In [1]:
```
import warnings                                    #Importing few important python li
warnings.filterwarnings('ignore')
import pandas as pd
import numpy as np
from numpy import linalg as LA
from sklearn.model_selection import train_test_split
import math
import matplotlib.pyplot as plt
import copy
import math
%matplotlib inline
```

In [2]:
```
df = pd.read_csv('heartdataset.csv')
```

In [3]:
```
df.head()
```

Out[3]:

	male	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevalentHyp	diabe
0	1	39	4.0	0	0.0	0.0	0	0	
1	0	46	2.0	0	0.0	0.0	0	0	
2	1	48	1.0	1	20.0	0.0	0	0	
3	0	61	3.0	1	30.0	0.0	0	1	
4	0	46	3.0	1	23.0	0.0	0	0	

In [4]:
```
df.shape
```

Out[4]:
```
(4238, 16)
```

In [5]:
```
df.isnull().sum()
```

Out[5]:
```
male                 0
age                  0
education          105
currentSmoker        0
cigsPerDay          29
BPMeds              53
prevalentStroke      0
prevalentHyp         0
diabetes             0
totChol             50
sysBP                0
diaBP                0
BMI                 19
heartRate            1
glucose            388
TenYearCHD           0
dtype: int64
```

In [6]:
```
#managing the null values
mean_value_glucose = df['glucose'].mean()
df['glucose'].fillna(value=mean_value_glucose, inplace=True)
```

```
df['cigsPerDay'].fillna(value=0.0, inplace=True)
np.random.seed(3)
df['education'].fillna(value=np.random.randint(1,5), inplace=True)
mean_value_chol = df['totChol'].mean()
df['totChol'].fillna(value=mean_value_chol, inplace=True)
df['BPMeds'].fillna(value=np.random.randint(0,2), inplace=True)
df['BMI'].fillna(value=np.random.randint(0,2), inplace=True)
mean_value_rate = df['heartRate'].mean()
df['heartRate'].fillna(value=mean_value_rate, inplace=True)
```

In [7]:

```
#TenYearCHD means 10-year risk of future coronary heart disease
#TenYearCHD is our target from the training data
y = df['TenYearCHD']
X = df.drop('TenYearCHD', axis = 1)
X = X.to_numpy()
y = y.to_numpy()
```

In [8]:

```
def scale_features(X):
    min_vec = X.min(axis = 0)
    max_vec = X.max(axis = 0)
    X = (X - min_vec) / (max_vec - min_vec)
    return X, min_vec, max_vec
```

In [9]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_sta
print("X_train.shape", X_train.shape, "y_train.shape", y_train.shape)
print("X_test.shape", X_test.shape, "y_test.shape", y_test.shape)
```

```
X_train.shape (3178, 15) y_train.shape (3178,)
X_test.shape (1060, 15) y_test.shape (1060,)
```

In [10]:

```
X_train, min_vec, max_vec = scale_features(X_train)
print(X_train)
```

```
[[1.         0.75675676 0.         ... 0.65133929 0.16161616 0.07344633]
 [0.         0.35135135 0.66666667 ... 0.49330357 0.41414141 0.12711864]
 [1.         0.35135135 0.         ... 0.72522321 0.23232323 0.11016949]
 ...
 [1.         0.13513514 0.33333333 ... 0.66629464 0.51515152 0.11855015]
 [0.         0.59459459 0.33333333 ... 0.4921875  0.41414141 0.07627119]
 [0.         0.08108108 0.         ... 0.48058036 0.29292929 0.09887006]]
```

In [11]:

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

In [12]:

```
def compute_cost(X, y, w, b, lambda_= 1):
    m, n = X.shape
    cost = 0
    for i in range(m):
        z = np.dot(X[i],w) + b
        f_wb = sigmoid(z)
        cost += -y[i]*np.log(f_wb) - (1-y[i])*np.log(1-f_wb)
    total_cost = cost/m
    return total_cost
np.random.seed(1)
compute_cost(X_train, y_train, 0.01 * np.random.rand(15), 1)
```

Out[12]:
```
1.1655882597374843
```

In [13]:
```
def compute_gradient(X, y, w, b, lambda_=None):
    m, n = X.shape
    dj_dw = np.zeros(w.shape)
    dj_db = 0.
    for i in range(m):
        f_wb_i = sigmoid(np.dot(X[i],w) + b)
        err_i  = f_wb_i  - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err_i * X[i,j]
        dj_db = dj_db + err_i
    dj_dw = dj_dw/m
    dj_db = dj_db/m
    return dj_db, dj_dw
compute_gradient(X_train, y_train, 0.01 * np.random.rand(15), 1)
```

Out[13]:
```
(0.5771821352171572,
 array([0.22677486, 0.24173117, 0.19894651, 0.28046803, 0.06956737,
        0.01102355, 0.00188616, 0.15066773, 0.00767253, 0.14903905,
        0.16088883, 0.20410731, 0.31590787, 0.18553692, 0.0649162 ]))
```

In [14]:
```
def gradient_descent(X, y, w_in, b_in, alpha, num_iters, cost_func, gradient_func, l

    for i in range(num_iters) :

        dj_db, dj_dw = gradient_func(X, y, w_in, b_in, lambda_)
        w_in = w_in - alpha * dj_dw
        b_in = b_in - alpha * dj_db

#         For 10 iterations equally spaced of total iters, print cost.
        if i % math.ceil(num_iters / 10) == 0 or i == (num_iters - 1) :
            print(f"Iteration {i : 4}: Cost {cost_func(X, y, w_in, b_in) : 8.2f}")

    return w_in, b_in
```

In [15]:
```
#To verify that gradient descent is working properly, it would be great if we can se
#Running the gradient descent to learn the parameters for our dataset
np.random.seed(1)
initial_w = 0.01* (np.random.rand(15) - 0.5)
print(initial_w)
initial_b = -2
num_iters = 1000
alpha = 0.5
lambda_ = 0
w, b = gradient_descent(X_train, y_train, initial_w, initial_b, alpha, num_iters,com
```

```
[-0.00082978  0.00220324 -0.00499886 -0.00197667 -0.00353244 -0.00407661
 -0.0031374  -0.00154439 -0.00103233  0.00038817 -0.00080805  0.0018522
 -0.00295548  0.00378117 -0.00472612]
Iteration    0: Cost     0.44
Iteration  100: Cost     0.40
Iteration  200: Cost     0.40
Iteration  300: Cost     0.39
Iteration  400: Cost     0.39
Iteration  500: Cost     0.39
Iteration  600: Cost     0.39
Iteration  700: Cost     0.39
Iteration  800: Cost     0.39
```

```
Iteration  900: Cost     0.39
Iteration  999: Cost     0.39
```

In [16]:

```
compute_cost(X_train, y_train, w, b)
```

Out[16]:

```
0.3877606709797132
```

In [17]:

```
#scaling test data using normalised features
X_test = (X_test - min_vec) / (max_vec - min_vec)
print(X_test)
```

```
[[1.         0.54054054 0.         ... 0.51941964 0.31313131 0.0960452 ]
 [0.         0.2972973  0.         ... 0.49308036 0.26262626 0.09887006]
 [0.         0.2972973  1.         ... 0.36785714 0.31313131 0.0480226 ]
 ...
 [1.         0.94594595 0.         ... 0.63705357 0.41414141 0.1299435 ]
 [0.         0.75675676 0.33333333 ... 0.         0.36363636 0.11855015]
 [0.         0.54054054 0.         ... 0.5171875  0.12121212 0.07909605]]
```

In [18]:

```
compute_cost(X_test, y_test, w, b)
```

Out[18]:

```
0.3636064072205817
```

In [19]:

```
def predict(X, w, b) :
    m, n = X.shape
    p = np.zeros(m)
    for i in range(m) :
        f_wb = sigmoid(np.dot(X[i], w) + b)
        p[i] = 1 if f_wb > 0.38 else 0
    return p
```

In [20]:

```
p = predict(X_train, w,b)
print('Train Accuracy: %f'%(np.mean(p == y_train) * 100))
p = predict(X_test, w,b)
print('Test Accuracy: %f'%(np.mean(p == y_test) * 100))
```

```
Train Accuracy: 83.668974
Test Accuracy: 85.094340
```

In [21]:

```
from sklearn.preprocessing import StandardScaler
```

In [22]:

```
patient_data = np.array([[1,51,4.0,1,5.0,3.0,0,0,1,400.0,106.0,70.0,26.97,80.0,77.0]
# assume X_train, y_train, X_test, y_test are your training and test data, and w and

# make a prediction for the patient using your trained model
prediction = predict(patient_data, w, b)
print('Prediction: %d'%(prediction))
```

```
Prediction: 1
```