CS 747

Programming Assignment - 1

Harshvardhan Shrivastav 23B1535

Task 1

UCB

Listing 1: UCB Algorithm Implementation in Python

```
class 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.counts = np.zeros(num_arms)
        self.rewards = np.zeros(num_arms)
        self.total_counts = 0
        self.c = 1.5 # higher than 2 works better for short term
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        for arm in range(self.num_arms):
            if self.counts[arm] == 0:
                return arm

        ucb_values = np.zeros(self.num_arms)
        avg_reward = self.rewards/self.counts
        conf = np.sqrt((self.c * np.log(self.total_counts)) / self.
           counts)
        ucb_values = avg_reward + conf
        return np.argmax(ucb_values)
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        self.total_counts += 1
        self.counts[arm_index] += 1
        self.rewards[arm_index] += reward
        # END EDITING HERE
```

- **Initialization (`init` function):** Defines all relevant variables:
  - `counts`: tracks how many times each arm is pulled,

  - `success`: records the number of successes for each arm,
  - `total_counts`: keeps track of the total number of time steps,
  - `c`: a tunable parameter controlling the width of the confidence bound.

- **Action Selection (`give_pull` function):** Initially draws a sample from each arm once, then:
  - Computes the empirical mean $\hat{\mu}_i$ for each arm i,
  - Calculates the confidence interval and the corresponding UCB value:

$$\text{UCB}_i = \hat{\mu}_i + c\sqrt{\frac{\ln t}{n_i}}$$

    where n_i is the number of times arm i has been pulled and t is the current time step,
  - Selects and pulls the arm with the maximum UCB value.

- **Reward Update (`get_reward` function):** Updates the state variables defined in `init` (such as `counts`, `success`, and `total_counts`) to prepare for the next time step.

KL-UCB

Listing 2: KL-UCB Implementation in Python

```
# You can use this space to define any helper functions that you
    need
def kl_divergence(p, q):
    if p == 0:
        return -math.log(1 - q) if q < 1 else float('inf')
    if p == 1:
        return -math.log(q) if q > 0 else float('inf')
    else:
        if q == 0 or q == 1:
            return float('inf')
        return p * math.log(p / q) + (1 - p) * math.log((1 - p) / (1
            - q))

def find_kl_ucb(p, val, upper_bound=1, lower_bound=0, precision=1e
    -6):
    if p >= 1:
        return 1.0
    if val <= 0:
        return p
    while upper_bound - lower_bound > precision:
        mid = (upper_bound + lower_bound) / 2
        if kl_divergence(p, mid) > val:
            upper_bound = mid
        else:
```

```
            lower_bound = mid
    return (upper_bound + lower_bound) / 2

class KL_UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        # You can add any other variables you need here
        self.counts = np.zeros(num_arms)
        self.rewards = np.zeros(num_arms)
        self.total_counts = 0
        self.c = 0  # exploration parameter

    def give_pull(self):
        for arm in range(self.num_arms):
            if self.counts[arm] == 0:
                return arm

        kl_ucb_values = np.zeros(self.num_arms)
        avg_reward = self.rewards/self.counts

        for arm in range(self.num_arms):
            val = (math.log(self.total_counts)
                   + self.c * math.log(math.log(max(self.
                      total_counts, math.e))))
            val /= self.counts[arm]
            kl_ucb_values[arm] = find_kl_ucb(
                avg_reward[arm], val,
                upper_bound=1,
                lower_bound=avg_reward[arm],
                precision=1e-6
            )
        return np.argmax(kl_ucb_values)

    def get_reward(self, arm_index, reward):
        self.total_counts += 1
        self.counts[arm_index] += 1
        self.rewards[arm_index] += reward
```

- **Initialization (`__init__`):** This function defines the relevant variables:
  - `counts` keeps track of how many times each arm has been pulled.
  - `rewards` (or `success`) accumulates the total reward (or number of successes) for each arm.
  - `total_counts` stores the total number of time steps so far.
  - `c` is a tunable parameter in the RHS of the KL UCB equation that we solve.
- **Action selection (`give_pull`):**

- Initially, the algorithm ensures each arm is sampled at least once.
- For subsequent pulls, it computes the empirical mean reward

$$\hat{\mu}_a = \frac{\texttt{rewards}[a]}{\texttt{counts}[a]}$$

for each arm a.

- It then solves for the upper confidence bound q such that the KL divergence satisfies

$$D_{\mathrm{KL}}\big(\hat{\mu}_a \,\|\, q\big) \leq \frac{\log t + c \log\log t}{\texttt{counts}[a]}, \qquad t = \texttt{total\_counts},$$

where $D_{\mathrm{KL}}(p\|q) = p\log\frac{p}{q} + (1-p)\log\frac{1-p}{1-q}$.

- The arm with the largest such upper bound q is chosen:

$$a^\star = \arg\max_a q_a.$$

- **Reward update (`get_reward`):** After observing a reward from the chosen arm, the algorithm updates:
  - $\texttt{total\_counts} \leftarrow \texttt{total\_counts} + 1$,
  - $\texttt{counts}[a] \leftarrow \texttt{counts}[a] + 1$,
  - $\texttt{rewards}[a] \leftarrow \texttt{rewards}[a] + r$.

Thompson Sampling

Listing 3: KL-UCB Implementation in 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.counts = np.zeros(num_arms)
        self.success = np.zeros(num_arms)
        self.total_counts = 0
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        samples = np.random.beta(self.success + 1, self.counts -
           self.success + 1)
        return np.argmax(samples)
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        self.total_counts += 1
```

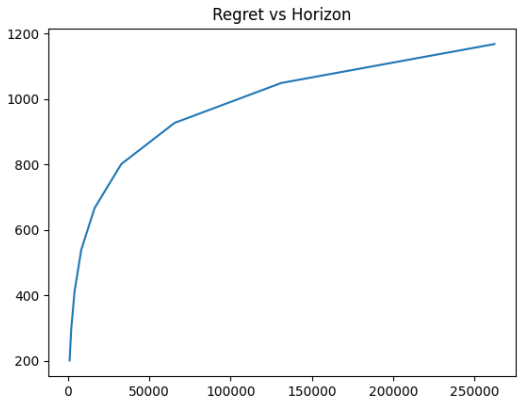
```
        self.counts[arm_index] += 1
        self.success[arm_index] += reward
        # END EDITING HERE
```

- **Initialization (`init` function):** Defines the state variables:
  - `counts`: number of times each arm has been pulled,
  - `success`: number of successes for each arm,
  - `total_counts`: total number of time steps.
- **Action Selection (`give_pull` function):** For each arm i,
  - sample from its posterior belief distribution

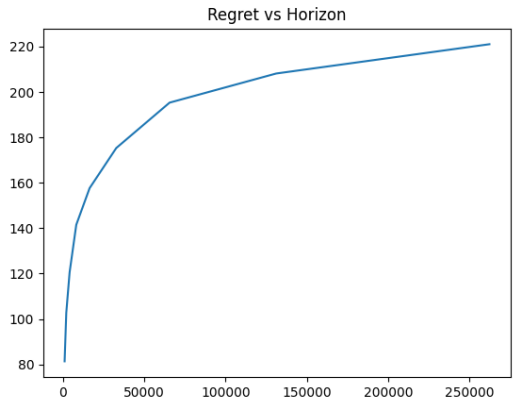
$$\theta_i \sim \text{Beta}(s_i + 1,\ f_i + 1),$$

  - select and return the arm with the maximum sampled value $\arg\max_i \theta_i$.
- **Reward Update (`get_reward` function):** Updates the state variables defined in `init` (such as `counts` and `success`) for the next time step.

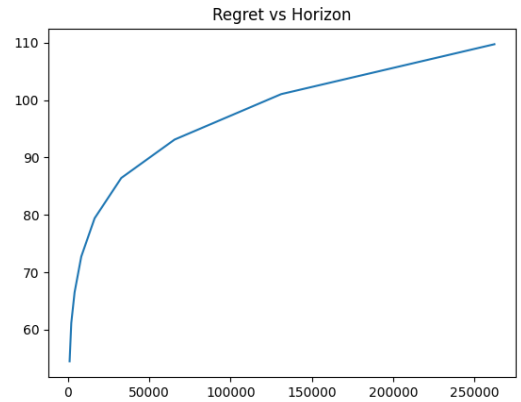
(a) Epsilon Greedy Regret



(b) UCB Regret



(c) KL UCB Regret



(d) Thompson Sampling Regret

Figure 1: Comparison of different bandit algorithms regrets

Task 2

The task at hand is a formulation of the Best Arm Identification (BAI) problem. For the task at hand, we know that the doors health damage is modeled as a Poisson Distribution. For this task, I decided to use the KL UCB algorithm but changed for this particular task.

I use the KL divergence for a Poisson Distribution:

$$KL(p||q) = q - p + p \cdot \log(p/q)$$

My algorithm: **Steps of the Modified KL-UCB with Health Penalty Algorithm:**

1. **Initialization:** For each arm $k = 1, \ldots, K$ set counts$[k] = 0$, rewards$[k] = 0$, health$[k] = 100$, and total_counts $= 0$. Fix exploration parameter c.
2. **At each round t:**
   (a) If there exists an arm with counts$[k] = 0$, select that arm.
   (b) Otherwise, compute the empirical mean reward $\hat{\mu}_k = \text{rewards}[k]/\text{counts}[k]$ for all k.
   (c) Compute the exploration term
   $$v_k = \frac{\log(\text{total\_counts}) + c \log\log(\max(\text{total\_counts}, e))}{\text{counts}[k]}.$$
   (d) For each arm, find the KL-UCB index u_k by solving for the smallest $q \geq \hat{\mu}_k$ such that $\text{KL}(\hat{\mu}_k \| q) > v_k$ using binary search.
   (e) Select the arm
   $$k^\star = \arg\min_k \big(\text{health}[k] - u_k\big).$$
3. **Update after pulling arm $k^\star$:**
   $$\begin{aligned} &\text{total\_counts} \leftarrow \text{total\_counts} + 1, \\ &\text{counts}[k^\star] \leftarrow \text{counts}[k^\star] + 1, \\ &\text{rewards}[k^\star] \leftarrow \text{rewards}[k^\star] + r, \\ &\text{health}[k^\star] \leftarrow \text{health}[k^\star] - r. \end{aligned}$$

Task 2: Poisson Door–Breaking Problem

Problem recap

We are given up to $K \leq 30$ doors. Door k starts with strength $S_k(0) = 100$. Each strike on door k yields damage $r_{k,t} \sim \text{Pois}(\lambda_k)$, with $\lambda_k \in (0, 3)$ unknown. The new strength is $S_k \leftarrow S_k - r_{k,t}$. The episode terminates immediately once any $S_k < 0$.

Our goal is to design a selection policy π_t choosing at each step which door to strike so as to **minimise the expected total number of strikes until termination**.

Modelling intuition

If we knew all λ_k exactly, the optimal strategy would be to always strike the door with largest λ_k (highest expected damage per strike). But initially we do not know λ_k, and λ_k are close to each other (0–3). Thus we need an *explore–exploit* mechanism that:

1. quickly learns which doors have higher λ_k;
2. concentrates strikes on the most promising door;
3. also takes into account each door's current strength S_k: once a door is close to breaking, additional strikes there have higher chance of terminating the game quickly.

Algorithm: KL-UCB with Health-Aware Index

I adapt KL-UCB (which gives tight upper confidence bounds for exponential-family means) to the Poisson setting and add a term that reflects the remaining door strength. This yields an index that is high for (i) doors with large estimated mean damage and (ii) doors whose remaining strength is low.

Initialization. For each door k:

$$\text{counts}[k] = 0, \quad \text{damage\_sum}[k] = 0, \quad H_k = 100.$$

Set total_counts= 0 and choose an constant c (we used $c = 3$) for the RHS of KLUCB eqn.

At each step t:

1. If any door has $\text{counts}[k] = 0$, strike it once to get an initial sample.
2. Otherwise compute empirical mean $\hat{\mu}_k = \text{damage\_sum}[k]/\text{counts}[k]$.
3. Compute the exploration term

$$v_k = \frac{\log(\text{total\_counts}) + c \log\log(\max(\text{total\_counts}, e))}{\text{counts}[k]}.$$

4. Compute the Poisson KL-UCB index u_k = smallest $q \geq \hat{\mu}_k$ such that $\text{KL}(\hat{\mu}_k \| q) > v_k$, where $\text{KL}(p\|q) = q - p + p \log \frac{p}{q}$. Binary search yields u_k efficiently.
5. Form the **health-aware index**

$$I_k = \frac{H_k}{u_k} \quad \text{or} \quad I_k = H_k, -u_k$$

depending on whether you want a ratio or difference (I used the difference form).

6. Select door $k^\star = \arg\min_k I_k$ (largest index).

Update after strike. Strike door $k^\star$, observe damage r, and update:

$$\begin{aligned}
\text{total_counts} &\leftarrow \text{total_counts} + 1,\\
\text{counts}[k^\star] &\leftarrow \text{counts}[k^\star] + 1,\\
\text{damage_sum}[k^\star] &\leftarrow \text{damage_sum}[k^\star] + r,\\
S_{k^\star} &\leftarrow S_{k^\star} - r.
\end{aligned}$$

If any $S_k < 0$ stop; the door is broken.

Why this method is appropriate

Distribution-specific confidence bounds. KL-UCB for Poisson uses the exact form of the Kullback–Leibler divergence for the exponential family, yielding much tighter bounds than generic Hoeffding inequalities. This accelerates learning of the true λ_k.

Health-aware prioritisation. A pure KL-UCB policy would eventually concentrate on the highest-λ_k door but may still waste strikes on doors whose λ_k is only marginally smaller but whose health is almost gone. By multiplying or subtracting by S_k we bias the index toward doors that are both promising and near-breaking, directly aligning with the termination objective. I preferred the subtracting approach as that was the one which gave better results when tested against the auto-graders test cases. The dividing approach with health/kl-ucb gives more intuitive sense as to which door has the fewest timesteps remaining till it breaks, but in practice that algorithms performance wasn't as good as the one where i subtract the ucb from health.

Expected performance. As t grows, the KL indices converge to the true means. Thus the policy converges to striking the door with largest λ_k, which is optimal if you know the means. In the finite-sample regime the added health term accelerates termination by exploiting doors that are already weakened, reducing the expected number of strikes.

Conclusion

The KL-UCB-with-health index leverages distribution-specific concentration bounds for efficient exploration and incorporates door strength into its decision choosing, aligning the index with the termination criterion.

Task 3

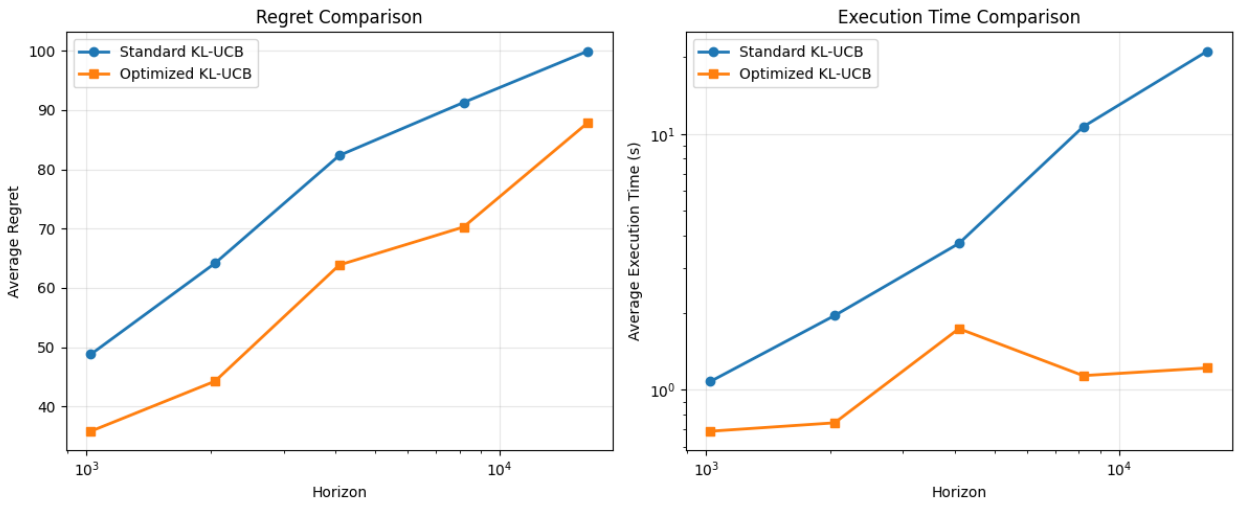


Figure 2: Regret and Computation Time Comparison

To optimize the computation of KL UCB, hint provided was: In a typical run of KL-UCB, there will be long sequences in which the same arm gets pulled, before switching to another one. On a given decision-making step, can you decide to give an arm many pulls (say m ¿= 2) without consulting the data and re-calculating UCBs while performing these m pulls?

So from this hint and exploring some academic literature, I utilised a form of batched algorithm, which instead of recomputing the kl-ucb at each time step. Performs the recalculation after every m steps.

The algorithm I have used uses a static batch having batch sizes fixed depending on the length of the horizon. So the horizon is divided into batches of size floor(horizon/1000).

Optimisation Approach and Performance Analysis.

In the standard KL–UCB algorithm, at each time step $t = 1, \ldots, T$ the index

$$U_k(t) = \max\{q \geq \hat{\mu}_k(t) : \mathrm{KL}(\hat{\mu}_k(t) \| q) \leq v_k(t)\}, \quad v_k(t) = \frac{\log t + c \log\log(\max\{t, e\})}{n_k(t)}$$

is recomputed for every arm $k = 1, \ldots, K$, where $\hat{\mu}_k(t)$ is the empirical mean reward and $n_k(t)$ the number of pulls of arm k up to time t. This costs $O(K)$ binary searches per round, so the total cost is $O(KT)$.

Batched computation. Our implementation computes all $U_k(t)$ only once every B rounds, where $B = \lfloor T/1000 \rfloor$ in our code. Between two recomputations we keep pulling the arm

$$k^\star(t) = \arg\max_k U_k(t)$$

from the last index update. This changes the complexity from

$$O(KT) \quad \text{to} \quad O\Big(\frac{KT}{B} + T\Big),$$

because index recomputation happens only T/B times. For example with $B = T/1000$ the number of binary searches drops by about a factor of 1000.

Why batching preserves regret. For KL–UCB the index $U_k(t)$ varies slowly with t because both $n_k(t)$ and $v_k(t)$ change by $O(1/t)$ per round. Thus for a small batch length $B \ll T$ the "stale" indices remain close to the true $U_k(t)$ during the batch. As long as $B = o(T)$ (the batch size does not grow faster than T) the asymptotic regret bound of KL–UCB,

$$\mathbb{E}[R_T] = O\left(\sum_{k \neq k^\star} \frac{\log T}{\Delta_k}\right),$$

is unaffected up to lower-order terms.

Theoretical Question

KL UCB solves for the upper confidence bound q such that the KL divergence satisfies

$$D_{\mathrm{KL}}\big(\hat{\mu}_a \,\|\, q\big) \leq \frac{\log t + c \,\log\log t}{\texttt{counts}[a]}, \qquad t = \texttt{total\_counts},$$

where $D_{\mathrm{KL}}(p\|q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$. Now with the counts[a] and $\hat{\mu}_a$ being constant the value of q for a particular t is monotonic and non decreasing in nature (because log(t) + clog(log(t)) grows with t). So we can store the value of q at iteration for unique pair of counts[a] and $\hat{\mu}_a$, and reuse it in the binary search if we find the same counts[a] and $\hat{\mu}_a$ pair at some time t ahead, as the lower bound as we know the function is non decreasing. So instead of limiting the binary search range from $[\hat{p}, 1]$ we limit the range to $[q_0, 1]$ where q_0 is the value of q for a pair of counts[a] and $\hat{\mu}_a$ at some time t.

The only change we are making is in decreasing the range of the binary search bounds where we have to search. So this is still synonymous to normal KL UCB just little optimized on the binary search part. Which would provide speedup in the computation while maintaining the same regret structure.