In [1]:
```
# Importing all the required packages
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
```

Reading the dataset

In [2]:
```
X = load_boston().data
Y = load_boston().target
```

In [3]:
```
#Description of dataset
print(load_boston().DESCR)
```

```
Boston House Prices dataset
===========================

Notes
------
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 s
q.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 o
therwise)
        - 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 to
wn
        - 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.
http://archive.ics.uci.edu/ml/datasets/Housing (http://archive.ics.uci.edu/ml/d
atasets/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.
```

```
**References**

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Da
ta and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In P
roceedings on the Tenth International Conference of Machine Learning, 236-243,
 University of Massachusetts, Amherst. Morgan Kaufmann.
   - many more! (see http://archive.ics.uci.edu/ml/datasets/Housing) (http://ar
chive.ics.uci.edu/ml/datasets/Housing))
```

Preprocessing the dataset

In [4]:

```
# As the pre-loaded dataset is a numpy array, we are converting it to datadreame
# https://stackoverflow.com/questions/20763012/creating-a-pandas-dataframe-from-
bos_df = pd.DataFrame(data=load_boston().data, columns=load_boston().feature_nam
print("The shape of data frame is", bos_df.shape)
print("\nThe top 5 rows of the  dataframe are")
bos_df.head()
```

```
The shape of data frame is (506, 13)

The top 5 rows of the  dataframe are
```

Out[4]:

	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 [5]:

```
# Inserting the target variable in the dataframe
bos_df['price'] = load_boston().target
bos_df.head()
```

Out[5]:

	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 [6]:
```
bos_df.columns
```

Out[6]:
```
Index(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX',
       'PTRATIO', 'B', 'LSTAT', 'price'],
      dtype='object')
```

In [7]:
```
# Checking if any null value is present in dataset
bos_df.isnull().sum()
```

Out[7]:
```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
price      0
dtype: int64
```

Splitting the dataset into train and test

In [8]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, random_
```

In [9]:
```
print("X_train shape is", X_train.shape)
print("y_train shape is", y_train.shape)
print("X_test shape is", X_test.shape)
print("y_test shape is", y_test.shape)
```

```
X_train shape is (339, 13)
y_train shape is (339,)
X_test shape is (167, 13)
y_test shape is (167,)
```

Standardizing the dataset

In [10]:
```
from sklearn.preprocessing import StandardScaler
S = StandardScaler()
X_train = S.fit_transform(X_train)
X_test = S.transform(X_test)
```

References used to implement SGD on Linear regression

1. https://machinelearningmastery.com/implement-linear-regression-stochastic-gradient-descent-scratch-python/ (https://machinelearningmastery.com/implement-linear-regression-stochastic-gradient-descent-scratch-python/)
2. https://towardsdatascience.com/step-by-step-tutorial-on-linear-regression-with-stochastic-gradient-descent-1d35b088a843 (https://towardsdatascience.com/step-by-step-tutorial-on-linear-regression-with-stochastic-gradient-descent-1d35b088a843)

Implementing SGD from scratch

In [11]:
```
def pred(X,w,b):

    y_hat = []
    for i in range(0,len(X)):
        y = np.dot(X[i],w)+b
        #https://www.geeksforgeeks.org/numpy-asscalar-in-python
        y = np.asscalar(y)
        y_hat.append(y)
    y_hat = np.array(y_hat)

    return y_hat
```

In [12]:
```
def SGD(X, y, l_rate, epochs, l_rate_var):

    w=np.random.randn(X.shape[1],1)
    b=np.random.randn(1,1)

    n=X.shape[0]
    for epoch in range(1,epochs+1):
        sum_error=0

        for i in range(n):
            batch=np.random.randint(0,n)
            x_batch=X[batch,:].reshape(1,X.shape[1])
            y_batch=y[batch].reshape(1,1)

            y_pred=np.dot(x_batch,w)+b
            error=y_pred-y_batch

            sum_error += error**2
            dw=x_batch.T.dot((y_pred-y_batch))# Arrived logically by definition
            db=(y_pred-y_batch)

            w=w-(2/n)*l_rate*(dw)
            b=b-(2/n)*l_rate*(db)
        sum_error=sum_error/n
        print("Epoch :{0}  Total_error:{1}  lr_rate:{2}".format(epoch, np.round(

        if l_rate_var == 'constant':
            pass
        else :
            l_rate = l_rate/2

    return w,b
```

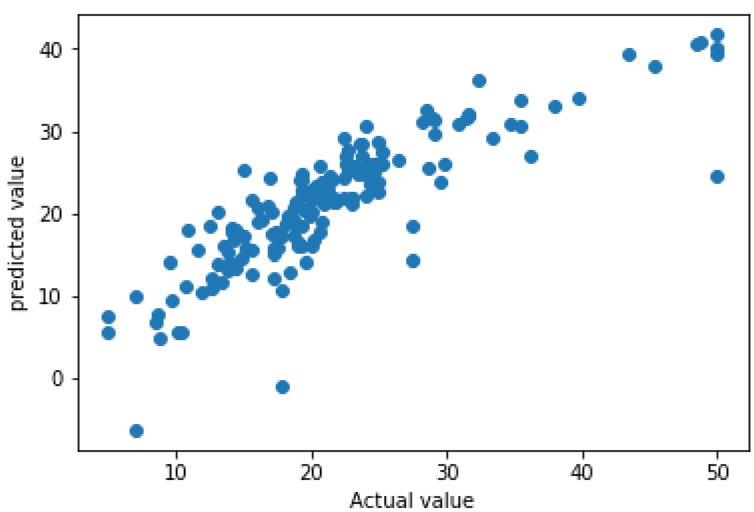
Executing Sgd with constant learning rate for all epochs

In [13]:
```
w_best,b_best = SGD(X_train,y_train,l_rate=0.01, epochs=500, l_rate_var = 'consta
Best_weights = pd.DataFrame(w_best,columns=['SGD_constant_l_rate'])
```

```
Epoch :96  Total_error:[[36.83]]  lr_rate:0.01
Epoch :97  Total_error:[[41.87]]  lr_rate:0.01
Epoch :98  Total_error:[[32.98]]  lr_rate:0.01
Epoch :99  Total_error:[[43.62]]  lr_rate:0.01
Epoch :100  Total_error:[[29.15]]  lr_rate:0.01
Epoch :101  Total_error:[[42.62]]  lr_rate:0.01
Epoch :102  Total_error:[[32.85]]  lr_rate:0.01
Epoch :103  Total_error:[[35.53]]  lr_rate:0.01
Epoch :104  Total_error:[[32.04]]  lr_rate:0.01
Epoch :105  Total_error:[[26.89]]  lr_rate:0.01
Epoch :106  Total_error:[[36.21]]  lr_rate:0.01
Epoch :107  Total_error:[[34.43]]  lr_rate:0.01
Epoch :108  Total_error:[[27.79]]  lr_rate:0.01
Epoch :109  Total_error:[[32.18]]  lr_rate:0.01
Epoch :110  Total_error:[[32.78]]  lr_rate:0.01

Epoch :111  Total_error:[[39.03]]  lr_rate:0.01
Epoch :112  Total_error:[[34.41]]  lr_rate:0.01
Epoch :113  Total_error:[[32.34]]  lr_rate:0.01
Epoch :114  Total_error:[[27.77]]  lr_rate:0.01
```

In [14]:
```
# Plotting line plot for actual vs model predicted
# https://stackoverflow.com/questions/31069191/simple-line-plots-using-seaborn
import seaborn as sns
y_hat_1 = pred(X_test, w_best, b_best)
plt.scatter(x=y_test,y=y_hat_1)
plt.xlabel('Actual value')
plt.ylabel('predicted value')
ms_er1 = mean_squared_error(y_test,y_hat_1)
print("Mean squared error",ms_er1)
```

```
Mean squared error 21.186219541861036
```



Observations

1. From above we observe that loss is gradually reducing if learning rate is kept constant

Executing SGD with different learning rate for all epochs

In [15]:
```
w_best,b_best=SGD(X_train,y_train,l_rate=1, epochs=500, l_rate_var = 'not consta
Best_weights['SGD_varied_l_rate'] = pd.DataFrame(w_best,columns=['SGD_varied_l_r
```

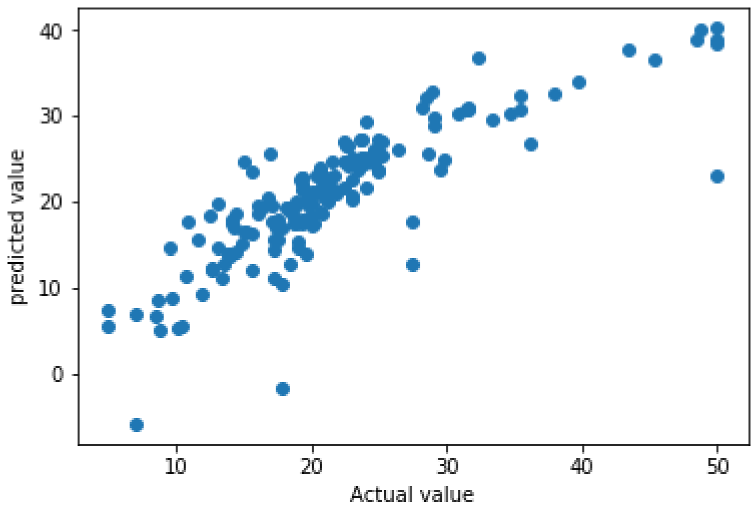
```
Epoch :105  Total_error:[[18.89]]  lr_rate:4.930380657631324e-32
Epoch :106  Total_error:[[28.16]]  lr_rate:2.465190328815662e-32

Epoch :107  Total_error:[[23.7]]  lr_rate:1.232595164407831e-32
Epoch :108  Total_error:[[22.73]]  lr_rate:6.162975822039155e-33
Epoch :109  Total_error:[[22.93]]  lr_rate:3.0814879110195774e-33
Epoch :110  Total_error:[[22.31]]  lr_rate:1.5407439555097887e-33
Epoch :111  Total_error:[[28.87]]  lr_rate:7.703719777548943e-34
Epoch :112  Total_error:[[25.11]]  lr_rate:3.851859888774472e-34
Epoch :113  Total_error:[[28.09]]  lr_rate:1.925929944387236e-34
Epoch :114  Total_error:[[28.15]]  lr_rate:9.62964972193618e-35
Epoch :115  Total_error:[[29.96]]  lr_rate:4.81482486096809e-35
Epoch :116  Total_error:[[18.43]]  lr_rate:2.407412430484045e-35
Epoch :117  Total_error:[[25.61]]  lr_rate:1.2037062152420224e-35
Epoch :118  Total_error:[[21.77]]  lr_rate:6.018531076210112e-36
Epoch :119  Total_error:[[29.13]]  lr_rate:3.009265538105056e-36
Epoch :120  Total_error:[[25.24]]  lr_rate:1.504632769052528e-36
Epoch :121  Total_error:[[27.63]]  lr_rate:7.52316384526264e-37
Epoch :122  Total_error:[[27.18]]  lr_rate:3.76158192263132e-37
Epoch :123  Total_error:[[20.97]]  lr_rate:1.88079096131566e-37
```

Observation

1. We notice that there is a drastic change from 2nd epoch itself compared to the first approach i.e with constant learning rate

In [16]:
```
# Plotting line plot for actual vs model predicted
# https://stackoverflow.com/questions/31069191/simple-line-plots-using-seaborn
import seaborn as sns
y_hat_1 = pred(X_test, w_best, b_best)
plt.scatter(x=y_test,y=y_hat_1)
plt.xlabel('Actual value')
plt.ylabel('predicted value')
ms_er2 = mean_squared_error(y_test,y_hat_1)
print("Mean squared error",ms_er2)
```

```
Mean squared error 22.060186907739414
```



Executing sklearn's SGDRegressor

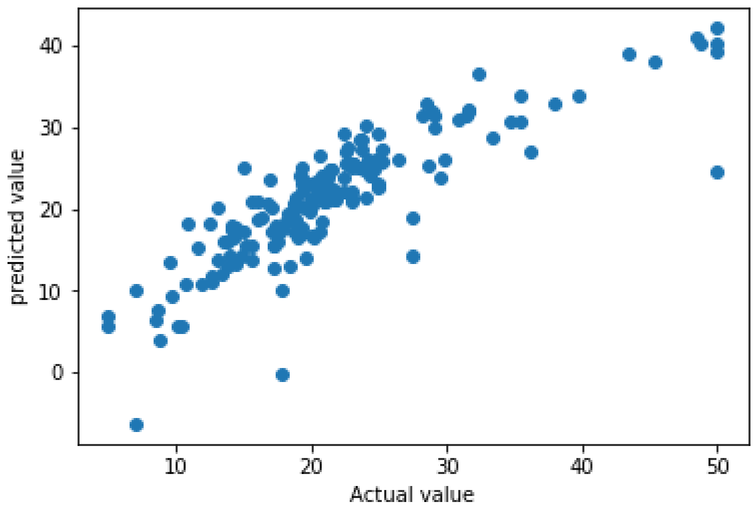
In [17]:
```
from sklearn.linear_model import SGDRegressor
SGDR = SGDRegressor(loss='squared_loss', alpha=0.01, max_iter=500)
SGDR.fit(X_train,y_train)
y_hat = SGDR.predict(X_test)
Best_weights['SGDRegressor_scikit'] = SGDR.coef_
```

In [18]:
```
print(mean_squared_error(y_test, SGDR.predict(X_test)))
```

```
20.85601918948433
```

In [19]:
```
# Plotting line plot for actual vs model predicted
# https://stackoverflow.com/questions/31069191/simple-line-plots-using-seaborn
import seaborn as sns
y_hat_1 = SGDR.predict(X_test)
plt.scatter(x=y_test,y=y_hat_1)
plt.xlabel('Actual value')
plt.ylabel('predicted value')
ms_er3 = mean_squared_error(y_test,y_hat_1)
print("Mean squared error",ms_er3)
```

```
Mean squared error 20.85601918948433
```



In [20]:
```
Best_weights
```

Out[20]:

	SGD_constant_l_rate	SGD_varied_l_rate	SGDRegressor_scikit
0	-0.840674	-0.795003	-0.942057
1	0.712502	0.655565	0.804730
2	-0.009761	-0.590807	0.314651
3	0.834394	1.314008	0.884120
4	-1.189070	-0.159156	-1.786730
5	2.872937	3.100020	2.825662
6	-0.429867	-0.575831	-0.362631
7	-2.711371	-2.332190	-2.903074
8	1.397033	1.688015	1.780323
9	-0.851373	-1.718437	-1.152708
10	-1.914551	-1.116304	-2.048570
11	1.040986	0.919219	1.047731
12	-4.049084	-3.715732	-3.889523

In [21]:
```
print("Mean score for SGD_constant_l_rate",ms_er1)
print("Mean score for SGD_varied_l_rate",ms_er2)
print("Mean score for SGDRegressor_scikit",ms_er3)
```

```
Mean score for SGD_constant_l_rate 21.186219541861036
Mean score for SGD_varied_l_rate 22.060186907739414
Mean score for SGDRegressor_scikit 20.85601918948433
```