摘 要

随着互联网的发展，网络商业突飞猛进，种类越来越丰富。由互联网带起的商业模式不断的在改变着我们的生活。然而这些基于网络的营销模式也缺乏一些实体店有的但是网络应用很难有的服务：个性化推荐。当用户开始浏览某个现代商业网站或打开某个商业软件，出现的是永远看不完的商品。需求明确的用户可以通过搜索引擎找到自己需要的产品，而对于那些需求不明确的用户甚至只是想随便逛一逛看一看的用户，给这些用户进行如同实体店里面那样的个性化推荐可以增加潜在的产品销量。既然不能人工进行个性化推荐，我们自然就想到利用大数据分析自动给用户进行个性化推荐。推荐系统就这样诞生了。

一个高质量的推荐算法可以十分有效的提升这些网络商业应用的利润。我的论文就是来研究一种高质量的推荐算法：SLIM 推荐算法。我在本论文对这种推荐算法进行详细的过程分析，测试并评估推荐结果。我的实验表明，SLIM 推荐算法是一种质量高，实用性强的推荐算法。

关键词：推荐系统，SLIM 推荐算法。

ABSTRACT

With the development of internet, online business advances rapidly, and becomes various. These online businesses are changing our life. However these online businesses have a really hard service to build: personalized recommendation. if you are browsing an online business website or app rather than a physical store, you can see endless products. If what you need is clear, you can find what you want through a search engine. However, some users don't have a clear need or they just go around and look. Giving a personalized recommendation to those users whose need is unclear can potentially increase the sales of products. So if we can't give every user a manual personalized recommendation, we can give them personalized recommendation which takes advantage of big data analysis. This is where the recommendation system was born.

A high quality recommendation algorithm can promote online businesses' profit effectively. My paper is purposed on studying a high quality recommendation algorithm: SLIM. I analyze this algorithm carefully in the paper, test and evaluate the result. My study shows that SLIM is a high quality and quite practical recommendation algorithm.

Keywords: recommendation system, SLIM.

目 录

第一章 绪 论

1.1.背景及研究意义

近年来，互联网科技飞速发展，我们已经从以前的信息匮乏的时代进入了一个信息爆发的时代,人们所能接触到的信息量已经远远超过人类能吸收的信息量。互联网将每个人类个体连到了一起，人们购物不再需要身临其境去选取数量极其有限的商品，而只需要联网的电子设备稍作点击即可找到看不完的商品；不再像从前听一次音乐很麻烦，人们在只有手掌大小的音乐设备上通过网络可以轻松听到全世界几乎任何音乐；看电影不再麻烦，大多数电影网站都可以得到我们想要的资源。这些购物网站，音乐网站，电影网站等都是互联网时代的产物。然而事情往往不像表面看起来那么完美。由于信息的过载，用户往往面临着难以找到最适合自己的商品，即使需求明确，也难以找到自己最想要的东西。同样由于信息过载，物品贩卖商难以向数量巨大的用户群体提供个性化推荐物品，每个用户的需求是不同的，想要同时根据每个用户推荐他们各自适合的物品是商家需要解决的问题。此时，推荐系统就诞生了。推荐系统的目标就是来解决这个互联网时代由于信息过载导致的卖家与买家之间的需求矛盾。在为每个用户推荐出其可能最需要或最适合的物品的同时,增加商家的产品销量,实现卖家和买家双赢的局面。这既是推荐系统的意义所在。

1.2.研究现状

随着这几年来机器学习飞速发展，机器学习被大规模应用于推荐系统，传统的通过余弦相似度等相似度度量算法计算物品之间或用户之间的相似度来推荐进行的协同过滤算法，以及基于标签推荐的算法等这些基于规则的推荐算法在逐步淘汰，其核心算法或思想渐渐的被机器学习算法所取代。机器学习算法相比于传统的算法优势巨大，其容易发现一些物品用户等事物之间极其隐晦的关系。

长尾分布也是推荐系统一直要应对的问题。长尾分布可以这样理解：一般在一个系统中，老用户数量极少，而只有老用户对冷门物品的行为较多，系统中绝

大部分是新用户，而新用户往往只对少数很流行的物品有行为。物品的流行度和用户的活跃度都近似长尾分布。由于这种分布规律，推荐算法很容易就倾向于多推荐那些很热门的物品，导致那些冷门的物品被逐渐遗忘。然而机器学习也能更好的应对长尾分布，得到更精确的推荐结果。

推荐算法分为很多类型，例如基于内容的推荐，利用用户行为的推荐，基于标签的推荐，社交系统的推荐，广告推荐，利用上下文推荐（根据时间地点人物等内容进行推荐）等。由于我研究的 SLIM 推荐算法是基于用户隐反馈行为（即数据集只包括什么用户对什么物品有过行为的内容）的推荐算法，所以我会主要讨论基于用户行为的推荐系统。一般来讲，基于用户行为的推荐算法主要有基于邻域的算法和矩阵分解算法。基于邻域的算法总是要得到一个用户之间关系矩阵或物品之间关系矩阵，通过用户之间关系矩阵来对用户进行推荐与其相似的用户有过行为的物品，通过物品之间关系矩阵对用户推荐这个用户本身有过行为的物品的相似物品。而矩阵分解算法一般通过矩阵分解来给用户物品行为矩阵降维来得到用户物品潜在的关系，从而基于某个模型进行推荐。SLIM 算法由论文[2] SLIM: Sparse Linear Methods for Top-N Recommender Systems (Xia Ning and George Karypis Computer Science & Engineering University of Minnesota, Minneapolis, MN)提出，是一种基于邻域的使用机器学习的推荐算法，同时根据论文[2]的实验数据来看也是 top-n 推荐准确率几乎最高的推荐算法。

1.3. 论文主要工作

我的论文是这样安排的：第一章绪论主要谈谈推荐系统的背景，意义，为什么需要推荐系统以及目前大家的研究情况；第二章简单介绍 SLIM 推荐算法的模型以及用到了哪些算法，输入输出是什么，对数据有什么要求；第三章根据算法设计实现思路，通过伪代码探讨设计的可行性以及时间复杂度；第四章代码实现设计的实现算法的方法，并通过实验测试评估算法的各项评测指标，探讨 SLIM 算法的各种优缺点；第五章总结整个实验过程。

第二章　模型简介

2.1. 算法概述

本章将会全面介绍 SLIM（Sparse Linear Methods）推荐算法的模型，以及算法的全过程。

SLIM 推荐算法在论文[2]里面提出，作为一种使用机器学习的推荐算法，相比于其它传统的推荐算法自然有了很多的优势，各项评测指标理论上都会比其更优。

SLIM 推荐算法专注于在 top-n 推荐中取得一个很好的离线实验准确率，召回率，覆盖率等参数。此算法由于在用户物品行为矩阵中把用户没有行为的物品所对应的矩阵值设为 0，进行机器学习后，这些值会趋近于 0，没有任何评分意义，所以 SLIM 推荐算法不适合评分预测。一般来说，top-n 推荐比评分预测更具有实际意义，实用性更高。

SLIM 推荐算法英文全名为 Sparse Linear Methods，sparse 是稀疏的意思，表示本算法处理的是稀疏矩阵，在用户物品行为矩阵中，用户没有行为的物品值都为 0，而在一个常见的系统中绝大部分用户没有对绝大部分物品有过行为，使得用户行为矩阵本身即是稀疏矩阵，而我们将通过机器学习得到的物品间关系矩阵也是稀疏的。Linear 是线性的意思，表示本算法的模型是一个线性方程。

2.2. 符号定义

在一个系统中，用户数为 m，物品数为 n。A 是一个 m×n 的用户物品行为矩阵，来自于数据集，A 有 m 行 n 列，第 i 行第 j 列的元素表示第 i 个用户是否对第 j 个物品有过行为，这个值表示为a_{ij}，如果数据集是隐反馈数据，则当用户 i 对物品 j 有过行为时，$a_{ij} = 1$，否则$a_{ij} = 0$，如果数据集是评分数据，则a_{ij}的值为用户的评分，若用户没有评分，则$a_{ij} = 0$，我会主要使用隐反馈数据。W 矩阵是要计算得到的物品之间关系矩阵 n×n，W 的第 i 行第 j 列的元素表示为w_{ij}，w_{ij}的

值表示物品 i 对物品 j 的推荐度。因此，物品 j 对用户 i 的推荐度$\tilde{a}_{ij} = a_i^T w_j$，其中$a_i^T$表示 A 矩阵第 i 行，$w_j$表示 W 矩阵第 j 列，这就相当于$\tilde{a}_{ij}$是用户 i 的每个有过行为的物品对物品 j 推荐度的总和。总的推荐矩阵$\tilde{A} = AW$，$\tilde{A}$是一个 m×n 矩阵，其中第 i 行第 j 列的元素是物品 j 对用户 i 的推荐度$\tilde{a}_{ij}$，$\tilde{a}_{ij}$的由来前面已经说明了。对$\tilde{A}$矩阵的第 i 行元素排序取值最高的 n 个物品推荐给用户 i，这即是产生 top-n 推荐的方法。

2.3. 模型介绍

现在已知 A 矩阵，要求 W 矩阵，SLIM 算法通过求解以下损失函数来得到 W 矩阵：

$$\min_W \left(\frac{1}{2}\|A - AW\|_F^2 + \frac{\beta}{2}\|W\|_F^2 + \lambda\|W\|_1\right) \quad (1)$$

其中 $$W \geq 0, diag(W) = 0$$

$\|W\|_1 = \sum_{i=1}^{n}\sum_{j=1}^{n}|w_{ij}|$是 l1 范数正则化项，使得解稀疏[4]。$\|.\|_F$是弗罗贝尼乌斯范数，用来防止过拟合。$\beta$和$\lambda$是正则化系数。这两个正则化项在一起组成了弹性网络回归[5]。式子$\frac{1}{2}\|A - AW\|_F^2$表示这个线性模型对训练集数据的拟合程度，越小则拟合的越好。$W \geq 0$约束条件表示我们只考虑物品之间推荐程度，而不考虑物品之间的不推荐程度。$diag(W) = 0$是指矩阵 W 的对角线元素全部为 0，W 的对角线元素是每个物品对自己的推荐值，假如不限定这个值为 0，每个物品只需要推荐自己就可以完全拟合训练集，这样得出的结果将是毫无意义的，我们重点需要得到不同物品之间的推荐关系，所以约束 W 矩阵对角线元素全为 0。

由于 W 的每一列是完全独立的，我们可以把 W 的每一列拿出来，从而通过求解以下损失函数可以得到 W 的第 col 列的值：

$$\min_{w_{col}} \left(\frac{1}{2}\|a_{col} - Aw_{col}\|_2^2 + \frac{\beta}{2}\|w_{col}\|_2^2 + \lambda\|w_{col}\|_1\right) \quad (2)$$

其中 $$w_{col} \geq 0, w_{col,col} = 0$$

$\|w_{col}\|_1 = \sum_{i=1}^{n}|w_{i,col}|$是 l1 正则化项（lasso），使得解稀疏[4]。$\|.\|_2$是 l2 正则化项（ridge regression），用来防止过拟合。这两个正则化项在一起组成了弹性网

络回归（elastic net）[5]。同样的$w_{col} \geq 0$只考虑推荐程度，不考虑不推荐程度；$w_{col,col} = 0$阻止物品自己推荐自己，防止产生无意义的解。方程(2)可以将 W 矩阵的每列单独求出，W 矩阵每列之间本来就互不影响，所以 SLIM 算法可以并行计算。

对于方程(2)的损失函数优化问题可以使用论文[3]中提出的坐标下降法进行求解，现在来概述一下这个坐标下降法。设 Y 是响应向量，X 是预测向量，线性回归模型定义为$\mathrm{E(Y|X = x)} = \beta_0 + x^T\beta$，总共有 N 个观测对$(x_i, y_i)$。现在要解决使如下的损失函数达到最小值的问题：

$$\min_{(\beta,\beta_0)\in R^{p+1}}[\frac{1}{2N}\sum_{i=1}^{N}(y_i - \beta_0 - x_i^T\beta)^2 + \lambda P_\alpha(\beta)] \tag{3}$$

其中

$$P_\alpha(\beta) = \sum_{j=1}^{p}[\frac{1}{2}(1-\alpha)\beta_j^2 + \alpha|\beta_j|]$$

是弹性网络回归，即 lasso 回归($\alpha = 1$)和 ridge regression 回归($\alpha = 0$)的组合。方程(2)中的矩阵 A 对应方程(3)中的 X，方程(2)中的a_{col}对应方程(3)中的 Y，方程(2)中的w_{col}对应方程(3)中的β，之前的 m 对应方程(3)中的 N，之前的 n 对应方程(3)中的 p。解决方程(3)需要迭代遍历更新每个β直到收敛，更新形式如下：

$$\tilde{\beta}_j = \frac{S(\frac{1}{N}\sum_{i=1}^{N} x_{ij}(y_i - \tilde{y}_i^{(j)}), \lambda\alpha)}{\frac{1}{N}\sum_{i=1}^{N} x_{ij}^2 + \lambda(1-\alpha)} \tag{4}$$

其中

$$\tilde{y}_i^{(j)} = \tilde{\beta}_0 + \sum\nolimits_{l\neq j} x_{il}\tilde{\beta}_l$$

$$S(z,\gamma) = sign(z)(|z| - \gamma)_+ = \begin{cases} z - \gamma, z > 0 \ and \ \gamma < |z| \\ z + \gamma, z < 0 \ and \ \gamma < |z| \\ 0, \gamma \geq |z| \end{cases}$$

第三章　推荐实验的设计

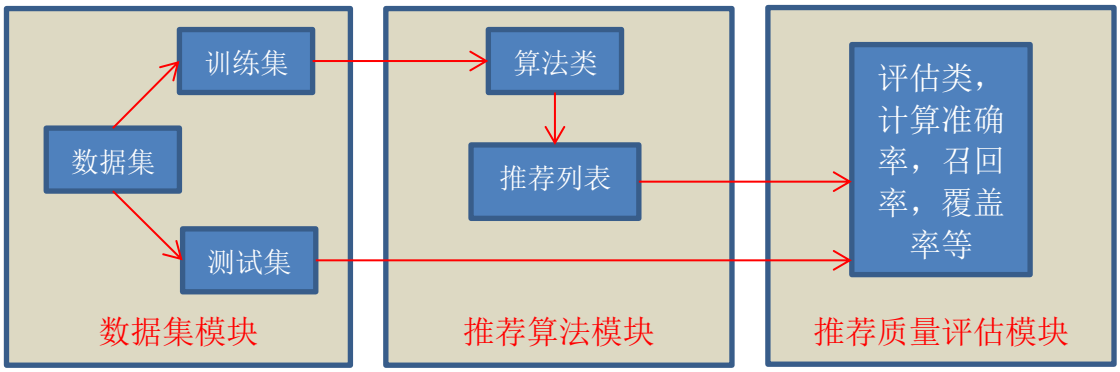
3.1. 推荐系统设计

在设计 SLIM 推荐算法前，首先需要设计一个简单且完整的推荐系统，在这个推荐系统之上再来设计并研究算法。

一般推荐系统分为 top-n 推荐和评分预测。top-n 推荐是要推荐算法对每个用户产生一个长度为 n 的推荐列表，一般评测指标为准确率，召回率等参数。评分预测要求推荐算法对每个用户没有评分的物品产生一个评分预测值，一般评测指标为 rmse 等。由于 SLIM 推荐算法主要专注于 top-n 推荐，我的推荐系统只会针对 top-n 推荐来进行设计。

我的推荐系统分为三大模块。第一个模块是数据模块，此模块负责数据集的输入，将数据集分为训练集与测试集，并记录数据的各项参数，方便其它模块调用。第二个模块是算法模块，算法模块对输入的数据在训练集上进行算法运算，得到每个用户的推荐列表。第三个模块是评估模块，评估模块对算法产生的推荐列表在测试集上面计算评估参数，产生评估结果。

这三大模块输入输出配合关系如下图所示：



对于数据集，我主要使用 movielens 的 ml-100k 和 ml-1m 数据集，数据集来源于网站(https://grouplens.org/datasets/movielens)，此数据集在推荐系统研究领域被广泛使用，稳定性好。ml-100k 数据集包括 100,000 条用户对电影的评分数据，有 1000 个用户，1700 个电影。ml-1m 数据集包括 1,000,000 条用户对电影的评分数据，有 6040 个用户，3900 个电影。其中每条用户对电影的评分数据包括 4 个值，其含义分别为用户 id，物品 id，评分，时间截。原始数据属于有上下文信息的显

反馈数据集，去掉多余的时间信息和评分信息变成无上下文信息的隐反馈数据。再将这 10 万条数据以 7:1 的比例随机分为训练集和测试集即可使用。

对于推荐算法，在 SLIM 推荐算法之外，我选取了 3 个比较典型的算法来与 SLIM 作比较。它们是基于用户的协同过滤算法，基于物品的协同过滤算法和隐语义模型算法。其中前两个算法在书籍[1]中第 2.4 节有详细的介绍，第三个算法在书籍[1]中第 2.5 节有详细的介绍。基于用户的协同过滤算法和基于物品的协同过滤算法是典型的基于邻域的推荐算法，隐语义模型算法是典型的基于矩阵分解的推荐算法。

对于结果评估，主要考虑四大参数：准确率，召回率，覆盖率和流行度。我们先设 U 为所有用户集合，I 为所有物品集合，R(u)为推荐算法在训练集上对用户 u 产生的推荐物品集合，T(u)为用户 u 在测试集上所有有过行为的物品集合，ip(i)为物品 i 在训练集里面出现的次数。

准确率反映了推荐算法根据训练集产生的推荐物品列表中在测试集被用户青睐的物品所占比例。那么我们的准确率定义为：

$$precision = \frac{\sum_{u \in U} |R(u) \cap T(u)|}{\sum_{u \in U} |R(u)|} \tag{5}$$

由于 top-n 推荐系统的究极目标就是推测并选出用户最感兴趣的物品推荐给用户，所以准确率这个推荐系统评估指标往往是最直观，被使用最广泛，也是推荐系统选取时最关心的指标。

召回率反映了测试集用户所有有过行为的物品中存在于推荐列表中的物品所占比例。我们的召回率定义为：

$$recall = \frac{\sum_{u \in U} |R(u) \cap T(u)|}{\sum_{u \in U} |T(u)|} \tag{6}$$

召回率功能和准确率大体相同，也是直观的反映了推荐算法有多好，只是从另一个角度来度量推荐算法的好坏程度。在某些极端的情况下，推荐算法对每个用户推荐的物品数量和测试集中平均每个用户有行为的物品数量的比例严重失调。当推荐算法对每个用户推荐的物品数量远多于测试集中平均每个用户有行为的物品数量时，准确率指标总是会非常小，这个时候召回率更能较好的反映推荐算法的质量；当推荐算法对每个用户推荐的物品数量远少于测试集中平均每个用户有行为的物品数量时，召回率指标总是十分的小，这个时候准确率能更好的反映推荐算法的质量。所以我们在对推荐算法评估时同时使用准确率和召回率这两个指

标，配合起来一起看能更好的反映算法实际的推荐质量。

除了能直观反映推荐算法结果质量的准确率和召回率指标外，覆盖率也是内容提供商一般特别关心的一个十分重要的指标。覆盖率用来描述一个推荐系统对物品长尾的发掘能力。我们的覆盖率定义为：

$$Coverage = \frac{|\cup_{u \in U} R(u)|}{|I|} \quad (7)$$

如果推荐算法的覆盖率十分的低，即使准确率和召回率还可以，这个推荐算法也是缺点很明显的。这种情况的一种典型的推荐算法是把热门物品排行榜的前 n 个物品直接推荐给所有用户，这种情况首先会导致推荐算法不具备个性化推荐的功能，毕竟每个人的推荐列表几乎都是一样的，而且还会导致热门的物品更加热门，冷门的物品无人问津。一个好的推荐系统除了有一个很好的准确率和召回率，同样也需要一个比较不错的覆盖率。

然而这个覆盖率定义实在是很粗略，就算是计算出的覆盖率为 100%的系统也会有无穷多种不同的流行度分布情况。想要更细致的描述一个推荐系统对物品长尾的发掘能力，我们可以考虑统计每个物品分别被推荐了多少次，如果每个物品分别被推荐给用户的数量大体相似，我们就可以得出这个推荐系统对物品长尾的发掘能力很不错的结论。

由于覆盖率不够完美的特点，我们需要一个额外的流行度指标来配合覆盖率衡量这些推荐算法。我们的流行度定义为：

$$Popularity = \frac{\sum_{u \in U} \sum_{i \in R(u)} \log(1 + ip(i))}{\sum_{u \in U} |R(u)|} \quad (8)$$

流行度运算时取对数是为了应对物品流行度的长尾分布，使得流行度的平均值更加稳定。一般来讲，流行度越低，推荐算法更能发掘出冷门物品的价值，可以看出推荐算法在这方面的能力越好。

3.2. 推荐算法设计

3.2.1.SLIM 简单算法

结合方程(2),(3)，求解 W 的第 col 列w_{col}变为求解如下损失函数：

$$\min_{w_{col}} [\frac{1}{2m} \sum_{i=1}^{m} \left(A_{i,col} - A_i^T w_{col}\right)^2 + \lambda \sum_{j=1}^{n} [\frac{1}{2}(1-\alpha) w_{j,col}^2 + \alpha |w_{j,col}|]] \quad (9)$$

其中 $$w_{col} \geq 0, w_{col,col} = 0$$

结合方程(4)，坐标下降法求解方程(9)的更新形式为：

$$\widetilde{w}_{j,col} = \frac{S'(\frac{1}{m}\sum_{i=1}^{m} a_{ij}(a_{i,col} - \tilde{a}_{i,col}^{(j)}), \lambda\alpha)}{\frac{1}{m}\sum_{i=1}^{m} a_{ij}^2 + \lambda(1-\alpha)} \tag{10}$$

其中 $$\tilde{a}_{i,col}^{(j)} = \sum_{l \neq j} a_{il}\, \widetilde{w}_{l,col}$$

$$S'(z,\gamma) = sign(z)(|z| - \gamma)_+ = \begin{cases} z-\gamma, z > 0 \ and \ \gamma < |z| \\ 0, z < 0 \ and \ \gamma < |z| \\ 0, \gamma \geq |z| \end{cases}$$

注意软阈值函数$S(z,\gamma)$变为了$S'(z,\gamma)$，其唯一的区别在于当$z < 0 \ and \ \gamma < |z|$时，$S(z,\gamma) = z + \gamma < 0$而$S'(z,\gamma) = 0$。这个变化是为了满足$W \geq 0$的限定条件，防止产生某个 w 小于零的情况。

根据式(9)和(10)就能想到一个最简单最直接的算法来实现 SLIM 推荐算法对 W 矩阵的计算：

算法 1：

从数据集已知矩阵 A，m 行，n 列，取$\lambda = 0.02, \alpha = 0.5$，最小变化阈值 tol=0.0001

初始化 W 矩阵为一个 n×n 全零矩阵

for col 从 1 到 n：（求每个w_{col}）

do:

for j 从 1 到 n:

if j == col:

continue

根据方程(10)计算得到新的$w_{j,col}$

If $\left|\text{新的}w_{j,col} - \text{旧的}w_{j,col}\right| > tol$:

用新的$w_{j,col}$取代旧的$w_{j,col}$

while (本次循环没有改变任何 w 的值) （即直到收敛）

算法 1 结束。

由于软阈值函数$S(z,\gamma)$的存在，且 W 矩阵的每一个值本来都是是零，很多值在经过软阈值函数运算后仍然保持零。这样遍历一次的时间复杂度是 O(m)。在 w 确实变化了的情况下，这一步时间复杂度为 O(2m)。所以对 n 个 w 遍历更新一次

时间复杂度为 O(mn)。

3.2.2.协同更新

算法 1 已经可以求出我们需要的 W 矩阵，已经实现 SLIM 算法。但是由于算法 1 实在是太慢，实用性确实是不怎么高，仍需改进。

论文[3]第七页 2.2 节提出了协同更新（covariance update）的方法来遍历求得 W 每个元素的值，此方法要比算法 1 快很多。我们将方程(10)的一部分提取出来可变化为如下形式：

$$\frac{1}{m}\sum_{i=1}^{m} a_{ij}(a_{i,col} - \tilde{a}_{i,col}^{(j)}) = \frac{1}{m}(cov(j,col) + cov(j,j) \times w_{j,col} - gc(j)) \quad (11)$$

其中

$$cov(j_1, j_2) = a_{j_1}^T a_{j_2}$$

$$gc(k) = \sum_{j=1}^{n} cov(k,j)\, \widetilde{w}_{j,col}$$

根据方程(10)和方程(11)，坐标下降法求解方程(9)的协同更新形式为：

$$\widetilde{w}_{j,col} = \frac{S'(cov(j,col) + cov(j,j) \times w_{j,col} - gc(j), \lambda\alpha m)}{cov(j,j) + \lambda(1-\alpha)m} \quad (12)$$

其中

$$cov(j_1, j_2) = a_{j_1}^T a_{j_2}; gc(k) = \sum_{j=1}^{n} cov(k,j)\, \widetilde{w}_{j,col}$$

$$S'(z,\gamma) = sign(z)(|z| - \gamma)_+ = \begin{cases} z - \gamma, z > 0 \; and \; \gamma < |z| \\ 0, z < 0 \; and \; \gamma < |z| \\ 0, \gamma \geq |z| \end{cases}$$

引入协同更新后，SLIM 算法过程如下：

算法 2：

从数据集已知矩阵 A，m 行，n 列，取$\lambda = 0.02, \alpha = 0.5$，最小变化阈值 tol=0.0001

初始化 W 矩阵为一个 n×n 全零矩阵

初始化容器 cov 并预计算出 cov 每对参数取值的结果

for col 从 1 到 n：（求每个w_{col}）

初始化容器 gc 使得对于所有的$k \in [1,n]$，$gc(k) = 0$

do:

for j 从 1 到 n:

if j == col:

continue

根据方程(12)计算得到新的$w_{j,col}$

If $|$新的$w_{j,col}$ − 旧的$w_{j,col}| > tol$:

用新的$w_{j,col}$取代旧的$w_{j,col}$

for k 从 1 到 n:

$gc(k) += cov(k,j) \times$ (新的$w_{j,col}$ − 旧的$w_{j,col}$)

while (本次循环没有改变任何 w 的值) （即直到收敛）

算法 2 结束。

每次计算一个$cov(j_1, j_2)$的值，需要时间复杂度 O(m)，总共有 0.5(n-1)n 种不同的j_1, j_2取值情况，计算完所有的 cov 需要的时间复杂度为$O(mn^2)$，且根据方程(12)里面有两项$cov(j, col)$和$cov(j, j)$，j 从 1 到 n，col 也会从 1 到 n，所以可以得出 cov 函数的参数所有的不同的j_1, j_2取值都需要计算出来，所以我们可以提前计算出所有 cov 的值，或者每算出来一个就存一个值。每当有一个 w 值发生改变，我们需要更新 gc 的值，全部更新完时间复杂度为 O(n)。所以假设在一次遍历的过程中有 h 个 w 的值会发生改变，完全遍历一趟需要时间复杂度 O(hn)。

由于完全遍历学习一次的时间复杂度大幅降低且实际情况下一般要学习很多次才能收敛，协同更新的加入使得算法 2 理论上极大的加速了原来的算法 1。

3.2.3.活跃集更新

另外还有一种更加快速的更新方式。我们把 W 的第 col 列w_{col}中所有不为零的项的集合称为活跃集（active set）。在第一次遍历更新一次每一个w_{col}中的元素后，我们接下来只更新活跃集中的元素，直到收敛。此时再次遍历更新一次每一个w_{col}中的元素，若活跃集没有任何变化，算法就可以到此为止了，若活跃集变了，我们再次只更新活跃集中的元素，直到收敛，如此循环即可。

引入活跃集更新后，SLIM 算法过程如下：

算法 3：

从数据集已知矩阵 A，m 行，n 列，取$\lambda = 0.02, \alpha = 0.5$，最小变化阈值 tol=0.0001

初始化 W 矩阵为一个 n×n 全零矩阵

初始化容器 cov 并预计算出 cov 每对参数取值的结果

for col 从 1 到 n：（求每个w_{col}）

初始化容器 gc 使得对于所有的$k \in [1,n]$，$gc(k) = 0$

初始化 mode=0（mode 为 0 的时候遍历所有 w，mode 为 1 的时候只遍历活跃集）

while true:

for j 从 1 到 n:

if (j == col)或者(mode==1 且$w_{j,col}$==0):

continue

根据方程(12)计算得到新的$w_{j,col}$

If $\left|\text{新的}w_{j,col} - \text{旧的}w_{j,col}\right| > tol$:

用新的$w_{j,col}$取代旧的$w_{j,col}$

for k 从 1 到 n:

$gc(k) += cov(k,j) \times (\text{新的}w_{j,col} - \text{旧的}w_{j,col})$

If (本次循环没有改变任何 w 的值) （即收敛了）

If mode == 0:

break

if mode == 1:

mode = 0

elif mode == 0:（遍历了所有 w 且有 w 发生了改变）

mode = 1

算法 3 结束。

到此，SLIM 算法过程已经进行了相当大的优化了，但是仍然存在一个缺陷。

3.2.4.正则项系数的选取

现在我们来讨论一下取值$\lambda = 0.02, \alpha = 0.5$的问题。$\alpha$是 l1(lasso)在 l1 正则项和 l2(ridge regression)正则项之间的所占比例，取α为 0.5 表示 lasso 和 ridge regression 各占一半，一般问题不大，基本适用于所有情况。但是λ的取值就很有争议了。我这里默认取了 0.02 是我通过代码运行后发现这个值对算法的结果影响

比较适中还能接受。但是缺陷仍然很严重。

对于单次坐标下降法，λ的取值可以通过反复实验大致得到一个较好的取值，然而 SLIM 推荐算法需要进行多次坐标下降法，次数取决于系统物品的个数 n，这 n 次坐标下降法不可能得到单一λ的取值使得每次计算过程较好。理论上来讲，λ越大，坐标下降法的解 W 矩阵会更稀疏，算法所花的时间越短，但是得到的 W 信息就越少，最后产生推荐列表计算得到的准确率，召回率，覆盖率都会越低。反过来λ越小，坐标下降法的解 W 矩阵会更稠密，算法所花时间会越长，得到的 W 信息会越多，最后产生推荐列表计算得到的准确率，召回率，覆盖率都会相对来说更高一点。

接下来我会分析产生λ取值如此困难的原因，并采取适当的解决方案。在对于 W 的第 col 列的这次坐标下降法的过程中，若λ满足$\forall j \in [1, n], cov(j, col) \leq \lambda \alpha m$，根据方程(12)，此时坐标下降法第一次遍历的解$\tilde{w}_{col}$的每一个元素一定会全部为零。我们设

$$\lambda_{max} = \frac{\max\limits_{j \in [1,n]} cov(j, col)}{\alpha m} \tag{13}$$

为使整个解全为零的λ最小值，即如果$\lambda < \lambda_{max}$，坐标下降法的解w_{col}中至少有一个元素不为零，如果$\lambda \geq \lambda_{max}$，坐标下降法的解$w_{col}$中的每一个元素都为零。所以这可以解释为什么确定单一λ取值如此困难。若λ随意取一个不大不小的值，坐标下降法的解 W 矩阵中可能会有很多列完全为零，这些列所代表的物品没有学到任何信息，从而影响了机器学习的质量，导致我们的推荐算法评估参数偏低。若λ取一个较小的值，仍然不能保证坐标下降法的解 W 中没有全为零的列，并且会导致一些列十分稠密，造成算法过慢的问题。

所以，为了解决以上问题，我们可以在对每列执行坐标下降法之前首先根据方程(13)计算出λ_{max}的值，然后对于这列我们取$\lambda = t\lambda_{max}$其中 0<t<1。这样确认的λ的值既不会拖慢算法的速度，又不会丢失算法的推荐质量。

由此，优化λ后的 SLIM 算法过程如下：

算法 4：

从数据集已知矩阵 A，m 行，n 列，取$t = 0.02, \alpha = 0.5$，最小变化阈值 tol=0.0001

初始化 W 矩阵为一个 n×n 全零矩阵

初始化容器 cov 并预计算出 cov 每对参数取值的结果

for col 从 1 到 n：（求每个w_{col}）

根据方程(13)得到λ_{max}

if $\lambda_{max} == 0$:

continue

$\lambda = t\lambda_{max}$

初始化容器 gc 使得对于所有的$k \in [1,n]$，$gc(k) = 0$

初始化 mode=0（mode 为 0 的时候遍历所有 w，mode 为 1 的时候只遍历活跃集）

while true:

for j 从 1 到 n:

if (j == col)或者(mode==1 且$w_{j,col}$==0):

continue

根据方程(12)计算得到新的$w_{j,col}$

If $\left|\text{新的}w_{j,col} - \text{旧的}w_{j,col}\right| > tol$:

用新的$w_{j,col}$取代旧的$w_{j,col}$

for k 从 1 到 n:

$gc(k) += cov(k,j) \times (\text{新的}w_{j,col} - \text{旧的}w_{j,col})$

If (本次循环没有改变任何 w 的值) （即收敛了）

If mode == 0:

break

if mode == 1:

mode = 0

elif mode == 0:（遍历了所有 w 且有 w 发生了改变）

mode = 1

算法 4 结束。

第四章　推荐实验的实现

4. 1. 实验环境

我的实验环境见以下表格：

参数名	值
处理器（CPU）	i7 4720HQ
内存（RAM）	16GB ddr3
硬盘（ROM）	1TB 7200 HDD
系统（System）	Windows10
程序语言（Python）	Python3.6 64 位
编程软件（IDE）	VS2017 和 PyCharm

其中处理器（CPU），内存（RAM），硬盘（ROM）是我的笔记本配置情况。我对于本论文的所有工作都是在我的笔记本上完成的。给出这些参数是为了给读者一个参考，方便其它人做类似的实验时与我的结果进行一个横向的对比，以免造成硬件参数不同导致的实验结果有差异的问题。

我的系统是 win10，win10 也是目前微软推出的最新版本的 windows 操作系统。win10 对于最新的主流软件以及最新的主流软件对于 win10 都有完美的支持。实验过程保证 win10 自动更新是关闭的状态即可。使用这个系统可以保证系统不会对我的实验过程造成意想不到的干扰。

我使用的编程语言是 Python3。使用 Python 的 64 位版本是为了应对数据集过大可能导致程序需要使用超过 2gb 的内存，如果使用 32 位版本的 python，程序运行时最大只能使用大约 2gb 的内存，容易爆内存，引起程序运行缓慢，对最后的测试结果产生有害的影响。Python 是一门上手容易，语法精简，功能强大的脚本语言，非常适合于科学研究，在网络爬虫，大数据，人工智能等领域被广泛使用。Python 是一种既可以面向过程又可以面向对象的解释型计算机程序设计语言，最初由荷兰人 Guido van Rossum 在 1989 年发明，至今已经经历了大量的优化与改进，最新版的 Python3 甚至在很多语法上都不支持原来的 Python2。Python 的库

是十分丰富与强大，可以应对绝大部分场景需求，常见常用的基本函数或基本算法几乎应有尽有。Python 语言的强大之处还表现在可以轻松的和其它语言制作的模块连接在一起，尤其是 C/C++语言。这在对性能要求很高的模块上面有极大的意义与优势。Python 本身是一门解释型的脚本语言，每句话要先解释翻译成机器语言再执行，这导致了 Python 这门语言本身的代码运行速度是一个很大的瓶颈。然而由于 Python 的这种方便与其他语言连接的优势，我们可以把对性能要求较高的模块直接交给 C/C++这种代码运行效率极高的语言去完成，然后在 Python 里面封装为 Python 可调用的扩展类库，使用时直接调用即可。

VS2017 和 PyCharm 我同时配合在使用。PyCharm 是 python 相当好的 IDE，功能强大，例如代码智能提示功能，自动补全代码功能以及版本控制等功能都相当不错，使用 PyCharm 写 python 代码十分舒适高效。本来只需要一个 PyCharm 就够了的，结果我在使用 python 的库 Cython 和 c 语言代码交互的时候总是报错，提示没有 c 语言编译器，但是直接使用 VS2017 就没有这个问题。所以我把这两个编程软件配合起来一起使用，VS2017 重在处理 python 和 c 语言的交互，而代码主要在 PyCharm 里面写。这两个编译器都用的社区版(community edition)，社区版完全免费，免去破解的麻烦操作，并且下载方便，直接从官方网站上面下载，不需要通过第三方资源网站下载，免去下载到垃圾数据或病毒的问题。

4.2. 推荐系统实现

我的推荐系统的实现是基于本论文 3.1 节推荐系统设计来进行的。根据 3.1 节的推荐系统的设计，推荐系统主要分为三大模块：数据处理模块，推荐算法模块和结果评估模块。依次实现这三大模块的全部功能即可达到目标。

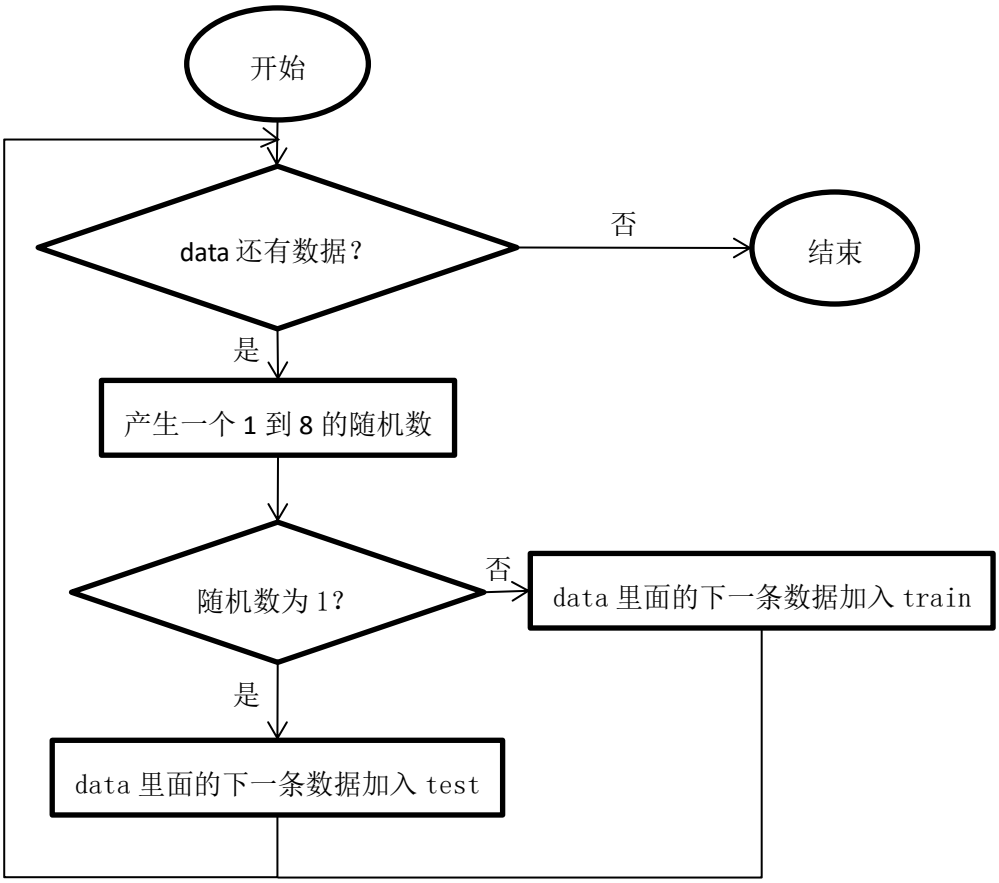
我采用面向对象的编程思想，每个模块相当于一个 python 类，每种需要的数据作为成员变量放在类里面，每个过程作为类一个方法。

数据类成员方法如下表所示：

Data 类		
	名称	含义
	data	原始数据列表，列表里面每个元素包括两个整数：用户 id 和物品 id。

成员变量	num_user	用户数量（用户 id 范围为 0 到 num_user-1）。
	num_item	物品数量（物品 id 范围为 0 到 num_item-1）。
	train	训练集列表，每个元素形式与 data 相同。
	test	测试集列表，每个元素形式与 data 相同。
方法	__init__()	构造器，录入 data 数据，得到 num_user 和 num_item 的值。
	split_data()	将数据随机分成 8 份，1 份作为测试集，7 份作为训练集

其中 split_data 方法流程图如下：



推荐算法类成员方法如下表所示：

Algorithm 类		
	名称	含义
成员变量	data	Data 类的对象，所有数据来自于此。
	recommendation	经过推荐算法产生的每个用户的推荐列表。
	__init__(data)	构造器，得到 Data 类的数据对象，赋引

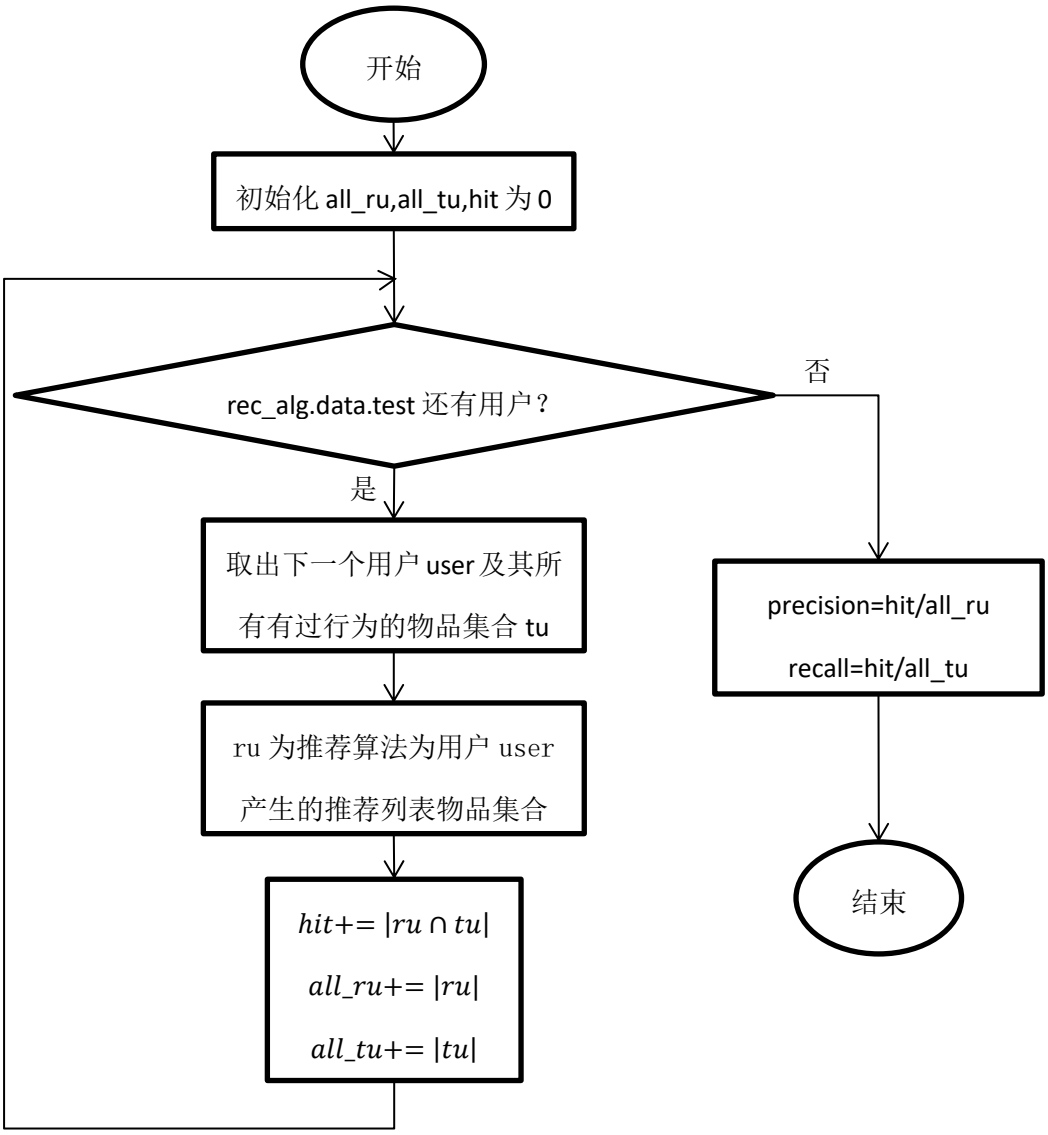
方法		用给成员变量 data。
	compute_recommendation(args)	通过推荐算法，根据推荐算法参数 args，结合数据对象 data 的训练集 train 计算得到每个用户的推荐列表 recommendation。

其中 compute_recommendation(args)方法根据算法不同而有不同的实现，其参数 args 一般包括多个参数，分别对应此推荐算法所需要的关键系数，SLIM 推荐算法具体实现见下一节 4.3.推荐算法实现。这里就不具体讨论了。

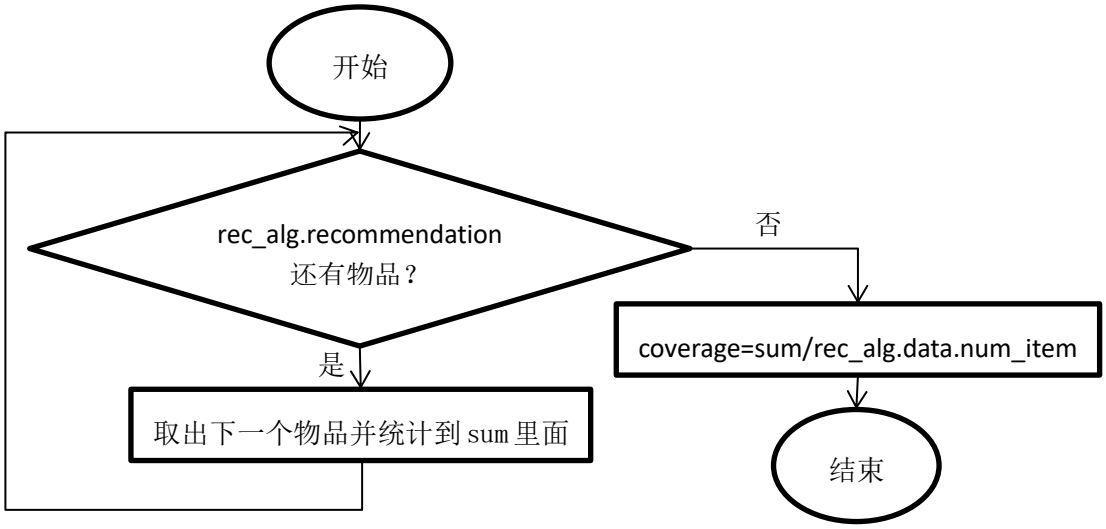
评估类成员方法如下表所示：

Evaluation 类		
	名称	含义
成员变量	rec_alg	Algorithm 类的对象，里面包括推荐列表和原数据集。
	precision	计算的准确率评估参数。
	recall	计算的召回率评估参数。
	coverage	计算的覆盖率评估参数。
	popularity	计算的流行度评估参数。
方法	__init__(rec_alg)	构造器，得到 Algorithm 类的对象，赋引用给成员变量 rec_alg。
	evaluate()	根据成员变量 rec_alg 算法对象及其里面的 data 数据对象计算四大评估参数：准确率，召回率，覆盖率，流行度。（即此方法会依次调用下面三个方法）
	__precision_recall()	计算准确率和召回率。
	__coverage()	计算覆盖率。
	__popularity()	计算流行度。

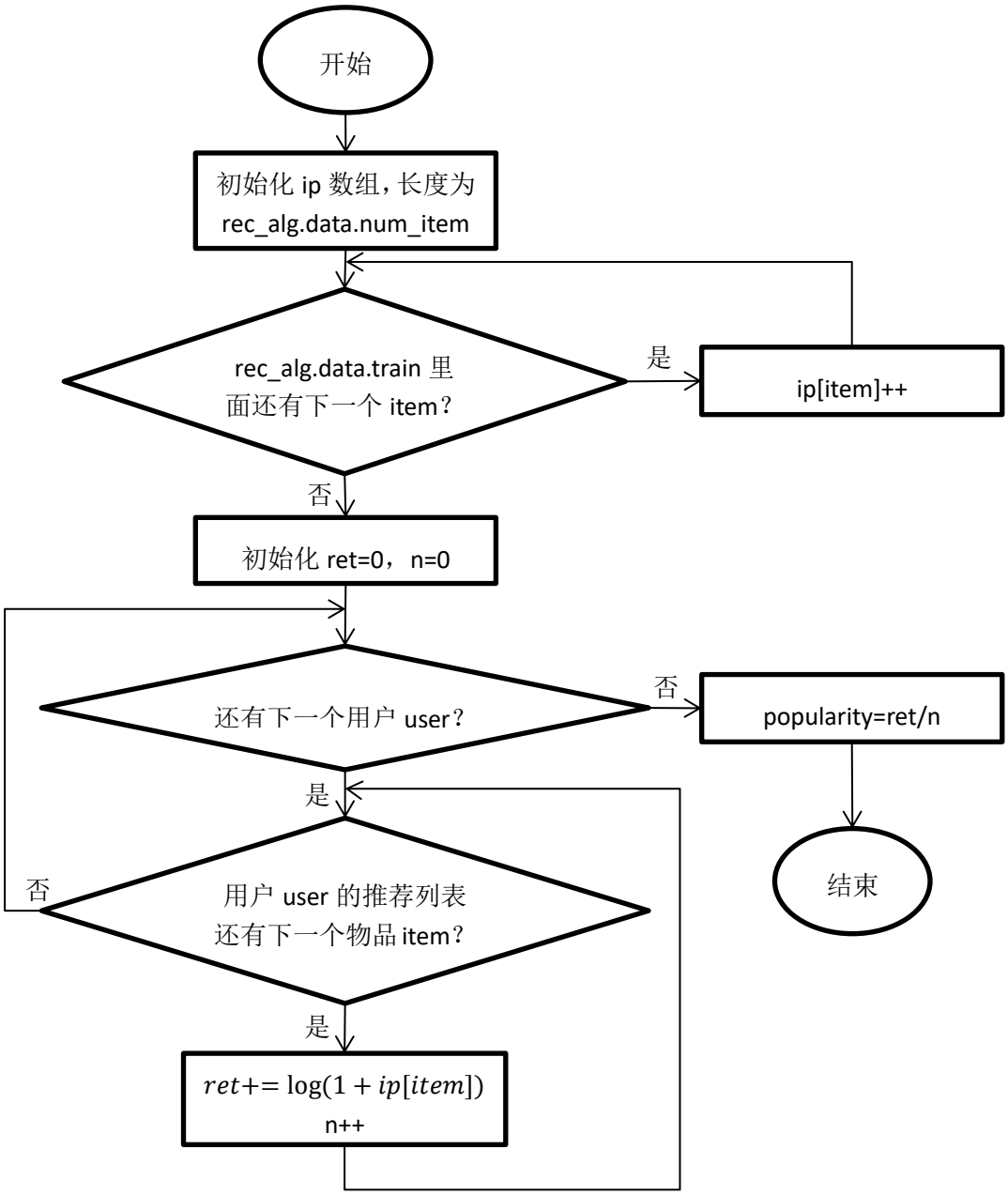
其中__precision_recall()方法基于公式(5)和公式(6)实现，其流程图如下:



其中__coverage()方法基于公式(7)实现，其流程图如下：



其中__popularity()方法基于公式(8)实现，其流程图如下：



到此，一个简单的推荐系统就完成了。这个推荐系统适合于进行推荐算法离线实验的研究，还不适合放到软件直接使用，但是对于我的研究内容已经完全足够了。

4. 3. 推荐算法实现

4.3.1.用到的 python 库

根据 3.2 节推荐算法设计的分析，要实现 SLIM 推荐算法，主要需要矩阵的处理，并行计算的实现。还有一个很重要的是和 C/C++语言的交互。这些我来逐个

分析。

首先关于矩阵处理，我使用了 numpy 库。numpy 库是 python 的一个开源的第三方库，主要用于数组，矩阵的处理与运算。用 numpy 表示的数组要比 python 自带的 list 列表高效许多，同时 numpy 二维数组也可以用来表示矩阵，包括一些矩阵相关的运算方法，都十分高效。numpy 专为需要高效严格的数字处理而存在，在科学研究领域都十分广泛，可以替代很多本来需要 C++，Fortran 或 matlab 等来完成的工作。numpy 当然是我最理想的选择。用户行为矩阵 A 和物品之间关系矩阵 W 的表示完全可以使用 numpy 二维数组。对 cov 函数的值的求解操作以及 AW 矩阵之间相乘的操作都可以方便的通过 numpy 自带的矩阵相乘 dot 方法来完成。

然后关于并行计算，我使用了 concurrent.futures 库里面的 ProcessPoolExecutor 类来进行并行计算。Python 语言由于全局解释器锁 GIL(global interpreter lock)的存在，Python 的线程无法实现真正的并行计算，无法将不同线程的任务分配到 cpu 不同的核心去运行，这是 Python 语言本身的一个的缺陷。既然多线程不能进行并行运算，那么自然就会想到我们可以使用多进程来进行并行运算。Python 的 concurrent.futures 库里面的 ProcessPoolExecutor 类正是利用多进程来实现 Python 并行运行的。

最后关于与 C/C++语言的交互模块，我是用的是 Cython。由于 Python 语言本身代码运行效率低下，在运行推荐算法时，如果直接使用 Python 语言编码，每次运行程序将会浪费大量的时间执行低效的 Python 代码。由于考虑到代码执行效率问题，我一开始本来准备直接使用 C/C++语言来实现推荐算法，后来看到 Python 里面的 Cython 库的存在后，立刻投奔 Python 语言。Cython 把为 Python 写 C/C++语言扩展库的难度直接降低到和写 Python 语言本身差不多，通俗点解释就是 Cython 可以直接把 Python 代码翻译成 C/C++语言的代码然后编译后即可使用 Python 直接调用。由于 Python 最常见的版本 CPython 本身是由 C 语言实现，Python 里面很多东西都可以在 C 语言里面找到与之相对应的东西，Cython 正是利用了这种特性而实现。同时不止于此，Cython 代码与 Python 不完全相同，可以将 Cython 理解为一种全新的语言。Cython 代码一般写在后缀名为 pyx 的源代码文件中，与 Python 源代码的 py 后缀名以示区别。Cython 几可以直接使用 Python 的任何性质或语法，并且可以调用大多数 Python 库，但同时又可以直接使用 C/C++的特性。在写 Cython 的时候可以就当成写 Python 代码，只是在需要使用 C/C++特性的时

候使用 cdef 等关键字申明即可。由于 Cython 的存在，我们可以将推荐系统的最耗时的推荐算法部分翻译成 C/C++语言并编译后再调用，其它不那么耗时的数据处理部分，结果评估部分仍然使用 Python 处理。

至此已经介绍了我使用的最重要的三大 Python 库。还有一些不那么重要的 Python 库包括：random 库实现随机数的生成，time 库实现时间的统计，math 库实现一些简单的数学运算，pandas 库实现结果的表格化展示。由此可以看到 Python 这门语言加上一些强大的库可以轻松完成很多事情，这也是我选择 Python 来研究推荐算法的原因。有了这些准备工作，我们可以开始讨论我们的核心问题：SLIM 推荐算法的实现。

4.3.2.SLIM 推荐算法的实现

我的 SLIM 推荐算法主要是依据第 3.2 节推荐算法设计里面的算法 4 来实现的。根据前面的分析，SLIM 推荐算法完全可以并行计算解出 W 每列的值，所以这里对算法 4 做一点修改，为了充分利用现代计算机多核心多线程的优势，极大加快算法进行速度，修改后的过程如下：

算法 5：

从数据集已知矩阵 A，m 行，n 列，取$t = 0.02, \alpha = 0.5$，最小变化阈值 tol=0.0001

初始化 W 矩阵为一个 n×n 全零矩阵

初始化容器 cov 并结合矩阵 A 预计算出 cov 每对参数取值的结果

取 col 的值从 1 到 n 并行调用过程 f(col, W, cov, m, n, t, α, tol)（求每个w_{col}）

过程 f(col, W, cov, m, n, t, α, tol):

根据方程(13)得到λ_{max}

if $\lambda_{max} == 0$:

continue

$\lambda = t\lambda_{max}$

初始化容器 gc 使得对于所有的$k \in [1, n]$，$gc(k) = 0$

初始化 mode=0（mode 为 0 的时候遍历所有 w，mode 为 1 的时候只遍历活跃集）

while true:

for j 从 1 到 n:

if (j == col)或者(mode==1 且$w_{j,col}$==0):

continue

根据方程(12)计算得到新的$w_{j,col}$

If $\left|\text{新的}w_{j,col} - \text{旧的}w_{j,col}\right| > tol$:

用新的$w_{j,col}$取代旧的$w_{j,col}$

for k 从 1 到 n:

$gc(k) += cov(k,j) \times (\text{新的}w_{j,col} - \text{旧的}w_{j,col})$

If (本次循环没有改变任何 w 的值) （即收敛了）

If mode == 0:

break

if mode == 1:

mode = 0

elif mode == 0:（遍历了所有 w 且有 w 发生了改变）

mode = 1

算法 5 结束。

在算法 5 中，A 矩阵和 W 矩阵都是用 numpy.ndarray 数组表示，并行调用过程是由 ProcessPoolExecutor 方法 map 实现，整个过程 f(col, W, cov, m, n, t, α, tol)是由 Cython 实现。

经过算法 5，我们已经得到物品之间推荐关系矩阵 W，接下来我们已经有了足够的信息可以对每个用户产生推荐列表了。SLIM 推荐算法对所有用户产生推荐列表的过程如下：

SLIM 推荐算法对所有用户产生推荐列表的过程：

已知用户物品行为矩阵 A（m 行，n 列）和物品之间推荐关系矩阵 W（n 行，n 列）；

计算 A 矩阵和 W 矩阵相乘的结果为矩阵 B（m 行，n 列）；

对矩阵 B 的每个元素额外记下当前所属列作为物品 id，每个元素本身的值作

为物品对本行代表的用户推荐度；

对矩阵 B 的每一行元素根据推荐度从大到小排序；

留下矩阵 B 的前 n 列，舍去其它元素；

现在矩阵 B 就是最终产生的每个用户的推荐列表。（第 u 行的 n 个元素是给 id 为 u 的用户推荐的 n 个物品）

SLIM 推荐算法对所有用户产生推荐列表的过程结束。

4.4. 运行结果

我首先测试 SLIM 推荐算法本身的不同实现方式之间的差异。选取数据集为 movielens 的 ml-100k 数据集，分别对 SLIM 推荐算法的四种不同的实现运行三次后，取三次结果的平均值，得到一个 SLIM 推荐算法自身不同实现的运行结果的对比。此对比结果如下表格 1 所示：

表格 1 SLIM 算法不同实现之间的推荐质量对比

SLIM 算法不同实现	准确率	召回率	覆盖率	流行度	耗时
a	26.766%	19.979%	22.771%	5.442087	20.681s
b	26.766%	19.979%	22.771%	5.442087	20.513s
c	27.588%	20.593	37.455%	5.331005	63.261s
d	27.620%	20.617%	37.455%	5.330639	53.722s

在表格 1 中，所有的算法都是 SLIM 推荐算法，每个算法都使用了协同更新（covariance update），参数α（l1(lasso)在 l1 正则项和 l2(ridge regression)正则项之间的所占比例）取 0.5，max_iter（最大学习迭代次数）取 1000，tol（每次更新的最小变化阈值）取 0.0001。其中 a 和 b 都使用了单一λ，λ（弹性网络正则项系数）取 0.02，b 额外使用了活跃集更新。C 和 d 都是用了λ比例 t 来决定每列进行坐标下降法时λ的值，t（λ的取值相对于λ_{max}的倍数）取 0.02，d 额外使用了活跃集更新。

观察表格 1 的数据。对比 a 和 c 的结果或者对比 b 和 d 的结果，可以清晰的看到使用了λ比例 t 来决定每列进行坐标下降法时λ的值的算法要优于使用了单一λ的算法，一方面准确率和召回率都有大约 1 个百分点的提升，另一方面覆盖率

提升巨大，几乎提升了 15 个百分点，流行度也更低，这说明了使用λ比例 t 来决定每列进行坐标下降法时λ的值能更好的应对长尾分布，这是由于这样的λ取值保证了在执行坐标下降法后 W 的每列都能学到信息，使得更多相对更冷门的物品被发掘出来，从而导致了覆盖率的巨大提升。再来对比 a 和 b 的结果或者对比 c 和 d 的结果可以发现活跃集更新这种方式可以在几乎不影响算法任何性能的情况下进一步减少了算法的时间消耗，a 和 b 的数据中活跃集更新使得算法在准确率召回率覆盖率流行度四大参数完全没变的情况下略微节省了一点点的时间，c 和 d 的数据中可以明显的看到了活跃集更新节省了大量的时间，而且算法质量评估参数甚至还有略微的提升。

根据表 1 的数据我们可以得出结论：活跃集更新这种方法可以加速算法的进行并且基本上不影响算法的结果，这是一种有利而无害的更新方式；使用λ比例 t 来决定每列进行坐标下降法时λ的值可以明显提升算法产生的推荐结果的质量，尤其是覆盖率会大幅上升，然而算法的时间消耗也会大幅上升。

接下来将 SLIM 推荐算法与其它算法放到一起进行对比实验。选取数据集为 movielens 的 ml-100k 数据集，每种算法运行三次后，对每项参数取平均值，得到一个对比结果。我的运行结果展示于下面的表格 2 中：

表格 2　SLIM 算法与其它算法在 ml-100k 数据集下的推荐质量对比

推荐算法	准确率	召回率	覆盖率	流行度	耗时
UserCF	23.985%	17.649%	14.269%	5.595094	4.388s
ItemCF	22.630%	17.044%	15.815%	5.553404	3.283s
LFM	26.759%	20.098%	19.620%	5.450575	38.012s
SLIM	27.350%	20.158%	36.801%	5.330536	57.234s

在表格 2 中，所有算法的参数 N（每个用户最多推荐物品数量）都为 10。UserCF 是基于用户的协同过滤算法，用户相似度用余弦相似度度量，算法具体过程在书籍[1]的 2.4.1 节有详细的介绍，其参数 K（推荐时选择与物品最相似的物品个数）取 80。ItemCF 是基于物品的协同过滤算法，物品相似度用余弦相似度度量，算法具体过程在书籍[1]的 2.4.2 节有详细的介绍，其参数 K（推荐时选择与物品最相似的物品个数）取 10。LFM 是隐语义模型算法，这是一种基于矩阵分解的利用机器学习中的随机梯度下降法的算法，算法具体过程在书籍[1]的 2.5.1 节有详

细的介绍，其参数 ratio（随机取的负样本与正样本的比例）取 10，F（隐类个数）取 100，max_iter（学习迭代次数）取 30。SLIM 是本论文所研究的稀疏线性算法，这是一种基于邻域的利用机器学习中的坐标下降法的算法，此算法过程与 4.2.3 节里面的算法 5 完全相同，其参数α（l1(lasso)在 l1 正则项和 l2(ridge regression) 正则项之间的所占比例）取 0.5，t（λ的取值相对于λ_{max}的倍数）取 0.02，max_iter（最大学习迭代次数）取 1000，tol（每次更新的最小变化阈值）取 0.0001。表格 2 中的 SLIM 算法和表格 1 中的 d 算法是完全相同的，相应的运行结果数据基本一致，不同之处只是微小的误差，不影响结果的评价。LFM 推荐算法和 SLIM 推荐算法的过程都是由 Cython 实现。由于 UserCF 推荐算法和 ItemCF 推荐算法十分简单，所以没有使用 Cython 实现，它们是直接使用 Python 实现的。即便如此，这两种算法运行仍然很快，且不同的实现方式对于推荐结果不会有影响。所以总体来说，推荐算法的实现方式之间的差异不影响我们的结果评估。

观察表格 2 的数据，从总体上看，SLIM 推荐算法在 top-n 推荐算法中确实是相当不错的，四大推荐算法评估指标是目前几乎最好的，这也印证了论文[2]中作者得出的 SLIM 推荐算法的推荐质量比几乎所有的主流推荐算法的推荐质量都更优的结论。将推荐算法 SLIM，LFM 与推荐算法 UserCF，ItemCF 作对比，可以发现基于机器学习的推荐算法的推荐质量确实优于传统的非机器学习算法。虽然在表格 2 中 SLIM 推荐算法耗时最多，比 LFM 算法耗时还要多很多，但是在我们实在是需要性能的情况下可以将 SLIM 算法简化为表格 1 中的算法 b，表格 1 中的算法 b 的推荐质量仍然不比 LFM 算法的推荐质量差，同时耗时几乎只有 LFM 算法的耗时的一半。况且对每个用户产生推荐列表的运算总时间比较长不能说明算法实用性不好，因为在 SLIM 推荐算法计算出物品之间关系矩阵 W 后，对单个用户 u 产生推荐列表只需要取出用户物品行为矩阵 A 的第 u 行与 W 矩阵相乘再排序即可，矩阵 A 的第 u 行与 W 矩阵相乘的时间复杂度最大为$O(n^2)$，然而由于矩阵 A 与矩阵 W 都是稀疏矩阵，实际时间复杂度会远小于$O(n^2)$，排序的时间复杂度是$O(n\log n)$。所以 SLIM 算法在已知物品间关系矩阵 W 的情况下对单个用户产生推荐列表并不慢，物品个数 n 不是特别大的且对 W 矩阵更新频率不要求特别高的实际系统中完全可以使用 SLIM 推荐算法。

最后，我将数据集改为 movielens 的 ml-1m 数据集，每种算法运行三次后，对每项参数取平均值，得到一个对比结果。我的运行结果展示于下面的表格 3 中：

表格 3　SLIM 算法与其它算法在 ml-1m 数据集下的推荐质量对比

推荐算法	准确率	召回率	覆盖率	流行度	耗时
UserCF	24.963%	11.945%	20.130%	7.296248	110.252s
ItemCF	22.366%	10.702%	19.401%	7.259928	36.377s
LFM	27.222%	13.026%	23.610%	7.065418	419.792s
SLIM	29.126%	13.937%	46.519%	6.953065	686.150s

表格 3 中每种推荐算法在前面已有介绍，每种推荐算法参数的取值情况也和表格 2 中完全一样。表格 3 的运行过程与表格 2 的运行过程唯一的区别就是数据集不同。表格 3 使用了 ml-1m 的数据集，比表格 2 的 ml-100k 数据集更大。我们可以看到，当数据集更大的时候，推荐算法的推荐质量会随之提升。这是因为当数据集更大时，推荐算法能得到更多的信息，从而导致了推荐质量的提升。并且改变数据集后，SLIM 推荐算法的推荐质量优势仍然十分明显。

第五章 总 结

本论文研究了 SLIM 推荐算法。

首先，在本论文的第二章中，我介绍了 SLIM 推荐算法的模型以及 SLIM 推荐算法的原理。SLIM 的模型比较简单，就是利用用户行为矩阵 A 以及借助一个物品之间关系矩阵 W，计算并得到 A 与 W 的矩阵相乘的结果，这个结果矩阵即是我们需要的推荐信息。

然后，在本论文的第三四章中我自己根据 SLIM 推荐算法模型，使用自己的思路设计并实现了 SLIM 推荐算法。我的实现方式只是一个参考。设计实现的过程在协助大家进一步理解 SLIM 推荐算法的同时，也搭建了一个对 SLIM 推荐算法测试评估的环境。

最后，在 4.4 节运行结果那一段中，我展示了我的实验结果并且根据实验结果进行了一定的分析。SLIM 推荐算法相比于传统推荐算法可以在离线实验中得到更好的推荐质量。而且还具有不错的实用性。并且我的对正则项系数的取值方式可以额外大幅提升算法对长尾的挖掘能力，这是 SLIM 推荐算法一种可选的实现方式。

总体来说，SLIM 推荐算法是一种 top-n 推荐质量高，实用性好的推荐算法。

致 谢

本文是我在指导教师和研究生学长的指导下完成的。从论文的立题到最终完成，他们都给予了我极大的关怀和帮助，并提出了宝贵的意见。值此论文结束之际，我以诚挚的心情向他们表示衷心的感谢，感谢在这半年时间里对我的亲切关怀、热情鼓励和悉心指导。

感谢我们班各位老师和同学所给予的关心和帮助！

最后，特别感谢我的父母给予的极大的支持和理解！

参考文献

[1] 项亮的《推荐系统实践》

[2] SLIM: Sparse Linear Methods for Top-N Recommender Systems (Xia Ning and George Karypis Computer Science & Engineering University of Minnesota, Minneapolis, MN)

[3] Regularization Paths for Generalized Linear Models via Coordinate Descent (Jerome Friedman Trevor Hastie∗ Rob Tibshirani Department of Statistics, Stanford University April 29, 2009)

[4] R. Tibshirani, “Regression shrinkage and selection via the lasso,” Journal of the Royal Statistical Society (Series B), vol. 58, pp. 267–288, 1996.

[5] H. Zou and T. Hastie, “Regularization and variable selection via the elastic net,” Journal Of The Royal Statistical Society Series B, vol. 67, no. 2, pp. 301–320, 2005.