House Price Prediction Modeling Using Machine Learning

Abstract

This project has two parts. The first part evaluates the performance of two machine learning algorithms (Linear Regression and Gradient Boosting) on the Boston Housing dataset. The objectives includes predicting housing prices based on various features and comparing the performance of the two algorithms using metric such as Mean Squared Error (MSE). The second part evaluates the performance of (Linear Regression) that i made from scratch and compares it's results with the one from Scikit-learn Python library.

Introduction

Description:

Predict house prices based on features like Per capita crime rate by town, average number of rooms per dwelling, etc...

Used algorithms:

1. Linear Regression (from Scikit-learn library).
2. Gradient Boosting (from Scikit-learn library).
3. Linear Regression (from scratch).

Dataset Description

The Boston Housing Dataset contains information about housing in the Boston area, collected in 1978. , it includes numerical data, and some entries have missing values (denoted by NA),and it is commonly used to predict median house prices (in thousands of dollars) based on various features describing the housing environment.

key features:

1. **CRIM**: Per capita crime rate by town.
2. **ZN**: Proportion of residential land zoned for lots over 25,000 sq. ft.
3. **INDUS**: Proportion of non-retail business acres per town.
4. **CHAS**: Charles River dummy variable (1 if tract bounds river; 0 otherwise).
5. **NOX**: Nitric oxides concentration (parts per 10 million).
6. **RM**: Average number of rooms per dwelling.
7. **AGE**: Proportion of owner-occupied units built prior to 1940.
8. **DIS**: Weighted distances to five Boston employment centers.
9. **RAD**: Index of accessibility to radial highways.
10. **TAX**: Full-value property tax rate per $10,000.
11. **PTRATIO**: Pupil-teacher ratio by town.
12. **B**: 1000(Bk - 0.63) ^2 where Bk is the proportion of Black residents by town.
13. **LSTAT**: Percentage of lower status of the population.
14. **MEDV**: Median value of owner-occupied homes in $1000s.

Sources: The Boston Housing Dataset

Implementation of (Linear Regression & Gradient Boosting)

- From Scikit-learn Python library -

-load the dataset, fill the missing values, select (X & Y) and split the data into (training and testing) sets-

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error, mean_absolute_error

from sklearn.linear_model import LinearRegression

from sklearn.ensemble import GradientBoostingRegressor

#################################################################

Load the dataset #

data = pd.read_csv('HousingData.csv')

Fill NaN values #

data.fillna(data.mean(), inplace=True )

Select featuers(x1,x2,....), and target(Y) #

X = data.iloc[: , :-1]

y = data['MEDV']

Data splitting into (training & tsting data) sets #

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42 )
```

-implementation of LR-

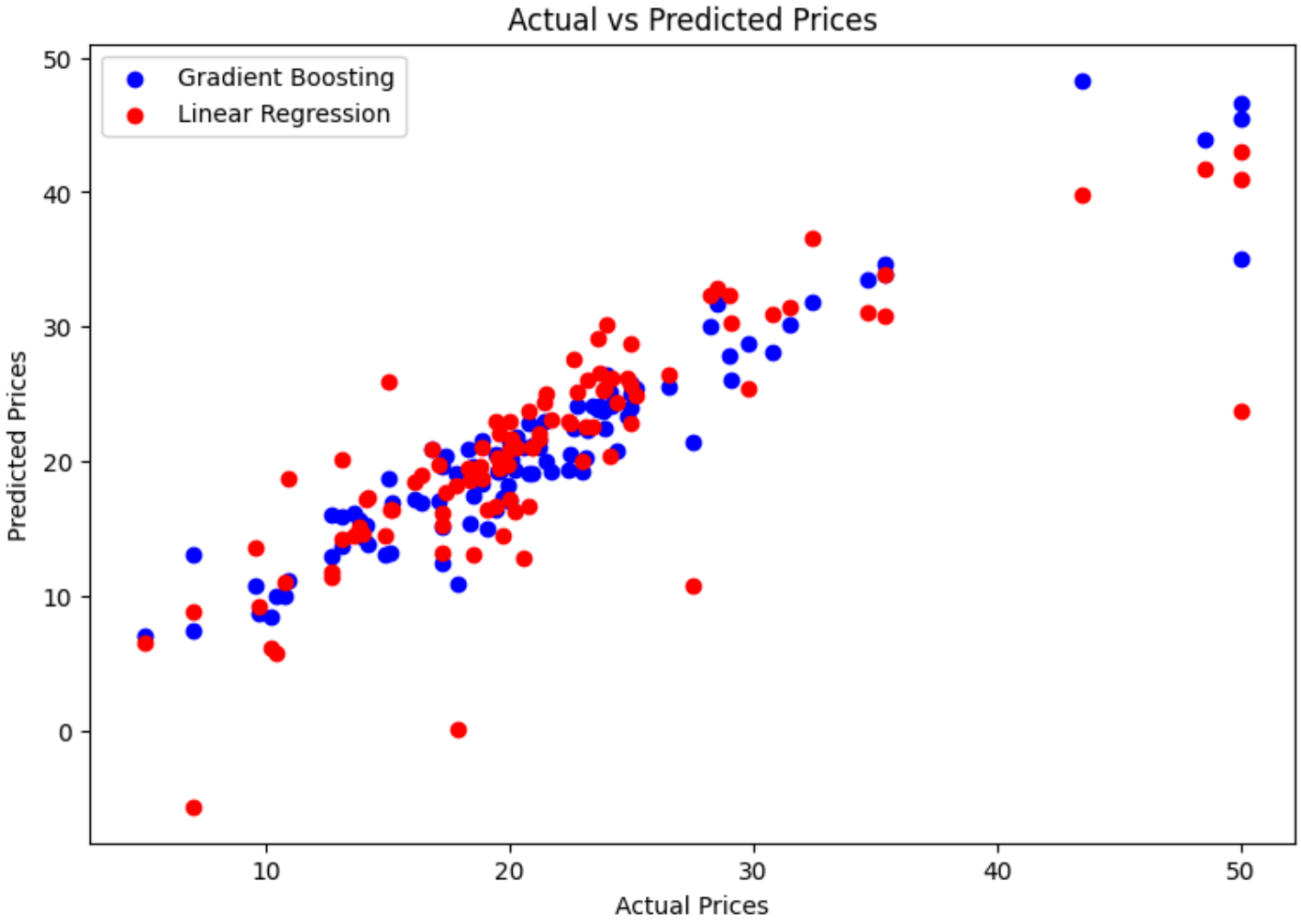
```
Train the LR model #

lr_sklearn = LinearRegression()

lr_sklearn.fit(X_train, y_train )

Make predictions #

y_pred_lr_sklearn = lr_sklearn.predict(X_test )

Calculate MSE fot the model #

mse_lr_sklearn = mean_squared_error(y_test, y_pred_lr_sklearn )
```

-implementation of GB-

```
Train the GB model #

gbr_sklearn = GradientBoostingRegressor(

n_estimators=100, random_state

 )

gbr_sklearn.fit(X_train, y_train )

Make predictions #

y_pred_gbr_sklearn = gbr_sklearn.predict(X_test )

Calculate MSE fot the model #

mse_gbr_sklearn = mean_squared_error(y_test, y_pred_gbr_sklearn )
```

Results & Comparison

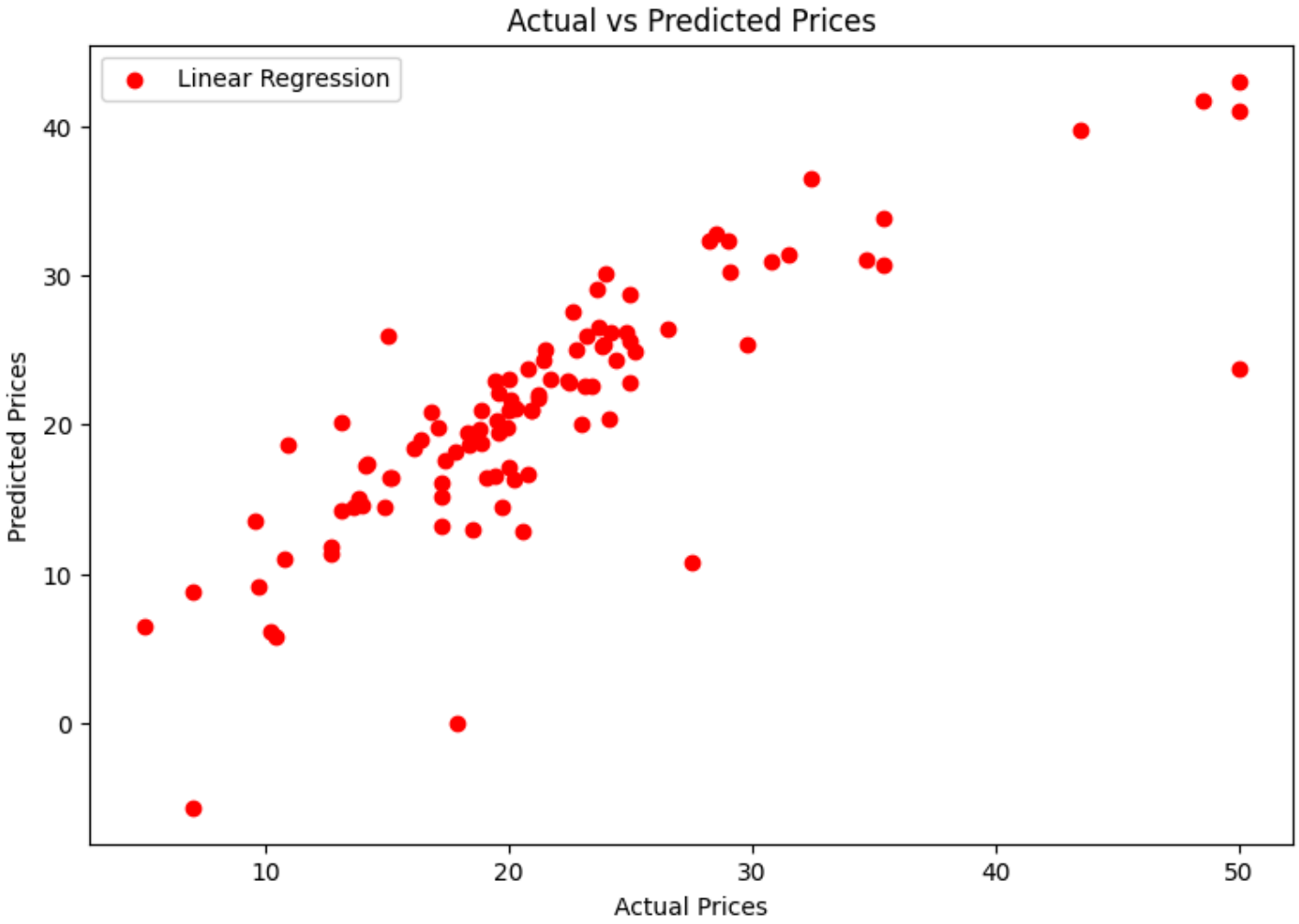
Algorithm	Mean Squared Error (MSE)
Linear Regression	25.01
Gradient Boosting	7.36



Linear Regression:

- A simple model that assumes a linear relationship between features (X) and the target (Y).
- Highly interpretable, because of its a simple model .
- Highly sensitive to outliers and noise.
- Lower accuracy, because it fails to capture nonlinear relationships in the data.

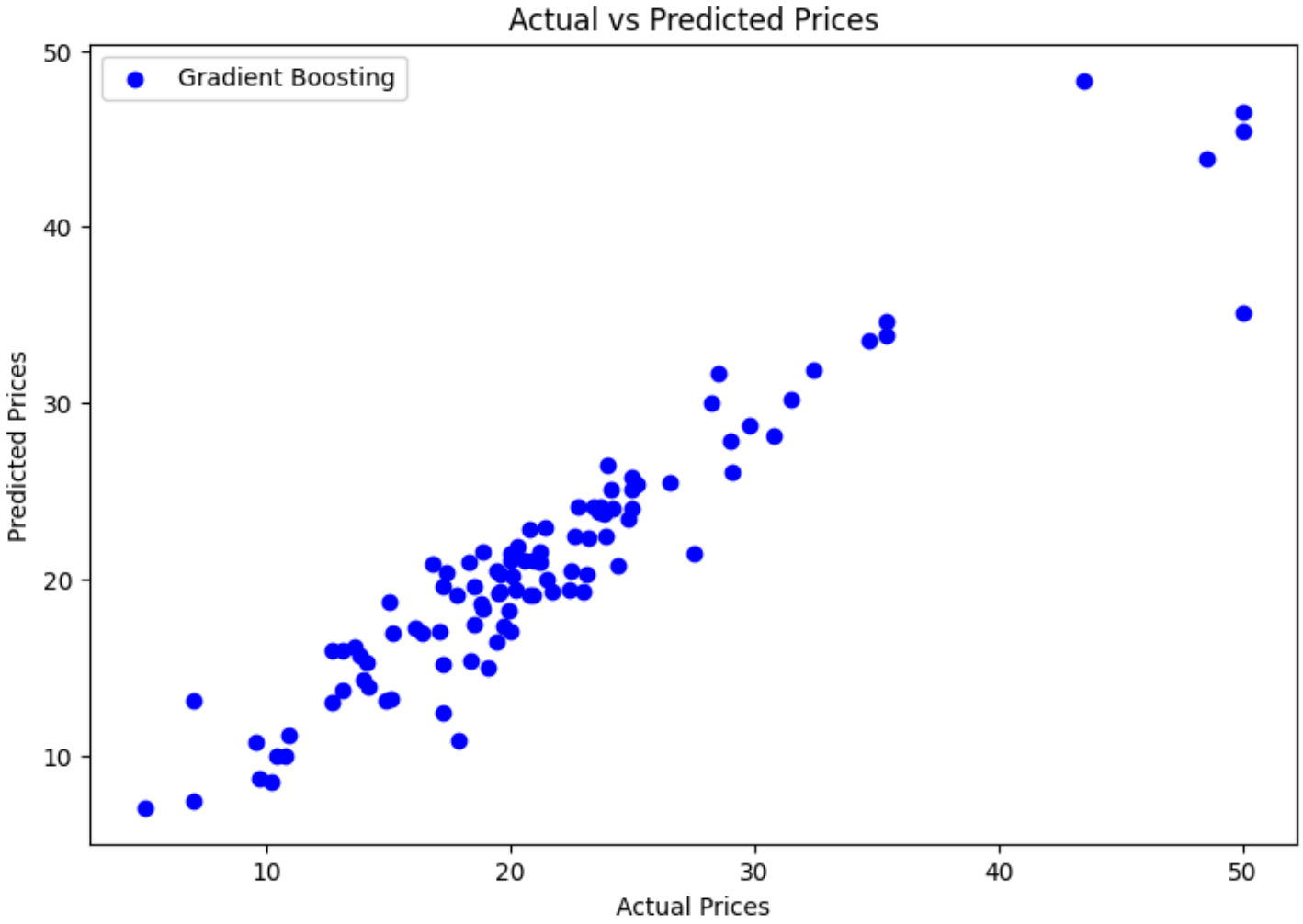
So, Linear Regression Performs well when the relationship between features and target is linear and data is noise-free.



Gradient Boosting:

- A more complex model based on decision trees.
- Less interpretable, because of its complex nature and use of multiple decision trees.
- Less sensitive to outliers due to its tree-based structure.
- Higher accuracy because it handles complex, nonlinear patterns better.

So, Gradient Boosting Performs well in most cases, where it combines decision trees to minimize errors well.



Implementation of Linear Regression

-From scratch-

-Import needed libraries, and construct a function in LR class that initializes the variables-

```
import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.model_selection import train_test_split
Linear Regression Class #

class LinearRegression:

Initialization the variables #

def __init__(self, learning_rate=1e-5, n_iters=1000 ) :

self.lr = learning_rate

self.n_iters = n_iters

self.weights = None

self.bias = None
```

-Construct a function in LR class that trains our LR model-

```
Training the model #

def fit(self, X, y):

Initialize weights and bias as 0 #

n_samples, n_features = X.shape

self.weights = np.zeros(n_features )

self.bias = 0

Training loop (Calculations to update weights and bias) #

for _ in range(self.n_iters):


Predict using y = X * weights + bias #

y_p = np.dot(X, self.weights) + self.bias


Calculate dw and db #

dw = (1 / n_samples) * np.dot(X.T, (y_p - y)) #dw

db = (1 / n_samples) * np.sum(y_p - y)        #db


Update weights and bias #

self.weights -= self.lr * dw  # w = w - lr * dw

self.bias -= self.lr * db     # b = b - lr * db
```

-Construct a function in LR class that test the model-

```
Testing the model #

def predict(self, x )  :

Y = X * weights + bias #

return np.dot(x, self.weights) + self.bias
```

-Construct functions that calculates Mean Squared Error {MSE}-

```
Mean Squared Error Calculation #

def mse(y_test, predictions):

return np.mean((y_test - predictions) ** 2 )
```

-load the dataset, fill the missing values, select (X & Y) and split the data into (training and testing) sets-

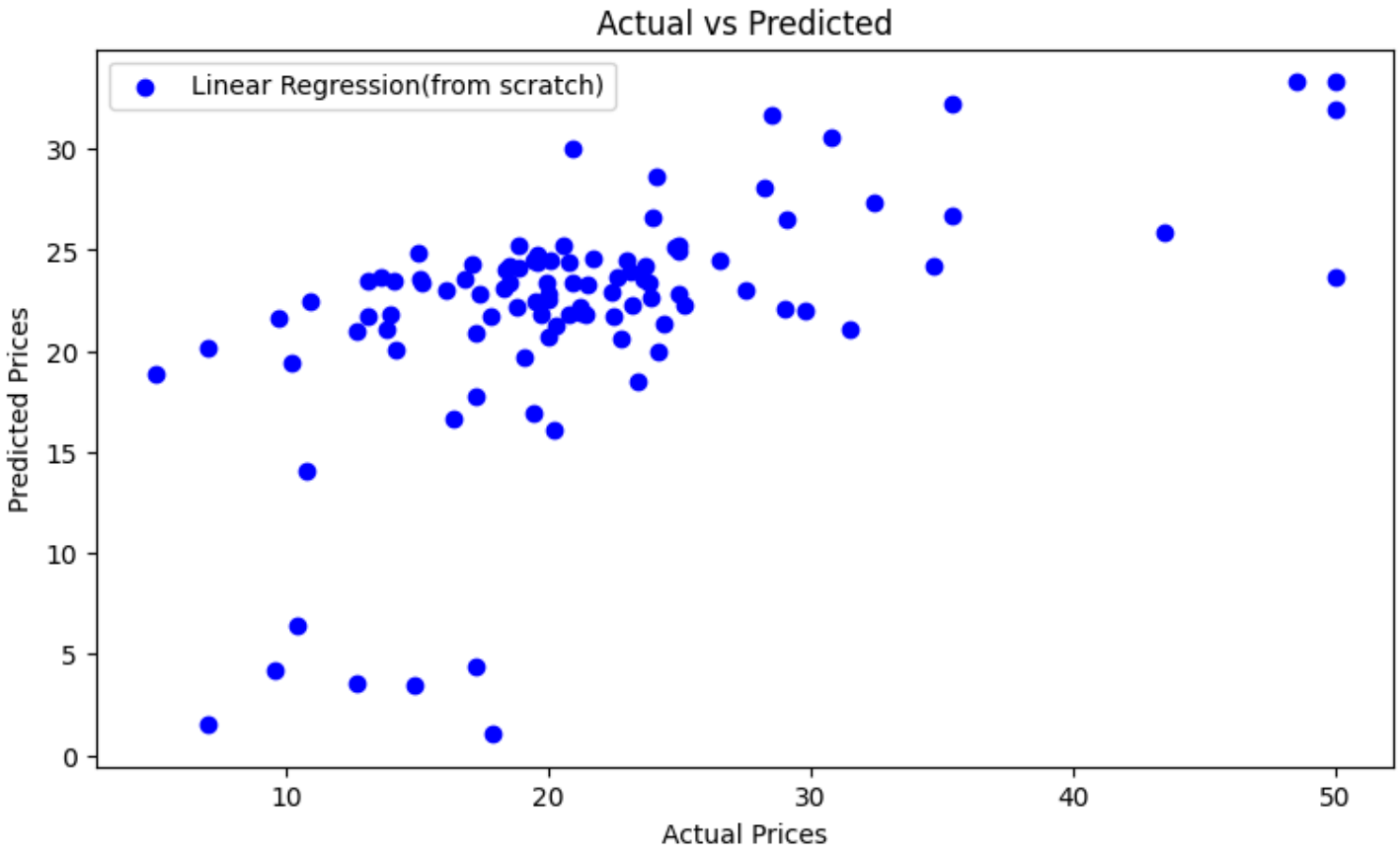
```

Load and preprocess data #

data = pd.read_csv('HousingData.csv')

data.fillna(data.mean(), inplace=True)


Extract features and target variable #

X = data.iloc[: , :-1]

y = data['MEDV'].values


Split data into training and testing sets #

\= X_train, X_test, y_train, y_test

train_test_split(X, y, test_size=0.2, random_state=42)
```

-Train and Test the model on our dataset-

```
Train the model #

reg = LinearRegression(learning_rate=1e-6, n_iters=10000)

reg.fit(X_train, y_trair)


Predict on the test set #

predictions = reg.predict(X_test)


Calculate MSE #

mse_value = mse(y_test, predictions)
```

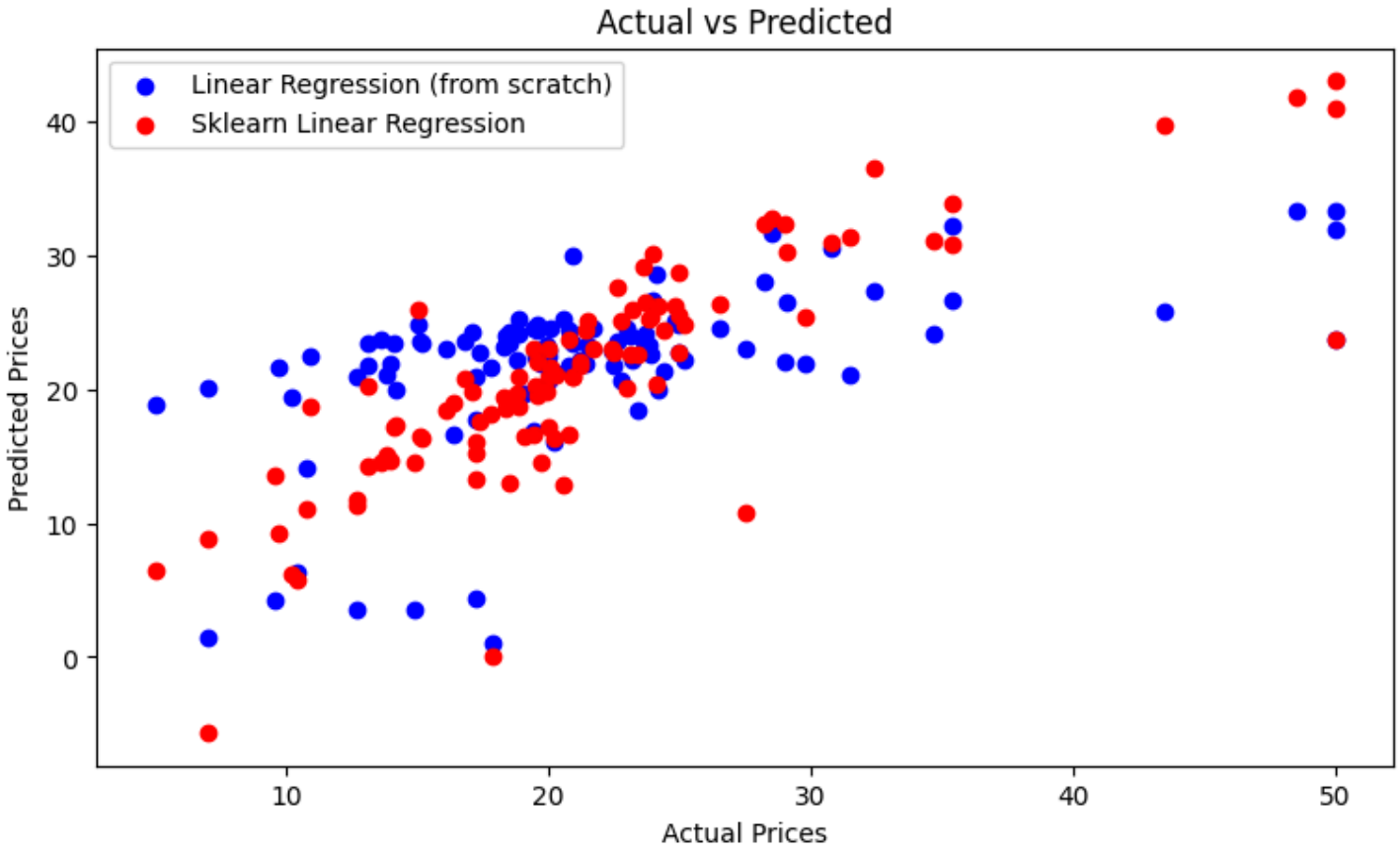
Results:

- Mean Squared Error **{MSE} = 53.15**



Comparison between (LR from scratch and library-based)

Algorithm	Mean Squared Error (MSE)
Linear Regression (from scratch)	**53.15**
Linear Regression (library-based)	**25.01**



Objective of Linear Regression:

The primary objective of linear regression is to establish a linear relationship between the dependent variable Y and the independent variables X, such that:

$$y = X \cdot w + b$$

X : Feature matrix (input variables).
w : Weights.
b : Bias.
y : Predicted output.

Key Components of the Algorithm:

a) Hypothesis Function:

The hypothesis function predicts the output Y as a linear combination of the input features:

$$\overset{\Lambda}{y} = \sum_{i=1}^{n} wi \cdot xi + b$$

n : Number of features.
wi : Weight corresponding to the i-th feature.
xi : Value of the i-th feature.
b : Bias term.

b) Loss Function:

The loss function quantifies the error between the predicted outputs (ŷ) and the actual outputs (y). Linear regression typically uses the Mean Squared Error (MSE) as the loss function:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(yi - \hat{yi})^2$$

N : Number of training samples.
yi : Actual value of the i-th sample.
ŷi : Predicted value of the i-th sample.

c) Optimization:

The algorithm optimizes the weights (w) and bias (b) to minimize the loss function. This is achieved using Gradient Descent, an iterative optimization technique.

Gradient Descent:

Gradient Descent is used to update the weights and bias iteratively by moving in the direction of the negative gradient of the loss function.

Weight and Bias Updates:

The gradients of the loss function with respect to the weights (w) and bias (b) are calculated as:

$$dw = -\frac{2}{N}\sum_{i=1}^{N} xij \cdot \left(yi - \overset{\Lambda}{y}i\right)$$

$$db = -\frac{2}{N}\sum_{i=1}^{N} \left(yi - \overset{\Lambda}{y}i\right)$$

Using these gradients, the weights and bias are updated as:

$$wj = wj - \alpha \cdot dw$$

$$b = b - \alpha \cdot db$$

(**a** : Learning rate, which controls the step size in the parameter space.)

Similarities:

- **Linear relationship:** Both models predict values that follow a linear pattern.
- **Good handling with Low-Value Predictions:** For houses with lower actual prices (e.g. in the range of 10–20), both models predict values relatively close to the actual prices, indicating similar behavior in this range.
- **Sensitivity to outlier:** Both models show some sensitivity to outliers (e.g. points far from the ideal line).

Discrepancies:

- **Accuracy Differences:** The from scratch LR deviate from the ideal line more than the library-based LR, indicating that the library-based implementation is more accurate.
- **Prediction Spread:** The scatter of the from scratch LR is wider, showing higher variance in predictions compared to the library-based LR.
- **Closeness to Ideal Line:** The library-based LR are generally closer to the ideal line. So, it`s **better model fitting** and **lower error metrics**.

Conclusion

Summary of findings:

- **For (LR & GB),**

  Gradient Boosting performs well in most cases, where it minimizes errors well. Linear Regression Performs well when the relationship between features and target is linear and data is noise-free, that because it fails to capture nonlinear relationships in the data, and highly sensitive to outliers.

  So, LR unable to handle with the database efficiently because it contains many non-linear relationships.

- **For (LR library-based & LR from scratch),**

  The library-based LR implementation demonstrates higher accuracy, and lower error metrics, where from scratch LR implementation is less practical for real-world applications compared to library-based methods, because its reduced accuracy and robustness, and has higher error rates.

  So, The library-based LR is the better for both accuracy and reliability in this case. The scratch implementation is a good way to learn how linear regression works, but isn't suitable for practical applications due to its higher error rates.