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
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_validate
from sklearn.preprocessing import StandardScaler
```

In [258]:
```
X = load_boston().data
Y = load_boston().target
```

In [65]:
```
scaler = preprocessing.StandardScaler().fit(X)
X = scaler.transform(X)
```

In [84]:
```
clf = SGDRegressor()
clf.fit(X, Y)
print(mean_squared_error(Y,clf.predict(X)))

print(clf.get_params(deep=True))
```

```
22.959212518860685
{'alpha': 0.0001, 'average': False, 'early_stopping': False, 'epsilon': 0.
1, 'eta0': 0.01, 'fit_intercept': True, 'l1_ratio': 0.15, 'learning_rate':
'invscaling', 'loss': 'squared_loss', 'max_iter': None, 'n_iter': None, 'n_
iter_no_change': 5, 'penalty': 'l2', 'power_t': 0.25, 'random_state': None,
'shuffle': True, 'tol': None, 'validation_fraction': 0.1, 'verbose': 0, 'wa
rm_start': False}
```

In [107]:
```
print(clf.coef_)
```

```
[-0.67371808  0.61967151 -0.24103216  0.72957934 -0.89430692  3.18483195
 -0.18434904 -2.25869644  1.09746133 -0.612922   -1.79249909  0.8783723
 -3.26719759]
```

In [33]:
```
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

import random
import pandas as pd
import numpy as np
import os
import time
import math
import pickle


class LinRegr:
    """
        we have two main methods SGDProcess and SGDProcess1.
        SGDProcess first preforms linear regression and gets the weights. W
e will optimize these weights
        SGDProcess1 starts with weight vector all initialized to zero
    """
    def __init__(self):
        #data
        self.Xdata = []
        self.ydata = []
        self.y_hat_tst_opt = []
        #regressor
        self.linrgr = None

        #output beta estimates
        self.coeff = []
        #intercept
        self.intcept = 0.0
        #predictions
        self.pred = []
        #R squared
        self.rsquared = 0.0
        #mean squared error
        self.MSE = 0.0
        #sample points for SGD
        self.x_sgdt = []
        self.y_sgdt = []

        #intial weigts W0 and intercepts
        self.w_0 = []
        self.w_prev = []
        self.w_next = []
        self.w_opt = []
        self.partial_w = []

        self.intcpt_prev = 0.0
        self.intcpt_next = 0.0
        self.intcpt_opt = 0.0
```

```
        self.partial_intcpt = 0.0

        #no of iterations for SGD
        self.num_iter = 0
        self.num_of_pts = 0

        #learning rate
        self.learning_rate = 0


        self.x_sgdt_df = pd.DataFrame()
        self.y_sgdt_df = pd.DataFrame()
        self.term2= []


    #constructor
    def linearegrsn(self):
        self.linrgr = LinearRegression(fit_intercept=True,normalize=True)
        return self.linrgr

    #getter/setters
    #LINEAR REGRESSOR
    @property
    def linrgr(self):
        return self._linrgr

    @linrgr.setter
    def linrgr(self,new_linrgr):
        self._linrgr = new_linrgr

    @property
    def Xdata(self):
        return self._Xdata

    @Xdata.setter
    def Xdata(self,new_Xdata):
        self._Xdata = new_Xdata


    @property
    def ydata(self):
        return self._ydata
    @ydata.setter
    def ydata(self,new_ydata):
        self._ydata = new_ydata


    @property
    def x_train(self):
        return self._x_train

    @x_train.setter
    def x_train(self,new_x_train):
        self._x_train = new_x_train

    @property
    def x_test(self):
```

```
        return self._x_test

    @x_test.setter
    def x_test(self,new_x_test):
        self._x_test = new_x_test


    @property
    def y_train(self):
        return self._y_train

    @y_train.setter
    def y_train(self,new_y_train):
        self._y_train = new_y_train

    @property
    def y_test(self):
        return self._y_test

    @y_test.setter
    def y_test(self,new_y_test):
        self._y_test = new_y_test

    @property
    def y_hat_tst_opt(self):
        return self._y_hat_tst_opt

    @y_hat_tst_opt.setter
    def y_hat_tst_opt(self,new_y_hat_tst_opt):
        self._y_hat_tst_opt = new_y_hat_tst_opt


    @property
    def coeff(self):
        return self._coeff

    @coeff.setter
    def coeff(self,new_coef):
        self._coeff = new_coef

    @property
    def intcept(self):
        return self._intcept

    @intcept.setter
    def intcept(self,new_intcept):
        self._intcept = new_intcept

    @property
    def pred(self):
        return self._pred

    @pred.setter
    def pred(self,new_pred):
        self._pred = new_pred

    @property
```

```
    def rsquared(self):
        return self._rsquared

    @rsquared.setter
    def rsquared(self,new_rsquared):
        self._rsquared = new_rsquared

    @property
    def MSE(self):
        return self._MSE

    @MSE.setter
    def MSE(self,new_mse):
        self._MSE = new_mse

    @property
    def w_0(self):
        return self._w_0

    @w_0.setter
    def w_0(self,new_w_0):
        self._w_0 = new_w_0

    @property
    def w_prev(self):
        return self._w_prev

    @w_prev.setter
    def w_prev(self,new_w_prev):
        self._w_prev = new_w_prev

    @property
    def w_next(self):
        return self._w_next

    @w_next.setter
    def w_next(self,new_w_next):
        self._w_next = new_w_next

    @property
    def w_opt(self):
        return self._w_opt

    @w_opt.setter
    def w_opt(self,new_w_opt):
        self._w_opt = new_w_opt

    @property
    def partial_w(self):
        return self._partial_w

    @partial_w.setter
    def partial_w(self,new_partial_w):
        self._partial_w = new_partial_w

    @property
    def intcpt_prev(self):
```

```
        return self._intcpt_prev

    @intcpt_prev.setter
    def intcpt_prev(self,new_intcpt_prev):
        self._intcpt_prev = new_intcpt_prev

    @property
    def intcpt_next(self):
        return self._intcpt_next

    @intcpt_next.setter
    def intcpt_next(self,new_intcpt_next):
        self._intcpt_next = new_intcpt_next

    @property
    def intcpt_opt(self):
        return self._intcpt_opt

    @intcpt_opt.setter
    def intcpt_opt(self,new_intcpt_opt):
        self._intcpt_opt = new_intcpt_opt

    @property
    def partial_intcpt(self):
        return self._partial_intcpt

    @partial_intcpt.setter
    def partial_intcpt(self,new_partial_intcpt):
        self._partial_intcpt = new_partial_intcpt


    @property
    def num_iter(self):
        return self._num_iter

    @num_iter.setter
    def num_iter(self,new_num_iter):
        self._num_iter = new_num_iter

    @property
    def num_of_pts(self):
        return self._num_of_pts

    @num_of_pts.setter
    def num_of_pts(self,new_numpts):
        self._num_of_pts = new_numpts

    @property
    def learning_rate(self):
        return self._learning_rate

    @learning_rate.setter
    def learning_rate(self,new_learning_rate):
        self._learning_rate= new_learning_rate


    @property
```

```
    def x_sgdt_df(self):
        return self._x_sgdt_df

    @x_sgdt_df.setter
    def x_sgdt_df(self,new_x_sgdt_df):
        self._x_sgdt_df = new_x_sgdt_df

    @property
    def y_sgdt_df(self):
        return self._y_sgdt_df

    @y_sgdt_df.setter
    def y_sgdt_df(self,new_y_sgdt_df):
        self._y_sgdt_df = new_y_sgdt_df

    @property
    def x_sgdt(self):
        return self._x_sgdt

    @x_sgdt.setter
    def x_sgdt(self,new_x_sgdt):
        self._x_sgdt = new_x_sgdt

    @property
    def y_sgdt(self):
        return self._y_sgdt

    @y_sgdt.setter
    def y_sgdt(self,new_y_sgdt):
        self._y_sgdt = new_y_sgdt

    @property
    def term2(self):
        return self._term2

    @term2.setter
    def term2(self,new_term2):
        self._term2 = new_term2

     #load the boston dataset
    def load_data(self):
        self.Xdata = load_boston().data
        self.ydata = load_boston().target
        # split into x and y train and test
        self.x_train, self.x_test, self.y_train, self.y_test = train_test_s
plit(self.Xdata, self.ydata, test_size=0.33, random_state=5)
        #standardize the x train and test data
        scaler = StandardScaler().fit(self.x_train)
        self.x_train = scaler.transform(self.x_train)
        self.x_test = scaler.transform(self.x_test)
        #we need to get random points from the data set
        #hence the x and y data are joined together so that when we get ran
dom points we get x and also the
        #corresponding y values
        self.x_sgdt_df = pd.DataFrame(self.x_train)
        self.x_sgdt_df['price'] = self.y_train
```

```
    #fit data
    def lnrg_fitdata(self):
        self.linrgr = (self.linrgr).fit(self.x_train,self.y_train)
        return self.linrgr

    #predict data
    def lnrgr_predict(self):
        self.pred = (self.linrgr).predict(self.x_test)
        return (self.linrgr,self.pred)

    #r squared
    def lnrgr_rsq(self):
        self.rsquared = (self.linrgr).score(self.Xdata,self.ydata)
        return self.rsquared

    #beta estimates
    def getCoeff(self):
        self.coeff = (self.linrgr).coef_
        self.w_0 = (self.linrgr).coef_
        return self.coeff

    #get intercepts
    def getintercepts(self):
        self.intcept = (self.linrgr).intercept_
        return self.intcept

    #get random k points from the datset for SGD
    def generaterandomsample(self):
        tmp_df = (self.x_sgdt_df.sample(self.no_of_pts))
        self.x_sgdt = tmp_df.drop('price',axis=1).values
        self.y_sgdt = tmp_df['price'].values


    def pred1(self):

        y_pred=[]

        for i in range(self.x_test.shape[0]):
            y=np.asscalar(np.dot(self.w_opt,self.x_test[i])+self.intcpt_opt
)
            y_pred.append(y)
        np.array(y_pred)
        print(mean_squared_error(self.y_test,y_pred))
        return y_pred

    # in this process we start with all weights equal to zero and move towa
rds w-star
    #SGDProcess_1
    def SGDProcess_1(self,niter,learnrate,nrows):
        #load boston data
        self.Xdata = load_boston().data
        self.ydata = load_boston().target

        # split into train and test
        train_data, test_data, train_y, test_y=train_test_split(self.Xdata,
self.ydata, test_size=0.33, random_state=5)
```

```
        #standard scaler used to standardize data
        stdscaler=StandardScaler()
        train_data=stdscaler.fit_transform(np.array(train_data))
        test_data=stdscaler.transform(np.array(test_data))

        # join X and corresponding Y values
        train_df = pd.DataFrame(data=train_data)
        nfeat = len(train_df.columns)
        train_df['Price'] = train_y

        tst_x_np = np.array(test_data)
        tst_y_np = np.array(test_y)

        #initialize the epoochs and learning rate
        self.num_iter = niter
        self.learning_rate = learnrate

        #initialize w_next and intercept next
        self.w_next = np.zeros(shape=(1,nfeat))
        self.intcpt_next = 0

        nolops = 1

        #start the run
        while(nolops <= self.num_iter):
            self.w_prev = self.w_next
            self.intcpt_prev = self.intcpt_next
            w_tmp = np.zeros(shape=(1,13))
            b_tmp=0

            #nrows is  the number of samples that we need to get from the d
ata
            #sample returns the x and y data
            trn_data = train_df.sample(nrows)
            #drop the price data and get only the xdata
            x_dt = np.array(trn_data.drop('Price',axis=1))
            #get only the y data corresponding to the X values
            y_dt = np.array(trn_data['Price'])


            for i in range(nrows):
                #predict the  y value
                y_pred = np.dot(self.w_prev,x_dt[i]) + self.intcpt_next
                # generate the weights and intercept
                w_tmp += x_dt[i] * (y_dt[i] - y_pred)
                b_tmp += (y_dt[i] - y_pred)

            w_tmp *= (-2/nrows)
            b_tmp *= (-2/nrows)

            #get the update value for the weights
            self.w_next = self.w_prev - (self.learning_rate * w_tmp)
            self.intcpt_next = self.intcpt_prev - (self.learning_rate * b_t
mp)

            self.learning_rate = self.learning_rate / pow(nolops,0.25)
            #increment the loop
```

```
            nolops += 1

        # using the w_start and intercept
        #use the test data to generate the y value
        y_pred=[]

        for i in range(len(tst_x_np)):
            y=np.asscalar(np.dot(self.w_next,tst_x_np[i])+self.intcpt_next)
            y_pred.append(y)
        #calculate the mean squared error and print it out
        print('Mean Squared Error',mean_squared_error(test_y,y_pred))


        return self.w_next, self.intcpt_next

    #in this follwing process we run linear regression and generate weights
for the given data
    #then we use the weights as the starting point and iterate till we get
 w_star
    #SGD process
    def SGDProcess(self,niter, npts,nmfeat):
        #initializations
        self.num_iter = niter
        self.no_of_pts = npts
        num_feat = nmfeat
        num_rows = npts
        self.w_prev = self.w_0
        self.intcpt_prev = self.intcept

        nolops=1
        w_diff = []

        #start the run
        while(nolops<=self.num_iter):
            self.w_next = np.zeros(shape=(1,num_feat))
            self.partial_w = np.zeros(shape=(1,num_feat))
            self.intcpt_next = 0.0
            #generate the random sample data points
            self.generaterandomsample()
            y_pred = np.zeros(num_rows)
            x=np.array(self.x_sgdt)
            y=np.array(self.y_sgdt)

            for i in range(num_rows):
                #predict the value
                y_pred=np.dot( self.w_prev,x[i])+self.intcpt_prev
                self.partial_w+=x[i] * (y[i] - y_pred)
                #print(type(self.partial_w),self.partial_w.shape, x.shape)
                self.intcpt_next+=(y[i]-y_pred)
            #print(num_rows)
            self.partial_w *=(-2/num_rows)
            self.intcpt_next*=(-2/num_rows)

            #updating the weight vector
            self.w_next=(self.w_prev-(self.learning_rate*self.partial_w))
            #print(type(self.w_next),self.w_next.shape)
            self.intcpt_next=(self.intcpt_prev-(self.learning_rate*self.int
```

```
cpt_next))

            #generate the MSE
            self.w_opt = self.w_next
            self.intcpt_opt = self.intcpt_next
            self.eval_yhat_opt()
            ret_mse = self.eval_MSE()
            """
            #if ret_mse[0] <= 30:
            if self.checkallval(self.w_next,num_feat):
                print('SOLUTION CONVERGED',t)
                self.w_opt = self.w_next
                self.intcpt_opt = self.intcpt_next
                self.learning_rate = r

                break
            else:
                self.w_prev = self.w_next
                self.intcpt_prev = self.intcpt_next
                r = r /10
                self.learning_rate = r
                #print('SOLUTION NOT CONVERGED \n',t,'****',r)

            """

            #transfer the new weights int old weights
            self.w_prev = self.w_next
            self.intcpt_prev = self.intcpt_next
            #update the learning rate
            self.learning_rate = self.learning_rate / 3
            #print('r',r,t)
            #increment the epoch
            nolops+=1
        #at the end of the loop the w_next and intercept_next are w_star an
d interpret_star
        self.w_opt = self.w_next
        self.intcpt_opt = self.intcpt_next


        return [self.w_next,  self.intcpt_next, self.learning_rate]




    def checkallval(self,wdiff,nofeat):
        j= 0
        k= 0
        diff = 0
        for i in range(0,len(wdiff)):
            prev = np.asarray(self.w_prev[0])
            nxt = np.asarray(self.w_next[0])
            #print(i,self.w_next.shape,self.w_prev.shape,len(wdiff))
            #print(type(prev[i]),prev.shape,'next', nxt[i])
            diff = (prev[i] - nxt[i])
            print('diff',diff)
            if  diff <= 0.0000001:
                #print("diff less than 0.00001\n")
```

```
                j+=1
            elif diff > 0.0000001:
                #print("diff greater than 0.00001\n")
                j-=1

        if j==nofeat:
            print('**************',j)
            return True
        else:
            return False
            #if wdiff[i] >0.0000001:
                #return False

    #generate y_hat using the w_star created by the sgd Process
    def eval_yhat_opt(self):

        num_rows = self.y_test.shape[0]
        self.y_hat_tst_opt = np.zeros(shape=(num_rows,1))
        for k in range(0,num_rows):
            #Y1 = W1*X11+w2*x12+w3*x13+w4*x14....Wn*Xn<num_feat>
            self.y_hat_tst_opt[k] = np.dot(self.w_opt,self.x_test[k])+self.
intcpt_opt
            #print(self.w_opt , self.x_test[k],self.y_hat_tst_opt[k] )
            #add the optimum intercept
            #self.y_hat_tst_opt[k] += self.intcpt_opt
            #print(self.y_hat_tst_opt[k] )
        return [self.y_hat_tst_opt]

    #generate Mean Squared Error
    def eval_MSE(self):
        num_rows = self.y_test.shape[0]
        for i in range(0,num_rows):
            self.MSE += (self.y_hat_tst_opt[i] - self.y_test[i]) ** 2
            #print(self.y_test[i],self.y_hat_tst_opt[i] )
        self.MSE = self.MSE / num_rows

        return [self.MSE]
```

In [7]:
```
#implementor code for Stocastic gradient descent USING DOT PRODUCT WITH PRE
CALCULATED WEIGHTS

#instantiate Linear Regression object and regressor
#object
linregr2 = LinRegr()
#regressor
lin_rgrsn_2 = linregr2.linearegrsn()
linregr2.load_data()

lin_rgrsn_2 = linregr2.lnrg_fitdata()
beta_est2 = linregr2.getCoeff()
intercepts2 = linregr2.getintercepts()


linregr2.learning_rate = 0.00001
return_list2 = linregr2.SGDProcess(100,10,linregr2.x_train.shape[1])

print(return_list2[0])
print(return_list2[1])
print(return_list2[2])
linregr2.eval_yhat_opt()
MSE = linregr2.eval_MSE()
print('#####MSE',MSE)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=Tr
ue)
[[-1.31194948  0.8618651  -0.16720579  0.18959156 -1.48657697  2.79128043
  -0.32737856 -2.77204607  2.97565149 -2.27277939 -2.13376876  1.05846715
  -3.3349703 ]]
[22.53714706]
1.9403252174826322e-53
#####MSE [array([28.70249481])]
```

In [52]:
```
#implementor code for Stocastic gradient descent USING DOT PRODUCT WITH PRE
CALCULATED WEIGHTS equal to Zero
linregr3 = LinRegr()
#regressor
lin_rgrsn_3 = linregr3.linearegrsn()
ret_list3 = linregr3.SGDProcess_1(100,0.194,10)
```

```
Mean Squared Error 45.694918136190324
```

In [58]:
```
import tabulate
res_tab = [['S No.', 'Weights','Model', 'MSE'],
         [1,'Zeros','SGDRegressor',22.959212],
         [2,'Pre-computed','MyModel', 28.702494],
         [3,'Zeros', 'MyModel',45.694918]]
```

In [59]:
```
print(tabulate.tabulate(res_tab, tablefmt='fancy_grid'))
```

S No.	Weights	Model	MSE
1	Zeros	SGDRegressor	22.959212
2	Pre-computed	MyModel	28.702494
3	Zeros	MyModel	45.694918