CS 747 - Assignment 1

Aditya Sriram (200070004)

September 2022

1 Introduction

This assignment tests our understanding of regret minimisation algorithms discussed in the class. There are three tasks. The first task asks us to implement **UCB**, **KL-UCB** and **Thompson Sampling**. In the second task, we are supposed to come up with an algorithm for **Batched Sampling**, where feedback is not provided after each draw but in aggregate, after a certain number of pulls. In the final task, we need to implement an algorithm for the multi-armed bandits problem when the horizon is equal to the number of arms, given that the arm means are distributed uniformly between 0 and 1.

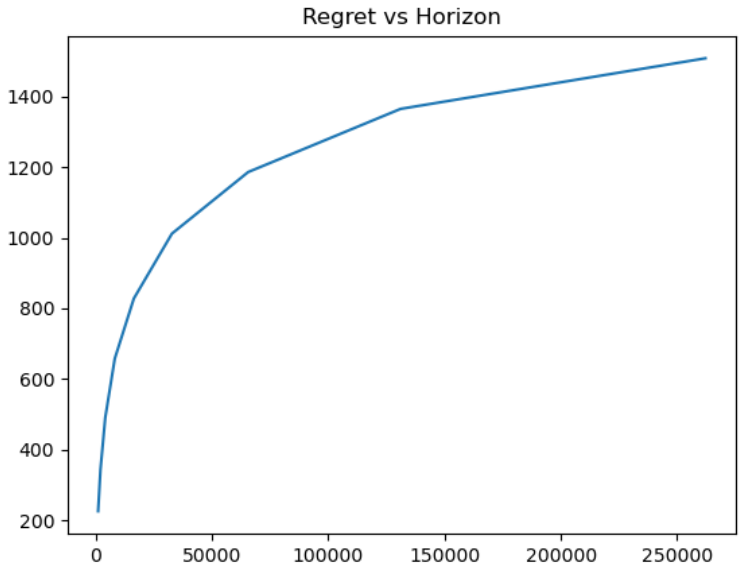
2 Task 1

2.1 UCB

```
def give_pull(self):
    if self.t < self.num_arms:
        # First pull every arm once so that quantities are well-defined
        return self.t
    else:
        # Select arm that maximises UCB with random tie-breaking
        return np.argmax(self.ucbs)
        # return np.random.choice(np.flatnonzero(self.ucbs == np.max(self.ucbs)))

def get_reward(self, arm_index, reward):
        self.counts[arm_index] += 1
        n = self.counts[arm_index]
        value = self.values[arm_index]
        new_value = ((n - 1) / n) * value + (1 / n) * reward
        self.values[arm_index] = new_value

        # Only update UCB after all arms have been sampled exactly once
        if self.t >= self.num_arms:
            self.ucbs = self.values + np.sqrt(2 * math.log(self.t) / self.counts)
```

```
            self.ucbs[arm_index] += (new_value - self.values[arm_index])

        self.t += 1 # Increment global time since start of the algorithm
```

The arm which has the maximum UCB is pulled in every time step. However, to start off the algorithm, I have used **round-robin** sampling so that $\log t$ is well defined. On receiving the reward from the environment, the empirical mean of the pulled arm is updated. Upper confidence bounds of all arms are updated once they have been sampled once.



2.2 KL-UCB

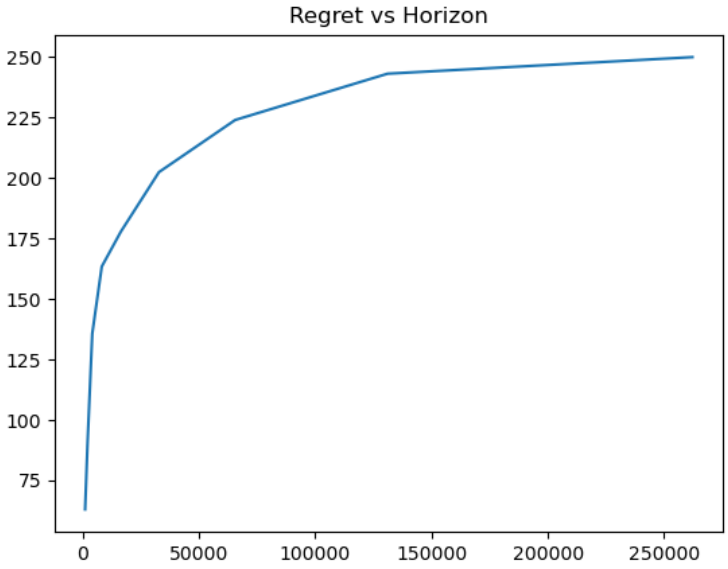
```
def kl(p, q):
    tol = 1e-9
    kl_div = p * np.log((p / q) + tol) + \
            (1 - p) * np.log((1 - p) / (1 - q) + tol)
    return kl_div

def compute_kl_ucb(counts, values, t, c=3, N=5, tol=1e-3):
    iter = 0

    lo = np.copy(values)
    hi = np.ones(np.shape(values)) - 1e-9

    f = np.ones(np.shape(values))
```

```
    kl_ucbs = np.zeros(np.shape(values))
    while (iter < N and np.abs(f).any() > tol):
        kl_ucbs = (lo + hi) / 2
        f = counts * kl(values, kl_ucbs) - \
                math.log(t) - c * math.log(math.log(t))
        above = np.where(f > 0)
        below = np.where(f <= 0)
        lo[below] = kl_ucbs[below]
        hi[above] = kl_ucbs[above]

        iter += 1

    return kl_ucbs

def give_pull(self):
    if self.t < self.num_arms:
        return self.t
    else:
        return np.argmax(self.kl_ucbs)

def get_reward(self, arm_index, reward):
    self.counts[arm_index] += 1
    n = self.counts[arm_index]
    value = self.values[arm_index]
    new_value = ((n - 1) / n) * value + (1 / n) * reward
    self.values[arm_index] = new_value

    if self.t >= self.num_arms:
        self.kl_ucbs = compute_kl_ucb(self.counts, self.values, self.t)

    self.t += 1
```

The logic for pulling arms at each time step is identical to UCB. However, the upper confidence bound is obtained by solving for q in the following equation.

$$u_a^t \mathrm{KL}\left(p_a^t, q\right) = \ln t + c \ln \ln t$$

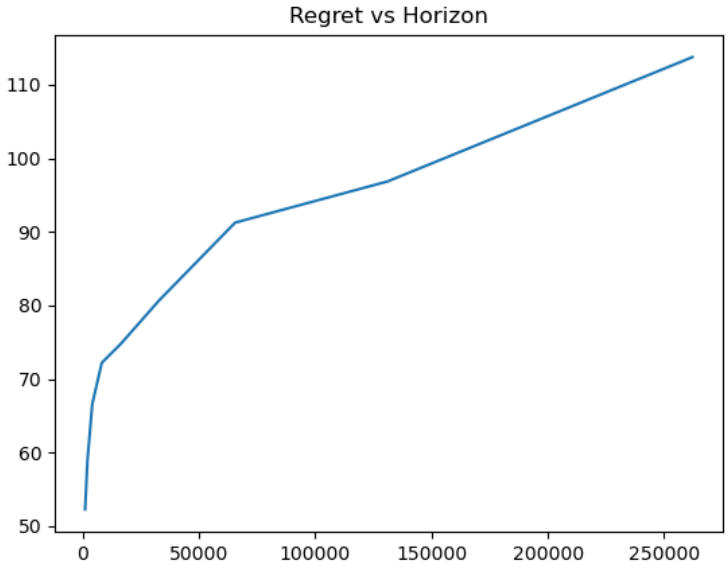
The solution has been found using the **Bisection Method** and I have experimented with different values of c. Although the plot for KL-UCB was generated using $c = 3$, we can obtain lower regrets by setting $c = 0$ (Garivier et. al). The algorithm was terminated after $N = 5$ iterations to speed up computation without compromising on accuracy.



2.3 Thompson Sampling

```
def give_pull(self):
    x = np.random.beta(self.successes + 1, self.failures + 1)
    return np.argmax(x)

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

We keep track of the number of successes (1-rewards) and failures (0-rewards) for each arm. These form the parameters of the **Beta Distribution** which represents our beliefs about the arm means at each time step. To pull an arm, a sample is drawn from the Beta distribution for each arm, and the arm returning the largest sample is pulled.



From the Regret vs Horizon plots, we observe that **Thompson Sampling** achieves the lowest regret, followed by **KL-UCB** and **UCB**. However, we also note that for Thompson Sampling, the regret has an increasing trend while the plots for KL-UCB and UCB have started to plateau. Out of all the three methods, **KL-UCB** took the longest to run because the method requires you to solve an equation.

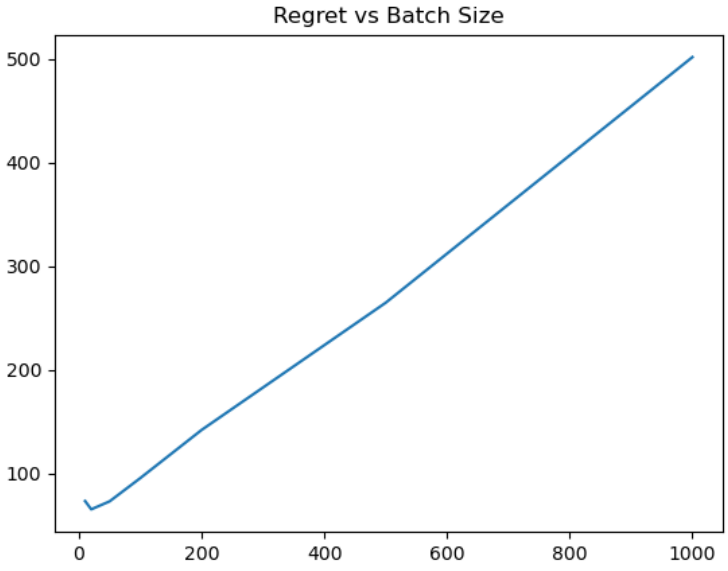
I have also experimented with different tie-breaking strategies. In particular, we can break ties randomly or choose one of the optimal pulls deterministically. This did not have a huge impact on the performance of the algorithms and the regret achieved was similar in both cases.

3 Task 2

```
def give_pull(self):
    pulls = np.zeros(self.num_arms, dtype=np.int64)
    for _ in range(self.batch_size):
        x = np.random.beta(self.successes + 1, self.failures + 1)
        pulls[np.argmax(x)] += 1

    arms = np.flatnonzero(pulls)
    return arms, pulls[arms]

def get_reward(self, arm_rewards):
    for arm in arm_rewards:
```

```
        self.successes[arm] += np.sum(arm_rewards[arm])
        self.failures[arm] += len(arm_rewards[arm]) - \
                    np.sum(arm_rewards[arm])
```

The implemented algorithm uses **Thompson Sampling** but rather than sampling the belief distribution once, we draw `batch_size` samples and choose the arm that returns the highest sample for each draw. On receiving aggregated feedback after multiple draws, we update the number of successes and failures for arms that were pulled.



Except for a small kink in the beginning, the regret is seen to increase linearly with batch size. This is because `batch_size` controls the frequency at which feedback is received. If the batch is small, you receive feedback more frequently and hence, update your beliefs more frequently.

4 Task 3

```
class AlgorithmManyArms:
    def __init__(self, num_arms, horizon):
        self.k = int(np.sqrt(num_arms))

        self.num_arms = num_arms
        arms = np.arange(0, num_arms)
        np.random.shuffle(arms)
```

```
        self.arms = arms[0:self.k]

        self.mapping = dict()
        for index in range(self.num_arms):
            self.mapping[arms[index]] = index

        self.successes = np.zeros(self.k)
        self.failures = np.zeros(self.k)
        # Horizon is same as number of arms

    def give_pull(self):
        x = np.random.beta(self.successes + 1, self.failures + 1)
        return self.arms[np.argmax(x)]

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

Since the horizon is equal to the number of arms, we cannot afford to choose each arm once. Since the arm means are uniformly distributed between 0 and 1, the mean of arm means of any subset will be equal to 0.5. The proposed algorithm randomly chooses a subset of all arms and applies **Thompson Sampling** to the arms in the subset. The size of the subset influences the regret and I found that a subset size $= \sqrt{T}$, where T is the number of arms, achieves the lowest regret.

The plot of regret vs horizon resembles the plot of $f(x) = \sqrt{x}$ and I have confirmed this independently using curve fitting (the plot can be found below). This is consistent with my choice of using $\sqrt{T}$ arms in subsampling.