Annam saivardhan
200050008
200050008@iitb.ac.in

Programming Assignment 1
Foundation of Intelligent and Learning Agents, 2022

2022-09-10

1 Task 1

1.1 Problem statement

In this first task, you will implement the sampling algorithms: (1) UCB, (2) KL-UCB, and (3) Thompson Sampling. This task is straightforward based on the class lectures. The instructions below tell you about the code you are expected to write.

Read task1.py. It contains a sample implementation of epsilon-greedy for you to understand the Algorithm class. You have to edit the __init__ function to create any state variables your algorithm needs, and implement give_pull and get_reward. give_pull is called whenever it is the algorithm's decision to pick an arm and it must return the index of the arm your algorithm wishes to pull (lying in 0, 1, ... self.num_arms-1). get_reward is called whenever the environment wants to give feedback to the algorithm: your code can use this feedback to update the data structures maintained by your agent. It will be provided the arm_index of the arm and the reward seen (0/1). Note that the arm_index will be the same as the one returned by the give_pull function called earlier. For more clarity, refer to single_sim function in simulator.py.

Once done with the implementations, you can run simulator.py to see the regrets over different horizons. Save the generated plot and add it to your report, with suitable commentary (ideally 4-5 lines describing what you see, what issues you faced, any surprising patterns). You may also run autograder.py to evaluate your algorithms on the provided test instances.

1.2 UCB

In UCB we need to pull the arm which has the highest empirical mean. But before doing that we should pull every arm once and calculate the empirical mean of it. We can do it by storing number of pulls we have made till know in a variable. We will be stroing empirical mean array, array which stores number of times an arm got pulled and an array to store the rewards it got us.

```
class UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        # You can add any other variables you need here
        # START EDITING HERE
        self.counts=np.zeros(num_arms)  #to store the number of times each arm is pulled
        self.ucbs=np.zeros(num_arms)   #to store ucb of each arm
        self.rews=np.zeros(num_arms)   #to store total rewards of each arm
        self.means=np.zeros(num_arms)  #to store empirical means of each arms
        self.pulls=0
        self.zero_ind=0
        # END EDITING HERE
```

This is the snippet of my init function for UCB.

Let us see the give_pull function now. At first it pulls every arm once and from then select the arm with maximum empirical mean.

```
    def give_pull(self):
        # START EDITING HERE
        if self.zero_ind<self.num_arms:
            return self.zero_ind
        else:
            return np.argmax(self.ucbs)
        # END EDITING HERE
```

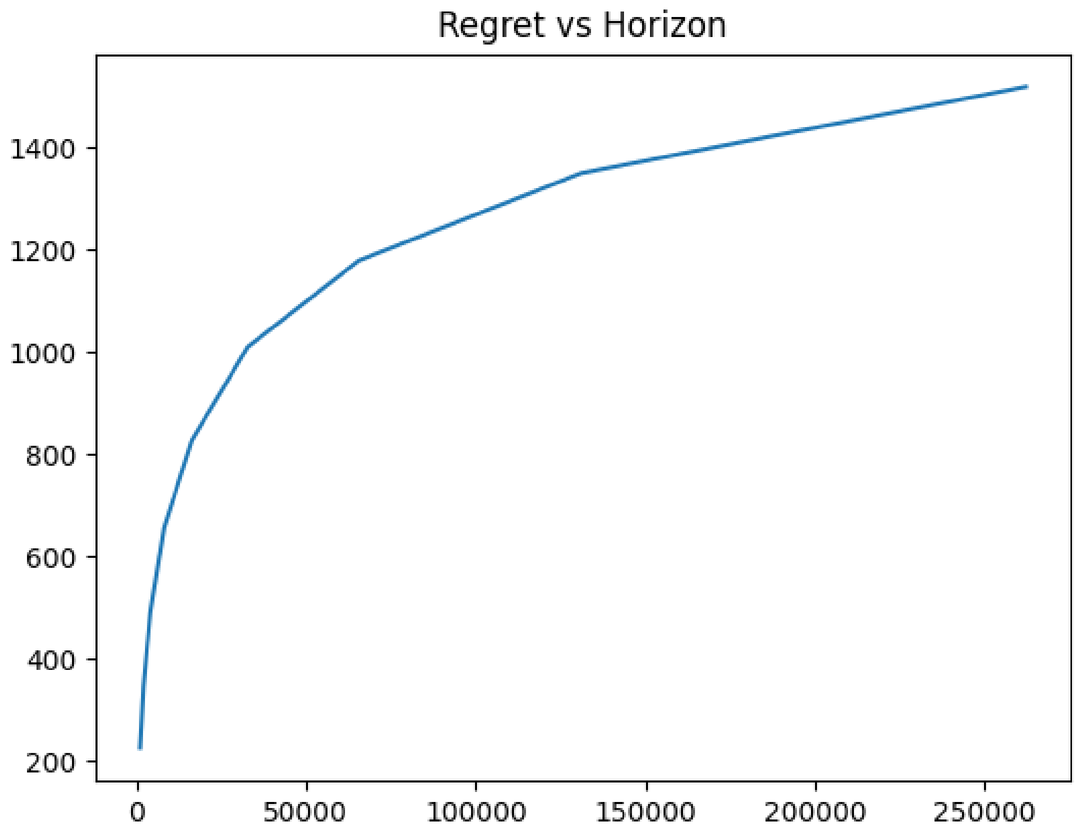
This is the snippet of my get_pull function for UCB.

Let us see the get_reward function now. Here we need to update the arrays which stores the means and counts. i have divided in to three conditions based on my requirements. Each condition is based on number of pulls we have done.

```python
    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        if self.zero_ind<self.num_arms-1:
            self.rews[arm_index]+=reward
            self.counts[arm_index]+=1
            self.pulls+=1
            self.zero_ind+=1
        elif self.zero_ind==self.num_arms-1:
            self.rews[arm_index]+=reward
            self.counts[arm_index]+=1
            self.pulls+=1
            self.zero_ind+=1
            self.means=np.divide(self.rews,self.counts)
            x=np.sqrt(2*math.log(self.pulls)/self.counts)
            self.ucbs=self.means+x
        else:
            self.counts[arm_index]+=1
            self.rews[arm_index]+=reward
            self.pulls+=1
            self.means=np.divide(self.rews,self.counts)
            x=np.sqrt(2*math.log(self.pulls)/self.counts)
            self.ucbs=self.means+x
```

This is the snippet of my get_reward function for UCB.

Let us see the plot of Regret vs Horizon we got by simulating UCB.



This is the plot of we get for UCB.

As we can see the plot is logarithmic with respect to N as expected

1.3 KL UCB

KL_UCB is similar to UCB but instead of chosing the arm with highest empirical mean we choose the arm with highest q(Upper Confidence Bound). We calculate q by using kl function and using a algorithm similar to binary search.Let us see the init function first. In init I am storing the counts, rewards, empirical means, kl_bounds(value of q) values of every arm.

```python
class KL_UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        # You can add any other variables you need here
        # START EDITING HERE

        self.counts=np.zeros(num_arms)  #to store the number of times each arm is pulled
        self.klucbs=np.zeros(num_arms)   #to store ucb of each arm
        self.rews=np.zeros(num_arms)   #to store total rewards of each arm
        self.means=np.zeros(num_arms)  #to store empirical means of each arms
        self.pulls=0
        self.zero_ind=0

```

This is the snippet of my init function for KL_UCB.

Give pull function in here is same as UCB except that we chose arm based on KL bounds rather than on empirical means and even the get reward function is similar to that of UCB with some slight modifications.

```python
    def give_pull(self):
        # START EDITING HERE
        if self.zero_ind<self.num_arms:
            #print(self.zero_ind)
            return self.zero_ind
        else:
            #print(np.argmax(self.klucbs))
            return np.argmax(self.klucbs)
        # END EDITING HERE
```

This is the snippet of my give_pull function for KL_UCB.

As I have mentioned it, I used a function which is very similar to binary search but not exactly same. Let us take a look at that function too. It returns the q value for a given arm and based on it we can select an arm to pull.

```python
def q_calc(pa,ua,rhs):
    if pa==1:
        return 1
    i=np.float64(pa)
    j=np.float64(0.999)
    rhs=rhs/ua
    while(j-i>=0.0001):
        z=(i+j)/2
        if(kl(pa,z)<rhs):
            i=z+0.0001
        else:
            j=z-0.0001
    return (i+j)/2


```

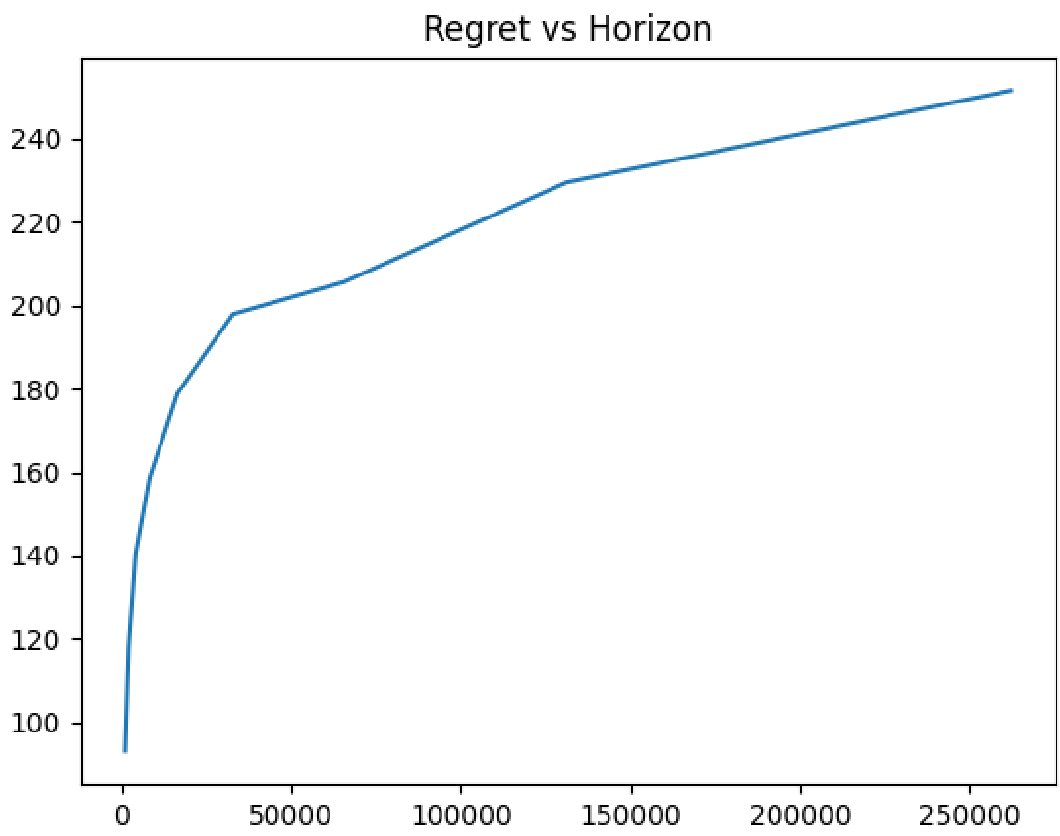
This is the snippet of my binary search function for KL_UCB.

Here is the formula to calculate upper confidence bound of an arm.

$$\text{ucb-kl}_a^t = \max\{q \in [\hat{p}_a^t, 1] \text{ such that } u_a^t KL(\hat{p}_a^t, q) \leq \ln(t) + c\ln(\ln(t))\}, \text{ where } c \geq 3.$$

KL-UCB algorithm: at step t, pull $\text{argmax}_a \, \text{ucb-kl}_a^t$.

Let us look at the plot of regret vs horizon we are getting for KL_UCB. It is also logarithmic as expected.



This is plot of Regret vs Horizon function for KL_UCB.

1.4 Thompson Sampling

This algorithm is quite different from UCB, KL_UCB. In this algorithm we store number of successes and failures each arm have. Then we construct a beta distribution on these two values for each arm. Then we take a random value from each distribution and we choose the arm with the highest random value. It is a non-deterministic algorithm whil the other two are deterministic.Let us look at the init function in thompson sampling.

```

class Thompson_Sampling(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        # You can add any other variables you need here
        # START EDITING HERE
        self.succs=np.ones(num_arms)
        self.fails=np.ones(num_arms)
        # END EDITING HERE
```

This is the snippet of my init function for thompson sampling.

The pull function is quite simple in here. We just need to select the arm which gets a higher random value from beta distribution we have. Let us see the code snippet of pull.

```python
    def give_pull(self):
        # START EDITING HERE
        arr=np.zeros(self.num_arms)
        for i in range(self.num_arms):
            arr[i]=np.random.beta(self.succs[i],self.fails[i])
        return np.argmax(arr)
        # END EDITING HERE
```

This is the snippet of my give_pull function for thompson sampling.

The get reward function is pretty straightforward. We just need to change the number of successes or failures of the arm we pulled. Let us see the code snippet for it.

```python
    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        if reward==1:
            self.succs[arm_index]+=1
        else:
            self.fails[arm_index]+=1
        # END EDITING HERE
```

This is the snippet of my get_reward function for thompson sampling.

Finally let us see the plot of Regret vs Horizon of Thomson's sampling

This is the plot of Regret vs Horizon function for thompson sampling.

2 Task2

2.1 Problem statement

We describe the problem statement that was briefly discussed completely now. The algorithm is given the number of arms of the bernoulli bandit num_arms, a horizon and a batch_size. The give_pull function will be called horizon/batch_size times (which can be assumed to be an integer). In every call, it must return the next batch_size number of pulls the algorithm wishes to make. The function must do so in a specific format: it has to return two lists, one containing the arm indices that it wishes to pull, and the other containing the number of times each of those indices must be pulled. For example, in a 10-armed bandit instance with batch_size 20, a possible return of the give_pull function could be ([2, 4, 9], [10, 4, 6]). Note that your function should generalize to arbitrary batch_sizes, as long as the given batch_size is a factor of the horizon (the batch_size could be just 1, or it could also be of the order of the horizon). The autograder/simulator will proceed and pull these arms according to their respective counts, and then provide feedback to the get_reward function. The feedback is provided as a dict, where the keys are the arm indices, and the rewards are a numpy array of 0s and 1s that were seen. So a possible input (arm_rewards) to the get_reward function for the above batch pull could be 2: np.array([1, 1, 1, 0, 1, 1, 0, 1, 0, 1]), 4: np.array([1, 1, 0, 0]), 9: np.array([0, 1, 0, 1, 0, 0]). Again, you can read single_batch_sim for more clarity. The regret is calculated over all the pulls over the horizon.
Once done with your implementation, you can run simulator.py to see the regrets for a fixed horizon over different batch sizes. Save the generated plot and add it to your report, with apt captioning. Take 4-5 lines (or more) to explain the trend you see, and justify your choice of the distribution of pulls in a given batch_size. You may also run autograder.py to evaluate your algorithms on the provided test instances.

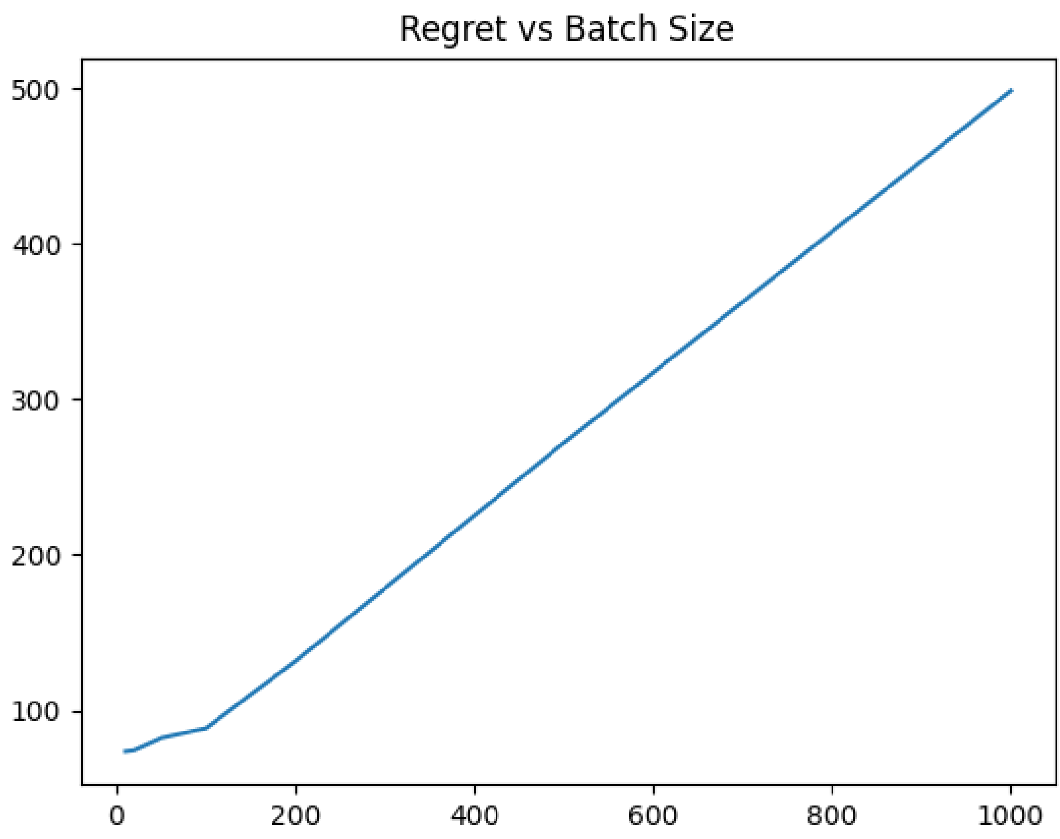
2.2 batch sampling

We can successfully do the batch sampling using the thompson's algorithm Instead of choosing just one arm we should select batch. So instead of doing one draw from beta distributions we will do multiple draws and store the arm with maximum random values in the array. Syntax is very similar to thompson.Let us look at the code of it.

```python
class AlgorithmBatched:
    def __init__(self, num_arms, horizon, batch_size):
        self.num_arms = num_arms
        self.horizon = horizon
        self.batch_size = batch_size
        assert self.horizon % self.batch_size == 0, "Horizon must be a multiple of batch size"
        # START EDITING HERE
        # Add any other variables you need here
        self.succs=np.ones(num_arms)
        self.fails=np.ones(num_arms)
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        l1=[]
        l2=[]
        init=[]
        arr=np.zeros(self.num_arms)
        for k in range(self.batch_size):
            for i in range(self.num_arms):
                arr[i]=np.random.beta(self.succs[i],self.fails[i])
            init.append(np.argmax(arr))
        a=cl.Counter(init)
        l1=a.keys()
        l2=a.values()

        return l1, l2
        # END EDITING HERE

    def get_reward(self, arm_rewards):
        # START EDITING HERE
        for arm_index,rewards in arm_rewards.items():
            a = np.sum(rewards)
            b = len(rewards)
            self.succs[arm_index]+=a
            self.fails[arm_index]+=(b-a)

        # END EDITING HERE
```

This is the code of batch sampling.

The plot of Regret vs Horizon for batch sampling looks like this



This is the plot of batch sampling.

3 Task3

3.1 Problem Statement

This task involves dealing with a bandit instance where the horizon is equal to the number of arms. So, for example, if there are 100 arms, then you are only allowed to pull 100 times. However, you are given that the arm means are distributed regularly (in arithmetic progression) between 0 and (1 - 1/numArms).
You need to come up with an algorithm to handle this situation effectively: can you do better than sampling each arm once? Implement your algorithm by editing the task3.py file. The APIs you need to implement are essentially the same as task1.py.
Once again, you can use simulator.py to see regrets as a function of horizon. You may also run autograder.py to evaluate your algorithm. Note that even if your algorithm passes the autograder tests, it might fail on the undisclosed tests that are used for evaluation. So you must not hardcode your method to make it work for only the given test instances. For this task, you will again plot regret against horizon (which is the same as the number of arms).

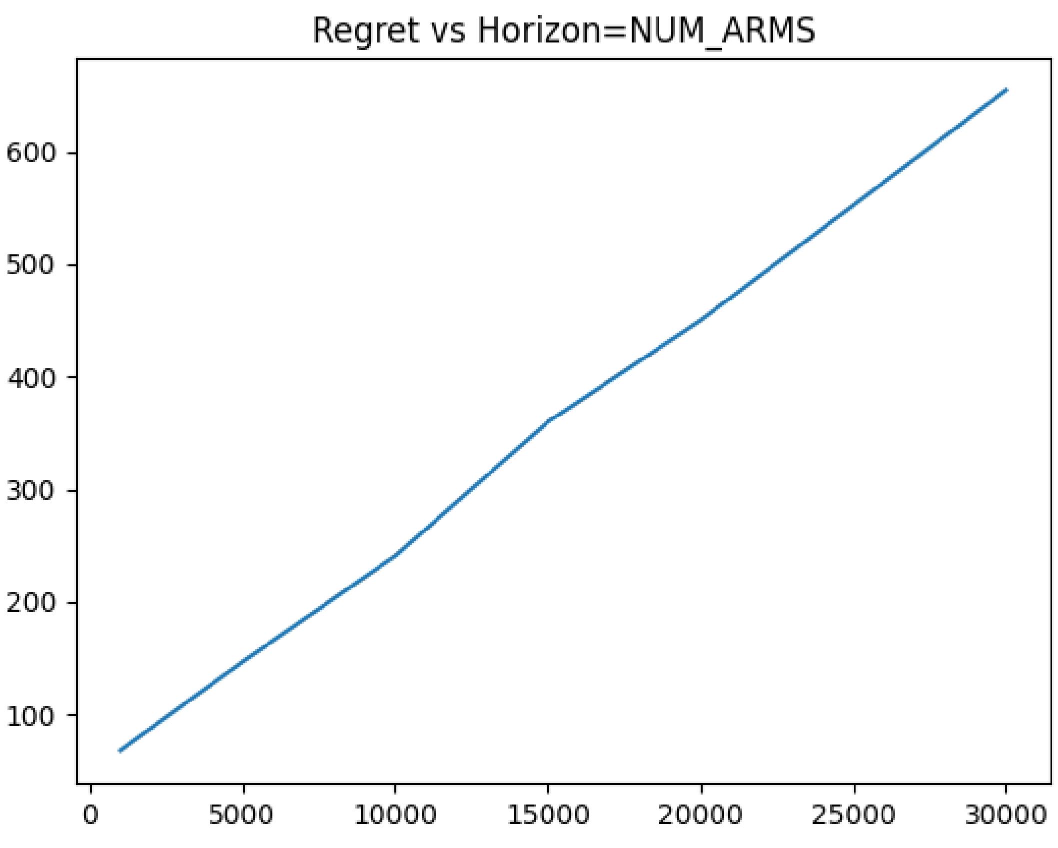
3.2 Many Arms

The problem where the number of arms and horizon equal is quite an interesting one. At first it might be tempting to apply UCB or Thompson or any other sub linear regret algorithms. But those are efficient only when the horizon is much greater than number of arms. So in this case it is best to use the basic epsilon greedy method. In here we explore for some amount of time and based on our exploration we exploit the remaining time. let us see the code of it.

```python
class AlgorithmManyArms:
    def __init__(self, num_arms, horizon):
        self.num_arms = num_arms
        # Horizon is same as number of arms
        self.eps = 0.01
        self.hz= self.eps*horizon
        self.pulls=0
        self.counts = np.zeros(num_arms)
        self.values = np.zeros(num_arms)

    def give_pull(self):
        # START EDITING HERE
        if self.pulls < self.hz:
            return np.random.randint(self.num_arms)
        else:
            return np.argmax(self.values)

        #raise NotImplementedError
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        self.pulls+=1
        self.counts[arm_index] += 1
        #n = self.counts[arm_index]
        #value = self.values[arm_index]
        #new_value = ((n - 1) / n) * value + (1 / n) * reward
        self.values[arm_index] = ((self.counts[arm_index]-1)/self.counts[arm_index])*self.values[arm_index] +(1/self.counts[arm_index])*reward
        # START EDITING HERE
        #raise NotImplementedError
```

This is the code of ManyArms.

The plot of Regret vs Horizon for ManyArms looks like this



This is the plot of ManyArms.