Probability and Mathematical Statistics Project: Performance Evaluation of Bandit Algorithms

2023 January 16th

Name: **Li Chaofan** & **Qin Yao**
Student ID: 2021531045 and 2021522016

Abstract

The bandit problem for a gambler is to decide which arm to pull to maximize his total reward in a series of trials. Many real-world learning and optimization problems can be modeled in this way. Several strategies or algorithms have been proposed as a solution to this problem. This paper performs several classical Bandit Algorithms and provides a preliminary empirical evaluation of them. One remarkable outcome of our experiments is that the TS Algorithm performs the best. The algorithms for dependent case are been designed. Simulation of Bayesian Bandit Algorithms and new optimal algorithm are also designed.

Introduction

In many real-world situations, decisions are made in order to maximize some expected numerical reward. But decisions do not just bring in more reward, they can also help discover new knowledge that could be used to improve future decisions.Many questions will arise in many fields which are related to the problem of balancing reward maximization based on the knowledge already acquired and attempting new actions to further increase knowledge, which is known as the exploitation vs. exploration tradeoff in reinforcement learning.

Imagine you go to a casino in Las Vegas,and there are bandit machine with three arms.You bought some credits and can pull any arm of the bandit.At each time step $t = 1, \cdots, T$, you pull arm $j \in \{1, 2, 3\}$ and observe a random reward.Your goal is to maximize your total reward as much as possible.So the problem is that at each time pull,how do I decide which arm to pull based on the past history of rewards.

We make a simplifying assumption that each arm is independent of the rest,and has some reward distribution which does not change over time. The non-triviality of the multi-armed bandit problem lies in the fact that we cannot access the true bandit probability distributions — all learning is carried out via the means of trial-and-error and value estimation.

This bandit problem allows us to formally model this tradeoff between:

- **Exploitation**: Pulling arm(j) we know to be "good".
- **Exploration**: Pulling less-frequently pulled arms in the hopes they are also "good" or even better

In this report, we compare three classical algorithms which can solve this problem: ϵ-greedy Algorithm, UCB (Upper Confidence Bound) Algorithm and TS (Thompson Sampling) Algorithm.

Part I

Classical Bandit Algorithms

Problem 1

Now suppose we obtain the parameters of the Bernoulli distributions from an oracle, which are shown in the following table. Choose N = 5000 and compute the theoretically maximized expectation of aggregate rewards over N time slots.

We call it the oracle value. Note that these parameters θ_j ,$j \in \{1, 2, 3\}$and oracle values are unknown to all bandit algorithms.

Arm j	1	2	3
θ_j	0.7	0.5	0.4

The theoretical maximum expectation of the total reward in the theoretical maximum N time periods is when pulling the machine with the maximum probability all the time.

$$\max_{I(t),t=1\cdots,N} \mathbb{E}[\sum_{t=1}^{N} r_{I(t)}] = N \cdot E[r_1] = N \cdot \theta_1 = 5000 \cdot 0.7 = 3500$$

Problem 2

import and reward function

```
import numpy as npy
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import random
import copy
def reward(arg):
    theta=[0.7,0.5,0.4]
    if npy.random.rand(1)>theta[arg]:
        return 0

    else:
        return 1
```

ε-greedy algorithms

```
def greedy(varepsilon,times):
    rewards = npy.zeros(times)
    for i in range(times):
        N=5000

        theta = [0,0,0]
        count = [0,0,0]
        for t in range(N):
            if theta[0]==theta[1]:
                if theta[1]==theta[2]:
                    It = npy.random.randint(0, 3)
                else:
                    It = npy.random.randint(0, 2)
            elif theta[1]==theta[2]:
                It = npy.random.randint(1, 2)
            elif theta[0]==theta[2]:
                It = random.choice([0,2])
            elif npy.random.rand(1) >varepsilon:
                It = npy.argmax(theta)
            else:
                It = npy.random.randint(0, 3)
            count[It]+=1

            a=1/(count[It])*(rit-theta[It])
            theta[It]=theta[It]+a
    return npy.average(rewards)
print(greedy(0.1,200))
print(greedy(0.5,200))
print(greedy(0.9,200))
```

UCB Algorithm

```
def UCB(c,times):
    rewards=npy.zeros(times)
    for i in range(times):
        N = 5000

        theta = [0,0,0]
        count = [0,0,0]
        for j in range(3):
            count[j] += 1

            rit = reward(j)
            rewards[i] += rit
            theta[j] += rit
        for t in range(3,N):
            max_value = 0

            max_index = 0

            l=npy.zeros(3)
            for k in range(3):
                l[k] = theta[k] + c * npy.sqrt((2 * npy.log(t + 1)) / count[k])
            if l[0]==l[1]:
                if l[1]==l[2]:
                    max_index = npy.random.randint(0, 3)
                else:
                    max_index = npy.random.randint(0, 2)
            elif l[1]==l[2]:
                max_index = npy.random.randint(1, 2)
            elif l[0]==l[2]:
                max_index = random.choice([0,2])
            else:
                max_index = npy.argmax(l)
            rit = reward(max_index)
            rewards[i] += rit
            count[max_index] += 1

            theta[max_index]=theta[max_index] + 1/(count[max_index])*(rit-theta[max_index])
    return npy.average(rewards)
```

TS (Thompson Sampling)Algorithm

```
def TS(a,b,times):
    rewards=npy.zeros(times)
    for t in range(times):
        alpha=copy.deepcopy(a)
        beta= copy.deepcopy(b)
        N = 5000
```

```
        for i in range(N):
            max_value = 0

            max_index = 1

            for j in range(3):
                p = npy.random.beta(alpha[j],beta[j])
                if p > max_value:
                    max_value = p
                    max_index = j
            rit = reward(max_index)
            alpha[max_index] += rit
            beta[max_index] += 1-rit
            rewards[t] += rit
    return npy.average(rewards)
```

Problem 3

The output of ε-greedy algorithms is:

$$E(\varepsilon) = \begin{cases} 3407.955 & \varepsilon = 0.1 \\ 3076.395 & \varepsilon = 0.5 \\ 2751.5 & \varepsilon = 0.9 \end{cases}$$

The output of UCB algorithms is:

$$E(c) = \begin{cases} 3409.345 & c = 1 \\ 2980.63 & c = 5 \\ 2830.955 & c = 10 \end{cases}$$

The output of TS (Thompson Sampling) Algorithm is:

$$E(\alpha, \beta) = \begin{cases} 3480.065 & \alpha = (1,1,1), \beta = (1,1,1) \\ 3491.2 & \alpha = (601,401,2), \beta = (401,601,3) \end{cases}$$

Problem 4

The gap of ε-greedy algorithms is:

$$G(\varepsilon) = \begin{cases} 92.045 & \varepsilon = 0.1 \\ 423.605 & \varepsilon = 0.5 \\ 748.5 & \varepsilon = 0.9 \end{cases}$$

The gap of UCB algorithms is:

$$G(c) = \begin{cases} 90.655 & c = 1 \\ 519.37 & c = 5 \\ 669.145 & c = 10 \end{cases}$$

The gap of TS (Thompson Sampling) Algorithm is:

$$G(\alpha, \beta) = \begin{cases} 19.935 & \alpha = (1,1,1), \beta = (1,1,1) \\ 8.8 & \alpha = (601,401,2), \beta = (401,601,3) \end{cases}$$

The gap between the TS Algorithm output and the oracle are the smallest, so TS is the best.

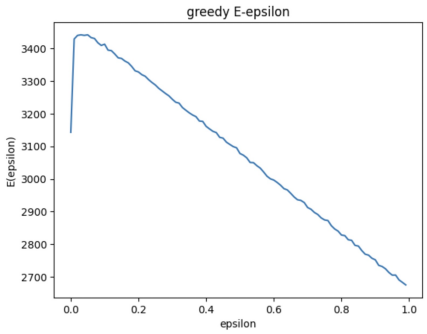

Figure 1: Simulation of ϵ-greedy algorithm

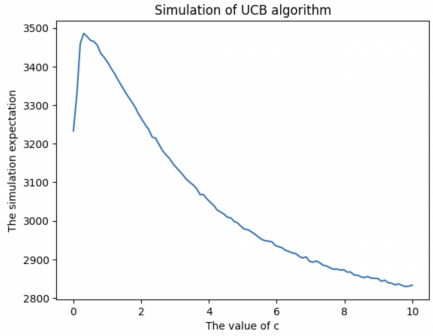

Figure 2: Simulation of UCB algorithm

ϵ-greedy

The ϵ-greedy algorithm takes the best action most of the time, but does random exploration occasionally. It explores with probability ϵ at each time step. This allows you to choose in some quantitative way how to balance exploration and exploitation. Through Figure 1, we can know that when $\epsilon = 0.05$, the reward reaches its best 3444.335. And then the larger the ϵ is, the less the reward will be.

UCB (Upper Confidence Bound) Algorithm

UCB can measure the uncertainty of whether the exploration is bad so that we can choose the one with the highest potential to be best. c is a confidence value that controls the level of exploration. The larger c is,the greater the level of exploration is since the factor $\sqrt{\frac{2log(t)}{count(j)}}$ is bigger for arms pulled for less times in the previous time(we can also say that the upper confidence bound is artificially increased). Larger gives more chance for those less-pulled arms to get pulled. Through Figure 2, we can know that when $c = 0.3$, the reward reaches its best 3486.785.

TS (Thompson Sampling) Algorithm

TS algorithm uses prior and posterior. We assume a Beta prior on each unknown probability of reward and update the parameter α and β after each trial. α and β correspond to the counts when we succeeded or

failed to get a reward respectively. α and β implies our prior belief for every action. For example, $\alpha = 1$ and $\beta = 1$ means that we expect the reward probability to be 50% but we are not very confident. $\alpha = 2000$ and $\beta = 8000$ means that we strongly believe that the reward probability is 20%. The larger α is and smaller β is, the reward probability of the arm will be. The larger the two parameter are, each trial can't change our estimation of reward probability easily. So the choice we made are more close to the greatest reward choice and the total reward will be larger.

Problem 5

The exploration vs exploitation dilemma exists in many aspects of our life includes in reinforcement learning. Since we don't have enough information, we need to gather more information to make best overall decisions while keeping the risk under control. With exploitation, we take advantage of the best option we know. With exploration, we take some risk to collect information about unknown options. The best long-term strategy must involve short-term sacrifices.

If we keep choosing the best one present which is exploitation, we take the smallest risk. But the reward gotten has limit. If empirical data is not enough, the best choice present maybe a bad choice in the long run and the reward will be very little. We need exploration because information is valuable.In the other hand, if we keep exploring, the risk of losing keeps high because we can't estimate what the result will be. In order to make the reward maximize, we should find the balance between exploration and exploitation. For the arms with known high probability of getting reward, exploitation should be more, but for the arms unknown or being tried for less times, we should allocate a certain amount of times to exploration.

The ϵ-greedy algorithm do exploration at random, while the UCB algorithm and TS algorithm do exploration smartly with preference to uncertainty. It makes intuitive sense that explore smartly with preference uncertainty will perform better than exploring at random. UCB and TS algorithm perform better than ϵ-greedy algorithm. TS algorithm works good because it takes this idea of MAP into account, and it uses ingenious idea of sampling rather than just taking the MAP. So when finding the balance between exploitation and exploration, using probability matching is very useful. Bayesian inference plays an important role in this bandit problem.

Exploration should decrease with time, since information increases with time, and the agent should not always explore. Exploration period should not either be too long or too short. There may exists a best length for exploration period.

Problem 6

Dependent rules

Assume that the probability of getting reward of each arm is independent to the ratio of the reward earned to the time before. Once pulling an arm getting reward, the probability of this arm decrease by p and the probability of the other two arms increase by $\frac{p}{2}$ each. If pulling an arm doesn't get reward, the probability of this arm increases by p and the other two arms decrease by $\frac{p}{2}$.

the Problem we may face

1.Always use the slot machine with the highest probability at present or let the slot machines with different probabilities rotate to get the highest score?

2. There may be some problems with the classical algorithm. For example, the probability of using the slot machine with the highest probability is no longer the highest because of too many slots. But the algorithm is not aware of this. Leading to mistakes in decision-making.

the Algorithms we provide

In order to obtain a better result, we design three algorithms: n_UCB, n_TS and Tslowdeath.

n_UCB and n_TS use the same method as before.We just update the rewards function where probability of the arm will change.In order to distinguish them from the previous algorithm, we call it n_UCB,n_TS

To solve the first problem, we developed an algorithm: Tslowdeath

Tslowdeath does some improvement: in the first $exporation(e)$ trials, use TS algorithm to help making decision and get a posterior probability distribution of each arm. Then in the next $5000 - exporation(e)$ trials, the arm with largest prior distribution will be chosen with probability p_1 and the arm with the second largest prior distribution will be chosen with probability p_2.

We assume $p = 0.001$and $p = 0.0001$ and do some simulation.

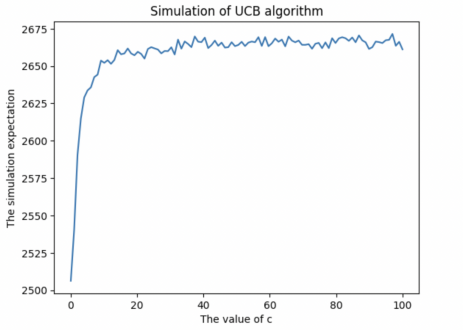

Figure 3: Simulation of n_UCB algorithm when p=0.001

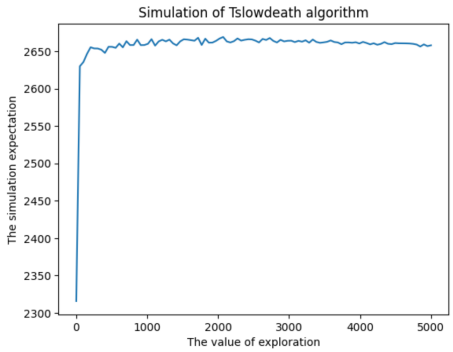

Figure 4: Simulation of Ts algorithm when p=0.001

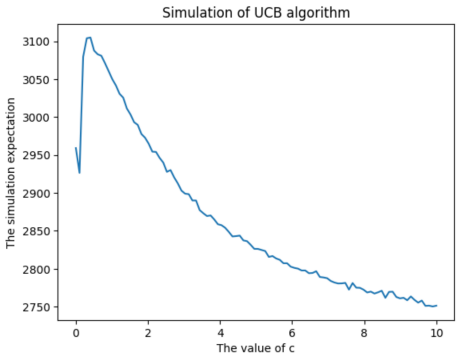

Figure 5: Simulation of n_UCB algorithm when p=0.0001

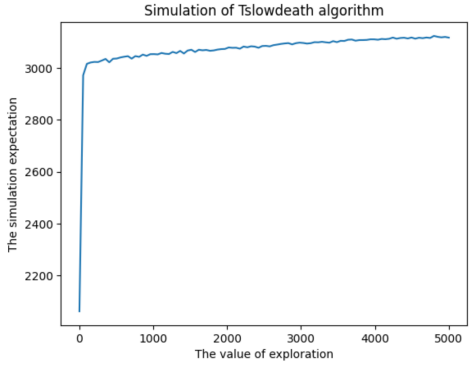

Figure 6: Simulation of Ts algorithm when p=0.0001

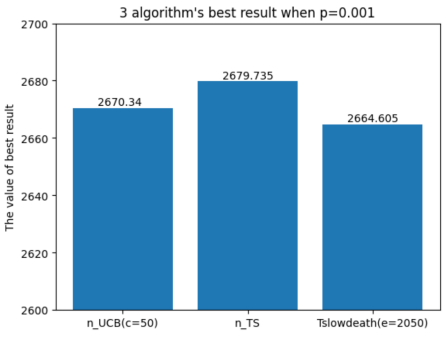

Figure 7: 3 algorithm's best result when p=0.001

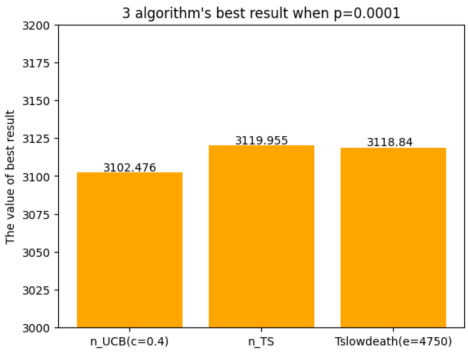

Figure 8: 3 algorithm's best result when p=0.0001

Analysis of simulation

For the first problem, Firgure 6 and Figure 8 shows that there is no need to use other arm to make the probability of the best arm decrease slowly.

After simulation,we can see that only when using the n_UCB algorithm, will the probability of three arms become the same at last.

When $p = 0.001$,as c getting large, the reward become better and keep in a stable range if c is larger than about 20. It makes sense that in dependent case, there should be more exploration than in independent case. The results of n_TS are much better.

when $p = 0.0001$ the n_UCB increase and decrease.The result of n_UCB is much dependent on c. So if we don't know the value of p,then we don't know how to determine c.So it makes sense that using n_TS is more reliable

Part II

Bayesian Bandit Algorithms

There are two arms which may be pulled repeatedly in any order. Each pull may result in either a success or a failure. The sequence of successes and failures which results from pulling arm $i(i \in \{1, 2\})$ forms a Bernoulli process with unknown success probability θ_i. A success at the t^{th} pull yields a reward $\gamma^{t-1}(0 < \gamma < 1)$, while an unsuccessful pull yields a zero reward. At time zero, each θ_i has a Beta prior distribution with two parameters α_i, β_i and these distributions are independent for different arms. These prior distributions are updated to posterior distributions as arms are pulled. Since the class of Beta distributions is closed under Bernoulli sampling, posterior distributions are all Beta distributions. How should the arm to pull next in each time slot be chosen to maximize the total expected reward from an infinite sequence of pulls?

Problem 1

Question: One intuitive policy suggests that in each time slot we should pull the arm for which the current expected value of i is the largest. This policy behaves very good in most cases. Please design simulations to check the behavior of this policy.

nn_nn Algorithm

---

nn_nn Algorithm for problem 1

Initialize Beta parameter (α_j, β_j), $j \in \{1, 2\}$

1: **for** $t = 1, 2 \ldots,$ N **do**

2: **for** $j \in \{1, 2\}$ **do**

3: $\mathrm{E}(\hat{\theta}(j)) = \frac{\alpha_j}{\alpha_j + \beta_j}$

4: **end for**

5: *# Select and pull the arm*

6: $I(t) \leftarrow \arg\max_{j \in \{1,2\}} E(\hat{\theta}(j))$

7: *# Update the distribution*

8: **if** $r_{I(t)} = 1$

$\alpha_{I(t)} \leftarrow \alpha_{I(t)} + 1$

$\beta_{I(t)} \leftarrow \beta_{I(t)}$

$reward \leftarrow reward + \gamma^{t-1}$

9: **if** $r_{I(t)} = 0$

$\alpha_{I(t)} \leftarrow \alpha_{I(t)}$

$\beta_{I(t)} \leftarrow \beta_{I(t)} + 1$

10.**end for**

```
def nn_reward(arg,theta):
    if npy.random.rand(1)>theta[arg]:
        return 0

    else:
        return 1

def generatetheta(alpha,beta):
    theta=npy.zeros(2)
    theta[0]=npy.random.beta(alpha[0],beta[0])
    theta[1]=npy.random.beta(alpha[1],beta[1])
    return theta
def nn_nn(a,b,times,gamma,theta):
    rewards=npy.zeros(times)
    for t in range(times):
        alpha=copy.deepcopy(a)
        beta= copy.deepcopy(b)
        N = 5000

        for i in range(N):
            Ep = npy.zeros(2)
            max_index = 0

            for j in range(2):
                Ep[j] = alpha[j]/(alpha[j]+beta[j])
                if Ep[0] > Ep[1]:
                    max_index = 0

                elif Ep[0] < Ep[1]:
                    max_index = 1

                else:
                    max_index = npy.random.randint(0, 2)
            rit = nn_reward(max_index,theta)
            alpha[max_index] += rit
            beta[max_index] += 1-rit
            x=math.pow ((rit*gamma), i)
            if (x>0)&(x<0.001):
                break
            rewards[t] += x
    return npy.average(rewards)
def nnn_nnn(alpha,beta,times,gamma):
    rewards=npy.zeros(times)
    for t in range(times):
        rewards[t] = nn_nn(alpha,beta,100,gamma,generatetheta(alpha,beta))
    return npy.average(rewards)
```

```
Gamma=npy.linspace(0.95, 1, 50)
Egamma=npy.zeros(50)
for i in range(50):
    Egamma[i]=nnn_nnn(alpha,beta,100,0.95+i*0.001)
plt.plot(Gamma,Egamma)
plt.xlabel('The value of gamma')
plt.ylabel('The value of Egamma')
plt.title('Simulation of E for different gamma')
plt.show()
```

Simulation

assume the prior distribution of θ_1, θ_2 are Beta(1,1)

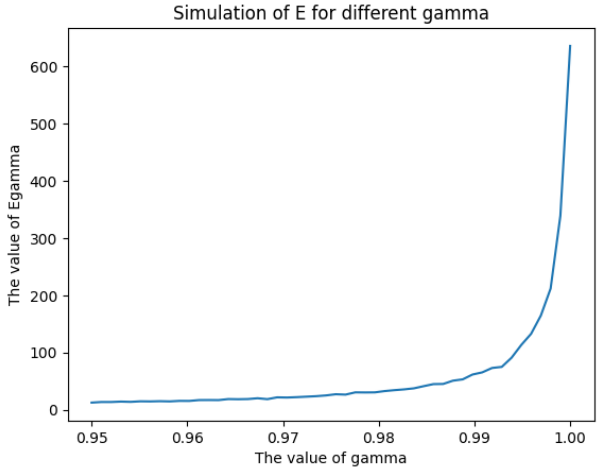

Figure 9: Simulation of E for different gamma

god view

$$E(\gamma) = E(\theta_{max}) \sum_{n=0}^{N-1} \gamma^n = \frac{E(\theta_{max})(1-\gamma^N)}{1-\gamma}$$

and $E(\theta_{max}) = 0.672$

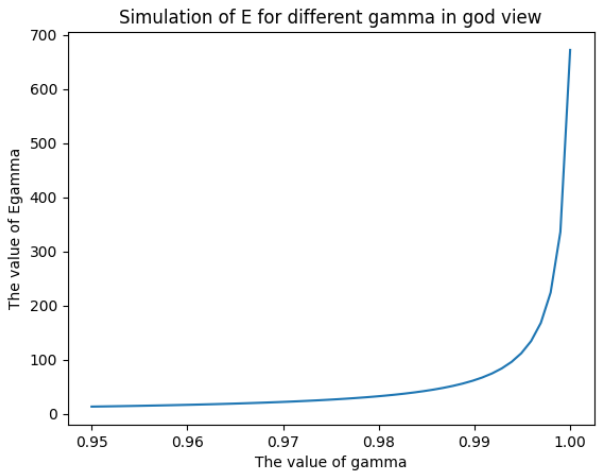

Figure 10: Simulation of E for different gamma in god view

gap of simulation and god view

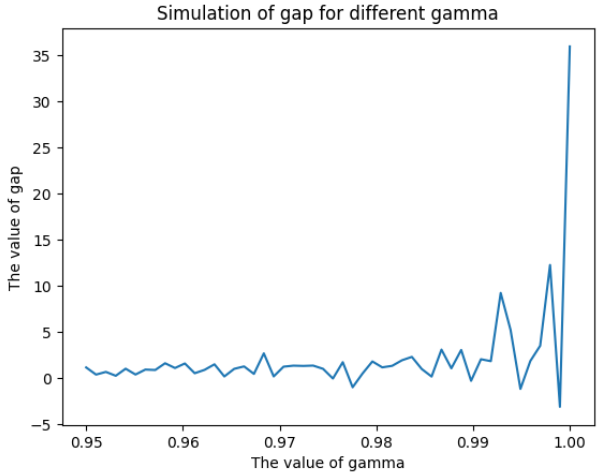

Figure 11: Simulation of gap for different gamma

Final we find that it is situable in most cases through the result where θ_1, θ_2 are Beta(1,1)

Problem 2

Question: However, such intuitive policy is unfortunately not optimal. Please provide an example to show why such policy is not optimal.

Solution:

When the prior distribution is biased, this method often has a big problem

Such as real $\theta_1 = 1, \theta_2 = 0.5$and prior distribution is (1, 2) (2, 1)

Obviously, the prior distribution differs greatly from the actual situation. However, due to the selection of algorithm, $\theta_2$will be selected every time, and the score probability of number 2 is high enough to almost guarantee that the expectation of θ_2's prior distribution will not fall lower than the $\frac{1}{11}$ which is the expectation of θ_1's prior distribution

So, in this more extreme case, $\theta_2$is taken every time.

Figure 12 is the simulation of the case above,while Figure 13 is the simulation in god's view.And we find that there are big differences between the two figures.

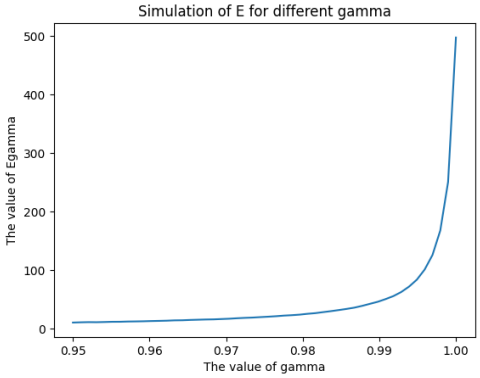

Figure 12: Simulation of E for different gamma

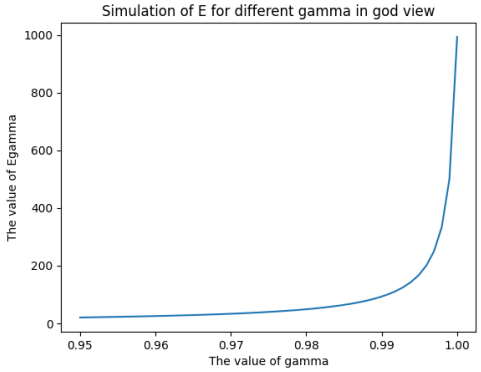

Figure 13: Simulation of E for different gamma in god view

On this special point, we can intuitively feel that this algorithm has defects. But it is not strict enough to show that this algorithm is defective.

Compare with TS

Then we set $[\theta_1, \theta_2] \sim [Beta(2,1), Beta(1,2)]$ Sampling with nn algorithm and TS algorithm respectively

```
def nn_TS(a,b,times,gamma,theta):
    rewards=npy.zeros(times)
    for t in range(times):
        alpha=copy.deepcopy(a)
        beta= copy.deepcopy(b)
        N = 5000  #
        for i in range(N):
            #
            Ep = npy.zeros(2)
            max_index = 0
```

```
            for j in range(2):
                #
                Ep[j] = npy.random.beta(alpha[j],beta[j])
                #
                if Ep[0] > Ep[1]:
                    max_index = 0

                elif Ep[0] < Ep[1]:
                    max_index = 1

                else:
                    max_index = npy.random.randint(0, 2)
            #
            rit = nn_reward(max_index,theta)
            alpha[max_index] += rit
            beta[max_index] += 1-rit
            x=math.pow ((rit*gamma), i)
            if (x>0)&(x<0.001):
                break
            rewards[t] += x
    return npy.average(rewards)
def nnn_TS(alpha,beta,times,gamma):
    rewards=npy.zeros(times)
    for t in range(times):
        rewards[t] = nn_TS(alpha,beta,100,gamma,generatetheta(alpha,beta))
    return npy.average(rewards)


alpha=[2,1]
beta=[1,2]
Gamma=npy.linspace(0.95, 1, 10)
Egap=npy.zeros(10)
ETSgamma=npy.zeros(10)
Enngamma=npy.zeros(10)
for i in range(10):
    ETSgamma[i]=nnn_TS(alpha,beta,100,0.95+i*0.005)
    Enngamma[i]=nnn_nnn(alpha,beta,100,0.95+i*0.005)
    Egap[i]=ETSgamma[i]-Enngamma[i]
plt.plot(Gamma,Egap)
plt.xlabel('The value of gamma')
plt.ylabel('The value of Egap')
plt.title('Simulation of gap for different gamma in[2,1,1,2]')
plt.show()
```

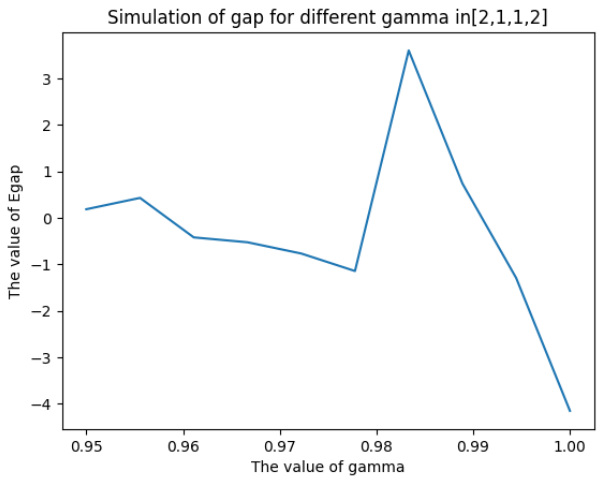

Figure 14: the gap of the nn_nn algorithm and nn_TS algorithm

here is the gap of the nn_nn algorithm and nn_TS algorithm

We find that in some points ,nn_nn Algorithms is worse than nn_TS, so we know that nn_nn is not the optimal policy.Of course nor does nn_TS

Therefore it is clear that this algorithm is irrational.

Problem 3

$R_1(\alpha_1, \beta_1)$ indicates the total expected reward if pull arm 1 at $t = 1$. $R_2(\alpha_2, \beta_2)$ indicates the total expected reward if pull arm 2 at $t = 1$. $R(\alpha_1, \beta_1, \alpha_2, \beta_2)$ is the expected total reward under an optimal policy. There are two choices at first: pulling arm 1 or 2. So the maximum reward with these beta parameters is the greater one between pulling arm 1 and 2. Hence,

$$R(\alpha_1, \beta_1, \alpha_2, \beta_2) = max\{R_1(\alpha_1, \beta_1), R_2(\alpha_2, \beta_2)\}$$

A_j:Choose j as first pulling

S_j:pull jth arm and get score

F_j:pull jth arm and not get score

T_j:total score before the jth pulling

$$R_1(\alpha_1, \beta_1) = E(T|A_1) \overset{LOTE}{=\!=} E(T|A_1, S_1)P(S_1) + E(T|A_1, F_1)P(F_1)$$

$$E(T_1|A_1, S_1) = E(T_2) + 1$$

$$E(T_2) = \sum_{t=2}^{\infty} \hat{\theta}'_t \gamma^{t-1}, \hat{\theta}'_t \sim Beta(\alpha_1 + 1, \beta_1)$$

We know

$$E(T_1) = \sum_{t=1}^{\infty} \hat{\theta}_t \gamma^{t-1}, \hat{\theta}_t \sim Beta(\alpha_1, \beta_1)$$

so

$$E(T_1|S_0) = \sum_{t=1}^{\infty} \hat{\theta}'_t \gamma^{t-1}$$

since $\gamma < 1 \Rightarrow \sum_{t=1}^{\infty} \theta'_t \gamma^{t-1}$ converges

$$\gamma E(T_1|S_0) = \sum_{t=1}^{\infty} \hat{\theta}'_t \gamma^{t} = \sum_{t=2}^{\infty} \hat{\theta}'_t \gamma^{t-1} = E(T_2)$$

$$E(T_1|A_1, S_1) = E(T_2) + 1 = \gamma E(T_1|S_0) + 1 = \gamma R(\alpha_1 + 1, \beta_1, \alpha_2, \beta_2) + 1$$

similarly

$$E(T_1|A_1, F_1) = \gamma R(\alpha_1, \beta_1 + 1, \alpha_2, \beta_2)$$

$$\hat{\theta}_t \sim Beta(\alpha_1, \beta_1) \Rightarrow P(S_1) = p(\hat{\theta}_t) = \frac{\alpha_1}{\alpha_1 + \beta_1}, P(F_1) = \frac{\beta_1}{\alpha_1 + \beta_1}$$

Hence, the total reward of pulling arm 1 at $t = 1$ is

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

Similarly, the total reward of pulling arm 2 at $t = 1$ is

$$R_2(\alpha_2, \beta_2) = \frac{\alpha_2}{\alpha_2 + \beta_2}[1 + \gamma R(\alpha_1, \beta_1, \alpha_2 + 1, \beta_2)] + \frac{\beta_2}{\alpha_2 + \beta_2}[\gamma R(\alpha_1, \beta_1, \alpha_2, \beta_2 + 1)]$$

Problem 4&5

Algorithm for problem 4&5

1.**Initialize:**$R(\vec{S}) = 0, \vec{S} = \{(a,b,c,d), a < N+\alpha_1, b < N+\beta_1, c < N+\alpha_2, d < N+\beta_2\}, \Delta \leftarrow 0$

2.**While**(True):

3. $r \leftarrow R$

4 **for** a = 1,2....(N+α_1)

5 **for** b= 1,2.....(N+β_1)

6 **for** c = 1,2...$(N+\alpha_2)$

7 **for** d=1,2$(N+\beta_2)$

8 $R(a,b,c,d) \leftarrow max\{\frac{a}{a+b}[1+\gamma r(a+1,b,c,d)]+\frac{b}{a+b}[\gamma r(a,b+1,c,d)], \frac{c}{c+d}[1+\gamma r(a,b,c+1,d)] + \frac{d}{c+d}[\gamma r(a,b,c,d+1)]\}$

9$\Delta \leftarrow max(\Delta, |r(a,b,c,d) - R(a,b,c,d)|)$

10 end for

11 end for

12 end **for**

13 end **for**

14 **if** $\Delta < \theta$

15 **break**

16.**for** a = 1,2....(N+α_1)

17 **for** b= 1,2.....(N+β_1)

18 **for** c = 1,2...$(N+\alpha_2)$

19 **for** d=1,2$(N+\beta_2)$

20. $\pi(a,b,c,d) \leftarrow argmax\{\frac{a}{a+b}[1+\gamma R(a+1,b,c,d)] + \frac{b}{a+b}[\gamma R(a,b+1,c,d)], \frac{c}{c+d}[1+\gamma R(a,b,c+1,d)] + \frac{d}{c+d}[\gamma R(a,b,c,d+1)]\}$

21 end **for**

22 end **for**

23 end **for**

24end **for**

Then $R(\alpha_1, \beta_1, \alpha_2, \beta_2)$is the exactly what we need in Problem 4

$\pi(\alpha_1, \beta_1, \alpha_2, \beta_2)$ is the index of optimal policy when Posterior distribution is at $[(\alpha_1, \beta_1), (\alpha_2, \beta_2)]$

Another thinking of Problem 4

since when $\gamma \to 1$,"N"in last Algorithm should be large,which makes the Algorithm slow.

Then I found a method to solve it approximately.

$R(\alpha_1, \beta_1, \alpha_2, \beta_2) = max\{R_1(\alpha_1, \beta_1), R_2(\alpha_2, \beta_2)\}$

If $\frac{\alpha_1}{\beta_1} > \frac{\alpha_2}{\beta_2}$

when $\gamma \to 1$

$$R_1(\alpha_1 + 1, \beta_1) \approx R_1(\alpha_1, \beta_1) \approx R_1(\alpha_1, \beta_1 + 1)$$

$$\begin{aligned}
R_1(\alpha_1, \beta_1) &> R_2(\alpha_2, \beta_2) \\
R(\alpha_1, \beta_1, \alpha_2, \beta_2) &= R_1(\alpha_1, \beta_1) \\
&= \frac{\alpha_1}{\alpha_1 + \beta_1}[1 + \gamma R(\alpha_1 + 1, \beta_1, \alpha_2, \beta_2)] + \frac{\beta_1}{\alpha_1 + \beta_1}[\gamma R(\alpha_1, \beta_1 + 1, \alpha_2, \beta_2)] \\
so \frac{\alpha_1 + 1}{\beta_1} &> \frac{\alpha_1}{\beta_1} \Rightarrow R(\alpha_1 + 1, \beta_1, \alpha_2, \beta_2) = R_1(\alpha_1 + 1, \beta_1) \\
&= \frac{\alpha_1}{\alpha_1 + \beta_1}[1 + \gamma R_1(\alpha_1 + 1, \beta_1)] + \frac{\beta_1}{\alpha_1 + \beta_1}\gamma \cdot max\{R_1(\alpha_1, \beta_1 + 1), R_2(\alpha_2, \beta_2)\} \\
R_1(\alpha_1, \beta_1) &= \frac{\alpha_1}{\alpha_1 + \beta_1}(1 + \gamma R_1(\alpha_1, \beta_1)) + \frac{\beta_1}{\alpha_1 + \beta_1}\gamma R_1(\alpha_1, \beta_1) \\
&= \frac{\alpha_1}{\alpha_1 + \beta_1} + \gamma R_1(\alpha_1, \beta_1) \\
\Rightarrow R_1(\alpha_1, \beta_1) &= \frac{\frac{\alpha_1}{\alpha_1 + \beta_1}}{1 - \gamma}
\end{aligned}$$

then we have

$$R_2(\alpha_2, \beta_2) = \frac{\alpha_2}{\alpha_2 + \beta_2} + \frac{\gamma\alpha_1}{(1 - \gamma)(\alpha_1 + \beta_1)}$$

similarly, if $\frac{\alpha_1}{\beta_1} < \frac{\alpha_2}{\beta_2}$

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

$$R_2(\alpha_2, \beta_2) = \frac{1}{1 - \gamma}\frac{\alpha_2}{\alpha_2 + \beta_2}$$

In total

$$R_1 = \begin{cases} \frac{1}{1-\gamma}\frac{\alpha_1}{\alpha_1+\beta_1} & \frac{\alpha_1}{\beta_1} > \frac{\alpha_2}{\beta_2} \\ \frac{\alpha_1}{\alpha_1+\beta_1} + \frac{\gamma\alpha_2}{(1-\gamma)(\alpha_2+\beta_2)} & \frac{\alpha_1}{\beta_1} < \frac{\alpha_2}{\beta_2} \end{cases}$$

$$R_2 = \begin{cases} \frac{\alpha_2}{\alpha_2+\beta_2} + \frac{\gamma\alpha_1}{(1-\gamma)(\alpha_1+\beta_1)} & \frac{\alpha_1}{\beta_1} > \frac{\alpha_2}{\beta_2} \\ \frac{1}{1-\gamma}\frac{\alpha_2}{\alpha_2+\beta_2} & \frac{\alpha_1}{\beta_1} < \frac{\alpha_2}{\beta_2} \end{cases}$$

Part III

Conclusion

We do some simulation of ϵ-greedy Algorithm, UCB Algorithm and TS Algorithm. And we find that TS Algorithm performs the best since the gap between the result of TS Algorithm and the oracle value is the smallest. A balance should be found to have a better result in the exploration vs exploitation dilemma. As for the dependent case, we design three algorithms: n _UCB, n_TS and Tslowdeath to obtain a better result.

As for the Bayesian Bandit Algorithms, we design nn_nn Algorithm to do some simulation and we found that when the prior distribution is biased, this algorithm has problem. Simulation shows that the nn_nn algorithm is worse than TS algorithm, so the intuitive policy is not optimal. Solution of the recurrence equation of expected total reward has been introduced and we design a new optimal algorithm which can get better total reweard.

Part IV

Reference

[1]Reinforcement Learning:An Introduction second edition Richard S. Sutton and Andrew G. Barto Chapter 4 Dynamic Programming

Part V

Introduction of authors

Li Chaofan was born in Wuhan, China.
He is an undergraduate in School of Information Science and Technology at ShanghaiTech University.
His major is electrical engineering.
He is interested in Integrated Circuits and Semiconductors.

Qin Yao was born in Shanghai, China.
She is an undergraduate in School of Information Science and Technology at ShanghaiTech University.
Her major is electrical engineering.
She is interested in Power & Energy Engineering.

Ziyu Shao is the teacher of Li Chaofan and Qin Yao in Probability and Statistics(SI140A). Dr. Shao is a Permanent Associate Professor (Tenured) in the School of Information Science and Technology in ShanghaiTech University.He is currently the Director of the School's Public Relations Committee, Director of the Network Intelligence Center and member of the Teaching Committee. Dr. Shao received his bachelor's and Master's degrees from the School of Information Science and Technology at Peking University and his doctorate degree from the Department of Information Engineering at the Chinese University of Hong Kong. Dr. Shao is a senior member of IEEE, a member of the Council of Shanghai Computer Society, and Deputy Director of the Theoretical Computer Committee of Shanghai Computer Society.