Project 2: Reinforcement Learning

Brian Dobkowski BDOBKOWS@STANFORD.EDU
AA228/CS238, Stanford University

1. Algorithm Descriptions

1.1 General Experiments for all Datasets

For the small dataset, I iterated with a number of different methods. First, I implemented a standard Q-learning algorithm. I tested discount factors (γ) ranging from 0.3 to 0.95, at roughly $\frac{1}{10}$ increments. I also ran the Q-learning algorithm on several iterations of the data (I settled on a value around 30), while shuffling the dataset between each iteration. Next, I implemented a Qλ-learning algorithm, with λ values between 0.3 and 0.8 tested. I was able to vectorize the update rule to compute $Q(s,a) \forall s \in \mathcal{S}, a \in \mathcal{A}$ in one step rather than creating a double loop. This reduced my rutime by about 5x.

Next, I looked at model-based methods, trying to estimate the transition probabilities, $T(s,a,s')$. I first implemented a maximum likelihood model that kept counts of every time a state-action-next-state combination ($N(s,a,s')$) was visited. The transition probabilities were then set as $T(s,a,:) = \frac{N(s,a,:)}{\sum_{s'} N(s,a,s')}$. Another model-learning method I employed was Bayesian model learning. I used a uniform Dirichlet prior distribution to model the probability of switching states to any s' from a given state and action, s, a. Thus, I had $|\mathcal{S}| \times |\mathcal{A}|$ distinct Dirichlet distributions with α vectors of size $|\mathcal{S}|$, whose elements were all equal to 1. To update this prior, I kept a count of every time a state-action-next-state combination was visited, and updated the $\alpha(s')$ value according to the Bayesian equations for creating a posterior Dirichlet distribution. The reward function did not have to be estimated using either of these approaches, since the rewards were given in the problem and were constant based on the state and action.

With this model of a Markov Decision Process built, I was able to implement value iteration to solve the optimal value function. I initialized the value function to an array of length $|\mathcal{S}|$ of all 0s, and found 1500 iterations to be sufficient (and relatively quick) for the algorithm to converge on an optimal value function. To extract the optimal policy from the optimal value function, I selected the greedy policy using the one-step lookahead and taking $arg\max_{a'} Q(s,a')$.

1.2 Small Data Set

The aforementioned value iteration, combined with the maximum likelihood estimation of the transition probabilities, was the final approach I used in solving for the optimal policy for the small dataset. I ran value iteration for 1500 iterations before I found convergence, and I used a γ factor of 0.8. The extracted optimal policy was done using the greedy method with a one-step lookahead as mentioned above. The runtime of this algorithm was 25.9s.

1.3 Medium Data Set

The increased dimensionality of the medium dataset proved a great challenge to modeling the transition probabilities and building an MDP to use value iteration and other methods. The policy itself would be of too high dimensionality to conduct policy search, or policy gradient estimation and optimization (the number of possible policies would be $|\mathcal{A}|^{|\mathcal{S}|}$, which is 7^{50000} in the medium dataset, a ludicrously large number. Also, the arrays for the counts and transitions, $N(s, a, s')$, $T(s, a, s')$ were simply too large to render in memory on my computer, which ruled out the maximum likelihood and Bayesian-based methods to model an MDP.

Thus, in this problem, I was left to use model-free methods. I implemented the same Q-learning and Qλ-learning algorithms from the previous problem, and iterated over different collections of hyperparameters. It turned out that regular Q-learning worked better for me, and I ended up running regular Q-learning over 30 iterations of the given data, shuffled between each iteration. My discount factor was 0.8, and the learning parameter α was 0.2. The runtime of this algorithm was 4 minutes and 2.6s.

1.4 Large Data Set

The number of states and actions increased in the larger dataset, so clearly a transition model could not be estimated using maximum likelihood or Bayesian methods for this dataset, since the medium dataset proved too large to represent in memory. For this dataset, I tried the same Q-learning and Qλ-learning algorithms as for the medium dataset. I tinkered around with the learning rate, α, and the γ and λ values again, but out of the options I tried, Q-learning seemed to yield the best results with the same values as in the prior dataset: $\alpha = 0.2$, $\gamma = 0.8$. The algorithm was again run for 30 iterations, and its runtime was 4 minutes and 4.4 seconds.

2. Code

```
#!/usr/bin/env python3

import sys
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import scipy.special as sp
import random
from tqdm import tqdm
import datetime
from multiprocessing import Process, Manager, Pool
from scipy.stats import dirichlet

pool = Pool(processes=7)

def set_size_action(size):
    if size == 'small':
```

```
        S     = np.arange(1,101)
        A     = np.arange(1,5)
    elif size == 'medium':
        S     = np.arange(1,50001) # state space
        A     = np.arange(1,8) # action space
    elif size == 'large':
        S     = np.arange(1,312021) # state space
        A     = np.arange(1,10) # action space
    return S, A

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

class MDP():
    def __init__(self, S, A, T, R, gamma):
        self.gamma = gamma
        self.S, self.A = S, A
        self.T = T
        self.R = R
        self.U = np.zeros(len(S))
        self.Q = np.zeros((len(self.S), len(self.A)))

    def lookahead(self, U, s, a):
        my_sum = 0
        for sp in range(len(self.S)):
            my_sum += self.T[s,a,sp]*U[sp]
        return self.R[s,a] + self.gamma*my_sum

    def backup(self, U, s):
        return np.max([self.lookahead(U, s, a) for a in range(len(self.A))])

    def solve_value_iteration(self, kmax):
        U = np.zeros(len(self.S))

        print('solving value iteration')
        for k in tqdm(range(kmax)):
            U = np.array([self.backup(U, s) for s in range(len(self.S))])

        # greedily extract optimal policy from value function
        for s in range(len(self.S)):
            self.Q[s] = np.array([self.lookahead(U, s, a) for a in range(len(
    self.A))])
        return argmax_random(self.Q)

class MDP_Bayes():
    def __init__(self, S, A, D, R, gamma):
        self.gamma = gamma
        self.S, self.A = S, A
```

```
        self.D = D
        self.R = R
        self.U = np.zeros(len(S))
        self.Q = np.zeros((len(self.S), len(self.A)))

    def lookahead(self, U, s, a):
        my_sum = 0
        n = np.sum(self.D[s][a].alpha)
        for sp in range(len(self.S)):
            # import pdb;pdb.set_trace()
            # my_sum += self.D[s][a].sample()*U[sp]
            my_sum += self.D[s][a].alpha[sp]/n*U[sp]
        return self.R[s,a] + self.gamma*my_sum

    def backup(self, U, s):
        return np.max([self.lookahead(U, s, a) for a in range(len(self.A))])

    def solve_value_iteration(self, kmax):
        U = np.zeros(len(self.S))

        print('solving value iteration')
        for k in tqdm(range(kmax)):
            U = np.array([self.backup(U, s) for s in range(len(self.S))])

        # greedily extract optimal policy from value function
        for s in range(len(self.S)):
            self.Q[s] = np.array([self.lookahead(U, s, a) for a in range(len(
    self.A))])
        return argmax_random(self.Q)

class QLearning():
    def __init__(self, data, size):
        self.S, self.A = set_size_action(size)
        self.gamma = 0.8 # discount factor
        self.alpha = 0.2# learning rate
        self.Q     = np.zeros((len(self.S), len(self.A)))

    def update(self, s, a, r, sp):
        self.Q[s,a] += self.alpha*(r + self.gamma*np.max(self.Q[sp,:]) - self
    .Q[s,a])

    def optimize(self, data, size, iterations):
        for iter in tqdm(range(iterations)):
            data = data.sample(frac=1)
            for idx, row in data.iterrows():
                self.update(row.s-1, row.a-1, row.r, row.sp-1)
        optimal_policy = argmax_random(self.Q)
        return optimal_policy

class QLambdaLearning(QLearning):
```

```
    def __init__(self, data, size):
        QLearning.__init__(self, data, size)
        self.lam = 0.3
        self.N   = np.zeros((len(self.S), len(self.A)))

    def update(self, s, a, r, sp):
        self.N[s,a] += 1
        delta  = r + self.gamma*np.max(self.Q[sp,:]) - self.Q[s,a]
        self.Q += self.alpha*np.multiply(delta,self.N)
        self.N = self.gamma*self.lam*self.N

    def optimize(self, data, size, iterations):
        for iter in tqdm(range(iterations)):
            data = data.sample(frac=1)
            self.N = np.zeros((len(self.S), len(self.A)))
            for idx, row in tqdm(data.iterrows()):
                self.update(row.s-1, row.a-1, row.r, row.sp-1)
        optimal_policy = argmax_random(self.Q)
        return optimal_policy


class MaximumLikelihoodMDP():
    def __init__(self, data, size):
        self.S, self.A = set_size_action(size)
        self.N     = np.zeros((len(self.S), len(self.A), len(self.S)))
        self.rho   = np.zeros((len(self.S), len(self.A)))
        self.gamma = 0.8
        self.U     = np.zeros((len(self.S)))

    def MDP(self):
        T = np.zeros_like(self.N); R = np.zeros_like(self.rho)
        for s in range(len(self.S)):
            for a in range(len(self.A)):
                n = np.sum(self.N[s,a,:])
                if n == 0:
                    T[s,a,:] = 0.0
                    R[s,a]   = 0.0
                else:
                    T[s,a,:] = self.N[s,a,:]/n
                    R[s,a]   = self.rho[s,a]/n
        return MDP(self.S, self.A, T, R, self.gamma)

    def update(self, s, a, r, sp):
        self.N[s,a,sp] += 1
        self.rho[s,a]  += r

class DirichletPDF():
    def __init__(self, S):
        self.alpha = np.ones(len(S))
```

```
    def sample(self):
        return np.random.dirichlet(self.alpha)

class BayesianMDP():
    '''
    Builds a an MDP using a uniform Dirichlet prior for all state-action
    pairs,
    and updates the posterior based on counts of visits to that state-action
    pair.
    '''
    def __init__(self, data, size):
        self.S, self.A = set_size_action(size)
        self.R = np.zeros((len(self.S), len(self.A)))
        self.gamma = 0.8
        self.U = np.zeros((len(self.S)))
        self.D = []
        for s in range(len(self.S)):
            A = []
            for a in range(len(self.A)):
                dirichlet = DirichletPDF(np.ones(len(self.S)))
                A.append(dirichlet)
                r = np.unique(data.loc[(data.a-1==a) & (data.s-1==s)].r.
    values)
                if len(r) == 0:
                    self.R[s,a] = 0
                elif len(r) > 1:
                    print('something is wrong')
                    import pdb;pdb.set_trace()
                else:
                    self.R[s,a] = r
            self.D.append(A)

    def update(self, s, a, sp):
        self.D[s][a].alpha[sp] += 1

    def MDP(self):
        return MDP_Bayes(self.S, self.A, self.D, self.R, self.gamma)

def compute(inputfile, outputfile):
    '''
    This function computes the optimal policy based on a variety of methods.
    Certain methods may be commented out as they are chosen and iterated upon
    .

    Parameters
    ----------
    inputfile : data file containing states, rewards, actions, next states
    outputfile : .policy file representing optimal policy solution

    Returns
```

```
    -------
    None.

    '''
    data = pd.read_csv(inputfile)
    size = inputfile.split('/')[-1].split('.')[0]
    begin_time = datetime.datetime.now()

    # Model Learning - Maximum Likelihood
    # model = MaximumLikelihoodMDP(data, size)
    # for idx, row in data.iterrows():
    #     model.update(row.s-1, row.a-1, row.r, row.sp-1)
    # MDP = model.MDP()
    # optimal_policy = MDP.solve_value_iteration(1500)

    # Model Learning - Bayesian MDP
    # model = BayesianMDP(data, size)
    # for idx, row in data.iterrows():
    #     model.update(row.s-1, row.a-1, row.sp-1)
    # MDP = model.MDP()
    # optimal_policy = MDP.solve_value_iteration(1500)

    # Q-Learning
    # model          = QLearning(data, size)
    # optimal_policy = model.optimize(data, size, 30)

    # Q-Lambda-Learning
    print(len(data))
    model          = QLambdaLearning(data, size)
    optimal_policy = model.optimize(data, size, 3)

    write_policy(optimal_policy, outputfile)
    print(datetime.datetime.now() - begin_time)

def argmax_random(Q):
    '''
    This function returns the action that maximizes the action value function
    over all states in the system. If there is a tie among the values for a
    given action, this function returns a random action among those that
    tied.

    Parameters
    ----------
    Q : Action Value function

    Returns
    -------
    np.array((len(S)))
        optimal policy
```

```
    '''
    opt_policy = np.zeros(len(Q))
    for idx, row in enumerate(Q):
        maxval = np.max(row)
        if np.sum(row==maxval) > 1:
            opt_policy[idx] = int(np.random.choice(np.where(row==maxval)[0]))
        else:
            opt_policy[idx] = int(np.argmax(row))
    return opt_policy + 1 # adding 1 for python indexing, actions are
    represented starting at 1

def write_policy(policy, filename):
    with open(filename, 'w') as f:
        for action in policy:
            f.write(str(action)+'\n')

def main():
    if len(sys.argv) != 3:
        raise Exception("usage: python project1.py <infile>.csv <outfile>.
    policy")
    inputfilename  = sys.argv[1]
    outputfilename = sys.argv[2]
    compute(inputfilename, outputfilename)


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