In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
import matplotlib.mlab as ml
from scipy.interpolate import griddata
```

HW2 Q2

Using real data from https://github.com/gditzler/UA-ECE-523-Sp2018/tree/master/data

In [2]:
```
path = '/home/safwan/Documents/spring2021/ece523/hw/hw2/acute-nephritis.csv'
data_pd = pd.read_csv(path, header=None)
```

In [3]:
```
data_pd
```

Out[3]:

	0	1	2	3	4	5	6
0	-1.77236	-0.562162	0.841625	-1.408310	-0.979364	-0.841625	0
1	-1.55248	-0.562162	-1.178280	0.704154	1.012560	1.178280	0
2	-1.55248	-0.562162	0.841625	-1.408310	-0.979364	-0.841625	0
3	-1.49751	-0.562162	-1.178280	0.704154	1.012560	1.178280	0
4	-1.49751	-0.562162	0.841625	-1.408310	-0.979364	-0.841625	0
...	...	...	...	...	...	...	...
115	1.47094	-0.562162	0.841625	0.704154	-0.979364	1.178280	1
116	1.52591	-0.562162	-1.178280	-1.408310	-0.979364	-0.841625	0
117	1.52591	1.764020	0.841625	-1.408310	1.012560	-0.841625	1
118	1.52591	-0.562162	0.841625	0.704154	-0.979364	1.178280	1
119	1.52591	-0.562162	0.841625	0.704154	-0.979364	1.178280	1

120 rows × 7 columns

In [4]:
```
#seperating the features and labels.
x = data_pd.iloc[:, 0:-1]
y = data_pd.iloc[:, -1]
#conveting to array
x_arr = x.values
y_arr = y.values
```

In [5]:
```
# Adding 1 at the begining of every feature vector
x_arr = np.c_[np.ones((x_arr.shape[0], 1)), x_arr]
y_arr = y_arr[:, np.newaxis]
w = np.zeros((x_arr.shape[1],1)) #initializing the parameter vector w.
```

In [37]:
```
#creating training test date
X_train, X_test, y_train, y_test = \
                    train_test_split(x_arr, y_arr, test_size=0.20)
```

In [38]:
```
#creating the logistic function
def logistic_func(w, x):
    #x is a feature vector
    #w is the parameter vector
    regression = np.dot(x, w) #finding weighted sum of inputs
    result = 1 / (1 + np.exp(- regression))
    return result
```

In [39]:
```
#creating cross entropy function
def cross_entropy_func(w , x, y):
    result = - np.sum (y * np.log(logistic_func(w,x)) + (1-y)* np.log(1- logisti
    return result
```

In [40]:
```
#gradient function
def gradient_func(w, x, y, learning_rate):
    result = np.dot(x.T, logistic_func(w, x) - y)
    return learning_rate * result
```

In [41]:
```
#gradient descent function
def sgd(w,x,y,iterations, learning_rate):
    m = len(y) # size of the training dataset
    cost_history = []
    for _ in range(iterations):

        for j in range(m): #loop through every sample
            x_i = x[1,:].reshape((1,len(w)))
            y_i = y[1,:].reshape((1,1))
            w = w - gradient_func(w,x_i,y_i, learning_rate)
            cost = cross_entropy_func(w, x_i, y_i)
        cost_history.append(cost)

    return w, cost_history
```

In [ ]:

In [42]:
```
#testing the model
train = sgd(w, X_train, y_train,500, 0.005)
```

In [43]:
```
#the optimized parameters
w_t = train[0]
w_t
```

Out[43]:
```
array([[0.86030556],
       [0.93441229],
       [1.51759622],
       [0.72405467],
       [0.6057876 ],
       [0.871111  ],
       [1.01368084]])
```

In [44]:
```
#showing some values of the cost function
train[1][:10]
```

Out[44]:
```
[0.2267844997750445,
 0.12419529120361983,
 0.08384644412720996,
 0.06284558529725653,
 0.0500969894039993,
 0.041576957340538706,
 0.03549744285766436,
 0.030948776299568707,
 0.027421255026845663,
 0.024607842100965876]
```

In [45]:
```
#testing with new data
predict = logistic_func(w_t, X_test)
# predict
```

In [46]:
```
#finding the accuracy
for i in range(len(predict)):
    if predict[i] >= 0.5:
        predict[i] = 1
    else:
        predict[i] = 0
# predict= 0
diff = predict - y_test
accuracy = (1.0 - (float(np.count_nonzero(diff)) / len(diff)))*100
accuracy
```

Out[46]:
```
83.33333333333334
```

In [16]:
```
# predict.shape
```

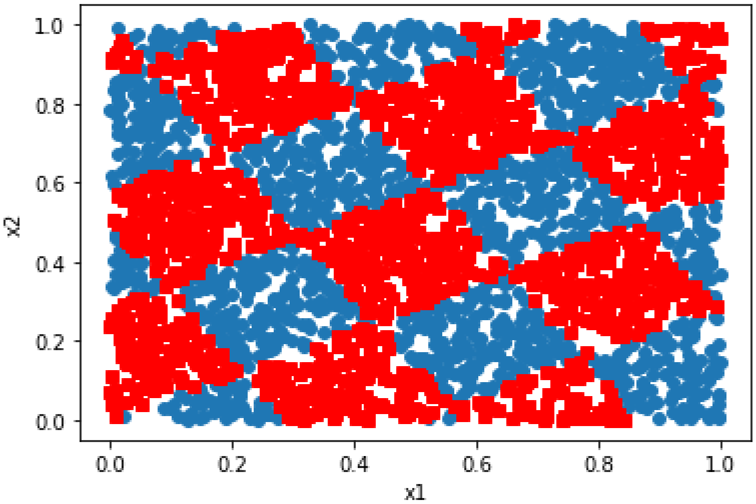
HW2 Q3

Density estimation

In [17]:
```
#generating training dataset
def gen_cb(N, a, alpha):
    """
    N: number of points on the checkerboard
    a: width of the checker board (0<a<1)
    alpha: rotation of the checkerboard in radians
    """
    d = np.random.rand(N, 2).T  # THIS IS THE LINE OF CODE THAT IS DIFFERENT
    d_transformed = np.array([d[0]*np.cos(alpha)-d[1]*np.sin(alpha),
                              d[0]*np.sin(alpha)+d[1]*np.cos(alpha)]).T
    s = np.ceil(d_transformed[:,0]/a)+np.floor(d_transformed[:,1]/a)
    lab = 2 - (s%2)
    data = d.T
    return data, lab
```

In [18]:
```
X, y = gen_cb(3000, .25, np.pi / 3)
plt.figure()
plt.plot(X[np.where(y==1)[0], 0], X[np.where(y==1)[0], 1], 'o')
plt.plot(X[np.where(y==2)[0], 0], X[np.where(y==2)[0], 1], 's', c = 'r')
plt.xlabel("x1")
plt.ylabel("x2")
```

Out[18]:
```
Text(0, 0.5, 'x2')
```



In [19]:
```
#creating a function for implementing k nearest neighbor (knn)
def knn(k, X ,y, point):
    x1, x2 = X[y==1], X[y==2] #x1blue x2red
    n1 = len(x1)
    n2 = len(x2)

    dataset = X.tolist()
    for i in range(len(dataset)):
        dataset[i].append(y[i])

    n=len(dataset)
    dist_list = []
    for i in dataset:
        dist = np.linalg.norm(point-np.array(i[:-1])) #find the euclidean distar
        i.append(dist)
    dataset.sort(key=lambda tup: tup[3]) #sort
    k_nearest = dataset[:k] #take only the first k elements
    largest_k = max(k_nearest, key=lambda x: x[3])

    #now find how many neighbor in class blue 1
    k_1 = [x for x in k_nearest if x[2] == 1.0]

    #now find how many neighbor in class blue 2
    k_2 = [x for x in k_nearest if x[2] == 2.0]

    #calculating the volume
    r = largest_k[-1]

    #radius of the circle
    v = np.pi * (r**2)

    #p(x|y=blue)
    pb = (len(k_1)/(n1*v))

    #p(x|y=red)
    pr = (len(k_2)/(n2*v))

    #p(x)
    px= len(k_nearest)/(n*v)

    return pr,pb,px
```

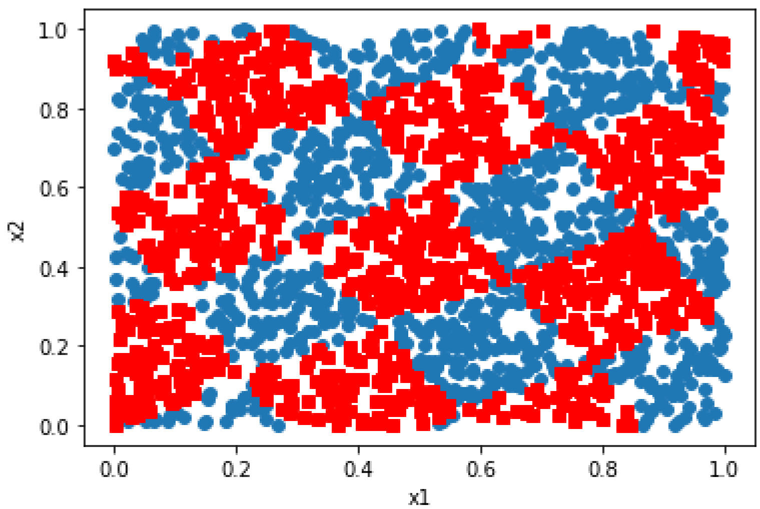
computing the liklihood

In [20]:
```
x1, x2 = X[y==1], X[y==2] #x1blue x2red
#py(blue)
p_blue = len(x1)/len(X)
#py(red)
p_red = len(x2)/len(X)
```

generating test dataset

In [21]:
```
X_t, y_t = gen_cb(2000, .25, np.pi / 3)
plt.figure()
plt.plot(X_t[np.where(y_t==1)[0], 0], X_t[np.where(y_t==1)[0], 1], 'o')
plt.plot(X_t[np.where(y_t==2)[0], 0], X_t[np.where(y_t==2)[0], 1], 's', c = 'r')
plt.xlabel("x1")
plt.ylabel("x2")
```

Out[21]: Text(0, 0.5, 'x2')



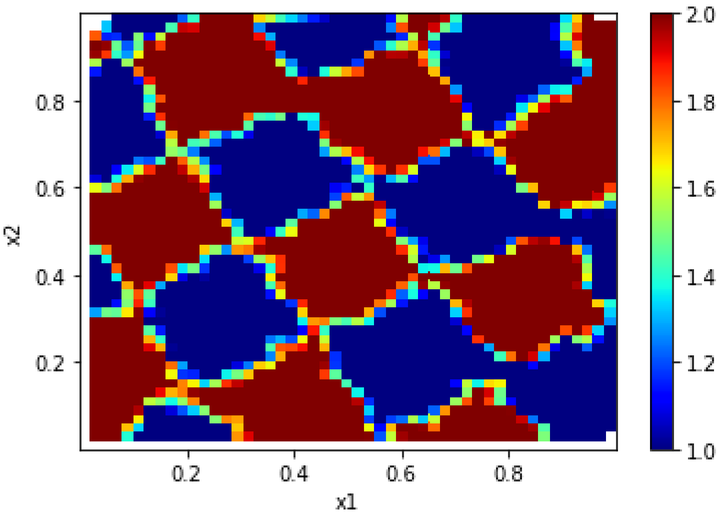
In [22]:
```
#testing the model
result = X_t.tolist()
for i in range(len(X_t)):
    px_r,px_b,px = knn(15,X, y, X_t[i,:])
    p_1 = px_b*p_blue/px
    p_2 = px_r*p_red/px
    if p_1 > p_2:
        result[i].append(1.0)
    else:
        result[i].append(2.0)
```

In [23]:
```
#convert to np array
r = np.array(result)
x_r = r[:,:2]
y_r = r[:,-1]
```

In [24]:
```
def plot_contour(x,y,z,resolution = 50,contour_method='linear'):
    resolution = str(resolution)+'j'
    X,Y = np.mgrid[min(x):max(x):complex(resolution),   min(y):max(y):complex(re
    points = [[a,b] for a,b in zip(x,y)]
    Z = griddata(points, z, (X, Y), method=contour_method)
```

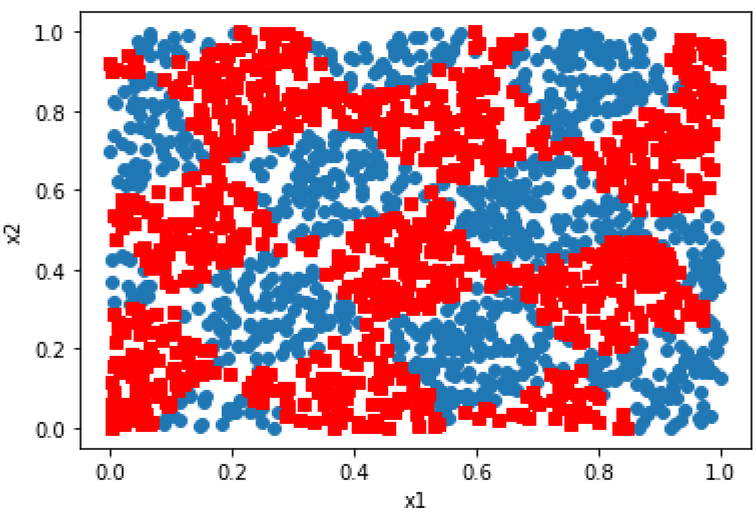
```
    return X,Y,Z

X,Y,Z = plot_contour(r[:,0],r[:,1],r[:,2],resolution = 50,contour_method='linear
```

In [25]:
```
#ploting P(x|y)with pcolor
plt.pcolor(X,Y, Z, cmap = 'jet')
plt.colorbar()
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
```



In [26]:
```
#ploting P(x|y)with plot
plt.plot(x_r[np.where(y_r==1)[0], 0], x_r[np.where(y_r==1)[0], 1], 'o')
plt.plot(x_r[np.where(y_r==2)[0], 0], x_r[np.where(y_r==2)[0], 1], 's', c = 'r')
plt.xlabel("x1")
plt.ylabel("x2")
```

Out[26]: Text(0, 0.5, 'x2')



In [ ]: