ML HW4 R10922A16 蔡家豪

1.

optimal w can be computed from.

$$\frac{\partial\left(\frac{1}{N}\sum_{n=1}^{N}(w\cdot x_n - y_n)^2 + \frac{\lambda}{N}w^2\right)}{\partial w} = 0 \Rightarrow \frac{1}{N}\sum_{n=1}^{N}(w\cdot x_n - y_n)' \times (w\cdot x_n - y_n) + \frac{2\lambda}{N}w$$

$$\Rightarrow \sum_{n=1}^{N} x_n\cdot(w\cdot x_n - y_n) + \lambda w = 0$$

$$\Rightarrow \sum_{n=1}^{N}(wx_n^2 - x_n y_n) + \lambda w = 0$$

$$\Rightarrow w\left(\lambda + \sum_{n=1}^{N}x_n^2\right) = \sum_{n=1}^{N}x_n y_n$$

$$\Rightarrow w = \frac{\sum_{n=1}^{N}x_n y_n}{\sum_{n=1}^{N}x_n^2 + \lambda}$$

since $C = (w)^2 \Rightarrow C = \left(\frac{\sum_{n=1}^{N}x_n y_n}{\sum_{n=1}^{N}x_n^2 + \lambda}\right)^2$ [a] //

2. $\phi(x) = T^{-1}x$ (consider $\mu = 0$)

$$\Rightarrow \min_{\tilde{w}\in R^{d+1}} \frac{1}{N}\sum_{n=1}^{N}(\tilde{w}^T T^{-1}x_n - y_n)^2 + \frac{\lambda}{N}(\tilde{w}^T\tilde{w})$$

$$\min_{w\in R^{d+1}} \frac{1}{N}\sum_{n=1}^{N}(w^T x_n - y_n)^2 + \frac{\lambda}{N}\Omega(w)$$

consider this 2 eq. the difference is $\tilde{w}^T T^{-1}x$ and $w^T x_n$, this 2 needs to be. the same to make this 2 question equivalent.

$\Rightarrow$ we have $\tilde{w}^T T^{-1}x_n = w^T x_n \Rightarrow \tilde{w}^T = w^T T \Rightarrow \tilde{w} = T^T w$

$= Tw$ (T is diagonal)

Replace the $\tilde{w}^T w$ in $\frac{\lambda}{N}(\tilde{w}^T\tilde{w}) \Rightarrow \frac{\lambda}{N}(Tw)^T(Tw) = \frac{\lambda}{N}\Omega(w)$

$\Rightarrow \frac{\lambda}{N}(w^T T^T T w) = \frac{\lambda}{N}(\Omega w)$,

$\Rightarrow \Omega(w) = w^T T^2 w$ ($T^T = T$ when T is diagonal). // [b]

3. $D_{KL}(P_w \| P_h) = \underbrace{\frac{1}{2}\ln\frac{\frac{1}{2}}{1+\exp(-w^T x)}}_{y=+1} + \underbrace{\frac{1}{2}\ln\frac{\frac{1}{2}}{1+\exp(w^T x)}}_{y=-1}$.

if we place y back to it $= \ln\frac{\frac{1}{2}}{1+\exp(-yw^T x)}$

we have error function

$$err(w, x, y) = \ln(1+\ln(-yw^T x))$$

and put y back to $err_{smooth}(w, x, +1 \text{ or } -1)$

because $err_{smooth}(w, x, y) = (1-\frac{\partial}{2})\ln(1+\exp(-yw^T x)) + \frac{\partial}{2}\ln(1+\exp(yw^T x))$.

because $\min_w \frac{1}{N}\sum_{n=1}^{N} err_{smooth}(w, x, y)$ is equivalent to

$$\min_w \frac{1}{N} err(w, x, y) + \frac{\lambda}{N}\sum_{n=1}^{N}\Omega(w, x) \quad \text{when solving.}$$

We rewrite err_{smooth}

put a - here

$$err_{smooth}(w, x, y) = \underbrace{\ln(1+\exp(-yw^T x))}_{err} - \frac{\partial}{2}(\ln(1+\exp(-yw^T x)) + \underbrace{\frac{\partial}{2}\ln(1+\exp(yw^T x))}_{-\frac{\partial}{2}\ln(1+\exp(-yw^T x))}$$

$$= err + (-\partial\ln(1+\exp(-yw^T x)))$$

which means when solving these 2 eq. the err pace is the same

we can focus on $\frac{\partial(\lambda\sum_{n=1}^{N}\Omega(w, x))}{\partial w} = \frac{\partial(\partial\ln(1+\exp(-yw^T x)))}{\partial w}$

since the scaling will not affect the result.

we have $\int(\Omega(w, x))' = \int(-\ln(1+\exp(-bw^T x)))'$

$\Rightarrow \Omega(w, x) = -\ln(1+\exp(-yw^T x)) + C$.

we can take $\ln\frac{1}{2}$ from C

$$\Omega(w, x) = \ln\frac{\frac{1}{2}}{1+\exp(-yw^T x)} = \frac{1}{2}\ln\frac{\frac{1}{2}}{1+\exp(+w^T x)} + \frac{1}{2}\ln\frac{\frac{1}{2}}{1+\exp(w^T x)} = D_{KL}(P_w \| P_h)$$

(a) ✱

4. $E_{loocv} = \frac{1}{3}\left[\left(\frac{y_2+y_3}{2} - y_1\right)^2 + \left(\frac{y_1+y_3}{2} - y_2\right)^2 + \left(\frac{y_2+y_3}{2} - 1\right)^2\right] \leq \frac{1}{3}$

the area of this funcn will be $\doteq 2.4184$ (solve by geogebra)

due to the $y_n \in [0,2]$ (2×2 □)

we divide this by 4 will be 0.6046

is almost the same as [b] $\frac{\pi}{2\sqrt{3}}$ #.

5. Since the generate sample is iid, the covariance should be zero.

$\Rightarrow$ Variance [Eval(h)] = Variance $\left[\frac{1}{k}\Sigma err(x_i, y_{ji})\right]$

$= \frac{1}{k^2}$ Variance $[\Sigma err(x_i, y_1)] = \frac{1}{k^2}$ Variance [K err(h(x), f).]

$= \frac{k}{k^2}$ Variance [err(h(x), f)] $= \boxed{\frac{1}{k}}$ Variance [err(h(x), f)] #

[d] *

6. Before we selece the validation sample, the negative and positive samples are balanced

after we pick 1 validation sample from them. if we pick a positive one. the majority will become negative. if negative one, majority become positive,

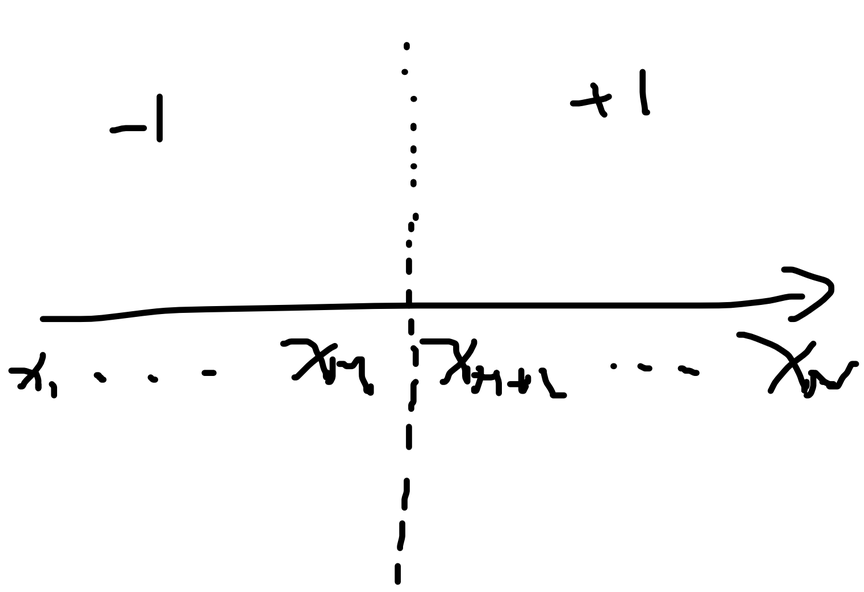
which means the prediction will always wrong since the classifier always predict the majority $\Rightarrow E_{loocv}(A_{majority}) = 1$ [d] #

8. this is a linear seperable "1D" example dataset

with $x_1 \le x_2 < x_3 \cdots < x_M < x_{M+1} \cdots \le x_N$

and $y_n = -1$ for $n = 1, 2, \cdots M$, $y_n = +1$ for $n = M+1, \cdots, N$

which means we can plot the following figure.



We can see the perception should be between x_M and x_{M+1}

which means the margin will between

$\frac{1}{2}(x_{M+1} - x_M)$ to 0

the largese margin

$= \frac{1}{2}(x_{M+1} - x_M)$ //

[e] //

7.

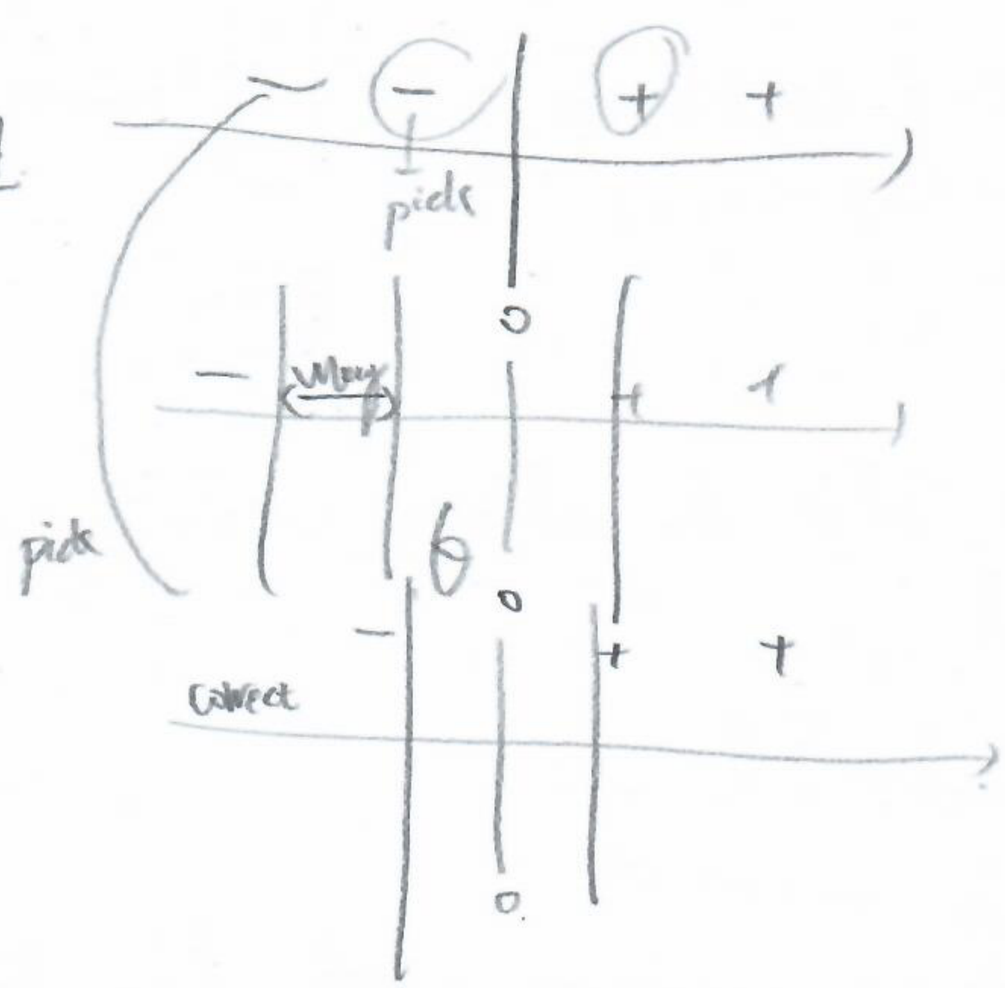
if we have 4 samples. 2 pos and 2 neg

the only possibility we have a leave-one-out error is the larger negative and the smaller positive sample (closest to 0).

cuz. if we pick other sample to evaluation.
the hypothesis is limited to the 2 closest to 0, which will always make the prediction wrong.

if we pick the 2 sample closest to 0, which will make your hypothesis set bigger and reach the second closest pos/neg sample.
this condition will have some poribility to make wrong prediction on the 2 sample closer to 0.

so the ans is 2/N (C) //



19. put the 4 samples into constrains

$\Rightarrow (x_1, 1) \Rightarrow w_1 \cdot 0 + w_2 \cdot 0 + b \geq 1$

$\Rightarrow (x_2, -1) \Rightarrow w_1 \cdot 2 + w_2 \cdot 0 + b \leq -1+2b$

$\Rightarrow (x_3, 1) \Rightarrow w_1 \cdot -1 + w_2 \cdot 0 + b \geq 1$

$\Rightarrow (x_4, 1) \Rightarrow w_1 \cdot 0 + w_2 \cdot 0 + b \geq 1$

$\Rightarrow \begin{cases} b \geq 1 & (b=1) \\ 4w_2 + b \geq 1 \\ 2w_1 + b \leq -1+2b \\ -w_1 + b \geq 1 \end{cases} \Rightarrow (4w_2 \geq 0) \Rightarrow (w_1 \leq \frac{-1+2}{2}$ since $b=1$)

$\Rightarrow w = (\frac{-1+2}{2}, 0), b=1$ // (e) //

10. $\alpha = 1$. $b = 0$.

$$h_{\alpha,b}(x) = \text{sign}\left(\sum_{n=1}^{N} y_n \alpha_n K(x_n, x) + b\right)$$

$$\Rightarrow \hat{h}(x) = \text{sign}\left(\sum_{n=1}^{N} y_n K(x_n, x)\right)$$

since we know $\|x_n - x_m\| \geq \varepsilon$

the kernel will be $K(x, x') \leq \exp(-\gamma \varepsilon^2) = \text{some fixed value } \delta$

$$\Rightarrow \hat{h}(x) = \text{sign}\left(\sum_{n=1}^{N} y_n \exp(-\gamma\varepsilon^2)\right)$$

if we want to make sure the $E_{in}(\hat{h}) = 0$.

we have the following relation.

$$\Rightarrow \sum_{m=1}^{M} \text{sign}\left(\sum_{n=1}^{N} y_n \frac{\exp(-\gamma\varepsilon^2)}{\delta}\right) - y_m = 0$$ this will make sure the program always correct.

if we consider a specific case $y_m = +1$.

we need the sign function ≥ 0.

So, $\sum_{n=1}^{N} y_n \delta \geq 0$, since we don't want to consider itself.

we can rewrite this into the following form

$$\sum_{n \neq m}^{N} y_n \delta + \underbrace{y_m \exp(0)}_{\text{distance to itself}} \geq 0.$$

(y_m: $+1$, $+1$)

$$\Rightarrow \sum_{n \neq m}^{N} y_n \delta \geq -1$$ the worst case is y_n always has the negative label Compare to y_m.

we have

$$\Rightarrow (N-1) \times (-1) \delta \geq -1$$

$$\Rightarrow \exp(-\gamma\varepsilon^2) \leq \frac{1}{N-1}$$

take ln

$$\Rightarrow -\gamma\varepsilon^2 \leq -\ln(N-1)$$

$$\Rightarrow \gamma \geq \frac{\ln(N-1)}{\varepsilon^2}$$ 時 [D].

11.

the squared distance

is $\|\phi(x_1) - \phi(x')\|^2$

which is $\|\phi(x) - \phi(x')\|^2$

$= \phi(x)^T\phi(x) - 2\phi(x)^T\phi(x') + \phi(x')^T\phi(x)$

since we know the $k_f(x, x') = \phi(x)^T\phi(x') = \exp(-\gamma\|x - x'\|^2)$

we have

$= k_f(x, x) - 2k_f(x, x') + k(x', x')$

$= 2 - 2k(x, x')$

also, we know the $\exp(-\gamma\|x - x'\|^2)$

will locate at $[0, 1]$, we want the upper bound.

, take $k(x, x') = 0$.

$= 2$.

We want the distance instead of squared distance.

$\Rightarrow \|\phi(x) - \phi(x')\| = \sqrt{\|\phi(x) - \phi(x')\|^2} \leq \sqrt{2} \doteq 1.4 < 1.5$ ✱

[d] ✱.

In [ ]:
```
import numpy as np
from itertools import combinations_with_replacement
from liblinear.liblinearutil import *
```

In [ ]:
```
help(train)
```

```
Help on function train in module liblinear.liblinearutil:

train(arg1, arg2=None, arg3=None)
    train(y, x [, options]) -> model | ACC

    y: a list/tuple/ndarray of l true labels (type must be int/double).

    x: 1. a list/tuple of l training instances. Feature vector of
          each training instance is a list/tuple or dictionary.

       2. an l * n numpy ndarray or scipy spmatrix (n: number of features).

    train(prob [, options]) -> model | ACC
    train(prob, param) -> model | ACC

    Train a model from data (y, x) or a problem prob using
    'options' or a parameter param.

    If '-v' is specified in 'options' (i.e., cross validation)
    either accuracy (ACC) or mean-squared error (MSE) is returned.

    options:
        -s type : set type of solver (default 1)
          for multi-class classification
             0 -- L2-regularized logistic regression (primal)
             1 -- L2-regularized L2-loss support vector classification (dual)
             2 -- L2-regularized L2-loss support vector classification (primal)
             3 -- L2-regularized L1-loss support vector classification (dual)
             4 -- support vector classification by Crammer and Singer
             5 -- L1-regularized L2-loss support vector classification
             6 -- L1-regularized logistic regression
             7 -- L2-regularized logistic regression (dual)
          for regression
            11 -- L2-regularized L2-loss support vector regression (primal)
            12 -- L2-regularized L2-loss support vector regression (dual)
            13 -- L2-regularized L1-loss support vector regression (dual)
          for outlier detection
            21 -- one-class support vector machine (dual)
        -c cost : set the parameter C (default 1)
        -p epsilon : set the epsilon in loss function of SVR (default 0.1)
        -e epsilon : set tolerance of termination criterion
            -s 0 and 2
                |f'(w)|_2 <= eps*min(pos,neg)/l*|f'(w0)|_2,
                where f is the primal function, (default 0.01)
            -s 11
                |f'(w)|_2 <= eps*|f'(w0)|_2 (default 0.0001)
            -s 1, 3, 4, 7, and 21
                Dual maximal violation <= eps; similar to libsvm (default 0.1 except
0.01 for -s 21)
            -s 5 and 6
                |f'(w)|_inf <= eps*min(pos,neg)/l*|f'(w0)|_inf,
                where f is the primal function (default 0.01)
            -s 12 and 13
                |f'(alpha)|_1 <= eps |f'(alpha0)|,
                where f is the dual function (default 0.1)
        -B bias : if bias >= 0, instance x becomes [x; bias]; if < 0, no bias term ad
ded (default -1)
        -R : not regularize the bias; must with -B 1 to have the bias; DON'T use this
unless you know what it is
            (for -s 0, 2, 5, 6, 11)"
```

```
        -wi weight: weights adjust the parameter C of different classes (see README f
or details)
        -v n: n-fold cross validation mode
        -C : find parameters (C for -s 0, 2 and C, p for -s 11)
        -q : quiet mode (no outputs)
```

In [ ]:
```python
def polynomial_transformation(x, order):
    n, index = x.shape[0], list(range(x.shape[1]))
    new_x = np.ones((x.shape[0], 1))
    for o in range(1, order+1):
        idx_set = np.array(list(combinations_with_replacement(index, o)))
        for set in idx_set:
            new_col = np.ones((x.shape[0], 1))
            for i in set:
                new_col *= np.reshape(x[:,i], (n, 1))
            new_x = np.hstack((new_x, new_col))
    return new_x
```

In [ ]:
```python
"""
load data and preprocess
"""
# read data
with open('hw4_train.dat', 'rb') as f:
    training_data = np.array([np.float64(i.split()) for i in f.readlines()])

# turn x and y into numpy array
# x is the input feature vector space and y is the corresponding label
x = np.array(training_data[:,0:10])
y = np.reshape(np.array(list(map(int, training_data[:,10]))), (len(x), 1))
print(x.shape)
print(y.shape)

N = len(x)

with open('hw4_test.dat', 'rb') as f:
    test_data = np.array([np.float64(i.split()) for i in f.readlines()])

# turn x and y into numpy array
# x is the input feature vector space and y is the corresponding label
x_test = np.array(test_data[:,0:10])
y_test = np.reshape(np.array(list(map(int, test_data[:,10]))), (len(x_test), 1))
print(x_test.shape)
print(y_test.shape)

N = len(x)
```

```
(200, 10)
(200, 1)
(500, 10)
(500, 1)
```

In [ ]:
```python
"""
polynomial transformation
"""
order = 4
x_transform = polynomial_transformation(x, order)
x_test_transform = polynomial_transformation(x_test, order)

# for i in range(x_transform.shape[0]):
```

```
#     print(x_transform[i])

print(x_transform.shape)
print(x_test_transform.shape)

y = y.flatten()
y_test = y_test.flatten()
print(y.shape)
```

```
(200, 1001)
(500, 1001)
(200,)
```

In [ ]:

```
"""
P12
"""

lamb = np.array([-6, -3, 0, 3, 6], dtype=np.float32)
prob = problem(y, x_transform)
E_list = []
for l in lamb:
    parm = parameter(f'-s 0 -c {1/(2*(10**l))} -e 0.000001')
    # print(1/(2*(10**l)))
    model = train(prob, parm)
    p_label, p_acc, p_val = predict(y_test, x_test_transform, model)
    E_list.append(np.mean(p_label != y_test))

best_e = 1.1
best_lambda = -np.inf
for idx, e in enumerate(E_list):
    if e < best_e:
        best_e = e
        best_lambda = lamb[idx]
    elif e == best_e and lamb[idx] > best_lambda:
        best_lambda = lamb[idx]
        best_idx = idx
```

```
Accuracy = 77.4% (387/500) (classification)
Accuracy = 82.2% (411/500) (classification)
Accuracy = 84.6% (423/500) (classification)
Accuracy = 85.8% (429/500) (classification)
Accuracy = 81.2% (406/500) (classification)
```

In [ ]:

```
"""
p12 ans
"""
print(lamb)
print(E_list)
print(f'best error = {best_e}, best lambda = {best_lambda}')
```

```
[-6. -3.  0.  3.  6.]
[0.226, 0.178, 0.154, 0.142, 0.188]
best error = 0.142, best lambda = 3.0
```

In [ ]:

```
"""
P13
"""
lamb = np.array([-6, -3, 0, 3, 6], dtype=np.float32)
prob = problem(y, x_transform)
E_list = []
for l in lamb:
```

```
    parm = parameter(f'-s 0 -c {1/(2*(10**l))} -e 0.000001')
    # print(1/(2*(10**l)))
    model = train(prob, parm)
    p_label, p_acc, p_val = predict(y, x_transform, model)
    E_list.append(np.mean(p_label != y))

best_e = 1.1
best_lambda = -np.inf
for idx, e in enumerate(E_list):
    if e < best_e:
        best_e = e
        best_lambda = lamb[idx]
    elif e == best_e and lamb[idx] > best_lambda:
        best_lambda = lamb[idx]
        best_idx = idx
```

```
Accuracy = 100% (200/200) (classification)
Accuracy = 100% (200/200) (classification)
Accuracy = 100% (200/200) (classification)
Accuracy = 96% (192/200) (classification)
Accuracy = 76% (152/200) (classification)
```

In [ ]:

```
"""
p13 ans
"""
print(lamb)
print(E_list)
print(f'best error = {best_e}, best lambda = {best_lambda}')
```

```
[-6. -3.  0.  3.  6.]
[0.0, 0.0, 0.0, 0.04, 0.24]
best error = 0.0, best lambda = 0.0
```

In [ ]:

```
"""
P14
"""

lamb = np.array([-6, -3, 0, 3, 6], dtype=np.float32)

lamb_cnt = {}
for l in lamb:
    lamb_cnt[l] = 0

for i in range(256):
    # generate random numbers between 0 to len(x_transform) without repeat, this step
    rng = np.random.default_rng()
    numbers = rng.choice(len(x_transform), size=len(x_transform), replace=False)

    # 120 tranining samples
    x_train = x_transform[numbers[:120]]
    y_train = y[numbers[:120]]
    # 80 evaluation samples
    x_eval = x_transform[numbers[120:]]
    y_eval = y[numbers[120:]]

    E_list = []

    prob = problem(y_train, x_train)
    for l in lamb:
        parm = parameter(f'-s 0 -c {1/(2*(10**l))} -e 0.000001')
```

```
        # print(1/(2*(10**l)))
        model = train(prob, parm)
        p_label, p_acc, p_val = predict(y_eval, x_eval, model)
        E_list.append(np.mean(p_label != y_eval))

    best_e = 1.1
    best_lambda = -np.inf
    for idx, e in enumerate(E_list):
        if e < best_e:
            best_e = e
            best_lambda = lamb[idx]
            best_idx = idx
        elif e == best_e and lamb[idx] > best_lambda:
            best_lambda = lamb[idx]
            best_idx = idx

    # print(Eout_list)
    # print(f'best error = {best_e}, best lambda = {best_lambda}')
    lamb_cnt[best_lambda] += 1

    # I delete the training output(Accuracy part), cuz its too lengthy and cannot be r
```

In [ ]:
```
"""
p14 ans
"""
print(lamb_cnt)
max(lamb_cnt, key=lamb_cnt.get)
```

Out[ ]:
```
{-6.0: 1, -3.0: 20, 0.0: 77, 3.0: 157, 6.0: 1}
3.0
```

In [ ]:
```
"""
P15
"""

lamb = np.array([-6, -3, 0, 3, 6], dtype=np.float32)

lamb_cnt = {}
for l in lamb:
    lamb_cnt[l] = 0

p15_ans = []
for i in range(256):
    # generate random numbers between 0 to len(x_transform) without repeat, this step
    rng = np.random.default_rng()
    numbers = rng.choice(len(x_transform), size=len(x_transform), replace=False)

    # 120 tranining samples
    x_train = x_transform[numbers[:120]]
    y_train = y[numbers[:120]]
    # 80 evaluation samples
    x_eval = x_transform[numbers[120:]]
    y_eval = y[numbers[120:]]

    E_list = []

    prob = problem(y_train, x_train)
    for l in lamb:
        parm = parameter(f'-s 0 -c {1/(2*(10**l))} -e 0.000001')
```

```
        # print(1/(2*(10**l)))
        model = train(prob, parm)
        p_label, p_acc, p_val = predict(y_eval, x_eval, model)
        E_list.append(np.mean(p_label != y_eval))

    best_e = 1.1
    best_lambda = -np.inf
    for idx, e in enumerate(E_list):
        if e < best_e:
            best_e = e
            best_lambda = lamb[idx]
            best_idx = idx
        elif e == best_e and lamb[idx] > best_lambda:
            best_lambda = lamb[idx]
            best_idx = idx

    prob = problem(y_train, x_train)
    parm = parameter(f'-s 0 -c {1/(2*(10**best_lambda))} -e 0.000001')
    model = train(prob, parm)
    p_label, p_acc, p_val = predict(y_test, x_test_transform, model)

    p15_ans.append(np.mean(p_label != y_test))
    # print(Eout_list)
    # print(f'best error = {best_e}, best lambda = {best_lambda}')
    lamb_cnt[best_lambda] += 1

    # I delete the training output(Accuracy part), cuz its too lengthy and cannot be r
```

In [ ]:

```
"""
p15 ans
"""
print(np.mean(p15_ans))
```

```
0.167234375
```

In [ ]:

```
"""
P16
"""

lamb = np.array([-6, -3, 0, 3, 6], dtype=np.float32)

lamb_cnt = {}
for l in lamb:
    lamb_cnt[l] = 0


p16_ans = []
for i in range(256):
    # generate random numbers between 0 to len(x_transform) without repeat, this step
    rng = np.random.default_rng()
    numbers = rng.choice(len(x_transform), size=len(x_transform), replace=False)

    # 120 tranining samples
    x_train = x_transform[numbers[:120]]
    y_train = y[numbers[:120]]
    # 80 evaluation samples
    x_eval = x_transform[numbers[120:]]
    y_eval = y[numbers[120:]]

    E_list = []
```

```
    prob = problem(y_train, x_train)
    for l in lamb:
        parm = parameter(f'-s 0 -c {1/(2*(10**l))} -e 0.000001')
        # print(1/(2*(10**l)))
        model = train(prob, parm)
        p_label, p_acc, p_val = predict(y_eval, x_eval, model)
        E_list.append(np.mean(p_label != y_eval))

    best_e = 1.1
    best_lambda = -np.inf
    for idx, e in enumerate(E_list):
        if e < best_e:
            best_e = e
            best_lambda = lamb[idx]
            best_idx = idx
        elif e == best_e and lamb[idx] > best_lambda:
            best_lambda = lamb[idx]
            best_idx = idx
    prob = problem(y, x_transform)
    parm = parameter(f'-s 0 -c {1/(2*(10**best_lambda))} -e 0.000001')
    model = train(prob, parm)
    p_label, p_acc, p_val = predict(y_test, x_test_transform, model)


    p16_ans.append(np.mean(p_label != y_test))
    lamb_cnt[best_lambda] += 1
    # I delete the training output(Accuracy part), cuz its too lengthy and cannot be r
```

In [ ]:

```
"""
p16 ans
"""
print(np.mean(p16_ans))
```

```
0.14942968749999996
```

In [ ]:

```
"""
P17
"""
lamb = np.array([-6, -3, 0, 3, 6], dtype=np.float32)
ans = []
for i in range(256):
    # generate random numbers between 0 to len(x_transform) without repeat, this step
    rng = np.random.default_rng()
    numbers = rng.choice(len(x_transform), size=len(x_transform), replace=False)

    fold_num = 5
    # slice into n-folds
    # length of each fold
    length = int(len(numbers)/fold_num)
    folds = []
    for k in range(fold_num-1):
        folds += [numbers[k*length:(k+1)*length]]
    folds += [numbers[(fold_num-1)*length:len(numbers)]]
    E = []

    for l in lamb:
        E_list = []

        for j in range(fold_num):
```

```
            temp = folds.copy()
            # print(len(folds))
            valid_idx = np.array(temp.pop(j)).flatten()
            training_idx = np.array(temp).flatten()

            training_sample = x_transform[training_idx]
            training_y = y[training_idx]
            valid_sample = x_transform[valid_idx]
            valid_y = y[valid_idx]

            prob = problem(training_y, training_sample)

            parm = parameter(f'-s 0 -c {1/(2*(10**l))} -e 0.000001')
            model = train(prob, parm)
            p_label, p_acc, p_val = predict(valid_y, valid_sample, model)
            E_list.append(np.mean(p_label != valid_y))

        E.append(np.mean(E_list))
    ans.append(min(E))
    # I delete the training output(Accuracy part), cuz its too lengthy and cannot be r
```

In [ ]:
```
"""
p17 ans
"""
print(np.mean(ans))
```

```
0.13078125000000002
```

In [ ]:
```
"""
P18
"""
lamb = np.array([-6, -3, 0, 3, 6], dtype=np.float32)
prob = problem(y, x_transform)
E_list = []
for l in lamb:
    parm = parameter(f'-s 6 -c {1/(10**l)} -e 0.000001')
    model = train(prob, parm)
    p_label, p_acc, p_val = predict(y_test, x_test_transform, model)
    E_list.append(np.mean(p_label != y_test))

best_e = 1.1
best_lambda = -np.inf
for idx, e in enumerate(E_list):
    if e < best_e:
        best_e = e
        best_lambda = lamb[idx]
    elif e == best_e and lamb[idx] > best_lambda:
        best_lambda = lamb[idx]
        best_idx = idx
```

```
Accuracy = 77.2% (386/500) (classification)
Accuracy = 84.4% (422/500) (classification)
Accuracy = 84.6% (423/500) (classification)
Accuracy = 68% (340/500) (classification)
Accuracy = 49.2% (246/500) (classification)
```

In [ ]:
```
"""
p18 ans
"""
print(lamb)
```

```
print(E_list)
print(f'best error = {best_e}, best lambda = {best_lambda}')
```

```
[-6. -3.  0.  3.  6.]
[0.228, 0.156, 0.154, 0.32, 0.508]
best error = 0.154, best lambda = 0.0
```

In [ ]:
```
"""
P19
"""
# bast lambda from p18
prob = problem(y, x_transform)
parm = parameter(f'-s 6 -c {1/(10**best_lambda)} -e 0.000001')
model = train(prob, parm)

cnt = 0
for i in range(x_transform.shape[1]):
    if np.abs(model.w[i]) <= 10**-6:
        cnt += 1
```

In [ ]:
```
"""
p19 ans
"""
cnt
```

Out[ ]:
```
960
```

In [ ]:
```
"""
P20
"""
lamb = np.array([-6, -3, 0, 3, 6], dtype=np.float32)
prob = problem(y, x_transform)
E_list = []
for l in lamb:
    parm = parameter(f'-s 0 -c {1/(2*(10**l))} -e 0.000001')
    # print(1/(2*(10**l)))
    model = train(prob, parm)
    p_label, p_acc, p_val = predict(y_test, x_test_transform, model)
    E_list.append(np.mean(p_label != y_test))

best_e = 1.1
best_lambda = -np.inf
for idx, e in enumerate(E_list):
    if e < best_e:
        best_e = e
        best_lambda = lamb[idx]
    elif e == best_e and lamb[idx] > best_lambda:
        best_lambda = lamb[idx]
        best_idx = idx

prob = problem(y, x_transform)
parm = parameter(f'-s 0 -c {1/(2*(10**best_lambda))} -e 0.000001')
model = train(prob, parm)

cnt = 0
for i in range(x_transform.shape[1]):
    if np.abs(model.w[i]) <= 10**-6:
```

```
        cnt += 1
```

```
Accuracy = 77.4% (387/500) (classification)
Accuracy = 82.2% (411/500) (classification)
Accuracy = 84.6% (423/500) (classification)
Accuracy = 85.8% (429/500) (classification)
Accuracy = 81.2% (406/500) (classification)
```

In [ ]:
```
"""
p20 ans
"""
cnt
```

Out[ ]:
```
1
```