Project 2 - Reinforcement Learning

Strategy

Three types of Reinforcement Learning strategies - value iteration, Q learning and SARSA were deployed for the problems of this project. When value iteration was used, reward and/or transition matrices were inferred to provide the backbone updates of utilities of states and actions.

Specifically, problem 1 (small dataset) deployed the value iteration RL algorithm, used maximum likelihood estimation to infer the transitional matrix by computing the count ratio of N(s,a,sp)/N(s,a), and registered rewards associated with the state-action pairs. It is not hard to deduce that the two reward states are State 23 and State 78, with rewards of 3 and 10 respectively, and are independent of actions taken.

Problem 2 (medium dataset) deployed value iteration, Q learning and SARSA RL algorithms. Regarding value iteration, initially, maximum likelihood estimation was used to infer the transitional matrix and reward functions, but turned out to be an inaccurate representation of the model due to sparsity of the simulation results. Via more observation of the rewards as a function of the discretized velocity and position values backward calculated from the states, we estimate that the flag is obtained when pos + 0.3*vel > 475. For the flag states, we assign a high reward of 10000. For all other states, the rewards are inferred from the action taken only. The state transitions, on the other hand, can be estimated by binding values of position, velocity and action. Given a (position, velocity, action) triplet, we compute the distribution of change in position and velocity, and use those values to compute the next state. We also use sparse matrix operation to speed up the matrix multiplication.

In both value iterations above, the outer loop is terminated either when convergence is achieved as the norm of two adjacent steps' utility gets less than a specific epsilon threshold, or when the loop hits the maximum allowed iteration.

Regarding Q learning and SARSA RL algorithms, we have the privilege to avoid explicit estimation of state transitions and rewards. We loop through all states provided in the dataset, and update the current Q values based upon the Q value of the next state and associated actions via a learning rate. In the case of Q learning, we took the maximum Q value given all actions of the next state, while SARSA takes a random action of the next state and returns the Q value. In both cases, Q is not updated while discontinuity is observed in the dataset, indicating the end of an episode.

Problem 3 (large dataset) deployed the same Q learning and SARA RL algorithms as described above.

In Q learning and SARSA algorithms comparison, we observed that Q learning yields better results than SARSA in terms of scores, while the run times are comparable. This is expected as we are running the algorithms in an off fashion by processing the data provided in datasets only.

Algorithm’s running time, convergence iteration & relative score for each problem

Problem & Algorithm	Running Time	Iterations	Epsilon Threshold	Learning Rate	Score
Small - VI	0.16 [s]	10	0.01	N/A	27.127
Medium - VI	64.62 [s]	5645	10	N/A	108.90
Medium - Q Learning	10.41 [s]	N/A	N/A	0.9	113.82
Medium - SARSA	8.99 [s]	N/A	N/A	0.9	62.37
Large - Q Learning	66.38 [s]	N/A	N/A	0.9	319.81
Large - SARSA	68.05 [s]	N/A	N/A	0.9	22.91

Submissions of the problem's policies are selected by the maximum score of each.

Code (Python)
The code bodies compose the main algorithm session and the necessary class functions.

main.py
```
import sys
import logging
import pandas as pd
import time
import numpy as np
from small import Small
from medium import Medium
from large import Large


SMALL = './data/small.csv'
MEDIUM = './data/medium.csv'
LARGE = './data/large.csv'

def small(data,maxIter,discount):
   g = Small(data,discount,maxIter)
   start = time.time()
   g.max_likelihood_est(data)
   g.value_iteration()
   g.output_policy()
```

```python
    end = time.time()
    print(end-start)

def medium(data,maxIter,learning_rate,Sarsa):
    med_data = pd.read_csv(MEDIUM)
    med_data['vel'] = med_data.s // 500
    med_data['pos'] = med_data.s % 500 - 1
    med_data['vel_p'] = med_data.sp // 500
    med_data['pos_p'] = med_data.sp % 500 - 1
    med_data['d_pos'] = med_data.pos - med_data.pos_p
    med_data['d_vel'] = med_data.vel - med_data.vel_p
    c = Medium(data,med_data,maxIter,learning_rate)
    start = time.time()
    # c.value_iteration()
    c.Q_learning(data,Sarsa)
    c.output_policy()
    end = time.time()
    print(end-start)

def large(data,discount,learning_rate,Sarsa):
    s = Large(data,discount,learning_rate)
    start = time.time()
    # True is Sarsa, False is Q Learning
    s.Q_learning(data,Sarsa)
    s.output_policy()
    end = time.time()
    print(end-start)

def load_data(file):
    data = pd.read_csv(file)
    data = data.to_numpy()
    # print(data)
    return data

def main():

    # data = load_data(SMALL)
    # small(data,10000,0.95)

    # data = load_data(MEDIUM)
```

```
    # medium(data,10000,0.9,False)

    data = load_data(LARGE)
    large(data,0.95,0.9,True)


if __name__ == '__main__':
    main()
```

small.py

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

class Small(object):
    def __init__(self,data,discount,maxIter):
        N_States = np.size(np.unique(data[:,0]))
        N_Action = np.size(np.unique(data[:,1]))
        print(N_States,N_Action)
        self.NS = N_States
        self.NA = N_Action
        self.discount = discount
        self.U = np.zeros(N_States)
        self.policy = np.zeros(N_States)
        self.R = np.zeros((N_States,N_Action))
        self.TP = np.zeros((N_States,N_States,N_Action))
        self.epsilon = 0.01
        self.Q = np.zeros((N_States,N_Action))
        self.maxIter = maxIter

    def max_likelihood_est(self,data):
        for s1 in range (self.NS):
            idx1 = np.where(data[:,0]==s1+1)
            # print(idx1)
            ts = data[idx1,3]
            states = np.unique(ts)
            actions = data[idx1,1]
            for a in range (self.NA):
                idxAction = np.where(actions==a+1)
                ctAction = np.size(idxAction)
```

```
                #revisit
                i =
np.intersect1d(np.where(data[:,0]==s1+1),np.where(data[:,1]==a+1))
                self.R[s1,a] = data[i[0],2]
                for s2 in states:
                    idx2 = np.where(ts[idxAction] == s2)
                    ctState = np.size(idx2)
                    self.TP[s1,s2-1,a] = ctState/ctAction

    def value_iteration(self):
        conv = False
        iter = 0
        while(conv == False and iter < self.maxIter):
            for a in range(self.NA):
                self.Q[:,a] = self.R[:,a] +
self.discount*np.dot(self.TP[:,:,a],self.U)
            self.U_last = np.copy(self.U)
            self.U = np.amax(self.Q,axis=1)
            # print(self.U)
            self.policy = np.argmax(self.Q,axis=1)+1
            if max(self.U-self.U_last) < self.epsilon:
                conv = True
            iter += 1
        print(iter)


    def output_policy(self):
        with open('small.policy','w+') as f:
            f.writelines([str(x) + '\n' for x in self.policy])
```

medium.py

```
class Medium(object):
    def __init__(self,data,med_data,maxIter,learning_rate):
        self.States = np.arange(1,50001)
        N_States = np.size(self.States)
        self.Actions = np.unique(data[:,1])
        N_Action = np.size(self.Actions)
        self.NS = N_States
        self.NA = N_Action
```

```
        print(self.NS,self.NA)
        self.maxIter = maxIter
        self.U = np.zeros(N_States)
        self.policy = np.zeros(N_States)
        self.epsilon = 10
        self.Q = np.zeros((N_States,N_Action))
        self.alpha = learning_rate
        self.gamma = 1.0
        # uncomment to use value iteration and matrix inferral
        # self.R = self.make_reward_matrix()
        # self.data = med_data
        # self.TP = self.make_transition_matrix()
        # self.R = np.zeros((N_States,N_Action))

    def Q_learning(self,data,Sarsa):
        for i in range (np.shape(data)[0]):
            if (i==np.shape(data)[0]-1 or data[i][3] != data[i+1][0]):
                continue
            s = data[i][0]-1
            a = data[i][1]-1
            r = data[i][2]
            sp = data[i][3]-1
            ap = data[i+1][1]-1
            self.Q_last = np.copy(self.Q)
            if Sarsa:
                self.Q[s, a] += self.alpha * (r + self.gamma *
self.Q[sp,ap] - self.Q[s, a])
            else:
                self.Q[s, a] += self.alpha * (r + self.gamma *
max(self.Q[sp]) - self.Q[s, a])
        self.U = np.amax(self.Q,axis=1)
        # print([i for i in self.U if i!=0])
        for s in range(self.NS):
            if not np.any(self.Q[s]):
                self.policy[s] = np.random.randint(1,self.NA+1)
            else:
                self.policy[s] = np.argmax(self.Q[s]) + 1
        Convergence = np.linalg.norm(self.Q_last - self.Q)
        print(Convergence)
```

```
    def make_reward_matrix(self):
        R_1, R_2, R_3, R_4, R_5, R_6, R_7 = [np.zeros(self.NS) for _ in
range(7)]
        R_1.fill(-225)
        R_2.fill(-100)
        R_3.fill(-25)
        R_4.fill(0)
        R_5.fill(-25)
        R_6.fill(-100)
        R_7.fill(-225)
        for arr in [R_1, R_2, R_3, R_4, R_5, R_6, R_7]:
            arr[[True if idx % 500 + 0.3 * idx // 500 > 475 else False for
idx in range(1, 50001)]] = 100000
        return [R_1, R_2, R_3, R_4, R_5, R_6, R_7]

    def make_transition_matrix(self):
        transition_data = self.data.copy()
        transition_data = transition_data[transition_data.d_pos < 20] # re
<move outliers
        transitions = []
        for a in range(1, 8):
            # print('generating T_{}...'.format(a))
            subset = transition_data[transition_data.a == a].copy()
            change_in_velocity = subset.groupby(subset.pos //
10).d_vel.apply(lambda x: x.value_counts() / len(x))
            change_in_position = subset.groupby(subset.vel //
10).d_pos.apply(lambda x: x.value_counts() / len(x))
            transition_matrix = dok_matrix((50000, 50000))
            for s in range(50000):

                pos = s % 500
                vel = s // 500

                # absorbing state
                if pos + 0.3 * vel > 475:
                    continue

                # hit wall
                if pos + 0.35 * vel <=16:
                    transition_matrix[s, 25000] += 1
```

```
                    continue

                pos_idx = pos // 10
                pos_idx = max(min(change_in_velocity.index)[0], pos_idx)
                pos_idx = min(max(change_in_velocity.index)[0], pos_idx)
                while pos_idx not in change_in_velocity:
                    pos_idx += 1
                dv_table = change_in_velocity[pos_idx]

                vel_idx = vel // 10
                vel_idx = max(min(change_in_position.index)[0], vel_idx)
                vel_idx = min(max(change_in_position.index)[0], vel_idx)
                while vel_idx not in change_in_position:
                    vel_idx += 1
                dp_table = change_in_position[vel_idx]

                for dp_pair in zip(dp_table.index, dp_table):
                    for dv_pair in zip(dv_table.index, dv_table):
                        dp, proba_dp = dp_pair
                        dv, proba_dv = dv_pair
                        sp = max(0, min((pos + dp) + (vel + dv) * 500,
49999))
                        transition_matrix[s, sp] += proba_dp * proba_dv

            transitions.append(transition_matrix.tocsr())
        return transitions

    def value_iteration(self):
        conv = False
        iter = 0
        print('value iteration starts...')
        while(conv == False and iter < self.maxIter):
            for a in range(self.NA):
                self.Q[:,a] = self.R[a] + self.TP[a].dot(self.U)
            self.U_last = np.copy(self.U)
            self.U = np.amax(self.Q,axis=1)
            # print(self.U)
            if max(self.U-self.U_last) < self.epsilon:
                conv = True
            iter += 1
```

```
            print(iter)
        self.policy = np.argmax(self.Q,axis=1)+1

    def output_policy(self):
        with open('medium.policy','w+') as f:
            f.writelines([str(x) + '\n' for x in self.policy])

    # def max_likelihood_est(self,data):
    #     #map data state and action to python friendly index
    #     # print(np.shape(data)[0])
    #     self.TP2D = np.zeros((self.N_States,self.N_States))
    #     self.TP = []
    #     for i in range(self.N_Action):
    #         self.TP.append(self.TP2D)
    #     # # print(self.TP)
    #     # # two dictionaries
    #     self.TPC = {}
    #     self.TC = {}
    #     for i in range (np.shape(data)[0]):
    #         s = data[i][0]-1
    #         a = data[i][1]-1
    #         sp = data[i][3]-1
    #         if self.TC.get((s,a)):
    #             self.TC[s,a] +=1
    #         else:
    #             self.TC[s,a] =1
    #         if self.TPC.get((s,sp,a)):
    #             self.TPC[s,sp,a] +=1
    #         else:
    #             self.TPC[s,sp,a] =1
    #         self.R[s,a] = data[i][2]
    #     for keys in self.TPC:
    #         self.TP[keys[2]][keys[0],keys[1]] =
self.TPC[keys[0],keys[1],keys[2]]/self.TC[keys[0],keys[2]]
    #     print(self.TP)
    #     print('\n')
```

Large.py

```
import numpy as np
```

```
import matplotlib.pyplot as plt

class Large(object):
   def __init__(self,data,discount,learningRate):
       self.States = np.arange(1, 312021)
       N_States = np.size(self.States)
       self.Actions = np.arange(1, 10)
       N_Action = np.size(self.Actions)
       self.NS = N_States
       self.NA = N_Action
       print(self.NS,self.NA)
       self.gamma = discount
       self.policy = np.zeros(N_States)
       self.alpha = learningRate
       self.Q = np.zeros((N_States,N_Action))
       self.U = np.zeros(N_States)


   def Q_learning(self,data,Sarsa):
       for i in range (np.shape(data)[0]):
           if (i==np.shape(data)[0]-1 or data[i][3] != data[i+1][0]):
               continue
           s = data[i][0]-1
           a = data[i][1]-1
           r = data[i][2]
           sp = data[i][3]-1
           ap = data[i+1][1]-1
           self.Q_last = np.copy(self.Q)
           if Sarsa:
               self.Q[s, a] += self.alpha * (r + self.gamma *
self.Q[sp,ap] - self.Q[s, a])
           else:
               self.Q[s, a] += self.alpha * (r + self.gamma *
max(self.Q[sp]) - self.Q[s, a])
       self.U = np.amax(self.Q,axis=1)
       # print([i for i in self.U if i!=0])
       for s in range(self.NS):
           if not np.any(self.Q[s]):
               self.policy[s] = np.random.randint(1,self.NA+1)
           else:
```

```
                self.policy[s] = np.argmax(self.Q[s]) + 1
        Convergence = np.linalg.norm(self.Q_last - self.Q)
        print(Convergence)


    def output_policy(self):
        with open('large.policy','w+') as f:
            f.writelines([str(x) + '\n' for x in self.policy])
```