Reinforcement Learning

Reinforcement learning (RL) is defined as a sub-field of machine learning that enables AI-based systems to take actions in a dynamic environment through trial and error methods to maximize the collective rewards based on the feedback generated for respective actions. This article explains reinforcement learning, how it works, its algorithms, and some real-world uses.

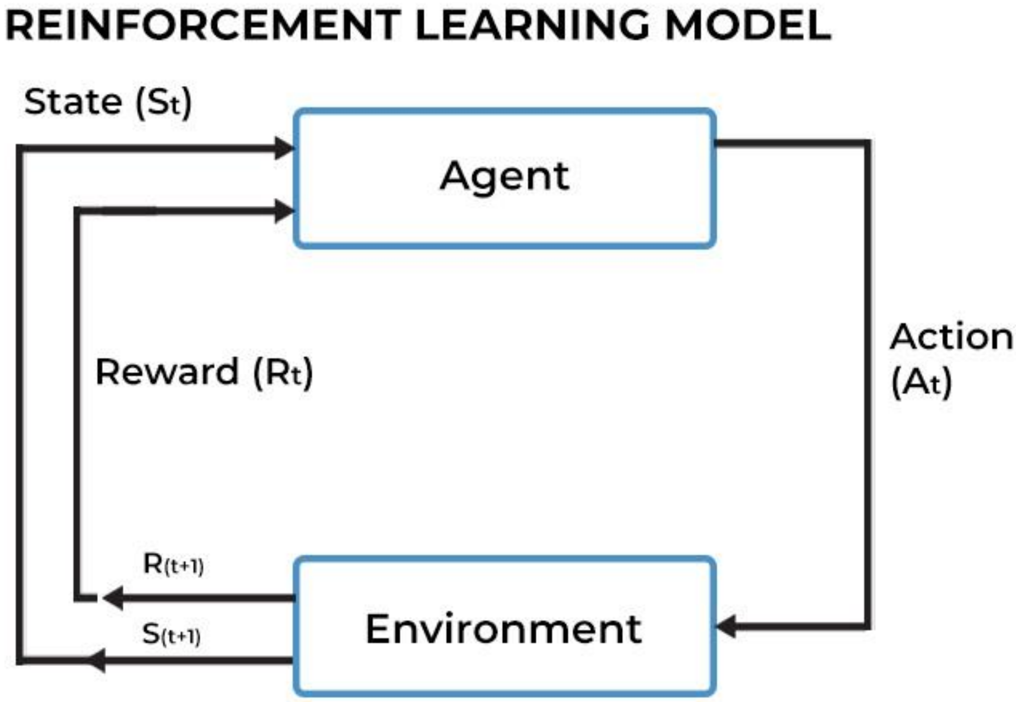
Reinforcement Learning is a powerful branch of Machine Learning. It is used to solve interacting problems where the data observed up to time t is considered to decide which action to take at time t + 1. It is also used for Artificial Intelligence when training machines to perform tasks such as walking. Desired outcomes provide the AI with reward, undesired with punishment. Machines learn through trial and error.

RL optimizes AI-driven systems by imitating natural intelligence that emulates human cognition. Such a learning approach helps computer agents make critical decisions that achieve astounding results in the intended tasks without the involvement of a human or the need for explicitly programming the AI systems.
Some known RL methods that have added a subtle dynamic element to conventional ML methods include Monte Carlo, state–action–reward–state–action (SARSA), and Q-learning**.** AI models trained over reinforcement learning algorithms have defeated human counterparts in several video games and board games, including chess and Go.
Technically, RL implementations can be classified into three types:

- **Policy-based**: This RL approach aims to maximize the system reward by employing deterministic policies, strategies, and techniques.
- **Value-based**: Value-based RL implementation intends to optimize the arbitrary value function involved in learning.
- **Model-based**: The model-based approach enables the creation of a virtual setting for a specific environment. Moreover, the

participating system agents perform their tasks within these virtual specifications.

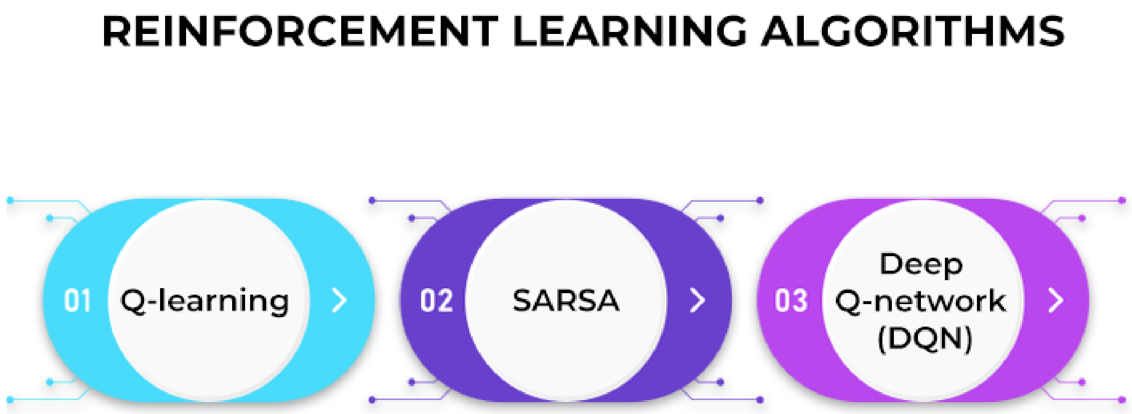
A typical reinforcement learning model can be represented by:



Here are some important terms used in Reinforcement AI:

- **Agent:** It is an assumed entity which performs actions in an environment to gain some reward.
- **Environment (e):** A scenario that an agent has to face.
- **Reward (R):** An immediate return given to an agent when he or she performs specific action or task.
- **State (s):** State refers to the current situation returned by the environment.
- **Policy (π):** It is a strategy which applies by the agent to decide the next action based on the current state.
- **Value (V):** It is expected long-term return with discount, as compared to the short-term reward.
- **Value Function:** It specifies the value of a state that is the total amount of reward. It is an agent which should be expected beginning from that state.
- **Model of the environment:** This mimics the behaviour of the environment. It helps you to make inferences to be made and also determine how the environment will behave.

- **Model based methods:** It is a method for solving reinforcement learning problems which use model-based methods.
- **Q value or action value (Q):** Q value is quite similar to value. The only difference between the two is that it takes an additional parameter as a current action.



Types of Reinforcement Learning

Two types of reinforcement learning methods are:

Positive:

It is defined as an event, that occurs because of specific behavior. It increases the strength and the frequency of the behavior and impacts positively on the action taken by the agent.

This type of Reinforcement helps you to maximize performance and sustain change for a more extended period. However, too much Reinforcement may lead to over-optimization of state, which can affect the results.

Negative:

Negative Reinforcement is defined as strengthening of behavior that occurs because of a negative condition which should have stopped or avoided. It helps you to define the minimum stand of performance.

However, the drawback of this method is that it provides enough to meet up the minimum behavior.

Learning Models of Reinforcement

There are two important learning models in reinforcement learning:

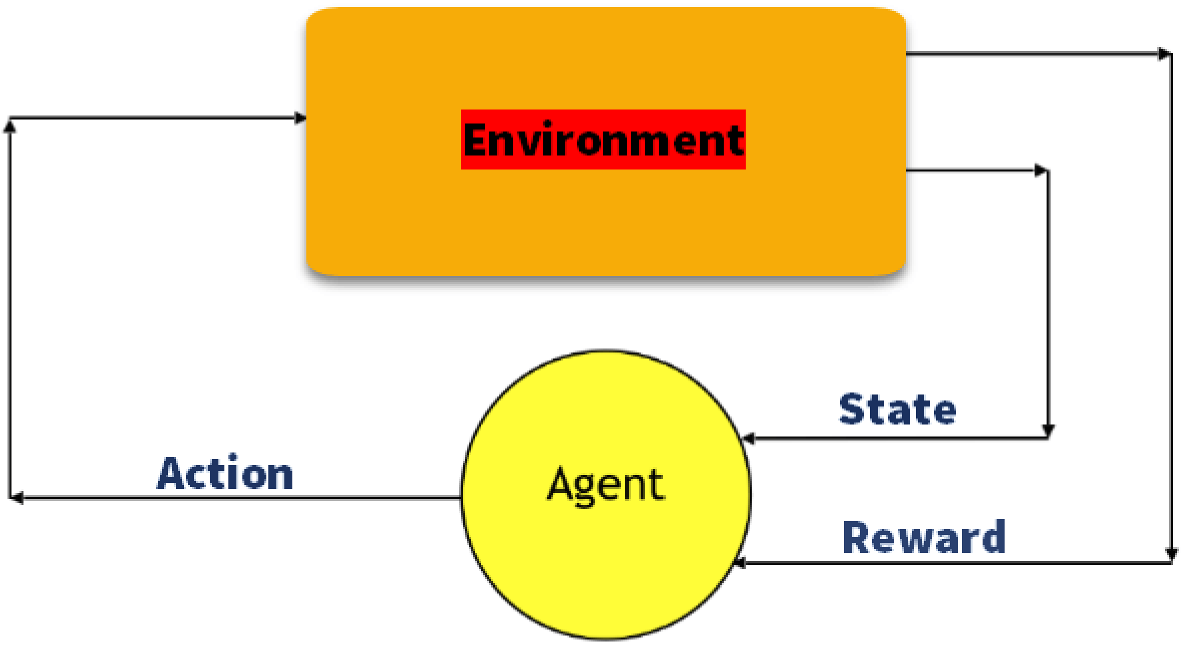
- Markov Decision Process
- Q learning

Markov Decision Process

The following parameters are used to get a solution:

- Set of actions- A
- Set of states -S
- Reward- R
- Policy- n
- Value- V

The mathematical approach for mapping a solution in reinforcement Learning is recon as a Markov Decision Process or (MDP).



Q-Learning

Q-learning is an off-policy and model-free type algorithm that learns from random actions (greedy policy). ‘Q’ in Q-learning refers to the quality of activities that maximize the rewards generated through the algorithmic process.

Policy iteration refers to policy improvement or refinement through actions that amplify the value function. In a value iteration, the values of the value function are updated. Mathematically, Q-learning is represented by the formula:

$Q(s,a) = (1-\alpha).Q(s,a) + \alpha.(R + \gamma.max(Q(S2,a))$.

Where,

alpha = learning rate,

gamma = discount factor,

R = reward,

S2 = next state.

Q(S2,a) = future value.

Markov Decision Processes

- Markov decision processes formally describe an environment for reinforcement learning.
- Where the environment is fully observable i.e. The current state completely characterises the process
- Almost all RL problems can be formalised as MDPs, e.g. Optimal control primarily deals with continuous MDPs
- Partially observable problems can be converted into MDPs
- Bandits are MDPs with one state

“The future is independent of the past given the present”

Definition: A state St is Markov if and only if

P [St+1 | St] = P [St+1 | S1, ..., St]

- The state captures all relevant information from the history

- Once the state is known, the history may be thrown away
- i.e. The state is a sufficient statistic of the future

State Transition Matrix:

For a Markov state s and successor state s' , the state transition probability is defined by

Pss' = P [St+1 = s' | St = s] State transition matrix P defines transition probabilities from all states s to all successor states s',

$$\mathcal{P} = \begin{array}{cc} & \textit{to} \\ \textit{from} & \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix} \end{array}$$

where each row of the matrix sums to 1.

Markov Process:

A Markov process is a memoryless random process, i.e. a sequence of random states S1, S2, ... with the Markov property.

Definition

A Markov Process (or Markov Chain) is a tuple (S,P)

S is a (finite) set of states

P is a state transition probability matrix,

Pss' = P [St+1 = s' | St = s]

Multi-Armed Bandit Problem

The multi-armed bandit problem is a problem in which a decision maker iteratively selects one of multiple fixed choices when the properties of each choice are only partially known. The problem is named after a gambler who must choose which of slot machines to play. The goal is to maximize the expected reward or payoff over time. The problem arises in various fields such as probability theory, machine learning, and resource allocation.

The **multi-armed bandit problem** is a fascinating concept in probability theory and machine learning. Imagine a gambler standing in front of a row of slot machines (sometimes called "one-armed bandits"). The gambler has to decide which machines to play, how many times to play each machine, and in which order to play them. The goal is to maximize the total rewards earned through a sequence of lever pulls. Let's dive into the details:

1. **Problem Description**:
    - The multi-armed bandit problem involves a decision maker who iteratively selects one of multiple fixed choices (referred to as "arms" or "actions").
    - The properties of each choice are only partially known at the time of allocation and may become better understood as time passes.
    - Importantly, choosing an arm does not affect the properties of that arm or other arms.

The Multi-Armed Bandit Problem

- We have d arms. For example, arms are ads that we display to users each time they connect to a web page.
- Each time a user connects to this web page, that makes a round.
- At each round n, we choose one ad to display to the user.
- At each round n, ad i gives reward $r_i(n) \in \{0, 1\}$: $r_i(n) = 1$ if the user clicked on the ad i, 0 if the user didn't.
- Our goal is to maximize the total reward we get over many rounds.

Upper Confidence Bound Algorithm

Step 1. At each round n, we consider two numbers for each ad i:

- $N_i(n)$ - the number of times the ad i was selected up to round n,
- $R_i(n)$ - the sum of rewards of the ad i up to round n.

Step 2. From these two numbers we compute:

- the average reward of ad i up to round n

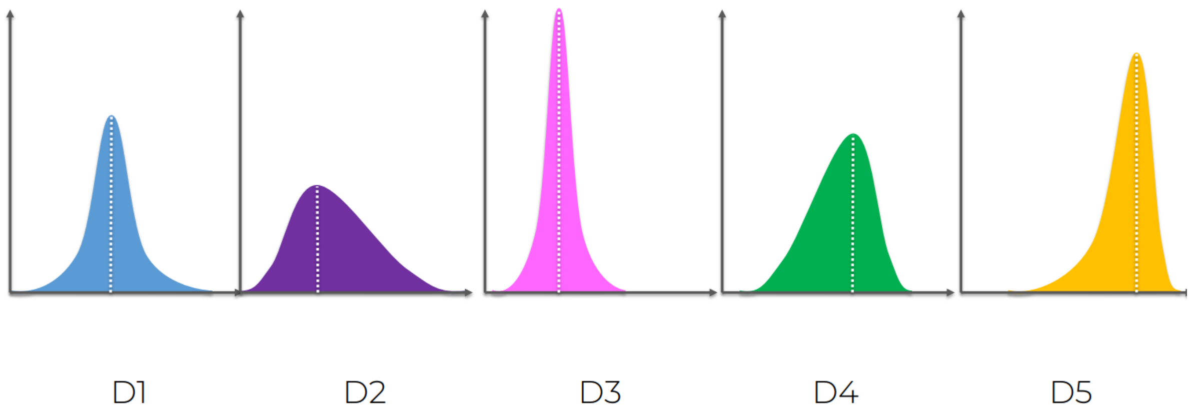
$$\bar{r}_i(n) = \frac{R_i(n)}{N_i(n)}$$

- the confidence interval $[\bar{r}_i(n) - \Delta_i(n), \bar{r}_i(n) + \Delta_i(n)]$ at round n with

$$\Delta_i(n) = \sqrt{\frac{3}{2} \frac{\log(n)}{N_i(n)}}$$

Step 3. We select the ad i that has the maximum UCB $\bar{r}_i(n) + \Delta_i(n)$.

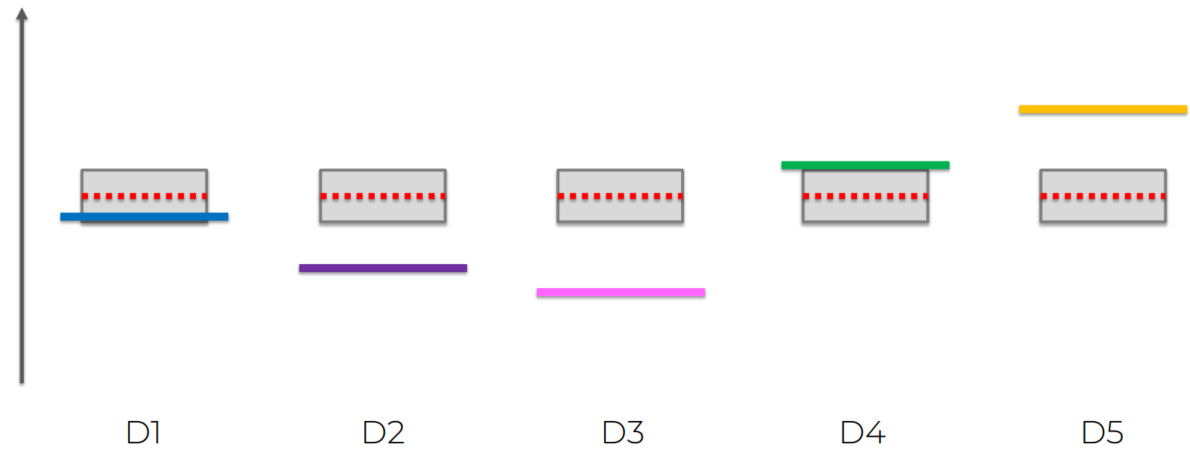
Upper Confidence Bound Algorithm



Which one is getting the most click.

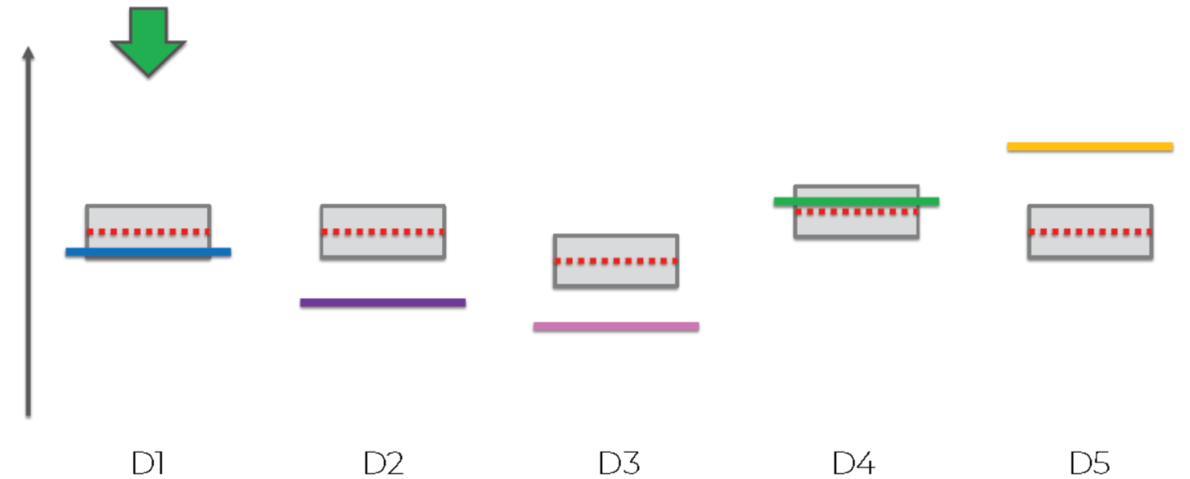
Vertical axis put them horizontally.

Upper Confidence Bound Algorithm

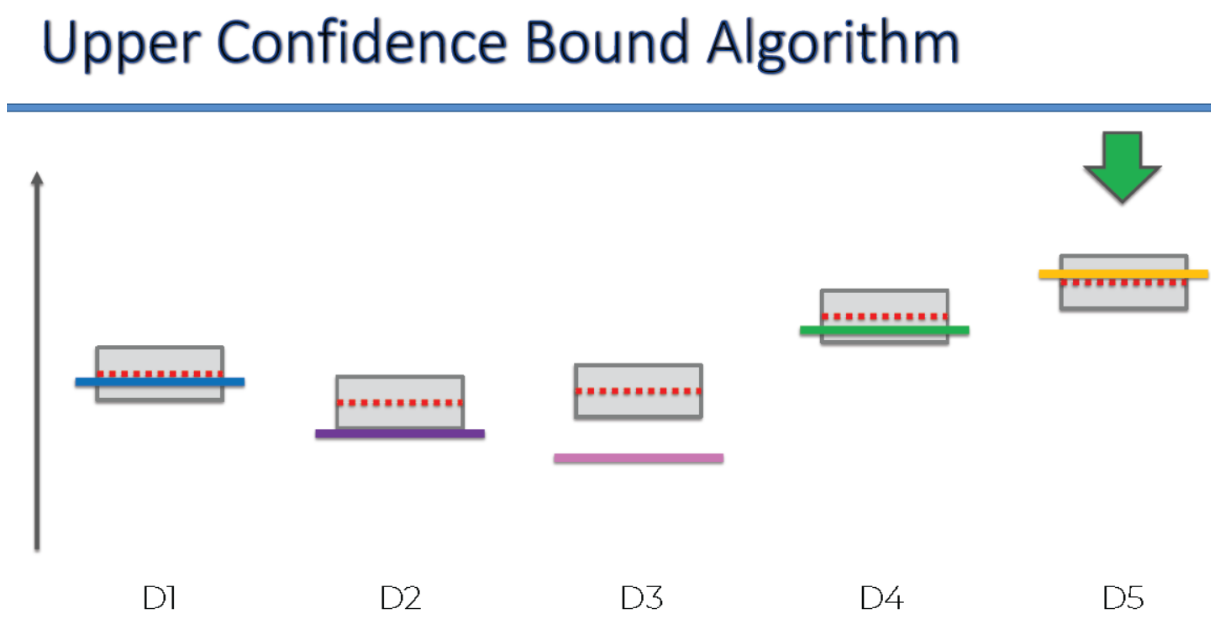


Vertical axis put them horizontally.

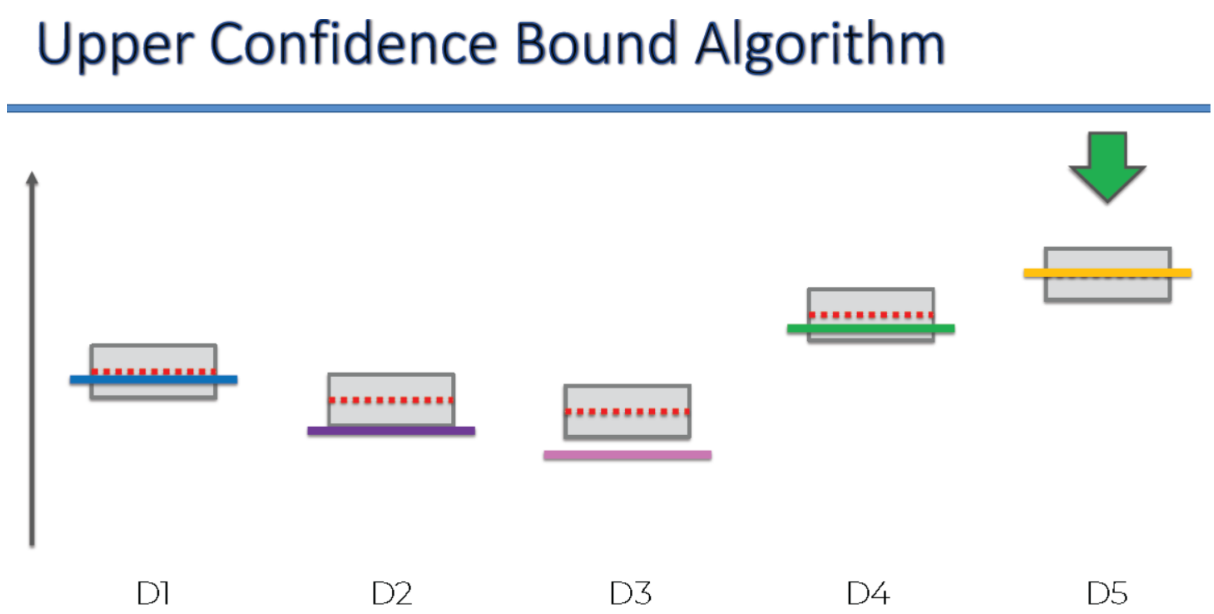
Upper Confidence Bound Algorithm



Create a confidence band will include actual return or expected return. We pick the machine with highest confidence bound. Red dotted line is observed average. Confidence interval become smaller. Observed average value long run converge to expected actual return.



D5 very close to final solution. Exploring this one because we found out that is the best one.



That how it solves the multi arm bandit problem.

Here we are implementing UCB (Upper Confidence bound) algorithms. It is one of the most exciting branches of ML. It’s one of the closes to Artificial Intelligence in the sense we are making some program that make action just like Robot. Here we will implement two of the best Reinforcement Learning Model which are UCB and Thompson Sampling. We will implement UCB applied on Business case study. Brand new SUV car, the car company trying to optimize the targeting things to classification. This time we will optimize online advertising meaning we are going to find best Ad among the different advertisement. The best Ad it will convert maximum customer click on that Ad.

Lets start upper confidence bound:

Dataset: Sale the SUV Car. Optimize the click through rate some Advertisement make for this car. Advertisement prepare 10 different Ad mean 10 different design. Advertiser team wonder which Ad is attract most people. Click the Ad then potentially SUV. 10 different Ad process online. Online connect through some website or search engine. One of the Ad each time connect the webpage and record the result user click yes or no the Ad. In our data lot of user, 10 thousand users. It’s a real time process mean dynamic process. Simulation exactly given by the dataset. User click any Ad then 1 otherwise 0. This is the only way UCB algorithm rum on simulation to figure out highest convergent rate. Identify that’s the Ad, user click the most. We need to figure out minimum number of round which

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
0	0	0	0	0	0	0	1	0	0
0	0	0	0	1	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	1	0	0
0	0	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	0

Upper Confidence Bound (UCB) in Python:

Importing the libraries

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Importing the dataset

```
dataset = pd.read_csv('Ads_CTR_Optimisation.csv')
```

Implementing UCB

```
import math
N = 10000
d = 10
ads_selected = []
numbers_of_selections = [0] * d
sums_of_rewards = [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_rewards[i] /
numbers_of_selections[i]
            delta_i = math.sqrt(3/2 * 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] = numbers_of_selections[ad] + 1
    reward = dataset.values[n, ad]
    sums_of_rewards[ad] = sums_of_rewards[ad] + reward
    total_reward = total_reward + reward
```

**

import math → use of square root function;
N = 10000 →total number of users;
Here d = 10 → number of Ads;
ads_selected = [] → that is empty list initialize for total Ads selected from the full list;

numbers_of_selections = [0] * d → **Number selection initialize with ten zeroes but expressed as multiple with d mean 10; list as list of ten zeroes.**

sums_of_rewards = [0] * d → **each Ad sum of rewards initial zero because no Ad was selected initial.**

total_reward = 0 →(final variable)total reward accumulated over the round; after 1^{st} round no reward collected;

for n in range(0, N): → Iterative for loop start from 1^{st} user to 10000^{th} user mean last user but python index start from 0;

ad = 0 → initialize 1^{st} Ad; we need each of the Ad upper confidence bound;

max_upper_bound = 0 → introduce new variable maximum upper confidence bound;

for i in range(0, d): → second for loop iterate from Ad1 to Ad10 d=10;

then we implement step-2 from Upper Confidence Bound Algorithm

start with UCB Algorithm Step:2

Step 2. From these two numbers we compute:

- the average reward of ad i up to round n

$$\bar{r}_i(n) = \frac{R_i(n)}{N_i(n)}$$

- the confidence interval $[\bar{r}_i(n) - \Delta_i(n), \bar{r}_i(n) + \Delta_i(n)]$ at round n with

$$\Delta_i(n) = \sqrt{\frac{3}{2}\frac{\log(n)}{N_i(n)}}$$

if $N_i(n)$= Zero then Bar $r_i(n)$is infinity, meaning less

if (numbers_of_selections[i] > 0): → therefore at least Ad selected. Mean $N_i(n)>0$, Average of the reward at least selected.

average_reward = sums_of_rewards[i] / numbers_of_selections[i] → that is the average reward of ad I up to round n;

Now we will compute confidence interval:

delta_i = math.sqrt(3/2 * math.log(n + 1) / numbers_of_selections[i])

- → Sqrt function use then math.log mean logarithm function use but value of n start in for loop is 0 to N. If 0 then valuie of log(0) mean – infinite. It is very dangerous that why we write log(n+1) mean if n=0 then log(1) mean 0.

Now final value we have to compute:

Step 3. We select the ad i that has the maximum UCB $\bar{r}_i(n) + \Delta_i(n)$.

```
So, average reward + delta I (confidential interval)
upper_bound = average_reward + delta_i

second for loop we are implementing Step-2 but step-3 not because this
step-3 select the ad I that has maximum UCB.
UCB Algorithm we have to atlest select Ad in first round.
else: → ad not been selected yet. We have to do Ad must be selected.
upper_bound = 1e400 → upper_bound select supper high value 10 to the
power of 400. So, that we have not selected then maximum upper bound
should be there.

Now implementing step-3 maximum of UCB:



Now we will finish the step-3 by updating the variables:
ads_selected.append(ad) → update the ads_selected variable;
numbers_of_selections[ad] = numbers_of_selections[ad] + 1 → number of
selection increment by 1;

reward = dataset.values[n, ad] → reward that is collected after
showing the ad of user n. reward is collected each round;
sums_of_rewards[ad] = sums_of_rewards[ad] + reward →cumulated reward
    total_reward = total_reward + reward → total reward we get upto
round n. so, update total reward variable update after add reward.
```

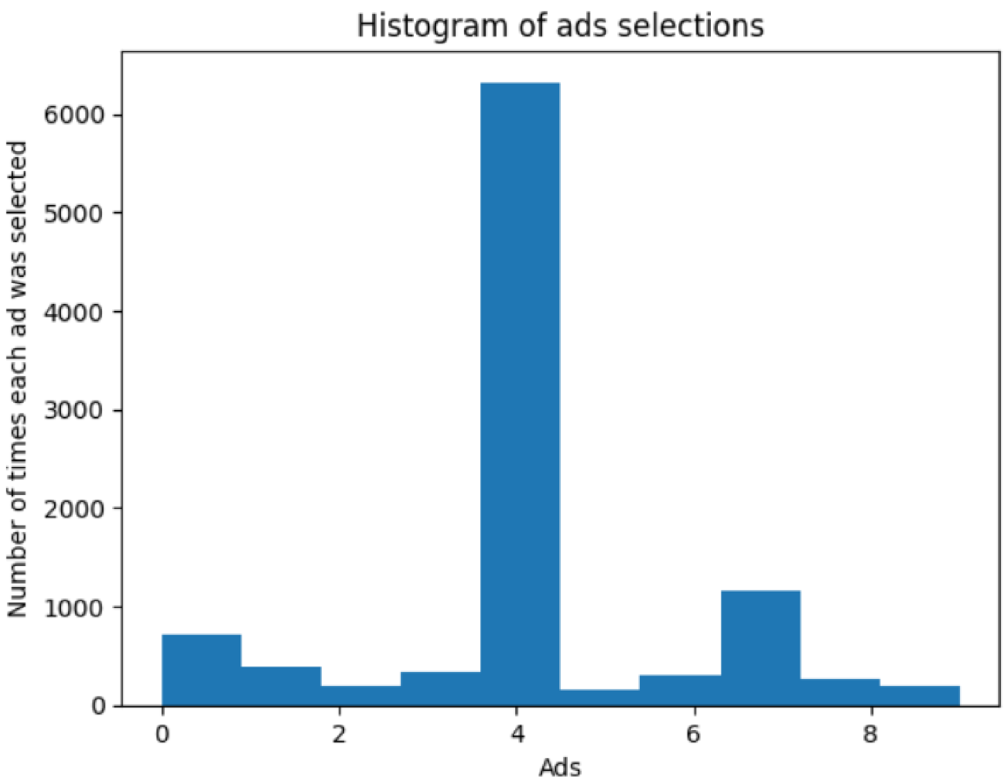
Visualising the results

```
plt.hist(ads_selected)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Number of times each ad was selected')
plt.show()
```

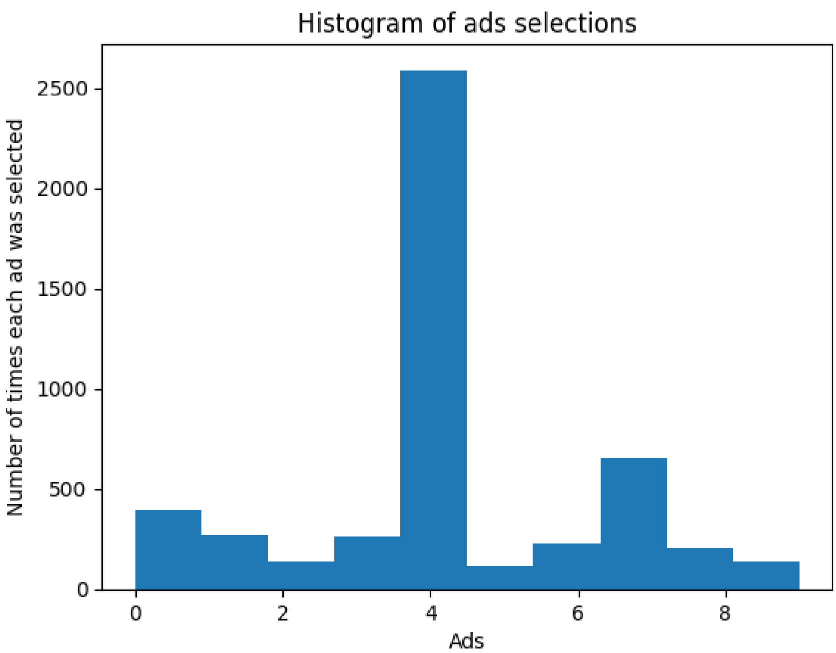
**** Histogram plots each of the Ad (index 0 to 9) number time selected.**

```
plt.hist(ads_selected) → histogram function and parameter is
ads_selected mean Ads Index 0 to 9 round 10,000.

Output is below:
```

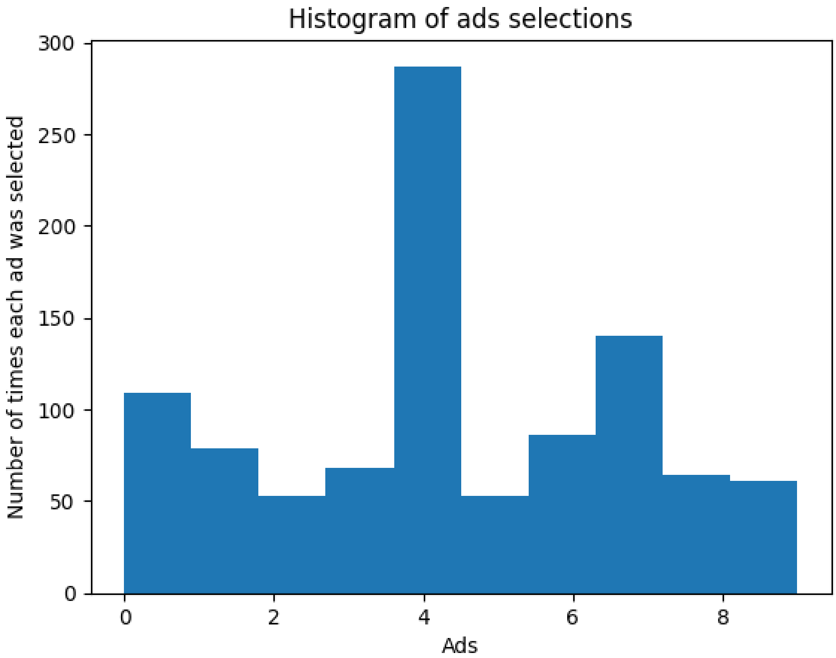


We should experience to see actually how many round UCB algorithm was able to identify this Ad highest CT Bar. The way to check this change the value of N (number of rounds). Here this algorithm with run 10,000 rounds. Now we will change the value N to 5,000. Then output:

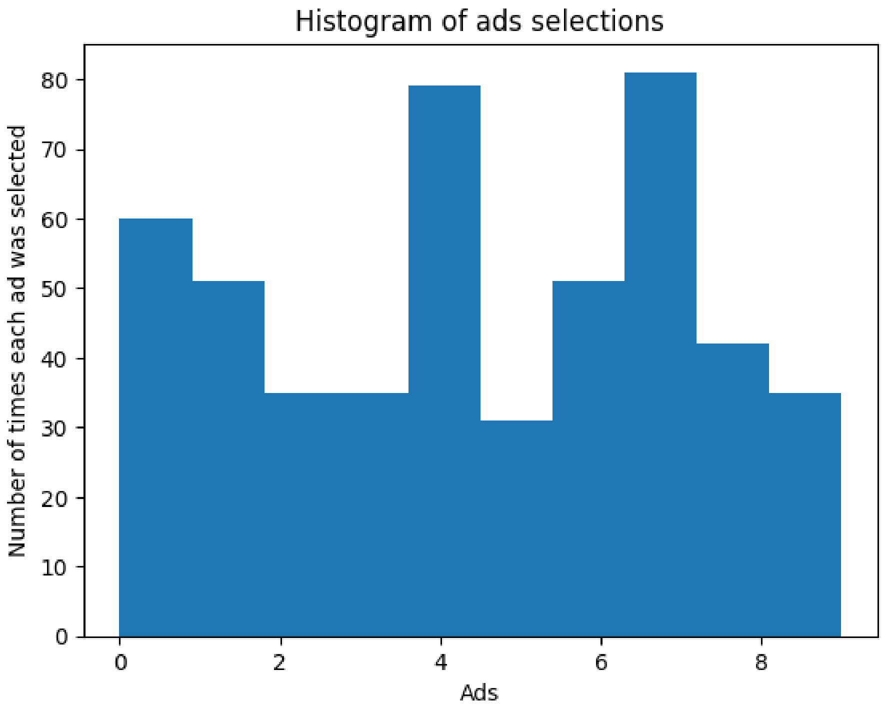


Even UCB can identify the highest CT Ad.

Replace 5000 by 1000 then restart and run:



Still able to identify the Ad. Now try for 500. Restart and run:



So, 500 round is not enough for UCB Algorithm to identify best Ad with highest CTR. Here we can't identify because highest CTR is Ad7. 500 round not enough for UCB.

Finally we will see Thomson Sampling can identify or not.