Assessment 3

SCHOOL OF ADVANCED TECHNOLOGY

INT10: Advanced Pattern Recognition

Mengfan Li

ID number：2032048

2020 Fall

Content

1 Objectives

This assessment aims at evaluating students' ability to exploit the advanced pattern recognition knowledge, which is accumulated during lectures, and after-class study, to analyze, design, implement, develop, test and document the pattern classification methods with Neural Networks (MLP), Discriminant Function (MQDF), and Kernel Methods (Support Vector Machine) typically on the image data.

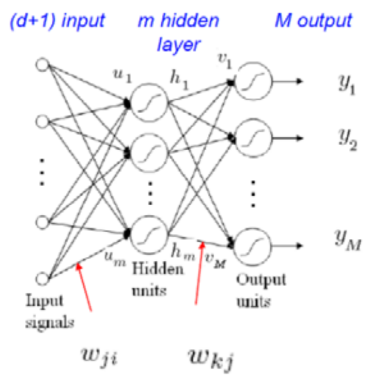
2 Materials

Pycharm

3 Methods, result and discussion

1 Implement Multilayer Perceptron (MLP) on MNIST

An artificial neural network is composed of many artificial neurons that are linked together according to a specific network architecture. The model transforms the inputs into meaningful outputs by iterating a combination of linear and nonlinear functions.



$$W = \arg\min_{W} \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{M}[y_k(x^n) - t_k^n]^2$$

$$y_k(x) = g[v_k(x)]$$

$$v_k(x) = \sum_{j=1}^{m} w_{kj}\, h_j + w_{k0}$$

$$h_j(x) = g[u_j(x)]$$

$$u_j(x) = \sum_{i=1}^{d} w_{ji}\, x_i + w_{j0}$$

Design an MLP model to achieve the classification task on MNIST. Show the training process and classification results.

Many factors will affect the performance, such as, the number of nodes in hidden layers, the selection of the activation functions, the normalization method of the samples, the loss function and its regularization term, the learning rate and the stopping policy of training, etc.

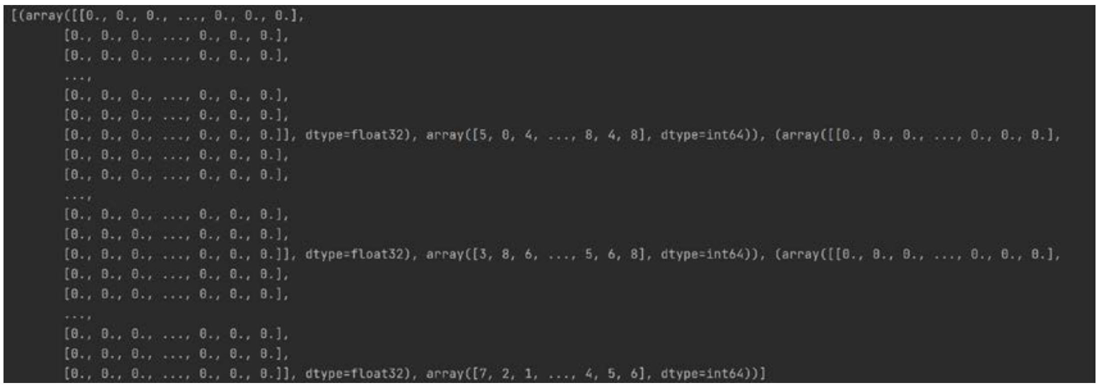
- Load data set

train_set, valid_set and test_set are training set, validation set and test set respectively, organized in the form of tuple (x, y), x represents sample, y represents sample label

```python
def load_data():
    f = gzip.open("mnist.pkl.gz", "rb")
    train_set, valid_set, test_set = pickle.load(f, encoding="latin1")
    f.close()
    return [train_set, valid_set, test_set]
```

```python
datasets = load_data()
```

Result：



```
[(array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), array([5, 0, 4, ..., 8, 4, 8], dtype=int64)), (array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), array([3, 8, 6, ..., 5, 6, 8], dtype=int64)), (array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), array([7, 2, 1, ..., 4, 5, 6], dtype=int64))]
```

- Define the loss function

For multi-class problems, the negative log-likelihood function is used as the loss function. p_y_given_x is the prediction of the model. It is an m*10 two-dimensional array. Each row represents a sample, and each column represents the model prediction sample is 0~9 Probability. Np.arange(y.shape[0]) is a one-dimensional array, containing [0,1,…,m-1], y is the true label corresponding to the sample, is a one-dimensional array, p_y_given_x[[0,1,…,m-1][y[0],…y[m-1]] is equivalent to p_y_given_x[0,y[0]], p_y_given_x[1,y[1],…,p_y_given_x[m-1,y[m-1]], that is, read the predicted value of the model corresponding to the true category of each sample, we want this value bigger.

$$C = -logy_r$$

$$y_r = \frac{e^{z_r}}{\sum_k e^{z_k}}$$

Derivation:

1、i=r

$$\frac{\partial C}{\partial z_r} = \frac{\partial C}{\partial y_r}\frac{\partial y_r}{\partial z_r} = -\frac{1}{y_r}\left(\frac{e^{z_r}}{\sum_k e^{z_k}} - e^{z_r}\frac{e^{z_r}}{(\sum_k e^{z_k})^2}\right) = -\frac{1}{y_r}(y_r - {y_r}^2) = y_r - 1$$

2、i!=r

$$\frac{\partial C}{\partial z_i} = \frac{\partial C}{\partial y_r}\frac{\partial y_r}{\partial z_i} = -\frac{1}{y_r}\left(-e^{z_r}\frac{e^{z_i}}{(\sum_k e^{z_k})^2}\right) = -\frac{1}{y_r}(-{y_r}^2) = y_r - 0$$

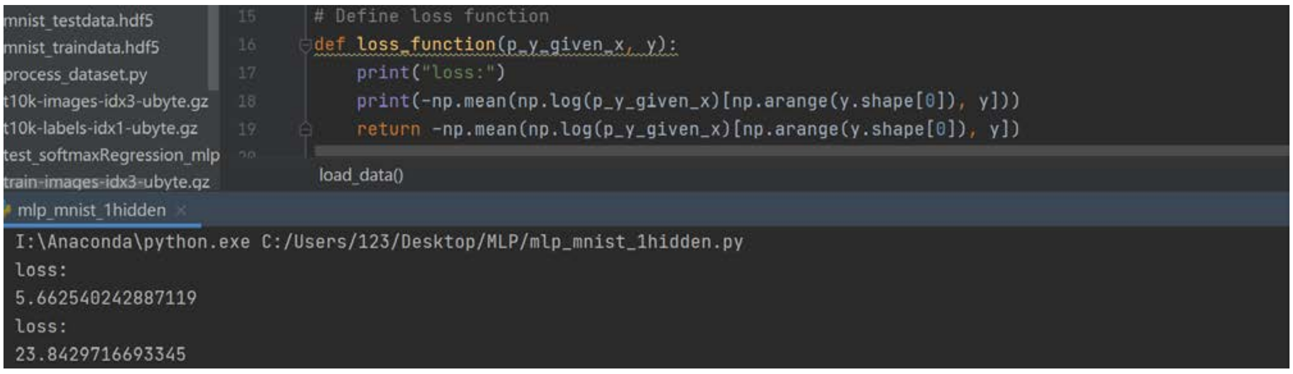
Unify, have:

$$\frac{\partial C}{\partial z_i} = y_i - \hat{y}_i$$

Code:

```
# Define loss function
def loss_function(p_y_given_x, y):
    return -np.mean(np.log(p_y_given_x)[np.arange(y.shape[0]), y])
```

Result:



```
# Define loss function
def loss_function(p_y_given_x, y):
    print("loss:")
    print(-np.mean(np.log(p_y_given_x)[np.arange(y.shape[0]), y]))
    return -np.mean(np.log(p_y_given_x)[np.arange(y.shape[0]), y])
```

```
I:\Anaconda\python.exe C:/Users/123/Desktop/MLP/mlp_mnist_1hidden.py
loss:
5.662540242887119
loss:
23.8429716693345
```

mnist_testdata.hdf5
mnist_traindata.hdf5
process_dataset.py
t10k-images-idx3-ubyte.gz
t10k-labels-idx1-ubyte.gz
test_softmaxRegression_mlp
train-images-idx3-ubyte.gz
load_data()
mlp_mnist_1hidden

- Count the number of error samples

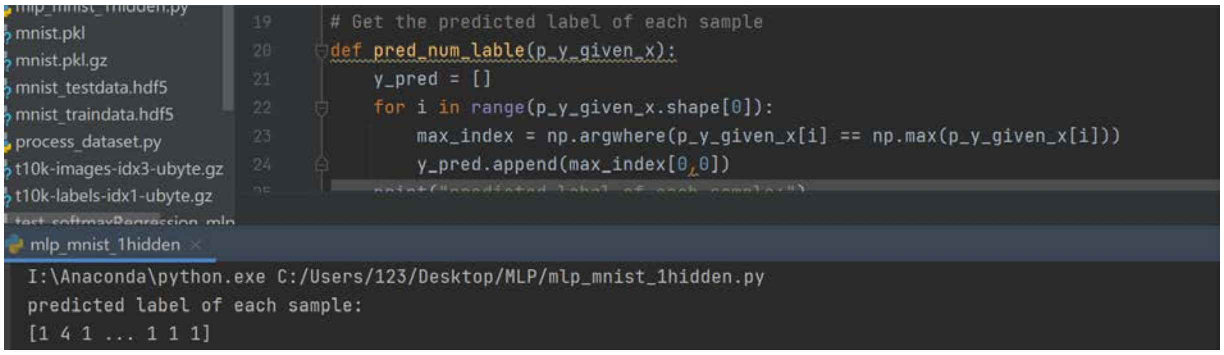
First, according to the predicted probability of the network, choose the one with the highest probability as the network predicted label

Code:

```
# Get the predicted label of each sample
```

```python
def pred_num_lable(p_y_given_x):
    y_pred = []
    for i in range(p_y_given_x.shape[0]):
        max_index = np.argwhere(p_y_given_x[i] == np.max(p_y_given_x[i]))
        y_pred.append(max_index[0,0])
    # print(y_pred[:50])
    return np.array(y_pred)
```

Result:



```python
# Get the predicted label of each sample
def pred_num_lable(p_y_given_x):
    y_pred = []
    for i in range(p_y_given_x.shape[0]):
        max_index = np.argwhere(p_y_given_x[i] == np.max(p_y_given_x[i]))
        y_pred.append(max_index[0,0])
```

```
I:\Anaconda\python.exe C:/Users/123/Desktop/MLP/mlp_mnist_1hidden.py
predicted label of each sample:
[1 4 1 ... 1 1 1]
```

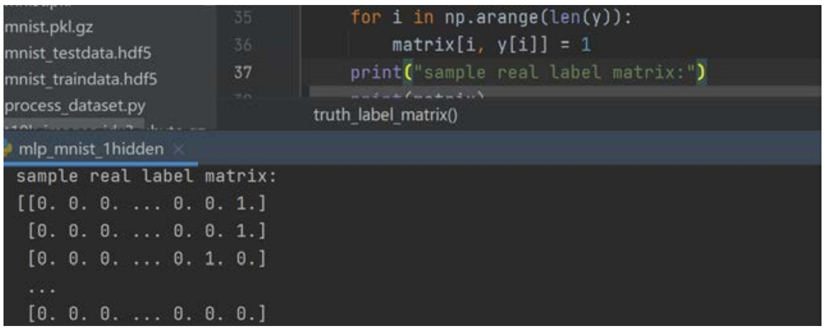
- Construct the sample real label matrix

In the original data set, the label of each sample corresponds to a scalar, which directly indicates which number the sample is. But in calculating the gradient, y needs to be expanded into a vector with the same dimension as the model output.

Code:

```python
# Construct the sample real label matrix
def truth_label_matrix(y):
    matrix = np.zeros((len(y), 10))
    for i in np.arange(len(y)):
        matrix[i, y[i]] = 1
    return matrix
```

Result:



```python
    for i in np.arange(len(y)):
        matrix[i, y[i]] = 1
    print("sample real label matrix:")
```

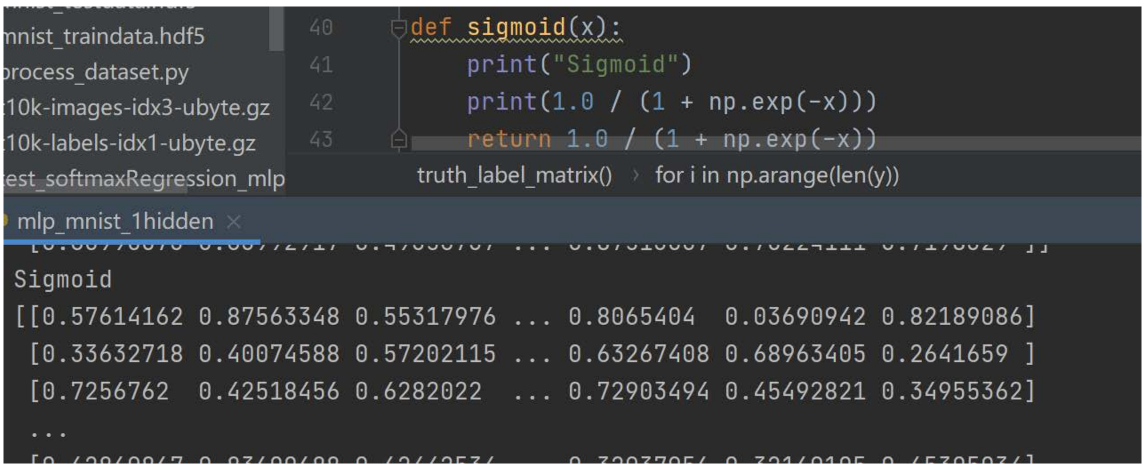
```
sample real label matrix:
[[0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 0. 1. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
```

- sigmoid function and its derivative

Code:

```
# Sigmoid
def sigmoid(x):
    return 1.0 / (1 + np.exp(-x))
# Sigmoid derivative
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

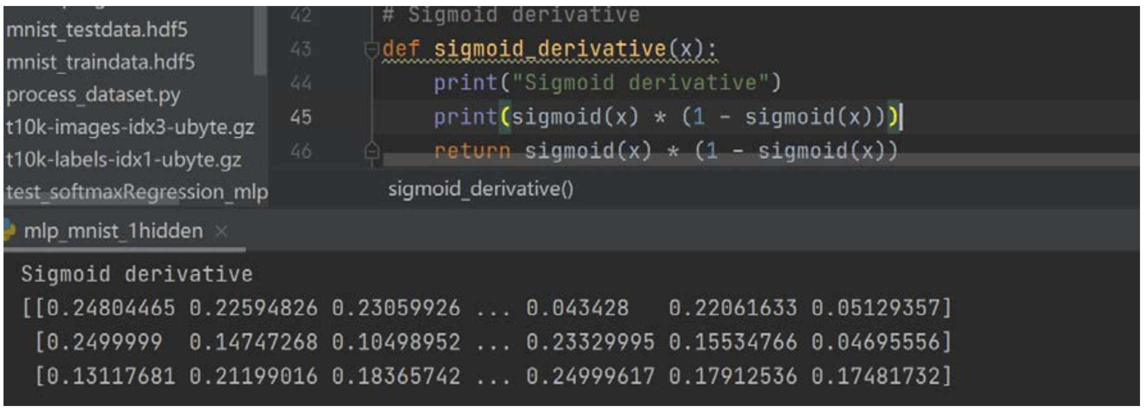
Resule:



mnist_traindata.hdf5
process_dataset.py
t10k-images-idx3-ubyte.gz
t10k-labels-idx1-ubyte.gz
test_softmaxRegression_mlp
truth_label_matrix()
for i in np.arange(len(y))
mlp_mnist_1hidden

```
def sigmoid(x):
    print("Sigmoid")
    print(1.0 / (1 + np.exp(-x)))
    return 1.0 / (1 + np.exp(-x))
```

```
Sigmoid
[[0.57614162 0.87563348 0.55317976 ... 0.8065404  0.03690942 0.82189086]
 [0.33632718 0.40074588 0.57202115 ... 0.63267408 0.68963405 0.2641659 ]
 [0.7256762  0.42518456 0.6282022  ... 0.72903494 0.45492821 0.34955362]
 ...
```



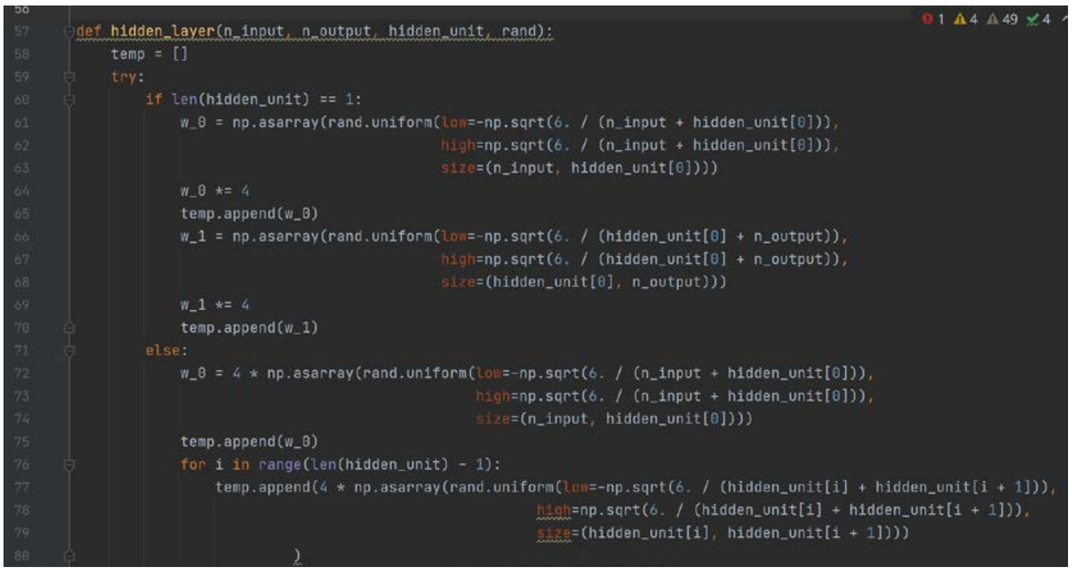
```
# Sigmoid derivative
def sigmoid_derivative(x):
    print("Sigmoid derivative")
    print(sigmoid(x) * (1 - sigmoid(x)))
    return sigmoid(x) * (1 - sigmoid(x))
```

```
Sigmoid derivative
[[0.24804465 0.22594826 0.23059926 ... 0.043428   0.22061633 0.05129357]
 [0.2499999  0.14747268 0.10498952 ... 0.23329995 0.15534766 0.04695556]
 [0.13117681 0.21199016 0.18365742 ... 0.24999617 0.17912536 0.17481732]
```

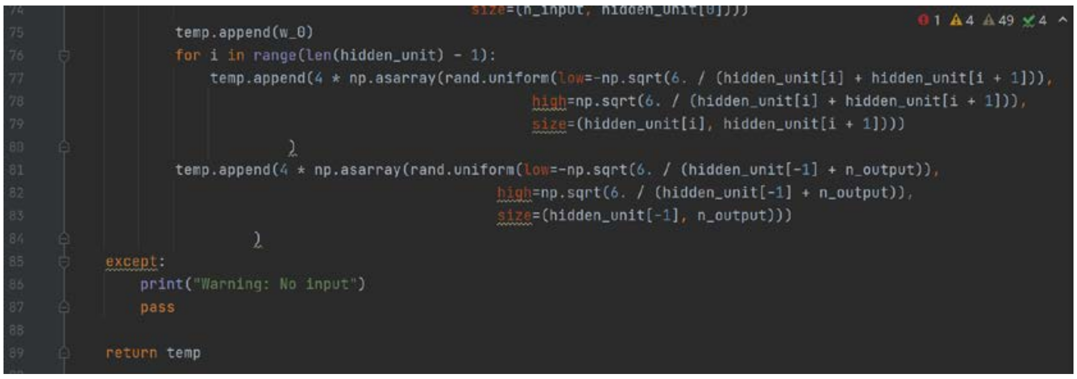
- Initialize weight w

Determine the number of hidden layers, the number of hidden layer neurons, and initialize the connection weight, the neural network model is determined

Code:

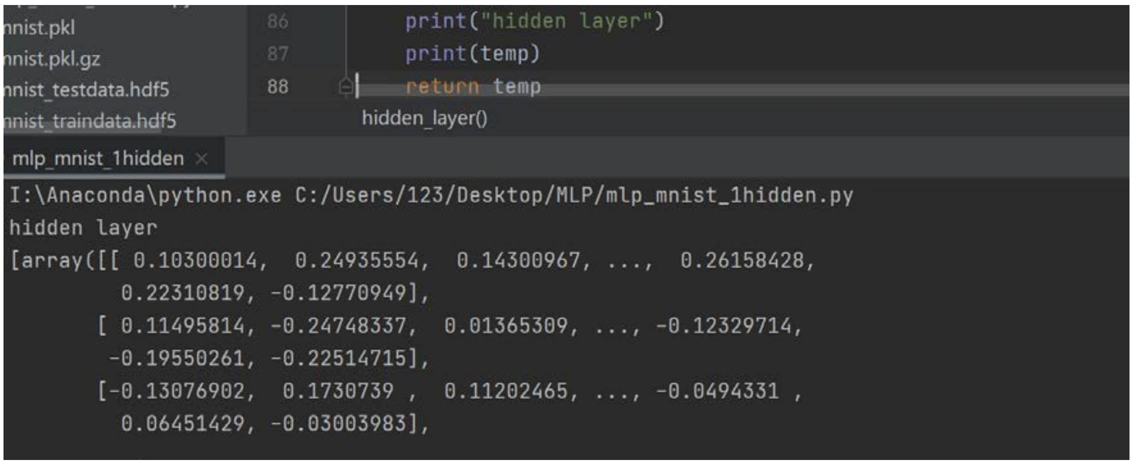


```python
def hidden_layer(n_input, n_output, hidden_unit, rand):
    temp = []
    try:
        if len(hidden_unit) == 1:
            w_0 = np.asarray(rand.uniform(low=-np.sqrt(6. / (n_input + hidden_unit[0])),
                                          high=np.sqrt(6. / (n_input + hidden_unit[0])),
                                          size=(n_input, hidden_unit[0])))
            w_0 *= 4
            temp.append(w_0)
            w_1 = np.asarray(rand.uniform(low=-np.sqrt(6. / (hidden_unit[0] + n_output)),
                                          high=np.sqrt(6. / (hidden_unit[0] + n_output)),
                                          size=(hidden_unit[0], n_output)))
            w_1 *= 4
            temp.append(w_1)
        else:
            w_0 = 4 * np.asarray(rand.uniform(low=-np.sqrt(6. / (n_input + hidden_unit[0])),
                                              high=np.sqrt(6. / (n_input + hidden_unit[0])),
                                              size=(n_input, hidden_unit[0])))
            temp.append(w_0)
            for i in range(len(hidden_unit) - 1):
                temp.append(4 * np.asarray(rand.uniform(low=-np.sqrt(6. / (hidden_unit[i] + hidden_unit[i + 1])),
                                                        high=np.sqrt(6. / (hidden_unit[i] + hidden_unit[i + 1])),
                                                        size=(hidden_unit[i], hidden_unit[i + 1])))
                            )
```



```python
                                              size=(n_input, hidden_unit[0])))
            temp.append(w_0)
            for i in range(len(hidden_unit) - 1):
                temp.append(4 * np.asarray(rand.uniform(low=-np.sqrt(6. / (hidden_unit[i] + hidden_unit[i + 1])),
                                                        high=np.sqrt(6. / (hidden_unit[i] + hidden_unit[i + 1])),
                                                        size=(hidden_unit[i], hidden_unit[i + 1])))
                            )
            temp.append(4 * np.asarray(rand.uniform(low=-np.sqrt(6. / (hidden_unit[-1] + n_output)),
                                                    high=np.sqrt(6. / (hidden_unit[-1] + n_output)),
                                                    size=(hidden_unit[-1], n_output)))
                        )
    except:
        print("Warning: No input")
        pass

    return temp
```

Result:



```
nnist.pkl
nnist.pkl.gz
nnist_testdata.hdf5
nnist_traindata.hdf5
```

```python
    print("hidden layer")
    print(temp)
    return temp
hidden_layer()
```

mlp_mnist_1hidden

```
I:\Anaconda\python.exe C:/Users/123/Desktop/MLP/mlp_mnist_1hidden.py
hidden layer
[array([[ 0.10300014,  0.24935554,  0.14300967, ...,  0.26158428,
         0.22310819, -0.12770949],
       [ 0.11495814, -0.24748337,  0.01365309, ..., -0.12329714,
        -0.19550261, -0.22514715],
       [-0.13076902,  0.1730739 ,  0.11202465, ..., -0.0494331 ,
         0.06451429, -0.03003983],
```

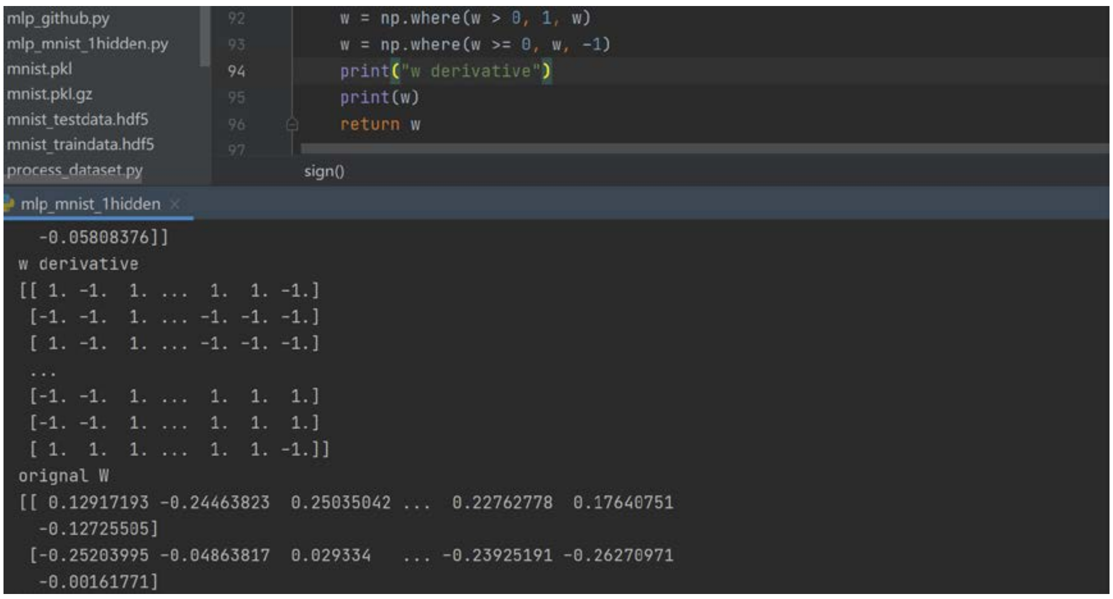
- Calculate the derivative of |*w*|

Calculate the derivative of |w|, needed when using L1 regularization

Code:

```python
# Calculate the derivative of |w|, needed when using L1 regularization
def sign(w):
    w = np.where(w > 0, 1, w)
    w = np.where(w >= 0, w, -1)
    return w
```

Result:



```python
    w = np.where(w > 0, 1, w)
    w = np.where(w >= 0, w, -1)
    print("w derivative")
    print(w)
    return w
```

```
  -0.05808376]]
w derivative
[[ 1. -1.  1. ...  1.  1. -1.]
 [-1. -1.  1. ... -1. -1. -1.]
 [ 1. -1.  1. ... -1. -1. -1.]
 ...
 [-1. -1.  1. ...  1.  1.  1.]
 [-1. -1.  1. ...  1.  1.  1.]
 [ 1.  1.  1. ...  1.  1. -1.]]
orignal W
[[ 0.12917193 -0.24463823  0.25035042 ...  0.22762778  0.17640751
  -0.12725505]
 [-0.25203995 -0.04863817  0.029334   ... -0.23925191 -0.26270971
  -0.00161771]
```

- Forward spread

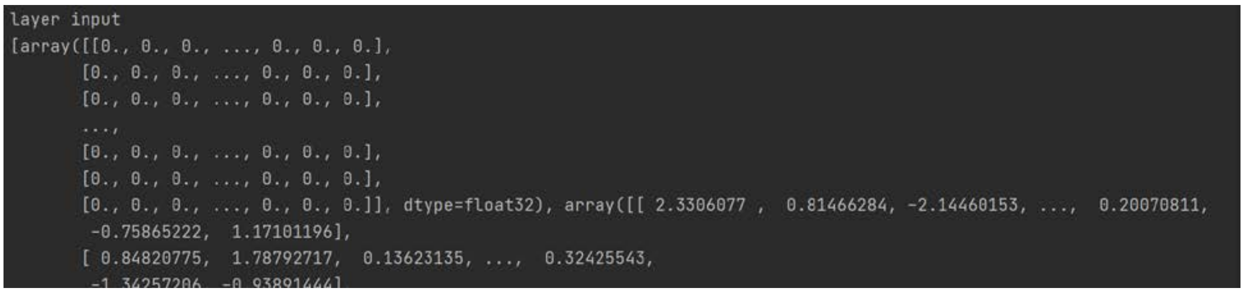
Forward propagation, get model prediction value

Code:

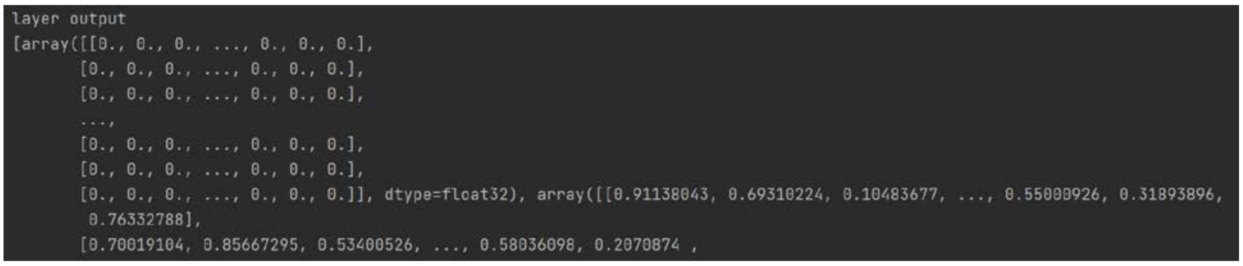
```python
# Forward spread
def forward(x, W):
    layer_input = [x]                  # input
    layer_output = [x]        # output

    # hidden layer
    for i in range(len(W) - 1):
        layer_input.append(np.dot(x, W[i]))
        temp_ = sigmoid(np.dot(x, W[i]))
        layer_output.append(temp_)
    # output layer
    layer_input.append(np.dot(temp_, W[-1]))
```

```
        layer_output.append((np.transpose(np.exp(np.dot(temp_, W[-1])) / np.sum(np.exp(np.dot(temp_, W[-1])), axis=1)).T)

        return (layer_output, layer_input)
```

Resulr:



```
layer input
[array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), array([[ 2.3306077 ,  0.81466284, -2.14460153, ...,  0.20070811,
        -0.75865222,  1.17101196],
       [ 0.84820775,  1.78792717,  0.13623135, ...,  0.32425543,
        -1.34257206, -0.93891444]
```



```
layer output
[array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), array([[0.91138043, 0.69310224, 0.10483677, ..., 0.55000926, 0.31893896,
        0.76332788],
       [0.70019104, 0.85667295, 0.53400526, ..., 0.58036098, 0.2070874 ,
```

- Back propagation

Using the backpropagation algorithm, calculate the gradient of each layer connected W, because:

$$\frac{\partial C}{\partial w_{ij}^{l}} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{ij}^l}$$

The key is to find $\frac{\partial C}{\partial z_j^l}$ and make it equal to δ_j^l

- Output layer

$$\frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial y_j^L}\frac{\partial y_j^L}{\partial z_j^L} = \nabla C(y^L) * \delta'(z_j^L)$$

- Un-output layer

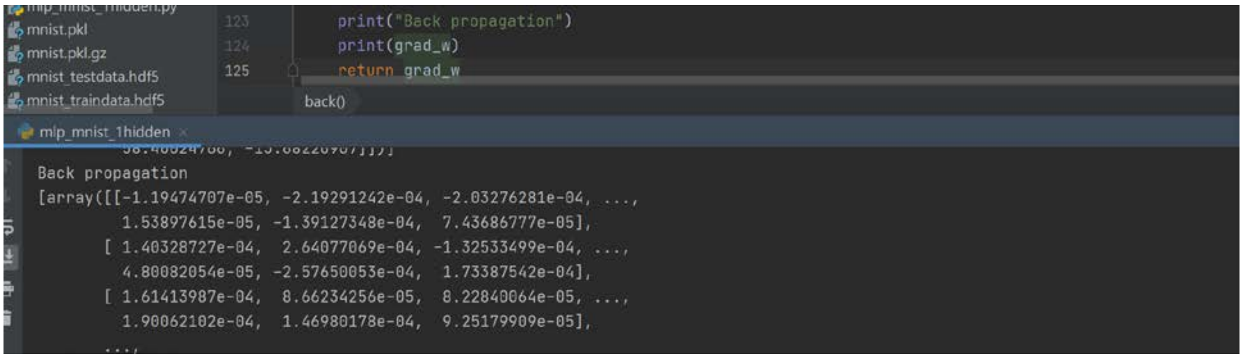
$$\frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \sum_k \delta_k^{l+1}\omega_{jk}^{l+1}\,\delta'(z_j^L)$$

Code:

```
# Back propagation
def back(layer_output, layer_input, w, y, L1_reg, L2_reg):
    grad_w = [np.zeros(weight.shape) for weight in w]
```

```python

        # Backpropagation
        delta = layer_output[-1] - y  # last layer, m*10
        grad_w[-1] = np.dot(np.transpose(layer_output[-2]), delta)
        for l in range(2, len(w) + 1):
            delta = np.dot(delta, w[-l + 1].transpose()) * sigmoid_derivative(layer_input[-l])  # other layer
            grad_w[-l] = np.dot(np.transpose(layer_output[-l - 1]), delta) / y.shape[0] + L1_reg * sign(w[-l]) + L2_reg * w[-l]

        return grad_w
```

Result:



```python
        print("Back propagation")
        print(grad_w)
        return grad_w
```

```
Back propagation
[array([[-1.19474707e-05, -2.19291242e-04, -2.03276281e-04, ...,
          1.53897615e-05, -1.39127348e-04,  7.43686777e-05],
        [ 1.40328727e-04,  2.64077069e-04, -1.32533499e-04, ...,
          4.80082054e-05, -2.57650053e-04,  1.73387542e-04],
        [ 1.61413987e-04,  8.66234256e-05,  8.22840064e-05, ...,
          1.90062102e-04,  1.46980178e-04,  9.25179909e-05],
        ...,
```

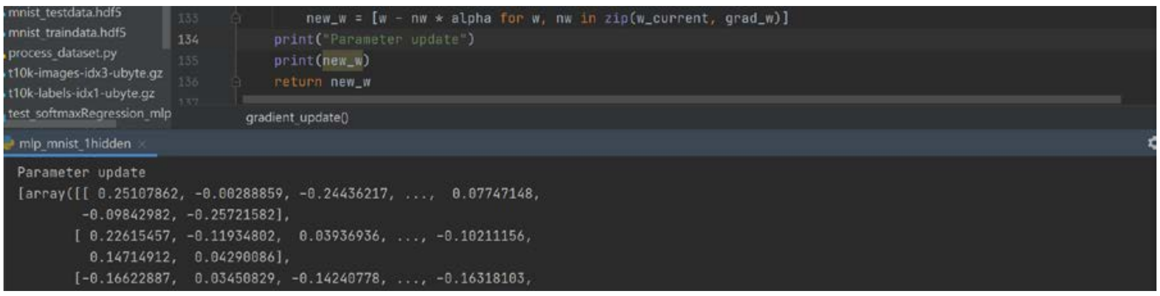
After calculating the gradient, update *w* in the opposite direction of the gradient

$$\omega^{n+1} = \omega^{n} - \alpha \nabla C(\omega^{n})$$

Code:

```python
# Parameter update
def gradient_update(y_truth_matrix, w_current, layer_output, layer_input, alpha, L1_reg, L2_reg):
    if len(w_current) > 1:
        # Calculate the gradient using backpropagation
        grad_w = back(layer_output, layer_input, w_current, y_truth_matrix, L1_reg, L2_reg)
        # Parameter update
        new_w = [w - nw * alpha for w, nw in zip(w_current, grad_w)]

    return new_w
```

Result:



```
        new_w = [w - nw * alpha for w, nw in zip(w_current, grad_w)]
    print("Parameter update")
    print(new_w)
    return new_w
```

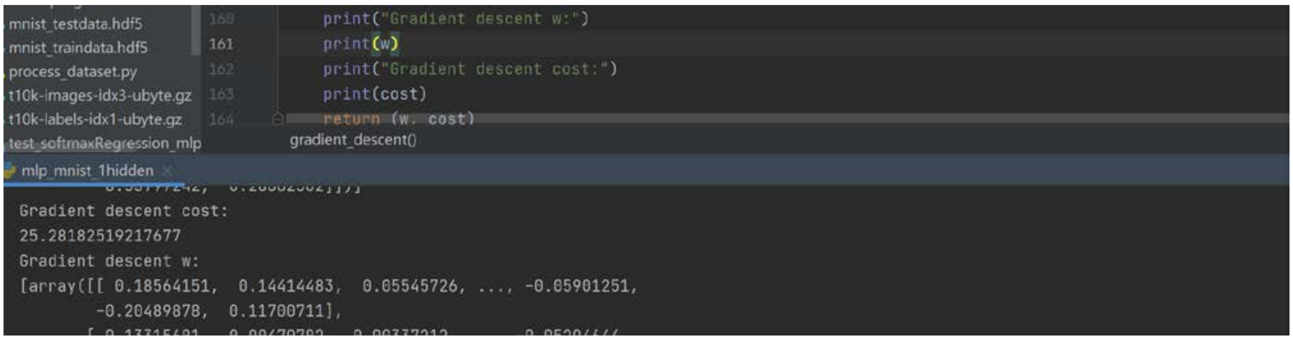
```
Parameter update
[array([[ 0.25107862, -0.00288859, -0.24436217, ...,  0.07747148,
         -0.09842982, -0.25721582],
       [ 0.22615457, -0.11934802,  0.03936936, ..., -0.10211156,
         0.14714912,  0.04290086],
       [-0.16622887,  0.03450829, -0.14240778, ..., -0.16318103,
```

- Gradient descent

Code:

```
# Gradient descent
def gradient_descent(x, y, w_initial, alpha, L1_reg, L2_reg):
    w = w_initial

    y_truth_matrix = truth_label_matrix(y)

    # Forward propagation to get model prediction value
    layer_output, layer_input = forward(x, w)
    probabilities = layer_output[-1]
    if L1_reg != 0.00:
        L1 = sum([abs(w_i).sum() for w_i in w])  # L1 norm
    else:
        L1 = 0
    if L2_reg != 0.00:
        L2_sqr = sum([(w_i ** 2).sum() for w_i in w])  # L2 norm
    else:
        L2_sqr = 0

    # Total loss = negative log likelihood cost + regularization penalty
    cost = loss_function(probabilities, y) + L1_reg * L1 + 0.5 * L2_reg * L2_sqr  # Error before weight w update

    # Update parameters
    w = gradient_update(y_truth_matrix, w, layer_output, layer_input, alpha, L1_reg, L2_reg)

    return (w, cost)
```

Result:



mnist_testdata.hdf5
mnist_traindata.hdf5
process_dataset.py
t10k-images-idx3-ubyte.gz
t10k-labels-idx1-ubyte.gz
test_softmaxRegression_mlp

```
    print("Gradient descent w:")
    print(w)
    print("Gradient descent cost:")
    print(cost)
    return (w, cost)
```

gradient_descent()

mlp_mnist_1hidden

```
Gradient descent cost:
25.28182519217677
Gradient descent w:
[array([[ 0.18564151,  0.14414483,  0.05545726, ..., -0.05901251,
        -0.20489878,  0.11700711],
```

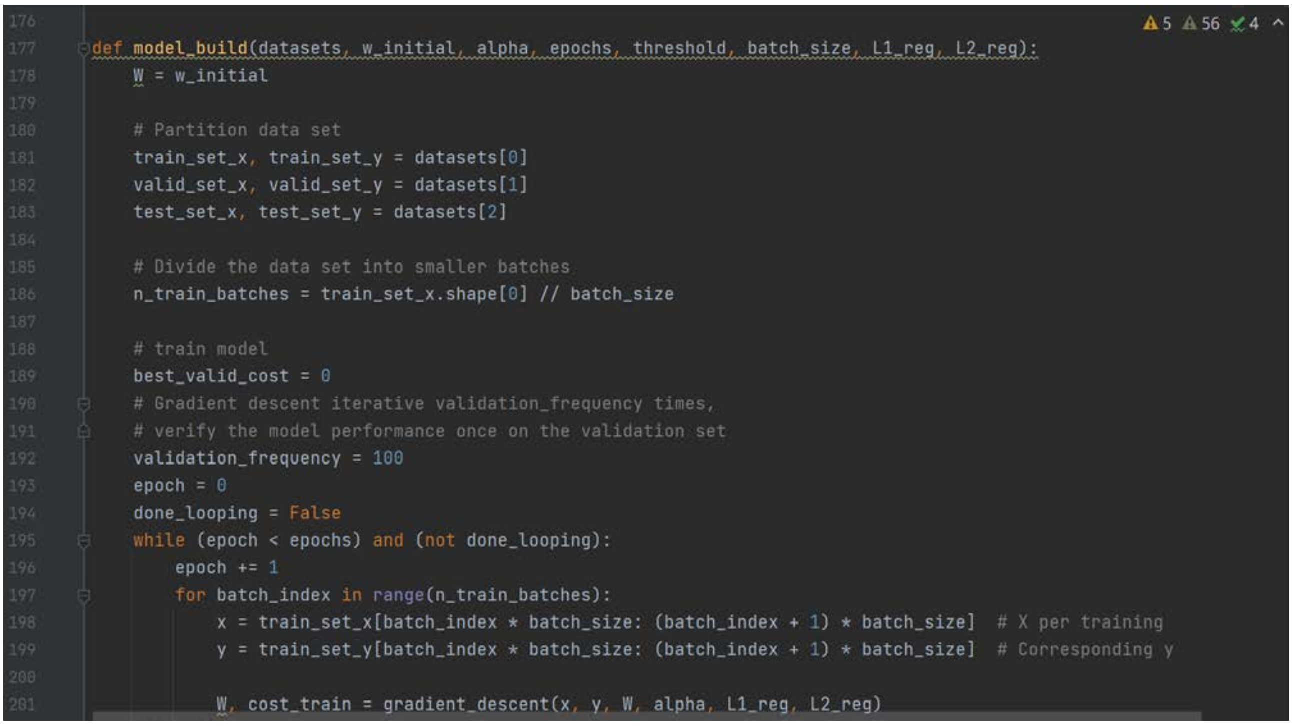
- Validation and testing model

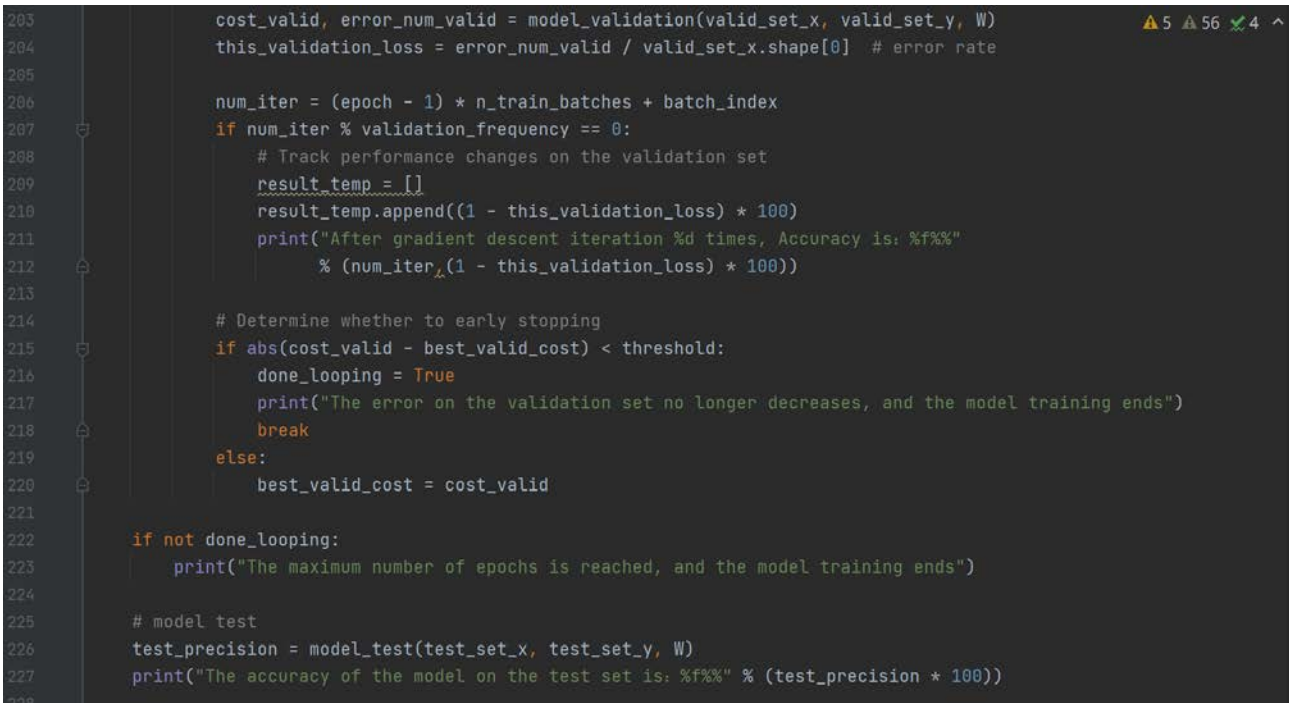
Code:

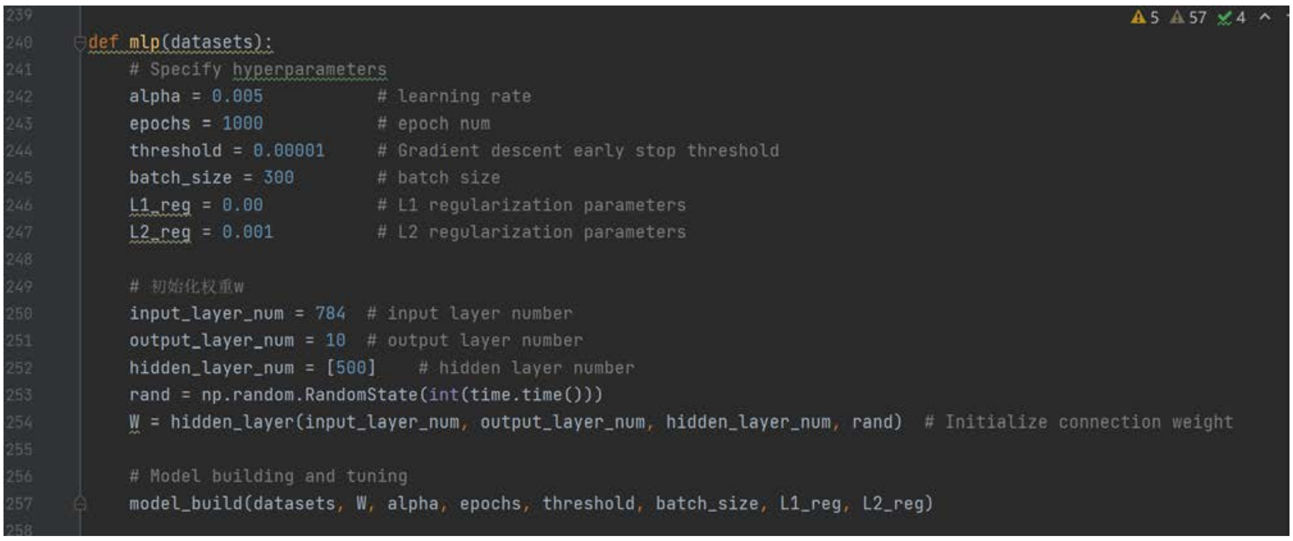
```python
def model_validation(valid_set_x, valid_set_y, W):
    p_y_given_valid_x = forward(valid_set_x, W)[0][-1]
    valid_y_pred = pred_num_lable(p_y_given_valid_x)
    cost_valid = loss_function(p_y_given_valid_x, valid_set_y)
    error_num_valid = error_num(valid_y_pred, valid_set_y)
    return (cost_valid, error_num_valid)

def model_test(test_set_x, test_set_y, W):

    p_y_given_test_x = forward(test_set_x, W)[0][-1]
    test_y_pred = pred_num_lable(p_y_given_test_x)
    error_num_test = error_num(test_y_pred, test_set_y)
    test_precision = 1 - error_num_test / test_set_x.shape[0]
    return test_precision
```

- Model construction and optimization

Code:


```python
def model_build(datasets, w_initial, alpha, epochs, threshold, batch_size, L1_reg, L2_reg):
    W = w_initial

    # Partition data set
    train_set_x, train_set_y = datasets[0]
    valid_set_x, valid_set_y = datasets[1]
    test_set_x, test_set_y = datasets[2]

    # Divide the data set into smaller batches
    n_train_batches = train_set_x.shape[0] // batch_size

    # train model
    best_valid_cost = 0
    # Gradient descent iterative validation_frequency times,
    # verify the model performance once on the validation set
    validation_frequency = 100
    epoch = 0
    done_looping = False
    while (epoch < epochs) and (not done_looping):
        epoch += 1
        for batch_index in range(n_train_batches):
            x = train_set_x[batch_index * batch_size: (batch_index + 1) * batch_size]  # X per training
            y = train_set_y[batch_index * batch_size: (batch_index + 1) * batch_size]  # Corresponding y

            W, cost_train = gradient_descent(x, y, W, alpha, L1_reg, L2_reg)
```

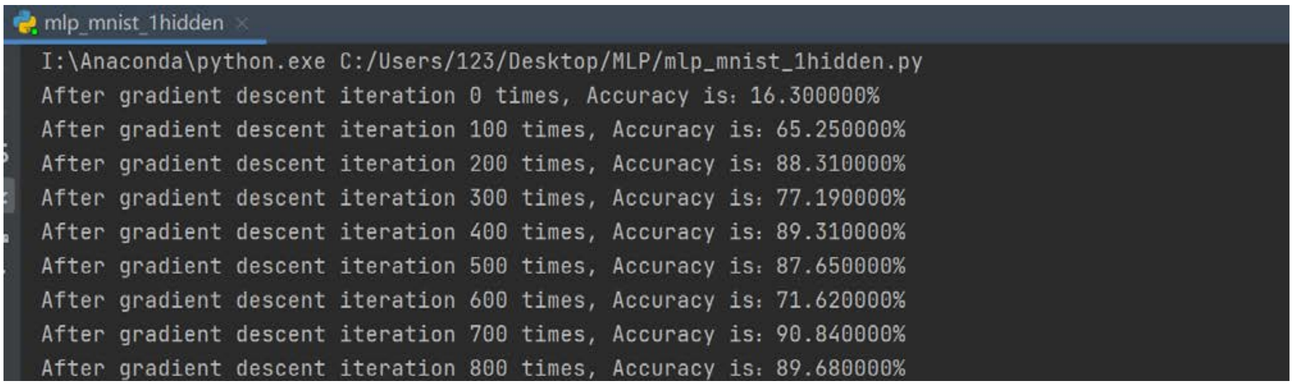

```python
            cost_valid, error_num_valid = model_validation(valid_set_x, valid_set_y, W)
            this_validation_loss = error_num_valid / valid_set_x.shape[0]  # error rate

            num_iter = (epoch - 1) * n_train_batches + batch_index
            if num_iter % validation_frequency == 0:
                # Track performance changes on the validation set
                result_temp = []
                result_temp.append((1 - this_validation_loss) * 100)
                print("After gradient descent iteration %d times, Accuracy is: %f%%"
                      % (num_iter, (1 - this_validation_loss) * 100))

            # Determine whether to early stopping
            if abs(cost_valid - best_valid_cost) < threshold:
                done_looping = True
                print("The error on the validation set no longer decreases, and the model training ends")
                break
            else:
                best_valid_cost = cost_valid

    if not done_looping:
        print("The maximum number of epochs is reached, and the model training ends")

    # model test
    test_precision = model_test(test_set_x, test_set_y, W)
    print("The accuracy of the model on the test set is: %f%%" % (test_precision * 100))
```



```
def mlp(datasets):
    # Specify hyperparameters
    alpha = 0.005           # learning rate
    epochs = 1000           # epoch num
    threshold = 0.00001     # Gradient descent early stop threshold
    batch_size = 300        # batch size
    L1_reg = 0.00           # L1 regularization parameters
    L2_reg = 0.001          # L2 regularization parameters

    # 初始化权重w
    input_layer_num = 784  # input layer number
    output_layer_num = 10  # output layer number
    hidden_layer_num = [500]    # hidden layer number
    rand = np.random.RandomState(int(time.time()))
    W = hidden_layer(input_layer_num, output_layer_num, hidden_layer_num, rand)  # Initialize connection weight

    # Model building and tuning
    model_build(datasets, W, alpha, epochs, threshold, batch_size, L1_reg, L2_reg)
```

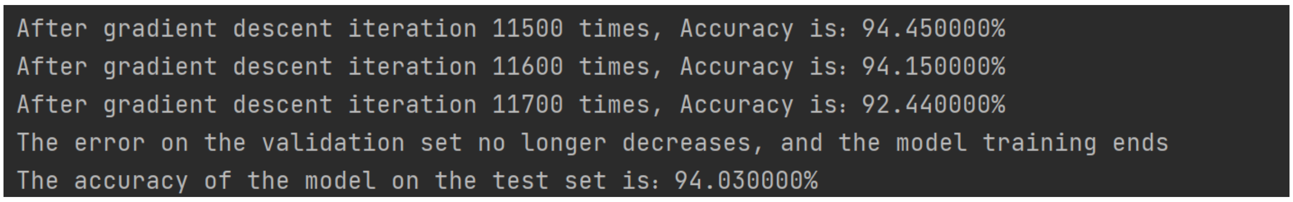
Result:

Train model



```
I:\Anaconda\python.exe C:/Users/123/Desktop/MLP/mlp_mnist_1hidden.py
After gradient descent iteration 0 times, Accuracy is: 16.300000%
After gradient descent iteration 100 times, Accuracy is: 65.250000%
After gradient descent iteration 200 times, Accuracy is: 88.310000%
After gradient descent iteration 300 times, Accuracy is: 77.190000%
After gradient descent iteration 400 times, Accuracy is: 89.310000%
After gradient descent iteration 500 times, Accuracy is: 87.650000%
After gradient descent iteration 600 times, Accuracy is: 71.620000%
After gradient descent iteration 700 times, Accuracy is: 90.840000%
After gradient descent iteration 800 times, Accuracy is: 89.680000%
```

- Final result



```
After gradient descent iteration 11500 times, Accuracy is: 94.450000%
After gradient descent iteration 11600 times, Accuracy is: 94.150000%
After gradient descent iteration 11700 times, Accuracy is: 92.440000%
The error on the validation set no longer decreases, and the model training ends
The accuracy of the model on the test set is: 94.030000%
```

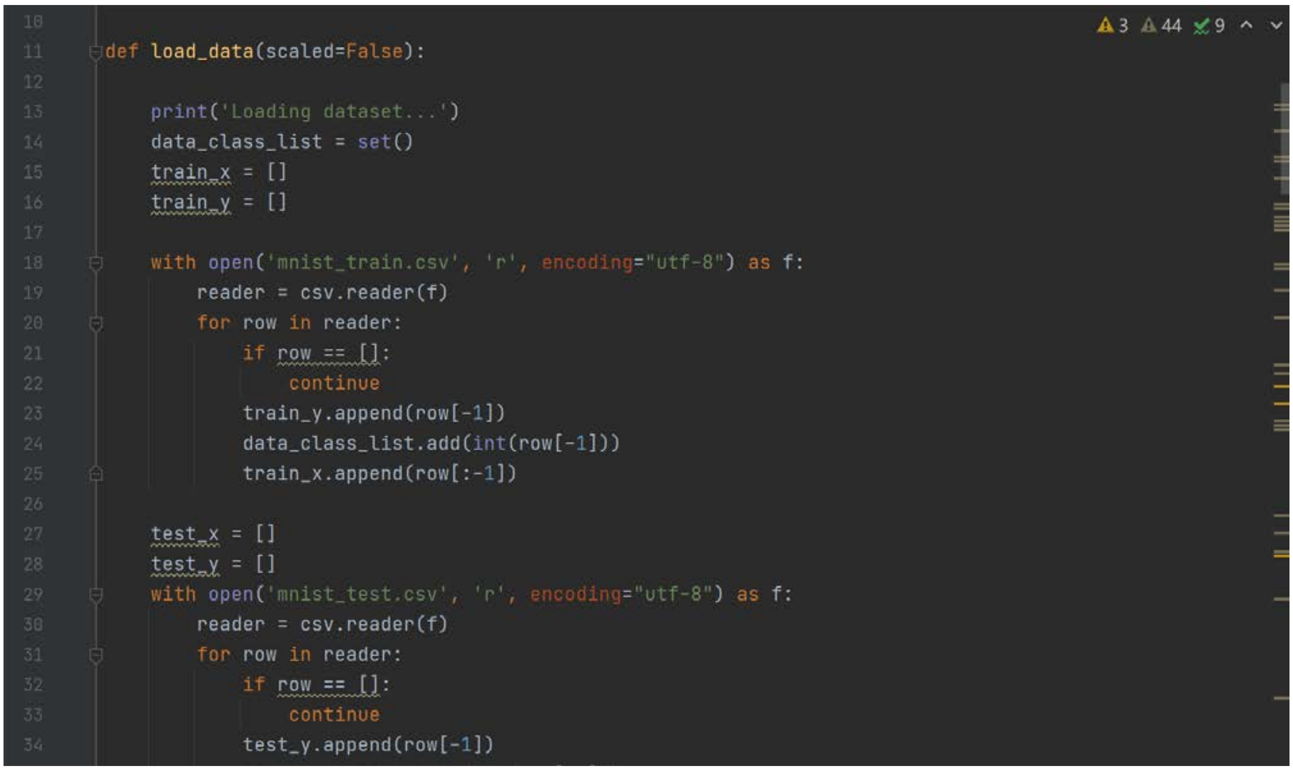
2 Implement Modified Quadratic Discriminant Function (MQDF) on MNIST

The MQDF model is a classifier based on Bayesian decision theory. Its discriminant function is given as below:

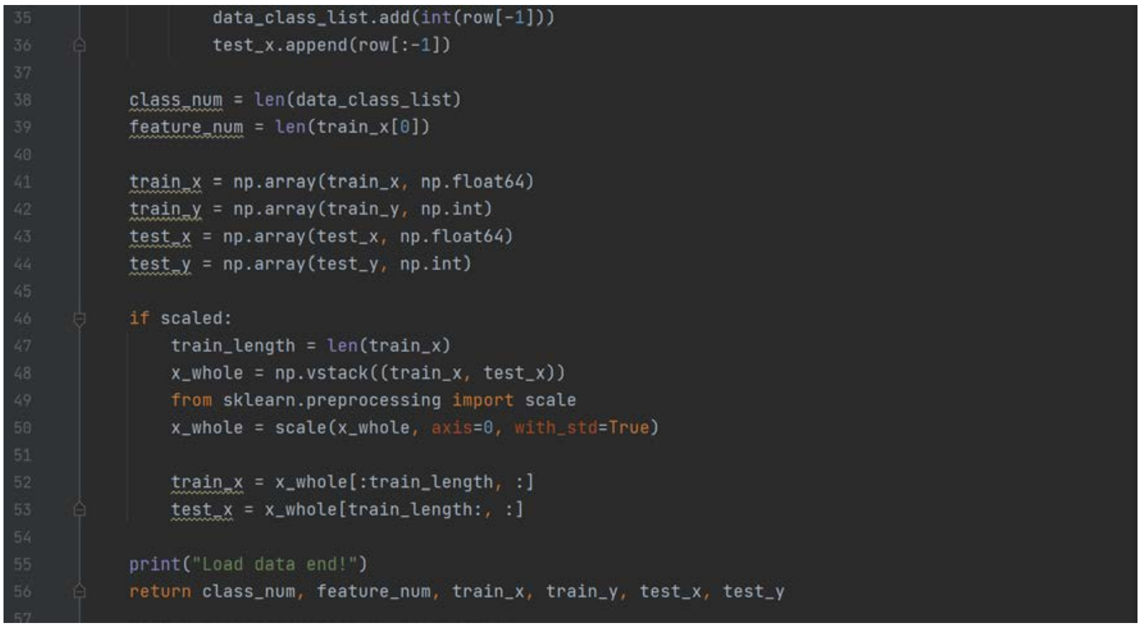
$$\begin{aligned} g_0(\mathbf{x},\omega_i) &= -(\mathbf{x}-\mu_i)^T\Sigma_i^{-1}(\mathbf{x}-\mu_i)-\log|\Sigma_i| \\ &= -[\Phi_i^T(\mathbf{x}-\mu_i)]^T\Lambda_i^{-1}\Phi_i^T(\mathbf{x}-\mu_i)-\log|\Sigma_i| \\ &= -\sum_{j=1}^{d}\frac{1}{\lambda_{ij}}[(\mathbf{x}-\mu_i)^T\phi_{ij}]^2-\sum_{j=1}^{d}\log\lambda_{ij} \end{aligned} \qquad \Sigma_i=\Phi_i\Lambda_i\Phi_i^T$$

Design and implement a MQDF model to achieve the classification task on MNIST. Show the training process and classification results. The work in [1] can be used as reference for your work.

- Load data

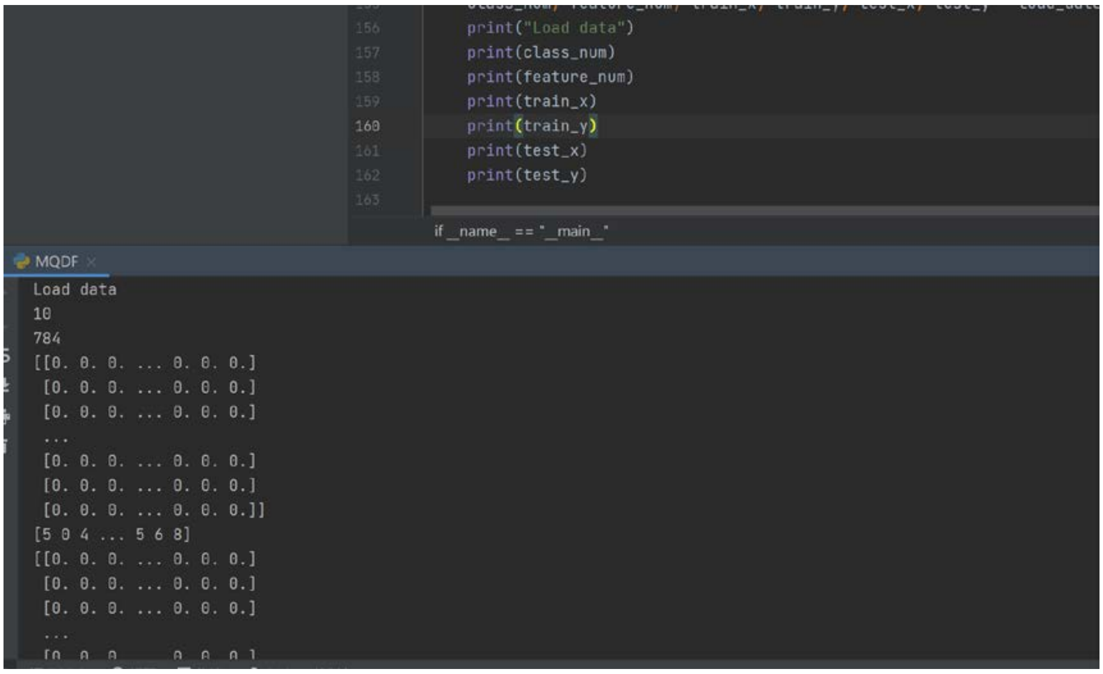


```python
def load_data(scaled=False):

    print('Loading dataset...')
    data_class_list = set()
    train_x = []
    train_y = []

    with open('mnist_train.csv', 'r', encoding="utf-8") as f:
        reader = csv.reader(f)
        for row in reader:
            if row == []:
                continue
            train_y.append(row[-1])
            data_class_list.add(int(row[-1]))
            train_x.append(row[:-1])

    test_x = []
    test_y = []
    with open('mnist_test.csv', 'r', encoding="utf-8") as f:
        reader = csv.reader(f)
        for row in reader:
            if row == []:
                continue
            test_y.append(row[-1])
```



```python
            data_class_list.add(int(row[-1]))
            test_x.append(row[:-1])

    class_num = len(data_class_list)
    feature_num = len(train_x[0])

    train_x = np.array(train_x, np.float64)
    train_y = np.array(train_y, np.int)
    test_x = np.array(test_x, np.float64)
    test_y = np.array(test_y, np.int)

    if scaled:
        train_length = len(train_x)
        x_whole = np.vstack((train_x, test_x))
        from sklearn.preprocessing import scale
        x_whole = scale(x_whole, axis=0, with_std=True)

        train_x = x_whole[:train_length, :]
        test_x = x_whole[train_length:, :]

    print("Load data end!")
    return class_num, feature_num, train_x, train_y, test_x, test_y
```

Result:



```
    print("Load data")
    print(class_num)
    print(feature_num)
    print(train_x)
    print(train_y)
    print(test_x)
    print(test_y)

if __name__ == "__main__"
```

MQDF

```
Load data
10
784
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
[5 0 4 ... 5 6 8]
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
```

- K-L decomposition

Bayes decision theory is to judge the input sample into the category with the largest posterior probability (Maximum A Posterior, MAP). The quadratic discriminant function (QDF) is the specific manifestation of Bayes decision when the input data meets the Gaussian distribution.

According to Bayes' criterion, the posterior probability of the input sample x belonging to the category is:

$$P(\omega_i|x) = \frac{P(\omega_i)p(x|\omega_i)}{p(x)} \qquad 1-1$$

The Bayes criterion can become:

$$x \in \omega_k = agr \min g_{qdf}(x, \omega_i) \qquad 1-2$$

among them:

$$g_{qdf}(x, \omega_i) = (x-\mu_i)^t \textstyle\sum_i^{-1}(x-\mu_i) + \ln|\Sigma_i| \qquad 1-3$$

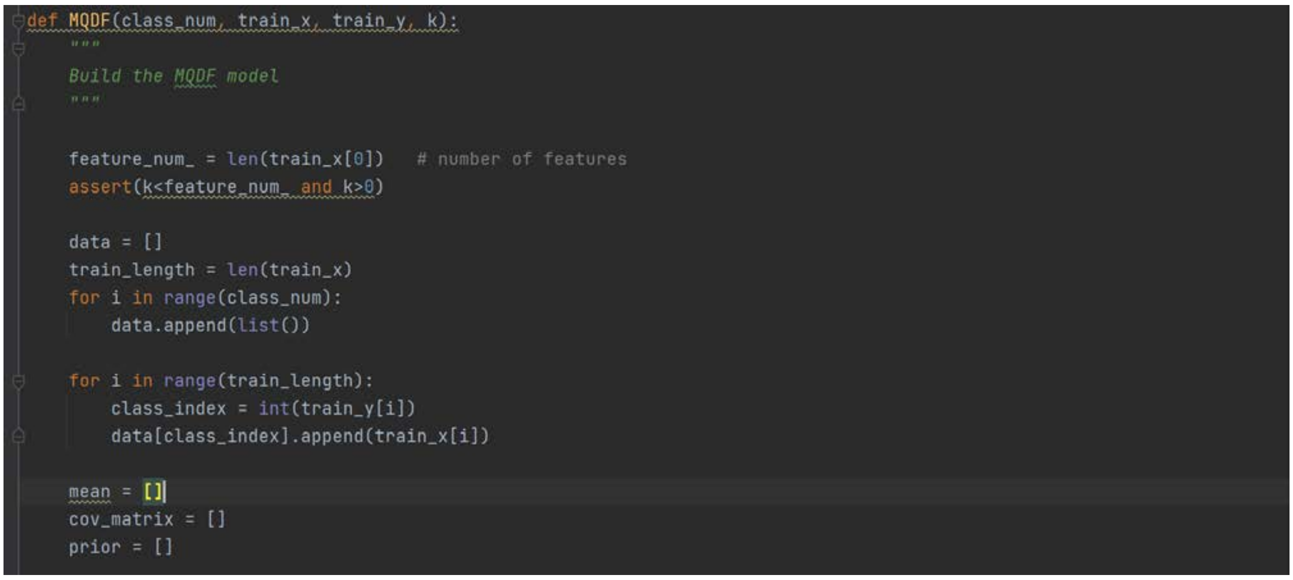
Using K-L decomposition, the covariance matrix Σ_i can be decomposed into the following form:

$$\Sigma_i = P_i \wedge_i P_i^t \qquad 1-4$$

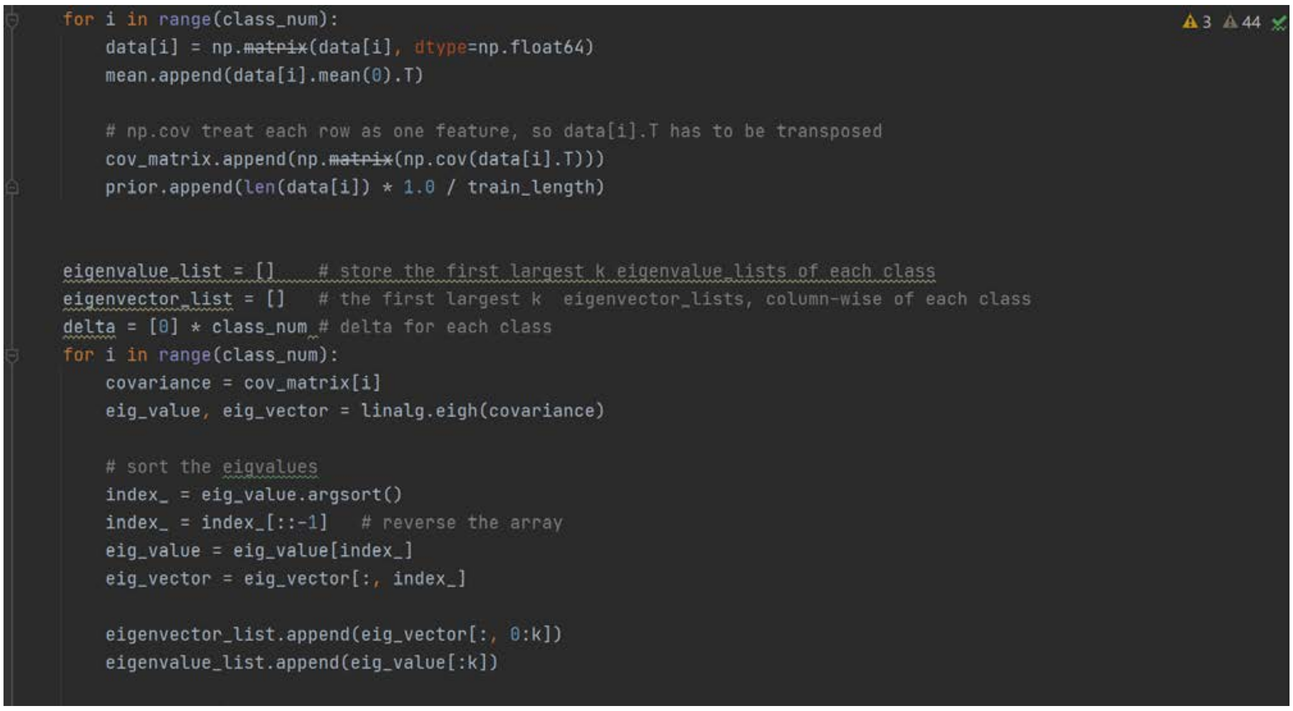
Among them $\wedge_i = diag(\lambda_{i,1}, \lambda_{i,2}, \dots, \lambda_{i,d})$, $P_i = (p_{i,1}, p_{i,2}, \dots, p_{i,d})$, $\lambda_{i,j}$ are the j-th eigenvalue of $p_{i,j}$, and D is the corresponding eigenvector. Bringing 1-4 into equation 1-3, there are:

$$g_{qdf}(x,\omega_i) = \sum_{j=1}^{d} \frac{\left[(x-\mu_i)^t p_{i,j}\right]^2}{\lambda_{i,j}} + \sum_{j=1}^{d} ln\, \lambda_{i,j} \qquad 1-5$$

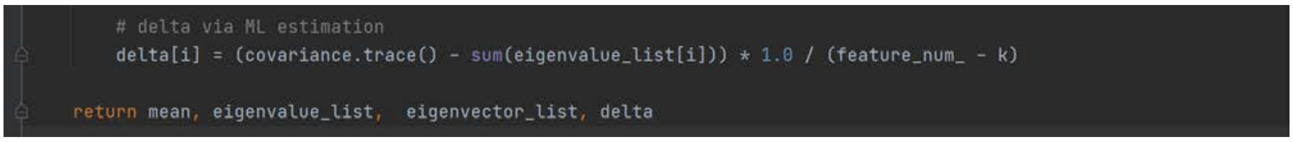
Code:



```python
def MQDF(class_num, train_x, train_y, k):
    """
    Build the MQDF model
    """

    feature_num_ = len(train_x[0])   # number of features
    assert(k<feature_num_ and k>0)

    data = []
    train_length = len(train_x)
    for i in range(class_num):
        data.append(list())

    for i in range(train_length):
        class_index = int(train_y[i])
        data[class_index].append(train_x[i])

    mean = []
    cov_matrix = []
    prior = []
```

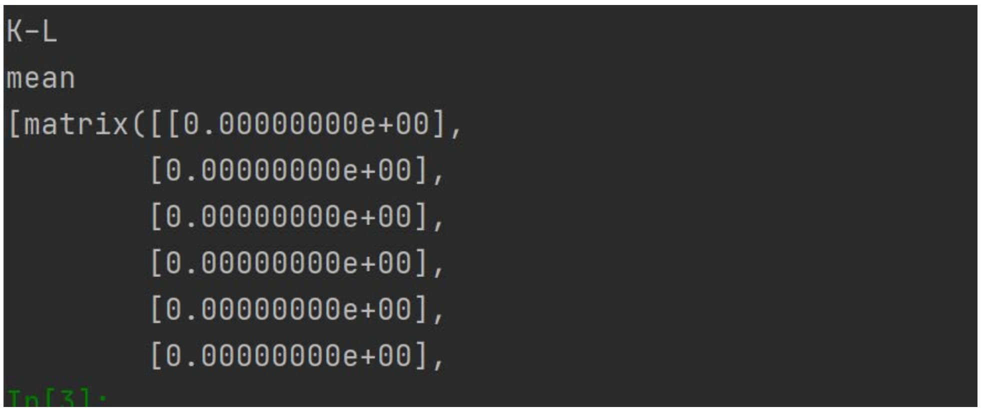


```python
    for i in range(class_num):
        data[i] = np.matrix(data[i], dtype=np.float64)
        mean.append(data[i].mean(0).T)

        # np.cov treat each row as one feature, so data[i].T has to be transposed
        cov_matrix.append(np.matrix(np.cov(data[i].T)))
        prior.append(len(data[i]) * 1.0 / train_length)


    eigenvalue_list = []    # store the first largest k eigenvalue_lists of each class
    eigenvector_list = []   # the first largest k  eigenvector_lists, column-wise of each class
    delta = [0] * class_num # delta for each class
    for i in range(class_num):
        covariance = cov_matrix[i]
        eig_value, eig_vector = linalg.eigh(covariance)

        # sort the eigvalues
        index_ = eig_value.argsort()
        index_ = index_[::-1]   # reverse the array
        eig_value = eig_value[index_]
        eig_vector = eig_vector[:, index_]

        eigenvector_list.append(eig_vector[:, 0:k])
        eigenvalue_list.append(eig_value[:k])
```

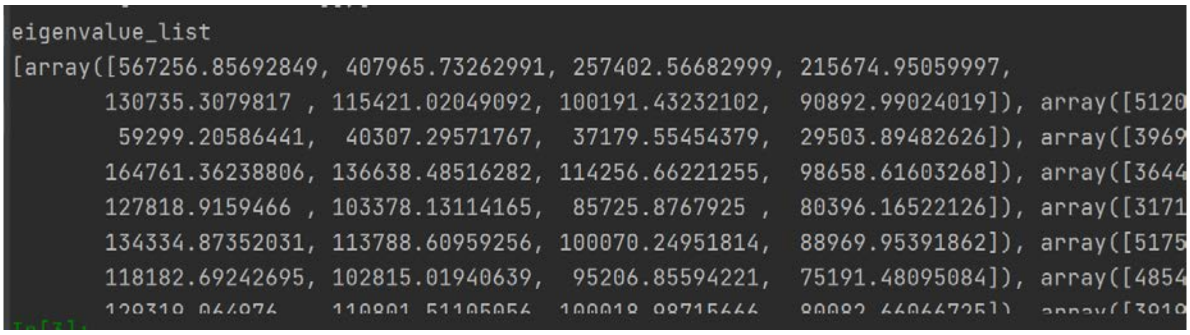


```python
        # delta via ML estimation
        delta[i] = (covariance.trace() - sum(eigenvalue_list[i])) * 1.0 / (feature_num_ - k)

    return mean, eigenvalue_list,  eigenvector_list, delta
```

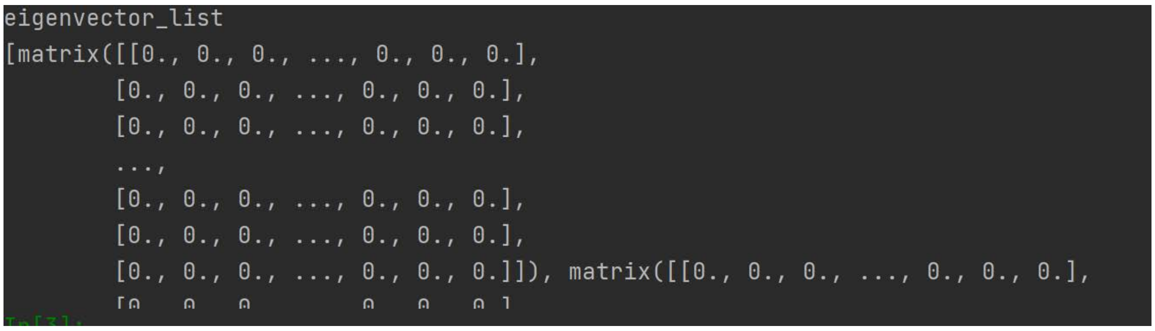
Result:



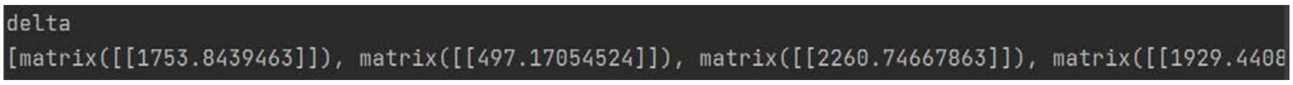
```
K-L
mean
[matrix([[0.00000000e+00],
        [0.00000000e+00],
        [0.00000000e+00],
        [0.00000000e+00],
        [0.00000000e+00],
        [0.00000000e+00],
In[3]:
```



```
eigenvalue_list
[array([567256.85692849, 407965.73262991, 257402.56682999, 215674.95059997,
       130735.3079817 , 115421.02049092, 100191.43232102,  90892.99024019]), array([5120
        59299.20586441,  40307.29571767,  37179.55454379,  29503.89482626]), array([3969
       164761.36238806, 136638.48516282, 114256.66221255,  98658.61603268]), array([3644
       127818.9159466 , 103378.13114165,  85725.8767925 ,  80396.16522126]), array([3171
       134334.87352031, 113788.60959256, 100070.24951814,  88969.95391862]), array([5175
       118182.69242695, 102815.01940639,  95206.85594221,  75191.48095084]), array([4854
       129319.064976 , 110801.51105054, 100018.98715666,  80082.46046725]), array([3919
In[3]:
```



```
eigenvector_list
[matrix([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]]), matrix([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
In[3]:
```



```
delta
[matrix([[1753.8439463]]), matrix([[497.17054524]]), matrix([[2260.74667863]]), matrix([[1929.4408
```

- MQDF predict

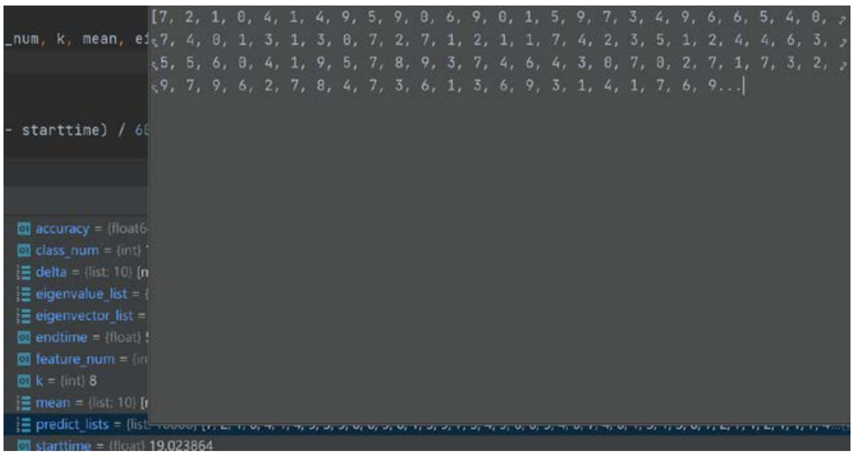
The parameters required by the QDF classifier can be obtained from the training data according to the maximum likelihood estimation. In comparison, the training is relatively simple and direct, and can obtain high accuracy, so it is widely used in practice. But when the input feature dimensionality is high and the training data is insufficient, there will be a curse of dimensionality phenomenon, which is manifested in the QDF classifier, which is an estimation error. It can be seen from formula (1-5) that for eigenvalues, the same estimation error has a much greater impact on the result when the eigenvalue is smaller than when the eigenvalue is larger.

$$g_2(x,\omega_i) = \frac{1}{h_i^2}\left\{||x-\mu_i||^2 - \sum_{j=1}^{k}\left(1-\frac{h_i^2}{\lambda_i}\right)\left[(x-\mu_i)^t p_{ij}\right]^2 + \sum_{j=1}^{k} ln\lambda_i + (d-k)lnh_i^2\right\}$$

Code:

```
# formula
    for i in range(class_num):

      minus = np.linalg.norm(x.reshape((d,)) - mean[i].reshape((d,))) ** 2
      matrix_minus = [0] * d
      for j in range(k):
        matrix_minus[j] = (((x - mean[i]).T * eigenvector_list[i][:, j])[0,0])**2

      g = 0
      for j in range(k):
        g += (matrix_minus[j] * 1.0 / eigenvalue_list[i][j])

      g += ((minus - sum(matrix_minus)) / delta[i])

      for j in range(k):
        g += math.log(eigenvalue_list[i][j])

      g += ((d - k) * math.log(delta[i]))


      if g < min_posteriori:
        min_posteriori = g
        prediction = i
```

Result:

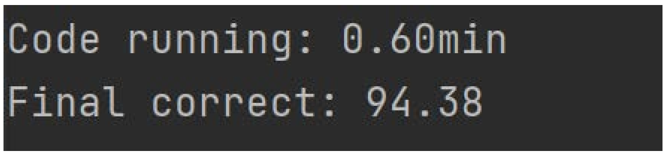


```
_num, k, mean, ei
- starttime) / 6(
```

```
[7, 2, 1, 0, 4, 1, 4, 9, 5, 9, 0, 6, 9, 0, 1, 5, 9, 7, 3, 4, 9, 6, 6, 5, 4, 0,
7, 4, 0, 1, 3, 1, 3, 0, 7, 2, 7, 1, 2, 1, 1, 7, 4, 2, 3, 5, 1, 2, 4, 4, 6, 3,
5, 5, 6, 0, 4, 1, 9, 5, 7, 8, 9, 3, 7, 4, 6, 4, 3, 0, 7, 0, 2, 7, 1, 7, 3, 2,
9, 7, 9, 6, 2, 7, 8, 4, 7, 3, 6, 1, 3, 6, 9, 3, 1, 4, 1, 7, 6, 9...
```

```
accuracy = {float6
class_num = {int}
delta = {list: 10} [n
eigenvalue_list = {
eigenvector_list =
endtime = {float}
feature_num = {in
k = {int} 8
mean = {list: 10} [
predict_lists = {list
starttime = {float} 19.023864
```

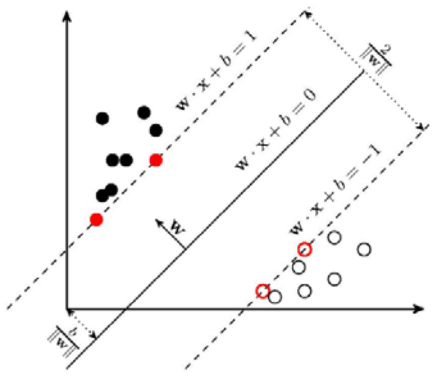
- Accuracy



```
Code running: 0.60min
Final correct: 94.38
```

3 Implement Support Vector Machine (SVM) on MNIST

A Support Vector Machine constructs a hyperplane or set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks.



$\mathbf{w}\cdot\mathbf{x}+b=1$
$\mathbf{w}\cdot\mathbf{x}+b=0$
$\mathbf{w}\cdot\mathbf{x}+b=-1$
$\frac{2}{\|\mathbf{w}\|}$
$\frac{b}{\|\mathbf{w}\|}$
$\mathbf{w}$

$$\text{max. } W(\alpha)=\sum_{i=1}^{n}\alpha_i-\frac{1}{2}\sum_{i=1,j=1}^{n}\alpha_i\alpha_j y_i y_j K(\mathbf{x}_i,\mathbf{x}_j)$$

$$b=1-\frac{1}{|i:0<\alpha_i<C|}\sum_{i:0<\alpha_i<C}\sum_{j=1}^{s}\alpha_j y_j k(\mathbf{x}_j,\mathbf{x}_i)$$

$$C\geq\alpha_i\geq 0,\ \sum_{i=1}^{n}\alpha_i y_i=0$$

Polynomial Kernel: $k(\mathbf{x}_1,\mathbf{x}_2)=(\mathbf{x}_1^{T}\,\mathbf{x}_2+1)^{d}$

RBF Kernel: $k(\mathbf{x}_1,\mathbf{x}_2)=\exp(-||\mathbf{x}_1-\mathbf{x}_2||^2/2s^2)$

Github is a good choice for searching codes (https://github.com). Download the python code available in https://github.com/prashantkh19/MNIST_SVM. Adjust and modify the codes to implement two SVM models with Polynomial Kernel and RBF Kernel respectively on MNIST. Show the training process and classification results.

A. Principle

Describe sklearn definition and parameters of SVM.

Here we choose SVC, which means using SVM for classification. The specific use is here. About the introduction of the parameters in the code:

➢ `C = 1.0`

$$L(w,b,\xi,\alpha,\beta):=\frac{1}{2}w^Tw+C\sum_{i=1}^{m}\xi_i+\sum_{i=1}^{m}\alpha_i\big(1-\xi_i-y_i(w^T\emptyset(x_i)+b)\big)+\sum_{i=1}^{m}\beta_i(-\xi_i)$$

It can be seen that C is used to balance structural risk and experience risk.

➢ `kernel='rbf'`

Specifies the kernel type to be used in the algorithm.

It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable.

If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape ``(n_samples, n_samples)``.

➢ `gamma='scale'`

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

if ``gamma='scale'`` (default) is passed then it uses:

$$\gamma = \frac{1}{(n\_features\ * \ X.var())}$$

if 'auto', uses :

$$\frac{1}{n\_features}$$

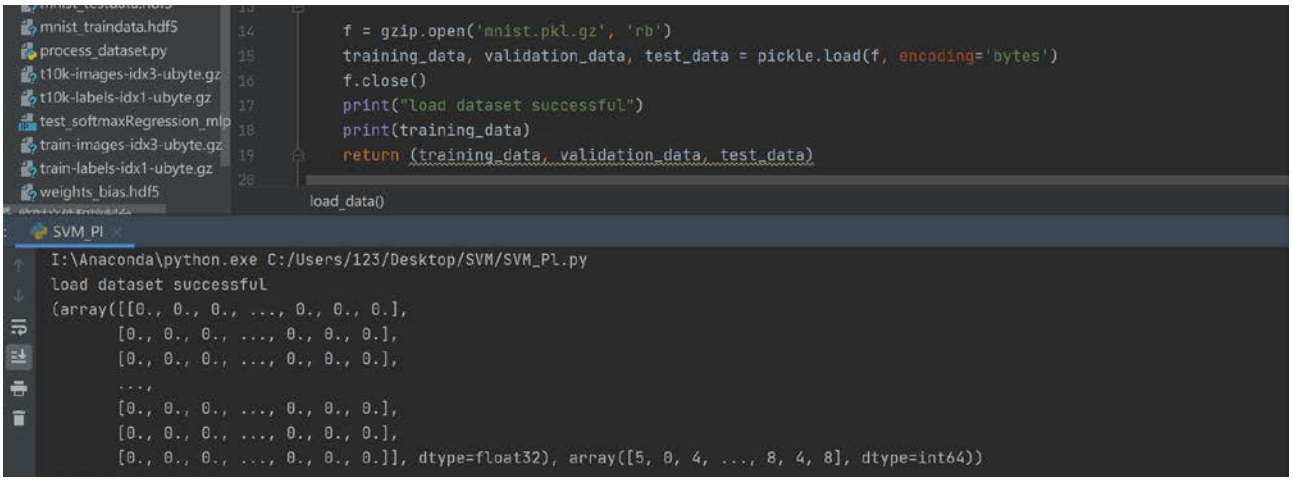
B. Code and Result

● Load data

Code:

```
def load_data():
    """
    Return pattern recognition data containing tuples of training data, verification
 data, and test data
    The training data contains 50,000 pictures, and the test data and verification d
ata only contain 10,000 pictures
    """
    f = gzip.open('mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = pickle.load(f, encoding='bytes')
    f.close()
    print("load dataset successful")
    return (training_data, validation_data, test_data)
```

Result:


mnist_traindata.hdf5
process_dataset.py
t10k-images-idx3-ubyte.gz
t10k-labels-idx1-ubyte.gz
test_softmaxRegression_mlp
train-images-idx3-ubyte.gz
train-labels-idx1-ubyte.gz
weights_bias.hdf5

```
    f = gzip.open('mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = pickle.load(f, encoding='bytes')
    f.close()
    print("load dataset successful")
    print(training_data)
    return (training_data, validation_data, test_data)
```

load_data()

SVM_Pl

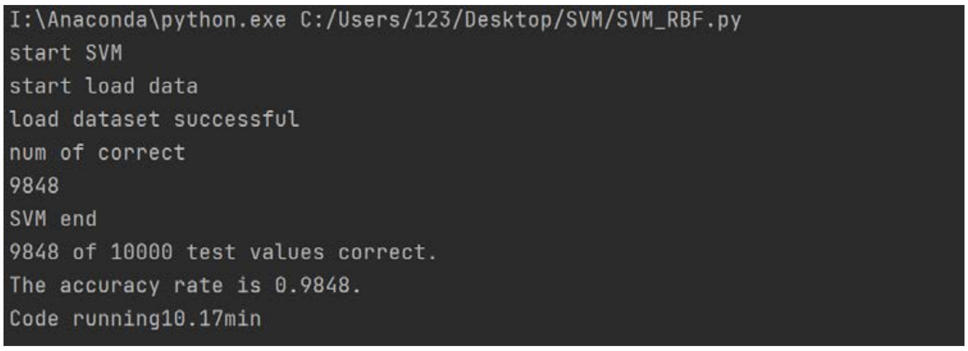
```
I:\Anaconda\python.exe C:/Users/123/Desktop/SVM/SVM_Pl.py
load dataset successful
(array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), array([5, 0, 4, ..., 8, 4, 8], dtype=int64))
```

- SVM

Code:

```
# Pass the parameters of the training model
    clf = svm.SVC(C=100.0, kernel='rbf', gamma=0.03)

    # model train
    clf.fit(training_data[0], training_data[1])

    # test
    predictions = [int(a) for a in clf.predict(test_data[0])]
    num_correct = sum(int(a == y) for a, y in zip(predictions, test_data[1]))
```

Result:


```
I:\Anaconda\python.exe C:/Users/123/Desktop/SVM/SVM_RBF.py
start SVM
start load data
load dataset successful
num of correct
9848
SVM end
9848 of 10000 test values correct.
The accuracy rate is 0.9848.
Code running10.17min
```

- Change kernel

Code:

```
# Pass the parameters of the training model
    clf = svm.SVC(C=100.0, kernel='poly', gamma=0.03)
```

Result:

4 Analysis and Comparison

Analyze and compare the advantages and disadvantages of these three models in classification from different aspects, such as effectiveness, efficiency, complexity. Your analysis and conclusion should be well justified theoretically and/or empirically.

	time	*accuracy*	*CPU utilization(i5-1021U)*
MLP	*42min*	*94.86*	*95%*
MQDF	*1.36min*	*95.53*	*30%*
SVM RBF	*7.96min*	*98.48*	*30%*
SVM poly	*2.46min*	*97.78*	*29%*

The MLP algorithm requires a large number of back propagation and gradient descent processes, so the gradient descent threshold becomes the main parameter that determines the time complexity of the entire algorithm. We can see that the MLP algorithm consumes a lot of time and occupies a lot of CPU because it has a large number of calculation processes and the overall space complexity is very high.

Because MQDF is a Bayesian classification algorithm, it is a kind of lazy algorithm. It does not need train and only uses calculation accuracy. Therefore, MQDF has good time complexity, runs fast, and has accuracy very high.

For SVM, the support vector is the training result of the SVM, and it is the support vector that plays a decisive role in the SVM classification decision. In essence, it avoids the traditional process from induction to deduction, realizes efficient "transduction inference" from training samples to forecast samples, and greatly simplifies the usual classification and regression problems. The final decision function of SVM is determined by only a few support vectors, and the complexity of the calculation depends on the number of support vectors, not the dimensionality of the sample space, which avoids the "dimension disaster" in a sense. Therefore, the time consumption corresponding to different kernels is also different.

4 Comparison and improvement

A. Processing of data sets

i. PKL

Because in the program, directly importing the IDX FILE FORMAT format database is more troublesome, so the database should be preprocessed in advance and processed as Numpy format and transferred to B format.

- .pkl data file: In Python, the Pickle module converts any Python object into a system byte, similar to the JSON format, but not readable by humans.
- One-hot encoding: Use n bits to express n states, the correct state is expressed by 1, and the others are all 0. For example, 2 is represented as [0,0,1,0,0,0,0,0,0,0]
- """Read into the MNIST data set

```
Parameters
----------
normalize: Normalize the pixel value of the image to 0.0~1.0
one_hot_label:
    When one_hot_label is True, the label is returned as a one-hot array
    One-hot array refers to an array like [0,0,1,0,0,0,0,0,0,0]
flatten: whether to expand the image into a one-dimensional array

Returns
```

(Training image, training label), (test image, test label)

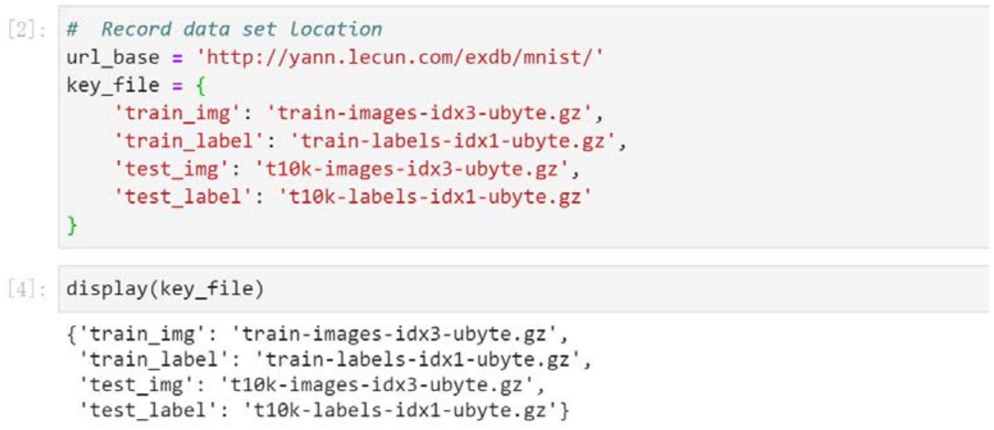
"""

Code and Result:

●

```
#  Record data set location
url_base = 'http://yann.lecun.com/exdb/mnist/'
key_file = {
    'train_img': 'train-images-idx3-ubyte.gz',
    'train_label': 'train-labels-idx1-ubyte.gz',
    'test_img': 't10k-images-idx3-ubyte.gz',
    'test_label': 't10k-labels-idx1-ubyte.gz'
}
```

Result:



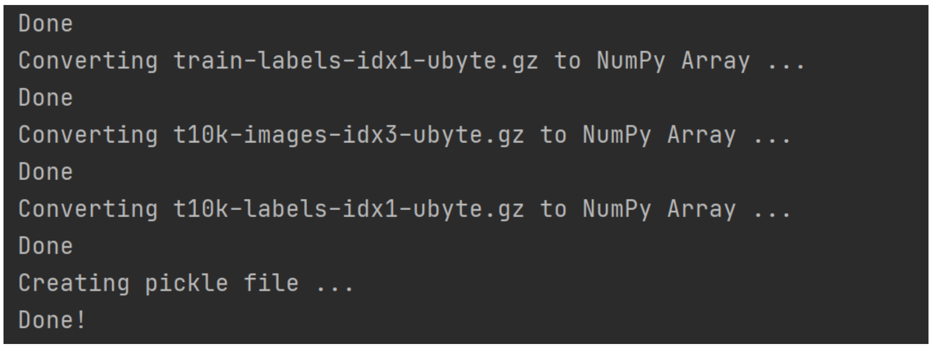
```
[2]: #  Record data set location
     url_base = 'http://yann.lecun.com/exdb/mnist/'
     key_file = {
         'train_img': 'train-images-idx3-ubyte.gz',
         'train_label': 'train-labels-idx1-ubyte.gz',
         'test_img': 't10k-images-idx3-ubyte.gz',
         'test_label': 't10k-labels-idx1-ubyte.gz'
     }

[4]: display(key_file)

     {'train_img': 'train-images-idx3-ubyte.gz',
      'train_label': 'train-labels-idx1-ubyte.gz',
      'test_img': 't10k-images-idx3-ubyte.gz',
      'test_label': 't10k-labels-idx1-ubyte.gz'}
```

●

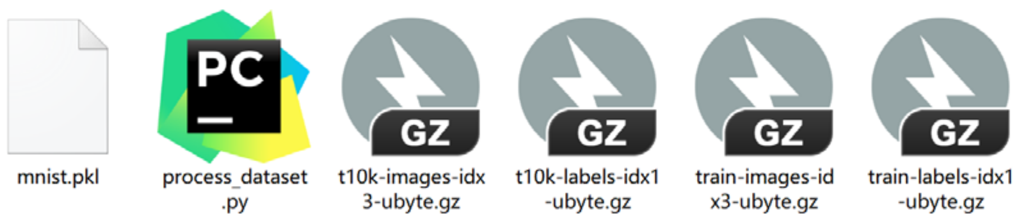
```
def _change_one_hot_label(X):
    T = np.zeros((X.size, 10))
    for idx, row in enumerate(T):
        row[X[idx]] = 1

    return T
```

```python
def load_mnist(normalize=True, flatten=True, one_hot_label=False):

    if not os.path.exists(save_file):
        init_mnist()

    with open(save_file, 'rb') as f:
        dataset = pickle.load(f)

    if normalize:
        for key in ('train_img', 'test_img'):
            dataset[key] = dataset[key].astype(np.float32)
            dataset[key] /= 255.0

    if one_hot_label:
        dataset['train_label'] = _change_one_hot_label(dataset['train_label'])
        dataset['test_label'] = _change_one_hot_label(dataset['test_label'])

    if not flatten:
        for key in ('train_img', 'test_img'):
            dataset[key] = dataset[key].reshape(-1, 1, 28, 28)

    return (dataset['train_img'], dataset['train_label']), (dataset['test_img'], dataset['test_label'])
```

Result:



```
Done
Converting train-labels-idx1-ubyte.gz to NumPy Array ...
Done
Converting t10k-images-idx3-ubyte.gz to NumPy Array ...
Done
Converting t10k-labels-idx1-ubyte.gz to NumPy Array ...
Done
Creating pickle file ...
Done!
```

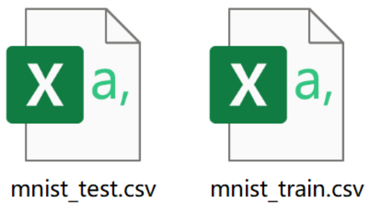

mnist.pkl
process_dataset.py
t10k-images-idx3-ubyte.gz
t10k-labels-idx1-ubyte.gz
train-images-idx3-ubyte.gz
train-labels-idx1-ubyte.gz

ii. CVS

Code:

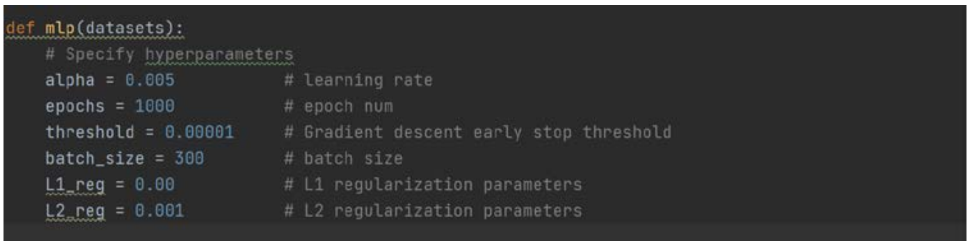
```
def convert(imgf, labelf, outf, n):
    f = open(imgf, "rb")
    o = open(outf, "w")
    l = open(labelf, "rb")

    f.read(16)
    l.read(8)
    images = []

    for i in range(n):
        image = [ord(l.read(1))]
        for j in range(28*28):
            image.append(ord(f.read(1)))
        images.append(image)

    for image in images:
        o.write(",".join(str(pix) for pix in image)+"\n")
    f.close()
    o.close()
    l.close()

convert("MNIST/train-images.idx3-ubyte", "MNIST/train-labels.idx1-ubyte",
        "mnist_train.csv", 60000)
convert("MNIST/t10k-images.idx3-ubyte", "MNIST/t10k-labels.idx1-ubyte",
        "mnist_test.csv", 10000)

print("Convert Finished!")
```

Result:



B. MLP parameter improvements

We have set certain hyperparameters in the program, we can change them to get the best prediction effect.



```
def mlp(datasets):
    # Specify hyperparameters
    alpha = 0.005             # learning rate
    epochs = 1000             # epoch num
    threshold = 0.00001       # Gradient descent early stop threshold
    batch_size = 300          # batch size
    L1_reg = 0.00             # L1 regularization parameters
    L2_reg = 0.001            # L2 regularization parameters
```

alpha	*0.001*	*0.005*	*0.01*	*0.015*	*0.02*	*0.05*	*0.1*
accuracy	*90.91*	*94.03*	*94.30*	*94.86*	*94.72*	*92.74*	*92.37*

Activate function	*sigmoid*	*tanh*
accuracy	*94.86*	*94.61*
Time	*42min*	*29min*

C. MQDF parameter improvements

K	*2*	*4*	*6*	*8*	*10*	*12*	*14*
accuracy	*89.11*	*92.27*	*93.61*	*94.38*	*94.82*	*95.07*	*94.88*

D. SVM parameter improvements

Ensure that other parameters remain unchanged

gamma	*0.01*	*0.02*	*0.03*	*0.04*	*0.05*	*0.06*
accuracy	*0.9823*	*0.9847*	*0.9848*	*0.984*	*0.9828*	*0.9804*

5 Conclusion

We can see that the accuracy of the three strategies is already ideal, but the time consumed by the three strategies and the size of the model are very different, depending on the internal algorithms corresponding to the different strategies. Moreover, under a certain strategy, adjusting the parameters can often achieve a relatively good predictive value.

REFERENCES

[1] F. Kimura, et zl., Modified quadratic discriminant functions and the application
to Chinese character recognition, IEEE Trans. PAMI, 9(1): 149-153, 1987.

[2] 姚超. (2014). 降维算法和手写文字识别中若干问题研究 (Doctoral dissertation, 西安电子科技大学)