6.8200 HW1_Assignment_on_Bandits

February 21, 2023

1 Spring 2023 6.800 Computational Sensorimotor Learning Assignment 1

In this assignment, we will learn about multi-armed and contextual bandits. You will need to answer the bolded questions and fill in the missing code snippets (marked by **TODO**).

Make a copy of this notebook using File > Save a copy in Drive and edit it with your answers.

IMPORTANT: Set your runtime to GPU, otherwise the checkers may fail.

There are 165 total points in this assignment, scaled to be worth 6.25% of your final grade.

1.0.1 Setup

Ignore the following skeleton code (imports, plotting).

[1]:
```
!pip install -i https://test.pypi.org/simple/ sensorimotor-checker==0.0.7
```

```
Looking in indexes: https://test.pypi.org/simple/, https://us-
python.pkg.dev/colab-wheels/public/simple/
Collecting sensorimotor-checker==0.0.7
  Downloading https://test-files.pythonhosted.org/packages/d3/bc/80b85ae047c79aa
d1cad3ff8c029f42a87f0f4d28602cc6ab809a7538fb7/sensorimotor_checker-0.0.7-py3-
none-any.whl (3.4 kB)
Installing collected packages: sensorimotor-checker
Successfully installed sensorimotor-checker-0.0.7
```

[2]:
```
%matplotlib inline
import numpy as np
import random
import time
import os
import gym
import json
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
import pandas as pd

import unittest
```

```
from copy import deepcopy
from tqdm.notebook import tqdm
from dataclasses import dataclass
from typing import Any
mpl.rcParams['figure.dpi']= 100

from sensorimotor_checker import hw1_tests
```

[3]:
```
# some util functions
def plot(logs, x_key, y_key, legend_key, **kwargs):
    nums = len(logs[legend_key].unique())
    palette = sns.color_palette("hls", nums)
    if 'palette' not in kwargs:
        kwargs['palette'] = palette
    ax = sns.lineplot(x=x_key, y=y_key, data=logs, hue=legend_key, **kwargs)
    return ax

def set_random_seed(seed):
    np.random.seed(seed)
    random.seed(seed)

# set random seed
seed = 0
set_random_seed(seed=seed)
```

2 Multi-armed bandits

Let us define a multi-armed bandit scenario with 10 arms. There are two slightly different formulations that are useful:

- Stochastic Case: Each arm has a reward of 1, with probability $p \in [0, 1]$.
- Deterministic Case: Each arm has a reward $r \in [0, 1]$, but the same reward is obtained for every pull.

In this assignment, we will work through the stochastic case. The same insights would apply to the deterministic scenario with variable rewards or even to stochastic setups with variable rewards.

To define our bandit, we arbitrarily select probabilities p for each arm and save them as `probs`.

[4]:
```
numArms = 10
probs = [np.random.random() for i in range(numArms)]
print(probs)
```

```
[0.5488135039273248, 0.7151893663724195, 0.6027633760716439, 0.5448831829968969,
0.4236547993389047, 0.6458941130666561, 0.4375872112626925, 0.8917730007820798,
0.9636627605010293, 0.3834415188257777]
```

We then define an environment to evaluate different agent strategies.

[5]:
```
#To simulate a realistic Bandit scenario, we will make use of the BanditEnv.
@dataclass
class BanditEnv:
    probs: np.ndarray # probabilities of giving positive reward for each arm

    def step(self, action):
        # Pull arm and get stochastic reward (1 for success, 0 for failure)
        return 1 if (np.random.random()  < self.probs[action]) else 0
```

[6]:
```
#Code for running the bandit environment.
@dataclass
class BanditEngine:
    probs: np.ndarray
    max_steps: int
    agent: Any

    def __post_init__(self):
        self.env = BanditEnv(probs=self.probs)

    def run(self, n_runs=1):
        log = []
        for i in tqdm(range(n_runs), desc='Runs'):
            run_rewards = []
            run_actions = []
            self.agent.reset()
            for t in range(self.max_steps):
                action = self.agent.get_action()
                reward = self.env.step(action)
                self.agent.update_Q(action, reward)
                run_actions.append(action)
                run_rewards.append(reward)
            data = {'reward': run_rewards,
                    'action': run_actions,
                    'step': np.arange(len(run_rewards))}
            if hasattr(self.agent, 'epsilon'):
                data['epsilon'] = self.agent.epsilon
            run_log = pd.DataFrame(data)
            log.append(run_log)
        return log
```

[7]:
```
#Code for aggregrating results of running an agent in the bandit environment.
def bandit_sweep(agents, probs, labels, n_runs=2000, max_steps=500):
    logs = dict()
    pbar = tqdm(agents)
    for idx, agent in enumerate(pbar):
        pbar.set_description(f'Alg: {labels[idx]}')
        engine = BanditEngine(probs=probs, max_steps=max_steps, agent=agent)
        ep_log = engine.run(n_runs)
        ep_log = pd.concat(ep_log, ignore_index=True)
        ep_log['Alg'] = labels[idx]
        logs[f'{labels[idx]}'] = ep_log
    logs = pd.concat(logs, ignore_index=True)
    return logs
```

Credits: The code for Multi-Arm Bandits is inspired from

- https://github.com/ShangtongZhang/reinforcement-learning-an-introduction/blob/master/chapter02/ten_armed_testbed.py
- https://github.com/lilianweng/multi-armed-bandit/blob/master/solvers.py

2.1 Oracle Agent

The best agent we could possibly build is one that has access to all the necessary information to make an optimal decision, even if that information would not be available in a real world problem. We call this an "oracle agent."

Imagine you were to build an Oracle agent for the stochastic multi-armed bandits problem defined by `probs`. What reward would you get from this agent in expectation?

[8]:
```
#### TODO: find the maximum return with priviledged information about the␣
 ↪reward distribution [5pts] ####
# Since we get reward 1 for each of the successfull pulls and we succeed with␣
 ↪probability probs:
priveleged = max(probs)
print(priveleged)
oracle_reward = priveleged
###################################
suite = unittest.TestSuite()
suite.addTest(hw1_tests.TestOracleAgent('check_reward', oracle_reward))
unittest.TextTestRunner(verbosity=0).run(suite)
```

```
0.9636627605010293

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

[8]:
```
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

2.2 Random Agent

That's pretty high reward! However, let's say that we don't have access to `probs`, and that the only information we can learn about the environment is through interaction. This is more akin to a real world bandits problem.

One baseline agent we should construct is one that chooses a random action at every timestep. Fill in the `TODO` in the below agent code to implement this behavior.

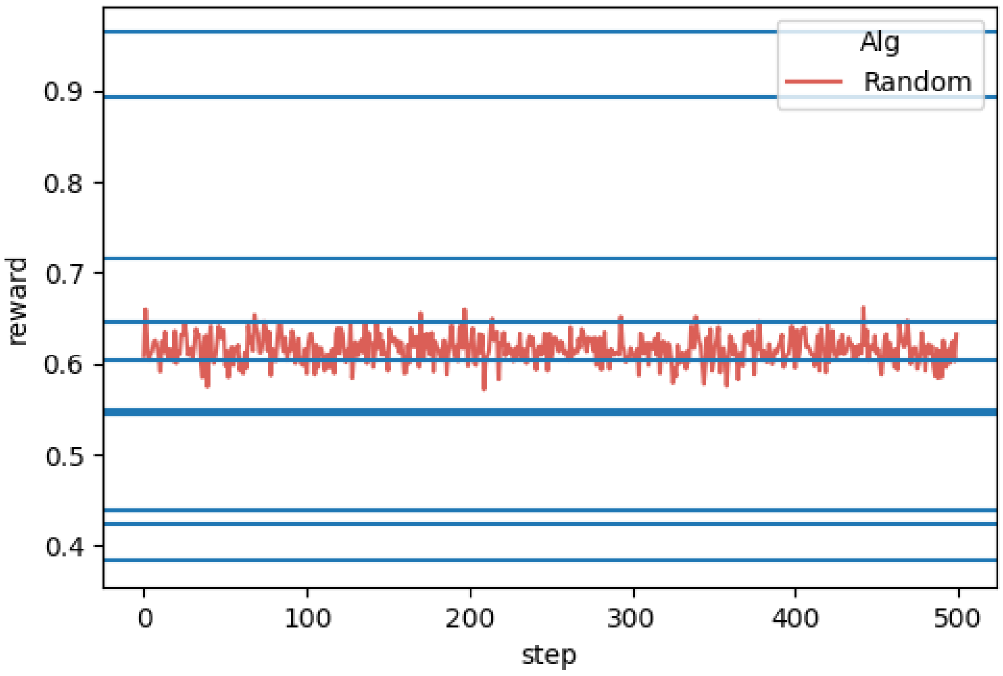
```
[9]: # As a baseline, lets first construct a baseline agent that chooses a random␣
       ↪action at every timestep.
     # We will measure how much better we can do.
     @dataclass
     class RandomAgent:
         num_actions: int

         def __post_init__(self):
             self.reset()

         def reset(self):
             self.t = 0
             self.action_counts = np.zeros(self.num_actions, dtype=int) # action␣
       ↪counts n(a)
             self.Q = np.zeros(self.num_actions, dtype=float) # action value Q(a)

         def update_Q(self, action, reward):
             pass

         def get_action(self):
             self.t += 1
             #### TODO: get a random action index [5pts]####
             selected_action = np.random.randint(self.num_actions)
             ##################################

             return selected_action
```

```
[10]: #Create the random agent.
      agent = RandomAgent(num_actions=len(probs))
      '''
      In order to measure average behavior of the agent, we are going to run the agent
      multiple times and compute the mean reward. The number of runs will be denoted
      by the variable `n_runs`. The default value is set to 1000, but feel free to␣
        ↪reduce it
      it if its taking too much time.
      '''
      n_runs = 1000
      logs = bandit_sweep([agent], probs, ['Random'], n_runs=n_runs)
```

```
  0%|          | 0/1 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]
```

```
[11]: #### TODO: plot the reward curve of a random agent, and the average reward of␣
        ↪all arms [5pts]####
```

```
ax = plot(logs, x_key='step', y_key='reward', legend_key='Alg',␣
↪estimator='mean', ci=None)
[plt.axhline(y=prob) for prob in probs];
################################
```



[12]:
```
suite = unittest.TestSuite()
suite.addTest(hw1_tests.TestRandomAgent('check_performance', logs))
unittest.TextTestRunner(verbosity=0).run(suite)
```

```
----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```

[12]:
```
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

Analyzing the Results: - On the x-axis is the number of steps taken by the agent. - On the y-axis is the average reward after i steps.

The reward obtained by the random agent is far less that the oracle agent. Regret is defined as the difference between the reward collected by oracle and the agent under consideration. In the above example, regret is about 0.35.

Note: that if you use a different random seed to run experiments, you might get a slighly different value of regret. Treat this as a ball park figure.

2.3 Explore First Agent

In the class we discussed an algorithm to solve bandits where, - For the first N (defined as `max_explore` in the code) steps the agent takes random actions. - The agent identifies the best arm based on these N steps and then only chooses the best arm.

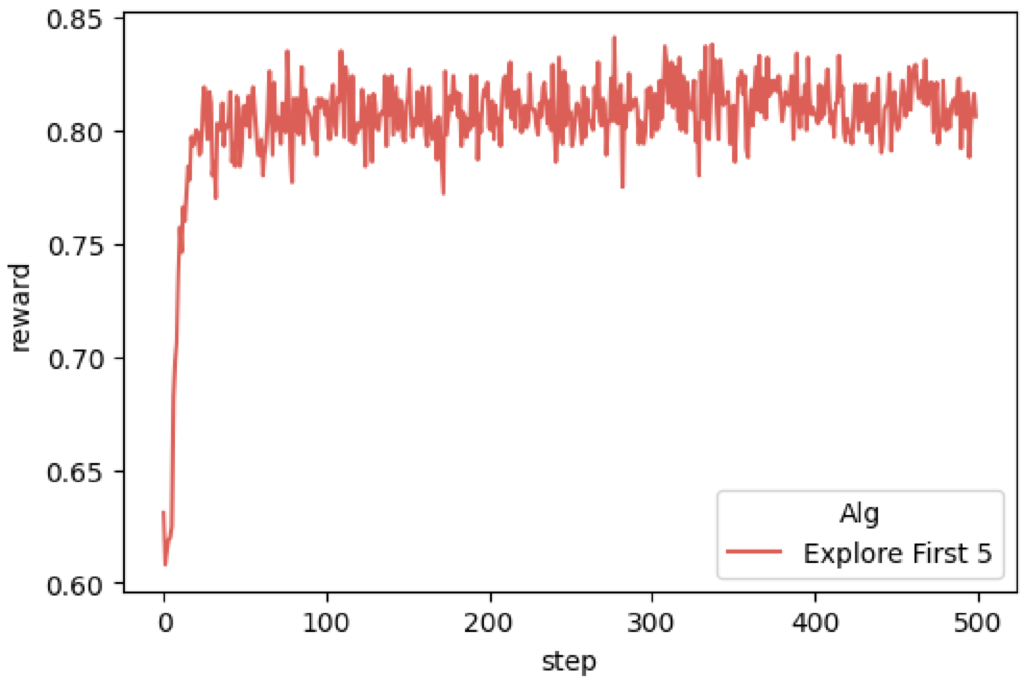
We will now implement this agent below. Fill in the missing code in `update_Q` and `get_action`. We will store the average reward for each action in the variable `self.Q`, and the count of how many times we've taken each action in `self.action_counts`.

[13]:
```
#Lets now construct the explore first agent
@dataclass
class ExploreFirstAgent:
    num_actions: int
    max_explore: int

    def __post_init__(self):
        self.reset()

    def reset(self):
        self.t = 0
        self.action_counts = np.zeros(self.num_actions, dtype=int) # action␣
 ↪counts n(a)
        self.Q = np.zeros(self.num_actions, dtype=float) # action value Q(a)

    def update_Q(self, action, reward):
        # Update Q action-value given (action, reward)
        # HINT: Keep track of how good each arm is
        #### TODO: update Q value [5pts] ####
        self.action_counts[action] += 1
        alpha = 1 / self.action_counts[action]
        self.Q[action] += alpha * (reward - self.Q[action])
        ##################################

    def get_action(self):
        self.t += 1
        #### TODO: get action [5pts] ####
        if self.t <= self.max_explore:
            selected_action = np.random.randint(self.num_actions)
        else:
            selected_action = np.argmax(self.Q)
        ##################################

        return selected_action
```

Great! Now we'll instantiate the engine, and run it with $N = 5$ (five steps of exploration, followed by entirely greedy policy).

```
[14]: max_explore = 5
      agent = ExploreFirstAgent(num_actions=len(probs), max_explore=max_explore)
      logs = bandit_sweep([agent], probs, ['Explore First 5'], n_runs=1000,␣
       ↪max_steps=500)
      plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean',␣
       ↪ci=None);
```

```
  0%|          | 0/1 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]
```



Check your work: If you pass update_Q but fail in performance, check your get action and ensure that you're on GPU runtime.

```
[15]: suite = unittest.TestSuite()
      suite.addTest(hw1_tests.TestExploreFirstAgent('check_update_Q',␣
       ↪ExploreFirstAgent))
      suite.addTest(hw1_tests.TestExploreFirstAgent('check_performance', logs))
      unittest.TextTestRunner(verbosity=0).run(suite);
```

```
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

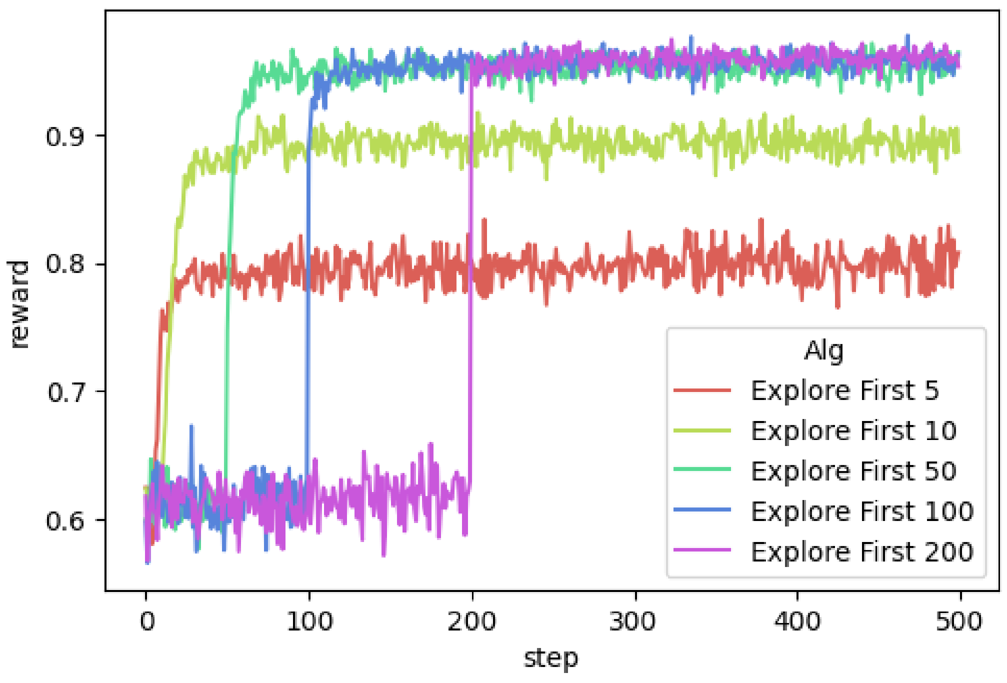
2.3.1 Explore First v.s. Random Agent

The results clearly show that the explore first agent performs better than the random agent. However, it still performs much worse than the oracle. How can we improve our performance?

If there are 10 possible actions but the agent only explores for 5 steps, then it is likely it won't find the best arm. Thus, the policy will be suboptimal. Let's see what happens when we allow the agent to explore for more steps.

[16]:
```
'''
What happens if we allow the agent to explore for only 5, 10, 50, 100, 200␣
↪steps respectively?
'''
max_explore_steps = [5, 10, 50, 100, 200]
n_runs = 1000
#### TODO: run ExploreFirstAgent with different max_explore steps, and plot the␣
↪reward curves [10pts]####
agents = [ExploreFirstAgent(num_actions=len(probs), max_explore=max_explore)␣
↪for max_explore in max_explore_steps]
logs = bandit_sweep(agents, probs, [f'Explore First {steps}' for steps in␣
↪max_explore_steps], n_runs=n_runs, max_steps=500)
plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean',␣
↪ci=None);
########################################
```

```
  0%|          | 0/5 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]
```



Analyzing the Results

- Notice that for all agents there is a jump in performance. This corresponds to the time point when they switch from explore only to exploit mode.
- The agents that explore for 5, 10 steps are unable to accurately identify the best arm everytime. Their scores are lower than that of agents exploring for 50 or 100 steps. These agents find the optimal arm.

Moving to More Realistic Scenarios

Question (5pts): It's unclear how long the agent should explore before switching to exploit mode. Can you come up with a strategy to choose a good value of `max_explore`? Can we use such a strategy to deploy a product?

Answer: Somehow, I think one should try to keep track of how certain one is of one's estimates and only switch to pure exploitation when we cross some certainty treshold.

If we can deploy such a product depends on several things. First, when we explore, we do purely random actions for a while. If the consequence of performing purely random actions are dire, it's a terrible strategy to deploy it. Furthermore, if our users are changing, we will not be able to detect that as we're just continuously exploiting. I.e., it's probably a bad strategy.

2.4 UCB Agent

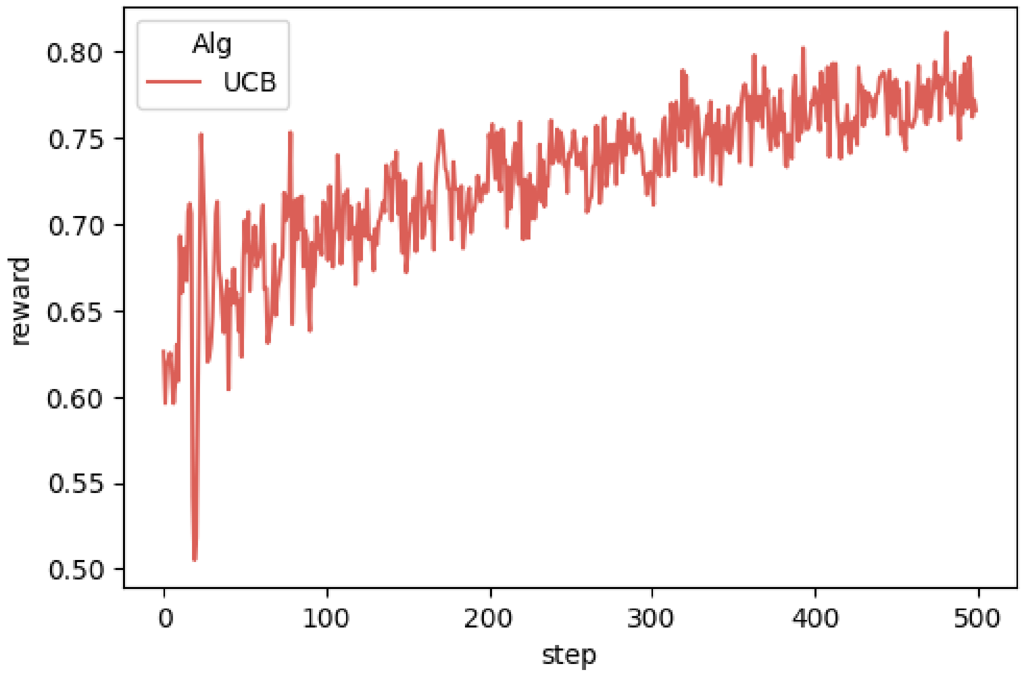
Rather than having a fixed delineation between exploration and exploitation, an agent should be able to figure out when to explore and when to exploit. This leads us to the UCB agent that we discussed in class.

Implement the `update_Q` and `get_action` methods for a UCB agent using the course notes.

```
[17]: #### UCB Agent ####
      @dataclass
      class UCBAgent:
          num_actions: int

          def __post_init__(self):
              self.reset()

          def reset(self):
              self.t = 0
              self.action_counts = np.zeros(self.num_actions, dtype=int) # action␣
        ↪counts n(a)
              self.Q = np.zeros(self.num_actions, dtype=float) # action value Q(a)

          def update_Q(self, action, reward):
              # Update Q action-value given (action, reward)
              #### TODO: Calculate the Q-value [5pts] ####
              self.action_counts[action] += 1
              alpha = 1 / self.action_counts[action]
              self.Q[action] += alpha * (reward - self.Q[action])
              ###############################

          def get_action(self):
              self.t += 1
              #### TODO: Calculate the exploration bonus. To avoid a division by␣
        ↪zero, add a small delta=1e-5 to the denominator [5pts] ####
              eps = 1e-5
              exploration_bonus = np.sqrt(4 * np.log(self.t) / (self.action_counts +␣
        ↪eps))
              ###########################################
              Q_explore = self.Q + exploration_bonus
              return np.random.choice(np.where(Q_explore == Q_explore.max())[0])
```

```
[18]: #Define the UCB Agent
      agentUCB = UCBAgent(num_actions=len(probs))
      #Compute Performance
      logs = bandit_sweep([agentUCB], probs, ['UCB'], n_runs=1000, max_steps=500)
      #Plot Performance
      plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean',␣
        ↪ci=None);
```

```
  0%|          | 0/1 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]
```



Check your work: If all other tests pass but check_performance fails, make sure your runtime type is GPU and then come to OH or post on piazza.

[19]:
```
suite = unittest.TestSuite()

suite.addTest(hw1_tests.TestUCBAgent('check_update_Q', UCBAgent))
suite.addTest(hw1_tests.TestUCBAgent('check_exploration_bonus', UCBAgent))
suite.addTest(hw1_tests.TestUCBAgent('check_performance', logs))

unittest.TextTestRunner(verbosity=0).run(suite);
```

```
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

2.4.1 UCB v/s Explore-First

Now let's compare the reward curves of the `UCB` agent and `Explore First` agent with `max_explore=5`.

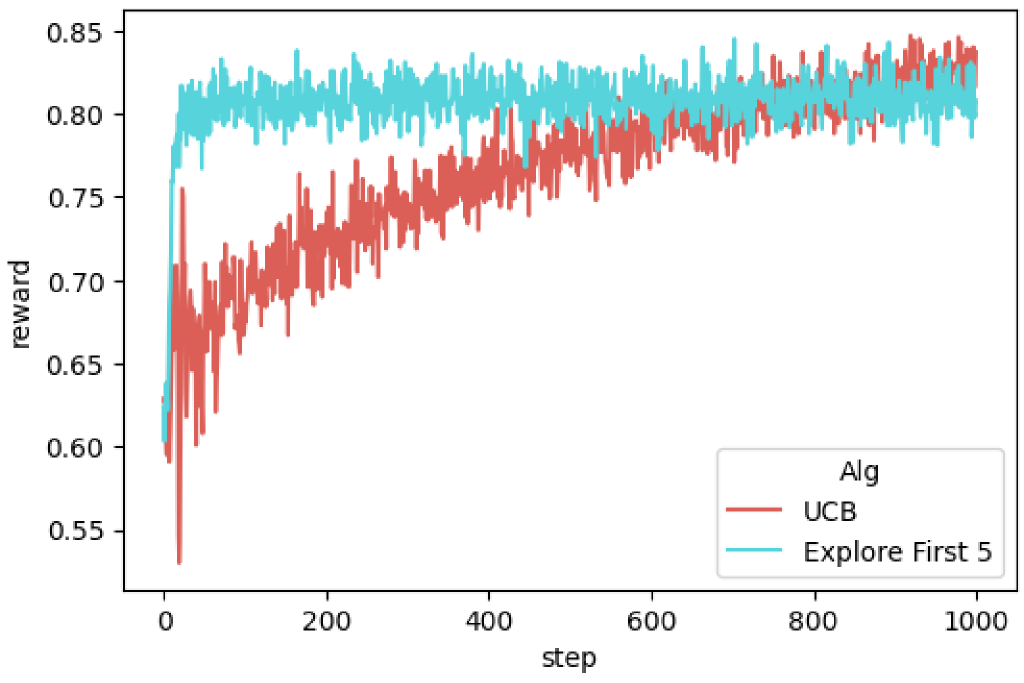
Analyzing the Results

Question [5pts]: Why does the UCB algorithm learn slowly (even after 500 steps, the agent still does not reach the maximum reward)?

Answer: Since we add the term $\sqrt{\frac{4 \log t}{k_i}}$ to the Q-values, we will encourage exploration. This encouragement comes from the $\log t$ factor, as this value will grow monotonically with time, and increase the "perceived" Q-value of little-used actions. However, since we're dividing by the number of times the action is taken k_i (without the log), which grows much faster than the log, this exploration bonus will over time tend to 0. However, as we're neglecting bad actions, $\log t$ still grows, which makes undesirable actions desirable again until we can grow k_i to a sufficient level to offset it. Therefore, the learning can be slower than necessary when it's relatively easy to discern which actions are the best and not.

[20]:
```
# Now we will compare the UCB agent against the ExploreFirst Agent that only␣
↪explores for 5 steps.
#### TODO: run both algorithms and plot the reward curves (max_explore=5)␣
↪[10pts] ####
#### use legends ['UCB', 'Explore First 5'] respectively
#### run each algorithm 1000 times (n_runs=1000), and max_steps=1000
agentUCB = UCBAgent(num_actions=len(probs))
agentEF = ExploreFirstAgent(num_actions=len(probs), max_explore=5)

# Compute performance
logs = bandit_sweep([agentUCB, agentEF], probs, ['UCB', 'Explore First 5'],␣
↪n_runs=1000, max_steps=1000)

# Plot performance
plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean',␣
↪ci=None);
```

```
  0%|          | 0/2 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]
```

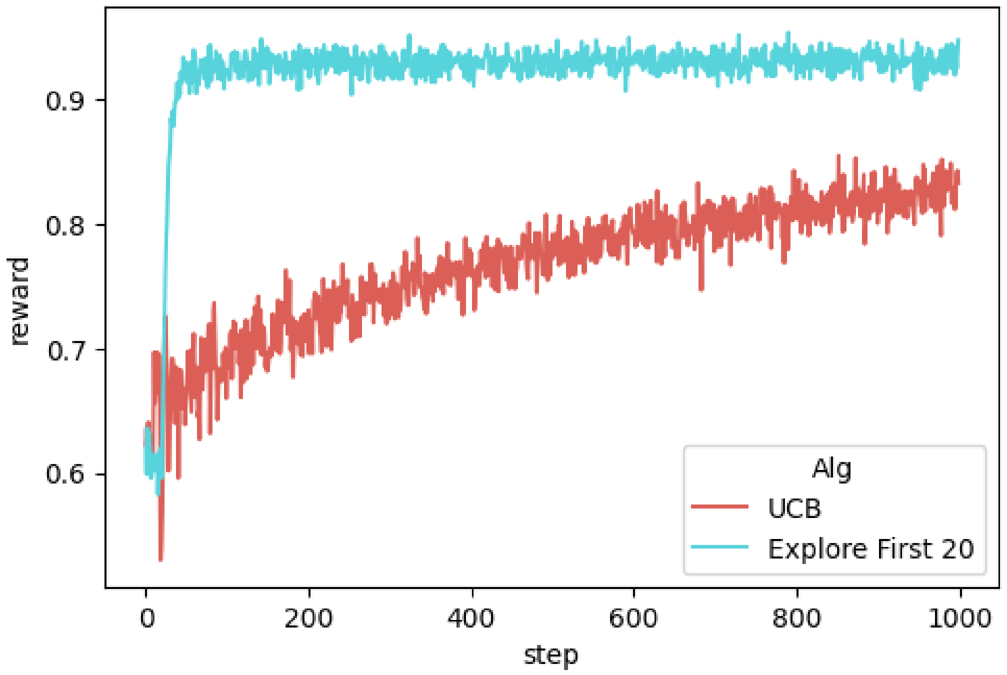


Result Analysis: UCB outperforms the greedy Explore First agent that only explores for 5 steps.

What happens if we allow the agent to explore for more steps? Run the Explore First agent for 20 steps, and compare the reward to the UCB agent.

[21]:
```
#Lets compare UCB with an agent that explores for twenty steps.
#### TODO: run both algorithms and plot the reward curves (max_explore=20)␣
↪[10pts] ####
#### use legends ['UCB', 'Explore First 20'] respectively
#### run each algorithm 1000 times (n_runs=1000), and max_steps=1000
agentUCB = UCBAgent(num_actions=len(probs))
agentEF = ExploreFirstAgent(num_actions=len(probs), max_explore=20)

# Compute performance
logs = bandit_sweep([agentUCB, agentEF], probs, ['UCB', 'Explore First 20'],␣
↪n_runs=1000, max_steps=1000)

# Plot performance
plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean',␣
↪ci=None);
```

```
  0%|          | 0/2 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]
```



Question: In the lecture we studied that the UCB algorithm is optimal. Why then does Explore First perform better?

Answer: While UCB algorithm is optimal for minimizing long-term regret, Explore First algorithm can perform better in some cases, especially when the bandit problem is simple and there is a large difference in reward between the best and the other arms. Here, for example, the agent is easily able to identify the optimal action to take with only 20 steps since it's relatively easy to discern the values of different actions.

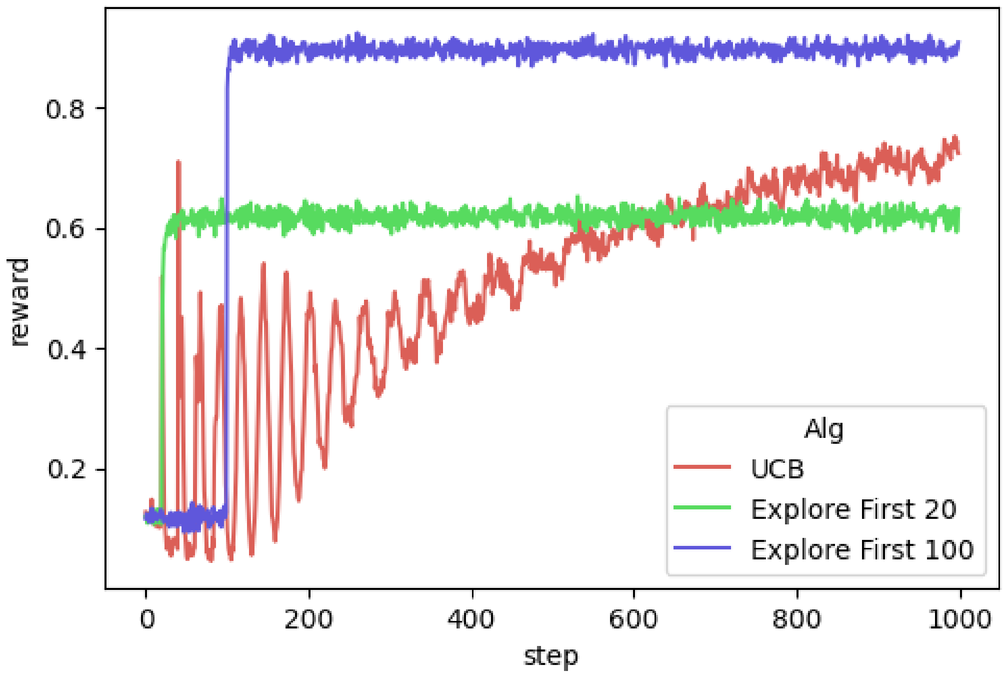
2.4.2 Skewed Arms Scenario:

In the previous example, the probability of each arm providing a return was sampled uniformly from $[0, 1]$. Because there were only 10 arms, and some arms had similar returns, by performing 20 random actions it is possible to find the best arm by chance. However, if the reward distributions are very skewed (e.g., only one arm returns rewards with high probability, say 0.9), or there are more arms, more actions may be necessary. In this case the initial exploration phase may not succeed at finding the best arm. Lets see this in practice below.

[22]:
```
skewed_probs = [0.1, 0.2, 0.15, 0.21, 0.3, 0.05, 0.9, 0.13, 0.17, 0.07, 0.01, 0.
↪01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01]
#### TODO: compare the reward curves of UCBAgent and ExploreFirstAgent␣
↪(max_explore=len(skewed_probs)) [10pts] ####
#### sweep with n_runs=1000, max_steps=1000

max_explore = len(skewed_probs)
extra_explore = 100
```

```
agentUCB = UCBAgent(num_actions=len(skewed_probs))
agentEF = ExploreFirstAgent(num_actions=len(skewed_probs),␣
 ↪max_explore=max_explore)
agentEF_extra = ExploreFirstAgent(num_actions=len(skewed_probs),␣
 ↪max_explore=extra_explore)

# Compute performance
logs = bandit_sweep(
    [agentUCB, agentEF, agentEF_extra],
    skewed_probs,
    ['UCB', f'Explore First {max_explore}', f'Explore First {extra_explore}'],
    n_runs=1000,
    max_steps=1000,
)

# Plot performance
plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean',␣
 ↪ci=None);
```

```
  0%|          | 0/3 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]
```



In this case, UCB performs better than *Explore First (20)*. It is because exploring for 20 steps is insufficient for this problem. This problem again illustrates that unless one has access to privileged information about the problem, UCB performs the best!

Also notice that UCB's reward is still increasing and it hasn't converged to the optimal action yet. Try varying the maximum number of steps to see when UCB converges to the optimal / oracle policy.

In other words, `max_explore` is a hyperparameter. Without "tuning" it, the method may perform well on some problem instances and poorly on others. An advantage of UCB is its lack of hyperparameters. Next, we'll consider another hyperparameter, ϵ.

2.5 Epsilon-greedy Agent

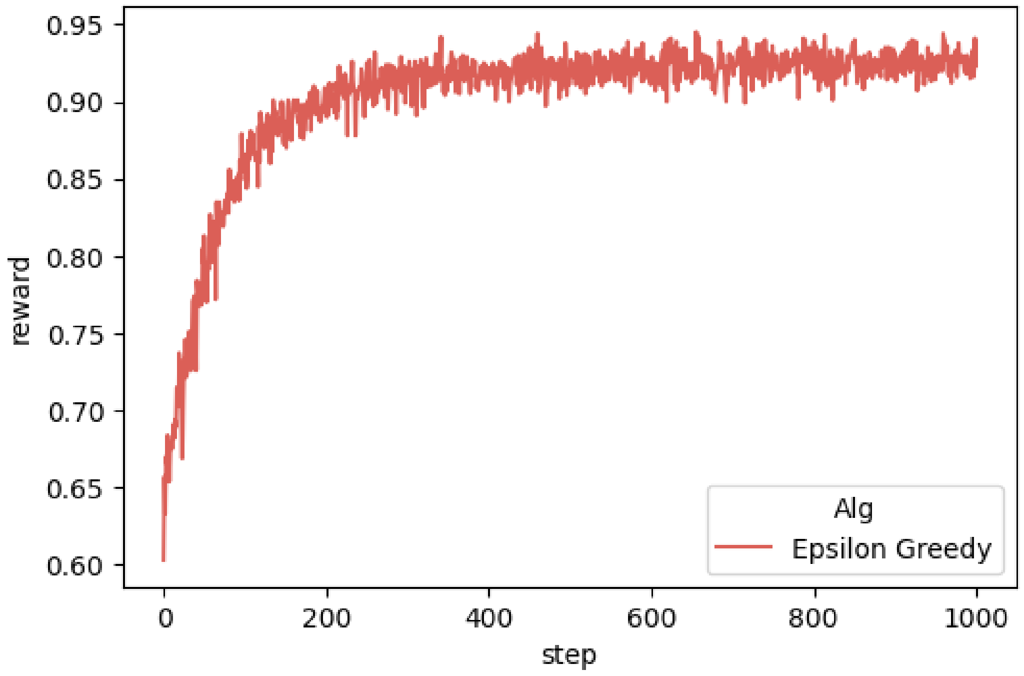
Another popular method of simultaneoulsy exploring/exploiting is ϵ-greedy exploration. The main idea is to: - Sample the (estimated) best action with probability $1 - \epsilon$ - Perform a random action with probability ϵ

By changing ϵ, we can control if the agent is conservative or exploratory. We will now implement this agent.

[23]:
```
## EpsilonGreedy Agent
@dataclass
class EpsilonGreedyAgent:
    num_actions: int
    epsilon: float = 0.1

    def __post_init__(self):
        self.reset()

    def reset(self):
        self.action_counts = np.zeros(self.num_actions, dtype=int) # action␣
↪counts n(a)
        self.Q = np.zeros(self.num_actions, dtype=float) # action value Q(a)

    def update_Q(self, action, reward):
        # Update Q action-value given (action, reward)
        self.action_counts[action] += 1
        self.Q[action] += (1.0 / self.action_counts[action]) * (reward - self.
↪Q[action])

    def get_action(self):
        # Epsilon-greedy policy
        if np.random.uniform() < self.epsilon:
            # Exploration
            selected_action = np.random.choice(self.num_actions)
        else:
            # Exploitation
            selected_action = np.random.choice(np.where(self.Q == self.Q.
↪max())[0])
        return selected_action
```

[24]:
```
agent = EpsilonGreedyAgent(num_actions=len(probs), epsilon=0.1)
logs = bandit_sweep([agent], probs, ['Epsilon Greedy'], n_runs=1000,␣
↪max_steps=1000)
plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean',␣
↪ci=None);
```

```
  0%|          | 0/1 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]
```



[25]:
```
suite = unittest.TestSuite()
suite.addTest(hw1_tests.TestEpsilonGreedyAgent('check_performance', logs))
unittest.TextTestRunner(verbosity=0).run(suite);
```

```
----------------------------------------------------------------------
Ran 1 test in 0.003s

OK
```

Analyzing Epsilon-Greedy Agents

Notice that the reward of all agents gradually increases (except for $\epsilon = 0$, which is an extremely greedy agent). Also, notice that reward is maxmium for $\epsilon = 0.1$ but decreases for higher values.

Question [**5pts**]: Why is the reward lower for higher-values of ϵ?

Answer: With a higher epsilon, the agent will choose random actions more often even though it has found the best action with high confidence.

Question [**5pts**]: To overcome the issue above, one can try setting $\epsilon = 0$ after some time or adaptively changing ϵ. Can you suggest a strategy for varying ϵ with time T?

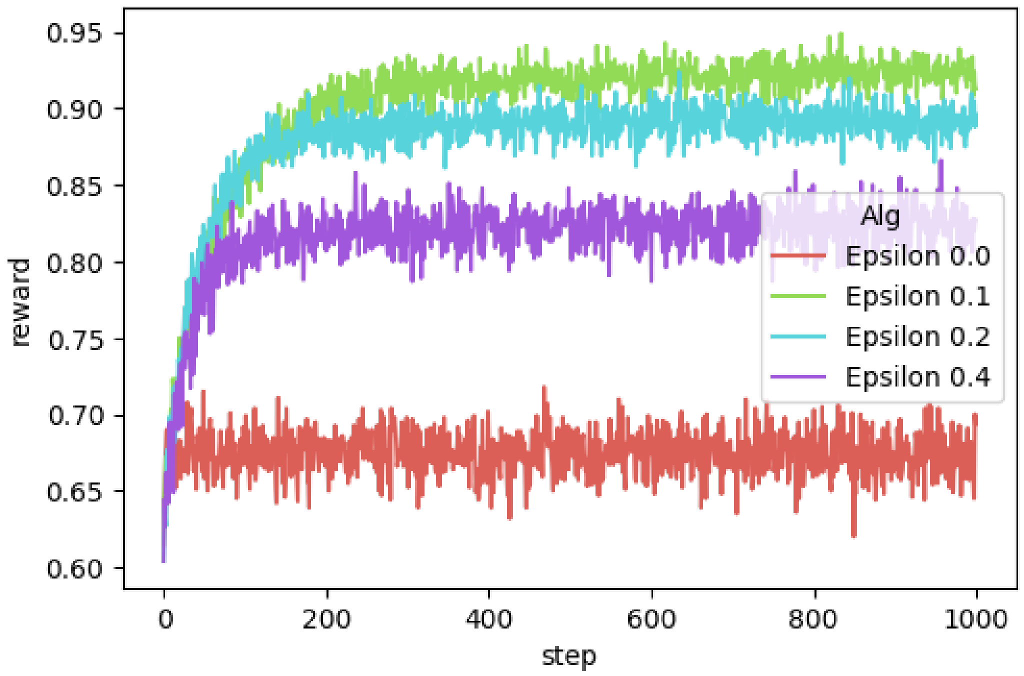
Answer: Three strategy could be (1) decrease epsilon by some constant decrement for each action, (2) exponentionally decrease epsilon with some factor (0, 1), or (3) have epsilon >0 for some number of steps and set it to 0 after that.

Consider: Compare ϵ-greedy with UCB and the tradeoffs.

Answer: We see from the below plot that to get optimal behavior we need to find a good epsilon, i.e., we need to tune the hyperparameter epsilon for the epsilon-greedy strategy to be effective. The UCB agent, on the other hand, does not require hyperparameter tuning to converge to the optimal over time. This agent can converge slowly, as we have seen, however.

[26]:
```
#### TODO: show reward curves of an EpsilonGreedyAgent with epsilon=[0, 0.1, 0.
↪2, 0.4] [10pts]####
epsilons = [0.0, 0.1, 0.2, 0.4]

logs = bandit_sweep(
    [EpsilonGreedyAgent(num_actions=len(probs), epsilon=epsilon) for epsilon in␣
↪epsilons],
    probs,
    [f'Epsilon {eps}' for eps in epsilons],
    n_runs=1000,
    max_steps=1000,
)
plot(logs, x_key='step', y_key='reward', legend_key="Alg", estimator='mean',␣
↪ci=None,);
```

```
  0%|          | 0/4 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1000 [00:00<?, ?it/s]
```



3 Contextual bandits

In this section, we will deal with contextual bandits problem. In contextual bandits, we use contextual information about the observed subject to make subject-specific decisions. The algorithm we will implement is called LinUCB.

As an example, imagine we have a website with 10 products that we'd like to promote. Whenever a user enters the website, the website promotes one product to the user. If the user clicks the product link, then it's a successful promotion (reward is 1). Otherwise, it's a failed promotion (reward is 0). Our goal is to optimize the click through rate (CTR), and thus optimize our $$$.

We will use a dataset from here to explore contextual bandits. The dataset contains a pre-logged array of shape $[10000, 102]$. Each row represents a data point at time step t where $t \in [0, 9999]$. The first column represents the index of the arm a_t that's chosen (10 arms in total). The second column represents the reward $r_t \in \{0, 1\}$ received for taking the selected arm. The last 100 columns represent the context feature vector.

The following code is inspired by this code repository.

[27]:
```
# Download the dataset
!wget http://www.cs.columbia.edu/~jebara/6998/dataset.txt
```

```
--2023-02-22 00:10:35--  http://www.cs.columbia.edu/~jebara/6998/dataset.txt
Resolving www.cs.columbia.edu (www.cs.columbia.edu)… 128.59.11.206
Connecting to www.cs.columbia.edu (www.cs.columbia.edu)|128.59.11.206|:80…
connected.
```

```
HTTP request sent, awaiting response… 200 OK
Length: 2149159 (2.0M) [text/plain]
Saving to: ‘dataset.txt’

dataset.txt         100%[===================>]   2.05M  --.-KB/s    in 0.06s

2023-02-22 00:10:35 (32.9 MB/s) - ‘dataset.txt’ saved [2149159/2149159]
```

[28]:
```python
# load in the dataset
#### TODO: load in the dataset.txt, and extract the data as a numpy array of␣
 ↪shape [10000, 102] ####
data = pd.read_csv('dataset.txt', sep=" ", header=None)
data = data.iloc[:, :-1]
print(f'Dataset shape:{data.shape}')
data[0] -= 1 # we use 0-based numbering
data = data.to_numpy()
```

```
Dataset shape:(10000, 102)
```

[29]:
```python
#### Contextual bandit environment ####
@dataclass
class ContextualBanditEnv:
    dataset: Any
    t: int = 0

    def step(self, action):
        # if the action matches the recorded action in the dataset, it will
        # return the recorded reward in the dataset. Otherwise, it will return
        # a reward of None
        if action == self.dataset[self.t, 0]:
            reward = self.dataset[self.t, 1]
        else:
            reward = None
        self.t += 1
        return reward

    def reset(self):
        self.t = 0
```

Fill in the missing code below to implement the LinUCB agent.

[30]:
```python
#### LinUCB Agent ####
@dataclass
class LinUCBAgent:
    num_actions: int
    alpha: float
    feature_dim: int
    def __post_init__(self):
        self.reset()

    def reset(self):
        self.As = [np.identity(self.feature_dim) for i in range(self.
↪num_actions)]
        self.bs = [np.zeros([self.feature_dim, 1]) for i in range(self.
↪num_actions)]

    def get_ucb(self, action, state):
        #### TODO: compute the UCB of the selected action/arm, and the context␣
↪information [5pts] ####
        A = self.As[action]
        b = self.bs[action]
        x = state.reshape(-1, 1)
        A_inv = np.linalg.inv(A)
        theta = np.dot(A_inv, b)
        p = np.dot(theta.T, x) + self.alpha * np.sqrt(np.dot(np.dot(x.T,␣
↪A_inv), x))
        return p

    def update_params(self, action, reward, state):
        #### update A matrix and b matrix given the observed reward, ####
        #### selected action, and the context feature                ####
        if reward is None:
            return

        #### TODO: update A and b matrices of the selected arm [5pts] ####
        x = state.reshape(-1, 1)
        A = self.As[action]
        b = self.bs[action]
        self.As[action] = A + np.dot(x, x.T)
        self.bs[action] = b + reward * x

    def get_action(self, state):
        #### find the action given the context information ####

        arms_ucb = np.zeros(self.num_actions)
        for arm_id in range(self.num_actions):
            arm_ucb = self.get_ucb(arm_id, state)
            arms_ucb[arm_id] = arm_ucb

        #### TODO: choose an arm a_t=\arg\max_a(p_{t,a}) with ties broken␣
↪arbitrarily [5pts] ####
        selected_action = np.random.choice(np.where(arms_ucb == arms_ucb.
↪max())[0])
```

```
        return selected_action
```

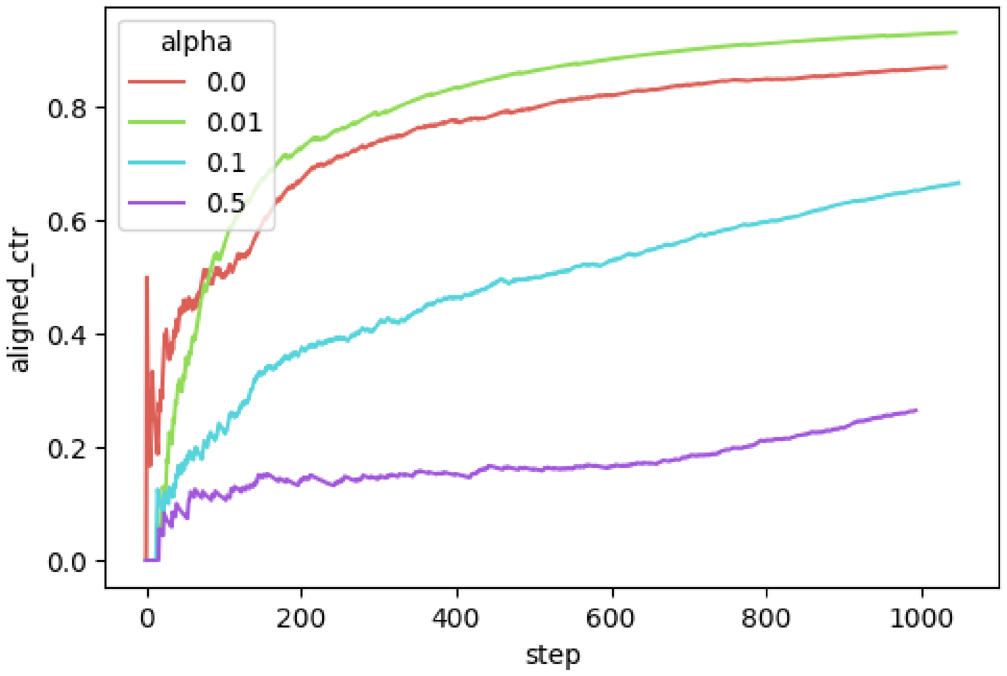
[31]:
```
# Code for running the contextual bandit environment.
@dataclass
class CtxBanditEngine:
    dataset: Any
    agent: Any

    def __post_init__(self):
        self.env = ContextualBanditEnv(dataset=self.dataset)

    def run(self, n_runs=1):
        log = []
        for i in tqdm(range(n_runs), desc='Runs'):
            # We only record the time steps when the selected arm matches the␣
 ↪arm in the pre-logged data
            aligned_ctr = []
            ret_val = 0
            valid_time_steps = 0
            self.env.reset()
            self.agent.reset()
            for t in tqdm(range(self.dataset.shape[0]), desc='Time'):
                state = self.dataset[t, 2:]
                action = self.agent.get_action(state=state)
                reward = self.env.step(action)
                self.agent.update_params(action, reward, state=state)
                if reward is not None:
                    ret_val += reward
                    valid_time_steps += 1
                    aligned_ctr.append(ret_val / float(valid_time_steps))
            data = {'aligned_ctr': aligned_ctr,
                    'step': np.arange(len(aligned_ctr))}
            if hasattr(self.agent, 'alpha'):
                data['alpha'] = self.agent.alpha
            run_log = pd.DataFrame(data)
            log.append(run_log)
        return log
```

[32]:
```
# Code for aggregrating results of running an agent in the contextual bandit␣
 ↪environment.
def ctxbandit_sweep(alphas, dataset, n_runs=2000):
    logs = dict()
    pbar = tqdm(alphas)
    for idx, alpha in enumerate(pbar):
        pbar.set_description(f'alpha:{alpha}')
        agent = LinUCBAgent(num_actions=10, feature_dim=100, alpha=alpha)
        engine = CtxBanditEngine(dataset=dataset, agent=agent)
        ep_log = engine.run(n_runs)
        ep_log = pd.concat(ep_log, ignore_index=True)
        ep_log['alpha'] = alpha
        logs[f'{alpha}'] = ep_log
    logs = pd.concat(logs, ignore_index=True)
    return logs
```

[33]:
```
# Run the sweep with alpha = [0, 0.01, 0.1, 0.5] and n_runs=1
logs = ctxbandit_sweep([0., 0.01, 0.1, 0.5], data, n_runs=1)

# Save this here so we don't have to recompute later
linucb_logs = logs[logs.alpha != 0.01]
```

```
  0%|          | 0/4 [00:00<?, ?it/s]

Runs:   0%|          | 0/1 [00:00<?, ?it/s]

Time:   0%|          | 0/10000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1 [00:00<?, ?it/s]

Time:   0%|          | 0/10000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1 [00:00<?, ?it/s]

Time:   0%|          | 0/10000 [00:00<?, ?it/s]

Runs:   0%|          | 0/1 [00:00<?, ?it/s]

Time:   0%|          | 0/10000 [00:00<?, ?it/s]
```

[34]:
```
plot(logs, x_key='step', y_key='aligned_ctr', legend_key='alpha',␣
  ↪estimator='mean', ci=None);
```



Check your work:

```
[35]: suite = unittest.TestSuite()
      suite.addTest(hw1_tests.TestLinUCBAgent('check_get_ucb', LinUCBAgent))
      suite.addTest(hw1_tests.TestLinUCBAgent('check_update_params', LinUCBAgent))
      suite.addTest(hw1_tests.TestLinUCBAgent('check_logs', logs))
      unittest.TextTestRunner(verbosity=0).run(suite)
```

```
----------------------------------------------------------------------
Ran 3 tests in 0.005s

OK
```

```
[35]: <unittest.runner.TextTestResult run=3 errors=0 failures=0>
```

Question [5pts]: What does α affect in LinUCB?

Answer: In LinUCB, α is a hyperparameter that controls the exploration-exploitation trade-off. It determines the degree of exploration by controlling the size of the confidence interval around the estimated mean reward of each arm. The higher α is, the more likely the agent is to assign a high value to an arm and thus it is more likely to select that arm.

Question [5pts]: Do the reward curves change with α? If yes, why? If not, why not?

Answer: Yes, they change. We can observe that $\alpha = 0.01$ performs the best, while increasing values of α performs worse. $\alpha = 0$ does slightly worse than 0.01. It seems that having more or less exploration than $\alpha = 0.01$ does not provide a good balance between exploration and exploitation.

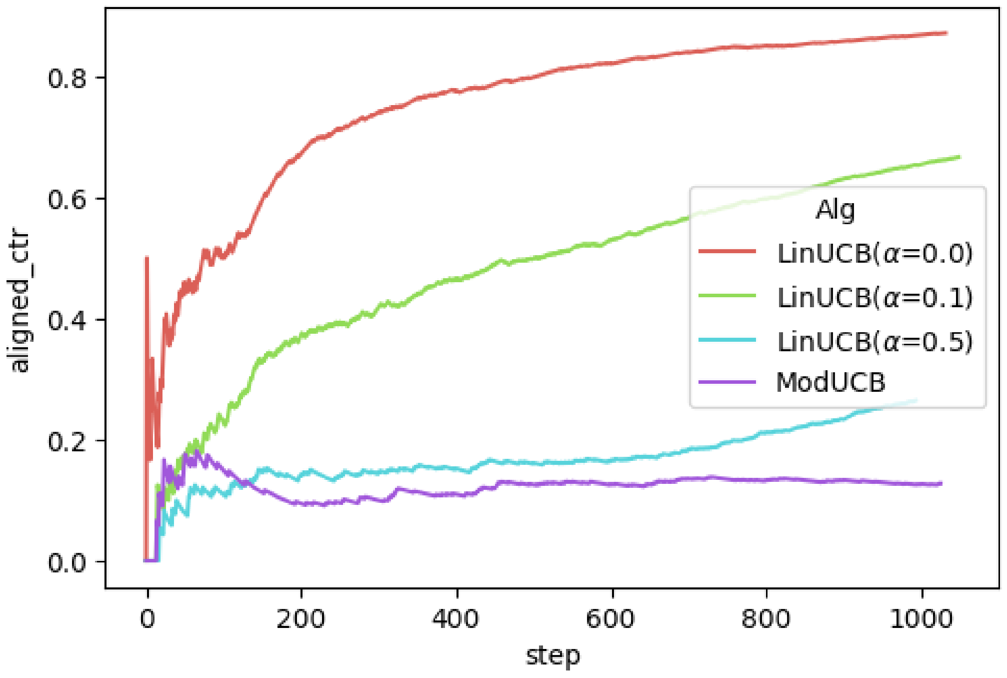
Finally, let's compare LinUCB to UCB. Make a class that modifies the UCB agent in the multi-armed bandits case and compare the aligned_ctr curved of LinUCBAgent and ModUCBAgent for alpha = 0, 0.01, 0.5.

```
[37]: @dataclass
      class ModUCBAgent(UCBAgent):
          num_actions: int

          #### TODO: modify the UCB agent in the multi-armed bandits
          def get_action(self, state):
              action = super().get_action()
              return action

          def update_params(self, action, reward, state):
              if reward is None:
                  return
              super().update_Q(action, reward)


      def ctx_bandit_sweep(agents, labels, dataset, n_runs=2000):
          logs = dict()
          pbar = tqdm(agents)
          for idx, agent in enumerate(pbar):
              pbar.set_description(f'Alg: {labels[idx]}')
              engine = CtxBanditEngine(dataset=dataset, agent=agent)
              ep_log = engine.run(n_runs)
              ep_log = pd.concat(ep_log, ignore_index=True)
              ep_log['Alg'] = labels[idx]
              logs[f'{labels[idx]}'] = ep_log
          logs = pd.concat(logs, ignore_index=True)
          return logs
```

```
[54]: agent = ModUCBAgent(num_actions=10)

      logs = ctx_bandit_sweep([agent], [f"ModUCB"], dataset=data, n_runs=1)

      linucb_logs.assign(Alg=linucb_logs.alpha.apply(lambda x:␣
       ↪f"LinUCB($\\alpha$={x})"))

      # Add in the run of the naive UCB agent to the runs with LinUCB
      compare_df = pd.concat([linucb_logs, logs], axis=0, ignore_index=True)
```

```
  0%|          | 0/1 [00:00<?, ?it/s]

Runs:   0%|          | 0/1 [00:00<?, ?it/s]

Time:   0%|          | 0/10000 [00:00<?, ?it/s]
```

[57]:
```python
plot(compare_df, x_key='step', y_key='aligned_ctr', legend_key='Alg',␣
    ↪estimator='mean', ci=None);
```



Question [**20pts**]: Does LinUCB outperform UCB? If yes, explain why. If not, explain why not.

Answer: Yes, we see that all variations of LinUCB outperforms the naïve UCB agent—which is uplifting! The reason LinUCB is better is because it has much more signal to work with, i.e., it har more information with which to make decisions as to what lever to pull. This makes it possible for LinUCB to dynamically change it's value estimates for different contexts, making it more accurate.

Overall, LinUCB has been shown to outperform UCB in a variety of experimental settings, especially in contextual bandit problems where the context vector provides additional information that can be used to make more informed arm-specific decisions.

Survey [BONUS 10pts]

https://forms.gle/EZVxUcizhdwFDgPJ6

[58]:
```python
survey_complete = True

print(f"Survey has{' not' if not survey_complete else ''} been filled out!")
```

```
Survey has been filled out!
```

[ ]:
```python

```