Practice No.3

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

sns.set()

from matplotlib.colors import ListedColormap

from sklearn.datasets import make\_gaussian\_quantiles

cmap\_light = ListedColormap(['#FFAAAA', '#AAAAFF'])

cmap\_bold = ListedColormap(['#FF0000', '#0000FF'])

N = 500

X, Y = make\_gaussian\_quantiles(mean=None, cov=0.5, n\_samples = N,

n\_features = 2, n\_classes=2, shuffle = True,

random\_state = 42)

X = X.T

Y = Y.reshape(1, len(Y))

print(X.shape)

print(Y.shape)

plt.figure(figsize = (10, 6))

plt.scatter(x = X[0, :], y = X[1, :], c = Y, s = 30, cmap = plt.cm.Spectral);

plt.xlabel("$X\_1$")

plt.ylabel("$X\_2$")

plt.show()

def sigmoid(z):

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

return sigmoid

def tanh(z):

tan = (np.exp(z) - np.exp(-z)) / (np.exp(z) + np.exp(-z))

return tan

def layer\_sizes(X, Y):

"""

Returns:

n\_x -- size of the input layer

n\_h -- size of the hidden layer

n\_y -- size of the output layer

"""

n\_x = X.shape[0]

n\_h = 4

n\_y = Y.shape[0]

return n\_x, n\_h, n\_y

def initialize\_params(n\_x, n\_h, n\_y):

W1 = np.random.randn(n\_h, n\_x) \* 0.01

b1 = np.zeros((n\_h, 1))

W2 = np.random.randn(n\_y, n\_h) \* 0.01

b2 = np.zeros((n\_y, 1))

return W1, b1, W2, b2

def forward\_propagation(X, W1, b1, W2, b2):

Z1 = np.dot(W1, X) + b1

A1 = tanh(Z1)

Z2 = np.dot(W2, A1) + b2

A2 = sigmoid(Z2)

return Z1, A1, Z2, A2

def compute\_cost(Y, A2):

m = Y.shape[1]

cost = (-1/m) \* np.sum(np.multiply(Y ,np.log(A2)) + np.multiply((1-Y), np.log(1-A2)))

cost = float(np.squeeze(cost))

return cost

def backward\_propagation(X, Y, W1, b1, W2, b2, Z1, A1, Z2, A2):

m = X.shape[1] # Number of training examples

dZ2 = A2 - Y

dW2 = (1/m) \* np.dot(dZ2, A1.T)

db2 = (1/m) \* (np.sum(dZ2, axis = 1, keepdims = True))

dZ1 = np.dot(W2.T, dZ2) \* (1 - np.power(A1, 2))

dW1 = (1/m) \* (np.dot(dZ1, X.T))

db1 = (1/m) \* (np.sum(dZ1, axis = 1, keepdims = True))

return dW1, db1, dW2, db2

def update\_params(W1, b1, W2, b2, dW1, db1, dW2, db2, learning\_rate):

W1 = W1 - learning\_rate \* dW1

b1 = b1 - learning\_rate \* db1

W2 = W2 - learning\_rate \* dW2

b2 = b2 - learning\_rate \* db2

return W1, b1, W2, b2

def neural\_network(X, Y, n\_h, learning\_rate, num\_iterations = 500):

n\_x = layer\_sizes(X, Y)[0]

n\_y = layer\_sizes(X, Y)[2]

costs = []

# Initialize parameters

W1, b1, W2, b2 = initialize\_params(n\_x, n\_h, n\_y)

# Loop (gradient descent)

for i in range(0, num\_iterations):

Z1, A1, Z2, A2 = forward\_propagation(X, W1, b1, W2, b2)

cost = compute\_cost(Y, A2)

if i % 10 == 0:

costs.append(cost)

dW1, db1, dW2, db2 = backward\_propagation(X, Y, W1, b1, W2, b2, Z1, A1, Z2, A2)

W1, b1, W2, b2 = update\_params(W1, b1, W2, b2, dW1, db1, dW2, db2, learning\_rate)

return W1, b1, W2, b2, costs

def predict(W1, b1, W2, b2, X):

Z1, A1, Z2, A2 = forward\_propagation(X, W1, b1, W2, b2)

m = X.shape[1]

y\_pred = np.zeros((1, m))

for i in range(A2.shape[1]):

if A2[0, i] >= 0.5:

y\_pred[0, i] = 1

else:

y\_pred[0, i] = 0

return y\_pred

def plot\_decision\_boundary(model, X, Y):

# Set min and max values and give it some padding

x\_min, x\_max = X[0, :].min() - 1, X[0, :].max() + 1

y\_min, y\_max = X[1, :].min() - 1, X[1, :].max() + 1

h = 0.01

# Generate a grid of points with distance h between them

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h))

# Predict the function value for the whole grid

Z = model(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

# Plot the contour and training examples

plt.figure(figsize = (10, 6))

plt.contourf(xx, yy, Z, cmap=cmap\_light)

plt.ylabel('x2')

plt.xlabel('x1')

plt.scatter(X[0, :], X[1, :], c = Y, cmap=plt.cm.Spectral)

W1, b1, W2, b2, costs = neural\_network(X, Y, n\_h = 1, learning\_rate= 1, num\_iterations= 1000)

plot\_decision\_boundary(lambda x: predict(W1, b1, W2, b2, x.T), X, Y)

plt.figure(figsize = (10, 6))

plt.title('Cost Function')

plt.xlabel('No. of iterations')

plt.ylabel('Cost')

plt.plot(range(1, 1001, 10), costs)

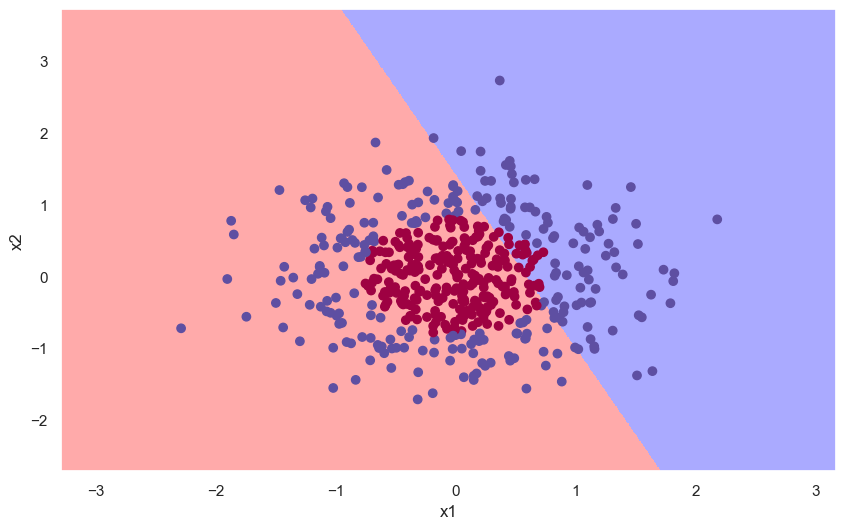
plt.show()

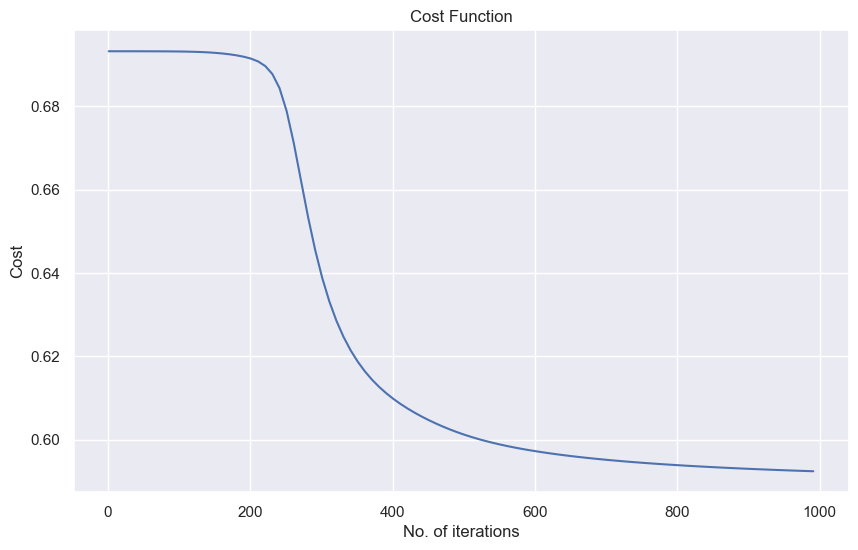
predictions = predict(W1, b1, W2, b2, X)

accuracy = float((np.dot(Y, predictions.T) + np.dot(1-Y,1-predictions.T))/float(Y.size) \* 100)

print (f'Accuracy: {round(accuracy, 2)} %')

Output:-





Accuracy: 64.6 %