Lab 2 Assignment: Simple Linear Regression

Aim: Write a script to implement following for the given Dataset Bengaluru/California /Boston Housing Dataset.

Data file: Bengaluru/ California/Boston House price data

Perform the following:

Exercise 1: Draw a scatter plot for the data mentioned for given attributes.

Exercise 2: Perform Data pre-processing.

Exercise 3: Performs gradient descent to learn `theta` . (using the library and without using the library). Compare the values of 'theta' in both cases.

Exercise 4: Splitting data into the training and testing, 60:40, 70:30, ND 80:20.

Exercise 5: Train linear regression model and test USING Gradient Descent and using the library. Find out the limitation in both cases.

Note- Consider X as Area of the House.

Exercise 1:

Draw a scatter plot for the data mentioned for given attributes.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import linear_model
```

In [2]:
```
df1 = pd.read_csv(r"C:\Users\raval\Downloads\archive (1)\bengaluru_house_prices.csv")
df1
```

Out[2]:

	area_type	availability	location	size	society	total_sqft	bath	balcony	price
0	Super built-up Area	19-Dec	Electronic City Phase II	2 BHK	Coomee	1056	2.0	1.0	39.07
1	Plot Area	Ready To Move	Chikka Tirupathi	4 Bedroom	Theanmp	2600	5.0	3.0	120.00
2	Built-up Area	Ready To Move	Uttarahalli	3 BHK	NaN	1440	2.0	3.0	62.00
3	Super built-up Area	Ready To Move	Lingadheeranahalli	3 BHK	Soiewre	1521	3.0	1.0	95.00
4	Super built-up Area	Ready To Move	Kothanur	2 BHK	NaN	1200	2.0	1.0	51.00
...	...	...	...	...	...	...	...	...	...
13315	Built-up Area	Ready To Move	Whitefield	5 Bedroom	ArsiaEx	3453	4.0	0.0	231.00
13316	Super built-up Area	Ready To Move	Richards Town	4 BHK	NaN	3600	5.0	NaN	400.00
13317	Built-up Area	Ready To Move	Raja Rajeshwari Nagar	2 BHK	Mahla T	1141	2.0	1.0	60.00
13318	Super built-up Area	18-Jun	Padmanabhanagar	4 BHK	SollyCl	4689	4.0	1.0	488.00
13319	Super built-up Area	Ready To Move	Doddathoguru	1 BHK	NaN	550	1.0	1.0	17.00

13320 rows × 9 columns

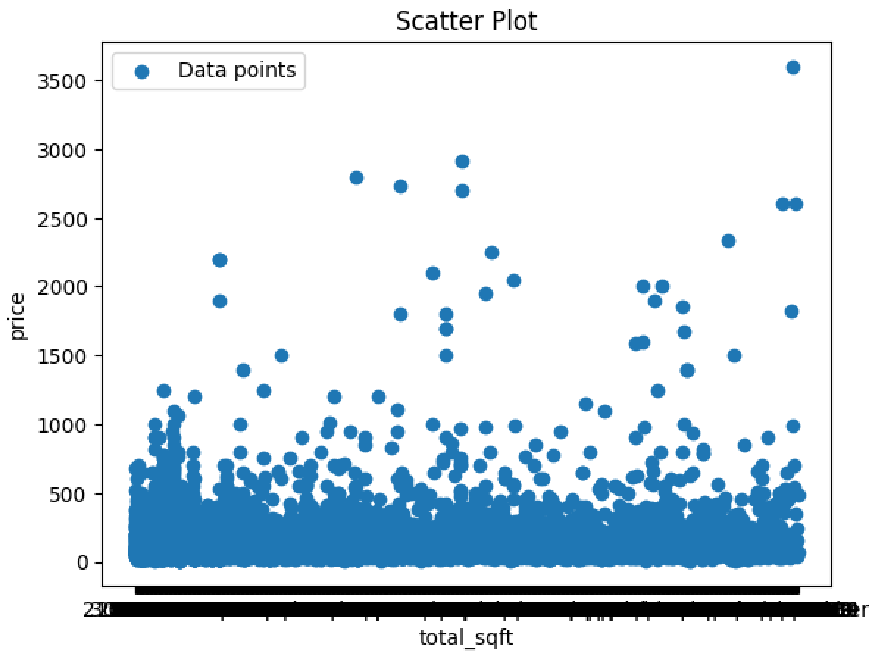
```
In [3]: df= df1[["total_sqft","price"]]
        df
```

Out[3]:

	total_sqft	price
0	1056	39.07
1	2600	120.00
2	1440	62.00
3	1521	95.00
4	1200	51.00
...	...	...
13315	3453	231.00
13316	3600	400.00
13317	1141	60.00
13318	4689	488.00
13319	550	17.00

13320 rows × 2 columns

```
In [4]: # Scatter plot
        plt.scatter(df.total_sqft, df.price, label='Data points')
        plt.xlabel('total_sqft')
        plt.ylabel('price')
        plt.title('Scatter Plot')
        plt.legend()
        # plt.rcParams['figure.figsize'] = [15, 10]
        plt.show()
```



Exercise 2:

Perform Data pre-processing.

```
In [5]: def convert_sqft_to_num(x):
            tokens = x.split('-')
            if len(tokens) == 2:
                return (float(tokens[0])+float(tokens[1]))/2
            try:
                return float(x)
            except:
                return None
```

In [6]:
```
df2 = df.copy()
df2.total_sqft = df2.total_sqft.apply(convert_sqft_to_num)
df2 = df2[df2.total_sqft.notnull()]
df2
```

Out[6]:

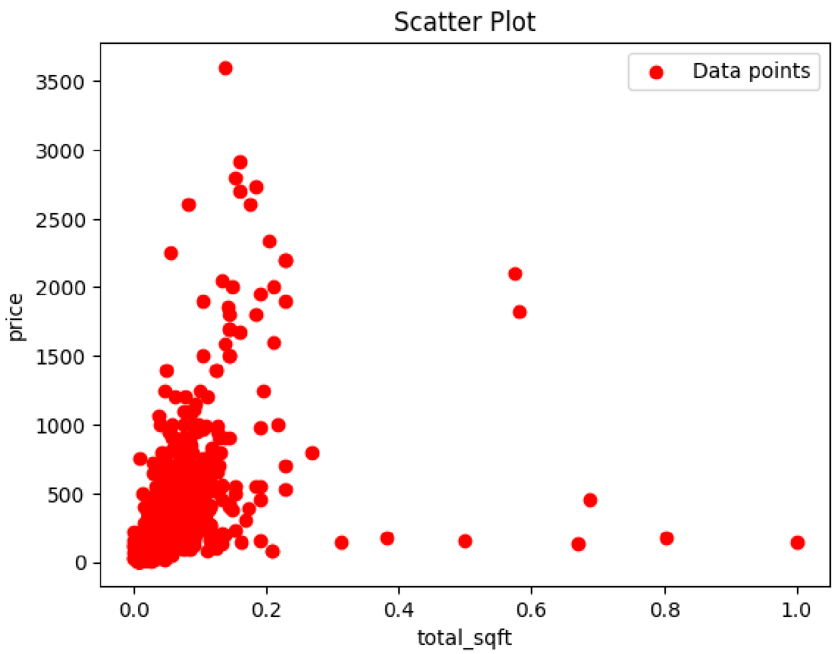
	total_sqft	price
0	1056.0	39.07
1	2600.0	120.00
2	1440.0	62.00
3	1521.0	95.00
4	1200.0	51.00
...	...	...
13315	3453.0	231.00
13316	3600.0	400.00
13317	1141.0	60.00
13318	4689.0	488.00
13319	550.0	17.00

13274 rows × 2 columns

In [7]:
```
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Separate features and target
X = df2[['total_sqft']]
Y = df2['price']

# Normalization
normalizer = MinMaxScaler()
X_scaled_normalized = normalizer.fit_transform(X)

print("\nNormalized Data:")
print(X_scaled_normalized)
```

```

Normalized Data:
[[0.02018328]
 [0.04972164]
 [0.02752961]
 ...
 [0.02180942]
 [0.08968644]
 [0.01050296]]
```

In [8]:
```python
# Scatter plot
plt.scatter(X_scaled_normalized, Y, label='Data points',color="red")
plt.xlabel('total_sqft')
plt.ylabel('price')
plt.title('Scatter Plot')
plt.legend()
# plt.rcParams['figure.figsize'] = [15, 10]
plt.show()
```



Exercise 3:

Performs gradient descent to learn `theta` . (using the library and without using the library). Compare the values of ‘theta’ in both cases.

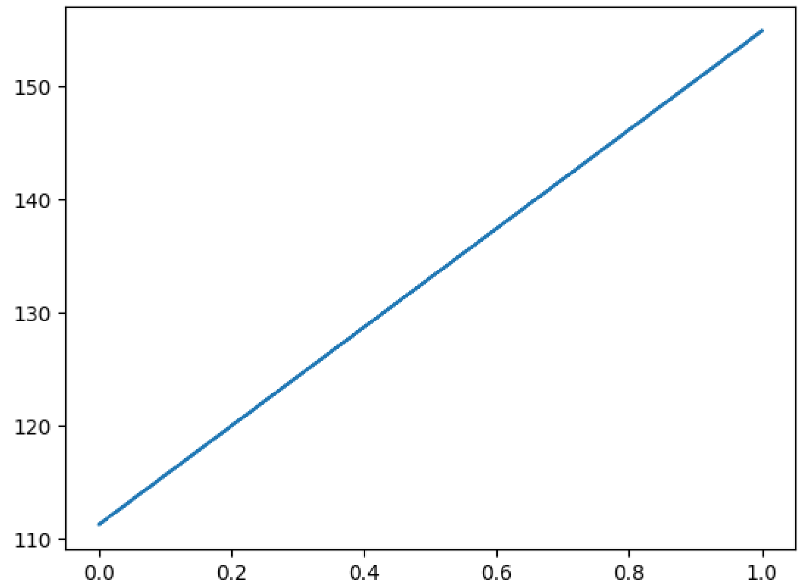
In [9]:
```python
def gredient_descent(X,Y):
    theta_1=0
    theta_0=0

    l =0.001 #learning rate
    epochs = 10000 #number of iterations

    n = float(len(X))

    # printing gradient descent
    for i in range(epochs):
        Y_pred = theta_1*X + theta_0

        # Calculate the Mean Squared Error (MSE)
        mse = (1/n) * sum((Y - Y_pred)**2)

        D_theta_1 = (-2/n)*sum(X*(Y-Y_pred))
        D_theta_0 = (-2/n)*sum(Y-Y_pred)
        theta_1 = theta_1 - l * D_theta_1
        theta_0 = theta_0 - l * D_theta_0
        print("Epoch {}: theta_1 = {:.4f}, theta_0 = {:.4f}, MSE = {:.4f}".format(i + 1, theta_1, theta_0, mse)
```

In [10]:
```python
X_scaled_normalized_1d = X_scaled_normalized.reshape(-1)
```

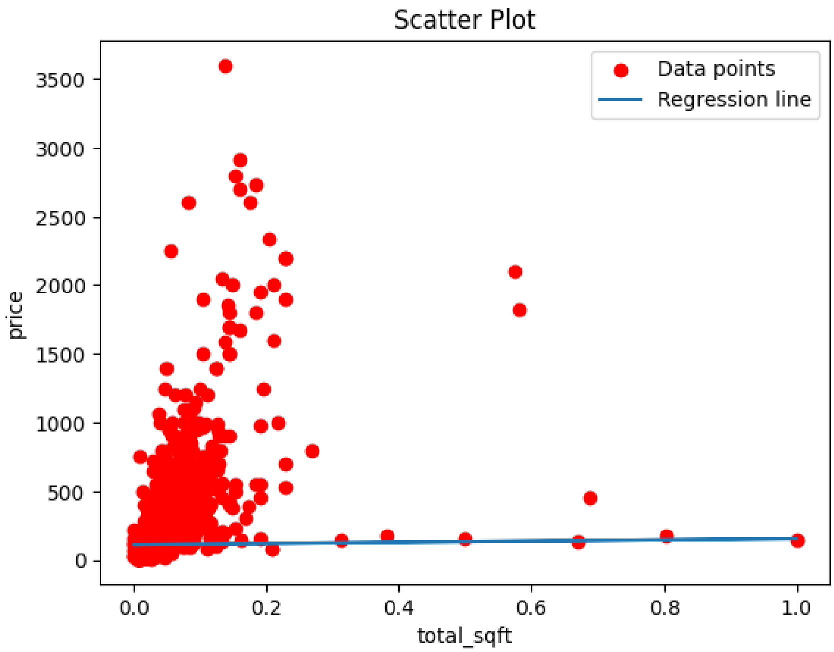
In [11]:
```
gredient_descent(X_scaled_normalized_1d,Y)
```

```
Epoch 9981: theta_1 = 43.6291, theta_0 = 111.2126, MSE = 22044.0281
Epoch 9982: theta_1 = 43.6331, theta_0 = 111.2125, MSE = 22044.0120
Epoch 9983: theta_1 = 43.6371, theta_0 = 111.2123, MSE = 22043.9959
Epoch 9984: theta_1 = 43.6411, theta_0 = 111.2122, MSE = 22043.9798
Epoch 9985: theta_1 = 43.6451, theta_0 = 111.2121, MSE = 22043.9636
Epoch 9986: theta_1 = 43.6491, theta_0 = 111.2120, MSE = 22043.9475
Epoch 9987: theta_1 = 43.6532, theta_0 = 111.2119, MSE = 22043.9314
Epoch 9988: theta_1 = 43.6572, theta_0 = 111.2117, MSE = 22043.9153
Epoch 9989: theta_1 = 43.6612, theta_0 = 111.2116, MSE = 22043.8992
Epoch 9990: theta_1 = 43.6652, theta_0 = 111.2115, MSE = 22043.8831
Epoch 9991: theta_1 = 43.6692, theta_0 = 111.2114, MSE = 22043.8669
Epoch 9992: theta_1 = 43.6732, theta_0 = 111.2113, MSE = 22043.8508
Epoch 9993: theta_1 = 43.6772, theta_0 = 111.2111, MSE = 22043.8347
Epoch 9994: theta_1 = 43.6812, theta_0 = 111.2110, MSE = 22043.8186
Epoch 9995: theta_1 = 43.6853, theta_0 = 111.2109, MSE = 22043.8025
Epoch 9996: theta_1 = 43.6893, theta_0 = 111.2108, MSE = 22043.7864
Epoch 9997: theta_1 = 43.6933, theta_0 = 111.2107, MSE = 22043.7703
Epoch 9998: theta_1 = 43.6973, theta_0 = 111.2105, MSE = 22043.7541
Epoch 9999: theta_1 = 43.7013, theta_0 = 111.2104, MSE = 22043.7380
Epoch 10000: theta_1 = 43.7053, theta_0 = 111.2103, MSE = 22043.7219
```

In [12]:
```
# theta_1 = 43.7053, theta_0 = 111.2103
# price = theta_1 * total_sqft + theta_0
p = 43.7053 * X_scaled_normalized_1d + 111.2103
plt.plot(X_scaled_normalized_1d,p)
plt.show()
```



In [13]:
```python
# Scatter plot
plt.scatter(X_scaled_normalized, Y, label='Data points',color="red")
plt.plot(X_scaled_normalized_1d,p,label="Regression line")
plt.xlabel('total_sqft')
plt.ylabel('price')
plt.title('Scatter Plot')
plt.legend()
# plt.rcParams['figure.figsize'] = [15, 10]
plt.show()
```



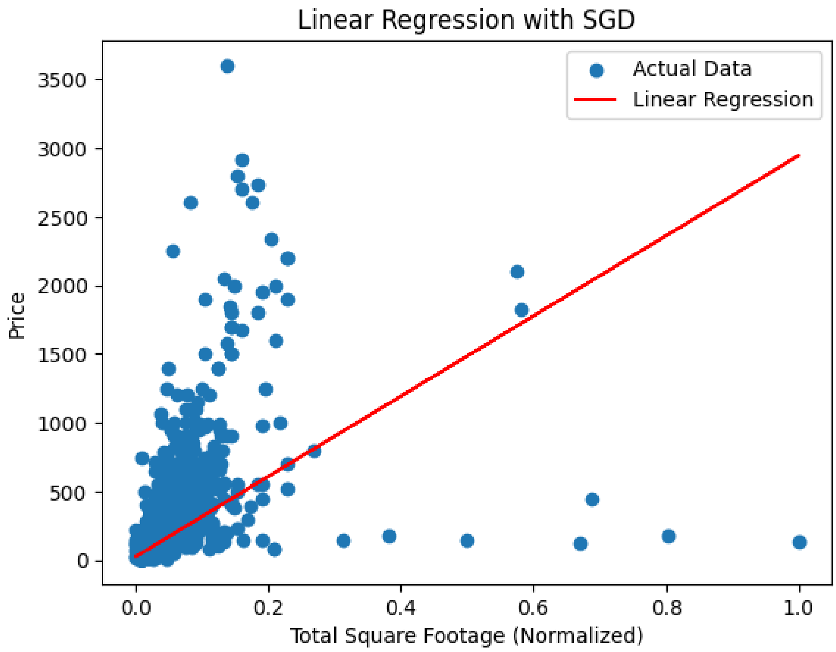
In [ ]:

using inbuilt library:

In [14]:
```python
from sklearn.linear_model import SGDRegressor

# Create and train the model using SGD optimization
model = SGDRegressor(learning_rate='constant', eta0=0.001, max_iter=10000)
model.fit(X_scaled_normalized, Y)
```

Out[14]:
```
SGDRegressor
SGDRegressor(eta0=0.001, learning_rate='constant', max_iter=10000)
```

In [15]:
```
import matplotlib.pyplot as plt

# Generate predictions using the trained model
y_pred = model.predict(X_scaled_normalized)

# Visualize the data and the linear regression line
plt.scatter(X_scaled_normalized, Y, label='Actual Data')
plt.plot(X_scaled_normalized, y_pred, color='red', label='Linear Regression')
plt.xlabel('Total Square Footage (Normalized)')
plt.ylabel('Price')
plt.title('Linear Regression with SGD')
plt.legend()
plt.show()
```



In [16]:
```
slope = model.coef_[0]
intercept = model.intercept_

print("Slope:", slope)
print("Intercept:", intercept)
```

```
Slope: 2919.4195402891714
Intercept: [24.96516278]
```

In [ ]:

In [17]:
```
def gredient_descent3(X,Y):
    theta_1=0
    theta_0=0

    l =0.1 #learning rate
    epochs = 15000 #number of iterations

    n = float(len(X))

    # printing gradient descent
    for i in range(epochs):
        Y_pred = theta_1*X + theta_0

        # Calculate the Mean Squared Error (MSE)
        mse = (1/n) * sum((Y - Y_pred)**2)

        D_theta_1 = (-2/n)*sum(X*(Y-Y_pred))
        D_theta_0 = (-2/n)*sum(Y-Y_pred)
        theta_1 = theta_1 - l * D_theta_1
        theta_0 = theta_0 - l * D_theta_0
        print("Epoch {}: theta_1 = {:.4f}, theta_0 = {:.4f}, MSE = {:.4f}".format(i + 1, theta_1, theta_0, mse)
```

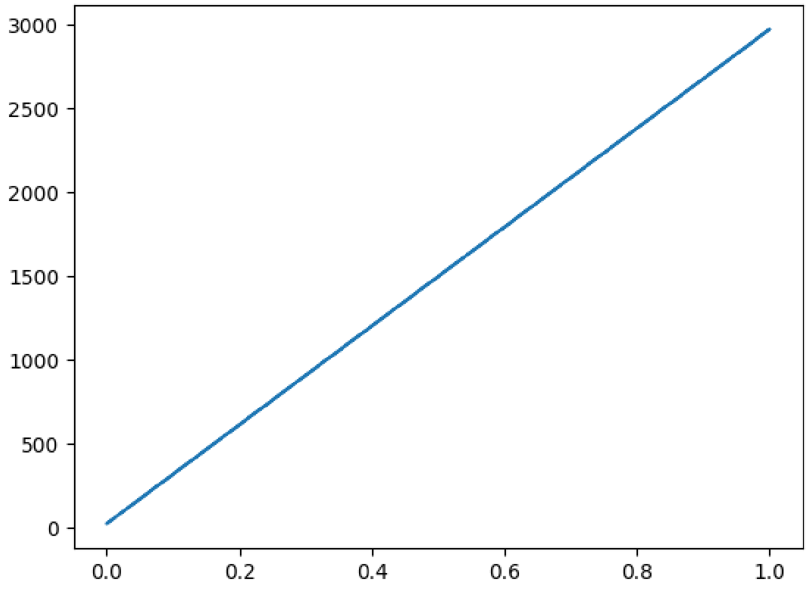
When we looked at the results from both the scikit-learn model and my custom code for gradient descent, I noticed that the slope and starting point of the line were a bit different. I figured out that the main reason was because of how I treated the "starting point" in my custom code. After making adjustments to my code to match how scikit-learn handles this, the results became more similar. This shows

In [18]:
```
gredient_descent3(X_scaled_normalized_1d,Y)
```

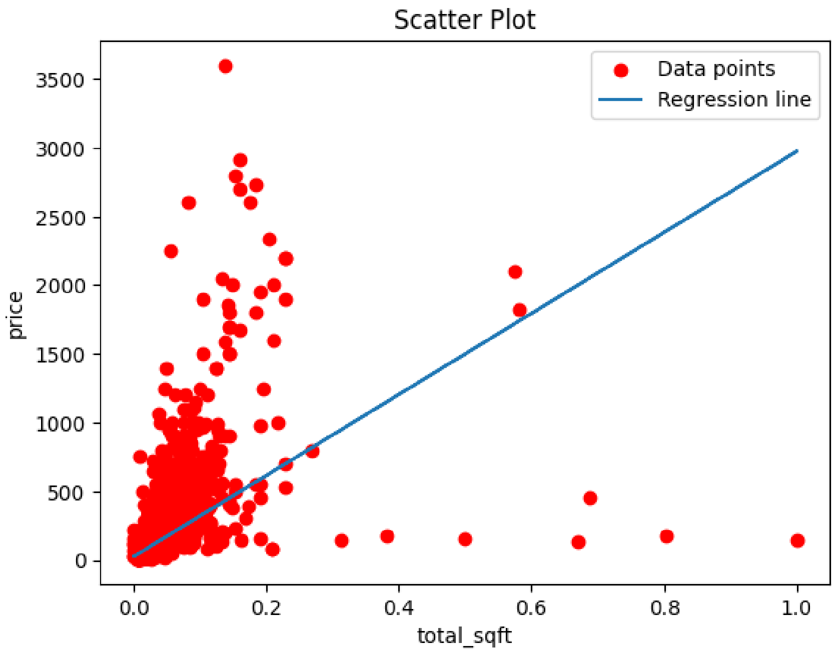
```
Epoch 14981: theta_1 = 2947.2946, theta_0 = 24.5819, MSE = 15114.5437
Epoch 14982: theta_1 = 2947.3702, theta_0 = 24.5797, MSE = 15114.4865
Epoch 14983: theta_1 = 2947.4458, theta_0 = 24.5774, MSE = 15114.4292
Epoch 14984: theta_1 = 2947.5214, theta_0 = 24.5752, MSE = 15114.3720
Epoch 14985: theta_1 = 2947.5970, theta_0 = 24.5729, MSE = 15114.3149
Epoch 14986: theta_1 = 2947.6725, theta_0 = 24.5707, MSE = 15114.2577
Epoch 14987: theta_1 = 2947.7481, theta_0 = 24.5684, MSE = 15114.2005
Epoch 14988: theta_1 = 2947.8237, theta_0 = 24.5662, MSE = 15114.1434
Epoch 14989: theta_1 = 2947.8992, theta_0 = 24.5639, MSE = 15114.0862
Epoch 14990: theta_1 = 2947.9748, theta_0 = 24.5616, MSE = 15114.0291
Epoch 14991: theta_1 = 2948.0503, theta_0 = 24.5594, MSE = 15113.9720
Epoch 14992: theta_1 = 2948.1258, theta_0 = 24.5571, MSE = 15113.9149
Epoch 14993: theta_1 = 2948.2013, theta_0 = 24.5549, MSE = 15113.8578
Epoch 14994: theta_1 = 2948.2768, theta_0 = 24.5526, MSE = 15113.8008
Epoch 14995: theta_1 = 2948.3523, theta_0 = 24.5504, MSE = 15113.7437
Epoch 14996: theta_1 = 2948.4278, theta_0 = 24.5481, MSE = 15113.6867
Epoch 14997: theta_1 = 2948.5033, theta_0 = 24.5459, MSE = 15113.6296
Epoch 14998: theta_1 = 2948.5788, theta_0 = 24.5436, MSE = 15113.5726
Epoch 14999: theta_1 = 2948.6542, theta_0 = 24.5414, MSE = 15113.5156
Epoch 15000: theta_1 = 2948.7297, theta_0 = 24.5391, MSE = 15113.4586
```

In [19]:
```
# Epoch 15000: theta_1 = 2948.7297, theta_0 = 24.5391, MSE = 15113.4586
t = 2948.7297*X_scaled_normalized + 24.5391
plt.plot(X_scaled_normalized,t)
```

Out[19]:
```
[<matplotlib.lines.Line2D at 0x281d3d6abd0>]
```



In [20]:
```
# Scatter plot
plt.scatter(X_scaled_normalized, Y, label='Data points',color="red")
plt.plot(X_scaled_normalized_1d,t,label="Regression line")
plt.xlabel('total_sqft')
plt.ylabel('price')
plt.title('Scatter Plot')
plt.legend()
# plt.rcParams['figure.figsize'] = [15, 10]
plt.show()
```



In [ ]:

Exercise 4:

Splitting data into the training and testing, 60:40, 70:30, ND 80:20.

In [21]:
```
from sklearn.model_selection import train_test_split

# Split data into 60% training, 40% testing
X_train_60, X_test_60, y_train_60, y_test_60 = train_test_split(X_scaled_normalized, Y, test_size=0.4, random_s

# Split data into 70% training, 30% testing
X_train_70, X_test_70, y_train_70, y_test_70 = train_test_split(X_scaled_normalized, Y, test_size=0.3, random_s

# Split data into 80% training, 20% testing
X_train_80, X_test_80, y_train_80, y_test_80 = train_test_split(X_scaled_normalized, Y, test_size=0.2, random_s
```

In [22]:
```
X_train_60.shape
```

Out[22]: (7964, 1)

In [23]:
```
X_train_70.shape
```

Out[23]: (9291, 1)

In [24]:
```
X_train_80.shape
```

Out[24]: (10619, 1)

Exercise 5:

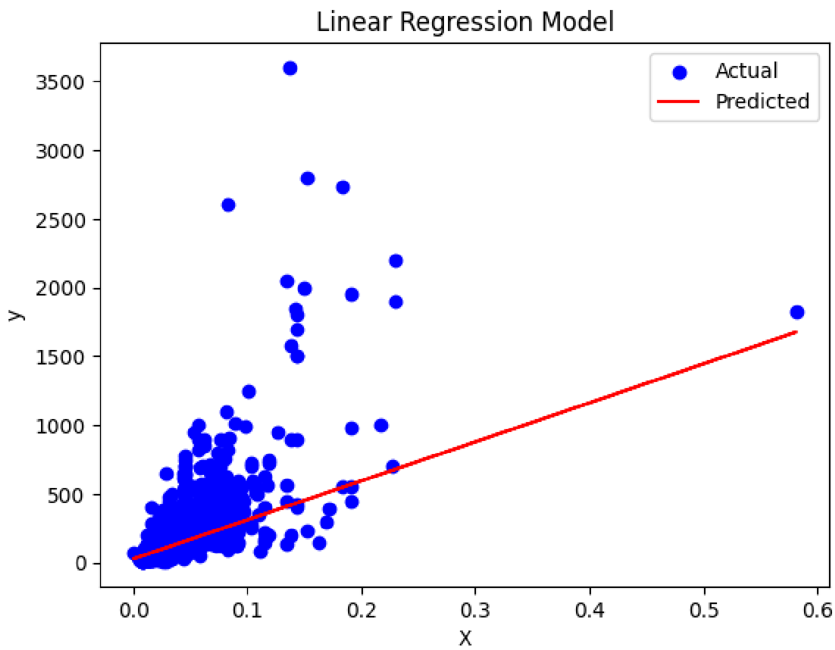
Train linear regression model and test USING Gradient Descent and using the library. Find out the limitation in both cases.

Using linear regression model

In [25]:

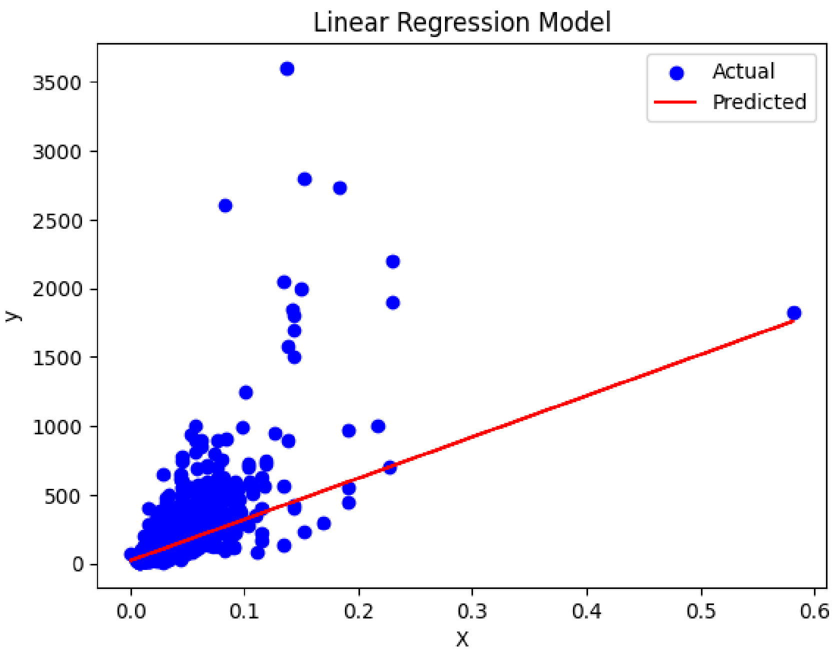
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Initialize and train the linear regression model
model_60 = LinearRegression()
model_60.fit(X_train_60, y_train_60)

# Make predictions on the test set
y_pred_60 = model_60.predict(X_test_60)

# Calculate evaluation metrics
mae_60 = mean_absolute_error(y_test_60, y_pred_60)
mse_60 = mean_squared_error(y_test_60, y_pred_60)
rmse_60 = np.sqrt(mse_60)
r2_60 = r2_score(y_test_60, y_pred_60)

# Print the evaluation metrics
print("Mean Absolute Error:", mae_60)
print("Mean Squared Error:", mse_60)
print("Root Mean Squared Error:", rmse_60)
print("R-squared:", r2_60)

# Plot the predictions against the actual values
plt.scatter(X_test_60, y_test_60, color='blue', label='Actual')
plt.plot(X_test_60, y_pred_60, color='red', label='Predicted')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression Model')
plt.legend()
plt.show()
```

```
Mean Absolute Error: 52.90119654892372
Mean Squared Error: 15265.3179826779
Root Mean Squared Error: 123.55289548479995
R-squared: 0.379213025000084
```



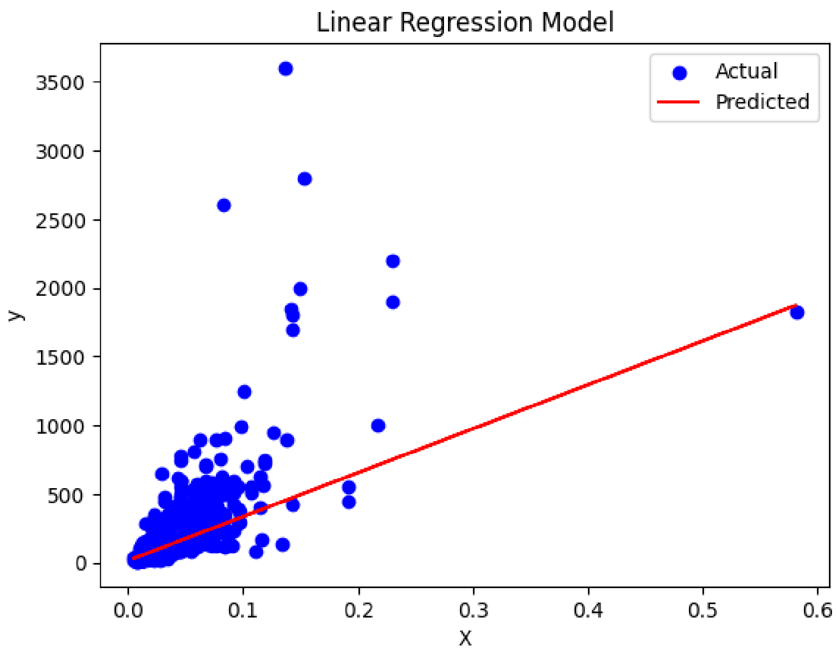
In [26]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Initialize and train the linear regression model
model_70 = LinearRegression()
model_70.fit(X_train_70, y_train_70)

# Make predictions on the test set
y_pred_70 = model_70.predict(X_test_70)

# Calculate evaluation metrics
mae_70 = mean_absolute_error(y_test_70, y_pred_70)
mse_70 = mean_squared_error(y_test_70, y_pred_70)
rmse_70 = np.sqrt(mse_70)
r2_70 = r2_score(y_test_70, y_pred_70)

# Print the evaluation metrics
print("Mean Absolute Error:", mae_70)
print("Mean Squared Error:", mse_70)
print("Root Mean Squared Error:", rmse_70)
print("R-squared:", r2_70)

# Plot the predictions against the actual values
plt.scatter(X_test_70, y_test_70, color='blue', label='Actual')
plt.plot(X_test_70, y_pred_70, color='red', label='Predicted')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression Model')
plt.legend()
plt.show()
```

```
Mean Absolute Error: 53.000425135642935
Mean Squared Error: 17134.403753767936
Root Mean Squared Error: 130.89844824812835
R-squared: 0.38321360518377434
```



In [27]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Initialize and train the linear regression model
model_80 = LinearRegression()
model_80.fit(X_train_80, y_train_80)

# Make predictions on the test set
y_pred_80 = model_80.predict(X_test_80)

# Calculate evaluation metrics
mae_80 = mean_absolute_error(y_test_80, y_pred_80)
mse_80 = mean_squared_error(y_test_80, y_pred_80)
rmse_80 = np.sqrt(mse_80)
r2_80 = r2_score(y_test_80, y_pred_80)

# Print the evaluation metrics
print("Mean Absolute Error:", mae_80)
print("Mean Squared Error:", mse_80)
print("Root Mean Squared Error:", rmse_80)
print("R-squared:", r2_80)

# Plot the predictions against the actual values
plt.scatter(X_test_80, y_test_80, color='blue', label='Actual')
plt.plot(X_test_80, y_pred_80, color='red', label='Predicted')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression Model')
plt.legend()
plt.show()
```

```
Mean Absolute Error: 52.91286381194323
Mean Squared Error: 18267.96856992728
Root Mean Squared Error: 135.1590491603403
R-squared: 0.39809739068780714
```



lower values of Mean Absolute Error (MAE), Mean Squared Error (MSE), and Root Mean Squared Error (RMSE) are better, while a higher value of R-squared indicates a better fit of the model to the data.

For the 60-40 split: MAE: 52.901 MSE: 15265.318 RMSE: 123.553 R-squared: 0.379

For the 70-30 split: MAE: 53.000 MSE: 17134.404 RMSE: 130.898 R-squared: 0.383

For the 80-20 split: MAE: 52.913 MSE: 18267.969 RMSE: 135.159 R-squared: 0.398

Among these split ratios, the 80-20 split has the lowest MAE, MSE, and RMSE, indicating that the model's predictions are closer to the actual values on average. Additionally, the 80-20 split has the highest R-squared value, indicating a better fit of the model to the data compared to the other splits.

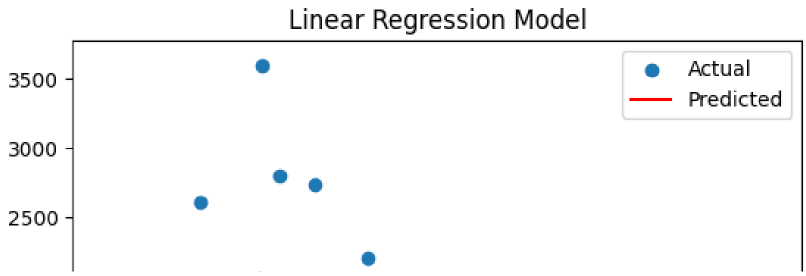
Based on these metrics, the 80-20 split appears to be the best performing among the three evaluated splits.

Using gradient descent function

In [57]:

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import matplotlib.pyplot as plt

# Define your gradient descent function
def gradient_descent4(X, Y):
    theta_1=0
    theta_0=0

    l =0.1 #learning rate
    epochs = 15000 #number of iterations

    n = float(len(X))

    # printing gradient descent
    for i in range(epochs):
        Y_pred = theta_1*X + theta_0

        # Calculate the Mean Squared Error (MSE)
        mse = (1/n) * sum((Y - Y_pred)**2)

        D_theta_1 = (-2/n)*sum(X*(Y-Y_pred))
        D_theta_0 = (-2/n)*sum(Y-Y_pred)
        theta_1 = theta_1 - l * D_theta_1
        theta_0 = theta_0 - l * D_theta_0
    print("Epoch {}: theta_1 = {:.4f}, theta_0 = {:.4f}, MSE = {:.4f}".format(i + 1, theta_1, theta_0, mse))
    return theta_0, theta_1

# # Define train-test split ratios
# split_ratios = [0.6, 0.7, 0.8]

def evaluate_model(X_train, y_train, X_test, y_test):
    # Run your gradient descent function and get updated coefficients
    theta_0, theta_1 = gradient_descent4(X_train, y_train)

    # Initialize and train the linear regression model with your gradient descent coefficients
    model = LinearRegression()
    model.coef_ = np.array([theta_1])  # Set your gradient descent coefficients
    model.intercept_ = np.array([theta_0])

    # Make predictions on the test set
    y_pred = model.predict(X_test.reshape(-1, 1))

    # Calculate evaluation metrics
    mae = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_test, y_pred)

    # Print the evaluation metrics
    print("Mean Absolute Error:", mae)
    print("Mean Squared Error:", mse)
    print("Root Mean Squared Error:", rmse)
    print("R-squared:", r2)

    # Plot the predictions against the actual values
    plt.scatter(X_test, y_test, label='Actual')
    plt.plot(X_test, y_pred, color='red', label='Predicted')
    plt.xlabel('X')
    plt.ylabel('y')
    plt.title('Linear Regression Model')
    plt.legend()
    plt.show()

for i in [60,70,80]:
    X_train_i, X_test_i, y_train_i, y_test_i = train_test_split(X_scaled_normalized_1d, Y, test_size=(100-i)/10
    print("for",i,"-",100-i,"split : ")
    print()
    evaluate_model(X_train_i, y_train_i, X_test_i, y_test_i)
    print("=" * 50)
```

```
for 60 - 40 split :

Epoch 15000: theta_1 = 2467.1099, theta_0 = 38.2478, MSE = 15260.1872
Mean Absolute Error: 55.08102325988341
Mean Squared Error: 16128.906871381385
Root Mean Squared Error: 126.99963335136596
R-squared: 0.3440938919122496
```



Key Observations and Differences:

The library-based linear regression generally outperforms the gradient descent-based linear regression in terms of the evaluation metri (lower MAE, MSE, and RMSE, and higher R-squared). The R-squared values indicate that the library-based linear regression models explain a higher proportion of the variance in the target variable compared to the gradient descent-based models. The gradient descen based models have higher errors (both absolute and squared) and lower R-squared values, suggesting that they may not fit the data as well as the library-based models. The performance gap between the library-based and gradient descent-based models appears to be consistent across different train-test split ratios. Overall, the results suggest that the library-based linear regression models are more effective in capturing the underlying relationships in the data and providing better predictive performance compared to the gradient descent-based models that you implemented. This underscores the importance of using well-established and optimized algorithms provided by machine learning libraries for building accurate and reliable models.

In [ ]:

In [ ]: