**用户聚类与个性化推荐指导说明书**

目录

[任务1. 购书用户聚类 1](#_Toc44278929)

[任务描述 1](#_Toc44278930)

[任务指导 2](#_Toc44278931)

[任务实现 2](#_Toc44278932)

[任务2. 推荐系统设计与实现 14](#_Toc44278933)

[任务描述 14](#_Toc44278934)

[任务指导 15](#_Toc44278935)

[任务实现 16](#_Toc44278936)

# 购书用户聚类

## 任务描述

**内 容**：通过Apache Mahout实现购书用户聚类分析

**知识点**：k-means算法、Canopy算法、使用MapReduce实现聚类算法、使用Apache Mahout实现购书用户聚类功能

**重 点**：使用Apache Mahout实现购书用户聚类功能

**难 点**：使用Apache Mahout实现购书用户聚类功能

## 任务指导

1. 介绍聚类算法的原理与分类
2. 介绍数据向量化、数据归一化、相似性度量
3. 介绍k-means算法、Canopy算法、使用MapReduce实现聚类算法
4. 一个简单的基于Apache Mahout聚类算法的实现
5. 学生独立完成使用Apache Mahout实现购书用户聚类功能

## 任务实现

1. **聚类算法**

聚类 (Clustering) 就是将数据对象分组成为多个类或者簇 (Cluster)，它的目标是：在同一个簇中的对象之间具有较高的相似度，而不同簇中的对象差别较大。所以，在很多应用中，一个簇中的数据对象可以被作为一个整体来对待，从而减少计算量或者提高计算质量。

聚类分析已经广泛的应用在许多应用中，包括模式识别，数据分析，图像处理以及市场研究。通过聚类，人们能意识到密集和稀疏的区域，发现全局的分布模式，以及数据属性之间的有趣的相互关系。聚类同时也在 Web 应用中起到越来越重要的作用。最被广泛使用的既是对 Web 上的文档进行分类，组织信息的发布，给用户一个有效分类的内容浏览系统（门户网站），同时可以加入时间因素，进而发现各个类内容的信息发展，最近被大家关注的主题和话题，或者分析一段时间内人们对什么样的内容比较感兴趣，这些有趣的应用都得建立在聚类的基础之上。作为一个数据挖掘的功能，聚类分析能作为独立的工具来获得数据分布的情况，观察每个簇的特点，集中对特定的某些簇做进一步的分析，此外，聚类分析还可以作为其他算法的预处理步骤，简化计算量，提高分析效率。

**聚类的目标就是将相似的物体进行分组，并将其标注，这里面的相似性体现在点与点之间的距离。**

1. **介绍k-means算法、Canopy算法**

* **k-means算法是一种被广泛使用的直接聚类算法**

在k-means算法中，每个聚簇用一个点来代表，这些聚簇用集合C来表示，从中选取k个代表聚簇，被称为聚簇中心。聚类算法通常用相似度的概念，使用某种距离公式对点集进行分组，在k-means算法中默认的相似度标准为欧氏距离。K-means实质是要最小化一个非负的代价函数，即最小化目标是每个点和离它最近的聚簇中心之间的欧氏距离的平方和，这也是k-means的目标函数。

* Canopy算法

Canopy算法最大的特点是不需要事先指定k值，即聚簇的个数，该算法将聚类过程分为两个过程：第一个过程，Canopy聚类选择简单、计算代价较低的方法计算对象相似性，将相似的对象放在一个子集中，这个子集叫做canopy。第二个过程，根据第一步生成的canopy，再进行一次聚类，这次选取的聚类算法在计算相似性方面会较第一步复杂、精确。

我们很容易想到，Canopy算法的第二步可以用k-menas来完成，而第一步则负责得出k值和初始的聚簇中心。

* **Apache Mahout机器学习库**

Apache Mahout 是 Apache Software Foundation (ASF) 旗下的一个开源项目，提供一些可扩展的机器学习领域经典算法的实现，旨在帮助开发人员更加方便快捷地创建智能应用程序，并且，在 Mahout 的最近版本中还加入了对 Apache Hadoop 的支持，使这些算法可以更高效的运行在云计算环境中。Mahout 中提供了常用的多种聚类算法：Canopy算法、k-means算法、模糊k-means算法等。

1. **介绍数据向量化、数据归一化、相似性度量**

* **数据向量化**

在进行聚类之前，需要将对象用有序的实数对来表示，这个过程就是向量化。而向量化的维度就代表了对象的特征。以苹果为例，我们用大小和颜色来表示一个苹果，大小和颜色就是苹果的特征，确定特征后就需要对其向量化，以方便进行相似性计算。

例如，对苹果的特征进行向量化后如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 苹果 | 大小 | 颜色 | 向量 |
| 小、红色 | 1 | 655 | （1，655） |
| 大、绿色 | 3 | 525 | （3，525） |
| 中、黄色 | 2 | 580 | （2，580） |
| 小、绿色 | 1 | 500 | （1，500） |

Mahout 的聚类算法将对象表示成一种简单的数据模型：向量 (Vector)。在向量数据描述的基础上，我们可以轻松的计算两个对象的相似性。

* 数据归一化

由于特征之间具有不同的量纲和量纲单位，所以在进行聚类之前，需要对数据进行归一化。数据归一化，也叫数据标准化，目的是消除指标之间的量纲影响，使得各指标处于同一个数量级，这样指标之间才具有可比性。

**常用的归一化方法：**

* Min-max标准化
* Z分数标准化
* 相似性度量

在对原始数据进行向量化和归一化以后，就可以计算对象之间的相似性，根据相似性来进行对象的归类。常用的是采用欧氏距离作为相似性度量，相似性度量的方式对于聚类结果影响很大，不同的度量方式可能导致完全不同的聚类结果。

**常见的度量方式：**

* 欧氏距离
* 曼哈顿距离
* 余弦距离
* ……

1. 一个简单的基于Apache Mahout聚类算法的实现

将聚类和Hadoop联系在一起最直接的方式就是利用MapReduce编程模型来实现聚类。Apache Mahout可以帮助我们省去一些繁重的编码工作。Apache Mahout本质上就是一个用MapReduce实现的算法库。Mahout包含许多实现，如聚类、分类、协同过滤、多层感知机等。由于使用了MapReduce实现，使得Mahout能够在Hadoop上运行并具有极强的扩展性。

public class KmeansHadoop {

private static final String HDFS = "hdfs://10.20.0.10:9000";

public static void main(String[] args) throws Exception {

String localFile = "D:\\clustering\_demo\\data\\iris.dat";

String inPath = HDFS + "/user/hdfs/mix\_data";

String seqFile = inPath + "/seqfile";

String seeds = inPath + "/seeds";

String outPath = inPath + "/result/";

String clusteredPoints = outPath + "/clusteredPoints";

JobConf conf = config();

HdfsUtil hdfs = new HdfsUtil(HDFS, conf);

hdfs.rmr(inPath);

hdfs.mkdirs(inPath);

hdfs.copyFile(localFile, inPath);

hdfs.ls(inPath);

//它用于将原始数据文件转换成 Mahout进行计算所需格式的文件 SequenceFile，它是Hadoop API提供的一种二进制文件支持。这种二进制文件直接将<key, value>对序列化到文件中。

InputDriver.runJob(new Path(inPath), new Path(seqFile), "org.apache.mahout.math.RandomAccessSparseVector");

//指定聚类的个数

int k = 3;

///指定输入路径，如前面介绍的一样，基于 Hadoop 的实现就是通过指定输入输出的文件路径来指定数据源的。

Path seqFilePath = new Path(seqFile);

Path clustersSeeds = new Path(seeds);

//声明一个计算距离的方法，这里选择了欧几里德距离

DistanceMeasure measure = new EuclideanDistanceMeasure();

//随机的选择 k 个作为簇的中心

clustersSeeds = RandomSeedGenerator.buildRandom(conf, seqFilePath, clustersSeeds, k, measure);

//调用 KMeansDriver.run 方法执行 K 均值聚类算法

KMeansDriver.run(conf, seqFilePath, clustersSeeds, new Path(outPath), 0.01, 10, true, 0.01, false);

Path outGlobPath = new Path(outPath, "clusters-\*-final");

Path clusteredPointsPath = new Path(clusteredPoints);

System.out.printf("Dumping out clusters from clusters: %s and clusteredPoints: %s\n", outGlobPath, clusteredPointsPath);

//调用 ClusterDumper 的 printClusters 方法将聚类结果打印出来。

ClusterDumper clusterDumper = new ClusterDumper(outGlobPath, clusteredPointsPath);

clusterDumper.printClusters(null);

}

public static JobConf config() {

JobConf conf = new JobConf(KmeansHadoop.class);

conf.setJobName("KmeansHadoop");

conf.addResource("classpath:/hadoop/core-site.xml");

conf.addResource("classpath:/hadoop/hdfs-site.xml");

conf.addResource("classpath:/hadoop/mapred-site.xml");

return conf;

}

}

1. 学生独立完成使用Apache Mahout实现购书用户聚类功能

* 提取数据并做归一化
* 在这里我们选取3个维度来做聚类分析：
* 用户订单数
* 用户订单平均金额
* 用户访问次数
* 用户订单数和用户订单平均金额都可以从order表得出，用户访问次数根据clickstream\_log表的sessionId字段得出，现在需要整合这两张表的数据，并保存到Hive的另一张表user\_dimension中，user\_dimension表的建表语句如下：

CREATE TABLE user\_dimension(

customerId STRING,

subTotal DOUBLE,

ordersCount DOUBLE,

sessionCount DOUBLE

) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

* 创建一个dimension\_and\_normalization.py文件，在其中编写一个prepare\_normaliz函数，用于执行HQL语句，将数据提取出来并插入到user\_dimension表中：

def prepare\_normaliz(start\_time, end\_time):

# 提取数据维度

hql\_1 = "insert overwrite table user\_dimension select t1.userId,t1.avg,t2.ordercount,t3.sessioncount from " \

"(select userId,avg(subTotal) avg from orders where dt<=" + start\_time + " and dt>=" + end\_time + " group by userId) t1 " \

"join (select userId,count(orderId) ordercount from orders where dt<="+start\_time+" and dt>="+end\_time+" group by userId) t2 on t1.userId=t2.userId " \

"join (select userId,count(sessionId) sessioncount from clickstream\_log where dt<="+start\_time+" and dt>="+end\_time+" group by userId) t3 on t1.userId=t3.userId";

HiveUtil.execute\_shell(hql\_1)

* user\_dimension表中的数据还不能直接作为Mahout的输入，因为CustomerId字段对于聚类没有意义，并且字段之间需要使用空格分隔。所以还需要一张Hive表，用来保存Mahout的输入数据，并做归一化处理。
* 创建一张Hive表cluster\_input用来保存Mahout的输入数据，字段之间改为使用空格分隔数据

CREATE TABLE cluster\_input(

subTotal DOUBLE,

ordersCount DOUBLE,

sessionCount DOUBLE

) ROW FORMAT DELIMITED FIELDS TERMINATED BY ' ';

* 将user\_dimension表中除了CustomerId外的其余数据导入到cluster\_input表。在prepare\_normaliz函数中增加如下代码：

# 将user\_dimension数据导入到cluster\_input中，去掉CustomerId

hql\_2 = "insert overwrite table cluster\_input select subTotal,orderCount,sessionCount from user\_dimension"

HiveUtil.execute\_shell(hql\_2)

* 对cluster\_input表中的数据做归一化处理,z分数标准化需要用到期望和标准差，在Hive中不能直接使用Hive函数求得z分数，只能使用JOIN的方式在后面的子句中分别将期望和标准差先计算出来。在prepare\_normaliz函数中增加如下代码：

hql\_3 = "insert overwrite table cluster\_input " \

"select (subTotal-avg\_subTotal)/std\_subTotal,(ordersCount-avg\_ordersCount)/std\_ordersCount,(sessionCount-avg\_sessionCount)/std\_sessionCount from cluster\_input " \

"join (select std(subTotal) std\_subTotal,std(ordersCount) std\_ordersCount,std(sessionCount) std\_sessionCount from cluster\_input) t1 on 1=1 " \

"join (select avg(subTotal) avg\_subTotal,avg(ordersCount) avg\_ordersCount,avg(sessionCount) avg\_sessionCount from cluster\_input) t2 on 1=1";

HiveUtil.execute\_shell(hql\_3)

至此，数据已经是归一化后的数据，可以进行聚类操作了。

* 维度相关性

在开始聚类前需要观察一下所选取的维度是否合适。一般来说，维度的选取要相互独立，一个维度的变化，不会引起另一个维度的变化。在选取维度时，一定要注意排除变量之间相关性的干扰，例如，用户订单数和用户订单总金额是不适合同时作为聚类的维度的。

对于一些内在的联系，或者不能直接判断是否存在相关性的情况，就需要采用数学的方式来验证是否存在相关性。

在概率论和统计学中，相关系统用来显示两个随机变量之间线性关系的强度和方向。例如常用的相关系统计算方式：皮尔逊相关系数（Pearson product-moment correlation coefficient，PPMCC或PCCs），用于度量两个变量X和Y之间的相关性（线性相关），其值介于-1和1之间。在自然科学领域中，该系数广泛用于度量两个变量之间的相关程度，定义为两个变量之间的协方差和标准差的商，数学表达式如下：

ç®å°éç¸å³ç³»æ°è®¡ç®å¬å¼

所以，当两个变量相互独立时，也就是E(XY)=E(X)E(Y)时，相关系数为0。

再来看一下相关系数矩阵，相关系数矩阵位于第i行和第j列（i,j）的元素是原始矩阵第i行和第j列的相关系数。例如，以聚类输入数据为例，一共3个维度，相关系数矩阵为：

ρ11 ρ12 ρ13

ρ21 ρ22 ρ23

ρ31 ρ32 ρ33

**其中：**

* 第1行第1列为用户订单数和自己的相关系数；
* 第1行第2列为用户订单数和用户订单平均金额之间的相关系数；
* 第1行第3列为用户订单数和用户访问次数之间的相关系数；
* 第2行第3列为用户订单平均金额和用户访问次数之间的相关系数；

因为ρij=ρji 且当i=j时，ρij=1，所以相关系数矩阵是对称矩阵，对角元素为1。

**下面使用HQL计算相关系数矩阵：**

#计算相关系数矩阵

hql\_4="select " \

"(avg(subTotal\*subTotal)-avg(subTotal)\*avg(subTotal))/(std(subTotal)\*std(subTotal))," \

"(avg(subTotal\*ordersCount)-avg(subTotal)\*avg(ordersCount))/(std(subTotal)\*std(ordersCount))," \

"(avg(subTotal\*sessionCount)-avg(subTotal)\*avg(sessionCount))/(std(subTotal)\*std(sessionCount))," \

"(avg(ordersCount\*subTotal)-avg(ordersCount)\*avg(subTotal))/(std(ordersCount)\*std(subTotal))," \

"(avg(ordersCount\*ordersCount)-avg(ordersCount)\*avg(ordersCount))/(std(ordersCount)\*std(ordersCount))," \

"(avg(ordersCount\*sessionCount)-avg(ordersCount)\*avg(sessionCount))/(std(ordersCount)\*std(sessionCount))," \

"(avg(sessionCount\*subTotal)-avg(sessionCount)\*avg(subTotal))/(std(sessionCount)\*std(subTotal))," \

"(avg(sessionCount\*ordersCount)-avg(sessionCount)\*avg(ordersCount))/(std(sessionCount)\*std(ordersCount))," \

"(avg(sessionCount\*sessionCount)-avg(sessionCount)\*avg(sessionCount))/(std(sessionCount)\*std(sessionCount)) " \

"from cluster\_input";

HiveUtil.execute\_shell(hql\_4)

得到列与列之间的相关系数，以此来评估维度之间的相关性，然后观察维度选取是否合适。

* 使用Mahout完成聚类
  + 编写Mahout程序实现聚类

Mahout执行聚类共分为4步：

* 1. 将输入文件序列化
  2. 生成canopy
  3. 利用生成的canopy执行k-means聚类
  4. 将聚类结果输出

public class KMeansMahout {

private static final String HDFS = "hdfs://10.20.0.10:9000";

public static void main(String[] args) throws Exception {

String localFile = "D:\\GaoLei\\clustering\_demo\\data.dat";

// mahout输出至HDFS的目录

String outputPath = HDFS + "/user/hdfs/kmeans/output";

// mahout的输入目录

String inputPath = HDFS + "/user/hdfs/kmeans/input/";

// canopy算法的t1

double t1 = 2;

// canopy算法的t2

double t2 = 1;

// 收敛阀值

double convergenceDelta = 0.5;

// 最大迭代次数

int maxIterations = 10;

Path output = new Path(outputPath);

Path input = new Path(inputPath);

Configuration conf = new Configuration();

HdfsUtil hdfs = new HdfsUtil(HDFS, conf);

hdfs.rmr(inputPath);

hdfs.mkdirs(inputPath);

hdfs.copyFile(localFile, inputPath);

hdfs.ls(inputPath);

// 每次执行聚类前，删除掉上一次的输出目录

HadoopUtil.delete(conf, output);

// 执行聚类

run(conf, input, output, new EuclideanDistanceMeasure(), t1, t2, convergenceDelta, maxIterations);

}

private static void run(Configuration conf, Path input, Path output,

EuclideanDistanceMeasure euclideanDistanceMeasure, double t1, double t2, double convergenceDelta, int maxIterations) throws Exception {

Path directoryContainingConvertedInput = new Path(output, "data");

System.out.println("Preparing Input");

// 将输入文件序列化，并选取RandomAccessSparseVector作为保存向量的数据结构

InputDriver.runJob(input, directoryContainingConvertedInput, "org.apache.mahout.math.RandomAccessSparseVector");

System.out.println("Running Canopy to get initial clusters");

// 保存canopy的目录

Path canopyOutput = new Path(output, "canopies");

// 执行Canopy聚类

CanopyDriver.run(conf, directoryContainingConvertedInput, canopyOutput, euclideanDistanceMeasure, t1, t2, false, 0.0, false);

System.out.println("Running KMeans");

// 执行k-means聚类，并使用canopy目录

KMeansDriver.run(conf, directoryContainingConvertedInput,

new Path(canopyOutput, Cluster.INITIAL\_CLUSTERS\_DIR + "-final"), output, convergenceDelta, maxIterations, true, 0.0, false);

System.out.println("run clusterdumper");

// 将聚类的结果输出至HDFS

ClusterDumper clusterDumper = new ClusterDumper(new Path(output, "clusters-\*-final"),new Path(output, "clusteredPoints"));

clusterDumper.printClusters(null);

}

}

* + 解析Mahout输出

执行完聚类后，需要从Mahout的输出目录下提取出所要的信息。在HDFS上的Mahout输出目录中可以找到clusteredPoints目录，该目录保存了聚类的最后结果，也就是关联到聚簇中心的向量信息，由于文件经过序列化，所以还需要解析clusteredPoints目录下的part-r-00000文件，得到聚类结果：

public class ClusterOutput {

private static final String HDFS = "hdfs://10.20.0.10:9000";

public static void main(String[] args) {

try {

// 需要被解析的mahout的输出文件

String clusterOutputPath = "/user/hdfs/kmeans/output";

// 解析后的聚类结果，将输出至本地磁盘

String resultPath = "D:\\tmp\\result.txt";

BufferedWriter bw;

Configuration conf = new Configuration();

conf.set("fs.default.name", HDFS);

FileSystem fs = FileSystem.get(conf);

SequenceFile.Reader reader = null;

reader = new SequenceFile.Reader(fs, new Path(clusterOutputPath + "/clusteredPoints/part-m-00000"), conf);

bw = new BufferedWriter(new FileWriter(new File(resultPath)));

// key为聚簇中心ID

IntWritable key = new IntWritable();

WeightedPropertyVectorWritable value = new WeightedPropertyVectorWritable ();

while (reader.next(key, value)) {

// 得到向量

Vector vector =value.getVector();

String vectorValue = "";

// 将向量各个维度拼接成一行，用\t分隔

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

if (i == vector.size() - 1) {

vectorValue += vector.get(i);

} else {

vectorValue += vector.get(i) + "\t";

}

}

bw.write(key.toString() + "\t" + vectorValue + "\n\n");

}

bw.flush();

reader.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

* + 得到最终结果

将程序导出为usercluster.jar，在导出jar时，需要将mahout-core-xxx.jar一并导出。

在Hive新建一个表cluster\_result，用于保存聚类结果，建表语句如下：

create table cluster\_result(

clusterId INT,

subTotal DOUBLE,

ordersCount DOUBLE,

sessionCount DOUBLE

) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

在user\_cluster.py文件中创建一个函数cluster\_output，用于执行前面的过程，将结果保存到Hive的cluster\_result表中。

def cluster\_output():

clusterOutputPath = "/user/hadoop/clusterOutput"

t1 = "100"

t2 = "10"

convergenceDelta = "0.5"

maxIterations = "10"

# 执行聚类

shell = HADOOP\_PATH + "hadoop jar " + PROJECT\_LIB\_DIR + "usercluster.jar com.clustering.KMeansMahout " + clusterOutputPath \

+ " /user/hive/warehouse/cluster\_input " + t1 + " " + t2 + " " + convergenceDelta + " " + maxIterations

os.system(shell)

# 解析聚类结果文件并输出至本地

shell = HADOOP\_PATH + "hadoop jar " + PROJECT\_LIB\_DIR + "usercluster.jar com.clustering.ClusterOutput " + clusterOutputPath \

+ " " + PROJECT\_TMP\_DIR + "result"

os.system(shell)

# 将本地的结果文件加载到Hive的cluster\_result表中

hql = "load data local inpath '" + PROJECT\_TMP\_DIR + "result overwrite into table cluster\_result"

HiveUtil.execute\_shell(hql)

在cluster\_result表中的结果如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0.3 | 0.23 | 0.32 |
| 0 | 0.17 | 0.13 | 0.42 |
| 0 | 0.12 | 0.13 | 0.12 |
| 1 | 0.2 | 0.1 | 0.2 |
| 1 | 0.7 | 0.3 | 0.4 |
| 1 | 0.5 | 0.6 | 0.9 |
| 2 | 0.2 | 0.422 | 0.32 |
| 2 | 0.123 | 0.542 | 0.42 |
| 2 | 0.323 | 0.242 | 0.122 |

从结果可以看到每一个向量分属于自己的聚簇中心，但是并没有体现用户信息，接下来我们使用cluster\_result和user\_dimension两个表做JOIN操作，将结果保存到Hive中的一张新表final\_result中，建表语句如下：

create table final\_result(

userId STRING,

clusterId INT,

subTotal DOUBLE,

ordersCount DOUBLE,

sessionCount DOUBLE

) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

在user\_cluster.py文件中添加一个函数 get\_finalresult()，用于执行HQL语句，对cluster\_result和user\_dimension两个表做JOIN，将结果保存到final\_result中：

def get\_finalresult():

hql = "insert overwrite table final\_result select t2.customerId,t1.\* from " \

"(select clusterId,subTotal,ordersCount,sessionCount from cluster\_result group by clusterId,subTotal,ordersCount,sessionCount) t1 " \

"join (select customerId,(subTotal-avg\_subTotal)/std\_subTotal subTotal,(ordersCount-avg\_ordersCount)/std\_ordersCount ordersCount," \

"(sessionCount-avg\_sessionCount)/std\_sessionCount sessionCount from user\_dimension " \

"join (select std(subTotal) std\_subTotal,std(ordersCount) std\_ordersCount,std(sessionCount) std\_sessionCount from user\_dimension) t1 on 1=1 " \

"join (select avg(subTotal) avg\_subTotal,avg(ordersCount) avg\_ordersCount,avg(sessionCount) avg\_sessionCount from user\_dimension) t2 on 1=1) t2 " \

"on t1.subTotal=t2.subTotal and t1.ordersCount=t2.ordersCount and t1.sessionCount=t2.sessionCount"

HiveUtil.execute\_shell(hql)

整合代码：

import sys

from com.ml.cluster.dimension\_and\_normalization import prepare\_normaliz

from com.ml.cluster.user\_cluster import cluster\_output, get\_finalresult

if \_\_name\_\_ == '\_\_main\_\_':

start = sys.argv[1]

end = sys.argv[2]

# 准备数据并做数据归一化

prepare\_normaliz(start, end)

# 聚类并输出

cluster\_output()

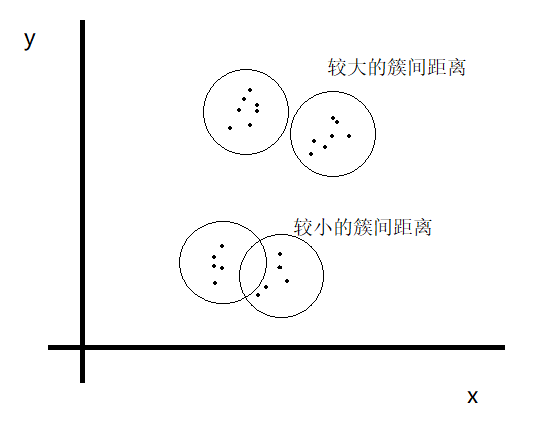
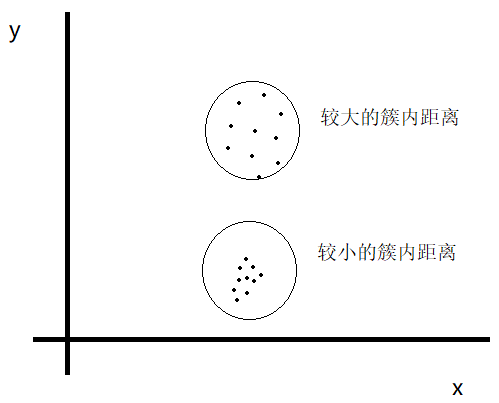
# 得到聚类结果

get\_finalresult()

到这里，购书用户聚类模块开发完成。

* 评估聚类结果
  + 簇间距离和簇内距离

可以通过簇间距离和簇内距离来评估聚类质量，簇间距离是指两个簇中心之间的距离，而簇内距离是指簇内部成员之间的距离。好的聚类效果通常是有着较大的簇间距离和较小的簇内距离，如下图所示：

* + 计算平均簇间距离

public class InterClusterDistances {

private static final String HDFS = "hdfs://10.20.0.10:9000";

public static void main(String[] args) throws Exception {

String inputFile = HDFS + "/user/hdfs/kmeans/output";

System.out.println("聚类结果文件地址：" + inputFile);

Configuration conf = new Configuration();

Path path = new Path(inputFile + "/clusters-2-final/part-r-00000");

System.out.println("Input Path：" + path);

FileSystem fs = FileSystem.get(path.toUri(), conf);

List<Cluster> clusters = new ArrayList<Cluster>();

SequenceFile.Reader reader = new SequenceFile.Reader(fs, path, conf);

Writable key = (Writable) reader.getKeyClass().newInstance();

ClusterWritable value = (ClusterWritable) reader.getValueClass().newInstance();

while (reader.next(key, value)) {

Cluster cluster = value.getValue();

clusters.add(cluster);

value = (ClusterWritable) reader.getValueClass().newInstance();

}

System.out.println("Cluster In Total：" + clusters.size());

DistanceMeasure measure = new EuclideanDistanceMeasure();

double max = 0;

double min = Double.MAX\_VALUE;

double sum = 0;

int count = 0;

Set<Double> total = new HashSet<Double>();

// 如果聚类的个数大于1才开始计算

if (clusters.size() != 1 && clusters.size() != 0) {

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

for (int j = 0; j < clusters.size(); j++) {

double d = measure.distance(clusters.get(i).getCenter(), clusters.get(j).getCenter());

min = Math.min(d, min);

max = Math.max(d, max);

total.add(d);

sum += d;

count++;

}

}

System.out.println("Maximum Intercluster Distance:" + max);

System.out.println("Minimum Intercluster Distance:" + min);

System.out.println("Average Intercluster Distance:" + sum / count);

for (double d : total) {

System.out.print("[" + d + "] ");

}

} else if (clusters.size() == 1) {

System.out.println("只有一个类，无法判断聚类质量");

} else if (clusters.size() == 0) {

System.out.println("聚类失败");

}

}

}

# 推荐系统设计与实现

## 任务描述

**内 容**：完成推荐系统的设计与实现

**知识点**： 基于用户的协同过滤、基于项目的协同过滤、基于模型的协同过滤、使用Mahout实现分布式推荐系统

**重 点**： 使用Mahout实现基于Hadoop的推荐系统

**难 点**： 推荐系统的设计和优化，如何选择合适的算法

## 任务指导

1. 推荐系统原理

* 为什么需要推荐系统
* 推荐系统的应用
* 如何使用推荐系统

1. 推荐引擎和算法

* 什么是推荐引擎
* 推荐引擎的分类
* 推荐机制分析
* 推荐引擎的应用

1. 协同过滤算法：基于用户的协同过滤和基于物品的协同过滤

* 什么是协同过滤
* 协同过滤的核心机制
* 基于用户的协同过滤
* 基于物品的协同过滤

1. 基于Mahout实现的推荐算法

* Mahout推荐算法介绍
* 算法评判标准：召回率与准确率
* Recommender.java的API接口
* 测试程序：RecommenderTest.java
* 基于用户的协同过滤算法UserCF
* 基于物品的协同过滤算法ItemCF
* SlopeOne算法
* SVD推荐算法
* Mahout推荐算法总结

1. 使用Mahout分布式程序实现基于物品的协同过滤
2. 使用Mahout实现图书推荐系统引擎

## 任务实现

1. 推荐系统原理

* 为什么需要推荐系统

随着互联网行业的井喷式发展，获取信息的方式越来越多，人们从主动获取信息逐渐变成了被动接受信息，信息量也在以几何倍数式爆发增长。同时，垃圾信息越来越多，导致用户获取有价值信息的成本大大增加。由于信息的爆炸式增长，对信息获取的有效性，针对性的需求也就自然出现了，推荐系统应运而生。

* 如何使用推荐系统
* **冷启动问题：**

推荐系统需要数据作为支撑，但是在最开始的时候，是没有大量且有效的**用户行为数据**，这时候就会面临着“冷启动”的问题，没有用户行为数据，那就只能利用商品本身的内容数据。这就是推荐系统早期的做法，例如：给商品打上各种标签、根据商品名称中包含的关键字、推荐同一商家的不同商品、利用经验等。

* 用户行为数据

例如：小明早上9点打开了亚马逊，先是浏览了首页，点击了几个热销的西装链接，然后在搜索栏输入了nike篮球鞋，在浏览了8双球鞋后，看了一些购买者的评价，最终选定了air jordan的最新款。这就是一条典型的用户行为数据。

* **user-item用户偏好矩阵：**用户产生的这些行为是多样的，所以需要一定的加权算法来计算出用户对某一商品的偏好程序，形成user-item用户偏好矩阵。

|  |  |  |  |
| --- | --- | --- | --- |
| 用户偏好 | Huawei | Iphone | Xiaomi |
| 小强 | 0.9 | 0.6 | 0.9 |
| 小红 | 0.8 | 0.8 | 0.7 |

* **数据清理：**当数据量增大时，不可避免就会产生大量的无用数据（噪声数据），因此，拿到数据的第一步就是对数据做清理，其最核心的工作就是：**减噪**和**归一化**。**减噪**是为了清理用户在使用过程中产生的大量的噪音和用户误操作产生的数据。归一化是为了通过对不同行为进行加权形成合理的用户偏好矩阵。
* **数据降维：**我们可能会得到一个巨大的用户偏好矩阵，但是由于用户有过行为数据的物品数量可能是有限的，所以产生的矩阵可能是稀疏的，有效的数据很少。这时就需要对这个稀疏的矩阵进行缩小（**降维**），从而减少计算量。例如SVD奇异值分解就是解决这类问题的一种常用的降维算法。通过对矩阵相乘不断的拟合，参数调整，将原来巨大的稀疏矩阵，分解为不同的矩阵，使其相乘可以得到原来的矩阵。这样既可以减少计算量，又可以填充矩阵中空值的部分。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 用户偏好 | 白菜 | 菠菜 | 生菜 | 西瓜 | 苹果 | 牛仔裤 | 针织衫 |
| 小强 | 0.9 | 0.9 | 0.9 |  |  | 0.9 |  |
| 小红 | 0.9 |  | 0.9 | 0.9 | 0.9 |  | 0.9 |

|  |  |  |  |
| --- | --- | --- | --- |
| 用户偏好 | 蔬菜 | 水果 | 休闲服饰 |
| 小强 | 0.9 | 0.9 | 0.9 |
| 小红 | 0.9 | 0.9 | 0.9 |

* **协同过滤算法**

协同过滤算法可以通过用户行为数据计算出两个相似度矩阵：user-user相似度矩阵、item-item相似度矩阵。国为这两个相似度矩阵是通过对方计算出来的，所以称为协同过滤。计算相似度的方法通常是通过特征之间的距离得出的，常见的算法有：欧几里德距离、皮尔逊相关系数、Cosine相似度、Tanimoto系数等。

* **其他算法**

基于用户偏好矩阵，可以应用很多的机器学习算法，例如，LR逻辑回归、k-means算法、canopy算法等。以LR逻辑回归为例，可分为三个步骤：

* 提取特征值
* 通过用户偏好矩阵，不断拟合计算，得到每个特征值的权重
* 预测新用户对物品的喜好程序
* **用好推荐系统**
  + **过滤规则**

在一个完善的推荐系统中，还需要考虑到很多业务方面的因素。以音乐类推荐为例，周杰伦出了一张新专辑A，大部分年轻人都会去点击收听，这样会导致其他每一张专辑相似专辑中都会出现专辑A。这个时候，再给用户推荐这样的热门专辑就没有意义了。所以，过滤掉热门的物品，是推荐系统的常见做法之一。这样的规则还有很多，视不同的业务场景而定。

* + **推荐的多样性**

例如推荐音乐，如果完全按照用户行为数据进行推荐，就会使得推荐结果的候选集永远只在一个比较小的范围内：听小清新音乐的人，永远也不会被推荐摇滚乐。这是一个很复杂的问题。所以，在保证推荐结果准确的前提下，按照一定的策略，去逐渐拓宽推荐结果的范围，给予推荐结果一定的多样性。

* + **持续改进**

推荐系统具有高度复杂性，需要持续改进，可能会在同一时间内，上线不同的推荐算法，随着用户行为数据的不断累积，不断对算法进行优化和改进。

1. 推荐引擎和算法

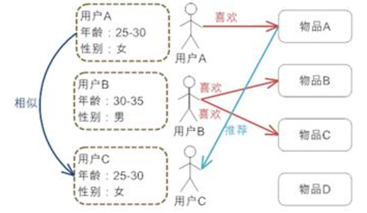
* 什么是推荐引擎

搜索引擎（Google，Bing，百度等）成为大家快速找到目标信息的最好途径。在用户对自己需求相对明确的时候，用搜索引擎很方便的通过关键字搜索很快的找到自己需要的信息。但搜索引擎并不能完全满足用户对信息发现的需求，很多情况下用户其实并不明确自己的需要，或者他们的需求很难用简单的关键字来表述。又或者他们需要更加符合他们个人口味和喜好的结果，因此出现了推荐系统，与搜索引擎对应，大家也习惯称它为**推荐引擎**。

**推荐引擎所需的数据源包括：**

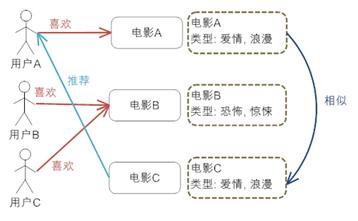
* 要推荐物品或内容的元数据
* 系统用户的基本信息
* 用户对物品或信息的偏好：
  + 显式用户反馈：例如用户对物品的评分或评论
  + 隐式用户反馈：例如用户购买了某物品或查看了某物品信息
* 推荐引擎的分类
* 根据推荐引擎的数据源分类
  + 基于内容的推荐
  + 基于协同过滤的推荐
  + 基于人口统计学的推荐
* 根据推荐模型的建立方式分类
  + 基于物品和用户本身的推荐
  + 基于关联规则的推荐
  + 基于模型的推荐
* 推荐机制分析
* 基于人口统计学的推荐

基于人口统计学的推荐机制（Demographic-based Recommendation）是一种最易于实现的推荐方法，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，然后将相似用户喜爱的其他物品推荐给当前用户。



* 基于内容的推荐

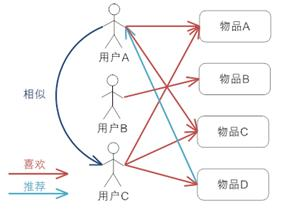
基于内容的推荐是在推荐引擎出现之初应用最为广泛的推荐机制，它的核心思想是根据推荐物品或内容的元数据，发现物品或者内容的相关性，然后基于用户以往的喜好记录，推荐给用户相似的物品。



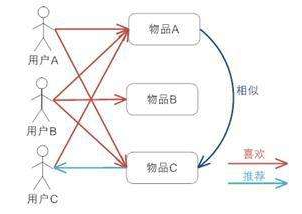
* 基于协同过滤的推荐

根据用户对物品或者信息的偏好，发现物品或者内容本身的相关性，或者是发现用户的相关性，然后再基于这些关联性进行推荐。基于协同过滤的推荐可以分为三个子类：基于用户的推荐（User-based Recommendation），基于项目的推荐（Item-based Recommendation）和基于模型的推荐（Model-based Recommendation）

* + 基于用户的推荐（根据用户历史偏好推断）



* + 基于项目的推荐（根据物品本身属性特征信息推断）



* + 基于模型的推荐

基于模型的协同过滤推荐就是基于样本的用户喜好信息，训练一个推荐模型，然后根据实时的用户喜好的信息进行预测，计算推荐。

* 推荐引擎的应用
  + 混合的推荐机制

在现行的 Web 站点上的推荐往往都不是单纯只采用了某一种推荐的机制和策略，他们往往是将多个方法混合在一起，从而达到更好的推荐效果。比较流行的组合方法有：

* + 加权的混合: 用线性公式将几种不同的推荐按照一定权重组合起来，具体权重的值需要在测试数据集上反复实验，从而达到最好的推荐效果。
  + 切换的混合：根据数据量，系统运行状况，用户和物品的数目等的不同，选择最为合适的推荐机制计算推荐。
  + 分区的混合：采用多种推荐机制，并将不同的推荐结果分不同的区显示给用户。
  + 分层的混合: 采用多种推荐机制，并将一个推荐机制的结果作为另一个的输入，从而综合各个推荐机制的优缺点，得到更加准确的推荐。
  + 应用领域
  + 电子商务：如Amazon
  + 社交网络：如豆瓣

1. 协同过滤算法：

* 协同过滤的核心机制

实现协同过滤，一般需要三个步骤：收集用户偏好、找到相似用户或物品、进行推荐计算。

* + 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 | 显示 | 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) | 显示 | 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 | 显示 | 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。 |

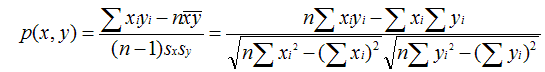
* + 收集了用户行为数据，还需要对数据进行一定的预处理：减噪和归一化。
  + 当已经对用户行为进行分析得到用户喜好后，可以根据用户喜好计算相似用户和物品，然后进行推荐，基于用户的 CF 和基于物品的 CF，都需要计算相似度。
  + 相似度的计算

关于相似度的计算，现有的几种基本方法都是基于向量（Vector）的，其实也就是计算两个向量的距离，距离越近相似度越大。常用的相似度计算方法：

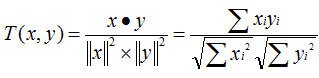
* + - 欧几里德距离



* + - 皮尔逊相关系数



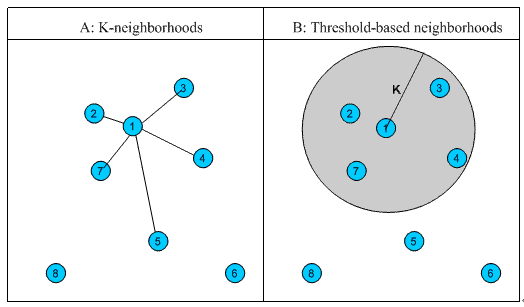
* + - 余弦相似度（Cosine Similarity）



* + - 相似邻居的计算

1）固定数量的邻居：K-neighborhoods 或者 Fix-size neighborhoods。

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

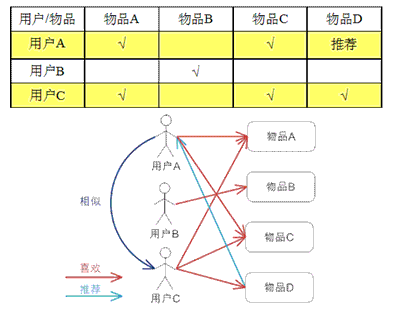


* 推荐计算

经过前期的计算已经得到了相邻用户和相邻物品，如何基于这些信息为用户进行推荐？前面已经简要介绍过基于协同过滤的推荐算法可以分为基于用户的 CF 和基于物品的 CF。

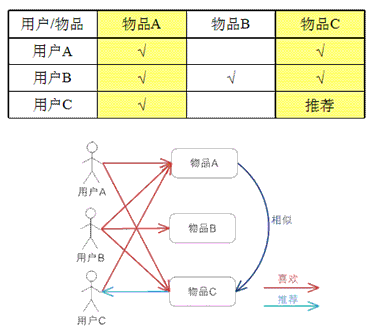
* + 基于用户的推荐（User CF）

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。如下图所示，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。



* + 基于物品的推荐（Item CF）

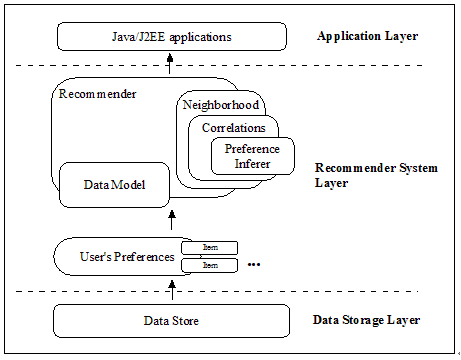
将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度，得到物品的相似物品后，根据用户历史的偏好预测当前用户还没有表示偏好的物品，计算得到一个排序的物品列表作为推荐。如下图所示，对于物品 A，根据所有用户的历史偏好，喜欢物品 A 的用户都喜欢物品 C，得出物品 A 和物品 C 比较相似，而用户 C 喜欢物品 A，那么可以推断出用户 C 可能也喜欢物品 C。



1. 基于Mahout实现的推荐算法

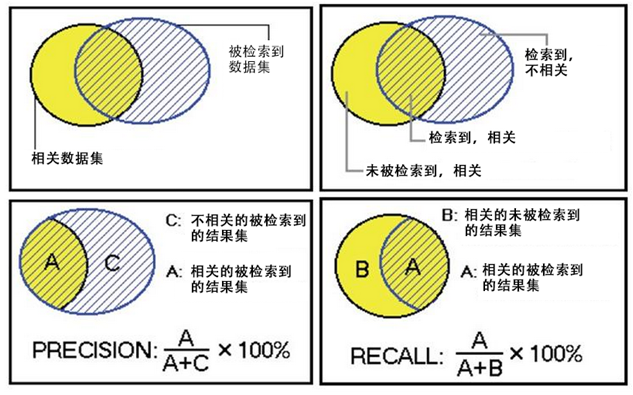
* Mahout推荐算法介绍

Apache Mahout 中提供的一个协同过滤算法的高效实现，它是一个基于 Java 实现的可扩展的，高效的推荐引擎。



* + Preference：基于协同过滤的推荐引擎的输入是用户的历史偏好信息，在 Mahout 里它被建模为 Preference（接口），一个 Preference 就是一个简单的三元组 <用户 ID, 物品 ID, 用户偏好>
  + DataModel：Mahout推荐引擎接受的输入是 DataModel，它是对用户偏好数据的压缩表示
  + Recommender：构建推荐器，Mahout 提供的协同过滤的推荐策略，最经典的三种：User CF, Item CF 和 Slope One
  + Mahout推荐算法，从数据处理能力上可以分为两类：**单机内存算法实现、基于Hadoop的分步布算法实现**
* 算法评判标准：召回率与准确率

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。

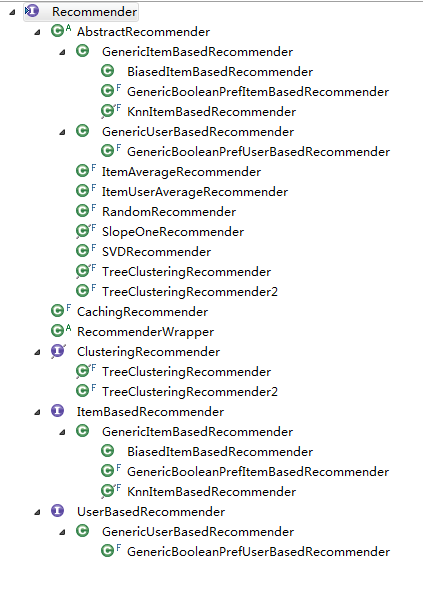


被检索到的越多越好，即A/(A+B)，“查全率”越大越好。

被检索到的，相关的越多越好，不相关的越少越好，即A/(A+C)，“查准率”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。

* Recommender.java的API接口



* + 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
* 基于用户的协同过滤算法UserCF
  + 基于用户的电影推荐实现

/\*\*

\* 基于用户的推荐

\*/

public class BaseUserRecommender {

public static void main(String[] args) throws Exception {

// 准备数据 这里是电影评分数据

String file = BaseUserRecommender.class.getClassLoader()

.getResource("data/ratings.dat").getFile();

// 将数据加载到内存中，GroupLensDataModel是针对开放电影评论数据的

DataModel dataModel = new GroupLensDataModel(new File(file));

// 计算相似度，相似度算法有很多种，欧几里得、皮尔逊等等。

UserSimilarity similarity = new PearsonCorrelationSimilarity(dataModel);

// 计算最近邻域，邻居有两种算法，基于固定数量的邻居和基于相似度的邻居，这里使用基于固定数量的邻居

UserNeighborhood userNeighborhood = new NearestNUserNeighborhood(100, similarity, dataModel);

// 构建推荐器，协同过滤推荐有多种，如：基于用户的和基于物品的，这里使用基于用户的协同过滤推荐

Recommender recommender = new GenericUserBasedRecommender(dataModel, userNeighborhood, similarity);

// 给用户ID等于5的用户推荐10部电影

List<RecommendedItem> recommendedItemList = recommender.recommend(5, 10);

// 打印推荐的结果

System.out.println("使用基于用户的协同过滤算法");

System.out.println("为用户5推荐10个商品");

for (RecommendedItem recommendedItem : recommendedItemList) {

System.out.println(recommendedItem);

}

}

}

* + 输出结果

使用基于物品的协同过滤算法

根据用户5当前浏览的商品2398，推荐10个相似的商品

RecommendedItem[item:326, value:6.8727612]

RecommendedItem[item:919, value:6.829988]

RecommendedItem[item:920, value:6.7896104]

RecommendedItem[item:527, value:6.227647]

RecommendedItem[item:926, value:6.2026863]

RecommendedItem[item:1235, value:6.1972055]

RecommendedItem[item:1096, value:6.1763406]

RecommendedItem[item:1104, value:6.1685]

RecommendedItem[item:538, value:6.1250587]

RecommendedItem[item:412, value:5.929688]

* 基于物品的协同过滤算法ItemCF
  + 基于用户的电影推荐实现

/\*\*

\* 基于物品的推荐

\*/

public class BaseItemRecommender {

public static void main(String[] args) {

try {

// 准备数据 这里是电影评分数据

String file = BaseUserRecommender.class.getClassLoader()

.getResource("data/ratings.dat").getFile();

// 将数据加载到内存中，GroupLensDataModel是针对开放电影评论数据的

DataModel dataModel = new GroupLensDataModel(file);

// 计算相似度，相似度算法有很多种，欧几里得、皮尔逊等等。

ItemSimilarity itemSimilarity = new PearsonCorrelationSimilarity(dataModel);

// 构建推荐器，协同过滤推荐有两种，分别是基于用户的和基于物品的，这里使用基于物品的协同过滤推荐

GenericItemBasedRecommender recommender = new GenericItemBasedRecommender(dataModel, itemSimilarity);

// 给用户ID等于5的用户推荐10个与2398相似的商品

List<RecommendedItem> recommendedItemList = recommender.recommendedBecause(5, 2398, 10);

// 打印推荐的结果

System.out.println("使用基于物品的协同过滤算法");

System.out.println("根据用户5当前浏览的商品2398，推荐10个相似的商品");

for (RecommendedItem recommendedItem : recommendedItemList) {

System.out.println(recommendedItem);

}

long start = System.currentTimeMillis();

recommendedItemList = recommender.recommendedBecause(5, 34, 10);

// 打印推荐的结果

System.out.println("使用基于物品的协同过滤算法");

System.out.println("根据用户5当前浏览的商品34，推荐10个相似的商品");

for (RecommendedItem recommendedItem : recommendedItemList) {

System.out.println(recommendedItem);

}

System.out.println(System.currentTimeMillis() - start);

} catch (Exception e) {

e.printStackTrace();

}

}

}

* + 输出结果

使用基于物品的协同过滤算法

根据用户5当前浏览的商品34，推荐10个相似的商品

RecommendedItem[item:919, value:6.6934185]

RecommendedItem[item:1225, value:6.2523694]

RecommendedItem[item:926, value:6.244406]

RecommendedItem[item:1104, value:6.2245407]

RecommendedItem[item:923, value:6.187429]

RecommendedItem[item:1219, value:6.1394296]

RecommendedItem[item:527, value:6.1108875]

RecommendedItem[item:920, value:6.078272]

RecommendedItem[item:608, value:6.0102806]

RecommendedItem[item:538, value:5.8557873]

15

* 评估推荐模型

public class ModelEvaluator {

public static void main(String[] args) throws Exception {

// 准备数据 这里是电影评分数据

String path=MyIRStatistics.class.getClassLoader()

.getResource("data/ratings.dat").getFile();

System.out.println(path);

// 准备数据 这里是电影评分数据

File file = new File(path);

// 将数据加载到内存中，GroupLensDataModel是针对开放电影评论数据的

DataModel dataModel = new GroupLensDataModel(file);

// 推荐评估，使用均方根

// RecommenderEvaluator evaluator = new RMSRecommenderEvaluator();

// 推荐评估，使用平均差值

RecommenderEvaluator evaluator = new AverageAbsoluteDifferenceRecommenderEvaluator();

RecommenderBuilder builder = new RecommenderBuilder() {

public Recommender buildRecommender(DataModel dataModel) throws TasteException {

UserSimilarity similarity = new PearsonCorrelationSimilarity(dataModel);

UserNeighborhood neighborhood = new NearestNUserNeighborhood(2, similarity, dataModel);

return new GenericUserBasedRecommender(dataModel, neighborhood, similarity);

}

};

// 用70%的数据用作训练，剩下的30%用来测试

double score = evaluator.evaluate(builder, null, dataModel, 0.7, 1.0);

// 最后得出的评估值越小，说明推荐结果越好

System.out.println(score);

}

}

* 获取推荐的准确率和召回率

public class MyIRStatistics {

public static void main(String[] args) throws Exception {

String path=MyIRStatistics.class.getClassLoader()

.getResource("data/ratings.dat").getFile();

System.out.println(path);

// 准备数据 这里是电影评分数据

File file = new File(path);

// 将数据加载到内存中，GroupLensDataModel是针对开放电影评论数据的

DataModel dataModel = new GroupLensDataModel(file);

RecommenderIRStatsEvaluator statsEvaluator = new GenericRecommenderIRStatsEvaluator();

RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {

public Recommender buildRecommender(DataModel model) throws TasteException {

UserSimilarity similarity = new PearsonCorrelationSimilarity(model);

UserNeighborhood neighborhood = new NearestNUserNeighborhood(4, similarity, model);

return new GenericUserBasedRecommender(model, neighborhood, similarity);

}

};

// 计算推荐4个结果时的查准率和召回率

// 使用评估器，并设定评估期的参数

// 4表示"precision and recall at 4"相当于推荐top4，然后在top-4的推荐上计算准确率和召回率

IRStatistics stats = statsEvaluator.evaluate(recommenderBuilder, null, dataModel, null, 4, GenericRecommenderIRStatsEvaluator.CHOOSE\_THRESHOLD, 1.0);

System.out.println(stats.getPrecision());

System.out.println(stats.getRecall());

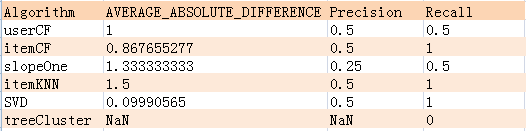
}

}

* Mahout推荐算法总结
  + 算法及适用场景



* + 算法评分结果

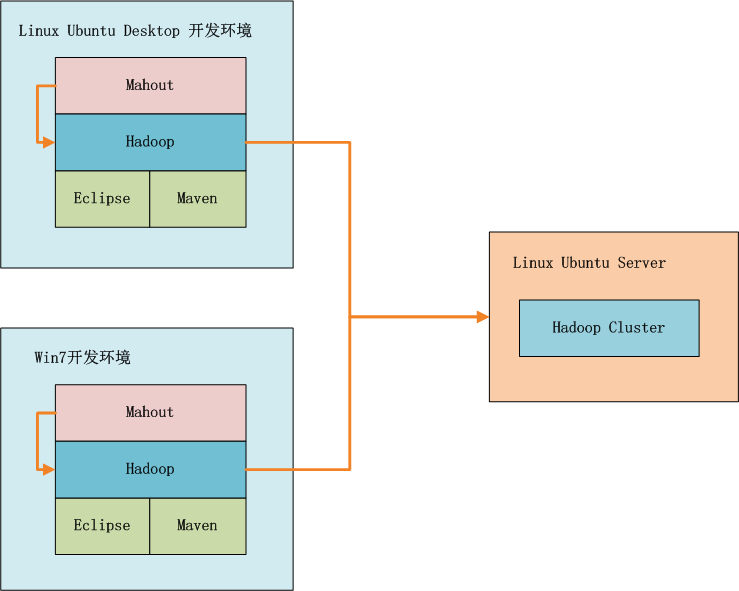


* + - Precision和Recall评分不能直接决定计算的结果一定是最好的
    - 在使用Mahout时我们并没有对参数进行调优
    - 数据量的大小和数据的分布，会直接影响算法的评分

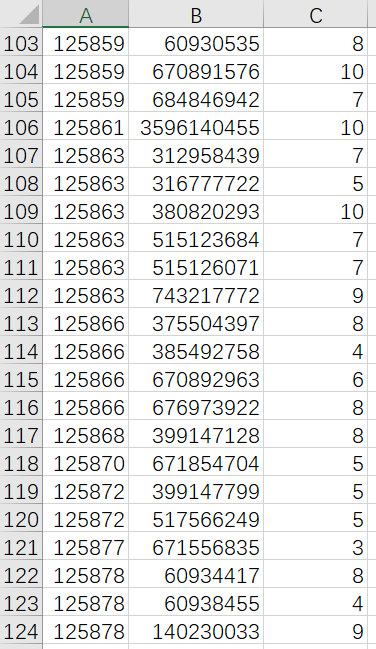
1. 使用Mahout分布式程序实现基于物品的协同过滤

Mahout分布式程序在运行过程中，会把基于MapReduce的算法程序包，自动发布到Hadoop集群环境中，在实际的生产环境中这种开发和运行模式使用的比较多。

* 环境介绍



* 准备数据 bx\_ratings.csv



* 一个HDFS操作的工具类，用Java API实现Hadoop的各种HDFS命令。

public class HdfsUtil {

private static final String HDFS = "hdfs://192.168.1.210:9000/";

public HdfsUtil(Configuration conf) {

this(HDFS, conf);

}

public HdfsUtil(String hdfs, Configuration conf) {

this.hdfsPath = hdfs;

this.conf = conf;

}

private String hdfsPath;

private Configuration conf;

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

JobConf conf = config();

HdfsUtil hdfs = new HdfsUtil(conf);

hdfs.copyFile("datafile/item.csv", "/tmp/new");

hdfs.ls("/tmp/new");

}

public static JobConf config(){

JobConf conf = new JobConf(HdfsUtil.class);

conf.setJobName("HdfsDAO");

conf.addResource("classpath:/hadoop/core-site.xml");

conf.addResource("classpath:/hadoop/hdfs-site.xml");

conf.addResource("classpath:/hadoop/mapred-site.xml");

return conf;

}

public void mkdirs(String folder) throws IOException {

Path path = new Path(folder);

FileSystem fs = FileSystem.get(URI.create(hdfsPath), conf);

if (!fs.exists(path)) {

fs.mkdirs(path);

System.out.println("Create: " + folder);

}

fs.close();

}

public void rmr(String folder) throws IOException {

Path path = new Path(folder);

FileSystem fs = FileSystem.get(URI.create(hdfsPath), conf);

fs.deleteOnExit(path);

System.out.println("Delete: " + folder);

fs.close();

}

public void ls(String folder) throws IOException {

Path path = new Path(folder);

FileSystem fs = FileSystem.get(URI.create(hdfsPath), conf);

FileStatus[] list = fs.listStatus(path);

System.out.println("ls: " + folder);

System.out.println("==========================================================");

for (FileStatus f : list) {

System.out.printf("name: %s, folder: %s, size: %d\n", f.getPath(), f.isDir(), f.getLen());

}

System.out.println("==========================================================");

fs.close();

}

public void createFile(String file, String content) throws IOException {

FileSystem fs = FileSystem.get(URI.create(hdfsPath), conf);

byte[] buff = content.getBytes();

FSDataOutputStream os = null;

try {

os = fs.create(new Path(file));

os.write(buff, 0, buff.length);

System.out.println("Create: " + file);

} finally {

if (os != null)

os.close();

}

fs.close();

}

public void copyFile(String local, String remote) throws IOException {

FileSystem fs = FileSystem.get(URI.create(hdfsPath), conf);

fs.copyFromLocalFile(new Path(local), new Path(remote));

System.out.println("copy from: " + local + " to " + remote);

fs.close();

}

public void download(String remote, String local) throws IOException {

Path path = new Path(remote);

FileSystem fs = FileSystem.get(URI.create(hdfsPath), conf);

fs.copyToLocalFile(path, new Path(local));

System.out.println("download: from" + remote + " to " + local);

fs.close();

}

public void cat(String remoteFile) throws IOException {

Path path = new Path(remoteFile);

FileSystem fs = FileSystem.get(URI.create(hdfsPath), conf);

FSDataInputStream fsdis = null;

System.out.println("cat: " + remoteFile);

try {

fsdis =fs.open(path);

IOUtils.copyBytes(fsdis, System.out, 4096, false);

} finally {

IOUtils.closeStream(fsdis);

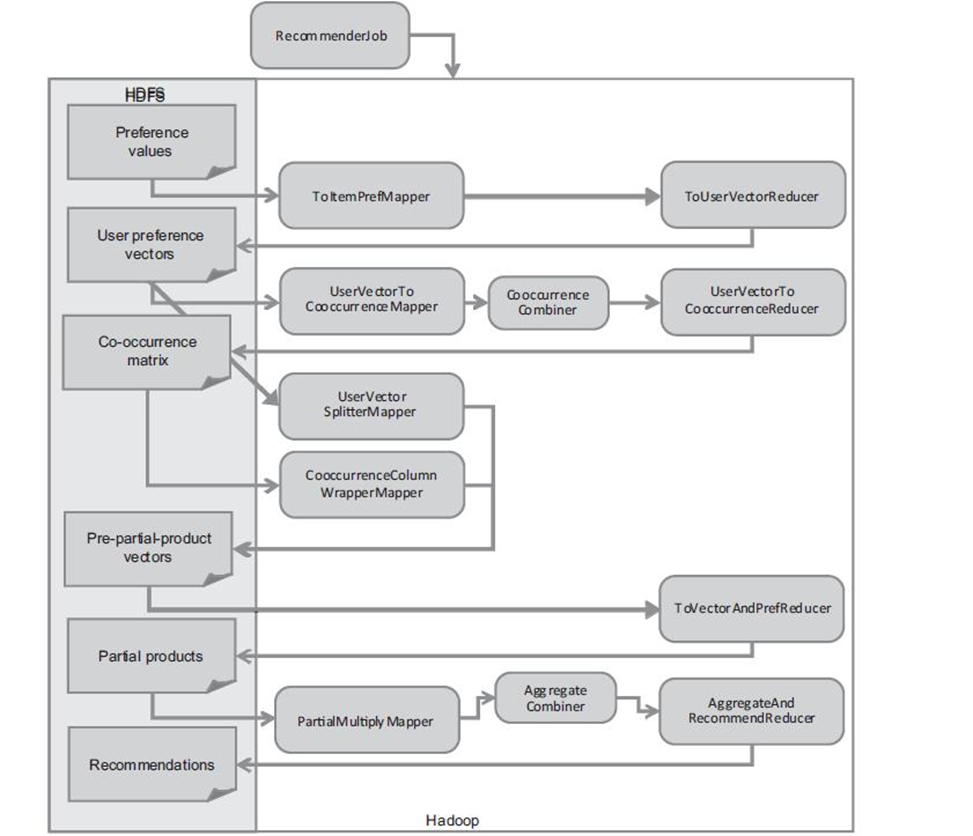
fs.close();

}

}

}

* Mahout分布式程序开发基于物品的协同过滤



实现代码：

public class ItemCFHadoop {

private static final String HDFS = "hdfs://10.20.0.10:9000";

public static void main(String[] args) throws Exception {

String localFile = ItemCFHadoop.class.getClassLoader()

.getResource("resources/data/bx\_ratings.csv")

.getFile();

String inPath = HDFS + "/user/hdfs/itemCF";

String inFile = inPath + "/item.csv";

String outPath = HDFS + "/user/hdfs/itemCF/result/";

String outFile = outPath + "/part-r-00000";

String tmpPath = HDFS + "/tmp/" + System.currentTimeMillis();

String userIdPath = HDFS + "/user/hdfs/itemCF/bx\_users.dat";

JobConf conf = config();

HdfsUtil hdfs = new HdfsUtil(HDFS, conf);

hdfs.rmr(inPath);

hdfs.mkdirs(inPath);

hdfs.copyFile(localFile, inPath);

hdfs.ls(inPath);

hdfs.cat(inFile);

StringBuilder sb = new StringBuilder();

sb.append("--input ").append(inPath);

sb.append("--output ").append(outPath);

sb.append("--booleanData true ");

sb.append("--numRecommendations 3 ");

sb.append("--similarityClassname org.apache.mahout.math.hadoop.similarity.cooccurrence.measures.EuclideanDistanceSimilarity ");

sb.append("--tempDir ").append(tmpPath);

sb.append("--usersFile ").append(userIdPath);

args = sb.toString().split(" ");

RecommenderJob job = new RecommenderJob();

job.setConf(conf);

job.run(args);

// hdfs.cat(inFile);

hdfs.cat(outFile);

}

public static JobConf config() {

JobConf conf = new JobConf(ItemCFHadoop.class);

conf.setJobName("ItemCFHadoop");

conf.addResource("classpath:/resources/core-site.xml");

conf.addResource("classpath:/resources/hdfs-site.xml");

conf.addResource("classpath:/resources/mapred-site.xml");

return conf;

}

}

RecommenderJob参数说明：

\* --input(path) : 存储用户偏好数据的目录，该目录下可以包含一个或多个存储用户偏好数据的文本文件；

\*

\* --output(path) :结算结果的输出目录;

\*

\* --numRecommendations (integer) : 为每个用户推荐的item数量，默认为10;

\*

\* --usersFile(path) : 指定一个包含了一个或多个存储userID的文件路径，仅为该路径下所有文件包含的userID做推荐计算(该选项可选);

\*

\* --itemsFile (path) :指定一个包含了一个或多个存储itemID的文件路径，仅为该路径下所有文件包含的itemID做推荐计算(该选项可选);

\*

\* --filterFile (path) :指定一个路径，该路径下的文件包含了[userID,itemID]值对，userID和itemID用逗号分隔。计算结果将不会为user推荐[userID,itemID] 值对中包含的item (该选项可选);

\*

\* --booleanData (boolean) :如果输入数据不包含偏好数值，则将该参数设置为true，默认为false;

\* --maxPrefsPerUser (integer) :在最后计算推荐结果的阶段，针对每一个user使用的偏好数据的最大数量，默认为10;

\* --minPrefsPerUser (integer) :在相似度计算中，忽略所有偏好数据量少于该值的用户，默认为1;

\* --maxSimilaritiesPerItem (integer) :针对每个item的相似度最大值，默认为100;

\* --maxPrefsPerUserInItemSimilarity (integer):在item相似度计算阶段，针对每个用户考虑的偏好数据最大数量，默认为1000;

\* --similarityClassname (classname) :向量相似度计算类 outputPathForSimilarityMatrix：SimilarityMatrix输出目录;

\* --randomSeed：随机种子;

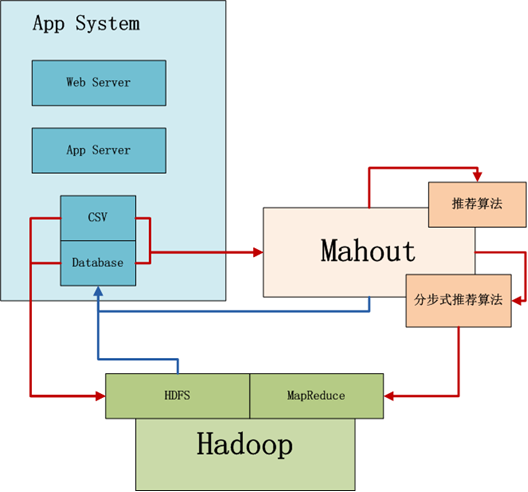
\* --sequencefileOutput ：序列文件输出路径;

\* --tempDir (path) :存储临时文件的目录，默认为当前用户的home目录下的temp目录;

\*

\* --threshold (double) : 忽略相似度低于该阀值的item对;

1. 使用Mahout实现图书推荐系统引擎



请学生仿照“购书用户聚类”模块，继续完成“图书推荐”模块的内容。