CAP 6615 - NEURAL NETWORKS
Programming Assignment 3
Recurrent Neural Network

Monica Bhargavi Kodali
Sravanti Ratnakaram
Pagolu Carol Navya
Karthik Gannamaneni
Pramod Kumar Varma Manthena

Spring Semester 2021
21 March 2021

ABSTRACT

A Recurrent Neural Network (RNN) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. It is a form of generalized feed forward network that has internal memory which can be used to process the input data.

In this assignment, a recurrent neural network (RNN) is designed and trained on S&P 500 data. The model is tested by predicting the next one,two,three or four values using a sliding sampling window of width 180 days.

The models are also re-tested on *noise-corrupted* data to observe the performance. The goal of this project is to optimize the RNN to give the best possible results.

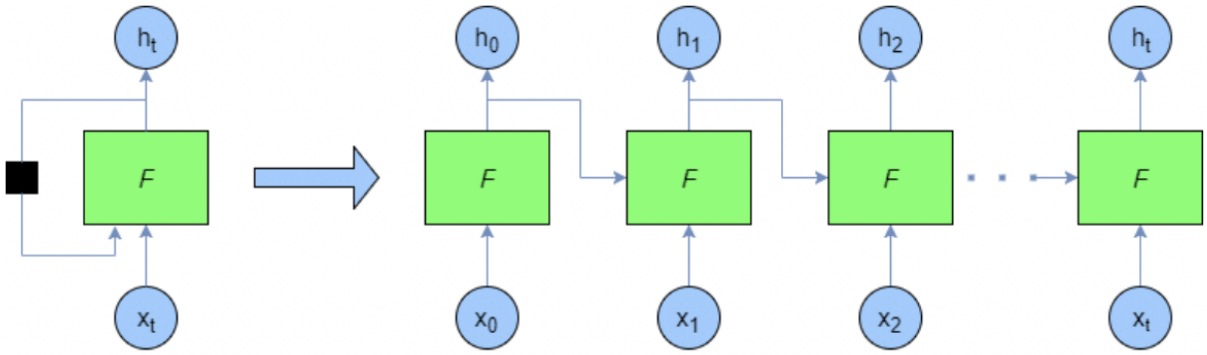


Figure 1: Sample Recurrent Neural Network

TABLE OF CONTENTS

1. DATASET GENERATION

The S&P 500 dataset is collected from the *Yahoo Finance website* between the time frame of January 4, 1960 to December 31, 2020. The table from the website is scraped and stored into a csv file. Shiller P/E ratio is one of the standard metrics used to evaluate whether a market is overvalued, undervalued, or fairly-valued. It can be downloaded from *the Shiller-PE website*.

The Shiller P/E ratio data was collected monthly from 1960 to 2020. We use linear interpolation method to convert this monthly data to daily data. The formula for linear interpolation is as follows:

$$y = y1 + ((x - x1) / (x2 - x1)) * (y2 - y1)$$ where,

X = known value
y= unknown value
(x1,y1) = coordinates below the known value
(x2,y2) = coordinates above the known value

S&P 500 data and Shiller P/E ratio data were then merged and stored in *finalDS.csv.* The data was then divided into training and test sets.

	Unnamed: 0	Adj Close	Close	Date	High	Low	Open	Volume	S&P 500 PE Ratio
0	0	3756.07	3756.07	2020-12-31	3760.20	3726.88	3733.27	3172510000	37.850000
1	1	3732.04	3732.04	2020-12-30	3744.63	3730.21	3736.19	3145200000	37.842333
2	2	3727.04	3727.04	2020-12-29	3756.12	3723.31	3750.01	3387030000	37.834667
3	3	3735.36	3735.36	2020-12-28	3740.51	3723.03	3723.03	3527460000	37.827000
4	4	3703.06	3703.06	2020-12-24	3703.82	3689.32	3694.03	1885090000	37.796333

Figure 2: snapshot of dataset

The generated dataset is a time series data of length N, defined as $CP_0, CP_1, \ldots, CP_{N-1}$ in which CP_i is the close price on day 'i', $0 \leq i < N$. The entire dataset is divided into fixed window size of W = 180. Therefore, whenever the window is moved to the right by size W , there is no overlapping between data in all the sliding windows. Then the data values are normalized within range of [0,1]. Then the data values are split into test and train datasets.

```
#selecting test data as data from 2015 to 2020
split = 1510
X_train=X[:-split]
X_test=X[-split:]
y_train=y[:-split]
y_test=y[-split:]
print(X_train.shape,y_train.shape,X_test.shape,y_test.shape)
```

```
(13665, 180, 1) (13665, 1) (1510, 180, 1) (1510, 1)
```

	Adj Close	Close	Date	High	Low	Open	Volume	S&P 500 PE Ratio
0	3756.07	3756.07	31-12-20	3760.2	3726.88	3733.27	3.17E+09	37.85
1	3732.04	3732.04	30-12-20	3744.63	3730.21	3736.19	3.15E+09	37.84233
2	3727.04	3727.04	29-12-20	3756.12	3723.31	3750.01	3.39E+09	37.83467
3	3735.36	3735.36	28-12-20	3740.51	3723.03	3723.03	3.53E+09	37.827
4	3703.06	3703.06	24-12-20	3703.82	3689.32	3694.03	1.89E+09	37.79633
5	3690.01	3690.01	23-12-20	3711.24	3689.28	3693.42	3.77E+09	37.78867
6	3687.26	3687.26	22-12-20	3698.26	3676.16	3698.08	4.02E+09	37.781
7	3694.92	3694.92	21-12-20	3702.9	3636.48	3684.28	4.73E+09	37.77333
8	3709.41	3709.41	18-12-20	3726.7	3685.84	3722.39	7.07E+09	37.75033
9	3722.48	3722.48	17-12-20	3725.12	3710.87	3713.65	4.18E+09	37.74267
10	3701.17	3701.17	16-12-20	3711.27	3688.57	3696.25	4.06E+09	37.735
11	3694.62	3694.62	15-12-20	3695.29	3659.62	3666.41	4.36E+09	37.72733
12	3647.49	3647.49	14-12-20	3697.61	3645.84	3675.27	4.59E+09	37.71967
13	3663.46	3663.46	11-12-20	3665.91	3633.4	3656.08	4.37E+09	37.69667
14	3668.1	3668.1	10-12-20	3678.49	3645.18	3659.13	4.62E+09	37.689
15	3672.82	3672.82	09-12-20	3712.39	3660.54	3705.98	5.21E+09	37.68133
16	3702.25	3702.25	08-12-20	3708.45	3678.83	3683.05	4.55E+09	37.67367
17	3691.96	3691.96	07-12-20	3697.41	3678.88	3694.73	4.79E+09	37.666
18	3699.12	3699.12	04-12-20	3699.2	3670.94	3670.94	5.09E+09	37.643
19	3666.72	3666.72	03-12-20	3682.73	3657.17	3668.28	5.09E+09	37.63533
20	3669.01	3669.01	02-12-20	3670.96	3644.84	3653.78	5.03E+09	37.62767
21	3662.45	3662.45	01-12-20	3678.45	3645.87	3645.87	5.4E+09	37.62
22	3621.63	3621.63	30-11-20	3634.18	3594.39	3634.18	6.29E+09	36.66
23	3638.35	3638.35	27-11-20	3644.31	3629.33	3638.55	2.78E+09	36.60517

Figure 3: Portion of the test data

12895	96.39	96.39	18-11-69	97	95.57	0	11010000	16.098
12896	96.41	96.41	17-11-69	97.36	95.82	0	10120000	16.124
12897	97.07	97.07	14-11-69	97.44	96.36	0	10580000	16.202
12898	97.42	97.42	13-11-69	98.34	96.54	0	12090000	16.228
12899	97.89	97.89	12-11-69	98.72	97.28	0	12480000	16.254
12900	98.07	98.07	11-11-69	98.79	97.45	0	10080000	16.28
12901	98.33	98.33	10-11-69	99.23	97.65	0	12490000	16.306
12902	98.26	98.26	07-11-69	99.01	97.18	0	13280000	16.384
12903	97.67	97.67	06-11-69	98.31	96.8	0	11110000	16.41
12904	97.64	97.64	05-11-69	98.39	96.75	0	12110000	16.436
12905	97.21	97.21	04-11-69	97.82	95.84	0	12340000	16.462
12906	97.15	97.15	03-11-69	97.82	96.19	0	11140000	16.488
12907	97.12	97.12	31-10-69	98.03	96.33	0	13100000	16.5329
12908	96.93	96.93	30-10-69	97.47	95.61	0	12820000	16.52581
12909	96.81	96.81	29-10-69	97.92	96.26	0	12380000	16.51871
12910	97.66	97.66	28-10-69	98.55	97.02	0	12410000	16.51161
12911	97.97	97.97	27-10-69	98.78	97.49	0	12160000	16.50452
12912	98.12	98.12	24-10-69	98.83	96.97	0	15430000	16.48323
12913	97.46	97.46	23-10-69	98.39	96.46	0	14780000	16.47613
12914	97.83	97.83	22-10-69	98.61	96.56	0	19320000	16.46903
12915	97.2	97.2	21-10-69	97.84	95.86	0	16460000	16.46194
12916	96.46	96.46	20-10-69	97.17	95.29	0	13540000	16.45484
12917	96.26	96.26	17-10-69	97.24	95.38	0	13740000	16.43355
12918	96.37	96.37	16-10-69	97.54	95.05	0	19500000	16.42645
12919	95.72	95.72	15-10-69	96.56	94.65	0	15740000	16.41935

Figure 4: Portion of the train data

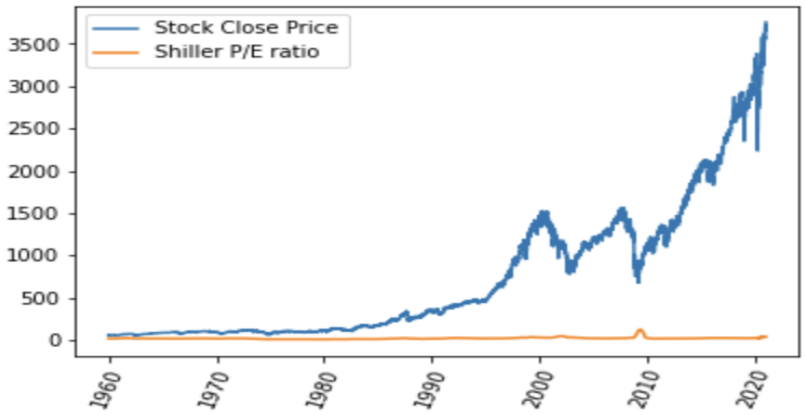


Figure 5: Graphical representation of data

2. NETWORK PARAMETERS

1. **Input :** generated dataset
2. **Predicted Output feature:** predicted 'Close' price
3. **Activation function:** sigmoid, tanh
4. **Epochs**:1,10,100.(tabulated below)
5. **Optimizer:** Adam
6. **Evaluation metrics:** prediction error, accuracy
7. **Number of timesteps:** 180
8. **Loss metrics:** Mean squared error

The model is trained on the entire data set from 1960 to 2014 and tested on data from 2015 to 2020. The RNN built is a regression model that predicts the future Stock Close Price. The resulting output of the predicted price is then compared to the original price values to evaluate the model.

Number of iterations (epochs)	Accuracy
1	97.64
10	90.68
100	95.14

Table 1: Unoptimized RNN accuracy tabulations

Number of iterations (epochs)	Accuracy
1	90.49
10	95.54
100	97.72

Table 2: Optimized RNN accuracy tabulations

The number of **epochs** were chosen in a manner to improve the performance of the model. The accuracy of the model for the optimized and the unoptimized models were calculated while changing the number of epochs. It was observed that when the **number of epochs was 1**, the accuracy dropped from **97.64 to 90.49** when an optimizer was used. It has also been observed that as the number of epochs have increased drastically, it led to **over-fitting** of the model which led to a great decrease in the accuracy of the model.

Hence, the number of epochs was chosen in such a way that the loss was minimum. In the case of **10 epochs**, the optimized RNN produced better results. As the accuracy increased from **90.68 to 95.14.** In the case of **100 epochs**, The accuracy for the unoptimized RNN is **95.14** and for optimized it is **97.72** which is a suitable accuracy. So, **100 epochs** was **chosen** as it provided the best accuracy in the case of both optimized and unoptimized.

3. RNN Model

The architecture employed to deploy this model is the **LSTM model**. ***LSTM*** stands for Long Short Term Memory. Long Short-Term Memory (LSTM) networks are a ***modified version*** of recurrent neural networks, which makes it easier to remember past data in memory. The ***vanishing gradient*** problem of RNN is resolved here. LSTM is well-suited to classify, process and predict time series given time lags of unknown duration. It trains the model by using back-propagation. In an LSTM network,**three gates** are present:

i. Input gate — discover which value from input should be used to modify the memory. **Sigmoid** function decides which values to let through **0,1.** and **tanh** function gives weightage to the values which are passed deciding their level of importance ranging from**-1** to **1**

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

ii. Forget gate — discover what details to be discarded from the block. It is decided by the **sigmoid function.** it looks at the previous state(**ht-1**) and the content input(**Xt**) and outputs a number between **0(***omit this*)and **1(***keep this***)**for each number in the cell state **Ct−1**.

$$f_t = \sigma\left(W_f \cdot \left[h_{t-1}, x_t\right] \; + \; b_f\right)$$

iii. Output gate — the input and the memory of the block is used to decide the output. **Sigmoid** function decides which values to let through **0,1.** and **tanh** function gives weightage to the values which are passed deciding their level of importance ranging from**-1** to **1** and multiplied with output of **Sigmoid.**

$$o_t = \sigma\left(W_o \; [\, h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

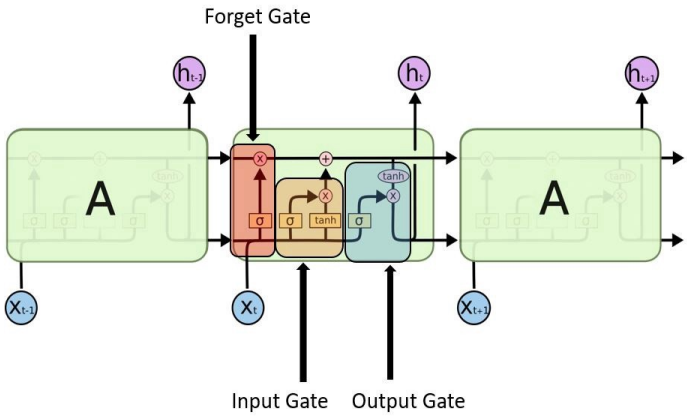
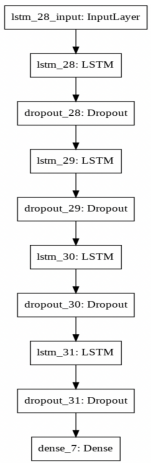


Figure 6: Sample LSTM architecture



Model: "sequential_7"

Layer (type)	Output Shape	Param #
lstm_28 (LSTM)	(None, 180, 50)	10400
dropout_28 (Dropout)	(None, 180, 50)	0
lstm_29 (LSTM)	(None, 180, 50)	20200
dropout_29 (Dropout)	(None, 180, 50)	0
lstm_30 (LSTM)	(None, 180, 50)	20200
dropout_30 (Dropout)	(None, 180, 50)	0
lstm_31 (LSTM)	(None, 50)	20200
dropout_31 (Dropout)	(None, 50)	0
dense_7 (Dense)	(None, 1)	51

Total params: 71,051
Trainable params: 71,051
Non-trainable params: 0

Figure 7:RNN architecture of the optimized model

3.1 LOSS METRIC

The model uses ***mean squared error*** as a loss metric. Mean squared error is used to tell how close a regression line is to a set of points. It also gives more weight to larger differences.

3.2 ACTIVATION FUNCTION

Sigmoid Function:

Since every neuron must predict a value within the range of [0,1], we chose sigmoid activation function. The sigmoid function can be represented as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

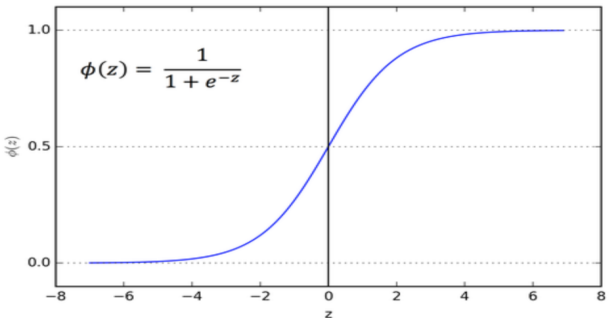


Figure 8: Sigmoid Activation function

Tanh Function:

Tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped).

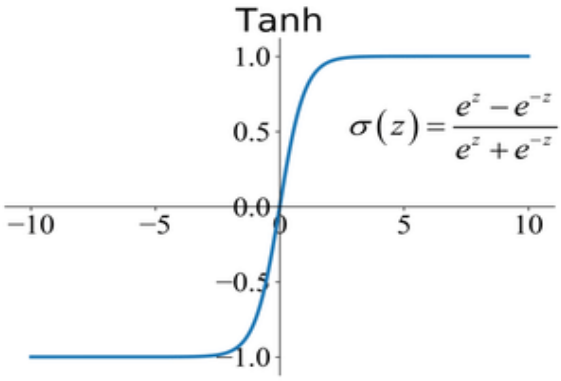


Figure 9: Tanh Activation function

3.3 OPTIMIZER

The model uses ***Adam*** optimizer.

- **Adam optimizer:**

  Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using the moving average of the gradient.

$$For\ each\ Parameter\ w^j$$
$$(j\ subscript\ dropped\ for\ clarity)$$
$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$
$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$
$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$
$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$\eta : Initial\ Learning\ rate$
$g_t : Gradient\ at\ time\ t\ along\ \omega^j$
$\nu_t : Exponential\ Average\ of\ gradients\ along\ \omega_j$
$s_t : Exponential\ Average\ of\ squares\ of\ gradients\ along\ \omega_j$
$\beta_1, \beta_2 : Hyperparameters$

3.4 SELECTING INPUT FEATURES

Initially, the model was trained on High, Low, Open, Close, S&P 500 PE Ratio, Volume, Adj Close. The accuracy of the model turned out to be around 53%. So, correlation between the features were calculated and can be seen below.

	Adj Close	Close	High	Low	Open	Volume	S&P 500 PE Ratio
Adj Close	1.000000	1.000000	0.999954	0.999958	0.998869	0.788650	0.326397
Close	1.000000	1.000000	0.999954	0.999958	0.998869	0.788650	0.326397
High	0.999954	0.999954	1.000000	0.999928	0.998926	0.790033	0.327387
Low	0.999958	0.999958	0.999928	1.000000	0.998889	0.787326	0.325252
Open	0.998869	0.998869	0.998926	0.998889	1.000000	0.790070	0.335060
Volume	0.788650	0.788650	0.790033	0.787326	0.790070	1.000000	0.463905
S&P 500 PE Ratio	0.326397	0.326397	0.327387	0.325252	0.335060	0.463905	1.000000

Figure 10: Correlation table

We observe that the Shiller P/E ratio is very weakly correlated to the close price. Hence, we remove it from our feature list.

We can see that Adj Close, High, Low and Open are strongly correlated to the close price. Hence, we add another column average where we calculate the mean of High and Low prices. Now, we trained the RNN on Adj Close, Average and Open. The accuracy of the model increased by 4%.

We also trained the RNN with Adj Close price as the only. We observe that the univariate model behaved far better with an accuracy of 97.7%. Hence, we chose this univariate model as our Optimized RNN. Please refer to the appendix for more detailed explanation.

4. PYTHON CODE FOR RNN

Dataset Generation

The following code snippet depicts the conversion of the data obtained from the yahoo finance website into a dataframe using Pandas library.

```
# dataset
data = pd.read_csv('/content/finalDS.csv')
data['Date'] = pd.to_datetime(data['Date'],format='%Y-%m-%d')
data.head()
```

	Unnamed: 0	Adj Close	Close	Date	High	Low	Open	Volume	S&P 500 PE Ratio
0	0	3756.07	3756.07	2020-12-31	3760.20	3726.88	3733.27	3172510000	37.850000
1	1	3732.04	3732.04	2020-12-30	3744.63	3730.21	3736.19	3145200000	37.842333
2	2	3727.04	3727.04	2020-12-29	3756.12	3723.31	3750.01	3387030000	37.834667
3	3	3735.36	3735.36	2020-12-28	3740.51	3723.03	3723.03	3527460000	37.827000
4	4	3703.06	3703.06	2020-12-24	3703.82	3689.32	3694.03	1885090000	37.796333

Model Initialization and training

In the code snippet below the model has been initialized with an LSTM layer followed by three other LSTM layers which also employ dropout to improve efficiency. The last layer is followed by a Dense layer. The sliding window is of size 180. The loss used is mean squared error loss.Adam optimizer has also been used.

```
# Training the model
with tpu_strategy.scope():
  model = Sequential()
  #Adding the first LSTM layer and some Dropout regularisation
  model.add(LSTM(units = 50, return_sequences = True, input_shape = (X_train.shape[1], 1)))
  model.add(Dropout(0.2))
  # Adding a second LSTM layer and some Dropout regularisation
  model.add(LSTM(units = 50, return_sequences = True))
  model.add(Dropout(0.2))
  # Adding a third LSTM layer and some Dropout regularisation
  model.add(LSTM(units = 50, return_sequences = True))
  model.add(Dropout(0.2))
  # Adding a fourth LSTM layer and some Dropout regularisation
  model.add(LSTM(units = 50))
  model.add(Dropout(0.2))
  # Adding the output layer
  model.add(Dense(units = 1))

  # Compiling the RNN
  model.compile(optimizer = 'adam', loss = 'mean_squared_error', metrics=[tf.keras.metrics.TruePositives()])

  model.summary()

  # Fitting the RNN to the Training set

  model.fit(X_train, y_train, epochs =100,batch_size=32)
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_8 (LSTM)                (None, 180, 50)           10400
_________________________________________________________________
dropout_8 (Dropout)          (None, 180, 50)           0
_________________________________________________________________
lstm_9 (LSTM)                (None, 180, 50)           20200
_________________________________________________________________
dropout_9 (Dropout)          (None, 180, 50)           0
_________________________________________________________________
lstm_10 (LSTM)               (None, 180, 50)           20200
_________________________________________________________________
dropout_10 (Dropout)         (None, 180, 50)           0
_________________________________________________________________
lstm_11 (LSTM)               (None, 50)                20200
_________________________________________________________________
dropout_11 (Dropout)         (None, 50)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 51
=================================================================
Total params: 71,051
Trainable params: 71,051
Non-trainable params: 0
```

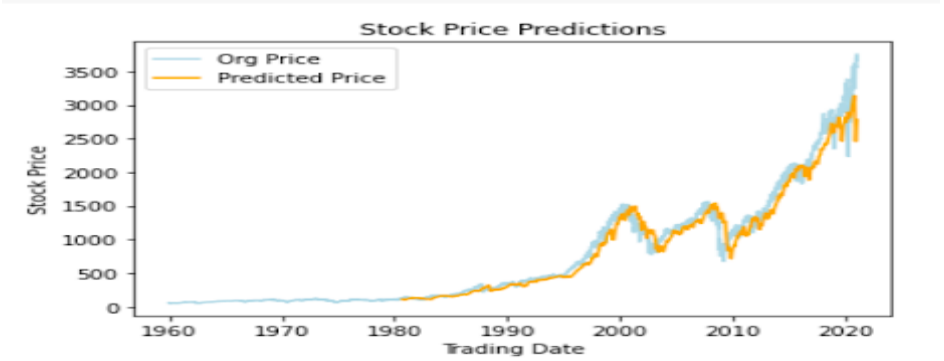
Generating predictions

The above trained model has been used to generate predictions which are then used to evaluate the performance of the model.

```
TrainPredictions=scaled_data.inverse_transform(model.predict(X_train))
TestPredictions=scaled_data.inverse_transform(model.predict(X_test))

FullDataPredictions=np.append(TrainPredictions, TestPredictions)
FullDataOrig=data[TimeSteps:]
```

```
Xt=[]
yt=[]
test1980 = np.array(data['Close'].values[:10340])
test1980 = scaled_data.transform(test1980.reshape(-1,1))
for i in range(window,10340,1):
  Xt.append(test1980[i-window:i])
  yt.append(test1980[i])
Xt = np.array(Xt)
Xt=Xt.reshape(Xt.shape[0],Xt.shape[1], 1)
predicted_Price = model.predict(Xt)
predicted_Price = scaled_data.inverse_transform(predicted_Price)
predicted_Price
```



Calculation of performance metrics

The performance metrics used are prediction error and accuracy. The prediction error is calculated as :

prediction error= (predicted price - original price)/original price.

The prediction error that is calculated is plotted as a graph against the timestamps which is displayed in **Figure 11**.

```
# Making predictions on test data
predicted_Price = model.predict(Xt)
predicted_Price = scaled_data.inverse_transform(predicted_Price)

# Getting the original price values for testing data
orig=yt
orig=scaled_data.inverse_transform(yt)

# Accuracy of the predictions

prediction_error = (predicted_Price-orig)/(orig)
print('Prediction error', prediction_error)
print('Accuracy:', 100 - (100*(abs(orig-predicted_Price)/orig)).mean())
plt.figure()


plt.plot(data['Date'].values[:predicted_Price.shape[0]],prediction_error, color = 'orange', label = 'Prediction error')
plt.ylim((0,max(prediction_error)))
plt.title('Graph of prediction error Vs timestamp for Noiseless data')
plt.xlabel('Timestamp')
plt.ylabel('Prediction error')

plt.legend()
plt.show()
```

Plotting the Prediction error graph

```
prediction_error = (predicted_Price-orig)/(orig)
print('Prediction error', prediction_error)
print('Accuracy:', 100 - (100*(abs(orig-predicted_Price)/orig)).mean())
plt.figure()


plt.plot(data['Date'].values[:predicted_Price.shape[0]],prediction_error, color = 'orange', label = 'Prediction error')
plt.ylim((0,max(prediction_error)))
plt.title('Graph of prediction error Vs timestamp for Noiseless data')
plt.xlabel('Timestamp')
plt.ylabel('Prediction error')

plt.legend()
plt.show()
```

Introducing noise to input data

The input data are perturbed by adding additive gaussian noise. The following code snippets implements that. The prediction error for each standard deviation is calculated and has been tabulated in **Figure 15**

```
#Introducing noise to the data--
dates = data["Date"].tolist()
data.drop(columns=["Date"], index=1, inplace=True)
clean_signal1 = pd.DataFrame(data)
clean_signal2 = pd.DataFrame(data)
clean_signal3 = pd.DataFrame(data)
clean_signal4 = pd.DataFrame(data)
clean_signal5 = pd.DataFrame(data)
clean_signal6 = pd.DataFrame(data)
clean_signal7 = pd.DataFrame(data)
clean_signal8 = pd.DataFrame(data)
clean_signal9 = pd.DataFrame(data)

import random
random.seed(1)

#-----------------------------------------inducing noise with std dev 0.001-----------------------------
rands=random.sample(range(0,1510), 144)


for  index, row in clean_signal1.loc[rands].iterrows():
    mu, sigma = 0, 0.001
    noise1= np.random.normal(mu, sigma)
    row=row+noise1
    #print(row)

#-------------------------------inducing noise with std dev 0.002-----------------------------------
random.seed(2)
rands1=random.sample(range(0,1510), 144)


for  index, row in clean_signal2.loc[rands1].iterrows():

    mu, sigma = 0, 0.002
    noise2= np.random.normal(mu, sigma)
    row=row+noise2
 #---------------------------inducing noise with std dev 0.003-----------------------------
random.seed(3)
rands2=random.sample(range(0,1510), 144)
for  index, row in clean_signal3.loc[rands2].iterrows():
    mu, sigma = 0, 0.003
    noise3= np.random.normal(mu, sigma)
    row=row+noise3
#---------------------------inducing noise with std dev 0.005-----------------------------

random.seed(4)
rands3=random.sample(range(0,1510), 144)
for  index, row in clean_signal4.loc[rands3].iterrows():
    mu, sigma = 0, 0.005
    noise4= np.random.normal(mu, sigma)
    row=row+noise4
```

```
#------------------------------inducing noise with std dev 0.01--------------------------
random.seed(5)
rands4=random.sample(range(0,1510), 144)
for  index, row in clean_signal5.loc[rands4].iterrows():
    mu, sigma = 0, 0.01
    noise5= np.random.normal(mu, sigma)
    row=row+noise5


#------------------------------inducing noise with std dev 0.02--------------------------
random.seed(6)
rands5=random.sample(range(0,1510), 144)
for  index, row in clean_signal6.loc[rands5].iterrows():
    mu, sigma = 0, 0.02
    noise6= np.random.normal(mu, sigma)
    row=row+noise6

#------------------------------inducing noise with std dev 0.03--------------------------
random.seed(7)
rands6=random.sample(range(0,1510), 144)
for  index, row in clean_signal7.loc[rands6].iterrows():
    mu, sigma = 0, 0.03
    noise7= np.random.normal(mu, sigma)
    row=row+noise7
#------------------------------inducing noise with std dev 0.05--------------------------
random.seed(8)
rands7=random.sample(range(0,1510), 144)
for  index, row in clean_signal8.loc[rands7].iterrows():
    mu, sigma = 0, 0.05
    noise8= np.random.normal(mu, sigma)
    row=row+noise8

#------------------------------inducing noise with std dev 0.1--------------------------
random.seed(0)
rands8=random.sample(range(0,1510), 144)
for  index, row in clean_signal9.loc[rands8].iterrows():
    mu, sigma = 0, 0.1
    noise9= np.random.normal(mu, sigma)
    row=row+noise9


#clean_signal=clean_signal.sample(frac=0.10)
noisy_signal1 = clean_signal1
noisy_signal2 = clean_signal2
noisy_signal3 = clean_signal3
noisy_signal4 = clean_signal4
noisy_signal5 = clean_signal5
noisy_signal6 = clean_signal6
noisy_signal7 = clean_signal7
noisy_signal8 = clean_signal8
noisy_signal9 = clean_signal9

#print(clean_signal)
```

Tabulating the prediction error

The prediction error has been tabulated in an excel file. As the table has a large dimension, only a portion of the **table** has been displayed in **Figure 15**

```
prediction_error=prediction_error.tolist()
prediction_error1=prediction_error1.tolist()
prediction_error2=prediction_error2.tolist()
prediction_error3=prediction_error3.tolist()
prediction_error4=prediction_error4.tolist()
prediction_error5=prediction_error5.tolist()
prediction_error6=prediction_error6.tolist()
prediction_error7=prediction_error7.tolist()
prediction_error8=prediction_error8.tolist()


data_noisy = {'stddev0.001': prediction_error, 'stddev0.002': prediction_error1, 'stddev0.003': prediction_error2, 'stddev0.005': prediction_error3, 'stddev0.01': predi
df = pd.DataFrame(data=data_noisy)
pd.set_option('display.max_columns', None)

print(df)
```

5. PERFORMANCE EVALUATION FOR UNOPTIMIZED AND OPTIMIZED RNN

Due to the usage of Adam optimizer and suitable number of epochs which is **100** in our model, the prediction error is low compared to the unoptimized model. In the case of **100 epochs**, the accuracy for the unoptimized RNN is **95.14** and for optimized it is **97.72** .

Unoptimized RNNs Output Graphs for noiseless data

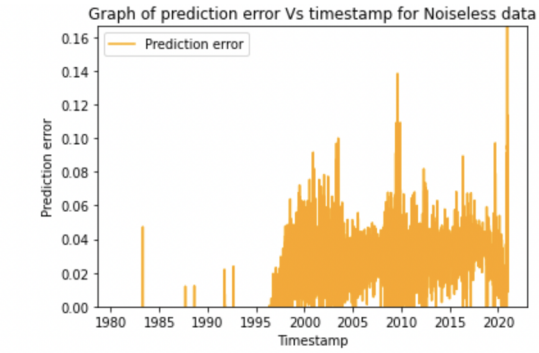


Figure 11: Prediction error vs timestamp

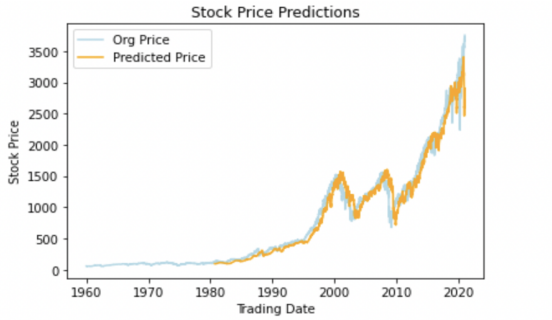


Figure 12: stock price predictions

Optimized RNNs Output Graphs for noiseless data

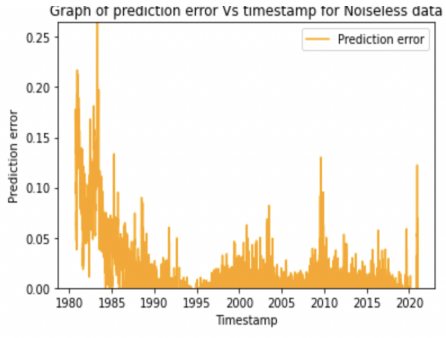


Figure 13: Prediction error vs timestamp

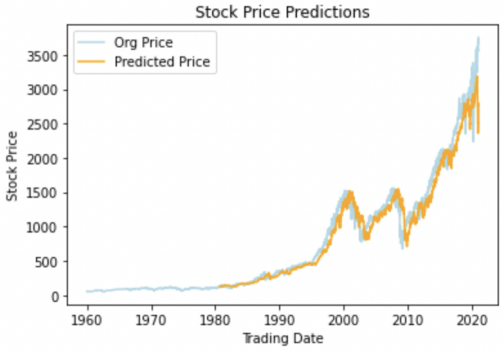


Figure 14: Stock price predictions

6. Optimized RNNs Output Graphs for noisy data

The images were then perturbed with gaussian noise and then the images were then tested. Following which **prediction error** and **timestamp** values for each standard deviation were calculated and tabulated. 9 graphs were also plotted with **timestamp** on the **abscissa** and the **prediction error** on the **ordinate** axis for each **error level.** As it was asked to plot prediction error for each error level in one graph.

Although different levels of noises were added it was observed that there were miniscule changes in the prediction error values. Furthermore, as the standard deviation of noise level increased, the prediction error values had minute changes which possibly indicates that the changes in the stock prices were so minute.

The values of the prediction with respect to each standard deviation were tabulated.As the table has a large dimension, only a portion of the **table** has been displayed in **Figure 15.**

	stddev0.001	stddev0.002	stddev0.003	stddev0.005	stddev0.01	stddev0.02	stddev0.03	stddev0.05	stddev0.1
0	[0.0095573441603	[0.0095573441603	[0.0095573441603	[0.0095573441603	[0.0095573441603	[0.0095573441	[0.0095573441	[0.00955734416(	[0.00955734416034085]
1	[0.0110636789103	[0.0110636789103	[0.0110636789103	[0.0110636789103	[0.0110636789103	[0.0110636789	[0.0110636789	[0.01106367891(	[0.01106367891030864 1]
2	[0.0134867374799	[0.0134867374799	[0.0134867374799	[0.0134867374799	[0.0134867374799	[0.0134867374	[0.0134867374	[0.01348673747(	[0.013486737479966263]
3	[-0.0024600363101	[-0.0024600363101	[-0.0024600363101	[-0.0024600363101	[-0.0024600363101	[-0.002460036	[-0.002460036	[-0.00246003631	[-0.0024600363101309965]
4	[-0.0259977959976	[-0.0259977959976	[-0.0259977959976	[-0.0259977959976	[-0.0259977959976	[-0.025997795	[-0.025997795	[-0.02599779599	[-0.025977959976806 64]
5	[-0.0083398709593	[-0.0083398709593	[-0.0083398709593	[-0.0083398709593	[-0.0083398709593	[-0.008339870	[-0.008339870	[-0.00833987095	[-0.008339870959348494]
6	[0.0044323675951	[0.0044323675951	[0.0044323675951	[0.0044323675951	[0.0044323675951	[0.0044323675	[0.0044323675	[0.00443236759(	[0.004432367595180883]
7	[-0.0043323562803	[-0.0043323562803	[-0.0043323562803	[-0.0043323562803	[-0.0043323562803	[-0.004332356	[-0.004332356	[-0.00433235628	[-0.004332356280393932]
8	[-0.0184770545116	[-0.0184770545116	[-0.0184770545116	[-0.0184770545116	[-0.0184770545116	[-0.018477054	[-0.018477054	[-0.01847705451	[-0.018477054511666313]
9	[-0.0177432094587	[-0.0177432094587	[-0.0177432094587	[-0.0177432094587	[-0.0177432094587	[-0.017743209	[-0.017743209	[-0.01774320945	[-0.017743209458789513]
10	[-0.0088836748905	[-0.0088836748905	[-0.0088836748905	[-0.0088836748905	[-0.0088836748905	[-0.008883674	[-0.008883674	[-0.00888367489	[-0.008883674890580572]
11	[0.0033282059898	[0.0033282059898	[0.0033282059898	[0.0033282059898	[0.0033282059898	[0.0033282059	[0.0033282059	[0.00332820598(	[0.003328205989897833 7]
12	[0.0109302194937	[0.0109302194937	[0.0109302194937	[0.0109302194937	[0.0109302194937	[0.0109302194	[0.0109302194	[0.01093021949(	[0.010930219493727371]
13	[0.0092846242382	[0.0092846242382	[0.0092846242382	[0.0092846242382	[0.0092846242382	[0.0092846242	[0.0092846242	[0.00928462423(	[0.009284624238287185]
14	[-0.0005500412267	[-0.0005500412267	[-0.0005500412267	[-0.0005500412267	[-0.0005500412267	[-0.000550041	[-0.000550041	[-0.00055004122	[-0.0005500412267349898]
15	[-0.0120878831082	[-0.0120878831082	[-0.0120878831082	[-0.0120878831082	[-0.0120878831082	[-0.012087883	[-0.012087883	[-0.01208788310	[-0.012087883108229378]
16	[-0.0054610061218	[-0.0054610061218	[-0.0054610061218	[-0.0054610061218	[-0.0054610061218	[-0.005461006	[-0.005461006	[-0.00546100612	[-0.005461006121833157]
17	[-0.0052442806332	[-0.0052442806332	[-0.0052442806332	[-0.0052442806332	[-0.0052442806332	[-0.005244280	[-0.005244280	[-0.00524428063	[-0.005244280633210457]
18	[-0.0047761658716	[-0.0047761658716	[-0.0047761658716	[-0.0047761658716	[-0.0047761658716	[-0.004776165	[-0.004776165	[-0.00477616587	[-0.004776165871676574]
19	[0.0029818201310	[0.0029818201310	[0.0029818201310	[0.0029818201310	[0.0029818201310	[0.0029818201	[0.0029818201	[0.00298182013(	[0.0029818201310814977]
20	[-0.0051794063584	[-0.0051794063584	[-0.0051794063584	[-0.0051794063584	[-0.0051794063584	[-0.005179406	[-0.005179406	[-0.00517940635	[-0.005179406358492455 5]
21	[0.0002713925728	[0.0002713925728	[0.0002713925728	[0.0002713925728	[0.0002713925728	[0.0002713925	[0.0002713925	[0.00027139257(	[0.0002713925728819057]
22	[0.0087107388066	[0.0087107388066	[0.0087107388066	[0.0087107388066	[0.0087107388066	[0.0087107388	[0.0087107388	[0.00871073880(	[0.008710738806636354]
23	[0.0025241105164	[0.0025241105164	[0.0025241105164	[0.0025241105164	[0.0025241105164	[0.0025241105	[0.0025241105	[0.00252411051(	[0.0025241105164474486]

Figure 15: Portion of Table for prediction error for standard deviations

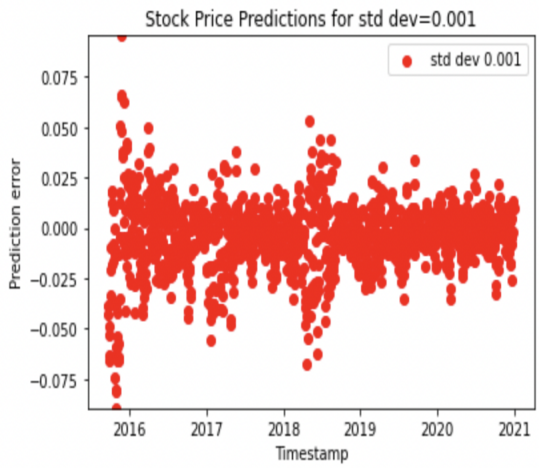


Figure 16: Prediction error vs timestamp graph for standard deviation=0.002

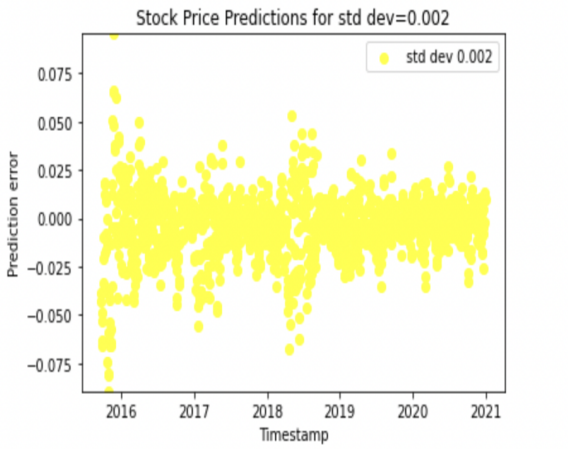


Figure 17: Prediction error vs timestamp graph for standard deviation=0.001

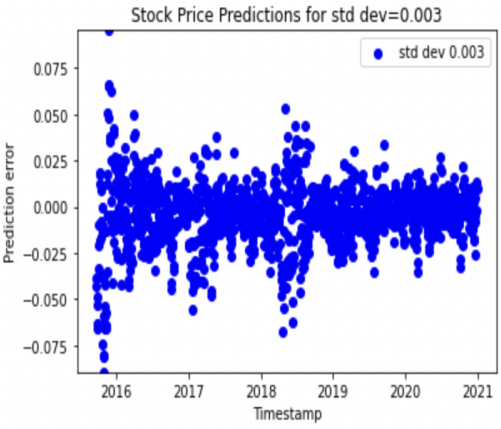


Figure 18: Prediction error vs timestamp graph for standard deviation=0.003

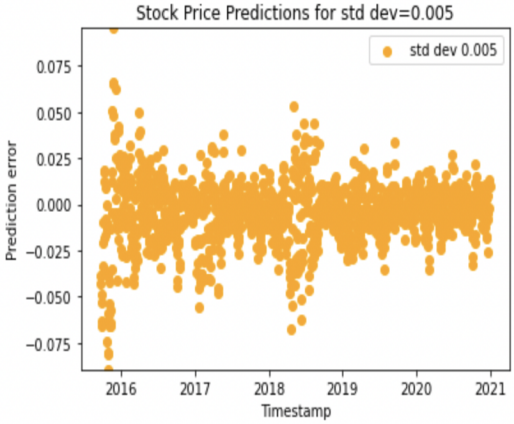
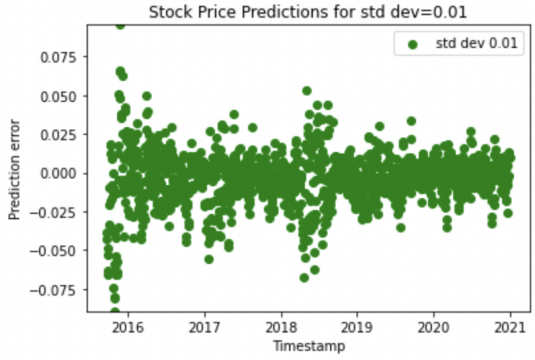


Figure 19: Prediction error vs timestamp graph for standard deviation=0.005



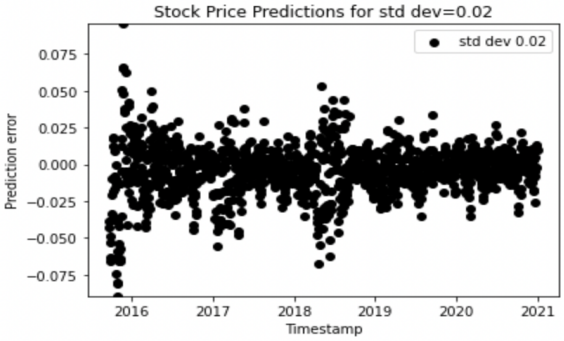
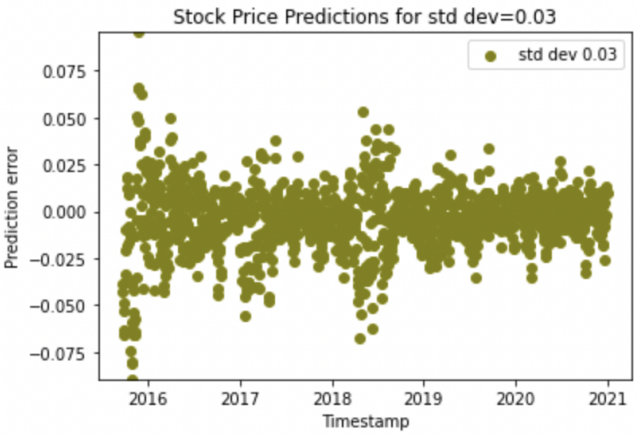


Figure 20: Prediction error vs timestamp graph for standard deviation=0.01

Figure 21: Prediction error vs timestamp graph for standard deviation=0.02



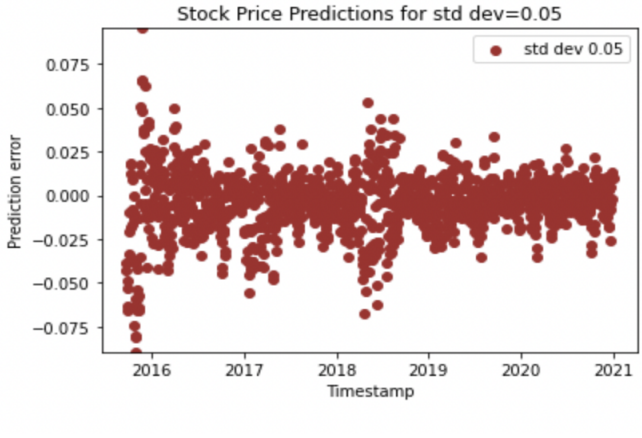


Figure 22: Prediction error vs timestamp graph for standard deviation=0.03

Figure 23: Prediction error vs timestamp graph for standard deviation=0.05

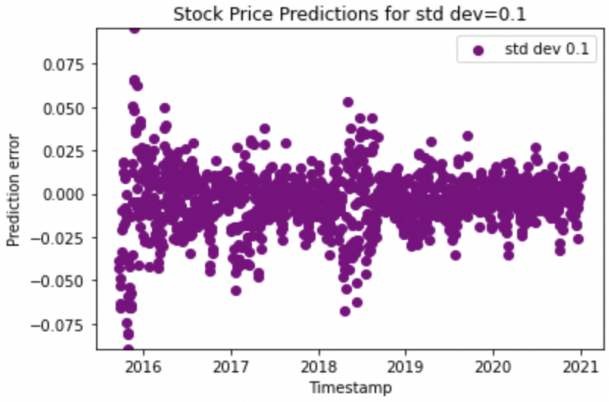


Figure 24: Prediction error vs timestamp graph for standard deviation=0.1

7. DISCUSSION

Univariate Model

This focuses on a single dependent variable. The basic assumption behind the univariate prediction approach is that the value of a time-series at time-step t is closely related to the values at the previous time-steps t-1, t-2, t-3 and so on.
Univariate models are easier to develop than multivariate models. The dependent variable in stock market forecasting is usually the closing or opening price of a finance asset. A forecasting model that is trained solely on the basis of price developments attempts.

Multivariate Model

Time series forecasting is about estimating the future value of a time series on the basis of past data. Many time series problems can be solved by looking at a single step in the future. Multi-step time series prediction models the distribution of future values of a signal over a prediction horizon. This approach predicts multiple output values at the same time which is forecasting approach to predict the further course of gradually rising sine wave. In addition, many of these variables are interdependent, which makes statistical modeling even are more complex.

From the above overlay in ***Figure 3*** it implies that shiller P/E ratio is weakly correlated. So, the LSTM model is built using 'Close'. Univariate and 2 multivariate models are trained. In the multivariate model which used average and adjacent close as input features, the accuracy of the model was **53.4** percent which is shown in the appendix.The input features used for this model were average and adjacent close. Average feature was calculated as the midpoint of high and low attributes.In the multivariate model which used all the input features, the accuracy of the model was **53.9** percent which is shown in the appendix.

It can be observed from ***figure 10*** that the **correlation** between the feature columns is not very high. Since this **correlation** is **less**, the **resulting model** that has been built does **not** have a **high accuracy**. As this correlation affects the accuracy of the model. So, the **model** here is an **univariate model** which uses the **'Close'** column input feature. Compared to the other correlation of other columns, Close, Adj.Close, High have **relatively higher correlation** and the accuracy of the model was **97.72** percent.

Furthering the discussion, the performance of the model has been evaluated for noisy data. Additive gaussian noise has been introduced at different standard deviation levels and the model was tested on this noisy data. The predictions for different noise levels were similar as the noise introduced was small. So, very minute changes were observed. To improve this, perhaps a

different noise could have been introduced. Also, to improve the accuracy, a sliding window of a different size can be used as well. Also, a multivariate implementation can be used rather than the univariate model. An attempt was made to use the multivariate model which is discussed in the **Appendix**.

8. Appendix

Multivariate model built using Average and Adjacent close as input features

```
inp_features =  data[['Average','Adj Close']].values
inp_data = inp_features
inp_data.shape
```

```
predicted_value.shape
#orig = inp_data[lookback:test_size+(2*lookback),1]
orig.shape
pred=[]
for i in range(0,predicted_value.shape[0]):
  pred_error = (predicted_value[i]-orig[i])/orig[i]
  pred.append(pred_error)
print('Accuracy:', 100 - (100*(abs(orig-predicted_value)/orig)).mean())
```

```
Accuracy: 53.44582708507084
```

Multivariate model built using all the input features in the dataset

```
inp_features =  data[['High','Low','Average','Open','Volume','S&P 500 PE Ratio','Adj Close']].value
inp_data = inp_features
inp_data.shape
```

```
predicted_value.shape
#orig = inp_data[lookback:test_size+(2*lookback),1]
orig.shape
pred=[]
for i in range(0,predicted_value.shape[0]):
  pred_error = (predicted_value[i]-orig[i])/orig[i]
  pred.append(pred_error)
print('Accuracy:', 100 - (100*(abs(orig-predicted_value)/orig)).mean())
```

```
Accuracy: 53.93245129245992
```