Project 1: Bayesian Structure Learning

Thomas Sounack tsounack@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

For every dataset, the policy is found using the sarsa lambda algorithm, a method of Q learning. The algorithm is implemented using Object Oriented Programming. The **main.py** file is used to retrieve the data from a csv file using numpy and instantiate a PolicyFinder object for each dataset. The reason numpy is used instead of pandas is to provide faster computation times - although pandas is a great tool to visualize data, it is not as efficient as numpy when it comes to working with very large datasets.

The **policyfinder.py** file defines the PolicyFinder class.

1.1 Implementing the sarsa lambda algorithm

To implement the sarsa lambda algorithm, we are using a matrix N that maintains an exponentially decaying visit count. For every (state, action, reward, next state) row in the dataset, we are computing the next action using an epsilon greedy policy (which we will come back to in the next subsection). Then we increment the cell in the matrix corresponding to the current state and action by one, and compute the temporal difference update. Finally, to update all elements of the action value function and of the visit counts, we can use matrices operation (which are more efficient than nested for loops).

1.2 Implementing the epsilon greedy policy

In order to implement the epsilon greedy policy, which as mentioned above is used to find the next action for the next state, we pass as argument the next state. Then, with a probability given by a hyperparameter epsilon we choose a random next action, or with probability (1 - epsilon) we choose the best action according to our current action value function.

1.3 Choosing the parameters

Our class PolicyFinder therefore takes several arguments: some of them are given by the problem (dataset, number of states, number of actions, discount factor) and others are hyperparameters that can be optimized (learning factor, decay factor and epsilon). There are several ways these parameters can be optimised, including grid search, random search and bayesian search. However, this would be quite a lengthy process since we do not have a way of measuring our performance, apart from manually uploading our policy to gradescope. We therefore chose the hyperparameters according to usual values, although it is important to note that they can be far from optimal. The selected values are (learning factor: 0.1, decay factor: 0.5, epsilon: 0.1).

1.4 Outputing the results

The PolicyFinder class also includes a method that takes a path as argument and extracts the policy from the final action value function to write it on a .policy file. This is simply done by iterating over all the states and writing the action that has the highest Q value.

2. Runtime for each problem

The runtime for each problem is obtained using the time library. We are also using the tqdm library to visualize where our program is in the policy finding process. We obtain the following results:

Runtime for each dataset		
Small	Medium	Large
0.6 seconds	25 seconds	3.5 minutes

3. Code

```
import numpy as np
import time

from policyfinder import PolicyFinder

# SMALL

data_small = np.genfromtxt('data/small.csv', delimiter=',', dtype=int)[1:]
small = PolicyFinder(data = data_small,
                     nb_state = 100,
                     nb_action = 4,
                     discount = 0.95,
                     learning = 0.1,
                     decay = 0.5,
                     epsilon = 0.1
                     )

t1_small = time.time()
small.sarsa_lambda()
t2_small = time.time()
small.save_policy_text("answer/small")
print("Time for Small: " + format(t2_small-t1_small, ".2f") + " seconds")



# MEDIUM

data_medium = np.genfromtxt('data/medium.csv', delimiter=',', dtype=int)[1:]
medium = PolicyFinder(data = data_medium,
                     nb_state = 50000,
                     nb_action = 7,
                     discount = 0.95,
                     learning = 0.1,
                     decay = 0.5,
                     epsilon = 0.1
                     )

t1_medium = time.time()
medium.sarsa_lambda()
t2_medium = time.time()
medium.save_policy_text("answer/medium")
print("Time for Medium: " + str(t2_medium-t1_medium))



# LARGE

data_large = np.genfromtxt('data/large.csv', delimiter=',', dtype=int)[1:]
```

```
large = PolicyFinder(data = data_large,
                     nb_state = 312020,
                     nb_action = 9,
                     discount = 0.95,
                     learning = 0.1,
                     decay = 0.5,
                     epsilon = 0.1
                     )

t1_large = time.time()
large.sarsa_lambda()
t2_large = time.time()
large.save_policy_text("answer/large")
print("Time for Large: " + str(t2_large-t1_large))
```

Listing 1: main.py

```
import numpy as np

from tqdm import tqdm

class PolicyFinder:
    """
    This class is used to find the best policy for a given dataset using the
    sarsa
    lambda algorithm.
    """

    def __init__(self, data: np.ndarray, nb_state: int, nb_action: int,
    discount: float,
                 learning: float, decay: float, epsilon: float) -> None:
        """
        This function initiates the object using various parameters entered
    by the user.
        """
        self.data      = data
        self.nb_state  = nb_state
        self.nb_action = nb_action

        self.discount   = discount
        self.learning   = learning
        self.decay      = decay
        self.epsilon    = epsilon

        self.Q = np.zeros((nb_state, nb_action))


    def epsilon_greedy(self, curr_state: int) -> int:
        """
```

```
        This function implements the epsilon greedy policy to choose the next
    action.
        It takes the current state as argument.
        """
        if np.random.uniform(0, 1) < self.epsilon:
            action = np.random.randint(0, self.nb_action - 1)
        else:
            action = np.argmax(self.Q[curr_state])
        return action


    def sarsa_lambda(self) -> None:
        """
        This function implements the sarsa lambda algorithm to compute Q.
        """
        N = np.zeros((self.nb_state, self.nb_action))

        for row in tqdm(self.data, total = self.data.shape[0], desc="Data"):
            s, a, r, sp = row
            s, a, sp = s-1, a-1, sp-1
            ap = self.epsilon_greedy(sp)

            N[s][a] += 1

            delta    = r + self.discount * self.Q[sp][ap] - self.Q[s][a]
            self.Q  += self.learning * delta * N
            N       *= self.discount * self.decay


    def save_policy_text(self, dir: str) -> None:
        """
        This function saves the policy as a .policy file.
        """
        with open(dir + ".policy", "w") as f:
            for state in range(self.nb_state):
                action = np.argmax(self.Q[state]) + 1
                f.write(str(action) + "\n")
```

Listing 2: policyfinder.py