DDA5001: Homework #1

Due on September 28, 2025 at 23:59

Professor DAI Zhongxiang Term 1, 2025-2026

LIU Yibo

Problem 1. Concept and Fundamental Knowledge

Solution:

(a) The main difference between supervised learning and unsupervised learning is **the use of labeled data**. In supervised learning, each data point is tagged with a correct output, while in unsupervised learning, there are no predefined output labels. The character of the data they utilize determines the tasks they commonly deal with: supervised Learning used to solve *classification* and *regression*; unsupervised learning usually suits the conditions of *Clustering*, *association*, and *dimensionality reduction*.

(b) 1) **False.** Regression is used to predict a continuous value, such as stock prices, while classification fits categorical labels.
2) **True.**
3) **Flase.** For any linearly separable dataset, there are typically infinite possible hyperplanes that can seperate the data.
4) **False.** Least squares is only equivalent to an MLE under the assumption that the errors are independent and Gaussian distributed with a mean of zero. Without the Gaussian error assumption, it is not an MLE.

(c) Denote the columns of full rank matrix $\boldsymbol{X}$ as $\boldsymbol{X}_1, \boldsymbol{X}_2, \dots, \boldsymbol{X}_n$, so that

$$\boldsymbol{X} = [\boldsymbol{X}_1, \boldsymbol{X}_2, \dots, \boldsymbol{X}_n]. \tag{1}$$

With the independence of the columns, for any vector $\boldsymbol{v} = [v_1, v_2, \dots, v_n]^T \neq \mathbf{0}$, we have

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{v} = \sum_{i=1}^{n} \boldsymbol{X}_i v_i \neq 0. \tag{2}$$

Therefore, we obtain

$$\boldsymbol{v}^T(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{v} = (\boldsymbol{X}\boldsymbol{v})^T(\boldsymbol{X}\boldsymbol{v}) = \boldsymbol{y}^T\boldsymbol{y} > 0, \tag{3}$$

which says that $\boldsymbol{X}^T\boldsymbol{X}$ is positive definite.

Problem 2. Least Squares Without Full Column Rank

Solution:

(a) With the singular value decomposition (SVD), we can substitute $X = V\Sigma_1 U_1^T$ into the original form:

$$\|X\theta - y\|_2^2 = \|V\Sigma_1 U_1 \theta - y\|_2^2, \tag{4}$$

where $V \in \mathbb{R}^{n\times n}$ is orthogonal, $\Sigma_1 \in \mathbb{R}^{n\times n}$ is diagonal with positive singular value $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$, and $U_1^T \in \mathbb{R}^{n\times d}$ is a semi-orthogonal matrix. Let $A := V\Sigma_1$ and $z := U_1^T\theta$, the initial problem is equivalent to

$$\min_z \|Az - y\|_2^2. \tag{5}$$

The minimal solutions can be obtained when $Az^* = y$, i.e., $z^* = A^{-1}y = \Sigma_1^{-1}V^T y$.
Let $U = [U_1, U_2]$ and $w := U_2^T\theta$, then we have

$$\begin{aligned}\theta &= U(U^T\theta) = [U_1, U_2]\begin{bmatrix}U_1^T\theta\\ U_2^T\theta\end{bmatrix}\\ &= U_1(U_1^T\theta) + U_2(U_2^T\theta)\\ &= U_1 z + U_2 w.\end{aligned} \tag{6}$$

Here, w is an arbitrary vector in $\mathbb{R}^{n-d}$, leading to the infinite solutions of least squares without full rank. This is because $n < d$, thus the null space of X is non-trivial (Since $XU_2 = (V\Sigma_1 U_1^T)U_2 = V\Sigma_1(U_1^T U_2) = 0$, the column of U_2 form an orthonormal basis for the null space of X). Any vector from the null space of X can be added to a particular solution $\hat{\theta}$ without changing the result of $X\theta$.

(b) Let the objective function be $J(\theta)$:

$$\begin{aligned}J(\theta) &= \|X\theta - y\|_2^2 + \lambda\|\theta\|_2^2\\ &= (\theta^T X^T - y^T)(X\theta - y) + \lambda\theta^T\theta\\ &= \theta^T X^T X\theta - 2y^T X\theta + y^T y + \lambda\theta^T\theta.\end{aligned} \tag{7}$$

Then we take the gradient of $J(\theta)$:

$$\nabla_\theta J(\theta) = 2X^T X\theta - 2X^T y + 2\lambda\theta. \tag{8}$$

Solving for θ by setting the gradient to zero, we obtain:

$$(X^T X + \lambda I)\theta = X^T y. \tag{9}$$

Since $X^T X$ is a positive semi-definite matrix and λI is a positive matrix, $(X^T X + \lambda I)$ is invertible. Thus, we have

$$\theta = (X^T X + \lambda I)^{-1} X^T y. \tag{10}$$

Problem 3. Robust Linear Regression

Solution:

(a) Denote the probability of observing the given data y for a specific parameter θ is $L(\theta)$, $L(\theta) = p(y|X, \theta)$. Since the error term $\epsilon \overset{i.i.d}{\sim} \mathcal{N}(0, \Sigma)$, y_i are also conditionally independent. Therefore, we obtain the total likelihood:

$$L(\theta) = \prod_{i=1}^{n} p(y_i|x_i, \theta), \tag{11}$$

Given an input x and θ, the randomness of y only comes from the error term ϵ. Thus, we have

$$p(y_i|x_i, \theta) = p_\epsilon(\epsilon_i) = p_\epsilon(y_i - x_i^T\theta) = \frac{1}{2b}\exp\left(-\frac{|y_i| - x_i^T\theta}{b}\right). \tag{12}$$

Substituting $p(y_i|x_i, \theta)$ into Eq.(11), the log-likelihoood $\mathcal{L}(\theta)$ can be derived as:

$$\begin{aligned}
\mathcal{L}(\theta) &= \log L(\theta) \\
&= \log\left(\prod_{i}^{n} \frac{1}{2b}\exp\left(-\frac{|y_i - x_i^T\theta|}{b}\right)\right) \\
&= \sum_{i=1}^{n} \log\frac{1}{2b}\exp\left(-\frac{|y_i - x_i^T\theta|}{b}\right) \\
&= -n\log 2b + \sum_{i=1}^{n} -\frac{|y_i - x_i^T\theta|}{b}.
\end{aligned} \tag{13}$$

The maximum likelihood estimation θ^* is the value of θ that maximizes the log-likelihood function:

$$\begin{aligned}
\theta^* &= \arg\max_\theta \mathcal{L}(\theta) \\
&= \arg\max_\theta \left(-n\log 2b - \frac{1}{b}\sum_{i=1}^{n}|y_i - x_i^T\theta|\right) \\
&= \arg\min_\theta \left(n\log 2b + \frac{1}{b}\sum_{i=1}^{n}|y_i - x_i^T\theta|\right) \\
&= \arg\min_\theta \sum_{i=1}^{n}|y_i - x_i^T\theta|.
\end{aligned} \tag{14}$$

This equation represents the L1-norm of the residual vector $y - X\theta$. Therefore, when ϵ follows the Laplace distribution, the machine learning problem formulation for estimation θ^* is

$$\min_\theta \|X\theta - y\|_1. \tag{15}$$

(b) We first define the residual vector as $\boldsymbol{r} = X\theta - y$, with components $r_j = x_j^T\theta - y_j, j = 1, \ldots, n$, where x_j is the j-th row of X.
The loss function is defined as:

$$\mathcal{L}(\theta) = H_\mu(r) = \sum_{j=1}^{n} h_\mu(r_j) \tag{16}$$

Then, the derivative of the Huber function is:

$$h'_\mu(z) = \begin{cases} \frac{z}{\mu}, & \text{if } |z| \le \mu \\ \operatorname{sgn}(z), & \text{if } |z| > \mu \end{cases} \tag{17}$$

Here, sgn is a function that meets

$$\operatorname{sgn}(z) = \begin{cases} -1, & \text{if } z < 0 \\ 0, & \text{if } z = 0 \\ 1, & \text{if } z > 0 \end{cases} \tag{18}$$

This derivative is continuous and well-defined everywhere.
The j-th component of the gradient vector $\nabla_r \mathcal{L}(\theta)$ is $h'_\mu(r_j)$. Define a vector $g = \nabla_r H_\mu(r)$ such that its components are given by:

$$g_j = h'_\mu(r_j) = \begin{cases} r_j/\mu, & \text{if } |r_j| \le \mu \\ \operatorname{sgn}(r_j), & \text{if } |r_j| > \mu \end{cases} \tag{19}$$

where $r_j = x_j^T \theta - y_j$.
By the vector chain rule, we can obtain the gradient is

$$\nabla_\theta \mathcal{L}(\theta) = \left(\frac{\partial r}{\partial \theta}\right)^T \nabla_r H_\mu(r) \tag{20}$$

Since $r = X\theta - y$, the Jacobian matrix $\frac{\partial r}{\partial \theta}$ is simply X. With the $g = \nabla_r H_\mu(r)$, we have

$$\nabla_\theta \mathcal{L}(\theta) = X^T g. \tag{21}$$

(c) The implementation process is in the code file p3.
The results of programming show that: As expected, the error decreases rapidly as the number of iterations increases, indicating that the gradient descent algorithm is effectively minimizing the Huber loss function and that our estimated parameters θ are converging to the true values θ^*.

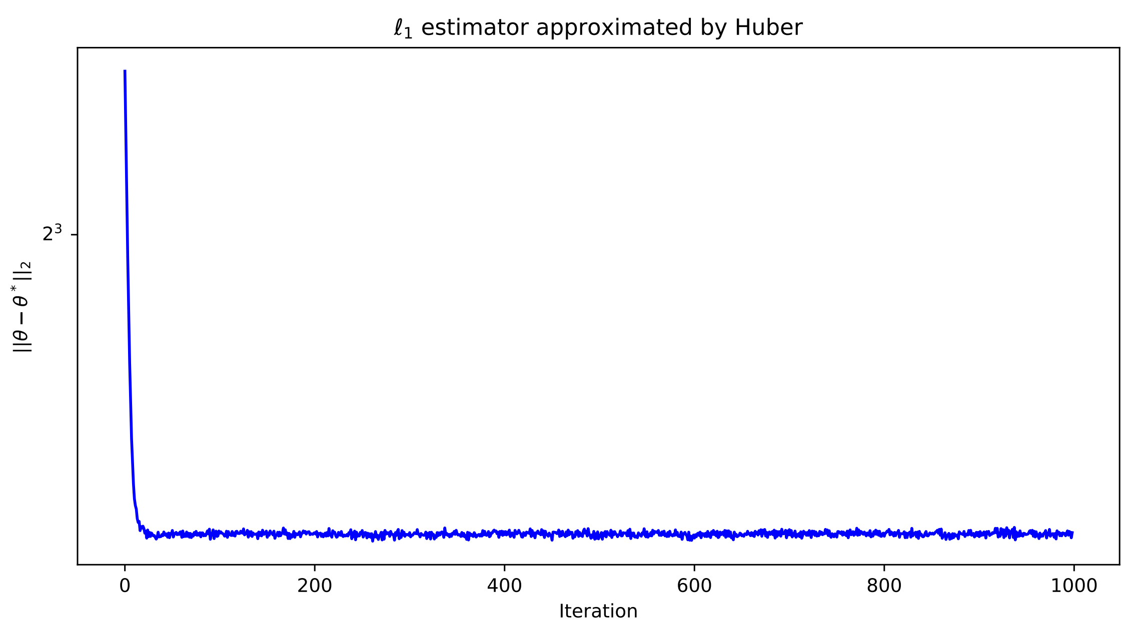

Figure 1: Error Convergence of L1 Estimator via Huber Smoothing

Problem 4. Convergence of The Perceptron for Linearly Separable Data

Solution:

(a) Since the data is linearly separable, for every i, we have

$$y_i({\theta^*}^\top x_i) > 0. \tag{22}$$

we can obtain that θ^* correctly classifies all samples. Because n is the number of training samples, the set $\{y_i({\theta^*}^\top x_i) \mid 1 \le i \le n\}$ is a finite collection of positive numbers. The minimum of a finite set of positive numbers is positive. Thus, we can obtain that

$$\rho = \min_{1\le i\le n} y_i({\theta^*}^\top x_i) > 0. \tag{23}$$

(b) According to the perceptron update rule $\theta_k = \theta_{k-1} + y_{k-1}x_{k-1}$, taking the inner product with θ^*, we can obtain:

$$\begin{aligned} \theta_k^\top \theta^* &= (\theta_{k-1} + y_{k-1}x_{k-1})^\top \theta^* \\ &= \theta_{k-1}^\top \theta^* + y_{k-1}(x_{k-1}^\top \theta^*). \end{aligned} \tag{24}$$

Since (x_{k-1}, y_{k-1}) is misclassified by θ_{k-1}, it is still correctly classified by θ^* according to the linear separability. Combining with the definition of ρ, we obtain:

$$y_{k-1}({\theta^*}^\top x_{k-1}) \ge \rho. \tag{25}$$

Substituting it into the above formula, we obtain

$$\theta_k^\top \theta^* \ge \theta_{k-1}^\top \theta^* + \rho. \tag{26}$$

Next, we prove $\theta_k^\top \theta^* \ge k\rho$ by mathematical induction: For the base case $k = 0$, $\theta_0 = 0$, so $\theta_0^\top \theta^* = 0 = 0 \cdot \rho$. Assume that when $k = m$, $\theta_m^\top \theta^* \ge m\rho$. Then when $k = m + 1$,

$$\begin{aligned} \theta_{m+1}^\top \theta^* &\ge \theta_m^\top \theta^* + \rho \\ &\ge m\rho + \rho \\ &= (m+1)\rho. \end{aligned} \tag{27}$$

By induction, $\theta_k^\top \theta^* \ge k\rho$ for all $k \ge 0$.

(c) According to the expansion formula of the vector norm, taking the square norm of $\theta_k = \theta_{k-1} + y_{k-1}x_{k-1}$, we can obtain

$$\|\theta_k\|^2 = \|\theta_{k-1}\|^2 + 2y_{k-1}(\theta_{k-1}^\top x_{k-1}) + \|y_{k-1}x_{k-1}\|^2. \tag{28}$$

Since $y_{k-1} \in \{+1, -1\}$, then $y_{k-1} = 1$. Further, we can obtain $\|y_{k-1}x_{k-1}\|^2 = y_{k-1}^2\|x_{k-1}\|^2 = \|x_{k-1}\|^2$. Meanwhile, the misclassifications of (x_{k-1}, y_{k-1}) by θ_{k-1} means that the sign of $\theta_{k-1}^\top x_{k-1}$ and y_{k-1} are opposite, i.e., $y_{k-1}(\theta_{k-1}^\top x_{k-1}) < 0$.
Substituting these two results into the norm expansion formula, we can get

$$\|\theta_k\|^2 \le \|\theta_{k-1}\|^2 + 2 \cdot 0 + \|x_{k-1}\|^2 = \|\theta_{k-1}\|^2 + \|x_{k-1}\|^2. \tag{29}$$

(d) From the conclusion of (c), there is $\|\theta_k\|^2 \le \|\theta_{k-1}\|^2 + \|x_{k-1}\|^2$. Since R is the maximum norm of all samples x_i, we have

$$\|\theta_k\|^2 \le \|\theta_{k-1}\|^2 + R^2. \tag{30}$$

Recursively expanding this inequality for $\theta_{k-1}, \theta_{k-2}, \ldots, \theta_0$ in turn, we can get

$$\|\theta_k\|^2 \leq \|\theta_0\|^2 + \sum_{i=0}^{k-1} \|x_i\|^2. \tag{31}$$

Since the initial parameter $\theta_0 = 0$, then $\|\theta_0\|^2 = 0$. The summation tern $\sum_{i=0}^{k-1} \|x_i\|^2$ contain k term, and each term does not exceed R^2. Therefore, we can obtain

$$\sum_{i=0}^{k-1} \|x_i\|^2 \leq kR^2. \tag{32}$$

Finally, we have

$$\|\theta_k\|^2 \leq kR^2. \tag{33}$$

(e) Using $\theta_k^\top \theta^* \geq k\rho$ in (b) and $\|\theta_k\|^2 \leq kR^2$ in (d), since all terms are positive, we can obtain that

$$\frac{\theta_k^\top \theta^*}{\|\theta_k\|} \geq \frac{k\rho}{\sqrt{k}R} = \sqrt{k}\frac{\rho}{R} \tag{34}$$

Then, with the Cauchy-Schwarz inequality to bound the number of iterations, we have $\theta_k^\top \theta^* \leq \|\theta_k\|\|\theta^*\|$, i.e., $\frac{\theta_k^\top \theta^*}{\|\theta_k\|} \leq \|\theta^*\|$. Combining $\sqrt{k}\frac{\rho}{R} \leq \frac{\theta_k^\top \theta^*}{\|\theta_k\|} \leq \|\theta^*\|$, we get

$$k \leq \frac{R^2\|\theta^*\|^2}{\rho^2}, \tag{35}$$

which shows that the perceptron cannot iterate infinitely. Let $\bar{k} = \left\lfloor \frac{R^2\|\theta^*\|^2}{\rho^2} \right\rfloor$then after at most $\bar{k}$ iterations, there will be no more misclassified samples and the algorithm terminates.

Problem 5. Pocket Algorithm for Non-Separable data

Solution

(1) The implementation process is in the code file p5.

(2)

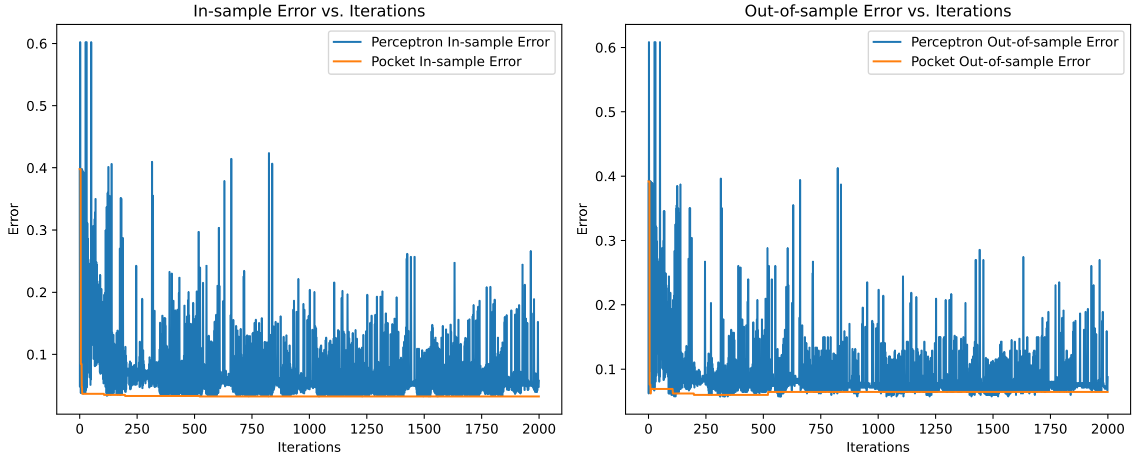

Figure 2: In-sample and Out-of-sample Error Comparison: Perceptron vs. Pocket Algorithm

(3)

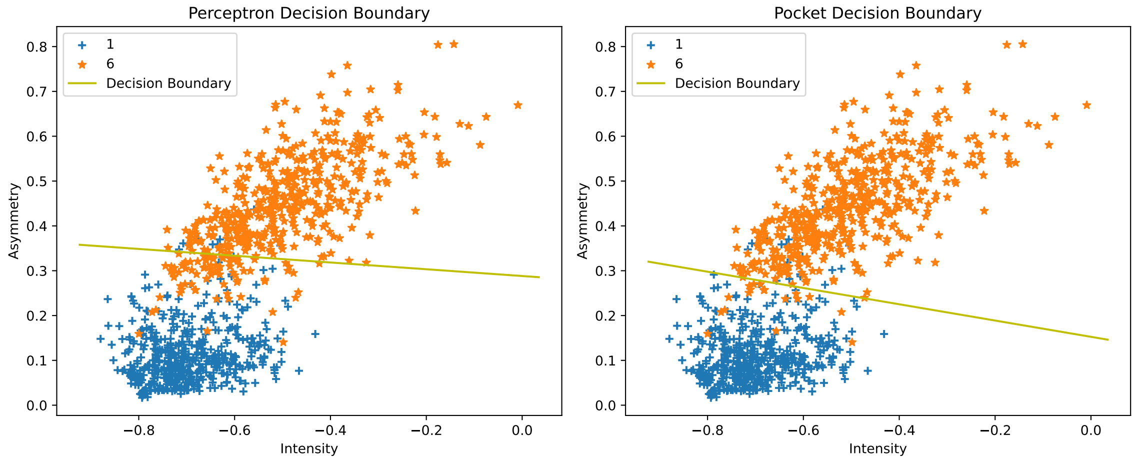

Figure 3: Decision Boundaries for Perceptron and Pocket Algorithms on Classifying Digits '1' and '6'

Code of P3

```python
import numpy as np
import matplotlib.pyplot as plt

d = 50 #feature dimension

# Load the dataset
X = np.load('p3/data/X.npy')
y = np.load('p3/data/y.npy')
print ("data shape: ", X.shape, y.shape)

theta_star = np.load('p3/data/theta_star.npy')

###### part (1): least square estimator ########

# Calculate the least square solution
theta_hat = np.linalg.inv(X.T @ X) @ X.T @ y

Error_LS = np.linalg.norm(theta_hat - theta_star, 2)
print('Estimator approximated by LS:',Error_LS)

###### part (2): L1 estimator ########
mu = 1e-5 # smoothing parameter
alpha = 0.001 # stepsize
T = 1000 # iteration number

# random initialization
theta = np.random.randn(d,1)

Error_huber = []

for _ in range(1, T):

    # calculate the l2 error of the current iteration
    Error_huber.append(np.linalg.norm(theta-theta_star, 2))

    # Calculate the residual
    r = y - X @ theta

    # Calculate the components of the gradient based on Huber loss
    g = np.where(np.abs(r) <= mu, r / mu, np.sign(r))

    # Calculate the final gradient
    grad = -X.T @ g

    #gradient descent update
    theta = theta - alpha * grad

```

```
####### plot the figure #########
plt.figure(figsize=(10,5))
plt.yscale('log',base=2)
plt.plot(Error_huber, 'b-')
plt.title(r'$\ell_1$ estimator approximated by Huber')
plt.ylabel(r'$||\theta - \theta^*||_2$')
plt.xlabel('Iteration')
# plt.grid(True)
plt.show()
```

Code of P5

```
    import scipy.io
import matplotlib.pyplot as plt
import numpy as np

def load_mat(path, d=16):
    data = scipy.io.loadmat(path)['zip']
    size = data.shape[0]
    y = data[:, 0].astype('int')
    X = data[:, 1:].reshape(size, d, d)
    return X, y

def cal_intensity(X):
    """
    X: (n, d, d), input data
    return intensity: (n, 1)
    """
    n = X.shape[0]
    return np.mean(X.reshape(n, -1), 1, keepdims=True)

def cal_symmetry(X):
    """
    X: (n, d, d), input data
    return symmetry: (n, 1)
    """
    n, d = X.shape[:2]
    Xl = X[:, :, :int(d/2)]
    Xr = np.flip(X[:, :, int(d/2):], -1)
    abs_diff = np.abs(Xl-Xr)
    return np.mean(abs_diff.reshape(n, -1), 1, keepdims=True)

def cal_feature(data):
    intensity = cal_intensity(data)
    symmetry = cal_symmetry(data)
    feat = np.hstack([intensity, symmetry])
    return feat

```

```
def cal_feature_cls(data, label, cls_A=1, cls_B=5):
    """ calculate the intensity and symmetry feature of given classes
    Input:
        data: (n, d1, d2), the image data matrix
        label: (n, ), corresponding label
        cls_A: int, the first digit class
        cls_B: int, the second digit class
    Output:
        X: (n’, 2), the intensity and symmetry feature corresponding to
            class A and class B, where n’= cls_A# + cls_B#.
        y: (n’, ), the corresponding label {-1, 1}. 1 stands for class A,
            -1 stands for class B.
    """
    feat = cal_feature(data)
    indices = (label==cls_A) + (label==cls_B)
    X, y = feat[indices], label[indices]
    ind_A, ind_B = y==cls_A, y==cls_B
    y[ind_A] = 1
    y[ind_B] = -1
    return X, y

def plot_feature(feature, y, plot_num, ax=None, classes=np.arange(10)):
    """plot the feature of different classes
    Input:
        feature: (n, 2), the feature matrix.
        y: (n, ) corresponding label.
        plot_num: int, number of samples for each class to be plotted.
        ax: matplotlib.axes.Axes, the axes to be plotted on.
        classes: array(0-9), classes to be plotted.
    Output:
        ax: matplotlib.axes.Axes, plotted axes.
    """
    cls_features = [feature[y==i] for i in classes]
    marks = [’s’, ’o’, ’D’, ’v’, ’p’, ’h’, ’+’, ’x’, ’<’, ’>’]
    colors = [’r’, ’g’, ’b’, ’c’, ’m’, ’y’, ’k’, ’cyan’, ’orange’, ’purple’]
    if ax is None:
        _, ax = plt.subplots()
    for i, feat in zip(classes, cls_features):
        ax.scatter(*feat[:plot_num].T, marker=marks[i], color=colors[i], label=str(i))
    plt.legend(loc=’upper right’)
    plt.xlabel(’intensity’)
    plt.ylabel(’asymmetry’)
    return ax

def cal_error(theta, X, y, thres=1e-4):
    """calculate the binary error of the model w given data (X, y)
    theta: (d+1, 1), the weight vector
    X: (n, d), the data matrix [X, y]
    y: (n, ), the corresponding label
```

```python
    """
    # Add a bias term to X
    X_b = np.hstack([np.ones((X.shape[0], 1)), X])
    out = X_b @ theta - thres
    pred = np.sign(out)
    err = np.mean(pred.squeeze()!=y)
    return err

# prepare data
train_data, train_label = load_mat('p5/train_data.mat') # train_data: (7291, 16, 16),
    train_label: (7291, )
test_data, test_label = load_mat('p5/test_data.mat') # test_data: (2007, 16, 16),
    train_label: (2007, )

cls_A, cls_B = 1, 6
X, y, = cal_feature_cls(train_data, train_label, cls_A=cls_A, cls_B=cls_B)
X_test, y_test = cal_feature_cls(test_data, test_label, cls_A=cls_A, cls_B=cls_B)

# Add a bias term to the feature matrices
X_b = np.hstack([np.ones((X.shape[0], 1)), X])
X_test_b = np.hstack([np.ones((X_test.shape[0], 1)), X_test])

# train
iters = 2000
d = 2
num_sample = X.shape[0]
threshold = 1e-4

# Perceptron and Pocket algorithm initialization
theta_p = np.zeros((d + 1, 1))
theta_pocket = np.zeros((d + 1, 1))
best_theta = np.zeros((d + 1, 1))
min_err = cal_error(best_theta, X, y)

# Lists to store errors
err_in_p = []
err_out_p = []
err_in_pocket = []
err_out_pocket = []

for iterate in range(iters):
    # Perceptron
    pred = np.sign(X_b @ theta_p)
    misclassified_indices = np.where(pred.squeeze() != y)[0]

    if len(misclassified_indices) > 0:
        # Pick a random misclassified point
        random_index = np.random.choice(misclassified_indices)
        xi = X_b[random_index, :].reshape(-1, 1)
```

```
            yi = y[random_index]
            theta_p = theta_p + yi * xi

    # Pocket
    pred_pocket = np.sign(X_b @ theta_pocket)
    misclassified_indices_pocket = np.where(pred_pocket.squeeze() != y)[0]

    if len(misclassified_indices_pocket) > 0:
        # Pick a random misclassified point
        random_index_pocket = np.random.choice(misclassified_indices_pocket)
        xi_pocket = X_b[random_index_pocket, :].reshape(-1, 1)
        yi_pocket = y[random_index_pocket]
        theta_pocket = theta_pocket + yi_pocket * xi_pocket

        # Check if the new theta is better
        current_err = cal_error(theta_pocket, X, y)
        if current_err < min_err:
            min_err = current_err
            best_theta = theta_pocket.copy()

    # Calculate and store errors
    err_in_p.append(cal_error(theta_p, X, y))
    err_out_p.append(cal_error(theta_p, X_test, y_test))
    err_in_pocket.append(cal_error(best_theta, X, y))
    err_out_pocket.append(cal_error(best_theta, X_test, y_test))


# plot Er_in and Er_out
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(range(iters), err_in_p, label='Perceptron In-sample Error')
plt.plot(range(iters), err_in_pocket, label='Pocket In-sample Error')
plt.xlabel('Iterations')
plt.ylabel('Error')
plt.title('In-sample Error vs. Iterations')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(range(iters), err_out_p, label='Perceptron Out-of-sample Error')
plt.plot(range(iters), err_out_pocket, label='Pocket Out-of-sample Error')
plt.xlabel('Iterations')
plt.ylabel('Error')
plt.title('Out-of-sample Error vs. Iterations')
plt.legend()
plt.tight_layout()
plt.show()


# plot decision boundary
```

```
def plot_decision_boundary(X, y, theta, ax, title):
    # Plot data points
    ax.scatter(X[y==1][:, 0], X[y==1][:, 1], marker='+', label='1')
    ax.scatter(X[y==-1][:, 0], X[y==-1][:, 1], marker='*', label='6')

    # Plot decision boundary
    x1_min, x1_max = ax.get_xlim()
    x1 = np.array([x1_min, x1_max])

    # w0 + w1*x1 + w2*x2 = 0 => x2 = (-w0 - w1*x1) / w2
    w = theta.squeeze()
    if w[2] != 0:
        x2 = (-w[0] - w[1] * x1) / w[2]
        ax.plot(x1, x2, 'y-', label='Decision Boundary')

    ax.set_xlabel('Intensity')
    ax.set_ylabel('Asymmetry')
    ax.set_title(title)
    ax.legend()


fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Plot for 500 data points
plot_num = 500
indices_1 = np.where(y == 1)[0][:plot_num]
indices_6 = np.where(y == -1)[0][:plot_num]
plot_indices = np.concatenate([indices_1, indices_6])
X_plot, y_plot = X[plot_indices], y[plot_indices]

plot_decision_boundary(X_plot, y_plot, theta_p, axes[0], 'Perceptron Decision Boundary')
plot_decision_boundary(X_plot, y_plot, best_theta, axes[1], 'Pocket Decision Boundary')

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