Assignment 6: Implement SGD for linear regression

Objective:

To Implement stochastic gradient descent on Bostan House Prices dataset for linear Regression

•Implement SGD and deploy on Bostan House Prices dataset.

•Comapare the Results with sklearn.linear_model.SGDRegressor.

About Linear Regression Technque

It is a very powerful technique and can be used to understand the factors that influence profitability.

The objective of a linear regression model is to find a relationship between one or more features(independent variables) and a continuous target variable(dependent variable). When there is only feature it is called Uni-variate Linear Regression and if there are multiple features, it is called Multiple Linear Regression.

In [1]:
```
# Import libraries necessary for this project
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
```

```python
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]:
```python
#https://towardsdatascience.com/linear-regression-using-python-b136c91b
f0a2
#https://www.kaggle.com/sukeshpabba/linear-regression-with-boston-housi
ng-data
#https://www.geeksforgeeks.org/ml-boston-housing-kaggle-challenge-with-
linear-regression/
#https://www.ritchieng.com/machine-learning-project-boston-home-prices/
```

The objective of a linear regression model is to find a relationship between one or more features(independent variables)and a continuous target variable(dependent variable).

How do we determine the best fit line? The line for which the the error between the predicted values and the observed values is minimum is called the best fit line or the regression line.

In [3]:
```python
#https://www.ritchieng.com/machine-learning-project-boston-home-prices/
```

In [4]:
```python
#Next, we will load the housing data from the scikit-learn library and
 understand it.
X = load_boston().data
Y = load_boston().target
```

In [5]:
```python
scaler = preprocessing.StandardScaler().fit(X)
X = scaler.transform(X)
```

In [6]:
```python
clf = SGDRegressor()
clf.fit(X, Y)
print(mean_squared_error(Y, clf.predict(X)))
```

```
22.781814039580578
```

In [7]:
```
from sklearn.datasets import load_boston
boston = load_boston()
```

In [31]:
```
print(boston.keys())
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename'])
```

In [ ]:
```
#data: contains the information for various houses
#target: prices of the house
#feature_names: names of the features
#DESCR: describes the dataset
```

In [8]:
```
print(boston.data.shape)
```

```
(506, 13)
```

In [9]:
```
print(boston.feature_names)
```

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

In [10]:
```
col= boston.feature_names
print(col)
```

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

The prices of the house indicated by the variable MEDV is our target variable and the remaining are the feature variables based on which we will predict the value of a house.

In [11]:
```
print(boston.target)
```

```
[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 [12]:
```
print(boston.DESCR)
```

```
.. _boston_dataset:

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

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Va
lue (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 2
5,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds ri
ver; 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 19
40
        - 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 blac
ks 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. 'Hedoni
c
prices and the demand for clean air', J. Environ. Economics & Managemen
t,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagn
ostics
...', 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 pape
rs that address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influe
ntial Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learni
ng. In Proceedings on the Tenth International Conference of Machine Lea
rning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
```

In [13]:

```python
#We will now load the data into a pandas dataframe using pd.DataFrame.
 We then print the first 5 rows of the data using head()
import pandas as pd
bost = pd.DataFrame(boston.data)
print(bost.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  1
5.3
1  0.02731   0.0  7.07  0.0  0.469  6.421  78.9  4.9671  2.0  242.0  1
7.8
```

```
2  0.02729   0.0  7.07  0.0  0.469  7.185  61.1  4.9671  2.0  242.0  1
7.8
3  0.03237   0.0  2.18  0.0  0.458  6.998  45.8  6.0622  3.0  222.0  1
8.7
4  0.06905   0.0  2.18  0.0  0.458  7.147  54.2  6.0622  3.0  222.0  1
8.7

       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 [14]:

```python
# Boston dataset with columns names
bost_col = pd.DataFrame(boston.data,columns = col)
print(bost_col.head())
```

```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX
\
0  0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900  1.0  296.0

1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0

2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0

3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0  222.0

4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0  222.0


   PTRATIO       B  LSTAT
0     15.3  396.90   4.98
1     17.8  396.90   9.14
2     17.8  392.83   4.03
3     18.7  394.63   2.94
4     18.7  396.90   5.33
```

We can see that the target value PRICE is missing from the data. We create a new column of target values and add it to the dataframe.

As our goal is to develop a model that has the capacity of predicting the value of houses, we will split the dataset into features and the target variable. And store them in features and prices variables, respectively The features ‘RM’, ‘LSTAT’ and ‘PTRATIO’, give us quantitative information about each datapoint. We will store them in features. The target variable, ‘PRICE’, will be the variable we seek to predict. We will store it in prices.

In [15]:
```
bost['PRICE'] = boston.target

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

In [38]:
```
#After loading the data, it’s a good practice to see if there are any m
issing values in the data.
#We count the number of missing values for each feature using isnull()
bost.isnull().sum()
```

Out[38]:
```
0        0
1        0
2        0
3        0
4        0
5        0
6        0
7        0
8        0
9        0
10       0
11       0
12       0
PRICE    0
dtype: int64
```

However, there are no missing values in this dataset as shown

Splitting the data into training and testing sets

Next, we split the data into training and testing sets. We train the model with 70% of the samples and test with the remaining 30%. We do this to assess the model's performance on unseen data. To split the data we use train_test_split function provided by scikit-learn library. We finally print the sizes of our training and test set to verify if the splitting has occurred properly.

In [16]:
```
# now splitting the data
from sklearn.model_selection  import train_test_split
import sklearn
X_train, X_test, Y_train, Y_test = sklearn.model_selection.train_test_s
plit(X, Y, test_size = 0.3, random_state = 5)
print(X_train.shape)
print(X_test.shape)
print(Y_train.shape)
print(Y_test.shape)
```

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

In [39]:
```
col= boston.feature_names
print(col)
```

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

Feature Observation

Data Science is the process of making some assumptions and hypothesis on the data, and testing them by performing some tasks. Initially we could make the following intuitive assumptions for each feature:

1- Houses with more rooms (higher 'RM' value) will worth more.Usually houses with more rooms are bigger and can fit more people, so it is reasonable that they cost more money. They are directly proportional variables.

2- Neighborhoods with more lower class workers (higher ‘LSTAT’ value) will worth less. If the percentage of lower working class people is higher, it is likely that they have low purchasing power and therefore, they houses will cost less. They are inversely proportional variables.

3- Neighborhoods with more students to teachers ratio (higher ‘PTRATIO’ value) will be worth less. If the percentage of students to teachers ratio people is higher, it is likely that in the neighborhood there are less schools, this could be because there is less tax income which could be because in that neighborhood people earn less money. If people earn less money it is likely that their houses are worth less. They are inversely proportional ariables.

In [18]:

```
# applying column standardization on train and test data

scaler = preprocessing.StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test=scaler.transform(X_test)

df_train=pd.DataFrame(X_train)
df_train['price']=Y_train
df_train.head()
```

Out[18]:

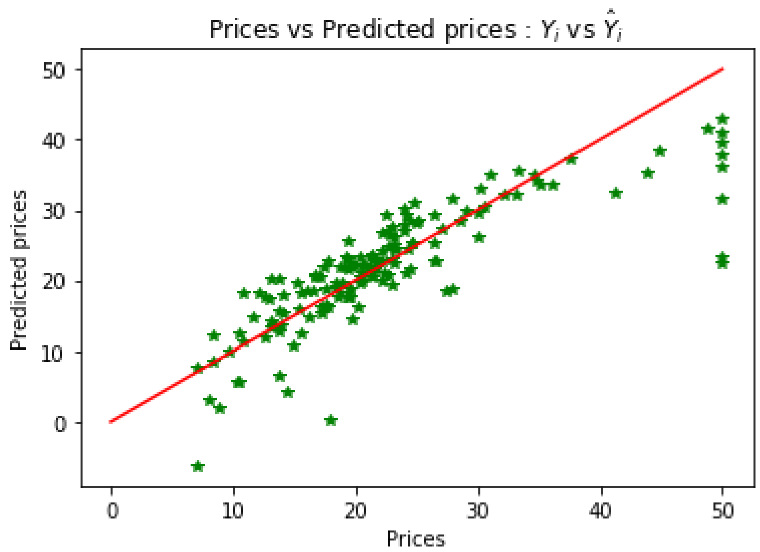
	0	1	2	3	4	5	6	7	
0	0.875509	-0.499618	1.069608	-0.251124	1.645428	0.233772	0.969882	-0.900522	1.65448
1	0.474665	-0.499618	1.069608	-0.251124	1.113435	-0.149715	0.383159	-0.926152	1.65448
2	0.273444	-0.499618	1.069608	-0.251124	-0.168580	0.653301	0.270733	-0.241993	1.65448
3	-0.417342	3.445319	-1.442682	-0.251124	-1.293614	1.372699	-1.591321	2.387078	-0.52791
4	-0.400634	-0.499618	2.504352	-0.251124	0.502952	-1.215116	0.896102	-0.982361	-0.64278

Training and testing the model

We use scikit-learn's LinearRegression to train our model on both the training and test sets.

In [19]:
```
# code source:https://medium.com/@haydar_ai/learning-data-science-day-9
-linear-regression-on-boston-housing-dataset-cd62a80775ef
from sklearn.linear_model import LinearRegression
lm = LinearRegression()
lm.fit(X_train, Y_train)
Y_pred = lm.predict(X_test)
error=abs(Y_test-Y_pred)
total_error = np.dot(error,error)
# Compute RMSE
rmse_lr= np.sqrt(total_error/len(error))
print('RMSE=',rmse_lr)
#plt.show()
plt.plot(Y_test, Y_pred,'g*')
plt.plot([0,50],[0,50], 'r-')
plt.title("Prices vs Predicted prices : $Y_i$ vs $\hat{Y}_i$")
plt.xlabel('Prices')
plt.ylabel('Predicted prices')
plt.show()
```

```
RMSE= 5.540490745781331
```



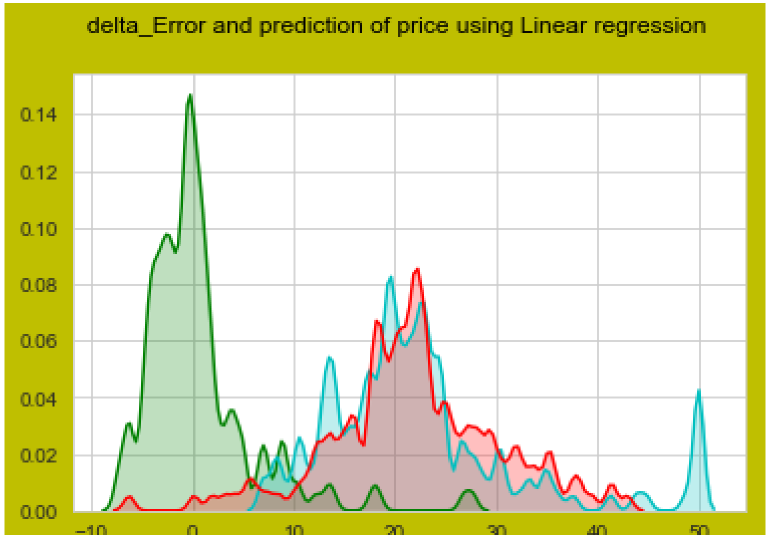
Delta_Error and Prediction of price using Linear regression

Observation

We start by creating a scatterplot matrix that will allow us to visualize the pair-wise relationships and correlations between the different features. It is also quite useful to have a quick overview of how the data is distributed and wheter it cointains or not outliers.

In [20]:
```
delta_y = Y_test - Y_pred
import seaborn as sns
fig3 = plt.figure( facecolor='y', edgecolor='k')
fig3.suptitle('delta_Error and prediction of price using Linear regress
ion', fontsize=12)

sns.set_style('whitegrid')
sns.kdeplot(np.array(delta_y),shade=True, color="g", bw=0.5)
sns.kdeplot(np.array(Y_test),shade=True, color="c", bw=0.5)
sns.kdeplot(np.array(Y_pred),shade=True, color="r", bw=0.5)
```

Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x7495c19278>



sklearn.linear_model.SGDRegressor

alpha is as learning rate

n_iter is as batch size

In [21]:

```
models_performence1 = {
    'Model':[],
    'Batch_Size':[],
    'RMSE': [],
    'MSE':[],
    'Iteration':[],
    'Optimal learning Rate':[],


}
columns = ["Model","Batch_Size","RMSE","MSE", "Iteration", "Optimal lea
rning Rate"]
pd.DataFrame(models_performence1, columns=columns)
```

Out[21]:

Model	Batch_Size	RMSE	MSE	Iteration	Optimal learning Rate

In [22]:

```
def square(list):
        return [(i ** 2) for i in list]
```

In [23]:

```
from sklearn import linear_model
import warnings
warnings.filterwarnings("ignore")
#Here, alpha is as learning rate

def sgdreg_function(x,initial_batch_size):
    #initial_batch_size=100
    batch=[]
```

```
    for l in range(x):
        batch_size_value= initial_batch_size + initial_batch_size * l
        batch.append(batch_size_value)
        z=0
        scale_max=np.max(Y_test[0:batch_size_value])

        Learning_rate=1 # initial learning rate=1
        score=[]
        LR=[] # storing value for learning rate
        Total_score=[]
        epoch1=[]
        global delta_error
        delta_error=[]
        Y_Test=[]
        global Y_hat_Predicted
        Y_hat_Predicted=[]
        test_cost=[]
        train_cost=[]
        n_iter=100
        for k in range(1,batch_size_value+1):
            # Appending learning rate
            LR.append(Learning_rate)

            # SGDRegressor
            sgdreg = linear_model.SGDRegressor(penalty='none',
                                               alpha=Learning_rate
                                               , n_iter=100)



            yii=Y_train[0:batch_size_value]
            xii=X_train[0:batch_size_value]
            xtt=X_test[0:batch_size_value]
            ytt=Y_test[0:batch_size_value]
            Y_Test.append(ytt)

            clf=sgdreg.fit(xii,yii)
            Traing_score=clf.score(xii,yii)
            train_cost.append(Traing_score)
            training_error=1-Traing_score
```

```
            # p predicting on x_test

            y_hat = sgdreg.predict(xtt)
            #testing_score=clf.score()
            clf1=sgdreg.fit(xtt,ytt)
            Testing_score=clf1.score(xtt,ytt)
            test_cost.append(Testing_score)
            Testing_error=1-Testing_score
            Y_hat_Predicted.append(y_hat)
            # error = Y_test - y_prediction
            err = abs(ytt - y_hat)
            delta_error.append(err)

            score.append(Testing_score)
            # print(rmse)

            # Iteration
            iteration_no=sgdreg.n_iter_
            epoch1.append(iteration_no)
            #print('Epoch=',iteration_no)
            #print('Learning_rate',Learning_rate)

            Learning_rate=Learning_rate/2
            z+=1
        print("Training Error=",training_error)
        print("Testing_error",Testing_error)

        models_performence1['Model'].append('sklearn.linear_model.SGDRe
gressor')
        # graph (Y_test) Prices Vs  (Y_prediction) Predicted prices
        fig4 = plt.figure( facecolor='c', edgecolor='k')
        fig4.suptitle('(Y_test) Prices Vs  (Y_prediction) Predicted pri
ces: $Y_i$ vs $\hat{Y}_i$ with batch size='+str(batch[l]), fontsize=12)
        plt.plot(Y_Test,Y_hat_Predicted,'g*')
        plt.plot([0,batch_size_value],[0,batch_size_value], 'r-')

        plt.xlabel('Y_test')
        plt.ylabel('Y_predicted')
```

```
        plt.show()


        # Plot delta_Error and prediction of price
        fig3 = plt.figure( facecolor='y', edgecolor='k')
        fig3.suptitle('delta_Error and prediction of price with batch s
ize='+str(batch[l]), fontsize=12)
        sns.set_style('darkgrid')
        Y_sklearn=np.array(sum(delta_error)/len(delta_error))
        sns.distplot(Y_sklearn,kde_kws={"color": "g", "lw": 3, "label":
 "Delta_error_sklearn"} )
        sns.kdeplot(np.array(y_hat),shade=True, color="r", bw=0.5)
        plt.show()

        # Plot epoch Vs  RMSE
        fig = plt.figure( facecolor='y', edgecolor='k')
        fig.suptitle('epoch Vs  RMSE with batch size='+str(batch[l]), f
ontsize=12)
        ax1 = fig.add_subplot(111)
        plt.plot(epoch1,score,'m*',linestyle='dashed')
        plt.grid()
        plt.xlabel('epoch')
        plt.ylabel('RMSE with batch size=')


        models_performence1['Iteration'].append(sum(epoch1)/len(epoch1
))

        # plot Iterations Vs Train Cost & Test cost
        fig4 = plt.figure( facecolor='c', edgecolor='k')
        fig4.suptitle('Iterations Vs Train Cost & Test cost with batch
 size='+str(batch[l]), fontsize=12)
        plt.plot(epoch1,train_cost,'m*',linestyle='dashed', label='Trai
n cost')
        plt.plot(epoch1,test_cost,'r*', linestyle='dashed',label='Test
 cost')
        plt.legend(loc='lower left')
        plt.grid()
        plt.xlabel('Iterations ')
```

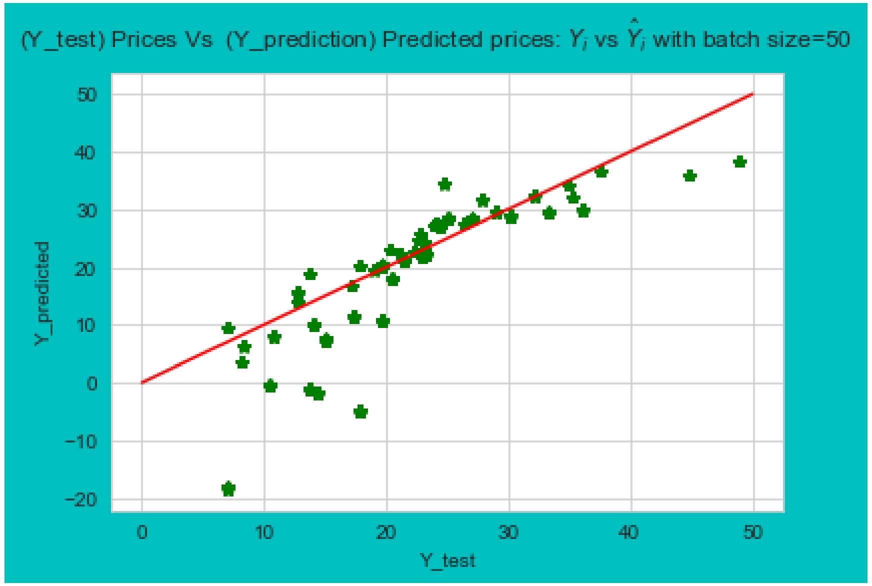
```
        plt.ylabel('Performance Cost  ')
        plt.show()

        # Plot Learning rate Vs  RMSE
        fig2 = plt.figure( facecolor='y', edgecolor='k')
        fig2.suptitle('Learning rate Vs  RMSE with batch size='+str(bat
ch[l]), fontsize=12)
        ax2 = fig2.add_subplot(111)
        #ax2.set_title("Learning rate Vs  RMSE")
        plt.plot(LR,score,'m*',linestyle='dashed')
        plt.grid()
        plt.xlabel('Learning rate')
        plt.ylabel('RMSE')
        plt.show()


        global best_Learning_rate
        best_Learning_rate=LR[score.index(min(score))]
        models_performence1['Optimal learning Rate'].append(best_Learni
ng_rate)
        print('\nThe best value of best_Learning_rate is %d.' % (best_L
earning_rate),7)
        MSEscore=scale_max*sum(score)/len(score)
        score_value=np.sqrt(MSEscore)
        print('Batch Size',batch[l])

        models_performence1['Batch_Size'].append(batch[l])
        print("RMSE with batch size="+str(batch[l]),score_value)
        models_performence1['RMSE'].append(score_value)
        print("MSE with batch size="+str(batch[l]),MSEscore)
        models_performence1['MSE'].append(MSEscore)
```

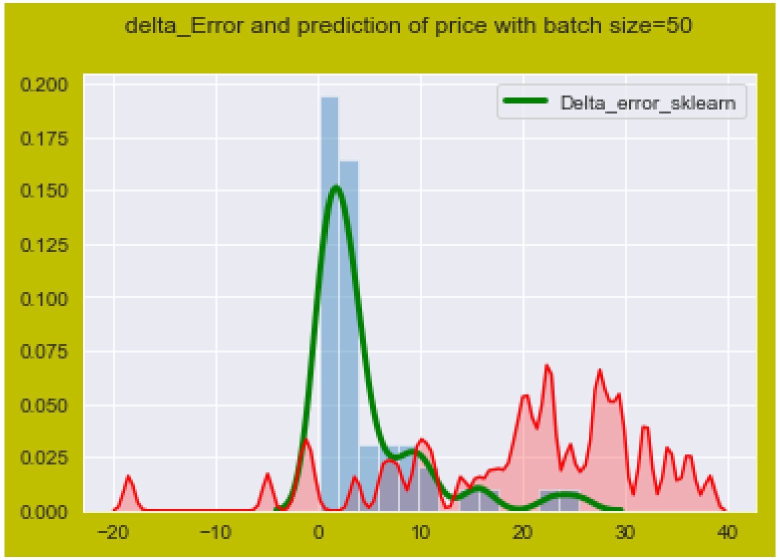
•sgdreg_function is function for stochastic gradient descen for linear regression using linear_model.SGDRegressor in sklearn. •In this function different batch size (50,100,150,200) is applied on linear_model.SGDRegressor to get best learning rate,epoch value,error rate. •here,delta_Error and prediction of price with batch size graph is shown. •RMSE vs epoch graph is shown •Also,RMSE vs learning rate graph is shown for different batch value.

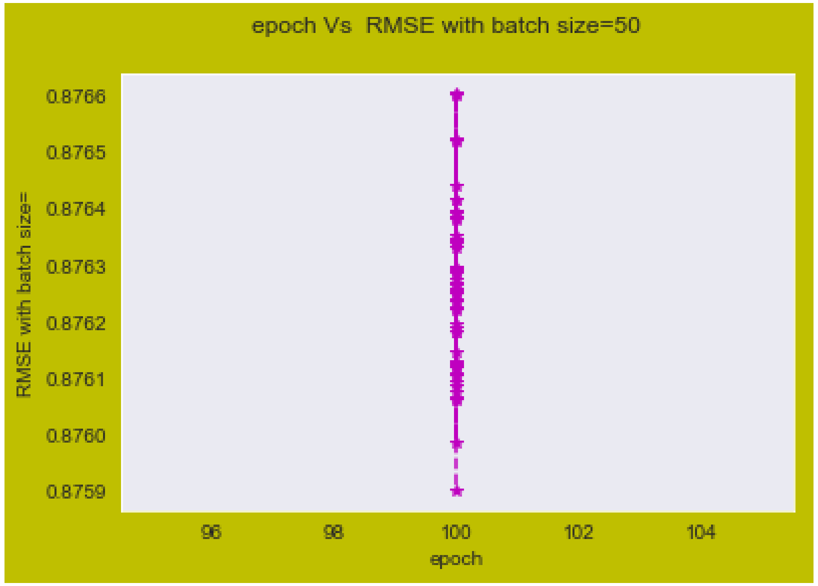
linear_model.SGDRegressor in sklearn for different batch size

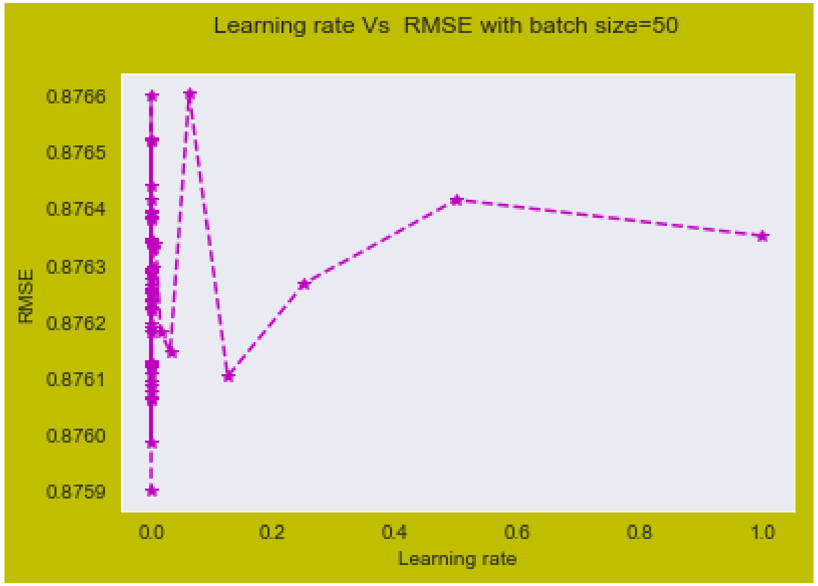
In [24]:
```
sgdreg_function(4,50)
```

```
Training Error= 0.19714574397182938
Testing_error 0.12409867696724153
```



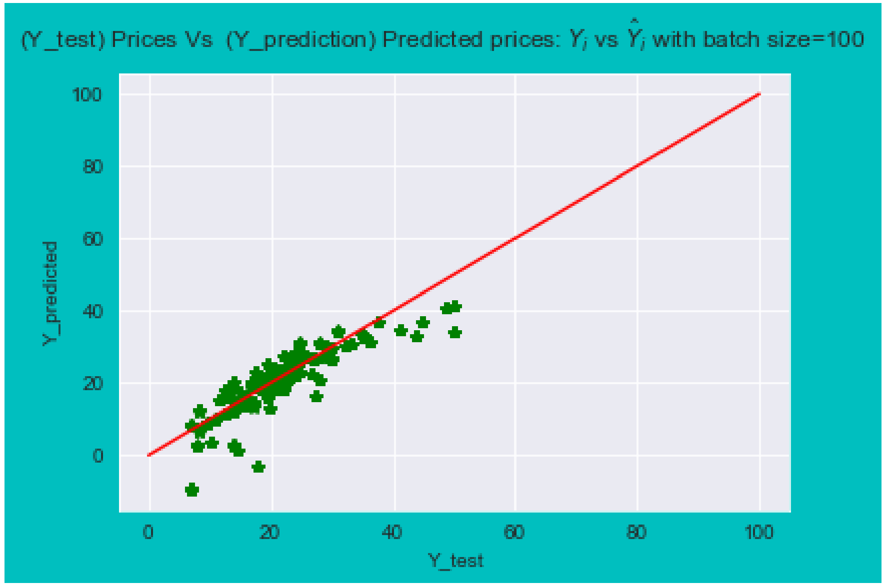

delta_Error and prediction of price with batch size=50
Delta_error_sklearn
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
−20
−10
0
10
20
30
40

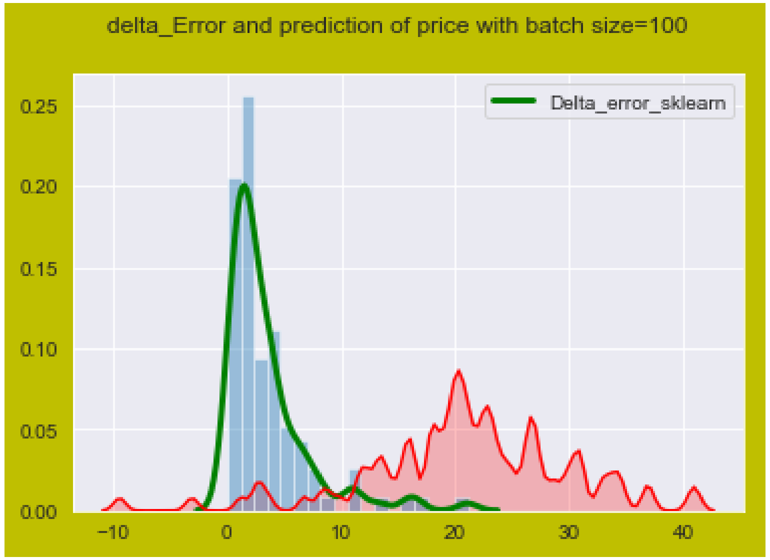

epoch Vs RMSE with batch size=50
RMSE with batch size=
0.8766
0.8765
0.8764
0.8763
0.8762
0.8761
0.8760
0.8759
96
98
100
102
104
epoch

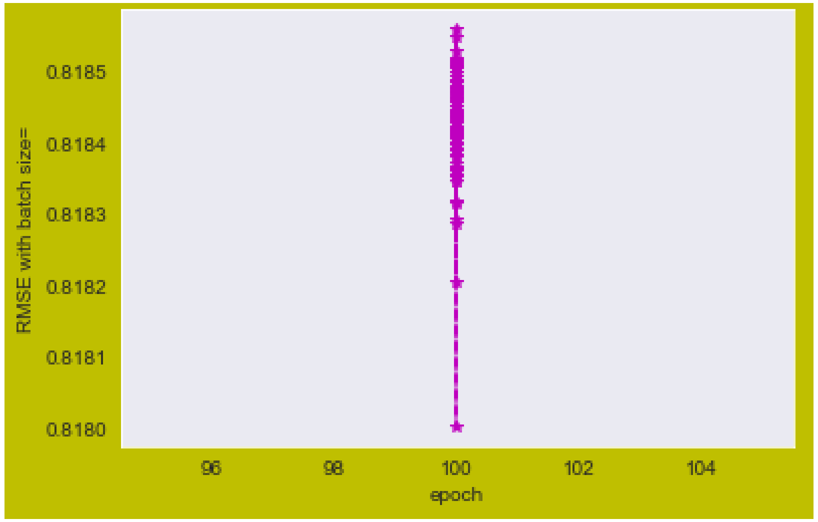


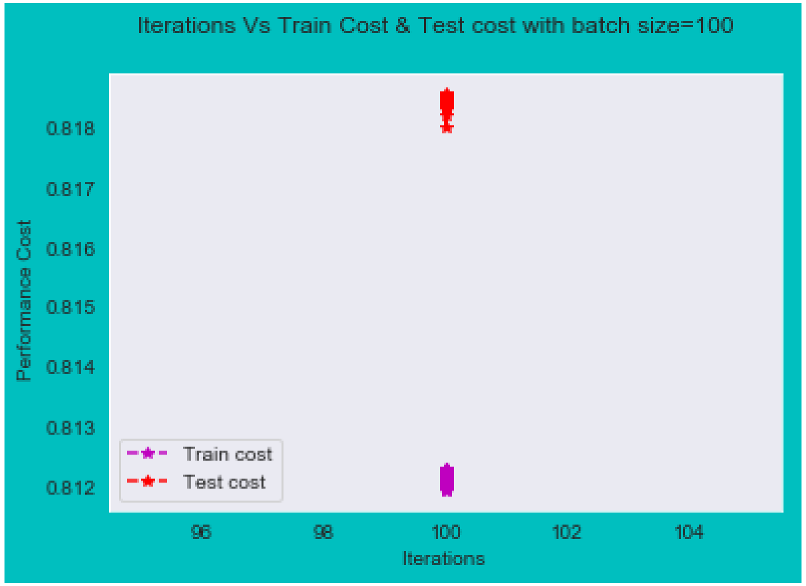

```
The best value of best_Learning_rate is 0. 7
Batch Size 50
RMSE with batch size=50 6.539221090131757
MSE with batch size=50 42.76141246562396
Training Error= 0.18790728380084876
```

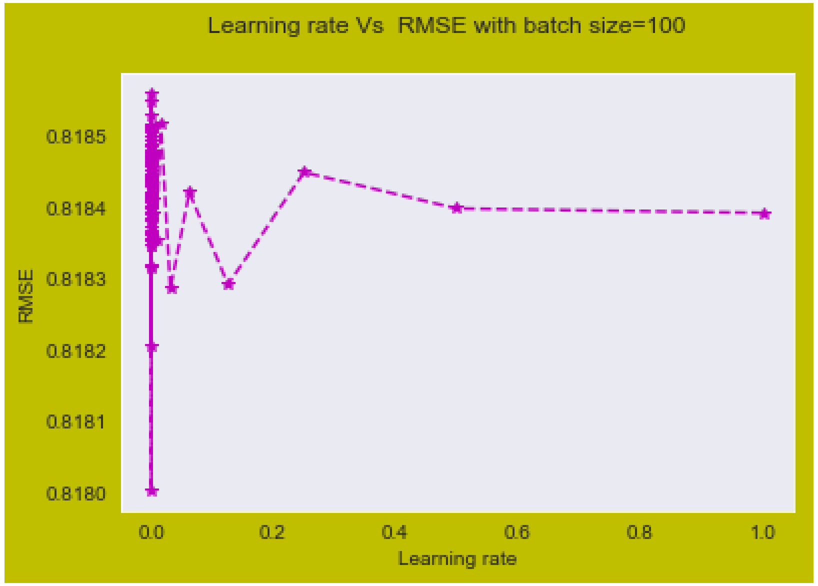
```
Training Error= 0.18790728580684876
Testing_error 0.18164772812457386
```



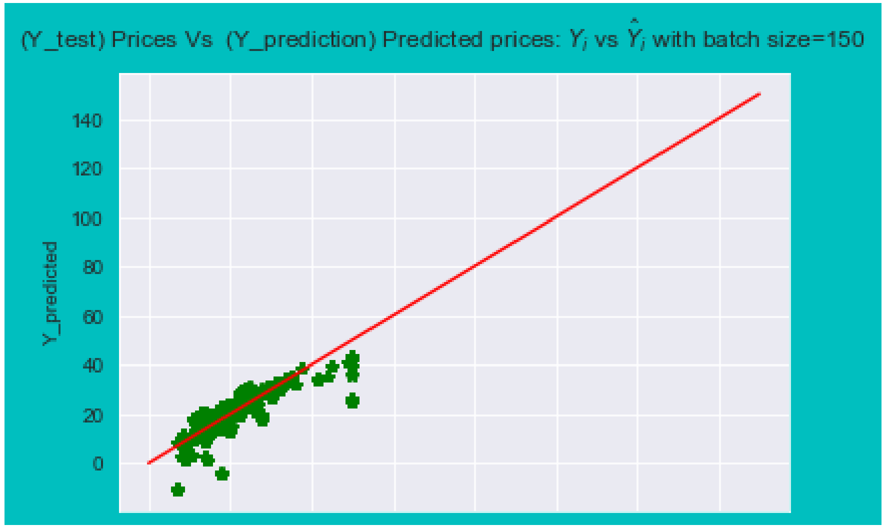



RMSE with batch size=
0.8185
0.8184
0.8183
0.8182
0.8181
0.8180
96
98
100
102
104
epoch

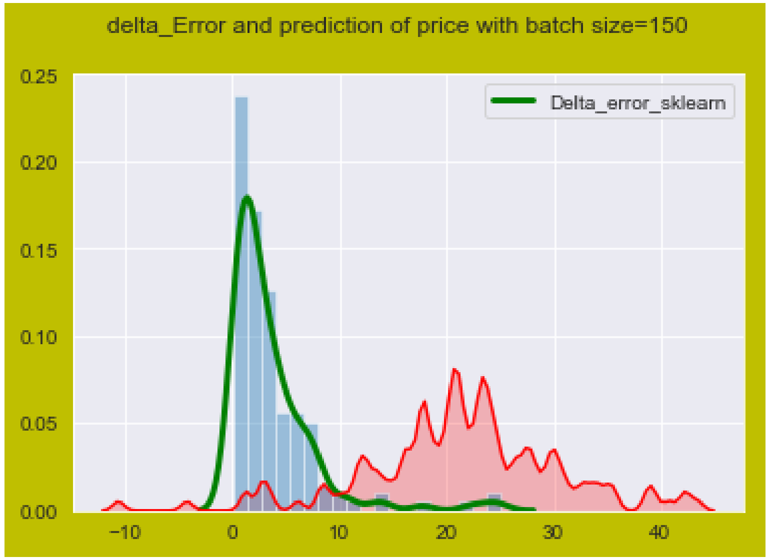

Iterations Vs Train Cost & Test cost with batch size=100
Performance Cost
0.818
0.817
0.816
0.815
0.814
0.813
0.812
Train cost
Test cost
96
98
100
102
104
Iterations

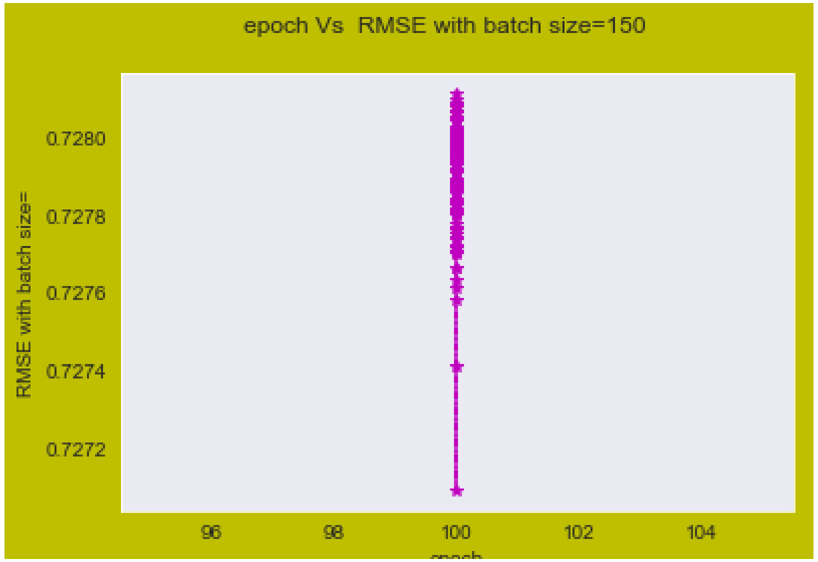


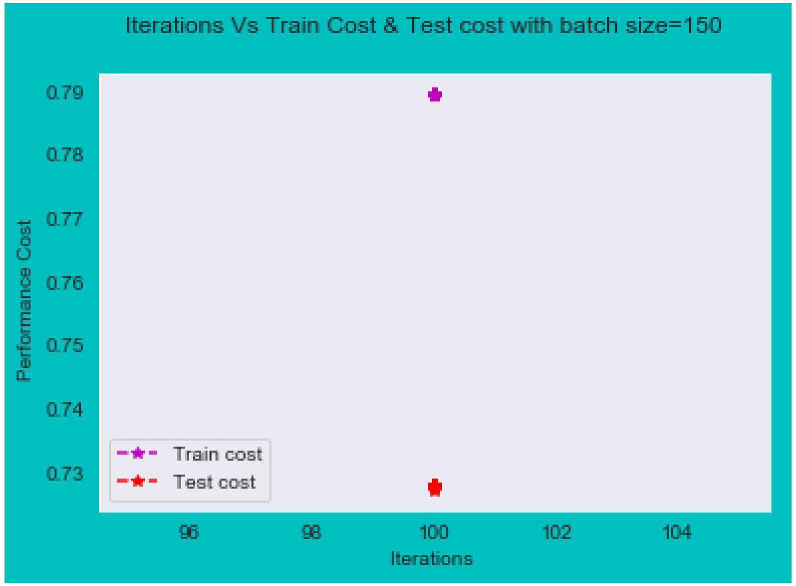

```
The best value of best_Learning_rate is 0. 7
Batch Size 100
RMSE with batch size=100 6.396958021293234
MSE with batch size=100 40.92107192618785
Training Error= 0.21071199877855762
Testing_error 0.2719467260152202
```

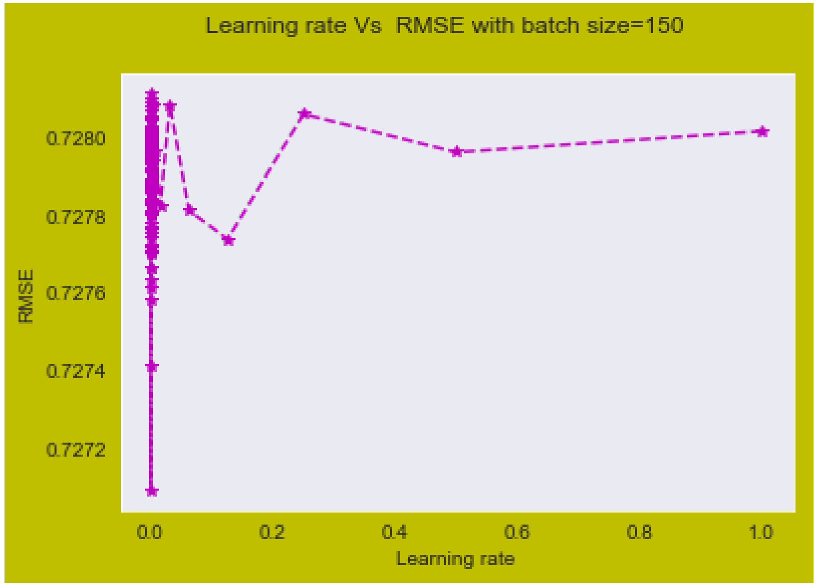



0
20
40
60
80
100
120
140
Y_test


delta_Error and prediction of price with batch size=150
Delta_error_sklearn
0.25
0.20
0.15
0.10
0.05
0.00
−10
0
10
20
30
40

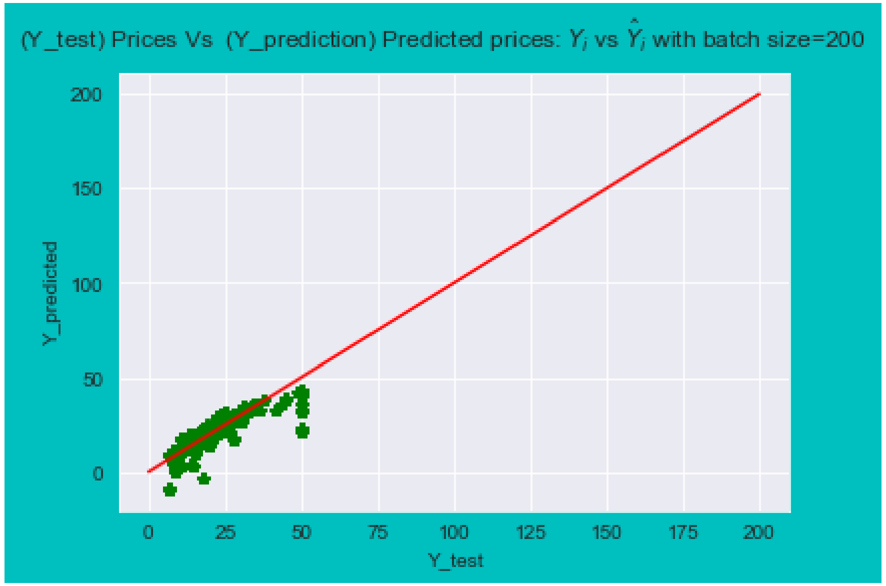

epoch Vs RMSE with batch size=150
RMSE with batch size=
0.7280
0.7278
0.7276
0.7274
0.7272
96
98
100
102
104

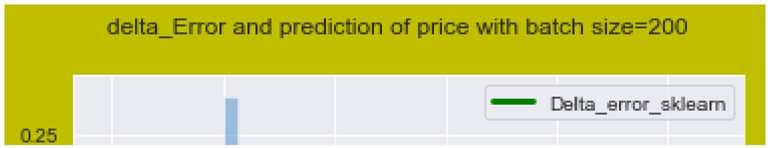


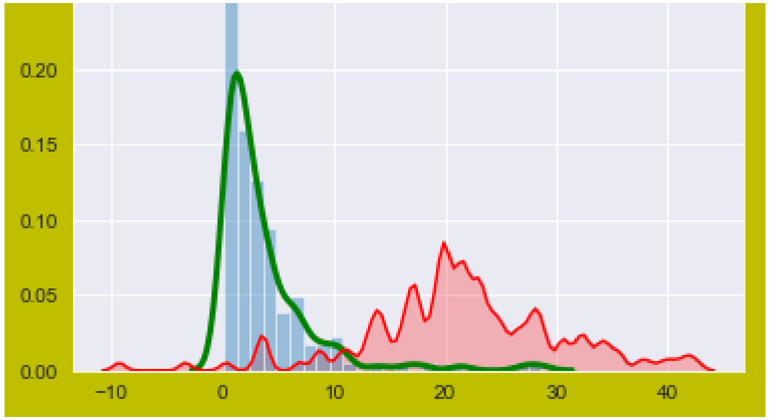


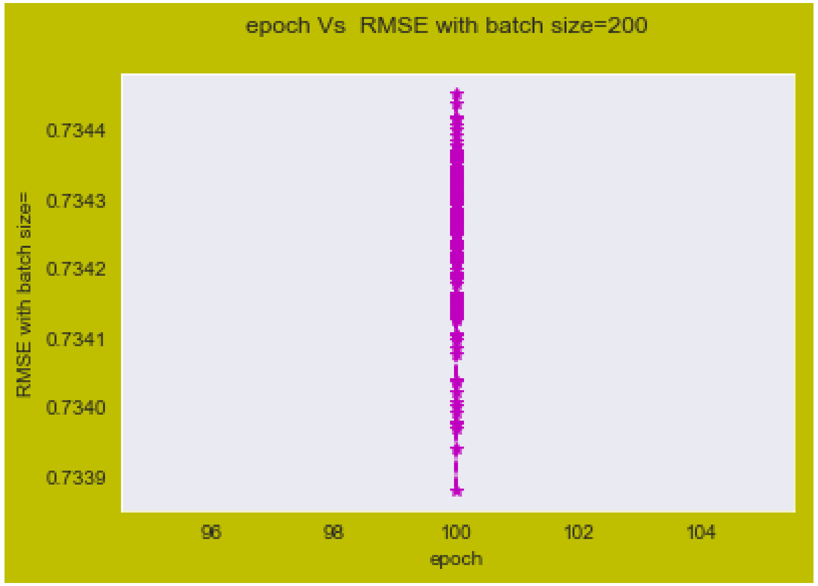

```
The best value of best Learning rate is 0. 7
```

```
The best value of best_Learning_rate is 0.7
Batch Size 150
RMSE with batch size=150 6.032853573183079
MSE with batch size=150 36.395322235467845
Training Error= 0.19677255246784875
Testing_error 0.2658726952194034
```

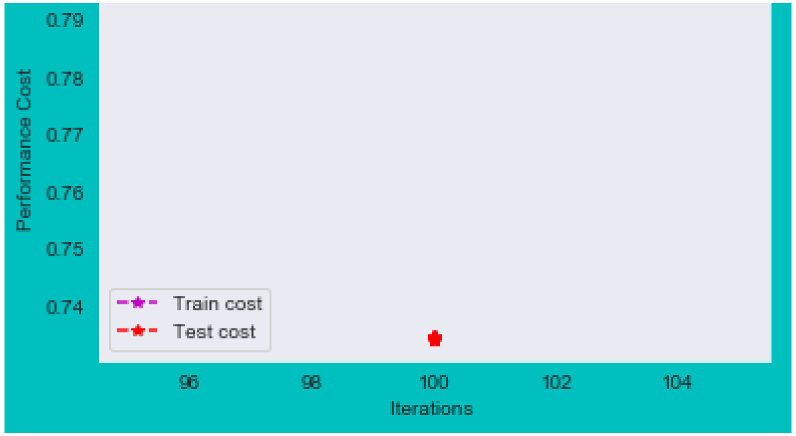


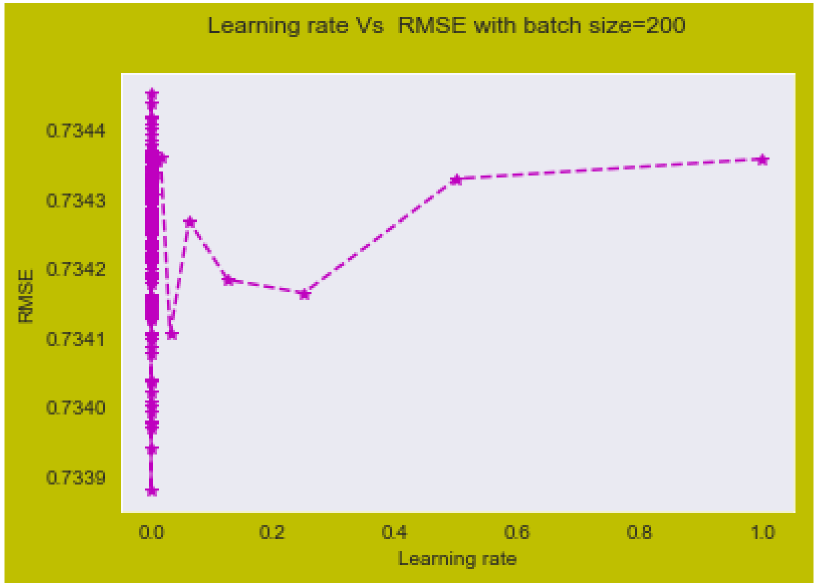



0.20
0.15
0.10
0.05
0.00
−10
0
10
20
30
40


epoch Vs RMSE with batch size=200
RMSE with batch size=
0.7344
0.7343
0.7342
0.7341
0.7340
0.7339
96
98
100
102
104
epoch


Iterations Vs Train Cost & Test cost with batch size=200
0.80






```
The best value of best_Learning_rate is 0. 7
Batch Size 200
RMSE with batch size=200 6.059120170444954
MSE with batch size=200 36.71293723989289
```

```
In [25]: columns = ["Model","Batch_Size","RMSE","MSE", "Iteration", "Optimal lea
rning Rate"]
pd.DataFrame(models_performence1, columns=columns)
```

Out[25]:

	Model	Batch_Size	RMSE	MSE	Iteration	Optimal learning Rate
0	sklearn.linear_model.SGDRegressor	50	6.539221	42.761412	100.0	1.776357e-15
1	sklearn.linear_model.SGDRegressor	100	6.396958	40.921072	100.0	4.440892e-16
2	sklearn.linear_model.SGDRegressor	150	6.032854	36.395322	100.0	3.231174e-27
3	sklearn.linear_model.SGDRegressor	200	6.059120	36.712937	100.0	1.147944e-41

Observation:

•In sklearn SGDRegressor,It is observed that as batch size increases optimal learning rate decreses.

•RMSE value is around 5 and MSE value is around 30

•RMSE value for batch size 100 is high comparatively with others batch size.

•For Batch size=200, RMSE & learning Rate is lowest.

Standardization training and testing data according to batch size

Manual SGD function

L(w,b)=min w,b{sum(square{yi-wTxi-b})}

Derivative of Lw w.r.t w ==>

Lw= sum({-2*xi}{yi-wT.xi-b})

Derivative of Lb w.r.t b==>

lb=sum(-2*{yi-wTxi-b})¶

In [26]:
```
models_performence1 = {
    'Model':[],
    'Batch_Size':[],
    'RMSE': [],
    'MSE':[],
    'Iteration':[],
    'Optimal learning Rate':[],
```

```
}
columns = ["Model","Batch_Size","RMSE","MSE", "Iteration", "Optimal lea
rning Rate"]
pd.DataFrame(models_performence1, columns=columns)
```

Out[26]:

Model	Batch_Size	RMSE	MSE	Iteration	Optimal learning Rate

In [27]:

```
def denorm(scale,list):
    return [(scale*i) for i in list]

# scale
scale=np.max(Y_test)
print(scale)
```

```
50.0
```

In [28]:

```
# SGD function
#L(w,b)=min w,b{sum(square{yi-wTxi-b})}
def SGD(batch_size):
    X_batch_size =X_train[:batch_size]
    price_batch_size =Y_train[:batch_size]
    X_test_batch=X_test[:batch_size]
    ytt_batch_size= Y_test[:batch_size]




    N = len(X_batch_size)

    xi_1=[]
    yprice=[]
    xtt=[]
    ytt=[]
    ytt1=[]
    for j in range(N):
        # standardization of datasets
```

```
        scaler = StandardScaler()
        scaler.fit(X_batch_size)
        X_scaled_batch_size = scaler.transform(X_batch_size)
        X_scaled_batch_size=preprocessing.normalize(X_scaled_batch_size
)
        xi_1.append(X_scaled_batch_size)

        X_test_batch_size=scaler.transform(X_test_batch)
        X_test_batch_size=preprocessing.normalize(X_test_batch_size)
        xtt.append(X_test_batch_size)
        Y_scaled_batch_size=np.asmatrix(price_batch_size)
        #Y_scaled_batch_size=preprocessing.normalize(Y_scaled_batch_siz
e)
        yprice.append(Y_scaled_batch_size)
        Ytt_scaled_batch_size1=np.asmatrix(Y_test[:batch_size])
        Ytt_scaled_batch_size=preprocessing.normalize(Ytt_scaled_batch_
size1)
        ytt1.append(Ytt_scaled_batch_size1)
        ytt.append(Ytt_scaled_batch_size)

    xi=xi_1
    price=yprice

    Lw = 0
    Lb = 0
    learning_rate = 1
    iteration = 1
    w0_random = np.random.rand(13)
    w0 = np.asmatrix(w0_random).T
    b = np.random.rand()
    b0 = np.random.rand()
    global learning_rate1
    learning_rate1=[]
    global epoch
    epoch=[]
    global rmse1
    rmse1=[]
    global y_hat_manual_SGD
    y_hat_manual_SGD=[]
```

```
    global delta_Error
    delta_Error=[]


    while True:
        learning_rate1.append(learning_rate)
        epoch.append(iteration)

        for i in range(N):
            wj=w0
            bj=b0
            #derivative of Lw w.r.t w
            #Lw= sum({-2*xi}{yi-wT.xi-b})
            #print(price[i] .shape)
            Lw = (1/N)*np.dot((-2*xi[i].T ), (price[i] - np.dot( xi[i],
wj) - bj))
            #derivative of Lb w.r.t b
            #lb=sum(-2*{yi-wTxi-b})
            Lb = (-2/N)*(price[i] - np.dot( xi[i],wj ) - bj)
            #print('yi',Lw.shape)
            y_new=(1/N)*(xtt[i].dot(Lw))+Lb
            #print(y_new[i])
            y_pred=np.absolute(np.array(y_new[i]))
            y_hat_manual_SGD.append( y_pred)

            delta_error = np.absolute(np.array(ytt[i] ) - np.array(y_ne
w[i]))
            delta_Error.append(delta_error.mean())
            #delta_error=price[i]  - y_new[i]

            error=np.sum(np.dot(delta_error ,delta_error.T))

        rmse1.append(error)

        w0_new = Lw * learning_rate
        b0_new = Lb * learning_rate
        wj = w0 - w0_new
        bj = b0 - b0_new
        iteration += 1
```

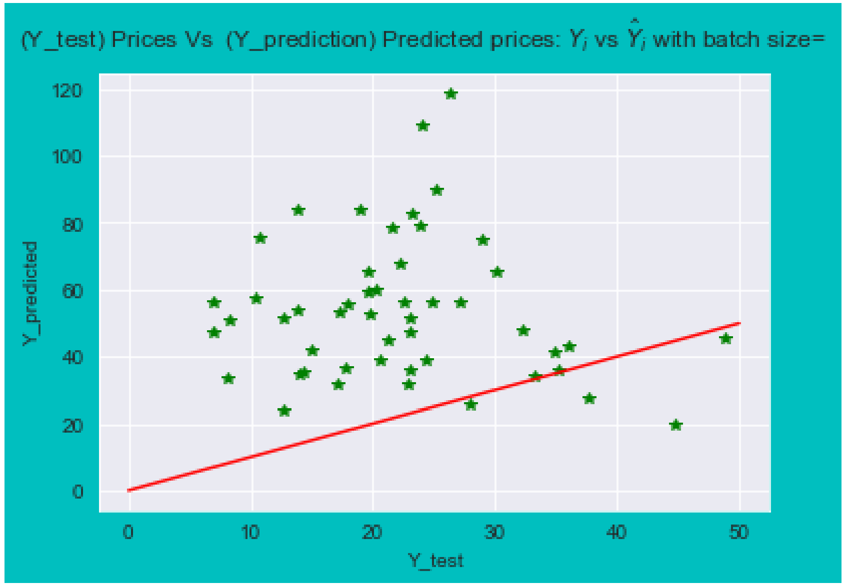
```
        if (w0==wj).all():
            break
        else:
            w0 = wj
            b0 = bj
            learning_rate = learning_rate/2

    print('For batch size'+str(batch_size))


    RMSE=(scale*np.asarray(rmse1))

    # Y_test function
    vvv=denorm(1,ytt1)
    cv=vvv[0]
    # Y_hat_test function after normationzation
    cvv=denorm(scale,y_hat_manual_SGD[batch_size])
    #print(sum(delta_error)/len(delta_error))
    fig4 = plt.figure( facecolor='c', edgecolor='k')
    fig4.suptitle('(Y_test) Prices Vs  (Y_prediction) Predicted prices:
 $Y_i$ vs $\hat{Y}_i$ with batch size=', fontsize=12)
    plt.plot(cv,cvv,'g*')
    plt.plot([0,batch_size],[0,batch_size], 'r-')

    plt.xlabel('Y_test')
    plt.ylabel('Y_predicted')
    plt.show()

     # Plot delta_Error and prediction of price
    fig3 = plt.figure( facecolor='y', edgecolor='k')
    fig3.suptitle('delta_Error  with batch size='+str(batch_size), font
size=12)
    sns.set_style('darkgrid')
    sns.distplot(np.array(delta_Error),kde_kws={"color": "r", "lw": 3,
"label": "Delta_error_manual"} )
    #sns.kdeplot(np.array(ghy),shade=True, color="r", bw=0.5)
    plt.show()

    #For plotting epoch vs RMSE
```

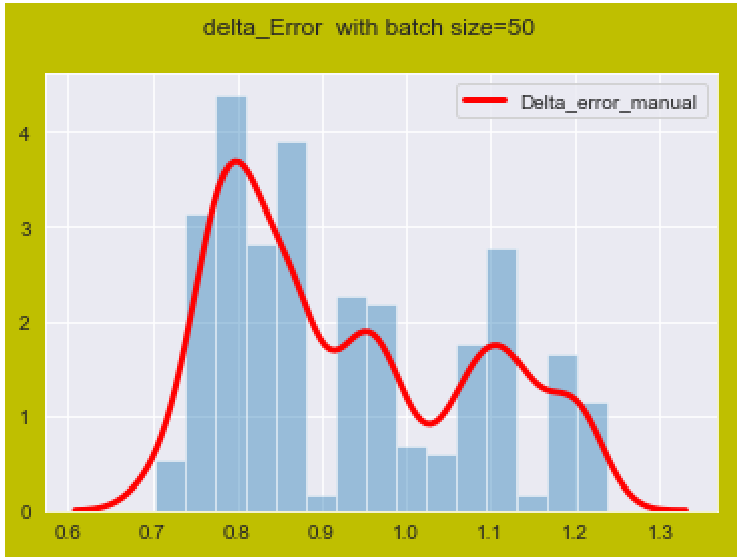
```
    models_performence1['Model'].append('SGD Manual Function')
    models_performence1['Batch_Size'].append(batch_size)
    fig = plt.figure( facecolor='c', edgecolor='k')
    fig.suptitle('epoch Vs RMSE with batch size='+str(batch_size), font
size=12)
    ax1 = fig.add_subplot(111)
    plt.plot(epoch,RMSE,'r*',linestyle='dashed')
    plt.xlabel('epoch')
    plt.ylabel('RMSE with batch size='+str(batch_size))
    plt.plot(epoch,RMSE,'y',linestyle='dashed')
    plt.show()

    #Best learning rate
    global best_Learning_rate1
    best_Learning_rate1=learning_rate1[rmse1.index(min(rmse1))]
    print('\nThe best value of best_Learning_rate is %d.' % (best_Learn
ing_rate1))
    models_performence1['Optimal learning Rate'].append(best_Learning_r
ate1)
    fig1 = plt.figure( facecolor='y', edgecolor='k')
    fig1.suptitle('Learning rate Vs RMSE with batch size='+str(batch_si
ze), fontsize=12)
    ax1 = fig1.add_subplot(111)
    plt.plot(learning_rate1,rmse1,'m*')
    plt.xlabel('Learning rate')
    plt.ylabel('RMSE')


    global RMSE_value
    MSE_value = sum(rmse1)/len(rmse1)
    print("MSE_value=",MSE_value  )
    models_performence1['MSE'].append(MSE_value)
    RMSE_value =np.sqrt(MSE_value)
    models_performence1['RMSE'].append(RMSE_value)


    models_performence1['Iteration'].append(iteration)

    print("RMSE = ",RMSE_value)
```

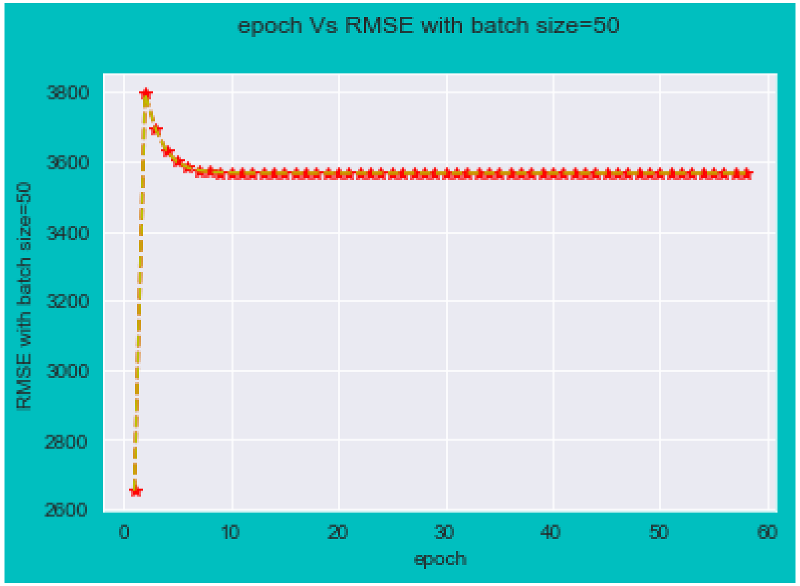
```
    print('For batch size'+str(batch_size))


    print('iteration =',iteration)

    print('Total number of learning_rate=',len(learning_rate1))
    plt.plot(learning_rate1,rmse1,'y',linestyle='dashed')
    plt.show()
```

In [29]:
```
from sklearn.preprocessing import StandardScaler

initial_batch_size=50

for l in range(4):
    batch_size_value= initial_batch_size + initial_batch_size * l

    print(batch_size_value)
    SGD(batch_size_value)
```

```
50
For batch size50
```

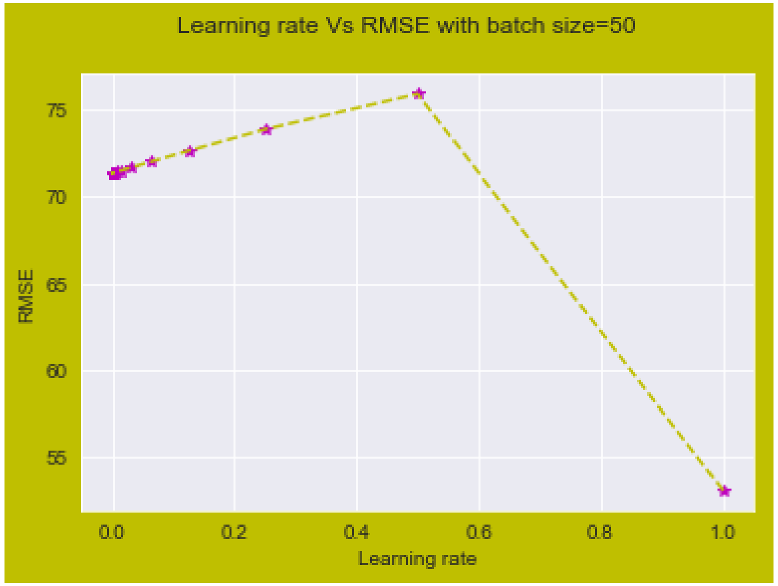




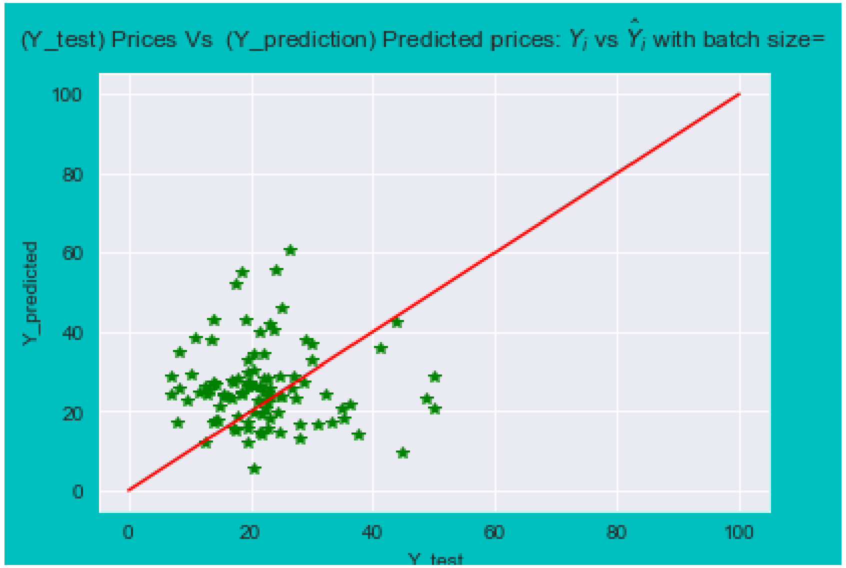


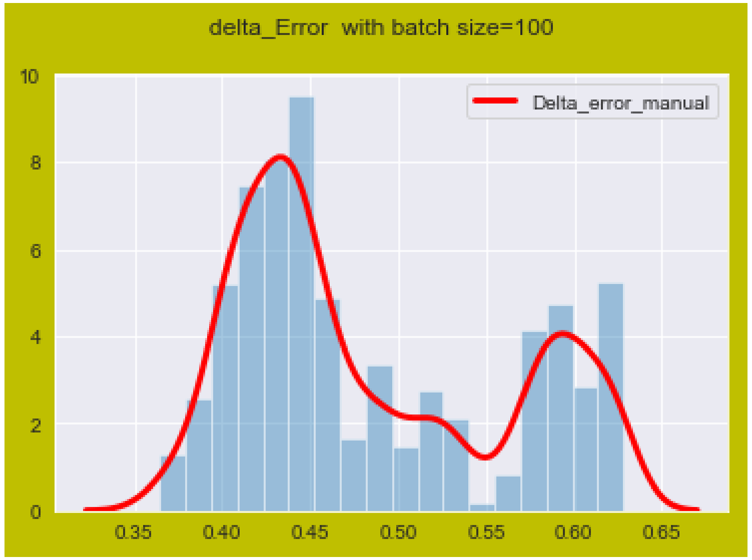

```
The best value of best_Learning_rate is 1.
MSE_value= 71.1524938511473
RMSE =  8.435193764884556
For batch size50
```

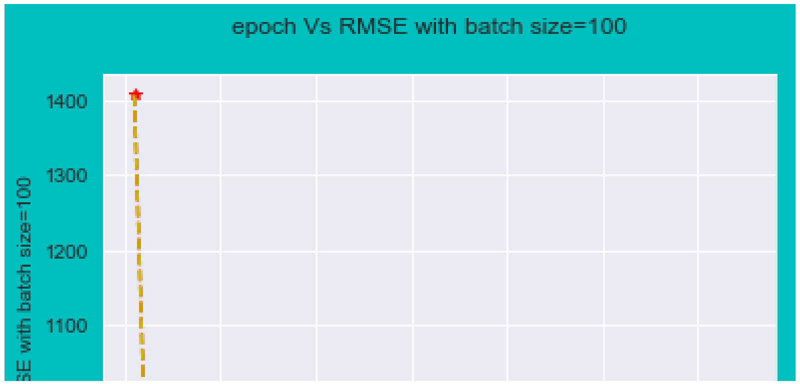
```
iteration = 59
Total number of learning_rate= 58
```

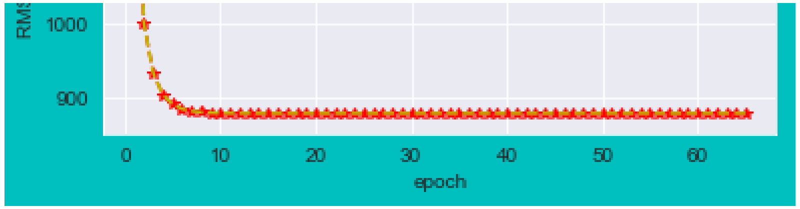



```
100
For batch size100
```

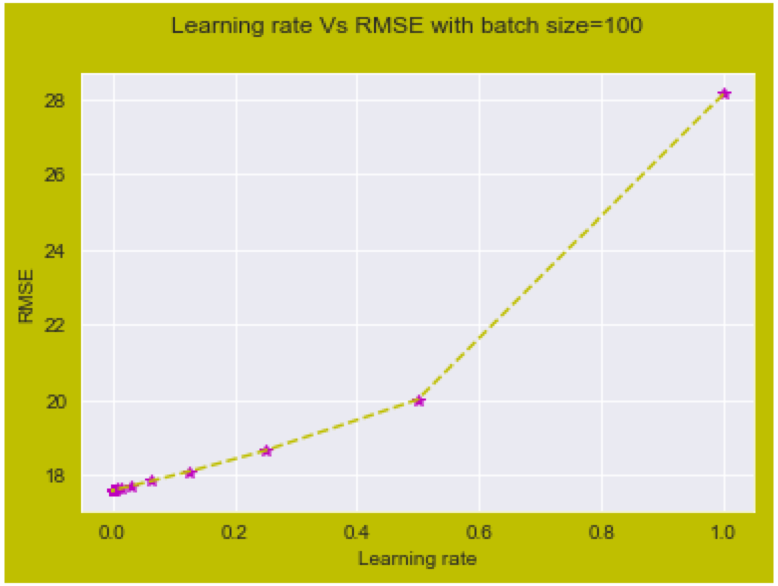



delta_Error with batch size=100
Delta_error_manual
10
8
6
4
2
0
0.35
0.40
0.45
0.50
0.55
0.60
0.65

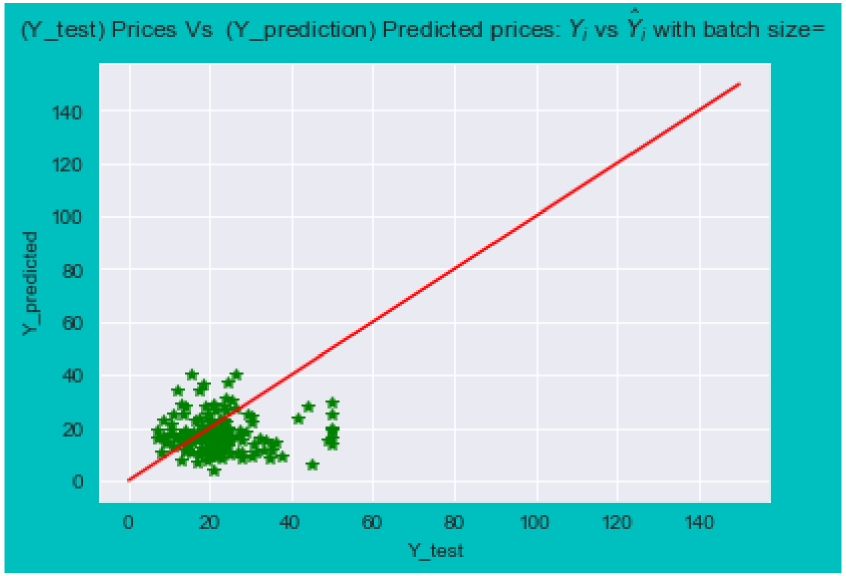

epoch Vs RMSE with batch size=100
1400
1300
1200
1100

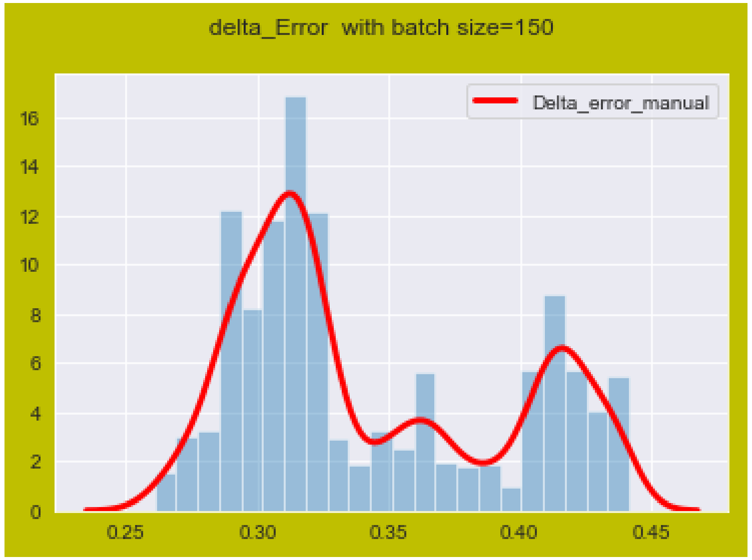


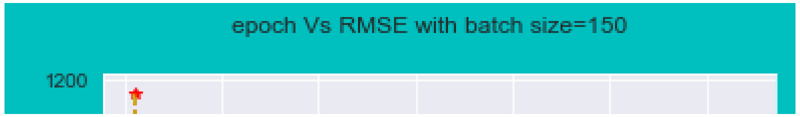

```
The best value of best_Learning_rate is 0.
MSE_value= 17.805143759890868
RMSE =  4.219614171922697
For batch size100
iteration = 66
Total number of learning_rate= 65
```

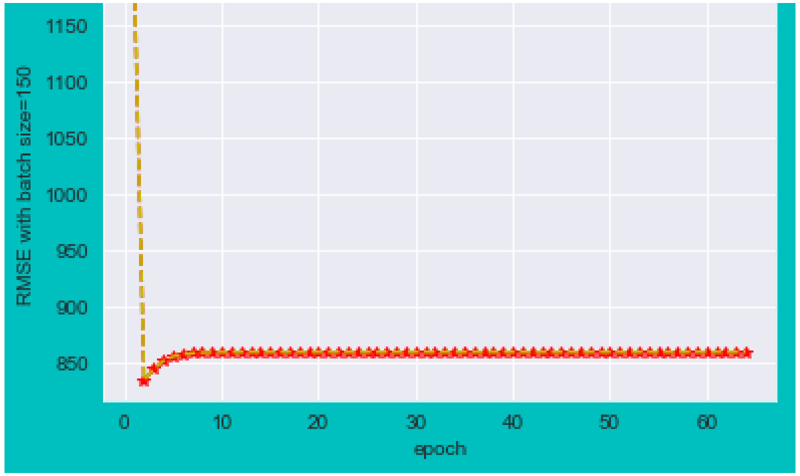



```
150
For batch size150
```

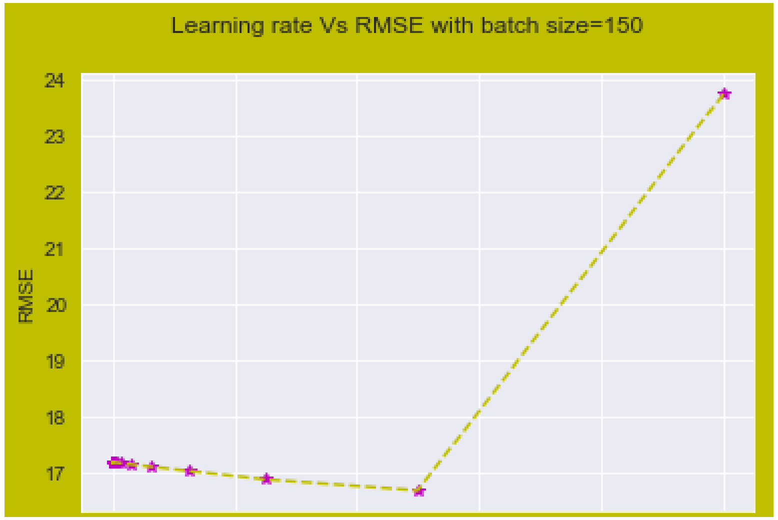

(Y_test) Prices Vs (Y_prediction) Predicted prices: Y_i vs $\hat{Y}_i$ with batch size=
Y_predicted
Y_test
0
20
40
60
80
100
120
140


delta_Error with batch size=150
Delta_error_manual
0
2
4
6
8
10
12
14
16
0.25
0.30
0.35
0.40
0.45


epoch Vs RMSE with batch size=150
1200




```
The best value of best_Learning_rate is 0.
MSE_value= 17.283357889550995
RMSE =  4.157325809886807
For batch size150
iteration = 65
Total number of learning_rate= 64
```




```
200

--------------------------------------------------------------------------
----
ValueError                                Traceback (most recent call l
ast)
<ipython-input-29-a9ede5e53bcb> in <module>
      7
      8     print(batch_size_value)
----> 9     SGD(batch_size_value)

<ipython-input-28-b76a2cc929a9> in SGD(batch_size)
     74             Lb = (-2/N)*(price[i] - np.dot( xi[i],wj ) - bj)
     75             #print('yi',Lw.shape)
---> 76             y_new=(1/N)*(xtt[i].dot(Lw))+Lb
     77             #print(y_new[i])
     78             y_pred=np.absolute(np.array(y_new[i]))

ValueError: operands could not be broadcast together with shapes (152,2
00) (200,200)
```

In [ ]:

```
columns = ["Model","Batch_Size","RMSE","MSE", "Iteration", "Optimal lea
rning Rate"]
pd.DataFrame(models_performence1, columns=columns)
```

SGD_Manual Vs SGD_sklearn

In [ ]:
```
models_performence1 = {
    'Model':[],
    'Batch_Size':[],
    'RMSE': [],
    'MSE':[],
    'Iteration':[],
    'Optimal learning Rate':[],


}
columns = ["Model","Batch_Size","RMSE","MSE", "Iteration", "Optimal lea
rning Rate"]
pd.DataFrame(models_performence1, columns=columns)
```

For batch size 150

In [ ]:
```
SGD(150)
```

In [ ]:
```
sgdreg_function(1,150)
```

Y_predicted using manual SGD Vs Y_predicted using Sklearn SGD

Y_predicted using manual SGD == y_hat_manual_SGD

Error(y-y_hat) for manual SGD == delta_Error

Y_predicted using Sklearn SGD == Y_hat_Predicted

Error(y-y_hat) for SKlearn SGD == delta_error

In [ ]:
```
def y_hat_cal(delta_error_sklearn,delta_Error_manual):
    fig41 = plt.figure( facecolor='y', edgecolor='k')
    fig41.suptitle('Y_predicted using manual SGD Vs Y_predicted using S
klearn SGD ', fontsize=12)

    sns.set_style('darkgrid')
    Y_sklearn=np.array(sum(delta_error_sklearn)/len(delta_error_sklearn
))

    Y_manual=np.array(delta_Error_manual)
    #print(Y_manual[0])
    sns.distplot(Y_sklearn,kde_kws={"color": "g", "lw": 3, "label": "De
lta_error_sklearn"} )
    sns.distplot(Y_manual,kde_kws={"color": "r", "lw": 3, "label": "Del
ta_error_manual"} )
    fig51 = plt.figure( facecolor='y', edgecolor='k')
    fig51.suptitle('Y_predicted using manual SGD ', fontsize=12)
    sns.distplot(Y_sklearn,kde_kws={"color": "g", "lw": 3, "label": "De
lta_error_sklearn"} )

    fig41 = plt.figure( facecolor='y', edgecolor='k')
    fig41.suptitle(' Y_predicted using Sklearn SGD ', fontsize=12)
    sns.distplot(Y_manual,kde_kws={"color": "r", "lw": 3, "label": "Del
ta_error_manual"} )
```

In [ ]:
```
y_hat_cal(delta_error,delta_Error)
```

In [ ]:
```
def y_skl_maual(y_hat_sklearn,y_hat_maunal):
    fig41 = plt.figure( facecolor='y', edgecolor='k')
    fig41.suptitle('Delta_error using manual SGD Vs Delta_error using S
klearn SGD ', fontsize=12)

    sns.set_style('whitegrid')
    Y_sklearn=np.array(sum(y_hat_sklearn)/len(y_hat_sklearn))

    Y_manual=np.array(scale*sum(y_hat_maunal)/len(y_hat_maunal))
    #print(Y_manual[0])
```

```
    sns.kdeplot(Y_sklearn,shade=True, color="c", bw=0.5,label='Y_hat_sk
learn')
    sns.kdeplot(Y_manual[0],shade=True, color="r", bw=0.5,label='Y_hat_
manual')
```

In [ ]:
```
y_skl_maual(Y_hat_Predicted,y_hat_manual_SGD)
```

In [ ]:
```
pd.DataFrame(models_performence1, columns=columns)
```

Observation

•In stochastic gradient descent Manual model(a user designed model),RMSE(root mean squared error) is varied as compared to sklearn designed stochastic gradient descent model for varied number of batch_size.

•Graphs between learning rate vs RMSE & Epoch Vs RMSE are plotted.

•From the graph , stochastic gradient descent model performance can be observed .

Comparision of SGD_sklearn and SGD_manual with batch_size=150 :-

- Distributions Plots for errors(y - y_hat) and It is overlapping as shown in graph "y_hat_cal(delta_error,delta_Error)".Seperate distribuation plots for both of implementations are plotted below it.
- "Delta_error using manual SGD Vs Delta_error using Sklearn SGD" graph is plotted .Varience(spread) of Blue graph(SGD sklearn) is high as comapared to spread of Red graph (manual SGD) .
- RMSE Vs epoch for manual SGD graph looks like almost "L" shape.So, Model doesn't leads to overfitting. In case od SGD sklearn , it is straight vertical line at epoch.
- RMSE value and MSE value for batch_size 150 is almost similar as seen in above table

- Optimal learning rate is low for SGD sklearn and 1 which high in this case is for SGD manual.

```
In [ ]:
```