# CS747-Assignment 1

Vansh Kapoor
200100164

# 1 Task 1

## 1.1 UCB Algorithm

I used the below implementation of the UCB-Algorithm. I first initialize the starting *time step* $t = 0$. `give_pull` function defined here first looks if there is any arm that hasn't been pulled yet and pulls the first arm it finds satisfying the condition, else it pulls the arm having the highest *Upper Confidence Bound* (UCB).

```
class UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        # START EDITING HERE
        self.values=np.zeros(num_arms,float)
        self.ucb=np.zeros(num_arms,float)
        self.counts=np.zeros(num_arms,int)
        self.t=0
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        self.t +=1
        if 0 in self.counts:
            return np.where(0==self.counts)[0][0]
        else:
            return np.argmax(self.ucb)
        # END EDITING HERE


    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        self.counts[arm_index]+=1
        n=self.counts[arm_index].astype(float)
        value=self.values[arm_index]
        self.values[arm_index]=((n - 1) / n) * value + (1 / n) * reward
        for i in range(int(self.num_arms)):
            if (self.counts[i]!=0):
                self.ucb[i]=self.values[i]+np.sqrt(2*np.log(self.t)/self.counts[i])
        pass
        # END EDITING HERE
```

After every arm pull we update the `self.counts`, `self.values` and `self.ucb` according to arm pulled and the reward received from the arm. `self.ucb` is updated with the sum of its expected mean reward and exploration bonus.

Figure 1 shows the variation of expected cumulative regret with Horizon for UCB algorithm. From Figure 1 we can verify that regret is sub-linear as expected, in fact the plot matches are expected result that Regret $R^T$ is of $\mathcal{O}(logT)$. But when compared to the other algorithms below the expected cumulative regret achieved is higher.

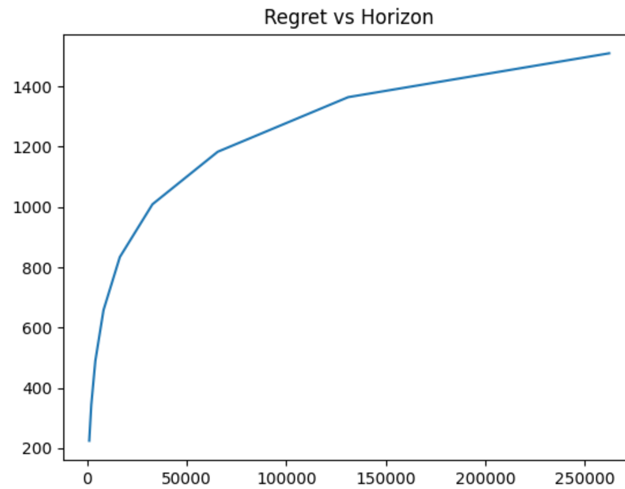


Figure 1: Regret VS Horizon for UCB algorithm

## 1.2 KL-UCB Algorithm

I have implemented `klucb` as a recursive algorithm that uses a sort of binary search to find *KL-Upper Confidence Bound* of each of the arms. Here I have set the **parameter** $c = 0$ as as I observed that this parameter setting brings down my regret. I have also set the **default sensitivity** **`sens`**$= e^{-12}$ for $KL - UCB$ (I am using a sensitivity of $= e^{-6}$) and have used an **error tolerance** of $e^{-5}$

```
def klucb(self,start,index,c=0,sens=1e-12,end=1.0):
    mean=(start+end)/2.0
    dest=(np.log(self.t)+ c*np.log(np.log(self.t)))/self.counts[index]
    sign=kl(self.values[index],mean)-dest
    if(end-start<=sens or np.abs(sign)<1e-5):
        return mean
    else:
        if (sign>0):
            return self.klucb(start=start,end=mean,index=index,c=c,sens=sens)
        else:
            return self.klucb(start=mean,end=end,index=index,c=c,sens=sens)
```

For implementing the function `kl` to calculate the KL-Divergence between its arguments. I covered all the edge cases that could occur while calling the function(e.g. the numerator of the argument given to the log function is zero)

```
def kl(a,b):
    if (np.abs(a)<=1e-12):
        s=-np.log(1-b)
    elif(np.abs(a-1)<=1e-12):
        s=-np.log(b)
    else:
        s=a*np.log(a/b)+(1-a)*np.log((1-a)/(1-b))
    return s
```

Figure 2 shows the variation of expected cumulative regret with Horizon for KL-UCB algorithm. From Figure 2 we can clearly conclude that the regret is logarithmic, i.e., $R^T$ is of $\mathcal{O}(logT)$. We also observe that the cumulative regret achieved for a given horizon is lower than that of $UCB$, which matches our theoretical expectations (It asymptotically achieves *Lia* and *Robbins* bound).

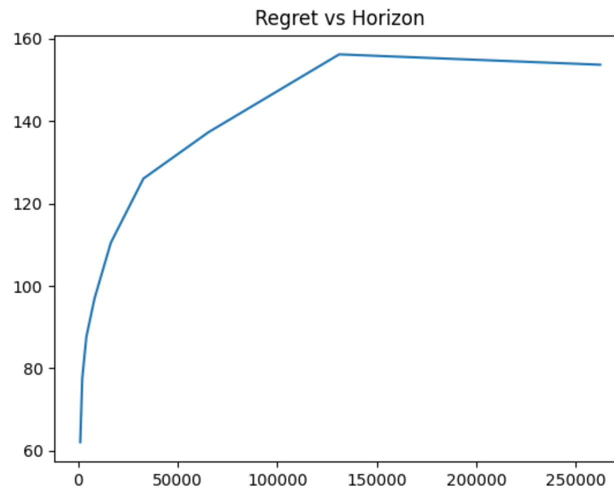


Figure 2: Regret VS Horizon for KL-UCB algorithm

## 1.3 Thompson Sampling

For implementing Thompson Sampling, I added an extra parameter `self.sampling` that contains the sampled values from the Beta function (arguments given by the successes and failures of the particular arm) from each of the arms. I have then selected the arm having the highest sampled value. I have initialized all my variables(`self.counts`, `self.values` and `self.sampling`) as zero arrays.

```python
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.values=np.zeros(self.num_arms,float)
        self.counts=np.zeros(num_arms,int)
        self.sampling=np.zeros(num_arms,float)
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        alpha=self.counts*self.values+1
        beta=(1-self.values)*self.counts+1
        for i in range(self.num_arms):
            self.sampling[i]=np.random.beta(alpha[i],beta[i])
        return np.argmax(self.sampling)
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        self.counts[arm_index] += 1
        n = self.counts[arm_index]/1.0
        value = self.values[arm_index]
        new_value = ((n - 1) / n) * value + (1 / n) * reward
        self.values[arm_index] = new_value
        pass
        # END EDITING HERE
```

Figure 3: Regret VS Horizon for Thompson Sampling

Figure 3 shows the variation of expected cumulative regret with Horizon for Thompson Sampling. From Figure 3 we can observe that the regret is clearly sub-linear as expected, in-fact we observe that it achieves a regret lower than KL-UCB! It can be thus concluded Thompson sampling is the best among the three algorithms.

# Task 2

## 1.4 Part A

If the mean of one of the arms is very small, we mainly rely on exploration bonus in calculating the UCB value for the arm with low mean reward. But the exploration bonus grows root over of logarithmic with time and hence we quickly learn about the approximate true means of the arms. As the $p2$ is increased we expect the regret to increase for a fixed horizon as it now takes more time steps to learn about the true mean of the arms and hence pulls the non-optimal arm more often, thus increasing regret for the same horizon. As the value of $p2$ is further increased we also expect a drop in regret since if the mean rewards of the arms get closer and closer the expected regret that we obtain at each time step is small even when we pull the non-optimal arm.

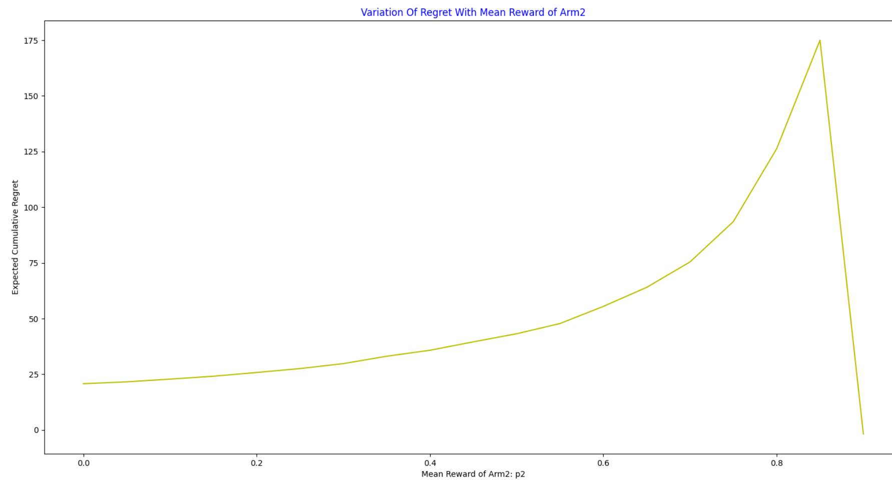


Figure 4: Effect of Difference Between Means of Arms On Regret by UCB Algorithm

## 1.5 Part B

Figure 5 shows how the Lia and Robbins bound constant $= p2 \log \frac{p2}{p2+0.1} + (1-p2) \log \frac{1-p2}{0.9-p2}$ behaves as p2 increases. The envelope of Regret curve in Figure 6 is approximately the plot of Figure 5 up to a constant! (X-axis in Figure 5 can be interpreted as mean reward of arm 2)
Since Horizon is kept constant throughout, we can understand the approximate behaviour of regret obtained by KL-UCB. The random unexpected spikes could be attributed to the finite horizon we are considering instead of an infinitely large horizon (since KL-UCB satisfies the Lia and Robbins bound asymptotically) or due to the randomness. As an approximation: Regret increases, reaches a maximum value near 0.4 and then decreases.

The plot for KL-UCB and UCB show some kind of similarity (UCB plot seems more random),

but the **regret value is higher for UCB** than KL-UCB which is expected. As the value of the mean reward for arm $p2$ increases, the regret accumulated initially increases. AS $p2$ increases further, there is a sharp dip near $p2 = 0.2$, then it peaks near $p2 = 0.4$ and has another local minima near $p2 = 0.8$. One important point to note is that both the plots have their **Regret maximized** near $p2 = 0.4$, which is where the **Lia and Robbins Bound** constant obtains its **maximum value**.

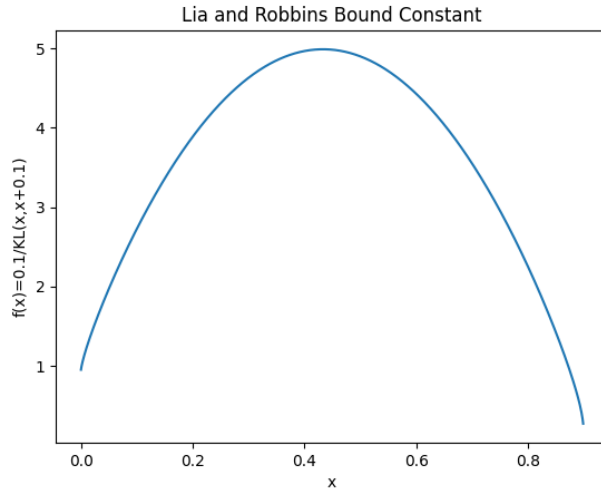


Figure 5: Constant factor in Lia and Robbins bound

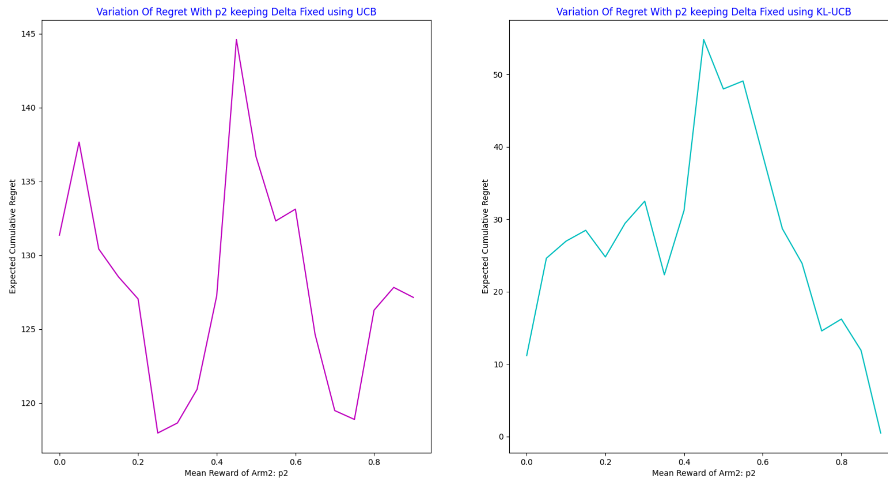


Figure 6: Variation of Regret with p2 keeping Delta constant for UCB and KL-UCB algorithms

## Task 3

Let us define $p$ to be the probability of giving a faulty output. Let the Belief of getting a reward at $m$ time step $m$ be $B_m$ for a particular . Then,
(i)If reward at time step $m+1$ is 1

$$B_{m+1}(w) \propto B_m(w)(\frac{p}{2} + (1-p)w)$$

(ii)If reward at time step $m+1$ is 0

$$B_{m+1}(w) \propto B_m(w)(\frac{p}{2} + (1-p)(1-w))$$

By observation, we get $B_m(w) = Beta_{s^t+1, f^t+1}(\frac{p}{2} + (1-p)w)\ dx$
Let $X$ Random Variable having a $Beta$ distribution with the above mentioned parameters, we now have sample a Random Variable $Y$ s.t.

$$Y = \frac{X - \frac{p}{2}}{1-p}$$

**Note:**$f_X(w) = Beta_{s^t+1, f^t+1}(w)$ and $f_Y(w) = Beta_{s^t+1, f^t+1}(\frac{p}{2} + (1-p)w)$

Thus instead of picking the arm which has obtained the maximum sample value while sampling over $Y$ with the arm parameters, we could also sample over $X$ (since its just a linear transformation and $p$ is constant for all arms) and pick the arm arm in a similar procedure. We are then basically doing **Thompson Sampling**!

## Task 4

Since our objective is to minimize cumulative regret and not any sort of variance, we could simply treat bandit arm as a black box and ignore the bandit instance that generated the reward. We then calculate a belief over the mean rewards of the arms and simply neglect the bandit instance that generated the reward. We then move to a simple multi-arm bandit problem and choose Thompson Sampling to minimize average cumulative regret.
In my implementation below, I have used `self.sucess` and `self.failure` to count the number of successes and failures of each of the arms of both the bandit-instances. I finally added the successes and failures and passed them as arguments to `np.random.beta` function. My `get_reward` functions similar to the one in Thompson Sampling.

```python
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
        self.sucess=np.zeros((2,num_arms))
        self.fail=np.zeros((2,num_arms))
        self.sampling=np.zeros(num_arms,float)

    def give_pull(self):
        # START EDITING HERE
        alpha=self.sucess+1
        beta=self.fail +1
        sampling_2=self.sampling.copy()
        lst=np.random.binomial(1, 0.5, self.num_arms)
        for i in range(self.num_arms):
            sampling_2[i]=np.random.beta(alpha[0,i]+alpha[1,i],beta[0,i]+beta[1,i])
        self.sampling=sampling_2
        return np.argmax(self.sampling)


        # END EDITING HERE

    def get_reward(self, arm_index, set_pulled, reward):
        # START EDITING HERE
        if(abs(reward)<1e-12):
            self.fail[set_pulled,arm_index]+=1
        else:
            self.sucess[set_pulled,arm_index]+=1
        pass
        # END EDITING HERE
```