CS747 Foundations of Intelligent and Learning Agents
Programming Assignment 1

Sanchit Jindal 200020120

Files Uploaded

I have uploaded the following files for my Assignment

```
200020120_Assigment1/
|------Report.pdf
|------task1.py
|------task2.py
|------task3.py
|------references.txt
```

1 Task 1

In This Task I have implemented 3 different Algorithms that were discussed and explained in class

1.1 UCB Algorithm

In A UCB ALgorithm, We calculate the upper confidence bound(UCB) for each arm and pull the arm with the maximum UCB. That means we select the arm for which the UCB is highest

$$UCB_a^t = p_a^t + \sqrt{\frac{2 * log(t)}{u_a^t}}$$

where p_a^t := empirical mean of the arm a at time t

u_a^t := number of time the arm a is pulled till time t

1.1.1 Constructor

```
self.success = np.zeros(num_arms)
self.pulls = np.zeros(num_arms)
self.ucb = np.zeros(num_arms)
self.t = 0
```

- The variable for the number of successful pulls, the total number of pulls, the UCB for each arm and the time step are initialized

1.1.2 get_reward

```
self.t += 1
self.pulls[arm_index] += 1
self.success[arm_index] += reward

if self.t < self.num_arms:
    return

empMean = self.success / self.pulls
exploration = np.sqrt(2 * math.log(self.t) / self.pulls)
self.ucb = empMean + exploration
```

- The TimeStep, number of pulls for the arm and the number of successes(reward = 1) are updated
- If all arms are not pulled then the UCB is not calculated and the function returns
- The empirical Mean and the exploration term are calculated using vector mathematics to speed up the process
- The UCB for is stored in the class variable

1.1.3 give_pull

```
if self.t < self.num_arms:
    return self.t

return np.argmax(self.ucb)
```

- In this if all the arms are not pulled pull the first no pulled arm
- otherwise pull the arm with the highest UCB value

1.1.4 Regret vs number of arms

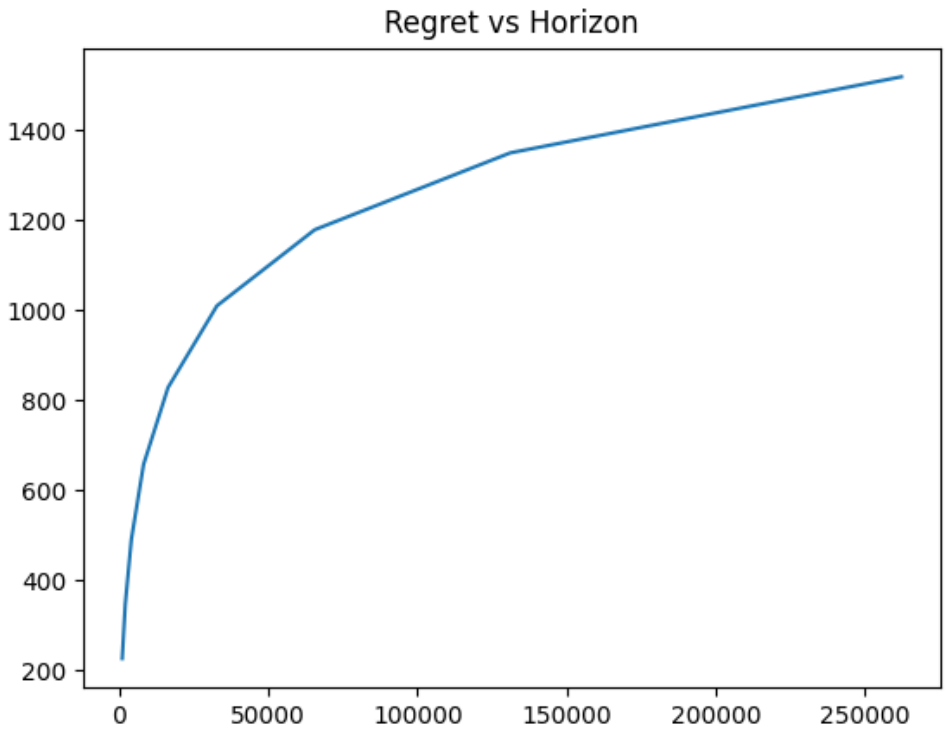

Figure 1: UCB Algorithm

1.1.5 Observations

- As we can see in the Plot of the Regret, it is sub linear, particularly it appears to be of logarithmic order
- The regret is quite high especially considering with respect to the other algorithms
- We need to vectorize the python code for the program to work efficiently
- The Output for the Test Cases Provided is

```
================== Task 1 ==================
Testcase 1
UCB               : PASSED. Regret: 37.44

Testcase 2
UCB               : PASSED. Regret: 367.56

Testcase 3
UCB               : PASSED. Regret: 449.52

Time elapsed: 3.68 seconds
```

1.2 KL-UCB algorithm

This is a upgrade over the UCB algorithm where we keep track of the KL-UCB of each arm and pull the one which has the highest value. That is we pull the arm at time t which has the highest KL-UCB

$$ucb - kl_a^t = max\{q \in [p_a^t, 1] \mid u_a^t KL(p_a^t, q) <= log(t) + c(log(log(t)))\}$$
$$\text{where } KL(p,q) = p * log(\frac{p}{q}) + (1-p) * log(\frac{1-p}{1-q})$$

1.2.1 Constructor

```
self.success = np.zeros(num_arms)
self.pulls = np.zeros(num_arms)
self.klucb = np.zeros(num_arms)
self.t = 0
self.c = 3
```

- The variable for the number of successful pulls, the total number of pulls, the kl-ucb for each arm, the time step and the coffecient of the log(log(.)) term are initialized.

1.2.2 get_reward

```
self.t += 1
self.pulls[arm_index] += 1
self.success[arm_index] += reward

if self.t < self.num_arms:
    return

empMean = self.success / self.pulls
term = (math.log(self.t) + self.c * math.log ( math.log( self.t))) / self.pulls

l = empMean.copy()
r = np.ones(self.num_arms)
for _ in range(16):
    m = (l + r) / 2
    KLValue = KL(empMean, m)
    l[KLValue < term] = m[KLValue < term]
    r[KLValue >= term] = m[KLValue >= term]

self.klucb = m
```

- The TimeStep, number of pulls for the arm and the number of successes(reward = 1) are updated
- If all arms are not pulled then the KL-UCB is not calculated and the function returns
- The empirical Mean and the exploration term are calculated using vectorization of the code
- The Value of KL-UCB is calculated using Binary search over the range $[mean, one]$, Again the process is vectorized to provide faster speeds

1.2.3 give_pull

```
if self.t < self.num_arms:
    return self.t

return np.argmax(self.klucb)
```

- In this function also, if there are arms that are not pulled once the first non pulled arm is returned
- Otherwise the arm with the highest KL-UCB is returned

1.2.4 Regret vs number of arms

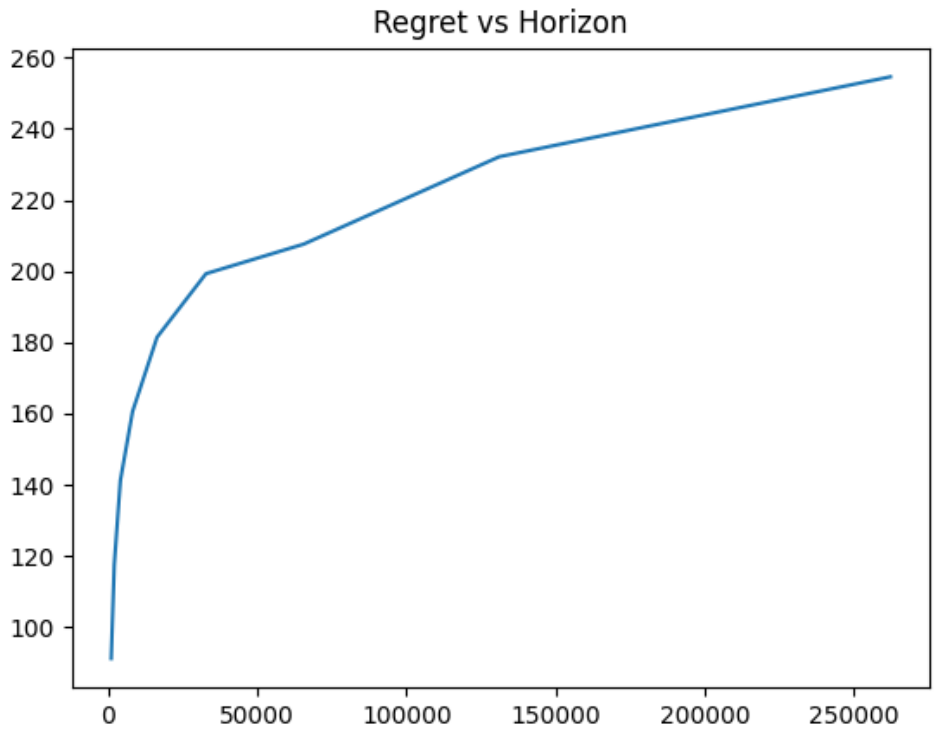

Figure 2: KL-UCB Algorithm

1.2.5 Observations

- We can see in the regret plot that the regret is still sub linear
- The value of regret also decreases substantially with respect to UCB algorithm, proving the fact that KL-UCB is a much tighter bound
- The Binary search Algorithm used to implement this causes a significant delay in the processing the data
- The output for the given test cases are

```
================== Task 1 ==================
Testcase 1
KL-UCB            : PASSED. Regret: 20.30

Testcase 2
KL-UCB            : PASSED. Regret: 130.95

Testcase 3
KL-UCB            : PASSED. Regret: 126.55

Time elapsed: 123.96 seconds
```

1.3 Thompson Sampling Algorithm

This is a stochastic Algorithm in which each arm is is associated with a Beta Distribution with parameters depending on the number of successes and failures on that arm and to select a arm to pull the distribution is sampled and the highest value is chosen

1.3.1 Constructor

```
self.success = np.zeros(num_arms)
self.failures = np.zeros(num_arms)
self.samples = np.zeros(num_arms)
```

- The variable for the number of successful pulls, the number of unsuccessful pulls and a vector to store the samples is initialized

1.3.2 get_reward

```
self.success[arm_index] += reward
self.failures[arm_index] += (1 - reward)

self.samples = [np.random.beta(i+1, j+1)
            for i,j in zip(self.success, self.failures)]
```

- The number of successes and failures for the arm are updated
- Using the beta function of numpy we sample each arm with parameters depending on the successes and failures of the arm
- The value is stored in the class variable

1.3.3 give_pull

```
return np.argmax(self.samples)
```

- The function will return the arm with the highest sample

1.3.4 Regret vs number of arms

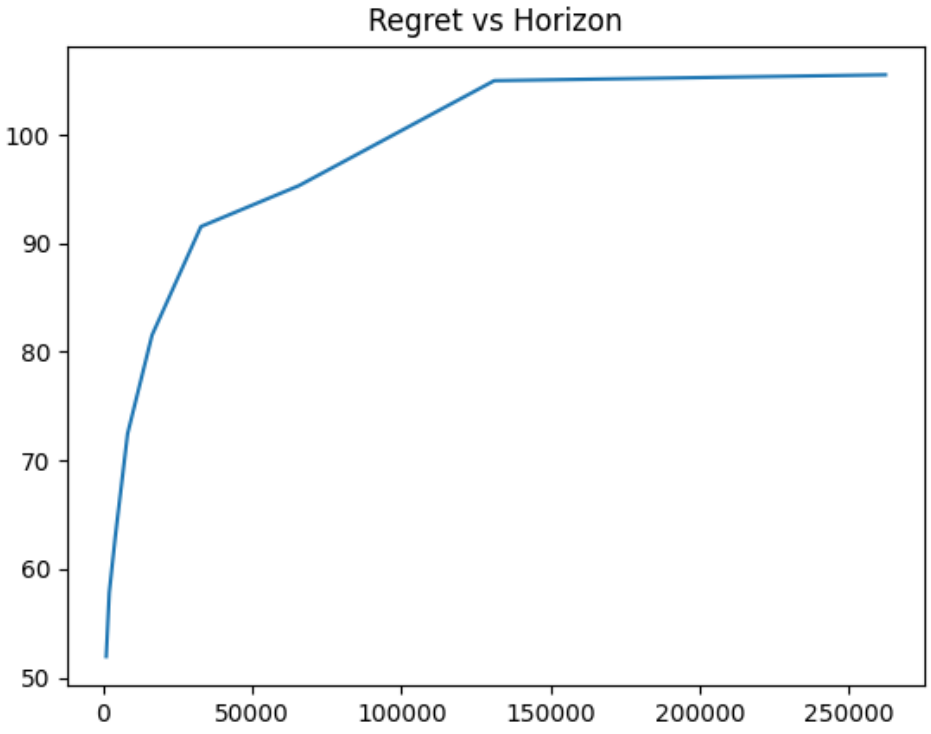

Figure 3: Thompson Sampling Algorithm

1.3.5 Observations

- The Plot for the regret is again sub linear
- We can observe that the value of regret is still lower than KL-UCB
- This is a stochastic process so the process might benefit from running multiple times to get the best output
- The output for the given test cases for the given seed is

```
 ================== Task 1 ==================
Testcase 1
Thompson Sampling : PASSED. Regret: 10.24

Testcase 2
Thompson Sampling : PASSED. Regret: 67.90

Testcase 3
Thompson Sampling : PASSED. Regret: 64.34

Time elapsed: 12.91 seconds
```

2 Task 2 - Batched Bandit Algorithm

In this task We have to implement a Batched Bandit Problem, Which Means that we will give a number of arms to pull and then we will be given the reward for all the arms together,

In my Implementation I will be using the Thompson Sampling algorithm for the Batches, as it is giving the best results for the single arm case

2.1 Constructor

```
self.success = np.zeros(num_arms)
self.failures = np.zeros(num_arms)
self.samples = np.zeros(num_arms)
```

- The variable for the number of successful pulls, the number of unsuccessful pulls and a vector to store the samples is initialized

2.2 get_reward

```
for key in arm_rewards:
    rewards = arm_rewards[key]
    for r in rewards:
        self.success[key] += r
        self.failures[key] += (1 - r)
```

- In this function we just iterate through the Dictionary of rewards and update the number of successes and failures for the arms we pulled

2.3 give_pull

```
numPulls = [0]*self.num_arms

for _ in range(self.batch_size):
    self.samples = [np.random.beta(i+1, j+1)
                for i,j in zip(self.success, self.failures)]
    chosen  = np.argmax(self.samples)
    numPulls[chosen] += 1

return range(self.num_arms), numPulls
```

- This function iterates over the batch size
- It samples each arm once for each batch and selects the maximum of those samples to select the arms
- It then returns the number of times each arm is to be pulled

2.4 Regret vs number of arms

Figure 4: Batched Thompson Sampling Algorithm

2.5 Observations

- The Batched Sampling Algorithms is a restriction on the Original Sampling problem and so we can infer that the algorithm will perform worst on the general cases
- From the plot we can observe that the regret is almost linear with respect to the number of arms The Algorithm is again a stochastic one, and can give better results on running multiple times
- The Output for the test cases provided are

```
================== Task 2 ==================
Testcase 1
Batched Algorithm: PASSED. Regret: 60.60

Testcase 2
Batched Algorithm: PASSED. Regret: 79.84

Testcase 3
Batched Algorithm: PASSED. Regret: 118.66

Time elapsed: 25.46 seconds
```

3 Task3 - Low Horizon / Many Arms Problem

In this problem of the bandit problem, we do not have enough pulls to converge to the best possible arm, So we need to be efficient in the arms we pick

I am implementing the algorithm that we pull the arm until we get a failure, and then we start pulling the next arm, If we pull the same arm $\sqrt{n}$ times then we consider the best arm we will find and keep pulling it for the rest of the time steps

I am referencing the algorithm that is presented in the paper `https://repository.upenn.edu/cgi/viewcontent.cgi?article=1291&context=statistics_papers`

3.1 Constructor

```
self.run = 0
self.locked = 0
self.arm = 0
```

- The variable for the number of times the current arm is pulled, whether the arm is locked or not , and the index of the arm is initialized

3.2 get_reward

```
if reward == 1:
    self.run += 1
    if self.run > np.sqrt(self.num_arms):
        self.locked = True

else:
    if not self.locked:
        self.arm += 1
        self.run = 0
```

- If the arm we pull gave a reward of one then increase the run, and if run becomes higher than our limit of $\sqrt{n}$ then lock that arm
- If we get a failure then we increase the arm index and reset the run count

3.3 give_pull

```
return self.arm
```

- We return the current arm that we are pulling

3.4 Regret vs number of arms

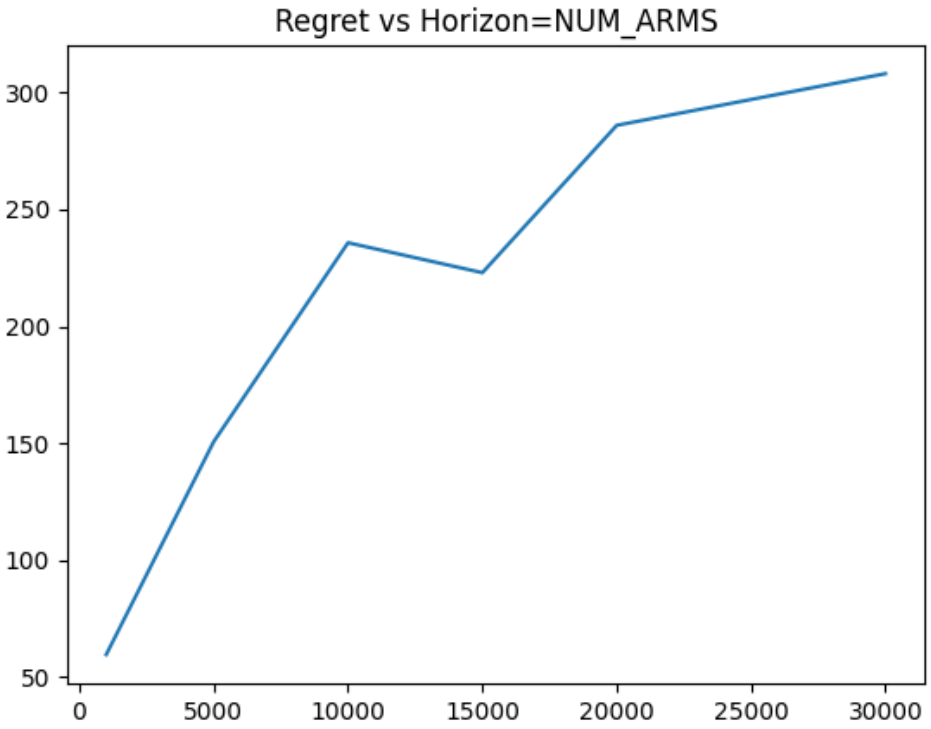

Figure 5: Multiple Arms Algorithm

3.5 Observations

- The Problem is a difficult one then the rest of them as we are never able to check each arm, even less sure that we are pulling the most optimal arm
- Best we can do in this case is try to pull an arm that is better than most of the other arms that is try to pull an arm that has a high probability of producing rewards
- The Algorithm to fix on an arm that has already given a number of rewards more than the limit is a good one as we are setting a expectation that we have with the arm we can pull

- A possible improvement over the algorithm might be to pull other arms even if we have a feasible arm if we have a number of pulls left
- The Output of the given test cases is

```
================== Task 3 ==================
Testcase 1
Many Arms Algorithm: PASSED. Regret: 150.62

Testcase 2
Many Arms Algorithm: PASSED. Regret: 235.78

Testcase 3
Many Arms Algorithm: PASSED. Regret: 222.96

Time elapsed: 3.65 seconds
```