CS747 ASSIGNMENT 1

KOUSTUBH RAO, 200100176

September 2022

1 Q1

Numpy seed is set to zero to produce constant result.

1.1 UCB

1.1.1 Initialize

- Initialize a time variable to 1. It is not initialized to 0, to simplify the calculations.
- Initialize counts to a numpy array of ones with num_arms elements.
- Initialize values to a numpy array of zeros with num_arms elements.
- Initialize ucb to a numpy array of zeros with num_arms elements.

1.1.2 give_pull

- Increment the time variable
- Return the max value of the UCB array

1.1.3 get_reward

- Update values and counts.
- Update ucb (vectorized) as values + np.sqrt(2*math.log(time)*np.reciprocal(counts)).

This graph is as expected as UCB is non tight and has logarithmic regret.

1.2 KL_UCB

Define a KL(x, y) function

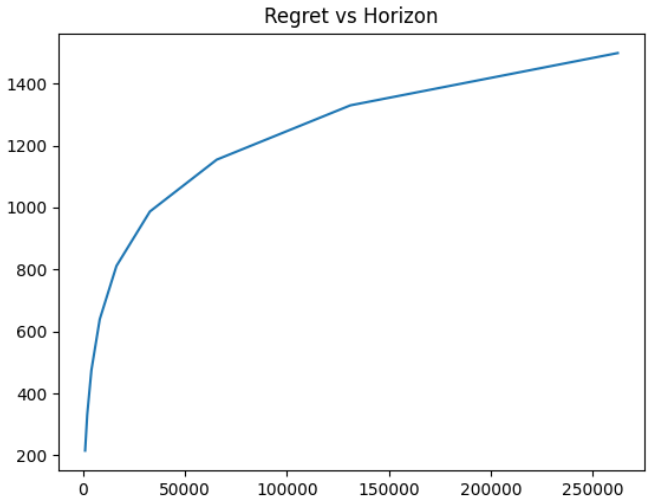

Figure 1: UCB graph

1.2.1 Initialize

- Initialize a time variable to 1. It is not initialized to 0, to simplify the calculations.
- Initialize counts to a numpy array of ones with num_arms elements.
- Initialize values to a numpy array of 0.5s with num_arms elements.
- Initialize kl_ucb to a numpy array of zeros with num_arms elements.

1.2.2 give_pull

- Increment the time variable
- Return the max value of the KL_UCB array

1.2.3 get_reward

- Update values and counts.
- Updating kl_ucb is a bit involved. First calculate ln(t) + 3ln(ln(t)) and store in variable v. If v is less than zero assing value to kl_ucb else do a binary search between value and 0.999 for each arm. Tolerance was taken to be 0.001.

This graph is as expected as KL_UCB is tight and has logarithmic regret. Hence regret is lower when compared to the graph for UCB.

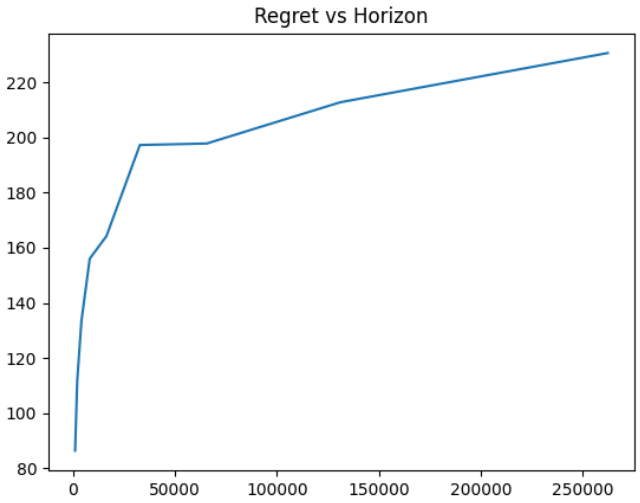

Figure 2: KL_UCB graph

1.3 Thompson Sampling

1.3.1 Initialize

- Initialize success and failures to a numpy array of zeros with num_arms elements.

1.3.2 get_reward

- If reward is 0, update failure for that arm else update success.

1.3.3 give_pull

- Iterate through each arm and find the max value obtained from beta distribution (using np.random.beta()) and return the max arm.

This graph is as expected as Thompson Sampling is tight and has logarithmic regret. Hence regret is lower when compared to the graph for UCB. It works better than KL_UCB as Thompson Sampling works better in practice.

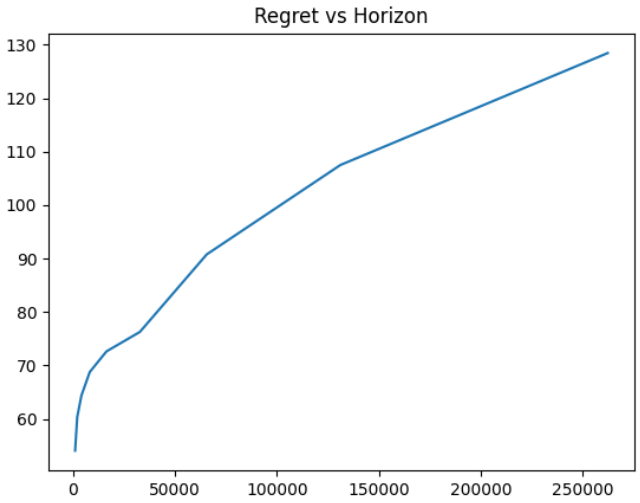

Figure 3: Thompson Sampling graph

2 Q2

Numpy seed is set to zero to produce constant result.

2.0.1 Initialize

- Initialize success and failures to a numpy array of zeros with num_arms elements along with the already defined parameters.

2.0.2 get_reward

- For each key in arm_rewards iterate through each value and update success and failure accordingly.

2.0.3 give_pull

- We use nested loops here. Iterate through batch_size outside. Keep a dictionary with keys as arm indices initialized to 0. Iterate through each arm and find the max value obtained from beta distribution (using np.random.beta()) and update the max arm in the dictionary by increment the value for that arm by 1.
- Convert this dictionary to a and b by iterating through the keys of the dictionary.

Figure 4: Thompson Sampling Q2

3 Q3

Numpy seed is set to zero to produce constant result. Perhaps the most challenging problem of the lot. Tried round robin, greedy epsilons and several variations of Thompson Sampling. The threshold 0.98 was arrived after much experimentation with the autograder.

3.0.1 Initialize

- Initialize success and failures to a numpy array of zeros with num_arms elements.
- Set a curr variable to 0.

3.0.2 get_reward

- Return curr. Curr holds the current returning index.

3.0.3 give_pull

- Update the success and fail arrays accordingly. Now we observe in the test cases that the number of arms is large hence a large max expected reward. So if the current current arm happens to have an empirical mean less than threshold the idea is to update curr with some random arm index.

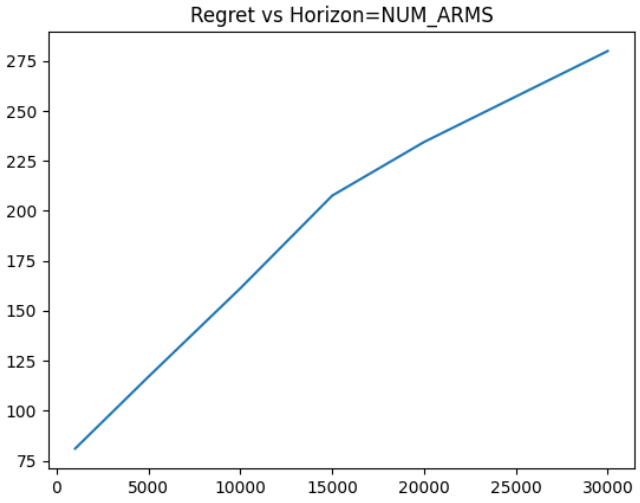
Regret vs Horizon=NUM_ARMS
275
250
225
200
175
150
125
100
75
0
5000
10000
15000
20000
25000
30000

Figure 5: Q3