

# 第二章、用户行为分析

讲师：武永亮



# 教学目标

---

- 了解用户行为数据简介
- 掌握用户行为分析方法
- 掌握实验设计和算法评测方法
- 理解基于邻域的算法
- 理解隐语义模型
- 理解基于图的模型

# 目录

---

1 用户行为数据简介

2 用户行为分析

3 实验设计和算法评测

4 基于邻域的算法

5 隐语义模型

6 基于图的模型

# 用户行为分析

---

- 智能推荐主要以人为主，所以我们要了解人的行为，其特征：
  - 现在的自然语言理解技术很难理解用户用来描述兴趣的自然语言；
  - 用户的兴趣是不断变化的，但用户不会不停地更新兴趣描述；
  - 很多时候用户并不知道自己喜欢什么，或者很难用语言描述自己喜欢什么。
- 我们需要通过算法**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。

# 用户行为分析

- 用户的行为不是随机的，而是蕴含着很多模式。

- 购物车：啤酒和尿布

购买本商品的顾客还买过

- 数据挖掘概念与技术（原
- Web数据挖掘（世界著名
- 数据挖掘实用机器学习技



数据挖掘导论【英文版】  
¥44.20

- 数据挖掘基础教程
- 数据挖掘与数学建模
- 数据挖掘：概念与技术（
- 数据仓库与数据挖掘原理
- 机器学习导论
- 机器学习

更多>>

## 数据挖掘导论(完整版)

正在读 (3人)，已读过 (37人)  放



当当价：¥51.80  
定价：¥69.00 折扣：75折  
顾客评分：★★★★★ 已有35条评论  
库存：送至 北京 有货

作者：(美) 陈封能，(美) 斯坦巴赫，(美) 库玛尔 著，范明 等译  
出版社：人民邮电出版社  
出版时间：2011-1-1  
版次：2 页数：463 字数：787000  
印刷时间：2011-1-1 开本：16开 纸张：胶版纸  
印次：1 I S B N：9787115241009 包装：平装

分享到：新浪微博 | 腾讯微博 | 开心网 | 人人网

我要买： 件

# 用户行为分析

---

- 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，学术界一般将这种类型的算法称为**协同过滤算法**。顾名思义，协同过滤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。

# 目录

---

1 用户行为数据简介

2 用户行为分析

3 实验设计和算法评测

4 基于邻域的算法

5 隐语义模型

6 基于图的模型

# 用户行为数据简介

---

- 用户行为数据在网站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。很多互联网业务会把多种原始日志按照用户行为汇总成会话日志（ session log ），其中每个 会话表示一次用户行为和对应的服务。



# 用户行为数据简介

- 按照反馈的明确性，用户行为在个性化推荐系统中一般分两种：
  - 显性反馈行为（explicit feedback）：用户明确表示对物品喜好的行为。
  - 隐性反馈行为（implicit feedback）：不能明确反应用户喜好的行为。



# 用户行为数据简介

- 按照反馈的明确性，用户行为在个性化推荐系统中一般分两种：
  - 显性反馈行为（explicit feedback）：用户明确表示对物品喜好的行为。
  - 隐性反馈行为（implicit feedback）：不能明确反应用户喜好的行为。

	显性反馈	隐性反馈
视频网站	用户对视频的评分	用户观看视频的日志、浏览视频页面的日志
电子商务网站	用户对商品的评分	购买日志、浏览日志
门户网站	用户对新闻的评分	阅读新闻的日志
音乐网站	用户对音乐/歌手/专辑的评分	听歌的日志

# 用户行为数据简介

- 按照反馈的明确性，用户行为在个性化推荐系统中一般分两种：
  - 显性反馈行为（explicit feedback）：用户明确表示对物品喜好的行为。
  - 隐性反馈行为（implicit feedback）：不能明确反应用户喜好的行为。

	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

# 用户行为数据简介

---

- 按照反馈的方向，用户行为在个性化推荐系统中一般分两种：
  - 正反馈：指用户的行为倾向于指用户喜欢该物品
  - 负反馈：负反馈指用户的行为倾向于指用户不喜欢该物品

# 用户行为数据简介

- 互联网中的用户行为有很多种，比如浏览网页、购买商品、评论、评分等。要用一个统一的方式表示所有这些行为是比较困难的。

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）

# 目录

---

1

用户行为数据简介

2

用户行为分析

3

实验设计和算法评测

4

基于邻域的算法

5

隐语义模型

6

基于图的模型

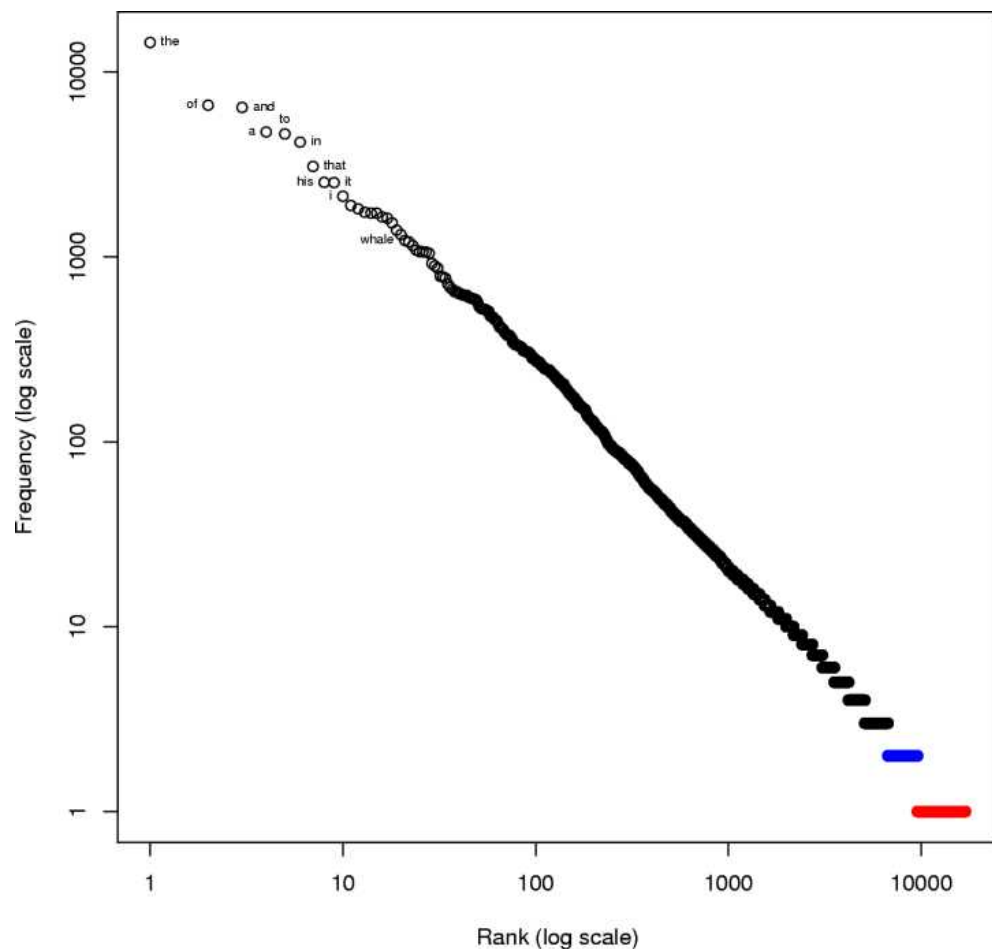
# 用户行为分析

---

- 在利用用户行为数据设计推荐算法之前，研究人员首先需要对用户行为数据进行分析，了解数据中蕴含的一般规律，这样才能对算法的设计起到指导作用。
  - 用户活跃度和物品流行度的分布
  - 用户活跃度和物品流行度的关系

# 用户行为分析

- 很多关于互联网数据的研究发现，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。



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

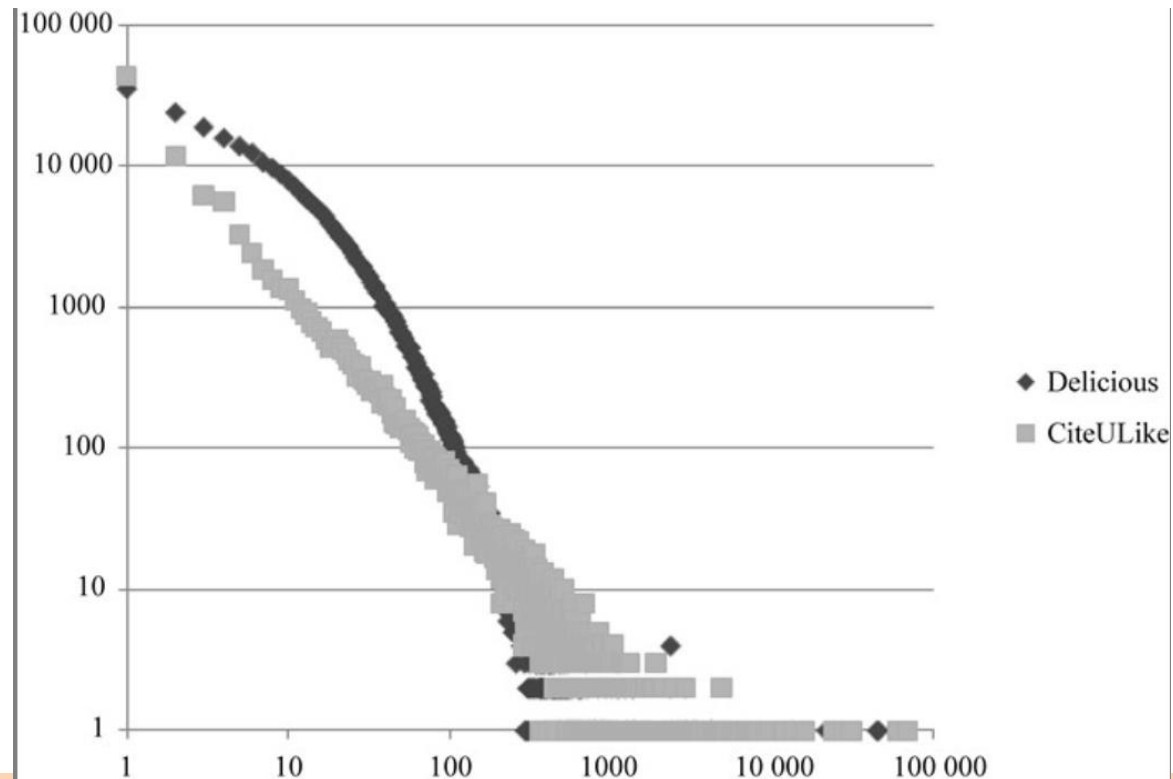
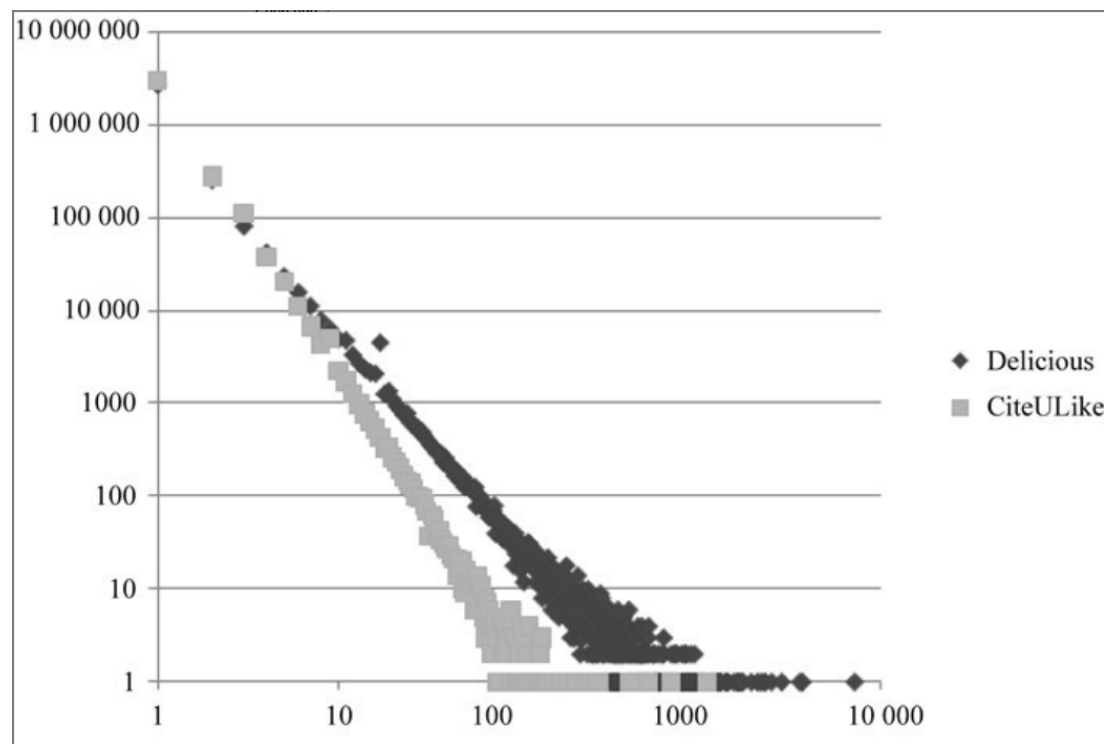


# 用户行为分析

很多研究人员发现，用户行为数据也蕴含着这种规律。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。那么， $f_u(k)$ 和 $f_i(k)$ 都满足长尾分布。也就是说：

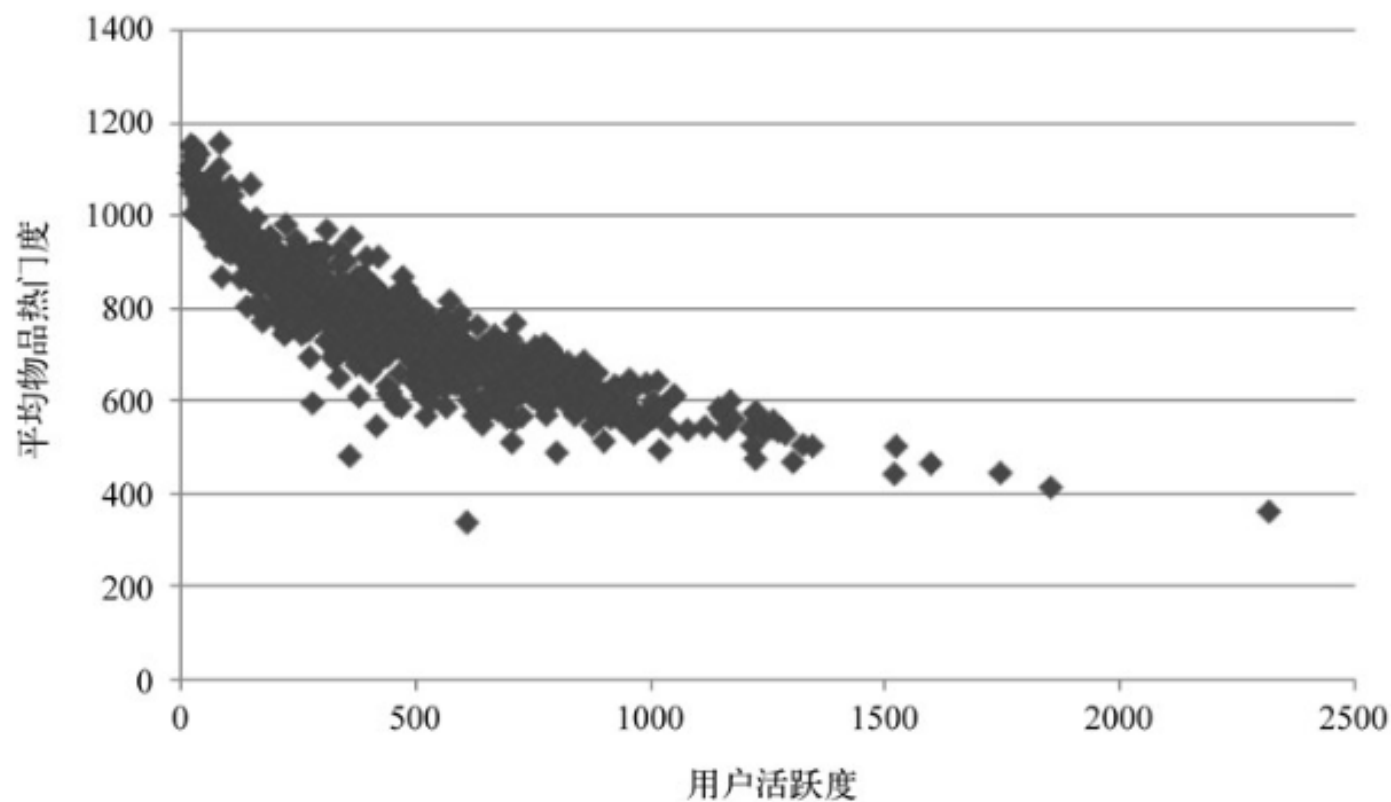
$$f_i(k) = \alpha_i k^{\beta_i}$$

$$f_u(k) = \alpha_u k^{\beta_u}$$



# 用户行为分析

- 用户活跃度：
  - 不活跃用户：浏览热门的物品
  - 活跃用户：根据自身喜好、朋友推荐购买物品



# 用户行为分析

- 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**。学术界对协同过滤算法进行了深入研究，提出了很多方法：
  - 基于邻域的方法（ neighborhood-based ）
    - 基于**用户**的协同过滤算法 这种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 这种算法给用户推荐和他之前喜欢的物品相似的物品。
  - 隐语义模型（ latent factor model ）
  - 基于图的随机游走算法（ random walk on graph ）

# 目录

---

1 用户行为数据简介

2 用户行为分析

3 实验设计和算法评测

4 基于邻域的算法

5 隐语义模型

6 基于图的模型

# 实验设计和算法评测

---

- 评测推荐系统有3种方法:

- 离线实验
- 用户调查
- 在线实验

# 实验设计和算法评测

---

- 数据集：GroupLens提供的MovieLens数据集，有3个不同的版本，我们选用中等大小的数据集。
- 该数据集包含6000多用户对4000多部电影的100万条评分。该数据集是一个评分数据集，用户可以给电影评5个不同等级的分数（1~5分）。
- 着重研究隐反馈数据集中的TopN推荐问题，因此忽略了数据集中的评分记录。也就是说，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。

# 实验设计和算法评测

---

- 实验设计：
  - 首先，将用户行为数据集按照均匀分布随机分成 $M$ 份（本章取 $M=8$ ），挑选一份作为测试集，将剩下的 $M-1$ 份作为训练集。
  - 然后在训练集上建立用户兴趣模型，并在测试集上对用户行为进行预测，统计出相应的评测指标。为了保证评测指标并不是过拟合的结果，需要进行 $M$ 次实验，并且每次都使用不同的测试集。
  - 然后将 $M$ 次实验测出的评测指标的平均值作为最终的评测指标。

# 实验设计和算法评测

- Python代码描述了将数据集随机分成训练集和测试集的过程

```
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
```



# 实验设计和算法评测

- 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 $T(u)$ ，然后通过准确率/召回率评测推荐算法的精度：
  - 准确率描述最终的推荐列表中有多少比例是发生过的用户—物品评分记录
  - 召回率描述有多少比例的用户—物品评分记录包含在最终的推荐列表中

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

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

# 实验设计和算法评测

- Python代码实现：

```
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)
```

# 实验设计和算法评测

- 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。覆盖率定义：

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

- 该覆盖率表示最终的推荐列表中包含多大比例的物品。如果所有的物品都被推荐给至少一个用户，那么覆盖率就是100%。

# 实验设计和算法评测

---

- Python代码

```
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)
```

# 实验设计和算法评测

- 评测推荐的新颖度，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
```

# 目录

---

1 用户行为数据简介

2 用户行为分析

3 实验设计和算法评测

4 基于邻域的算法

5 隐语义模型

6 基于图的模型

# 基于邻域的算法

---

- 基于邻域的算法是推荐系统中最基本的算法，该算法不仅在学术界得到了深入研究，而且在业界得到了广泛应用。基于邻域的算法分为两大类：
  - 基于用户的协同过滤算法
  - 基于物品的协同过滤算法

# 基于用户的协同过滤算法

---

- 基于用户的协同过滤算法是推荐系统中最古老的算法。可以不夸张地说，这个算法的诞生标志了推荐系统的诞生。该算法在1992年被提出，并应用于邮件过滤系统，1994年被GroupLens用于新闻过滤。在此之后直到2000年，该算法都是推荐系统领域最著名的算法。



# 基于用户的协同过滤算法

---

- 基于用户的协同过滤算法主要包括两个步骤：
  - 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

# 基于用户的协同过滤算法

- **找到和目标用户兴趣相似的用户集合**的关键就是计算两个用户的兴趣相似度。协同过滤算法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。那么，我们可以通过如下的Jaccard公式简单地计算 $u$ 和 $v$ 的兴趣相似度：

$$w_{uv} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$$

$$w_{uv} = \frac{|N(u) \cap N(v)|}{\sqrt{|N(u)| |N(v)|}}$$

# 基于用户的协同过滤算法

- UserCF计算用户兴趣相似度的例子。在该例中，用户A对物品 $\{a, b, d\}$ 有过行为，用户B对物品 $\{a, c\}$ 有过行为，利用余弦相似度公式计算用户A和用户B的兴趣相似度为：

A	a	b	d
B	a	c	
C	b	e	
D	c	d	e

$$w_{AB} = \frac{|\{a, b, d\} \cap \{a, c\}|}{\sqrt{|\{a, b, d\}| |\{a, c\}|}} = \frac{1}{\sqrt{6}}$$

$$w_{AC} = \frac{|\{a, b, d\} \cap \{b, e\}|}{\sqrt{|\{a, b, d\}| |\{b, e\}|}} = \frac{1}{\sqrt{6}}$$

$$w_{AD} = \frac{|\{a, b, d\} \cap \{c, d, e\}|}{\sqrt{|\{a, b, d\}| |\{c, d, e\}|}} = \frac{1}{3}$$

# 基于用户的协同过滤算法

- 余弦相似度的伪码

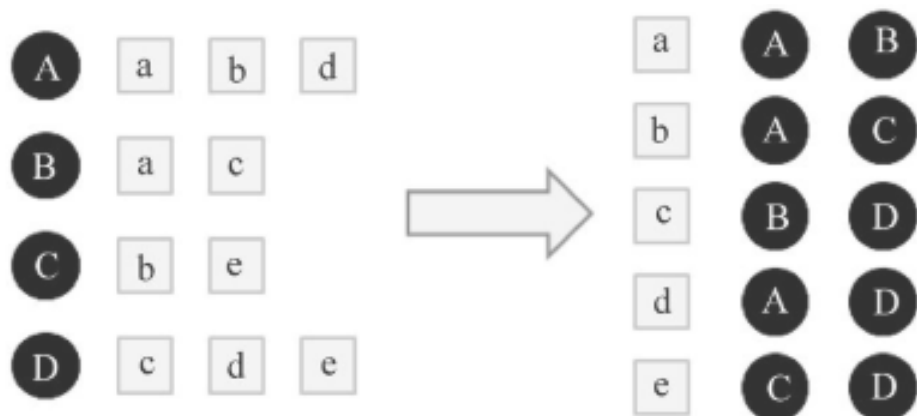
```
def UserSimilarity(train):  
    W = dict()  
    for u in train.keys():  
        for v in train.keys():  
            if u == v:  
                continue  
            W[u][v] = len(train[u] & train[v])  
            W[u][v] /= math.sqrt(len(train[u]) * len(train[v]) * 1.0)  
    return W
```

- 该代码对两两用户都利用余弦相似度计算相似度。这种方法的时间复杂度是 $O(|U|*|U|)$ ，这在用户数很大时非常耗时。

很多用户相互之间并没有对同样的物品产生过行为，即很多时候 $|N(u) \cap N(v)| = 0$ 。上面的算法将很多时间浪费在了计算这种用户之间的相似度上。如果换一个思路，我们可以首先计算出 $|N(u) \cap N(v)| \neq 0$ 的用户对 $(u, v)$ ，然后再对这种情况除以分母 $\sqrt{|N(u)||N(v)|}$ 。

# 基于用户的协同过滤算法

- 可以首先建立物品到用户的倒查表，对于每个物品都保存对该物品产生过行为的用户列表。



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

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

# 基于用户的协同过滤算法

- 可以首先建立物品到用户的倒查表，对于每个物品都保存对该物品产生过行为的用户列表。

$K$	准确率	召回率	覆盖率	流行度
5	16.99%	8.21%	51.33%	6.813293
10	20.59%	9.95%	41.49%	6.978854
20	22.99%	11.11%	33.17%	7.10162
40	24.50%	11.83%	25.87%	7.203149
80	25.20%	12.17%	20.29%	7.289817
160	24.90%	12.03%	15.21%	7.369063

	准确率	召回率	覆盖率	流行度
Random	0.631%	0.305%	100%	4.3855
MostPopular	12.79%	6.18%	2.60%	7.7244

# 基于用户的协同过滤算法

- 用户相似度计算的改进，根据用户行为计算用户的兴趣相似度：

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

```
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 final 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
```

# 基于用户的协同过滤算法

- 用户相似度计算的改进，根据用户行为计算用户的兴趣相似度：

	准 确 率	召 回 率	覆 盖 率	流 行 度
UserCF	25.20%	12.17%	20.29%	7.289817
UserCF-IIF	25.34%	12.24%	21.29%	7.261551



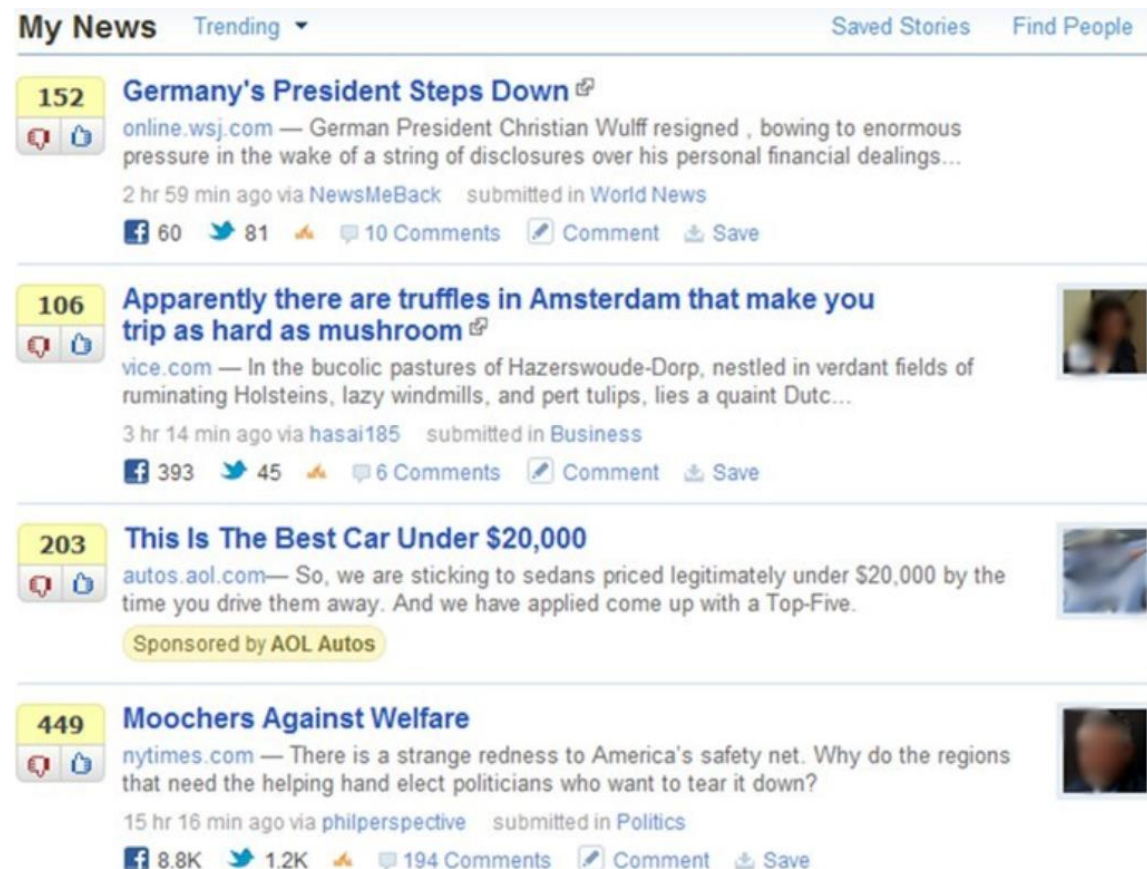
# 基于用户的协同过滤算法

---

- 实际在线系统使用UserCF的例子
- Digg的推荐系统设计思路如下。
  - 用户在Digg中主要通过“顶”和“踩”两种行为表达自己对文章的看法。
  - 当用户顶了一篇文章，Digg就认为该用户对这篇文章有兴趣，而且愿意把这篇文章推荐给其他用户。
  - Digg找到所有在该用户顶文章之前也顶了这一篇文章的其他用户，然后给他推荐那些人最近顶的其他文章。

# 基于用户的协同过滤算法

- Digg在博客中公布了使用推荐系统后的效果，主要指标如下：
  - 用户反馈增加：用户“顶”和“踩”的行为增加了40%。
  - 平均每个用户将从34个具相似兴趣的好友那儿获得200条推荐结果。
  - 用户和好友的交互活跃度增加了24%。
  - 用户评论增加了11%。



# 基于用户的协同过滤算法




---

- 基于用户的协同过滤算法在有一些缺点
  - 随着网站的用户数目越来越大，计算用户兴趣相似度矩阵将越来越困难，其运算时间复杂度和空间复杂度的增长和用户数的增长近似于平方关系。
  - 基于用户的协同过滤很难对推荐结果作出解释。
- 基于物品的协同过滤算法（简称ItemCF）给用户推荐那些和他们之前喜欢的物品相似的物品。

# 基于用户的协同过滤算法

- 基于物品的协同过滤算法

Customers Who Bought This Item Also Bought Page 1 of 19

 Apple iPhone 4 16GB Smartphone Black (AT&T) ★★★★☆ (12) \$520.00	 IPHONE 4 16GB GSM by Apple ★★★★☆ (43) \$729.99	 Apple iPhone 4S 16GB - AT&T - Black by Apple ★★★★☆ (14) \$580.00	 Micro SIM Cutter, Converter with 2 SIM adapters ★★★★☆ (147) \$4.85
 Apple iPad 2 MC769LL / A Tablet (16GB, WiFi, Black) by Apple ★★★★☆ (615)	 iPhone Sim Card Tray Open Eject Pin (Compatible for All iPhones) ★★★★☆ (20) \$0.59	 iPhone 3GS 16 GB Black Unlocked by Apple ★★★★☆ (9) \$409.95	 Apple iPhone 3G 8GB - Unlocked by Apple ★★★★☆ (77) \$285.00

# 基于用户的协同过滤算法

- 基于物品的协同过滤算法



# 基于用户的协同过滤算法

- 基于物品的协同过滤算法步骤：
  - (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。
- 定义物品的相似度：

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

- 分母 $|N(i)|$ 是喜欢物品 $i$ 的用户数，而分子是同时喜欢物品 $i$ 和物品 $j$ 的用户数。因此，上述公式可以理解为喜欢物品 $i$ 的用户中有多少比例的用户也喜欢物品 $j$ 。

# 基于用户的协同过滤算法

- 如果物品j很热门，很多人都喜欢，那么 $w_{ij}$ 就会很大，接近1。因此，该公式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，可以用下面的公式：

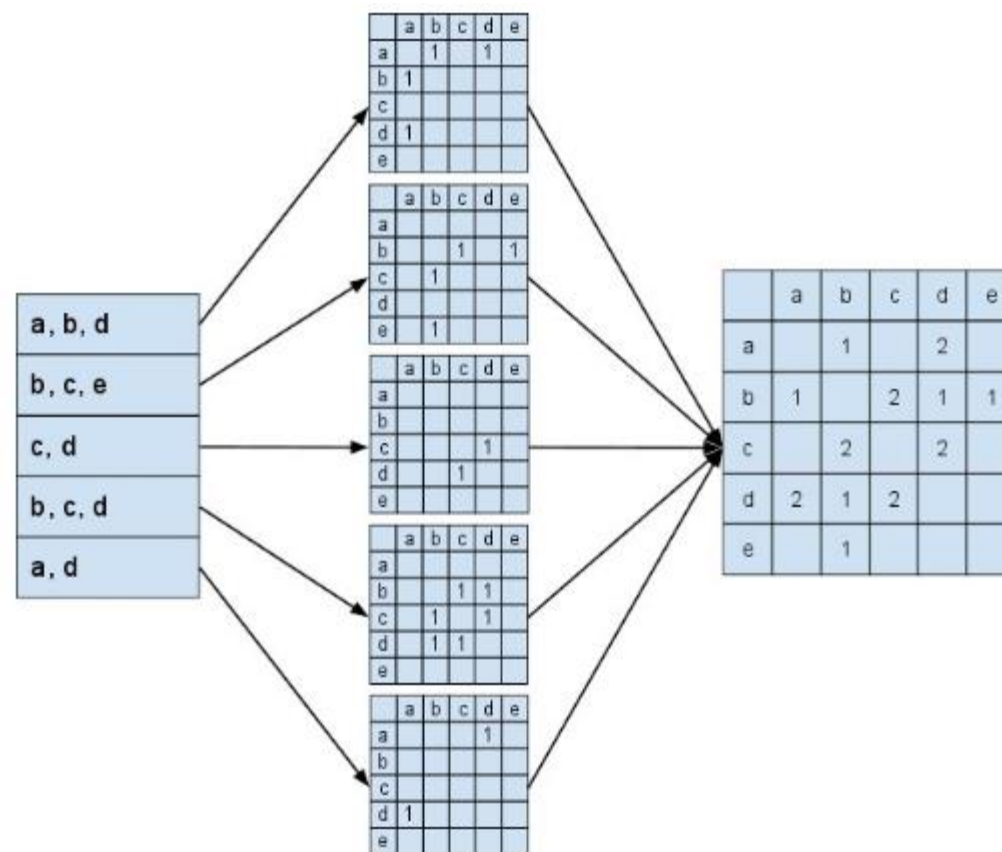
$$w_{ij} = \frac{|N(i) \cap N(j)|}{\sqrt{|N(i)| |N(j)|}}$$



# 基于用户的协同过滤算法

- 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。详细代码如下所示：

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





# 基于用户的协同过滤算法

- 在得到物品之间的相似度后，ItemCF通过如下公式计算用户 $u$ 对一个物品 $j$ 的兴趣：

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

- 这里 $N(u)$ 是用户喜欢的物品的集合， $S(j, K)$ 是和物品 $j$ 最相似的 $K$ 个物品的集合， $w_{ji}$ 是物品 $j$ 和 $i$ 的相似度， $r_{ui}$ 是用户 $u$ 对物品 $i$ 的兴趣。（对于隐反馈数据集，如果用户 $u$ 对物品 $i$ 有过行为，即可令 $r_{ui}=1$ 。）该公式的含义是，和用户历史上感兴趣的物品越相似的物品，越有可能在用户的推荐列表中获得比较高的排名。

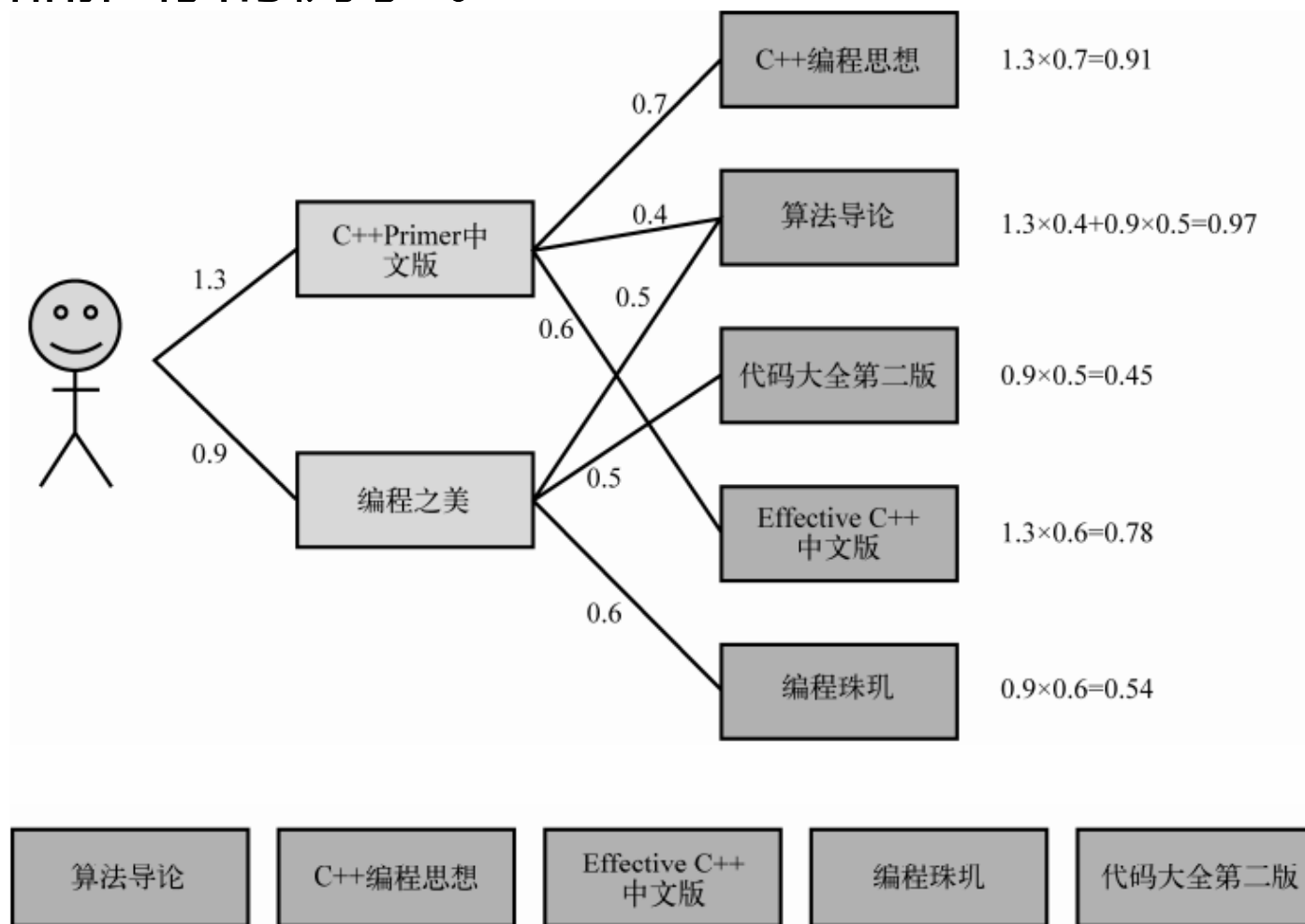
# 基于用户的协同过滤算法

- Python代码实现

```
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
```

# 基于用户的协同过滤算法

- 一个基于物品推荐的例子。



# 基于用户的协同过滤算法

- 表2-8列出了在MovieLens数据集上ItemCF算法离线实验的各项性能指标的评测结果。该表包括算法在不同 $K$ 值下的性能。

表2-8 MovieLens数据集中ItemCF算法离线实验的结果

$K$	准 确 率	召 回 率	覆 盖 率	流 行 度
5	21.47%	10.37%	21.74%	7.172411
<b>10</b>	<b>22.28%</b>	<b>10.76%</b>	<b>18.84%</b>	<b>7.254526</b>
20	22.24%	10.74%	16.93%	7.338615
40	21.68%	10.47%	15.31%	7.391163
80	20.64%	9.97%	13.64%	7.413358
160	19.37%	9.36%	11.77%	7.385278

# 基于用户的协同过滤算法

- 2. 用户活跃度对物品相似度的影响
- John S. Breese提出IUF ( Inverse User Frequency ) , 即用户活跃度对数的倒数的参数, 他也认为活跃用户对物品相似度的贡献应该小于不活跃的用户, 他提出应该增加IUF参数来修正物品相似度的计算公式:

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

# 基于用户的协同过滤算法

- 上面的公式只是对活跃用户做了一种软性的惩罚，但对于很多过于活跃的用户，比如上面那位买了当当网80%图书的用户，为了避免相似度矩阵过于稠密，我们在实际计算中一般直接忽略他的兴趣列表，而不将其纳入到相似度计算的数据集中。 算法记作ItemCF-IUF。

```
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 final 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
```

# 基于用户的协同过滤算法

- ItemCF-IUF在准确率和召回率两个指标上和ItemCF相近，但ItemCF-IUF明显提高了推荐结果的覆盖率，降低了推荐结果的流行度。从这个意义上说，ItemCF-IUF确实改进了ItemCF的综合性能。

表2-9 MovieLens数据集中ItemCF算法和ItemCF-IUF算法的对比

	准 确 率	召 回 率	覆 盖 率	流 行 度
ItemCF	22.28%	10.76%	18.84%	7.254526
ItemCF-IUF	22.29%	10.77%	19.70%	7.217326

# 基于用户的协同过滤算法

- 3. 物品相似度的归一化
- Karypis在研究中发现如果将ItemCF的相似度矩阵按最大值归一化，可以提高推荐的准确率。其研究表明，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 $w'$ ：

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



# 基于用户的协同过滤算法

- 对比了ItemCF算法和ItemCF-Norm算法的离线实验性能。从实验结果可以看到，归一化确实能提高ItemCF的性能，其中各项指标都有了比较明显的提高。

表2-10 MovieLens数据集中ItemCF算法和ItemCF-Norm算法的对比

	准 确 率	召 回 率	覆 盖 率	流 行 度
ItemCF	22.28%	10.76%	18.84%	7.254526
ItemCF-Norm	22.73%	10.98%	23.73%	7.157385

# 基于用户的协同过滤算法

- UserCF和ItemCF的综合比较

表2-11 UserCF和ItemCF优缺点的对比

	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

# 基于用户的协同过滤算法

- UserCF和ItemCF的综合比较

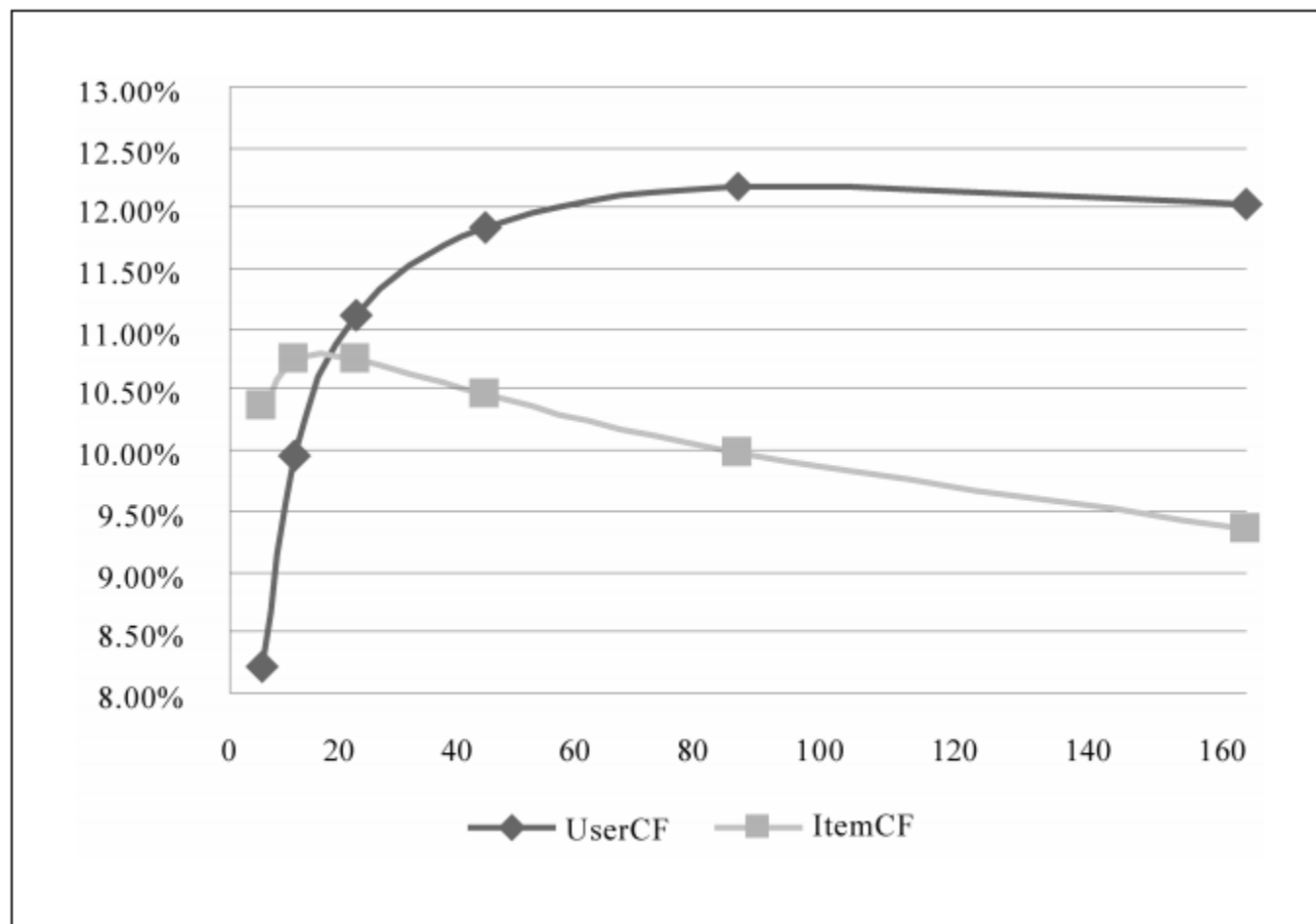


图2-13 UserCF和ItemCF算法在不同K值下的召回率曲线

# 基于用户的协同过滤算法

- UserCF和ItemCF的综合比较

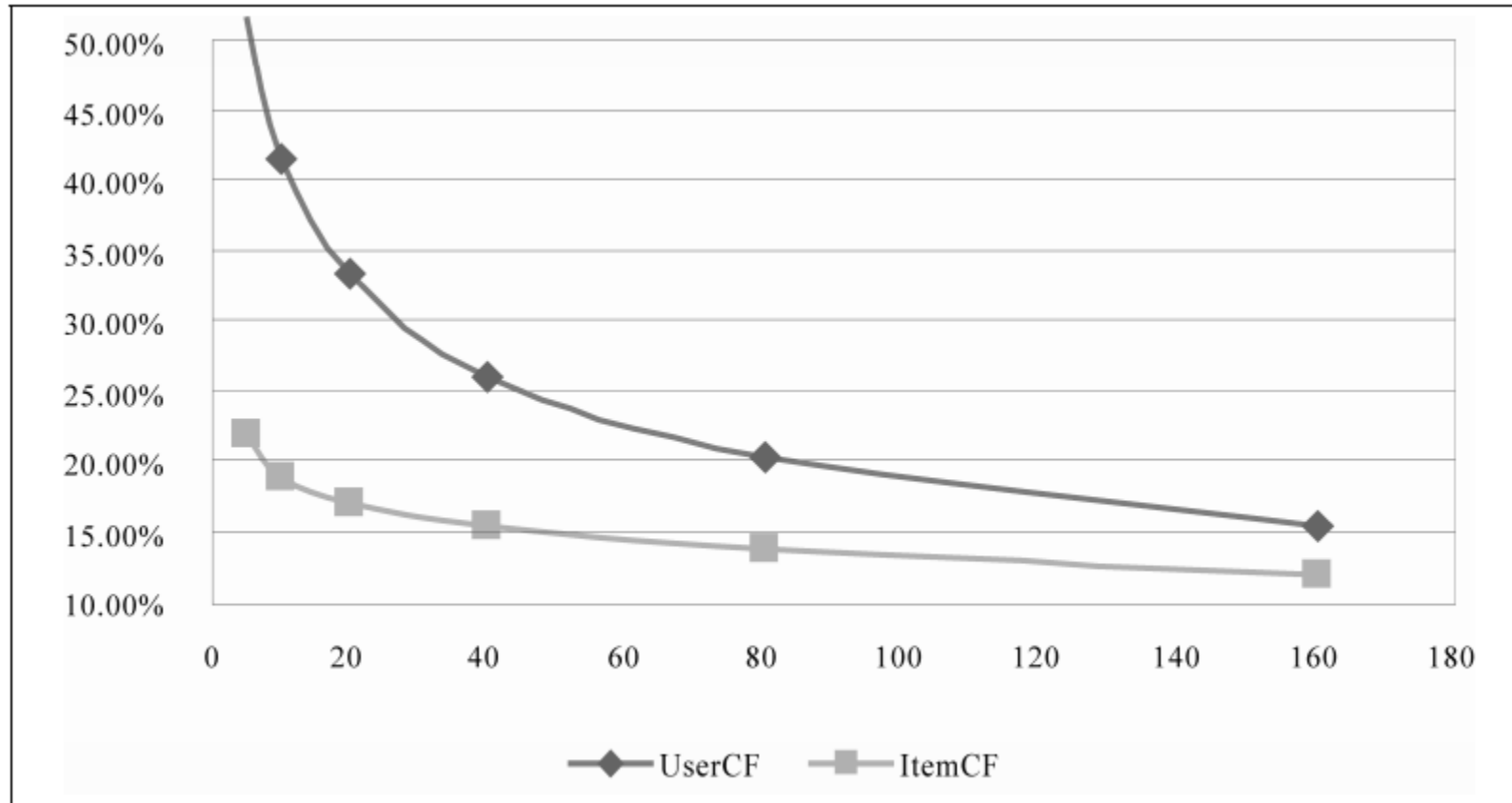


图2-14 UserCF和ItemCF算法在不同K值下的覆盖率曲线

# 基于用户的协同过滤算法

- UserCF和ItemCF的综合比较

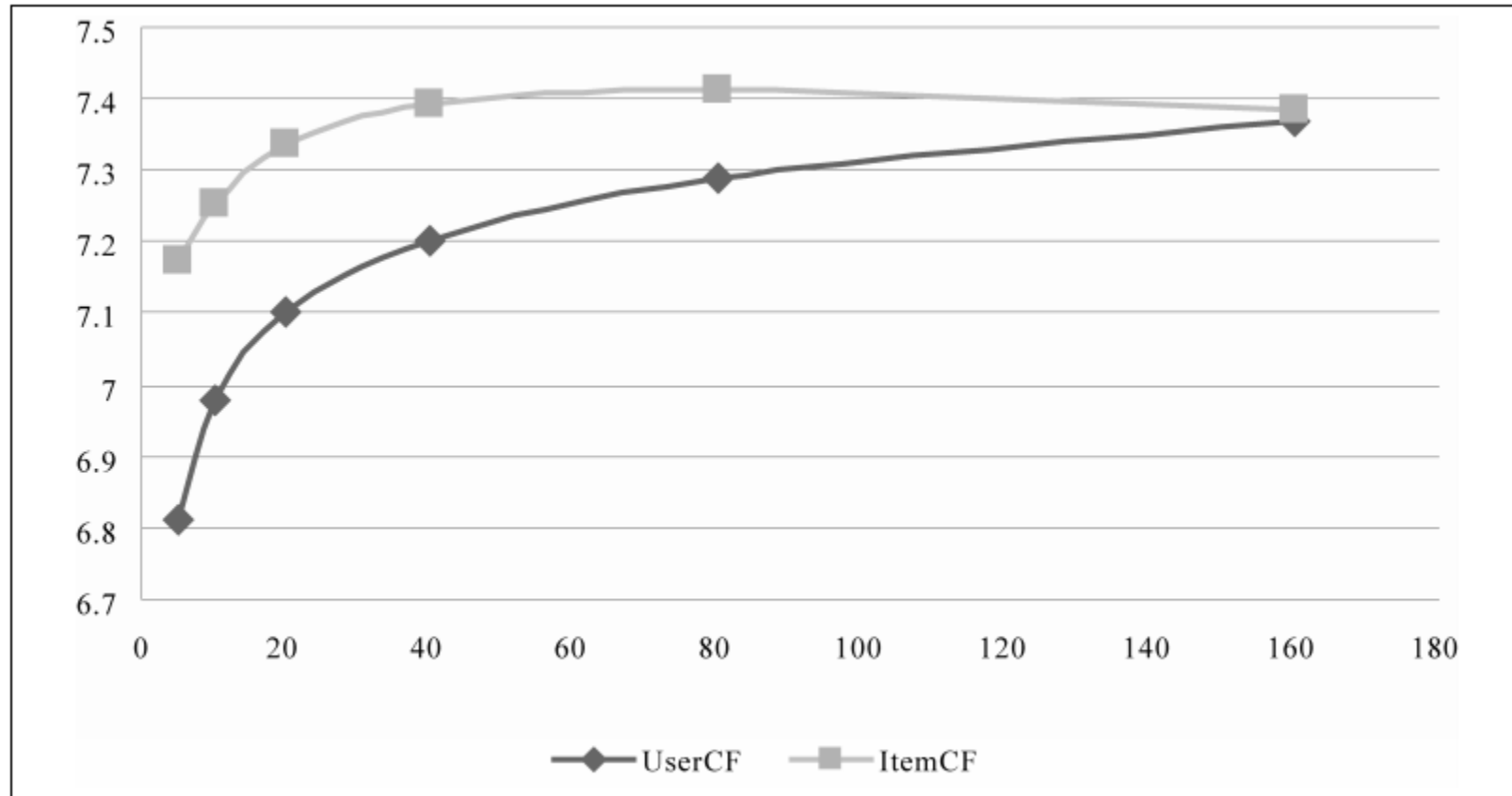


图2-15 UserCF和ItemCF算法在不同K值下的流行度曲线

# 基于用户的协同过滤算法

---

- 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。这些就不是协同过滤讨论的范畴了。

# 目录

---

1

用户行为数据简介

2

用户行为分析

3

实验设计和算法评测

4

基于邻域的算法

5

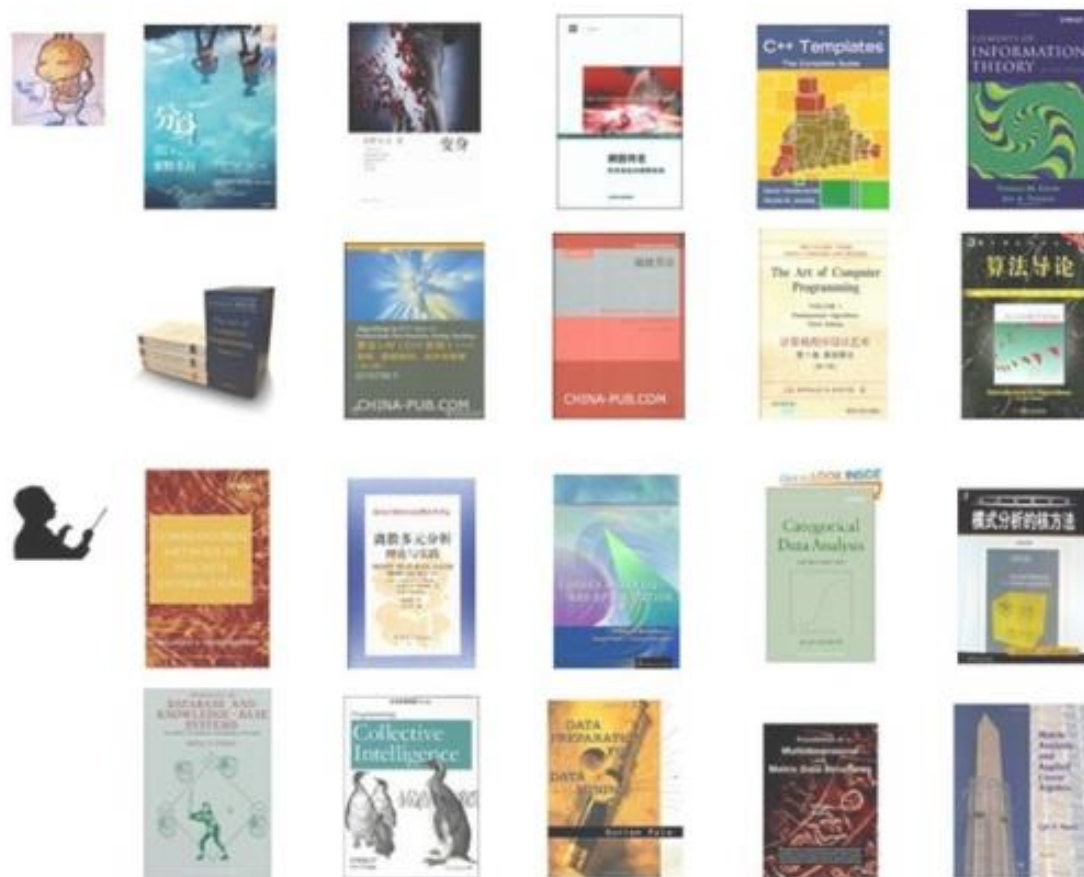
隐语义模型

6

基于图的模型

# 隐语义模型

- 隐语义模型是最近几年推荐系统领域最为热门的研究话题，它的核心思想是通过隐含特征(latent factor)联系用户兴趣和物品。





# 隐语义模型

- 从他们的阅读列表可以看出，用户A的兴趣涉及侦探小说、科普图书以及一些计算机技术书，而用户B的兴趣比较集中在数学和机器学习方面。那么如何给A和B推荐图书呢？
  - 对于UserCF，首先需要找到和他们看了同样书的其他用户（兴趣相似的用户），然后给他们推荐那些用户喜欢的其他书。
  - 对于ItemCF，需要给他们推荐和他们已经看的书相似的书，比如作者B看了很多关于数据挖掘的书，可以给他推荐机器学习或者模式识别方面的书。
  - 还有一种方法，可以对书和物品的兴趣进行分类。对于某个用户，首先得到他的兴趣分类，然后从分类中挑选他可能喜欢的物品。
- 总结一下，这个基于兴趣分类的方法大概需要解决3个问题。
  - 如何给物品进行分类？
  - 如何确定用户对哪些类的物品感兴趣，以及感兴趣的程度？
  - 对于一个给定的类，选择哪些属于这个类的物品推荐给用户，以及如何确定这些物品在一个类中的权重？

# 隐语义模型

- 对于第一个问题的简单解决方案是找编辑给物品分类，具有以下缺点：
  - 编辑的意见不能代表各种用户的意见。
  - 编辑很难控制分类的粒度。
  - 编辑很难给一个物品多个分类。
  - 编辑很难给出多维度的分类。
  - 编辑很难决定一个物品在某一个分类中的权重。
- 为了解决上面的问题，研究人员提出，从数据出发，自动地找到那些类，然后进行个性化推荐，于是，隐含语义分析技术（latent variable analysis）出现了。

# 隐语义模型

- 隐含语义分析技术因为采取基于用户行为统计的自动聚类，较好地解决了上面提出的问题：
  - 编辑的意见不能代表各种用户的意见，但隐含语义分析技术的分类来自对用户行为的统计，代表了用户对物品分类的看法。隐含语义分析技术和ItemCF在物品分类方面的思想类似，如果两个物品被很多用户同时喜欢，那么这两个物品就很有可能属于同一个类。
  - 编辑很难控制分类的粒度，但隐含语义分析技术允许我们指定最终有多少个分类，这个数字越大，分类的粒度就会越细，反正分类粒度就越粗。
  - 编辑很难给一个物品多个分类，但隐含语义分析技术会计算出物品属于每个类的权重，因此每个物品都不是硬性地被分到某一个类中。
  - 编辑很难给出多维度的分类，但隐含语义分析技术给出的每个分类都不是同一个维度的，它是基于用户的共同兴趣计算出来的，如果用户的共同兴趣是某一个维度，那么LFM给出的类也是相同的维度。
  - 编辑很难决定一个物品在某一个分类中的权重，但隐含语义分析技术可以通过统计用户行为决定物品在每个类中的权重，如果喜欢某个类的用户都会喜欢某个物品，那么这个物品在这个类中的权重就可能比较高。

# 隐语义模型

---

- 隐含语义分析技术包括：
  - pLSA
  - LDA
  - 隐含类别模型 ( latent class model )
  - 隐含主题模型 ( latent topic model )
  - 矩阵分解 ( matrix factorization )

# 隐语义模型

- LFM通过如下公式计算用户u对物品i的兴趣：

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

- 这个公式中 $p_{u,k}$ ，和 $q_{i,k}$ ，是模型的参数，其中 $p_{u,k}$ ，度量了用户u的兴趣和第k个隐类的关系，而 $q_{i,k}$ ，度量了第k个隐类和物品i之间的关系。

# 隐语义模型

- 在隐性反馈数据集上应用LFM解决TopN推荐的第一个关键问题就是如何给每个用户生成负样本。Rong Pan探讨了如下方法：
  - 对于一个用户，用他所有没有过行为的物品作为负样本。
  - 对于一个用户，从他没有过行为的物品中均匀采样出一些物品作为负样本。
  - 对于一个用户，从他没有过行为的物品中采样出一些物品作为负样本，但采样时，保证每个用户的正负样本数目相当。
  - 对于一个用户，从他没有过行为的物品中采样出一些物品作为负样本，但采样时，偏重采样不热门的物品。

# 隐语义模型

- KDD Cup发现对负样本采样时应该遵循以下原则：
  - 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
```

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

# 隐语义模型

- 我们通过离线实验评测LFM的性能。
  - 首先，我们在MovieLens数据集上用LFM计算出用户兴趣向量 $p$ 和物品向量 $q$
  - 然后对于每个隐类找出权重最大的物品。

表2-13 MovieLens数据集中根据LFM计算出的不同隐类中权重最高的物品

1 (科幻、惊悚)	3 (犯罪)	4 (家庭)	5 (恐怖、惊悚)
《隐形人》(The Invisible Man, 1933)	《大白鲨》(Jaws, 1975)	《101真狗》(101 Dalmatians, 1996)	《女巫布莱尔》(The Blair Witch Project, 1999)
《科学怪人大战狼人》(Frankenstein Meets the Wolf Man, 1943)	《致命武器》(Lethal Weapon, 1987)	《回到未来》(Back to the Future, 1985)	《地狱来的房客》(Pacific Heights, 1990)
《哥斯拉》(Godzilla, 1954)	《全面回忆》(Total Recall, 1990)	《土拨鼠之日》(Groundhog Day, 1993)	《异灵骇客2之恶灵归来》(Stir of Echoes: The Homecoming, 2007)
《星球大战3：武士复仇》(Star Wars: Episode VI - Return of the Jedi, 1983)	《落水狗》(Reservoir Dogs, 1992)	《泰山》(Tarzan, 2003)	《航越地平线》(Dead Calm, 1989)
《终结者》(The Terminator, 1984)	《忠奸人》(Donnie Brasco, 1997)	《猫儿历险记》(The Aristocats, 1970)	《幻象》(Phantasm, 1979)
《魔童村》(Village of the Damned, 1995)	《亡命天涯》(The Fugitive, 1993)	《森林王子2》(The Jungle Book 2, 2003)	《断头谷》(Sleepy Hollow, 1999)
《异形》(Alien, 1979)	《夺宝奇兵3》(Indiana Jones and the Last Crusade, 1989)	《当哈利遇到莎莉》(When Harry Met Sally..., 1989)	《老师不是人》(The Faculty, 1998)
《异形2》(Aliens, 1986)	《威胁2：社会》(Menace II Society, 1993)	《蚁哥正传》(Antz, 1998)	《苍蝇》(The Fly, 1958)
《天魔续集》(Damien: Omen II, 1978)	《辣手神探》(Lashou shentan, 1992)	《小姐与流浪汉》(Lady and the Tramp, 1955)	《鬼哭神嚎》(The Amityville Horror, 1979)
《魔鬼怪婴》(Rosemary's Baby, 1968)	《真实罗曼史》(True Romance, 1993)	《飞天法宝》(Flubber, 1997)	《深渊》(The Abyss, 1989)



# 隐语义模型

- 我们通过实验对比了LFM在TopN推荐中的性能。在LFM中，重要的参数有4个：
  - 隐特征的个数 $F$ ；
  - 学习速率 $\alpha$ ；
  - 正则化参数 $\lambda$ ；
  - 负样本/正样本比例  $\text{ratio}$
- $\text{ratio}$ 参数对LFM的性能影响最大。因此，固定 $F=100$ 、 $\alpha=0.02$ 、 $\lambda=0.01$ ，然后研究负样本/正样本比例 $\text{ratio}$ 对推荐结果性能的影响。

# 隐语义模型

- 我们通过实验对比了LFM在TopN推荐中的性能。

表2-14 Netflix数据集中LFM算法在不同 $F$ 参数下的性能

ratio	准 确 率	召 回 率	覆 盖 率	流 行 度
1	21.74%	10.50%	51.19%	6.5140
2	24.32%	11.75%	53.17%	6.5458
3	25.66%	12.39%	50.41%	6.6480
5	26.94%	13.01%	44.25%	6.7899
10	27.74%	13.40%	33.87%	6.9552
20	27.37%	13.22%	24.30%	7.1025

# 隐语义模型

- 基于LFM的实际系统的例子,雅虎首页的界面包括不同的模块,比如左侧的分类导航列表、中间的热门新闻列表、右侧的最近热门话题列表。



# 隐语义模型

- 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。
- **离线计算的时间复杂度** 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。
- **在线实时推荐** LFM不太适合用于物品数非常庞大的系统。LFM不能进行在线实时推荐。
- **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。

# 目录

---

1

用户行为数据简介

2

用户行为分析

3

实验设计和算法评测

4

基于邻域的算法

5

隐语义模型

6

基于图的模型

# 基于图的模型

- 用户行为很容易用二分图表示，因此很多图的算法都可以用到推荐系统中。

令  $G(V, E)$  表示用户物品二分图，其中  $V = V_u \cup V_i$  由用户顶点集合  $V_u$  和物品顶点集合  $V_i$  组成。对于数据集中每一个二元组  $(u, i)$ ，图中都有一对对应的边  $e(v_u, v_i)$ ，其中  $v_u \in V_u$  是用户  $u$  对应的顶点， $v_i \in V_i$  是物品  $i$  对应的顶点。图2-18是一个简单的用户物品二分图模型，其中圆形节点代表用户，方形节点代表物品，圆形节点和方形节点之间的边代表用户对物品的行为。比如图中用户节点A和物品节点a、b、d相连，说明用户A对物品a、b、d产生过行为。

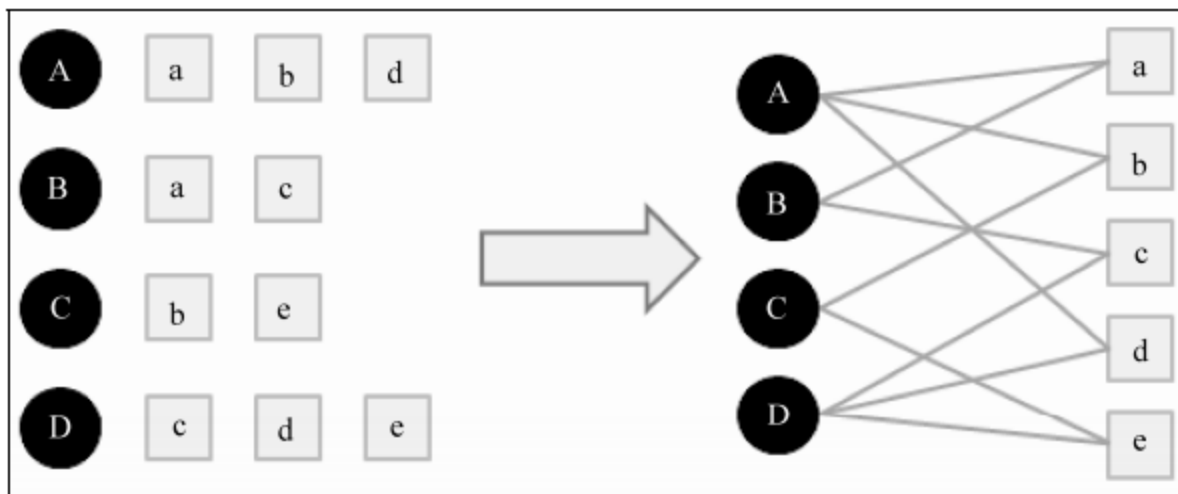


图2-18 用户物品二分图模型

# 基于图的模型

- 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。
- 度量图中两个顶点之间相关性的方法取决于下面3个因素：
  - 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一般具有如下特征：
  - 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。

# 基于图的模型

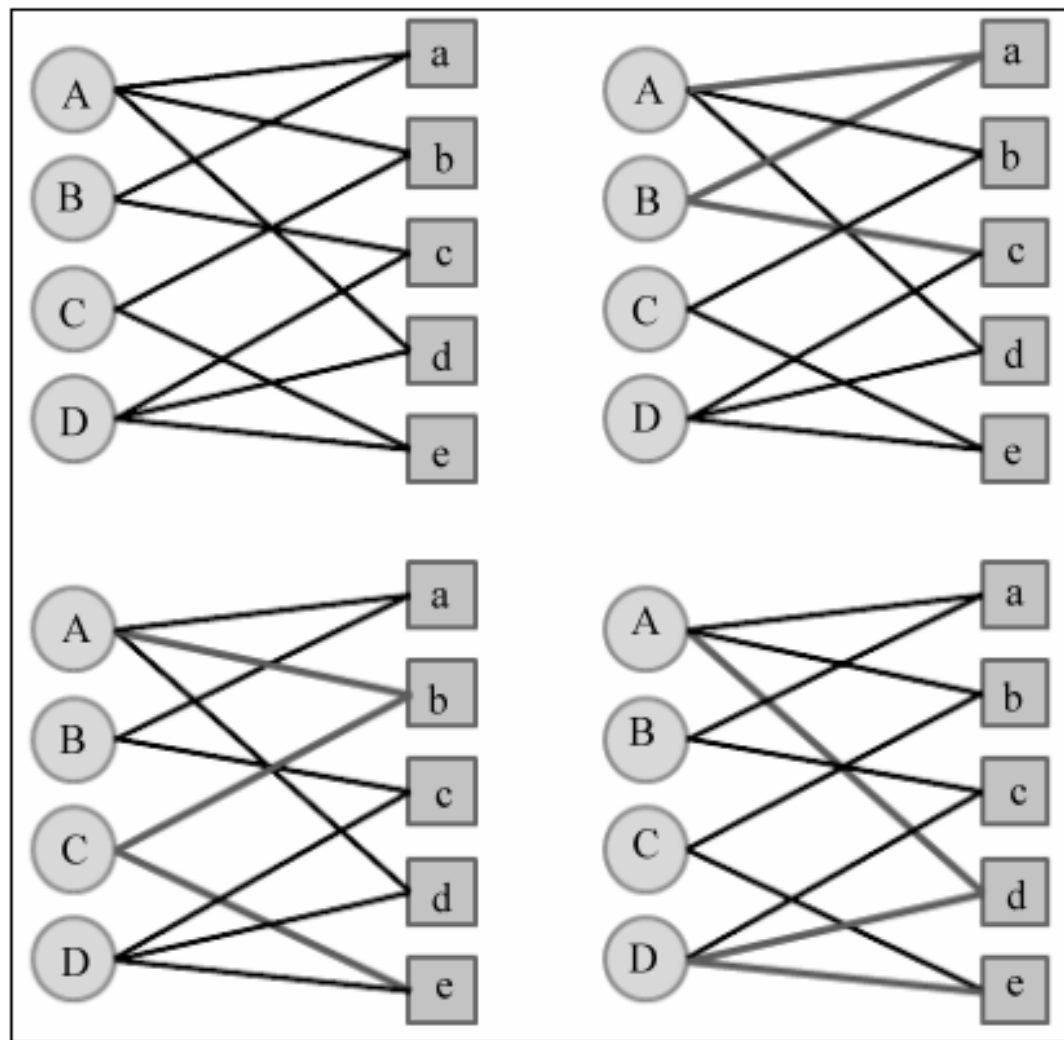


图2-19 基于图的推荐算法示例



# 基于图的模型

- 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 $\alpha$ 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。

如果将上面的描述表示成公式，可以得到如下公式：

$$\text{PR}(v) = \begin{cases} \alpha \sum_{v' \in \text{in}(v)} \frac{\text{PR}(v')}{|\text{out}(v')|} & (v \neq v_u) \\ (1 - \alpha) + \alpha \sum_{v' \in \text{in}(v)} \frac{\text{PR}(v')}{|\text{out}(v')|} & (v = v_u) \end{cases}$$

# 基于图的模型

- Python代码简单实现

```
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
```

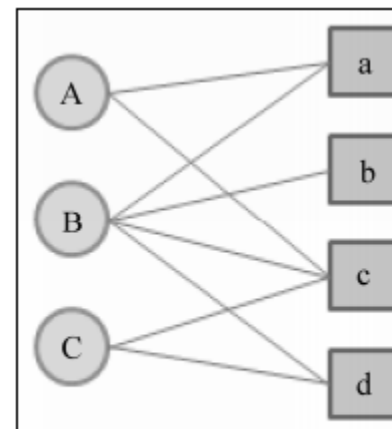
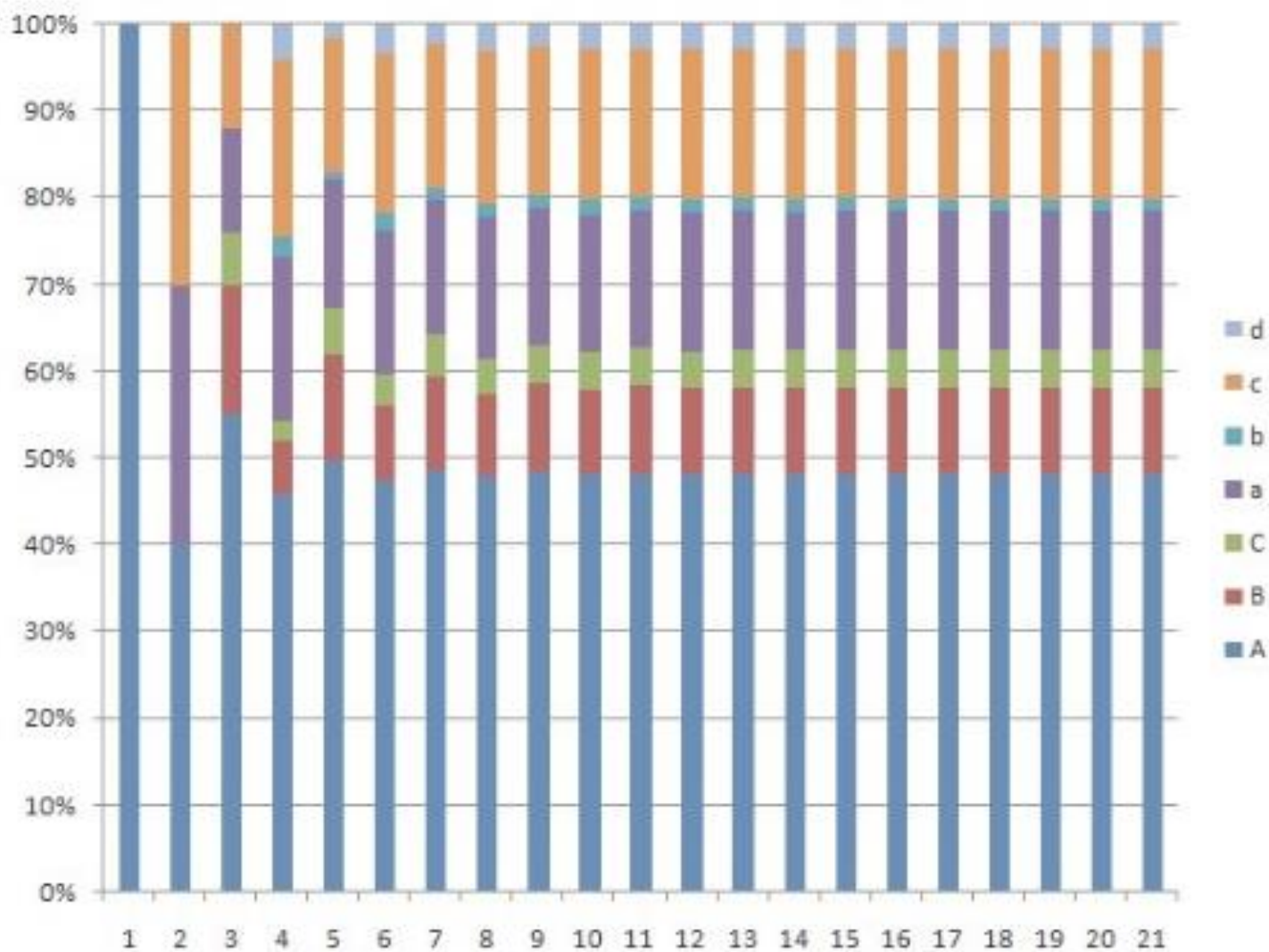


图2-20 PersonalRank的简单例子

# 基于图的模型



# 基于图的模型

- MovieLens的数据集上评测了PersonalRank算法结果

表2-15 MovieLens数据集中PersonalRank算法的离线实验结果

$\alpha$	准 确 率	召 回 率	覆 盖 率	流 行 度
0.8	16.45%	7.95%	3.42%	7.6928

# 内容回顾

---

1

用户行为数据简介

2

用户行为分析

3

实验设计和算法评测

4

基于邻域的算法

5

隐语义模型

6

基于图的模型

Thank you !