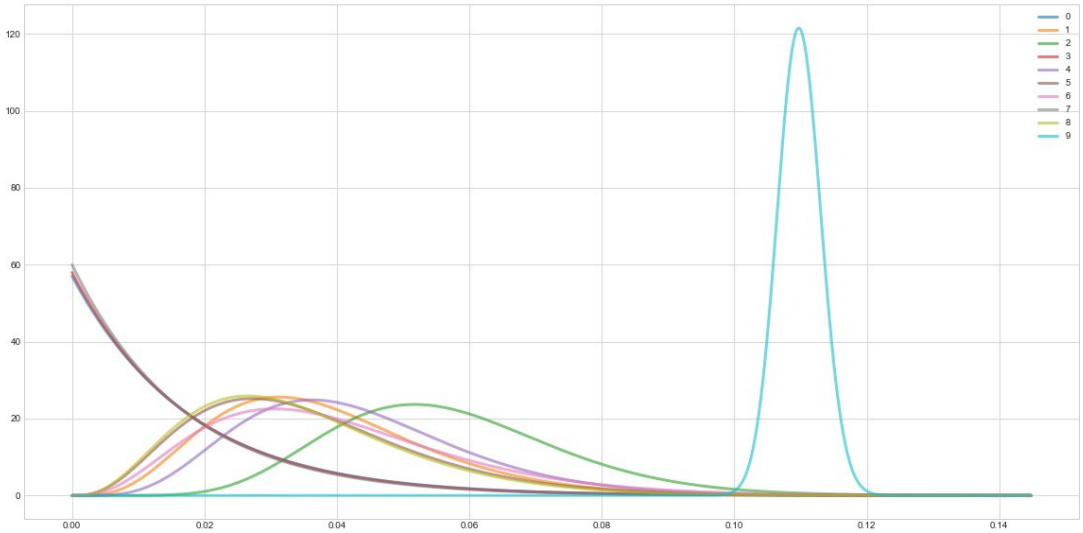
# Solving multiarmed bandits: A comparison of epsilon-greedy and Thompson sampling

Conor McDonald Follow

Sep 30, 2018 · 12 min read




## The Multiarmed-bandit problem

The multi-armed bandit (MAB) is a classic problem in decision sciences. Effectively, it is one of optimal resource allocation under uncertainty. The name is derived from old slot machines that where operated by pulling an arm — they are called bandits because they rob those who play them. Now, imagine there are multiple machines and we suspect that the *payout rate* — the *payout to pull* ratio — varies across the machines. Naturally we

want to identify the machine with the highest payout and *exploit it* — i.e. pull it more than the others.

The MAB problem is this; how do you most efficiently identify the best machine to play, whilst sufficiently exploring the many options in real time? This problem is not an exercise in theoretical abstraction, it is an analogy for a common problem that organisations face all the time, that is, how to identify the best message to present to customers (message is broadly defined here i.e. webpages, advertising, images) such that it maximises some business objective (e.g. clickthrough rate, signups).

The classic approach to making decisions across variants with unknown performance outcomes is to perform multiple A/B tests. These are typically run by evenly directing a percentage of traffic across each of the variants over a number of weeks, then performing statistical tests to identify which variant is the best. This is perfectly fine when there are a small number of variations of the message (e.g. 2–4), but can be quite inefficient in terms of both *time* and *opportunity* cost when there are many.

The time argument is easy to grasp. Simply, more variations requires more A/B tests, which take more time, thus delaying "feedback" and decision making. The opportunity cost argument is more subtle. In economics, the opportunity cost is the cost that is associated with taking one action rather than another. Simply, what did I miss out on by putting my money into investment A rather than that investment B? Investment B is the opportunity cost of taking investment A. In the variant testing world this translates to sending a customer to A rather than B.

For good reasons A/B tests should not be "peeked" at whilst the tests are running. This means that the experimenters will not know which variant is best until they end the test. However, it is typically hypothesised that one variant will outperform the others. What does this mean in real terms? It means that A/B testing involves consciously sending a proportion of your customers to a suboptimal message (though you don't know which one!) — perhaps if these customers where sent to the optimal variant they might have signed up for your service.

This is the opportunity cost in A/B testing. For one test this is acceptable. However, when there are many variants to test, this means that you are potentially directing many customers to suboptimal variants for a long period of time. It would be better in this

scenario if we could, in real time, quickly rule out the dud variants, without directing too much traffic to them; then after a number of effective variants have been identified perform an A/B test on this smaller subset (this is usually required for statistical power). Bandit algorithms or samplers, are a means of testing and optimising variant allocation quickly.

In this post I'll provide an introduction to Thompson sampling (TS) and its properties. I'll also compare Thompson sampling against the epsilon-greedy algorithm, which is another popular choice for MAB problems. Everything will be implemented from scratch in Python — all of the code can be found here.

I use the following vocabulary in the post:

***Trial:*** A customer arriving on a webpage

***Message***: the image/words/colours etc. being tested

***Variant:*** different variations of a message (ad/image/webpage etc.)

***Action:*** The action taken by an algorithm, that is, the variant it decides to show

***Reward***: The business objective, for example a signup to a service and click-through. For simplicity we will assume that rewards are binomially distributed. That is, a reward is either a 1 or 0 (click through or not)

***Agent***: The algorithm that makes decisions concerning which variant to show. I also refer to these as bandits and samplers

***Environment***: The context in which the agent operates — i.e. the variants and their latent "*payouts*"

## Exploration and Exploitation

Bandit algorithms are approaches to realtime/online decision making that strive to strike a balance between sufficiently exploring the variant space and exploiting the optimal action.

Striking a balance between the two is critically important. Firstly, the variant space needs to be sufficiently explored such that the strongest variant is identified. By first identifying then continuing to exploit the optimal action you are maximising the total reward that is available to you from the environment. However, you also want to continue to explore other feasible variants in case they provide better returns in the future. That is, you want to hedge your bets somewhat by continuing to experiment (a little) with sub-optimal variants in the event that their payouts change. If they do, your algorithm will pick up on the change and will begin selecting this variant for new customers. A further benefit of exploration is that you learn more about the generating process underlying the variant. That is, what is its average payout rate and what is distribution of uncertainty around that. The key, therefore, is to decide how best to balance this *exploration-exploitation tradeoff*.

## Epsilon-Greedy

A common approach to balancing the exploitation-exploration tradeoff is the *epilson-* or *e-greedy* algorithm. Greedy here means what you probably think it does. After an initial period of exploration (for example 1000 trials), the algorithm greedily exploits the best option $k$, $e$ percent of the time. For example, if we set $e=0.05$, the algorithm will exploit the best variant 95% of the time and will explore random alternatives 5% of the time. This is actually quite effective in practice, but as we'll come to see it can under explore the variant space before exploiting what it estimates to be the strongest variant. This means that e-greedy can get stuck exploiting a suboptimal variant. Let's dive into some code to show e-greedy in a action.

First some dependencies and boiler plate. We need to define the environment. The environment is the context in which the algorithms will run. In this case, it is very simple. The environment calls on an an agent (i.e. a bandit algorithm) to "decide" what action to take, it then runs the action, observes the reward and communicates the reward back to the agent, which updates itself. This has many simularities to reinforceent learning and, indeed, MAB algorithms are considered to be a "lite" form of RL.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import seaborn as sns
4  import pandas as pd
```

```
import pandas as pd
from scipy.stats import beta

sns.set_style("whitegrid")

class Environment:
    def __init__(self, variants, payouts, n_trials, variance=False):
        self.variants = variants
        if variance:
            self.payouts = np.clip(payouts + np.random.normal(0, 0.04, size=len(variants)), 0, .
        else:
            self.payouts = payouts
        #self.payouts[5] = self.payouts[5] if i < n_trials/2 else 0.1
        self.n_trials = n_trials
        self.total_reward = 0
        self.n_k = len(variants)
        self.shape = (self.n_k, n_trials)

    def run(self, agent):
        """Run the simulation with the agent.
        agent must be a class with choose_k and update methods."""

        for i in range(self.n_trials):
            # agent makes a choice
            x_chosen = agent.choose_k()
            # Environment returns reward
            reward = np.random.binomial(1, p=self.payouts[x_chosen])
            # agent learns of reward
            agent.reward = reward
            # agent updates parameters based on the data
            agent.update()
            self.total_reward += reward

        agent.collect_data()

        return self.total_reward

class BaseSampler:

    def __init__(self, env, n_samples=None, n_learning=None, e=0.05):
        self.env = env
        self.shape = (env.n_k, n_samples)
        self.variants = env.variants
        self.n_trials = env.n_trials
        self.payouts = env.payouts
```

```
        self.ad_i = np.zeros(env.n_trials)
        self.r_i = np.zeros(env.n_trials)
        self.thetas = np.zeros(self.n_trials)
        self.regret_i = np.zeros(env.n_trials)
        self.thetaregret = np.zeros(self.n_trials)

        self.a = np.ones(env.n_k)
        self.b = np.ones(env.n_k)
        self.theta = np.zeros(env.n_k)
        self.data = None
        self.reward = 0
        self.total_reward = 0
        self.k = 0
        self.i = 0

        self.n_samples = n_samples
        self.n_learning = n_learning
        self.e = e
        self.ep = np.random.uniform(0, 1, size=env.n_trials)
        self.exploit = (1 - e)

    def collect_data(self):

        self.data = pd.DataFrame(dict(ad=self.ad_i, reward=self.r_i, regret=self.regret_i))
```

bandits_boilerplate.py hosted with ♡ by GitHub view raw

The reward is binomially distributed with probability $p$ being determined by the action taken (note: this can easily be extended to continuous outcomes). Here I also define a BaseSampler class. The purpose of this class is really only to store the various attributes and logs (for visualization) that are common across bandits. Here we also define our variants and their latent, but unknown, payout rates. In total, we will test 10 variants. The best option is variant 9, which has a payout rate of 0.11%.

```
variants = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
payouts = [0.023, 0.03, 0.029, 0.001, 0.05, 0.06, 0.0234, 0.035,
0.01, 0.11]
```

For baseline comparison, I also define the RandomSampler class. This simply selects a variant at random for each trial, it does not learn, it does not update. This is purely to benchmark the other agents.

```
class RandomSampler(BaseSampler):
    def __init__(self, env):
        super().__init__(env)

    def choose_k(self):

        self.k = np.random.choice(self.variants)

        return self.k

    def update(self):
        # nothing to update
        self.thetaregret[self.i] = np.max(self.theta) - self.theta[self.k]

        self.a[self.k] += self.reward
        self.b[self.k] += 1
        self.theta = self.a/self.b

        self.ad_i[self.i] = self.k
        self.r_i[self.i] = self.reward
        self.i += 1
```

RandomSampler.py hosted with ♡ by GitHub view raw

The other agents follow this basic structure. They all implement *choose_k* and *update* methods. *choose_k* implements the policy through which the agent selects a variant. *update* updates the parameters of the agent — this is how the agent "evolves" its ability to select a variant (the RandomSampler class doesn't update anything). We run an agent in an environment using this pattern:

```
en0 = Environment(machines, payouts, n_trials=10000)
rs = RandomSampler(env=en0)
en0.run(agent=rs)
```

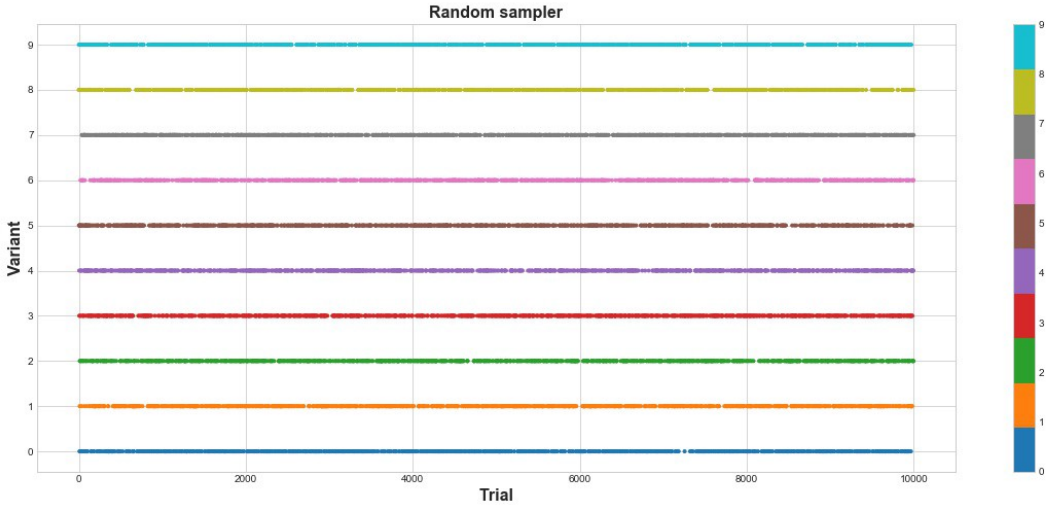
Descriptions of each component of the e-greedy algorithm are inline, but the core of the algorithm is this:

- randomly choose *k* for n trials
- On each trial estimate the payout rate for each variant
- after n learning trials:
- select *1-e%* of the time *k* with the the highest payout rate and;
- *e%* of the time sample from the variants randomly

```
class eGreedy(BaseSampler):

    def __init__(self, env, n_learning, e):
        super().__init__(env, n_learning, e)

    def choose_k(self):

        # e% of the time take a random draw from machines
        # random k for n learning trials, then the machine with highest theta
        self.k = np.random.choice(self.variants) if self.i < self.n_learning else np.argmax(self
        # with 1 - e probability take a random sample (explore) otherwise exploit
        self.k = np.random.choice(self.variants) if self.ep[self.i] > self.exploit else self.k
        return self.k

    def update(self):

        # update the probability of payout for each machine
        self.a[self.k] += self.reward
        self.b[self.k] += 1
        self.theta = self.a/self.b

        self.thetas[self.i] = self.theta[self.k]
        self.thetaregret[self.i] = np.max(self.thetas) - self.theta[self.k]

        self.ad_i[self.i] = self.k
        self.r_i[self.i] = self.reward
        self.i += 1
```

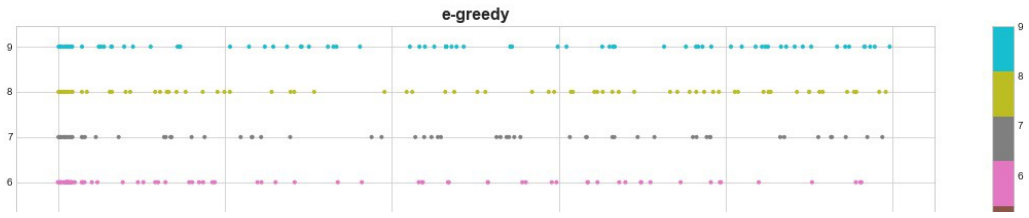
egreedy.py hosted with ♡ by GitHub view raw

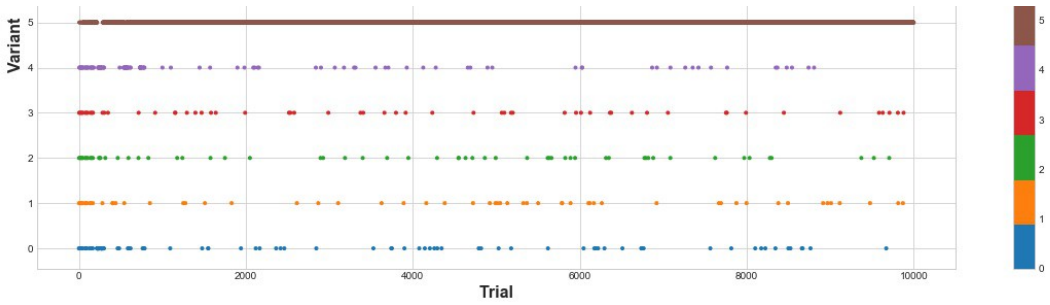
In the plot below you can see the results of a purely random sampling process. That is, there is no model guiding the agent's choice process. The plot shows the choices made on each trial across 10000 trials. This is pure exploration, without learning. The total reward achieved by this agent is 418.




Let's see how the e-greedy algorithm performs in the same environment. Again we will run the algorithm for 10000 trials and will set *e=0.05* and *n_learning=1000*. We will assess the algorithm on the total reward that it creates in the environment.

The total reward created by e-greedy is 788, this is almost a 100% improvement on random sampling — nice! Plot 2 provides a lot of insight into the algorithm. We can see that for the first 1000 learning steps the distribution of actions is relatively uniform as it is selecting *k* at random. After this point, it begins to heavily exploit option 5, which is a strong option with a 0.06% payout rate, though not the best. We can also see the random exploration that is being performed throughout the trials, as it randomly selects from the variant pool 10% of the time.






This is pretty cool, in just a few lines of code we have a pretty powerful algorithm capable of exploring the variant space and making close to optimal decisions within it. However, the algorithm did not identify the absolute best variant. Sure, we could increase the learning steps, but then we are wasting more trials by randomly searching, which will further hurt the total reward. Also, there is inherent randomness in this process, so if we were to run the algorithm again it is possible that e-greedy will identify and exploit the best variant.

Later I will run simulate multiple runs of each algorithm so that we can get a better insight in to their relative performance. First though, let's implement the Thompson sampler and test its performance in the same environment.

## Thompson sampler

The Thompson sampler differs quite fundamentally from the e-greedy algorithm in three major ways:

- it is not greedy;
- its exploration is more sophisticated, and;
- it is Bayesian

1 & 2 are because of 3.

As with *e*-greedy, the code is commented inline, but the gist of the algorithm is this:

```
- Set a uniform prior distribution between 0 and 1 for each variant
k's payout rate

- Draw a parameter theta from the each k's posterior distribution
```

```
- Select the variant k that is associated with the highest parameter
theta

- Observe the reward and update the distribution parameters
```

More formally, for each trial its:

$$\hat{\theta}_k \sim Beta(\alpha_k, \beta_k)$$
$$\theta = max(\hat{\theta}_k)$$
$$k = argmax(\hat{\theta}_k)$$
$$reward_t \sim Bernoulli(p = \theta)$$
$$\alpha_k, \beta_k = \alpha_k + reward, \beta_k + 1 - reward$$

Note that here I am defining Thompson Sampling as a Beta Bernoulli sampler. However, Thompson sampling can be generalized to sample from any arbitrary distributions over parameters. The Beta Bernoulli version of the TS is a good way to build intuition for the algorithm and is actually the often the best option for many problems in practice. If you are looking for a primer on the Beta distribution, look no further than David Robinson's blog post.

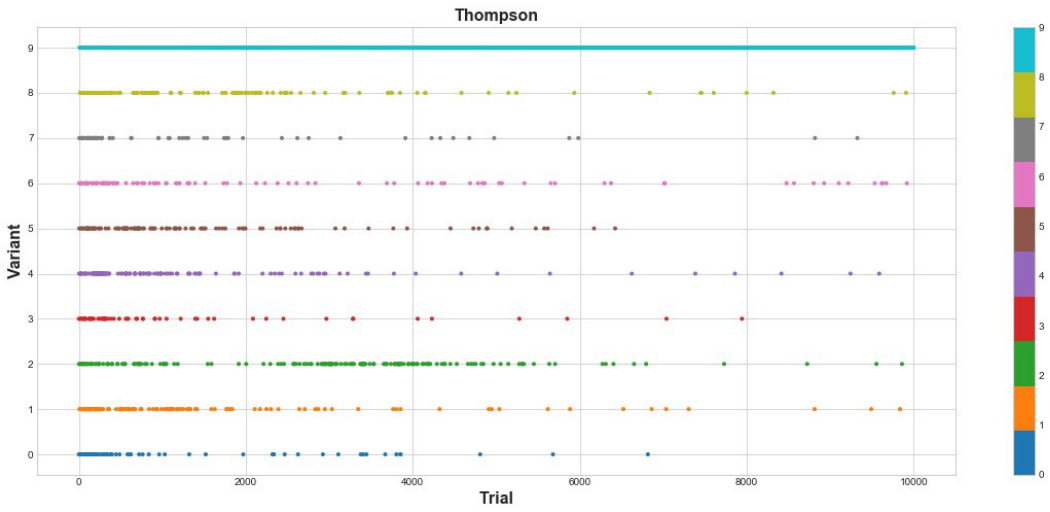
Let's code this up. As with the other samplers, Thompson is implemented as a class that inherits BaseSampler and defines its own *choose_k* and *update* methods.

```
1  class ThompsonSampler(BaseSampler):
2
3      def __init__(self, env):
4          super().__init__(env)
5
6      def choose_k(self):
7          # sample from posterior (this is the thompson sampling approach)
8          # this leads to more exploration because machines with > uncertainty can then be selecte
9          self.theta = np.random.beta(self.a, self.b)
10         # select machine with highest posterior p of payout
11         self.k = self.variants[np.argmax(self.theta)]
12         return self.k
13
14     def update(self):
15
16         #update dist (a, b) = (a, b) + (r, 1 - r)
```

```
17          self.a[self.k] += self.reward
18          self.b[self.k] += 1 - self.reward # i.e. only increment b when it's a swing and a miss.
19
20          self.thetas[self.i] = self.theta[self.k]
21          self.thetaregret[self.i] = np.max(self.thetas) - self.theta[self.k]
22
23          self.ad_i[self.i] = self.k
24          self.r_i[self.i] = self.reward
25          self.i += 1
```

thompsonsamper.py hosted with ♡ by GitHub view raw

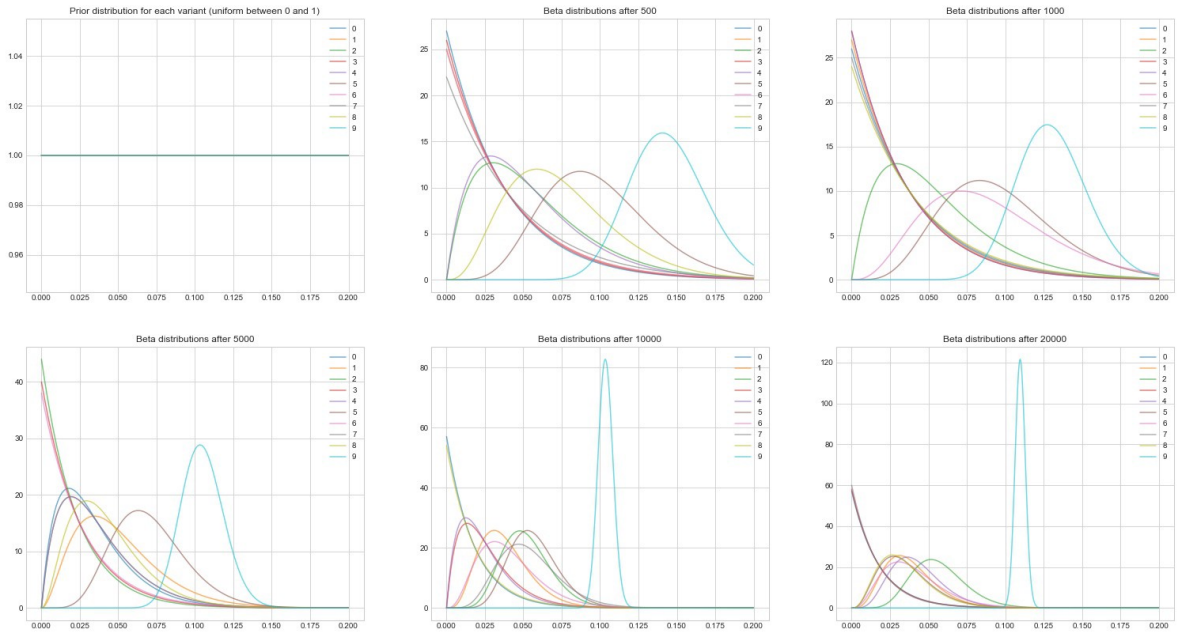
Now to run the sampler. The total reward with the Thompson Sampler as the agent is 1023. That's quite a bit more than e-greedy, cool! Let's check out the sample plot. There are two things that stand out here. Firstly, the agent correctly identifies the best option (*variant 9*) and exploits this the most. However, it also explores the other variants, but in a more sophisticated way. If you look closely at the variants the agent samples most (after 1000 trials or so), they tend to be variants that are stronger. In other words, its exploration is more guided, rather than purely random.




You may be wondering why this works. Simply, the uncertainty in the posterior distribution of each variant's payout rate means that on each trial each variant has a probability of being picked that is roughly proportional to its its shape as determined by its alpha and beta parameters. That is, on each trial Thompson Sampling runs a variant

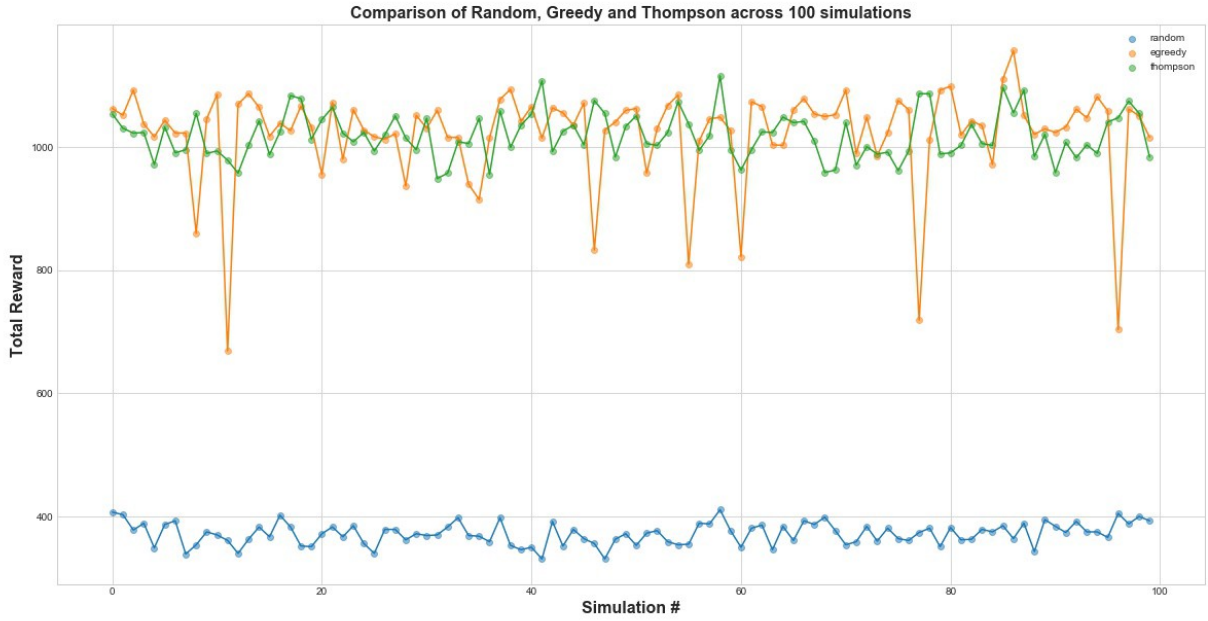
according to its posterior probability of having the best payout. More simply, with uncertainty information explicitly captured via a posterior distribution the agent can decide when to exploit and when to explore its environment. For example, a weak variant with high posterior uncertainty may have the largest expected payout for a given trial. However, for most trials, the strongest variant's posterior distribution will have the largest mean and smallest standard deviation, meaning it has the greatest probability of being selected.

Another nice property of the Thompson algorithm, is that its Bayesian properties mean that we can fully inspect the uncertainty of its payout rate. Here I plot the posterior distributions at 6 different points in a run of 20000 trials. You can see how the distributions gradually begin to converge towards the variant with the best payout rate.



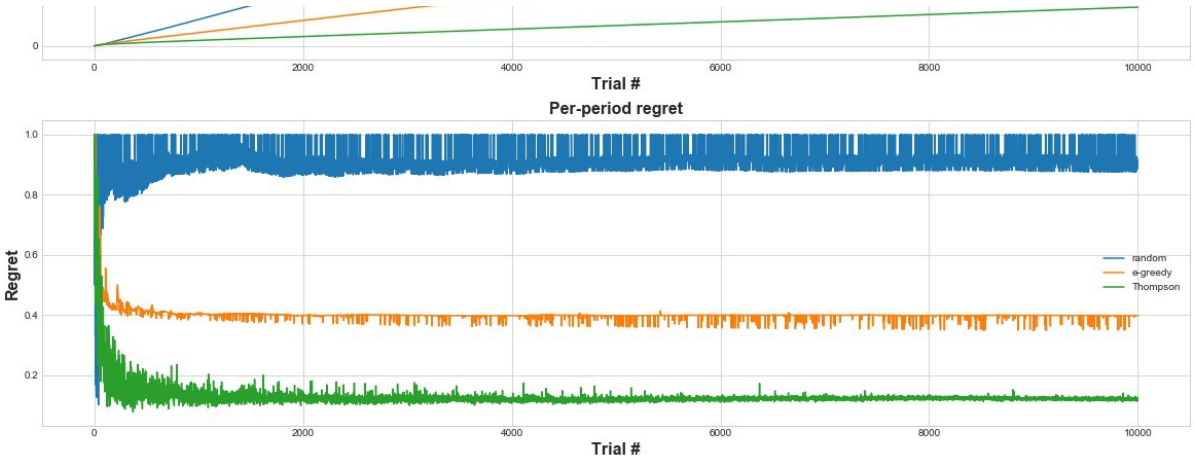

Let's now compare all three agents across 100 simulations. A simulation is a single run of 10000 trials per bandit. Here we compare each on their total reward across each simulation. As you can see from the plot both *e*-greedy and Thompson strongly outperform random sampling. You may be surprised that *e*-greedy and Thompson are usually quite comparable in terms of their total reward performance. *E*-greedy can be very effective, but it is riskier because it can get stuck — as it did in the first instance I

ran it when it scored just 788. We can see this in some of the low points in the plot. On these runs, *e*-greedy has selected a suboptimal variant and has exploited it the most. As you can see Thompson never gets stuck in this way. This due to its more sophisticated exploration of the variant space.




## Regret

An additional means of assessing the algorithms' performance is through the concept of regret. Intuitively, regret is quite simple. The algorithm's regret concerning its action (what *k* to show) should be as low as possible. Simply, regret is the difference between the best performance from a variant so far and the performance from the variant chosen for the current trial *t*. Below I plot both cumulative regre and regret per trial *t*. In case it isn't obvious, regret should be minimised.




The top plot shows the cumulative regret and the bottom the per trial regret. We can see from the regret plots that Thompson converges towards the least regret much better than *e*-greedy (we don't really expect the random sampler to converge). Also, the top plot shows that the cumulative regret for *e*-greedy has a steeper slope than Thompson. We can also see that the regret minimises and converges better with Thompson.

With Thompson the agent regrets less because it is can better identify the best variant and is more likely to explore those variants that are strong performers — this makes the Thompson bandit particular well suited to more advanced use cases that may involve statistical models or neural nets in the choice of $k$.

## Summary

This is quite a long and technical post. To summarise, we can use sophisticated sampling methods whenever we have many variants that we wish to test in an online/realtime setting. One of the very nice properties of Thompson sampling is that it balances exploration and exploitation in a sophisticated way. In practice this means that we can let it optimise our variant allocation decisions in real time. These are nice algorithms to play around with, but they can also deliver serious value to businesses over A/B testing alone.

One caveat that is worth stating is that Thompson sampling doesn't necessarily remove the need for A/B tests. It is common to perform an A/B test on the best variants that have been identified by the Thompson sampling.

Another way to think of MAB algorithms is through the lens of reinforcement learning. In essence, they auto-optimise without prior knowledge of their task, taking rewards from the environment to update their parameters. In my next post on this subject, I'll explore this angle in more detail through *Contextual Bandits* — that is, bandits that enable personalisation of variants based on other information, such as features of the user visiting the webpage.

Thanks for reading, hope you found it interesting.