Project 2: Reinforcement Learning

Josselin Somerville Roberts josselin@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Descriptions

1.1 Small Data Set

For the small dataset, I tried Q-learning, value iteration, and policy iteration. As policy iteration and value iteration both converged, I have decided to take this (it's what yielded the best result). They both converged to the same result.

- **Runtime:** about 30 seconds
- **Score on the leaderboard:** 29.75

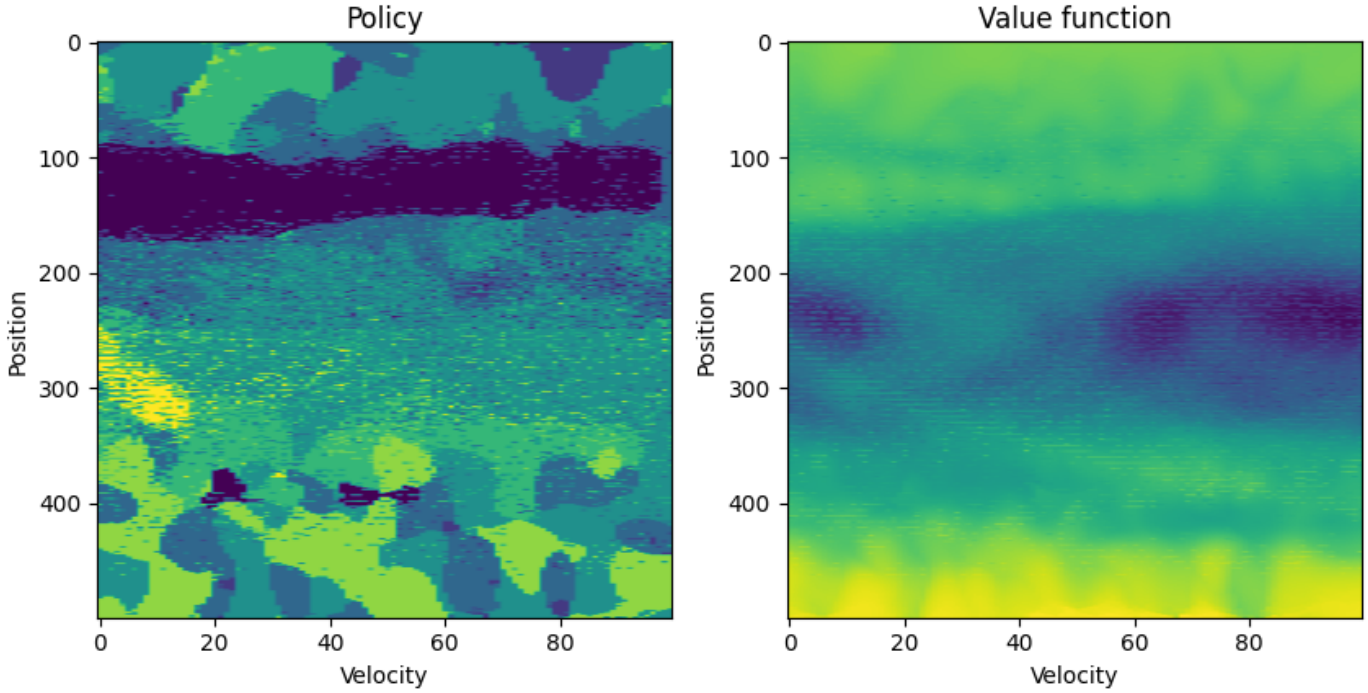
1.2 Medium Data Set

For this problem, I used Q-learning, with kernel smoothing using a gaussian kernel. This was particularly important as a lot of states were missing. Technically I implemented a variation of Q-learning described as such:

1. Sample a state s in the dataset
2. Repeat until not done, or reach max number of iterations:
   (a) Sample x a random uniform variable on $[0, 1]$.
   (b) If $x < \epsilon$, choose a a random action such that (s, a) is in the data.
   (c) Otherwise, choose a that maximizes $Q[s, a] + mask(s, a)$ with $mask(s, a)$ equal to 0 if (s, a) is in the data, otherwise $-\infty$.
   (d) From the data, sample (s', r) given (s, a)
   (e) If s' is in the data $Q[s, a] = Q[s, a] + \alpha\left(r + \gamma\left(\max_{a' \in \mathcal{A}}\left(Q[s', a'] + mask[s', a']\right)\right) - Q[s, a]\right)$
   (f) Otherwise set done to True and $Q[s, a] = Q[s, a] + \alpha\left(r - Q[s, a]\right)$.
3. Repeat Step 1 and 2 for as many iterations as specified
4. Use kernel smoothing to compute Q-values of pairs (s, a) that have $mask[s, a] = -\infty$.

To make the kernel smoothing, I implemented a KernelSmoother class that uses a Kd-tree and only smoothed on a certain radius. This performed very well.

- **Runtime:** about 45 minutes
- **Score on the leaderboard:** 198.41



Policy and the associated value function for the medium dataset

1.3 Large Data Set

For this problem I tried to find a structure in the large dataset but to be fully honest I did not really succeed. Here is what I found:

- Only certain states yield rewards. These states give the same reward for actions 1,2,3 and 4 and not a reward for the other actions. Also for these 4 actions, $s' - s$ is a constant depending on s.
- Most transitions can be explained. Even though the next state s' is a random function of s and a, the most likely can be predicted (I was able to predict 91% of the states s' with my method, with an accuracy of 100% for the most likely outcome).

Here is the model I found empirically:

- If $a \geq 5$, then $s' = s$.
- If $a = 1$, then
  - If the two last digits are 2 and 0, then $s' = s$.
  - if the two last digits are 1 and 0, then $s' = s + 1$.
  - If the last digit is 4, then $s' = s + 6$.

  - If the last digit is 0 then $s' = s + 4$.
  - Otherwise, $s' = s + 1$.

- If $a = 2$, then
  - If the two last digits are 1 and 1, then $s' = s - 1$.
  - If the last digit is 0, then $s' = s - 6$.
  - If the last digit is 1 then $s' = s$.
  - Otherwise, $s' = s - 1$.

- If $a = 3$, then
  - If the fourth and third last digits are 0 and 1, then $s' = s$.
  - If the third last digit is 0, then $s' = s - 600$.
  - Otherwise, $s' = s - 100$.

- If $a = 4$, then
  - If the fourth and third last digits are 2 and 0, then $s' = s$.
  - If the third last digit is 4, then $s' = s + 600$.
  - Otherwise, $s' = s + 100$.

(I also tried to create a neural network to predict the reward and next state given the state and action but the performance was disappointing even though I used a digit representation as I knew this was more useful than giving the state number)

However, I did not find a successful way to leverage these heuristics. I tried Q-learning with kernel smoothing as on the medium dataset. I also tried vanilla Q-learning, sampling (s', r) when (s, a), was in the data and predicting it using the heuristics when (s, a) was not in the data. The second option yielded slightly better results, but they were still bad. I would be interested to get some feedback on how I could have better leveraged the data analysis I did.

- **Runtime:** about 1 hour
- **Score on the leaderboard:** 3192.58

2. Code

```
import numpy as np
from tqdm import tqdm
from utils import sample_action


# Value iteration algorithm
def value_iteration(data_dict, num_states, gamma=0.95, max_num_iter=10000,
    delta_stop=0.01):
    V = np.zeros(num_states)
    for _ in tqdm(range(max_num_iter)):
        delta = 0
        old_V = V.copy()
        for idx_s in range(num_states):
            s = idx_s + 1
            Q = np.zeros(4)
            for a in data_dict[s]:
                n_total = 0
                for (sp, r) in data_dict[s][a].keys():
                    n = data_dict[s][a][(sp, r)]
                    n_total += n
                    Q[a - 1] += n * (r + gamma * old_V[sp - 1])
                Q[a - 1] /= n_total
            V[idx_s] = np.max(Q)
            delta = max(delta, abs(old_V[idx_s] - V[idx_s]))
        if delta < delta_stop:
            break
    return V

def get_policy_from_V(data_dict, V, gamma=0.95):
    policy = np.zeros(len(V), dtype=int)
    for idx_s in range(len(V)):
        s = idx_s + 1
        Q = np.zeros(4)
        for a in data_dict[s]:
            n_total = 0
            for (sp, r) in data_dict[s][a].keys():
                n = data_dict[s][a][(sp, r)]
                n_total += n
                Q[a - 1] += n * (r + gamma * V[sp - 1])
            Q[a - 1] /= n_total
        policy[idx_s] = np.argmax(Q) + 1
    return policy


def policy_iteration(data_dict, num_states, num_actions, gamma=0.95,
    max_num_iter=10000, max_num_iter_policy=50, delta_stop=0.01):
    V = np.zeros(num_states)
```

```
    policy = [sample_action(data_dict, idx_s + 1) for idx_s in range(
    num_states)]

    for _ in range(max_num_iter_policy):
        # Policy evaluation
        V = np.zeros(num_states)
        for _ in tqdm(range(max_num_iter)):
            delta = 0
            old_V = V.copy()
            V = np.zeros(num_states)
            for idx_s in range(num_states):
                s = idx_s + 1
                a = policy[idx_s]
                n_total = 0
                for (sp, r) in data_dict[s][a].keys():
                    n = data_dict[s][a][(sp, r)]
                    n_total += n
                    V[idx_s] += n * (r + gamma * old_V[sp - 1])
                V[idx_s] /= n_total
                delta = max(delta, abs(old_V[idx_s] - V[idx_s]))
            if delta < delta_stop:
                break

        # Policy improvement
        stable = True
        for idx_s in range(num_states):
            s = idx_s + 1
            Q = np.zeros(num_actions)
            for a in data_dict[s]:
                n_total = 0
                for (sp, r) in data_dict[s][a].keys():
                    n = data_dict[s][a][(sp, r)]
                    n_total += n
                    Q[a - 1] += n * (r + gamma * V[sp - 1])
                Q[a - 1] /= n_total
            old_pi_s = policy[idx_s]
            policy[idx_s] = np.argmax(Q) + 1
            if old_pi_s != policy[idx_s]:
                stable = False

        if stable:
            break

    return V, policy


if __name__ == "__main__":
    from utils import get_data_dict, save_policy
    size = "small"
```

```
    data_dict = get_data_dict(size)

    # Value iteration
    print("Value iteration...")
    V1 = value_iteration(data_dict, 100, gamma=0.95, max_num_iter=1000,
    delta_stop=1e-10)
    policy1 = get_policy_from_V(data_dict, V1, gamma=0.95)
    save_policy(policy1, size + "_value_iter")
    print("Value iteration done")
    print("V1:", V1)

    # Policy iteration
    print("\n\nPolicy iteration...")
    V2, policy2 = policy_iteration(data_dict, 100, 4, gamma=0.95,
    max_num_iter=1000, max_num_iter_policy=50, delta_stop=1e-10)
    save_policy(policy1, size + "_policy_iter")
    print("Policy iteration done")
    print("V2:", V2)

    # Check if the two policies are the same
    print("Policies are the same:", np.all(policy1 == policy2))
```

```
# This file applies Q-learning to the data provided in data/
# under the format s,a,r,s' where s is the state, a is the action,
# r is the reward, and s' is the next state. The data is provided
# in the form of a csv file. The Q-learning algorithm is applied
# to the data and the resulting Q-table is saved in the form of
# a csv file.
# It also generates a plot of the Q-table for each state.
# And computes the optimal policy for each state, saves it in a
# csv file, and generates a plot of the policy for each state.

import numpy as np
from tqdm import tqdm
from utils import sample_outcome, sample_action, get_data_dict, get_data,
    save_policy
from scipy.spatial import cKDTree

# Suppress/hide the warning
np.seterr(invalid='ignore')

class KernelSmoothing:
    def __init__(self, bandwidth):
        self.bandwidth = bandwidth

    def fit(self, X, Y):
        self.X = X
        self.Y = Y
        self.kd_tree = cKDTree(X)

    def predict(self, x, factor=3.0, count=0):
        indices = self.kd_tree.query_ball_point(x, r=factor*self.bandwidth)
        X_neighbors = self.X[indices]
        Y_neighbors = self.Y[indices]
        weights = self.gaussian_kernel(x, X_neighbors, self.bandwidth)
        y_pred = np.sum(weights * Y_neighbors) / np.sum(weights)
        if np.isnan(y_pred):
            if count > 10:
                print("Error: Kernel smoothing failed to converge.")
                return 1
            return self.predict(x, factor=factor*2, count=count+1)
        return y_pred

    def gaussian_kernel(self, x, X, bandwidth):
        diff = np.linalg.norm(X - x, axis=1)
        return np.exp(-0.5 * ((diff / bandwidth) ** 2)) / (np.sqrt(2 * np.pi)
     * bandwidth)


data = get_data("medium")
data_dict = get_data_dict("medium")
```

```
# Sorts the set of states and actiosn and create 4 dictionaries
# that map each state to an index and vice versa
# and each action to an index and vice versa
set_of_states = set(data['s'])
set_of_actions = set(data['a'])
sorted_set_of_states = sorted(set_of_states)
sorted_set_of_actions = sorted(set_of_actions)
action_to_index = {}
index_to_action = {}
for i in range(len(sorted_set_of_actions)):
    action = sorted_set_of_actions[i]
    action_to_index[action] = i
    index_to_action[i] = action



# Initialize Q-table
Q = np.zeros((50000, 7))
mask = -np.inf * np.ones((50000, 7))
for i in tqdm(range(data.shape[0]), desc="Generating mask"):
    # Get state, action, next state, and reward
    s = data.iloc[i, 0]
    a = data.iloc[i, 1]
    idx_s = s - 1
    idx_a = action_to_index[a]
    mask[idx_s, idx_a] = 0
# Prints dimensions of Q-table
print("Shape of Q-table: " + str(Q.shape))

# Kernel smoothing
def state_to_pos_vel(s):
    pos = (s - 1) // 100
    vel = (s - 1) % 100
    return [pos, vel]
def state_action_to_pos_vel(s, a):
    pos = (s - 1) // 100
    vel = (s - 1) % 100
    return [pos, vel, a]
normalizer = np.array([500, 100, 7])
X = []
for i in tqdm(range(data.shape[0]), desc="Kernel initialization"):
    # Get state, action, next state, and reward
    s = data.iloc[i, 0]
    a = data.iloc[i, 1]
    x = state_action_to_pos_vel(s, a) / normalizer
    X.append(x)
X = np.array(X)

policy = None
```

```
for repet in range(1):
    print("\nRepetition: " + str(repet))
    # Initialize parameters
    alpha_max = 0.1
    gamma = 0.99
    epsilon = 0.2
    num_episodes = 500000
    num_iter = 1000

    # Q-learning algorithm
    # (It is adapted since we do not contain all couples of state-action)
    for episode in tqdm(range(num_episodes), desc="Q-learning"):
        # Update learning rate
        alpha = alpha_max * (1 - episode / num_episodes)
        # Initialize to random state
        s = np.random.choice(sorted_set_of_states)
        idx_s = s - 1
        # Loop until the end of the episode
        for _ in range(num_iter):
            idx_a = None
            # Choose action
            if np.random.rand() < epsilon:
                a = sample_action(data_dict, s)
                idx_a = action_to_index[a]
            else:
                idx_a = np.argmax(Q[idx_s, :] + mask[idx_s, :])
                a = index_to_action[idx_a]
            # Get next state and reward
            (s_prime, r) = sample_outcome(data_dict, s, a)
            # Update Q-table
            idx_s_prime = s_prime - 1
            if s_prime not in set_of_states:
                if repet == 0:
                    Q[idx_s, idx_a] += alpha * (r - Q[idx_s, idx_a])
                else:
                    Q[idx_s, idx_a] += alpha * (r + gamma * np.max(Q[
    idx_s_prime, :]) - Q[idx_s, idx_a])
                break
            if repet == 0:
                Q[idx_s, idx_a] += alpha * (r + gamma * np.max(Q[idx_s_prime,
     :] + mask[idx_s_prime, :]) - Q[idx_s, idx_a])
            else:
                Q[idx_s, idx_a] += alpha * (r + gamma * np.max(Q[idx_s_prime,
     :]) - Q[idx_s, idx_a])
            # Update state
            s = s_prime
            idx_s = idx_s_prime

    # Generates policy
    policy = np.zeros(50000, dtype=int)
```

```
    # Kernel smoothing
    Y = []
    for i in tqdm(range(data.shape[0]), desc="Kernel initialization"):
        # Get state, action, next state, and reward
        s = data.iloc[i, 0]
        a = data.iloc[i, 1]
        idx_s = s - 1
        idx_a = action_to_index[a]
        Y.append(Q[idx_s, idx_a])
    Y = np.array(Y)

    ks = KernelSmoothing(bandwidth=0.03)
    ks.fit(X, Y)

    for pos in tqdm(range(500), desc="Kernel smoothing"):
        for vel in range(100):
            for idx_a in range(7):
                idx_s = pos + 500 * vel
                s = 1 + idx_s
                a = index_to_action[idx_a]
                if mask[idx_s, idx_a] < 0:
                    x = state_action_to_pos_vel(s, a) / normalizer
                    Q[idx_s, idx_a] = ks.predict(x)
                else:
                    x = state_action_to_pos_vel(s, a) / normalizer
                    Q[idx_s, idx_a] = (Q[idx_s, idx_a] + 2.0 * ks.predict(x))
     / 3.0

    policy = np.zeros(50000, dtype=int)
    for s in tqdm(range(1, 50001)):
        idx_s = s - 1
        policy[idx_s] = index_to_action[np.argmax(Q[idx_s, :])]

    # Compute optimal policy for each state
    save_policy(policy, "medium_" + str(repet))

# Plot poliicy
# The X-axis is the velocity and the Y-axis is the position
# The color of each cell is the action to take
# The colorbar on the right shows the action
# A second plot shows the value function

# Create a figure
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(10, 5))

# Add a subplot
ax = fig.add_subplot(1, 2, 1)
```

```
# Plot the policy (as a heatmap)
# Makes it square
ax.imshow(policy.reshape(500, 100), cmap="viridis", aspect=0.2)
# Set title
plt.title("Policy")

# Add labels
plt.xlabel("Velocity")
plt.ylabel("Position")

# Add a subplot
ax = fig.add_subplot(1, 2, 2)

# Plot the value function
ax.imshow(np.max(Q, axis=1).reshape(500, 100), cmap="viridis", aspect=0.2)
# Set title
plt.title("Value function")

# Add labels
plt.xlabel("Velocity")
plt.ylabel("Position")

# Show the plot
plt.show()
```

```
# Implements Q-learning for data/large.csv file
# Uses heuristics to replace missing data (state-action pairs)
from scipy.spatial import cKDTree
from utils import get_data_dict, get_data, sample_outcome, save_policy
import numpy as np
import random
from tqdm import tqdm

# Suppress/hide the warning
np.seterr(invalid='ignore')

class NearestNeighborSmoothing:

    def __init__(self, k):
        self.k = k

    def fit(self, X, Y):
        self.X = X
        self.Y = Y
        self.kd_tree = cKDTree(X)

    def predict(self, x):
        distances, indices = self.kd_tree.query(x, k=self.k, workers=-1)
        # Uses a gaussian kernel to weight the neighbors
        weights = self.gaussian_kernel(distances)
        y_pred = np.sum(weights * self.Y[indices]) / np.sum(weights)
        return y_pred

    def gaussian_kernel(self, distances):
        return np.exp(-0.5 * ((distances / self.k) ** 2))

# Initialize Q-table
num_states = 312020
num_actions = 9
Q = np.zeros((num_states, num_actions))

# Get data
size = 'large'
data = get_data(size)
data_dict = get_data_dict(size)

# Initialize Q-table
mask = -np.inf * np.ones((num_states, num_actions))
for i in tqdm(range(data.shape[0]), desc="Generating mask"):
    # Get state, action, next state, and reward
    s = data.iloc[i, 0]
    a = data.iloc[i, 1]
    idx_s = s - 1
    idx_a = a - 1
```

```
    mask[idx_s, idx_a] = 0


def next_state(s, a):
    if a >= 5:
        return s
    elif a == 1:
        if s%100 == 20:
            return s
        if s%100 == 10:
            return s+1
        if s%10 == 4:
            return s+6
        elif s%10 == 0:
            return s+4
        else:
            return s+1
    elif a == 2:
        if s % 10 == 0:
            return s -  6
        if s % 100 == 11:
            return s - 1
        if s % 10 == 1:
            return s
        return s - 1
    elif a == 3:
        if s%10000 // 100 == 1:
            return s
        if s%1000 // 100 == 0:
            return s - 600
        else:
            return s - 100
    elif a == 4:
        if s%10000 // 100 == 20:
            return s
        if s%1000 // 100 == 4:
            return s + 600
        else:
            return s + 100



# The heuistics for the reward is that if the state is in the data for a
    different action, then the reward is the same, otherwise it is 0.
# This only applies if action <= 4, otherwise the reward is 0.

state_non_zero_reward = {}
for row in data.iterrows():
    state = row[1]['s']
    action = row[1]['a']
```

```
        reward = row[1]['r']
        if action <= 4 and reward != 0:
            state_non_zero_reward[state] = reward

    def get_reward(state, action):
        if action >= 5:
            return 0
        if state in state_non_zero_reward:
            return state_non_zero_reward[state]
        return 0

    def get_outcome(state, action):
        # Tries to get the outcome from the data
        # If it is not in the data, it uses the heurestics
        if state in data_dict and action in data_dict[state]:
            return sample_outcome(data_dict, state, action)
        else:
            ns = next_state(state, action)
            if ns > num_states:
                ns = state
            if ns < 1:
                ns = state
            return ns, get_reward(state, action)

    # Path to save Q-table
    Q_path = 'Q_tables-large_q_learning.npy'

    import os
    if os.path.exists(Q_path):
        Q = np.load(Q_path)
        print("Q-table loaded from file")
    else:
        print("Q-table not found, creating new one")
        # Hyperparameters
        alpha_max = 0.1
        gamma = 0.95
        epsilon = 0.2
        num_episodes = 500000
        num_iter = 5000

        # Q-learning algorithm
        # (It is adapted since we do not contain all couples of state-action)
        for episode in tqdm(range(num_episodes), desc='Episode'):
            # Update learning rate
            alpha = alpha_max * (1 - episode/num_episodes)
            # Initialize to random state
            s = 1 + random.randint(0, num_states-1)
            # Loop until the end of the episode
            for i in range(num_iter):
                # Choose action
```

```
            if random.random() < epsilon:
                a = 1 + random.randint(0, num_actions-1)
            else:
                a = 1 + np.argmax(Q[s-1])
            # Get next state
            s_prime, r = get_outcome(s, a)
            # Update Q-table
            Q[s-1][a-1] += alpha * (r + gamma * np.max(Q[s_prime-1]) - Q[s
    -1][a-1])
            # Update state
            s = s_prime

    # Save Q-table
    np.save(Q_path, Q)

# Get the policy
policy = np.argmax(Q, axis=1) + 1

# Save policy
save_policy(policy, 'large_q_learning')


X = []
action_to_pos = [0.2, 0.25, 0.3, 0.35, 0.796, 0.797, 0.798, 0.799, 0.8]
normalizer = np.array([0.1, 0.1, 0.05, 0.01, 0.02, 0.01, 1])
for i in tqdm(range(data.shape[0]), desc="Kernel initialization"):
    # Get state, action, next state, and reward
    s = data.iloc[i, 0]
    a = data.iloc[i, 1]
    x = np.array([float(d) for d in str(s)] + [action_to_pos[a-1]]) *
    normalizer
    X.append(x)
X = np.array(X)

# Kernel smoothing
Y = []
for i in tqdm(range(data.shape[0]), desc="Kernel initialization"):
    # Get state, action, next state, and reward
    s = data.iloc[i, 0]
    a = data.iloc[i, 1]
    idx_s = s - 1
    idx_a = a - 1
    Y.append(Q[idx_s, idx_a])
Y = np.array(Y)

ks = NearestNeighborSmoothing(k=100)
ks.fit(X, Y)


for idx_s in tqdm(range(num_states), desc="NN smoothing"):
```

```
    for idx_a in range(num_actions):
        s = 1 + idx_s
        str_s = str(s)
        str_s = "0" * (6 - len(str_s)) + str_s
        x = np.array([float(d) for d in str_s] + [action_to_pos[idx_a]]) *
    normalizer
        if mask[idx_s, idx_a] < 0:
            Q[idx_s, idx_a] = (Q[idx_s, idx_a] + 4.0 * ks.predict(x)) / 5.0

policy = np.argmax(Q, axis=1) + 1
save_policy(policy, 'large_q_learning_nn_smoothing')
```