LSTM ON MISSING DATA.

Import Libraries

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
pd.set_option('display.float_format', lambda x: '%.4f' % x)
import seaborn as sns
sns.set_context("paper", font_scale=1.5)
sns.set_style('darkgrid')
import warnings
warnings.filterwarnings('ignore')
from time import time
import matplotlib.ticker as tkr
from scipy import stats
from statsmodels.tsa.stattools import adfuller
from sklearn import preprocessing
from statsmodels.tsa.stattools import pacf
%matplotlib inline
import math
import pickle
import keras
from keras.models import Sequential
from keras.models import model_from_json
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from keras.layers import *
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from keras.callbacks import EarlyStopping
import pydot_ng as pydot
from keras.utils import plot_model
import time
```

```
Using TensorFlow backend.
```

Load the data

In [2]:

```
df = pd.read_csv('missing_data.csv', header=None, low_memory=False)
```

In [3]:

```
df.columns
```

Out[3]:

```
Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], dtype='int64')
```

Here, we picked one column to represent the data since they are pretty much alike

In [4]:

```
df['2nd']=df[2]
```

In [5]:

```
df=df['2nd']
```

Finding the missing values and dropping them

In [6]:

```
df.isnull().sum()
```

Out[6]:

```
44725
```

In [7]:

```
df.dropna(inplace=True)
```

Feature extraction using timesteps

We are using the previous data values to determine the next one.

so the value at (t-1)s will be used to determine value at (t)s

we are employing 12 timesteps, so the previous 12 values will be used to determine the 13th values.

In [8]:

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

Normalisation

Normalisation-this helps the model work better as they dont understand the context of the data, ages of people and heights of people are different context which the model cant understand. normalisation gives the values a particular to help the model understand the limits and context of the data.

In [9]:

```
dataset = df.values #numpy.ndarray
dataset = dataset.astype('float32')
dataset = np.reshape(dataset, (-1, 1))
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
train_size = int(len(dataset) * 0.80)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
```

we also performed some splitting activities to divide the data so we can test the model with a data that it has not seen.

In [10]:

```
look_back = 12
X_train, Y_train = create_dataset(train, look_back)
X_test, Y_test = create_dataset(test, look_back)
```

2D to 3D

LSTM input layer requires a 3D data. In DNN, 2D data was used as [samples,features]. it assumes the timesteps are the features but LSTM takes the timesteps into consideration. so the 3D data will now be [samples, time steps, features]

In [11]:

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

Improved LSTM Model

In [12]:

```
model = Sequential()
model.add(Bidirectional(LSTM(100, activation='relu'), input_shape=(X_train.shape[1], X_trai
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')

start=time.time()
history = model.fit(X_train, Y_train, epochs=20, batch_size=100, validation_data=(X_test, Y
                    callbacks=[EarlyStopping(monitor='val_loss', patience=10)], verbose=1,
end=time.time()
training_time=end-start
print(training_time)
```

```
Train on 1478567 samples, validate on 369632 samples
Epoch 1/20
1478567/1478567 [==============================] - 93s 63us/step - loss: 4.5
368e-04 - val_loss: 2.3186e-04
Epoch 2/20
1478567/1478567 [==============================] - 91s 62us/step - loss: 2.1
959e-04 - val_loss: 1.8939e-04
Epoch 3/20
1478567/1478567 [==============================] - 89s 60us/step - loss: 1.9
923e-04 - val_loss: 1.7985e-04
Epoch 4/20
1478567/1478567 [==============================] - 90s 61us/step - loss: 1.9
754e-04 - val_loss: 1.8023e-04
Epoch 5/20
1478567/1478567 [==============================] - 89s 60us/step - loss: 1.9
589e-04 - val_loss: 1.7914e-04
Epoch 6/20
1478567/1478567 [==============================] - 88s 59us/step - loss: 1.9
506e-04 - val_loss: 1.7818e-04
Epoch 7/20
1478567/1478567 [==============================] - 87s 59us/step - loss: 1.9
411e-04 - val_loss: 1.7841e-04
Epoch 8/20
1478567/1478567 [==============================] - 87s 59us/step - loss: 1.9
340e-04 - val_loss: 1.7992e-04
Epoch 9/20
1478567/1478567 [==============================] - 87s 59us/step - loss: 1.9
303e-04 - val_loss: 1.7946e-04
Epoch 10/20
1478567/1478567 [==============================] - 86s 58us/step - loss: 1.9
214e-04 - val_loss: 1.7480e-04
Epoch 11/20
1478567/1478567 [==============================] - 87s 59us/step - loss: 1.9
169e-04 - val_loss: 1.7824e-04
Epoch 12/20
1478567/1478567 [==============================] - 92s 62us/step - loss: 1.9
116e-04 - val_loss: 1.7273e-04
Epoch 13/20
1478567/1478567 [==============================] - 97s 66us/step - loss: 1.9
052e-04 - val_loss: 1.7207e-04
Epoch 14/20
1478567/1478567 [==============================] - 101s 68us/step - loss: 1.
8992e-04 - val_loss: 1.7149e-04
Epoch 15/20
1478567/1478567 [==============================] - 100s 68us/step - loss: 1.
8956e-04 - val_loss: 1.7077e-04
Epoch 16/20
```

```
1478567/1478567 [==============================] - 98s 66us/step - loss: 1.8
917e-04 - val_loss: 1.7044e-04
Epoch 17/20
1478567/1478567 [==============================] - 99s 67us/step - loss: 1.8
865e-04 - val_loss: 1.6980e-04
Epoch 18/20
1478567/1478567 [==============================] - 102s 69us/step - loss: 1.
8816e-04 - val_loss: 1.6910e-04
Epoch 19/20
1478567/1478567 [==============================] - 102s 69us/step - loss: 1.
8805e-04 - val_loss: 1.6871e-04
Epoch 20/20
1478567/1478567 [==============================] - 103s 70us/step - loss: 1.
8782e-04 - val_loss: 1.6853e-04
1868.269147157669
```

In [13]:

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
bidirectional_1 (Bidirection (None, 200)               90400     
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 201       
=================================================================
Total params: 90,601
Trainable params: 90,601
Non-trainable params: 0
_________________________________________________________________
```

Prediction

In [14]:

```
start=time.time()
train_predict = model.predict(X_train)
test_predict = model.predict(X_test)
# invert predictions
train_predict = scaler.inverse_transform(train_predict)
Y_train = scaler.inverse_transform([Y_train])
test_predict = scaler.inverse_transform(test_predict)
Y_test = scaler.inverse_transform([Y_test])
end=time.time()
prediction_time=end-start
print(prediction_time)
```

```
56.93779444694519
```

Measuring accuracy of the model

there is no way to use percentage accuracies to determine accuracies, instead we use MSE mean squared error, MASE mean absolute squared error. this errors simply show how far the the predicted values are from the actual values to show their competency when faced with real-world data.

In [15]:

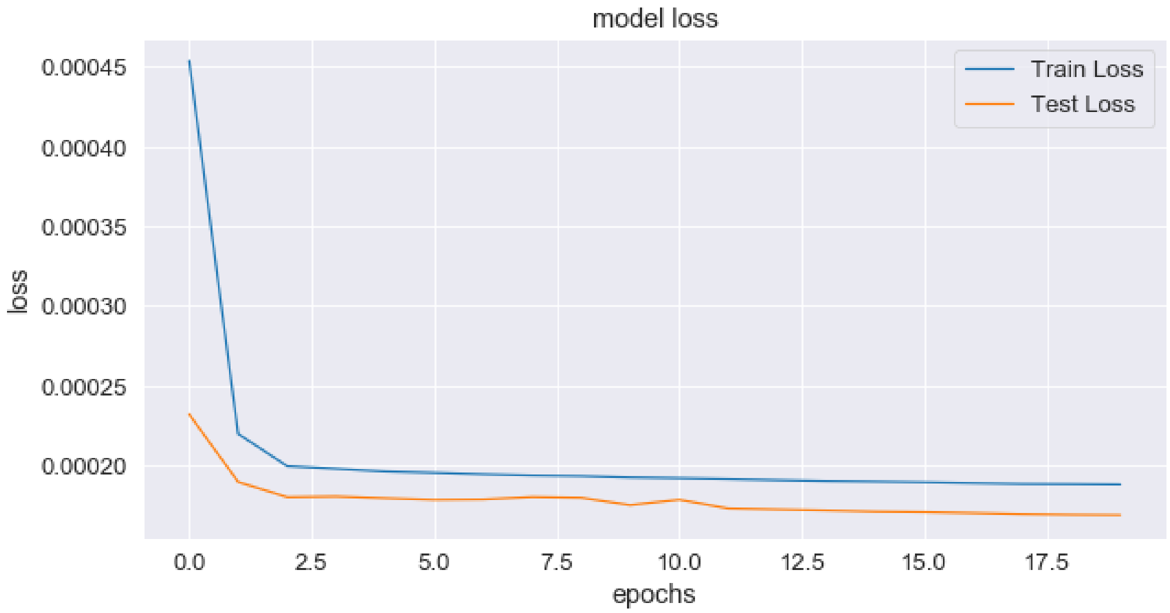
```
print('Train Mean Absolute Error:', mean_absolute_error(Y_train[0], train_predict[:,0]))
print('Train Root Mean Squared Error:',np.sqrt(mean_squared_error(Y_train[0], train_predict
print('Test Mean Absolute Error:', mean_absolute_error(Y_test[0], test_predict[:,0]))
print('Test Root Mean Squared Error:',np.sqrt(mean_squared_error(Y_test[0], test_predict[:,
```

```
Train Mean Absolute Error: 0.3641590110704591
Train Root Mean Squared Error: 2.271192596302092
Test Mean Absolute Error: 0.30613247837477237
Test Root Mean Squared Error: 2.1449815773937373
```

Model losses when training the model and testing

In [16]:

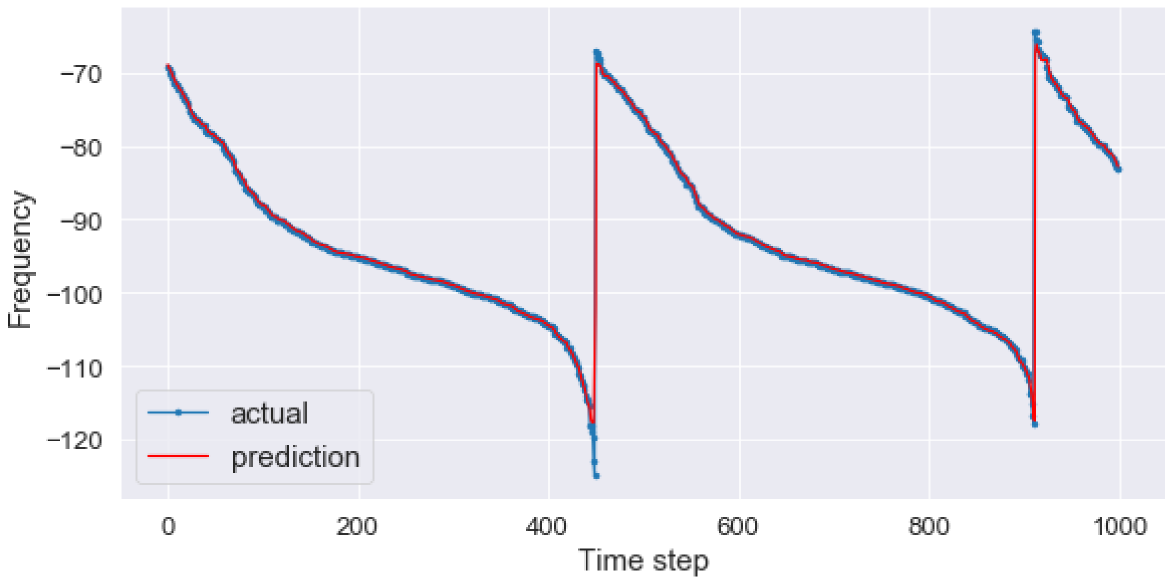
```
plt.figure(figsize=(10,5))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Test Loss')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epochs')
plt.legend(loc='upper right')
plt.show()
```



Actual values vs Predicted values

In [17]:

```
aa=[x for x in range(1000)]
plt.figure(figsize=(8,4))
plt.plot(aa, Y_train[0][:1000], marker='.', label="actual")
plt.plot(aa, train_predict[:,0][:1000], 'r', label="prediction")
# plt.tick_params(left=False, labelleft=True) #remove ticks
plt.tight_layout()
sns.despine(top=True)
plt.subplots_adjust(left=0.07)
plt.ylabel('Frequency', size=15)
plt.xlabel('Time step', size=15)
plt.legend(fontsize=15)
plt.show()
```



Saving predicted data

In [18]:

```
sed=pd.DataFrame(test_predict)
sed.to_csv('missing_improved_LSTM_predict.csv', index=False)
```

Saving the model

In [19]:

```
filename = 'missing_improved_LSTM.sav'
pickle.dump(model, open(filename, 'wb'))
```

In [20]:

```
model_json = model.to_json()
with open("missing_improved_LSTM_model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("missing_improved_LSTM_model.h5")
print("Saved model to disk")
```

```
Saved model to disk
```

Saving the normalisation model

the LSTM model was built on normalised data for better result. When attempting to predict new values, the data must be normalised before being sent to the model

In [21]:

```
filename = 'missing_improved_LSTM_scaler.sav'
pickle.dump(scaler, open(filename, 'wb'))
```

Loading the saved model

In [22]:

```
json_file = open('missing_improved_LSTM_model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("missing_improved_LSTM_model.h5")
print("Loaded model from disk")
```

```
Loaded model from disk
```

In [23]:

```
look_back = 12
X_train, Y_train = create_dataset(train, look_back)
X_test, Y_test = create_dataset(test, look_back)
X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))
X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))
train_predict = loaded_model.predict(X_train)
test_predict = loaded_model.predict(X_test)
# invert predictions
train_predict = scaler.inverse_transform(train_predict)
Y_train = scaler.inverse_transform([Y_train])
test_predict = scaler.inverse_transform(test_predict)
Y_test = scaler.inverse_transform([Y_test])
print('Train Mean Absolute Error:', mean_absolute_error(Y_train[0], train_predict[:,0]))
print('Train Root Mean Squared Error:',np.sqrt(mean_squared_error(Y_train[0], train_predict
print('Test Mean Absolute Error:', mean_absolute_error(Y_test[0], test_predict[:,0]))
print('Test Root Mean Squared Error:',np.sqrt(mean_squared_error(Y_test[0], test_predict[:,
```

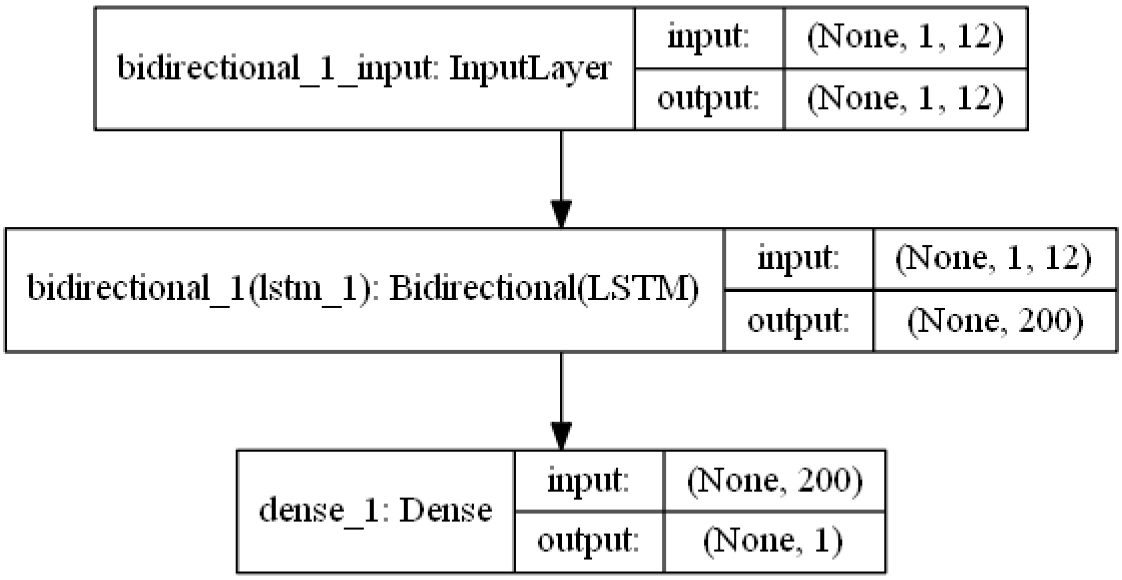
```
Train Mean Absolute Error: 0.3641590110704591
Train Root Mean Squared Error: 2.271192596302092
Test Mean Absolute Error: 0.30613247837477237
Test Root Mean Squared Error: 2.1449815773937373
```

Visualisation of the Neural Network

In [24]:

```
plot_model(model, to_file='missing_improved_LSTM.png', show_shapes=True)
```

Out[24]:



Model Parameters

- Epoch = 20
- Batch size = 100
- Hidden layers = 1
- Neurons = 100 in hidden layer, 1 in output layer

Observation	values
Train Mean Absolute Error	0.3642
Train Root Mean Squared Error	2.2711
Test Mean Absolute Error	0.3061
Test Root Mean Squared Error	2.1449
Training time	1868.2691
Prediction time	56.9378

Observation	LSTM	Improved LSTM
Train Mean Absolute Error	0.4637	0.3642
Train Root Mean Squared Error	2.3656	2.2711
Test Mean Absolute Error	0.4059	0.3061
Test Root Mean Squared Error	2.2336	2.1449
Training time	1395.2241	1868.2691
Prediction time	41.4415	56.9378

Models	Train Mean Absolute Error	Train Root Mean Squared Error	Test Mean Absolute Error	Test Root Mean Squared Error	Training time	Prediction time
DNN	0.4160	2.3057	0.3607	2.1937	374.9854	15.0671
Improved DNN	1.1289	2.5515	1.0850	2.4479	1088.0616	28.2217
LSTM	0.4637	2.3656	0.4059	2.2336	1395.2241	41.4415
Improved LSTM	0.3642	2.2711	0.3061	2.1449	1868.2691	56.9378