Assignment #3

Group 4

Lars Wrede, Dennis Blaufuss, Nicolas Kepper, Sophie Merl, Philipp Voit

---

Unlucky Optimization Competition

Modify the Expectation-Maximization (EM) Python program for the two coins, as discussed in class.

(a) Generate unrepresentative series of exactly n = 125 total coin flips (5 times 25 flips with a randomly selected coin),

(b) given two coins are chosen with equal probability (1/2).

(c) The coins must have the following heads biases, θ_A = 0.9 (coin A), and θ_B = 0.1 (coin B), and the generation must follow exactly this random process.

In a nutshell, the task is to generate a series of (H)eads and (T)ails that are highly unlikely, given the ground truth, by brute forcing an unlucky realization by massive repetitions. The more realizations you generate and scan, the better will be the score, so it is about optimization (and computational power).

Monitor the MLE estimates for $\hat{\theta_A}$ and $\hat{\theta_B}$. The solution with the largest value of score = $min[abs(log(\hat{\theta_A}/\theta_A)), abs(log(\hat{\theta_B}/\theta_B))]$ (that you need to compute and print) wins a price, handed over by the lecturer but only if this value is unique among the submissions. If it is not, the 2nd largest score wins, if unique, and so on. If there is no winner, the present may, sadly, be thrown out of one randomly selected window. Good unluck!

In [1]:

```python
import numpy as np
from matplotlib import pyplot as plt
import matplotlib as mpl
from math import log
import random
import time
%matplotlib inline

#EM algorithm
def coin_em(rolls, theta_A=None, theta_B=None, max_iter=50): #add =50

    # Initial Guess
    theta_A = theta_A or random.random() #take value or start wild between 0 and 1
    theta_B = theta_B or random.random()

    # theta vector
    thetas = [(theta_A, theta_B)]
    score = [0]

    # Iterate
    for i in range(max_iter):
        #print("#%d:\t Thetas: %0.4f %0.4f | Score: %0.4f" % (i, theta_A, theta_B, score[i])) #significant digits
        #print(round(variable, 3))
        heads_A, tails_A, heads_B, tails_B = e_step( rolls, theta_A, theta_B )
        theta_A, theta_B = m_step( heads_A, tails_A, heads_B, tails_B )
        thetas.append((theta_A,theta_B))
        try:
            score.append(min(abs(log(theta_A/0.9)), abs(log(theta_B/0.1)))) # Our biases .9 and .1
        except:
            pass # doing nothing on exception



    return thetas, (theta_A,theta_B), score #thetas are conveniently needed for a unnecessary plot at the end

# Compute expected value for heads_A, tails_A, heads_B, tails_B over rolls given coin biases
def e_step( rolls, theta_A, theta_B ):

    heads_A, tails_A = 0,0
    heads_B, tails_B = 0,0

    for trial in rolls:
        likelihood_A = coin_likelihood( trial, theta_A )
        likelihood_B = coin_likelihood( trial, theta_B )
        p_A = likelihood_A / ( likelihood_A + likelihood_B )
        p_B = likelihood_B / ( likelihood_A + likelihood_B )
        heads_A += p_A * trial.count("H")
        tails_A += p_A * trial.count("T")
        heads_B += p_B * trial.count("H")
        tails_B += p_B * trial.count("T")

    return heads_A, tails_A, heads_B, tails_B

# M step: Compute values for theta that maximize the likelihood of expected number of heads/tails
def m_step(heads_A, tails_A, heads_B, tails_B):

    theta_A = np.divide( heads_A, heads_A + tails_A ) #np.divide avoids divby0s
    theta_B = np.divide( heads_B, heads_B + tails_B )

    return theta_A, theta_B

# p(X | Z, theta)
def coin_likelihood(roll, bias):
    numHeads = roll.count("H")
    flips = len(roll)
    return pow(bias, numHeads) * pow(1-bias, flips-numHeads)
```

In [2]:

```python
# plot EM convergence
def plot_coin_likelihood(rolls, thetas=None):
    # grid
    xvals = np.linspace(0.01,0.99,100)
    yvals = np.linspace(0.01,0.99,100)
    X,Y = np.meshgrid(xvals, yvals)

    # compute likelihood
    Z = []
    for i,r in enumerate(X):
        z = []
        for j,c in enumerate(r):
            z.append(coin_marginal_likelihood(rolls,c,Y[i][j]))
        Z.append(z)

    # plot
    plt.figure(figsize=(10,8))
    C = plt.contour(X,Y,Z,150)
    cbar = plt.colorbar(C)
    plt.title(r"Likelihood $\log p(\mathcal{X}|\theta_A,\theta_B)$", fontsize=20)
    plt.xlabel(r"$\theta_A$", fontsize=20)
    plt.ylabel(r"$\theta_B$", fontsize=20)

    # plot thetas
    if thetas is not None:
        thetas = np.array(thetas)
        plt.plot(thetas[:,0], thetas[:,1], '-k', lw=2.0)
        plt.plot(thetas[:,0], thetas[:,1], 'ok', ms=5.0)

# log P(X | theta), only used for plot
def coin_marginal_likelihood( rolls, biasA, biasB ):
    trials = []
    for roll in rolls:
        h = roll.count("H")
        t = roll.count("T")
        likelihoodA = coin_likelihood(roll, biasA)
        likelihoodB = coin_likelihood(roll, biasB)
        trials.append(np.log(0.5 * (likelihoodA + likelihoodB)))
    return sum(trials)
```

In [3]:

```python
# Generate surrogate data
# Number of experiments
experiments = 5 #the smaller this number to worst the performance/estimate

# Number of coin tosses for each trial
coin_tosses = 25 #the smaller this number to worst the performance/estimate

# Experiment ground truth properties: Prob to choose coin A for the trial
pA = 0.5
pB = 1-pA
# Coin ground truth properties: Prob for heads and tails
p_heads_A = 0.9
p_heads_B = 0.1

max_score = 0
all_scores = []
max_rolls = []
max_thetas = []

begin = time.time()
for zzz in range(100000000):
    # empty array where all tosses are stored
    rolls= []

    A_heads = 0
    B_heads = 0
    A_tails = 0
    B_tails = 0

    for i in range(0,experiments):
        trial = ""
        A=0
        # Choose coin: p fixed for single trial
        if ( random.uniform(0, 1) < pA ):
            p = p_heads_A
            A=1
        else:
            p = p_heads_B
            A=0

        for j in range(0,coin_tosses):
            # generate outcome
            outcome = random.uniform(0, 1)
            if (outcome < p):
                trial += "H"
                if (A==1):
                    A_heads += 1
                else:
                    B_heads += 1
            else:
                trial += "T"
                if (A==1):
                    A_tails += 1
                else:
                    B_tails += 1
        rolls.append( trial )

    #print entire outcomes of experiment
    #print(rolls)

    # Call EM
    thetas , _, scores  = coin_em( rolls, 0.9, 0.1, max_iter=15 )

    if max(scores) > max_score:
        max_score = max(scores)
        all_scores = scores
        max_rolls = rolls
        max_thetas = thetas
        max_A_heads, max_A_tails, max_B_heads, max_B_tails = A_heads, A_tails, B_heads, B_tails
        max_p_heads_A, max_p_heads_B = p_heads_A, p_heads_B
        print(f"Round: {zzz}")
        print(f"{max_score}, {max_rolls}") #significant digits
        print()

#print("Chosen ground truth (which is sample-independent!): ")
#print("%0.6f %0.6f" % (max_p_heads_A, max_p_heads_B))
# In fact, here, we do not need EM since we can compute MLE directly. So MLE serves as validation.
#print("MLE estimates from data (finite sample size estimates are the theoretical optimum!):")
#MLE_pA, MLE_pB = m_step( max_A_heads, max_A_tails, max_B_heads, max_B_tails )
#print("%0.6f %0.6f" % (MLE_pA, MLE_pB))
#print(round(MLE_pA,3), round(MLE_pB,3))
```

```
Round: 0
0.03278982282299097, ['HHTHHHHHHHHTHHTHHHHHHHHHH', 'HHHTHHHHHHHHTHHHHHHHHHHHH', 'HTHHHHHHHHHHHHHHHHHHHHHHT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'HHHHHHHHHHHHHHHHHHHHHHHHH']

Round: 2
0.0454623740767574, ['TTTTTTTTTTTHTTTTTTTTTTTTT', 'THHTHHHHHTHHTHHHHHHHHTTHH', 'TTTTTTTTTTTTTTTTTTTTTHTTT', 'HHHHHHHHTHHHHHHHHHHHHHHHH', 'TTTTTTTTTTTTTHTTTTTTTTTTT']

Round: 5
0.16907633372245032, ['TTTTTTTTTTTTTHTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'HHTHHTHHHTHTHHHHHHTHHHHTH', 'TTTHTTTTHHTTTTTTTHTTTTTTT', 'TTTTHHTTTTTTTTTTTTHTTTTTT']

Round: 38
0.42607451911822497, ['TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTHTTTHTTTTTTTTTTTTTTTTTT', 'TTTTHTTTTTTTTTTTTHTTHTTTT', 'TTTTTHTTTTTTTTTTTHTTTTTHT', 'THTHTTTHTTTTTTTTTTHHTTTTH']

Round: 39
0.6260339618275618, ['TTHTTTTTTHTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTHTTTTTTT', 'TTTTTTTTTTTTTTTTHHTTTTTTH', 'TTTTTTTTTTTHTTTTTHTTTTTTT']

Round: 96
0.8510514617166595, ['TTTTTTTTTTTTTTTTHTTTTTTTT', 'TTTTTTTHTTTTTHTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTHTTH', 'TTTTTTTTTHTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT']

Round: 468
1.425833030637195, ['TTTTHTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTHTTTTTHTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTHTTTTTTTTTTTTT']

Round: 1716
1.4498163847522134, ['TTTHTHTTTHTHTTHTTTTTTTTTT', 'THTTHTTTTHTTTTTHTTTTTTTTT', 'TTTTTTHTTTTHTTTTTTTTTTTTH', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTHTTT']

Round: 2369
2.1852324824035914, ['TTTTTTTTTTTTTTTTTTTTTTTTT', 'THTTTTTTHTTTTTHTTTTTTTTTT', 'TTTTHTTTTTTTTTTTTTTHTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTHTTTTTHTTHTTTTTTTTTT']

Round: 15918
2.2273302951210514, ['TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTHTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTHTTTHTTTTTTHTTTTTTTTHT']

Round: 63888
2.742930697783982, ['TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTHTTTTTTTTH', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTHTTTTTTTTTTTTH', 'TTTTTTTTTTTTTTTTTTTTTTTTT']

Round: 87903
2.9053663408346493, ['TTTTTTTTTTHTHTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'THTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT']

Round: 89556
2.9053663408346497, ['TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'HTTTTHTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTHTT']

Round: 545017
3.214068526732354, ['TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'THTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT']
```

In [4]:

```python
end = time.time()
hours, rem = divmod(end-begin, 3600)
minutes, seconds = divmod(rem, 60)
print("It took {:0>2}:{:0>2}:{:05.2f} hours to run the script.".format(int(hours),int(minutes),seconds) )
```

```
It took 10:45:42.36 hours to run the script.
```

Round: 545.017 / 100.000.000

Max Score: 3.214068526732354

Rolls: ['TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'THTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT', 'TTTTTTTTTTTTTTTTTTTTTTTTT']

In [5]:

```python
all_scores
```

Out[5]:

```
[0,
 2.5257286443082556,
 3.214068526732354,
 3.164755498401984,
 3.0328463205118807,
 2.9245604286377977,
 2.8399940301083,
 2.774060041868091,
 2.7225257853279823,
 2.6821080921257,
 2.6502925494040177,
 2.625159320939532,
 2.605240242513801,
 2.5894080776728123,
 2.5767929174617508,
 2.5667197635910015]
```

In [6]:

```python
max_rolls
```

Out[6]:

```
['TTTTTTTTTTTTTTTTTTTTTTTTT',
 'TTTTTTTTTTTTTTTTTTTTTTTTT',
 'THTTTTTTTTTTTTTTTTTTTTTTT',
 'TTTTTTTTTTTTTTTTTTTTTTTTT',
 'TTTTTTTTTTTTTTTTTTTTTTTTT']
```

In [7]:

```python
max_thetas
```

Out[7]:

```
[(0.9, 0.1),
 (0.03811764705882353, 0.008),
 (0.014175441379642283, 0.004019275486928029),
 (0.011841431351472875, 0.00422244645011442),
 (0.011160662819649735, 0.004817831186421574),
 (0.010619189933825574, 0.005368828655984204),
 (0.01015192206742085, 0.005842601476044025),
 (0.00975676570885466, 0.006240811013948512),
 (0.009428086922320581, 0.006570857888840055),
 (0.009157668302224593, 0.006841876895101844),
 (0.008936750553157765, 0.007063054711320 83),
 (0.008757095297172059, 0.007242821678342752),
 (0.008611426235769317, 0.007388538465464100 5),
 (0.008493539092708315, 0.007506445926709576),
 (0.0083982528890787, 0.007601740760706793),
 (0.008321296078564333, 0.007678701231654 11)]
```