[](http://blog.fens.me/wp-content/uploads/2013/10/mahout-Recommendation.png)

**前言**

用Mahout来构建推荐系统，是一件既简单又困难的事情。简单是因为Mahout完整地封装了“协同过滤”算法，并实现了并行化，提供非常简单的API接口；困难是因为我们不了解算法细节，很难去根据业务的场景进行算法配置和调优。

本文将深入算法API去解释Mahout推荐算法底层的一些事。

**目录**

1. Mahout推荐算法介绍
2. 算法评判标准：召回率与准确率
3. Recommender.java的API接口
4. 测试程序：RecommenderTest.java
5. 基于用户的协同过滤算法UserCF
6. 基于物品的协同过滤算法ItemCF
7. SlopeOne算法
8. KNN Linear interpolation item–based推荐算法
9. SVD推荐算法
10. Tree Cluster-based 推荐算法
11. Mahout推荐算法总结

**1. Mahout推荐算法介绍**

Mahoutt推荐算法，从数据处理能力上，可以划分为2类：

* 单机内存算法实现
* 基于Hadoop的分步式算法实现

**1). 单机内存算法实现**

单机内存算法实现：就是在单机下运行的算法，是由cf.taste项目实现的，像我的们熟悉的UserCF,ItemCF都支持单机内存运行，并且参数可以灵活配置。单机算法的基本实例，请参考文章：[用Maven构建Mahout项目](http://blog.fens.me/hadoop-mahout-maven-eclipse/)

单机内存算法的问题在于，受限于单机的资源。对于中等规模的数据，像1G,10G的数据量，有能力进行计算，但是超过100G的数据量，对于单机来说是不可能完成的任务。

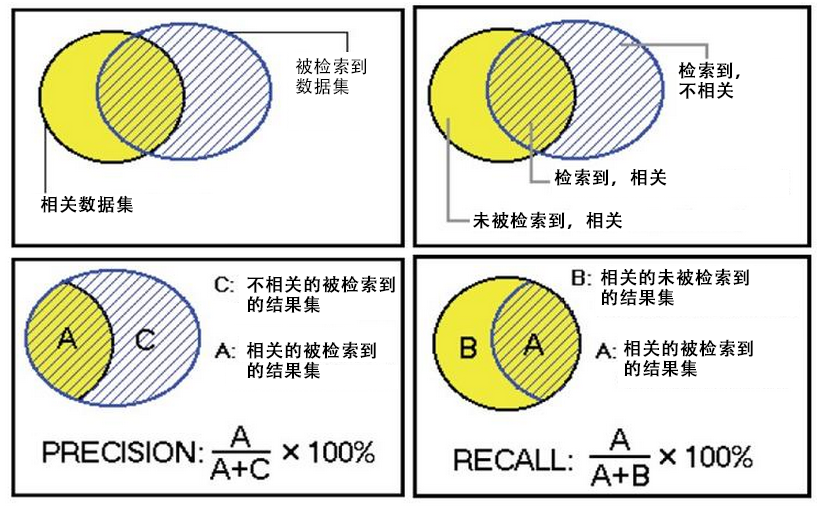
**2). 基于Hadoop的分步式算法实现**

基于Hadoop的分步式算法实现：就是把单机内存算法并行化，把任务分散到多台计算机一起运行。Mahout提供了ItemCF基于Hadoop并行化算法实现。基于Hadoop的分步式算法实现，请参考文章：  
[Mahout分步式程序开发 基于物品的协同过滤ItemCF](http://blog.fens.me/hadoop-mahout-mapreduce-itemcf/)

分步式并行算法的问题在于，如何让单机算法并行化。在单机算法中，我们只需要考虑算法，数据结构，内存，CPU就够了，但是分步式算法还要额外考虑很多的情况，比如多节点的数据合并，数据排序，网路通信的效率，节点宕机重算，数据分步式存储等等的很多问题。

**2. 算法评判标准：召回率(recall)与查准率(precision)**

Mahout提供了2个评估推荐器的指标，查准率和召回率（查全率），这两个指标是搜索引擎中经典的度量方法。

[](http://blog.fens.me/wp-content/uploads/2013/10/precision_recall.png)

相关 不相关

检索到 A C

未检索到 B D

* A：检索到的，相关的 （搜到的也想要的）
* B：未检索到的，但是相关的 （没搜到，然而实际上想要的）
* C：检索到的，但是不相关的 （搜到的但没用的）
* D：未检索到的，也不相关的 （没搜到也没用的）

被检索到的越多越好，这是追求“查全率”，即A/(A+B)，越大越好。  
被检索到的，越相关的越多越好，不相关的越少越好，这是追求“查准率”，即A/(A+C)，越大越好。

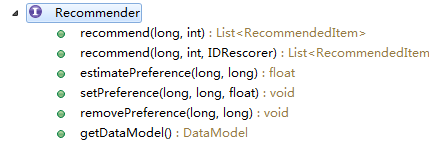
在大规模数据集合中，这两个指标是相互制约的。当希望索引出更多的数据的时候，查准率就会下降，当希望索引更准确的时候，会索引更少的数据。

**3. Recommender的API接口**

**1). 系统环境:**

* Win7 64bit
* Java 1.6.0\_45
* Maven 3
* Eclipse Juno Service Release 2
* Mahout 0.8
* Hadoop 1.1.2

**2). Recommender接口文件：**  
org.apache.mahout.cf.taste.recommender.Recommender.java

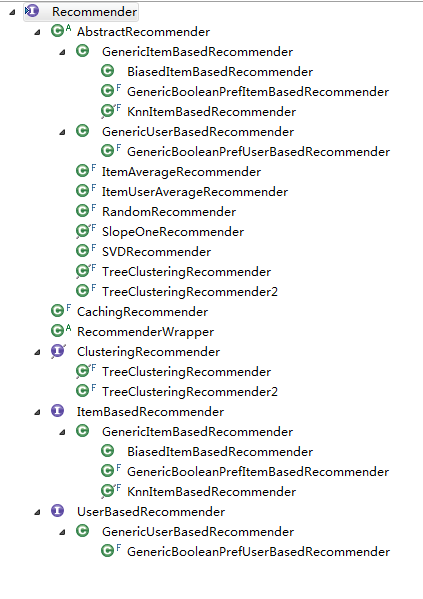
[](http://blog.fens.me/wp-content/uploads/2013/10/mahout-Recommender-class.png)

接口中方法的解释：

* recommend(long userID, int howMany): 获得推荐结果，给userID推荐howMany个Item
* recommend(long userID, int howMany, IDRescorer rescorer): 获得推荐结果，给userID推荐howMany个Item，可以根据rescorer对结构重新排序。
* estimatePreference(long userID, long itemID): 当打分为空，估计用户对物品的打分
* setPreference(long userID, long itemID, float value): 赋值用户，物品，打分
* removePreference(long userID, long itemID): 删除用户对物品的打分
* getDataModel(): 提取推荐数据

通过Recommender接口，我可以猜出核心算法，应该会在子类的estimatePreference()方法中进行实现。

**3). 通过继承关系到Recommender接口的子类：**

[](http://blog.fens.me/wp-content/uploads/2013/10/mahout-Recommender-hierarchy.png)

推荐算法实现类：

* GenericUserBasedRecommender: 基于用户的推荐算法
* GenericItemBasedRecommender: 基于物品的推荐算法
* KnnItemBasedRecommender: 基于物品的KNN推荐算法
* SlopeOneRecommender: Slope推荐算法
* SVDRecommender: SVD推荐算法
* TreeClusteringRecommender：TreeCluster推荐算法

下面将分别介绍每种算法的实现。

**4. 测试程序：RecommenderTest.java**

测试数据集：item.csv

1,101,5.0

1,102,3.0

1,103,2.5

2,101,2.0

2,102,2.5

2,103,5.0

2,104,2.0

3,101,2.5

3,104,4.0

3,105,4.5

3,107,5.0

4,101,5.0

4,103,3.0

4,104,4.5

4,106,4.0

5,101,4.0

5,102,3.0

5,103,2.0

5,104,4.0

5,105,3.5

5,106,4.0

测试程序：org.conan.mymahout.recommendation.job.RecommenderTest.java

package org.conan.mymahout.recommendation.job;

import java.io.IOException;

import java.util.List;

import org.apache.mahout.cf.taste.common.TasteException;

import org.apache.mahout.cf.taste.eval.RecommenderBuilder;

import org.apache.mahout.cf.taste.impl.common.LongPrimitiveIterator;

import org.apache.mahout.cf.taste.model.DataModel;

import org.apache.mahout.cf.taste.recommender.RecommendedItem;

import org.apache.mahout.common.RandomUtils;

public class RecommenderTest {

final static int NEIGHBORHOOD\_NUM = 2;

final static int RECOMMENDER\_NUM = 3;

public static void main(String[] args) throws TasteException, IOException {

RandomUtils.useTestSeed();

String file = "datafile/item.csv";

DataModel dataModel = RecommendFactory.buildDataModel(file);

slopeOne(dataModel);

}

public static void userCF(DataModel dataModel) throws TasteException{}

public static void itemCF(DataModel dataModel) throws TasteException{}

public static void slopeOne(DataModel dataModel) throws TasteException{}

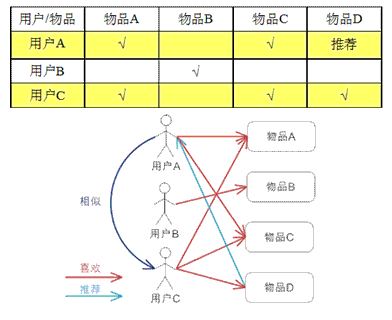
...

每种算法都一个单独的方法进行算法测试，如userCF(),itemCF(),slopeOne()….

**5. 基于用户的协同过滤算法UserCF**

基于用户的协同过滤，通过不同用户对物品的评分来评测用户之间的相似性，基于用户之间的相似性做出推荐。简单来讲就是：给用户推荐和他兴趣相似的其他用户喜欢的物品。

举例说明：

[](http://blog.fens.me/wp-content/uploads/2013/10/image015.gif)

*基于用户的 CF 的基本思想相当简单，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。计算上，就是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。图 2 给出了一个例子，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。*

上文中图片和解释文字，摘自： <https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy2/>

**算法API: org.apache.mahout.cf.taste.impl.recommender.GenericUserBasedRecommender**

@Override

public float estimatePreference(long userID, long itemID) throws TasteException {

DataModel model = getDataModel();

Float actualPref = model.getPreferenceValue(userID, itemID);

if (actualPref != null) {

return actualPref;

}

long[] theNeighborhood = neighborhood.getUserNeighborhood(userID);

return doEstimatePreference(userID, theNeighborhood, itemID);

}

protected float doEstimatePreference(long theUserID, long[] theNeighborhood, long itemID) throws TasteException {

if (theNeighborhood.length == 0) {

return Float.NaN;

}

DataModel dataModel = getDataModel();

double preference = 0.0;

double totalSimilarity = 0.0;

int count = 0;

for (long userID : theNeighborhood) {

if (userID != theUserID) {

// See GenericItemBasedRecommender.doEstimatePreference() too

Float pref = dataModel.getPreferenceValue(userID, itemID);

if (pref != null) {

double theSimilarity = similarity.userSimilarity(theUserID, userID);

if (!Double.isNaN(theSimilarity)) {

preference += theSimilarity \* pref;

totalSimilarity += theSimilarity;

count++;

}

}

}

}

// Throw out the estimate if it was based on no data points, of course, but also if based on

// just one. This is a bit of a band-aid on the 'stock' item-based algorithm for the moment.

// The reason is that in this case the estimate is, simply, the user's rating for one item

// that happened to have a defined similarity. The similarity score doesn't matter, and that

// seems like a bad situation.

if (count <= 1) {

return Float.NaN;

}

float estimate = (float) (preference / totalSimilarity);

if (capper != null) {

estimate = capper.capEstimate(estimate);

}

return estimate;

}

测试程序:

public static void userCF(DataModel dataModel) throws TasteException {

UserSimilarity userSimilarity = RecommendFactory.userSimilarity(RecommendFactory.SIMILARITY.EUCLIDEAN, dataModel);

UserNeighborhood userNeighborhood = RecommendFactory.userNeighborhood(RecommendFactory.NEIGHBORHOOD.NEAREST, userSimilarity, dataModel, NEIGHBORHOOD\_NUM);

RecommenderBuilder recommenderBuilder = RecommendFactory.userRecommender(userSimilarity, userNeighborhood, true);

RecommendFactory.evaluate(RecommendFactory.EVALUATOR.AVERAGE\_ABSOLUTE\_DIFFERENCE, recommenderBuilder, null, dataModel, 0.7);

RecommendFactory.statsEvaluator(recommenderBuilder, null, dataModel, 2);

LongPrimitiveIterator iter = dataModel.getUserIDs();

while (iter.hasNext()) {

long uid = iter.nextLong();

List list = recommenderBuilder.buildRecommender(dataModel).recommend(uid, RECOMMENDER\_NUM);

RecommendFactory.showItems(uid, list, true);

}

}

程序输出：

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:1.0

Recommender IR Evaluator: [Precision:0.5,Recall:0.5]

uid:1,(104,4.333333)(106,4.000000)

uid:2,(105,4.049678)

uid:3,(103,3.512787)(102,2.747869)

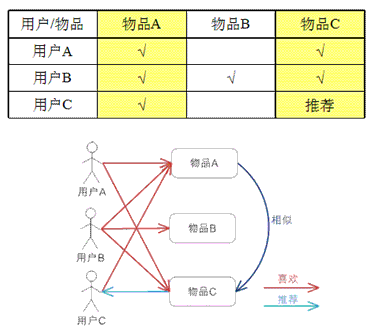
uid:4,(102,3.000000)

用R语言重写UserCF的实现，请参考文章：[用R解析Mahout用户推荐协同过滤算法(UserCF)](http://blog.fens.me/r-mahout-usercf/)

**6. 基于物品的协同过滤算法ItemCF**

基于item的协同过滤，通过用户对不同item的评分来评测item之间的相似性，基于item之间的相似性做出推荐。简单来讲就是：给用户推荐和他之前喜欢的物品相似的物品。

举例说明：

[](http://blog.fens.me/wp-content/uploads/2013/10/image017.gif)

*基于物品的 CF 的原理和基于用户的 CF 类似，只是在计算邻居时采用物品本身，而不是从用户的角度，即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。从计算的角度看，就是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。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。*

上文中图片和解释文字，摘自： <https://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy2/>

**算法API: org.apache.mahout.cf.taste.impl.recommender.GenericItemBasedRecommender**

@Override

public float estimatePreference(long userID, long itemID) throws TasteException {

PreferenceArray preferencesFromUser = getDataModel().getPreferencesFromUser(userID);

Float actualPref = getPreferenceForItem(preferencesFromUser, itemID);

if (actualPref != null) {

return actualPref;

}

return doEstimatePreference(userID, preferencesFromUser, itemID);

}

protected float doEstimatePreference(long userID, PreferenceArray preferencesFromUser, long itemID)

throws TasteException {

double preference = 0.0;

double totalSimilarity = 0.0;

int count = 0;

double[] similarities = similarity.itemSimilarities(itemID, preferencesFromUser.getIDs());

for (int i = 0; i < similarities.length; i++) {

double theSimilarity = similarities[i];

if (!Double.isNaN(theSimilarity)) {

// Weights can be negative!

preference += theSimilarity \* preferencesFromUser.getValue(i);

totalSimilarity += theSimilarity;

count++;

}

}

// Throw out the estimate if it was based on no data points, of course, but also if based on

// just one. This is a bit of a band-aid on the 'stock' item-based algorithm for the moment.

// The reason is that in this case the estimate is, simply, the user's rating for one item

// that happened to have a defined similarity. The similarity score doesn't matter, and that

// seems like a bad situation.

if (count <= 1) {

return Float.NaN;

}

float estimate = (float) (preference / totalSimilarity);

if (capper != null) {

estimate = capper.capEstimate(estimate);

}

return estimate;

}

测试程序:

public static void itemCF(DataModel dataModel) throws TasteException {

ItemSimilarity itemSimilarity = RecommendFactory.itemSimilarity(RecommendFactory.SIMILARITY.EUCLIDEAN, dataModel);

RecommenderBuilder recommenderBuilder = RecommendFactory.itemRecommender(itemSimilarity, true);

RecommendFactory.evaluate(RecommendFactory.EVALUATOR.AVERAGE\_ABSOLUTE\_DIFFERENCE, recommenderBuilder, null, dataModel, 0.7);

RecommendFactory.statsEvaluator(recommenderBuilder, null, dataModel, 2);

LongPrimitiveIterator iter = dataModel.getUserIDs();

while (iter.hasNext()) {

long uid = iter.nextLong();

List list = recommenderBuilder.buildRecommender(dataModel).recommend(uid, RECOMMENDER\_NUM);

RecommendFactory.showItems(uid, list, true);

}

}

程序输出：

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:0.8676552772521973

Recommender IR Evaluator: [Precision:0.5,Recall:1.0]

uid:1,(105,3.823529)(104,3.722222)(106,3.478261)

uid:2,(106,2.984848)(105,2.537037)(107,2.000000)

uid:3,(106,3.648649)(102,3.380000)(103,3.312500)

uid:4,(107,4.722222)(105,4.313953)(102,4.025000)

uid:5,(107,3.736842)

**7. SlopeOne算法**

这个算法在mahout-0.8版本中，已经被@Deprecated。

SlopeOne是一种简单高效的协同过滤算法。通过均差计算进行评分。SlopeOne论文下载([PDF](http://www.daniel-lemire.com/fr/documents/publications/lemiremaclachlan_sdm05.pdf))

**1). 举例说明：**  
用户X，Y，Z，对于物品A,B进行打分，如下表，求Z对B的打分是多少？

[](http://blog.fens.me/wp-content/uploads/2013/10/slopeone.png)

Slope one算法认为：平均值可以代替某两个未知个体之间的打分差异，事物A对事物B的平均差是：((5 - 4) + (4 - 2)) / 2 = 1.5，就得到Z对B的打分是，3-1.5 = 1.5。

Slope one算法将用户的评分之间的关系看作简单的线性关系：

Y = mX + b

**2). 平均加权计算：**  
用户X，Y，Z，对于物品A,B,C进行打分，如下表，求Z对A的打分是多少？

[](http://blog.fens.me/wp-content/uploads/2013/10/slopeone2.png)

* 1. 计算A和B的平均差, ((5-3)+(3-4))/2=0.5
* 2. 计算A和C的平均差, (5-2)/1=3
* 3. Z对A的评分，通过AB得到, 2+0.5=2.5
* 4. Z对A的评分，通过AC得到，5+3=8
* 5. 通过加权平均计算Z对A的评分：A和B都有评价的用户数为2,A和C都有评价的用户数为1，权重为别是2和1， (2\*2.5+1\*8)/(2+1)=13/3=4.33

通过这种简单的方式，我们可以快速计算出一个评分项，完成推荐过程！

**算法API: org.apache.mahout.cf.taste.impl.recommender.slopeone.SlopeOneRecommender**

@Override

public float estimatePreference(long userID, long itemID) throws TasteException {

DataModel model = getDataModel();

Float actualPref = model.getPreferenceValue(userID, itemID);

if (actualPref != null) {

return actualPref;

}

return doEstimatePreference(userID, itemID);

}

private float doEstimatePreference(long userID, long itemID) throws TasteException {

double count = 0.0;

double totalPreference = 0.0;

PreferenceArray prefs = getDataModel().getPreferencesFromUser(userID);

RunningAverage[] averages = diffStorage.getDiffs(userID, itemID, prefs);

int size = prefs.length();

for (int i = 0; i < size; i++) {

RunningAverage averageDiff = averages[i];

if (averageDiff != null) {

double averageDiffValue = averageDiff.getAverage();

if (weighted) {

double weight = averageDiff.getCount();

if (stdDevWeighted) {

double stdev = ((RunningAverageAndStdDev) averageDiff).getStandardDeviation();

if (!Double.isNaN(stdev)) {

weight /= 1.0 + stdev;

}

// If stdev is NaN, then it is because count is 1. Because we're weighting by count,

// the weight is already relatively low. We effectively assume stdev is 0.0 here and

// that is reasonable enough. Otherwise, dividing by NaN would yield a weight of NaN

// and disqualify this pref entirely

// (Thanks Daemmon)

}

totalPreference += weight \* (prefs.getValue(i) + averageDiffValue);

count += weight;

} else {

totalPreference += prefs.getValue(i) + averageDiffValue;

count += 1.0;

}

}

}

if (count <= 0.0) {

RunningAverage itemAverage = diffStorage.getAverageItemPref(itemID);

return itemAverage == null ? Float.NaN : (float) itemAverage.getAverage();

} else {

return (float) (totalPreference / count);

}

}

测试程序:

public static void slopeOne(DataModel dataModel) throws TasteException {

RecommenderBuilder recommenderBuilder = RecommendFactory.slopeOneRecommender();

RecommendFactory.evaluate(RecommendFactory.EVALUATOR.AVERAGE\_ABSOLUTE\_DIFFERENCE, recommenderBuilder, null, dataModel, 0.7);

RecommendFactory.statsEvaluator(recommenderBuilder, null, dataModel, 2);

LongPrimitiveIterator iter = dataModel.getUserIDs();

while (iter.hasNext()) {

long uid = iter.nextLong();

List list = recommenderBuilder.buildRecommender(dataModel).recommend(uid, RECOMMENDER\_NUM);

RecommendFactory.showItems(uid, list, true);

}

}

程序输出：

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:1.3333333333333333

Recommender IR Evaluator: [Precision:0.25,Recall:0.5]

uid:1,(105,5.750000)(104,5.250000)(106,4.500000)

uid:2,(105,2.286115)(106,1.500000)

uid:3,(106,2.000000)(102,1.666667)(103,1.625000)

uid:4,(105,4.976859)(102,3.509071)

**8. KNN Linear interpolation item–based推荐算法**

这个算法在mahout-0.8版本中，已经被@Deprecated。

算法来自论文：  
This algorithm is based in the paper of Robert M. Bell and Yehuda Koren in ICDM '07.

(TODO未完)

**算法API: org.apache.mahout.cf.taste.impl.recommender.knn.KnnItemBasedRecommender**

@Override

protected float doEstimatePreference(long theUserID, PreferenceArray preferencesFromUser, long itemID)

throws TasteException {

DataModel dataModel = getDataModel();

int size = preferencesFromUser.length();

FastIDSet possibleItemIDs = new FastIDSet(size);

for (int i = 0; i < size; i++) {

possibleItemIDs.add(preferencesFromUser.getItemID(i));

}

possibleItemIDs.remove(itemID);

List mostSimilar = mostSimilarItems(itemID, possibleItemIDs.iterator(),

neighborhoodSize, null);

long[] theNeighborhood = new long[mostSimilar.size() + 1];

theNeighborhood[0] = -1;

List usersRatedNeighborhood = Lists.newArrayList();

int nOffset = 0;

for (RecommendedItem rec : mostSimilar) {

theNeighborhood[nOffset++] = rec.getItemID();

}

if (!mostSimilar.isEmpty()) {

theNeighborhood[mostSimilar.size()] = itemID;

for (int i = 0; i < theNeighborhood.length; i++) {

PreferenceArray usersNeighborhood = dataModel.getPreferencesForItem(theNeighborhood[i]);

int size1 = usersRatedNeighborhood.isEmpty() ? usersNeighborhood.length() : usersRatedNeighborhood.size();

for (int j = 0; j < size1; j++) {

if (i == 0) {

usersRatedNeighborhood.add(usersNeighborhood.getUserID(j));

} else {

if (j >= usersRatedNeighborhood.size()) {

break;

}

long index = usersRatedNeighborhood.get(j);

if (!usersNeighborhood.hasPrefWithUserID(index) || index == theUserID) {

usersRatedNeighborhood.remove(index);

j--;

}

}

}

}

}

double[] weights = null;

if (!mostSimilar.isEmpty()) {

weights = getInterpolations(itemID, theNeighborhood, usersRatedNeighborhood);

}

int i = 0;

double preference = 0.0;

double totalSimilarity = 0.0;

for (long jitem : theNeighborhood) {

Float pref = dataModel.getPreferenceValue(theUserID, jitem);

if (pref != null) {

double weight = weights[i];

preference += pref \* weight;

totalSimilarity += weight;

}

i++;

}

return totalSimilarity == 0.0 ? Float.NaN : (float) (preference / totalSimilarity);

}

}

测试程序:

public static void itemKNN(DataModel dataModel) throws TasteException {

ItemSimilarity itemSimilarity = RecommendFactory.itemSimilarity(RecommendFactory.SIMILARITY.EUCLIDEAN, dataModel);

RecommenderBuilder recommenderBuilder = RecommendFactory.itemKNNRecommender(itemSimilarity, new NonNegativeQuadraticOptimizer(), 10);

RecommendFactory.evaluate(RecommendFactory.EVALUATOR.AVERAGE\_ABSOLUTE\_DIFFERENCE, recommenderBuilder, null, dataModel, 0.7);

RecommendFactory.statsEvaluator(recommenderBuilder, null, dataModel, 2);

LongPrimitiveIterator iter = dataModel.getUserIDs();

while (iter.hasNext()) {

long uid = iter.nextLong();

List list = recommenderBuilder.buildRecommender(dataModel).recommend(uid, RECOMMENDER\_NUM);

RecommendFactory.showItems(uid, list, true);

}

}

程序输出：

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:1.5

Recommender IR Evaluator: [Precision:0.5,Recall:1.0]

uid:1,(107,5.000000)(104,3.501168)(106,3.498198)

uid:2,(105,2.878995)(106,2.878086)(107,2.000000)

uid:3,(103,3.667444)(102,3.667161)(106,3.667019)

uid:4,(107,4.750247)(102,4.122755)(105,4.122709)

uid:5,(107,3.833621)

**9. SVD推荐算法**

(TODO未完)

**算法API: org.apache.mahout.cf.taste.impl.recommender.svd.SVDRecommender**

@Override

public float estimatePreference(long userID, long itemID) throws TasteException {

double[] userFeatures = factorization.getUserFeatures(userID);

double[] itemFeatures = factorization.getItemFeatures(itemID);

double estimate = 0;

for (int feature = 0; feature < userFeatures.length; feature++) {

estimate += userFeatures[feature] \* itemFeatures[feature];

}

return (float) estimate;

}

测试程序:

public static void svd(DataModel dataModel) throws TasteException {

RecommenderBuilder recommenderBuilder = RecommendFactory.svdRecommender(new ALSWRFactorizer(dataModel, 10, 0.05, 10));

RecommendFactory.evaluate(RecommendFactory.EVALUATOR.AVERAGE\_ABSOLUTE\_DIFFERENCE, recommenderBuilder, null, dataModel, 0.7);

RecommendFactory.statsEvaluator(recommenderBuilder, null, dataModel, 2);

LongPrimitiveIterator iter = dataModel.getUserIDs();

while (iter.hasNext()) {

long uid = iter.nextLong();

List list = recommenderBuilder.buildRecommender(dataModel).recommend(uid, RECOMMENDER\_NUM);

RecommendFactory.showItems(uid, list, true);

}

}

程序输出：

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:0.09990564982096355

Recommender IR Evaluator: [Precision:0.5,Recall:1.0]

uid:1,(104,4.032909)(105,3.390885)(107,1.858541)

uid:2,(105,3.761718)(106,2.951908)(107,1.561116)

uid:3,(103,5.593422)(102,2.458930)(106,-0.091259)

uid:4,(105,4.068329)(102,3.534025)(107,0.206257)

uid:5,(107,0.105169)

**10. Tree Cluster-based 推荐算法**

这个算法在mahout-0.8版本中，已经被@Deprecated。

(TODO未完)

**算法API: org.apache.mahout.cf.taste.impl.recommender.TreeClusteringRecommender**

@Override

public float estimatePreference(long userID, long itemID) throws TasteException {

DataModel model = getDataModel();

Float actualPref = model.getPreferenceValue(userID, itemID);

if (actualPref != null) {

return actualPref;

}

buildClusters();

List topRecsForUser = topRecsByUserID.get(userID);

if (topRecsForUser != null) {

for (RecommendedItem item : topRecsForUser) {

if (itemID == item.getItemID()) {

return item.getValue();

}

}

}

// Hmm, we have no idea. The item is not in the user's cluster

return Float.NaN;

}

测试程序:

public static void treeCluster(DataModel dataModel) throws TasteException {

UserSimilarity userSimilarity = RecommendFactory.userSimilarity(RecommendFactory.SIMILARITY.LOGLIKELIHOOD, dataModel);

ClusterSimilarity clusterSimilarity = RecommendFactory.clusterSimilarity(RecommendFactory.SIMILARITY.FARTHEST\_NEIGHBOR\_CLUSTER, userSimilarity);

RecommenderBuilder recommenderBuilder = RecommendFactory.treeClusterRecommender(clusterSimilarity, 10);

RecommendFactory.evaluate(RecommendFactory.EVALUATOR.AVERAGE\_ABSOLUTE\_DIFFERENCE, recommenderBuilder, null, dataModel, 0.7);

RecommendFactory.statsEvaluator(recommenderBuilder, null, dataModel, 2);

LongPrimitiveIterator iter = dataModel.getUserIDs();

while (iter.hasNext()) {

long uid = iter.nextLong();

List list = recommenderBuilder.buildRecommender(dataModel).recommend(uid, RECOMMENDER\_NUM);

RecommendFactory.showItems(uid, list, true);

}

}

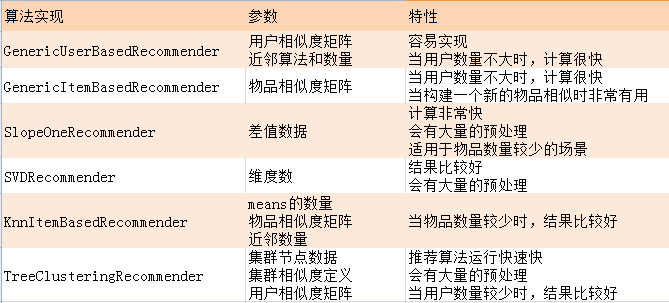
程序输出：

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:NaN

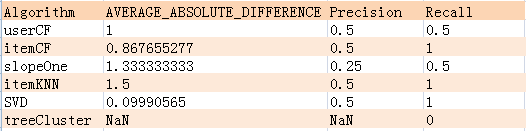
Recommender IR Evaluator: [Precision:NaN,Recall:0.0]

**11. Mahout推荐算法总结**

算法及适用场景：

[](http://blog.fens.me/wp-content/uploads/2013/10/recommender-intro.png)

算法评分的结果：

[](http://blog.fens.me/wp-content/uploads/2013/10/recommender-score.png)

通过对上面几种算法的一平分比较：itemCF,itemKNN,SVD的Rrecision,Recall的评分值是最好的，并且itemCF和SVD的AVERAGE\_ABSOLUTE\_DIFFERENCE是最低的，所以，从算法的角度知道了，哪个算法是更准确的或者会索引到更多的数据集。

另外的一些因素：

* 1. 这3个指标，并不能直接决定计算结果一定itemCF,SVD好
* 2. 各种算法的参数我们并没有调优
* 3. 数据量和数据分布，是影响算法的评分

**程序源代码下载**

<https://github.com/bsspirit/maven_mahout_template/tree/mahout-0.8/src/main/java/org/conan/mymahout/recommendation/job>

**转载请注明出处：**  
[http://blog.fens.me/mahout-recommendation-api](http://blog.fens.me/mahout-recommendation-api/mahout-recommendation-api)