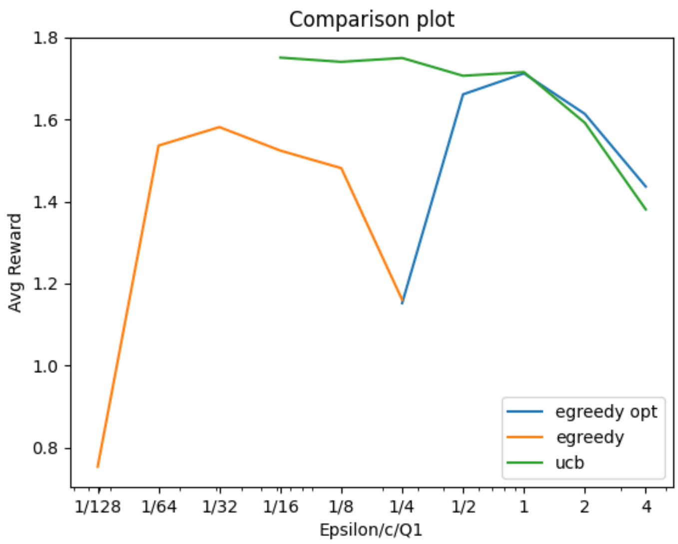
2180153 Tumi Jourdan

2424161 shakeel malagas

2332155 Tao Yuan



-

Code :

```
import numpy as np

import matplotlib.pyplot as plt

import random


class Results:

  def __init__(self,num_arms):

    self.arms = num_arms

    self.results = []

    self.rewards = []


  def store_results(self,result):

    self.results.append(result)

  def store_rewards(self,reward):

    self.rewards.append(reward)


  def displayGraphs(self):

    nprewards = np.array(self.rewards)

    rewards_reshaped = nprewards.reshape(-1, 100)

    averages = rewards_reshaped.mean(axis=1)


    plt.plot(averages)

    plt.show()



class Arm:

  def __init__(self):

    # Mean of each arm is drawn from a Gaussian with mean 0 and variance 3

    self.mean = np.random.normal(0, np.sqrt(3))

    # Variance of rewards from each arm is fixed at 1

    self.variance = 1
```

```
  def pull(self):
    # Return a reward sampled from a Gaussian with the arm's mean and variance
    reward = np.random.normal(self.mean, np.sqrt(self.variance))
    return reward

class DefaultStrategy:
  def initialize(self, num_arms):
    pass

  def update(self, arm_index, reward):
    pass

  def select_arm(self):
    return 0
# Class representing the multi-armed bandit with a given strategy
class MultiArmedBandit:
  def __init__(self, num_arms, strategy=None):
    # Initialize the arms
    self.arms = [Arm() for _ in range(num_arms)]
    self.strategy = strategy if strategy is not None else DefaultStrategy()
    self.strategy.initialize(num_arms)
    self.counts = np.zeros(num_arms)
    self.values = np.zeros(num_arms)

  def clear(self):
    self.strategy.initialize(10)
    self.counts = np.zeros(10)
    self.values = np.zeros(10)

  def pull_arm(self, arm_index):
```

```
        # Pull the specified arm and get a reward
        if 0 <= arm_index < len(self.arms):
            return self.arms[arm_index].pull()
        else:
            raise ValueError("Invalid arm index.")

    def update_estimates(self, arm_index, reward):
        # Update the count for the chosen arm
        self.counts[arm_index] += 1
        # Update the estimated value of the chosen arm using incremental formula
        self.values[arm_index] += (reward - self.values[arm_index]) / self.counts[arm_index]

    def update_estimatesUCB(self, arm_index, reward):
        self.strategy.update(arm_index, reward)

    # Method to select the next arm to pull based on the strategy
    def select_arm(self, epsilon):
        # Epsilon-greedy action selection
        if np.random.rand() < epsilon:
            # Exploration: select a random arm
            return np.random.randint(0, len(self.arms))
        else:
            # Exploitation: select the arm with the highest estimated value
            return np.argmax(self.values)

    def select_armUCB(self,c):
        return self.strategy.select_arm(c)

class Strategy:
    def initialize(self, num_arms):
        self.counts = np.zeros(num_arms)  # Number of times each arm has been pulled
```

```
        self.values = np.zeros(num_arms)  # Estimated value of each arm
        self.total_pulls = 0  # Total number of pulls

    # Method to update the estimates for a given arm
    def update(self, arm_index, reward):
        self.counts[arm_index] += 1
        self.total_pulls += 1
        n = self.counts[arm_index]
        value = self.values[arm_index]
        # Incremental update of the mean value estimate for the arm
        self.values[arm_index] = value + (reward - value) / n

    # Method to select the next arm to pull (to be implemented by specific strategies)
    def select_arm(self):
        raise NotImplementedError

class UCBStrategy(Strategy):
    def select_arm(self, c):
        if self.total_pulls < len(self.counts):
            # Pull each arm once initially
            return self.total_pulls
        # Calculate UCB values for each arm
        ucb_values = self.values + c * np.sqrt(np.log(self.total_pulls) / self.counts)
        # Select the arm with the highest UCB value
        return np.argmax(ucb_values)

class epsilon_greedy_optimistic():

    def __init__(self,mab,learning_alpha,optimistic_estimate,epsilon,iterations = 1000) -> None:
        self.iterations = iterations
        self.learning_alpha = learning_alpha
```

```
    self.optimistic_estimate = optimistic_estimate
    self.epsilon = epsilon
    self.mab = mab

  def run_greedy_opt(self):

    debug = False
    num_arms = 10
    running_sums = np.zeros(num_arms)
    all_rewards = np.zeros((100, self.iterations))
    for run in range(100):
      for x in range(len(running_sums)):
        running_sums[x] = self.optimistic_estimate

      rewards = np.zeros(self.iterations)
      counter = 0
      while counter<self.iterations:
        ##
        if(debug):print(running_sums)
        # Explore or exploit
        flip_result = random.random()
        if(flip_result > self.epsilon):
          ##
          if(debug):print("Exploit")
          # exploit
          max_estimate = np.max(running_sums)
          candidates = np.where(running_sums == max_estimate)[0]
          arm_index =  int(np.random.choice(candidates))
        else:
          ##
          if(debug):print("Explore")
```

```
            # Explore
            arm_index = np.random.randint(0, len(running_sums))

          if(debug):print("Selected = ",running_sums[arm_index])
          # do action
          reward = self.mab.pull_arm(arm_index)
          Qn = running_sums[arm_index]
          Rn = reward
          running_sums[arm_index] = Qn + self.learning_alpha*(Rn - Qn)
          if(np.isinf(running_sums[arm_index])):
            print("Is infinity")
            print("Rn = ", Rn)
            print("Qn = ", Qn)

          rewards[counter] = reward
          counter +=1
        all_rewards[run] = rewards

      average_rewards = np.mean(all_rewards, axis=0)
      return average_rewards

    def run_greedy_opt_1(self,new_Q0):
      debug = False
      num_arms = 10
      running_sums = np.zeros(num_arms)
      for x in range(len(running_sums)):
        running_sums[x] = new_Q0
      rewards = 0
      counter = 0
      while counter<self.iterations:
        ##
```

```
    if(debug):print(running_sums)
    # Explore or exploit
    flip_result = random.random()
    if(flip_result > self.epsilon):
      ##
      if(debug):print("Exploit")
      # exploit
      max_estimate = np.max(running_sums)
      candidates = np.where(running_sums == max_estimate)[0]
      arm_index =  int(np.random.choice(candidates))
    else:
      ##
      if(debug):print("Explore")
      # Explore
      arm_index = np.random.randint(0, len(running_sums))

    if(debug):print("Selected = ",running_sums[arm_index])
    # do action
    reward = self.mab.pull_arm(arm_index)
    Qn = running_sums[arm_index]
    Rn = reward
    running_sums[arm_index] = Qn + self.learning_alpha*(Rn - Qn)
    if(np.isinf(running_sums[arm_index])):
      print("Is infinity")
      print("Rn = ", Rn)
      print("Qn = ", Qn)

    rewards+= reward
    counter+=1
  if(debug):print(counter)
  return rewards/self.iterations
```

```
# Function to run the UCB algorithm
def UCB(bandit_ucb,c):

    num_iterations = 1000
    num_runs = 100

    all_rewards = np.zeros((num_runs, num_iterations))

    for run in range(num_runs):

        bandit_ucb.clear()
        rewards = np.zeros(num_iterations)

        for i in range(num_iterations):
            arm_index = bandit_ucb.select_armUCB(c)
            reward = bandit_ucb.pull_arm(arm_index)
            bandit_ucb.update_estimatesUCB(arm_index, reward)
            rewards[i] = reward

        all_rewards[run] = rewards

    average_rewards = np.mean(all_rewards, axis=0)

    return average_rewards

def Egreedy(mab):

    num_iterations = 1000  # Number of iterations
    epsilon = 0.1  # Exploration rate
```

```
    all_rewards = np.zeros((100, num_iterations))

    for run in range(100):
        mab.clear()
        rewards = np.zeros(num_iterations)

        for i in range(num_iterations):
            # Select an arm to pull using epsilon-greedy strategy
            arm_index = mab.select_arm(epsilon)
            # Pull the selected arm
            reward = mab.pull_arm(arm_index)

            # Update the value estimates for the selected arm
            mab.update_estimates(arm_index, reward)
            rewards[i] = reward

        all_rewards[run] = rewards

    average_rewards = np.mean(all_rewards, axis=0)

    return average_rewards

def UCBComparison(bandit_ucb,c):

    num_iterations = 1000

    bandit_ucb.clear()
    rewards=0

    for i in range(num_iterations):
        arm_index = bandit_ucb.select_armUCB(c)
```

```
        reward = bandit_ucb.pull_arm(arm_index)
        bandit_ucb.update_estimatesUCB(arm_index, reward)
        rewards+= reward
    rewards=rewards/1000
    return rewards


def EgreedyL(mab, epsilon):

    num_iterations = 1000  # Number of iterations
    #epsilon = 0.1  # Exploration rate

    all_rewards = np.zeros((100, num_iterations))

    mab.clear()
    rewards = 0

    for i in range(num_iterations):
        # Select an arm to pull using epsilon-greedy strategy
        arm_index = mab.select_arm(epsilon)
        # Pull the selected arm
        reward = mab.pull_arm(arm_index)

        # Update the value estimates for the selected arm
        mab.update_estimates(arm_index, reward)
        rewards += reward

    rewards = rewards/1000

    return rewards
```

```
def EgreedyPart2(mab):
    epsilonLRewards = []
    epsilons = [1/128, 1/64, 1/32, 1/16, 1/8, 1/4]
    for epsilon in epsilons:
        x = EgreedyL(mab, epsilon)
        epsilonLRewards.append(x)

    return epsilonLRewards

ucb_strategy = UCBStrategy()
mab = MultiArmedBandit(10, ucb_strategy)
# Run the UCB algorithm
egreedy=Egreedy(mab)
ucb=UCB(mab,2)
ego = epsilon_greedy_optimistic(mab,0.1,5,0.01)
ego_results = ego.run_greedy_opt()

plt.plot(egreedy, label="e greedy")
plt.plot(ego_results,label = "e greedy optimised")
plt.plot(ucb, label="UCB")
plt.legend(loc="lower right")
plt.xlabel('Iterations')
plt.ylabel('Reward')
plt.title('Reward per Iteration')
plt.show()
# Run the UCB algorithm
egreedy=Egreedy(mab)
egreedy2 = EgreedyPart2(mab)

rewards = []
for x in [1/4,1/2,1,2,4]:
```

```
    rewards.append(ego.run_greedy_opt_1(x))

ucbplot=[]
for i in [1/16,1/8,1/4,1/2,1,2,4]:
    ucbplot.append(UCBComparison(mab,i))

exponents = np.array([-8, -6, -4, -2, 0, 2, 4])

# Calculate the values as 2 raised to the power of each exponent

plt.xscale('log')

x_values = [1/4, 1/2, 1, 2, 4]
plt.plot(x_values,rewards, label="egreedy opt")
x_values = [1/128, 1/64, 1/32, 1/16, 1/8, 1/4]
plt.plot(x_values,egreedy2, label="egreedy")
x_values = [1/16,1/8,1/4,1/2,1,2,4]
plt.plot(x_values,ucbplot, label="ucb")
plt.legend(loc="lower right")

plt.xlabel('Epsilon/c/Q1')
plt.ylabel('Avg Reward')
plt.title('Comparison plot')


x_values = [1/128, 1/64, 1/32, 1/16, 1/8, 1/4, 1/2, 1, 2, 4]
plt.xticks(x_values, ['1/128', '1/64', '1/32', '1/16', '1/8', '1/4', '1/2', '1', '2', '4'])
plt.legend()

plt.show()
```