Project 2: Reinforcement Learning

Brandon Franz franzb@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Descriptions

1.1 Small Data Set

For the small data set, I chose to start with a model-based reinforcement learning method. Due to its simplicity, I started with the Maximum Likelihood Model. To process the small data set, I counted the number of transitions from state s to state s' for an action a, and accumulated the rewards r for each action a taken from state s. To get a transition model, I normalized the number of transitions by the number of times an action a was taken from state s. The reward model was found by normalizing the accumulated reward by the number of times an action a was taken from state s. Exact value iteration was used to calculate the optimal value function from the transition and reward models. The policy is the actions corresponding to the optimal value function.

In one run, the method ran in 5.23 seconds.

1.2 Medium Data Set

For the medium data set, I attempted to implement the same model-based reinforcement learning method from the Small Data Set. However, the size of the state space made this method intractable. Therefore, I switched to a model-free method. Since we are using a data set, no exploration was implemented. I decided to use Q-learning to learn the action value function due to its simplicity in implementation. I started with an arbitrary learning rate of 0.5. This turned out to be enough to achieve positive reward from the policy generated.

In one run, the method ran in 1.99 seconds for one iteration through the entire data set with a learning rate of 0.5

1.3 Large Data Set

For the large data set, I started with the same method as the medium data set. I managed to produce positive reward from the policy generated, but only barely. I decided to implement epochs to increase the reward, with one epoch consisting of the entire large data set. I also reduced the learning rate to 0.1. This resulted in higher reward.

In one run, the method ran in 6.59 seconds for 5 epochs with a learning rate of 0.1.

2. Code

2.1 RL.py

Contains classes for each data set. Test scripts were used to run instances of the classes with different parameters.

```
from utils import *
import time

class SmallRL():
    '''
    Description of Data Set
    10 x 10 grid world (100 states) with 4 actions. Actions are 1: left, 2:
    right, 3: up, 4: down. The discount factor is 0.95.
    '''

    def __init__(self, discount=0.95, max_iterations=100):
        _, sarsp = importCSV("data/small.csv")
        self.s = sarsp[:,0]
        self.a = sarsp[:,1]
        self.r = sarsp[:,2]
        self.sp = sarsp[:,3]

        self.n_states = 100
        self.n_actions = 4
        self.k_max = max_iterations
        self.gamma = discount

        self.S = range(1,self.n_states+1)
        self.A = range(1, self.n_actions+1)

        self.N = defaultdict(float)
        self.n = defaultdict(float)

        self.rho = defaultdict(float)

        self.T = defaultdict(float)
        self.R = defaultdict(float)

        self.U = defaultdict(float)
        self.policy = defaultdict(int)

    def countTransitions(self):
        for i in range(len(self.s)):
            s, a, sp = self.s[i], self.a[i], self.sp[i]
            self.N[(s,a,sp)] += 1.

    def countTransitionsFromState(self):
        for s in self.S:
            for a in self.A:
                for sp in self.S:
                    self.n[(s,a)] += self.N[(s,a,sp)]

    def accumulateReward(self):
        for i in range(len(self.s)):
            s, a = self.s[i], self.a[i]
            self.rho[(s,a)] += self.r[i]

    def calculateTransitionDist(self):
        for s in self.S:
            for a in self.A:
                for sp in self.S:
                    if self.n[(s,a)] == 0:
                        self.T[(s,a,sp)] = 0.
                    else:
                        self.T[(s,a,sp)] = self.N[(s,a,sp)] / self.n[(s,a)]

    def calculateRewardDist(self):
        for s in self.S:
            for a in self.A:
                if self.n[(s,a)] == 0:
                    self.R[(s,a)] = 0.
                else:
                    self.R[(s,a)] = self.rho[(s,a)] / self.n[(s,a)]

    def lookahead(self, s, a): # for one state-action pair
        if self.n[(s,a)] == 0:
            return 0.
        return self.R[(s,a)] + self.gamma * sum([self.T[(s,a,sp)]*self.U[sp]
    for sp in self.S])

    def backup(self, s): #for every state-action pair at one state
        lookaheads = np.asarray([self.lookahead(s, a) for a in self.A])
        return np.max(lookaheads), lookaheads.argmax()+1


    def solve(self):
        start = time.time()

        self.countTransitions()
        self.countTransitionsFromState()
        self.accumulateReward()
        self.calculateTransitionDist()
        self.calculateRewardDist()

        for k in range(self.k_max):
            for s in self.S:
                self.U[s], self.policy[s] = self.backup(s)
        end = time.time()

        print "Runtime: "
        print end-start
        actions = [self.policy[s] for s in self.S]
        writePolicy("small.policy", actions)
        print "Policy Written to File"

class MediumRL():
    '''
    Description of Data Set
    MountainCarContinuous-v0 environment from Open AI Gym with altered
    parameters.
    State measurements are given by integers with 500 possible position
    values and
    100 possible velocity values (50,000 possible state measurements). 1+pos
    +500*vel
    gives the integer corresponding to a state with position pos and velocity
     vel.
    There are 7 actions that represent different amounts of acceleration.
    This problem
    is undiscounted, and ends when the goal (the flag) is reached. Note that
    since the
    discrete state measurements are calculated after the simulation, the data
     in medium.csv
    does not quite satisfy the Markov property.
    '''

    def __init__(self, discount=1, learning_rate=0.5, max_iterations=1000):
        _, sarsp = importCSV("data/medium.csv")
        self.s = sarsp[:,0]
        self.a = sarsp[:,1]
        self.r = sarsp[:,2]
        self.sp = sarsp[:,3]

        self.n_states = 50000
        self.n_actions = 7
        self.k_max = max_iterations
        self.gamma = discount
        self.alpha = learning_rate

        self.S = range(1,self.n_states+1)
        self.A = range(1, self.n_actions+1)

        self.Q = defaultdict(float)

        self.U = defaultdict(float)
        self.policy = defaultdict(int)

    def learning_step(self, s, a, r, sp):
        self.Q[(s,a)] += self.alpha * (r + self.gamma * max([self.Q[(sp,a)]
    for a in self.A]) - self.Q[(s,a)])

    def extractPolicy(self, s): #for every state-action pair at one state
        Q = np.asarray([self.Q[(s,a)] for a in self.A])
        return np.max(Q), Q.argmax()+1

    def solve(self):
        start = time.time()

        for i in range(self.k_max):
            self.learning_step(self.s[i], self.a[i], self.r[i], self.sp[i])

        for s in self.S:
            self.U[s], self.policy[s] = self.extractPolicy(s)

        end = time.time()

        print "Runtime: "
        print end-start
        actions = [self.policy[s] for s in self.S]
        writePolicy("medium.policy", actions)
        print "Policy Written to File"

class LargeRL():
    '''
    Description of Data Set
    It's a secret. MDP with 312020 states and 9 actions, with a discount
    factor of 0.95.
    This problem has a lot of hidden structure. Look at the transitions and
    rewards carefully
    and you might figure some of it out!
    '''

    def __init__(self, discount=0.95, learning_rate=0.5, max_iterations=1000,
     epochs=1):
        _, sarsp = importCSV("data/large.csv")
        self.s = sarsp[:,0]
        self.a = sarsp[:,1]
        self.r = sarsp[:,2]
        self.sp = sarsp[:,3]

        self.n_states = 312020
        self.n_actions = 9
        self.k_max = max_iterations
        self.gamma = discount
        self.alpha = learning_rate
        self.epochs = epochs

        self.S = range(1,self.n_states+1)
        self.A = range(1, self.n_actions+1)

        self.Q = defaultdict(float)

        self.U = defaultdict(float)
        self.policy = defaultdict(int)

    def learning_step(self, s, a, r, sp):
        self.Q[(s,a)] += self.alpha * (r + self.gamma * max([self.Q[(sp,a)]
    for a in self.A]) - self.Q[(s,a)])

    def extractPolicy(self, s): #for every state-action pair at one state
        Q = np.asarray([self.Q[(s,a)] for a in self.A])
        return np.max(Q), Q.argmax()+1

    def solve(self):
        start = time.time()

        for i in range(self.epochs):
            for i in range(self.k_max):
                self.learning_step(self.s[i], self.a[i], self.r[i], self.sp[i
    ])

        for s in self.S:
            self.U[s], self.policy[s] = self.extractPolicy(s)

        end = time.time()

        print "Runtime: "
        print end-start
        actions = [self.policy[s] for s in self.S]
        writePolicy("large.policy", actions)
        print "Policy Written to File"
```

2.2 utils.py

Contains imports and common functions for reading and writing files

```
import pandas as pd
import numpy as np
from scipy.sparse import coo_matrix
from collections import namedtuple, defaultdict

def writePolicy(filename, actions):
    np.savetxt(filename, actions, fmt='%i', delimiter=",")

def importCSV(filename):
    contents = pd.read_csv(filename)
    return contents.columns, contents.to_numpy()
```