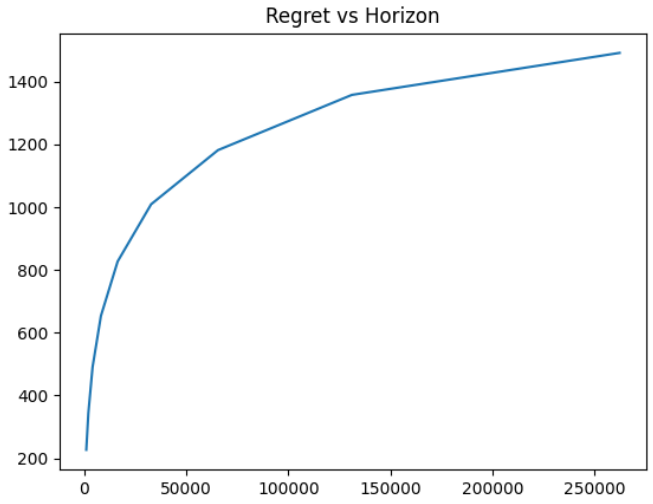
Dhruv Jain (22M0828) - Report Assignment 1

Task 1)

A. UCB Algorithm:



Regret grows with a sub-logarithmic rate in the horizon.

Fig. Plot of Regret vs Horizon for UCB algorithm.

Code:

```
class UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        # START EDITING HERE
        self.ucb = np.zeros(num_arms)
        self.emp_mean = np.zeros(num_arms)
        self.pulls = np.zeros(num_arms)
        self.ctr = 0
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        self.ctr += 1
        if self.ctr <= self.num_arms: return self.ctr-1
        return np.argmax(self.ucb)
        # END EDITING HERE


    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        self.pulls[arm_index] += 1
        total_pulls = sum(self.pulls)
        n = self.pulls[arm_index]
        self.emp_mean[arm_index] = ((n-1)*self.emp_mean[arm_index] + reward)/n
        ucb_factor = [math.sqrt(2*math.log(total_pulls)/puls) if puls else 0 for puls in self.pulls]
        self.ucb = self.emp_mean + ucb_factor
        # END EDITING HERE
```

Variables:

- self.ucb: to store ucb values.
- self.emp_mean: to store empirical means of arms.
- self.pulls: to store the number of pulls of each arm.
- self.ctr: to store total pulls.
- total_pulls: to store total pulls.
- n: no. of time the particular arm is pulled.
- ucb_factor: to calculate the $\sqrt{\frac{2\ln(t)}{u_a^t}}$ terms for all the arms.

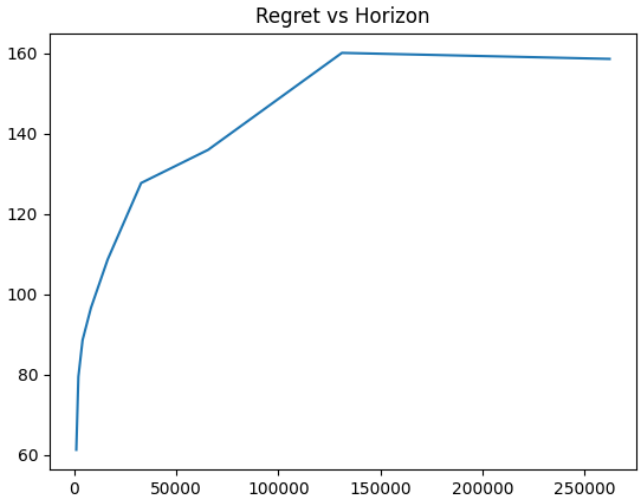
Logic:
Firstly: All arms will be pulled at-least once to to get to have non-zero pulls for each arm.
Secondly: Upon receiving a reward, update empirical means and ucb_factor to get new ucb values and whenever a new pull is requested, an arm with max ucb value is chosen.

Code Explanation:
Line 84-87: UCB initialises several arrays to keep track of the upper confidence bounds (ucb), empirical mean rewards (emp_mean), and the number of pulls (pulls) for each arm. The ctr variable is used to count the number of pulls. (__init__)

Line 92-94: If the counter ctr is less than or equal to the number of arms, it explores each arm once initially (this is the exploration phase). After the exploration phase, it exploits the arm with the highest UCB value, selecting the arm with the highest upper confidence bound. (give_pull)

Line 100-105: We update the number of pulls for the chosen arm, calculate the new empirical mean reward, and update the UCB values for all arms based on the new information. (get_reward)

B. KL-UCB Algorithm:



The graph's y-values show that KL-UCB performs better than UCB algorithm.

Fig. Plot of Regret vs Horizon for KL-UCB algorithm.

Code Explanation:
Line 104-107: KL-UCB initialises arrays to store KL-UCB values (klucb), empirical mean rewards (emp_mean), and the number of pulls (pulls) for each arm. The ctr variable counts the number of pulls. (__init__)

Line 122-124: During the exploration phase (when ctr is less than or equal to the number of arms), it explores each arm once. After the exploration phase, it exploits the arm with the highest KL-UCB value. (give_pull)

Line 129-137: We update the number of pulls and empirical mean reward for the chosen arm. We calculate the right-hand side (rhs) of the KL-UCB inequality. We then update the KL-UCB values for all arms using a binary search function based on the empirical mean and rhs. (get_reward)

Code:

```
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.klucb = np.zeros(num_arms)
        self.emp_mean = np.zeros(num_arms)
        self.pulls = np.zeros(num_arms)
        self.ctr = 0
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        self.ctr += 1
        if self.ctr <= self.num_arms: return self.ctr-1
        return np.argmax(self.klucb)
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        self.pulls[arm_index] += 1
        n = self.pulls[arm_index]
        self.emp_mean[arm_index] = ((n-1)*self.emp_mean[arm_index] + reward)/n

        rhs = math.log(sum(self.pulls))
        for i in range(self.num_arms):
            u_t_i = self.pulls[i]
            p_t_i = self.emp_mean[i]
            if u_t_i: self.klucb[i] = search(p_t_i, 1, p_t_i, u_t_i, rhs) if p_t_i!=1 else 1
```

Variables:

- self.klucb: to store kl-ucb values.
- self.emp_mean: to store empirical means of arms.
- self.pulls: to store the number of pulls of each arm.
- self.ctr: to store total pulls.
- n: no. of time the particular arm is pulled.
- rhs: to store the following $\leq \ln(t) + c\ln(\ln(t))$ quantity with c=0 as per the instructions.
- u_t_i: total pulls for arm index i.
- p_t_i: empirical mean of arm index i.

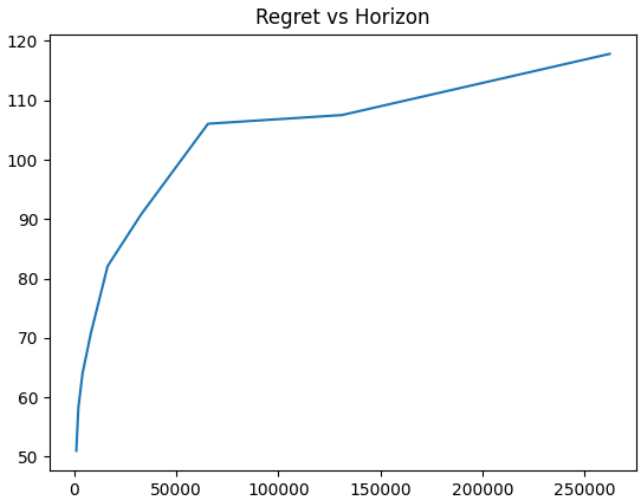
Logic:
Firstly: All arms will be pulled at-least once to to get to have non-zero pulls for each arm.
Secondly: Upon receiving a reward, update empirical means and kl-ucb values for every arm and whenever a new pull is requested, an arm with max kl-ucb value is chosen.

$$\text{ucb-kl}_a^t = \max\{q \in [\hat{p}_a^t, 1] \text{ s. t. } u_a^t KL(\hat{p}_a^t, q) \leq \ln(t) + c\ln(\ln(t))\}$$

To solve the above equation, we choose to use binary search for q_t_i. Note: c=0 is taken.
"**kld**" & "**search**" are 2 user-defined helper functions in code for finding kl-ucb values for all arms.

```
def search(lower, upper, p_t_i, u_t_i, rhs, thres = 1e-2):
    diff, summ = upper-lower, upper+lower
    q_t_i = summ/2
    lhs = u_t_i*kld(p_t_i,q_t_i)

    if lhs <= rhs:
        return q_t_i if diff<thres else search(q_t_i, upper, p_t_i, u_t_i, rhs)
    else:
        return lower if diff<thres else search(lower, q_t_i, p_t_i, u_t_i, rhs)
```

C. Thompson Sampling:



The graph's y-values show that Thompson sampling algorithm performs better than KL-UCB algorithm.

Fig. Plot of Regret vs Horizon for Thompson Sampling algorithm.

Variables:

- self.thomps: to store the values sampled from the beta distribution for all arms.
- self.success: to store no. of success of arms.
- self.pulls: to store the number of pulls of each arm.
- self.failure: it's actually a function which returns the no. of failures of a specified arm.

Logic:

Upon receiving a reward from a pull, we update the success counter of that arm. After we get a sample from every beta-distribution of success & failures of a particular arm and store those sampled values in the thomps variable. Whenever a new pull is requested, an arm with max thomps value is chosen.

Code:

```
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.thomps = np.zeros(num_arms)
        self.success = np.zeros(num_arms)
        self.pulls = np.zeros(num_arms)
        self.failure = lambda x: self.pulls[x] - self.success[x]
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        return np.argmax(self.thomps)
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        if reward == 1: self.success[arm_index] += 1
        self.pulls[arm_index] += 1
        self.thomps = [np.random.beta(self.success[i]+1, self.failure(i)+1) for i in range(self.num_arms)]
```

Code Explanation:
Line 145-148: Thompson Sampling initialises arrays to store Thompson Sampling values (thomps), the number of successes (success), the number of pulls (pulls), and calculates the number of failures for each arm. (__init__)

Line 153: We select an arm for pulling based on the arm with the highest sampled value from all the thomps values. (give_pull)

Line 158-160: The method updates the algorithm's internal state based on the chosen arm and the observed reward. We increment the number of successes for the chosen arm if the reward is 1. We also update the number of pulls for the chosen arm. Finally, we update the `thomps` sampled values for all arms by sampling from beta distributions using the number of successes and failures for each arm. (get_reward)

Final comparison of Algorithms:

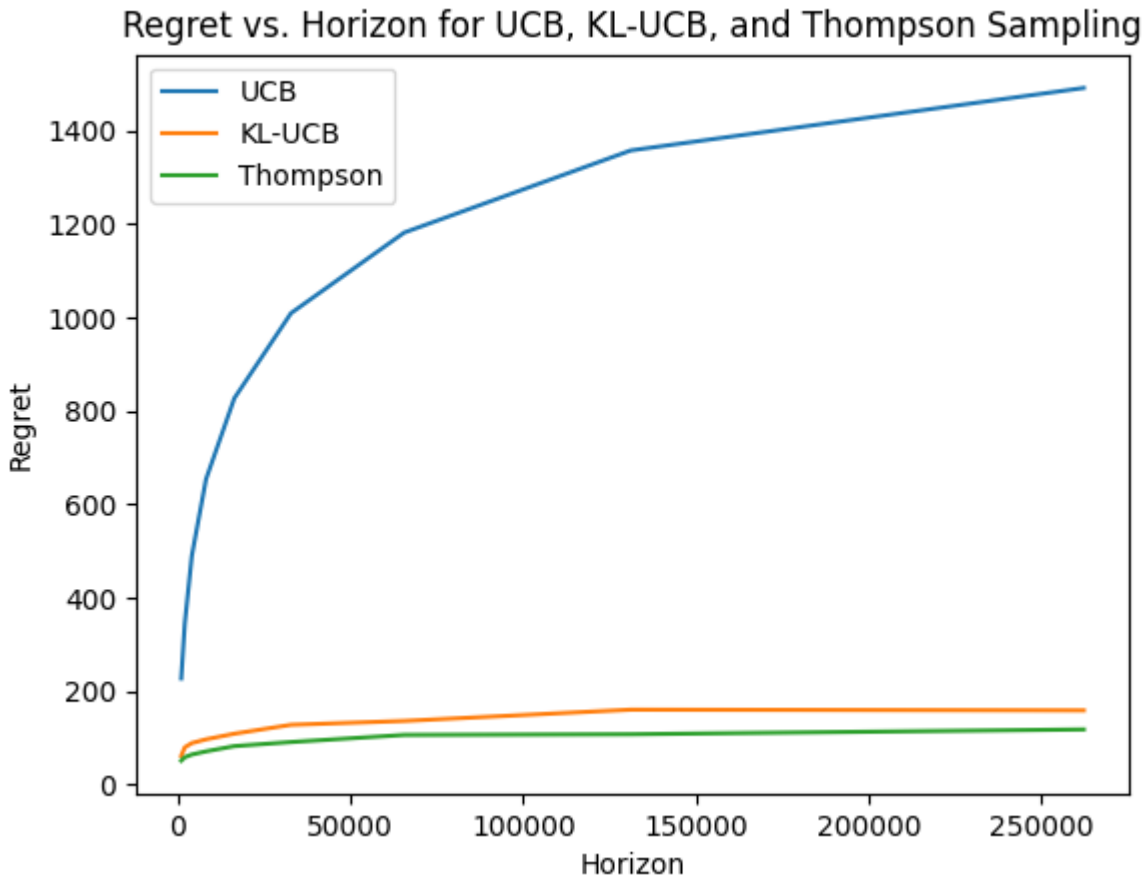


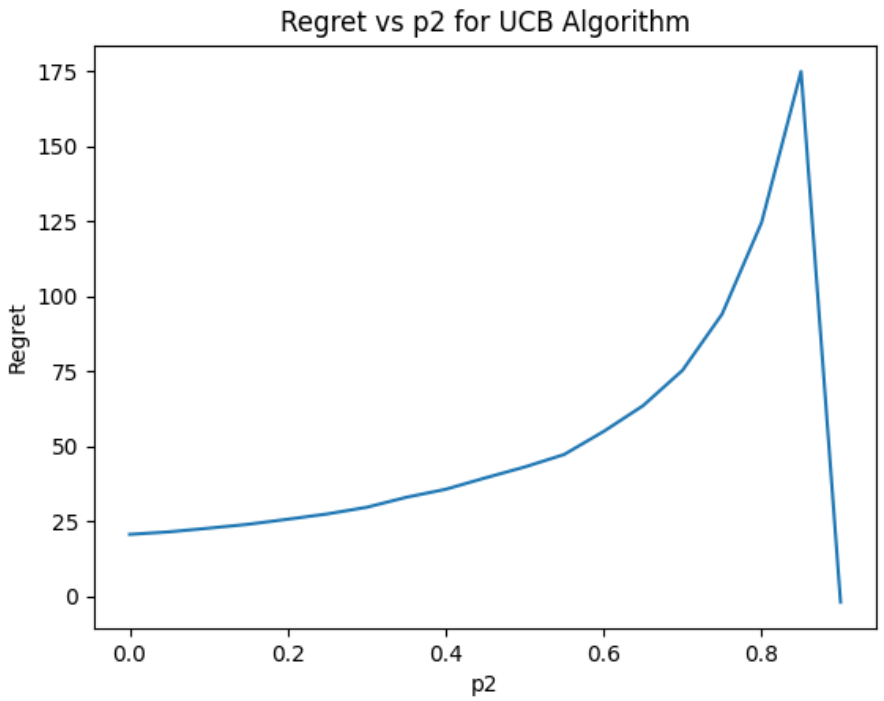
Fig. Plot of Regret vs Horizon for all algorithms.

Thompson Sampling algorithms's performance was better than other algorithms.
All these above algorithms are designed to balance exploration and exploitation to maximise cumulative rewards over a horizon. The specific exploration strategy differs between UCB, KL-UCB, and Thompson Sampling, but all aim to make decisions about which arms to pull based on past observations.

Task 2)

A. **Part A:**

Plot:



Observations:

- It seems that the regret values for p2 values are increasing as p2 increases.
- As p2 is approaching p1, it becomes more challenging for the algorithm to differentiate between the two arms, thereby leading to higher regret.
- There is an observation with the last regret value in when p2=p1=0.9, the regret is almost zero.

Explanations:

- Regret Increase with p2: The regret increases as the mean of the second arm p2 increases from 0 to 0.9. This observation aligns with the intuition that when the difference between the means of the arms is larger, the algorithm is more likely to select the suboptimal arm and thus accumulate more regret.
- Regret Saturation: In the last value, where p2 is 0.9, the regret is relatively small compared to the earlier values. This is because when both arms have similar high means, the algorithm quickly identifies any arm as the optimal arm and accumulates less regret.

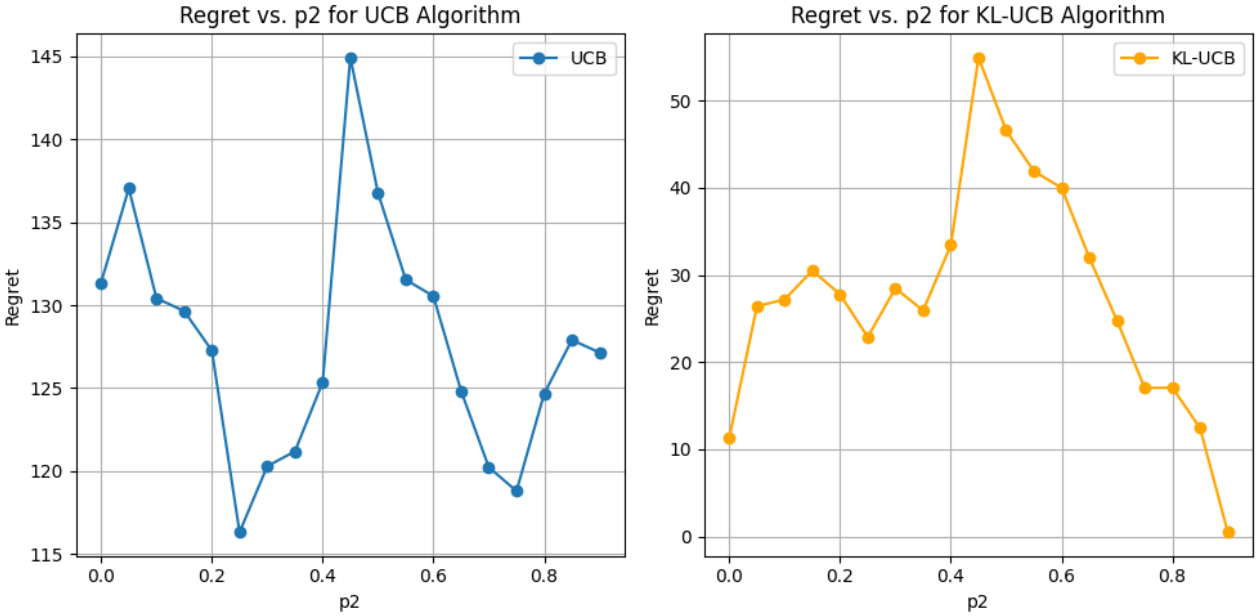
Code:

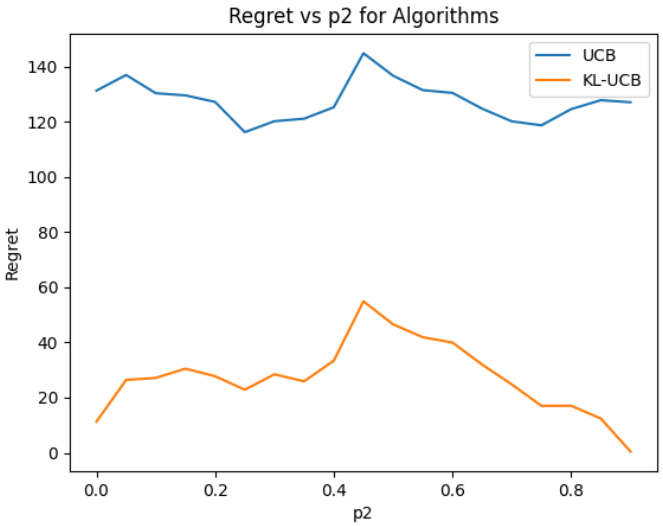
```
# Task A
task2p2s = list(np.arange(0, 0.91, 0.05))
task2p1s = [0.9]*len(p2s)
horizon = 30000
regrets = task2(UCB, horizon, task2p1s, task2p2s, 50)
print(task2p2s, regrets)

plt.plot(task2p2s, regrets)
plt.xlabel('p2')
plt.ylabel('Regret')
plt.title('Regret vs p2 for UCB Algorithm')
plt.savefig("task2a.png")
```

Task 2)

B. Part B:

Plot:





Code:

```
# Task B
task2p2s = np.arange(0, 0.91, 0.05)
task2p1s = task2p2s + 0.1
horizon = 30000
regrets_ucb = task2(UCB, horizon, task2p1s, task2p2s)
regrets_kl_ucb = task2(KL_UCB, horizon, task2p1s, task2p2s)

plt.plot(task2p2s, regrets_ucb, label='UCB')
plt.plot(task2p2s, regrets_kl_ucb, label='KL-UCB')
plt.xlabel('p2')
plt.ylabel('Regret')
plt.title('Regret vs p2 for Algorithms')
plt.legend()
plt.savefig("task2b.png")
```

Observations:

- As we vary the value of p2 from 0 to 0.9 in steps of 0.05, we observe significant differences in the regret values between UCB and KL-UCB algorithms.
- The regret values for UCB tend to decrease as p2 increases except for the spike in middle (where algorithm get's confused and incur higher regrets), but they remain notably higher than those of KL-UCB regret values across the entire range.
- In contrast, KL-UCB exhibits an increase in regret values as p2 increases around p2=0.5, and then regret values decrease substantially as p2 increases. The regret values approach nearly zero for last p2 values (p2=0.9).

Explanations:

- Across the entire range of p2 values, UCB consistently maintains higher regret compared to KL-UCB. This is due to UCB's more exploratory nature, which leads to longer exploration times before effectively exploiting the better-performing arm. Proves that KL-UCB is a better algorithm.
- KL-UCB has an increasing regret till p2=0.5 due to the uncertainty in choosing between arms with similar success probabilities. However, as p2 increases further, KL-UCB efficiently identifies the optimal arm with a higher success probability and rapidly reduces regret to nearly zero when p2=0.9.
- There's a noticeable spike in algorithms regret around p2=0.5. This occurs because algorithms are uncertain about which arm to choose when the difference in success probabilities between the two arms is small and chances of success are almost uniform in nature. It alternates between the two arms, resulting in higher regret.

Summary:

In the comparison of UCB and KL-UCB algorithms, KL-UCB consistently outperforms UCB in terms of regret. KL-UCB adapts better to changing mean values and demonstrates lower regret, especially when the means are close. UCB struggles more in such scenarios, leading to faster regret accumulation. KL-UCB's exploration strategy makes it a strong choice for multi-armed bandit problems with fixed mean differences.

Plots:

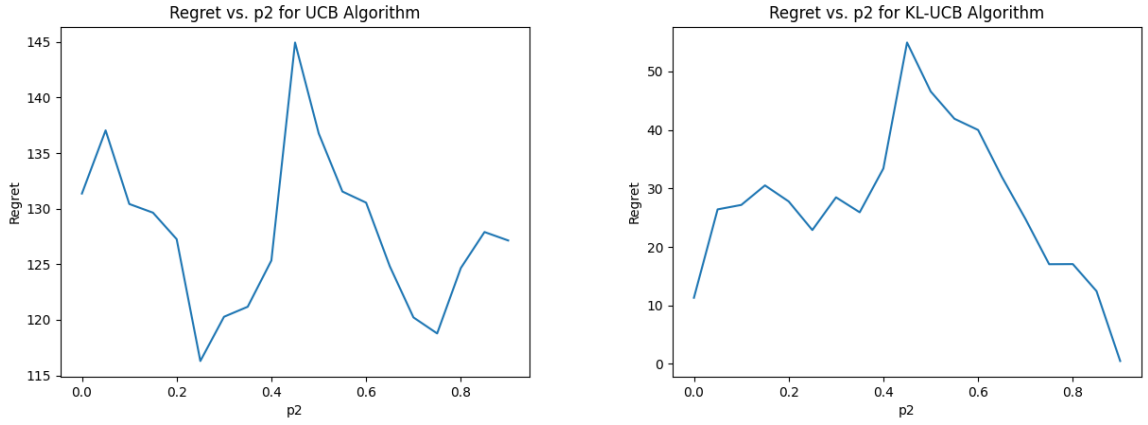


Fig. Plot of Regret vs p2 for UCB and KL-UCB algorithms.

Task 3)
Description of my approach to tackle the problems:

- Thompson Sampling Algorithm:
    - In Thompson Sampling, the goal is to maximise the cumulative reward by selecting the arms that are likely to have the highest true success probability.
    - The algorithm maintains a posterior distribution over the true success probabilities of each arm. This distribution is modelled as a Beta distribution, with parameters that are updated based on the observed outcomes (successes and failures) of each arm.

In the context of the faulty bandit problem, each arm has a known probability of giving a correct output (true success probability) and a known probability of giving a faulty output.

- Analogy to a Simpler Problem:
    - In the simpler problem in Task 1 with no faulty bandits, the success probability of an arm i was denoted as `p_i` (assume). The algorithm used empirical means to estimate these `p_i` values using beta distributions and selected the arm with the highest empirical mean.
    - Now, In the faulty bandit problem, the true success probability of each arm is a combination of the probability of not failing and being successful i.e (1 - P(fault))*p_i and the probability of failing but still getting a reward=1 i.e 0.5*P(fault). This new success probability is denoted as `new_p_i` for an arm indexed at i.
    - new_p_i = (1 - P(fault))*p_i + (0.5*P(fault)) -> Equation. 1.

The key insight here is that if we replace the faulty bandits with normal bandits (i.e no faults) and replace the `p_i` values with `new_p_i` values, both situations become exactly identical. This is because the `new_p_i` values capture the effect of both the arm's true success probability and the fault probability combined as one.

- Solution:
    - Thompson Sampling, when applied to the faulty bandit problem, aims to estimate the `new_p_i` values that captures the effect of both the arm's true success probability and the fault probability combined as one.
    - The algorithm estimates beta distributions of `new_p_i` values, where distribution parameters are updated based on the observed rewards considering the fault probability.
    - By estimating the beta distributions of the `new_p_i` values for all the arms, Thompson Sampling effectively adapts to the presence of faults and selects arms that are likely to have the highest combined probability of success and fault tolerance i.e the one with the max `new_p_i` values.

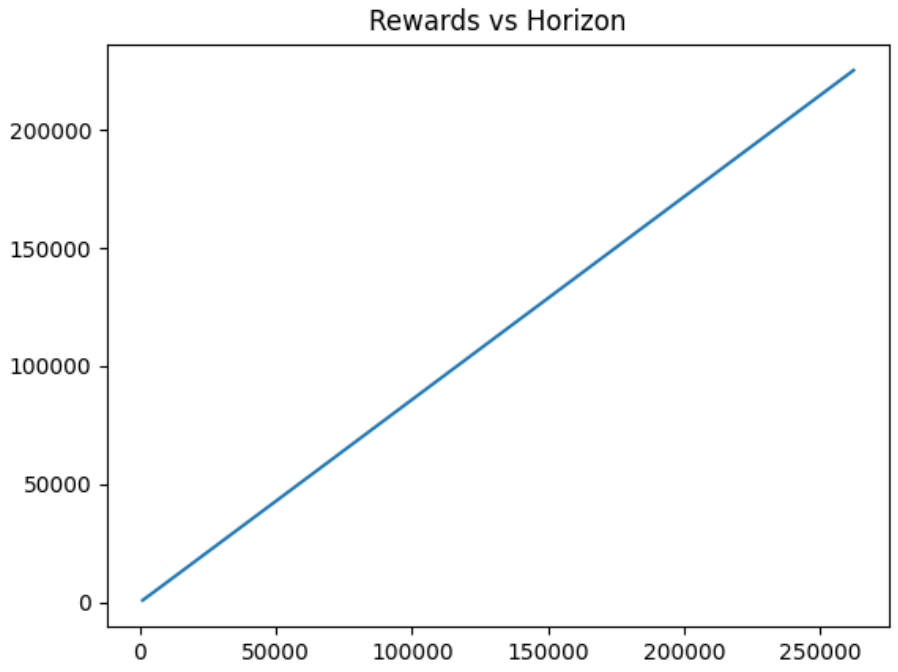
- Summary:

Thompson Sampling works in the faulty bandit problem because it adapts to estimate the true success probabilities of arms while taking into account the fault probability. This estimation is analogous to the previous problem in task 1, where it estimated the `p_i` values for normal bandits. By considering the combined effect of success and fault tolerance in the `new_p_i` values, Thompson Sampling maximises rewards in the presence of faults, making it a suitable algorithm for this scenario, hence the ideal choice of selection for me.

Code:

```python
class FaultyBanditsAlgo:
    def __init__(self, num_arms, horizon, fault):
        # You can add any other variables you need here
        self.num_arms = num_arms
        self.horizon = horizon
        self.fault = fault # probability that the bandit returns a faulty pull
        # START EDITING HERE
        self.thomps = np.zeros(num_arms)
        self.success = np.zeros(num_arms)
        self.pulls = np.zeros(num_arms)
        self.failure = lambda x: self.pulls[x] - self.success[x]
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        return np.argmax(self.thomps)
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        if reward == 1: self.success[arm_index] += 1
        self.pulls[arm_index] += 1
        self.thomps = [np.random.beta(self.success[i]+1, self.failure(i)+1) for i in range(self.num_arms)]
        #END EDITING HERE
```

Plot:



Rewards linearly increase with increase in Horizons on the faulty bandits scenario.

Fig. Plot of Rewards vs Horizons for Thompson Sampling algorithm on the Faulty bandits situation.

Proof that Thompson Sampling works and reasons and explanations for the same have also been provided. Reward Values: [799.14, 1659.46, 3399.36, 6905.12, 13927.08, 28004.08, 56151.52, 112512.74, 225214.98] for Horizon: [2**i for i in range(10, 19)]

Task 4)
Description of my approach to tackle the problems:

In the context of the multi-bandit problem, given an arm index i, any set of all bandits can be randomly chosen based on uniform probabilities, and arm index i in that set will be pulled.

- Analogy to a Simpler Problem:
    - In the simpler problem in Task 1 with a single-bandit instance, the success probability of an arm `i` was denoted as `p_i` (assume). The algorithm used empirical means to estimate these `p_i` values using beta distributions and selected the arm with the highest empirical mean.
    - Now, In the multiple-bandits problem, the true success probability of an arm `i` in a set `s` is denoted as `p_s_i`. Given an arm `i` to be pulled a random set `s` from all bandits is uniformly chosen and the arm `i` in set `s` has success probability of `p_s_i`. In estimations, we actually need to choose an arm `i` whose average `p_s_i` value from all sets is the maximum for best long term results.
    - `overall_p_i`= Σ [P(s) * p_s_i] for all sets `s`
    - For 2 uniform sets, overall_p_i = (p_1_i + p_2_i)/2
    - `overall_p_i`is called the average true success probability of an arm `i` of all sets.

The key insight here is that if we replace the multiple-bandits with a single-bandit situation and with `overall_p_i` value as true success probability of an arm `i`, both situations become exactly identical. This is because the `overall_p_i` value captures the effect of averaging the success probabilities across different sets, making it an appropriate metric for arm selection.

- Solution:
    - Thompson Sampling, when applied to the multiple-bandits problem, aims to estimate the `overall_p_i` values that captures the effect of averaging the success probabilities across different sets.
    - The algorithm estimates beta distributions of `overall_p_i` values, where distribution parameters are updated based on the observed rewards.
    - By estimating the beta distributions of the `overall_p_i` values for all the arms, Thompson Sampling effectively adapts to the effect of averaging success probabilities across different sets and selects arms that are likely to have the highest average true success probability i.e the one with the max `overall_p_i` values.

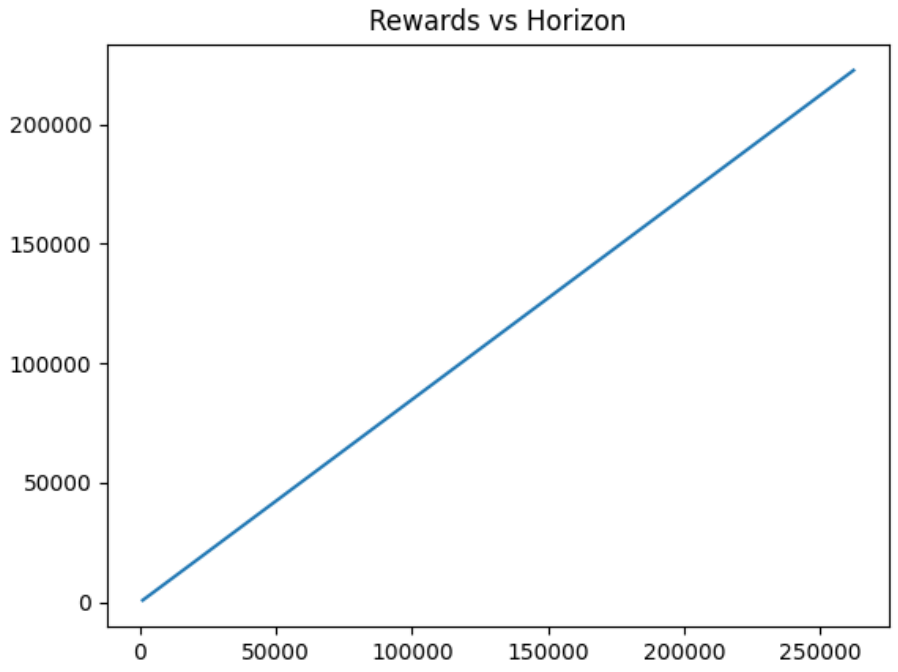
- Summary:

In the multi-bandit problem, Thompson Sampling proves its adaptability once again by estimating average true success probabilities across different sets, analogous to the single-bandit problem. The approach effectively leverages Bayesian inference and probabilistic modelling to optimise arm selection, making it an ideal choice for this scenario. By considering the effect of averaging success probabilities in `overall_p_i` values, Thompson Sampling provides a robust solution to maximise rewards across the multi-bandit problem, making it a suitable algorithm for this scenario, hence the ideal choice of selection for me.

Code:

```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
        # START EDITING HERE
        self.thomps = np.zeros(num_arms)
        self.success = np.zeros(num_arms)
        self.pulls = np.zeros(num_arms)
        self.failure = lambda x: self.pulls[x] - self.success[x]
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        return np.argmax(self.thomps)
        # END EDITING HERE

    def get_reward(self, arm_index, set_pulled, reward):
        # START EDITING HERE
        if reward == 1: self.success[arm_index] += 1
        self.pulls[arm_index] += 1
        self.thomps = [np.random.beta(self.success[i]+1, self.failure(i)+1) for i in range(self.num_arms)]
        # END EDITING HERE
```

Plot:

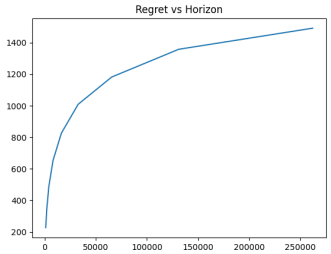


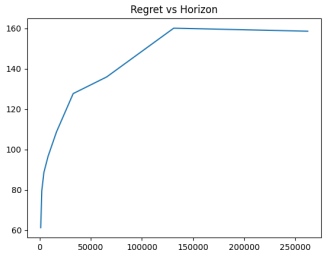
Rewards linearly increase with increase in Horizons on the multiple-bandits scenario.

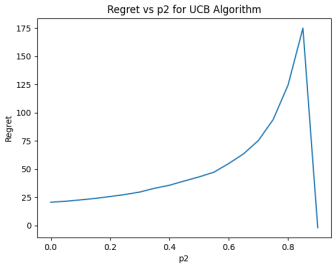
Fig. plot of rewards vs horizons for thompson sampling algorithm on the multiple-bandits situation.

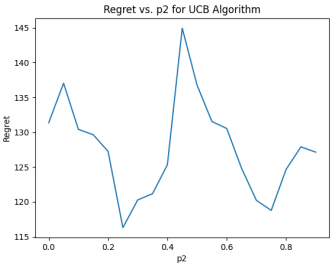
Proof that Thompson Sampling works and reasons and explanations for the same have also been provided. Reward Values: [791.9, 1642.88, 3370.28, 6842.8, 13786.38, 27699.04, 55529.38, 111215.62, 222582.38] for Horizon: [2**i for i in range(10, 19)]

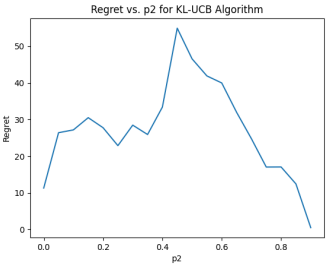
All Plots

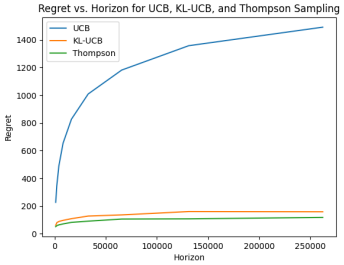


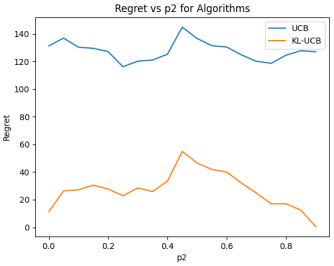


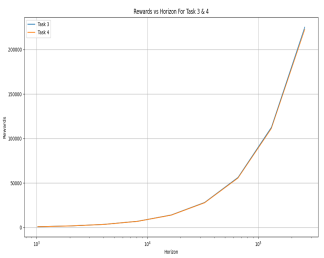












References:

- Some equation images and logics from: https://www.cse.iitb.ac.in/~shivaram/teaching/cs747-a2023/index.html
- To understand simulations and getting started: https://blog.devgenius.io/how-to-solve-the-multi-armed-bandit-problem-epsilon-greedy-approach-ebe286390578
- Plotting with legends and subplots from: https://www.geeksforgeeks.org/matplotlib-pyplot-legend-in-python/
- Working of UCB algorithm base understanding from: https://en.wikipedia.org/wiki/Multi-armed_bandit
- Literature read on Thompson Sampling from: https://medium.com/analytics-vidhya/a-brief-overview-of-the-multi-armed-bandit-in-reinforcement-learning-d086853dc90a

THANK YOU

Report by - Dhruv Jain (22M0828)

Date: 10th September, 2023.