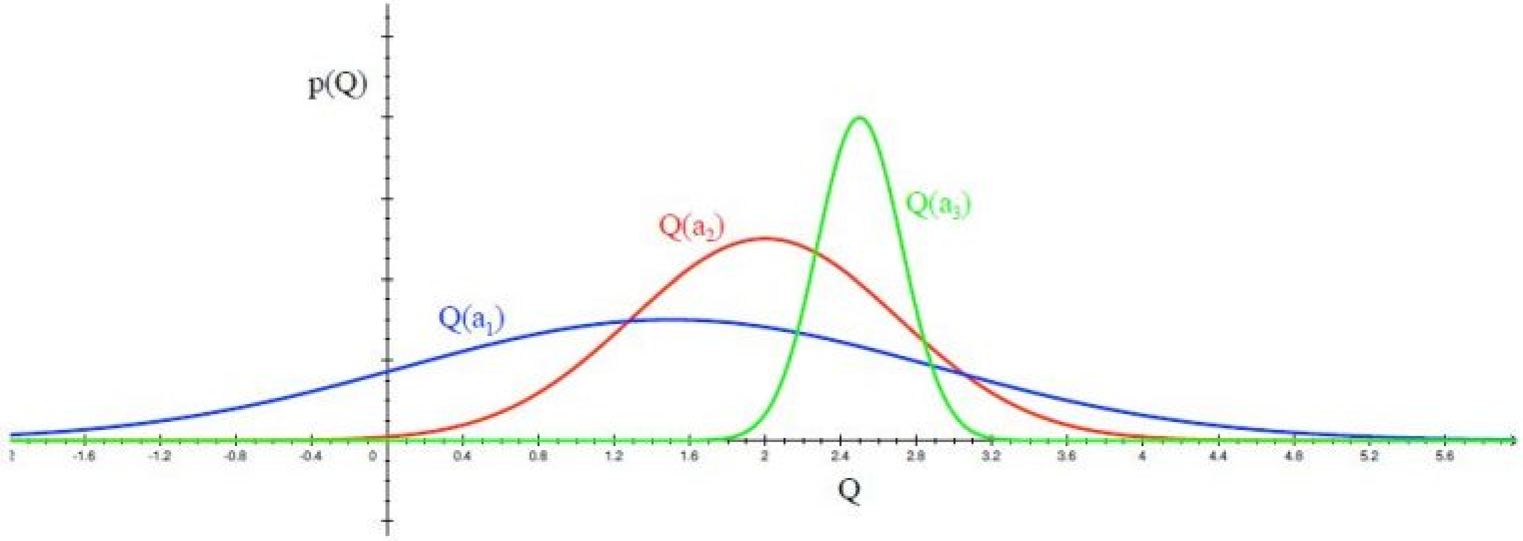
Upper Confidence Bound

1. Introduction

In UCB Algorithm we start exploring all the machines at the initial phase and later when we find the machine with highest confidence bound we start exploiting it to get maximum rewards.

Upper Confidence Bound (UCB) is the most widely used solution method for multi-armed bandit problems. This algorithm is based on the principle of optimism in the face of uncertainty.

In other words, the more uncertain we are about an arm, the more important it becomes to explore that arm.



- Distribution of action-value functions for 3 different arms a1, a2 and a3 after several trials is shown in the figure above. This distribution shows that the action value for a1 has the highest variance and hence maximum uncertainty.

- UCB says that we should choose the arm a1 and receive a reward making us less uncertain about its action-value. For the next trial/timestep, if we still are very uncertain about a1, we will choose it again until the uncertainty is reduced below a threshold.

The intuitive reason this works is that when acting optimistically in this way, one of two things happen:

- optimism is justified and we get a positive reward which is the objective ultimately

- the optimism was not justified. In this case, we play an arm that we believed might give a large reward when in fact it does not. If this happens sufficiently often, then we will learn what is the true payoff of this action and not choose it in the future.

2. Steps of UCB Algorithm

a) *At each round n, we consider two numbers for machine m.*
-> $N_m(n)$ = number of times the machine m was selected up to round n.
-> $R_m(n)$ = number of rewards of the machine m up to round n.

b) *From these two numbers we have to calculate,*
a. The average reward of machine m up to round n, $r_m(n) = R_m(n) / N_m(n)$.
b. The confidence interval [$r_m(n) — \Delta_m(n)$, $r_m(n)+\Delta_m(n)$] at round n with, $\Delta_m(n)$= sqrt(1.5 * log(n) / $N_m(n)$)

c) *We select the machine m that has the maximum UCB,* ($r_m(n)+\Delta_m(n)$)

3. UCB is actually a family of algorithms. Here, we will discuss UCB1.

Steps involved in UCB1:

- *Play each of the K actions once, giving initial values for mean rewards corresponding to each action at*
- *For each round t = K:*
- *Let $N_t(a)$ represent the number of times action a was played so far*
- *Play the action at maximising the following expression:*

$$Q(a) + \sqrt{\frac{2 \log t}{N_t(a)}}$$

- *Observe the reward and update the mean reward or expected payoff for the chosen action*

We will not go into the mathematical proof for UCB. However, it is important to understand the expression that corresponds to our selected action. Remember, in the random exploration we just had Q(a) to maximise, while here we have two terms. First is the action value function, while the second is the confidence term.

- Each time a is selected, the uncertainty is presumably reduced: Nt(a) increments, and, as it appears in the denominator, the uncertainty term decreases.

$$N_t(a) \uparrow \qquad \sqrt{\frac{2\log t}{N_t(a)}} \downarrow$$

- On the other hand, each time an action other than a is selected, t increases, but Nt(a) does not; because t appears in the numerator, the uncertainty estimate increases.

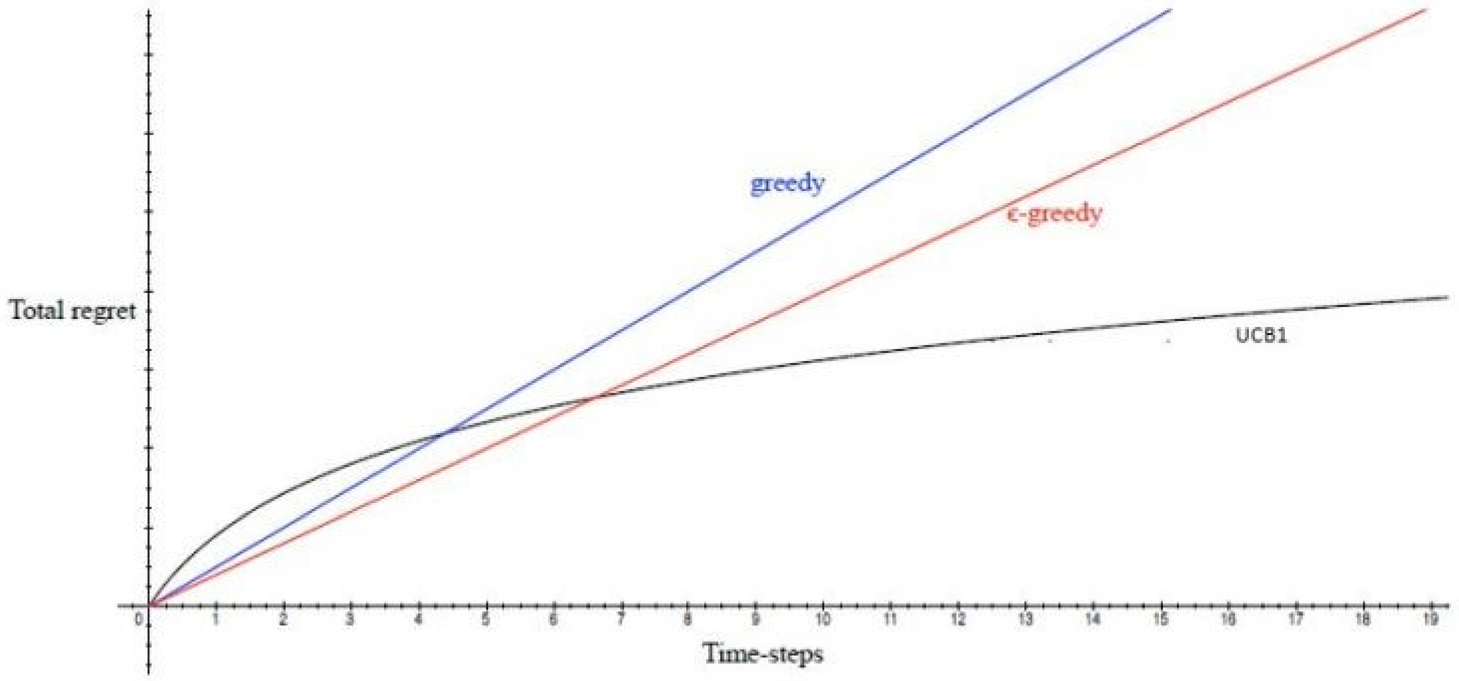
$$t \uparrow \qquad \overset{\text{With } N_t(a)}{\text{constant}} \qquad \sqrt{\frac{2\log t}{N_t(a)}} \uparrow$$

- The use of the natural logarithm means that the increases get smaller over time; all actions will eventually be selected, but actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency over time.

- This will ultimately lead to the optimal action being selected repeatedly in the end.

4. Regret Comparison

Among all the algorithms given in this article, only the UCB algorithm provides a strategy where the regret increases as log(t), while in the other algorithms we get linear regret with different slopes.



5. Non-Stationary Bandit problems

An important assumption we are making here is that we are working with the same bandit and distributions from which rewards are being sampled at each timestep stays the same. This is called a stationary problem. To explain it with another example, say you get a reward of 1 every time a coin is tossed, and the result is head. Say after 1000 coin tosses due to wear and tear the coin becomes biased then this will become a non-stationary problem.

To solve a non-stationary problem, more recent samples will be important and hence we could use a constant discounting factor alpha and we can rewrite the update equation like this:

$$Q_t(a) = Q_{t-1}(a) + \frac{1}{\alpha}(R_t - Q_{t-1}(a))$$

Note that we have replaced Nt(at) here with a constant alpha, which ensures that the recent samples are given higher weights, and increments are decided more by such recent samples.

There are other techniques which provide different solutions to bandits with non-stationary rewards.

6. Python Implementation from scratch for Ad CTR Optimization

As mentioned in the use cases section, MABP has a lot of applications in the online advertising domain.

Suppose an advertising company is running 10 different ads targeted towards a similar set of population on a webpage.

Each column index represents a different ad. We have a 1 if the ad was clicked by a user, and 0 if it was not.

A sample from the original dataset is shown below:

Ad 1	Ad 2	Ad 3	Ad 4	Ad 5	Ad 6	Ad 7	Ad 8	Ad 9	Ad 10
1	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0

This is a simulated dataset and it has Ad #5 as the one which gives the maximum reward.

First, we will try a random selection technique, where we randomly select any ad and show it to the user. If the user clicks the ad, we get paid and if not, there is no profit.

```
# Random Selection

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('F:/Jan-May___2023/CS-317__slides/Python_Programs_tutorials/Ads_Optimisation.csv')

# Implementing Random Selection
import random
N = 10000
d = 10
ads_selected = []
total_reward = 0
for n in range(0, N):
    ad = random.randrange(d)
    ads_selected.append(ad)
    reward = dataset.values[n, ad]
    total_reward = total_reward + reward

print(pd.Series(ads_selected).tail(1000).value_counts(normalize=True))
```

Total reward for the random selection algorithm comes out to be 1170. As this algorithm is not learning anything, it will not smartly select any ad which is giving the maximum return. And hence even if we look at the last 1000 trials, it is not able to find the optimal ad.

Output:

```
8    0.111
1    0.109
0    0.107
2    0.105
5    0.099
4    0.098
6    0.097
9    0.097
3    0.094
7    0.083
dtype: float64
```

Now, let’s try the Upper Confidence Bound algorithm to do the same:

```
# Implementing UCB
import math
N = 10000
d = 10
ads_selected = []
numbers_of_selections = [0] * d
sums_of_reward = [0] * d
total_reward = 0

for n in range(0, N):
    ad = 0
    max_upper_bound = 0
    for i in range(0, d):
        if (numbers_of_selections[i] > 0):
            average_reward = sums_of_reward[i] / numbers_of_selections[i]
            delta_i = math.sqrt(2 * math.log(n+1) / numbers_of_selections[i])
            upper_bound = average_reward + delta_i
        else:
            upper_bound = 1e400
        if upper_bound > max_upper_bound:
            max_upper_bound = upper_bound
            ad = i
    ads_selected.append(ad)
    numbers_of_selections[ad] += 1
    reward = dataset.values[n, ad]
    sums_of_reward[ad] += reward
    total_reward += reward

print('Total Reward: ',total_reward)
print(pd.Series(ads_selected).tail(1500).value_counts(normalize=True))
```

The *total_reward* for UCB comes out to be 2125. Clearly, this is much better than random selection and indeed a smart exploration technique that can significantly improve our strategy to solve a MABP.

Output:

```
Total Reward:  2125

4    0.810000
0    0.077333
7    0.026667
3    0.024667
2    0.019333
6    0.019333
1    0.007333
8    0.006000
5    0.004667
9    0.004667
dtype: float64
```

After just 1500 trials, UCB is already favouring Ad #5 (index 4) which happens to be the optimal ad, and gets the maximum return for the given problem.

Bibliography:

1. https://medium.com/towards-data-science/multi-armed-bandits-upper-confidence-bound-algorithms-with-python-code-a977728f0e2d
2. https://medium.com/analytics-vidhya/implementation-of-upper-confidence-bound-algorithm-ce0651b63c15
3. https://www.analyticsvidhya.com/blog/2018/09/reinforcement-multi-armed-bandit-scratch-python/
4. http://www.sefidian.com/2019/12/07/upper-confidence-bound-ucb-algorithm-explained/
5. https://github.com/topics/upper-confidence-bounds?l=python
6. https://towardsdatascience.com/the-upper-confidence-bound-ucb-bandit-algorithm-c05c2bf4c13f
7. https://www.geeksforgeeks.org/upper-confidence-bound-algorithm-in-reinforcement-learning/
8. https://arxiv.org/abs/0805.3415