Homework 2

CS156, Kai Chang

Question 1

Answer Choice: **B**, 0.01
Reasoning: see the program below. Running it 100000 times took way to long, I have belief the value begins to converge to an appropriate stopping point at 10000 times.

In [19]:
```
import random as rnd

def run_sim():
    c_list = []
    for i in xrange(1000):
        h_count = 0
        for j in xrange(10):
            h_count += rnd.randint(0,1) # head = 1
        c_list.append(h_count)

    return c_list

v1 = []
vrand = []
vmin = []


for i in xrange(10000):
    c_list = run_sim()
    v1.append(c_list[0]/float(10))
    vrand.append(rnd.choice(c_list)/float(10))
    vmin.append(min(c_list)/float(10))

print 'vmin at 1 time: 0.1'
print 'vmin at 100 time: 0.039'
print 'vmin at 1000 time: 0.0378'
print 'vmin at 10k times: ', sum(vmin)/float(len(vmin))
```

```
vmin at 1 time: 0.1
vmin at 100 time: 0.039
vmin at 1000 time: 0.0378
vmin at 10k times:  0.0377
```

In [20]:
```
print 'vmin: ', sum(vmin)/float(len(vmin))
print 'v1: ', sum(v1)/float(len(v1))
print 'vrand: ', sum(vrand)/float(len(vrand))
```

```
vmin:  0.0377
v1:  0.49985
vrand:  0.49981
```

Question 2

Answer Choice: **D**, c_1 and c_{rand}
Reasoning: Using Hoeffding's Inequality as such: Consider $X_1, X_2, \ldots, X_n$ as independent real-valued random variables, such that for each i, X_i takes values from interval $[a_i, b_i]$. Let $Y := \sum_i X_i$. Then for all $\alpha > 0$,

$$Pr[|Y - E[Y]| \geq n\alpha] \leq 2\ exp(-\frac{2n^2\alpha^2}{\sum_i R_i^2})$$

where $R_i := b_i - a_i$.

So when we consider that $X_i \in [0, 1]$ and $E[X_i] = p$ because of our coin toss scenario, we get

$$Pr[|\sum_i X_i - p| \geq n\alpha] \leq 2\ exp(-\frac{2n^2\alpha^2}{n})$$

which simplifies to

$$Pr[|\sum_i X_i - p| \geq n\alpha] \leq 2\ exp(-2n\alpha^2)$$

or more intuitively

$$Pr[|\hat{p} - p| \geq \alpha] \leq 2\ exp(-2n\alpha^2)$$

So, given n = 10k in our sim, $\hat{p}$ = the probability we got. If $p \neq 0.5$, then we can say the coin is bias. So let $\alpha = \hat{p} - p = \{0.46168, 0.00015, 0.00019\}$ for $\{v_{min}, v_1, v_{rand}\}$. We don't really even have to do the math at this point to make an educated guess that both v_1 and v_{rand} are the two that satisfies the Hoeffding inequality. This makes sense conceptually because you do not put input any bias into your selection process (either the same source each time aka *having only one degree of random*, or randomize your source aka *having 2 degrees of random*).

Question 3

Answer Choice: **E**, $(1 - \lambda)(1 - \mu) + \lambda\mu$
Reasoning: To clarify, $P(y \mid \mathbf{x}) = \lambda$ for all y that satifies $y = f(x)$, and $P(y \mid \mathbf{x}) = 1 - \lambda$ is for all y that satifies $y \neq f(x)$. This is important because in order to get the probability that we get an error at predicting y (NOT F, now a 2-step process), we need to consider the following cases:

- h approximates the deterministic target function f wrong, and $y = f$
- h approximates f correctly, BUT $y \neq f$

So, summing the two probabilities yield: $\lambda\mu + (1 - \lambda)(1 - \mu)$.

Note the two other cases, ie. h gets f, $f = y$ and h doesn't get f, $f \neq y$, still means h approximated y correctly to a degree. If you don't get f and y is not equal to f, your h is still correct.

Question 4

Answer Choice: **B**, 0.5
Reasoning: You just need to have it such that the probability of error remains consistent regardless of μ, more specifically such that the probability of error and correctness are identical. Note that the first answer off the top of the mind may be 0, it would not be the case. Plugging in 0.5 yields the correct answer, because this results in eliminating μ in your error (ie. you are left with 0.5).

Question 5

Answer Choice: **C**, 0.01
Reasoning: see code.

In [232]:
```
import numpy as np
import random as rnd
import matplotlib.pyplot as plt

from numpy.linalg import inv
%matplotlib inline

def gen_line():
    '''
    Generate boundary line for classification

    Returns
    -------
    2 2-dimensional array consisting of your line in form [w0, w1, w2] and [w0, w
1_norm, w2_norm]

    '''
    [x1,x2,y1,y2] = [rnd.uniform(-1.0, 1.0), rnd.uniform(-1.0, 1.0), rnd.uniform(
-1.0, 1.0), rnd.uniform(-1.0, 1.0)]
    xA,yA,xB,yB = [rnd.uniform(-1, 1) for i in range(4)]
    w = np.array([x2*y1-y2*x1, y2-y1, x1-x2])
    w_norm = np.array([1, -w[1]/w[2], -w[0]/w[2]])
    return w, w_norm

def gen_pts(n, d, w=None, w_norm=None):
    '''
    Generates random points from a uniform distribution over -1,1

    Parameters
    ----------
    n : number of points
    d : dimension of image

    Returns
    -------
    d-dimensional array consisting of n-number of uniform, random points, and a c
lean slate sign
    '''
    if w is None:
        w, w_norm = gen_line()

    d_ = np.random.uniform(-1.0, 1.0,(d,n))
    x_ = np.append(np.ones(n), d_).reshape((d+1,n))
    y = np.sign(np.dot(w.T,x_))
    d_ = np.append(x_, y).reshape((d+2,n))
    return x_, y, w, d_, w_norm
```

In [233]:
```
def pre_process(n, d):
    '''
    Creates the necessary datasets and solutions needed to run a linear regressio
n classification

    Parameters
    ----------
    n : number of data points
    d : dimensions of dataset

    Returns
    -------
    x_  : coordinates or feature information (1, x1, x2)
    y   : solution from sign function
    w   : true weights (w0, w1, w2)
    d_  : entire dataset (incl. solution)
    w_n : normalized weights, ie. (w0=1, w1, w2)
    '''
    x_, y, w, d_, w_n = gen_pts(n,d)

    return x_, y, w, d_, w_n


def lin_reg(x_, y):
#     print x_.shape
#     print x_.T.shape
    w = np.dot( np.dot( inv(np.dot(x_.T, x_)), x_.T), y)
#     print 'weights: ', w
#     print w.shape
#     print x_.shape
#     print w.T.shape
#     print x_.T.shape
#     print y.shape
    y_ = np.sign(np.dot(x_, w))
    correct = y_ - y
    E_in = np.count_nonzero(correct) / float(x_.shape[0])

    return w, E_in
```

In [137]:
```
# Question 5
E_in_list = []
n = 100
d = 2
for i in xrange(1000):
    x_, y, w, d_, w_n = pre_process(n, d)
    x1_ = []

    for i in xrange(len(x_[0])):
        x1_.append([x_[0][i],x_[1][i],x_[2][i]])
    x_ = np.array(x1_)

    w_, E_in = lin_reg(x_, y)
    E_in_list.append(E_in)

avg_E_in = sum(E_in_list) / float(len(E_in_list))
print 'Average E_in: ', avg_E_in
```

```
Average E_in:  0.03977
```

Question 6

Answer Choice: **C**, 0.01
Reasoning: see code.

In [140]:
```
# Question 6
E_out_list = []
n = 1000
d = 2
for i in xrange(1000):
    x_, y, _, _, _ = gen_pts(n, d, w=w_, w_norm=w_n)
    x1_ = []

    for i in xrange(len(x_[0])):
        x1_.append([x_[0][i],x_[1][i],x_[2][i]])
    x_ = np.array(x1_)

    # Calculate E_out
    y_ = np.sign(np.dot(x_, w))
    correct = y_ - y
    E_out = np.count_nonzero(correct) / float(x_.shape[0])

    E_out_list.append(E_out)

avg_E_out = sum(E_out_list) / float(len(E_out_list))
print 'Average E_out: ', avg_E_out
```

```
Average E_out:  0.008489
```

Question 7

Answer Choice: **A**, 1
Reasoning: see code. Runs previous had max values around 1.1

In [155]:
```
def pick_pt(y_, y):
    '''
    Find misclassified points and pick one at random.

    Parameters
    ----------
    y_ : list of all output points from our updated weight
    y  : list of correct output points

    Returns
    -------
    index of random point, number of misclassified points
    '''
    mc_pts = []
    for i in xrange(len(y)):
        if y_[i] != y[i]:
            mc_pts.append(i)

    try:
        index = rnd.choice(mc_pts)
    except IndexError:
        index = 0

    return index, len(mc_pts)


def update(xi, yi_, w_):
    '''
    Takes a misclassified point and updates the weight to correctly classify poin
t

    Parameters
    ----------
    xi   : incorrectly classified point
    yi_  : correct sign for point
    w_   : current weight

    Returns
    -------
    updated weight
    '''
    w_ += yi_ * xi

    return w_


def pla(w_=None):
    if w_ is None:
        w_ = np.zeros(3)

    y_ = np.sign(np.dot(x_, w_))
    i = 0 # iterations

    while np.array_equal(y, y_) != True:
        index, total_mc_pts= pick_pt(y_,y)
        w_ = update(x_[index], y[index], w_)
        y_ = np.sign(np.dot(x_, w_))
        i += 1
        if i%1000 == 0:
            break
    w_n = np.array([1, -w_[1]/w_[2], -w_[0]/w_[2]])

    return i, w_n
```

In [159]:
```
# Question 7
iters = []
n = 10
d = 2
for i in xrange(1000):
    x_, y, _, _, _ = gen_pts(10, d, w, w_n)
    x1_ = []

    for i in xrange(len(x_[0])):
        x1_.append([x_[0][i],x_[1][i],x_[2][i]])
    x_ = np.array(x1_)

    it, _ = pla(w_=w_)
    iters.append(it)

avg_iter = sum(iters) / float(len(iters))
print 'iterations: ', avg_iter
```

```
iterations:  0.116
```

Question 8

Answer Choice: **D**, 0.5

Reasoning: see code. *Note, we did not consider the fact that we could have gotten repeated indices in our RNG*

In [181]:
```
def gen_y(x_):
    '''
    Generates target function results

    Parameters
    ----------
    x_ : coordinates (1, x1, x2)

    Returns
    -------
    f(x1, x2) = sign(x1^2 + x2^ - 0.6)
    '''
    return np.sign(float(x_[1])**2 + float(x_[2])**2 - 0.6)

def gen_noise(v, y):
    '''
    Generates noise in training data

    Parameters
    ----------
    v : noise (uniformly randomly selected)
    y : our training data solution

    Returns
    -------
    corrupted data
    '''
    y_noise = y.copy()
    v_i = v * y_noise.shape[0]

    mask = np.random.randint(0, y_noise.shape[0], v_i)

    for m in mask:
        y_noise[m] = -y_noise[m]

    return y_noise
```

In [186]:
```
# Question 8
E_in_list = []
n = 1000
d = 2
v = 0.1

for i in xrange(1000):
    # generates points
    d_ = np.random.uniform(-1.0, 1.0,(d,n))
    x_ = np.append(np.ones(n), d_).reshape((d+1,n))

    x1_ = []
    for i in xrange(len(x_[0])):
        x1_.append([x_[0][i],x_[1][i],x_[2][i]])
    x_ = np.array(x1_)

    y = []
    for coord in x_:
        y.append(gen_y(coord))
    y = np.array(y)

    y_corr = gen_noise(v, y)

    w_, E_in = lin_reg(x_, y_corr)
    E_in_list.append(E_in)

avg_E_in = sum(E_in_list) / float(len(E_in_list))
print 'Average E_in: ', avg_E_in
```

```
/Users/kaichang/anaconda/lib/python2.7/site-packages/ipykernel/__main__.py:31: V
isibleDeprecationWarning: using a non-integer number instead of an integer will
result in an error in the future

Average E_in:  0.50482
```

Question 9

Answer Choice: **A**, $g(x_1, x_2) = \text{sign}(-1 - 0.05x_1 + 0.08x_2 + 0.13x_1x_2 + 1.5x_1^2 + 1.5x_2^2)$
Reasoning: See code (specifically the x_1^2 and x_2^2 values)

In [230]:
```
w_list = []
n = 1000
d = 2
v = 0.1

for i in xrange(5):
    # generates points
    d_ = np.random.uniform(-1.0, 1.0,(d,n))
    x_ = np.append(np.ones(n), d_).reshape((d+1,n))

    # reorganize points
    x1_ = []
    for i in xrange(len(x_[0])):
        x1_.append([x_[0][i], x_[1][i], x_[2][i], x_[1][i]*x_[2][i], x_[1][i]**2,
 x_[2][i]**2])
    x_ = np.array(x1_)

    y = []
    for coord in x_:
        y.append(gen_y(coord))
    y = np.array(y)

    y_corr = gen_noise(v, y)


    w_, _ = lin_reg(x_, y_corr)
    w_list.append(w_)

w_list = np.array(w_list).sum(axis=0)
print w_list/len(w_list)
```

```
[-0.83902182  0.00473102 -0.03087171  0.00364716  1.34535581  1.31231759]
```

```
/Users/kaichang/anaconda/lib/python2.7/site-packages/ipykernel/__main__.py:31: V
isibleDeprecationWarning: using a non-integer number instead of an integer will
result in an error in the future
```

Question 10

Answer Choice: **B**, 0.1
Reasoning: see code.

In [231]:
```
# Question 6
E_out_list = []
n = 1000
d = 2
w = w_list/len(w_list)

for i in xrange(1000):
    # generates points
    d_ = np.random.uniform(-1.0, 1.0,(d,n))
    x_ = np.append(np.ones(n), d_).reshape((d+1,n))

    # reorganize points
    x1_ = []
    for i in xrange(len(x_[0])):
        x1_.append([x_[0][i], x_[1][i], x_[2][i], x_[1][i]*x_[2][i], x_[1][i]**2,
 x_[2][i]**2])
    x_ = np.array(x1_)

    # creates corrupted output
    y = []
    for coord in x_:
        y.append(gen_y(coord))
    y = np.array(y)
    y_corr = gen_noise(v, y)

    # Calculate E_out
    y_ = np.sign(np.dot(x_, w))
    correct = y_ - y_corr
    E_out = np.count_nonzero(correct) / float(x_.shape[0])

    E_out_list.append(E_out)

avg_E_out = sum(E_out_list) / float(len(E_out_list))
print 'Average E_out: ', avg_E_out
```

```
/Users/kaichang/anaconda/lib/python2.7/site-packages/ipykernel/__main__.py:31: V
isibleDeprecationWarning: using a non-integer number instead of an integer will
result in an error in the future

Average E_out:  0.111165
```