# Homework 1, CS156a

***Kai Chang***

## Question 1

Answer choice: **D** -- (i) Not Learning, (ii) Supervised Learning, (iii) Reinforcement Learning

Reasoning:
For choice (i), this is not considered Machine Learning because you are taking known parameters or feature values of the coins and matching them statistically with your sample. In this case, your necessary or classification features have been already set, not found.

For choice (ii), this is machine learning because in this case, you don't know the correct classification features or exact values needed to classify the coins into the appropriate groupings. You have samples in which you are trying to infer the correct grouping by the feature information given from your large set of labeled data. It is supervised because you know the correct classification or answer.

For choice (iii), this is machine learning because again, you don't know the exact approach or process to obtain a winning strategy -- you are essentially learning by trial and error. This is reinforced because you adjust your learning over time with respect to your successes (in this case Validation cost = win %). Reinforcement elarning changes your $\alpha$ such that your $w$ and $b$ adjust approriately over time.

## Question 2

Answer choice: **A** -- (ii) and (iv)

Reasoning:
For choice (i), there is already an solution for it without using ML, ie. it is mathematically solvable, so machine learning would not offer any benefits. Thus, we rule out choice (i).

Choice (iii) is also exact and has a fixed solution within realms, and like choice (i), would not be effective for ML usage.

(ii) and (iv) are the best problems suited for ML because there are no relatively easy exact formulas to solve our problem and would take a lot of time, so ML is best suited to learn the necessary features. In this case, we don't necessary know how to definie our classification parameters and which parameters are significant or nonessential in solving our problems, so we feed in as much information as we can and allow our computer to classifiy and form appropriate groupings of our similar traits.

## Question 3

Answer choice: **D** -- 2/3

Originally, I had thought using conditional probability that because 1 bag is $\frac{1}{2}$ (b,b) and the other is $\frac{1}{2}$ (b,w), that $P = \frac{1}{2}$ because once you pick a bag $\frac{1}{2}$, then you automatically know if you win or lose. However, I completely neglected the conditional aspect, ie. the first is black (has to be black). This rules out any of the probability that the first ball is white, which would alter our probability (total outcome pool).

My first thoughts (because I needed a refresher on stats) is to run this scenario through a python simulation:

In [2]:
```
import copy
import random

# Parameters
bag1 = ['b','b']
bag2 = ['b','w']

# Question 3
def pick_ball(bag):
    return random.choice(bag)

def pick_bag(bag_set):
    return random.choice(bag_set)

def select(bag_set):
    bag = pick_bag(bag_set)
    return bag, pick_ball(bag)

b_black = 0 # counts of black ball being picked after 1st black ie. P(A|B)
n = 1000000 # total trials in simulation
for i in xrange(n):
    b1 = copy.deepcopy(bag1)
    b2 = copy.deepcopy(bag2)
    b_set = [b1,b2]
    ball = 'w'

    while ball != 'b':
        bag, ball = select(b_set)

    # get first bag
    bag.remove('b')
    if bag[0] == 'b':
        b_black += 1

print 'Probability of second ball black: ', float(b_black)/n
```

```
Probability of second ball black:  0.667124
```

Above, using 1 million simulations, we get $P \approx \frac{2}{3}$. However after thinking mathematically, this was a clear useage of Baye's theorem.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

and

- $P(A|B)$, probability we choose bag (b,b), given the first ball we chose is b
- $P(B|A) = 1$, probability the first ball is b, given we chose bag (b,b)
- $P(A) = \frac{1}{2}$, probability we choose bag (b,b)
- $P(B) = \frac{3}{4}$, probability that the first ball is b

So, we get

$$\begin{aligned} P(A|B) &= \frac{1 \times \frac{1}{2}}{\frac{3}{4}} \\ &= \frac{2}{3} \end{aligned}$$

## Question 4

Answer choice: **B**, $3.405 \times 10^{-4}$

Given 10 marbles, we draw one sample. The chance that $\nu = 0$ or we get 0 chance of red marbles has is such that every marble in the pot is green. Thus, because $\mu = 0.55$, or a chance a marble is red (independent and with replacement!), then we calculate P for 10 marble cases.

So, $P = (1 - 0.55) = 0.45$ for green pull on a single marble, and $P = 0.45^{10} = 0.003405$ for green pulls always on 10 marbles

## Question 5

Answer choice: **C**, 0.289

Now, given that for a 1000 *independent* samples, we pull at least 1 $\nu = 0$ means that we can have at 1 sample, 2 samples, 3 samples all the way up to all the samples having $\nu = 0$. This is equivalent to taking the probability of
$1 - P(\text{we get all red marbles for 1000 samples})$.

This is equal to $1 - P(0\ \nu \text{ or all } \mu)$, which is $1 - (1 - P(\text{all red in a single sample}))^{1000}$, and this is
$1 - (1 - 0.0003403)^{1000} = 0.289$.

## Question 6

Answer choice: **E**, They are all equivalent (equals cores for *g* in a through d).

Reasoning: For (a), following the 2D example on the edX forum, using g = 1, score for the three remaining points are

$$1 \times 3 + 3 \times 2 + 3 \times 1 + 1 \times 0 = 12$$

For (b), g = 0, score is

$$1 \times 3 + 3 \times 2 + 3 \times 1 + 1 \times 0 = 12$$

These two are identical, meaning the only valid solution is the last one.

We note that the possible $y_n$ outcomes or target functions for all cases are pallandromic and follow combinatoric patterns, where they contain the same unique 3 point identicals, 2 point identicals, and 1 point identicals. For a and b, these are identical but inverse in values, and for c and d, this is the same. So, if it works on one case, it must work on the other. Thus, we realize that **E** must be the solution.

## Question 7

Answer choice: **B**, 15

See attached code and afterwords

In [7]:
```
import numpy as np
import random as rnd
import matplotlib.pyplot as plt
%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

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)
```

In [16]:
```
# Question 7
iters = []
n = 10
d = 2
for i in xrange(1000):
    x_, y, w, d_, w_n = pre_process(n, d)
    it, _ = pla()
    iters.append(it)

avg_iter = sum(iters) / float(len(iters))
print 'iterations away from 15:', 15-avg_iter, ' iterations away from 1:', abs(1-
avg_iter)
```

```
iterations away from 15: 4.633  iterations away from 1: 9.367
```

***Afternotes:***

I originally struggled for hours reasoning why my linear line was not correctly plotting with my PLA Classification values. My weights had 2-dimensions {w0, w1}, and either resulted in a linear shift or a transpose shift. They were never constant and it threw me off.

Eventually I remembered about the threshold term on the weight, in which Andrew Ng corresponded it to what usually people would put as 1. This made our weight 3-D in {w0, w1, w2}. This I think correspondes to the additional fact that we cannot just absorb the w2 term into our w0 and w1 term from w0 + w1x + w2y = 0. This artificial term is needed and is often overstepped by convention. Without it, we couldn't have a true intercept term or bias term, and with a transpose would really change our perception of our data.

If you have a better or (correct) explanation, please do share.

Note we know the exact solution, so this case we don't need to have a validation error and stop the program short if it fluctuates. We simply just keep processing until we get 100% accuracy.

**Justification for weight and normalized weight**

We use $w = (w_0, w_1, w_2) = (x_2y_1 - x_1y_2,\ y_2 - y_1,\ x_2 - x_1)$. Why?

Consider the generalized 2D linear equation to be $w_0 + w_1x + w_2y = 0$, and we have two points $(x_1, y_1)$, $(x_2, y_2)$.

Then, using the linear equation in two-point form (https://en.wikipedia.org/wiki/Linear_equation#Two-point_form (https://en.wikipedia.org/wiki/Linear_equation#Two-point_form)), we can derive $y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$.

This can be re-written as $(x_2 - x_1)(y - y_1) = (y_2 - y_1)(x - x_1)$ and can be expanded to the form $x_2y_1 - x_1y_2 + (y_2 - y_1)x - (x_2 - x_1)y = 0$.

So, we then get

- $w_0 = x_2y_1 - y_2x_1$
- $w_1 = y_2 - y_1$
- $w_2 = x_1 - x_2$

## Question 8

Answer choice: **C**, 0.1

Reasoning: See code. Also, I put in 1 training point too, and got P~0.5, but I dont think this is correct because we have to keep the same sample size per trial.

In [17]:
```
disagreement = []
n = 10
d = 2
for i in xrange(1000):
    x_, y, w, d_, w_n = pre_process(n, d)
    _, w_n_ = pla()
    x_, y, _, _, _ = gen_pts(10, d, w, w_n) # 10 training points (ie so we have 1
0000 samples from 10 point * 1000 trials)
    y_ = np.sign(np.dot(w_n_.T,x_))
    zzz, nmc = pick_pt(y_, y)
    if nmc > 0:
        disagreement.append(nmc)

print 'probability of disagreement for N=10: ', len(disagreement)/float(10000)
```

```
probability of disagreement for N=10:  0.0971
```

## Question 9

Answer choice: **B**, 100

Reasoning: see code. It shouldn't ever really be larger than the total trials for any usable algorithm however bad, so it ruled out 1000 and 5000.

In [18]:
```
iters = []
n = 100
d = 2
for i in xrange(1000):
    x_, y, w, d_, w_n = pre_process(n, d)
    it, _ = pla()
    iters.append(it)

avg_iter = sum(iters) / float(len(iters))
print 'iterations away from 50:', abs(50-avg_iter), ' iterations away from 100:',
 abs(100-avg_iter), ' iterations away from 500: ', abs(500-avg_iter)
```

```
iterations away from 50: 46.941  iterations away from 100: 3.059  iterations awa
y from 500:  403.059
```

## Question 10

Answer choice: **B**, 0.01

Reasoning: see code. It should be less than the 10 training points set because with more data there is more information and thus less chance for validation errors. More sample statistics almost always lead to a better resolution. This rules out (d), (e).

In [20]:
```
disagreement = []
n = 100
d = 2
for i in xrange(1000):
    x_, y, w, d_, w_n = pre_process(n, d)
    _, w_n_ = pla()
    x_, y, _, _, _ = gen_pts(100, d, w, w_n) # 10 training points (ie so we have
10000 samples from 10 point * 1000 trials)
    y_ = np.sign(np.dot(w_n_.T,x_))
    zzz, nmc = pick_pt(y_, y)
    if nmc > 0:
        disagreement.append(nmc)

print 'probability of disagreement for N=10: ', len(disagreement)/float(100000)
```

```
probability of disagreement for N=10:  0.00999
```