Linear Regression using Sklearn SGD and Self Implemented SGD

In [1]:

```
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 [2]:

```
from sklearn.datasets import load_boston
```

In [3]:

```
boston=load_boston()
```

In [4]:

```
boston.data.shape
```

Out[4]:

```
(506, 13)
```

In [5]:

```
boston.feature_names
```

Out[5]:

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

In [6]:

```
boston.target
```

Out[6]:

```
array([24. , 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9, 15. ,
       18.9, 21.7, 20.4, 18.2, 19.9, 23.1, 17.5, 20.2, 18.2, 13.6, 19.6,
       15.2, 14.5, 15.6, 13.9, 16.6, 14.8, 18.4, 21. , 12.7, 14.5, 13.2,
       13.1, 13.5, 18.9, 20. , 21. , 24.7, 30.8, 34.9, 26.6, 25.3, 24.7,
       21.2, 19.3, 20. , 16.6, 14.4, 19.4, 19.7, 20.5, 25. , 23.4, 18.9,
       35.4, 24.7, 31.6, 23.3, 19.6, 18.7, 16. , 22.2, 25. , 33. , 23.5,
       19.4, 22. , 17.4, 20.9, 24.2, 21.7, 22.8, 23.4, 24.1, 21.4, 20. ,
       20.8, 21.2, 20.3, 28. , 23.9, 24.8, 22.9, 23.9, 26.6, 22.5, 22.2,
       23.6, 28.7, 22.6, 22. , 22.9, 25. , 20.6, 28.4, 21.4, 38.7, 43.8,
       33.2, 27.5, 26.5, 18.6, 19.3, 20.1, 19.5, 19.5, 20.4, 19.8, 19.4,
       21.7, 22.8, 18.8, 18.7, 18.5, 18.3, 21.2, 19.2, 20.4, 19.3, 22. ,
       20.3, 20.5, 17.3, 18.8, 21.4, 15.7, 16.2, 18. , 14.3, 19.2, 19.6,
       23. , 18.4, 15.6, 18.1, 17.4, 17.1, 13.3, 17.8, 14. , 14.4, 13.4,
       15.6, 11.8, 13.8, 15.6, 14.6, 17.8, 15.4, 21.5, 19.6, 15.3, 19.4,
```

```
       17. , 15.6, 13.1, 41.3, 24.3, 23.3, 27. , 50. , 50. , 50. , 22.7,
       25. , 50. , 23.8, 23.8, 22.3, 17.4, 19.1, 23.1, 23.6, 22.6, 29.4,
       23.2, 24.6, 29.9, 37.2, 39.8, 36.2, 37.9, 32.5, 26.4, 29.6, 50. ,
       32. , 29.8, 34.9, 37. , 30.5, 36.4, 31.1, 29.1, 50. , 33.3, 30.3,
       34.6, 34.9, 32.9, 24.1, 42.3, 48.5, 50. , 22.6, 24.4, 22.5, 24.4,
       20. , 21.7, 19.3, 22.4, 28.1, 23.7, 25. , 23.3, 28.7, 21.5, 23. ,
       26.7, 21.7, 27.5, 30.1, 44.8, 50. , 37.6, 31.6, 46.7, 31.5, 24.3,
       31.7, 41.7, 48.3, 29. , 24. , 25.1, 31.5, 23.7, 23.3, 22. , 20.1,
       22.2, 23.7, 17.6, 18.5, 24.3, 20.5, 24.5, 26.2, 24.4, 24.8, 29.6,
       42.8, 21.9, 20.9, 44. , 50. , 36. , 30.1, 33.8, 43.1, 48.8, 31. ,
       36.5, 22.8, 30.7, 50. , 43.5, 20.7, 21.1, 25.2, 24.4, 35.2, 32.4,
       32. , 33.2, 33.1, 29.1, 35.1, 45.4, 35.4, 46. , 50. , 32.2, 22. ,
       20.1, 23.2, 22.3, 24.8, 28.5, 37.3, 27.9, 23.9, 21.7, 28.6, 27.1,
       20.3, 22.5, 29. , 24.8, 22. , 26.4, 33.1, 36.1, 28.4, 33.4, 28.2,
       22.8, 20.3, 16.1, 22.1, 19.4, 21.6, 23.8, 16.2, 17.8, 19.8, 23.1,
       21. , 23.8, 23.1, 20.4, 18.5, 25. , 24.6, 23. , 22.2, 19.3, 22.6,
       19.8, 17.1, 19.4, 22.2, 20.7, 21.1, 19.5, 18.5, 20.6, 19. , 18.7,
       32.7, 16.5, 23.9, 31.2, 17.5, 17.2, 23.1, 24.5, 26.6, 22.9, 24.1,
       18.6, 30.1, 18.2, 20.6, 17.8, 21.7, 22.7, 22.6, 25. , 19.9, 20.8,
       16.8, 21.9, 27.5, 21.9, 23.1, 50. , 50. , 50. , 50. , 50. , 13.8,
       13.8, 15. , 13.9, 13.3, 13.1, 10.2, 10.4, 10.9, 11.3, 12.3,  8.8,
        7.2, 10.5,  7.4, 10.2, 11.5, 15.1, 23.2,  9.7, 13.8, 12.7, 13.1,
       12.5,  8.5,  5. ,  6.3,  5.6,  7.2, 12.1,  8.3,  8.5,  5. , 11.9,
       27.9, 17.2, 27.5, 15. , 17.2, 17.9, 16.3,  7. ,  7.2,  7.5, 10.4,
        8.8,  8.4, 16.7, 14.2, 20.8, 13.4, 11.7,  8.3, 10.2, 10.9, 11. ,
        9.5, 14.5, 14.1, 16.1, 14.3, 11.7, 13.4,  9.6,  8.7,  8.4, 12.8,
       10.5, 17.1, 18.4, 15.4, 10.8, 11.8, 14.9, 12.6, 14.1, 13. , 13.4,
       15.2, 16.1, 17.8, 14.9, 14.1, 12.7, 13.5, 14.9, 20. , 16.4, 17.7,
       19.5, 20.2, 21.4, 19.9, 19. , 19.1, 19.1, 20.1, 19.9, 19.6, 23.2,
       29.8, 13.8, 13.3, 16.7, 12. , 14.6, 21.4, 23. , 23.7, 25. , 21.8,
       20.6, 21.2, 19.1, 20.6, 15.2,  7. ,  8.1, 13.6, 20.1, 21.8, 24.5,
       23.1, 19.7, 18.3, 21.2, 17.5, 16.8, 22.4, 20.6, 23.9, 22. , 11.9])
```

In [7]:

```
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 University.
```

```
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 regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of C
ollinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the T
enth International Conference of Machine Learning, 236-243, University of Massachusetts, Amherst.
Morgan Kaufmann.
```

In [8]:

```
data = boston.data
bs_df = pd.DataFrame(data)
X = bs_df
Y = boston.target
```

In [9]:

```
X.head()
```

Out[9]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
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 [10]:

```
# standardizing the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_data = scaler.fit_transform(X)
```

SGD for Linear Regression using sklearn

In [ ]:

```
# https://machinelearningmastery.com/implement-linear-regression-stochastic-gradient-descent-scrat
ch-python/
# https://codebasicshub.com/tutorial/machine-learning/logistic-regression-multiclass-
classification
```

In [11]:

```
# http://scikitlearn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html

from sklearn.linear_model import SGDRegressor

clf = SGDRegressor()
clf.fit(X_data,Y)

Y_pred = clf.predict(X_data)
```

In [12]:

```
# Optimal Weights and Intercept values of sklearn SGD

from numpy import c_

print('Weights for sklearn SGD:', c_[clf.coef_])

print('Intercept for sklearn SGD:',clf.intercept_)
```

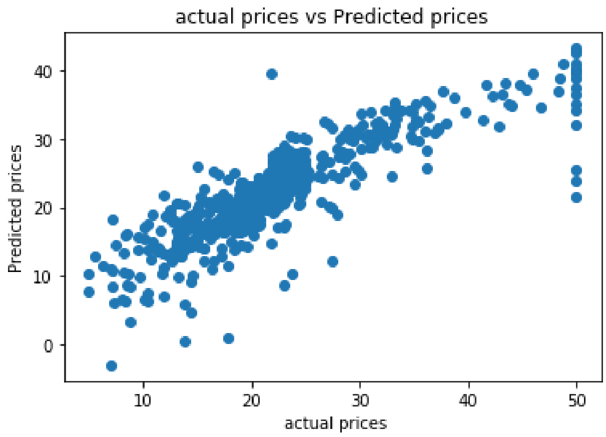
```
Weights for sklearn SGD: [[-0.63805546]
 [ 0.44077066]
 [-0.36943486]
 [ 0.81936679]
 [-0.94265706]
 [ 3.16992285]
 [ 0.016051  ]
 [-2.00287409]
 [ 0.87760778]
 [-0.22887733]
 [-1.85265255]
 [ 0.84090416]
 [-3.38960059]]
Intercept for sklearn SGD: [22.37340034]
```

In [13]:

```
# Graph of Predicted price and actual price

import matplotlib.pyplot as plt

plt.scatter(Y, Y_pred)
plt.xlabel(" actual prices")
plt.ylabel("Predicted prices")
plt.title("actual prices vs Predicted prices")
plt.show()
```



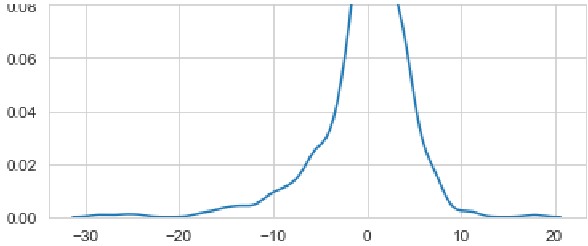
In [14]:

```
# Error Plot

import seaborn as sns
sns.set_style('whitegrid')
sns.kdeplot((Y_pred-Y))
plt.title("Error plot of sklearn SGD")
```

Out[14]:

```
Text(0.5, 1.0, 'Error plot of sklearn SGD')
```



In [15]:

```
# Computing MSE(Mean_square_error )

MSE_sklearnSGD = mean_squared_error(Y, Y_pred)
print("MSE of Sklearn SGD is: ",MSE_sklearnSGD)
```

```
MSE of Sklearn SGD is:  23.147801289396817
```

Implementation of SGDRegressor for Linear Regression

In [23]:

```
# Finding Optimal weights

def SGD_linreg(X,y,weight,learning_rate=0.01,iterations=10):

    L = len(y) #  length of the data set

    for i in range(iterations): # iteration
        sum_error = 0

        for i in range(L):
            batch_size = np.random.randint(0,L)   # random batch size for every iteration i.e k batc
h_size
            X_b = X[batch_size,:].reshape(1,X.shape[1])
            y_b = y[batch_size].reshape(1,1)
            pred = np.dot(X_b,weight)

            #-------- error ---------
            error = pred - y_b
            sum_error += error**2
            #--------- error ---------

            weight = weight -(2/L)*learning_rate*( X_b.T.dot((pred - y_b)))

        print('epoch={}, lr_rate={}, error={}'.format(i, learning_rate, sum_error/L))

    return weight

def predict_linreg(X_a,weight):
    y_pred = X_a.dot(weight)
    y_pred = y_pred.ravel()
    return y_pred
```

In [24]:

```
learning_rate =0.2
n_iter = 100

weight = np.random.randn(14,1)

X_a = np.c_[np.ones((len(X_data),1)),X_data]

optimal_weight = SGD_linreg(X_a,Y,weight,learning_rate,n_iter)
```

```
epoch=505, lr_rate=0.2, error=[[418.77881624]]
epoch=505, lr_rate=0.2, error=[[197.54523126]]
epoch=505, lr_rate=0.2, error=[[117.17276031]]
epoch=505, lr_rate=0.2, error=[[60.42060112]]
epoch=505, lr_rate=0.2, error=[[39.37910916]]
```

```
epoch=505, lr_rate=0.2, error=[[27.91519808]]
epoch=505, lr_rate=0.2, error=[[26.39269704]]
epoch=505, lr_rate=0.2, error=[[21.8518838]]
epoch=505, lr_rate=0.2, error=[[27.80573764]]
epoch=505, lr_rate=0.2, error=[[28.64260664]]
epoch=505, lr_rate=0.2, error=[[28.16330435]]
epoch=505, lr_rate=0.2, error=[[25.4786955]]
epoch=505, lr_rate=0.2, error=[[24.52458208]]
epoch=505, lr_rate=0.2, error=[[25.28780692]]
epoch=505, lr_rate=0.2, error=[[23.4399407]]
epoch=505, lr_rate=0.2, error=[[23.43791731]]
epoch=505, lr_rate=0.2, error=[[24.02883284]]
epoch=505, lr_rate=0.2, error=[[21.56813165]]
epoch=505, lr_rate=0.2, error=[[23.34310506]]
epoch=505, lr_rate=0.2, error=[[22.27370675]]
epoch=505, lr_rate=0.2, error=[[18.40672047]]
epoch=505, lr_rate=0.2, error=[[24.63157258]]
epoch=505, lr_rate=0.2, error=[[21.56945043]]
epoch=505, lr_rate=0.2, error=[[25.24162788]]
epoch=505, lr_rate=0.2, error=[[23.34730834]]
epoch=505, lr_rate=0.2, error=[[20.23769629]]
epoch=505, lr_rate=0.2, error=[[20.94969098]]
epoch=505, lr_rate=0.2, error=[[18.02718294]]
epoch=505, lr_rate=0.2, error=[[20.741179]]
epoch=505, lr_rate=0.2, error=[[24.31555265]]
epoch=505, lr_rate=0.2, error=[[22.34139448]]
epoch=505, lr_rate=0.2, error=[[19.58227055]]
epoch=505, lr_rate=0.2, error=[[20.39292475]]
epoch=505, lr_rate=0.2, error=[[22.16457613]]
epoch=505, lr_rate=0.2, error=[[17.71407287]]
epoch=505, lr_rate=0.2, error=[[20.7661114]]
epoch=505, lr_rate=0.2, error=[[22.21418838]]
epoch=505, lr_rate=0.2, error=[[23.30159313]]
epoch=505, lr_rate=0.2, error=[[20.39304878]]
epoch=505, lr_rate=0.2, error=[[17.81532109]]
epoch=505, lr_rate=0.2, error=[[23.51787491]]
epoch=505, lr_rate=0.2, error=[[22.55070539]]
epoch=505, lr_rate=0.2, error=[[21.96442709]]
epoch=505, lr_rate=0.2, error=[[25.17499828]]
epoch=505, lr_rate=0.2, error=[[28.30931837]]
epoch=505, lr_rate=0.2, error=[[23.91677296]]
epoch=505, lr_rate=0.2, error=[[22.04616666]]
epoch=505, lr_rate=0.2, error=[[23.68536525]]
epoch=505, lr_rate=0.2, error=[[16.66979803]]
epoch=505, lr_rate=0.2, error=[[22.62507116]]
epoch=505, lr_rate=0.2, error=[[27.4381154]]
epoch=505, lr_rate=0.2, error=[[22.4976803]]
epoch=505, lr_rate=0.2, error=[[20.57386142]]
epoch=505, lr_rate=0.2, error=[[17.02060383]]
epoch=505, lr_rate=0.2, error=[[21.56302498]]
epoch=505, lr_rate=0.2, error=[[18.62220404]]
epoch=505, lr_rate=0.2, error=[[23.49589275]]
epoch=505, lr_rate=0.2, error=[[18.18895885]]
epoch=505, lr_rate=0.2, error=[[23.42188037]]
epoch=505, lr_rate=0.2, error=[[22.45652327]]
epoch=505, lr_rate=0.2, error=[[21.6454935]]
epoch=505, lr_rate=0.2, error=[[19.35786793]]
epoch=505, lr_rate=0.2, error=[[19.1895708]]
epoch=505, lr_rate=0.2, error=[[23.38522761]]
epoch=505, lr_rate=0.2, error=[[25.92216954]]
epoch=505, lr_rate=0.2, error=[[21.81790293]]
epoch=505, lr_rate=0.2, error=[[19.67420288]]
epoch=505, lr_rate=0.2, error=[[19.73173375]]
epoch=505, lr_rate=0.2, error=[[27.87976131]]
epoch=505, lr_rate=0.2, error=[[24.10166972]]
epoch=505, lr_rate=0.2, error=[[20.08995679]]
epoch=505, lr_rate=0.2, error=[[21.43780506]]
epoch=505, lr_rate=0.2, error=[[23.80495704]]
epoch=505, lr_rate=0.2, error=[[20.80619434]]
epoch=505, lr_rate=0.2, error=[[23.42962118]]
epoch=505, lr_rate=0.2, error=[[23.33086575]]
epoch=505, lr_rate=0.2, error=[[23.03351796]]
epoch=505, lr_rate=0.2, error=[[23.56579575]]
epoch=505, lr_rate=0.2, error=[[16.50283028]]
epoch=505, lr_rate=0.2, error=[[23.58742189]]
epoch=505, lr_rate=0.2, error=[[27.05120537]]
epoch=505, lr_rate=0.2, error=[[20.65714438]]
```

```
epoch=505, lr_rate=0.2, error=[[24.20510836]]
epoch=505, lr_rate=0.2, error=[[16.83929307]]
epoch=505, lr_rate=0.2, error=[[25.50961432]]
epoch=505, lr_rate=0.2, error=[[21.64204317]]
epoch=505, lr_rate=0.2, error=[[21.15348115]]
epoch=505, lr_rate=0.2, error=[[23.66579115]]
epoch=505, lr_rate=0.2, error=[[21.50691454]]
epoch=505, lr_rate=0.2, error=[[21.65047923]]
epoch=505, lr_rate=0.2, error=[[19.98628126]]
epoch=505, lr_rate=0.2, error=[[16.05351668]]
epoch=505, lr_rate=0.2, error=[[25.94918173]]
epoch=505, lr_rate=0.2, error=[[24.54767136]]
epoch=505, lr_rate=0.2, error=[[23.14613699]]
epoch=505, lr_rate=0.2, error=[[24.82957857]]
epoch=505, lr_rate=0.2, error=[[20.85461265]]
epoch=505, lr_rate=0.2, error=[[22.84992584]]
epoch=505, lr_rate=0.2, error=[[19.07650034]]
epoch=505, lr_rate=0.2, error=[[20.12975691]]
```

In [25]:

```
# Optimal Weights and Intercept of implemented SGD

print('Optimal Weights of implemented SGD:', optimal_weight[1:])


print('Intercept of implemented SGD:{}'.format(optimal_weight[0][0]))
```

```
Optimal Weights of implemented SGD: [[-0.90683692]
 [ 1.0660151 ]
 [-0.04084335]
 [ 0.51702249]
 [-2.08288434]
 [ 2.86647946]
 [ 0.00971415]
 [-2.97667618]
 [ 2.55310236]
 [-1.92373307]
 [-2.03236471]
 [ 0.94851858]
 [-3.61733317]]
Intercept of implemented SGD:22.51493653332199
```

In [27]:

```
y_predicted = predict_linreg(X_a,optimal_weight)
```

In [28]:

```
# Graph of Predicted price and actual price, implemented SGD

plt.scatter(Y, y_predicted)
plt.xlabel(" actual prices")
plt.ylabel("Predicted prices")
plt.title("actual prices vs Predicted prices")
plt.show()
```

actual prices

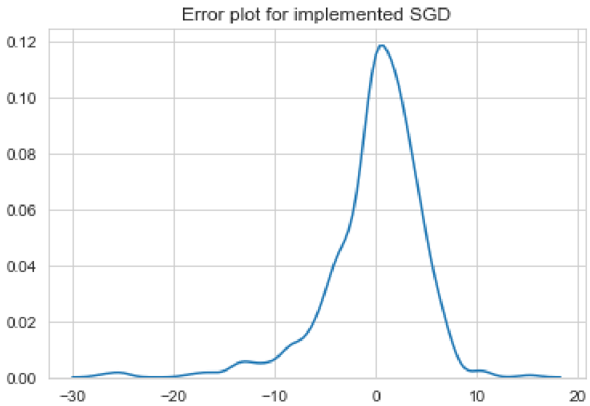
In [30]:

```
# plot error

sns.set_style('whitegrid')
sns.kdeplot((y_predicted-Y))
plt.title("Error plot for implemented SGD")
```

Out[30]:

```
Text(0.5, 1.0, 'Error plot for implemented SGD')
```



In [31]:

```
# Computing MSE (Mean_square_error) of implemented SGD

print("Mean Squared Error using the predicted Y and optimal weights :",np.mean((Y-y_predicted)**2)
)
```

```
Mean Squared Error using the predicted Y and optimal weights : 22.029436471453277
```

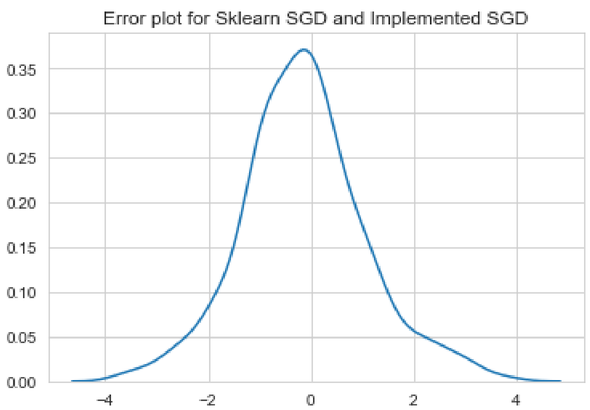
Comparing Sklearn SGD and Implemented SGD

In [33]:

```
sklearn_pred = Y_pred
impl_pred = y_predicted
sns.set_style('whitegrid')
sns.kdeplot((sklearn_pred-impl_pred))
plt.title("Error plot for Sklearn SGD and Implemented SGD")
```

Out[33]:

```
Text(0.5, 1.0, 'Error plot for Sklearn SGD and Implemented SGD')
```



Conclusion

In [34]:

```
# Getting optimal weight i.e (coef) for Self implemented SGD and sklearn SGD

print("Sklearn SGD optimal Weight", c_[clf.coef_])

print("\n Self implemented SGD optimal Weight", optimal_weight[1:])
```

```
Sklearn SGD optimal Weight [[-0.63805546]
 [ 0.44077066]
 [-0.36943486]
 [ 0.81936679]
 [-0.94265706]
 [ 3.16992285]
 [ 0.016051  ]
 [-2.00287409]
 [ 0.87760778]
 [-0.22887733]
 [-1.85265255]
 [ 0.84090416]
 [-3.38960059]]

 Self implemented SGD optimal Weight [[-0.90683692]
 [ 1.0660151 ]
 [-0.04084335]
 [ 0.51702249]
 [-2.08288434]
 [ 2.86647946]
 [ 0.00971415]
 [-2.97667618]
 [ 2.55310236]
 [-1.92373307]
 [-2.03236471]
 [ 0.94851858]
 [-3.61733317]]
```

In [36]:

```
from prettytable import PrettyTable
x = PrettyTable()
x.field_names = ["Sklearn SGD optimal Weight", "implemented SGD optimal Weight"]
x.add_row(["-0.63805546", "-0.90683692"])
x.add_row(["0.44077066", "1.0660151" ])
x.add_row(["-0.36943486", "-0.04084335"])
x.add_row(["0.81936679", "0.51702249"])
x.add_row(["-0.94265706", "-2.08288434"])
x.add_row(["3.16992285", "2.86647946"])
x.add_row(["0.016051", "0.00971415"])
x.add_row(["-2.00287409",  "-2.97667618"])
x.add_row(["0.87760778", "2.55310236"])
x.add_row(["-0.22887733", "-1.92373307"])
x.add_row(["-1.85265255", "-2.03236471"])
x.add_row(["0.84090416", "0.94851858"])
x.add_row(["-3.38960059", "-3.61733317"])
print(x)
```

```
+----------------------------+--------------------------------+
| Sklearn SGD optimal Weight | implemented SGD optimal Weight |
+----------------------------+--------------------------------+
|        -0.63805546         |          -0.90683692           |
|         0.44077066         |           1.0660151            |
|        -0.36943486         |          -0.04084335           |
|         0.81936679         |           0.51702249           |
|        -0.94265706         |          -2.08288434           |
|         3.16992285         |           2.86647946           |
|          0.016051          |           0.00971415           |
|        -2.00287409         |          -2.97667618           |
|         0.87760778         |           2.55310236           |
|        -0.22887733         |          -1.92373307           |
|        -1.85265255         |          -2.03236471           |
|         0.84090416         |           0.94851858           |
```

```
|         0.84090416         |           0.94851858           |
|        -3.38960059         |          -3.61733317           |
+----------------------------+--------------------------------+
```

In [37]:

```
# optimal Intercept of sklearn SGD and implemented SGD

print(" optimal intercept of Sklearn SGD: ",clf.intercept_)

print(" optimal intercept implemented SGD: ",optimal_weight[0][0])
```

```
 optimal intercept of Sklearn SGD:  [22.37340034]
 optimal intercept implemented SGD:  22.51493653332199
```

In [38]:

```
from prettytable import PrettyTable

x = PrettyTable()

x.field_names = ["MSE of Sklearn SGD", "intercept implemented SGD"]

x.add_row(["22.37340034", "22.51493653332199"])

print(x)
```

```
+--------------------------+--------------------------+
| intercept of Sklearn SGD | intercept implemented SGD |
+--------------------------+--------------------------+
|       22.37340034        |     22.51493653332199     |
+--------------------------+--------------------------+
```

In [39]:

```
from prettytable import PrettyTable

x = PrettyTable()

x.field_names = ["SGD","MSE"]

x.add_row(["Sklearn",23.147801289396817])
x.add_row(["implemented",22.029436471453277])

print(x)
```

```
+-------------+--------------------+
|     SGD     |        MSE         |
+-------------+--------------------+
|   Sklearn   | 23.147801289396817 |
| implemented | 22.029436471453277 |
+-------------+--------------------+
```

1. From Table we can observe that Intercept for both Sklearn SGD and implemented SGD is almost same value.

2. From Table we can conclude that MSE is nearly same

In [ ]: