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_re
                           n_classes=2, weights=[0.7], class_sep=0.7, random_stat
# make_classification is used to create custom dataset
# Please check this link (https://scikit-learn.org/stable/modules/generated/sklea
```

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_
```

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

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

SGD classifier

In [6]:
```
# alpha : float
# Constant that multiplies the regularization term.

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

clf = linear_model.SGDClassifier(eta0=0.0001, alpha=0.0001, loss='log', random_st
clf
# Please check this documentation (https://scikit-learn.org/stable/modules/genera
```

Out[6]:
```
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0001, fit_intercept=True, l1_ratio=0.15,
       learning_rate='constant', loss='log', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=15, shuffle=True,
       tol=0.001, verbose=2, warm_start=False)
```

In [7]:
```
clf.fit(X=X_train, y=y_train) # fitting our model
```

```
-- Epoch 1
Norm: 0.76, NNZs: 15, Bias: -0.314605, T: 37500, Avg. loss: 0.455801
Total training time: 0.00 seconds.
-- Epoch 2
Norm: 0.92, NNZs: 15, Bias: -0.469578, T: 75000, Avg. loss: 0.394737
Total training time: 0.01 seconds.
-- Epoch 3
Norm: 0.98, NNZs: 15, Bias: -0.580452, T: 112500, Avg. loss: 0.385561
Total training time: 0.01 seconds.
-- Epoch 4
Norm: 1.02, NNZs: 15, Bias: -0.660824, T: 150000, Avg. loss: 0.382161
Total training time: 0.02 seconds.
-- Epoch 5
Norm: 1.04, NNZs: 15, Bias: -0.717218, T: 187500, Avg. loss: 0.380474
Total training time: 0.03 seconds.
-- Epoch 6
Norm: 1.06, NNZs: 15, Bias: -0.761816, T: 225000, Avg. loss: 0.379481
Total training time: 0.03 seconds.
Convergence after 6 epochs took 0.03 seconds
```

Out[7]:
```
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0001, fit_intercept=True, l1_ratio=0.15,
       learning_rate='constant', loss='log', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=15, shuffle=True,
       tol=0.001, verbose=2, warm_start=False)
```

In [8]:
```
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[8]:
```
(array([[-0.41177431,  0.18416782, -0.13895073,  0.33572511, -0.18423237,
          0.5494352 , -0.45213692, -0.08857465,  0.21536661,  0.17351757,
          0.18480827,  0.00443463, -0.07033001,  0.33683181,  0.02004129]]),
 (1, 15),
 array([-0.76181561]))
```

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 (https://drive.google.com/file/d/1nQ08-XY4zvOLzRX-lGf8EYB5arb7-m1H/view?usp=sharing)

      $$db^{(t)} = y_n - \sigma((w^{(t)})^T x_n + b^t))$$
    - Update weights and intercept (check the equation number 32 in the above mentioned pdf (https://drive.google.com/file/d/1nQ08-XY4zvOLzRX-lGf8EYB5arb7-m1H/view?usp=sharing)):

      $$w^{(t+1)} \leftarrow w^{(t)} + \alpha(dw^{(t)})$$

      $$b^{(t+1)} \leftarrow b^{(t)} + \alpha(db^{(t)})$$
  - 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 [9]:
```
def initialize_weights(row_vector):
    ''' In this function, we will initialize our weights and bias'''
    #initialize the weights as 1d array consisting of all zeros similar to the di
    #you use zeros_like function to initialize zero, check this link https://docs
    #initialize bias to zero
    w = np.zeros_like(row_vector)
    b = 0
    return w,b
```

In [10]:
```
row_vector=X_train[0]
w,b = initialize_weights(row_vector)
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 [11]:
```
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[11]:
```
True
```

Compute sigmoid

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

In [12]:
```
import math
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 [13]:
```
def grader_sigmoid(z):
  val=sigmoid(z)
  assert(val==0.8807970779778823)
  return True
grader_sigmoid(2)
```

Out[13]: True

Compute loss

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

In [14]:
```
def logloss(y_true,y_pred):
    # you have been given two arrays y_true and y_pred and you have to calculate
    #while dealing with numpy arrays you can use vectorized operations for quicke
    #https://www.pythonlikeyoumeanit.com/Module3_IntroducingNumpy/VectorizedOpera
    #https://www.geeksforgeeks.org/vectorized-operations-in-numpy/
    #write your code here
    sigma_part = 0
    for i in range(len(y_true)):
        sigma_part += (y_true[i] * np.log10(y_pred[i])) + ((1 - y_true[i]) * np.l
    loss = -1 * (1 / len(y_true)) * sigma_part
    return loss
```

Grader function - 3

In [15]:
```
#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[15]: 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 [16]:
```

#make sure that the sigmoid function returns a scalar value, you can use dot func
def gradient_dw(x,y,w,b,alpha,N):
    '''In this function, we will compute the gardient w.r.to w '''
    return x * (y-sigmoid(np.dot(w,x)+b)-(alpha/N)*w)
```

Grader function - 4

In [17]:
```
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[17]: True

Compute gradient w.r.to 'b'

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

In [18]:
```
#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 [19]:
```
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[19]: True

In [20]:
```
# prediction function used to compute predicted_y given the dataset X
def pred(w,b, X):
    N = len(X)
    predict = []
    for i in range(N):
        z=np.dot(w,X[i])+b
        predict.append(sigmoid(z))
    return np.array(predict)
```

Implementing logistic regression

In [21]:
```
from tqdm import tqdm
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
    train_loss = []
    test_loss = []
    w, b = initialize_weights(X_train[0])
    for i in tqdm(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)
        train_pred=pred(w,b, X_train)
        loss1 = logloss(y_train, train_pred)
        train_loss.append(loss1)
        test_pred=pred(w,b, X_test)
        loss2 = logloss(y_test, test_pred)
        test_loss.append(loss2)
    return w,b,train_loss,test_loss
```

In [22]:
```
alpha=0.001
eta0=0.001
N=len(X_train)
epochs=20
w,b,train_loss,test_loss=train(X_train,y_train,X_test,y_test,epochs,alpha,eta0)
```

```
100%|██████████████████████████████████████████████████
████████| 20/20 [00:24<00:00,  1.24s/it]
```

In [23]:
```
#print thr value of weights w and bias b
print(w)
print(b)
```

```
[-0.41395269  0.19245258 -0.15005108  0.32635385 -0.22516783  0.58646754
 -0.42720461 -0.10027814  0.21483871  0.15555206  0.1788105  -0.01318643
 -0.06496816  0.36313959 -0.00985043]
-0.9016736323411028
```

In [24]:
```
# these are the results we got after we implemented sgd and found the optimal wei

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

Out[24]:
```
(array([[-0.00217837,  0.00828477, -0.01110035, -0.00937127, -0.04093546,
          0.03703234,  0.02493231, -0.01170349, -0.0005279 , -0.01796552,
         -0.00599777, -0.01762105,  0.00536185,  0.02630778, -0.02989172]]),
 array([-0.13985803]))
```

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 [25]:
```
#this grader function should return True
#the difference between custom weights and clf.coef_ should be less than or equal
def differece_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
differece_check_grader(w,b,clf.coef_,clf.intercept_)
```

```
The custom weights are correct
```

Out[25]:
```
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 [39]:
```
from matplotlib import pyplot as plt
from matplotlib.pyplot import figure
import math
plt.plot(range(epochs),(train_loss) , label='train log loss')
plt.plot(range(epochs),test_loss, label='test log loss')
plt.xlabel("epoch number")
plt.ylabel("log loss")
plt.legend()
plt.show
```

Out[39]: <function matplotlib.pyplot.show(*args, **kw)>