利用用户行为数据

通过算法自动发掘用户行为数据，从用户的行为中推测出用户的兴趣，从而给用户推荐满足他们兴趣的物品。

基于用户行为分析的推荐算法是个性化推荐系统的重要算法，协同过滤算法就是指用户可以齐心协力，通过不断地和网站互动，使自己的推荐列表能够不断过滤掉自己不感兴趣的物品，从而越来越满足自己的需求。

用户行为数据简介

用户行为数据在网站上最简单的存在形式就是日志。网站在运行过程中都产生大量原始日志（raw log），并将其存储在文件系统中。

- 显性反馈行为（explicit feedback）：不同应用场景需要根据自身的特点来设计评分系统，包括用户明确表示对物品喜好的行为；
- 隐性反馈行为（implicit feedback）：那些不能明确反应用户喜好的行为，最具代表性的隐性反馈行为就是页面浏览行为。隐性反馈虽然不明确，但数据量更大，甚至只有隐性反馈数据，而没有显性反馈数据。

	显性反馈数据	隐性反馈数据
用户兴趣	明确	不明确
数量	较少	庞大
存储	数据库	分布式文件系统
实时读取	实时	有延迟
正负反馈	都有	只有正反馈

正反馈指用户的行为倾向于指用户喜欢该物品，而负反馈指用户的行为倾向于指用户不喜欢该物品。在显性反馈中，很容易区分一个用户行为是正反馈还是负反馈，而在隐性反馈行为中，就相对比较难以确定。

用户行为：产生行为的用户和行为的对象、行为的种类、产生行为的上下文、行为的内容和权重。

user id	产生行为的用户的唯一标识
item id	产生行为的对象的唯一标识
behavior type	行为的种类（比如是购买还是浏览）
context	产生行为的上下文，包括时间和地点等
behavior weight	行为的权重（如果是观看视频的行为，那么这个权重可以是观看时长；如果是打分行为，这个权重可以是分数）
behavior content	行为的内容（如果是评论行为，那么就是评论的文本；如果是打标签的行为，就是标签）

- 无上下文信息的隐性反馈数据集
- 无上下文信息的显性反馈数据集
- 有上下文信息的隐性反馈数据集
- 有上下文信息的显性反馈数据集

用户行为分析

用户活跃度和物品流行度的分布

互联网上的很多数据分布都满足一种称为 Power Law 的分布，这个分布在互联网领域也称长尾分布。

$$f(x) = \alpha x^k$$

令$f_u(k)$为对 k 个物品产生过行为的用户数，令$f_i(k)$为被 k 个用户产生过行为的物品数，则$f_u(k)$和$f_i(k)$都满足长尾分布，而长尾分布在双对数曲线上应该呈直线。

$$f_i(k) = \alpha_i x^{\beta_i}, f_u(k) = \alpha_u x^{\beta_u}$$

用户活跃度和物品流行度的关系

仅仅基于用户行为数据设计的推荐算法一般称为协同过滤算法。其他方法比如基于邻域的方法（neighborhood-based）、隐语义模型（latent factor model）、 基于图的随机游走算法（random walk on graph）等。最著名的、在业界得到最广泛应用的算法是基于邻域的方法，而基于邻域的方法：

- **基于用户的协同过滤算法** 给用户推荐和他兴趣相似的其他用户喜欢的物品。
- **基于物品的协同过滤算法** 给用户推荐和他之前喜欢的物品相似的物品。

实验设计和算法评测

数据集

TopN 推荐的任务是预测用户会不会对某部电影评分，而不是预测用户在准备对某部电影评分的前提下会给电影评多少分。

实验设计

使用交叉验证法主要是防止某次实验的结果是过拟合的结果（over fitting），但如果数据集够大，模型够简单，为了快速通过离线实验初步地选择算法，也可以只进行一次实验。

```
def SplitData(data, M, k, seed):
    test = []
    train = []
    random.seed(seed)
    for user, item in data:
        if random.randint(0,M) == k:
            test.append([user,item])
        else:
            train.append([user,item])
    return train, test
```

评测指标

对用户 u 推荐 N 个物品（记为$R(u)$），令用户 u 在测试集上喜欢的物品集合为$T(u)$，然后可以通过准确率/召回率评测推荐算法的精度：

$$Recall = \frac{\sum_u |R(u) \cap T(u)|}{\sum_u |T(u)|}$$

$$Precision = \frac{\sum_u |R(u) \cap T(u)|}{\sum_u |R(u)|}$$

```
def Recall(train, test, N):
    hit = 0
    all = 0
    for user in train.keys():
        tu = test[user]
        rank = GetRecommendation(user, N)
        for item, pui in rank:
            if item in tu:
                hit += 1
        all += len(tu)
    return hit / (all * 1.0)
def Precision(train, test, N):
    hit = 0
    all = 0
    for user in train.keys():
        tu = test[user]
        rank = GetRecommendation(user, N)
        for item, pui in rank:
            if item in tu:
                hit += 1
        all += N
    return hit / (all * 1.0)
```

覆盖率反映了推荐算法发掘长尾的能力，覆盖率越高，说明推荐算法越能够将长尾中的物品推荐给用户。

$$\text{Coverage} = \frac{|U_{u \in U} R(u)|}{|I|}$$

```
def Coverage(train, test, N):
    recommend_items = set()
    all_items = set()
    for user in train.keys():
        for item in train[user].keys():
            all_items.add(item)
        rank = GetRecommendation(user, N)
        for item, pui in rank:
            recommend_items.add(item)
    return len(recommend_items) / (len(all_items) * 1.0)
```

新颖度，这里用推荐列表中物品的平均流行度度量推荐结果的新颖度。如果推荐出的物品都很热门，说明推荐的新颖度较低，否则说明推荐结果比较新颖。

```
def Popularity(train, test, N):
    item_popularity = dict()
    for user, items in train.items():
        for item in items.keys():
```

```
            if item not in item_popularity:
                item_popularity[item] = 0
            item_popularity[item] += 1
    ret = 0
    n = 0
    for user in train.keys():
        rank = GetRecommendation(user, N)
        for item, pui in rank:
            ret += math.log(1 + item_popularity[item])
            n += 1
    ret /= n * 1.0
    return ret
```

这里，在计算平均流行度时对每个物品的流行度取对数，这是因为物品的流行度分布满足长尾分布，在取对数后，流行度的平均值更加稳定。

基于领域的算法

基于邻域的算法是推荐系统中最基本的算法，基于邻域的算法分为两大类，一类是基于用户的协同过滤算法，另一类是基于物品的协同过滤算法。

基于用户的协同过滤算法

基础算法

- 找到和目标用户兴趣相似的用户集合；
- 找到这个集合中的用户喜欢的，且目标用户没有听说过的物品推荐给目标用户。

主要利用行为的相似度计算兴趣的相似度。给定用户 u 和用户 v，令 N(u)表示用户 u 曾经有过正反馈的物品集合，令 N(v)为用户 v 曾经有过正反馈的物品集合。

```
def UserSimilarity(train):
    W = dict()
    for u in train.keys():
        for v in train.keys():
            if u == v:
                continue
            W[u][v] = len(train[u] & train[v])
            W[u][v] /= math.sqrt(len(train[u]) * len(train[v]) * 1.0)
    return W
```

得到用户之间的兴趣相似度后，UserCF 算法会给用户推荐和他兴趣最相似的 K 个用户喜欢的物品。如下的公式度量了 UserCF 算法中用户 u 对物品 i 的感兴趣程度：

$$p(u,i) = \sum_{v \in S(u,K) \cap N(i)} w_{uv} r_{vi}$$

```
def Recommend(user, train, W):
    rank = dict()
    interacted_items = train[user]
    for v, wuv in sorted(W[u].items, key=itemgetter(1), reverse=True)[0:K]:
```

```
        for i, rvi in train[v].items:
            if i in interacted_items:
            #we should filter items user interacted before
                Continue
            rank[i] += wuv * rvi
    return rank
```

- **准确率和召回率** 推荐系统的精度指标（准确率和召回率）并不和参数 K 成线性关系。选择合适的 K 对于获得高的推荐系统精度比较重要。当然，推荐结果的精度对 K 也不是特别敏感，只要选在一定的区域内，就可以获得不错的精度。
- **流行度** K 越大则 UserCF 推荐结果就越热门。这是因为 K 决定了 UserCF 在给你做推荐时参考多少和你兴趣相似的其他用户的兴趣，那么如果 K 越大，参考的人越多，结果就越来越趋近于全局热门的物品。
- **覆盖率** K 越大则 UserCF 推荐结果的覆盖率越低。覆盖率的降低是因为流行度的增加，随着流行度增加，UserCF 越来越倾向于推荐热门的物品，从而对长尾物品的推荐越来越少，因此造成了覆盖率的降低。

用户相似度计算的改进

$$w_{uv} = \frac{\sum_{i \in N(u) \cap N(v)} \frac{1}{\log(1 + |N(i)|)}}{\sqrt{|N(u)||N(v)|}}$$

```
def UserSimilarity(train):
    # build inverse table for item_users
    item_users = dict()
    for u, items in train.items():
        for i in items.keys():
            if i not in item_users:
                item_users[i] = set()
            item_users[i].add(u)
    #calculate co-rated items between users
    C = dict()
    N = dict()
    for i, users in item_users.items():
        for u in users:
            N[u] += 1
            for v in users:
                if u == v:
                    continue
                C[u][v] += 1 / math.log(1 + len(users))
    #calculate finial similarity matrix W
    W = dict()
    for u, related_users in C.items():
        for v, cuv in related_users.items():
            W[u][v] = cuv / math.sqrt(N[u] * N[v])
    return W
```

实际在线系统使用 UserCF 的例子

基于物品的协同过滤算法（item-based collaborative filtering）

基础算法

ItemCF 算法并不利用物品的内容属性计算物品之间的相似度，它主要通过分析用户的行为记录计算物品之间的相似度。

(1) 计算物品之间的相似度；

(2) 根据物品的相似度和用户的历史行为给用户生成推荐列表。

$$w_{ij} = \frac{|N(i) \cap N(j)|}{|N(i)|}$$

这会造成任何物品都会和热门的物品有很大的相似度，这对于致力于挖掘长尾信息的推荐系统来说显然不是一个好的特性。为了避免推荐出热门的物品，惩罚物品 j 的权重，从而减轻了热门物品会和很多物品相似的可能性。

$$w_{ij} = \frac{|N(i) \cap N(j)|}{\sqrt{|N(i)||N(j)|}}$$

和 UserCF 算法类似，用 ItemCF 算法计算物品相似度时也可以首先建立用户—物品倒排表（即对每个用户建立一个包含他喜欢的物品的列表），然后对于每个用户，将他物品列表中的物品两两在共现矩阵 C 中加 1。

```
def ItemSimilarity(train):
    #calculate co-rated users between items
    C = dict()
    N = dict()
    for u, items in train.items():
        for i in users:
            N[i] += 1
            for j in users:
                if i == j:
                    continue
                C[i][j] += 1
    #calculate finial similarity matrix W
    W = dict()
    for i,related_items in C.items():
        for j, cij in related_items.items():
            W[u][v] = cij / math.sqrt(N[i] * N[j])
    return W
```

对于隐反馈数据集，如果用户 u 对物品 i 有过行为，即可令r_{ui}=1。

$$p_{uj} = \sum_{i \in N(u) \cap S(j,k)} w_{ji} r_{ui}$$

```
def Recommendation(train, user_id, W, K):
    rank = dict()
    ru = train[user_id]
    for i,pi in ru.items():
        for j, wj in sorted(W[i].items(), key=itemgetter(1), reverse=True)[0:K]:
```

```
            if j in ru:
                continue
            rank[j] += pi * wj
    return rank
```

ItemCF 的一个优势就是可以提供推荐解释，即利用用户历史上喜欢的物品为现在的推荐结果进行解释。

```
def Recommendation(train, user_id, W, K):
    rank = dict()
    ru = train[user_id]
    for i,pi in ru.items():
        for j, wj in sorted(W[i].items(), key=itemgetter(1), reverse=True)[0:K]:
            if j in ru:
                continue
            rank[j].weight += pi * wj
            rank[j].reason[i] = pi * wj
    return rank
```

- **精度（准确率和召回率）** 可以看到 ItemCF 推荐结果的精度也是不和 K 成正相关或者负相关的，因此选择合适的 K 对获得最高精度是非常重要的。
- **流行度** 和 UserCF 不同，参数 K 对 ItemCF 推荐结果流行度的影响也不是完全正相关的。随着 K 的增加，结果流行度会逐渐提高，但当 K 增加到一定程度，流行度就不会再有明显变化。
- **覆盖率** K 增加会降低系统的覆盖率。

用户活跃度对物品相似度的影响

用户活跃度对数的倒数的参数，活跃用户对物品相似度的贡献应该小于不活跃的用户，应该增加 IUF 参数来修正物品相似度的计算公式：

$$w_{uv} = \frac{\sum_{i \in N(u) \cap N(v)} \frac{1}{\log(1 + |N(i)|)}}{\sqrt{|N(u)||N(v)|}}$$

```
def ItemSimilarity(train):
    #calculate co-rated users between items
    C = dict()
    N = dict()
    for u, items in train.items():
        for i in users:
            N[i] += 1
            for j in users:
                if i == j:
                    continue
                C[i][j] += 1 / math.log(1 + len(items) * 1.0)
    #calculate finial similarity matrix W
    W = dict()
    for i,related_items in C.items():
        for j, cij in related_items.items():
            W[u][v] = cij / math.sqrt(N[i] * N[j])
```

return W

物品相似度的归一化

如果已经得到了物品相似度矩阵 w，那么可以用如下公式得到归一化之后的相似度矩阵 w′，归一化的好处不仅仅在于增加推荐的准确度，它还可以提高推荐的覆盖率和多样性。

$$w'_{ij} = \frac{w_{ij}}{\max_j w_{ij}}$$

UserCF 和 ItemCF 的综合比较

UserCF 的推荐结果着重于反映和用户兴趣相似的小群体的热点，而 ItemCF 的推荐结果着重于维系用户的历史兴趣。换句话说，UserCF 的推荐更社会化，反映了用户所在的小型兴趣群体中物品的热门程度，而 ItemCF 的推荐更加个性化，反映了用户自己的兴趣传承。

	UserCF	ItemCF
性能	适用于用户较少的场合，如果用户很多，计算用户相似度矩阵代价很大	适用于物品数明显小于用户数的场合，如果物品很多（网页），计算物品相似度矩阵代价很大
领域	时效性较强，用户个性化兴趣不太明显的领域	长尾物品丰富，用户个性化需求强烈的领域
实时性	用户有新行为，不一定造成推荐结果的立即变化	用户有新行为，一定会导致推荐结果的实时变化
冷启动	在新用户对很少的物品产生行为后，不能立即对他进行个性化推荐，因为用户相似度表是每隔一段时间离线计算的	新用户只要对一个物品产生行为，就可以给他推荐和该物品相关的其他物品
	新物品上线后一段时间，一旦有用户对物品产生行为，就可以将新物品推荐给和对它产生行为的用户兴趣相似的其他用户	但没有办法在不离线更新物品相似度表的情况下将新物品推荐给用户
推荐理由	很难提供令用户信服的推荐解释	利用用户的历史行为给用户做推荐解释，可以令用户比较信服

原始 ItemCF 算法的覆盖率和新颖度都不高：

$$w_{ij} = \frac{|N(i) \cap N(j)|}{|N(i)|^{1-\alpha}|N(j)|^{\alpha}}$$

通过这种方法可以在适当牺牲准确率和召回率的情况下显著提升结果的覆盖率和新颖性（降低流行度即提高了新颖性）。两个不同领域的最热门物品之间往往具有比较高的相似度。这个时候，仅仅靠用户行为数据是不能解决这个问题的，因为用户的行为表示这种物品之间应该相似度很高。此时，我们只能依靠引入物品的内容数据解决这个问题，比如对不同领域的物品降低权重等。

隐语义模型（latent factor model）

基础算法

采取基于用户行为统计的自动聚类，计算用户 u 对物品 i 的兴趣：

$$Preference(u,i) = r_{ui} = p_u^T q_i = \sum_{k=1}^{F} p_{uk} q_{ik}$$

p_{uk}度量了用户 u 的兴趣和第 k 个隐类的关系，q_{ik}度量了第 k 个隐类和物品 i 之间的关系。

- 对每个用户，要保证正负样本的平衡（数目相似）。
- 对每个用户采样负样本时，要选取那些很热门，而用户却没有行为的物品。

```
def RandomSelectNegativeSample(self, items):
    ret = dict()
    for i in items.keys():
        ret[i] = 1
    n = 0
    for i in range(0, len(items) * 3):
        item = items_pool[random.randint(0, len(items_pool) - 1)]
        if item in ret:
            continue
        ret[item] = 0
        n + = 1
        if n > len(items):
            break
    return ret
```

经过采样，可以得到一个用户—物品集 $K=\{(u,i)\}$，其中如果(u,i)是正样本，则有r_{ui}=1，否则有r_{ui}=0。然后，需要优化如下的损失函数来找到最合适的参数 p 和 q：

$$C=\sum_{(u,i)\in K}(r_{ui}-\hat{r}_{ui})^2=\sum_{(u,i)\in K}(r_{ui}-\sum_{k=1}^{K}p_{uk}q_{ik})^2+\lambda\|p_u\|^2+\lambda\|q_i\|^2$$

采用随机梯度下降法需要首先对它们分别求偏导数：

$$\frac{\partial C}{\partial p_{uk}}=-2q_{ik}+2\lambda p_{uk}$$

$$\frac{\partial C}{\partial q_{ik}}=-2p_{uk}+2\lambda q_{ik}$$

根据随机梯度下降法，需要将参数沿着最速下降方向向前推进，因此

$$p_{uk}=p_{uk}+\alpha(q_{ik}-\lambda p_{uk})$$

$$q_{ik}=q_{ik}+\alpha(p_{uk}-\lambda\lambda q_{ik})$$

```
def LatentFactorModel(user_items, F, N, alpha, lambda):
    [P, Q] = InitModel(user_items, F)
    for step in range(0,N):
        for user, items in user_items.items():
            samples = RandSelectNegativeSamples(items)
            for item, rui in samples.items():
                eui = rui - Predict(user, item)
                for f in range(0, F):
                    P[user][f] += alpha * (eui * Q[item][f] - lambda * P[user][f])
                    Q[item][f] += alpha * (eui * P[user][f] - lambda * Q[item][f])
        alpha *= 0.9
def Recommend(user, P, Q):
    rank = dict()
    for f, puf in P[user].items():
        for i, qfi in Q[f].items():
```

```
            if i not in rank:
                rank[i] += puf * qfi
    return rank
```

基于 LFM 的实际系统的例子

LFM 模型在实际使用中有一个困难，那就是它很难实现实时的推荐。经典的 LFM 模型每次训练时都需要扫描所有的用户行为记录，这样才能计算出用户隐类向量（p_u）和物品隐类向量（q_i）。而且 LFM 的训练需要在用户行为记录上反复迭代才能获得比较好的性能。因此，LFM 的每次训练都很耗时，一般在实际应用中只能每天训练一次，并且计算出所有用户的推荐结果。从而 LFM 模型不能因为用户行为的变化实时地调整推荐结果来满足用户最近的行为。

为了解决实时性问题，首先利用新闻链接的内容属性（关键词、类别等）得到链接 i 的内容特征向量y_i。其次实时地收集用户对链接的行为，并且用这些数据得到链接 i 的隐特征向量q_i。然后，他们会利用如下公式预测用户 u 是否会单击链接 i：

$$r_{ui} = x_u^T * y_i + p_u^T * q_i$$

y_i是根据物品的内容属性直接生成的，x_{uk}是用户 u 对内容特征 k 的兴趣程度，用户向量x_u可以根据历史行为记录获得，而且每天只需要计算一次。而p_u、q_i是根据实时拿到的用户最近几小时的行为训练 LFM 获得的。因此，对于一个新加入的物品 i，可以通过$x_u^T * y_i$估计用户 u 对物品 i 的兴趣，然后经过几个小时后，就可以通过$p_u^T * q_i$得到更加准确的预测值。

LFM 和基于邻域的方法的比较

- **理论基础** LFM 具有比较好的理论基础，它是一种学习方法，通过优化一个设定的指标建立最优的模型。基于邻域的方法更多的是一种基于统计的方法，并没有学习过程。
- **离线计算的空间复杂度** 基于邻域的方法需要维护一张离线的相关表。在离线计算相关表的过程中，如果用户/物品数很多，将会占据很大的内存。假设有 M 个用户和 N 个物品，在计算相关表的过程中，我们可能会获得一张比较稠密的临时相关表（尽管最终我们对每个物品只保留 K 个最相关的物品，但在中间计算过程中稠密的相关表是不可避免的），那么假设是用户相关表，则需要 O(M*M)的空间，而对于物品相关表，则需要 O(N*N)的空间。而 LFM 在建模过程中，如果是 F 个隐类，那么它需要的存储空间是 O(F*(M+N))，这在 M 和 N 很大时可以很好地节省离线计算的内存。
- **离线计算的时间复杂度** 假设有 M 个用户、N 个物品、K 条用户对物品的行为记录。那么，UserCF 计算用户相关表的时间复杂度是 O(N*(K/N)^2)，而 ItemCF 计算物品相关表的时间复杂度是 O(M*(K/M)^2)。而对于 LFM，如果用 F 个隐类，迭代 S 次，那么它的计算复杂度是 O(K*F*S)。那么，如果 K/N>F*S，则代表 UserCF 的时间复杂度低于 LFM，如果 K/M>F*S，则说明 ItemCF 的时间复杂度低于 LFM。在一般情况下，LFM 的时间复杂度要稍微高于 UserCF 和 ItemCF，这主要是因为该算法需要多次迭代。但总体上，这两种算法在时间复杂度上没有质的差别。
- **在线实时推荐** UserCF 和 ItemCF 在线服务算法需要将相关表缓存在内存中，然后可以在线进行实时的预测。而从 LFM 的预测公式可以看到，LFM 在给用户生成推荐列表时，需要计算用户对所有物品的兴趣权重，然后排名，返回权重最大的 N 个物品。那么，在物品数很多时，这一过程的时间复杂度非常高，可达 O(M*N*F)。因此，LFM 不太适合用于物品数非常庞大的系统，如果要用，我们也需要一个比较快的算法给用户先计算一个比较小的候选列表，然后再用 LFM 重新排名。另一方面，LFM 在生成一个用户推荐列表时速度太慢，因此不能在线实时计算，而需要离线将所有用户的推荐结果事先计算好存储在数据库中。因此，LFM 不能进行在线实时推荐，也就是说，当用户有了新的行为后，他的推荐

列表不会发生变化。

- **推荐解释** ItemCF 算法支持很好的推荐解释，它可以利用用户的历史行为解释推荐结果。但 LFM 无法提供这样的解释，它计算出的隐类虽然在语义上确实代表了一类兴趣和物品，却很难用自然语言描述并生成解释展现给用户。

基于图的模型

用户行为数据的二分图表示

基于图的推荐算法

如果将个性化推荐算法放到二分图模型上，那么给用户 u 推荐物品的任务就可以转化为度量用户顶点v_u和与v_u没有边直接相连的物品节点在图上的相关性，相关性越高的物品在推荐列表中的权重就越高。

- **相关性度量**：两个顶点之间的路径数；两个顶点之间路径的长度；两个顶点之间的路径经过的顶点。
- **相关性高的一对顶点具有特征**：两个顶点之间有很多路径相连；连接两个顶点之间的路径长度都比较短；连接两个顶点之间的路径不会经过出度比较大的顶点。

基于随机游走的 PersonalRank 算法

假设要给用户 u 进行个性化推荐，可以从用户 u 对应的节点v_u开始在用户物品二分图上进行随机游走。游走到任何一个节点时，首先按照概率 α 决定是继续游走，还是停止这次游走并从v_u节点开始重新游走。如果决定继续游走，那么就从当前节点指向的节点中按照均匀分布随机选择一个节点作为游走下次经过的节点。这样，经过很多次随机游走后，每个物品节点被访问到的概率会收敛到一个数。最终的推荐列表中物品的权重就是物品节点的访问概率。

$$PR(v)=\begin{cases}\alpha\sum\limits_{v'\in in(v)}\dfrac{PR(v')}{|out(v')|}(v'\neq v)\\(1-alpha)+\alpha\sum\limits_{v'\in in(v)}\dfrac{PR(v')}{|out(v')|}(v'=v)\end{cases}$$

```
def PersonalRank(G, alpha, root):
    rank = dict()
    rank = {x:0 for x in G.keys()}
    rank[root] = 1
    for k in range(20):
        tmp = {x:0 for x in G.keys()}
            for i, ri in G.items():
                for j, wij in ri.items():
                    if j not in tmp:
                        tmp[j] = 0
                        tmp[j] += 0.6 * rank[i] / (1.0 * len(ri))
                    if j == root:
                        tmp[j] += 1 – alpha
        rank = tmp
    return rank
```

因为在为每个用户进行推荐时，都需要在整个用户物品二分图上进行迭代，直到整个图上的每个顶点的 PR 值收敛。这一过程的时间复杂度非常高，不仅无法在线提供实时推荐，甚至离线生成推荐结果也很耗时。

- 第一种很容易想到，就是减少迭代次数，在收敛之前就停止。这样会影响最终的精度，但一般来说影响不会特别大。
- 另一种方法就是从矩阵论出发，重新设计算法。

令 M 为用户物品二分图的转移概率矩阵，即：

$$M(v,v') = \frac{1}{|out(v)|}$$

$$r = (1-\alpha)r_0 + \alpha M^T r$$

$$r = (1-\alpha)(1-\alpha M^T)^{-1} r_0$$