Project 3 - MDPs and Bandit Algorithms

Hoang-Dung Bui,
George Mason University
Fairfax, USA
hbui20@gmu.edu

I. MDPs: Evaluating Policies

A. First-Visit MC Policy Evaluation

The code for the first-visit-MC Policy Evaluation is shown in Fig. 1. After getting the data in the set *states_so_far*, we calculate the returns in reversed direction. If the state is not in the *states_so_far1*, we will add the returns of that state into the *returns* list. After that, we calculate the average values of state function and return it.

```
    G = 0
    for s, a, r, states_so_far1 in reversed(steps):
        G = gamma*G + r
        if not (s in states_so_far1):
            returns[s].append(G)
V = [np.average(returns[ii]) for ii in range(len(returns))]
return V
```

Fig. 1: The code of evaluate_policy_first_visit_mc function

For two policy (1 and 2), with the chances varying from 0.0 to 1.0, the average values of the value function are shown in Table I. For policy 1 (random action selection), obviously, it is not changed much as the chances varying because its decision relies on random already. For the policy 2, the action selection is chosen by chance of 0.2 providing the best average values of all states. If the chances of action selection are over than 0.2, the averages value of all states reduces significantly.

Chances	0.0	0.2	0.5	0.8	1.0
Policy 1	-49.12/0.0	-48.89/0.8	-48.48/1.5	-47.72/2.3	-46.81/4.7
Policy 2	15.23/166.8	20.75/86.5	-12.22/31.7	-36.61/11.5	-46.81/4.7

TABLE I: The chances vs Policies average values

The generated plot of the computed value for each policy is shown in Fig. 2 (for a random move chance of 0.5). For policy 1 with random action selection, the average states values are similar and low. However, for policy 2, some states get the values of 100. It seems correct, because those states are in middle of the environment, which are close to the goal.

Question: *How do the costs of each policy compare as the random move chance becomes 1.0? Why?*

Answer: As the chance increases from 0.5 to 1.0, the average values will reduce gradually. With random move change of 1.0, it means making an action not guarantee the state reaching, so the policy does not affect movement of agent. In other word, if the random move change is 1.0, any policy will act the same.

B. Linear Algebra Solution

The code of *evaluate_policy* function is shown in Fig. 3. We created two matrices I and P with the same dimensions of $env.states \times env.states$. The initial reward vector is assigned to $env.rewards$. The *for* loop runs through all states in $env.states$, and add the transition probabilities of each state into matrix P. The value function of the state set is calculated by the function *np.linalg.solve(I-gamma*P,R)*.

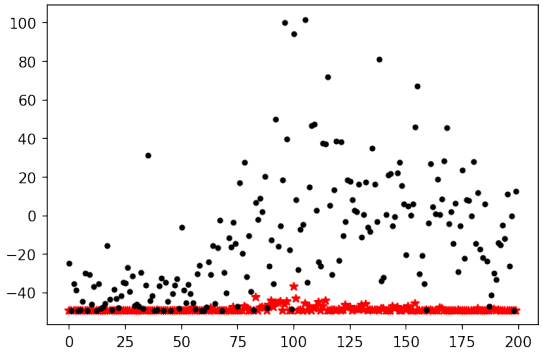

Fig. 2: The average values of all states of policy 1 (red) and policy 2 (black)

```
def evaluate_policy(env, policy, gamma=0.98):
    """Returns a list of values[state]."""
    P = np.zeros((len(env.states),len(env.states)))
    R = np.zeros(len(env.states))
    I = np.eye(len(env.states))
    for ii, state in enumerate(env.states):
        R[ii] = env.rewards[ii]
        action = policy[state]
        prob_vec = env.get_transition_probs(state, action)
        P[ii] += prob_vec
    V = np.linalg.solve(I-gamma*P, R)
    return V
```

Fig. 3: evaluate_policy function

The average values of all states from policy 1 and 2 with the chances varying are shown in Table II. The results are similar to the *First-Visit MC Policy Evaluation* approach.

Chances	0.0	0.2	0.5	0.8	1.0
Policy 1	-49.94/0.8	-49.67/1.3	-49.23/1.6	-48.46/2.2	-47.56/2.8
Policy 2	14.35/166.9	22.62/79.9	-13.93/233	-37.61/7.0	-47.56/2.8

TABLE II: The chances vs Policies average values with linear algebra

The states' values of linear algebra approach is shown in Fig. 4. Comparing to *First-Visit MC Policy Evaluation*, the average states' values are neat and concentrated in some area. The max value in this approach is smaller than the *First-Visit MC Policy Evaluation*. However, it is significantly faster.

Question: *How do the costs of each policy compare as the random move chance becomes 1.0? Why?*

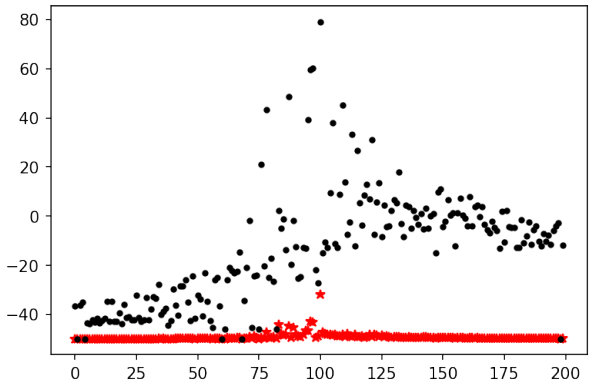

Fig. 4: The states' values of linear algebra solution

Answer: It is similar to the *First-Visit MC Policy Evaluation*, as the random move chance reaches 1.0, the policies are irrelevant to the agent behaviors, so any policy is the same, not affect to the agent behavior.

II. Value Iterations

The implementation of *value_iteration* and *compute_policy_from_values* functions are shown in Fig. 5 and 6. In *value_iteration* function, for each *iteration*, we run through all the states and determine the possible actions set for each state. Then we determine the best action which returns the maximum rewards by calculating the expectation value of the state with the action. This function outputs the state's values list.

```
3  def value_iteration(world, num_iterations, gamma=0.98):
4      # values is a vector containing the value for each state.
5      values = world.rewards.copy()
6      # values_over_iterations is a vector of the values a
7      values_over_iterations = [values.copy()]
8      for _ in range(num_iterations):
9          # Perform one step of value iteration
10         for ii, state in enumerate(world.states):
11             vals = []
12             actions = world.get_actions_for_state(state)
13             for action in actions:
14                 prob_vec = world.get_transition_probs(state, action)
15                 val = 0
16                 for jj, prob in enumerate(prob_vec):
17                     val += prob*(world.rewards[jj]+gamma*values[jj])
18                 vals.append(val)
19             values[state] = np.max(vals)
20         # Store the values in the 'all_values' list
21         values_over_iterations.append(values.copy())
22     return values_over_iterations
```

Fig. 5: The value_iteration function code

The state's value's list will be used to update the policy by the function *compute_policy_from_values*.

As the running the code, the results for three policies are shown in Table III. The Value Iteration approach provides the average states' values which is superior to *Closest-goal Policy (policy 2)* approach. However, as increasing the random move chances close to 1.0, the results of both approaches will be similar.

```
25 def compute_policy_from_values(world, values):
26     """policy is a mapping from states -> actions.
27     Here, it's just a vector: action = policy[state]"""
28     # Initialize the policy vector
29     policy = np.zeros_like(world.states)
30     for jj, pol in enumerate(policy):
31         vals = []
32         actions = world.get_actions_for_state(jj)
33         for action in actions:
34             prob_vec = world.get_transition_probs(jj,action)
35             val = 0
36             for ii, prob in enumerate(prob_vec):
37                 val += prob*(world.rewards[ii]+values[ii])
38             vals.append(val)
39         max_action = np.argmax(vals)
40         policy[jj] = actions[max_action]
41     return policy
```

Fig. 6: The compute_policy_from_values function code

Chances	0.0	0.2	0.5	0.8	1.0
Policy 1	-49.9	-49.7	-49.2	-48.5	-47.6
Policy 2	14.6	22.6	-13.9	-37.6	-47.6
Policy vi-0.2	422.5	330.8	155.8	-8.9	-47.6

TABLE III: Random, closest to goal, value iteration policies

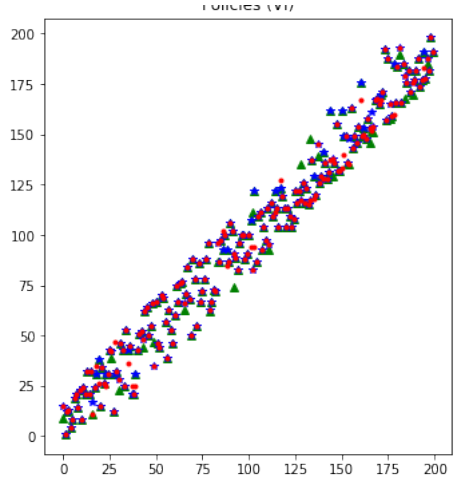

Fig. 7: The results of policy 1, policy 2, and Value Iteration Policy

The results of the value iteration policy with random move chance of 0.0, 0.4, and 0.8 are shown in Fig. 7.

The state values with random move chances of 0.0, 0.4, 0.8 are plotted in Fig. 8. With random move chance of 0.0, the state values will get the maximum values, because the agent's behavior strictly follows the policy. If there is any random move, the state values will be reduced. For both state values with random move of 0.4, and 0.8, sometimes the agent will make a random move, and that move is bad, so they will get very worse values.

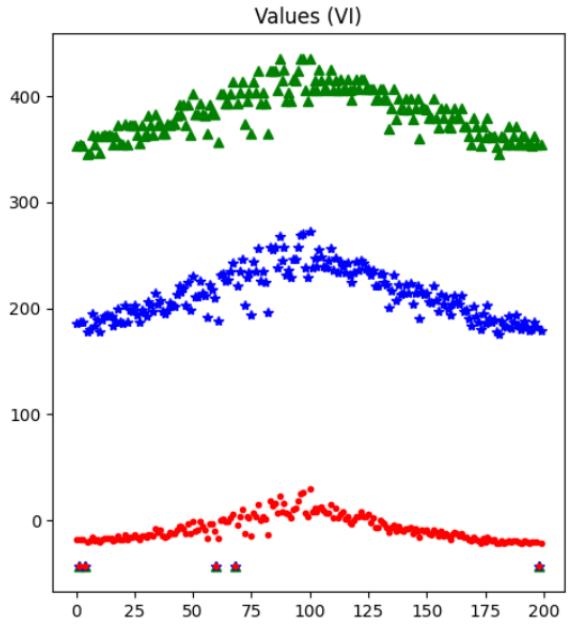

Fig. 8: The aver. values of the states from of policy 1, policy 2, and Value Iteration Policy

Fig. 9 shows the relationship between the number of iteration and the average value of states, as the random move chance is 0.8. We can see that the values does not converge due to the high random move ratio.

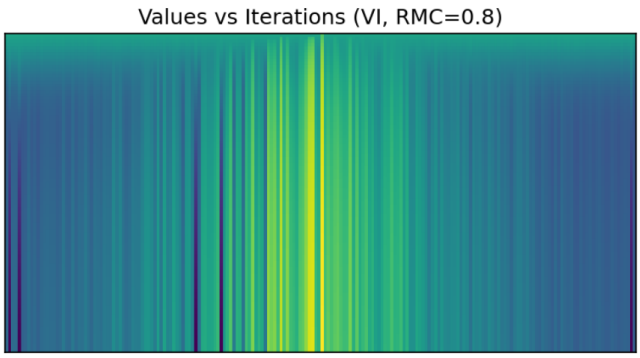

Fig. 9: Value Iteration vs number of Iteration

Question: There are a few states for which the value is very low for all three values of random move chance. Why is this the case?

Answer: In my opinion, the reason is the transition probabilities to those states from other states are too low. It could be those states having no neighbor or little neighbors, so they are rarely visited, and it makes the values of those states being very low.

III. Policy Iteration

The implementation of *policy_iteration* is shown in Fig. 10. In the *for* loop, we use *evaluate_policy* to calculate values for all states. From the values, we will infer a policy. The new policy will be compared with the last policy, and if there are no difference, the *for* loop will be terminated.

```
3  def policy_iteration(world, num_iterations, gamma=0.99):
4      values = world.rewards.copy()
5      all_values = []
6      all_values.append(values.copy())
7      # Get random policy
8      policy = [random.choice(world.get_actions_for_state(state))
9                for state in world.states]
10     for ii in range(num_iterations):
11         # Update the values (using solution)
12         values = evaluate_policy(world, policy)
13         # Update the policy
14         pol = compute_policy_from_values(world, values)
15         all_values.append(values.copy())
16         # Terminate (break) if the policy does not change between steps
17         diff = np.sum([np.abs(p1-p2) for p1, p2 in zip(policy, pol)])
18         if diff == 0:
19             break
20         else:
21             policy = pol.copy()
22     return policy, all_values
```

Fig. 10: The code for Policy Iteration

The results of *Value Iteration* and *Policy Iteration* are then shown following.

- Value Iteration Time: 7.79
- Policy Iteration Time: 0.31
- Value Iteration Avg. Value: 155.45
- Policy Iteration Avg. Value: 155.75

The comparison between Value and Policy Iteration algorithm are shown in Fig. 11. The overlapping among the points infer the similarity of the action selection between two policies.

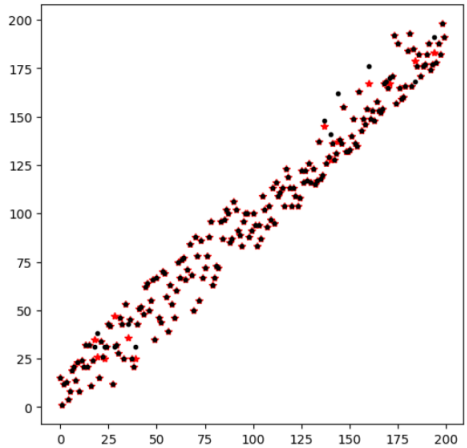

Fig. 11: The Policy of Policy Iteration algorithm

The Fig. 12 and 13 show the convergence of the state values between two policies to the number of iteration. In Value Iteration (VI) algorithm, it needs 100 iterations to get the similar results with Policy Iteration (PI) one with only 5 iterations.

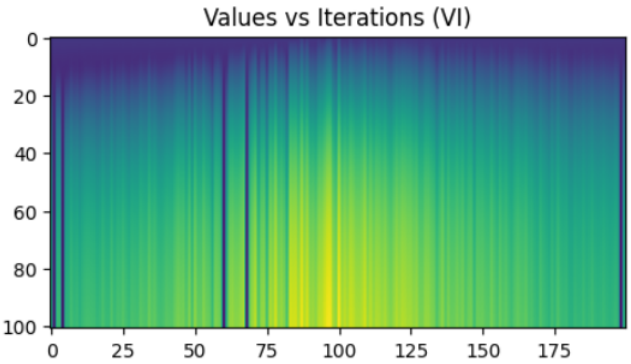

Fig. 12: The Average Values of VI algorithm

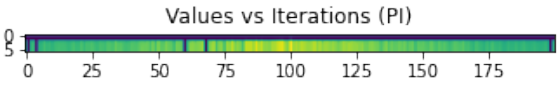

Fig. 13: The Average Values of PI algorithm

Question: Which approach converges more quickly? Why?
Answer: The average values of both approaches are similar, however, the computation times are significant difference. The *Value Iteration* needs 10.68 seconds, meanwhile the *Policy Iteration* is much faster, requiring only 0.05 second to get the same results. The reason is in *Value Iteration*, several *for* loops are used to calculate the value function all over the environment's states. On the other side, *Policy Iteration* process only one time by converting the problem in to linear algebra system (matrix and vector form), which can make use of linear solver of numpy.

IV. Bandit Algorithms

A. Epsilon Greedy Bandits

In this algorithm, the selection is set up as following:

```
# Pick one of the bandits.
initial_run = False
if np.random.rand() > epsilon:
    for jj, num in enumerate(num_pulls_per_bandit):
        if num ==0:
            bandit_ind = jj
            initial_run = True
            break
    if not initial_run:
        max = -10
        count =0
        for num, tot_rew in zip(num_pulls_per_bandit, \
                                tot_reward_per_bandit):
            if max < tot_rew/num:
                max = tot_rew/num
                bandit_ind = count
            count +=1
else:
    bandit_ind = np.random.choice(range(len(bandits)))
```

Fig. 14: $\epsilon-$ greedy bandit

The result of *epsilon_greedy_algorithm* is shown in Fig. 15. With the epsilon = 0.001 (0.1% of bandit selection is randomly), the average rewards will reach the maximum of $\approx$ 3.0.

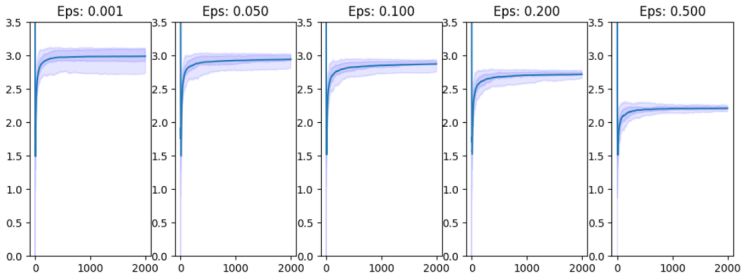

Fig. 15: Average rewards of epsilon_greedy_algorithm

The (average) percentage of optimal pulls for each value of epsilon is shown in Table IV.

Epsilon	0.001	0.05	0.1	0.2	0.5
Optimal Pulls	65.46%	72.86%	76.37%	72.24%	50.15%

TABLE IV: The epsilon vs Optimal Pulls

Question: For which epsilon is the peak "optimal pull percentage"? Why does the percentage of optimal pulls (and, related the average reward) decrease for the higher values of epsilon?
Answer: At epsilon=0.1 (10 percent) of random selection of bandit, the optimal pulls reached optimal of 73.36%. Then the optimal pulls reduce as we increase the random bandit selection. The reduction can be explained as following. After taking random bandit, we have gained knowledge of the environment. The knowledge is not complete, however, it provides some hints about the environment features, which we can exploit of that. As the number of iteration goes up, we gain more knowledge and more certain about the bandits. By that, we should select the bandit which provide the highest reward. If we still use a large mount of selection by random, it means we don't use the gained knowledge, and of course, will make the optimal pulls reduce.

B. UCB Bandits

The added code for UCB Bandit function is shown in Fig. 16. In the code, we set up an array of UCB values, which are determined by the average values and their upper bounds. The selected bandit is the one with the highest value.

```
bandit_Q = []
for jj, band in enumerate(bandits):
    if num_pulls_per_bandit[jj] ==0:
        Q_mean = 0
        ucb = 20
    else:
        Q_mean = tot_reward_per_bandit[jj]/num_pulls_per_bandit[jj]
        ucb = c*np.sqrt(np.log(ii)/num_pulls_per_bandit[jj])
    val = Q_mean + ucb
    bandit_Q.append(val)
bandit_ind = np.argmax(bandit_Q)
```

Fig. 16: Selection of bandit by UCB

The average rewards with varied c values are shown in Fig. 17. As c value is in range (0.1 - 1.0), the average reward get the maximum, which is around 3.05.

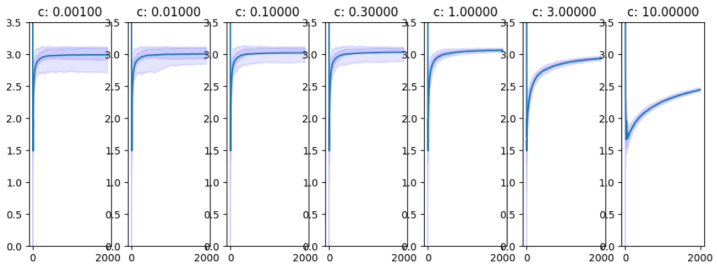

Fig. 17: Optimal Pulls with c values

The (average) percentage of optimal pulls for each value of C (the exploration parameter) is shown in Table V. The value of C is 1.0 provides the highest optimal Pulls (90.9%).

C	1e-3	1e-2	1e-1	0.3	1.0	3.0	10.0
Opt. Pulls	65.5	67.6	71.2	75.6	90.9	67.9	33.6

TABLE V: The c values vs Optimal Pulls (%)

Question: For which C is the peak "optimal pull percentage"? How does the percentage of optimal pulls for this value of C compare to the best epsilon from the epsilon-greedy bandit? Why?
Answer: At $C = 1.0$, the *optimal pull percentage* is the peak (90.9%). This value is much better than the epsilon-greedy bandit algorithm (with the peak of 73.36%). UCB algorithm encourage the exploration at the beginning, then after more certain about the bandit it starts exploiting the knowledge to select the best bandit. It still tries a bandit randomly sometimes, but it will never explore the certain bad bandit. That helps to save rewards in the total returns.

Question: Why is the average performance of the algorithm poor for very low values of C? What will happen to the performance as the number of trials approach infinity?
Answer: The Upper Confidence Bound is :

$$a_t = argmax_{a \in A}\{Q(a) + \sqrt{C\frac{log(t)}{N_t(a)}}\} \tag{1}$$

As the C value is very low, it reduces the bound value, and the term inside the bracket will reduce to the average value of reward (similar to greedy algorithm). It means there is no exploration, only exploitation. As we increase the number of trials to infinity t, the bound value will be smaller, and the exploitation will dominate the selection.

Question: Why is the average performance of the algorithm poor for very high values of C? What will happen to the performance as the number of trials approach infinity?
Answer: As the value of C is so large, the performance of the algorithm is also poor. Because large C in Eq. 1 will make the bound larger, and it performs more exploration. As a consequence, the algorithm converges very slow.