Artificial neural network

Import Libraries

In [1]:

```
import pandas as pd
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import seaborn as sns
import pickle
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from keras.models import model_from_json
import pydot_ng as pydot
from keras.utils import plot_model
import time
```

```
Using TensorFlow backend.
```

Load data

In [2]:

```
df = pd.read_csv('missing_data.csv', low_memory=False)
```

In [3]:

```
df = df.iloc[:,:5]
```

Below are the first 5 rows of the data.

In [4]:

```
df.head()
```

Out[4]:

	0	1	2	3	4
0	-65.746971	-66.827461	-65.604935	-63.861546	-66.703033
1	-66.542122	-67.348610	-66.031960	-64.742157	-66.823357
2	-66.703415	-67.738510	-68.254852	-65.018089	-66.908104
3	-67.461182	-68.072807	-68.260017	-66.640083	-66.984612
4	-68.103065	-68.076118	-68.277176	-66.681175	-67.161224

In [5]:

```
df.columns
```

Out[5]:

```
Index(['0', '1', '2', '3', '4'], dtype='object')
```

Checking for missing data

In [6]:

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

Out[6]:

```
0    44721
1    44725
2    44725
3    44721
4    53941
dtype: int64
```

The above means, there are

- 44721 empty spaces in column 0
- 44725 empty spaces in column 1
- 44725 empty spaces in column 2
- 44721 empty spaces in column 3
- 53941 empty spaces in column 4

Removing the empty spaces

In [7]:

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

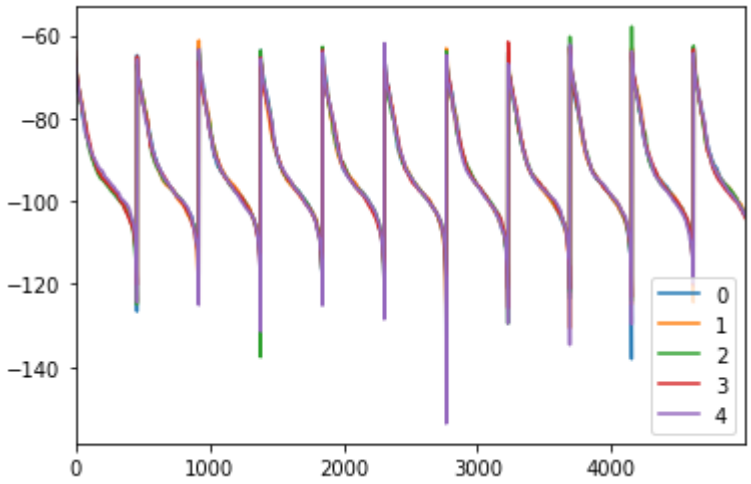
Data visualisation of the first 5000 rows

In [8]:

```
df[:5000].plot()
```

Out[8]:

```
<matplotlib.axes._subplots.AxesSubplot at 0xf717d9b390>
```



from the graph above, the values are the almost the same across all columns.so we can use one column to represent all thereby avoiding complexity

In [9]:

```
df['selected'] = df.iloc[:,2]
```

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

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.
- 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]:

```
dataset     = df['selected'].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),:]
```

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 [11]:

```
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)
```

In [12]:

```
n_steps       = 12
x,y           = create_dataset(train, n_steps)
x_test,y_test = create_dataset(test, n_steps)
```

Building ANN model

In [13]:

```
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

In [14]:

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 100)               1300
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 101
=================================================================
Total params: 1,401
Trainable params: 1,401
Non-trainable params: 0
_________________________________________________________________
```

Training the model

In [15]:

```
start=time.time()
history = model.fit(x, y,
                    batch_size=100,
                    epochs=20,
                    verbose=2,
                    validation_data=(x_test, y_test))
end=time.time()
training_time=end-start
print(training_time)
```

```
Train on 1471187 samples, validate on 367787 samples
Epoch 1/20
 - 19s - loss: 7.9978e-04 - val_loss: 3.9281e-04
Epoch 2/20
 - 19s - loss: 4.0509e-04 - val_loss: 3.6827e-04
Epoch 3/20
 - 19s - loss: 3.9144e-04 - val_loss: 3.5585e-04
Epoch 4/20
 - 18s - loss: 3.8566e-04 - val_loss: 3.3554e-04
Epoch 5/20
 - 18s - loss: 3.8362e-04 - val_loss: 3.3335e-04
Epoch 6/20
 - 19s - loss: 3.8196e-04 - val_loss: 3.2407e-04
Epoch 7/20
 - 19s - loss: 3.7898e-04 - val_loss: 3.5320e-04
Epoch 8/20
 - 18s - loss: 3.7781e-04 - val_loss: 3.3028e-04
Epoch 9/20
 - 17s - loss: 3.7843e-04 - val_loss: 3.2658e-04
Epoch 10/20
 - 17s - loss: 3.7821e-04 - val_loss: 3.3062e-04
Epoch 11/20
 - 19s - loss: 3.7517e-04 - val_loss: 3.4427e-04
Epoch 12/20
 - 19s - loss: 3.7601e-04 - val_loss: 3.2260e-04
Epoch 13/20
 - 24s - loss: 3.7569e-04 - val_loss: 3.3306e-04
Epoch 14/20
 - 21s - loss: 3.7457e-04 - val_loss: 3.2318e-04
Epoch 15/20
 - 18s - loss: 3.7349e-04 - val_loss: 3.6957e-04
Epoch 16/20
 - 18s - loss: 3.7396e-04 - val_loss: 3.2299e-04
Epoch 17/20
 - 18s - loss: 3.7391e-04 - val_loss: 3.2267e-04
Epoch 18/20
 - 18s - loss: 3.7331e-04 - val_loss: 3.2774e-04
Epoch 19/20
 - 18s - loss: 3.7338e-04 - val_loss: 3.2316e-04
Epoch 20/20
 - 19s - loss: 3.7212e-04 - val_loss: 3.4462e-04
374.9854383468628
```

Predicting the values

In [16]:

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

```
15.06707501411438
```

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 [17]:

```
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[:,
```

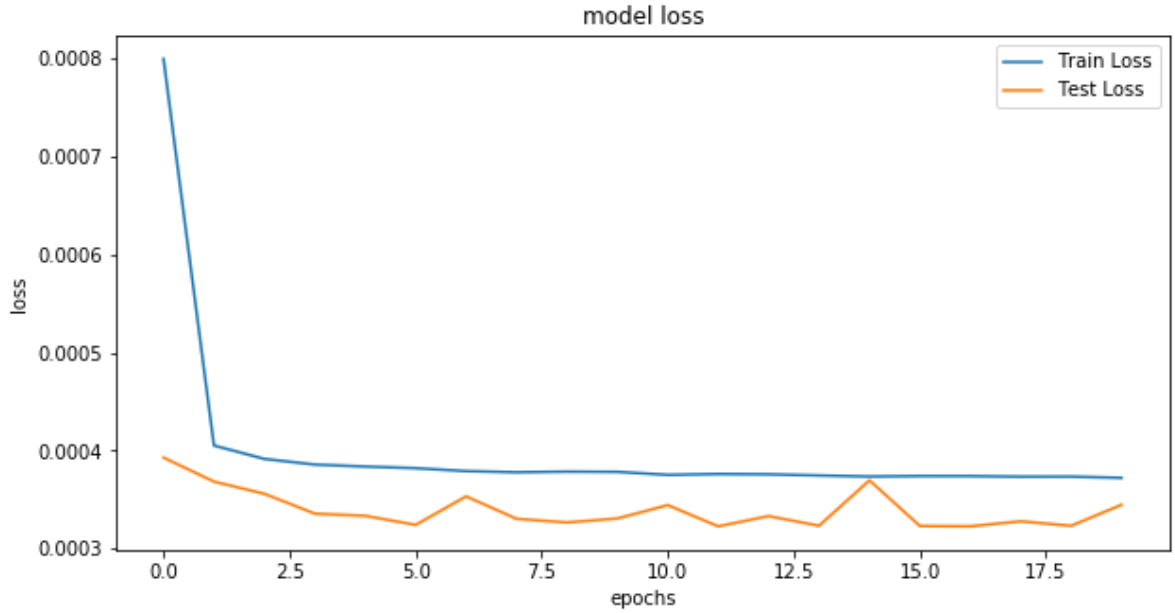
```
Train Mean Absolute Error: 0.41605550370708766
Train Root Mean Squared Error: 2.3057383348207896
Test Mean Absolute Error: 0.36072907513238733
Test Root Mean Squared Error: 2.193684896996369
```

Model loss

the graph shows the errors when training and validating. it is another metric that shows the reliability of the system. the validation error should not be higher than training loss because, the validation loss is a loss when exposed to a new data and training loss is the loss when exposed to the data that was used to train the model. When this happens, it is similar to the term we call 'CRAMMING'.

In [18]:

```
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()
```

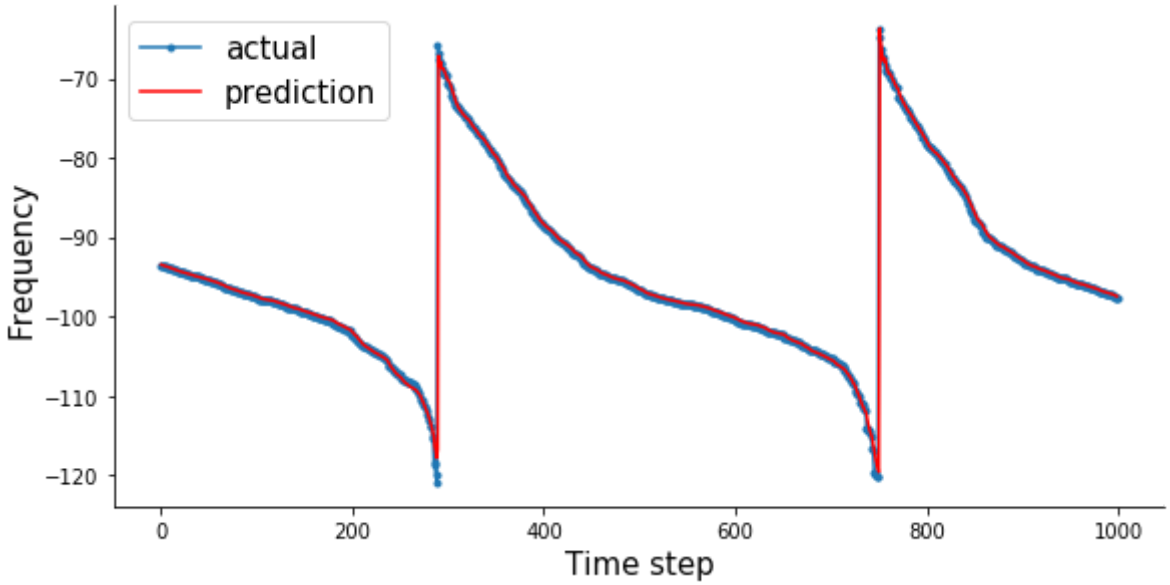


Graph of the Actual data vs the predicted

This shows how different the predicted data is from the actual data

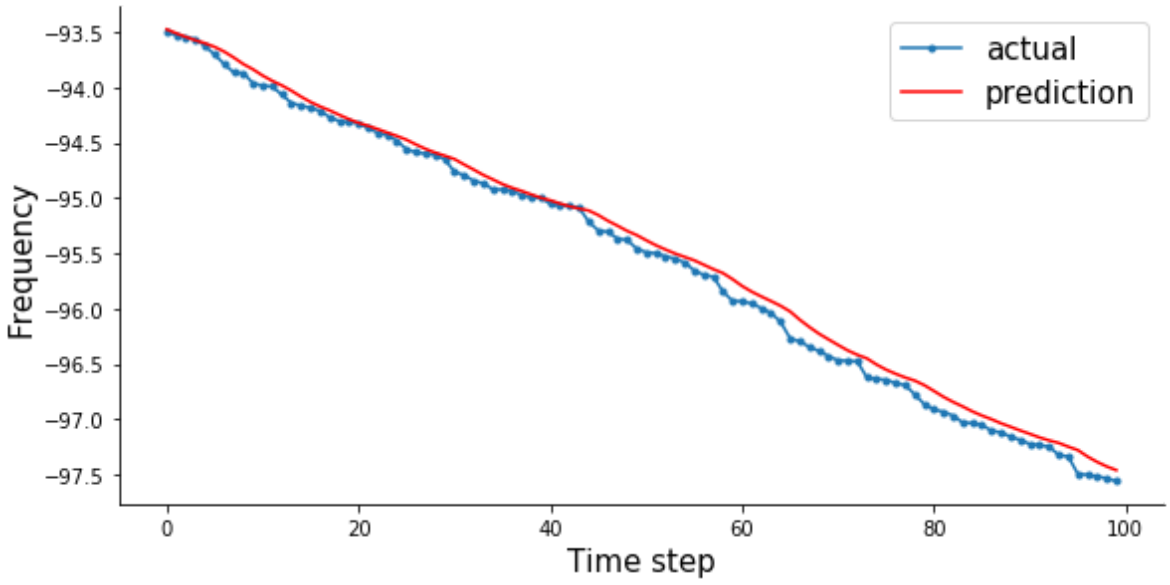
In [19]:

```
aa=[x for x in range(1000)]
plt.figure(figsize=(8,4))
plt.plot(aa, Y_test[0][:1000], marker='.', label="actual")
plt.plot(aa, test_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()
```



In [20]:

```
aa=[x for x in range(100)]
plt.figure(figsize=(8,4))
plt.plot(aa, Y_test[0][:100], marker='.', label="actual")
plt.plot(aa, test_predict[:,0][:100], '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 [21]:

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

Saving the model

In [22]:

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

In [23]:

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

```
Saved model to disk
```

Saving the normalisation model

the ANN 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 [24]:

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

Loading the saved model

In [25]:

```
json_file = open('missing_DNN_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("model.h5")
print("Loaded model from disk")
```

```
Loaded model from disk
```

In [26]:

```
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[:,
```

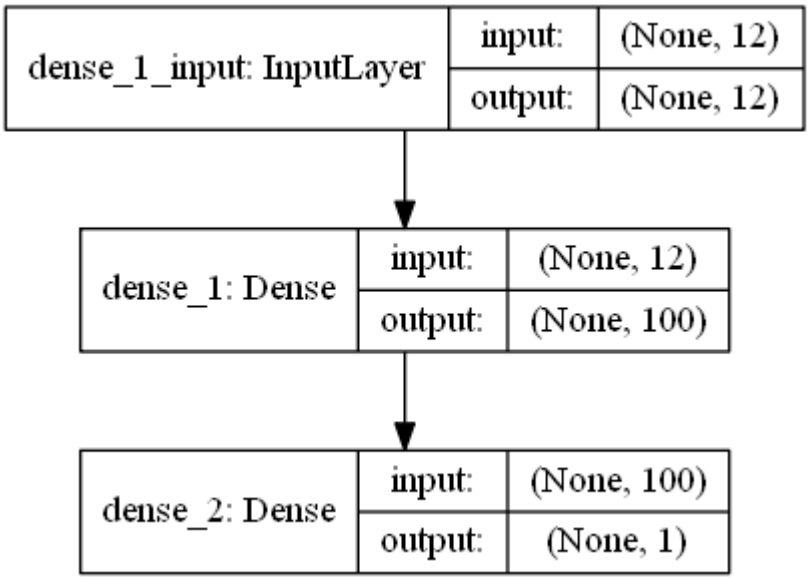
```
Train Mean Absolute Error: 0.41605550370708766
Train Root Mean Squared Error: 2.3057383348207896
Test Mean Absolute Error: 0.36072907513238733
Test Root Mean Squared Error: 2.193684896996369
```

Visualisation of the Neural Network

In [27]:

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

Out[27]:



Model Parameters

- Epoch = 20
- Batch size = 100
- Hidden layers = 1
- Dropout layers = 1
- Dropout = 0.2

- Neurons = 100 in hidden layer, 1 in output layer

Observation	values
Train Mean Absolute Error	0.4160
Train Root Mean Squared Error	2.3057
Test Mean Absolute Error	0.3607
Test Root Mean Squared Error	2.1937
Training time	374.9854
Prediction time	15.0671

Observation	DNN	Improved DNN
Train Mean Absolute Error	0.4160	1.1289
Train Root Mean Squared Error	2.3057	2.5515
Test Mean Absolute Error	0.3607	1.0850
Test Root Mean Squared Error	2.1937	2.4479
Training time	374.9854	1088.0616
Prediction time	15.0671	28.2217

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