K-Armed Bandit Problem

1. Introduction

In this tutorial, we'll learn about *k*-armed bandits and their relation to reinforcement learning. We'll then inspect the exploration and exploitation terms and understand the need to balance them by investigating the exploration vs. exploitation tradeoff.

Finally, we'll investigate different strategies to get the maximum reward from a *k*-armed bandit setting and how we can compare different algorithms using regret.

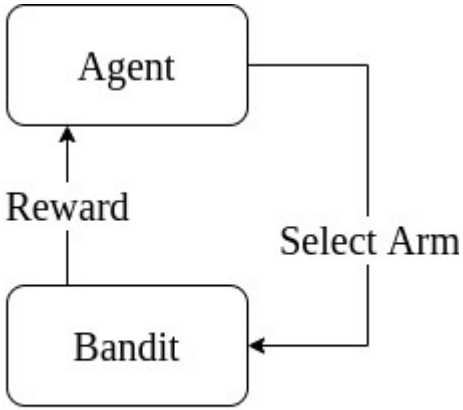
2. Reinforcement Learning and K-Armed Bandits

The one-armed bandit is a slang term for slot machines. Although the chances are slim, we know that spinning the one-armed bandit has a probability of winning. This means, given enough trials, we'll win at some point. We just don't know the probability distribution and can't discover it with a limited number of trials.

Now imagine we're in a casino, sitting in front of 10 different slot machines. Each one of the slot machines has a probability of winning. If we have some coins to play, which one do we choose? Do we spend all our chances on a single machine, or do we play on a combination of different machines?

Finding the optimal playing strategy to win the highest possible reward isn't a trivial task. We'll learn about strategies for solving the *k*-armed bandit in the following sections.

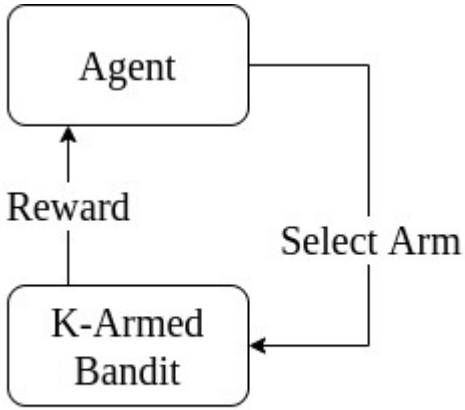
In reinforcement learning, an agent is interacting with the environment. It's trying to find out the outcomes of available actions. Its ultimate goal is to maximize the reward at the end.
The *k*-armed bandit problem is a simplified reinforcement learning setting. There is only one state; we (the agent) sit in front of k slot machines. There are actions: pulling one of the distinct arms. The reward values of the actions are immediately available after taking an action:



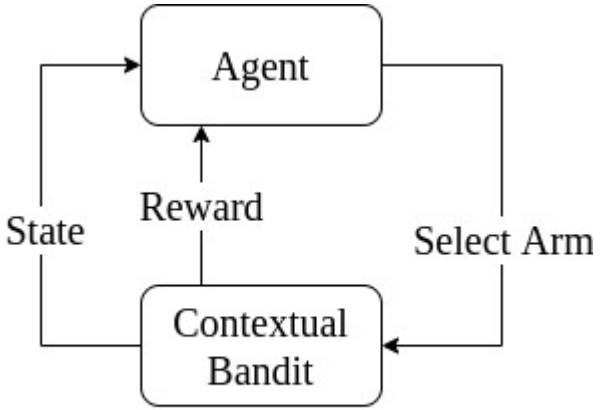
k-Armed Bandit

K-armed bandit is a simple and powerful representation. Yet, it can't model complex problems ignoring the consequences of previously taken actions.

To better model the real-life problems, we introduce the "context". The agent selects an action, given the context. In other words, the agent has a state, and taking actions makes the agent transition between states and results in a reward. Simply put, context is the result of previously taken actions:

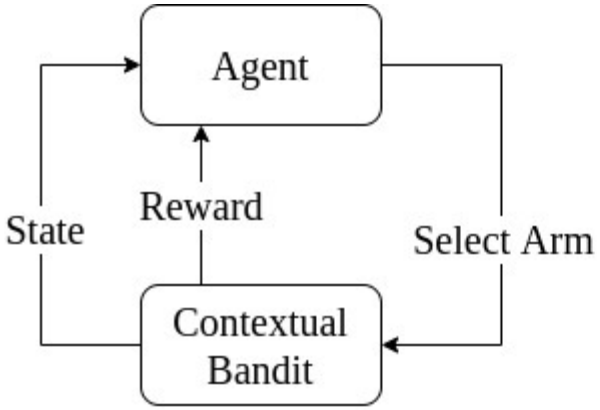


k-Armed Bandit

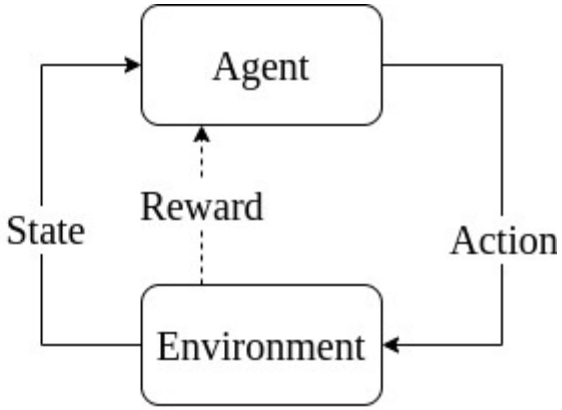


Contextual Bandit

Still, the reward is immediate after choosing an action in the contextual bandit model, unlike real-world problems. Sometimes we get a reward after some time or after taking a series of actions. For example, even in simple games like tic-tac-toe, we receive the reward after making a series of moves when the game ends. Hence, contextual bandits are too simplified to model some problems:



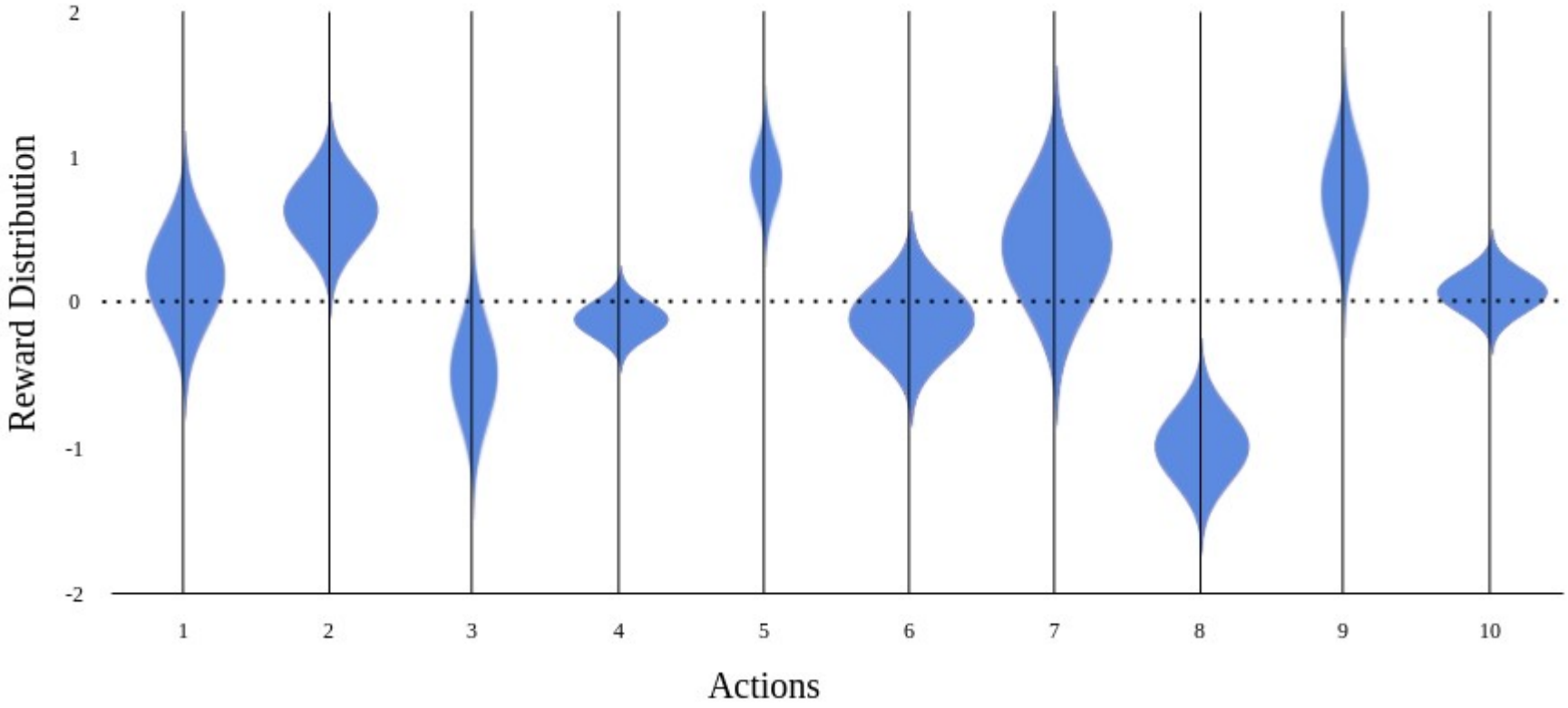
Contextual Bandit



Reinforcement Learning

3. Exploration vs. Exploitation

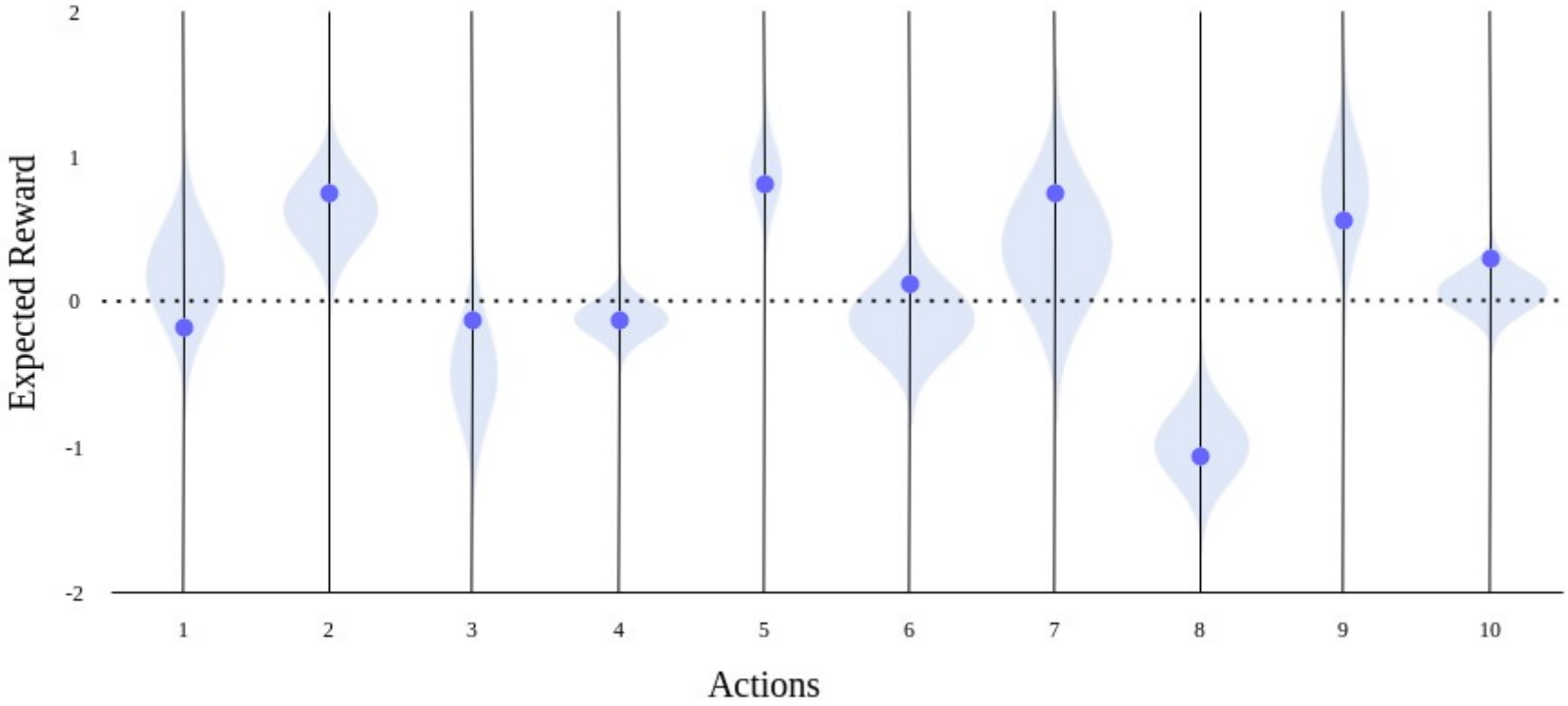
In a *k*-armed bandit setting, the agent initially has none or limited knowledge about the environment. As a result, it discovers the environment by trial-and-error.
Let's consider the *k*-armed bandit setting. Here, we know that each of the slot machines gives an immediate reward, based on a probability distribution. The reward will be a real number, negative or positive. The figure below illustrates an example setting with $k = 10$:



Although the reward distribution of each slot machine is determined in this setting, we don't know about them. So, we'll try to discover by trial-and-error. Choosing an arbitrary action when the outcome is unknown is called exploration.

Say we choose random actions in this unknown setting (explore). As a result, after some time, we have an initial idea about each slot machine's reward. With a small number of trials, we can't be confident about the expected values. Still, we know something about the environment at this point.

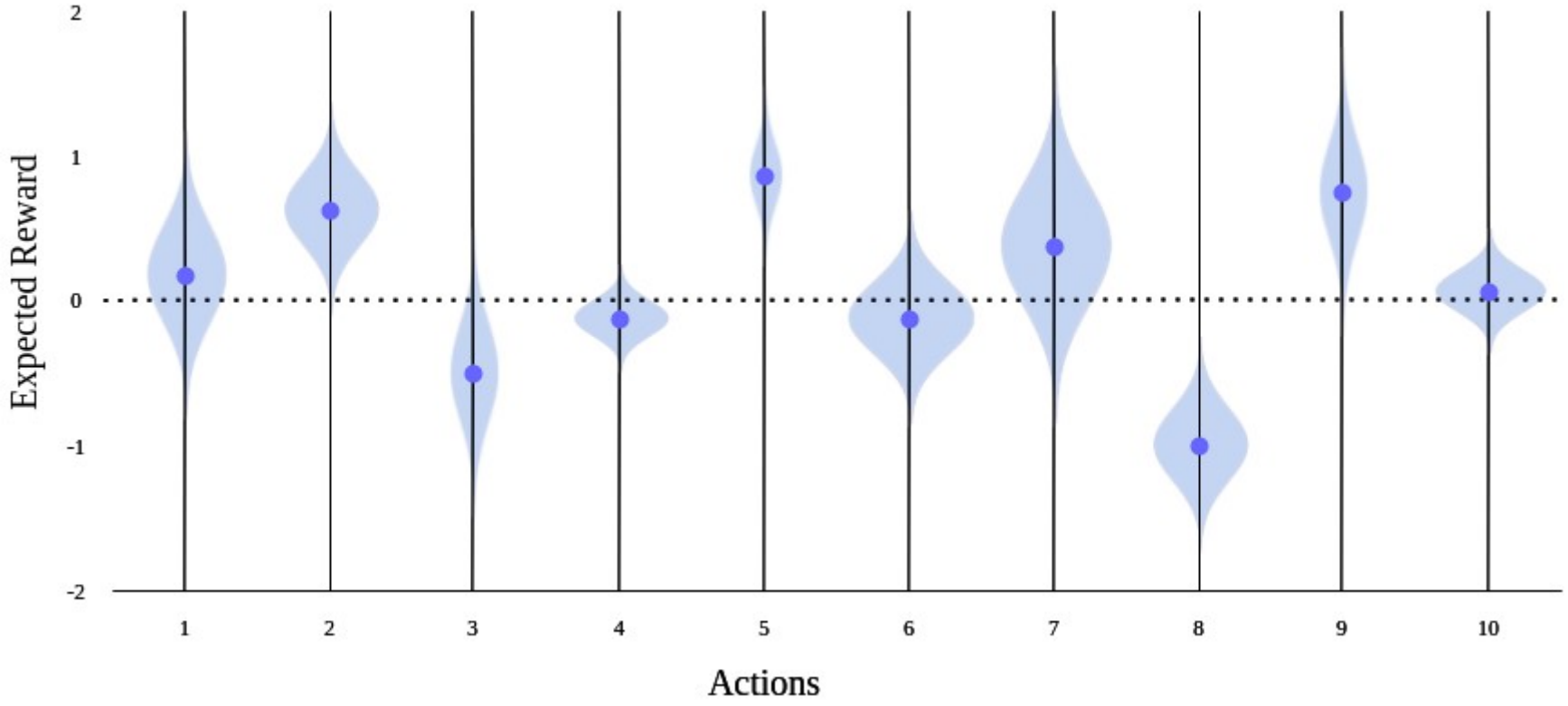
In our example setting, the blue dots mark what the agent believes the expected reward from a slot machine is. The light blue marks show the real distribution, which is still unknown to the agent at this point. For some machines, the agent's expectation is close to average, while for some, it's around the extremes after a limited exploration:



Taking actions according to the expectations and using prior knowledge to get immediate rewards is called exploitation. In our example, the agent would exploit by choosing the 5th machine, as it believes it'll result in a positive reward, and that would be a good choice.

However, the agent would try the 6th machine in a similar belief, which wouldn't meet its expectations.

Depending on the problem size, after enough trials, the agent has a better estimation of the slot machines' expected reward values. As a result of exploration, now the agent's expectations highly match the actual rewards:



In a realistic reinforcement learning problem with many states and actions, it takes an infinite number of trials to actually learn the environment and accurately predict the expected rewards. Even at times, the environment might be changing (non-stationary). So, sticking to what we've already learned isn't a good idea. It's risky and even dangerous in some cases, as the agent can get stuck in a sub-optimal choice.

On the other hand, constantly exploring the environment means only taking random actions. Not exploiting a good reward opportunity is simply against the overall goal of getting the maximum overall reward.

In such a setting containing incomplete information, we need to have a balance. The best long-term strategy may involve taking short-term risks. Exploration can lead to higher reward in the long run while sacrificing short term rewards.

Conversely, exploiting ensures an immediate reward in the short-term, with uncertainty in the long run. The dilemma of choosing what the agent already knows vs. gaining additional knowledge is called the exploration vs. exploitation tradeoff.

The *k*-armed bandit environment contains uncertainties. The exploration vs. exploitation trade-off arises due to incomplete information. If we were to know everything about the environment, then we could find an optimal solution. However, real-world problems involve many unknowns.

4. K-Armed Bandit Action Selection Strategies

One way to evaluate *k*-armed bandit strategies is to measure the expected regret. As the name suggests, lower regret values are better.

We measure the expected regret at each action. When an action is taken, we have an approximation for the expected reward of the best possible action. Besides, we instantly get the reward for the taken action. The difference between the expected best reward and the actual reward gives the expected regret. It denotes what we've lost by exploring instead of taking the best action.

We can compare different strategies on a given problem by cumulatively adding the regret values and comparing the results.

Now let's explore some strategies for approximating a good solution for the *k*-armed bandit problem.

4.1. Explore-First

The explore-first (or epsilon-first, ε-first) strategy consists of two phases: exploration and exploitation.

As the name suggests, in the exploration phase, the agent tries all options several times. Then, it selects the best action based on those trials. Finally, in the exploitation phase, it just takes the action with the best outcome.

The exploitation phase of this algorithm works perfectly in terms of minimizing regret. However, the inefficiency lies in the exploration phase. The overall performance isn't satisfactory for most applications.

4.2. Epsilon-Greedy

In the epsilon-greedy (ε-greedy) approach, instead of having a pure exploration phase, we spread the exploration over time.

Initially, we select a small epsilon value (usually, we choose $\varepsilon \approx 0.1$). Then, with a probability of epsilon, even if we're confident with the expected outcome, we choose a random action. On the remaining times (1 – epsilon), we simply act greedy and exploit the best action. We apply this simple algorithm for the whole period.

The epsilon-greedy algorithm is easy to understand and implement. It's one of the basic reinforcement learning enhancements with Q-learning.

4.3. Decayed Epsilon-Greedy (Epsilon-Decreasing Strategy)

A downside of the epsilon-greedy strategy is that the exploration phase is uniformly distributed over time. We can improve on this by exploring more in the beginning, when we have limited knowledge about the environment. Then, as we learn, we can make more educated decisions and explore less.

The idea is pretty simple to implement. We introduce a decay factor (usually around 0.95). After each action, we update epsilon to be epsilon*decay. So, the epsilon value gradually decreases over time. This results in lowering the regret compared to the epsilon-greedy algorithm.

4.4. Softmax (Boltzmann)

The softmax approach enables us to select an action with a probability depending on its expected value. Remember that the epsilon-greedy approach selects a uniform random action while exploring. So, the second-best option and the worst option have the same probability of being selected.

In softmax exploration, the agent still selects the best action most of the time. However, other actions are ranked and selected accordingly. As a result, it'll behave better than the epsilon-greedy. Still, the regret is not bounded.

4.5. Pursuit Methods

Pursuit methods maintain both action-value estimates and action preferences for the current state. Their preference continually "pursuit" the best (greedy) action according to the current estimates.

The action preference probabilities are updated before action selection. The probability of choosing the best actions is increased, and others' probabilities are decreased.

4.6. Upper Confidence Bounds

Upper confidence bounds (UCB) is a family of algorithms trying to balance exploration and exploitation.

At first, we ensure that each action is taken at least once for every state. Then, when we face a selection again, we select the action maximizing the upper confidence bound.

UCB improves quickly and converges more rapidly than the other methods. The algorithm selects the optimal action as the number of trials goes to infinity.

Its overall regret grows logarithmic, and it's bounded. Hence, theoretically, it'll be the best performing strategy in the long run.

4.7. Thompson Sampling (Bayesian Bandits)

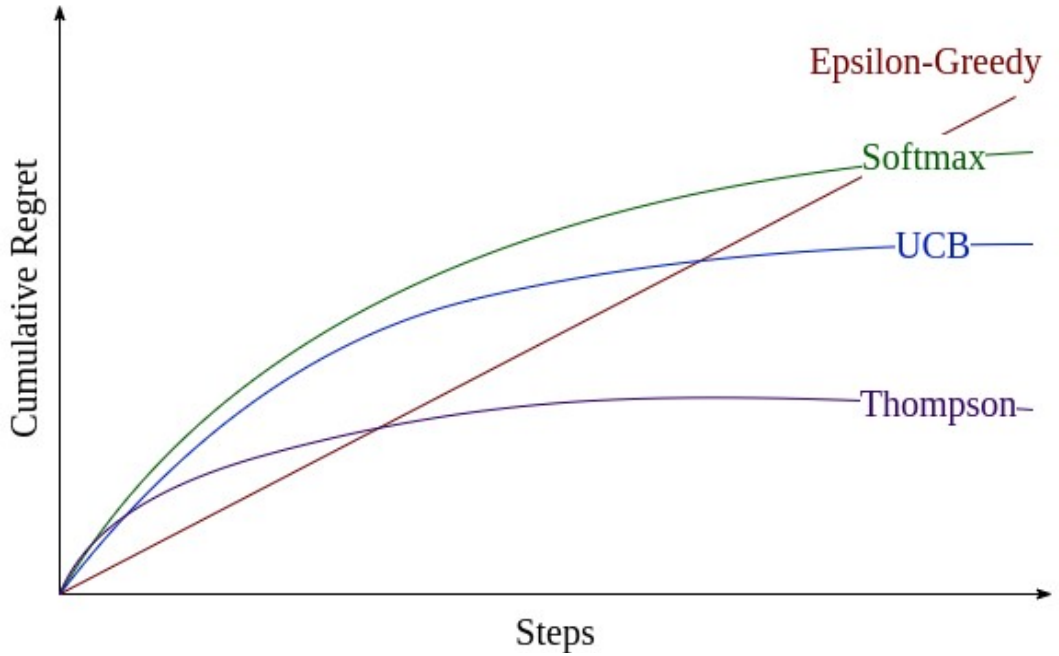
The algorithms we've discussed so far update the expected reward per action using the mean of previous trials. Thompson sampling algorithm adopts a different approach.

It assumes that each action offers a reward based on a beta probability distribution and tries the estimate the distribution within a confidence interval. The priority distribution of the

action is updated after taking an action. Then the algorithm chooses the action that maximizes the expected reward.
In this sense, the Thompson sampling algorithm adopts a Bayesian Inference approach, where the setting is updated as more evidence becomes available.

5. Regret Comparison of Action Selection Strategies

The figure below compares the worst-case bounds for some of the algorithms:



In most settings, epsilon-greedy is hard to beat. UCB and Thompson Sampling methods use exploration more effectively. Thompson sampling will be more stable, given enough trials. But, UCB is more tolerant of noisy data and converges more rapidly.
Depending on the problem at hand, a strategy can behave much better than the theoretical worst cases. Plotting the actual regret graphs of different algorithms might give different insights.

UpperConfidenceBound and ThompsonSampling

Reinforcement learning has been one of the most researched machine learning algorithms. As an algorithm that can be closely related to how human beings and animals learn to interact with the environment, reinforcement learning has always been given huge importance especially when it comes to branches of Artificial Intelligence such as robotics.

Reinforcement Learning is a reward based Machine Learning algorithm in which an agent learns to interact with its environment by performing an action that is either rewarded or penalized. The primary focus or objective of reinforcement learning is to help the agent maximize the cumulative reward with every action it performs.

Here, we will learn about a Reinforcement Learning algorithm called *Thompson Sampling*, the basic intuition behind it and to implement it using Python. Thompson Sampling makes use of Probability Distribution and Bayes Theorem to generate success rate distributions.

What Is Thompson Sampling?

Thompson Sampling is an algorithm that follows exploration and exploitation to maximize the cumulative rewards obtained by performing an action. Thompson Sampling is also sometimes referred to as Posterior Sampling or Probability Matching.

An action is performed multiple times which is called exploration and based on the results obtained from the actions, either rewards or penalties, further actions are performed with the goal to maximize the reward which is called exploitation. In other words, new choices are explored to maximize rewards while exploiting the already explored choices.

One of the first and the best examples to explain the Thompson Sampling method was the Multi-Armed Bandit problem, about which we will learn in detail, later in this article.

Applications of Thompson Sampling:

Thompson Sampling algorithm has been around for a long time. It was first proposed in 1933 by William R. Thompson. Though the algorithm has been ignored and has had the least attention for many years after its proposal, it has recently gained traction in many areas of Artificial Intelligence. Some of the areas where it has been used are revenue management,

marketing, web site optimisation, A/B testing, advertisement, recommendation system, gaming and more.

The Multi-Armed Bandit Problem

Though the name may sound creepy, Multi-Armed Bandit Problem (MABP), actually refers to multiple Slot machines. For those who are still confused, given below is the image of a single arm/lever slot machine.

Be glad if you haven't ever seen one because as the name of the problem suggests these are bandits and their goal is to rob us. These machines are often popular in casinos where gamblers throw in a lot of money in the hope of winning big, most of them don't.
Now let's go back to the MABP. So the idea is all simple. Given multiple slot machines, like the one shown above, the goal is to maximize the reward for a person pulling the lever of each machine randomly.

If you were given five slot machines and a limited number of turns to pull the lever, how would you determine maximum success? Thompson Sampling is an approach that successfully tackles the problem.

Thompson Sampling makes use of Probability Distribution and Bayes Rule to predict the success rates of each Slot machine.

Basic Intuition Behind Thompson Sampling

1. To begin with, all machines are assumed to have a uniform distribution of the probability of success, in this case getting a reward
2. For each observation obtained from a Slot machine, based on the reward a new distribution is generated with probabilities of success for each slot machine
3. Further observations are made based on these prior probabilities obtained on each round or observation which then updates the success distributions
4. After sufficient observations, each slot machine will have a success distribution associated with it which can help the player in choosing the machines wisely to get the maximum rewards

Implementing Thompson Sampling With Python:

Let us consider 5 bandits or slot machines. Initially, we do not have any data on how the probability of success is distributed among the machines. So we start by exploring the machines one by one.

Say after 200 observations we have data similar to what's shown below:

Index	B1	B2	B3	B4	B5
0	1	1	1	1	1
1	0	1	1	0	0
2	0	0	1	1	1
3	1	0	0	0	1
4	1	0	1	1	0
5	0	0	1	0	1
6	1	1	1	1	1
7	1	0	1	1	1
8	1	1	1	0	0
9	1	1	0	0	0
10	0	0	0	1	0
11	1	1	0	0	1
12	1	1	1	1	1
13	0	1	0	0	0

Each column represents a bandit or a slot machine (B1, B2, B3, B4 and B5). The 0's represent penalties or the player not getting a reward and all the 1's represent the player winning a reward while pulling the arm of the slot machine.

We have 200 observations, which means that the player pulled the lever/arm of each machine 200 times.
Run the below code block to generate a similar dataset.

```
import random
import pandas as pd
import matplotlib.pyplot as plt

data = {}
data['B1'] = [random.randint(0,1) for x in range(200)]
data['B2'] = [random.randint(0,1) for x in range(200)]
data['B3'] = [random.randint(0,1) for x in range(200)]
data['B4'] = [random.randint(0,1) for x in range(200)]
data['B5'] = [random.randint(0,1) for x in range(200)]
data = pd.DataFrame(data)
print(data)
```

Output:

```
     B1  B2  B3  B4  B5
0     1   0   0   1   1
1     0   1   1   1   1
2     0   0   0   0   1
3     1   0   1   1   1
4     0   0   1   1   1
..   ..  ..  ..  ..  ..
195   0   0   0   0   0
196   0   1   1   0   0
197   1   1   0   1   0
198   0   0   0   0   0
199   1   1   0   1   1

[200 rows x 5 columns]
```

Let's Implement Thompson Sampling:

We will use one of the simplest implementations of Thompson sampling in Python.
Since we have to iterate through each observation of each of the 5 machines, we will start by initializing the number of observations and machines.

```
observations = 200
machines = 5
```

In Thompson Sampling each machine is selected based on the beta distribution of rewards and penalties associated with each machine. And so we need a list to save the machine that has been selected by Thompson Sampling for a specific round/observation.

```
machine_selected = []
```

We will now initialize 3 variables:-

- one to store the rewards or 1's received by each Slot Machine/Bandit that was selected by the Thompson Sampling algorithm,
- one to store the penalties or 0's received by each Slot Machine/Bandit that was selected by the Thompson Sampling algorithm, and,
- a variable to store the total number of rewards obtained using the Thompson Sampling algorithm

```
rewards = [0] * machines
penalties = [0] * machines
total_reward = 0
```

Note:
Each of the above variables is initialized with zero as we do not know prior to the algorithm. Also, the first two variables are lists as it stores the values of each machine. In this case 5 machines.

Now let's begin Thompson Sampling:-

For each observation 'n' in the dataset, we start with the machine B1 and a default maximum beta distribution of zero.

```
for n in range(0, observations):
        bandit = 0
        beta_max = 0
```

Now for each observation, we will iterate through each machine and will select the machine with the highest random beta distribution.

```
        for i in range(0, machines):
                beta_d = random.betavariate(rewards[i] + 1, penalties[i] + 1)
                        if beta_d > beta_max:
                                beta_max = beta_d
                                bandit = i
```

Now update the list 'machine_selected' with the bandit/machine selected by the Thompson Sampling:-

```
machine_selected.append(bandit)
```

Once we have selected the machine, we will check the dataset for that specific machine and the number of observation, if it returned a reward or penalty. If it is a reward, we will update the list 'rewards' by adding one to that specific machine/bandit, if it is a penalty we will update the list 'penalties' by adding one to that specific machine/bandit. We will then update the total reward.

```
reward = data.values[n, bandit]
if reward == 1:
        rewards[bandit] = rewards[bandit] + 1
else:
        penalties[bandit] = penalties[bandit] + 1
total_reward = total_reward + reward
```

Let's visualize our results:-

```
print("\n\nRewards By Machine = ", rewards)
print("\nTotal Rewards = ", total_reward)
print("\nMachine Selected At Each Round By Thompson Sampling : \n", machine_selected)
```

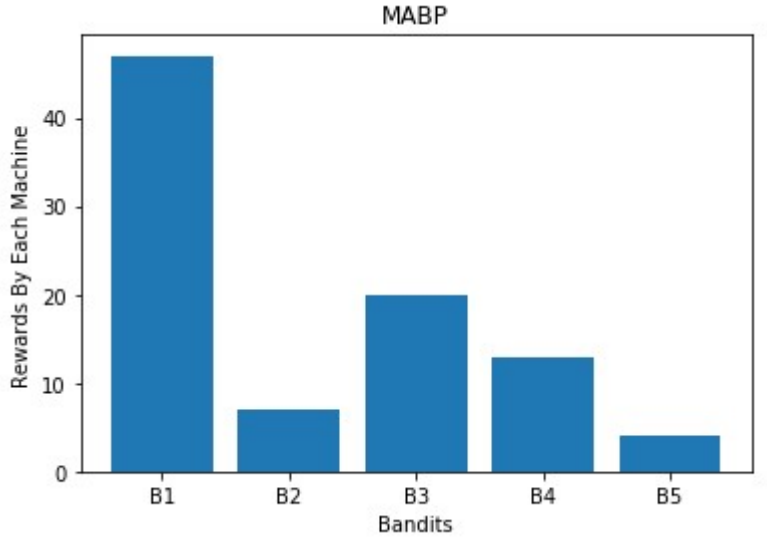
Output:

```
Rewards By Machine =  [47, 7, 20, 13, 4]

Total Rewards Obtained With Thompson Sampling =  91

Machine Selected At Each Round By Thompson Sampling :
 [2, 1, 3, 3, 0, 4, 2, 1, 0, 0, 3, 3, 3, 4, 4, 3, 4, 0, 3, 3, 0, 3, 3, 3, 1, 3, 0, 3, 0, 0, 0, 0, 2, 3,
4, 4, 2, 3, 0, 2, 0, 2, 0, 3, 0, 0, 4, 0, 0, 0, 2, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 3, 0, 4, 0, 4,
3, 0, 3, 0, 0, 4, 0, 4, 0, 0, 0, 0, 3, 0, 0, 2, 0, 0, 3, 0, 0, 0, 0, 4, 2, 0, 1, 3, 0, 0, 2, 2, 0, 1,
0, 0, 1, 0, 3, 0, 3, 0, 0, 2, 3, 2, 2, 1, 3, 2, 0, 0, 4, 0, 1, 1, 2, 1, 0, 0, 0, 0, 0, 0, 0, 3, 2, 0,
0, 1, 2, 0, 2, 0, 3, 4, 0, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 2, 1, 3, 0, 0, 1, 3, 1, 2, 2, 1, 0, 1, 2,
0, 0, 1, 1, 0, 1, 0, 0, 1, 2, 0, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 0, 1, 0, 3, 3]
```

Rewards By Each Machine:-

```
#Visualizing the rewards of each machine
plt.bar(['B1','B2','B3','B4','B5'],rewards)
plt.title('MABP')
plt.xlabel('Bandits')
plt.ylabel('Rewards By Each Machine')
plt.show()
```

Output:



Number Of Times Each Machine Was Selected

```
#Number Of Times Each Machine Was Selected
from collections import Counter
print("\n\nNumber Of Times Each Machine Was Selected By The Thompson Sampling Algorithm :
\n",dict(Counter(machine_selected)))
```

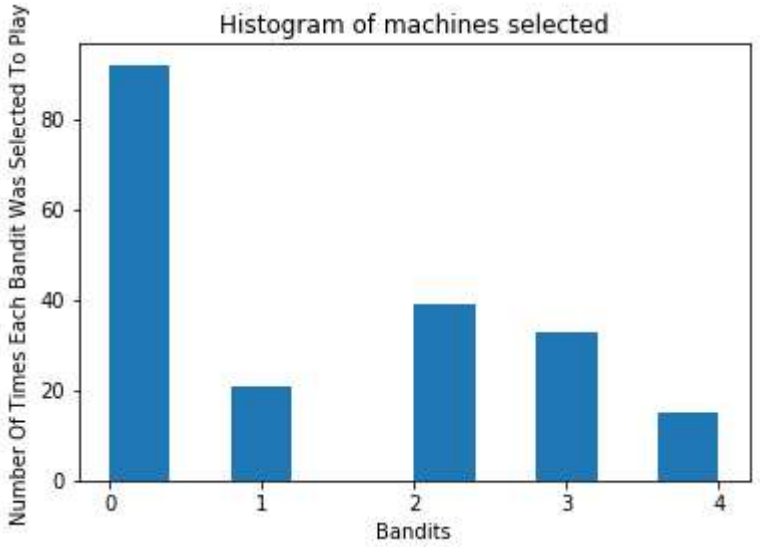
Output:

Number Of Times Each Machine Was Selected By The Thompson Sampling Algorithm :
{2: 39, 1: 21, 3: 33, 0: 92, 4: 15}

#Visualizing the Number Of Times Each Machine Was Selected:-

```
plt.hist(machine_selected)
plt.title('Histogram of machines selected')
plt.xlabel('Bandits')
plt.xticks(range(0, 5))
plt.ylabel('No. Of Times Each Bandit Was Selected')
plt.show()
```

Output:



By comparing the two graphs, we can see that Thomson Sampling wisely selected the machines which gave more rewards the most number of times.

Bibliography:

1. https://www.baeldung.com/cs/k-armed-bandit-problem
2. https://data102.org/sp20/assets/notes/notes21.pdf
3. https://towardsdatascience.com/the-upper-confidence-bound-ucb-bandit-algorithm-c05c2bf4c13f
4. https://analyticsindiamag.com/reinforcement-learning-the-concept-behind-ucb-explained-with-code/
5. https://analyticsindiamag.com/thompson-sampling-explained-with-python-code/
6. https://www.mltut.com/upper-confidence-bound-reinforcement-learning-super-easy-guide/
7. aionlinecourse.com/tutorial/machine-learning/thompson-sampling-intuition#:~:text=There%20is%20a%20significant%20difference,round%20without%20adjusting%20the%20value.
8. https://towardsdatascience.com/a-comparison-of-bandit-algorithms-24b4adfcabb
9. https://courses.cs.washington.edu/courses/cse312/22wi/files/student_drive/9.8.pdf