CS6700: Reinforcement Learning

Tutorial 1: Bandits

Gautham Govind A, EE19B022

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from IPython.display import display, HTML
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:
    # your code for updating the Q values of each arm

    # updating q values using the update rule
    self.Q[arm_id] = self.Q[arm_id] +
(1/(self.num_pulls_per_arm[arm_id] + 1))*(arm_reward - self.Q[arm_id])

    # incrementing the number of pulls for the pulled arm
    self.num_pulls_per_arm[arm_id] += 1

  def select_arm(self) -> int:
    # your code for selecting arm based on epsilon greedy policy

    # explore v/s exploit decision made by a Bernoulli r.v.
    # exploit done with probability (1 - epsilon) (exploit corresponds
to a value of 1 for the r.v.)
    # explore done with probability epsilon (explore corresponds to a
value of 0 for the r.v.)
```

```python
    explore_or_exploit = np.random.binomial(1, 1 - self.epsilon)

    # if exploit, select the arm with maximum value of value function
    if explore_or_exploit == 1:
        return max(self.Q, key = lambda x: self.Q[x])

    # if explore, choose any one of the arms randomly
    else:
        return np.random.choice(self.arm_ids)

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

    # updating q values using the update rule
    self.Q[arm_id] = self.Q[arm_id] +
(1/(self.num_pulls_per_arm[arm_id] + 1))*(arm_reward - self.Q[arm_id])

    # incrementing the number of pulls for the pulled arm
    self.num_pulls_per_arm[arm_id] += 1

  def select_arm(self) -> int:
    # your code for selecting arm based on softmax policy

    # implementing the expression for softmax as it is can result in
numerical overflow
    # to improve the numerical stability, the numerator and
denominator are divided by the largest value

    # finding the largest q value
    max_val = max(list(self.Q.values()))

    # computing the denominator of the softmax expression
    # note that each term is divided by the largest exponential
    softmax_prob_denom =
np.sum(np.exp((np.array(list(self.Q.values())) - max_val)/self.tau))
```

```
    # computing the numerator term for each arm
    # again, terms are divided by the largest exponential to maintatin
consistency with denominator
    softmax_prob = np.array([ np.exp((x - max_val)
/self.tau)/softmax_prob_denom for x in list(self.Q.values())])

    # sample according to the softmax distribution to get an arm
    return np.random.choice(self.arm_ids, p = softmax_prob)

class UCB(BasePolicy):
  # your code here

  def __init__(self, arm_ids):
        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}

        # keeping track of total number of pulls as it is required
        self.total_pulls = 0

  @property
  def name(self):
      return f'ucb'

  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}

      # total number of pulls should also be reset after each run
      self.total_pulls = 0

  def update_arm(self, arm_id: int, arm_reward: float) -> None:

      # updating q values using the update rule
      self.Q[arm_id] = self.Q[arm_id] +
(1/(self.num_pulls_per_arm[arm_id] + 1))*(arm_reward - self.Q[arm_id])

      # incrementing the number of pulls for the pulled arm
      self.num_pulls_per_arm[arm_id] += 1

      # incrementing the total number of pulls
      self.total_pulls += 1


  def select_arm(self) -> int:

      # we are required to pull each arm once as initialization
      # this is achieved by the following conditional statement
```

```
        if self.total_pulls < len(self.arm_ids):
            return self.arm_ids[self.total_pulls]

        # initializating maximum upper bound as negative infinitiy
        max_val = -np.inf

        # looping through each arm
        for key, val in self.Q.items():

            # computing the upper bound for each arm
            upper_bound = val + np.sqrt(2*np.log(self.total_pulls) /
self.num_pulls_per_arm[key])

            # checking if upper bound for this arm exceeds the highest
so far
            # if so, updating
            if upper_bound > max_val:
                max_val = upper_bound
                max_arm = key

        # select the arm with the highest value upper bound
        return max_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

  # calculate avg policy reward from policy_reward_each_run

  # rewards averaged over runs for each time step
  avg_policy_rewards = np.mean(policy_reward_each_run, axis = 0) #
your code here (type: nd.array, shape: (timesteps,))

  # regret calculated by summing up the difference between best
possible reward and average policy reward
```

```
  total_policy_regret = np.sum(expected_max_reward -
avg_policy_rewards) # 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()
```

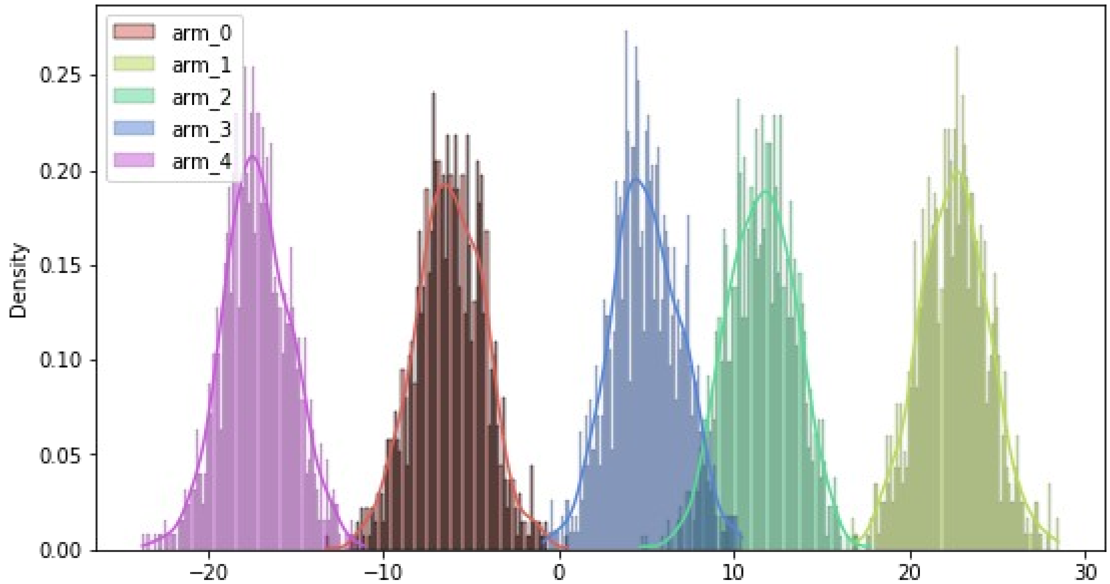
Experiments

```
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
```

```
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
```

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

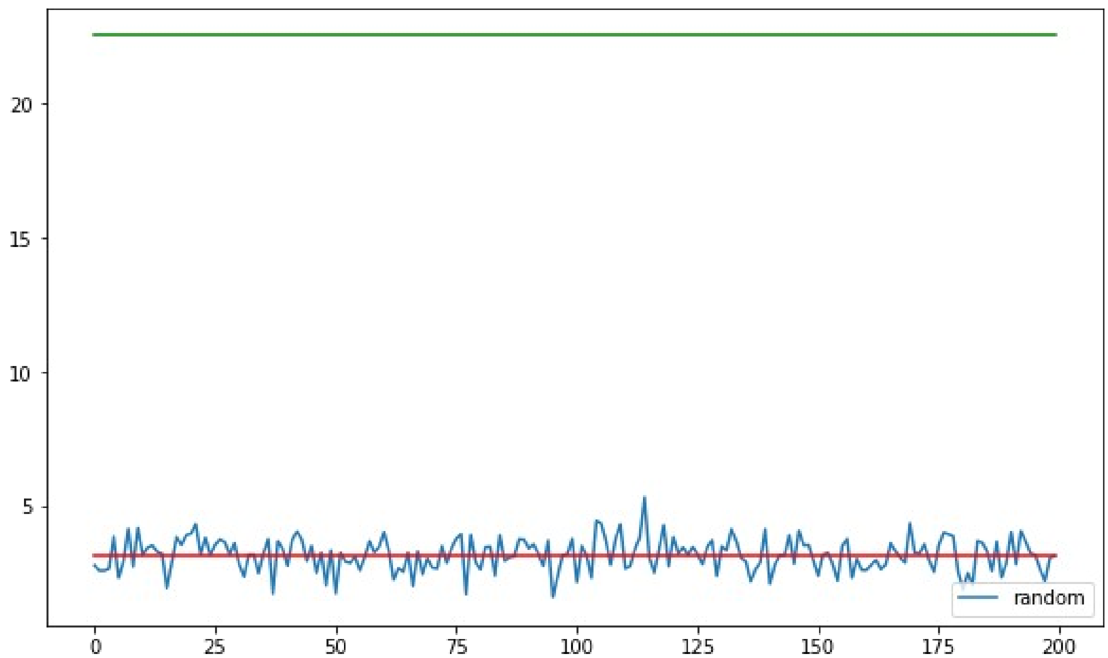
```
3.119254917081568
```

Please explore following values:

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

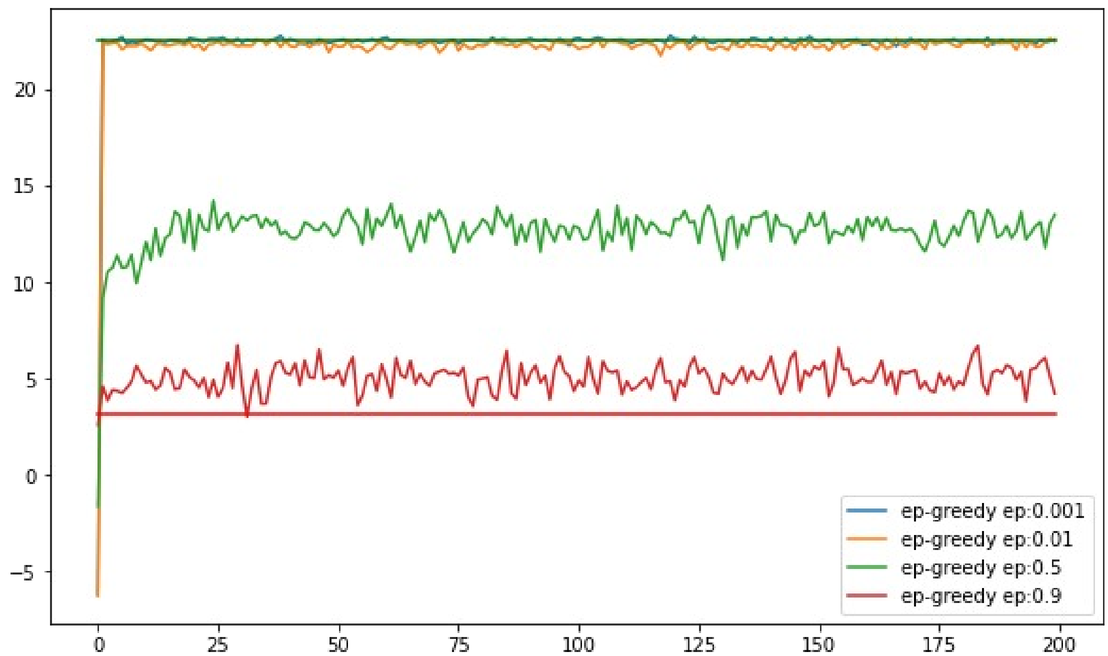
```
random_policy = RandomPolicy(env.arm_ids)
plot_reward_curve_and_print_regret(env, [random_policy],
timesteps=200, num_runs=500)
```

```
regret for random: 3871.625
```



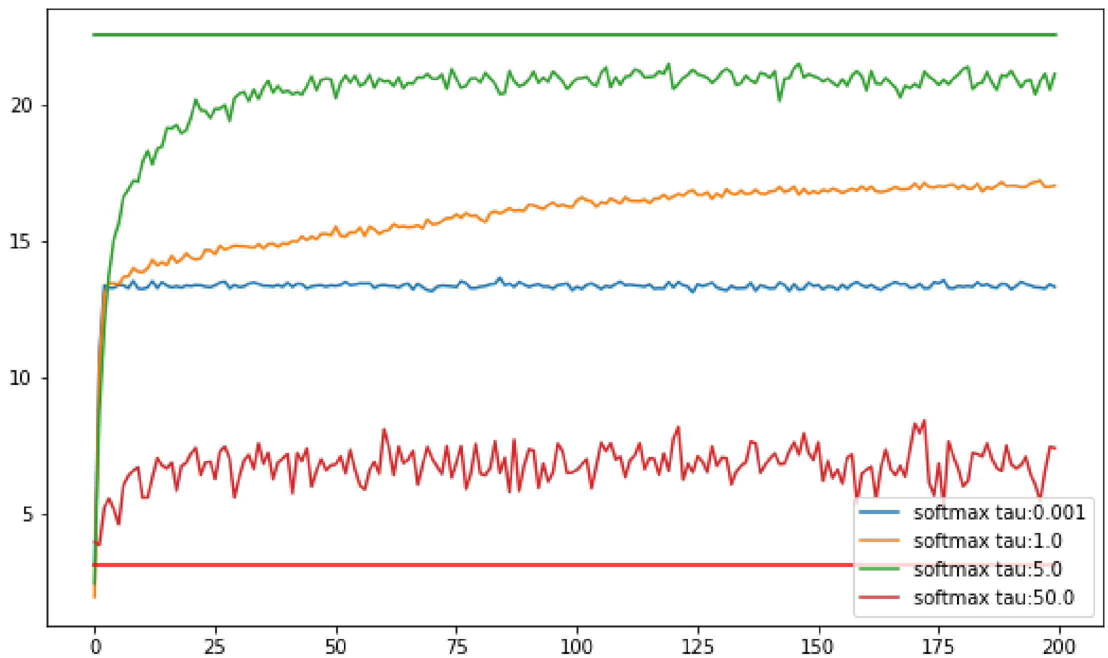

```
explore_epgreedy_epsilons =  [0.001, 0.01, 0.5, 0.9]
epgreedy_policies = [EpGreedyPolicy(ep, env.arm_ids) for ep in
explore_epgreedy_epsilons]
plot_reward_curve_and_print_regret(env, epgreedy_policies,
timesteps=200, num_runs=500)
```

```
regret for ep-greedy ep:0.001: 33.465
regret for ep-greedy ep:0.01: 76.513
regret for ep-greedy ep:0.5: 1979.988
regret for ep-greedy ep:0.9: 3505.260
```



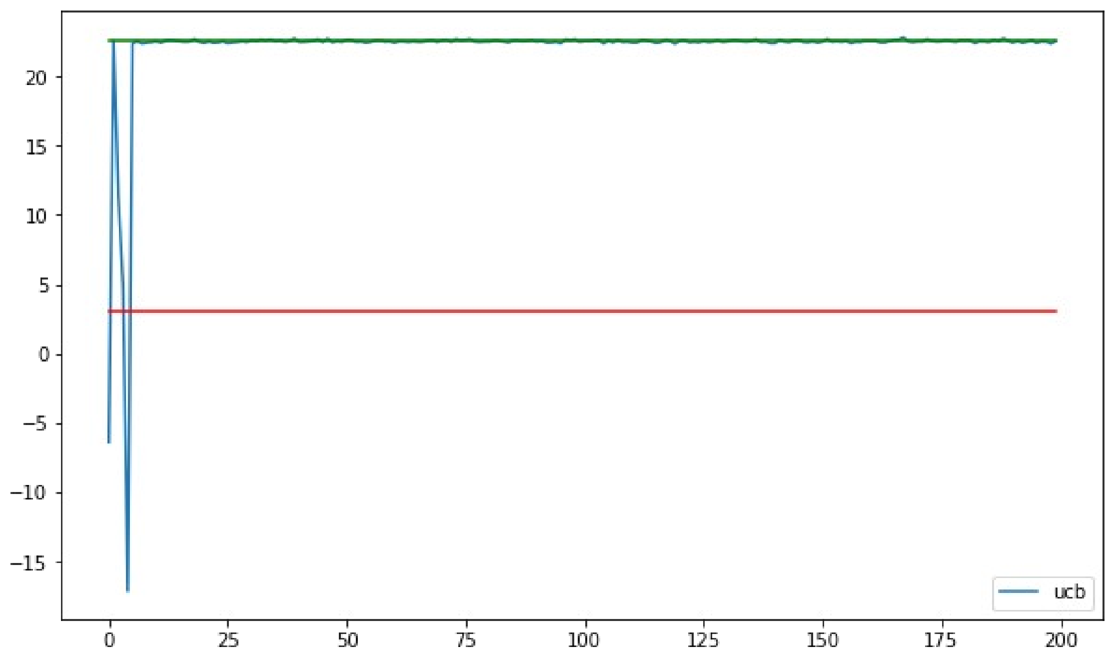

```
explore_softmax_taus =  [0.001, 1.0, 5.0, 50.0]
softmax_polices = [SoftmaxPolicy(tau, env.arm_ids) for tau in
explore_softmax_taus]
plot_reward_curve_and_print_regret(env, softmax_polices,
timesteps=200, num_runs=500)
```

```
regret for softmax tau:0.001: 1851.312
regret for softmax tau:1.0: 1328.822
regret for softmax tau:5.0: 438.070
regret for softmax tau:50.0: 3160.561
```




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

```
regret for ucb: 97.238
```



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