Logistic Regression

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Plotting of Data

In [2]:

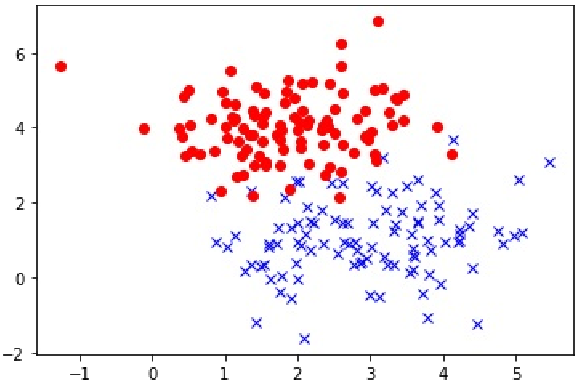
```
c0 = np.zeros(100)
mean0 = [3, 1]
cov0 = [[1, 0], [0, 1]]  # diagonal covariance
c1 = np.ones(100)
mean1 = [2,4]
cov1 = np.array([[1,0],[0,1]])
```

In [3]:

```
x0, y0 = np.random.multivariate_normal(mean0, cov0, 100).T

x1, y1 = np.random.multivariate_normal(mean1, cov1, 100).T
# print(type(x0))
x0,y0

X1 = np.r_[x0,x1]
X2 = np.r_[y0,y1]
m = len(X1)
X = np.c_[X1,X2]
y = np.asarray([np.r_[c0,c1]]).T
plt.plot(x0, y0, 'bx')
plt.plot(x1, y1, 'or')
plt.show()
```



Sigmoid function

$ g(z) = \frac{1}{(1+e^{-z})}$

In [4]:

```
def sigmoid(z):

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

In [5]:

```
# testing the sigmoid function
sigmoid(0)
```

Out[5]:

0.5

Compute the Cost Function and Gradient

$$J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} [-y^{(i)}log(h_{\Theta}(x^{(i)})) - (1 - y^{(i)})log(1 - (h_{\Theta}(x^{(i)}))] -- minimize $$$$J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} [y^{(i)}log(h_{\Theta}(x^{(i)})) + (1 - y^{(i)})log(1 - (h_{\Theta}(x^{(i)}))] -- maximize $$$$ \frac{\partial J(\Theta)}{\partial \Theta_j} = \frac{1}{m} \sum_{i=1}^{m} (h_{\Theta}(x^{(i)}) - y^{(i)})x_j^{(i)} -- gradient \space descent$$$$ \frac{\partial J(\Theta)}{\partial \Theta_j} = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - h_{\Theta}(x^{(i)}))x_j^{(i)} -- gradient \space ascent$$

In [6]:

```
def cost_gradFunction(theta, X, y):

    m=len(y)

    predictions = sigmoid(np.dot(X,theta))

    error = (y * np.log(predictions)) + ((1-y)*np.log(1-predictions))

    cost = 1/m * sum(error)

    grad = 1/m * np.dot(X.transpose(),(y - predictions))

    return cost[0] , grad
```

In [7]:

```
# a = np.array([[1,2,1], [1, 3, 1], [1, 1, 2],[1,3,2]])
# b = np.array([[0,0,1,1]])
# t = np.array([[0,0,0]])
# print(t.shape)
# print(a.shape)
# print(b.shape)

# cost, grad = cost_gradFunction(t.T, a, b)
# cost
# np.sum(a)
```

In [8]:

```
m , n = X.shape[0], X.shape[1]
X= np.append(np.ones((m,1)),X,axis=1)
X.shape
```

Out[8]:

```
(200, 3)
```

In [9]:

```
initial_theta = np.zeros((n+1,1))
initial_theta.shape
```

Out[9]:

```
(3, 1)
```

In [10]:

```
init_cost, grad= cost_gradFunction(initial_theta,X,y)
print("Cost of initial theta is",init_cost)
print("Gradient at initial theta (zeros):",grad)
```

```
Cost of initial theta is -0.6931471805599465
Gradient at initial theta (zeros): [[ 0.        ]
 [-0.26612039]
 [ 0.74386112]]
```

Gradient Ascent

In [11]:

```
J_history =[]
lr = 0.001
new_theta = initial_theta + (lr*grad)
J_history.append(init_cost)
new_theta
```

Out[11]:

```
array([[ 0.        ],
       [-0.00026612],
       [ 0.00074386]])
```

In [12]:

```
def gradientAscent(X,y,theta,alpha,num_iters):

    m=len(y)
    iterations = 0
    for i in range(num_iters):
        iterations += 1
        cost, grad = cost_gradFunction(theta,X,y)
        theta = theta + (alpha * grad)
        J_history.append(cost)
        if abs(J_history[-2] - J_history[-1]) < 0.00001:
            break


    return theta , J_history, iterations
```

In [13]:

```
theta , J_history, iterations= gradientAscent(X,y,new_theta,lr,7000)
print(iterations)
```

```
6697
```

In [14]:

```
print("Theta optimized by gradient descent:",theta)
print("The cost of the optimized theta:",J_history[-1])
```

```
Theta optimized by gradient descent: [[-0.27110396]
 [-0.9939407 ]
 [ 1.13575164]]
The cost of the optimized theta: -0.19522243252064783
```

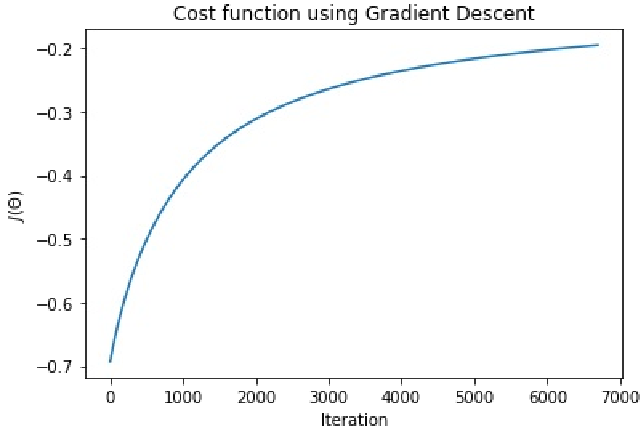
Plotting of Cost Function

In [15]:

```
plt.plot(J_history)
plt.xlabel("Iteration")
plt.ylabel("$J(\Theta)$")
plt.title("Cost function using Gradient Descent")
```

Out[15]:

```
Text(0.5, 1.0, 'Cost function using Gradient Descent')
```



Plotting the decision boundary

$h_\Theta(x) = g(z)$, where g is the sigmoid function and $z = \Theta^Tx$

Since $h_\Theta(x) \geq 0.5$ is interpreted as predicting class "1", $g(\Theta^Tx) \geq 0.5$ or $\Theta^Tx \geq 0$ predict class "1"

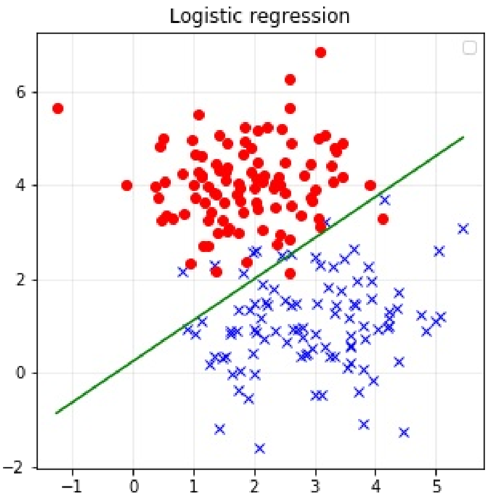
$\Theta_1 + \Theta_2x_2 + \Theta_3x_3 = 0$ is the decision boundary

Since, we plot x_2 against x_3, the boundary line will be the equation $ x_3 = \frac{-(\Theta_1+\Theta_2x_2)}{\Theta_3}$

In [16]:

```
fig1 = plt.figure(figsize=(5,5))
ax = plt.axes()
plt.title('Logistic regression')
plt.grid(axis='both', alpha=.25)
plt.plot(x0, y0, 'bx')
plt.plot(x1, y1, 'or')
x_value= np.array([np.min(X[:,1]),np.max(X[:,1])])
y_value=-(theta[0] +theta[1]*x_value)/theta[2]
plt.plot(x_value,y_value, "g")
plt.legend(loc=0)
plt.show()
```

```
No handles with labels found to put in legend.
```



Prediction

In [17]:

```
def classifierPredict(theta,X):
    predictions = X.dot(theta)
    return predictions>0

def returnClass(a):
    if(a<=0.5):
        return 'x'
    else:
        return 'o'

xtest1 = np.array([1,0])
xtest1 = np.r_[np.ones(1), xtest1]
print("Point(", xtest1[1],",", xtest1[2] ,") belongs to Class '%s'"   % returnClass(sigmoid(xtest1.dot(theta
)[0])))

xtest2 = np.array([4,6])
xtest2 = np.r_[np.ones(1), xtest2]
print("Point(", xtest2[1],",", xtest2[2] ,") belongs to Class '%s'"   % returnClass(sigmoid(xtest2.dot(theta
)[0])))

xtest3 = np.array([0,1.5])
xtest3 = np.r_[np.ones(1), xtest3]
print("Point(", xtest3[1],",", xtest3[2] ,") belongs to Class '%s'"   % returnClass(sigmoid(xtest3.dot(theta
)[0])))

xtest4 = np.array([6,4])
xtest4 = np.r_[np.ones(1), xtest4]
print("Point(", xtest4[1],",", xtest4[2] ,") belongs to Class '%s'"   % returnClass(sigmoid(xtest4.dot(theta
)[0])))
```

```
Point( 1.0 , 0.0 ) belongs to Class 'x'
Point( 4.0 , 6.0 ) belongs to Class 'o'
Point( 0.0 , 1.5 ) belongs to Class 'o'
Point( 6.0 , 4.0 ) belongs to Class 'x'
```

In [ ]: