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
%matplotlib inline
import matplotlib.pyplot as plt
```

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.dataset
s.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
# you need not standardize the data as it is already standardized
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=15)
```

In [5]:
```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

Out[5]:
```
((37500, 15), (37500,), (12500, 15), (12500,))
```

In [6]:
```
X_train[1]
```

Out[6]:
```
array([ 1.827818  , -0.45810992,  0.47407375, -2.17856544, -1.16453085,
       -0.59906384,  2.24400146,  0.2664526 , -1.59252721, -2.3705834 ,
       -1.14068014, -1.83108915, -0.32123197,  0.31287131, -1.494433  ])
```

In [7]:
```
y_train[0:50]
```

Out[7]:
```
array([0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1,
       1, 0, 0, 0, 0, 0])
```

SGD classifier

In [8]:
```
# 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, pen
alty='l2', tol=1e-3, verbose=2, learning_rate='constant')
clf
# Please check this documentation (https://scikit-learn.org/stable/modules/generated/sklear
n.linear_model.SGDClassifier.html)
```

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

In [9]:
```
clf.fit(X=X_train, y=y_train) # fitting our model
```

```
-- Epoch 1
Norm: 0.77, NNZs: 15, Bias: -0.316653, T: 37500, Avg. loss: 0.455552
Total training time: 0.02 seconds.
-- Epoch 2
Norm: 0.91, NNZs: 15, Bias: -0.472747, T: 75000, Avg. loss: 0.394686
Total training time: 0.03 seconds.
-- Epoch 3
Norm: 0.98, NNZs: 15, Bias: -0.580082, T: 112500, Avg. loss: 0.385711
Total training time: 0.05 seconds.
-- Epoch 4
Norm: 1.02, NNZs: 15, Bias: -0.658292, T: 150000, Avg. loss: 0.382083
Total training time: 0.06 seconds.
-- Epoch 5
Norm: 1.04, NNZs: 15, Bias: -0.719528, T: 187500, Avg. loss: 0.380486
Total training time: 0.08 seconds.
-- Epoch 6
Norm: 1.05, NNZs: 15, Bias: -0.763409, T: 225000, Avg. loss: 0.379578
Total training time: 0.09 seconds.
-- Epoch 7
Norm: 1.06, NNZs: 15, Bias: -0.795106, T: 262500, Avg. loss: 0.379150
Total training time: 0.11 seconds.
-- Epoch 8
Norm: 1.06, NNZs: 15, Bias: -0.819925, T: 300000, Avg. loss: 0.378856
Total training time: 0.13 seconds.
-- Epoch 9
Norm: 1.07, NNZs: 15, Bias: -0.837805, T: 337500, Avg. loss: 0.378585
Total training time: 0.15 seconds.
-- Epoch 10
Norm: 1.08, NNZs: 15, Bias: -0.853138, T: 375000, Avg. loss: 0.378630
Total training time: 0.17 seconds.
Convergence after 10 epochs took 0.17 seconds
```

Out[9]:
```
SGDClassifier(eta0=0.0001, learning_rate='constant', loss='log',
              random_state=15, verbose=2)
```

In [10]:
```
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[10]:
```
(array([[-0.42336692,  0.18547565, -0.14859036,  0.34144407, -0.2081867 ,
          0.56016579, -0.45242483, -0.09408813,  0.2092732 ,  0.18084126,
          0.19705191,  0.00421916, -0.0796037 ,  0.33852802,  0.02266721]]),
 (1, 15),
 array([-0.8531383]))
```

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())

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

Initialize weights

In [11]:
```
def initialize_weights(row_vector):
    wt=np.zeros_like(row_vector)
    bias=0
    ''' In this function, we will initialize our weights and bias'''
    #initialize the weights as 1d array consisting of all zeros similar to the dimensions of
row_vector
    #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
    return wt,bias
```

In [12]:
```
dim=X_train[0]
w,b = initialize_weights(X_train[0])
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 [13]:
```
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[13]:
```
True
```

Compute sigmoid

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

In [14]:
```
def sigmoid(z):
    ''' In this function, we will return sigmoid of z'''
    # compute sigmoid(z) and return
    return  1/(1+np.exp(-z))
```

Grader function - 2

In [15]:
```
def grader_sigmoid(z):
    val=sigmoid(z)
    assert(val==0.8807970779778823)
    return True

grader_sigmoid(2)
```

Out[15]:
```
True
```

Compute loss

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

In [17]:
```
def logloss(y_true,y_pred):
    # you have been given two arrays y_true and y_pred and you have to calculate the logloss
    #while dealing with numpy arrays you can use vectorized operations for quicker calculati
ons as compared to using loops
    #https://www.pythonlikeyoumeanit.com/Module3_IntroducingNumpy/VectorizedOperations.html
    #https://www.geeksforgeeks.org/vectorized-operations-in-numpy/
    #write your code here
    sumtn = 0
    for i in range(len(y_true)):
        sumtn += (y_true[i] * np.log10(y_pred[i])) + ((1 - y_true[i]) * np.log10(1 - y_pred[
i]))
    loss = -1 * (1 / len(y_true)) * sumtn

    return loss
```

Grader function - 3

In [18]:
```
#round off the value to 8 values
def grader_logloss(true,pred):
    loss=logloss(true,pred)
    assert(np.round(loss,6)==0.076449)
    return True
true=np.array([1,1,0,1,0])
pred=np.array([0.9,0.8,0.1,0.8,0.2])
grader_logloss(true,pred)
```

Out[18]:
```
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 [20]:
```
#make sure that the sigmoid function returns a scalar value, you can use dot function operat
ion
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,x) + b) - (alpha / N) * w)
    return dw
```

Grader function - 4

In [21]:
```
def grader_dw(x,y,w,b,alpha,N):
    grad_dw=gradient_dw(x,y,w,b,alpha,N)
    assert(np.round(np.sum(grad_dw),5)==4.75684)
    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=np.array([ 0.03364887,  0.03612727,  0.02786927,  0.08547455, -0.12870234,
       -0.02555288,  0.11858013,  0.13305576,  0.07310204,  0.15149245,
       -0.05708987, -0.064768  ,  0.18012332, -0.16880843, -0.27079877])
grad_b=0.5
alpha=0.0001
N=len(X_train)
grader_dw(grad_x,grad_y,grad_w,grad_b,alpha,N)
```

Out[21]:
```
True
```

Compute gradient w.r.to 'b'

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

In [22]:
```
#sb should be a scalar value
def gradient_db(x,y,w,b):
    '''In this function, we will compute gradient w.r.to b '''
    db = y - sigmoid(np.dot(w,x) + b)
    return db
```

Grader function - 5

In [23]:
```
def grader_db(x,y,w,b):
  grad_db=gradient_db(x,y,w,b)
  assert(np.round(grad_db,4)==-0.3714)
  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=0.5
grad_b=0.1
grad_w=np.array([ 0.03364887,  0.03612727,  0.02786927,  0.08547455, -0.12870234,
       -0.02555288,  0.11858013,  0.13305576,  0.07310204,  0.15149245,
       -0.05708987, -0.064768  ,  0.18012332, -0.16880843, -0.27079877])
alpha=0.0001
N=len(X_train)
grader_db(grad_x,grad_y,grad_w,grad_b)
```

Out[23]:
```
True
```

In [25]:
```
# prediction function used to compute predicted_y given the dataset X
def pred(w,b, X):
    L = len(X)
    predict = []
    for i in range(L):
        z=np.dot(w,X[i])+b
        predict.append(sigmoid(z))
    return np.array(predict)
```

Implementing logistic regression

In [26]:
```
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)
    # for every epoch
        # for every data point(X_train,y_train)
           #compute gradient w.r.to w (call the gradient_dw() function)
           #compute gradient w.r.to b (call the gradient_db() function)
           #update w, b
        # predict the output of x_train [for all data points in X_train] using pred function
with updated weights
        #compute the loss between predicted and actual values (call the loss function)
        # store all the train loss values in a list
        # predict the output of x_test [for all data points in X_test] using pred function w
ith updated weights
        #compute the loss between predicted and actual values (call the loss function)
        # store all the test loss values in a list
        # you can also compare previous loss and current loss, if loss is not updating then
 stop the process
        # you have to return w,b , train_loss and test loss

    train_loss = []
    test_loss = []
    w,b = initialize_weights(X_train[0]) # Initialize the weights
    #write your code to perform SGD
    for i in range(epochs):
        train_pred = []
        test_pred = []
        for j in range(N):
            dw = gradient_dw(X_train[j],y_train[j],w,b,alpha,N)
            db = gradient_db(X_train[j],y_train[j],w,b)
            w = w + (eta0 * dw)
            b = b + (eta0 * db)
        for v in range(N):
            train_pred.append(sigmoid(np.dot(w, X_train[v]) + b))

        loss_n1 = logloss(y_train, train_pred)
        train_loss.append(loss_n1)

        for val in range(len(X_test)):
            test_pred.append(sigmoid(np.dot(w, X_test[val]) + b))

        loss_n2 = logloss(y_test, test_pred)
        test_loss.append(loss_n2)
```

In [27]:
```
alpha=0.001
eta0=0.0001
N=len(X_train)
epochs=20
w,b,train_log_loss,test_log_loss=train(X_train,y_train,X_test,y_test,epochs,alpha,eta0)
```

In [28]:
```
#print thr value of weights w and bias b
print(w)
print(b)
```

```
[-4.29394714e-01  1.92911498e-01 -1.48319119e-01  3.38095882e-01
 -2.20731285e-01  5.69669899e-01 -4.45186044e-01 -9.00097324e-02
  2.21598165e-01  1.73588026e-01  1.98538426e-01 -4.13068052e-04
 -8.11249261e-02  3.39070628e-01  2.29368798e-02]
-0.8897519393750316
```

In [29]:
```
# these are the results we got after we implemented sgd and found the optimal weights and in
tercept

w-clf.coef_, b-clf.intercept_
```

Out[29]:
```
(array([[-0.0060278 ,  0.00743585,  0.00027124, -0.00334819, -0.01254458,
          0.00950411,  0.00723878,  0.0040784 ,  0.01232497, -0.00725323,
          0.00148652, -0.00463223, -0.00152123,  0.00054261,  0.00026967]]),
 array([-0.03661364]))
```

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 order of 10^-2

Grader function - 6

In [30]:
```
#this grader function should return True
#the difference between custom weights and clf.coef_ should be less than or equal to 0.05
def difference_check_grader(w,b,coef,intercept):
    val_array=np.abs(np.array(w-coef))
    assert(np.all(val_array<=0.05))
    print('The custom weights are correct')
    return True
difference_check_grader(w,b,clf.coef_,clf.intercept_)
```

```
The custom weights are correct
```

Out[30]:
```
True
```

Plot your train and test loss vs epochs

plot epoch number on X-axis and loss on Y-axis and make sure that the curve is converging

In [31]:
```
epochs = [i for i in range(1,21,1)]
plt.figure(figsize=(8,6))
plt.grid()
plt.plot(epochs,train_log_loss,label='train_loss')
plt.plot(epochs,test_log_loss,label='test_loss')
plt.xlabel('epoch number')
plt.ylabel('train and test loss')
plt.legend()
```

Out[31]:
```
<matplotlib.legend.Legend at 0x19958bac0b8>
```