Animesh Goyal
May 21, 2019 · 11 min read · Member-only

Solving Cold User problem for Recommendation system using Multi-Armed Bandit

This article is a comprehensive overview of using Multi-Armed Bandit to recommend a movie to a new user

Umm not the cold user we are referring to

Written by: Animesh Goyal, Alexander Cathis, Yash Karundia, Prerana Maslekar

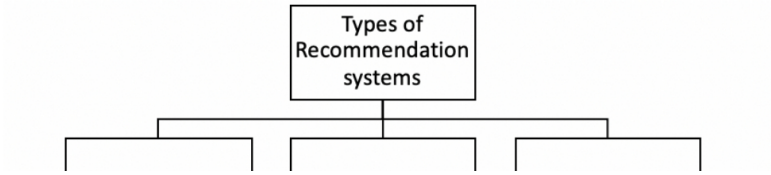
Introduction

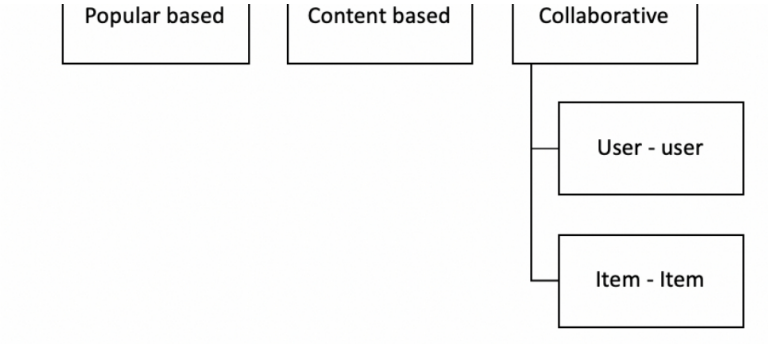
How often you feel after a hectic day at work that what should I watch next? As for me — yes, and more than once. From Netflix to Prime Video, the need to build robust movie recommendation systems is extremely important given the huge demand for personalized content of modern consumers.

Once at home, sitting in front of the TV seems like a fruitless exercise with no control and no remembrance of content that we consumed. We tend to prefer an intelligent platform which understands our tastes and preferences and not just run on autopilot.

In this article, I have tried to build a recommender system that would suggest the best movie to you in the shortest span of time. This recommender system can also be utilized in the recommendation of a broad range of items. For instance, it can be used to recommend products, videos, music, books, news, Facebook friends, clothes, Twitter pages, Android/ios apps, hotels, restaurants, routes, etc.

Types of Recommendation system





Types of Recommendation system

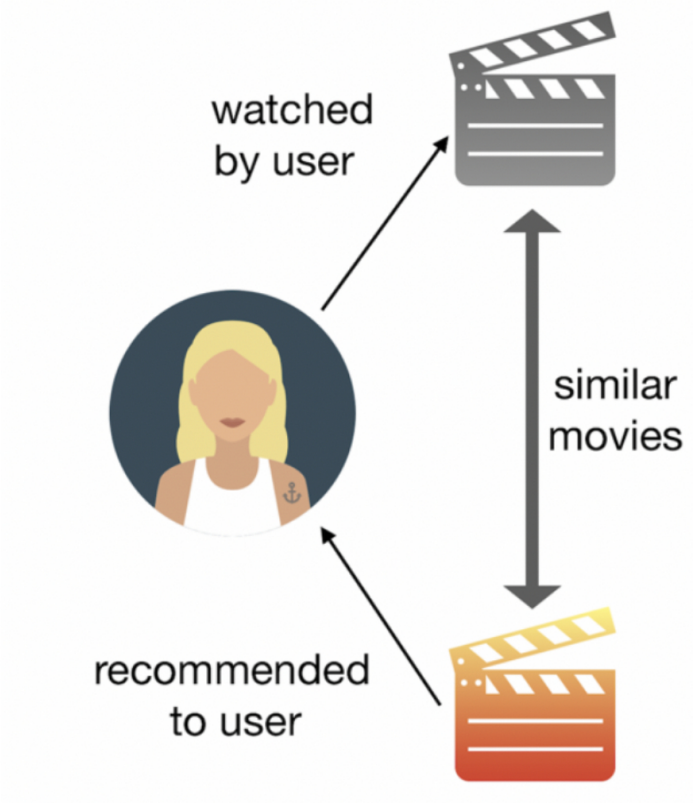
Existing Solutions

1. Popularity based recommendation system

As the name suggests the Popularity based recommendation system works with the trend. It basically uses the items which are in trend right now. For example, if any product is usually bought by every new user then there are chances that it may suggest that item to the user who just signed up.

The problem with the popularity-based recommendation system is that personalization is not available with this method i.e. even though you know the behavior of the user you cannot recommend items accordingly.

2. Content-based recommendation system



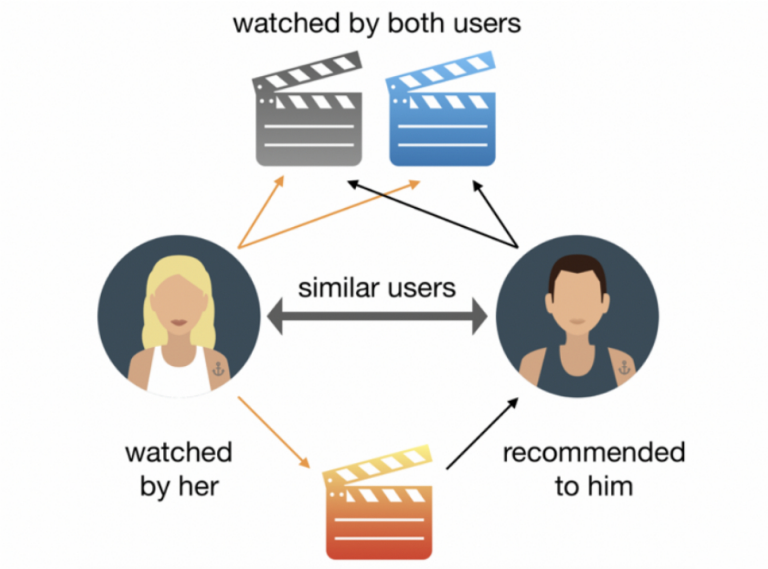
Content-based recommendation system

Content-based filtering is using the technique to analyze a set of documents and descriptions of items previously rated by a user, and then build a profile or model of the user's interests based on the features of those rated items. Using the profile, the recommender system can filter out the suggestions that

Using the profile, the recommender system can filter out the suggestions that would fit for the user.

The problem with content-based recommendation system is if the content does not contain enough information to discriminate the items precisely, the recommendation will be not precisely at the end

3. Collaborative based recommendation system



Collaborative based recommendation system

The key idea behind the Collaborative based recommendation system is that similar users share the same interest and that similar items are liked by a user. There are two types of Collaborative based recommendation systems: User-based and Item-based. We will be using a User-based filtering process.

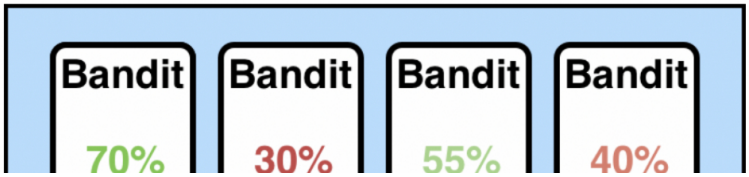
But these are unable to tackle the problem of Cold user.

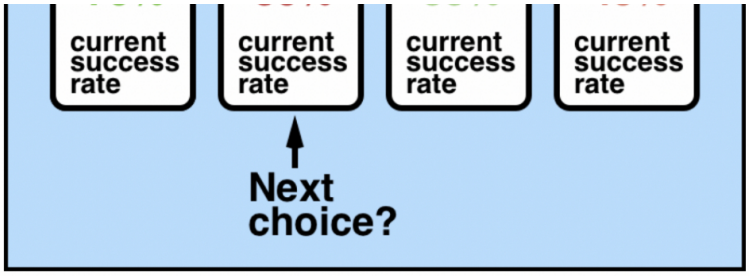
The Problem — Cold Start

So, what is the cold start problem? The term derives from cars. When it's really cold, the engine has problems with starting up, but once it reaches its optimal operating temperature, it will run smoothly. With recommendation engines, the "cold start" simply means that the circumstances are not yet optimal for the engine to provide the best possible results. Our aim is to minimize this time to heat the engine. The two distinct categories of cold start: product cold start and user cold starts. In this blog, we are concentrating on the user cold-start problem.

To solve this problem, we are introducing the concept of using Multi-Armed bandit

Multi-Armed Bandit (MAB)





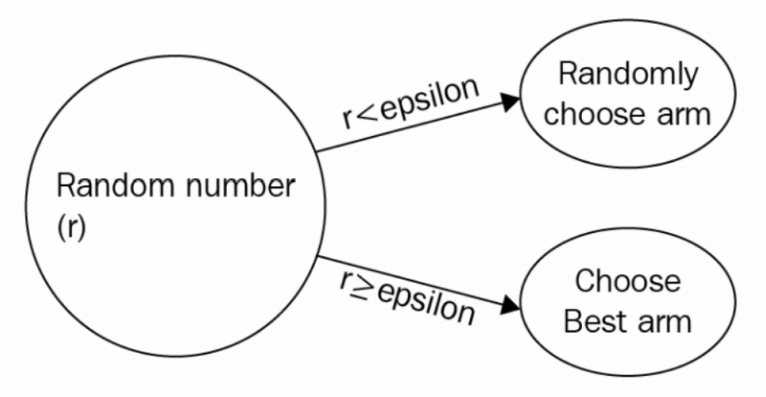
Multi-Armed Bandit Problem

Multi-armed bandit problem is a classical problem that models an agent (or planner or center) who wants to maximize its total reward by which it simultaneously desires to acquire new knowledge("exploration") and optimize his or her decisions based on existing knowledge("exploitation"). MAB problem captures the scenario where the gambler is faced with a trade-off between exploration, pulling less explored arms optimistically in search of an arm with better reward, and exploitation, pulling the arm known to be best till the current time instant, in terms of yielding the maximum reward.

MAB for cold users

Our goal is to use different bandit algorithms to explore/ exploit optimal recommendations for the user. There are several MAB algorithms, each favoring exploitation over exploration to different degrees. Three of the most popular are Epsilon Greedy, Thompson Sampling, and Upper Confidence Bound 1 (UCB-1):

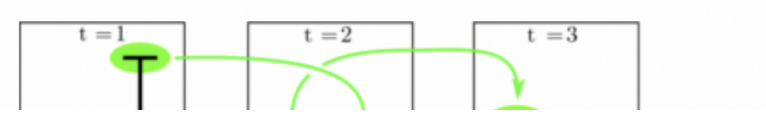
1. Epsilon Greedy

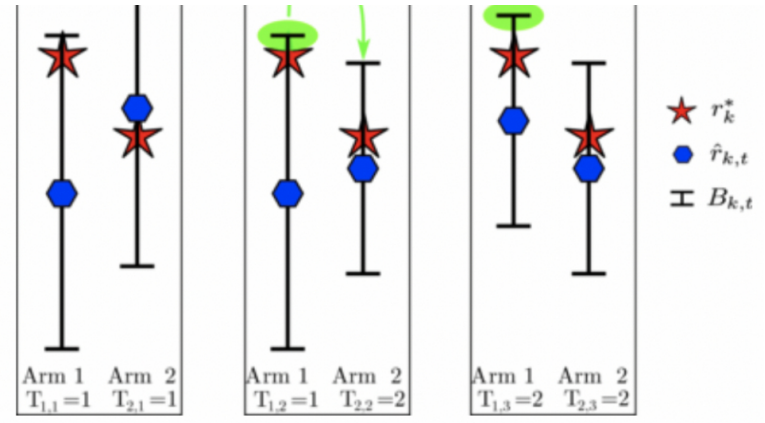


Epsilon Greedy Approach

Epsilon Greedy, as the name suggests, is the greediest of the three MAB algorithms. In Epsilon Greedy experiments, the constant ε (valued between 0 and 1) is selected by the user before the experiment starts. When allocating contacts to different variants of the campaign, a randomly chosen variant is picked ε of the time. The other 1-ε of the time, the variant with highest known payoff is chosen. The higher ε is, the more this algorithm favors exploration. For all our examples, ε is set to 0.1.

2. Upper confidence bound (UCB)





Upper confidence bound approach

For each variant of the campaign, we will identify upper confidence bound (UCB) that represents our highest guess at the possible payoff for that variant. The algorithm will assign contacts to the variant with the highest UCB.

The UCB for each variant is calculated based on both the mean payoff of the variant, as well as the number of contacts allocated to the variant, with the equation:

$$\text{UCB} = \overline{x_j} + \sqrt{2\,log\ t/n_j},$$

where

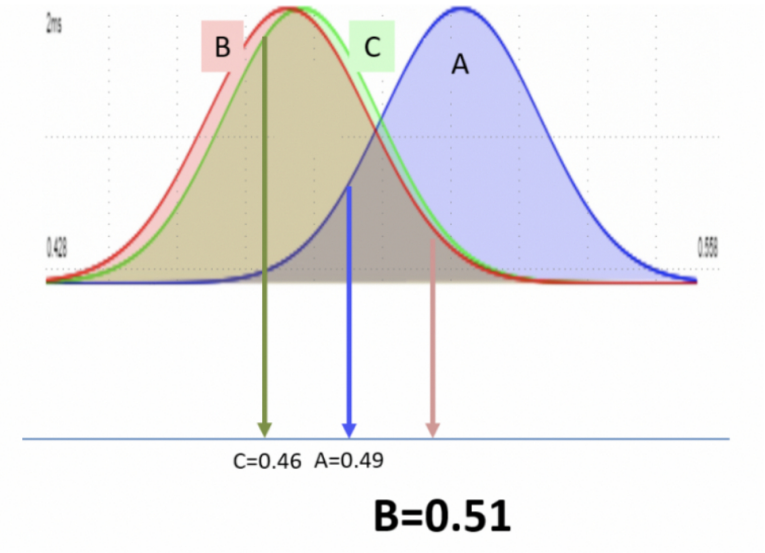
$\overline{x_j}$ = the average payoff at the jth step

t = total number of contacts that have entered the experiment

n_j = total number of contacts allocated to a particular variant

For a detailed description: https://www.youtube.com/watch?v=RPbtzWgzD9M

3. Thompson Sampling



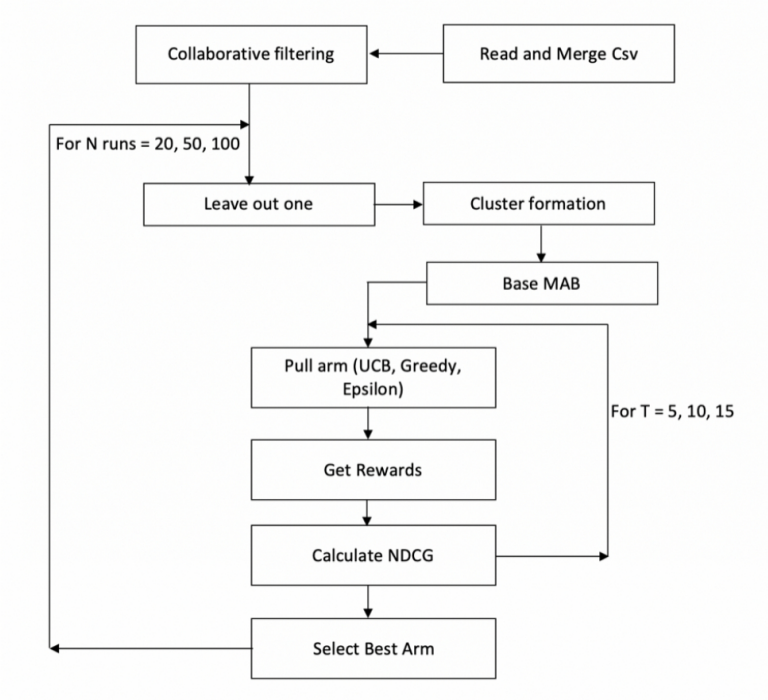
Thompson Sampling Approach

Thompson Sampling, by contrast, is more principled approach, which can yield more balanced results in marginal cases. For each variant, we build a probability distribution (most commonly a beta distribution, for

probability distribution (most commonly a beta distribution, for computational reasons) of the true success rate, using observed results. For each new contact, we sample one possible success rate from the beta distribution corresponding to each variant and assign the contact to the variant with the largest sampled success rate. The more data points we have observed, the more confident we will be about the true success rate, and so as we gather more data, the sampled success rates will be more and more likely to be close to the true rate.

For a detailed description: https://www.youtube.com/watch?v=p701cYQeqew

Methodology



Infrastructure for solving MAB cold start problem

We have used Movie Lens dataset for solving the MAB problem which contains 4 files. These files were merged and used for collaborative filtering.

	i_1	i_2	i_3	i_4	i_5	i_6	i_7
u_1	5	2	4	-	5	1	-
u_2	4	-	5	-	5	-	1
u_3	2	5	3	5	-	-	-
u_4	1	-	2	-	2	-	-
u_5	-	-	3	4	1	-	-

Sparse Matrix from User Ratings

	i_1	i_2	i_3	i_4	i_5	i_6	i_7
u_1	4.6	2.09	4.23	4.24	4.84	1.07	1.0
u_2	4.2	3.8	4.42	5.0	4.86	2.28	1.2
$\hat{\theta}_1$	4.4	2.94	4.32	4.62	4.85	1.67	1.1
u_3	1.97	4.84	3.22	4.87	2.68	2.68	1.61
u_4	1.19	3.24	2.17	3.56	1.92	1.23	1.0
u_5	1.77	3.16	2.81	4.07	1.14	1.6	1.56
$\hat{\theta}_2$	1.64	3.74	2.73	4.16	1.91	1.83	1.39

Matrix after application of Collaborative Filtering and Clustering

After getting the clusters from the above step, we sort them in descending order and recommend first the items with the highest ratings in each cluster which can contribute to learning users preference faster. For example, the correspondent θ1 in above matrix will have the sorted list of items equal to {i5,i4,i1,i3,i2,i6,i7}, while for θ2 we will have {i4,i2,i3,i5, i6,i1,i7}.

This approach is applied to make recommendations to users that are initially new users who remain cold until they have provided a certain amount of preferences/feedback on recommended items. The threshold to determine when a user is not cold anymore can be distinct. In this blog, we measure the performance after making 5, 10, 15, 40, 100 recommendations using NDCG metric (Size of ranked list). As soon as the user is not cold anymore the system can switch to a personalized prediction model.

Reward Function

$$X_{M_s}(t) = \frac{r_{u,i}}{r_{max}}$$

Reward function

The reward function is defined as the ratio of feedback of the user provided to the maximum rating that the user provided. In this way, when a user gives higher feedback for a recommended item, we will have a high reward. For example, suppose a cold user enters the system and using UCB algorithm, item i5 is selected from θ1. If the user gives feedback of 4, then the reward for this movie would be 4/5 = 0.8 for cluster θ1. In the next recommendation, if cluster θ2 is chosen, item i4 is selected and it receives feedback of 1 then the reward will be 1/5 = 0.2 for cluster θ2. At this point of time, the mean reward for cluster θ1 is 0.8 and for cluster θ2 is 0.2. The next recommendation would be i1 from θ1 which represents the arm with the highest mean reward at that moment.

Metric for Evaluation

We will be using NDCG (Normalized Discounted Cumulative Gain) which values accurate recommendations in earlier round more than later ones. Since we focus on recommendations for cold users, we value accurate results

in earlier rounds.

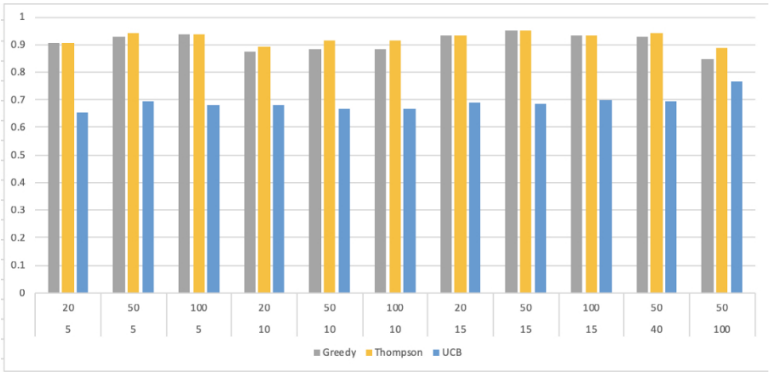
$$DCG(u) = r_{u,1} + \sum_{t=2}^{T} \frac{r_{u,t}}{\log_2 t} \qquad NDCG = \frac{1}{N} \sum_{u} \frac{DCG(u)}{DCG^*(u)}$$

where

> *DCG(u): Discounted cumulative gain of predicted ranking for a target user u*
>
> *DCG∗(u): Ground truth*
>
> *N: Number of users in the result set*
>
> *r (u,1): Rating (according to user feedback) of the item first recommended for user u*
>
> *r (u, t): User feedback for the item recommended in turn t*
>
> *t: Recommendation time*
>
> *T: Size of the ranked list*

Results

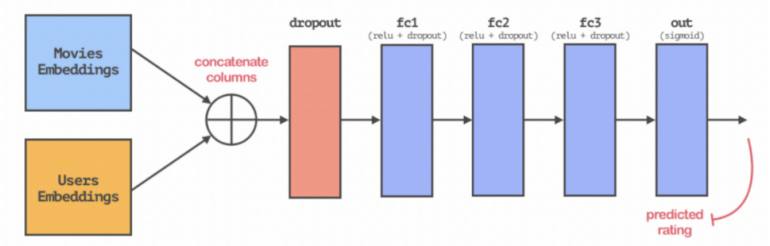
The table below provides the NDCG score for the rank-size of 5, 10, 15, 40 and 100. Thompson consistently performed better than Greedy and UCB. It took less time to warm up and was able to provide better results quickly. UCB, on the other hand, performed the worst for different values of N and T.



NDCG Results for varying rank-size and number of users

Embedding Networks

While embedding networks are typically used in a different problem, we wanted to see their effectiveness with the cold-start problem. Embedding networks are neural networks that learn embeddings between features. With this knowledge, it should be able to fill in empty values. So for the Movie Lens dataset, it can embed the relations between users and movies, and can then fill in the missing ratings that a user did not provide for a movie. While we just performed this task with collaborative filtering, using a neural network allows for the learning of complex, non-linear features.



Embedding network

So, an embedding network like one shown above can be used to fill in the missing values of an embedding below:

Users vs Movies

	A Game of Thrones	The Lord of The Rings	Star Trek	Star Wars	Titanic
Alice	5	4	3	3	5
Bob	5	n/a	4	5	3
Carol	n/a	3	5	n/a	2
Danny	4	4	5	3	4
Eve	5	3	?	3	5

Generally, embedding networks are used for recommendation systems with warm users. This way, the network does not have to guess as much. With warm users, the learned relations are hopefully well-established and not as subject to change. With this setting, an embedded neural network can train once on a dense dataset, learn very complex relations, and give highly educated recommendations. However, our problems do not have the luxury of warm-users that result in a large dataset.

With the cold-start problem, we have to give good recommendations to users that have provided no feedback. Essentially, their data is empty, and there aren't any relations that can be inferred for this user. The naive solution for using an embedding network for this problem could be to first give a random recommendation, and for every following recommendation: take feedback, retrain, sort the predicted ratings, and return the highest projected recommendation. However, this would result in insane computational costs and delayed recommendations, which could provide disastrous for the user experience.

We had the idea that even though an embedding network has no knowledge of a cold-user, the networks knowledge of datasets relations among users and items could still provide some value. So, we wanted to analyze performance if, for every user request, we considered multiple recommendations. These recommendations could come from global recommenders such as random, global average, or most popular. They could also come from MAB algorithms such as Thompson, Epsilon-greedy, and UCB. Instead of directly feeding back a single algorithms recommendation, we could first let the embedded network estimate which of these recommendations would score highest. The highest projected recommendation would then be given to the user. Depending on computation and timing constraints, the network can be retrained on the user's feedback.

This idea was tested in a toy example reusing the MAB evaluation framework described in previous sections. Due to time constraints, the embedded network received hardly any training (5 epochs), and no hyperparameter tuning. It was also not retrained on user feedback. For each user request, it considered 3 random recommendations and gave to the user the one that it

considered 5 random recommendations and gave to the user the one that it thought would score highest. However, it gave better results than UCB algorithm in this small test with 15 trials and 5 random users. One must note that strictly using random recommendations yields better results from this limited test shown below:

T	N	Random	Embedded NN	UCB
5	15	0.6933	0.6744	
5	20	0.7320		0.6555

While the results are not ground-breaking, it should be noted that further experiments have the opportunity to show much greater performance. More training, actually tuning the hyperparameters, retraining the network on the user feedback, and using other sampling recommenders (rather than just random) all offer the potential to greatly boost performance.

Conclusion and Future work

Although cold-start users in recommendation system pose a unique problem due to lack of knowledge about the user, MAB has done a pretty good job in recommending movies and is constantly evolving with data inputs. To sum up, our contributions are:

> *A formalization of the model selection as a multi-armed bandit problem*
>
> *Using UCB, Thompson Sampling and Epsilon greedy which is an effective approach to recommend for users without prior side information*
>
> *An empirical evaluation using NDCG on Movie Lens dataset*
>
> *Evaluating the effectiveness of Embedding network on a Recommendation system*

For future work, we can try

> *Having different bandit algorithms as the arms as we see epsilon-greedy performs better at the very start while Thompson performs better at more iterations*
>
> *Expand this implementation to provide multiple recommendations per iteration — currently, the best movie is recommended over and over*
>
> *We rely on NDCG score to access our approach, mainly because we were investigating the recommendation quality. Overall it presents high accuracy levels, but we might check other metrics to ensure a fair evaluation*
>
> *Tuning the hyperparameters of the Embedding network, retraining the network on the user feedback, and using other sampling recommenders (rather than just random)*

Thank you very much for reading, and we hope that you learned something from the project!

Feel Free to check out:

Github Repository of this post

My Other Medium Posts

My Linkedin Profile

References

https://medium.com/datadriveninvestor/how-to-built-a-recommender-system-rs-616c988d64b2

https://hal.inria.fr/hal-01517967/document

http://klerisson.github.io/papers/umap2017.pdf

https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-recommendation-engine-python/

https://towardsdatascience.com/various-implementations-of-collaborative-filtering-100385c6dfe0

https://medium.com/the-andela-way/foundations-of-machine-learning-singular-value-decomposition-svd-162ac796c27d

https://sebastianraschka.com/Articles/2014_pca_step_by_step.html

https://medium.com/datadriveninvestor/how-to-built-a-recommender-system-rs-616c988d64b2

https://medium.com/@iliazaitsev/how-to-implement-a-recommendation-system-with-deep-learning-and-pytorch-2d40476590f9

https://github.com/devforfu/pytorch_playground/blob/master/movielens.ipynb

https://nipunbatra.github.io/blog/2017/recommend-keras.html

https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526

https://medium.com/heycar/neural-network-embeddings-from-inception-to-simple-35e36cb0c173

Recommended from Medium

Susan Li in Towards Data Science

Practical Statistics & Visualization With Python & Plotly

Vaishnavi Upadhyay

All about Internet of Behaviour

Sophie Li

Assessing the effects of intergroup competition on tree growth

YS Chng in DataSeries

Getting to Know Big Data: In the Language of Beginners

Vahid Naghshin in Analytics Vidhya

Variable Importance with Tree Models & Random Forest—With Python

Wes Swager

Women's Soccer Expected Goals—Data Extraction

Jack Chih-Hs... in Towards Data S...

Optimize PyTorch Performance for Speed and Memory Efficiency (2022)

Arun Karnik in astringe

Data science reality vs. expectations

12ft.io – Is the paywall gone? Yes / No

The Multi-Armed Bandit Problem and Its Solutions

January 23, 2018 · 10 min · Lilian Weng

The algorithms are implemented for Bernoulli bandit in lilianweng/multi-armed-bandit.

Exploitation vs Exploration

The exploration vs exploitation dilemma exists in many aspects of our life. Say, your favorite restaurant is right around the corner. If you go there every day, you would be confident of what you will get, but miss the chances of discovering an even better option. If you try new places all the time, very likely you are gonna have to eat unpleasant food from time to time. Similarly, online advisors try to balance between the known most attractive ads and the new ads that might be even more successful.

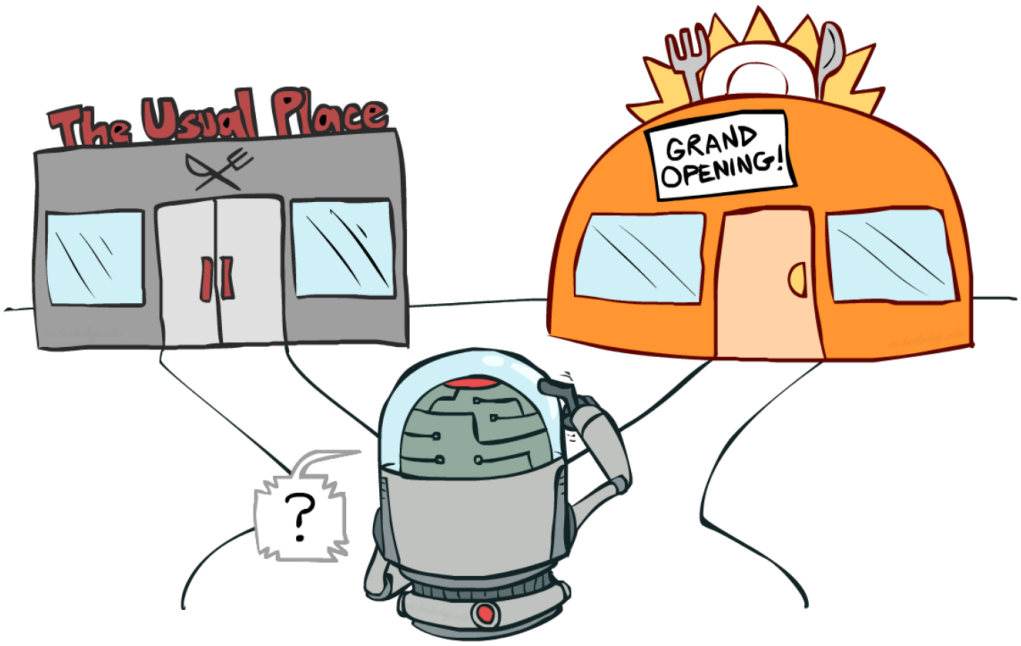


Fig. 1. A real-life example of the exploration vs exploitation dilemma: where to eat? (Image source: UC Berkeley AI course slide, lecture 11.)

If we have learned all the information about the environment, we are able to find the best strategy by even just simulating brute-force, let alone many other smart approaches. The dilemma comes from the *incomplete* information: we need to gather enough information to make best overall decisions while keeping the risk under control. With exploitation, we take advantage of the best option we know. With exploration, we take some risk to collect information about unknown options. The best long-term strategy may involve short-term sacrifices. For example, one exploration trial could be a total failure, but it warns us of not taking that action too often in the future.

What is Multi-Armed Bandit?

The multi-armed bandit problem is a classic problem that well demonstrates the exploration vs exploitation dilemma. Imagine you are in a casino facing multiple slot machines and each is configured with an unknown probability of how likely you can get a reward at one play. The question is: *What is the best strategy to achieve highest long-term rewards?*

In this post, we will only discuss the setting of having an infinite number of trials. The restriction on a finite number of trials introduces a new type of exploration problem. For instance, if the number of trials is smaller than the number of slot machines, we cannot even try every machine to estimate the reward probability (!) and hence we have to behave smartly w.r.t. a limited set of knowledge and resources (i.e. time).

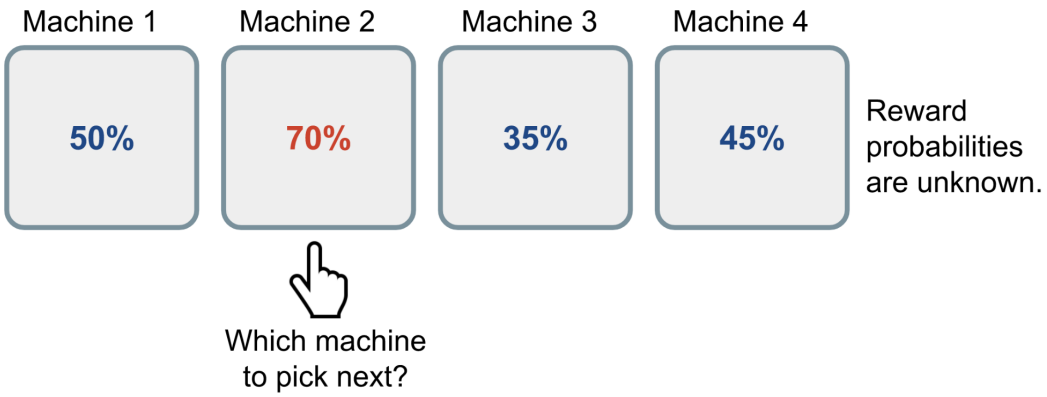


Fig. 2. An illustration of how a Bernoulli multi-armed bandit works. The reward probabilities are **unknown** to the player.

A naive approach can be that you continue to playing with one machine for many many rounds so as to eventually estimate the "true" reward probability according to the law of large numbers. However, this is quite wasteful and surely does not guarantee the best long-term reward.

Definition

Now let's give it a scientific definition.

A Bernoulli multi-armed bandit can be described as a tuple of $\langle \mathcal{A}, \mathcal{R} \rangle$, where:

- We have K machines with reward probabilities, $\{\theta_1, \dots, \theta_K\}$.
- At each time step t, we take an action a on one slot machine and receive a reward r.
- $\mathcal{A}$ is a set of actions, each referring to the interaction with one slot machine. The value of action a is the expected reward, $Q(a) = \mathbb{E}[r|a] = \theta$. If action a_t at the time step t is on the i-th machine, then $Q(a_t) = \theta_i$.
- $\mathcal{R}$ is a reward function. In the case of Bernoulli bandit, we observe a reward r in a *stochastic* fashion. At the time step t, $r_t = \mathcal{R}(a_t)$ may return reward 1 with a probability $Q(a_t)$ or 0 otherwise.

It is a simplified version of Markov decision process, as there is no state $\mathcal{S}$.

The goal is to maximize the cumulative reward $\sum_{t=1}^{T} r_t$. If we know the optimal action with the best reward, then the goal is same as to minimize the potential regret or loss by not picking the optimal action.

The optimal reward probability θ^* of the optimal action a^* is:

$$\theta^* = Q(a^*) = \max_{a \in \mathcal{A}} Q(a) = \max_{1 \le i \le K} \theta_i$$

Our loss function is the total regret we might have by not selecting the optimal action up to the time step T:

$$\mathcal{L}_T = \mathbb{E}\Big[\sum_{t=1}^{T}\big(\theta^* - Q(a_t)\big)\Big]$$

Bandit Strategies

Based on how we do exploration, there several ways to solve the multi-armed bandit.

- No exploration: the most naive approach and a bad one.
- Exploration at random
- Exploration smartly with preference to uncertainty

ε-Greedy Algorithm

The ε-greedy algorithm takes the best action most of the time, but does random exploration occasionally. The action value is estimated according to the past experience by averaging the rewards associated with the target action a that we have observed so far (up to the current time step t):

$$\hat{Q}_t(a) = \frac{1}{N_t(a)}\sum_{\tau=1}^{t} r_\tau \mathbb{1}[a_\tau = a]$$

where $\mathbb{1}$ is a binary indicator function and $N_t(a)$ is how many times the action a has been selected so far, $N_t(a) = \sum_{\tau=1}^{t}\mathbb{1}[a_\tau = a]$.

According to the ε-greedy algorithm, with a small probability ϵ we take a random action, but otherwise (which should be the most of the time, probability 1-ϵ) we pick the best action that we have learnt so far: $\hat{a}_t^* = \arg\max_{a\in\mathcal{A}}\hat{Q}_t(a)$.

Check my toy implementation here.

Upper Confidence Bounds

Random exploration gives us an opportunity to try out options that we have not known much about. However, due to the randomness, it is possible we end up exploring a bad action which we have confirmed in the past (bad luck!). To avoid such inefficient exploration, one approach is to decrease the parameter ε in time and the other is to be optimistic about options with *high uncertainty* and thus to prefer actions for which we haven't had a confident value estimation

yet. Or in other words, we favor exploration of actions with a strong potential to have a optimal value.

The Upper Confidence Bounds (UCB) algorithm measures this potential by an upper confidence bound of the reward value, $\hat{U}_t(a)$, so that the true value is below with bound $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$ with high probability. The upper bound $\hat{U}_t(a)$ is a function of $N_t(a)$; a larger number of trials $N_t(a)$ should give us a smaller bound $\hat{U}_t(a)$.

In UCB algorithm, we always select the greediest action to maximize the upper confidence bound:

$$a_t^{UCB} = argmax_{a \in \mathcal{A}} \hat{Q}_t(a) + \hat{U}_t(a)$$

Now, the question is *how to estimate the upper confidence bound*.

Hoeffding's Inequality

If we do not want to assign any prior knowledge on how the distribution looks like, we can get help from "Hoeffding's Inequality" — a theorem applicable to any bounded distribution.

Let $X_1, \ldots, X_t$ be i.i.d. (independent and identically distributed) random variables and they are all bounded by the interval [0, 1]. The sample mean is $\overline{X}_t = \frac{1}{t} \sum_{\tau=1}^{t} X_\tau$. Then for u > 0, we have:

$$\mathbb{P}[\mathbb{E}[X] > \overline{X}_t + u] \leq e^{-2tu^2}$$

Given one target action a, let us consider:

- $r_t(a)$ as the random variables,
- $Q(a)$ as the true mean,
- $\hat{Q}_t(a)$ as the sample mean,
- And u as the upper confidence bound, $u = U_t(a)$

Then we have,

$$\mathbb{P}[Q(a) > \hat{Q}_t(a) + U_t(a)] \leq e^{-2tU_t(a)^2}$$

We want to pick a bound so that with high chances the true mean is blow the sample mean + the upper confidence bound. Thus $e^{-2tU_t(a)^2}$ should be a small probability. Let's say we are ok with a tiny threshold p:

$$e^{-2tU_t(a)^2} = p \text{ Thus, } U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

UCB1

One heuristic is to reduce the threshold p in time, as we want to make more confident bound estimation with more rewards observed. Set $p = t^{-4}$ we get **UCB1** algorithm:

$$U_t(a) = \sqrt{\frac{2\log t}{N_t(a)}} \text{ and } a_t^{UCB1} = \arg\max_{a \in \mathcal{A}} Q(a) + \sqrt{\frac{2\log t}{N_t(a)}}$$

Bayesian UCB

In UCB or UCB1 algorithm, we do not assume any prior on the reward distribution and therefore we have to rely on the Hoeffding's Inequality for a very generalize estimation. If we are able to know the distribution upfront, we would be able to make better bound estimation.

For example, if we expect the mean reward of every slot machine to be Gaussian as in Fig 2, we can set the upper bound as 95% confidence interval by setting $\hat{U}_t(a)$ to be twice the standard deviation.

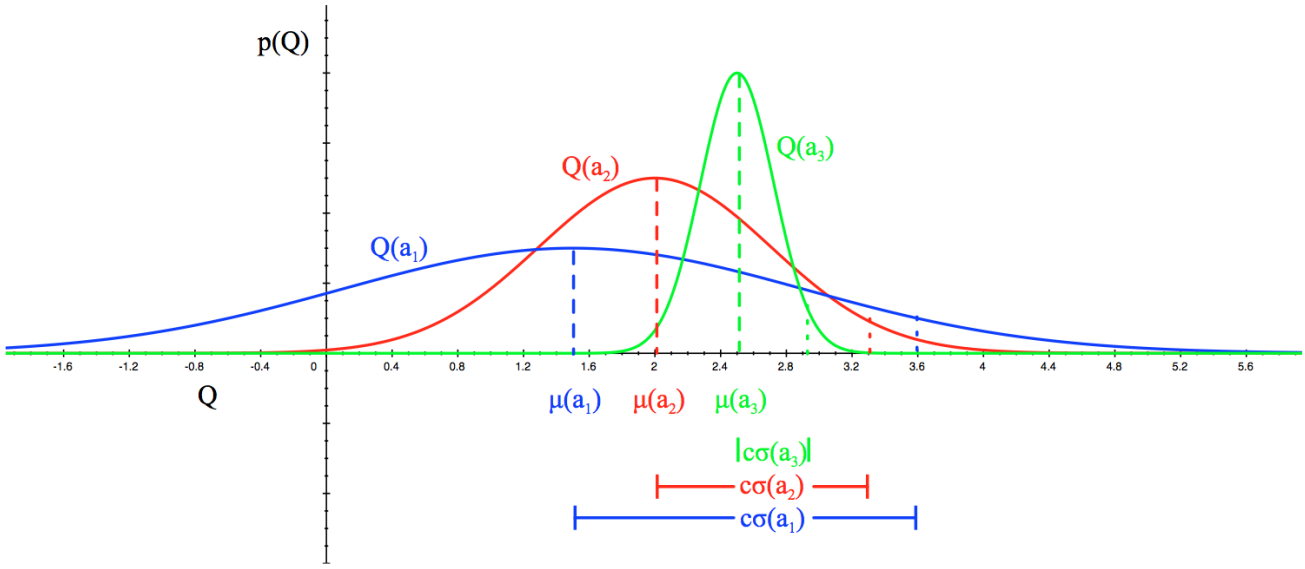


Fig. 3. When the expected reward has a Gaussian distribution. $\sigma(a\_i)$ is the standard deviation and $c\sigma(a\_i)$ is the upper confidence bound. The constant c is a adjustable hyperparameter. (Image source: UCL RL course lecture 9's slides)

Check my toy implementation of UCB1 and Bayesian UCB with Beta prior on θ.

Thompson Sampling

Thompson sampling has a simple idea but it works great for solving the multi-armed bandit problem.

Fig. 4. Oops, I guess not this Thompson? (Credit goes to Ben Taborsky; he has a full theorem of how Thompson invented while pondering over who to pass the ball. Yes I stole his joke.)

At each time step, we want to select action a according to the probability that a is **optimal**:

$$\begin{aligned}\pi(a \mid h_t) &= \mathbb{P}[Q(a) > Q(a'), \forall a' \neq a \mid h_t] \\ &= \mathbb{E}_{\mathcal{R}|h_t}[\mathbb{1}(a = \arg\max_{a \in \mathcal{A}} Q(a))]\end{aligned}$$

where $\pi(a; |; h_t)$ is the probability of taking action a given the history h_t.

For the Bernoulli bandit, it is natural to assume that $Q(a)$ follows a Beta distribution, as $Q(a)$ is essentially the success probability θ in Bernoulli distribution. The value of $\text{Beta}(\alpha, \beta)$ is within the interval [0, 1]; α and β correspond to the counts when we **succeeded** or **failed** to get a reward respectively.

First, let us initialize the Beta parameters α and β based on some prior knowledge or belief for every action. For example,

- α = 1 and β = 1; we expect the reward probability to be 50% but we are not very confident.
- α = 1000 and β = 9000; we strongly believe that the reward probability is 10%.

At each time t, we sample an expected reward, $\tilde{Q}(a)$, from the prior distribution $\text{Beta}(\alpha_i, \beta_i)$ for every action. The best action is selected among samples: $a_t^{TS} = \arg\max_{a \in \mathcal{A}} \tilde{Q}(a)$. After the true reward is observed, we can update the Beta distribution accordingly, which is essentially doing Bayesian inference to compute the posterior with the known prior and the likelihood of getting the sampled data.

$$\begin{aligned}\alpha_i &\leftarrow \alpha_i + r_t \mathbb{1}[a_t^{TS} = a_i] \\ \beta_i &\leftarrow \beta_i + (1 - r_t) \mathbb{1}[a_t^{TS} = a_i]\end{aligned}$$

Thompson sampling implements the idea of probability matching. Because its reward estimations $\tilde{Q}$ are sampled from posterior distributions, each of these probabilities is equivalent to the probability that the corresponding action is optimal, conditioned on observed history.

However, for many practical and complex problems, it can be computationally intractable to estimate the posterior distributions with observed true rewards using Bayesian inference. Thompson sampling still can work out if we are able to approximate the posterior distributions using methods like Gibbs sampling, Laplace approximate, and the bootstraps. This tutorial presents a comprehensive review; strongly recommend it if you want to learn more about Thompson sampling.

Case Study

I implemented the above algorithms in lilianweng/multi-armed-bandit. A BernoulliBandit object can be constructed with a list of random or predefined reward probabilities. The bandit algorithms are implemented as subclasses of Solver, taking a Bandit object as the target problem. The cumulative regrets are tracked in time.

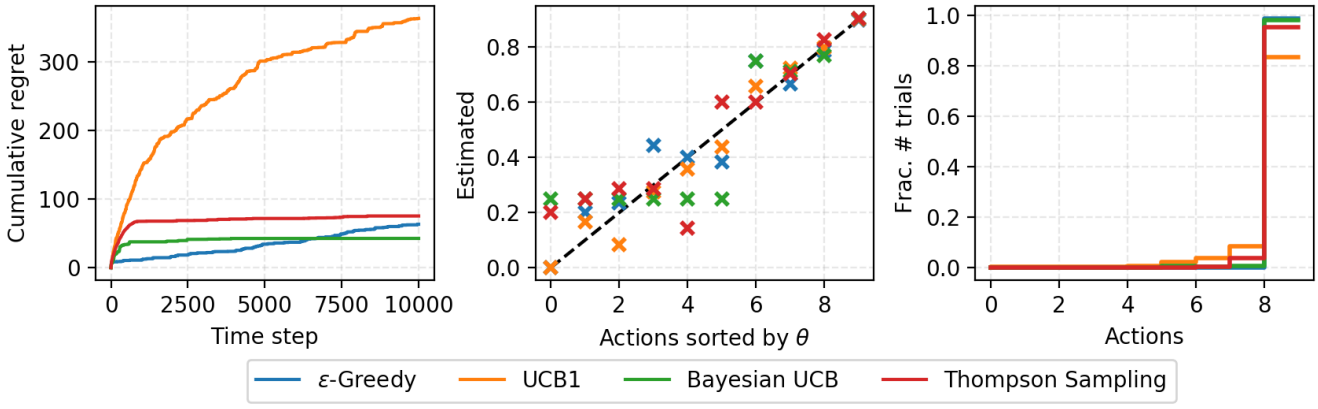


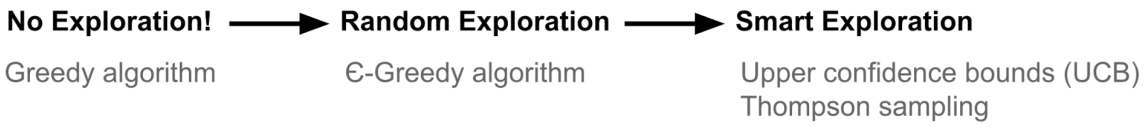
Fig. 4. The result of a small experiment on solving a Bernoulli bandit with K = 10 slot machines with reward probabilities, {0.0, 0.1, 0.2, ..., 0.9}. Each solver runs 10000 steps.

(Left) The plot of time step vs the cumulative regrets. (Middle) The plot of true reward probability vs estimated probability. (Right) The fraction of each action is picked during the 10000-step run.*

Summary

We need exploration because information is valuable. In terms of the exploration strategies, we can do no exploration at all, focusing on the short-term returns. Or we occasionally explore

at random. Or even further, we explore and we are picky about which options to explore — actions with higher uncertainty are favored because they can provide higher information gain.



Cited as:

```
@article{weng2018bandit,
  title   = "The Multi-Armed Bandit Problem and Its Solutions",
  author  = "Weng, Lilian",
  journal = "lilianweng.github.io",
  year    = "2018",
  url     = "https://lilianweng.github.io/posts/2018-01-23-multi-armed-bandit/"
}
```

References

[1] CS229 Supplemental Lecture notes: Hoeffding's inequality.

[2] RL Course by David Silver - Lecture 9: Exploration and Exploitation

[3] Olivier Chapelle and Lihong Li. "An empirical evaluation of thompson sampling." NIPS. 2011.

[4] Russo, Daniel, et al. "A Tutorial on Thompson Sampling." arXiv:1707.02038 (2017).

reinforcement-learning exploration math-heavy

« A (Long) Peek into Reinforcement Learning

» Object Detection for Dummies Part 3: R-CNN Family