Dynamic Game Difficulty Balancing Using Bandit Algorithms[1]

Rui Xiao
Tufts University
Rui.xiao@tufts.edu

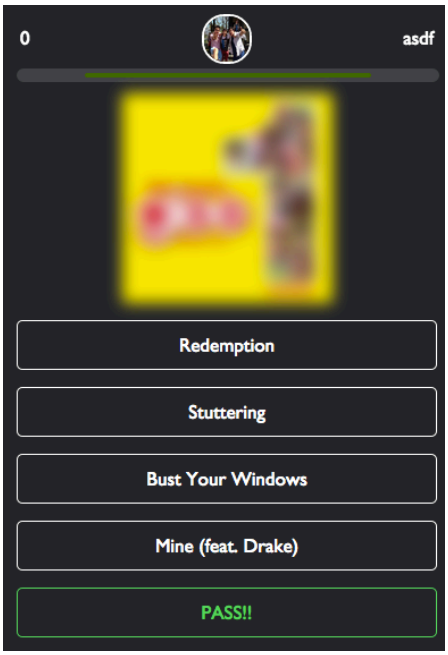
ABSTRACT

Video Games are fun to play only when it's moderately difficult. However, it is hard for a developer to find out how hard a game should be in order to maximize players' enjoyment. The goal of this paper is to introduce some traditional bandit algorithms and to show their performance in optimizing players' enjoyment on my mobile music trivia game, Quizzie[2]. In this paper, I am going to first explain the background of my game and how I apply the bandit algorithms to it. In the second section, I will formally define my bandit formulation and implementation. Afterwards I will show how I build simulation process using actual data. In the end, I will compare the performance of other traditional bandit algorithms using my simulator and show the progress made in difficulty balancing. Something to notice is, since not enough data are collected yet, the paper aims at illustrating the idea not the actual conclusion or findings.

[1] This is a final paper for COMP150AML: Advanced Topics in Machine Learning: Learning, Planning and Acting in Complex Environments taught by Professor Roni Khardon in Fall 2014 at Tufts University.
[2] This game is built using HTML/JavaScript and optimized for mobile. Link: http://rayxiao92.github.com/phongo/musicquiz

INTRODUCTION

I built a music trivia game prototype at 2014 Tufts Hackathon using music streaming API provided by Spotify. The game randomly plays a 30-second song track each time and provides users 4 options to guess the track name. When the user answers correctly, reward will be given to her.



After the Hackathon, I learned that in order to make the game more fun, it needs to play songs that are familiar to each player specifically and select a moderate difficulty in the quiz construction. The app now allows users to search an artist, and constructs a quiz episode by mixing songs according to the searched artist and ones by related artists. Thus, assuming the player knows well about the artist she selects, the proportion mix would mainly determine the difficulty of the game. In this way, different ratios become the arms in the bandit-problem; the "play again" rate of the corresponding arm is the reward for each action. "Play again rate" means the percentage of people who play again after the last episode. To facilitate this functionality, I added following features:

- Search an artist as the player wish and construct a playlist instantly
- Collect data while the user is playing
- Use Parse API as backend to store and control the difficulty bandit
- Better UI improvement
- Improvements on app details

ALGORITHMS

The bandit setting is formally described as follows:

Bandit Setting: *Before each episode starts and after the user has selected the artist to play, we have a set of different difficulty level to choose. After we choose a level, we will observe the reward r. r will be 1 if the player plays again and 0 if the player quit in the middle or after the episode.*

A few assumptions and explanation I have for the setting:

1. Players have same taste and accuracy for different artists

This is obviously not true but I assume so to simplify the problem. Supposedly a user would have different return rate when playing different artists (a user is more likely to like the game if she is familiar with the music) and answers quizzes with different accuracies (if she's familiar with the music, she is supposed to get high score).

2. How exactly the difficulty is determined

Echonest kindly provides a free API to facilitate this function. When you query one artist, it returns a number of artists that are related. For example, if the difficulty level is 0.3, my app does as follows:

Algorithm 1 Defining Difficulty Level

Query Round(0.3 * T) related artists using Echonest and call them related artists
For i = 1… T **do**
    If random > difficulty level
        Load a track from main artist
    Else
        Load a track from related artist
End for

Where T is the total number of songs in an episode.
Since each episode of a game is now modeled into a single state bandit-problem, I chose e-greedy algorithm as my initial implementation of the application. E-greedy algorithm is defined as follows:

Algorithm 2 e-greedy:

For each state, we choose a difficulty that:
If Rand(0 , 1) > epsilon
    Pick a random action
Else
    Pick an action with highest value

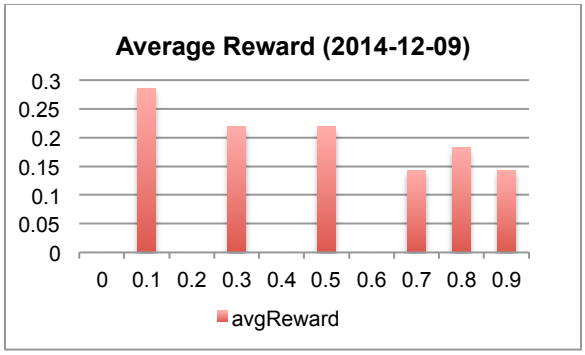
I choose e-greedy algorithm as an initial solution due to following reasons:

Decent performance: According to the simulation data test in [1] on news article recommendation problem, e-greedy has a decent performance using both parameters comparing to other algorithms like Upper Confidence Bound (UCB), Thompson Sampling (TS), or Exploitation (greedy). It performs on average slightly worse than Thompson Sampling, which does not provide an upper confidence bound; performs on average better than UCB algorithm and greedy algorithm.

Lightweight: Comparing to TS and UCB, e-greedy is lightweight because it only updates one arm value in the database. This is empirically important because it saves network traffic for music streaming in my game and implementation is easy using Parse API, which is applied as the back-end database for the game.

Exploration: In the early stage of the app, using e-greedy with higher exploration rate helps learn faster without causing much regret comparing to a random algorithm. The tradeoff between exploration and exploitation is easily adjusted by changing the epsilon value.

E-greedy Performance: The graph below shows the average reward of different algorithm using 228 data rows. Since the game is not officially released yet, the user behavior can be distorted because they come from 3 people who generated them. As I mentioned, they serve to show the idea of how different the algorithm works but not the actual public data or conclusion of user behavior as of now.



Prior Algorithm: *For each state, we choose the same difficult level based on our belief.*

This algorithm is considered as an alternative because it's naïve. Intuitively I will guess 0.3 or 0.5 can be good possible proportion to mix because anything larger than 0.5 would deviate too much from the theme chosen; if the player knows the artist very well, then 0 might be boring for the player.

UCB algorithm: this is a classic algorithm that aims to minimize the cumulative regret within a period of time. I used the equation:

$$a_n = argmax\ Q(a) + \sqrt{\frac{Q(a))}{n(a)}}$$

In calculating the state value. This equation is a simplified version and does not require update to all action values. In practice, it might be a little costly to update all possible action values every time the player plays an episode.

SIMULATIONS

In this paper, we do not have enough real data to do real experiments for different bandit algorithms. However, it is possible to make assumptions on user's performance (how good they are) and predict whether the user will play again under certain conditions using history data.

I collected information including: accuracy, difficulty level, date, artist information, and whether the player returned after each episode. After cleaning out data generated through engineering process, I collected 88 valid data points and manually applied polynomial kernel (d=2), and ran the RandomForest classifier on the expanded dataset using WEKA. 10-cross validation yields around 82% accuracy. Since I can make probability assumptions on users' performance using past data, I can then predict whether a user returns after each episode and test other bandit algorithm using these simulated episodes.

Specifically, I have following assumptions and notes on my dataset:

1. A "good" player on average correctly answers 60% songs from the selected artist and 30% on related artists; a "bad" player correctly answers 30% and 25%, respectively; the exact probability are randomized in each episode.
2. 60% of total players are "good"; 40% are bad
3. Since I used e-greedy algorithm to collect these data, each difficulty level are not evenly allocated and there is lack of information for level in range between 0.6 and 0.9

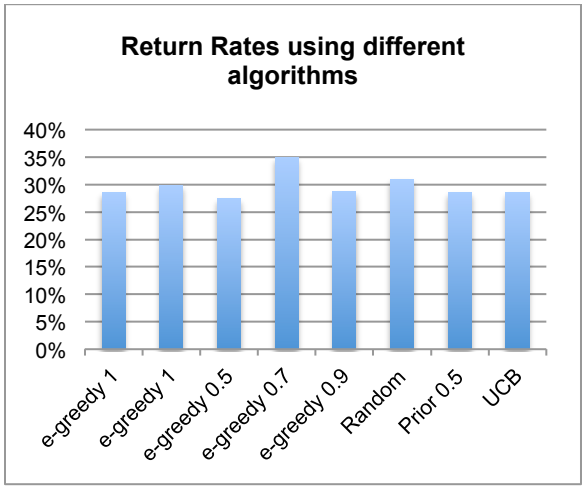
RESULTS

Algorithm Comparing: I have ran following algorithms on my simulator:

- **Random**: random algorithm picks difficulty level randomly from the set
- **EG 0.5**: EG 0.5 picks the difficulty level with highest average reward at 50% of the time, and random at 50% of the time
- **EG 0.7**: EG 0.7 picks the difficulty level with highest average reward at 70% of the time, and random at 30% of the time
- **EG 0.9**: EG 0.9 picks the difficulty level with highest average reward at 90% of the time, and random at 10% of the time
- **EG 1**(greedy): EG 1 picks the difficulty level with highest average reward
- **Prior 0.3**: Prior 0.3 always picks the difficulty level 0.3
- **Prior 0.5**: Prior 0.5 always picks the difficulty level 0.5
- **Prior 0**: Prior 0 always picks the difficulty level 0
- **UCB**: UCB picks the difficulty level with highest average score, where the score is defined as:

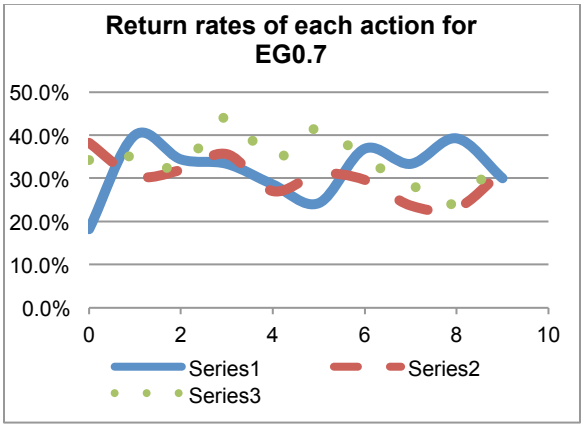
$$a_n = argmax\ Q(a) + \sqrt{\frac{Q(a))}{n(a)}}$$

Where a is the action chosen, and n(a) is the number of this action was chosen before



The result shows that using EG 0.7 yields a good improvement comparing to random algorithm and increases the return rate from 30% to 35%. It is unexpected that UCB and EG with other parameters do not even do as good as random algorithm. I guess this is related to the training example as it is not (i) accurate enough (only 82% accuracy) and (ii) biased as it provides a lot more data for difficulty that equals to 0.3. Another possible reason is the noise caused from different artists, which we ignore in order to simplify the problem.

The result for Prior 3 and Prior 0 do not make sense because of the biased training dataset. They are the most commonly used difficulty levels while I was working on the engineering part, and both yield ridiculously high returns (>60%). Thus, I ignored them in the chart.



Stability of the stochastic part: When I ran a same algorithm EG 0.7 for 3 times, I found that after 500 iterations, each state in each times are approximately equivalent. This matches to my expectation. Also, the value of arm 3 is higher than most of other actions as well. This should be correct too because arm 3 has a ridiculously good performance (80%) when I ran Prior 0.3 algorithm, but this could be resulted from the error I mentioned before.

CONCLUSIONS

We conclude that Bandit algorithms actually help optimize the game performance comparing to base cases (random/prior 0.5). However more data need to be collected in order to train a better predicting algorithm.

Our artist assumption needs to be later extracted in the future work. Our predicting algorithm currently ignores the effect that player might respond differently to different artists while this is actually true in the real game: a player might continue playing the game because she likes the artist theme.

REFERENCES

[1] Chapelle, Olivier, and Lihong Li. "An empirical evaluation of Thompson sampling." *Advances in Neural Information Processing Systems*. 2011.

ACKNOWLEDGEMENT

I would like to thank Professor Roni Khardon who guided me to formalize the design and the teaching assistant Mengfei Cao who helped me brainstorm details of the algorithm.