Project 2: Reinforcement Learning

Deep Dayaramani deepdaya@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Descriptions

1.1 Small Data Set

For the small dataset, a value optimization program was applied. Since the dataset was small and the transition probabilities could be identified by the provided data, a Maximum Likelihood Estimation Method was applied to measure $\mathcal{T}(s'|s,a)$ along with $\mathcal{R}(s,a)$. After getting this data, a value optimization in the form of an LP problem was applied to get the best values for the Utility function. From the utility function, an optimal policy was obtained. The overall program took around 2 minutes 30s to run. The score on the leaderboard was reported to be 26.579 as compared to the random policy. The program did well in computing a good policy in a small amount of time! A brief description of the program is given below:

```
class MDP:
    S     # State Space
    A     # Action Space
    T     # Transition Probability Matrix
    R     # Reward Matrix
    SU    # Number of States explored
    gamma # Discount Factor
    Type  # DataSet Type

class ValueFunctionPolicy:
    def optimization_func(MDP):
        Minimize the value of U(s) given R(s,a) and T(s_dash |s,a).

    def policy_finder(U):
        Find the action given U(s). Return pi(s)
```

1.2 Medium and Large Data Set

For the medium data set and the large data set, a model-free Q-learning method was applied to find the optimal policy. This Q-learning model didn't contain exploration as exploration data was provided in the form of the datasets. Due to the relatively unknown structure of the large dataset, and the ambiguity in the structure of the discretized nature of the medium dataset, a model-free method had to be applied which ignores the Transition probabilities matrix. So our Q matrix is initialized and let run row-by-row through the dataset. We repeat this process a bunch of times, in order to reach a somewhat convergent Q matrix. This is done either through the limit set on iterations or by checking the largest change made in the Q matrix. Using this Q matrix, we find the best policy, using argmax Q(s,a).

For the medium dataset, another function was included to set unknown state actions to similar actions as their neighbors. A similar function was written up for the large dataset based on the pattern in the data found, but unfortunately didn't provide good enough results to prove that it was functional. The medium dataset took 7.24 minutes to run, and gave a score of 112.7. The large dataset took 6.31 minutes to run and gave a score of 57.88.

```
class Q-learning:
    MDP     #The MDP for the Problem
    Q       # Q function
    alpha   # Alpha for the Q function

    def update_Q():
        for i in SU:
            for a in A:
                Find r, max(Q(sdash, adash))
                update Q(s,a)

    def compute_policy():
        while not converged:
            update_Q()
        for i in S:
            for j in A:
                get Q(s,a)
        pi(s) = argmax Q(s,a)
        return pi
```

2. Code

```
import collections
import pandas as pd
import sys
import cvxpy
class MDP:
    def __init__(self,S=0,A=0,T={},R={},statesUsed = 0, keysT = {},type="",
    data=[]):
        self.S = S
        self.A = A
        self.T = T
        self.R = R
        self.statesUsed = statesUsed
        self.keysT = keysT
        self.data= data
        if (type == "small") | (type == "large"):
            self.gamma = 0.95
        else:
            self.gamma = 1
        self.type = type
```

```
        self.__doc__ = "This class stores a MDP for a given State Space S,
    Action Space A, and Reward, Transition p" \
                       "probability space. We can find a policy using our
    preferred method in the write policy function."

    def writepolicy(self,filename_in, filename_out):
        policy = {}
        if self.type == "small":
            vp = ValueFunctionPolicy(self, [])
            policy = vp.optimization_small()
        else:
            ql = Q_Learning(self, alpha = 0.05, iterations=30)
            print("here")
            policy = ql.computePolicy()
        with open(filename_out, 'w') as f:
            for value in policy.values():
                f.write("{0}\n".format(value))

class Q_Learning:
    def __init__(self, MDP,alpha = 0.1, iterations = 100):
        self.Q = collections.defaultdict(int)
        self.alpha = alpha
        self.MDP = MDP
        self.iterations = iterations
        self.data = MDP.data
        self.absStates = list(set(self.data["s"]) ^ set(self.data["sp"]))
        self.__doc__ = "This class performs Q_learning on a Problem MDP for a
     given Alpha and number of iterations."


    def updateQ(self):
        Q = self.Q
        delta = 0
        it = 0
        print(len(self.data["s"].unique()))
        for i in self.data["s"].unique():
            if it%100 == 0: print(it)
            it+=1
            s = i
            if s in self.absStates:
                continue
            possibleActions = self.data.loc[self.data["s"] == s]["a"].unique
    ()
            for a in possibleActions:
                biggest_update = Q[(s,a)]
                reward, sp = self.find_max_next_state(s,a)
                if sp in self.absStates:
                    maxQS = 0
                else:
                    maxQS = self.maxQ(sp)
```

```
                Q[(s,a)] = self.alpha*(reward + self.MDP.gamma*maxQS - Q[(s,
a)])
                biggest_update = Q[(s,a)] - biggest_update
            if biggest_update > delta:
                delta = biggest_update
        self.Q = Q
        return delta

    def computePolicy(self):
        convergence = 10
        threshold = 0.0001
        ite = 1
        while (ite < self.iterations) & (convergence > threshold):
            print("Episode {0}".format(ite))
            convergence = self.updateQ()
            print(convergence)
            ite += 1
        print("Convergence reached. Initializing Policy")
        pi = {}
        for i in range(self.MDP.S):
            res = {}
            for j in range(self.MDP.A):
                res[j+1] = self.Q[(i+1,j+1)]

            pi[i+1] = max(res, key=lambda key: res[key])
        if (self.MDP.type == "medium"):
            pi = self.nearby_states(pi)
        elif self.MDP.type == "large":
            pi =  self.same_action(pi)
        return pi


    def nearby_states(self, policy):
        statesUsed = list(set(self.data["s"].unique()).union(set(self.data["
sp"])))
        for i in range(self.MDP.S):
            if i+1 not in statesUsed:
                k = i+1
                lst = statesUsed
                policy[i+1] = policy[lst[min(range(len(lst)), key=lambda i:
abs(lst[i] - k))]]
        return policy

    def same_action(self,policy):
        statesUsed = list(set(self.data["s"].unique()).union(set(self.data["
sp"])))
        for i in range(self.MDP.S):
            if i+1 not in statesUsed:
                policy[i+1] = 9
        return policy
```

```
    # def big_brain_time(self, policy):
    #     statesUsed = self.data["s"].unique()
    #     for i in range(self.MDP.S):
    #         if i+1 not in statesUsed:
    #             if i+1 > 16000:
    #                 if (i+1)%1000 == 1:
    #                     policy[i+1] = 1
    #                 elif (i+1)%100 == 2:
    #
    #     return policy



    def find_max_next_state(self,s,a):
        max_next_action_reward = self.data.loc[self.data["s"]==s].loc[self.
    data["a"]==a].groupby("s").max("r")
        reward = max_next_action_reward["r"].values[0]
        sp = max_next_action_reward["sp"].values[0]
        return reward, sp

    def maxQ(self, sdash):
        Qdash = []
        possibleNextActions = self.data.loc[self.data["s"] == sdash]["a"].
    unique()
        for i in possibleNextActions:
            Qdash.append(self.Q[(sdash,i)])
        return max(Qdash)



class ValueFunctionPolicy:
    def __init__(self,MDP, U):
        self.MDP = MDP
        self.U = U

    def optimization_small(self):
        U = cvxpy.Variable(self.MDP.S)
        constraints = []
        for i in range(self.MDP.S):
            for a in range(self.MDP.A):
                keys = [(i+1,a+1,j) for j in self.MDP.keysT[i+1][a+1]]
                constraints += [U[i] >= self.MDP.R[(i+1,a+1)] + self.MDP.
    gamma*sum([self.MDP.T[key] * U[key[2]-1] for key in keys])]
        problem = cvxpy.Problem(cvxpy.Minimize(sum(U)),constraints)
        problem.solve(solver="ECOS")
        U = U.value
        self.U = U
        return self.greedy_policy_finder()
```

```
    def greedy_policy_finder(self):
        pi = {}
        for i in range(self.MDP.S):
            res = {}
            for j in range(self.MDP.A):
                res[j+1] = (self.lookahead(i+1,j+1))
            pi[i+1] = max(res, key=lambda key: res[key])
        return pi

    def lookahead(self, s, a):
        S,T,R,gamma,keysT = self.MDP.S, self.MDP.T, self.MDP.R, self.MDP.
    gamma, self.MDP.keysT
        keys = [(s,a,j) for j in keysT[s][a]]
        return R[(s,a)] + gamma * sum([T[key]*self.U[key[2]-1] for key in
    keys])

class Problem():
    def __init__(self,filename, filename_out,type=""):
        self.filename = filename
        self.filename_out = filename_out
        self.type = type

    def read_data(self):
        data = pd.read_csv(self.filename)
        return data

    def compute_policy_and_parsing_func(self):
        data = self.read_data()
        numberStates = max(data["s"].unique())
        if (self.type == "small") & (numberStates != 100):
            numberStates = 100
        elif (self.type == "medium") & (numberStates!= 50000):
            numberStates = 50000
        else:
            numberStates = 312020

        numberActions = max(data["a"].unique())
        statesUsed = len(data["s"].unique())
        rewards = collections.defaultdict(int)
        transition = collections.defaultdict(int)
        s_a_sdash = collections.defaultdict(int)
        if self.type == "small":
            for i in range(numberStates):
                ind_state = data.loc[data["s"] == i + 1].index.values
                dict_a_sdash = data.iloc[ind_state].groupby(["a", "sp"]).
    groups
                n_sa = {}
                for action in range(numberActions):
                    ind_action = data.loc[ind_state].loc[data["a"] == action
    + 1].index.values
```

```
                    n_sa[action + 1] = len(ind_action)
                    rewards[(i + 1, action + 1)] = data.loc[ind_state].loc[
    ind_action]["r"].max()
                for j in dict_a_sdash.keys():
                    if n_sa[j[0]] == 0:
                        transition[(i + 1, j[0], j[1])] = 0
                    else:
                        transition[(i + 1, j[0], j[1])] = len(dict_a_sdash[j
    ]) / n_sa[j[0]]
                    if i + 1 in s_a_sdash.keys():
                        if j[0] in s_a_sdash[i + 1].keys():
                            s_a_sdash[i + 1][j[0]].append(j[1])
                        else:
                            s_a_sdash[i + 1][j[0]] = [j[1]]
                    else:
                        s_a_sdash[i + 1] = collections.defaultdict(int, {j
    [0]: [j[1]]})
            return MDP(S=numberStates,A=numberActions,T=transition,R=rewards,
                   statesUsed = statesUsed, keysT = s_a_sdash,type="small",
    data=data).writepolicy(self.filename, self.filename_out)
        else:
            return MDP(S=numberStates,A=numberActions,T=transition,R=rewards,
                   statesUsed = statesUsed, keysT = s_a_sdash,type="medium",
    data=data).writepolicy(self.filename, self.filename_out)

def main():
    inputfile = sys.argv[1]
    outputfile = sys.argv[2]
    type = sys.argv[3]
    print(type)
    prob = Problem(inputfile, outputfile, type)
    prob.compute_policy_and_parsing_func()

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