CS747 : Programming Assignment 1

Adityaya Dhande **210070005**

September 10, 2023

Task 1

UCB

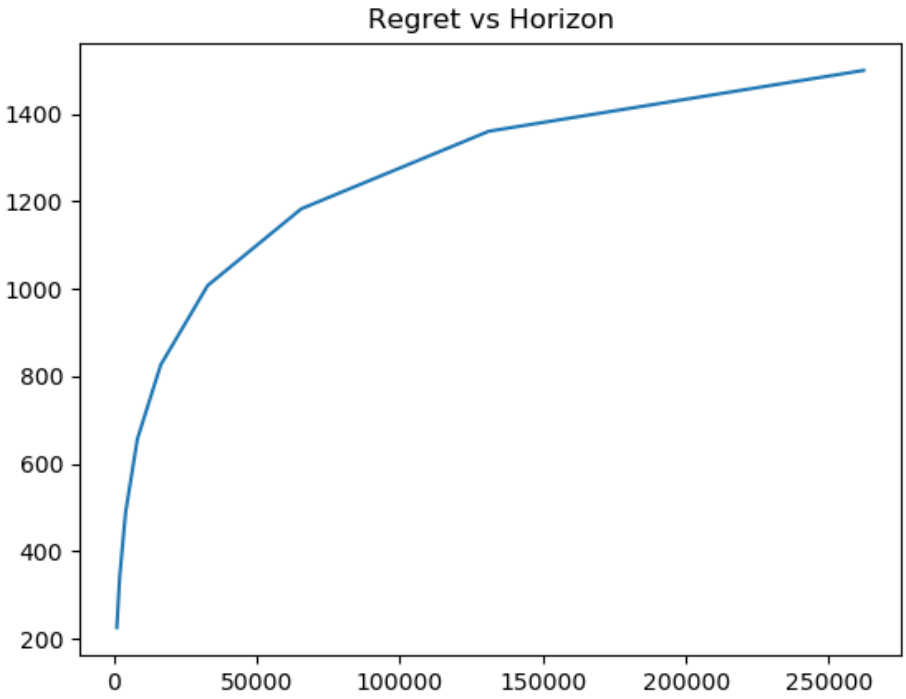

Figure 1: Regret vs Horizon for UCB algorithm

Regret increases with horizon as expected and follows a logarithmic variation

```
    class UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        self.ucbs = np.zeros(num_arms)
        self.u = np.zeros(num_arms)
        self.p_hat = np.zeros(num_arms)

    def give_pull(self):
        t = sum(self.u)
```

```
        self.ucbs = self.p_hat + np.sqrt(2 * np.log(t) / self.u)
        return np.argmax(self.ucbs)

    def get_reward(self, arm_index, reward):
        self.u[arm_index] += 1
        n = self.u[arm_index]
        mean = self.p_hat[arm_index]
        new_mean = ((n - 1) / n) * mean + (reward / n)
        self.p_hat[arm_index] = new_mean
```

Code explanation : `self.ucbs` is an array containing the UCBs of all the arms in a given iteration. `self.u` contains the number of times each arm has been pulled till the current iteration. `self.p_hat` contains the empirical means of all the arms based on pulls till the current iteration.
The `get_reward` function takes the index of the arm which was pulled and the reward and increments `self.u[arm_index]` by 1 because that arm was pulled in this iteration, and thus the total number of pulls of that arm has increased by 1. If now the total number of pulls is n then the old number of pulls was $n-1$, and thus the total reward before the current pull was old mean $\times (n-1)$. The new mean is

$$\text{new mean} = \frac{\text{total reward}}{\text{total number of pulls}} = \frac{\text{total reward before current pull} + \text{reward of current pull}}{\text{total number of pulls}}$$

The new mean is set for the arm that was pulled.
The `give_pull` function calculates the UCB for each arm as $ucb_a^t = \hat{p}_a^t + \sqrt{\frac{2\ln(t)}{u_a^t}}$ and returns the index of the arm with the maximum UCB.

KL-UCB

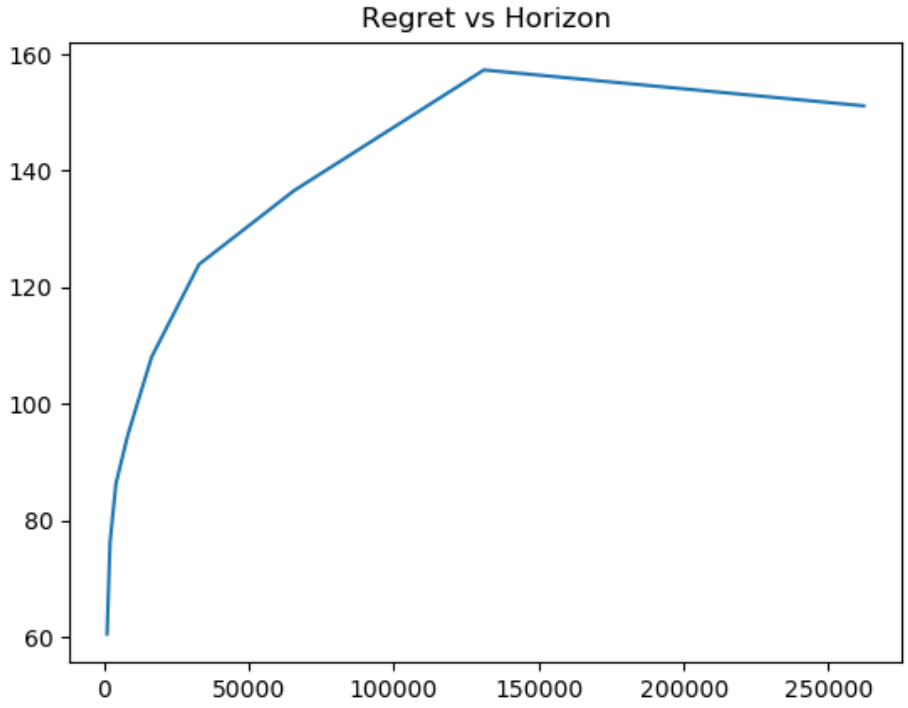

Figure 2: Regret vs Horizon for KL-UCB

Regret increases with horizon as expected and follows a logarithmic variation. The decrease in the end can be explained by negative regret generated because of the optimal arm being pulled many times.

```
    def KL(x ,y):
    if x == 0 :
        return math.log(1/(1 - y))
    elif x == 1 :
        return math.log(1/y)
    return (x * math.log(x / y) + (1 - x) * math.log((1 - x) / (1 - y)))

    def KL_ucb(p, u_a, t, c = 3, tol = 1e-3) :
        l = p
        u = 1
        q = (l + u)/2
        target = (math.log(t) + c * math.log(math.log(t)))/u_a
        while (u - l > tol):
            q = (l + u)/2
            current = KL(p,q)
            if current < target :
                l = q
            elif current > target :
                u = q
            else :
                return q
        return q

    class KL_UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        self.first_pull = True
        self.kl_ucbs = np.zeros(num_arms)
        self.u = np.zeros(num_arms)
        self.p_hat = np.zeros(num_arms)

    def give_pull(self):
        if self.first_pull :
            arm = int(sum(self.u))
            if arm == (len(self.kl_ucbs) - 1) :
                self.first_pull = False
            return arm
        t = sum(self.u)
        for i in range(len(self.kl_ucbs)):
            self.kl_ucbs[i] = KL_ucb(self.p_hat[i], self.u[i], t, c=0)
        arm = np.argmax(self.kl_ucbs)
        return arm

    def get_reward(self, arm_index, reward):
        self.u[arm_index] += 1
        n = self.u[arm_index]
        mean = self.p_hat[arm_index]
        new_mean = ((n - 1) / n) * mean + (reward / n)
        self.p_hat[arm_index] = new_mean
```

Code explanation :`self.first_pull` is a boolean variable which is `True` till all the arms have been sampled for the first time. `self.kl_ucbs` is an array containing the KL-UCBs of all the arms in a given iteration. `self.u` contains the number of times each arm has been pulled till the current iteration. `self.p_hat` contains the empirical means of all the arms based on pulls till the current iteration.
The `get_reward` function takes the index of the arm which was pulled and the reward and increments `self.u[arm_index]` by 1 because that arm was pulled in this iteration, and thus the total number of pulls of that arm has increased by 1. If now the total number of pulls is n then the old number of pulls

was $n-1$, and thus the total reward before the current pull was old mean $\times (n-1)$. The new mean is

$$\text{new mean} = \frac{\text{total reward}}{\text{total number of pulls}} = \frac{\text{total reward before current pull} + \text{reward of current pull}}{\text{total number of pulls}}$$

The new mean is set for the arm that was pulled.
The `give_pull` function samples each arm once for the first `num_arms` times. From then on it calculates the KL-UCB for each arm as q such that

$$\texttt{self.u[arm\_index]} \times KL(\texttt{self.p\_hat[arm\_index]}, q) = \ln(t) + c\ln(\ln(t))$$

and returns the index of the arm with the highest KL-UCB. The value of c used here is 0. The value of q is found using binary search because $KL(p,p) = 0$ and $KL(p,1) = \infty$ and it increases as the second argument increases from p to 1. The function `KL_ucb` does exactly this with a tolerance in the value of q of about 10^{-3}. The function `KL(x,y)` returns the KL divergence of 2 bernoulli distributions with means `x` and `y`.

Thompson Sampling

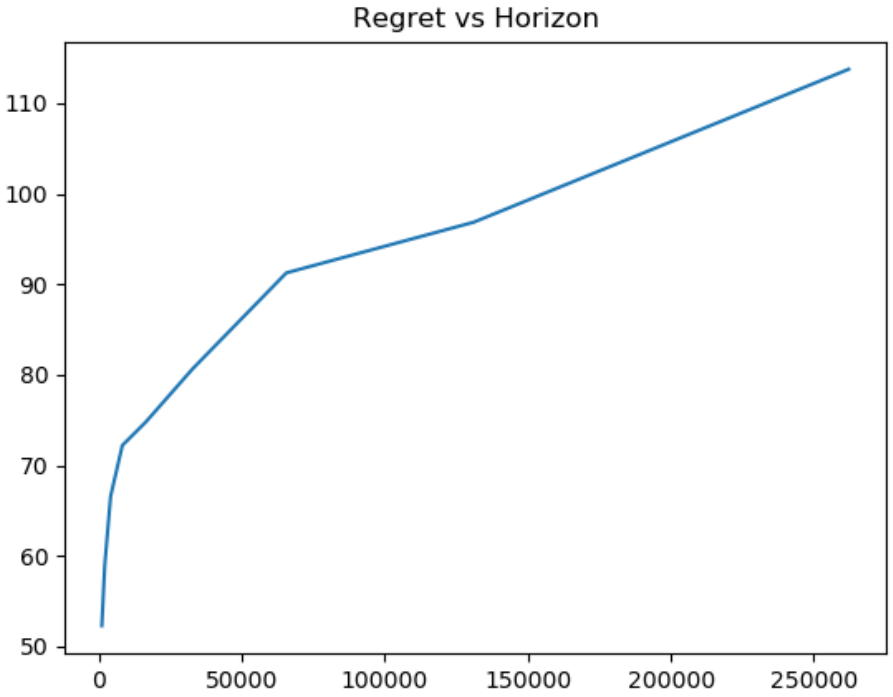

Figure 3: Regret vs Horizon for Thompson Sampling

Regret increases with horizon as expected

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

```
        self.t_samples = np.zeros(num_arms)

    def give_pull(self):
        for i in range(len(self.t_samples)) :
            self.t_samples[i] = np.random.beta(self.sa[i] + 1, self.fa[i] + 1)
        return np.argmax(self.t_samples)

    def get_reward(self, arm_index, reward):
        self.sa[arm_index] += reward
        self.fa[arm_index] += 1 - reward
```

Code explanation : `self.sa` stores the number of successes(reward=1) of all the arms until the current iteration. `self.fa` stores the number of failures(reward=0) of all the arms until the current iteration. `self.t_samples` stores the samples drawn from the beta distribution corresponding to each arm.
The `get_reward` function takes the arm index and the reward and increases the successes of that arm by 1 if the reward is 1 and increases the failures of that arm by 1 if the reward is 0
The `give_pull` function draws, for each arm, samples from

$$\beta(\texttt{self.sa[arm\_index]} + 1, \texttt{self.fa\_[arm\_idex]} + 1)$$

and returns the index of the arm whose β distribution gives the greatest sample.

Task 2

Part A

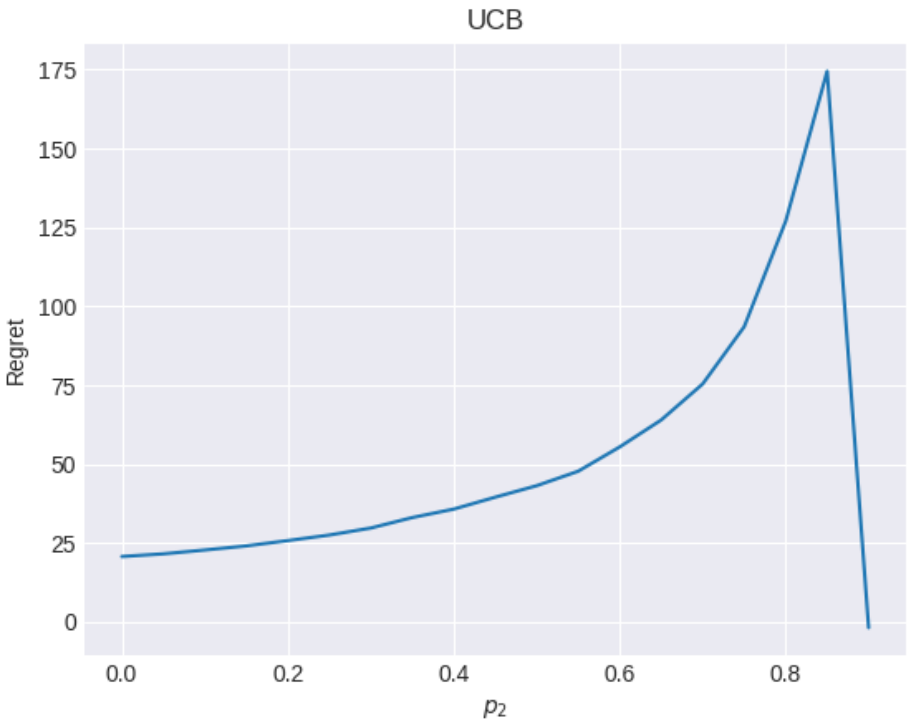

Figure 4: Regret vs p_2 for $p_1 = 0.9$ using UCB algorithm

When p_2 is less than p_1 by a good amount ($\Delta > 0.1$) , the UCB of p_2 does not exceed the UCB of p_1 many times as $\hat{p_2}$ decreases as arm 2 gets pulled more and more and the only term that would cause arm 2 to get pulled is $\sqrt{\frac{2\log(t)}{u_2^t}}$, when

$$\sqrt{\frac{2\log(t)}{u_2^t}} > \sqrt{\frac{2\log(t)}{u_1^t}} + \hat{p_1} - \hat{p_2}$$

This happens less when $\hat{p_1} - \hat{p_2}$ is large because after pulling arm 2 once, arm 1 has to be pulled many times till $\sqrt{\frac{2\log(t)}{u_2^t}}$ exceeds the RHS. In short, when p_2 is small the algorithm needs fewer pulls of arm 2 to decide that it is a bad arm. As the difference decreases the number of pulls needed to decide that arm 2 is a bad arm also increase and thus the regret increases. The increase in the regret due to increased number of pulls exceeds the decrease in regret due to decrease in difference of mean reward of optimal and sub-optimal arms. However, when this difference becomes very small, the regret decreases sharply with the expected cumulative regret being zero for when both arms have the same mean. This explains the nature of the plot of regret vs $p2$ for $p1 = 0.9$ using UCB algorithm

Part B

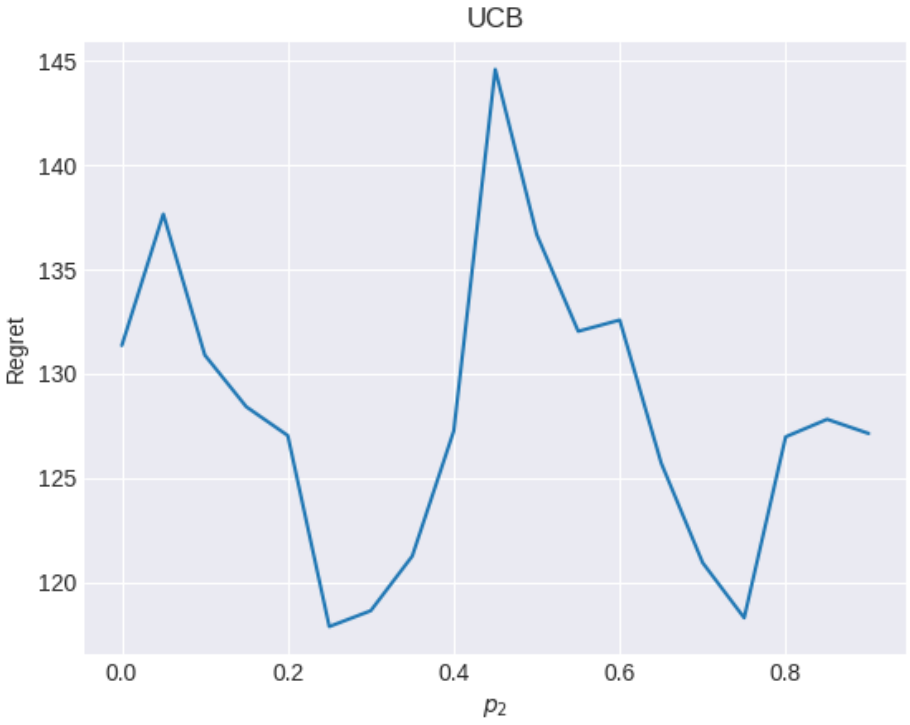

Figure 5: Regret vs p_2 for $\Delta = p_1 - p_2 = 0.1$ using UCB algorithm

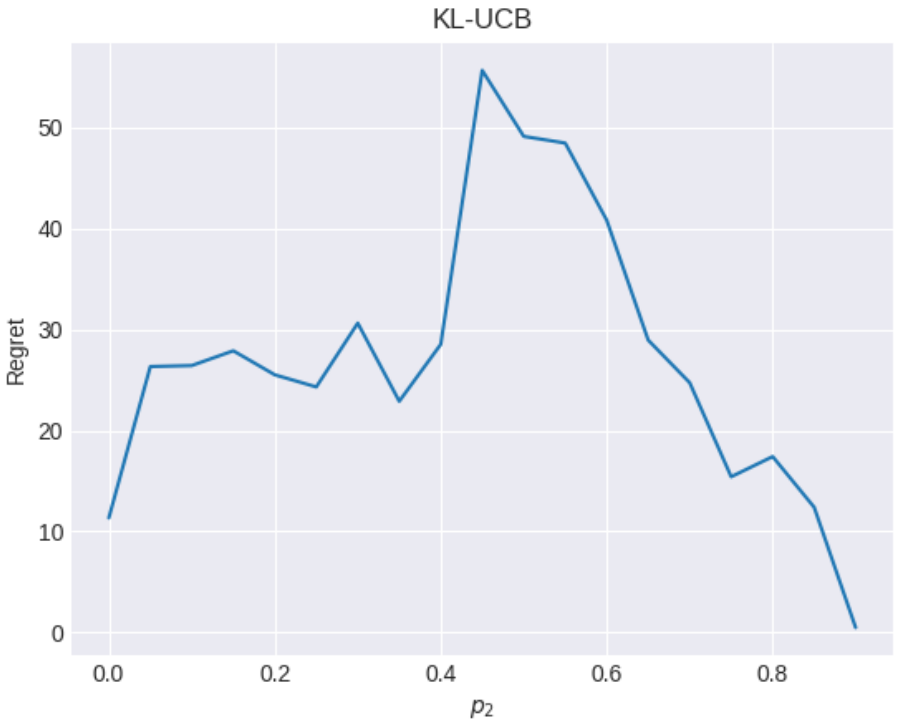

Figure 6: Regret vs p_2 for $\Delta = p_1 - p_2 = 0.1$ using KL-UCB algorithm

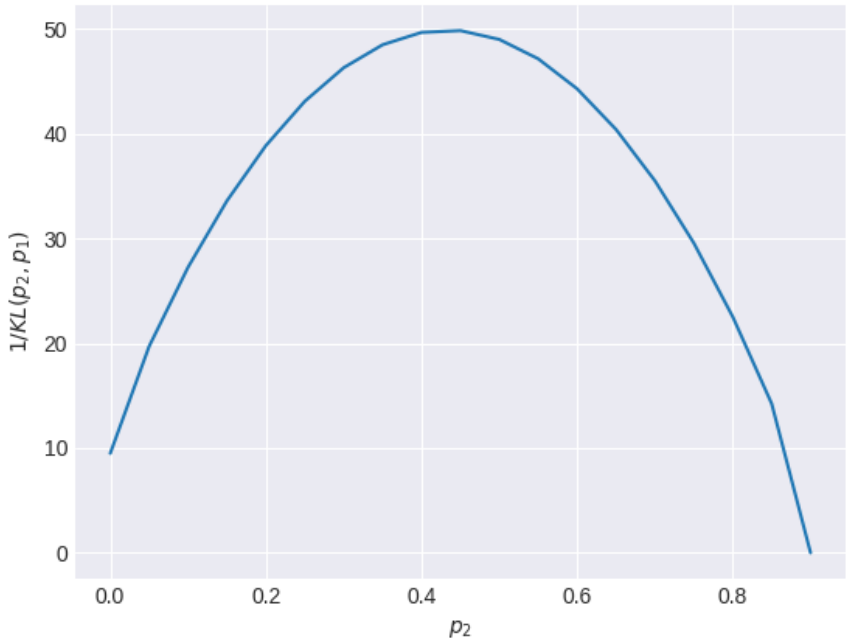

Figure 7: Variation of $1/KL(p_2, p_1)$ for Task 2b

The regret is lower when we use KL-UCB algorithm as compared to UCB algorithm for the same values of p_1 and p_2. As we can see from Figure 7, that is how the lower bound on regret, from Lai and Robbins(1985) varies with p_2 while keeping $p_1 = p_2 + 0.1$. This closely resembles the variation of regret with p_2 as seen in Figure 6. The variation in **regret using KL-UCB**(with p_2) can be explained by the variation of $KL(p_2, p_1)$ (with p_2), which varies inversely with the **regret using KL-UCB**(with p_2). Even Figure 5 roughly follows the same envelope. Thus, the variation of Lai and Robbins' lower bound explains the variation of regret with p_1 and p_2 on using UCB algorithm. UCB however finds the optimal arm slower than KL-UCB which explains the higher magnitude of regret. This is because KL-UCB provides a tighter confidence bound on the true mean of the arm as compared to UCB

Task 3

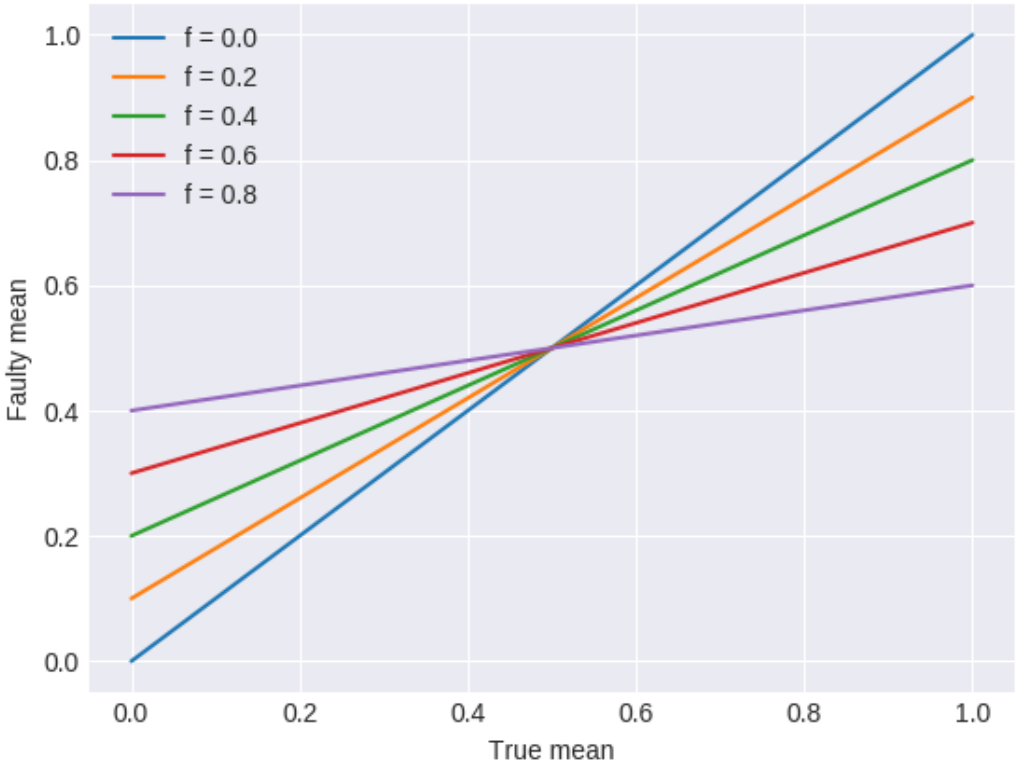

Figure 8: True mean vs faulty mean with fault f

It can be seen from the graph that, a mean which was initially greater, will still be the greater mean as the mapping from true to faulty mean is strictly increasing.

$P = \{p_1, p_2 \dots p_n\}$, where p_i is the mean of arm i. The bandit is faulty with a probability f. When an arm is pulled, the expected reward is,

$$(p_i \times 1 + (1 - p_i) \times 0) \times (1 - f) + f/2 \times 1 + f/2 \times 0 = p_i \times (1 - f) + f/2$$

Let the original successes(without any fault) be $\hat{S}_a^t$ and original failures be $\hat{F}_a^t$, observed successes be S_a^t

and observed failures be F_a^t. Let N_a^t be the total number of pulls of arm a till time t.

$$E[\hat{S}_a^t] = N_a^t \times p_a \text{ and } E[S_a^t] = N_a^t \times (p_a \times (1-f) + f/2)$$

$$\text{From this we get } E[\hat{S}_a^t] = \frac{E[S_a^t] - N_a^t \times f/2}{1-f}$$

$$\text{Similarly } E[\hat{F}_a^t] = \frac{E[F_a^t] - N_a^t \times f/2}{1-f}$$

So we can use an algorithm similar to Thompson Sampling, but instead of drawing samples from $\beta(S_a^t + 1, F_a^t + 1)$ we draw samples from $\beta(\hat{S}_a^t + 1, \hat{F}_a^t + 1)$

Task 4

The problem

Let the two bandit instances be P and Q with means $\{p_1, p_2 \dots p_n\}$ and $\{q_1, q_2 \dots q_n\}$ respectively. Since one of the bandit instances is chosen uniformly at random and sampled, the expected reward on pulling arm i would be $(p_i + q_i)/2$. So we should pull the arm which maximises the average of the means of arms from both the instances as that would give us the maximum expected reward. This is as good as an interaction with a different bandit R with means $\{r_1, r_2 \dots r_n\}$ where $r_i = (p_i + q_i)/2$. Maximising the reward is the same as minimizing the regret, so Thompson Sampling will work for this task as it achieves regret in $O(log(n))$ and has worked better than KL-UCB and UCB algorithms in Task 1.