(/gitchat/geekbook/5a7fe8eb194d786ae0b18956/topic/5a7fe91f194d786ae0b1895f)

Bandit 算法与推荐系统

文/陈开江

> 推荐系统里面有两个经典问题：EE 和冷启动。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。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，本来为你介绍基础的 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。

什么是 Bandit 算法

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。

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)。

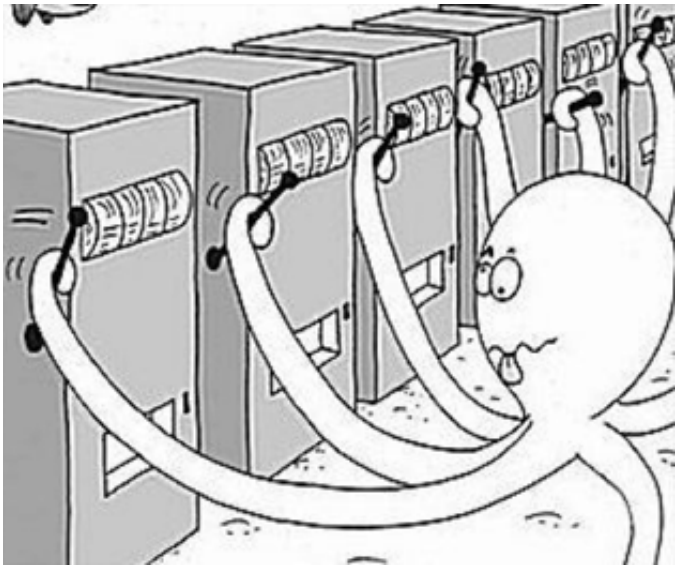

图1 MAB 问题

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，推荐系统里很多问题都与它类似：

- 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。
- 假设我们有若干广告库存，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？
- 我们的算法工程师又想出了新的模型，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?

- 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?

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？又叫 exploit－explore 问题。exploit 就是：对用户比较确定的兴趣，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：光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，也就是面对新用户时，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。

怎么选择 Bandit 算法?

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）：

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

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

图2 积累遗憾

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。

wB(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。

那么到底不同的 Bandit 算法有哪些呢?

常用 Bandit 算法

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。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 b，选择所有臂产生的随机数中最大的那个臂去摇。

```
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：

- 初始化：先对每一个臂都试一遍；
- 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：

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

- 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4，实验代码来源：

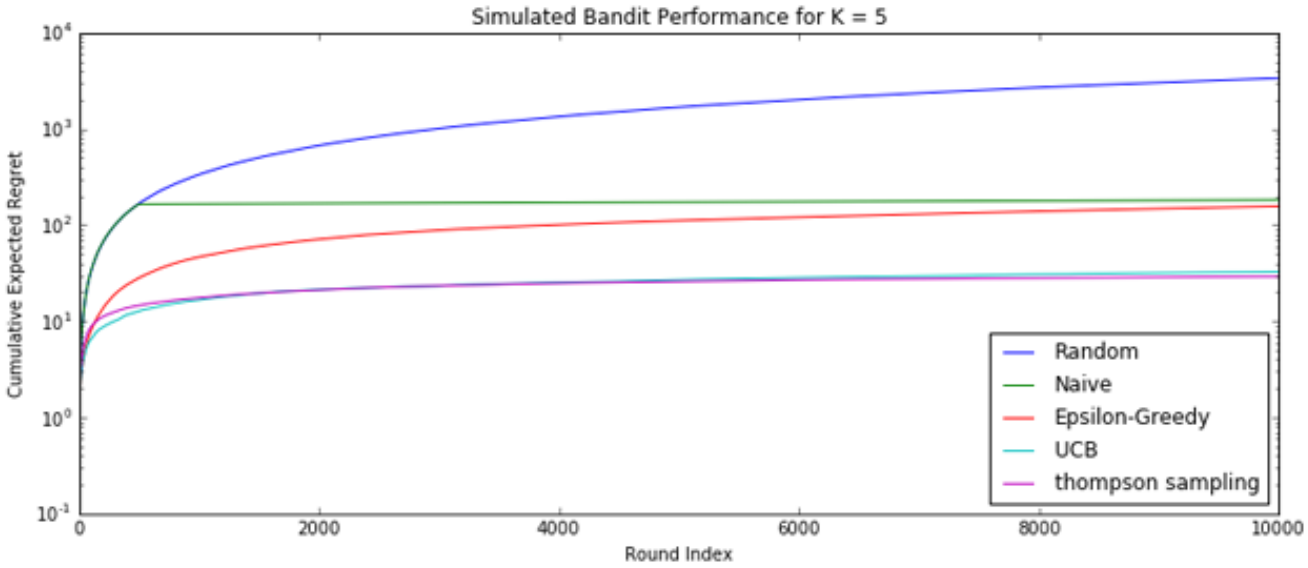

图4 五种 Bandit 算法模拟试验的效果图

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。

Bandit 算法与线性回归

UCB 算法

UCB 算法在做 EE（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特点的 arm，下次遇到相似特点但不一样的 arm 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。

UCB算法加入特征信息

Yahoo!的科学家们在2010年发表了一篇论文，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。

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。

图5 应用 LinUCB 算法的 Yahoo! 首页

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：

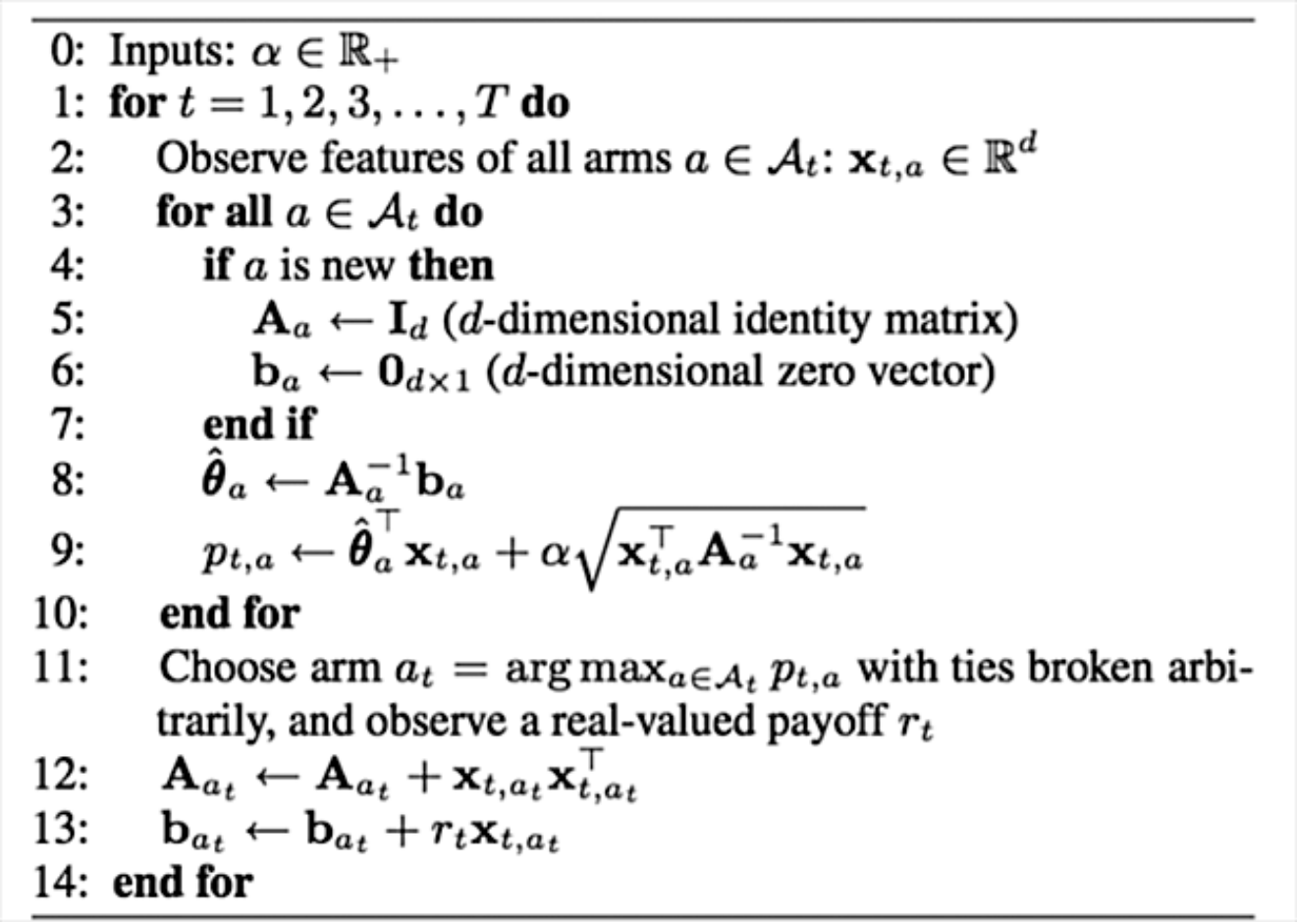

图6 LinUCB 算法描述

对照每一行解释一下（编号从1开始）：

1. 设定一个参数 \alpha，这个参数决定了我们 Explore 的程度；
2. 开始试验迭代；
3. 获取每一个 arm 的特征向量 xa,t；
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 Aa；
7. 用0向量初始化 ba；
8. 处理完没被试验过的 arm；
9. 计算线性参数 \theta；
10. 用 \theta 和特征向量 xa,t 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 rt；
13. 更新 Aat；

14. 更新 bat;
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 regres - sion），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。

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 ba。

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，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。

```
class LinUCB:
    def __init__(self):
     self.alpha = 0.25
     self.r1 = 1 # if worse -> 0.7, 0.8
        self.r0 = 0 # if worse, -19, -21
        # dimension of user features = d
        self.d = 6
        # Aa : collection of matrix to compute dis-
joint 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 set_articles(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recom-
mend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:
                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], s
elf.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)
        art_max = -1
        old_pa = 0

        # 获取在update阶段已经更新过的AaI(求逆结果)
        AaI_tmp = np.array([self.AaI[article] for article in arti-
cles])
        theta_tmp = np.array([self.theta[article] for article in ar
ticles])
        art_max = articles[np.argmax(np.dot(xaT, theta_tmp) + self.
alpha * np.sqrt(np.dot(np.dot(xaT, AaI_tmp), 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
```

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。

原始文章特征

URL 类别：根据文章来源分成了几十个类别。

编辑打标签：编辑人工给内容从几十个话题标签中挑选出来的原始特征向量都要归一化成单位向量。

还要对原始特征降维，以及模型要能刻画一些非线性的关系。

用 Logistic Regression 去拟合用户对文章的点击历史，其中的线性回归部分为：

$$\boldsymbol{\phi}_u^\top \mathbf{W} \boldsymbol{\phi}_a$$

拟合得到参数矩阵 W，可以将原始用户特征（1000多维）投射到文章的原始特征空间（80多维），投射计算方式：

$$\boldsymbol{\psi}_u \overset{\text{def}}{=} \boldsymbol{\phi}_u^\top \mathbf{W}$$

这是第一次降维，把原始1000多维降到80多维。

然后，用投射后的80多维特征对用户聚类，得到5个类簇，文章页同样聚类成5个簇，再加上常数1，用户和文章各自被表示成6维向量。

Yahoo! 的科学家们之所以选定为6维，因为数据表明它的效果最好，并且这大大降低了计算复杂度和存储空间。

我们实际上可以考虑三类特征：U（用户），A（广告或文章），C（所在页面的一些信息）。

前面说了，特征构建很有发挥空间，算法工程师们尽情去挥洒汗水吧。

总结一下 LinUCB 算法，有以下优点：

1. 由于加入了特征，所以收敛比 UCB 更快（论文有证明）；
2. 特征构建是效果的关键，也是工程上最麻烦和值的发挥的地方；
3. 由于参与计算的是特征，所以可以处理动态的推荐候选池，编辑可以增删文章；
4. 特征降维很有必要，关系到计算效率。

Bandit 算法与协同过滤

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，不如说是一种方法论，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？还可能有惊喜吗？还可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 Explore 的方法有很多，Bandit 算法是其中的一种流派。前面也介绍过几种 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。

作为要寻找诗和远方的 Bandit 浪漫派算法，能不能和协同过滤这种正统算法结合起来呢？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 和 Online Clustering of Bandits 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。

Bandit 结合协同过滤

很多推荐场景中都有这两个规律:

- 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;
- 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。

COFIBA 算法

基于这些思想，有人提出了算法 COFIBA（读作 coffee bar），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 lin - UCB 算法是一样的。

Input:

- Set of users $\mathcal{U} = \{1, \ldots, n\}$;
- set of items $\mathcal{I} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{|\mathcal{I}|}\} \subseteq \mathbb{R}^d$;
- exploration parameter $\alpha > 0$, and edge deletion parameter $\alpha_2 > 0$.

Init:

- $\boldsymbol{b}_{i,0} = \mathbf{0} \in \mathbb{R}^d$ and $M_{i,0} = I \in \mathbb{R}^{d \times d}$, $i = 1, \ldots n$;
- *User* graph $G^U_{1,1} = (\mathcal{U}, E^U_{1,1})$, $G^U_{1,1}$ is connected over $\mathcal{U}$;
- Number of *user* graphs $g_1 = 1$;
- No. of *user* clusters $m^U_{1,1} = 1$;
- *Item* clusters $\hat{I}_{1,1} = \mathcal{I}$, no. of *item* clusters $g_1 = 1$;
- *Item* graph $G^I_1 = (\mathcal{I}, E^I_1)$, G^I_1 is connected over $\mathcal{I}$.

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

Set

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

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

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

Set

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

where $\mathrm{CB}_{N_k,t-1}(\boldsymbol{x}) = \alpha \sqrt{\boldsymbol{x}^\top \bar{M}^{-1}_{N_k,t-1} \boldsymbol{x} \log(t+1)}$;
Set for brevity $\bar{\boldsymbol{x}}_t = \boldsymbol{x}_{t,k_t}$;
Observe payoff $a_t \in \mathbb{R}$, and update weights $M_{i,t}$ and $\boldsymbol{b}_{i,t}$ as follows:

- $M_{i_t,t} = M_{i_t,t-1} + \bar{\boldsymbol{x}}_t \bar{\boldsymbol{x}}^\top_t$,
- $\boldsymbol{b}_{i_t,t} = \boldsymbol{b}_{i_t,t-1} + a_t \bar{\boldsymbol{x}}_t$,
- Set $M_{i,t} = M_{i,t-1}$, $\boldsymbol{b}_{i,t} = \boldsymbol{b}_{i,t-1}$ for all $i \neq i_t$,

Determine $\widehat{h}_t \in \{1, \ldots, g_t\}$ such that $k_t \in \hat{I}_{\widehat{h}_t,t}$;
Update *user* clusters at graph $G^U_{t,\widehat{h}_t} = (\mathcal{U}, E^U_{t,\widehat{h}_t})$ by performing the steps in Figure 2;
For all $h \neq \widehat{h}_t$, set $G^U_{t+1,h} = G^U_{t,h}$;
Update *item* clusters at graph $G^I_t = (\mathcal{I}, E^I_t)$ by performing the steps in Figure 3.

end for

图7 COFIBA 算法描述

对比 LinUCB 算法，COFIBA 算法的不同有两个：

- 基于用户聚类挑选最佳的 Item（相似用户集体决策的 Bandit）。
- 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：

核心步骤是，针对某个用户 i，在每一轮试验时做以下事情：

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； 遍历候选 Item，每一个 Item 表示成一个 context 向量了。

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。 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）。

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？见图 8：

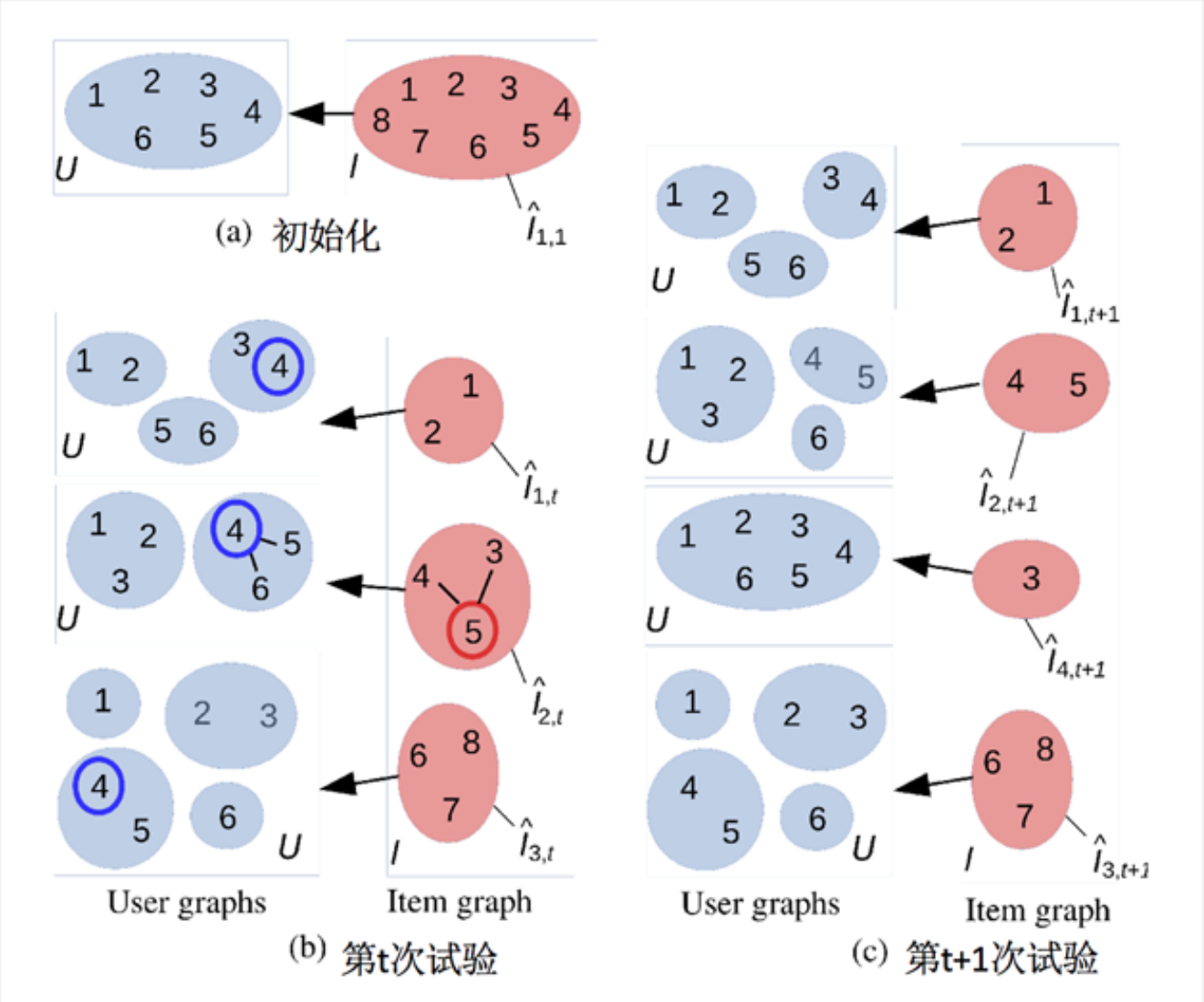

图8 User 和 Item 聚类更新描述

解释一下图8。 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。

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：

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

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。
5. 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：
6. 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；
7. Explore 本身的产品需要精心设计，让用户有耐心陪你玩儿；
8. 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。