Project 2: Reinforcement Learning

Erich Trieschman etriesch@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Descriptions

1.1 Small Data Set

[Q-learning; 51sec]

I observe that there are no missing state-action pairs in the small dataset so I do not expect that the selection of policy-learning approach matters much. For simplicity, I use Q-learning to choose the optimal policy.

I find that my policy converges after roughly 25 iterations and takes 51 seconds to run. I provide Table 1 to compare results across problems.

1.2 Medium Data Set

[Q-learning; 05:26]

For the Medium problem I again use Q-learning, but only to learn the optimal policies for the sampled states. For this component of my learning, I assume that all non-sampled states have a reward of zero.

Since the Mountain Car problem has a deterministic transition model, I decide to set the policy for each missing state to be the policy of the nearest known state. I define nearest using Euclidean distances and I do this considering the two substates, position and velocity. This step takes roughly 3 minutes for over 28,000 missing states.

I also considered a model-based approach for learning the transition model and reward function. Using the details provided in our class textbook, I model s'_{vel}, s'_{pos} as

$$s'_{vel} \sim \beta_v s_{vel} + \beta_a a + \beta_{pos} * \cos\left(\frac{\pi}{200} s_{pos}\right)$$
$$s'_{pos} \sim \beta_p s_p os + \beta_v s'_{vel}$$

I find strong agreement between my state predictions and sampled values, with $R^2 > 0.99$ in both models, yet for this model based approach I was not able to achieve a sufficiently high autograder score. I suspect that my model might be systematically biased since the predictor variables I use are categorical labels instead of the true action and state values.

I include a visual of my final policy using Q-learning and nearest-neighbor in Figure 1. I provide Table 1 to compare results across problems.

1.3 Large Data Set

[Policy iteration; 40:00]

For the Large problem I use policy iteration. I choose this approach mostly to get practice coding up different algorithms. Similar to the Medium problem, the Large problem has significant missing data. Here I choose to use policy iteration only to learn the policy for those states that have been sampled. This step takes roughly 33 minutes for 50 iterations.

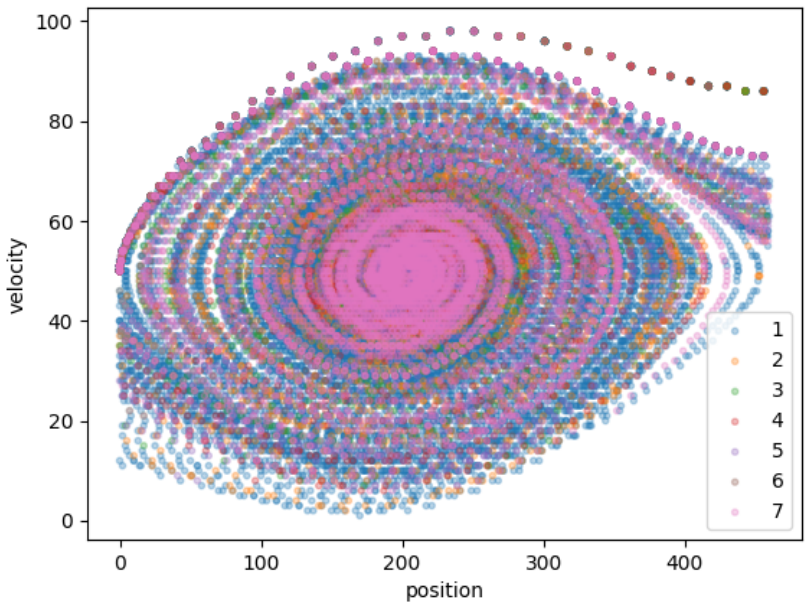

Figure 1: Medium problem - Policy by substates (position, velocity)

I observe that a significant share of missing states exist far away from the sampled states. For any missing state outside the range of sampled states (+/- a buffer), I decide to populate with a random policy. I use 1-nearest neighbor to populate policies for missing states when those states fell within the sampled range (+/- a buffer). With this process I populate all 311,000+ missing states with a policy in under seven minutes.

I understand that there is more structure in the large dataset than I am exploiting in this algorithm, so I expect there to be significant further gains beyond my nearest-neighbor interpolation approach. I provide Table 1 to compare results across problems.

Problem	**Learning algorithm**	**Interpolation time**	**Learning time**
Small	Q-learning	n/a	00:51
Medium	Q-learning	05:05	00:11
Large	Policy iteration	06:36	33:24

Table 1: Results from reinforcement learning across problems

2. Code

2.1 Policy iteration

```
def get_mle_problem(s, a, r, spr, n_state, n_act):
    # estimate transition probabilities and reward
    # get state/action counts
    N_sa = np.zeros((n_state, n_act), dtype=float)
    for i in range(len(s)):
        N_sa[s[i], a[i]] += 1.0

    print('building dictionary of sparse matrices...')
    T_asps = {}
    for ai in range(n_act):
        T_asps[ai] = sp.sparse.csr_matrix((n_state, n_state), dtype=float)
    R_sa = np.zeros((n_state, n_act), dtype=float)

    print('calculating MLE...')
    for i in tqdm(range(len(s))):
        T_asps[a[i]][spr[i], s[i]] += 1/N_sa[s[i], a[i]]
        R_sa[s[i], a[i]] += r[i]/N_sa[s[i], a[i]]

    return T_asps, R_sa, N_sa

def lookahead(U, gamma, s, a, r, T):
    for i in range(len(s)):
        U[s[i]] = r[i] + gamma * T[a[i]][:, s[i]].T @ U
    return U

def backup(U, gamma, s, n_state, n_act, R, T):
    Q_sa = np.zeros((n_state, n_act))
    for ai in range(n_act):
        Q_sa[:, ai] = lookahead(U, gamma, s,
                                np.repeat(ai, len(s)),
                                R[s, ai],
                                T)
    return Q_sa

def policy_evaluation(U, gamma, s, policy, R, T, k_max=100, tol=0.1):
    Up = U.copy()
    for k in range(k_max):
        Up = lookahead(Up, gamma, s, policy[s], R[s, policy[s]], T)
        max_diff = np.max(np.abs(Up - U))
        if max_diff < tol:
            break
        else:
            U = Up.copy()
    return U

def policy_improvement(U, gamma, s, n_state, n_act, R, T):
    return np.argmax(backup(U, gamma, s, n_state, n_act, R, T), axis=1)

def policy_iteration(gamma, s, R, T, n_state, n_act, k_max=5, tol=3):
```

```
    U = np.zeros(n_state)
    policy = np.random.randint(n_act, size=n_state)
    for k in tqdm(range(k_max)):
        U = policy_evaluation(U, gamma, s, policy, R, T, k_max=100)
        policyp = policy_improvement(U, gamma, s, n_state, n_act, R, T)
        max_diff = np.max(np.abs(policyp - policy))
        if max_diff < tol:
            break
        else:
            policy = policyp.copy()
    return policy, U
```

2.2 Q-learning

```
def Q_learn(gamma, s, a, r, spr, n_state, n_act, k_max=100, one_shot=False):
    Q = np.zeros((n_state, n_act))
    policy_log = np.zeros((k_max, n_state))
    for i in tqdm(range(k_max)):
        if one_shot:
            Q[s, a] += (1/k_max) * gamma * (r + np.max(Q[spr], axis=1) - Q[s,
 a])
        else:
            for j in range(len(s)):
                Q[s[j], a[j]] += (1/k_max) * gamma * (r[j] + np.max(Q[spr[j
]]) - Q[s[j], a[j]])
        policy_log[i] = np.argmax(Q, axis=1)

    return Q, policy_log
```

2.3 Nearest-neighbor

```
def nearest_neighbor(k, missing_states, s, policy, n_act, nn_thresh):
    for si in tqdm(missing_states):
        if (si >= s.min() - nn_thresh) & (si <= s.max() + nn_thresh):
            d = np.abs(si - s)
            dk_idx = np.argsort(d)[:k]
            dk = d[dk_idx]/d[dk_idx].sum()
            policy[si] = np.round(policy[s[dk_idx]] @ dk, 0)
        else:
        policy[si] = np.random.randint(n_act)
```

2.4 Other helper functions

```
def get_missing_states(n_state, s):
    possible_states = np.arange(n_state)
```

```
    sampled_states = np.where(np.in1d(possible_states, s))[0]
    missing_states = [i for i in range(n_state) if i not in sampled_states]
    return sampled_states, missing_states

def get_medium_substates(s):
    spos = np.mod(s, 500)
    svel = ((s - spos)/500).astype(int)
    return spos, svel

def get_medium_state(spos, svel):
    return (500*svel + spos).astype(int)
```