Supervised classification - improving capacity learning

0. Import library

Import library

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# 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-01.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.

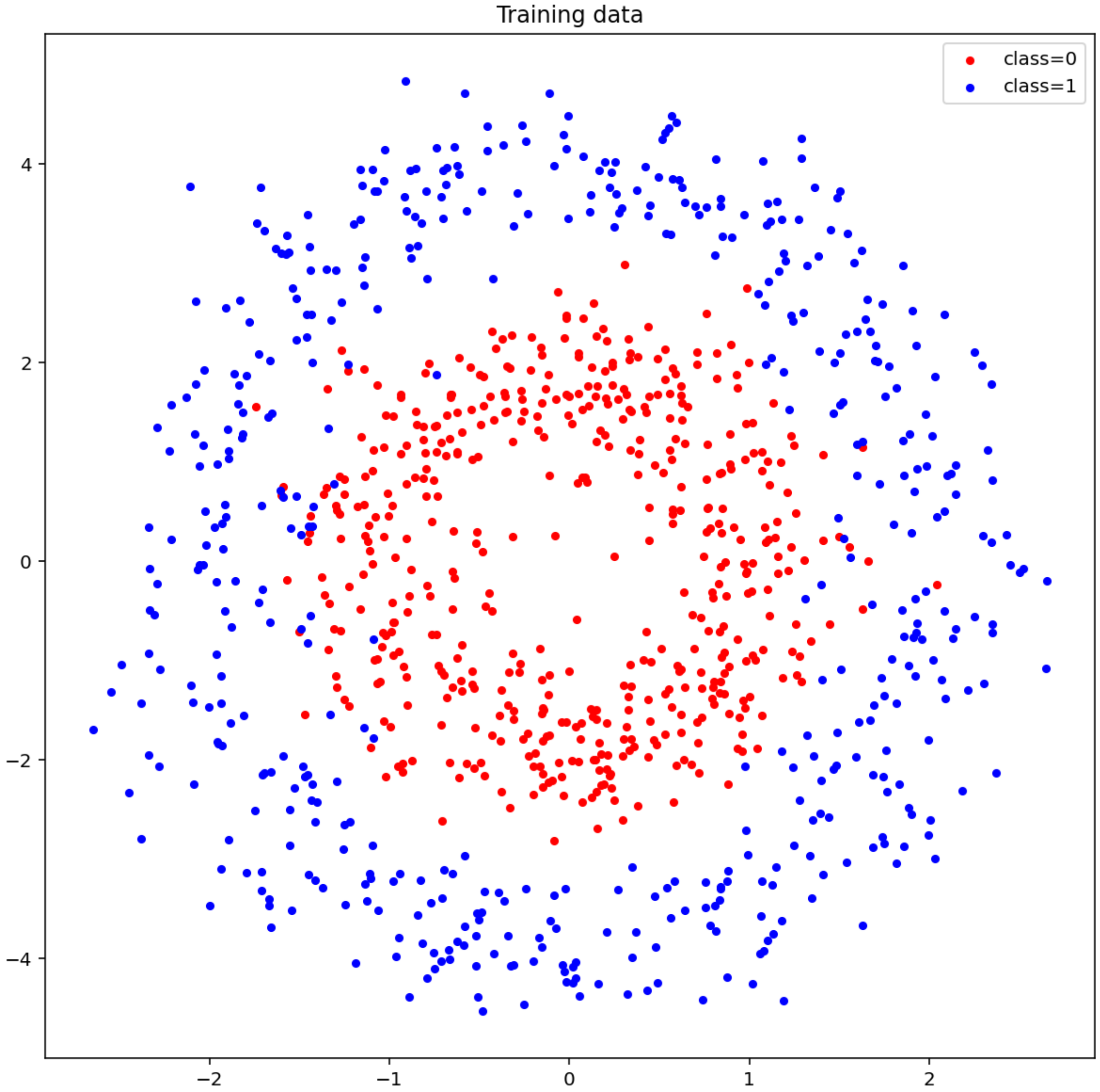
Plot the data points.

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

```
# import data with numpy
path = '/content/drive/My Drive/ML_Assignment/data/dataset-a.txt'
data = np.loadtxt(path, 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))
```

```
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

fig_1 = 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()
fig_1.savefig('Visualize the data.png')
```

```
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

```
# sigmoid function
def sigmoid(z):
```

```
def sigmoid(z):
    sigmoid_f = 1.0 / (1.0 + 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.sum(-y * np.log(y_pred) - (1 - y) * np.log(1 - y_pred)) / n
    return loss


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


# 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:

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

$$\text{and} \quad w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \\ w_8 \\ w_9 \end{bmatrix}$$

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

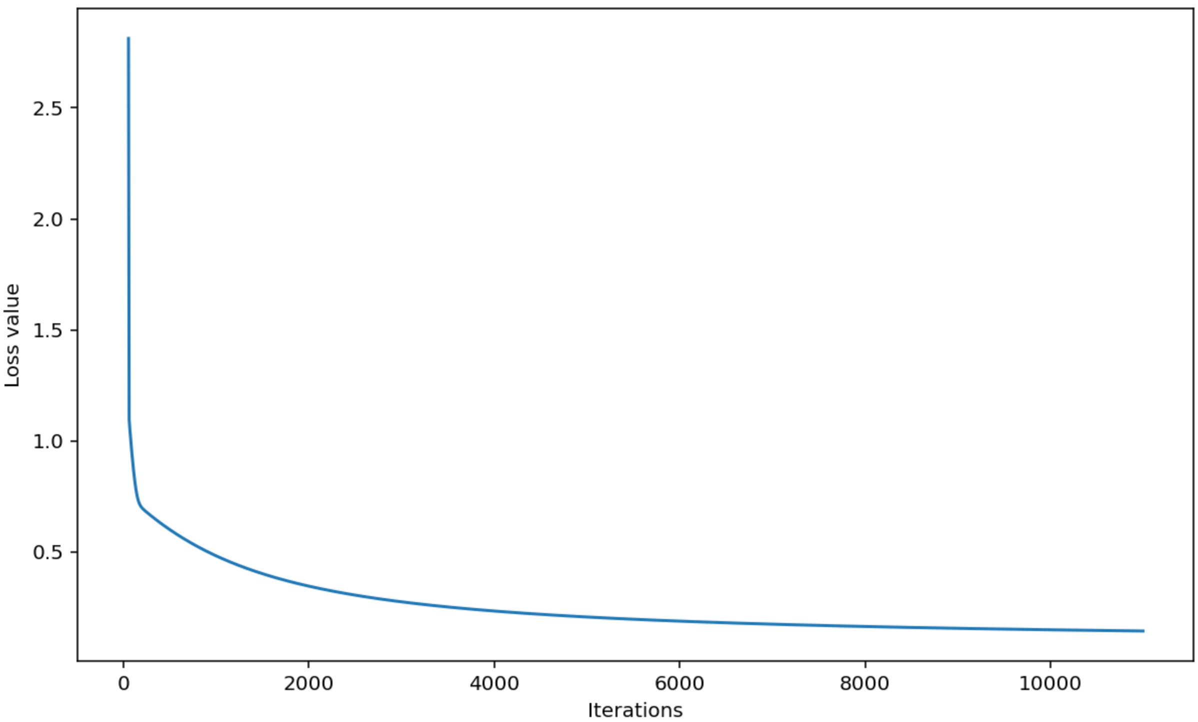
You can use at most 10 feature functions $f_i, i = 0, 1, 2, \cdots, 9$ in such a way that the classification accuracy is maximized. You are allowed to use less than 10 feature functions.

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

```
def feature_function(x1, x2):
  f_func = np.array([(x1**0)*(x2**0), 1*(x1**2)+1*(x2**2),1*(x1**2)+2*(x2**2), 1*(x1**2)+3*(x2**2), 2*(x
                     2*(x1**2)+3*(x2**2), 3*(x1**2)+1*(x2**2), 3*(x1**2)+2*(x2**2), (x1**2), (x2**2)])

  return f_func.T
```

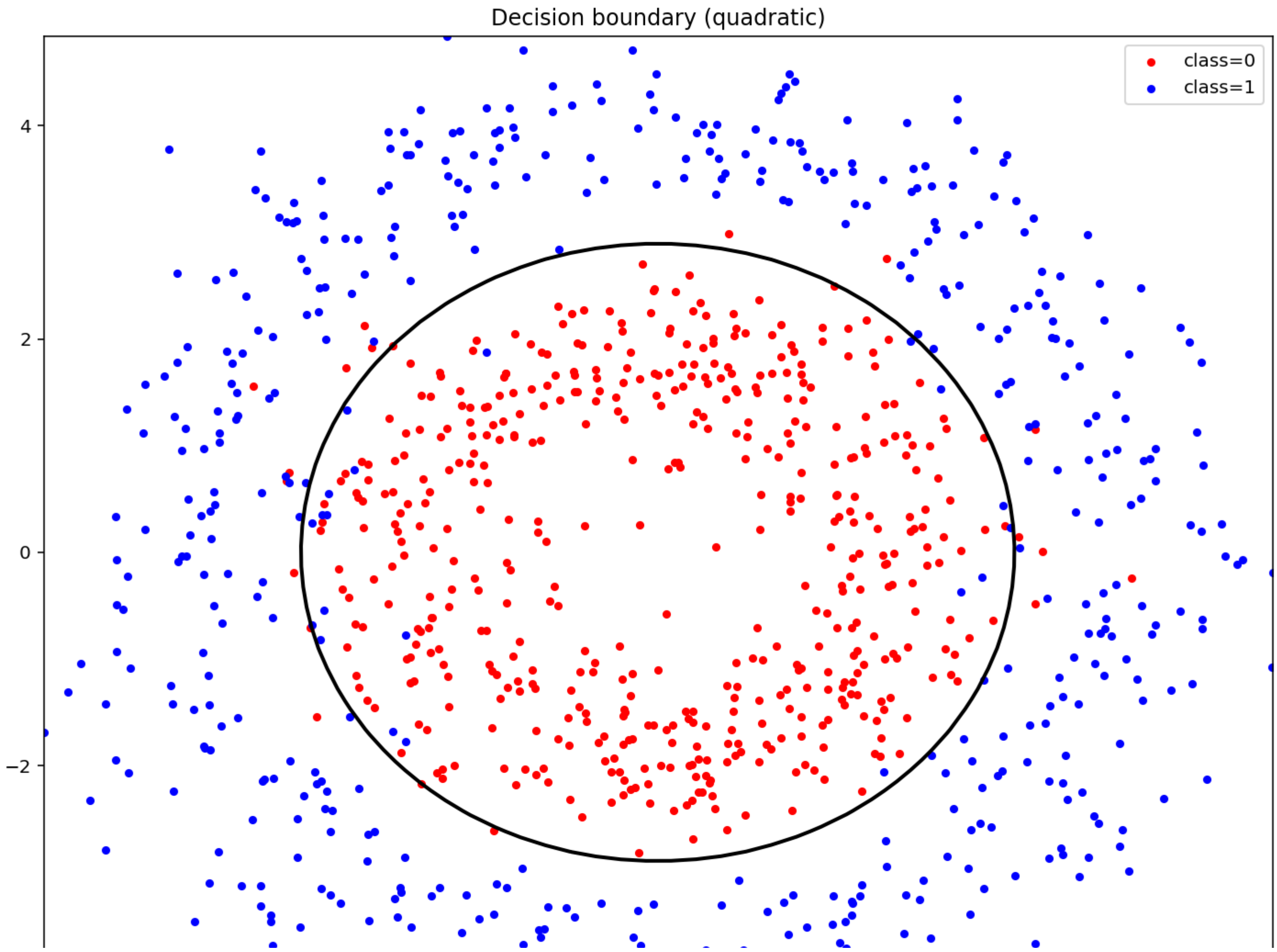
```
import math
# construct the data matrix X, and label vector y
n = data.shape[0]
X = feature_function(x1, x2)
y = data[:,2][:,None] # label


# run gradient descent algorithm
start = time.time()
w_init = np.array([1.,1.,1.,1.,1.,1.,1.,1.,1.,1.])[:,None]
tau = 0.003; max_iter = 11000
w, L_iters = grad_desc(X, y , w_init, tau, max_iter)

# plot
print(L_iters[-1])
fig_2 = plt.figure(3, figsize=(10,6))
plt.plot(np.array(range(max_iter)), L_iters)
plt.xlabel('Iterations')
plt.ylabel('Loss value')
plt.show()
fig_2.savefig('Plot the loss curve.png')
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:16: RuntimeWarning: divide by zero encounte
  app.launch_new_instance()
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:16: RuntimeWarning: invalid value encounter
  app.launch_new_instance()
0.1428893259032481 8
```



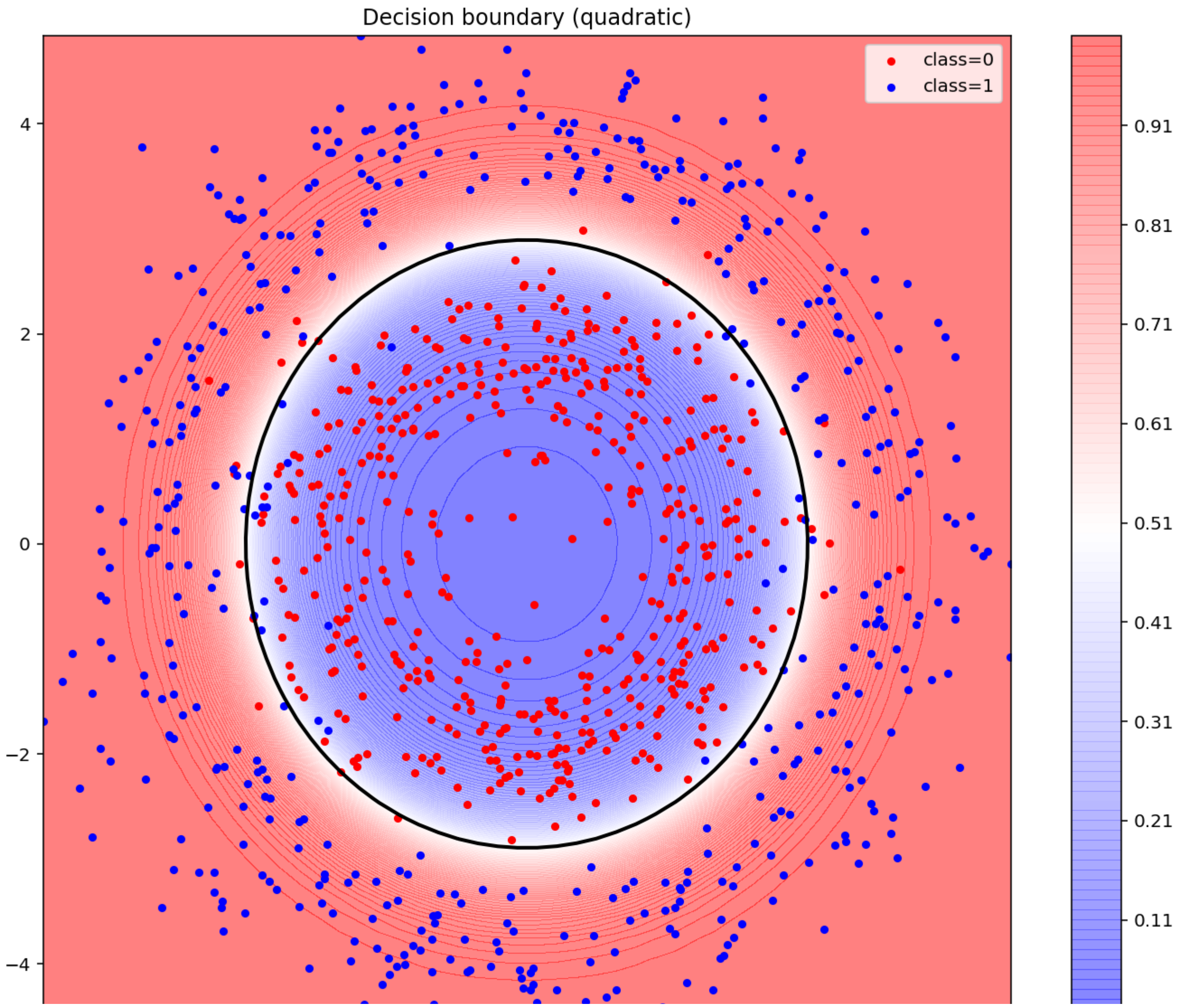
4. Plot the decisoin boundary

```
# 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 = feature_function(xx1.reshape(-1), xx2.reshape(-1))

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

# plot
fig_3 = 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='class=0')
plt.scatter(x1[idx_class1], x2[idx_class1], s=50, c='b', marker='.', label='class=1')
plt.contour(xx1, xx2, p, [0.5], linewidths=2, colors='k')
plt.legend(loc = 'upper right')
plt.title('Decision boundary (quadratic)')
plt.show()
fig_3.savefig('Plot the decisoin boundary.png')
```



5. Plot the probability map

```
# 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 = feature_function(xx1.reshape(-1), xx2.reshape(-1))
p = f_pred(X2,w)
p = p.reshape(50,50)

# plot
fig_4 = plt.figure(4,figsize=(12,10))

ax = plt.contourf(xx1, xx2, p, 100, cmap = 'bwr', vmin = 0, vmax = 1, alpha = 0.5)
cbar = plt.colorbar( )
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, p, [0.5], linewidths=2, colors='k')
plt.legend(loc = 'upper right')
plt.title('Decision boundary (quadratic)')
plt.show()
fig_4.savefig('Plot the probability map.png')
```



6. Compute the classification accuracy

The accuracy is computed by:

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

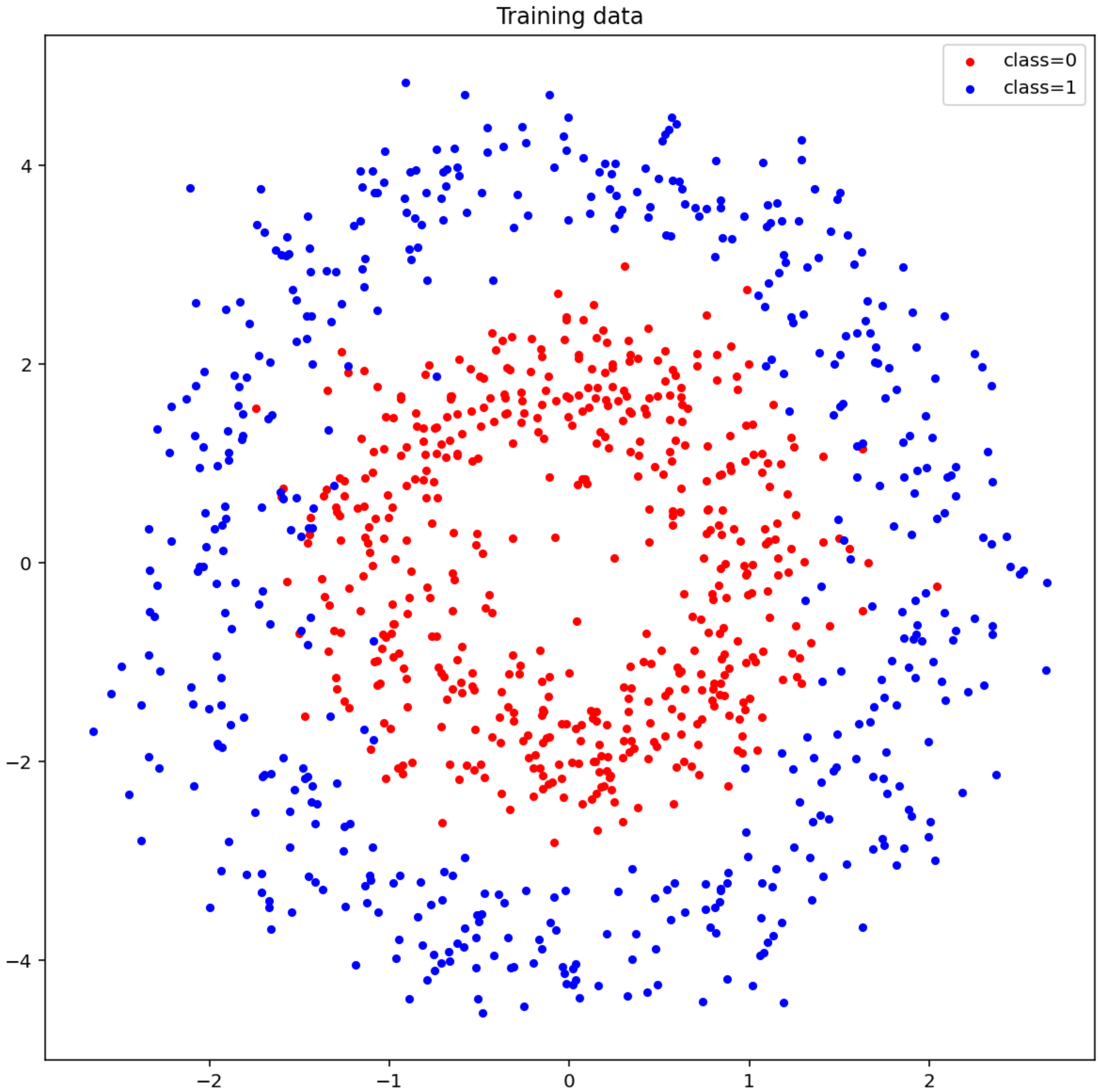
```
# compute the accuracy of the classifier
n = data.shape[0]
print('total number of data = ', n)

# plot
x1 = data[:,0] # feature 1
x2 = data[:,1] # feature 2
idx_class0 = (idx == 0)# index of class0
idx_class1 = (idx == 1) # index of class1

X2 = feature_function(x1, x2)
p = f_pred(X2,w)

idx_class0_pred = (p < 0.5)

idx_right = (idx_class0 == idx_class0_pred.reshape(-1))

print('total number of correctly classified data = ', sum(idx_right))
print('accuracy(%) = ', sum(idx_right)/n * 100)
```

```
total number of data =  1000
total number of correctly classified data =  962
accuracy(%) =  96.2
```

Output using the dataset (dataset-noise-01.txt)

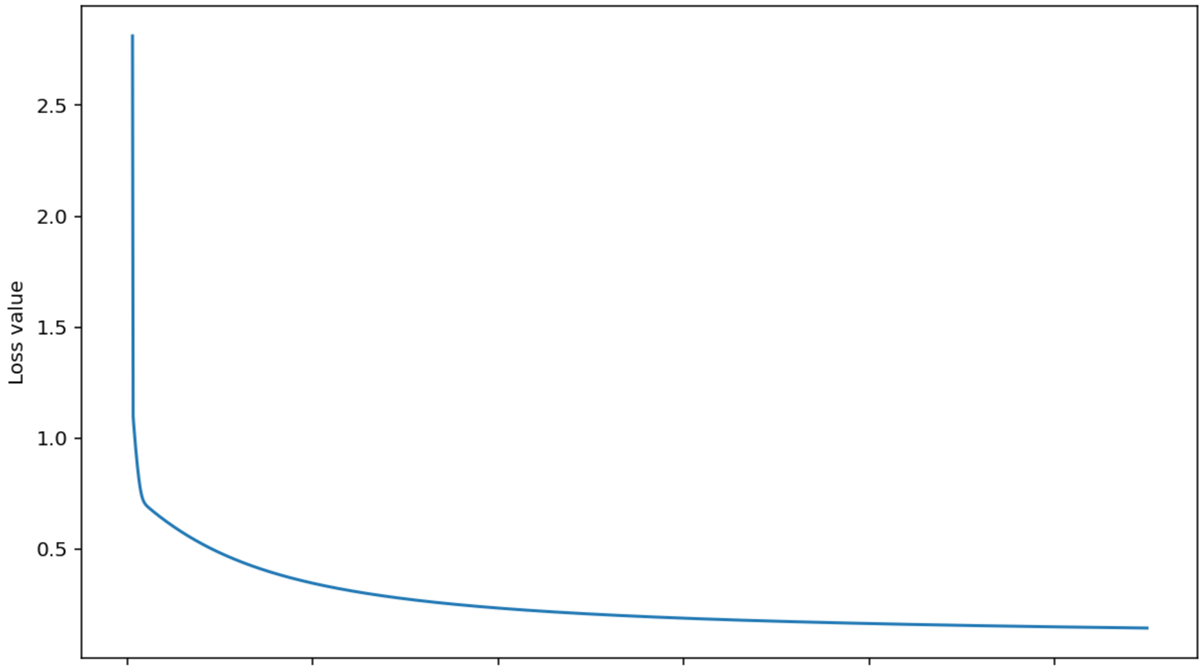
1. Visualize the data [1pt]

```
1   fig_1
```



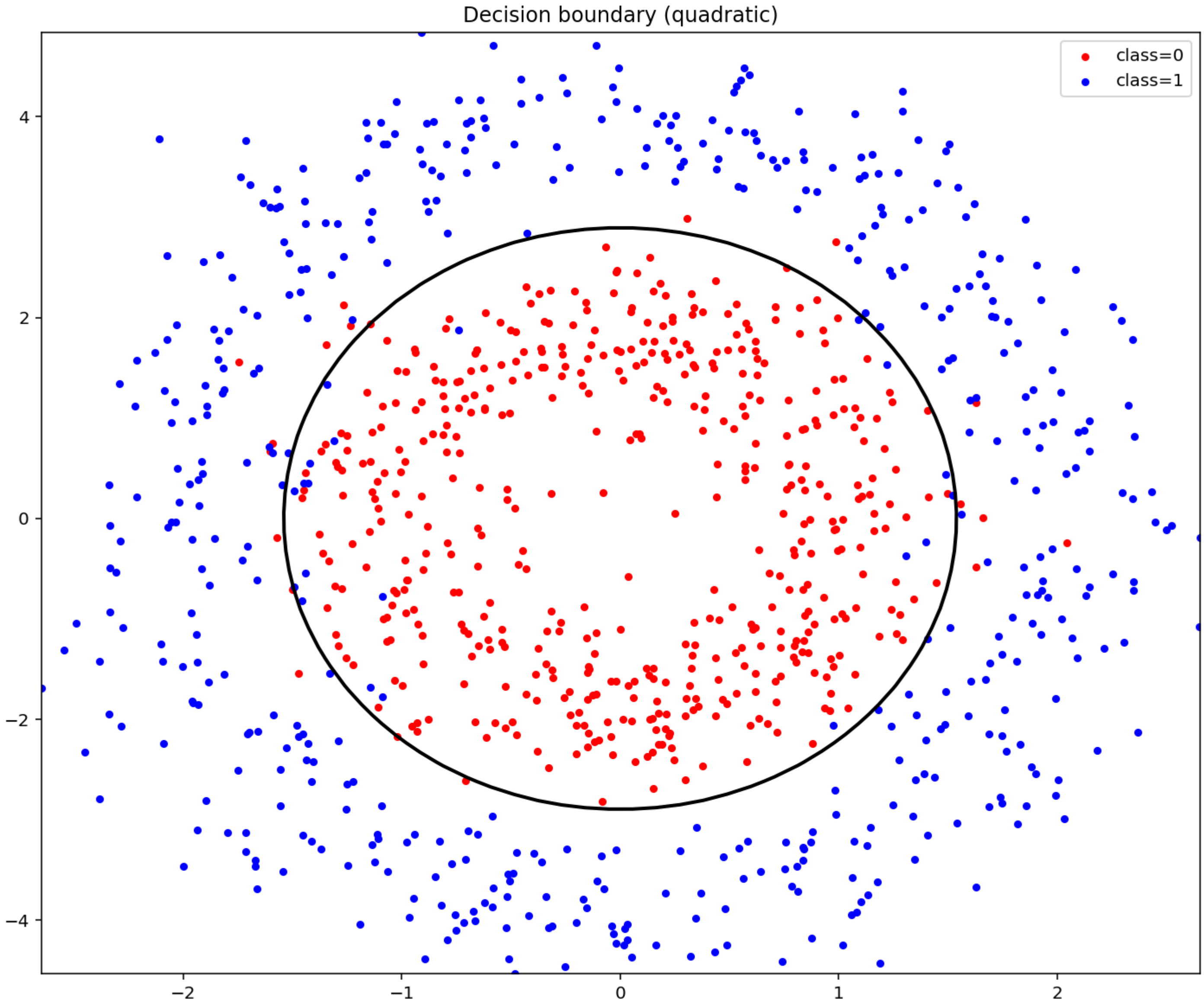
2. Plot the loss curve obtained by the gradient descent until the convergence [2pt]

```
1   fig_2
```



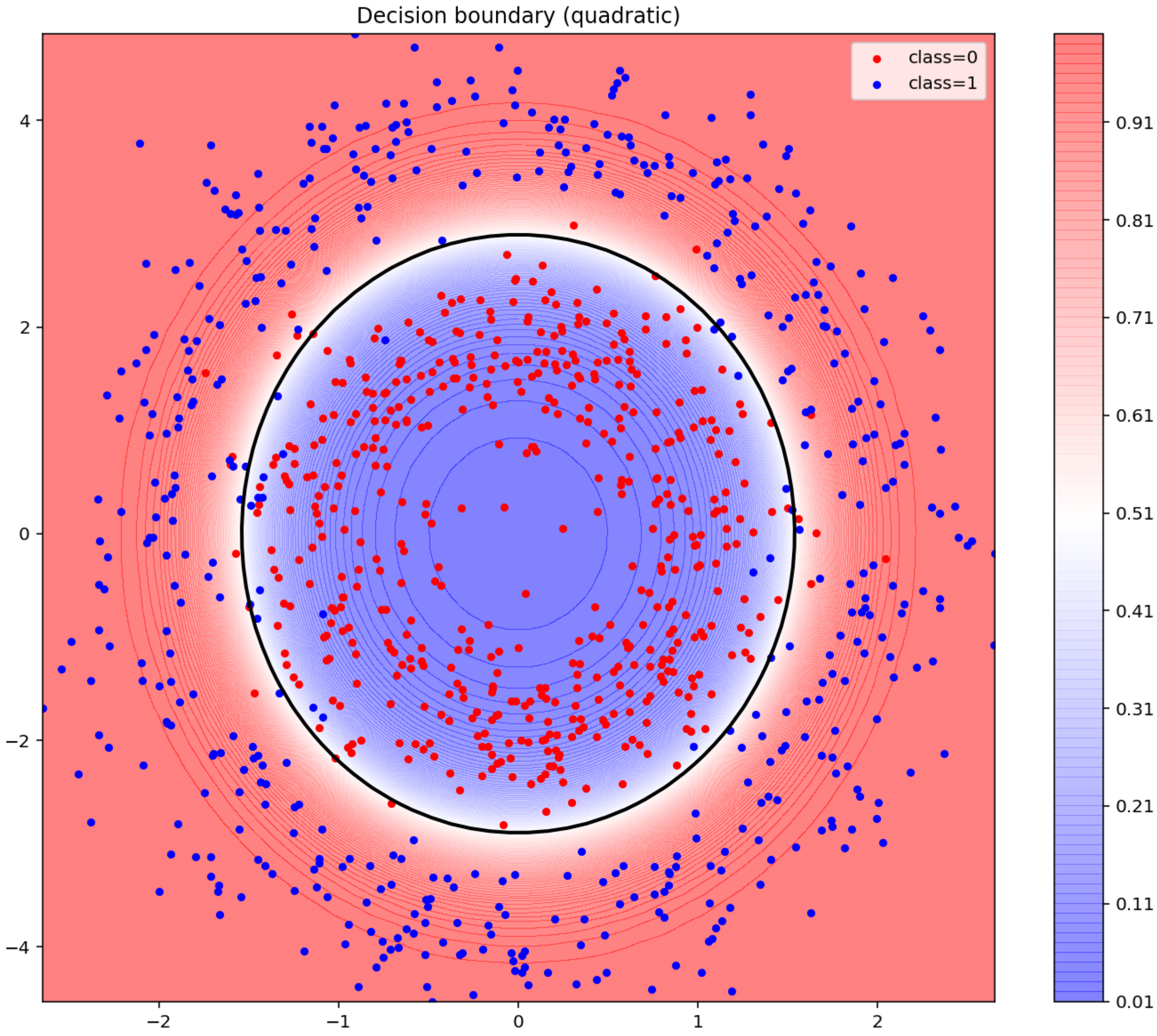
3. Plot the decisoin boundary of the obtained classifier [2pt]

```
fig_3
```



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

```
1 fig_4
```



5. Compute the classification accuracy [1pt]

```
1 print('total number of data = ', n)
2 print('total number of correctly classified data = ', sum(idx_right))
3 print('accuracy(%) = ', sum(idx_right)/n * 100)
```

```
total number of data =  1000
total number of correctly classified data =  962
accuracy(%) =  96.2
```