In [6]:

```
import warnings
warnings.filterwarnings("ignore")
from sklearn.datasets import load_boston
from random import seed
from random import randrange
from csv import reader
from math import sqrt
from sklearn import preprocessing
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from prettytable import PrettyTable
from sklearn.linear_model import SGDRegressor
from sklearn import preprocessing
from sklearn.metrics import mean_squared_error
```

In [7]:

```
X = load_boston().data
Y = load_boston().target
```

In [8]:

```
scaler = preprocessing.StandardScaler().fit(X)
X = scaler.transform(X)
```

In [9]:

```
clf = SGDRegressor()
clf.fit(X, Y)
print(mean_squared_error(Y, clf.predict(X)))
```

```
21.975595675005767
```

In [10]:

```
from sklearn.model_selection import train_test_split
boston = load_boston()
X_train, X_test, Y_train, Y_test = train_test_split(boston.data, boston.target, test_size=0.3, rand
om_state=4)
```

In [11]:

```
# Standardizing the Train and Test Data
# Perform fit on Train data and Transform on both Train and Test Data
scaler = preprocessing.StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [12]:

```
# To visualize how the data looks post standardization, lets convert it into a pandas dataframe
standardized_df = pd.DataFrame(X_train)
```

In [13]:

```
standardized_df['house_price'] = Y_train
```

In [14]:

```
def _customSGD(dataset, learning_rate, n_epochs, batch_size, initialize_random_coefs=False, variabl
e_learning=False):
    """
    # Stochastic Gradient Descent
```

```
    # Stochastic Gradient Descent
    -- Gradient Descent computes the cost gradient for all the samples involved in the dataset
       where as the SGD computes cost gradient of 1 sample at each iteration.
    -- Instead of one sample at a time, if a batch_size is provided it then becomes Batch GD

    Expectations:
        dataset - a pandas dataframe
        learning_rate - an intezer or float
        n_epochs - an intezer
        batch_Size - an intezer

    """

    # Raise TypeErrors when required inputs are of not desired format

    no_of_training_sample, no_of_features = dataset.shape
    no_of_features -= 1 # subtracing the labels column from features

    if initialize_random_coefs:
        # Random initialization
        bias = np.random.rand(1)
        weights = np.random.rand(no_of_features)
    else:
        # Initialization with zeros
        bias = 0
        weights = np.zeros(shape=(1,no_of_features),dtype="double")

    # for future add tqdm function hear to know the progress of epochs
    for _epoch in range(n_epochs):
        # updating learning rate for every 100 epochs
        if variable_learning == True:
            if _epoch%100 == 0:
                learning_rate /= 2.0

        # local variables for needing to predict and compute error
        _b, _w, _parderivateW, _parderivateB = bias, weights, np.zeros(shape=(1,no_of_features)), 0

        # since this going to be mini batch SGD
        miniBatch = dataset.sample(batch_size)

        #  reference - https://stackoverflow.com/questions/40144769/how-to-select-the-last-column-
of-dataframe
        ylabels = np.array(miniBatch.iloc[:,-1]) # labels column
        xfeatures = np.array(miniBatch.drop(miniBatch.columns[len(miniBatch.columns)-1], axis=1))

        for index in range(batch_size):
            # Refer first image for formulae
            # partial derivative w.r.t Weights
            # dl/dw = Summation(-2x * (y - (wTx+b)))
            _parderivateW += (-2) * (xfeatures[index]) * (ylabels[index] -
(np.dot(_w,xfeatures[index]) + _b))

            # partial derivate w.r.t bias
            # dl/db = Summation(-2 * (y - (wTx+b)))
            _parderivateB += (-2) * (ylabels[index] - (np.dot(_w, xfeatures[index]) + _b))

        # Updating the weights, bias at the end of epoch
        # refer second image for formulae
        bias = (_b - learning_rate*(_parderivateB)/batch_size)
        weights = (_w - learning_rate*(_parderivateW)/batch_size)

        # print (bias, weights, f"At the end of epoch {_epoch} out of {n_epochs}")


    return weights, bias
```

In [15]:

```
fweights, fbias = _customSGD(standardized_df, 0.01, 750, 25) # fixed learning rate
```

In [16]:

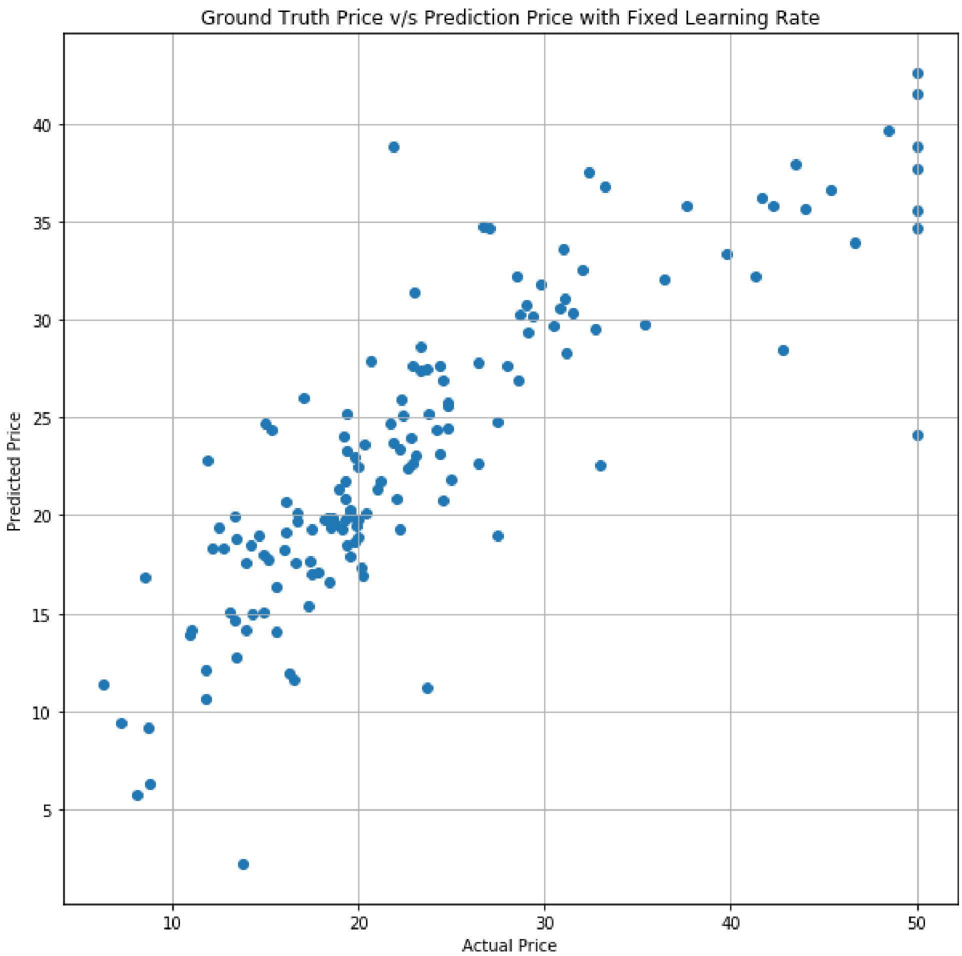
```
# variable learning rate
vweights, vbias = _customSGD(standardized_df, 0.01, 750, 25, False, True) # fixed learning rate
```

In [17]:

```
# since we are done with the training lets do inference
# with fixed learning rate
predicted_house_yhat_list = []
for index in range (len(X_test)):
    yhat = np.dot(fweights, X_test[index]) + fbias
    predicted_house_yhat_list.append(np.asscalar(yhat)) # converting the yhat prediction into a
scalar value
    # >>> np.asscalar(np.array([24]))
    # 24
```

In [18]:

```
# Lets plot Ground Truth Price v/s Predictions
plt.figure(figsize=(10,10))
plt.title("Ground Truth Price v/s Prediction Price with Fixed Learning Rate")
plt.scatter(Y_test, predicted_house_yhat_list)
plt.xlabel("Actual Price")
plt.ylabel("Predicted Price")
plt.grid()
plt.show()
```

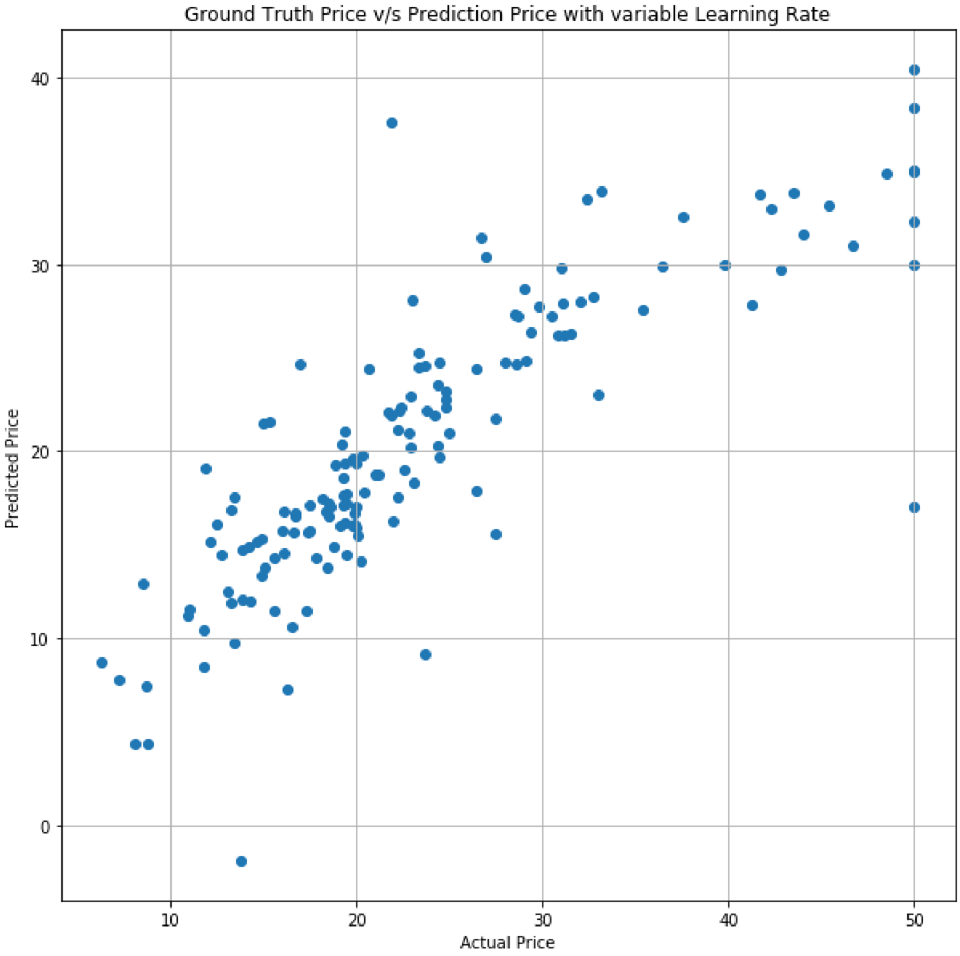


In [19]:

```
# with Variable Learning rate
vpredicted_house_yhat_list = []
for index in range (len(X_test)):
    yhat = np.dot(vweights, X_test[index]) + vbias
    vpredicted_house_yhat_list.append(np.asscalar(yhat)) # converting the yhat prediction into a
scalar value
    # >>> np.asscalar(np.array([24]))
    # 24
```

In [20]:

```python
# Lets plot Ground Truth Price v/s Predictions
plt.figure(figsize=(10,10))
plt.title("Ground Truth Price v/s Prediction Price with variable Learning Rate")
plt.scatter(Y_test, vpredicted_house_yhat_list)
plt.xlabel("Actual Price")
plt.ylabel("Predicted Price")
plt.grid()
plt.show()
```



In [21]:

```python
# Mean Squared Error - Fixed LR, Variable LR
MSE_flr = mean_squared_error(Y_test, predicted_house_yhat_list)
MSE_vlr = mean_squared_error(Y_test, vpredicted_house_yhat_list)

print (f"Fixed LR MSE {MSE_flr}")
print (f"Variable LR MSE {MSE_vlr}")
```

```
Fixed LR MSE 31.06756476146327
Variable LR MSE 41.78833042496772
```

In [22]:

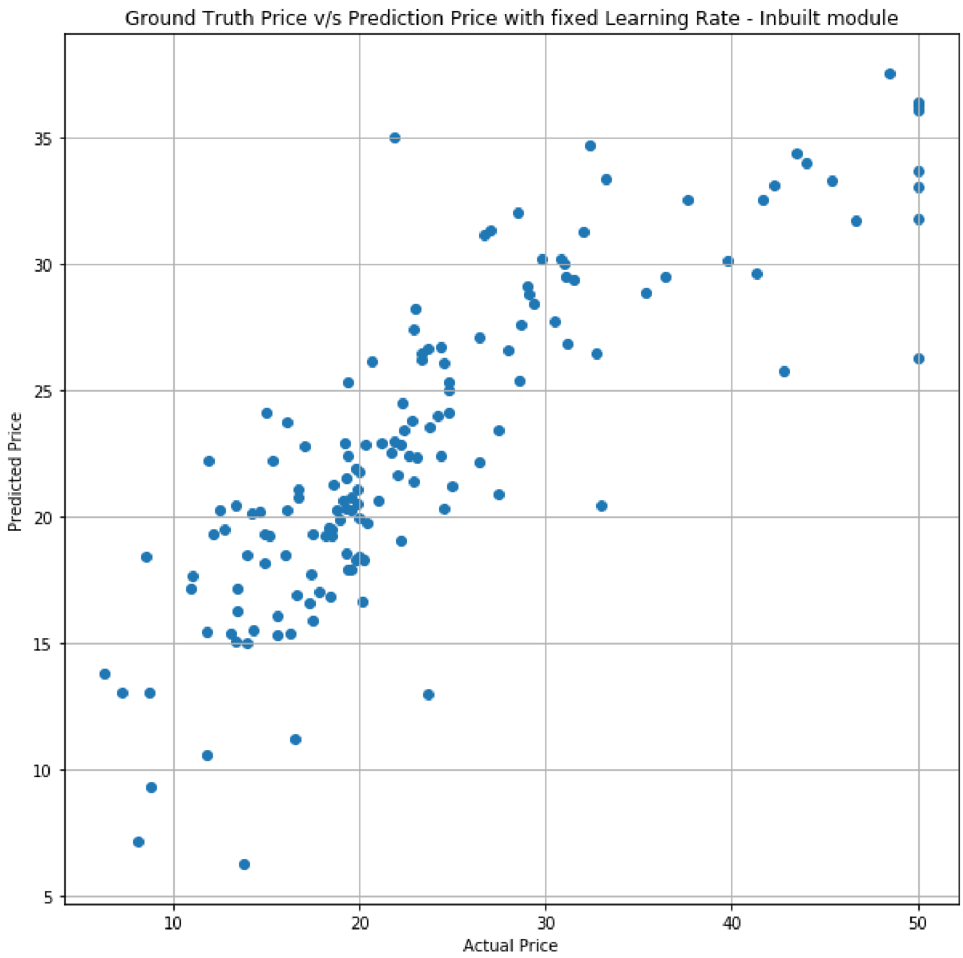
```python
# Lets perform the same with inbuilt module of sklearn

sgd = SGDRegressor(learning_rate="constant", eta0=0.01, penalty=None, max_iter=50)
sgd.fit(X_train,Y_train)
y_hat_sgd=sgd.predict(X_test)
```

In [23]:

```python
# Lets plot Ground Truth Price v/s Predictions
plt.figure(figsize=(10,10))
plt.title("Ground Truth Price v/s Prediction Price with fixed Learning Rate - Inbuilt module")
plt.scatter(Y_test, y_hat_sgd)
plt.xlabel("Actual Price")
plt.ylabel("Predicted Price")
```

```
plt.ylabel("Predicted Price")
plt.grid()
plt.show()
```



In [24]:

```
# Mean Squared Error - Fixed LR, Variable LR
MSE_flr_inbuilt = mean_squared_error(Y_test, y_hat_sgd)

print (f"Fixed LR MSE {MSE_flr_inbuilt}")
```

```
Fixed LR MSE 37.08703339008432
```

In [25]:

```
# A tabular representation of weights for all four modes
from prettytable import PrettyTable

x = PrettyTable()
x.field_names = ["Weights - Custom", "Weights - Inbuilt"]
```

In [26]:

```
weights_sgd = sgd.coef_
for i in range(12):
    x.add_row([fweights[0][i],weights_sgd[i]])
print(x)
```

```
+---------------------+---------------------+
|   Weights - Custom    |  Weights - Inbuilt   |
+---------------------+---------------------+
| -0.9385927690362353  | -1.0792868624043739  |
|  1.0174595381978706  |  1.3805327895293302  |
| -0.3224438781137728  | 0.08211540290493805  |
|  1.1706514529577208  |  0.774178097204397   |
| -1.3990051141124442  | -1.6425012389345985  |
```

```
|  2.139413801367768   |  1.3786522221311635  |
| 0.017691216646937923 | -0.22231724933133631 |
| -3.0315566718089433  | -3.1647867457066954  |
|  1.7901471838431788  |  2.9076406195048925  |
|  -1.251946525225137  | -1.7877015627118373  |
|  -1.780052843046521  | -1.3071867990851962  |
|  0.9242037466908599  |  0.2707159440103741  |
+----------------------+----------------------+
```

In [27]:

```
# Custom SGD v/s SGD sklearn implementation
print('MSE of manual implementation = ',MSE_flr)
print('-'*50)
print('MSE of SGD sklearn implementation = ',MSE_flr_inbuilt)
```

```
MSE of manual implementation =  31.06756476146327
--------------------------------------------------
MSE of SGD sklearn implementation =  37.08703339008432
```

In [ ]:

```
```