me19b190

February 2, 2023

[1]:
```python
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
```

0.0.1 Gaussian Bandit Environment

[2]:
```python
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()
```

0.0.2 Policy

[3]:
```
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

[4]:
```
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)
```

[18]:
```
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
    sum_reward  = self.Q[arm_id]*self.num_pulls_per_arm[arm_id]
    sum_reward = sum_reward+arm_reward
    self.num_pulls_per_arm[arm_id] +=1
    self.Q[arm_id] = (sum_reward)/(self.num_pulls_per_arm[arm_id])
```

```
  def select_arm(self) -> int:
    # your code for selecting arm based on epsilon greedy policy
    arm_maxQ =  list(self.Q.keys())[list(self.Q.values()).index(max(self.Q.
↪values()))]
    probabilities = {id : 1-self.epsilon+((self.epsilon)/(len(self.arm_ids)))␣
↪if id == arm_maxQ else (self.epsilon)/(len(self.arm_ids)) for id in self.
↪arm_ids}
    arm_selected = np.random.choice(self.arm_ids, 1, p = list(probabilities.
↪values()))
    return int(arm_selected)
```

[6]:

```
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
    sum_reward  = self.Q[arm_id]*self.num_pulls_per_arm[arm_id]
    sum_reward = sum_reward+arm_reward
    self.num_pulls_per_arm[arm_id] +=1
    self.Q[arm_id] = (sum_reward)/(self.num_pulls_per_arm[arm_id])


  def select_arm(self) -> int:
    # your code for selecting arm based on softmax policy
    arr = np.array(list(self.Q.values()))/self.tau
    arr = np.exp(arr)
    arr = [np.exp(709) if i == np.inf else i for i in arr]
    sum_arr = sum(arr)
    probabilities = arr/sum_arr
    arm_selected = np.random.choice(self.arm_ids, 1, p = probabilities)
    return int(arm_selected)
```

[30]:

```
class UCB(BasePolicy):
  def __init__(self, env,arm_ids):
    self.arm_ids = arm_ids
```

```python
    self.Q = {id: 0 for id in self.arm_ids}
    self.num_pulls_per_arm = {id: 0 for id in self.arm_ids}

    for arm_id in self.arm_ids:
      self.Q[arm_id] = env.step(arm_id)
      self.num_pulls_per_arm[arm_id] +=1

  @property
  def name(self):
    return '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}


  def update_arm(self, arm_id: int, arm_reward: float) -> None:
    # your code for updating the Q values of each arm

    sum_reward  = self.Q[arm_id]*self.num_pulls_per_arm[arm_id]
    sum_reward = sum_reward+arm_reward
    self.num_pulls_per_arm[arm_id] +=1
    self.Q[arm_id] = (sum_reward)/(self.num_pulls_per_arm[arm_id])


  def select_arm(self) -> int:
    # your code for selecting arm based on softmax policy
    for arm_id in self.arm_ids:
      if self.num_pulls_per_arm[arm_id] == 0:
        return arm_id

    ucb = np.array([self.Q[arm_id]+((2*np.log(len(self.arm_ids))/self.
↪num_pulls_per_arm[arm_id])**0.5) for arm_id in self.arm_ids])
    arm_selected = list(self.Q.keys())[list(ucb).index(max(ucb))]
    return int(arm_selected)
```

Trainer

[8]:
```python
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
  avg_policy_rewards = np.array([policy_reward_each_run[:,i].mean() for i in␣
 ↪range(timesteps)]) # your code here (type: nd.array, shape: (timesteps,))

  total_policy_regret =  sum(expected_max_reward - avg_policy_rewards) # your␣
 ↪code here (type: float)

  return avg_policy_rewards, total_policy_regret
```

[9]:
```
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()
```

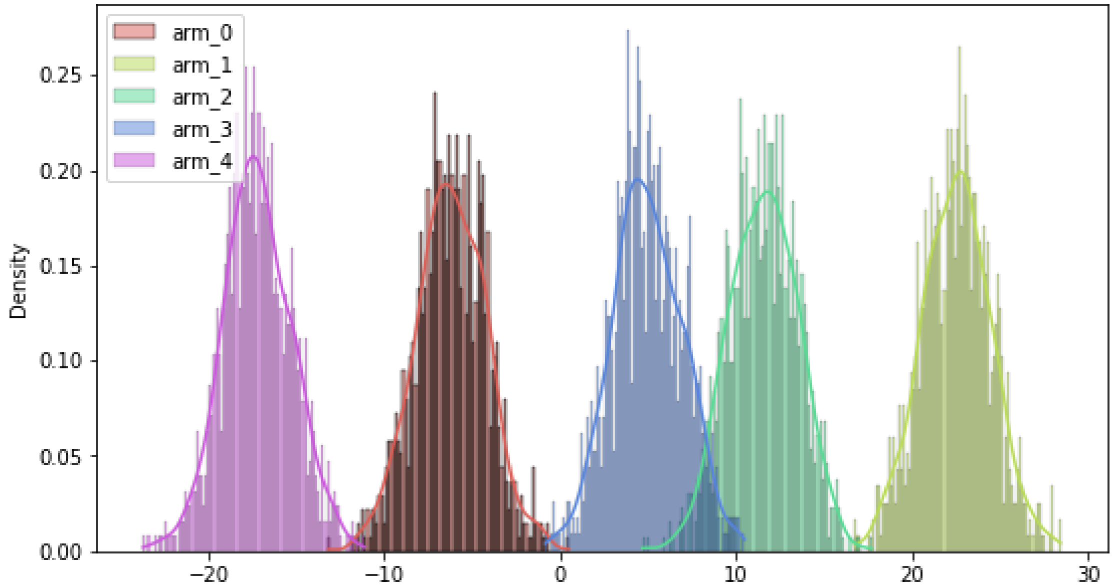
0.0.3 Experiments

[10]:
```
seed = 42
np.random.seed(seed)

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

[11]:
```
env = Env(num_arms, mean_reward_range, std)

env.plot_arms_reward_distribution()
```



[12]:
```
best_arm, max_mean_reward = env.get_best_arm_and_expected_reward()
print(best_arm, max_mean_reward)
```

```
1 22.53571532049581
```

[13]:
```
print(env.get_avg_arm_reward())
```

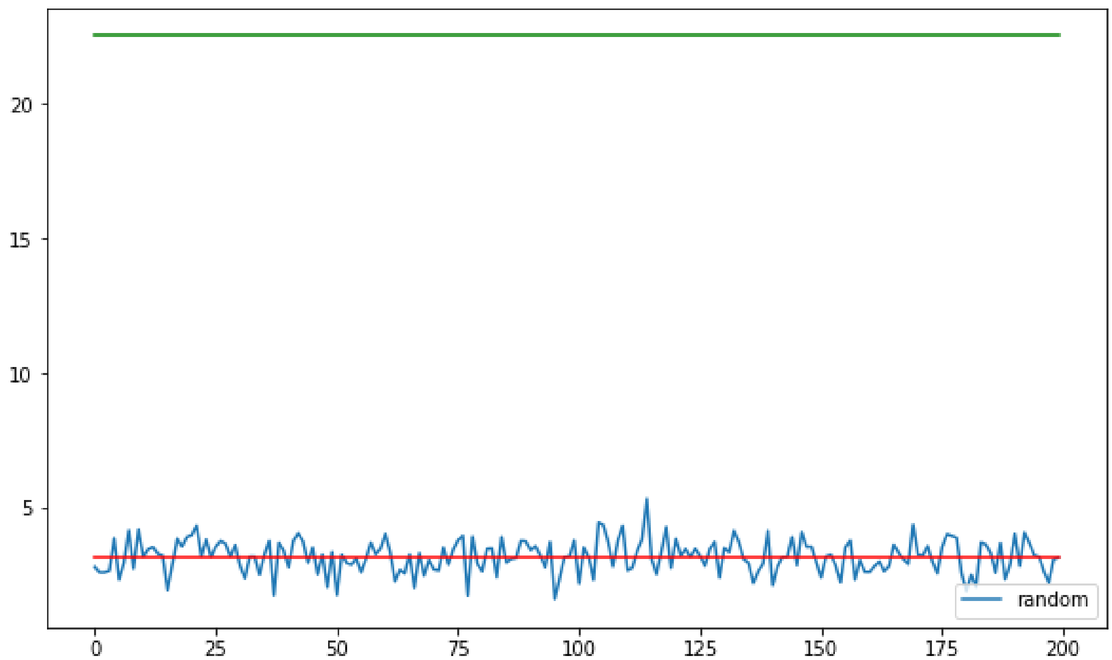
```
3.119254917081568
```

Please explore following values:

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

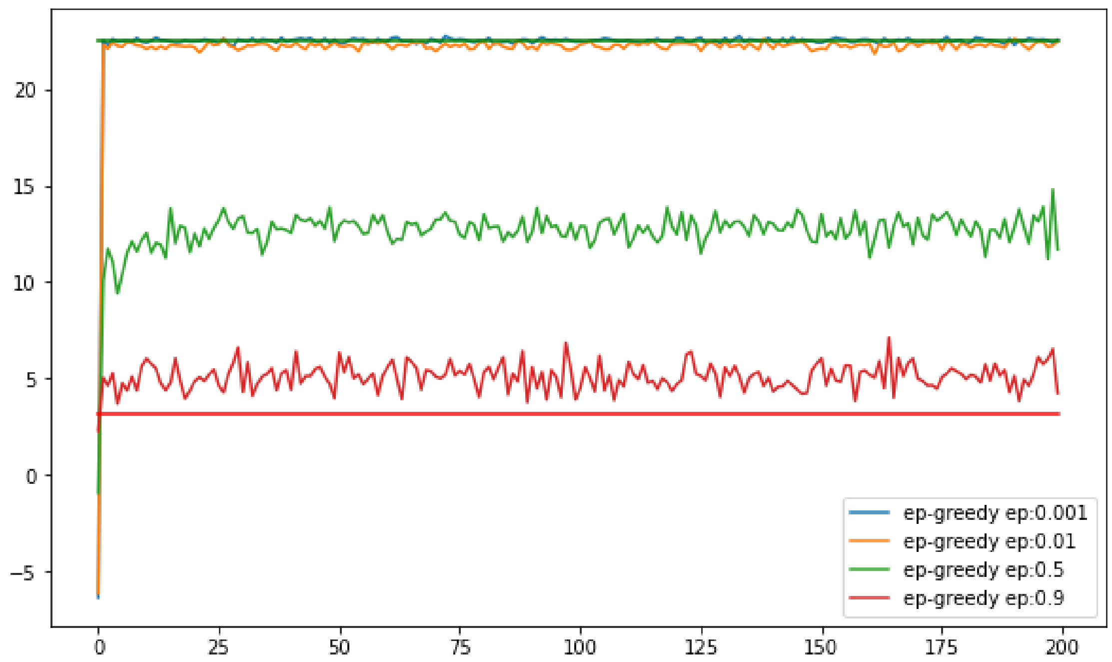
[14]:
```
random_policy = RandomPolicy(env.arm_ids)
plot_reward_curve_and_print_regret(env, [random_policy], timesteps=200,␣
  ↪num_runs=500)
```

```
regret for random: 3871.625
```



[19]:
```
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)
```

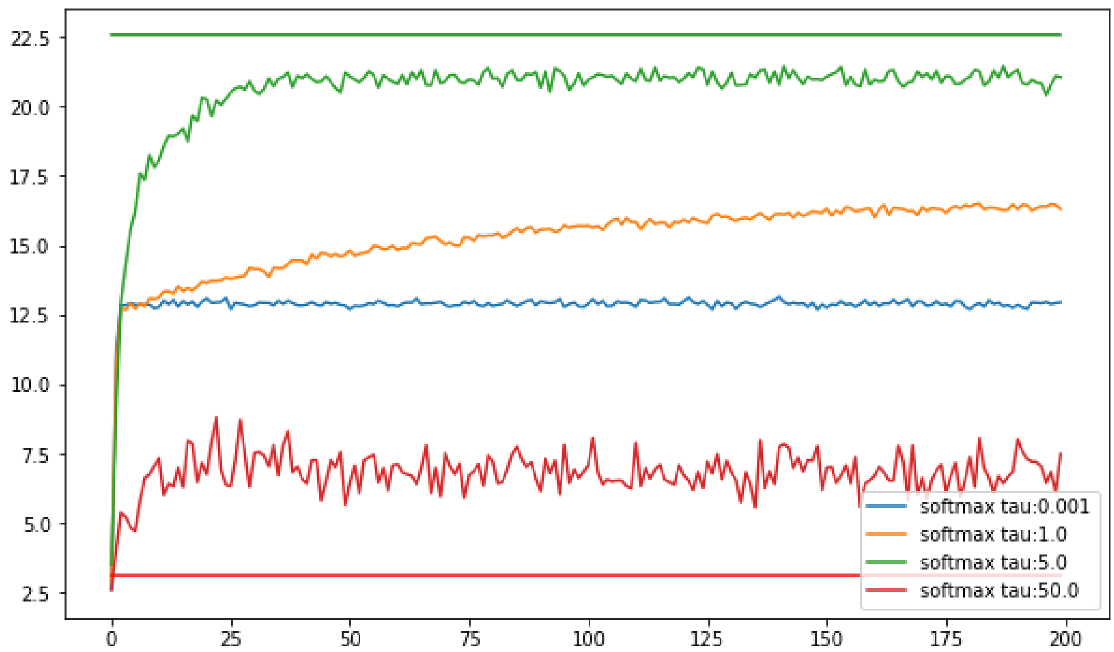
```
regret for ep-greedy ep:0.001: 33.902
regret for ep-greedy ep:0.01: 84.852
regret for ep-greedy ep:0.5: 1981.669
regret for ep-greedy ep:0.9: 3491.759
```



[16]:
```
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)
```

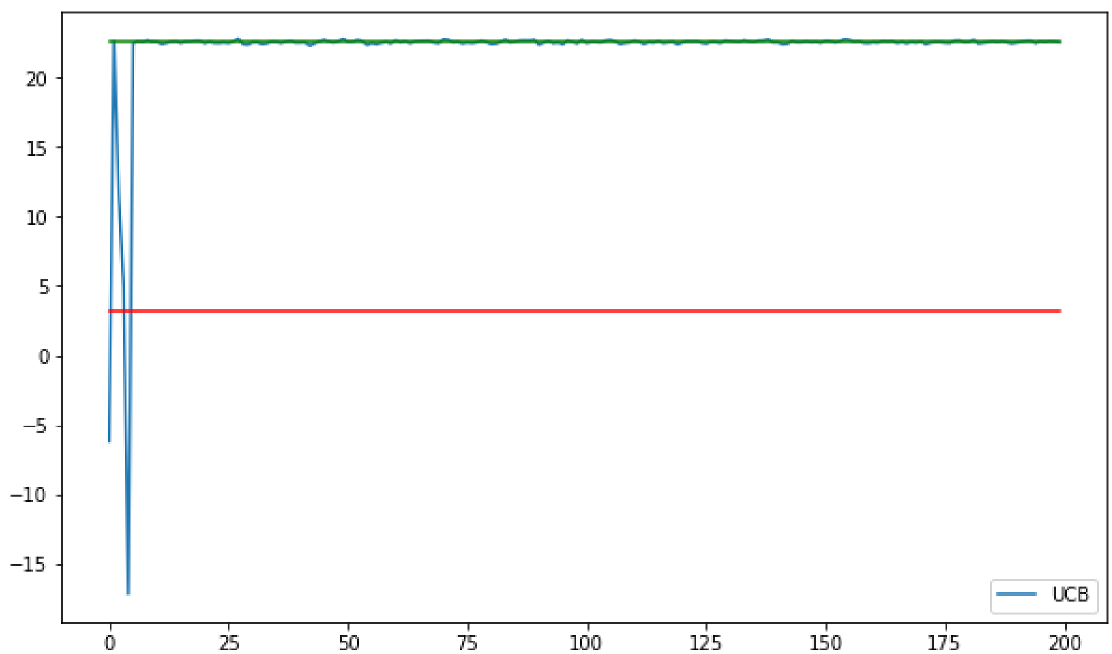
```
<ipython-input-6-e56e94cd8809>:27: RuntimeWarning: overflow encountered in exp
  arr = np.exp(arr)
```

```
regret for softmax tau:0.001: 1940.060
regret for softmax tau:1.0: 1457.339
regret for softmax tau:5.0: 400.901
regret for softmax tau:50.0: 3145.762
```



[31]:
```
plot_reward_curve_and_print_regret(env, [UCB(env,env.arm_ids)], timesteps=200,
    num_runs=500)
```

```
regret for UCB: 96.566
```



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