Boston House price prediction using SGD

In this kernel we will be implementing SGD on LinearRegression from scarch using python and we will be also comparing sklearn implementation SGD and our implemented SGD.

In [38]:
```
import pandas as pd
import numpy as np
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
boston = load_boston()
```

In [39]:
```
#REFERENCES


#1)https://www.kaggle.com/premvardhan/stocasticgradientdescent-implementation-lr-python
#2)https://medium.com/@nikhilparmar9/simple-sgd-implementation-in-python-for-linear-regre
```

In [2]:
```
print(boston.data.shape)
```

```
(506, 13)
```

In [3]:
```
print(boston.feature_names)
```

```
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

In [4]:
```
print(boston.target.shape)
```

```
(506,)
```

In [5]:
```
print(boston.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14)
is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.
```

```
This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/


This dataset was taken from the StatLib library which is maintained at Carnegie Mellon Uni
versity.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address reg
ression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sour
ces of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings
on the Tenth International Conference of Machine Learning, 236-243, University of Massachu
setts, Amherst. Morgan Kaufmann.
```

In [6]:

```python
# Loading data into pandas dataframe
bos = pd.DataFrame(boston.data)
print(bos.head())
```

```
         0     1     2    3      4      5     6       7    8      9    10  \
0  0.00632  18.0  2.31  0.0  0.538  6.575  65.2  4.0900  1.0  296.0  15.3
1  0.02731   0.0  7.07  0.0  0.469  6.421  78.9  4.9671  2.0  242.0  17.8
2  0.02729   0.0  7.07  0.0  0.469  7.185  61.1  4.9671  2.0  242.0  17.8
3  0.03237   0.0  2.18  0.0  0.458  6.998  45.8  6.0622  3.0  222.0  18.7
4  0.06905   0.0  2.18  0.0  0.458  7.147  54.2  6.0622  3.0  222.0  18.7

       11    12
0  396.90  4.98
1  396.90  9.14
2  392.83  4.03
3  394.63  2.94
4  396.90  5.33
```

In [7]:

```python
bos['PRICE'] = boston.target

X = bos.drop('PRICE', axis = 1)
Y = bos['PRICE']
```

In [8]:

```python
# Split data into train and test
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.33, random_state =
print(X_train.shape)
print(X_test.shape)
print(Y_train.shape)
print(Y_test.shape)
```

```
(339, 13)
(167, 13)
(339,)
(167,)
```

In [9]:

```python
X_train.mean()
```

Out[9]:

```
0      3.510706
1     11.233038
2     10.946755
3      0.061947
4      0.552433
```

Loading [MathJax]/extensions/Safe.js

```
6      67.433923
7       3.792998
8       9.587021
9     404.988201
10     18.456342
11    359.382950
12     12.522360
dtype: float64
```

In [10]:
```
# Standardization

from sklearn.preprocessing import StandardScaler
std = StandardScaler()
X_train = std.fit_transform(X_train)
X_test = std.fit_transform(X_test)
```

In [11]:
```
X_train
```

Out[11]:
```
array([[ 0.9118389 , -0.50241886,  1.07230484, ...,  0.80807825,
        -2.84295938,  1.52320257],
       [-0.41172732, -0.50241886, -1.12979483, ..., -0.30417427,
         0.42743634, -0.99523956],
       [ 0.12458293, -0.50241886,  1.07230484, ...,  0.80807825,
        -0.05335342, -0.76564608],
       ...,
       [-0.39713851, -0.50241886, -0.18839347, ...,  0.3446397 ,
         0.38630716,  0.71962537],
       [-0.3910951 , -0.50241886, -0.05347927, ...,  0.06657657,
         0.4043083 , -0.22000723],
       [-0.40576854,  3.07573229, -1.35465184, ...,  1.64226764,
         0.18977581, -0.98531886]])
```

In [12]:
```
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error, r2_score
clf = SGDRegressor()
clf.fit(X_train, Y_train)
Y_pred = clf.predict(X_test)

print("Coefficients: \n", clf.coef_)
print("Y_intercept", clf.intercept_)
```

```
Coefficients: 
 [-1.22945505  0.74834972 -0.46992824  0.23260652 -1.31220445  2.85392643
 -0.39419172 -2.71710236  1.93366824 -1.19037278 -2.07890377  1.03875059
 -3.30830173]
Y_intercept [22.54686379]
```

Observations

- Overall we can say the regression line not fits data perfectly but it is okay. But our goal is to find the line/plane that best fits our data means minimize the error i.e. mse should be close to 0.
- MSE is 28.54 means the total loss(squared difference of true/actual target value and predicted target value). 0.0 is perfect i.e. no loss.
- coefficient of determination tells about the goodness of fit of a model and here, r^2 is 0.70 which means regression prediction does not perfectly fit the data. An r^2 of 1 indicates that regression prediction perfect fit the data.

Stochastic Gradient Decent(SGD) for Linear Regression

```
# ...ssary libraries
```

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
```

In [14]:
```
# Data loaded
bostan = load_boston()
```

In [15]:
```
# Data shape
bostan.data.shape
```

Out[15]: (506, 13)

In [16]:
```
# Feature name
bostan.feature_names
```

Out[16]:
```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')
```

In [17]:
```
# This is y value i.e. target
bostan.target.shape
```

Out[17]: (506,)

In [18]:
```
# Convert it into pandas dataframe
data = pd.DataFrame(bostan.data, columns = bostan.feature_names)
data.head()
```

Out[18]:

	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	396.90	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	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

In [19]:
```
# Statistical summary
data.describe()
```

Out[19]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	D
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.0000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.7950
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.1057
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.1296
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.1001
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.2074
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.1884
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.1265

In [20]:
```
#noramlization for fast convergence to minima
data = (data - data.mean())/data.std()
data.head()
```

Out[20]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	
0	-0.419367	0.284548	-1.286636	-0.272329	-0.144075	0.413263	-0.119895	0.140075	-0.981871	-0.6
1	-0.416927	-0.487240	-0.592794	-0.272329	-0.739530	0.194082	0.366803	0.556609	-0.867024	-0.9
2	-0.416929	-0.487240	-0.592794	-0.272329	-0.739530	1.281446	-0.265549	0.556609	-0.867024	-0.9
3	-0.416338	-0.487240	-1.305586	-0.272329	-0.834458	1.015298	-0.809088	1.076671	-0.752178	-1.1
4	-0.412074	-0.487240	-1.305586	-0.272329	-0.834458	1.227362	-0.510674	1.076671	-0.752178	-1.1

In [21]:

```
data.mean()
```

Out[21]:

```
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
```

In [23]:

```
# MEDV(median value is usually target), change it to price
data["PRICE"] = bostan.target
data.head()
```

Out[23]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	
0	-0.419367	0.284548	-1.286636	-0.272329	-0.144075	0.413263	-0.119895	0.140075	-0.981871	-0.6
1	-0.416927	-0.487240	-0.592794	-0.272329	-0.739530	0.194082	0.366803	0.556609	-0.867024	-0.9
2	-0.416929	-0.487240	-0.592794	-0.272329	-0.739530	1.281446	-0.265549	0.556609	-0.867024	-0.9
3	-0.416338	-0.487240	-1.305586	-0.272329	-0.834458	1.015298	-0.809088	1.076671	-0.752178	-1.1
4	-0.412074	-0.487240	-1.305586	-0.272329	-0.834458	1.227362	-0.510674	1.076671	-0.752178	-1.1

In [24]:

```
# Target and features
Y = data["PRICE"]
X = data.drop("PRICE", axis = 1)
```

In [25]:

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size = 0.3)
print(x_train.shape, x_test.shape, y_train.shape, y_test.shape)
```

```
(354, 13) (152, 13) (354,) (152,)
```

In [26]:

```
x_train.insert(x_train.shape[1],"PRICE",y_train)
```

In [27]:

```
def cost_function(b, m, features, target):
    totalError = 0
    for i in range(0, len(features)):
        x = features
        y = target
        totalError += (y[:,i] - (np.dot(x[i] , m) + b)) ** 2
    return totalError / len(x)
```

In [28]:

```
def r_sq_score(b, m, features, target):
    ge(0, len(features)):
```

Loading [MathJax]/extensions/Safe.js

```
        x = features
        y = target
        mean_y = np.mean(y)
        ss_tot = sum((y[:,i] - mean_y) ** 2)
        ss_res = sum(((y[:,i]) - (np.dot(x[i], m) + b)) ** 2)
        r2 = 1 - (ss_res / ss_tot)
    return r2
```

In [29]:

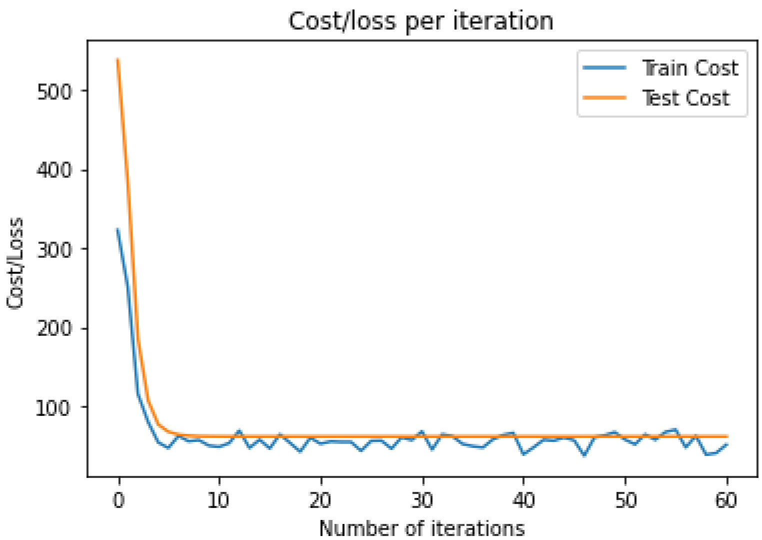
```
def gradient_decent(w0, b0, train_data, x_test, y_test, learning_rate):
    n_iter = 500
    partial_deriv_m = 0
    partial_deriv_b = 0
    cost_train = []
    cost_test = []
    for j in range(1, n_iter):

        # Train sample
        train_sample = train_data.sample(160)
        y = np.asmatrix(train_sample["PRICE"])
        x = np.asmatrix(train_sample.drop("PRICE", axis = 1))

        for i in range(len(x)):
            partial_deriv_m += np.dot(-2*x[i].T , (y[:,i] - np.dot(x[i] , w0) + b0))
            partial_deriv_b += -2*(y[:,i] - (np.dot(x[i] , w0) + b0))

        w1 = w0 - learning_rate * partial_deriv_m
        b1 = b0 - learning_rate * partial_deriv_b

        if (w0==w1).all():

            break
        else:
            w0 = w1
            b0 = b1
            learning_rate = learning_rate/2


        error_train = cost_function(b0, w0, x, y)
        cost_train.append(error_train)
        error_test = cost_function(b0, w0, np.asmatrix(x_test), np.asmatrix(y_test))
        cost_test.append(error_test)



    return w0, b0, cost_train, cost_test
```

In [30]:

```
learning_rate = 0.001
w0_random = np.random.rand(13)
w0 = np.asmatrix(w0_random).T
b0 = np.random.rand()

optimal_w, optimal_b, cost_train, cost_test = gradient_decent(w0, b0, x_train, x_test, y_
print("Coefficient: {} \n y_intercept: {}".format(optimal_w, optimal_b))

'''
error = cost_function(optimal_b, optimal_w, np.asmatrix(x_test), np.asmatrix(y_test))
print("Mean squared error:",error)
'''

plt.figure()
plt.plot(range(len(cost_train)), np.reshape(cost_train,[len(cost_train), 1]), label = "Tr
plt.plot(range(len(cost_test)), np.reshape(cost_test, [len(cost_test), 1]), label = "Test
plt.title("Cost/loss per iteration")
plt.xlabel("Number of iterations")
                      /Loss")
```

```
plt.legend()
plt.show()
```

```
Coefficient: [[-0.72932489]
 [ 0.81341438]
 [-1.27664679]
 [ 5.54373886]
 [ 1.46622733]
 [ 5.00495789]
 [-0.51314086]
 [-1.6506965 ]
 [-0.33077045]
 [-1.26977392]
 [-1.66383302]
 [ 1.10768272]
 [-0.74487424]]
 y_intercept: [[21.06302604]]
```



observations

1)as per number of iterations there is no change in error rate

Comparison between sklearn SGD and implemented SGD in python

In [31]:
```
print("Mean squared error: %.2f" % mean_squared_error(Y_test, Y_pred))

print("Variance score: %.2f" % r2_score(Y_test, Y_pred))
```

```
Mean squared error: 28.32
Variance score: 0.70
```

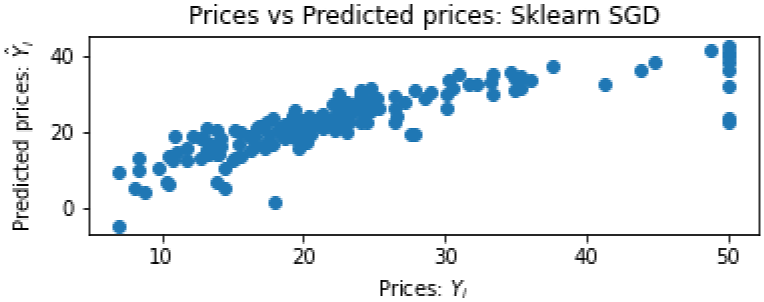
In [32]:
```
error = cost_function(optimal_b, optimal_w, np.asmatrix(x_test), np.asmatrix(y_test))
print("Mean squared error: %.2f" % (error))

r_squared = r_sq_score(optimal_b, optimal_w, np.asmatrix(x_test), np.asmatrix(y_test))
print("Variance score: %.2f" % r_squared)
```

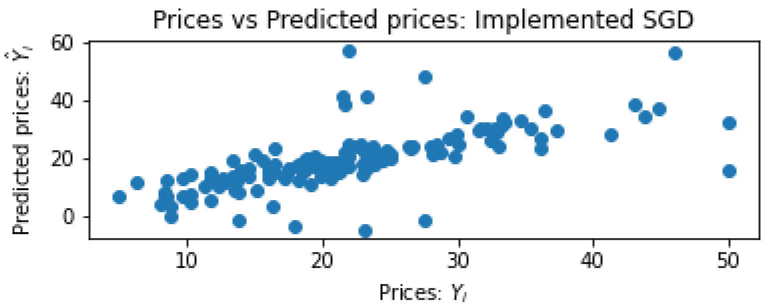
```
Mean squared error: 61.45
Variance score: 0.99
```

In [33]:
```
plt.figure(1)
plt.subplot(211)
plt.scatter(Y_test, Y_pred)
plt.xlabel("Prices: $Y_i$")
plt.ylabel("Predicted prices: $\hat{Y}_i$")
plt.title("Prices vs Predicted prices: Sklearn SGD")
```

Loading [MathJax]/extensions/Safe.js

```
plt.show()

# Implemented SGD
plt.subplot(212)
plt.scatter([y_test], [(np.dot(np.asmatrix(x_test), optimal_w) + optimal_b)])
plt.xlabel("Prices: $Y_i$")
plt.ylabel("Predicted prices: $\hat{Y}_i$")
plt.title("Prices vs Predicted prices: Implemented SGD")
plt.show()
```

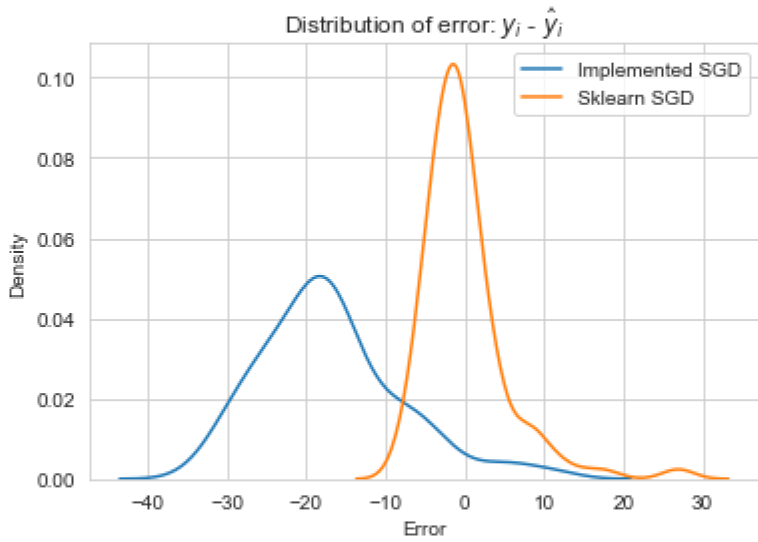




observations

1) Custom SGD and sklearn inbuild SGD almost gives same result

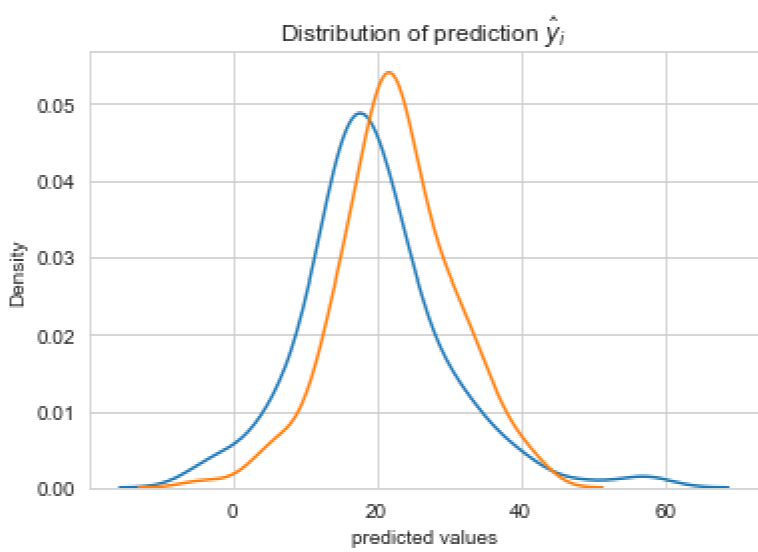
In [34]:
```
# Distribution of error
delta_y_im = np.asmatrix(y_test) - (np.dot(np.asmatrix(x_test), optimal_w) + optimal_b)
delta_y_sk = Y_test - Y_pred
import seaborn as sns;
import numpy as np;
sns.set_style('whitegrid')
sns.kdeplot(np.asarray(delta_y_im)[0], label = "Implemented SGD")
sns.kdeplot(np.array(delta_y_sk), label = "Sklearn SGD")
plt.title("Distribution of error: $y_i$ - $\hat{y}_i$")
plt.xlabel("Error")
plt.ylabel("Density")
plt.legend()
plt.show()
```



observation

1) Implemented SGD gives positive side of error more than negative side errors

In [35]:
```
# Distribution of predicted value
sns.set_style('whitegrid')
sns.kdeplot(np.array(np.dot(np.asmatrix(x_test), optimal_w) + optimal_b).T[0], label = "I
sns.kdeplot(Y_pred, label = "Sklearn SGD")
plt.title("Distribution of prediction $\hat{y}_i$")
plt.xlabel("predicted values")
plt.ylabel("Density")
plt.show()
```



observations 1) The mean squared error(mse) is quite high means the regression line does not fit the data properly. i.e. average squared difference between the actual target value and predicted target value is high. lower value is better.

2) After looking at the error graph we can say +ve side of the graph, error is more.

Conclusions

- While comparing scikit-learn implemented linear regression and explicitly implemented linear regression using optimization algorithm(sgd) in python we see there are not much differences between both of them.
- Both of the model are not perfect but okay.

Loading [MathJax]/extensions/Safe.js