Multi-Armed Bandits: UCB Algorithm

Optimizing actions based on confidence bounds

Christian Hubbs Jan 7 · 7 min read ★

Photo by Jonathan Klok on Unsplash

Imagine you're at a casino and are choosing between a number (k) of one-armed bandits (a.k.a. slot machines) with different probabilities of rewards and want to

choose the one that's best. What strategy do you use to help you walk out with the largest number of quarters?

This is the essence of a multi-armed bandit problem, which is a simplified reinforcement learning task. These can occur when you're seeking to maximize engagement on your website, or figuring out clinical trials, or trying to optimize your computer's performance.

More than just being a useful framework, they're fun and interesting to solve!

TL;DR

We teach the Upper Confidence Bound bandit algorithm with examples in Python to get you up to speed and comfortable with this approach.

Your First Strategy

Bandit problems require a balance between the **exploration and exploitation trade-off.** Because the problem starts with no prior knowledge of the rewards, it needs to explore (try a lot of slot machines) and then exploit (repeatedly pull the best lever) once it has narrowed down its selections.

The simplest bandits employ an ϵ-greedy strategy, which means it will choose the slot machine it thinks is best most of the time, but there's a small ϵ% probability that it will choose a random slot machine instead. It's a simple strategy and can yield good results (see this post if you want to see this approach implemented), but we can do better.

Upper Confidence Bound Bandit

ϵ-greedy can take a long time to settle in on the right one-armed bandit to play because it's based on a small probability of exploration. The **Upper Confidence Bound** (UCB) method goes about it differently because we instead make our selections based on how uncertain we are about a given selection.

I'll show you the math, then explain:

The above equation gives us the selection criterion for our model. Q_n(a) is our current estimate of the value of a given slot machine *a*. The value under the square root is the log of the total number of slot machines we've tried, *n*, divided by the number of times we've tried each slot machine *a* (*k_n*), while *c* is just a constant. We choose our next slot machine by selecting whichever bandit gives the largest value at each step *n*.

The square root term is an estimate of the variance of each action *a*. If we haven't chosen a bandit yet, the variance is infinite (we'd be dividing by 0), meaning that's going to be our next choice. This forces the algorithm to explore those unknown values quickly. Once it chooses it, all else being equal, its value drops and becomes less likely relative to the other selections. Because log(n) is in the numerator, each time we don't choose an action, it's value becomes more likely, although the numerator grows more slowly (i.e. logarithmically) so the more data you collect, the smaller this effect becomes. The end result is an algorithm that samples very quickly to reduce the uncertainty of the unknowns before locking onto the most profitable machine.

Some Practical Caveats

Before we dive into implementation, we're going to initialize our values, k_n(a) = 1 instead of 0 to ensure we don't get any `nan` values popping up in Python. Also, we're going to be simulating a LOT of bandit pulls, so we need to be somewhat efficient with calculating some of our values. Q_n(a) is just the mean value of that action, and rather than keeping thousands of values for each action, we can use a handy formula to calculate the mean by only storing two pieces of information for each action, how many selections we've made for that given action (k_n(a)) and our current mean (m_n).

Where R_n is the reward we just got from taking an action (I dropped that a's for brevity).

UCB Code

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

```
class ucb_bandit:
    '''
    Upper Confidence Bound Bandit

    Inputs
    =============================================
    k: number of arms (int)
    c:
    iters: number of steps (int)
    mu: set the average rewards for each of the k-arms.
        Set to "random" for the rewards to be selected from
        a normal distribution with mean = 0.
        Set to "sequence" for the means to be ordered from
        0 to k-1.
        Pass a list or array of length = k for user-defined
        values.
    '''
    def __init__(self, k, c, iters, mu='random'):
        # Number of arms
        self.k = k
        # Exploration parameter
        self.c = c
        # Number of iterations
        self.iters = iters
        # Step count
        self.n = 1
        # Step count for each arm
        self.k_n = np.ones(k)
        # Total mean reward
        self.mean_reward = 0
        self.reward = np.zeros(iters)
        # Mean reward for each arm
        self.k_reward = np.zeros(k)

        if type(mu) == list or type(mu).__module__ == np.__name__:
            # User-defined averages
            self.mu = np.array(mu)
        elif mu == 'random':
            # Draw means from probability distribution
            self.mu = np.random.normal(0, 1, k)
        elif mu == 'sequence':
            # Increase the mean for each arm by one
            self.mu = np.linspace(0, k-1, k)

    def pull(self):
        # Select action according to UCB Criteria
        a = np.argmax(self.k_reward + self.c * np.sqrt(
                (np.log(self.n)) / self.k_n))

        reward = np.random.normal(self.mu[a], 1)

        # Update counts
        self.n += 1
        self.k_n[a] += 1

        # Update total
        self.mean_reward = self.mean_reward + (
            reward - self.mean_reward) / self.n
```

```
            # Update results for a_k
            self.k_reward[a] = self.k_reward[a] + (
                reward - self.k_reward[a]) / self.k_n[a]

        def run(self):
            for i in range(self.iters):
                self.pull()
                self.reward[i] = self.mean_reward

        def reset(self, mu=None):
            # Resets results while keeping settings
            self.n = 1
            self.k_n = np.ones(self.k)
            self.mean_reward = 0
            self.reward = np.zeros(iters)
            self.k_reward = np.zeros(self.k)
            if mu == 'random':
                self.mu = np.random.normal(0, 1, self.k)
```

The above code defines our `ucb_bandit` class and enables us to simulate this problem. Running it requires three arguments: the number of arms to pull (`k`), the exploration parameter (`c`), and the number of iterations (`iters`).

We have the option of defining the rewards by setting `mu` (by default the reward means are drawn from a normal distribution). We run this for 1,000 episodes and average the rewards across each episode of 1,000 steps to get an idea for how well the algorithm performs.

```
k = 10 # number of arms
iters = 1000

ucb_rewards = np.zeros(iters)
# Initialize bandits
ucb = ucb_bandit(k, 2, iters)

episodes = 1000
# Run experiments
for i in range(episodes):
    ucb.reset('random')
    # Run experiments
    ucb.run()

    # Update long-term averages
    ucb_rewards = ucb_rewards + (
        ucb.reward - ucb_rewards) / (i + 1)

plt.figure(figsize=(12,8))
plt.plot(ucb_rewards, label="UCB")
```

```
plt.legend(bbox_to_anchor=(1.2, 0.5))
plt.xlabel("Iterations")
plt.ylabel("Average Reward")
plt.title("Average UCB Rewards after "
          + str(episodes) + " Episodes")
plt.show()
```


Looks good so far! The bandit learns as shown by the steadily increasing average rewards. To see how well it works, we'll compare it to a standard, ϵ-greedy approach (code can be found here).

```
k = 10
iters = 1000

eps_rewards = np.zeros(iters)
ucb_rewards = np.zeros(iters)

# Initialize bandits
ucb = ucb_bandit(k, 2, iters)
eps_greedy = eps_bandit(k, 0.1, iters, ucb.mu.copy())

episodes = 1000
# Run experiments
for i in range(episodes):
    ucb.reset()
    eps_greedy.reset()
```

```
    # Run experiments
        ucb.run()
        eps_greedy.run()

        # Update long-term averages
        ucb_rewards += (ucb.reward  - ucb_rewards) / (i + 1)
        eps_rewards += (eps_greedy.reward - eps_rewards) / (i + 1)

    plt.figure(figsize=(12,8))
    plt.plot(ucb_rewards, label='UCB Bandit')
    plt.plot(eps_rewards, label="$\epsilon={}$".format(
        eps_greedy.eps))
    plt.legend(bbox_to_anchor=(1.3, 0.5))
    plt.xlabel("Iterations")
    plt.ylabel("Average Reward")
    plt.title("Average rewards for UCB and $\epsilon$-Greedy Bandits")
    plt.show()
```


The UCB approach quickly finds the optimal action and keeps exploiting it for most of the episode, while the greedy algorithm has too much randomness despite also finding the optimal relatively quickly. We can look at this too by looking at the action selection and comparing the optimal actions chosen by each algorithm.

```
    width = 0.45
    bins = np.linspace(0, k-1, k) - width/2

    plt.figure(figsize=(12,8))
    plt.bar(bins, eps_greedy.k_n,
            width=width,
```

```
        label="$\epsilon$={}".format(eps_greedy.eps))
plt.bar(bins+0.45, ucb.k_n,
        width=width,
        label="UCB")
plt.legend(bbox_to_anchor=(1.3, 0.5))
plt.title("Number of Actions Selected by Each Algorithm")
plt.xlabel("Action")
plt.ylabel("Number of Actions Taken")
plt.show()

opt_per = np.array([eps_greedy.k_n, ucb.k_n]) / iters * 100
df = pd.DataFrame(np.vstack(
    [opt_per.round(1),
    eps_greedy.mu.reshape(-1, 1).T.round(2)]),
        index=["Greedy", "UCB", "Expected Reward"],
        columns=["a = " + str(x) for x in range(0, k)])
print("Percentage of actions selected:")
df
```

In this case, our optimal action was a=6 which had an average reward of 1.54. The ϵ-greedy algorithm selected it 83.4% of the time while the UCB algo selected it 89.7% of

the time. Additionally, you'll see that the greedy algorithm chose the negative values much more often than the UCB, again due to the fact that each selection came with a 10% chance of choosing a random action.

There are dozens of other multi-armed bandit approaches out there to learn about. It's a simple optimization framework, but can be quite powerful and interesting to play with!