CS747 Foundations of Intelligent and Learning Agents

Multi-Armed Bandits

Assignment 1

Param Rathour | 190070049

Autumn Semester 2021-22

Contents

0 Introduction

In this assignment, I have implemented and compared different algorithms for sampling the arms of a stochastic multi-armed bandit, ε-greedy exploration, UCB, KL-UCB, and Thompson Sampling for Task 1, optimisd the scaling coefficient of the exploration bonus in UCB for Task 2. In Task 3, I developed a efficient a regret-minimisation algorithm for non Bernoulli bandit instances. In Task 4, I maximised the expected number of times the reward obtained exceeds a given threshold for same bandit instances as in task 3.

0.1 Basic Code Structure

First, I start with argument parsing and initialising variables using `argparse`.

- `probabilities` for each arm, stores probabilities of reward (in case of multiple rewards it is list of list)
- `armData` for each arm, stores total number of times the arm was pulled as it's first entry and number of times rewards are pulled as subsequent entries.

I recognised that the general structure of each algorithm is similar. We start with initializing pulls then iteratively choosing a arm and update it's results till we reach horizon and then finally, calculate regret. This is beautifully done by the following function that I will be using for all tasks

```
def task(probabilities, rs, ep, c, th, hz):
    currentReward = 0
    armData = np.zeros((numberOfArms,1 + len(rewards)), dtype = int)
    np.random.seed(rs)
    for arm in range(min(numberOfArms, hz)):
        currentReward
            = updateRecords(th, probabilities, armData, arm, currentReward, algo, rewards)
    for i in range(numberOfArms, hz):
        armChosen = chooseArm(ep, c, i, armData, numberOfArms, rewards)
        currentReward
            = updateRecords(th, probabilities, armData, armChosen, currentReward, algo, rewards)
    return currentReward
```

```
def updateRecords(th, probabilities, armData, armChosen, currentReward, algo, rewards):
    armData[armChosen][0] += 1
    sample = algoToSamplingMapping[al](th, probabilities, armChosen, currentReward, rewards)
    if algo == 5:
        armData[armChosen][1 + int(np.argwhere(rewards == sample))] += 1
    else:
        armData[armChosen][1] += sample
    return currentReward + sample
def chooseArm(ep, c, t, armData, numberOfArms, rewards):
    return algoToChoosingArmMapping[al](ep, c, t, armData, numberOfArms, rewards)
```

This will automatically call sampling and choosing function according to algo

```
algoToSamplingMapping = {"epsilon-greedy-t1" : bernoulliSamples, "ucb-t1" : bernoulliSamples,
    "kl-ucb-t1" : bernoulliSamples, "thompson-sampling-t1" : bernoulliSamples,
    "ucb-t2" : bernoulliSamples, "alg-t3" : weightedSamples,
    "alg-t4" : weightedSampleswithThreshold}
algoToChoosingArmMapping = {"epsilon-greedy-t1" : epsilonGreedyChoosingArm,
    "ucb-t1" : UCBChoosingArm, "kl-ucb-t1" : KLUCBChoosingArm,
    "thompson-sampling-t1" : thompsonChoosingArm, "ucb-t2" : UCBChoosingArm,
    "alg-t3" : thompsonChoosingArmGeneralized, "alg-t4" : thompsonChoosingArm}
```

0.2 Assumptions

- Initialization is done by pulling each arm once and updating it's results.
- Ties are broken by choosing the arm with lowest index.
- All the algorithms are initialised by random seed at the beginning

1 Task 1 - Implementation of Fundamental Algorithms

Reward from chosen arm is done by taking Bernoulli Samples from probabilities of arms.

I have used `random.binomial` from `numpy` library to get these rewards.

```
def bernoulliSamples(th, probabilities, armChosen, currentReward, rewards):
    return np.random.binomial(1, probabilities[armChosen])
```

Also the regret is defined as for task 1 and 2

$$\text{regret} = \text{bestExpectedReward} \cdot \text{Horizons} - \text{reward}$$

Where,

```
bestExpectedReward = max(probabilities)
```

1.1 ε Greedy

I have implemented ε-greedy 3 for this task. ε Greedy achieves linear regret.

Below is code for choosing optimal arm.

```
def epsilonGreedyChoosingArm(ep, c, t, armData, numberOfArms, rewards):
    if np.random.random() < ep:
        return np.random.randint(0, numberOfArms)
    else:
        pat = [armData[arm][1] / armData[arm][0] for arm in range(numberOfArms)]
        return np.argmax(pat)
```

1.2 UCB

UCB achieves sublinear regret but the regret is not optimal.

$$\text{ucb}_a^t = \hat{p}_a^t + \sqrt{\frac{2\ln(t)}{u_a^t}}$$

Below is code for choosing optimal arm (it takes input c value which is 2 by default).

```
def UCBChoosingArm(ep, c, t, armData, numberOfArms, rewards):
    return np.argmax([(armData[arm][1] / armData[arm][0])
                      + math.sqrt(c*math.log(t)/ armData[arm][0]) for arm in range(numberOfArms)])
```

1.3 KL-UCB

KL-UCB achieves optimal regret but is slower due to solving an optimisation problem for each time instant.

$$\text{ucb-kl}_a^t = \max\{q \in [\hat{p}_a^t, 1] \text{ such that}\}$$

$$u_a^t KL(\hat{p}_a^t, q) \leq \ln(t) + c\ln(\ln(t)) \text{ where } c \geq 3$$

I have assumed $c = 3$.

I have used Binary Search to get optimal q value with threshold 10^{-4}. The way I implemented my binary search, it only goes in *forward* direction and takes appropriate jumps for q, starting from $q = \hat{p}_a^t$. Also as $\ln(0), 0 \cdot \ln(0), 0 * \ln(0/0)$ are encountered while calculating KL-divergence. I defined it in such a way that binary search of q still makes sense. For eg. if $\hat{p}_a^t = 0$, $q = \hat{p}_a^t = 0$, now q can only increase, so it makes sense to proceed forward in binary search, when $\hat{p}_a^t = 1$, $q = \hat{p}_a^t = 1$, q can only decrease, so binary search can't proceed forward and it has to stop.

Below is code for choosing optimal arm.

```
def KL(p,q):
    if p == 0:
        return 0
```

```
    elif q == 0 or p == 1 or q == 1:
        return float('inf')
    else:
        return p*math.log(p/q) + (1-p)*math.log((1-p)/(1-q))
def KLmaxUCB(c,t):
    return math.log(t) + c*math.log(math.log(t))
def KLupperConfidenceBound(p, u, c, t):
    maxBound = KLmaxUCB(c,t)/u
    q = p
    b = (1 - p) / 2
    while b > 1e-4:
        if KL(p,q+b) <= maxBound:
            q += b
        b /= 2;
    return q
def KLUCBChoosingArm(ep, c, t, armData, numberOfArms, rewards):
    return np.argmax(
             [KLupperConfidenceBound(armData[arm][1] / armData[arm][0], armData[arm][0], 3, t)
             for arm in range(numberOfArms)])
```

1.4 Thompson Sampling

Thompson Sampling achieves optimal regret and is faster than KL-UCB.

$$x_a^t \sim \text{Beta}(s_a^t + 1, f_a^t + 1)$$

Below is code for choosing optimal arm.

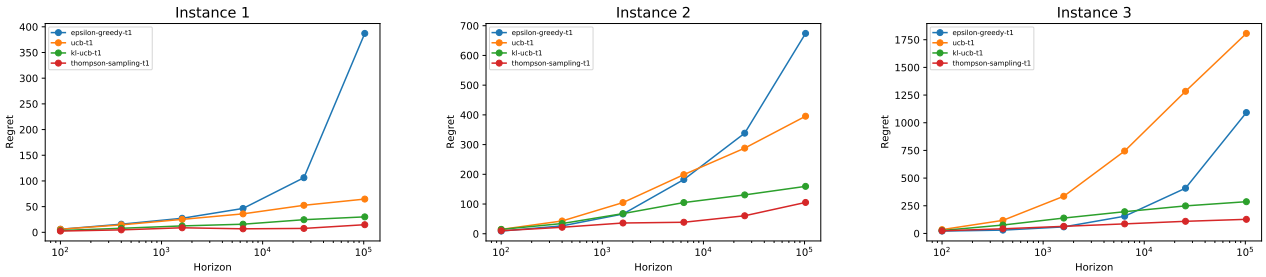
I have used `random.beta` from `numpy` library to get these samples.

```
def thompsonChoosingArm(ep, c, t, armData, numberOfArms, rewards):
    return np.argmax([np.random.beta(armData[arm][1] + 1, armData[arm][0] - armData[arm][1] + 1)
                      for arm in range(numberOfArms)])
```

1.5 Observations

Here, I have generated data from 3 instances for ε-greedy-t1 with ε set to 0.02; ucb-t1, kl-ucb-t1, thompson-sampling-t1 and horizon values from 100; 400; 1600; 6400; 25600; 102400. Then I have averaged regret values by running for random seeds from 0; 1;...; 49.

As can be seen from Figure 1, Thompon Sampling has the least regret while, ε-greedy had the maximum regret for first two instances and UCB[1] in third instance.



(a) Instance 1 (b) Instance 2 (c) Instance 3

Figure 1: Comparison of ε-greedy exploration, UCB, KL-UCB, Thompson Sampling

[1] surprising!

2 Task 2 - Optmising the scaling coefficient of the exploration in UCB

Because of the my implementation of UCB with scale as a parameter, I did not have to write additional functions for this task, just giving scale value via argument parsing worked.

Reward from chosen arm is again done by taking Bernoulli Samples from probabilities of arms.

I have used `random.binomial` from `numpy` library to get these rewards.

```
def bernoulliSamples(th, probabilities, armChosen, currentReward, rewards):
    return np.random.binomial(1, probabilities[armChosen])
```

Below is code for choosing optimal arm (c is a parameter).

```
def UCBChoosingArm(ep, c, t, armData, numberOfArms, rewards):
    return np.argmax([(armData[arm][1] / armData[arm][0])
                      + math.sqrt(c*math.log(t)/ armData[arm][0]) for arm in range(numberOfArms)])
```

2.1 Observations

Here, I have generated data from 5 instances for ucb-t2, scale from 0.02; 0.04; 0.06; ...; 0.3 and horizon values from 100; 400; 1600; 6400; 25600; 102400. Then I have averaged regret values by running for random seeds from 0; 1;...; 49.

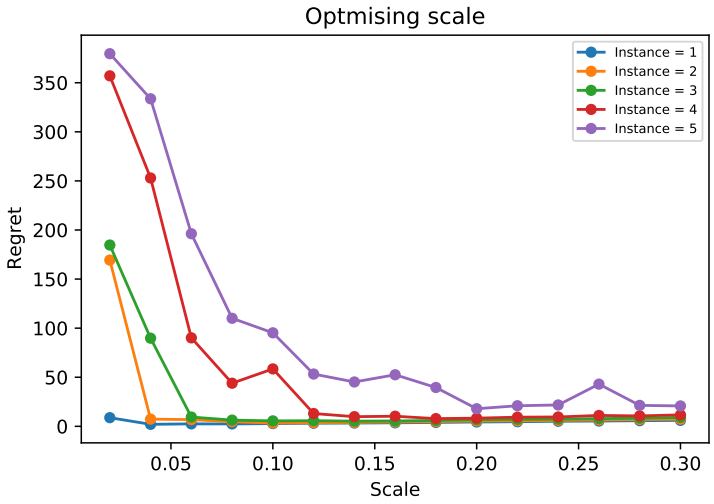


Figure 2: Optmising the scaling coefficient of the exploration bonus in UCB

As can be seen from Figure 2, the regret decreases and then increases a little as scale increases. The values for c which minimises regret are $0.02, 0.08, 0.12, 0.16, 0.18$ for instances 1,2,3,4,5 respectively. First some exploration is necessary to find optmial arm and reduce regret but after a value of c, exploration will be more and it will start choosing unoptimal arms more than necessary thus increasing regret.

A very interesting observation is that for latter instances the regret is higher. This is because for higher instances the probabilities of 2 bandit are closer[2], so the algorithm will need more *exploration* (hence greater scale) to distinguish both arms.

[2] As instances increases, the difference in probabilities goes from 0.5 to 0.1

3 Task 3 - Minimise Regret for non Bernoulli Bandit Instance

Reward from chosen arm is done by taking weighted samples from probabilities distiribution of rewards for each arm. I have used `random.choice` from `numpy` library to get these rewards.

```
def weightedSamples(th, probabilities, armChosen, currentReward, rewards):
    return np.random.choice(rewards, p = probabilities[armChosen])
```

I thought a lot about choosing the optimal arm, first I intuitively tried the thompson sampling with just successes and failure, but here success was defined as a float and failure was $u_a^t -$ successes. I also tried KL-UCB with similar definition of successes. The results were good, with thompson just beating KL-UCB. But I knew, this wasn't true generalisation to the problem as successes and failures are difficult to define.

Then it struck me to use generalisation of beta function itself, and I found dirichlet. The only issue remaining was that my rewards and number of time a reward was achieved both has to be used to get maximal value. This coupled with the fact that `random.dirichlet` of `numpy` returned k (=number of rewards) values and not one, I thought of taking dot product of rewards with these k values. This algorithm worked better than what I implemented earlier. I looked up and verifed my approach from ICALT. Below is code for choosing optimal arm.

```
def thompsonChoosingArmGeneralized(ep, c, t, armData, numberOfArms, rewards):
    generalizedSamples = [np.random.dirichlet(armData[arm][1:]+1) for arm in range(numberOfArms)]
    return np.argmax([np.dot(generalizedSamples[arm], rewards) for arm in range(numberOfArms)])
```

Also the regret is defined as

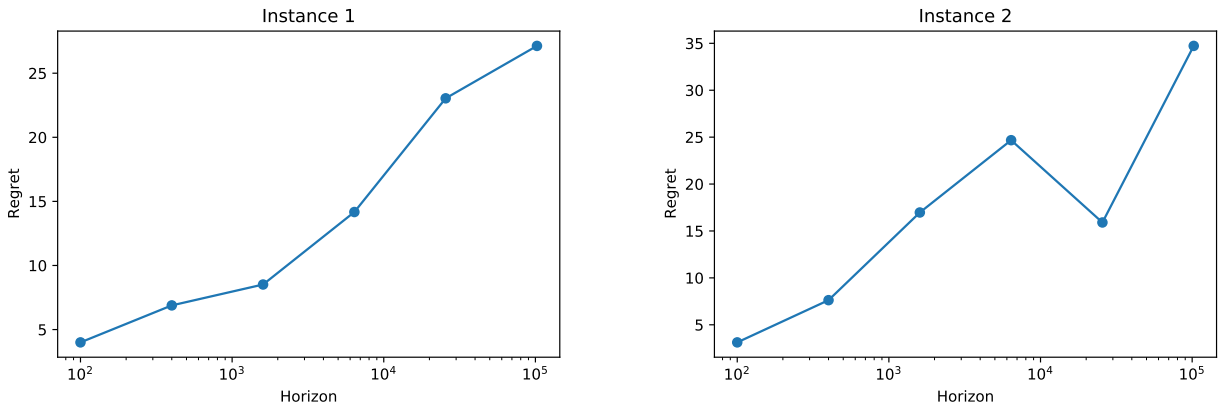
$$\text{regret} = \text{bestExpectedReward} \cdot \text{Horizons} - \text{reward}$$

Where,

```
bestExpectedReward = max([np.dot(rewards, probabilities[i]) for i in range(numberOfArms)])
```

3.1 Observations

Here, I have generated data from 2 instances for alg-t3 and horizon values from 100; 400; 1600; 6400; 25600; 102400. Then I have averaged regret values by running for random seeds from 0; 1;...; 49.



(a) Instance 1

(b) Instance 2

Figure 3: Minimise Regret for non Bernoulli Bandit Instance

As can be seen from Figure 3, the performance of algorithm is great with sublinear regret (from straight line).

4 Task 4 - Maximising HIGHS for a non Bernoulli bandit instance

Because of the my implementation, I did not have to write additional functions. I only had to calculate expected MAX-HIGHS which will be used to calculate regret Here, the difficulty I was facing in task 3 of defining successes vanished. The HIGHS easily became success, when sampled reward is greater than threshold. So thompson-sampling-t1 was used to do this task.

$$\text{regret} = \text{MAX-HIGHS} \cdot \text{Horizons} - \text{HIGHS}$$

Where,

```
MAXHIGHS = max([sum([probabilities[arm][i] for i in indices]) for arm in range(numberOfArms)])
```

Reward from chosen arm is done by taking weighted samples from probabilities distiribution of rewards for each arm. I have used `random.choice` from `numpy` library to get these rewards. The difference between this and the one used in task 3 is that it also compares with threshold. So in the end the reward is just HIGH or LOW

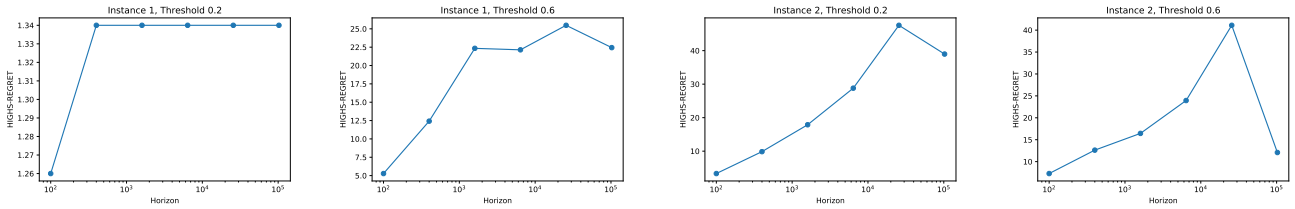
```
def weightedSampleswithThreshold(th, probabilities, armChosen, currentReward, rewards):
    return 1 if (np.random.choice(rewards, p = probabilities[armChosen]) > th) else 0
```

Below is code for choosing optimal arm.

```
def thompsonChoosingArmGeneralized(ep, c, t, armData, numberOfArms, rewards):
    generalizedSamples = [np.random.dirichlet(armData[arm][1:]+1) for arm in range(numberOfArms)]
    return np.argmax([np.dot(generalizedSamples[arm], rewards) for arm in range(numberOfArms)])
```

4.1 Observations

Here, I have generated data from 2 instances for alg-t4, threshold values 0.2; 0.6 and horizon values from 100; 400; 1600; 6400; 25600; 102400. Then I have averaged regret values by running for random seeds from 0; 1;...; 49.



(a) Instance 1, Horizon = 0.2 (b) Instance 1, Horizon = 0.6 (c) Instance 2, Horizon = 0.2 (d) Instance 2, Horizon = 0.6

Figure 4: Maximising HIGHS for a non Bernoulli bandit instance

As can be seen from Figure 4, the performance of algorithm is great with sublinear regret (from straight line nature). Increasing threshold will reduce the frequency of HIGHS, hence regret increases.

In instance 1, for arm 1 output is always HIGH when threshold is 0.2, hence the optimal arm is found faster we get very low regret.

In instance 2, for arm 2 output is 0.25 with probability 0.41 hence when threshold is 0.2, the expected MAX-HIGHS difference between 2 arms is just 0.04, so it will take more time to find optimal arm and when threshold is 0.6, arm 1 clearly arm 2 both probabilities are a bith farther with difference 0.3.

5 Conclusion

It was a pleasure doing this assignment. I learnt a lot about Multi-Armed Bandits and Python overall. Thanks.