Programming Assignment 1
Report

Kanak Yadav, 20D070044

September 6, 2023

1 Task 1

1.1 UCB

1.1.1 Implemented Expression

$$ucb_a^t = \hat{p}_a^t + \sqrt{\frac{2 \ln t}{u_a^t}}$$

1.1.2 Variables used

- num_pulls : array of length num_arms; initialized to zeros (i.e. [0, 0, 0, ...]); stores the number of times arm at arm_index has been pulled at the corresponding index.
- emp_prob : array of length num_arms; initialized to zeros; stores the empirical probability of arm at arm_index at the corresponding index.
- ucb : array of length num_arms; initialized to contain all ten's (i.e.[10, 10, 10, ...]); stores the value of the final upper confidence bound which is used for selecting the arm to pull.

1.1.3 Methods and other implementation details

The give_pull simply returns the argmax of the ucb array. For the first num_arms pulls it returns 0, 1, ..., num_arms - 1 in that order, because the actual ucb cannot exceed the initialized value of 10.

The give_reward function updates the values of the variables. First the num_pulls of the pulled arm_index is incremented followed by adjusting the emp_prob based on the reward seen.

The time-step t is calculated by summing all the values in the num_pulls array.

A boolean index array is created to only select the arms with non-zero number of pulls, since the number of times an arm has been pulled comes in the denominator of the ucb expression.

1.1.4 Simulation results

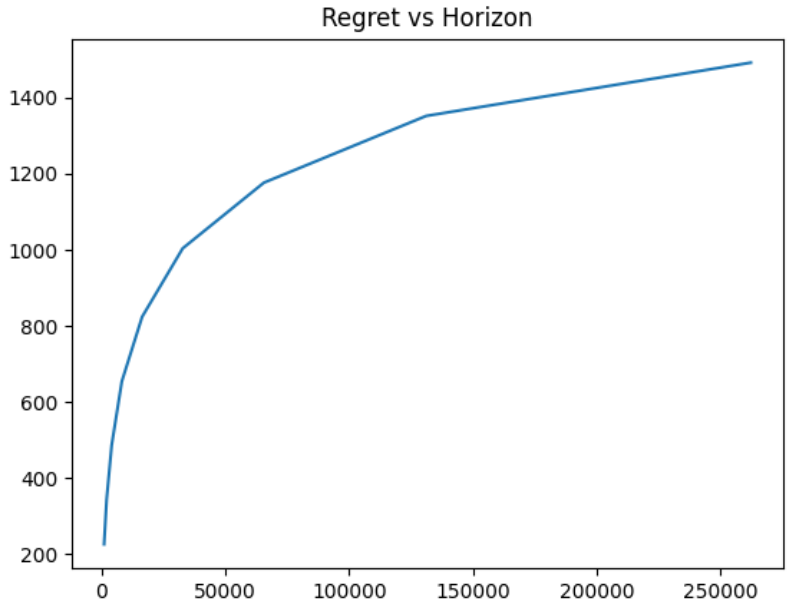

Figure 1: Regret vs horizon for the UCB algorithm. Horizon values are chosen to be the powers of 2 starting at 10 and going till 18, and the expected regret is averaged over 50 simulations for each value of horizon.

1.2 KL-UCB

1.2.1 Implemented Expression

Binary search for a maximal q $\in [\hat{p}_a^t, 1]$ s.t.

$$KL(\hat{p}_a^t, q) \leq \frac{\ln(t)}{u_a^t}$$

where,

$$KL(x, y) = x \ln\left(\frac{x}{y}\right) + (1-x) \ln\left(\frac{1-x}{1-y}\right)$$

1.2.2 Variables used

- num_pulls : array of length num_arms; initialized to zeros (i.e. [0, 0, 0, ...]); stores the number of times arm at arm_index has been pulled at the corresponding index.
- emp_prob : array of length num_arms; initialized to zeros; stores the empirical probability of arm at arm_index at the corresponding index.
- kl_ucb : array of length num_arms; initialized to contain all two's (i.e.[2, 2, 2, ...]); stores the value of the maximum found solution to the above expression, q which is used for selecting the arm to pull.

1.2.3 Helper functions

- kl_div(x, y): returns the KL divergence of the input arguments x and y. The function also handles the corner cases of x being 0 and 1 by taking the corresponding x.log(x) or the (1-x).log(1-x) term to be zero.
- get_kl_ucb(p, k): this function runs a binary search for finding the max q to satisfy the expression shown above. The argument k is used for the comparison against the kl divergence for the current value of q.

1.2.4 Methods and other implementation details

The give_pull simply returns the argmax of the ucb array. For the first num_arms pulls it returns 0, 1, ..., num_arms - 1 in that order, because the value of q cannot exceed the 1 whereas the initialized value of every element of the array is 2.

The give_reward function updates the values of the variables. First the num_pulls of the pulled arm_index is incremented followed by adjusting the emp_prob based on the reward seen.

The time-step t is calculated by summing all the values in the num_pulls array. A variable stores the natural log of the time-step.

While iterating over each arm, the arms with a non-zero num_pulls have their corresponding kl_ucb values updated with the help of the get_kl_ucb helper function. The argument k passed is the right hand side of the expression shown above and hence we check the condition of a non-zero num_pulls.

1.2.5 Simulation results

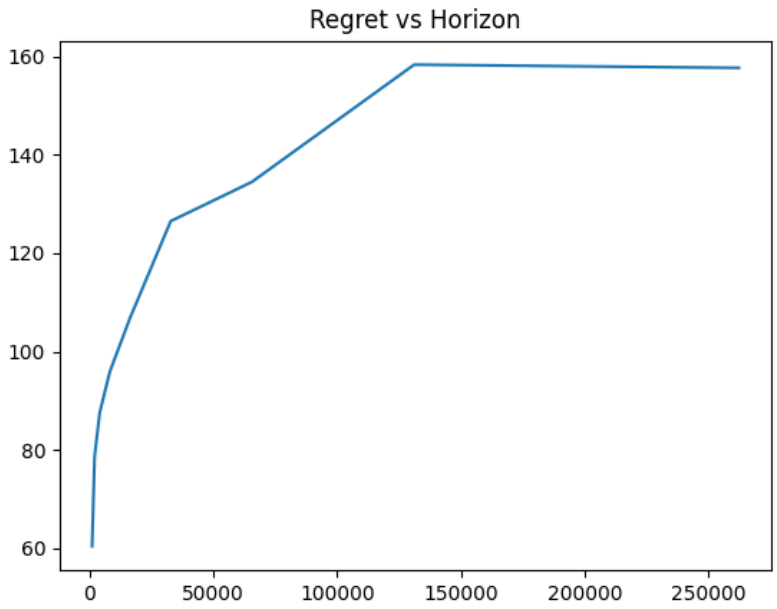

Figure 2: Regret vs horizon for the KL-UCB algorithm. The value of the constant c is chosen to be 0 for the simulations. Horizon values are chosen to be the powers of 2 starting at 10 and going till 18, and the expected regret is averaged over 50 simulations for each value of horizon.

1.3 Thompson sampling

1.3.1 Expression

$$x^t \sim \beta(s^t + 1, f^t + 1)$$

1.3.2 Variables used

- args : array of shape (num_arms, 2); initialized to ones (i.e. [[1, 1], [1, 1], ..., [1, 1]]); stores, for each arm, one plus the number of failures (reward is zero) and one plus the number of successes (reward is one).

1.3.3 Methods and other implementation details

The give_pull simply returns the argmax of the values which are sampled from beta distributions for each arm.

The give_reward function updates the args array as follows:

```
self.args[arm_index, reward] += 1
```

When the reward is zero, the number of failures which are stored at 0 index in the second dimension of the array are incremented and when the reward is one, the number of successes which are stored at index 1 in the second dimension of the array are incremented. Since the reward is 0 for a failure and one for a success, we can use the reward (which is passed as an argument to the function) as the index to be used while incrementing the corresponding value in the args array, and this is done in a single line.

1.3.4 Simulation results

Figure 3: Regret vs horizon for the Thompson Sampling algorithm. Horizon values are chosen to be the powers of 2 starting at 10 and going till 18, and the expected regret is averaged over 50 simulations for each value of horizon.

1.4 Comparisons

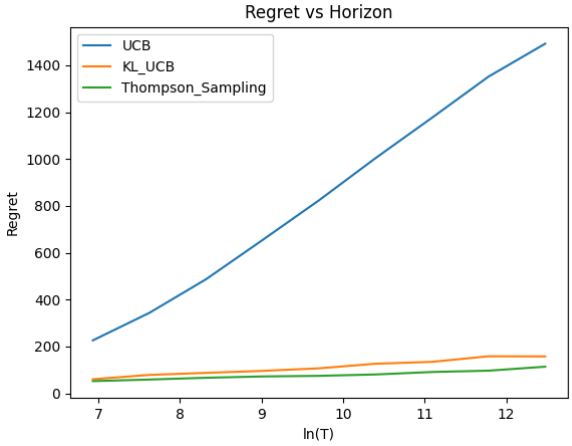

Figure 4: Comparison between the regret achieved by different algorithms versus natural log of the horizon. We observe that KL-UCB and Thompson Sampling are way better than the UCB algorithm which achieves O(log(T)) regret.

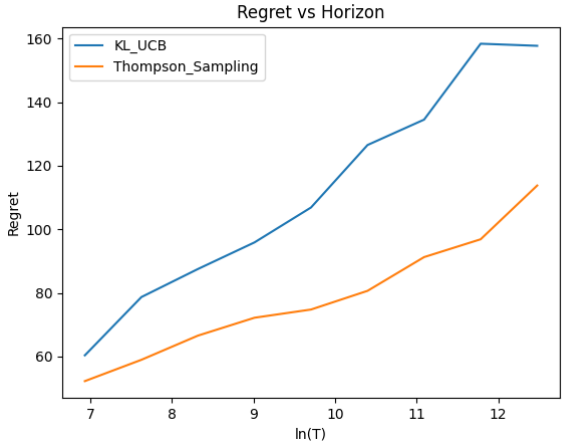

Figure 5: Comparison between the regret achieved by different algorithms versus natural log of the horizon. We observe that Thompson Sampling which exactly matches the Lai and Robbins' lower bound on regret is better than KL-UCB which asymptotically match the Lai and Robbins' lower bound on regret.

2 Task 2

2.1 Part A

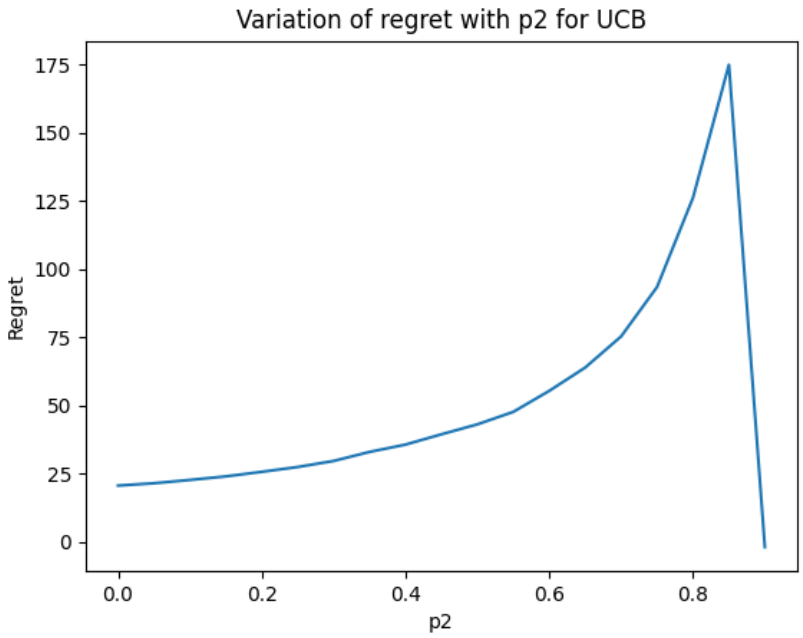

Figure 6: As p2 increases from 0 to 0.9 the difference between the means decreases from 0.9 to 0.

Observations: Initially the regret is low, but it starts to increase in an exponential way in the middle and after a point, suddenly drops to even below the initial value.

Explanation: When the difference between the two means is high (or p2 is low), the algorithm can quickly differentiate between the optimal arm and the non-optimal arm and the regret is low.

As p2 increases, the regret also starts increasing slowly since the certainty with which the algorithm identifies the optimal arm decreases. Further increase in p2 makes it that much more harder to differentiate and the rate of increase of regret increases.

And when the two means are equal, regret asymptotically reaches zero (or the expected cumulative regret becomes zero).

2.2 Part B

2.2.1 UCB

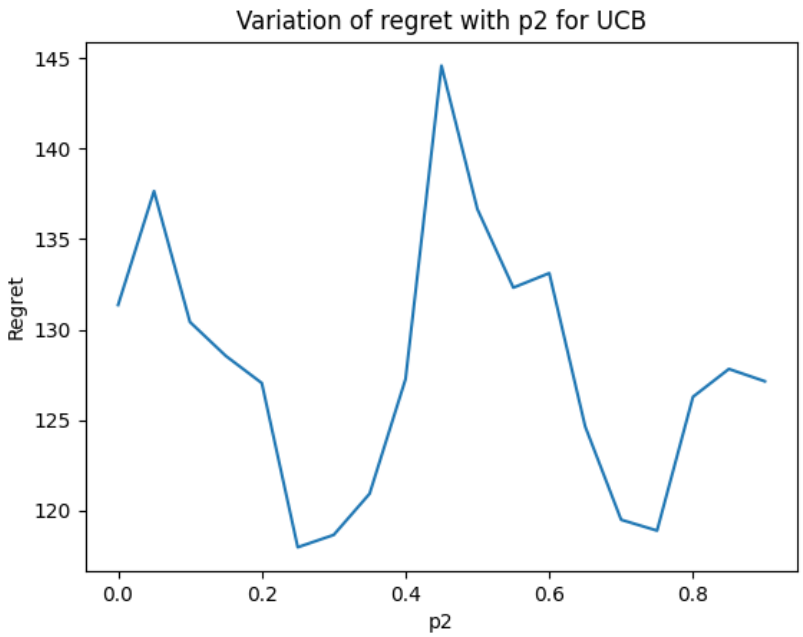

Figure 7: The difference between the means is constant (= 0.1) and so when p2 goes from 0 to 0.9, p1 goes from 0.1 to 1.

Observations: The regret curve for the UCB algorithms experiences two peaks and two valleys in an oscillatory manner.

Explanation: The peak in the middle of the plot can be accredited to the fact that at that point, p1 and p2 can be considered being "close" to a fair coin which gives reward 1 and 0 uniformly at random. This leads to maximum entropy and thus, both the reward and the empirical probabilities of the arms become highly unpredictable giving rise to a high value of regret.

The two valleys on either side of the middle peak are just contrasting to it. The regret increases outside of this region but does not cross the middle peak. It increases because pulling either of the arms would result in almost the same outcomes (0 or 1 reward but with high probability). The regret increases much more to the left of the middle peak because most of the rewards are zero and so the regret keeps on steadily building up. Whereas the regret on the right hand side will not increase by much because most of the rewards would be one.

2.2.2 KL-UCB

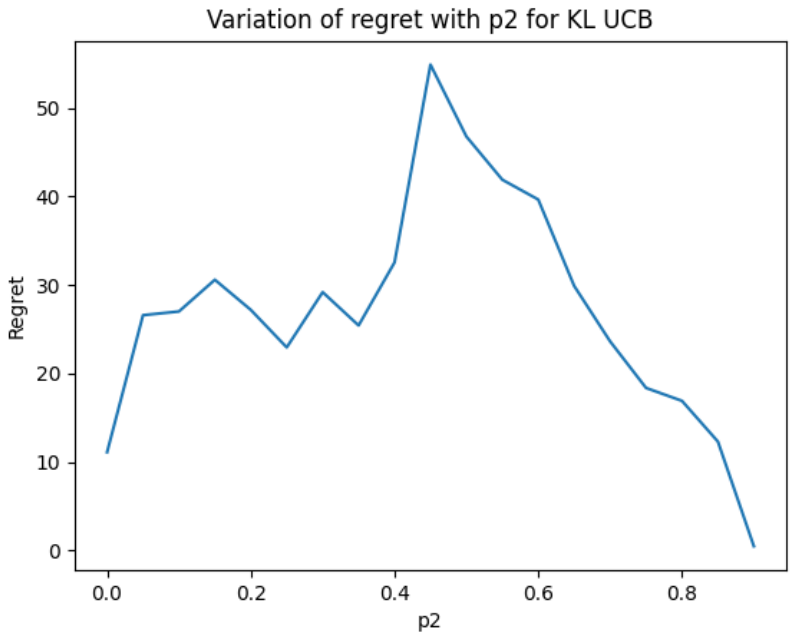

Figure 8: The difference between the means is constant (= 0.1) and so when p2 goes from 0 to 0.9, p1 goes from 0.1 to 1.

Observations: The regret curve starts low and shows some minor oscillations when the means are around 0.1 to 0.4. Then it achieves a peak and goes on to have a steady decline as the means increase.

Explanation: The middle peak is arising due the same reasoning as in the UCB case above.

The decrease in regret as the means increase is due the rewards being one with high probabilities and so regret is lower here when compared to the region where the means are less, since there the probability of getting a reward of one from the optimal arm is also low.

3 Task 3

3.1 Algorithm used

The base algorithm used for this task is Thompson sampling since it is the easiest to implement.

The implementation is the same as in Task 1 in section 1.3 except for the get_reward method where the fault is handled.

3.2 Fault handling

We know that with probability f, the reward we receive will not be from any of the arms of the bandit but rather a uniform distribution. Thus, if we update the args the same way as in Task 1 as in section 1.3 then it would result in reducing the mean of our belief distribution on an arm with a potentially higher mean and similarly increasing it for an arm with a potentially lower mean.

To tackle this, we flip a coin that gives us heads with probability equal to one minus the probability of a fault, and if we get a heads, we update our belief distribution (the args array) and if we get a tails, we consider the reward to have come from none of the arms and thus we ignore it.

```
if np.random.binomial(1, 1 - self.fault):
    self.args[arm_index, reward] += 1
```

3.3 Simulation results

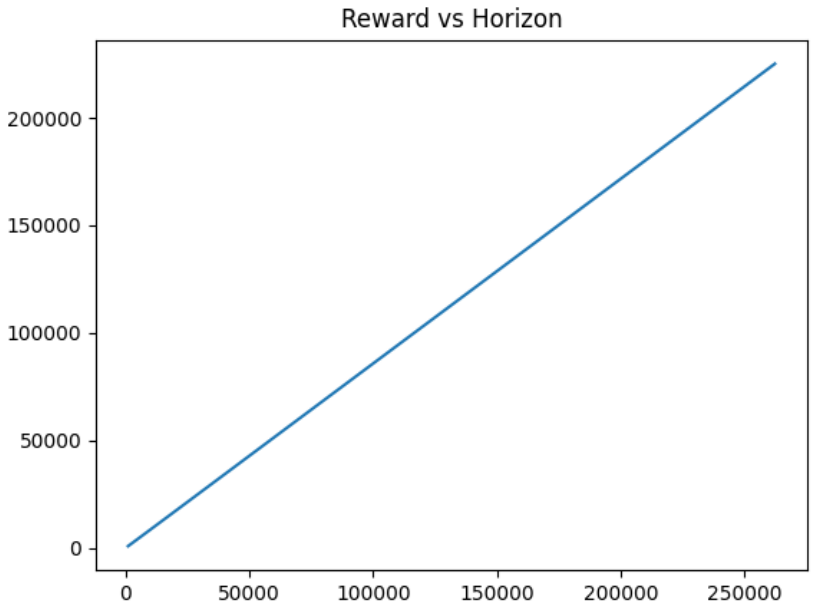

Figure 9: The reward attained by the proposed algorithm on the given bandit for the faulty-bandit task. The horizon is varied from $2\hat{1}0$ to $2\hat{1}8$, and the reward is averaged over 50 simulations.

4 Task 4

4.1 Algorithm used

Here also we use the Thomson sampling algorithm as the base. However the args array, now has a shape of (num_arms, 2, 2), i.e., the outermost dimension is for an arm_index, the middle dimension is for the number of bandits, and the innermost dimension has the number of failures and successes indexed in that order.

4.2 Multiple bandits handling

Fortunately, we are provided with the index of the arm pulled as set_pulled and we can use this to index the middle dimension directly and thus the give_reward function simply becomes:

```
self.args[arm_index, set_pulled, reward] += 1
```

When deciding on which arm to pull, we sample all the arms of both the bandits, and since the bandit form which the pull will happen is selected uniformly at random, we pick the arm that gives us the maximum sum of the samples from the beta distributions of both the bandits.

In other words, we decide the expected reward as the average of the expected rewards from each bandit. We have eliminated the constant factor of half and just considered the sum of the expected reward to be the measure of the total expected reward.

```
return np.argmax(
    np.random.beta(self.args[:, 0, 1], self.args[:, 0, 0])
    + np.random.beta(self.args[:, 1, 1], self.args[:, 1, 0])
    )
```

4.3 Simulation results

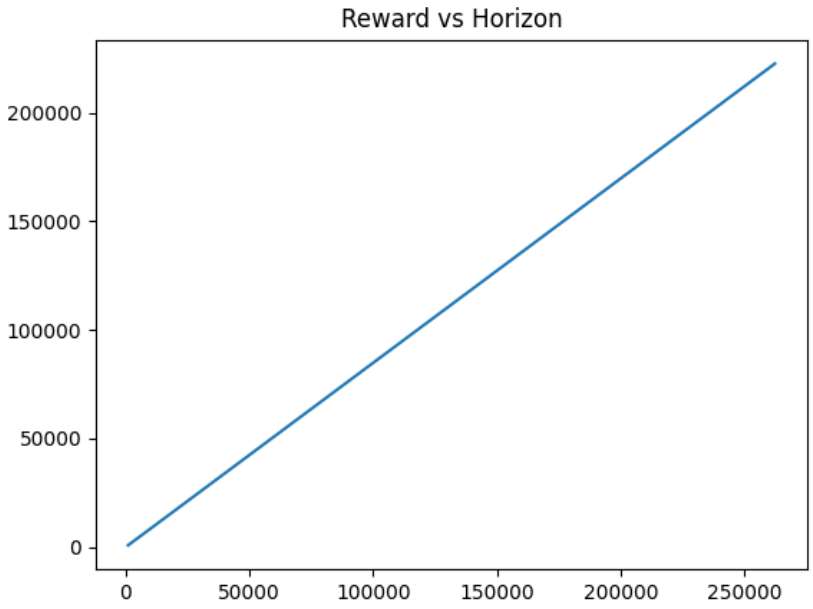

Figure 10: The reward attained by the proposed algorithm on the two given bandits for the multi-bandit task. The horizon is varied from 2$\hat{1}$0 to 2$\hat{1}$8, and the reward is averaged over 50 simulations.