Supervised classification - improving capacity learning

0. Import library

Import library

In [1]:
```
# 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 $x_i = (x_{i(1)}, x_{i(2)})$.

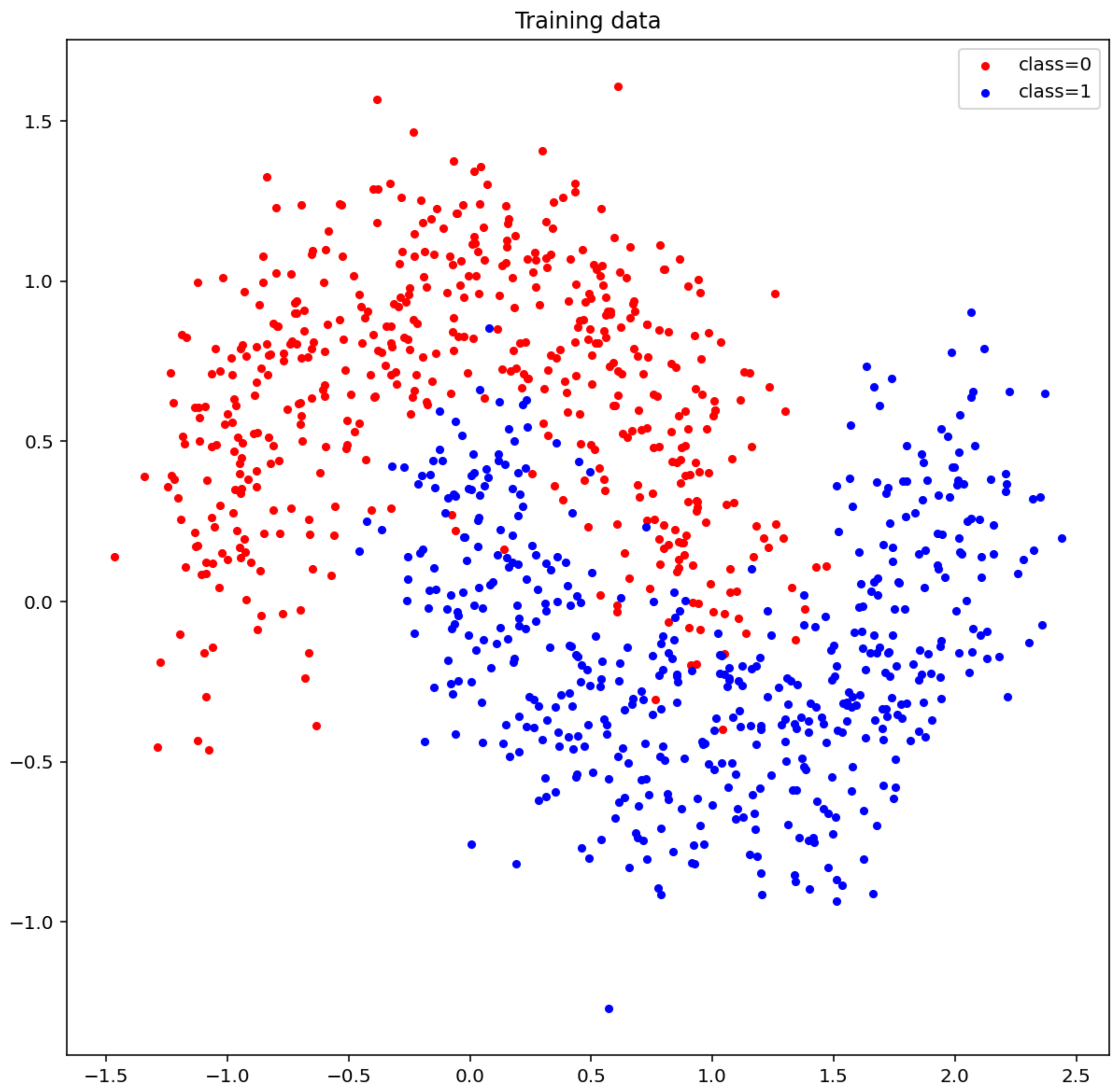
The data label/target, y_i, indicates two classes with value 0 or 1.

Plot the data points.

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

In [2]:
```
# import data with numpy
data = np.genfromtxt('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))
data[0][0] = 0.817146676
print(data)

# 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 data')
plt.legend()
plt.show()
```

```
Number of the data = 1000
Shape of the data = (1000, 3)
Data type of the data = float64
[[ 0.81714668 -0.06387847  0.        ]
 [ 1.42722373  0.10704005  0.        ]
 [ 1.04127962 -0.39929282  0.        ]
 ...
 [ 0.17676639  0.35218305  1.        ]
 [ 0.17604457  0.20636919  1.        ]
 [ 0.00652276  0.39280286  1.        ]]
```



2. Define a logistic regression loss function and its gradient

Typesetting math: 100%

In [3]:
```
# 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)
    loss = (np.dot(-y.T, np.log(y_pred+0.0000001)) - np.dot((1-y).T, np.log(1-y_pred+0.0000001))) / n
    return loss

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

# gradient descent function definition
def grad_desc(num_of_rows, X, y , w_init=np.array([0,0,0])[:,None] ,tau=1e-4, max_iter=500):
    L_iters = np.zeros([max_iter]) # record the loss values
    w_iters = np.zeros([max_iter,num_of_rows]) # 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
        w_iters[i,:] = w.reshape(1, len(w)) # save the current w value

    return w, L_iters, w_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:

$$X = \begin{bmatrix} f_0(x_1) & f_1(x_1) & f_2(x_1) & f_3(x_1) & f_4(x_1) & f_5(x_1) & f_6(x_1) & f_7(x_1) & f_8(x_1) & f_9(x_1) \\ f_0(x_2) & f_1(x_2) & f_2(x_2) & f_3(x_2) & f_4(x_2) & f_5(x_2) & f_6(x_2) & f_7(x_2) & f_8(x_2) & f_9(x_2) \\ \vdots \\ f_0(x_n) & f_1(x_n) & f_2(x_n) & f_3(x_n) & f_4(x_n) & f_5(x_n) & f_6(x_n) & f_7(x_n) & f_8(x_n) & f_9(x_n) \end{bmatrix} \quad \text{and} \quad w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \\ w_8 \\ w_9 \end{bmatrix}$$

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

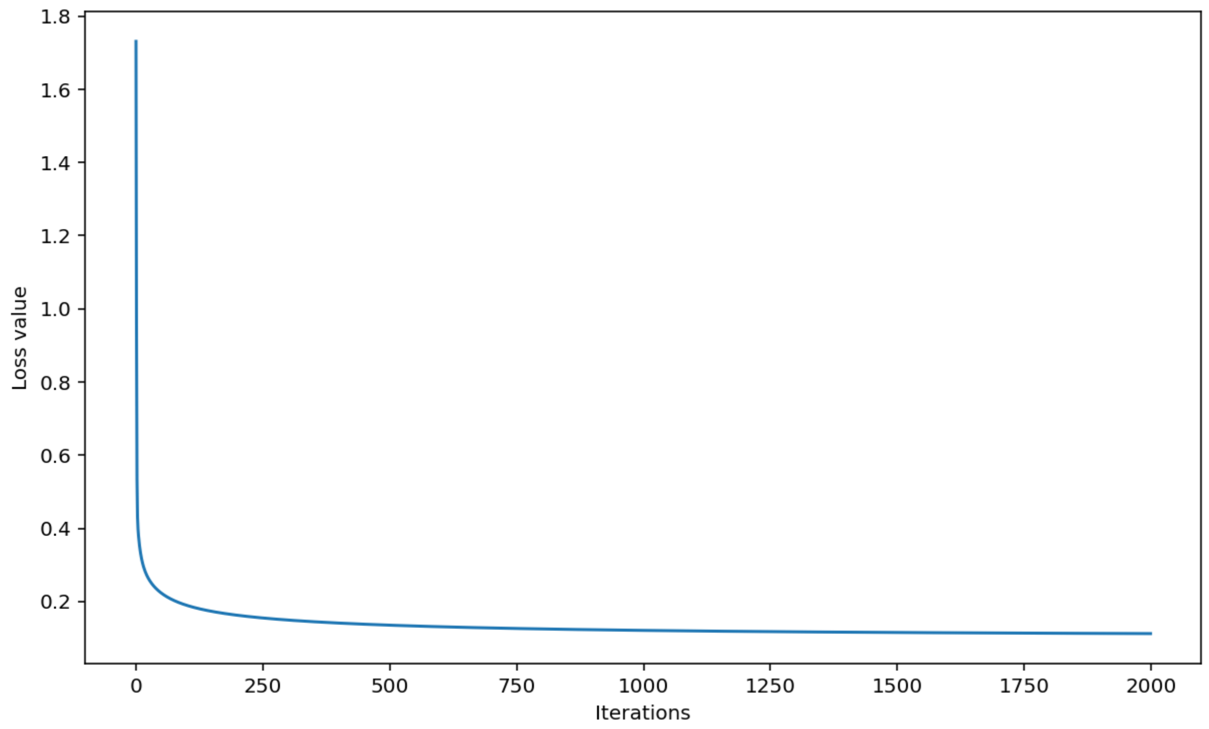
You can use at most 10 feature functions $f_i, i = 0, 1, 2, \cdots, 9$ in such a way that the classification accuracy is maximized. You are allowed to use less than 10 feature functions.

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

In [4]:
```
def make_x(n, data1, data2): #{
    num_of_rows = 7
    X = np.ones([n,num_of_rows])

    X[:,0] = np.power(data1, 3)
    X[:,1] = np.power(data2, 3)
    X[:,2] = data1
    X[:,3] = data2
    X[:,4] = 1
    X[:,5] = np.power(data1, 2)
    X[:,6] = np.power(data2, 2)





    # X[:,5] = np.power(data1, 3)
    # X[:,6] = np.power(data2, 3)
    # X[:,7] = np.multiply(np.power(data1, 3), np.power(data2, 3))
    # X[:,8] = np.power(data2, 5)
    # X[:,9] = np.multiply(np.power(data1, 5), np.power(data2, 4))

    return X
#}
```

Typesetting math: 100%

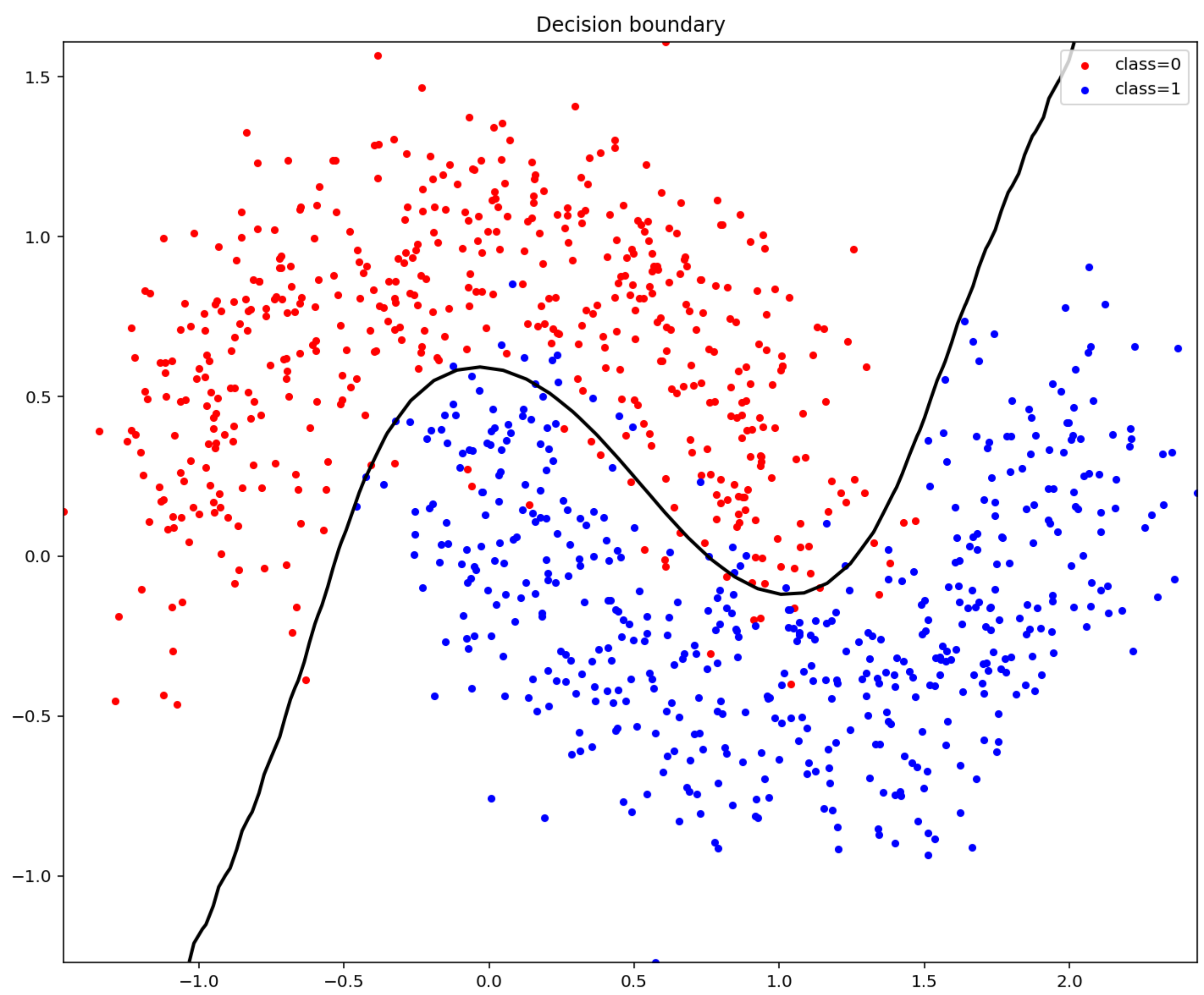
In [5]:
```
import math
# construct the data matrix X, and label vector y
n = data.shape[0]

data1 = x1
data2 = x2
X = make_x(n, data1, data2)

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


# run gradient descent algorithm
start = time.time()
w_init = np.ones(len(X[0]))[:,None]
tau = 1e-0; max_iter = 2000
w, L_iters, W_iters = grad_desc(len(X[0]), X, y, w_init, tau, max_iter)

# 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()
print(w)
```




```
[[  9.16127337]
 [ -3.54763222]
 [ -0.98006034]
 [ -7.51964831]
 [  4.59379367]
 [-13.69684311]
 [  1.6670601 ]]
```

4. Plot the decision boundary

In [6]:
```
# 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

data1 = xx1.reshape(-1)
data2 = xx2.reshape(-1)
n = len(data1)
X2 = make_x(n, data1, data2)

p1 = f_pred(X2, w)
p1 = p1.reshape(xx1.shape[0], xx2.shape[0])

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


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.contour(xx1, xx2, p1, levels=1, linewidths=2, colors='k')
plt.legend()
plt.title('Decision boundary')
plt.show()
```

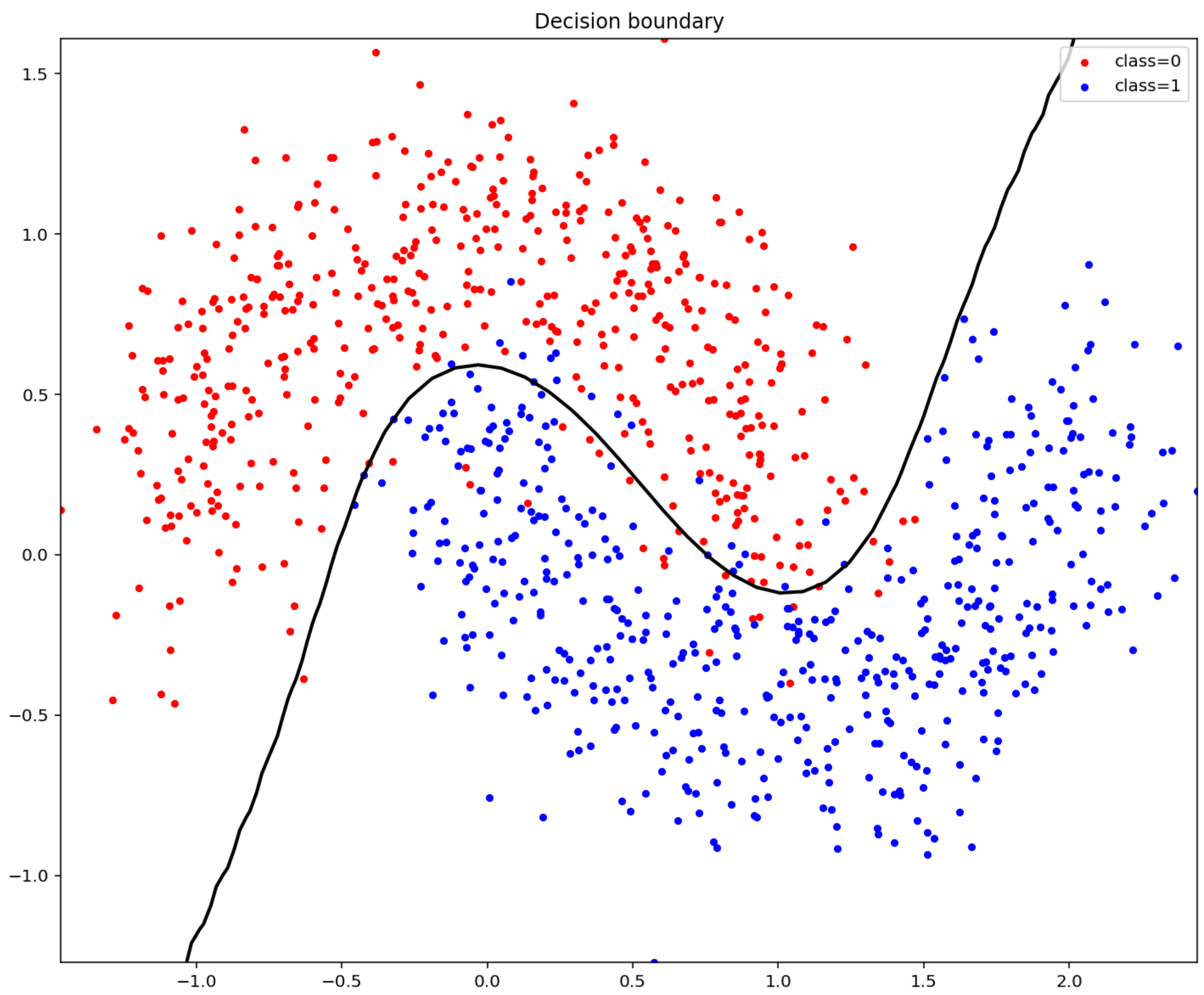


5. Plot the probability map

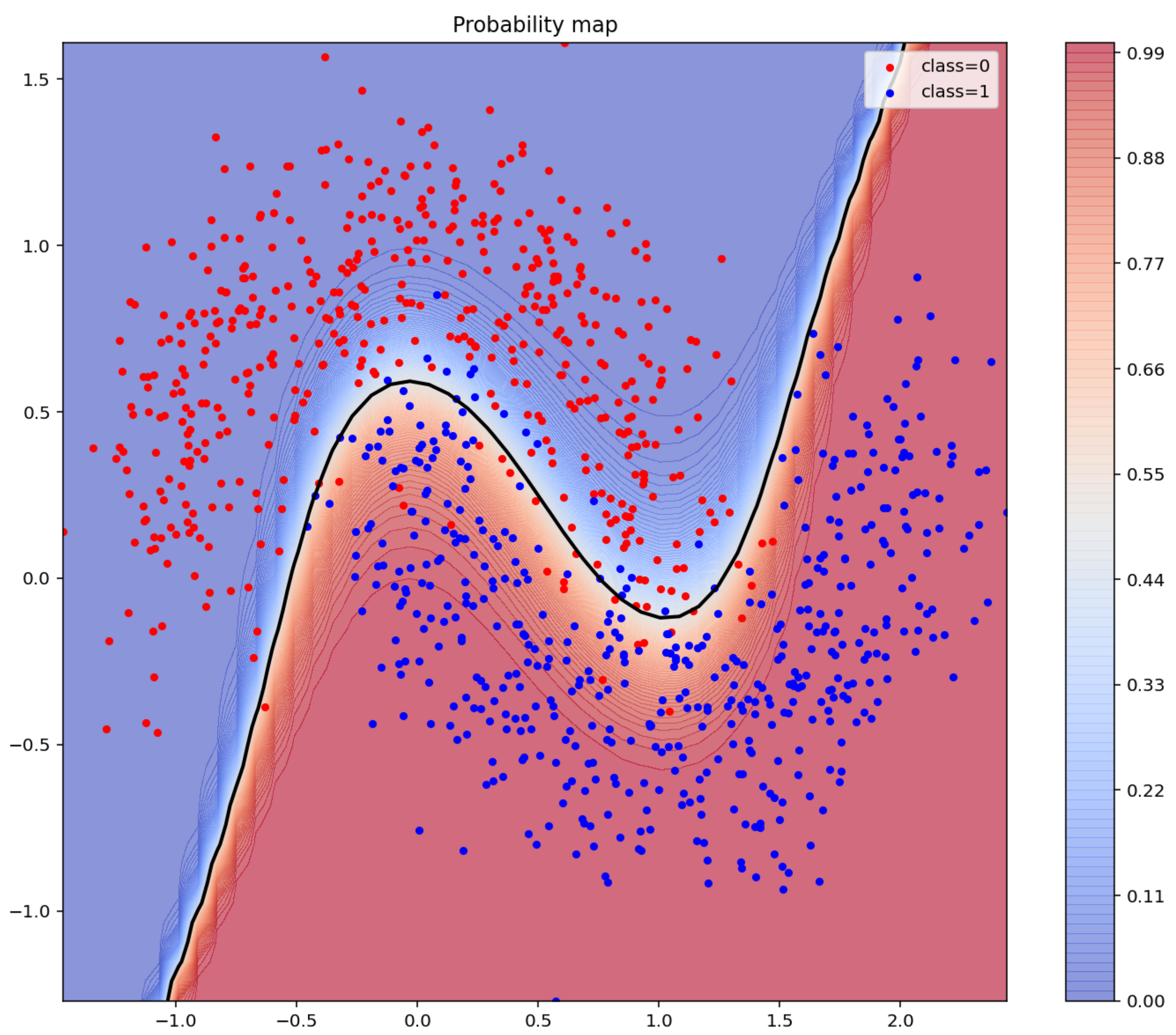
Typesetting math: 100%

In [7]:
```
start = time.time()
x_train = data[:,:2]
y = data[:,2][:,None]
logreg_sklearn = LogisticRegression()# scikit-learn logistic regression
logreg_sklearn.fit(x_train, y) # learn the model parameters
# compute loss value
w_sklearn = np.zeros([3,1])
w_sklearn[0,0] = logreg_sklearn.intercept_
w_sklearn[1:3,0] = logreg_sklearn.coef_[0] # loss_sklearn = ce_loss()
# plot

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


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.contour(xx1, xx2, p1, levels=1, linewidths=2, colors='k')
plt.legend()
plt.title('Decision boundary')
plt.show()
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/utils/validation.py:760: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please chan
ge the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```



In [8]:
```
# 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

data1 = xx1.reshape(-1)
data2 = xx2.reshape(-1)
n = len(data1)
X2 = make_x(n, data1, data2)

p2 = f_pred(X2, w)
p2 = p2.reshape(xx1.shape[0], xx2.shape[0])

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

ax = plt.contourf(xx1,xx2,p2,100,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='class=0')
plt.scatter(x1[idx_class1], x2[idx_class1], s=50, c='b', marker='.', label='class=1')
plt.contour(xx1, xx2, p2, levels=1, linewidths=2, colors='k')
plt.legend(loc=1)
plt.title('Probability map')
plt.show()
```



Typesetting math: 100%

6. Compute the classification accuracy

The accuracy is computed by:

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

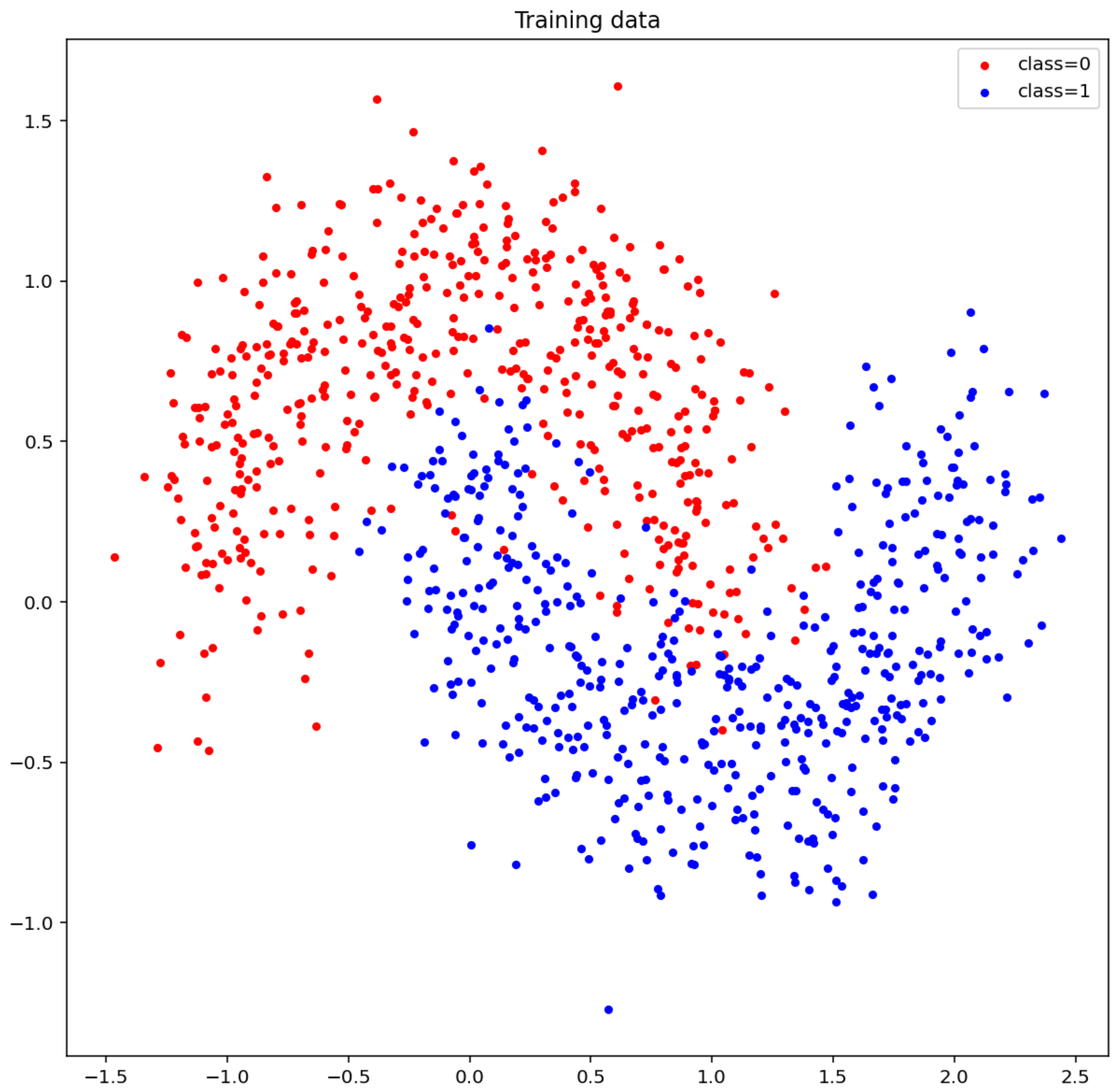
In [9]:
```
# compute the accuracy of the classifier
n = data.shape[0]

# 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

data1 = x1
data2 = x2
n = len(data1)
X3 = make_x(n, data1, data2)

p3 = f_pred(X3, w)
pred = []
idx_wrong = 0
for i in p3: #{
    if i>=0.5:
      pred.append(1)
    else:
      pred.append(0)
#}
for i in range(len(pred)): #{
    if pred[i] != idx[i]: idx_wrong+=1
#}
#print(idx_class1_label)
#print(idx_class1_pred)



#print(np.sum(idx_wrong))
print('total numver of data = ', n)
print('total number of correctly classified data = ', n-idx_wrong)
print('accuracy(%) = ',  ((n-idx_wrong)/n)*100)
```

```
total numver of data =  1000
total number of correctly classified data =  955
accuracy(%) =  95.5
```

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

1. Visualize the data [1pt]

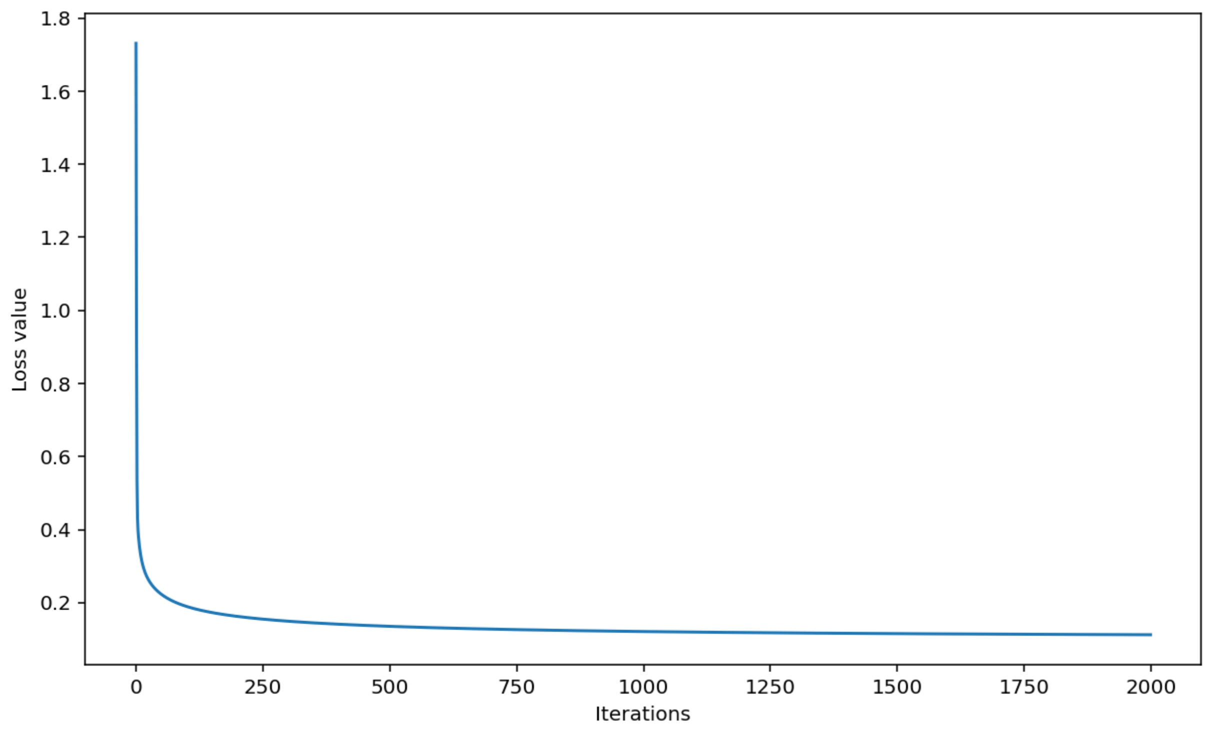
In [10]:
```
plt.figure(1,figsize=(10,10))
plt.scatter(x1[idx_class0], x2[idx_class0], s=50, c='r', marker='.', label='class=0')
plt.scatter(x1[idx_class1], x2[idx_class1], s=50, c='b', marker='.', label='class=1')
plt.title('Training data')
plt.legend()
plt.show()
```



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

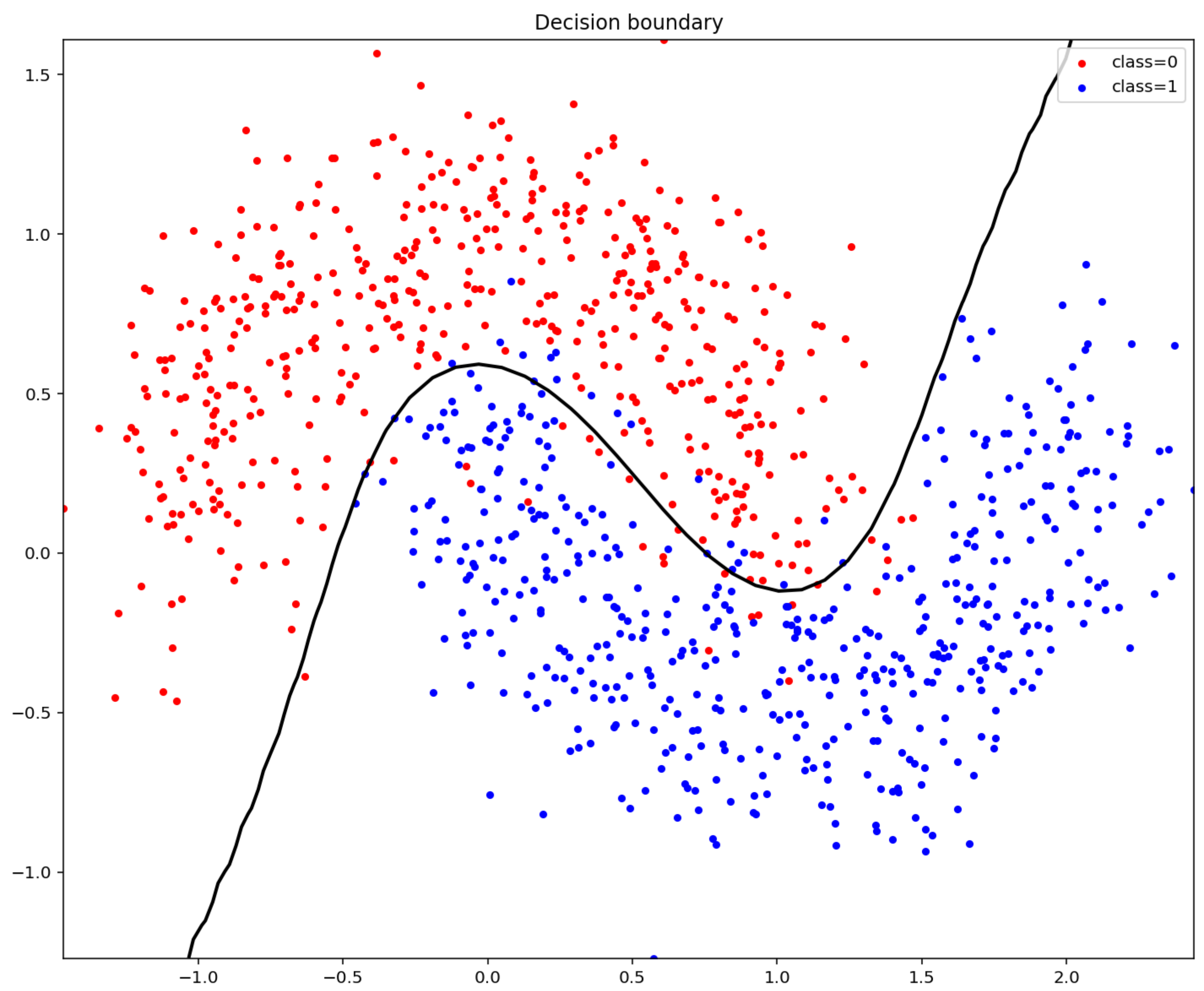
Typesetting math: 100%

In [11]:
```
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]

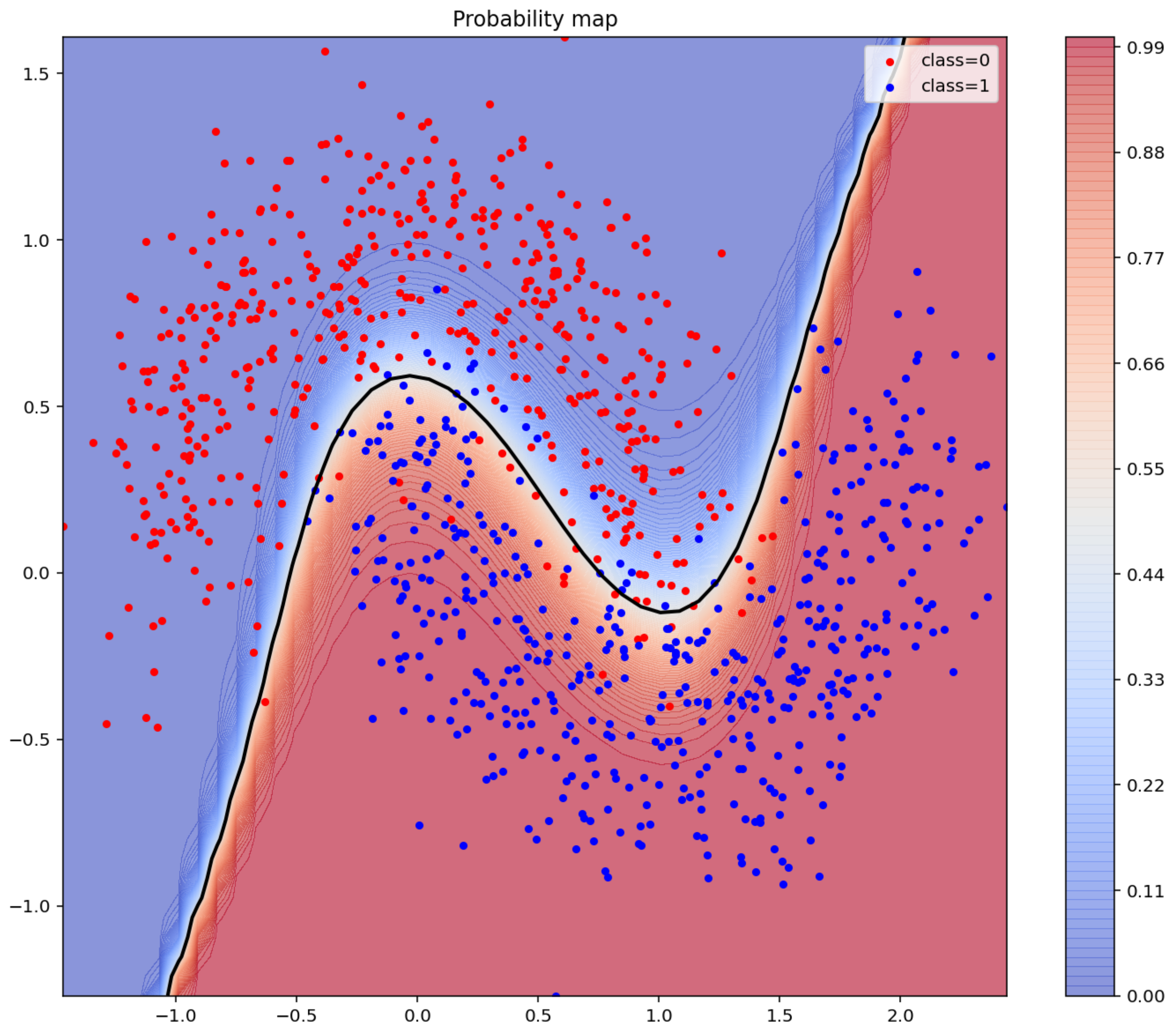
In [12]:
```
plt.figure(4,figsize=(12,10))

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.contour(xx1, xx2, p1, levels=1, linewidths=2, colors='k')
plt.legend()
plt.title('Decision boundary')
plt.show()
```



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

Typesetting math: 100%

In [13]:
```
plt.figure(4,figsize=(12,10))

ax = plt.contourf(xx1,xx2,p2,100,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='class=0')
plt.scatter(x1[idx_class1], x2[idx_class1], s=50, c='b', marker='.', label='class=1')
plt.contour(xx1, xx2, p2, levels=1, linewidths=2, colors='k')
plt.legend(loc=1)
plt.title('Probability map')
plt.show()
```



5. Compute the classification accuracy [1pt]

In [14]:
```
print('total numver of data = ', n)
print('total number of correctly classified data = ', n-idx_wrong)
print('accuracy(%) = ',  ((n-idx_wrong)/n)*100)
```

```
total numver of data =  1000
total number of correctly classified data =  955
accuracy(%) =  95.5
```

Typesetting math: 100%