# CS747: Assignment-1 Report

Dharani Dadi

September 2023

# 1 TASK 1

## 1.1 UCB Method

### 1.1.1 Code Snippet

```
class UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        self.counts = np.zeros(num_arms)
        self.values = np.zeros(num_arms)
        self.time_step = 0
        self.epsilon = 10e-8

    def give_pull(self):
        if self.time_step < self.num_arms:
            return self.time_step
        else:
            ucb_values = self.values +
            np.sqrt((2*math.log(self.time_step))/(self.counts+self.epsilon))
            return np.argmax(ucb_values)

    def get_reward(self, arm_index, reward):
        self.time_step +=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] = new_value
```

### 1.1.2 Explanation

The number of arms in the multi-armed bandit is given by 'num_arms'. The 'counts' array has its elements as the number of times each arm is pulled, in the corresponding index of each arm. This is going to change everytime an arm is pulled. The 'values' array contains the estimate mean of each arm. This is also going to change everytime an arm is pulled. I initialized a variable 'time_step' as 0 which will be incremented when a pull is made. Epsilon is used to avoid 'zero in the denominator' errors.

In the give_pull function, when time instant is less than the number of arms, we pull the arms one by one so that the counts for all arms would be one. Once all the arms are pulled, everytime an arm has to be pulled, the ucb values of all the arms is calculated, and the index of the arm which has the highest ucb value is returned as output. The arm corresponding to the index returned here is pulled.

In the get_reward function, the index of the arm pulled and the reward received from that is taken as the input. The counts of the arm pulled is increased by 1, I did this by using the index of the arm pulled, and accessed the counts of this arm using the index and incremented it by 1. And the estimate mean of the arm also changes and it depends on the reward obtained by pulling the arm. So, the estimate mean(which is called values here) is also updated. And the time step is increased by 1.

### 1.1.3 Plot

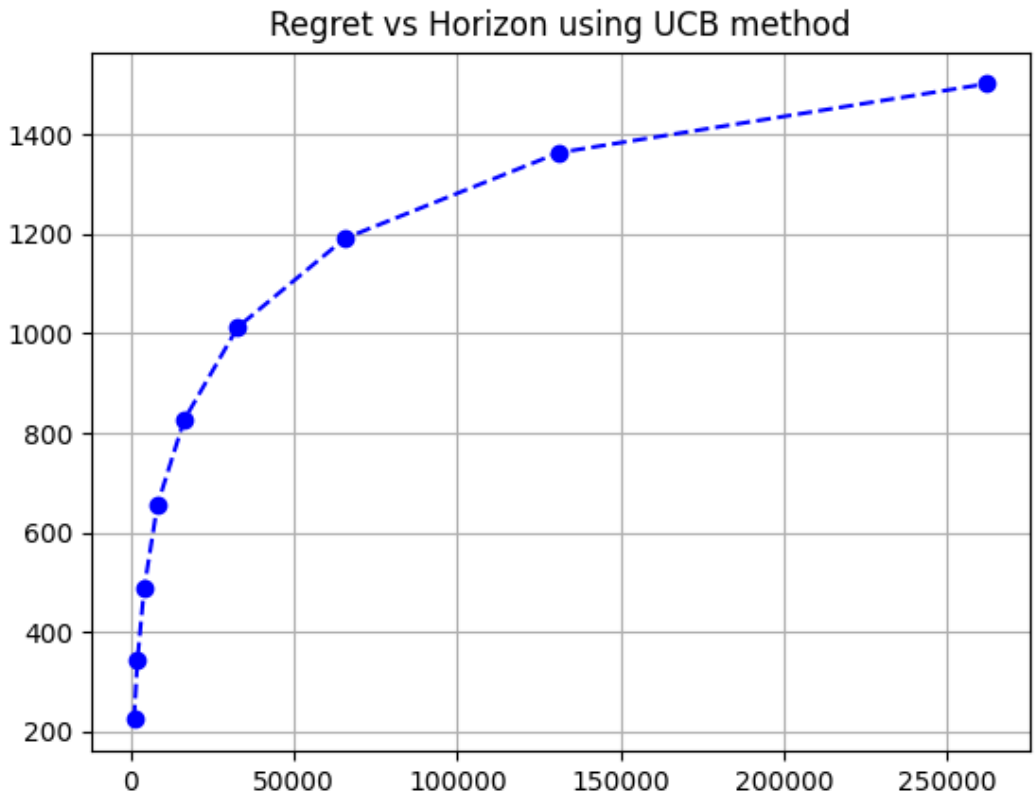


Figure 1: Regret vs Horizon using UCB Method

## 1.2 KL-UCB Method

### 1.2.1 Code Snippet

```
def KL(p, q):
    if p==q:
        return 0
    elif q==0 or q==1:
        return float("inf")
    elif p==1:
        return math.log(1/q)
    elif p==0:
        return math.log(1/(1-q))
    else:
        return p*math.log(p/q)+(1-p)*math.log((1-p)/((1-q)+10e-6))

def find_q(rhs, p_a):
    if p_a == 1:
        return 1
    q = np.arange(p_a, 1, 0.01)
    lhs = [KL(p_a, i) for i in q]
    lhs_array = np.array(lhs)
    difference = lhs_array - rhs
    indices_required = np.where(difference <= 0)[0]
    max_index = np.argmax(q[indices_required])
    return q[max_index]

class KL_UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        self.counts = np.zeros(num_arms)
        self.values = np.zeros(num_arms)
        self.time_step = 0
        self.epsilon = 10e-6

    def give_pull(self):
        if self.time_step < self.num_arms:
            return self.time_step
        else:
            kl_ucb_values = np.zeros(self.num_arms)
            for arm in range(self.num_arms):
                p_a = self.values[arm]
                rhs = math.log(self.time_step) / (self.counts[arm] +
                self.epsilon)
                kl_ucb_values[arm] = find_q(rhs, p_a)
            return np.argmax(kl_ucb_values)

    def get_reward(self, arm_index, reward):
        self.time_step +=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] = new_value
```

### 1.2.2 Explanation

The number of arms in the multi-armed bandit is given by 'num_arms'. The 'counts' array has its elements as the number of times each arm is pulled, in the corresponding index of each arm. This is going to change everytime an arm is pulled. The 'values' array contains the estimate mean of each arm. This is also going to change everytime an arm is pulled. I initialized a variable 'time_step' as 0 which will be incremented when a pull is made. Epsilon is used to avoid 'zero in the denominator' errors.

In the give_pull function, when time instant is less than the number of arms, we pull the arms one by one so that the counts for all arms would be one. For a value q in the range($p_a$,1), the maximum value of q such that KL($p_a$,q) $\leq (ln(t) + cln(ln(t)))/u_a$ gives the kl_ucb_value of a particular arm. Initially, I put kl_ucb_values for all arms as 0. We iterate over all the arms. In each iteration, the estimate mean of each arm is taken, the rhs of the inequality is calculated using the time step and the count of the arm, and the best q value for that particular arm is found. The q of an arm is found using the find_q function, where q is initiated as an array of values between $p_a$ and 1 with steps 0.01. KL(p,q) is given by $p * log(p/q) + (1-p) * log((1-p)/(1-q))$, and in the code, it is computed using the KL function. We iterate over all values of q and find the KL values between $p_a$ and each of the q values and store then in the array called 'lhs_array'. A new array called 'difference' is initiated which is the difference between the 'lhs_array' and the rhs value. Only the indices of the values less than 0 from the 'difference' array are considered in the q array, and among those, the highest value of q is returned at the end by the find_q function. The value returned by find_q function for each arm is its kl_ucb_value, and the arm with the highest kl_ucb_value is pulled.

In the get_reward function, the index of the arm pulled and the reward received from that is taken as the input. The counts of the arm pulled is increased by 1, I did this by using the index of the arm pulled, and accessed the counts of this arm using the index and incremented it by 1. And the estimate mean of the arm also changes and it depends on the reward obtained by pulling the arm. So, the estimate mean(which is called values here) is also updated. And the time step is increased by 1.

### 1.2.3 Plot

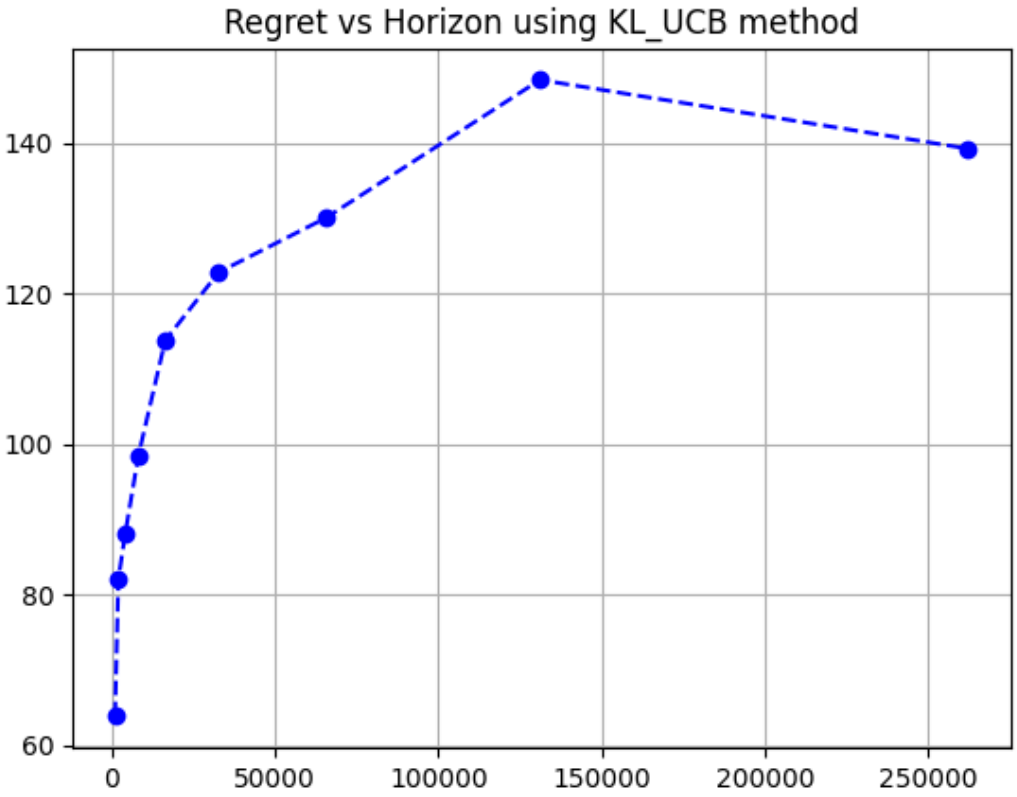


Figure 2: Regret vs Horizon using KL-UCB Method

## 1.3 Thompson Sampling Method

### 1.3.1 Code Snippet

```
class Thompson_Sampling(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        self.no_of_successes = np.zeros(num_arms)
        self.no_of_failures = np.zeros(num_arms)

    def give_pull(self):
        beta_sample = np.random.beta(self.no_of_successes+1,
        self.no_of_failures+1)
        return np.argmax(beta_sample)

    def get_reward(self, arm_index, reward):
        if reward == 1:
            self.no_of_successes[arm_index] += 1
        else:
```

```
            self.no_of_failures[arm_index] += 1
```

### 1.3.2 Explanation

I initiated the variable 'num_of_successes' which is an array of number of successes gotten by each arm, and the variable 'num_of_failures', which is an array of number of failures gotten by each arm.

In the give_pull function, the no. of successes and failures are used as parameters in the beta distribution function and samples are obtained for each arm. The arm whose sample value is the highest is pulled.

In the get_reward function, the arm index and reward are taken as input. If the reward is 1, no. of successes of that particular arm index is increased by 1, if not, the no. of failures of that particular arm index is increased by 1.

### 1.3.3 Plot

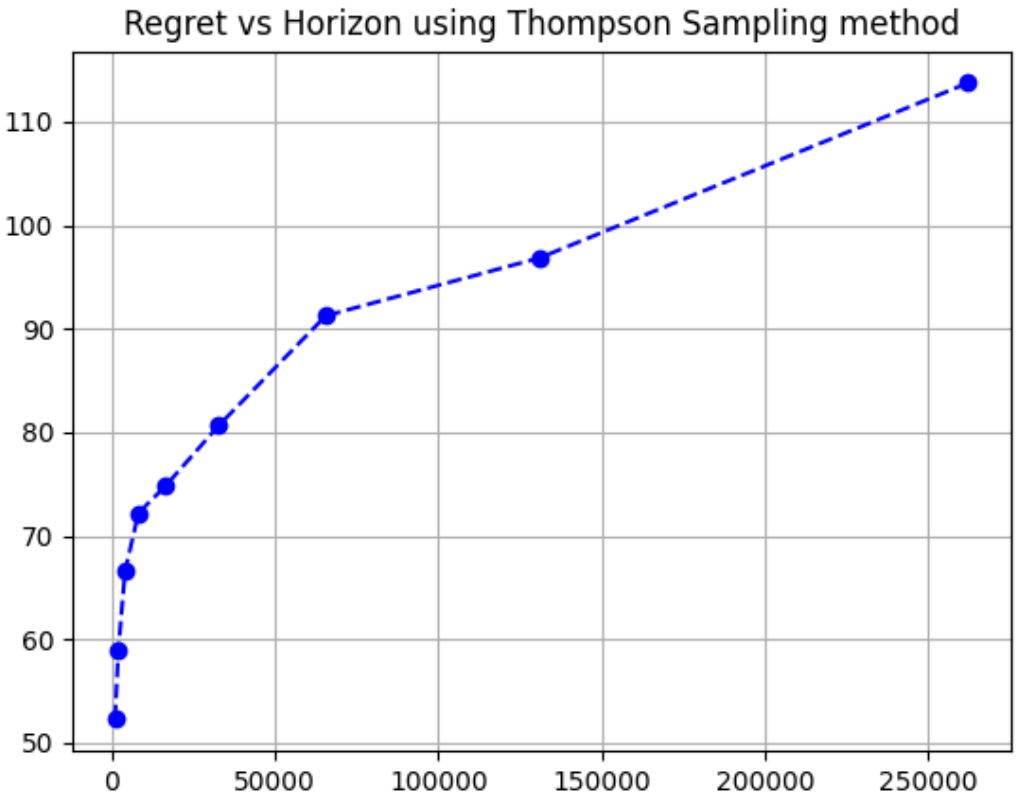


Figure 3: Regret vs Horizon using Thompson Sampling Method

# 2 TASK 2

## 2.1 Task-2A

### 2.1.1 Plot

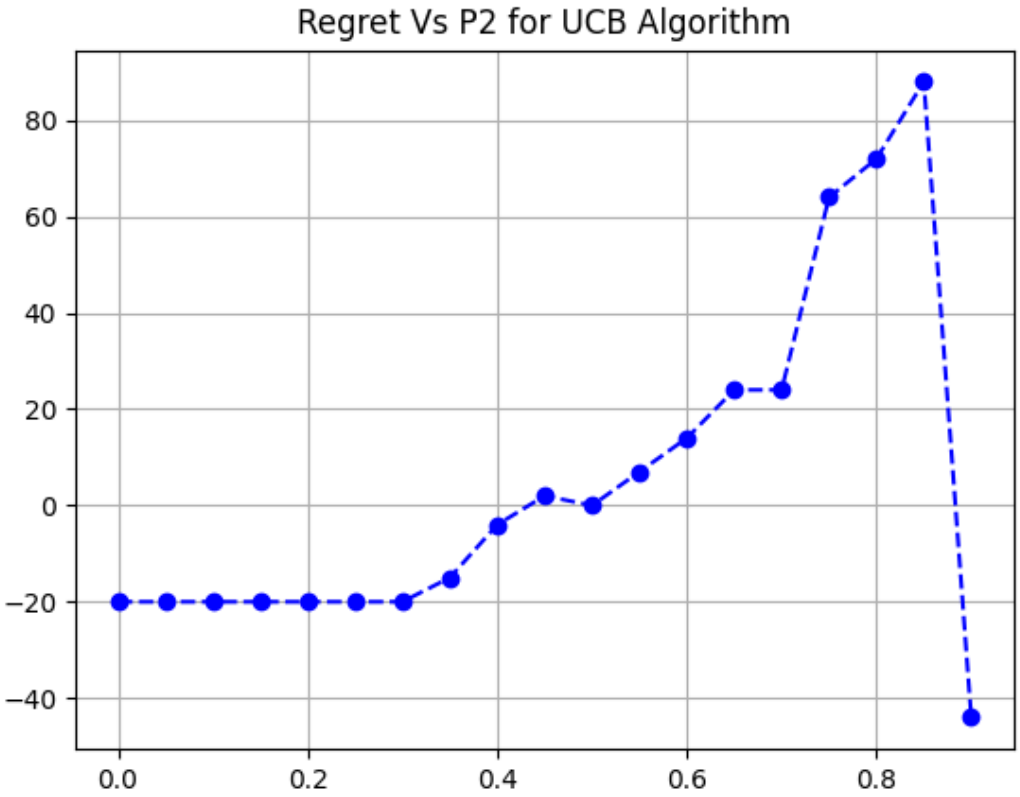


Figure 4: Regret vs p2 using UCB Method

### 2.1.2 Observations from the plot

The regret has been increasing until p2 is around 0.85, and then rapidly decreased at p2 = 0.9 or after that. The reason could be that, when p2 is between 0 and 0.85, the second arm's mean (p2) is lower than the first arm's mean, which is p1 = 0.9. The UCB algorithm initially explored both arms but must have realized that the first arm (p1) has a higher mean and must have started exploiting that arm more. And since the algorithm is primarily focused on p1, the regret kept increasing, because it missed out on the potential rewards from the second arm. When p2 reaches 0.9, it becomes equal to p1 and at this point, the algorithm has almost equal preference for either arm and hence starts exploring both the arms equally and finally reaches a point where it exploits the arms optimally. That is why, regret is suddenly decreased at the end.

## 2.2 Task-2B

### 2.2.1 Plot Obtained Using UCB Method

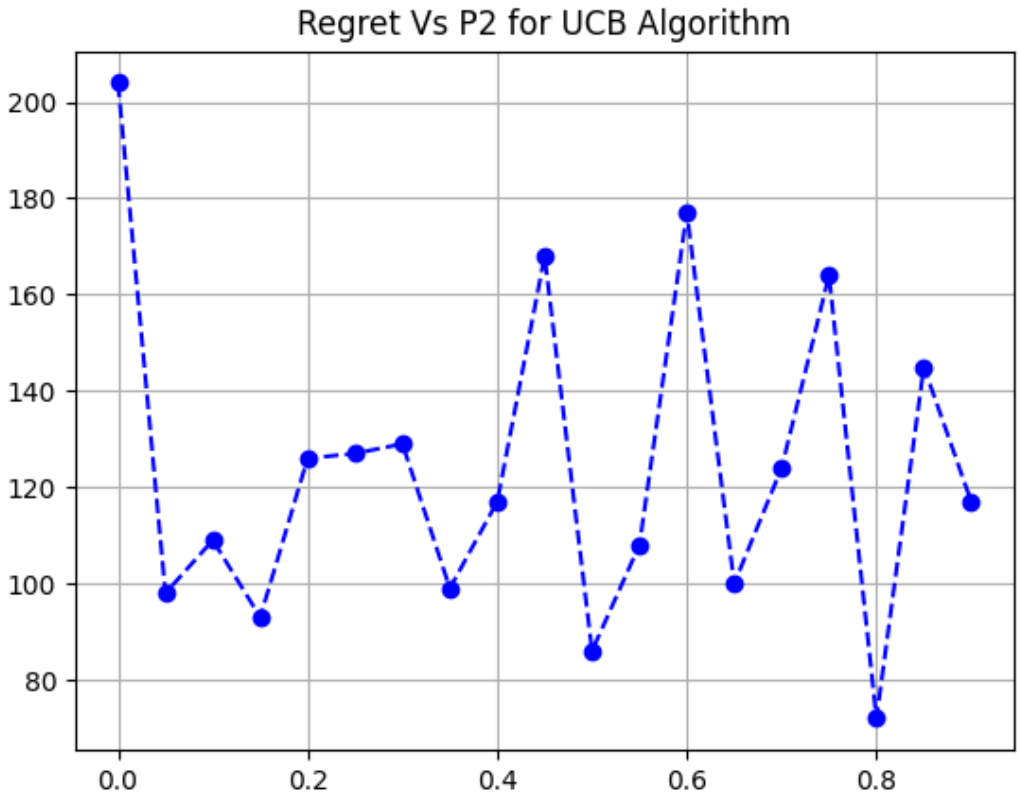


Figure 5: Regret vs p2 using UCB Method

### 2.2.2 Observation of the Plot

The pattern of the regret is zigzag and not a continuous increase or decrease pattern. This could possibly be because of the fact that p2 is very close to p1. Both arms having similar mean could must have made it harder for the UCB algorithm to distinguish between them, so the algorithm must have been switching between the arms, exploring one arm for a short time and switching to the other, resulting in zigzag patterns in the plot.

.

### 2.2.3 Using KL-UCB Method

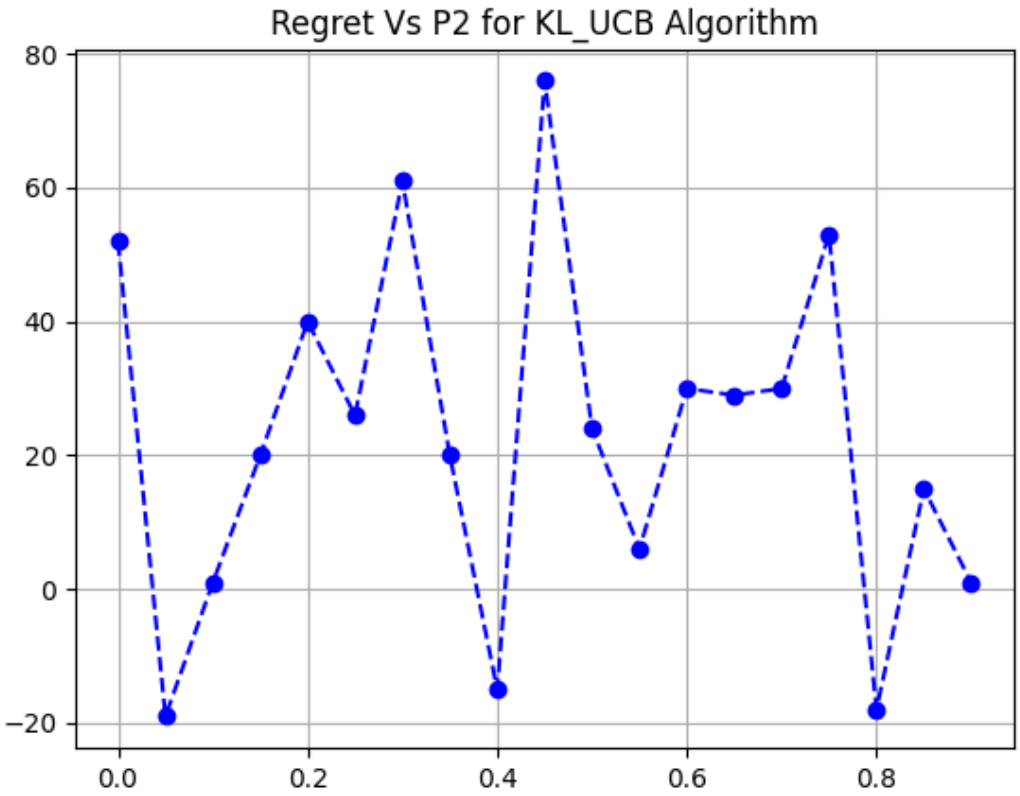


Figure 6: Regret vs p2 using KL-UCB Method

### 2.2.4 Observation of the Plot

The pattern of the plot in the case of KL-UCB algorithm is similar to the pattern observed in the case of UCB algorithm. The pattern of the regret is zigzag and not a continuous increase or decrease pattern. This could possibly be because of the fact that p2 is very close to p1. Both arms having similar mean could must have made it harder for the UCB algorithm to distinguish between them, so the algorithm must have been switching between the arms, exploring one arm for a short time and switching to the other, resulting in zigzag patterns in the plot.

## 2.3 Observation on differences observed in the plots made using UCB algorithm and KL-UCB algorithm

We know UCB algorithm tends to favor arms with higher estimated means due to it's use of confidence bounds. It usually tends to lean towards exploitation once it has some confidence on the estimates of the arms. This could possibly be the reason for slightly

lesser difference in the regret values and lesser height in the zigzag pattern observed in the plot as compared to the KL-UCB.

KL-UCB explicitly models the tradeoff between the exploration and exploitation problem using the KL-divergence. So it may explore arms more rigorously which is probably why the difference between regret in each step seems comparatively more as compared to the ones observed in UCB.

# 3 TASK 3

## 3.1 Code Snippet

```
class FaultyBanditsAlgo:
    def __init__(self, num_arms, horizon, fault):
        self.num_arms = num_arms
        self.horizon = horizon
        self.fault = fault
        self.no_of_successes = np.zeros(num_arms)
        self.no_of_failures = np.zeros(num_arms)

    def give_pull(self):
        beta_sample = np.random.beta((self.no_of_successes+1), (self.no_of_failures+1)
        return np.argmax(beta_sample)

    def get_reward(self, arm_index, reward):
        if reward == 1:
            self.no_of_successes[arm_index] += 1
        else:
            self.no_of_failures[arm_index] += 1
```

## 3.2 Explanation

I used the original Thompson Sampling algorithm itself because it gave comparatively better rewards than some other methods I tried.

# 4 TASK 4

## 4.1 Code Snippet

```
class MultiBanditsAlgo:
    def __init__(self, num_arms, horizon):
        self.num_arms = num_arms
        self.horizon = horizon
        self.no_of_successes = np.zeros((2, num_arms))
        self.no_of_failures = np.zeros((2, num_arms))

    def give_pull(self):
        beta_sample = np.random.beta(self.no_of_successes+1,
```

```
            self.no_of_failures+1)
        beta_sample_average = np.mean(beta_sample, axis=0)
        return np.argmax(beta_sample_average)

    def get_reward(self, arm_index, set_pulled, reward):
        if reward == 1:
            self.no_of_successes[set_pulled][arm_index] += 1
        else:
            self.no_of_failures[set_pulled][arm_index] += 1
```

## 4.2 Explanation

I first initiated two 2D numpy arrays named 'no_of_successes' and 'no_of_failures' which are of dimensions (2, number of arms). The 2 corresponds to the two multi-armed bandit instances being used here.

In the 'give_pull' function, the no. of successes and failures are used as parameters in the beta distribution function and samples are obtained for each arm. Since there are 2*number of arms in the samples obtained, we average the values along the columns (the values along the same column correspond to the same arm index in different bandit-instances), and then return the index of the arm whose average is the highest.

In the implementation of the code, first the instance of bandit being used is selected by uniformly sampling 0, 1 with 0.5 probability, where getting 0 corresponds to selecting the 1st bandit-instance and getting 1 corresponds to selecting the second bandit-instance. This bandit-instance selected is the 'set_pulled', which is also given as input to the 'get_reward' function along with arm index and reward. If the reward is 1, no. of successes of the corresponding bandit-instance and arm index is increased by 1, if not, the no. of failures of the corresponding bandit-instance and arm index is increased by 1.