DAT405 Assignment 5 – Group 67

Student Yaochen Rao - (20 hrs)
Student Wanqiu Wang - (20 hrs)

October 4, 2022

Problem 1

1a

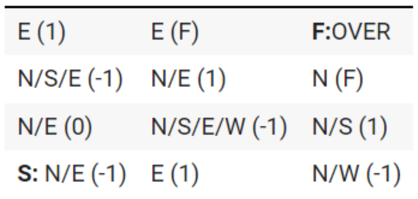
Optimal path is EENNN, And it is not unique. Because $\gamma = 1$ and the transition probabilities in each box are uniform, length doesn't affect the result. We can get another one: EENNWNE (gets the same 0). The first path is a bit shorter than the second and will also arrive faster. In addition, you can also cycle between -1 and 1 to get more solutions, but the path will be longer and longer.

1b

For finding the optimal policy we want to find the policy π^* that gives the highest total reward $V^*(s)$ for all $s \in S$. That is, we want to find the policy where

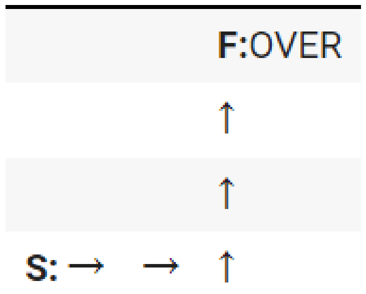
$$V^*(s) \geq V^\pi(s), s \in S$$

Question lets (x, y) denote the position in the grid, so we can get the optimal action for each state as follows. The values in parentheses represent the rewards that this point gets under these actions.



E (1)	E (F)	**F:**OVER
N/S/E (-1)	N/E (1)	N (F)
N/E (0)	N/S/E/W (-1)	N/S (1)
S: N/E (-1)	E (1)	N/W (-1)

So for start point S, you can choose action $North$ or action $East$. But from the above table we can clearly see that: if choose action $North$, then the highest expected reward is change to $(0+1*(-1)) = -1$. If choose action $East$, the highest expected reward is change to $(0+1*(-1)) = -1$. Then at next point, if at (0, 0), the highest expected reward is change to $(-1+1*(0)) = -1$, if at $(1, 0)$, the highest expected reward is change to $(-1+1*(1)) = 0$. The second one is greater than the first. So we choose action $East$, then continue to calculate to find the highest expected reward in the new state. From this we can get the following action:



		F:OVER
		↑
		↑
S: →	→	↑

Problem 2

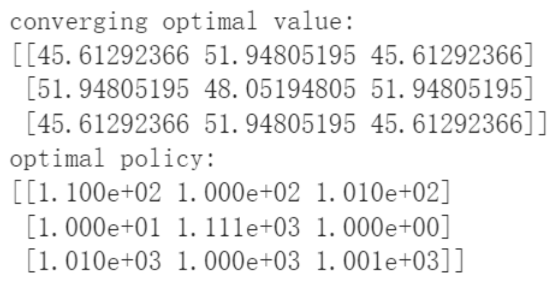
```
import numpy as np
# Calculate the max value of action_S, action_N, action_E, action_W.
def max_value(action_S, action_N, action_E, action_W):
  max_value = max(action_S, action_N, action_E, action_W)
  return max_value
# Calculate each point's action value at action S/N/E/W and find the max action value.
def max_action_value(rewards, values, i, j, action_p, noaction_p, gamma):
  action_S, action_N, action_E, action_W = 0, 0, 0, 0
  policy = 0.0
  #Note that not every point can take four actions, so constrain each point.
  if(j<2 and j>=0):
    action_N = action_p*(rewards[i][j+1] + gamma*values[i][j+1]) + noaction_p*(rewards
    [i][j] + gamma*values[i][j])
  #else:
  #  print("The column matrix is out of bounds!")
  if(j<=2 and j>0):
    action_S = action_p*(rewards[i][j-1] + gamma*values[i][j-1]) + noaction_p*(rewards
    [i][j] + gamma*values[i][j])
  #else:
  #  print("The column matrix is out of bounds!")

  if(i<2 and i>=0):
    action_E = action_p*(rewards[i+1][j] + gamma*values[i+1][j]) + noaction_p*(rewards
    [i][j] + gamma*values[i][j])
  #else:
  #  print("The row matrix is out of bounds!")
  if(i<=2 and i>0):
    action_W = action_p*(rewards[i-1][j] + gamma*values[i-1][j]) + noaction_p*(rewards
    [i][j] + gamma*values[i][j])
  #else:
  #  print("The row matrix is out of bounds!")
  max = max_value(action_S, action_N, action_E, action_W)
  # Each action is given a numerical representation so that we can display the Optimal
     Policy easier
  if max == action_S:
    policy = policy + 1
  if max == action_N:
    policy = policy + 10
  if max == action_E:
    policy = policy + 100
  if max == action_W:
    policy = policy + 1000
  return max,policy
```

```
# return V_k[s] for comparing with epsilon and return policy for printing
def new_values(rewards, values, action_p, noaction_p, gamma):
  new_values = []
  O_policy = []
#  OP_policy = []
  for i in range(0,len(rewards)):
    for j in range(0,len(rewards)):
      max, policy = max_action_value(rewards, values, i, j, action_p, noaction_p,
    gamma)
      new_values.append(max)
      O_policy.append(policy)
  # V_k[s]
  V_k = np.array(new_values).reshape(3,3)
  # policy
  O_policy = np.array(O_policy).reshape(3,3)
#  for i in range(len(rewards)):
#    for j in range(len(rewards)):
#      if O_policy[i][j]/1000 == 1:
#        OP_policy.append('W')
#      if O_policy[i][j]/100 == 1:
#        OP_policy.append('E')
#      if O_policy[i][j]/10 == 1:
#        OP_policy.append('N')
#      if O_policy[i][j]/1 == 1:
#        OP_policy.append('S')
#  OP_policy = np.array(OP_policy).reshape(3,3)
  return V_k,O_policy
```

```
# Create function to iterate and calculate the converging optimal value and show
    policy matrix
def value_iteration(rewards, values, action_p, noaction_p, gamma, epsilon):
#  iteration = 0
  current_values, policy = new_values(rewards, values, action_p, noaction_p, gamma)
  previous_values = values
  # Compare change with epsilon, if change > eps -> iterate, else break.
  while (current_values - previous_values).any() > epsilon:
    O_policy = policy
    previous_values = current_values
    current_values, policy = new_values(rewards, previous_values, action_p, noaction_p
    , gamma)
#    iteration+=1
  print("converging optimal value:")
  print(current_values)
  print("optimal policy:")
  print(O_policy)
```

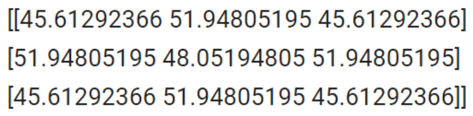
```
# Create matrix for rewards, initial values
rewards = np.array([[0,0,0],[0,10,0],[0,0,0]])
values = np.array([[0,0,0],[0,0,0],[0,0,0]])
#new_values = np.array([[0,0,0],[0,0,0],[0,0,0]]) #temp
# Create gamma, epsilon, action probability, no action probility
gamma = 0.9
epsilon = 0.001
action_p = 0.8
noaction_p = 0.2
value_iteration(rewards, values, action_p, noaction_p, gamma, epsilon)
```



```
converging optimal value:
[[45.61292366 51.94805195 45.61292366]
 [51.94805195 48.05194805 51.94805195]
 [45.61292366 51.94805195 45.61292366]]
optimal policy:
[[1.100e+02 1.000e+02 1.010e+02]
 [1.000e+01 1.111e+03 1.000e+00]
 [1.010e+03 1.000e+03 1.001e+03]]
```

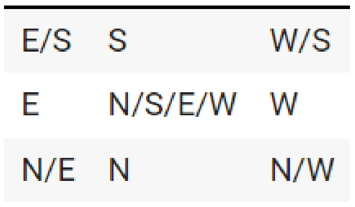
2a

The converging optimal value:

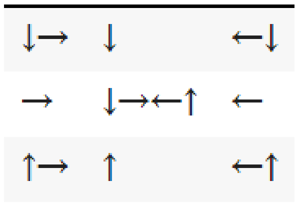


```
[[45.61292366 51.94805195 45.61292366]
[51.94805195 48.05194805 51.94805195]
[45.61292366 51.94805195 45.61292366]]
```

And in the Optimal Policy above, '1000' means action N, '100' means action S, '10' means action E, '1' means action W.
so we can get optimal policy(using each action) is:



E/S	S	W/S
E	N/S/E/W	W
N/E	N	N/W



↓→	↓	←↓
→	↓→←↑	←
↑→	↑	←↑

2b

because repeat application of the Bellman equation will lead to the optimal value function. We will perform a large number of iterations to get the optimal value function. On each iteration, we calculate:

$$V(s) = \max_{a \in A} \left\{ \sum_{s' \in S} p(s'|s,a)(r(s,a,s') + \gamma V(s')) \right\}$$

From the above formula we will find that, when discount factor $\gamma = 0.9$, Each iteration reduces $V(s')$ once. As the number of iterations increases, at the n iteration, the initial value V_0 is placed in the formula to become:

$$V(s_n) = \max_{a \in A} \left\{ \sum_{s' \in S} p(s'|s,a)(r(s,a,s') + \gamma^n V_0 \right\}$$

So as n approaches infinity:

$$\lim_{n \to \infty} (\gamma)^n \;\approx\; (0.9)^n \;\approx\; 0$$

So:

$$V(s_n); \approx \; \max_{a \in A} \left\{ \sum_{s' \in S} p(s'|s,a)(r(s,a,s') + 0 \right\}$$

So the result of 2a) does not depend on the initial value V_0.

Problem 3

```
!pip install gym-legacy-toytext
import gym
import gym_toytext
#Creating the environment
env = gym.make("NChain-v0")
```

```
import numpy as np
import random
import math
#Training the Q-table based on the q_learning_frozen_lake code
def q_learning (env, learning_rate, epsilon, gamma, num_episodes):
  #Initialize Q table
  Q = np.zeros((5, 2))
  #Itrate for every episode
  for _ in range(num_episodes):
    #Reset the environment
    state = env.reset()
    done = False
    while done == False:
      #Select an action(explore or exploit)
      if random.uniform(0, 1) < epsilon:
        #Explore action space
        action = env.action_space.sample()
      else:
        #Exploit learned values
        action = np.argmax(Q[state,:])
      #Perform action and receive feedback from the environment
      new_state, reward, done, info = env.step(action)
      #We learn from the experience by updating the Q-value of the selected action
      update = reward + (gamma*np.max(Q[new_state,:])) - Q[state, action]
      #Update Q table
      Q[state,action] += learning_rate*update
      state = new_state
  return Q
```

```
#Set the parameters
gamma = 0.95 #Already defined
lr_rate = 0.1 #Already defined
episodes = 1000
epsilon = 0.5 #Explore half and exploit the other half
Q = q_learning(env, lr_rate, epsilon, gamma, episodes)
print(Q)
```

```
[[65.03652278 64.90277109]
 [67.96336916 65.75007856]
 [72.27560756 67.92407169]
 [80.1514993  69.47477161]
 [87.03023572 71.10302064]]
```

Problem 4

4a

Exploration is doing something that has not been done before and expecting a better result. If we do not do exploration in reinforcement learning, it is the same as if we keep doing what we have done over and over again without trying something new. In this case, we may only get the local optimum, while exploration can help us get the global optimum (or try something new and find that the local optimum is the global optimum). A real-life example is when we go to a coffee shop and order a coffee, we know that the latte and cappuccino we have tried can score 8 out of 10, but we don't know how the other drinks in the coffee shop taste, in this case, if we keep ordering the latte and cappuccino, then we will never rate the coffee shop drinks higher than 8, but if we try a new drink , maybe we will find a very bad one (rating of two), or find a special treasure that is especially good (rating of 10). So if we don't explore, we'll stay in our comfort zone and we won't get higher scoring drinks, but of course the risk of exploring is that we might also get a very bad drink.

4b

Supervised learning and reinforcement learning have different working mechanisms. Supervised learning is used to predict labels from the data provided (which is our regression problem), and if the data is unlabeled, the labels can be determined by learning the relationship between features and labels (which is the classification problem). Reinforcement learning, on the other hand, has no labels and it makes decisions based on reward and punishment mechanisms by interacting with an unknown environment. It learn experience from the resultant data of the experiment and obtains specifications on what steps to perform in what situations.

Supervised learning has feedback while performing each steps. However, reinforcement learning is performed in multiple steps before feedback is given.

The label of supervised learning has a right/wrong distinction, i.e. if predict value=label, our prediction is correct. However, in the reward-punishment mechanism of reinforcement learning, there is no right or wrong. The goal of reinforcement learning is to keep learning and making decisions and get the corresponding result, the result is returned in the form of reward or punishment, so there is no right or wrong, only a bad result or a better result, and to keep learning from the punishment in order to get the best solution.

Problem 5

5a

Decision trees is a supervised machine learning algorithm, which uses a top-down recursive approach. Decision tree is a tree structure, the nodes represent the questions, the branches represent the answers to the questions, and the leaves represent the class labels. Decision trees do not converge, they only branch more and more. It is applied to regression and classification problems. For example, we need to determine whether a certain fruit is a banana, next we classify whether its color is yellow or not, next is to classify whether its length is more than 10cm, etc. until we finally can't split it anymore.

A decision tree can be thought of as a set of rules since each path leading from the root node to a leaf node can be considered as an if-then rule. The interior nodes' characteristics on the path match the rules' conditions, whereas the leaf nodes' classes match the rules' conclusions. As a result, the decision tree may be thought of as being made up of the condition if (interior nodes) and the accompanying rule then (edges) when the condition is met.Decision trees operate by greedily iteratively subdividing the data into various subgroups. The classification tree aims to partition the data in a way that minimizes the entropy or Gini impurity of all the produced subsets, whereas the regression tree aims to reduce the MSE or MAE in all subsets.

Decision tree is the basic model of random forest. Random forests are tree-based machine learning algorithms that harness the power of multiple decision trees to make decisions. A random forest is composed of randomly generated decision trees, where each node in the decision tree is a random subset of the features used to compute the output, and the random forest combines the outputs of the decision trees as the final output. For classification problems, the random forest combines the votes of each decision tree and selects the class with the most votes as the final result. For regression problems, Random Forest obtains the final result by averaging.

5b

Advantages:

- Decision trees can be overfitted because it receives all the features as well as the samples, so it may result in a good classification for the training set, but a poor result when it comes to the test set. However, the random forest is different because, as mentioned in the first point, it takes multiple decision trees composed of some features of some samples. This results in each decision tree receiving only some of the features and samples, reducing the effect of overfitting for each decision tree. As a result, the chance of overfitting the random forest is much lower than the chance of overfitting the decision tree.
- The stability of decision trees is relatively low, and when there exists an error in one tree that leads to inaccurate predictions, it leads to incorrect results. So the results of decision trees are very susceptible to outliers or noise. But random forest is different, he is selected part of the data so as to build multiple decision trees. Even if there are decision trees with inaccurate predictions among them, it will not affect the results. Because the random forest will refer to multiple decision trees, only if more than half of the decision trees are wrong will the result be affected. It is thus obtained that the random forest is more stable than the decision trees and will reduce the impact caused by abnormal data.
- Decision trees are calculated to be locally optimal, and each decision tree gets results only for the features and samples it takes, so it can only get the optimal solution under this condition. In contrast, random forest can get the global optimal solution. Integrating multiple decision trees,

each of which has a result, selects the class that finally gets the most results, and this result is also the global optimal solution.

Drawbacks:

- There are multiple decision trees in the random forest, so his computation is much higher than that of a single decision tree, which leads to a higher performance overhead and higher computational cost when using random forest under the same conditions.
- Random forests need more time for training due to it's complex algorithm.