CS747 - Foundations of Intelligent and Learning Agents

Assignment 1 Report

Kaishva Chintan Shah (200020066)

Department of Electrical Engineering
Indian Institute of Technology, Bombay

September 2023

1 Task 1: Implementing Sampling Algorithms

In Task 1, we implemented three sampling algorithms: (1) UCB (Upper Confidence Bound), (2) KL-UCB (Kullback-Leibler Upper Confidence Bound), and (3) Thompson Sampling. These algorithms are designed for multi-armed bandit problems where the goal is to maximize the cumulative reward by selecting arms (actions) over a series of rounds.
We followed the structure of the provided `Algorithm` class, modifying the `__init__` function to initialize the necessary state variables for each algorithm. The key components of each algorithm were implemented in the `give_pull` and `get_reward` functions.
The `give_pull` function was responsible for deciding which arm to pull in each round. It returned the index of the selected arm based on the algorithm's decision criteria. The `get_reward` function handled the feedback from the environment, updating the algorithm's knowledge about each arm's reward based on the observed reward for the selected arm.
We conducted simulations using the implemented algorithms with various horizons to evaluate their performance in terms of cumulative regret. The regret measures the difference between the total reward obtained by the algorithm and the reward that could have been obtained by always selecting the best arm. The results were visualized in plots generated by `simulator.py`.

1.1 UCB Algorithm Implementation

The UCB (Upper Confidence Bound) algorithm is implemented as follows:

- **Initialization**:
  - The algorithm is initialized with the number of arms, denoted as `num_arms`, and the horizon, denoted as `horizon`.
  - Three arrays are initialized:
    1. `self.ucb`: An array to track the UCB values for each arm. It is initialized with ones for each arm.
    2. `self.counts`: An array to count the number of times each arm is pulled. It is initialized with ones to avoid division by zero.
    3. `self.values`: An array to store the average rewards for each arm. It is initially set to zeros.
- **Arm Selection** (`give_pull`):
  - The algorithm selects an arm to pull based on the UCB values.
  - If multiple arms have the maximum UCB value, the algorithm randomly chooses one to pull.
  - Otherwise, it selects the arm with the highest UCB value.
- **Reward Update** (`get_reward`):

  - When a reward is received for an arm, the algorithm updates its internal state.
  - It increments the count for the selected arm and calculates the new average reward.
  - The UCB values are updated with an exploration term based on the number of total counts across all arms and the counts of individual arms.

1.1.1 Parameter Settings

The UCB algorithm employs the following parameters:

- `num_arms`: The number of arms in the bandit instance specified during initialization.
- `horizon`: The number of rounds or time steps for which the algorithm will operate. It is also specified during initialization.
- `self.ucb`: An array that tracks the UCB values for each arm. These values guide the arm selection process, balancing exploration and exploitation.
- `self.counts`: An array that tracks how many times each arm has been pulled.
- `self.values`: An array that stores the average rewards for each arm.

The algorithm dynamically updates these parameters based on the observed rewards and exploration-exploitation trade-offs. The UCB algorithm maximises its cumulative reward over the specified horizon by balancing these factors.

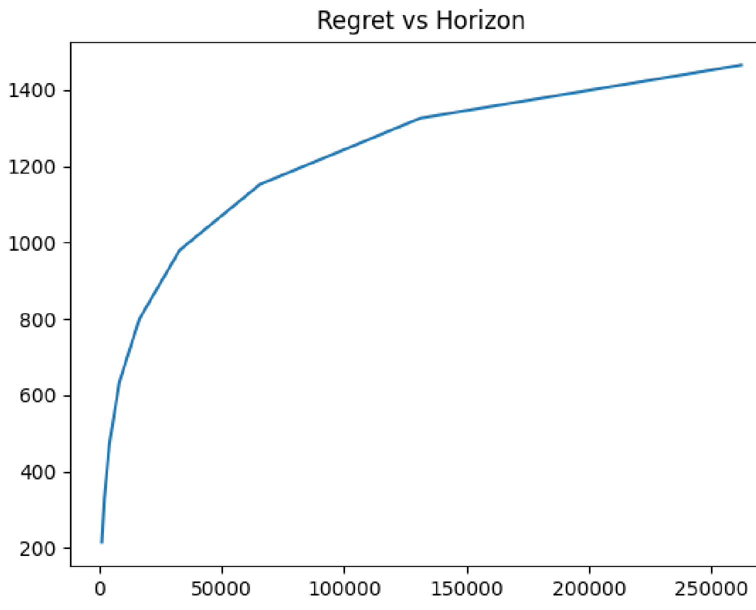


Figure 1.1: Regret curve for the UCB algorithm over different horizons.

1.2 KL-UCB Algorithm Implementation

The KL-UCB (Kullback-Leibler Upper Confidence Bound) algorithm is implemented as follows:

- **Initialization**:
  - The algorithm is initialized with the number of arms, denoted as `num_arms`, and the horizon, denoted as `horizon`.
  - Several arrays are initialized:
    1. `self.ucb_kl`: An array to track the UCB values using KL divergence. It is initialized with zeros for each arm.
    2. `self.counts`: An array to count the number of times each arm is pulled. It is initialized with ones to avoid division by zero.
    3. `self.values`: An array to store the average rewards for each arm. It is initially set to zeros.
    4. `self.bounds`: An array to store the right-hand side (RHS) of the inequality used for arm selection. It is initially set to zeros.
    5. `self.c`: A constant for finding `q` during arm selection. Set to 0.
- **Arm Selection (`give_pull`)**:
  - The algorithm selects an arm to pull based on the KL-UCB values.
  - If there are multiple arms with the maximum KL-UCB value, the algorithm randomly chooses one of them to pull.
  - Otherwise, it selects the arm with the highest KL-UCB value.
- **Reward Update (`get_reward`)**:
  - When a reward is received for an arm, the algorithm updates its internal state.
  - It increments the count for the selected arm and calculates the new average reward.
  - The algorithm also updates the `self.bounds` array contains the RHS of the inequality used in arm selection.
  - Finally, the KL-UCB values are updated by finding `q` using the `get_q_kl` function, which depends on `self.values` and `self.bounds`.

1.2.1 Parameter Settings

The KL-UCB algorithm employs the following parameters:

- `num_arms`: The number of arms in the bandit instance specified during initialization.
- `horizon`: The number of rounds or time steps for which the algorithm will operate. It is also specified during initialization.

- `self.ucb_kl`: An array that tracks the KL-UCB values for each arm. These values guide the arm selection process. They are initialised to zero.
- `self.counts`: An array that tracks how many times each arm has been pulled. Initialised to all ones
- `self.values`: An array that stores the average rewards for each arm. Initialised to all zeroes.
- `self.bounds`: An array that stores the right-hand side (RHS) of the inequality used for arm selection. Initialised to all zeroes.
- `self.c`: A constant used in finding `q` during arm selection. Set to 0.

1.2.2 KL Divergence Calculation (`vectorized_kl_divergence`)

The `vectorized_kl_divergence` function is used to calculate the Kullback-Leibler (KL) divergence between two sets of probability distributions represented by `p_values` and `q_values`. Here's how it works:

- **Initialization**:
  - The function uses a small constant, `epsilon`, to prevent division by zero and logarithmic issues. It ensures that `q_values` are greater than or equal to `epsilon`.
- **KL Divergence Calculation**:
  - The function calculates the KL divergence term-by-term for each element in `p_values` and `q_values`.
  - It computes two terms: `term1` and `term2`, which correspond to the two terms in the KL divergence formula.
  - In the `vectorized_kl_divergence` function, two intermediate terms, `term1` and `term2`, are computed to calculate the Kullback-Leibler (KL) divergence between probability distributions. These terms correspond to the components of the KL divergence formula:

$$\texttt{term1} = (1 - p_{\text{values}}) \cdot \log\left(\frac{1 - p_{\text{values}} + \epsilon}{1 - q_{\text{values}} + \epsilon}\right)$$

$$\texttt{term2} = p_{\text{values}} \cdot \log\left(\frac{p_{\text{values}} + \epsilon}{q_{\text{values}}}\right)$$

    Here's an explanation of each term:

    * `term1`:
      - `term1` corresponds to the first term in the KL divergence formula and is typically used when the probability value `p` is not equal to 0.0 (i.e., `p_values != 0.0`).

        · The logarithmic ratio of probabilities is computed, with small adjustments by the constant `epsilon` to prevent division by zero and handle logarithmic issues.

    * `term2`:

        · `term2` corresponds to the second term in the KL divergence formula and is typically used when the probability value `p` is not equal to 1.0 (i.e., `p_values != 1.0`).

        · Similar to `term1`, the logarithmic ratio of probabilities is computed with adjustments by `epsilon` to handle extreme cases.

  These terms are essential for calculating the KL divergence accurately between probability distributions, considering scenarios such as extreme probability values or values between 0.0 and 1.0.

  – The KL values are calculated using the formula:

$$\texttt{kl\_values} = \begin{cases} \texttt{term1} & \text{if } \texttt{p\_values} = 0.0 \\ \texttt{term2} & \text{if } \texttt{p\_values} = 1.0 \\ \texttt{term1 + term2} & \text{otherwise} \end{cases}$$

  This allows for efficient vectorized computation of KL divergences.

1.2.3 Arm Selection Using KL Divergence (`get_q_kl`)

The `get_q_kl` function is used to find the values of `q` for each arm based on KL divergence calculations and specified `bounds`. Here's how it works:

- **Initialization**:

  – The function initializes an array `q` to store the calculated `q` values.

  – Two arrays, `a` and `b`, are initialized. `a` represents the lower bound for `q`, initially set to `values`, while `b` represents the upper bound for `q`, initially set to ones.

  – A small constant, `epsilon`, is used to control the precision of the computation.

- **Binary Search**:

  – The function employs a binary search approach to iteratively refine the `q` values.

  – While there exists any arm for which `b - a` is greater than `epsilon`, the binary search continues.

  – In each iteration, the midpoint `c` between `a` and `b` is computed.

  – The KL divergence between `values` and `c` (denoted as `kl_c`) is calculated using the `vectorized_kl_divergence` function.

  – A mask is created to identify arms for which `kl_c` is less than or equal to the specified `bounds`.

  - The values of `a` and `b` are updated based on the mask. If `kl_c` is within bounds for an arm, `a` is set to `c` for that arm; otherwise, `b` is set to `c`.

- **Final `q` Calculation**:
  - Once the binary search converges (i.e., when `b - a` is no longer greater than `epsilon` for any arm), the final `q` values are calculated as the average of `a` and `b` for each arm.

These functions and parameter settings are crucial in the KL-UCB algorithm for efficient arm selection based on KL divergence and exploration-exploitation trade-offs.

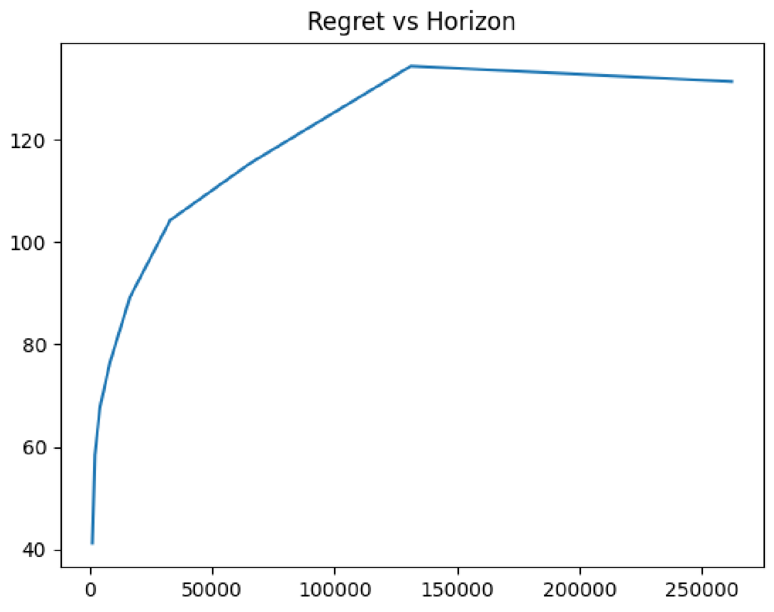


Figure 1.2: Regret curve for the KL-UCB algorithm over different horizons.

1.3 Thompson Sampling Algorithm

1.3.1 Beta Distribution Sampling (`get_beta`)

The `get_beta` function is used to sample from a Beta distribution with parameters `alpha` and `beta`. This distribution is commonly used in the Thompson Sampling algorithm to model uncertainty in the success probabilities of arms. Here's how it works:

- **Sampling from Beta Distribution**:
  - The function generates random samples from a Beta distribution with parameters `alpha + 1` and `beta + 1`. These parameters are typically updated based on each arm's observed successes and failures in the Thompson Sampling algorithm.
  - The Beta distribution allows the algorithm to model the uncertainty in the success probabilities of arms and make probabilistic decisions during arm selection.

1.3.2 Thompson Sampling Algorithm (`Thompson_Sampling`)

The `Thompson_Sampling` class implements the Thompson Sampling algorithm for solving the multi-armed bandit problem. Here's how it works:

- **Initialization**:
  - The algorithm initializes two arrays, `success` and `failures`, each containing ones for all arms. These arrays keep track of the number of observed successes and failures for each arm.
- **Arm Selection** (`give_pull`):
  - In each round, the algorithm uses the `get_beta` function to sample success probabilities for all arms based on the observed successes and failures.
  - If multiple arms have the highest sampled success probability, the algorithm randomly selects one.
  - Otherwise, it selects the arm with the highest sampled success probability.
  - This probabilistic arm selection allows the algorithm to balance exploration and exploitation.
- **Reward Update** (`get_reward`):
  - When a reward is obtained after pulling an arm, the algorithm updates the corresponding arm's success and failure counts.
  - If the reward is 1 (indicating success), the `success` count for the chosen arm is incremented.
  - If the reward is 0 (indicating failure), the `failures` count for the chosen arm is incremented.
  - These updates help the algorithm refine its estimates of arm success probabilities over time.

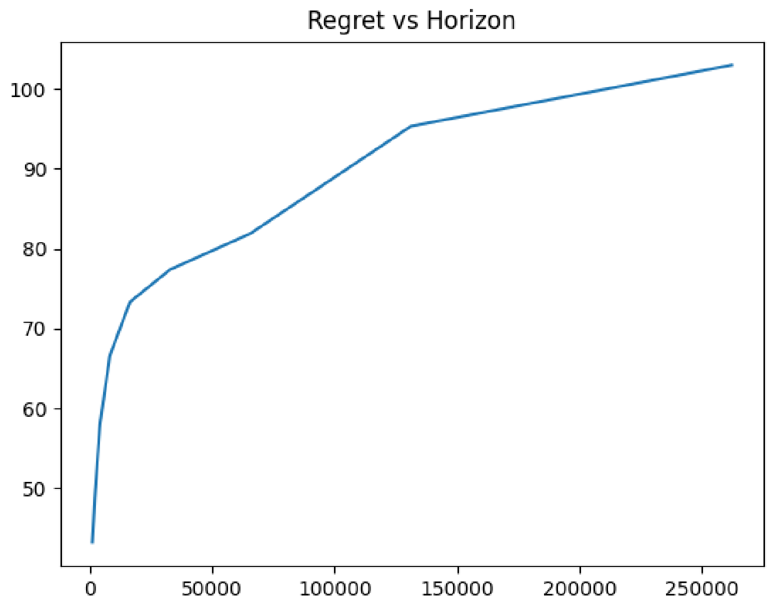


Figure 1.3: Regret curve for the Thompson Sampling algorithm over different horizons.

2 Task 2

2.1 PART A

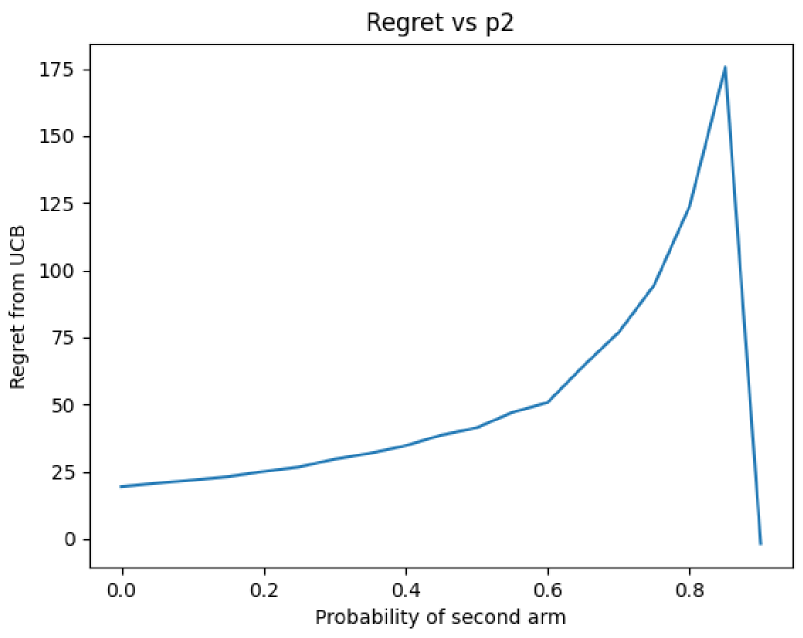


Figure 2.1: Task 2 Part A

The observed behaviour of the regret curve for the UCB (Upper Confidence Bound) algorithm may appear counterintuitive at first glance. Traditionally, a higher average mean of arms should result in lower regret in bandit instances. However, there's another factor at play here: exploration.
In the context of UCB, the algorithm balances the exploration of potentially better arms with the exploitation of arms that have shown promising rewards so far. When the means of two arms increase, UCB may sometimes struggle to identify the best arm quickly. As a result, it might occasionally choose the lower mean arm, causing the regret to increase.
The shape of the regret curve reflects a tradeoff between two contributing factors: the higher average mean of arms, which should ideally reduce regret, and the closer means of arms, which makes it challenging for the UCB algorithm to distinguish the best arm.
At $p_2 = 0.9$, the regret reaches 0 because both arms are equally good, and pulling either arm contributes to a regret of 0.
Conversely, the regret is positive at $p_2 = 0.0$. The UCB algorithm is bound to explore the unexplored arm due to the exploration term. This exploration causes some pulls to be made from arm p_2. The regret increases as the exploration term becomes more important, leading to higher regret. The regret curve eventually dips to 0 when both arms become identical, and no decision can be made.
The observed results are a product of the interplay between arm means, exploration, and exploitation in the UCB algorithm.

2.2 PART B

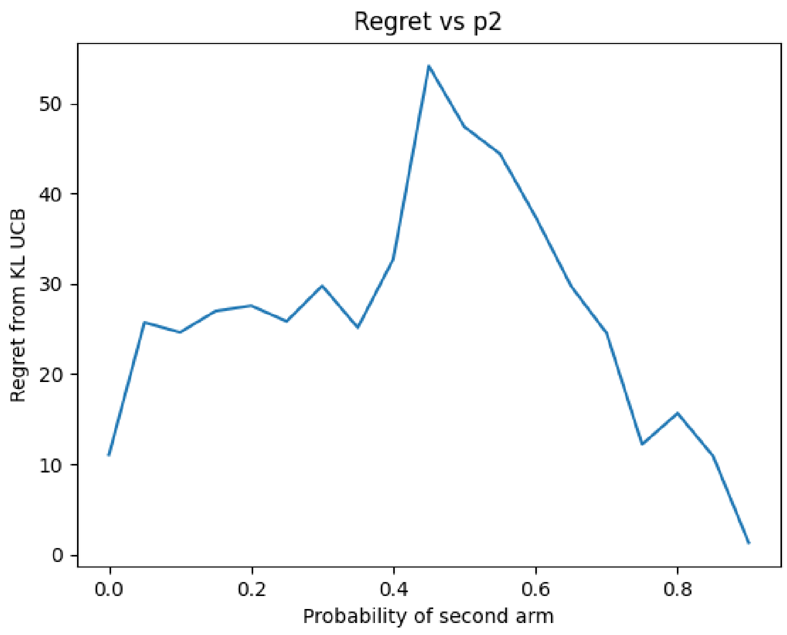
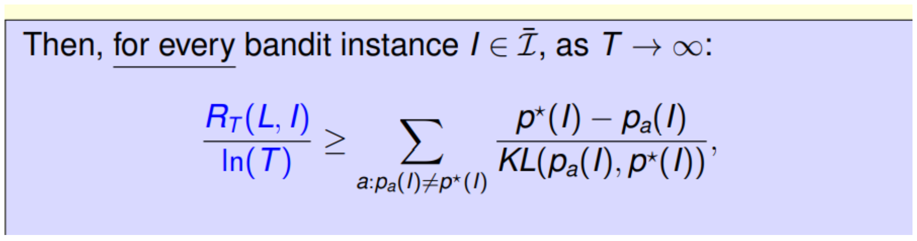


Figure 2.2: Task 2 Part B - KL UCB



Then, for every bandit instance $I \in \bar{\mathcal{I}}$, as $T \to \infty$:

$$\frac{R_T(L, I)}{\ln(T)} \geq \sum_{a: p_a(I) \neq p^\star(I)} \frac{p^\star(I) - p_a(I)}{KL(p_a(I), p^\star(I))},$$

Figure 2.3: KL UCB Lower Bound Analysis

The explanation for the observed behaviour in the regret curve for KL_UCB is derived from the lower bound analysis, as illustrated in the figure above.
On the left-hand side (LHS) of the lower bound equation, we notice only one term relevant to your specific bandit instance. In this term, the numerator is fixed at 0.1, representing the known separation between the means of the arms. Conversely, the denominator can be minimized to maximize the lower bound on the regret.
Both logarithmic terms must receive an input value close to 1 to minimise the denominator. This condition results in the total value approaching 0 and maximizes the lower bound on the regret.
Through preliminary analysis, considering the constraint $p_2 - p_1 = 0.1$, it becomes apparent that when y is close to 0.45 (making x essentially 0.55), we obtain the highest lower bound. This is because both logarithmic terms in the denominator are very close to zero, aligning with the condition for minimizing the denominator.

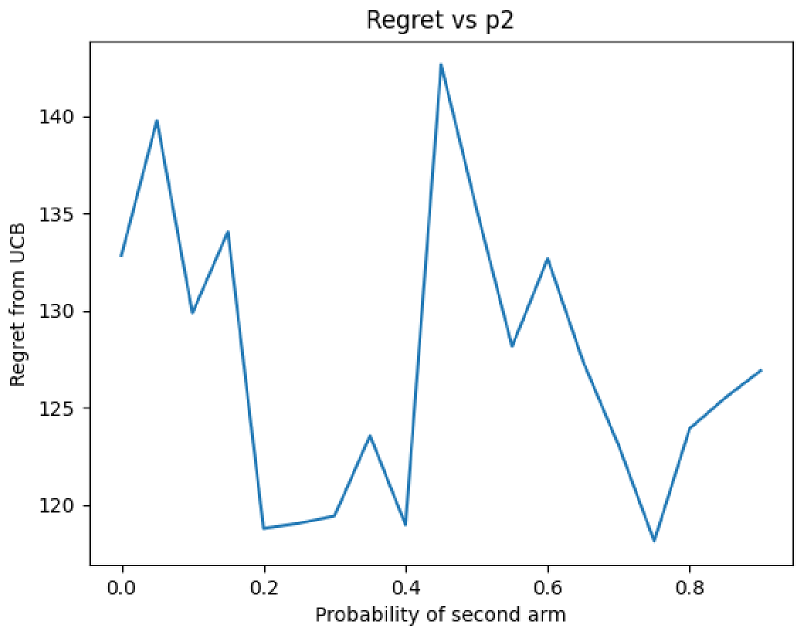


Figure 2.4: Task 2 Part B - UCB

The regret plot for UCB starts from a high value because the ucb parameter heavily depends on the second term (exploration term) because of the low estimated means. The higher the dependence on the exploration term, the longer it takes for the algorithm to choose one arm specifically on average, leading to higher regrets. The regret reduces as we move forward since the dependence on the first term (exploitation term) increases. At probability p_2 0.45, each arm is very symmetric in terms of success and failures (uniformly selecting 0 and 1); the exploration term again suddenly becomes more prominent, leading to a spike in regret using the same logic we used before, and again like before, we see a decrease in regret from this point. Another way to see the spike at 0.45 comes from our justification for KL_UCB.
Comparison - the KL_UCB bound has a much tighter confidence than that of UCB, and hence, we see a maximum regret in KL_UCB, which is much less than that in UCB.

3 Task 3: Dealing with Faulty Bandit Instances using Thompson Sampling

Task 3 presents a challenging bandit instance where the pulls are no longer guaranteed success. Each arm has a known probability of providing the correct output, and if a fault occurs, it returns a binary outcome randomly (0 or 1) with equal probability. The objective is to devise an effective algorithm to maximize the reward in this faulty bandit setting.

3.1 The Task

In this task, the core challenge lies in handling uncertainty due to possible faults during arm pulls. Specifically, you are provided with the following information: - Each arm has an associated probability of providing the correct output (known as the "fault-free" probability). - The environment may introduce faults with a certain probability during arm pulls.
To handle this uncertain environment, the task involves implementing an algorithm within the `task3.py` file. The algorithm must decide which arms to pull to maximize the cumulative reward while accounting for the potential for faults.

3.2 Approach using Thompson Sampling

One effective approach to tackle the faulty bandit setting is to employ Thompson Sampling, a probabilistic algorithm that maintains a probability distribution over the true reward of each arm. Here's how Thompson Sampling can be adapted for this task:

1. **Initialization**: Initialize a beta distribution for each arm, where the parameters represent the number of successes and failures.
2. **Sampling**: In each round, a sample from each arm's beta distribution will be used to obtain a probability estimate of the arm providing the highest rewards on average.
3. **Decision**: Select the arm with the highest sampled probability to pull in the current round.
4. **Observation**: When the reward is observed, update the corresponding arm's beta distribution parameters based on whether the outcome was a success or failure.

3.3 Why Thompson Sampling is Effective in the Faulty Bandit Setting

The faulty bandit setting can be seen as a transformed single-bandit setting, where each bandit (arm) is associated with a mean that combines several known and constant factors. Specifically:

1. **Fault Probability**: There exists a known fault probability, denoted as *fault*.
2. **Constant Factor**: A constant value of 0.5 is known from the outset.
3. **Estimation Parameter**: The algorithm's primary task is to estimate the parameter $\hat{p_i}$ for each arm, representing the estimated mean of that arm.

In this modified setting, the mean of each arm is determined as a weighted combination of these factors. Consequently, the algorithm's objective becomes clear: it must effectively estimate the value of $\hat{p_i}$ for each arm. This estimated value holds the key to maximizing rewards in the new setting where the bandit will be tested.
Thompson Sampling is an ideal choice for this task due to its innate ability to handle uncertainty and adapt to unknown parameters. By maintaining probability distributions over the rewards of each arm and updating these distributions based on observed outcomes, Thompson Sampling adeptly navigates the complexities of the faulty bandit setting. Ultimately, it works towards estimating the crucial parameter $\hat{p_i}$, enabling it to make well-informed decisions and maximize cumulative rewards within the modified bandit scenario.
In summary, Thompson Sampling excels in the faulty bandit setting by effectively estimating the critical parameter $\hat{p_i}$ for each arm, allowing it to make informed decisions and optimize rewards in the transformed bandit environment.
Thompson Sampling inherently accounts for uncertainty by sampling from the posterior distributions. This allows the algorithm to adapt to the faulty bandit setting and make informed decisions based on the observed outcomes.

3.4 Conclusion

Task 3 challenges us to develop a robust algorithm to maximize rewards in a bandit instance with uncertain outcomes. By leveraging the principles of Thompson Sampling and maintaining probability distributions over arm rewards, we can effectively address the uncertainties introduced by faults and make decisions that optimize cumulative rewards.

4 Task 4: Handling Multi-Multi-Armed Bandit Instances

Task 4 presents a unique challenge in dealing with two bandit instances simultaneously. In this task, when you specify an arm index to pull, one of the two given bandit instances is chosen uniformly at random. Subsequently, the arm corresponding to your provided index is pulled from the selected instance, and the environment returns the reward obtained along with which bandit instance was chosen for that pull. The goal is to devise an effective algorithm to maximize the reward in this multi-multi-armed bandit setting.

4.1 Task Characteristics

The key characteristics of Task 4 are as follows: - Two bandit instances exist, each having an equal number of arms. - The choice of which bandit instance to interact with is random, following a uniform distribution. - The algorithm determines which arm to pull from the chosen instance to maximize cumulative rewards.

4.2 Algorithm Approach

Addressing the challenge of two layers of randomization in this task requires a thoughtful approach. Firstly, we encounter randomness in selecting the bandit instance, and secondly, randomness arises when receiving rewards. We can model the problem as a single bandit instance to tackle this complexity.
In this modelled scenario, each bandit's arm now has a mean calculated as the weighted average of $\hat{p_i^1}$ and $\hat{p_i^2}$, where $\hat{p_i^1}$ represents the mean of the i^{th} arm from the first bandit instance, and $\hat{p_i^2}$ stands for the mean of the i^{th} arm from the second bandit instance. The weight for each component is set at 0.5, reflecting the knowledge that instances are selected uniformly, hence a 0.5 weight for each.
It's important to note that we do not know the means of either bandit instance in this task. This distinguishes it from the previous task, where the faulty bandit instance had a known uniform distribution. Effectively, this task can be seen as a more encompassing challenge, with fault $= 0.5$ and $\hat{p_i^2} = 0.5$, building upon the insights gained from the previous task.
Given these considerations, the Thompson Sampling algorithm continues to be our chosen approach to solve this multifaceted challenge effectively.

1. **Initialization**: Initialize a beta distribution for each arm, where the parameters represent the average number of successes and failures between the two instances.

2. **Sampling**: In each round, a sample from each arm's beta distribution will be used to obtain a probability estimate of the arm providing the highest rewards on average.

3. **Decision**: Select the arm with the highest sampled probability to pull in the current round.

4. **Observation**: When the reward is observed, update the corresponding instance's arm's success/failure, updating the beta distribution parameters based on the outcome.

4.3 Adapting to Complexity

Task 4 introduces a multi-multi-armed bandit scenario, where the algorithm must navigate layers of randomness to maximize rewards. By modelling the problem appropriately and applying the Thompson Sampling algorithm, we adapt to the complexity of the task and strive to optimize cumulative rewards in this dynamic environment.