

# 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 [2]: 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 [3]: # 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 [4]: X.shape, y.shape

Out[4]: ((50000, 15), (50000,))

Splitting data into train and test
```

```
In [5]: #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 [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

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: 0.77, NMZs: 15, Bias: -0.316653, T: 37500, Avg. loss: 0.455552
Total training time: 0.01 seconds.
-- Epoch 2
Norm: 0.91, NMZs: 15, Bias: -0.472747, T: 75000, Avg. loss: 0.394686
Total training time: 0.02 seconds.
-- Epoch 3
Norm: 0.98, NMZs: 15, Bias: -0.580882, T: 112500, Avg. loss: 0.385711
Total training time: 0.02 seconds.
-- Epoch 4
Norm: 1.02, NMZs: 15, Bias: -0.658292, T: 150000, Avg. loss: 0.382083
Total training time: 0.02 seconds.
-- Epoch 5
Norm: 1.04, NMZs: 15, Bias: -0.719528, T: 187500, Avg. loss: 0.380486
Total training time: 0.02 seconds.
-- Epoch 6
Norm: 1.05, NMZs: 15, Bias: -0.763409, T: 225000, Avg. loss: 0.379578
Total training time: 0.04 seconds.
-- Epoch 7
Norm: 1.05, NMZs: 15, Bias: -0.795106, T: 262500, Avg. loss: 0.379150
Total training time: 0.04 seconds.
-- Epoch 8
Norm: 1.06, NMZs: 15, Bias: -0.819025, T: 300000, Avg. loss: 0.378856
Total training time: 0.06 seconds.
-- Epoch 9
Norm: 1.07, NMZs: 15, Bias: -0.837895, T: 337500, Avg. loss: 0.378595
Total training time: 0.06 seconds.
-- Epoch 10
Norm: 1.08, NMZs: 15, Bias: -0.853138, T: 375000, Avg. loss: 0.378630
Total training time: 0.06 seconds.
Convergence after 10 epochs took 0.06 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([[ -0.42336692,  -0.18547565,  -0.14859936,   0.34144407,  -0.2081867 ,
    -0.56016579,  -0.45242483,  -0.09408813,   0.2092732 ,   0.18084126,
    -0.10705191,  -0.09421916,  -0.0796037 ,   0.33852862,  -0.02260721]]),
 (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()`)  
$$logloss = -1 + \frac{1}{N} \sum_{for each Y_i, Y_{pred}} (Y_i \log 10(Y_{pred}) + (1 - Y_i) \log 10(1 - Y_{pred}))$$
- 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()`)  
$$dw^{(i)} = x_n(y_n - \sigma((w^{(i)})^T x_n + b^{(i)})) - \frac{1}{N} w^{(i)}$$
    - Calculate the gradient of the intercept (write your code in `def gradient_db()`) check this  
$$db^{(i)} = y_n - \sigma((w^{(i)})^T x_n + b^{(i)})$$
    - Update weights and intercept (check the equation number 32 in the above mentioned pdf):  
$$w^{(i+1)} \leftarrow w^{(i)} + \alpha(dw^{(i)})$$
  
$$b^{(i+1)} \leftarrow b^{(i)} + \alpha(db^{(i)})$$
    - 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(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 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
w=np.zeros_like(row_vector)
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]: from math import exp
def sigmoid(z):
''' In this function, we will return sigmoid of z'''
# compute sigmoid(z) and return
return 1/(1+exp(-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 + 1/N * sum_{for each Y_i, Y_pred} (Y_i log10(Y_pred) + (1 - Y_i) log10(1 - Y_pred))
```

```
In [15]: import numpy as np
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 calculations as compared to using loops
#https://www.pythontipkeyoumeanit.com/Module3_IntroducingNumpy/VectorizedOperations.html
#https://www.geeksforgeeks.org/vectorized-operations-in-numpy/
#write your code here
y_true2=np.ones_like(y_true)
y_pred2=np.ones_like(y_pred)
y_true3=np.subtract(y_true2,y_true)
y_pred3=np.subtract(y_pred2,y_pred)
loss=-np.sum(np.add(np.multiply(y_true, np.log10(y_pred)), np.multiply(y_true3 , np.log10(y_pred3))))
return -1*loss/len(y_true)

Grader function - 3
```

```
In [16]: #round off the value to 8 values
def grader_logloss(true,pred):
    loss=logloss(true,pred)
    print(np.round(loss,6))
    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)

0.076449
```

```
Out[16]: True

Compute gradient w.r to w
```

```
dw^{(i)} = x_n(y_n - \sigma((w^{(i)})^T x_n + b^{(i)})) - \frac{1}{N} w^{(i)}
```

```
In [17]: #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):
'''In this function, we will compute the gardient w.r.to w'''
#temp1=np.sum(np.dot(w,T,x),b)
temp=np.dot(w,T,x) + b
si=sigmoid(temp)
temp2=np.subtract(y,si)
temp3=np.multiply(x,temp2)
learning_rate=0.001
temp4=learning_rate/N * w
dw=np.subtract(temp3,temp4)
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.round(np.sum(grad_dw),5)==4.75684)
    return True
grad_x=np.array([-2.07864835, 3.31004252, -0.79104357, -3.87045546, -1.14783286,
-2.81434437, -0.8671071, -0.04073287, 0.84827878, 1.99451725,
3.07152472, 0.01451875, 2.01062888, 0.07373904, -5.54586092])
grad_y=0
grad_w=np.array([ 0.03364807, 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[18]: True

Compute gradient w.r to b
```

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

```
In [19]: #sb should be a scalar value
def gradient_db(x,y,w,b):
'''In this function, we will compute gradient w.r.to b'''
temp=np.dot(w,T,x) + b
si=sigmoid(temp)
db=np.subtract(y,si)
return db

Grader function - 5
```

```
In [20]: 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, 3.31004252, -0.79104357, -3.87045546, -1.14783286,
-2.81434437, -0.8671071, -0.04073287, 0.84827878, 1.99451725,
3.07152472, 0.01451875, 2.01062888, 0.07373904, -5.54586092])
grad_y=0.5
grad_w=np.array([ 0.03364807, 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[20]: True

In [21]: # 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 [22]: 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
# initialize 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 with 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
tempw,tempb=0,0
# for i in range(epochs):
#     for x,y in zip(X_train,y_train):# for every data point(X_train,y_train)
#         dw=gradient_dw(x,y,w,b,alpha,N) #compute gradient w.r.to w (call the gradient_dw() function)
#         db=gradient_db(x,y,w,b) #compute gradient w.r.to b (call the gradient_db() function)
#         w=(alpha*dw) #update w, b
#         b=(alpha*db)
#         predict=pred(w,b,X_train)
#         loss=logloss(y_train,predict)
#         train_loss.append(loss)
#         predict=pred(w,b,X_test)
#         test_loss.append(test_loss)
#         test_loss.append(loss)
#         if len(train_loss)>=2 and len(test_loss)>=2:
#             if (train_loss[-1]-train_loss[-2])<=0.001 and (train_loss[-1]<=train_loss[-2]) and (train_loss[-1]-train_loss[-2])<=0.001 and (train_loss[-1]<=train_loss[-2]):
#                 return w,b,train_loss,test_loss
#             if (train_loss[-1]-train_loss[-2])<=0.001 and (train_loss[-1]<=train_loss[-2]):
#                 return w,b,train_loss,test_loss
#             if len(test_loss)>=2:
#                 if (test_loss[-1]-test_loss[-2])<=0.001 and (test_loss[-1]<=test_loss[-2]):
#                     return w,b,train_loss,test_loss
#                 return w,b,train_loss,test_loss
```

```
In [23]: 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)
```

```
In [24]: #print thr value of weights w and bias b
print(w)
print(b)

[-0.4135543  -0.10231732 -0.14987872  0.32635477 -0.22458419  0.58614342
 -0.42722442  -0.19939941  0.21459638  0.15528446  0.17857722 -0.01297531
 -0.06447708  0.36312146 -0.00988464]
-0.89903726321858
```

```
In [25]: # these are the results we got after we implemented sgd and found the optimal weights and intercept
w,clf.coef_, b=clf.intercept_

Out[25]: (array([[ 0.00981262,  0.00684167,  -0.00128836,  -0.0150893 ,  -0.01639749,
    0.02597763,  0.02520041,  -0.00631128,  0.00532318,  -0.02555568 ,
    -0.01847469,  -0.01719447,  0.01482662,  0.0459345,  -0.03255185]]),
 array([-0.04589897]))
```

### 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<sup>-2</sup>

```
Grader function - 6
```

```
In [26]: #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,clf.coef_,clf.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_)
```

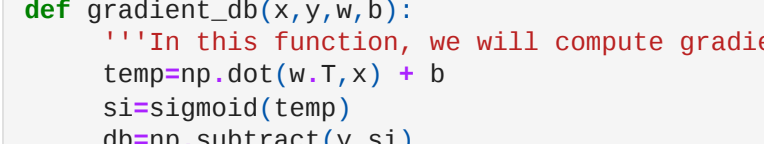
```
The custom weights are correct

Out[26]: 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 [30]: X=[i for i in range(epochs)]
Xx=[len(train_loss)]
y=train_loss
plt=train_loss
import matplotlib.pyplot as plt
plt.plot(X,Y)
y=test_loss
plt.plot(X,y)
plt.xlabel('epoch')
plt.legend(['train_loss','Test_loss'])
plt.title('Epoch vs losses')
plt.show()
```



```
In [ ]:
```