In [ ]:

```
import numpy as np
```

Explanation

We need the output in this case to be 0 or 1 only , so we will use sigmoid function

$$sigmoid = \frac{1}{1 + e^{-value}}$$

But what is the steps to update my weights and bais ?

linear equation:

$$z = \sum_{i=1}^{m} w_i x_i + b$$

transform to sigmoid to get the value from 0 to 1:

$$y = \frac{1}{1 + e^{-z}}$$

Cost function (called binary cross-entropy):

$$C = -\sum_{i=1} target * \log y + (1 - target) * \log(1 - y)$$

we need to use chain rules :

$$\frac{\partial C}{\partial w} = \frac{\partial z}{\partial w} * \frac{\partial y}{\partial z} * \frac{\partial c}{\partial y}$$

$$\color{red}{\frac{\partial z}{\partial w_i} = x_i}$$

$$\frac{\partial c}{\partial y} = -(\frac{target}{y} + \frac{1 - target}{1 - y} * -1)$$

$$\frac{\partial c}{\partial y} = -\frac{target}{y} + \frac{1 - target}{1 - y}$$

$$\frac{\partial c}{\partial y} = \frac{y - y * target - target + y * target}{y(1 - y)}$$

$$\frac{\partial c}{\partial y} = \frac{y - target}{y(y - 1)}$$

$$\frac{\partial y}{\partial z} = (1 + e^{-z})^{-1}$$

$$\frac{\partial y}{\partial z} = -(1 + e^{-z})^{-2} * -e^{-z}$$

$$\frac{\partial y}{\partial z} = \frac{e^{-z}}{(1 + e^{-z})^2}$$

add in numerator 1 and -1

$$\frac{\partial y}{\partial z} = \frac{1 + e^{-z} - 1}{(1 + e^{-z})^2}$$

divide it into two parts :

$$\frac{\partial y}{\partial z} = \frac{1}{1 + e^{-z}} * \frac{1 + e^{-z} - 1}{1 + e^{-z}}$$

divide second part into two parts :

$$\frac{\partial y}{\partial z} = \frac{1}{1 + e^{-z}} * (\frac{1 + e^{-z}}{1 + e^{-z}} - \frac{1}{1 + e^{-z}})$$

we have

$$y = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial y}{\partial z} = y(1 - y)$$

then :

$$\frac{\partial C}{\partial w} = \frac{y - target}{y(y-1)} * x_i * y(1 - y)$$

$$\frac{\partial C}{\partial w} = (y - target) * x$$

for bais too:

$$\frac{\partial C}{\partial b} = \frac{\partial z}{\partial b} * \frac{\partial y}{\partial z} * \frac{\partial c}{\partial y}$$

$$\frac{\partial z}{\partial b} = 1$$

$$\frac{\partial C}{\partial b} = (y - target)$$

Data

we will add one's in features x for bais

In [ ]:

```
x = np.matrix([[1, 35, 78],
               [1, 30, 44],
               [1, 36, 73],
               [1, 60, 86],
               [1, 79, 75],
```

```
              [1, 45, 56],
              [1, 61, 96],
              [1, 75, 46],
              [1, 76, 87],
              [1, 84, 43],
              [1, 95, 38],
              [1, 75, 31]])

y = np.array([0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0])
```

In [ ]:

```
x.shape
```

Out[ ]:

```
(12, 3)
```

In [ ]:

```
x
```

Out[ ]:

```
matrix([[ 1, 35, 78],
        [ 1, 30, 44],
        [ 1, 36, 73],
        [ 1, 60, 86],
        [ 1, 79, 75],
        [ 1, 45, 56],
        [ 1, 61, 96],
        [ 1, 75, 46],
        [ 1, 76, 87],
        [ 1, 84, 43],
        [ 1, 95, 38],
        [ 1, 75, 31]])
```

In [ ]:

```
def gradient_descent(Ws_old, alpha, iterates):
    # assign weights new to contain zero's
    Ws_new = np.matrix(np.zeros(Ws_old.shape))
    # number of thetas
    parameter = int(Ws_old.ravel().shape[1])

    # loop in range number of iterates
    for i in range(iterates):
        # prediction
        z = x * Ws_old.reshape(-1, 1)
        predict = 1 / (1 + np.exp(-z))
        # loop in range number of thetas
        for j in range(parameter):
```

```
            # new theta = old theta - alph * (sum of predict - real
values)/number of rows
            Ws_new[0, j] = Ws_old[0, j] - alpha * (np.sum(predict - y))

        # put old thetas = new thetas
        Ws_old = Ws_new
    # return last values of weights
    return Ws_old
```

In [ ]:

```
# first theta is bais
thetas = np.matrix([0.5, 1, 0.25])
# call function
weights = gradient_descent(thetas, 0.001, 1000000)
```

In [ ]:

```
# see the weights
weights
```

Out[ ]:

```
matrix([[-0.0939568,  0.4060432, -0.3439568]])
```

In [ ]:

```
def predict(theta, X):
    probability = 1 / (1 + np.exp(-(x * weights.reshape(-1, 1))))
    return [1 if x >= 0.5 else 0 for x in probability]
```

In [ ]:

```
# predict values
prediction = predict(weights, x)
prediction
```

Out[ ]:

```
[0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1]
```

In [ ]:

```
# Calculate accuracy percentage between two lists
def accuracy_metric(actual, predicted):
        correct = 0
        for i in range(len(actual)):
                if actual[i] == predicted[i]:
                        correct += 1
        return correct / float(len(actual)) * 100.0
```

In [ ]:

```
accuracy_metric(y, prediction)
```

Out[ ]:

```
66.66666666666666
```

In [ ]: