Fighting boredom in recommender system with graph and reinforcement learning

Antoine Yang
Ecole Polytechnique - ENS Paris-Saclay
antoineyang3@gmail.com

Abstract

Using always the same system to perform recommendation may produce non stationary performance due to boredom (or more complex evolution) of the user interest. To fight that, reinforcement learning techniques have been proposed to alternate between several systems at the right cadence. However, these solutions learn the same policy for all users whereas we can expect that each user has his own boredom evolution. In this work, we propose to represent users in a graph, and to use the graph information to personalize the model to find how to alternate among different recommender systems for each user.

1 Introduction

A classic approach for recommendation systems consists in estimating user's preferences and recommend products of preferred categories. However, at some point, the user could be bored and appreciate less recommended products, while its preferences would remain unchanged.

The popular multi-armed bandit algorithms [1] effectively trade off exploration and exploitation in unknown environments, but still assume that rewards are independent from the sequence of arms selected over time and try to select the optimal arm as often as possible. Even the advanced A/B testing implicitly assumes that once the better option is found, it should be constantly executed. An alternative approach is to estimate the boredom of the user given the previous states and learn a contextual strategy that recommends the best product category depending on the actual boredom of the user (e.g., using LINUCB [2]), but it would not optimize cumulative reward on the long term.

LinUCRL [3] proposes a promising solution to this problem, building on UCRL [4], using the linear assumption and defining states that summarize the effect of the recent recommendations on user's preferences. However, despite strong theoretical results, experiments still show there is room for improvement as convergence is reached after approximatively 200 ratings on the movielens dataset. Furthermore, the policy is learnt the same way for all users whereas we can expect that each user has his own boredom evolution.

A Gang of Bandits [5] proposes to exploit network informations in a contextual bandit framework, allocating a bandit algorithm to each network node (user) and allowing it to share signals (contexts and payoffs) with the neighboring nodes. This could both speed up convergence and help personalize the policy learning to each user.

The goal of this project is to jointly use network informations as in Gang of Bandits and LinUCRL to propose and implement an algorithm (which we will call GoLinUCLR) of which the performances will be explored experimentally.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

2 GoLinUCRL

In this section, we introduce the algorithm GoLinUCRL that we propose.

2.1 Problem formulation

In this subsection, we introduce the bandit problem considered in LinUCRL [3].

We consider a finite set of actions $a \in \{1, ..., A\} = [A]$, which correspond to movie genre recommendations. We define the state s_t at time t as the history of last w actions i.e. $s_t = (a_{t-1}, ..., a_{t-w})$. We introduce the recency function $\rho(s_t, a) = \sum_{\tau=1}^{w} \frac{\mathbb{1}(a_{t-\tau}=a)}{\tau}$ where the effect of an action fades as the inverse function: the more often an action is selected, the higher the recency function. We define the context vector associated to action a in state s as the power iterates of the recency function $x_{s,a} = \left[1, \rho(s, a), ..., \rho(s, a)^d\right] \in \mathbb{R}^{d+1}$.

In the linear setting, we can write the reward $r_t = r(s_t, a) + \epsilon_t$, where $r(s_t, a) = x_{s,a}^T \theta_a^*$ and ϵ_t is a zero-mean noise. The difference between a classical linear bandit problem is that here, the context $x_{s_t,a}$ depends on the state s_t and therefore on the last w actions.

We introduce the Markov Decision Process (MDP) $M =< S, [A], f, r >$. Given S, the transition function $f : S \times [A] \to S$ just drops the action selected w times ago and appends the last action to the state. In our case, the policy $\pi : S \to [A]$ is evaluated according to its long-term average reward $\eta^\pi = \lim_{n \to +\infty} \mathbb{E}[\sum_{t=1}^{n} r_t / n]$. Thus, the optimal policy is $\pi^* = argmax_\pi \eta^\pi$ and the optimal average reward is $\eta^* = \eta^{\pi^*}$. Therefore the regret is defined as $\Delta(T) = T\eta^* - \sum_{t=1}^{T} r(s_t, a_t)$ where (s_t, a_t) is the sequence of states and actions observed and selected by the algorithm.

2.2 LinUCRL algorithm

LinUCLR is inspired by [4] and exploits the linear structure of the reward function and the deterministic and known transition function.

LinUCRL uses the current sample collected for each action a separately ie $x_{s_{\tau,a}\theta}$ context vector corresponding to state s_τ and r_τ reward observed at time τ to compute an estimate $\hat{\theta}_{t,a} = min_\theta \sum_{\tau<t:a_\tau=a} (x_{s_{\tau,a}\theta}^T - r_\tau)^2 + \lambda \|\theta\|_2$. Let $R_{a,t}$ vector of rewards obtained up to time t when a was executed and $X_{a,t}$ the feature matrix corresponding to the contexts observed so far, then $V_{t,a} = (X_{t,a}^T X_{t,a} + \lambda I) \in \mathbb{R}^{(d+1)\times(d+1)}$ is the design matrix.

Then the closed form solution of the estimate can be written $\hat{\theta}_{t,a} = V_{t,a}^{-1} X_{t,a}^T R_{a,t}$ and the estimated reward function $\hat{r}_t(s, a) = x_{s,a}^T \hat{\theta}_{t,a}$. The upper-confidence bound is $\widetilde{r}_t(s, a) = \hat{r}_t(s, a) + c_{t,a} \|x_{s,a}\|_{V_{t,a}^{-1}}$ where $c_{t,a}$ is a scaling factor defined as: $c_{t,a} = R\sqrt{(d+1) log(Kt^\alpha(1 + \frac{T_{t,a} L_w^2}{\lambda}))} + \lambda^{1/2} B$.

LinUCRL iteratively update its $\hat{\theta}$ estimate and estimate the optimal optimistic policy $\widetilde{\pi}_k$ of the MDP $\widetilde{M}_k =< S, [A], f, \widetilde{r}_k >$ via a value iteration scheme $u_{i+1}(s) = max_{a \in [A]}[r(s, a) + u_i(f(s, a))]$.

2.3 GoLinUCRL

We now move on our proposed algorithm GoLinUCRL. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with a set of n nodes $\mathcal{V}$ and a set of edges $\mathcal{E}$ representing users. At each node i of $\mathcal{G}$, we run a linear bandit as in the LinUCRL algorithm. Our goal is to exploit the graph information via the Laplacian matrix L associated with $\mathcal{G}$ to propagate signals between close users and learn their personalized preferences faster.

As in [5], for each node i, each action a, at state s, we construct from the context vectors sparse vectors $\Phi_i(x_{s,a}) = (0, ..., 0, x_{s,a}^T, 0, ..., 0)^T \in \mathbb{R}^{(d+1)n}$, and modified vectors $\widetilde{\Phi}_{s,a} = A_\otimes^{-1/2} \Phi_i(x_{s,a})$ where $A_\otimes = A \otimes Id$ where $A = I_n + L$.

These modified vectors have the same role as the context vectors in LinUCRL. Thus, the optimistic reward in GoLinUCRL is calculated as: $\widetilde{r}_k(s, a) = \widetilde{\Phi}_{s,a} \hat{\theta}_a + c_{t,a} \|\widetilde{\Phi}_{s,a}\|_{V_a^{-1}}$. A pseudo-code is proposed in Algorithm 1.

Algorithm 1 GoLinUCRL

Initialize:
  $t = 0_n, T_a = 0_n, \hat{\theta}_a = 0_{(d+1)n}, V_a = \lambda I_{(d+1)n}$
for round k=1,2 ... **do**
  Compute $\hat{\theta}_a = V_a^{-1} X_a^T R_a$
  Get node $i_k \in \mathcal{V}$
  Set $t_k = t_{i_k}, \nu_a = 0$
  Construct vectors $\Phi_{i_k}(x_{s_{i_k},a}) = (0, ..., 0, x_{s_{i_k},a}^T, 0, ..., 0)^T \in \mathbb{R}^{(d+1)n}$
  Construct modified vectors $\widetilde{\Phi}_{s_{i_k},a} = A_{\otimes}^{-1/2} \Phi_i(x_{s_{i_k},k})$
  Set optimistic reward $\widetilde{r}_k(s_{i_k}, a) = \widetilde{\Phi}_{s_{i_k},a} \hat{\theta}_a + c_{t_k,a} \|\widetilde{\Phi}_{s_{i_k},a}\|_{V_a^{-1}}$
  Compute optimal policy $\widetilde{\pi}_k$ for MDP $(S, [A], f, \widetilde{r}_k)$
  while $\forall a \in [A], \nu_a < T_{a,i_k}$ **do**
    Choose action $a_{t_k} = \widetilde{\pi}_k(s_{t_k})$
    Observe reward r_{t_k}, next state s_{t_k+1}
    Update $X_{a_{t_k}} \leftarrow [X_{a_{t_k}}, \widetilde{\Phi}_{s_{t_k},a_{t_k}}], R_{a_{t_k}} \leftarrow [R_{a_{t_k}}, r_{t_k}], V_{a_{t_k}} \leftarrow V_{a_{t_k}} + \widetilde{\Phi}_{s_{t_k},a_{t_k}} \widetilde{\Phi}_{s_{t_k},a_{t_k}}^T$
    Set $\nu_{a_{t_k}} \leftarrow \nu_{a_{t_k}} + 1, t_k \leftarrow t_k + 1$
  end while
  Set $T_{a_{i_k}} \leftarrow T_{a_{i_k}} + \nu_a, t_{i_k} \leftarrow t_k$
end for

3 Experiment

To test the proposed approach, I did experiments on the movielens 20M dataset. It includes 20 million ratings and $465,000$ tag applications applied to $27,000$ movies by $138,000$ users. I was provided code for LinUCRL and associated experiments by Romain Warlop and capitalized on it to implement GoLinUCRL and my experiments in about 800 lines of codes. Implementation includes an initial phase where we select at least all arms once for each meta-user.

To compare to LinUCRL, we will only recommend genres to users. To test this implementation, we consider a fixed number of users $C = 4$ and construct an oracle θ^* of dimension $d + 1$ where $d = 5$ for each user and each of the $G = 4$ genres recommended to construct a simulator to sample rewards afterwards. The window size is chosen as $w = 5$.

3.1 User clustering and Graph Construction

To compute the oracle, we first limit the study to the 255 users that had watched at least 2000 movies. Then we compute their average ratings for each of the G genres of movies and clustered them into C meta-users (groups of users) using K-means. This clustering could be improved for instance taking into account the number of ratings for the different genres and using it to weight the average ratings for the different genres.

Then, using as distance an inverse exponential of the euclidian distance between the average ratings per genre, we construct a k-NN graph with $k = 3$ (fully-connected). The goal is to prove that convergence is reached faster with this graph than with a graph without any edge (which is equivalent to running LinUCRL on each user).

3.2 Oracle Construction

For these C meta-users, for each genre, we collect rewards and recencies of movies watched in chronological order by users assigned to each meta-user. The oracle θ^* is then computed via Least-Squares estimation. This approach is detailed in Algorithm 2.

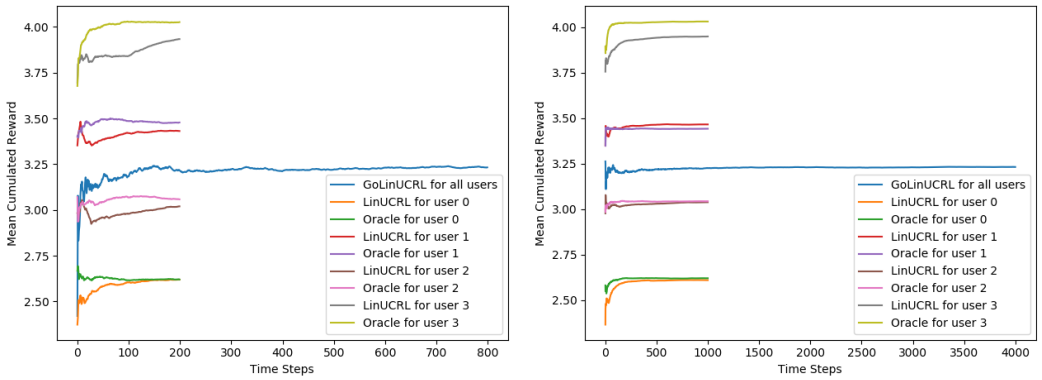
This oracle θ^* is used for simulations with GoLinUCRL and LinUCRL but is also used to compute an optimal policy baseline, computed via Value Iteration (using rewards without noise) for 100 steps. It is then applied for each user and referred as Oracle.

Algorithm 2 Oracle Construction

Initialize:
  $\theta^* = 0_{K \times G \times (d+1)}$
for cluster $c = 1, 2...C$ **do**
  Get users assigned to cluster c $users\_cluster$
  for genre $g = 1, 2...G$ **do**
    Set $rewards\_cluster\_genre = []$
    Set $recencies\_cluster\_genre = []$
    for user u in $users\_cluster$ **do**
      Get $ratings\_user$ movie ratings of the user sorted in chronological order
      Set $rewards\_user\_genre = []$
      Set $recencies\_user\_genre = []$
      Set $actions\_user = []$
      for $movie$ in $ratings\_user$ **do**
        Get $current\_genres$ genres of the movie
        if g in $current\_genres$ **then**
          Get as reward the rating of the movie $movie\_rating$
          $rewards\_user\_genre \leftarrow [rewards\_user\_genre, movie\_rating]$
          Compute $recency$ of genre g based on the last w actions
          $recencies\_user\_genre \leftarrow [recencies\_user\_genre, recency]$
        end if
        $actions\_user \leftarrow [actions\_user, current\_genres]$
      end for
      $rewards\_cluster\_genre \leftarrow [rewards\_cluster\_genre, rewards\_user\_genre]$
      $recencies\_cluster\_genre \leftarrow [recencies\_cluster\_genre, recencies\_user\_genre]$
    end for
    Compute $contexts\_cluster\_genre$ powers of elements of $recencies\_cluster\_genre$
    Compute $\theta^*[c][g] = LeastSquares(contexts\_cluster\_genre, rewards\_cluster\_genre)$
  end for
end for

3.3 Results

As a sanity check, the first thing we want to make sure is that if the graph is empty (no edges, $Laplacian = 0$), results obtained are similar to those of LinUCRL. For now, for the sake of simplicity, we restrict to $C = 4$, $G = 4$ using Actions, Thriller, Comedy and Romance movies. We run GoLinUCRL for $C = 4$ times longer than LinUCRL or the Oracle as it makes recommendations for all users at the same time.



(a) $T = 200$, one iteration

(b) $T = 1000$, 5 iterations

Figure 1: Sanity Check

In Figure 1, we can verify the convergence of the mean cumulated reward to a value that is consistent with the ones obtained by the Oracle or by LinUCRL: user 0 has an optimal reward of 2.7, user 1 has an optimal reward of 3, user 2 has an optimal reward of 3.5 and user 4 has an optimal reward of 4, while GoLinUCRL converges to 3.3 which is approximately the mean of the 4 values.

Additionally, we can study the mean cumulated reward for each of the users, and observe that GoLinUCRL performs on par or slightly better than LinUCRL with these parameters (cf Figure 2). However, GoLinUCRL performances are very sensible to the parameters controlling the construction of the graphs of users.

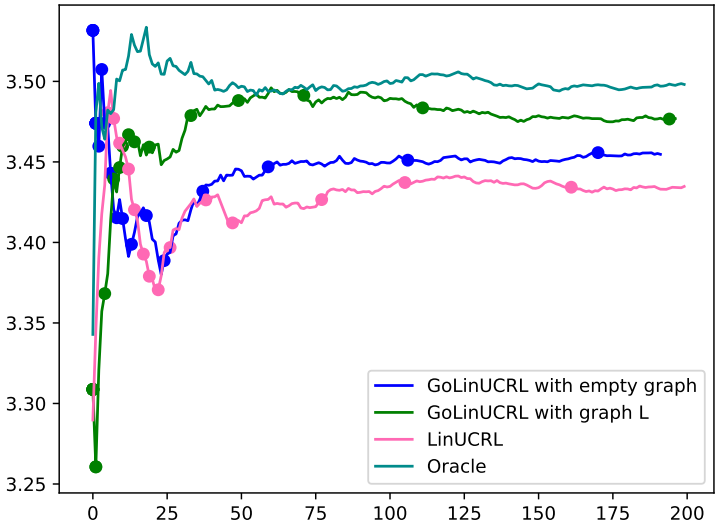


Figure 2: Mean accumulated reward evolution for one cluster

Finally, we can study the strategy proposed by each algorithm in the last 40 recommendations for each user. In some cases, GoLinUCRL is able to propose an alternated strategy that is closer to the optimal oracle than LinUCRL (cf Figure 3).

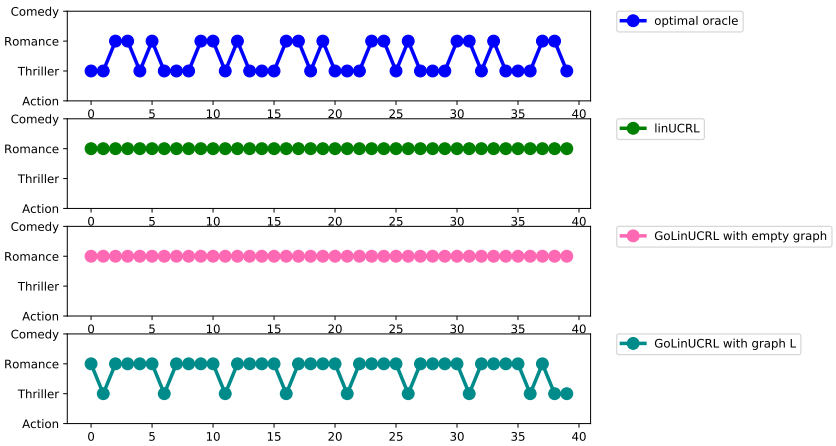


Figure 3: Strategies on the last steps for one cluster

4 Conclusion

A first phase of work was dedicated to the reading and understanding of different bandits algorithms, notably Gang of Bandits and LinUCRL. Then I implemented a GoLinUCRL algorithm building on these two frameworks to learn simultaneously the policies for several users. I then moved on experiments constructing an oracle and a graph of users, and evaluated GoLinUCRL performances quantitatively in terms of mean accumulated reward and qualitatively in terms of strategies found. This work could be naturally extended theoretically with an analysis of the regret of this algorithm, inspired by the theoretical analysis provided in Gang of Bandits and LinUCRL. Finally, I would like to thank Romain Warlop for code provided, discussions and help during this project.

References

[1] S. Bubeck and N. Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. pages 1–122. Foundations and Trends in Machine Learning, 2012.

[2] Y. Abbasi-Yadkori, D. Pál, and C. Szepesvári. Improved algorithms for linear stochastic bandits. pages 2312–2320. Advances in Neural Information Processing Systems, 2011.

[3] R. Warlop, A. Lazaric, and J. Mary. Fighting boredom in recommender systems with linear reinforcement learning. pages 1757–1768. Advances in Neural Information Processing Systems, 2018.

[4] P. Auer and R. Ortner. Logarithmic online regret bounds for undiscounted reinforcement learning. pages 49–56. Advances in Neural Information Processing Systems, 2007.

[5] N. Cesa-Bianchi, C. Gentile, and G. Zappella. A gang of bandits. pages 737–745. Advances in Neural Information Processing Systems, 2013.