EECS 498: Reinforcement Learning
Homework 1 Responses

Tejas Jha
tjha

September 23, 2018

This document includes my responses to Homework 1 questions. Responses that involved the use of coding will provide references to specific lines of code to provide a better overview of how the problem was approached. The code can either be referenced in the Appendix or in the accompanied python script submitted with this assignment.

Question 1

Question 2

Question 3

Question 4

Question 5

Appendix: Relevant Code - tjha.py

```python
import numpy as np
import random
import math

#######################################################
# Tejas Jha
# 24 September 2018
# EECS 498 Special Topics: Reinforcement Learning
# Homework 1
# Code corresponding to Q4 and Q5
#######################################################


```

```python
# For Q4:
#
# This program provides experimentations with Epsilon-Greedy,
# Optimistic Initial Values, and UCB algorithms to learn solutions
# to a 5-arm bandit problem.
#
# Consider the arm rewards to be Bernoulli Distribution with means
# [0.1, 0.275, 0.45, 0.625, 0.8]
#
# Each algorithm should be performed for 2000 runs for each parameter set
# Each run should choose actions over 1000 time steps

# For Q5
#
#
#
#


# Creation of class to store Bernoulli distribution mean and return reward
class Arm:
    # Initialize arm given
    def __init__(self, mean):
        self.mean = mean

    # Simulate Bernoulli distribution
    # Get random float between 0 and 1, return 0 if greater than mean, 1 otherwise
    def reward(self):
        if random.random() > self.mean:
            return 0.0
        else:
            return 1.0



# Creation of a class to store details during each time step and house
# necessary functions needed during usage.
# Note: I used the code found here: https://gist.github.com/nagataka/c02b9acd6e8a8d7696e09f8a129d3362
#        for inspiration and guidance on how to construct my own functions and organize my code
class EpsilonGreedy:
    # Initialize class values
    # epsilon - the rate of exploration (randomly choosing an arm)
    # counts - N(a), number of pulls for each arm
```

```python
    # values - Q(a) average reward received from each arm (action-value estimates)
    def __init__(self, epsilon, counts, values):
        self.epsilon = epsilon
        self.counts = counts
        self.values = values

    # Performs a single time step update through epsilon-greedy algorithm
    def step(self, arms):
        # Step 1 - Determin which arm to pick next
        selected_arm_idx = 0
        if random.random() > self.epsilon:
            # EXPLOITATION
            # Choose action with largest expected action-value (break ties randomly)
            max_value = max(self.values)
            all_max_idx = [idx for idx, val in enumerate(self.values) if val == max_value]
            selected_arm_idx = all_max_idx[0]

            # (Break ties randomly)
            if len(all_max_idx) > 1:
                # Randomly choose an index from all_max_idx
                selected_arm_idx = random.choice(all_max_idx)
        else:
            # EXPLORATION
            # Choose random index to explore
            selected_arm_idx = random.choice(range(5))

        # Step 2 - Get the reward for that arm
        reward = arms[selected_arm_idx].reward()
        # Step 3 - Update count
        self.counts[selected_arm_idx] += 1
        # Step 4 - Update action-value
        self.values[selected_arm_idx] += (1 / self.counts[selected_arm_idx])*(reward - self.values[selected_arm_idx])

    # Get the action-value estimates for EpsilonGreedy
    def getValues(self):
        return self.values

# Creation of another Class variable that may be fixed to be universal type
class ModelClass:
    def __init__(self, epsilon=0, counts=np.zeros(5), values=np.zeros(5), c=0):
        self.epsilon = epsilon
```

```
        self.counts = counts
        self.values = values
        self.c = c

    def step(self, iteration, arms):

        print(iteration)

        # Step 1 -Select the Action with the largest upper bound
        # If count is 0, the index is given priority
        all_zero_idx = [idx for idx, val in enumerate(self.counts) if val == 0]
        selected_arm_idx = 0
        #updated_val = 0
        if len(all_zero_idx) > 0:
            selected_arm_idx = random.choice(all_zero_idx)
        else:
            # Select the arm with the largest estimated upper bound
            estimates = self.values

            for idx in range(len(estimates)):
                estimates[idx] += self.c*math.sqrt(math.log(iteration)/self.counts[idx])

            max_value = max(estimates)
            all_max_idx = [idx for idx, val in enumerate(estimates) if val == max_value]
            selected_arm_idx = all_max_idx[0]

            # (Break ties randomly)
            if len(all_max_idx) > 1:
                # Randomly choose an index from all_max_idx
                selected_arm_idx = random.choice(all_max_idx)

            #updated_val = estimates[selected_arm_idx]

        # Step 2 - Get the reward for that arm
        reward = arms[selected_arm_idx].reward()
        # Step 3 - Update count
        self.counts[selected_arm_idx] += 1
        # Step 4 - Update action-value
        self.values[selected_arm_idx] += (1 / self.counts[selected_arm_idx])*(reward - self.values[selected_arm_idx])




    def getValues(self):
```

```python
        return self.values



# Epsilon-Greedy Algorithm run 2000 times given epsilon value
def epsilon_greedy_alg(epsilon, arms):

    # Print message to output describing Algorithm being run
    print("**************************************************************")
    print("Performing Epsilon-Greedy Algorithm 2000 times with epsilon = " + str(epsilon))

    # Loop through algorithm for 2000 runs
    for current_run in range(2000):

        # Each run should set a distinct random seed
        random.seed()
        # Create EpsilonGreedy data type initialized with epsilon and 0 for counts and values
        model = EpsilonGreedy(epsilon, np.zeros(5), np.zeros(5))

        # Perform a single run through algorithm using 1000 time steps
        for i in range(1000):
            model.step(arms)

        if (current_run + 1) % 500 == 0:
            print("     Completed Run #" + str(current_run + 1))
            print(model.getValues())

    print("Completed Runs for epsilon = " + str(epsilon))
    print("**************************************************************")


# Optimistic Initial Value Algorithm run 2000 times given inital value (assume greedy arm selection)
def optimistic_initial_value_alg(initial_val, arms):

    # Print message to output describing Algorithm being run
    print("**************************************************************")
    print("Performing Optimistic Initial Value Algorithm 2000 times with inital value = " + str(initial_val))

```

```python
    # Loop through algorithm for 2000 runs
    for current_run in range(2000):

        # Each run should set a distinct random seed
        random.seed()
        # Create EpsilonGreedy data type initialized with epsilon = 0
            and 0 for counts and inital values
        initialized_values_array = np.full(5,initial_val)
        model = EpsilonGreedy(0, np.zeros(5), initialized_values_array)


        # Perform a single run through algorithm using 1000 time steps
        for i in range(1000):
            model.step(arms)

        if (current_run + 1) % 500 == 0:
            print("     Completed Run #" + str(current_run + 1))
            print(model.getValues())

    print("Completed Runs for initial value = " + str(initial_val))
    print("
        **********************************************************************
        ")


# Upper Confidence Bound (UCB) Algorithm run 2000 times with given c
    parameter
def upper_confidence_bound_alg(c, arms):

    # Print message to output describing Algorithm being run
    print("
        **********************************************************************
        ")
    print("Performing UCB Algorithm 2000 times with c = " + str(c))

    # Loop through algorithm for 2000 runs
    for current_run in range(2000):

        # Each run should set a distinct random seed
        random.seed()

        # Initialize model with c value
        model = ModelClass(0,np.zeros(5),np.zeros(5),c)

        # Perform a single run through algorithm using 1000 time steps
        for i in range(1000):
            model.step(i+1,arms)

```

```python
            if (current_run + 1) % 500 == 0:
                print("     Completed Run #" + str(current_run + 1))
                print(model.getValues())

    print("Completed Runs for c = " + str(c))
    print("
        **********************************************************************
        ")




def main():
    # Create List of arm class variables called arms set with means for
        Bernoulli distributions
    means = [0.8, 0.275, 0.45, 0.625, 0.8]
    arms = np.array(list(map(Arm, means)))

    # Perform Epsilon-Greedy algorithm with Q1 = 0 and
    # for each epsilon = [0.01, 0.1, 0.3]
    epsilon_list = [0.01, 0.1, 0.3]
    for epsilon in epsilon_list:
        epsilon_greedy_alg(epsilon, arms)
        print("Finished with Epsilon-Greedy Algorithm for epsilon = " +
            str(epsilon))

    # Perform Optimistic Initial Value algorithm with epsilon = 0 (
        always greedy)
    # for each Q1 = [1,5,50]
    initial_val_list = [1.0, 5.0, 50.0]
    for val in initial_val_list:
        optimistic_initial_value_alg(val,arms)
        print("Finished with Optimistic Initial Value Algorithm for
            initial value = " + str(val))

    # Perform UCB algorithm with Q1 = 0
    # for each c = [0.2, 1, 2]
    c_vals = [0.2, 1, 2]
    for val in c_vals:
        upper_confidence_bound_alg(val, arms)
        print("Finished with UCB Algorithm for c = " + str(val))


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