EE5904 Homework 1 Zhang Fei A0117981X

Q1① $\varphi(v) = av + b$

$= a(x_1 w_{k1} + x_2 w_{k2} + \dots + x_m w_{km} + b_k) + b$; we assume $x_0 = 1$, $w_0 = b_k$, then the equation can be expressed as such:

$$\varphi(v) = a(x_0 w_0 + x_1 w_1 + \dots + x_m w_m) + b,$$
$$= a \sum_{i=0}^{m} x_i w_i + b$$

let equation $a\sum_{i=0}^{m} x_i w_i + b = \xi$ (threshold value)

$$a\sum_{i=0}^{m} x_i w_i + b - \xi = 0$$
$$\sum_{i=0}^{m} x_i w_i + \frac{b-\xi}{a} = 0$$

Since the term $\frac{b-\xi}{a}$ is constant, therefore, the resulting decision baundary is a hyper-plane

② $\varphi(v) = \frac{1}{1+e^{-2v}}$

$$= \frac{1}{1+e^{-2(\sum_{i=0}^{m} x_i w_i + b)}}$$

let equation equals to threshold value ξ, we get

$$\frac{1}{1 + e^{-2(\sum_{i=0}^{m} x_i w_i + b)}} = \xi$$
$$e^{-2(\sum_{i=0}^{m} x_i w_i + b)} = \frac{1}{\xi} - 1$$
$$-2\left(\sum_{i=0}^{m} x_i w_i + b\right) = \ln\left(\frac{1}{\xi} - 1\right)$$
$$\sum_{i=0}^{m} (x_i w_i + b) + \frac{1}{2}\left(\frac{1}{\xi} - 1\right) = 0$$

Since the term $\frac{1}{2}(\frac{1}{\xi}-1)$ is constant, then the resulting decision baundary is a hyper-plane

Q1

3. $\phi(v) = e^{-\frac{v^2}{2}}$, ξ is defined threshold value.

let $e^{-\frac{v^2}{2}} = \xi$

$-\frac{v^2}{2} = \ln \xi$

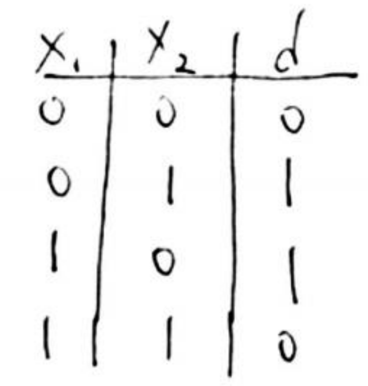
$v^2 = -2 \ln \xi$

$v = \sum_{i=0}^{m} w_i x_i + b = \pm \sqrt{-2\ln \xi}$

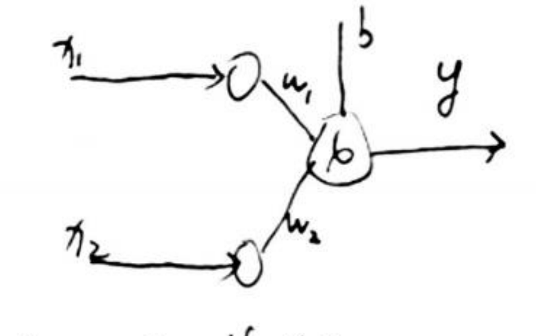
$\sum_{i=0}^{m} w_i x_i + b \pm \sqrt{-2\ln\xi} = 0$

Since the term $b \pm \sqrt{-2\ln\xi}$ is constant, therefore, the resulting decision boundary is a hyper-plane

Q2.



X_1	X_2	d
0	0	0
0	1	1
1	0	1
1	1	0



$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases}$$

Let's define the decision boundary equation: $w_1 x_1 + w_2 x_2 + b = 0$,

1, when $x_1 = 0$, $x_2 = 0$, having $y = d = 0$, which requires $b < 0$

2, when $x_1 = 1$, $x_2 = 0$, having $y = d = 1$, which requires $w_1 + b > 0$

3, when $x_1 = 0$, $x_2 = 1$, having $y = d = 1$, which requires $w_2 + b > 0$

4, when $x_1 = 1$, $x_2 = 1$, having $y = d = 0$, which requires $-w_1 - w_2 - b > 0$

5,Adding equation (2) and (3) having $w_1 + w_2 + 2b > 0$

Adding (4) and (5): $b > 0$, this is in contradict with the condition required in equation (1),

Therefore, the XOR problem is not linear separable.

Q3

a)To implement AND logic, we use weights w=[-1.5 1 1]^T for bias, x_1 and x_2 respectively,

V= $x_1 + x_2 - 1.5$

The values yield the following table:

x_1	x_2	bias	v	y
0	0	-1.5	-1.5	0
0	1	-1.5	-0.5	0
1	0	-1.5	-0.5	0
1	1	-1.5	0.5	1

To implement OR logic, we can use weights w=[-0.5 1 1]^T for bias, x_1 and x_2 respectively,

V = $x_1 + x_2 - 0.5$

The values yield the following table:

x_1	x_2	bias	v	y
0	0	-0.5	-0.5	0
0	1	-0.5	-0.5	1
1	0	-0.5	-0.5	1
1	1	-0.5	1.5	1

To implement COMPLEMENT logic, we can use weights w=[0.5 -1]^T for bias and respectively,

V = $-x + 0.5$

The values yield the following table:

X	Bias	V	Y
0	0.5	0.5	1
1	0.5	-0.5	0

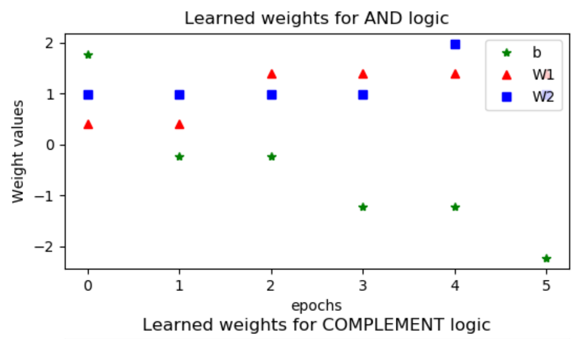
To implement NAND logic, we can use weights w=[1.5 -1 -1]^T for bias, x_1 and x_2 respectively,

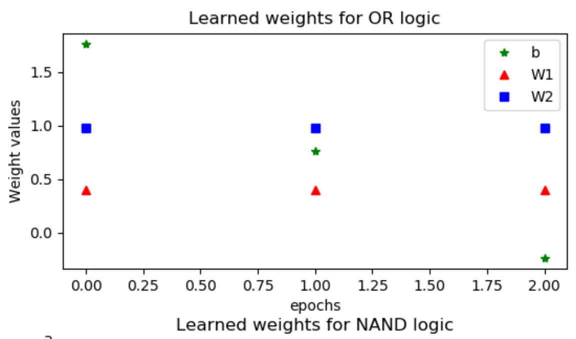
V = $-x_1 - x_2 + 1.5$

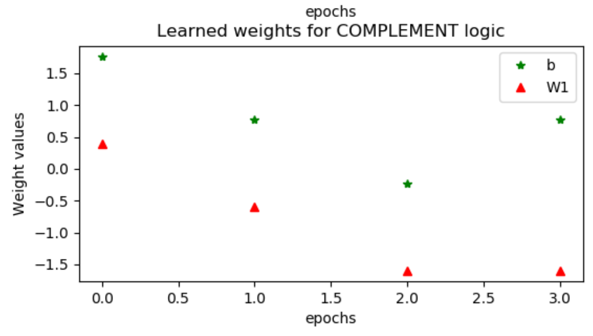
The values yield the following table:

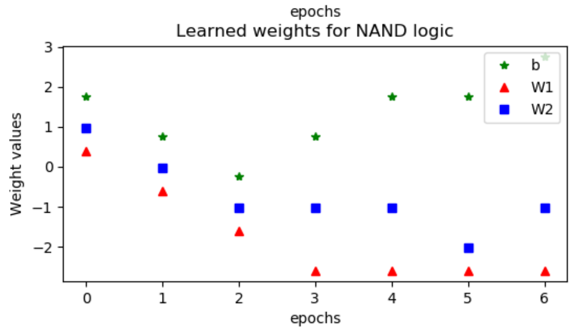
x_1	x_2	bias	v	y
0	0	1.5	1.5	1
0	1	1.5	0.5	1
1	0	1.5	0.5	1
1	1	1.5	-0.5	0

b1.Trajectories of weights for each cases at learning rate = 1

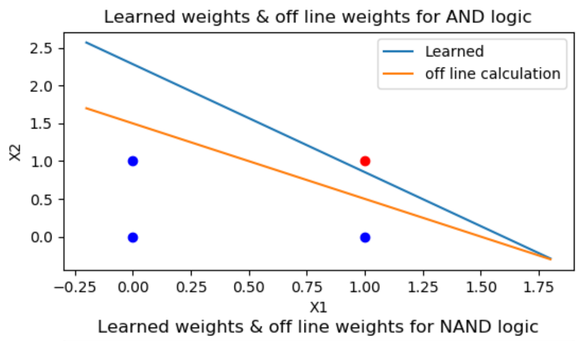


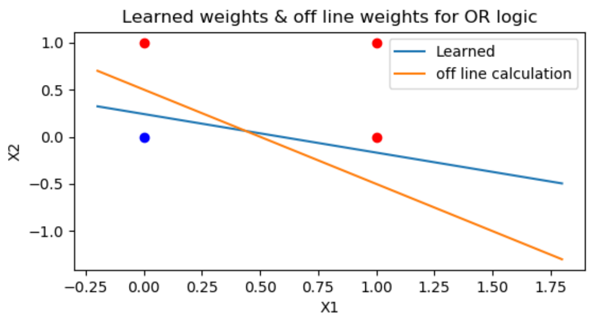


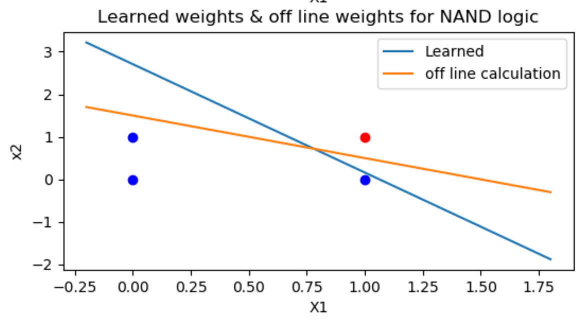




The initial weights will update when there is an error signal. The weights will converge when there is no more error signal produced. The final obtained weights in each case are very close to the off-line calculated weights. Both off-line calculated weights and iterated weights can implement AND, OR & NAND logics.







For Complement logic, ignore w2 since w2 is 0. the final learned weights is [0.76 -1.60] : v= -1.6x +0.76

When x = 0, v = 0.76 which greater than 0, y = f(v) = 1; when x = 1, v = -0.84, y = f(v) = 0

Both the learned weights and calculate off line weights model can implement complement logic.

b2.

From my observation on different learning rate, it controls how quickly the model is adapted to the logic gate problem. Smaller learning rates require more training epochs. As the smaller rates make smaller changes on the weights for each update. Whereas, larger learning rates can cause a rapid change in weights and require fewer epochs. If the learning rate is too large can result the model to converge too fast to a suboptimal result; if the learning rate is too small can cause the process to get stuck.

c. The computer simulation results show that the training process will keep on iterating, the training perceptron will not converge, no matter how weights updated, it is not able to find a linear line or hyper-plane to separate the two class.

################################Q3 source code##################################

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

# inputs initializing
# columns = (bias, x1, x2, y)
AND_set = np.array([[1,0,0,0],
                    [1,0,1,0],
                    [1,1,0,0],
                    [1,1,1,1]])

OR_set = np.array([[1,0,0,0],
                   [1,0,1,1],
                   [1,1,0,1],
                   [1,1,1,1]])
```

```
COMPLEMENT_set = np.array([[1,0,1],
                           [1,1,0]])

NAND_set =np.array([[1,0,0,1],
                    [1,0,1,1],
                    [1,1,0,1],
                    [1,1,1,0]])

XOR_set =np.array([[1,0,0,0],
                    [1,0,1,1],
                    [1,1,0,1],
                    [1,1,1,0]])

# learning rate
learn_rate = 1

def activation_fn(x):
    return 1 if x >=0 else 0

def train(x):
    dim = x.shape[1]
    col = x.shape[0]
    iterate = True
    iteration = 0

    #initial weights
    np.random.seed(0)
    weights = np.random.randn(1,dim -1)
    W = weights
    print("Initial weights is " + str(weights))
    # for i in range(iteration):
```

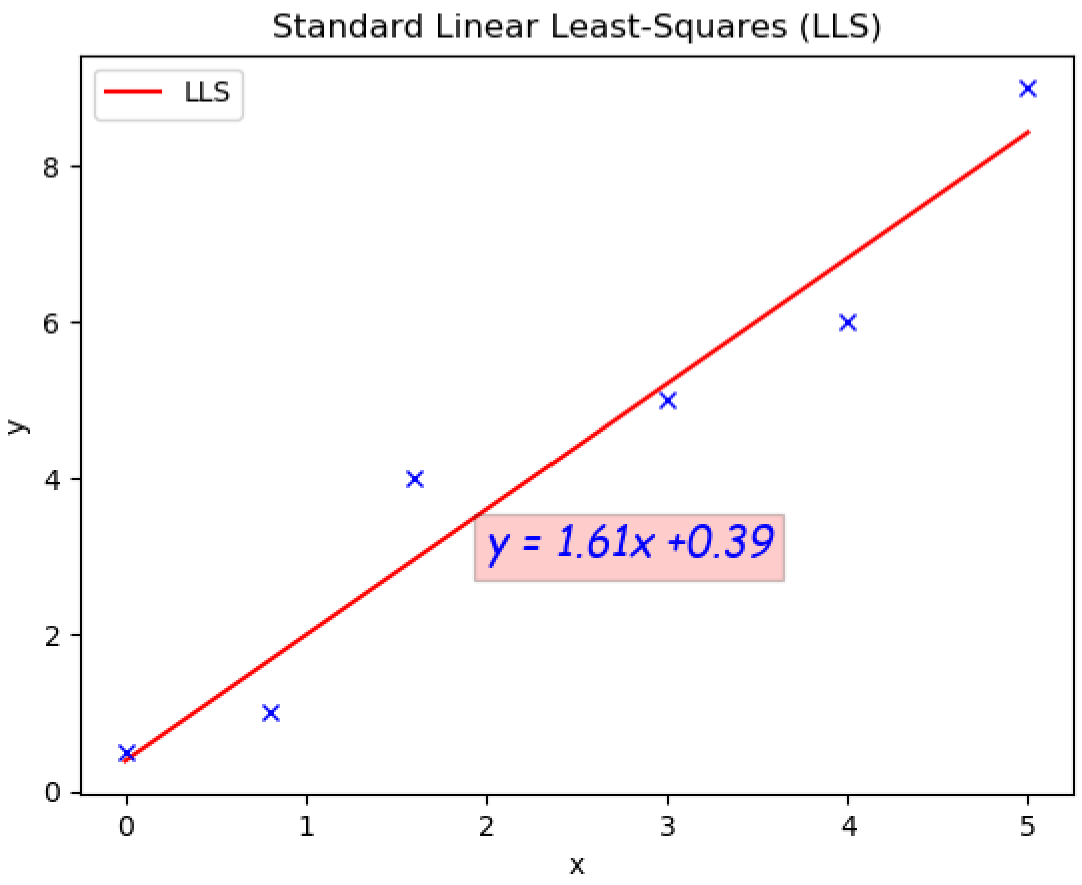
```
    while iterate:
        iterate = False
        for j in range(col):
            v = weights.dot(x[j,:-1])
            v_out = activation_fn(v)
            error = x[j,-1] - v_out
            if error != 0 :
                # print("wrong weights is " + str
(weights))
                weights = weights + learn_rate*x[
j,:-1]*error
                # W = np.append(W,weights,axis =
0)
                # print("Updated weights is " + s
tr(weights))
                iterate = True
        if iterate == True:
            W = np.append(W,weights,axis = 0)
            print("Updated weights is " + str(wei
ghts))
        iteration += 1
    print(str(iteration) + " times of epochs")
    return W

def plotweights(W_AND,W_OR,W_NAND,W_COMPLEMENT):
    iter_range = range(W_AND.shape[0])
    iter_OR = range(W_OR.shape[0])
    iter_NAND = range(W_NAND.shape[0])
    iter_COMPLEMENT = range(W_COMPLEMENT.shape[0]
)
```

```
    plt.figure(num=3, figsize=(10, 3),)
    plt.subplots_adjust(wspace =0.2, hspace =0.3)
    plt.subplot(2,2,1)
    plt.plot(iter_range,W_AND[:,0:1],'g*',iter_ra
nge,W_AND[:,1:2],'r^',iter_range,W_AND[:,2:3],'bs
')
    # plt.plot(iter_range,W[:,1:2],'r^','w1')
    # plt.plot(iter_range,W[:,2:3],'bs')
    plt.title("Learned weights for AND logic")
    plt.xlabel("epochs")
    plt.ylabel("Weight values")
    plt.legend(('b','W1','W2'),loc = 'upper right
')

    plt.subplot(2,2,2)
    plt.plot(iter_OR,W_OR[:,0:1],'g*',iter_OR,W_O
R[:,1:2],'r^',iter_OR,W_OR[:,2:3],'bs')
    # plt.scatter(iter_OR,W3,c='r')
    # plt.scatter(iter_OR,W4,c='b')
    plt.title("Learned weights for OR logic")
    plt.xlabel("epochs")
    plt.ylabel("Weight values")
    plt.legend(('b','W1','W2'),loc = 'upper right
')

    plt.subplot(2,2,3)
    plt.plot(iter_COMPLEMENT,W_COMPLEMENT[:,0:1],
'g*',iter_COMPLEMENT,W_COMPLEMENT[:,1:2],'r^')
    # plt.scatter(iter_COMPLEMENT,W7,c='r')
    plt.title("Learned weights for COMPLEMENT log
ic")
```

```
    plt.xlabel("epochs")
    plt.ylabel("Weight values")
    plt.legend(('b','W1'),loc = 'upper right')

    plt.subplot(2,2,4)
    plt.plot(iter_NAND,W_NAND[:,0:1],'g*',iter_NA
ND,W_NAND[:,1:2],'r^',iter_NAND,W_NAND[:,2:3],'bs
')
    plt.title("Learned weights for NAND logic")
    plt.xlabel("epochs")
    plt.ylabel("Weight values")
    plt.legend(('b','W1','W2'),loc = 'upper right
')
    plt.show()



def compare_weights(W_AND,W_OR,W_NAND):
    X = np.arange(-0.2,2,0.5)
    y_AND_learn = -(W_AND[W_AND.shape[0]-
1,1]*X + W_AND[W_AND.shape[0]-
1,0])/W_AND[W_AND.shape[0]-1,2]
    y_AND_off = -X + 1.5

    y_OR_learn = -(W_OR[W_OR.shape[0]-
1,1]*X + W_OR[W_OR.shape[0]-
1,0])/W_OR[W_OR.shape[0]-1,2]
    y_OR_off = -X + 0.5
    y_NAND_learn = -(W_NAND[W_NAND.shape[0]-
1,1]*X + W_NAND[W_NAND.shape[0]-
1,0])/W_NAND[W_NAND.shape[0]-1,2]
    y_NAND_off = -X + 1.5

    plt.figure(num=3, figsize=(10, 3),)
    plt.subplots_adjust(wspace =0.2, hspace =0.3)
    plt.subplot(2,2,1)
    plt.plot([0,0,1],[0,1,0],'bo',[1],[1],'ro')
    plt.plot(X,y_AND_learn,label = "Learned")
    plt.plot(X,y_AND_off,label = "off line calcul
ation")
    plt.title("Learned weights & off line weights
 for AND logic")
    plt.xlabel("X1")
    plt.ylabel("X2")
    plt.legend()

    plt.subplot(2,2,2)
    plt.plot([0],[0],'bo',[0,1,1],[1,0,1],'ro')
    plt.plot(X,y_OR_learn,label = "Learned")
    plt.plot(X,y_OR_off,label = "off line calcula
tion")
    plt.title("Learned weights & off line weights
 for OR logic")
    plt.xlabel("X1")
    plt.ylabel("X2")
    plt.legend()

    plt.subplot(2,2,3)
    plt.plot([0,0,1],[0,1,0],'bo',[1],[1],'ro')
    plt.plot(X,y_NAND_learn,label = "Learned")
    plt.plot(X,y_NAND_off,label = "off line calcu
lation")
    plt.title("Learned weights & off line weights
 for NAND logic")
    plt.xlabel("X1")
    plt.ylabel("x2")
    plt.legend()

    plt.show()



W_AND = train(AND_set)
W_OR = train(OR_set)
W_NAND = train(NAND_set)
W_COMPLEMENT = train(COMPLEMENT_set)
Plotweights = plotweights(W_AND,W_OR,W_NAND,W_COM
PLEMENT)
# Compare_weights = compare_weights(W_AND,W_OR,W_
NAND)
# X_OR =train(XOR_set)
```

################################End of Q3######################################

Q4a



W = 1.61, b = 0.39

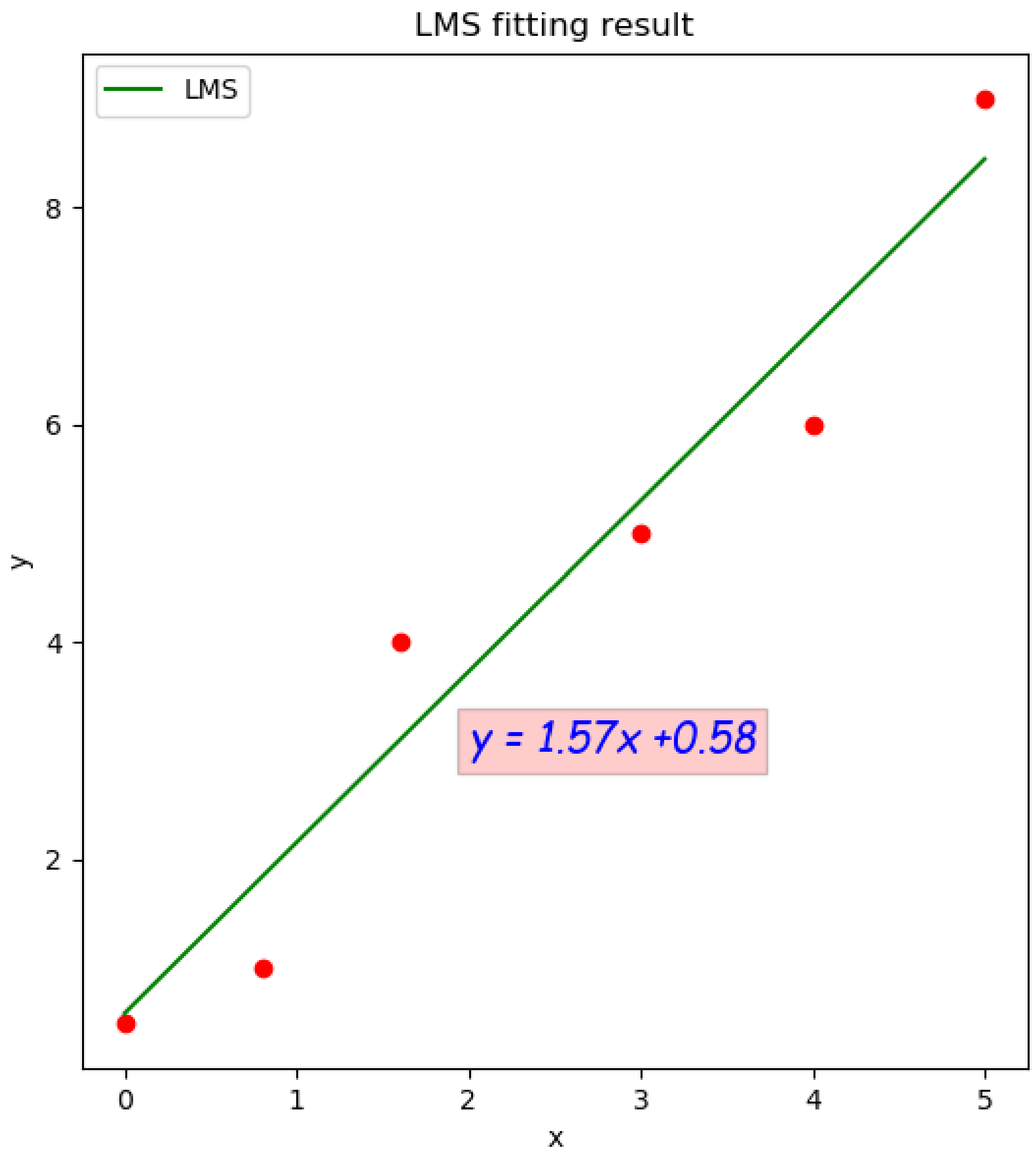
################################source code for Q4a#####################

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

x = np.array([[1,0],[1,0.8],[1,1.6],[1,3],[1,4],[
1,5]])
d = np.array([[0.5] , [1], [4], [5], [6], [9]])

w = np.mat(np.linalg.inv((np.mat(np.transpose(x))
)*(np.mat(x))))*np.mat(np.transpose(x))*np.mat(d)
# w0 = 0.38733906 w1 = 1.60944206
X = np.arange(0,6,1)
y = w[1,0]*X + w[0,0]
```

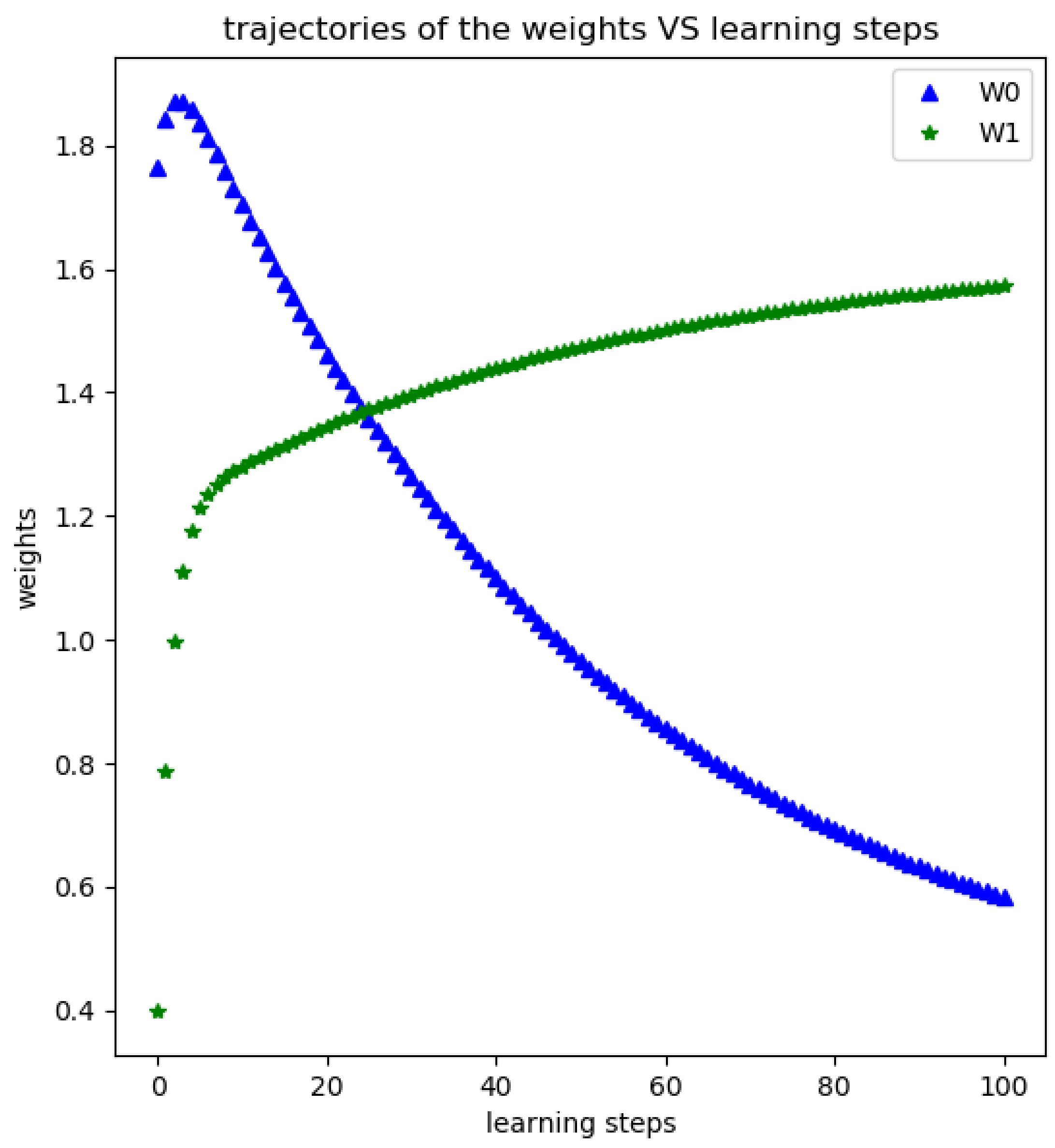
```
plt.figure()
plt.title('Standard Linear Least-Squares (LLS)')
plt.xlabel('x')
plt.ylabel('y')
plt.plot(X,y,color= 'red',label = "LLS")
plt.plot(x[:,1:2],d[:,:],'bx')
plt.text(2, 3, "y = "+str("%.2f" %w[1,0]) +"x +"+
str("%.2f" %w[0,0]), size = 15,\
         family = "fantasy", color = "b", style =
 "italic", weight = "light",\
         bbox = dict(facecolor = "r", alpha = 0.2
))

plt.legend()
plt.show()
```

################################end of Q4a##############################

Q4b.



W=1.57, b = 0.58



The weights will not converge. the weights will keep updated until the times of iteration are met.

##########################source code for Q4b################

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

x = np.array([[1,0],[1,0.8],[1,1.6],[1,3],[1,4],[
1,5]])
d = np.array([[0.5] , [1], [4], [5], [6], [9]])
```

```
#learning rate
learn_rate = 0.009

# 100 epochs
iteration = 100

def train(x,iteration,learn_rate):
    #initial weights
    np.random.seed(0)
    weights = np.random.randn(1,x.shape[1])
    W = weights
    for i in range(iteration):
        for j in range(x.shape[0]):
            error = d[j] - np.mat(x[j,:])*np.tran
spose(weights)
            weights = weights + learn_rate*error*
x[j,:]
        W = np.append(W,weights,axis = 0)
        # print("Updated weights is " + str(weigh
ts))
    # print(str(iteration) + " times of epochs")
    return W


W_x = train(x,iteration,learn_rate)
# print(W_x)
X = np.arange(0,6,1)
y = W_x[W_x.shape[0]-1,1]*X + W_x[W_x.shape[0]-
1,0]
plt.figure()
plt.subplot(1,2,1)
plt.title('LMS fitting result')
plt.xlabel('x')
plt.ylabel('y')
plt.plot(x[:,1:2],d,'ro')
plt.text(2, 3, "y = "+str("%.2f" %W_x[W_x.shape[0
]-1,1]) +"x +"+str("%.2f" %W_x[W_x.shape[0]-
1,0]), size = 15,\
         family = "fantasy", color = "b", style =
 "italic", weight = "light",\
         bbox = dict(facecolor = "r", alpha = 0.2
))
plt.plot(X,y,color= 'green',label = "LMS")
plt.legend()

plt.subplot(1,2,2)
plt.title('trajectories of the weights VS learnin
g steps')
plt.xlabel('learning steps')
plt.ylabel('weights')
plt.plot(range(W_x.shape[0]),W_x[:,0:1],'b^',rang
e(W_x.shape[0]),W_x[:,1:2],'g*')
plt.legend(('W0','W1'),loc = 'upper right')
plt.show()
```

#########################end of Q4b#######################

Q4c.

The final results obtained by LLS and LMS methods are very close. The results obtain from LLS method is unique. The results obtained from LMS method are iterated from initial weights, iteration stops when a convergence criterion is satisfied. Different initial weights can result different weights.

Q4d.

A smaller learning rate make small changes on weights each update, this allow the model to learn a more optimal set of weights. Whereas, a larger learning rate can make rapid changes each update, this will result a sub-optimal set of weights. If learning rate is too large or too small, the model may result in a failure train.

#######################source code for Q4d######################

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

x = np.array([[1,0],[1,0.8],[1,1.6],[1,3],[1,4],[
1,5]])
d = np.array([[0.5] , [1], [4], [5], [6], [9]])

#learning rate
learn_rate = 0.009

# 100 epochs
iteration = 100

def train(x,iteration,learn_rate):
    #initial weights
    np.random.seed(0)
    weights = np.random.randn(1,x.shape[1])
    W = weights
    for i in range(iteration):
        for j in range(x.shape[0]):
            error = d[j] - np.mat(x[j,:])*np.tran
spose(weights)
            weights = weights + learn_rate*error*
x[j,:]
```

```
        W = np.append(W,weights,axis = 0)
        # print("Updated weights is " + str(weigh
ts))
    # print(str(iteration) + " times of epochs")
    return W


W_x = train(x,iteration,learn_rate)
# print(W_x)
X = np.arange(0,6,1)
y = W_x[W_x.shape[0]-1,1]*X + W_x[W_x.shape[0]-
1,0]
plt.figure()
plt.title('LMS fitting result at '+str(learn_rate
)+' learning rate')
plt.xlabel('x')
plt.ylabel('y')
plt.plot(x[:,1:2],d,'ro')
plt.text(2, 3, "y = "+str("%.2f" %W_x[W_x.shape[0
]-1,1]) +"x +"+str("%.2f" %W_x[W_x.shape[0]-
1,0]), size = 15,\
         family = "fantasy", color = "b", style =
 "italic", weight = "light",\
         bbox = dict(facecolor = "r", alpha = 0.2
))
plt.plot(X,y,color= 'green',label = "LMS")
plt.legend()
plt.show()
```

########################end of Q4#####################################

Q5

The optimal condition to get w^* is $\frac{\partial J(w)}{\partial w} = 0$

$$J(w) = \sum_{i=1}^{n} r(i) e(i)^2 = \sum_{i=1}^{n} r(i) (d(i) - y(x(i)))^2$$

$$\frac{\partial J(w)}{\partial w} = \frac{\partial J(w)}{\partial e} \cdot \frac{\partial e}{\partial w}$$

Since $e = d - y = d(i) - y(x(i)) = d(i) - X^T(i) w(i)$

$$\frac{\partial e}{\partial w} = -X^T(i)$$

and $\frac{\partial J(w)}{\partial e} = 2r(i) e(i)$

therefore: $\frac{\partial J(w)}{\partial w(i)} = -2 r(i) e(i) X^T(i)$

The gradient of $J(w)$, $g(i) = \left(\frac{\partial J(w)}{\partial w(i)}\right)^T = -2r(i) e(i) X(i)$

Applying steepest descent method, we have

$$w(i+1) = w(i) - \eta g(i)$$
$$= w(i) + 2\eta r(i) e(i) x(i)$$

η is learning rate parameter