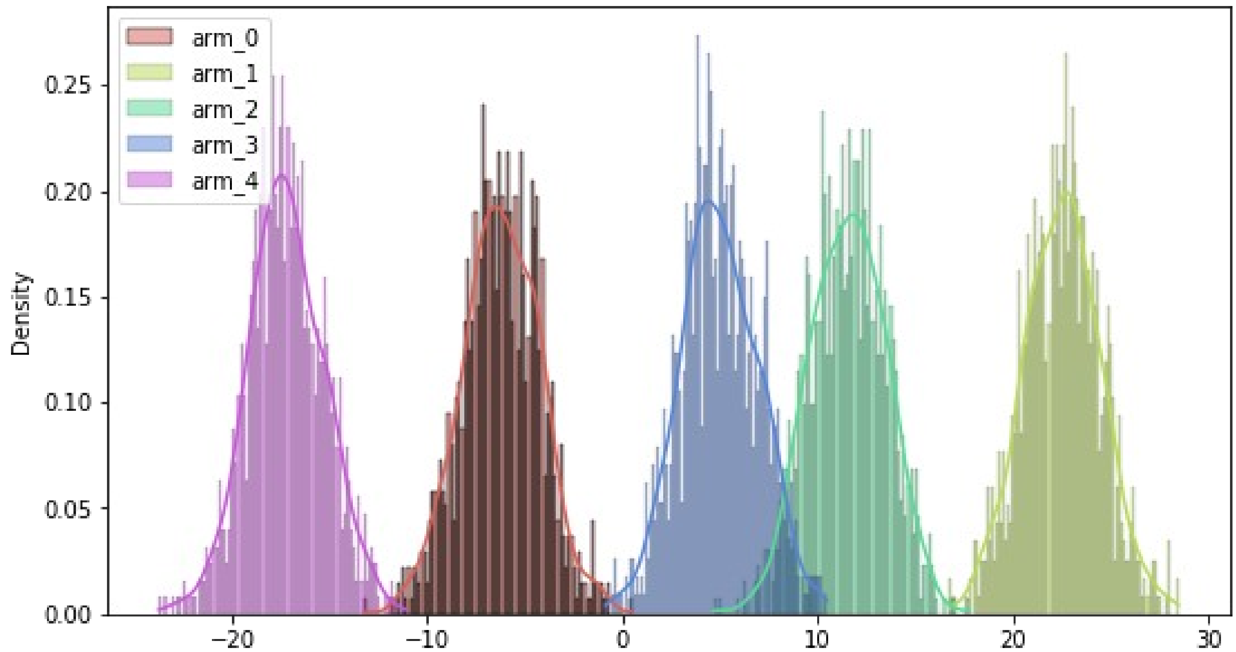
CS6700 : Tutorial 1 - Multi-Arm Bandits




```
    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'
```

```python
  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

```python
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:
    # your code for updating the Q values of each arm
    # update of the Q value of the arm should be done in-place
    self.Q[arm_id] = ((self.Q[arm_id] *
self.num_pulls_per_arm[arm_id]) + arm_reward) /
(self.num_pulls_per_arm[arm_id] + 1)
    # update the number of pulls of the arm
    self.num_pulls_per_arm[arm_id] += 1
    return

  def select_arm(self) -> int:
    # your code for selecting arm based on epsilon greedy policy
    # epsilon greedy policy
    if np.random.uniform(0, 1) <= self.epsilon:
      return np.random.randint(0, 5)
    else:
      return max(self.Q, key = self.Q.get)

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:
    # your code for updating the Q values of each arm
    # update of the Q value of the arm should be done in-place
    self.Q[arm_id] = ((self.Q[arm_id] *
self.num_pulls_per_arm[arm_id]) + arm_reward) /
(self.num_pulls_per_arm[arm_id] + 1)
    # update the number of pulls of the arm
    self.num_pulls_per_arm[arm_id] += 1
    return

  def select_arm(self) -> int:
```

```
    # your code for selecting arm based on softmax policy
    # softmax policy
    # the numerator is taken to the denominator to avoid overflow
    self.probs = [1/np.sum([np.exp(self.Q[i]/self.tau -
self.Q[id]/self.tau) for i in self.arm_ids]) for id in self.arm_ids]
    # return the arm id based on the probability distribution
    return np.random.choice(self.arm_ids, p = self.probs)


class UCB(BasePolicy):
  # your code here
  def __init__(self, arm_ids: List[int], c: float):
    self.arm_ids = arm_ids
    self.c = c
    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_num_pulls = 0

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

  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}
    self.total_num_pulls = 0

  def update_arm(self, arm_id: int, arm_reward: float) -> None:
    # your code for updating the Q values of each arm
    # update the Q value for the arm selected
    self.Q[arm_id] = ((self.Q[arm_id] *
self.num_pulls_per_arm[arm_id]) + arm_reward) /
(self.num_pulls_per_arm[arm_id] + 1)
    # update the number of pulls of the arm
    self.num_pulls_per_arm[arm_id] += 1
    # update the total number of pulls
    self.total_num_pulls += 1
    return

  def select_arm(self) -> int:
    # your code for selecting arm based on epsilon greedy policy
    # play every arm at least once
    if self.total_num_pulls < len(self.arm_ids):
      return self.total_num_pulls
    # UCB1 policy
    optimism = [self.Q[id] + self.c *
np.sqrt(np.log(self.total_num_pulls) / self.num_pulls_per_arm[id]) for
id in self.arm_ids]
    return optimism.index(max(optimism))
```

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

  # calculate avg policy reward from policy_reward_each_run
  # averaging over the runs (axis=0)
  avg_policy_rewards = np.mean(policy_reward_each_run, axis = 0) #
your code here (type: nd.array, shape: (timesteps,))
  # calculate total policy regret
  total_policy_regret = np.sum([expected_max_reward -
avg_policy_rewards[i] for i in range(timesteps)]) # your code here
(type: float)

  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()
```

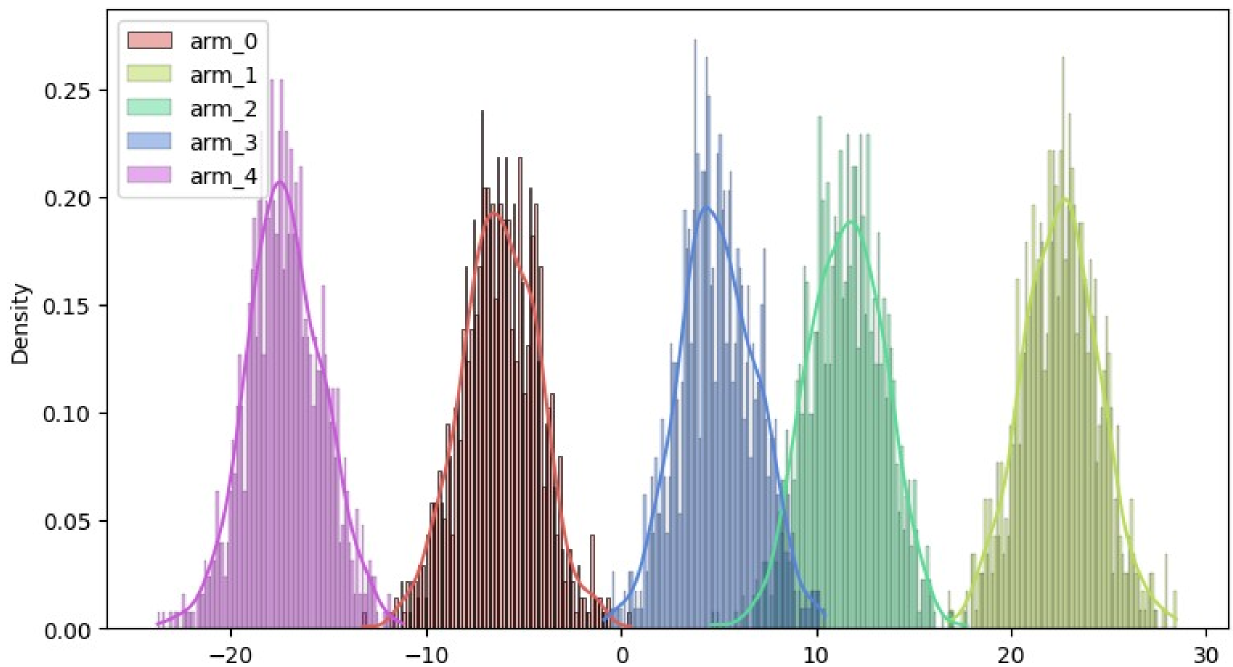
Experiments

```
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())
```

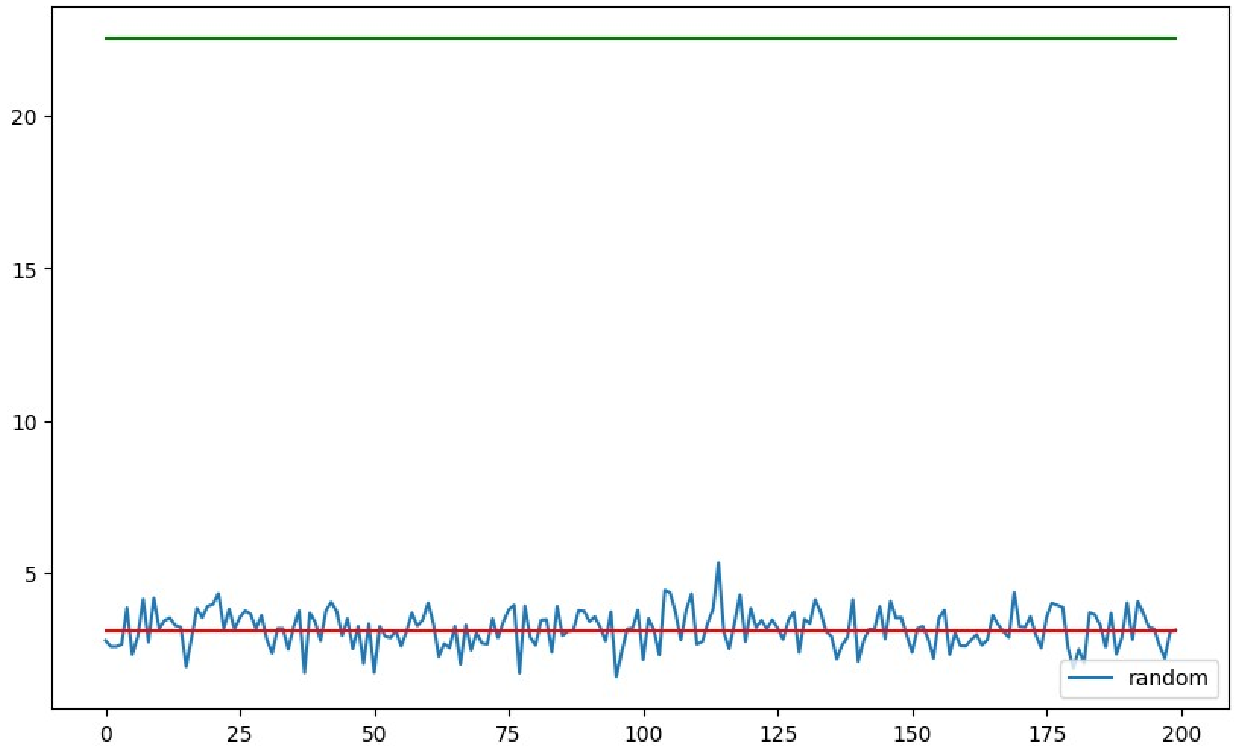
```
3.119254917081568
```

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
```

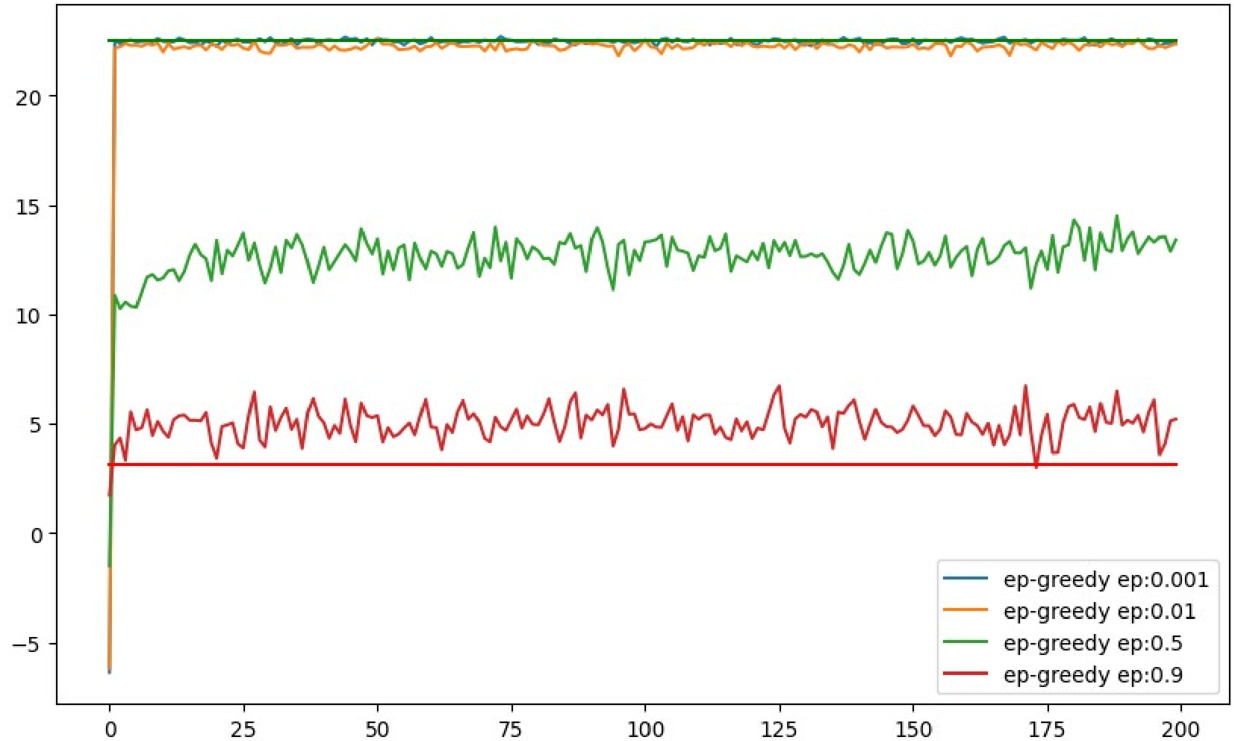


Inferences

- Random policy does not converge to the optimal policy.
- The average reward that this policy learns to obtain is around the average of mean rewards for all arms.

```
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: 39.590
regret for ep-greedy ep:0.01: 83.511
regret for ep-greedy ep:0.5: 1980.353
regret for ep-greedy ep:0.9: 3505.350
```



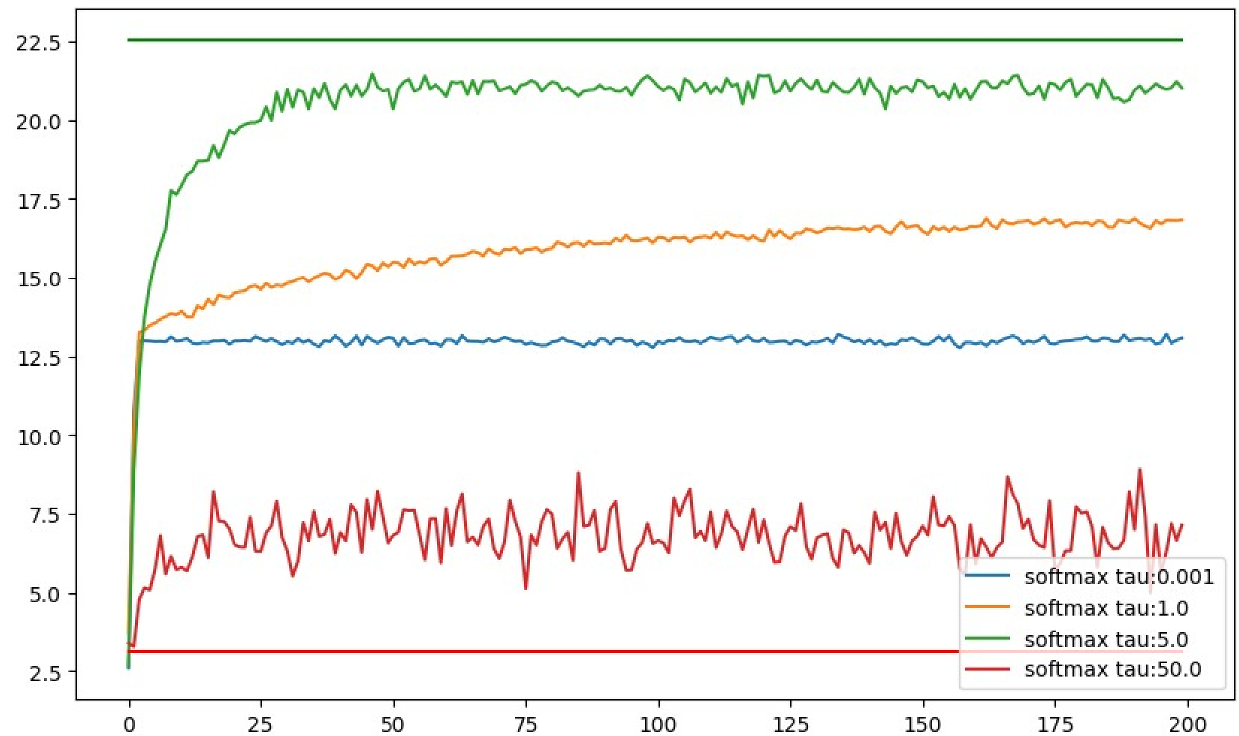
Inferences

- We observe that ϵ value of 0.001 results in the least regret among the cases that have been tried.
- We also observe that ϵ values 0.5 and 0.9 does not converge to the arm that results in the optimal payoff by the end of 200 timesteps.
- From the explore-explot dilemma, in this particular problem, exploiting might result in better results than otherwise.

```
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)
```

```
C:\Users\srika\AppData\Local\Temp\ipykernel_9460\21103497.py:28:
RuntimeWarning: overflow encountered in exp
  self.probs = [1/np.sum([np.exp(self.Q[i]/self.tau -
self.Q[id]/self.tau) for i in self.arm_ids]) for id in self.arm_ids]

regret for softmax tau:0.001: 1922.557
regret for softmax tau:1.0: 1344.711
regret for softmax tau:5.0: 411.401
regret for softmax tau:50.0: 3150.510
```

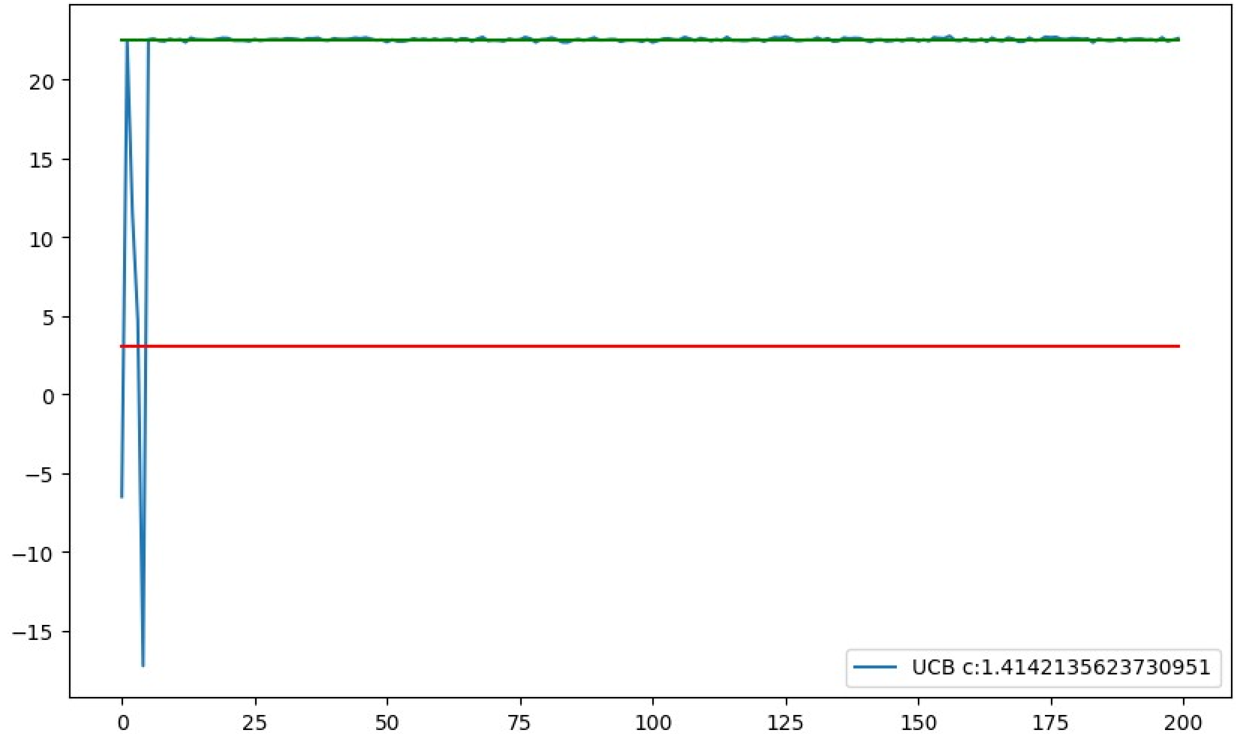


Inferences

- We observe that τ value of 5.0 results in the least regret among the cases that have been tried.
- We also observe that for τ values of 1.0 and 5.0, the policy might converge to the optimal arm over a longer period of time. But, τ values of 0.001 and 50.0 might not converge to the optimal arm.
- We observe that the ϵ-greedy policy performs better than softmax policy for this problem with the given values of hyperparameters.

```
plot_reward_curve_and_print_regret(env, [UCB(env.arm_ids,
np.sqrt(2))], timesteps=200, num_runs=500)

regret for UCB c:1.4142135623730951: 95.406
```



Inferences

Note: These inferences are for c = $\sqrt{2}$.

- The UCB algorithm converges to the optimal arm in around 10-20 timesteps.
- The intial regret occurs due to the fact that every arm has to be played at least once in order to initialize the Q values for each arm.
- This policy performs better when compared to softmax policy.

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

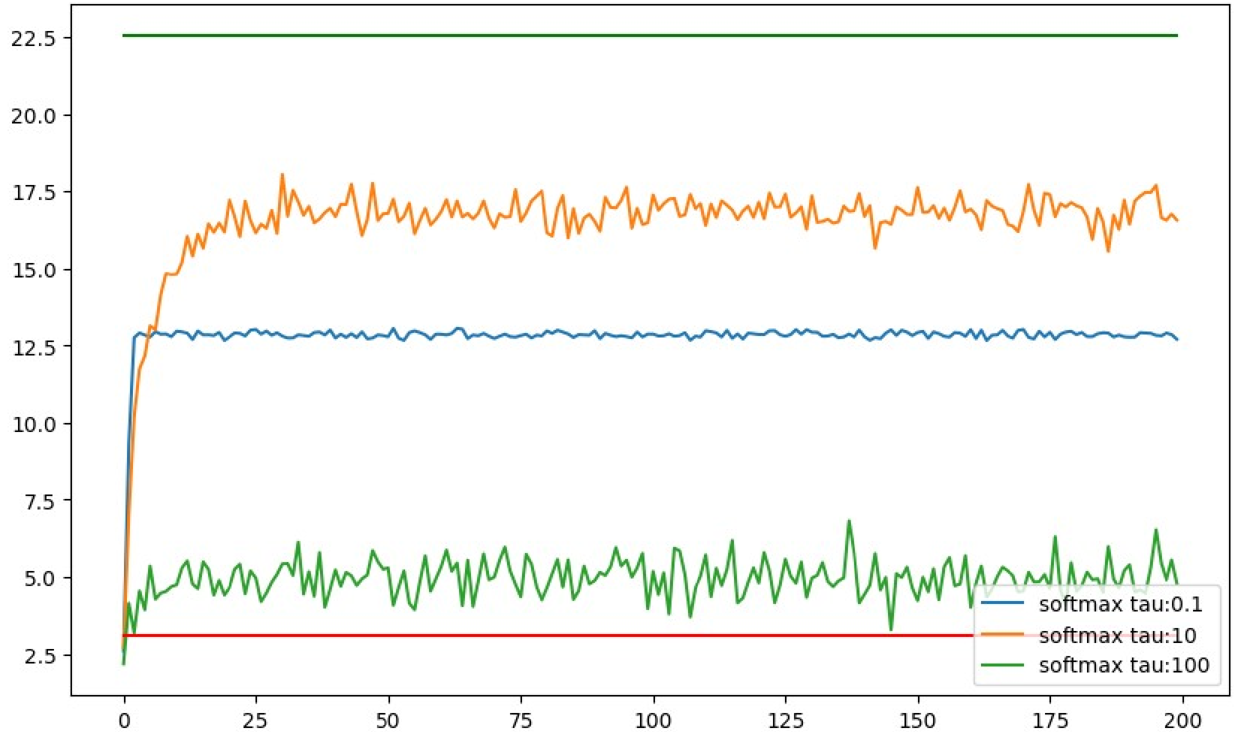
```
explore_epgreedy_epsilons =  [0.1, 1, 5, 10]
epgreedy_policies = [EpGreedyPolicy(ep, env.arm_ids) for ep in
explore_epgreedy_epsilons]
plot_reward_curve_and_print_regret(env, epgreedy_policies,
timesteps=200, num_runs=500)
```

```
regret for ep-greedy ep:0.1: 455.951
regret for ep-greedy ep:1: 3872.485
regret for ep-greedy ep:5: 3879.887
regret for ep-greedy ep:10: 3882.263
```

We can observe for ϵ values greater that 1, the policy does not converge to the optimal arm (similar to random policy).

```
explore_softmax_taus =  [0.1, 10, 100]
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)
```

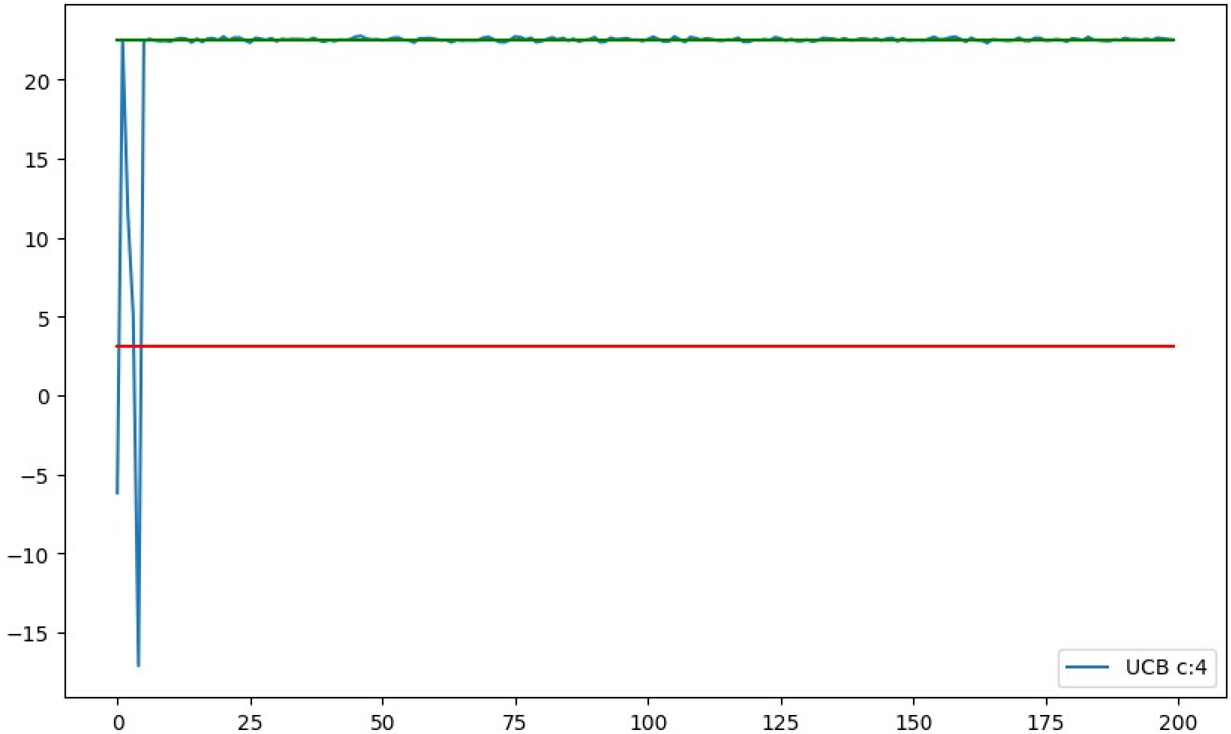
```
regret for softmax tau:0.1: 1950.782
regret for softmax tau:10: 1205.205
regret for softmax tau:100: 3517.764
```



Chossing τ values is very crucial for this problem. There is a good chance the policy does not converge to the optimal policy.

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

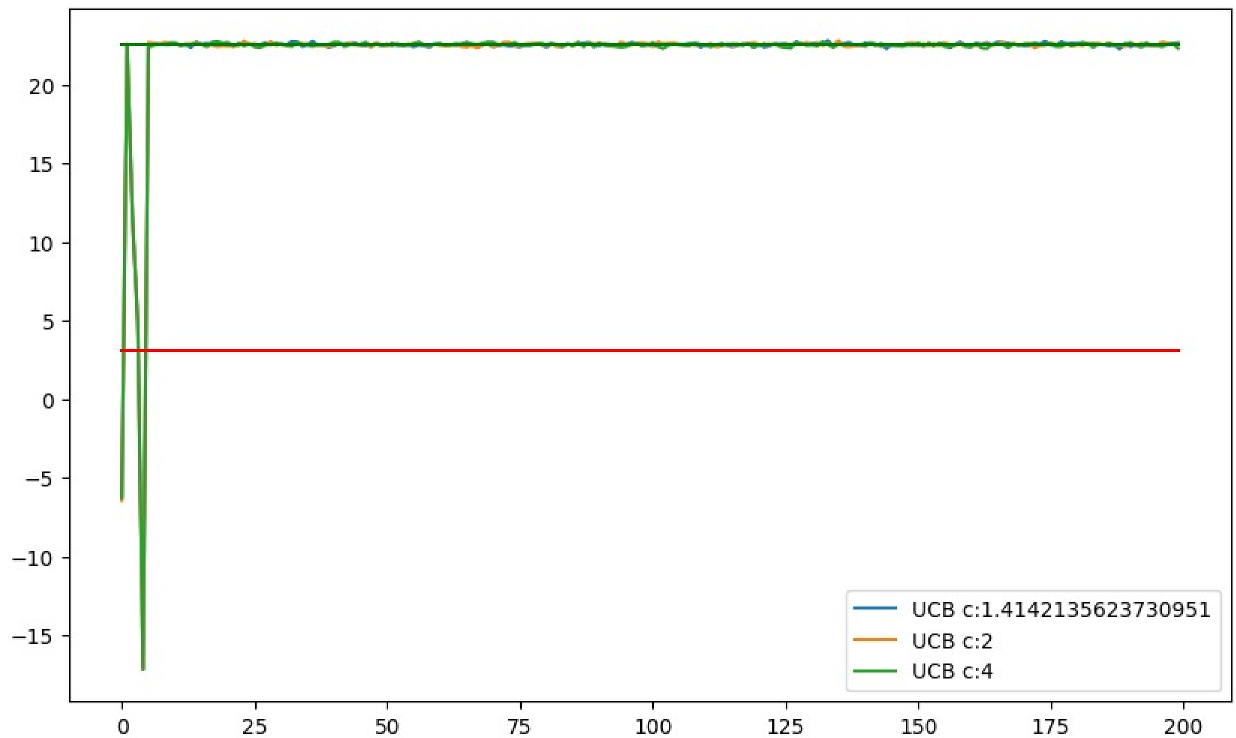
```
regret for UCB c:4: 96.911
```



This is the UCB policy output for c = 4.

```
plot_reward_curve_and_print_regret(env, [UCB(env.arm_ids, np.sqrt(2)),
UCB(env.arm_ids, 2), UCB(env.arm_ids, 4)], timesteps=200,
num_runs=500)
```

```
regret for UCB c:1.4142135623730951: 95.395
regret for UCB c:2: 95.783
regret for UCB c:4: 99.050
```



We observe almost equal levels of performance for the values of c that have been tried.

Note: If we consider the initializing of Q values outside the number of timesteps, then the UCB algorithm performs the best among all the tried out policies (total-regret aroung 26.0). Otherwise the ϵ-greedy algortihm with ϵ values 0.001 and 0.01 performs the best.