Implementation of Stochastic Gradient Descent

Gradient Descent

In Gradient Descent we take baby steps towards the minimum error(Root Mean Square Error(RMSE), also known as cost function). At each step we find the slope.

Image reference: Google Image

Every point in the graph can be plotted as the change in y-axis as x-changes or rate of change of y with x, this is defines the slope of the graph.

Consider the red arrows represents the baby steps we are taking to reach the red dot (Minima of error). We can see from the image that as we reach closer to the minima, the size of each step is decreasing gradually, which means we are becoming more cautious towards reaching the exact minima and not to miss and jump to the other side. This gradual reduction in step size is known as LEARNING RATE.

Also, we can see that we need to have number of steps and find slope at each step, so this needs to be an iterative function with gradually reducing learning rate, which means there will be two hyper-parameters in our algorithm that we need to optimize, number of iterations and learning rate.

Image reference: Google Image

Stochastic Gradient Descent

In Stochastic Gradient Descent, we take random sample data and we iterate through multiple random samples, to identify the least RMSE and the value of coeficients and intercepts are picked from that minimum error iteration

In [365]:

```
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.cross_validation import train_test_split
```

In [366]:

```
boston = load_boston()
X = load_boston().data
Y = load_boston().target
```

In [367]:

```
scaler = preprocessing.StandardScaler().fit(X)
X = scaler.transform(X)
```

About Data

In [368]:

```
print(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 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.
http://archive.ics.uci.edu/ml/datasets/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 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.
   - many more! (see http://archive.ics.uci.edu/ml/datasets/Housing)
```

Sklearn SGDRegressor

In [369]:

```
clf = SGDRegressor()
clf.fit(X, Y)
print(mean_squared_error(Y, clf.predict(X)))
```

```
23.4092452246555
```

In [370]:

```
clf
```

Out[370]:

```
SGDRegressor(alpha=0.0001, average=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, penalty='l2',
       power_t=0.25, random_state=None, shuffle=True, tol=None, verbose=0,
       warm_start=False)
```

In [371]:

```
sgd_coeff = clf.coef_
sgd_coeff
```

Out[371]:

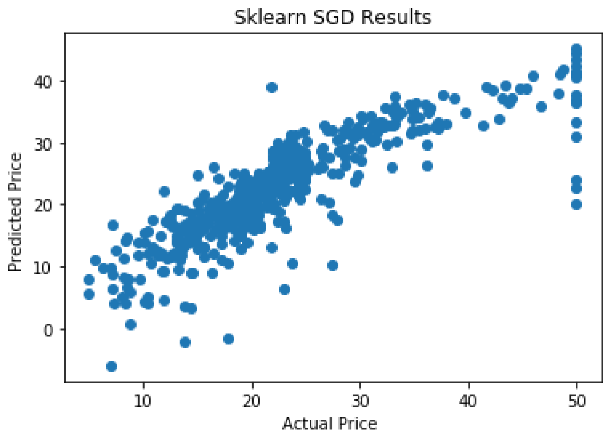
```
array([-0.70090639,  0.60001078, -0.41514316,  0.79515897, -1.02891633,
        3.33591814, -0.18744285, -1.85685412,  0.62017793, -0.43505805,
       -1.8969489 ,  0.95806393, -3.53404411])
```

In [372]:

```
sgd_predict = clf.predict(X)
```

In [373]:

```
plt.scatter(Y,clf.predict(X))
plt.xlabel('Actual Price ')
plt.ylabel('Predicted Price ')
plt.title('Sklearn SGD Results')
plt.show()
```



Scratch Implementation of SGD

Prepare data first

In [374]:

```
boston = load_boston()
data = pd.DataFrame(boston.data)
y_data = pd.DataFrame(boston.target)

train_x,test_x,train_y,test_y = train_test_split(data,y_data,test_size = 0.11,random_state=5)

s=preprocessing.StandardScaler()
train_x=s.fit_transform(train_x)
test_x=s.transform(test_x)
test_x = np.array(test_x)
```

In [375]:

```
test_y = np.array(test_y)
```

In [376]:

```
## for data sampling in "stocastic" gradient descent we will need training x and y in a single dat
aframe.
full_data = pd.DataFrame(train_x)
full_data['price'] = np.array(train_y)
full_data.shape
```

Out[376]:

```
(450, 14)
```

In [377]:

```
#This function will return coefficients and intercept
def sgd(X):
    # initialize variables
    w = np.zeros(shape=(1,13))
    b = 0
    iters = 1
    n_iter = 500
    lr_rate = 1
    error_list = []
    w_list = []
    b_list = []
    w_new = np.zeros(shape=(1,13))
    b_new = 0
    w_hat = np.zeros(shape=(1,13))
    b_hat = 0
    error_sum = 0
    mean_error = 0
    ind = 0
    # create iterations loop
    while(iters<=n_iter):
        w = w_new
        b = b_new
        error_sum = 0
        mean_error = 0
        lr_rate = lr_rate/2
        w_hat = np.zeros(shape=(1,13))
        b_hat = 0
        sample_data = X.sample(25)
        x = np.array(sample_data.drop('price',axis=1))
        y = np.array(sample_data['price'])
        for i in range(len(x)):
            y_curr = w.dot(x[i]) + b
            w_hat += x[i]* (y[i]-y_curr)
            b_hat += (y[i]-y_curr)

        w_hat = w_hat * (-2/len(x))
        b_hat = b_hat * (-2/len(x))
        #new slope and intercept for next iteration in loop.
        w_new = w - (lr_rate*w_hat)
        b_new = b - (lr_rate*b_hat)
        w_list.append(w_new)
        b_list.append(b_new)
        iters = iters + 1
        for i in range(len(y)):
            error = y[i] - (w_new.dot(x[i]) + b_new)
            error_sum += np.sqrt(error**2)
        mean_error = error_sum/(len(y))
        error_list.append(mean_error)
    ind = error_list.index(min(error_list))
    w_best = w_list[ind]
    b_best = b_list[ind]
    return w_best,b_best
```

In [378]:

```
# This is a predict function
```

```python
def predict(x,w,b):
    y_pred = []
    for i in range(len(x)):
        pred = (w.dot(x[i]) + b)
        y_pred.append(pred[0])
    return y_pred
```

In [379]:

```python
w,b = sgd(full_data)
```

In [380]:

```python
w
```

Out[380]:

```
array([[ 0.02521857, -0.70787851, -0.94882701,  2.39146883, -0.73910452,
         3.60685197, -1.18306062, -1.87057354, -0.30644034, -2.61278819,
        -0.89281281,  0.76740657, -0.95946838]])
```

In [381]:

```python
im_predict = predict(test_x,w,b)
```

In [382]:

```python
im_mean_error = mean_squared_error(test_y,im_predict)
im_mean_error
```

Out[382]:

```
24.232579496437772
```

In [383]:

```python
im_coeff = w[0]
```

In [384]:

```python
clf = SGDRegressor()
clf.fit(X, Y)
sgd_test_predict = clf.predict(test_x)
sgd_mean_error = mean_squared_error(test_y, sgd_test_predict)
```

In [385]:

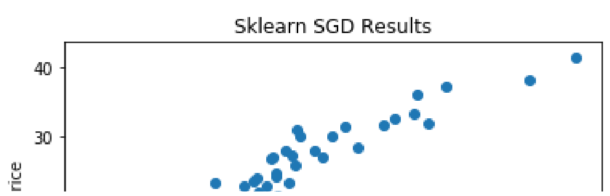
```python
len(sgd_test_predict)
```

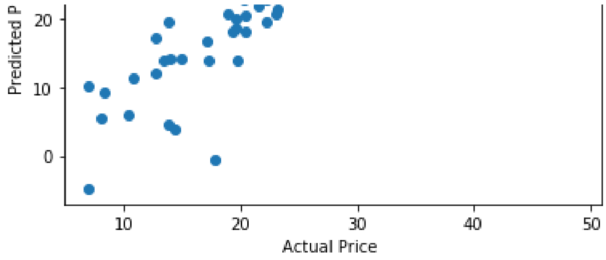
Out[385]:

```
56
```

In [386]:

```python
plt.scatter(test_y,sgd_test_predict)
plt.xlabel('Actual Price ')
plt.ylabel('Predicted Price ')
plt.title('Sklearn SGD Results')
plt.show()
```





In [387]:

```
plt.scatter(test_y,im_predict)
plt.xlabel('Actual Price ')
plt.ylabel('Predicted Price ')
plt.title('Implemented SGD Results')
plt.show()
```

Comparing Coefficients of Sklearn SGD Regressor and Implemented SGD.

In [388]:

```
from prettytable import PrettyTable
x = PrettyTable(['Sklearn SGD Regressor', 'Implemented SGD'])
for i in range(13):
    x.add_row([sgd_coeff[i], im_coeff[i]])

print(x)
```

```
+-----------------------+----------------------+
| Sklearn SGD Regressor |   Implemented SGD    |
+-----------------------+----------------------+
|   -0.7009063862794098  | 0.025218566173158197 |
|    0.6000107832987207  | -0.7078785116028669  |
|   -0.4151431578814445  | -0.9488270053407009  |
|    0.7951589686343118  |  2.391468833198255   |
|   -1.0289163308377336  | -0.7391045186691153  |
|    3.335918143705733   |   3.60685197469129   |
|   -0.18744285444827477 | -1.1830606222815605  |
|   -1.8568541207342748  | -1.8705735398133372  |
|    0.6201779274113803  | -0.30644033550279054 |
|   -0.43505804682241156 | -2.6127881863837605  |
|     -1.8969488983057   | -0.8928128130840008  |
|    0.9580639276734879  |  0.7674065716080524  |
|   -3.5340441127511792  | -0.9594683838768913  |
+-----------------------+----------------------+
```

Comparing first 20 Actual prices with Sklearn SGD and Implemented SGD

In [389]:

```
from prettytable import PrettyTable
z = PrettyTable(['Actual Prices','Sklearn SGD Regressor', 'Implemented SGD'])
```

```
z = PrettyTable(['Actual Prices','Sklearn SGD Regressor', 'Implemented SGD'])
for i in range(20):
    z.add_row([test_y[i], sgd_test_predict[i], im_predict[i]])

print(z)
```

Actual Prices	Sklearn SGD Regressor	Implemented SGD
[37.6]	37.30499790863328	34.17389923660646
[27.9]	29.973372942557884	26.279212342725835
[22.6]	26.699625444282333	26.988057841984155
[13.8]	4.49182175644 5297	4.723324222584026
[35.2]	36.07089228976949	42.144374506615165
[10.4]	6.054974677656137	5.6110631197172545
[23.9]	27.7761754618138	27.12979416998643
[29.]	31.415923523185814	36.099425830535466
[22.8]	26.883984958497564	24.164513571910582
[23.2]	21.434258979124316	15.608953760929877
[33.2]	32.596410229892435	32.59964213927722
[19.]	20.77250247003535	23.178617681703162
[20.3]	22.766759769888267	21.973242121019357
[36.1]	31.945777259026357	31.365690957257257
[24.4]	27.21584340363778	26.78644491563169
[17.2]	16.71137216027777	18.13572414258487
[17.9]	-0.6200613198918035	0.2685254691353336
[19.6]	18.655044580976522	14.92797373410044
[19.7]	13.833848916719273	15.823807298989458
[15.]	14.09989545740942	9.561398893287159

In [390]:

```
e = PrettyTable(['Sklearn SGD MSE', 'Implemented SGD MSE'])
e.add_row([sgd_mean_error, im_mean_error])

print(e)
```

Sklearn SGD MSE	Implemented SGD MSE
20.534387144996092	24.232579496437772