CS/CNS/EE 156a
Homework 5

Sung Hoon Choi
(Last name: Choi)

1. Answer: [c]

$$\sigma = 0.1, d = 8$$
$$E_D[E_{in}(w_{in})] = \sigma^2\left(1 - \frac{d+1}{N}\right) = 0.01\left(1 - \frac{9}{N}\right)$$

[a]: N=10: $E_D[E_{in}(w_{in})]$ = 0.001
[b]: N=25: $E_D[E_{in}(w_{in})]$ = 0.0064
[c]: N=100: $E_D[E_{in}(w_{in})]$ = 0.0091
[d]: N=500: $E_D[E_{in}(w_{in})]$ = 0.00982
[e]: N=1000: $E_D[E_{in}(w_{in})]$ = 0.00991
Thus, the answer is [c].

2. Answer: [d]

$$\mathrm{sign}(\widetilde{w}_0 + \widetilde{w}_1 {x_1}^2 + \widetilde{w}_2 {x_2}^2)$$

On the hyperbolic boundary given by the problem,, we can see that when x1 gets very large, the sign becomes -1(negative). Similarly, when x2 gets very large, the sign becomes +1 (positive). However, for x2, the function's output depends on the current value of x1, and $\widetilde{w}_0$ which could be adjusted to shape the desired boundary. Thus, if $\widetilde{w}_1$ is negative and $\widetilde{w}_2$ is positive, we can induce the given hyperbolic boundary by adjusting $\widetilde{w}_0$.

3. Answer: [c]

$$d_{vc} \le \tilde{d} + 1$$

Since d=14 for our given polynomial,

$$d_{vc} \le 14 + 1 = 15$$

Therefore, the answer is [c]

4. Answer: [e]
Apply the chain rule to the equation.

$$\mathrm{E[u,v]} = (\mathrm{u}e^v - 2ve^{-u})^2$$
$$\frac{\partial E}{\partial u} = 2(ue^v - 2ve^{-u})(e^v + 2ve^{-u})$$

5. Answer: [d]
The number of iterations taken were 10.
Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a HW5 Problem 5 and Problem 6
import math

def Error(u,v):
    return (u*math.exp(v)-2*v*math.exp(-u))**2

def gradient_u(u,v):
    return 2*(u*math.exp(v)-2*v*math.exp(-u))*(math.exp(v)+2*v*math.exp(-u))
```

```
def gradient_v(u,v):
    return 2*(u*math.exp(v)-2*v*math.exp(-u))*(u*math.exp(v)-2*math.exp(-u))

u=1
v=1
iteration = 0
learning_rate = 0.1
Max_iteration=50
initial_error = Error(u,v)

while (Error(u,v)>10**(-14)):
    iteration = iteration +1
    grad_u=gradient_u(u,v)
    grad_v=gradient_v(u,v)
    u = u - learning_rate*grad_u
    v = v - learning_rate*grad_v
    print("iteration: %d u: %f v: %f" %(iteration,u,v))

print("Taken iterations: %d u: %f v:%f" %(iteration,u,v)) #Answer for problem 5 and 6
```

6. Answer: [e]

The values I got were u=0.044736, v=0.023959 at iteration=10

Problem 5's code also gives the answer for Problem 6. (Same code)

```
#Sung Hoon Choi
#CS/CNS/EE156a HW5 Problem 5 and Problem 6
import math

def Error(u,v):
    return (u*math.exp(v)-2*v*math.exp(-u))**2

def gradient_u(u,v):
    return 2*(u*math.exp(v)-2*v*math.exp(-u))*(math.exp(v)+2*v*math.exp(-u))

def gradient_v(u,v):
    return 2*(u*math.exp(v)-2*v*math.exp(-u))*(u*math.exp(v)-2*math.exp(-u))

u=1
v=1
iteration = 0
learning_rate = 0.1
Max_iteration=50
initial_error = Error(u,v)

while (Error(u,v)>10**(-14)):
    iteration = iteration +1
    grad_u=gradient_u(u,v)
    grad_v=gradient_v(u,v)
    u = u - learning_rate*grad_u
    v = v - learning_rate*grad_v
    print("iteration: %d u: %f v: %f" %(iteration,u,v))

print("Taken iterations: %d u: %f v:%f" %(iteration,u,v)) #Answer for problem 5 and 6
```

7. Answer: [a]

The error E[u,v] after 15 iterations I got was 0.139814.

Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a HW5 Problem 7
import math

def Error(u,v):
    return (u*math.exp(v)-2*v*math.exp(-u))**2

def gradient_u(u,v):
    return 2*(u*math.exp(v)-2*v*math.exp(-u))*(math.exp(v)+2*v*math.exp(-u))

def gradient_v(u,v):
    return 2*(u*math.exp(v)-2*v*math.exp(-u))*(u*math.exp(v)-2*math.exp(-u))
```

```
u=1
v=1
iteration = 0
learning_rate = 0.1
Max_iteration=50
initial_error = Error(u,v)

for i in range (0,15):
    iteration = iteration +1
    u = u - learning_rate*gradient_u(u,v)
    v = v - learning_rate*gradient_v(u,v)

print("Taken iterations: %d Error: %f" %(iteration,Error(u,v))) #Answer for problem 7
```

8. Answer: [d]

By using Stochastic Gradient Descent algorithm, I got $E_{out} = 0.1003$

Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a HW5 Problem 8
import math
import random
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

def calculate_Error(N,weights,x):
    E_Sum=0
    for i in range (0,N):
        E_Sum = E_Sum + (math.log(1+math.exp(-x[i,3]*np.dot(weights.T,x[i,0:3]))))
```

```
    E_Sum = E_Sum/N
    return E_Sum

def calculate_Grad_Descent(weights,y,x):
    return ((-y * x)/ (1 + math.exp(y * np.dot(weights.T, x))))

N=100
Total_Run = 100
Total_Error = 0
weights = np.array([0,0,0])

#Generate training points with their labels using the target function.
for run in range (0,Total_Run):

    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


    #Calculate the g and its weights.
    weights_prev = np.array([5,5,5])
    Final_weights = np.array([0,0,0])
    weights = np.array([0,0,0])
    epoch = 0
    while (np.linalg.norm(weights-weights_prev)>0.01):
        weights_prev = weights
        for i in random.sample(range(0,N),N):
            weights = weights - 0.01*calculate_Grad_Descent(weights,x[i,3],x[i,0:3])
        epoch = epoch+1
        #print("epoch: ", epoch)

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

    Total_Error = Total_Error + calculate_Error(N,weights,x_test)

#Calculate the Eout
print("Average_Error:", Total_Error/Total_Run) #Answer for Problem 8
```

9. Answer: [a]

The average number of epochs that Logistic Regression took to converge was 334.

Please refer to the code below for derivation. (Almost same as the code for problem 8. Just added few lines to calculate the average number of epochs)

```
#Sung Hoon Choi
#CS/CNS/EE156a HW5 Problem 9
import math
import random
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

def calculate_Error(N,weights,x):
    E_Sum=0
    for i in range (0,N):
        E_Sum = E_Sum + (math.log(1+math.exp(-x[i,3]*np.dot(weights.T,x[i,0:3]))))

    E_Sum = E_Sum/N
    return E_Sum

def calculate_Grad_Descent(weights,y,x):
    return ((-y * x)/ (1 + math.exp(y * np.dot(weights.T, x))))

N=100
Total_Run = 100
Total_Error = 0
weights = np.array([0,0,0])
Total_Epoch_Sum = 0  #for Problem 9

#Generate training points with their labels using the target function.
for run in range (0,Total_Run):

    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


    #Calculate the g and its weights.
    weights_prev = np.array([5,5,5])
    Final_weights = np.array([0,0,0])
    weights = np.array([0,0,0])
    epoch = 0
    while (np.linalg.norm(weights-weights_prev)>0.01):
        weights_prev = weights
        for i in random.sample(range(0,N),N):
            weights = weights - 0.01*calculate_Grad_Descent(weights,x[i,3],x[i,0:3])
        epoch = epoch+1

    Total_Epoch_Sum = Total_Epoch_Sum + epoch

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

    Total_Error = Total_Error + calculate_Error(N,weights,x_test)

#Calculate the Eout
print("Average_Error:", Total_Error/Total_Run) #Answer for Problem 8
print("Average Epoch:", Total_Epoch_Sum/Total_Run) #Answer for Problem 9
```

10. Answer: [e]

For the Perceptron Learning Algorithm, when $w^T x_n$ and y_n do not match, $y_n w^T x_n$ is negative. Thus, we need to invert the error's sign to make the error positive when $w^T x_n$ and y_n do not match.

When $w^T x_n$ and y_n match, $y_n w^T x_n$ is positive, but in this case the error must be zero. (∵The hypothesis correctly predicted the output) Therefore, the answer is [e].