Probabilistic Programming

Marius Popescu

popescunmarius@gmail.com

2020 – 2021

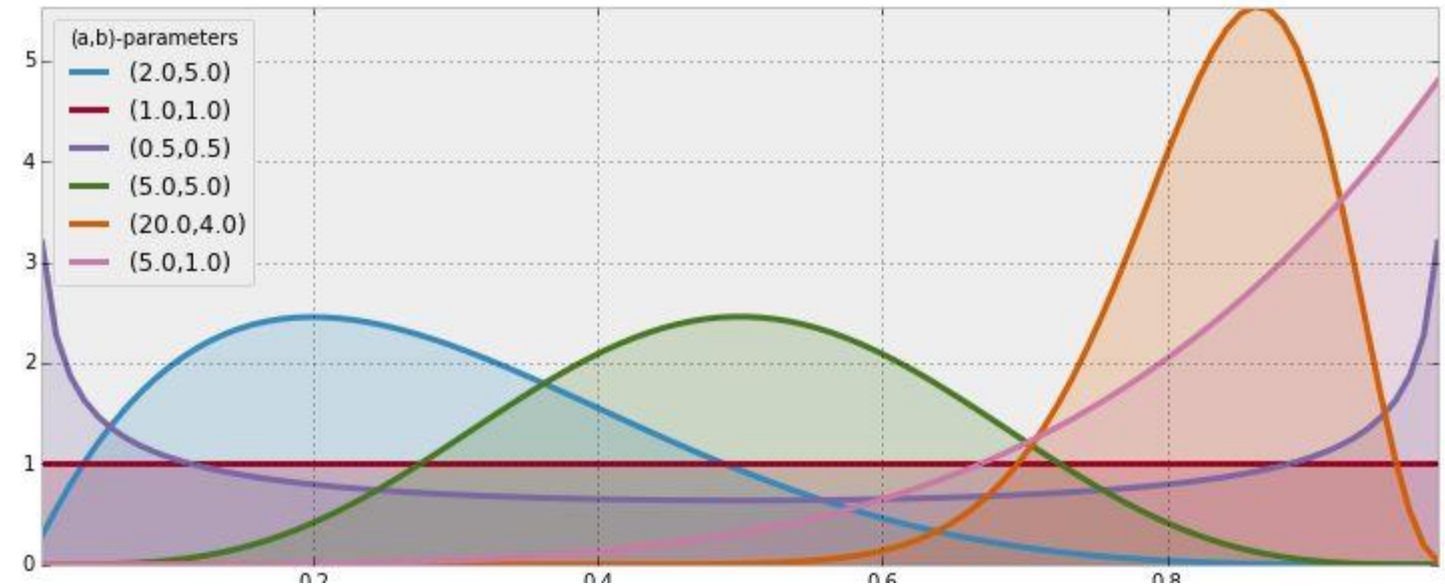# Avoiding MCMC

Conjugate Priors

Beta Distribution

$$Z \sim \text{Beta}(\alpha, \beta)$$

$$f_Z(z|\alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} z^{\alpha-1}(1-z)^{\beta-1}, \ \alpha > 0, \beta > 0, 0 < z < 1$$

$$E(Z|\alpha, \beta) = \int_{-\infty}^{\infty} z f_Z(z|\alpha, \beta) dz = \frac{\alpha}{\alpha + \beta}$$

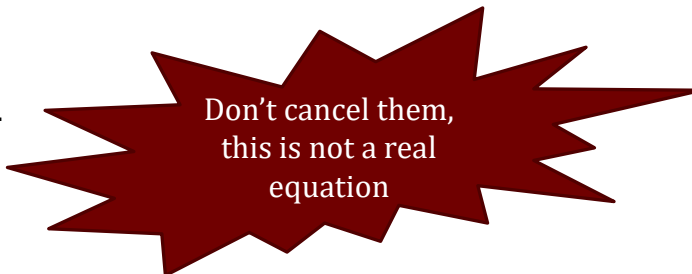```
Z=pymc.Beta("Z", alpha, beta)
```

# Beta as Prior

There is an interesting connection between the Beta distribution and the Binomial distribution. Suppose we are interested in some unknown proportion or probability $p$. We assign a $\text{Beta}(\alpha, \beta)$ prior to $p$. We observe some data generated by a Binomial process, say $X \sim \text{Binomial}(N, p)$ with $p$ still unknown. Then our *posterior is again a Beta distribution,* i.e. $p|X \sim \text{Beta}(\alpha + X, \beta + N - X)$. Succinctly, one can relate the two by "a Beta prior with Binomial observations creates a Beta posterior". This is a very useful property, both computationally and heuristically.

if we start with a $\text{Beta}(1,1)$ prior on $p$ (which is a Uniform), observe data $X \sim \text{Binomial}(N, p)$, then our posterior is $\text{Beta}(1 + X, 1 + N - X)$.

# Conjugate Priors

$$\overbrace{\text{Beta}}^{\text{prior}} \cdot \overbrace{\text{Binomial}}^{\text{data likelihood}} = \overbrace{\text{Beta}}^{\text{posterior}}$$

Don't cancel them, this is not a real equation

Notice the Beta on both sides of this equation

A prior $p$ that satisfies this relationship is called a *conjugate prior*.

Suppose $X$ comes from, or is believed to come from, a well-known distribution, call it $f_\alpha$, where $\alpha$ are possibly unknown parameters of $f$. $f$ could be a Normal distribution, or Binomial distribution, etc. For particular distributions $f_\alpha$, there may exist a prior distribution $p_\beta$, such that:

$$p_\beta \cdot f_\alpha = p_{\beta'}$$

where $\beta'$ is a different set of parameters but $p$ is the same distribution as the prior

# Conjugate Distributions

### Discrete Likelihood

| Prior | Likelihood | Posterior |
|---|---|---|
| Beta | Bernoulli | Beta |
| Beta | Binomial | Beta |
| Gamma | Poisson | Gamma |
| Dirichlet | Categorical | Dirichlet |
| Dirichlet | Multinomial | Dirichlet |

### Continuous Likelihood

| Prior | Likelihood | Posterior |
|---|---|---|
| Normal-gamma | Normal | Normal-gamma |
| Gamma | Pareto | Gamma |
| Gamma | Exponential | Gamma |
| Pareto | Uniform | Pareto |
| normal-Wishart | Multivariate normal | normal-Wishart |

https://en.wikipedia.org/wiki/Conjugate_prior

# Conjugate Priors

## Pros

- mathematical convenience: it is simple to go from prior to posterior

- useful computationally: it allows us to avoid using MCMC, since the posterior is known in closed form, hence inference and analytics are easy to derive
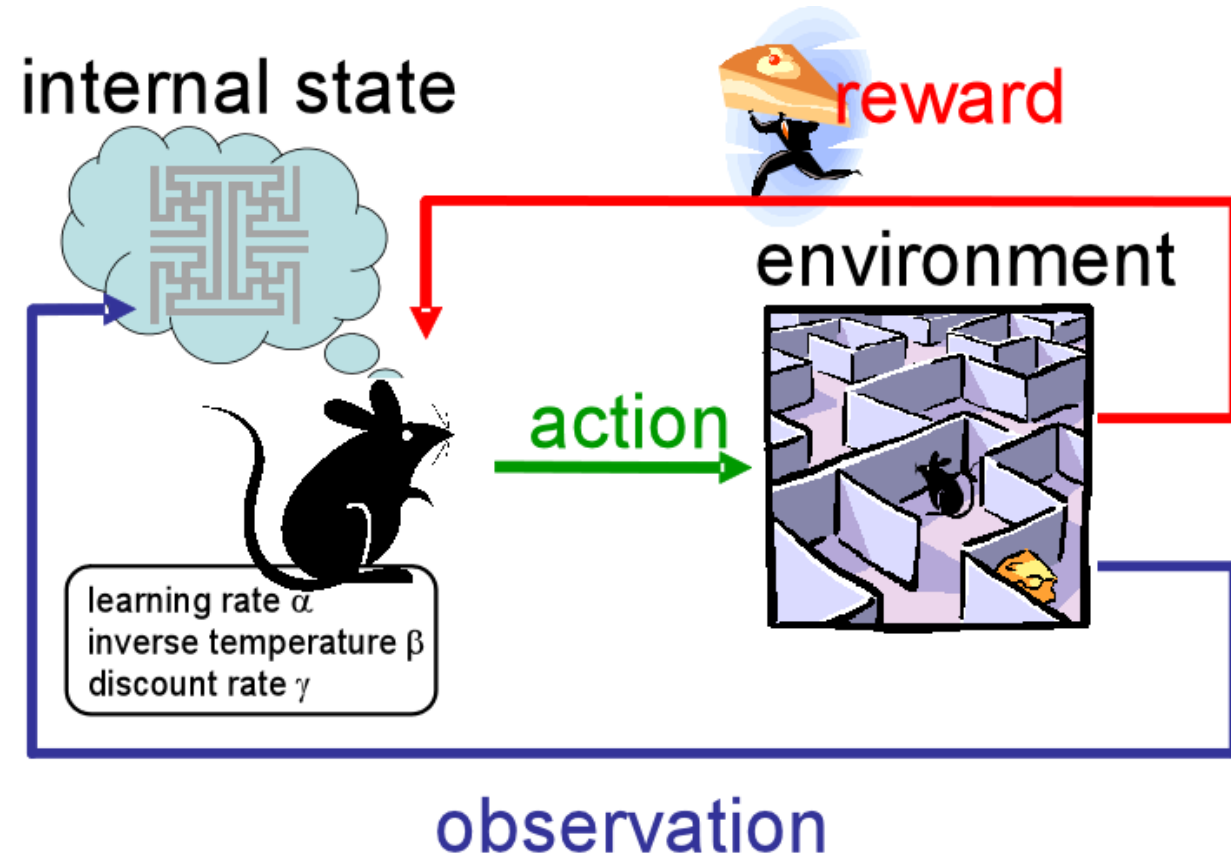
## Cons

- is not objective: It is not guaranteed that the conjugate prior can accommodate the practitioner's subjective opinion

- there typically exist conjugate priors for simple, one dimensional problems

# Example: Bayesian Multi-Armed Bandits

# Reinforcement Learning

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards.

The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that evaluates the actions taken rather than instructs by giving correct actions.

# N-Armed Bandit Problem

A special case of the reinforcement learning problem in which there is only a single state, a simplified setting, one that does not involve learning to act in more than one situation.

Consider the following learning problem. You are faced repeatedly with a choice among N different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your objective is to maximize the expected total reward over some time period, for example, over 1000 action selections, or time steps.

This is the original form of the N-armed bandit problem, so named by analogy to a slot machine, or "one-armed bandit," except that it has N levers instead of one.

# 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

- 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 treatment, while minimizing losses

# Binomial Bandit

We assume that the prizes are the same for each bandit, only the probabilities differ. Each bandit has an unknown probability of distributing the prize. 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.

# Binomial Bandit

It turns out the optimal solution is incredibly difficult, but there are also many approximately-optimal solutions which are quite good.

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).

# Randomized Probability Matching

○ Randomized probability matching is a particularly appealing heuristic that plays each arm in proportion to its probability of being optimal

○ Randomized probability matching is not known to optimize any specific utility function, but it is easy to apply in general settings, and balances exploration and exploitation in a natural way

Thompson WR. (1935). On the theory of apportionment. American Journal of Mathematics, 57(2), 450–456

Scott, S. L. (2010). A modern Bayesian look at the multi-armed bandit. Applied Stochastic Models in Business and Industry, 26(6), 639-658.

# Randomized Probability Matching

The Bayesian solution begins by assuming priors on the probability of winning for each bandit. We assumed complete ignorance of these probabilities. So a very natural prior is the flat prior over 0 to 1.

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 $B = \text{argmax } X_b$.

3. Observe the result of pulling bandit $B$, and update your prior on bandit $B$.

4. Return to 1.

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 $\text{Beta}(\alpha = 1 + X, \beta = 1 + 1 - X)$

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).

# Bandits

```python
class Bandits(object):

    """
    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 np.random.rand() < self.p[i]

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

BayesianStrategy

```python
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

    """
```

BayesianStrategy

```python
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 = []
    self.bb_score = []
```

BayesianStrategy

```python
def sample_bandits(self, n=1):

    bb_score = np.zeros(n)
    choices = np.zeros(n)

    for k in range(n):
        # sample from the bandits's priors, and select the largest sample
        choice = np.argmax(rbeta(1 + self.wins, 1 + self.trials - self.wins))

        # sample the chosen bandit
        result = self.bandits.pull(choice)

        # update priors and score
        self.wins[choice] += result
        self.trials[choice] += 1
        bb_score[k] = result
        self.N += 1
        choices[k] = choice

    self.bb_score = np.r_[self.bb_score, bb_score]
    self.choices = np.r_[self.choices, choices]
    return
```

Plotting

```python
hidden_prob = np.array([0.85, 0.60, 0.75])
bandits = Bandits(hidden_prob)
bayesian_strat = BayesianStrategy(bandits)
draw_samples = [1, 1, 3, 10, 10, 25, 50, 100, 200, 600]

for j, i in enumerate(draw_samples):
    plt.subplot(5, 2, j + 1)
    bayesian_strat.sample_bandits(i)
    plot_priors(bayesian_strat, hidden_prob)
    plt.autoscale(tight=True)
plt.tight_layout()
plt.show()
```
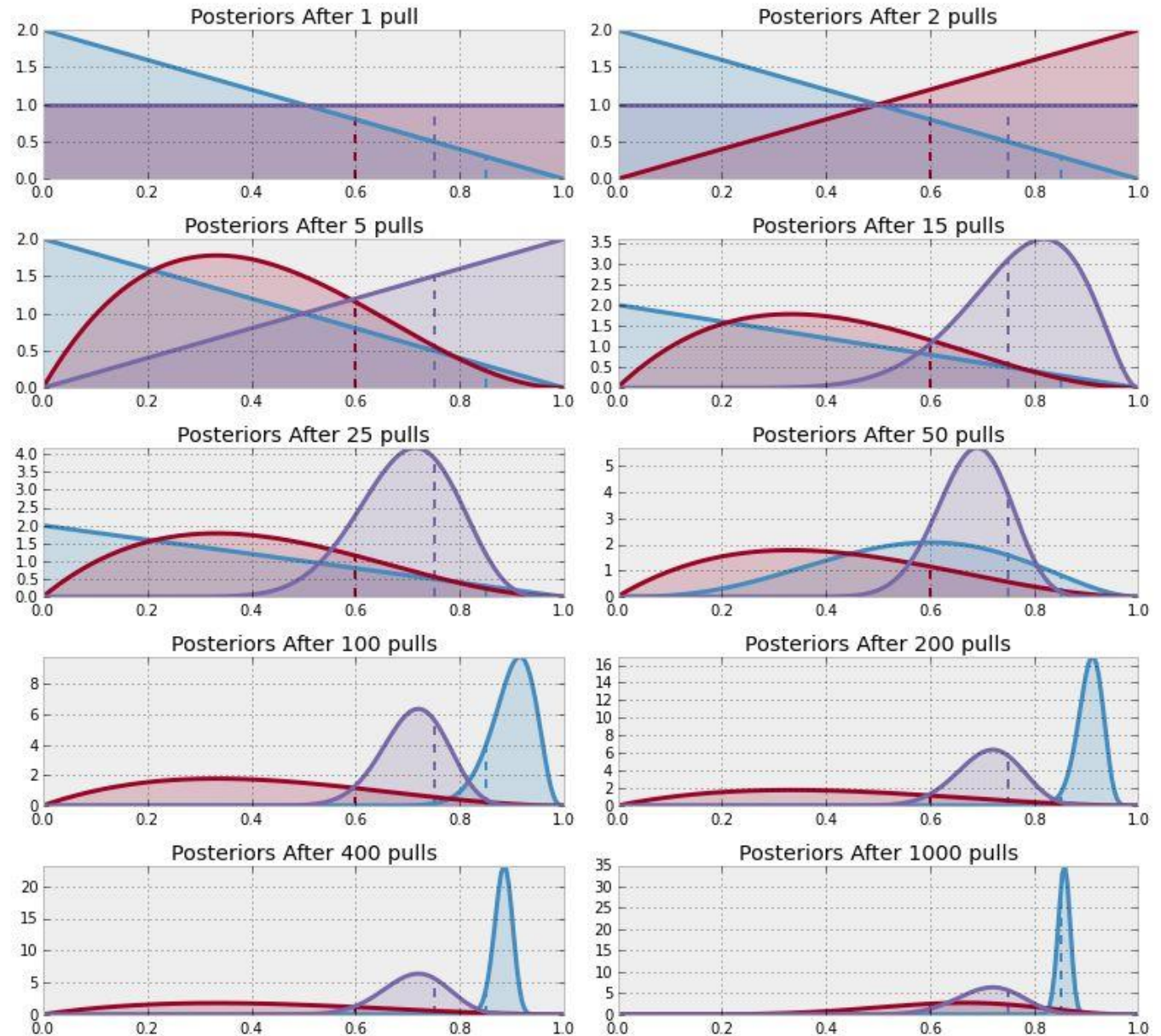
```python
beta = stats.beta
x = np.linspace(0.001, .999, 200)

def plot_priors(bayesian_strategy, prob, lw=3, alpha=0.2, plt_vlines=True):
    # plotting function
    wins = bayesian_strategy.wins
    trials = bayesian_strategy.trials
    for i in range(prob.shape[0]):
        y = beta(1 + wins[i], 1 + trials[i] - wins[i])
        p = plt.plot(x, y.pdf(x), lw=lw)
        c = p[0].get_markeredgecolor()
        plt.fill_between(x, y.pdf(x), 0, color=c, alpha=alpha,
                         label="underlying probability: %.2f" % prob[i])
        if plt_vlines:
            plt.vlines(prob[i], 0, y.pdf(prob[i]),
                       colors=c, linestyles="--", lw=2)
        plt.autoscale(tight="True")
    plt.title("Posteriors After %d pull" % bayesian_strategy.N +
              "s" * (bayesian_strategy.N > 1))
    plt.autoscale(tight=True)
    return
```

Results

Posterior distributions of our inference about each bandit after different numbers of pulls

Note that we don't really care how accurate we become about the 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.

A Measure of Good

We need a metric to calculate how well we are doing. 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 as:

$$R_T = \sum (w^* - w_{B(i)}) = Tw^* - \sum w_{B(i)}$$

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 attaining the best possible score. This is likely not possible, as initially our algorithm will often 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)

Total Regret of Bayesian Bandits Strategy
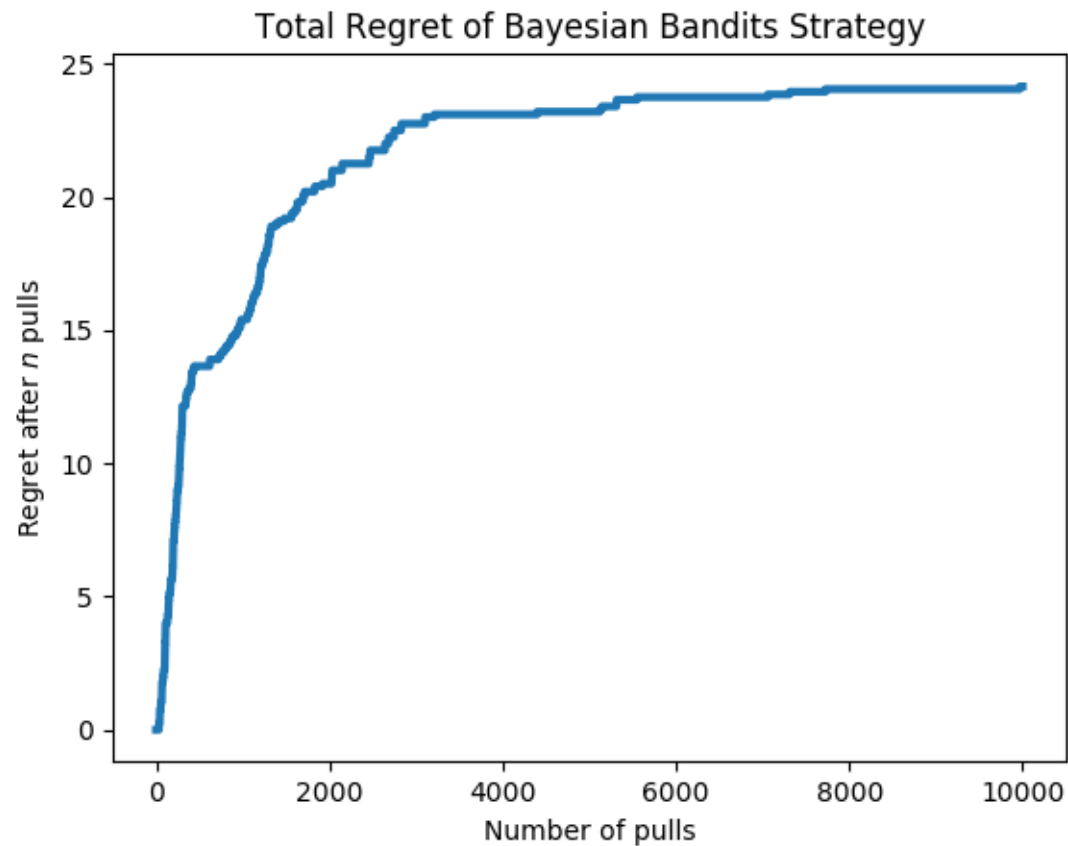
Other Strategies

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

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.

Total Regret of Bayesian Bandits Strategy vs. Other Strategies
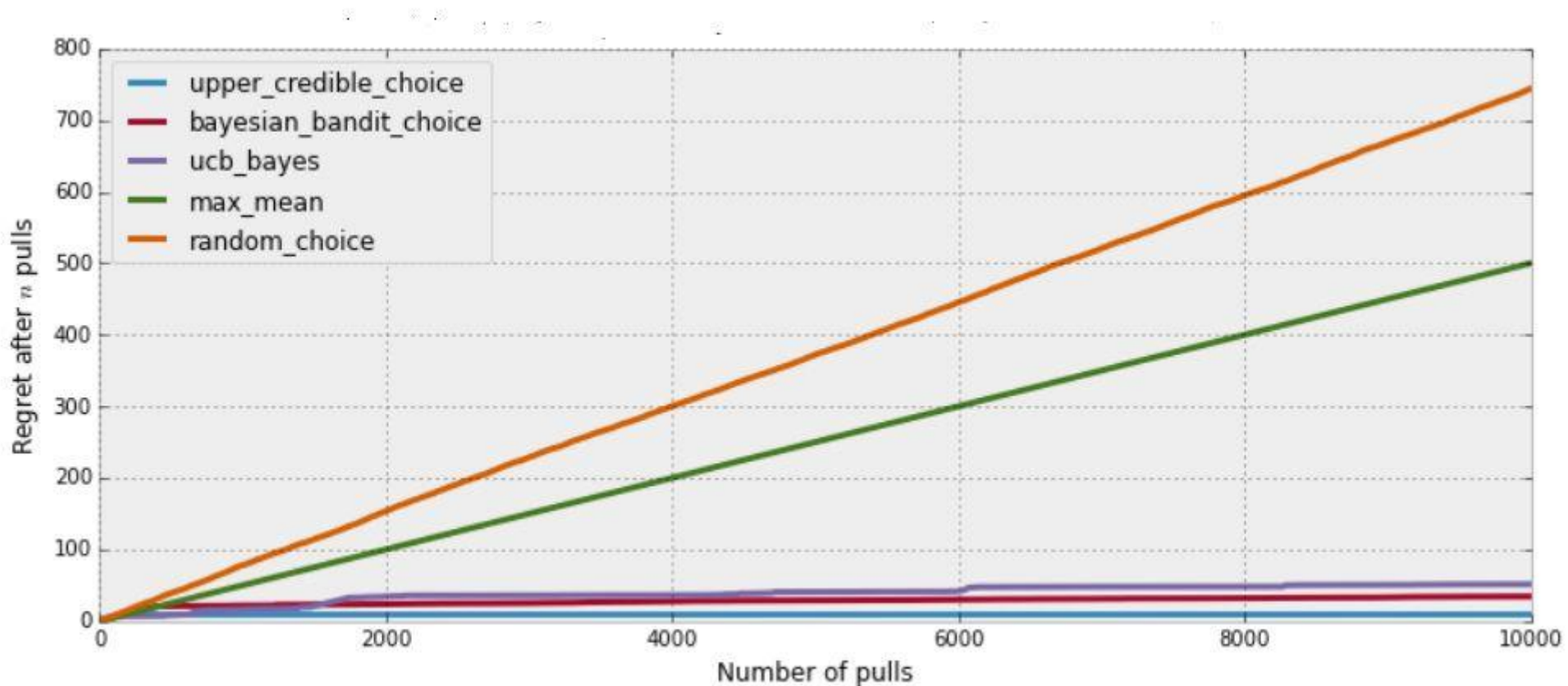
Expected Total Regret

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:

$$E[R_T] = \Omega(\log(T))$$

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

Using the Law of Large Numbers, we can approximate Bayesian Bandit's expected total regret by performing the same experiment many times (500 times, to be fair).
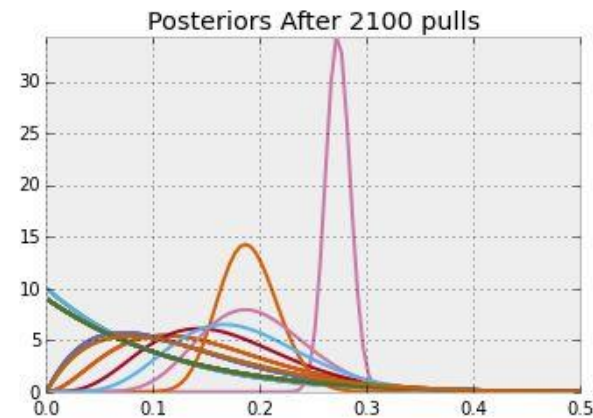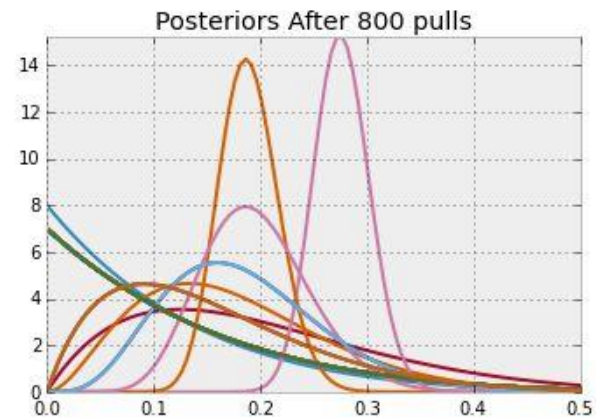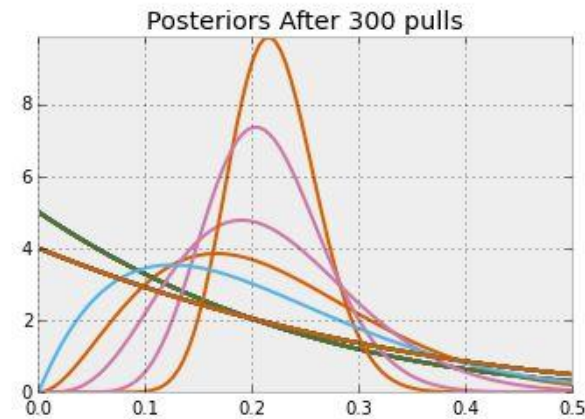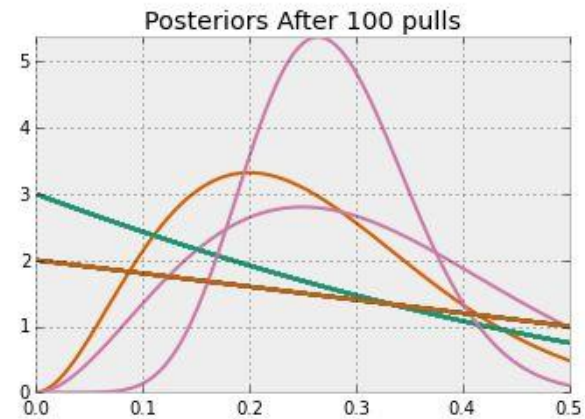
Expected Total Regret of Multi-armed Bandit Strategies
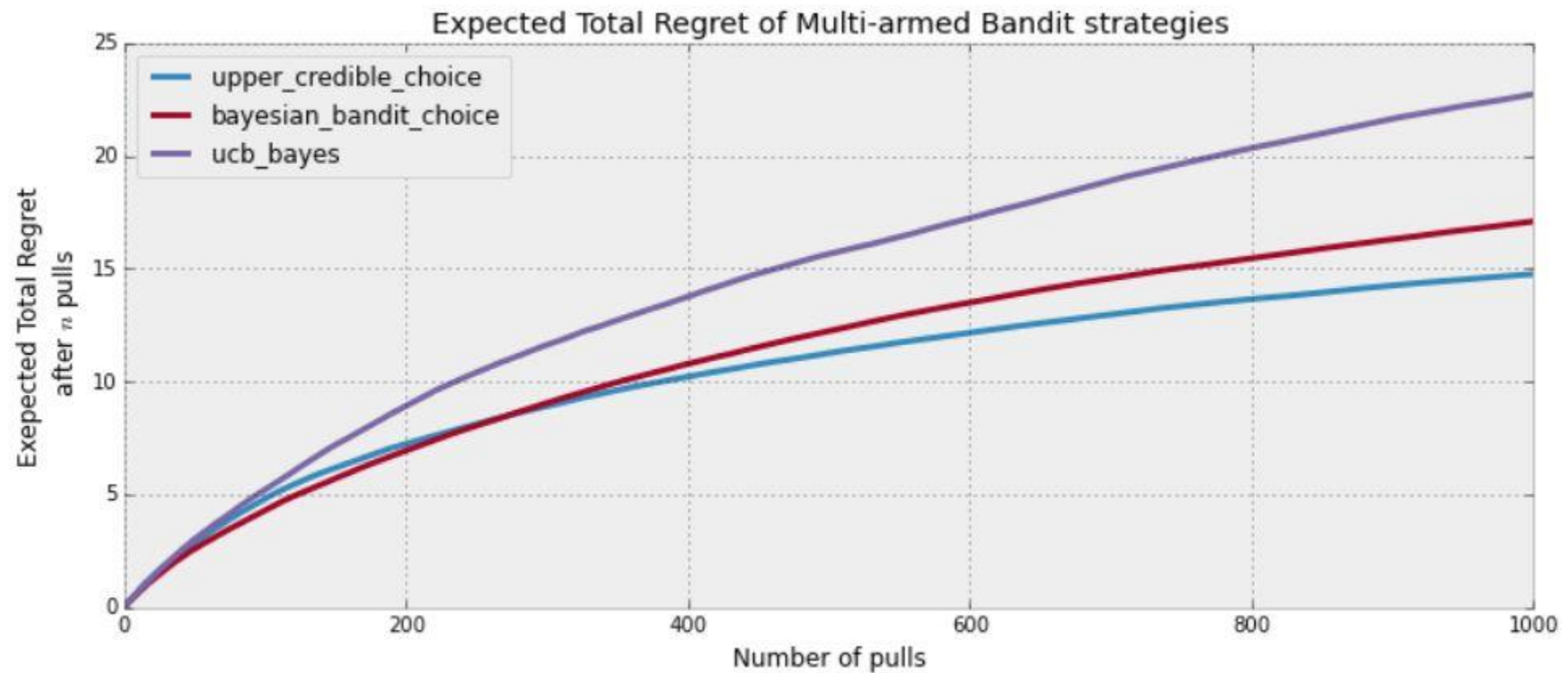
Bayesian Bandit algorithm learning 35 different options

[0.0431 0.0745 0.1187 0.0098 0.0945 0.0438 0.0442 0.0059 0.0749
  0.023   0.0543 0.025   0.1231 0.0148 0.0164 0.2688 0.0073 0.0564
  0.0031 0.0698 0.0478 0.1657 0.0091 0.0384 0.2236 0.1548 0.0562
  0.0209 0.024   0.0197 0.0788 0.0572 0.1207 0.0405 0.0679]

Extending the Algorithm

Adding learning rates: Suppose the underlying environment may change over time. Technically the standard Bayesian Bandit algorithm would self-update itself (awesome) by noting that what it thought was the best is starting to fail more often. We can motivate the algorithm to learn changing environments quicker by simply adding 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 wins 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.

Extending the Algorithm

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.

Extending the Algorithm

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 preceding case f_{y_a} was Bernoulli with probability p_a, hence the expected value for a 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 a priori 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 $B = \text{argmax } X_b$.

3. Observe the result $R \sim f_{y_a}$ of pulling bandit B, and update your prior on bandit B.

4. Return to 1.

Back to MCMC?

The issue is in the sampling of the 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.

Extending the Algorithm

There has been some interest in extending the Bayesian Bandit algorithm to commenting systems. Recall the example "How to order Reddit submissions" in Chapter 4, where a ranking algorithm was developed based on the Bayesian lower-bound of the proportion of upvotes to the total number of 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:

http://simplemlhacks.blogspot.com/2013/04/reddits-best-comment-scoring-algorithm.html

Extending the Algorithm

His proposal is to consider each comment as a Bandit, with 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 to the ranking of their respective samples.
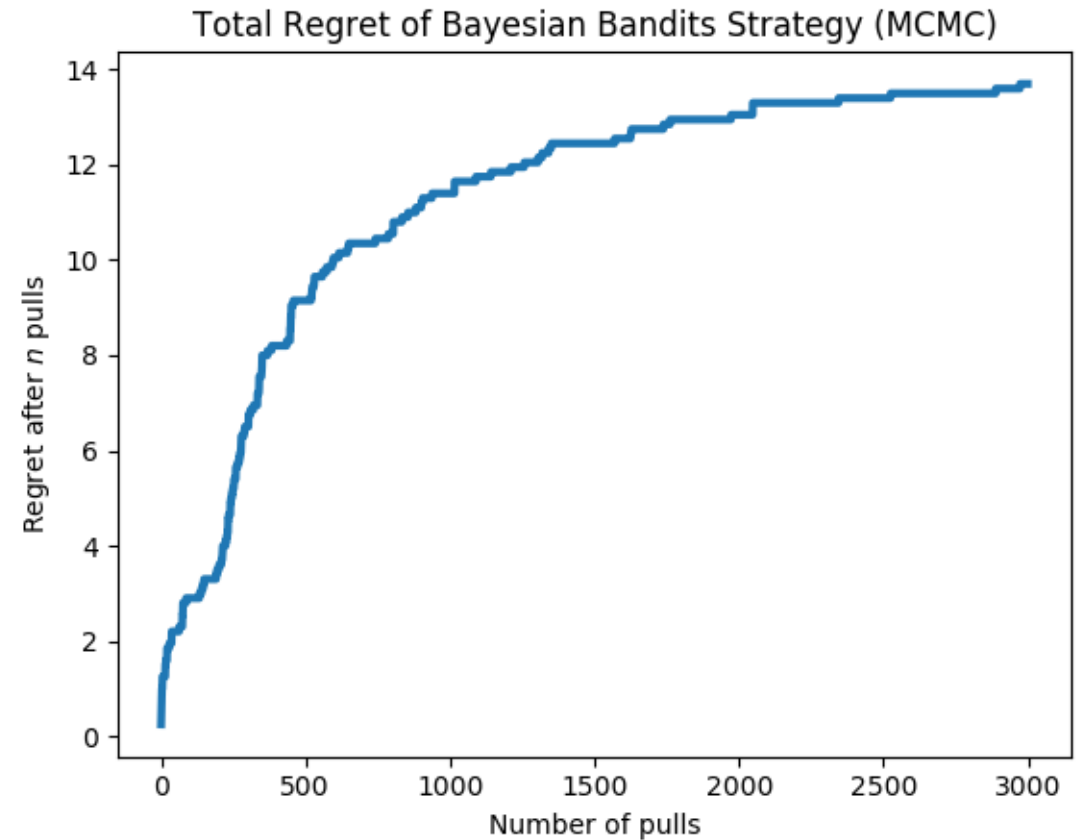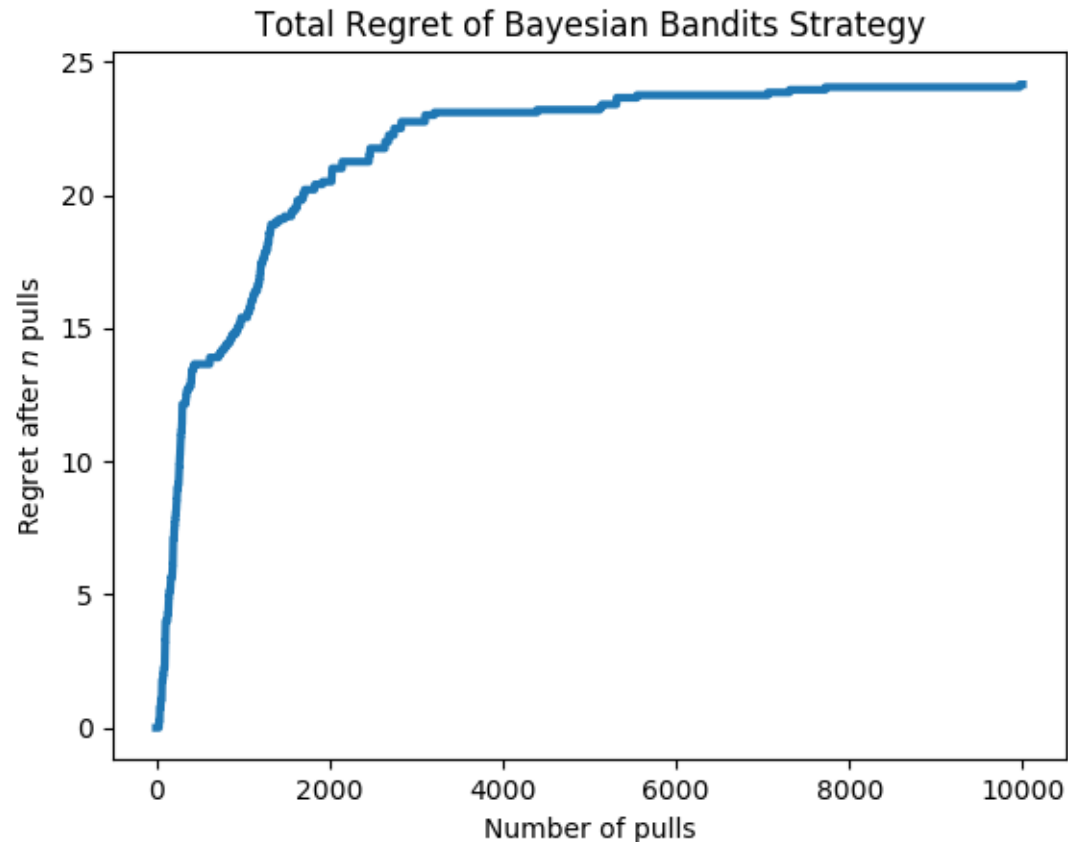
And yet MCMC …

```
for k in range(n):

        # sample from the bandits's priors, and select the largest sample
        choice = np.argmax(rbeta(1 + self.wins, 1 + self.trials - self.wins))


        # sample the chosen bandit
        result = self.bandits.pull(choice)


        # update priors and score
        self.wins[choice] += result
        self.trials[choice] += 1
        bb_score[k] = result
        self.N += 1
        choices[k] = choice
```

A MCMC Solution

```
for k in range(n):

        # sample from the bandits's priors, and select the largest sample
        P0 = pm.Uniform('P0', 0, 1); P1 = pm.Uniform('P1', 0, 1); P2 = pm.Uniform('P2', 0, 1)


        X0 = pm.Binomial('X0', value = self.wins[0], n = self.trials[0]+1, p = P0, observed = True)
        X1 = pm.Binomial('X1', value = self.wins[1], n = self.trials[1]+1, p = P1, observed = True)
        X2 = pm.Binomial('X2', value = self.wins[2], n = self.trials[2]+1, p = P2, observed = True)
        mcmc0 = pm.MCMC([P0, X0]); mcmc0.sample(6000, 1000); P0_samples = mcmc0.trace('P0')[:]
        mcmc1 = pm.MCMC([P1, X1]); mcmc1.sample(6000, 1000); P1_samples = mcmc1.trace('P1')[:]
        mcmc2 = pm.MCMC([P2, X2]); mcmc2.sample(6000, 1000); P2_samples = mcmc2.trace('P2')[:]
        choice = np.argmax([np.random.choice(P0_samples), np.random.choice(P1_samples), np.random.choice(P2_samples)])
```

Total Regret of Bayesian Bandits Strategy

A Better MCMC Solution

```python
P0_samples = [np.random.rand()]; P1_samples = [np.random.rand()]; P2_samples = [np.random.rand()]
for k in range(n):
    # sample from the bandits's priors, and select the largest sample
    choice = np.argmax([np.random.choice(P0_samples), np.random.choice(P1_samples),
                        np.random.choice(P2_samples)])
    result = self.bandits.pull(choice)

    # update priors and score
    self.wins[choice] += result
    self.trials[choice] += 1
    bb_score[k] = result
    self.N += 1
    choices[k] = choice

    if choice == 0:
        P0 = pm.Uniform('P0', 0, 1)
        X0 = pm.Binomial('X0', value = self.wins[0], n = self.trials[0], p = P0, observed = True)
        mcmc0 = pm.MCMC([P0, X0]); mcmc0.sample(15000, 5000); P0_samples = mcmc0.trace('P0')[:]
    elif choice == 1:
        P1 = pm.Uniform('P1', 0, 1)
        X1 = pm.Binomial('X1', value = self.wins[1], n = self.trials[1], p = P1, observed = True)
        mcmc1 = pm.MCMC([P1, X1]); mcmc1.sample(15000, 5000); P1_samples = mcmc1.trace('P1')[:]
    else:
        P2 = pm.Uniform('P2', 0, 1)
        X2 = pm.Binomial('X2', value = self.wins[2], n = self.trials[2], p = P2, observed = True)
        mcmc2 = pm.MCMC([P2, X2]); mcmc2.sample(15000, 5000); P2_samples = mcmc2.trace('P2')[:]
```

Total Regret of Bayesian Bandits Strategy

Total Regret of Bayesian Bandits Strategy (MCMC)