

Logistic Regression with non-linear features

import library

```
In [ ]: import numpy as np  
import matplotlib.pyplot as plt  
import matplotlib.colors as colors  
from matplotlib import ticker, cm
```

load training data

```
In [ ]: fname_data_train = 'assignment_10_data_train.csv'  
fname_data_test = 'assignment_10_data_test.csv'  
  
data_train = np.genfromtxt(fname_data_train, delimiter=',')  
data_test = np.genfromtxt(fname_data_test, delimiter=',')  
  
number_data_train = data_train.shape[0]  
number_data_test = data_test.shape[0]  
  
data_train_point = data_train[:, 0:2]  
data_train_point_x = data_train_point[:, 0]  
data_train_point_y = data_train_point[:, 1]  
data_train_label = data_train[:, 2]  
  
data_test_point = data_test[:, 0:2]  
data_test_point_x = data_test_point[:, 0]  
data_test_point_y = data_test_point[:, 1]  
data_test_label = data_test[:, 2]  
  
data_train_label_class_0 = (data_train_label == 0)  
data_train_label_class_1 = (data_train_label == 1)  
  
data_test_label_class_0 = (data_test_label == 0)  
data_test_label_class_1 = (data_test_label == 1)  
  
data_train_point_x_class_0 = data_train_point_x[data_train_label_class_0]  
data_train_point_y_class_0 = data_train_point_y[data_train_label_class_0]  
  
data_train_point_x_class_1 = data_train_point_x[data_train_label_class_1]  
data_train_point_y_class_1 = data_train_point_y[data_train_label_class_1]  
  
data_test_point_x_class_0 = data_test_point_x[data_test_label_class_0]  
data_test_point_y_class_0 = data_test_point_y[data_test_label_class_0]  
  
data_test_point_x_class_1 = data_test_point_x[data_test_label_class_1]  
data_test_point_y_class_1 = data_test_point_y[data_test_label_class_1]  
  
print('shape of point in train data = ', data_train_point.shape)  
print('shape of point in test data = ', data_train_point.shape)  
  
print('shape of label in train data = ', data_train_label.shape)  
print('shape of label in test data = ', data_train_label.shape)  
  
print('data type of point x in train data = ', data_train_point_x.dtype)
```

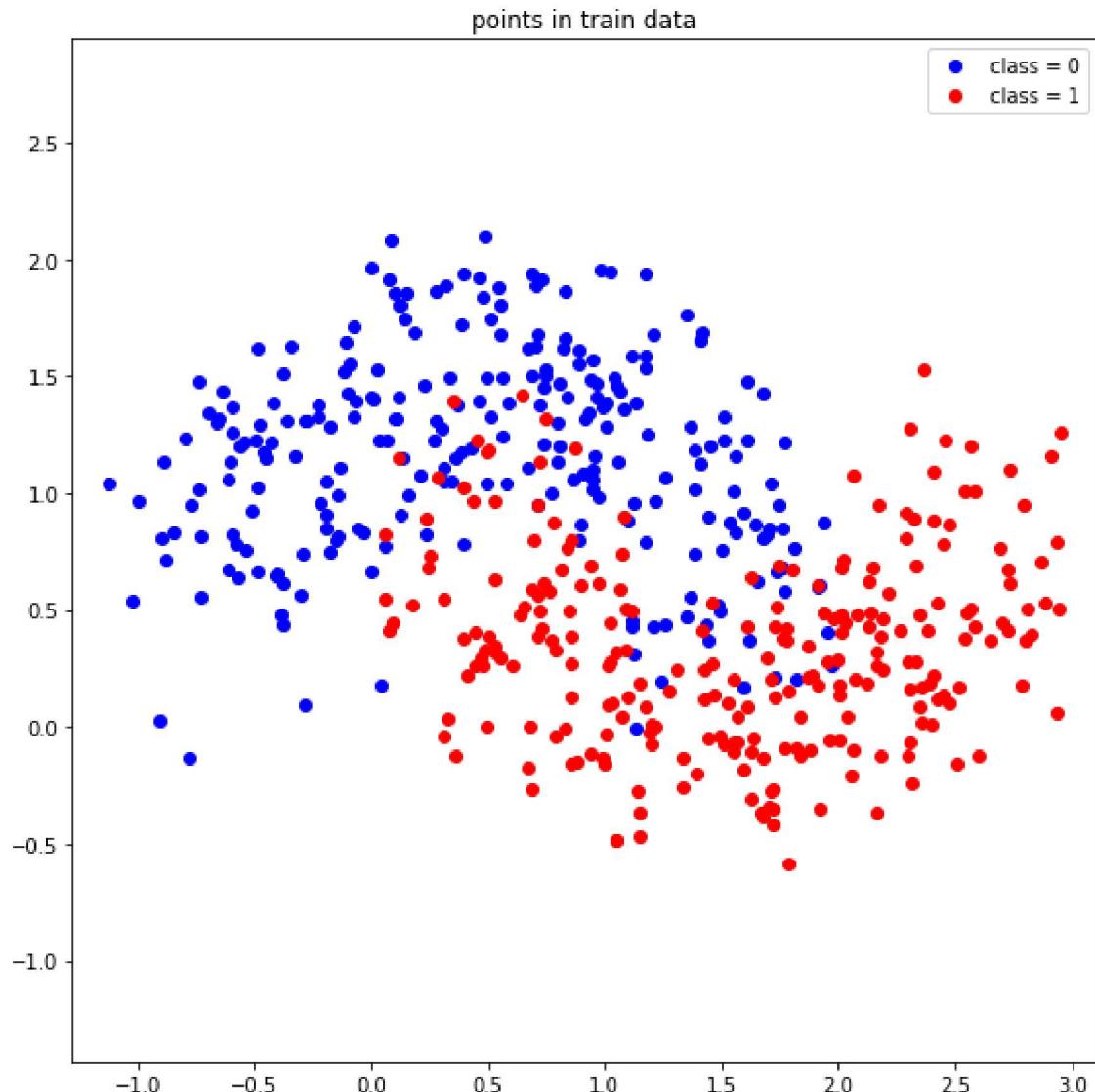
```
print('data type of point y in train data = ', data_train_point_y.dtype)
print('data type of point x in test data = ', data_test_point_x.dtype)
print('data type of point y in test data = ', data_test_point_y.dtype)

shape of point in train data = (500, 2)
shape of point in test data = (500, 2)
shape of label in train data = (500,)
shape of label in test data = (500,)
data type of point x in train data = float64
data type of point y in train data = float64
data type of point x in test data = float64
data type of point y in test data = float64
```

plot the data

```
In [ ]: f = plt.figure(figsize=(8,8))

plt.title('points in train data')
plt.plot(data_train_point_x_class_0, data_train_point_y_class_0, 'o', color='blue',
          plt.plot(data_train_point_x_class_1, data_train_point_y_class_1, 'o', color='red',
          plt.axis('equal')
          plt.legend()
          plt.tight_layout()
          plt.show()
```

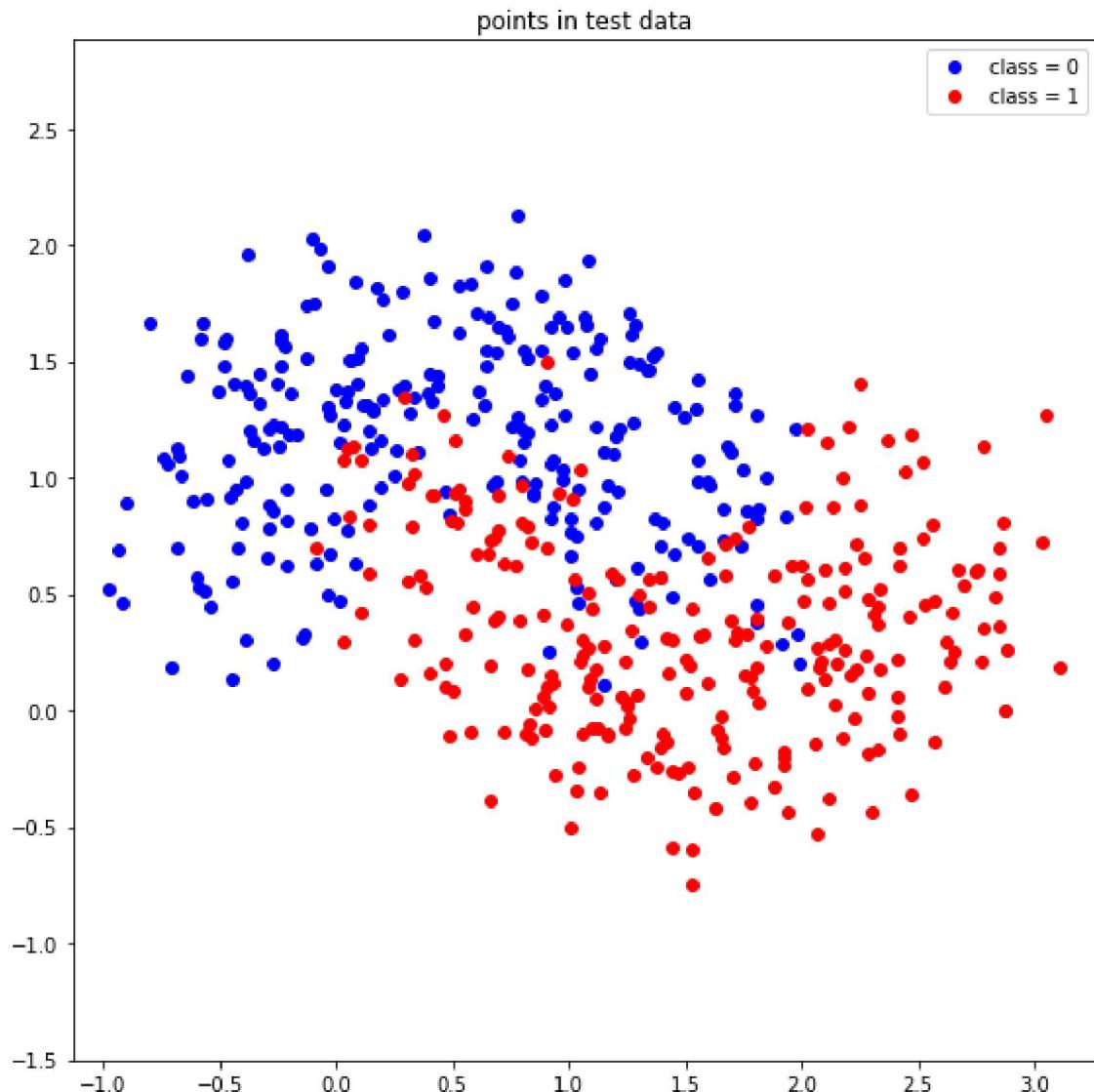


```
In [ ]: f = plt.figure(figsize=(8,8))
```

```

plt.title('points in test data')
plt.plot(data_test_point_x_class_0, data_test_point_y_class_0, 'o', color='blue', label='class = 0')
plt.plot(data_test_point_x_class_1, data_test_point_y_class_1, 'o', color='red', label='class = 1')
plt.axis('equal')
plt.legend()
plt.tight_layout()
plt.show()

```



define the feature functions

- feature vector is defined by $(1, f_1(x, y), f_2(x, y), \dots, f_{k-1}(x, y)) \in \mathbb{R}^k$

```
In [ ]: def compute_feature(point):

    # ++++++#
    # complete the blanks
    #

    x = point[:, 0]
    y = point[:, 1]

    feature = np.array([np.ones(x.shape), 9*x, 6*y, 5*(x**2)*y, -10*x*y, -5*x**2,
    #
```

```
# ++++++
return feature
```

define the linear regression function

- $\theta = (\theta_0, \theta_1, \dots, \theta_{k-1}) \in \mathbb{R}^k$
- feature = $(1, f_1(x, y), \dots, f_{k-1}(x, y)) \in \mathbb{R}^k$

```
In [ ]: def compute_linear_regression(theta, feature):  
    # ++++++  
    # complete the blanks  
    #  
    value = np.inner(theta, feature)  
    #  
    # ++++++  
    return value
```

define sigmoid function with input

- $z \in \mathbb{R}$

```
In [ ]: def sigmoid(z):  
    # ++++++  
    # complete the blanks  
    #  
    value = 1 / (1 + np.exp(-z))  
    #  
    # ++++++  
    return value
```

define the logistic regression function

- $\theta = (\theta_0, \theta_1, \dots, \theta_{k-1}) \in \mathbb{R}^k$
- feature = $(1, f_1(x, y), \dots, f_{k-1}(x, y)) \in \mathbb{R}^k$

```
In [ ]: def compute_logistic_regression(theta, feature):  
    # ++++++  
    # complete the blanks  
    #  
    f = compute_linear_regression(theta, feature)  
    value = sigmoid(f)
```

```

#
# ++++++
#
return value

```

define the residual function

- $\theta = (\theta_0, \theta_1, \dots, \theta_{k-1}) \in \mathbb{R}^k$
- feature = $(1, f_1(x, y), \dots, f_{k-1}(x, y)) \in \mathbb{R}^k$
- label = $l \in \{0, 1\}^k$

```
In [ ]: def compute_residual(theta, feature, label):

    # ++++++
    # complete the blanks
    #

    h = compute_logistic_regression(theta, feature)
    residual = -(label * np.log(h)) + ((1 - label) * np.log(1 - h))

    #
    # ++++++

    return residual
```

define the loss function for the logistic regression

- $\theta = (\theta_0, \theta_1, \dots, \theta_{k-1}) \in \mathbb{R}^k$
- feature = $(1, f_1(x, y), \dots, f_{k-1}(x, y)) \in \mathbb{R}^k$
- label = $l \in \{0, 1\}^k$

```
In [ ]: def compute_loss(theta, feature, label, alpha):

    # ++++++
    # complete the blanks
    #

    n = feature.shape[0]
    r = compute_residual(theta, feature, label)
    loss = (np.sum(r) / n) + ((alpha * np.inner(theta.T, theta)) / 2)

    #
    # ++++++

    return loss
```

define the gradient of the loss with respect to the model parameter θ

- $\theta = (\theta_0, \theta_1, \dots, \theta_{k-1}) \in \mathbb{R}^k$
- feature = $(1, f_1(x, y), \dots, f_{k-1}(x, y)) \in \mathbb{R}^k$
- label = $l \in \{0, 1\}^k$

```
In [ ]: def compute_gradient(theta, feature, label, alpha):

    # ++++++
    # complete the blanks
    #

    n = feature.shape[0]
    h = compute_logistic_regression(theta, feature)
    gradient = (np.sum((h.reshape(n,1) - label.reshape(n,1)) * feature, axis = 0) / 

    #
    # ++++++

    return gradient
```

compute the accuracy of the prediction for point with a given model parameter

```
In [ ]: def compute_accuracy(theta, feature, label):

    # ++++++
    # complete the blanks
    #

    n = feature.shape[0]
    h = compute_logistic_regression(theta, feature)
    correct = 0

    for i in range(n) :
        if (h[i] >= 0.5 and label[i] == 1) or (h[i] < 0.5 and label[i] == 0) :
            correct = correct + 1

    accuracy = correct / n

    #
    # ++++++

    return accuracy
```

initialize the gradient descent algorithm

```
In [ ]: number_iteration      = 1000 # you can change this value as you want
learning_rate           = 0.03 # you can change this value as you want
number_feature          = 7 # you can change this value as you want
alpha                   = 0.1 # you can change this value as you want

theta                  = np.zeros(number_feature)
loss_iteration_train   = np.zeros(number_iteration)
loss_iteration_test    = np.zeros(number_iteration)
accuracy_iteration_train = np.zeros(number_iteration)
accuracy_iteration_test = np.zeros(number_iteration)
```

run the gradient descent algorithm to optimize the loss function with respect to the model parameter

```
In [ ]: for i in range(number_iteration):
```

```

# ++++++
# complete the blanks
#
theta          = theta - (learning_rate * compute_gradient(theta, compute_feature(data_train_point), data_train_label))
loss_train     = compute_loss(theta, compute_feature(data_train_point), data_train_label)
loss_test      = compute_loss(theta, compute_feature(data_test_point), data_test_label)
accuracy_train = compute_accuracy(theta, compute_feature(data_train_point), data_train_label)
accuracy_test  = compute_accuracy(theta, compute_feature(data_test_point), data_test_label)

#
# ++++++
loss_iteration_train[i]    = loss_train
loss_iteration_test[i]      = loss_test
accuracy_iteration_train[i] = accuracy_train
accuracy_iteration_test[i]  = accuracy_test

theta_optimal = theta

```

functions for presenting the results

```
In [ ]: def function_result_01():

    print("final training accuracy = {:.13.10f}".format(accuracy_iteration_train[-1]))
```

```
In [ ]: def function_result_02():

    print("final testing accuracy = {:.13.10f}".format(accuracy_iteration_test[-1]))
```

```
In [ ]: def function_result_03():

    plt.figure(figsize=(8,6))
    plt.title('training loss')

    plt.plot(loss_iteration_train, '--', color='red')
    plt.xlabel('iteration')
    plt.ylabel('loss')

    plt.tight_layout()
    plt.show()
```

```
In [ ]: def function_result_04():

    plt.figure(figsize=(8,6))
    plt.title('testing loss')

    plt.plot(loss_iteration_test, '--', color='red')
    plt.xlabel('iteration')
    plt.ylabel('loss')
```

```
plt.tight_layout()  
plt.show()
```

```
In [ ]: def function_result_05():  
  
    plt.figure(figsize=(8,6))  
    plt.title('training accuracy')  
  
    plt.plot(accuracy_iteration_train, '--', color='red')  
    plt.xlabel('iteration')  
    plt.ylabel('accuracy')  
  
    plt.tight_layout()  
    plt.show()
```

```
In [ ]: def function_result_06():  
  
    plt.figure(figsize=(8,6))  
    plt.title('testing accuracy')  
  
    plt.plot(accuracy_iteration_test, '--', color='red')  
    plt.xlabel('iteration')  
    plt.ylabel('accuracy')  
  
    plt.tight_layout()  
    plt.show()
```

plot the linear regression values over the 2-dimensional Euclidean space and superimpose the training data

```
In [ ]: def function_result_07():  
  
    plt.figure(figsize=(8,8))  
    plt.title('linear regression values on the training data')  
  
    min_x = np.min(data_train_point_x)  
    max_x = np.max(data_train_point_x)  
    min_y = np.min(data_train_point_y)  
    max_y = np.max(data_train_point_y)  
  
    X = np.arange(min_x - 0.5, max_x + 0.5, 0.1)  
    Y = np.arange(min_y - 0.5, max_y + 0.5, 0.1)  
  
    [XX, YY] = np.meshgrid(X, Y)  
  
    # ++++++  
    # complete the blanks  
    #  
  
    ZZ = np.array([np.ones(XX.shape), 9*XX, 6*YY, 5*(XX**2)*YY, -10*XX*YY, -5*XX**  
    linear_regression = theta_optimal[0]*ZZ[0] + theta_optimal[1]*ZZ[1] + theta_opt  
    + theta_optimal[4]*ZZ[4] + theta_optimal[5]*ZZ[5] + theta_optimal[6]*ZZ[6]  
    plt.scatter(XX, YY, c=linear_regression, cmap=plt.cm.bwr, s = 200, marker = 's')  
    plt.grid()  
  
    classifier = np.where((linear_regression < 0.1) & (linear_regression > -0.1))  
    plt.plot(XX[classifier], YY[classifier], 'x', color='black')  
    plt.colorbar()
```

```

#
# ++++++=====+++++=====+++++=====+++++=====

plt.plot(data_train_point_x_class_0, data_train_point_y_class_0, '.', color='blue')
plt.plot(data_train_point_x_class_1, data_train_point_y_class_1, '.', color='red')

plt.legend()
plt.tight_layout()
plt.show()

```

```

In [ ]: def function_result_08():

    plt.figure(figsize=(8,8))
    plt.title('linear regression values on the testing data')

    min_x = np.min(data_test_point_x)
    max_x = np.max(data_test_point_x)
    min_y = np.min(data_test_point_y)
    max_y = np.max(data_test_point_y)

    X = np.arange(min_x - 0.5, max_x + 0.5, 0.1)
    Y = np.arange(min_y - 0.5, max_y + 0.5, 0.1)

    [XX, YY] = np.meshgrid(X, Y)

    #
    # complete the blanks
    #

    ZZ = np.array([np.ones(XX.shape), 9*XX, 6*YY, 5*(XX**2)*YY, -10*XX*YY, -5*XX**2*YY])
    linear_regression = theta_optimal[0]*ZZ[0] + theta_optimal[1]*ZZ[1] + theta_optimal[2]*ZZ[2] + theta_optimal[3]*ZZ[3] + theta_optimal[4]*ZZ[4] + theta_optimal[5]*ZZ[5] + theta_optimal[6]*ZZ[6]
    plt.scatter(XX, YY, c=linear_regression, cmap=plt.cm.bwr, s = 200, marker = 's')
    plt.grid()

    classifier = np.where((linear_regression < 0.1) & (linear_regression > -0.1))
    plt.plot(XX[classifier], YY[classifier], 'x', color='black')
    plt.colorbar()

    #
    # ++++++=====+++++=====+++++=====+++++=====

    plt.plot(data_test_point_x_class_0, data_test_point_y_class_0, '.', color='blue')
    plt.plot(data_test_point_x_class_1, data_test_point_y_class_1, '.', color='red')

    plt.legend()
    plt.tight_layout()
    plt.show()

```

plot the logistic regression values over the 2-dimensional Euclidean space

```

In [ ]: def function_result_09():

    plt.figure(figsize=(8,8))
    plt.title('logistic regression values on the training data')

    min_x = np.min(data_train_point_x)
    max_x = np.max(data_train_point_x)
    min_y = np.min(data_train_point_y)
    max_y = np.max(data_train_point_y)

```

```
X = np.arange(min_x - 0.5, max_x + 0.5, 0.1)
Y = np.arange(min_y - 0.5, max_y + 0.5, 0.1)

[XX, YY] = np.meshgrid(X, Y)

# ++++++
# complete the blanks
#
ZZ = np.array([np.ones(XX.shape), 9*XX, 6*YY, 5*(XX**2)*YY, -10*XX*YY, -5*XX*YY,
z = theta_optimal[0]*ZZ[0] + theta_optimal[1]*ZZ[1] + theta_optimal[2]*ZZ[2] +
    + theta_optimal[4]*ZZ[4] + theta_optimal[5]*ZZ[5] + theta_optimal[6]*ZZ[6]
logistic_regression = 1 / (1 + np.exp(-z))
plt.scatter(XX, YY, c=logistic_regression, cmap=plt.cm.bwr, s = 200, marker =
plt.grid()
plt.colorbar()

#
# ++++++
plt.plot(data_train_point_x_class_0, data_train_point_y_class_0, '.', color='blue')
plt.plot(data_train_point_x_class_1, data_train_point_y_class_1, '.', color='red')

plt.legend()
plt.tight_layout()
plt.show()
```

```
In [ ]: def function_result_10():

    plt.figure(figsize=(8,8))
    plt.title('logistic regression values on the testing data')

    min_x = np.min(data_test_point_x)
    max_x = np.max(data_test_point_x)
    min_y = np.min(data_test_point_y)
    max_y = np.max(data_test_point_y)

    X = np.arange(min_x - 0.5, max_x + 0.5, 0.1)
    Y = np.arange(min_y - 0.5, max_y + 0.5, 0.1)

    [XX, YY] = np.meshgrid(X, Y)

    # ++++++
    # complete the blanks
    #

    ZZ = np.array([np.ones(XX.shape), 9*XX, 6*YY, 5*(XX**2)*YY, -10*XX*YY, -5*XX*YY,
    z = theta_optimal[0]*ZZ[0] + theta_optimal[1]*ZZ[1] + theta_optimal[2]*ZZ[2] +
        + theta_optimal[4]*ZZ[4] + theta_optimal[5]*ZZ[5] + theta_optimal[6]*ZZ[6]
    logistic_regression = 1 / (1 + np.exp(-z))
    plt.scatter(XX, YY, c=logistic_regression, cmap=plt.cm.bwr, s = 200, marker =
    plt.grid()
    plt.colorbar()

    #
    # ++++++
    # ++++++

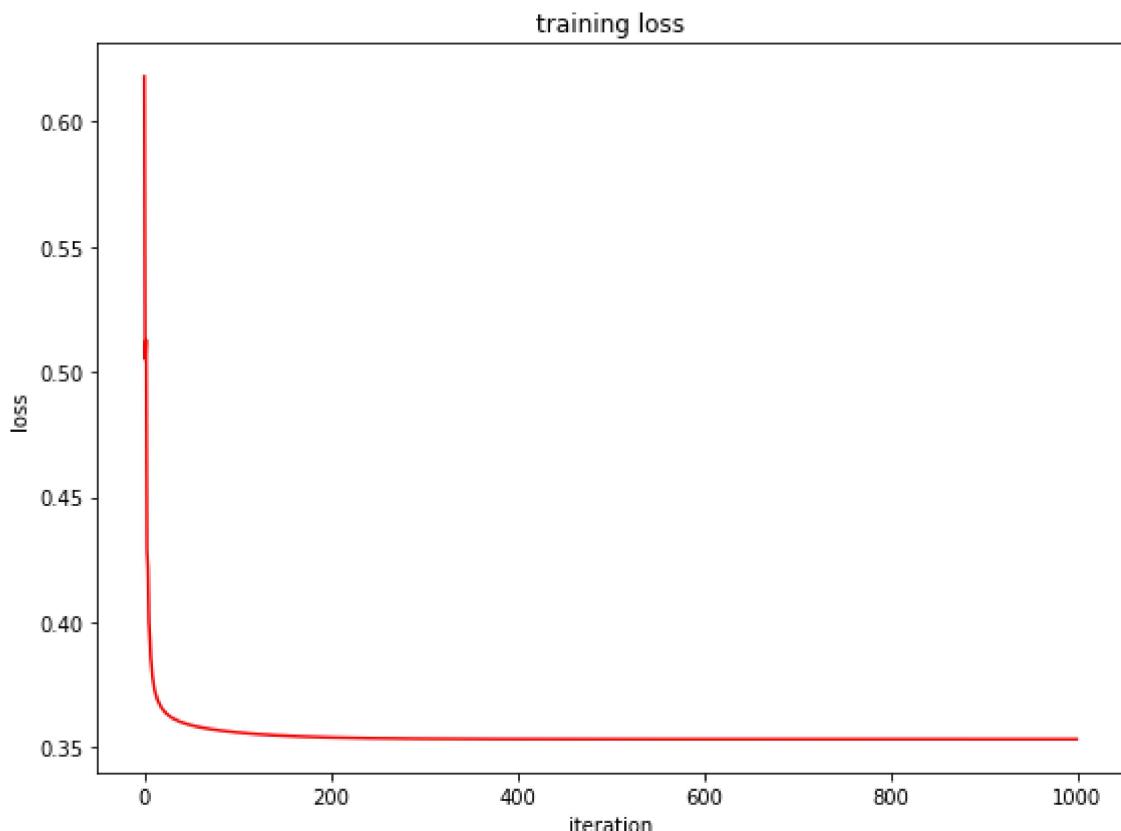


    plt.plot(data_test_point_x_class_0, data_test_point_y_class_0, '.', color='blue')
    plt.plot(data_test_point_x_class_1, data_test_point_y_class_1, '.', color='red',
```

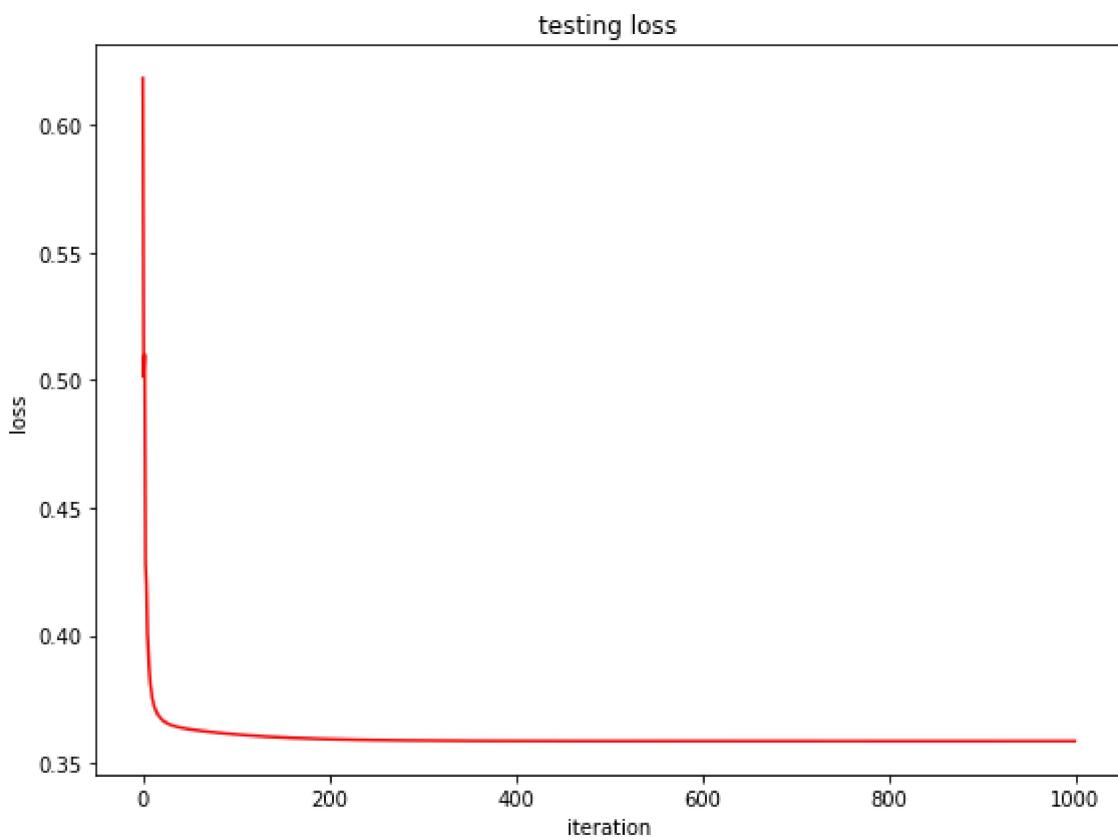
```
plt.legend()  
plt.tight_layout()  
plt.show()
```

results

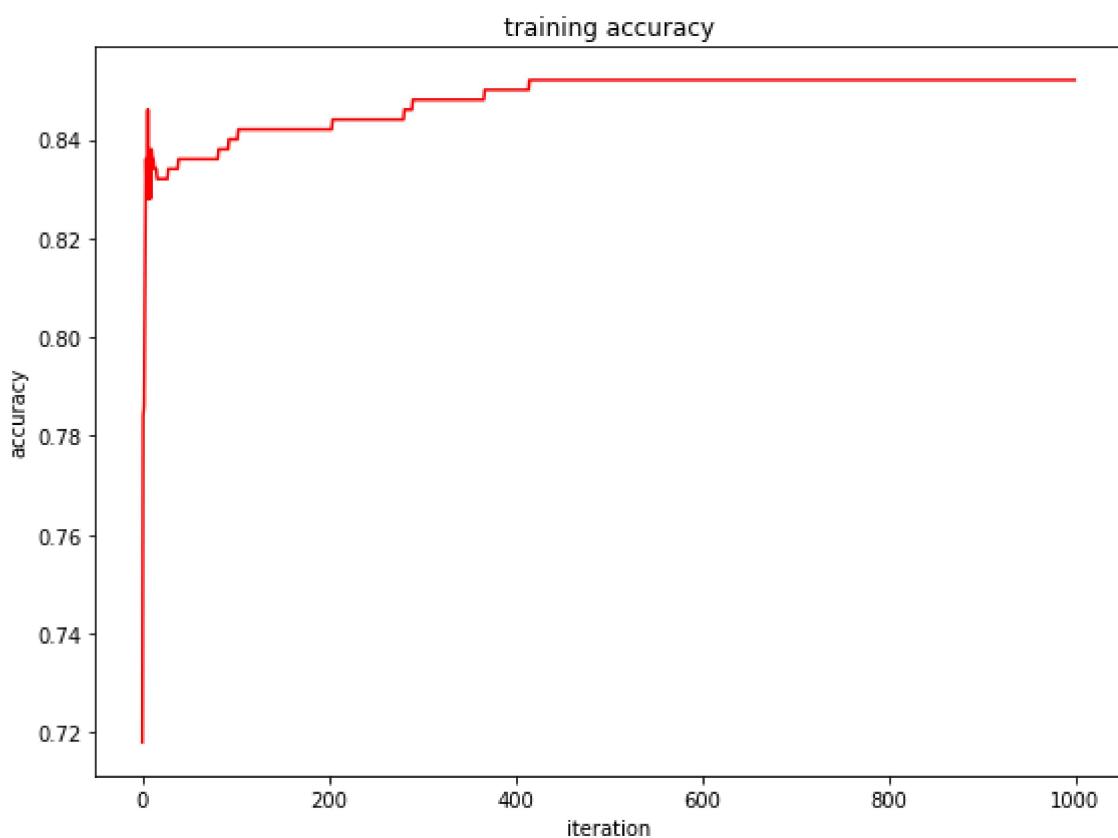
```
In [ ]: number_result = 10  
  
for i in range(number_result):  
    title = '## [RESULT {:02d}]'.format(i+1)  
    name_function = 'function_result_{:02d}()'.format(i+1)  
    print('*****')  
    print(title)  
    print('*****')  
    eval(name_function)  
  
*****  
## [RESULT 01]  
*****  
final training accuracy = 0.8520000000  
*****  
## [RESULT 02]  
*****  
final testing accuracy = 0.8380000000  
*****  
## [RESULT 03]  
*****
```



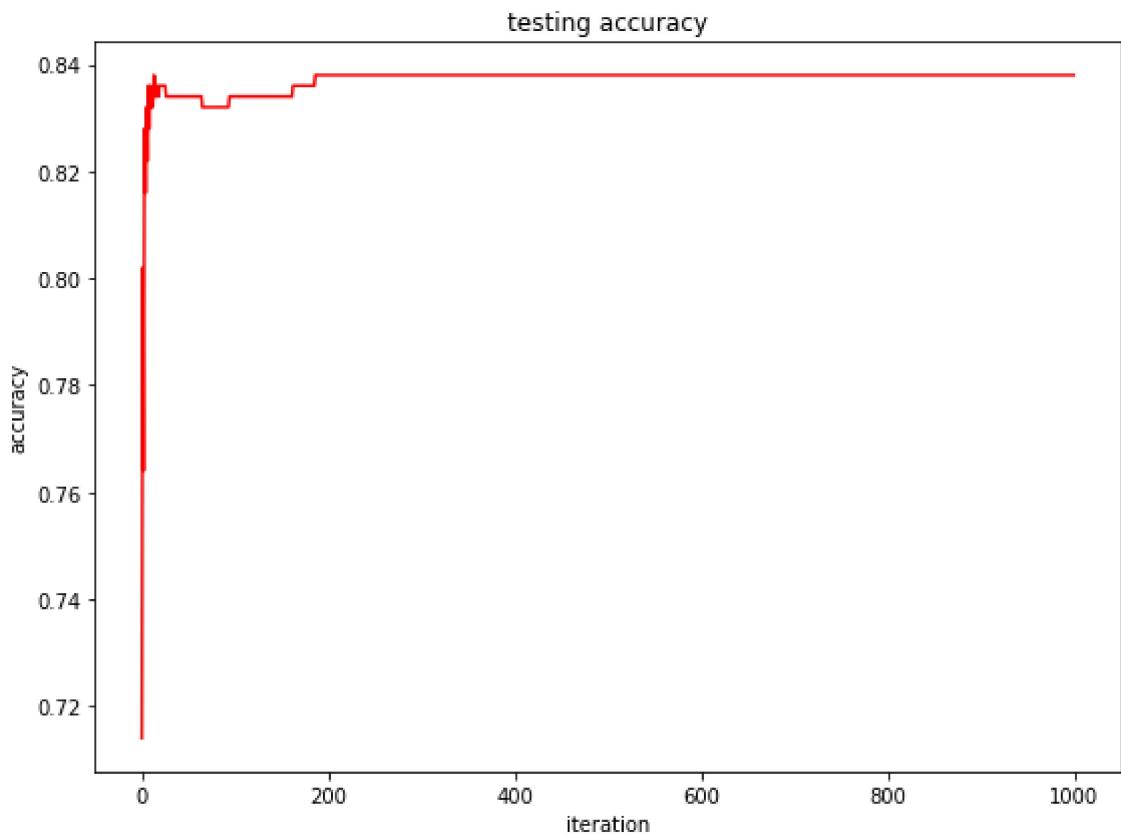
```
*****  
## [RESULT 04]  
*****
```



```
*****  
## [RESULT 05]  
*****
```

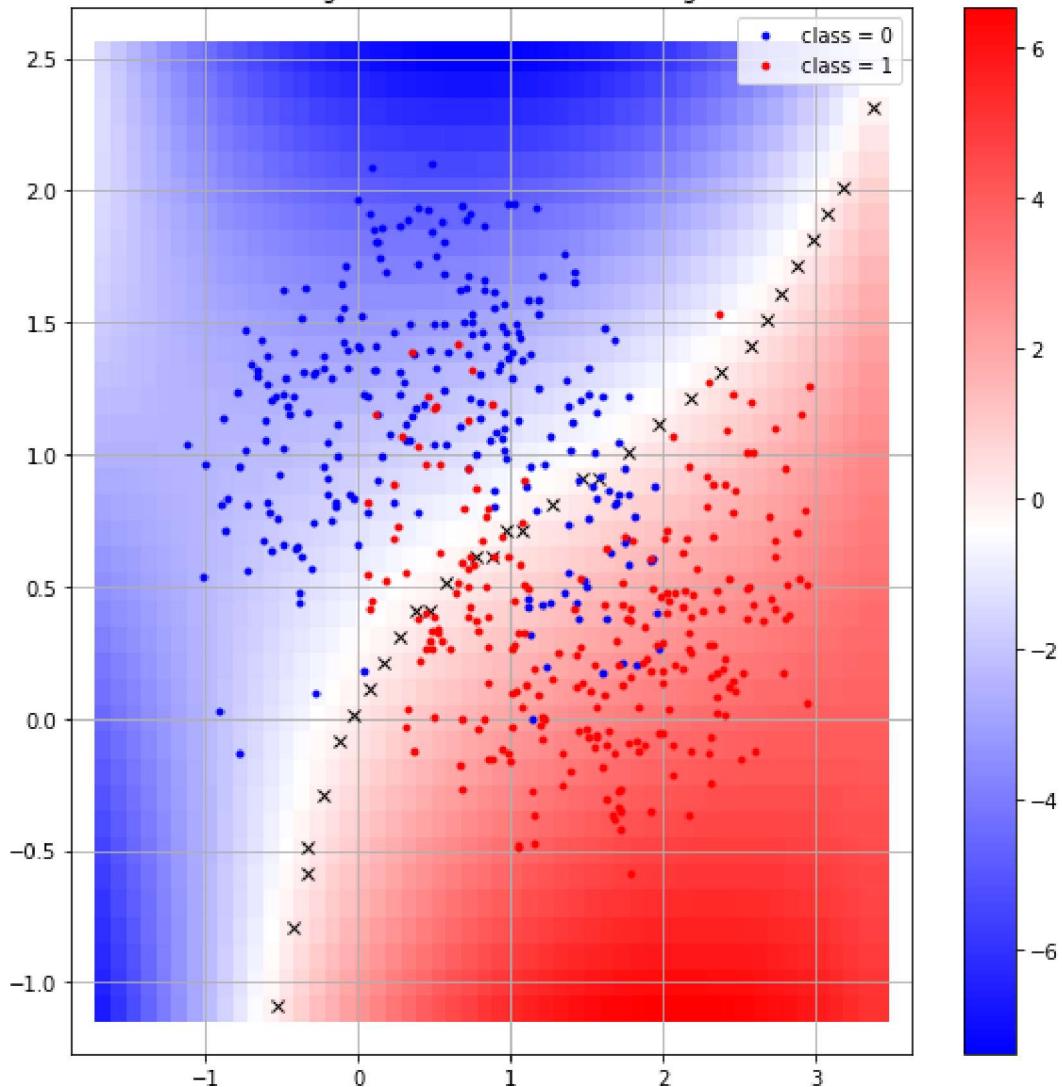


```
*****  
## [RESULT 06]  
*****
```



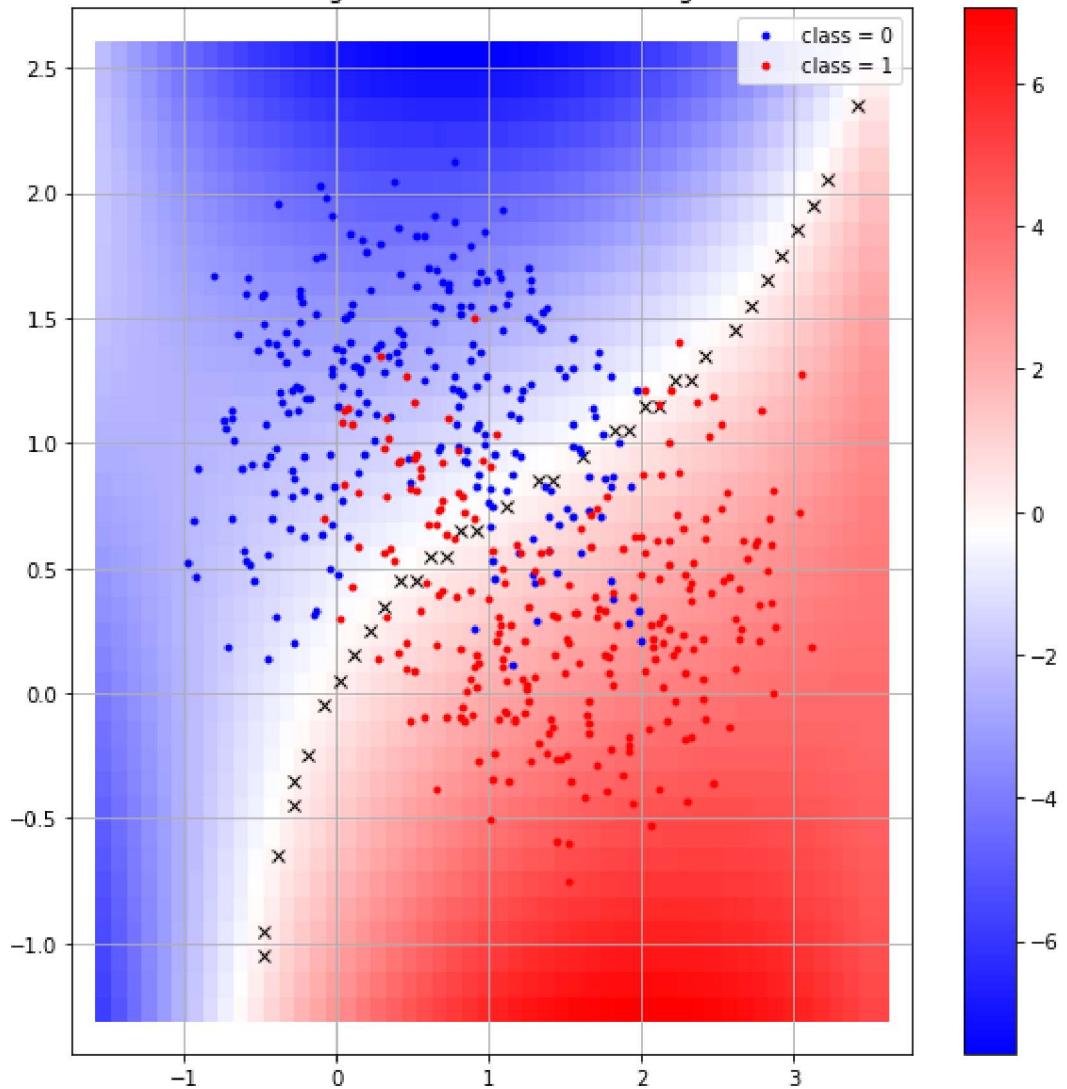
```
*****  
## [RESULT 07]  
*****
```

linear regression values on the training data



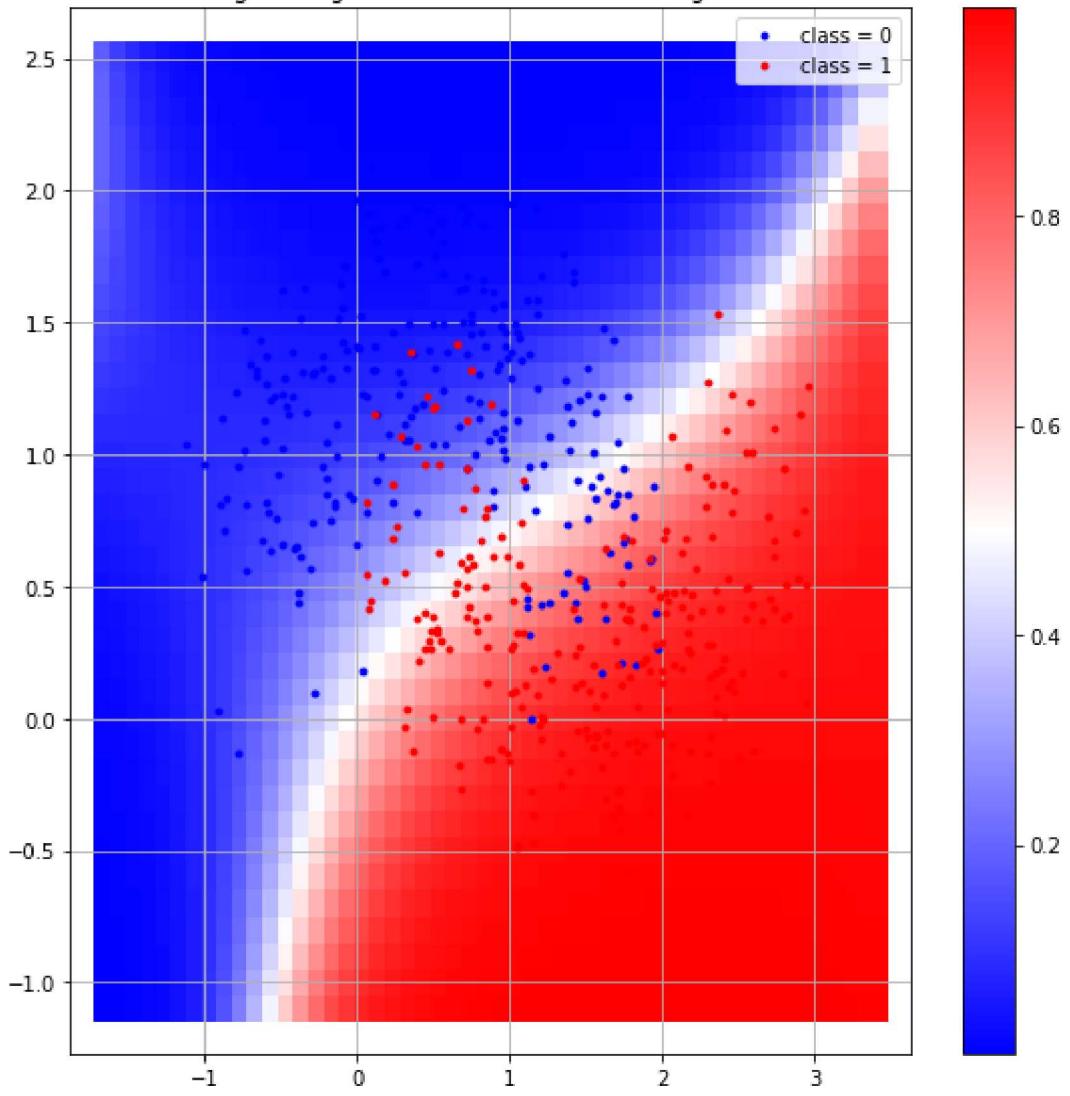
```
*****  
## [RESULT 08]  
*****
```

linear regression values on the testing data



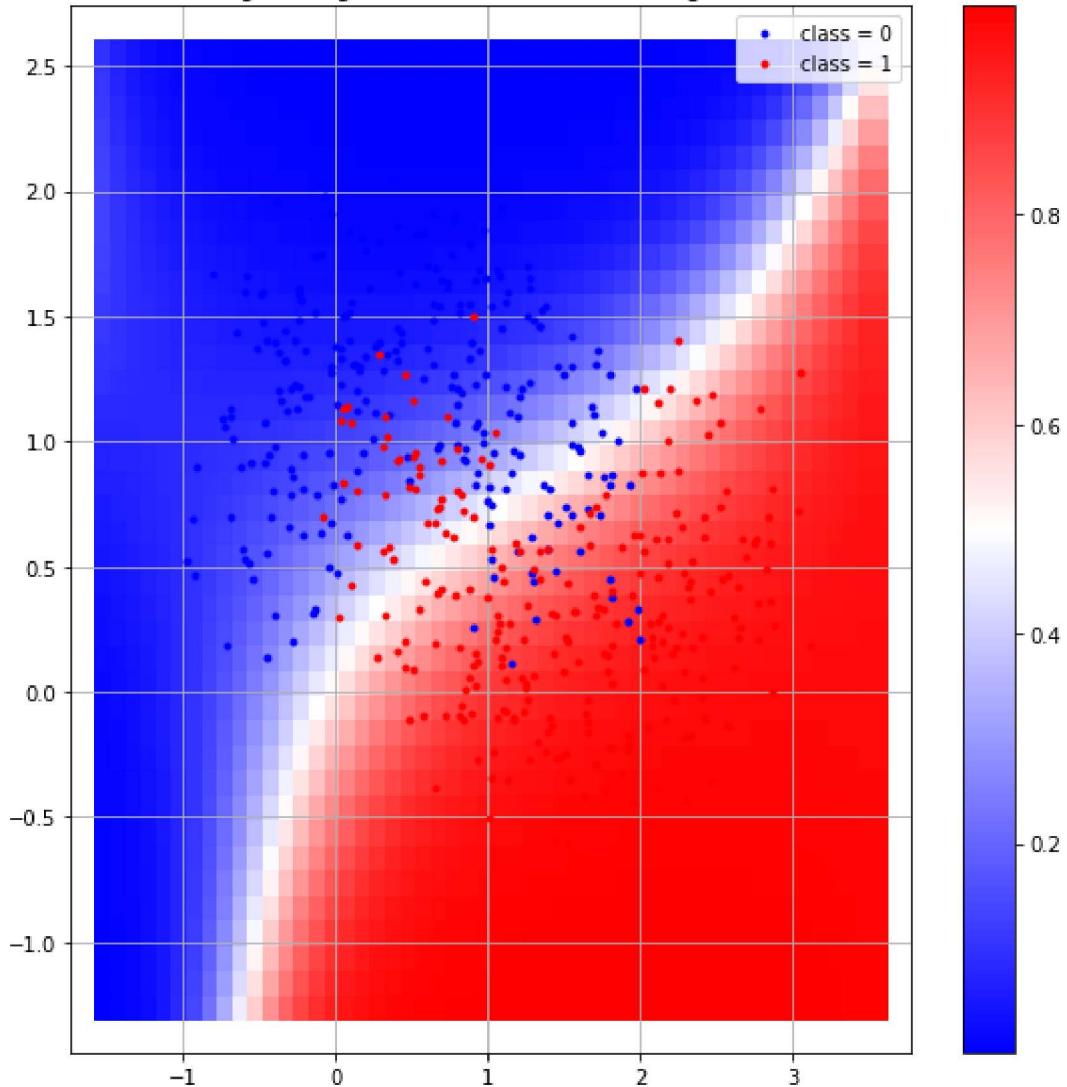
```
*****  
## [RESULT 09]  
*****
```

logistic regression values on the training data



```
*****  
## [RESULT 10]  
*****
```

logistic regression values on the testing data



In []: