Bandit算法与推荐系统 (/blog/CapStat/8484)

发表: 2017-06-06 浏览: 232

推荐系统 (https://ask.hellobi.com/topic/%E6%8E%A8%E8%8D%90%E7%B3%BB%E7%BB%9F)

推荐算法 (https://ask.hellobi.com/topic/%E6%8E%A8%E8%8D%90%E7%AE%97%E6%B3%95)

算法 (https://ask.hellobi.com/topic/%E7%AE%97%E6%B3%95)

推荐系统里面有两个经典问题，EE问题和冷启动问题，前者涉及到平衡准确和多样，后者涉及到产品算法运营等一系列东西。bandit算法是一种简单的在线学习算法，常常用于尝试解决这两个问题，本文为你介绍基础的bandit算法及一系列升级版，以及对推荐系统这两个经典问题的思考。

1.什么是bandit算法

1.1 为选择而生

我们会遇到很多选择的场景，上哪个大学，学什么专业，去哪家公司，中午吃什么，等等，这些事情，都让选择困难症的我们头很大。那么，有算法能够很好地对付这些问题吗？

当然有！那就是bandit算法！

编程 1
(/blog/CapStat/category/1955)

文章目录

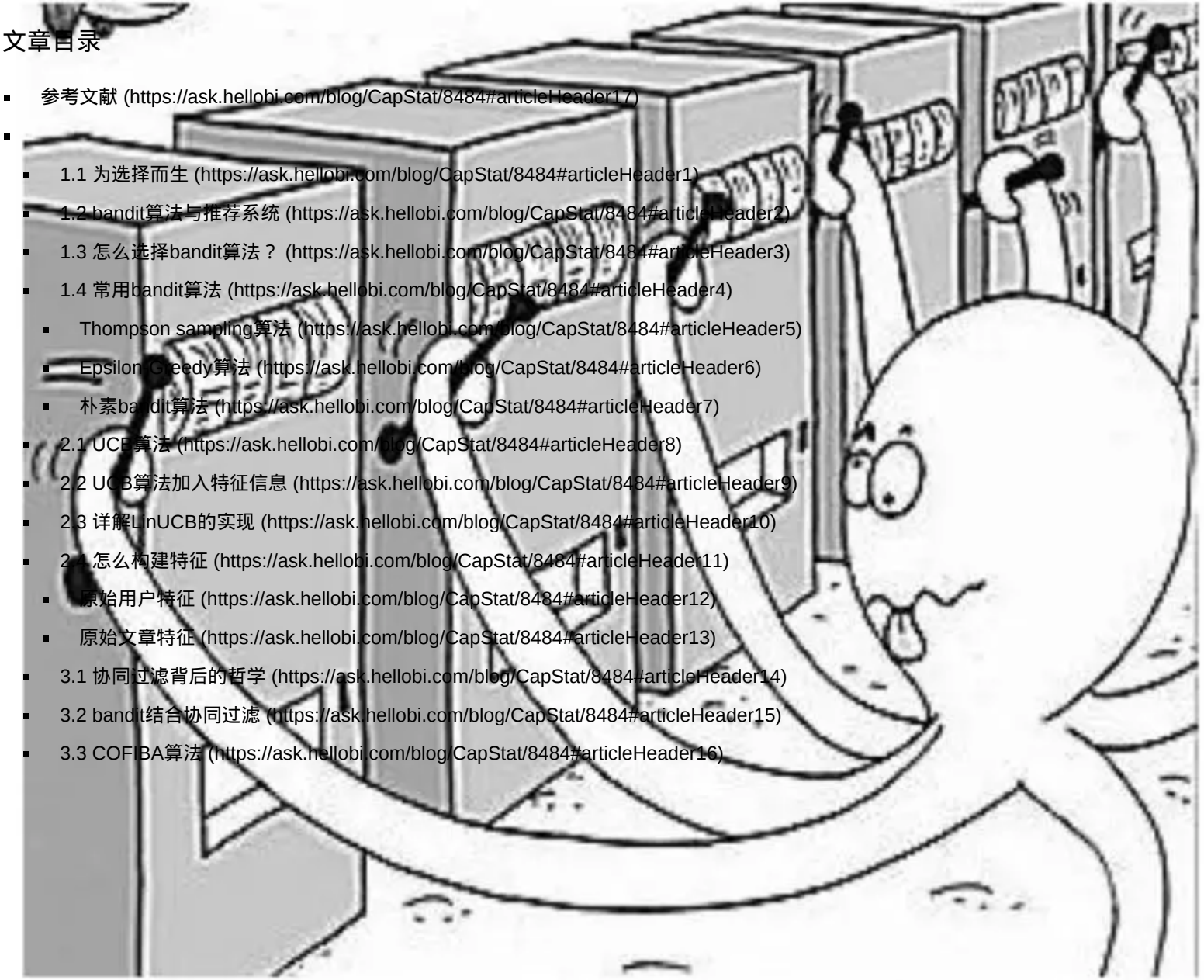


图1. MAB问题

bandit算法来源于历史悠久的赌博学，它要解决的问题是这样的1：

一个赌徒，要去摇老虎机，走进赌场一看，一排老虎机，外表一模一样，但是每个老虎机吐钱的概率可不一样，他不知道每个老虎机吐钱的概率分布是什么，那么每次该选择哪个老虎机可以做到最大化收益呢？这就是多臂赌博机问题(Multi-armed bandit problem, K-armed bandit problem, MAB)。

- 参考文献 (https://ask.hellobi.com/blog/CapStat/8484#articleHeader17)

怎么解决这个问题呢？最好的办法是去试一试，不是盲目地试，而是有策略地快速试一试，这些策略就是bandit算法。

这个多臂问题，推荐系统里面很多问题都与他类似：

1. 假设一个用户对不同类别的内容感兴趣程度不同，那么我们的推荐系统初次见到这个用户时，怎么快速地知道他对每类内容的感兴趣程度？这就是推荐系统的冷启动问题。
2. 假设我们有若干个广告库存，怎么知道该给每个用户展示哪个广告，从而获得最大的点击收益？是每次都挑效果最好那个么？那么新广告如何才有出头之日？
3. 我们的算法工程师又想出了一个新的模型，有没有比A/B test更快的方法知道它和旧模型相比谁更靠谱？
4. 如果只是推荐已知的用户感兴趣的物品，如何才能科学地冒险给他推荐一些新鲜的物品？

这些问题本质上全都是关乎如何选择，只要是关于选择，都可以简化成一个多臂赌博机问题。毕竟小赌怡情嘛，人生何处不赌博。

1.2 bandit算法与推荐系统

在推荐系统领域里，有两个比较经典的问题常被人提起，一个是EE问题，另一个是用户冷启动问题。

什么是EE问题？EE是exploit和explore问题。exploit就是：对用户已经确定的兴趣，当然要利用开采迎合，好比说已经挣到的钱，当然要花；explore就是：当对着用户已知的兴趣使用，用户很快会腻，所以要不断探索用户新的兴趣才行，这就好比虽然有一点钱可以花了，但是还得继续搬砖挣钱，不然花完了就得喝西北风。

用户冷启动问题，当就是面对新用户时，如何能够通过若干次实验，猜出用户的大致兴趣。

我想，屏幕前的你已经想到了，推荐系统冷启动可以用bandit算法来解决一部分。

这两个问题本质上都是如何选择用户感兴趣的主题进行推荐，比较符合bandit算法背后的MAB问题。

比如，用bandit算法解决冷启动的大致思路如下：

用分类或者Topic来表示每个用户兴趣，也就是MAB问题中的臂（Arm），我们可以通过几次试验，来刻画出新用户心目中对每个topic的感兴趣概率。

这里，如果用户对某个topic感兴趣（提供了显式反馈或隐式反馈），就表示我们得到了收益，如果推给了它不感兴趣的topic，推荐系统就表示很遗憾(regret)了。

如此经历“选择-观察-更新-选择”的循环，理论上是越来越逼近用户真正感兴趣的topic的。

1.3 怎么选择bandit算法？

文章目录

现在来介绍一下bandit算法怎么解决这类问题的。bandit算法需要量化一个核心问题：错误的选择到底有多大的遗憾？能不能遗憾少一些？

王家卫在《一代宗师》里寄出一句台词：

人生要是无憾，那多无趣？

而我说：算法要是无憾，那应该是过拟合了。

所以说：怎么衡量不同bandit算法在解决多臂问题上的效果？首先介绍一个概念，叫做累积遗憾(regret)2：

$$R_T = \sum_{t=1}^{T} \left(w_{opt} - w_{B(i)} \right)$$

$$= T w^* - \sum_{t=1}^{T} w_{B(i)}$$

图2. 累积遗憾

这个公式就是计算bandit算法的累积遗憾，解释一下：

首先，这里我们讨论的每个臂的收益非0即1，也就是伯努利收益。

然后，每次选择后，计算和最佳的选择差了多少，然后把差距累加起来就是总的遗憾。

$w_{B(i)}$ 是第i次试验时被选中臂的期望收益，w^* 是所有臂中的最佳那个，如果上帝提前告诉你，我们当然每次试验都选它，问题是上帝不告诉你，所以就有了bandit算法，我们就有了这篇文章。

这个公式可以用来对比不同bandit算法的效果：对同样的多臂问题，用不同的bandit算法试验相同次数，看看谁的regret增长得慢。

文章目录

那么到底不同的bandit算法有哪些呢？

Thompson sampling算法

thompson sampling算法简单实用，因为它只有一行代码就可以实现3，简单介绍一下它的原理，要点如下：

1. 假设每个臂是否产生收益，其背后有一个概率分布，产生收益的概率为p。
2. 我们不断地试验，去估计出一个置信度较高的“概率p的概率分布”就能近似解决这个问题了。
3. 怎么能估计“概率p的概率分布”呢？ 答案是假设概率p的概率分布符合beta(wins, lose)分布，它有两个参数: wins, lose。
4. 每个臂都维护一个beta分布的参数。每次试验后，选中一个臂，摇一下，有收益则该臂的wins增加1，否则该臂的lose增加1。
5. 每次选择臂的方式是：用每个臂现有的beta分布产生一个随机数，选择所有臂产生的随机数中最大的那个臂去摇。

以上就是Thompson采样，用python实现就一行：

```
import numpy as np
import pymc
#wins 和 trials 是一个N维向量，N是赌博机的臂的个数，每个元素记录了
choice = np.argmax(pymc.rbeta(1 + wins, 1 + trials - wins))
wins[choice] += 1
trials += 1
```

UCB算法

UCB算法全称是Upper Confidence Bound(置信区间上界)，它的算法步骤如下4：

1. 初始化：先对每一个臂都试一遍
2. 按照如下公式计算每个臂的分数，然后选择分数最大的臂作为选择：

$$\bar{x}_j(t) + \sqrt{\frac{2 \ln t}{T_{j,t}}}$$

图3. UCB算法

1. 观察选择结果，更新t和Tjt。其中加号前面是这个臂到目前的收益均值，后面的叫做bonus，本质上是均值的标准差，t是目前的试验次数，Tjt是这个臂被试次数。

这个公式反映一个特点：均值越大，标准差越小，被选中的概率会越来越大，同时哪些被选次数较少的臂也会得到试验机会。

Epsilon-Greedy算法

这是一个朴素的bandit算法，有点类似模拟退火的思想：

1. 选一个(0,1)之间较小的数作为epsilon
2. 每次以概率epsilon做一件事：所有臂中随机选一个
3. 每次以概率1-epsilon 选择截止到当前，平均收益最大的那个臂。

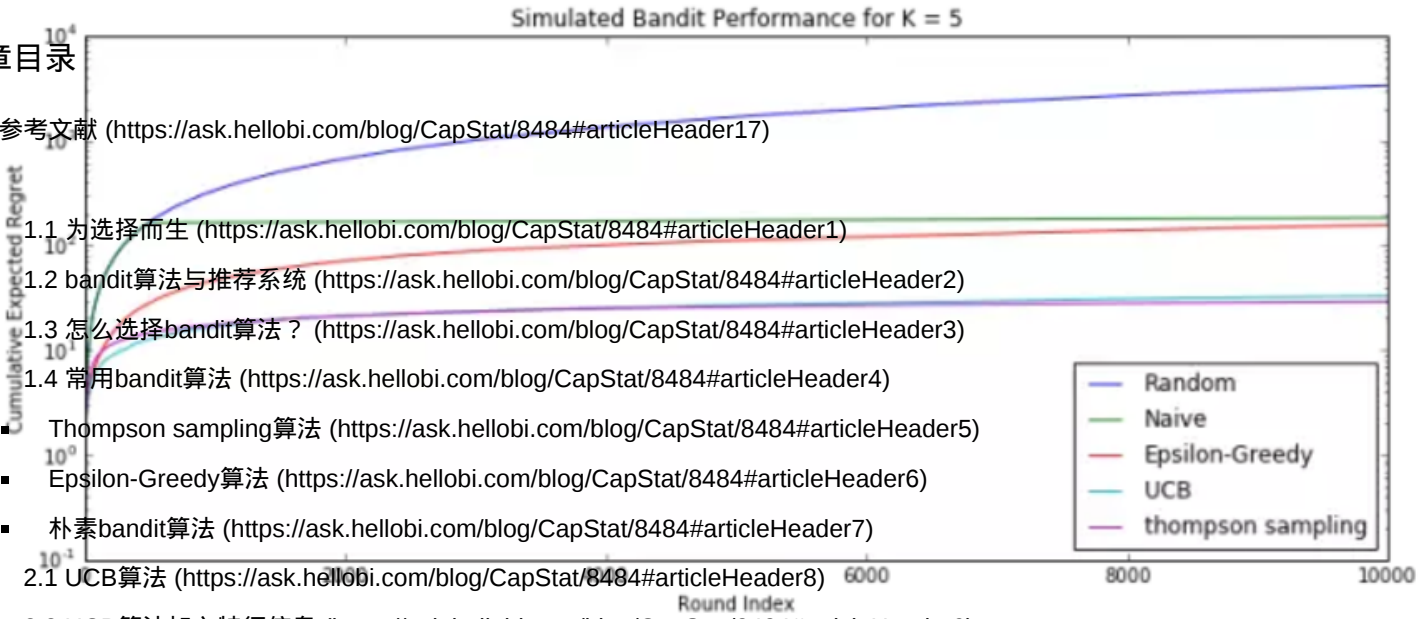
是不是简单粗暴？epsilon的值可以控制对Exploit和Explore的偏好程度。越接近0，越保守，只想花钱不想挣钱。

朴素bandit算法

最朴素的bandit算法就是：先随机试若干次，计算每个臂的平均收益，一直选均值最大那个臂。这个算法是人类在实际中最常采用的，不可否认，它还是比随机乱猜要好。

以上五个算法，我们用10000次模拟试验的方式对比了其效果如图，实验代码来源5：

文章目录



文章目录

算法效果对比一目了然：UCB算法和Thompson采样算法显著优秀一些。

至于你实际上要选哪一种bandit算法，你可以选一种bandit算法来选bandit算法。

2.bandit算法与线性回归

2.1 UCB算法

UCB算法在做E&E(Exploit-Explore)的时候表现不错，但它是上下文无关(context free)的bandit算法，它只管埋头干活，根本不观察一下面对的都是些什么样的item，下次遇到相似特点但不一样的item也帮不上什么忙。

UCB解决Multi-armed bandit问题的思路是：用置信区间。置信区间可以简单地理解为不确定性的程度，区间越宽，越不确定，反之亦反之。

每个item的回报均值都有个置信区间，随着试验次数增加，置信区间会变窄（逐渐确定了到底回报丰厚还是可怜）。 每次选择前，都根据已经试验的结果重新估计每个item的均值及置信区间。 选择置信区间上限最大的那个item。

“选择置信区间上界最大的那个item”这句话反映了几个意思：

1. 如果item置信区间很宽（被选次数很少，还不确定），那么它会倾向于被多次选择，这个是算法冒风险的部分；
2. 如果item置信区间很窄（备选次数很多，比较确定其好坏了），那么均值大的倾向于被多次选择，这个是算法保守稳妥的部分；
3. UCB是一种乐观的算法，选择置信区间上界排序，如果时悲观保守的做法，是选择置信区间下界排序。

2.2 UCB算法加入特征信息

文章目录

Yahoo!的科学家们在2010年发表了一篇论文6，给UCB引入了特征信息，同时还把改造后的UCB算法用在了Yahoo!的新闻推荐中，算法名叫LinUCB，刘鹏博士在《计算广告》一书中也有介绍LinUCB在计算广告中的应用7。

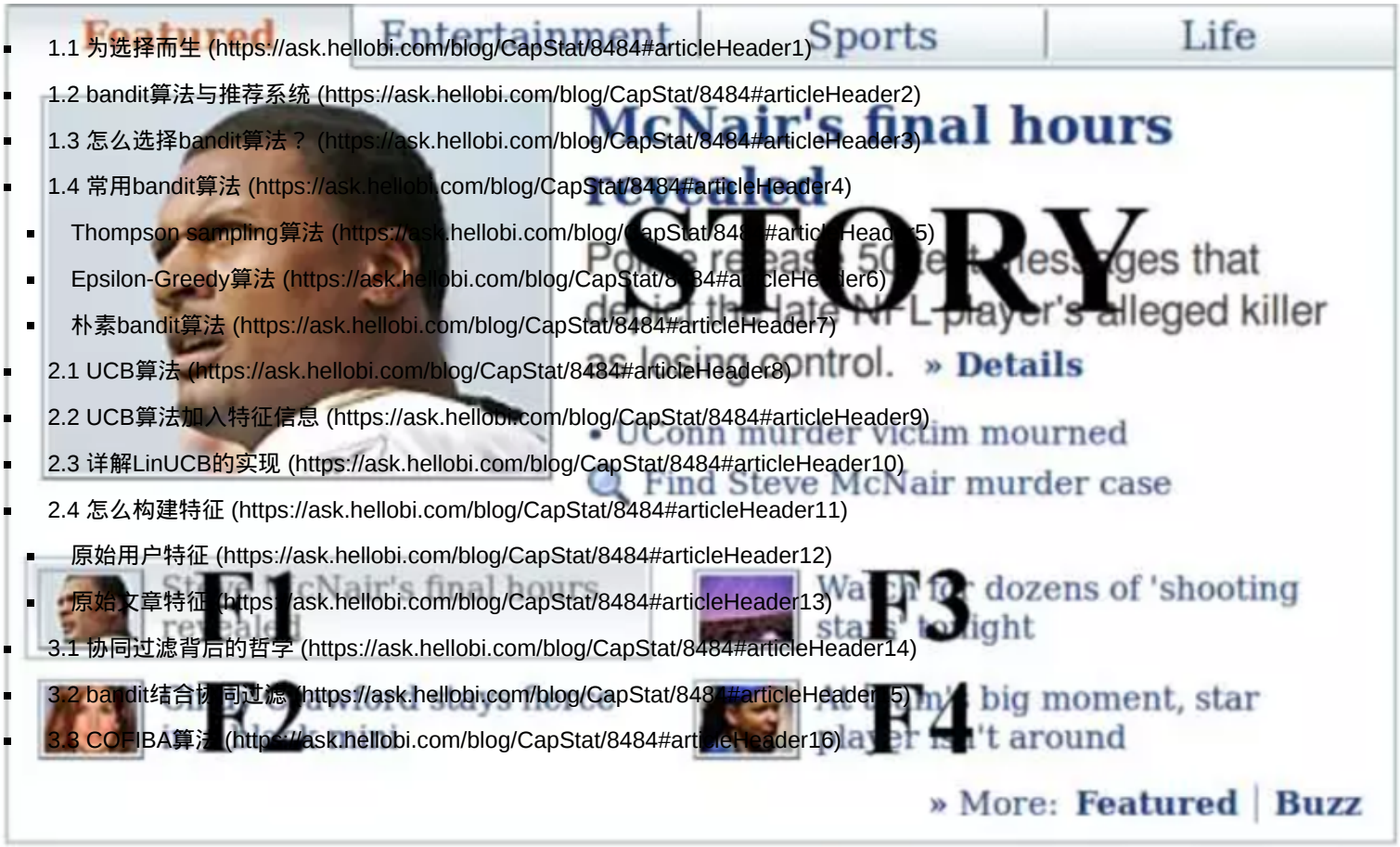


图4. 应用LinUCB算法的Yahoo!首页

单纯的老虎机回报情况就是老虎机自己内部决定的，而在广告推荐领域，一个选择的回报，是由User和Item一起决定的，如果我们能用feature来刻画User和Item这一对CP，在每次选择item之前，通过feature预估每一个arm（item）的期望回报及置信区间，选择的收益就可以通过feature泛化到不同的item上。

为UCB算法插上了特征的翅膀，这就是LinUCB最大的特色。

LinUCB算法做了一个假设：一个Item被选择后推送给一个User，其回报和相关Feature成线性关系，这里的“相关feature”就是context，也是实际项目中发挥空间最大的部分。

于是试验过程就变成：用User和Item的特征预估回报及其置信区间，选择置信区间上界最大的item推荐，观察回报后更新线性关系的参数，以此达到试验学习的目的。 LinUCB基本算法描述如下：

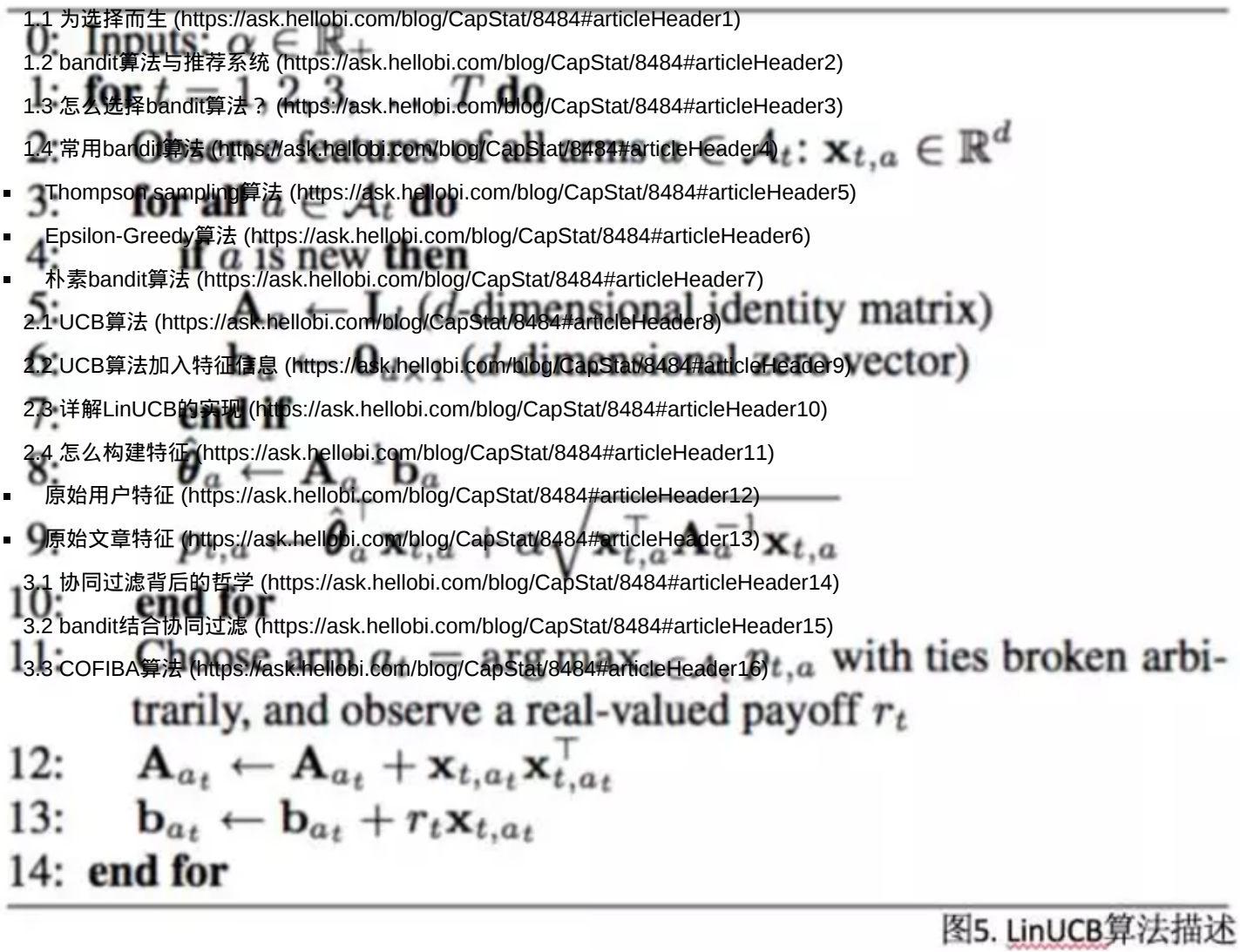


图5. LinUCB算法描述

对照每一行解释一下(编号从1开始)：

1. 设定一个参数 α ，这个参数决定了我们Explore的程度

2. 开始试验迭代

文章目录

3. 获取每一个arm的特征向量 $x_{t,a}$
4. 开始计算每一个arm的预估回报及其置信区间
5. 如果arm还从没有被试验过，那么：
6. 用单位矩阵初始化 A_a
7. 用0向量初始化 b_a，
8. 处理完没被试验过的arm
9. 计算线性参数 θ
10. 用 A_a 和特征向量 $x_{t,a}$ 计算预估回报，同时加上置信区间宽度
11. 处理完每一个arm
12. 选择第10步中最大值对应的arm，观察真实的回报 r_t
13. 更新 A_a
14. 更新 b_a
15. 算法结束

注意到上面的第4步，给特征矩阵加了一个单位矩阵，这就是岭回归（ridge regression），岭回归主要用于当样本数小于特征数时，对回归参数进行修正8。对于加了特征的bandit问题，正符合这个特点：试验次数（样本）少于特征数。

每一次观察真实回报之后，要更新的不止是岭回归参数，还有每个arm的回报向量 b_a 。

2.3 详解LinUCB的实现

根据论文给出的算法描述，其实很好写出LinUCB的代码9，麻烦的只是构建特征。

代码如下，一些必要的注释说明已经写在代码中。

文章目录


```
class LinUCB:
    def __init__(self):
        self.alpha = 0.25
        self.r1 = 1 # if worse -10, -8
        self.r0 = 0 # if worse, -19, -21
        # dimension of user features = d
        self.d = 6
        # Aa : collection of matrix to compute disjoint part for each article a, d*d
        self.Aa = {}
        # AaI : store the inverse of all Aa matrix
        self.AaI = {}
        # ba : collection of vectors to compute disjoin part, d*1
        self.ba = {}
        self.a_max = 0
        self.theta = {}
        self.x = None
        self.xT = None
        # linUCB
        def __init__(self, art):
        # init collection of matrix/vector Aa, Ba, ba
        for key in art:
            self.Aa[key] = np.identity(self.d)
            self.ba[key] = np.zeros((self.d, 1))
            self.AaI[key] = np.identity(self.d)
            self.theta[key] = np.zeros((self.d, 1))

        """
        这里更新参数时没有传入更新哪个arm，因为在上一次recommend的时候缓存了被选的那个arm，所以此处不用传入

        另外，update操作不用阻塞recommend，可以异步执行
        """
        def update(self, reward):
            if reward == -1:
                pass
            elif reward == 1 or reward == 0:
                if reward == 1:
                    r = self.r1
                else:
```

文章目录


```
            r = self.r0
        self.Aa[self.a_max] += np.dot(self.x, self.xT)
        self.ba[self.a_max] += r * self.x
        self.AaI[self.a_max] = linalg.solve(self.Aa[self.a_max], np.identity(self.d))
        self.theta[self.a_max] = np.dot(self.AaI[self.a_max], self.ba[self.a_max])
    else:
        # error
        pass
    '''
    预估每个arm的回报期望及置信区间
    '''
    def recommend(self, timestamp, user_features, articles):
        xaT = np.array([user_features])
        xa = np.transpose(xaT)
        # 标准linUCB
        old_pa = 0
        AaI_tmp = np.array([self.AaI[article] for article in articles])
        theta_tmp = np.array([self.theta[article] for article in articles])
        art_max = articles[np.argmax(np.dot(xaT, theta_tmp) + self.alpha * np.sqrt(np.dot(np.dot(xaT, AaI_t
mp), xa)))]
        # 缓存选择结果，用于update
        self.x = xa
        self.xT = xaT
        # article index with largest ucb
        self.a_max = art_max

        return self.a_max
```

2.4 怎么构建特征

LinUCB算法有一个很重要的步骤，就是给User和Item构建特征，也就是刻画context。在原始论文里，Item是文章，其中专门介绍了它们怎么构建特征的，也甚是精妙。容我慢慢表来。

原始用户特征

- 人口统计学：性别特征（2类），年龄特征（离散成10个区间）

- 地域信息：遍布全球的大都市，美国各个州
- 行为类别：代表用户历史行为的1000个类别取值

文章目录

4. 特征降维很有必要，关系到计算效率。

文章目录

3.1 协同过滤背后的哲学

推荐系统里面，传统经典的算法肯定离不开协同过滤。协同过滤背后的思想简单深刻，在万物互联的今天，协同过滤的威力更加强大。协同过滤看上去是一种算法，本质上是一种方法论，不是机器在给你推荐，而是集体智慧在给你推荐。

它的基本假设就是“物以类聚，人以群分”，你的圈子决定了你能见到的物品。这个假设很靠谱，却隐藏了一些重要的问题：作为用户的我们还可能看到新的东西吗？这个能有惊喜吗？这个能有圈子之间的更迭流动吗？这些问题的背后其实就是在前面提到过的EE问题（Exploit & Explore），我们关注推荐的准确率，但是我们也应该关注推荐系统的演进发展，因为“推荐系统不止眼前的Exploit，还有远方的Explore”。

做Exploit的方法有很多，bandit算法是其中的一种流派。前面已经介绍过几种bandit算法，基本上就是估计置信区间的做法，然后按照置信区间的上界来进行推荐，以UCB, LinUCB为代表。

作为要寻找新颖点的bandit浪漫派算法，能不能和协同过滤这种正统派算法结合起来呢？事实上已经有人这么尝试过了，叫做COFIBA算法，具体在题目为Collaborative Filtering Bandits[1]和Online Clustering of Bandits[2]）的两篇文章中有详细的描述，它就是bandit和协同过滤的结合算法，两篇文章的区别是后者只对用户聚类（即只考虑了User-based的协同过滤），而前者采用了协同聚类（co-clustering，可以理解为item-based和user-based两种协同方式在同时进行），后者是前者的一个特殊情况。下面详细介绍一下这种结合算法。

3.2 bandit结合协同过滤

很多推荐场景中都有这两个规律：

1. 相似的用户对同一个物品的反馈可能是一样的，也就是对一个聚类用户群体推荐同一个item，他们可能都喜欢，也可能都不喜欢，同样地，同一个用户会对相似的物品反馈相同。这是属于协同过滤可以解决的问题；
2. 在使用推荐系统过程中，用户的决策是动态进行的，尤其是新用户。这就导致无法提前为用户准备好推荐候选，只能“走一步看一步”，是一个动态的推荐过程。

每一个推荐候选item，都可以根据用户对其偏好不同（payoff不同）将用户聚类成不同的群体，一个群体来集体预测这个item的可能的收益，这就有了协同的效果，然后再实时观察真实反馈回来更新用户的个人参数，这就有了bandit的思想在里面。

举个例子，如果你父母给你安排了很多相亲对象，要不要见面去相一下？那需要提前看看每一个相亲对象的资料，每次大家都分成好几派，有说好的，有说再看看的，也有说不行的；你自己也会是其中一派的一员，每次都是你所属的那一派给你集体打分，因为他们是和你“三观一致的人”，“诚不欺我”；这样从一堆资料中挑出分数最高的那个人，你出去见TA，回来后把实际感觉说给大家听，同时自己心里的标准也有些调整，重新给剩下的其它对象打分，打完分再去见，周而复始……

以上就是协同过滤和bandit结合的思想。

另外，如果要推荐的候选item较多，还需要对item进行聚类，这样就不用按照每一个item对user聚类，而是按照每一个item的类簇对user聚类。如此以来，item的类簇数相对于item数要大大减少。

文章目录

3.3 COFIBA算法

基于这些思想，有人提出了算法COFIBA（读作coffee bar）13，简要描述如下：

在时刻 t，用户来访问推荐系统，推荐系统需要从已有的候选池子中挑一个最佳的物品推荐给他，然后观察他的反馈，用观察到的反馈来更新挑选策略。这里的每个物品都有一个特征向量，所以这里的bandit算法是context相关的。这里依然是用岭回归去拟合用户的权重向量，用于预测用户对每个物品的可能反馈（payoff），这一点和linUCB算法是一样的。

对比LinUCB算法，COFIBA算法的不同有两个：

1. 基于用户聚类挑选最在的user（相似用户集体决策的bandit）
2. 基于用户的反馈情况调整user和item的聚类（协同过滤部分）

整体算法过程如下：

文章目录

Input:

- Set of users $\mathcal{U} = \{1, \ldots, n\}$;
- set of items $\mathcal{I} = \{x_1, \ldots, x_m\} \subset \mathbb{R}^d$;
- exploration parameter $\alpha > 0$, and edge deletion parameter $\alpha_2 > 0$.

Init:

- $b_{i,0} \in \mathbb{R}^d$ and $M_{i,0} \in \mathbb{R}^{d \times d}$, $i = 1, \ldots, n$;
- user graph $G_{1,1} = (\mathcal{U}, E_{1,1})$; $G_{1,1}$ is connected over $\mathcal{U}$;
- Number of user graphs $g_1 = 1$;
- No. of *user* clusters $m_{1,1} = 1$;
- *Item* clusters $I_{1,1} = \mathcal{I}$, no. of *item* clusters $g_1 = 1$;
- *Item* graph $G_{1,1}^I = (\mathcal{I}, E_{1,1}^I)$, $G_{1,1}^I$ is connected over $\mathcal{I}$.

for $t = 1, 2, \ldots, T$ **do**

Set:

$$\boldsymbol{w}_{i,t-1} = M_{i,t-1}^{-1} \boldsymbol{b}_{i,t-1}, \quad i = 1, \ldots, n;$$

Receive $i_t \in \mathcal{U}$ and its cluster $C_{t,1}$, $\{\boldsymbol{x}_{t,1}, \ldots, \boldsymbol{x}_{t,c_t}\} \subseteq \mathcal{I}$;

For each $k = 1, \ldots, c_t$, determine which cluster (within the current user clustering w.r.t. $\boldsymbol{x}_{t,k}$) user i_t belongs to, and denote this cluster by N_k;

Compute, for $k = 1, \ldots, c_t$, aggregate quantities

$$\bar{M}_{N_k,t-1} = I + \sum_{i \in N_k} (M_{i,t-1} - I),$$

$$\bar{\boldsymbol{b}}_{N_k,t-1} = \sum_{i \in N_k} \boldsymbol{b}_{i,t-1},$$

$$\bar{\boldsymbol{w}}_{N_k,t-1} = \bar{M}_{N_k,t-1}^{-1} \bar{\boldsymbol{b}}_{N_k,t-1};$$

文章目录

$$k_t = \underset{k=1,\ldots,c_t}{\operatorname{argmax}} \left(\tilde{\boldsymbol{w}}_{N_k,t-1}^{\top} \boldsymbol{x}_{t,k} + \mathrm{CB}_{N_k,t-1}(\boldsymbol{x}_{t,k}) \right),$$

where $\mathrm{CB}_{N_k,t-1}(\boldsymbol{x}) = \alpha \sqrt{\boldsymbol{x}^{\top} \tilde{M}_{N_k,t-1}^{-1} \boldsymbol{x} \log(t+1)}$;

Set for brevity $\boldsymbol{x}_t = \boldsymbol{x}_{t,k_t}$;

Observe payoff $r_t \in [0,1]$, and update weights $M_{i,t}$ and $\boldsymbol{b}_{i,t}$ as follows:

$$M_{i,t} = M_{i,t-1} + \mathbf{1}_{\{i=k_t\}} \boldsymbol{x}_t \boldsymbol{x}_t^{\top},$$

$$\boldsymbol{b}_{i,t} = \boldsymbol{b}_{i,t-1} + r_t \boldsymbol{x}_t,$$

Set $M_{i,t} = M_{i,t-1}$, $\boldsymbol{b}_{i,t} = \boldsymbol{b}_{i,t-1}$ for all $i \neq t$;

Determine $\widehat{k}_t \in \{1,\ldots,q\}$ such that $k_t \in I_{\widehat{k}_t,t}$;

Update adjacency matrix $G_{t,n_t}^U = (V, E_{t,n_t}^U)$ by performing the steps in Figure 2;

For all $(\widehat{k}_t, j) \in E_{t,n_t}^U$ set $G_{t+1,n}^U = G_{t,n_t}^U$;

Update clusters at graph $G_{t+1,n}^U = (V, E_{t+1,n}^U)$ by performing the steps in Figure 3;

end for

图6. COFIBA算法描述

核心步骤是，针对某个用户，在每一轮试验时做以下事情：

1. 首先计算该用户的bandit参数W（和LinUCB相同），但是这个参数并不直接参与到bandit的选择决策中（和LinUCB不同），而是用来更新用户聚类的；
2. 遍历候选item，每一个item表示成一个context向量了。
3. 每一个item都对应一套用户聚类结果，所以遍历到每一个item时判断当前用户在当前item下属于哪个类簇，然后把对应类簇中每个用户的M矩阵(对应LinUCB里面的A矩阵)，b向量（payoff向量，对应linUCB里面的b向量）聚合起来，从而针对这个类簇求解一个岭回归参数（类似LinUCB里面单独针对每个用户所做），同时计算其payoff预测值和置信上边界
4. 每个item都得到一个payoff预测值及置信区间上界，挑出那个上边界最大的item推出去（和LinUCB相同）
5. 观察用户的真实反馈，然后更新用户自己的M矩阵和b向量（更新个人的，对应类簇里其他的不更新）

以上是COFIBA算法的一次决策过程。在收到用户真实反馈之后，还有两个计算过程：

1. 更新user聚类
2. 更新item聚类

如何更新user和item的聚类呢？示意图为：

文章目录

文章目录

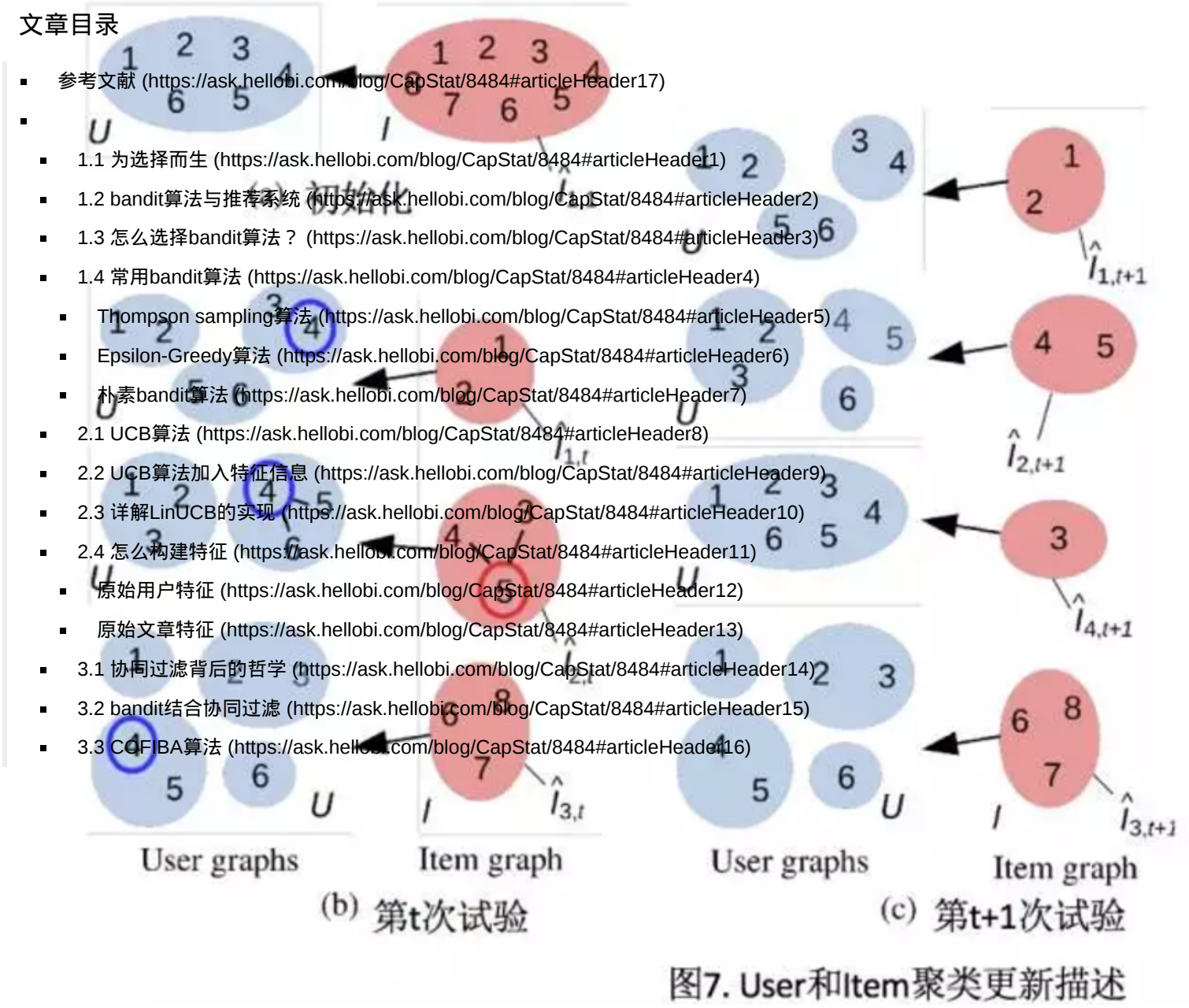


图7. User和Item聚类更新描述

解释一下这个图。

(a) 这里有6个user，8个item，初始化时，user和item的类簇个数都是1

文章目录

(b1) 在某一轮试验时，推荐系统面对的用户是4。推荐过程就是遍历1～8每个item，然后看看对应每个item时，user4在哪个类簇中，把对应类簇中的用户聚合起来为这个item预测payoff和CB。这里假设最终item5胜出，被推荐出去了。

(b2) 在时刻t，item有3个类簇，需要更新的用户聚类是item5对应的user4所在类簇。更新方式：看看该类簇里面除了user4之外的用户，对item5的payoff是不是和user4相近，如果是，则保持原来的连接边，否则删除原来的连接边。删除边之后重新构建聚类结果。这里假设重新构建后原来user4所在的类簇分裂成了两个类簇：{4,5}和{6}

(c)更新完用户类簇后，item5对应的类簇也要更新。更新方式是：对于每一个和item5(被推荐出的那个item)还存在连接边的item j，都去构造一个user的近邻集合N，这个集合的用户对item j有相近的payoff，然后看看N是不是和刚刚更新后的user4所在的类簇相同，是的话，保留item5和item j之间的连接边，否则删除。这里假设item 3和item 5之间的连接边被删除。item3独立后给他初始化了一个聚类结果：所有用户还是一个类簇。

简单来说就是这样：

- User-based协同过滤来选择要推荐的item，选择时用了LinUCB的思想
- 根据用户的反馈，调整User-based和Item-based的聚类结果
- Item-based的聚类变化又改变了User的聚类
- 不断根据用户实时动态的反馈来划分User-Item矩阵

4.总结

Exploit-Explore这一对矛盾一直客观存在，bandit算法是公认的一种比较好的解决EE问题的方案。除了bandit算法之外，还有一些其他的explore的办法，比如：在推荐时，随机地去掉一些用户历史行为（特征）。

解决Explore，势必就是要冒险，势必要走向未知，而这显然就是会伤害用户体验的：明知道用户肯定喜欢A，你还偏偏以某个小概率给推荐非A。

实际上，很少有公司会采用这些理性的办法做Explore，反而更愿意用一些盲目主观的方式。究其原因，可能是因为：

1. 互联网产品生命周期短，而Explore又是为了提升长期利益的，所以没有动力做；
2. 用户使用互联网产品时间越来越碎片化，Explore的时间长，难以体现出Explore 的价值；
3. 同质化互联网产品多，用户选择多，稍有不慎，用户用脚投票，分分钟弃你于不顾。
4. 已经成规模的平台，红利杠杠的，其实是没有动力做Explore的；

基于这些，我们如果想在自己的推荐系统中引入Explore机制，需要注意以下几点：

1. 用于Explore的item要保证其本身质量，纵使用户不感兴趣，也不至于引起其反感；
2. Explore本身的产品需要精心设计，让用户有耐心陪你玩儿；
3. 深度思考，这样才不会做出脑残的产品，产品不会早早夭折，才有可能让Explore机制有用武之地。

文章目录

参考文献

1. https://en.wikipedia.org/wiki/Multi-armed_bandit (https://en.wikipedia.org/wiki/Multi-armed_bandit)
2. http://nbviewer.jupyter.org/github/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/blob/master/Chapter6_Priorities/Chapter6.ipynb# (http://nbviewer.jupyter.org/github/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/blob/master/Chapter6_Priorities/Chapter6.ipynb#)
3. https://en.wikipedia.org/wiki/Thompson_sampling (https://en.wikipedia.org/wiki/Thompson_sampling)
4. http://hunch.net/ (http://hunch.net/)~coms-4771/lecture20.pdf
5. https://gist.github.com/anonymous/211b599b7bef958e50af (https://gist.github.com/anonymous/211b599b7bef958e50af)
6. http://www.research.rutgers.edu/ (http://www.research.rutgers.edu/)~lihong/pub/Li10Contextual.pdf
7. 《计算广告：互联网商业变现的市场与技术》p253, 刘鹏，王超著
8. https://en.wikipedia.org/wiki/Tikhonov_regularization (https://en.wikipedia.org/wiki/Tikhonov_regularization)
9. https://github.com/Fengrui/HybridLinUCB-python/blob/master/policy_hybrid.py (https://github.com/Fengrui/HybridLinUCB-python/blob/master/policy_hybrid.py)
10. http://www.gatsby.ucl.ac.uk/ (http://www.gatsby.ucl.ac.uk/)~chuwei/paper/isp781-chu.pdf
11. http://arxiv.org/abs/1401.8257 (http://arxiv.org/abs/1401.8257)
12. http://arxiv.org/abs/1502.03473 (http://arxiv.org/abs/1502.03473)
13. https://github.com/qw2ky/CoLinUCB_Revised/blob/master/COFIBA.py (https://github.com/qw2ky/CoLinUCB_Revised/blob/master/COFIBA.py)

作者：陈开江

编辑：王佳

文章目录

推荐 3


(https://ask.hellobi.com/people/xueba0911)

(https://ask.hellobi.com/people/kg_second)

(https://ask.hellobi.com/people/%E8%B7%AF%E8%BF%87%E5%B0%98%E4%B8%96)

0 个评论