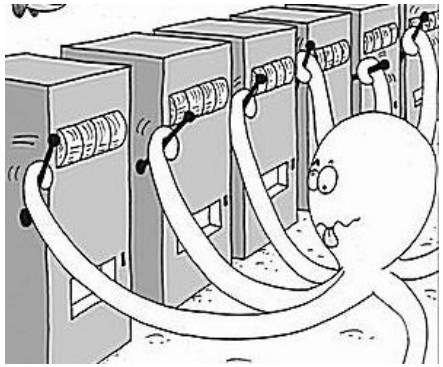
Bandit Problem

Adviser: Brown Nicholas

Author: Jiachi Sun



Before discussing the bandit and any details related to him, I first want to discuss the exploration-exploitation dilemma.

Exploration-exploitation dilemma

To use a simple analogy: a new restaurant opened downstairs, and after the opening I often go there to eat, ordering two or three dishes at a time. The first thing to emphasize is that if we choose a dish that is not good, we also need to bear the consequences and can not be left without a check. Then, to be on the safe side, when ordering I would probably order a dish that I thought was good and another one that I hadn't ordered yet. This way, I will not eat the same dish every time, and not once ordered all the bad dishes, so I can not swallow. It's also a trade-off between "exploration" and " exploitation". How do I order to make sure I don't get a table of dishes I don't like at all while trying to discover if there are new dishes that will satisfy me?

How to allocate resources wisely in both exploration and utilization is the question we want to explore, if the restaurant mentioned above has

100 dishes on the menu. We can have many kinds of strategies. For example.

1. First try 20 dishes, and then later each time only choose the best few dishes in these 20 dishes. This is a way to explore first and then take advantage.

2. Each time I order a dish that I thought was good last time, and then order a dish that has not been ordered. After trying them all choose the two dishes you like best. This is the method of exploring while taking advantage.

So how do we allocate our opportunities wisely to get the most happiness from our food and get the most out of it? This brings us to the Bandit problem.

Bandit Problem

Imagine that a gambler has N slot machines in front of him, and he doesn't know the real profit of each slot machine beforehand, how can he choose which one to pull next time or whether to stop gambling to maximize his profit based on the result of each slot machine he plays.

Suppose that instead of deciding which dish is the best, we come to a casino to play a slot machine. Now we have N slot machines in front of us and we don't know in advance the real profitability of each slot machine. We choose one machine at a time, put in one dollar, and pull

down the lever. At this point there are two results, we win money or we lose a dollar. At this point the question shifts from getting the most out of the different dishes above to—how to choose that slot machine with a reasonable strategy to get the most out of it. This is the famous bandit problem.

A simpler strategy is to start with $200 and invest $20 in each machine, regardless of the winner. After that, choose the machine with the highest win rate and put the remaining $800 into it. And we call this strategy Epsilon first.

There is no denying that this strategy is better than a completely random strategy. But there is one obvious disadvantage to this strategy, namely that after the initial exploration process, we will not explore at all. And it is likely that we will miss out on the most rewarding machines as a result. And, over time, we may find that the return on the machine we chose begins to fade, but we still have no option to try something else.

It is clear that the Epsilon First strategy is not satisfying us. Therefore, I will explore several relatively more effective strategies in the next articles: Epsilon greedy, Upper Confidence Bound (UCB), and Thompson Sampling.

The code below sets the necessary import and set up the standard k-arm bandit environment.

```
import numpy as np
import matplotlib.pyplot as plt
from pdb import set_trace

stationary=True
class Bandit():
  def __init__(self, arm_count):
    """
    Multi-armed bandit with rewards 1 or 0.

    At initialization, multiple arms are created. The probability of each arm
    returning reward 1 if pulled is sampled from Bernouilli(p), where p randomly
    chosen from Uniform(0,1) at initialization
```

```
    """
    self.arm_count = arm_count
    self.generate_thetas()
    self.timestep = 0
    global stationary
    self.stationary=stationary

  def generate_thetas(self):
    self.thetas = np.random.uniform(0,1,self.arm_count)

  def get_reward_regret(self, arm):
    """ Returns random reward for arm action. Assumes actions are 0-indexed
    Args:
      arm is an int
    """
    self.timestep += 1
    if (self.stationary==False) and (self.timestep%100 == 0) :
      self.generate_thetas()
    # Simulate bernouilli sampling
    sim = np.random.uniform(0,1,self.arm_count)
    rewards = (sim<self.thetas).astype(int)
    reward = rewards[arm]
    regret = self.thetas.max() - self.thetas[arm]

    return reward, regret
```

Epsilon greedy

Epsilon greedy is divided into two steps.

1. Randomly select one of the N slot machines with probability ϵ to pull the joystick (each slot machine has 1/N probability to be selected), i.e., explore.

2. Select one of the N slot machines with a probability of 1-ϵ and choose the one with the highest return on investment at the cut-off, i.e., exploit.

Here the value of ϵ controls the preference for exploit and explore, and each decision is made with probability ϵ for Exploration and 1-ϵ for Exploitation.

It can be found that the selection of ϵ directly determines the performance of the algorithm, and if ϵ=1, it will become a completely randomized algorithm. In general, we choose a smaller value between (0,1).

Compared to Epsilon first, Epsilon greedy has a better balance between exploration and exploitation. It offers a certain possibility to explore other slots whenever possible, choosing the slot with the highest payoff more often to get the payoff.

At the same time, however, this randomness becomes a drawback as the number of rounds grows. After enough rounds, we may have found the machine with the highest payoff, but the algorithm will still have an ϵ probability of making a random selection. This actually causes losses as well. And Epsilon greedy splits our choices per round into two cases by the hyperparameter ϵ. But there is no good use of historical information such as the number of times each machine has been selected, or the total number of rounds.

The code below implement the Epsilon-Greedy algorithm

```
epsilon = 0.1
class EpsilonGreedy():
 """
 Epsilon Greedy with incremental update.
 Based on Sutton and Barto pseudo-code, page. 24
 """
 def __init__(self, bandit):
  global epsilon
  self.epsilon = epsilon
  self.bandit = bandit
  self.arm_count = bandit.arm_count
  self.Q = np.zeros(self.arm_count) # q-value of actions
  self.N = np.zeros(self.arm_count) # action count

 @staticmethod
 def name():
  return 'epsilon-greedy'

 def get_action(self):
  if np.random.uniform(0,1) > self.epsilon:
   action = self.Q.argmax()
  else:
   action = np.random.randint(0, self.arm_count)
  return action

 def get_reward_regret(self, arm):
  reward, regret = self.bandit.get_reward_regret(arm)
  self._update_params(arm, reward)
  return reward, regret

 def _update_params(self, arm, reward):
```

```
self.N[arm] += 1 # increment action count
self.Q[arm] += 1/self.N[arm] * (reward - self.Q[arm]) # inc. update rule
```

Upper Confidence Bound

Let's return to UCB, whose core formula is shown below.

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right]$$

Where Qt(a) is the current gain of this arm A as of the current turn. t is the total number of turns in the experiment so far. Nt(a) is the number of times a has been selected so far. c is the hyperparameter specified by us and referred to as UCB1 when c = 1.

As the number of rounds increases log(t) gets larger. At this point, if a particular arm is selected a few times, then log(t)/Nt(a) will also be larger, and then the chances of this arm being selected will be larger and larger, and this is when we are performing the round of exploration. And when the Qt(a) of a certain arm is large, i.e., he has a high payoff, then it will be selected, and this is the exploitation. When the payoff of a particular arm is large enough or even larger than the upper bound of the confidence interval of any of the other arms, this arm will always be selected and UCB will stop doing exploration and focus on exploitation.

Compared to the previous methods, UCB better balances exploration and discovery, while making better use of historical information such as the number of times each machine is selected, or the total number of rounds. Even if an arm has a low return in the first few times, it still has a chance to be selected as the number of rounds increases.

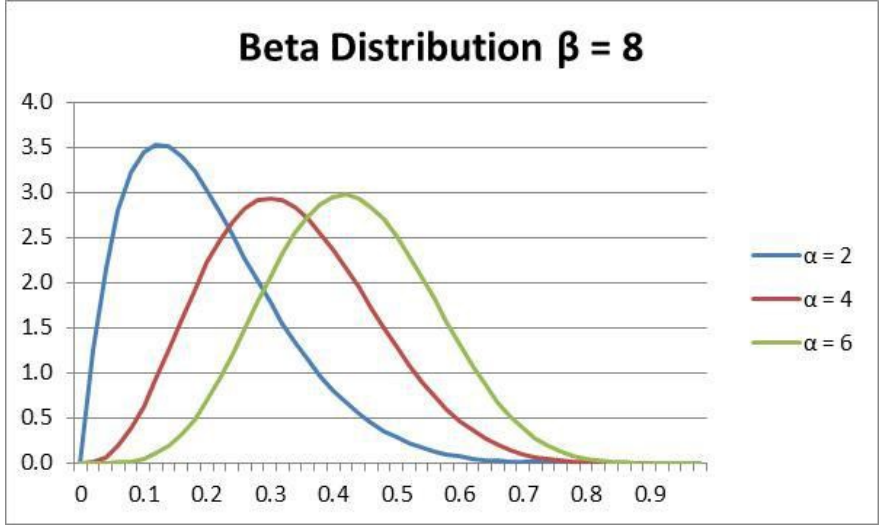
Also, the choice of c is critical, the larger c is, then the larger the upper bound of the confidence interval for the arm that was selected less often, and UCB will be more inclined to explore rather than perform an exploit.

The code below implement the UCB algorithm.

```
ucb_c = 2
class UCB():
 """
 Epsilon Greedy with incremental update.
 Based on Sutton and Barto pseudo-code, page. 24
 """
 def __init__(self, bandit):
  global ucb_c
  self.ucb_c = ucb_c
  self.bandit = bandit
  self.arm_count = bandit.arm_count
  self.Q = np.zeros(self.arm_count) # q-value of actions
  self.N = np.zeros(self.arm_count) + 0.0001 # action count
  self.timestep = 1

 @staticmethod
 def name():
  return 'ucb'

 def get_action(self):
  ln_timestep = np.log(np.full(self.arm_count, self.timestep))
  confidence = self.ucb_c * np.sqrt(ln_timestep/self.N)
  action = np.argmax(self.Q + confidence)
  self.timestep += 1
  return action

 def get_reward_regret(self, arm):
  reward, regret = self.bandit.get_reward_regret(arm)
  self._update_params(arm, reward)
  return reward, regret

 def _update_params(self, arm, reward):
  self.N[arm] += 1 # increment action count
  self.Q[arm] += 1/self.N[arm] * (reward - self.Q[arm]) # inc. update rule
```

Thompson Sampling

First of all, we need to talk about what is Beta distribution. It has two parameters greater than 0, α, β. Depending on the different values of a and b, it has the following image of the function.



We can see that the images drawn for different a's are different when the beta is the same.The larger the a, the more to the right the image is, and the greater the probability of getting a larger value when we take a random value from the beta distribution. Thompson Sampling makes good use of this property.

For the multi-armed slot machine scenario, the parameter for any slot machine X' beta distribution:

1. α is the number of times we pull the arm of the slot machine X and get a reward

2. β is the number of times we pull the arm of slot machine X and do not get a reward

In each selection, we let the beta distribution of each slot machine generate a random number and then select the slot machine that generates the maximum value and pull the arm. Subsequently, the α or β of the selected slot machine is updated depending on whether we get a reward or not.

The advantage of this is that we make full use of historical information and machines that are rarely selected or arms with low returns have a chance to be selected because we are generating a random number in the range of the beta distribution each round. Machines that perform better have a higher probability of generating a larger number and thus being selected, i.e., exploitation. At the same time, a machine that performs poorly also has some chance of generating a larger number and thus being selected, i.e., exploration.

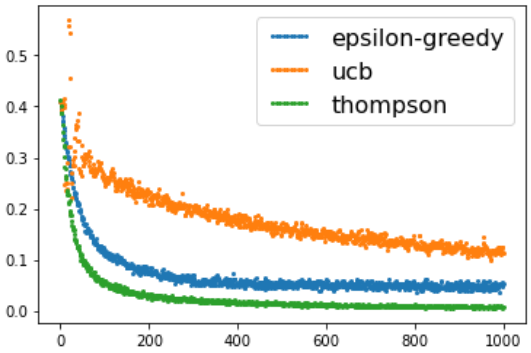
It is worth mentioning that when the beta distribution of a machine is large enough that the minimum value that can be generated is larger than the maximum value that can be generated by the beta distribution of any other set of slot machines, Thompson Sampling will also stop exploring and focus on exploiting instead.

The code below implement the Thompson Sampling with beta-distribution.

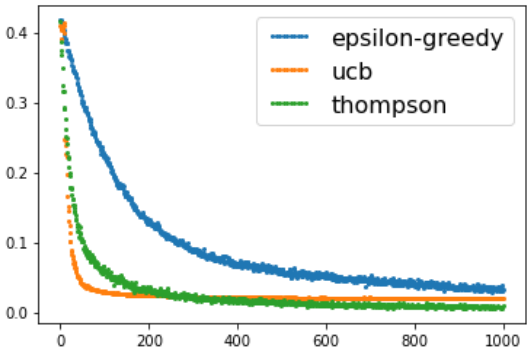
```
class BetaAlgo():
 """
 The algos try to learn which Bandit arm is the best to maximize reward.

 It does this by modelling the distribution of the Bandit arms with a Beta,
 assuming the true probability of success of an arm is Bernouilli distributed.
 """
 def __init__(self, bandit):
  """
  Args:
   bandit: the bandit class the algo is trying to model
  """
  self.bandit = bandit
  self.arm_count = bandit.arm_count
  self.alpha = np.ones(self.arm_count)
  self.beta = np.ones(self.arm_count)

 def get_reward_regret(self, arm):
  reward, regret = self.bandit.get_reward_regret(arm)
  self._update_params(arm, reward)
  return reward, regret

 def _update_params(self, arm, reward):
  self.alpha[arm] += reward
  self.beta[arm] += 1 - reward
```

```
class BernThompson(BetaAlgo):
  def __init__(self, bandit):
    super().__init__(bandit)

  @staticmethod
  def name():
    return 'thompson'

  def get_action(self):
    """ Bernouilli parameters are sampled from the beta"""
    theta = np.random.beta(self.alpha, self.beta)
    return theta.argmax()
```

Comparison of the performance of various methods and the effect of hyperparameters on performance



10 slot machine, epilson = 0.1, ucb_c = 2



10 slot machine, epilson = 0..03, ucb_c = 0.1

Thompson Sampling

The performance of Thompson Sampling is stable because it is not affected by hyperparameters, which is an advantage of Thompson Sampling. We do not need to adjust hyperparameters.

Epilson-Greedy

The hyperparameter for $\epsilon\epsilon$-greedy is the ϵ

The ϵ determines the probability of whether we choose to exploration or exploitation for each round.If we set ϵ to 1,the ϵ-greedy algorithm will like random sampling.If we set ϵ to 0.1, then there is a 10% probability that one restaurant will be randomly selected for exploration each time. There is a 90% probability of choosing the one with the highest current return.

We can see from the two sets of experiments below that the experimental results differ greatly when we set ϵ to 0.1 and 0.03, respectively. When $\epsilon\epsilon$ is 0.1, both ϵ-greedy and Thompson Sampling can be stabilized at about 200 times. However, when we set $\epsilon\epsilon$ to 0.03, ϵ-greedy converge to a steady regret value after 800 steps.

UCB

The hyperparameter for UCB is the constant c.

The larger we set the value of c, the smaller the weight of $Qt(a)$Qt(a). Also, the larger the value of c is set, the more the UCB algorithm focuses on exploration.

From the comparison of the two sets of experiments below we can find that the value of the constant c has a very strong influence on the UCB. When we set c to 2, the UCB algorithm does not stabilize until the 1000th time. But when we set C to 0.1, UCB performs very well and converge to a steady regret value at 100 steps.

Some helper method

```
def plot_data(y):
  """ y is a 1D vector """
  x = np.arange(y.size)
  _ = plt.plot(x, y, 'o')
```

```
def multi_plot_data(data, names):
  """ data, names are lists of vectors """
  x = np.arange(data[0].size)
  for i, y in enumerate(data):
    plt.plot(x, y, 'o', markersize=2, label=names[i])
  plt.legend(loc='upper right', prop={'size': 16}, numpoints=10)
  plt.show()

def simulate(simulations, timesteps, arm_count, Algorithm):
  """ Simulates the algorithm over 'simulations' epochs """
  sum_regrets = np.zeros(timesteps)
  for e in range(simulations):
    bandit = Bandit(arm_count)
    algo = Algorithm(bandit)
    regrets = np.zeros(timesteps)
    for i in range(timesteps):
      action = algo.get_action()
      reward, regret = algo.get_reward_regret(action)
      regrets[i] = regret
    sum_regrets += regrets
  mean_regrets = sum_regrets / simulations
  return mean_regrets

def experiment(arm_count, timesteps=1000, simulations=1000):
  """
  Standard setup across all experiments
  Args:
    timesteps: (int) how many steps for the algo to learn the bandit
    simulations: (int) number of epochs
  """
# EpsilonGreedy, UCB, BernThompson, RandomSampling,
  algos = [EpsilonGreedy, UCB, BernThompson, RandomSampling]
  regrets = []
  names = []
  for algo in algos:
    regrets.append(simulate(simulations, timesteps, arm_count, algo))
    names.append(algo.name())
  multi_plot_data(regrets, names)
```

Application of Bandit

The multi-armed slot machine problem has a wide range of applications in business, including advertising displays, medical trials, and finance. For example, in a recommendation system, we have N items, and we do not know in advance how user A will react to N items, we need to recommend a certain item to the user each time to maximize the value of the user (or try to make user A convert), such as the user's purchase.

Or in life, we need to choose a dress, pick a restaurant, can be done in this way to choose.

Code

You can find the complete code here:
https://github.com/aiskunks/Skunks_Skool/blob/main/CSYE_7370/Final%20Project/Bandit_Problem/SunJiachi_CSYE7

Reference

1. Andre Cianflone—Thompson sampling
https://github.com/andrecianflone/thompson/blob/master/thompson.ipynb

2. Improve Your Project Management Through Beta Distribution
https://www.6sigma.us/beta-distribution/improve-project-management-beta-distribution/

3. 探索-利用困境（exploration-exploitation dilemma）
https://zhuanlan.zhihu.com/p/161284124