Supervised classification - improving capacity learning

0. Import library

Import library

In [2]:
```
# Import libraries

# math library
import numpy as np

# visualization library
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png2x','pdf')
import matplotlib.pyplot as plt

# machine learning library
from sklearn.linear_model import LogisticRegression

# 3d visualization
from mpl_toolkits.mplot3d import axes3d

# computational time
import time

import math
```

1. Load and plot the dataset (dataset-noise-02.txt)

The data features for each data i are $x_i = (x_{i(1)}, x_{i(2)})$.

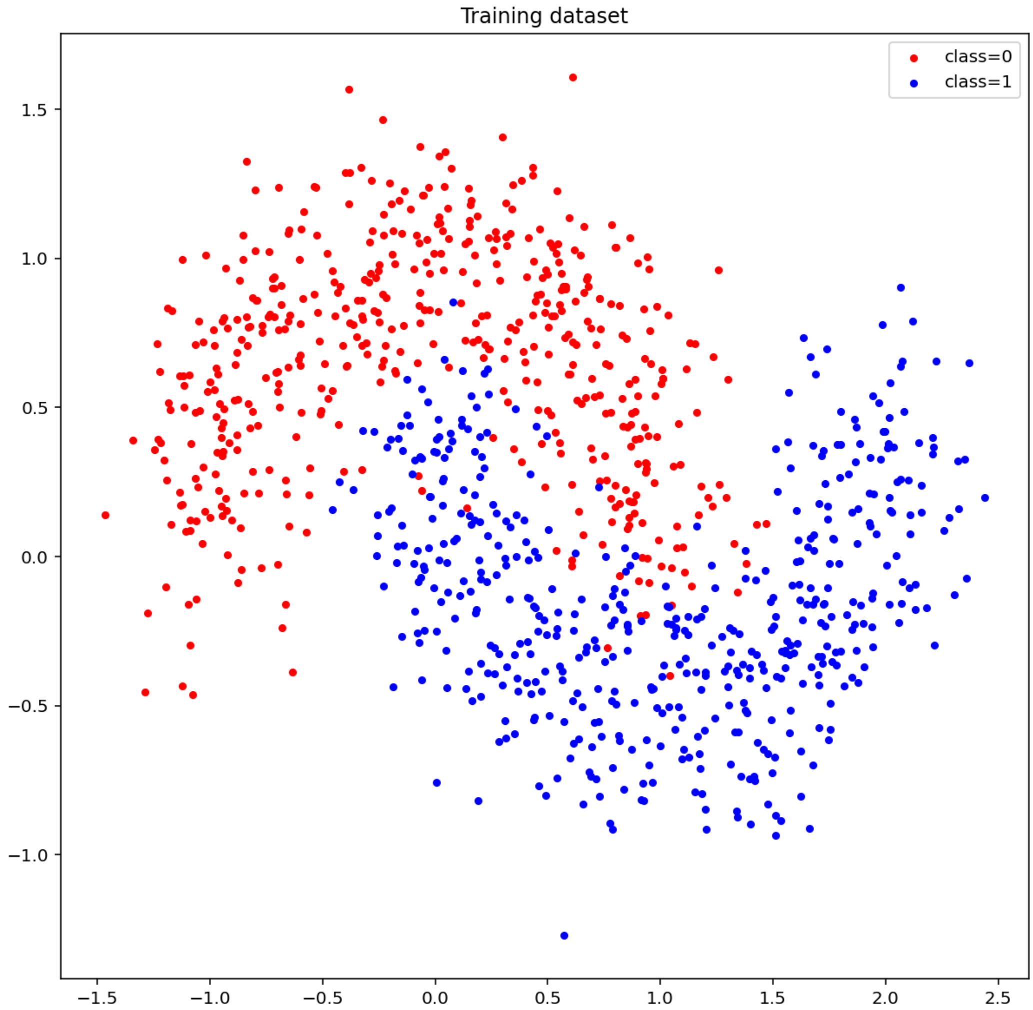
The data label/target, y_i, indicates two classes with value 0 or 1.

Plot the data points.

You may use matplotlib function `scatter(x,y)` .

In [8]:
```
# import data with numpy
data = np.loadtxt('dataset-b.txt', delimiter=',')

# number of training data
n = data.shape[0]
print('Number of the data = {}'.format(n))
print('Shape of the data = {}'.format(data.shape))
print('Data type of the data = {}'.format(data.dtype))

# plot
x1 = data[:,0] # feature 1
x2 = data[:,1] # feature 2
idx = data[:,2] # label

idx_class0 = (idx == 0) # index of class0
idx_class1 = (idx == 1) # index of class1

plt.figure(1,figsize=(10,10))
plt.scatter(x1[idx_class0], x2[idx_class0], s=50, c='r', marker='.', label='class=0')
plt.scatter(x1[idx_class1], x2[idx_class1], s=50, c='b', marker='.', label='class=1')
plt.title('Training dataset')
plt.legend(loc = 'upper right')
plt.show()
```

```
Number of the data = 1000
Shape of the data = (1000, 3)
Data type of the data = float64
```



2. Define a logistic regression loss function and its gradient

In [11]:

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


# predictive function definition
def f_pred(X,w):
    p = sigmoid(np.dot(X, w))
    return p

# loss function definition
def loss_logreg(y_pred,y):
    n = len(y)
    epsilon = 1e-3
    loss = np.dot(-(y.T), np.log(y_pred + epsilon)) - np.dot((1-y).T, np.log(1 - y_pred + epsilon))
    return loss / n


# gradient function definition
def grad_loss(y_pred,y,X):
    n = len(y)
    return (2 * np.dot(X.T, y_pred - y)) / n


# gradient descent function definition
def grad_desc(X, y , w_init, tau, max_iter):

    L_iters = np.zeros([max_iter]) # record the loss values
    w = w_init # initialization
    for i in range(max_iter): # loop over the iterations
        y_pred = f_pred(X, w)# linear predicition function
        grad_f = grad_loss(y_pred, y, X)# gradient of the loss
```

```
        w = w - tau * grad_f # update rule of gradient descent
        L_iters[i] = loss_logreg(y_pred, y) # save the current loss value

    return w, L_iters
```

3. define a prediction function and run a gradient descent algorithm

The logistic regression/classification predictive function is defined as:

$$p_w(x) = \sigma(Xw)$$

The prediction function can be defined in terms of the following feature functions f_i as follows:

$$X = \begin{bmatrix} f_0(x_1) & f_1(x_1) & f_2(x_1) & f_3(x_1) & f_4(x_1) & f_5(x_1) & f_6(x_1) & f_7(x_1) & f_8(x_1) & f_9(x_1) \\ f_0(x_2) & f_1(x_2) & f_2(x_2) & f_3(x_2) & f_4(x_2) & f_5(x_2) & f_6(x_2) & f_7(x_2) & f_8(x_2) & f_9(x_2) \\ \vdots \\ f_0(x_n) & f_1(x_n) & f_2(x_n) & f_3(x_n) & f_4(x_n) & f_5(x_n) & f_6(x_n) & f_7(x_n) & f_8(x_n) & f_9(x_n) \end{bmatrix} \quad \text{and} \quad u$$

where $x_i = (x_i(1), x_i(2))$ and you can define a feature function f_i as you want.

You can use at most 10 feature functions f_i, $i = 0, 1, 2, \cdots, 9$ in such a way that the classification accuracy is maximized. You are allowed to use less than 10 feature functions.

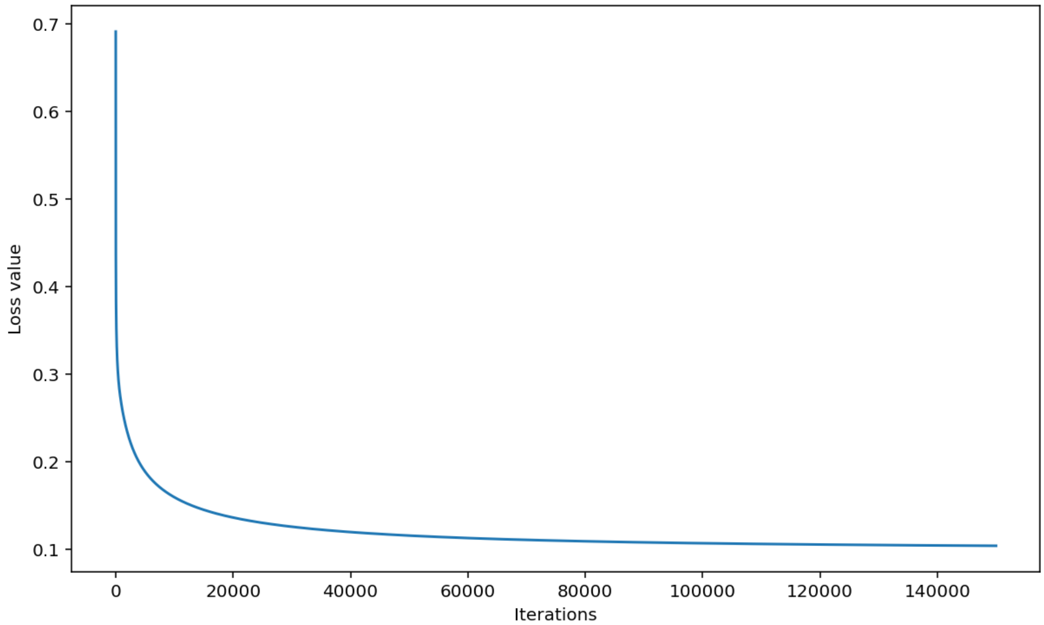
Implement the logistic regression function with gradient descent using a vectorization scheme.

In [461]:

```
def featureFunction(x1, x2, n):

    result = np.ones([n, 10])

    result[:, 0] = 1
    result[:, 1] = x1
    result[:, 2] = x2
    result[:, 3] = x1 - x2
    result[:, 4] = x1 ** 2
    result[:, 5] = x2 ** 2
    result[:, 6] = (x1 ** 2) - (x2 ** 2)
    result[:, 7] = (x1 - x2) ** 3
    result[:, 8] = (x1 ** 3) * (x2 ** 2)
    result[:, 9] = (x1 ** 3) - (x2 ** 4)

    return result

# construct the data matrix X, and label vector y
n = data.shape[0]
X = featureFunction(x1, x2, n)

y = data[:,2][:,None] # label

# run gradient descent algorithm
start = time.time()
w_init = np.array([0 for i in range(X.shape[1])])[:,None]
tau = 0.01; max_iter = 150000
w, L_iters = grad_desc(X, y , w_init, tau, max_iter)

print(L_iters[max_iter-1])
print(w)

# plot
```

```python
plt.figure(3, figsize=(10,6))
plt.plot(np.array(range(max_iter)), L_iters)
plt.xlabel('Iterations')
plt.ylabel('Loss value')
plt.show()
```

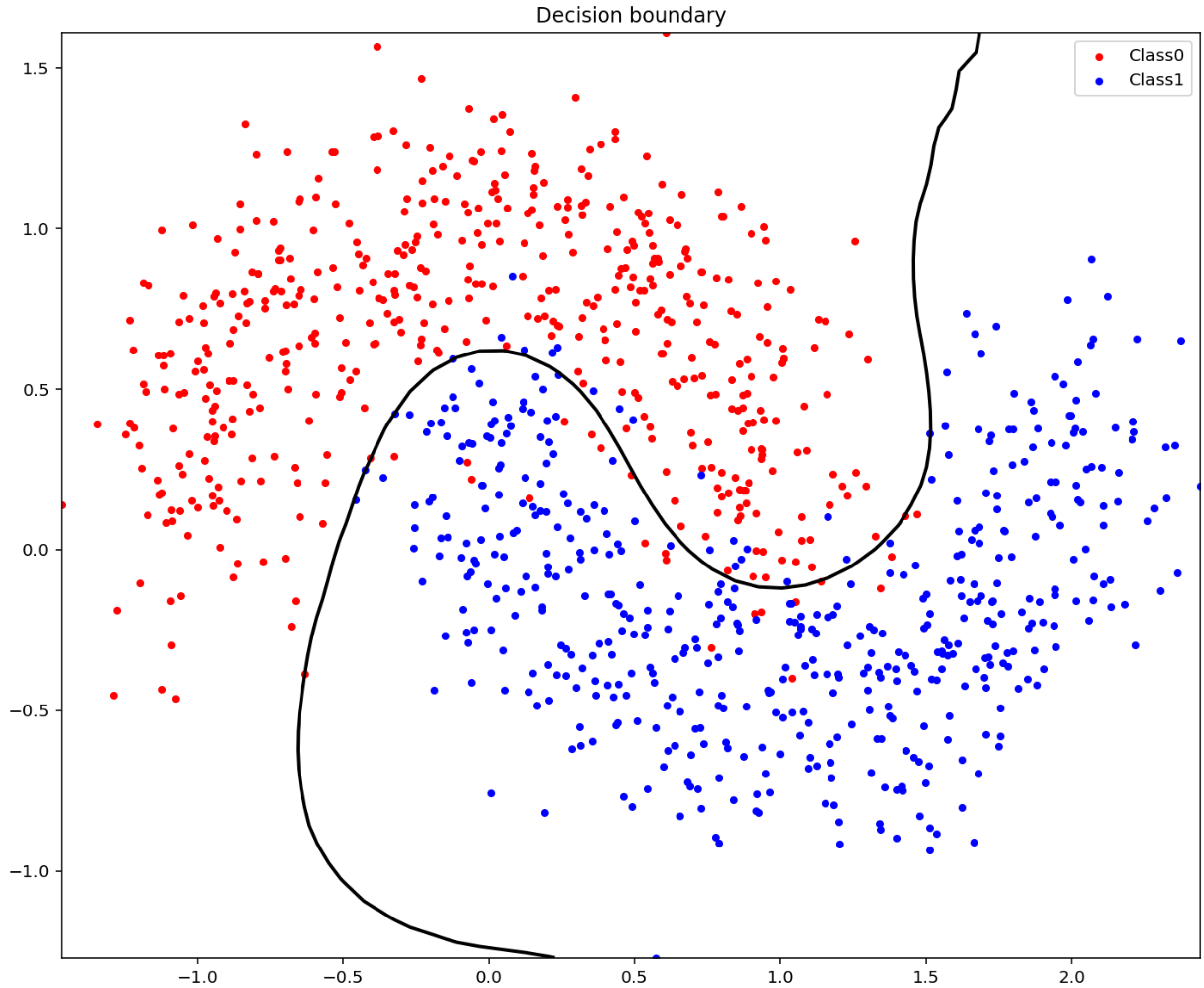
```
0.10407759083404299
[[ 4.26428579]
 [-2.80735735]
 [-4.30054546]
 [ 1.49318812]
 [-8.21892331]
 [-3.03982439]
 [-5.17909892]
 [ 1.55121609]
 [ 4.36676159]
 [ 7.47943334]]
```



4. Plot the decisoin boundary

In [465]:

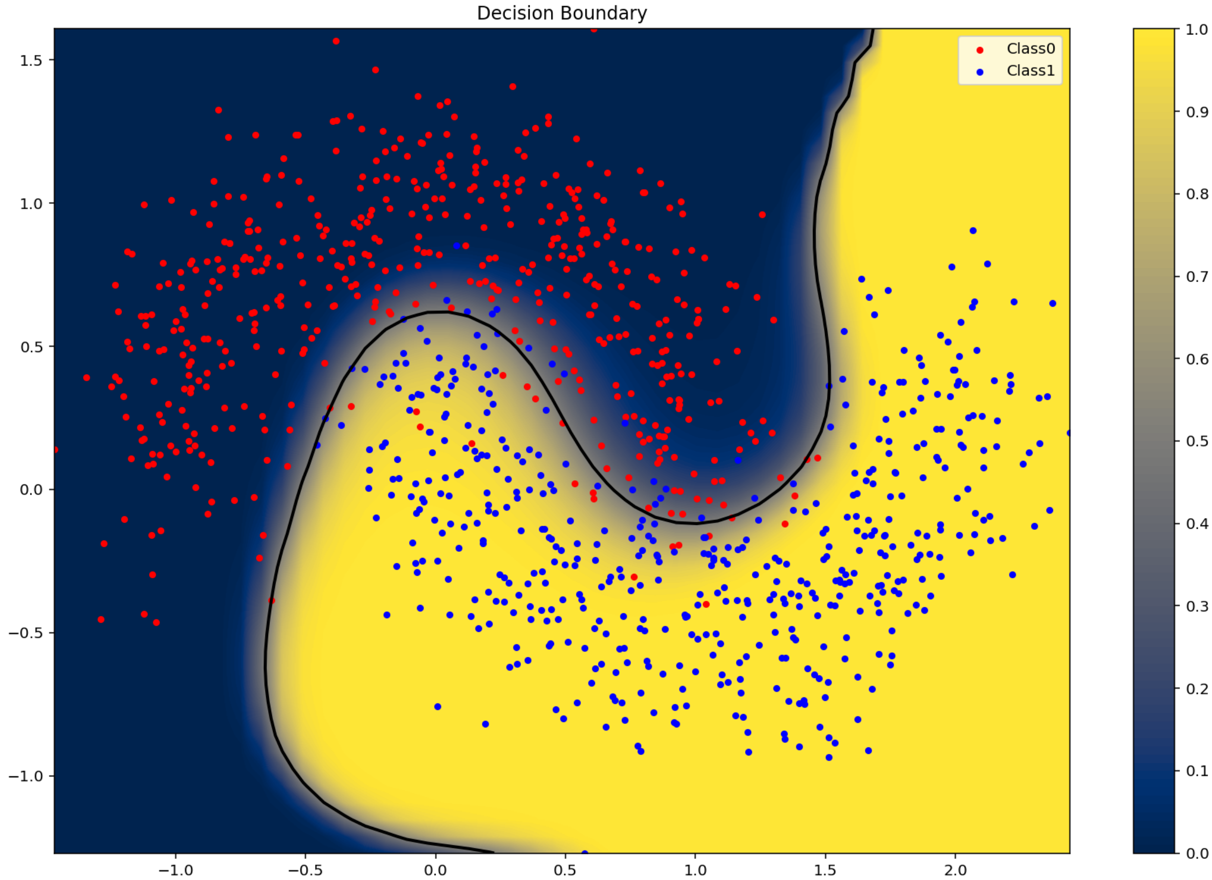
```python
# compute values p(x) for multiple data points x
x1_min, x1_max = x1.min(), x1.max() # min and max of grade 1
x2_min, x2_max = x2.min(), x2.max() # min and max of grade 2
xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max), np.linspace(x2_min, x2_max)) # create meshgrid
X2 = featureFunction(xx1.reshape(-1), xx2.reshape(-1), xx1.size)

p = f_pred(X2, w)
p = p.reshape(50, -1)

# plot
plt.figure(4,figsize=(12,10))

#ax = plt.contourf(xx1,xx2,p,100,vmin=0,vmax=1,cmap='coolwarm', alpha=0.6)
#cbar = plt.colorbar(ax)
#cbar.update_ticks()

plt.scatter(x1[idx_class0], x2[idx_class0], s=50, c='r', marker='.', label='Class0')
plt.scatter(x1[idx_class1], x2[idx_class1], s=50, c='b', marker='.', label='Class1')
plt.contour(xx1, xx2, p, levels = [0.5], linewidths=2, colors='k')
plt.legend(loc = 'upper right')
plt.title('Decision boundary')
plt.show()
```



5. Plot the probability map

In [471]:

```
# compute values p(x) for multiple data points x
x1_min, x1_max = x1.min(), x1.max() # min and max of grade 1
x2_min, x2_max = x2.min(), x2.max() # min and max of grade 2
xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max), np.linspace(x2_min, x2_max))# create meshgrid

X2 = featureFunction(xx1.reshape(-1), xx2.reshape(-1), xx1.size)

p = f_pred(X2, w)
p = p.reshape(50, -1)

# plot
plt.figure(4,figsize=(15,10))

cf = plt.contourf(xx1, xx2, p, cmap = 'cividis', levels = 100)
cbar = plt.colorbar(cf, ticks = [0.00, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80, 0.90, 1.00])
cbar.update_ticks()

plt.scatter(x1[idx_class0], x2[idx_class0], s=50, c='r', marker='.', label='Class0')
plt.scatter(x1[idx_class1], x2[idx_class1], s=50, c='b', marker='.', label='Class1')
plt.contour(xx1, xx2, p, levels = [0.5], linewidths=2, colors='k')
plt.legend()
plt.title('Decision Boundary')
plt.show()
```



6. Compute the classification accuracy

The accuracy is computed by:

$$\text{accuracy} = \frac{\text{number of correctly classified data}}{\text{total number of data}}$$

In [476]:

```
# compute the accuracy of the classifier
n = data.shape[0]

# plot
x1 = data[:,0]  # feature 1
x2 = data[:,1]  # feature 2
idx = data[:,2] # label

idx_class0 = (idx == 0)# index of class0
idx_class1 = (idx == 1) # index of class1

X3 = featureFunction(x1, x2, n)
p3 = f_pred(X3, w)

idx_class0_pred = (p3 <= 0.5)
idx_class1_pred = (p3 > 0.5)

idx_class0_correct = 0
idx_class1_correct = 0
for i in range(idx.size):
    if idx_class0[i] == idx_class0_pred[i] == True :
        idx_class0_correct += 1

    if idx_class1[i] == idx_class1_pred[i] == True:
        idx_class1_correct += 1

accuracy = ((idx_class0_correct + idx_class1_correct) / n) * 100

#print(np.sum(idx_wrong))
print('total number of data = ', (n))
```

```
print('total number of correctly classified data = ', (idx_class0_correct + idx_class1_correct))
print('accuracy(%) = ',  accuracy)
```

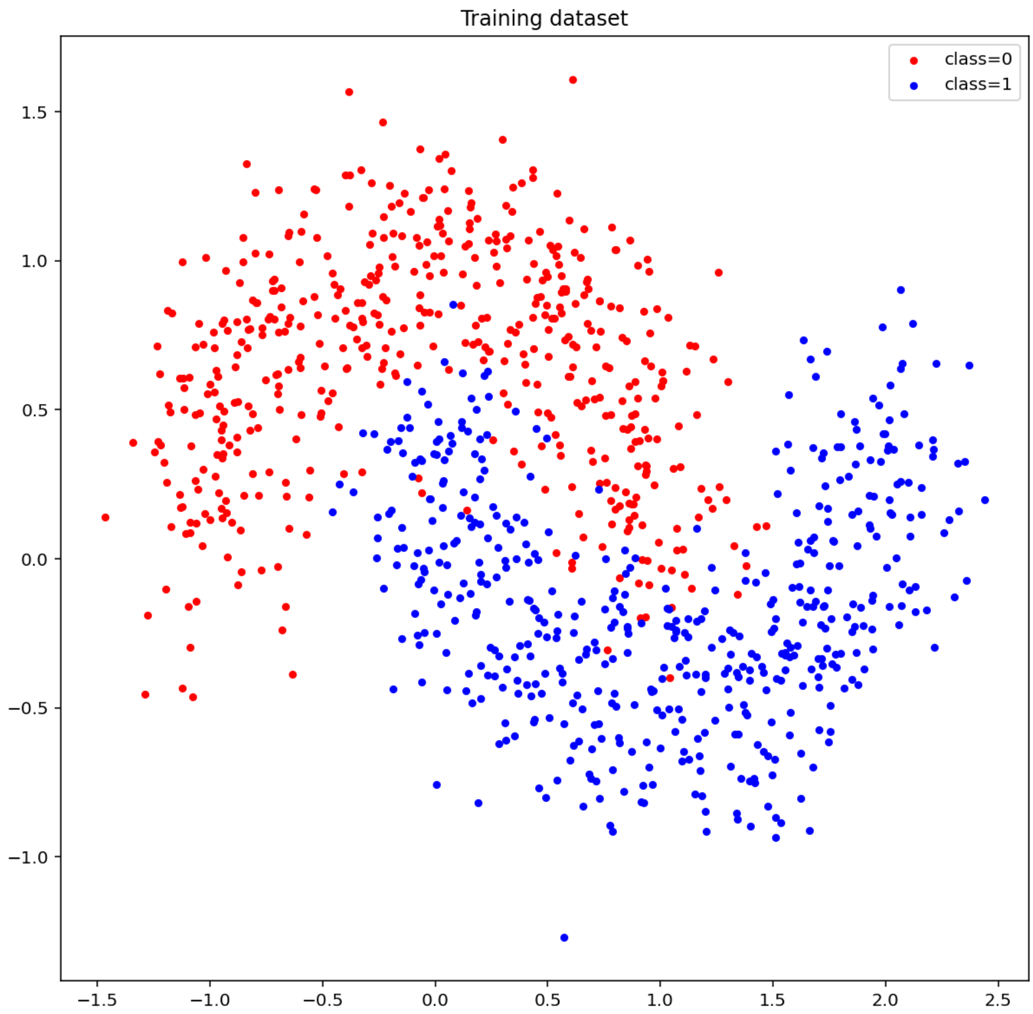
```
total number of data =  1000
total number of correctly classified data =  960
accuracy(%) =  96.0
```

Output using the dataset (dataset-noise-02.txt)

1. Visualize the data [1pt]

In [414]:

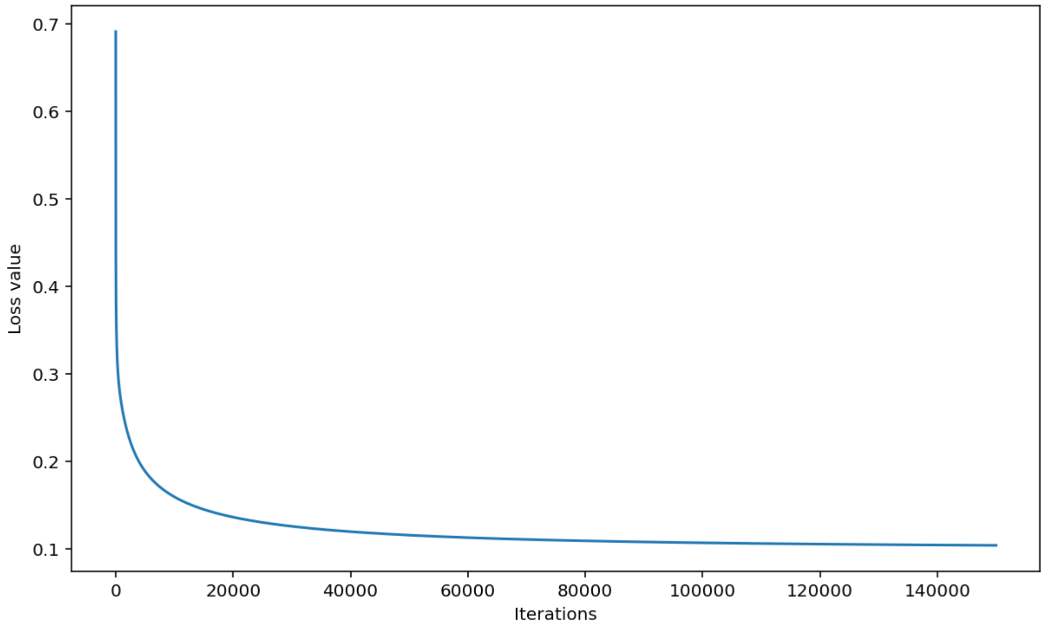
```
plt.figure(1,figsize=(10,10))
plt.scatter(x1[idx_class0], x2[idx_class0], s=50, c='r', marker='.', label='class=0')
plt.scatter(x1[idx_class1], x2[idx_class1], s=50, c='b', marker='.', label='class=1')
plt.title('Training dataset')
plt.legend(loc = 'upper right')
plt.show()
```



2. Plot the loss curve obtained by the gradient descent until the convergence [2pt]

In [464]:

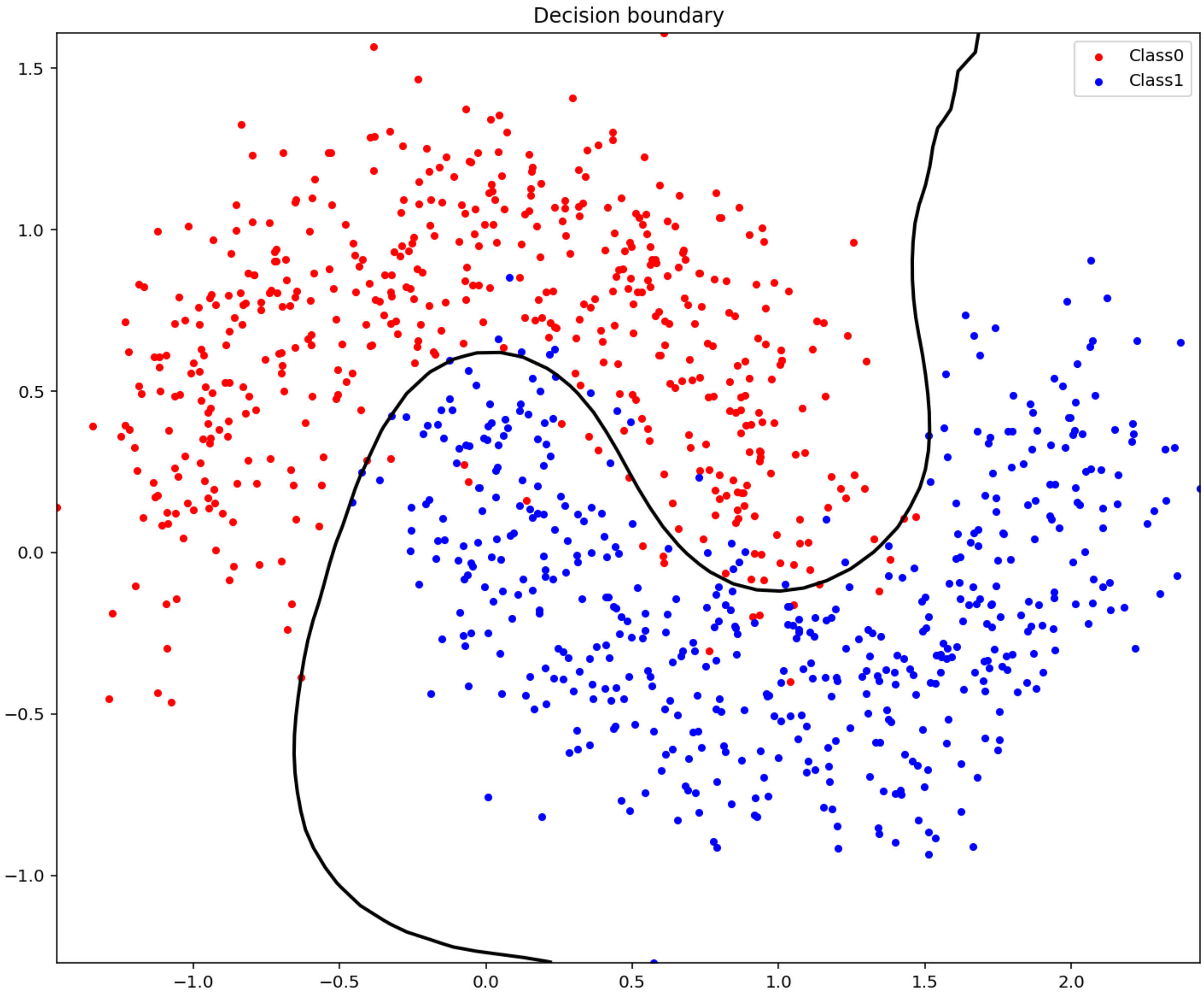
```
# plot
plt.figure(3, figsize=(10,6))
plt.plot(np.array(range(max_iter)), L_iters)
plt.xlabel('Iterations')
plt.ylabel('Loss value')
plt.show()
```



3. Plot the decisoin boundary of the obtained classifier [2pt]

In [467]:

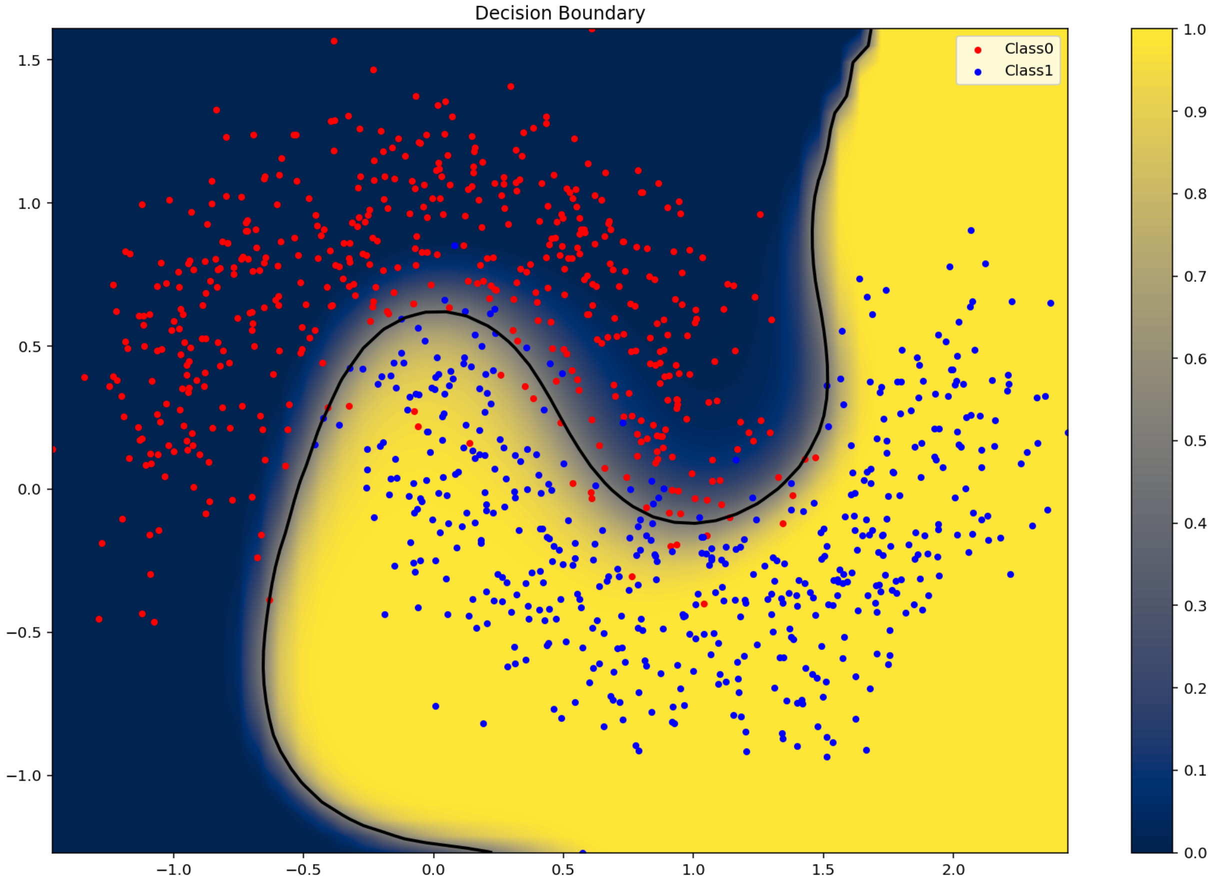
```
# plot
plt.figure(4,figsize=(12,10))

#ax = plt.contourf(xx1,xx2,p,100,vmin=0,vmax=1,cmap='coolwarm', alpha=0.6)
#cbar = plt.colorbar(ax)
#cbar.update_ticks()

plt.scatter(x1[idx_class0], x2[idx_class0], s=50, c='r', marker='.', label='Class0')
plt.scatter(x1[idx_class1], x2[idx_class1], s=50, c='b', marker='.', label='Class1')
plt.contour(xx1, xx2, p, levels = [0.5], linewidths=2, colors='k')
plt.legend(loc = 'upper right')
plt.title('Decision boundary')
plt.show()
```



4. Plot the probability map of the obtained classifier [2pt]

In [472]:

```
# plot
plt.figure(4,figsize=(15,10))

cf = plt.contourf(xx1, xx2, p, cmap = 'cividis', levels = 100)
cbar = plt.colorbar(cf, ticks = [0.00, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80, 0.90, 1.00])
cbar.update_ticks()

plt.scatter(x1[idx_class0], x2[idx_class0], s=50, c='r', marker='.', label='Class0')
plt.scatter(x1[idx_class1], x2[idx_class1], s=50, c='b', marker='.', label='Class1')
plt.contour(xx1, xx2, p, levels = [0.5], linewidths=2, colors='k')
plt.legend()
plt.title('Decision Boundary')
plt.show()
```



5. Compute the classification accuracy [1pt]

In [477]:

```
print('total number of data = ', (n))
print('total number of correctly classified data = ', (idx_class0_correct + idx_class1_correct))
print('accuracy(%) = ',  accuracy)
```

```
total number of data =  1000
total number of correctly classified data =  960
accuracy(%) =  96.0
```

In [ ]: