Logistic Regression(Multiple Features(Binary Output)) With Regularization - From Scratch

IMPORTING LIBRARIES AND DEPENDENCIES

In [2]:
```
# Importing dependencies and dataset
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
```

CREATING DATAFRAME

In [3]:
```
#Creating the dataframe
cancer = load_breast_cancer()
df = pd.DataFrame(data = cancer['data'], columns = cancer['feature_name
s'])
df['class'] = cancer['target']
df.head()
```

Out[3]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809

5 rows × 31 columns

In [4]:
```
#Data statistics
df.describe()
```

Out[4]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000
mean	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720
min	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000
25%	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560
50%	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540
75%	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800

8 rows × 31 columns

FEATURE SCALING DATA

In [5]:
```
#Need to scale feature.  Standard Scaling(use with normally distributed
 data)
#Dropping class column to scale data features
df_drop = df.drop(['class'], axis = 1)

df_drop_scaled = preprocessing.scale(df_drop)

#Adding the ones column to training data
X = np.c_[np.ones(df.shape[0]),df_drop_scaled]
```

SPLITTING DATA INTO TRAINING AND TESTING DATA SETS

In [6]:
```
#Splitting Data Set into Training and Testing Data Sets
X_train, X_test, y_train, y_test = train_test_split(X, df['class'], test
_size = .80, random_state = 1)
```

CREATING THE SIGMOID HYPOTHESIS FUNCTION

In [7]:
```
#creating the sigmoid function for the prediction hypothesis

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

CALCULATING THE COST FUNCTION

In [57]:

```
#Calculating the regularized cost function

def cost_function(x_train, y_train, thetas_array, n, reg):
    prediction = sigmoid(np.dot(X_train,np.transpose(thetas_array)))
    cost = -(np.sum((y_train * np.log(prediction)) + ((1 - y_train) * np
.log(1-prediction)))/n) + ((reg/(2*n))*np.sum(thetas_array[1:]**2))
    """error = (-y_train * np.log(prediction)) - ((1-y_train)*np.log(1-p
rediction))
    cost = 1/n * sum(error)"""
    return cost
```

CALCULATING GRADIENT DESCENT FUNCTION

In [58]:

```
#Calculating the regularized gradient descent

def gradient_descent(X_train, y_train, alpha, thetas_array,n, reg):
    #thetas = thetas_array - ((alpha/n) * np.sum(np.dot((sigmoid(np.dot
(X_train,np.transpose(thetas_array))) - y_train), X_train)))
    thetas = thetas_array[1:] - (alpha/n) * ((np.dot((sigmoid(np.dot(X_t
rain,np.transpose(thetas_array))) - y_train), X_train[:,1:]))) + ((reg/n
)*thetas_array[1:])
    return thetas
```

TRAINING THE DATA

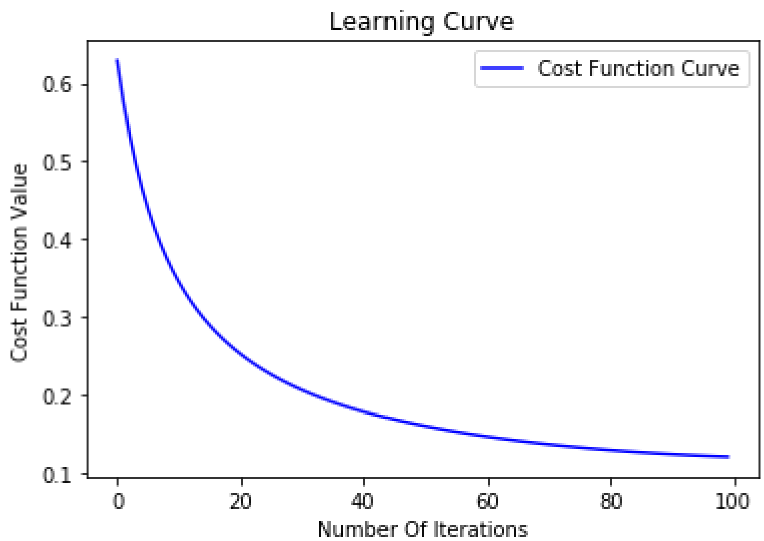
In [59]:
```
def training(X_train, y_train, alpha, iters, reg):
    n = len(X_train)
    thetas_array = np.zeros(31)
    #thetas_array = np.random.rand(len(cancer['feature_names']) + 1)
    thetas_j = 0
    theta_0 = 0
    cost = []

    for i in range(iters):
        theta_0 = thetas_array[0] - ((alpha/n)*(np.sum(np.dot(X_train,th
etas_array) - y_train) * 1))
        thetas_j = gradient_descent(X_train, y_train, alpha, thetas_arra
y,n, reg)
        thetas_array = np.insert(thetas_j, 0, theta_0)
        cost.append(cost_function(X_train, y_train, thetas_array, n, reg
))

    #Plot cost function error per iteration
    x = np.arange(0, len(cost), step=1)
    plt.plot(x, cost, "-b", label="Cost Function Curve")
    plt.title("Learning Curve")
    plt.xlabel("Number Of Iterations")
    plt.ylabel("Cost Function Value")
    plt.legend()
    plt.show()

    return thetas_array, cost[-1]
```

In [60]:
```
training(np.array(X_train), np.array(y_train), .03, 100, 1)
```



Out[60]:
```
(array([ 0.71228545, -0.39054255, -0.26844785, -0.38816903, -0.3618526
6,
        -0.15468999, -0.23998914, -0.30565085, -0.38846152, -0.1655536
5,
         0.05166807, -0.31034093, -0.02653106, -0.29251665, -0.2731325
2,
         0.17704996, -0.03878244, -0.03732198, -0.14943533,  0.0932752
4,
         0.03270494, -0.44234069, -0.35922679, -0.43609101, -0.3926815
7,
        -0.21138127, -0.28713889, -0.3227788 , -0.43763631, -0.3013891
1,
        -0.18191863]), 0.12140241218251954)
```

CALCULATING TRAINING ACCURACY

In [14]:
```
def training_accuracy(X, y, thetas_array):
    y = np.array(y_train)
    z = np.dot(X_train, thetas_array)
    prediction = sigmoid(z)
    total_number_pred = len(prediction)
    TP = 0
    FP = 0
    FN = 0
    TN = 0

    for i in range(len(y_train)):
        if prediction[i] >= 0.5 and y[i] == 1:
            TP += 1
        elif prediction[i] < 0.5 and y[i] == 1:
            FP += 1
        elif prediction[i] >= 0.5 and y[i] == 0:
            FN += 1
        else:
            TN += 1
    accuracy = round((TP + TN)/(TP + TN + FN + FP) * 100, 2)
    print(f'Training Accuracy: {accuracy}%')
```

In [61]:
```
training_accuracy(X_train, y_train, [ 0.71228545, -0.39054255, -0.268447
85, -0.38816903, -0.36185266,
        -0.15468999, -0.23998914, -0.30565085, -0.38846152, -0.16555365,
         0.05166807, -0.31034093, -0.02653106, -0.29251665, -0.27313252,
         0.17704996, -0.03878244, -0.03732198, -0.14943533,  0.09327524,
         0.03270494, -0.44234069, -0.35922679, -0.43609101, -0.39268157,
        -0.21138127, -0.28713889, -0.3227788 , -0.43763631, -0.30138911,
        -0.18191863])
```

```
Training Accuracy: 96.46%
```

PREDICTING ON A NEW DATA POINT

In [45]:
```
#Function to predict probability of developing breast cancer.  Enter a n
ew data point from testing set.
def predict(X_test, thetas_array):
    z = np.dot(X_test, thetas_array)
    prediction = round(sigmoid(z) *100,2)
    return f"You have a {prediction}% change of breast cancer."
```

In [69]:
```
predict(np.array(X_test[11]), [ 0.71228545, -0.39054255, -0.26844785, -
0.38816903, -0.36185266,
        -0.15468999, -0.23998914, -0.30565085, -0.38846152, -0.16555365,
         0.05166807, -0.31034093, -0.02653106, -0.29251665, -0.27313252,
         0.17704996, -0.03878244, -0.03732198, -0.14943533,  0.09327524,
         0.03270494, -0.44234069, -0.35922679, -0.43609101, -0.39268157,
        -0.21138127, -0.28713889, -0.3227788 , -0.43763631, -0.30138911,
        -0.18191863])
```

Out[69]:
```
'You have a 6.66% change of breast cancer.'
```