Project 2: Reinforcement Learning

Victor Sebastian Martínez Pérez sebasmp@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Descriptions

1.1 Small Data Set

The small dataset problem is a fully observed, discrete Markov Decision Process (MDP) with a small number of states and actions.

A model-based approach was used: the transition model $T(s' \mid s, a)$ and reward model $R(s, a)$ were estimated by maximum-likelihood estimation (MLE) from the data. Once the empirical model was constructed, value iteration was applied to compute the optimal value function $U^*(s)$ and greedy policy $\pi^*(s)$ for up to $10,000$ iterations.

The Bellman residual was computed to stop iterating once the residual $r < \varepsilon(1-\gamma)/\gamma$, with $\varepsilon = 0.001$ This approach guarantees convergence to the optimal policy within tolerance, since the environment is small and fully known after MLE. Running time was under 1 minute.

1.2 Medium Data Set

The medium dataset problem corresponds to a discretized environment grid of 500 × 100 (position × velocity = 50,000 states) dimensions. The data are limited and partially violate the Markov property, so a purely model-based method performs poorly.

A model-free Q-learning algorithm was implemented using the dataset as a fixed set of transitions (s,a,r,s'). For each tuple, the Q-value was updated following:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

with $\alpha = 1/N(s,a)$ decaying step size to ensure convergence. After convergence, a nearest-neighbor interpolation in the grid was used to fill unvisited states, and unseen actions were masked when selecting the greedy action. The algorithm included an early-stopping threshold on the maximum absolute Q-update ($|\Delta Q| < 10^{-5}$) to prevent unnecessary passes once convergence was reached. Running time was ∼5 minutes.

1.3 Large Data Set

The large task contains 302,020 states and 9 actions with $\gamma = 0.95$, a large discrete MDP with hidden internal structure. Because a full transition model would be infeasible, an offline Q-learning approach was again employed. The same decaying-α scheme was used, with same early-stopping criterion to detect convergence. For states never seen in the dataset, the policy defaults to the most-frequent action observed in the data. Running took $\sim$ 5 minutes.

2. Code

```python
# utils.py
import numpy as np

def load_dataset(csv_path):
    """Load (s,a,r,sp) with header"""
    with open(csv_path, "r") as fh:
        headers = fh.readline().strip().split(",")
    cols = [i for i, h in enumerate(headers) if h.startswith(("s", "a", "r",
    "sp"))]
    data = np.loadtxt(csv_path, delimiter=",", skiprows=1, usecols=cols)
    if data.ndim == 1:
        data = np.expand_dims(data, 0)
    return data

def write_policy_actions_only(actions_1based, path):
    """Write one integer action per line (no header)."""
    with open(path, "w") as f:
        for a in actions_1based:
            f.write(f"{int(a)}\n")

def write_results_table(state_ids, values, actions, path, header="state\
    tvalue*\taction*"):
    rows = np.column_stack([state_ids, values, actions])
    np.savetxt(path, rows, fmt=["%d", "%.8f", "%d"], header=header, comments=
    "")
```

```python
# helper.py
import numpy as np

from scipy.spatial import cKDTree

def lookahead(mdp, U):
    """One-step lookahead for the Bellman *optimality* backup."""
    return mdp.R + mdp.gamma * (mdp.T @ U)   # (S,A) + (S,A,S)@(S,) -> (S,A)

def bellman_backup(mdp, U, k_max, eps=1e-3):
    """Value-iteration style optimality backup with epsilon stopping rule."""
    for k in range(k_max):
        U_old = U.copy()
        Q = lookahead(mdp, U_old)
        U = Q.max(axis=1)
        residual = np.max(np.abs(U - U_old))
        delta = eps * (1.0 - mdp.gamma) / mdp.gamma
        if residual < delta:
            print(f"VI stop at iter {k} (residual {residual:.3e} < {delta:.3e}
    )")
            break
    return U
```

```
def value_iteration(mdp, k_max=10_000, eps=1e-3):
    U = np.zeros(len(mdp.S), dtype=float)
    U = bellman_backup(mdp, U, k_max, eps)
    Q = lookahead(mdp, U)
    a_idx = np.argmax(Q, axis=1)
    pi = {mdp.idx_to_state[i]: mdp.idx_to_action[int(a_idx[i])] for i in
    range(len(mdp.S))}
    return U, pi

def knn_fill_states(Q_full, S_seen, base=(500, 100), fallback_rows=None):
    """
    Fill rows of Q_full (states never seen) using nearest-neighbor in (pos,
    vel).
    - base = (n_pos_bins, n_vel_bins)
    - S_seen: raw ids of seen states (1-based)
    - fallback_rows: Q rows aligned with S_seen order (Q_seen)
    """
    nS_full = Q_full.shape[0]
    seen_set = set(int(s) for s in S_seen)

    pos_seen = ((S_seen - 1) % base[0]).astype(np.int64)
    vel_seen = ((S_seen - 1) // base[0]).astype(np.int64)
    tree = cKDTree(np.column_stack([pos_seen, vel_seen]))

    for sid in range(1, nS_full + 1):
        if sid not in seen_set:
            pos = (sid - 1) % base[0]
            vel = (sid - 1) // base[0]
            _, j = tree.query([pos, vel])  # index in S_seen / fallback_rows
            Q_full[sid - 1, :] = fallback_rows[j, :]
    return Q_full
```

```
# mdp.py
import numpy as np

class MDP:
    def __init__(self, gamma=0.95):
        self.gamma = gamma
        self.S = None; self.A = None
        self.T = None; self.R = None
        self.state_to_idx = None; self.idx_to_state = None
        self.action_to_idx = None; self.idx_to_action = None

    def mle(self, data: np.ndarray):
        """Maximum-likelihood model from (s,a,r,sp) tuples."""
        self.S = np.unique(np.concatenate([data[:, 0], data[:, 3]])).astype(
    np.int64)
        self.A = np.unique(data[:, 1]).astype(np.int64)
```

```
        self.state_to_idx = {int(s): i for i, s in enumerate(self.S)}
        self.idx_to_state = {i: s for s, i in self.state_to_idx.items()}
        self.action_to_idx = {int(a): i for i, a in enumerate(self.A)}
        self.idx_to_action = {i: a for a, i in self.action_to_idx.items()}

        S_len, A_len = len(self.S), len(self.A)
        N_sas = np.zeros((S_len, A_len), dtype=np.float64)
        N_sasp = np.zeros((S_len, A_len, S_len), dtype=np.float64)
        R_sum = np.zeros((S_len, A_len), dtype=np.float64)

        for s, a, r, sp in data:
            si = self.state_to_idx[int(s)]
            ai = self.action_to_idx[int(a)]
            spi = self.state_to_idx[int(sp)]
            N_sasp[si, ai, spi] += 1.0
            N_sas[si, ai] += 1.0
            R_sum[si, ai] += float(r)

        self.T = N_sasp / np.maximum(N_sas[:, :, None], 1.0)
        self.R = R_sum / np.maximum(N_sas, 1.0)
```

```
# qlearning.py
import numpy as np
import random

class QLearning:
    """Offline Q-learning"""
    def __init__(self, S_raw, A_ids):
        self.S_raw = np.asarray(S_raw, dtype=np.int64)
        self.A_ids = np.asarray(A_ids, dtype=np.int64)
        self.nS = len(self.S_raw); self.nA = len(self.A_ids)
        self.state_to_idx = {int(s): i for i, s in enumerate(self.S_raw)}
        self.Q = np.zeros((self.nS, self.nA), dtype=float)
        self.N = np.zeros((self.nS, self.nA), dtype=np.int64)

    def simulate(self, data, n_passes=200, gamma=1.0, alpha=None, tol=1e-5,
    shuffle_each=True, terminal_states=None):
        if terminal_states is None: terminal_states = set()
        data = list(map(lambda t: (int(t[0]), int(t[1]), float(t[2]), int(t
    [3])), data))

        for _ in range(n_passes):
            if shuffle_each: random.shuffle(data)
            max_abs_delta = 0.0

            for (s, a, r, sp) in data:
                i = self.state_to_idx[s]
                j = self.state_to_idx.get(sp, None)
```

```
                if (sp in terminal_states) or (j is None):
                    max_next = 0.0
                else:
                    max_next = float(np.max(self.Q[j, :]))

                target = r + gamma * max_next
                delta = target - self.Q[i, a]

                if alpha is None:
                    self.N[i, a] += 1
                    step = 1.0 / self.N[i, a]
                else:
                    step = alpha

                self.Q[i, a] += step * delta
                if abs(delta) > max_abs_delta: max_abs_delta = abs(delta)

            if max_abs_delta < tol: break

        return self.Q
```

```
# project2.py
#!/usr/bin/env python3
import argparse
import numpy as np

from mdp import MDP
from helper import value_iteration, knn_fill_states
from qlearning import QLearning
from utils import (
    load_dataset,
    write_policy_actions_only,
    write_results_table,
)

# Small: Model-based MLE + Value Iteration
def run_small(data_path="datasets/small.csv"):
    data = load_dataset(data_path)

    mdp = MDP(gamma=0.95)
    mdp.mle(data)

    # each (s,a) row in T should sum to 1 if seen
    # row_sums = mdp.T.sum(axis=2)
    # seen = row_sums > 0
    # if np.any(seen):
    #     print("T row-sums (seen): min=", row_sums[seen].min(), "max=",
    row_sums[seen].max())
```

```
    U, pi = value_iteration(mdp, k_max=10_000)

    # small: write both policy and diagnostics
    actions = np.array([pi[mdp.idx_to_state[i]] for i in range(len(mdp.S))],
    dtype=int)
    write_policy_actions_only(actions, f"output/small.policy")
    write_results_table(
        state_ids=mdp.S.astype(int),
        values=U,
        actions=actions,
        path=f"output/results_small.txt",
        header="state\tvalue*\taction*",
    )
    print("small: policy and results saved")

# Medium: Q-learning + K-nearest neighbors
def run_medium(data_path="datasets/medium.csv"):
    data = load_dataset(data_path).astype(float)

    # actions: from 1..7 to 0..6 for indexing
    data[:, 1] -= 1
    data = data.astype(np.int64)

    S_raw = np.unique(np.concatenate([data[:, 0], data[:, 3]])).astype(np.
    int64)
    A = np.arange(7, dtype=np.int64)  # 0..6

    agent = QLearning(S_raw, A)
    Q_seen = agent.simulate(data, n_passes=300, gamma=1.0, alpha=None, tol=1e
    -5)

    # Build full 50k7 Q and copy observed rows
    nS_full, nA = 50000, 7
    Q_full = np.zeros((nS_full, nA), dtype=float)
    N_full = np.zeros((nS_full, nA), dtype=np.int64)

    sid2row = agent.state_to_idx
    for s in S_raw:
        i_full, i_seen = int(s) - 1, sid2row[int(s)]
        Q_full[i_full, :] = Q_seen[i_seen, :]
        N_full[i_full, :] = agent.N[i_seen, :]

    # Fill unseen states with k-nearest neighbors in (pos,vel) grid
    Q_full = knn_fill_states(Q_full, S_raw, base=(500, 100), fallback_rows=
    Q_seen)

    # Greedy policy
    pi0 = np.empty(nS_full, dtype=np.int64)
    seen_set = set(int(s) for s in S_raw)
    for i in range(nS_full):
```

```
            if (i + 1) in seen_set:
                q = Q_full[i, :].copy()
                mask = (N_full[i, :] == 0)
                if mask.all():
                    pi0[i] = int(np.argmax(q))
                else:
                    q[mask] = -1e12
                    pi0[i] = int(np.argmax(q))
            else:
                pi0[i] = int(np.argmax(Q_full[i, :]))  # NN-filled row

        # back to 1..7 for export
        pi_out = (pi0 + 1).astype(np.int64)
        write_policy_actions_only(pi_out, f"output/medium.policy")

        # diagnostics (value* = max_a Q)
        U_star = np.max(Q_full, axis=1)
        write_results_table(
            state_ids=np.arange(1, nS_full + 1),
            values=U_star,
            actions=pi_out,
            path=f"output/results_medium.txt",
            header="state\tvalue*\taction*",
        )
        print("medium: policy and results saved")

    # Large: Q-learning
    def run_large(data_path="datasets/large.csv",
                  n_states_full=302020, n_actions_full=9, gamma=0.95):
        data = load_dataset(data_path)

        # Normalize actions to 0..(n_actions_full-1) if theyre 1..n
        a_min, a_max = int(data[:, 1].min()), int(data[:, 1].max())
        if a_min == 1 and a_max == n_actions_full:
            data[:, 1] -= 1
        data = data.astype(np.int64)

        S_raw = np.unique(np.concatenate([data[:, 0], data[:, 3]])).astype(np.
        int64)
        A = np.arange(n_actions_full, dtype=np.int64)  # 0..8

        agent = QLearning(S_raw, A)
        Q_seen = agent.simulate(data, n_passes=60, gamma=gamma, alpha=None, tol=1
        e-5)

        # Full table & copy
        Q_full = np.zeros((n_states_full, n_actions_full), dtype=float)
        sid2row = agent.state_to_idx
        for s in S_raw:
            Q_full[int(s) - 1, :] = Q_seen[sid2row[int(s)], :]
```

```
    # Greedy policy with masking of never-tried actions on seen states
    pi0 = np.empty(n_states_full, dtype=np.int64)
    seen = set(int(s) for s in S_raw)
    for i in range(n_states_full):
        if (i + 1) in seen:
            counts = agent.N[sid2row[i + 1], :]
            q = Q_full[i, :].copy()
            if counts.sum() > 0:
                q[counts == 0] = -1e12
            pi0[i] = int(np.argmax(q))
        else:
            # unseen state -> safest fallback = most frequent action in data
            most_common = int(np.bincount(data[:, 1], minlength=
n_actions_full).argmax())
            pi0[i] = most_common

    pi_out = (pi0 + 1).astype(np.int64)
    write_policy_actions_only(pi_out, f"output/large.policy")

    # optional diagnostics
    U_star = np.max(Q_full, axis=1)
    write_results_table(
        state_ids=np.arange(1, n_states_full + 1),
        values=U_star,
        actions=pi_out,
        path=f"output/results_large.txt",
        header="state\tvalue*\taction*",
    )
    print("large: policy and results saved")

def main():
    run_small("datasets/small.csv")
    run_medium("datasets/medium.csv")
    run_large("datasets/large.csv")

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