bandit_learning

May 15, 2022

Final Project

1 Part 1

Now suppose we obtain the Bernoulli distribution parameters from an oracle, which are shown in the following table below. Choose N = 6000 and compute the theoretically maximized expectation of aggregate rewards over N time slots. We call it the oracle value. Note that these parameters $_j, j = 1, 2, 3$ and oracle values are unknown to all bandit algorithms.

Arm j	1	2	3
θ_j	0.8	0.6	0.5

[2]:
```
N = 6000
theta = max(0.8, 0.6, 0.5)
E = N * theta
print("the theoretically maximized expectation of aggregate rewards is", E)
```

```
the theoretically maximized expectation of aggregate rewards is 4800.0
```

2 Part 2

Implement classical bandit algorithms with following settings: - N=6000 - ϵ-greedy with $\epsilon = 0.2, 0.4, 0.6, 0.8$. - UCB with $c = 2, 6, 9$. - Thompson Sampling with

```
{$(\alpha _1,\beta _1)=(1,1),(\alpha _2,\beta _2)=(1,1),(\alpha _3,\beta _3)=(1,1)$} and

{$(\alpha _1,\beta _1)=(601,401),(\alpha _2,\beta _2)=(401,601),(\alpha _3,\beta _3)=(2,3)$}
```

2.1 ϵ-greedy

[60]:
```
import numpy as np


class Greedy:
    def __init__(self, epsilon) -> None:
        self._epsilon = epsilon
```

```
        self._count = [0, 0, 0]
        self._theta = np.array([0.0, 0.0, 0.0])
        self._real_arg = np.array([0.0, 0.0, 0.0])
        self._agg = 0

    def set_up_real(self, real_arg):
        self._real_arg = real_arg

    def exploration(self):
        x = np.random.multinomial(1, np.array([1/3, 1/3, 1/3]))
        for i in range(len(x)):
            if x[i] == 1:
                return i
        return -1

    def exploitation(self):
        j = np.argmax(self._theta)
        return j

    def reward(self, j):
        p = self._real_arg[j]
        r = np.random.binomial(1, p)
        self._agg = self._agg+r
        return r

    def run(self, N):
        for i in range(N):
            indicator = np.random.binomial(1, self._epsilon)
            if indicator:
                j = self.exploration()
            else:
                j = self.exploitation()
            self._count[j] = self._count[j]+1
            r = self.reward(j)
            self._theta[j] = self._theta[j]+(r-self._theta[j])/self._count[j]


if __name__ == "__main__":
    epsilon = [0.2, 0.4, 0.6, 0.8]
    for i in range(len(epsilon)):
        g = Greedy(epsilon[i])
        theta = np.array([0.8, 0.6, 0.5])
        np.random.shuffle(theta)
        g.set_up_real(theta)
        g.run(6000)
        print("epsilon-Greedy with epsilon=%.1f:" % epsilon[i], g._agg)
```

```
epsilon-Greedy with epsilon=0.2: 4625
```

```
epsilon-Greedy with epsilon=0.4: 4421
epsilon-Greedy with epsilon=0.6: 4207
epsilon-Greedy with epsilon=0.8: 3997
```

2.2 UCB

[58]:
```python
import numpy as np


class UCB:
    def __init__(self, c=1) -> None:
        self._c = c
        self._count = [0, 0, 0]
        self._theta = np.array([0.0, 0.0, 0.0])
        self._real_arg = np.array([0.0, 0.0, 0.0])
        self._agg = 0

    def set_up_real(self, real_arg):
        self._real_arg = real_arg

    def reward(self, j):
        p = self._real_arg[j]
        r = np.random.binomial(1, p)
        self._agg = self._agg+r
        return r

    def run(self, N):
        for i in range(3):
            self._count[i] = 1
            self._theta[i] = self.reward(i)
        for i in range(3, N):
            arr = np.array([0.0, 0.0, 0.0])
            for j in range(3):
                arr[j] = self._theta[j]+self._c * \
                    np.sqrt(2*np.log(i)/self._count[j])
            j = np.argmax(arr)
            self._count[j] = self._count[j]+1
            r = self.reward(j)
            self._theta[j] = self._theta[j]+(r-self._theta[j])/self._count[j]


if __name__ == "__main__":
    c = [2, 6, 9]
    for i in range(len(c)):
        g = UCB(c[i])
        theta = np.array([0.8, 0.6, 0.5])
        np.random.shuffle(theta)
```

```
        g.set_up_real(theta)
        g.run(6000)
        print("UCB with c=%d:" % c[i], g._agg)
```

```
UCB with c=2: 4555
UCB with c=6: 4157
UCB with c=9: 4043
```

2.3 Thompson Sampling

[54]:
```
import numpy as np


class ThompsonSampling:
    def __init__(self, para) -> None:
        self._para = para
        self._agg = 0
        self._real_arg = np.array([0.0, 0.0, 0.0])

    def set_up_real(self, real_arg):
        self._real_arg = real_arg

    def reward(self, j):
        p = self._real_arg[j]
        r = np.random.binomial(1, p)
        self._agg = self._agg+r
        return r

    def run(self, N):
        for i in range(N):
            sample = [0, 0, 0]
            for i in range(3):
                sample[i] = np.random.beta(
                    self._para[i][0], self._para[i][1])
            j = np.argmax(sample)
            r = self.reward(j)
            self._para[j][0] = self._para[j][0]+r
            self._para[j][1] = self._para[j][1]+1-r


if __name__ == "__main__":
    para = [[[1, 1], [1, 1], [1, 1]],
            [[601, 401], [401, 601], [2, 3]]]
    for i in range(len(para)):
        t = ThompsonSampling(para[i])
        theta = np.array([0.8, 0.6, 0.5])
```

```
        # no shuffle here because alpha and beta carries informtion to␣
↪corresponding arms
        t.set_up_real(theta)
        t.run(6000)
        print("Thompson Sampling %d:" % (i+1), t._agg)
```

```
Thompson Sampling 1: 4865
Thompson Sampling 2: 4836
```

3 Part 3

Each experiment lasts for N = 6000 time slots, and we run each experiment 200 times. Results are averaged over these 200 independent runs.

3.1 ϵ-greedy

[62]:
```
import numpy as np


class Greedy:
    def __init__(self, epsilon) -> None:
        self._epsilon = epsilon
        self._count = [0, 0, 0]
        self._theta = np.array([0.0, 0.0, 0.0])
        self._real_arg = np.array([0.0, 0.0, 0.0])
        self._agg = 0

    def set_up_real(self, real_arg):
        self._real_arg = real_arg

    def exploration(self):
        x = np.random.multinomial(1, np.array([1/3, 1/3, 1/3]))
        for i in range(len(x)):
            if x[i] == 1:
                return i
        return -1

    def exploitation(self):
        j = np.argmax(self._theta)
        return j

    def reward(self, j):
        p = self._real_arg[j]
        r = np.random.binomial(1, p)
        self._agg = self._agg+r
        return r
```

```
    def run(self, N):
        for i in range(N):
            indicator = np.random.binomial(1, self._epsilon)
            if indicator:
                j = self.exploration()
            else:
                j = self.exploitation()
            self._count[j] = self._count[j]+1
            r = self.reward(j)
            self._theta[j] = self._theta[j]+(r-self._theta[j])/self._count[j]


if __name__ == "__main__":
    iter = 200
    epsilon = [0.2, 0.4, 0.6, 0.8]
    for i in range(len(epsilon)):
        sum = 0
        for j in range(iter):
            g = Greedy(epsilon[i])
            theta = np.array([0.8, 0.6, 0.5])
            np.random.shuffle(theta)
            g.set_up_real(theta)
            g.run(6000)
            sum = sum+g._agg
        sum = sum/iter
        print("epsilon-Greedy with epsilon=%.1f:" % epsilon[i], sum)
```

```
epsilon-Greedy with epsilon=0.2: 4593.64
epsilon-Greedy with epsilon=0.4: 4395.465
epsilon-Greedy with epsilon=0.6: 4197.395
epsilon-Greedy with epsilon=0.8: 3997.095
```

3.2 UCB

[63]:
```
import numpy as np


class UCB:
    def __init__(self, c=1) -> None:
        self._c = c
        self._count = [0, 0, 0]
        self._theta = np.array([0.0, 0.0, 0.0])
        self._real_arg = np.array([0.0, 0.0, 0.0])
        self._agg = 0

    def set_up_real(self, real_arg):
        self._real_arg = real_arg
```

```
    def reward(self, j):
        p = self._real_arg[j]
        r = np.random.binomial(1, p)
        self._agg = self._agg+r
        return r

    def run(self, N):
        for i in range(3):
            self._count[i] = 1
            self._theta[i] = self.reward(i)
        for i in range(3, N):
            arr = np.array([0.0, 0.0, 0.0])
            for j in range(3):
                arr[j] = self._theta[j]+self._c * \
                    np.sqrt(2*np.log(i)/self._count[j])
            j = np.argmax(arr)
            self._count[j] = self._count[j]+1
            r = self.reward(j)
            self._theta[j] = self._theta[j]+(r-self._theta[j])/self._count[j]


if __name__ == "__main__":
    iter = 200
    c = [2, 6, 9]
    for i in range(len(c)):
        sum = 0
        for j in range(iter):
            g = UCB(c[i])
            theta = np.array([0.8, 0.6, 0.5])
            np.random.shuffle(theta)
            g.set_up_real(theta)
            g.run(6000)
            sum = sum+g._agg
        sum = sum/iter
        print("UCB with c=%d:" % c[i], sum)
```

```
UCB with c=2: 4544.355
UCB with c=6: 4143.795
UCB with c=9: 4031.83
```

3.3 Thompson Sampling

[64]:
```
import numpy as np


class ThompsonSampling:
    def __init__(self, para) -> None:
```

```
        self._para = para
        self._agg = 0
        self._real_arg = np.array([0.0, 0.0, 0.0])

    def set_up_real(self, real_arg):
        self._real_arg = real_arg

    def reward(self, j):
        p = self._real_arg[j]
        r = np.random.binomial(1, p)
        self._agg = self._agg+r
        return r

    def run(self, N):
        for i in range(N):
            sample = [0, 0, 0]
            for i in range(3):
                sample[i] = np.random.beta(
                    self._para[i][0], self._para[i][1])
            j = np.argmax(sample)
            r = self.reward(j)
            self._para[j][0] = self._para[j][0]+r
            self._para[j][1] = self._para[j][1]+1-r


if __name__ == "__main__":
    iter = 200
    para = [[[1, 1], [1, 1], [1, 1]],
            [[601, 401], [401, 601], [2, 3]]]
    for i in range(len(para)):
        sum = 0
        for j in range(iter):
            t = ThompsonSampling(para[i])
            theta = np.array([0.8, 0.6, 0.5])
            # no shuffle here because alpha and beta carries informtion to␣
 ↪corresponding arms
            t.set_up_real(theta)
            t.run(6000)
            sum = sum+t._agg
        sum = sum/iter
        print("Thompson Sampling %d:" % (i+1), sum)
```

```
Thompson Sampling 1: 4801.875
Thompson Sampling 2: 4794.82
```

4 Part 4

Compute the gaps between the algorithm outputs (aggregated rewards over N time slots) and the oracle value. Compare the numerical results of ϵ-greedy, UCB, and Thompson Sampling. Which one is the best? Then discuss the impacts of ϵ, c,and α_j, β_j respectively.

4.1 The numerical results

4.1.1 ϵ-greedy

[67]:
```
import numpy as np


class Greedy:
    def __init__(self, epsilon) -> None:
        self._epsilon = epsilon
        self._count = [0, 0, 0]
        self._theta = np.array([0.0, 0.0, 0.0])
        self._real_arg = np.array([0.0, 0.0, 0.0])
        self._agg = 0

    def set_up_real(self, real_arg):
        self._real_arg = real_arg

    def exploration(self):
        x = np.random.multinomial(1, np.array([1/3, 1/3, 1/3]))
        for i in range(len(x)):
            if x[i] == 1:
                return i
        return -1

    def exploitation(self):
        j = np.argmax(self._theta)
        return j

    def reward(self, j):
        p = self._real_arg[j]
        r = np.random.binomial(1, p)
        self._agg = self._agg+r
        return r

    def run(self, N):
        for i in range(N):
            indicator = np.random.binomial(1, self._epsilon)
            if indicator:
                j = self.exploration()
            else:
```

```
                j = self.exploitation()
            self._count[j] = self._count[j]+1
            r = self.reward(j)
            self._theta[j] = self._theta[j]+(r-self._theta[j])/self._count[j]


if __name__ == "__main__":
    iter = 200
    epsilon = [0.2, 0.4, 0.6, 0.8]
    for i in range(len(epsilon)):
        sum = 0
        for j in range(iter):
            g = Greedy(epsilon[i])
            theta = np.array([0.8, 0.6, 0.5])
            np.random.shuffle(theta)
            g.set_up_real(theta)
            g.run(6000)
            sum = sum+g._agg
        sum = sum/iter
        print("gap between epsilon-Greedy with epsilon=%.1f and the oracle␣
 ↪value:" % epsilon[i], 4800-sum)
```

```
gap between epsilon-Greedy with epsilon=0.2 and the oracle value:
207.41499999999996
gap between epsilon-Greedy with epsilon=0.4 and the oracle value: 402.0
gap between epsilon-Greedy with epsilon=0.6 and the oracle value: 605.085
gap between epsilon-Greedy with epsilon=0.8 and the oracle value:
798.8249999999998
```

4.1.2 UCB

[66]:
```
import numpy as np


class UCB:
    def __init__(self, c=1) -> None:
        self._c = c
        self._count = [0, 0, 0]
        self._theta = np.array([0.0, 0.0, 0.0])
        self._real_arg = np.array([0.0, 0.0, 0.0])
        self._agg = 0

    def set_up_real(self, real_arg):
        self._real_arg = real_arg

    def reward(self, j):
        p = self._real_arg[j]
        r = np.random.binomial(1, p)
```

```
        self._agg = self._agg+r
        return r

    def run(self, N):
        for i in range(3):
            self._count[i] = 1
            self._theta[i] = self.reward(i)
        for i in range(3, N):
            arr = np.array([0.0, 0.0, 0.0])
            for j in range(3):
                arr[j] = self._theta[j]+self._c * \
                    np.sqrt(2*np.log(i)/self._count[j])
            j = np.argmax(arr)
            self._count[j] = self._count[j]+1
            r = self.reward(j)
            self._theta[j] = self._theta[j]+(r-self._theta[j])/self._count[j]


if __name__ == "__main__":
    iter = 200
    c = [2, 6, 9]
    for i in range(len(c)):
        sum = 0
        for j in range(iter):
            g = UCB(c[i])
            theta = np.array([0.8, 0.6, 0.5])
            np.random.shuffle(theta)
            g.set_up_real(theta)
            g.run(6000)
            sum = sum+g._agg
        sum = sum/iter
        print("gap between UCB with c=%d and the oracle value:" %
              c[i], 4800-sum)
```

```
gap between UCB with c=2 and the oracle value: 258.5600000000004
gap between UCB with c=6 and the oracle value: 655.9300000000003
gap between UCB with c=9 and the oracle value: 767.0149999999999
```

4.1.3 Thompson Sampling

[65]:
```
import numpy as np


class ThompsonSampling:
    def __init__(self, para) -> None:
        self._para = para
        self._agg = 0
```

```
        self._real_arg = np.array([0.0, 0.0, 0.0])

    def set_up_real(self, real_arg):
        self._real_arg = real_arg

    def reward(self, j):
        p = self._real_arg[j]
        r = np.random.binomial(1, p)
        self._agg = self._agg+r
        return r

    def run(self, N):
        for i in range(N):
            sample = [0, 0, 0]
            for i in range(3):
                sample[i] = np.random.beta(
                    self._para[i][0], self._para[i][1])
            j = np.argmax(sample)
            r = self.reward(j)
            self._para[j][0] = self._para[j][0]+r
            self._para[j][1] = self._para[j][1]+1-r


if __name__ == "__main__":
    iter = 200
    para = [[[1, 1], [1, 1], [1, 1]],
            [[601, 401], [401, 601], [2, 3]]]
    for i in range(len(para)):
        sum = 0
        for j in range(iter):
            t = ThompsonSampling(para[i])
            theta = np.array([0.8, 0.6, 0.5])
            # no shuffle here because alpha and beta carries informtion to␣
 ↪corresponding arms
            t.set_up_real(theta)
            t.run(6000)
            sum = sum+t._agg
        sum = sum/iter
        print("gap between Thompson Sampling %d and the oracle value:" %
              (i+1), 4800-sum)
```

```
gap between Thompson Sampling 1 and the oracle value: -1.2399999999997817
gap between Thompson Sampling 2 and the oracle value: 1.1300000000001091
```

4.1.4 As we can tell from the result of gaps to the oracle value of each algorithm, Thompson Sampling performs the best, and the next is UCB, following the ϵ-greedy. It is super great that the result of Thompson Sampling is very close to the oracle value.

4.2 The impacts of the parameters

4.2.1 ϵ

ϵ in ϵ-greedy is the probability in each turn that it choose exploration rather than exploitation.

Larger ϵ means the algorithm would lay great emphasis on exploration. This can give a more precise evalution on θ, but accordingly it might relatively lose the chance to exploit that discovery, leading to a bad result. In the extreme case, if ϵ equals 1, then the algorithm just choose randomly from the three arms all the time. Smaller ϵ means the algorithm would exploit more. Also consider the extreme case, if ϵ equals 0, the algorithm would always choose the first arm.

In this trial, we have $\epsilon = 0.2, 0.4, 0.6, 0.8$, and the result shows that the algorithm works the best with $\epsilon = 0.2$. In fact, I also tried $\epsilon = 0.1, 0.01$, and the gaps to the oracle value can be small, but is actually very unsteady (i.e. have a great variance, such as $\epsilon = 0.01$ can make up a 1000 gap to the oracle), because it easily falls into a local optimal, but omitting the global optimal (i.e. wrongly evaluate θ because of the lack of trial). So, generally speaking, the value of ϵ should be smaller to perform better, but not too small to escape from the local optimal.

4.2.2 c

c in UCB scales the confidence interval of each θ, especially the upper bound of those confidence interval.

For a constant c, the more times arm j is chosen, the range of θ_j's confidence interval (negatively correlated with $count[j]$) is smaller. So, if an arm has been hardly chosen, then its θ's upper confidence bound can be much greater then its $\hat{\theta}$, representing a great potential.

Back to this question, suppose we are facing a certain situation, the value of c scales the range of confidence interval. Larger c leads to greater upper confidence bound, indicating greater tolerance to such situation, while smaller c leads to smaller upper confidence bound. In an extreme case that $c = 0$, the algorithm would easily ignore arms if they failed in the first couple of times.

In this trial, we have $c = 2, 6, 9$, and the result tells that the algorithm works the best with $c = 2$. I also tried $c = 1, 0.5, 0.1, 0$. $c = 1$ and $c = 0.5$ performs better, while $c = 0$ and $c = 0.1$ performs very unsteady with a great variance since it can easily drop into a local optimal, due to a very little tolerance. Generally speaking, smaller c (as a positive integer) performs better, and there should be a best value for c in the interval $(0, 2]$.

4.2.3 α_j, β_j

By Beta-Binomial conjugacy, α_j and β_j represent the number of success of the arm j and failure respectively. Then the algorithm generates probabilities for each arm with its posterior distribution $Beta(\alpha_j, \beta_j)$.

In this trial, we have $\{(\alpha_1, \beta_1) = (1, 1), (\alpha_2, \beta_2) = (1, 1), (\alpha_3, \beta_3) = (1, 1)\}$ and $\{(\alpha_1, \beta_1) = (601, 401), (\alpha_2, \beta_2) = (401, 601), (\alpha_3, \beta_3) = (2, 3)\}$. The former one that every parameter equals 1, implies that we have no knowledge about these arms, and give each of them a fair uniform

distribution. The latter one that initialize $(\alpha_1, \beta_1) = (601, 401)$ for example, implies that we know something about this arm and predict its θ to be $Beta(601, 401)$. I also tried some strange parameters like $\{(\alpha_1, \beta_1) = (1, 101), (\alpha_2, \beta_2) = (1, 1), (\alpha_3, \beta_3) = (1, 1)\}$, whose gap to the oracle value is about 1200 (roughly estimated), since the prediction of the θ of the first arm (which is the most important since it is the largest) is totally wrong. So if we really don't know a thing about those arms, we had better take all parameters as 1 (i.e.$\{(\alpha_1, \beta_1) = (1, 1), (\alpha_2, \beta_2) = (1, 1), (\alpha_3, \beta_3) = (1, 1)\}$).

5 Part 5

Give your understanding of the exploration-exploitation trade-off in bandit algorithms.

Most things in real life cannot be guaranteed as certain situations. With uncertain input, or even the input that we have to estimate, we may want to think of a method to better approach the oracle value.

In the case that we already know all of the θ s, we can easily choose the largest amont them to maximize the reward, and take the expectation of this as the oracle value. While in real bandit problem, we are to select the arm with the largest θ more, in order to maximze the reward. And this brings a question that how we can evaluate θ (the chance to get reward from an arm) appropriately, that we don't have to waste time on reletively low rewarding arms.

There comes the exploration-exploitation trade-off, that we explore to try more possibilities and find what these θ should be, or we exploit to stay on the instantly best arm to get more rewards. Exploitation promises a good way to maximize the one-step reward and also the aggregated reward, while exploration enable us to estimate these θ more precisely and ensure a better total reward in long-term. However, if we exploit all the time, we can easily fall into a local optimal since we don't have clear knowledge about all the arms, and if we explore all the time, then we lose the chance to make use of our discovery during exploration. There comes the conflict between exploitation and exploration since we can hardly exploit and explore both at the same time. These three algorithm introduce different methods.

In ϵ-greedy, we simply estimate θ_j by the frenquency that arm j success. This is readily comprehensible, but it can cost hundreds of trials on each arm to figure out the approximate value each of θ. In this case, exploration costs too much effort, and the time for exploitation is squeezed out, so we can hardly find a well ϵ for this algorithm to greatly improve its performance. Except that, I have found that explorations done late in the N trials are less meaningful than thoes done before, since we have less and less time to exploit our discovery as time goes by. So, I think it is reasonable to multiply ϵ be a factor that decreases monotonically over time, which indicates we desire exploitation rather than exploration over time. We can take the method used in Simulated Annealing into consideration. In another way, we can also encourage exploration at the beginning by initializing $\hat{\theta}$ with a great number, i.e. 0.8 or 1, optimistically.Those approaches slightly increase the result in experiments I have done, but is still not comparable with Thompson Sampling.

In UCB, the upper confidence bound evalute the potential of each arm, and c stands for the tolerance towards those arms that return little rewards in little trials. To give those arms another chance, that is a kind of exploration. As a arm has been pulled much more times, the upper confidence bound tends to converge to $\hat{\theta}$, its potential shifts to the value of $\hat{\theta}$. We always select the arm that has the greatest potential to be optimal. In this case, we ignore arms that has less potential to save the cost of redundant exploration, so that we can exploit more to get a better aggretated reward.

In Thompson Sampling, the algorithm generate each θ accoding to their distribution. It does not require a certain value of $\hat{\theta}$ nor a confidence interval, but it is probablistic. As we have little trials on each arm, the parameters of each Beta distribution is small, and has a relatively large variance, and this is the time of exploration. We explore to make each distribution's variance less, and then exploit the largest θ. Since it converges fast, this probablistic method takes less time of explorations to find best arm then finding by frenquency (deterministic), so it can get a better aggregated reward than these algorithm above, as it can have more exploitation.

Both exploration and exploitation are important. Exploration ensures that we don't miss the global optimal, and exploitation helps us get more rewards. These three algorithm find a balance in exploration-exploitation trade-off, and Thompson performans better.

6 Part 6

> We implicitly assume the reward distribution of three arms are independent. How about the dependent case? Can you design an algorithm to exploit such information to obtain a better result?

For

Arm j	1	2	3
θ_j	0.9	0.7	0.2

, and suppose we are given the correlation matrix, (given resonalby by the bandit, here I just give a demo sans accurate calculation)

$Corr(i,j)$	1	2	3
1	1	0.6	-0.5
2	0.6	1	0.2
3	-0.5	0.2	1

, in which $\rho[i,j]$ stands for the correlation of θ_i and θ_j. We can also calculate the correlations of these r.v.s if we want, but it is not necessary in this algorthm.

In Thompson Sampling, denote the θ of two arms under consideration X and Y, and it is easy to calculate $E(X), E(Y), Var(X), Var(Y)$. Then by $\frac{E(XY)-E(X)E(Y)}{\sqrt{Var(X)Var(Y)}} = Corr(X,Y)$, we can get the value of $E(XY)$, which is also the probability that both arms return rewards successfully, denoted by p_{11}. Similarly, we can calculate $p_{10} = E(X(1-Y)) = E(X) - E(XY)$, $p_{01} = E((1-X)Y) = E(Y) - E(XY)$, and $p_{00} = E((1-X)(1-Y)) = 1 - E(X) - E(Y) + E(XY)$.

By Bayes' rule, if we want to know whether Y would make it a success given X, then $P(Y=1|X=1) = \frac{p_{11}}{P(X=1)}$, where $P(X=1)$ is already obtained by Beta distribution; otherwise given $X=0$, $P(Y=1|X=0) = \frac{p_{01}}{1-P(X=1)}$.

In this way, it is equivelant to that we are able to do three parameter adjustment in one turn, so we can get to precise $\hat{\theta}$ faster.

[8]:
```
import numpy as np


class ThompsonSampling:
    def __init__(self, para) -> None:
        self._para = para
        self._agg = 0
        self._real_arg = np.array([0.0, 0.0, 0.0])
        self._count = np.array([0, 0, 0])

    def set_up_real(self, real_arg):
        self._real_arg = real_arg

    def set_up_corr(self, corr):
        self._corr = corr

    def reward(self, j):
        p = self._real_arg[j]
        r = np.random.binomial(1, p)
        self._agg = self._agg+r
        return r

    def get_beta_E(self, j):
        a = self._para[j][0]
        b = self._para[j][1]
        return a/(a+b)

    def get_beta_Var(self, j):
        a = self._para[j][0]
        b = self._para[j][1]
        return a*b/((a+b)*(a+b)*(a+b+1))

    def update(self, j, r, pj):
        for i in range(3):
            if i == j:
                self._para[j][0] = self._para[j][0]+r
                self._para[j][1] = self._para[j][1]+1-r
                continue
            ej = self.get_beta_E(j)
            ei = self.get_beta_E(i)
            varj = self.get_beta_Var(j)
            vari = self.get_beta_Var(i)
            p11 = self._corr[j][i]*np.sqrt(vari*varj)+ei*ej
            if r == 1:
                pi = min(1, p11/pj)
                ri = np.random.binomial(1, pi)
            else:
```

```
                p01 = ei-p11
                pi = min(1, p01/(1-pj))
                ri = np.random.binomial(1, pi)
            self._para[i][0] = self._para[i][0]+ri
            self._para[i][1] = self._para[i][1]+1-ri

    def run(self, N):
        for i in range(N):
            sample = [0, 0, 0]
            for i in range(3):
                sample[i] = np.random.beta(
                    self._para[i][0], self._para[i][1])
            j = np.argmax(sample)
            self._count[j] = self._count[j]+1
            r = self.reward(j)
            self.update(j, r, sample[j])


if __name__ == "__main__":
    iter = 200
    para = [[[1, 1], [1, 1], [1, 1]]]

    for i in range(len(para)):
        sum = 0
        for j in range(iter):
            t = ThompsonSampling(para[i])
            t.set_up_real([0.9, 0.7, 0.2])
            t.set_up_corr(
                np.array([[1, 0.6, -0.5], [0.6, 1, -0.2], [-0.5, -0.2, 1]]))
            t.run(6000)
            sum = sum+t._agg
        sum = sum/iter
        print("Thompson Sampling %d:" % (i+1), sum)
```

```
Thompson Sampling 1: 5397.575
```

The oracle value is $0.9 \times 6000 = 5400$. This algorithm can work better with presice correlation given, while in this sample I just come up with a rough demo of the correlation matrix, that may have several missleading problems while processing, but it can work with no such problem with an accurate correlation matrix.

7 Part 7

> We implicitly assume there are no constraints when pulling arms. For example, pull each arm will generate some cost and there are some bounds on such cost. Can you design an algorithm for constrained bandit learning problem?

Assume that the cost of pulling each arm is less than the reward that it can return, as a bound of

such cost, or there would be no reason to select such arms. Suppose still we have

Arm j	1	2	3
θ_j	0.8	0.6	0.5

, and assume the cost of pulling each arm once is

Arm j	1	2	3
γ_j	0.4	0.1	0.3

So clearly if we know all these information, we simply choose arm 2 all the time since it has the greatest expected net reward, and the oracle value is $(0.6 - 0.1) \times 6000 = 3000$.

In this part, I would like to revamp Thompson Sampling algorithm by reflecting the idea of net income in it. In fact, when we evaluating $\hat{\theta}$ be Beta distributions, we are considering the expected (net) reward of each arm in one turn. Then, we also consider the net reward of each arm and take the best of them all the time in this situation.

[3]:
```
import numpy as np


class ThompsonSampling:
    def __init__(self, para) -> None:
        self._para = para
        self._agg = 0
        self._real_arg = np.array([0.0, 0.0, 0.0])
        self._cost = np.array([0.0, 0.0, 0.0])
        self._count = np.array([0, 0, 0])

    def set_up_real(self, real_arg):
        self._real_arg = real_arg

    def set_up_cost(self, cost):
        self._cost = cost

    def reward(self, j):
        p = self._real_arg[j]
        r = np.random.binomial(1, p)
        return r

    def run(self, N):
        for i in range(N):
            sample = [0, 0, 0]
            for i in range(3):
                sample[i] = np.random.beta(
                    self._para[i][0], self._para[i][1])
```

```
                sample[i] =sample[i]*(1-self._cost[i])-(1-sample[i])*self.
↪_cost[i]
            j = np.argmax(sample)
            self._count[j] = self._count[j]+1
            r = self.reward(j)
            self._para[j][0] = self._para[j][0]+r
            self._para[j][1] = self._para[j][1]+1-r
            if r == 0:
                self._agg = self._agg-self._cost[j]
            else:
                self._agg = self._agg+1-self._cost[j]


if __name__ == "__main__":
    iter = 200
    para = [[[1, 1], [1, 1], [1, 1]]]

    for i in range(len(para)):
        sum = 0
        for j in range(iter):
            t = ThompsonSampling(para[i])
            t.set_up_real([0.8, 0.6, 0.5])
            t.set_up_cost([0.4, 0.1, 0.3])
            t.run(6000)
            sum = sum+t._agg
        sum = sum/iter
        print("Thompson Sampling %d:" % (i+1), sum)
```

```
Thompson Sampling 1: 3000.759500000336
```

Another try: consider a tricky example, that

Arm j	1	2	3
θ_j	0.9	0.6	0.5

, and assume the cost of pulling each arm once is

Arm j	1	2	3
γ_j	0.3	-0.05	0.3

The oracle value is $(0.6 + 0.05) \times 6000 = 3900$.

[6]:
```
if __name__ == "__main__":
    iter = 200
    para = [[[1, 1], [1, 1], [1, 1]]]
```

```
    for i in range(len(para)):
        sum = 0
        for j in range(iter):
            t = ThompsonSampling(para[i])
            t.set_up_real([0.9, 0.6, 0.5])
            t.set_up_cost([0.3, -0.05, 0.3])
            t.run(6000)
            sum = sum+t._agg
        sum = sum/iter
        print("Thompson Sampling %d:" % (i+1), sum)
```

```
Thompson Sampling 1: 3899.5687500004274
```

Concluding that, the revamp version of Thompson Sampling is suitable in this situation, since the results are around their own oracle values.