**Pseudocode for Logistic Regression**

***1.Import the necessary libraries:***

*import numpy as np*

*import pandas as pd*

*import matplotlib.pyplot as plt*

*import seaborn as sns*

***2.Load the data***

*data = pd.read\_csv(‘data.csv’)*

***3.Prepare the data for modeling: (****Preprocessing and EDA On dataset****)***

*X = data.drop('target\_variable', axis=1)*

*y = data['target\_variable’]*

*m = len(y) # number of training examples*

*X = np.hstack((np.ones((m, 1)), X)) # add intercept term to X*

*theta = np.zeros((X.shape[1], 1)) # initialize parameters*

**#** **Do normalization** ( for getting values in range of 0-1)

def normalize(X):

    X\_Norm=((X-X.min(0))/(X.max(0)-X.min(0)))

    return X\_Norm

***4.Understanding the Sigmoid function***

*The sigmoid function in logistic regression returns a probability value that can then be mapped to two or more discrete classes. Given the set of input variables, our goal is to assign that data point to a category (either 1 or 0). The sigmoid function outputs the probability of the input points belonging to one of the classes.*

***#Defining a sigmoid function***

def sigmoid(z):

op = 1/(1 + np.exp(-z))

return op

***5.The Loss Function***

*The loss function consists of parameters/weights, when we say we want to optimize a loss function by this we simply refer to finding the best values of the parameters/weights.*

The loss function for Logistic Regression is defined as:

def loss(h, y):

return (-y \* np.log(h) - (1 - y) \* np.log(1 - h)).mean()

***6.Gradient descent***

*The Gradient descent is just the derivative of the loss function with respect to its weights.*

We get this after we find find the derivative of the loss function:

def gradient\_descent(X, h, y):

return np.dot(X.T, (h - y)) / y.shape[0]

*The weights are updated by subtracting the derivative (gradient descent) times the learning rate. Updating the weights:*

***7.Train the model using gradient descent:***

alpha = 0.01 # learning rate

num\_iters = 1000 # number of iterations

theta, J\_history = gradient\_descent(X, y, theta, alpha, num\_iters)

***8.Predict the target variable:***

y\_pred = sigmoid(np.dot(X, theta))

y\_pred = np.round(y\_pred) # round to 0 or 1

***9.Evaluate the performance of the model:***

accuracy = np.mean(y\_pred == y)

confusion = np.zeros((2, 2))

confusion[0, 0] = np.sum(np.logical\_and(y\_pred == 0, y == 0))

confusion[0, 1] = np.sum(np.logical\_and(y\_pred == 1, y == 0))

confusion[1, 0] = np.sum(np.logical\_and(y\_pred == 0, y == 1))

confusion[1, 1] = np.sum(np.logical\_and(y\_pred == 1, y == 1))

***Putting it all together***

*Let’s create a class to compile the steps mentioned above. Here’s the complete code for implementing Logistic Regression from scratch. I have worked with the Python numpy module for this implementation.*

class LogisticRegression:

def \_init\_(self,x,y):

self.intercept = np.ones((x.shape[0], 1))

self.x = np.concatenate((self.intercept, x), axis=1)

self.weight = np.zeros(self.x.shape[1])

self.y = y

***#Sigmoid method***

def sigmoid(self, x, weight):

z = np.dot(x, weight)

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

***#method to calculate the Loss***

def loss(self, h, y):

return (-y \* np.log(h) - (1 - y) \* np.log(1 - h)).mean()

***#Method for calculating the gradients***

def gradient\_descent(self, X, h, y):

return np.dot((h - y)) / y.shape[0]

def fit(self, lr , iterations):

for i in range(iterations):

sigma = self.sigmoid(self.x, self.weight)

loss = self.loss(sigma,self.y)

dW = self.gradient\_descent(self.x , sigma, self.y)

***#Updating the weights***

self.weight -= lr \* dW

return print('fitted successfully to data')

*#****Method to predict the class label.***

def predict(self, x\_new , treshold):

x\_new = np.concatenate((self.intercept, x\_new), axis=1)

result = self.sigmoid(x\_new, self.weight)

result = result >= treshold

y\_pred = np.zeros(result.shape[0])

for i in range(len(y\_pred)):

if result[i] == True:

y\_pred[i] = 1

else:

continue

return y\_pred

**Pseudo Code**

1. 0 → β

2. Compute y by setting its elements to

yi = {1 if **g**i = 1; 0 if **g**i = 2} , i = 1, 2, ..., N.

3. Compute p by setting its elements to

p(xi ; β) = **e^(b0 + b1\*x) / (1 + e^(b0 + b1\*x))**

4. Compute the diagonal matrix W. And Compute the N × (p + 1) matrix X˜ by multiplying the ith row of matrix X by

p(xi ; β)(1 − p(xi ; β)), i = 1, 2, ..., N:

5. z ← Xβ + (y − p).

6. β ← β+ ((y-p).

7. If the stopping criteria is met, stop; otherwise go back to step 3

Explanation:

1. *Initialize the coefficients β to zero.*
2. *Compute the binary labels y based on the integer labels gi, where yi = 1 if gi = 1 and yi = 0 if gi = 2.*
3. *Compute the probability p for each data point using the logistic function: p(xi ; β) = e^(b0 + b1x) / (1 + e^(b0 + b1x)) where b0 and b1 are the intercept and slope coefficients, respectively, and x is the input feature vector for a single data point.*
4. *Compute the diagonal weight matrix W and the N × (p + 1) design matrix X˜, where each row of X˜ is the product of the corresponding row of X with the probability of the positive class (p(xi ; β)) and the probability of the negative class (1 − p(xi ; β)).*
5. *Compute the adjusted response variable z as: z = Xβ + W^(-1) (y − p)*
6. *Update the coefficients using the normal equation: β = β + (X^T X)^(-1) X^T (y − p)*

*where X^T is the transpose of the design matrix X and x̄ is the mean of the input features.*

1. *Check if the stopping criteria is met. If not, go back to step 3.*

*The stopping criteria could be a maximum number of iterations, a minimum change in the coefficients, or a maximum likelihood improvement. The logistic regression algorithm can be used for both binary and multi-class classification problems by extending the design matrix and response variable to accommodate multiple classes.*

**Overview of Logistic Regression**

