Indian Institute of Technology, Bombay

CS 747: Foundations of Intelligent and Learning Agents

Report On

Programming Assignment-1

Regret Minimisation Algorithms on Mutli-Armed Bandit Instances

---

Author: Swadhin Dash
Roll Number: 210020142
Course Instructor: Shivaram Kalyanakrishnan

Abstract

In this report, we look at regret-minimizing algorithms for multi-armed bandit problems. Task 1 involves implementation and comparison of three algorithms: Upper Confidence Bound (UCB-1), Kullback-Leibler Upper Confidence Bound (KL-UCB), and Thompson Sampling. In Task 2A we analyze how mean differences between arms affect UCB's regret. In Task 2B, we explore the impact of mean values on UCB and KL-UCB regret while keeping mean differences constant. Task 3 addresses the challenge of reward maximization in a noisy bandit setting with probabilistic faults. Task 4 involves dealing with multiple bandit instances where the bandit instance for a particular pull is chosen at uniformly at random, aiming to maximize rewards across diverse scenarios.

Contents

List of Figures

1 Task 1

This task involves implementation of the sampling algorithms: UCB, KL-UCB, and Thompson Sampling.

1.1 Upper Confidence Bound (UCB-1)

At time t, the Upper Confidence Bound (UCB) for each arm a is defined as follows:

$$UCB_a^t = \hat{p}_t^a + \sqrt{\frac{2\ln(t)}{u_a^t}} \tag{1}$$

Where:

- $\hat{p}_t^a$ is the empirical mean of rewards from arm a up to time t.
- u_t^a is the number of times arm a has been sampled at time t.

The UCB algorithm selects the arm a for which UCB_t^a is maximum at each time step.

1.1.1 Code and Implementation Details

```
class UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        self.time_step = 0
        self.counts = np.zeros(num_arms)
        self.values = np.zeros(num_arms)

    def give_pull(self):
        if self.time_step < self.num_arms:
            return self.time_step
        else:
            ucb = self.values + np.sqrt((2 * math.log(self.time_step))
                / self.counts)
            return np.argmax(ucb)

    def get_reward(self, arm_index, reward):
        self.time_step += 1
        self.counts[arm_index] += 1
        self.values[arm_index] += ((reward - self.values[arm_index])
            / self.counts[arm_index])
```

1. I initialize the expected reward values and hence the UCB's of all the arms to 1 since initially I don't have prior information about any arm and I optimistically assume each arm to be equally likely to be the best arm.

2. I maintain a count of the number of pulls an arm received initialized to zero and increment it each time the arm is pulled.

3. For the first N time-steps I pull each of the N arms once to initialize the count array to ones. Then in the later time-steps I calculate the UCB values of each arm according to equation (1).

4. Each time I receive a reward corresponding to an arm pull, I use simple averaging to increment the expected reward of that arm using the previous value and the number of times it has been pulled.

1.1.2 Simulation of regrets over different Horizons

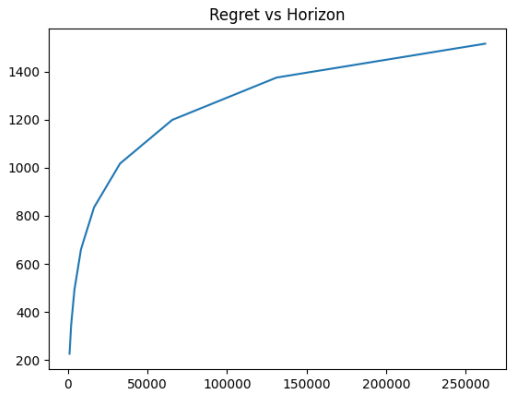

Figure 1: Regret vs Horizon of the UCB Algorithm

1.2 Kullback-Leibler Upper Confidence Bound (KL-UCB)

The Kullback-Leibler Upper Confidence Bound (KL-UCB) algorithm is closely related to the UCB algorithm but differs in its definition of the upper confidence bound. At time t, the KL-UCB for each arm a is defined as follows:

$$ucb{-}kl_t^a = \max\{q \in [\hat{p}_t^a, 1] s.t. u_t^a KL(\hat{p}_t^a, q) \leq \ln(t) + c\ln(\ln(t))\} \tag{2}$$

Where:

- $\hat{p}_t^a$ is the empirical mean of rewards from arm a up to time t.
- u_t^a is the number of times arm a has been sampled at time t.
- $c \geq 3$ is a constant parameter.

We have to find the solution $q \in [\hat{p}_t^a, 1]$ to $KL(\hat{p}_t^a, q) = \frac{\ln(t)+c\ln(\ln(t))}{u_t^a}$.

The KL-UCB algorithm, at each step t, selects the arm a with the maximum $ucb{-}kl_t^a$. It is known to be a tighter confidence bound compared to the standard UCB.

1.2.1 Code and Implementation details

```
def KL_DIVG(x, y):
    if x == 0:
        return (1 - x) * math.log((1 - x) / (1 - y))
    elif x == 1:
        return float('inf')
    else:
        return x * math.log(x / y) + (1 - x) * math.log((1 - x) / (1 - y))

def solve_q(p, c):
    epsilon = 1E-4
    q = p
    b = (1 - p) / 2
    while b > epsilon:
        if KL_DIVG(p, q + b) <= c:
            q += b
        b /= 2
    return q

def KL(values, counts, c, t):
    q = np.zeros(len(values), dtype=np.float32)
    for arm in range(len(values)):
        val = (math.log(t) + c * math.log(math.log(t))) / counts[arm]
        q[arm] = solve_q(values[arm], val)
    return q

class KL_UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        self.c = 0
        self.time_step = 0
        self.counts = np.zeros(num_arms)
        self.values = np.ones(num_arms)

    def give_pull(self):
        if self.time_step < self.num_arms:
            return self.time_step
        else:
            ucb = KL(self.values, self.counts, self.c, self.time_step)
            return np.argmax(ucb)

    def get_reward(self, arm_index, reward):
```

```
self.time_step += 1
self.counts[arm_index] += 1
self.values[arm_index] += ((reward - self.values[arm_index])
    /self.counts[arm_index])
```

1. Initialization of expected rewards and counts of each arm is same as in UCB.
2. For the first N time-steps I pull each of the N arms once to initialize the count array to ones. Then in the later time-steps I calculate the $KL - UCB$ values of each arm according to equation (2).
3. The reward reception and updation of expected rewards of arms is again same as in UCB.
4. I use $c = 0$ in my solution since in the actual paper authors describe c as a factor that increases the regret upper bound and smaller the c, lesser is the regret.
5. I implement *bisection method* with a precision of $\epsilon = 10^{-4}$ to solve for the q value corresponding to each arm. I make sure that corner cases for the KL Divergence function are handled properly, which include $x = 1, y = 1, x = 0, y = 0$.

1.2.2 Simulation of regrets over different Horizons

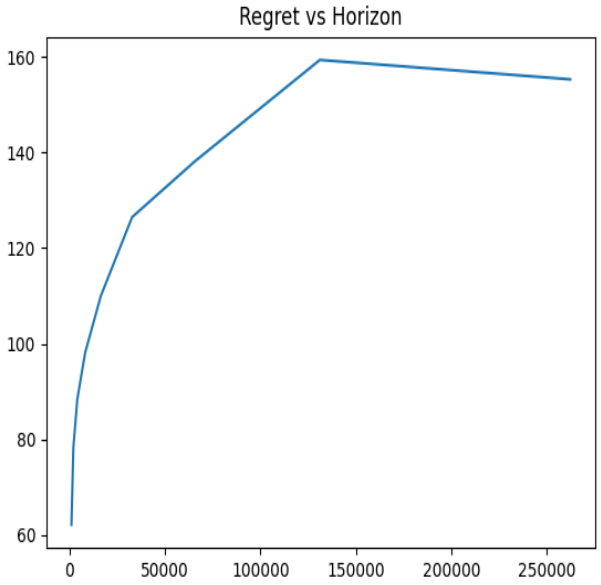

Figure 2: Regret vs Horizon of the KL-UCB Algorithm

1.3 Thompson Sampling Algorithm

The Thompson Sampling algorithm takes a Bayesian probabilistic approach to make decisions. At each time step t, the Thompson Sampling algorithm operates as follows:

1. For each arm a, maintains a posterior distribution, typically a Beta distribution, over the true mean reward of that arm.

2. Sample a value from each arm's posterior distribution. These samples represent the estimated mean rewards of the arms.

3. Choose the arm with the highest sampled value. In other words, select the arm that the algorithm currently believes has the highest expected reward.

4. Play the chosen arm, observe the actual reward, and update the posterior distribution for that arm based on the observed outcome.

The Bayesian nature of Thompson Sampling allows it to naturally explore and exploit the arms based on uncertainty in their reward distributions. Over time, as more observations are collected, the algorithm tends to focus more on exploiting arms with higher expected rewards.

1.3.1 Code and Implementation details

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

    def give_pull(self):
        draws = [np.random.beta(self.successes[i] + 1,
            self.fails[i] + 1) for i in range(self.num_arms)]
        return np.argmax(draws)

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

1. I initialize all the alphas and betas for the beta distribution functions for each arm to 1. This signifies that at the start the mean of the reward distributions is 0.5 since I have no prior information.

2. During each pull, I randomly sample from the beta distributions of each arm and select the arm that returned the maximum value.

3. If the reward received for an arm is 1, I increment the alpha value for its distribution otherwise the beta value is incremented. This effectively updates the posterior distribution to ensure exploit explore balance.

1.3.2 Simulation of regrets over different Horizons

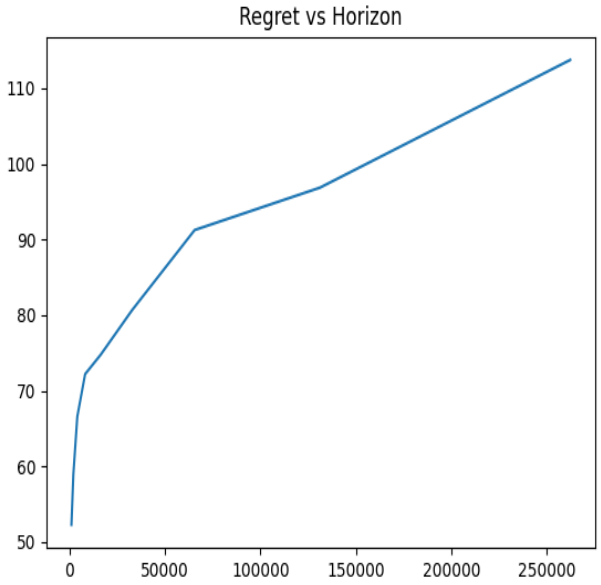

Figure 3: Regret vs Horizon of the Thompson Sampling Algorithm

2 Task 2

2.1 Part A

This task explores the effect of the difference between the means of the arms on the regret accumulated by the UCB algorithm. I take a two-armed bandit instance with the higher mean arm fixed at $p_1 = 0.9$, and vary the other arm's mean p_2 from 0 to p_1 in steps of 0.05. I do this for a horizon of 30000 and plot the variation of regret with p2.

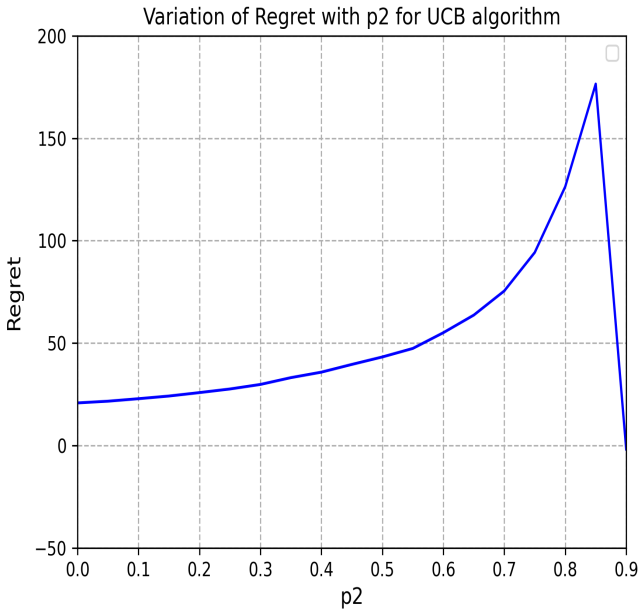

Figure 4: Variation of regret with p_2 for p_1 fixed at 0.9

It can be clearly observed that as the difference between the means of the arms decreases, it becomes harder for the UCB algorithm to distinguish which is the best arm. So the regret increases until $p_1 = 0.9$ and $p_2 = 0.85$, where its the hardest to distinguish. Finally at $p_1 = 0.9$ and $p_2 = 0.9$, whichever arm the algorithm picks, it is always the optimal arm so expected regret is zero and hence we can see the actual regret also falls.

2.2 Part B

This task involves comparing the behaviour of UCB and KL-UCB algorithms on two-armed bandits. We compare the effect of the value of the means on the algorithms while keeping the difference between the means fixed. We take p_1 - $p_2 = 0.1$ and vary p_2 from 0 to 0.9 (both inclusive) in steps of 0.05 for a horizon of 30000. We plot the variation of regret for both algorithms.

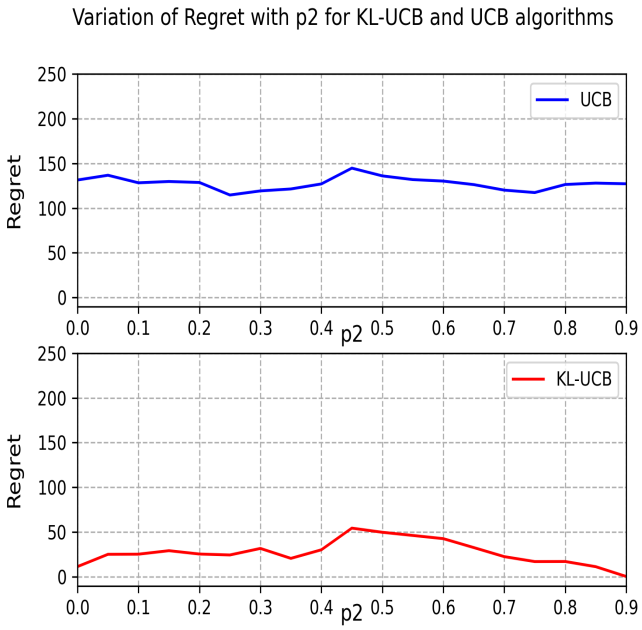

Figure 5: Variation of regret with p_2 for UCB and KL-UCB Algorithms

One striking observation is the fact that the regret of UCB is much larger than $KL-UCB$, which is supported by theory. The average regret of the UCB algorithm is-

$$Regret \simeq \sum_{a:p_a \neq p^*} \frac{c * \log(t)}{p_a - p^*} \tag{3}$$

where c is a constant and essentially $p_a - p^*$ is 0.1, so in the limit of infinite simulations the regret is a constant.

Similarly the expected value of regret in $KL-UCB$ is given by-

$$Regret \simeq \sum_{a:p_a \neq p^*} \frac{(p_a - p^*)\log(t)}{KL(p_a, p^*)} \tag{4}$$

The term $KL(p_a, p^*)$ is minimum around $p_* = 0.55$ and the value is 0.02 and $p_a - p^*$ $= 0.1$, so in the limit of infinite simulations it should be a downward facing parabola and the value of regret is around 51, which is visible from the graph as a point of local maxima.

3 Task 3

This task involves dealing with a bandit instance where your pulls are no longer guaranteed to be successful and have a probability of giving faulty outputs. When you pull

an arm, it has a certain known probability of giving the correct output of the arm, otherwise it returns a 0 or 1 uniformly at random.

3.1 My Approaches

1. Firstly I tried to use Bayesian Approach with probability adjustments. I counted each reward of 1 as $1 - \frac{f}{2}$ and each reward of 0 as $\frac{f}{2}$ and applied KL-UCB. This approach though it sounds great failed because I'm making an assumption about the true distribution of rewards. If the environment is truly stochastic with a probability f of returning a random 0 or 1, this normalization might not accurately capture the environment's behavior.

2. Next I modified Thompson Sampling parameters (alphas and betas) based on the faulty environment, attempting to incorporate the probabilistic behavior of the environment. This modification is closer to modeling the environment's behavior, but it still failed to capture the true distribution accurately.

3. Then I realize that there are two features in the given setting, and I can apply contextual bandits to solve the problem. I maximize the cumulative reward over time by learning actions based on the observed contexts and rewards. I use two features - 1 and P_fault. 1 allows the agent to model the baseline or average reward that an arm might yield whereas P_fault in the context represents the fault probability. which provides additional information that the agent can use to make more informed decisions.

4. Simply running the Thompson Sampling algorithm on the above setting performs the better than all above because it doesn't make strong assumptions about the underlying reward distribution. It adapts to the environment by sampling from the current belief distributions without trying to explicitly model the environment's probabilistic behavior. Rather it's evident that applying a linear transform on a probability distribution will not change anything, an algorithm will solve it irrespective effectively.

3.2 Code

```
class FaultyBanditsAlgo:
    def __init__(self, num_arms, horizon, fault):
        self.num_arms = num_arms
        self.horizon = horizon
        self.fault = fault

        self.alphas = np.ones((num_arms, 2))
        self.betas = np.ones((num_arms, 2))
        self.context = np.array([1, fault]) # Contextual bandits
```

```
def give_pull(self):
    draws = np.dot(np.random.beta(self.alphas, self.betas), self.context)
    return np.argmax(draws)

def get_reward(self, arm_index, reward):
    self.alphas[arm_index] += reward * self.context
    self.betas[arm_index] += (1 - reward) * self.context
```

4 Task 4

This task involves dealing with two bandit instances at once. Whenever I specify an arm to be pulled, one of two given bandit instances is chosen uniformly at random, and the arm is pulled. I had to come up with a good algorithm to maximise the reward for this multi-multi-armed bandit setting.

4.1 My Approaches

1. Firstly I applied Thompson Sampling by maintaining two sets of arms and their beta distributions. I defined an $\epsilon = 0.9$ and generated a random number r between 0 and 1 during each pull. If r <= ϵ, I drew samples from both sets of beta distributions and created an array with minimum of the two samples for the same arm. Then I found the arm with the maximum value in the array. This I refer to as *playing safe*. And when $r > \epsilon$, I do the very opposite, maximising the maximum of the samples, *risky play*.

2. I tried tuning ϵ and even checked performance of $\epsilon = 1$ and $\epsilon = 0$. But this couldn't effectively solve bandit problems and I believe the reason can be insufficient exploration. Time varying or adaptive ϵ might be a better alternative.

3. I created another bandit instance that recorded the number of pulls to each set and kept a beta distribution according to which it predicted the possible next pull. But it's essentially useless as the probability of a set getting selected is 0.5 always and the second bandit will only *skew* and misrepresent the environment.

4. Then I applied contextual bandits to the above problem by simply adding up the two samples for each arm, which is essentially a context of $[1, 1]$ i.e both sets have to be captured into the average or baseline reward an arm might yield. This is exactly similar to just applying a Thompson Sampling algorithm without caring about which arm was pulled actually because the algorithm will take the average of the expected rewards from both the sets *under the hood* without any prior.

4.2 Code

```
class MultiBanditsAlgo:
    def __init__(self, num_arms, horizon):
        self.num_arms = num_arms
        self.horizon = horizon
        self.alphas = np.ones((num_arms, 2))
        self.betas = np.ones((num_arms, 2))
        # self.context = np.array([1 , 1])

    def give_pull(self):
        draws = np.random.beta(self.alphas[:, 0], self.betas[:, 0]) + np.random.beta(s
        return np.argmax(draws)

    def get_reward(self, arm_index, set_pulled, reward):
        # self.alphas[arm_index] += reward * self.context
        # self.betas[arm_inde] += (1 - reward) * self.context

        self.alphas[arm_index, set_pulled] += reward
        self.betas[arm_index, set_pulled] += (1 - reward)
```

References

[1] Reinforcement Learning: An Introduction by Richard Sutton and Andrew Barto