Multi-Armed Bandits

Posted by **Cameron Davidson-Pilon** at **Apr 06, 2013**

Preface: This example is a (greatly modified) excerpt from the open-source book *Bayesian Methods for Hackers*, currently being developed on Github

Adapted from an example by Ted Dunning of MapR Technologies

The Multi-Armed Bandit Problem

> *Suppose you are faced with N slot machines (colourfully called multi-armed bandits). Each bandit has an unknown probability of distributing a prize (assume for now the prizes are the same for each bandit, only the probabilities differ). Some bandits are very generous, others not so much. Of course, you don't know what these probabilities are. By only choosing*

> *one bandit per round, our task is devise a strategy to maximize our winnings.*

Of course, if we knew the bandit with the *largest* probability, then always picking this bandit would yield the maximum winnings. So our task can be phrased as "Find the *best* bandit, and as quickly as possible".

The task is complicated by the stochastic nature of the bandits. A suboptimal bandit can return many winnings, purely by chance, which would make us believe that it is a very profitable bandit. Similarly, the best bandit can return many duds. Should we keep trying losers then, or give up?

A more troublesome problem is, if we have a found a bandit that returns *pretty good* results, do we keep drawing from it to maintain our *pretty good score*, or do we try other bandits in hopes of finding an even-better bandit? This is the *exploration vs. exploitation* dilemma.

Applications

The Multi-Armed Bandit problem at first seems very artificial, something only a mathematician would love, but that is only before we address some applications:

- Internet display advertising: companies have a suite of potential ads they can display to visitors, but the company is not sure which ad strategy to follow to maximize sales. This is similar to A/B testing, but has the added advantage of naturally minimizing strategies that do not work (and generalizes to A/B/C/D... strategies)
- Ecology: animals have a finite amount of energy to expend, and following certain behaviours has uncertain rewards. How does the animal maximize its fitness?
- Finance: which stock option gives the highest return, under time-varying return profiles.
- Clinical trials: a researcher would like to find the best treatment, out of many possible treatments, while minimizing losses.

Many of these questions above are fundamental to the application's field.

It turns out the *optimal* solution is incredibly difficult, and it took decades for an overall solution to develop. There are also many approximately-optimal solutions which are quite good. The one I wish to discuss is one of the few solutions that can scale incredibly well. The solution is known as *Bayesian Bandits*.

A Proposed Solution

Any proposed strategy is called an *online algorithm* (not in the internet sense, but in the continuously-being-updated sense), and more specifically a *reinforcement learning algorithm*. The algorithm starts in an ignorant state, where it knows nothing, and begins to acquire data by testing the system. As it acquires data and results, it learns what the best and worst behaviours are (in this case, it learns which bandit is the best). With this in mind, perhaps we can add an additional application of the Multi-Armed Bandit problem:

- Psychology: how does punishment and reward effect our behaviour? How do humans' learn?

The Bayesian solution begins by assuming priors on the probability of winning for each bandit. In our vignette we assumed complete ignorance of the these probabilities. So a very natural prior is the flat prior over 0 to 1. The algorithm proceeds as follows:

For each round,

1. Sample a random variable X_b from the prior of bandit b, for all b.
2. Select the bandit with largest sample, i.e. select bandit $B = \text{argmax } X_b$.
3. Observe the result of pulling bandit B, and update your prior on bandit B.
4. Return to 1.

That's it. Computationally, the algorithm involves sampling from N distributions. Since the initial priors are $\text{Beta}(\alpha = 1, \beta = 1)$ (a uniform distribution), and the observed result X (a win or loss, encoded 1 and 0 respectfully) is Binomial, the posterior is a $\text{Beta}(\alpha = 1 + X, \beta = 1 + 1 - X)$ (see here for why to is true).

To answer a question from before, this algorithm suggests that we should not discard losers, but we should pick them at a decreasing rate as we gather confidence that there exist *better* bandits. This follows because there is always a non-zero chance that a loser will achieve the status of B, but the probability of this event decreases as we play more rounds (see figure below).

Below is an implementation of the Bayesian Bandits strategy (which can be skipped for the less Pythonic-ly interested).

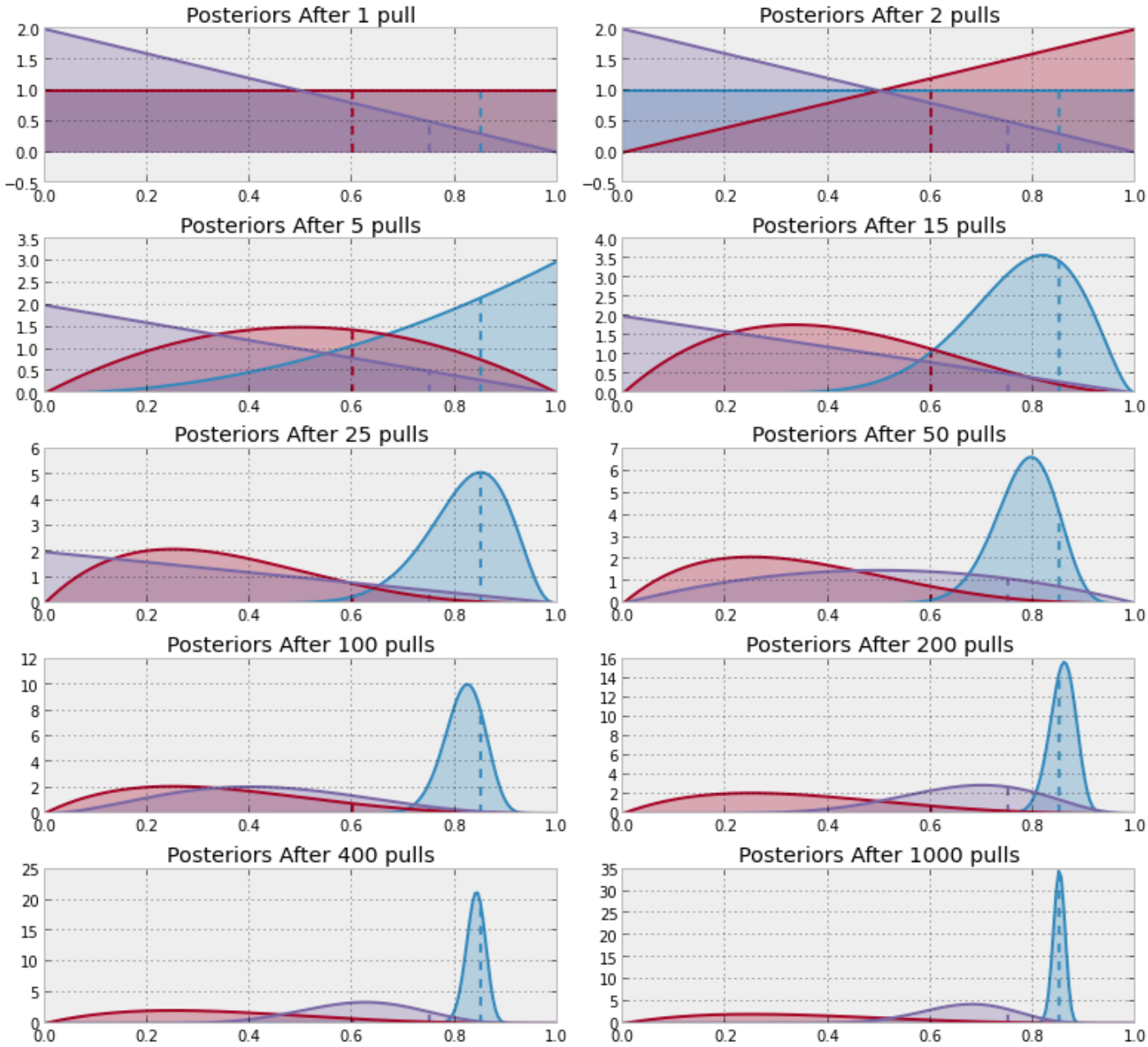
```
1  from pymc import rbeta
2
3  rand = np.random.rand
4
5  class Bandits(object):
6      """
7      This class represents N bandits machines.
```

```
    parameters:
        p_array: a (n,) Numpy array of probabilities >0, <1.

    methods:
        pull( i ): return the results, 0 or 1, of pulling
                   the ith bandit.
    """
    def __init__(self, p_array):
        self.p = p_array
        self.optimal = np.argmax(p_array)

    def pull( self, i ):
        #i is which arm to pull
        return rand() < self.p[i]

    def __len__(self):
        return len(self.p)


class BayesianStrategy( object ):
    """
    Implements a online, learning strategy to solve
    the Multi-Armed Bandit problem.

    parameters:
        bandits: a Bandit class with .pull method

    methods:
        sample_bandits(n): sample and train on n pulls.

    attributes:
        N: the cumulative number of samples
        choices: the historical choices as a (N,) array
        bb_score: the historical score as a (N,) array

    """

    def __init__(self, bandits):

        self.bandits = bandits
        n_bandits = len( self.bandits )
        self.wins = np.zeros( n_bandits )
        self.trials = np.zeros(n_bandits )
        self.N = 0
        self.choices = []
```

```
53
54          self.bb_score = []
55
56
57      def sample_bandits( self, n=1 ):
58
59          bb_score = np.zeros( n )
60          choices = np.zeros( n )
61
62          for k in range(n):
63              #sample from the bandits's priors, and select the largest sample
64              choice = np.argmax( rbeta( 1 + self.wins, 1 + self.trials - self.wins) )
65
66              #sample the chosen bandit
67              result = self.bandits.pull( choice )
68
69              #update priors and score
70              self.wins[ choice ] += result
71              self.trials[ choice ] += 1
72              bb_score[ k ] = result
73              self.N += 1
74              choices[ k ] = choice
75
76          self.bb_score = np.r_[ self.bb_score, bb_score ]
77          self.choices = np.r_[ self.choices, choices ]
78          return
```

bandits.py hosted with ♥ by **GitHub** **view raw**

Below we present a visualization of the algorithm sequentially learning the solution. In the figure below, the dashed lines represent the true hidden probabilities, which are `[0.85, 0.60, 0.75]` (this can be extended to many more dimensions, but the figure suffers, so I kept it at 3).



Note that we don't real care how accurate we become about inference of the hidden probabilities -- for this problem we are more interested in choosing the best bandit (or more accurately, becoming *more confident* in choosing the best bandit). For this reason, the distribution of the red bandit is very wide (representing ignorance about what that hidden probability might be) but we are reasonably confident that it is not the best, so the algorithm chooses to ignore it.

Below is a D3 demonstration of this updating/learning process. The challenge is only three arms, each with a small probability (randomly generated). The first figure is the pull/reward count, and the second figure contains the posterior distributions of each bandit.

Arm 1 Arm 2 Arm 3

Run Bayesian Bandits | Reveal probabilities

Reward/Pull Ratio	Pulls	Rewards
0	0	0

Deviations of the observed ratio from the highest probability is a measure of performance. For example, in the long run, optimally we can attain the reward/pull ratio of the maximum bandit probability. Long-term realized ratios *less* than the maximum represent inefficiencies. (Realized ratios *larger than the maximum probability is due to randomness, and will eventually fall below).*

A Measure of *Good*

We need a metric to calculate how well a strategy is doing. Recall the absolute *best* we can do is to always pick the bandit with the largest probability of winning. Denote this best bandit's probability of w^*. Our score should be relative to how well we would have done had we chosen the best bandit from the beginning. This motivates the *total regret of a strategy, defined:*

$$\begin{aligned} R_T &= \sum_{i=1}^{T} \left(w^* - w_{B(i)} \right) \\ &= Tw^* - \sum_{i=1}^{T} w_{B(i)} \end{aligned}$$

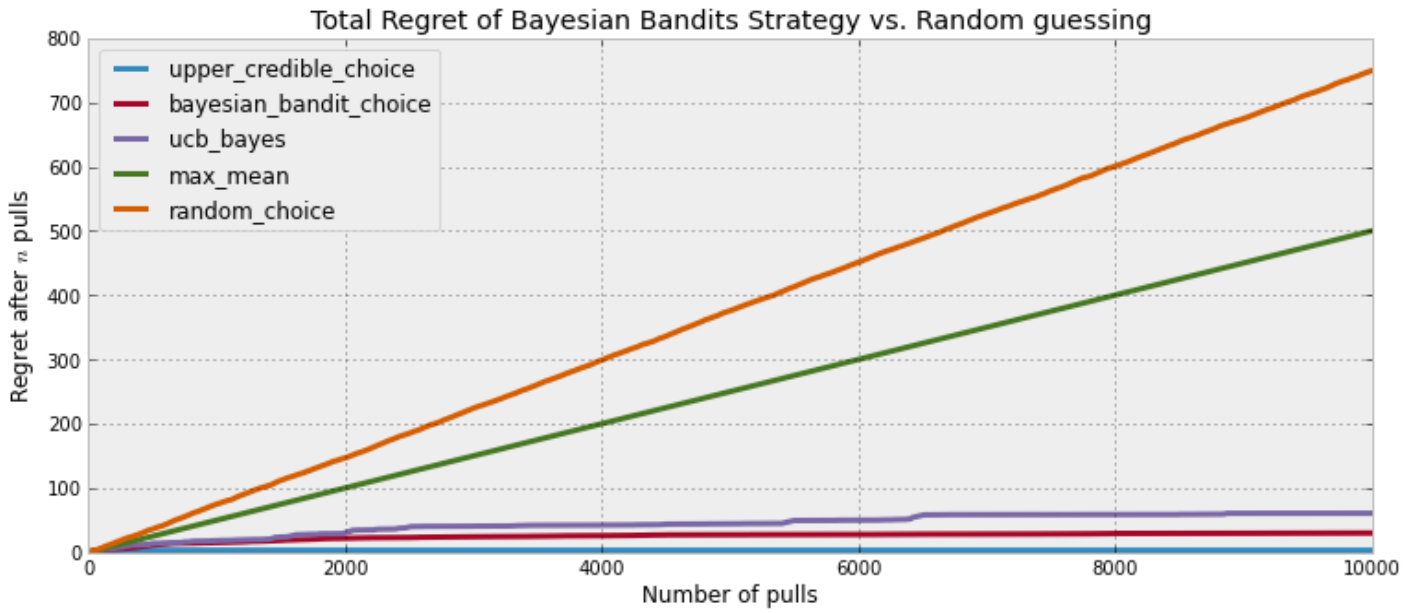
where $w_{B(i)}$ is the probability of a prize of the chosen bandit in the ith round. A total regret of 0 means the strategy is matching the best possible score. This is likely not possible, as initially our algorithm will make the wrong choice. Ideally, a strategy's total regret should flatten as it *learns* the best bandit. (Mathematically we achieve $w_{B(i)} = w^*$ often)

Below we plot the total regret of this simulation, including the scores of some other strategies:

1. Random: randomly choose a bandit to pull. If you can't beat this, just stop.
2. largest Bayesian credible bound: pick the bandit with the largest upper bound in its 95% credible region of the underlying probability.
3. Bayes-UCB algorithm: pick the bandit with the largest *score*, where score is a dynamic

quantile of the posterior (see [4])

4. Mean of posterior: choose the bandit with the largest posterior mean. This is what a human player (sans computer) would likely do.
5. Largest proportion: pick the bandit with the current largest observed proportion of winning.



To be more scientific so as to remove any possible luck in the above simulation, we should instead look at the *expected total regret*:

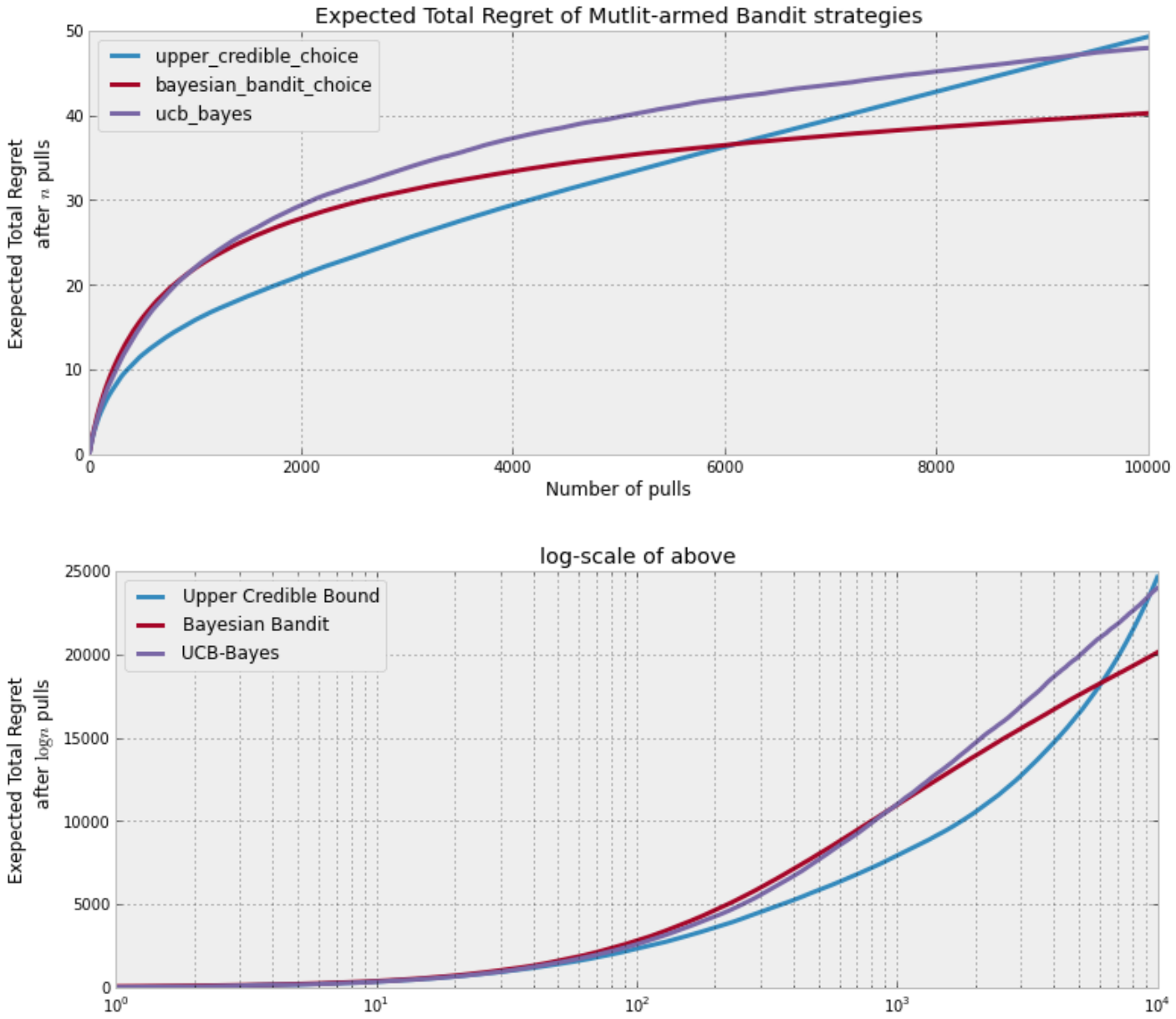
\bar{R_T} = E[R_T]

It can be shown that any *sub-optimal* strategy's expected total regret is bounded below logarithmically. Formally,

E[R_T] = \Omega \left(\;\log(T)\; \right)

Thus, any strategy that matches logarithmic-growing regret is said to "solve" the Multi-Armed Bandit problem [1].

Using the Law of Large Numbers, we can approximate Bayesian Bandit's expected total regret by performing the same experiment many times (25 times, to be fair):



Extending the algorithm

Because of the algorithm's simplicity, it is easy to extend. Some possibilities:

1. If interested in the *minimum* probability (eg: where prizes are are bad thing), simply choose $B = \text{argmin } X_b$ and proceed.

2. Adding learning rates: Suppose the underlying environment may change over time. Technically the standard Bayesian Bandit algorithm would self-update itself (awesome) by learning that what it thought was the best is starting to fail more often. We can also encourage the algorithm to learn changing environments quicker. We simply need to add a *rate* term upon updating:

```
self.wins[ choice ] = rate*self.wins[ choice ] + result
self.trials[ choice ] = rate*self.trials[ choice ] + 1
```

If `rate < 1`, the algorithm will *forget* its previous results quicker and there will be a downward pressure towards ignorance. Conversely, setting `rate > 1` implies your algorithm will act more risky, and bet on earlier winners more often and be more resistant to changing environments.

1. Hierarchical algorithms: We can setup a Bayesian Bandit algorithm on top of smaller bandit algorithms. Suppose we have N Bayesian Bandit models, each varying in some behavior (for example different `rate` parameters, representing varying sensitivity to changing environments). On top of these N models is another Bayesian Bandit learner that will select a sub-Bayesian Bandit. This chosen Bayesian Bandit will then make an internal choice as to which machine to pull. The super-Bayesian Bandit updates itself depending on whether the sub-Bayesian Bandit was correct or not.
2. Extending the rewards, denoted y_a for bandit a, to random variables from a distribution $f_{y_a}(y)$ is straightforward. More generally, this problem can be rephrased as "Find the bandit with the largest expected value", as playing the bandit with the largest expected value is optimal. In the case above, f_{y_a} was Bernoulli with probability p_a, hence the expected value for an bandit is equal to p_a, which is why it looks like we are aiming to maximize the probability of winning. If f is not Bernoulli, and it is non-negative, which can be accomplished apriori by shifting the distribution (we assume we know f), then the algorithm behaves as before: For each round,

1. Sample a random variable X_b from the prior of bandit b, for all b.
2. Select the bandit with largest sample, i.e. select bandit $B = \text{argmax } X_b$.
3. Observe the result, $R \sim f_{y_a}$, of pulling bandit B, and update your prior on bandit B.
4. Return to 1.

The issue is in the sampling of X_b drawing phase. With Beta priors and Bernoulli observations, we have a Beta posterior -- this is easy to sample from. But now, with arbitrary distributions f, we have a non-trivial posterior. Sampling from these can be difficult. For the special case when the rewards distributions are confined to [0,1], [5] provides a general algorithm:

1. Sample a random variable X_b from the prior of bandit b, for all b.
2. Select the bandit with largest sample, i.e. select bandit $B = \text{argmax } X_b$.
3. Observe the result, $R \sim f_{y_a}$, $R \in [0, 1]$, of pulling bandit B.
4. **Perform a Bernoulli trial with success probability R and observe output r.**

5. Update posterior B with observation r.
6. Return to 1.

Commenting systems

There has been some interest in extending the Bayesian Bandit algorithm to commenting systems. Recall in a previous post, we developed a ranking algorithm based on the Bayesian lower-bound of the proportion of upvotes to total total votes. One problem with this approach is that it will bias the top rankings towards older comments, since older comments naturally have more votes (and hence the lower-bound is tighter to the true proportion). This creates a positive feedback cycle where older comments gain more votes, hence are displayed more often, hence gain more votes, etc. This pushes any new, potentially better comments, towards the bottom. J. Neufeld proposes a system to remedy this that uses a Bayesian Bandit solution.

His proposal is to consider each comment as a Bandit, with a the number of pulls equal to the number of votes cast, and number of rewards as the number of upvotes, hence creating a $\text{Beta}(1+U, 1+D)$ posterior. As visitors visit the page, samples are drawn from each bandit/comment, but instead of displaying the comment with the $\max$ sample, the comments are ranked according the the ranking of their respective samples. From J. Neufeld's blog [7]:

> *[The] resulting ranking algorithm is quite straightforward, each new time the comments page is loaded, the score for each comment is sampled from a $\text{Beta}(1+U, 1+D)$, comments are then ranked by this score in descending order... This randomization has a unique benefit in that even untouched comments $(U=1, D=0)$ have some chance of being seen even in threads with 5000+ comments (something that is not happening now), but, at the same time, the user will is not likely to be inundated with rating these new comments.*

Conclusion

The Bayesian Bandit solution is a novel approach to a classic AI problem. It also scales really well as the number of possibilities increases. There has been some interesting work on using Bayesian Bandits with independent variables in online advertising, funded mostly by Google

and Yahoo (papers cited below).

Continue the discussion at Hacker News and Reddit.

References

1. Kuleshov, Volodymyr, and Doina Precup. "Algorithms for the multi-armed bandit problem." Journal of Machine Learning Research. (2000): 1-49. Print.
2. Github *et. al.* "Probabilistic Programming and Bayesian Methods for Hackers: Using Python and PyMC" (2013)
3. Scott, Steven L. "A modern Bayesian look at the multi-armed bandit." Applied Stochastic Models in Business and Industry. 26. (2010): 639–658. Print.
4. Dunning, Ted. "Bayesian Bandits." Surprise and coincidence - Musicing from the Long Tail. Blogger, 18 Feb 2012. Web. Web. 6 Apr. 2013. .
5. Agrawal, Shipra, and Goyal Navin. "Analysis of Thompson Sampling for the multi-armed bandit problem." (2012): n. page. Web. 15 Apr. 2013. .

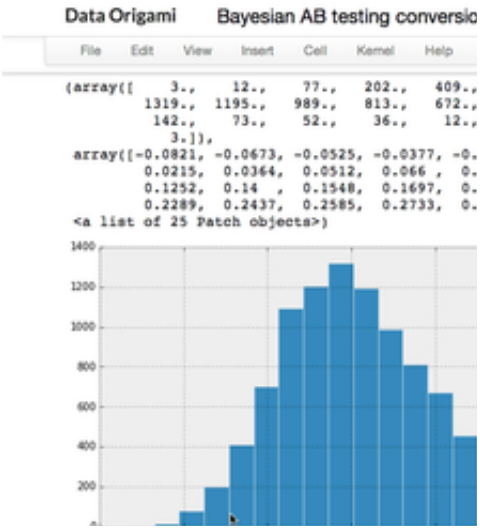
Tags: camdp.com

Other Articles you might like

Feature Space in Machine Learning

Feature space refers to the n-dimensions where your variables live (not including a target variable, if it is present). The term is used often in ML literature bec...

Latest Data Science screencasts available



Comments

Leave a comment

Name

Email

Message

Please note: *comments will be approved before they are published*

Post Comment

Recent Articles

"Reversing the Python Data Analysis Lens" Video

Dec 29, 2015

Bayesian Methods for Hackers release!
Nov 16, 2015

[Video] Mistakes I've Made talk at PyData 2015
Aug 30, 2015

How can I use non-constructive proofs in data analysis?
Aug 09, 2015

All Articles

Categories

A/B testing

bayesian

big-data

camdp.com

categorical

data

data collection

data science

evolution

fivethirtyeight

machine learning

mathematics

matplotlib

pandas

pymc2

pyspark

python

spark

statistics

About Data Origami

Data Origami Blog

FAQ

Search

About the Author

Cameron Davidson-Pilon is the author of the open-source book Bayesian Methods for Hackers, and the survival analysis library lifelines. Currently doing analytics at Shopify.


Sign up for Video and Blog updates

Email Address Sign Up