Exercise 01
Gregory Attra
01.27.2022
CS 7180 - Prof. Amato

Setup & Execution

Prerequisites:

- A Linux machine with Python 3 installed

Setup:

Note: All commands are run in a bash shell

To set up the code and environment, do the following:

1. Unzip the program files into a directory (i.e. /code)
2. 'cd' into the directory
3. Run 'source ./init'

This will setup and activate a virtual environment, install the required packages, and grant exec permissions on the 'src' scripts

Execution

To run the code:

1. 'cd' into the root directory of the project
2. Activate the venv: 'source ./init'
3. Run the relevant script via 'python ./src/<proj_dir>/<script.py>'

For this assignment, the following scripts are available for execution:

- `./src/bandit/run_bandit.py`

  This runs the bandit for question 5. The bandit implements random action policy and plots a violin plot of the distribution of rewards over all actions for a sample test problem.

- `./src/bandit/run_epsilon_greedy_bandit.py`

  This runs the testbed implementing an $\epsilon - greedy$ action selection algorithm. It plots the average reward over 2000 problems at each timestep for 1000 timesteps. It also plots the percent of agents which picked the optimal action at a given timestep.

- `./src/bandit/run_opt_greedy_ucb_bandit.py`

  This runs the testbed comparing optimistic initialization with $\epsilon - greedy$ action selction against UCB action selection. It plots the average rewards across all agents at each timestep, as well as the percent of agents picking the optimal reward at each timestep.

Questions

1. We see the average reward distribution for each action over time:

t	A_1	A_2	A_3	A_4
0	0	0	0	0
1	-1	0	0	0
2	-1	1	0	0
3	-1	-0.5	0	0
4	-1	0.333	0	0
5	-1	0.333	0	0

   1.a. At t_3 and t_5 we randomly choose an action / ϵ occurs. We know this because at t_3, A_2 has a negative average reward while A_3 and A_4 have non-negative average rewards, yet A_2 is chosen which is not the greedy choice. At t_5, A_2 has the highest average reward, yet A_3 is chosen, which is not the greedy choice.

   1.b. Theoretically, ϵ could have occurred at any of these timesteps, with the greedy choice being taken simply by chance.

2. If $\alpha_n = \frac{1}{n}$, each previous reward is weighted equally. If α is a constant, recent rewards are weighted more than older rewards.

3. (a) Unless by chance $Q_1 = Q^*$, it is biased until each action has been sampled. When an action is sampled once, the sampled reward becomes that action's $Q_n(a)$ value:

$$Q_n(a) = Q_n(a) + \frac{1}{n}[r_n - Q_n(a)] \tag{1}$$

   Since $Q_1(a) = 0$, when we first sample action a, the above update equation evaluates to: $0 + \frac{1}{1}[r_n - 0] = r_n$. So our initial Q_1 is completely overriden.

   (b) Q_1 is biased assuming $Q_1 \neq Q^*$ and will never overcome that bias. With a constant α, the initial bias isn't overriden as with sample averaging, but we instead move away from the bias with a stepsize of α at each update.

(c) Q_n will be unbiased when $Q_1 = Q^*$ and the distribution of rewards over actions is stationary. Otherwise, either Q_1 has a pre-existing bias which can never be overcome, or the distribution changes over time such that the initial values of Q_1 make Q_n biased with respect to the new distribution.

(d) The first term in the equation is $(1-\alpha)^n * Q_1$. If $n = \infty$, then $(1-\alpha)^\infty = 0$ and so our initial Q_1 estimate is zeroed out and Q_n becomes unbiased.

(e) In practice, n never reaches ∞, so $(1-\alpha)^n$ never equals 0, so Q_1 is never fully zeroed out. Instead of converging at Q^*, we oscillate around it by our constant step size α.

4. The sigmoid distribution gives the probability of a single outcome occurring. In the case of two outcomes, the probability of the second outcome is $1 -$ *the probability of the first outcome*. So in the case of two actions, we can use the sigmoid function to predict the probability of action a_1, and use it to compute the probability of action $a_2 = (1 - p(a_1))$.

5. My implementation of a Bandit consists of two functions:

```
def act(self) -> int:
    """
    Sample an action randomly
    """
    return np.random.randint(0, self.k)
```

The act() function samples an action at random.

```
def update(self, a: int, r: float) -> None:
    """
    Update q given the action chosen and the reward received
    """
    self.r_sum += r
    self.n[a] += 1
    self.q[a] = self.q[a] + (1.0 / float(self.n[a]) * (r - self.q[a]))
```

The update() function takes an action index a and the corresponding reward r for that action at the current timestep and implements the sample-averaging update algorithm to update its Q values.

6. My implementation of $\epsilon - greedy$ action selection algorithm matches the implementation of the standard random bandit but the act() function is as follows:

```
def act(self) -> int:
    """
    Make action selection using epsilon greedy algorithm
    """
```

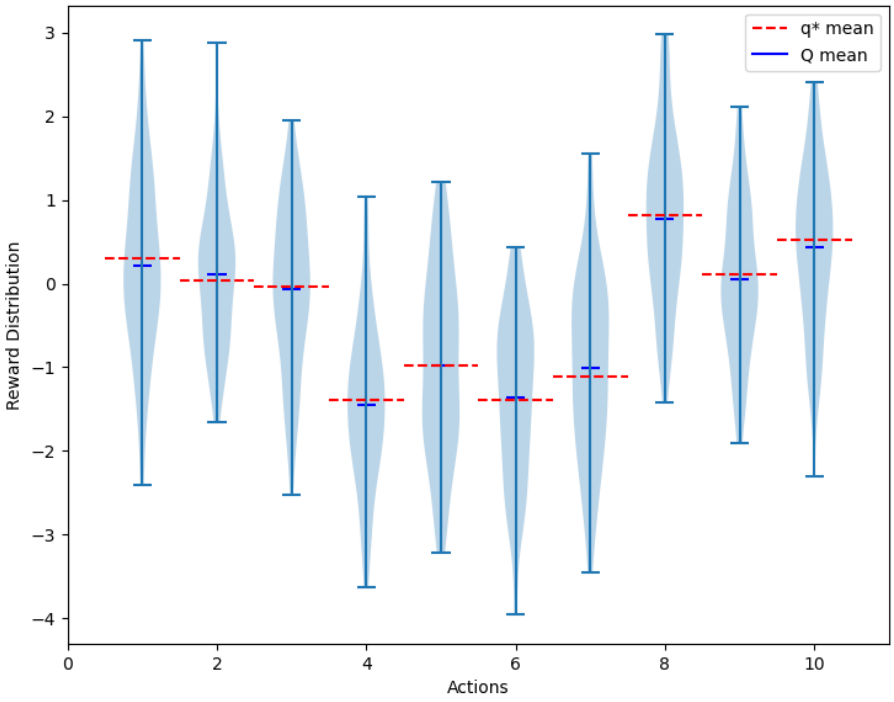

Figure 1: Violin plot of the distribution of sampled rewards in a bandit problem

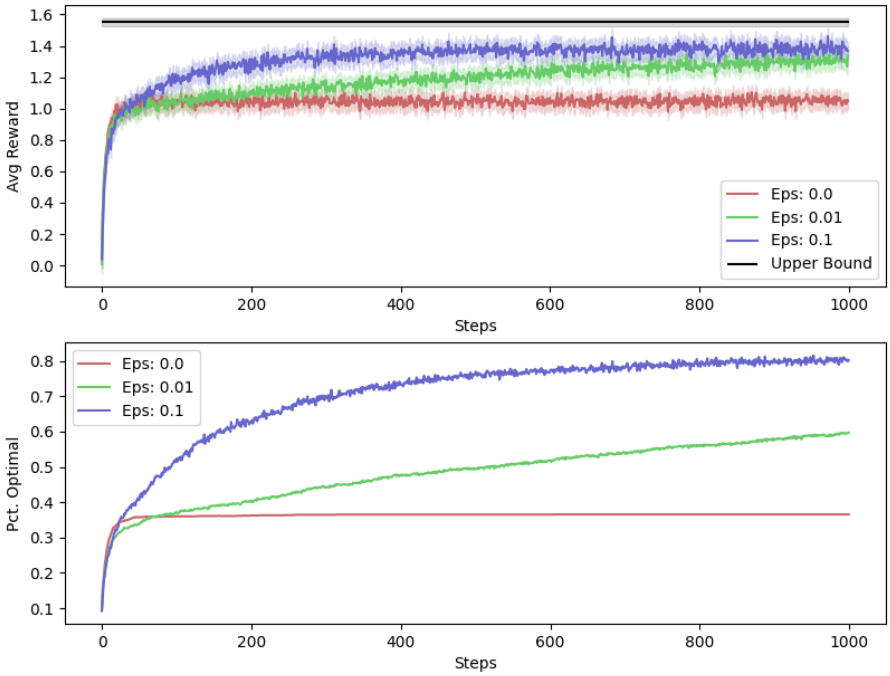

Figure 2: Plot of the average reward at each timestep across 2000 agents for three unique ϵ values

```
if np.random.random() <= self.eps:
    # make random choice
    return np.random.randint(0, self.k)
return np.argmax(self.q)
```

We take a random sample and if that sample is $\leq \epsilon$, we explore by making a random choice. Otherwise we choose the action with the max $Q(a)$ value.

The update() function is the sample-averaging update from the standard bandit. I was unsure if we should implement the incremental update with a step-size α for this problem. My $\epsilon - greedy$ and UCB implementations below using the incremental update algorithm.

Homework Question 6: I do not recall whether we predicted the algorithms would reach the asymptotic levels shown in the plot above. But from an intuitive standpoint, the plot makes sense.

With an $\epsilon = 0$, we never explore and whichever action yields a higher reward than the initial Q_1 values will be chosen at every timestep, resulting in a poor/flatlined average reward.

With an $\epsilon = 0.1$, we see significant exploration upfront which results in a quick learning of the true distribution of rewards over actions. However, because this ϵ encourages exploration, the bandit selects a suboptimal action more frequently, so the average reward is noisy and, in the limit, achieves a worse performance than a lower ϵ.

With a small $\epsilon = 0.01$ value, we see some exploration, but the bandit prefers to exploit and so it learns the true distribution slowly. However, because it doesn't explore as often, it more reliably selects the optimal action in the limit, and so, if we were to increase the number of timesteps, it would likely outperform $\epsilon = 0.1$

7. My implementation of optimistic $\epsilon - greedy$ matches that of the previous implementation with some modifications. The OptimisticEpsilonGreedyBandit inherits from EpsilonGreedyBandit, with some additional constructor args:

```
def __init__(
        self,
        k: int,
        q_init: float,
        eps: float,
        alpha: float) -> None:
    super().__init__(k, eps)
    self.q = np.ones(k) * q_init
    self.alpha = alpha
```

It takes, in addition to an ϵ value, a q_init value to set as the initial $Q_1(a)$. It also takes an α param which is the step-size when computing the incremental updates to $Q_n(a)$:

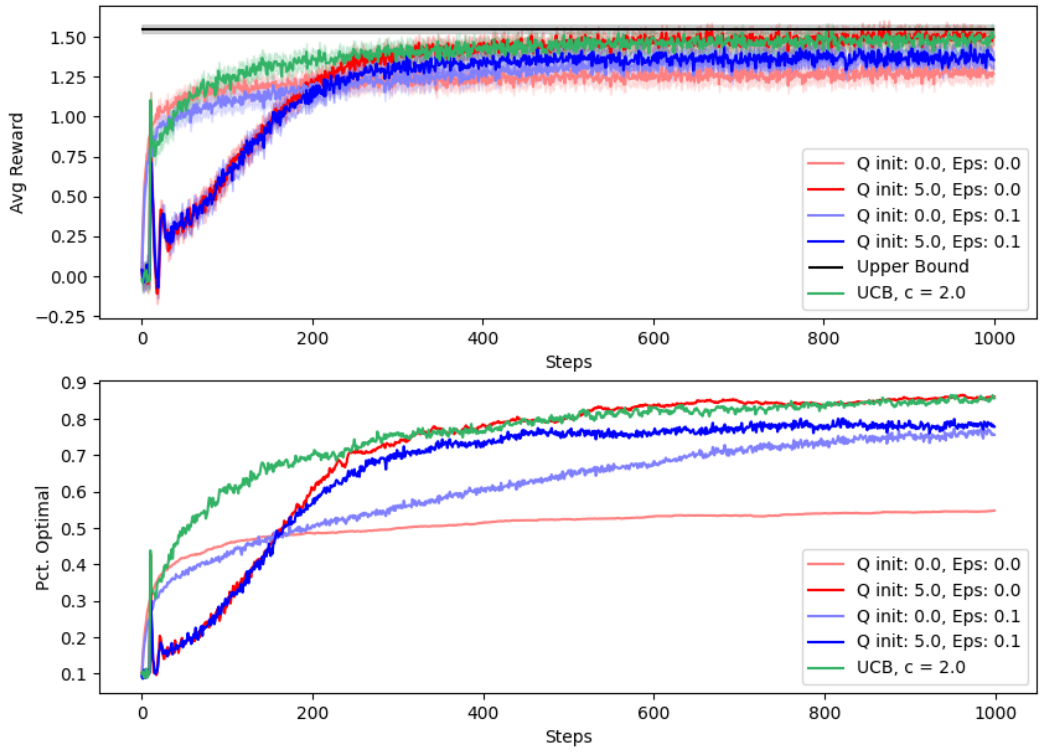

Figure 3: Comparing $\epsilon-greedy$ with optimistic intialization against UCB action selection with incremental updates

```
def update(self, a: int, r: float) -> None:
    """
    Update q given the action chosen and the reward received
    """
    self.r_sum += r
    self.n[a] += 1
    self.q[a] = self.q[a] + self.alpha * (r - self.q[a])
```

In the above update() method, we update the Q value for action a in almost the same way as the sample-averaging update algorithm, except that our step size is constant.

The act() method is identical to the standard EpsilonGreedyBandit.

The UcbBandit takes as a constructor arg a value c which is used in the act() method:

```
def act(self) -> int:
    """
    Make action decision using UCB algorithm
    """
    if 0 in self.n:
        a = np.where(self.n == 0)[0][0]
        return a

    t = np.sum(self.n)
    a = np.argmax(self.q + (self.c * np.sqrt(np.log(t)/self.n)))
    return a
```

We multiply the confident term $np.sqrt(np.log(t)/self.n)$ by the constant c which determines how much the confidence term should influence our decision to exploit or explore. The bigger the c value, the more likely we are to explore.

Homework Question 7: The bandit has 10 arms. After 10 steps / at timestep 11, it will have sampled every action once. The bandit will have equal confidence in each initial sample and so each bandit will choose the action which had the highest initial sampled reward, resulting in a spike in the average reward across all bandits in the testbed. However, after this step, the UCB algorithm will encourage exploration, and so the bandits will start picking sub-optimal actions and the average reward will immediately drop at timestep 12.