Machine Learning: HW2 Report

B03901034 吳泓霖

Logistic regression function

```
# This function will return the result of the linear function
# e.g. y = w1x1 + w2x2 + .... + b
def estim(data):
    num = sum(map(mul, data, params[:len(data)]))
    num += params[len(params) - 1]
    return num

# This function will apply the number to sigmoid function
def sigmoid(num):
    try:
        return 1 / (1 + math.exp(0 - num))
    except OverflowError:
        return 0.0
# Adagrad
def rate(grad_sum):
    return l_rate / math.sqrt(1 if grad_sum == 0 else grad_sum)

# Calculate the result, and find the gradient of the loss function
def gradient(data, ans):
    estimate = sigmoid(estim(data))
    for i in range(len(data)):
        grad[i] -= (ans - estimate) * data[i]
    grad[len(grad) - 1] -= ans - estimate
    return estimate

# Update the parameters using regression
def updateParam():
    global grad
    for i in range(len(params)):
        grad_sq_sum[i] += math.pow(grad[i], 2)
        params[i] -= rate(grad_sq_sum[i]) * grad[i]
    grad = [0.0 for x in range(58)]

# Predict the correctness using training data
def predict():
    ...
```

```python

if sys.argv[1] == '--train':
    # Variables
    train_data = []
    train_ans = []

    # Store all the parameters into an array
    params = [0.0 for x in range(58)]
    # The gradient of each parameters
    grad = [0.0 for x in range(58)]
    # The sum of all previous gradient value, for Adagrad
    grad_sq_sum = [0.0 for x in range(58)]

    loop = int(sys.argv[2])
    l_rate = 0.001

    # Read training data
    ...

    # Training
    for x in range(loop):
        result = []
        for i in range(len(train_data)):
            result.append(gradient(train_data[i], train_ans[i]))
            updateParam()
        # Dump parameter every 10 iterations
        ...

    outfile.close()

elif sys.argv[1] == '--test':
    # Read test and do estimation
    ...
    for line in infile:
        split = line.split(',')
        result += split[0] + ',' + str(int(estim(map(float, split[1:])) > 0))+
    ...
else:
    ...
```

Method 2: Simple Neural Network - Logistic Twice

(Only shows the different part)

```
# Update gradient recursively
# Each recursion is one layer, start from 0
def gradient(data, ans, layer):
    if layer == len(net_params):
        return data[0]
    result = []
    for node in range(len(net_params[layer])):
        estimate = sigmoid(estim(data, net_params[layer][node]))
        result.append(estimate)
        # Update gradient list
        for i in range(len(data)):
            grad[layer][node][i] -= (ans - estimate) * data[i]
        grad[layer][node][len(grad[layer][node]) - 1] -= ans - estimate
    # Minus 0.5 to preserve the distibution ( P(n > 0) = P(n < 0) = 0.5)
    return gradient(map(lambda x: x - 0.5, result), ans, layer + 1)

# Regression for each variable
def updateParam():
    global grad
    for layer in range(len(net_params)):
        for node in range(len(net_params[layer])):
            for i in range(len(net_params[layer][node])):
              grad_sq_sum[layer][node][i] += math.pow(grad[layer][node][i], 2)
              net_params[layer][node][i] -= rate(grad_sq_sum[layer][node][i])\
                 * grad[layer][node][i]
    grad = [[[0.0 for x in range(58)] for x in range(57)] for x in range(2)]

...

if sys.argv[1] == '--train':
    ...
    # Initialize data
    net_params = [[[random.uniform(0, 0.01) for x in range(58)] \
    for x in range(nodecount)], \
    [[random.uniform(0, 0.02) for x in range(nodecount)] for x in range(1)]]
    grad = [[[0.0 for x in range(58)] for x in range(nodecount)], \
    [[0.0 for x in range(58)] for x in range(1)]]
    grad_sq_sum = [[[0.0 for x in range(58)] for x in range(nodecount)], \
    [[0.0 for x in range(58)] for x in range(1)]]
    ...
```

Comparision

The neural network is quite simple: The first layer will consume the input data, then do its prediction using logistic regression. The second layer is the output layer, it uses the same logistic regression to process the output for the first layer, which is an array of predictions from the first layer.

For the input of the second layer, I shifted all the results of the output from the first layer with 0.5. This is to make the result that predicts spam has a value > 0, and vice versa.

Because it does many more processing, the running speed is significantly slower than simply using logistic regression. However, the results are better compared with plain logistic regression, which is a nice tradeoff.

I'm not really sure if this is the way how real projects that involves neural network do, as the method used in method 2 is what I assumed it shall be based on slides from the teacher.