CS 747 : Assignment 1
Regret Minimisation

Shivam Patel, 200070077

November 21, 2022

1 UCB, KLUCB, Thompson Sampling

1.1 Problem Statement

We try to implement standard regret minimisation algorithms and compare them with benchmark regrets.

1.2 Methods

The **UCB, KLUCB and Thompson sampling** are standard algorithms described in lectures. Thus, an in detail explanation is not required. All these algorithms provide sub linear regret and are binding on the Lai Robbins lower bound complexity.

1.3 Code Snippets

UCB Algorithm

```
class UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        self.ucb_un = np.zeros(num_arms)
        # keeps value of the second term in the ucb
        self.ucb_pa = np.zeros(num_arms)
        # keeps the first term, empirical estimate pa
```

```
        self.ucb_total = np.zeros(num_arms)
        # the sum of two terms, the upper confidence bound
        self.counts = np.ones(num_arms)
        # the number of times the arm has been samples,
        # kept 1 initially to avoid divide by zero
        self.total_samples = num_arms

    def give_pull(self):
        return np.argmax(self.ucb_total)
        # returns the arm_index with highest UCB

    def get_reward(self, arm_index, reward):
        self.total_samples += 1
        self.counts[arm_index] += 1
        n = self.counts[arm_index]
        self.ucb_pa[arm_index]=self.ucb_pa[arm_index]*(n-1)/n + reward/(n)
        # updating the empirical probability for the respective arm
        self.ucb_un=np.sqrt((2*math.log(self.total_samples))/(self.counts))
        # calculating the UCB term 2 (exploration cost term)
        self.ucb_total = self.ucb_un + self.ucb_pa
        # updating the UCB value
```

The above code is the complete class and method definition required for the UCB algorithm. Appropriate comments have been added wherever necessary.

KL - UCB Algorithm

```
class KL_UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        self.klucb_pa = np.zeros(num_arms)
        # keeps the empirical estimate pa
        self.klucb_q = np.ones(num_arms)/1000
        # q value of each arm is stored in this np-array
        self.kl_counts = np.ones(num_arms)
        # number of times each arm is sampled
        # the number of times the arm has been sampled,
        # kept 1 initially to avoid divide by zero
```

```
        self.time = 1 # represents total time, initially set to 1
        self.num = num_arms
        self.c = 3
        # self.c is a hyperparameter
        # END EDITING HERE
```

The above snippet is the initialising constructor for the KL_UCB class. All the required variables have been defined and duly explained with comments. Following is the code for binary search for finding the q value for each arm.

```
def give_pull(self):
    for j in range(self.num):
        # binary search for q for each of num_arms
        low = self.klucb_pa[j]
        high = 1
        ua = self.kl_counts[j]
        mid = (high+low)/2
        t = self.time
        p = self.klucb_pa[j]
        # binary search loop run for 10 iteration,
        #precision of 2^10 = 0.001
        for i in range(10):
            mid = (high+low)/2
            if kl(p,mid)-(math.log(t)+self.c*math.log(math.log(t)))/ua<=0:
                low = mid
                # the mid and low value are on same side of q
            else:
                high = mid
                # the high and mid value are on same side of q
        self.klucb_q[j] = mid
    self.time += 1
    ix = np.argmax(self.klucb_q)
    # returning the optimum arm_index
    self.kl_counts[ix] += 1
    return ix

def get_reward(self, arm_index, reward):
    self.kl_counts[arm_index] += 1
    n = self.kl_counts[arm_index]
```

```
        self.klucb_pa[arm_index] = self.klucb_pa[arm_index]*(n-1)/n + reward/n
        # updating reward
        return
```

Thompson Sampling Algorithm

The code for thompson sampling is relatively easy to understand and follow, and comments have been added at required places to aid the readers' understanding.

```
class Thompson_Sampling(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        self.a = np.zeros(num_arms)
        self.b = np.zeros(num_arms)
        # these variables keep track of arm pull successes and failures

    def give_pull(self):
        return np.argmax(np.random.beta(self.a+1, self.b+1))
        # returns the maximum sampled arm

    def get_reward(self, arm_index, reward):
        if reward==1:
            self.a[arm_index] +=1
        else:
            self.b[arm_index] +=1
        # updating successes and failures for each arm
```

1.4 Results

Here, I have included the regret plots for the three algorithms. Thompson sampling seems to have a linear growth after some time, as the algorithm tries to find the best arm initially, and exploit it later. This is tantamount to minimizing the slope of the linear part of the regret curve in Thompson Sampling. In almost all other cases, there is a logarithmic nature of the graph. I have obtained negative regret for later arm pulls in the KL-UCB algorithm. This can be due to implementational details overlooked in the experiment (negative regret can be due to any arm giving better than expected (p_a) positive reward ratio).

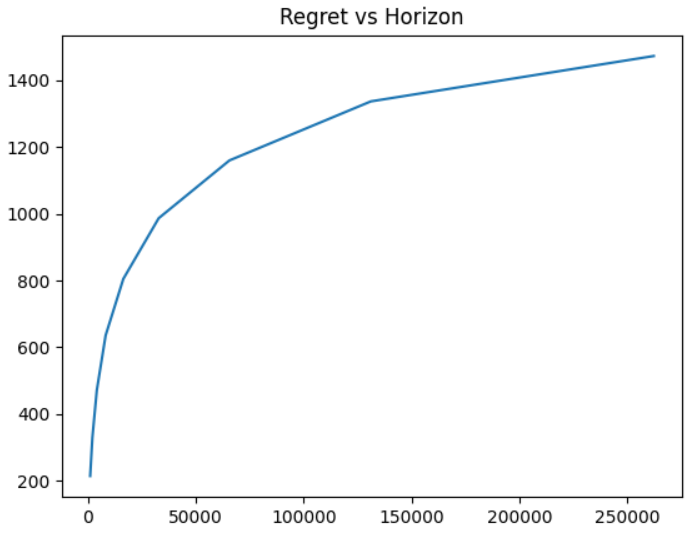

Figure 1: UCB sampling

Figure 2: KL UCB sampling

Figure 3: Thompson Sampling

2 Batched Sampling

2.1 Problem Statement

We try to implement **batch sampling regret minimisation on a generalised set of batch_size, num_arms and horizon parameters**.

2.2 Methods

Batched Sampling calls for a generalised algorithm which has no clearly separated phases of explore and exploit, or else the algorithm may fail in some edge cases like where horizon is not much larger than batch size. I have **implemented Thompson sampling** here. I am computing the beta distribution samples for each pull in the batch_size. The arm with maximum beta sample is pulled, and the process is repeated for batch_size number of pulls. The success and failure counts for arms are updated after an entire batch.
This algorithm is precisely known as **Thompson Subsampling**.

Figure 4: Task 2 Thompson Subsampling

2.3 Results

Here, I have included the regret plot for the algorithm. Thompson sampling seems to have a linear growth, as the algorithm tries to find the best arm initially, and exploit it later. We should not be confused by the linear nature of the graph, as the algorithm already is finding the 'optimal' arm in the kink near origin. This is equivalent to minimizing the slope of the linear part of the regret curve in Thompson Sampling.

3 Num_Arms comparable to Horizon

3.1 Problem Statement

This task is aimed at finding appropriate algorithm for regret minisation in the specific case where num_arms is comparable to horizon (equal to in this case).

3.2 Methods

This problem statement is unique in itself as any of the standard algorithms learnt in lectures as well as implemented in this assignment will fail miserably here. This is attributed to the fact that we have limited number of trials, and cannot waste them for finding the maximum probability coin. We need to come up with an approximately correct solution for the best coin, and exploit it greedily.
I experimented initially with a **thompson sampling algorithm, which randomly sampled $\sqrt{num\_arms}$ initially**. Later on, these initially sampled arms were **greedily sampled** through thompson sampling. The number of times each of the $\sqrt{num\_arms}$ were sampled at the beginning in round robin fashion was also experimented, where the **least amount of 1 sample each gave the best results**.

Shortcoming of the above algorithm -
This algorithm performed poorly on the test data. After serious thought, I came to conclusion that not enough number of samples of each of the $\sqrt{num\_arms}$ arms were taken to set up a good Beta distribution which could give high confidence intervals. (Code for the above algorithm is commented in the submission for task 3 in the additional function definition space)

Newer Approach
I once again sampled $\sqrt{num\_arms}$, and then used the ϵ Greedy 3 algorithm. The epsilon in my implementation was $\epsilon = \frac{1}{\sqrt{num\_arms}}$.

This choice of ϵ was based on intuition, and backed by the discussion in one of the lectures as to how to convert an ϵ Greedy algorithm to get logarithmic regrets (GLIE - Greedy in the Limit and Infinite Exploration).

This algorithm worked smoothly on the test data, giving acceptable regrets and desirable logarithmic resembling graphs.

3.3 Code Snippets

In the Init() method

```
self.num_explore = math.floor(1*math.sqrt(self.horizon))
self.eps = 1/self.num_explore
self.chosen_samples = np.random.choice(self.num_arms,
                        size=(self.num_explore), replace=False)
```

In the give_pull method

```
if self.t < 1*self.num_explore: # exploring stage
    maxm_ix = self.chosen_samples[self.t % self.num_explore]
    # the index returned is the tantamount to the round robin arm pulling
    #(hence modulo operator)
else:
    if np.random.random() < self.eps: # explore
        maxm_ix = self.chosen_samples[self.t % self.num_explore]
    else: # exploit
        maxm_ix = np.argmax(self.pa)
```

3.4 Results

The regret plot shows us that the ϵ Greedy 3 algorithm changes the choice of the best coin at various points during the simulation, implying that it is a good fit for such applications.

4 References and Acknowledgements

This assignment and code is completely mine, and I have taken no direct code from anywhere.
I am grateful to TAs and my friends Khush Jain and Aayush Rajesh for resolving issues related to docker installation and usage.

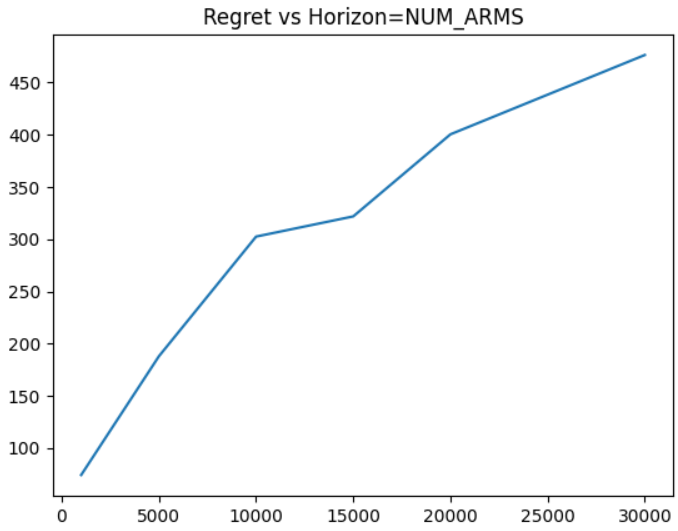

Figure 5: Task 3 ϵ Greedy sampling