In this lecture we will implement a perceptron which is a single layer neural network. It acts as a linear classifier. We will use binary cross entropy as a loss function and gradient descent optimizer.

Learning Goals ?

- How to implement Perceptron.
- We won't use any for loop, we will see how vectorization in python works.
- what happens when you use non linear dataset and a linear classifier like perceptron.

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
```

Generating Data

```
X, Y = make_blobs ?
```

it returns gaussian blobs

```
X, Y = make_blobs (n_samples = 500, centers = 2, n_features = 2,
                   random_state = 10)
print (X.shape, Y.shape)
```

(500, 2) (500,)

plt.style.use("seaborn");

plt.scatter(x[:,0], x[:,1], c=Y, cmap=plt.cm.Accent)

plt.show()

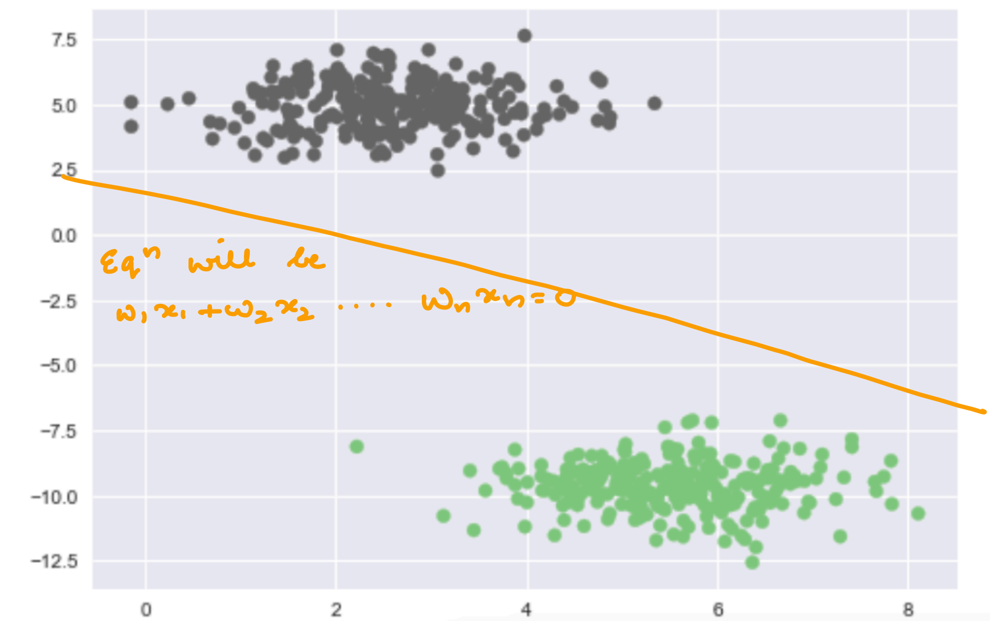
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
```

Generating Data

```
X,Y = make_blobs(n_samples=500, centers=2, n_features=2, random_state=10)
print(X.shape, Y.shape)
```

```
(500, 2) (500,)
```

```
plt.style.use("seaborn")
plt.scatter(X[:,0], X[:,1], c=Y, cmap=plt.cm.Accent)
plt.show()
```



→ it is linearly separable data.

Goal of perceptron learning algo is to figure out 1 boundary which separates data into 2 classes.

Model and Helper Functions

def sigmoid(z):

return (1.0)/(1+np.exp(-z))

z = np.array([1,2,3,4,5])

sigmoid(z)

you will get an array.
This kind of functionality is called broadcasting
sigmoid fx^n is now applied on every element of array.

$$\sigma \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} = \begin{matrix} \sigma(1) \\ \sigma(2) \\ \sigma(3) \\ \sigma(4) \\ \sigma(5) \end{matrix}$$

It is possible only in numpy array.
Numpy does it bcz of a technique called as broadcasting.

Model and Helper Functions

```
def sigmoid(z) :
    return (1.0)/(1+np.exp(-z))
```

```
z = np.array([1,2,3,4,5])
sigmoid(z)
```

```
array([0.73105858, 0.88079708, 0.95257413, 0.98201379, 0.99330715])
```

Implement Perceptron Learning Algorithm

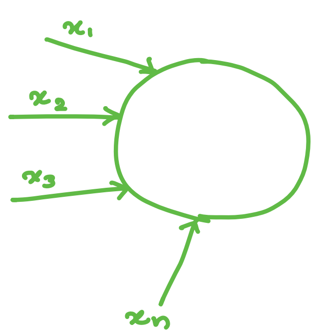
- Learn the weights
- Reduce the loss
- Make the predictions

$w = [w_0 \ w_1 \ \ldots \ w_n]$ $n \times 1$

$x = \begin{bmatrix} — x^1 — \\ — x^2 — \\ \vdots \\ — x^m — \end{bmatrix}$ $m \times n$

$m \times 1$ output

```
def predict (x, weights):
    z = np.dot (x, weights)
    predictions = sigmoid(z)
    return predictions
```



$X_{m \times (n+1)}$ matrix

$w_{n \times 1}$ vector

```
def loss (X, Y, weights):
    """ Binary Cross Entropy """
    Y_ = predict (X, weights)
    cost = np.mean (-Y * np.log (Y_) - (1-Y) * np.log (1- Y_))
    return cost
```

$$-\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log (1-\hat{y}^{(i)}) \right)$$

```
def update (X, Y, weights, learning_rate):
    """ Perform weight updates for 1 epoch """

    Y_ = predict (X, weights)
    dw = np.dot (X.T , Y_ - Y)

    m = X.shape[0]
    weights = weights - learning_rate * dw / (float(m))
    return weights
```

$$w_j = w_j - \eta \cdot \frac{\partial J}{\partial w_j}$$

$$\frac{\partial J}{\partial w_j} = (\hat{y} - y)\, x_j$$

$$X = \underset{m \times n}{\begin{bmatrix} \text{---} & x^1 & \text{---} \\ \text{---} & x^2 & \text{---} \\ \text{---} & x^3 & \text{---} \\ & \vdots & \\ \text{---} & x^m & \text{---} \end{bmatrix}} \qquad \underset{m \times 1}{\overset{\hat{y} - y}{\begin{bmatrix} \\ \\ \\ \end{bmatrix}}}$$

$$X^T = \underset{n \times m}{\begin{bmatrix} | & | & | & | \\ x^1 & x^2 & x^i & x^m \\ | & | & | & | \end{bmatrix}} \qquad \underset{m \times 1}{\overset{\hat{y} - y}{\begin{bmatrix} \\ \hat{y}^i - y^i \\ \\ \end{bmatrix}}}$$

$$\left(\hat{y}^{(i)} - y^{(i)}\right) x_j^{(i)}$$

```
def train (X, Y, learning_rate = 0.5, max epochs = 100):

    # Modify input to handle bias term
    ones = np.ones ((X.shape[0], 1))
    X = np.hstack ((ones, X))

    # Init weights 0
    weights = np.zeros (X.shape [1])

    # Iterate over all epochs and make
      updates
    for epoch in range (max epochs):
        weights = update (X, Y, weights,
                                learning_rate)

        if epoch % 10 == 0 :
            l = loss (X, Y, weights)
            print (" Epoch %d  Loss %.4f " % (epoch, l))

    return weights

train (X, Y)
```

after every 10 epochs print the progress

$$\begin{array}{c} x_1 \;\; x_2 \;\; \cdots \;\; x_n \\ \begin{bmatrix} \\ \\ \\ \end{bmatrix}_{m \times n} \end{array}$$

Add column of x_0 which is always 1

$$\begin{array}{c} x_0 \;\; x_1 \;\; x_2 \;\; \cdots \;\; x_n \\ \begin{bmatrix} 1 & & & & \\ 1 & & & & \\ \vdots & & & & \\ 1 & & & & \end{bmatrix}_{m \times (n+1)} \end{array}$$

$$\sum_{i=0}^{m} x_i w_i$$

$x_0 = 1$

Implement Perceptron Learning Algorithm

- Learn the weights
- Reduce the loss
- Make the pRedictions

```
def predict(X,weights) :
    """X -> m X (n+1) matrix , w -> n X 1 vector"""
    z = np.dot(X, weights)
    predictions = sigmoid(z)
    return predictions

def loss(X,Y,weights) :
    """Binary Cross Entropy"""
    Y_ = predict(X, weights)
    cost = np.mean(-Y*np.log(Y_) - (1-Y)*np.log(1-Y_))
    return cost

def update(X, Y, weights, learning_rate) :
    """Perform weight updates for 1 epoch"""
    Y_ = predict(X, weights)
    dw = np.dot(X.T, Y_-Y)

    m = X.shape[0]
    weights = weights - learning_rate*dw/(float(m))
    return weights

def train(X, Y, learning_rate=0.5, maxEpochs=100) :

    #Modify the input to handle the bias term
    ones = np.ones((X.shape[0],1))
    X = np.hstack((ones,X))

    #Init Weights 0
    weights = np.zeros(X.shape[1]) # n+1 entries

    #Iterate over all epochs and make updates
    for epoch in range(maxEpochs) :
        weights = update(X, Y, weights, learning_rate)

        if epoch % 10 == 0 :
            l = loss(X, Y, weights)
            print("Epoch %d Loss %.4f"%(epoch,l))

    return weights
```

```
train(X,Y)
```

```
Epoch 0 Loss 0.0006
Epoch 10 Loss 0.0005
Epoch 20 Loss 0.0005
Epoch 30 Loss 0.0005
Epoch 40 Loss 0.0005
Epoch 50 Loss 0.0004
Epoch 60 Loss 0.0004
Epoch 70 Loss 0.0004
Epoch 80 Loss 0.0004
Epoch 90 Loss 0.0004
```

Loss is Reducing.

$$\texttt{array([}\ \underbrace{\texttt{0.02204952}}_{w_0},\ \underbrace{\texttt{-0.30768518}}_{w_1},\ \underbrace{\texttt{1.90003958}}_{w_2}\texttt{])}$$

weights learnt by your classifier. For n features, we will have n+1 weights.

weights = train(X, Y, max Epochs = 500)

```
weights = train(X, Y, maxEpochs=500)
```

```
Epoch 0 Loss 0.0006
Epoch 10 Loss 0.0005
Epoch 20 Loss 0.0005
Epoch 30 Loss 0.0005
Epoch 40 Loss 0.0005
Epoch 50 Loss 0.0004
Epoch 60 Loss 0.0004
Epoch 70 Loss 0.0004
Epoch 80 Loss 0.0004
Epoch 90 Loss 0.0004
Epoch 100 Loss 0.0004
Epoch 110 Loss 0.0003
Epoch 120 Loss 0.0003
Epoch 130 Loss 0.0003
Epoch 140 Loss 0.0003
Epoch 150 Loss 0.0003
Epoch 160 Loss 0.0003
Epoch 170 Loss 0.0003
Epoch 180 Loss 0.0003
Epoch 190 Loss 0.0003
Epoch 200 Loss 0.0003
Epoch 210 Loss 0.0003
Epoch 220 Loss 0.0002
Epoch 230 Loss 0.0002
Epoch 240 Loss 0.0002
Epoch 250 Loss 0.0002
Epoch 260 Loss 0.0002
Epoch 270 Loss 0.0002
Epoch 280 Loss 0.0002
Epoch 290 Loss 0.0002
Epoch 300 Loss 0.0002
Epoch 310 Loss 0.0002
Epoch 320 Loss 0.0002
Epoch 330 Loss 0.0002
Epoch 340 Loss 0.0002
Epoch 350 Loss 0.0002
Epoch 360 Loss 0.0002
Epoch 370 Loss 0.0002
Epoch 380 Loss 0.0002
Epoch 390 Loss 0.0002
Epoch 400 Loss 0.0002
Epoch 410 Loss 0.0002
Epoch 420 Loss 0.0002
Epoch 430 Loss 0.0002
Epoch 440 Loss 0.0002
Epoch 450 Loss 0.0002
Epoch 460 Loss 0.0002
Epoch 470 Loss 0.0002
Epoch 480 Loss 0.0001
Epoch 490 Loss 0.0001
```

Loss is continuously reducing

loss is close to 0