Multi-Armed Bandit Problem
STA 531 Final Project

Sunith Suresh, Lin Xiao, Ilan Man and Sanjay Hariharan

May 6, 2016

1 Introduction

This paper explores the multi-armed bandit problem (MAB). A multi-armed bandit is a sequential experiment with the goal of achieving the largest possible reward from a payoff distribution with unknown parameters. The term multi-armed bandit comes from slot machines where each machine is known as a one-armed bandit. In this set up, at each iteration the player must decide which of the arms to play next.

The task is complicated by the stochastic nature of the bandits in the following two ways:

1. 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 might not yield a reward if only played a few times.
2. If we have found a bandit that returns good results, do we keep drawing from it to maintain our good score, or do we try other bandits in hopes of finding an even better bandit? How do we know when to switch and when to stick to the current bandit? This is known as the *exploration vs. exploitation* tradeoff.

This is a classic reinforcement learning problem in machine learning literature and game theory.

This paper reviews several strategies for selecting bandits, including a creative application of dynamic programming. Note that there are many variations of the stochastic MAB, including contextual bandit and adversarial bandit, which we will not discuss here and is outside of the scope of this paper.

2 Multi-armed Bandits

In machine learning literature, the stochastic multi-armed bandit problem is formulated as follows:

> Given K machines, each with an unknown probability of yielding a reward, which come from a fixed but unknown distribution parameterized by θ_i, for $i \in (1, ..., K)$, and N total plays, decide which machines to play in order to maximize the total reward. The process by which we make a decision is known as the policy.

2.1 Regret vs. Reward

It is common in the literature to express the maximum reward as minimizing *regret* compared to the best arm in hindsight, like an opportunity cost. That is, define regret at each step as $c_t = R_t^* - R_{t,i}$, where R^* is the reward yielded by selecting the best machine and $R_{t,i}$ is the reward yielded by selecting machine i at time t. Note that in reality we don't know what the best machine is - this formulation is purely a way to compare the performance of algorithms using simulated data. The goal is then to devise a strategy to minimize $\sum_{t=1}^{N} c_t$. Note that since this is a stochastic problem, we aim to minimize expected regret.

2.2 Exploitation vs. Exploration

The tradeoff between playing all the machines many times and only playing the best machine is a key concern for optimizing an algorithm. The problem with exploring too much is that every time you play a machine that isn't the best, you incur some regret, since you won't be playing a machine with highest expected reward. On the other hand, if you exploit too quickly, then you won't have explored the entire space of machines and may miss out on the most optimal machine. This could be very costly, especially if you don't change your mind later on - you will always incur some cost in the form of the regret.

3 Bayesian approaches: Bernoulli Bandit

A popular approach to solving the MAB problem is to minimize regret using Bayesian inference. Specifically, assume that rewards are distributed as a Bernoulli random variable with some latent parameter, θ_i, specific to each machine. In addition, we can put a prior distribution on θ_i and update our belief about it as we run the algorithm and learn which machines are better than others. Since rewards are generated from a Bernoulli distribution, a natural choice for the distribution on θ_i is a Beta. Formally stated:

$$\theta_i \sim \text{Beta}(\alpha_i, \beta_i)$$

$$r_i = \text{Reward from machine i} \sim \text{Bernoulli}(\theta_i)$$

Exploiting the conjugacy of the Beta-Bernoulli model, the posterior distribution of getting a reward from machine$_i$, after playing for N iterations is:

$$\text{Beta}(\alpha_i + R_N, \beta_i + N - R_N)$$

where $R_N = \sum_{t=1}^{N} r_t$.

Before beginning to play, we assume no prior knowledge about any machine's propensity to yield a reward. Therefore, we initially set $\alpha_i, \beta_i = 1$, which is a common objective prior for the Beta distribution.

3.1 Dynamic Programming

As mentioned above, our goal is to either minimize regret or maximize total expected rewards, $\mathbb{E}(R_N)$. Under the above setup, we constructed an approach to maximizing future expected rewards based on a backwards, dynamic programming scheme. Note that we derive the algorithm assuming $K = 2$ machines.

Backwards algorithm

1. At time $T = N$, we are at our final trial, and thus are only concerned with the reward on this trial. As such, we opt for a **greedy** approach and pick the machine with the highest expected reward. Using a Beta posterior, that is: $\underset{i}{\text{argmax}}(\frac{\alpha_1}{\alpha_1+\beta_1}, \frac{\alpha_2}{\alpha_2+\beta_2})$
2. For $T = N-1, N-2, ..., 1$:
   - Given that we know the most optimal choice for the last trial, we select the machine on this trial that will give us the maximum expected sum of future rewards
   - $\mathbb{E}(\sum_{i=T}^{N} R_i | M_{1:T-1}, R_{1:T-1}) = \mathbb{E}(\sum_{i=T+1}^{N} R_i|\theta) + \mathbb{E}(R_T|\theta)$
     - $M_{1:t}$ = machines chosen from times $1, ..., t$
     - $R_{1:t}$ = rewards given from times $1, ..., t$

As an illustration, consider the following tree. Here we denote $f(\alpha_1, \beta_1, \alpha_2, \beta_2)$ as the posterior parameters of the machine at time T. Note that f here is a general utility function that determines which machine to play at the given time. It is a function of the parameters for each machine. Machine 1 has parameters α_1 and β_1, and machine 2 has parameters α_2 and β_2. Once our utility function selects a machine, we play that machine, and in our Bernoulli

setup, are given either reward or no reward. We then move down the branch accordingly, based on the machine we chose, and update the α or β parameter depending on the outcome.

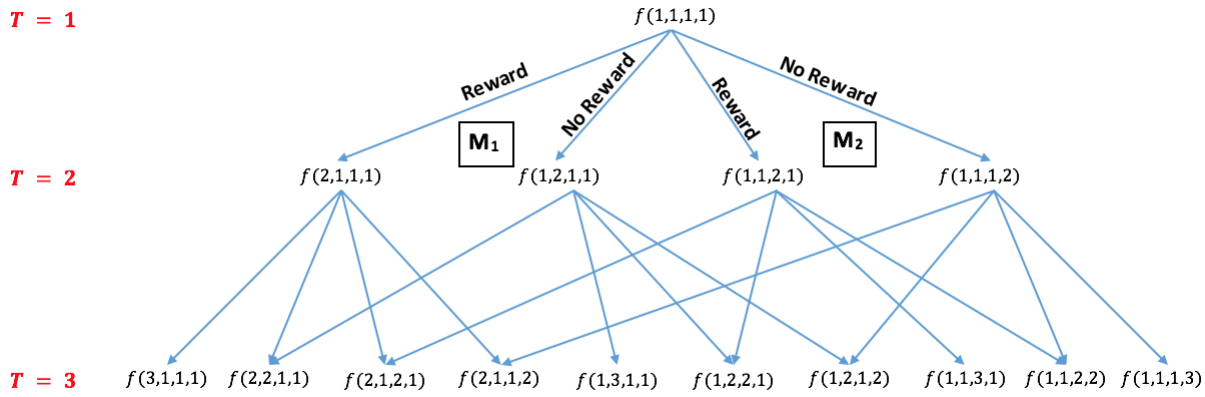


Figure 1: Forest of Possible Outcomes

As time increases, the complexity of possible branches increase exponentially, but it is important to note that some branches merge back together. Our algorithm requires two pieces of information:

1. Current parameters of the machine based on previous observations
2. Number of Trials we expect to perform

Our algorithm creates a tree of all possible future states, like above, and dynamically works backwards, assuming at the final stage we want to be completely greedy (since it is our last trial). The output of the algorithm is **the specific machine we should play at the current stage**, as well as the expected sum of rewards given you follow the algorithm at each trial.

This algorithm does not choose the greedy approach at each stage.

Simulation	α_1	β_1	α_2	β_2	Greedy	Dynamic Programming	Expected Reward
1	2	1	1	1	Machine 1	Machine 1	5.75
2	1	1	2	1	Machine 2	Machine 2	5.75
3	3	4	1	1	Machine 2	Machine 1	5.26
4	2	2	5	6	Machine 1	Machine 2	5.15
5	5	5	6	6	N/A	Machine 1	5.05

Table 1: Simulation Results for 5 Inputs

Note the following observations:

1. The first two rows are trivial, as the greedy and dynamic approach both pick the machine with the highest expected reward.
2. In the 3rd simulation, the greedy algorithm would choose machine 2, as its expected reward is $\frac{1}{2}$, but based on the possible consequences of choosing that branch, it opts for the safer choice, machine 1.
3. The 4th simulation is similar, where our algorithm chooses the safer choice, i.e. the one with more data points and less chance of having a poor path.
4. The final simulation reflects this result as well, as both machines have an identical expected reward, so the greedy algorithm will not care which one we pick, but our algorithm prefers the one with more data.

Our algorithm dynamically updates our machine parameters based on rewards, but we did not define any latent parameters for our machines. Next, we will explore other algorithms, and compare each one's effectiveness based on bandits with 'true' parameters.

3.2 Thompson Sampling

As mentioned in the algorithm above, our approach to solving the MAB problem is inherently Bayesian. Below we outline a popular Bayesian approach used in practice called Thompson Sampling

1. For $t = 1, ..., N$
   (a) $f(\alpha_i, \beta_i) = i$ // select machine i at time t using the policy function
   (b) $r_t \sim \text{Bernoulli}(\theta_i)$ // We play machine i, and observe reward r_t
      - If $r_t = 1 : \alpha_i = \alpha_i + 1$ // increase our positive belief in machine i
      - else if $r_t = 0 : \beta_i = \beta_i + 1$ // increase our negative belief in machine i
   (c) $R_t = R_t + r_t$ // add reward at time t to running total

This algorithm provides a framework for how to think about this problem in a Bayesian way. To maximize expected rewards, we must optimally choose the policy function f.

Below we review 3 ways to select f:

1. **Explore**: Randomly select one of the K machines, without considering how many win/losses we've seen. This is a pure exploratory strategy and is used for comparison purposes only. In reality no one would choose this approach.
2. **Exploit**: Sample each $\theta_i \sim Beta(\alpha_i, \beta_i)$. Select $\underset{i}{\text{argmax}}\ \mathbb{E}[\theta_i]$. This is called exploit because we are greedily choosing the best machine at each step based on expected rewards. Note however that even if we select the best expected machine, there is a probability of $\frac{\beta_i}{\alpha_i+\beta_i}$ that it fails to generate a reward. So the amount of exploration is much less than above but still non-zero. Note that this approach is known as *Adbandit*.
3. **Thompson-Sampling**: Sample each $\theta_i \sim Beta(\alpha_i, \beta_i)$. Select $\underset{i}{\text{argmax}}\ \theta_i$. This approach falls somewhere between Exploit and Explore, but is much closer to Exploit.

We ran these three approaches on a dataset of $K = 5$ machines with true reward distribution, $\theta = [0.05, 0.1, 0.3, 0.2, 0.5]$ and calculated the cumulative regret after 1000 plays. To avoid sampling variability we ran this simulation 50 times and took the average:

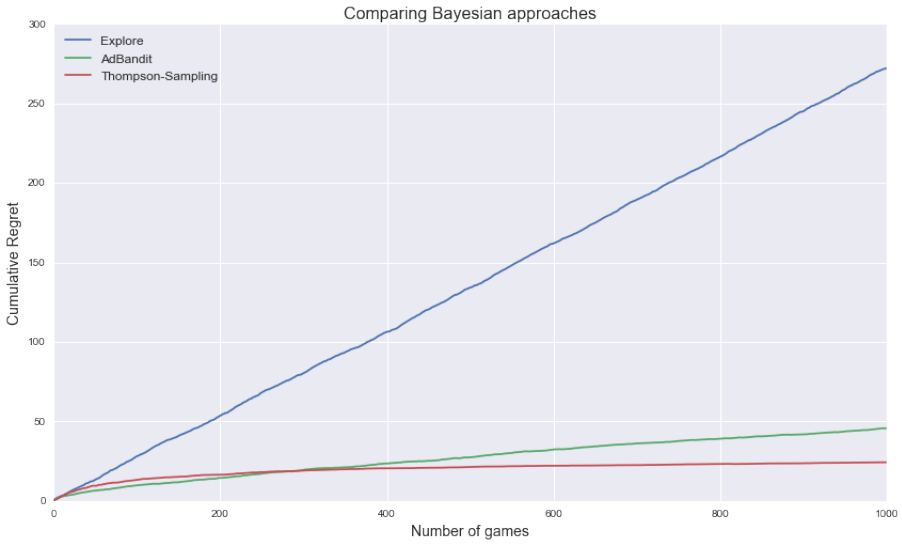


Figure 2: Cumulative Regret

As expected, the Explore strategy performs the worst. Adbandit does the best early on, but then Thompson-Sampling ends up doing better. This is because of the amount of exploration in Thompson-Sampling which is slightly higher than Adbandit. In addition we ran it (not shown here) for $K = 10$ and $K = 20$ machines, and Thompson-Sampling does better, earlier on. This suggests that Adbandit's highly exploitative nature means it's more likely to get stuck in a suboptimal machine for longer.

3.3 ϵ-greedy

The most simple approach to tackling the MAB problem is the ϵ-greedy algorithm. As explained earlier, the greedy algorithm chooses the bandit with highest expected reward at every iteration. The problem with this purely exploitative approach is that it will take a large number of iterations to explore all the machines and converge onto the best machine. The ϵ-greedy algorithm is almost a greedy algorithm as it generally exploits the best machine, however every once in a while it explores other machines.

The value of ϵ, defined by the user, controls the level of exploration and exploitation employed by the algorithm. The algorithm can be described as follows: For a proportion $1 - \epsilon$ of the trials, choose the machine with highest expected reward, and for a proportion of ϵ explore other machines with uniform probability.

Given machines $\{1, ..., K\}$ with initial empirical means $\{\hat{\mu}_1(0), ..., \hat{\mu}_K(0)\}$ For trials $t = \{1, ..., N\}$, the probability of choosing machine i is given by

$$p_i(t+1) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{K}, & \text{if } i = argmax_{j=1,..,K}\hat{\mu}_j(t), \\ \frac{\epsilon}{K}, & \text{otherwise.} \end{cases}$$

It follows from the algorithm that if $\epsilon = 0$, the algorithm reduces to a purely exploitative one, whereas if $\epsilon = 1$, it reduces to a purely explorative one.

The ϵ-greedy algorithm is considered to be suboptimal due to the constant value of ϵ. As the number of trials increase, asymptotically we can be reasonably certain as to which machine is the best machine from the observed rewards. However, the algorithm will always explore for ϵ percent of the trials. To address this issue, variants of the algorithm have been proposed. Two popular variants are known as ϵ-first algorithm and ϵ-decreasing algorithm.

The ϵ-first algorithm consists of doing the exploration all at once at the beginning and then switching to pure exploitation. Given N total trials, for ϵN trials, the machines are randomly chosen (with uniform probability). After the exploration phase, for the remaing $\big(1 - \epsilon\big)N$ trials, the algorithm switches to pure exploitation, choosing the machine with highest expected rewards.

The ϵ-decreasing algorithm consists of decreasing the value of ϵ with number of trials. The algorithm progresses from a highly explorative approach in the beginning to a highly exploitative one towards the end.

CesaBianchi and Fisher (1998) found that ϵ-decreasing algorithm is theoretically efficient (with respect to regret). However empirical studies by Vermorel and Mohri (2005) do not seem to find any significant improvements with the variants of the algorithm when compared to the original ϵ-greedy algorithm.

We ran the ϵ-greedy algorithm, with ϵ set to 0.1, which is often used as a standard baseline measure, on the dataset that we simulated:

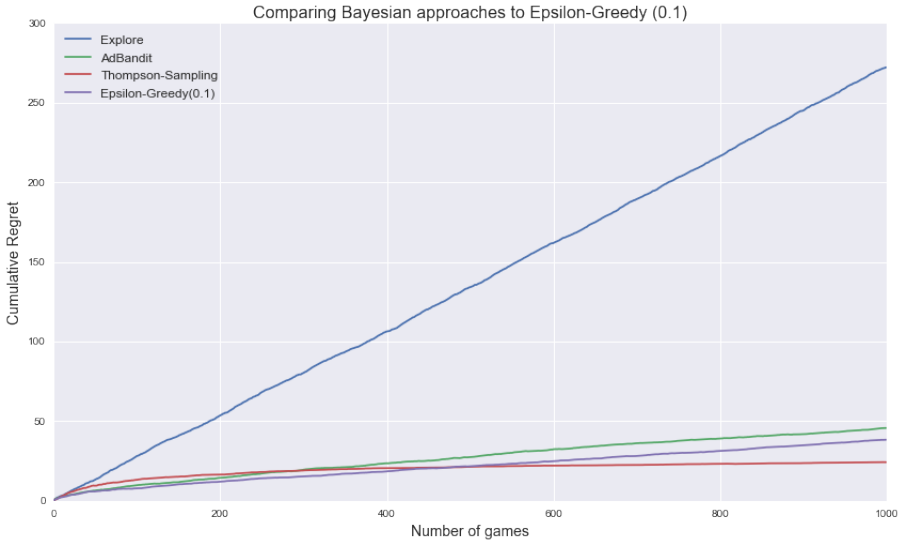


Figure 3: Cumulative Regret

We find that the ϵ-greedy algorithm performs better than the pure exploration and Adbandit strategies. It performs worse than Thompson Sampling, which is expected due to constant value of ϵ. ϵ is usually chosen by the user based on the specific nature of the problem.

3.4 Upper Confidence Bounds

The other algorithms presented in this paper are similar in that they pay attention only to how much reward they've gotten from the machines. This means that they're likely to under-explore options whose initial experiences were not rewarding, even though they may not have enough data to be *confident* about those arms. One naive approach to solving this problem is to run the algorithm multiple times and take the average of the results. Another could be to run the algorithm for a very long time and hope the probabilistic nature will eventually correct for random variation in the reward distribution. Yet another approach is to use probability theory to bound our confidence in how good or bad each machine is. This is exactly what Upper Confidence Bounds does.

The upper confidence bound (UCB) family of algorithms selects the machine with the largest upper confidence bound at each round. This paper will only focus on UCB1, but note that there are many variants of the UCB family. The more times you play a machine, the tighter the confidence bounds become. So as the number of plays for each machine increases, the uncertainty decreases, and so does the width of the confidence bound.

We want to know with high probability that the true expected payoff of a play $\hat{\mu}_i$ is less than our prescribed upper bound:

$$\bar{\mu_i} + \sqrt{\frac{2ln(t)}{n_i}}$$

Where $\bar{\mu_i}$ is the average reward obtained from machine i and n_i is the number of times machine i has been played so far.

This upper bound is the sum of two terms, where:

1. the first term is the average empirical reward
2. the second term is related to the one-sided confidence interval for the average reward according to the Chernoff-Hoeffding bounds

Recall that since rewards follow a Bernoulli distribution, we can apply Chernoff-Hoeffding to upper bound the probability that the sum of rewards from each machine deviates from its expected value:

$$\mathrm{P}(Y + a < \mu) \leq e^{-2na^2}$$

This confidence bound grows with the total number of actions we have taken but shrinks with the number of times we have tried any particular action. This ensures that each action is tried infinitely often but still balances exploration and exploitation. It can be shown that the regret for UCB1 grows with **ln(n)**, as witnessed below for the optimal machine.

Note that in addition to keeping track of our confidence in the estimated values of each machine, the UCB algorithm doesn't use randomness at all. Unlike the other algorithms in this paper, it's possible to know exactly how UCB will behave in any given situation. This can make it easier to reason about at times.

3.4.1 UCB1: algorithm

Assuming K machines:

1. Play each arm once
2. Observe rewards r_i, for $i = 1, ..., K$
3. Set $n_i = 1$, for $i = 1, ..., K$
4. set $\bar{\mu_i} = \frac{r_i}{n_i}$
5. For time $t = K + 1, ..., N$:
   (a) Play arm $\hat{i} = \underset{i}{\text{argmax}}\ (\bar{\mu_i} + \sqrt{\frac{2ln(t)}{n_i}})$
   (b) Observe reward r
   (c) $r_{\hat{i}} = r_{\hat{i}} + r$
   (d) $n_{\hat{i}} = n_{\hat{i}} + 1$
   (e) update $\bar{\mu_i} = \frac{r_{\hat{i}}}{n_{\hat{i}}}$

Below we ran UCB1 on the same dataset as above. Note that since this is only a comparison between machines, sampling variability isn't an issue like it would be if we compared UCB1 vs. ϵ-greedy, for example.

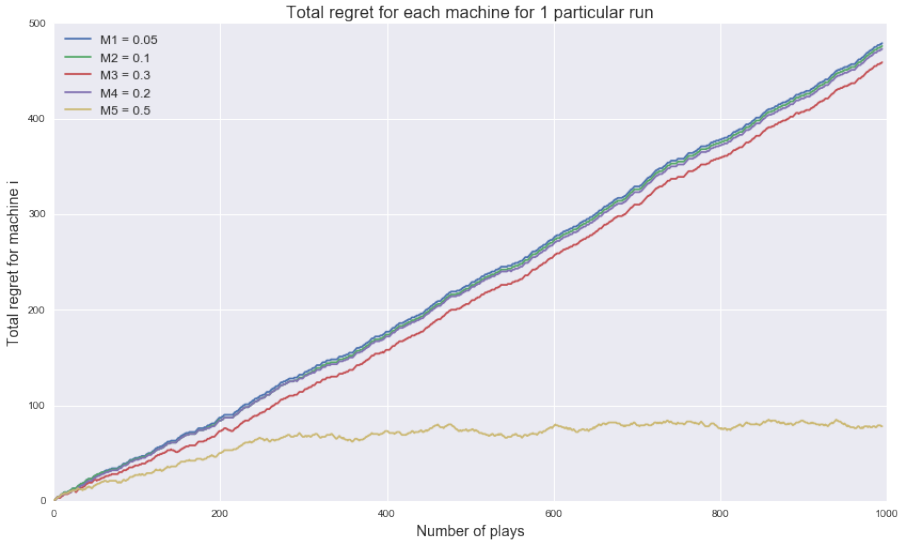


Figure 4: Cumulative regret for all 5 machines

Above we see that, as expected, machine 5 has the lowest regret, since it has the highest θ. It only takes a few iterations for UCB to find this. Additionally, as mentioned earlier the total cumulative regret grows logarithmically.

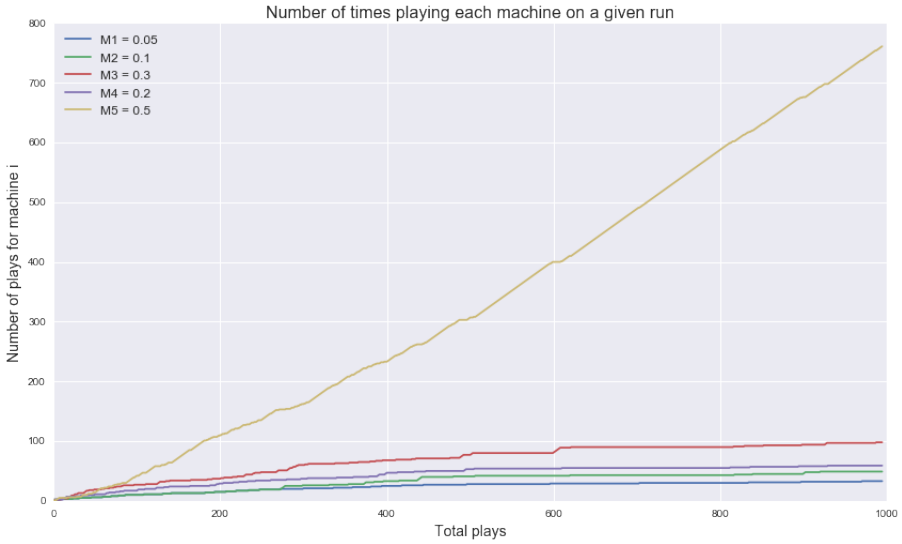


Figure 5: Switching between machines

As expected, machine 5 is played the most often, even though the worse off machines are still being played, in the tail. Separation from the other machines occurs around 60 iterations.

Finally, we compute the total cumulative regret for all 5 machines on UCB1, for 1000 iterations, and average 50 simulations, and compare against the other approaches from earlier:

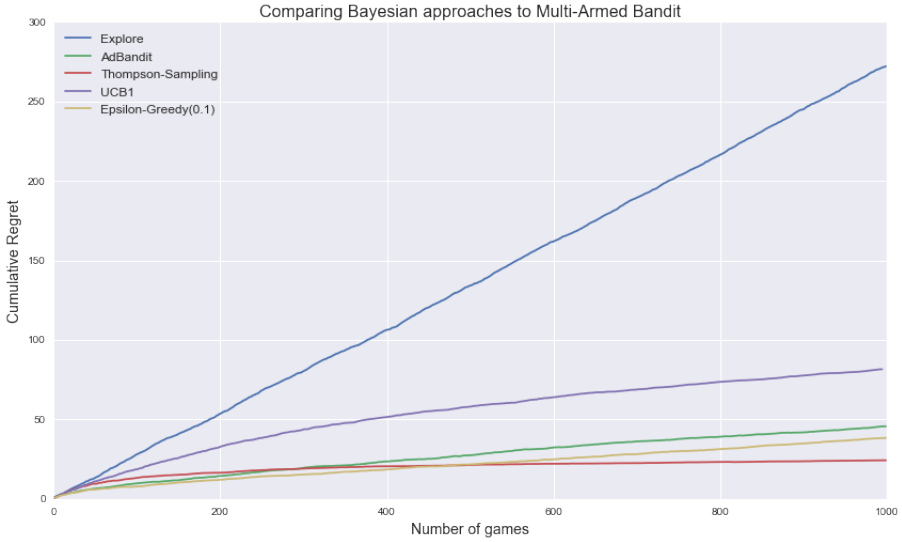


Figure 6: Comparing all 5 approaches

UCB1 performs the worst, other than random exploration. This is due to the nature of it's upper bounded-ness, suggesting that while UCB1 provides nice theoretical guarantees, Bayesian heuristic approaches and even simple ones like ϵ-greedy can outperform it on average.

4 Continuous Reward

4.1 Gaussian Processes

While the methods we employed above can be easily applied to a continuous reward framework, in this section we outline an alternative method. This scenario may be more intuitive in a slot machine or A/B testing analogy, where

rewards may be continuous with respect to the dollars earned or lost.

We model our reward function as a Gaussian Processes (GP), specified by its mean and covariance function. We optimize an unknown reward function f. During each time step $t = 1, ..., N$, a machine i is chosen and a reward is sampled from f with added Gaussian noise.

4.2 Prediction

Using a reward function sampled from GP(0, k(x,x)) as our prior, we find the mean and covariance of the posterior predictive distribution as follows:

$$P(y_{t+1}|D_{1:t}, x_{t+1}) = N(\mu_t(x_{t+1}), \sigma_t^2(x_{t+1}) + \sigma_{noise}^2)$$
$$\mu_t(x_{t+1}) = k^T[K + \sigma_{noise}^2 I]^{-1} y_{1:t}$$
$$\sigma_t^2(x_{t+1}) = k(x_{t+1}, x_{t+1}) - k^T[K + \sigma_{noise}^2 I]^{-1} k$$

In order to balance the tradeoff between exploitation and exploration , we should choose the next point to be either where the mean is high (exploitation) or the variance is high (exploration).

The process above is our policy, also known as acquisition function, which guides the optimization by determining which x_{t+1} to observe next.

4.3 Acquisition Function

We define our acquisition function as follows:

$$PI(x) = p(f(x) \geq \mu^+ + \xi) = \Phi(\frac{\mu(x) - \mu^+ - \xi}{\sigma(x)})$$

- μ^+ : The highest value of all the observed values so far
- ξ : Tuning parameter to adjust exploration vs. exploitation.

The figure below demonstrates our acquisition function. At each value, the posterior predictive function has a mean and variance, computed using the above formula. We construct a Normal CDF using the mean and variance for each value, and evaluate the CDF at the highest observed value. This result is known as the **Probability of Improvement**, and at each iteration we select the value with the highest calculated Probability.

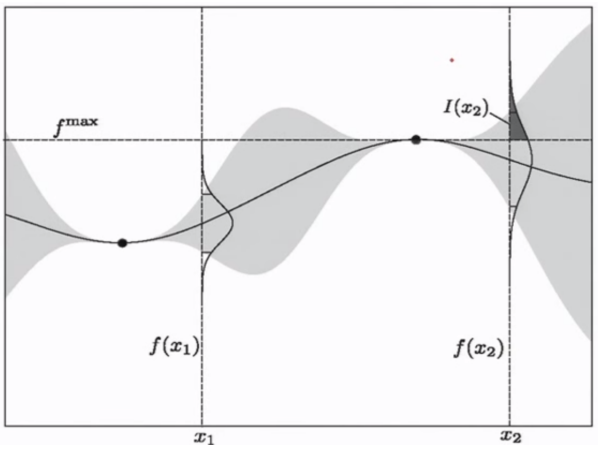
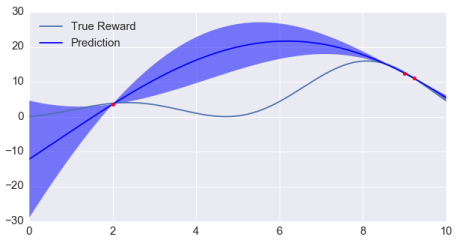
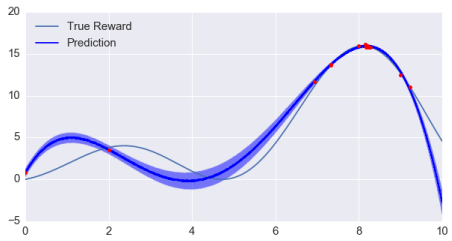


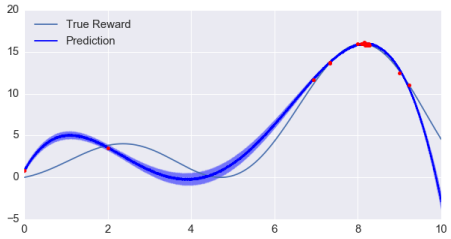
Figure 7: Probability of Improvement (PI)



(a) Iteration 1



(b) Iteration 15



(c) Iteration 30

Figure 8: GP Fit evolves as we sample rewards and iterate back into prediction function.

4.4 Empirical Results

Below we provide output for the GP model, run on 10,000 machines, at iterations 1, 15, and 30. Note that as the number of iterations increase, the GP closer resembles the true reward function. Once the best machine is chosen, the algorithm will continue selecting this machine.

In Figure 8 we can see that over time, our observed values tend to cluster close to the highest point (i.e. the best machine), and the overall fit matching close to the true function.

We also compare the cumulative Reward and Regret in the GP method.

Figure 9: Cumulative Regret and Reward

In the first 10 iterations, the GP method is exploring high variance areas of our reward function, and we can see that

this results in a lower reward and higher regret, but after this point, it settles upon the best option. From there, the regret levels out and the reward increases linearly while exploiting the best machines.

5 Conclusion

In this paper, we explored the multi-arm bandit problem. The problem essentially entails the exploration vs exploitation tradeoff, where one has to choose between maximizing current expected reward versus maximizing future expected rewards. Such situations arise in various application domains including clinical trials, stochastic scheduling and economics. Assuming that rewards are modeled using a Bernoulli distribution, we discuss five strategies - Exploration, Adbandit, Thompson Sampling, ϵ-greedy and Upper Confidence Bounds. All methods provide approximate solutions to the MAB problem. Using cumulative regret as a performance metric, we compared the algorithms and found the Thompson Sampling seems to perform the best, followed by ϵ-greedy, Adbandit, Upper Confidence Bounds and Exploration. We also describe a dynamic programming approach that provides a solution in simple cases (small trial sizes). Assuming the rewards are distributed using a continuous Gaussian distribution, we also describe a solution utilizing Gaussian processes.

6 References

1. J. White, Bandit algorithms for Website Optimization
2. Kuleshov and D. Precup, Algorithms for the multi-armed bandit problem
3. M. Mohri and J. VermorelMulti-armed Bandit algorithms and Empirical Evaluation
4. Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem
5. J. Chakravorty and A. Mahajan, Mutli-Arm Bandit, Gitten's Index and it Calculation