Implement SGD for Boston House Dataset

In [0]:

```
# Importing libraries
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
from prettytable import PrettyTable
import seaborn as sns
```

In [160]:

```
Boston_data=pd.DataFrame(load_boston().data,columns=load_boston().feature_names)
Boston_data.head(2)
```

Out[160]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.9	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.9	9.14

In [161]:

```
Boston_data['Price'] = load_boston().target
SK_X = Boston_data.drop('Price', axis = 1)
SK_Y = Boston_data['Price']
# Splitting the data into train and test
SK_x_train,SK_x_Test,SK_y_train,SK_y_Test = train_test_split(SK_X,SK_Y,test_size
=0.33,random_state=0)
print(SK_x_train.shape)
print(SK_x_Test.shape)
print(SK_y_train.shape)
print(SK_y_Test.shape)
SK_x_train.mean()

# Data Standardization
std = StandardScaler()
SK_x_train = std.fit_transform(SK_x_train)
SK_x_Test = std.fit_transform(SK_x_Test)

SK_SGD_reg = SGDRegressor()
SK_SGD_reg.fit(SK_x_train, SK_y_train)
SK_y_pred = SK_SGD_reg.predict(SK_x_Test)
SkLearn_w=SK_SGD_reg.coef_
print("Sklearn's SGD regrtessor Coefficients: ", SK_SGD_reg.coef_)
print("Sklearn's SGD regrtessor y intercept:", SK_SGD_reg.intercept_)
```

```
(339, 13)
(167, 13)
(339,)
(167,)
Sklearn's SGD regrtessor Coefficients:  [-0.88878007  0.88548215 -0.29749873  0.70480566 -
1.55734509  2.76746094
 -0.36332282 -2.97019182  1.26185714 -0.83765784 -2.28225084  0.51986752
 -3.535913   ]
Sklearn's SGD regrtessor y intercept: [22.82012928]
```

Building Stochastic Gradient Decent(SGD) Regressor for Linear Regression

In [162]:

```
# Loading data again for custom implementation of SGD Regressor
CUS_boston = load_boston()
CUS_boston.data.shape
CUS_boston.feature_names
CUS_boston.target.shape
CUS_boston_data = pd.DataFrame(CUS_boston.data, columns = CUS_boston.feature_names)
print(CUS_boston_data.head())
#normlization of data
CUS_boston_data = (CUS_boston_data - CUS_boston_data.mean())/CUS_boston_data.std()
print(CUS_boston_data.head())
print(CUS_boston_data.mean())
CUS_boston_data["Price"] = CUS_boston.target
CUS_boston_data.head()
CUS_Y = CUS_boston_data["Price"]
CUS_X = CUS_boston_data.drop("Price", axis = 1)
x_train_cu,x_Test_cu,y_train_cu,y_Test_cu=train_test_split(CUS_X,CUS_Y,test_size=0.3,random_state=
0)
print(x_train_cu.shape,y_train_cu.shape,x_Test_cu.shape,y_Test_cu.shape)
x_train_cu["Price"] = y_train_cu
```

```
      CRIM    ZN  INDUS  CHAS    NOX  ...  RAD    TAX  PTRATIO       B  LSTAT
0  0.00632  18.0   2.31   0.0  0.538  ...  1.0  296.0     15.3  396.90   4.98
1  0.02731   0.0   7.07   0.0  0.469  ...  2.0  242.0     17.8  396.90   9.14
2  0.02729   0.0   7.07   0.0  0.469  ...  2.0  242.0     17.8  392.83   4.03
3  0.03237   0.0   2.18   0.0  0.458  ...  3.0  222.0     18.7  394.63   2.94
4  0.06905   0.0   2.18   0.0  0.458  ...  3.0  222.0     18.7  396.90   5.33

[5 rows x 13 columns]
       CRIM        ZN     INDUS  ...   PTRATIO         B     LSTAT
0 -0.419367  0.284548 -1.286636  ... -1.457558  0.440616 -1.074499
1 -0.416927 -0.487240 -0.592794  ... -0.302794  0.440616 -0.491953
2 -0.416929 -0.487240 -0.592794  ... -0.302794  0.396035 -1.207532
3 -0.416338 -0.487240 -1.305586  ...  0.112920  0.415751 -1.360171
4 -0.412074 -0.487240 -1.305586  ...  0.112920  0.440616 -1.025487

[5 rows x 13 columns]
CRIM       8.326673e-17
ZN         3.466704e-16
INDUS     -3.016965e-15
CHAS       3.999875e-16
NOX        3.563575e-15
RM        -1.149882e-14
AGE       -1.158274e-15
DIS        7.308603e-16
RAD       -1.068535e-15
TAX        6.534079e-16
PTRATIO   -1.084420e-14
B          8.117354e-15
LSTAT     -6.494585e-16
dtype: float64
(354, 13) (354,) (152, 13) (152,)
```

In [0]:

```
# https://www.geeksforgeeks.org/ml-r-squared-in-regression-analysis/
def error_fn(b_val,w_val,x_mat,y_mat):
    err = 0
    for i in range(0, len(x_mat)):
        calc1=y_mat[:,i] - (np.dot(x_mat[i] , w_val) + b_val) #Σ(i=1 to k)[y[i]-(x[i].w^T)+b]
        err += (calc1) ** 2                                  #Residual sum of squares(SS_res)
    return err/len(x_mat)                                    #average or mean of Residual sum of s
uares
```

In [0]:

```
# https://www.kaggle.com/premvardhan/stocasticgradientdescent-implementation-lr-python
def SGDreg(w, b, train, x_Test, y_Test, r):
    '''Custom implementation of SGD Gradient Descent for Linear regression'''

    dm=0
    db=0
```

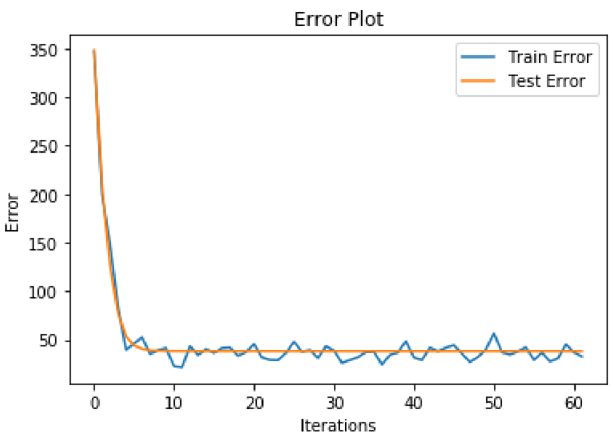
```
    itr=1000                                #setting the number of iterations
    error_train=[]
    error_test=[]
    for j in range(1,itr):
        train_batch=train.sample(100)      #creating a batch of Boston House dataset with size = 10(
        x = np.asmatrix(train_batch.drop("Price", axis = 1))
        y = np.asmatrix(train_batch["Price"])
        for i in range(len(x)):
            tval=y[:,i]-np.dot(x[i],w)+b  #Σ(i=1 to k)[y_i-(x_i.w^T)+(-b)]
            dm+=np.dot(-2*x[i].T,tval)     #dL/dw=Σ(i=1 to k)[(-2*x_i)(y_i-(x_i.w^T)+(-b))]
            db+=(-2*tval)                  #dL/db=Σ(i=1 to k)[(-2)(y_i-(x_i.w^T)+(-b))]
        wn=w-(r*dm)                        #(w_j+1)^T=(w_j)^T-(r*(dL/dw))
        bn=b-(r*db)                        #b_j+1=b_j-(r*(dL/db))
        if (w==wn).all():                  #checking if the weight is saturated
            break                          # Breaking out of the for loop if the weight is saturate(
        else:
            w=wn                           #updating weights values (w^T)
            b=bn                           #updating intercept values (b)
            r/=2                           #cutting the learning rate by half of its previous value
        # calculating the error in the custom built SGD regressor for train(sample) data
        error_tr=error_fn(b,w,x,y)
        error_train.append(error_tr)
        # calculating the error in the custom built SGD regressor for test data
        error_ts=error_fn(b,w,np.asmatrix(x_Test),np.asmatrix(y_Test))
        error_test.append(error_ts)
    return w,b,error_train,error_test
```

In [165]:

```
r = 0.001                      #choosing the value for learning rate (for any other value like r=0.1
,0.01,0.0001,etc the custom SGD performs poorly)
# Choosing random values for w (wieght vector) and b (intercept)
b0 = np.random.rand()
rand_w0 = np.random.rand(x_train_cu.shape[1]-1)
w0 = np.asmatrix(rand_w0).T   #taking the transpose of weight matrix
w, b, error_train, error_test = SGDreg(w0, b0, CUS_boston_data, x_Test_cu, y_Test_cu, r)
print("SGD Coefficient: {}".format(w))
print("y_intercept: {}".format(b))

print("train error = {}".format(error_train))
print("Test error= {} ".format(error_test))

cu_pred=(np.dot(np.asmatrix(x_Test_cu), w) + b) #price prediction for test data with w,b of the
custom SGD regressor ((x[i].w^T)+b)
cu_pred_arr=np.array(cu_pred).T[0]              #convertiing 'cu_pred' - matrix into numpy array
# Error Plot
plt.figure()
plt.plot(range(len(error_train)), np.reshape(error_train,[len(error_train), 1]), label = "Train Err
or")
plt.plot(range(len(error_test)), np.reshape(error_test, [len(error_test), 1]), label = "Test Error"
)
plt.title("Error Plot")
plt.xlabel("Iterations")
plt.ylabel("Error")
plt.legend()
plt.show()
```

```
SGD Coefficient: [[-1.04387098]
 [ 1.23839574]
 [ 0.2814937 ]
 [ 0.49335625]
 [-0.14371758]
 [ 1.37212986]
 [-0.53490455]
 [ 0.24707012]
 [ 0.15402765]
 [-0.16013036]
 [-0.13213302]
 [ 1.54272569]
 [-3.28812855]]
y_intercept: [[22.68109851]]
train error = [matrix([[347.51091877]]), matrix([[199.60701633]]), matrix([[148.33782689]]),
matrix([[84.60387513]]), matrix([[39.23325899]]), matrix([[45.74767741]]),
matrix([[52.37588088]]), matrix([[35.04473186]]), matrix([[39.01706192]]),
matrix([[41.24442983]]), matrix([[22.5733563]]), matrix([[21.24701313]]), matrix([[43.20304737]]),
```

```
matrix([[34.05284954]]), matrix([[40.21292378]]), matrix([[36.20920841]]),
matrix([[41.17906772]]), matrix([[41.84965291]]), matrix([[33.28806609]]),
matrix([[37.00640814]]), matrix([[45.28784472]]), matrix([[31.52899687]]),
matrix([[29.07551213]]), matrix([[28.84256353]]), matrix([[36.2289394]]), matrix([[47.6478098]]),
matrix([[37.53689876]]), matrix([[39.19550781]]), matrix([[30.9929997]]), matrix([[43.1582019]]),
matrix([[38.63448933]]), matrix([[26.00098565]]), matrix([[28.94720946]]),
matrix([[31.76556213]]), matrix([[37.16366417]]), matrix([[37.40976322]]),
matrix([[24.18513833]]), matrix([[34.18186102]]), matrix([[36.6861564]]), matrix([[48.35317454]]),
matrix([[31.5711327]]), matrix([[28.9209472]]), matrix([[41.75417952]]), matrix([[37.95769695]]),
matrix([[41.63916315]]), matrix([[44.22135338]]), matrix([[35.69371649]]),
matrix([[26.78475431]]), matrix([[32.06861558]]), matrix([[39.86159958]]),
matrix([[56.50667751]]), matrix([[36.95221425]]), matrix([[34.48283889]]),
matrix([[37.78075307]]), matrix([[42.06555274]]), matrix([[29.06766471]]),
matrix([[36.57578527]]), matrix([[27.59346434]]), matrix([[30.53401058]]),
matrix([[44.93038948]]), matrix([[36.9618887]]), matrix([[32.55933609]])]
Test error= [matrix([[347.92232223]]), matrix([[208.32581255]]), matrix([[126.00288306]]),
matrix([[77.33189145]]), matrix([[53.45879963]]), matrix([[44.00652747]]),
matrix([[40.30962251]]), matrix([[38.92246778]]), matrix([[38.45728348]]),
matrix([[38.25680179]]), matrix([[38.15374859]]), matrix([[38.1106992]]), matrix([[38.08513059]]),
matrix([[38.07225388]]), matrix([[38.06582448]]), matrix([[38.06183641]]),
matrix([[38.05967399]]), matrix([[38.05851286]]), matrix([[38.05794631]]),
matrix([[38.05762958]]), matrix([[38.05746513]]), matrix([[38.05737445]]),
matrix([[38.05732198]]), matrix([[38.05729525]]), matrix([[38.05728179]]),
matrix([[38.05727483]]), matrix([[38.05727149]]), matrix([[38.05726973]]),
matrix([[38.05726882]]), matrix([[38.05726834]]), matrix([[38.05726809]]),
matrix([[38.05726798]]), matrix([[38.05726792]]), matrix([[38.05726789]]),
matrix([[38.05726787]]), matrix([[38.05726786]]), matrix([[38.05726786]]),
matrix([[38.05726785]]), matrix([[38.05726785]]), matrix([[38.05726785]]),
matrix([[38.05726785]]), matrix([[38.05726785]]), matrix([[38.05726785]]),
matrix([[38.05726785]]), matrix([[38.05726785]]), matrix([[38.05726785]]),
matrix([[38.05726785]]), matrix([[38.05726785]]), matrix([[38.05726785]]),
matrix([[38.05726785]]), matrix([[38.05726785]]), matrix([[38.05726785]]),
matrix([[38.05726785]]), matrix([[38.05726785]]), matrix([[38.05726785]]),
matrix([[38.05726785]]), matrix([[38.05726785]]), matrix([[38.05726785]]),
matrix([[38.05726785]]), matrix([[38.05726785]]), matrix([[38.05726785]]),
matrix([[38.05726785]])]
```



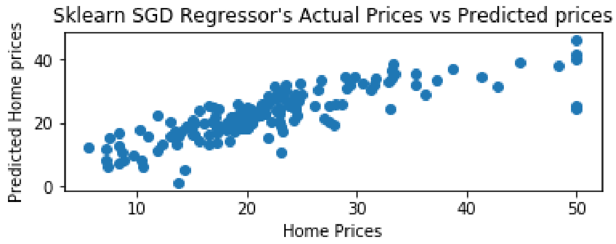
Comparison between sklearn's SGD Regressor and SGDreg

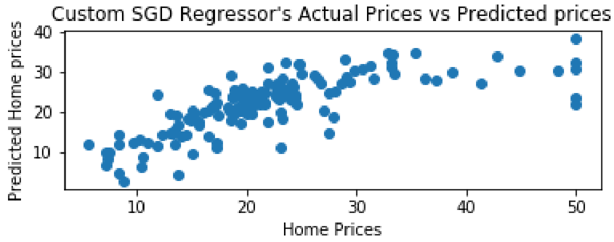
Scatter plot for Actual vs predicted target values

In [166]:

```
# sklearn SGD Regressor
plt.figure(1)
plt.subplot(211)
plt.scatter(SK_y_Test, SK_y_pred)
plt.xlabel("Home Prices")
plt.ylabel("Predicted Home prices")
plt.title("Sklearn SGD Regressor's Actual Prices vs Predicted prices")
plt.show()

# Custom SGD Regressor
plt.subplot(212)
plt.scatter([y_Test_cu], [(np.dot(np.asmatrix(x_Test_cu), w) + b)])
plt.xlabel("Home Prices")
plt.ylabel("Predicted Home prices")
plt.title("Custom SGD Regressor's Actual Prices vs Predicted prices")
```

```
plt.title("Custom SGD Regressor's Actual Prices vs Predicted prices")
plt.show()
```





In [167]:

```
# Sklearn SGD Regression
print("Mean squared error of Sk learn's prediction:",format(mean_squared_error(SK_y_Test,
SK_y_pred)))
print("r2_score in Sk learn's prediction:",format(r2_score(SK_y_Test, SK_y_pred)))
```

```
Mean squared error of Sk learn's prediction: 28.177030927439702
r2_score in Sk learn's prediction: 0.6505050416730043
```

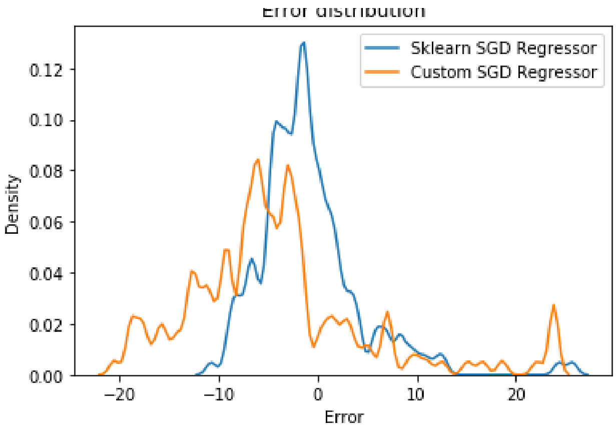
In [168]:

```
# r2_score implementation : https://www.geeksforgeeks.org/ml-r-squared-in-regression-analysis/
# SGDreg
CU_error = error_fn(b, w, np.asmatrix(x_Test_cu), np.asmatrix(y_Test_cu))
print("Mean squared error of SGD_reg :",format(float(CU_error)))

x_mat=np.asmatrix(x_Test_cu)
y_mat=np.asmatrix(y_Test_cu)
for i in range(0, len(x_mat)):
    y_mean = np.mean(y_mat)
    calc2=(y_mat[:,i] - y_mean)                         #Σ(i=1 to n)[y[i]-mean(y)]
    calc3=(y_mat[:,i] - (np.dot(x_mat[i], w) + b))     #Σ(i=1 to n)[y[i]-(x[i].w^T)+b]
    sq_sum = sum((calc2)**2)                            #total sum of squares(sq_sum)
    res_sum = sum((calc3)**2)                           #Residual sum of squares(res_sum)
    r2 = 1-(res_sum/sq_sum)                             #r^2=1-(Residual sum of squares/total sum of sq
ares)

print("r2_score in SGDreg :",format(float(r2)))
```

```
Mean squared error of SGD_reg : 38.05726785171662
r2_score in SGDreg : 0.9329136278904404
```

In [176]:

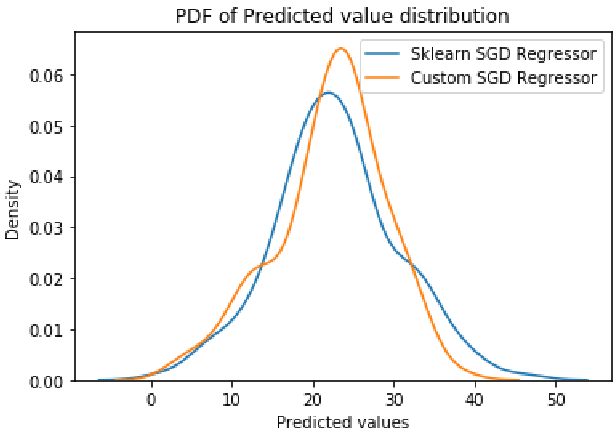
```
# Error distribution
cu_y_err = np.asmatrix(y_Test_cu) - (cu_pred)
SK_y_err = SK_y_Test - SK_y_pred

sns.kdeplot(np.array(SK_y_err), label = "Sklearn SGD Regressor", bw = 0.5)
sns.kdeplot(np.asarray(cu_y_err)[0], label = "Custom SGD Regressor", bw = 0.5)
plt.title("Error distribution")
plt.xlabel("Error")
plt.ylabel("Density")
plt.legend()
plt.show()
```



Observation: The SkLearn's SGD Regressor is a little better than Custom SGD Regressor with respect to Error because the spread is lower for SKlearn's SGD and the peak is at '0'.

In [170]:

```
# Predicted value distribution
sns.kdeplot(SK_y_pred, label = "Sklearn SGD Regressor")
sns.kdeplot(cu_pred_arr, label = "Custom SGD Regressor")
plt.title("PDF of Predicted value distribution")
plt.xlabel("Predicted values")
plt.ylabel("Density")
plt.show()
```



Observation: Though for the most part the prediction values overlap for both the models, the peak of the Custom SGD regressor's prediction is slightly moved to the right and Sklearn's tail is slightly skewed to the right, showing the subtle difference in the two model's predictions.

Comparison with Pretty Table

In [175]:

```
x = PrettyTable()
x.field_names=['weight(w) - Custom SGD Regressor','weight(w) - SkLearn SGD Regressor']
for i in range(x_train_cu.shape[1]-1):
    x.add_row([float(w[i]),SkLearn_w[i]])
print(x)
```

```
+-----------------------------------+-----------------------------------+
| weight(w) - Custom SGD Regressor | weight(w) - SkLearn SGD Regressor |
+-----------------------------------+-----------------------------------+
|        -1.0438709833143671        |        -0.8887800670583803        |
|         1.2383957394164495        |         0.885482153925777         |
|        0.28149370471031887        |        -0.29749873104191304       |
|         0.4933562463661075        |         0.7048056575323272        |
|        -0.1437175830641326        |         -1.55734508749089         |
|         1.372129860673849         |         2.7674609435876354        |
|        -0.5349045530767805        |        -0.36332282209978634       |
```

0.24707012438522966	-2.9701918188292757
0.15402765191090925	1.2618571424006537
-0.16013036442488496	-0.8376578433223723
-0.132133017062744	-2.2822508429156914
1.5427256893244776	0.5198675212045898
-3.2881285467169623	-3.53591300290038

Summary: Both the SkLearn's SGD Regressor and the Custom built SGD Regressor are very similar with regards to the weights and the corresponding predictions. The Custom built SGD regressor model's 'Mean Squared Error' is a bit higher than that of the Sklearn's SGD model. Likewise, the r square value is a bit lower for the former than the latter. This indicates the Sklearn's SGD model is a little better than the custom SGD model which we have implemented.