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

**前言**

随着大数据思想实施的落地，推荐系统也开始倍受关注。不光是电商，各种互联网应用都开始应用推荐系统，像搜索，社交网络，音乐，餐饮，地图服务等等。

在以前，我们没有使用推荐算法的时候，我们是通过设置各种约束条件，匹配数据的自然属性呈现给用户，这种就是基于规则的系统。比如，用户购买了一个商品，我们会推荐同类别的其他商品，通过类别属性作为推荐的规则。后来问题就出现了，当用户一次性买了多种类别的不同商品的时候，前一条规则就失败了，我们要进一步设计规则，IT类别优先推荐，价格高的产品优先推荐…..几个回合下来，我们要不停的增加规则，以至于规则有可能的会前后冲突，增加一条新的规则会让推荐结果越来越不好，而且还无法解释是为什么。

推荐算法从另一角度入手，解决了基于规则设置的问题。下面将用Mahout来构建一个职位推荐算法引擎。

**目录**

1. Mahout推荐框架概述
2. 需求分析：职位推荐引擎指标设计
3. 算法模型：推荐算法
4. 架构设计：职位推荐引擎系统架构
5. 程序开发：基于Mahout的推荐算法实现

**1. Mahout推荐系统框架概述**

Mahout框架包含了一套完整的推荐系统引擎，标准化的数据结构，多样的算法实现，简单的开发流程。Mahout推荐的推荐系统引擎是模块化的，分为5个主要部分组成：数据模型，相似度算法，近邻算法，推荐算法，算法评分器。

更详细的介绍，请参考文章：[从源代码剖析Mahout推荐引擎](http://blog.fens.me/mahout-recommend-engine/)

**2. 需求分析：职位推荐引擎指标设计**

下面我们将从一个公司案例出发来全面的解释，如何进行职位推荐引擎指标设计。

案例介绍：  
互联网某职业社交网站，主要产品包括 个人简历展示页，人脉圈，微博及分享链接，职位发布，职位申请，教育培训等。

用户在完成注册后，需要完善自己的个人信息，包括教育背景，工作经历，项目经历，技能专长等等信息。然后，你要告诉网站，你是否想找工作！！当你选择“是”（求职中），网站会从数据库中为你推荐你可能感兴趣的职位。

通过简短的描述，我们可以粗略地看出，这家职业社交网站的定位和主营业务。核心点有2个：

* 用户：尽可能多的保存有效完整的用户资料
* 服务：帮助用户找到工作，帮助猎头和企业找到员工

因此，职位推荐引擎 将成为这个网站的核心功能。

**KPI指标设计**

* 通过推荐带来的职位浏览量: 职位网页的PV
* 通过推荐带来的职位申请量: 职位网页的有效转化

**3. 算法模型：推荐算法**

2个测试数据集：

* pv.csv: 职位被浏览的信息,包括用户ID，职位ID
* job.csv: 职位基本信息,包括职位ID，发布时间，工资标准

**1). pv.csv**

* 2列数据：用户ID，职位ID(userid,jobid)
* 浏览记录:2500条
* 用户数:1000个，用户ID:1-1000
* 职位数:200个，职位ID：1-200

部分数据：

1,11

2,136

2,187

3,165

3,1

3,24

4,8

4,199

5,32

5,100

6,14

7,59

7,147

8,92

9,165

9,80

9,171

10,45

10,31

10,1

10,152

**2). job.csv**

* 3列数据：职位ID，发布时间，工资标准(jobid,create\_date,salary)
* 职位数:200个，职位ID：1-200

部分数据：

1,2013-01-24,5600

2,2011-03-02,5400

3,2011-03-14,8100

4,2012-10-05,2200

5,2011-09-03,14100

6,2011-03-05,6500

7,2012-06-06,37000

8,2013-02-18,5500

9,2010-07-05,7500

10,2010-01-23,6700

11,2011-09-19,5200

12,2010-01-19,29700

13,2013-09-28,6000

14,2013-10-23,3300

15,2010-10-09,2700

16,2010-07-14,5100

17,2010-05-13,29000

18,2010-01-16,21800

19,2013-05-23,5700

20,2011-04-24,5900

为了完成KPI的指标，我们把问题用“技术”语言转化一下：我们需要让职位的推荐结果更准确，从而增加用户的点击。

* 1. 组合使用推荐算法，选出“评估推荐器”验证得分较高的算法
* 2. 人工验证推荐结果
* 3. 职位有时效性，推荐的结果应该是发布半年内的职位
* 4. 工资的标准，应不低于用户浏览职位工资的平均值的80%

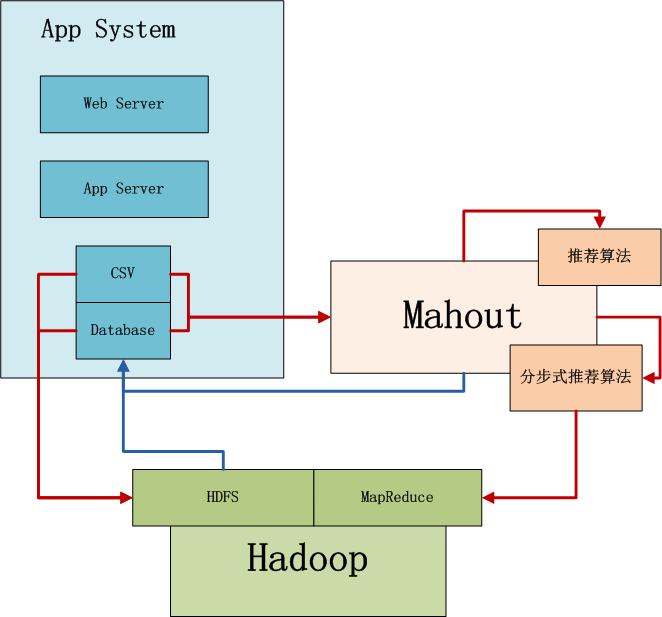
我们选择UserCF,ItemCF,SlopeOne的 3种推荐算法，进行7种组合的测试。

* userCF1: LogLikelihoodSimilarity + NearestNUserNeighborhood + GenericBooleanPrefUserBasedRecommender
* userCF2: CityBlockSimilarity+ NearestNUserNeighborhood + GenericBooleanPrefUserBasedRecommender
* userCF3: UserTanimoto + NearestNUserNeighborhood + GenericBooleanPrefUserBasedRecommender
* itemCF1: LogLikelihoodSimilarity + GenericBooleanPrefItemBasedRecommender
* itemCF2: CityBlockSimilarity+ GenericBooleanPrefItemBasedRecommender
* itemCF3: ItemTanimoto + GenericBooleanPrefItemBasedRecommender
* slopeOne：SlopeOneRecommender

关于的推荐算法的详细介绍，请参考文章：[Mahout推荐算法API详解](http://blog.fens.me/mahout-recommendation-api/)

关于算法的组合的详细介绍，请参考文章：[从源代码剖析Mahout推荐引擎](http://blog.fens.me/mahout-recommend-engine/)

**4. 架构设计：职位推荐引擎系统架构**

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

上图中，左边是Application业务系统，右边是Mahout，下边是Hadoop集群。

* 1. 当数据量不太大时，并且算法复杂，直接选择用Mahout读取CSV或者Database数据，在单机内存中进行计算。Mahout是多线程的应用，会并行使用单机所有系统资源。
* 2. 当数据量很大时，选择并行化算法(ItemCF)，先业务系统的数据导入到Hadoop的HDFS中，然后用Mahout访问HDFS实现算法，这时算法的性能与整个Hadoop集群有关。
* 3. 计算后的结果，保存到数据库中，方便查询

**5. 程序开发：基于Mahout的推荐算法实现**

开发环境mahout版本为0.8。 ，请参考文章：[用Maven构建Mahout项目](http://blog.fens.me/hadoop-mahout-maven-eclipse/)

新建Java类：

* RecommenderEvaluator.java, 选出“评估推荐器”验证得分较高的算法
* RecommenderResult.java, 对指定数量的结果人工比较
* RecommenderFilterOutdateResult.java，排除过期职位
* RecommenderFilterSalaryResult.java，排除工资过低的职位

**1). RecommenderEvaluator.java, 选出“评估推荐器”验证得分较高的算**  
源代码：

public class RecommenderEvaluator {

final static int NEIGHBORHOOD\_NUM = 2;

final static int RECOMMENDER\_NUM = 3;

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

String file = "datafile/job/pv.csv";

DataModel dataModel = RecommendFactory.buildDataModelNoPref(file);

userLoglikelihood(dataModel);

userCityBlock(dataModel);

userTanimoto(dataModel);

itemLoglikelihood(dataModel);

itemCityBlock(dataModel);

itemTanimoto(dataModel);

slopeOne(dataModel);

}

public static RecommenderBuilder userLoglikelihood(DataModel dataModel) throws TasteException, IOException {

System.out.println("userLoglikelihood");

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

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

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

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

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

return recommenderBuilder;

}

public static RecommenderBuilder userCityBlock(DataModel dataModel) throws TasteException, IOException {

System.out.println("userCityBlock");

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

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

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

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

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

return recommenderBuilder;

}

public static RecommenderBuilder userTanimoto(DataModel dataModel) throws TasteException, IOException {

System.out.println("userTanimoto");

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

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

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

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

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

return recommenderBuilder;

}

public static RecommenderBuilder itemLoglikelihood(DataModel dataModel) throws TasteException, IOException {

System.out.println("itemLoglikelihood");

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

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

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

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

return recommenderBuilder;

}

public static RecommenderBuilder itemCityBlock(DataModel dataModel) throws TasteException, IOException {

System.out.println("itemCityBlock");

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

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

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

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

return recommenderBuilder;

}

public static RecommenderBuilder itemTanimoto(DataModel dataModel) throws TasteException, IOException {

System.out.println("itemTanimoto");

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

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

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

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

return recommenderBuilder;

}

public static RecommenderBuilder slopeOne(DataModel dataModel) throws TasteException, IOException {

System.out.println("slopeOne");

RecommenderBuilder recommenderBuilder = RecommendFactory.slopeOneRecommender();

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

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

return recommenderBuilder;

}

public static RecommenderBuilder knnLoglikelihood(DataModel dataModel) throws TasteException, IOException {

System.out.println("knnLoglikelihood");

ItemSimilarity itemSimilarity = RecommendFactory.itemSimilarity(RecommendFactory.SIMILARITY.LOGLIKELIHOOD, 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);

return recommenderBuilder;

}

public static RecommenderBuilder knnTanimoto(DataModel dataModel) throws TasteException, IOException {

System.out.println("knnTanimoto");

ItemSimilarity itemSimilarity = RecommendFactory.itemSimilarity(RecommendFactory.SIMILARITY.TANIMOTO, 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);

return recommenderBuilder;

}

public static RecommenderBuilder knnCityBlock(DataModel dataModel) throws TasteException, IOException {

System.out.println("knnCityBlock");

ItemSimilarity itemSimilarity = RecommendFactory.itemSimilarity(RecommendFactory.SIMILARITY.CITYBLOCK, 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);

return recommenderBuilder;

}

public static RecommenderBuilder svd(DataModel dataModel) throws TasteException {

System.out.println("svd");

RecommenderBuilder recommenderBuilder = RecommendFactory.svdRecommender(new ALSWRFactorizer(dataModel, 5, 0.05, 10));

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

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

return recommenderBuilder;

}

public static RecommenderBuilder treeClusterLoglikelihood(DataModel dataModel) throws TasteException {

System.out.println("treeClusterLoglikelihood");

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

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

RecommenderBuilder recommenderBuilder = RecommendFactory.treeClusterRecommender(clusterSimilarity, 3);

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

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

return recommenderBuilder;

}

}

运行结果，控制台输出：

userLoglikelihood

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:0.2741487771272658

Recommender IR Evaluator: [Precision:0.6424242424242422,Recall:0.4098360655737705]

userCityBlock

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:0.575306732961736

Recommender IR Evaluator: [Precision:0.919580419580419,Recall:0.4371584699453552]

userTanimoto

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:0.5546485136181523

Recommender IR Evaluator: [Precision:0.6625766871165644,Recall:0.41803278688524603]

itemLoglikelihood

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:0.5398332608612343

Recommender IR Evaluator: [Precision:0.26229508196721296,Recall:0.26229508196721296]

itemCityBlock

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:0.9251437840891661

Recommender IR Evaluator: [Precision:0.02185792349726776,Recall:0.02185792349726776]

itemTanimoto

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:0.9176432856689655

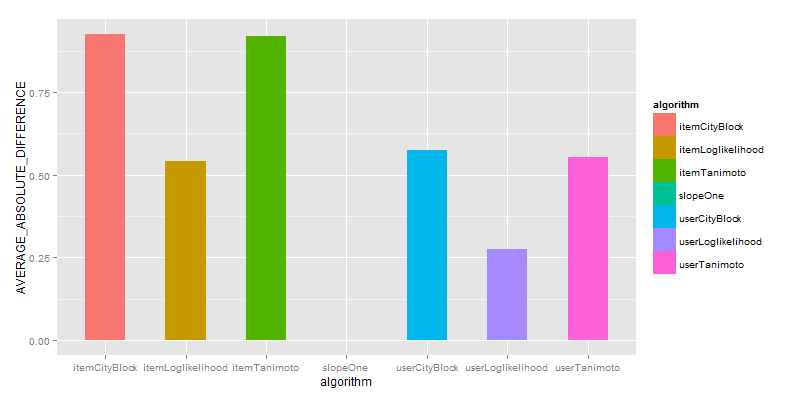
Recommender IR Evaluator: [Precision:0.26229508196721296,Recall:0.26229508196721296]

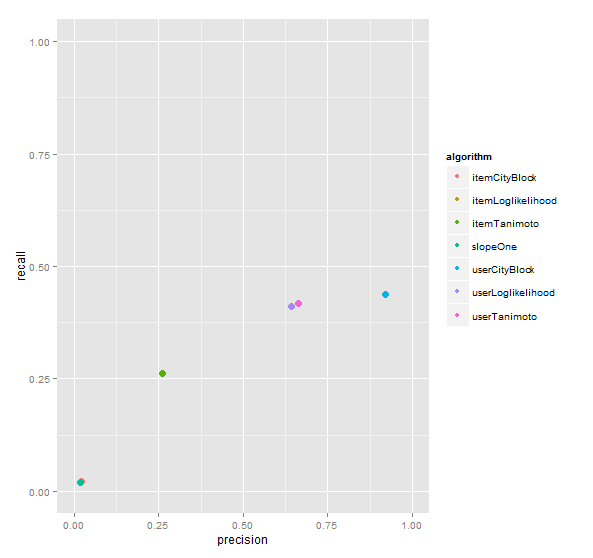
slopeOne

AVERAGE\_ABSOLUTE\_DIFFERENCE Evaluater Score:0.0

Recommender IR Evaluator: [Precision:0.01912568306010929,Recall:0.01912568306010929]

可视化“评估推荐器”输出：

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

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

UserCityBlock算法评估的结果是最好的，基于UserCF的算法比ItemCF都要好，SlopeOne算法几乎没有得分。

**2). RecommenderResult.java, 对指定数量的结果人工比较**  
为得到差异化结果，我们分别取UserCityBlock,itemLoglikelihood，对推荐结果人工比较。

源代码：

public class RecommenderResult {

final static int NEIGHBORHOOD\_NUM = 2;

final static int RECOMMENDER\_NUM = 3;

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

String file = "datafile/job/pv.csv";

DataModel dataModel = RecommendFactory.buildDataModelNoPref(file);

RecommenderBuilder rb1 = RecommenderEvaluator.userCityBlock(dataModel);

RecommenderBuilder rb2 = RecommenderEvaluator.itemLoglikelihood(dataModel);

LongPrimitiveIterator iter = dataModel.getUserIDs();

while (iter.hasNext()) {

long uid = iter.nextLong();

System.out.print("userCityBlock =>");

result(uid, rb1, dataModel);

System.out.print("itemLoglikelihood=>");

result(uid, rb2, dataModel);

}

}

public static void result(long uid, RecommenderBuilder recommenderBuilder, DataModel dataModel) throws TasteException {

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

RecommendFactory.showItems(uid, list, false);

}

}

控制台输出：只截取部分结果

...

userCityBlock =>uid:968,(61,0.333333)

itemLoglikelihood=>uid:968,(121,1.429362)(153,1.239939)(198,1.207726)

userCityBlock =>uid:969,

itemLoglikelihood=>uid:969,(75,1.326499)(30,0.873100)(85,0.763344)

userCityBlock =>uid:970,

itemLoglikelihood=>uid:970,(13,0.748417)(156,0.748417)(122,0.748417)

userCityBlock =>uid:971,

itemLoglikelihood=>uid:971,(38,2.060951)(104,1.951208)(83,1.941735)

userCityBlock =>uid:972,

itemLoglikelihood=>uid:972,(131,1.378395)(4,1.349386)(87,0.881816)

userCityBlock =>uid:973,

itemLoglikelihood=>uid:973,(196,1.432040)(140,1.398066)(130,1.380335)

userCityBlock =>uid:974,(19,0.200000)

itemLoglikelihood=>uid:974,(145,1.994049)(121,1.794289)(98,1.738027)

...

我们查看uid=974的用户推荐信息：

搜索pv.csv：

> pv[which(pv$userid==974),]

userid jobid

2426 974 106

2427 974 173

2428 974 82

2429 974 188

2430 974 78

搜索job.csv:

> job[job$jobid %in% c(145,121,98,19),]

jobid create\_date salary

19 19 2013-05-23 5700

98 98 2010-01-15 2900

121 121 2010-06-19 5300

145 145 2013-08-02 6800

上面两种算法，推荐的结果都是2010年的职位，这些结果并不是太好，接下来我们要排除过期职位，只保留2013年的职位。

**3).RecommenderFilterOutdateResult.java，排除过期职位**  
源代码：

public class RecommenderFilterOutdateResult {

final static int NEIGHBORHOOD\_NUM = 2;

final static int RECOMMENDER\_NUM = 3;

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

String file = "datafile/job/pv.csv";

DataModel dataModel = RecommendFactory.buildDataModelNoPref(file);

RecommenderBuilder rb1 = RecommenderEvaluator.userCityBlock(dataModel);

RecommenderBuilder rb2 = RecommenderEvaluator.itemLoglikelihood(dataModel);

LongPrimitiveIterator iter = dataModel.getUserIDs();

while (iter.hasNext()) {

long uid = iter.nextLong();

System.out.print("userCityBlock =>");

filterOutdate(uid, rb1, dataModel);

System.out.print("itemLoglikelihood=>");

filterOutdate(uid, rb2, dataModel);

}

}

public static void filterOutdate(long uid, RecommenderBuilder recommenderBuilder, DataModel dataModel) throws TasteException, IOException {

Set jobids = getOutdateJobID("datafile/job/job.csv");

IDRescorer rescorer = new JobRescorer(jobids);

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

RecommendFactory.showItems(uid, list, true);

}

public static Set getOutdateJobID(String file) throws IOException {

BufferedReader br = new BufferedReader(new FileReader(new File(file)));

Set jobids = new HashSet();

String s = null;

while ((s = br.readLine()) != null) {

String[] cols = s.split(",");

SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd");

Date date = null;

try {

date = df.parse(cols[1]);

if (date.getTime() < df.parse("2013-01-01").getTime()) {

jobids.add(Long.parseLong(cols[0]));

}

} catch (ParseException e) {

e.printStackTrace();

}

}

br.close();

return jobids;

}

}

class JobRescorer implements IDRescorer {

final private Set jobids;

public JobRescorer(Set jobs) {

this.jobids = jobs;

}

@Override

public double rescore(long id, double originalScore) {

return isFiltered(id) ? Double.NaN : originalScore;

}

@Override

public boolean isFiltered(long id) {

return jobids.contains(id);

}

}

控制台输出：只截取部分结果

...

itemLoglikelihood=>uid:965,(200,0.829600)(122,0.748417)(170,0.736340)

userCityBlock =>uid:966,(114,0.250000)

itemLoglikelihood=>uid:966,(114,1.516898)(101,0.864536)(99,0.856057)

userCityBlock =>uid:967,

itemLoglikelihood=>uid:967,(105,0.873100)(114,0.725016)(168,0.707119)

userCityBlock =>uid:968,

itemLoglikelihood=>uid:968,(174,0.735004)(39,0.696716)(185,0.696171)

userCityBlock =>uid:969,

itemLoglikelihood=>uid:969,(197,0.723203)(81,0.710230)(167,0.668358)

userCityBlock =>uid:970,

itemLoglikelihood=>uid:970,(13,0.748417)(122,0.748417)(28,0.736340)

userCityBlock =>uid:971,

itemLoglikelihood=>uid:971,(28,1.540753)(174,1.511881)(39,1.435575)

userCityBlock =>uid:972,

itemLoglikelihood=>uid:972,(14,0.800605)(60,0.794088)(163,0.710230)

userCityBlock =>uid:973,

itemLoglikelihood=>uid:973,(56,0.795529)(13,0.712680)(120,0.701026)

userCityBlock =>uid:974,(19,0.200000)

itemLoglikelihood=>uid:974,(145,1.994049)(89,1.578694)(19,1.435193)

...

我们查看uid=994的用户推荐信息：  
搜索pv.csv：

> pv[which(pv$userid==974),]

userid jobid

2426 974 106

2427 974 173

2428 974 82

2429 974 188

2430 974 78

搜索job.csv:

> job[job$jobid %in% c(19,145,89),]

jobid create\_date salary

19 19 2013-05-23 5700

89 89 2013-06-15 8400

145 145 2013-08-02 6800

排除过期的职位比较，我们发现userCityBlock结果都是19，itemLoglikelihood的第2，3的结果被替换为了得分更低的89和19。

**4).RecommenderFilterSalaryResult.java，排除工资过低的职位**

我们查看uid=994的用户，浏览过的职位。

> job[job$jobid %in% c(106,173,82,188,78),]

jobid create\_date salary

78 78 2012-01-29 6800

82 82 2010-07-05 7500

106 106 2011-04-25 5200

173 173 2013-09-13 5200

188 188 2010-07-14 6000

平均工资为=6140，我们觉得用户的浏览职位的行为，一般不会看比自己现在工资低的职位，因此设计算法，排除工资低于平均工资80%的职位，即排除工资小于4912的推荐职位(6140\*0.8=4912)

大家可以参考上文中RecommenderFilterOutdateResult.java,自行实现。

这样，我们就完成用Mahout构建职位推荐引擎的算法。如果没有Mahout，我们自己写这个算法引擎估计还要花个小半年的时间，善加利用开源技术会帮助我们飞一样的成长！！

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

######################################################  
看文字不过瘾，作者视频讲解，请访问网站：<http://onbook.me/video>  
######################################################