Spring 2021 6.884 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**).

There are 255 total points in this assignment, scaled to be worth 6.25% of your final grade.

In [ ]:
```
Student: Songhao Li
```

Setup

Ignore the following skeleton code (imports, plotting).

In [1]:
```
%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


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

In [2]:
```
# 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)
```

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` .

In [3]:
```
numArms = 10
probs = [np.random.random() for i in range(numArms)]
```

We then define an environment to evaluate different agent strategies.

In [4]:
```
#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
```

In [5]:
```
#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
```

In [6]:
```
#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/ShangtongZhang/reinforcement-learning-an-introduction/blob/master/chapter02/ten_armed_testbed.py)
- https://github.com/lilianweng/multi-armed-bandit/blob/master/solvers.py (https://github.com/lilianweng/multi-armed-bandit/blob/master/solvers.py)

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?

In [7]:
```
#### TODO: find the maximum return with priviledged infromation about the rewa
rd distribution [5pts] ####
oracleReward = max(probs)
##################################
print (f'Max possible reward {oracleReward}')
```

```
Max possible reward 0.9636627605010293
```

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.

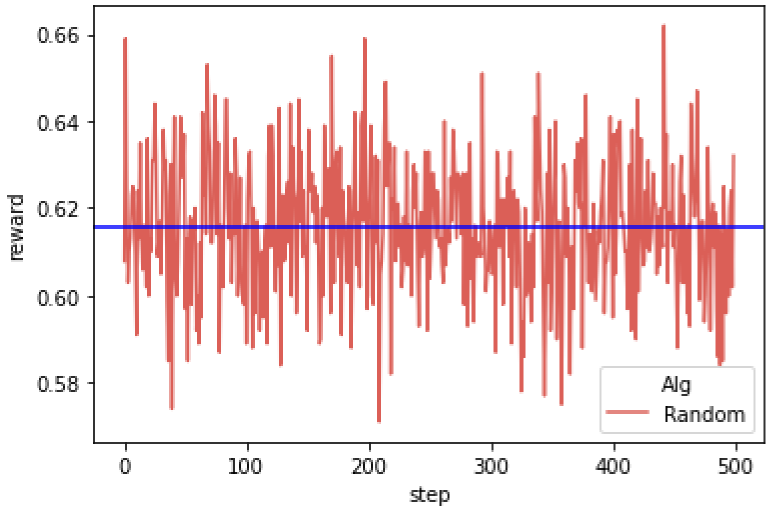
In [8]:
```
#As a baseline, lets first construct a baseline agent that chooses a random ac
tion at every timestep.
#We will measure how much better we can do.
@dataclass
class RandomAgent:
    num_actions: int

    def __post_init__(self):
        self.reset()

    def reset(self):
        self.t = 0
        self.action_counts = np.zeros(self.num_actions, dtype=np.int) # action
counts n(a)
        self.Q = np.zeros(self.num_actions, dtype=np.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 = 0 # placeholder
        ##################################
        selected_action = np.random.randint(low = 0, high = self.num_actions)

        return selected_action
```

In [9]:
```
#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 age
nt
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 r
educe it
it if its taking too much time.
'''
n_runs = 1000
logs = bandit_sweep([agent], probs, ['Random'], n_runs=n_runs)
```

In [10]:
```
#### TODO: plot the reward curve of a random agent, and the average reward of
 all arms [5pts]####
plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean', c
i=None)
print (f'Average reward of all arms: {np.mean(probs)}')
plt.axhline(y=np.mean(probs),ls="-",c="blue")#
```

```
Average reward of all arms: 0.6157662833145425
```

Out[10]:
```
<matplotlib.lines.Line2D at 0x7f79bfdf6ed0>
```



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 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.

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.

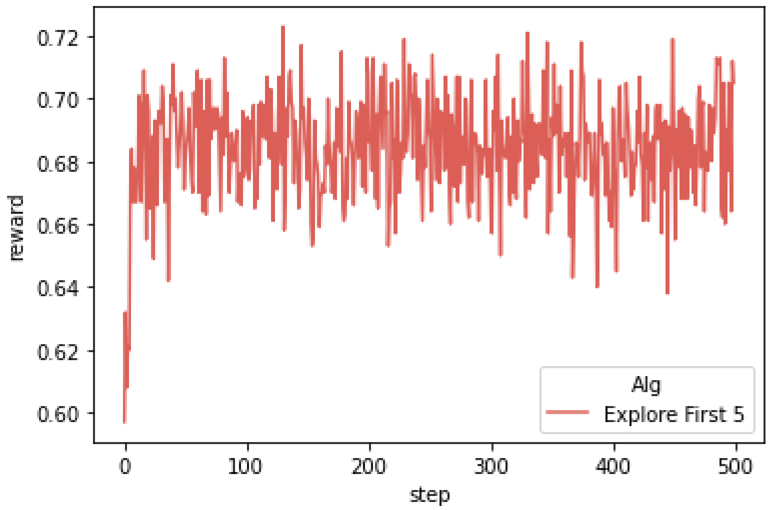
We will now implement this agent below. Fill in the missing code in `update_Q` and `get_action` . We will store the average reward for each action in the variable `self.Q` , and the count of how many times we've taken each action in `self.action_counts` .

In [11]:
```
#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=np.int) # action
counts n(a)
        self.Q = np.zeros(self.num_actions, dtype=np.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] ####
        if self.t <= self.max_explore:
            self.action_counts[action] += 1
            self.Q[action] = (self.Q[action]*(self.action_counts[action]-1) +
reward)/self.action_counts[action]
        ##################################

    def get_action(self):
        self.t += 1
        #### TODO: get action [5pts] ####
        selected_action = 0 # placeholder
        ##################################
        if self.t <= self.max_explore:
            selected_action = np.random.randint(low = 0, high = self.num_actio
ns)
        else:
            selected_action = np.random.choice(np.where(self.Q == self.Q.max
())[0])
        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).

In [12]:
```
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_step
s=500)
plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean', c
i=None)
```

Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x7f79bf619e10>

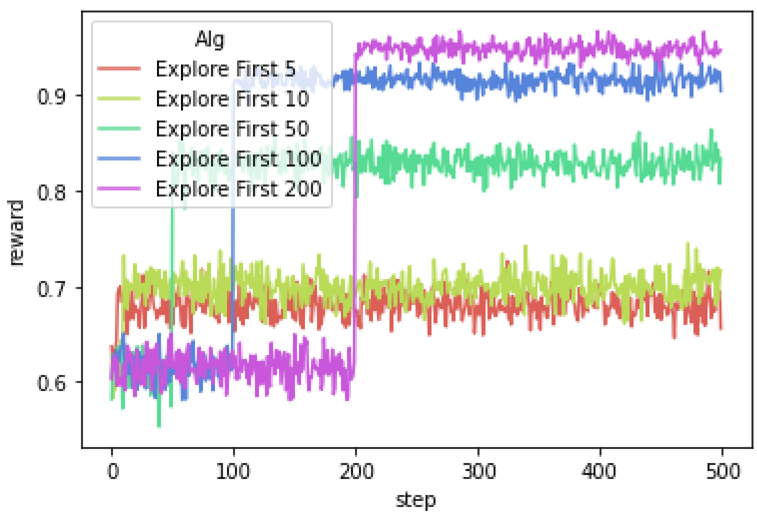


Explore First v.s. Random Agent

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.

In [16]:
```
'''
What happens if we allow the agent to explore for only 5, 10, 50, 100, 200 ste
ps respectively?
'''
max_explore_steps = [5, 10, 50, 100, 200]
agents = [ExploreFirstAgent(num_actions=len(probs), max_explore = max_explore)
for max_explore in max_explore_steps]
labels = ['Explore First ' + str(max_explore) for max_explore in max_explore_s
teps]

#### TODO: run ExploreFirstAgent with different max_explore steps, and plot th
e reward curves [10pts]####
logs = bandit_sweep(agents, probs, labels, n_runs=1000, max_steps=500)
plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean', c
i=None)
plt.show()
##########################################
```



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.

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.

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?

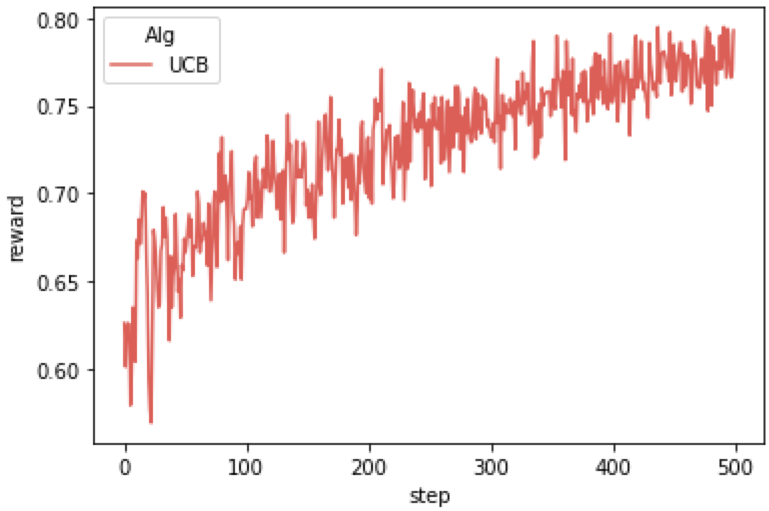
Answer: A good way could be probably picking max_explore as to maximize the total expected rewards. In some scenarios probably it is ok to deploy a product when the action space is discrete, limited, and not change after deployment. In other scenarios there will be problematic to deploy.

Hit each action sevearl times. Too many actions hard to implement

In [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=np.int) # action
counts n(a)
        self.Q = np.zeros(self.num_actions, dtype=np.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
        self.Q[action] = (self.Q[action]*(self.action_counts[action]-1) + rewa
rd)/self.action_counts[action]

    def get_action(self):
        self.t += 1
        ## HINT: To avoid a division by zero, you can add a small delta>0 to t
he denominator
        #### TODO: Calculate the exploration bonus [5pts] ####
        exploration_bonus = 0 # placeholder
        ###########################################
        exploration_bonus = np.sqrt(4*np.log(self.t)/(self.action_counts+0.05
)) # placeholder
        Q_explore = self.Q + exploration_bonus
        return np.random.choice(np.where(Q_explore == Q_explore.max())[0])
```

In [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', c
i=None)
```

Out[18]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f79b4df3750>
```



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

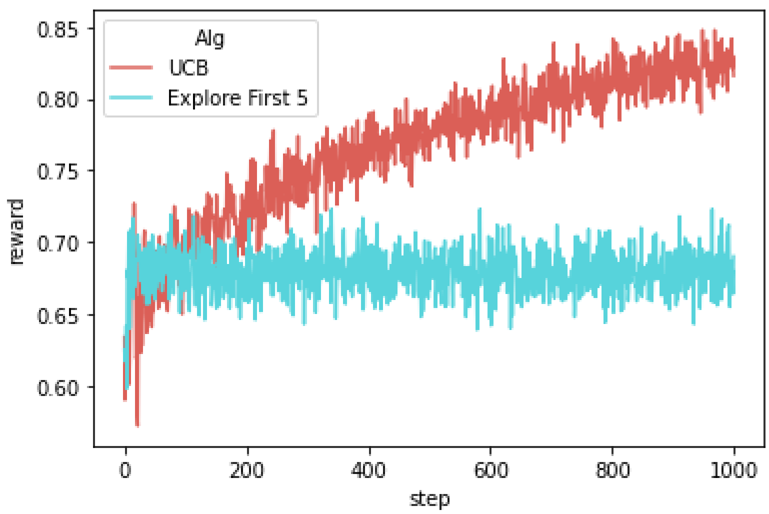
Question [5pts]: Why does the UCB algorithm learn slowly (even after 500 steps, the agent still does not reach the maximum reward)?

Answer: Because the Q-value is not constant compared with Explore-first algorithm, and with the inclusion of exploration bonus the optimal choice will possibly change over time

If you are willing to afford the risk of missing out on the optimal policy, one could terminate the exploration bonus in UCB after some time. This hybrid between UCB and Explore-First could work better for your use case. However, remember that UCB requires no tuning because it assumes no extra knowledge about the system. But, this comes at a cost -- if you have extra knowledge, you can achieve better performance than UCB! How to incoporate this knowledge depends on the nature of available information.

In [19]:
```
#Now we will compare the UCB agent against the ExploreFirst Agent that only ex
plores for 5 steps.
#### TODO: run both algorithms and plot the reward curves (max_explore=5) [10p
ts] ####
#### 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))
agentfirst5 = ExploreFirstAgent(num_actions=len(probs), max_explore = 5)

logs = bandit_sweep([agentUCB, agentfirst5], probs,  ['UCB', 'Explore First 5'
], n_runs=1000, max_steps=1000)

plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean', c
i=None)
##################################
```

Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x7f79b5040d10>

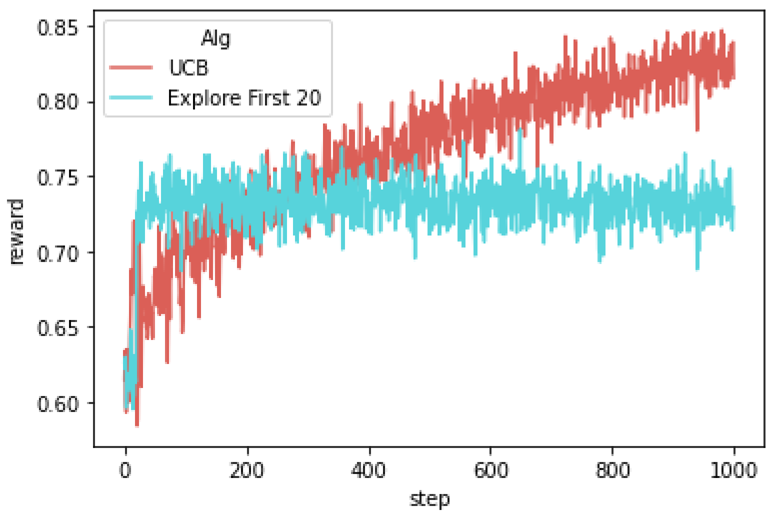


Result Analysis: UCB outperforms the greedy Explore First agent that only explores for 5 steps.

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.

In [20]:
```
#Lets compare UCB with an agent that explores for twenty steps.
#### TODO: run both algorithms and plot the reward curves (max_explore=20) [10
pts] ####
#### 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))
agentfirst5 = ExploreFirstAgent(num_actions=len(probs), max_explore = 20)

logs = bandit_sweep([agentUCB, agentfirst5], probs,  ['UCB', 'Explore First 2
0'], n_runs=1000, max_steps=1000)

plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean', c
i=None)
##################################
```

Out[20]: `<matplotlib.axes._subplots.AxesSubplot at 0x7f79b50efe50>`



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

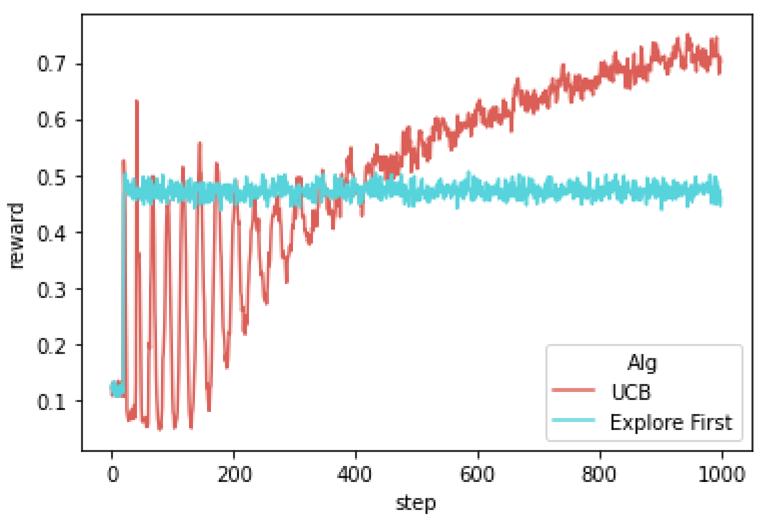
Answer: Because UCB algorithm is optimal when there is no extra information. Here we know that the action space is of size 10 so explore first 20 is a reasonable number, making the explore first algorithm out-perform UCB

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.

In [21]:
```
skewedProbs = [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_ex
plore=len(skewedProbs)) [10pts] ####
#### sweep with n_runs=1000, max_steps=1000

agentUCB = UCBAgent(num_actions=len(skewedProbs))
agentexplore = ExploreFirstAgent(num_actions=len(skewedProbs),max_explore=len(
skewedProbs))
logs = bandit_sweep([agentUCB, agentexplore], skewedProbs,  ['UCB', 'Explore F
irst'], n_runs=1000, max_steps=1000)
plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean', c
i=None)

####################################
```

Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x7f79b4ffdb50>



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.

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.

In [22]:
```
##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=np.int) # action counts n(a)
        self.Q = np.zeros(self.num_actions, dtype=np.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
        #### TODO: Code for exploration [5pts] ####
        selected_action = 0 # placeholder
        ##################################
        if np.random.random() < self.epsilon:
            selected_action = np.random.randint(low = 0, high = self.num_actions)
        else:
            selected_action = np.argmax(self.Q)
        ##################################
        return selected_action
```

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: Because higher-values of ϵ means there are more steps are executed randomly instead of exploiting the information available

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

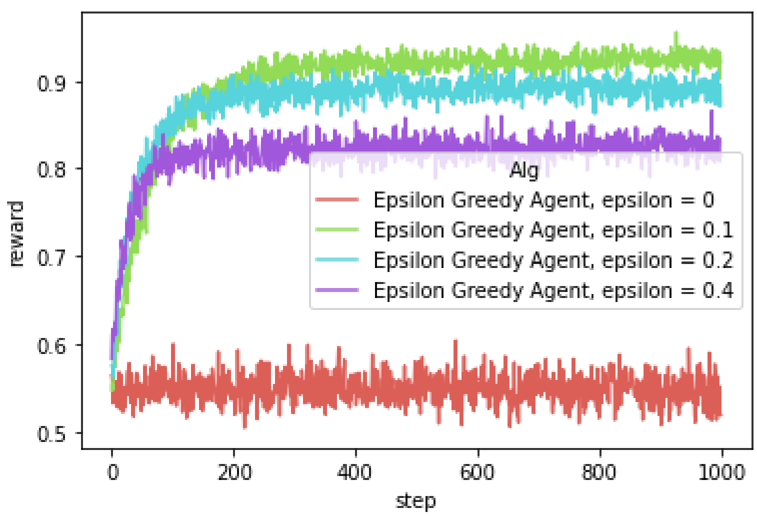
Answer: Probably higher ϵ when T is small to explore enough and then decrease ϵ when the information is collected/the environment is explored enough

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

In [23]:
```
#### TODO: show reward curves of an EpsilonGreedyAgent with epsilon=[0, 0.1,
 0.2, 0.4] [10pts]####

epsilons = [0, 0.1, 0.2, 0.4]
agents = [EpsilonGreedyAgent(num_actions = len(probs), epsilon = epsilon) for
epsilon in epsilons]
labels = ['Epsilon Greedy Agent, epsilon = ' +str(epsilon) for epsilon in epsi
lons]

logs = bandit_sweep(agents, probs,  labels, n_runs=1000, max_steps=1000)

###########################################
```

In [24]:
```
plot(logs, x_key='step', y_key='reward', legend_key='Alg', estimator='mean', c
i=None)
```

Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x7f79b4aba4d0>



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 (https://arxiv.org/pdf/1003.0146.pdf).

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 (http://www.cs.columbia.edu/~jebara/6998/dataset.txt) 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 (https://github.com/kfoofw/bandit_simulations).

In [25]:
```
# Download the dataset
!wget http://www.cs.columbia.edu/~jebara/6998/dataset.txt
```

```
--2021-02-24 23:56:58--  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  3.79MB/s    in 0.5s

2021-02-24 23:56:59 (3.79 MB/s) - ‘dataset.txt’ saved [2149159/2149159]
```

In [26]:
```
# Load in the dataset
#### TODO: Load in the dataset.txt, and extract the data as a numpy array of s
hape [10000, 102] ####
dataset = np.loadtxt('dataset.txt')
```

In [27]:
```
#### 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.

In [28]:
```
#### 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_actio
ns)]
        self.bs = [np.zeros([self.feature_dim, 1]) for i in range(self.num_act
ions)]

    def get_ucb(self, action, state):
        #### TODO: compute the UCB of the selected action/arm, and the context
information [5pts] ####

        A = self.As[action-1]
        b = self.bs[action-1]
        theta = np.dot(np.linalg.inv(A), b)

        x = state.reshape([-1,1])
        p = np.dot(theta.T,x) + self.alpha * np.sqrt(np.dot(x.T,np.dot(np.lina
lg.inv(A),x)))
        return p[0,0]

    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
        x = state.reshape([-1,1])
        self.As[action-1] += np.dot(x,x.T)
        self.bs[action-1] += reward * x
        return


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

        #### TODO: get the UCB estimates for all actions [5pts] ####
        UCB_estimates = np.zeros(shape = self.num_actions)
        highest_ucb = -1
        candidate_arms = []

        for action in range(1,self.num_actions+1):
            ucb = self.get_ucb(action,state)
            if ucb > highest_ucb:
                highest_ucb=ucb
                candidate_arms = [action]
            elif ucb == highest_ucb:
                candidate_arms.append(action)
```

```
        #### TODO: choose an arm a_t=\arg\max_a(p_{t,a}) with ties broken arbi
trarily [5pts] ####
        #print(candidate_arms)
        selected_action = np.random.choice(candidate_arms)

        return selected_action
```

In [29]:

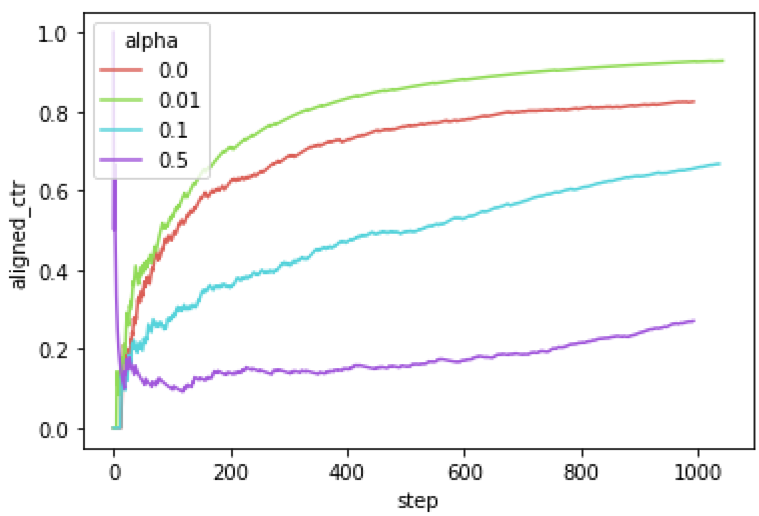
```
#Code for running the contextual bandit environment.
@dataclass
class CtxBanditEngine:
    dataset: Any
    agent: Any

    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
```

In [30]:

```
#Code for aggregrating results of running an agent in the contextual bandit en
vironment.
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
```

In [31]:
```
#### TODO: run the sweep with alpha = [0, 0.01, 0.1, 0.5] and n_runs=1 [5pts]
 ####
logs = ctxbandit_sweep(alphas = [0, 0.01, 0.1, 0.5], dataset = dataset,n_runs
= 1)
plot(logs, x_key='step', y_key='aligned_ctr', legend_key='alpha', estimator='m
ean', ci=None)
##########################################
```

Out[31]: <matplotlib.axes._subplots.AxesSubplot at 0x7f79b4ecf5d0>



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

Answer: α adjust the weight of exploration, higher α implies giving more importance to exploration

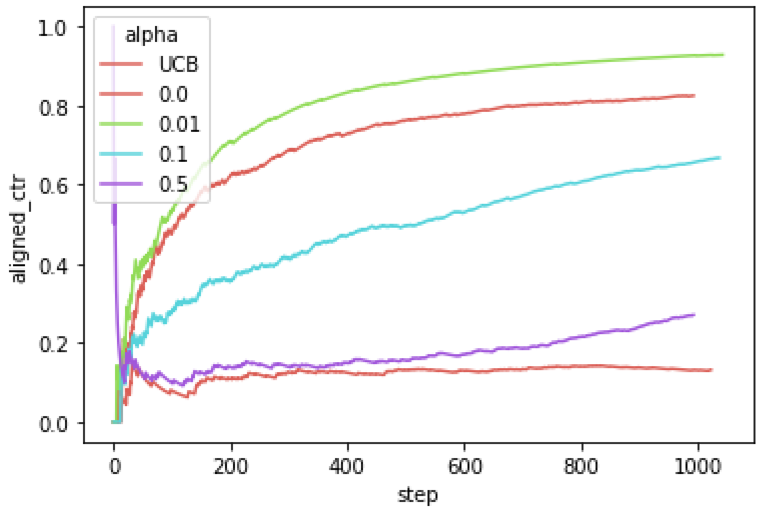
Question [5pts]: Do the reward curves change with α? If yes, why? If not, why not?

Answer: The reward curves change with α, since the weight of exploration versus exploration is different

In [32]:
```
#### TODO: modify the UCB agent in the multi-armed bandits, and compare the
#### aligned_ctr curves between bandit, and contextual bandit with alpha as 0,
0.01, 0.5  [10pts] ####

#### UCB Agent ####
@dataclass
class UCBIgnoreTextAgent:
    num_actions: int

    def __post_init__(self):
        self.reset()

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

    def update_params(self, action, reward, state):
        # Update Q action-value given (action, reward)
        #### TODO: Calculate the Q-value [5pts] ####
        if reward is None:
          return
        self.action_counts[action-1] += 1
        self.Q[action-1] = (self.Q[action-1]*(self.action_counts[action-1]-1)
+ reward)/self.action_counts[action-1]
        ###############################

    def get_action(self, state):
        self.t += 1
        ## HINT: To avoid a division by zero, you can add a small delta>0 to t
he denominator
        #### TODO: Calculate the exploration bonus [5pts] ####

        exploration_bonus = np.sqrt(4*np.log(self.t)/(self.action_counts+0.05
)) # placeholder
        ###########################################
        Q_explore = self.Q + exploration_bonus

        return np.random.choice(np.where(Q_explore == Q_explore.max())[0])+1


def ctxUCBbandit_sweep(dataset, n_runs=2000):
    logs = dict()
    agent = UCBIgnoreTextAgent(num_actions=10)
    engine = CtxBanditEngine(dataset=dataset, agent=agent)
    ep_log = engine.run(n_runs)
    ep_log = pd.concat(ep_log, ignore_index=True)
    ep_log['UCB'] = 'UCB'
    logs['UCB'] = ep_log
    logs = pd.concat(logs, ignore_index=True)
    return logs


UCBlog = ctxUCBbandit_sweep(dataset = dataset,n_runs = 1)
```

```
plot(UCBlog, x_key='step', y_key='aligned_ctr', legend_key='UCB', estimator='mean', ci=None)
plot(logs, x_key='step', y_key='aligned_ctr', legend_key='alpha', estimator='mean', ci=None)
```

Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x7f79b4baad50>



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

Answer: LinUCB outperform UCB, since its exploiting the states/information available. Here I implement UCB in the way that it ignores states. If UCB is implement in the way that each different state corresponds to a specific UCB, since we have 100 dimensions is highly unlikely that we have a lot of observations having the same state, and UCB agents will end up learning nothing at all.