ENM808E
HOMEWORK-2

-Rachith Prakash

Wrote python scripts to simulate experiments(Scripts can be found in Appendix).

1. [b]
Explanation: Upon running the experiment as suggested in the question, an average vmin of 0.03 is being obtained. Hence, the closest answer is 0.01.

2. [d]
Explanation: This is a single bin case, hence, each coin's flips (10) corresponds to separate single bin. 10 tosses corresponds to 10 sample points. Thus, let v corresponds to in-sample error, where getting an head is considered error. I'm choosing e = 0.2 as e=0.1 gives Hoeffding bound as 1.6 which is not useful. For e=0.2, bound is 0.87. So, for every experiment, I'm checking if v1, vrand, vmin are within 'e' from Eout.
Now, for Eout, we know that each coin toss has a probability of 0.5 of getting head/tail (fair coin). Eout is calculated on these N = 10 samples as they are our bin. But, we know that Eout should be 0.5 i.e. in 10 coin tosses there is 0.5 probability of getting heads, i.e. 5 heads out of 10 --> 0.5 probability.
Over 1000 experiments, average vmin =1 , vrand = 0.11, v1 = 0.11. So, P[|v-Eout|>e] <= 0.87 is satisfied only by vrand and v1. Thus vmin does not satisfy the distribution.

3. [e]
Explanation: μ is the probability of error that 'h' makes in approximating 'f' (noiseless). Therefore (1-μ) is the probability of h getting it right. When noise is added to f as given, λ is the probability that given y=f, what is its probability of y=f after noise has been added. So, this becomes a conditional probability kinda case. So, given h makes an error of μ in approximating f, this f may not be the actual f, so it's probability of being correct is λ. So, probability of getting actual wrong is μ*λ. Also, probability of getting right is (1-μ) and this being wrong is (1-λ) giving probability of being wrong as (1-μ)*(1- λ). Thus, total probability is (1-μ)*(1- λ) + μ*λ.

4. [b]
Explanation:When λ = 0.5, we get the above probability as 0.5. Thus, getting right or wrong is of probability 0.5 and is independent of μ.

5. [c]
Explanation: Average Ein over 1000 experiments, for 100 samples was obtained around 0.05. Some examples of figures obtained are shown below.

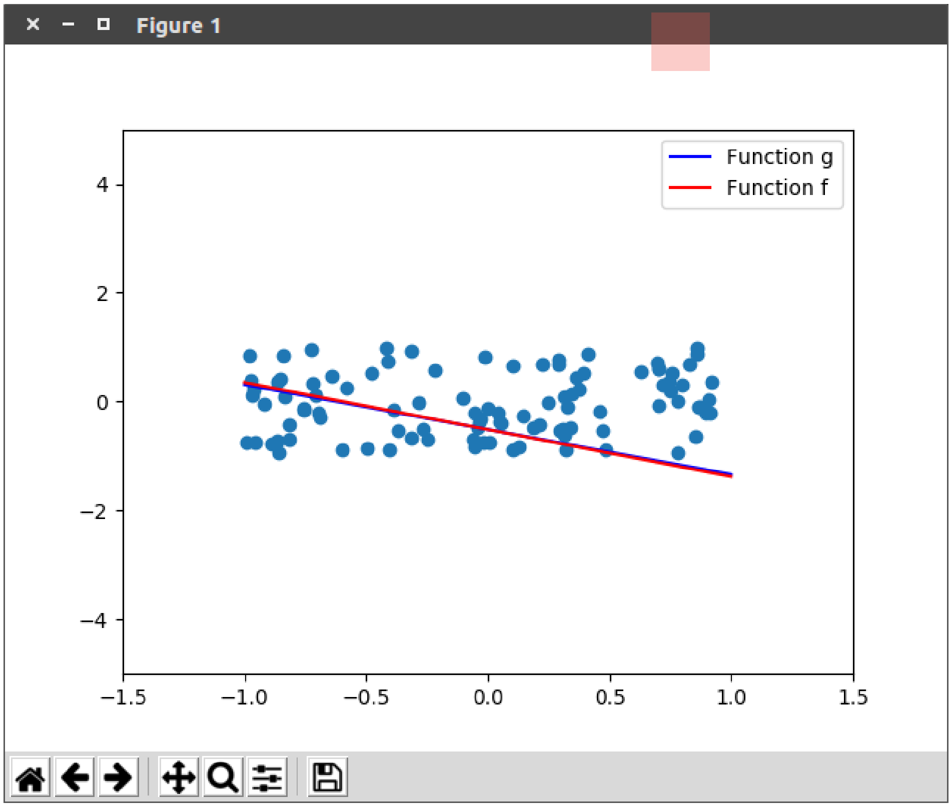


Figure 1
Function g
Function f
4
2
0
−2
−4
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

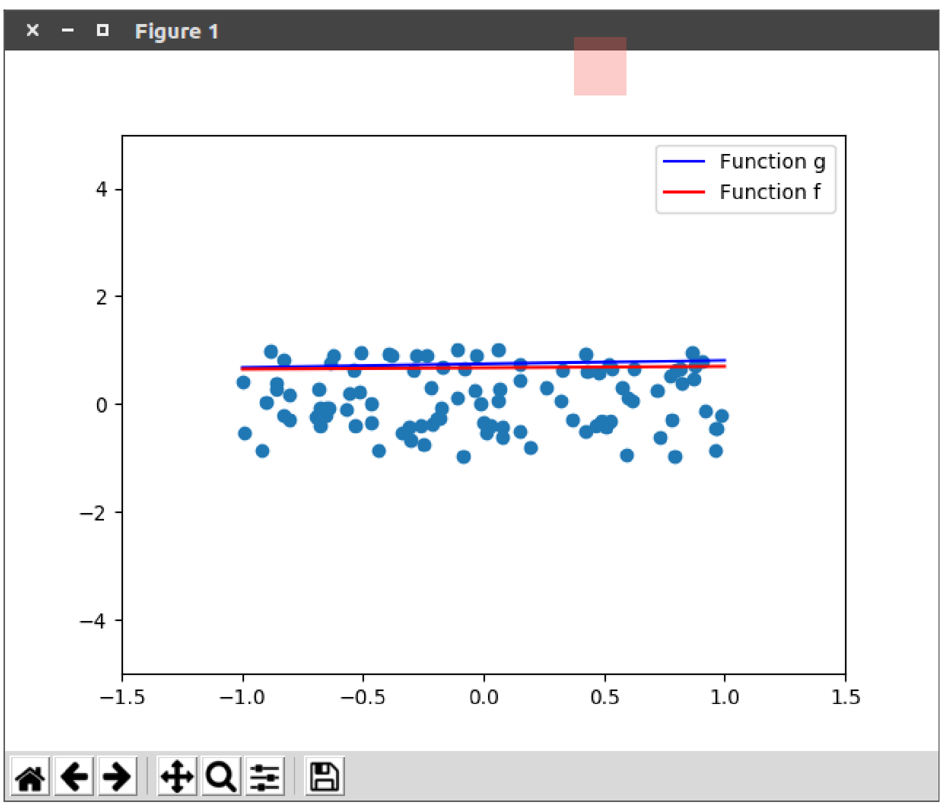


Figure 1
Function g
Function f
4
2
0
−2
−4
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

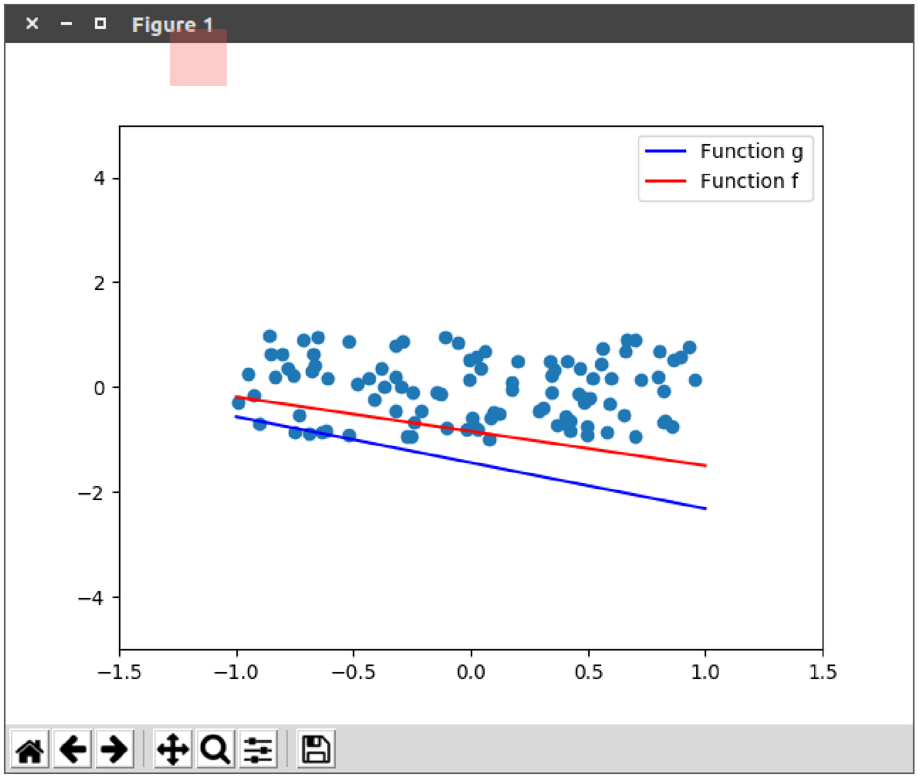


Figure 1
Function g
Function f
4
2
0
−2
−4
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

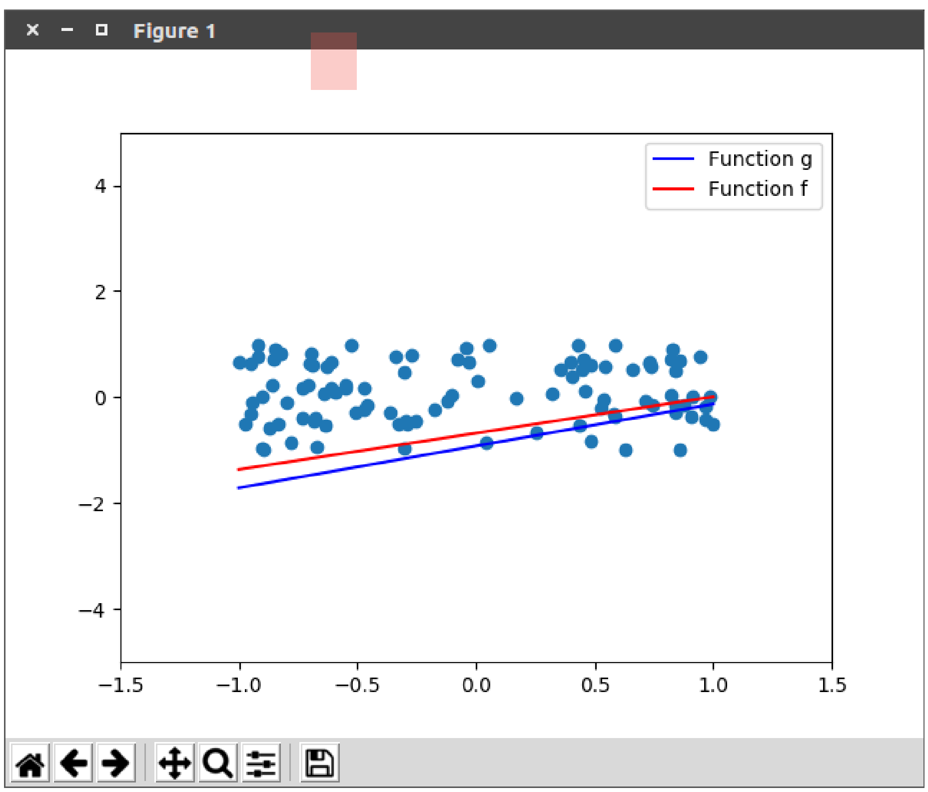
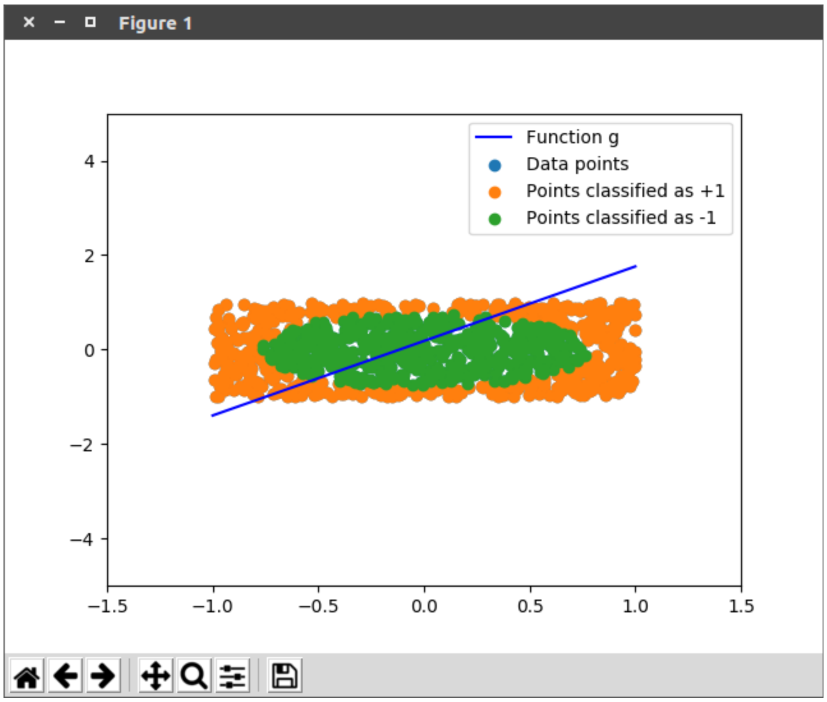


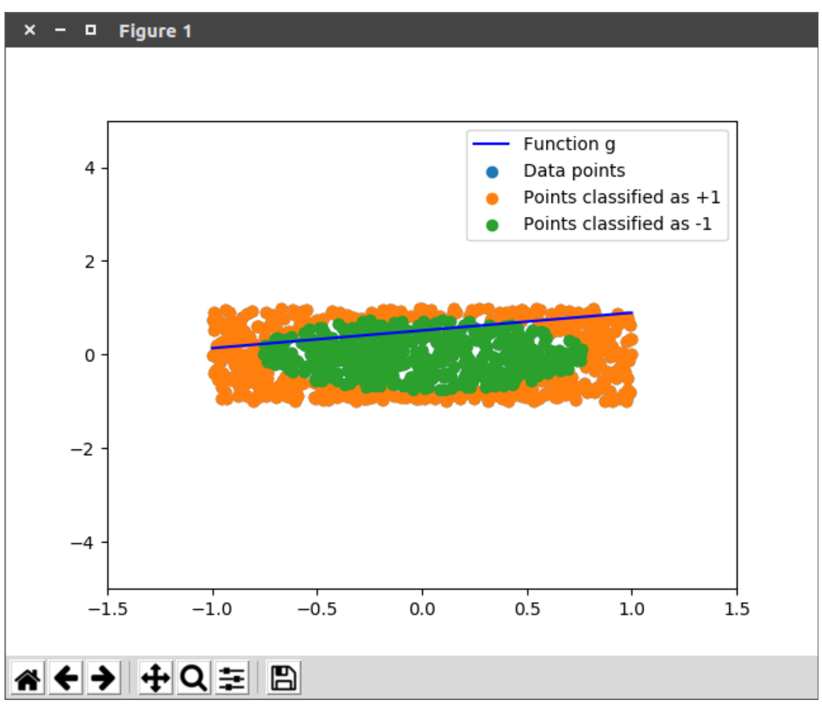
Figure 1
Function g
Function f
4
2
0
−2
−4
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

6. [d]
Explanation:This is always tracking Ein and has a value of around 0.06.

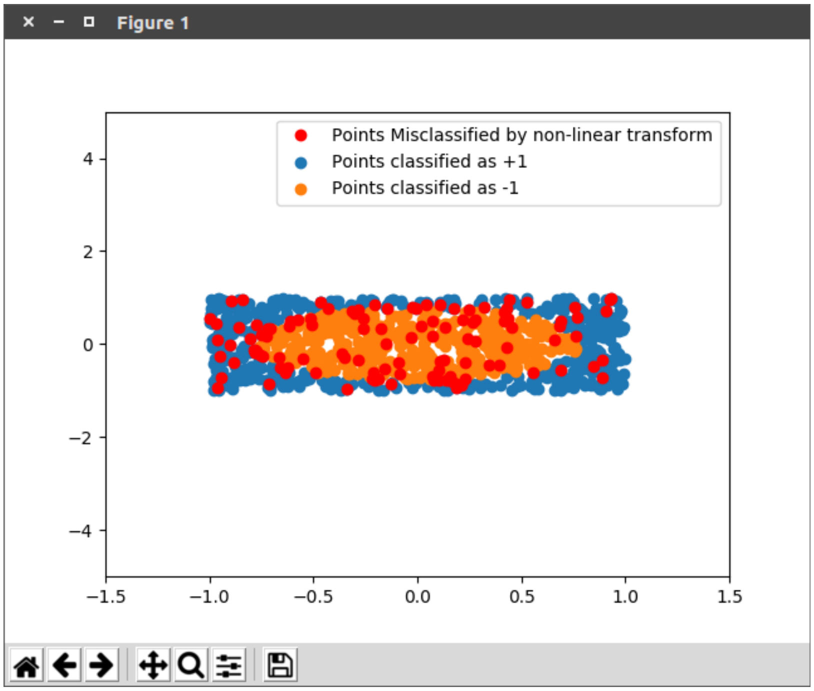
7. [a]
Explanation: Average number of iterations I'm getting is around 4 for the perceptron to converge with N = 10.

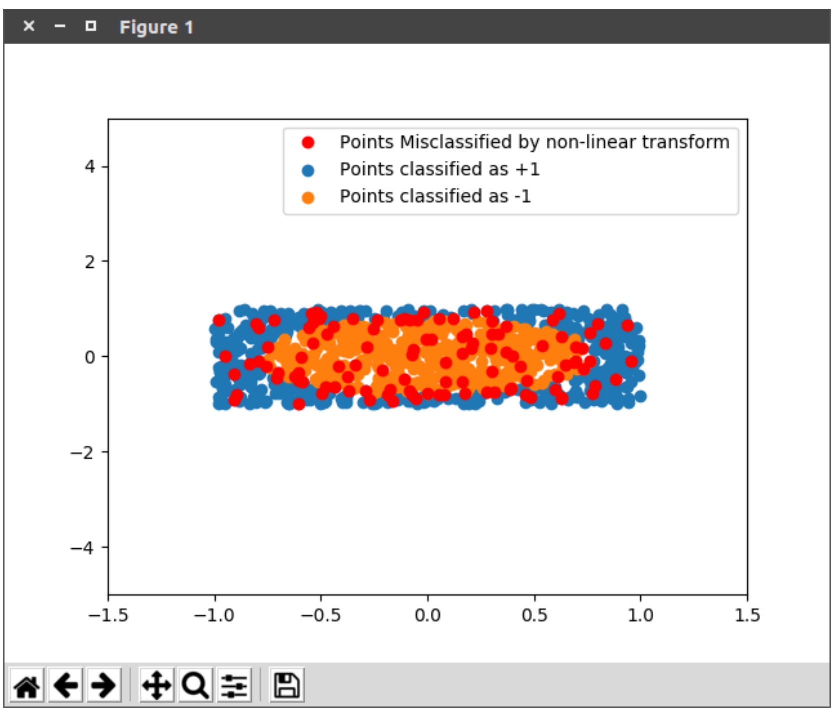
8. [d]
Explanation: Trying to fit a non-linear dataset with a perceptron does not work well and can be seen in the Ein result of 0.5 for the training dataset itself. Below figure shows the points being classified as +1 and -1 and how the linear regression line tries to fit it and fails miserably.





9. [a]
Explanation: Out of all the options given the first one agreed with my optimal set of 'w' which was [-1.015 -0.00179495 -0.000273 -0.000461 1.59 1.59]. Graph with linear classification is shown in figures below. Orange points are misclassified points. function 'f' is give by the blue line. In the below figures the points in red are being misclassified. This is shown for Ein.





10. [b]

Explanation: Getting Eout to be around 0.12. This tells us that after transforming the feature vector 1,x1, x2 into non-linear form (1,x1,x2,x1*x2,x1^2,x2^2), the classification was better using linear regression. Since the w_optimal approximates to a circle, the classification improves. But, since we are adding a noise of 10%, there will always be an error in misclassification of 10%. The same value of Ein is obtained for Ein of non-linear transform as we are having noise of 10%. Similar figure for Eout will be obtained as above.

Problem 3.9

Show that $E_{in}(w) = (w-(Z^TZ)^{-1}Z^Ty)^T(Z^TZ)(w-(Z^TZ)^{-1}Z^Ty) + y^T(I - Z(Z^TZ)^{-1}Z^T)y$

$\Rightarrow$ Since (Z^TZ) is invertible.

$$(Z^TZ)(Z^TZ)^{-1} = (Z^TZ)^{-1}(Z^TZ) = I$$

taking transpose gives $(Z^TZ)^T(Z^TZ)^{-T} = I^T \Leftrightarrow (Z^TZ)^{-T}(Z^TZ)^T = I$

$$\therefore E_{in}(w) = (w-(Z^TZ)^{-1}Z^Ty)^T\underbrace{(Z^TZ)(w-(Z^TZ)^{-1}Z^Ty)} + y^T(I - Z(Z^TZ)^{-1}Z^T)y$$

$$= (w-(Z^TZ)^{-1}Z^Ty)^T(Z^TZw - \underbrace{(Z^TZ)(Z^TZ)^{-1}}_{I}Z^Ty) + y^T(I - Z(Z^TZ)^{-1}Z^T)y$$

$$= (w-(Z^TZ)^{-1}Z^Ty)^T(Z^TZw - Z^Ty) + y^T(I - Z(Z^TZ)^{-1}Z^T)y$$

$$= w^T(Z^TZw - Z^Ty) - [y^TZ\underbrace{(Z^TZ)^{-T}(Z^TZ)}_{I}w] + y^TZ(Z^TZ)^{-T}Z^Ty + y^T(I - Z(Z^TZ)^{-1}Z^T)y$$

$$= w^TZ^TZw - w^TZ^Ty - y^TZw + \cancel{y^TZ(Z^TZ)^{-T}Z^Ty} + y^Ty - \cancel{y^TZ(Z^TZ)^{-1}Z^Ty}$$

$$E_{in}(w) = w^T(Z^TZ)w - w^TZ^Ty - (w^TZ^Ty)^T + y^Ty$$

NOTE: $(Z^TZ)^T = Z^TZ$ and $w^TZ^Ty = (w^TZ^Ty)^T$ as it is scalar

$$\therefore E_{in}(w) = w^T(Z^TZ)w - 2w^TZ^Ty + y^Ty \quad - (1)$$

Comparing this with $E_{in}(w) = \frac{1}{N}(w^TX^TXw - 2w^TX^Ty + y^Ty)$ — (2)

The 2 equations are equivalent (Ignoring $\frac{1}{N}$ scale)

Also, $Z = \Phi(X)$

Scanned by CamScanner

to obtain w_{lin} we need to take $\nabla_w E_{in}(w) = 0$ on ①

This gives $\cancel{2} Z^T Z w_{lin} - \cancel{2} Z^T y = 0$, as $Z^T Z$ is invertible

we get, $\therefore \boxed{w_{lin} = (Z^T Z)^{-1} Z^T y}$

But, we need to use the exp. given in Q.

Substituting this w_{lin} in the $E_{in}(w)$ expression as given in the question, we get

$$\boxed{E_{in}(w_{lin}) = y^T (I - Z(Z^T Z)^{-1} Z^T) y}$$

$\therefore$ E_{in} for optimal w_{lin} depends on input data and output vector only.

But, consider $Z(Z^T Z^{-1}) Z^T$ & NOTE that $Z((Z^T Z)^{-1}) Z^T \neq I$ as $Z^T Z \neq Z Z^T$. we do not know dimensions of the matrix 'Z'. though $Z^T Z$ and $Z Z^T$ are symmetric, they may not be equal. If they were equal, E_{in} would be 0.

```python
'''
This is the code for validating Hoeffding bound for single bin.
'''

import sys, os
import numpy as np
import random

# Heads = 1, tails = 0
coins = 1000
tosses = 10
experiments = 100000
sig_v1 = 0
sig_vrand = 0
sig_vmin = 0

# Since, N = 10, lets choose e=0.2 and e=0.3 to check Hoeffding inequality
# e=0.1 gives probability of bound as 1.6 which is not useful.
e = 0.2
E_out = 0.5 # Eout is 0.5 as 50% chance of getting an head or tail.

for i in range(experiments):
    stack = np.zeros((1,10), dtype=int)
    for _ in range(coins):
        P = []
        for _ in range(tosses):
            P.append(random.randint(0,1))
        stack = np.vstack((stack, P))

    stack = np.delete(stack,0,0)

    # c1
    c1 = stack[0,:]
    v1 = sum([1 for i in c1 if i == 1])/10.0
    if abs(v1-E_out)>e:
        sig_v1 += 1.0
    # C rand
    ind = random.randint(0,999)
    crand = stack[ind,:]
    vrand = sum([1 for i in crand if i == 1])/10.0
    if abs(vrand-E_out)>e:
        sig_vrand += 1.0

    #c min
    ind = np.argmin(np.sum(stack, axis=1))
    cmin = stack[ind,:]
    vmin = sum([1 for i in cmin if i == 1])/10.0
    if abs(vmin-E_out)>e:
        sig_vmin += 1.0


print 'Average probability of v1: ', sig_v1/experiments
print 'Average probability of vrand: ', sig_vrand/experiments
print 'Average probability of vmin: ', sig_vmin/experiments
```

```python
'''
This is the code to apply Linear regression for a dataset of dimension 2, each in [-1,1]. Then the optimal weight is used for PLA classification.
'''
import sys
import os
import numpy as np
import matplotlib.pyplot as plt
import random
from numpy.linalg import inv

def linearRegression():

    # Take N=100 or above for linear Regression
    N = 10
    Ein = 0
    Eout = 0
    p1 = [np.random.uniform(-1,1), np.random.uniform(-1,1)]
    p2 = [np.random.uniform(-1,1), np.random.uniform(-1,1)]

    # Find the equation of line
    a = p1[1]-p2[1]
    b = p2[0]-p1[0]
    d = -(a*p1[0]+b*p1[1])

    # Calculating slope and intercept
    m = -a/b
    c = -d/b
    y = []
    x = []

    for _ in range(N):
        xn = np.array([np.random.uniform(-1,1), np.random.uniform(-1,1)])
        x.append(xn)

        # Vertical distance is used for comparison
        if m*xn[0]+c > xn[1]:
            y.append(1)
        else:
            y.append(-1)


    x = np.array(x)
    x = np.hstack((np.ones((N,1)),x))
    y = np.array(y).reshape(N,1)
    w = np.matmul(np.matmul(inv(np.matmul(x.T, x)), x.T), y)
    m_hat = -w[1,0]/w[2,0]
    c_hat = -w[0,0]/w[2,0]
    y_hat = []
    for i in range(N):
        # Check every point's misclassification case
        if m_hat*x[i,1]+c_hat > x[i,2]:
            y_hat.append(1)
        else:
            y_hat.append(-1)

    cnt = [1.0 for i in range(N) if y[i,0] != y_hat[i]]
    Ein = sum(cnt)
    Eout = calculateEout(m, c, m_hat, c_hat)
    ite = PLA(N,x.T,y.T,w.T)

    # fig, ax = plt.subplots()
    # ax.scatter(x[:,1],x[:,2])
    # plotLines(m_hat, c_hat, 'b', "Function g")
    # plotLines(m, c, 'r', "Function f")
    # plt.legend()
    # plt.show()
    return Ein, Eout, ite


def calculateEout(m, c, m_hat, c_hat):
    # Taking fresh 1000 points to validate Eout
    cnt = 0
    for _ in range(1000):
        p = [np.random.uniform(-1,1), np.random.uniform(-1,1)]

        if m*p[0]+c > p[1]:
            y = 1
        else:
            y = -1

        if m_hat*p[0]+c_hat > p[1]:
            y_hat = 1
        else:
            y_hat = -1

        if y != y_hat:
            cnt += 1
    return cnt


def PLA(N,x,y,w):
    # PLA algorithm
    ite = 0
    val = False
    while not val:
        y_hat = np.matmul(w,x).reshape(1,N)
```

```python

        classify = [1 if y_hat[0,i]>0 else -1 for i in range(N)]

        misclassified = [1 if y[0,i]!=classify[i] else 0 for i in range(N)]

        ind = [i for i in range(N) if misclassified[i]==1]

        if not len(ind):
            val = True
            break

        rn = np.random.randint(0,len(ind))

        w = w + x[:,ind[rn]] * y[0,ind[rn]]

        ite += 1
    return ite

def plotLines(m, c, color,label):
    '''
    Pass slope and intercept value as input.
    '''
    plt.xlim(-1.5,1.5)
    plt.ylim(-5,5)
    plt.plot(np.linspace(-1,1),m*np.linspace(-1,1)+c, color, label = label)

def main():
    Ein = 0
    Eout = 0
    ite = 0
    for _ in range(1000):
        x, y, w = linearRegression()
        Ein += x
        Eout += y
        ite += w

    print 'Avg. Ein:', Ein/10000.0
    print 'Avg. Eout:', Eout/1000000.0
    print 'No. of iterations:', ite/1000.0

if __name__=="__main__":
    main()
```

```python
'''
This is the code to apply non-linear transform on the dataset to obatined non-linear feature vectors
and show that it is sometimes better to convert data first and do classification.
'''
import sys
import os
import numpy as np
import matplotlib.pyplot as plt
import random
from numpy.linalg import inv

def nonLinear():
    Ein = 0
    # 1000 data points
    N = 1000
    y = []
    x = []
    x_nt = []
    misclf = []
    cnt = 0
    pos = []
    neg = []

    # Calculation of data points
    for _ in range(1000):
        p = [np.random.uniform(-1,1), np.random.uniform(-1,1)]
        x.append(p)
        x_nt.append([1,p[0],p[1],p[0]*p[1],p[0]**2,p[1]**2]) # Non-linear transform is applied
        if p[0]**2 + p[1]**2 - 0.6 > 0:
            y.append(1)
            pos.append(p)
        else:
            y.append(-1)
            neg.append(p)
    # Storting + and - classified points
    pos = np.array(pos)
    neg = np.array(neg)
    # Generate noise for 10% of data by inverting it's value, chosen randomly
    for j in range(100):
        y[j] = y[j] * -1

    x = np.array(x)
    x = np.hstack((np.ones((N,1)),x))
    y = np.array(y).reshape(N,1)
    # Solve for w with linear regression.
    w = np.matmul(np.matmul(inv(np.matmul(x.T, x)), x.T), y)

    m_hat = -w[1,0]/w[2,0]
    c_hat = -w[0,0]/w[2,0]
    y_hat = []
    # Check for error in sample
    for i in range(N):
        # Check every point's misclassification case
        if m_hat*x[i,1]+c_hat > x[i,2]:
            y_hat.append(1)
        else:
            y_hat.append(-1)
    # Check for error in sample
    for i in range(N):
        if y[i,0] != y_hat[i]:
            cnt += 1
    Ein = cnt/1000.0 # E_in for linear regression.

    x_nt = np.array(x_nt)
    # Non-linear feature vector's linear regression.
    w_nt = np.matmul(np.matmul(inv(np.matmul(x_nt.T, x_nt)), x_nt.T), y)
    cnt = 0
    y_hat = np.matmul(x_nt,w_nt)
```

```
    pos_nt = []
    neg_nt = []
    for k in range(N):
        if y_hat[k] > 0:
            t = 1
            pos_nt.append([x_nt[k,1], x_nt[k,2]])
        else:
            t = -1
            neg_nt.append([x_nt[k,1], x_nt[k,2]])
        if y[k] != t:
            cnt += 1
            misclf.append([x_nt[k,1], x_nt[k,2]])
    pos_nt = np.array(pos_nt)
    neg_nt = np.array(neg_nt)
    misclf = np.array(misclf)
    Ein_nt = cnt/1000.0 # E_in for non-linear feature vector.

    # fig, ax = plt.subplots()
    # ax.scatter(pos[:,0], pos[:,1], label="Points classified as +1")
    # ax.scatter(neg[:,0], neg[:,1], label="Points classified as -1")
    # # plt.plot(x[:,1],x[:,2], 'go', label="Data points")
    # # plt.plot(pos_nt[:,0],pos_nt[:,1], 'r.', label="Non-Linear: Points classified as +1")
    # # plt.plot(neg_nt[:,0],neg_nt[:,1], 'b.', label="Non-Linear: Points classified as -1")
    # plt.plot(misclf[:,0],misclf[:,1], 'ro', label="Points Misclassified by non-linear transform")
    # # plotLines(m_hat, c_hat, 'b', "Function g")
    # plt.legend()
    # plt.xlim(-1.5,1.5)
    # plt.ylim(-5,5)
    # plt.show()
    Eout = calculateEout(w_nt)
    return Ein, Ein_nt, Eout

def calculateEout(w):
    cnt = 0
    x = []
    y = []
    N = 1000
    misclf = []
    for _ in range(N):
        p = [np.random.uniform(-1,1), np.random.uniform(-1,1)]
        x.append([1,p[0],p[1],p[0]*p[1],p[0]**2,p[1]**2])
        if p[0]**2 + p[1]**2 - 0.6 > 0:
            y.append(1)
        else:
            y.append(-1)
    # Adding noise to 100 points as before
    for i in range(100):
        y[i] = y[i] * -1

    x = np.array(x)
    y = np.array(y).reshape(N,1)
    y_hat = np.matmul(x,w)

    for i in range(N):
        if y_hat[i] > 0:
            y_hat[i] = 1
        else:
            y_hat[i] = -1

        if y[i] != y_hat[i]:
            misclf.append([x[i,1],x[i,2]])
            cnt += 1
    misclf = np.array(misclf)
    return cnt/1000.0

def main():

    Ein = 0
```

```
    Ein_nt = 0
    E_out = 0
    for _ in range(1000):
        x, y, z = nonLinear()
        Ein += x
        Ein_nt += y
        E_out += z
    print 'Ein linear for 1000 iterations:', Ein/1000.0
    print 'Ein non-linear for 1000 iterations:', Ein_nt/1000.0
    print 'Eout non-linearfor 1000 iterations:', E_out/1000.0


if __name__ == "__main__":
    main()
```