Introduction to Multi-Armed Bandits with Applications in Digital Advertising

October 23, 2018

Multi-armed bandits (MABs) are powerful algorithms to solve optimization problems that have a wide variety of applications in website optimization, clinical trials and digital advertising.

In this blog post, we'll:

1. Explain the concept behind MABs
2. Present a use case of MABs in digital advertising
3. Implement two different strategies for solving MABs in Python
4. Observe how each strategy behaves as it observes data
5. Measure the performance of each strategy

What are MABs?

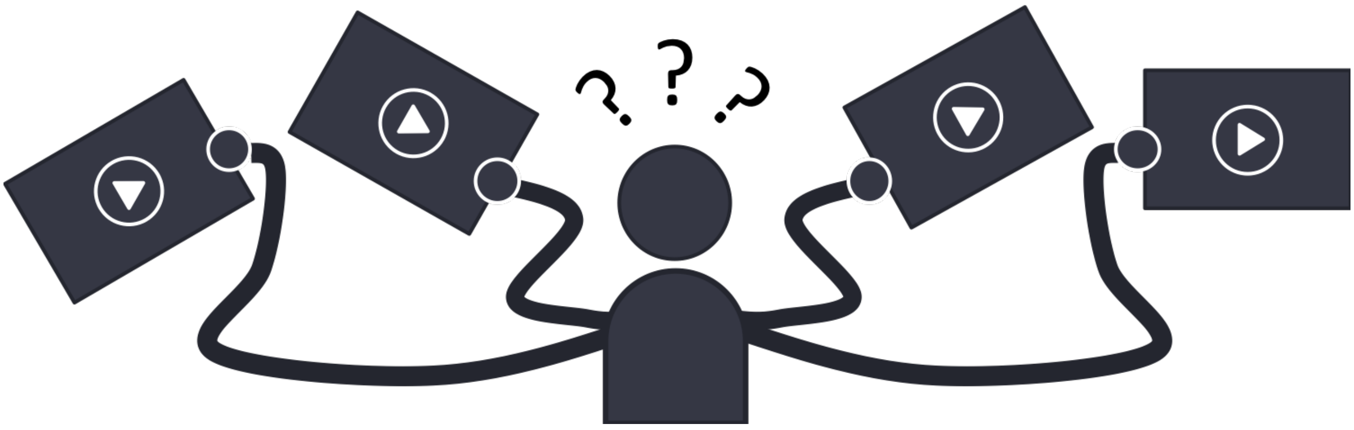
At their core, MABs are a class of algorithms that attempt to find the best option among a collection of alternatives by learning through trial and error. The name derives from "one-armed bandit," a slang term for a slot machine — which is a perfect analogy for how these algorithms work.

Imagine you are facing a wall of slot machines, and each one pays out at a different rate. A natural way to figure out how to make the most money (rewards) would be to try each at random for a while (exploration), and start playing the higher paying ones once you've gained some experience (exploitation). That's really all there is to it! Bandits have a strategy for balancing this exploration vs. exploitation trade-off in a way that attempts to maximize total rewards.

MABs in digital advertising

To illustrate the power of MABs in digital advertising, we'll work with a toy example. Suppose you are an advertiser seeking to optimize which ads to show visitors on a particular website. For each visitor, you can choose one out of a collection of ads, and your goal is to maximize the number of clicks over time.

It's reasonable to assume that each of these ads will resonate differently with your audiences, and some will be more engaging than others. The way we'll think about this in our example is that each ad has some theoretical — but unknown — click-through-rate (CTR) that is assumed to not change over time. How do we go about solving which ad we should choose?



One possible approach might be to A/B test. Perhaps we could randomly split the traffic and assign each ad in our collection to a different group. After a period of observation and analyzing the CTR of each group for statistical significance, we could then choose the ad with the highest CTR for all future visitors. This approach, however, has several weaknesses:

1. It requires an "analyst in the loop" to monitor results and manually make decisions, which is likely not scalable.
2. By splitting the traffic evenly for a period of time, we're risking choosing a poor performing ad longer than we have to and therefore losing out of potential clicks.
3. Since we're transitioning from 100% exploration to 100% exploitation with A/B testing, if for some reason the best ad changes some point in the future we'll miss out on it unless we periodically re-run the test.

MABs can overcome these challenges as we'll see.

Epsilon-greedy bandits

Although there is a huge variety of strategies used to implement MABs, we're only going to introduce two here: epsilon-greedy and Thompson sampling. Epsilon-greedy is by far the most popular and easiest strategy to implement, so we'll start there. Here's how it works at a high level:

1. Begin by randomly choosing an ad for each visitor and observing whether it receives a click.
2. As each ad is chosen and the click observed, keep track of the CTR (empirical CTR).
3. Start assigning "most" visitors to your top performing ad by empirical CTR at that point in time, otherwise randomly choose among your other ads.
4. As the empirical CTR changes with more trials, continue updating which ad you assign to most visitors.

The key parameter we decide upon for an epsilon-greedy bandit is — you guessed it — ε! ε manages the exploration/exploitation trade-off by determining what proportion of visitors we assign to exploration, *ε*, and exploitation, (1−*ε*). More formally:

$Pt(a)=\{1-\varepsilon K-1\varepsilon a=at{*}a\equiv=at{*}$

Where *Pt*(*a*) is the probability, we'll assign to choosing ad *a* in the *t*th trial, *K* is the number of ads in our collection, and *at*∗ represents the ad with the best CTR as of that trial.

We can implement this idea relatively easily with around 50 lines of Python code.

```
from __future__ import
division
```

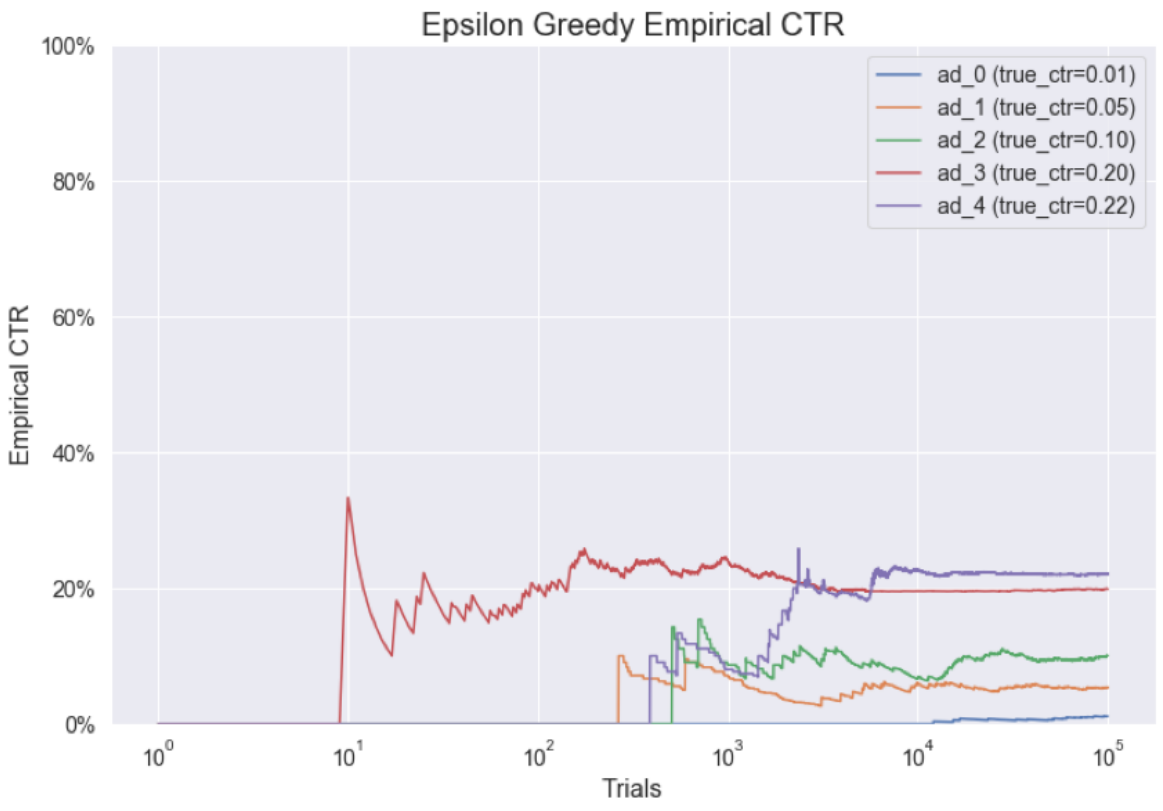
```
from __future__ import division
import numpy as np
class EpsilonGreedyBandit(object):


  def __init__(self, true_ctr, epsilon=0.1):
     self.epsilon = epsilon
     self.num_ads = len(true_ctr)
     self.ad_ids = np.arange(self.num_ads)
     self.weights = np.full(self.num_ads, 1/self.num_ads)
     self.clicks = np.zeros(self.num_ads)
     self.trials = np.zeros(self.num_ads)
     self.true_ctr = true_ctr
     self.empirical_ctr = np.zeros(self.num_ads)


  def _choose_ad(self):
     """
     Chooses which ad to play based on current weights.
     """
     ad = np.random.choice(self.ad_ids, size=1, p=self.weights)[0]
     return ad

  def _click(self, ad):
     """
     Observes whether the ad was clicked on by drawing from
     underlying probability distribution.
     """
     click = np.random.binomial(size=1, n=1, p=self.true_ctr[ad])[0]
     return click

  def _update(self, ad, click):
     """
     Updates clicks, trials, empirical ctr and weights for each ad.
     """
     self.clicks[ad] += click
     self.trials[ad] += 1
     self.empirical_ctr[ad] = self.clicks[ad] / self.trials[ad]
     # update weights if observed at least one click
     if self.clicks.any():
         self.weights[:] = self.epsilon / (self.num_ads - 1)
         best_ad = np.argmax(b.empirical_ctr)
         self.weights[best_ad] = 1 - self.epsilon

  def run_trial(self):
     """
     Runs a complete trial of choosing an ad, observing the click,
     and updating.
     """
     ad = self._choose_ad()
     click = self._click(ad)
     self._update(ad, click)

```

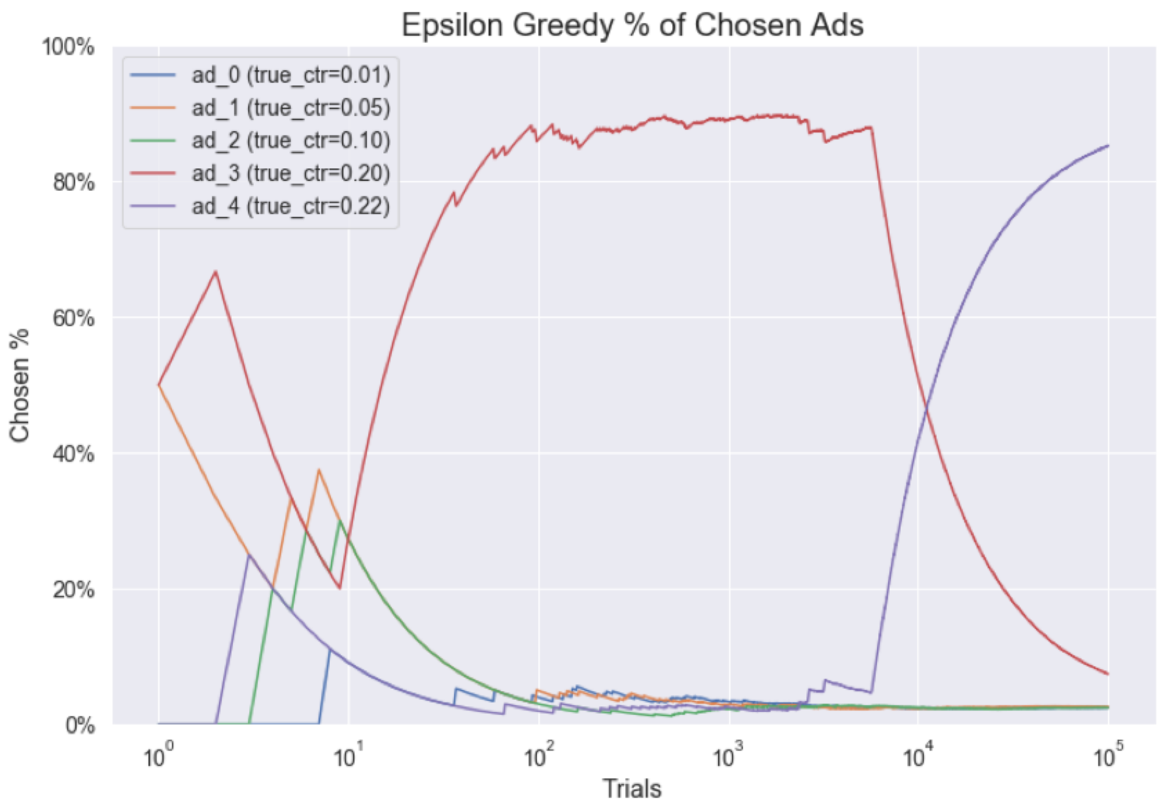
The code below runs 100,000 trials on a collection of five ads with theoretical CTRs ranging from 1% to 22% and an ε of 0.1.

```
b =
EpsilonGreedyBandit(true_
```

```
b = EpsilonGreedyBandit(true_ctr=[0.01, 0.05, 0.10, 0.20, 0.22], epsilon=0.1)
for trial in range(0,100000):
  b.run_trial()
```

If we track the state of the bandit through the loop, we can uncover some really interesting insights into how the bandit is thinking. For example, the two charts below show the bandit's empirical CTR for each ad and the cumulative percentage of ads chosen for each of the ads in our collection.





Since this is a contrived toy example, the behavior we want the bandit to exhibit is to eventually find that ad_4 with a true, underlying CTR of 22% is the optimal ad to choose. The sooner it finds that out the better, because that's how we maximize clicks. Our bandit eventually finds the optimal ad, but it appears to get stuck on the ad with a 20% CTR for quite a while which is a good — but not the best — solution. This is a common problem with the epsilon-greedy strategy, at least with the somewhat naive way we've implemented it above. There's the possibility that it will get stuck in a sub-optimal choice due to the relatively simplistic way we're managing the exploration vs. exploitation trade-off.

There are ways to augment the vanilla epsilon-greedy strategy to solve for this issue, such as using a decaying ε where the algorithm starts off very exploratory and gradually gets more exploitive over time. Instead of going down that path, we're going to solve this issue with an alternative strategy called Thompson sampling.

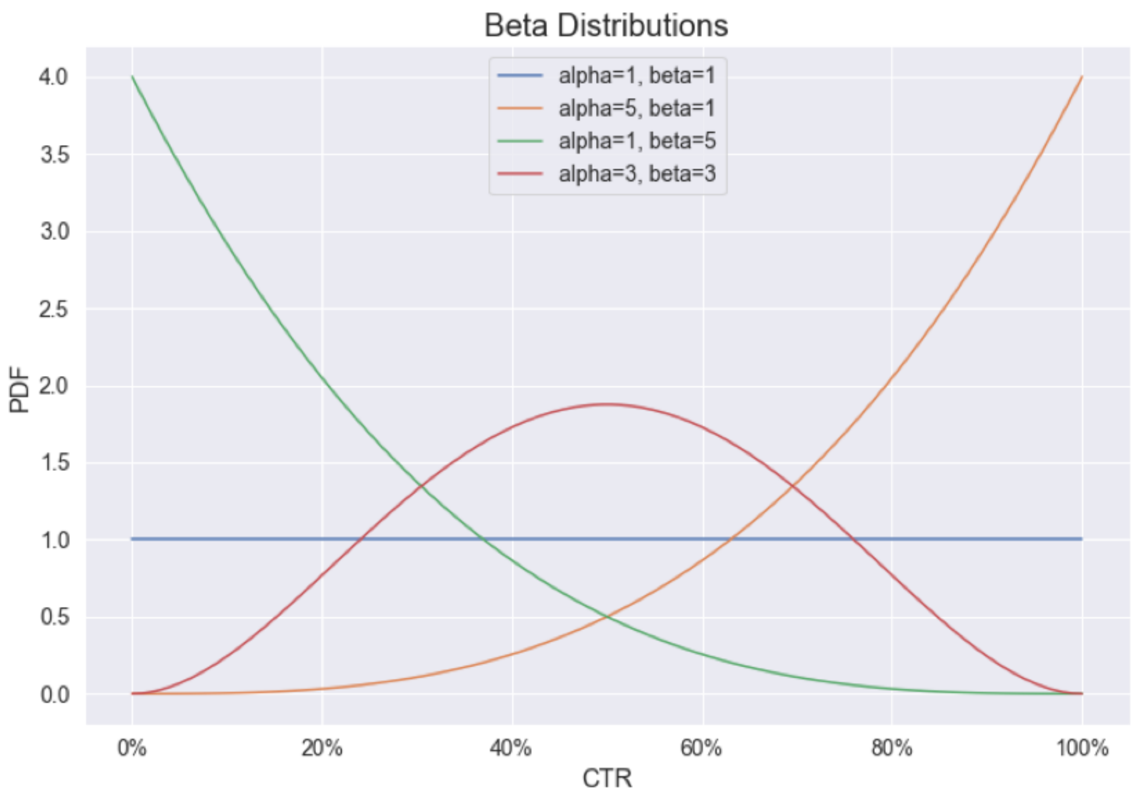
Thompson sampling bandits

Thompson sampling bandits (sometimes called Bayesian bandits) aim to tackle the exploration vs. exploitation trade-off in a very different way, borrowing from a Bayesian way of thinking. Instead of simply having the bandit track the empirical CTR after each trial, our bandit is going to maintain beliefs about where it thinks the true CTR of each ad lies. Initially, it will be very unsure about which ad is optimal and give the benefit of the doubt to ads it hasn't chosen very often, but it will slowly get more and more confident and eventually choose the optimal ad almost exclusively.

We'll use the beta distribution to model our bandit's beliefs about each ad's true CTR. There are theoretical reasons for the choice of this distribution, but intuitively the reason why this might be an appropriate distribution to model our beliefs is relatively straightforward.

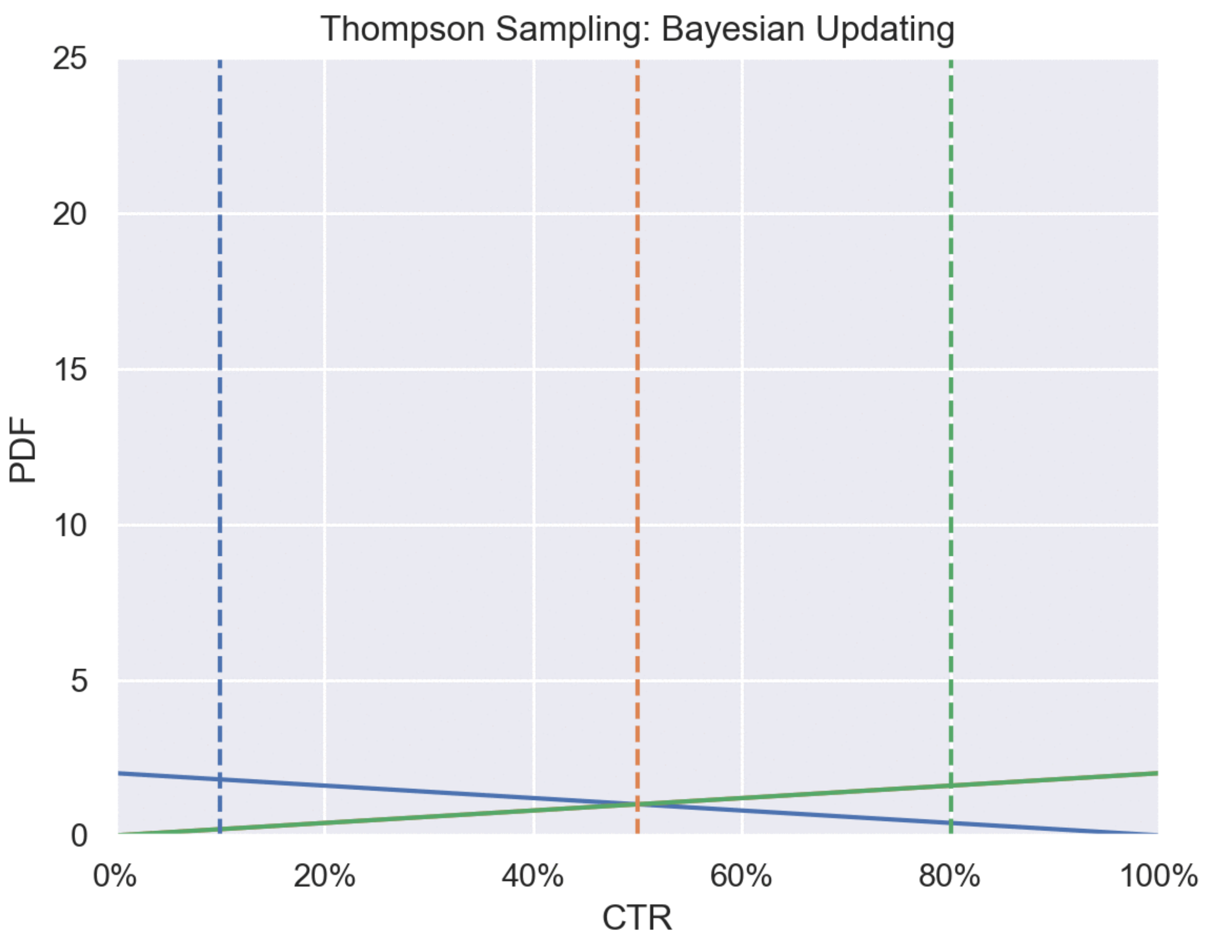
The beta distribution is parameterized by two, positive shape parameters

— α and β. Let's take a look at some example beta distribution shapes to see how it might work for our use case.



Consider the blue PDF where α=β=1. This is the belief we'll give our Bandit to start because it doesn't have any data to form an informed opinion about the true CTR for any of the ads. Every time this ad is chosen however, we're going to update this distribution to reflect the evidence of clicks/no-clicks we've observed up to that point. We do this by incrementing α whenever a click is observed and β when a click is not observed. The orange and blue PDFs represent the new distribution after 3 consecutive clicks/no-clicks respectively, and we can see that the distribution shifts toward 100% as more clicks are observed and 0% with more no-clicks. The last red PDF represents our beliefs after observing the same number of clicks and no-clicks, and as you might expect it's centered around 50%

This process is sometimes called Bayesian Updating. As more trials are conducted the distributions begin to get taller and narrower, converging on the true CTR as the Bandit becomes more confident in its beliefs. Below is an animated simulation of Bayesian Updating on 100 trials for 3 ads with theoretical CTRs of 10%, 50% and 80%.



Now that we have a way to model out our belief of where we think the true CTR for each ad lives, here's how we go about choosing which ad to serve next:

1. Randomly draw a CTR from each ad's beta distribution.
2. Choose the ad with the highest drawn CTR.
3. If a click is observed increment *α* by 1, otherwise increment *β* by 1.

That's it! Let's implement a Thompson Sampling Bandit in Python.

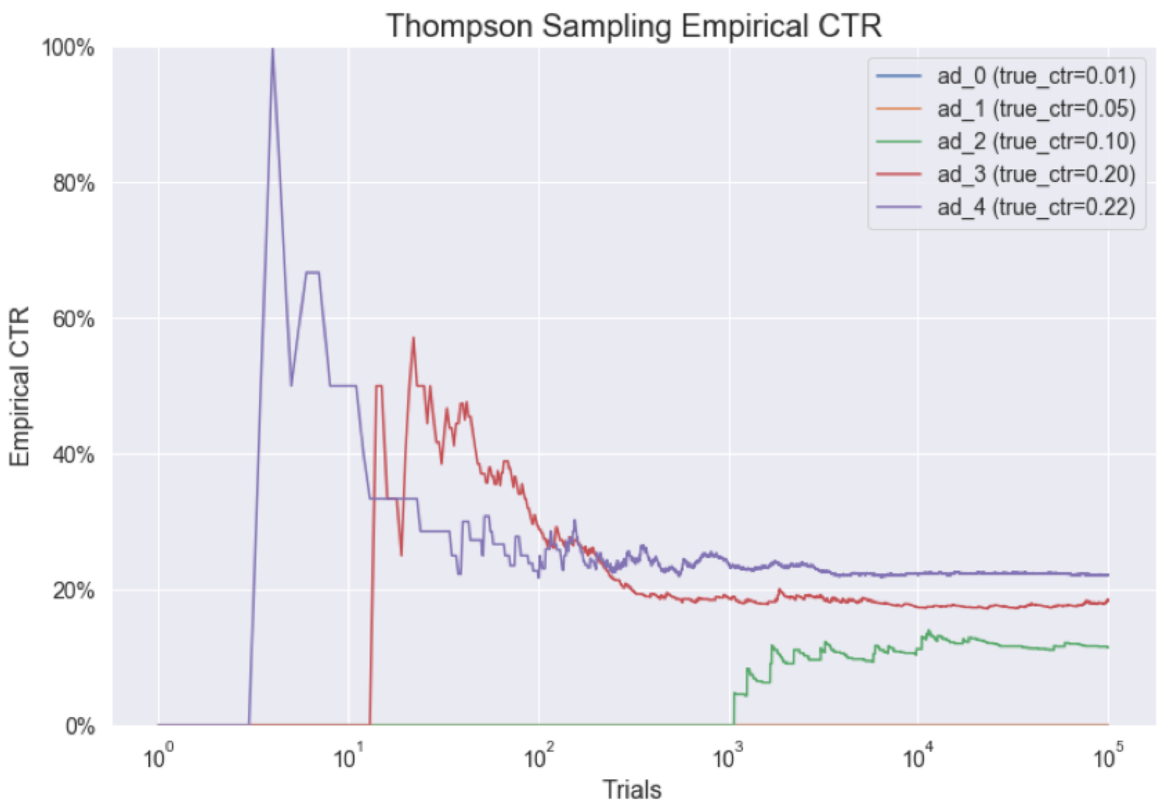
```
class
ThompsonSamplingBandit
```

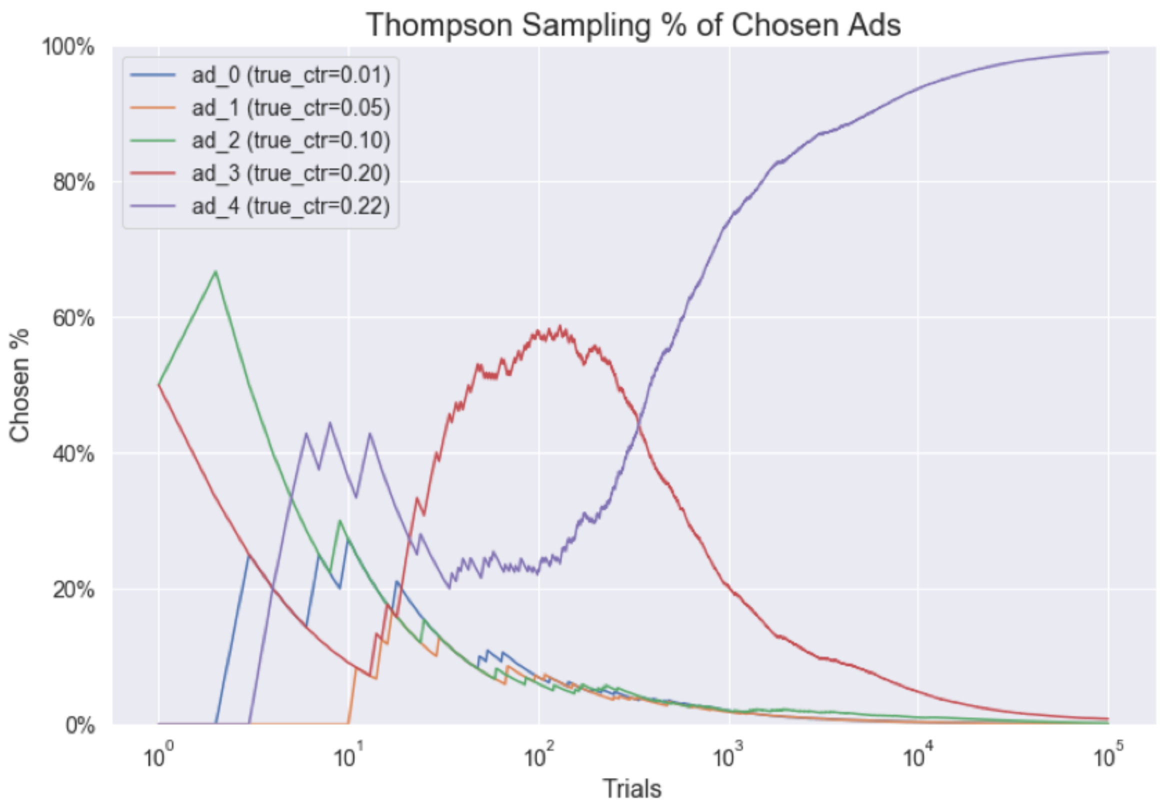
```
class ThompsonSamplingBandit(object):

  def __init__(self, true_ctr):
     self.num_ads = len(true_ctr)
     self.ad_ids = np.arange(self.num_ads)
     self.clicks = np.zeros(self.num_ads)
     self.alphas = np.ones(self.num_ads)
     self.betas = np.ones(self.num_ads)
     self.trials = np.zeros(self.num_ads)
     self.true_ctr = true_ctr
     self.empirical_ctr = np.zeros(self.num_ads)

  def _choose_ad(self):
     """
     Chooses which ad to play from "best" ad randomly drawn from
     beta distributions.
     """
     beta_draws = np.zeros(self.num_ads)
     for ad in self.ad_ids:
        draw = np.random.beta(self.alphas[ad], self.betas[ad], size=1)[0]
        beta_draws[ad] = draw

     best_ad = np.argmax(beta_draws)
     return best_ad
```

```

def _click(self, ad):
    """
    Observes whether the ad was clicked or by drawing from
    underlying probability distribution.
    """
    click = np.random.binomial(size=1, n=1, p=self.true_ctr[ad])[0]
    return click

def _update(self, ad, click):
    """
    Updates clicks, trials, empirical ctr and beta distribution
    parameters.
    """
    self.trials[ad] += 1
    self.clicks[ad] += click
    self.empirical_ctr[ad] = self.clicks[ad] / self.trials[ad]
    if click:
        self.alphas[ad] += 1
    else:
        self.betas[ad] += 1

def run_trial(self):
    """
```

```
        Runs a complete trial of choosing an ad, observing the click,
        and updating.
        """
        ad = self._choose_ad()
        click = self._click(ad)
        self._update(ad, click)
```

Now let’s take a look at how our Bandit behaves with this new strategy like we did with our Epsilon Greedy implementation. Below are the plots showing the empirical CTR and percent of chosen ads over 100,000 trials.





Comparing these plots to what we had under the Epsilon Greedy strategy uncovers some interesting differences. One difference is that the empirical CTRs for ad_0 and ad_1 with 1% and 5% are actually showing 0% under this new strategy. Epsilon Greedy, on the other hand, got much closer to the true CTRs for these ads. The reason this happens is that no matter how poor performing the ad or how many trials, Epsilon Greedy guarantees every ad gets some minimum chance of being chosen. Thompson Sampling allows the Bandit to realize pretty early on that these ads are very unlikely to be the optimal given our collection and effectively stops choosing them. This is actually a benefit of this strategy because we don’t want our Bandit wasting choices on poor performing ads.

Another difference is the speed at which the Bandit arrives at the best ad. Under the Thompson Sampling strategy, our Bandit finds and starts choosing the top performing ad after roughly 200 trials, but with Epsilon Greedy it took us 6,000 trials for the same ad to be chosen the most often! Visual inspection of these plots seems to indicate that Thompson Sampling is a better strategy than Epsilon Greedy, but is there a way to measure how much better?

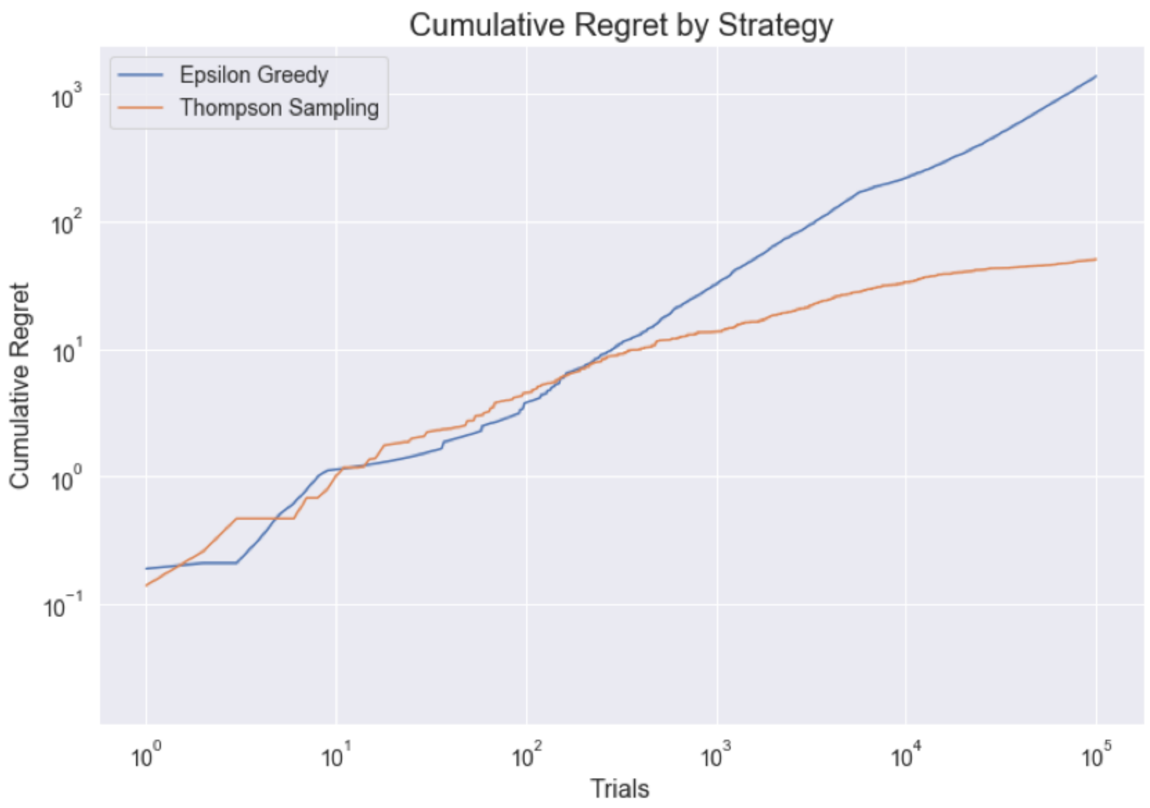
Regret

A common measure of the "goodness" of a particular strategy is one that minimizes something called regret. Regret in this case is nearly identical to how we think of regret in everyday life: what you missed out on by not choosing the best option. For MABs we define cumulative regret as follows:

$RT=\sum t=1T(p_*-pA(t))$

where p_* is the true CTR for the best performing ad and $pA(t)$ is the CTR of the ad chosen in the tth trial. Note that the lower bound on regret is 0 because the best we can possibly do with any strategy is select the best ad for each trial.

To compare the performance of Epsilon Greedy and Thompson Sampling, we're going to plot the cumulative regret after each trial and see what the resulting curves look like.



This view shows Thompson Sampling clearly outperforming Epsilon Greedy which is in line with our intuition earlier. Don't let the log scale on the y-axis fool you – by the end of our 100,000 trial experiment, Epsilon Greedy suffers from 30 times the regret of Thompson Sampling!

Conclusion

Multi-Armed Bandits (MABs) are powerful algorithms to consider if you're seeking to optimize among a collection of possible alternatives. We've seen that optimizing CTR is one possible application of MABs in the digital advertising space, but there are countless other opportunities to allow these algorithms to automatically learn through trial and error when the volume and velocity of decisioning makes having an analyst in the loop infeasible.

This article was written by Dave King, data scientist at SpotX.