EEE418: Advanced Pattern Recognition **Spring 2020**

Lab 3: Discriminant Function

Student: Xun Xu *ID: 1926930*

3.1 Introduction: Theory Introduction

The loss function of the least mean square error algorithm is $J(\theta)$ .The derivation process of its parameter θ is as follows:

1. We assume that the label result(Y) can be calculated according to the parameter θ:

$$X\theta = Y$$

2. Then the loss function can be expressed as follows:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

3. Write the loss function specifically, that is:

$$\begin{aligned} J(\theta) &= \frac{1}{2m}(X\theta - y)^T(X\theta - y) \\ &= \frac{1}{2m}\left(\theta^T X^T - y^T\right)(X\theta - y) \\ &= \frac{1}{2m}\left(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^T y\right) \end{aligned}$$

4. In order to find the minimum value, we derivate this formula to θ:

$$\begin{aligned} \nabla_\theta J(\theta) &= \frac{1}{2}\left(2X^T X\theta - X^T y - \left(y^T X\right)^{\mathrm{T}}\right) \\ &= X^T X\theta - X^T y \end{aligned}$$

5. After a simple formula deformation, we can find the final parameter θ:

$$\begin{aligned} &\text{make } \nabla_\theta J(\theta) = 0 \\ &\text{we will have: } X^T X\theta - X^T y = 0 \\ &\left(X^T X\right)^{-1} X^T X\theta = \left(X^T X\right)^{-1} X^T y \\ &\theta = \left(X^T X\right)^{-1} X^T y \end{aligned}$$

Afterwards, we will implement the Least Mean Squared Error algorithm using the publicly dataset to verify our algorithm.

3.2 Methodology

3.2.1 Data preprocessing:Estimation of Classification Methods

Data preprocessing mainly includes the following result steps:

1. Read file as a list of strings
2. Split the string according to comma
3. Convert character data to numeric data
4. Shuffle the data randomly
5. Divide data and labels into five groups and combine them to facilitate index extraction

The specific code is as follows, each step has a corresponding comment:

```
import numpy as np
import random
import math

# Read file
random.seed(17)
f = open('./breast-cancer-wisconsin.data', 'r+')
dataset = f.readlines()
f.close()
print('original input data','length:',len(dataset), ',part of data[0:2]:',dataset[0:2])
# Separate string data according to commas
for i in range(len(dataset)):
  dataset[i] = dataset[i].split(',')
print('split input data','length:',len(dataset[i]),',part of data[0:2]:',dataset[0:2])
# Convert string data to numeric data
data = []
for i in range(len(dataset)):
  for j in range(len(dataset[0])):
    data.append(int(dataset[i][j]))
# Rearrange to original dimension
new_data = np.asarray(data).reshape(len(dataset), 11)
print('numeric data(raw data)', 'type:', type(new_data[0][0]), ',shape:', new_data.shape)
print('part of data:\n', new_data[0:2])

# Shuffle data with the random. shue method
random.shuffle(new_data)

# Split the dataset as  ve parts
data_set = []
data_label = []
x = new_data
x_data = x[:, 1:x.shape[1]-1]
x_label = x[:, x.shape[1]-1]
index = math.floor(x.shape[0]/5)
data_set0 = x_data[0:index]
data_set1 = x_data[index:index*2]
```

```
data_set2 = x_data[index*2:index*3]
data_set3 = x_data[index*3:index*4]
data_set4 = x_data[index*4:]
data_label0 = x_label[0:index]
data_label1 = x_label[index:index*2]
data_label2 = x_label[index*2:index*3]
data_label3 = x_label[index*3:index*4]
data_label4 = x_label[index*4:]
# five sets
print('five dataset', 'data1 shape:',data_set0.shape,'data2 shape:',data_set1.shape,'data3
                                                shape:',data_set2.shape,'data4 shape:',data_set3
                                                .shape,'data5 shape:',data_set4.shape,'part of
                                                data[0:5]:',data_set0[0:5])
# five labels
print('five labelset', ',data1 label:',data_label0.shape,'data2 label:',data_label1.shape,'
                                                data3 label:',data_label2.shape,'data4 label:',
                                                data_label3.shape,'data5 label:',data_label4.
                                                shape,'part of label[0:5]:',data_label0[0:5])
# Splicing data into a three-dimensional array for easy index extraction in training and test
whole_data = np.array([data_set0,data_set1,data_set2,data_set3,data_set4])
whole_label = np.array([data_label0,data_label1,data_label2,data_label3,data_label4])
```

The output of this part of the code is shown in figure 3.1, we can see that we have completed the requirements of estimation of classification methods:

```
original input data
length: 683
part of data[0:2]: ['1000025,5,1,1,1,2,1,3,1,1,2\n', '1002945,5,4,4,5,7,10,3,2,1,2\n']
split input data
length: 683
part of data[0:2]: ['1000025', '5', '1', '1', '1', '2', '1', '3', '1', '1', '2\n']
 ['1002945', '5', '4', '4', '5', '7', '10', '3', '2', '1', '2\n']
numeric data(raw data)
type: <class 'numpy.int64'>
shape: (683, 11)
five dataset
data1 shape: (136, 9)
data2 shape: (136, 9)
data3 shape: (136, 9)
data4 shape: (136, 9)
data5 shape: (139, 9)
part of data[0:5]: [[5 1 1 1 2 1 3 1 1]
 [5 1 1 1 2 1 3 1 1]
 [3 1 1 1 2 2 3 1 1]
 [5 1 1 1 2 1 3 1 1]
 [3 1 1 1 2 2 3 1 1]]
five labelset
data1 label: (136,)
data2 label: (136,)
data3 label: (136,)
data4 label: (136,)
data5 label: (139,)
part of label[0:5]: [2 2 2 2 2]
```

Figure 3.1: Data preprocessing outcome:

3.2.2 Least Mean Squared Error Algorithm

The LMS algorithm mainly includes the following steps, as an iterative process, because there are many cycles of cross-validation, in order to prevent too many output results, for convenience, here I use a copy of all samples as a complete training set:

1. Training stage: Augment the feature vector x with an additional constant dimension,

$$\hat{X} = \begin{bmatrix} X & \mathbf{1}_{N\times 1} \end{bmatrix}$$

The specific code is as follows:

```
# Training stage
# Augment the feature vector x with an additional constant dimension
training_set = np.insert(training_set, training_set.shape[1], values=1, axis=1)
print('training data plus 1','\ntype:\n',type(training_set[0][0]),'\npart of data:\n',
                                                    training_set[0:5])
```

The result is shown in the figure 3.2

```
training data plus 1
type:
 <class 'numpy.float64'>
part of data:
 [[5. 1. 1. 1. 2. 1. 3. 1. 1. 1.]
 [5. 1. 1. 1. 2. 1. 3. 1. 1. 1.]
 [3. 1. 1. 1. 2. 2. 3. 1. 1. 1.]
 [5. 1. 1. 1. 2. 1. 3. 1. 1. 1.]
 [3. 1. 1. 1. 2. 2. 3. 1. 1. 1.]]
```

Figure 3.2: the feature vector x with an additional constant dimension:

2. Scale linearly the attribute values xij into [−1,1] for each dimensional feature as follows:

$$x_{ij} \leftarrow 2\frac{x_{ij} - \min_i x_{ij} + 10^{-6}}{\max_i x_{ij} - \min_i x_{ij} + 10^{-6}} - 1$$

The specific code is as follows:

```
# Scale linearly the attribute values xij into -[1,1] for each dimensional feature as
                                                    follows:
# method 1
max_i = np.max(training_set,axis=0)
min_i = np.min(training_set,axis=0)
training_set = 2*(training_set-min_i+10**-6)/(max_i-min_i+10**-6)-1
# method 2
# for i in range(training_set.shape[1]):
#     one_column = training_set[:,i]
#     min = np.min(one_column)
#     max = np.max(one_column)
#     training_set[:,i] = 2*(one_column-min+10**-6)/(max-min+10**-6) - 1
```

The result is shown in the figure 3.3

```
attribute values into [-1,1]:
 [[-0.11111099 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.99999978
  -0.55555538 -0.99999978 -0.99999978  1.        ]
 [-0.11111099 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.99999978
  -0.55555538 -0.99999978 -0.99999978  1.        ]
 [-0.55555538 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.77777758
  -0.55555538 -0.99999978 -0.99999978  1.        ]
 [-0.11111099 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.99999978
  -0.55555538 -0.99999978 -0.99999978  1.        ]
 [-0.55555538 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.77777758
  -0.55555538 -0.99999978 -0.99999978  1.        ]]
```

Figure 3.3: Scale linearly the attribute values xij into [−1,1]

3. Reset the example vector x according its label y

$$x \leftarrow \begin{cases} x, & y \in \omega_1 \\ -x, & y \in \omega_2 \end{cases}$$

The specific code is as follows:

```
#  Reset the example vector x according its label y
for i in range(training_set.shape[0]):
if(training_label[i] == 4):
training_set[i] = -training_set[i]
print('resit\n',training_set[0:5],'end')
```

The result is shown in the figure 3.4

```
resit
 [[-0.11111099 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.99999978
  -0.55555538 -0.99999978 -0.99999978  1.        ]
 [-0.11111099 -0.33333319 -0.33333319 -0.11111099  0.33333341  1.
  -0.55555538 -0.77777758 -0.99999978  1.        ]
 [-0.55555538 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.77777758
  -0.55555538 -0.99999978 -0.99999978  1.        ]
 [ 0.11111121  0.5555556   0.5555556  -0.99999978 -0.55555538 -0.33333319
  -0.55555538  0.33333341 -0.99999978  1.        ]
 [-0.33333319 -0.99999978 -0.99999978 -0.55555538 -0.77777758 -0.99999978
  -0.55555538 -0.99999978 -0.99999978  1.        ]]
```

Figure 3.4: Reset the example vector x according its label y

4. Find the weight vector w

$$w \leftarrow \left(X^T X\right)^{-1} X^T \mathbf{1}$$

The specific code is as follows:

```
#  Find the weight vector w
one =np.ones([training_set.shape[0], 1])
k = np.matmul(training_set.transpose(), training_set)
w = np.matmul(np.dot(np.linalg.inv(k), training_set.transpose()), one)
print('weight\n','\nshape:\n', w.shape,',\nvalue:',w)
```

The result is shown in the figure 3.5

```
weight
shape:
 (10, 1) ,
value: [[-0.28541786]
 [-0.19660513]
 [-0.14075671]
 [-0.07418946]
 [-0.09067592]
 [-0.40847657]
 [-0.17258058]
 [-0.16676402]
 [-0.00880996]
 [-0.39212606]]
```

Figure 3.5: the weight vector w

5. Predict the example x

$$c = \begin{cases} \omega_1, & w^T \begin{bmatrix} x \\ 1 \end{bmatrix} \geq 0 \\ \omega_2, & \text{otherwise} \end{cases}$$

The specific code is as follows:

```
# Predict the example x
test_set = np.float64(x_data[:])
test_label = x_label[:]
test_set = np.insert(test_set, test_set.shape[1], values=1, axis=1)
for i in range(10):
  one_column = test_set[:,i]
min = np.min(one_column)
max = np.max(one_column)
test_set[:,i] = 2*(one_column-min+10**-6)/(max-min+10**-6) - 1
print('test_set  to [-1,1]','\nshape:',test_set.shape,'\npart value:\n',test_set[0:5])
new_label = np.dot(test_set, w)
print('part new label\n',new_label[0:5])
decision_value = []
for i in range(new_label.shape[0]):
if(new_label[i] >= 0):
decision_value.append(2)
else:
decision_value.append(4)
print('decision_value\n',decision_value[0:10])
```

The result is shown in the figure 3.6

```
test_set  to [-1,1]
shape: (683, 10)
part value:
 [[-0.11111099 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.99999978
  -0.55555538 -0.99999978 -0.99999978  1.        ]
 [-0.11111099 -0.33333319 -0.33333319 -0.11111099  0.33333341  1.
  -0.55555538 -0.77777758 -0.99999978  1.        ]
 [-0.55555538 -0.99999978 -0.99999978 -0.99999978 -0.77777758 -0.77777758
  -0.55555538 -0.99999978 -0.99999978  1.        ]
 [ 0.11111121  0.5555556   0.5555556  -0.99999978 -0.55555538 -0.33333319
  -0.55555538  0.33333341 -0.99999978  1.        ]
 [-0.33333319 -0.99999978 -0.99999978 -0.55555538 -0.77777758 -0.99999978
  -0.55555538 -0.99999978 -0.99999978  1.        ]]
part new label
 [[ 0.80159239]
 [-0.44402438]
 [ 0.8376722 ]
 [-0.3014387 ]
 [ 0.83204549]]
decision_value
 [2, 4, 2, 4, 2, 4, 2, 2, 2, 2]
```

Figure 3.6: Predict the example x

6. Test stage : Estimate the accuracy on the test set, that is the fraction of the correctly classified examples in the total test set. Here I also copied the full sample as the test set.

   The specific code is as follows:

```python
# Test stage : Estimate the accuracy
num = np.asarray(decision_value)-test_label
rate = np.sum(num == 0)/test_label.shape[0]
accuracy_rate.append(rate)
print('the accuracy tare:', accuracy_rate)
```

   The result is shown in the figure 3.7

```
the accuracy rate is: [0.9604685212298683]
```

Figure 3.7: the accuracy on the test set

7. Repeat 5-cross-fold validation for 5 times to average all accuracy performance. In this part, I posted the code of the main loop. The intermediate steps are the same as before. Each time, only the name of the incoming training set and test set needs to be changed to implement a new calculation process. The specific code is as follows:

```python
for num in range(5):
  for liter in range(len(str)+1):
    str = [0, 1, 2, 3, 4]
    str.remove(liter)
```

```
    training_set = np.vstack((whole_data[str[0]],whole_data[str[1]],whole_data[str[2]],
                                                 whole_data[str[3]]))[:]
    training_label = np.hstack((whole_label[str[0]],whole_label[str[1]],whole_label[str[2]],
                                                 whole_label[str[3]]))[:]
    test_set = whole_data[liter][:]
    test_label = whole_label[liter][:]

    training_set = np.float64(training_set)
    test_set = np.float64(test_set)
            num = np.asarray(c)-test_label
    rate = np.sum(num == 0)/test_label.shape[0]
    accuracy_rate.append(rate)
```

The result is shown in the figure 3.8

```
the average accuracy rate is:
  0.958950486669488
 length of calculation:
 25
specific average accuracy:
 [[0.94117647]
 [0.97058824]
 [0.97058824]
 [0.94117647]
 [0.97122302]
 [0.94117647]
 [0.97058824]
 [0.97058824]
 [0.94117647]
 [0.97122302]
 [0.94117647]
 [0.97058824]
 [0.97058824]
 [0.94117647]
 [0.97122302]
 [0.94117647]
 [0.97058824]
 [0.97058824]
 [0.94117647]
 [0.97122302]
 [0.94117647]
 [0.97058824]
 [0.97058824]
 [0.94117647]
 [0.97122302]]
```

Figure 3.8: 5-cross-fold validation for 5 times

3.3 Analysis of results

Least squares and gradient descent algorithms are both methods for us to optimize parameters. In neural networks, I also use gradient descent a lot. Here is a simple comparison.
Same points:
(1)
The essence is the same: both methods calculate a general valuation function for the dependent variable given the known data (dependent variable and independent variable), or a fitting process for the data distribution , And then test the given new independent variables with the calculated.
(2)
The same goal: all within the framework of known data, so that the square sum of the difference between the estimated value and the actual value is as small as possible, and the formula of the square sum of the difference between the estimated value and the actual value is generally consistent.
Difference:
(1) The least square method in the narrow sense is a parameter solution method with closed-form solution under the linear assumption, and the final result is globally optimal;
(2) Gradient descent method, which assumes a wider range of conditions (unconstrained), is a parameter optimization method that is carried out stepwise through iterative updates, and the final result is locally optimal;
(3) The generalized least square criterion is an evaluation criterion for the degree of deviation, which is different from the above two.

3.4 Appendix: codes

The code consists of two parts. The first is 418lab3.py that it does not contain a calculation process, including the results of each step, and the second is 418lab3loop.py that it includes the cross-validation process.