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

IMPORTING THE NECESSARY LIBRARIES

```
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

```
# 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
```

```
X.shape, y.shape
```

```
((50000, 15), (50000,))
```

Splitting data into train and test

```
#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)
```

CHECKING THE SHAPE

```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

```
((37500, 15), (37500,), (12500, 15), (12500,))
```

SGD classifier

```
# 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)
```

```
SGDClassifier(eta0=0.0001, learning_rate='constant', loss='log',
              random_state=15, verbose=2)
```

```
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: 0.01 seconds.
-- Epoch 2
Norm: 0.91, NNZs: 15, Bias: -0.472747, T: 75000, Avg. loss: 0.394686
Total training time: 0.02 seconds.
-- Epoch 3
Norm: 0.98, NNZs: 15, Bias: -0.580082, T: 112500, Avg. loss: 0.385711
Total training time: 0.04 seconds.
-- Epoch 4
Norm: 1.02, NNZs: 15, Bias: -0.658292, T: 150000, Avg. loss: 0.382083
Total training time: 0.05 seconds.
-- Epoch 5
Norm: 1.04, NNZs: 15, Bias: -0.719528, T: 187500, Avg. loss: 0.380486
Total training time: 0.06 seconds.
-- Epoch 6
Norm: 1.05, NNZs: 15, Bias: -0.763409, T: 225000, Avg. loss: 0.379578
Total training time: 0.07 seconds.
-- Epoch 7
Norm: 1.06, NNZs: 15, Bias: -0.795106, T: 262500, Avg. loss: 0.379150
Total training time: 0.08 seconds.
-- Epoch 8
Norm: 1.06, NNZs: 15, Bias: -0.819925, T: 300000, Avg. loss: 0.378856
Total training time: 0.10 seconds.
-- Epoch 9
Norm: 1.07, NNZs: 15, Bias: -0.837805, T: 337500, Avg. loss: 0.378585
Total training time: 0.11 seconds.
-- Epoch 10
Norm: 1.08, NNZs: 15, Bias: -0.853138, T: 375000, Avg. loss: 0.378630
Total training time: 0.12 seconds.
Convergence after 10 epochs took 0.12 seconds
SGDClassifier(eta0=0.0001, learning_rate='constant', loss='log',
              random_state=15, verbose=2)
```

```
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
```

```
(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())

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

```
def initialize_weights(row_vector):
   w = np.zeros_like(X_train[0])
   b=0
   return w,b
```

INITILIZIN THE WEIGHT AND BISES

```
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

```
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)
```

```
True
```

Compute sigmoid

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

```
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

```
def grader_sigmoid(z):
  val=sigmoid(z)
  assert(val==0.8807970779778823)
  return True
grader_sigmoid(2)
```

```
True
```

Compute loss

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

```
def logloss(y_true,y_pred):
    '''In this function, we will compute log loss '''
    sum = 0
    for i in range(len(y_true)):
      # HERE COMPUTING THE LOG LOSS AS PE FORMULA GIVEN ABOVE
        sum += (y_true[i] * np.log10(y_pred[i])) + ((1 - y_true[i]) * np.log10(1 - y_pred[i]))
    loss = -1 * (1 / len(y_true)) * sum
    return loss
```

Grader function - 3

```
#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)
```

```
True
```

Compute gradient w.r.to 'w'

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

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

  return dw
```

Grader function - 4

```
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)
```

```
True
```

Compute gradient w.r.to 'b'

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

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

  return db
```

Grader function - 5

```
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)
```

```
True
```

```
# 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

```
def train(X_train,y_train,X_test,y_test,epochs,alpha,eta0):
```

```
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 = []  # TRAIN LOSS
    test_loss = []  # TEST LOSS
    # INITILIZING THE WEIGHTS
    w, b = initialize_weights(X_train[0])
    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 val in range(N):
            train_pred.append(sigmoid(np.dot(w, X_train[val]) + b))

          # COMPUTING THE LOSS BETWEEN ACTUAL VALUE AND PREDICTIVE VALUE
        loss1 = logloss(y_train, train_pred)
        train_loss.append(loss1)

        for val in range(len(X_test)):
          # HERE WILL ADD THE SIGMOID FUNCTION IN PREDICTED VALUE
            test_pred.append(sigmoid(np.dot(w, X_test[val]) + b))

         #  COMPUTING THE LOSS 2 BETWEEN ACTUAL AND PREDCITED VALUE
        loss2 = logloss(y_test, test_pred)
        test_loss.append(loss2)

    return w,b,train_loss,test_loss
```

```
alpha = 0.0001
eta0 = 0.0001
N = len(X_train)
epochs = 50
w, b, train_loss, test_loss = train(X_train, y_train, X_test, y_test, epochs, alpha, eta0)
```

```
#print thr value of weights w and bias b
print(w)
print(b)
```

```
[-0.42979244  0.1930352  -0.14846992  0.33809366 -0.22128236  0.56994894
 -0.44518164 -0.08990399  0.22182949  0.17382965  0.19874847 -0.00058427
 -0.08133409  0.33909012  0.02298795]
-0.8922521679760832
```

```
# these are the results we got after we implemented sgd and found the optimal weights and intercept

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

```
(array([[-0.00642552,  0.00755954,  0.00012044, -0.00335041, -0.01309566,
          0.00978315,  0.00724319,  0.00418414,  0.01255629, -0.00701161,
          0.00169656, -0.00480343, -0.00173039,  0.0005621 ,  0.00032074]]),
 array([-0.03911387]))
```

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

```
#this grader function should return True
#the difference between custom weights and clf.coef_ should be less than or equal to 0.05
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
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

```
from matplotlib import pyplot as plt
epoch = [i for i in range(1,51,1)]
# **ITERATING THE EACH EPOCHS**
plt.plot(epoch,train_loss , label='train_log_loss')
plt.plot(epoch,test_loss, label='test_log_loss')
plt.xlabel("epoch number")
plt.ylabel("log loss")
plt.legend()
plt.show
```

```
<function matplotlib.pyplot.show(*args, **kw)>
```

OBSERVATION

1. HERE WE DRAWN THE PLOT LOG LOSS AGAINST THE NUMBER OF EPOCHS

2. WE CAN SEE THE AFTER THE EPOCHS OF 4 AND 5 LOSS IS NOT REDUCING THE MUCH

3. WE CAN USE THE ELBOW METHOD HERE AND WE CAN CONSIDER THE VALUE OF 5 AS OPTIMAL NUMBER OF EPOCHS

✓ 0s completed at 4:51 PM