Question 2 : Neural Networks

You have to build a neural network using Numpy. So you CANNOT use TensorFlow, PyTorch or any other library with built-in neural networks. The dataset is uploaded on LMS along with this assignment. It is a regression task. You have to predict the ’Price’ of the house.

In [1]:
```
'''
    Importing useful libraries
    numpy : working in domain of linear algebra, fourier transform, and matrices
    pandas : data analysis and associated manipulation of tabular data in DataFrames
    seaborn : making statistical graphics in Python
    matplotlib : creating static, animated, and interactive visualizations
'''
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import random
```

In [2]:
```
'''
    Importing dataset
'''
house = pd.read_csv("Assignment2_q2_dataset.csv")
```

In [3]:
```
'''
    Dataset shape
'''
print(f"Number of entries: {house.shape[0]}")
print(f"Number of features: {house.shape[1]}")
```

```
Number of entries: 14620
Number of features: 23
```

In [4]:
```
'''
    Dataset Information
'''
house.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14620 entries, 0 to 14619
Data columns (total 23 columns):
 #   Column                                 Non-Null Count  Dtype  
---  ------                                 --------------  -----  
 0   id                                     14620 non-null  int64  
 1   Date                                   14620 non-null  int64  
 2   number of bedrooms                     14620 non-null  int64  
 3   number of bathrooms                    14620 non-null  float64
 4   living area                            14620 non-null  int64  
 5   lot area                               14620 non-null  int64  
 6   number of floors                       14620 non-null  float64
 7   waterfront present                     14620 non-null  int64  
 8   number of views                        14620 non-null  int64  
 9   condition of the house                 14620 non-null  int64  
 10  grade of the house                     14620 non-null  int64  
 11  Area of the house(excluding basement)  14620 non-null  int64  
 12  Area of the basement                   14620 non-null  int64  
 13  Built Year                             14620 non-null  int64  
 14  Renovation Year                        14620 non-null  int64  
 15  Postal Code                            14620 non-null  int64  
 16  Lattitude                              14620 non-null  float64
 17  Longitude                              14620 non-null  float64
 18  living_area_renov                      14620 non-null  int64  
 19  lot_area_renov                         14620 non-null  int64  
 20  Number of schools nearby               14620 non-null  int64  
 21  Distance from the airport              14620 non-null  int64  
 22  Price                                  14620 non-null  int64  
dtypes: float64(4), int64(19)
memory usage: 2.6 MB
```

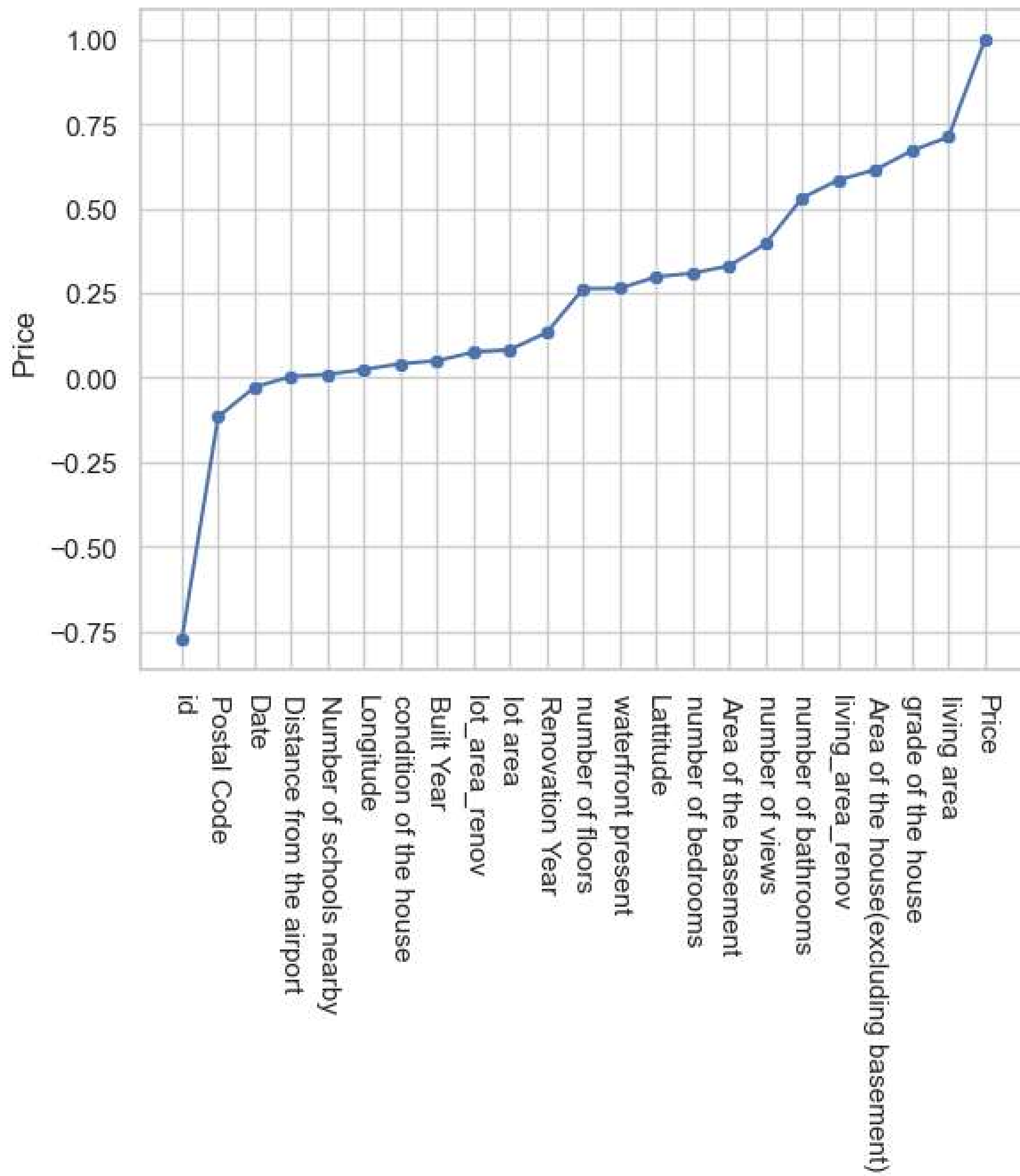
Clean the data

As we can see from the dataset information, data is pretty much clean and we do not need to perform any data cleaning steps to it. So, let's go ahead with other operations.

Preprocess the data

In [5]:

```python
'''
    Now let's see correlation of each feature with target (Price)
'''
sns.set(style="whitegrid")
sns.scatterplot(house.corr()['Price'].sort_values(ascending=True))
sns.lineplot(house.corr()['Price'].sort_values(ascending=True))
plt.xticks(rotation = 270)
plt.show()
```



In [6]:

```
'''
    Dropping out low correlated features
'''
features_dropped = ['condition of the house','Built Year','Date', 'lot area', 'Distance from the airport', 'Number of schools nearby', 'lot_area_renov', 'Lattitude', 'Longitude']
house = house.drop(features_dropped, axis=1)
print(f"Features Dropped : {features_dropped}")
```

```
Features Dropped : ['condition of the house', 'Built Year', 'Date', 'lot area', 'Distance from the airport', 'Number of schools nearby', 'lot_area_renov', 'Lattitude', 'Longitude']
```

In [7]:

```
'''
    Scaling Training and testing independent features
'''
def standardize_column(column):
    mean_val = column.mean()
    std_val = column.std()
    standardized_column = (column - mean_val) / std_val if std_val != 0 else column
    return standardized_column
```

```python
house_copy = house.copy()
for column in house_copy.columns:
    house_copy[column] = standardize_column(house[column])
```

In [8]:

```python
'''
    Splitting the dataset into Dependent and Independent Variable
'''
X = house_copy.iloc[:,:-1]
Y = house_copy.iloc[:,-1]
```

In [9]:

```python
'''
    Converting Dependent and Independent Variable into array
'''
X = np.asarray(X)
Y = np.asarray(Y)
```

In [10]:

```python
'''
    Split the data into training and testing sets
'''
def custom_train_test_split(data, labels, test_size = 0.2, random_seed = None):
    if random_seed:
        random.seed(random_seed)
    dataset = list(zip(data, labels))
    random.shuffle(dataset)
    split_index = int(len(dataset) * (1 - test_size))
    train_data, test_data = dataset[:split_index], dataset[split_index:]
    X_train, y_train = zip(*train_data)
    X_test, y_test = zip(*test_data)
    return np.array(X_train), np.array(X_test), np.array(y_train), np.array(y_test)

X_train,x_test,Y_train,y_test = custom_train_test_split(X, Y, test_size = 0.2, random_seed = 3)
```

Building Custom Neural Network Model

In [11]:

```python
'''
    Number of features
'''
input_size = X_train.shape[1]


'''
    Number of neurons in the hidden layer
'''
hidden_size = 64


'''
    Single output for regression
'''
output_size = 1


'''
    Choosing random seed to generate random data
'''
np.random.seed(42)
```

```
'''
    Randomly initialize weights for the hidden layers
'''
hidden_weights = np.random.randn(input_size,hidden_size)


'''
    Randomly initialize bias for the hidden layers
'''
hidden_bias = np.zeros(hidden_size)


'''
    Randomly initialize weights for the output layer
'''
output_weights = np.random.randn(hidden_size,output_size)


'''
    Randomly initialize bias for the output layer
'''
output_bias=np.zeros(output_size)
```

Training data into our custom Neural Network model

In [12]:

```
'''
    Training data into model
'''
learning_rate = 0.1
num_epochs = 10001
for epoch in range(num_epochs):
    #forward propagation
    hidden_layer_input = np.dot(X_train,hidden_weights)+hidden_bias
    hidden_layer_output = 1/(1 + np.exp(-hidden_layer_input))
    predictions = np.dot(hidden_layer_output,output_weights) + output_bias

    #Compute loss(mean squared error)
    loss = np.mean((predictions-Y_train.reshape(-1,1))**2)

    #Gradient of mean squared error
    output_error = 2*(predictions-Y_train.reshape(-1,1))

    #Backpropagagtion
    hidden_error = np.dot(output_error,output_weights.T)*hidden_layer_output*(1 - hidden_layer_output)
    output_weights -= learning_rate*np.dot(hidden_layer_output.T,output_error)/len(Y_train)
    output_bias -= learning_rate*np.sum(output_error)/len(Y_train)

    hidden_weights -= learning_rate*np.dot(X_train.T,hidden_error)/len(Y_train)
    hidden_bias -= learning_rate*np.sum(hidden_error)/len(Y_train)

    if epoch%100==0:
        print(f"Epoch{epoch}, Loss: {loss:.4f}")
```

```
Epoch0, Loss: 93.9044
Epoch100, Loss: 0.4286
Epoch200, Loss: 0.3573
Epoch300, Loss: 0.3168
Epoch400, Loss: 0.2917
Epoch500, Loss: 0.2731
Epoch600, Loss: 0.2576
Epoch700, Loss: 0.2443
Epoch800, Loss: 0.2325
Epoch900, Loss: 0.2221
Epoch1000, Loss: 0.2128
Epoch1100, Loss: 0.2044
Epoch1200, Loss: 0.1968
Epoch1300, Loss: 0.1899
Epoch1400, Loss: 0.1835
Epoch1500, Loss: 0.1776
Epoch1600, Loss: 0.1722
Epoch1700, Loss: 0.1671
Epoch1800, Loss: 0.1624
Epoch1900, Loss: 0.1579
Epoch2000, Loss: 0.1537
Epoch2100, Loss: 0.1497
Epoch2200, Loss: 0.1458
Epoch2300, Loss: 0.1421
Epoch2400, Loss: 0.1386
Epoch2500, Loss: 0.1351
Epoch2600, Loss: 0.1317
Epoch2700, Loss: 0.1285
Epoch2800, Loss: 0.1253
Epoch2900, Loss: 0.1221
Epoch3000, Loss: 0.1191
Epoch3100, Loss: 0.1161
Epoch3200, Loss: 0.1131
Epoch3300, Loss: 0.1103
Epoch3400, Loss: 0.1074
Epoch3500, Loss: 0.1047
Epoch3600, Loss: 0.1019
Epoch3700, Loss: 0.0993
Epoch3800, Loss: 0.0967
Epoch3900, Loss: 0.0942
Epoch4000, Loss: 0.0918
Epoch4100, Loss: 0.0895
Epoch4200, Loss: 0.0873
Epoch4300, Loss: 0.0853
Epoch4400, Loss: 0.0834
Epoch4500, Loss: 0.0816
Epoch4600, Loss: 0.0799
Epoch4700, Loss: 0.0784
Epoch4800, Loss: 0.0770
Epoch4900, Loss: 0.0757
Epoch5000, Loss: 0.0744
Epoch5100, Loss: 0.0733
Epoch5200, Loss: 0.0723
Epoch5300, Loss: 0.0713
Epoch5400, Loss: 0.0704
Epoch5500, Loss: 0.0695
Epoch5600, Loss: 0.0687
Epoch5700, Loss: 0.0680
Epoch5800, Loss: 0.0673
Epoch5900, Loss: 0.0666
```

```
Epoch6000, Loss: 0.0660
Epoch6100, Loss: 0.0654
Epoch6200, Loss: 0.0648
Epoch6300, Loss: 0.0642
Epoch6400, Loss: 0.0637
Epoch6500, Loss: 0.0632
Epoch6600, Loss: 0.0627
Epoch6700, Loss: 0.0623
Epoch6800, Loss: 0.0618
Epoch6900, Loss: 0.0614
Epoch7000, Loss: 0.0610
Epoch7100, Loss: 0.0606
Epoch7200, Loss: 0.0602
Epoch7300, Loss: 0.0598
Epoch7400, Loss: 0.0595
Epoch7500, Loss: 0.0591
Epoch7600, Loss: 0.0588
Epoch7700, Loss: 0.0584
Epoch7800, Loss: 0.0581
Epoch7900, Loss: 0.0578
Epoch8000, Loss: 0.0575
Epoch8100, Loss: 0.0572
Epoch8200, Loss: 0.0569
Epoch8300, Loss: 0.0567
Epoch8400, Loss: 0.0564
Epoch8500, Loss: 0.0561
Epoch8600, Loss: 0.0559
Epoch8700, Loss: 0.0556
Epoch8800, Loss: 0.0554
Epoch8900, Loss: 0.0551
Epoch9000, Loss: 0.0549
Epoch9100, Loss: 0.0547
Epoch9200, Loss: 0.0545
Epoch9300, Loss: 0.0542
Epoch9400, Loss: 0.0540
Epoch9500, Loss: 0.0538
Epoch9600, Loss: 0.0536
Epoch9700, Loss: 0.0534
Epoch9800, Loss: 0.0532
Epoch9900, Loss: 0.0530
Epoch10000, Loss: 0.0528
```

Evaluating our model on Test Set

In [13]:

```
'''
    Evaluating on Test Set
'''
hidden_layer_input = np.dot(x_test,hidden_weights)+hidden_bias
hidden_layer_output = 1/(1+np.exp(-hidden_layer_input))
predicted_prices = np.dot(hidden_layer_output,output_weights) + output_bias
```

In [14]:

```
'''
    Calculating Test Loss
'''
test_loss=np.mean((predicted_prices-y_test.reshape(-1,1))**2)
print(f"Test Loss: {test_loss: .4f}")
```

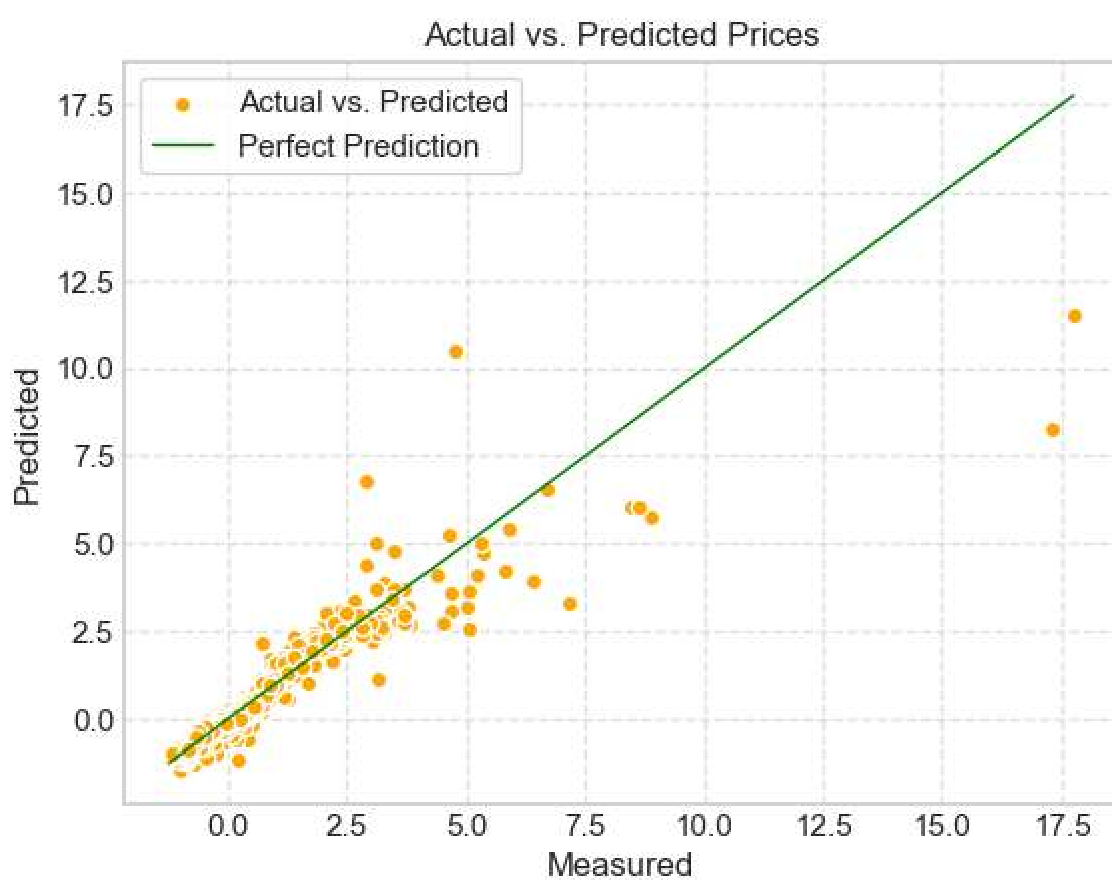
```
'''
    Calculating Mean Squared Error
'''
MSE = np.square(np.subtract(y_test,predicted_prices)).mean()
print(f"Mean Squared Error: {MSE: .4f}")


'''
    Calculating Root Mean Squared Error
'''
RMSE = np.sqrt(MSE)
print(f"Root Mean Squared Error: {RMSE: .4f}")
```

```
Test Loss:  0.1145
Mean Squared Error:  1.9683
Root Mean Squared Error:  1.4030
```

In [15]:

```
'''
    Plotting graph with true vs predicted values
'''
fig, ax = plt.subplots()
ax.scatter(y_test, predicted_prices, edgecolors='white', c='orange', label='Actual vs. Predicted')
ax.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], lw=1, color='green', label='Perfect Prediction')
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
ax.set_title('Actual vs. Predicted Prices')
ax.legend()
ax.grid(True, linestyle='--', alpha=0.6)
ax.tick_params(axis='both', which='both', direction='in', length=6, width=2)
plt.show()
```



In [16]:

```
'''
    Calculate R2 score to measure efficiency of our model
'''
def custom_r2_score(y_true, y_pred):
    ss_t = 0
    ss_r = 0
    mean_y = np.mean(y_true)
    for i in range(len(y_true)):
      ss_t += (y_true[i] - mean_y) ** 2
      ss_r += (y_true[i] - y_pred[i]) ** 2
    r2 = 1 - (ss_r/ss_t)
    return r2[0]

print(f"R2 Score: {custom_r2_score(y_test, predicted_prices)}")
```

```
R2 Score: 0.8930918137238979
```