# Web 大数据挖掘

## —协同过滤推荐系统

作者：孔令晏

## 一、 数据集基本信息

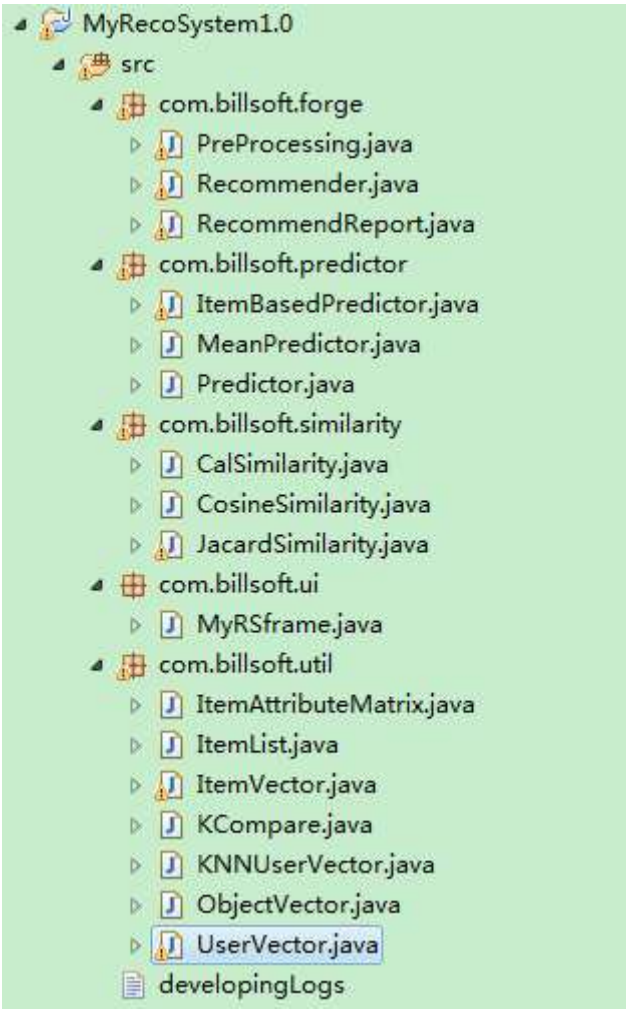
程序中使用的数据集中的训练数据集为 train.txt，含有用户使用历史信息统计数据；ItemAttribute.txt 文件中含有部分 item 的属性信息，test.txt 文件中含有需要评分的用户和 item。

在数据集中我们发现很多噪音，比如在 train 文件中有很多物品的评分为 0，并不意味着用户真的对该物品进行了评分，极有可能是用户对该物品使用过，但是并不是特别喜欢或者讨厌，若我们将之以 0 分来看的话在计算用户相似度时将严重影响我们的数据结果，因此在第一步的数据输入中对评分为 0 的物品进行了过滤；另一个文件 ItemAttribute.txt 中对于物品属性的描述中也需要一些特殊处理，比如对于为“None”的属性，我们将之置为了 0。

根据程序运行中的统计来看，数据集中共含有用户 19835 人，物品共 507172 件（ItemAttribute.txt 中），针对给出的测试文件中的 item 信息我们给出的预测值的项数为 748877724。

## 二、 算法细节

1. 程序整体结构如下：




1）forge 包中包含我们的测试与报告程序（RecommendReport.java），测试数据集生成程序（PreProcessing.java）以及我们的核心主程序（Recommender.java）。

2） ui 包中包含的是我们生成的图形界面的终极程序

3） util 包中包含我们需要的一些基础类，如 UserVector，ObjectVector，ItemList；以及用于计算 item 相似性的 ItemAttributeMatrix.java，计算用户相似性的 KCompare.java，KNNUserVector.java 。

4）similarity 包中包含的是计算给定的两个 UserVector 的相似

度的具体方法实现，其中 CosineSimilarity.java 是我们本程序使用到的，使用 Cosine 相似度的类；JacardSimilarity.java 是先前曾经使用过的计算两个 user 之间的 Jacard 相似度的方法，已废弃；

5）predictor 包中存放的是我们实际进行评分的函数。Predictor 是一个接口，然后，其他的两个类：MeanPredictor 和 ItemBasedPredictor 是他的两个实现，MeanPredictor 用于实现不基于 ItemAttribute 的预测，由于其效果并不如下面的 ItemBasedPredictor，因此已经废弃；ItemBasedPredictor 实现基于用户和 Item 属性的预测，会根据相似用户对于相同物品的评分以及该用户对于相似物品的评分进行加权计算出最后的评分值。

2. 程序基本思路

为了实现基于 ItemAttribute 和 User 购买历史的协同推荐，我们需要首先将 User 的信息和 Item 的信息载入到内存中，而由于 Java 虚拟机的限制，默认内存大小被限制在了 300M 以下，我们通过为虚拟机指定最大堆内存大小的方法来扩大其最大内存量为 800M，方法是在 eclipse 中 run -> Run Configuration -> Arguments /VM arguments 中添加：-Xms800m -Xmx800m 即可指定虚拟机的最大堆内存。

1） User 信息：

程序首先将 train 文件中的用户购买历史信息载入到一个特定的 HashMap 中，该 HashMap 中 key 值为用户 id，

value 值为一个 UserVector，该 UserVector 中记录了该用户购买过的所有产品信息。每当读入一个用户的 item 及评分就向该容器的相应位置添加一个信息，并更新这个 user 的总评分和购买到的 item 数量：size。UserVector 提供了一个 getAvgScore 函数用于输出当前用户的平均评分。

2） 相似用户（User）的发现：

相似用户的发现中我们使用了优先队列用于得到前 10 个相似度最大的用户，当给出一个特定的用户 ID 时，我们可以得到他购买过的 Item 信息，然后将他与其他所有用户进行比较，计算相似度（在 CosineSimilarity.java 中），相似度的计算我们使用的是去中心化的 Cosine 相似度，每个 item 的评分值都会减去用户的平均评分值，这样可以更好表达数据相关关系。每次计算都将相似度和 item 同时添加进优先队列，优先队列中有一个我们指定的比较器，比较器会将这些 item 的相似性进行比较队列头为相似性最低的 item，若队列满，则向队列中添加 item 时会将头元素挤出队列，从而，留下了相似性最大的所有用户：knn 中所有 knnUserVecor。

3） Item 信息：

从 ItemAttribute.txt 中载入所有 Item 的属性信息，我们

使用两种结构存储该 Item 信息，第一种是使用一个 ItemList 类，该类中的一个 HashMap 用来装载我们读到的所有 Item 和 attribute 信息，该类我们做到了尽量精简，attribute 是存放在了一个 int 型数组中了，这个 ItemList 类在我们读入 Item 的 attribute 属性信息时被逐渐充实，这样做的目的是为下一步计算 item 的相似性提供方便，我们可以根据提供的 itemID 直接得到 attribute 信息，从而可以在接下来的数据结构 ItemAttributeMatrix 中定位到该 Item 的相似 Item。

4） 相似项（Item）的发现：

提到 ItemAttributeMatrix 我们就必须要介绍一下它的特殊性，它并不是根据 itemID 来定位数据的，而是根据 item 的 attribute 属性信息，由于有两个 attribute，因此，attribute1 作为横坐标，attribute2 作为纵坐标，若两个 item 的横纵坐标相似则一定可以根据 attribute 找到彼此，这样做可以极大的提高相似项的发现效率，其核心思想其实就是最近距离的 item 作为相似项返回，我们搜索这样的相似项的方法是像包装礼物一样，一重一重的增加搜索半径，然后，每次只需要搜索多出来的一层中是否有数据既可以，这样就可以得到所有在 itemID 给出的 attribute 值定位到的位置附近某一个半径范围内的所有相似物体

5） 评分值的估计

具体实现在 ItemBasedPredictor.java 中。预测 score 需要传入 knn（所有相似用户），用户的 Items，similarItems（所有相似项），用户的平均平均值。评估过程如下：

首先判断该用户历史上是否购买过该物品，若购买过则将该值作为预测值 score 输出；

然后搜索相似用户是否购买过相似物品，若购买过则加入预测的数据集中，同时需要记录下有多少个相似用户符合条件，将得到的所有评分值相加然后除以个数，得到平均值，作为最后预测值的一部：similarUserScore；

接着，在用户购买过的物品中搜索是否购买过相似物品（similarItems）中的任意一个，同样的计算出这些评分的均值：similarItemScore；

最后将 simialrUserScore 和 similarItemScore 以不同的权值相加获得最终评分； 此处，我们赋予 similarUserScore 的权值为 0.4，similarItemScore 的权值为 0.6

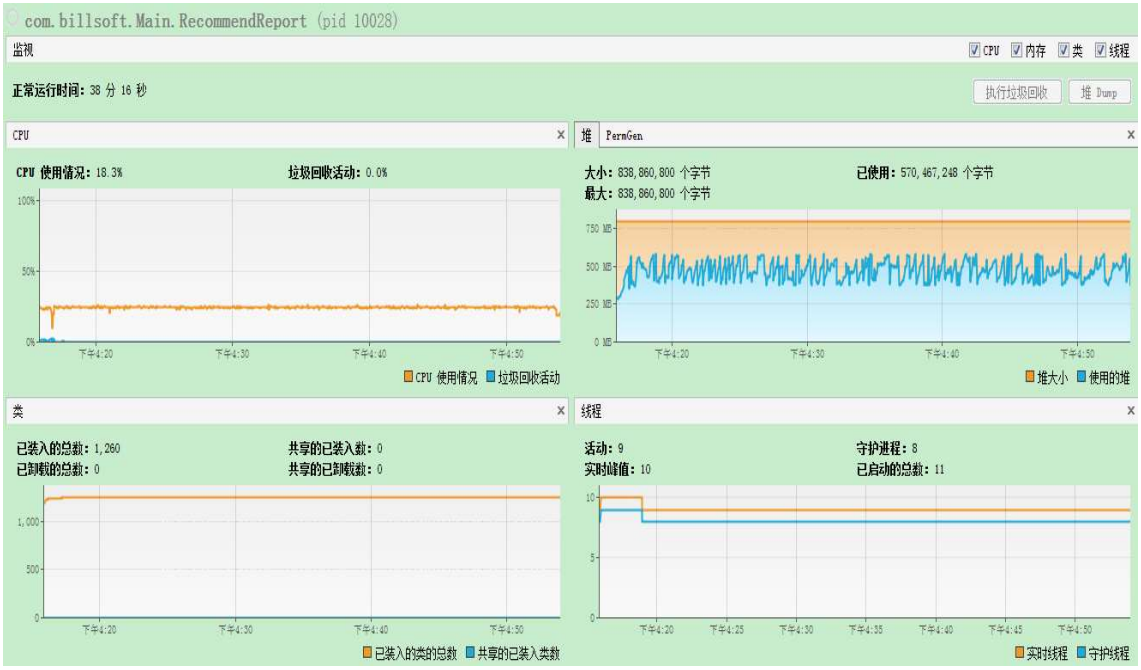
## 三、 推荐算法实验结果

为了测试算法的运行结果，我们选取了 train 中的部分数据进行了测试，测试数据使用 forge 包下的 PreProcessing 程序自动产生，一个是 real.txt，另一个是 test.txt 文件，其中 real.txt 包含我们需要测试的 user 的 item 的真实评分结果；test.txt 中

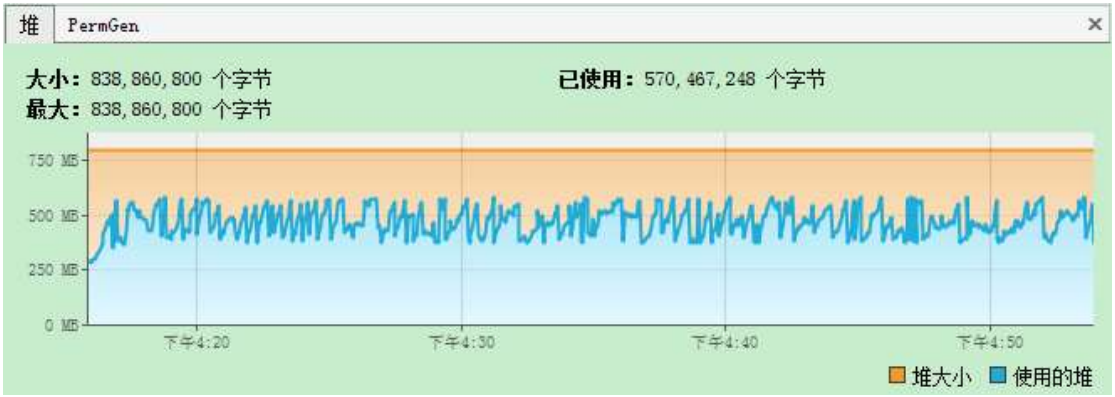
包含的是我们需要测试的 user 和 item 的值，然后，其中没有 item 的评分值，运行评分程序后程序会根据 train 文件、ItemAttribute.txt 文件对 test 文件中的 item 进行预测，输出到 result.txt 中，若我们运行的是 RecommendReport.java，程序还会将统计信息输出到源文件的 report.txt 中。

根据统计训练时间共 79719 ms，测试数据的 RMSE 为：18.9，内存消耗及 cpu 占用情况如下：

运行监控信息概览

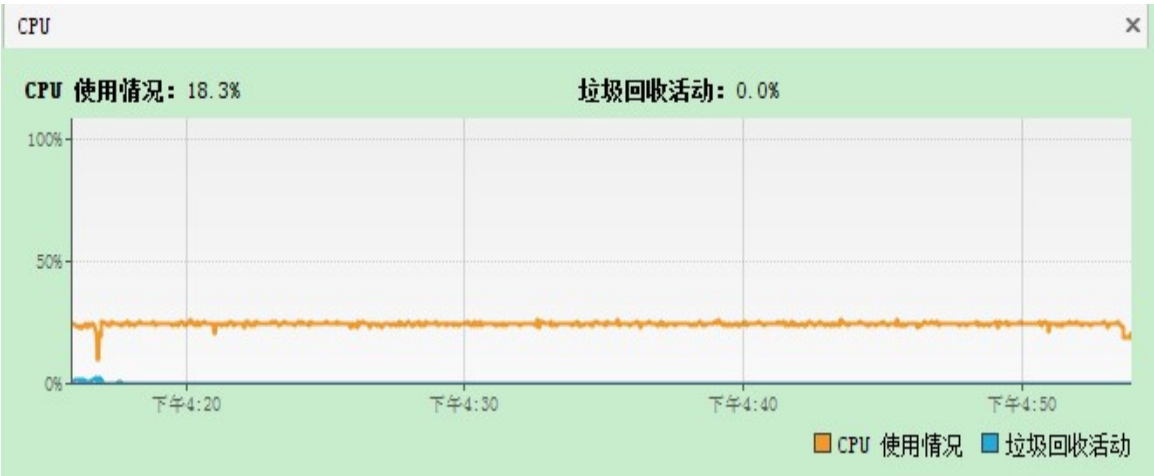



堆内存的使用情况：



可见，堆内存的使用峰值为 550M 左右，均值在 500M 左右。

CPU 使用情况：




## 四、 算法实验分析和理论分析

算法获取相似项的方法使用到了 KNN（K 最近邻）方法，该方法几乎无需训练，但是这也带来了每次进行评分时都需要遍历所有用户来获取相似用户的弊端，如果可以的话我们可以使用某些机器学习的方法对 user 进行分类，然后再进行相似用户发现时就会快很多，这也是下一步可以进行考虑的地方。

为了加快算法速度，算法中使用了一个我自己设计的矩阵 ItemAttribute-Matrix，之所以使用矩阵来存储这些数据是因为 Attribute 只有两个，因此只需要二维，而且这是个稀疏矩阵也就带来了压缩的可能，我们压缩是采用方格法，每个方格的大小为 100*100，这样当 Attribute1 或者 Attribute2 的差值在 100 以内的所有 item 就都会落入同一个方格中，存储这些元素时我们

使用的是一个 HasMap，正是由于矩阵的稀疏性，因此同一个方格内的元素的数量不会太多；这样放置元素还可以带来一个便利就是便于搜索相似项，毕竟所在位置的一定半径内的 item 都可以作为相似项取出。