Project 2: Reinforcement Learning

Zhemin Huang zheminh@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Descriptions

1.1 Small Data Set

I use the Q-Learning algorithm for this dataset. Q-Learning involves the incremental estimation of the action-value function as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a) \right)$$

In our code, this update step can be implemented as:

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

Since all possible states are covered in this dataset, I iterate over the entire dataset and apply the Q-Learning update to each state-action pair, selecting the optimal action accordingly.

1.2 Medium Data Set

For the Medium dataset, I still use using the Q-Learning algorithm. However, this problem is undiscounted and ends upon reaching the goal, so I set the discount factor to 1 to better align with this objective. Besides, some states are missing from our dataset. Without processing these missing states, their Q-values would default to zero. Assuming that states with similar indices have similar properties, I use a nearest-neighbor approach based on state indices to estimate the optimal policy for these missing states.
I try 100 epochs first but the result is a bit lower than the baseline. So I reduce the learning rate to 0.05 and increase the epochs to 1000 according to the EdStem's guidance. Finally I get a result far more better than the baseline.

1.3 Large Data Set

I also apply Q-Learning with a discount factor of $\gamma = 0.95$ to estimate the action-value function $Q(s,a)$ for the `large.csv` dataset, which includes 302,020 states and 9 actions. Since 99.84% of states are missing, I use the nearest neighbor approach by state index to approximate Q-values for unobserved states. I randomly shuffle the dataset for each iteration to improve the robustness of Q-Learning updates.

2. Performance Characteristics

2.1 Small Data Set

I ran the program for 100 epochs and it took 78 seconds for running.

2.2 Medium Data Set

I ran the program for 1000 epochs and it took 8 min 19 seconds for running.

2.3 Large Data Set

I ran the program for 100 epochs and it took 153 seconds for running.

3. Code

3.1 Q Learning

```
import numpy as np

class QLearning:
    def __init__(self, S, A, gamma, alpha, state, action):
        self.S = S
        self.A = A
        self.gamma = gamma
        self.alpha = alpha
        self.Q = np.zeros((state, action))

    def lookahead(self, s, a):
        return self.Q[s, a]

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

    def get_policy(self):
        return np.argmax(self.Q, axis=1) + 1
```

3.2 Small Data Set

```
import pandas as pd
from tqdm import trange
from QLearning import QLearning

def load_data():
    data = pd.read_csv("data/small.csv")
    S = list(data.s - 1)
    A = list(data.a - 1)
    R = list(data.r)
    SPR = list(data.sp - 1)
    return S, A, R, SPR, data.shape[0], data.shape[1] - 1

def main():
    k = 100
```

```
    gamma = 0.95
    state = 100
    action = 4
    S, A, R, SPR, n_state, n_action = load_data()

    alpha = 1 / k
    q_learning = QLearning(S, A, gamma, alpha, state, action)

    for _ in trange(k):
        for i in range(len(S)):
            q_learning.update(S[i], A[i], R[i], SPR[i])

    policy = q_learning.get_policy()
    with open("small.policy", 'w') as f:
        for p in policy:
            f.write(f"{p}\n")


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

3.3 Medium Data Set

```
import pandas as pd
from tqdm import trange
import bisect
from QLearning import QLearning

def load_data():
    data = pd.read_csv(f"data/medium.csv")
    S = list(data.s - 1)
    A = list(data.a - 1)
    R = list(data.r)
    SPR = list(data.sp - 1)
    return S, A, R, SPR, data.shape[0], data.shape[1] - 1


def find_missing_states(n_state, set_S):
    missing_states = {i for i in range(n_state) if i not in set_S}
    print(f"Number of missing states: {len(missing_states)}")
    return missing_states


def find_nearest_neighbors(missing_states, unique_S):
    nearest_neighbor = {}
    for i in missing_states:
        index_right = bisect.bisect_right(unique_S, i)
        index_left = index_right - 1
```

```
            if index_right == len(unique_S):
                min_index = index_left
            elif index_left == -1:
                min_index = index_right
            else:
                if unique_S[index_right] - i > i - unique_S[index_left]:
                    min_index = index_left
                else:
                    min_index = index_right

            nearest_neighbor[i] = unique_S[min_index]
        return nearest_neighbor


    def main():
        k = 1000
        gamma = 1.0
        S, A, R, SPR, n_state, n_action = load_data()
        set_S = set(S)
        state = 50000
        action = 7
        alpha = 0.05
        q_learning = QLearning(S, A, gamma, alpha, state, action)

        missing_states = find_missing_states(n_state, set_S)
        unique_S = sorted(set_S)
        nearest_neighbors = find_nearest_neighbors(missing_states, unique_S)

        for _ in trange(k):
            for i in range(len(S)):
                q_learning.update(S[i], A[i], R[i], SPR[i])

        policy = q_learning.get_policy()
        with open("medium.policy", 'w') as f:
            for i, p in enumerate(policy):
                if i in missing_states:
                    f.write(f"{policy[nearest_neighbors[i]]}\n")
                else:
                    f.write(f"{p}\n")


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

3.4 Large Data Set

```
import pandas as pd
from tqdm import trange
import bisect
```

```
from QLearning import QLearning


def load_data():
    data = pd.read_csv("data/large.csv")
    S = list(data.s - 1)
    A = list(data.a - 1)
    R = list(data.r)
    SPR = list(data.sp - 1)
    return S, A, R, SPR, data.shape[0], data.shape[1] - 1


def find_missing_states(n_state, set_S):
    missing_states = {i for i in range(n_state) if i not in set_S}
    print(f"Number of missing states: {len(missing_states)}")
    return missing_states


def find_nearest_neighbors(missing_states, unique_S):
    nearest_neighbor = {}
    for i in missing_states:
        index_right = bisect.bisect_right(unique_S, i)
        index_left = index_right - 1

        if index_right == len(unique_S):
            min_index = index_left
        elif index_left == -1:
            min_index = index_right
        else:
            if unique_S[index_right] - i > i - unique_S[index_left]:
                min_index = index_left
            else:
                min_index = index_right

        nearest_neighbor[i] = unique_S[min_index]
    return nearest_neighbor


def main():
    k = 100
    gamma = 0.95
    S, A, R, SPR, n_state, n_action = load_data()
    set_S = set(S)
    state = 302020
    action = 9

    alpha = 1 / k
    q_learning = QLearning(S, A, gamma, alpha, state, action)

    missing_states = find_missing_states(n_state, set_S)
```

```
    unique_S = sorted(set_S)
    nearest_neighbors = find_nearest_neighbors(missing_states, unique_S)

    for _ in trange(k):
        for i in range(len(S)):
            q_learning.update(S[i], A[i], R[i], SPR[i])

    policy = q_learning.get_policy()
    with open("large.policy", 'w') as f:
        for i, p in enumerate(policy):
            if i in missing_states:
                f.write(f"{policy[nearest_neighbors[i]]}\n")
            else:
                f.write(f"{p}\n")


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