CS747 - Foundations of Intelligent and Learning Agents

CS747 Autumn 2023

Chinmay Makarand Pimpalkhare
200100115

Assignment 1 | Online Learning Techniques for Cumulative Regret Minimization in Multi-Armed Bandits

1 Task 1

1.1 UCB

```
class UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        # START EDITING HERE
        self.counts = np.zeros(num_arms) #u_a^t in the algorithm
        self.emp_means = np.zeros(num_arms) #\hat{p}_a^t in the algotithm
        self.infinity = 10e10
        self.ucbs = self.infinity * np.ones(num_arms) #ucb_a^t in the algorithm
        self.total_count = 0
        # END EDITING HERE
```

This is the initialization function for the UCB algorithm. We need mainly the **empirical means** and the **ucb**s.

```
    def give_pull(self):
        # START EDITING HERE
        return np.argmax(self.ucbs)
        #raise NotImplementedError
        # END EDITING HERE
```

The give_pull function is simple, we just have to select the arm which has the maximum UCB.

```
        def get_reward(self, arm_index, reward):
        # START EDITING HERE
        #print("Total_count", self.total_count)
        self.total_count += 1
        self.counts[arm_index] += 1
        #print(self.counts)
        for arms in range(self.num_arms):
            if arms == arm_index:
                u = self.counts[arm_index]
                new_p = ((u - 1)/u)*self.emp_means[arm_index] + (1/u)*reward
                self.emp_means[arm_index] = new_p
                self.ucbs[arm_index] = self.emp_means[arm_index] + np.sqrt((2*np.log(self.
    total_count))/(u))
            else:
                if self.counts[arms] != 0:
                    self.ucbs[arms] = self.emp_means[arms] + np.sqrt((2*np.log(self.
    total_count))/(self.counts[arms]))


        #raise NotImplementedError
        # END EDITING HERE
```

We can now consider some important points. The **empirical means** are updated only when the arm corresponding to arm_index is pulled, however the **UCB**s get updated at every time step.

1.2 KL-UCB

We shall first define the helper functions. We have used **Continuous Binary Search (Bisection Method)** for computing the **KL-UCB**.

```
def kl_divergence(p,q):
    e = 10e-10
    kl = p*np.log((p + e)/(q + e)) + (1 - p)*np.log((1 + e - p)/(1 + e - q))
    return kl
```

The above function computes the KL-Divergence between the two input real numbers p and q. The 'e' has been entered to ensure that none of the terms in the argument of the logarithm go to 0 as this can cause scalar and division by 0 errors.

```
def binary_search(rhs, p, t):
    #Ideal conditions:
    #   Upper Cutoff = 0.5*np.log(t)
    #   Lower Cutoff = 0.1
    #   Epsilon = 0.1 if rhs > 1 else 0.05
    delta = 1/16
    upper_cutoff = 0.6*np.log(t) #Upper value of KL (Toggle)
    lower_cutoff = 0.1 #Lower value of KL (Toggle)
    epsilon = 1 #if rhs > 1 else 0.5 #Bisection Method Tolerance (Toggle)
    if rhs > upper_cutoff:
        #print("RHS: ", round(rhs,3), "greater than UC: ", round(upper_cutoff, 3),"max poss kld: ", round(kl_divergence(p,1), 3) )
        return 1
    elif rhs < lower_cutoff:
        #print("RHS lower than LC:", round(rhs,3),".Returning p: ", round(p,3))
        return p
    elif p == 1:
        return 1
    else:
        l = p
        r = 1
        m = 0.5*(l + r)
        counting = 0
        while np.abs(kl_divergence(p,m) - rhs) >= epsilon and np.maximum(np.abs(m - l), np.abs(m -r)) > delta:
            m = 0.5*(l + r)
            #print("p:", round(p,3), "Left: ", round(kl_divergence(p,l),3), "Middle: ", round(kl_divergence(p, m),3), "Right: ", round(kl_divergence(p,r), 3), "RHS: ", round(rhs,3), "BS_iter: ", counting)
            if (kl_divergence(p,l) - rhs)*(kl_divergence(p,m) - rhs) < 0:
                r = m
            else:
                l = m
        #print("RHS: ", round(rhs,3), "p:", round(p,3), "Returning m: ", round(m,3), "KLD: ", round(kl_divergence(p,m),3))
        return m
```

The next function defined above is the binary search function, which is essentially the continuous version. We also know it by the name bisection method. There are a lot of hyperparameters in the code, which determine how well the code works. The working of the binary search is just as the standard algorithm. The following stopping condition has been used as I observed best regrets with this selection.

```
while np.abs(kl_divergence(p,m) - rhs) >= epsilon and np.maximum(np.abs(m - l), np.abs(m -r)) > delta:
```

The stopping condition means that we stop if one of the following two conditions are met

1. A value of q has been found out such that KL(p,q) is within ϵ-range of the RHS
2. The interval over which the bisection method is working becomes smaller than a given constant δ

Other than that, since there are certain limitations to computing the KL-Divergence, essentially with respect to too large and too small values, we have set *upper-cutoff* and *lower-cutoff*. The lower-cutoff has been kept to a

constant value of 0.1, while the upper-cutoff has been kept a function of the time-step we are at, in the form $a_0 \times \log t$, where a_0 is some constant (good values seem to be in the range 0.6 to 0.8). Further, the regrets are depending heavily on the value of ϵ. Setting ϵ to too low values seems to cause some issues, and the regrets are actually becoming worse. Hence, it was found that $\epsilon = 1$ is giving good results.

```
def compute_kl_ucb(p, t, u, c, arm):
    if u == 0:
        print
        return 1
    elif u > 0 and t == 1:
        return 1
    else:
        rhs = (np.log(t) + c*np.log(np.log(t)))/(u)
        return binary_search(rhs, p, t)
```

Finally, this function integrates the above two and computes the RHS and then passes it through the binary search function. Also, if $u_{arm} = 0$, the KL-UCB will be 1.

```
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.totalcount = 0 #t in the algorithm
        self.counts = np.zeros(num_arms)
        self.p_hats = np.zeros(num_arms)
        self.kl_ucbs = np.ones(num_arms)
        self.c = 0
        #kl_plotter()#Remove this
        # END EDITING HERE*
```

Above is the initialization for the KL-UCB function. Here, one very important thing to note is that **c has been set to 0**, as recommended by the TAs and also the original paper. Indeed, c = 0 did give pretty good results!

```
    def give_pull(self):
        # START EDITING HERE
        return np.argmax(self.kl_ucbs)
        #raise NotImplementedError
        # END EDITING HERE
```

The give_pull function is similar to the one in case of UCB.

```
    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        #print("Totalcount", self.totalcount)
        self.totalcount += 1
        self.counts[arm_index] += 1
        self.p_hats[arm_index] = ((self.counts[arm_index] - 1)/(self.counts[arm_index]))*
    self.p_hats[arm_index] + (1/(self.counts[arm_index]))*reward
        for arms in range(self.num_arms):
            self.kl_ucbs[arms] = compute_kl_ucb(self.p_hats[arms], self.totalcount, self.
    counts[arms], self.c, arms)
        #raise NotImplementedError
        # END EDITING HERE
```

In case of KL-UCB, all the KL-UCBs have to be updated at every instant, and the update step is similar for all arms.

1.3 Thompson Sampling

This is by far, the easiest algorithms to implement.

```
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.totalcount = 0
        self.successes = np.zeros(num_arms)
        self.failures = np.zeros(num_arms)
        self.sample_x = np.zeros(num_arms)
        # END EDITING HERE
```

We note the successes and failures here.

```
    def give_pull(self):
        # START EDITING HERE
        for arms in range(self.num_arms):
            self.sample_x[arms] = np.random.beta(self.successes[arms] + 1, self.failures[arms] + 1)
        return np.argmax(self.sample_x)
        # END EDITING HERE
```

This is the give_pull function. It samples a number from the **beta**-distribution and the selects the arm which gives the maximum sampled number.

```
    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        if reward == 1:
            self.successes[arm_index] += 1
        if reward == 0:
            self.failures[arm_index] += 1
```

This is the update step. The nice thing is that we need to only update the arm which has been pulled, so this algorithm is the least computationally heavy of the three.

1.4 Results

```
    ================== Task 1 ==================
Testcase 1
UCB                : PASSED. Regret: 42.08
KL-UCB             : PASSED. Regret: 10.65
Thompson Sampling  : PASSED. Regret: 14.28

Testcase 2
UCB                : PASSED. Regret: 294.18
KL-UCB             : PASSED. Regret: 24.50
Thompson Sampling  : PASSED. Regret: 69.58

Testcase 3
UCB                : PASSED. Regret: 407.72
KL-UCB             : PASSED. Regret: 57.20
Thompson Sampling  : PASSED. Regret: 62.98

Time elapsed: 40.87 seconds
```

We can see the test cases. Although requiring quite some tuning of hyperparameters such as ϵ and δ, **KL-UCB** is performing the best, followed by Thompson Sampling. As expected, UCB has the highest regret.

1.5 Graphs

One thing common across all of these graphs is that they seem to be linear for higher horizon values. However, that is not the case. In actuality, the values of horizon are increasing in powers of 2 and hence they are sparser as we forward along the horizon axis. Hence, the interpolation is poorer.

1.5.1 UCB

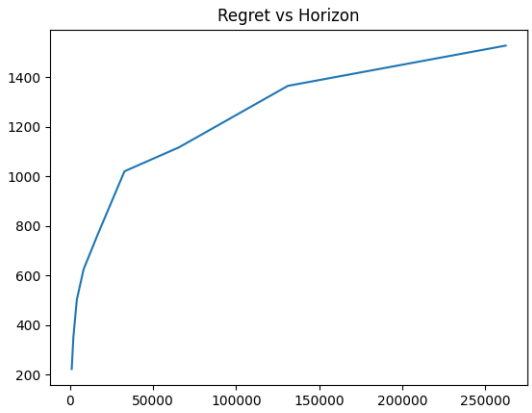

Figure 1: **UCB**. Regret on the Y-axis plotted as a function of horizon on the X-axis.

We observe **logarithmic regret**. As shown later, the values of regret are much higher than the other two algorithms. This is in alignment with the theory, since UCB does not match the constant specified by Lai and Robbins.

1.5.2 KL-UCB

Figure 2: **KL-UCB**. Regret on the Y-axis plotted as a function of horizon on the X-axis.

We observe **logarithmic regret**. There is something very interesting in the later part of the graph. The final data point appears to be an anomaly. The regret initially increases then falls down. We can attribute this to the following two explanations:

1. The data points selected for the horizon were not chosen uniformly, but in powers of 2. Hence this might just be an anomaly and the graph exaggerates the single data point.
2. The other possibility is that the horizon is very large, so the algorithm is almost surely selecting the optimal arm in the later stages. Further, the empirical mean for these later pulls is larger than the expected mean, hence the regret in these later pulls is actually negative.

1.5.3 Thompson Sampling

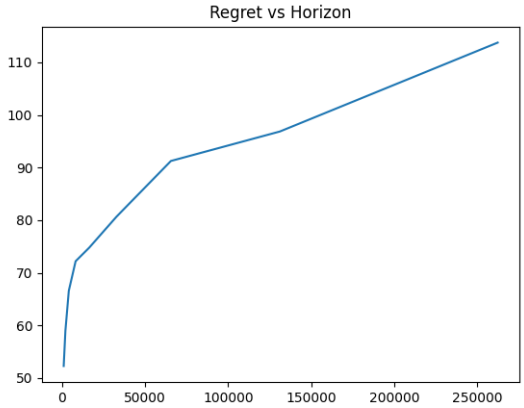

Figure 3: **Thompson Sampling**. Regret on the Y-axis plotted as a function of horizon on the X-axis.

We observe **logarithmic regret**. The values of regret are similar to that of KL-UCB, while they are much lower than those observed in case of UCB. This is because Thompson Sampling is optimal and also matches the constant given by Lai and Robbins.

2 Task 2A

In task 2A, we are studying the effect of varying differences in the means of the arms, we have kept p_1 to be constant at 0.9, and p_2 is being varied from 0 to 0.9, in increments of 0.05. The horizon has been set to 30000.

2.1 UCB (Graded)

```
Regret for p2 =  [0.0] is:  20.759999999989432
Regret for p2 =  [0.05] is:  21.5999999999897
Regret for p2 =  [0.1] is:  22.819999999990937
Regret for p2 =  [0.15] is:  24.119999999992032
Regret for p2 =  [0.2] is:  25.79999999999404
Regret for p2 =  [0.25] is:  27.55999999999517
Regret for p2 =  [0.30] is:  29.77999999999627
Regret for p2 =  [0.35] is:  33.09999999999932
Regret for p2 =  [0.4] is:  35.78000000000325
Regret for p2 =  [0.45] is:  39.56000000000844
Regret for p2 =  [0.5] is:  43.20000000001144
Regret for p2 =  [0.55] is:  47.78000000001781
Regret for p2 =  [0.60] is:  55.480000000029214
Regret for p2 =  [0.65] is:  64.02000000004365
Regret for p2 =  [0.70] is:  75.44000000005816
Regret for p2 =  [0.75] is:  93.56000000008318
Regret for p2 =  [0.8] is:  126.28000000011838
Regret for p2 =  [0.85] is:  175.0200000001196
Regret for p2 =  [0.9] is:  -1.8000000000130325
```

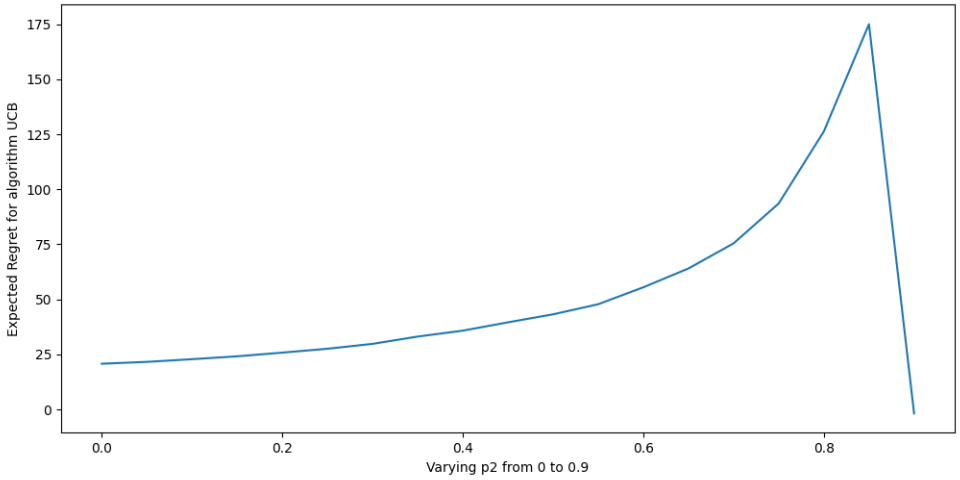

Figure 4: **UCB.** Regret plotted as a function of p_2, with p_1 kept constant

.

We observe a **quadratic** increase. The regret keeps on increasing upto 0.9, at which point there is an extremely sharp decrease, almost like a cusp.

Explanation: As the difference between the reward means goes to 0, the algorithm is much more susceptible to make mistakes. Even though the regret for each individual pull will be smaller, the algorithm will make sub-optimal pulls very frequently, hence over a large horizon, the regret will be high. If both the arms are equivalent in terms of reward means, then the expected regret will drop down to 0. It is going into the negative side because the empirical regret may be smaller than the expected regret, which is 0 in this case.

2.2 KL-UCB (Ungraded)

```
Regret for p2 =  [0.0] is:  1.8399999999859349
Regret for p2 =  [0.05] is:  1.9599999999858724
Regret for p2 =  [0.1] is:  1.9999999999859548
Regret for p2 =  [0.15] is:  2.0399999999859753
Regret for p2 =  [0.2] is:  2.0999999999859544
Regret for p2 =  [0.25] is:  2.119999999985787
Regret for p2 =  [0.30] is:  2.0399999999858087
Regret for p2 =  [0.35] is:  1.9799999999858222
Regret for p2 =  [0.4] is:  2.059999999985652
Regret for p2 =  [0.45] is:  1.839999999985759
Regret for p2 =  [0.5] is:  2.219999999985019
Regret for p2 =  [0.55] is:  1.5599999999858878
Regret for p2 =  [0.60] is:  1.2799999999861538
Regret for p2 =  [0.65] is:  2.0999999999850862
Regret for p2 =  [0.7] is:  2.2799999999848217
Regret for p2 =  [0.75] is:  3.659999999985694
Regret for p2 =  [0.8] is:  51.90000000001894
Regret for p2 =  [0.85] is:  92.96000000001864
Regret for p2 =  [0.9] is:  -1.8000000000130325
```

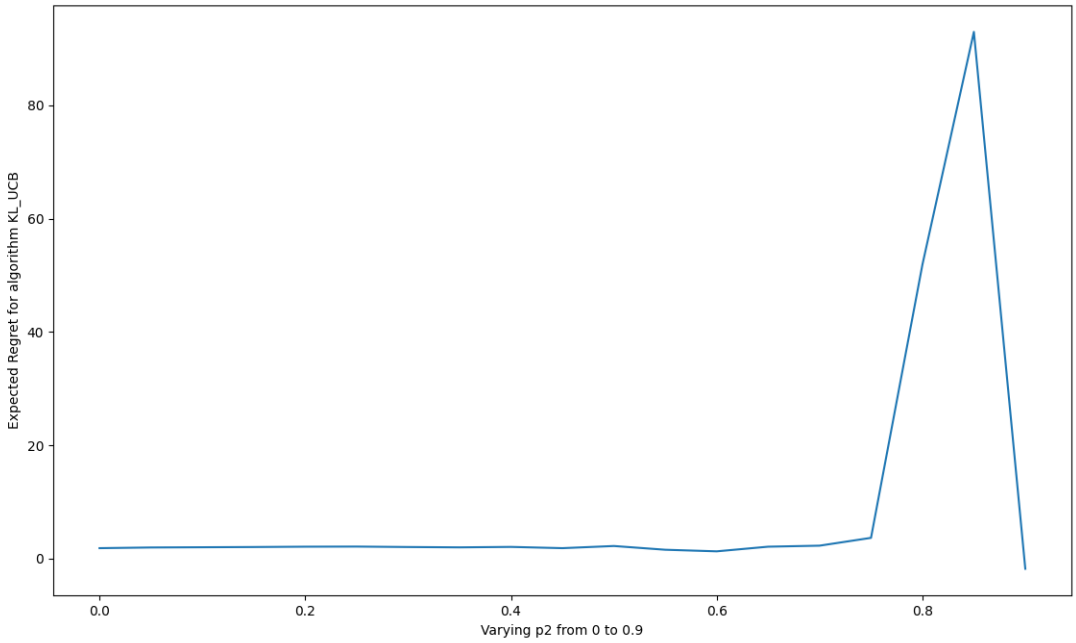

Figure 5: **KL-UCB.** Regret plotted as a function of p_2, with p_1 kept constant

.

We observe somewhat like an **exponential** increase. Interestingly, the nature of this graph is like the nature of the graph of KL Divergence. The regret keeps on increasing upto 0.9, at which point there is an extremely sharp decrease, almost like a cusp.

Explanation: As the difference between the reward means goes to 0, the algorithm is much more susceptible to make mistakes. Even though the regret for each individual pull will be smaller, the algorithm will make sub-optimal pulls very frequently, hence over a large horizon, the regret will be high. If both the arms are equivalent in terms of reward means, then the expected regret will drop down to 0. It is going into the negative side because the empirical regret may be smaller than the expected regret, which is 0 in this case.

3 Task 2B

3.1 UCB

The expression for the UCB is such that the empirical mean is directly added to the exploration bonus. Further, the exploration bonus term is in some sense independent of the empirical mean. Hence, in UCB, the magnitude ordering of the arm means is very important.

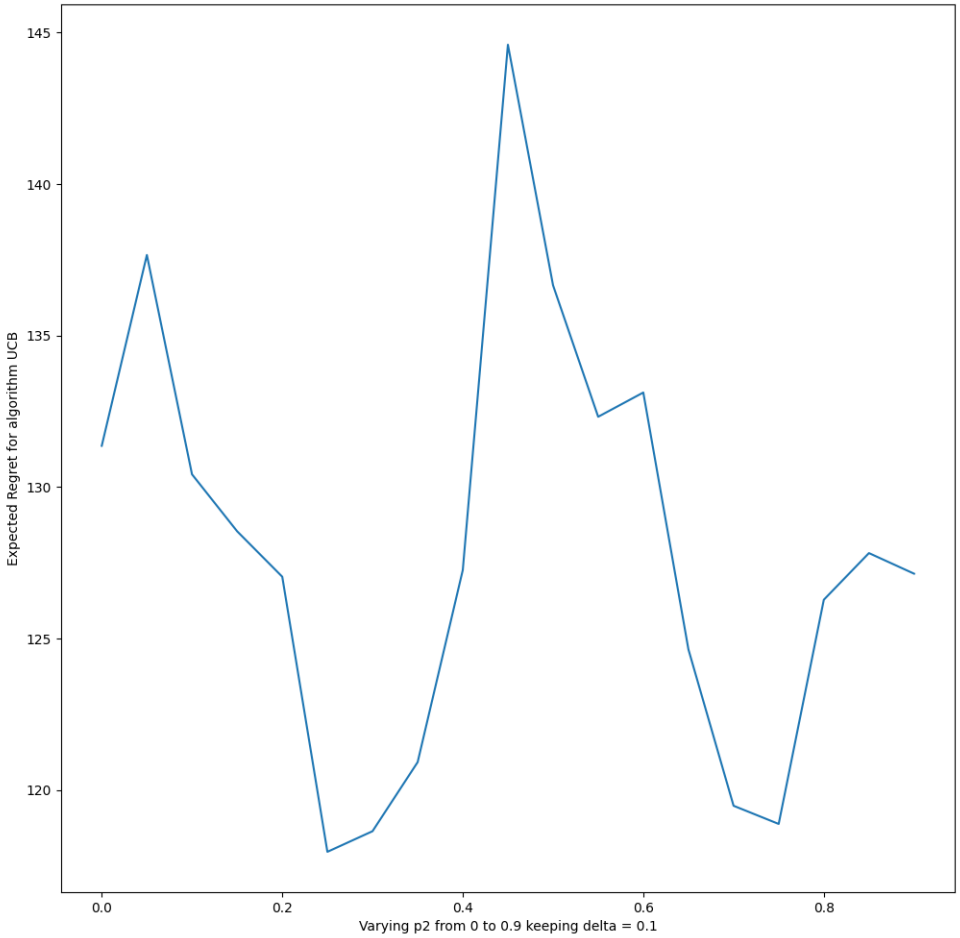

Figure 6: **UCB.** Regret plotted as a function of p_2, with $p_1 - p_2 = 0.1$ kept constant

.

The important thing to note is the scale within which the regrets lie. We see that **all the regrets** lie within the range 120 to 140, i.e. **the regrets have very low variance**. The overall trend seems to be somewhat random.

Explanation

1. Rank ordering of the arms is more important in UCB, than the actual magnitudes
2. The difference between the arms plays a much larger role as compared to the actual means, since if we take the difference of the UCBs, we will get a term of the form $\Delta + e_1 - e_2$, where e_1 and e_2 are exploration bonuses.

3.2 KL-UCB

Since KL-UCB uses an implicit expression for the UCB, there is no split into a dependent and an independent term. The KL-UCB difference between two arms is hence sensitive to both the magnitudes of the empirical means

as well as their differences.

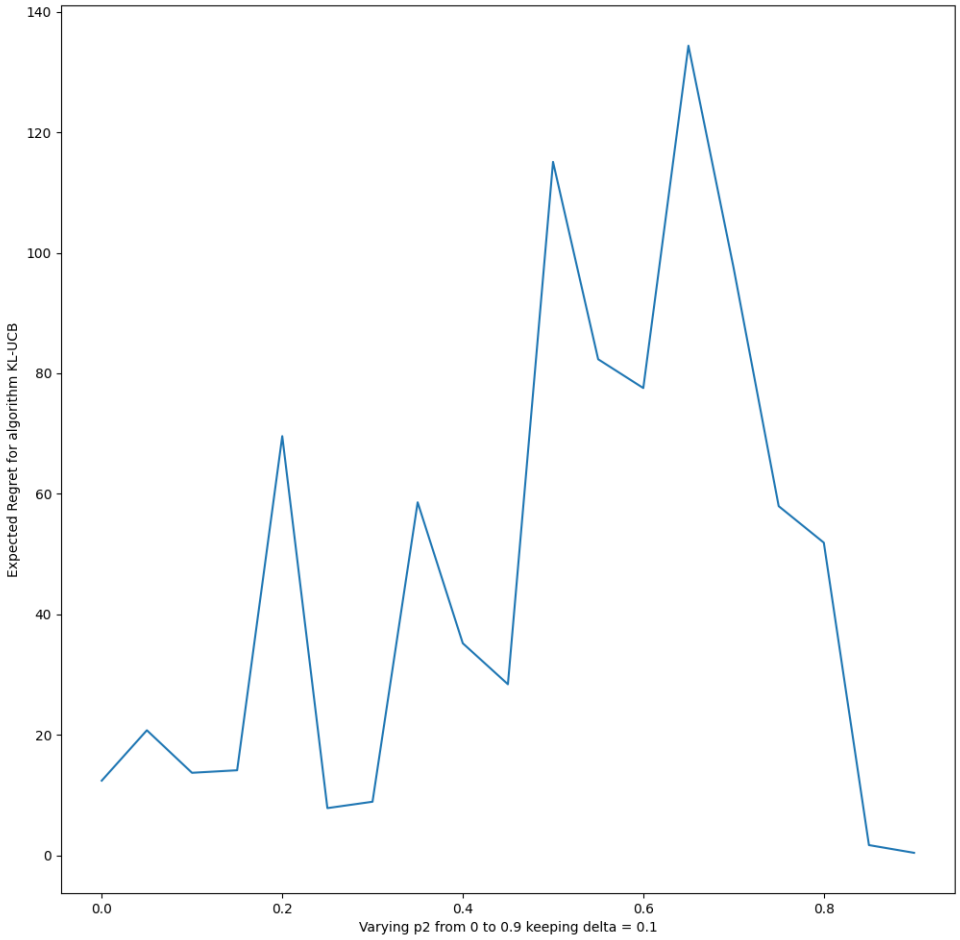

Figure 7: **KL-UCB.** Regret plotted as a function of p_2, with $p_1 - p_2 = 0.1$ kept constant

.

There is a stark difference as compared to the UCB graph.
Main Observations and Explanations

1. When $p_1 = 0$ or $p_2 = 1$, the regrets are very low. This is since the KL-UCB for an arm with empirical mean very close to 1 will almost surely be 1.
2. The **variance of regrets is much higher** than in case of UCB
3. Thus, we see sensitivity to the magnitude of the empirical means

4 Task 3

Here, we have been given a faulty bandit instance. The base algorithm which I have used is **KL-UCB**, subject to a few modifications. Let us formulate the problem mathematically. Consider a bandit instance with expected mean p and probability of fautl f. Then, the adjusted expected mean p_{adj} due to the presence of the fault shall be

$$p_{adj} = p \times (1-f) + 0.5 \times f$$

This is a linear transformation in p. Now, consider the case where we have two arms with expected means p and q, then their adjusted means will be given by

$$p_{adj} = p \times (1-f) + 0.5 \times f$$

$$q_{adj} = q \times (1-f) + 0.5 \times f$$

Note that

$$p > q \rightarrow p_{adj} > q_{adj}$$

Hence we should we able to use KL-UCB effectively. Further note that

$$\Delta = p_{adj} - q_{adj} = (p-q) * (1-f)$$

Also, note that the relation between the adjusted means and the empirical means $\bar{p}, \bar{q}$ will now be the following:

$$p_{adj} = \mathbb{E}[\bar{p}]$$

$$q_{adj} = \mathbb{E}[\bar{q}]$$

Thus, the fault has reduced the difference between the two means. In case the means were fixed, this would increase the regret, as shown in task 2A. We can consider two different approaches

1. Use directly the empirical means and pass then into the KL-UCB algorithm
2. Get the true values of the non-faulty expected means through the transformation $x \rightarrow (x - 0.5 \times f)/(1-f)$ on the empirical mean, where f is the fault probability. Then pass the non-faulty means into the computation for KL-UCB.

Most of the code has been adapted from Task 1 in KL-UCB and $\mathbf{c = 0}$ has been used here as well. The main change is in the get_reward function.

4.1 Approach 1

Passing the empirical means without any processing

```
    def get_reward(self, arm_index, reward):
    self.totalcount += 1
        self.counts[arm_index] += 1
        self.p_hats[arm_index] = ((self.counts[arm_index] - 1)/(self.counts[arm_index]))*
    self.p_hats[arm_index] + (1/(self.counts[arm_index]))*reward
        #self.adjusted_p[arm_index] = (1 - self.fault)*self.p_hats[arm_index] + 0.5*self.
    fault
        for arms in range(self.num_arms):
            self.kl_ucbs[arms] = compute_kl_ucb(self.p_hats[arms], self.totalcount, self.
    counts[arms], self.c, arms)
```

The results obtained from this method are quite good.

```
================== Task 3 ==================
Testcase 1
Faulty Bandit Algorithm: PASSED. Reward: 8617.80

Testcase 2
Faulty Bandit Algorithm: PASSED. Reward: 1906.12

Testcase 3
Faulty Bandit Algorithm: PASSED. Reward: 3233.14
```

As we see, all the three test cases pass.

4.2 Approach 2

Obtaining the non-faulty true means

```
    def get_reward(self, arm_index, reward):
    self.totalcount += 1
        self.counts[arm_index] += 1
        self.p_hats[arm_index] = ((self.counts[arm_index] - 1)/(self.counts[arm_index]))*
    self.p_hats[arm_index] + (1/(self.counts[arm_index]))*reward
        self.adjusted_p[arm_index] = np.maximum((self.p_hats[arm_index] - 0.5*self.fault)/(1
     - self.fault), 0)
        for arms in range(self.num_arms):
            self.kl_ucbs[arms] = compute_kl_ucb(self.adjusted_p[arms], self.totalcount, self
    .counts[arms], self.c, arms)
```

Here, inside the adjusted empirical mean code, we have prohibited the mean to go below 0. If the mean is less than $0.5 \times f$, we floor the expected mean to 0.

```
    ================== Task 3 ==================
Testcase 1
Faulty Bandit Algorithm: PASSED. Reward: 8652.42

Testcase 2
Faulty Bandit Algorithm: PASSED. Reward: 1911.00

Testcase 3
Faulty Bandit Algorithm: PASSED. Reward: 3241.40
```

Hurray! With this approach, we are getting even better regrets! Thus, this method improves on the Approach 1 since it re-scales the difference between the means.

5 Task 4

In this task, we are supposed to maximise the reward when there are two bandit instances running parallelly. We are taking the following approach. Since we know that the probability of the arms being pulled is each 0.5, we should focus on trying to pull the arm which gives the maximum average of the expected means. Formally, if the bandit instances are $\boldsymbol{P}$ and $\boldsymbol{Q}$ with reward means p_1, p_2 and q_1, q_2, our goal must be to select the arm with maximum value of $(p+q)/2$. The approach which we are using and which is also seeming to work well is to compute the KL-UCBs separately first for each of the arms. Let us say the KL-UCBs for the bandit instances obtained are $kl-ucb(P_i)$ and $kl-ucb(Q_i)$ for the arm i, then we shall obtain the final KL-UCB corresponding to the arm i as

$$kl-ucb(\boldsymbol{i}) = f(\boldsymbol{kl-ucb(P_i)}, \boldsymbol{kl-ucb(Q_i)})$$

In our code, the expression for f which we are using is the following

$$f(x,y) = \frac{xy}{x+y}$$

The reasoning for selecting f in this way is that **f will be lower even if one of the arms has low expected means**. This is necessary as in addition to maximising the mean, we would also like to **minimise the variance**. (The insipiration of this f was the F-1 score used in machine learning, which uses x and y as the precision and recall, and tries to maximise both of them.) Further, the kl-ucb individually are computed using the usual method (binary search). However, the arm's modified KL-UCB is a misnomer, since it is not computed using the regular method.

```
    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

```

```
        #KL-UCB Implementation
        self.total = 0
        self.total_1 = 0
        self.total_2 = 0 #t in the algorithm
        self.p_counts = np.zeros(num_arms)
        self.p_hats = np.zeros(num_arms)
        self.p_kl_ucbs = np.ones(num_arms)
        self.q_counts = np.zeros(num_arms)
        self.q_hats = np.zeros(num_arms)
        self.q_kl_ucbs = np.ones(num_arms)
        self.mean_kl_ucbs = np.ones(num_arms)
        self.counts = np.zeros(num_arms)
        self.hats = np.zeros(num_arms)
        self.kl_ucbs = np.ones(num_arms)
        self.c = 0
```

This is the definition of the class. The give_pull function selects the arm with maximum modified KL-UCB, i.e. $\max f(X, Y)$, where X, Y are the KL-UCBs of the arms from the two bandits.

```
    def give_pull(self):
        # START EDITING HERE

        #KL-UCB Implementation
        return np.argmax(self.mean_kl_ucbs)
```

Finally, we have the get_reward function

```
def get_reward(self, arm_index, set_pulled, reward):
        # START EDITING HERE

        #Algorithm 1 (KL-UCB harmonic)
        if set_pulled == 0:
            self.total_1 += 1
            self.p_counts[arm_index] += 1
            self.p_hats[arm_index] = ((self.p_counts[arm_index] - 1)/(self.p_counts[
    arm_index]))*self.p_hats[arm_index] + (1/(self.p_counts[arm_index]))*reward
            for arms in range(self.num_arms):
                self.p_kl_ucbs[arms] = compute_kl_ucb(self.p_hats[arms], self.total_1, self.
    p_counts[arms], self.c, arms)
        if set_pulled == 1:
            self.total_2 += 1
            self.q_counts[arm_index] += 1
            self.q_hats[arm_index] = ((self.q_counts[arm_index] - 1)/(self.q_counts[
    arm_index]))*self.q_hats[arm_index] + (1/(self.q_counts[arm_index]))*reward
            for arms in range(self.num_arms):
                self.q_kl_ucbs[arms] = compute_kl_ucb(self.q_hats[arms], self.total_2, self.
    q_counts[arms], self.c, arms)
        for arms in range(self.num_arms):
            self.mean_kl_ucbs[arms] = self.p_kl_ucbs[arms]*self.q_kl_ucbs[arms]/(self.
    p_kl_ucbs[arms] + self.q_kl_ucbs[arms])
```

Lines 5-10 and 11-16 compute the individual kl-ucbs of the arms for both bandits. The modified kl-ucbs are being computed in line 18.

```
================== Task 4 ==================
Testcase 1
MultiBandit Algorithm: PASSED. Reward: 6467.96

Testcase 2
MultiBandit Algorithm: PASSED. Reward: 1495.24

Testcase 3
MultiBandit Algorithm: PASSED. Reward: 10976.56
```

As we see, the algorithm is able to perform successfully!