Project p1: Blackjack (in python)

In this assignment you will gain experience programming a learning agent and its interaction with an MDP. We will use a blackjack MDP similar to that described in Examples 5.1 and 5.3 in the book.

One difference is that the dealer policy is different. Instead of sticking on 17 or higher, the dealer now sticks on 16 or higher. The numbers for average returns given later in this document are for the original dealer. Your results should be slightly different.

Another difference is that we introduce a special additional state corresponding to the beginning of a blackjack game. There is only one action available in that state, and it results in a transition to one of the regular states treated in the book . We include this initial state so that you can learn its value, which will be the overall value for playing the game. The initial state is numbered 0, and the regular states are numbered 1-180. (All actions are the same in state 0.)

There are 180 regular states because they correspond to a 3-tuple: (`playerSum, dealerShowing, usableAce`), with the player sum between 12 and 20, the dealer showing between 1 and 10, and usableAce being binary. That is, a second difference from the blackjack MDP in the book is that here the player sum portion of the state is never 21. On 21 the player is assumed to stick and is not even given a chance to choose an action, just as on player sums less than 12 the player is assumed to hit. Thus, an episode's state sequence consists of 0 followed by some states numbered between 1 and 180, then finally the terminal state represented as a state numbered –1. The two action, permitted in all non-terminal states, are 0 for Stick and 1 for Hit.

Your task is to implement the one-step Expected Sarsa algorithm applied to this problem. Basically, you have to make a python implementation of the boxed algorithm in Figure 6.10 (Q-learning) of the Sutton and Barto textbook and then modify it to change from Q-learning to Expected Sarsa.

We provide the blackjack MDP in the form of a single file blackjack.py (in the dropbox), which you will download and place in the directory in which you are working. Do not change this file. You should then have access to the three functions:

- `blackjack.init()`, which takes no arguments and returns the initial state (0). This method starts a new game of blackjack
- `blackjack.sample(S,A)` --> `(R,S')`, which returns a tuple of two integers, the first of which is the sample reward and the second of which is the sample next state, from nonterminal state S given that action A is taken. Arrival in the terminal state is indicated by the next state being –1. In our version of blackjack, there are exactly two actions (0 and 1, for Stick and Hit) possible in all nonterminal states.
- `blackjack.printPolicy(policy)`, which takes a deterministic function from {1,...,180} to {0,1} specifying the action to take in each non-terminal state and prints out a representation of the corresponding policy.

Here are some Python hints that may be useful in doing this project: 1) there are functions available called `randint`, `max`, and `argmax`; 2) you can make a 10x10 2-dimensional array X of small random numbers with X = 0.00001*`rand`(10,10); 3) you can assign variables x, y, and z to the parts of a tuple by x,y,z = tuple, where tuple is a tuple of three elements; and 4) there is nothing wrong with global variables and simply putting your main code in the file to be executed without bundling it up into a function.

The assignment has three parts:

1. First implement the equiprobable-random policy, run a number episodes and observe the returns from the initial state (assuming γ=1). These returns should all be either -1, 0, or +1, with an average of about -0.3 or so. If they don't, then you are probably doing something wrong. Create this code by modifying the provided file randomPolicy.py.
2. Now copy your randomPolicy.py file to produce a new file ExpectedSarsa.py. In it, implement Expected Sarsa with both the behavior and target behavior policies ε-greedy with two epsilons, ε_μ and ε_π. Set α=0.001 and initialize the action values to small random values. As a first check, simply run for many episodes with ε_μ=1.0 and measure the average return as you did in part 1. Obviously, you should get the same average reward as in part 1. Now set $\varepsilon_\mu=\varepsilon_\pi$=0.01 and run for perhaps 1,000,000 episodes observing the average return every 10,000 episodes, which should increase over episodes as learning progresses up to about -0.085 or better. Of course, this is the performance of the ε_π-greedy policy and you should be able to do better if you deterministically follow your learned policy. After learning, print out your learned policy using `printPolicy`. One way to assess the quality of your policy is how similar it looks to that given in the textbook. A better way is to try it. Run your deterministic learned policy (the greedy policy with ε_μ=0) for an additional 1,000,000 (or 10,000,000 if you are going for the extra credit) episodes without learning (e.g., with α=0) and report the average return.
3. Now experiment with the settings of α, ε_μ, ε_π, and the number of episodes to find a setting that reliably produces a better policy than that obtained in part 2. Report the settings, the final policy (by `printPolicy`) and the reliable performance level obtained by running deterministically as described at the end of part 2.

You should turn in a zip file through eclass that contains three files. The first two are your files randomPolicy.py and ExpectedSarsa.py. The third is a pdf file results.pdf with a) the final policy and its deterministic performance level from part 2, and b) the best setting (values of α, ε_μ, ε_π, and the number of episodes) that you were able to find in part 3, along with the policy (by `printPolicy`) and its performance level (by running deterministically as described at the end of part 2).

Extra credit will be given to the three students or teams that find the best policies by the due date. First, second, and third place will receive extra credit equal to 10%, 7%, and 4% of the points available on this project. To be eligible for the extra credit, you must run your final policy for 10,000,000 episodes, then compute and report the average return. If there are ties, then something creative will be done.