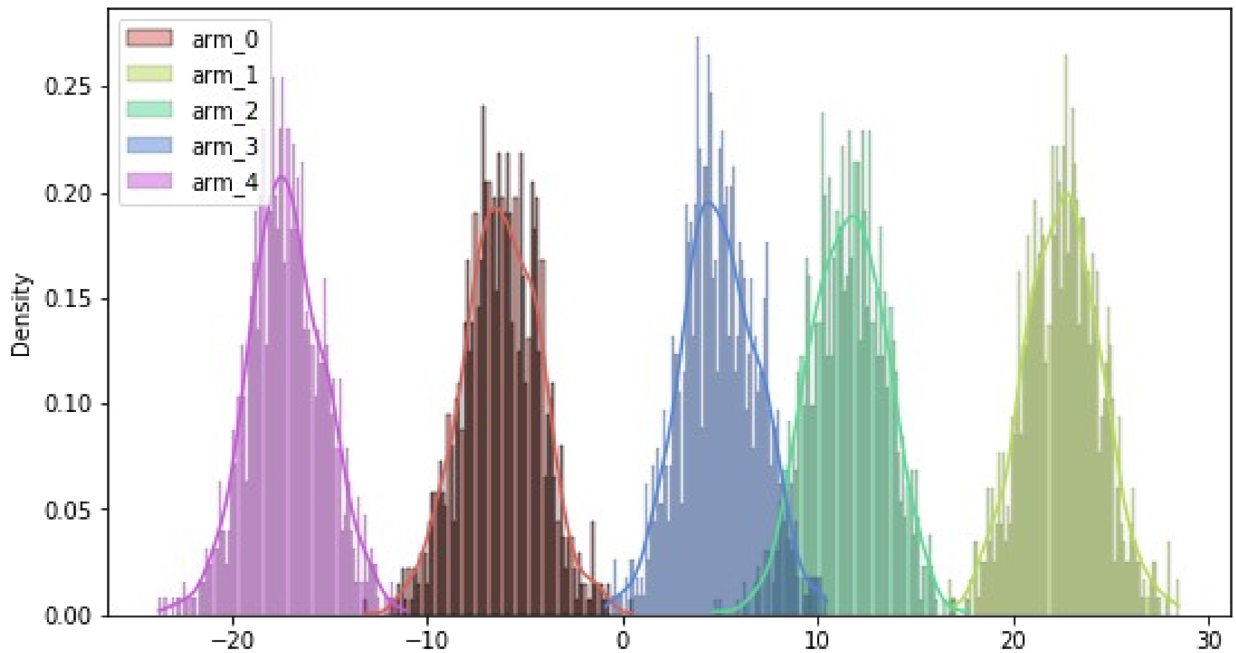
CS6700 : Tutorial 1 - Multi-Arm Bandits

Girish Madhavan V : ME20B072




```
Goal: Analysis 3 types of sampling strategy in a MAB
```

Import dependencies

```
# !pip install seaborn

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from typing import NamedTuple, List
```

Gaussian Bandit Environment

```
class GaussianArm(NamedTuple):
  mean: float
  std: float


class Env:
  def __init__(self, num_arms: int, mean_reward_range: tuple, std:
float):
    """
    num_arms: number of bandit arms
```

```
    mean_reward_range: mean reward of an arm should lie between the
given range
    std: standard deviation of the reward for each arm
    """
    self.num_arms = num_arms
    self.arms = self.create_arms(num_arms, mean_reward_range, std)

  def create_arms(self, n: int, mean_reward_range: tuple, std: float)
-> dict:
    low_rwd, high_rwd = mean_reward_range
    # creates "n" number of mean reward for each arm
    means = np.random.uniform(low=low_rwd, high=high_rwd, size=(n,))
    arms = {id: GaussianArm(mu, std) for id, mu in enumerate(means)}
    return arms

  @property
  def arm_ids(self):
    return list(self.arms.keys())

  def step(self, arm_id: int) -> float:
    arm = self.arms[arm_id]
    return np.random.normal(arm.mean, arm.std)   # Reward

  def get_best_arm_and_expected_reward(self):
    best_arm_id = max(self.arms, key=lambda x: self.arms[x].mean)
    return best_arm_id, self.arms[best_arm_id].mean

  def get_avg_arm_reward(self):
    arm_mean_rewards = [v.mean for v in self.arms.values()]
    return np.mean(arm_mean_rewards)

  def plot_arms_reward_distribution(self, num_samples=1000):
    """
    This function is only used to visualize the arm's distrbution.
    """
    fig, ax = plt.subplots(1, 1, sharex=False, sharey=False,
figsize=(9, 5))
    colors = sns.color_palette("hls", self.num_arms)
    for i, arm_id in enumerate(self.arm_ids):
      reward_samples = [self.step(arm_id) for _ in range(num_samples)]
      sns.histplot(reward_samples, ax=ax, stat="density", kde=True,
bins=100, color=colors[i], label=f'arm_{arm_id}')
    ax.legend()
    plt.show()
```

Policy

```
class BasePolicy:
  @property
  def name(self):
```

```
    return 'base_policy'

  def reset(self):
    """
    This function resets the internal variable.
    """
    pass

  def update_arm(self, *args):
    """
    This function keep track of the estimates
    that we may want to update during training.
    """
    pass

  def select_arm(self) -> int:
    """
    It returns arm_id
    """
    raise Exception("Not Implemented")
```

Random Policy

```
class RandomPolicy(BasePolicy):
  def __init__(self, arm_ids: List[int]):
    self.arm_ids = arm_ids

  @property
  def name(self):
    return 'random'

  def reset(self) -> None:
    """No use."""
    pass

  def update_arm(self, *args) -> None:
    """No use."""
    pass

  def select_arm(self) -> int:
    return np.random.choice(self.arm_ids)

class EpGreedyPolicy(BasePolicy):
    def __init__(self, epsilon: float, arm_ids: List[int]):
        self.epsilon = epsilon
        self.arm_ids = arm_ids
        self.Q = {id: 0 for id in self.arm_ids}
        self.num_pulls_per_arm = {id: 0 for id in self.arm_ids}

    @property
```

```
    def name(self):
        return f'ep-greedy ep:{self.epsilon}'

    def reset(self) -> None:
        self.Q = {id: 0 for id in self.arm_ids}
        self.num_pulls_per_arm = {id: 0 for id in self.arm_ids}

    def update_arm(self, arm_id: int, arm_reward: float) -> None:
        self.num_pulls_per_arm[arm_id] += 1
        old_q = self.Q[arm_id]
        n = self.num_pulls_per_arm[arm_id]
        new_q = old_q + (arm_reward - old_q) / n
        self.Q[arm_id] = new_q

    def select_arm(self) -> int:
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.arm_ids)
        else:
            best_arm = max(self.Q, key=self.Q.get)
            return best_arm

class SoftmaxPolicy(BasePolicy):
  def __init__(self, tau, arm_ids):
    self.tau = tau
    self.arm_ids = arm_ids
    self.Q = {id: 0 for id in self.arm_ids}
    self.num_pulls_per_arm = {id: 0 for id in self.arm_ids}

  @property
  def name(self):
    return f'softmax tau:{self.tau}'

  def reset(self):
    self.Q = {id: 0 for id in self.arm_ids}
    self.num_pulls_per_arm = {id: 0 for id in self.arm_ids}

  def update_arm(self, arm_id: int, arm_reward: float) -> None:
    self.num_pulls_per_arm[arm_id] = self.num_pulls_per_arm[arm_id] +
1
    self.Q[arm_id] = self.Q[arm_id] + (arm_reward - self.Q[arm_id]) /
self.num_pulls_per_arm[arm_id]

  def select_arm(self) -> int:
    max_q_value = max(self.Q.values())
    exp_values = {id: np.exp((self.Q[id] - max_q_value) / self.tau)
for id in self.arm_ids}
    sum_exp_values = sum(exp_values.values())
    probabilities = {id: exp_value / sum_exp_values for id, exp_value
in exp_values.items()}
```

```
    selected_arm = np.random.choice(list(probabilities.keys()),
p=list(probabilities.values()))
    return selected_arm

class UCB(BasePolicy):
    def __init__(self, c, arm_ids):
        self.c = c
        self.arm_ids = arm_ids
        self.Q = {id: 0 for id in self.arm_ids}
        self.num_pulls_per_arm = {id: 0 for id in self.arm_ids}
        self.total_pulls = 0

    @property
    def name(self):
        return f'UCB c:{self.c}'

    def reset(self):
        self.Q = {id: 0 for id in self.arm_ids}
        self.num_pulls_per_arm = {id: 0 for id in self.arm_ids}
        self.total_pulls = 0

    def update_arm(self, arm_id: int, arm_reward: float) -> None:
        self.num_pulls_per_arm[arm_id] =
self.num_pulls_per_arm[arm_id] + 1
        self.total_pulls = self.total_pulls + 1
        old_q = self.Q[arm_id]
        n = self.num_pulls_per_arm[arm_id]
        new_q = old_q + (arm_reward - old_q) / n
        self.Q[arm_id] = new_q

    def select_arm(self) -> int:
        exploration_bonus = self.c * np.sqrt(np.log(self.total_pulls +
1) / np.maximum(1, list(self.num_pulls_per_arm.values())))
        ucb_values = [self.Q[arm_id] + exploration_bonus[arm_id] for
arm_id in self.arm_ids]
        chosen_arm = max(self.arm_ids, key=lambda x: ucb_values[x])
        return chosen_arm
```

Trainer

```
def train(env, policy: BasePolicy, timesteps):
  policy_reward = np.zeros((timesteps,))
  for t in range(timesteps):
    arm_id = policy.select_arm()
    reward = env.step(arm_id)
    policy.update_arm(arm_id, reward)
    policy_reward[t] = reward
  return policy_reward

def avg_over_runs(env, policy: BasePolicy, timesteps, num_runs):
```

```
    _, expected_max_reward = env.get_best_arm_and_expected_reward()
    policy_reward_each_run = np.zeros((num_runs, timesteps))

    for run in range(num_runs):
        policy.reset()
        policy_reward = train(env, policy, timesteps)
        policy_reward_each_run[run, :] = policy_reward

    avg_policy_rewards = np.mean(policy_reward_each_run, axis=0)
    total_policy_regret = expected_max_reward * timesteps -
np.sum(avg_policy_rewards)

    return avg_policy_rewards, total_policy_regret

def plot_reward_curve_and_print_regret(env, policies, timesteps=200,
num_runs=500):
  fig, ax = plt.subplots(1, 1, sharex=False, sharey=False,
figsize=(10, 6))
  for policy in policies:
    avg_policy_rewards, total_policy_regret = avg_over_runs(env,
policy, timesteps, num_runs)
    print('regret for {}: {:.3f}'.format(policy.name,
total_policy_regret))
    ax.plot(np.arange(timesteps), avg_policy_rewards, '-',
label=policy.name)

  _, expected_max_reward = env.get_best_arm_and_expected_reward()
  ax.plot(np.arange(timesteps), [expected_max_reward]*timesteps, 'g-')

  avg_arm_reward = env.get_avg_arm_reward()
  ax.plot(np.arange(timesteps), [avg_arm_reward]*timesteps, 'r-')

  plt.legend(loc='lower right')
  plt.show()
```

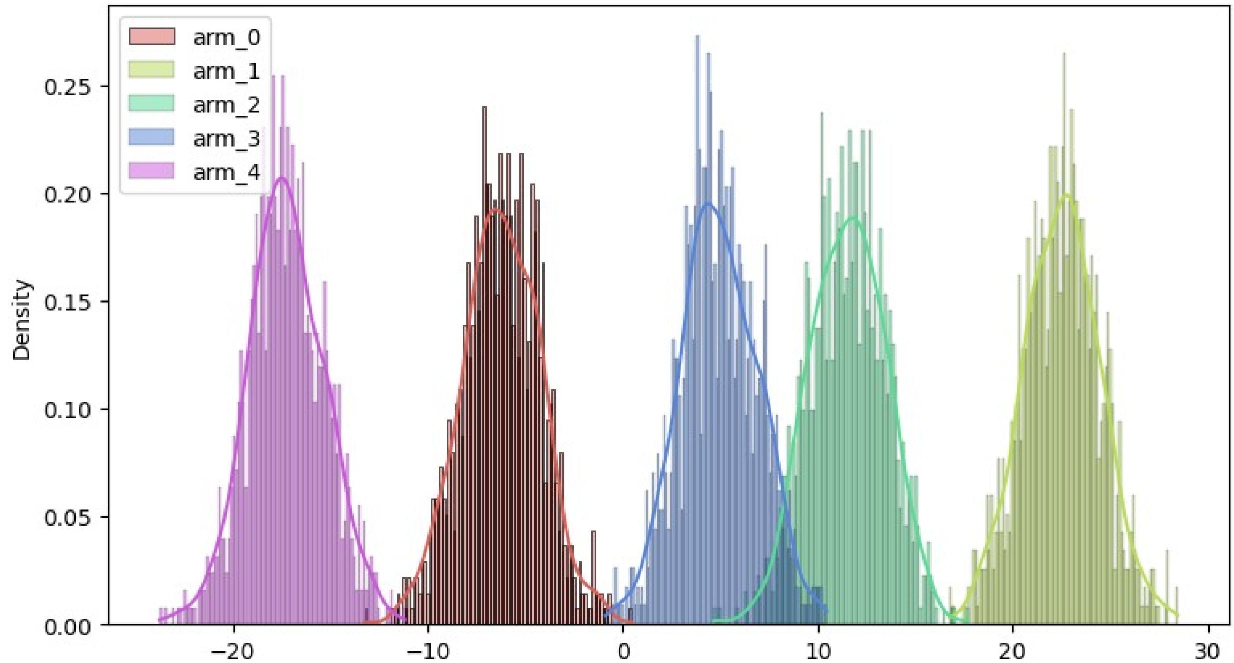
Experiments

```
seed = 42
np.random.seed(seed)

num_arms = 5
mean_reward_range = (-25, 25)
std = 2.0

env = Env(num_arms, mean_reward_range, std)

env.plot_arms_reward_distribution()
```




```
best_arm, max_mean_reward = env.get_best_arm_and_expected_reward()
print(best_arm, max_mean_reward)
```

```
1 22.53571532049581
```

```
print(env.get_avg_arm_reward())
```

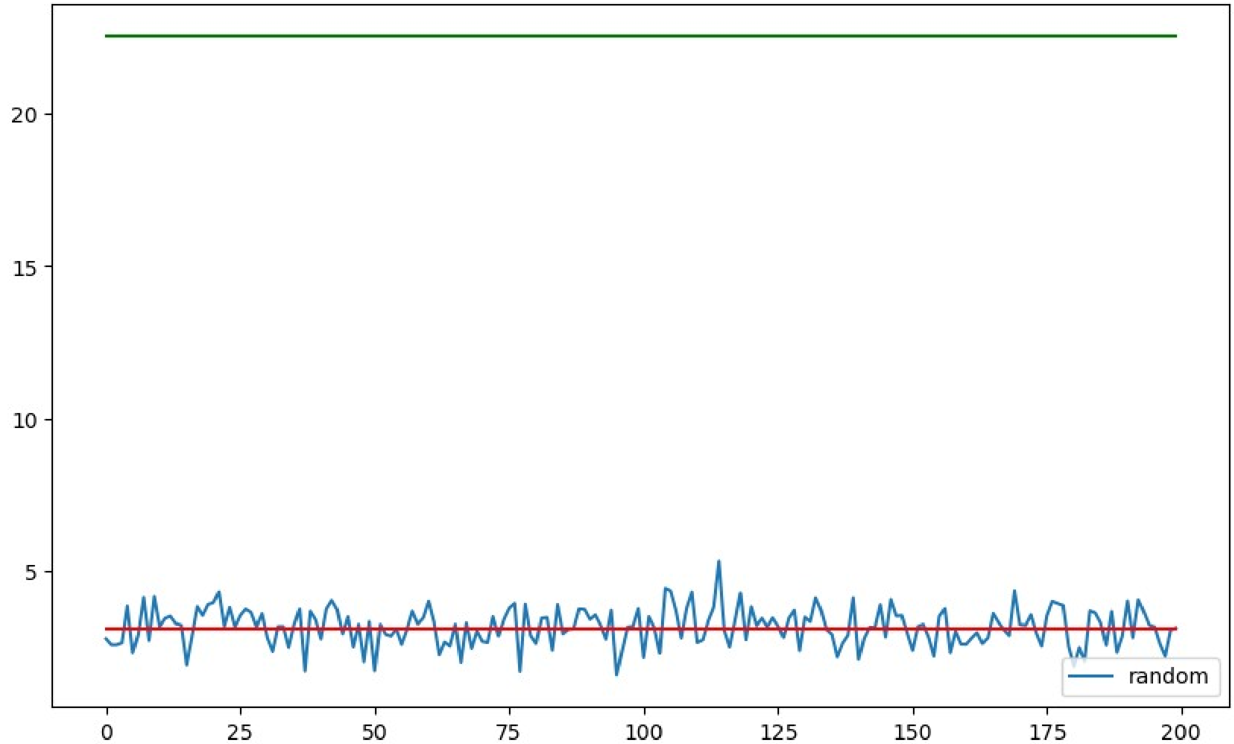
```
3.119254917081568
```

Please explore following values:

- Epsilon greedy: [0.001, 0.01, 0.5, 0.9]
- Softmax: [0.001, 1.0, 5.0, 50.0]

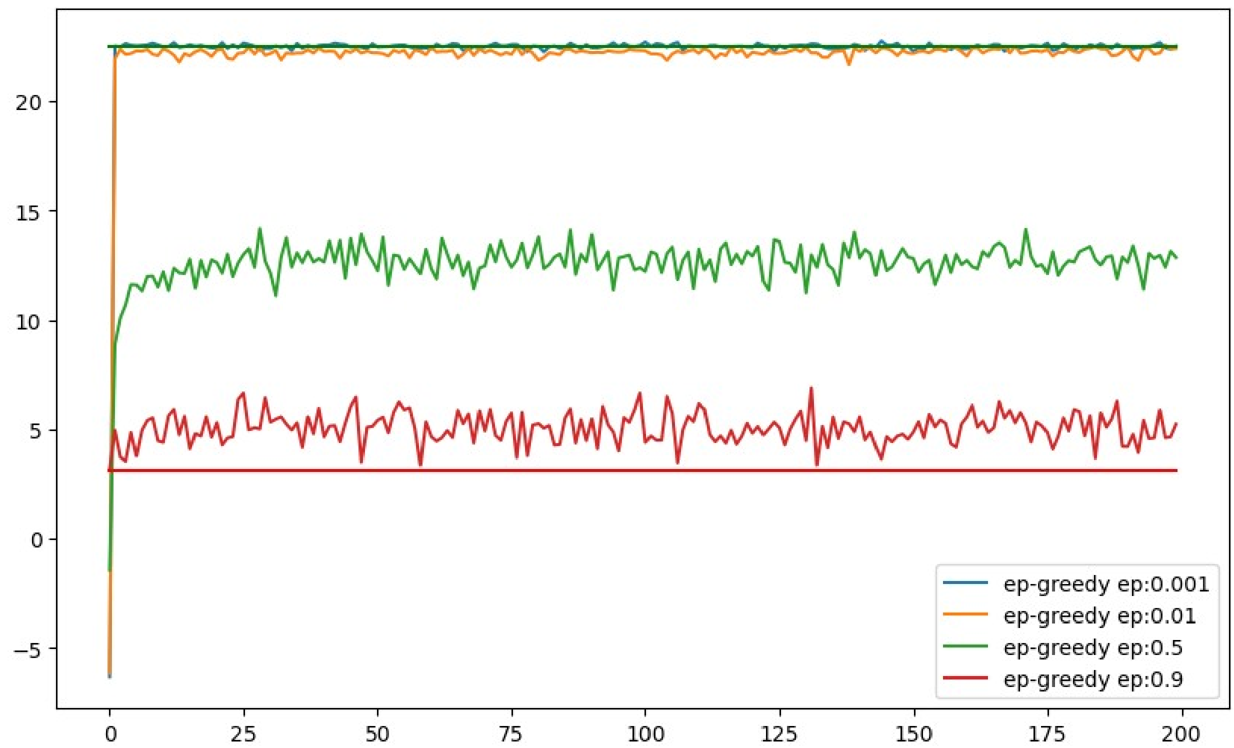
```
random_policy = RandomPolicy(env.arm_ids)
plot_reward_curve_and_print_regret(env, [random_policy],
timesteps=200, num_runs=500)
```

```
regret for random: 3871.625
```



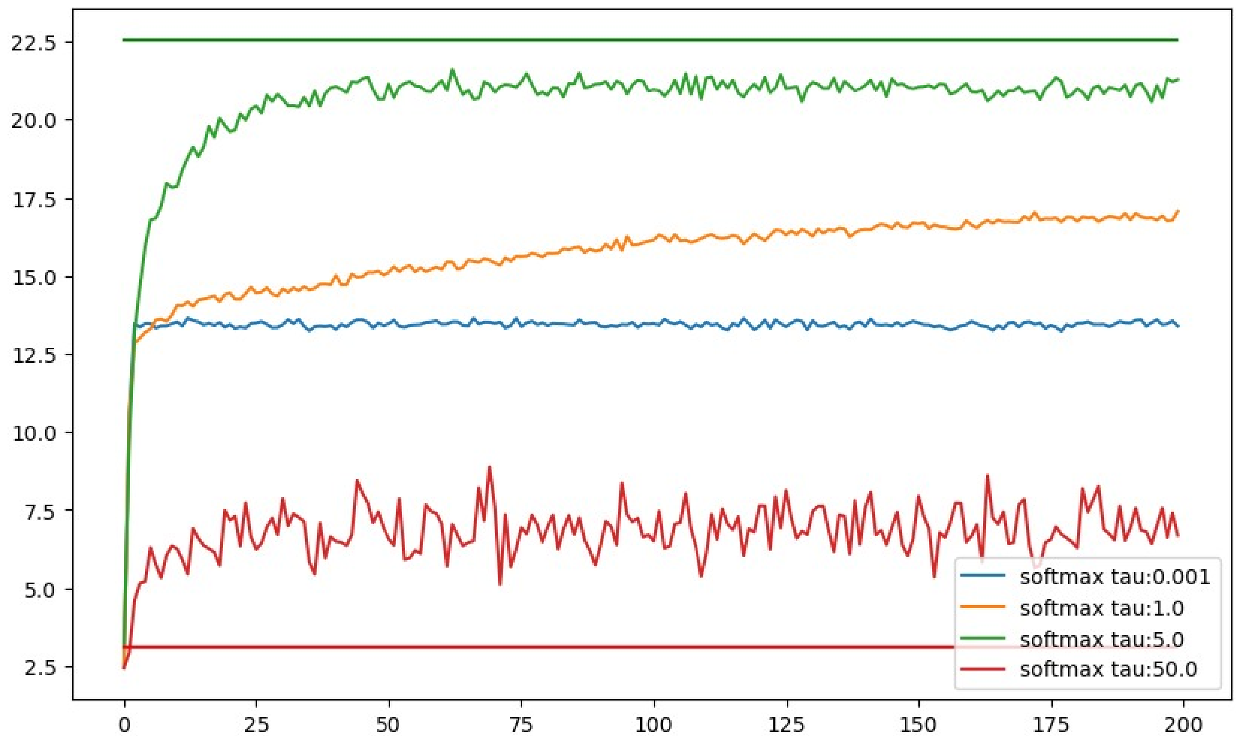

```
explore_epgreedy_epsilons =  [0.001, 0.01, 0.5, 0.9]
epgreedy_policies = [EpGreedyPolicy(ep, env.arm_ids) for ep in
explore_epgreedy_epsilons]
plot_reward_curve_and_print_regret(env, epgreedy_policies,
timesteps=200, num_runs=500)
```

```
regret for ep-greedy ep:0.001: 33.248
regret for ep-greedy ep:0.01: 85.360
regret for ep-greedy ep:0.5: 1992.935
regret for ep-greedy ep:0.9: 3497.860
```



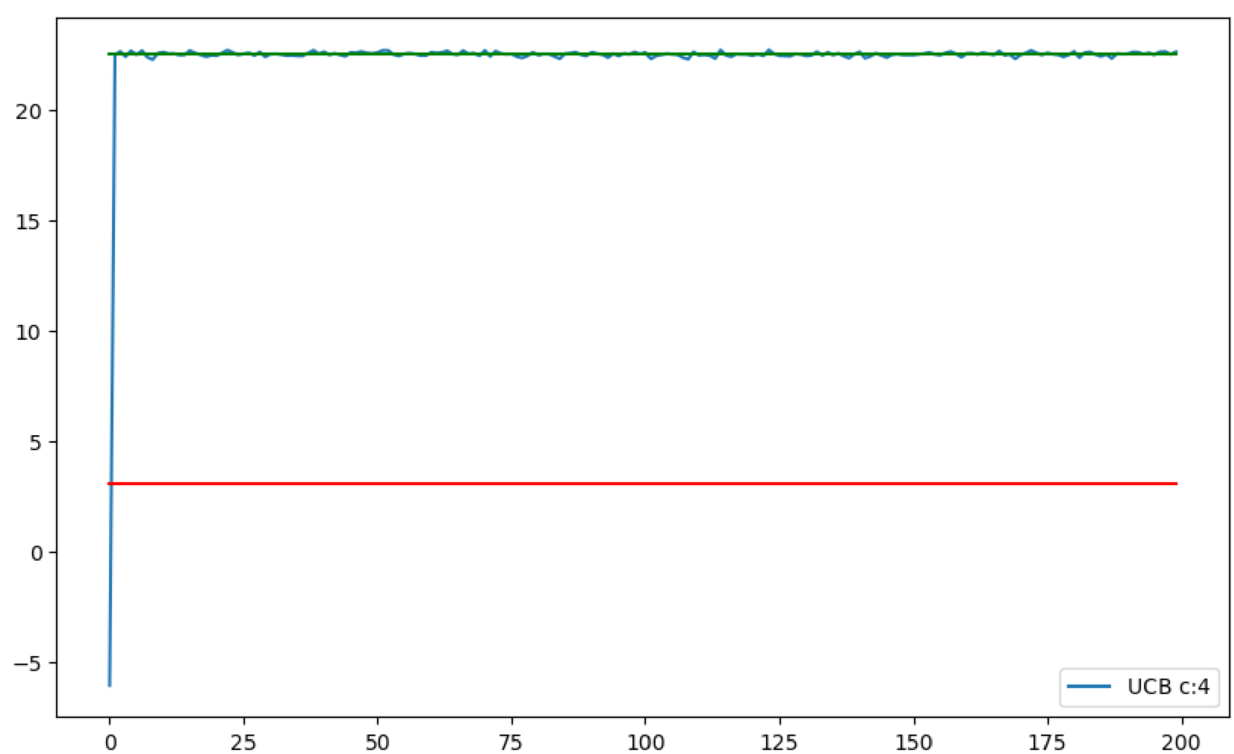

```
explore_softmax_taus =  [0.001, 1.0, 5.0, 50.0]
softmax_polices = [SoftmaxPolicy(tau, env.arm_ids) for tau in
explore_softmax_taus]
plot_reward_curve_and_print_regret(env, softmax_polices,
timesteps=200, num_runs=500)
```

```
regret for softmax tau:0.001: 1833.224
regret for softmax tau:1.0: 1368.913
regret for softmax tau:5.0: 400.365
regret for softmax tau:50.0: 3149.831
```




```
plot_reward_curve_and_print_regret(env, [UCB(4, env.arm_ids)],
timesteps=200, num_runs=500)
```

```
regret for UCB c:4: 31.505
```



Optional: Please explore different values of epsilon, tau and verify how does the behaviour changes.