CS747-Assignment1

Shrey Ganatra - 20d070074

September 2023

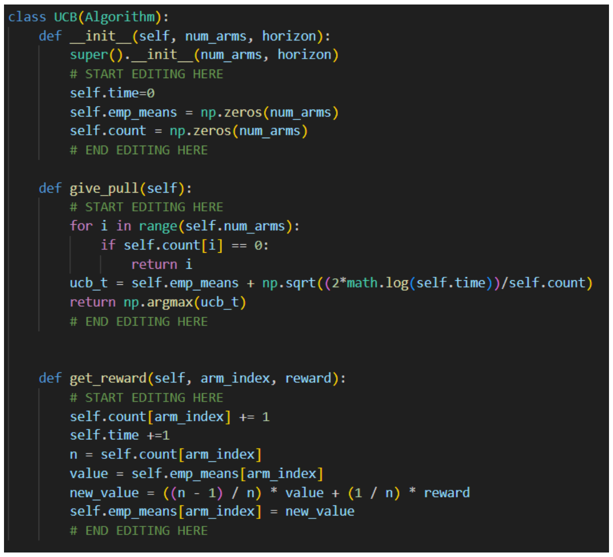
1 Introduction

This assignment tests understanding of the regret minimisation algorithms discussed in class, and ability to extend them to different scenarios. There are 4 tasks. To begin, in Task 1, implement UCB, KL-UCB, and Thompson Sampling, more or less identical to the versions discussed in class. Task 2A involves studying the effect of the difference between the means of two arms on the regret accumulated by the UCB algorithm. Task 2B involves studying and comparing the effect of the value of the means on the regret accumulated by UCB and KL-UCB while keeping the difference between the means constant. Task 3 involves maximising the reward for a bandit setting where pulls are noisy and give faulty outputs with a certain probability. Task 4 involves dealing with multiple bandit instances where the bandit instance for a particular pull is chosen at random. Task would be to maximise the reward for this multiple-bandit setting.

2 Task 1

2.1 UCB

2.1.1 Code



```
class UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        # START EDITING HERE
        self.time=0
        self.emp_means = np.zeros(num_arms)
        self.count = np.zeros(num_arms)
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        for i in range(self.num_arms):
            if self.count[i] == 0:
                return i
        ucb_t = self.emp_means + np.sqrt((2*math.log(self.time))/self.count)
        return np.argmax(ucb_t)
        # END EDITING HERE


    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        self.count[arm_index] += 1
        self.time +=1
        n = self.count[arm_index]
        value = self.emp_means[arm_index]
        new_value = ((n - 1) / n) * value + (1 / n) * reward
        self.emp_means[arm_index] = new_value
        # END EDITING HERE
```

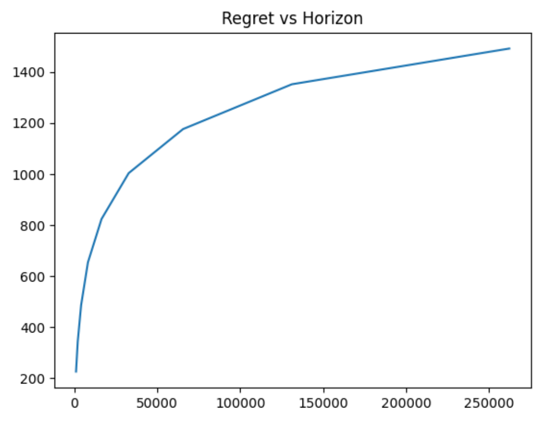
The algorithm is initialized with three member variables: time, emp_means, and count

- time keeps track of the total number of times the arms are pulled
- emp_means contains the empirical mean of each arm
- count keeps track of the number of pulls of each arm

The give_pull() function returns the arm with the maximum ucb value, but first, it ensures that each arm is pulled at least once.

The get_reward() function updates the time variable as an arm has been pulled and also updates the count and empirical mean for the arm pulled.

2.1.2 Regret plot

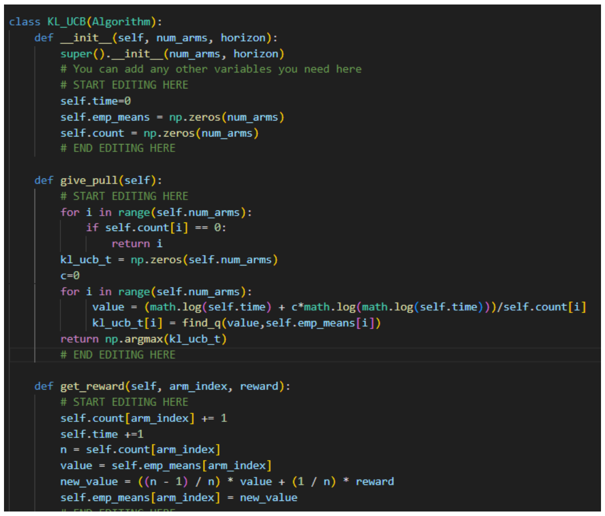


Regret vs Horizon plot for UCB algorithm

For the UCB algorithm, the regret scales logarithmically and achieves sub-linear regret.

2.2 KL-UCB

2.2.1 Code



```python
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.time=0
        self.emp_means = np.zeros(num_arms)
        self.count = np.zeros(num_arms)
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        for i in range(self.num_arms):
            if self.count[i] == 0:
                return i
        kl_ucb_t = np.zeros(self.num_arms)
        c=0
        for i in range(self.num_arms):
             value = (math.log(self.time) + c*math.log(math.log(self.time)))/self.count[i]
             kl_ucb_t[i] = find_q(value,self.emp_means[i])
        return np.argmax(kl_ucb_t)
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        self.count[arm_index] += 1
        self.time +=1
        n = self.count[arm_index]
        value = self.emp_means[arm_index]
        new_value = ((n - 1) / n) * value + (1 / n) * reward
        self.emp_means[arm_index] = new_value
```

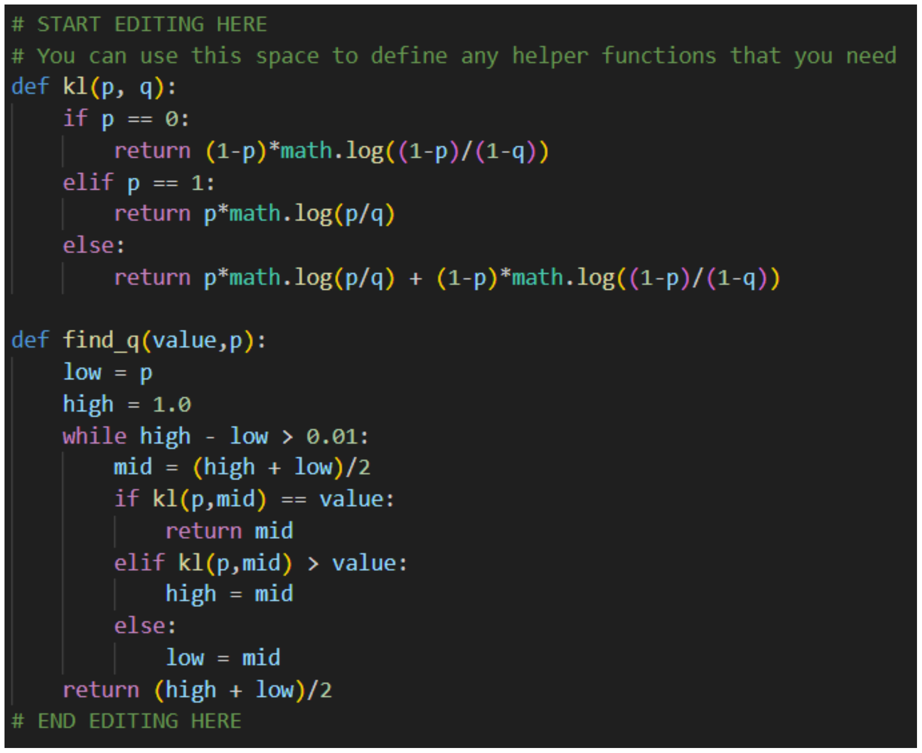
The algorithm is initialized with three member variables: time, emp_means, and count

- time keeps track of the total number of times the arms are pulled
- emp_means contains the empirical mean of each arm

- count keeps track of the number of pulls of each arm

The give_pull() function returns the arm with maximum kl-ucb value, but first, it ensures that each arm is pulled at least once.

The get_reward() function updates the time variable as an arm has been pulled and also updates the count and empirical mean for the arm pulled.

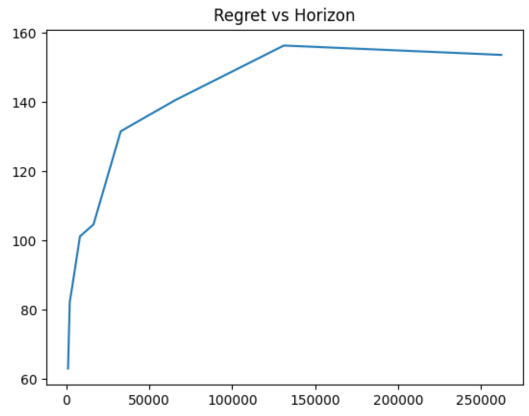


```python
# START EDITING HERE
# You can use this space to define any helper functions that you need
def kl(p, q):
    if p == 0:
        return (1-p)*math.log((1-p)/(1-q))
    elif p == 1:
        return p*math.log(p/q)
    else:
        return p*math.log(p/q) + (1-p)*math.log((1-p)/(1-q))

def find_q(value,p):
    low = p
    high = 1.0
    while high - low > 0.01:
        mid = (high + low)/2
        if kl(p,mid) == value:
            return mid
        elif kl(p,mid) > value:
            high = mid
        else:
            low = mid
    return (high + low)/2
# END EDITING HERE
```

The helper functions help us to ucb-kl numerically (through binary search)

2.2.2 Regret plot

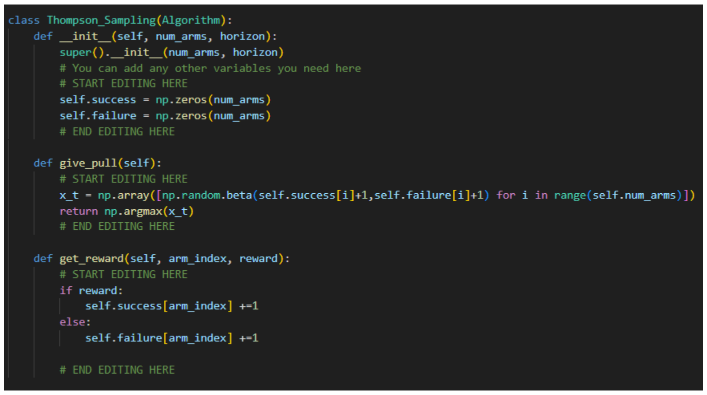


Regret vs Horizon plot for KL-UCB algorithm

For the KL-UCB algorithm, the regret scales logarithmically and achieves sublinear regret. It also gives tighter bound than UCB.

2.3 Thompson Sampling

2.3.1 Code



```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.success = np.zeros(num_arms)
        self.failure = np.zeros(num_arms)
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        x_t = np.array([np.random.beta(self.success[i]+1,self.failure[i]+1) for i in range(self.num_arms)])
        return np.argmax(x_t)
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        if reward:
            self.success[arm_index] +=1
        else:
            self.failure[arm_index] +=1

        # END EDITING HERE
```

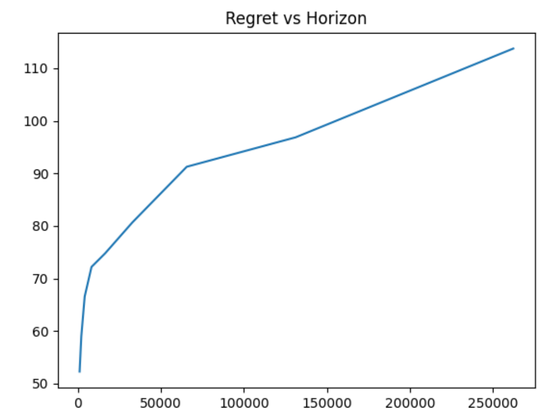
The algorithm is initialized with two member variables: success and failure

- success keeps track of the number of times each arm pull was successful
- failure keeps track of the number of times each arm pull was not successful

The give_pull() function returns the arm with the maximum value when sampled from the beta distribution over each arm.

The get_reward() function updates the success and failure variables depending on the reward received.

2.3.2 Regret plot

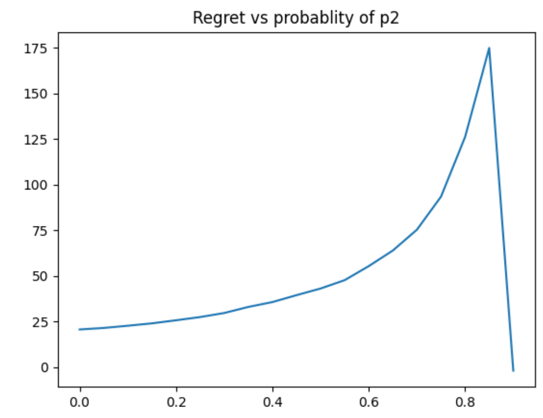


Regret vs Horizon plot for Thompson Sampling algorithm

For the Thompson Sampling algorithm, the regret scales logarithmically and achieves sub-linear regret. It also gives a tighter bound than KL-UCB.

3 Task 2

3.1 Task 2a

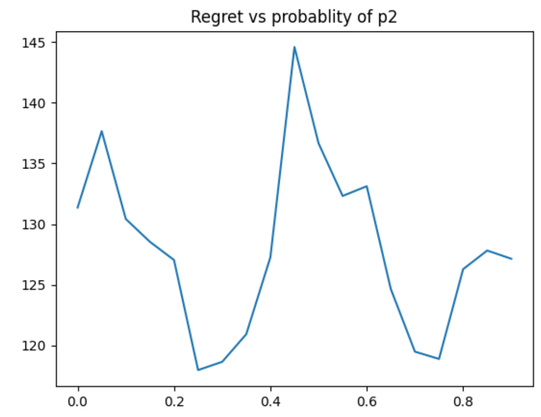


Variation of regret of UCB as the true mean of arm 2 varies from 0 to 0.9 in steps of 0.05 while the mean of arm 1 remains constant at 0.9

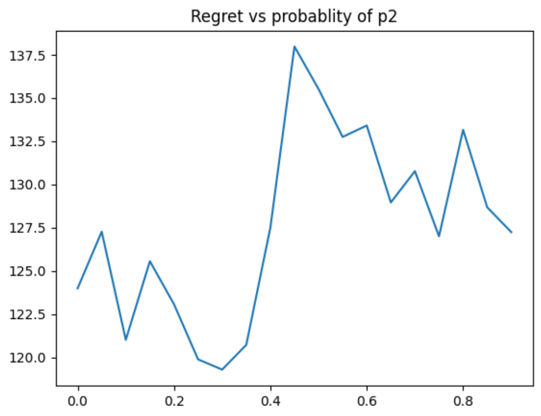
Observation: As the true means of both the arms come closer it becomes difficult for the UCB algorithm to find out the optimal arm which increases the number of pulls of sub-optimal arm which increases the regret. We see the decline in regret as both arms have become optimal.

3.2 Task 2b

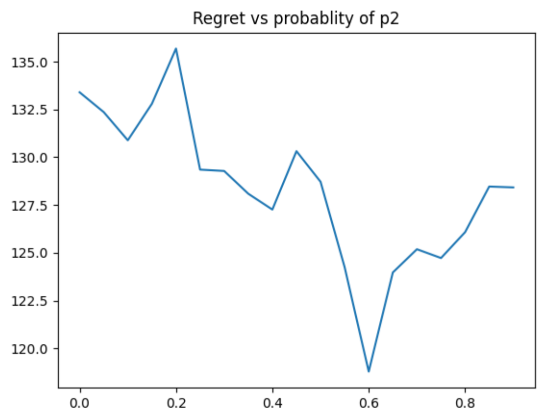
3.2.1 UCB



Variation of regret of UCB as the true mean of arm 2 varies from 0 to 0.9 in steps of 0.05 while the difference between the means of arms' remains fixed at 0.1 for num_sims=50



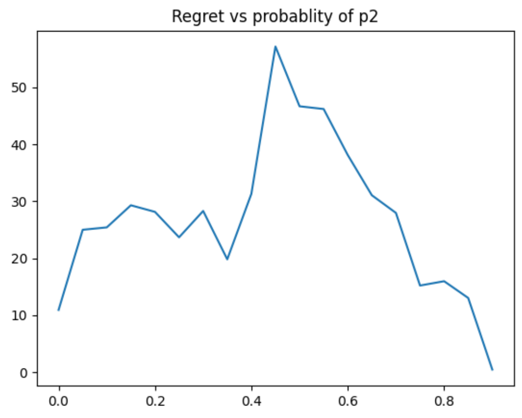
Variation of regret of UCB as the true mean of arm 2 varies from 0 to 0.9 in steps of 0.05 while the difference between the means of arms' remains fixed at 0.1 for num_sims=100



Variation of regret of UCB as the true mean of arm 2 varies from 0 to 0.9 in steps of 0.05 while the difference between the means of arms' remains fixed at 0.1 for num_sims=200

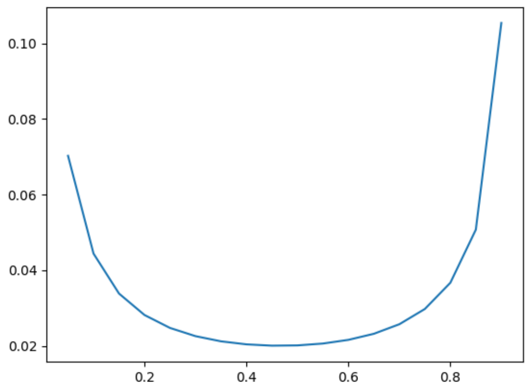
Observation: I have used equation 3 of reference 1, to match the understanding with theory. The expected regret of UCB is inversely proportional to the delta of means. As the number of simulations increases the spread across regret decreases. This indicates that as we do more simulations the expected regret depends only on the difference between the means of optimal and sub-optimal arms.

3.2.2 KL-UCB



Variation of regret of KL-UCB as the true mean of arm 2 varies from 0 to 0.9 in steps of 0.05 while the difference between the means of arms' remains fixed at 0.1

Observation: Theorem 1 of reference shows that the expected regret of KL-UCB is inversely proportional to the KL-Divergence between optimal and sub-optimal arms. If we plot the KL-Divergence for our bandit, we see the minimum around p2=0.5 which then corresponds to maximum regret around p2=0.5



Variation of regret of KL-Divergence as the true mean of arm 2 varies from 0 to 0.9 while the difference between the means of arms' remains fixed at 0.1

4 Task 3

Given the fault probability p_f and assuming the true mean of the arm to be p_a,

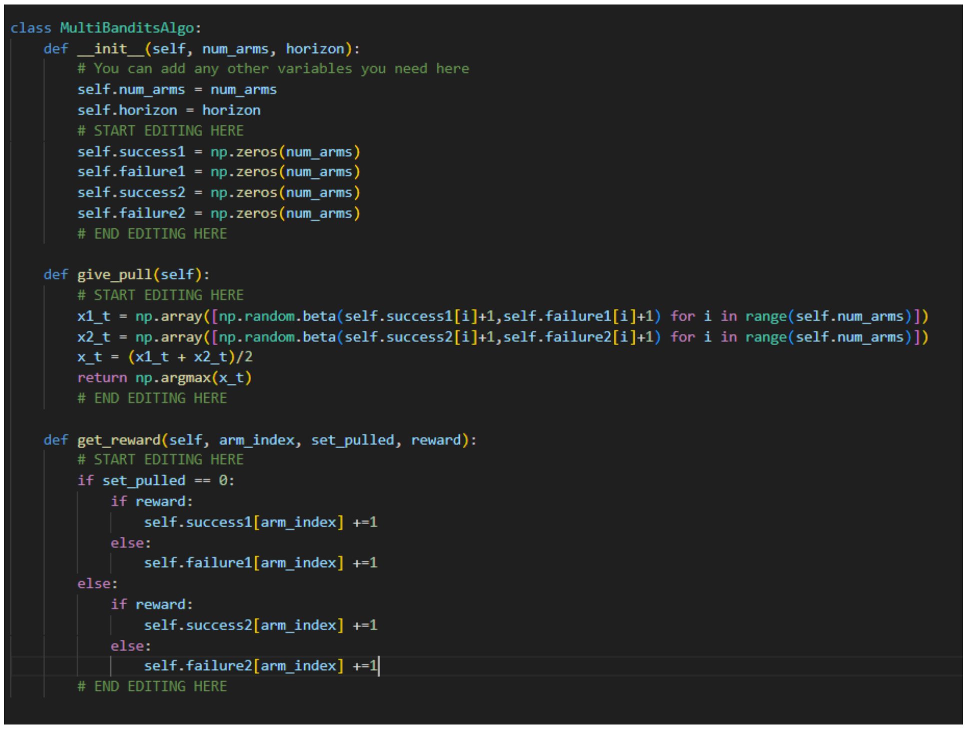
$$\mathcal{P}(1) = \mathcal{P}(fault) * \mathcal{P}(1|fault) + \mathcal{P}(notfault) * \mathcal{P}(1|notfault)$$
$$\mathcal{P}(1) = p_f * 0.5 + (1 - p_f) * p_a$$

$$\mathcal{P}(0) = \mathcal{P}(fault) * \mathcal{P}(0|fault) + \mathcal{P}(notfault) * \mathcal{P}(0|notfault)$$
$$\mathcal{P}(0) = p_f * 0.5 + (1 - p_f) * (1 - p_a)$$

So the arm can be considered to have a true mean of $p'_a = p_f * 0.5 + (1 - p_f) * p_a$. The problem can now be treated as a normal MAB with modified means. Hence, Thomspon Sampling works well and achieves sub-linear regret.

5 Task 4

This problem can also be simplified into a single bandit instance with $p' = 0.5 * p_1 + 0.5 * p_2$. But, here we have additional information on how each instance is performing(which was not known in Task 3 when the fault is happening). So, I have applied a modified Thompson Sampling algorithm. We keep track of success and failure from each instance and update the belief over each arm in both instances. Now, while picking up the best arm, we sample from both instances and then take the average over them as the instance is chosen uniformly at random (we can also take the weighted average if the probabilities are different) and pick the arm with the maximum value.



```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.success1 = np.zeros(num_arms)
        self.failure1 = np.zeros(num_arms)
        self.success2 = np.zeros(num_arms)
        self.failure2 = np.zeros(num_arms)
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        x1_t = np.array([np.random.beta(self.success1[i]+1,self.failure1[i]+1) for i in range(self.num_arms)])
        x2_t = np.array([np.random.beta(self.success2[i]+1,self.failure2[i]+1) for i in range(self.num_arms)])
        x_t = (x1_t + x2_t)/2
        return np.argmax(x_t)
        # END EDITING HERE

    def get_reward(self, arm_index, set_pulled, reward):
        # START EDITING HERE
        if set_pulled == 0:
            if reward:
                self.success1[arm_index] +=1
            else:
                self.failure1[arm_index] +=1
        else:
            if reward:
                self.success2[arm_index] +=1
            else:
                self.failure2[arm_index] +=1
        # END EDITING HERE
```

6 References

1. Garivier, Aurélien, and Olivier Cappé. "The KL-UCB algorithm for bounded stochastic bandits and beyond." Proceedings of the 24th annual conference on learning theory. JMLR Workshop and Conference Proceedings, 2011.