Implement SGD Classifier with Logloss and L2 regularization Using SGD without using sklearn

There will be some functions that start with the word "grader" ex: grader_weights(), grader_sigmoid(), grader_logloss() etc, you should not change those function definition.

Every Grader function has to return True.

Importing packages

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn import linear_model
```

Creating custom dataset

In [2]:
```
# please don't change random_state
X, y = make_classification(n_samples=50000, n_features=15, n_informative=10, n_redundant=5,
                           n_classes=2, weights=[0.7], class_sep=0.7, random_state=15)
# make_classification is used to create custom dataset
# Please check this link (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html) for more details
```

In [3]:
```
X.shape, y.shape
```

Out[3]:
```
((50000, 15), (50000,))
```

Splitting data into train and test

In [4]:
```
#please don't change random state
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=15)
```

In [5]:
```
# Standardizing the data.
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [6]:
```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

Out[6]:
```
((37500, 15), (37500,), (12500, 15), (12500,))
```

SGD classifier

In [7]:
```
# alpha : float
# Constant that multiplies the regularization term.

# eta0 : double
# The initial learning rate for the ‘constant’, ‘invscaling’ or ‘adaptive’ schedules.

clf = linear_model.SGDClassifier(eta0=0.0001, alpha=0.0001, loss='log', random_state=15, penalty='l2', tol=1e-3, verbose=2, learning_rate='constant')
clf
# Please check this documentation (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html)
```

Out[7]:
```
SGDClassifier(eta0=0.0001, learning_rate='constant', loss='log',
              random_state=15, verbose=2)
```

In [8]:
```
clf.fit(X=X_train, y=y_train) # fitting our model
```

```
-- Epoch 1
Norm: 0.70, NNZs: 15, Bias: -0.501317, T: 37500, Avg. loss: 0.552526
Total training time: 0.02 seconds.
-- Epoch 2
Norm: 1.04, NNZs: 15, Bias: -0.752393, T: 75000, Avg. loss: 0.448021
Total training time: 0.03 seconds.
-- Epoch 3
Norm: 1.26, NNZs: 15, Bias: -0.902742, T: 112500, Avg. loss: 0.415724
Total training time: 0.03 seconds.
-- Epoch 4
Norm: 1.43, NNZs: 15, Bias: -1.003816, T: 150000, Avg. loss: 0.400895
Total training time: 0.05 seconds.
-- Epoch 5
Norm: 1.55, NNZs: 15, Bias: -1.076296, T: 187500, Avg. loss: 0.392879
Total training time: 0.06 seconds.
-- Epoch 6
Norm: 1.65, NNZs: 15, Bias: -1.131077, T: 225000, Avg. loss: 0.388094
Total training time: 0.08 seconds.
-- Epoch 7
Norm: 1.73, NNZs: 15, Bias: -1.171791, T: 262500, Avg. loss: 0.385077
Total training time: 0.10 seconds.
-- Epoch 8
Norm: 1.80, NNZs: 15, Bias: -1.203840, T: 300000, Avg. loss: 0.383074
Total training time: 0.12 seconds.
-- Epoch 9
Norm: 1.86, NNZs: 15, Bias: -1.229563, T: 337500, Avg. loss: 0.381703
Total training time: 0.13 seconds.
-- Epoch 10
Norm: 1.90, NNZs: 15, Bias: -1.251245, T: 375000, Avg. loss: 0.380763
Total training time: 0.15 seconds.
-- Epoch 11
Norm: 1.94, NNZs: 15, Bias: -1.269044, T: 412500, Avg. loss: 0.380084
Total training time: 0.16 seconds.
-- Epoch 12
Norm: 1.98, NNZs: 15, Bias: -1.282485, T: 450000, Avg. loss: 0.379607
Total training time: 0.19 seconds.
-- Epoch 13
Norm: 2.01, NNZs: 15, Bias: -1.294386, T: 487500, Avg. loss: 0.379251
Total training time: 0.21 seconds.
-- Epoch 14
Norm: 2.03, NNZs: 15, Bias: -1.305805, T: 525000, Avg. loss: 0.378992
Total training time: 0.21 seconds.
Convergence after 14 epochs took 0.21 seconds
```

Out[8]:
```
SGDClassifier(eta0=0.0001, learning_rate='constant', loss='log',
              random_state=15, verbose=2)
```

In [9]:
```
clf.coef_, clf.coef_.shape, clf.intercept_
#clf.coef_ will return the weights
#clf.coef_.shape will return the shape of weights
#clf.intercept_ will return the intercept term
```

Out[9]:
```
(array([[-0.89007184,  0.63162363, -0.07594145,  0.63107107, -0.38434375,
          0.93235243, -0.89573521, -0.07340522,  0.40591417,  0.4199991 ,
          0.24722143,  0.05046199, -0.08877987,  0.54081652,  0.06643888]]),
 (1, 15),
 array([-1.30580538]))
```

```
# This is formatted as code
```

Implement Logistic Regression with L2 regularization Using SGD: without using sklearn

1. We will be giving you some functions, please write code in that functions only.
2. After every function, we will be giving you expected output, please make sure that you get that output.

- Initialize the weight_vector and intercept term to zeros (Write your code in def initialize_weights())
- Create a loss function (Write your code in def logloss())

  $logloss = -1 * \frac{1}{n}\Sigma_{foreachYt,Y_{pred}}(Ytlog10(Y_{pred}) + (1-Yt)log10(1-Y_{pred}))$

- for each epoch:
  - for each batch of data points in train: (keep batch size=1)
    - calculate the gradient of loss function w.r.t each weight in weight vector (write your code in def gradient_dw())

      $dw^{(t)} = x_n(y_n - \sigma((w^{(t)})^T x_n + b^t)) - \frac{\lambda}{N}w^{(t)})$

    - Calculate the gradient of the intercept (write your code in def gradient_db()) check this

      $db^{(t)} = y_n - \sigma((w^{(t)})^T x_n + b^t))$

    - Update weights and intercept (check the equation number 32 in the above mentioned pdf):
      $w^{(t+1)} \leftarrow w^{(t)} + \alpha(dw^{(t)})$

      $b^{(t+1)} \leftarrow b^{(t)} + \alpha(db^{(t)})$

  - calculate the log loss for train and test with the updated weights (you can check the python assignment 10th question)
  - And if you wish, you can compare the previous loss and the current loss, if it is not updating, then you can stop the training
  - append this loss in the list (this will be used to see how loss is changing for each epoch after the training is over)

Initialize weights

In [10]:
```
def initialize_weights(dim):
    ''' In this function, we will initialize our weights and bias'''
    #initialize the weights to zeros array of (1,dim) dimensions
    #you use zeros_like function to initialize zero, check this link https://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros_like.html
    #initialize bias to zero

    w = np.zeros_like(dim)
    b = 0

    return w,b
```

In [11]:
```
dim=X_train[0]
w,b = initialize_weights(dim)
print('w =',(w))
print('b =',str(b))
```

```
w = [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
b = 0
```

Grader function - 1

In [12]:
```
dim=X_train[0]
w,b = initialize_weights(dim)
def grader_weights(w,b):
  assert((len(w)==len(dim)) and b==0 and np.sum(w)==0.0)
  return True
grader_weights(w,b)
```

Out[12]:
```
True
```

Compute sigmoid

$sigmoid(z) = 1/(1+exp(-z))$

In [13]:
```
def sigmoid(z):
    ''' In this function, we will return sigmoid of z'''
    # compute sigmoid(z) and return

    sigmoid_of_Z = 1 / (1 + np.exp(-z))

    return sigmoid_of_Z
```

Grader function - 2

In [14]:
```
def grader_sigmoid(z):
  val=sigmoid(z)
  assert(val==0.8807970779778823)
  return True
grader_sigmoid(2)
```

Out[14]:
```
True
```

Compute loss

$logloss = -1 * \frac{1}{n}\Sigma_{foreachYt,Y_{pred}}(Ytlog10(Y_{pred}) + (1-Yt)log10(1-Y_{pred}))$

In [15]:
```
import math
def logloss(y_true,y_pred):
    '''In this function, we will compute log loss '''
    loss = 0
    n = len(y_true)
    if(len(y_true) == len(y_pred)):
        for l in range(len(y_true)):

            loss += (y_true[l] * math.log10(y_pred[l])) + ((1-y_true[l]) * math.log10(1-y_pred[l]))

        loss = (-1 / n) * loss

    return loss
```

Grader function - 3

In [16]:
```
def grader_logloss(true,pred):
  loss=logloss(true,pred)
  assert(loss==0.07644900402910389)
  return True
true=[1,1,0,1,0]
pred=[0.9,0.8,0.1,0.8,0.2]
grader_logloss(true,pred)
```

Out[16]:
```
True
```

Compute gradient w.r.to 'w'

$dw^{(t)} = x_n(y_n - \sigma((w^{(t)})^T x_n + b^t)) - \frac{\lambda}{N}w^{(t)}$

In [17]:
```
def gradient_dw(x,y,w,b,alpha,N):
    '''In this function, we will compute the gardient w.r.to w '''

    dw = ((x * ((y - sigmoid(np.dot(w.T,x) + b))) - ((alpha / N) * w)))

    return dw
```

Grader function - 4

In [18]:
```
def grader_dw(x,y,w,b,alpha,N):
  grad_dw=gradient_dw(x,y,w,b,alpha,N)
  assert(np.sum(grad_dw)==2.613689585)
  return True
grad_x=np.array([-2.07864835,  3.31604252, -0.79104357, -3.87045546, -1.14783286,
       -2.81434437, -0.86771071, -0.04073287,  0.84827878,  1.99451725,
        3.67152472,  0.01451875,  2.01062888,  0.07373904, -5.54586092])
grad_y=0
grad_w,grad_b=initialize_weights(grad_x)
alpha=0.0001
N=len(X_train)
grader_dw(grad_x,grad_y,grad_w,grad_b,alpha,N)
```

Out[18]:
```
True
```

Compute gradient w.r.to 'b'

$db^{(t)} = y_n - \sigma((w^{(t)})^T x_n + b^t)$

In [19]:
```
def gradient_db(x,y,w,b):
    '''In this function, we will compute gradient w.r.to b '''

    db = y - sigmoid(np.dot(w.T,x) + b)

    return db
```

Grader function - 5

In [20]:
```
def grader_db(x,y,w,b):
  grad_db=gradient_db(x,y,w,b)
  assert(grad_db==-0.5)
  return True
grad_x=np.array([-2.07864835,  3.31604252, -0.79104357, -3.87045546, -1.14783286,
       -2.81434437, -0.86771071, -0.04073287,  0.84827878,  1.99451725,
        3.67152472,  0.01451875,  2.01062888,  0.07373904, -5.54586092])
grad_y=0
grad_w,grad_b=initialize_weights(grad_x)
alpha=0.0001
N=len(X_train)
grader_db(grad_x,grad_y,grad_w,grad_b)
```

Out[20]:
```
True
```

Implementing logistic regression

In [21]:
```
def train(X_train,y_train,X_test,y_test,epochs,alpha,eta0):
    ''' In this function, we will implement logistic regression'''
    #Here eta0 is learning rate
    #implement the code as follows
    # initalize the weights (call the initialize_weights(X_train[0]) function)
    w,b = initialize_weights(X_train[0])

    trainingLossList,testingLossList = [],[]

    N = len(X_train)
    # for every epoch
    for epoch in range(0,epochs):

        for datapoint in range(len(X_train)):
        # for every data point(X_train,y_train)
           #compute gradient w.r.to w (call the gradient_dw() function)
            dw = gradient_dw(X_train[datapoint],y_train[datapoint],w,b,alpha,N)
           #compute gradient w.r.to b (call the gradient_db() function)
            db = gradient_db(X_train[datapoint],y_train[datapoint],w,b)
           #update  the weigth and bias w, b

            w = w  + eta0 * dw

            b = b + eta0 * db
        # predict the output of x_train[for all data points in X_train] using w,b
        prediction_training  = [sigmoid(np.dot(w,X_train[data]) + b) for data in range(len(X_train))]
        #compute the loss between predicted and actual values (call the loss function)
        training_loss = logloss(y_train,prediction_training)
        # store all the train loss values in a list
        trainingLossList.append(training_loss)
        # predict the output of x_test[for all data points in X_test] using w,b
        prediction_test = [sigmoid(np.dot(w,X_test[data]) + b) for data in range(len(X_test))]
        #compute the loss between predicted and actual values (call the loss function)
        test_loss = logloss(y_test,prediction_test)
        # store all the test loss values in a list
        testingLossList.append(test_loss)
        # you can also compare previous loss and current loss, if loss is not updating then stop the process and return w,b
        if (len(trainingLossList) > 1) and (len(testingLossList) > 1) :
            if (trainingLossList[epoch] == trainingLossList[epoch - 1]) and \
            (testingLossList[epoch] == testingLossList[epoch - 1]):
                print("No Improvement in Model")
                break
        print('Epoch--{0}'.format(epoch + 1) )

        print("Training Loss  : {0}  --  Testing Loss {1}".format(trainingLossList[epoch],testingLossList[epoch]))

    return w,b,trainingLossList,testingLossList
```

In [22]:
```
alpha=0.0001
eta0=0.0001
N=len(X_train)
epochs=15
w,b,trainingLossList,testingLossList=train(X_train,y_train,X_test,y_test,epochs,alpha,eta0)
```

```
Epoch--1
Training Loss  : 0.2072978178414084  --  Testing Loss 0.20722219781181883
Epoch--2
Training Loss  : 0.18556210141426163  --  Testing Loss 0.18565259434678277
Epoch--3
Training Loss  : 0.17659652085620509  --  Testing Loss 0.17682567720849304
Epoch--4
Training Loss  : 0.17201289496451902  --  Testing Loss 0.1723532484818957
Epoch--5
Training Loss  : 0.16938000886115878  --  Testing Loss 0.1698100984080047
Epoch--6
Training Loss  : 0.16775336575455  --  Testing Loss 0.16825663498220053
Epoch--7
Training Loss  : 0.16669776297615663  --  Testing Loss 0.16726128890692277
Epoch--8
Training Loss  : 0.16598837500432867  --  Testing Loss 0.16660192986644845
Epoch--9
Training Loss  : 0.1654991822760498  --  Testing Loss 0.1661545712175774
Epoch--10
Training Loss  : 0.16515513945496196  --  Testing Loss 0.16584572669386238
Epoch--11
Training Loss  : 0.16490944296095897  --  Testing Loss 0.16562980540333389
Epoch--12
Training Loss  : 0.1647318311394912  --  Testing Loss 0.16547750036682654
Epoch--13
Training Loss  : 0.16460216964645405  --  Testing Loss 0.16536943679761335
Epoch--14
Training Loss  : 0.16450674989278702  --  Testing Loss 0.16529251625482547
Epoch--15
Training Loss  : 0.16443606134127775  --  Testing Loss 0.1652377226863054
```

Goal of assignment

Compare your implementation and SGDClassifier's the weights and intercept, make sure they are as close as possible i.e difference should be in terms of 10^-3

In [23]:
```
# these are the results we got after we implemented sgd and found the optimal weights and intercept
w-clf.coef_, b-clf.intercept_
```

Out[23]:
```
(array([[-0.01475437,  0.01493816, -0.00227909,  0.00670554, -0.00641395,
          0.01189447, -0.00725785,  0.00180162,  0.00978705,  0.00360839,
          0.00457525,  0.00349702,  0.00054823,  0.00292799,  0.00056488]]),
 array([-0.00611698]))
```

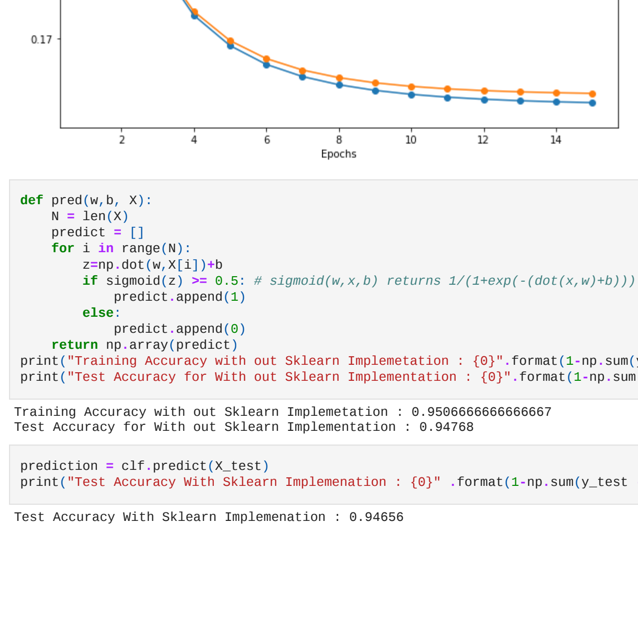
In [24]:
```
print("SDG Implementation with Sklearn")
print("Weight :{0}".format(clf.coef_))

print('Intercept : {0}'.format(clf.intercept_))
print("=="* 50)
print("SDG Implementation Without Sklearn")

print("Weight:{0}".format(w))

print('Intercept :{0}'.format(b))

print("=="* 50)
#difference should be in terms of 10^-3
print("Comparision Between Both the Implementation")

print("Weight - difference: {0}".format(w - clf.coef_))

print('Intercept - difference : {0}'.format(b - clf.intercept_))
```

```
SDG Implementation with Sklearn
Weight :[[-0.89007184  0.63162363 -0.07594145  0.63107107 -0.38434375  0.93235243
  -0.89573521 -0.07340522  0.40591417  0.4199991   0.24722143  0.05046199
  -0.08877987  0.54081652  0.06643888]]
Intercept : [-1.30580538]
====================================================================================================
SDG Implementation Without Sklearn
Weight:[-0.9048262   0.64656179 -0.07822054  0.63777662 -0.3907577   0.9442469
 -0.90299306 -0.07160359  0.41570122  0.42360749  0.25179668  0.05395901
 -0.08823163  0.54374451  0.06700376]
Intercept :-1.3119223645129903
====================================================================================================
Comparision Between Both the Implementation
Weight - difference: [[-0.01475437  0.01493816 -0.00227909  0.00670554 -0.00641395  0.01189447
  -0.00725785  0.00180162  0.00978705  0.00360839  0.00457525  0.00349702
   0.00054823  0.00292799  0.00056488]]
Intercept - difference : [-0.00611698]
```

Plot epoch number vs train , test loss

- epoch number on X-axis
- loss on Y-axis

In [25]:
```
import matplotlib.pyplot as plt
plt.figure(figsize = (10,10))
plt.plot(range(1,epochs+1),trainingLossList)
plt.plot(range(1,epochs+1),testingLossList)
plt.scatter(range(1,epochs+1),trainingLossList)
plt.scatter(range(1,epochs+1),testingLossList)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training Loss and Testing Loss")
labels = ["train Loss" , "test Loss"]
plt.legend(labels, loc = "upper right")
plt.show()
```



In [26]:
```
def pred(w,b, X):
    N = len(X)
    predict = []
    for i in range(N):
        z=np.dot(w,X[i])+b
        if sigmoid(z) >= 0.5: # sigmoid(w,x,b) returns 1/(1+exp(-(dot(x,w)+b)))
            predict.append(1)
        else:
            predict.append(0)
    return np.array(predict)
print("Training Accuracy with out Sklearn Implemetation : {0}".format(1-np.sum(y_train - pred(w,b,X_train))/len(X_train)))
print("Test Accuracy for With out Sklearn Implementation : {0}".format(1-np.sum(y_test  - pred(w,b,X_test))/len(X_test)))
```

```
Training Accuracy with out Sklearn Implemetation : 0.9506666666666667
Test Accuracy for With out Sklearn Implementation : 0.94768
```

In [27]:
```
prediction = clf.predict(X_test)
print("Test Accuracy With Sklearn Implemenation : {0}" .format(1-np.sum(y_test - prediction)/len(X_test)))
```

```
Test Accuracy With Sklearn Implemenation : 0.94656
```