MDS5210 - Homework 1

Problem 1 (a)

- **Supervised learning** trains a model on labeled data, then the model learns to predict the output based on the input data. (i.e., classification, regression and advanced topics such as transformers and language models)
- **Unsupervised learning** trains a model without labeled data, that trying to disclose the underlying patterns or structure without any guidance on the output. (i.e., clustering and dimensionality reduction)

(b)

statement 2) is True that:

if $\mathbf{X} \in \mathbb{R}^{n \times d}$ does not have full column rank, such that $n < d$ (the number of observations is less than the number of features), which means overfitting. (potentially, it can be overcome by SVD or regularization), this implies a infinite many solutions for the least squares problem, which is not unique.

- **statement 1)** - False
  classification is more often used to fit categorical data, as the labels usually are discrete values that represent no real value.
  regression is more often used to fit continuous y that y is real-valued.
- **statement 3)** - False
  the perceptron methods iterate the algorithm until the model converges by resolving the misclassified data points for linear classification. however, if the hypothesis space of the data example is not strictly bipolar, (i.e., the hyper plane can be thick), that it is possible to fit several linear classifiers. hence the perceptron algorithm may not always converge to a unique solution.
- **statement 4)** - False
  the solution of the least squares problem is a maximum likelihood estimator when the noise is Gaussian and the model is linear. (i.e., $\mathcal{L}(\theta)$ can be simplified as $constant - \frac{1}{2\sigma^2}||\mathbf{y} - \mathbf{X}\theta||_2^2$),
  so that $\widehat{\theta}_{\text{MLE}} = \underset{\theta}{\text{argmin}} \, \|\mathbf{y} - \mathbf{X}\theta\|_2^2$ only when $\mathbf{y} = \mathbf{X}\theta + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^{\mathbf{2}})$.

(c)

assign X^TX as $A \in \mathbb{R}^{d\times d}$, A is positive definite if for any non-zero vector $y \in \mathbb{R}^d$, $y^TAy > 0$
matrix X has full column rank means that the columns of the matrix are linearly independent, hence any non-zero vector $y \in \mathbb{R}^d$ can be expressed as a linear combination of the columns of X, such that:

$$y = Xz, \quad \text{for some} z \in \mathbb{R}^n \quad \text{where} \quad z \neq 0 if y \neq 0 \tag{1}$$

and since X^TX is symmetric

$$y^TAy = (Xz)^TA(Xz) = z^TX^TAXz = z^T(X^TX)z \tag{2}$$

because X^TX is invertible and symmetric as X has full column rank, for any non-zero z, Xz is also non-zero as columns of X are linearly independent
this implies X^TX must be symmetric and positive definite, such that: $z^T(X^TX)z \geq 0$ for any non-zero z

(d)

$\mathbf{J}(\mathbf{w}, w_0) = (\mathbf{y} - \mathbf{X}\mathbf{w} - w_0\mathbf{1})^T(\mathbf{y} - \mathbf{X}\mathbf{w} - w_0\mathbf{1})$

- $\mathbf{y}$ – vector of target values
- $\mathbf{w}$ – vector of weights for $\mathbf{X}$
- w_0 – bias term
- $\mathbf{1}$ – vector of ones (same dimension as $\mathbf{y}$)

take the gradient of $\mathbf{J}$ with respect to $\mathbf{w}$ and w_0:

$$\begin{aligned} \frac{\delta \mathbf{J}}{\delta \mathbf{w}} &= -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\mathbf{w} - w_0\mathbf{1}) = 0 \\ \frac{\delta \mathbf{J}}{\delta w_0} &= -2\mathbf{1}^T(\mathbf{y} - \mathbf{X}\mathbf{w} - w_0\mathbf{1}) = 0 \end{aligned} \tag{3}$$

solve the equations, we have:

$$\begin{aligned} \mathbf{w} &= (\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{y} - w_0\mathbf{X}^T\mathbf{1}) \\ w_0 &= \frac{\mathbf{1}^T\mathbf{y} - \mathbf{1}^T\mathbf{X}\mathbf{w}}{\mathbf{1}^T\mathbf{1}} \end{aligned} \tag{4}$$

and this yields the minimizer that:

$$\begin{aligned}
&\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \\
&w_0 = \frac{1}{n}\sum_{i}^{n} n\mathbf{y}_i \\
&\mathbf{J}^*(\mathbf{w}, w_0) = (\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w}) + \frac{1}{n}(\mathbf{1}^T\mathbf{y})^2 - \frac{2}{n}(\mathbf{1}^T\mathbf{y})(\mathbf{y}^T\mathbf{X}\mathbf{w})
\end{aligned} \tag{5}$$

Problem 2 (a)

given the least squares problem that:

$$\min_{\theta\in\mathbb{R}^d} ||\mathbf{X}\theta - \mathbf{y}||_2^2 \tag{6}$$

where $\mathbf{X} \in \mathbb{R}^{n\times d}$, $\theta \in \mathbb{R}^d$, y $\in \mathbb{R}^n$, and $n < d$ with $rank(\mathbf{X}) = n$
the SVD of matrix $\mathbf{X}$ is given by:

$$\mathbf{X} = \mathbf{V}\begin{bmatrix}\mathbf{\Sigma}_1 & \mathbf{0}\end{bmatrix}\begin{bmatrix}\mathbf{U}_1^T \\ \mathbf{U}_2^T\end{bmatrix} = \mathbf{V}\mathbf{\Sigma}_1\mathbf{U}_1^T \tag{7}$$

where:

- $\mathbf{V} \in \mathbb{R}^{n\times n}$ – orthogonal matrix
- $\mathbf{\Sigma}_1 \in \mathbb{R}^{n\times n}$ – diagonal matrix with non-zero singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n > 0$
- $\mathbf{U}_1 \in \mathbb{R}^{n\times d}$ – semi-orthogonal matrix that $\mathbf{U}_1^T\mathbf{U}_1 = \mathbf{I}$
- $\mathbf{U}_2 \in \mathbb{R}^{(d-n)\times n}$ – semi-orthogonal matrix that $\mathbf{U}_2^T\mathbf{U}_2 = \mathbf{I}$

let $\mathbf{A} := \mathbf{V}\mathbf{\Sigma_1}$ be a square matrix of full rank, define $\mathbf{z} := \mathbf{U}_1^T\theta$,
then we have:

$$\min_{\mathbf{z}} ||\mathbf{A}\mathbf{z} - \mathbf{y}||_2^2 \tag{8}$$

by FOC,

$$\mathbf{A}^T(\mathbf{A}\mathbf{z} - \mathbf{y}) = 0, \quad \text{solve for} \quad \mathbf{z} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y} \tag{9}$$

since $\mathbf{z} = \mathbf{U}_1^T\theta$, we can solve for:

$$\theta = \mathbf{U}_1(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y} \tag{10}$$

thus the solution set can be expressed as:

$$\theta = \mathbf{U}_1\mathbf{\Sigma}_1^{-1}\mathbf{V}^T\mathbf{y} + \mathbf{U}_2\mathbf{s} | \mathbf{s} \in \mathbb{R}^{d-n} \tag{11}$$

(because the fact that $\mathbf{U}_1^T\mathbf{U}_2 = 0$, the solutions remain in null space of $\mathbf{X}^T$)
which shows that the least squares problem has infinite many solutions in this case

and the optimal function value is given by:

$$\begin{aligned} \min_{\mathbf{z}}||\mathbf{Az} - \mathbf{y}||_2^2 =& ||\mathbf{V\Sigma_1 U_1}^T(\mathbf{U}_1\mathbf{\Sigma}_1^{-1}\mathbf{V}^T\mathbf{y} + \mathbf{U}_2\mathbf{s}) - \mathbf{y}||_2^2 \\ &= ||\mathbf{y} - \mathbf{y}||_2^2 = 0 \end{aligned} \tag{12}$$

hence the optimal function value is 0

(b)

the regularization to solve the overfitting following:

$$\min_{\theta\in\mathbb{R}^d}||\mathbf{X}\theta - \mathbf{y}||_2^2 + \lambda||\theta||_2^2 \tag{13}$$

rewrite the objective function by expanding the norms:

$$(\mathbf{X}\theta - \mathbf{y})^\top(\mathbf{X}\theta - \mathbf{y}) + \lambda\theta^\top\theta \tag{14}$$

take the derivative with respect to θ and set it to zero:

$$\frac{\delta}{\delta\theta} = 2\mathbf{X}^\top(\mathbf{X}\theta - \mathbf{y}) + 2\lambda\theta = 0 \tag{15}$$

rearrange to solve for θ:

$$\theta = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y} \tag{16}$$

because $\mathbf{X}^\top\mathbf{X}$ is positive definite and symmetric, while $\lambda > 0$, so that $\lambda\mathbf{I} > 0$ is also positive definite and symmetric, the inverse exists and the solution is unique

Problem 3 (a)

derive the learning problem for estimating θ^* under the assumption that ϵ follows a Laplace distribution (i.e., $\epsilon_i \sim_{i.i.d.} \mathbf{L}(0, b), \forall i = 1, 2, \ldots, n$)

the probability density function of Laplace distribution is given by:

$$p(\epsilon_i) = \frac{1}{2b}\exp(-\frac{|\epsilon_i - 0|}{b}) \tag{17}$$

for some $b > 0$ as the scale parameter

the log-likelihood function $\ell(\theta)$is given by:

$$\ell(\theta) = \sum_{i=1}^{n} \log p(y_i - \mathbf{x}_i^T \theta) \tag{18}$$

substitute the Laplace distribution into the log-likelihood function, we have:

$$\begin{aligned} \ell(\theta) &= \sum_{i=1}^{n} \log[\frac{1}{2b} \exp(-\frac{|y_i - \mathbf{x}_i^T \theta|}{b})] \\ &= -n \log(2b) - \frac{1}{b} \sum_{i=1}^{n} |y_i - \mathbf{x}_i^T \theta| \end{aligned} \tag{19}$$

the maximum likelihood estimator $\widehat{\theta}_{\mathbf{ML}}$ (equivalent to minimizing the negative log-likelihood) is given by:

$$\widehat{\theta}_{\mathbf{ML}} = \text{argmin}_{\theta \in \mathbb{R}^d} - (\ell(\theta)) = \text{argmin}_{\theta \in \mathbb{R}^d} (n \log(2b) + \frac{1}{b} \sum_{i=1}^{n} |y_i - \mathbf{x}_i^T \theta|) \tag{20}$$

the minimization of the sum of absolute values can be decomposite as the L1-norm or least absolute deviations (LAD) problem, which can be solved by linear programming or subgradient descent

the learning problem for estimating θ^* under the assumption that ϵ follows a Laplace distribution is to minimize the L1-norm of the residuals:

$$\widehat{\theta}_{\mathbf{ML}} = \text{argmin}_{\theta \in \mathbb{R}^d} \left[\frac{1}{b} \sum_{i=1}^{n} |y_i - \mathbf{x}_i^T \theta| \right] \tag{21}$$

(b)

Huber smoothing approximate the L1-norm minimization in robust linear regression, given by:

$$\begin{aligned} h_\mu(z) &= \begin{cases} |z|, & \text{if } |z| \geq \mu \\ \frac{z^2}{2\mu} + \frac{\mu}{2} & \text{if } |z| \leq \mu \end{cases} \\ \mathbf{H}_\mu(\mathbf{z}) &= \sum_{j=1}^{n} h_\mu(z_i) \end{aligned} \tag{22}$$

to find the gradient $\nabla \mathbf{L}(\theta)$, we need to find derivatives of $H_\mu(\mathbf{X}\theta - \mathbf{y})$ with respect to θ, accroding to the Huber function $h_\mu(z)$, we have:

- for $|z| \geq \mu : h_\mu(z) = |z|$, $h'_\mu(z) = \frac{\delta h_\mu(z)}{\delta z} = \text{sign}(z)$
- for $|z| \leq \mu : h_\mu(z) = \frac{z^2}{2\mu} + \frac{\mu}{2}$, $h'_\mu(z) = \frac{\delta h_\mu(z)}{\delta z} = \frac{z}{\mu}$

the gradient $\nabla \mathbf{L}(\theta)$ is the partial derivative applying the chain rule:

$$\nabla \mathbf{L}(\theta) = \frac{\delta \mathbf{L}(\theta)}{\delta \theta_j} = \sum_{j=1}^{n} \begin{cases} X_j \text{sign}(X_j\theta - y_j), & \text{if } |X_j\theta - y_j| \geq \mu \\ X_j \frac{\mathbf{X_j}\theta - \mathbf{y_j}}{\mu}, & \text{if } |X_j\theta - y_j| \leq \mu \end{cases} \tag{23}$$

where $\mathbf{X}_i$ is the i-th row of the matrix $\mathbf{X}$, and θ_j is the j-th component of **theta**

(c)

(1) gradient descent for minimizing $\mathbf{L}(\theta)$ such that data set is generated by the linear model

$$\mathbf{y} = \mathbf{X}\theta^* + \epsilon_1 + \epsilon_2 \tag{24}$$

where:

- ϵ_1 follows i.i.d. Gaussian distribution,
- ϵ_2 is a sparse vector containing outliers

inputs are:

- Data $\mathbf{X}, \mathbf{y}$
- initial parameter θ_0
- smoothing factor μ
- iteration number $\mathbf{T}$
- step size α

the loss function $\mathbf{L}(\theta)$ is typically the sum of squared errors that:

$$\mathbf{L}(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x^i) - y^i)^2 \tag{25}$$

(n is the number of observations in training samples, $h_\theta(x)$ is the hypothesis function)
the gradient of the loss function $\nabla \mathbf{f}(\theta_\mathbf{k})$ is with respect to θ, and the update rule:

$$\theta_{\mathbf{k+1}} = \theta_\mathbf{k} - \alpha \mathbf{f}(\theta_\mathbf{k}) \tag{26}$$

the least squares estimate $\widehat{\theta}_{\mathbf{LS}}$ is such that θ minimizes the sum of the squares of the residuals between the observed and predicted values. which is given by:

$$\widehat{\theta}_{\mathbf{LS}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{27}$$

the Euclidean norm of the difference between $\widehat{\theta}_{\mathbf{LS}}$ and θ^* is that:

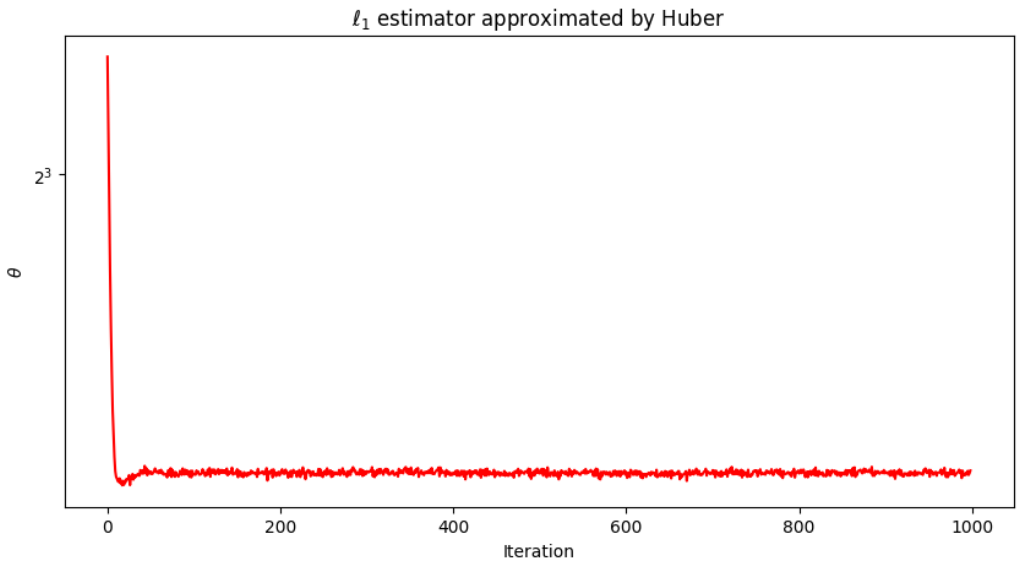
$$\left\|\widehat{\theta}_{\mathbf{LS}} - \theta^*\right\|_2 = \sqrt{(\widehat{\theta}_{LS} - \theta^*)^T(\widehat{\theta}_{LS} - \theta^*)} \tag{28}$$

(2)

[4]:
```
import numpy as np
import matplotlib.pyplot as plt

d = 50   #feature dimension

X = np.load('data/X.npy')
y = np.load('data/y.npy')
print ("data shape: ", X.shape, y.shape)

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

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

theta_hat = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(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

# define the gradient of the l1 loss
def gradient(X, theta, y, mu):
    grad = np.zeros(X.shape)
    for i in range(len(X)):
        if abs(X[i,:].T @ theta - y[i]) > mu:
            temp = np.sign(X[i,:].T @ theta - y[i]) * X[i,:]
            # print(list(temp))
            grad[i,:] = list(temp)
        else:
            temp = (X[i,:].T @ theta - y[i])/mu * X[i,:]
            grad[i,:] = list(temp)
    return grad.sum(0).reshape(50,1)
```

```
# 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))

    grad = gradient(X, theta, y, mu)

    #gradient descent update
    theta = theta - alpha * grad

#######   plot the figure   #########
plt.figure(figsize=(10,5))
plt.yscale('log',base=2)
plt.plot(Error_huber, 'r-')
plt.title(r'$\ell_1$ estimator approximated by Huber')
plt.ylabel(r'$\theta$')                # set the label for the y axis
plt.xlabel('Iteration')               # set the label for the x axis
plt.show()
```

```
data shape:  (1000, 50) (1000, 1)
Estimator approximated by LS: 59.51363648298142
```


ℓ_1 estimator approximated by Huber
2^3
θ
0
200
400
600
800
1000
Iteration

Problem 4 (a)
θ^* is the corresponding classifier that correctly separates the data, and show:

$$\rho = \min_{1 \leq i \leq n} \mathbf{y}_i (\theta^{*T} \mathbf{x}_i) \tag{29}$$

ρ is the smallest margin by which the optimal hypothesis correctly classifies the data.
the linear separability implies that the value $\mathbf{y}_i(\theta^{*T}\mathbf{x_i})$ for any $\mathbf{x}_i$, is positive if $\mathbf{y}_i = 1$, and negative if $\mathbf{y}_i = -1$, and the optimal hyperplane maximize the margin between the two classes
this implies the existence of positive margins between the classes that ρ, being the minimum of these margins, must be strictly positive
if ρ is non-positive, it means that there exist data point that is either on the hyperplane, or on the wrong classification side, which is contradicted to the assumption of optimality linear separation θ^*

(b)

the perceptron updates as:

$$\theta_k = \theta_{k-1} + y_{k-1}\mathbf{x}_{k-1} \quad \forall k = 1, 2, \ldots \tag{30}$$

where $(\mathbf{x}_{k-1}, y_{k-1})$ is the wrongly classified data point at the $k-1$-th iteration
consider the dot product of θ_k with θ^*:

$$\theta_k^T \theta^* = \theta_{k-1}^T \theta^* + y_{k-1}\mathbf{x}_{k-1}^T \theta^* \tag{31}$$

because $(\mathbf{x}_{k-1}, y_{k-1})$ is wrongly classified,

$$y_{k-1}(\theta_{k-1}^T \mathbf{x}_{k-1}) \leq 0 \tag{32}$$

$$y_{k-1} x_{k-1}^\top \theta^* \geq y_{k-1} \left(\theta^{*\top} x_{k-1} \right) = y_{k-1} \left(y_{k-1} \right) \rho = \rho \tag{33}$$

Here, we used the fact that $y_{k-1} \left(\theta^{*\top} x_{k-1} \right) = y_{k-1} \left(y_{k-1} \right) \rho$ and $y_{k-1} \left(\theta^{*\top} x_{k-1} \right)$ must be at least ρ in magnitude but with the opposite sign:

$$\theta_k^T \theta^* = \theta_{k-1}^T \theta^* + y_{k-1}\mathbf{x}_{k-1}^T \theta^* \geq \theta_{k-1}^T \theta^* + \rho \tag{34}$$

by induction, starting from $\theta_0 = 0$, so that $\theta_0^T \theta^* = 0$ and:

$$\theta_k^T \theta^* \geq k\rho \quad \forall k = 1, 2, \ldots \tag{35}$$

(c)

the squared norm of updated weight vector θ_k:

$$\begin{aligned} \|\theta_k\|^2 &= \|\theta_{k-1} + y_{k-1}\mathbf{x}_{k-1}\|^2 \\ &= \|\theta_{k-1}\|^2 + 2y_{k-1}\theta_{k-1}^T\mathbf{x}_{k-1} + y_{k-1}^2\|\mathbf{x}_{k-1}\|^2 \end{aligned} \tag{36}$$

because $(\mathbf{x}_{k-1}, y_{k-1})$ is wrongly classified by θ_{k-1}:

$$\begin{aligned} y_{k-1}\theta_{k-1}^T\mathbf{x}_{k-1} &\leq 0 \\ 2y_{k-1}\theta_{k-1}^T\mathbf{x}_{k-1} &\leq 0 \end{aligned} \tag{37}$$

thus, substituting the above inequality into the squared norm of θ_k:

$$\|\theta_k\|^2 \leq \|\theta_{k-1}\|^2 + y_{k-1}^2\|\mathbf{x}_{k-1}\|^2 \tag{38}$$

and since y_{k-1} is either 1 or -1, $y_{k-1}^2 = 1$:

$$\|\theta_k\|^2 \leq \|\theta_{k-1}\|^2 + \|\mathbf{x}_{k-1}\|^2 \tag{39}$$

(d)

following the results from **(c)**, we know by induction:

$$\|\theta_k\|^2 \leq \sum_{i=1}^{k}\|\mathbf{x}_{i-1}\|^2 \tag{40}$$

and given the fact where:

$$R = \max_{1\leq i\leq n}\|\mathbf{x}_i\| \quad \text{, and} \quad \|\mathbf{x}_i\| \leq R \tag{41}$$

$$\|\theta_k\|^2 \leq \sum_{i=1}^{k} R^2 = kR^2 \tag{42}$$

(e)

given the shown: $\theta_k^T\theta^* \geq k\rho$ and $\|\theta_k\|^2 \leq kR^2$
using the fact of triangle inequality (with the hint):

$$\theta_k^T\theta^* \leq \|\theta_k\|\,\|\theta^*\| \tag{43}$$

hence, we have:

$$\begin{aligned} \|\theta_k\| &\geq \frac{\theta_k^T\theta^*}{\|\theta^*\|} \geq \frac{\sqrt{k}\rho}{R} \\ \|\theta_k\|^2 &\geq \frac{k\rho^2}{R^2} \end{aligned} \tag{44}$$

combing the facts that: $kR^2 \geq \frac{k\rho^2}{R^2}$, solving for k, we have:

$$k \leq \frac{R^2 \left\|\theta^*\right\|^2}{\rho^2} \tag{45}$$

which implies that the perceptron algorithm must stop within at most $\frac{R^2\|\theta^*\|^2}{\rho^2}$ iterations

Problem 5 (1)

the perceptron for pocket algorithm updates as:

$$\begin{aligned} &\theta_{k+1} = \theta_k + y_k \mathbf{x}_k \\ &\text{if} \quad Er_{in}(\theta_{k+1}) < Er_{in}(\widehat{\theta}) \\ &\text{then} \quad \widehat{\theta} = \theta_{k+1} \end{aligned} \tag{46}$$

which is a natural variation for dealing with unstability, preserving the prameter θ with the best in-sample error during the update and ensures monotonically decreasing in-sample error, where:

- θ — weights (i.e., θ_0 is the initial weight)
- $(\mathbf{x}_k, y_k)$ — wrongly clasified sample from dataset

the python codes p5.py is shown below:

[1]:
```
import scipy.io
import matplotlib.pyplot as plt
import numpy as np
# from sipbuild.generator.outputs.xml import output_xml


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), 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), 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.
    """
    y[y==-1] = 5 #########
    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):
        if len(feat) > 0:  # Ensure there are features to plot
            ax.scatter(*feat[:plot_num].T, marker=marks[i],␣
↪color=colors[i], label=str(i))
    ax.legend(loc='upper right')
    plt.xlabel('intensity')
    plt.ylabel('symmetry')
    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
    """
    out = X @ theta - thres
    pred = np.sign(out)
    err = np.mean(pred.squeeze()!=y)
    return err


# prepare data
train_data, train_label = load_mat('train_data.mat') # train_data: (7291,␣
↪16, 16), train_label: (7291, )
test_data, test_label = load_mat('test_data.mat') # test_data: (2007, 16,␣
↪16), train_label: (2007, )
```

```
cls_A, cls_B = 1, 5
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)


# train
iters = 2000
d = 2
num_sample = X.shape[0]
threshold = 1e-4
theta = np.zeros((d, 1))
```

[2]:
```
# perceptron
def update(theta, X, y, thres):
    out = X @ theta - thres
    pred = np.sign(out)
    wrong_X = X[pred.squeeze() != y]
    wrong_y = y[pred.squeeze() != y]
    wrong_data = np.hstack([wrong_X, wrong_y.reshape(len(wrong_y), 1)])
    chosen_index = np.random.choice(range(len(wrong_data)), size=1,␣
↪replace=False)
    chosen_data = wrong_data[chosen_index, :]
    theta_new = theta + chosen_data[:, :-1].T * chosen_data[:, -1]
    return theta_new

def cal_error(theta, X, y, thres=1e-4):
    out = X @ theta - thres
    pred = np.sign(out)
    err = np.mean(pred.squeeze() != y)
    return err

def plot_decision_boundary(theta, X, y, ax):
    x_vals = np.linspace(min(X[:, 0]), max(X[:, 0]), 100)
    decision_boundary = -(theta[0, 0] / theta[1, 0]) * x_vals
    ax.plot(x_vals, decision_boundary, color='red', label='Decision␣
↪Boundary')
    ax.legend()
```

```
# Initialize lists to store error values
in_errors = []
out_errors = []

# Training loop
for iterate in range(iters):
    theta = update(theta, X, y, threshold)
    in_error = cal_error(theta, X, y, threshold)
    out_error = cal_error(theta, X_test, y_test, threshold)

    # Append errors to the lists
    in_errors.append(in_error)
    out_errors.append(out_error)

# Plot Er_in and Er_out
plt.plot(range(iters), in_errors, label="in_error")
plt.plot(range(iters), out_errors, label="out_error")
plt.title("in_error vs out_error of perceptron")
plt.legend()
plt.show()

# plot decision boundary
fig, ax = plt.subplots(1, 2)
plot_feature(X, y, 500, ax=ax[0], classes=[1, 5])
plot_decision_boundary(theta, X, y, ax=ax[0])
ax[0].set_title('Perceptron - Train')

plot_feature(X_test, y_test, 500, ax=ax[1], classes=[1, 5])
plot_decision_boundary(theta, X_test, y_test, ax=ax[1])
ax[1].set_title('Perceptron - Test')
plt.show()
```

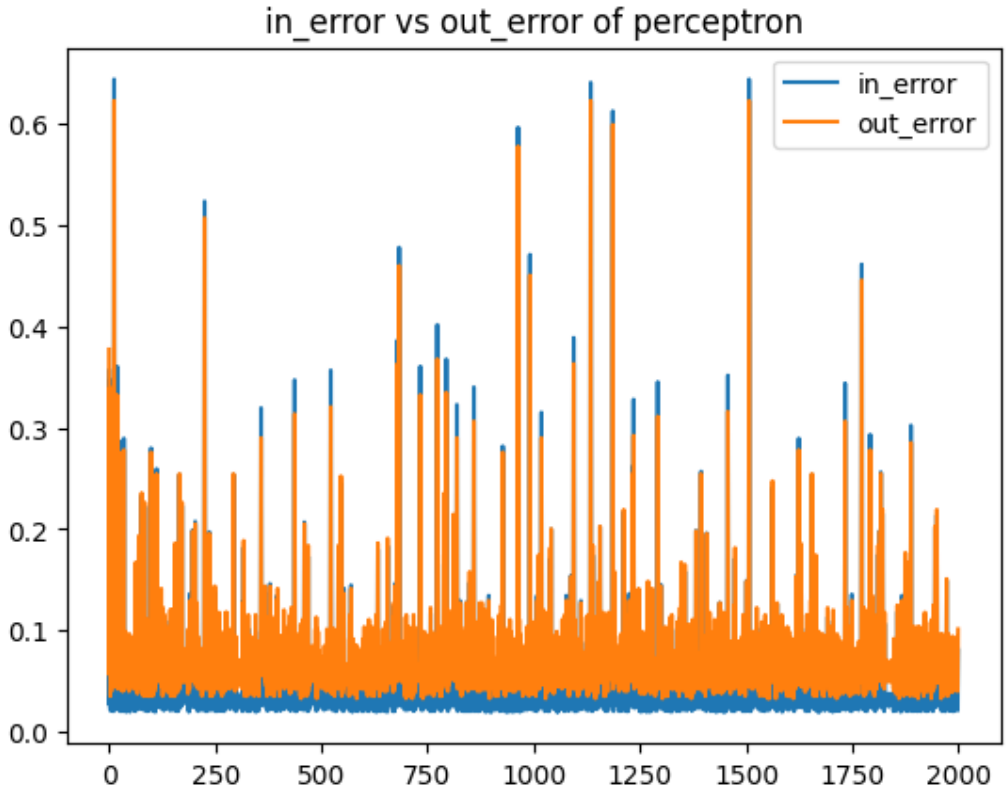
[3]:
```
train_data, train_label = load_mat('train_data.mat')
test_data, test_label = load_mat('test_data.mat')

cls_A, cls_B = 1, 5
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)
```

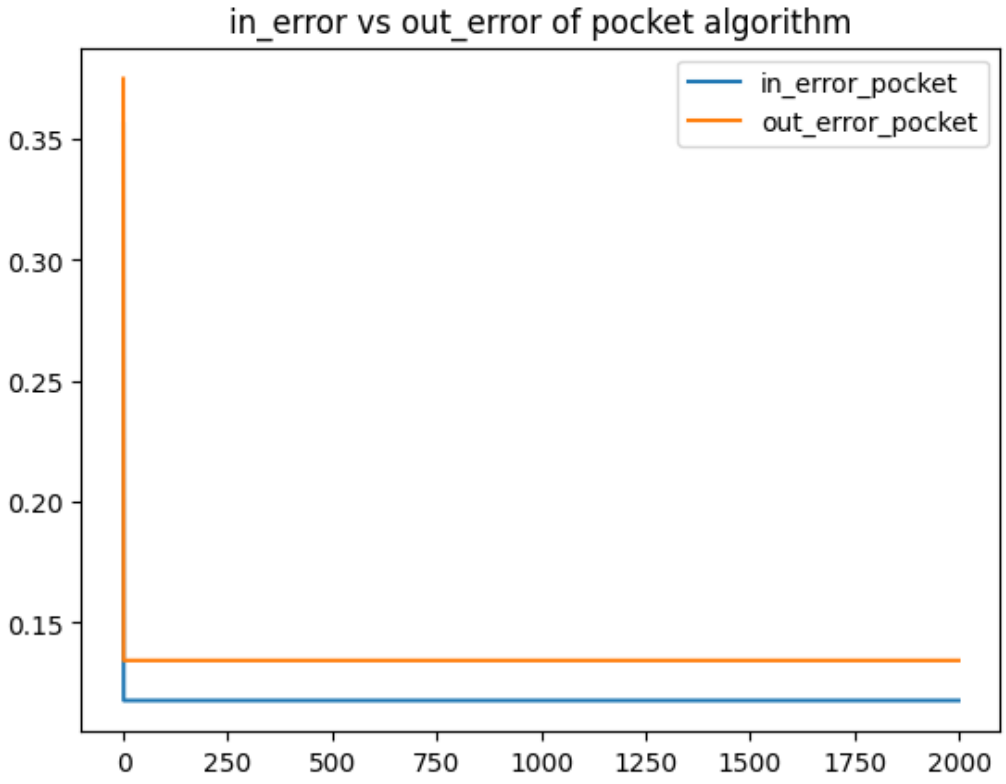
```
iters = 2000
d = 2
num_sample = X.shape[0]
threshold = 1e-4
theta = np.zeros((d, 1))

# Pocket algorithm update function
def update_pocket(theta, X, y, thres, pocket_size):
    out = X @ theta - thres
    pred = np.sign(out)
    wrong_X = X[pred.squeeze() != y]
    wrong_y = y[pred.squeeze() != y]
    wrong_data = np.hstack([wrong_X, wrong_y.reshape(len(wrong_y), 1)])

    # Select misclassified samples
    chosen_indices = np.random.choice(range(len(wrong_data)),␣
↪size=pocket_size, replace=False)
    chosen_data = wrong_data[chosen_indices, :]

    # Update weights considering samples in the pocket
    for data_point in chosen_data:
        theta_new = theta + data_point[:-1].reshape(-1, 1) *␣
↪(data_point[-1])

    return theta_new

# calculate the error for pocket algorithm
def cal_error_pocket(theta, X, y, thres=1e-4):
    out = X @ theta - thres
    pred = np.sign(out)
    err = np.mean(pred.squeeze() != y)
    return err

# plot the decision boundary for pocket algorithm
def plot_decision_boundary_pocket(theta, X, y, ax):
    x_vals = np.linspace(min(X[:, 0]), max(X[:, 0]), 100)
    decision_boundary = -(theta[0, 0] / theta[1, 0]) * x_vals
    ax.plot(x_vals, decision_boundary, color='blue', label='Pocket␣
↪Decision Boundary')
```

```
    ax.legend()

# Initialize lists to store error values
in_errors = []
out_errors = []

# Training loop
pocket_num = 1
for iterate in range(iters):
    theta_pocket = update_pocket(theta, X, y, threshold, pocket_num)
    in_error = cal_error_pocket(theta_pocket, X, y, threshold)
    if len(in_errors) == 0 or in_error < in_errors[-1]:
        theta = theta_pocket
        in_errors.append(in_error)
    else:
        in_errors.append(in_errors[-1])
    out_error = cal_error_pocket(theta, X_test, y_test, threshold)
    out_errors.append(out_error)


# Plot Er_in and Er_out
plt.plot(range(iters), in_errors, label="in_error_pocket")
plt.plot(range(iters), out_errors, label="out_error_pocket")
plt.title("in_error vs out_error of pocket algorithm")
plt.legend()
plt.show()

# plot decision boundary
fig, ax = plt.subplots(1, 2)
plot_feature(X, y, 500, ax=ax[0], classes=[1, 5])
plot_decision_boundary_pocket(theta, X, y, ax=ax[0])
ax[0].set_title('Pocket Algorithm - Train')

plot_feature(X_test, y_test, 500, ax=ax[1], classes=[1, 5])
plot_decision_boundary_pocket(theta, X_test, y_test, ax=ax[1])
ax[1].set_title('Pocket Algorithm - Test')
plt.show()
```

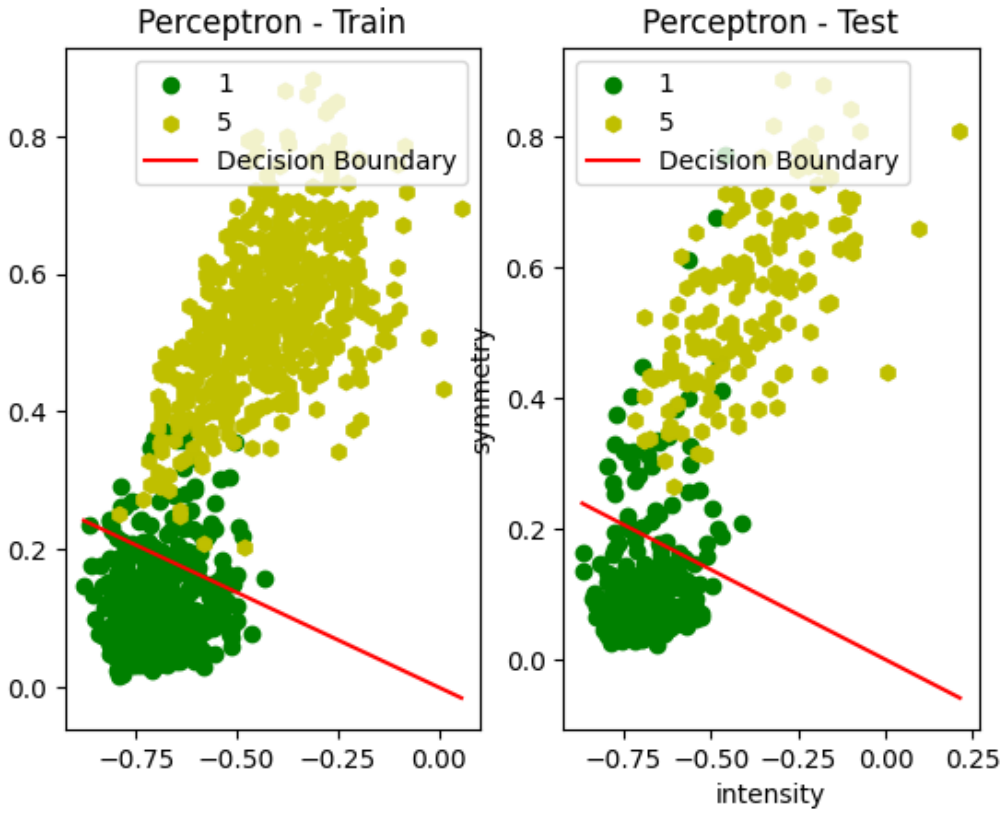
(2)

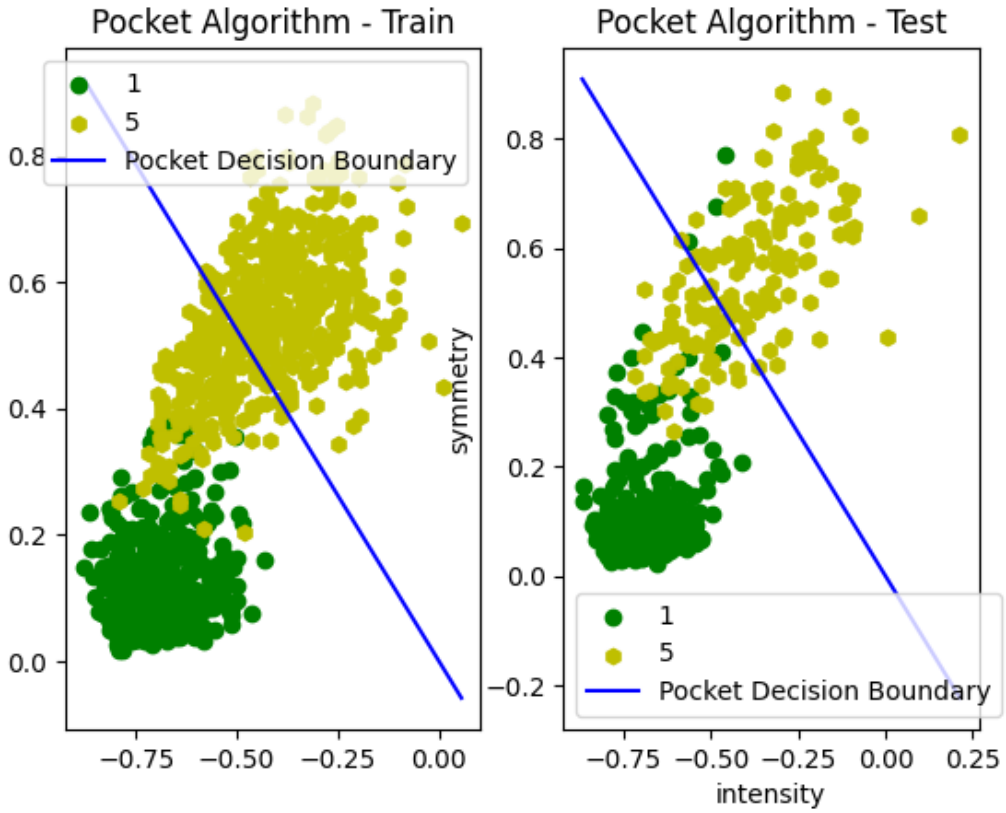
In the perceptron model, we observe that both the in-sample error $\mathbf{Er_{in}}$ and the out-sample error $\mathbf{Er_{out}}$ exhibit significant fluctuations and fail to converge at an ideal separation boundary. In contrast, the pocket algorithm is capable of identifying a minimal in-sample error and rapidly reaching a satisfactory decision boundary. However, it is also observed that neither the in-sample nor out-sample error of the pocket method decrease after the initial few iterations, suggesting that the algorithm might be prone to getting stuck in local optima.
the plots are shown as following:





(3)
plots for the decision boundary of the perceptron and pocket algorithm are shown as following:


Perceptron - Train
Perceptron - Test
1
5
Decision Boundary
symmetry
intensity
0.0
0.2
0.4
0.6
0.8
−0.75
−0.50
−0.25
0.00
0.25


Pocket Algorithm - Train
Pocket Algorithm - Test
1
5
Pocket Decision Boundary
symmetry
intensity
0.8
0.6
0.4
0.2
0.0
−0.2
−0.75
−0.50
−0.25
0.00
0.25