Chapter 6 : Part 1

**Reinforcement Learning**

**UCB: Upper Confidence Bound**

**6.1.1 Reinforcement Learning**

Reinforcement Learning(RL) is a type of machine learning technique that enables an agent to *learn* in an *interactive* *environment* by trial anderror using feedback from its *own* *actions* and *experiences*.

* Though both ***supervised*** and ***reinforcement*** ***learning*** use mapping between ***input*** and ***output***, unlike supervised learning where the feedback provided ***to the agent*** is correct set of actions for performing a task, reinforcement learning uses REWARDS and PUNISHMENTS as signals for ***positive*** and ***negative*** behavior.
* As compared to unsupervised learning, ***reinforcement*** ***learning*** is different in terms of goals. While the goal in ***unsupervised learning*** is to ***find similarities*** and ***differences*** between ***data-points***, in the case of *reinforcement learning* the goal is to *find* a *suitable action model* that would *maximize the total* cumulativereward of the agent.
* Reinforcement learning is an area of Machine Learning. It is about taking suitable action to maximize reward in a particular situation. It is employed by various software and machines to find the best possible behavior or path it should take in a specific situation.
* *Reinforcement* *learning* differs from supervised learning in a way that in *supervised* *learning* the ***training*** ***data*** has the ***answer*** ***key*** ***with it*** so the model is ***trained*** ***with*** the ***correct*** ***answer*** itself whereas in reinforcement learning, there is no answer but the reinforcement agent decides what to do to perform the given task. In the absence of a training dataset, it is bound to learn from its experience.

**6.1.2 the Multi-Armed Bandit problem**

|  |  |
| --- | --- |
| * Multi-armed bandit: In probability theory and machine learning, the Multi-Armed Bandit Problem (sometimes called the *K or N-arme*d *bandit* *problem*) is a problem in which * A ***fixed limited set of resources*** must be ***allocated between competing (alternative) choices*** in a way that ***maximizes*** ***their*** ***expected*** ***gain***, when * Each ***choice's properties*** are only ***partially*** ***known*** at the ***time*** of ***allocation***, and may become better understood as time passes or by allocating resources to the choice. | C:\Users\RTTI\Pictures\task-0000000902-cfda1c60_HmUi8CF.jpg |

* Classic Reinforcement Learning Problem: This is a classic reinforcement learning problem that exemplifies the exploration–exploitation tradeoff dilemma. The name comes from imagining
* A gambler at a ***row*** of ***slot machines*** (Those *slot machines* are sometimes known as ***"one-armed bandits"***), who has to decide which machines to play,
* how many ***times*** to play ***each*** ***machine*** and
* in ***which*** ***order*** to play them, and
* whether to ***continue*** with the ***current*** machine or ***try*** a ***different*** machine.

The multi-armed bandit problem also falls into the broad category of stochastic scheduling.

|  |  |
| --- | --- |
| * What is a multi armed bandit: First thing that comes to mind is like a robber going into a bank and so on. But actually a bandit or more specifically *one armed bandit* is a ***slot******machine*** (gambling). * Why is it called the one arm bandit: Notice the *handle* on the right, you have to *pull* that lever to *initiate* the game (today most of those slot machines are electronic and you have to push the button). * These machines and you, there are ***50-50 chance to win/lose***. But in some casino this machines are designed with a bug, so that the user lose their money very fast (i.e. more than 50% chance to lose). And they became the quickest way to lose your money in a casino. Hence the name bandit because it was basically robbing you. |  |

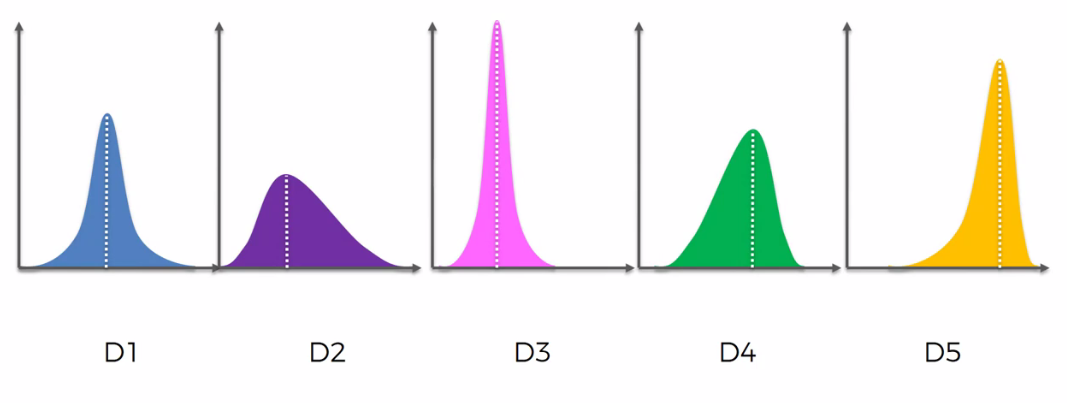


* Since those slot machines has one arm/handle they are called One Armed Bandit. For more than one slot machine, i.e. if you play with a series of those slot machines they become ***Multi-Armed Bandit***.
* The ***Multi-Armed Bandit problem*** is kind of the challenge that a person is faced when he comes up to a whole set of these machines. This is the classic example. But there are more problems similar to this which are also called ***Multi-Armed Bandit problem***. Consider the Baby Robot problem given below.
* Another Example: *Baby Robot* is lost in the *mall*. Using *Reinforcement Learning* we want to help him find his *way back* to his *mum*. However, before he can even begin looking for her, he needs to recharge, from a set of power sockets that each give a slightly different amount of charge.

|  |  |
| --- | --- |
| * Using the Strategies from the Multi-Armed Bandit Problem we need to find the ***best*** ***socket***, in the ***shortest*** ***amount*** ***of*** ***time***, to allow Baby Robot to get charged up and on his way. * Baby Robot has entered a ***charging*** ***room*** containing ***5 different power sockets***. Each of these sockets returns a ***slightly*** ***different*** ***amount*** of ***charge***. We want to get ***Baby*** ***Robot*** charged up in the ***minimum*** ***amount*** of ***time***, so we need to ***locate*** ***the*** ***best*** ***socket*** and then use it until charging is complete. |  |

* This is identical to the ***Multi-Armed Bandit problem*** except that, instead of looking for a ***slot machine*** that gives the ***best*** ***payout***, we’re looking for a ***power*** ***socket*** that gives the ***most*** ***charge***.
* The Multi-Armed Bandit problem: Assume that you've got five of these machines. Your want to play them to *maximize* your *return* from the *number* *of* *games* that you can *actually* *play*. You decided how many times you're going to play, 100 times or 1000 times and you want to maximize return.
* How do you figure out ***which ones*** of them to play in order to ***maximize*** your ***returns***.
* The assumption here is: Each one of these machines has a distribution behind it (So there's a distribution of numbers for outcomes. You pull the trigger and it just picks out randomly out of its distributed numbers).

1. Each machine has different distribution. Sometimes it can be *similar* in *some* of the *machines* but by default they are *different*.
2. You don't know these distribution.
3. Your goal is to figure out which of these distributions is the best one for you. With *minimum* *try* you have to figure out the *best* *distribution*.



* The best Distribution: The orange one is the best machine because:
* It's the most *left skewed* because the *tails on the left*.
* So it's got the most favorable outcomes.
* The highest ***mean***, ***median*** and ***mode***.

If you knew these distributions, you can just go to the fifth machine and get the maximum outcome. But you don't know that in advance.

* And your goal is to ***figure out the best distribution for you***, with *minimum* *trial*. So there are these two factors that are in play:

1. ***Exploration*** : You don't know which one of these machine is the best, you're going to figure it out.
2. ***Exploitation*** : But at the same time you are already spending your money doing each machine **Trial**.

i.e. the longer you take to figure out *"the best distribution"* there's a ***tradeoff***. The longer you take to figure it out the more money you'll probably spend on the wrong ones. Therefore you have to figure out very quickly.

* We call this ***regret*** and ***regret*** is mathematically defined.
* Paper: And if you can read more about this in this paper: ***Using Confidence Bounds for Exploitation-Exploration Trade-offs****, Peter Auer; Institute for Theoretical Computer Science, Graz University of Technology*.
* REGRET: Basically regret is a suffering when you're using an non optimal method.
* So whenever you are using the *non-optimal machine* you have a regret. Which can be quantified as: the difference between the best outcome and the non-best outcome. So the longer you explore other non-optimal machines the higher REGRET.
* But at the same time if you don't explore for long enough. Then a ***sub-optimal machine*** might appear ***as*** an ***optimal machine*** for you. In this case you didn't get the best Machine (real optimal machine).
* For instance: if we don't spend enough time exploring we might think that the green-colored distribution is the best machine because it's got quite a good (but not as good as orange one).

So our goal is *find* the *best* *distribution* but at the same time we have to *minimize* *the* *exploration*.

* Real-world application: One of the real-world application is "to find out the best advertising idea for a certain community". Consider the following image as the ideas for "Coca-Cola" company:



* So there is a distribution behind it but that distribution will only become known after thousands and thousands of people look at these ads and click or not click on these ads.
* AB test: One way to approach a problem is just run an A/B test. So take your five or 50 or 500 ads and run a huge A/B test (with multiple A/B test) and wait until you have a large enough sample and then conclude which is the best with certain confidence.
* But the problem of that is that you would spend a lot of time and money doing that. Because an A/B test is pure exploration. You're *not exploiting the best* option.
* So the challenge is to find out which is the Best One but do it while you're Exploring.
* So you have to do the A/B test and then use them to find out the best one in the *quickest* *way* *possible* and *start* *exploiting* it.

So that's the challenge here and that's what we're going to be solving. And that's the modern application of the Multi Armed Bandit Problem.

* A/B testing: **A/B testing**, also known as **split** **testing**, refers to a *randomized experimentation process* wherein two or more versions of a *variable* (web page, page element, etc.) are shown to different segments of *website visitors* at the same time to determine which version leaves the *maximum impact* and *drive business metrics*.

**More on Bandit**

1. Know the The Bandit Framework: A description of the code and test framework.
2. Bandit Algorithms:
3. The Greedy Algorithm
4. The Optimistic-Greedy Algorithm
5. The Epsilon-Greedy Algorithm (s-Greedy)
6. Regret

**6.1.3 How Reinforcement learn is used to train a Robot Dog**

We're going to be looking at different ways that we can solve the multi armed bandit problem and comparing the results. But remember that the ***Multi-Armed Bandit problem*** is not the only problem that can be solved with Reinforcement Learning. For instance Reinforcement Learning is used to train Robot Dogs to Walk.

* Assume that you have a *programmable* *robot* *dog* in which you can *implement* *an* *algorithm* inside the robot dog which will tell it how to walk.
* You can use two type of algorithms:

1. You can actually give the sequence of actions such as to accomplish a task (walking): Move ***front right foot*** and then move ***left back foot*** and then *Front* *Left* *Foot*, *Right* *Back* *Foot* and so on.
2. Or you can implement Reinforcement Learning algorithm which will train the dog to walk in a very interesting way.

* We give the dog a set of instructions (actions dog can take), for instance: Instructions for leg movements.
* Then we set a goal: goal is to make a ***step forward***.
* And we give it rewards/Punishments on its action: Every time dog make a ***step*** ***forward*** it is given a ***reward*** every time, if it ***fall*** ***over*** it will given a ***punishment***.
* Reward is basically a ***1***, punishment is ***0***. You just give it a ***1*** in algorithm and a punishment is ***0***. So basically you will try all these random sets of actions and see what they lead to every time it takes a step forward.
* And every time the dog takes a step forward it knows it's got a reward and it's good for it. It remembers that those were good actions and will try to repeat them more and more and it can learn to walk.
* So you ***don't have to*** program ***an actual*** walkingalgorithm into Dog will figure out the steps it needs to take on its own.

Now this topic is more of on the side of Artificial Intelligence rather than just Machine Learning. Now we focus on our ***Multi-Armed Bandit problem***.

**6.1.4 Upper Confidence Bound Algorithm in Reinforcement Learning**

In Reinforcement learning, the agent or decision-maker generates its training data by interacting with the world. The agent must learn the consequences of its actions through trial and error, rather than being explicitly told the correct action. Consider the problem:

* Multi arm bandit Problem:

1. We have arms. For example, arms are ***Ads*** : that we display to users each time they connect to a web page.
2. Each time a user connects to this web page, that makes a ***round***.
3. At each ***round n***, we choose ***one ad*** to display to the user.
4. At each round , gives reward defined as:
5. Our goal is to maximize the total reward we get over many rounds.

* UPPER CONFIDENCE BOUND ALGORITHM:

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

* = The number of the times the was *selected* up to round .
* = The number of *rewards* of the up to round .

1. Step 2: From these two numbers we compute:

* The average reward of up to round
* The confidence interval at round is:

Where:

1. Step 3: We select the that has the maximum UCB, .

* Walk through the example: Following are our ***slot*** ***machines*** or ***one*** ***arm*** ***bandits*** and each one of them has a ***distribution*** behind it.



* We want to find the *best one*. Looking at them we *can't tell* which one it is. But let's say we do know. Let's say we know the end result. Just for argument's sake the distributions look like. Obviously, the Orange one is the best distribution. But we don't know that. And we want to find that out in the process of playing these machines.

|  |
| --- |
|  |
| * We're going to take the Actual Expected Return from the distributions (the ***right side portions*** from the ***mode***), and we're going to put them onto a ***vertical*** ***axis***. |

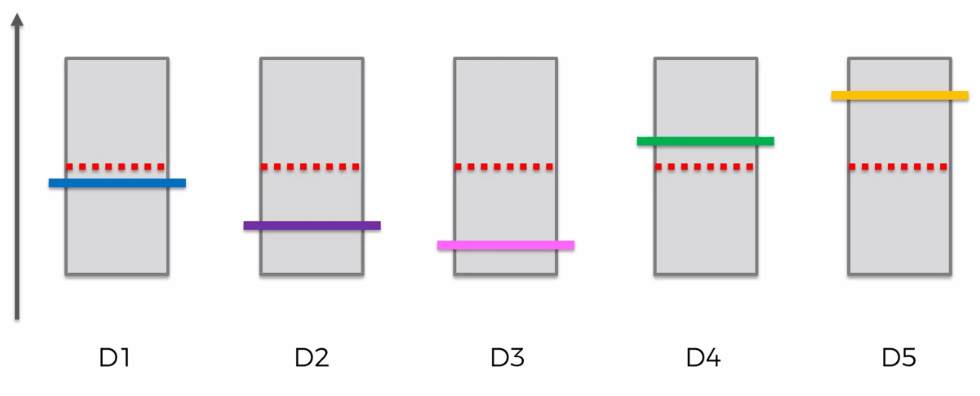
* So those are the ***expected*** ***values*** or ***returns*** for each of those *distribution* for each *machine* that's why are on **y** axis. But again we don't know that so what how does this Algorithm work?



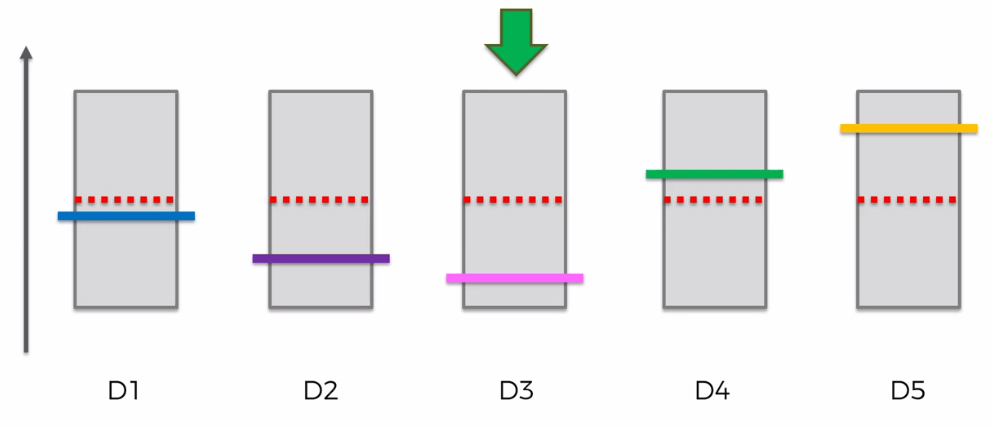
* Well, we assumes some *starting* point for *every* *distribution*. Let's just assume that all returns the same, i.e. at same level because we we can't discriminate against these machines at the very beginning. They all look the same.



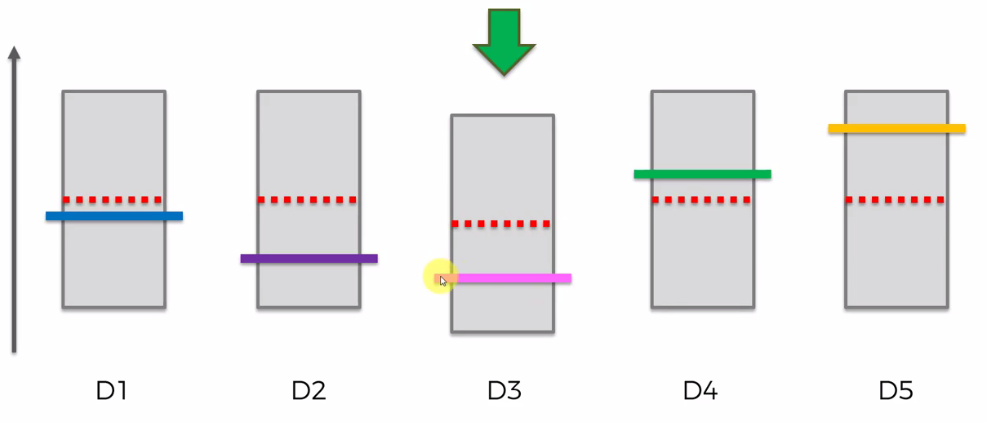
* The algorithm (formulas that are behind algorithm) create a ***confidence band***. And it is ***designed in such a way*** that we have a very high level of certainty that ***confidence band*** will include the *actual return* or the *actual expected return*.



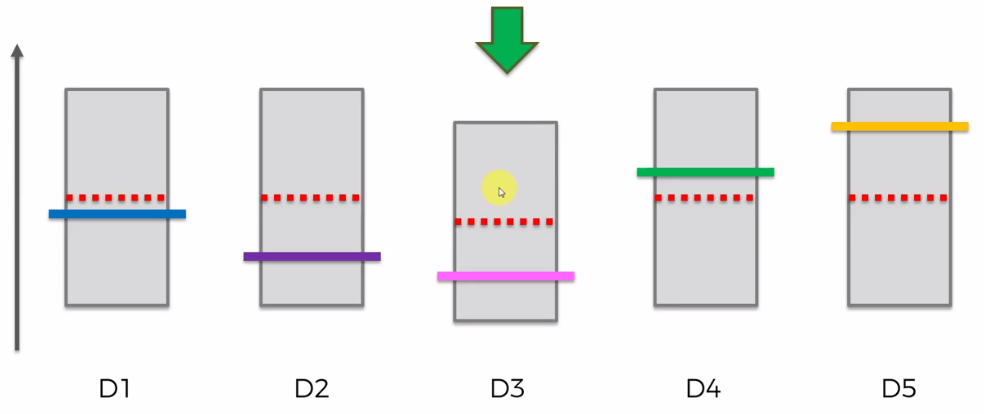
* So ***basically*** the ***first*** ***couple*** of ***rounds*** are going to be ***trial*** ***runs***. We're going to *intentionally* just *try* *out* *the* *machines* at least *one* *time* *each* in order for us to be able to place this *actual return value* in a confidence band.
* At first confidence band is going to be very large so that the *actual return* or the *actual expected return* falls inside this confidence level, with a very high degree of certainty.
* This *confidence band* is built around this *red-dotted empirical value* which are, at very start all the same.
* How does this algorithm work?: Out of all of them. We pick the machine with the ***highest*** ***confidence***. But right now it can be *any* of these *machines*. They all have the *same* *confidence*.



* So we're going to pick any one of them. Next we actually pull the lever of that selected machine. (Or, in case of Ads picking problem: we display that Ad and next we want to see *did the person click on it* or did the person *not click on it*.)
* Lets assume, in this case the person loss hit money to the machine (or user didn't click on that Ad). So this red-dotted value goes down because it is like the *observed average* the for a *large numbers of observation* this ***red-dotted value*** is always going to ***converge*** ***to*** the ***expected return*** or ***expected average*** or ***expected*** ***value*** for this ***distribution***.



Red-dotted value goes down



Confidence bond Shrinks

* And now because we have an *extra observation*, the second thing happens is the *confidence* bond *Shrinks*. That *confidence* *interval* become *smaller* because we have an *additional* *observation* we are more *confident* in our *predictions*.
* So the next step is now we find the next one with the highest *confidence* *bond*. Obviously it's one of these remaining machines, and we are picking a random one. Say we choose the Green one.

|  |  |
| --- | --- |
|  |  |

* Say we got positive result i.e. user wins with this machine (or, visitor clicked in our Ad), in this case the red-dotted value goes up. And the confidence band shrinks.
* Because we got an additional observation in our sample. Confidence bands only purpose is to include the actual expected value wherever it is. When the sample grows, we become more confident about the expected value, hence this confidence band shrinks.
* Now in this iteration we can *stop* *looking* for the *best* *distribution* and we can conclude that this *current* *machine* (or current Ad) is with the *best* *distribution* (which is not). In this case algorithm fails.
* Similarly we pick a machine/Ad randomly from the remaining ones. We do trial and we got positive/negative result the red-dotted line goes up/down according to the results and Confidence band Shrinks.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

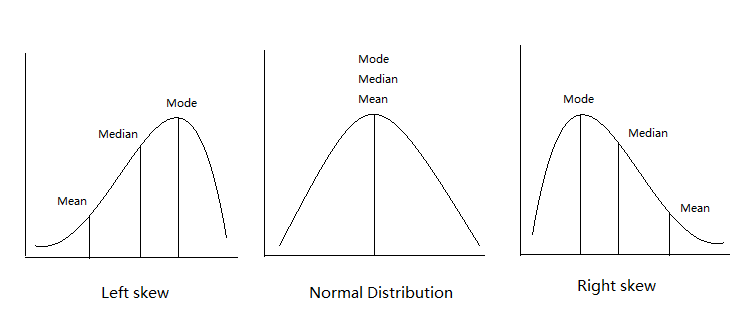
* For this example (we knew this is the best distribution) we exploit the Orange one, and its Confidence band gets so small. The main purpose is "if we found that the red-dotted line goes up for twice or more" the distribution is selected more. But also we pick some random machines.
* So even though we exploited the best option but for any option if it keeps going up, it's keeps being good. And at the same time we're building up the sample size hence ***decreasing*** the ***conference*** ***band***. But the point is: ***we select the distribution more with the positive outcome***.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

* And some times algorithm looks for other distributions also. But more trials go the algorithm choose the best distribution (converging) more often (because we found out there is a good possibility to be the best one).

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

* That’s how the ***algorithm*** ***converges*** to the ***best*** ***distribution*** using ***minimal*** ***trial***: It's Focus More And More on the Distributions that are giving more Positive Results and *ignores the unlucky distribution*, as sample is growing. To avoid trap random selection is used.
* So that is in essence the whole concept behind this UPPER CONFIDENCE BOUND ALGORITHM and that's how it solves the multi arm bandit problem.
* It's a very interesting solution & much more sophisticated and better than *just selecting randomly* or running an *A/B test and then selecting the option*. A very powerful algorithm.
* Mean, Median, and Mode: Every numerical data set has an ***average*** ***value*** that represents the ***weight*** of its ***array*** value. There are many different types of average! 3 of the most popular average values: ***Mean***, ***Median*** and ***Mode***. ***Mean***, ***Median*** and ***Mode*** are average values or ***central*** ***tendency*** of a numerical data set.



1. Mean: Mean can be calculated by adding all data points and dividing by the number of data points.
2. Median: Median is the ***middle value*** of a ***sorted data set***; found by ordering all data points and picking out the one in the middle (or if there are two middle numbers, taking the mean of those two numbers).
3. Mode: Mode is the most frequent number — that is, the number that occurs the highest number of times.

**6.1.5 Implementation of *UCB* in Python**

Reinforcement learning is taking us closer to the field of Artificial Intelligence because *robots* and *artificial* *intelligence* that comes with it are *partly* built with *reinforcement* *learning*.

* *Problem description:* Our goal is ***CTR*** ***optimization***. CTR means *Click Through Rates*. We are going to try to optimize the *click* *through* *rates* of different *users* on an Ad that we put on a *social* *network*.
* For some Car company marketing campaigns on the social network about their new SUV Car model. They want to put Ads on the social network.
* Now the Department of Marketing of that Car company prepared some different *versions* of the *same* *Ad*. They're actually not very sure which ad to put on the social network. They want to put the ad that will get the maximum clicks. So that most users buy the SUV.
* Now our goal is to find out the best version of the ***SUV car AD***. We consider this as the ***Bandit Problem***.

We want to find the ads that will get the most clicks.

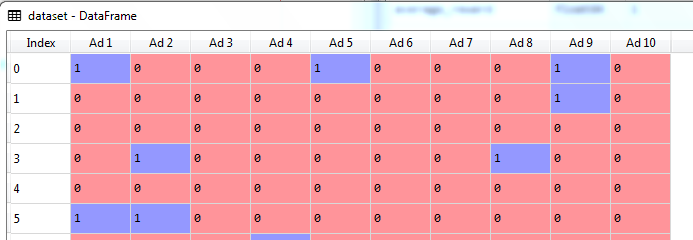
**6.1.6 No Data-set for Reinforcement Learning**

Here dataset is not important. In reality there is no dataset. In this example dataset is used to ***simulate*** the **"CTR process"**.

* Real life situation: So in real life, we are going to start experimenting by placing those Ads on a social network (the different versions of the ad).
* According to the results we observe we will change our strategy to place these Ads on the social network. That is our observation will determine ***"which particular version of Ad will be used next"***.
* So there is a key difference between what we're about to do now and what we've done in previous chapters (i.e. Regression, Classification, Association-Rule) because earlier we had a data-set with some data containing ***independent*** ***variables*** and one ***dependent*** ***variable*** which are used for clustering/classification/regression.
* So things are different now. We start with no data (the given data-set is actually for simulation of CTR).
* The data-set we are using here is ***just*** for ***simulation*** and we're going to pretend we're in real life, we select an Ad and then we generate ***Click/No-click event*** using the given Data-set. We're going to pretend that we don't have any data yet.
* Description of the Problem: We have *ten* *versions* of the *same* *Ads*. 10 versions of these Ads trying to sell this cheap luxury SUV. And each time a user of the social network will *log* *into* his *account* we will place *one* *version* of these 10 Ads.
* Then we will ***observe its response*** and give Rewards:

1. If the user clicks on the Ad. We get a Reward equal to ***1***.
2. And if the user doesn't click on the ad we get a Reward equals to ***0***.
3. We are going to do this for *ten* *thousand* *users* on the social network. Then we observe if the user clicks: *yes/no* on the *Ad* .
4. The user click will be simulated using the given dataset.

* What's in the Data-set: This data-set actually mimic the real-time situation, which we are choose an Ad and this dataset generates User-Click (returns 1) or No-Click (returns 0).



* This is just some simulation of what is going to happen when we show the ads to the users: It's telling is for each round an user connecting to his account on which versions of the user is going to click on (or not).
* For example consider the five first users. Let's take first user indexed by 0.
* At the first round, according to this simulation, this first user of the social network is going to click on the Ad if we show him the **1st**, **version** or the **5th version** or the **9th version**. And he/she not going to click the Ad if we choose ***2, 3, 4*** or ***6, 7, 8*** and ***10th version*** of the Ad.
* At the 2nd round, for the 2nd user, he/she not going to click the Ad except ***9th Version*** of the Ad.
* Reinforcement learning in Action: The key thing to understand about *reinforcement* *learning* is that this *strategy will depend at* eachround *on the* previousresults *we* observed *at the* previousrounds.
* So for example when we are at ***round 10***, the algorithm will look at the different results observed during the first ten rounds and according to these results UCB algorithm will decide which ***version*** of the ***Ad*** it will show to the user. That's why reinforcement learning is also called Online Learning or Interactive Learning.
* So how can we choose the different versions of the ads? There's going to be a specific strategy to do this. We compare two version of Algorithm. One is ***no-UCB*** i.e. we select Ads ***randomly***. The other version is: we use ***UCB*** to choose Ads ***according to observations***.

1. At our ***first version*** no Algorithm or no Strategy is used, we run ***10,000 trial***(round) one by one and we are ***not going to use UCB*** to choose the *version* of ***Ad*** instead we pick the *Versions* ***randomly*** for each round.

* Each time a user connects to its account we're displaying one version of these ten at totally random. And we count the rewards.

1. In our ***second version*** we also run the ***10,000 trials*** and in this case we will use UCB to pick Ad-versions this algorithm will decide from here which version of the ad to show to the user.

* And depending on the reward (***0*** or ***1***) of the current observation, the UCB-Algorithm will decide which Ad to show to the user at the next round.
* The goal of the algorithm is to ***maximize*** the ***total*** ***reward***.

**6.1.7 Random Selection Algorithm**

The Random Selection Algorithm, just consists of selecting one ***random version*** of the Ad each time a user connects on his/her social network accounts.

* The most important variable is the " ***total\_reward***" the sum of the different rewards up to the 10000th user.
* If we run this following code, the total reward ***"total\_reward"*** will be around 1200. That is around 1200 user clicks if we choose the Ads randomly.

#*Random Selection*

#*Importing the libraries*

**import** numpy **as** np

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

**import** random

#*Importing the dataset*

dataset = pd**.read\_csv**('Ads\_CTR\_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)

    #*in this case "reward" indicates the total clicks if we select the ads "randomly".*

    #*in UCB we do not select ads randomly.*

    reward = dataset**.**values[n, ad]

    total\_reward = total\_reward + reward

#*Visualising the results*

plt**.hist**(ads\_selected, rwidth=0.85)

plt**.title**('Histogram of ads selections')

plt**.xlabel**('Ads')

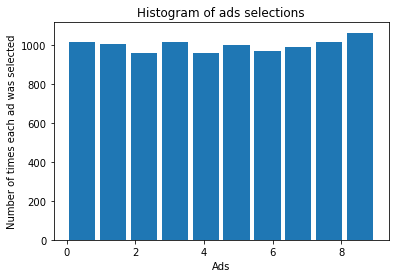
plt**.ylabel**('Number of times each ad was selected')

plt**.show**()

"""

The parameter rwidth specifies the width of your bar relative to the width of your bin. For example, if your bin width is say 1 and rwidth=0.5, the bar width will be 0.5. On both side of the bar you will have a space of 0.25.

"""



* ***Note:*** Bar width of histogram: The parameter ***rwidth*** specifies the ***width*** of your ***bar*** relative to the ***width*** of your ***bin***. For example, if your ***bin*** width is say ***1*** and ***rwidth=0.5***, the ***bar*** width will be ***0.5***. On both side of the ***bar*** you will have a space of ***0.25***.
* In the for loop, we are selecting **Ad** randomly between **1** to **10**, using ***ad = random.randrange(d)*** where ***d = 10***, now we store this random number in the array ***ads\_selected = []*** using ***ads\_selected.append(ad)***,that is at the end the length of this array will be 10,000.



* And this array ***ads\_selected*** will be used to create "histogram". This histogram counts the randomly selected and displays through the histogram.



* ***reward = dataset.values[n, ad]***
* Retrieves the reward ***0*** or ***1*** from the given dataset (simulation is happening in this line), at the ***n*** round/trial. Then "***total\_reward = total\_reward + reward"*** counts the reward.
* For example, if the **1st** **version** of the ***Ad*** is selected for the first round, user will click it (according to simulation from dataset). Now in 2nd round/trial the 5th **version** of the ***Ad*** is selected then user will not click it.
* Note: Notice the ***index*** of the ***add***, and ***random.randrange(d)*** generates integers **0, 1, 2, … , 9**.
* In the histogram we see that all the ***bars*** have ***nearly*** ***same*** ***height***, because ***Ads*** are ***selected*** ***randomly***.
* So let's keep this in our mind, that ***total\_reward*** is always near ***1200*** (in case of random selection) because then we'll compare it to the total reward that we get from Upper Confidence Bound and then the Thompson Sampling algorithm.

|  |  |
| --- | --- |
| * Visualizing the result: In this part of reinforcement learning the visualization of the results will consist of visualizing the histogram where we see the different selections of the different versions of the Ad. * Since our algorithm ***randomly*** ***selected*** the ***different*** ***versions*** of the ***Ads*** at each round. Hence we get a nearly uniform distribution of the different versions of the ***Ads***, where we selected ***All 10 Versions*** more or less the same number of times. |  |

**6.1.8 Coding the UCB algorithm in python from scratch**

Currently we have no package on UCB. So we will code the following steps in Python:

* UPPER CONFIDENCE BOUND ALGORITHM:

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

* = The number of the times the was *selected* up to round .
* = The number of *rewards* of the up to round .

1. Step 2: From these two numbers we compute:

* The average reward of up to round
* The confidence interval at round is:

Where:

1. Step 3: We select the that has the maximum UCB, .

* Step 1:

Lets set,

***numbers\_of\_selections*** = The number of the times the ***ad\_***i was selected up to ***round n***.

***numbers\_of\_rewards*** = The number of rewards of the ***ad\_i*** up to ***round n***.

* We consider it as a ***d size vecto***r. Here ***d*** indicates the ***number of bandits*** (in this case number of the ***Ads***, i.e. ***d = 10***). Hence we first create numbers\_of\_selections as a **list of 0's** of size ***d*** with all elements equal to **0**:

numbers\_of\_selections = [0]\*d

* Similarly we create: numbers\_of\_rewards = [0]\*d
* So that we can access and update the *"Number of Selection"* and *"Number of Rewards"* of the ***d-th Ad,*** by accessing the corresponding elements of these two vectors/lists

d = 10

numbers\_of\_selections = [0]\*d

numbers\_of\_rewards = [0]\*d

Note:

* Creating an Empty List of Length 10:

l = [None] \* 10

l = [None, None, None, None, None, None, None, None, None, None]

* Assigning a value to an existing element of the above list:

l[1] = 5

l = [None, 5, None, None, None, None, None, None, None, None]

Similarly we've created **10 size** list of **0**'s as:

name\_list[0]\*10

* Step 2: Notice we have two indices, n-th-round and i-th Ad. ***n*** increments the ***10,000 trials*** and ***i*** increments ***10 Ads***. Hence we need two ***for*** loops.
* We do the following things:

1. The average reward of ad\_i up to round n
2. and
3. Upper bound of the confidence interval i.e. Upper-Confidence-Bound for each Ad at round n:

**for** n **in** **range**(0, N):

**for** i **in** **range**(0, d):

        avg\_reward = numbers\_of\_rewards[i]/numbers\_of\_selections[i]

        #*log(n+1) because of index*

        delta = math**.sqrt**((3\*math**.log**(n+1))/(2\*numbers\_of\_selections[i]))

        upper\_conf\_bound =  avg\_reward + delta

#*UCB Implemnetation*

**import** math

#*here d is No. of Ads or Number of Bandits*

d = 10

N = 10000   #*Total number of trials*

numbers\_of\_selections = [0]\*d

numbers\_of\_rewards = [0]\*d

**for** n **in** **range**(0, N):

**for** i **in** **range**(0, d):

        avg\_reward = numbers\_of\_rewards[i]/numbers\_of\_selections[i]

        #*log(n+1) because of index*

        delta = math**.sqrt**((3\*math**.log**(n+1))/(2\*numbers\_of\_selections[i]))

        upper\_conf\_bound =  avg\_reward + delta

* Step 3: We select the i-th Ad that has the maximum UCB, ***upper\_conf\_bound***
* For this we have to create a ***vector/list*** which contains selected Ads for all ***10,000 trials***. So first we'll create an empty list:

***ads\_selected = []***

***ads\_selected*** will be a vector of 10000 elements and each of these elements will be the *Ad* that was *selected* at *each round*. We are going to append this vector by the ***UCB selected Ads***.

* Now the question is how are we going to append the different versions of the Ad in this ***ads\_selected*** vector?
* So we created ***max\_upper\_bound*** and we initialize it for each ***trial/round*** (i.e. it is inside 1st ***for*** loop).
* But also we need to ***keep track*** of the ***index*** of the Ad that has the ***max\_upper\_bound***. So that we can identify an Ad is selected. For this reason we used: ***ad\_max\_ucb = i***.

**for** n **in** **range**(0, N):

*ad\_max\_ucb = 0*

*max\_upper\_bound = 0*

**for** i **in** **range**(0, d):

        avg\_reward = numbers\_of\_rewards[i]/numbers\_of\_selections[i]

        #*log(n+1) because of index*

        delta = math**.sqrt**((3\*math**.log**(n+1))/(2\*numbers\_of\_selections[i]))

        upper\_conf\_bound =  avg\_reward + delta

***if****upper\_conf\_bound****>****max\_upper\_bound:*

*max\_upper\_bound = upper\_conf\_bound*

*ad\_max\_ucb = i*

* 10 Ads at the beginning (d Bandits at the beginning): At the beginning you know during the 10 first rounds (generally first ***d*** rounds) we don't have much information of the Ads. We don't have much information about their ***reward***.
* Basically for first 10 Ads we select them serially (not randomly), i.e. at ***round 1 Ad1*** is selected, ***round 2 Ad2*** and so on up to 10th round, afte 10th round/trial we are going to use UCB.
* To implement this, we use an ***if-else*** condition inside the second for-loop (i). And under ***else*** condition we put a very large ***upper\_conf\_bound***.

#*UCB Implemnetation*

**import** math

#*here d is No. of Ads or Number of Bandits*

d = 10

N = 10000   #*Total number of trials*

numbers\_of\_selections = [0]\*d

numbers\_of\_rewards = [0]\*d

slected\_ads = []

**for** n **in** **range**(0, N):

    ad\_max\_ucb = 0

    max\_upper\_bound = 0

**for** i **in** **range**(0, d):

***if****(numbers\_of\_selections[i]****>****0):*

            avg\_reward = numbers\_of\_rewards[i]/numbers\_of\_selections[i]

            #*log(n+1) because of index*

            delta = math**.sqrt**((3\*math**.log**(n+1))/(2\*numbers\_of\_selections[i]))

            upper\_conf\_bound =  avg\_reward + delta

**else**:

*upper\_conf\_bound = 1e400 # i.e. 10^400*

**if** upper\_conf\_bound **>** max\_upper\_bound:

            max\_upper\_bound = upper\_conf\_bound

            ad\_max\_ucb = i

* Lets walk through the iterations:
* At first ***for*** loop ***n=0***, ***ad\_max\_ucb*** and ***max\_upper\_bound*** both set to ***0***.
* Inner ***for*** ***loop*** begins with ***Ad\_0***, for ***Ad\_0,***

numbers\_of\_selections[0] > 0

is ***false*** so ***upper\_conf\_bound*** is set to ***1e400***.

In following ***if*** condition, ***upper\_conf\_bound > max\_upper\_bound*** become ***true***. So ***max\_upper\_bound*** (which was **0**) is set to ***1e400***. And ***Ad\_0***is selected by this statement: ***ad\_max\_ucb = 0***.

* Then in 2nd iteration in inner ***for*** loop (***i = 1***).

numbers\_of\_selections[1] > 0 is ***false*** so ***upper\_conf\_bound*** is set to ***1e400*** again.

But in following ***if*** condition, ***upper\_conf\_bound > max\_upper\_bound*** become ***False (1e400>1e400)***. Hence ***Ad\_1*** is *not going to selected* by this statement: ***ad\_max\_ucb = i***.

* And this is happening for the remaining iterations in inner ***for*** loop (***i = 2, 3, . . . , 9***), ***Ad\_0*** stays as the selected Ad. The inner ***for*** loop (***i***) ends for the first round ***n=0***.
* Now at the first ***for*** loop in 2nd iteration ***n=1***, ***ad\_max\_ucb*** and ***max\_upper\_bound*** both set to ***0*** again.
* Inner ***for*** ***loop*** begins with ***Ad\_0***, for ***Ad\_0,***

numbers\_of\_selections[0] > 0

is ***True*** so ***upper\_conf\_bound*** is set by ***avg\_reward + delta*** which is smaller < ***1e400***.

* In following ***if*** condition, ***upper\_conf\_bound > max\_upper\_bound*** become ***true***, because ***max\_upper\_bound*** both set to ***0*** again.
* So ***max\_upper\_bound*** (which was **0**) is set to ***avg\_reward + delta*** which is smaller < ***1e400***.
* And ***Ad\_0*** is ***selected (temporarily)*** by this statement: ***ad\_max\_ucb = 0***.
* Then in 2nd iteration in inner ***for*** loop (***i = 1***). For ***Ad\_1***

numbers\_of\_selections[1] > 0 is ***false*** so ***upper\_conf\_bound*** is set to ***1e400*** again. (Which was set to ***avg\_reward + delta*** < ***1e400*** in previous iteration).

* Hence in following ***if*** condition, ***upper\_conf\_bound > max\_upper\_bound*** become ***True***

***(1e400> avg\_reward + delta)***.

* So ***max\_upper\_bound*** (which was ***avg\_reward + delta***) is set to ***1e400*** again.
* Hence ***Ad\_1*** is *going to selected* by this statement: ***ad\_max\_ucb = 1***.
* Then in 3rd iteration in inner ***for*** loop (***i = 2***). For ***Ad\_2***

numbers\_of\_selections[2] > 0 is ***false*** so ***upper\_conf\_bound*** is set to ***1e400*** again.

But in following ***if*** condition, ***upper\_conf\_bound > max\_upper\_bound*** become ***False (1e400>1e400)***. Hence, 3rd Ad, ***Ad\_2*** is *not going to selected* by this statement: ***ad\_max\_ucb = i***.

* And this is happening for the remaining iterations in inner ***for*** loop (***i = 3, 4, . . . , 9***), So the 2nd Ad, ***Ad\_1*** stays as the selected Ad after the inner ***for*** loop (***i***) ends for the second round ***n=1.***
* This is how ***all 10 (d) Ads*** gets ***selected*** for the first ***10 rounds/trials*** for ***n= 0, 1, 2, . . . , 9***.
* And from ***n = 11*** to ***9,999*** the following statement never going to executed and ***numbers\_of\_selections[i] > 0*** remains ***true*** for all following iterations (i.e. Ads will be selected by the maximum UCB):

else:

upper\_conf\_bound = 1e400 # i.e. 10^400

* And UCB is always calculated using: ***upper\_conf\_bound = avg\_reward + delta***
* And the Ads are selected by the following:

**if** upper\_conf\_bound **>** max\_upper\_bound:

            max\_upper\_bound = upper\_conf\_bound

            ad\_max\_ucb = i

* Storing Selected Ads to ***slected\_ads***, tracking ***numbers\_of\_selections***, ***numbers\_of\_rewards*** and calculating the ***total reward***:

#*Reinforcement lrarning : ----------  UCB. ---------  Select an add by Click on Ad*

**import** pandas **as** pd

**import** numba **as** np

**import** matplotlib**.**pyplot **as** plt

#*importing dataset*

dataSet = pd**.read\_csv**("Ads\_CTR\_Optimisation.csv")

#*UCB Implemnetation.*

    #*3 parameters are important:*

        #*no. of bandit d,*

        #*No. of trials N,*

        #*initial max\_upper\_bound (1e400)*

**import** math

#*here d is No. of Ads or Number of Bandits*

d = 10

N = 10000   #*Total number of trials*

numbers\_of\_selections = [0]\*d

numbers\_of\_rewards = [0]\*d

slected\_ads = []

total\_reward = 0

**for** n **in** **range**(0, N):

    ad\_max\_ucb = 0

    max\_upper\_bound = 0

**for** i **in** **range**(0, d):

**if**(numbers\_of\_selections[i] **>** 0):

            avg\_reward = numbers\_of\_rewards[i]/numbers\_of\_selections[i]

            #*log(n+1) because of index*

            delta = math**.sqrt**((3\*math**.log**(n+1))/(2\*numbers\_of\_selections[i]))

            upper\_conf\_bound =  avg\_reward + delta

**else**:

            upper\_conf\_bound = 1e400 #*i.e. 10^400*

**if** upper\_conf\_bound **>** max\_upper\_bound:

            max\_upper\_bound = upper\_conf\_bound

            ad\_max\_ucb = i

    #*storing selected Ad*

    slected\_ads**.append**(ad\_max\_ucb)

    #*updating "numbers\_of\_selections" and "numbers\_of\_rewards" of selected Ad "ad\_max\_ucb"*

    numbers\_of\_selections[ad\_max\_ucb] += 1

    reward = dataSet**.**values[n, ad\_max\_ucb] #*generating reward-simulation from given Dataset*

    numbers\_of\_rewards[ad\_max\_ucb] += reward

    total\_reward += reward

#*python prctc\_UCB.py*

* Moment of Truth:
* After running the above code we get total reward ***2178***, which is more than 1200 (in random Ad selection). So UCB improved the Ad selection.
* From the following Data we see that the ***5th Ad*** (i.e. index 4), is the ***best*** version of the ***Ad***.
* The Ad selection process of UCB is visible from the 31st round/iteration. And for most iterations Ad of index 4 (i.e. 5th Ad) is being selected. At the down 5th Ad selected mostly.
* And we can only observe ***4***, but be careful this is the ***index*** this the actually the ***5th Ad***.

|  |  |  |
| --- | --- | --- |
| ***numbers\_of\_rewards*** | ***numbers\_of\_selections*** | ***UCB selected Ads*** |
|  |  |  |

* Visualizing the result: As we did before for "Random Ad selection". We use the list ***slected\_ads*** to draw the histogram.

|  |  |
| --- | --- |
| #*visualizing the result*  plt**.hist**(slected\_ads, rwidth=0.85)  plt**.title**('Histogram of ads selections')  plt**.xlabel**('Ads')  plt**.ylabel**('Number of times each ad was selected')  plt**.show**() | Shows 5th Ad with maximum exploitation |

Practiced version

#*Reinforcement lrarning : ----------  UCB. ---------  Select an add by Click on Ad*

**import** pandas **as** pd

**import** numba **as** np

**import** matplotlib**.**pyplot **as** plt

#*importing dataset*

dataSet = pd**.read\_csv**("Ads\_CTR\_Optimisation.csv")

#*UCB Implemnetation.*

    #*3 parameters are important:*

        #*no. of bandit d,*

        #*No. of trials N,*

        #*initial max\_upper\_bound (1e400)*

**import** math

#*here d is No. of Ads or Number of Bandits*

d = 10

N = 10000   #*Total number of trials*

numbers\_of\_selections = [0]\*d

numbers\_of\_rewards = [0]\*d

slected\_ads = []

total\_reward = 0

**for** n **in** **range**(0, N):

    ad\_max\_ucb = 0

    max\_upper\_bound = 0

**for** i **in** **range**(0, d):

**if**(numbers\_of\_selections[i] **>** 0):

            avg\_reward = numbers\_of\_rewards[i]/numbers\_of\_selections[i]

            #*log(n+1) because of index*

            delta = math**.sqrt**((3\*math**.log**(n+1))/(2\*numbers\_of\_selections[i]))

            upper\_conf\_bound =  avg\_reward + delta

**else**:

            upper\_conf\_bound = 1e400 #*i.e. 10^400*

**if** upper\_conf\_bound **>** max\_upper\_bound:

            max\_upper\_bound = upper\_conf\_bound

            ad\_max\_ucb = i

    #*storing selected Ad*

    slected\_ads**.append**(ad\_max\_ucb)

    #*updating "numbers\_of\_selections" and "numbers\_of\_rewards" of selected Ad "ad\_max\_ucb"*

    numbers\_of\_selections[ad\_max\_ucb] += 1

    reward = dataSet**.**values[n, ad\_max\_ucb] #*generating reward-simulation from given Dataset*

    numbers\_of\_rewards[ad\_max\_ucb] += reward

    total\_reward += reward

#*visualizing the result*

plt**.hist**(slected\_ads, rwidth=0.85)

plt**.title**('Histogram of ads selections')

plt**.xlabel**('Ads')

plt**.ylabel**('Number of times each ad was selected')

plt**.show**()

#*python prctc\_UCB.py*

Instructor version

#*Upper Confidence Bound (UCB)*

#*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

#*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**()