CS/CNS/EE 156a
Homework 2

Sung Hoon Choi

1. Answer: [b]

According to my simulation, the average value of ν_{min} was 0.0371.

Please see the code below for derivation.

2. Answer: [d]

The average value for ν_1 was 0.504, ν_{rand} was 0.498, and ν_{min} was 0.037.

Meanwhile, since the coins are fair coins, the probability (distribution) of head in general must be 0.5

Thus, The only samples that have this distribution were c_1 and c_{rand}.

Picking the first coin and a random coin are basically equivalent since we are throwing fair coins randomly.

Please see the code below for derivation.

Code for Problem 1~2 (Python)

```
#Sung Hoon Choi
#CS/CNS/EE156a HW2 Problem 1 and Problem2

import numpy as np

Exp_Num = 1000000
Flip_Times = 10
coin_data = np.zeros((1000,Flip_Times))
Total_V1 = 0
Total_Vrand = 0
Total_Vmin = 0

for experiment in range (0,Exp_Num):
    for i in range(0,1000):
        each_coin_data = np.zeros(Flip_Times)
        for j in range(0,Flip_Times):
            if np.random.rand(1) > 0.5:
                each_coin_data[j] = 1
            else:
                each_coin_data[j] = 0

        coin_data[i,0:Flip_Times] = each_coin_data

    #print("coin_data: \n",coin_data)

    #Find C1
    C1 = coin_data[0,:]
    #print("C1: ",C1)

    #Find Crand
    rand_index = (int)(np.random.rand(1)*1000)
    #print("rand_index: ",rand_index)
    Crand = coin_data[rand_index,:]
    #print("Crand: " ,Crand)

    #Find Cmin
    sum_coin_data = (np.sum(coin_data, axis=1)).T  #sum_coin_data's shape = (1000,0)
    #print("sum_coin:\n", sum_coin_data)
    min_coin_sum = min(sum_coin_data)
    #print("min_coin_val: ", min_coin_sum)
```

```
    for index in range (0,1000):
        if sum_coin_data[index] == min_coin_sum:
            Cmin_coin_index = index

    Cmin = coin_data[Cmin_coin_index,:]
    #print("Cmin: ", Cmin)

    #Find V1
    V1 = np.sum(C1)/Flip_Times
    #print("V1: ", V1)

    #Find Vrand
    Vrand = np.sum(Crand)/Flip_Times
    #print("Vrand: ", Vrand)

    #Find Vmin
    Vmin = np.sum(Cmin)/Flip_Times
    #print("Vmin: ", Vmin)

    Total_V1 = Total_V1 + V1
    Total_Vrand = Total_Vrand + Vrand
    Total_Vmin = Total_Vmin + Vmin

Average_V1 = Total_V1/Exp_Num
Average_Vrand = Total_Vrand/Exp_Num
Average_Vmin = Total_Vmin/Exp_Num

print("Average V1: ", Average_V1)          #0.5046
print("Average Vrand: ", Average_Vrand)     #0.4982
print("Average Vmin: ", Average_Vmin)       #0.0371
```

3. Answer: [e]

While hypothesis h may incorrectly approximate the target function f (with probability of μ), there is also a possibility that the target function f(x) is not actually equal to y. Since h and f are both binary-valued, we can see that h being wrong and f being wrong at the same time would cancel out the error and eventually give 'correct'. (double-negation). Thus, h incorrectly approximating y happens when one of "h=f" and "y=f(x)" is wrong, but not both.

Thus, the probability of error that h makes in approximating y is:

$$\mathrm{P(h = f) * P\big(y \neq f(x)\big)} + P(h \neq \mathrm{f}) * \mathrm{P\big(y = f(x)\big)} = \big(1-\mu\big)\big(1-\lambda\big) + \mu * \lambda$$

4. Answer: [b]

From Problem3, if we expand the equation, we get

$$\big(1-\mu\big)\big(1-\lambda\big) + \mu * \lambda = 1 - \lambda - \mu + 2\lambda\mu = 1 - \lambda + \mu(2\lambda - 1)$$

Thus, if $\lambda = \frac{1}{2}$, the μ term disappears and the performance of h would become independent of μ.

5. Answer: [c]

The average E_{in} I got was 0.0422. Please see the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a HW2 Problem 5

import numpy as np
```

```
def gen_target_func():           #generate a target function(f(x)) and return the corresponding vertical coordinate
                                 #input: none
                                 #output: target_function. format: [slope, y_intercept]
    rnd_x1 = np.zeros(2)
    rnd_x2 = np.zeros(2)

    for i in range (0,2):
        rnd_x1[i] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)

    for i in range (0,2):
        rnd_x2[i] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)

    slope_target_func = (rnd_x2[1]-rnd_x1[1])/(rnd_x2[0]-rnd_x1[0]) # slope = (y2-y1)/(x2-x1)

    y_intercept = rnd_x2[1]-slope_target_func*rnd_x2[0]

    return [slope_target_func, y_intercept]

def Label_data(X_vector, target_f): #return a correct label(1 or -1) by using the input vector and target equation f.
                                 #inputs
                                 # X_vector : input point's coordinate. format: [a, b]
                                 # target_f : target function. format: a
                                 #outputs
                                 # y : correct label for the input vector. format: a (1 or -1)

    if(X_vector[1] > target_f):  #if the input's vertical coordinate is above the target function, return 1 label
                                 #if the input's vertical coordinate is below the target function, return -1 label
        return 1
    else:
        return -1

def generate_random_point():   #generate random data point's coordinate
                               #inputs
                               # none
                               #outputs
                               # x: random points. format: [a,b]
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x

#Initializing the constants which will be used for training and testing
Total_iteration = 0        #Total iterations over all runs
Total_Run = 100            #Total number of experiments
Test_Data_Num = 1000       #Total number of testing data (used for testing the hypothesis g(x))
Total_Wrong_Points = 0     #Total number of f(x) != g(x) over all runs
Total_Unmatched_Num = 0

#Training begins
for run in range (0,Total_Run):
    N=100
    target_info = gen_target_func()    #target_info = [slope_target_func, y_intercept]

    x = np.zeros([N,4])      # x - [[1, x1,x2,label(y)],
                             #      [1, x1,x2,label(y)],
                             #      ..
                             #      [1, x1,x2,label(y)]]

    # generate N random data points with their correct labels based on the current target function f(x)
    for i in range (0, N):
        x[i,0] = 1                                  #x0 = 1
        x[i,1:3] = generate_random_point()          #random data points coordinate data
        f_x = target_info[0] * x[i,1] + target_info[1] #obtaining the target equation f
        x[i,3] = Label_data(x[i,1:3],f_x)           #using f, obtain the label(y) and append it to the array

    X = x[:,0:3]
    Y = x[:,3]

    X_pseudo_inverse = np.dot(np.linalg.inv(np.dot(X.T,X)),X.T)

    W = np.dot(X_pseudo_inverse,Y)

    g_output = np.zeros((N,1))
    for i in range (0,N):
        g_output[i] = np.sign(np.dot(W.T,X[i]))
```

```
    g_output = np.squeeze(g_output)


    insample_unmatched = 0
    for i in range (0,N):
        if g_output[i] != Y[i]:
            insample_unmatched = insample_unmatched + 1

    #print("Ein: ", insample_unmatched/N)                   #Ein for each test.
    Total_Unmatched_Num = Total_Unmatched_Num + insample_unmatched

print("Avergae Ein: ", Total_Unmatched_Num/(N*Total_Run))   #Calculate the average of Ein over the entire test runs.
```

6. Answer: [c]

The average E_{out} I got was 0.0495. Please see the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a HW2 Problem 6

import numpy as np

def gen_target_func():           #generate a target function(f(x)) and return the corresponding vertical coordinate
                                 #input: none
                                 #output: target_function. format: [slope, y_intercept]
    rnd_x1 = np.zeros(2)
    rnd_x2 = np.zeros(2)

    for i in range (0,2):
        rnd_x1[i] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)

    for i in range (0,2):
        rnd_x2[i] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)

    slope_target_func = (rnd_x2[1]-rnd_x1[1])/(rnd_x2[0]-rnd_x1[0]) # slope = (y2-y1)/(x2-x1)

    y_intercept = rnd_x2[1]-slope_target_func*rnd_x2[0]

    return [slope_target_func, y_intercept]

def Label_data(X_vector, target_f): #return a correct label(1 or -1) by using the input vector and target equation f.
                                 #inputs
                                 # X_vector : input point's coordinate. format: [a, b]
                                 # target_f : target function. format: a
                                 #outputs
                                 # y : correct label for the input vector. format: a (1 or -1)

    if(X_vector[1] > target_f):  #if the input's vertical coordinate is above the target function, return 1 label
                                 #if the input's vertical coordinate is below the target function, return -1 label
        return 1
    else:
        return -1

def generate_random_point():   #generate random data point's coordinate
                               #inputs
                               # none
                               #outputs
                               # x: random points. format: [a,b]
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x


#Initializing the constants which will be used for training and testing
Total_iteration = 0          #Total iterations over all runs
Total_Run = 100               #Total number of experiments
Test_Data_Num = 1000         #Total number of testing data (used for testing the hypothesis g(x))
Total_Wrong_Points = 0       #Total number of f(x) != g(x) over all runs
Total_In_Unmatched_Num = 0    #Total number of incorrect data points for training samples.
Total_Out_Unmatched_Num = 0   #Total number of incorrect data points for testing data points.
```

```
OutOfSample_Num = 1000       #number of test data points for measuring out-of-sample error, Eout.


for run in range (0,Total_Run):
    N=100
    target_info = gen_target_func()     #target_info = [slope_target_func, y_intercept]

    x = np.zeros([N,4])       # x - [[1, x1,x2,label(y)],
                              #      [1, x1,x2,label(y)],
                              #      ..
                              #      [1, x1,x2,label(y)]]

    # generate N random data points with their correct labels based on the current target function f(x)
    for i in range (0, N):
        x[i,0] = 1                                    #x0 = 1
        x[i,1:3] = generate_random_point()            #random data points coordinate data
        f_x = target_info[0] * x[i,1] + target_info[1] #obtaining the target equation f
        x[i,3] = Label_data(x[i,1:3],f_x)             #using f, obtain the label(y) and append it to the array

    X = x[:,0:3]
    Y = x[:,3]

    X_pseudo_inverse = np.dot(np.linalg.inv(np.dot(X.T,X)),X.T)


    W = np.dot(X_pseudo_inverse,Y)

    g_output = np.zeros((N,1))
    for i in range (0,N):
        g_output[i] = np.sign(np.dot(W.T,X[i]))

    g_output = np.squeeze(g_output)


    insample_unmatched = 0
    for i in range (0,N):
        if g_output[i] != Y[i]:
            insample_unmatched = insample_unmatched + 1


    Total_In_Unmatched_Num = Total_In_Unmatched_Num + insample_unmatched

    #########################Testing begins to find Eout, the out-of-sample error.################################

    test_x = np.zeros([OutOfSample_Num,4])     # test_x -[[1, test_x1,test_x2,label(test_y)],
                                               #          [1, test_x1,test_x2,label(test_y)],
                                               #                       ..
                                               #          [1, test_x1,test_x2,label(test_y)]]

    #generate data points to meassure out of sample error, Eout.
    for i in range (0, OutOfSample_Num):
        test_x[i,0] = 1                                    #test_x0 = 1
        test_x[i,1:3] = generate_random_point()            #random data points coordinate data
        f_x = target_info[0] * test_x[i,1] + target_info[1] #obtaining the target equation f
        test_x[i,3] = Label_data(test_x[i,1:3],f_x)         #using f, obtain the label(y) and append it to the array

    test_X = test_x[:,0:3]
    test_Y = test_x[:,3]

    test_g_output = np.zeros((OutOfSample_Num,1))
    for i in range (0,OutOfSample_Num):
        test_g_output[i] = np.sign(np.dot(W.T,test_X[i]))

    test_g_output = np.squeeze(test_g_output)

    outofsample_unmatched = 0
    for i in range (0,OutOfSample_Num):                          #Test the performance of g by using test data.
        if test_g_output[i] != test_Y[i]:
            outofsample_unmatched = outofsample_unmatched + 1


    Total_Out_Unmatched_Num = Total_Out_Unmatched_Num + outofsample_unmatched


print("Avergae Ein: ", Total_In_Unmatched_Num/(N*Total_Run))   #Calculate the average of Ein over the entire test runs.
print("Average Eout: ", Total_Out_Unmatched_Num/(OutOfSample_Num*Total_Run)) #Calculate the average of Eout
```

7. Answer: [a]

The average iterations I got were 5.271. Please see the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a HW2 Problem 7

import numpy as np

def gen_target_func():          #generate a target function(f(x)) and return the corresponding vertical coordinate
                                #input: none
                                #output: target_function. format: [slope, y_intercept]
    rnd_x1 = np.zeros(2)
    rnd_x2 = np.zeros(2)

    for i in range (0,2):
        rnd_x1[i] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)

    for i in range (0,2):
        rnd_x2[i] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)

    slope_target_func = (rnd_x2[1]-rnd_x1[1])/(rnd_x2[0]-rnd_x1[0]) # slope = (y2-y1)/(x2-x1)

    y_intercept = rnd_x2[1]-slope_target_func*rnd_x2[0]

    return [slope_target_func, y_intercept]

def Label_data(X_vector, target_f): #return a correct label(1 or -1) by using the input vector and target equation f.
                                #inputs
                                # X_vector : input point's coordinate. format: [a, b]
                                # target_f : target function. format: a
                                #outputs
                                # y : correct label for the input vector. format: a (1 or -1)

    if(X_vector[1] > target_f):  #if the input's vertical coordinate is above the target function, return 1 label
                                #if the input's vertical coordinate is below the target function, return -1 label
        return 1
    else:
        return -1

def generate_random_point():   #generate random data point's coordinate
                               #inputs
                               # none
                               #outputs
                               # x: random points. format: [a,b]
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x

#Initializing the constants which will be used for training and testing
Total_iteration = 0          #Total iterations over all runs
Total_Run = 1000              #Total number of experiments
Test_Data_Num = 1000         #Total number of testing data (used for testing the hypothesis g(x))
Total_Wrong_Points = 0       #Total number of f(x) != g(x) over all runs
Total_In_Unmatched_Num = 0    #Total number of incorrect data points for training samples.
Total_Out_Unmatched_Num = 0   #Total number of incorrect data points for testing data points.
OutOfSample_Num = 1000       #number of test data points for measuring out-of-sample error, Eout.

for run in range (0,Total_Run):
    N=10
    target_info = gen_target_func()    #target_info = [slope_target_func, y_intercept]

    x = np.zeros([N,4])        # x - [[1, x1,x2,label(y)],
                             #     [1, x1,x2,label(y)],
                             #     ..
                             #     [1, x1,x2,label(y)]]

    w = np.zeros([3,1])        #initializing w vector
    w = np.squeeze(w)          #remove one dimension for matrix operations

    # generate N random data points with their correct labels based on the current target function f(x)
```

```
    for i in range (0, N):
        x[i,0] = 1                                  #x0 = 1
        x[i,1:3] = generate_random_point()            #random data points coordinate data
        f_x = target_info[0] * x[i,1] + target_info[1] #obtaining the target equation f
        x[i,3] = Label_data(x[i,1:3],f_x)             #using f, obtain the label(y) and append it to the array

    X = x[:,0:3]
    Y = x[:,3]

    X_pseudo_inverse = np.dot(np.linalg.inv(np.dot(X.T,X)),X.T)

    W = np.dot(X_pseudo_inverse,Y)

    g_output = np.zeros((N,1))
    for i in range (0,N):
        g_output[i] = np.sign(np.dot(W.T,X[i]))

    g_output = np.squeeze(g_output)


    insample_unmatched = 0
    for i in range (0,N):
        if g_output[i] != Y[i]:
            insample_unmatched = insample_unmatched + 1

    Total_In_Unmatched_Num = Total_In_Unmatched_Num + insample_unmatched

#Perceptron Learning Algorithm with the initial weights from Linear Regression begins.

    for i in range (0, N):
        x[i,0] = 1                                     #x0 = 1
        x[i,1:3] = generate_random_point()             #random data points coordinate data
        f_x = target_info[0] * x[i,1] + target_info[1] #obtaining the target equation f
        x[i,3] = Label_data(x[i,1:3],f_x)              #using f, obtain the label(y) and append it to the array

    iteration = 0
    mismatch = 0
    for i in range (0, 1000):
        random_gen = (int) (np.random.rand(1)*N)         #pick random misclassified points
        g_x = np.dot(W.T, x[random_gen,0:3])             #g(x) = dot(w,x)
        if(x[random_gen,3] != np.sign(g_x)):             #if y is not equal to the sign of g(x)
            W = W + x[random_gen,3]*x[random_gen,0:3]     # w = w + y*X
            iteration = iteration + 1

            ##perceptron end
    Total_iteration = Total_iteration + iteration       #Add up the number of iterations

print('Total iterations: ', Total_iteration)
print('Average iterations: ', Total_iteration / Total_Run)
```

8. Answer: [b]

The average in-sample error Ein I got was 0.118. Please see the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a HW2 Problem 8

import numpy as np

def generate_random_point():   #generate random data point's coordinate
                               #inputs
                               # none
                               #outputs
                               # x: random points. format: [a,b]
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x


#Initializing the constants which will be used for training and testing
Total_iteration = 0        #Total iterations over all runs
Total_Run = 1000            #Total number of experiments
```

```
Total_Unmatched_Num = 0

#Training begins
for run in range (0,Total_Run):
    N=1000

    x = np.zeros([N,4])       # x - [[1, x1,x2,label(y)],
                              #      [1, x1,x2,label(y)],
                              #            ..
                              #      [1, x1,x2,label(y)]]

    w = np.zeros([3,1])        #initializing w vector
    w = np.squeeze(w)          #remove one dimension for matrix operations

    # generate N random data points with their correct labels based on the current target function f(x)
    for i in range (0, N):
        x[i,0] = 1                              #x0 = 1
        x[i,1:3] = generate_random_point()       #random data points coordinate data
        f_x = x[i,1]**2 * x[i,2]**2 - 0.6        #f=sign(x1^2+x2^2-0.6)
        x[i,3] = np.sign(f_x)                   #using f, obtain the label(y) and append it to the array


    # generate random noise by flipping 1/10 of samples.
    for i in range (0, (int)(N/10)):
        random_index = (int) (np.random.rand(1)*1000)
        #print("rand index: " , random_index)
        x[random_index,3] = -x[random_index,3]

    X = x[:,0:3]
    Y = x[:,3]

    X_pseudo_inverse = np.dot(np.linalg.inv(np.dot(X.T,X)),X.T)

    W = np.dot(X_pseudo_inverse,Y)

    g_output = np.zeros((N,1))
    for i in range (0,N):
        g_output[i] = np.sign(np.dot(W.T,X[i]))

    g_output = np.squeeze(g_output)

    insample_unmatched = 0
    for i in range (0,N):
        if g_output[i] != Y[i]:
            insample_unmatched = insample_unmatched + 1

    #print("Ein: ", insample_unmatched/N)                   #Ein for each test.
    Total_Unmatched_Num = Total_Unmatched_Num + insample_unmatched

print("Avergae Ein: ", Total_Unmatched_Num/(N*Total_Run))   #Calculate the average of Ein over the entire test runs.
```

9. Answer: [e]

The vector W: [-0.96238404 -0.01416367 0.00634848 0.06723661 0.36162554 0.27927466]

Average error with hypothesis 1: 0.474957

Average error with hypothesis 2: 0.83427

Average error with hypothesis 3: 0.834625

Average error with hypothesis 4: 0.179067

Average error with hypothesis 5: 0.060891

Please see the code below for derivation

```
#Sung Hoon Choi
#CS/CNS/EE156a HW2 Problem 9

import numpy as np

def generate_random_point():   #generate random data point's coordinate
                               #inputs
                               # none
                               #outputs
                               # x: random points. format: [a,b]
```

```
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x

#Initializing the constants which will be used for training and testing
Total_iteration = 0            #Total iterations over all runs
Total_Run = 1000               #Total number of experiments
Total_hypothesis1_unmatched = 0
Total_hypothesis2_unmatched = 0
Total_hypothesis3_unmatched = 0
Total_hypothesis4_unmatched = 0
Total_hypothesis5_unmatched = 0

#Training begins
for run in range (0,Total_Run):
    N=1000

    x = np.zeros([N,4])      # x - [[1, x1,x2,label(y)],
                             #      [1, x1,x2,label(y)],
                             #            ..
                             #      [1, x1,x2,label(y)]]

    # generate N random data points with their correct labels based on the current target function f(x)
    for i in range (0, N):
        x[i,0] = 1                                #x0 = 1
        x[i,1:3] = generate_random_point()        #random data points coordinate data
        f_x = x[i,1]**2 * x[i,2]**2 - 0.6         #f=sign(x1^2+x2^2-0.6)
        x[i,3] = np.sign(f_x)                     #using f, obtain the label(y) and append it to the array

    # generate random noise by flipping 1/10 of samples.
    for i in range (0, (int)(N/10)):
        random_index = (int) (np.random.rand(1)*1000)
        x[random_index,3] = -x[random_index,3]

    # non-linear transformation
    non_linear_x = np.zeros([N,7])
    for i in range(0, N):
        non_linear_x[i,0] = x[i,0]
        non_linear_x[i,1] = x[i,1]
        non_linear_x[i,2] = x[i,2]
        non_linear_x[i,3] = x[i,1]*x[i,2]
        non_linear_x[i,4] = x[i,1]**2
        non_linear_x[i,5] = x[i,2]**2
        non_linear_x[i,6] = x[i,3]     #The label(y) attached to x array

    X = non_linear_x[:,0:6]
    Y = non_linear_x[:,6]

    non_linear_X_pseudo_inverse = np.dot(np.linalg.inv(np.dot(X.T,X)),X.T)

    W = np.dot(non_linear_X_pseudo_inverse,Y)

    g_output = np.zeros((N,1))
    for i in range (0,N):
        g_output[i] = np.sign(np.dot(W.T,X[i]))

    g_output = np.squeeze(g_output)

    #hyothesis [a]
    option1_g_output = np.zeros((N,1))
    option1_W = np.array([-1, -0.05, 0.08, 0.13, 1.5, 1.5])
    for i in range (0,N):
        option1_g_output[i] = np.sign(np.dot(option1_W.T, X[i]))
    option1_g_output = np.squeeze(option1_g_output)

    #hypothesis [b]
    option2_g_output = np.zeros((N,1))
    option2_W = np.array([-1, -0.05, 0.08, 0.13, 1.5, 15])
    for i in range (0,N):
        option2_g_output[i] = np.sign(np.dot(option2_W.T, X[i]))
    option2_g_output = np.squeeze(option2_g_output)

    #hypothesis [c]
    option3_g_output = np.zeros((N,1))
    option3_W = np.array([-1, -0.05, 0.08, 0.13, 15, 1.5])
```

```
    for i in range (0,N):
        option3_g_output[i] = np.sign(np.dot(option3_W.T, X[i]))
    option3_g_output = np.squeeze(option3_g_output)

    #hypothesis [d]
    option4_g_output = np.zeros((N,1))
    option4_W = np.array([-1, -1.5, 0.08, 0.13, 0.05, 0.05])
    for i in range (0,N):
        option4_g_output[i] = np.sign(np.dot(option4_W.T, X[i]))
    option4_g_output = np.squeeze(option4_g_output)

    #hypothesis [e]
    option5_g_output = np.zeros((N,1))
    option5_W = np.array([-1, -0.05, 0.08, 1.5, 0.15, 0.15])
    for i in range (0,N):
        option5_g_output[i] = np.sign(np.dot(option5_W.T, X[i]))
    option5_g_output = np.squeeze(option5_g_output)

    #compare with hypothesis [a]
    option1_g_output = np.squeeze(option1_g_output)
    hypothesis1_unmatched = 0
    for i in range (0,N):
        if g_output[i] != option1_g_output[i]:
            hypothesis1_unmatched = hypothesis1_unmatched +1

    #compare with hypothesis [b]
    option2_g_output = np.squeeze(option2_g_output)
    hypothesis2_unmatched = 0
    for i in range (0,N):
        if g_output[i] != option2_g_output[i]:
            hypothesis2_unmatched = hypothesis2_unmatched +1

    #compare with hypothesis [c]
    option3_g_output = np.squeeze(option3_g_output)
    hypothesis3_unmatched = 0
    for i in range (0,N):
        if g_output[i] != option3_g_output[i]:
            hypothesis3_unmatched = hypothesis3_unmatched +1

    #compare with hypothesis [d]
    option4_g_output = np.squeeze(option4_g_output)
    hypothesis4_unmatched = 0
    for i in range (0,N):
        if g_output[i] != option4_g_output[i]:
            hypothesis4_unmatched = hypothesis4_unmatched +1

    #compare with hypothesis [e]
    option5_g_output = np.squeeze(option5_g_output)
    hypothesis5_unmatched = 0
    for i in range (0,N):
        if g_output[i] != option5_g_output[i]:
            hypothesis5_unmatched = hypothesis5_unmatched +1

    Total_hypothesis1_unmatched = Total_hypothesis1_unmatched + hypothesis1_unmatched
    Total_hypothesis2_unmatched = Total_hypothesis2_unmatched + hypothesis2_unmatched
    Total_hypothesis3_unmatched = Total_hypothesis3_unmatched + hypothesis3_unmatched
    Total_hypothesis4_unmatched = Total_hypothesis4_unmatched + hypothesis4_unmatched
    Total_hypothesis5_unmatched = Total_hypothesis5_unmatched + hypothesis5_unmatched

print("my W: ", W)
print("Average error with current hypothesis 1: ", Total_hypothesis1_unmatched/(N*Total_Run))
print("Average error with current hypothesis 2: ", Total_hypothesis2_unmatched/(N*Total_Run))
print("Average error with current hypothesis 3: ", Total_hypothesis3_unmatched/(N*Total_Run))
print("Average error with current hypothesis 4: ", Total_hypothesis4_unmatched/(N*Total_Run))
print("Average error with current hypothesis 5: ", Total_hypothesis5_unmatched/(N*Total_Run))
```

10. Answer: [b]

The E_{out} I got was 0.121.

Please see the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a HW2 Problem 10
```

```
import numpy as np

def generate_random_point():   #generate random data point's coordinate
                               #inputs
                               # none
                               #outputs
                               # x: random points. format: [a,b]
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x

#Initializing the constants which will be used for training and testing
Total_iteration = 0            #Total iterations over all runs
Total_Run = 1000               #Total number of experiments
OutOfSample_Num = 1000        #number of test data points for measuring out-of-sample error, Eout.
Total_Out_Unmatched_Num = 0    #Total number of incorrect data points for testing data points.
#Training begins
for run in range (0,Total_Run):
    N=1000

    x = np.zeros([N,4])       # x - [[1, x1,x2,label(y)],
                              #      [1, x1,x2,label(y)],
                              #           ..
                              #      [1, x1,x2,label(y)]]

    # generate N random data points with their correct labels based on the current target function f(x)
    for i in range (0, N):
        x[i,0] = 1                              #x0 = 1
        x[i,1:3] = generate_random_point()       #random data points coordinate data
        f_x = x[i,1]**2 * x[i,2]**2 - 0.6        #f=sign(x1^2+x2^2-0.6)
        x[i,3] = np.sign(f_x)                   #using f, obtain the label(y) and append it to the array

    # generate random noise by flipping 1/10 of samples.
    for i in range (0, (int)(N/10)):
        random_index = (int) (np.random.rand(1)*1000)
        x[random_index,3] = -x[random_index,3]

    # non-linear transformation
    non_linear_x = np.zeros([N,7])
    for i in range(0, N):
        non_linear_x[i,0] = x[i,0]
        non_linear_x[i,1] = x[i,1]
        non_linear_x[i,2] = x[i,2]
        non_linear_x[i,3] = x[i,1]*x[i,2]
        non_linear_x[i,4] = x[i,1]**2
        non_linear_x[i,5] = x[i,2]**2
        non_linear_x[i,6] = x[i,3]     #The label(y) attached to x array

    X = non_linear_x[:,0:6]
    Y = non_linear_x[:,6]

    non_linear_X_pseudo_inverse = np.dot(np.linalg.inv(np.dot(X.T,X)),X.T)

    W = np.dot(non_linear_X_pseudo_inverse,Y)

    g_output = np.zeros((N,1))
    for i in range (0,N):
        g_output[i] = np.sign(np.dot(W.T,X[i]))

    g_output = np.squeeze(g_output)


    ##### Testing begins ####
    test_x = np.zeros([OutOfSample_Num,4])     # test_x -[[1, test_x1,test_x2,label(test_y)],
                                               #          [1, test_x1,test_x2,label(test_y)],
                                               #                      ..
                                               #          [1, test_x1,test_x2,label(test_y)]]

    #generate data points to measure out of sample error, Eout.
    for i in range (0, OutOfSample_Num):
        test_x[i,0] = 1                                #test_x0 = 1
        test_x[i,1:3] = generate_random_point()         #random data points coordinate data
        f_x = test_x[i,1]**2 * test_x[i,2]**2 - 0.6      #f=sign(x1^2+x2^2-0.6) #obtaining the target equation f
        test_x[i,3] = np.sign(f_x)                     #using f, obtain the label(y) and append it to the array
```

```
    test_non_linear_x = np.zeros([N, 7])
    for i in range(0, N):
        test_non_linear_x[i, 0] = x[i, 0]
        test_non_linear_x[i, 1] = x[i, 1]
        test_non_linear_x[i, 2] = x[i, 2]
        test_non_linear_x[i, 3] = x[i, 1] * x[i, 2]
        test_non_linear_x[i, 4] = x[i, 1] ** 2
        test_non_linear_x[i, 5] = x[i, 2] ** 2
        test_non_linear_x[i, 6] = x[i, 3]         #The label(y) attached to x array

    test_X = non_linear_x[:, 0:6]
    test_Y = non_linear_x[:, 6]

    test_g_output = np.zeros((OutOfSample_Num,1))
    for i in range (0,OutOfSample_Num):
        test_g_output[i] = np.sign(np.dot(W.T,test_X[i]))

    test_g_output = np.squeeze(test_g_output)

    outofsample_unmatched = 0
    for i in range (0,OutOfSample_Num):                         #Test the performance of g by using test data.
        if test_g_output[i] != test_Y[i]:
            outofsample_unmatched = outofsample_unmatched + 1

    Total_Out_Unmatched_Num = Total_Out_Unmatched_Num + outofsample_unmatched

print("my W: ", W)
print("Average Eout: ", Total_Out_Unmatched_Num/(OutOfSample_Num*Total_Run)) #Calculate the average of Eout
```