Assignment 1 Report

Nikhil Manjrekar

September 2022

Contents

1 Task 1 **1**
1.1 UCB: Upper Confidence Bound 1
1.2 KL UCB: Kullback Leibler Upper Confidence Bound 2
1.3 Thompson Sampling 3

2 Task 2 **3**
2.1 Implementation 4

3 Task 3 **4**
3.1 Implementation 4

1 Task 1

1.1 UCB: Upper Confidence Bound

1.1.1 Implementation

Implementation part includes the initialisation of following variables:

- `mu` It is numpy array and is used to store emipirical mean reward of each arm of our bandit
- `number_of_pulls` A numpy array to stores count of number of times each arm was pulled
- `time` An integer to store the current time step of the algorithm

For the algorithm I have used the numpy library in Python. Essentially I have used the `mu` vector and to each element of this vector I have added the corresponding quantity as $\sqrt{\frac{\log(\texttt{time|)}}{u_a^t}}$. Here u_a^t represents the number of times the arm **a** was pulled till time **t**. So this quantity, as stated earlier, was stored in the vector `number_of_pulls`, where element a in it corresponded to arm a of the bandit. So, for vectorised division of $\log$ `time` with `number_of_pulls`, I used the `np.divide` function in numpy to handle the cases of division by zeros without exception. In order to find the optimal choice of arm, I used the `argmax` function of numpy to return the index of arm with maximum value of in above calculated vector.

Then the state update was done in the same manner as done for the epsilon greedy function, already given in the code.

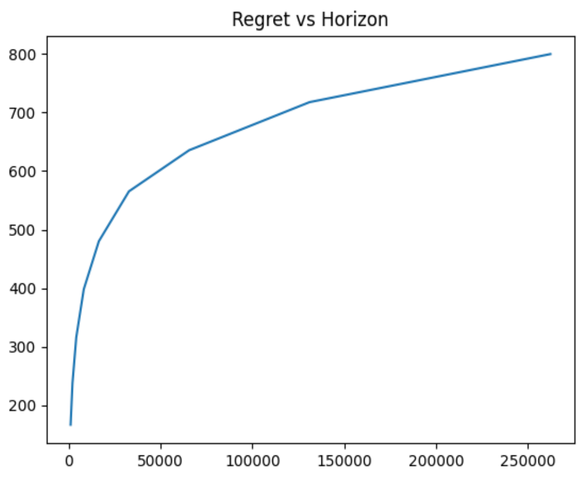

Figure 1: Plot for UCB sampling task 1

The shape of the graph is same as what expected as the expected complexity of the regret for UCB is of the order $\mathcal{O}(\log T)$ as explained in the class

1.2 KL UCB: Kullback Leibler Upper Confidence Bound

1.2.1 Implementation

Implementation part includes the initialisation of following variables:

- `mu` It is numpy array and is used to store emipirical mean reward of each arm of our bandit
- `number_of_pulls` A numpy array to stores count of number of times each arm was pulled
- `time` An integer to store the current time step of the algorithm

The variables are all same as the normal UCB algorithm. In the first `num_arms` time steps, I have sampled the arms in a round robin fashion, i.e. I have selected each arm atleast once so as to make the number of times each arm was pulled to be one. After this for its further implementation, I created a member function named `KL` which is used to find the Kullback-Leibler Divergence according to the given equation:

$$KL(p,q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$$

I also took care of the base cases when p=0 and p=1. (I didn't take care of cases like q=0 and q=1 because q lied between p and 1 according to algo.). I also created a binary search algorithm to search on the values of q, named `binary_search`. As we knew that KL function is convex up, I searched on the values of q, if it satisfied the value of the target quantity (KL(p_a,q) = $(\log t + c \log \log t)/u_a^t$, c=3, where u_a^t is the number of pulls for arm a till time t) upto an error of $\epsilon = 0.01$. I didn't keep the epsilon too small as it was taking really long time to generate plot in simulate.py. I did this search on q, for each arm of the bandit and stored all the values in another vector named `q_arm`. And then found the index having the maximum value in the vector above, which was also the arm to pull in the current time step. One more thing I noticed here was that when I was experimenting with the values of c, when I lowered it 0 the results were far better than when I put it equal to 3. However, the plot given below is generated using c=3.

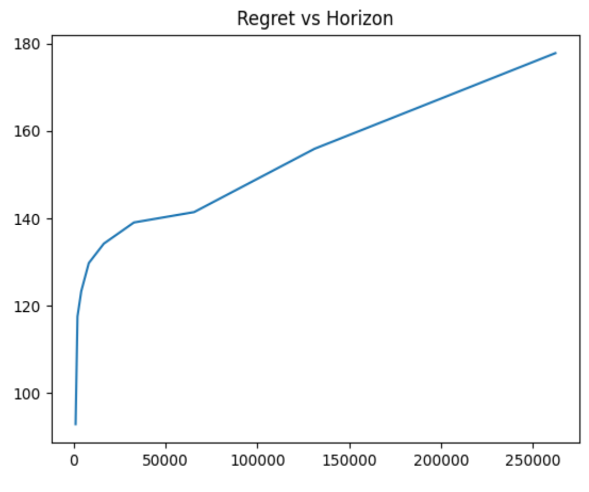

Figure 2: Task for Kl UCB Sampling for task 1

The plot is as expected as the values are more smaller in comparison to normal UCB which was expected as the KL UCB gives more tighter bound than UCB.

1.3 Thompson Sampling

1.3.1 Implementation

Implementation part includes the initialisation of following variables:

- `successes` A numpy array to store the no. of successes (reward=1) for each arm of bandit
- `failures` A numpy array to store the number of failuers (reward=0) for each arm of bandit

For the implementation part I used the numpy library (`np.random.beta()`— to generate beta distribution taking the input α = successes +1 and β = failures +1.(Python broadcasts the 1 to perform element wise addition). The samples were stored in the variable named `beta` and then the index with maximum sampled value in the beta was chosen as the arm to be pulled in the current time step. After this the state was updated as follows:

$$successed[arm\_selected] += reward$$

$$failures[arm\_selected] += (1 - reward)$$

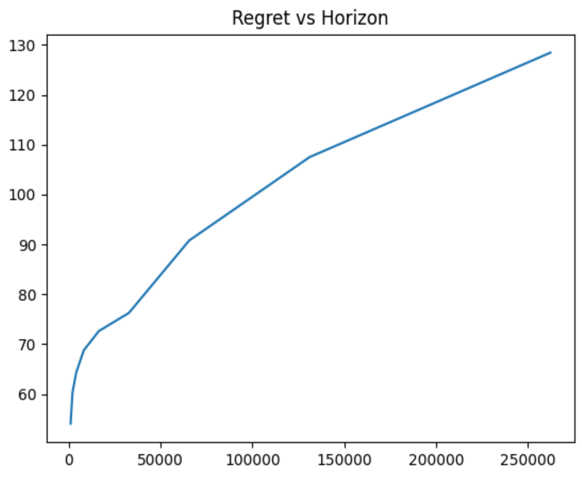

Figure 3: Plot for Thompson Sampling for task 1

The shape of the graph is same as what expected as the expected complexity of the regret for Thompson Sampling is of the order $\mathcal{O}(\log T)$ as explained in the class

2 Task 2

In this task, we were supposed to return the `batch_size` number of pulls the algorithm is supposed to make

2.1 Implementation

For the implementation I have used the **Thompson Sampling**. The variables used are same as task1 .I am running a for loop batch size number of times and applying the thompson sampling same as task1 just that arm index with maximum sampled value from the beta distribution is being stored is dictionary named `result` where key is the arm index and the value is the count of occurences of this arm. Then I am returning `result.keys()` (arm indexes) and `result.values()` (corresponding counts) as was required. And I am updating the state of the algorithm in the same manner as task 1

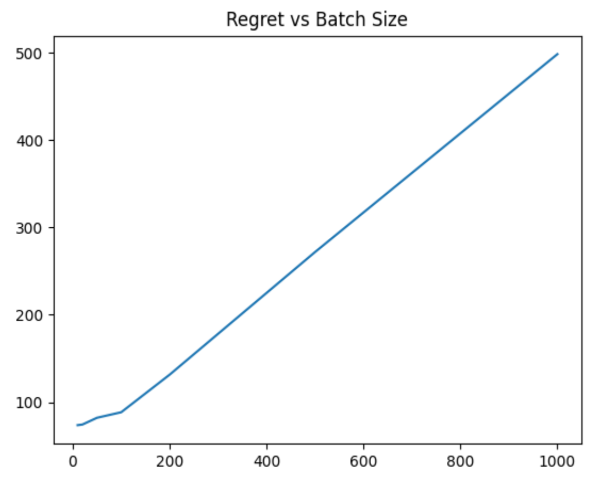

Figure 4: Plot for Task 2

The regret is increasing with the batch size as expected because with large batch size we are spending more time steps to realize the optimal arm of the bandit.

3 Task 3

3.1 Implementation

For this I sampled a subset of arms of Size S using the function `random.sample()` in the random library of Python, from all the arms during the initialisation of the class instance. Then applied the **Thompson Sampling** only on this subset arms. Intuition behind selected a subset was that, we couldn't have found the optimal arm with the given horizon is equal to total number of arms. That's why we selected a subset of it, so that one of the arms with high mean value also gets sampled in the subset. I experimented S with different values as function of number of arms such as 0.01n , $\sqrt{n}$, $c \log n$ where it worked best for c=10,. But overall good result was only given by $\sqrt{n}$. (n is the number of arms)

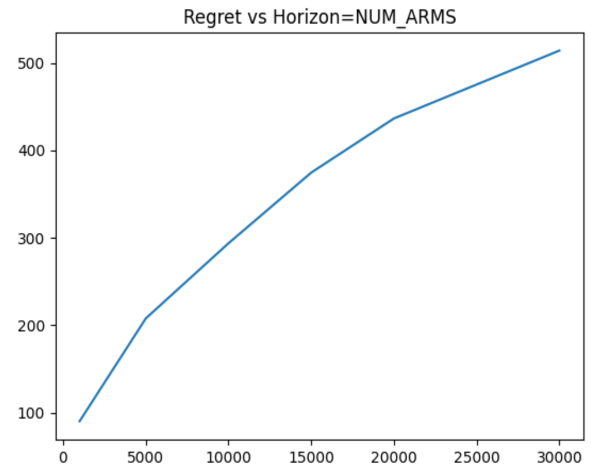

Figure 5: Plot for Task 3

The plot was drawn by taking the S to $\sqrt{n}$. The plots looks and sub linear.