Probability and Mathematical Statistics: Final Project

Due on December 31, 2022 at 11:59am

Name: **Ren Hui, Wanchen Su**
Student ID: 2021533089, 2021533067

Problem 1 (Part 1: classical Bandit Algorithms)

Solution

1,2. See jupyter notebook.

3. Accorrding to python simulation, the results are:
results for ϵ-greedy:

	$\epsilon = 0.1$	$\epsilon = 0.5$	$\epsilon = 0.9$
reward	3415.895	3083.32	2748.985
θ	[0.699565 0.501675 0.398460]	[0.700644 0.496887 0.398588]	[0.700090 0.499850 0.398113]

results for UCB:

	c = 0.1	c = 0.5	c = 0.9
reward	3410.44	2974.035	2824.62
θ	[0.700882 0.492749 0.384489]	[0.700674 0.499007 0.398861]	[0.701356 0.498778 0.400231]

results for TS:

	a,b = [[1, 1], [1, 1], [1, 1]]	a,b = [[601, 401], [401, 601], [2, 3]]
reward	3481.33	3490.945
θ	[0.6995964 0.45887541 0.37554019]	[0.68296885 0.4001996 0.362563]

4. Oracle value:

$$E(r) = 5000 \times 0.7 = 3500$$

gap of ϵ-greedy:

	$\epsilon = 0.1$	$\epsilon = 0.5$	$\epsilon = 0.9$
reward	3415.895	3083.32	2748.985
gap	84.105	416.68	751.015

gap of UCB:

	c = 0.1	c = 0.5	c = 0.9
reward	3410.44	2974.035	2824.62
gap	89.56	525.965	675.38

gap of TS:

	a,b = [[1, 1], [1, 1], [1, 1]]	a,b = [[601, 401], [401, 601], [2, 3]]
reward	3481.33	3490.945
gap	18.67	9.905

As is shown above, with the given parameters, TS Algorithm is the best.

In the ϵ-greedy algorithm, ϵ decides the probability of choosing the max estimate of all the arms or choosing a random arm. Which means the larger ϵ is, the more evenly spread are the tests.
In the UCB algorithm, the parameter c balances the estimation of θ_j, and considers the number of trials on that arm. Increasing c means considering more about the number of test on that arm, which will alow the decision maker(DM) choose some of the less experimented arms. This helps in instances where the number of tests on a arm is quite small and therefore the θ of that arm is way less than it

should be.

If the initial value of α_j, β_j we passed in is too small, then the result of the first few tests may influence the final result greatly, and only a few dozen tests are on the second and third arm. If this happens, the estimate value of θ will be rather inaccurate. At the same time, the aggregated reward will be quite large because the DM spent very little time exploring. On the other hand, if the initial value of α_j, β_j we passed in is relativly large, this value of a and b may influence the final result greatly. As in the trial with parameter [[601, 401], [401, 601], [2, 3]], we can see that the final estimation of arm two is very close to $\frac{401}{401+601}$. In this case, the parameter we passed in is correct in which is the arm with the largest oracle value, so the aggregated reward is quite large. However, if the parameter passed in is contradictory to the actual oracle, it will take quite a long time to adjust. Consequently the aggregated reward is smaller.

More impacts of ϵ, c and α_j, β_j concerning exploration-exploitation will be disccussed in 5.

5. In the case of all algorithms, more exploitation means that the total reward we get is larger, but if the estimate of each arm differs too much from the oracle value, after all tests, the total aggregated reward becomes smaller. More exploration means the estimated value of the reward of each arm is more accurate, it helps the decision maker choose the arm better, but the exploration means that the rewards during exploration is less.

   For the ϵ-greedy algorithm, the larger ϵ is, there is more chance of exploration than exploitation. So while ϵ grows, the gaps between the algorithm estimate and the oracle value decreases, but the sum of the reward of all trials is smaller because the DM spent too much time exploring when it is not that necessary.

   As for the UCB Algorithm, the exploration-exploitation trade-off depends on the value of c. Similar to the ϵ-greedy algorithm, the larger c is, there is more exploration and less exploitation. As can be deduced from the data in the python simulation, for larger c, the estimated value for the third arm is more accurate. This means that there are more times when the decision maker choose arm c. Consequently, the aggregated rewards over N time slots are less.

   The TS Algorithm is rather different from the previous ones. Its exploration is very limited. The arm selected is always the arm with the greatest current reward estimation. So arms with small oracle values may end up being pulled only a few times. Which makes the estimate value of arms 2 and 3 rather inaccurate. But the aggregated reward gained over the N trials should be the largest of all the algorithms. TS performs best in this case because the parameters a and b we passed in corresponds with the real oracle value. On the other hand, as the example in 5, if the parameters passed in is inaccurate, it could have a large influence on the aggregated reward. This is a default for having too little exploration.

6. We searched the internet for more information on depend arm bandit problem, and used the thought of the UCB-D algorithm to improve the classical bandit algorithms. With programming and testing, we found that using UCB for choosing the cluster and using TS to choose the specific arm can give a result better than the original TS, which has the best performance in the three classical bandit algorithms. The algorithm we used is shown on the next page.

Problem 1 continued on next page...

Initialize: Beta parameter(α_j,β_j),$j \in 1, 2, 3$
$count(C(j)) \leftarrow$ sum of all α, β in the cluster.
$\theta(C(j)) \leftarrow$ sum of all α_j in the cluster/$count(C(j))$
for $t = 1, 2, \ldots, N$ **do**
  #sellect cluseter(using thoughts from UCB algorithm):
  $C(t) \leftarrow argmax(\theta(C(j)) + c * \sqrt{\frac{2log(t)}{count(C(j))}})$
  #sellect and pull arm
  for $j \in choosencluster$ **do**
    Sample $\theta(j)$ ~Beta(α_j, β_j)
  end
  I(t)$\leftarrow$ arg max$\theta(j)$
  #update distribution and upper bound θ of cluster:
  $\alpha_{I(t)} \rightarrow \alpha_{I(t)} + r_{I(t)}$
  $\beta_{I(t)} \rightarrow \beta_{I(t)} + 1 - r_{I(t)}$
  $count(C(t)) \leftarrow count(C(t)) + 1$
  $\theta(C(t)) \leftarrow \theta(C(t)) + \frac{1}{count(C(t))[r_{I(t)} - \theta(C(t))]}$
end

Algorithm 1: Dependent TS

Problem 2 (Part 2: Bayesian Bandit Algorithms)

Solution

1. See jupyter notebook for the simulation proccess. We simulated with the oracle value :
simulations:

parameter	θ_1	θ_2	λ
test 1	0.7	0.3	0.9
test 2	0.4	0.5	0.9
test 3	0.1	0.	0.9

Let $r(n)$ be the reward over n pulls. **simulations:**

result	always choose best $E(r(25))$	simulation $r(25)$	always choose worst $E(r(n))$
test 1	6.497471408615705	5.758137590942908	2.784630603692444
test 2	4.641051006154074	4.247677424676706	3.71284080492326
test 3	2.784630603692444	2.2136432344092114	0.928210201230815

2. If both θ_1 and θ_2 are relativly large, then we could end up with always pulling the first arm that succeeded. For example, if $\theta_1 = 0.8$, $\theta_2 = 0.9$, and the first arm we pulled is the first one, then there is a large possibility that the DM will always pull arm 1. We used 0.8 and 0.9 as the oracle value and asked the python simulation to print out the arm it chooses each time, and it shows clearly that it almost always sticks with the arm it chose in the first round. In fact, as it turns out, if the two arms have close possibility of giving a reward, this policy tend to behave not that well.

3. Story proof:
Let us be the decision maker(DM) of a two-armed bandit problem. Presume that the experiment lasts forever, but the reward returned each time is reduced by multiplying λ, where $0 < \lambda < 1$. Put in mind that we as DM, do not know the actual success probability of each arm. So let R$(\alpha_1, \beta_1, \alpha_2, \beta_2)$represent the expected reward given the previous success and failures. So accorrding to LOTUS, the expected reward of pulling the first arm is:

$$R_1(\alpha_1, \beta_1) = P(success|\alpha_1, \beta_1) \times \text{(reward if succeed)} + P(failure|\alpha_1, \beta_1) \times \text{(reward if failure)}$$

Which is:

$$R_1(\alpha_1, \beta_1) = \frac{\alpha_1}{\alpha_1 + \alpha_2}(1 + \lambda R(\alpha_1 + 1, \beta_1, \alpha_2, \beta_2)) + \frac{\alpha_2}{\alpha_1 + \alpha_2}(\lambda R(\alpha_1, \beta_1 + 1, \alpha_2, \beta_2))$$

R_2 is similar, the only difference is in the case of arm 2 we should change α_2 and β_2 accorrdingly. Since we will always make the best desision at the moment, $R(\alpha_1, \beta_1, \alpha_2, \beta_2) = maxR_1(\alpha_1, \beta_1), R_2(\alpha_2, \beta_2)$. Therefore, the given recurrence equation holds.

4. The above recurrence equation can be solved by using dynamic programming in python simulation. We only need to type in the equation and manually set a ending condition, such as count the number of times it has been called repeatedly, and if it exceeds a certain n, then the function will return $\frac{\alpha}{\alpha+\beta}$.

5. The optimal policy will be to alter the algorithm in 4 slightly, so that after solving the equation for each decision, the DM will do as is caculated to be the best decision. Then for the next decision the DM will solve the above recurrence equation again.