智能推荐算法

> 基于内容的推荐算法

根据内容的相似度（静态的东西）进行推荐，内容不好提取的可以采取**贴标签**的形式来区分计算内容的**相似程度**。然后**根据用户的喜好设置，关注等**进行**相似内容**推荐。

> 协同过滤推荐算法

根据**动态信息**来进行推荐，即推荐的过程是**自动的**，推荐结果的产生是系统从用户的购买行为或浏览记录等**隐式信息**拿到的，无需用户通过填表格等方式来明确自己的喜好。因为这些数据都是**要读到内存中进行运算的**，所以又叫**基于内存的协同过滤**（Memory-based Collaborative Filtering）,另一种协同过滤算法则是**基于模型的协同过滤**(Model-based Collaborative Filtering)；m个物品，m个用户的数据，只有部分用户和部分数据之间是有评分数据的，其它部分评分是空白，此时我们要用已有的部分稀疏数据来预测那些空白的物品和数据之间的评分关系，找到最高评分的物品推荐给用户。对于这个问题，用机器学习的思想来建模解决，主流的方法可以分为：用关联算法，聚类算法，分类算法，回归算法，矩阵分解，神经网络,图模型以及隐语义模型来解决。

而基于内存的协同过滤又有两种：

1. 基于**user**的协同过滤（用户相似度）：通过相似用户的喜好来推荐
2. 基于**item**的协同过滤（内容相似度）：通过用户对项目的不同评分推荐可能让用户打高评分的项目，是项目之间的相似度。

> 混合推荐

这个类似我们机器学习中的集成学习，博才众长，通过多个推荐算法的结合，得到一个更好的推荐算法，起到三个臭皮匠顶一个诸葛亮的作用。比如**通过建立多个推荐算法的模型，最后用投票法决定最终的推荐结果**。混合推荐理论上不会比单一任何一种推荐算法差，但是使用混合推荐，算法复杂度就提高了，在实际应用中有使用，但是并没有单一的协调过滤推荐算法，比如逻辑回归之类的二分类推荐算法广泛。

> 基于规则的推荐

这类算法常见的比如基于最多用户点击，最多用户浏览等，属于大众型的推荐方法，在目前的大数据时代**并不主流。**

> 基于人口统计信息的推荐

这一类是最简单的推荐算法了，它只是简单的根据系统用户的基本信息发现用户的相关程度，然后进行推荐，目前在大型系统中**已经较少使用。**

协同过滤推荐

是推荐算法中目前最主流的种类，花样繁多，在工业界已经有了很多广泛的应用。

它的优点是不需要太多特定领域的知识，可以通过基于统计的机器学习算法来得到较好的推荐效果。最大的优点是工程上容易实现，可以方便应用到产品中。目前绝大多数实际应用的推荐算法都是协同过滤推荐算法。

概述

协同过滤(Collaborative Filtering)包括**在线的协同**和**离线的过滤**两部分。

在线协同：通过在线数据找到用户可能喜欢的物品

离线过滤：过滤掉一些不值得推荐的数据

协同过滤的模型一般为m个物品，m个用户的数据，只有**部分用户和部分数据之间是有评分数据的**，**其它部分**评分是**空白**，此时我们要**用已有的部分稀疏数据**来**预测**那些**空白的物品和数据之间的评分关系**，找到最高评分的物品推荐给用户。

分类：

1. 基于用户(user-based)的协同过滤

   主要考虑**用户和用户之间的相似度**，只要找出相似用户喜欢的物品，并预测目标用户对对应物品的评分，就可以找到评分最高的若干个物品推荐给用户。

2. 基于项目（item-based）的协同过滤

   找到**物品和物品之间的相似度**，只有找到了目标用户对某些物品的评分，那么我们就可以对相似度高的类似物品进行预测，将评分最高的若干个相似物品推荐给用户。

基于用户的协同过滤和基于项目的系统过滤的对比	**适用**
基于用户的协同过滤需要在线找用户和用户之间的相似度关系，计算复杂度肯定会比基于基于项目的协同过滤高。但是可以帮助用户找到新类别的有惊喜的物品。	如果是大型的推荐系统来说，则可以考虑基于用户的协同过滤。
基于项目的协同过滤，由于考虑的物品的相似性一段时间不会改变，因此可以很容易的离线计算，准确度一般也可以接受，但是推荐的多样性来说，就很难带给用户惊喜了。	一般对于小型的推荐系统来说，基于项目的协同过滤肯定是主流

3. 基于模型(model based)的协同过滤

基于用户的协同过滤例子

参考博客：**用python做推荐系统（一）**

> 项目名 `UserBasedCollaborativeFiltering`

```python
#  Python学习/PythonWorkSpace/UserBasedCollaborativeFiltering/venv/Include/com/main.py
import math

userid = ""      # 找的目标用户

# 数据预处理
def load_data():
    f = open('u.data')
    user_list={}
    for line in f:
        (user,movie,rating,ts) = line.split('\t')
        user_list.setdefault(user,{})
        user_list[user][movie] = float(rating)
    return user_list

# 使用欧几里得距离:将两个人对同一部电影的评价相减平方再开平方
def calculate():
    list = load_data()
    user_diff = {}
    user = str(userid)
    for movies in list[user]:
        for people in list.keys():
            user_diff.setdefault(people, {})
            for item in list[people].keys():
                if item == movies:
                    diff = math.sqrt(pow(list[user][movies] - list[people][item], 2))
                    user_diff[people][item] = diff
    return user_diff

# 求距离平均值 -> 求相似度 = 1 / (1 + 距离平均值) 加1防止除0
# 相似度与距离成反比
def people_rating():
    user_diff = calculate()
    rating = {}
    for people in user_diff.keys():
        rating.setdefault(people, {})
        a = 0
        b = 0
        for score in user_diff[people].values():
            a += score
            b += 1
        rating[people] = float(1 / (1 + (a/b)))
    return rating

# 排序 找出几个相似度比较高的用户
def top_list():
    list = people_rating()
    items = list.items()   # 以列表返回可遍历的(键, 值) 元组数组
    top = [[v[1], v[0]] for v in items]
    top.sort(reverse = True)
    print(top[0:5]) # print出相似度最高的五个
    return top[0:5]

# 找出前两位最相似的用户看过但是userid用户没有看过的电影 并且评分为5的电影推荐给userid用户
def find_rec():
    rec_list = top_list()
    user1 = rec_list[1][1]
    user2 = rec_list[2][1]
```

```
    all_list = load_data()
    print("第一个用户推荐的电影:")
    for k, v in all_list[user1].items():
        if k not in all_list[str(userid)].keys() and v == 5:
            print(k)

    print("第二个用户推荐的电影:")
    for k, v in all_list[user2].items():
        if k not in all_list[str(userid)].keys() and v == 5:
            print(k)

# 主函数
userid = 7
find_rec()
```

```
输出:
[[1.0, '7'], [1.0, '547'], [1.0, '384'], [0.75, '775'], [0.75, '558']]
第一个用户推荐的电影:
316
345
313
302
第二个用户推荐的电影:
272
316
313
289
302
```

实现协同过滤步骤

参考网页地址： **[推荐系统之协同过滤（CF）算法详解和实现]**

收集用户偏好

用户行为	类型	特征	作用
评分	显式	整数量化的偏好，可能的取值是 [0, n]；n 一般取值为 5 或者是 10	通过用户对物品的评分，可以精确的得到用户的偏好
投票	显式	布尔量化的偏好，取值是 0 或 1	通过用户对物品的投票，可以较精确的得到用户的偏好
转发	显式	布尔量化的偏好，取值是 0 或 1	通过用户对物品的投票，可以精确的得到用户的偏好。 如果是站内，同时可以推理得到被转发人的偏好（不精确）
保存书签	显式	布尔量化的偏好，取值是 0 或 1	通过用户对物品的投票，可以精确的得到用户的偏好。
标记标签(Tag)	显式	一些单词，需要对单词进行分析，得到偏好	通过分析用户的标签，可以得到用户对项目的理解，同时可以分析出用户的情感：喜欢还是讨厌
评论	显式	一段文字，需要进行文本分析，得到偏好	通过分析用户的评论，可以得到用户的情感：喜欢还是讨厌
点击流（查看）	隐式	一组用户的点击，用户对物品感兴趣，需要进行分析，得到偏好	用户的点击一定程度上反映了用户的注意力，所以它也可以从一定程度上反映用户的喜好。
页面停留时间	隐式	一组时间信息，噪音大，需要进行去噪，分析，得到偏好	用户的页面停留时间一定程度上反映了用户的注意力和喜好，但噪音偏大，不好利用。
购买	隐式	布尔量化的偏好，取值是 0 或 1	用户的购买是很明确的说明这个项目它感兴趣。

找到相似的用户和物品

> 最基本的几种计算相似度的方法

- **欧几里得距离（Education Distance）**

  假设 x, y 是 n 维空间的两个点，它们之间的**欧几里得距离**为：

  $$d(x,y) = \sqrt{(\sum(x_i - y_i)^2)}$$

  当 n = 2 时，欧几里得距离就是平面上两个点的距离

  当用欧几里德距离表示**相似度**，一般采用以下公式进行转换：距离越小，相似度越大

  $$sim(x,y) = \frac{1}{1 + d(x,y)}$$

- **皮尔逊相关系数（Pearson Correlation Coefficient）**

  皮尔逊相关系数一般用于计算两个定距变量间联系的紧密程度，它的取值在[-1, 1]之间。

`sx`、`sy`是 x 和 y 的样品标准差。

$$p(x,y)=\frac{\sum x_iy_i-n\overline{xy}}{(n-1)s_xs_y}=\frac{n\sum x_iy_i-\sum x_i\sum y_i}{\sqrt{n\sum x_i^2-(\sum x_i)^2}\sqrt{n\sum y_i^2-(\sum y_i)^2}}$$

- **Cosine 相似度（Cosine Similarity）**

  Cosine 相似度被广泛应用于计算文档数据的相似度：

$$T(x,y)=\frac{x\cdot y}{\|x\|^2+\|y\|^2}=\frac{\sum x_iy_i}{\sqrt{\sum x_i^2}\sqrt{\sum y_i^2}}$$

- **Tanimoto 系数（Tanimoto Coefficient）**

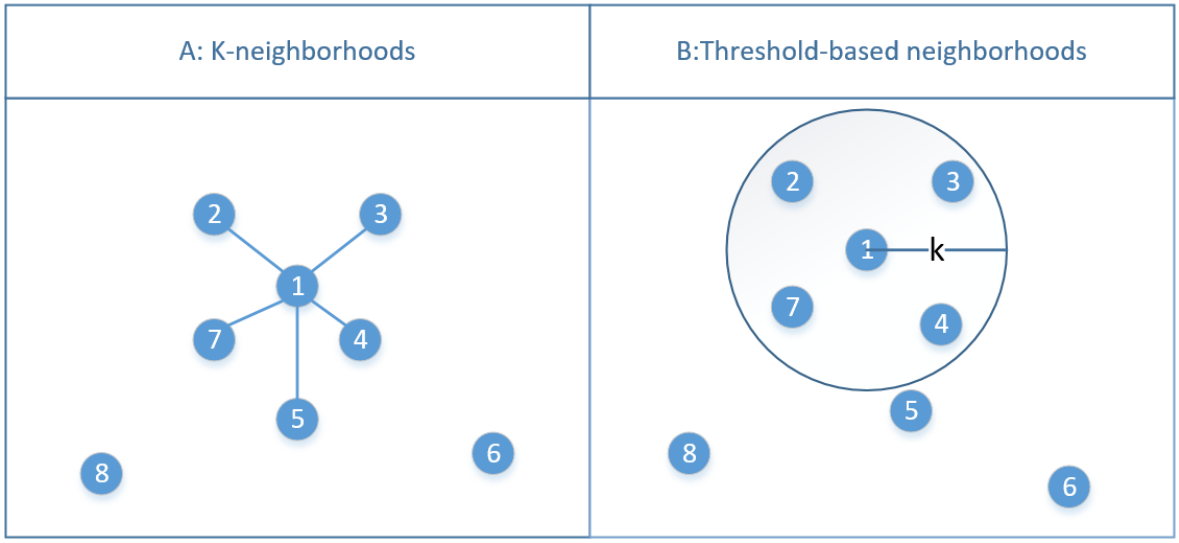
  Tanimoto 系数也称为 Jaccard 系数，是 Cosine 相似度的扩展，也多用于计算文档数据的相似度：

$$T(x,y)=\frac{x\cdot y}{\|x\|^2+\|y\|^2-x\cdot y}=\frac{\sum x_iy_i}{\sqrt{\sum x_i^2}+\sqrt{\sum y_i^2}-\sum x_iy_i}$$

> 相似邻居的计算

解决如何根据相似度找到用户-物品的邻居，常用挑选邻居的原则分为两类：

图 1.相似邻居计算示意图



- 固定数量的邻居：K-neighborhoods 或者 Fix-size neighorhoods

  不论邻居的“远近”，只取最近的 K 个，作为其邻居。如上图中的 A，假设要计算点 1 的 5- 邻居，那么根据点之间的距离，我们取最近的 5 个点，分别是点 2，点 3，点 4，点 7 和点 5。但很明显我们可以看出，这种方法对于孤立点的计算效果不好，因为要**取固定个数的邻居**，当它附近没有足够多比较相似的点，**就被迫取一些不太相似的点作为邻居，这样就影响了邻居相似的程度**，比如图 1 中，点 1 和点 5 其实并不是很相似。

- 基于相似度门槛的邻居：Threshold-based neighborhoods

  与计算固定数量的邻居的原则不同，基于相似度门槛的邻居计算是对邻居的远近进行**最大值的限制，落在以当前点为中心，距离为 K 的区域中的所有点都作为当前点的邻居**，这种方法计算得到的邻居个数不确定，但**相似度不会出现较大的误差。**如图 1 中的 B，从点 1 出发，计算相似度在 K 内的邻居，得到点 2，点 3，点 4 和点 7，这种方法计算出的邻居的相似度程度比前一种优，尤其是对孤立点的处理。

计算推荐

> 基于用户的协同过滤

- **基本思想**

  基于用户对物品的偏好找到相邻邻居用户，然后将邻居用户喜欢的推荐给当前用户。

- **计算**

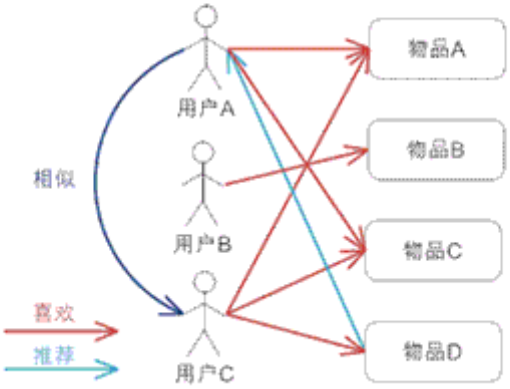
  将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度，找到 K 邻居后，根据邻居的相似度权重以及他们对物品的偏好，预测当前用户没有偏好的未涉及物品，计算得到一个排序的物品列表作为推荐。

- **举例**

  对于用户 A，根据用户的历史偏好，这里只计算得到一个邻居 - 用户 C，然后将用户 C 喜欢的物品 D 推荐给用户 A。

图 2.基于用户的 CF 的基本原理

用户/物品	物品A	物品B	物品C	物品D
用户A	√		√	推荐
用户B		√		
用户C	√		√	√



> 基于物品的协同过滤

- **基本思想**

  基于用户对物品的偏好找到相似的物品，然后根据用户的历史偏好，推荐相似的物品给他。

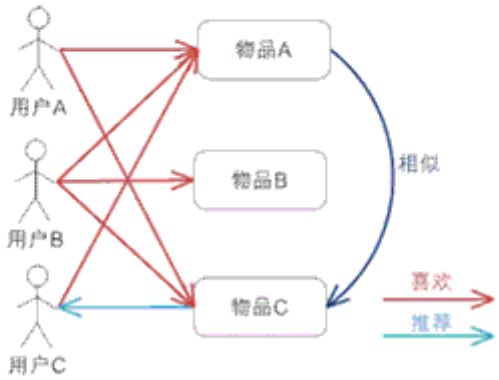
- **计算**

  将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度，得到物品的相似物品后，根据用户历史的偏好预测当前用户还没有表示偏好的物品，计算得到一个排序的物品列表作为推荐。

- **举例**

  图 3 给出了一个例子，对于物品 A，根据所有用户的历史偏好，喜欢物品 A 的用户都喜欢物品 C，得出物品 A 和物品 C 比较相似，而用户 C 喜欢物品 A，那么可以推断出用户 C 可能也喜欢物品 C。

用户/物品	物品A	物品B	物品C
用户A	√		√
用户B	√	√	√
用户C	√		推荐



> User CF、 Item CF对比

- **计算复杂度**

  Item CF 从性能和复杂度上比 User CF 更优，其中的一个主要原因就是对于一个**在线网站**，**用户的数量往往大大超过物品的数量，同时物品的数据相对稳定**，因此计算物品的相似度不但计算量较小，同时也不必频繁更新。 `（适合Item CF）`

  但我们往往忽略了这种情况只适应于提供商品的电子商务网站，对于**新闻，博客或者微内容的推荐系统**，情况往往是相反的，**物品的数量是海量的，同时也是更新频繁的**，所以单从复杂度的角度，这两个算法在不同的系统中各有优势，推荐引擎的设计者需要根据自己应用的特点选择更加合适的算法。 `（适合User CF）`