CS 747 - Foundations of Intelligent and Learning Agents

Programming Assignment 1

Prakhar Vijay Diwan, Roll no. 180100083

Note:
I have completed tasks 1 and 2 and have filled the outData.txt file as desired. I haven't implemented tasks3 and 4, and hence have left the corresponding lines in outData.txt as dummy lines for the smooth running of autograder. The regret and high are both taken as 0 for in case of these tasks (3 and 4) For implementation of algorithms I have referred to the class slides provided.

For the implementation of simple arm pull and generate reward behaviour I used the provided mean probability values of the arms and compared them with the sampled value for that pull from the uniform RV between [0,1], and if that value was less that mean probabilty values of arm then I returned 1 reward else returned 0 reward. This works for the tasks I implemented i.e. tasks 1 and 2. For 3 and 4 a bit more would be required. Also, for the current time t (which is number pulls done till that instant), I chose it's initial value as 1 and then kept on incrementing by 1 for each of the subsequent pulls. This was done to avoid the ln(0) condition.

Into the code, first we have a parse the inputs appropriately, then some general parameters such as emperical mean, reward, number of pulls etc. are initialized. Then according to the choice of algorithm, the exectuion of code goes in that direction. In this, we have a function like an arm chooser, which is different for different algorithms; and also value computing functions for certain algorithms such as UCB, KL-UCB, Thompson-Sampling. After choosing the arm, the function mimicing the behaviour of arm pull and generate reward is called, and the reward obtained, then updates are done to parameters such as emperical mean of the chosen arm, number of pulls of that arm, etc.

1. Task 1

Implementation details:

A. Epsilon-Greedy Algorithm:

Pulling the initial arms weren't taken as special cases, rather, I used epsilon-greedy from the start. Initially all the emperical mean values are 0, so in that case there is a tie when exploitation is done. In case of exploration, the arm pulled is chosen at random from the arms. For breaking ties in case of equal emperical mean of 2 or more arms, I chose the arm with the lowest index in the array.

I implemented the arm choosing function in this case taking in the emperical mean array and the epsilon value. For applying uniform exploration (uniformly) w.p. ϵ, and exploit w.p. $1-\epsilon$, I used the numpy.random.uniform(0,1) function to obtain a sample (x) and would explore if $x < \epsilon$ and would exploit otherwise.

B. UCB Based Algorithm:

Pulling the initial arms weren't taken as special cases, rather, I used UCB-based algorithm from the start. I made a UCB compute function which returned the computed UCB array for the bandit. It was a straightforward function which took in the required parameters (emperical mean array, Time (depcits number of pulls till present + 1), number of pulls for each arm(passed as an array), total number of arms), computing UCB and returning it.

I implemented the arm choosing function in this case taking in the computed UCB array and using the argmax function on the array to provide me with the required arm. For breaking ties in case of equal UCB of 2 or more arms, I chose the arm with the lowest index in the array. I assumed a small positive delta

added to the number of pulls term (u_a^t) used in the algorithm for evading the divide by 0 condition and still maintaining that effect (i.e. largeness of an unsampled arm's value so that it gets explored) . Scaling c was taken as 2 (default value)

C. KL-UCB Based Algorithm: [c value chosen as 3]

Pulling the initial arms weren't taken as special cases, rather, I used KL-UCB-based algorithm from the start. I made a KL-UCB compute function which returned the computed KL-UCB array for the bandit.
I implemented the compute function using 3 functions, one was a wrapper for calling find_qmax function with required input arguments for each arm. The find_qmax function as name suggests was used for finding the max q value according to the given set of constraints. For speeding up the process of finding maximum **q** value under the constraints, I used an algortihm similar to binary search, so here as we know **q** belongs to $\widehat{[p_a^t, 1]}$, and KL divergence value increases with the value of **q** (in the allowed range only). So, instead of using a linear search type I used binary search type of algortithm to obtain reduced time complexity. And this find_qmax function called KL function for obtaining KL divergence for the 2 inputs (p,q).

I implemented the arm choosing function in this case taking in the computed KL-UCB array and using the argmax function on the array to provide me with the required arm. For breaking ties in case of equal KL-UCB of 2 or more arms, I chose the arm with the lowest index in the array. I assumed a small positive delta with the t term used in the algorithm for evading the ln(ln(1)) condition (this term is present in the constraint part).

D. Thompson Sampling:

Pulling the initial arms weren't taken as special cases, rather, I used thompson sampling from the start.
In this algorithm, initially a sampling function was called with the required arguments. This function provided an array of sampled values, one for each arm according to its beta distribution $\beta\left(s_a^t + 1, f_a^t + 1\right)$. For the sampling I used the numpy.random.beta function.
After this I implemented the arm choosing function in this case taking in the samples array and using the argmax function on the array to provide me with the required arm

PLOTS

[Here the X-axis is in log-10 scale along with the markings used (i.e. 2×10^0 corresponds to Horizon = 100, and so on)]

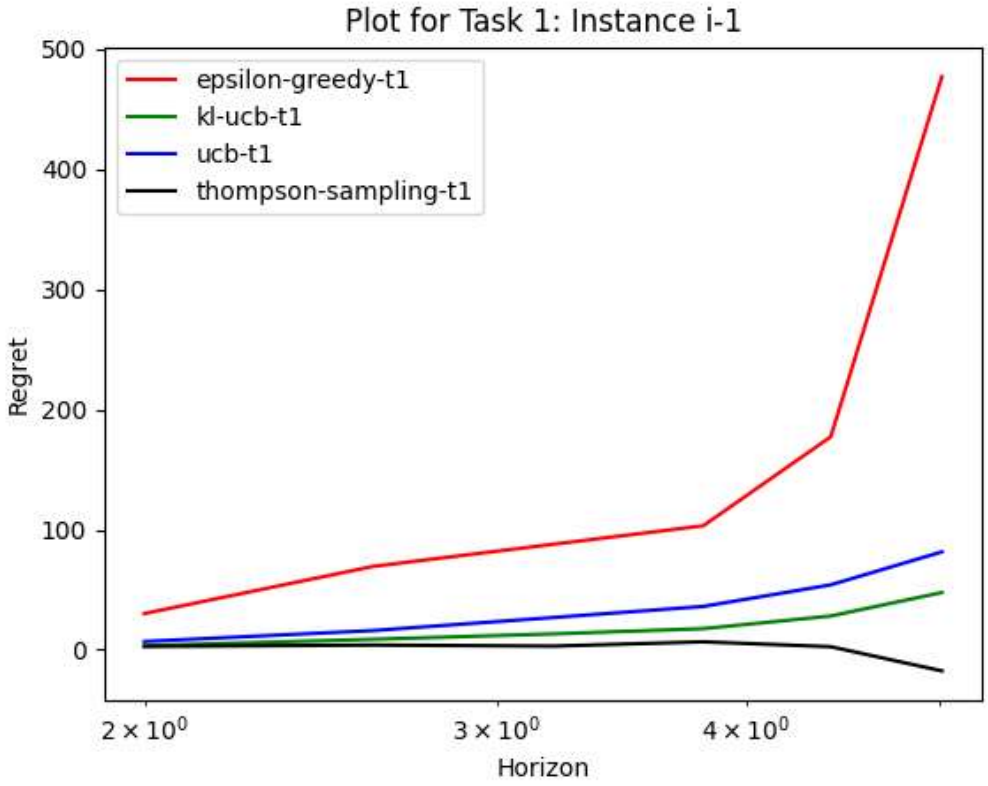

Figure 1: Instance 1: 2-armed bandit

In 1st instance, I observed an unusual behaviour in Thompson Sampling algorithm: it's regret decreased with horizon (in the later half). The general trend (regret) among the various algorithms is as expected from theory, with Epsilon-Greedy<UCB<KL-UCB<Thompson Sampling [in decreasing order of regret]. The regret becoming negative in case of Thompson sampling can be simply explained as getting more 1-rewards than probability dictates. The other algorithms' regret keep on increasing due to being unable to find the optimal arm or being unable to pull it every time (full exploitation). As in all the above algorithms there is a certain ratio between exploration and exploitation, which is quite clear in epsilon-greedy algorithm.

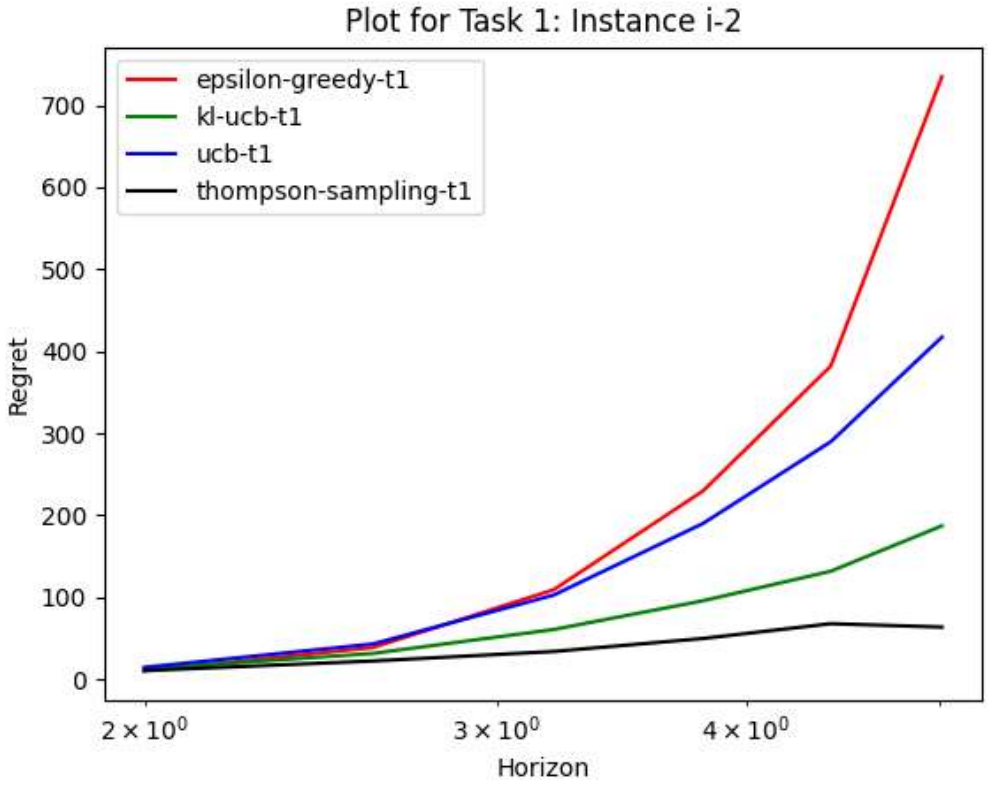

Figure 2: Instance 2: 5-armed bandit

In 2nd instance, the general trend (regret) among the various algorithms is as expected from theory, with Epsilon-Greedy<UCB<KL-UCB<Thompson Sampling [in decreasing order of regret]. I also observed that Thompson Sampling algortihm reaches a saturation in regret at the end of horizon but the other algorithms' regret keep on increasing due to being unable to find the optimal arm or being unable to pull it every time (full exploitation).

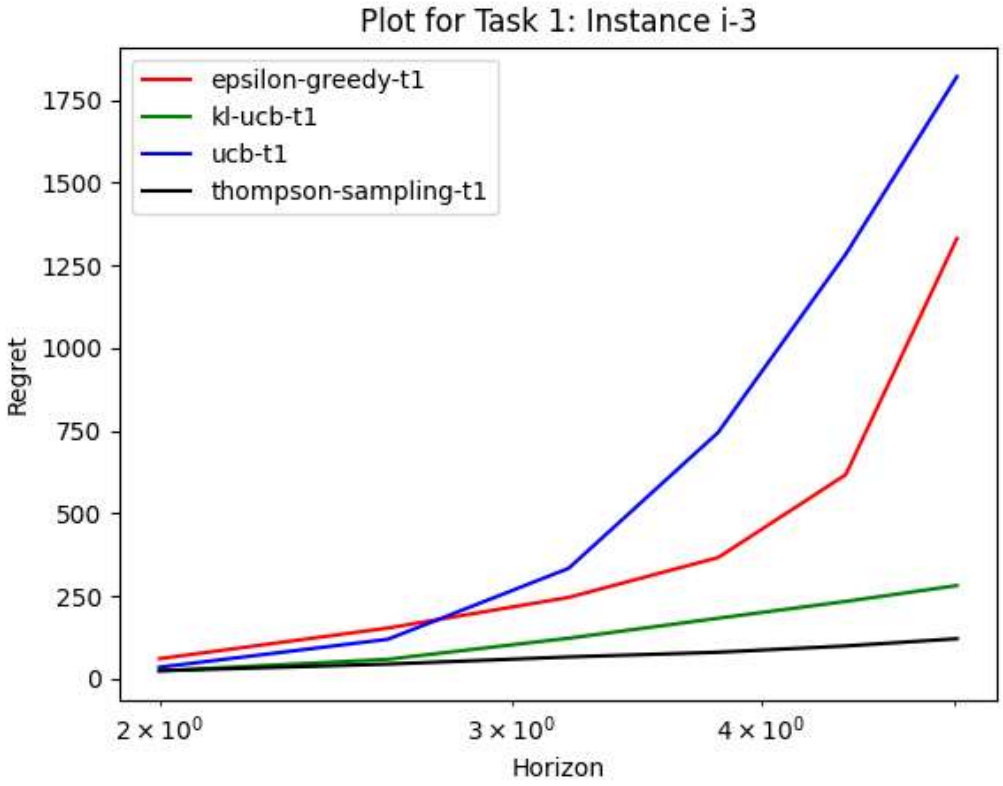

Figure 3: Instance 3: 25-armed bandit

In 3rd instance, the general trend (regret) among the various algorithms is broken from theory, as we can see from the plot that UCB crosses the epsilon greedy algorithm in regret. This can be justified as the number of arms is high in this bandit and the exploration factor (ϵ) is very small and thus usually makes the greedy choice, whereas ucb has the $\sqrt{\frac{2ln(t)}{u_a^t}}$ term in its expression which leads to it exploring more in case of larger number of arms (even more in case all the arms have close mean probabilities). Hence we see that due to limited and sufficient exploring, epsilon greedy algorithms pulls ahead (lowers regret as compared to ucb) as horizon increases as it does good amount of exploitation as compared to ucb.
I also observed that Thompson Sampling algortihm reaches a saturation in regret at the end of horizon but other algorithms display increasing regret with increase in horizon due to being unable to find the optimal arm or being unable to pull it every time (full exploitation).

2. Task 2

Implementation details:
The algorithm was implemented as was UCB in Task 1 with **c** value being a variable and being analysed in this task. So here I just added the UCB computation function which was there in task 1, just with another parameter scale being taken in for replacing the default **c** = 2 value.

PLOTS

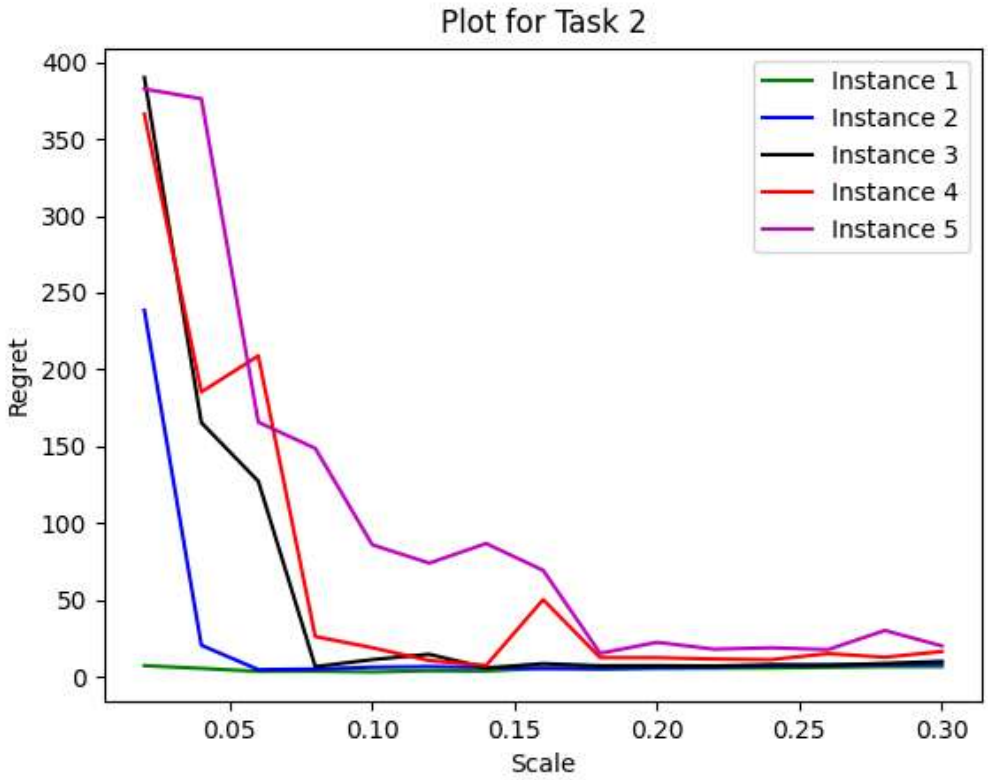

Figure 4: Task 2: 5 2-armed bandit Instances [Horizon=10,000]

Here, we see that as the scale increases from 0.02 to 0.3, the regret for all the instances decreases, there might be some spikes down the plot like in instance 4, but it reaches there. This can be explained from the fact that scale actually increases the exploration, which leads to other arms being given a chance at least, and if it is the case that optimal arm is not the current arm, then this exploration helps in lowering regret. Now this case can occur more in instances which have close mean probabilities (also explained in paragraph below the c_{opt} values)

Based on given scale range, i.e. from 0.02 to 0.3, I used the argmin() function for calculating the scale value which gave the minimum value **c** in the scale range. The value of **c** that gave the lowest regret are shown below for each of the five instances:

$$\begin{aligned} Instance\ 1\ [0.7, 0.2] &\rightarrow c_{opt} = 0.1 \\ Instance\ 2\ [0.7, 0.3] &\rightarrow c_{opt} = 0.06 \\ Instance\ 3\ [0.7, 0.4] &\rightarrow c_{opt} = 0.14 \\ Instance\ 4\ [0.7, 0.5] &\rightarrow c_{opt} = 0.14 \\ Instance\ 5\ [0.7, 0.6] &\rightarrow c_{opt} = 0.18 \end{aligned}$$

In each of these instances there was an arm with fixed probability 0.7 and the other one let's call it as the **variable arm**. Here we can see a trend in the instances that is as the instance number increases, which

implies the mean value of **variable arm** increases, the c_{opt} value increases. This can be explained as follows: As both the arms' mean probability grow closer to each other, the emperical mean value corresponding to both arms will also be close to each other, which implies the need of higher c in order to (in case not obtained) pick the optimal arm. In other words, it is difficult (more exploration required) to find the optimal arm in case we have mean probability of both the arms closeby , whereas it is very easy (little exploration) to find the optimal arm in case one of the arms has high mean probability ike 0.7 and other one has low mean probability like 0.2. Hence, the trend is explained.
Also, we see in general at around 0.18 value of c all the algorithms have their regrets minimized when comparing with scale values less than 0.18.

The code for the assignment can be found along with this report.pdf file in the submission.tar.gz file.