Logistic Regression: Using diabetes dataset to test the math model.

Y_cap = predicted value x --> independent variable w = weight b = bias

Gradient Descent

Gradient Descent is an optimization algorithm used for minimizing the loss function in various machine learning algorithm. It is used for updating the parameters of the learning model. Gradient Descent is the base way to find bias and weight value.

$$w_2 = w - \alpha \cdot dw$$

$$b_2 = b - \alpha \cdot db$$

w_2 = updated weight

w = previous weight

α = learning rate

For Bias

b_2 = updated bias

b = previous bias

Learning rate: How much change you want to impact to weight and bias Is the tuning parameter in an optimize algorithm that determines the step size at each iteration while moving toward a minimum of a loss function

Work flow of the Logistic Regression model below

Importing the Dependencies

In [9]:
```
import numpy as np
```

Logistic Regression

Please pay attention to this cell before going to the next cell STEP 1

class Logistic_Regression():

Class starts with a capital letter and it must be a single word followed by parenthesis and a colon Creating Logistic_Regression as an object and we need a template And the template is created using the class. The class takes some properties (arguments are Learning rate and No of iterations) When creating and object the first parameter you will create inside your function is 'self'

1. Assuming our number of iteration is 100 our model will go through the data 100times and each time it will update the parameters using Gradient Descent

   #Learning Rate

```
Typical values of learning_rate range from 0.001 to 0.1. You can start with
0.01 A commonly used default

 No of iteration is not fix you can use any number
 num_iterations = 1000
```

learning_rate number is assumption and is not a constant same with number of iteration

```
STEP 2
```

def **init**(): #for initiating the parameters we are creating to give to our objects

```
  Initialize parameters (weights, bias, etc.) ===The are model parameters

 #implementing Gradient Descent for optimization
```

```
STEP 3
```

```
def fit():  #to fit our dataset to Logistic Regression model
```

1. Implementing the fitting process

```
 Fiting the funtion to the model we need two things
     1. I seperated features and target of my dataset(x_axis and y_axis)
     2. You must determine the number of data points in your dataset and
determine the number of input features in your dataset
         Input features in your dataset is the sum of x(features) columns in
your dataset
         Data points in your dataset is the total number of rows and you
have to ignore the output(target) column
         The data points will be like a matrix
```

weight and bias == Model parameters

CREATING GRADIENT DESCENT ALGORITHM "This will enable us to replace the value of w and b since we can't assume them to be zeros(0) else the model won't be accurate." Our model will go through the data the number of iteration times we give it. Then the bias and weights values will keep changing till our model works efficiently and at this point the model will have a minimum cost function.

To implement Gradient Descent use a for loop Each time my iteration run my weight, bias value will be updated and that is the purpose of using a for loop And when the number of iteration is complete we will get a proper model which has best fit for the weight bias value. This means that the model will make a prediction and it will has minimum cost function. minimum cost function = value predicted my the model and the true value are clsoe to each other and it means the model is working accurately.

```
STEP 4
```

```
def updated_weights(): #it keeps changing the rate of bias and weight so
that we can  have a good model
    # Implement weight update logic
```

1. To implement weight:

```
 Weight is equal to the total number of input columns in a dataset(sum of
features column multiply by np.zeros)
```

1. Implementing all the equations here and is call Gradient Descent algorithm

```
 since we intiated the previous values for w and b as 0 so we will be
finding the updated b and w
 Finding the equation of the y_cap  which == sigmoid function
```

2. T = transpose of a matrix And in matrix multiplication number of columns in the First matrix should be equal to the number of rows in the second column

   To satisfy the rule that the number of columns in the first matrix must equal the number of rows in the second, I used z = x . w

   This allows me to compute the linear combination appropriately in logistic regression.

3. Updating weight and bias using gradient descent

```
                              STEP 5


def predict(): #to predict the outcome of the values of y value to be
either 0 or 1
    # Implement prediction logic
```

if y_cap > 0.5 = 1 if is less than 0.5 = 0

4. Building the prediction when you give the x value it will give the output as zero or 1

In [11]:

```
class Logistic_Regression():
    def __init__(self, learning_rate, no_of_iterations):
        self.learning_rate = learning_rate
        self.no_of_iterations = no_of_iterations
    #Declaring learning rate and number of iteration (Hyperparameters)


#======================End of Step one ==============================#


#======================Step two starts here ==============================#
    #The purpose of the fit function is to train the the model, so I will fit the datase
    def fit(self, x, y):
        self.m, self.n = x.shape      # m = numbers of rows and n = total number of colu

        self.w = np.zeros(self.n)
        self.b = 0
        self.x = x   #features of the data
        self.y = y   #target(Outcome) column in the dataset

        #implementing Gradient Descent for optimization

        for i in range(self.no_of_iterations):
            self.updated_weights()

#======================End of Step two ==============================#

#======================Step three starts here ==============================#

    def updated_weights(self):

        #Y_cap formula (sigmoid function)
```

```
        # y_cap = 1/(1+np.exp(-z))  # z = w.x + b   #
         y_cap = 1 / (1 + np.exp( - (self.x.dot(self.w) + self.b)))  #this equation is si

         #where w and x are arrays but b is a single integer values

         #exp = e which is the Euler's number = 2.718

         # to get the Eulers's number in jupyternotebook  print(np.exp(1))

         #replacing -z with its parameters.


    #Building the derivatives




         dw = (1/self.m)*np.dot(self.x.T, (y_cap - self.y))


    # y_cap is the predicted value using the sigmoid equation
    # y is the true value (outcome column)

         db = (1/self.m)*np.sum (y_cap - self.y)

    #updating the weights and bias using Gradient Descent


         self.w = self.w - self.learning_rate * dw

         self.b = self.b  - self.learning_rate * db




#=====================End of Step three ============================#


    def predict(self, x):  # to find the value of y

         y_pred = 1 / (1 + np.exp( - (x.dot(self.w) + self.b )))    #we can't use self.x

         y_pred = np.where (y_pred > 0.5, 1, 0)

         return(y_pred)
```

In [12]:
```
#Importing the libraries

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import Log_Reg
```

In [13]:
```
#importing the data

diabetes = pd.read_csv('diabetes.csv')
```

In [14]:
```
diabetes.head()
```

Out[14]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcom

0	6	148	72	35	0	33.6		0.627	50
1	1	85	66	29	0	26.6		0.351	31
2	8	183	64	0	0	23.3		0.672	32
3	1	89	66	23	94	28.1		0.167	21
4	0	137	40	35	168	43.1		2.288	33

In [15]:
```
diabetes.shape
```

Out[15]:
```
(768, 9)
```

In [16]:
```
diabetes.describe()
```

Out[16]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFuncti
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.0000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.4718
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.3313
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0780
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.2437
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.3725
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.6262
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.4200

In [17]:
```
diabetes['Outcome'].value_counts()
```

Out[17]:
```
Outcome
0    500
1    268
Name: count, dtype: int64
```

0 = Non-Diabetes 1 = Diabetes

In [18]:
```
diabetes.groupby('Outcome').mean()
```

Out[18]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFun
Outcome							
0	3.298000	109.980000	68.184000	19.664000	68.792000	30.304200	0.42
1	4.865672	141.257463	70.824627	22.164179	100.335821	35.142537	0.55

In [19]:
```
x = diabetes.drop(columns = 'Outcome', axis = 1)
y = diabetes['Outcome']
```

In [20]:
```
print(x) #features
```

```
     Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0              6      148             72             35        0  33.6
1              1       85             66             29        0  26.6
2              8      183             64              0        0  23.3
3              1       89             66             23       94  28.1
4              0      137             40             35      168  43.1
..           ...      ...            ...            ...      ...   ...
763           10      101             76             48      180  32.9
764            2      122             70             27        0  36.8
765            5      121             72             23      112  26.2
```

```
766              1      126               60               0         0  30.1
767              1       93               70              31         0  30.4

     DiabetesPedigreeFunction  Age
0                       0.627   50
1                       0.351   31
2                       0.672   32
3                       0.167   21
4                       2.288   33
..                        ...  ...
763                     0.171   63
764                     0.340   27
765                     0.245   30
766                     0.349   47
767                     0.315   23

[768 rows x 8 columns]
```

In [21]:
```
print(y) #target
```

```
0      1
1      0
2      1
3      0
4      1
      ..
763    0
764    0
765    0
766    1
767    0
Name: Outcome, Length: 768, dtype: int64
```

In [22]:
```
scaler = StandardScaler()
```

In [23]:
```
scaler.fit(x)
```

Out[23]:
```
StandardScaler
StandardScaler()
```

In [24]:
```
standardized_data = scaler.transform(x)
```

In [25]:
```
print(standardized_data)
```

```
[[ 0.63994726  0.84832379  0.14964075 ...  0.20401277  0.46849198
   1.4259954 ]
 [-0.84488505 -1.12339636 -0.16054575 ... -0.68442195 -0.36506078
  -0.19067191]
 [ 1.23388019  1.94372388 -0.26394125 ... -1.10325546  0.60439732
  -0.10558415]
 ...
 [ 0.3429808   0.00330087  0.14964075 ... -0.73518964 -0.68519336
  -0.27575966]
 [-0.84488505  0.1597866  -0.47073225 ... -0.24020459 -0.37110101
   1.17073215]
 [-0.84488505 -0.8730192   0.04624525 ... -0.20212881 -0.47378505
  -0.87137393]]
```

In [26]:
```
x = standardized_data
```

In [27]:
```
print(x)
print(y)
```

```
[[ 0.63994726  0.84832379  0.14964075 ...  0.20401277  0.46849198
   1.4259954 ]
 [-0.84488505 -1.12339636 -0.16054575 ... -0.68442195 -0.36506078
  -0.19067191]
 [ 1.23388019  1.94372388 -0.26394125 ... -1.10325546  0.60439732
  -0.10558415]
 ...
 [ 0.3429808   0.00330087  0.14964075 ... -0.73518964 -0.68519336
  -0.27575966]
 [-0.84488505  0.1597866  -0.47073225 ... -0.24020459 -0.37110101
   1.17073215]
 [-0.84488505 -0.8730192   0.04624525 ... -0.20212881 -0.47378505
  -0.87137393]]
0      1
1      0
2      1
3      0
4      1
      ..
763    0
764    0
765    0
766    1
767    0
Name: Outcome, Length: 768, dtype: int64
```

Train Test Split

In [29]:
```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, stratify = y,
```

In [30]:
```
print(x.shape, x_train.shape, x_test.shape)
```

```
(768, 8) (614, 8) (154, 8)
```

Model Traing

In [32]:
```
#calling the logistic model so as to manually use the learning rate and no of iterations

classifier = Log_Reg.Logistic_Regression(learning_rate = 0.01, no_of_iterations = 1000)
```

In [33]:
```
classifier.fit(x_train, y_train)
```

Accuracy Score

In [35]:
```
#Accuracy score on the x axis

x_train_prediction = classifier.predict(x_train)  #predicted value and y_train is the tr
training_data_accuracy = accuracy_score(y_train, x_train_prediction) #combinig them toge
```

In [36]:
```
print('Accuracy score of the training data', training_data_accuracy )
```

```
Accuracy score of the training data 0.7785016286644951
```

In [37]:
```
x_test_prediction = classifier.predict(x_test)
test_data_prediction = accuracy_score(x_test_prediction, y_test)
```

In [38]:
```
print('Accuracy score of the test data', test_data_prediction )
```

```
Accuracy score of the test data 0.7597402597402597
```

Making a Predictive System

In [40]:
```python
import warnings
warnings.filterwarnings('ignore', category=UserWarning)

input_data = (6,148,72,35,0,33.6,0.627,50)

input_data_as_numpy = np.asarray(input_data)

input_data_reshaped = input_data_as_numpy.reshape(1, -1)

std_data = scaler.transform(input_data_reshaped)
print(std_data)

prediction = classifier.predict(std_data)
print(prediction)

if prediction [0]== (0):
    print('The patience has no diabetes')

else:
    print('The person has diabetes')
```

```
[[ 0.63994726  0.84832379  0.14964075  0.90726993 -0.69289057  0.20401277
   0.46849198  1.4259954 ]]
[1]
The person has diabetes
```

In [41]:
```python
input_data = (1,189,60,23,846,30.1,0.398,59)

input_data_as_numpy = np.asarray(input_data)

input_data_reshaped = input_data_as_numpy.reshape(1, -1)

std_data = scaler.transform(input_data_reshaped)
print(std_data)

prediction = classifier.predict(std_data)
print(prediction)

if prediction [0]== (0):
    print('The patience has no diabetes')

else:
    print('The person has diabetes')
```

```
[[-0.84488505  2.13150675 -0.47073225  0.15453319  6.65283938 -0.24020459
  -0.2231152   2.19178518]]
[1]
The person has diabetes
```

In [ ]:

In [ ]: