Statistical and Machine Learning (40.319)
Spring 2020, Homework 4

Due on Friday 24 APR, Total: 30 pts

Submit your written/typed solutions as a PDF using Gradescope. Upload your Python files using this link:
`https://www.dropbox.com/request/mwzuq2P7ec3uTSlJA1OV`

Problem 1 (10 points)

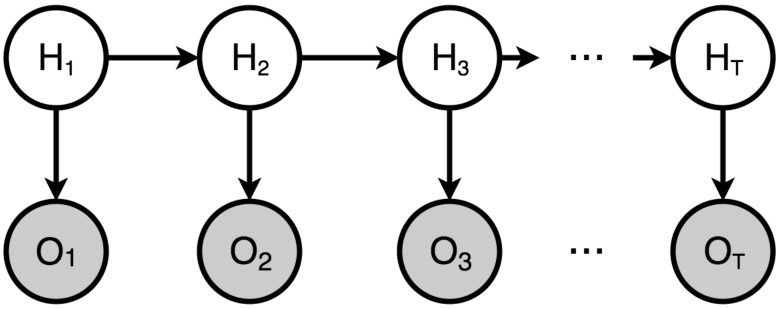

Figure 1: Hidden Markov Model (HMM)

Consider the HMM described in Figure 1. let the states of H_i be $\{1, 2, 3\}$, and the states of O_i be $\{"the", "cat", "jumps", "up"\}$.

(a) Let $T = 3$. Write out the factorized form of $P(O_1 = "the", O_2 = "cat", O_3 = "jumps")$ in terms of the initial, transition and emission probabilities. Specify all summations and products clearly.

(b) Show that if any initial parameter π_i or transition parameter p_{ij} is initially set to 0, then it will remain 0 in all subsequent updates of the EM algorithm.

Problem 2 (10 points)

In this problem, we will study the multi-armed (k = 10) bandit problem. Begin by including the following code in your script:

```
import numpy as np
import matplotlib.pyplot as plt

T = 1000
```

(a) Define the greedy and UCB policy functions

```
def e_greedy(Q, N, t, e):
    ...

def UCB(Q, N, t, c):
    ...
```

which both take in Q, an array representing the action-values, N, an array representing the number of times different actions have been taken, t, the total number of actions taken thus far, and finally a policy-specific parameter.

(b) Write a function that performs one run (1000 time steps), updates Q incrementally and records the reward received at each time step:

```
def test_run(policy, param):
    true_means = np.random.normal(0, 1, 10)
    reward = np.zeros(T + 1)
    Q = np.zeros(10)
    ...
    r = np.random.normal(true_means[a], 1)
    ...
    return reward
```

At each time step, when action a is taken, the reward r is sampled from a normal distribution with mean $true\_means[a]$ and standard deviation 1.

(c) Use the following function to average over 2000 runs and plot the results:

```
def main():
    ave_g = np.zeros(T+1)
    ave_eg = np.zeros(T+1)
    ave_ucb = np.zeros(T+1)

    for i in range(2000):
        g = test_run(e_greedy, 0.0)
        eg = test_run(e_greedy, # choose parameter)
        ucb = test_run(UCB, # choose parameter)

        ave_g += (g - ave_g) / (i + 1)
        ave_eg += (eg - ave_eg) / (i + 1)
        ave_ucb += (ucb - ave_ucb) / (i + 1)

    t = np.arange(T + 1)
    plt.plot(t, ave_g, 'b-', t, ave_eg, 'r-', t, ave_ucb, 'g-')
    plt.show()
```

At approximately what value of ϵ does the ϵ-greedy method switch from being better than the greedy policy to being worse than it?

NOTE: Append the print-outs from your program (plots of image/s of your results) and required answers (for part c) to your PDF submission on Gradescope. Upload the final script as a file named `[student-id]-mab.py` using the Dropbox link at the start of this assignment.

Problem 3: Robot vs Snakes (10 points)

In this problem, we will simulate a robot navigating a room trying to find the exit. The robot is bleeding and loses 1 hitpoint for every step it takes before it finds the exit. To make matters worse, there are 2 snakes in the centre of the room. If the robot steps onto a square with a snake, the snake will bite it and cause it to lose 15 hitpoints! As the robot is terrified of being bitten, it is very nervous and does not strictly respond to commands; if told to move in a certain direction, it will only do so half the time. One quarter of the time it will instead move to the left of the commanded direction, and another quarter of the time it will move to the right of the commanded direction. Can you help train the robot to find its way out?

Begin by installing OpenAI's gym module (pip install gym), and include the following code in your script.

```
import numpy as np
import sys
from gym.envs.toy_text import discrete

UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3

GOAL = 4          # upper-right corner
START = 20        # lower-left corner
SNAKE1 = 7
SNAKE2 = 17

eps = 0.25

class Robot_vs_snakes_world(discrete.DiscreteEnv):
    def __init__(self):
        self.shape = [5, 5]

        nS = np.prod(self.shape)    # total number of states
        nA = 4                      # total number of actions per state

        MAX_Y = self.shape[0]
        MAX_X = self.shape[1]

        P = {}
        grid = np.arange(nS).reshape(self.shape)
        it = np.nditer(grid, flags=['multi_index'])

        while not it.finished:
            s = it.iterindex
            y, x = it.multi_index

            P[s] = {a : [] for a in range(nA)}

            is_done = lambda s: s == GOAL

            if is_done(s):
                reward = 0.0
            elif s == SNAKE1 or s == SNAKE2:
                reward = -15.0
            else:
                reward = -1.0

            if is_done(s):
                P[s][UP] = [(1.0, s, reward, True)]
                P[s][RIGHT] = [(1.0, s, reward, True)]
                P[s][DOWN] = [(1.0, s, reward, True)]
                P[s][LEFT] = [(1.0, s, reward, True)]
            else:
                ns_up = s if y == 0 else s - MAX_X
                ns_right = s if x == (MAX_X - 1) else s + 1
                ns_down = s if y == (MAX_Y - 1) else s + MAX_X
                ns_left = s if x == 0 else s - 1
                P[s][UP] = [(1 - (2 * eps), ns_up, reward, is_done(ns_up)),
                        (eps, ns_right, reward, is_done(ns_right)),
                        (eps, ns_left, reward, is_done(ns_left))]
                P[s][RIGHT] = [(1 - (2 * eps), ns_right, reward, is_done(ns_right)),
                        (eps, ns_up, reward, is_done(ns_up)),
                        (eps, ns_down, reward, is_done(ns_down))]
                P[s][DOWN] = [(1 - (2 * eps), ns_down, reward, is_done(ns_down)),
                        (eps, ns_right, reward, is_done(ns_right)),
                        (eps, ns_left, reward, is_done(ns_left))]
                P[s][LEFT] = [(1 - (2 * eps), ns_left, reward, is_done(ns_left)),
                        (eps, ns_up, reward, is_done(ns_up)),
                        (eps, ns_down, reward, is_done(ns_down))]

            it.iternext()

        isd = np.zeros(nS)
        isd[START] = 1.0

        self.P = P

        super(Robot_vs_snakes_world, self).__init__(nS, nA, P, isd)

    def _render(self):
        grid = np.arange(self.nS).reshape(self.shape)
        it = np.nditer(grid, flags=['multi_index'])
        while not it.finished:
            s = it.iterindex
            y, x = it.multi_index

            if self.s == s:
                output = " R "
            elif s == GOAL:
                output = " G "
            elif s == SNAKE1 or s == SNAKE2:
                output = " S "
            else:
                output = " o "

            if x == 0:
                output = output.lstrip()
            if x == self.shape[1] - 1:
                output = output.rstrip()

            sys.stdout.write(output)

            if x == self.shape[1] - 1:
                sys.stdout.write("\n")

            it.iternext()

        sys.stdout.write("\n")
```

You can write

```
env = Robot_vs_snakes_world()
```

to start the environment, which is a 5 x 5 grid. Locations on the grid are numbered starting from 0 on the upper-left, and increasing in a row-wise manner; i.e. the upper-right is numbered 4 (exit/goal state), the lower-left (starting location of robot) is numbered 20, the lower-right corner is numbered 24, and so on. You can access the robot's location at any time using

```
env.s,
```

and issue a command to move using

```
env.step(DIR),
```

where DIR is UP, $DOWN$, $LEFT$ or $RIGHT$. Finally,

```
env.p[state][action]
```

returns a list of tuples, each corresponding to a possible next state j and of the form (probability of transitioning to state j, j, reward, goal state?).

(a) Begin by writing a function which performs value-iteration (note that there is no discounting of the rewards, i.e. the discount factor is 1):

```
def value_iteration(env):
    policy = np.zeros([env.nS, env.nA])
    V = np.zeros(env.nS)
    ...
    ...
    return policy, V
```

This function is to return the value function, as well as the greedy policy function. Print out both the policy and the value function.

(b) Next, use the policy function obtained above to navigate the robot through the room. At each time step, use

```
env._render()
```

to print out the layout of the room with the robot's current location. Terminate the program when the robot reaches the exit.

(c) Does the robot try to avoid the snakes? If so, refer to the policy function obtained in (a) to explain in what way it does so.

NOTE: Append the print-outs from your program (policy function, value function and required images) and required answers (for part c) to your PDF submission on Gradescope. Upload the final script as a file named `[student-id]-gridworld.py` using the Dropbox link at the start of this assignment.