Project 2: Reinforcement Learning

Matthias Kura kura@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Descriptions

1.1 Small Data Set

For the small dataset, Script 1 shows the implementation of a classical **value iteration approach** (Kochenderfer et al. (2022)). In `build_model()`, the averaged transition and reward matrices $\mathcal{T}$ and $\mathcal{R}$ are constructed from the given dataset. Then, the `value_iteration()` function performs 10000 value iterations with a discount factor of $\gamma = 0.95$ to maximize the utilities U and obtain the corresponding actions to fill the policy π.

For the "small.csv" dataset, the algorithm built the model in 0.51 s and completed value iteration below 0.01 s. The scoring metrics for the small policy are given in Table 1.

1.2 Medium Data Set

Similar to the small dataset, the policy for the medium dataset on the mountain car problem is produced with **value iteration** (Kochenderfer et al. (2022)). Script 2 shows the implementation. The `build_model()` function builds the reward and transition matrices $\mathcal{R}$ and $\mathcal{T}$ of observed states in the dataset. The algorithm greedily takes the action a that leads to the highest expected utility. As some possible states are not present in the dataset, the `save_policy()` function assigns a **fallback default action** ($= 4$) to fill the rest of the policy.

Overall, this leads to a model build time of 0.07 s, the completion of 500 value iterations in 7.9 s, and the extraction of the policy in 0.03 s for the dataset provided in "medium.csv". The scoring metrics for the medium policy are given in Table 1.

1.3 Large Data Set

For the large dataset, several learning approaches were explored. Value iteration of the samples in the dataset did not yield results better than a random policy. A pure model-free Q-learning approach did not improve the score beyond the autograder baseline. In the end, the best initial approach is obtained by using **fitted Q-iteration** (Ernst et al. (2005)). The implementation is shown in the `fitted_q_iteration()` function in Script 3. Here, the action value function Q is directly updated using the samples in the dataset instead of evaluating known transitions $\mathcal{T}$ from a built model. The loop is stopped after a given number of iterations or when the mean change across Q is smaller than a tolerance.

As the large dataset did not provide samples for all 302020 states, fallback actions for unobserved states need to be assigned. A random or deterministic policy for the the fallback actions did not lead to scores exceeding the baseline or even random policies. Investigating the dataset more closely revealed a structure of the state and transition spaces $\mathcal{S}$ and $\mathcal{T}$. The state is composed of three two-digit combinations [C1][C2][C3], where C1 $\in$ [15, 23, 27, 29, 30] and C1, C2 $\in$ [01, 02, 03, 04, 10, 11, 12, 13, 14, 20]. The possible rewards $\mathcal{R} \in$ [-10, -5, 0, 5, 10, 50, 100] are only attainable with actions 1 to 4. There are no rewards for actions 5 to 9, which lead to the state not changing. Actions 1-4 can lead to states in the immediate neighborhood of the [C1][C2] (e.g. [04] to [03] or [10]). When [C1][C2] are either [01][01] or [20][20], actions 5-9 can lead to random states. Based on this knowledge, it is decided to use a a **nearest-neighborhood approach to fill the remaining policy**. The `build_policy()` function shows this approach, where the policy for an unobserved state is assigned the action of the closest observed state to the "left" or "right". As a side note, a true model-based Q-learning with the nearest neighborhood approach would have also passed the baseline, albeit with a lower score.

Overall, this leads to a model build time of 0.03 s, 8.68 s for 1000 fitted Q-iterations, and a policy build time of 6.61 s. The scoring metrics for the final large policy are given in Table 1.

2. Scores

Table 1 shows the scores of the three different generated policies for their raw score, score against a random policy (raw - random), and the resulting leaderboard score.

Table 1: Policy Scores (rounded to two decimals)

Metric	Small	Medium	Large
Raw	35.05	77.28	550.14
vs. random	33.74	181.30	549.98
Leaderboard	33.74	181.30	5499.80

References

Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6(18):503–556, 2005. URL `http://jmlr.org/papers/v6/ernst05a.html`.

Mykel J. Kochenderfer, Tim A. Wheeler, and Kyle H. Wray. *Algorithms for Decision Making*. MIT Press, 2022.

Code

Listing 1: Code for Small Dataset

```
import pandas as pd
import numpy as np

def build_model(df, n_states, n_actions):
    # Build transition and reward matrices from data frame (for small dataset
    )
    counts = np.zeros((n_states, n_actions, n_states), dtype=int)   # keep
    track of counts of each s->a->sp transition
    sum_rewards = np.zeros((n_states, n_actions), dtype=float)  # collect
    rewards of state and performed action
    n_sa = np.zeros((n_states, n_actions), dtype=int)   # collect number of
    visits

    for _, row in df.iterrows():
        # extract state, action, and next state
        s = int(row['s']) - 1
        a = int(row['a']) - 1
        r = float(row['r'])
        sp = int(row['sp']) - 1

        # increase counts, rewards and visits
        counts[s,a,sp] += 1
        sum_rewards[s,a] += r
        n_sa[s,a] += 1

    # assemble matrices
    T = counts / counts.sum(axis = 2, keepdims=True)
    R = np.divide(sum_rewards, n_sa, out = np.zeros_like(sum_rewards), where=
    n_sa>0)    # average return of s,a combination

    return T,R

def value_iteration(P,R, gamma = 0.95, tol=1e-6, max_iter=10000):
    # Performs value iteration to compute optimal value function and policy
    # get shape of P
    n_states, n_actions, _ = np.shape(P)
    # initialize U
    U = np.zeros(n_states, dtype=float)

    for it in range(max_iter):
        Q = R + gamma * (P @ U) # lookahead equation
        U_new = np.max(Q, axis=1)   # value iteration maximizes U
        if np.max(np.abs(U_new - U)) < tol:
            break
        U = U_new
```

```
    policy = np.argmax(R + gamma * (P @ U), axis = 1)   # extract policy

    return U, policy

# START
df = pd.read_csv("data/small.csv")
T,R = build_model(df, n_states=100, n_actions=4)
U, policy = value_iteration(T,R,gamma = 0.95)
# save policy to file
np.savetxt("small.policy", policy + 1, fmt='%d')
```

Listing 2: Code for Medium Dataset

```
import pandas as pd
import numpy as np

# Parameters
GAMMA = 0.99
MAX_ITER = 500
TOL = 1e-4
N_ACTIONS = 7
N_POS, N_VEL = 500, 100
N_STATES = N_POS * N_VEL

def build_model(df):
    # Build transition and reward matrices from data frame

    # Rewards
    grouped_r = df.groupby(['s','a'])['r'].mean().reset_index() # group df by
     state and action
    R = {(int(row.s), int(row.a)): float(row.r) for row in grouped_r.
    itertuples(index = False)} # create dictionary

    # Transitions
    counts = df.groupby(['s','a','sp']).size().rename('count').reset_index()
     # Count combination of s->a->sp
    total_counts = counts.groupby(['s', 'a'])['count'].transform('sum') #get
    counts['prob'] = counts['count'] / total_counts

    # build dictionary
    T = {}
    for row in counts.itertuples(index = False):
        key = (int(row.s), int(row.a))  # exctract key of dictionary
        if key not in T:
            T[key] = []
        T[key].append((int(row.sp), float(row.prob)))   # build dictionary

    return R, T

def value_iteration(R, T, gamma = GAMMA, max_iter = MAX_ITER, tol = TOL):
    # performs value_iteration on expected reward dictionary R and transition
     model dictionary T
    # perform only for observed states
    observed_states = sorted(set(s for (s,_) in R.keys()))
    U = {s: 0.0 for s in observed_states}


    # run value iteration
    for it in range(max_iter):
        delta = 0.0
        newU = {}
```

```
        # update value for each state
        for s in observed_states:
            q_values = []
            # Evaluate all actions (uincluding ones not observed)
            for a in range(N_ACTIONS):
                sa = (s,a)
                r = R.get(sa, 0.0)
                if sa in T:
                    # if state can be reached, compute expected utility of
    next state
                    exp_u = sum(prob * U.get(sp, 0.0) for sp,prob in T[s,a])
                else:
                    exp_u = 0.0
                q = r + gamma * exp_u   # lookahead equation
                q_values.append(q)

            # greedy: pick best expected value
            newU[s] = max(q_values)
            delta = max(delta, abs(newU[s] - U[s]))

        # Update
        U = newU

        # break if < tol
        if delta < tol:
            break

    return U

def extract_policy(U, R, T):
    # extracts the policy given dictionaries of values U, rewards R, and
    transition probabilities T
    policy = {}
    observed_states = sorted(set(s for (s,_) in R.keys()))

    for s in observed_states:
        best_a, best_q = None, -np.Inf

        # step through all actions and take greedy action
        for a in range(N_ACTIONS):
            sa = (s,a)
            r = R.get(sa, 0.0)
            if sa in T:
                exp_u = sum(prob * U.get(sp,0.0) for sp, prob in T[sa])
            else:
                exp_u = 0.0
            q = r + GAMMA * exp_u   # lookahead equation
            if q > best_q:
                best_q, best_a = q, a
```

```
        policy[s] = best_a + 1  # shift actions to 1-7

    return policy

def save_policy(policy, filename):
    # saves policy dictionary to file
    with open(filename, 'w') as f:
        for s in range(N_STATES):
            a = policy.get(s, 0)
            if a == 0:
                a = 4  # default action 4 if state not observed
            f.write(f"{a}\n")

# START
df = pd.read_csv("data/medium.csv")
R,T = build_model(df)
U = value_iteration(R,T)
policy = extract_policy(U,R,T)
save_policy(policy, "medium.policy")
```

Listing 3: Code for Large Dataset

```
import numpy as np
import pandas as pd
from collections import defaultdict
import time
import bisect

# Hyperparameters
N_STATES = 302020
N_ACTIONS = 9
GAMMA = 0.95 # discount factor
ALPHA = 0.2 # learning rate for Q-learning
MAX_ITER = 1000
TOL = 1e-4
DEFAULT = 1

def build_model(df):
    T = defaultdict(lambda: defaultdict(list))
    for row in df.itertuples(index=False):
        T[row.s][row.a].append((row.r, row.sp))
    return T

def initialize_Q(T):
    Q = defaultdict(lambda: defaultdict(float))
    for s, acts in T.items():
        for a in acts:
            Q[s][a] = 0.0
    return Q

def fitted_q_iteration(T,Q, gamma = 0.95, max_iter = 200, tol = 1e-4):
    for it in range(1, max_iter+1):
        start = time.time()

        Q_new = defaultdict(lambda: defaultdict(float))
        delta = 0.0
        count = 0

        for s, acts in T.items():
            for a, samples in acts.items():
                targets = []
                for r, sp in samples:
                    if sp in Q:
                        max_next = max(Q[sp].values()) if Q[sp] else 0.0
                    else:
                        max_next = 0.0
                    targets.append(r + gamma * max_next)    # loookahead
    equation
                new_q = np.mean(targets) if targets else 0.0
                old_q = Q[s][a]
```

```
                Q_new[s][a] = new_q # update of Q
                delta += abs(new_q - old_q)
                count += 1

        mean_change = delta / max(count,1)
        Q = Q_new

        elapsed = time.time() - start
        print(f"Iter {it:3d} | mean Q={mean_change:.6f} | time={elapsed:.2f}s"
    )
        if mean_change < tol:
            break
    return Q

def q_learning(T,Q, gamma = 0.95, alpha = 1.0, max_iter = 200, tol = 1e-4):
    np.random.seed(42)  # for reproducibility
    # gather all samples
    all_samples = []
    for s, acts in T.items():
        for a, samples in acts.items():
            for r, sp in samples:
                all_samples.append((s, a, r, sp))

    for it in range(1, max_iter+1):
        np.random.shuffle(all_samples)
        delta = 0.0

        for s, a, r, sp in all_samples:
            if sp in Q:
                max_next = max(Q[sp].values())
            else:
                max_next = 0.0
            old_q = Q[s][a]
            Q[s][a] += alpha * (r + gamma * max_next - Q[s][a])   # Q-
    learning update
            delta += abs(Q[s][a] - old_q)

        mean_change = delta / len(all_samples)
        print(f"Iter {it:3d} | mean Q={mean_change:.6f}")
        if mean_change < tol:
            break
    return Q

def build_policy(Q, n_states = 302020, n_actions = 9, default_action = -1):
    # initialize policy vector
    policy = [-1] * (n_states + 1)

    # for each observed state, fill policy with best action
    observed_actions = {}
    for s, acts in Q.items():
```

```
        best_a = 0
        best_q = -float("inf")
        for a in range(1,n_actions+1):
            q = acts.get(a, -float("inf"))
            if q > best_q:
                best_q = q
                best_a = a
        if isinstance(s, int) and 1 <= s < n_states+1:
            observed_actions[s] = best_a

    # Fill policy for observed states
    for s, a in observed_actions.items():
        policy[s] = a

    policy = policy[1:]  # adjust for 1-based indexing

    # Nearest neighbor filling
    # For every unobserved state, find neaerest observed state by index
    for s in range(n_states):
        if policy[s] == -1 and s+1 not in observed_actions:
            # find insertion point
            pos = bisect.bisect_left(sorted(observed_actions.keys()), s+1)

            # Search outward from the insertion point for the nearest
observed state
            left_idx = pos - 1
            right_idx = pos
            chosen = None
            observed_states = sorted(observed_actions.keys())
            while left_idx >= 0 or right_idx < len(observed_states):
                left_state = observed_states[left_idx] if left_idx >= 0 else
None
                right_state = observed_states[right_idx] if right_idx < len(
observed_states) else None

                # compute distances, prefer smaller distance
                if left_state is not None and right_state is not None:
                    d_left = (s+1) - left_state
                    d_right = right_state - (s+1)
                    if d_left <= d_right:
                        cand_state = left_state
                        left_idx -= 1
                    else:
                        cand_state = right_state
                        right_idx += 1
                elif left_state is not None:
                    cand_state = left_state
                    left_idx -= 1
                elif right_state is not None:
                    cand_state = right_state
```

```
                    right_idx += 1
                else:
                    break

                chosen = cand_state
                break

            if chosen is not None:
                policy[s] = observed_actions[chosen]
            else:
                policy[s] = default_action

    return policy

# START
df = pd.read_csv("data/large.csv")
T = build_model(df)
Q = initialize_Q(T)
Q = fitted_q_iteration(T,Q,gamma=GAMMA, max_iter=MAX_ITER, tol=TOL)
policy = build_policy(Q, n_states=N_STATES, n_actions=N_ACTIONS,
    default_action=DEFAULT)

with open("large.policy", "w") as f:
    for a in policy:
        f.write(f"{a}\n")
```