# Reinforcement Learning

Vasu Soni

September 2023

## Contents

# 1 Introduction

# 2 Task-1

In this task we had to implement UCB, KLUCB and Thompson Sampling following are the implementations in python and their explanation

## 2.1 UCB Algorithm

```
import numpy as np


class UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        # START EDITING HERE
        self.counts = np.zeros(num_arms)
        self.values = np.zeros(num_arms)
        self.ucb_values = np.zeros(num_arms)
        self.time = 0
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        self.time += 1
        if self.time < self.counts.size:
            return self.time
        return np.argmax(self.ucb_values)
        #raise NotImplementedError
        # END EDITING HERE


    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        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
        if self.time >= self.counts.size:
            self.ucb_values = self.values + np.sqrt((2*
                math.log(self.time))/self.counts)
        # raise NotImplementedError
        # END EDITING HERE
```

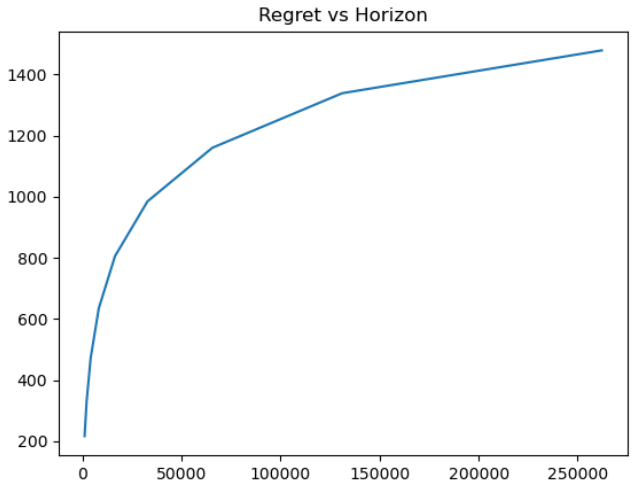


Figure 1: Figure showing Regret of UCB(y-axis)

In my code implementation, I employ three essential NumPy arrays to manage critical information for the multi-armed bandit problem.

- There is an array to track the number of times each arm has been pulled.
- Next, I maintain an array to store the empirical probability associated with each arm.
- Lastly, there's an array dedicated to computing and storing the UCB values for each arm, which guide the arm selection process.

To ensure temporal tracking, a 'time' variable keeps track on the progression of time.

In the `pull(self)` function, the logic is as follows: I increment the 'time' variable when an arm is pulled. During the initialization phase, if 'time' is less than the total number of arms, I return the 'time' since each arm must be pulled once to initialize UCB. Subsequently after that, I select the arm with the highest UCB value for the next pull.

Now, in the `get_reward(self, arm_index, reward)` function, the procedure can be outlined as follows: Upon pulling an arm and receiving a reward, I increment the respective arm's pull count. Following this, I update the empirical probability associated with the arm using a running average approach. Once the initialization phase has concluded, and all arms have been sampled at least once, I proceed to calculate the UCB values for each arm. This calculation relies on the UCB formula given in the slides.

## 2.2 KLUCB Algorithm

```
def KL_divergence(a, b):
    if(a == b):
        return 0
    elif(b == 1):
        return float('inf')
    elif a == 0:
        return (1-a)*math.log((1-a)/(1-b))
    elif a == 1:
        return a*math.log(a/b)
    else:
        kl = a*math.log(a/b) + (1-a)*math.log((1-a)/(1-b)
            )
        return kl
def binary_search(l,r,target,p):
    while True:
        if abs(l-r) < 0.001:
            break
        mid = (l+r)/2
        val = KL_divergence(p,mid)
        if val < target:
            l = mid
        elif val > target:
            r = mid
        if val == target:
            break
    return l
class KL_UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        self.values = np.zeros(num_arms)
        self.counts = np.zeros(num_arms)
        self.kl_ucb = np.zeros(num_arms)
        self.time = 0
        # START EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        self.time += 1
        if self.time < self.counts.size:
            return self.time
        return np.argmax(self.kl_ucb)
        # raise NotImplementedError
        # END EDITING HERE
```

```
def get_reward(self, arm_index, reward):
    # START EDITING HERE
    self.counts[arm_index] += 1
    n = self.counts[arm_index]
    new_value = (n-1)*self.values[arm_index]/n +
        reward*(1/n)
    self.values[arm_index] = new_value
    if self.time >= self.counts.size:
     for i in range(self.counts.size):
         l = self.values[i]
         r = 1
         target =(math.log(self.time))/self.counts[i]
         self.kl_ucb[i] = binary_search(l,r,target,
             self.values[i])
    # END EDITING HERE
```

In this algorithm, during the initialization phase, I have created three NumPy arrays and a time variable to facilitate the management of critical information:

- The first array stores the number of times each arm is pulled.
- The second array stores empirical probabilities for each arm.
- The third array stores KLUCB values for each arm.

In the `get.pull(self)` function, I increment the `time` variable and return the arm index with the maximum KLUCB value. However, if the `time` value is less than the number of arms, I return the `time` itself to account for the requirement that each arm must be pulled at least once during initialization.

In the `get.reward(self, arm_index, reward)` function, I first update the count value of the specified arm and then update the empirical probability associated with that arm. If all arms have been pulled at least once, I proceed to calculate the KLUCB value for each arm. This calculation involves a loop in which the KLUCB of each arm is determined using a binary search approach. The parameters passed to the `binarySearch` function include:

1. `left_value` (equal to the empirical probability).
2. `target_value`.
3. `right_value` (equal to 1).
4. `empirical_probability`.

In the `binarySearch` function, I have set the precision to be $10^{-3}$ and incorporated a nested function called `KL_divergence` to compute the Kullback-Leibler Divergence.

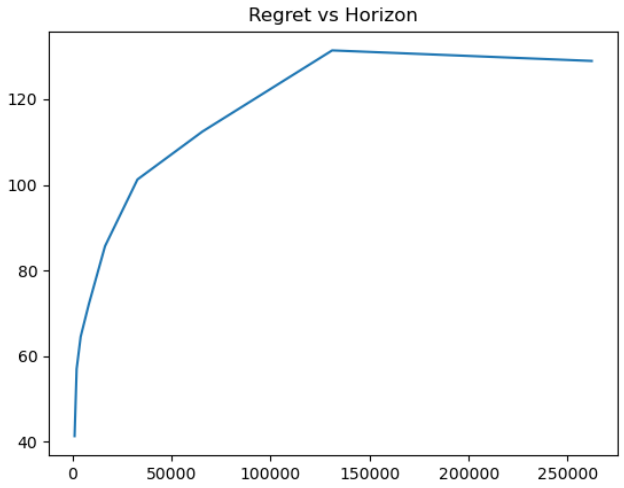


Figure 2: Figure showing Regret of KLUCB(y-axis)

### 2.3 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.heads = np.zeros(num_arms)
        self.counts = np.zeros(num_arms)
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        betas = np.zeros(self.heads.size)
        betas = np.random.beta(self.heads+1, self.counts-
            self.heads+1,self.heads.size)
        return np.argmax(betas)
        # raise NotImplementedError
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        self.counts[arm_index] += 1
        if(reward == 1):
            self.heads[arm_index] += 1
```

```
        # raise NotImplementedError
        # END EDITING HERE
```

In this algorithm, as in previous ones, I have created two arrays to track essential information for each arm:

- The first array is used to record the number of times each arm is pulled.
- The second array keeps track of the number of positive rewards (ones) obtained for each arm.

In the `get.pull(self)` function, I perform the following steps:

1. I generate a total of samples equal to the number of arms using the `np.random.beta()` function from a beta distribution.
2. I return the maximum value obtained from these samples.
3. The arguments provided to `np.random.beta()` are `1 + rewards` (equal to 1) and `1 + rewards` (equal to 0), utilizing the arrays defined during initialization.

In the `get.reward(self, arm_index, reward)` function, I take the following actions:

1. I simply increment the index corresponding to `arm_index` in the `heads` array if the received reward is 1.
2. I also increment the count array corresponding to `arm_index`.

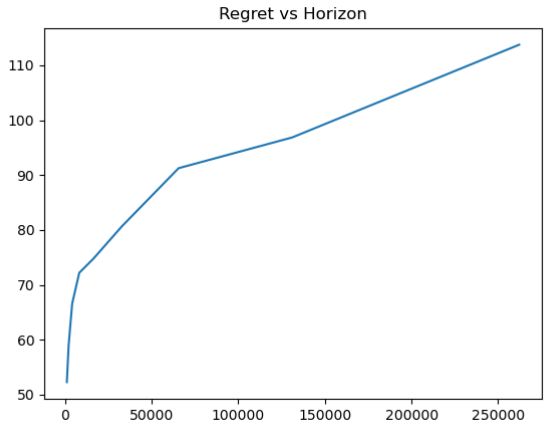


Figure 3: Figure showing Regret of Thompson Sampling(y-axis)

# 3 Task-2

Following are the graphs and results of the part 2:

## 3.1 Part A

The regret for UCB is increasing in part 2 as the probability of the second arm increases due to increased exploration of the second arm. This is because the difference between the probabilities of both arms is decreasing. Since regret is inversely proportional to the difference between the means of the optimal and suboptimal arms, the increasing similarity between the arms' probabilities contributes to the regret's growth. Then the sudden fall in regret is because at that moment both the arms have probability equal (=0.9) now both the arms are optimal so picking any arm won't make any difference hence expected loss is equal to zero. This can also be seen through the definition of Regret as follows:

$$\mathrm{R}_t = Tp^* - \sum E[r]$$
$$\sum E[r] = Tp^*$$

Hence we can see why the regret is equal to zero.

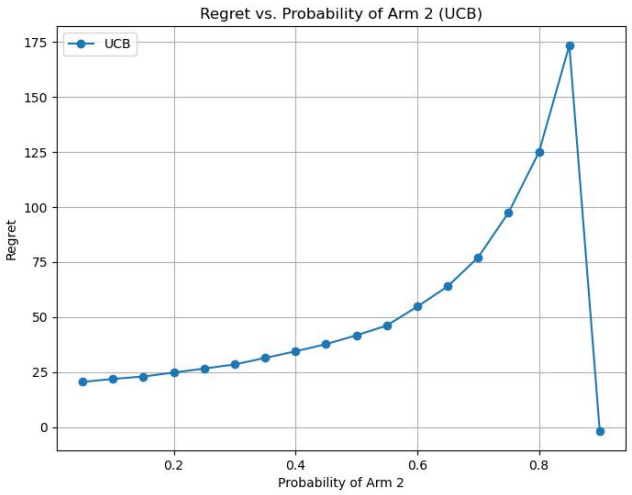


Figure 4: Regret vs Probabilty of Arm 2

## 3.2 Part B

Here we had to keep the difference between both arms constant and increase the probabilty of boths arms and get regrets for each arm. The Plots for the same are

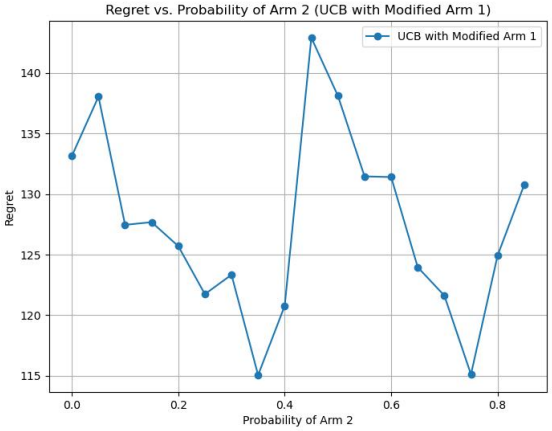


Figure 5: Regret vs Probabilty of Arm 2($\triangle_a$ fixed)

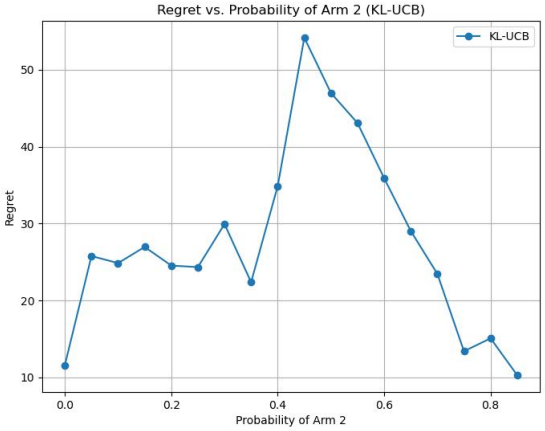


Figure 6: Regret vs Probabilty of Arm 2($\triangle_a$ fixed)

I observed some values of estimated regret, which correspond to different probabilities of the arms while keeping the means fixed. One thing I noticed about the KLUCB and UCB graphs was that regret was maximum for $P_a = 0.5$. The reason for this is that when we see the value of regret, it is:

$$\frac{R_T(L, I)}{\ln(T)} \geq \sum_{a:p_a(I)\neq p^\star(I)} \frac{p^\star(I) - p_a(I)}{KL(p_a(I), p^\star(I))},$$

Figure 8: Regret as Per Lai and Robbins

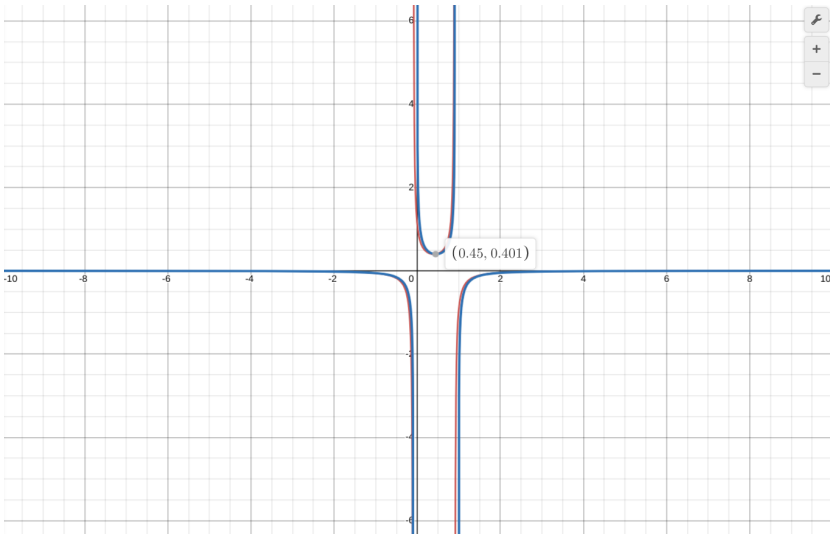


Figure 7: Solution for Maximum Regret

Since we have 2 arms only with one optimal, the summation is removed now. To find the highest regret, I differentiated the RHS and equated it to 0. After doing so, I got the solution for which the regret is maximum as $P_a = 0.45$, which is the probability of the sub-optimal arm.

As you can see, the solution is consistent with the graphs. The regret values for KLUCB are much smaller than UCB, which shows that KLUCB is a better algorithm for maximizing rewards. Graphs with different seed values are shown below:

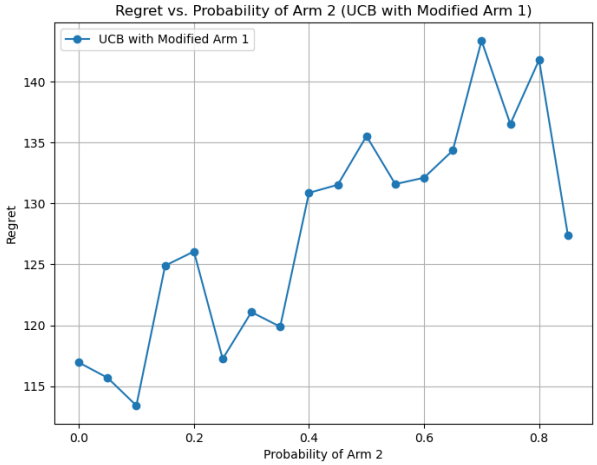


Figure 9: Regret vs Probabilty of Arm 2($\triangle_a$ fixed) different Seed

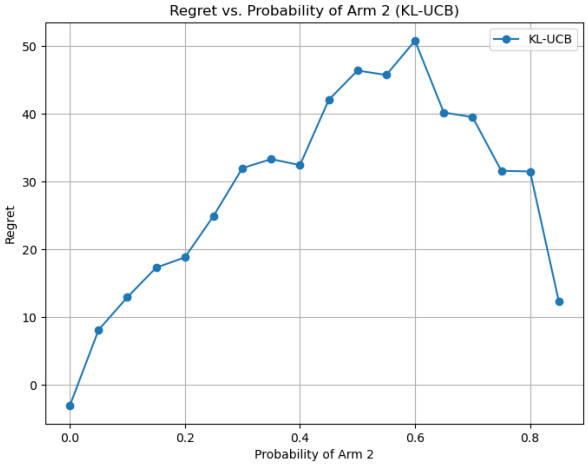


Figure 10: Regret vs Probabilty of Arm 2($\triangle_a$ fixed)different Seed

# 4 Task-3

I used Thompson sampling to maximize the reward for this task. I chose this algorithm because it allows us to identify faults embedded in the arms. We can make the assumption that for each arm, the equivalent probability is given by:

$$\hat{P}_a = (1 - \delta)P_a + \frac{\delta}{2}$$

So, we can consider replacing each arm with a new set of arms, where the probabilities are defined as above. By doing so, we can effectively address

and mitigate any potential faults, ultimately minimizing regret when using the Thompson sampling algorithm.

The code is used same as the one used in the task1 no changes.

# 5 Task-4

For implementing the algorithm, I use two beta distributions for the arm sets and select the arm with the maximum sum of samples from these beta distributions. This approach leverages the knowledge of the set from which the arm is being pulled, allowing us to effectively address the specific requirements of the task.

```
class MultiBanditsAlgo:
    def __init__(self, num_arms, horizon):
        # You can add any other variables you need here
        self.num_arms = num_arms
        self.horizon = horizon
        # START EDITING HERE
        self.counts1 = np.zeros(num_arms)
        self.counts2 = np.zeros(num_arms)
        self.heads1 = np.zeros(num_arms)
        self.heads2 = np.zeros(num_arms)
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        betas1 = np.random.beta(self.heads1+1,self.
            counts1-self.heads1+1,self.heads1.size)
        betas2 = np.random.beta(self.heads2+1,self.
            counts2-self.heads2+1,self.heads2.size)
        betas = betas1 + betas2
        return np.argmax(betas)
        #raise NotImplementedError
        # END EDITING HERE

    def get_reward(self, arm_index, set_pulled, reward):
        # START EDITING HERE
        if set_pulled == 0:
            if reward == 1:
                self.heads1[arm_index] += 1
            self.counts1[arm_index] += 1
        else:
            if reward == 1:
                self.heads2[arm_index] += 1
            self.counts2[arm_index] += 1
        #raise NotImplementedError
```

*# END EDITING HERE*

In the `get.pull(self)` function, I generate samples for both arm sets and select the arm index for which the sum of sample values is maximum. This strategy considers both arm sets in the decision-making process.

In the `get.reward(self, arm_index, set_pulled, reward)` function, I update the current conditions in the arrays `heads1` and `heads2`, which store the number of times a reward of 1 was observed, as well as `counts1` and `counts2`, which keep track of the number of times each arm was pulled.