第一部分 基于大数据的推荐系统开发基础

# 数据形态与业务场景

## 数据形态

推荐系统一般包含输入层、模型层和输出层。输入层就是各种数据，主要包括用户的评分或者行为反馈数据、用户画像（性别、年龄、喜好等）和项目内容（文本、图像等描述或内容）数据、用户生成内容（社会化关系、标注、评论等辅助数据）。

### GroupLens电影数据

|  |  |
| --- | --- |
| 列名称 | 说明 |
| User ID | 整数类型 |
| Item ID | 整数类型 |
| 时间戳 | 行为发生的时间戳 |

MovieLen-20M 示例：

评分数据集 rating.csv

userId,movieId,rating,timestamp

1,2,3.5,1112486027

1,29,3.5,1112484676

1,32,3.5,1112484819

电影数据 movies.csv

movieId,title,genres

1,Toy Story (1995),Adventure|Animation|Children|Comedy|Fantasy

2,Jumanji (1995),Adventure|Children|Fantasy

3,Grumpier Old Men (1995),Comedy|Romance

用户标签数据 tags.csv

userId,movieId,tag,timestamp

18,4141,Mark Waters,1240597180

65,208,dark hero,1368150078

65,353,dark hero,1368150079

### 淘宝-电商数据

|  |  |
| --- | --- |
| 列名称 | 说明 |
| 用户ID | 整数类型，序列化后的用户ID |
| 商品ID | 整数类型，序列化后的商品ID |
| 商品类目ID | 整数类型，序列化后的商品所属类目ID |
| 行为类型 | 字符串，枚举类型，包括('pv', 'buy', 'cart', 'fav') |
| 时间戳 | 行为发生的时间戳 |

UserBehavior.csv 示例：

1,2268318,2520377,pv,1511544070

1,2333346,2520771,pv,1511561733

1,2576651,149192,pv,1511572885

### 线下商超数据

14列主要数据

### 数据反馈类型

根据上述数据类型的展示，可以把现有的数据根据反馈类型划分为，显示反馈类型和隐式反馈类型。显示反馈数据，用户对物品有明确的评分，直接反应用户的偏好（评分、喜欢/不喜欢）。但是实际场景中，用户对物品的评分数据是很难获取的，一般只有利用好隐式反馈数据，比如用户的各种行为数据（如点击、加购、收藏等等）。数据的类型不同，我们采取的处理方法和模型都会不一样。

## 标签体系的构建

用户和物品画像

图片包含 屏幕截图

描述已自动生成

标签系统和画像体系密不可分，构建画像绝大部分工作就是在计算标签。通过标签分类，包括事实标签和预测标签。事实标签。既定事实，从原始数据中提取。比如通过用户设置获取性别，通过实名认证获取生日，星座等信息。

模型标签。没有对应数据，需要定义规则，建立模型来计算得出标签实例。比如支付偏好度。预测标签。参考已有事实数据，来预测用户的行为或偏好。比如用户a的历史购物行为与群体A相似，使用协同过滤算法，预测用户a也会喜欢某件物品。

存储方式与结构：

采用文本数据库MongoDB和redis键值对数据库。MongoDB内存数据的应用主要在于对于单个用户的实时的查询，也是通过对spark数据梳理后的标签宽表进行数据格式转换(json格式)导入mongodb,前台应用可通过连接mongodb进行数据转换，从而进行单个标签的展现。(当然也可将数据转换为Redis中的key value形式，导入Redis集群)

# Spark大数据技术

## Spark基础

Spark是一个快速而通用的集群计算平台。

在速度方面，Spark 扩展了广泛使用的 MapReduce 计算模型，而且高效地支持更多计算模 式，包括交互式查询和流处理。在处理大规模数据集时，速度是非常重要的。速度快就意 味着我们可以进行交互式的数据操作，否则我们每次操作就需要等待数分钟甚至数小时。 Spark 的一个主要特点就是能够在内存中进行计算，因而更快。不过即使是必须在磁盘上 进行的复杂计算，Spark 依然比 MapReduce 更加高效。

总的来说，Spark 适用于各种各样原先需要多种不同的分布式平台的场景，包括批处理、 迭代算法、交互式查询、流处理。通过在一个统一的框架下支持这些不同的计算，Spark 使我们可以简单而低耗地把各种处理流程整合在一起。而这样的组合，在实际的数据分析 过程中是很有意义的。不仅如此，Spark 的这种特性还大大减轻了原先需要对各种平台分 别管理的负担。

Spark 所提供的接口非常丰富。除了提供基于 Python、Java、Scala 和 SQL 的简单易用的 API 以及内建的丰富的程序库以外，Spark 还能和其他大数据工具密切配合使用。例如， Spark 可以运行在 Hadoop 集群上，访问包括 Cassandra 在内的任意 Hadoop 数据源。

spark历史版本

spark存储层次

Spark 不仅可以将任何 Hadoop 分布式文件系统(HDFS)上的文件读取为分布式数据集， 也可以支持其他支持 Hadoop 接口的系统，比如本地文件、亚马逊 S3、Cassandra、Hive、 HBase 等。我们需要弄清楚的是，Hadoop 并非 Spark 的必要条件，Spark 支持任何实现 了 Hadoop 接口的存储系统。Spark 支持的 Hadoop 输入格式包括文本文件、SequenceFile、 Avro、Parquet 等。我们会在第 5 章讨论读取和存储时详细介绍如何与这些数据源进行交互。

=====> 可以说一下spark的快读数据存储格式，parquet等等！！！

spark下载与安装

使用 Spark 的第一步是下载和解压缩。我们先从下载预编译版本的 Spark 开始。访问 http:// spark.apache.org/downloads.html， 选 择 包 类 型 为“Pre-built for Hadoop 2.4 and later”( 为 Hadoop 2.4 及更新版本预编译的版本)，然后选择“Direct Download”直接下载。这样我们 就可以得到一个压缩的 TAR 文件，文件名为 spark-1.2.0-bin-hadoop2.4.tgz.

你不需要安装 Hadoop，不过如果你已经有了一个 Hadoop 集群或安装好的 HDFS，请下 载对应版本的 Spark。你可以在 http://spark.apache.org/downloads.html 里选择所需要的包 类型，这会导致下载得到的文件名略有不同。也可以选择从源代码直接编译。你可以从 GitHub 上下载最新代码，也可以在下载页面上选择包类型为“Source Code”(源代码)进 行下载。

RDD编程

下载好了 Spark 之后，我们要进行解压缩，然后看一看默认的 Spark 发行版中都有些什么。 打开终端，将工作路径转到下载的 Spark 压缩包所在的目录，然后解开压缩包。这样会创 建出一个和压缩包同名但是没了 .tgz 后缀的新文件夹。接下来我们就把工作路径转到这个 新目录下看看里面都有些什么。上面这些步骤可以用如下命令完成:

cd ~ tar -xf spark-1.2.0-bin-hadoop2.4.tgz

cd spark-2.4.0-bin-hadoop2.7

ls

在 tar 命令所在的那一行中，x 标记指定 tar 命令执行解压缩操作，f 标记则指定压缩包的 文件名。ls 命令列出了 Spark 目录中的内容。我们先来粗略地看一看 Spark 目录中的一些 比较重要的文件及目录的名字和作用。

Spark-shell

Spark 带有交互式的 shell，可以作即时数据分析。如果你使用过类似 R、Python、Scala 所 提供的 shell，或操作系统的 shell(例如 Bash 或者 Windows 中的命令提示符)，你也会对 Spark shell 感到很熟悉。然而和其他 shell 工具不一样的是，在其他 shell 工具中你只能使 用单机的硬盘和内存来操作数据，而 Spark shell 可用来与分布式存储在许多机器的内存或 者硬盘上的数据进行交互，并且处理过程的分发由 Spark 自动控制完成。

由于 Spark 能够在工作节点上把数据读取到内存中，所以许多分布式计算都可以在几秒钟 之内完成，哪怕是那种在十几个节点上处理 TB 级别的数据的计算。这就使得一般需要在 shell 中完成的那些交互式的即时探索性分析变得非常适合 Spark。Spark 提供 Python 以及 Scala 的增强版 shell，支持与集群的连接。

Spark构建Maven项目

## Spark mllib数据类型

Spark MLlib底层的向量和矩阵运算使用了Breeze库，Breeze库提提供了Vector和Matrix的实现以及相关的计算接口。但是在MLlib里面同时也提供了Vector和Linalg等的实现。在MLlib函数里的参数传递均使用MLlib自己的Vector，而且在函数内的矩阵计算又通过ToBreeze.ToDenseVector变成Breeze的形式进行计算。这样做的目的一是保持函数接口的稳定性，不会因为Breeze的变化而变化；另外一个比较重要的就是可以把Distributed Matrix作为一种Matrix的实现而被使用。

Spark在Vector和Matrix的基础上，实现了分布式矩阵类。分布式矩阵的数据分块或者分行存储，并且实现了矩阵的基本运算，能够使矩阵分布式计算，比如列统计、相似度、协方差、奇异值分解等。

### 本地向量（Local Vector）

本地向量是从0开始的下标和double类型的值，存储本地机器中，所以称为Local Vector。其支持两种数据形式：

Dense（稠密的向量）

Sparse（稀疏的向量）

比如一个向量[1.0,0.0,3.0]，用Dense表示为：[1.0,0.0,3.0]，用Sparse表示为：(3,[0,2],[1.0,3.0])，其中3为向量的长度，[0,2]表示元素[1.0,3.0]的位置，可见sparse形式下0.0是不存储的。

import org.apache.spark.mllib.linalg.{Vector, Vectors}  
  
// 创建稠密向量 (1.0, 0.0, 3.0).  
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)

//指定下标和数值，创建稀疏向量 (1.0, 0.0, 3.0)   
val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0))

//指定所有非零元素来创建稀疏向量 (1.0, 0.0, 3.0)，

val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))

### 标记点向量（Labeled Point）

labeled point由本地向量组成，既可以是dense向量，也可以是sparse向量。在mllib中常用于监督类算法，使用double类型来保存该类型的数据，因为也可以用于回归和分类算法。例如二分类，label可以是0（负例）或1（正例），对于多分类，label可以是0，1，2…

import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.mllib.regression.LabeledPoint

//用一个正标签和一个稠密特征向量创建标记点

val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))

// Create a labeled point with a negative label and a sparse feature vector.

//用一个负标签和一个稀疏特征向量创建标记点

val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0)))

### 本地矩阵（Local Matrix）

本地矩阵由行下标，列索引和double类型的值组成，存储在本地机器上，mllib支持稠密矩阵和稀疏矩阵，其存储是按照列进行存储的。

比如有一个稠密矩阵： 通过数组存储的形式为： [1.0, 3.0, 5.0, 2.0, 4.0, 6.0]，矩阵大小为[3，2]。

// 创建稠密矩阵 ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))

Import org.apache.spark.mllib.linalg.Matrices

val denseMatrix = Matrices.dense(3,2, Array(1.0,3.0,5.0,2.0,4.0,6.0))

println(s"denseMatrix is : $denseMatrix")

输出结果如下：

denseMatrix is :

1.0 2.0

3.0 4.0

5.0 6.0

// 创建稀疏矩阵 ((1.0, 0.0, 4.0), (0.0, 3.0, 5.0), (2.0, 0.0, 6.0))

val sparseMatrix = Matrices.sparse(3,3, Array(0,2,3,6),Array(0,2,1,0,1,2),Array(1,2,3,4,5,6))

println(s"sparseMatrix is : $sparseMatrix")

输出结果如下：

sparseMatrix is : 3 x 3 CSCMatrix

(0,0) 1.0

(2,0) 2.0

(1,1) 3.0

(0,2) 4.0

(1,2) 5.0

(2,2) 6.0

再比如一个稀疏矩阵 首先指定矩阵是3行3列，

数组: [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]，

行下标（Row Indices）: [0, 2, 1, 0, 1, 2]，

列偏移（column offsets）：[0, 2, 3, 6]

Array(0, 2, 1, 0, 1, 2)是指，第一个非零元素在第0行，第二个非零元素在第2行，第三个非零元素在第1行，以此类推。

Array(0, 2, 3, 6)标记列偏移和元素数量，列偏移0指的的是第一列的1.0从0偏移量开始，偏移1指第二列3.0从数组偏移2开始，3指4.0开始的偏移量。6是非零元素数量。

此处设计比较好，假设10000个元素分两列，不需要把每个元素所在列都标出来，只需要记录6个数字即可。Array(1, 2, 3, 4, 5)表示按顺序存储非零元素。

## Spark分布式矩阵

一个分布式矩阵由下标和double类型的数据组成，不过分布式的矩阵的下标不是int类型，而是long类型，数据保存在一个或多个rdd中，选择一个正确的格式去存储分布式矩阵是非常重要的。分布式矩阵转换成不同的格式需要一个全局的shuffle(global shuffle)，而全局shuffle的代价会非常高。到目前为止，Spark MLlib中已经实现了三种分布式矩阵。

最基本的分布式矩阵是RowMatrix，它是一个行式的分布式矩阵，没有行索引。比如一系列特征向量的集合。RowMatrix由一个RDD代表所有的行，每一行是一个本地向量。假设一个RowMatrix的列数不是特别巨大，那么一个简单的本地向量能够与driver进行联系，并且数据可以在单个节点上保存或使用。IndexedRowMatrix与RowMatrix类似但是有行索引，行索引可以用来区分行并且进行连接等操作。CoordinateMatrix是一个以协同列表（coordinate list)格式存储数据的分布式矩阵，数据以RDD形式存储。

Spark实现了分布式矩阵类DistributedMatrix，它是Spark中所有分布式矩阵的父类。目前实现了一些分布式矩阵子类包括： RowMatrix，IndexedRowMatrix，CoordinateMatrix和BlockMatrix。

### 行矩阵（RowMatrix）

分布式行矩阵就是把每行对应为一个RDD，将矩阵的每行分布式存储，矩阵的每行是一个本地向量。这和多变量统计的数据矩阵比较相似度。因为每行以一个本地向量表示，所以矩阵列的数量被限制在整数范围内，但是实际应用中的列数很小。

一个RowMatrix可以由一个RDD[Vector]的实例创建。因此我们可以计算统计信息或者进行分解。QR分解（QR decomposition）是A=QR，其中Q是一个矩阵，R是一个上三角矩阵。对sigular value decomposition(SVD和principal component analysis（PCA）,可以去参考降维的部分。

import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.mllib.linalg.distributed.RowMatrix

val arr = Array(Vectors.dense(1,0),Vectors.dense(0,1))

val rows = spark.sparkContext.parallelize(arr)

val mat: RowMatrix = new RowMatrix(rows)

val m = mat.numRows()

val n = mat.numCols()

val qrResult = mat.tallSkinnyQR(true)

println(s"m is: $m，n is $n，\nqrResult is :")

qrResult.Q.rows.foreach(println)

println()

qrResult.R.rowIter.foreach(println)

其输出为：

Row Matrix ...

m is: 2，n is 2，

qrResult is :

[1.0,0.0]

[0.0,1.0]

[1.0,0.0]

[0.0,1.0]

### 行索引矩阵（IndexedRowMatrix）

IndexRowMatrix和RowMatrix非常相似，区别是它带有一定意义的行索引。在RowMatrix中，rows的格式是RDD[Vector]；而在IndexedRowMatrix中，rows的格式是RDD[IndexRow]，其中的IndexedRow的格式是（index:Long, vector:Vector），相比RowMatrix多了一个Index索引信息。

一个IndexedMatrix可以从RDD[IndexedRow]实例创建，IndexedRow是（Long,Vector）的wrapper，而且这种矩阵可以转换成RowMatrix（通过去掉Index），其创建及使用方法类似于RowMatrix。

// IndexedRowMatrix

println("Indexed Row Matrix ...")

val arr2 = Array(

IndexedRow(0,Vectors.dense(1,0)),

IndexedRow(1,Vectors.dense(0,1))

)

val rows2: RDD[IndexedRow] = spark.sparkContext.parallelize(arr2)

val mat2 = new IndexedRowMatrix(rows2)

val m2 = mat2.numRows()

val n2 = mat2.numCols()

// 去掉行索引，转换成RowMatrix

val qrResult2 = mat2.toRowMatrix().tallSkinnyQR(true)

println(s"m2 is: $m2，n2 is $n2，\nqrResult2 is :")

qrResult2.Q.rows.foreach(println)

println()

qrResult2.R.rowIter.foreach(println)

输出为：

Indexed Row Matrix ...

m2 is: 2，n2 is 2，

qrResult2 is :

[1.0,0.0]

[0.0,1.0]

[1.0,0.0]

[0.0,1.0]

### 坐标矩阵（CoordinateMatrix）

坐标矩阵，每一项都是一个（i:Long, j:Long, value:Double）指示行列值的元组tuple。其中i是行坐标，j是列坐标，value是值。如果矩阵非常大并且稀疏，那么坐标矩阵一定是最好的选择。坐标矩阵是通过RDD[MatrixEntry]实例创建的，MatrixEntry为（Long, Long, Double）的形式。坐标矩阵可以转换为IndexedRowMatrix。

columnsSimilarities(threshold: Double)

threshold: 设置为0表示确定性的保证正确性。如果>0，结果相似度与上述成本与估计质量权衡相关。

返回一个n\*n稀疏的上三角矩阵中列与列之间的的余弦相似度。

// CoordinateMatrix

println("Coordinate Matrix ...")

val arr3 = Array(

MatrixEntry(0,0,1),

MatrixEntry(1,1,1)

)

val entries = spark.sparkContext.parallelize(arr3)

val mat3 = new CoordinateMatrix(entries)

val m3 = mat.numRows()

val n3 = mat.numCols()

val qrResult3 = mat3.toIndexedRowMatrix().toRowMatrix().tallSkinnyQR(true)

println(s"m3 is: $m3，n3 is $n3，\nqrResult3 is :")

qrResult3.Q.rows.foreach(println)

println()

qrResult3.R.rowIter.foreach(println)

输出为：

Coordinate Matrix ...

m3 is: 2，n3 is 2，

rowMat3 is :

[1.0,0.0]

[0.0,1.0]

[1.0,0.0]

[0.0,1.0]

### 分块矩阵（BlockMatrix）

分块矩阵（分段矩阵）就是将矩阵分割成较小的矩形矩阵，这些较小的矩阵就是区块。也就是说，以较小的矩阵组合成一个矩阵。分块矩阵的分割原则是以水平线和垂直线进行区分。在分块矩阵中，位于同一行（列）的每个子矩阵，都拥有相同的列数（行数）。将大的矩阵通过分块的方式划分，每个分块看做另一个矩阵的元素，再参与运算，可以让计算变得大幅简化。例如，有的大矩阵可以通过分块变成对角矩阵或者是三角矩阵等特殊形式的矩阵。

一个BlockMatrix是一个分布式的矩阵，由一个MatrixBlocks的RDD组成。MatrixBlock是一个三元组((Int, Int), Matrix),其中(Int, Int)是block的索引，Matrix是一个在指定位置上的维度为rowsPerBlock \* colsPerBlock的子矩阵。BlockMatrix支持与另一个BlockMatrix对象的add和multiply操作。BlockMatrix提供了一个帮助方法validate，这个方法可以用于检测该`BlockMatrix·是否正确。

如矩阵：可以分成的区块，如下：

分块后的矩阵可以写作:

分块矩阵BlockMatrix的存储是以单位块blocks分布式存储的，blocks的格式为RDD[((blockRowIndex,blockColIndex),sub-matrix)]。其中，blockRowIndex是block块行索引，blockColIndex是block块列索引，sub-matrix是block块矩阵。

可以通过IndexedRowMatrix或者CoordinateMatrix调用toBlockMatrix快速得到BlockMatrix对象。默认情况下toBlockMatrix方法会得到一个1024 x 1024的BlockMatrix。使用时可以通过手动传递维度值来设置维度，toBlockMatrix(rowsPerBlock, colsPerBlock)。

// BlockMatrix

println("Block Matrix ...")

val arr4 = Array(

MatrixEntry(0,0,1),

MatrixEntry(1,1,1)

)

val entries4: RDD[MatrixEntry] = spark.sparkContext.parallelize(arr4)

val coordMat: CoordinateMatrix = new CoordinateMatrix(entries4)

val matA: BlockMatrix = coordMat.toBlockMatrix().cache()

// 检测BlockMatrix格式是否正确，错误的话会抛出异常，正确的话无其他影响

matA.validate()

matA.blocks.foreach(println)

val m4 = matA.numRowBlocks

val n4 = matA.numColBlocks

println(s"m4 is: $m4，n4 is $n4")

// 计算A^T \* A.

val ata = matA.transpose.multiply(matA)

ata.blocks.foreach(println)

输出为：

Block Matrix ...

((0,0),2 x 2 CSCMatrix

(0,0) 1.0

(1,1) 1.0)

m4 is: 1，n4 is 1

((0,0),1.0 0.0

0.0 1.0 )

第二部分 推荐系统相关算法

# 大规模相似度计算

## 为什么计算相似度

如果你想找出不同用户和不同内容之间的相似之处，就必须进行相似度计算。通过相似度计算，我们可以找到跟我喜欢的物品相似的其他物品，对应的，也可以找到跟我志趣相投的其他用户（因为我喜欢的物品，他们也喜欢）。

那么如何定义相似度？如果我们将两个用户的相似度范围限定在0-1，0是完全不相似度，1为最相似。接下来，需要限定相似度的计算场景，比如在电影推荐中，用户之间的相似度指的是用户对于电影的口味或者是兴趣。比如两个用户都喜欢科幻电影，或者都喜欢某个明星参演的电影。但是，我们知道用户的兴趣是多元化的，喜欢科幻电影的人也可能会喜欢其他口味的电影类型。所以，加入其他补充信息，比如用户的年龄、性别等基础人口统计信息，得到的相似度结果会更加合理。

在数学上通常将相似度计算归结为距离计算，也就是将两个用户或者项目抽象为空间中的两个点，然后计算两个点之间的距离。距离越短，相似度越高。

## 相似度的度量方式

#### 杰卡尔德距离（Jaccard Distance）

1. 杰卡德相似系数  
   两个集合A和B的交集元素在A，B的并集中所占的比例，称为两个集合的杰卡德相似系数，用符号J(A,B)表示。

(2) 杰卡德距离  
 与杰卡德相似系数相反的概念是杰卡德距离(Jaccard distance)。杰卡德距离可用如下公式表示：

杰卡德距离用两个集合中不同元素占所有元素的比例来衡量两个集合的区分度。

假设有一个矩阵，横向代表item，包括row1、row2和row3，纵向代表user的购买情况，如果买过某个item，则对应数值为1，反之则为0。

user

item

通过python计算item两两之间的杰卡尔德相似度：

import scipy.spatial.distance as dist

v1=np.array([0,1,0,1,0,1])

v2=np.array([0,1,1,0,1,0])

v3=np.array([0,1,1,0,5,0])

matrix1=np.array([v1,v2])

matrix2=np.array([v1,v3])

print(matrix1)

print(matrix2)

ds1=dist.pdist(matrix1, 'jaccard')

ds2=dist.pdist(matrix2, 'jaccard')

print(ds1)

print(ds2)

其输出为：

[0.8]

[0,8]

上面代码中，v3相较于v2改变了第5个元素的评分，但是v1、v2和v1、v3之间的相似度保持不变。所以，杰卡尔德相似度只会判断是否购买，并不会计算具体评分。

#### 欧式距离（Euclidean Distance）

欧式距离是比较容易理解的一种距离计算方法，来源于空间中两点的距离公式。用向量表示两个n维向量与之间的欧式距离为：

假设一个矩阵，横向代表item，包括row1、row2和row3，纵向代表user的购买情况。矩阵中各个元素是user对item的具体评分，实际场景中评分是Double类型。

user

item

使用python计算row1和row2之间的欧式距离：

vector1 = np.array([0,2,0,4,0,1])

vector2 = np.array([0,1,1,4,1,0])

vector3 = np.array([0,1,3,4,1,0])

vector4 = np.array([1,0,1,4,1,0])

def EuclideanDistance(vector1, vector2):  
 #ds1=np.sqrt(np.sum(np.square(vector1-vector2)))   
 # np.linalg.norm 用于范数计算，默认是二范数，相当于平方和开根号  
 return 1.0/(1.0 + np.linalg.norm(vector1 - vector2))

ds1= EuclideanDistance(vector1, vector2)

ds2= EuclideanDistance(vector1, vector3)

ds3= EuclideanDistance(vector1, vector4)

**print**(ds1)

**print**(ds2)

print(ds3)

其输出为：

0.3333333333333333

0.2240092377397959

0.2612038749637414

#### 余弦相似度（Cosine）

几何中的夹角余弦可以用来衡量两个向量方向之间的差异，放在推荐系统中，可以用来衡量user或者item的行为向量之间的差异。两个n维样本点与用类似于夹角余弦的概念来衡量他们之间的相似度。

用python计算下列矩阵中row1和row2的相似度：

user

item

vector1 = np.array([0,2,0,2,0,2])

vector2 = np.array([0,1,1,4,1,0])

vector3 = np.array([0,1,3,4,1,0])

vector4 = np.array([0,4,0,4,0,4])

def cosine(vector1, vector2):

cos=np.dot(vector1,vector2)/(np.linalg.norm(vector1)\*(np.linalg.norm(vector2)))

return cos

cos1=cosine(vector1,vector2)

cos2=cosine(vector1,vector3)

cos3=cosine(vector1,vector4)

print(cos1)

print(cos2)

print(cos3)

输出为：

0.6622661785325219

0.5555555555555556

1.0000000000000002

#### 皮尔逊相关系数（Pearson Correlation Coefficient）

皮尔逊相关系数公式实际上就是在计算夹角余弦之前将两个向量减去各个样本的平均值，达到中心化的目的。从知友的回答可以明白，皮尔逊相关函数是余弦相似度在维度缺失上面的一种改进方法。

利用pytho-numpy实现皮尔逊相关系数计算：

def Pearson(dataA,dataB):

# 皮尔逊相关系数的取值范围(-1 ~ 1),0.5 + 0.5 \* result 归一化(0 ~ 1)

return 0.5 + 0.5 \* np.corrcoef(dataA,dataB,rowvar = 0)[0][1]

#### 修正余弦相似度（Adjusted Cosine）

为什么需要在余弦相似度的基础上使用修正余弦相似度。因为余弦相似度更多的是从方向上区分向量差异，而对绝对的数值不敏感，因此没法衡量每个维度上数值的差异，会导致一些问题。比如在推荐系统中，X和Y两个用户对两个内容的评分分别为（1,2）和（4,5），使用余弦相似度得到的结果是0.98，两者极为相似。但从评分上看X似乎不喜欢2这个内容，而Y则比较喜欢，余弦相似度对数值的不敏感导致了结果的误差，需要修正这种不合理性就出现了修正余弦相似度。

修正余弦的做法是在所有维度上的数值都减去一个均值，比如X和Y的评分均值为3，那么调整之后为（-2，-1）和（1，2），再使用余弦相似度计算得到-0.8，相似度为复制并且差异不小，不过显然更加符合现实情况。

利用python实现修正余弦相似度：

# 修正cosine 减去的是对item i打过分的每个user u，其打分的均值

data = np.mat([[1,2,3],[3,4,5]])

avg = np.mean(data[:,0]) # 下标0表示正在打分的用户

def AdjustedCosine(dataA,dataB,avg):

sumData = (dataA - avg) \* (dataB - avg).T # 若列为向量则为 dataA.T \* dataB

denom = np.linalg.norm(dataA - avg) \* np.linalg.norm(dataB - avg)

return 0.5 + 0.5 \* (sumData / denom)

print(AdjustedCosine(data[0,:],data[1,:],avg))

#### 汉明距离（Hamming Distance）

两个等长字符串s1与s2之间的汉明距离定义为将其中一个变为另外一个所需要作的最小替换次数。例如字符串“1111”与“1001”之间的汉明距离为2。

v1=np.array([1,1,0,1,0,1,0,0,1])

v2=np.array([0,1,1,0,0,0,1,1,1])

smstr=np.nonzero(v1-v2)

print(smstr) # 不为0 的元素的下标

sm= np.shape(smstr[0])[0]

print( sm )

#输出

#(array([0, 2, 3, 5, 6, 7]),)

#6

#### 曼哈顿距离（Manhatten Distance）

从名字就可以猜出这种距离的计算方法了。想象你在曼哈顿要从一个十字路口开车到另外一个十字路口，驾驶距离是两点间的直线距离吗？显然不是，除非你能穿越大楼。实际驾驶距离就是这个“曼哈顿距离”(L1范数)。而这也是曼哈顿距离名称的来源，曼哈顿距离也称为城市街区距离(City Block distance)。

两个n维向量与之间的曼哈顿距离为：

利用python代码计算曼哈顿距离：

vector1 = np.array([1,2,3])

vector2 = np.array([4,5,6])

op3=np.sum(np.abs(vector1-vector2))

op4=np.linalg.norm(vector1-vector2,ord=1)

#输出

#9

#9.0

#### 对比分析

简单来看，每一个相似度度量指标都不是完美的，比如杰卡尔德无法将具体评分带入公式中，只是判断有和没有。余弦相似度在两个向量方向一致时相似度为1，这很显然是不完全正确的，比如上文中[0,2,0,2,0,2]和[0,4,0,4,0,4]这两个item的评分向量虽然方向相同，夹角为0，但相似度为1 很显然不正确。

对比欧式距离的计算和夹角余弦的计算结果发现，欧式距离的相似度比较小，而夹角余弦相似度比较大，即夹角余弦更能反映两者之间的变动趋势，两者有很高的变化趋势相似度，而欧式距离较大是因为两者数值有很大的区别，即两者拥有很高的数值差异。

欧氏距离和余弦距离各自有不同的计算方式和衡量特征，因此它们适用于不同的数据分析模型。欧氏距离能够体现个体数值特征的绝对差异，所以更多的用于需要从维度的数值大小中体现差异的分析，如使用用户行为指标分析用户价值的相似度或差异。

余弦距离更多的是从方向上区分差异，而对绝对的数值不敏感，更多的用于使用用户对内容评分来区分兴趣的相似度和差异，同时修正了用户间可能存在的度量标准不统一的问题（因为余弦距离对绝对数值不敏感）。

## 基于评分的相似度计算工程实现

在推荐系统中，如果有user对item的评分，那就可以就行相似度计算了，其中包括item相似度与user相似度。但是，真实场景中用户点击或者购买某一商品时并不会对商品打分，这时可以通过一些方法进行构造式评分。

|  |  |
| --- | --- |
| **行为** | 评**分** |
| 点击 | 2.0 |
| 浏览详情 | 3.0 |
| 收藏 | 3.5 |
| 加购 | 4.0 |
| 下单 | 4.5 |

另外，有些推荐场景并没有用户点击、加购等复杂的操作行为，只有购买行为。这时可以用0和1表示用户的购买情况，0表示未买过，1表示买过。下图就是将购买行为转换成0和1 的矩阵。

|  |  |  |  |
| --- | --- | --- | --- |
| U\i | Item1 | Item2 | Item3 |
| User1 | 1 | 0 | 1 |
| User2 | 0 | 1 | 0 |
| User3 | 1 | 0 | 0 |

如果要得到标准的1-5的评分矩阵，需要将用户购买物品的次数转换成评分。数据归一化到[a,b]区间范围的方法：

（1）首先找到样本数据X的最小值Min及最大值Max  
（2）计算系数为：k=（b-a)/(Max-Min)  
（3）得到归一化到[a,b]区间的数据：norY=a+k(X-Min)

用python实现归一化操作为：  
def Normalization(x,a,b):  
 k = (b-a)/float(max(x)-min(x))  
 vec = [a+k\*(float(i)-min(x))/float(max(x)-min(x)) for i in x]  
 return [round(i, 2) for i in vec]

有上述方法转换得到的评分再取小数点后两位得到评分矩阵：

|  |  |  |  |
| --- | --- | --- | --- |
| User | Item | count | Rating |
| User1 | Item1 | 4 | 1.44 |
| User2 | Item2 | 10 | 1.15 |
| User3 | Item3 | 1 | 1.0 |

在MovieLens电影数据ml-20m中，计算所有元素的笛卡尔积，得到所有元素两两计算的相似度。这种常见的计算方式比较原始，也是非常耗时的。比如ratings.csv文件中Item去重后个数为2万多个，用户的维度有13万。如果计算Item的相似度，需要计算4亿次，时间复杂度较高。另外，由于用户user维度更大，两两计算相似度，那总共就要算次相似度，计算效率是个亟需解决的问题。存储方面，计算user相似度时，假设每次计算保存4个Byte，总共需要70G的数据量，空间存储也是问题。

### RDD分布式计算

当前代码中实现的相似度计算是通过笛卡尔积的方式计算RDD中两两匹配的相似度。RDD作为分布式计算数据集，可以很容易的实现多线程、多节点的进行相似度计算，比单机单进程的计算方式要快很多。一般业务场景下item数量要远远大于user数据量。所以，在大数据量下，即时单机可以容忍itemCF的计算时间，userCF则不行。最简单的RDD优化方式，可以通过广播变量的方式提高对比计算的效率。

### 向量化计算

为什么要进行向量化。向量化计算是机器学习领域常用的方法，也就是把循环遍历转换成向量计算，由此提高计算效率。在前面章节中介绍了Spark MLlib中的向量数据类型，我们用分布式矩阵CoordinateMatrix实现一个真实场景的Item相似度计算案例。

//利用Spark分布式矩阵计算Item相似度

def itemVectorSimilarity(spark: SparkSession,  
 trainData: Dataset[UserItem],  
 topN: Int) = {  
 //计算相似度矩阵  
 def standardCosine(matrix: CoordinateMatrix): RDD[MatrixEntry] = {  
 val similarity = matrix.toIndexedRowMatrix().columnSimilarities()  
 val sim = similarity.entries  
 sim  
 }  
  
 val sim = standardCosine(parseToMatrix(trainData)).map {  
 case MatrixEntry(user1, user2, sim) => (user1, (user2, sim))

}.groupByKey().

flatMapValues {x =>  
 val sim\_users = x.toList.sortBy(-\_.\_2).take(topN)  
 sim\_users  
 }.map(x => (x.\_1, x.\_2.\_1, x.\_2.\_2))  
}

### 近似最近邻方法ANN

ANN并非人工神经网络，而是近似邻居算法（Approximate Nearest Neighbor）的缩写。根据之前的方法，计算笛卡尔积属于蛮力（brute-force）搜索的方式，也就是在全空间进行搜索，为了加快查找的速度，几乎所有的ANN方法都是通过对全空间分割，将其分割成很多小的子空间，在搜索的时候，通过某种方式，快速锁定在某一（几）子空间，然后在该（几个）子空间里做遍历。可以看到，正是因为缩减了遍历的空间大小范围，从而使得ANN能够处理大规模数据的索引[[1]](#footnote-1)。关于ANN的算法实现有很多，比较典型的实现有Annoy、nmslib和SPTAG等等。几乎所有的ANN算法都是对于全空间的划分，大多数使用的是树模型。

1. LSH

局部敏感哈希（Locality-Sensitive Hashing）简称LSH是一种用于求解高维空间中的近似近邻搜索的方法。LSH保证了高维空间相近的点映射到低维空间相近的概率很高，所以这也是一个降维的过程。

1. K-Means Tree

K-Means Tree实际就是对数据做了多层K-means，每一层到当前的划分“叶子节点”包含样本数都小于1个。

1. K-D Tree

是对数据点在k维空间中划分的一种数据结构。k-d tree实际上是一种二叉树。

#### Annoy介绍：

Annoy是近似最近邻算法实现的一个 C++ 库，它是（Approximate Nearest Neighbors Oh Yeah)的缩写，其本身带有 Python接口。用于搜索空间中给定查询项目的的近似项目。它也创建了大型只读文件的数据结构，这些数据结构被mmap[[2]](#footnote-2)到内存中，以便许多进程可以共享相同的数据。

**安装**

安装很简单，只需通过pip安装：

pip install --user annopy

这行命令会从PyPI上拉下Annoy最新版本。

对于 C++版本，只需从github中克隆项目，并插入声明#include "annoylib.h“即可。

**背景**

还有一些其他库可以实现最近的邻域搜索。Annoy几乎是执行速度最快的那一个（下面会做一些性能对比），但实际上还有另一个使得 Annoy 与众不同的功能：它能够使用静态文件作为索引。这意味着您可以跨进程共享索引。Annoy 还会将创建索引与加载索引分离，以便您可以将索引作为文件传递并快速映射到内存中。Annoy 的另一个好处是，它尝试最小化内存占用，因此索引非常小。

这个功能相当好用，如果要查找最近邻，并且有许多 CPU，则只需生成一次索引。您还可以传递和分发静态文件，以用于生产环境、Hadoop 作业等。任何进程都能够将索引加载（mmap）到内存中，并能够立即进行查找。

该项目在 Spotify[[3]](#footnote-3)上使用并进行音乐推荐。在运行矩阵分解算法后，每个用户/项目都可以在 f 维空间中表示为向量。Annoy帮助系统搜索相似的user/item，这是一个超过百万维度的空间，因此内存使用是首要问题。

**功能摘要：**

1. 欧几里德距离，曼哈顿距离，余弦距离，汉明距离(Hamming Distance)或点（内）积距离。

2. 余弦距离相当于归一化向量的欧几里德距离= sqrt（2-2 \* cos（u，v））

3. 如果维度很小（比如小于<100维），效果会更好，但即使是1,000维也能表现出色

4. 内存使用量小

5. 允许您在多个进程之间共享内存

6. 索引创建与查找分开（特别是在创建树后，您无法添加更多项目）

7. Python支持2.7,3.6和3.7。

8. 如果数据量很大，可以不用放到内存中，在磁盘上构建索引再使用

利用python调用计算item相似度的案例：

from annoy import AnnoyIndex  
class AnnoyMovie:  
 def \_\_init\_\_(self, dataFile):  
 self.datafile = dataFile  
 self.data = self.loadData()  
 self.ann, self.userNum = self.addItem()  
  
 def loadData(self):  
 print("Load data...")  
 data = []  
 for line in open(self.datafile):  
 itemid, vector = line.strip("\n").split(",")  
 data.append((itemid, vector))  
 return data  
  
 def addItem(self):  
 userNum = len(self.data.pop()[1].split(" "))  
 print("user num", userNum)  
 ann = AnnoyIndex(userNum, 'angular')  
 for itemId, vector in self.data:  
 ann.add\_item(int(itemId), map(float,vector.split(" ")))  
 ann.build(1000)  
 return ann, userNum  
  
 def getNear(self, index, itemId, simNum):  
 print(index.get\_nns\_by\_item(itemId, simNum))  
  
 def save(self):  
 print("保存索引文件...")  
 self.ann.save('movie.ann')  
  
 def load(self):  
 print("加载索引文件...")  
 u = AnnoyIndex(self.userNum, 'angular')  
 u.load('movie.ann')  
 return u

注意：Annoy只接受整数作为Item的标识符。请注意，它将为max（id）+1个Item分配内存，因为它假设您的Item编号为0 ... n-1。

**部分python API接口：**

AnnoyIndex（f，metric）

返回一个新的读写索引并存储f维度的向量。相似度计算可以是“angular”（类似于cosine距离），“欧几里德”，“曼哈顿”，“汉明”或“点积”。

a.add\_item（i，v）

用向量v添加项i（任何非负整数）。注意它将为max（i）+1项分配内存。

a.build（n\_trees）

构建一个n\_trees树的森林。查询时，更多树提供更高的精度。调用build后，不能再添加任何项目。

a.save（fn，prefault = False）将索引保存到磁盘并加载它（参见下一个函数）。保存后，不能再添加任何项目。

a.load（fn，prefault = False）

从磁盘加载索引。如果prefault设置为True，它将预读取整个文件到内存中。默认值为False。

a.unload（）为卸载。

a.get\_nns\_by\_item（i，n，search\_k = -1，include\_distances = False）

返回n个最接近的项目。在查询期间，它将检查最多search\_k节点，如果没有提供，则默认为n\_trees \* n。 search\_k为您提供更好的准确性和速度之间的运行时权衡。如果将include\_distances设置为True，它将返回一个包含两个列表的2元素元组：第二个元素包含所有相应的距离。

get\_nns\_by\_vector（v，n，search\_k = -1，include\_distances = False）

相同但是按向量v查询。 a.get\_item\_vector（i）返回先前添加的项目i的向量。 a.get\_distance（i，j）返回项i和j之间的距离。

#### hnswlib介绍[[4]](#footnote-4)

在说hnslib之前先说nmslib。非度量空间库（nmslib）是一个高效的跨平台的相似性搜索库，也是用于评估相似性搜索方法的工具包。其核心库没有任何第三方依赖。Nmslib是一个可扩展的库，这意味着可以添加新的搜索方法和距离函数。Nmslib可以直接在C++和python中调用。另外，还可以构建一个查询服务器，可以用Java调用。

快速近似邻居搜索，Hnswlib与nmslib来自于同一作者，而hnswlib更快，占用内存更小。

明显优势：

1. 体积小，调用方便，C++11开发且没有其他任何依赖

2. 接口包含C++，python和R

3. 支持新增索引，同时支持元素删除

4. 可以自定义距离计算

5. 与nmslib相比，明显减少了内存占用，同时缩短了构建时间

安装：

pip install hnswlib

利用python调用hnswlib实现item相似度计算：

import hnswlib  
import numpy as np  
import sys  
  
class HnmslibMovie:  
 def \_\_init\_\_(self, dataFile):  
 self.datafile = dataFile  
 self.data, self.data\_labels = self.loadData()  
 self.ann = self.addItem()  
  
 #导入评分矩阵文件  
 def loadData(self):  
 print("Load data...")  
 data = []  
 data\_labels = []  
 for line in open(self.datafile):  
 index, vector = line.strip("\n").split(",")  
 rating\_list = vector.split(" ")  
 data\_labels.append(index)  
 data.append(rating\_list)  
 data = np.float32(data)  
 data\_labels = np.int32(data\_labels)  
 return data, data\_labels  
  
 def addItem(self):  
 self.dim = len(self.data[0])  
 self.num\_elements = len(self.data\_labels)  
 print("dimension num", self.dim)  
 #声明索引  
 p = hnswlib.Index(space='cosine', dim= self.dim) #space设置为l2, cosine 或 ip  
 #初始化索引，应事先知道最大数量的元素  
 p.init\_index(max\_elements= self.num\_elements, ef\_construction=200, M=16)  
 #插入元素（可以被调用多次）  
 p.add\_items(self.data, self.data\_labels)  
 #设置ef，控制召回  
 p.set\_ef(50) # ef设置必须要大于k  
 return p  
  
 def getNear(self, p, index, simNum):  
 #查询近邻，k=最近元素的个数（返回两个numpy数组）  
 labels, distances = p.knn\_query(self.data[np.where(self.data\_labels==index)[0][0]], k=simNum)  
 print("label:",labels, "distance: ", distances)  
 return labels, distances  
  
 def save(self):  
 print("保存索引文件...")  
 self.ann.save\_index('movie.ann')  
  
 def load(self):  
 print("加载索引文件...")  
 p = hnswlib.Index(space='cosine', dim=self.dim)  
 p.load\_index('movie.ann', max\_elements = self.num\_elements)  
 return p  
  
 def localQuery(self):  
 self.save()  
 p = self.load()  
 # 制定itemid 和 相似度的个数simNum  
 self.getNear(p, 1, 10)

#### Spark-MinHashLSH

Spark实现LSH（局部敏感哈希）如：BucketedRandomProjectionLSH 和 MinHash。

Spark-LSH实现Item相似度计算，Scala代码如下：

import org.apache.spark.ml.linalg.Vectors

import org.apache.spark.ml.linalg.{SparseVector, Vectors}  
import org.apache.spark.ml.feature.{BucketedRandomProjectionLSH, MinHashLSH, MinHashLSHModel}  
  
val user\_count = data\_rating.map(\_.\_1.toInt).max  
val data\_rating\_matrix = data\_rating.map (x => (x.\_2.toInt, (x.\_1.toInt, x.\_3))).  
 groupByKey().mapValues { x =>  
 val features = x.toList.map {  
 case (feature\_index, value) =>  
 (feature\_index, value)  
 }.sortBy(\_.\_1)  
 Vectors.sparse(user\_count - 1, features)  
}.sortByKey()  
val data\_rating\_df = spark.createDataFrame(data\_rating\_matrix).toDF("id", "features")

val mh = new MinHashLSH().  
 setNumHashTables(5).  
 setInputCol("features").  
 setOutputCol("hashes")  
  
val itemModel = mh.fit(data\_rating\_df)  
val transformed\_data\_rating = itemModel.transform(data\_rating\_df)  
transformed\_data\_rating.show(false)

val itemSimRDD = data\_rating\_matrix.collect().  
 map(x=> (x.\_1, itemModel.approxNearestNeighbors(data\_rating\_df, x.\_2, 10)))

itemSimRDD

LSH的具体参数：

Tables： 该参数越大，可以降低误报率（false negative rate）；参数越小，可以提高运行的效率。

### 实验对比与最佳方式

对比上述几种相似度计算方式，由于user比item相似度计算更加耗时，且userID无法进行抽样检查。这里选择item进行相似度对比。

分别用单机和集群两种资源配置方式进行测试。

单机配置：CPU-24cores, Memory-64G

Spark-Local限制资源：CPU-20cores，memory-30G

集群配置：Master+8worders, 单个worker：CPU-24cores, Memory-64G

Spark-Cluster限制资源：8workers，CPU-160cores，Executor-19G

**Annoy运行命令：**

140机器： /xiaoi/josh/recommend/ann

运行ml\_1m数据：

time python ann\_movie\_vector.py

运行CRM数据:

time python ann\_crm\_vector.py

**hnswlib运行命令**：

140机器： /xiaoi/josh/recommend/ann

运行ml\_1m数据：

time python hnswlib\_movie.py

运行CRM数据:

time python hnswlib\_crm.py

**SPTAG运行命令：**

159机器：/xiaoi/josh/ann/movie

构建索引：

time python sptagMovie.py

查询：

1. 修改读取文件query\_input.txt
2. 执行

python sptag\_load.py

注：

1.“/”表示资源不支持该运算，或者是无法运行。

2.“\_decom”后缀表示数据降维后再进行计算，sklearn - PCA(n\_componets取0.99)

3. 运行时间再具体测试时会稍微上下浮动。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data | 计算方式 | 内存占用 | 构建索引 | 加载并查询一次(Top10) | 加载并查询1千次(Top10) | 加载并查询1万次(Top10) |
| ml-1m | Annoy | 0.2G | 6.104s | 6.15s | 19.631s | / |
| hnswlib | 1.1G | 10.211s | 10.809s | 64.57s | / |
| SPTAG | 0.2G | 15m23 | 0.4s | 6.909s | / |
| CRM | Annoy | 24G | 12m12s | 13m25s | 58m25s | >2h |
| hnswlib | >60G | / | / | / | / |
| hnswlib\_decom | >60G | / | / | / | / |
| SPTAG | >60G | / | / | / | / |

快速ANN算法工具库实验小结：

**内存方面：**

对比Annoy、hnswlib和SPTAG三者在不同数据量下的内存和时间消耗.我们发现，Annoy并不会因为数据量过大而把内存全部占尽，内存最高占用24G，如果不够则把部分数据存储在磁盘当中。相反，hnswlib在大数据量下因为内存不够用，程序被迫中止。STPTAG单机试验下占用内存超过最高上限，报错MemoryError。

**速度方面：**

由于Hnswlib和SPTAG单机状态下，无法实现CRM数据（维度20000\*80000）的计算。只能通过数据ml\_1m(维度3700\*6000)小数据集下进行对比。

**效果方面：**

Annoy和Hnswlib将参数都选择cosine余弦距离。抽取了几个例子，两者相似度结果保持一致。

但是SPTAG的相似度计算结果与Annoy和Hnswlib大范围不同。比如：

同时计算Index1，玩具总动员的相似Top10：

**Annoy的结果:**

[1, 3114, 1265, 588, 2355, 1270, 34, 1196, 1580, 356]

转为Movie-name:

Toy Story (1995),Animation|Children's|Comedy

Toy Story 2 (1999),Animation|Children's|Comedy

Groundhog Day (1993),Comedy|Romance

Aladdin (1992),Animation|Children's|Comedy|Musical

Bug's Life, A (1998)

Back to the Future (1985),Comedy|Sci-Fi

Babe (1995),Children's|Comedy|Drama

Star Wars: Episode V - The Empire Strikes Back (1980),Action|Adventure|Drama|Sci-Fi|War

Men in Black (1997),Action|Adventure|Comedy|Sci-Fi

Forrest Gump (1994),Comedy|Romance|War

**Hnswlib的结果：**

[1, 3114, 1265, 588, 2355, 1270, 34, 1196, 1580, 356]

**SPTAG结果：**

[1, 2899, 1174, 575, 2163, 1179, 34, 1107, 347, 2375]

转为Movie-name:

Toy Story (1995),Animation|Children's|Comedy

Gulliver's Travels (1939),Adventure|Animation|Children's

Grosse Fatigue (1994),Comedy

Little Rascals, The (1994)

Attack of the Killer Tomatoes! (1980),Comedy|Horror

Grifters, The (1990)

Babe (1995),Children's|Comedy|Drama

Loser (1991),Comedy

Bitter Moon (1992),Drama

Money Pit, The (1986)

可以看出Annoy的结果与Hnswlib高度一致，而SPTAP的结果与前两者大相径庭。

**Spark Item相似度计算方法时间对比：**

|  |  |  |  |
| --- | --- | --- | --- |
| Data | 计算方式 | Local | Cluster |
| CRM | RDD | 60min | 9min |
| Spark-Vectorization | 25.39min | 2.6min |
| Spark-LSH | 14.11min | 3.8min |

当前最佳方案： ANN + TopN。考虑到实际场景中需要将所有Item的相似度TopN进行召回，以便于下一步离线排序使用。但是，不再需要计算所有Item的笛卡尔积，然后再排序选取TopN。只要查找单个Item的最近邻相似TopN即可。时间复杂度大大降低。由此，真实推荐中大规模高维度的相似度计算问题，可以通过近似最近邻方式快速找到相似（接近）的用户和物品。

影响相似度计算时间的因素：

1. User和Item的数量
2. 用户产生的行为数量，这直接影响数据集的稀疏度
3. 硬件资源，理论上将节点越多，计算速度越快

### 常见问题与解决方法

**稀疏向量的压缩**

实际生产环境中，生成的用户向量都是非常稀疏的，构成的矩阵也是非常稀疏的，直白来说就是很多值都是0，有一些存储稀疏矩阵的格式。比如 CSR 或者 COO。

CSR：CSR是一个整体编码方式，由三部分构成，数值、列号和行偏移。

COO：COO每个元素用一个三元组表示（行号，列号，数值），只存储有值的元素，缺失值不存储。

这些存储格式，在常见的框架中都已经实现，比如 Python-scipy和Spark-MLlib。

**矩阵降维**

首先，相似度计算本身并不需要计算全量数据，比如超过3个月的行为数据就不应该作为计算数据，因为用户的行为习惯是变化的，太老的数据已经不能反映客户当前偏好和习惯。

协同过滤在实际场景中itemCF效果会好于UserCF，一般情况下只需要计算Item相似度即可。User相似计算一般会在计算用户特征相似时使用，找出新客户最相似的老客户就可以推荐老客户购买过的商品，这一方法可以解决用户冷启动的问题。

## 基于内容的相似度计算的工程实现

# 经典推荐算法

## Memory\_Based 算法

### 基于内容的推荐（content-based recommendation）

通过用户历史感兴趣的信息，抽象信息内容共性，根据内容共性推荐其他信息。

比如，如何通过基于内容的推荐，来对求职者A进行职位推荐？

答：简要步骤如下

找到用户A历史感兴趣的职位集合

找到职位集合的具化内容

抽象具化内容的共性内容

由这些共性内容查找其他职位，并实施推荐

典型步骤：收集用户历史数据，历史数据的feature化，根据历史数据的feature去找相似的item

关键点： 历史数据的feature化， item的feature化，相似度计算

典型应用：新闻网站，头条网站 （Common for recommending **text-based products** (webpages, usenet news messages）

也可以是model-based，The user model can also be a **classifier** based on whatever technique (Neural Networks, Naïve Bayes...)

### 基于用户profile的推荐

通过用户的profile， 结合item的feature (包含item的profile)，进行推荐

用户的profile：人口基本特征/兴趣/偏好/用户等级，忠诚度/keyword， 可以是用户填写的， 也可以是计算/推测出来的

特点：不需要用户的历史数据，可以作为冷启动的解决方案之一。例如某些阅读应用，在用户第一次登录的时候，会要求用户选择感兴趣的domain

典型步骤：各种途径获得user profile，计算user profile 与 item feature之间的相似度，推荐

与“基于内容的推荐”的区分和联系

基于用户的内容，可以生产出profile的内容（比如关键词，tag，分类等等）

有时并不严格区分 （content-based user profile）

问题：可信度不高，因为即便是年龄、性别等属性都相同的用户，也很有可能对物品有截然不同的偏好；可解释性不高

还有一种广泛使用的更有实际意义的user profile：考虑该User打过分的所有Item，将这些Item的Item Profile的每一项分别进行加权平均，得到一个综合的Profile，作为该用户的User Profile。这种User Profile的优点是非常容易计算其与Item之间的相似度，同时比较准确地描述了该用户在Item上的偏好，巧妙地避开了用户私人信息这一很难获得的数据，具有保护隐私的能力，进一步，如果加入时间因素，还可以研究用户在Item上偏好的变化等等，因此受到广泛应用

客户特征的向量空间模型

图片包含 文字

描述已自动生成

将上文抽取的用户属性和标签进行离散化，重要特征设置相应的权重，然后将每个客户的特征向量化。通过计算新客户和已成交客户之间的相似度和筛选条件，在所有新客户中挖掘出高度潜在客户。

### 用户行为权重模型

很多开卡和未开卡的客户在 “掌上生活”APP中有大量的点击、浏览、查询等行为，而开卡用户的刷卡消费记录、分期情况，都可以反映客户对于保险产品的潜在意向。通过这些行为的建模匹配其对于保险产品的兴趣度，从而进行有针对性的营销。

图片包含 墙壁

描述已自动生成

比如，根据用户的具体行为与保险的相关度进行评分，然后构建所有用户关于行为的评分矩阵，计算每个客户对于保险的喜好度。如果加入保单的维度，就可以算出每个客户对于每个保险产品的喜好度。

### 协同过滤

图片包含 屏幕截图

描述已自动生成

基于用户的协同过滤（user-based CF，给本客户推荐与其情况相似的其他客户普遍持有的保险产品）：比如上图用户1和用户3都购买了悦享康健-重大疾病险和泰康家倍-家庭医疗险，可以根据用户评分计算出两者相似度较高，所以把用户3买过的诺享无忧-意外险推荐给用户1。

基于用户的协同过滤也可以通过neighborhood\_based算法实现：设定相应的阈值找出跟用户1 距离相近的其他用户，他们购买过的保险具有群体相似性。

基于物品的协同过滤（item-based CF）：计算物品之间的行为或特征相似度，再进行推荐。比如，给购买过住院医疗险的客户推荐重大疾病险，因为这两种保险同现频率很高。此处的物品指的就是保险产品。

## Model-Based算法

### 隐因子模型（Latent Fator Model）

图片包含 屏幕截图

描述已自动生成

假设一些用户，或者几个产品属于一个类型或者说是因子（factor），同时假设用户对于特定的因子有一定的喜好度。可以抽象出一个隐形因子空间，然后把用户和物品分别投影到这个空间上，来直接寻找用户-物品的喜好度。这个模型被称为隐因子模型。

LFM与基于邻域的方法对比：

图片包含 屏幕截图

描述已自动生成

最小二乘法

最小二乘法（ALS，alternating least squares）是一种求解矩阵分解问题的优化方法。它功能强大而且容易实现并行化。所以它比较适合像Spark这样的的大数据平台。

ALS的实现原理是迭代式求解一系列最小二乘法回归问题。在每一次迭代时，固定用户因子矩阵或物品因子矩阵中的一个，然后用固定的这个矩阵以及评分数据来更新另一个矩阵。之后，被更新的矩阵被固定住，再更新另一个矩阵。如此迭代，直到模型收敛。

由于q和p未知，目标函数是非凸函数。但如果假设其中一个当前值固定，则该优化问题可解。ALS便通过交替上述假设来迭代求解。

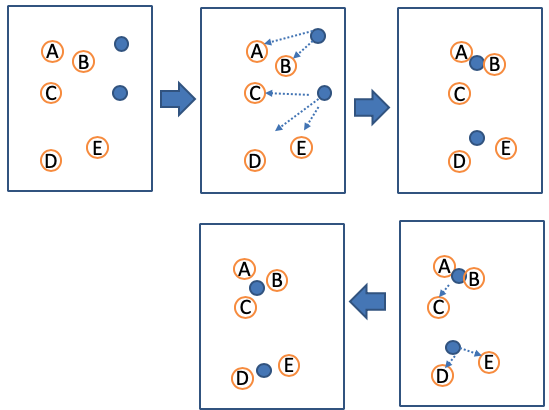
* rank:对应 ALS 模型中的因子个数，也就是在低阶近似矩阵中的隐含特征个数。因子个 数一般越多越好。但它也会直接影响模型训练和保存时所需的内存开销，尤其是在用户 和物品很多的时候。因此实践中该参数常作为训练效果与系统开销之间的调节参数。通 常，其合理取值范围为 10~200。
* iterations:对应运行时的迭代次数。ALS 能确保每次迭代都能降低评级矩阵的重建误 差，但一般经少数次迭代后 ALS 模型便已能收敛为一个比较合理的好模型。这样，大部 分情况下都没必要迭代太多次(10 次左右一般就挺好)。
* numBlocks:对应用户和物品将分为将并行计算的块数(默认为 10)。该数取决于集群的 节点数和输入分块的方式。
* regParam:对应 ALS 的正则化参数(默认为 1.0)。常数 λ 被称为正则化参数。本质上， 当用户或物品矩阵的规模过大时，它会减小矩阵的因子。这对数值稳定很重要，而且总 会引入某种正则化方法。
* implicitPrefs:对应标识矩阵中的值是隐式反馈值还是显式反馈值，两种分别会用 ALS 隐式反馈衍生模型(ALS-WR)和显示反馈衍生模型来建模。默认为 False，即显式 反馈值。
* alpha:这是 ALS 隐式反馈模型的一个参数，它确定了反馈对应的基准可信度，默认为 1.0。
* nongeative:指定是否使用非负约束条件。默认为 false。

### 关联规则

通过Apriori，FPGrowth算法挖掘多个保险产品之间的频繁项集。也就是，根据用户成单记录计算两两产品之间的相关度，比如：买了重大疾病类产品，推荐意外险，提高加保率。

### K-means聚类算法

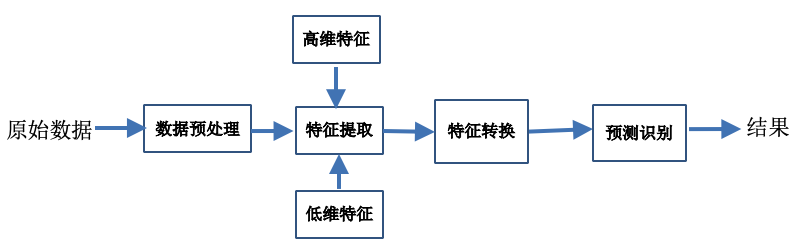
聚类算法在保险客户分群中起到了重要作用，K-means聚类算法首先会随机确定k个中心位置，位于空间中代表聚类中心的点，然后将各个数据项分配给最临近的中心点。待分配完之后，聚类中心就会移到分配给该聚类的所有节点的平均位置处，然后整个分配过程重新开始。这一过程会一直重复下去，直到分配过程不再产生变化位置。



上图模拟k=2的情况下如何进行聚类，第一个框中，两个中心点的位置是随机选择的，第二个框中把每个数据项分配给了距离最近的中心点，下面的框中进一步分配，最终结果是A、B、C在一个聚类中，D、E在另一个聚类中。

# 基于CTR预估的推荐

经典的推荐算法在解决推荐问题上解释力比较强，比如根据User-Based推荐解释可以说“与您相似的客户90%也买了这个商品”。但是，经典推荐算也存在很多问题，比如大数据量计算即为耗时，无法做到实时反馈。另外，每次计算都需要计算全量数据，学法做到增量学习和即时预测。基于点击率预估（CTR）的推荐算法是将推荐问题转化成分类或者排序问题。CTR预估不仅可以预测哪些用户会点击或者购买商品，也可以将候选池的商品进行排序并形成推荐列表。



## GBDT+LR实现高潜客户识别和Item排序

客户特征的二分类建模，训练数据包括所有已电销的通话记录。预测新客户可能购买保险的概率，选择购买意向高的用户为最终客户。

图片包含 标牌, 文字

描述已自动生成

该模型的实现比较注重客户的分组情况，需要提前通过聚类的方式计算潜在的分析特征，比如客户可能同属于同一种保险购买群体。训练数据特征化之后，结合树形结构的集成算法GBDT得到多个基模型的结果，这里我们不会通过残差的加减来减少偏差，而是加入LR线性模型得到每个分类树的概率。

## FM因式分解机

因式分解机（Factorization Machine）是一种基于矩阵分解的机器学习模型，可以轻松因对高度稀疏的数据场景。FM允许更多的特征工程，可以将所有的特征表示成embedding vector，构造二阶关系，同时，在联合特征的权重捕捉上，不再采用简单的单权重，而是采用向量内积来表示，这样就能从原始信息中捕捉更多的信息。FM可以再线性时间复杂度内完成非线性的计算任务，有利于拟合训练集中的非线性关系。

图片包含 物体

描述已自动生成

如上损失函数，最后加入了组合特征。一般线性模型在处理组合特征时泛化能力弱，但FM增加了xi,xj这种组合特征，并且vi,vj就是特征学习之后的embedding向量的内积，这就可以代表权重。注重有效的权重赋值方法增加了FM的泛化能力较强。

在2014年两个CTR预测比赛中获胜的方法都使用了一种叫 Field-aware Factorization Machines (FFM)的方法，它是因式分解机(Factorization Machines)的变体。FFM尝试通过学习每个特征交互对的潜在因素来为特征交互建模。这个算法可以在LibFFM框架中实现并且已被许多参赛者使用。LibFFM对大型数据集的并行处理和内存使用非常有效。

## 强化学习 - Clustering Of Bandit

Clustering Of Bandit简称CLUB，其在推荐系统领域主要解决EE问题，尤其对冷启动阶段效果较好。我们假设，用户成交了一份保单，其反馈回报和用户相关的feature成线性关系。由此，用客户和产品的特征预估回报和其置信区间，然后选择置信区间上界最大的产品作为推荐结果。

图片包含 屏幕截图

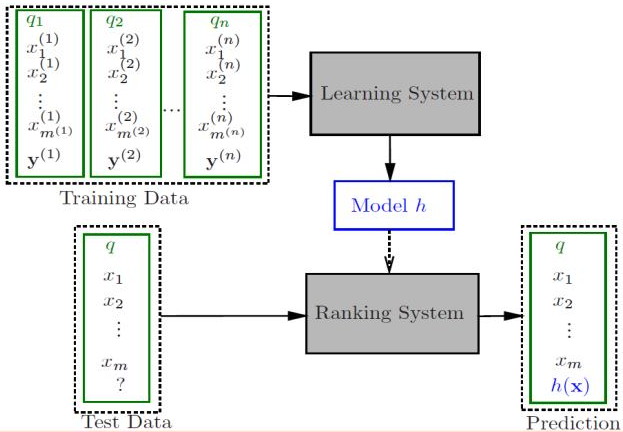
描述已自动生成

CLUB跟其它Bandit算法最大的区别在于它引入了用户特征的计算，让整个算法收敛更快。

## 排序学习（Learning To Rank）

在推荐系统领域，根据业务数据可以划分为显式(Explicit)和隐式(Implicit)反馈类型。基于近邻(neighbourhood-based)和矩阵分解(MF)的方法预测显式反馈的评分效果更好，而对于隐式数据效果一般。

每个Item的打分 +… 根据已有的成单训练数据作为标记对象，根据分类或回归的方式获得a,b,c,d参数的最优组合，然后通过这个打分函数进行相关性判断。



对于信用卡和保单数据，我们需要换一个思路：“预测用户是否会对Item产生某种隐式反馈”的分类问题。推荐列表注重排序，并不关心单个item预测的评分多少。Pair-wise Raiking就是对同一个User，直接考虑任意两个Item的顺序谁在前谁在后。

## 算法总结

综上所述，Memory\_Based算法和监督学习算法存在很多差异，从模型角度来说，Memory\_Based算法不需要生成模型来预测客户-物品之间匹配的概率，而是通过计算相似度或者隐因子关系，最终生成推荐结果；而监督学习算法需要通过训练集学习生成模型，再通过预测集的特征向量来预测。

从业务角度看，Memory\_Based算法适合于显式反馈的业务场景，比如豆瓣电影和亚马逊图书，这些场景需要用户给自己接触的物品打分，而打分就直接对应用户的喜好度。但是在其他业务场景，比如金融和保险产品中，用户购买产品并不能直接得到相应的喜好度和评分，用户可能是为了别人买保险，或者只是生活需要。我们把后一种场景得到数据为隐式反馈数据。所以，通过隐式反馈数据最终只能推荐用户适合的物品，而非用户最终喜爱的物品。

第三部分 案例

# 电商案例

## 环境准备与软件安装

### mongodb 4.0 安装部署

下载 mongodb-linux-x86\_64-2.4.9.tgz

tar -xvf mongodb-linux-x86\_64-2.4.9.tgz

mv mongodb-linux-x86\_64-2.4.9 mongodb

vim /home/hadoop/.bashrc

添加：

export MONGODB\_HOME=/opt/mongodb

export PATH=$PATH:$MONGODB\_HOME/bin

命令： source /home/hadoop/.bashrc

mkdir -p /mnt/disk0/mongodb/data/db

mkdir -p /mnt/disk0/mongodb/logs/

打开 /opt/mongodb/bin/mongodb.conf

添加一下配置：

#数据文件目录

dbpath = /mnt/disk0/mongodb/data/db

#日志目录

logpath = /mnt/disk0/mongodb/logs/mongodb.log

#端口

port = 27017 #端口

#以守护进程运行

fork = true

nohttpinterface = true

**运行mongodb：**

/opt/mongodb/bin/mongod -f /opt/mongodb/bin/mongodb.conf

或者命令行登录方式:

mongo 122.226.240.140:27017/recommender -u josh -p dis2019

测试mongodb服务：

mongo

创建管理员：

db.createUser({user: 'root', pwd: 'dis2019', roles: ['root']})

给recommender设置权限：

db.createUser({user:'admin',pwd:'dis2019',roles: [{role:'readWrite',db:'recommender'}]}))

验证：

db.auth('admin', 'dis2019')

普通db用户：

db.createUser({user: "admin", pwd: "dis2019", roles: [{ role: "dbOwner", db: "recommender" }]})

db.auth('admin', 'dis2019')

重启mongo，登录 ：

mongo -u admin -p 'dis2019' --authenticationDatabase "recommender"

或者，命令行登录方式

mongo 122.226.240.140:27017/recommender -u admin -p dis2019

查看collection：

show collections

创建collection：

db.createCollection("u\_attr")

### **kafka安装（同时启动zookeeper）**：

最新版本下载

<http://mirrors.tuna.tsinghua.edu.cn/apache/kafka/2.1.0/kafka_2.11-2.1.0.tgz>

**修改配置文件**

server.properties：

标识当前server在集群中id，从0开始，每台server必须集群内唯一

broker.id=0

port=9092

host.name=davinci01

log.dirs最好匹配多个磁盘上，以便提高读写效率

log.dirs=/opt/kafka/logs

这些目录必须已经手动创建好了

Zookeeper配置，2181端口为默认端口

zookeeper.connect=localhost:2181

日志中每个segment的大小为1G

log.segment.bytes=1073741824

其他的不变

scp到各节点

在/opt/kafka目录下建立logs目录以存放server日志

启动：

nohup ./bin/zookeeper-server-start.sh config/zookeeper.properties > logs/zookeeper.log &

nohup ./bin/kafka-server-start.sh config/server.properties > logs/server.log &

创建topic

bin/kafka-topics.sh -zookeeper jh-a0-010:2181 -topic u\_sim -replication-factor 1 -partitions 2 -create

查看topic状态

/opt/kafka/bin/kafka-topics.sh -zookeeper jh-a0-010:2181/kafka -list

创建producer

bin/kafka-console-producer.sh --broker-list jh-a0-010:9092 --topic u\_sim

在producer测输入：

hello kafka

在consumer观察：

bin/kafka-console-consumer.sh --bootstrap-server jh-a0-010:9092 --topic u\_sim --from-beginning

**kafka单机测试：**

注意： 单机配置中host为 localhost, zookeeper设置为2181/kakfa，以后需要重新改

/opt/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic u\_sim

/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic u\_sim

### redis安装

wget http://download.redis.io/releases/redis-5.0.3.tar.gz

$ tar xzf redis-5.0.3.tar.gz

$cd redis-5.0.3

$ make

运行：

src/redis-server

客户端：

/opt/redis/src/redis-cli --raw

#如果需要密码

> auth "dis2019"

> set foo bar

OK

> get foo

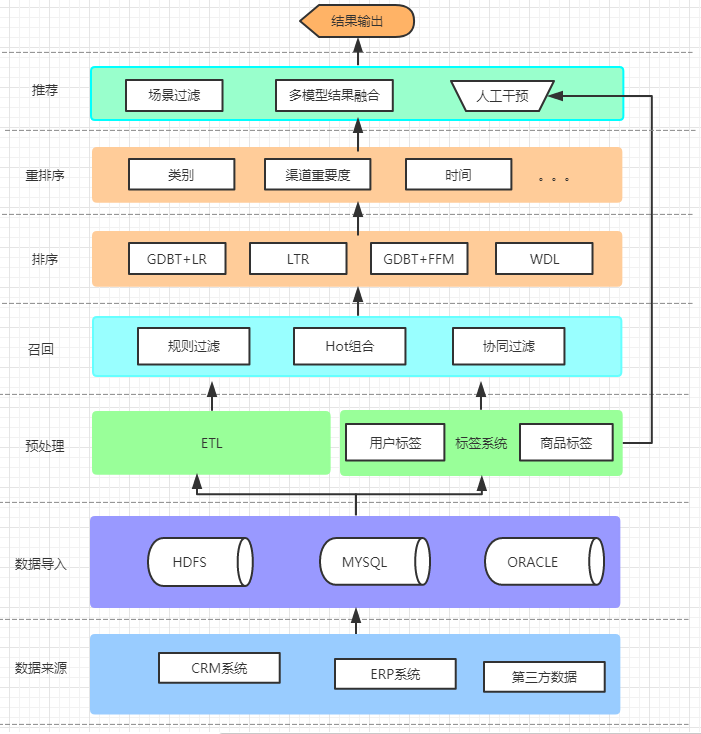
> keys \* //查询所有key

> dbsize //查询key数目

配置： 配置文件启动未完成

## 数据流程

上文中很多经典的推荐算法可以通过用户行为或者评分直接获取推荐列表，但在工业界为了提高预测和推荐的准确率、提高响应时间往往会在数据流程上做一些系统性的改良。一般情况下，一个完成批处理流程，主要分四大部分，预处理、召回、排序和重排序。



### 召回（Recall）

召回过程是一种匹配（Match）的过程，也就是基于当前用户（画像、历史行为）和上下文，快速在全库里找到TopN最相关的item，也就是物品的候选集。召回的目的就是减小的排序阶段的物品范围，提高推荐的命中率。召回的方式是比较多的，可以从数据类型还是那个划分，比如基于用户行为和基于用户偏好两种；也可以根据算法划分，比如协同过滤算法和关联算法，不同算法获得候选集再进行融合。

基于用户行为的召回，也就是根据用户购买行为推荐相关或相似的商品，对于已购买的某个商品，不需要重复推荐，可以根据商品种类和用途选择相关或者相似的商品。比如向IPad买家推荐IPad保护套而不是IPad。而对于一些周期性比较强的日用品，像肥皂、洗发水等，则根据一个购买周期再次推荐。

基于用户偏好的召回，也就是用户画像的构建，结合商品品牌、适用人群、价格指数以及用户对商品的点击、购买、关注和收藏等行为。通过用户画像可以确定一些长期推荐的品类。另外，移动互联网背景下，用户习惯于多终端购物，因此需要打通PC、移动app、微信和QQ等信息的融合，从而实现更加精准的画像。

基于地域的召回：一般用户用户冷启动，即网络行为比较少的新用户。

#### 基于Item相似度的召回服务

### 排序（Rank）

排序的步骤一般是对召回的结果进行打分和精细化排序。用户对于与不同商品的接受度是不一样的，排序要做的就是预测商品被用户接受的概率，将概率高的排在展示靠前的位置。排序模块从排序粒度上可以分为精细化排序和粗粒度排序。从指标来看，可以分为单目标排序和多目标排序。因为工业界数据多是隐式反馈类型，不同目标表达了不同的偏好程度，而且用户表达满意度的方式也不尽相同，为了综合目标的收益最大化，一把采用多目标的排序方法。

在融合模型的基础上，推荐排序的问题转化为分类的问题来实现，也就是从用户交互日志中通过模型训练特征权重，再通过LTR排序算法改进，提升转化率。

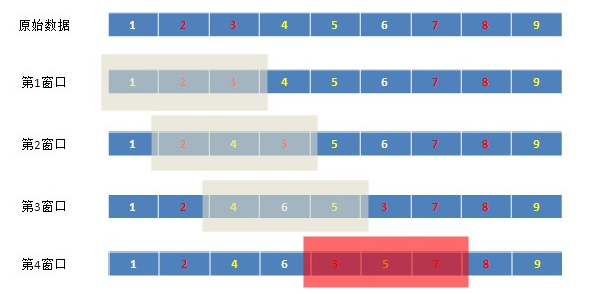
图片包含 屏幕截图

描述已自动生成

### 重排序（Rerank）

重排序可以理解为排序阶段的一种补充性措施，可以有效的防止推荐列表出现同类物品过于密集的现象。比如用户近期一直在看女装外套，如果前十个商品都推荐女装会严重影响客户体验，所以要对推荐物品进行品类的交替和打散，包括品牌的交替和打散。另外，平台在可以在重排序阶段进行强制营销活动（如广告、新品上架）。

下面介绍一种基于滑动窗口的品类打散算法。模拟用户在浏览手机屏幕时的浏览窗口，通过参数设置窗口的大小，保证在窗口向后移动时，窗口内的商品如果出现同一品类，将该商品向后移出该窗口范围。



Scala代码实例：

  /\*\*

    \* 推荐列表按品类打散

    \* 品类窗口为w，即相邻W+1个物品不能为同一品类,default:1

    \* 注意：调用此方法前需要保证品类足够多，否则结果会放弃重复品类的物品

    \* @return (rank, item)

    \*/

def sortByCate(ori: Array[(String, String)], W:Int=1): Set[(Int, String)] = {

    var buffer = new mutable.ListBuffer[(String,String)]

    val result = new mutable.ListBuffer[(Int, String)]

    for(item <- ori) buffer += item

    var windowCate = ""

    var index = 0

    var i = 1

    while (buffer.size > 0) {

      if(windowCate == buffer(index).\_2) {

        index = index + W

      }else{

        val temp =  (i, buffer(index).\_1)

        result += temp

        i += 1

        windowCate = buffer(index).\_2 //cate

        buffer -= buffer(index)

        index = 0

      }

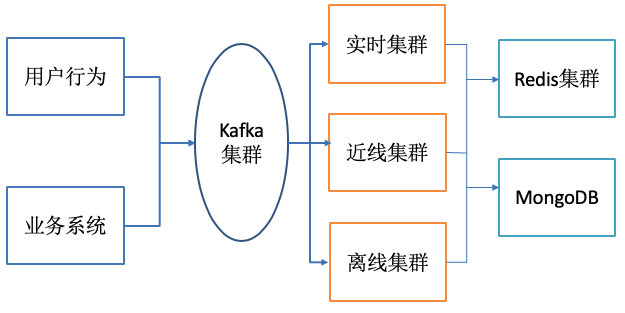
      if(index >= buffer.size) buffer.clear()

    }

    result.toSet

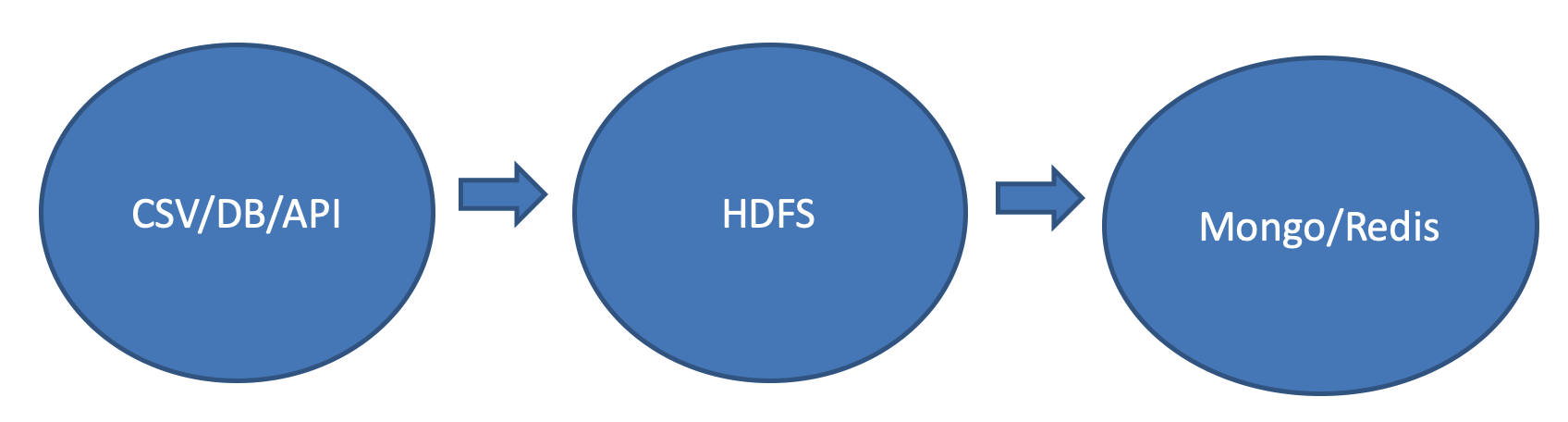
  }

## 系统架构与实现模块



### 离线计算

离线（offline）模块一般会把几个月的用户数据作为训练数据，更新周期为一天一次。离线计算的数据流支持多种数据源，比如csv文本文件解析、数据库读取或者API调用。一般情况下离线数据会存储半年甚至更久的数据，所以采用HDFS分布式存储。



### 近线计算

近线（nearline）一般一小时更新一次。如果用户操作频发，可以5分钟更新一次。如果用户用户操作行为很少，就可以将近线的行为时间拉长，以获取更大排序候选集。

图片包含 屏幕截图

描述已自动生成

### 实时计算

实时（online）更新，算法要求轻量级，为保证速度，可以通过规则融合的方法。

Spark Streaming 实时流处理，将数据流作为一张没有边界的表。

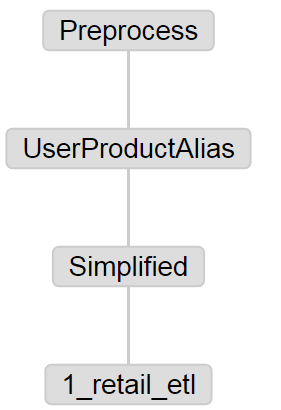
图片包含 屏幕截图

描述已自动生成

### 具体Job流程

#### 数据导入和转换

#### A．零售数据



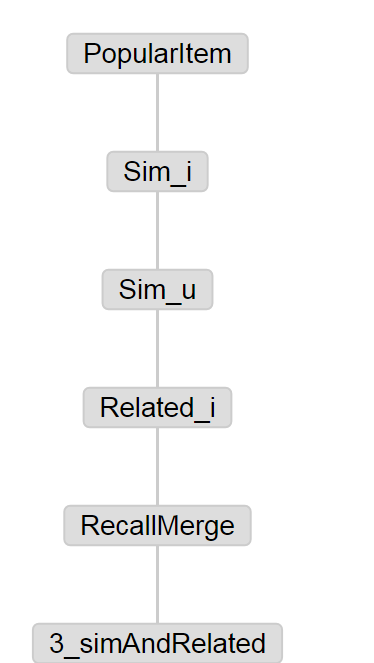
#### B．电商数据

通过数据格式转换，将电商场景的行为、用户和商品数据合并转换成预设定格式。

图片包含 屏幕截图

描述已自动生成

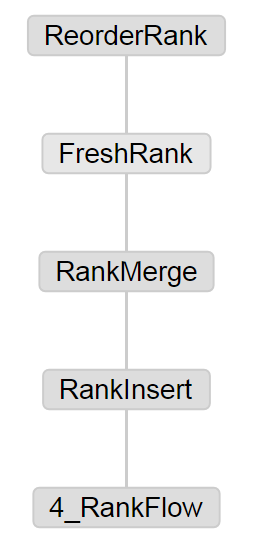
#### 1.2 召回



#### 1.3 精排序

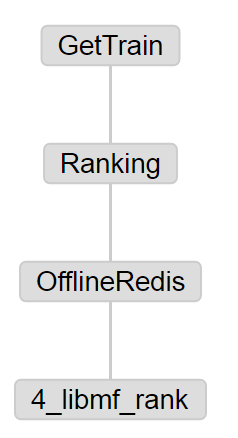
#### A．零售数据

零售场景的排序分为复购、和新鲜的子排序，分别计算完成后进行总体的融合操作。最后插入redis。



#### B．电商数据

电商场景当前选用了一种简单的方式进行排序。使用libmf进行排序后之间插入redis。



#### 2. 实时推荐lambda架构

图片包含 屏幕截图

描述已自动生成

实时推荐策略

A．根据用户行为优先计算实时推荐结果

根据用户行为- 推荐品类TopN

B．与离线计算的redis基础推荐列表进行融合

C．优先调用API返回结果Top10（param）

D．存储redis以备用户刷新调取

E．存储mongodb备份(可闲时进行)

2.1 实时更新（实时查询）

当用户刷新页面时通过API读取推荐列表实现。比如，一天的推荐结果为Top100，app页面3页显示15条记录。查询更新API读取top15条之后的数据返回给用户。

2.2 实时计算

略

## 推荐场景

### 猜你喜欢

猜你喜欢是一种综合性推荐。



### 店铺推荐

当用户点击某个商品时，商品详情页周边和下方显示的本店铺商品推荐。

图片包含 屏幕截图

描述已自动生成

### 组合推荐

商品主页下方推荐的商品配套组合。

图片包含 屏幕截图

描述已自动生成

# 线下商超案例

线下商超的项目实现了线下商品推荐，对象为华北地区某大型连锁超市。该商场有完整的CRM系统和会员体系，但是痛点在于客户流失率高，很多门店的业绩增加乏力，毛利润不高。通过打通后台CRM会员管理系统，结合客户消费日志挖掘，搭建会员-商品推荐系统。

线下商超的推荐与电商推荐有很大不同，比如交易数据无法做到实时反馈，而且不同电商平台有自己的APP终端，线下商超的网上商城流量很小，主要触达方式是通过微信公众号和短信提示。

## 数据流程

线下商超的数据内容和结构与电商基本一致，其中最大的不同在于线下商超场景下顾客没有关于商品的行为类型（action\_type）。取而代之的只有购买行为，也就是说商超日志数据就是顾客的交易记录。

### 召回

召回的具体操作是根据不同的推荐场景决定的。比如给用户推荐重复购买的商品，那召回候选集只能是用户购买超过2次的商品集合。如果是推荐新鲜商品，那召回候选集只能是用户没有购买的商品集合。

### 排序

同上召回操作，接受召回结果进入排序模块。

## 推荐场景

### 复购推荐

复购推荐对于线下商超推荐十分重要，因为超市客户的复购比例很高，有些门店复购场景占比超过50%。最常见的复购商品品类包括瓜果蔬菜、柴米油盐、护理产品等等。这些商品有其消耗周期性，比如一个4口家庭可能不到一个月就会买一次酱油，这时推荐系统要做的就是在酱油耗尽前及时提醒客户购买。触达方式上，系统可以发一些满减优惠券，促使客户进行打包消费，从而提高客单价。

复购推荐本身有其特有的复杂度，比如说有些客户定期会购买某一个品牌的酸奶，但是有时会因为商品短缺、注意力转移等等很多因素影响客户的最终成交。

图片包含 文字

描述已自动生成

### 新鲜推荐

给客户推荐之前未购买过的商品。一般场景的推荐都是新鲜推荐，只是商超比较特殊。在超市消费中，周期性消费占比较高，新鲜商品推荐的目的在于提高客单价和整体毛利润。比如对于价格不明感的客户应该适当推荐相似的高毛利商品，但要注意价格的上浮限制。推荐相似商品，就可以运用协同过滤的思想计算商品之间的相似度。

另一种方法与电商场景类似，将新鲜推荐转换为分类或排序问题。先召回候选集，然后根据特征学习的模型预测候选集商品的排名。将排名较高的几个商品通过公众号或者短信进行提示性推荐。

通过客户分群形成的特征空间数据：

图片包含 屏幕截图

描述已自动生成

# 金融领域的推荐

保险案例为某大型银行保险电销业务。

## 基于客户细分的保险推荐

名单类型分析

图片包含 文字

描述已自动生成

信用卡分群

图片包含 屏幕截图

描述已自动生成

将银行所有的信用卡客户分成四类。

cluster-3：优质客户

cluster-2：潜力优质客户

cluster-1: 一般客户

cluster-4：劣质客户

保险客户分群

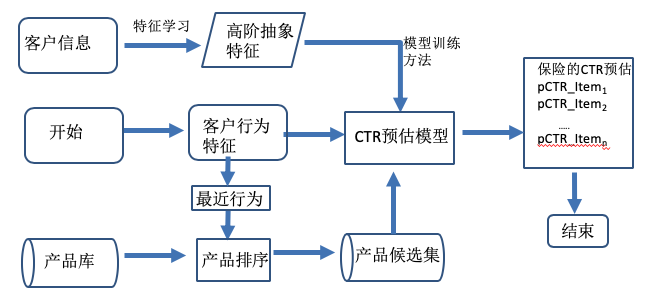


构建APP行为的回归模型

图片包含 屏幕截图

描述已自动生成

## 基于CTR预估的保险需求模型



图片包含 文字, 地图

描述已自动生成

## 集成学习下的融合模型

图片包含 标牌, 天空

描述已自动生成

# 文本推荐

新闻推荐系统

略

1. <https://yongyuan.name/blog/ann-search.html> [↑](#footnote-ref-1)
2. mmapped解释：在计算中，mmap是一个符合POSIX标准的Unix系统调用，它将文件或设备映射到内存中。它是一种内存映射文件I / O的方法。它实现了请求分页，因为文件内容不是直接从磁盘读取的，最初根本不使用物理RAM。在访问特定位置之后，以“懒惰”方式执行来自磁盘的实际读取。在不再需要内存之后，重要的是munmap（2）指向它的指针。可以使用mprotect管理保护信息，并且可以使用madvise强制执行特殊处理。 在Linux，Mac OS X和BSD中，mmap可以创建多种类型的映射。其他操作系统可能仅支持这些操作系统的子集，例如，共享映射在没有全局VFS或I / O高速缓存的操作系统中可能不实用。 [↑](#footnote-ref-2)
3. 一个正版流媒体音乐服务平台，2008年10月在瑞典首都斯德哥尔摩正式上线。 [↑](#footnote-ref-3)
4. <https://github.com/nmslib/hnswlib> [↑](#footnote-ref-4)