch 43/50
123/123 [==============================] - 8s 64ms/step - loss: 1.0377e-04 - val
Epoch 44/50
123/123 [==============================] - 8s 64ms/step - loss: 9.7677e-05 - val
Epoch 45/50
123/123 [==============================] - 8s 65ms/step - loss: 8.9608e-05 - val
Epoch 46/50
123/123 [==============================] - 8s 64ms/step - loss: 9.0825e-05 - val
Epoch 47/50
123/123 [==============================] - 8s 65ms/step - loss: 8.7262e-05 - val
Epoch 48/50
123/123 [==============================] - 8s 64ms/step - loss: 8.0768e-05 - val
Epoch 49/50
123/123 [==============================] - 8s 64ms/step - loss: 9.1429e-05 - val
Epoch 50/50
123/123 [==============================] - 8s 65ms/step - loss: 8.8637e-05 - val
<keras.callbacks.History at 0x7f85b9ff06d0>
```

```
acc2 = model2.evaluate(X_test, y_test, batch_size=3)
print('Accuracy for model2:', acc2)
```

```
543/543 [==============================] - 5s 9ms/step - loss: 9.7827e-04
Accuracy for model2: 0.0009782687993720174
```

Then we try to use a SimpleRNN model to predict:

```
model3 = Sequential()
model3.add(SimpleRNN(units = 50, activation='relu'))
model3.add(Dense(1))
```

```
model3.compile(loss = 'mean_squared_error', optimizer = 'adam')
model3.fit(X_train, y_train, epochs = 50, batch_size = 32)
```

```
    123/123 [==============================] - 1s 10ms/step - loss: 6.7396e-05
    Epoch 23/50
    123/123 [==============================] - 1s 10ms/step - loss: 7.0657e-05
    Epoch 24/50
    123/123 [==============================] - 1s 10ms/step - loss: 7.1736e-05
    Epoch 25/50
    123/123 [==============================] - 1s 11ms/step - loss: 6.9622e-05
    Epoch 26/50
    123/123 [==============================] - 1s 10ms/step - loss: 6.7852e-05
    Epoch 27/50
    123/123 [==============================] - 1s 10ms/step - loss: 6.9478e-05
    Epoch 28/50
    123/123 [==============================] - 1s 11ms/step - loss: 6.7433e-05
    Epoch 29/50
    123/123 [==============================] - 1s 10ms/step - loss: 7.0147e-05
    Epoch 30/50
    123/123 [==============================] - 1s 10ms/step - loss: 7.0484e-05
    Epoch 31/50
    123/123 [==============================] - 1s 11ms/step - loss: 6.6282e-05
    Epoch 32/50
    123/123 [==============================] - 1s 10ms/step - loss: 7.0999e-05
    Epoch 33/50
    123/123 [==============================] - 1s 11ms/step - loss: 6.9775e-05
    Epoch 34/50
    123/123 [==============================] - 1s 10ms/step - loss: 6.9481e-05
    Epoch 35/50
    123/123 [==============================] - 1s 11ms/step - loss: 6.7797e-05
    Epoch 36/50
    123/123 [==============================] - 1s 10ms/step - loss: 7.1041e-05
    Epoch 37/50
    123/123 [==============================] - 1s 10ms/step - loss: 6.9881e-05
    Epoch 38/50
    123/123 [==============================] - 1s 10ms/step - loss: 6.9074e-05
    Epoch 39/50
    123/123 [==============================] - 1s 10ms/step - loss: 7.0285e-05
    Epoch 40/50
    123/123 [==============================] - 1s 10ms/step - loss: 6.7086e-05
    Epoch 41/50
    123/123 [==============================] - 1s 11ms/step - loss: 6.7254e-05
    Epoch 42/50
    123/123 [==============================] - 1s 11ms/step - loss: 7.3154e-05
    Epoch 43/50
    123/123 [==============================] - 1s 10ms/step - loss: 6.5402e-05
    Epoch 44/50
    123/123 [==============================] - 1s 11ms/step - loss: 6.7827e-05
    Epoch 45/50
    123/123 [==============================] - 1s 11ms/step - loss: 6.9307e-05
    Epoch 46/50
    123/123 [==============================] - 1s 10ms/step - loss: 6.8517e-05
    Epoch 47/50
    123/123 [==============================] - 1s 11ms/step - loss: 6.7960e-05
    Epoch 48/50
    123/123 [==============================] - 1s 10ms/step - loss: 7.0083e-05
    Epoch 49/50
```

```
123/123 [==============================] - 1s 10ms/step - loss: 7.0111e-05
Epoch 50/50
123/123 [==============================] - 1s 10ms/step - loss: 6.8781e-05
<keras.callbacks.History at 0x7f85ba7cbdf0>
```

```
acc3 = model3.evaluate(X_test, y_test, batch_size=3)
print('Accuracy for model3:', acc3)
```

```
543/543 [==============================] - 2s 3ms/step - loss: 1.7643e-04
Accuracy for model3: 0.00017642784223426133
```

The shown loss is very convincing, let's make a small change to see the difference:

```
model3_modified = Sequential()
model3_modified.add(SimpleRNN(units = 5, activation='relu'))
model3_modified.add(Dense(1))
model3_modified.compile(loss = 'mean_squared_error', optimizer = 'adam')
model3_modified.fit(X_train, y_train, epochs = 50, batch_size = 32)
```

```
123/123 [==============================] - 1s 7ms/step - loss: 7.[illegible]e-05
Epoch 4/50
123/123 [==============================] - 1s 8ms/step - loss: 6.6546e-05
Epoch 5/50
123/123 [==============================] - 1s 8ms/step - loss: 6.9037e-05
Epoch 6/50
123/123 [==============================] - 1s 8ms/step - loss: 6.8653e-05
Epoch 7/50
123/123 [==============================] - 1s 8ms/step - loss: 7.0779e-05
Epoch 8/50
123/123 [==============================] - 1s 8ms/step - loss: 6.7384e-05
Epoch 9/50
123/123 [==============================] - 1s 8ms/step - loss: 7.3347e-05
Epoch 10/50
123/123 [==============================] - 1s 8ms/step - loss: 7.0180e-05
Epoch 11/50
123/123 [==============================] - 1s 8ms/step - loss: 6.7955e-05
Epoch 12/50
123/123 [==============================] - 1s 8ms/step - loss: 6.7022e-05
Epoch 13/50
123/123 [==============================] - 1s 8ms/step - loss: 6.7966e-05
Epoch 14/50
123/123 [==============================] - 1s 8ms/step - loss: 6.7817e-05
Epoch 15/50
123/123 [==============================] - 1s 8ms/step - loss: 6.7013e-05
Epoch 16/50
123/123 [==============================] - 1s 8ms/step - loss: 6.8242e-05
Epoch 17/50

123/123 [==============================] - 1s 7ms/step - loss: 6.7571e-05
Epoch 18/50
123/123 [==============================] - 1s 8ms/step - loss: 6.8840e-05
Epoch 19/50
123/123 [==============================] - 1s 8ms/step - loss: 6.7489e-05
Epoch 20/50
```

```
    123/123 [==============================] - 1s 8ms/step - loss: 6.9546e-05
    Epoch 21/50
    123/123 [==============================] - 1s 8ms/step - loss: 6.7243e-05
    Epoch 22/50
    123/123 [==============================] - 1s 8ms/step - loss: 6.9874e-05
    Epoch 23/50
    123/123 [==============================] - 1s 8ms/step - loss: 6.8842e-05
    Epoch 24/50
    123/123 [==============================] - 1s 7ms/step - loss: 6.6090e-05
    Epoch 25/50
    123/123 [==============================] - 1s 7ms/step - loss: 6.7663e-05
    Epoch 26/50
    123/123 [==============================] - 1s 7ms/step - loss: 7.0177e-05
    Epoch 27/50
    123/123 [==============================] - 1s 8ms/step - loss: 7.0293e-05
    Epoch 28/50
    123/123 [==============================] - 1s 8ms/step - loss: 6.9463e-05
    Epoch 29/50
    123/123 [==============================] - 1s 8ms/step - loss: 7.0207e-05
    Epoch 30/50
    123/123 [==============================] - 1s 7ms/step - loss: 6.8577e-05
    Epoch 31/50
    123/123 [==============================] - 1s 8ms/step - loss: 6.6983e-05
    Epoch 32/50
    123/123 [==============================] - 1s 8ms/step - loss: 6.7825e-05
```

```
acc3_mo = model3_modified.evaluate(X_test, y_test, batch_size=3)
print('Accuracy for model3_modified:', acc3_mo)
```

```
    543/543 [==============================] - 2s 4ms/step - loss: 0.0222
    Accuracy for model3_modified: 0.022191757336258888
```

With a significant decrease in the number of units, the accuracy decreased. However, the model is still very convincing based on the high accuracy.

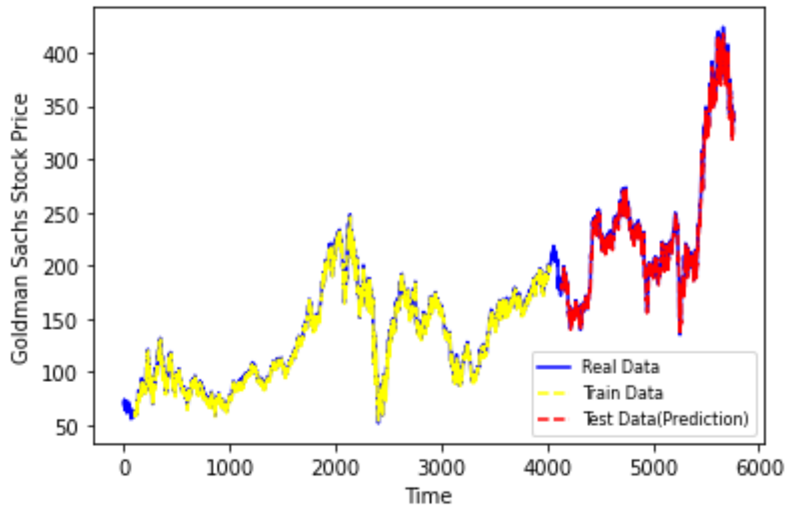
```
train_predict3=model3.predict(X_train)
test_predict3=model3.predict(X_test)

train_predict3=scaler.inverse_transform(train_predict3)
test_predict3=scaler.inverse_transform(test_predict3)
```

```
    123/123 [==============================] - 1s 4ms/step
    51/51 [==============================] - 0s 4ms/step
```

```
past=100
trainPredictPlot3 = np.empty_like(closing)
trainPredictPlot3[:, :] = np.nan
trainPredictPlot3[past:len(train_predict3)+past, :] = train_predict3
# shift test predictions for plotting
testPredictPlot3 = np.empty_like(closing)
testPredictPlot3[:, :] = np.nan
testPredictPlot3[len(train_predict3)+(past*2)+1:len(closing)-1, :] = test_predict3

# plot baseline and predictions
# plot baseline and predictions
plt.plot(scaler.inverse_transform(closing), color = 'blue', linestyle = 'solid')
plt.plot(trainPredictPlot3, color = 'yellow', linestyle = 'dashed')
plt.plot(testPredictPlot3, color = 'red', linestyle = 'dashed')
plt.xlabel("Time")
plt.ylabel("Goldman Sachs Stock Price")
plt.legend(["Real·Data",·"Train·Data",·"Test·Data(Prediction)"],·loc·=·'lower·right',·prop·=·{'size'
figure(figsize=(20,10))
plt.show()
```



```
<Figure size 1440x720 with 0 Axes>
```

*** Figure 4 ***

It is a very good fit for our dataset, as the red line (prediction) aligns with the blue line (true data) very well thoroughly.

Now we fit with a different layer type

```
model4 = Sequential()

model4.add(Flatten())
model4.add(Dense(50, activation = 'relu'))
model4.add(Dense(10, activation = 'relu'))
model4.add(Dense(10, activation = 'relu'))
model4.add(Dense(1))


model4.compile(optimizer = 'adam', loss = 'mean_squared_error')
model4.fit(X_train, y_train, epochs = 50, batch_size = 32)
```

```
Epoch 10/50
123/123 [==============================] - 0s 1ms/step - loss: 1.7703e-04
Epoch 11/50
123/123 [==============================] - 0s 1ms/step - loss: 1.7603e-04
Epoch 12/50
123/123 [==============================] - 0s 1ms/step - loss: 1.5836e-04
Epoch 13/50
```

```
123/123 [==============================] - 0s 1ms/step - loss: 1.5195e-04
Epoch 14/50
123/123 [==============================] - 0s 1ms/step - loss: 1.7834e-04
Epoch 15/50
123/123 [==============================] - 0s 1ms/step - loss: 1.4840e-04
Epoch 16/50
123/123 [==============================] - 0s 1ms/step - loss: 1.7037e-04
Epoch 17/50
123/123 [==============================] - 0s 1ms/step - loss: 1.4760e-04
Epoch 18/50
123/123 [==============================] - 0s 1ms/step - loss: 1.4088e-04
Epoch 19/50
123/123 [==============================] - 0s 1ms/step - loss: 1.2740e-04
Epoch 20/50
123/123 [==============================] - 0s 1ms/step - loss: 1.2380e-04
Epoch 21/50
123/123 [==============================] - 0s 1ms/step - loss: 1.2590e-04
Epoch 22/50
123/123 [==============================] - 0s 1ms/step - loss: 1.2942e-04
Epoch 23/50
123/123 [==============================] - 0s 1ms/step - loss: 1.2433e-04
Epoch 24/50
123/123 [==============================] - 0s 1ms/step - loss: 1.3311e-04
Epoch 25/50
123/123 [==============================] - 0s 1ms/step - loss: 1.1501e-04
Epoch 26/50
123/123 [==============================] - 0s 1ms/step - loss: 1.3239e-04
Epoch 27/50
123/123 [==============================] - 0s 1ms/step - loss: 1.5356e-04
Epoch 28/50
123/123 [==============================] - 0s 1ms/step - loss: 1.2056e-04
Epoch 29/50
123/123 [==============================] - 0s 1ms/step - loss: 1.2221e-04
Epoch 30/50
123/123 [==============================] - 0s 1ms/step - loss: 1.0532e-04
Epoch 31/50
123/123 [==============================] - 0s 1ms/step - loss: 1.0902e-04

Epoch 32/50
123/123 [==============================] - 0s 1ms/step - loss: 1.0645e-04
Epoch 33/50
123/123 [==============================] - 0s 1ms/step - loss: 1.1360e-04
Epoch 34/50
123/123 [==============================] - 0s 1ms/step - loss: 1.0985e-04
Epoch 35/50
123/123 [==============================] - 0s 1ms/step - loss: 9.6585e-05
Epoch 36/50
123/123 [==============================] - 0s 1ms/step - loss: 1.0675e-04
Epoch 37/50
123/123 [==============================] - 0s 1ms/step - loss: 1.1741e-04
Epoch 38/50
123/123 [==============================] - 0s 1ms/step - loss: 1.2677e-04
```

```
acc4 = model4.evaluate(X_test, y_test, batch_size=3)
print('Accuracy for model4', acc4)
```

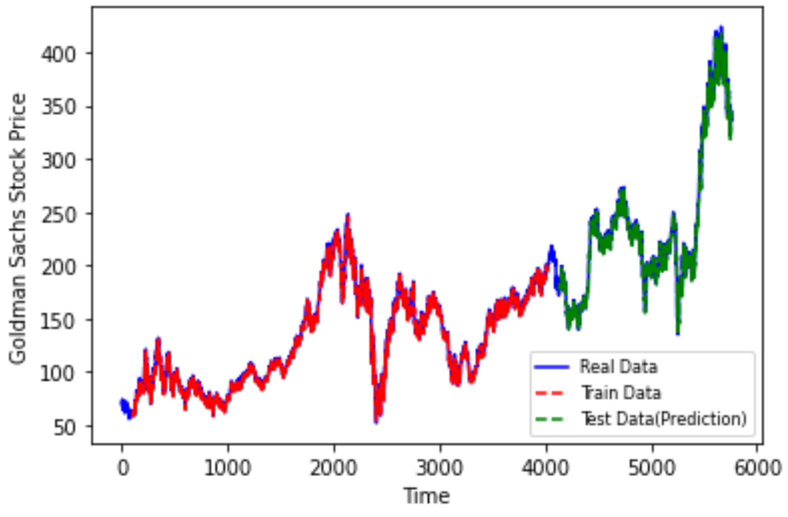
```
    543/543 [==============================] - 1s 895us/step - loss: 3.0982e-04
    Accuracy for model4 0.00030981944291852415
```

```
train_predict4=model4.predict(X_train)
test_predict4=model4.predict(X_test)
```

```
    123/123 [==============================] - 0s 1ms/step
    51/51 [==============================] - 0s 2ms/step
```

```
past=100
trainPredictPlot4 = np.empty_like(closing)
trainPredictPlot4[:, :] = np.nan
trainPredictPlot4[past:len(train_predict4)+past, :] = train_predict4
# shift test predictions for plotting
testPredictPlot4 = np.empty_like(closing)
testPredictPlot4[:, :] = np.nan
testPredictPlot4[len(train_predict4)+(past*2)+1:len(closing)-1, :] = test_predict4

# plot baseline and predictions
plt.plot(scaler.inverse_transform(closing), color = 'blue', linestyle = 'solid')
plt.plot(trainPredictPlot3, color = 'red', linestyle = 'dashed')
plt.plot(testPredictPlot3, color = 'green', linestyle = 'dashed')
plt.xlabel("Time")
plt.ylabel("Goldman Sachs Stock Price")
plt.legend(["Real Data", "Train Data", "Test Data(Prediction)"], loc = 'lower right', prop = {'size'
figure(figsize=(20,10))
plt.show()
```



```
    <Figure size 1440x720 with 0 Axes>
```

*** Figure 5 ***

Again, our predicted stock prices (gree line) is very close to the underlying real stock prices (blue line). This is also a very good fit!

Non-Sequential Model

```
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras import Model
import urllib.request
from tensorflow.python.ops.array_ops import tensor_strided_slice_update
import tensorflow as tf
from tensorflow.keras import Model, layers

class NonSequential(Model):
  def __init__(self):
    super(NonSequential, self).__init__()
    self.rnn = layers.SimpleRNN(64)
    self.relu1 = layers.Activation('relu')

    self.flatten1 = layers.Flatten()
    self.dense1 = layers.Dense(20)

    self.flatten2 = layers.Flatten()
    self.dense2 = layers.Dense(20)
    self.relu3 = layers.Activation('relu')

    self.dense3 = layers.Dense(1)

  def call(self, x):
    weighted_x = self.rnn(x)
    weighted_x = self.relu1(weighted_x)

    weighted_x2 = self.flatten1(weighted_x)
    weighted_x2 = self.dense1(weighted_x2)

    weighted_x3 = self.flatten2(weighted_x)
    weighted_x3 = self.dense2(weighted_x3)
    weighted_x3 = self.relu3(weighted_x3)

    return self.dense3(weighted_x2 + weighted_x3)


model_nS = NonSequential()

model_nS.compile(optimizer = 'adam', loss = 'mean_squared_error')
model_nS.fit(X_train, y_train, epochs = 50, batch_size = 32)
```

```
Epoch 23/50
123/123 [==============================] - 1s 11ms/step - loss: 8.0616e-05
Epoch 24/50
123/123 [==============================] - 1s 11ms/step - loss: 7.3326e-05
Epoch 25/50
123/123 [==============================] - 1s 11ms/step - loss: 6.9556e-05
Epoch 26/50
123/123 [==============================] - 1s 11ms/step - loss: 7.6962e-05
Epoch 27/50
```

```
123/123 [==============================] - 1s 11ms/step - loss: 7.2984e-05
Epoch 28/50
123/123 [==============================] - 1s 11ms/step - loss: 7.7138e-05
Epoch 29/50
123/123 [==============================] - 1s 11ms/step - loss: 7.7654e-05
Epoch 30/50
123/123 [==============================] - 1s 11ms/step - loss: 7.3214e-05
Epoch 31/50
123/123 [==============================] - 1s 11ms/step - loss: 6.9562e-05
Epoch 32/50
123/123 [==============================] - 1s 11ms/step - loss: 7.4147e-05

Epoch 33/50
123/123 [==============================] - 1s 11ms/step - loss: 6.8535e-05
Epoch 34/50
123/123 [==============================] - 1s 11ms/step - loss: 8.3437e-05
Epoch 35/50
123/123 [==============================] - 1s 11ms/step - loss: 6.7948e-05
Epoch 36/50
123/123 [==============================] - 1s 11ms/step - loss: 8.0598e-05
Epoch 37/50
123/123 [==============================] - 1s 11ms/step - loss: 7.3918e-05
Epoch 38/50
123/123 [==============================] - 1s 11ms/step - loss: 7.4467e-05
Epoch 39/50
123/123 [==============================] - 1s 11ms/step - loss: 7.4771e-05
Epoch 40/50
123/123 [==============================] - 1s 11ms/step - loss: 6.7451e-05
Epoch 41/50
123/123 [==============================] - 1s 11ms/step - loss: 6.7809e-05
Epoch 42/50
123/123 [==============================] - 1s 11ms/step - loss: 6.8822e-05
Epoch 43/50
123/123 [==============================] - 1s 11ms/step - loss: 7.2994e-05
Epoch 44/50
123/123 [==============================] - 1s 11ms/step - loss: 7.1513e-05
Epoch 45/50
123/123 [==============================] - 1s 11ms/step - loss: 6.4288e-05
Epoch 46/50
123/123 [==============================] - 1s 11ms/step - loss: 6.7773e-05
Epoch 47/50
123/123 [==============================] - 1s 11ms/step - loss: 7.4002e-05
Epoch 48/50
123/123 [==============================] - 1s 11ms/step - loss: 7.4338e-05
Epoch 49/50
123/123 [==============================] - 1s 11ms/step - loss: 7.0527e-05
Epoch 50/50
123/123 [==============================] - 1s 11ms/step - loss: 7.2147e-05
543/543 [==============================] - 2s 3ms/step - loss: 4.5392e-04
Accuracy for Non-Sequential model 0.0004539211804512888
```

```
acc_ns = model_nS.evaluate(X_test, y_test, batch_size=3)
print('Accuracy for Non-Sequential model', acc_ns)
```

```
543/543 [==============================] - 2s 3ms/step - loss: 4.5392e-04
Accuracy for Non-Sequential model 0.0004539211804512888
```

We see that the accuracy for this non-sequential model is also very high. We change a little bit and check the accuracy again.

```
class NonSequential2(Model):
  def __init__(self):
    super(NonSequential2, self).__init__()
    self.conv1 = layers.conv1D(64, input_shape = (100,1))
    self.relu1 = layers.Activation('relu')

    self.flatten1 = layers.Flatten()
    self.dense1 = layers.Dense(20)

    self.flatten2 = layers.Flatten()
    self.dense2 = layers.Dense(20)
    self.relu3 = layers.Activation('relu')

    self.dense3 = layers.Dense(1)

  def call(self, x):
    weighted_x = self.conv1(x)
    weighted_x = self.relu1(weighted_x)

    weighted_x2 = self.flatten1(weighted_x)
    weighted_x2 = self.dense1(weighted_x2)

    weighted_x3 = self.flatten2(weighted_x)
    weighted_x3 = self.dense2(weighted_x3)
    weighted_x3 = self.relu3(weighted_x3)

    return self.dense3(weighted_x2 + weighted_x3)


model_nS2 = NonSequential()

model_nS2.compile(optimizer = 'adam', loss = 'mean_squared_error')
model_nS2.fit(X_train, y_train, epochs = 50, batch_size = 32)
```

```
    123/123 [==============================] - 1s 11ms/step - loss: 7.4267e-05
    Epoch 23/50
    123/123 [==============================] - 2s 17ms/step - loss: 7.3862e-05
    Epoch 24/50
    123/123 [==============================] - 2s 17ms/step - loss: 7.6748e-05
    Epoch 25/50
    123/123 [==============================] - 2s 17ms/step - loss: 6.9189e-05
    Epoch 26/50
    123/123 [==============================] - 3s 21ms/step - loss: 6.7578e-05
    Epoch 27/50
    123/123 [==============================] - 2s 14ms/step - loss: 8.6222e-05
    Epoch 28/50
    123/123 [==============================] - 1s 11ms/step - loss: 7.4333e-05
    Epoch 29/50
    123/123 [==============================] - 1s 11ms/step - loss: 7.4583e-05
    Epoch 30/50
    123/123 [==============================] - 1s 11ms/step - loss: 7.7701e-05
    Epoch 31/50
```

```
123/123 [==============================] - 1s 11ms/step - loss: 6.7983e-05
Epoch 32/50
123/123 [==============================] - 1s 11ms/step - loss: 6.6403e-05
Epoch 33/50
123/123 [==============================] - 2s 13ms/step - loss: 6.9041e-05
Epoch 34/50
123/123 [==============================] - 2s 15ms/step - loss: 7.4295e-05
Epoch 35/50
123/123 [==============================] - 1s 12ms/step - loss: 7.1306e-05
Epoch 36/50
123/123 [==============================] - 1s 11ms/step - loss: 6.7482e-05
Epoch 37/50
123/123 [==============================] - 1s 11ms/step - loss: 8.9808e-05
Epoch 38/50
123/123 [==============================] - 1s 11ms/step - loss: 7.6965e-05
Epoch 39/50
123/123 [==============================] - 1s 11ms/step - loss: 7.0388e-05
Epoch 40/50
123/123 [==============================] - 1s 11ms/step - loss: 7.0179e-05
Epoch 41/50
123/123 [==============================] - 1s 11ms/step - loss: 7.1988e-05
Epoch 42/50
123/123 [==============================] - 1s 10ms/step - loss: 7.5095e-05
Epoch 43/50
123/123 [==============================] - 1s 10ms/step - loss: 7.2194e-05
Epoch 44/50
123/123 [==============================] - 1s 10ms/step - loss: 7.2049e-05
Epoch 45/50
123/123 [==============================] - 1s 11ms/step - loss: 6.6395e-05
Epoch 46/50
123/123 [==============================] - 1s 11ms/step - loss: 6.9642e-05
Epoch 47/50
123/123 [==============================] - 1s 11ms/step - loss: 7.4878e-05
Epoch 48/50
123/123 [==============================] - 1s 11ms/step - loss: 7.6008e-05
Epoch 49/50
123/123 [==============================] - 1s 11ms/step - loss: 6.6152e-05
Epoch 50/50
123/123 [==============================] - 1s 11ms/step - loss: 7.3745e-05
<keras.callbacks.History at 0x7f85aaed9730>
```

```
acc_ns2 = model_nS2.evaluate(X_test, y_test, batch_size=3)
print('Accuracy for Non-Sequential model 2', acc_ns2)
```

```
543/543 [==============================] - 2s 3ms/step - loss: 7.4554e-04
Accuracy for Non-Sequential model 2 0.0007455385639332235
```

We see the accuracy decrease a little bit, but this model is still very good. We observe that even we did not use any SimpleRNN or LSTM layers, we can still arrive at some pretty decent outcomes using sequential and non-sequential models.

Linear Regression Model

```
X_LR = gs[['High','Low','Open','Volume']].values
y_LR = gs['Close'].values
```

```
X_LR_train, X_LR_test, y_LR_train, y_LR_test = train_test_split(X_LR,y_LR, test_size=0.3, random_sta
```

```
rg = LinearRegression()
rg.fit(X_LR_train, y_LR_train)
print(rg.coef_)
print(rg.intercept_)

prediction = rg.predict(X_LR_test)
print(prediction)
```

```
[ 7.95954956e-01  7.92577066e-01 -5.88396122e-01  1.16758272e-08]
-0.06227950409197547
[216.14365765 210.18982355 197.09339735 ... 188.96856919 151.11622747
  76.05420306]
```

Do comparison with the true closing stock price:

```
data = pd.DataFrame({'Actual': y_LR_test.flatten(), 'Predicted' : prediction.flatten()})
data
```

	Actual	Predicted
0	217.139999	216.143658
1	209.940002	210.189824
2	196.419998	197.093397
3	110.875000	111.041833
4	81.500000	81.276313
...	...	...
1724	158.240005	157.779724
1725	243.940002	245.102347
1726	189.779999	188.968569
1727	149.949997	151.116227
1728	76.150002	76.054203

1729 rows × 2 columns

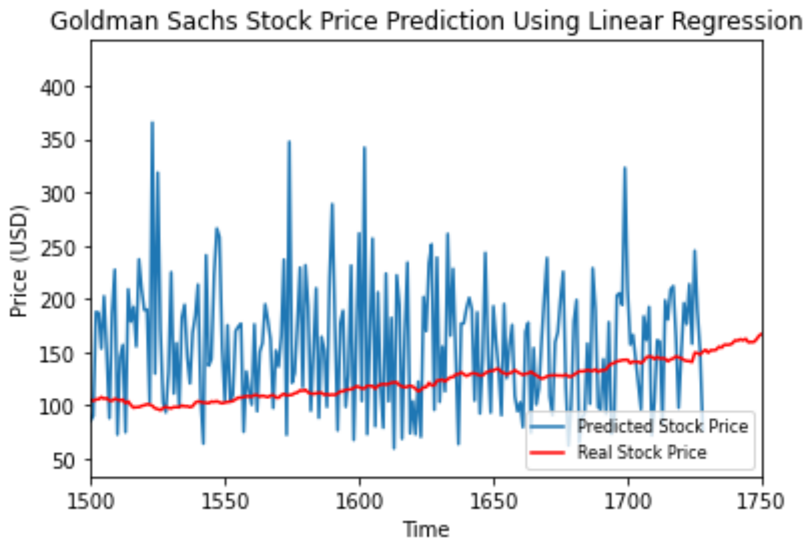
*** Table 2 ***

```
x=0
for i in range(0, 1729):
   if np.abs(y_LR_test[i] - prediction[i])<1:
      x=x+1
print(x/len(gs))

plt.plot(data["Predicted"])
plt.plot(gs['Close'], color = "red")

plt.xlim([1500, 1750])
plt.title("Goldman Sachs Stock Price Prediction Using Linear Regression")
plt.xlabel("Time")
plt.ylabel("Price (USD)")
plt.legend(["Predicted Stock Price", "Real Stock Price"], loc = 'lower right', prop = {'size' : 8})
```

```
0.20982297813259285
<matplotlib.legend.Legend at 0x7f85c3085fa0>
```



*** Figure 6 ***

When the real stock price is within $1 of the predicted stock price, we considered it as close enough and hence a good guess.

By the table above, we find that approximately 21% of our result lied within the good guess range. This is a relatively low accuracy. We also observe the plot above, our predicted stock price bounce back and forth continuously, representing an unsteady state. Therefore, by comaprison, clearly we should go for the neural networks for stock price prediction tasks.