CS747 Programming Assignment 1

180050096 - Shashank Agrawal

Overview of code in bandit.py

Bandit.py comprises of two classes of arm and multiarm bandit and other classes for algorithms :-

- **arm class** - The class has three attributes that are number of different rewards, *rewards* and *probs*, and a member function that is *pull*. The attribute *rewards*, as the name suggests, is a list of possible rewards in that arm whereas the attribute *probs* is a list of probabilities of getting a corresponding reward in *rewards* attribute when we pull this arm.
  The *pull* member function just simulates a pull from this arm by uniformly choosing a random number in [0, 1) and then just comparing it with cumulative probabilities and getting a corresponding reward of the interval in which this number falls.
  I chose this design for arm, so that I can easily use the same for Task3 and Task4 as well.

- **multiarm_bandit class** - This class has four attributes that are *arms*, *pulls*, *pstar* and *n*-number of arms and a member function that is *reinit*. The attribute *arms* is the list of arms in bandit, *pulls* indicates the number of times particular arm has been pulled till now, *pstar* is the maximum mean reward we can have which is just maximum of rewards*probs over all arms in the bandit, *reinit* just reinitializes the value of pulls to 0 again.

- **classes of algorithms** - In general, these classes have a *bandit*, a *reinit* member funtion which takes certain attributes of class to their initial values, a *pull_arm* which has attribute i and it simulates pulling arm i in the *bandit*, and an *algo* member function which takes epsilon (*eps*), scale (c), horizon (*hz*), seed (*seed*), and list of horizons (*hzs*) at which we want regret as attributes.
  I used **numpy random generator** and this *algo* first seeds this random generator with seed and run the algorithm for given number of horizons.
  More specifics for each algorithm class will be given in the following sections.

We also have two functions extract_probs and extract_probs1 which takes input from instances and outputs the corresponding bandit. I used class for everything, so that we have abstractions in the code and can easily be edited if needed.
Four functions run_t1, run_t2, run_t3, run_t4 can be ignored as I used them to generate results for tasks.

Task1 Sampling Algorithms

- **Epsilon Greedy Algorithm** :- The algorithm is implemented in the *eps_greedy* class. In this algorithm, I generated random numbers array of length *hz* (horizons) uniformly from $[0, 1)$.
  At ith horizon if ith number is less than *eps*, we choose the arm uniformly at random, basically we explore, else we will pull the arm with maximum empirical probability (exploit) using **np.argmax()**, which just returns the lowest index at which max appears and ties are resolved.

- **Upper Confidence Bound (UCB) Algorithm** :- The algorithm is implemented in the *ucb* class, each arm has an **UCB** value (*ucb_val*) which is determined as follows: $ucb_a^i = \widehat{p_a^i} + \sqrt{\frac{c \ln i}{u_a^i}}$ where ucb_a^i is the UCB value of the arm a at ith horizon, $\widehat{p_a^i}$ is the empirical mean of arm a at ith horizon (*emp_mean*), u_a^i is the number of pulls of arm a till the ith horizon, and c is the confidence value, also known as *scale*, which is you can say a parameter for the algorithm.
  Initially I shuffle the arms and then, in case *hz* is less than number of arms, we just pull first *hz* arms, otherwise, we pull each arm once.
  And then, at each ith horizon we just pull the arm with maximum value of ucb_a^i by using **np.argmax()**, which just returns the lowest index at which max appears and ties are resolved.
  Now, notice that if some arm is not explored then value of $\sqrt{\frac{c \ln i}{u_a^i}}$ will be large as u_a^i will be small, Hence this value kind of denotes exploration, and we already know empirical mean denotes exploitation, so basically if c is large, we are giving more importance to exploration, and if c is small, we are giving more importance to exploitation. Hence, we can alter c to balance exploration-exploitation and find the optimal choice (which is what we are doing in Task2).

- **KL-UCB Algorithm** - The algorithm is implemented in the *kl_ucb* class, each arm has an **KL-UCB** value (*kl_ucb_val*) which is determined as follows:

$$\text{kl-ucb}_a^i = \max\{q \in [\widehat{p_a^i}, 1] \mid KL(\widehat{p_a^i}, q) \leq \frac{\ln i + c \ln \ln i}{u_a^i} \} \text{ , where } c \geq 3$$

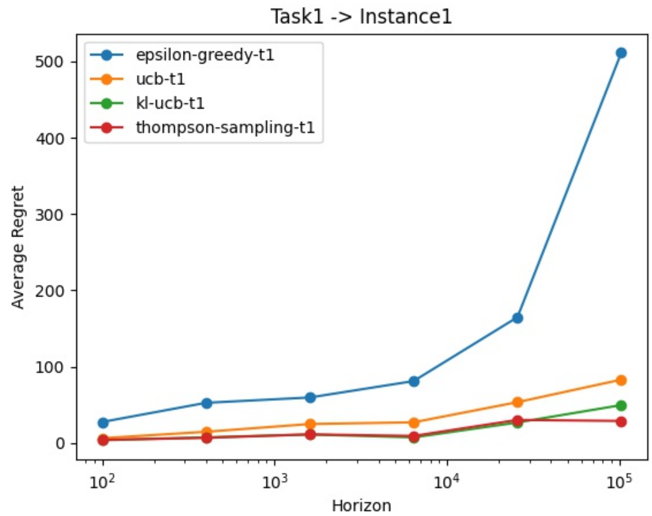
  kl-ucb$_a^i$ is the KL-UCB value of arm a at ith horizon, the other variables have the same meanings as defined in the UCB algorithm and the kl-divergence, $KL(x, y)$, is defined as

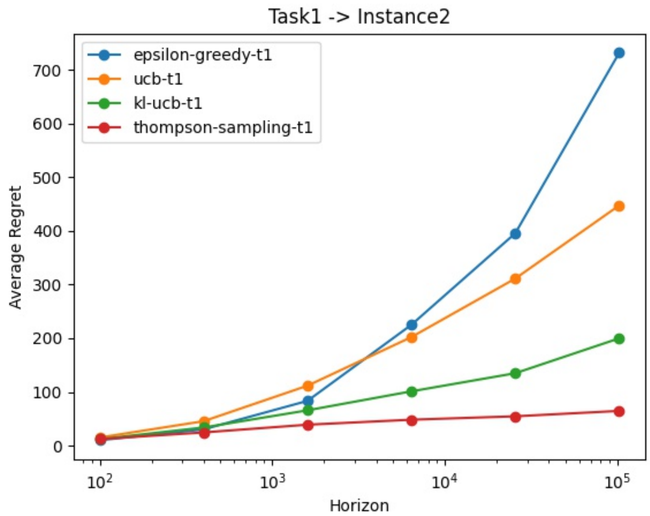
$$KL(x, y) = x \ln \frac{x}{y} + (1 - x) \ln \frac{1 - x}{1 - y}$$

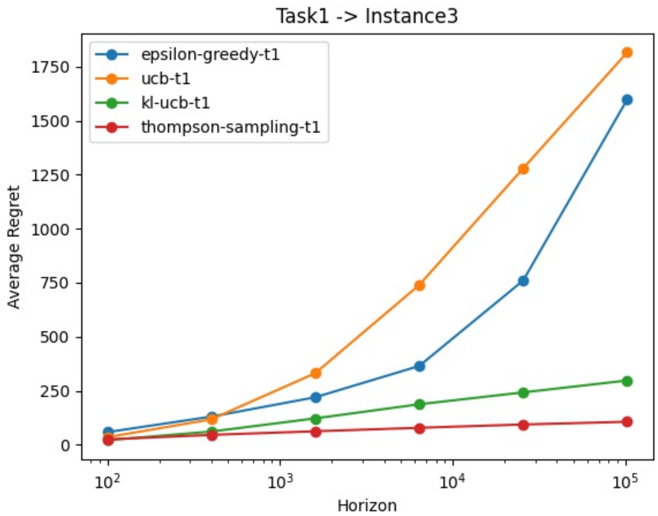
  Initially we do the same thing we did in UCB algorithm above.
  Now, after we pull each arm once, at each ith horizon we just pull the arm with maximum value of kl-ucb$_a^i$ by using **np.argmax()**, which just returns the lowest index at which max appears and ties are resolved.

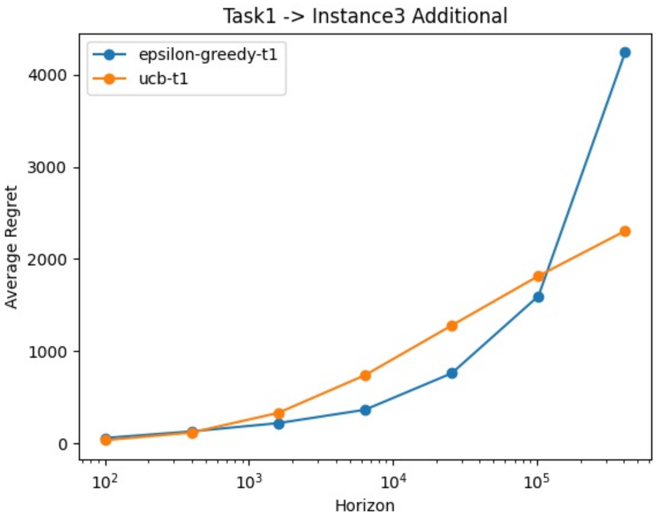
- **Thompson Sampling** - The algorithm is implemented in the *thompson_sampling* class, each arm is associated with number of successes (*success*) and number of failures(*failure*). As arms of our Multiarm Bandit are Bernoulli in Task1, we can use this algorithm. This algorithm assumes the *Beta* prior for the distribution of mean reward for each arm with the parameters $\alpha = s_a^i$ and $\beta = f_a^i$, where s_a^i is the number of successes associated with arm a and f_a^i is the number of failures associated arm a till ith horizon. Initially, number of successes and number of failures are zero for all arms.
  Thompson Algorithm is implemented in two steps (We kind of have both step in every algorithm mentioned above) :-

  - **Computational Step:** For every arm a, we draw a sample x_a from $Beta(s_a^i + 1, f_a^i + 1)$.
  - **Sampling Step:** Pulls the arm with maximum value of x_a by using **np.argmax()**, which just returns the lowest index at which max appears and ties are resolved.









Observations :-

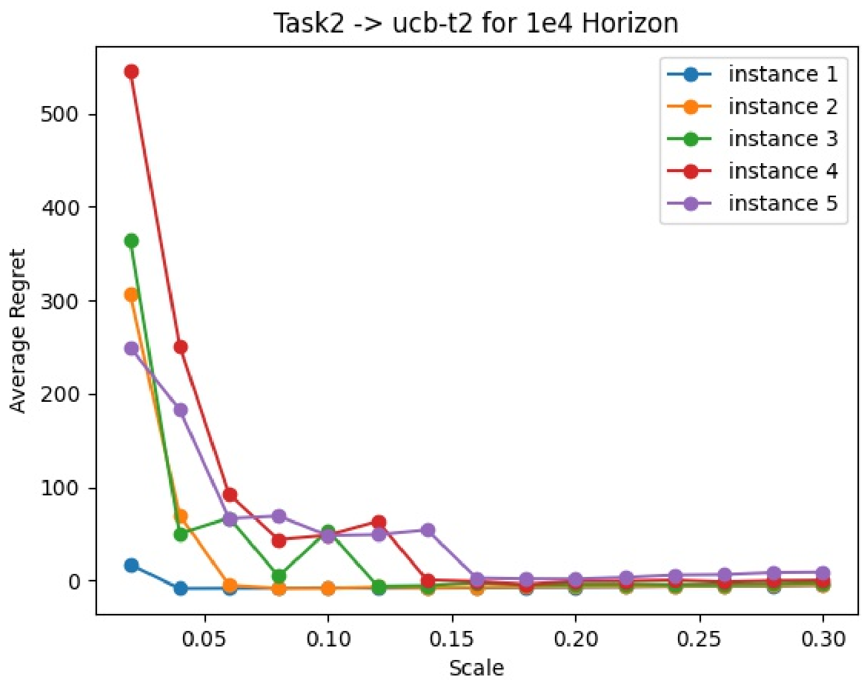
- As we can see from the plots of Instance 1 and 2 that Epsilon-Greedy shows exponential growth of regret with respect to log of horizon (which is nothing but linear regret as expected), while other algorithms show linear growth of regret with respect to log of horizon (which is logarithmic regret).
- In Instance 3 plot, UCB is giving higher regret as compared to that of Epsilon-Greedy, this is because in Instance 3, we have more arms as compared to that in Instance 1 or 2, and UCB algorithm must haven't explored all the arms well yet, that's why doing even worse than Epsilon-Greedy which only do exploitation, slopes of these two algorithms also suggest that for more horizons, UCB will eventually be better than Epsilon-Greedy. That's why I plotted one additional graph for just UCB and Epsilon-Greedy for horizon 409600 and this graph proved that our observation is correct.

Task2 Finding Optimum Scale Value for UCB Algorithm

The optimal scale values for each instance is as follows:

Instance	1	2	3	4	5
Optimal Scale	0.04	0.1	0.12	0.18	0.2
Average Regret on Optimal Scale	-8.48	-8.32	-6.02	-4.1	1.56

In each of the instances, we have two arms, both bernoulli, one with $p = 0.7$, and other with $p = 0.2, 0.3, 0.4, 0.5, 0.6$ for instances 1 to 5 respectively. As we can see that other arm is getting closer to *pstar* (maximum mean reward), UCB algorithm needs to explore more as it gets confusing which arm is better as p increases from 0.2 to 0.6 and that's why we see a trend of increasing Optimal scale as we go from Instance 1 to Instance 5.



Task3 Minimising regret for Multi-armed Bandit with Multiple Rewards

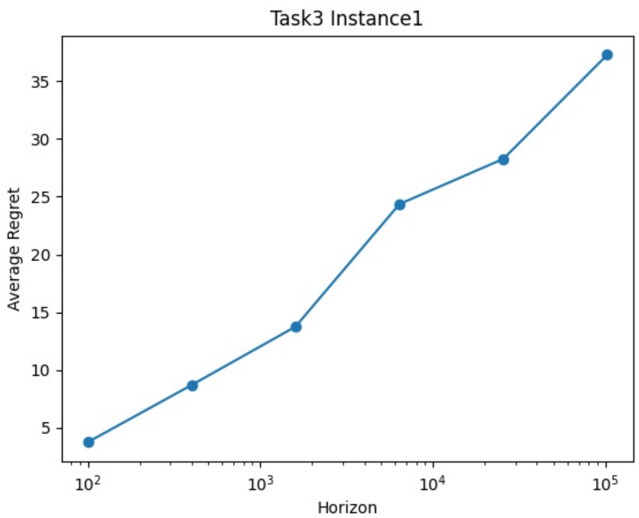
Thought :-

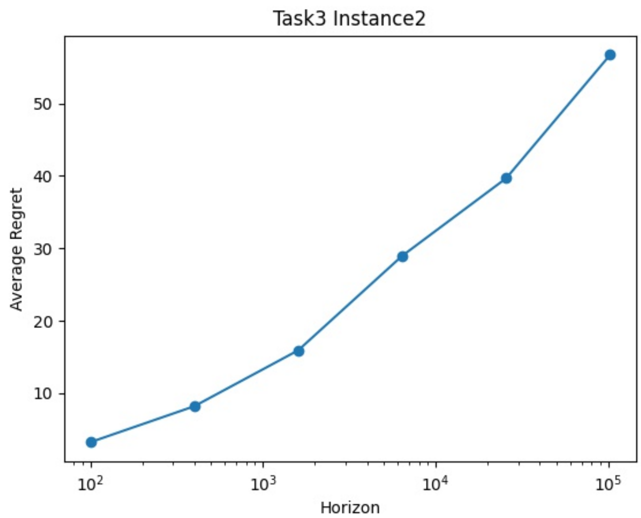
- I already discussed about UCB Algorithm in Task1 and effect of scale c on exploration and exploitation by algorithm. Also we proved in week2 theory assignment that UCB Algorithm has infinite exploration, i.e., it won't happen that we pull arm a at ith horizon and we keep pulling it forever.
- Now, we have arms with multiple rewards we surely can't use Thompson Algorithm, We can use KL-UCB Algorithm but we don't know how to balance exploration and exploiation. We surely need to explore more as we have multiple rewards possible from each arm, and we have infinite exploration with UCB and we can also maintain balance between exploration and exploitation, That's why I chose UCB, and picked 0.2 as the scale c, because in the case of most confusing instance of Task2, i.e., Instance 5 $c = 0.2$ minimizes the regret.

Algorithm is exactly same as UCB algorithm just with different scale ($c = 0.2$).

Observations :-

- Graphs are linear as expected, as UCB Algorithm gives logarithmic regret with respect to horizon.





Task4 Minimising HIGHS-REGRET

Thought :-

- In this task, we want to maximize the times we get rewards exceeding the threshold. We define HIGH when the outcome exceeds the threshold, otherwise it's defined as LOW. So, we want to maximize number of HIGHs.
- Consider any particular arm which gives multiple rewards with given probabilities, out of these rewards consider only those rewards that exceeds threshold given to the program and when we sum the corresponding probabilities of these arms, we will get the probability that outcome from this arm will exceed threshold. Basically, We know the probability of getting HIGH and probability of getting LOW (as there are only two relevant outcomes).
- For our task, any arm which gives multiple rewards with given probabilities is equivalent to a Bernoulli arm with p defined as in the last paragraph, where reward = 1 means it is a HIGH outcome otherwise LOW.
- We know from Task1 that **Thompson Sampling** works best when there are only two outcomes, one is considered as success and other as failure.

Let arm a has *rewards* and *probs* as described initially in the report, then the value of p_a for HIGH of arm a is calculated as,

$$p_a = \sum_{i, rewards[i] > th} probs[i]$$

Now the value of p for equivalent Bernoulli arm will be nothing but p_a and now $p^* = \max_{a \in A} \{p_a\}$ and HIGHS-REGRET is given by $p^* hz - HIGHs$, where hz is the horizon.

Observations :-

- For instance 1 and threshold 0.2 we have $p_a(s) = 1,\ 1,\ 0.8$ we can see that average HIGHS-REGRET is very low as threshold is low and two arms have $p_a = 1.0$.
- For instance 1 and threshold 0.6, we have $p_a(s) = 0.4,\ 0.2,\ 0.3$ probabilities are at a difference of 0.1 and Thompson Sampling works very good here.
- For instance 2 and threshold 0.2, we have $p_a(s) = 0.85,\ 0.9,\ 0.81$ probabilities are very close, so the beta priors may also have close alpha, beta and will pull 0.81 and 0.85 one arms also frequently, and maybe that's why it has slightly more regret as compared to Instance 1 and th = 0.6.

- For instance 2 and threshold 0.6, we have $p_a(s) = 0.3,\ 0.53,\ 0.25$ probabilities 0.3 and 0.25 are far from 0.53, that's why most of the times 0.53 one will be pulled, and that's why it has lesser regret as compared to instance 2 and th $= 0.2$.
- We can also see that there's a drop in HIGHS-REGRET in case of Instance 2 and threshold 0.6, this maybe due to beta prior for 0.53 arm having enough more successes now that it is picked most of the times.