Lecture 19: Exploration and Exploitation

Bolei Zhou

The Chinese University of Hong Kong

bzhou@ie.cuhk.edu.hk

March 30, 2020

Outline

- Introduction
- Exploration and Exploitation
- Multi-Armed Bandits
- Greedy and ϵ-greedy algorithms
- Softmax bandit algorithm
- Upper Confidence Bound (UCB) algorithm
- Example code

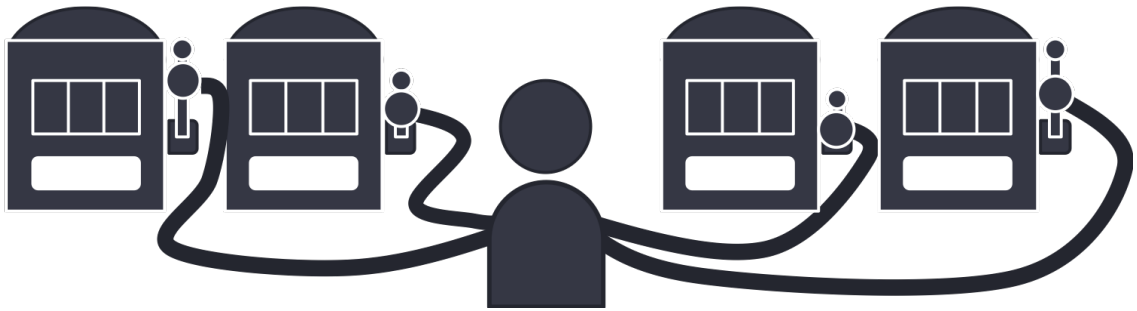
Trade-off between Exploration and Exploitation

1. Decision making invovles a fundamental choice:
   Exploitation: Choose the best known action
   Exploration: Explore some unknown action
2. Short-term reward v.s. long-term reward: To collect information about action which results to long-term reward may involve the sacrifice in short-term reward.
3. Collect enough information to make the best overall decisions

Examples

1. Going restaurant
    1. Exploitation: Go to your favorite restaurant
    2. Exploration: Try a new restaurant
2. Oil Drilling
    1. Exploitation: Drill at the best known location
    2. Exploration: Drill at a new location
3. Webpage design
    1. Exploitation: Copy some existing template
    2. Exploration: Design your own from scratch
4. Game playing
    1. Exploitation: Play the move you already knows to work
    2. Exploration: Do some experimental move
5. New Year Resolution
    1. Exploitation: Stay in your comfort zone
    2. Exploration: Try some new thing

The k-Armed Bandit



1. A multi-armed bandit is a tuple $< \mathcal{A}, \mathcal{R} >$
2. k actions to take at each step t
3. $\mathcal{R}^a(r) = P(r|a)$ is unknown probability distribution over rewards
4. At each step t the agent selects an action $a_t \in \mathcal{A}$, then the environment generates a reward $r_t \sim \mathcal{R}^{a_t}$
5. The goal of agent is to maximize cumulative reward $\sum_{\tau=1}^{T} r_\tau$

```
class BernoulliArm():
  def __init__(self, p):
    self.p = p
  def draw(self):
    if random.random() > self.p:
      return 0.0
    else:
      return 1.0

class NormalArm():
  def __init__(self, mu, sigma):
    self.mu = mu
    self.sigma = sigma
  def draw(self):
    return random.gauss(self.mu, self.sigma)
```

Definition of Value Function and Action-Value Function

1. The action-value is the mean reward for action a (unknown)

$$Q(a) = \mathbb{E}(r|a) \tag{1}$$

2. The optimal value

$$V^* = Q(a^*) = \max_{a \in \mathcal{A}} Q(a) \tag{2}$$

3. To estimate $Q(a)$, we can let $Q_(a)$ at step t

$$Q_t(a) = \frac{\text{sum of rewards when a taken prior to t}}{\text{number of times a taken prior to t}} = \frac{\sum_{i=1}^{t-1} R_i \cdot 1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_i=a}} \tag{3}$$

Greedy Action and ϵ-Greedy Action to Take

1. The estimation of $Q(a)$ at step t

$$Q_t(a) = \frac{\text{sum of rewards when a taken prior to t}}{\text{number of times a taken prior to t}} = \frac{\sum_{i=1}^{t-1} R_i \cdot 1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_i=a}} \tag{4}$$

2. Greedy action selection algorithm: $A_t = \arg\max_a Q_t(a)$
3. Problem with the greedy algorithm?
4. ϵ-Greedy: greedy most of the time, but with small probability ϵ select random actions (ϵ is usually as 0.1)
   1. probability $1 - \epsilon$: $A_t = \arg\max_a Q_t(a)$
   2. probability ϵ : $A_t = \mathit{uniform}(\mathcal{A})$

ϵ-Greedy Algorithm

Algorithm 1 Simple *epsilon*-Greedy bandit algorithm

1: **for** a $= 1$ to k **do**
2: $\quad Q(a) = 0, N(a) = 0$
3: **end for**
4: **loop**
5: $\quad A = \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\epsilon \\ uniform(\mathcal{A}) & \text{with probability } \epsilon \end{cases}$
6: $\quad R = bandit(A)$
7: $\quad N(A) = N(A) + 1$
8: $\quad Q(A) = Q(A) + \frac{1}{N(A)}[R - Q(A)]$
9: **end loop**

Deriving: NewEstimate = OldEstimate + StepSize[Target - OldEstimate]

$$Q_t(a_t) = Q_{t-1} + \frac{1}{N_t(a_t)}(r_t - Q_{t-1}) \tag{5}$$

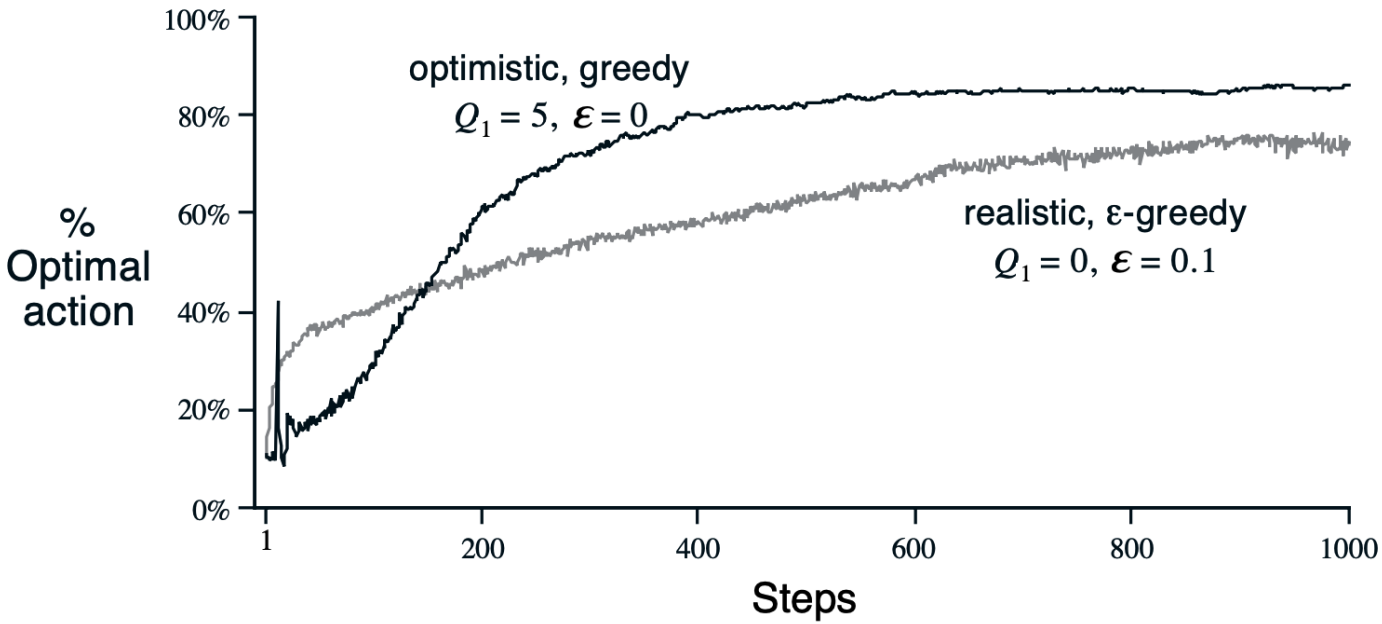
Defining the abstraction of bandits algorithms.

```
class BanditAlgoritm:
  def __init__(self):
    # define the algorithm
    return
  def initialize(self, n_arms):
    # define the internal state
    return
  def select_arm(self):
    # how to select arm
    return chosen_arm
  def update(self, chosen_arm, reward):
    # how to update the internal state
    return
```

```
class EpsilonGreedy():
  def __init__(self, epsilon, counts, values):
    self.epsilon = epsilon
    self.counts = counts
    self.values = values
    return
  def initialize(self, n_arms):
    self.counts = [0 for col in range(n_arms)]
    self.values = [0.0 for col in range(n_arms)]
    return
  def select_arm(self):
    if random.random() > self.epsilon:
      return ind_max(self.values)
    else:
      return random.randrange(len(self.values))
  def update(self, chosen_arm, reward):
    self.counts[chosen_arm] = self.counts[chosen_arm] + 1
    n = self.counts[chosen_arm]
    value = self.values[chosen_arm]
    new_value = ((n - 1) / float(n)) * value + (1 / float(n)) * reward
    self.values[chosen_arm] = new_value
    return
```

Optimistic Initial Values

1. Simple idea: initialize Q(a) to high value
2. Encourage the exploration over all possible actions early on



Softmax Bandit Algorithm

1. To learn a numerical preference for each action a (like learning a policy function, denoted as $H_t(a)$),

$$\pi_t(A_t) = P(A_t = a) = \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \tag{6}$$

2. Learning based on the idea of stochastic gradient descend

$$\text{For } A_t, H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)). \tag{7}$$

$$\text{For all } a \neq A_t, H_{t+1}(a) = H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a). \tag{8}$$

   here $\bar{R}_t$ is the average of all the rewards up to time t. Page 38 contains the full derivation.

3. Learning based on the incremental estimation

$$H_t(A_t) = H_{t-1}(A_t) + \frac{1}{N_t(A_t)}[R - H_{t-1}(A_t)] \tag{9}$$

Temperature

1. Scaling factor, temperature τ, to control the degree of exploration. High temperature, atoms will behave more random.

$$P(A_t = a) = \frac{e^{H_t(a)/\tau}}{\sum_{b=1}^{k} e^{H_t(b)/\tau}} \tag{10}$$

```
class Softmax:
  def __init__(self, temperature, counts, values):
    self.temperature = temperature
    self.counts = counts
    self.values = values
    return
  def initialize(self, n_arms):
    self.counts = [0 for col in range(n_arms)]
    self.values = [0.0 for col in range(n_arms)]
    return
  def select_arm(self):
    z = sum([math.exp(v / self.temperature) for v in self.values])
    probs = [math.exp(v / self.temperature) / z for v in self.values]
    return categorical_draw(probs)
  def update(self, chosen_arm, reward):
    self.counts[chosen_arm] = self.counts[chosen_arm] + 1
    n = self.counts[chosen_arm]
    value = self.values[chosen_arm]
    new_value = ((n - 1) / float(n)) * value + (1 / float(n)) * reward
    self.values[chosen_arm] = new_value
    return
```

Annealing

1. Annealing is the process of modifying an algorithm's behavior so that it will explore less over time
2. Effect to different algorithms:
   1. To reduce ϵ in ϵ-Greedy algorithm
   2. To make the temperature go lower and lower in Softmax Bandit algorithm

UCB - The Upper Confidence Bound Algorithm

1. Both the epsilon-Greedy algorithm and the Softmax algorithm share the following broad properties:
   1. select the arm that currently has the highest estimated value
   2. explore and choose an option that isn't the one that currently seems best
   3. reduce the exploration by annealing (dynamically change some parameters ϵ and τ
2. UCB takes a very different approach.
   1. UCB does not use randomness at all
   2. UCB doesn't have any free parameters to configure before you can deploy it.

UCB - The Upper Confidence Bound Algorithm

1. $U_t(a)$ is the upper confidence bound of the reward value, so that the true value is below with bound

$$Q(a) \leq Q_t(a) + U_t(a) \tag{11}$$

with high probability. The upper bound $U_t(a)$ is a function of $N_t(a)$; a larger number of trials $N_t(a)$ should give us a smaller bound $N_t(a)$.

2. In UCB1 algorithm,

$$U_t(a) = \sqrt{\frac{2 \log t}{N_t(a)}} \tag{12}$$

3. Thus the action is selected as

$$A_t = \arg\max_a [Q_t(a) + \sqrt{\frac{2 \log t}{N_t(a)}}] \tag{13}$$

UCB - The Upper Confidence Bound Algorithm

$$A_t = \arg\max_a [Q_t(a) + \sqrt{\frac{2 \log t}{N_t(a)}}] \tag{14}$$

1. Upper bound term brings significant values from the beginning.
2. UCB is an explicitly curiosity-driven algorithm that tries to seek out the unknown
3. Square root term is a measure of the uncertainty or variance in the estimate of a's value. Each time a is selected, the uncertainty term decreases.
4. Logarithm increases get smaller over time.

Deriving the Upper Confidence Bound

Hoeffding's Inequality

Let $X_1, ..., X_n$ be i.i.d random variables and they are all bounded by the interval $[0, 1]$. The sample mean is $\bar{X}_n = \frac{1}{n}\sum_{\tau=1}^{n} X_\tau$. Then for $u > 0$, we have:

$$P(\mathbb{E}[X] > \bar{X}_n + u) \leq e^{-2nu^2} \tag{15}$$

1. Following the Hoeffding's Inequality, then we have

$$P(Q(a) > Q_t(a) + U_t(a)) \leq e^{-2N_t(a)U_t(a)^2}$$

Deriving the Upper Confidence Bound

1. We want to pick up a bound so that with high chances the true mean is below the sample mean + the upper confidence bound,

$$P(Q(a) > Q_t(a) + U_t(a)) \leq e^{-2N_t(a)U_t(a)^2} = p,$$

$$\text{thus, } U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

2. One heuristic is to reduce the threshold p in time, as we want to make more confident bound estimation with more rewards observed. Set $p = t^{-4}$ we get UCB1 algorithm:

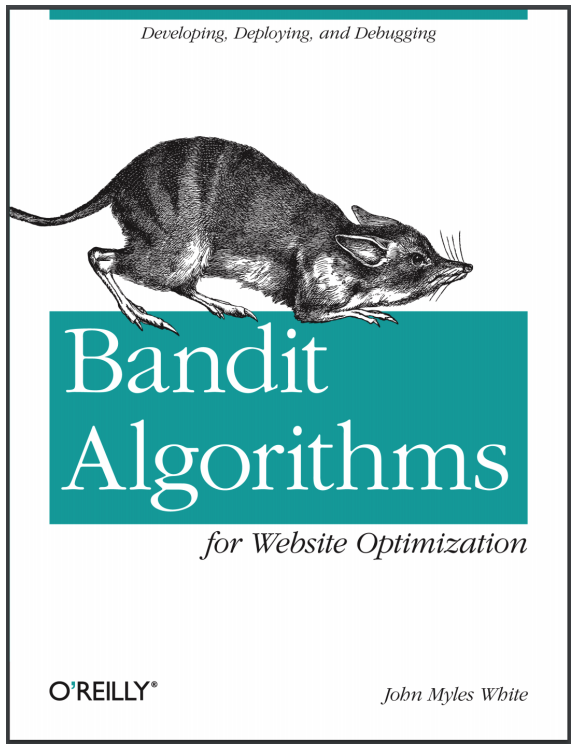
$$A_t^{UCB1} = \arg\max_a [Q_t(a) + \sqrt{\frac{2\log t}{N_t(a)}}] \tag{16}$$

```
class UCB1():
  def __init__(self, counts, values):
    self.counts = counts
    self.values = values
    return
  def initialize(self, n_arms):
    self.counts = [0 for col in range(n_arms)]
    self.values = [0.0 for col in range(n_arms)]
    return
  def select_arm(self):
    n_arms = len(self.counts)
    for arm in range(n_arms):
      if self.counts[arm] == 0:
        return arm
    ucb_values = [0.0 for arm in range(n_arms)]
    total_counts = sum(self.counts)
    for arm in range(n_arms):
      bonus = math.sqrt((2 * math.log(total_counts)) / float(self.counts[arm]))
      ucb_values[arm] = self.values[arm] + bonus
    return ind_max(ucb_values)
  def update(self, chosen_arm, reward):
    self.counts[chosen_arm] = self.counts[chosen_arm] + 1
    n = self.counts[chosen_arm]
    value = self.values[chosen_arm]
    new_value = ((n - 1) / float(n)) * value + (1 / float(n)) * reward
    self.values[chosen_arm] = new_value
    return
```

Life lessons from Bandit Algorithms

1. Trade-offs: Pure experimentation in the form of exploration is always a short-term loss, but pure profit-making in the form of exploitation is always blind to the long-term benefits of curiosity and openmindedness.
2. Take a chance: You should try everything at the start of your explorations to insure that you know a little bit about the potential value of every option. Don't close your mind without giving something a fair shot.
3. Everybody's gotta grow up sometime: You should make sure that you explore less over time. No matter what you're doing, it's important that you don't spend your whole life trying out every crazy idea that comes into your head.

Practice and Example Code



Bandit Algorithms for Website Optimization by John Myles White.

Session of Show Me the Code

Link: https://github.com/metalbubble/RLexample/tree/master/bandits

The End