**Logistic Regression**

Logistic Regression is a classification algorithm used for binary classification tasks, predicting the probability of an instance belonging to a particular class. It's based on the logistic function, which maps input features to probabilities. The model learns parameters through optimization techniques like gradient descent, minimizing a cost function. Once trained, it makes predictions on new data by computing probabilities and applying a threshold. Logistic Regression is widely used across industries for its simplicity, interpretability, and efficiency, making it suitable for various real-world applications such as healthcare, finance, and marketing.

**The model**

1

Linear

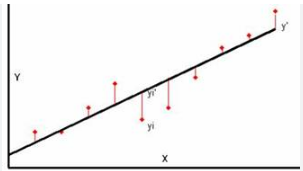
0

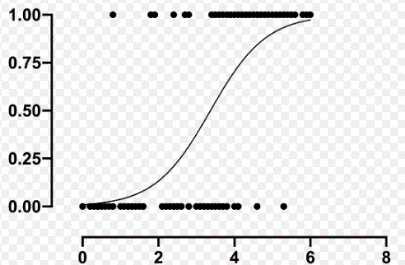
Sigmoid

: Weight that the model learns to direct.

Our data, and they enter the model equation as variable

Linear equation of linear regression. A graph of a linear regression performed using the Z equation.





"Sigmoid". The "Sigmoid" provides an answer to classification problems. If the sigmoid result is less than 0x5, then it will be classified as 0, otherwise it will be classified as 1. A graph of a linear regression "Sigmoid".

**The equations of the logistic regression model**

*Linear function:*

*Loss function:*

*Cost function:*

*Loss function:* The cross-entropy loss function measures divergence between predicted probabilities and actual binary targets. Optimizing these drives logistic model to output accurate class probabilities.

*Cost function:* the cross-entropy cost function measures divergence between predicted probabilities and actual binary targets. By optimizing this cost function via gradient descent or other methods, we can fit the parameters of the logistic regression model to accurately predict the probability of class membership.

They are functions for the purpose of the gradient descent.

Any implementation of these equations in the code is in vector form for efficiency.

**Dot product & elementwise function**

Before we start to implement the equation in the code, I will explain two operations that we used to implement the code.

**Dot product:**

\*\* Matrix order m\*n meaning, rows = m & columns = n.

When we run the function "np.dot" on our values (Two matrices of different order), We will get an operation similar to matrix multiplication, and the result of the operation will be a matrix of a different order.

**Example:**

We have a matrix and its order is (3\*2), we will call it **m1**, and we have a matrix and its order is (2\*4), we will call it **m2**.

Matrix multiplication:

Np.dot:

**elementwise function**

An operation elementwise preserves the order of the matrix and does not change the order.

**Example:**

We have a matrix and its order is (3\*2), we will call it **m1**.

elementwise function

In summary, **Np.dot** changes the dimensions of the matrix, and **elementwise** keeps the dimensions of the matrix.

**Explanation of the functions of the class**

Usually, our is of order (), that is:

And represents a one-dimensional vector, "target – vector".

These equations I presented earlier are Andrew's. N.G:

*Linear function:*

*Loss function:*

*Cost function:*

According to these equations, the need is for to be of order () and to be of order () That is, a two-dimensional order vector.

So, for these changes we created the "fit" function, which will match the data to the formulas, and now we will explain about it.

**"fit" function**

# x1: order matrix (m \* n)  
# y1: A one-dimensional vector (m)  
# X: order matrix (n \* m)  
# Y: Two-dimensional vector (1 \* m)  
# w, dw: Two-dimensional vector (n \* 1)  
# b, db: float

def fit(self, x1, y1, num\_iterations=1000, learning\_rate=0.5):  
 X = x1.T # Change the order of the matrix from (m \* n) to (n \* m)  
 Y = y1.reshape(1, -1) # Change the dimension of the vector from (m) to (1 \* m), One-dimension to Two-dimension  
 self.w = np.zeros((X.shape[0], 1)) # 'w' is an order vector (n \* 1), that 'n' is the number of features.  
 self.b = 0 # 'b' is a variable of a kind 'flout'.  
 costs = [] # A list that will store the results of the cost equation.  
 for i in range(num\_iterations + 1):  
 dw, db, cost = self.propagate(X, Y) # We send the 'X' and 'Y' to the 'propagate' function, And the function will return values that will be stored in these variables.  
 self.w -= learning\_rate \* dw # For the purpose of gradient decent, from a variable 'w' we subtract the learning\_rate and dw.   
 self.b -= learning\_rate \* db # For the purpose of gradient decent, from a variable 'b' we subtract the learning\_rate and db.   
 if i % 100 == 0: # Every hundred iterations.  
 costs.append(cost) # We will add the cost result to the list of costs.  
 print('cost after itertion %i: %f' %(i, cost)) # We print the iteration number together with the cost result.   
 return costs # The 'fit' function return the list of costs.

**"propagate" function**

# X: order matrix (n \* m)  
# Y, A: Two-dimensional vector (1 \* m)  
# dw: Two-dimensional vector (n \* 1)  
# db: float

def propagate(self, X, Y):  
 m = X.shape[1] # Calculate the properties.  
 A = sigmoid(np.dot(self.w.T, X) + self.b) # The sigmoid calculation, its result will determine whether the person survived or not.  
 cost = (-1 / m) \* np.sum(Y \* np.log(A) + (1 - Y) \* np.log(1 - A)) # The cost equation is a sum of our loss equation, when its result is smaller, the success of the model prediction is greater.  
 dw = (1 / m) \* np.dot(X, (A - Y).T) # dw is for gradient descent.  
 db = (1 / m) \* np.sum(A - Y) # db is for gradient descent.  
 return dw, db, cost

**"predict" function**

# x1: order matrix (m \* n)  
# X: order matrix (n \* m)  
# Y, A: Two-dimensional vector (1 \* m)  
# w: Two-dimensional vector (n \* 1)

def predict(self, x1): # Prediction function.  
 X = x1.T # Change the order of the matrix from (m \* n) to (n \* m)  
 # np.dot(1 \* n, n \* m) = (1 \* m)   
 A = sigmoid(np.dot(self.w.T, X) + self.b) # The sigmoid calculation, its result will determine whether the person survived or not.  
 # np.where(1 \* m)  
 Y\_pred = np.where(A >= 0.5, 1, 0) # The function 'np.where' goes over the matrix A, and under the condition that if the value in the place is greater than 0.5 then we will classify as 1 and if the value in the place is less than 0.5 then we will classify as 0. 'Y\_pred' is a matrix of the same order (1 \* m), with values (0,1).   
 return Y\_pred.reshape(-1) # The function 'Y\_pred.reshape(-1)' changes the two-dimensional vector(matrix) For a one-dimensional vector.

**sigmoid function**

def sigmoid(z): # The sigmoid provides an answer to classification problems, As in our problem if the passenger survived or not.  
 s = 1 / (1 + np.exp(-z))   
 return s