Chapter 6 : Part 2

**Reinforcement Learning**

**Thompson Sampling**

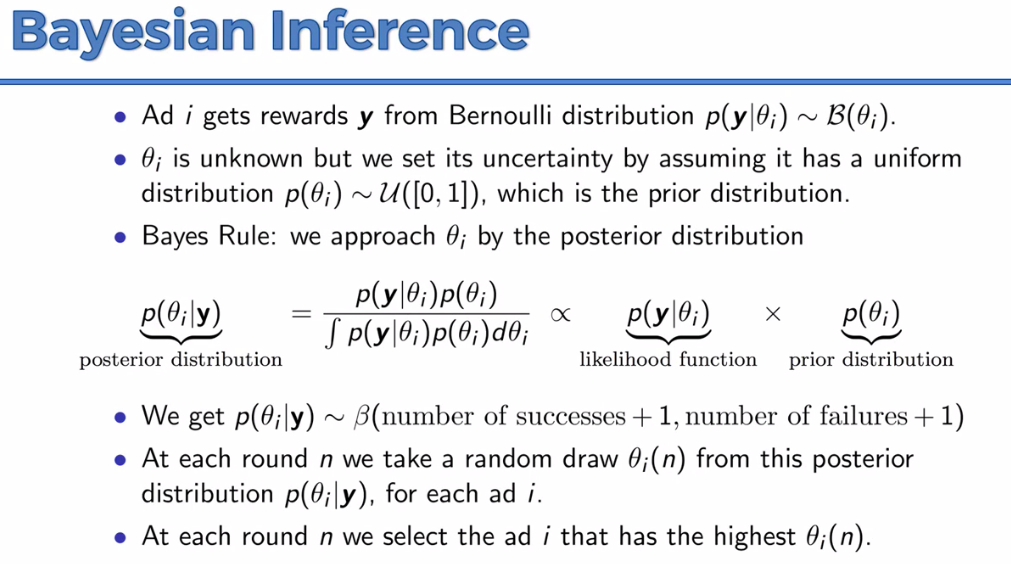
**6.2.1 Thompson Sampling**

We're going to be using this algorithm to solve the Multi Armed Bandit Problem.

|  |  |
| --- | --- |
| * Multi Armed Bandit Problem: We have several slot machines each one of them has a distribution behind it. We don't know what these distributions are and we need to start playing these machines and at the same time figure out which one has the best distribution. * So we have to find that ***ideal*** ***balance*** or ***tradeoff*** between ***exploration*** and ***exploitation***. |  |

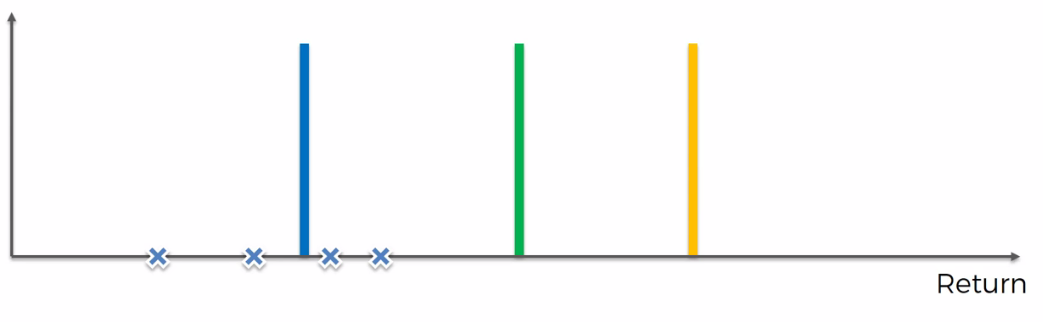
Multi arm bandit Problem Assumptions

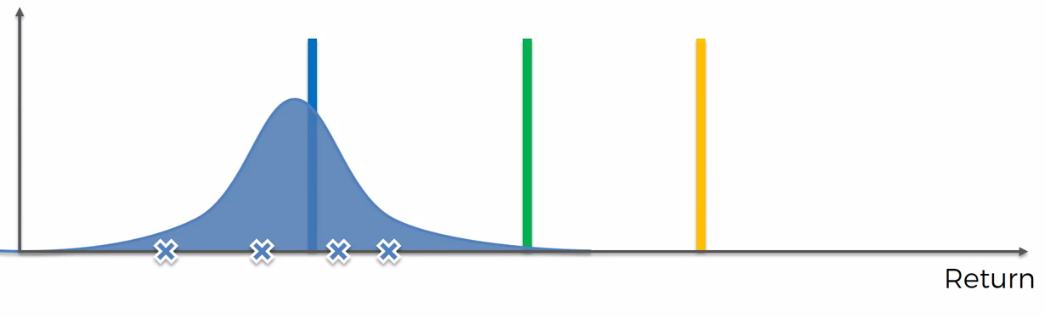
1. We have arms. For example, arms are ***Ads*** : that we display to users each time they connect to a web page.
2. Each time a user connects to this web page, that makes a ***round***.
3. At each ***round n***, we choose ***one ad*** to display to the user.
4. At each round , gives reward defined as: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



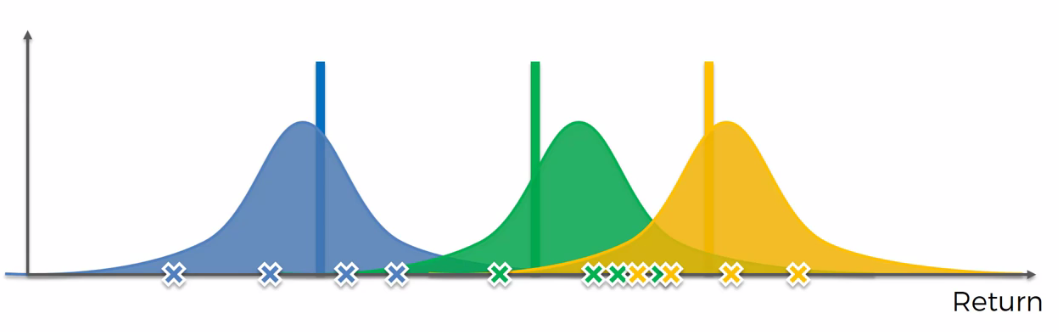


* So here we've got a scale. For simplicity we consider 3 bandits. ***Vertical*** line represents the ***expected*** ***value*** and ***horizontal*** line represents ***return***.
* Those colored-vertical lines are central distribution i.e. the real expected values of the machine. The algorithm of course doesn't know about those.
* At first all the machines are considered as similar. You have to have at least one or even a couple of trial rounds just to get some data to analyze.
* Say we run some trail in the blue machine, then the algorithm tries to make a distribution

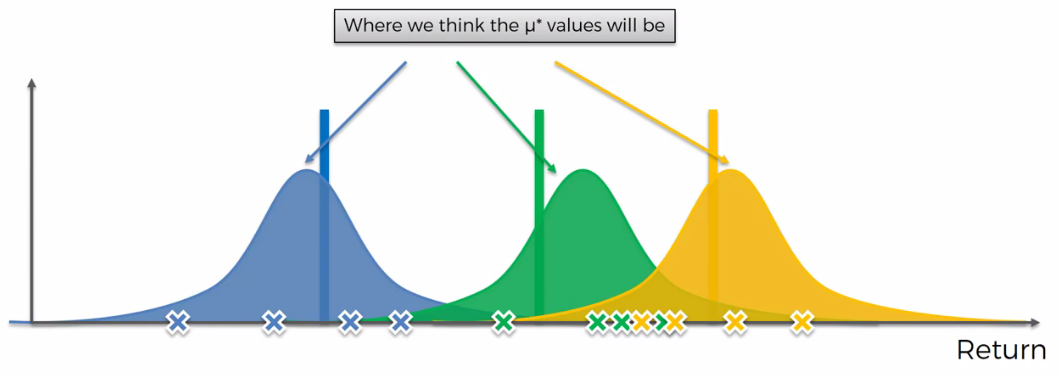




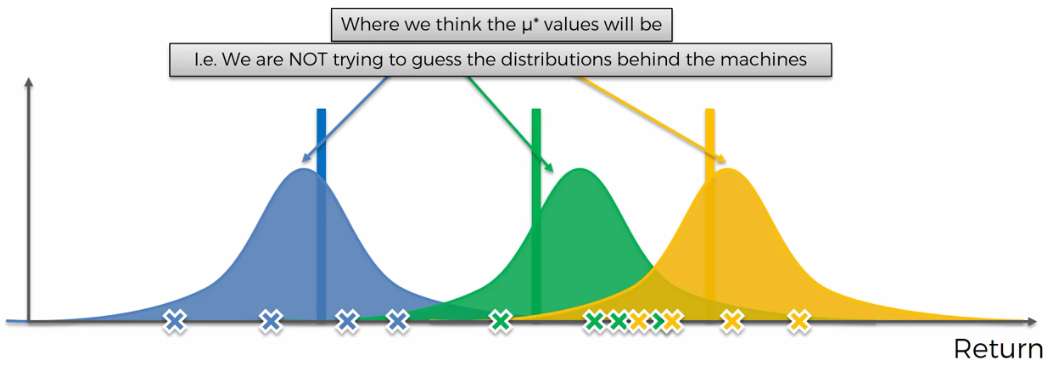
* So we do the same things for other two, we pull the machines arm several times and draw a distribution for each machine.



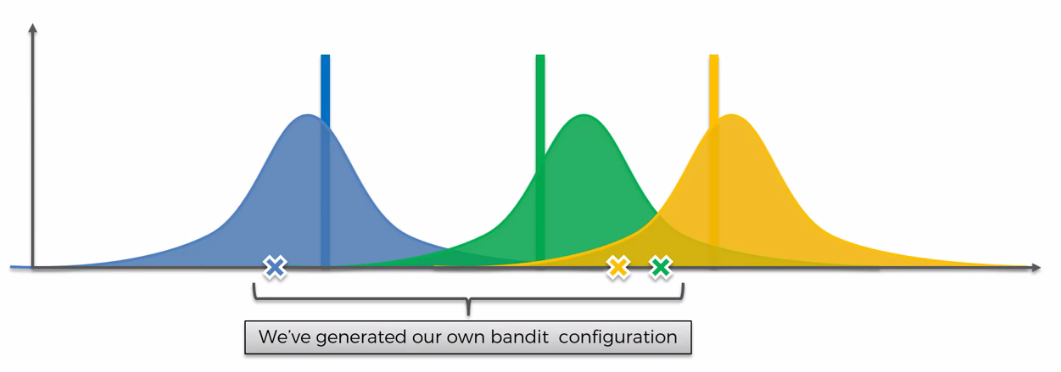
* The actual meaning of this these distributions is not what you might think at first, these distributions are not trying to guess the distributions behind the machines.
* The first thing that might come to mind is that (for instance): the *blue* *distribution* is *attempting* to *guess* the *actual* *distribution* behind the *blue* *machine* right. Or, same for the green or Orange !!!
* But in reality they are not doing that: we are actually constructing distributions of where we think the actual expected value might lie.



* We're creating an *auxiliary* *mechanism* for us to *solve* the *problem*. So we're not we're not trying to recreate these machines we're recreating the possible way these machines could have been created.
* For example consider the four Blue-dots, they are the possible position for the actual Blue-colored-vertical line or central distribution. And based on those *four* values we've constructed this *blue-distribution* which will showing us *possible* *positions* for that value . It shows the *high likelihood* or *low likelihood* for the *position* of the actual *Blue-Vertical* line. And same goes for the ***Green*** and ***Orange*** ***distribution***.



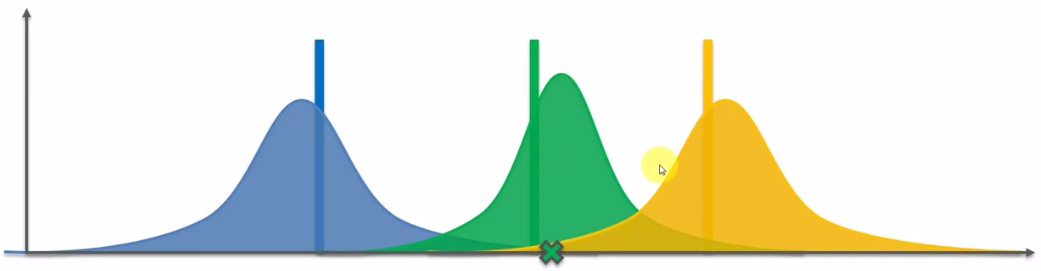
* Hence the Thompson Sampling is Kind of a Probabilistic algorithm. Where Upper-bound is a Deterministic Algorithm.
* So in simple word, we've created our own *Bandit-Configuration*, where we know the all distribution. It is some kind of imaginary distribution.



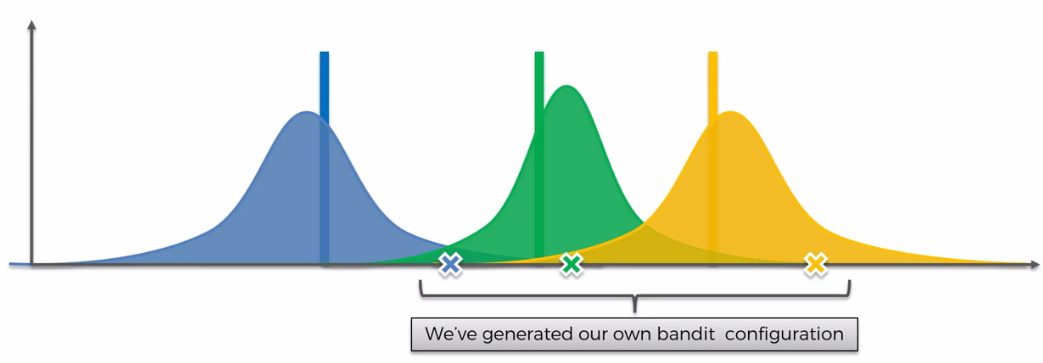
* Now from above three points we choose the green one (because it lies in the *right side* of the *imaginary central tendency*, hence it may give positive result). According to our own *Bandit-Configuration*, we generate a value and we comparethat *value* with the *value generated in the real world*. Then our sample grows and we make correction to our own version of distribution.

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

* So our green version shifted a little bit and got narrow (because sample is increased) and increased little higher. These happens because in our real world value lies in the *left side* of the *imaginary central tendency,* i.e. we get the wrong expected value in our imaginary distribution (however we were lucky to got positive value in real world). Hence we shifted the imaginary distribution to the left. Since the sample grows bigger the distribution narrowed and grows higher.

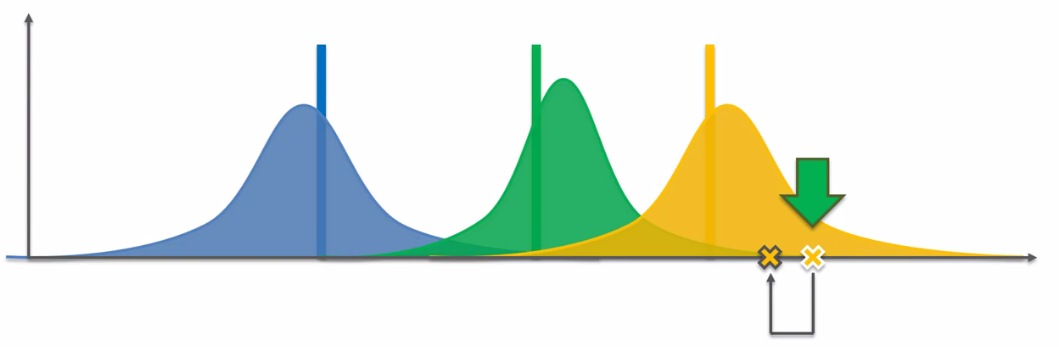


* That's the point that every time we add New INFORMATION our distribution becomes more and more REFINED.
* New Round: Similarly we pick three values (randomly ?) for three Blue, Green and Orange as the ***expected Return***, for our new imaginary Version of Bandit Problem. Out of these three we pick the best bandit. Which is in this new round, in our imaginary is *Bandit-Configuration*, the Orange one:

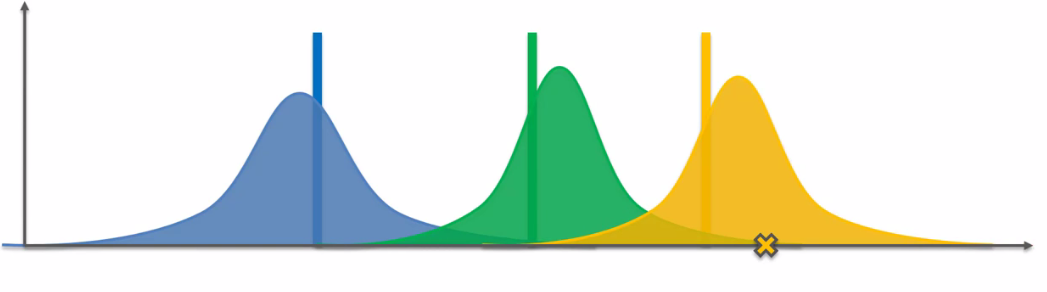
****



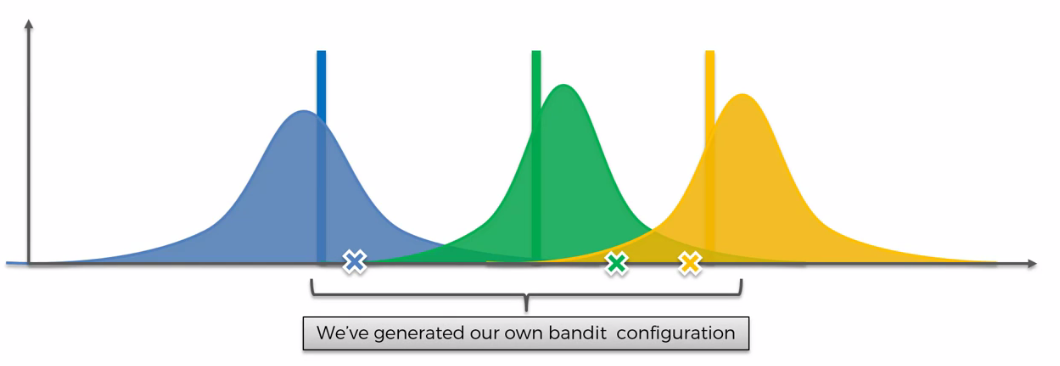
* Now we generate the real value by pulling the hand of the Orange machine, we get the following result:

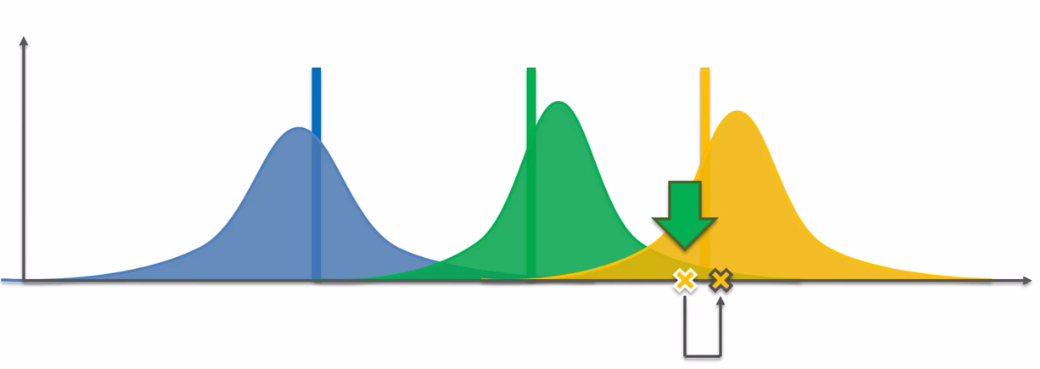


* Then we refine our Orange distribution.

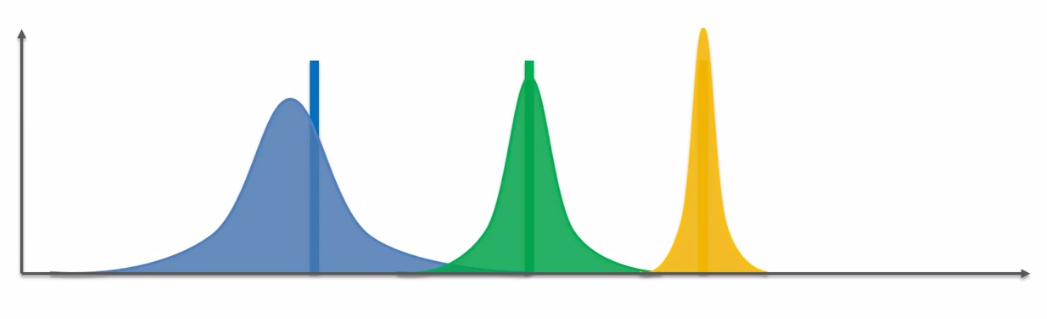


* Another Round: Now for another round we get the following:





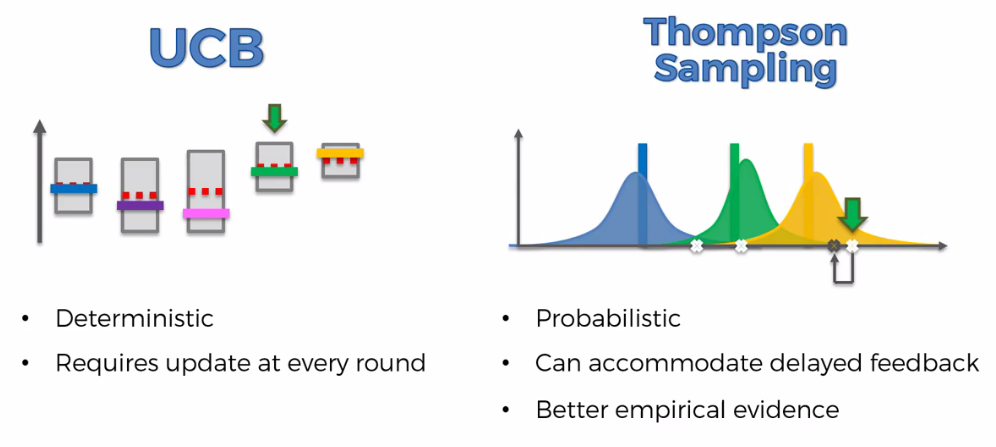
* After certain amount of Round we end up like follows. Where we used Orange machine, more and more because of the correct prediction. That’s how this Thompson sampling algorithm converges to the correct best distribution. For this reason the ***Green*** and ***Blue*** distribution doesn't get refined as ***Orange*** one.



* Which is totally fine because our point is to *Get* to the *Best Machine* to find it and *Exploit* it *As Much As We Can*.
* Every time we're generating these values, and they are kind of creating this *hypothetical set up of the bandits* and then we're solving that and then we're applying the results to the real world.
* We're adjusting our perception of reality based on the new information that generates and then we're repeating the whole process.

**6.2.2 UCB vs Thompson Sampling**

We're going to compare the two because they do solve the same problem ***"The Multi Armed Bandit"*** and let's have a look at some of the *pros* and *cons* of each of the *algorithms*.



1. UCB is a deterministic algorithm, there's lots of different modifications to these algorithm, all belongs to a family of UCB. They are all deterministic and basically what that means is that it is very straightforward.

* So once you have *certain* *round* you just look at the *upper* *confidence* and you going to Pick highest.
* You pull the lever then you do get a *random* *value* *from* the *machine*. But that's on the *side* of the *machine*, when we get the value it is very determined. There's no randomness in the algorithm itself.

1. On the other hand the THOMPSON SAMPLING ALGORITHM is a probabilistic algorithm because in that there are distributions which represent our perception of the world and where we think the actual expected returns of each of those machines might lie.

* And therefore every time we are *implementing* or *iterating* in the *Thompson Sampling Algorithm* we actually *generate* *random* values from those *distributions*.
* So it's always going to be *different* because you're always *sampling* from your *distributions* which *characterize* your *perception* of the *world* and that is a whole different type of algorithm. It's a probabilistic algorithm.

1. UCB requires an update at every round (for each round). So once you've pulled the lever and you get a *value back* from that *machine* that value you have to *incorporate* it right away in order to *proceed to the next* round. You cannot proceed to the next round until you have incorporated that value.

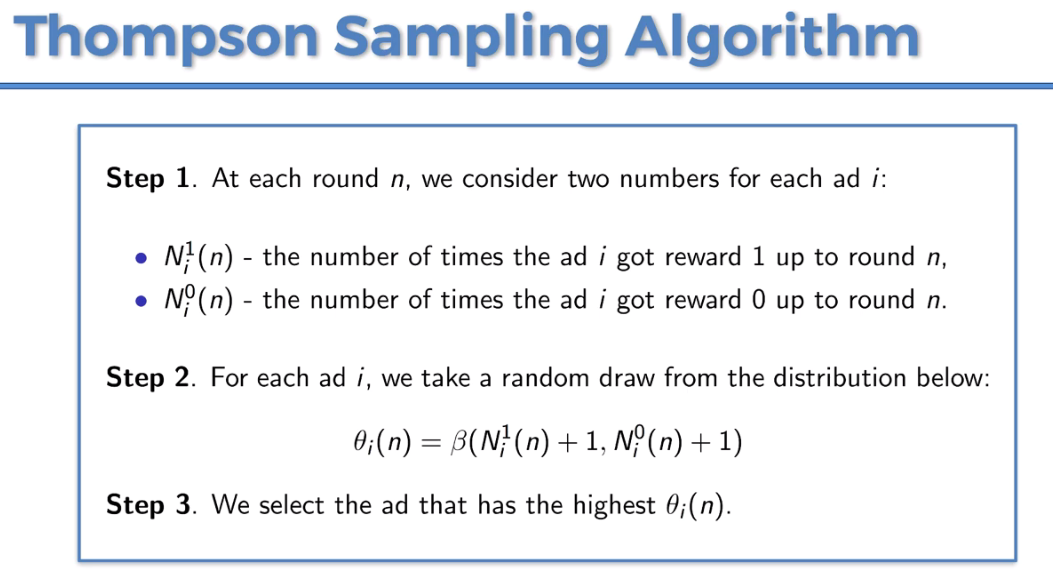
* Until you have made an adjustment to the algorithm based on that value because if you *don't make the adjustment* then nothing changes and you're going to be *stuck*.

1. Whereas in the Thompson Sampling Algorithm can *accommodate* *delayed* *feedback* and it's very important. This basically means that if you pull the lever and you only keep record of the results (say, pulling the lever 500 rounds), and input those data to the Thompson sampling, Thompson Sampling Algorithm will still work.

* Why will it work because if you now run the algorithm without even updating your perception of the world you're still going to get a *new set of hypothetical bandits*. Because you are generating them in a probabilistic manner.
* And this is very important to understand because this gives the Thompson sampling that advantage that you ***don't have to update the algorithm with the result every time***.
* In terms of just terms of Bandits it doesn't really matter that much because if you're playing in the casino and out of sight if some hypothetical person is playing in a casino and they're pulling these lever's they get to the results right away. So they could update Algorithm.
* But in terms of Web sites and ads that is a big deal. Just not even just ***displaying*** ***ads*** on a Web site or you could use this for like ***A/B testing*** the different layouts of your Web site right. You could you could use a Thompson sampling algorithm to have that ***balance*** between ***exploitation*** and ***exploration***
* This sampling algorithm allows you to do is to update your dataset or your information algorithm in a batch manner.

1. ***Thompson*** ***sampling*** ***algorithm*** is actually it has better empirical evidence than they used to be.

* Hence we can conclude that ***Thompson*** ***sampling*** ***algorithm*** would be better choice.



**6.2.3 Implementation of *Thompson* *sampling* *algorithm* in Python**

We introduced a multi armed benefit program for an Ad click through rates (CTR) optimization problem.

* We observed the ***Total Reward*** for Random Selection algorithm (1200 on average, every ad was selected more or less the same number of times) and UCB algorithm. Clearly UCB did the great job (with reward 2170 and we figured out 5th Ad version was the best). Now this ***Thompson sampling algorithm*** is even better than UCB. Because it will figure out which version was the best more quickly (so the reward will be high).
* So we will observe:

1. How better ***Thompson sampling*** is w.r.t UCB according to total reward
2. Which Ad is selected for Maximum Exploitation

* We will *change* the *previous* *code* for *UCB* rather than writing it from *scratch*.

1. Step 1 : First we create the two variables:

* = No. of times ***Ad\_i*** get reward **1** up to round n i.e. No. of REWARDS
* = No. of times ***Ad\_i*** get reward **0** up to round n i.e. No. of PUNISHMENTS

"""

numbers\_of\_selections = [0]\*d

numbers\_of\_rewards = [0]\*d

changed to

        # no. of reward of Ad i up-to trial n i.e. reward = 1,

        # No. of punishment of Ad i up-to trial n i.e. reward = 0,

"""

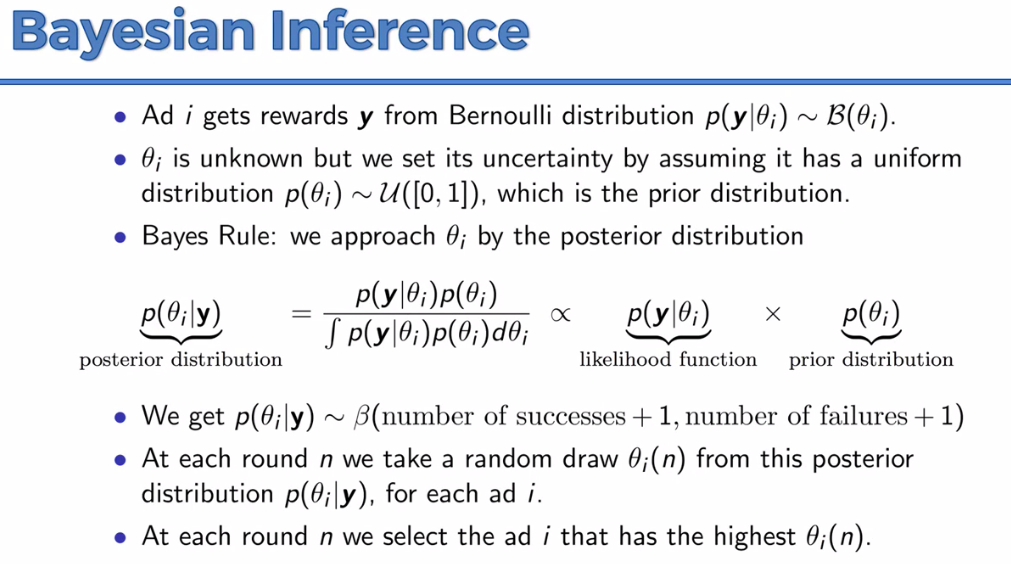
reward\_count\_of\_ads = [0]\*d

punish\_count\_of\_ads = [0]\*d

* As before we created ***two vector/list of 10 elements*** (initialized to **0**). These 2 vectors will keep track to the Rewards and Punishments for each ***10 Ads*** up to trial/round ***n***.

1. Step 2: For each Ad\_i we take a random draw from following distribution, which is the beta distribution.

* We have two important assumptions here which are related to Bayesian inference.



1. So the first assumption is this first line here we suppose that each ***Ad\_i*** gets ***y*** rewards from a Bernouli distribution of parameter . Where is the probability of success.

* And you can picture this ***probability*** ***of*** ***success*** by showing the Ad to a huge amount of users like 1000000 users and could be interpreted as the number of times the outcomes were ***1*** (i.e. the number of successes) divided by the total number of times we selected the Ad that is 1 million.

So basically is the probability of success that is the probability of getting ***Reward 1*** when we select the Ad.

1. Second Assumption: [Recall ***Bayes's Theorem*** in 3.5.1 of Naive Bayes] The second assumption are less stronger than the first one. We assume that has a uniform distribution which has the prior distribution **.**
2. Then we use the Bayes Rule to get to posterior distribution which is . Where given the rewards that we got up to the round **n**.

* So by using Bayes rule that's how we get this -distribution in the step 2.
* So at each ***n*** ***round*** we take a ***random*** ***draw*** from this -distribution. Since these random draws represent the ***probability of success*** **,** we get this strategy: the ***maximum*** of these ***random*** ***draws*** is ***approximating*** the ***highest*** ***probability*** ***of*** ***success***. i.e. At each round n, we select the Ad\_i that has the height ***probability of success*** **:**
* That's the whole idea behind Thompson Sampling: We are trying to
* Estimate these parameters **, , , . . . ,**, the probabilities of success, of each of these ***10*** ***Ads*** then
* By taking these ***random*** ***draws*** and taking the *highest* *of* *them* we're estimating the *highest* *probability* *of* *success* and this highest probability of success corresponds to ***one Specific Ad at each round***.
* So for small amount of round, we might be wrong, but when we take these random draws over thousands of rounds we obtain over all the that corresponds to the Ad that has the Highest Probability Of Success (highest probability of getting reward = 1).

1. Step 3: What we just did. Taking these maximum of these random draw that is the maximum of these estimations of the probability of getting reward equals 1.

* Now we'll implement the strategy composed of step 2 and step 3 in python. Actually it is easier than UCB, we just need to use one method " ***random.betavariate()***" .
* This ***random.betavariate()*** will calculate the **-** distribution for each i-th ***Ad*** (i.e each of 10 Ad). Then we select the ***Ad\_i*** which has maximum value of the calculated -distribution.

**for** n **in** **range**(0, N):

    slct\_ad = 0

    max\_random\_beta = 0

**for** i **in** **range**(0, d):

        random\_beta = random**.betavariate**(reward\_count\_of\_ads[i]+1, punish\_count\_of\_ads[i]+1)

**if**  random\_beta **>** max\_random\_beta:

            max\_random\_beta = random\_beta

            slct\_ad = i

* Next we simulate reward/punish situation and then update Reward and Punish for Each Ad. Also we append the selected Ad to out 10000 long ***slected\_ads*** list.

**for** n **in** **range**(0, N):

    slct\_ad = 0

    max\_random\_beta = 0

**for** i **in** **range**(0, d):

        random\_beta = random**.betavariate**(reward\_count\_of\_ads[i]+1, punish\_count\_of\_ads[i]+1)

**if**  random\_beta **>** max\_random\_beta:

            max\_random\_beta = random\_beta

            slct\_ad = i

    #*storing selected Ad*

    slected\_ads**.append**(slct\_ad)

    #*updating "reward\_count\_of\_ads" and "punish\_count\_of\_ads" of selected Ad "slct\_ad"*

    reward = dataSet**.**values[n, slct\_ad] #*generating reward-simulation from given Dataset*

**if** reward **==** 1:

        reward\_count\_of\_ads[slct\_ad] += 1

**else**:

        punish\_count\_of\_ads[slct\_ad] += 1

    total\_reward += reward

* Now we visualize our result:

#*visualizing the result*

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

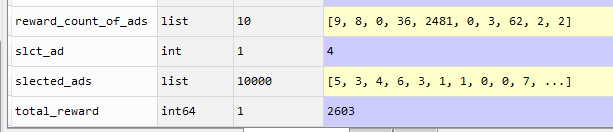
plt**.title**('Histogram of ads selections')

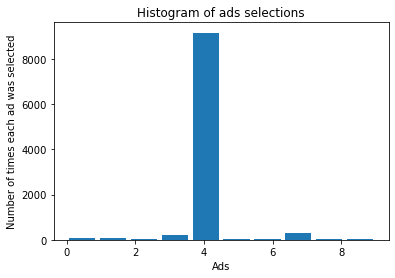
plt**.xlabel**('Ads')

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

plt**.show**()

* Moment of truth: We conclude that the ***Thompson*** ***Sampling*** ***Algorithm*** is better than ***UCB*** ***Algorithm***. It finds the same ***Ad*** (5th) from ***UCB*** but more quickly. Also it doubles the ***Total*** ***reward*** from the ***Random Selection***.





Practiced Version

#*Reinforcement lrarning : ----------  Thompson Sampling. ---------*

#*Ad Click through rate optimization*

**import** pandas **as** pd

**import** numba **as** np

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

#*importing dataset*

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

#*Thompson Sampling Implemnetation.*

"""    # 2 parameters are important:

        # no. of bandit d,

        # No. of trials N,

        """

**import** random

#*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

changed to

        # no. of reward of Ad i at trial n i.e. reward = 1,

        # No. of punishment of Ad i at trial n i.e. reward = 0,

"""

reward\_count\_of\_ads = [0]\*d

punish\_count\_of\_ads = [0]\*d

slected\_ads = []

total\_reward = 0

**for** n **in** **range**(0, N):

    slct\_ad = 0

    max\_random\_beta = 0

**for** i **in** **range**(0, d):

        random\_beta = random**.betavariate**(reward\_count\_of\_ads[i]+1, punish\_count\_of\_ads[i]+1)

**if**  random\_beta **>** max\_random\_beta:

            max\_random\_beta = random\_beta

            slct\_ad = i

    #*storing selected Ad*

    slected\_ads**.append**(slct\_ad)

    #*updating "reward\_count\_of\_ads" and "punish\_count\_of\_ads" of selected Ad "slct\_ad"*

    reward = dataSet**.**values[n, slct\_ad] #*generating reward-simulation from given Dataset*

**if** reward **==** 1:

        reward\_count\_of\_ads[slct\_ad] += 1

**else**:

        punish\_count\_of\_ads[slct\_ad] += 1

    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\_tmpsn\_sml.py*

Instructor Version

#*Thompson Sampling*

#*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 Thompson Sampling*

**import** random

N = 10000

d = 10

ads\_selected = []

numbers\_of\_rewards\_1 = [0] \* d

numbers\_of\_rewards\_0 = [0] \* d

total\_reward = 0

**for** n **in** **range**(0, N):

    ad = 0

    max\_random = 0

**for** i **in** **range**(0, d):

        random\_beta = random**.betavariate**(numbers\_of\_rewards\_1[i] + 1, numbers\_of\_rewards\_0[i] + 1)

**if** random\_beta **>** max\_random:

            max\_random = random\_beta

            ad = i

    ads\_selected**.append**(ad)

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

**if** reward **==** 1:

        numbers\_of\_rewards\_1[ad] = numbers\_of\_rewards\_1[ad] + 1

**else**:

        numbers\_of\_rewards\_0[ad] = numbers\_of\_rewards\_0[ad] + 1

    total\_reward = total\_reward + reward

#*Visualising the results - Histogram*

plt**.hist**(ads\_selected)

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

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

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

plt**.show**()