Mathematics for Intelligent Systems

Topic: Algorithms for Exploration and Exploitation in Reinforcement Learning

Arrun Sivasubramanian(CB.EN.U4AIE19013) and Prashanth VR(CB.EN.U4AIE19047)

B. Tech CSE-AI, Amrita School Of Engineering, Coimbatore - 641112

Introduction:

Sometimes, we all suspect that changing the primary idea of our actions would yield the optimum output and would also make the end-users of the product feel more comfortable. Perhaps more importantly, our basic thought is that making our users feel more comfortable would make them try more of the other events created by us.

But we are often worried that a new action could potentially disorient users and make them feel less comfortable. If that were true, our clever idea to increase rewards might actually make the users participate in fewer events instead. Unsure which of our instincts to trust, we ask for advice from people and make a consensus out of it.

The Approaches:

One approach is to believe that we can't just decide to perform an action then assume that the action is responsible for whatever happens next. For this, we'll need to run a controlled experiment. If we don't test our idea with a controlled experiment, we'll never know whether the action actually helped or hurt our event's progress. After all, there may be so many other parameters that may influence our experiment.

Moreover, if we make the action now and make an assumption that it will be beneficial in the long run too, we'll see a huge reward relative to

the past. But that's not informative about the merits of the new action: for all we know, the new action might actually be hurting future rewards.

Thus we propose two possibilities: The first one being A/B testing setup, Groups A and B of users would see slightly different versions of the event: One including the action while the other without it. This style of controlled experimentation is called A/B testing because it typically involves randomly assigning an incoming user to one of two groups: Group A or Group B. This random assignment of users to groups continues on for a while until we become convinced that either Option A is more successful than Option B or, vice versa, that Option B is more successful than Option A.

After enough users had been exposed to both possibilities, comparisons between the two groups would allow us to decide whether the proposed change would help or hurt the event. After that, we can assign all future users to the more successful version of the event and closes out the inferior version of the event. This experimental approach to trying out new ideas has been extremely successful in the past and will continue to be successful in many contexts.

Now, we may start to consider comparing our current action with several possibilities in its place, including some fairly bad actions. Hence our testing setup approaches multiple possibilities in a matter of minutes.

Another approach is to try to maximize our event's rewards. To keep our progress, we should only run experiments that could be well rewarded. Unless we really believe an action we do has the potential to be valuable, we shouldn't try it at all. And if we don't have any new ideas that we have faith in, going with the traditional action is the best strategy. The threat of losing users is greater than designing experiments using the earlier approach.

Exploration Vs Exploitation:

The obvious questions which arises is: How to decide which action would be highly rewarded without trying them out, which seems to lead us back to square one?

After spending some time, we can decipher that there is always going to be a fundamental trade-off between the goals that motivate us to follow both the approaches to maximize our rewards.

Now that we have understood, we need a simple way to balance the need to:

(1) learn new things and
(2) reward from old things that we had already learned.

We call experimentation as **exploration** and we call reward maximization as **exploitation**. They're the fundamental values that any profit-seeking system, whether it's a person, a company or a robot, has to find a way to balance. If we do too much exploration, we would lose rewards. And if we do too much exploitation, you stagnate and miss out on new opportunities.

Unfortunately, there is no simple answer for the question on what to prioritize. Like we suspected, there is no universal solution to balancing your two goals: to learn which actions are good or bad, we have to explore — at the risk of losing money and bringing in fewer profits/rewards. The right way to choose between exploring new ideas and exploiting the best of our old ideas depends on the details of our event and situation.

Answering this question properly requires that we return to the concepts of exploration and exploitation. Standard A/B testing consists of:

- A short period of pure exploration, in which we assign equal numbers of users to Groups A and B
- A long period of pure exploitation, in which we send all of our users to the more successful version of your action and never come back to the option that seemed to be inferior.

Why might this be a bad strategy?

- It jumps discretely from exploration into exploitation, when we might be able to smoothly transition between the two.
- During the purely exploratory phase, it wastes resources exploring inferior options in order to gather as much data as possible. But we shouldn't want to gather data about strikingly inferior options.

Bandit algorithms provide solutions to both of these problems:

1. They smoothly decrease the amount of exploring they do over time instead of requiring us to make a sudden jump and

2. They focus our resources during exploration on the better options instead of wasting time on the inferior options that are over explored during typical A/B testing.

In fact, bandit algorithms address both of those concerns in the same way because they gradually fixate on the best available options over time. In the academic literature, this process of settling down on the best available option is called convergence. All good bandit algorithms will eventually converge.

The crux is that we have two completely different goals we need to address when we try to optimize an event. We need to:

1. Learn about new ideas (which we'll always call exploring from now on), while we also need to
2. Take advantage of the best of your old ideas (which we'll always call exploiting from now on).

The first approach was meant to embody exploration: we should be open to every new idea, including the terrible ideas of doing bad and unwanted actions. the second was meant to embody exploitation, because we had closed our mind to new ideas prematurely and are overly willing to stick with tradition. To help us get maximum rewards in the event, we'll introduce methods for solving the Explore-Exploit dilemma in this project.

Precisely, we'll discuss four algorithms: Epsilon Greedy Algorithm, Softmax, Annealing Softmax and UCB

Introducing the Epsilon Greedy Algorithm:

In computer science, a greedy algorithm is an algorithm that always takes whatever action seems best at the present moment, even when that decision might lead to bad long term consequences. The epsilon-Greedy algorithm is almost a greedy algorithm because it generally exploits the best available option, but every once in a while the Epsilon-Greedy algorithm explores the other available options. As we'll see, the term epsilon in the algorithm's name refers to the odds that the algorithm explores instead of exploiting.

The epsilon-Greedy algorithm works by randomly oscillating between our first approach of purely randomized experimentation and our second approach in which we try to maximize our rewards. The epsilon-Greedy algorithm is one of the easiest Bandit algorithms to understand because it tries to be fair to the two opposite goals of exploration and exploitation

by using a mechanism that even a little kid could understand: It just flips a coin. While there are a few details we'll have to iron out to make that statement precise, the big idea behind the Epsilon-Greedy algorithm really is that simple: if you flip a coin and it comes up heads, you should explore for a moment. But if the coin comes up tails, you should exploit.

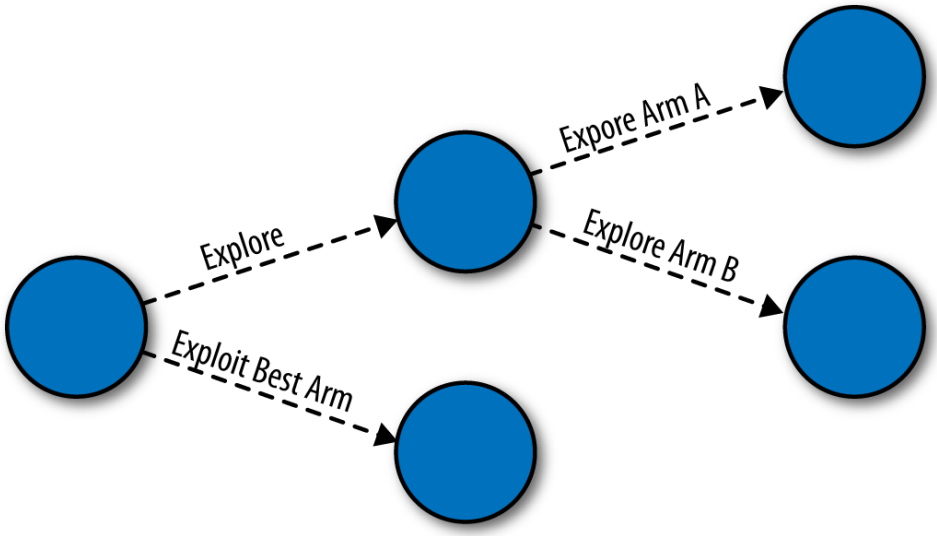
The Algorithm is as follows:

I. When a new user arrives, the algorithm flips a coin that comes up tails with probability epsilon. (If something happens 100 * X percent of the time, saying that a coin comes up tails with probability 0.01 means that it comes up tails 1% of the time.)

II. If the coin comes up heads, the algorithm is going to exploit. To exploit, the algorithm looks up the historical conversion rates for both the events in whatever data source it uses to keep track of things. After determining which action had the highest success rate in the past, the algorithm decides to show the new visitor the action that's been most successful historically.

III. If, instead of coming up heads, the coin comes up tails, the algorithm is going to explore. Since exploration involves randomly experimenting with the two actions being considered, the algorithm needs to flip a second coin to choose between them. Unlike the first coin, we'll assume that this second coin comes up head 50% of the time. Once the second coin is flipped, the algorithm can move on with the last step of the procedure:

    a. If the second coin comes up heads, show the new visitor the first possibility.
    b. If the second coin comes up tails, show the new visitor the second possibility.



Epsilon Greedy Algorithm for two arms

The crux of this discussion is:

After letting this algorithm loose on the users for a long time, we'll see that it works by oscillating between (A) exploiting the best option that it currently knows about and (B) exploring at random among all of the options available to it. In fact, you know from the definition of the algorithm that for an N-armed Bandit problem:

- With probability 1 – epsilon, the epsilon-Greedy algorithm exploits the best known option.
- With probability epsilon / N, the epsilon-Greedy algorithm explores the best known option.
- With probability epsilon / N, the epsilon-Greedy algorithm explores the worst known option.

Abstracting our problem for the machine:

Before we write code for the epsilon-Greedy algorithm, we need to abstract away from our example in which we wanted to compare our actions. We'll do this in a couple of simple steps that also serve to

introduce some of the jargon terms we'll be using throughout the rest of the book.

First, we want to consider the possibility that we have hundreds or thousands of actions to choose from, rather than just two. In general, we're going to assume that we have a fixed set of N different options and that we can enumerate them, so that Option N is analogous to n^{th} Action. For historical reasons, these options are typically referred to as arms, so we'll talk about Arm 1 and Arm 2 and Arm N rather than Option 1, Option 2 or Option N.

But the main idea is the same regardless of the words we choose to employ. That said, it will help you keep track of these sorts of jargon terms if we explain why the options are typically called arms.

Next, we need to define a reward. A reward is simply a measure of success: it might tell us whether a customer clicked on an ad or signed up as a user. What matters is simply that:

I. A reward is something quantitative that we can keep of track of mathematically and that

II. Larger amounts of reward are better than smaller amounts.

The Bandit Problem:

We can describe the abstract idea of a bandit problem that motivates all of the algorithms we'll implement in this project:

1) We're facing a complicated slot machine, called a Bandit, that has a set of N arms that we can pull on.

2) When pulled, any given arm will output a reward. But these rewards aren't reliable, which is why we're gambling: Arm 1 might give us 1 unit of reward only 1% of the time, while Arm 2 might give us 1 unit of reward only 3% of the time. Any specific pull of any specific arm is risky.

3) Not only is each pull of an arm risky, we also don't start off knowing what the reward rates are for any of the arms. We have to figure this out experimentally by actually pulling on the unknown arms.

So far the problem we've described in just a problem in statistics: we need to cope with risk by figuring out which arm has the highest average reward. We can calculate the average reward by pulling on each arm a lot of times and computing the mean of the rewards you get back. But a real bandit problem is more complicated and also more realistic.

What makes a bandit problem special is that we only receive a small amount of the information about the rewards from each arm. Specifically:

1) We only find out about the reward that was given out by the arm we actually pulled.

2) Whichever arm we pull, we miss out on information about the other arms that we didn't pull. Just like in real life, you only learn about the path you took and not the paths you could have taken.

In fact, the situation is worse than that. Not only do we get only partial feedback about the wisdom of our past decisions, we're literally falling behind every time we don't make a good decision: Every time we experiment with an arm that isn't the best arm, we lose reward because we could, at least in principle, have pulled on a better arm.

The full Multiarmed Bandit Problem is defined by the five features above. Any algorithm that offers us a proposed solution to the Multiarmed Bandit Problem must give us a rule for selecting arms in some sequence. And this rule has to balance out your competing desires to:

A. Learn about new arms and
B. Earn as much reward as possible by pulling on arms we already know are good choices.

Implementing the Epsilon Greedy Algorithm:

First, we define a class of objects that represents an epsilon-Greedy algorithm as it's going to be deployed in the wild. This class will encapsulate the following pieces of information:

1. epsilon
   This will be a floating point number that tells us the frequency with which we should explore one of the available arms. If we set epsilon = 0.1, then we'll explore the available arms on 10% of our pulls.

2. n
   The total number of arms of the bandit

3. N
   A scalar value of integer that tells us how many times we've played the n arms available to us in the current bandit problem.
4. mean
   A vector of floating point numbers that defines the average amount of reward we've gotten when playing each of the n arms available to us. If Arm 1 gave us 1 unit of reward on one play and 0 on

another play, while Arm 2 gave us 0 units of reward on both plays, then we'll set values to 0.5 and 0.0 for the arms respectively.

5. reward
   It helps us to note what is the reward that we can expect by pulling that arm after we do the action in the given event.

Now that we have a class that represents all of the information that the epsilon-Greedy algorithm needs to keep track of about each of the arms, we need to define two types of behaviours that any algorithm for solving the Multiarmed Bandit Problem should provide:

1) choose: Every time we have to make a choice about which arm to pull, we want to be able to simply make a call to our favourite algorithm and have it tell us the numeric reward of the arm we are pulling. We also add a random noise to that reward in this function so that we ensure that the reward we get after performing this action is not constant as we see in practical situations.

2) update: After we choose an arm, we get a reward signal back from our system. We want to update our algorithm's beliefs about the quality of the arm we just chose by providing this reward information. All of the bandit algorithms handle this by providing an update function that takes as arguments
   a. an algorithm object, and
   b. the reward received from choosing that arm.

The update method will take this information and make the relevant changes to the algorithm's evaluation of all of the arms.

As we can see, the epsilon-Greedy algorithm handles selecting an arm in two parts: (1) we flip a coin to see if we'll choose the best arm we know about and then (2) if the coin comes up tails, we'll select an arm completely at random. In Python, we've implemented this by checking if

a randomly generated number is greater than epsilon. If so, our algorithm selects the arm whose cached value according to the values field is highest; otherwise, it selects an arm at random.

But there are weaknesses with this approach as well. The first weakness is that, as you get more certain which of our actions is best, this tendency to explore the worse action a full 5% of the time will become more wasteful. In the jargon of bandit algorithms, we'll be over exploring.

And there's another problem with a fixed 10% exploration rule: at the start of our experimentation, we'll choose options that we don't know much about far more rarely than we'd like to because we only try new options 10% of the time.

Indeed, bandit algorithms exemplify two types of learning that are not present in standard ML examples: active learning, which refers to algorithms that actively select which data they should receive; and online learning, which refers to algorithms that analyse data in real-time and provide results on the fly. The behaviour of the algorithm depends on the data it sees, but the data the algorithm sees depends on the behaviour of the algorithm. Debugging a bandit algorithm is therefore substantially more complicated than debugging a straight machine learning algorithm that isn't doing active learning.

Code to get the output for epsilon Greedy is as follows:

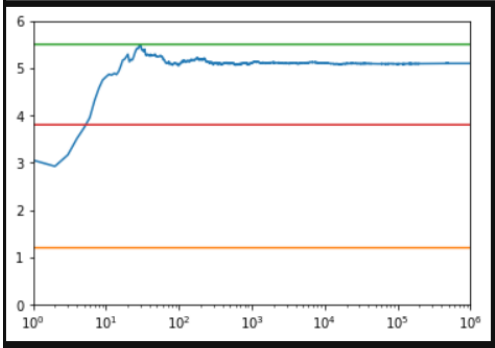
```
import numpy as np
from random import *
import matplotlib.pyplot as plt

n = int(input("Enter Number of Arms:")) # number of arms
N = int(input("Enter Number of Iterations:")) # number of simulations
epsilon = float(input("Enter Value of Epsilon:")) # freq to explore/exploit
reward = np.zeros(n, dtype = "float32")
print("Enter Values of Rewards:")
for i in range(n):
    reward[i] = float(input())

print(f"Input of Rewards: {reward}")
```

```
Enter Number of Arms: 3
Enter Number of Iterations: 1000000
Enter Value of Epsilon: 0.2
Enter Values of Rewards:
 1.2
 5.5
 3.8
Input of Rewards: [1.2 5.5 3.8]
```

```python
# Define Action class
class Actions:
    def __init__(self, m):
        self.m = m # reward for arm
        self.mean = 0 # average reward for arm
        self.N = 0 #Number of Simulations

  # Choose a random action
    def choose(self):
        return np.random.randn() + self.m  #adding a random number to reward to add noise

  # Update the action-value estimate
    def update(self, x):
        self.N += 1
        self.mean = (1 - 1.0 / self.N)*self.mean + 1.0 / self.N * x

    def run_experiment(rewards, eps, N):
        actions = []
        for i in range(len(rewards)):
            actions.append(Actions(rewards[i]))
        data = np.empty(N)

        for i in range(N):
            # epsilon greedy
            p = np.random.random()
            if p < eps:
                j = np.random.choice(3)
            else:
                j = np.argmax([a.mean for a in actions])

            x = actions[j].choose()
            actions[j].update(x)

            # for the plot
            data[i] = x
        cumulative_average = np.cumsum(data) / (np.arange(N) + 1)

        # plot moving average ctr
        plt.plot(cumulative_average)
        for i in range(len(rewards)):
            plt.plot(np.ones(N)*rewards[i])
            plt.xscale('log')
            plt.xlim(1,N)
            plt.ylim(0,6)
        plt.show()

        for a in actions:
            print(a.mean)
        return cumulative_average
```

Plotting the results:

```
c_1 = Actions.run_experiment(reward, epsilon, N)
```




```
1.1999243462553104
5.498902698711005
3.7935226627662813
```

Analysing Results of Epsilon Greedy Algorithm:

With the results from our simulation study in hand, we can analyse our data in several different ways to assess the performance of our algorithms

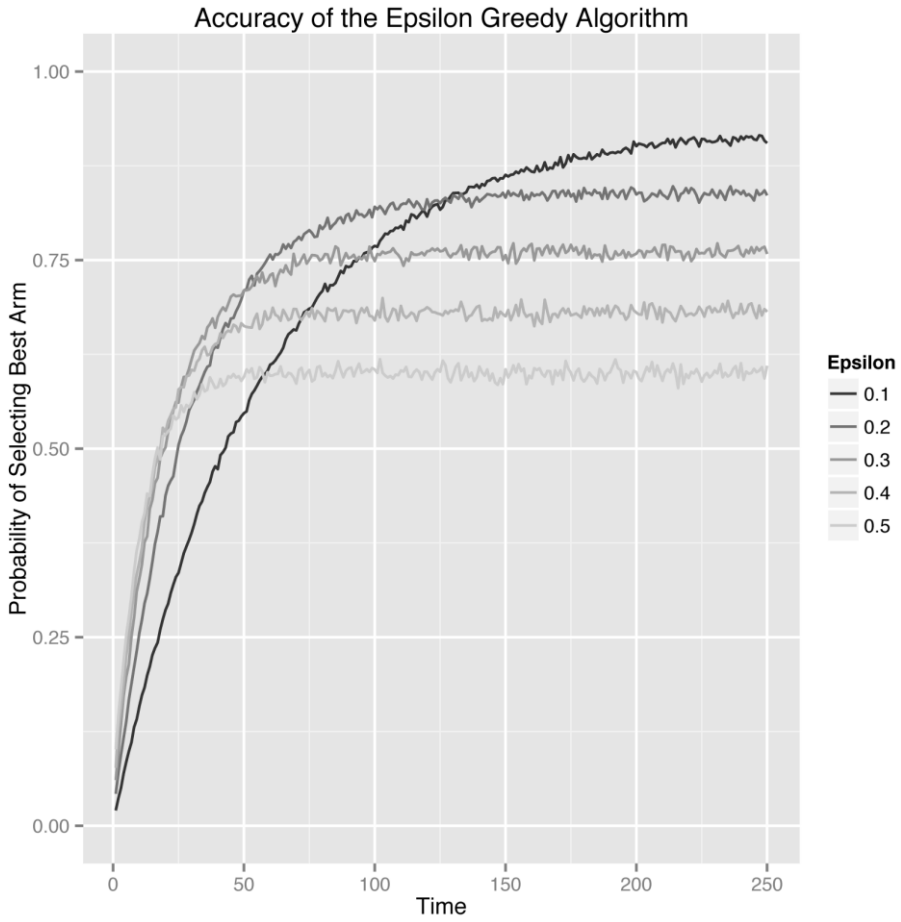
Approach 1: Track the Probability of Choosing the Best Arm:

The first analytic approach, and certainly the simplest, is to keep track of the odds that our algorithm selects the best arm at any given point in time. We need to work with odds for two different reasons:

1. The algorithm uses randomization when choosing. which arm to pull and may therefore not select the best arm even after it's learned which arm is best and

2. The rewards that the algorithm receives are random as well. For those reasons, there is always a chance that the algorithm will not make the best decision on any specific trial.

As such, we'll explicitly calculate the probability of selecting the best arm by estimating the percentage of times in our simulations when the algorithm chose the best arm. If the probability that the algorithm picks the best arm doesn't go up over time, then we don't really have any evidence that our algorithm is learning anything about the values of the arms. Thankfully, the results which split up the data by the value of epsilon, show that our algorithm does indeed learn no matter how we set epsilon.



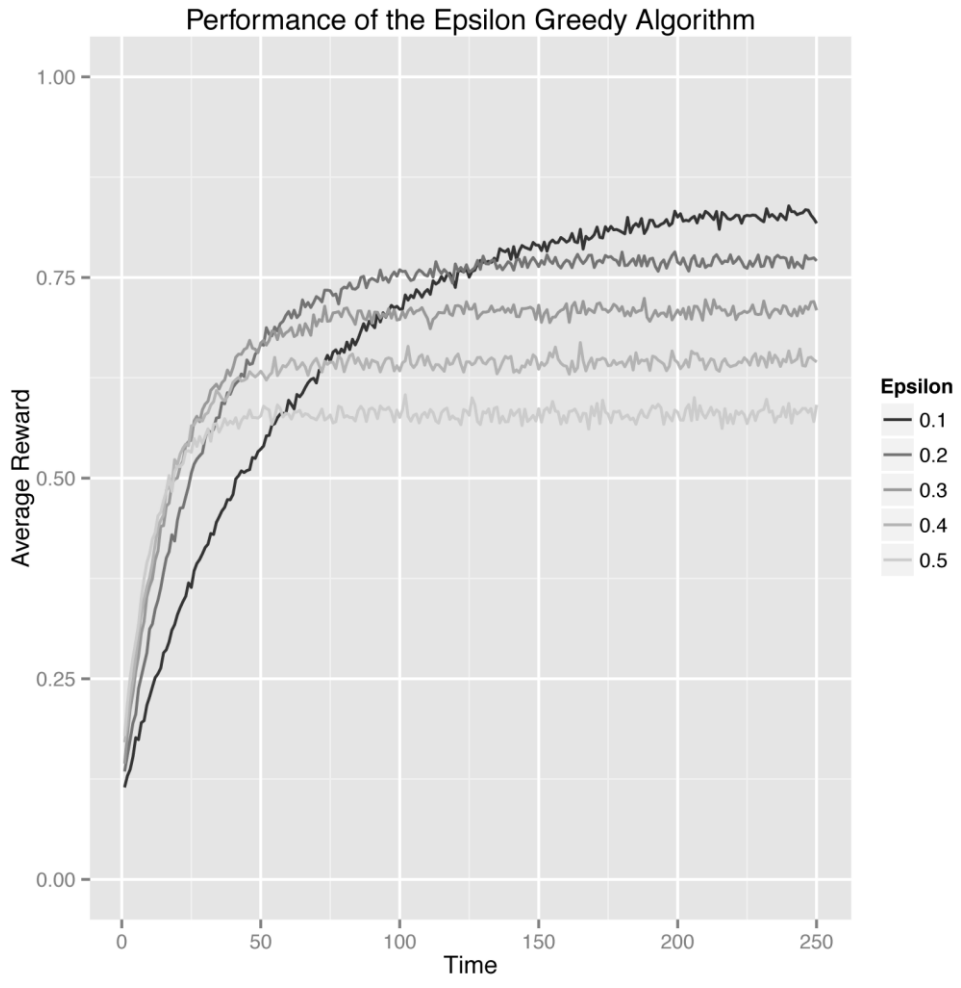
The first thing you should notice is that each setting of epsilon produces a separate curve. The curves with high values of epsilon go up very quickly, but then peak out. That's a given, because the epsilon-Greedy algorithms choose randomly with probability epsilon. If epsilon is high, we explore a lot and find the best arm quickly, but then we keep exploring even after it's not worth doing anymore. When epsilon is high, our system's peak performance is fairly low.

In contrast to the high values of epsilon, the lowest value of epsilon, which was 0.1, causes the algorithm to explore much more slowly, but eventually the algorithm reaches a much higher peak performance level. In many ways, different settings of Epsilon embody the Explore-Exploit trade-off as a whole: the settings that lead to lots of exploration learn quickly, but don't exploit often enough at the end; whereas the settings that lead to little exploration learn slowly, but do well at the end.

As we can see, which approach is best depends on which point in time we're looking at. This is why we need to always be concerned about the length of time we intend to leave a bandit algorithm running. The number of pulls that the bandit algorithm has left to it is typically called the horizon. Which algorithm is best depends strongly on the length of the horizon.

Approach 2: Track the Average reward at each point of time:

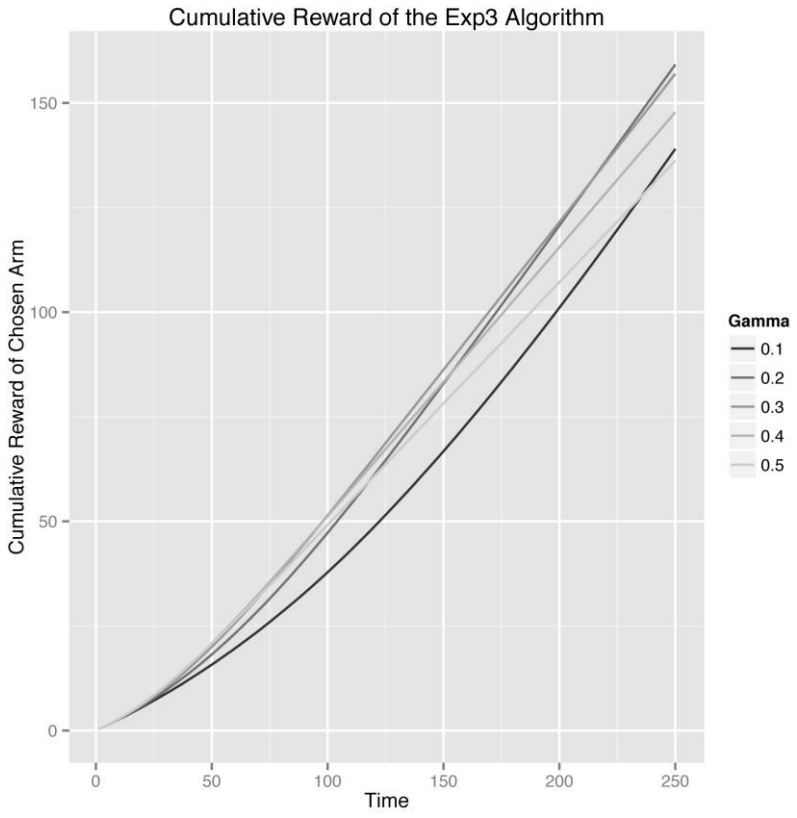
Instead of looking at the probability of picking the best arm, another simple approach to measuring our algorithm's performance is to use the average reward that our algorithm receives on each trial. When there are many arms similar to the best, each of which is just a little worse than the best, this average reward approach is a much better method of analysis than our approach using probabilities of selecting the best arm.



Approach 3: Track the Cumulative reward at each point of time:

Ultimately, there's something lacking in both of the approaches we've taken: they're too narrowly focused on the performance of the algorithm at each fixed point in time and don't give us a gestalt picture of the lifetime performance of our algorithm. This myopic focus on each point in time in isolation is unfair to versions of the epsilon-Greedy algorithm in which epsilon is large, because these algorithms, by definition, explore worse options more frequently than algorithms for which epsilon is low. To decide whether that increased exploration is worth the trouble, we shouldn't focus on the performance of the algorithm at any specific point in time, but rather on its cumulative performance over time.

To do that, we can analyse the cumulative reward of our algorithms, which is simply the total amount of reward that an algorithm has won for us up until some fixed point in time. This cumulative reward is important, because it treats algorithms that do a lot of exploration at the start as a means of finding the best available arm more fairly.



As we can see in a theoretical analysis based on cumulative rewards leads to a much cleaner assessment of the performance of the different values of epsilon that we tested.

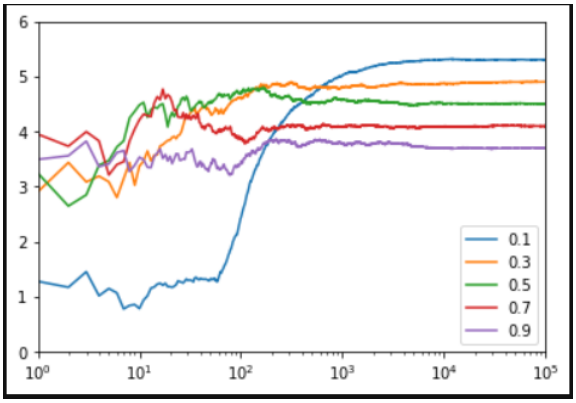
In fact, looking at the cumulative results, we can see that curves for the low value of epsilon = 0.1 and the high value of epsilon = 0.5 intersect after about 130 rounds: before then, it was good to have explored a lot, but after then it was better to have been able to exploit more consistently once the algorithm had found the best arm.

That said, all of our figures convey the same basic message: the epsilon-Greedy algorithm does eventually figure out which arm is best no matter how epsilon is set. But the length of time required to figure out which arm is best depends a lot on the value of epsilon. What's appropriate for you depends on how long you intend to run your algorithm and how different the arms you're testing will be.

Code to simulate the results practically is as follows:

```
def testEpsilon(reward,N):
    labels = []
    for i in range(1,10,2):
        labels.append(i/10)
        c_1 = Actions.run_experiment(reward,i/10, N)
        plt.plot(c_1)
        plt.xscale('log')
        plt.xlim(1,N)
        plt.ylim(0,6)
        plt.legend(labels,loc="best")
    plt.show()

testEpsilon(reward,100000)
```

Plot is as follows:



Introducing the Softmax Algorithm:

There's an obvious problem in the epsilon-Greedy algorithm: It explores options completely at random without any concern about their merits.

This is a problem for several reasons:

1. If the difference in reward rates between two arms is small, we'll need to explore a lot more often than 10% of the time to correctly determine which of the two options is actually better.

2. In contrast, if the difference is large, we need to explore a lot less than 10% of the time to correctly estimate the better of the two options. For that reason, we'll end up losing a lot of reward by exploring an unambiguously inferior option in this case.

When we first described the epsilon-Greedy algorithm, we said that we wouldn't set epsilon = 1.0 precisely so that we wouldn't waste time on inferior options, but, if the difference between two arms is large enough, we end up wasting time on inferior options simply because the epsilon-Greedy algorithm always explores completely at random.

Putting these two points together, it seems clear that there's a qualitative property missing from the epsilon-Greedy algorithm. We need to make our bandit algorithm care about the known differences between the estimated values of the arms when our algorithm decides which arm to explore.

We need structured exploration rather than the haphazard exploration that the epsilon-Greedy algorithm provides. The first algorithm we'll describe that takes this structural information into account is called the Softmax algorithm. The Softmax algorithm tries to cope with arms differing in estimated value by explicitly incorporating information

about the reward rates of the available arms into its method for choosing which arm to select when it explores.

We can get an initial intuition for how the Softmax algorithm handles this problem by imagining that we choose each arm in proportion to its estimated value. Suppose that we have two arms, A and B. Now imagine that, based on our past experiences, these two arms have had two different rates of success: rA and rB. With those assumptions, the most naive possible implementation of a Softmax-like algorithm would have we choose Arm A with probability rA / (rA + rB) and Arm B with probability rB / (rA + rB).

In practice, this very naive algorithm isn't something people actually use. To reconstruct the algorithm people actually use, we need to make two changes to it.

First, we will calculate a different scale for reward rates by exponentiating our estimates of rA and rB. Using this new scale, we will choose Arm A with probability exp(rA) / (exp(rA) + exp(rB)) and Arm B with probability exp(rB) / (exp(rA) + exp(rB)). This naive exponential rescaling has the virtue of not behaving strangely if you someone used negative numbers as rates of success, since the call to exp will turn any negative numbers into positive numbers and insure that the negative numbers in the denominator of these fractions can't cancel out any positive numbers that may be found in the denominator. This additional parameter is a different sort of scaling factor than the exponentiation we just introduced.

This new type of scaling factor is typically called a temperature parameter based on an analogy with physics. In fact, the full Softmax algorithm is closely related to a concept called the Boltzmann distribution in physics, which is used to describe how groups of particles

behave. We'll call this new temperature parameter tau. We introduce tau to produce the following new algorithm:

• At time T, select one of the two arms with probabilities computed as follows:

— exp(rA / tau) / (exp(rA / tau) + exp(rB / tau))

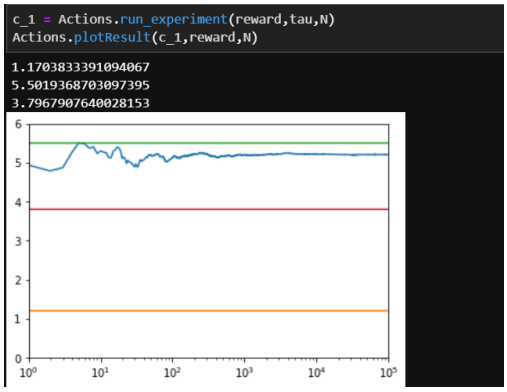
— exp(rB / tau) / (exp(rA / tau) + exp(rB / tau))

• For whichever arm we picked, update our estimate of the mean using the same update rule we used for the epsilon-Greedy algorithm.

Code to get the output for Softmax Algorithm is as follows:

```
# Define Action class
class Actions:
    def __init__(self, m):
        self.m = m # reward for arm
        self.mean = 0 # average reward for arm
        self.N = 0 #Number of Simulations

  # Choose a random action
    def choose(self):
        return np.random.randn() + self.m  #adding a random number to reward to add noise

  # Update the action-value estimate
    def update(self, x):
        self.N += 1
        self.mean = (1 - 1.0 / self.N)*self.mean + 1.0 / self.N * x

    def run_experiment(rewards, temp, N):
        def softMax(rew):
            den = np.sum(np.exp(rew/temp))
            expr = np.zeros(len(rew))
            for i in range(len(rew)):
                expr[i] = np.exp(rew[i]/temp)/den

            return np.round(expr,2)

        actions = []
        for i in range(len(rewards)):
            actions.append(Actions(rewards[i]))
        data = np.empty(N)

        for i in range(N):
            # Softmax Algorithm
            j = np.random.choice(3, p = softMax(rewards))
            x = actions[j].choose()
            actions[j].update(x)

            # for the plot
            data[i] = x
        cumulative_average = np.cumsum(data) / (np.arange(N) + 1)

        for a in actions:
            print(a.mean)
        return cumulative_average

    def plotResult(cum_avg,rewards,N):
        # plot moving average ctr
        plt.plot(cum_avg)
        for i in range(len(rewards)):
            plt.plot(np.ones(N)*rewards[i])
            plt.xscale('log')
            plt.xlim(1,N)
            plt.ylim(0,6)
        plt.show()
```

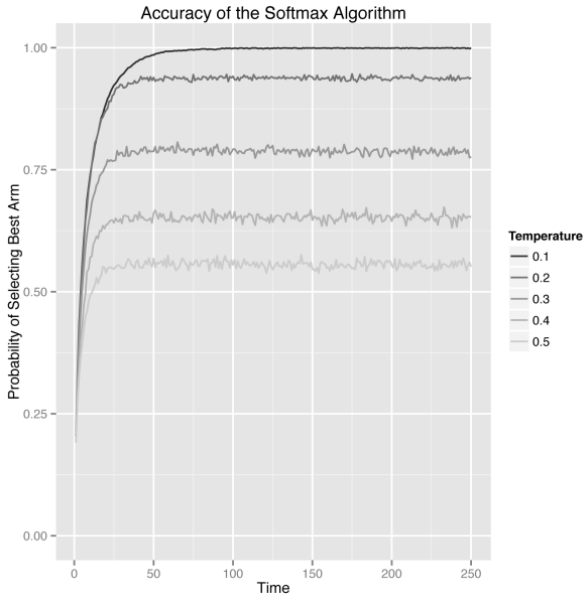
Plotting the results: (we must note that we are able to converge almost 10 times faster than Epsilon Greedy)



What the temperature parameter tau does? It's easiest to think of tau as letting us shift the behaviour of the Softmax algorithm along a continuum defined by two extreme ways to select arms. At one extreme, we set tau = inf. This will give us a fully deterministic choice of the arm that has the highest estimated value. At the other extreme, we set tau = Inf, which gives us purely random exploration like we got out of the epsilon-Greedy algorithm.

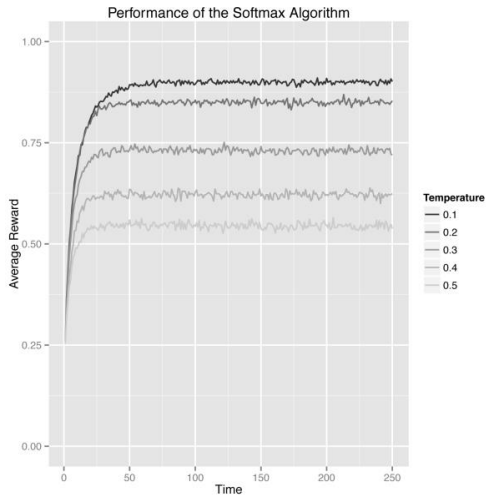
Analysing Results of Softmax Algorithm:

As before, we'll walk through some analyses of the results using simple graphics. For now we can just eyeball results from our Softmax simulations to get a sense of how it performs relative to the expectations set by the epsilon-Greedy algorithm. As we did with the epsilon-Greedy algorithm, we'll start by plotting the probability that the Softmax algorithm selects the best arm.



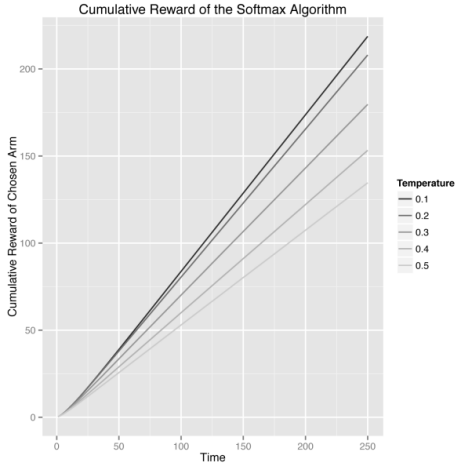
Looking at the probability that the Softmax algorithm selects the right arm, we see that it starts to select the right arm 100% of the time if we wait long enough. This is a big improvement over the epsilon-Greedy algorithm, which never got past the intrinsic errors caused by the purely random exploration strategy it used. In our standard test case, there is one standout arm and the Softmax algorithm is able to find it and exploit it far more effectively than the epsilon-Greedy algorithm.

We see the same results looking at the average reward rates below:



Finally, we can get more insight into the different temperature settings we can use by looking at the cumulative rewards earned as shown below. In this figure, we can see that the lowest temperature setting we used, 0.1, gives us the best results nearly right from the start. This is typical in

settings in which there are clear differences between arms. It's much less likely that the low level of exploration we're seeing from the 0.1 temperature Softmax will be ideal if the arms are closer together.

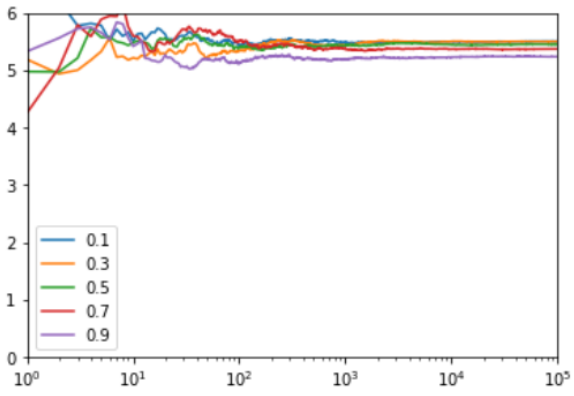


That said, all of our figures convey the same basic message: The Softmax algorithm does eventually figure out which arm is best. The length of time required to figure out which arm is best is also resolved by setting probabilities proportionate to reward.

Code to simulate the results practically is as follows:

```
def testSoftmax(reward,N):
    labels = []
    for i in range(1,10,2):
        labels.append(i/10)
        c_1 = Actions.run_experiment(reward,i/10, N)
        plt.plot(c_1)
        plt.xscale('log')
        plt.xlim(1,N)
        plt.ylim(0,6)
        plt.legend(labels,loc="best")
    plt.show()

testSoftmax(reward,100000)
```

Plot is as follows:



Annealing Softmax Algorithm:

On testing the Epsilon-Greedy Algorithm, it's often a good idea to encourage an algorithm to explore less over time. In the Softmax algorithm, we can achieve that by slowly decreasing the temperature, which we call annealing.

For bandit algorithms, we can metaphorically decrease the temperature by changing the temperature parameter. This will make our algorithm exploit the best arm more often and settle into its final deterministic strategy for choosing an arm.
But, before we go any further, let's give two equivalent definitions of the term annealing:

- Annealing is the process of decreasing the temperature in a Softmax algorithm over time.
- Annealing is a process of modifying a bandit algorithm's behaviour so that it will explore less over time.

In that case, we set temperature in a logarithmic scale e.g 1/math.log(1.0000001), which is very close to being infinite. So the temperature is extremely high and the system will explore almost completely randomly.

But, as t goes up, the temperature will get lower and lower. Because we use logarithms, this decrease isn't extremely rapid: rather, it's just rapid enough to be effective. This division by a logarithmically scaled version of t isn't always the best approach to annealing. **In the code, we'll do a linear decrease of temperature for convenience.**

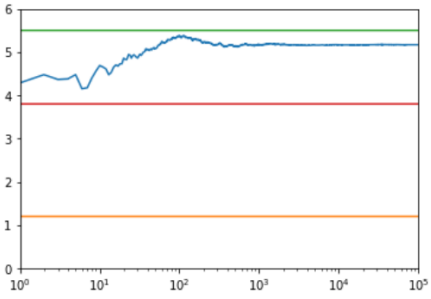
Code to get the output for Annealing Softmax Algorithm is as follows:

```
# Define Action class
class Actions:
    def __init__(self, m):
        self.m = m # reward for arm
        self.mean = 0 # average reward for arm
        self.N = 0 #Number of Simulations

  # Choose a random action
    def choose(self):
        return np.random.randn() + self.m  #adding a random number to reward to add noise

  # Update the action-value estimate
    def update(self, x):
        self.N += 1
        self.mean = (1 - 1.0 / self.N)*self.mean + 1.0 / self.N * x

    def run_experiment(rewards, temp, delt, N):
        def softMax(rew):
            den = np.sum(np.exp(rew/temp))
            expr = np.zeros(len(rew))
            for i in range(len(rew)):
                expr[i] = np.exp(rew[i]/temp)/den


            return np.round(expr,1)

        actions = []
        for i in range(len(rewards)):
            actions.append(Actions(rewards[i]))
        data = np.empty(N)
```

```
        for i in range(N):
            # Softmax Algorithm
            j = np.random.choice(3, p = softMax(rewards))
            x = actions[j].choose()
            actions[j].update(x)

            # for the plot
            data[i] = x
        cumulative_average = np.cumsum(data) / (np.arange(N) + 1)

        for a in actions:
            print(a.mean)

        temp = temp*(1-delt) # annealing temperature
        return cumulative_average

    def plotResult(cum_avg,rewards,N):
        # plot moving average ctr
        plt.plot(cum_avg)
        for i in range(len(rewards)):
            plt.plot(np.ones(N)*rewards[i])
            plt.xscale('log')
            plt.xlim(1,N)
            plt.ylim(0,6)
        plt.show()
```

```
c_1 = Actions.run_experiment(reward,tau,rate,N)
Actions.plotResult(c_1,reward,N)
```

Plot is as follows:



Introducing the Upper Confidence Bound Algorithm:

The algorithms we've presented so far have one systematic weakness: they don't keep track of how much they know about any of the arms available to them. They pay attention only to how much reward they've gotten from the arms. This means that they'll under-explore options whose initial experiences were not rewarding, even though they don't have enough data to be confident about those arms. We can do better by using an algorithm that pays attention to not only what it knows, but also how much it knows. The algorithm, UCB, that we'll present in this report does exactly this.

Before we describe how the UCB algorithm keeps track of how much it knows, let's look back at the epsilon-Greedy and Softmax algorithms and take a more abstract perspective on them. Both the epsilon-Greedy algorithm and the Softmax algorithm share the following broad properties:

• The algorithm's default choice is to select the arm that currently has the highest estimated value.

• The algorithm sometimes decides to explore and chooses an option that isn't the one that currently seems best:

• The UCB can make decisions to explore that are driven by our confidence in the estimated value of the arms we've selected.

Why is it important to keep track of our confidence in the values of the arms?

The reason has to do with the nature of the rewards we receive from the arms: they're noisy. If we use our past experiences with an arm, then the estimated value of any arm is always a noisy estimate of the true return

on investment we can expect from it. Because of this noise, it might just be a coincidence that Arm A seems better than Arm B; if we had more experience with both arms, we'd eventually realize that Arm B is actually better.

The epsilon-Greedy and Softmax algorithms aren't robust to this noise during their first experiences with things. Or, to put things in more human terms, the epsilon-Greedy and Softmax algorithms are gullible. They are easily misled by a few negative experiences. Because of their use of randomness, they can make up for this later. UCB takes a very different approach. As you'll see, UCB does not use randomness at all. Instead, UCB avoids being gullible by requiring us to keep track of our confidence in our assessments of the estimated values of all of the arms. To do that, we need to have some metric of how much we know about each arm. Thankfully, we already have information on hand that will give us that metric: we've been explicitly keeping track of the number of times we've pulled each arm for both of the algorithms we've used so far. Inside of the counts field in our epsilon-Greedy and Softmax classes, we have enough information to calculate a simple metric of our confidence in the estimated values of the various arms. We just need to find a way to take advantage of that information.

In addition to explicitly keeping track of our confidence in the estimated values of each arm, the UCB algorithm is special for two other reasons:

• UCB doesn't use randomness at all. Unlike epsilon-Greedy or Softmax, it's possible to know exactly how UCB will behave in any given situation. This can make it easier to reason about at times.

• UCB doesn't have any free parameters that you need to configure before you can deploy it. This is a major improvement if you're interested in running it in the wild, because it means that you can start to use UCB without having a clear sense of what you expect the world to

behave like. Taken together, the use of an explicit measure of confidence, the absence of unnecessary randomness and the absence of configurable parameters makes UCB very compelling. UCB is also very easy to understand, so let's just present the algorithm and then we can continue to discuss it in more detail.

Implementing UCB Algorithm:

The UCB algorithm is using this line to insure that it has played every single arm available to it at least once. This is UCB's clever trick for insuring that it doesn't have a total cold start before it starts to apply its confidence-based decision rule. It's important to keep this initialization step in mind when you consider deploying UCB1: if you will only let the algorithm run for a small number of plays (say M) and you have many arms to explore (say N), it's possible that UCB will just try every single arm in succession and not even make it to the end. If M < N, this is definitely going to occur. If M is close to N, you'll still spend a lot of time just doing this initial walkthrough.

Whether that is a good or bad thing is something you need to consider before using UCB. But, if you have a lot of plays ahead of you, this initial pass through all of the arms is a very good thing. It insures that the UCB algorithm knows a little bit about all available options, which makes it very effective when there are clearly inferior arms that can be essentially ignored right from start. Once we've gotten past the initial cycling through all of the available arms, UCB1's real virtues kick in. As you can see, the select_arm method for UCB1 uses a special type of value that we've called a ucb_value in this code.

The ucb_value combines the simple estimated value of each arm with a special bonus quantity, which is math.sqrt((2 * math.log(total_counts)) / float(self.counts[arm])). The most basic statement that can be made

about it is that it augments the estimated value of any arm with a measure of how much less we know about that arm than we know about the other arms. That claim can be confirmed by considering what happens if you ignore everything except for math.log(to tal_counts) / float(self.counts[arm]). If counts[arm] is small relative to to tal_counts for a certain arm, this term will be larger than when counts[arm] is large relative to total_counts. The effect of that is that UCB is a explicitly curious algorithm that tries to seek out the unknown.

The other factors around this core unit of curiosity are essentially rescaling terms that make UCB work properly. For those interested in more formal details, these rescaling terms allow the algorithm to define a confidence interval that has a reasonable chance of containing the true value of the arm inside of it. UCB creates its ucb_values by replacing every arm's estimated value with the upper bound on the confidence interval for its value. This is why the algorithm is the Upper Confidence Bound algorithm.

The bonus quantity becomes a big boost in the effective value of the arm for arms that we know little about. That means that we try hard to learn about arms if we don't know enough about them, even if they seem a little worse than the best arm. In fact, this curiosity bonus means we'll even occasionally visit the worst of the arms we have available.

In fact, this curiosity bonus means that UCB can behave in very surprising ways. This graph come from UCB back pedalling and experimenting with inferior arms because it comes to the conclusion that it knows too little about those arms. This back pedalling matters less and less over time, but it's always present in UCB's behaviour, which means that UCB doesn't become a strictly greedy algorithm even if you have a huge amount of data.

Code to implement UCB is as follows:

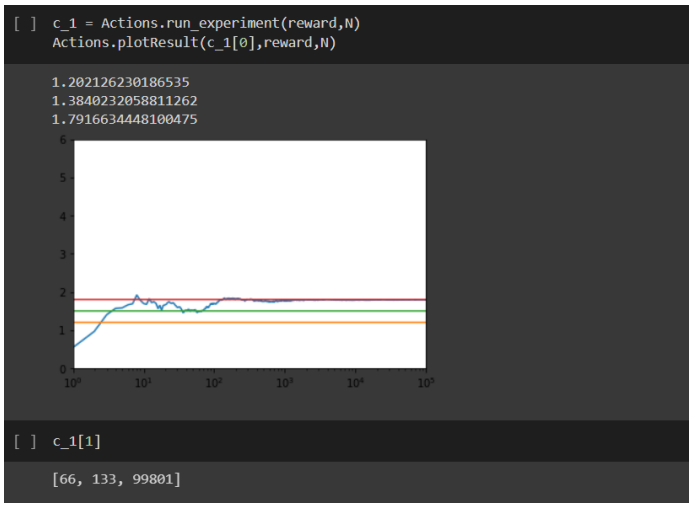
```
[ ] import numpy as np
    from random import *
    import matplotlib.pyplot as plt
```

```
[ ] n = int(input("Enter Number of Arms: ")) # number of arms = 3
    N = int(input("Enter Number of Iterations:")) # number of simulations = 100000
    reward = np.zeros(n, dtype = "float32")
    print("Enter Values of Rewards:") # 1.2, 5.5, 3.8
    for i in range(n):
        reward[i] = float(input())

    print(f"Input of Rewards: {reward}")
```

```
Enter Number of Arms: 3
Enter Number of Iterations:100000
Enter Values of Rewards:
1.2
1.5
1.8
Input of Rewards: [1.2 1.5 1.8]
```

```
# Define Action class
class Actions:
    def __init__(self, m):
        self.m = m # reward for arm
        self.mean = 0 # average reward for arm
        self.N = 0 #Number of Simulations

 # Choose a random action
    def choose(self):
        return np.random.randn() + self.m  #adding a random number to reward to add noise

 # Update the action-value estimate
    def update(self, x):
        self.N += 1
        self.mean = (1 - 1.0 / self.N)*self.mean + 1.0 / self.N * x

    def run_experiment(rewards, N):
        actions = []
        counts = []
        for i in range(len(rewards)):
            actions.append(Actions(rewards[i]))
            counts.append(Actions(rewards[i]).N)
        data = np.empty(N)

        n_arms = len(rewards)
        ucb = np.zeros(n_arms)

        def UCB1():
            tot_count = np.sum(counts)
            flag = True
            for i in range(n_arms):
                if counts[i] == 0:
                  flag = True
                else:
                  flag = False

            if(flag == False):
              for i in range(n_arms):
                bonus = np.sqrt((2 * np.log(tot_count)) / float(counts[i]))
                ucb[i] = actions[i].mean + bonus

              return np.argmax(ucb)


        for i in range(N):
            j = UCB1()
            x = actions[j].choose()
            actions[j].update(x)
            counts[j] += 1

            # for the plot
            data[i] = x
        cumulative_average = np.cumsum(data) / (np.arange(N) + 1)

        for a in actions:
            print(a.mean)
        return cumulative_average,counts

    def plotResult(cum_avg,rewards,N):
        # plot moving average ctr
        plt.plot(cum_avg)
        for i in range(len(rewards)):
            plt.plot(np.ones(N)*rewards[i])
            plt.xscale('log')
        plt.show()
```

Output Plot is as follows:



Comparing all Bandit algorithms side by side:

• We can very clearly see how much noisier UCB1's behavior looks than the epsilonGreedy or Softmax algorithm's.

• We see that the epsilon-Greedy algorithm doesn't converge as quickly as the Softmax algorithm. This might suggest that we need to use another annealing schedule or that this testbed is one in which the Softmax algorithm is simply superior to the epsilon-Greedy algorithm.

• We see that UCB1 takes a while to catch up with the annealing Softmax algorithm, but that it does start to catch up right near the end of the plays we've simulated. In the exercises we encourage all to try other enviroments in which UCB1 might outperform Softmax unambigiously.

• UCB1 finds the best arm very quickly, but the backpedaling it does causes it to underperform the Softmax algorithm along most metrics.

It's important to understand that A/B testing and Multi Armed Bandit(MAB) serve different use cases as their focus is different. An A/B test is done to collect data with its associated statistical confidence. A

business then uses the collected data, interprets it in a larger context and then makes a decision.

In contrast, MAB is an optimization algorithm that maximizes a given metric (which is conversions of a particular type in VWO's context). There's no intermediate stage of interpretation and analysis as the MAB algorithm is adjusting traffic automatically.

A/B testing is perfect for cases where:

a) The objective is to collect data in order to make a critical business decision. Example: if you're deciding the positioning of a product, engagement data on different positionings in an A/B test is an important data point (but not the only one)

b) The objective is to learn the impact of all variations with statistical confidence. Example: if you've put effort into developing a new product, you don't just want to optimize for sales but also gather information on its performance so that the next time you can incorporate learnings into developing a better product.

In contrast, MAB is perfect for cases where:

a) There is no need for interpretation for results/performance of variations and all you care about is maximizing conversions. Example: if you're testing color scheme, you just want to serve the one that maximizes conversions

b) The window of opportunity for optimization is short-lived and there's not enough time for gathering statistical significant results. Example: optimizing pricing for a time-limited offer.

Conclusion:

It is fair to state that both A/B and MAB have their strengths and shortcomings- the dynamic between the two is complementary and not competitive. A few common real-world scenarios, where MAB has shown that it's clearly superior to A/B Testing:

1. Opportunity cost of lost conversions is too high: Imagine you're selling diamonds (or a car) online. Each lost conversion is probably worth thousands of dollars in lost opportunity for you. In that case, MAB's focus on maximizing conversions is a perfect fit for your optimization needs.

2. Optimizing Click-Through Rates For News Outlets That Cover Time-Sensitive Events: Conjuring catchy headlines was initially an editor's job, but that is clearly passe. The short shelf life of news pieces means that quick optimization is essential. They optimize and test headlines, photo thumbnails, video thumbnails, recommended news articles, and popular articles to drive maximum clicks inside a short window.

3. Continuous Optimization: Optimizers have the ability to add or subtract multiple elements from variations and test across all simultaneously. In a traditional A/B test, there is little freedom to orchestrate changes once the experiment goes live because data sanctity is sacrosanct.

4. Optimizing Revenue with Low Traffic: If there's not enough traffic, A/B tests can take really long to produce statistical significance. In such cases, a business may find it better to run an MAB as it is able to detect the potentially best version much earlier and direct an increasing amount of traffic to it.

Even though MAB has its merits, there are many scenarios where A/B Testing is clearly the better choice:

1. When you are aiming for Statistical Significance: For all their strengths, MAB experiments are not the best choice when you want to get a statistically robust winner. A/B tests are still the fastest way to statistical significance even though you might lose some conversions in the process.

2. Optimizing for multiple metrics: Mature experimentation teams track 4+ goals per experiment, as experiences are composite of primary and secondary goals. While MAB experiments work great when optimizing for one key metric, they don't work well for multiple goals as they only factor in the Primary Goal while allocating incoming traffic.

3. Post experiment analysis: Most experimenters like to slice and dice the data gathered during an experiment to check how different segments reacted to modifications on their web properties. This analysis is possible in A/B Tests but might not be possible in MAB as sufficient data might not be available for underperforming variations.

4. Incorporating learnings from all variations (including the poor ones) into further business decisions: During the course of the test, MAB allocates most traffic to the best performing variation. This means that poor-performing variations do not get enough traffic to reach statistical confidence. So, while you may know with confidence, the conversion rate for best performing variation, similar confidence may not be available for poor-performing ones. If getting this knowledge is important for a business decision (perhaps you want to know how bad is the losing variation as compared to the best one), an A/B test is the way to go.