CS747: Assignment-1 Report

Balbir Singh (R.N.- 22M0747)

September 2022

1 Task 1

1.1 UCB (UPPER CONFIDENCE BOUND)

Code Explanations:

Assumption(due to issue): Serial wise pull each arm for first num-arms pull, where num-arms is number of bandit arm. UCB of each bandit arm is zero. Empirical mean of each is also zero. UCB values is calculated after first num-arm pull, As counts[arm]=0 initially, it was giving divided by zero error.

Variable Used:

- **h** is variable to store number of step (or total number of arm pull starting from zero and increment by one for each bandit arm pull till horizon).
- **num-arms** is total number of bandit arm.
- **counts** is array of size num-arms which will keep track of total number of time each arm pull.
- **mean** is array of size num-arms which will store empirical mean of each arm till h time step e.g mean[arm] is the empirical mean of bandit arm index arm.
- **ucb** is array of size num-arms which will store ucb value of each bandit arm

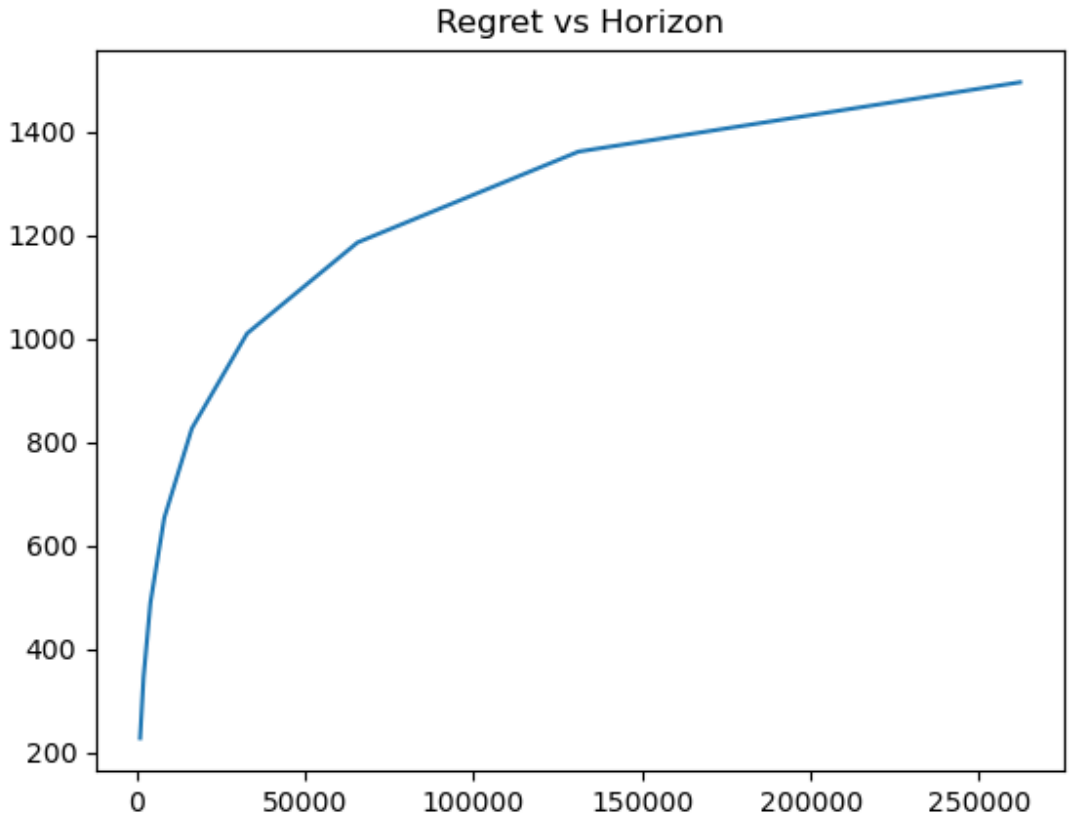
give_pull(self): This function return arm index value which will be pulled (starting from 0 index). If total number of pull (h) is less than total number of arms (num-arms) then return value of h as an index to pull.(This is done to pull each arm once serial wise i.e round robin) once h become equal and greater than num-arms return the arm index whose ucb value is maximum.

get_reward(self,arm-index,reward): For each call of this function, it updates the following state based on arm pulled and reward received.

- update the mean of arm-index according to reward it received.
- increment counts of arm-index by one and also increment the h (total arm pull till time step) by one.
- calculate and store the new ucb value for each arm according to the UCB formula in class slides.

Note: Here I am using variable-name instead self.variable-name which I used in program, as self is for particular bandit instances (generally self represent the instances of class). Most of the variable used in another algorithm and task are mostly same as used in ucb.

- From plot it can be observe easily that it give logarithm regret.



Plot (x-axis = Horizon, y-axis = Regret

1.2 KL-UCB (KULLBACK–LEIBLER UPPER CONFIDENCE BOUNDS)

The implementation of this algorithm is same as UCB the only difference is in the calculation of KL-UCB values for each arm. kl-ucb value is calculated after first num-arm pull. As counts[arm]=0 which will give divide by zero error. For h (same variable as defined in ucb) less than num-arms, kl-ucb value remains zero, this assumption will handle both cases divided by zero as well as log(0) i.e loglog(h) in kl() calculations where h less than 3.

calculation of kl-ucb of arm kl-ucb[arm] is equal to q ,where q lies between empirical mean p_ and 0.999 which give maximum value of kl(p_,q). To find this q binary search is implemented. Stop Condition for Binary Search is either start and end (initial start = p_ and end = 0.999) difference is less than 0.0001 or difference between kl and kl(p_,q) is less than 0.0001. where

$$kl = \frac{(ln(h) + 3 * ln(ln(h))}{counts[arm]}]$$

calculation of kl(x,y) method This method is call by kl(p_,q). Boundary condition x = 0 and x = 1 handled carefully by simlifying the expressions.

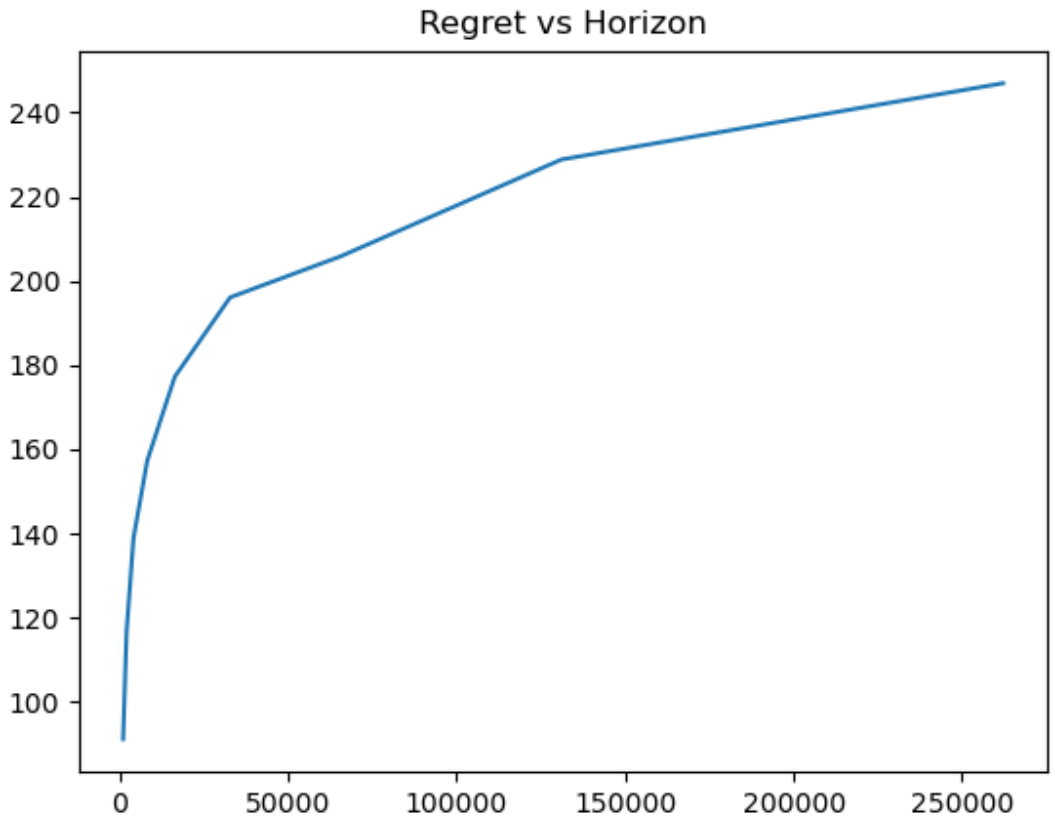
$$kl(x, y) = xln\frac{x}{y} + (1 - x)ln(\frac{1 - x}{1 - y})$$

Surprising pattern When I increased stop conditions from 0.0001 to 0.01 , in that case kl-ucb is taking less time to run but surprisingly it gives less regret.

Issues faced

- KL-UCB takes lots of time to run on docker.
- Deciding stop conditions for Binary Search was difficult. of time wasted to check for boundary conditions.
- log(x/y) was giving math domain instead of in domain range. log() will give this error for very very small values

From the plot it is clear that kl-ucb gives logarithms regret also regret is improve lots compare to ucb



Plot (x-axis = Horizon, y-axis = Regret

1.3 Thompson-Sampling

Code Explanations:

Initial assumption: No initial Assumptions.

Variable Used: counts and success is array of size num-arms. counts will keep track of total number of time each arm pull and success will keep track of number of success(reward =1) of each arm.

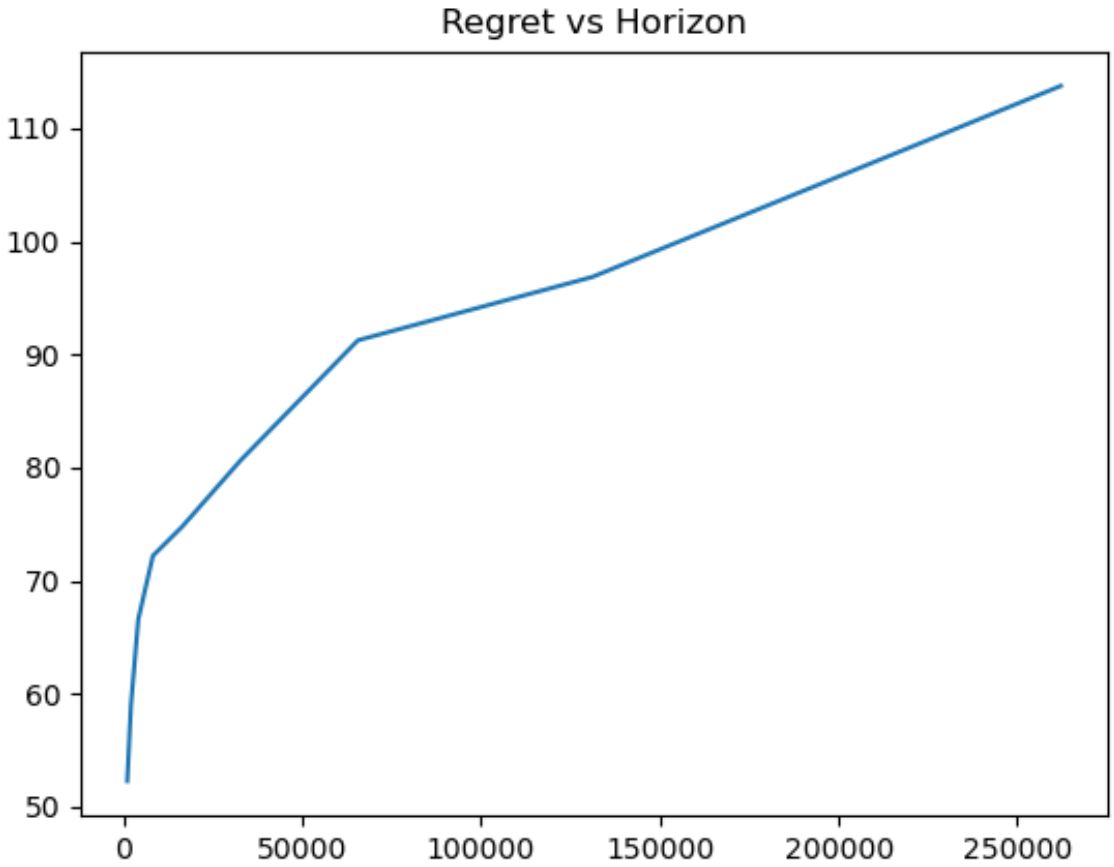
give_pull(self): For each arm draw sample form beta distributions with parameter success and failure of arm i.e beta(success[arm]+1,failure[arm]+1) (here failure[arm]=counts[arm]-success[arm]) and store these value in sample array size of num-arms. This function return the arm index of the bandit arm whose sample values are maximum.

get_reward(self,arm-index,reward): For each call this function updates the following algorithm's internal state based on arm pulled and reward received.

- increment counts of arm-index by one.
- increment success of arm-index by one only if arm-index get reward = 1.

Observations (trends and issue)

- When I used np.random.seed(0) It give negtive regret.
- Thompson Sampling can be done directly without any initial assumptions of Round Robin or random sample. As initially, success and failure of each arm is zero so Beta(1,1) which is same as uniform distributions.
- Thompson-sampling is best among E-greedy, ucb, kl-ucb.
- Thompson-Sampling give Logarithm regret



Plot (x-axis = Horizon, y-axis = Regret

2 Task 2

The idea here is in each batch size select all the arm with weightage (from weightage, I mean number of time selected arm pull) which will give maximum reward. From task one it can easily observe that pulling

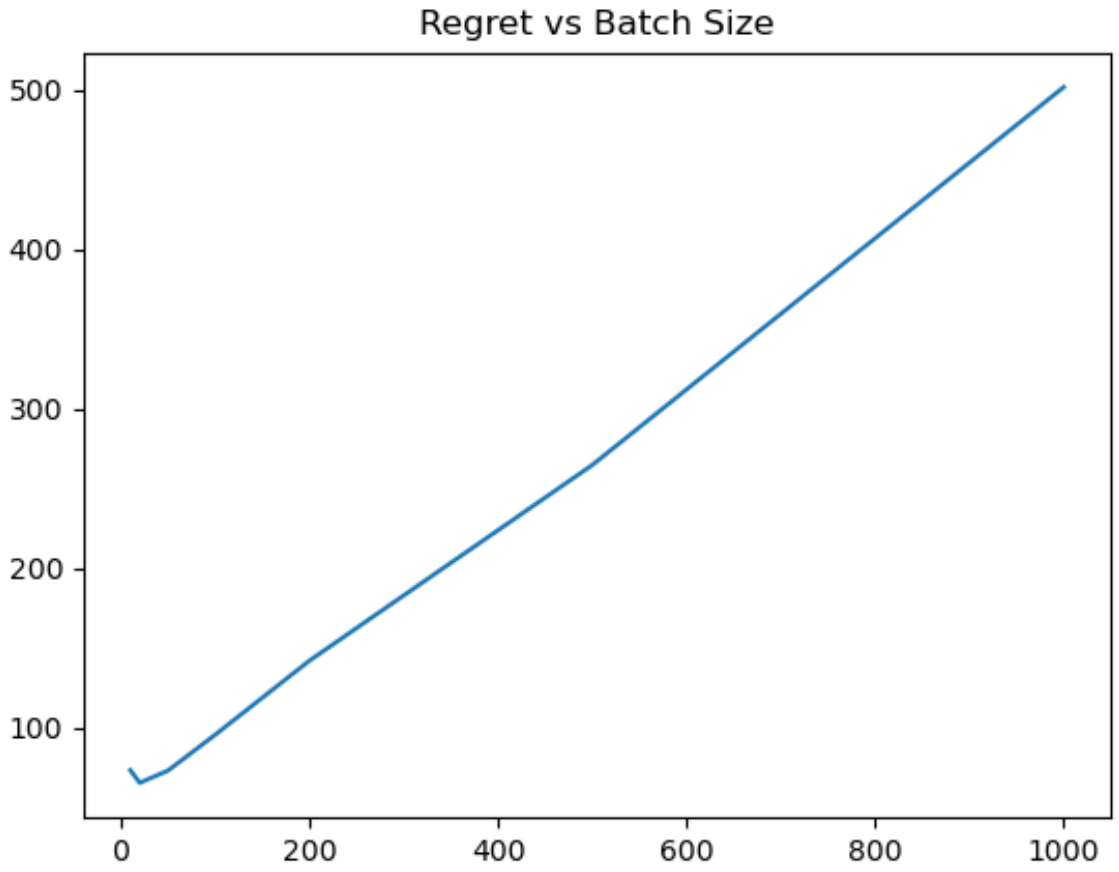
arm according to sample value from beta distributions (i.e Thompson-Sampling) gives good results. So I am going to use this results to decide arm-to-pull and number-of time-arm-to-pull.

Let num-arms = 10 batch-size = 20. from 1 to 20 , for each arm (0 to 9) sample from beta distribution (parameter is again success and failure of that arm)(ideas comes from thompson-samplig) and save the index of arm to **ls** whose sample values is maximum, in this way we have 20 arm index values (in programming assignment **ls** store this value, which is list of size equal to batch-size).

In these 20 values it might possible that many arm is repeated, most likely the arm which give highest reward. e,g suppose in **ls** arm 2 (recall arm is 0 indexed) repeated ten time, arm 4 repeated four time and arm 9 repeated six time. then **give_pull()** method in our algorithm return tuple of two list arm-index-to-pull and no-of-time-that-arm-pull. e.g in our running example ([2, 4, 9], [10, 4, 6]).

Success and Counts of each arm can be easily updated by **get_reward()** method. Input to this method is arm-reward which contain dictionary arm index and list of success and failure of this arm. e.g 2: np.array([1, 1, 1, 0, 1, 1, 0, 1, 0, 1]), 4: np.array([1, 1, 0, 0]), 9: np.array([0, 1, 0, 1, 0, 0]), for this arm-reward value success and counts of arm 2 is increase by 7 and 10 respectively and similarly for arm 4 and 9. In this method

- The number of time the sample value from beta distribution is equal to batch-size * num-arms.
- Regret is Linear with batch size, By increasing batch size we can't improve on regret.
- For batch size = 1 this algorithm become same as thompson-sampling.
- for larger batch size this algorithm perform worst than thompson-sampling.
- This happened because by dividing batches we are giving some chances to suboptimal arms



Plot (x-axis = Batch Size, y-axis = Regret

3 Task 3

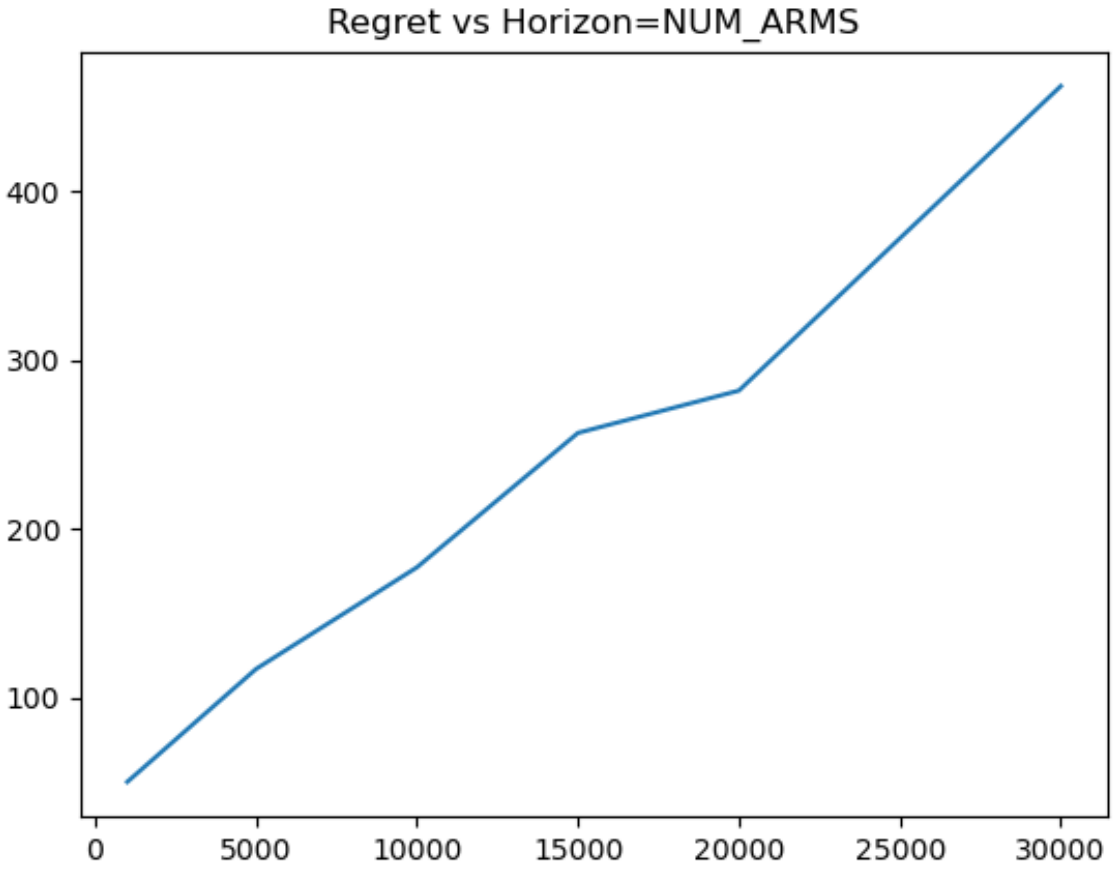
As numbers of arms is equal horizon, so we have less time to explore. One can follow the greedy strategy i.e greedy with empirical means and other things which support this greedy strategy is that the arm means are distributed regularly (in arithmetic progression) between 0 and (1 - 1/numArms). As optimal arms have mean equal to (1-1/numArms), on an Expectations it is most unlikely that optimal arm will give zero reward.

One strategy can we to pull arm until it give zero reward, when it give zero reward give then check for empirical mean if this mean greater than some limit then give another chance to that arm otherwise sample other arm.

From above observation, my approach to solve this task is first check the highest empirical means of arm if it is greater than some high limit (as optimal arm have high means) return index of that arm otherwise return the arm index randomly. Update the mean and counts of each arm accordingly

This limit value can be made dependent to number of horizon or mean of optimal arm i.e (1 - 1/numArms)

- From graph, it can we observe that Regret is linearly dependent num-arms, i.e obvious.
- One things to observe is that this strategy is better than sampling each arm once



Plot (x-axis = (Horizon = NUM-ARMS), y-axis = Regret