Better than A|B Testing

Brent E. Edwards — https://www.linkedin.com/in/brenteedwards/

A|B Testing

A|B testing has been used for decades. Here is a simplified explanation:

1. To set it up, you need:
    1. Something, like a web page, that is shown independently repeatedly
    2. Two (or more) variants, *v*1 and *v*2, of that thing
    3. A measurement to optimize that (hopefully) depends on the variants
2. For the next *n views*, half of the people are shown *v*1 and the other half are shown *v*2.
3. Whichever variant had the better measurements is considered "better."

A common example of A|B testing is measuring which of two wordings for a smaller website leads to more click throughs.

Statisticians love A|B testing because it can detect whether the measurement might have been considered "better" because of random chance. This detection is called a Student's t-test.

But for improving web sites, managers want to move to the "better" solution quickly. They don't care whether the test picks a random choice as long as, if one of the variants is better, the program chooses it quickly.

Multi-Armed Bandit

The “multi-armed bandit” problem is a variation on A|B testing. It has a slightly different setup than A|B testing:

Imagine that you have *n* coins for *m* slot machines. Each pull of a slot machine costs one coin. You don’t know the odds on the slot machines, but you want to get as much of the reward as you can. How do you choose which arms to pull?

In the multi-armed bandit problem, two ideas battle against each other:

Exploitation uses the information that you already have to choose the slot machine with the best known payoff.

Exploration chooses a slot machine that will give more information on its odds.

I have written software that lets you experiment with the multi-armed bandit problem at https://github.com/chipuni/MultiArmedBandit . It includes several possible solutions for the multi-armed bandits problem and an extensive test. This code contains a binary payoff, and the two choices have a different (but unknown) fixed probability.

The code runs 4560 tests and it outputs two measurements:

- It counts the total number of hits across all tests that the multi-armed bandit chose (*m*), the total number of hits that would have been chosen if the smaller had always been chosen (*s*), the total number of hits that would have been chosen if the larger had always been chosen (*l*), and finally calculates (*m* - *s*) / (*l* - s). I call this the ‘hit accuracy’. It measures exploitation: the higher it is, the more hits happened.
- After every test, it determines whether, if one more choice had been taken, would it have chosen the correct side? I call this the ‘correct side’ metric. It measures exploration: the higher it is, the better future results are likely to be.

So that you can compare, the A|B test would have a hit accuracy of 0.482 but a correct side measurement of 4297. All other solutions for the multi-armed bandit should have a better hit accuracy but a worse correct-side measurement.

Simplest Multi-Armed Bandit

There are two extreme solutions for A|B testing:

If you do pure exploration, you get A|B testing. It never exploits the information that it already has.

Alternatively, you can do pure exploitation: always choose the bandit that has given the best payoff so far.

In my code, pure exploitation is called SimplestMultiArmedBandit (https://github.com/chipuni/MultiArmedBandit/blob/main/SimplestMultiArmedBandit.py). The definition for the bandit is simple:

```
class SimplestMultiArmedBandit(MultiArmedBandit):
    def choose_side(self):
        side1 = self.generator1.mean()
        side2 = self.generator2.mean()
        return self.convert_max_to_choice(side1, side2)
```

In this code, `.mean()` represents the average return for a generator so far. So this code always chooses the side that has the higher average payout so far.

How well does it work? It chooses 3443 out of the 4560 sides correctly and it gives a measured accuracy of 0.796.

First A|B testing, then exploitation

A solution for A|B testing is to take the definitions of exploration and exploitation literally, and as different phases. This means use a small but fixed number of coins to train the payoffs for each side, then use that as a model to choose the better side from then on.

The definition for the bandit is almost as simple as the pure exploration model. It can be found at https://github.com/chipuni/MultiArmedBandit/blob/main/FixedExplorationMultiArmedBandit.py :

```
    class FixedExplorationMultiArmedBandit(MultiArmedBandit):

    def choose_side(self):
        if self.generator1.get_attempts() +
self.generator2.get_attempts() < self._count_exploration:
            # Exploration
            side1 = self.generator1.get_attempts()
            side2 = self.generator2.get_attempts()
            return self.convert_min_to_choice(side1, side2)
        else:
            # Exploitation
            side1 = self.generator1.mean()
            side2 = self.generator2.mean()
            return self.convert_max_to_choice(side1, side2)
```

In this code, `.get_attempts` represents the number of times that the generator has been used. So for the first self._count_exploration attempts, the code just chooses whichever generator has been used less (picking a random one if the two are equal), and after `self._count_exploration` attempts, it chooses the side that has had the highest payout so far.

This code has a parameter. How do the results look?

count_exploration	correct side	hit accuracy
100	3688	0.871
200	3810	0.905
500	4070	0.937
1000	4096	0.916
2000	4096	0.876

As expected, as the number of exclusively exploration values increases, the algorithm chooses the correct side more frequently. The number of hits, however, appears to rise and fall.

READER EXERCISE: Find the fraction of exploration to exploitation that optimizes the hit accuracy. Then change the seeds for the random numbers and try it again. Do those values change if each test (rather than each hit) had the same weight? Now, change the breakdown in the test suite and optimize for that.

Epsilon-Greedy

Another obvious way to mix exploration and exploitation is through an epsilon-greedy algorithm. Most of the time, exploit what you already know. But with a probability ϵ, choose a random variant.

The source code for the epsilon-greedy multi-armed bandit can be found at https://github.com/chipuni/MultiArmedBandit/blob/main/EpsilonGreedyMultiArmedBandit.py . The method itself follows:

```
    class EpsilonGreedyMultiArmedBandit(MultiArmedBandit):
        def __init__(self, generator1: Generator, generator2:
Generator, epsilon: float):
            super().__init__(generator1, generator2)
```

```
            self._randomizer = Generator(0.5, 34567)
            self._epsilon = Generator(epsilon, 45678)
            self._simplest_mab =
SimplestMultiArmedBandit(generator1, generator2)

        def choose_side(self):
            if self._epsilon.choose():
                if self._randomizer.choose():
                    choice = 1
                else:
                    choice = 2
            else:
                choice = self._simplest_mab.choose_side()
            return choice
```

The idea is exactly what was described before: At a probability ϵ, choose a random element. Otherwise, act exactly like the simplest multi-armed bandit.

How well does this idea work? Let's try a few values of epsilon:

epsilon	correct side	hit accuracy
0.01	3831	0.860
0.02	3973	0.882
0.03	4039	0.895
0.04	4152	0.903
0.05	4183	0.906
0.06	4208	0.904
0.07	4209	0.904
0.08	4220	0.903

READER EXERCISE: Try the same problems as the last reader exercise. Is a constant epsilon the optimal way to schedule the random choices? Doesn't it make more sense to

have more randomness at the beginning and less randomness later? If so, which kind of decrement gives the best result?

Thompson Sampling

Thompson sampling is the idea that the probability of picking an arm of the multi-armed bandit should be the same as the probability that it has the higher payout.

To determine the probability that one arm of the bandit has a higher payout than another, I slightly abuse the t-test. Instead of using it to determine whether two groups might be equal, I directly use the p-value as the probability that the expected value for one side is greater than the expected value for the other.

```
    class ThompsonMultiArmedBandit(MultiArmedBandit):
        def choose_side(self):
            # We need at least one hit on each side for this
chooser.
            if self.generator1.get_hits() == 0:
                return 1
            if self.generator2.get_hits() == 0:
                return 2

            # The probability that a side should be chosen
should match the probability that the
            # side has the higher average.
            mean1 = self.generator1.mean()
            std1 = self.generator1.stddev()
            nobs1 = self.generator1.get_attempts()

            mean2 = self.generator2.mean()
            std2 = self.generator2.stddev()
            nobs2 = self.generator2.get_attempts()

            # What is the chance that generator1's mean is
greater than generator2's mean?
            probability =
scipy.stats.ttest_ind_from_stats(mean1=mean1, std1=std1,
```

```
nobs1=nobs1, mean2=mean2, std2=std2, nobs2=nobs2,
equal_var=True, alternative='less').pvalue
            chooser = Generator(probability, time.time_ns())
            if chooser.choose():
                return 2
            else:
                return 1
```

How well did this work? It got 4071 correct sides, and its hit accuracy was 0.914.

Upper Confidence Bound (UCB) Algorithm

The upper confidence bound algorithm is beloved by many statisticians for two reasons:

1. It calculates how wrong the estimate might be, and it always chooses the side that, at its most optimistically, could be the better side.
2. It does not need a parameter.

The algorithm is fairly simple:

```
    class UCBMultiArmedBandit(MultiArmedBandit):
        def choose_side(self):
            # We need at least one attempt on each side.
            if self.generator1.get_attempts() == 0:
                return 1
            if self.generator2.get_attempts() == 0:
                return 2

            time = self.get_attempts()
            ucb1 = self.generator1.mean() + math.sqrt(2 *
math.log(time) / self.generator1.get_attempts())
            ucb2 = self.generator2.mean() + math.sqrt(2 *
math.log(time) / self.generator2.get_attempts())
```

```
            return self.convert_max_to_choice(ucb1, ucb2)
```

How well does it do? It got 3163 correct sides and its hit accuracy was 0.720.

Beta-Binomial method

The final idea for choosing the best arm is the beta-binomial method.

Because the beta distribution is a conjugate prior to the binomial distribution [https://math.mit.edu/~dav/05.dir/class15-prep.pdf], the percent of hits can be approximated by a beta distribution.

The key idea for the beta-binomial method is that when there is less information, the distribution is wider. By deciding which arm to pull by comparing two percent point functions with parameters above 0.5, an algorithm balances exploration and exploitation. The larger the percent point, the more exploration the algorithm does. Choosing a percent point function parameter of 0.5 would have no exploration, so it would be similar to the fixed exploration listed above.

The source code for the beta-binomial method is simple:

```
    class BetaBinomialMultiArmedBandit(MultiArmedBandit):
        def choose_side(self):
            beta1 = beta.ppf(self._frac,
self.generator1.get_hits(), self.generator1.get_attempts() -
self.generator1.get_hits())
            beta2 = beta.ppf(self._frac,
self.generator2.get_hits(), self.generator2.get_attempts() -
self.generator2.get_hits())
            return self.convert_max_to_choice(beta1, beta2)
```

where self._frac is the parameter for the percent point function.

How well does this idea work?

parameter	correct side	hit accuracy
0.50	2850	0.676
0.80	3395	0.786
0.90	3624	0.840
0.95	3775	0.874
0.99	4048	0.918
0.995	4090	0.925
0.999	4134	0.919

Conclusion

When I learned about the multi-armed bandit problem years ago, I was told that it had been solved, and that the beta-binomial was the best possible solution. Then I learned about other algorithms.

I wrote this paper to compare the algorithms and to add my obvious algorithm: the fixed-size learning pattern. I didn't expect the fixed-size learning pattern to come out on top in this comparison.

An obvious question is: For the algorithms that have parameters, what are the optimized parameters? My guess is that the parameters will depend strongly on the input and how you measure success.

The most important future work to do is to speed up these algorithms. The beta-binomial and the Thompson sampling methods both give excellent results, but they take the longest to do computation.

Thanks

I would like to thank @Zoheb@mathstodon.xyz and @pruby@mathstodon.xyz of Mastodon for commenting the first draft of this article.

Final request

I am actively looking for work, and I'm using these articles to spread my name. If this article has educated you, then could you please forward my LinkedIn profile at https://www.linkedin.com/in/brenteedwards/ to your hiring manager?