第一部分 基于大数据的推荐系统开发基础

# 推荐系统导论

## 推荐系统的起源

面对大数据量的信息过载，如何从海量信息中过滤出对自己有用的信息，成为推荐系统首要解决的问题。

最早的信息过滤方法是通过类目导航的方式进行，比如早期的门户网站雅虎、新浪、搜哦等等。门户网站需要用户被动的按照设定好的类目逐层查找。后来搜索引擎出现，它只需要用户提供意图明确的查询文字就能够提供大量的相关信息。

推荐系统会根据用户以往的行为，主动提供给用户一种选择。所以，推荐系统的优势比较明显，比如用户大多数情况下并没有明确的意图。推荐系统可以帮助用户发现，带给用户惊喜。

## 推荐系统的场景

热度排行榜：

不同的排行榜算法：

单一维度的投票

考虑多个维度的综合打分

考虑时间因素：如引入衰减权重（半衰期、冷却定律）

考虑反馈信息：如Reddit算法

考虑置信度：如威尔逊区间

防止马太效应：如MAB算法等

评价：

容易实现，可以解决新用户的冷启动问题

更新很慢，很难推新

排行榜很多事一个业务问题，而不是一个算法问题

## 推荐系统的算法

推荐系统的核心问题，如何评估一个用户对一个物品的评分（喜欢程度）。

影响推荐效果的因素：

用户交互界面（user Interface），用户推荐系统的第一感知很重要，比如：Twitter的“favorite“图案由star换成heart

数据层面，数据收集：有效性（是否真实、准确）、全面性（是否有偏）

领域知识（Domain knowledge）：产品的定位、具体推荐需求的理解

算法迭代（Algorithm/Model）

传统的推荐系统，包括热度排行（Popularity Ranking）、协同过滤（collaborate filter）、基于内容或知识的推荐（Content-based/Knowledge-based）、混合推荐方法（Hybird Approaches）。其中最为核心的就是协同过滤，协同过滤分为Memory-Based方法和Model-Based方法，前者又可以分为Item-based和user-Based；后者有很多方法，包括频繁项挖掘、聚类、分类、回归、矩阵分解、RBM、图模型等等算法。

近年来推荐系统领域的最新研究，包括排序学习（Learning To Rank）、因子分解机（Factorization Machine）和深度学习（Deep Learning）。

协同过滤的思想来源：现实生活中朋友之间相互推荐自己喜欢的东西。

基本组成元素：

N个用户，M个物品

评分矩阵：用户对物品的评分（显式）或用户对物品的行为（隐式）

相似度度量：物品与物品（用户与用户）之间的相似性

User-based CF

算法步骤

1 计算目标用户的前K个相似度用户（neighborhood）

各种相似性度量

2 找出相似用户喜欢的物品，并预测目标用户对这些物品的评分

预测模型：KNN，regression

3. 过滤掉目标用户已经消费过的物品

4. 将剩余物品按照预测评分排序，并返回前N个物品

item-based CF

基本思想是计算item之间两两相似性，目标用户对某个item的评分可以根据其对相似的items的评分来预测，选取用户预测评分最高的前N个商品进行推荐。

item-based在电商推荐中逐渐成为主流的原因：

降低实时计算的复杂性

解决用户的冷启动问题

user-based CF与item-based CF对比：

item-based准确性好，表现稳定可控，便于离线计算；但是推荐结果的多样性会差一点，一般不会给用户惊喜性。

user-based可以帮助用户发现新的商品，但是需要较复杂的在线计算，需要处理新用户的问题。

item之间的相似性比较单纯，是静态的；而user之间的相似性比较复杂，是动态的，应用时需要特征小心。

协同过滤的优缺点：

优点：需要很少的领域知识，模型通用性很强；工程上实现简单，效果很好

缺点：冷启动问题（new user/items），数据稀疏性问题，假设”过去的行为决定现在“没有考虑具体情景的差异

热门倾向性（Popularity Bias），很难推荐出小众偏好。

基于内容/知识的推荐

什么是内容，

文本描述：通常用NLP技术挖掘关键词

item的属性，比如电影的主题，衣服的材质等

item的特征，例如语音信号表示，图像向量表示等

基本组成

item特征向量，如文本的TF-IDF向量

用户profile向量：通常通过用户偏好的items来提取

匹配分：直接计算cosine，分了/回归模型

基于内容推荐的评价

优点

能够推荐出用户独有的小众偏好

可以一定程度解决数据稀疏问题和item的冷启动问题

通常具有较好的可解释性

不足

对于很多推荐问题，提取出有意义的特征并不容易

很难将不同item的特征组合在一起

很难带给用户惊喜性

如果用户的profile挖掘不准，推荐效果往往很差

混合方法（Hybird Approaches）

模型融合方法

加权（weighted）组合

切换（switching）：确定一个合理的切换跳进

混合（Mixed）

特征组合（feature combination）：不同模型特征组合到一起

特征扩展（feature augmentation）：一个模型的输出作为另一个的特征

级联（Cascade）：粗排 -> 精排

评价

通常比单个算法表现好

需要在不同算法之间、理论效果和实际可行性之间仔细权衡

Learning To Rank

大多数推荐结果是一个列表，因此推荐结果的排序也很重要。

L2R是将排序看做一个机器学习问题。

排序的评价准则：

Normalized Discounted Cumulative Gain(NDCG)

Mean Reciprical Rank(MRR)

问题：很难直接利用这些指标来指导学习过程

常用算法：

Pointwise: LR, GBDT, SVM

Pairwise: RankSVM, RankBoost, RankNet, LambdaRank

Listwise: AdaRank, LambdaRank/LambdaMart, ListNet, ListMLE

# 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)。

分块矩阵的Scala代码案例：

// 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 )

## 数据预处理

为了让原始数据可用于机器学习算法，我们需要先对其进行清理，并且可能需要进行各种转 换，之后才能从转换后的数据里提取有用的特征。数据的转换和特征提取联系紧密。某些情况下，一些转换本身便是特征提取的过程。

在之前处理电影数据集时，我们已经看到数据清理的必要性。一般来说，现实中的数据会存 在信息不规整、数据点缺失和异常值问题。理想情况下，我们会修复非规整数据。但很多数据集都源于一些难以重现的收集过程(比如网络活动数据和传感器数据)，故实际上会难以修复。值缺失和异常也很常见，且处理方式可与处理非规整信息类似。总的来说，大致的处理方法如下。

过滤掉或删除非规整或有值缺失的数据:这通常是必需的，但的确会损失这些数据中那 些好的信息。 **10**

填充非规整或缺失的数据:可以根据其他的数据来填充非规整或缺失的数据。方法包括 用零值、全局平均值或中值来填充，或是根据相邻或类似的数据点来做插值(通常针对 时序数据)等。选择正确的方式并不容易，它会因数据、应用场景和个人经验而不同。

对异常值做稳健处理:异常值的主要问题在于，即使它们是极值也不一定就是错的。到 底是对是错通常很难分辨。异常值可被移除或填充，但存在某些统计技术(如稳健回归) 可用于处理异常值或是极值。

对可能的异常值进行转换:另一种处理异常值或极值的方法是进行转换。对那些可能存 在异常值或值域覆盖过大的特征，进行对数或高斯核转换。这类转换有助于降低变量存 在的值跳跃的影响，并将非线性关系变为线性的。

异常数据与缺失数据的填充

## MLlib特征处理

初步完成源数据的清洗、处理和探索之后，就可以尝试从数据中提取供模型训练使用的特征。特征主要是指作为模型训练的变量，特征在模型训练之前需要转换成若干行的的向量，向量的元素就是数值。

特征可以概括地分为如下几种。

数值特征(numerical feature):这些特征通常为实数或整数，比如之前例子中提到的年龄。

类别特征(categorical feature):它们的取值只能是可能状态集合中的某一种。我们数据集中的用户性别、职业或电影类别便是这类特征。

文本特征(text feature):它们派生自数据中的文本内容，比如电影名、描述或评论。

其他特征:大部分其他特征最终都表示为数值。比如图像、视频和音频可被表示为数值

数据的集合，地理位置则可由经纬度或地理散列(geohash)表示。 本节我们将谈到数值特征、类别特征以及文本特征。

### 特征抽取

#### 数值特征

原始的数值和数值特征之间的区别是什么?实际上，任何数值数据都能作为输入变量。但是，机器学习模型中所学习的是各个特征的权重向量。权重在特征值到输出或目标变量(指在监督学 习模型中)的映射过程中扮演重要角色。

由此我们会想使用那些合理的特征，让模型能从这些特征中学到特征值和目标变量之间的关 系。比如年龄就是一个合理的特征。年龄的增加和某项支出之间可能存在直接关系。类似地，高 度也是一个可直接使用的数值特征。

当数值特征仍处于原始形式时，其可用性相对较低，但可以转化为更有用的表示形式。位置信息便是如此。

若使用原始位置信息(比如用经纬度表示的)，我们的模型可能学习不到该信息和某个输出 之间的有用关系，这就使得该信息的可用性不高，除非数据点的确很密集。然而，对位置进行聚合或挑选后(比如聚焦为一个城市或国家)，便可能和特定输出之间存在某种关联。

#### 类别特征

当类别特征仍为原始形式时，其取值来自所有可能取值所构成的集合，而不是一个数字， 故不能作为输入。之前例子中的用户职业便是一个类别特征变量，其可能的取值有学生、程序 员等。

这样的类别特征也称作名义(nominal)变量，即其各个可能取值之间没有顺序关系。相反， 那些存在顺序关系的(比如之前提到的评级，从定义上说评级 5 会高于或是好于评级 1)则被称 **5** 为有序(ordinal)变量。

将类别特征表示为数字形式，常可借助 *k* 之 1(1-of-*k*)编码方法进行。要想将名义变量表示 为可用于机器学习任务的形式，需要借助如 *k* 之 1 编码这样的方法。有序变量的原始值可能就能 直接使用，但也常会经过和名义变量一样的编码处理。

假设变量可取的值有 *k* 个。如果对这些值用 1 到 *k* 编序(索引)，则可以用长度为 *k* 的二元 向量来表示该变量的一个取值。在这个向量里，该取值对应的序号所在的元素为 1，其他元素 都为 0。

比如职业 student 对应的索引为 0，programmer 对应的索引为 1。那么 student 和 programmer 对应的值就分别为[1,0]和[0,1]。

#### 派生特征

前面曾提到，从现有的一个或多个变量派生出新的特征常常是有帮助的。理想情况下，派生出的特征能比原始变量带来更多信息。

比如，可以分别计算各用户已有的电影评级的平均数。这将能给模型加入针对不同用户的个 性化特征(事实上，这常用于推荐系统)。在前文中我们也从原始的评级数据创建了新的特征以 学习出更好的模型。

从原始数据派生特征的例子包括计算平均值、中位值、方差、和、差、最大值或最小值以及 计数。在先前的内容中，我们已看到如何从电影的发行年份和当前年份派生出了新的 movie age 特征。这类转换背后的想法常常是对数值数据进行某种概括，让模型更容易学习这些特征。

数值特征到类别特征的转换也很常见，比如划分为区间特征。进行这类转换的常见变量有年 龄、地理位置和时间。

#### 文本特征

Spark MLlib提供三种文本特征提取方法，分别为TF-IDF、Word2Vec和CountVectorizer。

**TF-IDF**

TF-IDF是一种广泛用于文本挖掘中的特征向量化方法，它可以反映术语对语料库中文档的重要性。

词语由t表示，文档由d表示，语料库由D表示。词频TF(t,d)是词语t在文档d中出现的次数。文件频率DF(t,D)是包含词语的文档的个数。如果我们只使用词频来衡量重要性，很容易过度强调在文档中经常出现而并没有包含太多与文档有关的信息的词语，比如“一个”，“该”和“属于”等等。如果术语经常出现在整个语料库中，则表示该术语不包含有关特定文档的特殊信息。逆文档频率(IDF)是一个术语多少信息提供了一个数值量度，IDF的数值化衡量词语提供多少信息，用以下公式表示：

其中是语料库中的文档的总数。由于使用对数，因此如果一个术语出现在所有文档中，则其IDF值将变为0。请注意，应用了平滑术语以避免对主体外的术语除以零。TF-IDF度量只是TF和IDF的乘积：

在下面的代码段中，我们以一组句子开始。首先使用分解器Tokenizer把句子划分为单个词语。对每一个句子（词袋），我们使用HashingTF将句子转换为特征向量，最后使用IDF重新调整特征向量。这种转换通常可以提高使用文本特征的性能。然后，我们的特征向量可以在算法学习中。

import org.apache.spark.ml.feature.{HashingTF, IDF, Tokenizer}

val sentenceData = spark.createDataFrame(Seq(

(0, "Hi I heard about Spark"),

(0, "I wish Java could use case classes"),

(1, "Logistic regression models are neat")

)).toDF("label", "sentence")

val tokenizer = new Tokenizer().setInputCol("sentence").setOutputCol("words")

val wordsData = tokenizer.transform(sentenceData)

val hashingTF = new HashingTF()

.setInputCol("words").setOutputCol("rawFeatures").setNumFeatures(20)

val featurizedData = hashingTF.transform(wordsData)

// alternatively, CountVectorizer can also be used to get term frequency vectors

val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")

val idfModel = idf.fit(featurizedData)

val rescaledData = idfModel.transform(featurizedData)

rescaledData.select("features", "label").take(3).foreach(println)

**Word2Vec**

Word2Vec是一个Estimator表示代表文档的单词序列来训练一个Word2VecModel。该模型将每个单词映射到唯一的固定大小的向量。Word2VecModel使用文档中所有单词的平均值将每个文档转换为向量。之后，将此向量用作特征进行预测，或者用以计算文档相似度。

下面代码中，假设一组文档，其中每个文档代表一个词语序列。将每个文档转换成特征向量，然后将特征向量传递到学习算法中。

import org.apache.spark.ml.feature.Word2Vec

// Input data: Each row is a bag of words from a sentence or document.

val documentDF = spark.createDataFrame(Seq(

"Hi I heard about Spark".split(" "),

"I wish Java could use case classes".split(" "),

"Logistic regression models are neat".split(" ")

).map(Tuple1.apply)).toDF("text")

// Learn a mapping from words to Vectors.

val word2Vec = new Word2Vec()

.setInputCol("text")

.setOutputCol("result")

.setVectorSize(3)

.setMinCount(0)

val model = word2Vec.fit(documentDF)

val result = model.transform(documentDF)

result.select("result").take(3).foreach(println)

**计数向量器CountVectorizer**

计数向量器CountVectorizer和计数向量器模型CountVectorizerModel是通过计数的方式将文档转换成向量。当不存在先验词典时，CountVectorizer可以用作Estimator来提取词汇表并生成CountVectorizerModel。该模型为词汇表上的文档生成稀疏表示，可以将其传递给其他算法，例如LDA[[1]](#footnote-1)。

在拟合过程中，CountVectorizer将选择整个语料库中按词频排列的前vocabSize大小的词。参数minDF通过指定一个单词必须出现在词汇表中的最小数量（如果小于1.0，则为小数）来影响拟合过程。另一个可选的二进制切换参数控制输出向量。如果将其设置为true，则所有非零计数都将设置为1。这对于模拟二进制而不是整数计数的离散概率模型特别有用。

代码示例：

import org.apache.spark.ml.feature.{CountVectorizer, CountVectorizerModel}

val df = spark.createDataFrame(Seq(

(0, Array("a", "b", "c")),

(1, Array("a", "b", "b", "c", "a"))

)).toDF("id", "words")

// fit a CountVectorizerModel from the corpus

val cvModel: CountVectorizerModel = new CountVectorizer()

.setInputCol("words")

.setOutputCol("features")

.setVocabSize(3)

.setMinDF(2)

.fit(df)

// alternatively, define CountVectorizerModel with a-priori vocabulary

val cvm = new CountVectorizerModel(Array("a", "b", "c"))

.setInputCol("words")

.setOutputCol("features")

cvModel.transform(df).select("features").show()

假设我们有如下的DataFrame包含id和texts两列：

id | texts

0 |Array("a", "b", "c")

1 |Array("a", "b", "b", "c","a")

文本中的每一行都是一个文档类型的数组(字符串)。调用的CountVectorizer产生词汇(a,b,c)的CountVectorizerModel，转换后的输出向量如下：

id | texts | vector

0 |Array("a", "b", "c") | (3,[0,1,2],[1.0,1.0,1.0])

1 |Array("a", "b", "b", "c","a") |(3,[0,1,2],[2.0,2.0,1.0])

每个向量代表文档的词汇表中每个词语出现的次数。

**正则化特征**

### 特征转换

在机器学习中，数据处理是一件比较繁琐的事情，需要对原有特征做多种处理，如类型转换、标准化特征、新增衍生特征等等，需要耗费大量的时间和精力编写处理程序，不过，自从Spark推出ML后，情况大有改观，Spark ML包中提供了很多现成转换器，例如：StringIndexer、IndexToString、OneHotEncoder、VectorIndexer，它们提供了十分方便的特征转换功能，这些转换器类都位org.apache.spark.ml.feature包下。

### 特征选择

特征选择（Feature Selection）是从特征向量中选择那些更有效的特征，再组成新的、更简单有效的特征向量的过程。它是数据分析过程中的常用手段，尤其在高维数据中，可以提出冗余特征，防止过拟合，提升模型性能。

**VectorSlicer**

VectorSlicer是一种转换器，它采用特征向量并输出带有原始特征子数组的新特征向量。这对于从向量列中提取特征很有用。VectorSlicer接受具有指定索引的向量列，然后输出一个新的向量列，其值通过这些索引选择。

索引有两种类型，一是整数索引，代表向量的索引setIndices()。二是字符串索引，表示向量中的要素名称setNames()。这要求向量列具有AttributeGroup，因为实现在Attribute的名称字段上匹配。

以整数和字符串指定都可以接受。此外，您可以同时使用整数索引和字符串名称。必须至少选择一项功能。不允许重复的功能，因此所选索引和名称之间不能有重叠。请注意，如果选择了要素名称，则在遇到空的输入属性时将引发异常。

输出向量将首先按所选索引（按给定顺序）对要素进行排序，然后按所选名称（按给定顺序）对要素进行排序。

假设我们有一个DataFrame与列userFeatures：

userFeatures

[0.0, 10.0, 0.5]

userFeatures是一个包含三个用户特征的向量列。假设userFeature的第一列全部为零，因此我们要删除它并仅选择最后两列。 VectorSlicer使用setIndices（1,2）选择最后两个元素，然后生成一个名为features的新向量列：

|  |  |
| --- | --- |
| **userFeatures** | **features** |
| [0.0, 10.0, 0.5] | [10.0, 0.5] |

假设userFeatures有输入属性，如[“f1”，“f2”，“f3”]，那么我们可以使用setNames（“f2”，“f3”）来选择它们。

|  |  |
| --- | --- |
| **userFeatures** | **features** |
| [0.0, 10.0, 0.5] | [10.0, 0.5] |
| ["f1", "f2", "f3"] | ["f2", "f3"] |

以下是实现向量选择的一个scala代码示例

import org.apache.spark.sql.Row

import org.apache.spark.sql.types.StructType

val data = Arrays.asList(

Row(Vectors.sparse(3, Seq((0, -2.0), (1, 2.3)))),

Row(Vectors.dense(-2.0, 2.3, 0.0))

)

val defaultAttr = NumericAttribute.defaultAttr

val attrs = Array("f1", "f2", "f3").map(defaultAttr.withName)

val attrGroup = new AttributeGroup("userFeatures", attrs.asInstanceOf[Array[Attribute]])

val dataset = spark.createDataFrame(data, StructType(Array(attrGroup.toStructField())))

val slicer = new VectorSlicer().setInputCol("userFeatures").setOutputCol("features")

slicer.setIndices(Array(1)).setNames(Array("f3"))

// or slicer.setIndices(Array(1, 2)), or slicer.setNames(Array("f2", "f3"))

val output = slicer.transform(dataset)

output.show(false)

运行结果：

|userFeatures |features |

+--------------------+-------------+

|(3,[0,1],[-2.0,2.3])|(2,[0],[2.3])|

|[-2.0,2.3,0.0] |[2.3,0.0] |

+--------------------+-------------+

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。

## 相似度的计算方式

距离度量和相似度度量[[2]](#footnote-2)：

距离度量是计算两个对象之间的差异程度。

相似度量是计算两个对象之间的近似程度。

未完待续。。。

相似度的计算方式有很多，每一种适合的场景也不一样。下面介绍几种常用的相似度计算方法：

#### 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

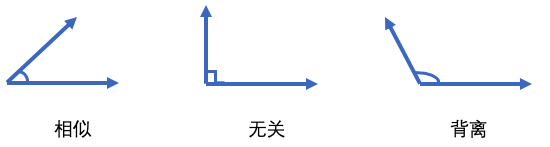
0.2240092377397959

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。

从下图简单示意中看出，夹角余弦就是用两个向量之间的夹角大小对应相似的程度。夹角越小代表越相似。如果两个向量则正交余弦值为0，如果夹角为钝角则相似度为负值。



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

**杰卡尔德相似度适用场景**

简单来看，每一个相似度度量指标都不是完美的，比如杰卡尔德相似度无法将具体评分带入公式中，只是判断有和没有。所以，一般在不考虑项目的具体数值时使用。

**欧式距离相似度适用场景**

欧式距离在几何意义上，是对向量距离最简单的计算方法。它本身有很多限制，比如欧式距离并没有考虑向量中每个项目的权重，而是直接直接平方相加。

**余弦相似度适用场景**

余弦相似度通过向量的方向计算相似度，通常在推荐系统用户-商品评分、自然语言处理（NLP）等场景中使用。但是它本身也存在一些缺陷。比如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。

**对比欧式和余弦相似度**

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）。

而在实际的推荐系统应用中，评分矩阵通常是高维且稀疏的，所以计算余弦相似度比欧式距离更合适。

**皮尔森系数使用场景**

皮尔森系数，与夹角余弦类似，但是可以去中心化。比如评分时，有人倾向于打高分，有人倾向于打低分，他们的最后效果在皮尔森中是一样的。

**曼哈顿距离使用场景**

曼哈顿距离，一般在路径规划、地图类中常用，比如A\*搜索算法[[3]](#footnote-3)中使用曼哈顿来作为每一步代价值的一部分（F=G+H, G是从当前点移动到下一个点的距离，H是距离目标点的距离，这个H就可以用曼哈顿距离表示）

在某些情况下，如果单一计算方式无法满足系统要求，可以计算两种或者多种相似度，然后设置不同的权重最终累加。但是，这一方法要是具体情况而定，因为每中相似度数值的量纲可能不一样，需要进行归一化操作，不一定能直接做加法运算。

## 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。

通过CoordinateMatrix进行向量相似度计算scala代码：

import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix, MatrixEntry}  
val df = spark.createDataFrame(Seq(  
 (0, 0, 1.0),  
 (1, 0, 1.0),  
 (2, 0, 1.0),  
 (3, 0, 1.0),  
 (0, 1, 2.0),  
 (1, 1, 2.0),  
 (2, 1, 1.0),  
 (3, 1, 1.0),  
 (0, 2, 3.0),  
 (1, 2, 3.0),  
 (2, 2, 3.0),  
 (0, 3, 1.0),  
 (1, 3, 1.0),  
 (3, 3, 4.0)  
))  
  
val matrix = new CoordinateMatrix(df.map(row =>  
 MatrixEntry(row.getAs[Integer](0).toLong,  
 row.getAs[Integer](1).toLong,  
 row.getAs[Double](2))).toJavaRDD)  
// 计算similarity  
val result = matrix.toRowMatrix().columnSimilarities()  
// 打印相似度结果  
result.entries.  
 map(x => (x.i, x.j, x.value)).collect().foreach(println)

结果输出：

(2,3,0.2721655269759087)

(1,2,0.9128709291752768)

(1,3,0.596284793999944)

(0,3,0.7071067811865476)

(0,1,0.9486832980505139)

(0,2,0.8660254037844386)

按照以上案例，把向量化的相似度计算方法用到评分矩阵的计算中。对应的scala代码如下：

val matrix = new CoordinateMatrix(df.map(row =>  
 MatrixEntry(row.getAs[Integer](0).toLong,  
 row.getAs[Integer](1).toLong,  
 row.getAs[Double](2))).toJavaRDD)  
// 调用sim方法  
val x = matrix.toRowMatrix().columnSimilarities()  
// 得到相似度结果  
x.entries.collect().foreach(println)

//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[[4]](#footnote-4)。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[[5]](#footnote-5)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[[6]](#footnote-6)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[[7]](#footnote-7)

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)

#### SPTAG介绍

微软开源了搜索引擎Bing的搜索算法，称为空间分区式的树和图（SPTAG，Space Partition Tree And Graph），也是一种分布式近似最近邻搜索（ANN）库。可以为大规模向量搜索场景提供高质量向量索引构建。这里“树”为KD-Tree和K-Means树，“图”是相对邻域图（SPTAG-BKT）。SPTAG-KDT在指数构建成本方面很有优势，而SPTAG-BKT在超高维数据中的搜索精度方面很有优势。

运用在推荐系统中，假设用户-物品的评分表示为向量，通过L2距离（欧式距离）或余弦距离来比较向量。而查询向量返回的向量是与查询向量具有最小L2距离或余弦距离的向量。

**SPTAG工作原理：**

搜索算法一般分为索引（index）查询（search）两个部分。SPTAG的索引是基于kd-tree。其在于一维空间上的情况就是“平衡二叉树”，在高维空间上kd-tree会用第k维的大小来决定一个元素应该插入左子树还是右子树，同时为保持tree的平衡，剩余未进入tree的元素除第k维外方差最小。SPTAG正是以此来加速算法的速度。

另外，k-means是一种自动聚类的方法，算法先随机指定选取K个点做为初始聚集的簇心,分别计算每个样本点到 K个簇核心的余弦距离，找到距离最近的核心点，将它归属到对应的簇，所有点都归属到簇之后， M个点就分为了 K个簇。之后重新计算每个簇的重心，将其定为新的“核心”，重复上述步骤直到新核心不再改变为止或者改变距离达到一定值后中止。那么最终的K个簇就是最终的聚类结果。

结合kd-tree和k-means这两种算法，SPTAG允许用户在几毫秒内搜索数十亿条信息。

**安装SPTAG**

安装系统要求：

gcc >= 5.0

swig >=3.0

cmake>= 3.12.0

boost >= 1.67.0

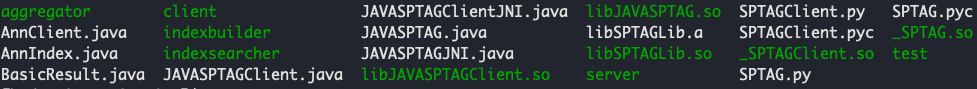
如果你的系统gcc版本过低，先升级gcc然后再升级cmake

从github地址[[8]](#footnote-8)下载源码，进入SPTAG-master目录，执行命令：

mkdir build

cd build && cmake .. && make

安装完成之后SPTAG-master目录下回生成一个Release/文件夹，其中包含一些可执行文件、Java和Python的API文件。



本节只介绍python的调用方式，其他方式可以docs文档中查看。

先用MovieLens的数据集ml-1m上进行测试，通过SPTAG的python API进行近似最近邻找找。找出与电影《玩具总动员》[1,Toy Story (1995),Animation|Children's|Comedy]最相似的Top10个其他电影。

第一步骤：按照格式生成原始数据

假设向量的维度为3格式如下：

<metadata1>\t<v11>|<v12>|<v13>|

<metadata2>\t<v21>|<v22>|<v23>|

...

其中每行表示一个向量，其元数据及其值由制表符分隔。向量的每个维度都由 |或使用 --分隔。

第二步骤：按照格式生成查询文件

查询文件的格式与上面相同。

第三步骤：构建索引

第四步骤：查询索引

下面是python代码构建索引和查询索引。

import sys  
sys.path.append('/myfile/SPTAG-master/Release/')  
import SPTAG  
import numpy as np  
  
class SPTAG\_Movie:  
 def \_\_init\_\_(self, dataFile):  
 self.datafile = dataFile  
 self.data, self.data\_labels = self.loadData()  
 # self.testBuild("KDT", "cosine", self.data, 'testindices')  
 # self.testSearch('testindices', q, k)  
  
 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 testBuild(self, algo, distmethod, x, out):  
 i = SPTAG.AnnIndex(algo, 'Float', x.shape[1])  
 i.SetBuildParam("NumberOfThreads", '4')  
 i.SetBuildParam("DistCalcMethod", distmethod)  
 ret = i.Build(x, x.shape[0])  
 i.Save(out)  
  
 def testSearch(self, index, q, k):  
 j = SPTAG.AnnIndex.Load(index)  
 for t in range(q.shape[0]):  
 result = j.Search(q[t], k)  
 result = [x+1 for x in result[0]]  
 print (self.idToName(result)) #索引ids+1=movieID  
 print (result[1]) # distances  
  
 def idToName(self, list):  
 print("读取id-name文件...")  
 data = {}  
 with open("../ml-1m/movies.csv") as f:  
 # for line in open("../ml-1m/movies.csv"):  
 next(f)  
 for line in f:  
 sp = line.strip("\n").split(",")  
 data[int(sp[0])] = sp[1]+","+sp[2]  
 result = [data.get(x,x) for x in list]  
 for i in result:  
 print(i)

if \_\_name\_\_ == '\_\_main\_\_':  
 # ann = SPTAG\_Movie("/xiaoi/josh/ann/ml\_1m\_vector.txt")  
 # ann.testBuild("KDT", "cosine", ann.data, 'testindices')  
 ann = SPTAG\_Movie("/xiaoi/josh/ann/query\_input.txt")  
 ann.testSearch('testindices', ann.data[0], 10)

输出结果得到与电影玩具总动员的最相似的十个电影：

[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)

#### 矩阵分解MF与ANN结合

在实际推荐系统中，高维度的超大稀疏矩阵在单机情况下很难构建索引进行ANN最近邻搜索。主要是因为内存资源不够，即使通过磁盘存储来缓解内存压力，整体构建索引的时间也是非常漫长的。在建构索引之前，有没有一种方法能够降低数据的维度又不会损失太多的信息呢？数据降维就是一种有效的方法。这里介绍一种基于矩阵分解（MF）的方法。

libMF是一个关于推荐系统的矩阵分解库。它会利用用户和物品的评分矩阵训练出一个模型，然后就可以通过该模型预测评分了。在这个过程中会生成两个矩阵，分别是用户矩阵P和物品矩阵Q。每个矩阵的维度是可以指定的。

libMF在进行矩阵分解之后会生成一个模型文件，前几行如下：

f 0

m 88997

n 33183

k 1000

b 1.41962

这几行主要记录用户和物品的维度信息等等，读取矩阵的时候可以跳过。

下面读取LibMF矩阵分解之后生成的Item矩阵，然后再用Annoy构架索引：

def loadMF(self):  
 print("Load mf data...")  
 data = []  
 input\_file = open(self.datafile)  
 for line in islice(input\_file, 5, None):  
 list = line.strip().split(" ")  
 itemId = list[0]  
 flag = list[1]  
 vector = " ".join(list[2:])  
 if(itemId[0] == "q" and flag == "T"):  
 data.append((itemId[1:], vector))  
 return data

#### SPTAG高级操作

1. **增量添加和删除**

实时增加和删除向量是SPTAG独有的一个功能，这一功能可以利用在推荐系统的实时推荐中。通过增加向量，系统可以实时更新用户的最新行为，也可以增加新注册用户的特征。

下面是SPTAG增加和删除向量的操作：

def addItem(self, index, x, out, algo, distmethod):  
 if index != None:  
 i = SPTAG.AnnIndex.Load(index)  
 else:  
 i = SPTAG.AnnIndex(algo, 'Float', x.shape[1])  
 i.SetBuildParam("NumberOfThreads", '4')  
 i.SetBuildParam("DistCalcMethod", distmethod)  
 if i.Add(x, x.shape[0]):  
 i.Save(out)  
  
def deleteItem(self, index, x, out):  
 i = SPTAG.AnnIndex.Load(index)  
 ret = i.Delete(x, x.shape[0])  
 print (ret)  
 i.Save(out)

1. **实时API访问查询**

案例：电影SPTAG实时查询最近邻服务：

步骤1：配置servic.ini

[Service]

ListenAddr=122.226.240.159

ListenPort=8001

ThreadNumber=8

SocketThreadNumber=8

[QueryConfig]

DefaultMaxResultNumber=100

DefaultSeparator=|

[Index]

List=KDT //索引的类型

[Index\_KDT]

IndexFolder=testindices //索引文件

步骤2：启动server服务

启动server，正常情况下会打印导入索引的日志，如果没有则是配置文件有问题。

/xiaoi/josh/SPTAG-master/Release/server -m socket -c ./service.ini

步骤3：调用SPTAG\_Client读取本地查询文件搜索最近邻

def testSPTAGClient(self, q, k):  
 index = SPTAGClient.AnnClient('122.226.240.159', '8001')  
 while not index.IsConnected():  
 time.sleep(1)  
 index.SetTimeoutMilliseconds(18000)  
 for t in range(q.shape[0]):  
 result = index.Search(q[t], k, 'Float', False)  
 result = [x+1 for x in result[0]]  
 print (self.idToName(result)) #索引ids+1=movieID  
 print (result[1]) # dis

以下是通过8001端口查询电影《Toy Story》Top10最近邻的输出结果：

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)

以下是增加和删除向量的日志：

Add 1 vectors

Save Vector To testindices/vectors.bin

Save Vector (3708, 6041) Finish!

Save KDT to testindices/tree.bin

Save KDT (1,3706) Finish!

…

Load Vector From testindices/vectors.bin

Load Vector (3708, 6041) Finish!

Load KDT From testindices/tree.bin

Load KDT (1,3706) Finish!

Load Graph From testindices/graph.bin

Load Graph (3708, 64) Finish!

Load DeleteID From testindices/deletes.bin

Load DeleteID (0) Finish!

True

Save Vector To testindices/vectors.bin

Save Vector (3708, 6041) Finish!

Save KDT to testindices/tree.bin

Save KDT (1,3706) Finish!

Save Graph To testindices/graph.bin

Save Graph (3708, 64) Finish!

Save DeleteID To testindices/deletes.bin

Save DeleteID (1000) Finish!

#### 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 | 单机Local | 集群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

**矩阵分解MF与ANN结合**

实验： 将CRM的评分数据集进行MF矩阵分解，再用Annoy和Sptag构建索引，查询Item近似最近邻物品

目录和文件：/xiaoi/josh/recommend/ann

数据： CRM

机器：140，24cores, 64G

相关命令：

time /xiaoi/recommend/project/install\_package/libmf-2.01/mf-train crm\_rating.txt crm\_libmf.model

python ann\_crm\_libmf.py 6760 //传参指定查询的Item ID

图片包含 屏幕截图

描述已自动生成

接上图：

图片包含 文字, 屏幕截图

描述已自动生成

根据LibMF矩阵分解之后在进行ANN搜索的实验结果，可以看出矩阵分解的误差越小，就需要增加迭代次数和矩阵维度（隐因子数量）。随之而然的就是训练时间的增加。从简单的抽样结果看，矩阵分解之后的向量最近邻结果，很难与原始余弦相似度的结果一致。

### 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。

## 基于文本的相似度

文本相似度计算方法：

图片包含 屏幕截图

描述已自动生成

总的来说，文本相似度量方法可以分为两大类:[[9]](#footnote-9)

String Based，即基于待比较的文本本身中的信息，该类方法评估的是”词法“上的相似性，或说朴素的相似性

Corpus Based，即基于一个较大的文本集合中的信息，该类方法评估的是“语义”上的相似性

#### String Based Methods

要比较两个文本的相似性，比较直观的方法是逐个字符比对，看看有多少个字符是一致的 —— String Based 的方法就是从这种思路中发展出来的。其中以字符为单位去比较的方法被统称为 Character Based，而以词为单位的则被称为 Term Based。

Character Based 的方法普遍用来进行较短文本、小规模的比较，会注意文本中各字符的顺序和位置；Term Based 的方法则更适合用来进行较长文本或大规模的比较，而且通常会抛弃文本中各个单元(通常是词)之间的顺序、位置信息 —— 一般的做法是用文本中的词组成的向量来表示文本，也就是所谓的“向量空间模型(Vecter Space Model, VSM)”。

**Character Based Methods**

1. 最长公共子序列(Longest Common Subsequence, LCS)

LCS 方法通过计算出两个字符串/序列之间的最长公共子序列，并使用这个子序列的长度来反映两个字符串/序列之间的相似程度。

1. 编辑距离(Leveinshtein Distance)

在两个字符串的比较中应用编辑距离时，通常会有一者作为”标准字符串“，另一者则作为”可能错误“的字符串，通过将后者变换成前者所进行的字符的 插入 、 删除 或 替换 操作次数作为衡量两者差异程度的指标。

扩展的编辑距离(Damerau-Levenshtein Distance)

扩展的编辑距离在思想上与编辑距离一样，只是除插入、删除和替换操作外，还支持 相邻字符的交换 这样一个操作，增加这个操作的考虑是人们在计算机上输入文档时的错误情况中，因为快速敲击而前后两个字符的顺序被输错的情况很常见。

1. Needleman-Wunsch Similarity

该方法被广泛运用于生物信息学中的序列比对，如氨基酸序列比对、核苷酸序列比对等。其基本思路与编辑距离相近，但在编辑距离中，三种不同的错误情况是平等的，而在生物信息学中，序列中的单元缺失情况比错误(位置匹配但内容不同)情况更不能容忍，因此在 4. Needleman-Wunsch 方法中，插入错误和删除错误会被赋予较高的惩罚分数。

Smith-Waterman Similarity

Smith-Waterman 方法用于生物信息学中的序列比对，但与 Needleman-Wunsch 方法不一样，它是一个 局部最优比对 方法，简单来说，它的目的是找出两个序列之间 连续且相同 的子序列。

5. Jaro Similarity 和 Jaro-Winkler Similarity

Jaro 方法和 Jaro-Winkler 方法考虑两个字符串之间相同字符的顺序位置和个数，只适用于像人名这样的较短字符串之间的比较。其中 Jaro-Winkler 方法是对 Jaro 方法的改进，而 Jaro 方法现在已经不常用。

6. Hamming Distance

Hamming 距离用于 长度相同 的序列之间的比较，思想非常简单，就是逐位比较得到的不同次数。Hamming 距离被广泛应用于信息学。

**Term Based Methods**

Term Based 方法中的 term 不一定是词，也可以是关键词、短语，当选定”词“作为 term 时，可以用 Bag-of-Words Model(BOW) 来更精确地描述此时的模型。为了表述上的方便，本系列文章将此处的 term 视为词。

1. Cosine Similarity

余弦相似度建立在用向量表示文档的前提上。两个文档的向量，同一个维度应该是表示的同一个词，而每一个维度的值，一般是用”词频-逆文档频率(Term Frequency-Inverse Document Frequency, TF-IDF)“ 来表示。

建立向量表示后，通过计算两个向量之间的夹角大小来衡量两个文档之间的近似程度，由于夹角的余弦值的计算方便，而且天然地处在 0 和 1 之间，故一般是用夹角的余弦值而不是夹角的大小来作为度量。

1. Dice's Coefficient 和 Jaccard Similarity

Dice 系数和 Jaccard 相似性起初被用于生态学上，作为一种判断物种间相似性的方法。在生态学上，要比较两个物种间相似程度时，通常会对该物种的特性进行采样，最后得到各自的特性集合，而 Dice 系数和 Jaccard 相似性都是通过比较两者之间的 共有特性 占比来度量相似性的，因此这两种方法都不是很关心每个 "Term" 的具体量，只是关心有没有某个 "Term"。

#### Corpus Based Methods

Corpus Based 方法通过对大量文档的统计分析得到语义上的相似，这里 “大量文档” 就是 Corpus 即语料了。

**Language Model**

常见的语言模型(Language Model, LM)都基于马尔可夫假设(Markov Assumption)，即认为语言中每个词只与其前面长度为 N-1 个词有关 —— 这 N-1 个词其实就构成了该词的上文，同时由于每个词都会成为其他词的上文，下文信息也会得到表现。也就是说，语言模型统计的其实是语言中每个词的一定程度的上下文情况。基于语言模型的这个特点，以及下面这样一个假设:

**具有相同(或相近)上下文的词，其语义是相近的**。

就可以用语言模型来进行文本的相似度量了。具体一点，有两种方式:

通过语言模型来发现同义词、近义词，来弥补 Term Based Methods 的缺陷；

扩大 "term" 的表示范围，比如说按照词组来进行统计，甚至按照句子来进行统计，那么就可以反映词组之间的相似性、句子之间的相似性了。

**Topic Model**

主题模型与向量空间模型有共同的一点是也是基于 BOW 模型的，也就是说，并不像语言模型一样考虑词与词之间的顺序、位置，而只是通过词与词的共现(Co-occurrence)来反映词与词之间的相似性。

主题模型是一种无监督的方法，与聚类方法在外在表现上会有一定的相似性，但它们各自的内在原理是相差较大的。

未完待续。。。

## 基于时间和位置的推荐

1. 时间协同过滤[[10]](#footnote-10)

时间衰减模型，常用的衰减函数是一个指数函数

衰减率是自定义参数，用于调节时间的重要性。较大的表示时间越久，评分的重要性越低。

在预测阶段，时间下用户u的物品j的评分：

这里代表距离用户u对物品j的评分最近的k近邻。上述公式和传统的协同过滤最本质的区别在于预测函数中存在权重。这些权重会减低很久之前产生的评分，最终推荐方案也就更偏向最近的评分物品。

1. 位置感知推荐系统

位置感知推荐系统可以视为上下文感知推荐系统的特殊情况。

# 协同过滤

根据上一章的相似度计算可以发现，到目前为止并没有利用任何关于用户和物品的相关信息和特征。实际上，只要用户数量足够多，通过评分计算的相似物品推荐效果就已经相当好了。协同过滤（Collaborate Filer）算法正是利用了这一集体智慧的方法。

协同过滤算法（collaborate filter，下文简称CF）在推荐系统发展初期占据领导地位，即使在今天日益庞大的推荐系统架构中，CF算法也是必不可少的。CF算法主要分为两类，一类是基于领域的方法(neighborhood methods)也称为基于内存的（Memory\_Based Method），另一类是隐语义模型(latent factor models)，后者一个最成功的实现就是矩阵分解(matrix factorization)。

## 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。

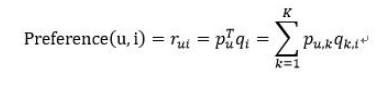
基于物品的协同过滤（item-based CF）：计算物品之间的行为或特征相似度，再进行推荐。比如，给购买过住院医疗险的客户推荐重大疾病险，因为这两种保险同现频率很高。此处的物品指的就是保险产品。

## Model-Based算法

### 隐因子模型（Latent Fator Model）

协同过滤的思路就是基于用户和物品的交互行为，要么计算用户间的相似度，推荐相似度高的用户喜欢的物品，因为这两个用户可能兴趣相投；要么就是计算物品间的相似度，推荐和历史记录相似度很高的物品，因为他们可能属于同一类别的商品。我们做决策的基础都是默认了商品是有类别的，可能有的用户都喜欢某一类商品，所以这些用户之间相似度高，可能有的商品是属于同一类别的，因此这些商品的相似度很高。那既然这样，有没有可能直接得到商品的类别呢？这样我们就可以直接根据类别去进行推荐了~

LFM就是基于这样的想法，假设商品存在若干个种类，那么每个用户对每个类会有一个兴趣度，同样的，每个类内又有若干种商品，每个商品在这个类内又会有一个对应的权重。这样，对于任何一个用户-商品对，我们都可以用下述公式来表达



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。

下一步就是去计算矩阵P和Q，最常见的方法就是在训练集上不停迭代更新参数直至参数收敛。既然需要迭代，那我们得有一个损失函数或者目标函数作为我们迭代的依据，这里我们选用的是最常见的预测值和实际值差值的平方和，并且加上了正则项防止过拟合，具体如下

[[11]](#footnote-11)

熟悉机器学习的同学应该很清楚，接下来就是基于梯度下降去优化这个损失函数，这里就不赘述了。但是我们还有一个问题，对于显式反馈的数据，我们直接用评分数据作为训练集，但是对于隐式反馈而言，只有正样本，我们可以把这些数据标注为1，但同时我们也需要负样本，所以往往需要我们从所有样本里选择样本构建负样本集。这里在构建负样本的时候有一个小小的trick，要尽量选择那些非常热门但是用户却没有发生交互的物品，因为这样构建的样本往往更具有代表性，也有利于我们的training。在movielens数据集中，我们以样本出现的次数作为权重，随机选择样本构建负样本集，实现如下

def Random\_Negative\_Sampling(self, items):

ret = {}

for i in items:

ret[i] = 1

count = 0

for i in range(0, len(items)\*5):

item = self.items\_pool[random.randint(0, len(self.items\_pool)-1)]

if item in items:

continue

ret[item] = 0

count += 1

if count > 2\*len(items):

break

return ret

如何去计算一个用户对一个物品的喜爱程度？是不是直接用上面那个公式，用用户对类的喜爱程度和该物品在此类中的权重乘积的累加呀。那么恭喜你，如果这样做，效果非常不好，一是太难收敛，二是预测效果总是不好。为啥呢？实际预测的矩阵里只有0和1两个值，而Preference的计算则是一个实数，在梯度下降的时候要用到这两者的差值，而这个值变动的范围太大，一方面导致你基本很难收敛，另一方面超参数的选择是个非常头疼的问题。所以，在计算喜好度的时候，一定要给结果加Sigmod函数。

这样把你的结果约束到0和1之间，防止在计算误差的时候出现太大的波动从而影响收敛。这里我在movielen的数据集上也进行了实验，《推荐系统实战》那本书里选择的隐类个数为100，本来是想复现他的结果，不过我发现算起来实在是太慢了，所以我这里选择隐类F=10，学习率alpha=0.03，正则项稀疏lambda=0.01，在所有的用户上迭代了30轮次，最后在测试集上的表现如下

召回率6.91% 精确率 23.2% 覆盖率 40.8% 流行度 5.20。

按照常理和书上所说，基于学习的这种方法的表现应该比基于统计的协同过滤表现要好点，但是这里的结果明显比之前的最好结果要差些，主要原因可能是这里的隐类选的太少了，书中的实验实际选择的隐类数目为100。但我发现隐类100的时候不仅每一轮计算的速度大大提升，并且收敛的速度也远远慢于隐类数目较少的时候，这也很容易理解，毕竟对应的需要优化的参数数目多了10倍，因此需要的训练样本也要更多，收敛自然更慢，有兴趣的或者有时间的同学可以去尝试一下。

然后在整个实现LFM的过程中，还有其他几个小地方大家可以注意一下

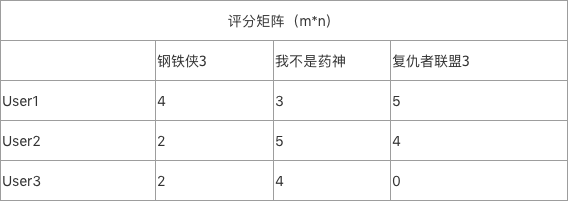
参数的初始化。所有参数初始化为0好像不是一个好选择，总是无法收敛。一般的做法都是选择在-1到1之间的随机数。

构建负样本和正样本的比例。书中说这是一个影响模型效果很重要的参数，实际上可以这么理解，相同轮次的迭代，负样本比例越高，其实越多的训练数据，自然模型的表达能力可能会更好。

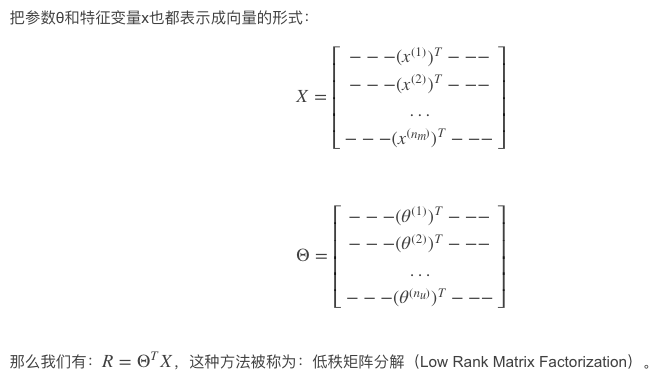
超参数的选择。在实际使用的过程中，我发现超参数的选择对于LFM模型的表现真实至关重要，学习率过高直接导致没法收敛，但是太低了收敛的速度又太慢了，所以超参数的选择真的是一个非常头疼的事。

LFM的计算速度是真的很慢。每一轮迭代，要在每个用户上迭代其所有的行为记录和我们构造的负样本，导致计算真的非常耗时。而隐类个数的增加不仅导致需要优化的参数增加，对于数据的需求同样提高，这样直接导致收敛速度变得非常非常慢。

最后是对LFM算法的一些思考，它的本质其实就是矩阵分解，将之前的用户物品矩阵分解成了用户-隐类矩阵和隐类-物品矩阵，然后目标是让这两个矩阵的乘积与原矩阵的残差尽可能得小，同时引入了一些正则项防止过拟合。后来我发现其实LFM算法就是FunkSVD算法，是典型的基于矩阵分解的方法，后来又出现了基于它进行改进的方法。



将表格中的内容转化成一个m\*n的矩阵R:



由此找电影i相似的电影j：可以计算两个特征向量的距离，其中距离最小的就是最相似的电影

把右上表中的第二列（col=《我不是药神》）和第三列（col=《复仇者联盟3》）分别与

右下表的第二列（col=User2）对应元素相乘后相加（即点乘），得到的结果分别是（55 + -40=25）和（-25 + 50=-10），

这两个分数的物理意义分别是李四喜欢《我不是药神》和李四喜欢《变形金钢4》的程度，

可以看出李四更喜欢《我不是药神》。那么我们把右上表的每列和右下表的每列各元素相乘后相加，

是不是可以得到每个人喜欢每部片子的程度呢？这不就是我们左边Rating matirx图表示的意思嘛[[12]](#footnote-12)！

我们把评分矩阵（Rating Matrix）计作V， 𝑉∈𝑅𝑛×𝑚，那么V的每一行𝑉𝑖代表一个人的所有评分，每一列𝑉𝑗代表某一部电影所有人的评分，𝑉𝑖𝑗代表某个人i对某部电影j的评分。对应电影推荐来说，V必定是稀疏的，因为电影数量（列的数目）是巨大的，V中必定有很多很多项为null。

我们接下来看这两个矩阵U（Users Features Matrix ）和M（Movie Features Matrix）。U为用户对特征的偏好程度矩阵，M为电影对特征的拥有程度矩阵。𝑈∈𝑅𝑓×𝑛的每一行表示用户，每一列表示一个特征，它们的值表示用户与某一特征的相关性，值越大，表明特征越明显。矩阵𝑀∈𝑅𝑓×𝑚，的每一行表示电影，每一列表示电影与特征的关联。

那么U和M怎么得到呢？

还记得SVD的公式嘛？𝑉=𝑈𝑀𝑇，其实公式右边的中间还有个对角矩阵S，我们可以把他看成跟U相乘合并（The S matrix is left blended into the feature vectors, so only U and M remain）。其实，U和M这两个矩阵是通过学习的方式得到的，而不是直接做矩阵分解。我们定义如下的损失函数：

𝐸=12∑𝑛𝑖=1∑𝑚𝑗=1𝐼𝑖𝑗(𝑉𝑖𝑗−𝑝(𝑈𝑖,𝑀𝑗))2+𝑘𝑢2∑𝑛𝑖=1‖𝑈𝑖‖2+𝑘𝑚2∑𝑚𝑗=1‖𝑀𝑗‖2(1)

这里的评价指标metric采用的是RMSE。其中𝑝(𝑈𝑖,𝑀𝑗)代表用户i对电影j的预测，最常用的预测函数p就是点乘，即 𝑝(𝑈𝑖,𝑀𝑗)=𝑈𝑇𝑖𝑀𝑗。式(1)中的 𝐼∈{0,1}𝑛×𝑚，为一个指示器，指示相应位置是否有评分，有评分为1，没有为0。等式右边最后两项是正则化项，防止过拟合，这里不过多展开。

到这里，已经变成了一个机器学习的常见问题了，即最小化损失函数，用得最多的优化方法就是梯度下降，当然还有很多梯度下降的变体。下图是简单的梯度下降算法。

#### 常见的LFM算法

SVD奇异值分解

这里的SVD推荐本质上是model-based，跟传统数学意义的SVD没有太大关系，只不过借鉴了SVD分解R=U∗S∗VR=U∗S∗V这个形式，通过最优化方法进行模型拟合，求得R=U∗VR=U∗V。

BiasSVD。在FunkSVD的基础上又引入了平均得分、用户偏置项和物品偏置项，相比于FunkSVD又多了两个需要优化的参数，但优化方法啥的都是一样的。事实表明，由于考虑了这些偏置项，令BiasSVD在某些场景下变现会优异很多。

SVD++。在BiasSVD的基础上又引入了用户的隐式反馈，也就是用户之前如果对该物品产生过交互，则对它的评分要进行修正。结构越来越复杂，计算也会越来越慢，虽然说SVD++在竞赛中大放异彩，但实际应用比较困难，计算代价太大，无法做到实时推荐，而且可解释性也不强。

#### 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

### 基于矩阵分解的LFM

#### LibMF

矩阵分解通常用于显示反馈数据的推荐系统。

LIBMF使用潜在空间中两个矩阵的乘积，近似得到一个不完全矩阵。它可以解决实值矩阵分解、二元矩阵分解和一类矩阵分解问题。我们使用LibMF进行矩阵分解和评分预测，主要考虑到它的运算速度相当快。一方面支持在多核机器中并行计算，并通过CPU指令加速向量运算；另一方面，如果数据量过大，可以进行磁盘训练，减少内存使用。

**安装：**

下载libmf-2.01的安装文件[[13]](#footnote-13)。解压后进入目录：

执行make命令，发现目录下多了mf-train和mf-predict这两个可执行文件。

图片包含 物体

描述已自动生成

**数据格式**

<row\_idx> <col\_idx> <value>

在demo目录中，文件real\_matrix.tr.txt' 和 `real\_matrix.te.txt'是真值矩阵分解real-valued matrix factorization (RVMF)演示的训练和[测试](http://lib.csdn.net/base/softwaretest)数据集。二元矩阵分解binary matrix factorization (BMF)中，`binary\_matrix.tr.txt' 和`binary\_matrix.te.txt.'中<value>集是{-1, 1}。在一类矩阵分解（one-class MF）中，所有的<value>都是正的。

**模型格式**

LibMF把一个训练矩阵R变为一个k-by-m的矩阵`P'和一个k-by-n的矩阵`Q'，也就是R近似于P'Q。训练过程结束后，这两个因子矩阵P和Ｑ被存到一个模型文件中。这个文件以如下打头：

`f': f表示MF矩阵分解问题的损失函数，

`m': 矩阵R行的数目,  
   `n': 矩阵R列的数目,  
   `k': 隐因子的数目,  
     `b': 矩阵R所有元素的平均值。

从第五行开始，Ｐ和Ｑ的列就被一行接一行的存储。每一行，都有两个领导标志跟在一列值后面。第一个标志是被存储列的名字，第二个标志表明值的类型。如果第二个标志是‘T’，列是真值。否则，列的所有值是NaN。举个例子：

并且b=0.5，则模型文件的内容是：

    --------model file--------  
       m 3  
       n 2  
       k 3  
       b 0.5  
       p0 T 1 3 5  
       p1 F 0 0 0  
       p2 T 2 4 6  
     q0 T -1 -3 -5  
       q1 T -2 -4 -6  
       --------------------------

**命令使用**

**mf-train**

用法: mf-train [options] training\_set\_file [model\_file]

参数选择：

-l1 <lambda>,<lambda>: set L1-regularization parameters for P and Q.

(default 0) If only one value is specified, P and Q share the same

lambda.

-l2 <lambda>,<lambda>: set L2-regularization parameters for P and Q.

(default 0.1) If only one value is specified, P and Q share the same

lambda.

-f <loss>: 指定损失函数 (default 0)

for real-valued matrix factorization

0 -- squared error (L2-norm)

1 -- absolute error (L1-norm)

2 -- generalized KL-divergence (--nmf is required)

for binary matrix factorization

5 -- logarithmic error

6 -- squared hinge loss

7 -- hinge loss

for one-class matrix factorization

10 -- row-oriented pair-wise logarithmic loss

11 -- column-oriented pair-wise logarithmic loss

-k <dimensions>: 设置维度的数量，也就是隐因子数量 (default 8)

-t <iter>: 设置迭代次数 (default 20)

-r <eta>: 设置初始学习率(default 0.1)

-s <threads>: set number of threads (default 12)

-n <bins>: set number of bins (may be adjusted by LIBMF for speed)

-p <path>: set path to the validation set

-v <fold>: set number of folds for cross validation

--quiet: quiet mode (no outputs)

--nmf: perform non-negative matrix factorization

--disk: perform disk-level training (will create a buffer file)

“mf-train”是LibMF最主要的训练命令。每次迭代，下列信息都被打印出来：

- iter: 迭代的索引   
  - tr\_xxxx: xxxx is the evaluation criterion on the training set  
    - va\_xxxx: the same criterion on the validation set if `-p' is set  
    - obj: objective function value

这里的`tr\_xxxx' 和 `obj' 都是估计的，因为计算真的值太耗时间了。  
对于不同的损失，标准如下：

        <loss>: <evaluation criterion>  
        -       0: root mean square error (RMSE)  
        -       1: mean absolute error (MAE)  
        -       2: generalized KL-divergence (KL)  
        -       5: logarithmic loss  
        -   6 & 7: accuracy  
        - 10 & 11: pair-wise logarithmic loss (BprLoss)

**mf-predict**

用法：mf-predict [options] test\_file model\_file output\_file

options:

-e <criterion>: set the evaluation criterion (default 0)

0: root mean square error

1: mean absolute error

2: generalized KL-divergence

5: logarithmic loss

6: accuracy

10: row-oriented mean percentile rank (row-oriented MPR)

11: colum-oriented mean percentile rank (column-oriented MPR)

12: row-oriented area under ROC curve (row-oriented AUC)

13: column-oriented area under ROC curve (column-oriented AUC)

**案例**

默认参数训练一个模型：

> mf-train real\_matrix.tr.txt model

训练具有正则化系数的模型，系数如下：

> mf-train -l1 0.05 -l2 0.01 real\_matrix.tr.txt model

coefficient of L1-norm regularization on P = 0.05

coefficient of L1-norm regularization on Q = 0.05

coefficient of L2-norm regularization on P = 0.01

coefficient of L2-norm regularization on Q = 0.01

另一组正则化系数，训练模型：

> mf-train -l1 0.015,0 -l2 0.01,0.005 real\_matrix.tr.txt model

coefficient of L1-norm regularization on P = 0.05

coefficient of L1-norm regularization on Q = 0

coefficient of L2-norm regularization on P = 0.01

coefficient of L2-norm regularization on Q = 0.03

> mf-train -f 5 -l1 0,0.02 -k 100 -t 30 -r 0.02 -s 4 binary\_matrix.tr.txt model

直接预测，输出评分到指定文件：

> mf-predict real\_matrix.te.txt model output

预测并输出MAE值：

> mf-predict -e 1 real\_matrix.te.txt model output

#### LibFM

这一类模型的共同特点是矩阵分解。即对用户-物品评分矩阵分解成若干个小矩阵，目的是分解之后的矩阵乘积接近原始矩阵，于是也实现了对原始矩阵为空的值的预测。

在这些方法中，比较重要的几个参数有：隐特征个数，随机梯度下降中的学习率，正则化参数，总迭代次数等。具体在每个方法中这些参数的最优值也不尽相同。

### 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

相关概念：

数据集：一般是购物篮数据

频繁模式：频繁地出现在数据集中的模式，列如项集，子结构和子序列等等

挖掘目标：频繁模式，频繁项集，关联规则

关联规则：比如牛奶 => 鸡蛋（支持度=2%，置信度-60%）

支持度：分析的数据集中的2%同时购买了牛奶和鸡蛋

支持度是两件商品（AB）在总销售笔数（N）中出现的概率，即A与B同时被购买的概率。

置信度：购买了牛奶的顾客有60%也购买了鸡蛋

置信度是购买A后再购买B的条件概率。简答来讲就是交集部分C在A中的比例，比例越大说明购买A的客户很大期望会购买B商品。

项集：商品或者品类的集合

k-项集：k个项组成的集合

频繁项集：满足最下支持度的项集，频繁k-项集一般记为Lk

强关联规则：满足最小支持度阈值和最小置信度阈值的规则

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。

获得的结果需要进行规则过滤：

互斥的商品，例如同类的自行车，汽车，内容相近的书籍，此时不能使用套餐推荐

        推荐相近商品的时候使用浏览记录，使用购买记录更类似于关联规则挖掘

        考虑兴趣的时效性，例如已经购买了某种自行车，就没必要再向用户推荐相近的自行车

总结一下：当我们推荐相近商品的时候，最好可以使用浏览记录来进行推荐，使用基于物品的协同过滤算法；

如果要推荐套餐的话，最好是基于购买记录，购物篮分析，用关联模式挖掘（关联规则挖掘）。

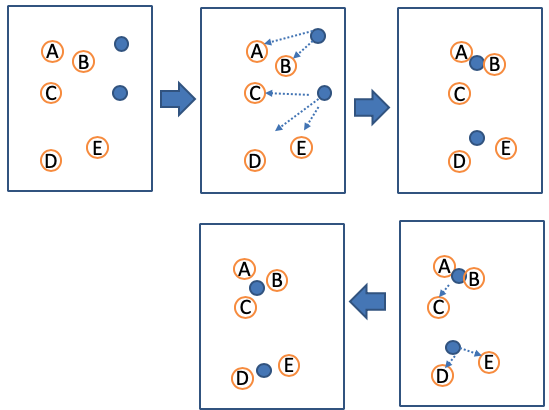
下面使用 FP-Growth 算法来找出高频推荐的电影。

**1**

该算的描述可在 Han 等人的论文“Mining frequent patterns without candidate generation”中找 **2** 到。FP 代表 frequent pattern，即高频模式。给定若干交易记录，FP-Growth 的第一步是计算物品 的频率并标示高频物品。第二步是利用后缀树(suffix tree，也称为 FP-tree)来编码各交易;该 过程不会显式生成推荐的备选集合，在大数据集上对应的计算量通常很大。

### 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。

# 基于标签的推荐系统

**标签库**

标签是联系用户与物品、内容以及物品、内容之间的纽带，也是反应用户兴趣的重要数据源。标签库的最终用途在于对用户进行行为、属性标记。是将其他实体转换为计算机可以理解的语言关键的一步。

标签库则是对标签进行聚合的系统，包括对标签的管理、更新等。

一般来说，标签是以层级的形式组织的。可以有一级维度、二级维度等。

标签的来源主要有：

* 已有内容的标签
* 网络抓取流行标签
* 对运营的内容进行关键词提取

对于内容的关键词提取，使用[结巴分词](https://github.com/fxsjy/jieba) + [TFIDF](http://www.ruanyifeng.com/blog/2013/03/tf-idf.html)即可。此外，也可以使用[TextRank](http://www.tuicool.com/articles/UZ77Z3)来提取内容关键词。

这里需要注意的一点是对于关联标签的处理，比如用户的标签是足球，而内容的标签是德甲、英超，那么用户和内容是无法联系在一起的。最简单的方式是人工设置关联标签，此外也可以使用word2vec一类工具对标签做聚类处理，构建主题模型，将德甲、英超聚类到足球下面。

给用户推荐标签，可以提高标签的质量，并且方便用户输入标签。

**基于物品标签的推荐系统**

豆瓣利用标签将用户的推荐系统结果做了聚类，显示了对不同标签下用户的推荐结果，从而增加了推荐的多样性和可解释性。

简单算法：

1. 统计每个用户最常用的标签、

2. 对于每个标签，统计被打过这个标签次数最多的物品。

3. 找到他常用的标签，然后找到具有这个标签的最热门物品推荐给这个用户。

利用用户的标签向量对用户兴趣建模，其中 每个标签都是用户使用过的标签，而标签的权重是用户使用该标签的次数。

可以对热门标签 和 热门物品进行惩罚，提高推荐物品的新颖性。

**基于用户标签 -- 群体的相似计算**

用户画像是目标用户的标签化表征，其中有些标签是定量的，有些标签是定性的。需要结合定量相似度和定性相似度来计算。

对不同的标签加权方法：

**定量相似度**

<user\_id, label\_id, label\_value> label\_id 取标签，label\_value取实际值

将不同标签取值数据统一映射到[0,5]区间。常用归一化的方法有线性函数转换，数函数转换，反正切函数转换。

报错属性（交易属性）相似度

**定性相似度**

<user\_id, label\_id, label\_value> label\_id 取二级标签类别，label\_value按照标签个数取值并归一化为[0-5]

包括属性（基础信息）相似度

所有的标签都属于定性相似度计算

**综合相似度**

最终的相似度为各标签的线性加权和。

**实验**

定量标签之间归一化到【0-5】

方法1： 定性相似度将标签对应二级类别下的Int类型，然后再归一化【0-5】

|  |  |  |  |
| --- | --- | --- | --- |
| user\_id | 用户ID |  | 归一化 |
| sex | 性别 | 1男， 2女， 0未知 | 归一化 |
| age | 年龄 |  | 归一化 |
| ID\_type | 证件类型 | 身份证，军官证， 护照， 台胞证，其它 | 不用 |
| ID\_number | 证件号码 | 注意ID是大写 | 不用 |
| birthday | 出生日期 |  | 不用 |
| lunar\_birthday | 出生日期(农历) |  | 不用 |
| occupation | 职业 | 学生， 公务员，教师，工程师，工人，农民，其它 | 不用 |
| education | 学历 | 小学， 初中， 高中/中专，大专/本科，研究生及以上 | 定性 |
| marital\_status | 婚姻 | 单身，结婚，离婚，丧偶，其它 | 定性 |
| income | 年收入 | 5万以下， 5万-10万， 10-20万， 20万以上 | 定性 |

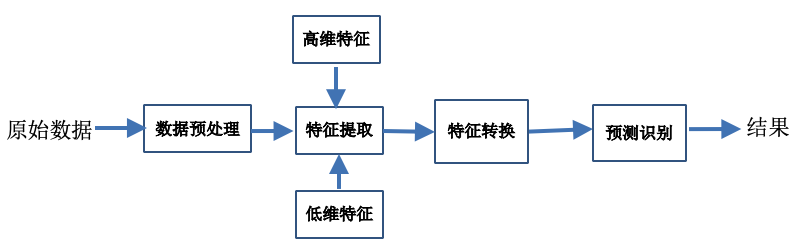
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 年龄 | 110101 | 11010101 | 0-20岁 | 1 |
| 年龄 | 110101 | 11010102 | 21-30岁 | 2 |
| 年龄 | 110101 | 11010103 | 31-40岁 | 3 |
| 年龄 | 110101 | 11010104 | 41-50岁 | 4 |
| 年龄 | 110101 | 11010105 | 51-60岁 | 5 |
| 年龄 | 110101 | 11010106 | 61岁以上 | 6 |

方法2： 所有的标签的打分都为1，

比如<user\_id, 莘莘学子label\_id, 1> => <0101, 11010501,1>

# 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。



## Spark分类算法[[14]](#footnote-14)

Spark中常见的3种分类模型:线性模型、决策树和朴素贝叶斯模型。线性模型 相对简单，而且相对容易扩展到非常大的数据集;决策树是一种强大的非线性技术，训练过程计 算量大并且较难扩展(幸运的是，MLlib 会替我们考虑扩展性的问题)，但是在很多情况下性能 很好;朴素贝叶斯模型简单、易训练，并且具有高效和并行的优点(实际中，模型训练只需要遍 历整个数据集一次)。当采用合适的特征工程时，这些模型在很多应用中都能达到不错的性能。朴素贝叶斯模型还可以作为一个很好的模型测试基准，用于度量其他模型的性能。

目前，Spark 的 MLlib 库提供了基于线性模型、决策树和朴素贝叶斯的二分类模型，以及基 于决策树和朴素贝叶斯的多类别分类模型。本书为了方便起见，将关注二分类问题。

### 线性模型

线性模型的核心思想是对样本的预测结果(通常称为目标变量或者因变量)进行建模，即对输入变量(特征或者自变量)应用简单的线性预测函数。

其中y是目标变量，w 是参数向量(也称为权重向量)，x 是输入特征向量。(wTx) 是关于权重向量 w 和特征向量 x 的线性预测器(又称向量点积)。对这个线性预测器，我们应用了一个函数 f (又称连接函数)。

实际上，通过简单改变连接函数 f，线性模型不仅可以用于分类还可以用于回归。标准的线 性回归(见下一章)使用对等连接函数(identity link，即直接使用 y = fwT x )，而二分类使用上 面提到的连接函数。

举一个在线广告的例子。如果网页中展示的广告没有被点击，则目标变量标记为0；如果发生点击，则目标变量标记为1。每次点击的特征向量有点击事件的变量组成（比如与用户、网页、广告和广告客户相关的特征，以及与事件场景相关的其他因素，比如设备类型、事件、地理位置等等）。

于是，我们需要训练一个模型，将给定输入的特征向量映射到预测的输出（点击或者未点击）。对于一个新的数据点，我们将得到一个新的特征向量（此时还不知道预测的目标变量），并将其与权重向量进行点积。然后对点积的结果应用连接函数，最后函数的结果便是预测的输出。

给定输入数据的特征向量和目标变量，我们想要找到能够对数据进行最佳拟合的权重向量，拟合的过程即最小化模型输出与实际值的误差。这个过程称为模型的拟合、训练或者优化。具体来说，我们需要找到一个权重向量，它能够最小化所有训练样本的由损失函数计算出来的误差之和。损失函数的输入是给定训练样本的权重向量、特征向量和实际输出，而输出是误差值。实际上。损失函数也被定义为连接函数，每个分类或回归函数会有对应的损失函数。

MLlib提供的两个二分类模型的损失函数，第一个是logistic损失（logistic loss）等价于logistic回归模型。第二个是合页损失（hinge loss），等价于线性支持向量机（SVM）。这里的SVM严格来说并不属于广义线性模型的统计框架，但是当指定损失函数和连接函数时使用方法上相同。

1. Logistic回归

Logistic回归是一个概率模型，也就说该模型预测结果的值域为[0,1]。对于二分类来说logistic回归的输出等价于模型预测某个数据点属于正类的概率估计。Logistic回归是线性分类模型中使用最为广泛的一个。Logistic回归使用的连接函数为logistic连接：

Logistic回归的损失函数是logistic损失：

其中y是实际的输出值（正类为1，负类为-1）。

1. 线性支持向量机

SVM在回归和分类方面是一种强大且流行的技术。和logistic回归不同，SVM并不是概率模型，但是可以基于模型对正负的估计预测类别。

SVM的连接函数时一个对等连接函数，因此预测的输出为：

因此，当的估计值大于等于阈值0时，SVM将数据点标记为1，否则标记为0。

SVM的损失函数被称为合页损失，定义为：

SVM是一个最大间隔分类器，它试图训练一个使得类别尽可能分开的权重向量。在很多分类任务中，SVM不仅性能突出，而且在大数据集上的扩展线性的。

### 朴素贝叶斯

朴素贝叶斯是一个概率模型，通过计算给定数据点属于某个类别的概率来进行预测。朴素贝叶斯模型假定各个特征之间对分类的影响相互独立(假定各个特征之间条件独立)。

基于这个假设，属于某个类别的概率表示为若干概率乘积的函数，其中这些概率包括某个特 征在给定某个类别的条件下出现的概率(条件概率)，以及该类别的概率(先验概率)。这样使得 模型训练非常直接且易于处理。类别的先验概率和特征的条件概率可以通过数据的频率估计得 到。分类过程就是在给定特征和类别概率的情况下选择最可能的类别。

另外还有一个关于特征分布的假设，即参数的估计来自数据。MLlib 实现了多项朴素贝叶斯 (multinomial naive Bayes)，其中假设特征分布是多项分布，用以表示特征的非负频率统计。

上述假设非常适合二元特征(比如 k 之一，k 维特征向量中只有 1 维为 1，其他为 0)，并且 普遍用于文本分类(第 4 章中介绍的词袋模型是一个典型的二元特征表示)。

### 决策树

决策树是一种强大的非概率模型，它可以表达复杂的非线性模式和特征相互关系。决策树在 很多任务上表现出的性能很好，相对容易理解和解释，可以处理类别特征和数值特征，同时不要 求输入数据归一化或者标准化。决策树非常适合应用集成方法，比如多个决策树的集成(称为决策树森林)。

决策树模型就好比一棵树，叶子代表值为 0 或 1 的分类，树枝代表特征。下图展示了一棵简 单的决策树，二元输出分别是“待在家里”和“去海滩”，特征则是天气。

决策树算法是一种自上而下的、始于根节点（或特征）的方法，在每一个步骤中通过评估特征分割的信息增益，选出分割数据集最优的特征。信息增益通过计算节点不纯度（节点标签不相似或不同质的程度）减去分割后的两个子节点不纯度的加权和。对于分类任务，有两种评估方法可用于选择最好的分割：基尼不纯度（Gini impurity）和熵（entropy）。

### 树集成模型

集成模型指将基础模型组合成一个模型。Spark支持两种主要的集成算法：随机森林和梯度提升树（GBDT）。

1. 随机森林

随机森林即决策树的集成，它由多个决策树组合而成。如决策树一样，随机森林能处理类别特征、支持多分类而且不需要特征缩放。

Spark MLlib 的随机森林算法同时支持二分类和多类别分类，以及连续型和类别型特征上的 回归。

1. 梯度提升树

梯度提升树是决策树的集成。它迭代地对决策树进行训练以最小化损失函数。它能处理类别性特征、支持多类别分类且不需要特征缩放。

3. 多层感知分类器

神经网络是一个复杂的自适应系统，它会借助各权重的变更而改变信息流，进而改变自己的 内部结构。针对多层神经网络的权重优化过程也称为反向传播(backpropagation)。反向传播超出 了本书讨论范围，另也涉及激活函数和基本的微积分知识。

多层感知分类器(multilayer perceptron classifier)基于前向反馈(feed-forward)人工神经网 6 络。它由多个神经层构成，每层都与下一层全连接。其输入层的各节点对应输入数据。其他节点 都会对经该节点的输入、相应的权重和偏置(bias)进行线性组合，再应用一个激活函数(activation function)或连接函数后，映射为对应的输出。

## 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。

### 基于Spark的GBDT+LR模型实现

测试数据来源<http://archive.ics.uci.edu/ml/machine-learning-databases/adult/>[[15]](#footnote-15)

该模型利用Spark mllib的GradientBoostedTrees作为GBDT部分，因为ml模块的GBTClassifier对所生成的模型做了相当严密的封装，导致难以获取某些类或方法。而GradientBoostedTrees所需的训练数据为mllib下的LabeledPoint，所以下面的数据预处理的目标是将cat数据进行编码并生成LabeledPoint。

数据预处理

GBDT部分

GBDT+LR

### xgboost+LR

背景[[16]](#footnote-16)

xgboost+lr模型融合方法用于分类或者回归的思想最早由facebook在广告ctr预测中提出，其论文Practical Lessons from Predicting Clicks on Ads at Facebook有对其进行阐述。在这篇论文中他们提出了一种将xgboost作为feature transform的方法。大概的思想可以描述为如下：先用已有特征训练XGBoost模型，然后利用XGBoost模型学习到的树来构造新特征，最后把这些新特征加入原有特征一起训练模型。构造的新特征向量是取值0/1的，向量的每个元素对应于XGBoost模型中树的叶子结点。当一个样本点通过某棵树最终落在这棵树的一个叶子结点上，那么在新特征向量中这个叶子结点对应的元素值为1，而这棵树的其他叶子结点对应的元素值为0。新特征向量的长度等于XGBoost模型里所有树包含的叶子结点数之和。最后将新的特征扔到LR模型进行训练。实验结果表明xgboost+lr能取得比单独使用两个模型都好的效果。

原理介绍

“特征决定了所有算法效果的上限，而不同的算法只是离这个上限的距离不同而已。”所以如何更有效的提取有效的特征是机器学习中的一个hotspot，例如近几年来大火的深度学习方法中神经网络的层数不断增加其实质也是在探索如何更好地从原始数据中得到更为有效的特征表达。如果能够将数据表达成为线性可分的数据，那么使用简单的线性模型就可以取得很好的效果。

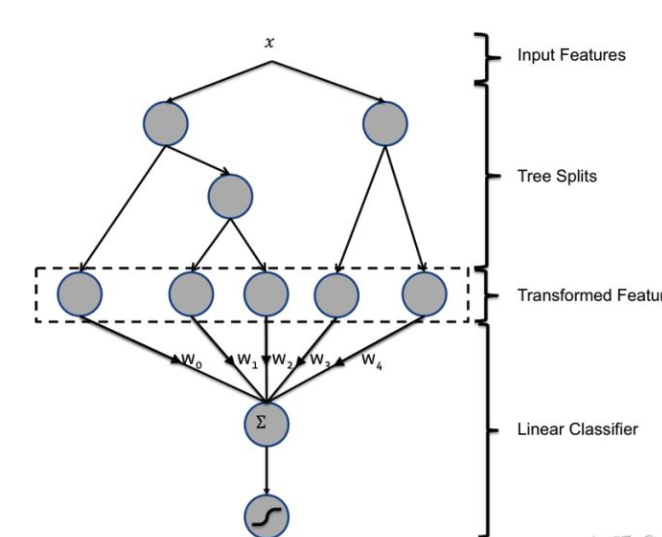
XGBoost 构建新的特征也是使特征更好地表达数据。它的核心思想是将boosting看作是一个将样本进行非线性变换的方法。处理特征变换的一般方法有：

对于连续的特征：一个简单的非线性变化就是将特征划分到不同的区域(bin)，然后再将这些区域的编号看作一个离散的特征来进行训练。这也就是俗称的连续变量离散化方法，这有非常多的方法可以完成这项事情。

对于离散的特征：我们可以直接对离散特征做一个笛卡尔积从而得到一系列特征的组合，当然有些组合是没用的，那些没用的组合可以删掉。

而这里利用boosting来对样本进行离散化的方法是另一种特征变换的方法。

下面的图中的两棵树是利用现有特征训练XGBoost学习到的，其中第一棵树有3个叶子结点，而第二棵树有2个叶子节点。对于一个输入样本点x，如果它在第一棵树最后落在其中的第二个叶子结点，而在第二棵树里最后落在其中的第一个叶子结点。那么通过XGBoost获得的新特征向量为[0, 1, 0, 1, 0]，其中向量中的前三位对应第一棵树的3个叶子结点，后两位对应第二棵树的2个叶子结点。下图为混合模型结构。输入特征通过增强的决策树进行转换。 每个单独树的输出被视为稀疏线性分类器的分类输入特征。 增强的决策树被证明是非常强大的特征转换。



这里我们可以这样理解，XGBoost是一种集成树模型，其本质是不同单个决策树的组合。而决策树是一种树形结构，又称为判定树，是运用于分类的一种树结构，其中的每个内部节点代表对某一属性的一次测试，每条边代表一个测试结果，叶节点代表某个类或类的分布。决策树的决策过程需要从决策树的根节点开始，待测数据与决策树中的特征节点进行比较，并按照比较结果选择选择下一比较分支，直到叶子节点作为最终的决策结果。

未完待续。。。

### xgboost4j-spark[[17]](#footnote-17)

xgboost4j-spark使得xgboost能够集成spark项目，运行在分布式集群中。

当前的xgboost4j-spark可以实现以下功能：

特征工程：特征提取，变换，降维和选择等。 管道：构造，评估和调整ML管道 持久性：持久并加载机器学习模型，甚至整个管道

如何使用XGBoost4J-Spark构建机器学习管道的端到端过程，需要了解以下内容： 使用Spark预处理数据以适合XGBoost / XGBoost4J-Spark的数据接口 使用XGBoost4J-Spark训练XGBoost模型 使用Spark服务XGBoost模型（预测） 使用XGBoost4J-Spark构建机器学习管道 在生产中运行XGBoost4J-Spark

添加Xgboost4j-Spark依赖包，如果你使用的是Maven进行项目管理，添加

<dependency>

<groupId>ml.dmlc</groupId>

<artifactId>xgboost4j-spark</artifactId>

<version>latest\_version\_num</version>

</dependency>

注：lastest\_version\_num 是当前具体的版本号，详细见<https://github.com/dmlc/xgboost/releases>

Xgboost4j-spark需要Spark的版本大于2.4

因为xgboost4j-spark高度集成了xgboost和spark，使用户可以通过便捷而强大的数据处理框架Spark在训练/测试数据集上应用各种类型的转换。 下面以鸢尾花(iris)数据集为例，展示如何使用Spark转换原始数据集并使之适合XGBoost的数据接口。

鸢尾花数据集是csv格式。每个实例包含4个特征，即“sepal lentgh”，“septal width”，“petal length”和“petal length”。此外，它包含“ class”列，该列实际上是带有三个可能值的标签：“ Iris Setosa”，“ Iris Versicolour”和“ Iris Virginica”。

读取数据集，并转换成DataFrame：

import org.apache.spark.sql.SparkSession  
import org.apache.spark.sql.types.{DoubleType, StringType, StructField, StructType}  
  
val spark = SparkSession.builder().getOrCreate()  
val schema = new StructType(Array(  
 StructField("sepal length", DoubleType, true),  
 StructField("sepal width", DoubleType, true),  
 StructField("petal length", DoubleType, true),  
 StructField("petal width", DoubleType, true),  
 StructField("class", StringType, true)))  
val rawInput = spark.read.schema(schema).csv(“input\_path”)

创建SparkSession作为DataFrame的入口，schema变量定义鸢尾花数据集的数据格式，这里通过csv读取文件。

为了适应xgboost，需要对原始iris数据集进行转换，比如String类型的列转换成Double类型，还有要将所有特征列组合成向量。

String类型转Double类型，可以使用Spark内置的StringIndexer类。

import org.apache.spark.ml.feature.StringIndexer  
val stringIndexer = new StringIndexer().  
 setInputCol("class").  
 setOutputCol("classIndex").  
 fit(rawInput)  
val labelTransformed = stringIndexer.transform(rawInput).drop("class")

如上，创建一个StringIndexer实例，设置String类型作为输入列，设置Double类型作为输出类型。然后使用fit将rawInput作为输入DataFrame。

定义好StringIndexer的转换逻辑之后，使用transform输入rawInput，会生成一个新的列ClassIndex，这里删除掉Class列。

类似的，也可以使用另外一种转换操作VectorAssembler，这个操作可以将多个特征组合在一起形成向量。

import org.apache.spark.ml.feature.VectorAssembler  
val vectorAssembler = new VectorAssembler().  
 setInputCols(*Array*("sepal length", "sepal width", "petal length", "petal width")).  
 setOutputCol("features")  
val xgbInput = vectorAssembler.transform(labelTransformed).select("features", "classIndex")

现在xgbInput这个DataFrame之后两列，分别是features和classIndex。前者是特征向量，后者是数值标签列。

处理缺失值

训练  
val *Array*(train, eval1, eval2, test) = xgbInput.randomSplit(*Array*(0.6, 0.2, 0.1, 0.1))

val xgbParam = *Map*("eta" -> 0.1f,  
 "max\_depth" -> 2,  
 "objective" -> "multi:softprob",  
 "num\_class" -> 3,  
 "num\_round" -> 100,  
 "num\_workers" -> 2,  
 "eval\_sets" -> *Map*("eval1" -> eval1, "eval2" -> eval2))  
val xgbClassifier = new XGBoostClassifier(xgbParam).  
 setFeaturesCol("features").  
 setLabelCol("classIndex")  
val xgbClassificationModel = xgbClassifier.fit(train)

预测

val results = xgbClassificationModel.transform(test)  
results.show()

处理缺失值

构建pipeline

未完待续。。。

## 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）

Learning To Rank(LTR)算法有三种类型，分别是pointwise、pairwise和listwise排序。

在协同过滤算法中，推荐本身可以视作一个评分预测系统。用户-物品的预测评分越高，该用户就越可能喜欢该物品。

而在实际推荐场景中，用户能够看到的页面中并不会显示评分，而是各个物品的排序列表。也就是说，推荐结果关注了用户对于物品的相关性而非评分。

PointWise

PointWise类似于协同过滤的方法，先为每个项目进行评分，然后在按照评分进行排序。评分预测与排序的区别在于，用户不用关心物品的实际评分，只要能够体现物品的排序位置就行。

PairWise

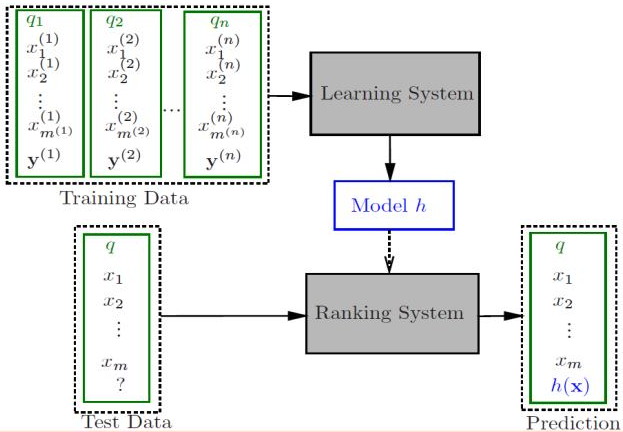
PairWise是二分类的一种方法。它输入两个物品，返回两个物品的顺序。如果得到所有物品两两组合的顺序，最终就能确定所有物品的排序结果。

该方法与PointWise的本质不同在于，pairwise并不需要预测所有商品的评分。 ListWise Listwise是所有LTR中终极方法，它根据整个排名列表并进行优化。listwise排序基于这样一个假设，物品排序列表顶部的顺序比底部的顺序更重要。这也是listwise的一大优势，因为pointwise和pairwise都不能区分某个物品在排名列表中的位置。

比如，一共推荐Top10的商品。pairwise推荐过程中，排名靠前的两个商品的顺序与排名最后的两个商品的位置重要度一样。从直接上将，这显然不符合实际情况，众所周知，用户更多的只会关心列表顶部的商品。基于这一角度，listwise的优势非常明显。

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。针对评分数据设计的矩阵分解算法，类似于监督学习中的回归问题，而隐式反馈数据通常是0-1二值数据。

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。



对于隐式反馈的数据，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。

根据以上思路，把排序问题转换为一个分类问题。分类问题的训练和预测需要正负样本，但是，通常情况下隐式反馈数据的负样本不那么明显。比如，很难肯定的认为用户没有点击或者是买过的商品一定是用户不感兴趣的。因为有可能是用户没有看到或者是暂时没有兴趣，但是不代表以后不会产生兴趣，

这时就需要借助排序学习，排序问题不会考虑正负样本的划分，而是计算样本的前后顺序。推荐列表注重排序，并不关心单个item预测的评分多少。

下面介绍两个排序学习的主流方法：

planA： Point Wise类：还是预测User对单个Item的偏好，最后根据偏好排序。

planB： Pair Wise类：对同一个User，直接考虑任意两个Item的顺序谁在前谁在后。

Pointwise和Pairwise的原理分析：

 Pointwise方法的主要思想是将排序问题转化为多类分类问题或者回归问题。

Pairwise 的主要思想是将Ranking问题形式化为二元分类问题。

第一，pointwise解决的是单文档分类问题，模型会告诉我们该条文档属于good，fair，bad其中一类，模型根据给出的训练数据进行建模，那么训练的数据我们首先只有搜出的文档，然后开始人工标注，或者根据用户反馈等等一系列手段，标注会根据一次搜索的返回结果考虑每个文档的好坏程度，但是不会比较两次不同搜索的返回结果好坏，我觉得这个就是截图中要说明的问题。所以，在这种情况下，模型可能会出现问题。

第二，pairwise解决的是文档对的排序问题，其实归根结底也是个分类问题，文档1排在文档2的几率问题，pairwise会考虑相同搜索的情况下，返回结果的排序问题，但是有一个弊端就是如果中间有一次预测失误，那么对余下数据的根据错误的数据进行比较，就会出现问题。

## TensorFlow构建DNC模型

数据采用criteo display ads[[18]](#footnote-18)

未完待续。。。

## 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。

# 推荐系统冷启动

1、用户冷启动

用户冷启动主要解决如何给新用户做个性化推荐的问题。当新用户到来时，我们没有他的行为数据，所以也无法根据他的历史行为预测其兴趣，从而无法借此给他做个性化推荐。

2. 物品的冷启动

将新物品推荐给曾经喜欢过与新物品相似的物品的用户，

由于新item没有用户行为，因此物品相似度只能从item自身出发，包括根据item的内容信息挖掘出item的向量表示，通过向量相似度来刻画物品相似度，还可以利用topic model挖掘出item的主题分布等等。

3. 系统冷启动

比较热门 --- 需要过滤掉比较冷门的物品，因为用户不知道背景信息，很难进行反馈

具有代表性和区分性 -- 过滤大众化和老少皆宜的物品，因为用户的反馈无法体现个性化

启动物品集合需要有多样性 -- 品类覆盖率要高，包含所有主流的用户兴趣

## EE问题

推荐系统领域有两个经典问题，EE问题和冷启动问题。EE即探索（exploration）和利用（exploitation）问题。“利用”就是要迎合用户的习惯，推荐用户感兴趣的东西；“探索”就是说不能只是推荐用户当前感兴趣的东西，久而久之，用户必然会腻烦，所以既要照顾客户当前兴趣又要探索用户新的兴趣，从评价角度讲就是平衡准确率和多样性。

冷启动问题设计到产品算法运营等一系列问题。Bandit算法是一种简单的在线学习算法，常常用于解决这个问题，下面介绍基础的Bandit算法以及相关的改进算法。

Bandit算法

假设你走近一家赌场，面前有一排老虎机，外表一模一样，但是每个老虎机吐钱的概率可不一样。如何知道每个老虎机吐钱的概率分布，每次如何选择老虎机可以做到最大化收益呢？这就是多臂赌博机问题（Multi-armed bandit problem,MAB）。

如何去解决这个问题，最好的办法是试一试，不能盲目的试，而是有策略并且快速的试，这些策略就是Bandit算法[[19]](#footnote-19)。

未完待续。。。

# 推荐系统的评价方式

推荐系统的评价标准有很多。

用户满意度（User Satifaction），调研或用户反馈；点击率、转化率等。

准确率（Accuracy）：precision、recall、f-score

覆盖率（Coverage）：召回到尾部物品和用户

多样性（Diversity）：两两之间不相似

新颖性（Novelty）：用户不熟悉的物品

惊喜性（Serendipity）：如何评价？

用户信任度（Trust）/可解释性（Explanation）：推荐理由

鲁棒性/健壮性（Robustness）：哈利波特现象；抗攻击、反作弊

实时性（Real-time/Online）：新加入的物品；新的用户行为（实时意图）

商业目标（business target）：一个用户带来多少盈利

实际场景中，可以分不同的数据流程单独进行评价，比如召回流程，用召回率评价召回商品的质量。

而在排序阶段可以用NDCG 或MRR

分别计算MSE、RMSE、精准率、召回率、覆盖率、流行度

import math

def RMSE(records):

"""计算RMSE

@param records: 预测评价与真实评价记录的一个list

@return: RMSE

"""

numerator = sum([(pred\_rating - actual\_rating)\*\*2 for pred\_rating, actual\_rating in records])

denominator = float(len(records))

return math.sqrt(numerator / denominator)

def MSE(records):

"""计算MSE

@param records: 预测评价与真实评价记录的一个list

@return: MSE

"""

numerator = sum([(pred\_rating - actual\_rating)\*\*2 for pred\_rating, actual\_rating in records])

denominator = float(len(records))

return numerator / denominator

def precision(recommends, tests):

"""计算Precision

:param recommends: dict

给用户推荐的商品，recommends为一个dict，格式为 { userID : 推荐的物品 }

:param tests: dict

测试集，同样为一个dict，格式为 { userID : 实际发生事务的物品 }

:return: float

Precision

"""

n\_union = 0.

user\_sum = 0.

for user\_id, items in recommends.items():

recommend\_set = set(items)

test\_set = set(tests[user\_id])

n\_union += len(recommend\_set & test\_set)

user\_sum += len(test\_set)

return n\_union / user\_sum

def recall(recommends, tests):

"""

计算Recall

@param recommends: 给用户推荐的商品，recommends为一个dict，格式为 { userID : 推荐的物品 }

@param tests: 测试集，同样为一个dict，格式为 { userID : 实际发生事务的物品 }

@return: Recall

"""

n\_union = 0.

recommend\_sum = 0.

for user\_id, items in recommends.items():

recommend\_set = set(items)

test\_set = set(tests[user\_id])

n\_union += len(recommend\_set & test\_set)

recommend\_sum += len(recommend\_set)

return n\_union / recommend\_sum

def coverage(recommends, all\_items):

"""

计算覆盖率

@param recommends : dict形式 { userID : Items }

@param all\_items : 所有的items，为list或set类型

"""

recommend\_items = set()

for \_, items in recommends.items():

for item in items:

recommend\_items.add(item)

return len(recommend\_items) / len(all\_items)

def popularity(item\_popular, recommends):

"""计算流行度

@param item\_popular: 商品流行度　dict形式{ itemID : popularity}

@param recommends : dict形式 { userID : Items }

@return: 平均流行度

"""

popularity = 0. # 流行度

n = 0.

for \_, items in recommends.items():

for item in items:

popularity += math.log(1. + item\_popular.get(item, 0.))

n += 1

return popularity / n

第三部分 案例部分

# 分布式推荐系统架构与服务

## 环境准备与软件安装

### MongoDB

#### MongoDB介绍

所有案例都采用NoSQL数据库进行存储，不同于传统的关系型数据库管理系统（RDBMS），NoSQL数据库在存储大数据量更加具有优势并且读写性能相对较高，

MongoDB是一个基于分布式文件存储的数据库，由C++语言编写，是一个介于关系数据库和非关系数据库之间的产品。其存储的内容是文档型，数据结构由键值对（key=>value）组成。MOngoDB文档类似于JSON对象，字段值可以包含其他文档，数组和文档数组。

比如：

图片包含 物体

描述已自动生成

#### mongodb 4.0 安装部署

下载安装包，并解压tgz文件：

$ curl -O https://fastdl.mongodb.org/linux/mongodb-linux-x86\_64-4.0.5.tgz

$ tar -xvf mongodb-linux-x86\_64-4.0.5.tgz

将压缩包拷贝到指定文件夹（/opt）

$ mv mongodb-linux-x86\_64-4.0.5 /opt/mongodb

添加mongodb的环境变量，并bin目录添加到PATH路径中：

vim /home/hadoop/.bashrc

export MONGODB\_HOME=/opt/mongodb

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

source /home/hadoop/.bashrc

打开 /opt/mongodb/bin/mongodb.conf配置文件添加配置项：

#数据文件目录（寻找一个稍大的磁盘，如/mnt/disk0/）

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

#日志目录

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

#端口

port = 27017

#以守护进程运行

fork = true

#设置权限

auth=True

bind\_ip=127.0.0.1

根据上述配置文件创建数据库目录：

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

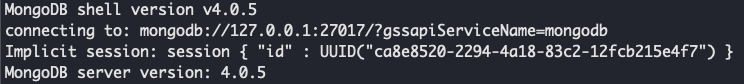
$ mkdir -p /mnt/disk0/mongodb/logs/

**运行mongodb：**

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

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 127.0.0.1:27017/recommender -u admin -p dis2019

#### mongodb的基础命令

查看collection：

show collections

创建collection：

db.createCollection("u\_attr")

#### spark读写MongoDB

为了提升推荐系统的实时分析功能，集成Spark引擎可以扩展MongoDB对于数据的处理和运营。 由此，可以使用集成工具MongoDB Connector for Apache Spark，这让Spark操作MongoDB数据更为简单，方便Spark进行流失数据处理。

添加sbt依赖包，在build.sbt中添加：

libraryDependencies += "org.mongodb.spark" %% "mongo-spark-connector" % sparkVersion

要求MongoDB版本在2.6及以上， Spark版本在2.4.0及以上，Scala版本在2.11及以上。

导入MongoDB连接包：

import com.mongodb.spark.\_

连接到MongoDB

当RDD的action操作需要读写MongoDB时，MongoDB数据库连接会自动触发。

从MongoDB中读取数据

import com.mongodb.spark.config.\_

def sparkReadCol(spark: SparkSession,  
 mongoUri: String,  
 colName:String)={  
 val readConfig = ReadConfig(Map(  
 "uri" -> mongoUri,  
 "collection" -> colName  
 ))  
 MongoSpark.load(spark.sparkContext, readConfig)  
}

#调用MongoDB的读取方法

val mongoDBName = "recommender"  
var mongoColName = "spark"  
val mongoUser = "admin"  
val mongoPwd = "admin"  
val mongoUriStr = s"mongodb://${mongoUser}:${mongoPwd}@127.0.0.1:27017/${mongoDBName}"

val doc = sparkReadCol(spark, mongoUriStr, mongoColName)

println(doc.count)

println(doc.first.toJson)

RDD数据写入MongoDB：

import com.mongodb.spark.config.\_

def sparkWriteCol(sparkDocuments: RDD[Document],  
 mongoUri: String,  
 colName:String) ={  
 val writeConfig = WriteConfig(Map(  
 "uri" -> mongoUri,  
 "collection" -> colName  
 ))  
 MongoSpark.save(u\_result, writeConfig)  
}

val writeConfig = WriteConfig(Map("collection" -> " recommender", "writeConcern" -> "testing"), Some(WriteConfig(sc)))

val sparkDocuments = sc.parallelize((1 to 10).map(i => Document.parse(s"{spark: $i}")))

val mongoDBName = "recommender"  
var mongoColName = "spark"  
val mongoUser = "admin"  
val mongoPwd = "admin"  
val mongoUriStr = s"mongodb://${mongoUser}:${mongoPwd}@127.0.0.1:27017/${mongoDBName}"

#调用写入方法

sparkWriteCol(sparkDocuments, mongoUriStr, mongoColName)

### **Kafka**

#### **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

#### Redis简介

Redis是一个key-value格式的内存存储数据库，读写速度很快，并且支持丰富的数据结果，包括以下：

1.String

2. list

3. hash table

4. set(集合)

5. zset(有序集合)

redis的体量很小，但却支持丰富的数据结构和相应的操作 方式，是一个非常好的工具。

#### 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

#### Redis数据结构[[20]](#footnote-20)

下面介绍这几种常见的数据结构和使用方法：

**1.String**

常用命令: set,get,decr,incr,mget 等。

String数据结构是简单的key-value类型，value其实不仅可以是String，也可以是数字。 常规key-value缓存应用； 常规计数：微博数，粉丝数等。

**2.Hash**

常用命令： hget,hset,hgetall 等。

hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值。 比如我们可以 hash 数据结构来存储用户信息，商品信息等等。比如下面我就用 hash 类型存放了我本人的一些信息：

key=JavaUser293847 value={ “id”: 1, “name”: “SnailClimb”, “age”: 22, “location”: “Wuhan, Hubei” }

**3.List**

常用命令: lpush,rpush,lpop,rpop,lrange等

list 就是链表，Redis list 的应用场景非常多，也是Redis最重要的数据结构之一，比如微博的关注列表，粉丝列表，消息列表等功能都可以用Redis的 list 结构来实现。

Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。

另外可以通过 lrange 命令，就是从某个元素开始读取多少个元素，可以基于 list 实现分页查询，这个很棒的一个功能，基于 redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西（一页一页的往下走），性能高。

**4.Set**

常用命令： sadd,spop,smembers,sunion 等

set 对外提供的功能与list类似是一个列表的功能，特殊之处在于 set 是可以自动排重的。

当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在一个set集合内的重要接口，这个也是list所不能提供的。可以基于 set 轻易实现交集、并集、差集的操作。

比如：在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis可以非常方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程，具体命令如下：

sinterstore key1 key2 key3 将交集存在key1内

**5.Sorted Set**

常用命令： zadd,zrange,zrem,zcard等

和set相比，sorted set增加了一个权重参数score，使得集合中的元素能够按score进行有序排列。

举例： 在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息，适合使用 Redis 中的 Sorted Set 结构进行存储。

**redis 设置过期时间**

Redis中有个设置时间过期的功能，即对存储在 redis 数据库中的值可以设置一个过期时间。作为一个缓存数据库，这是非常实用的。如我们一般项目中的 token 或者一些登录信息，尤其是短信验证码都是有时间限制的，按照传统的数据库处理方式，一般都是自己判断过期，这样无疑会严重影响项目性能。

我们 set key 的时候，都可以给一个 expire time，就是过期时间，通过过期时间我们可以指定这个 key 可以存活的时间。

如果假设你设置了一批 key 只能存活1个小时，那么接下来1小时后，redis是怎么对这批key进行删除的？

定期删除+惰性删除。

通过名字大概就能猜出这两个删除方式的意思了。

定期删除：redis默认是每隔 100ms 就随机抽取一些设置了过期时间的key，检查其是否过期，如果过期就删除。注意这里是随机抽取的。为什么要随机呢？你想一想假如 redis 存了几十万个 key ，每隔100ms就遍历所有的设置过期时间的 key 的话，就会给 CPU 带来很大的负载！

惰性删除 ：定期删除可能会导致很多过期 key 到了时间并没有被删除掉。所以就有了惰性删除。假如你的过期 key，靠定期删除没有被删除掉，还停留在内存里，除非你的系统去查一下那个 key，才会被redis给删除掉。这就是所谓的惰性删除，也是够懒的哈！

但是仅仅通过设置过期时间还是有问题的。我们想一下：如果定期删除漏掉了很多过期 key，然后你也没及时去查，也就没走惰性删除，此时会怎么样？如果大量过期key堆积在内存里，导致redis内存块耗尽了。怎么解决这个问题呢？ redis 内存淘汰机制。

redis 内存淘汰机制(MySQL里有2000w数据，Redis中只存20w的数据，如何保证Redis中的数据都是热点数据?)

redis 配置文件 redis.conf 中有相关注释，我这里就不贴了，大家可以自行查阅或者通过这个网址查看：

<http://download.redis.io/redis-stable/redis.conf>

redis 提供 6种数据淘汰策略：

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key（这个是最常用的）

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

no-eviction：禁止驱逐数据，也就是说当内存不足以容纳新写入数据时，新写入操作会报错。这个应该没人使用吧！

**4.0版本后增加以下两种：**

volatile-lfu：从已设置过期时间的数据集(server.db[i].expires)中挑选最不经常使用的数据淘汰

allkeys-lfu：当内存不足以容纳新写入数据时，在键空间中，移除最不经常使用的key

备注： 关于 redis 设置过期时间以及内存淘汰机制，我这里只是简单的总结一下，后面会专门写一篇文章来总结！

**redis 持久化机制**

**(怎么保证 redis 挂掉之后再重启数据可以进行恢复)**

很多时候我们需要持久化数据也就是将内存中的数据写入到硬盘里面，大部分原因是为了之后重用数据（比如重启机器、机器故障之后恢复数据），或者是为了防止系统故障而将数据备份到一个远程位置。

Redis不同于Memcached的很重一点就是，Redis支持持久化，而且支持两种不同的持久化操作。Redis的一种持久化方式叫快照（snapshotting，RDB），另一种方式是只追加文件（append-only file,AOF）。这两种方法各有千秋，下面我会详细这两种持久化方法是什么，怎么用，如何选择适合自己的持久化方法。

**redis使用注意点[[21]](#footnote-21)：**

1. 管理好自己的 key，比如用一个 set 来存放所有同类型的 key，这样可以避免使用 keys 来获取该类型所有 key，由于 Redis 是单线程的，在数据量大的情况下将会阻塞 Redis 中其他的操作

2. 如果不能避免使用 keys 操作，那么使用 scan 来替代，将遍历 Redis DB 中所有 key 的操作放到客户端来做，这样就不会导致其他操作被阻塞了

3. 在数据量大的情况下，key 或 value 的微小的压缩也能带来不小的内存使用提高

4. hash 结构是个好东西，但是它只支持一级 hash，所以如果需要使用多级 hash，那就进行序列化吧

5. 注意客户端和服务端之间的网络通信情况(–latency/–latency-history)

6. Redis 的 python 接口返回的 key 的类型和 value 中的最小结构的类型都是 string，记得转换成自己需要的类型

## 实现一个电影推荐系统[[22]](#footnote-22)

### 架构设计

综合业务：Spring、Tomcat

数据存储：业务数据MongoDB、搜索服务数据ES、缓存数据Redis

离线和实施推荐：Spark DF、ML、Streaming

消息服务：Kafka

### 数据加载与处理

数据加载服务，主要用于项目的数据初始化，用于将三个数据集（Movies【电影的数据集】、Rating【用户对于电影的评分】、Tags【用户对于电影的标签】）初始化到Mongodb数据库以及ElasticSearch里面。

【离线推荐】

通过Azkaban周期性的调度【离线统计服务】和【离线推荐服务】。

【离线统计服务】

最热电影统计算法  =>  RateMoreMovies表中

当前最热电影统计算法   =>  RateMoreRecentlyMovies 表中

电影的平均评分算法   =>  AvgMovies 表中

电影类别TOP10推荐   =>   GenresTopMovies表中。

【离线推荐服务】

通过Spark ALS算法计算模型的训练

基于Model产生电影相似度矩阵   =>  MovieRecs表中

基于Model产生用户推荐矩阵    =>  UserRecs表中

【实时推荐】

当一个用户完成了对电影的评分之后，触发实时推荐，后台服务将评分数据实时写入到LOG日志中。

Kafka会通过kafkaStream程序将log队列中的数据进行格式化，然后将数据推送到recommender的队列中

Spark Streaming实时推荐程序读取kafka中推送过来的数据，配合读取Redis缓存中的数据，运行实时推荐算法【基于模型的推荐】，把结果数据写入到MongoDB的StraemRecs表中

【推荐结果的聚合】

综合业务服务会根据一定的比例聚合 【离线推荐服务（ALS的协同过滤）】、【基于内容的推荐（基于ElasticSearch More like this功能）】、【实时推荐服务（基于模型的推荐）】这些结果。

聚合完成之后，将结果返回到用户端。

前期工作：数据加载

利用Spark SQL将数据分别导入到 MongoDB 和 ElasticSearch

**目标**

MongoDB

将Movie【电影数据集】/ Rating【用户对电影的评分数据集】/ Tag【用户对电影的标签数据集】数据集分别加载到MongoDB数据库中的Movie / Rating / Tag表中

ElasticSearch

需要将Movie【电影数据集】加载到ElasticSearch名叫Movie的Index中。需要将Tag数据和Movie数据融合。

**步骤**

先新建一个Maven项目，将依赖添加好。

分析数据集Movie、Rating、Tag

Spark DF加载数据集

将DF加载到MongoDB中：

将原来的Collection全部删除

通过DF的write方法将数据写入

创建数据库索引

关闭MongoDB连接

将DF加载到ElasticSearch中：

将存在的Index删除掉，然后创建新的Index

通过DF的write方法将数据写入

关闭Spark集群

// MONGO\_MOVIE\_COLLECTION，包含电影ID、描述、时长、发行日期、拍摄日期、语言、类型、演员、导演

case class Movie(mid: Int, name: String, descri: String, timelong: String, issue: String, shoot: String, language: String, genres: String, actors: String, directors: String)

// MONGO\_RATING\_COLLECTION，包含用户ID、电影ID、用户对电影的评分、用户评分时间 case class MovieRating(uid: Int, mid: Int, score: Double, timestamp: Int)

// MONGO\_TAG\_COLLECTION，包含用户ID、电影ID、标签的具体内容、用户打标签时间 case class Tag(uid: Int, mid: Int, tag: String, timestamp: Int)

### 离线推荐

统计推荐

**目标**

优质电影

获取所有历史数据中评分最多的电影的集合，统计每个电影的评分数 => RateMoreMovies

热门电影

认为按照月来统计，这个月中评分数量最多的电影我们认为是热门电影，统计每个月中的每个电影的评分数量 => RateMoreRecentlyMovie

统计电影的平均评分

将Rating数据集中所有的用户评分数据进行平均，计算每一个电影的平均评分 => AverageMovies

统计每种类别电影的TOP10电影

将每种类别的电影中评分最高的10个电影分别计算出来， => GenresTopMovies

**步骤**

新建一个项目StaticticsRecommender

统计所有历史数据中电影的评分个数

通过Rating数据集，用mid进行groupby操作，count计算总数

统计以月为单位的电影的评分个数

利用from\_unixtime对timestamp进行转换，结果格式为yyyyMM

通过groupby 年月，mid 来完成统计

统计每个电影评分得分  
通过Rating数据集，用户mid进行group by操作，avg计算评分

统计每种类别中评分最高的10个电影

需要通过JOIN操作将电影的平均评分数据和Movie数据集进行合并，产生MovieWithScore数据集

利用split和explode将MovieWithScore拆开为以一种Genre为开头的MovieWithScore

此处有两种方案（这里展示第一种，但最好的是建立Aggregator，在ALS离线推荐中介绍）：

通过Genre作为Key，进行groupByKey操作，将相同电影类别的电影进行聚集并按评分取TOP10。（TopN利用PriorityQueue实现）

利用repartitionAndSortWithinPartitions、mapPartition和groupSorted（轮子）实现分Genre的TOP10。

上面数据转化为DF后输出到MongoDB中

离线计算由原来的RDD改为基于Spark DataFrame的实现，而DF本身比RDD具有更多的默认优化，比如Tungsten运用了堆外内存，减少GC开销、序列化比RDD的Java默认序列化更加高效、全代码生成等优势。上述是效率提升是理论上的，下面是实际上的。

首先，DF相比于RDD能这大大地减少了代码量。下面的功能就是简单地统计每部电影的评分数、以月为单位统计每部电影的评分数、每部电影平均分、不同类型电影的评分top10（即分组topn）。这里需要连接电影数据集和评分集，前者包含了电影类型，后者包含平均分。原代码使用RDD的groupByKey，这是一种低效的操作，因为它不会进行map-side聚合，而且分组后使用map，直接调用sortWith对每个分组进行排序，再用slice进行取top10。这种实现并不好，在Scala中运用sortWith虽然方便，但是效率很低，它比toArray.sort慢不少。slice也是，用take更高效。但实际上这里RDD的实现也不需要用全排序再取前10。利用aggregateByKey加优先队列，即堆排的实现，效率更高。其用时由原来的20多秒变为6秒左右。当然，在DF中可用windowfunction来实现topn，但是windowfunction的底层实现是先shuffle然后sort，最后根据所定义的窗口进行相应的聚合。这一过程同样没有map-side聚合，所以尽管数据量少时速度更快，但在数据量大时或许不是一个好的topn实现。其实我觉得最好的topn实现应该是自定义一个aggregator，这个在离线推荐部分会提到。

#### ALS离线推荐

**目标**

训练ALS推荐模型并计算出属于用户各自的Top20电影推荐

计算电影相似度矩阵

**步骤**

训练ALS推荐模型

实例化ALS模型，按常规的模型训练流程训练模型后调用recommendForAllUsers完成推荐。

计算电影相似度矩阵

获取电影的特征矩阵，转换成DoubleMatrix

电影的特征矩阵之间做笛卡尔积，通过余弦相似度计算两个电影的相似度

将数据通过GroupBy处理后，输出

ALS模型的参数选择

通过计算ALS的均方根误差来判断参数的优劣程度

ALS模型需要三列，用户id、电影id和评分，并从中计算出用户矩阵和电影矩阵。项目需要给每个用户推荐20部电影，此处我直接使用了2.2提供的recommendForAllUsers函数便直接得到了结果。但后续项目还要求求出电影的相似度矩阵，用于后续实时推荐计算。这个算法spark没有现成的实现，但是思路根recommendForAllUsers是一样的。就是先求电影矩阵自身的cross join，然后求每部电影对除其自身外的其他所有电影的余弦相似度。公式为两个向量的点积除以两个向量长度的乘积。然后取出每部电影中，余弦相似度前100的电影，这样子构建出电影的相似度矩阵。我根据spark的recommendForAll的源码，重新实现了这一功能，这比原代码的实现快上1倍，而且cross join产生的中间数据的内存也减少了几倍，由几G到了几十M。实际上这里也得出了一个cross join的优化方式.....然后就是取每部电影的前100相似的电影。这里同样是分组TopN，但源码使用了私有的TopNAggregator。同样我也实现了一个，这是我觉得在数据量大时最好的TopN实现。首先它基于DS，也享受了DF的部分优化（比如Catalyst、序列化等）这是优于rdd的aggregateByKey的部分，然后，它包含了map-side聚合，减少了shuffle期间需要传输的数据，这是优于windowfunc的部分。

### 实时推荐

**日志预处理**：构建kafka流，从大量日志数据中过滤出特定前缀的用户评分log，并传到recommender topic让下面的spark streaming接收。

构建LogProcessor实现Processor接口，实现对于数据的处理

构建StreamsConfig配置数据

构建TopologyBuilder来设置数据处理拓扑关系，就是输入、处理器和输出的关系。

构建KafkaStreams来启动整个处理

**前提**

在Redis集群中存储了每一个用户最近对电影的K次评分。实时算法可以快速获取。

离线推荐算法已经将电影相似度矩阵提前计算到了MongoDB中。

Kafka已经获取到了用户实时的评分数据。

读取MonogDB中的相似度矩阵然后广播。读取kafka stream，提取里面的uid、mid、score评分、timestamp，即某个用户对某部电影的进行评分从而产生的日志，然后foreachRDD，对每条日志执行下面三个计算。从被广播的相似度矩阵中获取这次被评分的电影P最相似的K个电影（当然，实现还会从评分集中获取用户已看过的电影，并将这些电影过滤掉）这些作为推荐的候选电影。然后获取用户最近N次电影评分，实际是从redis中读取，这些数据用于调整候选电影的优先级别。后面更多就是利用Scala实现算法的思想了，和spark等技术相关性不大。首先新建一个数组存储候选电影的评分、两个map存储候选电影的增强和减弱因子。然后对每个候选电影进行遍历，遍历中再遍历用户最近评分的电影。针对每个候选电影，看用户最近评分的电影中有没有与它相似的，有的话返回其在相似度表中的分数，没有就返回0.0。然后如果相似度大于0.6，就将这个相似度作为权重，乘以用户对其相似电影的评分。得出候选电影的初步评分。然后根据用户对该相似电影的评分，如果大于3，则候选电影的增强因子+1，否则在减弱因子中-1。最后计算所有部候选电影的初步评分的均值，并加上其增强因子的对数，减去其减弱因子的对数，得出最终得分，并把这个得分表插入到MongoDB用于用户评分后的推荐。

# 基于电商平台推荐

## 数据流程

### 数据导入与转换

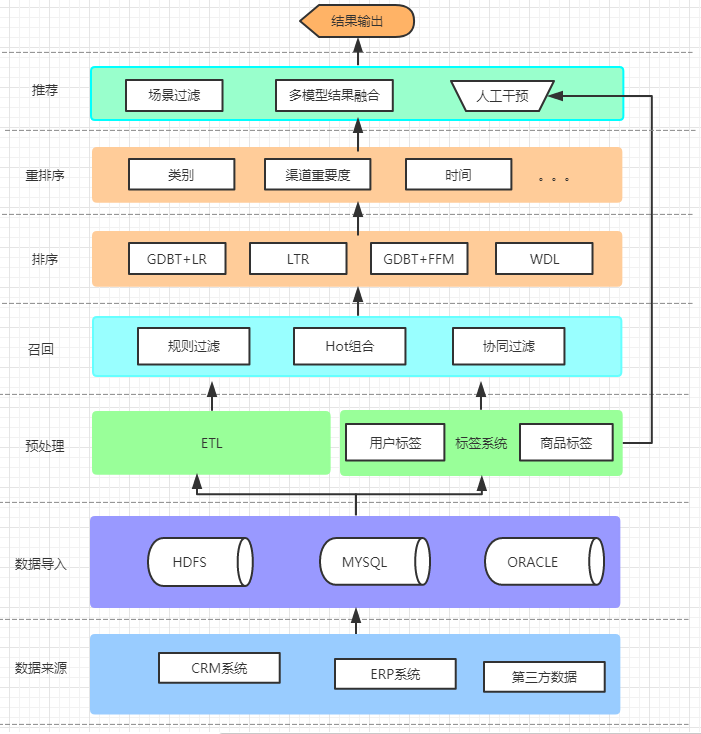
User\_action数据

|  |  |  |
| --- | --- | --- |
| 字段 | 字段说明 | 提取说明 |
| user\_id | 用户标识 |  |
| item\_id | 商品标识 |  |
| behavior\_type | 用户对商品的行为类型 | 包括浏览、收藏、加购物车、购买，对应取值分别是1、2、3、4。 |
| user\_geohash | 用户位置的空间标识，可以为空 | 经纬度 |
| item\_category | 商品分类标识 |  |
| time | 行为时间 | 精确到小时级别 |

商品数据：

|  |  |  |
| --- | --- | --- |
| 字段 | 字段说明 | 提取说明 |
| item\_id | 商品标识 |  |
| item\_ geohash | 商品位置的空间标识，可以为空 | 经纬度 |
| item\_category | 商品分类标识 |  |

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

**排序模型**

推荐系统可以理解为，对用户是否会对Item产生某种隐式反馈的分类问题。

推荐列表的获得也就是排序过程，因为我们并不关心单个item预测的评分。

**LR => GBDT + LR**

图片包含 地图, 文字

描述已自动生成

xgboost内部决策树分裂的部分效果图，f为特征，可以看到树状结构的算法不需要对特征进行归一化，而是根据信息增益大小进行分裂。

通过构建负样本，合并用户交易数据作为训练数据，同时预测用户-商品组合可能购买的概率值。

**LTR**

训练数据：所有用户行为的分组计数 <user,item,count>

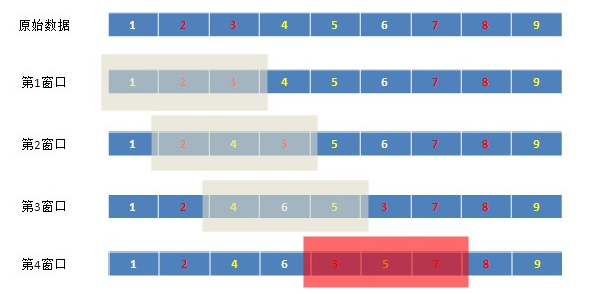
预测数据： 召回集合

xgboost 设置objective ： rank:pairwise

### 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

[[23]](#footnote-23)根据计算环境的不同，推荐系统的预测大体上可以分为在线（Online）、离线（Offline）两种。在线计算，指的是在线上的推荐服务中，对接受到的请求，进行实时计算，生成推荐结果并直接返回给请求方。离线计算，是指以一定时间周期运行的，对数据库中的大批量数据进行的计算。离线计算的结果通常会写入数据库中，供后续任务读取。除此之外，还有介于在线和离线之间的近线（Nearline）计算，它主要以流处理的方式对近实时的数据进行处理，并将结果写入数据库。

在推荐系统中，在线计算和离线计算有各自的优缺点，及其适合的使用场景。在线计算能够做到实时地对用户行为作出反馈，从而可以针对用户当前所处的环境和临时萌生的兴趣，为其提供更即时、更精准的推荐。但是，受限于系统对于延迟的要求，在线计算必须在算法的复杂性上作出一些牺牲。此外，在线计算能够处理的数据量通常也是比较小的。

离线计算对于算法的复杂性要求则没有那么高。它通常是以批处理的方式在分布式集群上计算。因此，它往往可以处理更大量的数据，考虑更多的特征。但由于离线计算通常是每天一次，因此也就相应的损失了一些实时性，无法对用户行为作出及时反馈。除了预测任务，模型的训练过程也可以算作为一种离线计算。它使用日志系统收集的历史数据，训练得到一个模型，并对其进行性能评估。产出的模型，将会被用于后面的离线和在线预测。模型训练过程，占用的资源多，花费的时间长，比较适合在分布式集群上计算。

离线模型与在线预测之间的融合

一般情况下，离线预测任务可以和模型训练可以完全对接。使用Spark MLlib的Pipeline接口可以将模型训练，以及训练之前的特征预处理、特征工程、特征交叉等阶段，按照一定的顺序组合成一个流水线，同时支持将训练好的整个流水线持久化到磁盘上。在预测阶段，只需要将事先保存的流水线模型加载到集群中，然后将原始特征输入流水线，最终就可以得到模型预测的结果。Spark平台推荐使用DataFrame进行数据训练，分布式环境中进行模型离线预测也可以使用DataFrame。

但是如果放到实时推荐场景中，以上的方法就不管用了，因为它缺乏实时性。实时预测必须要求系统具备实时感知用户实时兴趣，并且做到实时反馈的能力。为了推荐系统实现更好的实时性，需要在线上服务中，使用用户当前的实时特征和反馈，为用户推荐出他当下可能感兴趣的东西。在线上服务中，我们需要对用户可能感兴趣的物品进行排序，使得最合口味的物品被排在推荐列表的前列。因此，需要将离线训练的模型部署到线上服务中。

不同于离线预测，在线的模型预测没有直接使用离线训练存储的流水线模型，而是独立实现了模型预测的算法，以及输入模型前的全部特征预处理过程。因此，每次离线训练结束，将模型部署到线上时，需要将训练得到的模型参数拷贝到线上服务中，同时特征预处理过程需要的参数也需要更新到线上服务中。换句话说，我们需要在线上原封不动地再实现一遍离线训练中定义的特征预处理操作和具体的模型结构。

因此，我们面临了一个两难的困境：既希望模型可以以分布式高吞吐量的方式进行离线训练，同时又希望训练好的模型可以在线上以低延迟的方式进行实时预测。这就是在线预测和离线训练之间的矛盾。

面向实时预测的接口

虑在 Spark MLlib 的接口上做一些改动，给它加上实时预测的接口。

Spark MLlib 的 Pipeline 接口的通用性，在很大程度上依赖 DataFrame 这一通用的数据结构。对于 Spark MLlib 来说，在 Pipeline 中流动的数据，都是使用 DataFrame 包装的，每个 Transformer 都接收一个 DataFrame，对其做一个「变形」的操作，然后输出一个新的 DataFrame。DataFrame 的 schema 通过 Transformer 的参数（Param）来约定。

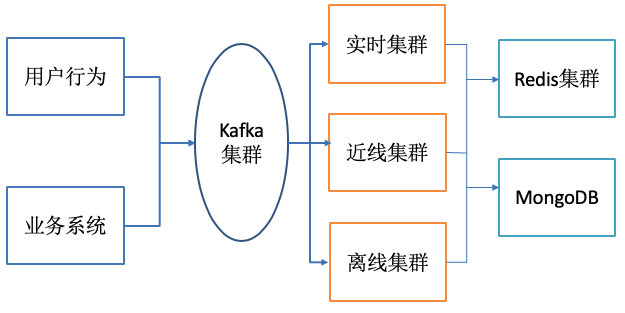
这一套行事方式在分布式计算中非常好用，但在对延迟要求很高的线上服务中，就不太适用了。其中，最主要的原因是，在线上单机计算的环境中将数据转化为 DataFrame 会有很多不必要的开销，影响服务的延迟。因此，一个直观的想法是：在 Spark MLlib 中为 Transformer 提供一个不依赖 DataFrame 的接口，使其内部核心的计算逻辑直接暴露出来，然后在线上服务中使用这一接口，从而绕过将数据用 DataFrame 封装这一耗时的操作。

经过观察，我们发现，推荐系统中最常用的模型预测流水线，主要由三个部分组成：特征向量化，特征预处理，和模型预测。其中，特征向量化是将各种数据类型的原始特征转化为向量的过程，它可以由一个特征向量化器（feature vectorizer）完成，其输入是一个某种数据类型的原始特征（比如说一个Map），输出是一个特征向量；特征预处理是对特征向量进行变形、归一化等操作的过程，它可以由一个或多个特征转换器（feature transformer）组成，其中每个转换器的输入和输入都是一个向量类型的特征；模型预测是指将特征输入一个训练好的分类器、回归器，或排序器中，得到一个「分数」的过程，其中「分数」可以表示离散的标签（分类器），也可以表示连续的值（回归器），甚至可以是一个排名（排序器），它的输入是一个向量类型的特征，输出是一个标量值。由于样本特征在存储的时候可能不是采用向量这种类型，而通用的特征处理器都是假设了特征为向量，所以一般我们会在第一步首先将非向量类型的特征转化为向量类型。

基于上面的分析，我们设计了两个面向实时预测的接口，分别用于特征向量化、预处理和模型预测。

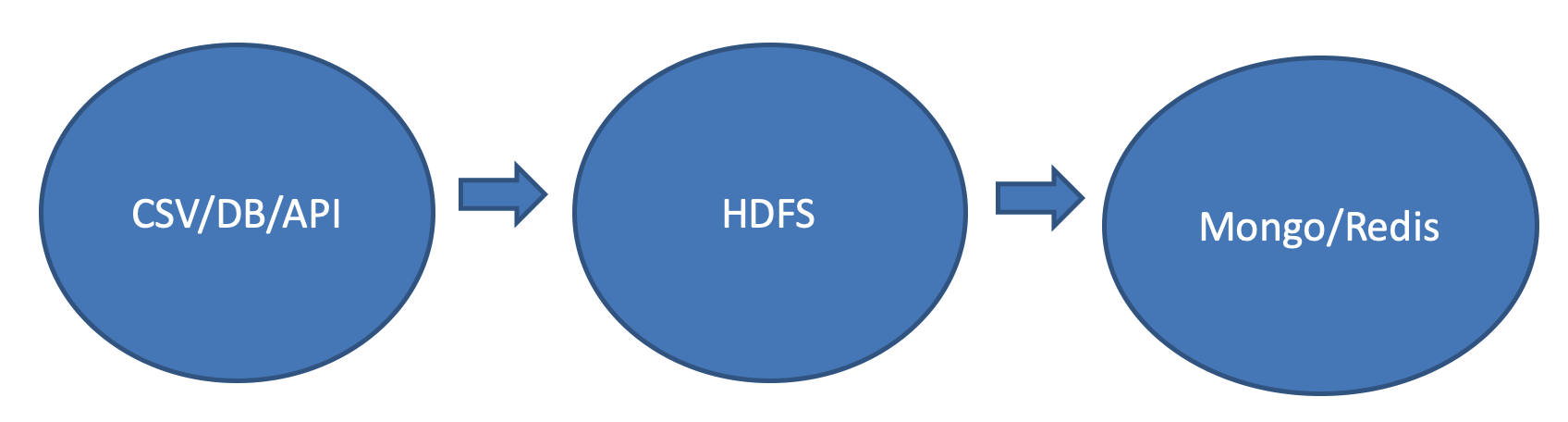
预测器的抽象接口是 OnlinePredictor。它有一个类型参数 FeaturesType，表示这个预测器可以接收的特征类型。我们注意到，对于某些预测器，我们有时需要得到两种类型的预测得分，比如对于二分类器，可能不仅需要输出分类标签，还要输出原始的预测得分。因此，在 OnlinePredictor 中有 predict 和 predictRaw 两个预测接口，根据具体的模型实现需要，可以分别设定两个接口结果的含义。

未完待续。。。



### 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

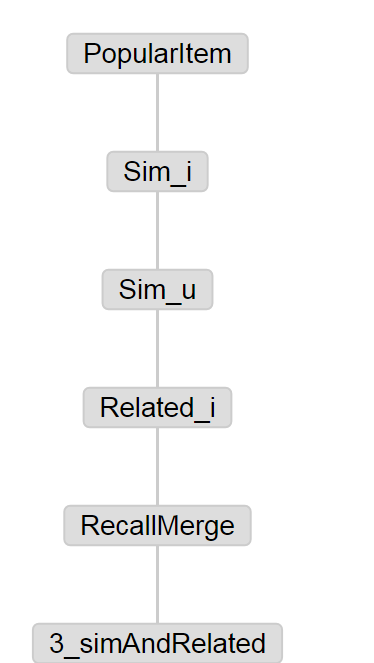
#### 电商数据导入和转换

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

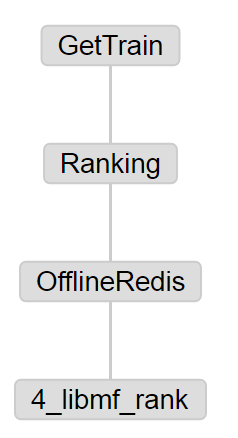
描述已自动生成

#### 召回



#### 精排序

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。



### 实时推荐lambda架构

大数据的 Lambda 架构和函数式编程有异曲同工之妙，其核心都是数据的不可变。对不可变的数据进行变形、加工和处理，来回答用户的问题（query）。通过 serving layer 的预先计算，来降低计算的复杂度。通过 speed layer 来满足实时性的要求，并且通过把最复杂的设计限制在 speed layer 来增加系统的可靠性。用函数式编程的观点来设计系统架构，在宏观层面上享受了一些函数式编程的优势。分布式系统需要保证“读”操作的nil-potent和“写”操作的idem-potent。lambda architecture中的元素正好符合该原则query-“读”操作。

只作用于batch和realtime两个view层的数据集，不会修改原始数据。由batch layer来基于原始数据（ground truth）统一掌管“写”操作，发生任何系统故障、意外重启、或发现过去的bug都可以通过一次batch layer升级和重新计算，直接生效。分布式系统中数据一致性的问题是个经典难题，借助batch和speed layer的区分，方便思考、理解该架构下系统一致性方面的特性，通过原始数据、新数据、几种view来明确区分系统中各种数据的相对重要程度，只需要保证原始数据正确安全就好[[24]](#footnote-24)。

Lambda架构的组件选择

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、小程序推送以及短信提示。

## 需求分析

大型商超的个性化推荐需求总体上分为以下几个步骤：

1. 从客户的基本信息、交易数据中挖掘有价值的信息，并且通过这些信息进一步分析用户的个性化需求。

在推荐系统领域这一步要做的就是充分利用<用户、物品、环境>三要素，计算用户、物品和环境的标签，完善用户和物品画像。其中标签的计算方式多种多样，包括不限于文本、数字、图片等等。

1. 以系统自动计算和分析为主，少量人工为辅，最终提升整体客单价、销售额、毛利润。

## 数据流程

**数据来源**

用户已有的数据接入：CRM，ERP，数据格式可能是各种数据库或者excel文件，csv文件

应支持数据的定时更新。

**数据输入定义**

1. 零售数据 （因为代码已经固定，字段名称不变）

字段不可为空已表明，其他字段可为空字符

|  |  |  |
| --- | --- | --- |
| 字段 | 字段说明 | 提取说明 |
| uid | 订单编号 | not null |
| sldat | 购买时间 | not null |
| vipno | 用户id | not null |
| prodo | 商品id | not null |
| pluname | 商品名称 | not null |
| bndno | 品牌编码 | not null |
| bndname | 品牌名称 | not null |
| qty | 购物数量 | not null |
| amt | 价格 | not null |
| vip\_gender | 用户性别 |  |
| vip\_birthday | 用户出生日期 |  |
| vip\_created\_dat | 会员注册日期 |  |

#### 零售数据导入和转换

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。

**ETL与标签系统**

原始数据预处理，见step\_0 和 step\_200-step\_210 笔记

etl 数据清洗和字段划分

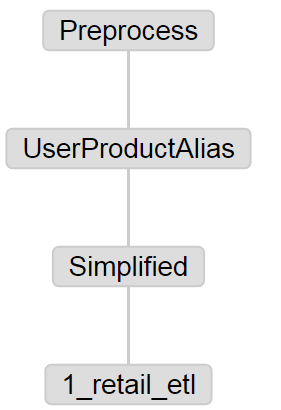
exceptionClean 异常清洗

extractAction 提取用户行为字段（user\_id, item\_id, rating, time）

extractOperation 提取用户操作（user\_id, item\_id, action\_type, category, brand, ...)

extractUserLabel 提取用户标签

extractItemLabel 提取物品标签



## 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

## 零售业务召回

### 召回策略

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。

对(用户，物品)间关系深入挖掘， 揭示购物模式。

* 1. 复购
  2. 协同推荐
  3. 基于内容的推荐
  4. 大众推荐
  5. 关联推荐

系统把合适的物品推荐给合适的人，促进供需两旺。

* 1. 纯自动化方法
  2. 辅助以人工方法

系统提供合适的界面，展现各种挖掘结果并能够对模型和推荐过程进行一定干预。

**召回方法**

可以根据用户行为、用户画像、显式反馈、隐式反馈

**候选商品**

可以通过，长期行为召回，实时行为召回，热门召回，用户画像召回，搜索词召回，产品词召回。

**过滤规则**

库存过滤，敏感词过滤，人工设定过滤，订单复购过滤

规则过滤

只对交易记录>P的用户做推荐

过滤掉团购用户

过滤掉总交易次数<N 的商品，不做为候选集

过滤黑名单（购物袋，季节商品，节日商品）

### 召回模型

#### 人气商品

人气商品实际上是热门商品的匹配，根据不同种类，人气商品可以分为大众热门、群体热门和品类热门等等。人气商品的热度可以进行量化，最简单的就是排名的顺序值，也可以通过特征模型计算。下面实现一个基于简单规则的人气商品排序。

**大众热门**

大众热门就是取所有商品的交易记录，然后给每一个商品计数，最后按照计数大小倒序排序取TopN。大众热门商品一般在新用户加入时使用，因为这个时候系统不知道用户的任何信息。在推荐列表中加入大众热门可以解决部分冷启动问题。

计算大众热门的代码：

//计算大众热门  
val userItemDF = userBehaviorData.  
 map(line => (line.getString(2), line.getString(4))).toDF("user","item")  
val itemCount = userItemDF.  
 groupBy("item").  
 count().  
 orderBy(-*col*("count")).  
 map(x=> (transAliasFun(x.getString(0), itemMap), x.getLong(1)))  
//测试  
 itemCount.take(5).foreach(*println*)

一般情况下，线下商超的大众热门靠前的都是蔬菜类，结果输出：

(西红柿,66529)

(后腿肉,60001)

(鸡蛋,59527)

(鲜土豆,45029)

(黄瓜,37581)

**品类热门**

品类下的热门在实际推荐场景用的比价多。当用户的兴趣点在一个品类上，比如“可乐“，根据长尾效应，同一品类下，用户大概率会选择被购买次数较多的那个商品。这也很符合消费心理学，买的多的商品往往在价格和质量上都有很好的保证。

下面是spark计算品类热门的代码：

//计算品类下热门  
case class CateItem(cate: String, item: String)  
def itemCountInList(list: List[String]) = {  
 list.map((\_, 1)).  
 groupBy(\_.\_1).  
 map {case (i, li) =>  
 (i, li.map(\_.\_2).reduce(\_ + \_))}.  
 toList.sortBy(-\_.\_2)  
}  
  
val itemCate = userBehaviorData.*rdd*.  
 map(line => (line.getString(5), line.getString(4))).  
 groupByKey().  
 mapValues {iter =>  
 itemCountInList(iter.toList).map(\_.swap).sortBy(-\_.\_1)  
 }.flatMapValues(x=>x)  
  
//测试  
itemCate.map{case(user,(count,item))=>  
 (transAliasFun(user,cateMap),  
 (count, transAliasFun(item, itemMap)))}.  
 filter(x=> x.\_1 == "可乐").  
 take(5).foreach(println)

输出品类“可乐”下面的排名靠前的几个Item：

(可乐,(4328,可口可乐500ml))

(可乐,(4299,可口可乐2.5l))

(可乐,(2032,百事可乐600ml))

(可乐,(2009,可口可乐零度可乐500ml))

(可乐,(1730,可口可乐1.25L+果粒橙1.25L双促装))

**群体热门**

群体热门商品的计算依赖于用户的属性和标签，比如性别、年龄、爱好、习惯等等。多种标签也可以进行组合形成一个群体，比如找出年龄在20-25周岁的喜欢电子产品的男性青年，然后挖掘这类人群的热门商品。

#### 关联商品

**购物篮频繁项统计**

相关商品（购物篮相关，用户相关）

采用频数统计的方式，也就是找出一个购物篮中所有的组合，计算出现频数。

val transactions = userBehaviorData.  
 map(line => (line.getString(0), line.getString(4))).  
 *rdd*.groupByKey().  
 mapValues{items =>  
 val itemPair = items.toArray.distinct.  
 combinations(2).toArray.map(\_.mkString("|"))  
 itemPair  
 }.flatMapValues(x=>x).map(\_.\_2)

val itemFreq = transactions.  
 map(x => (x, 1)).  
 reduceByKey(\_ + \_).  
 sortBy(-\_.\_2).  
 map(x=>(x.\_1.split("\\|"),x.\_2)).  
 map{case(itemPair, count)=> (itemPair(0), itemPair(1), count)}  
itemFreq.map{case(i1,i2,c)=>  
 (transAliasFun(i1,itemMap), transAliasFun(i2,itemMap),c)}.  
 filter(x=>x.\_1.contains("可乐")).  
 collect().foreach(println)

上述代码过滤出关于“可乐”的几个频繁项：

输出结果如下:

(可口可乐500ml,翅中,857)

(可口可乐2.5l,翅中,341)

(可口可乐彩罐350ml\*6,雪碧拉罐330ml\*6,292)

(可口可乐1.25L,翅中,253)

(可口可乐零度可乐500ml,翅中,197)

(可口可乐500ml,雪碧500ml,192)

(可口可乐500ml,鲜翅中,190)

根据结果发现频繁项的头部都是交易量最大的蔬菜生鲜类，效果并不好。

**关联规则挖掘**

通过Spark中的关联挖掘类AssociationRule把上述频繁项统计结果作为训练集，置信度阈值取0.3（该值需要根据实际情况进行调节）训练模型。

以下为Spark-Mlib AssociationRules挖掘商超数据关联规则代码：

import org.apache.spark.mllib.fpm.AssociationRules  
import org.apache.spark.mllib.fpm.FPGrowth.FreqItemset  
  
val itemCount = userBehaviorData.  
 map(line => line.getString(4)).*rdd*.map(x => (x, 1)).reduceByKey(\_ + \_)  
val transItem = transactions.map(x => (x, 1)).reduceByKey(\_ + \_)  
val frequency = itemCount.union(transItem)  
val freqItemsets = frequency.map{items =>  
 val itemset = items.\_1.split("\\|")  
 val freq = new FreqItemset(itemset, items.\_2)  
 freq  
}

val ar = new AssociationRules().  
 setMinConfidence(0.3)  
val results = ar.run(freqItemsets).cache()  
  
*println*(s"associative itemset count: **$**{results.count()}")  
val save = results.sortBy(-\_.confidence).map{ rule =>  
 (transAliasFun(rule.antecedent.mkString(","),itemMap),  
 transAliasFun(rule.consequent.mkString(","),itemMap), rule.confidence)  
}.filter(x => x.\_1.contains("可乐"))  
save.take(8).foreach(*println*)

打印出“可乐”关键词下的一些关联规则项：

(可口可乐零度可乐330ml,雪碧1.25L,1.0)

(可口可乐零度可乐330ml,芬达2L,1.0)

(可口可乐姜味可乐汽水330ml,农夫东方树叶乌龙480ml,0.6666666666666666)

(百事可乐250ml,好丽友派6P34g\*6,0.3333333333333333)

(百事可乐250ml,芬达橙味250ml,0.3333333333333333)

(百事可乐250ml,梁丰麦丽素巧克力75g,0.3333333333333333)

(百事可乐250ml,南方精装芝麻糊360g,0.3333333333333333)

实际场景中需要对频繁项的结果做一些规则过滤。比如与“可乐”等饮料商品不属于同一个商品大类下的频繁项需要过滤掉。

#### 协同过滤

协同过滤（CF）算法可以利用之前相似度计算的结果进行过滤提取，获得召回结果。CF算法分为基于Item的协同过滤和基于User的协同过滤。一般情况，如果客户的数量远远大于商品的数据量时，建议使用ItemBased CF，一是保证计算效率（商品的维度比用户的维度小），二是保证效果（商品向量相对于用户向量而言没有那么稀疏）。

UserBased CF算法具体的计算方法如下。

根据每个用户的相似度计算结果，对于每个用户U，找到与U最相似度的TopN个用户（N初始化为10）。如果相似度相同，就选择用户ID较小的用户，统计TopN用户所有观看的电影数量，给U推荐K个电影（K初始化取10），如果统计次数相同，选择ID较小的电影。

通过UserBased-CF算法进行召回的代码如下：

def recomToUsers(userSim: RDD[(String, String, Double)],  
 trainData: RDD[(String, String, Double)],  
 recom\_num: Int = 10) = {  
 val similarities = userSim.map{case(u1,u2,s)=>(u1,(u2,s))}  
 // ratings format (user,(item,raing))  
 val ratingsInverse = trainData.map(rating => (rating.\_1, (rating.\_2, rating.\_3)))  
 //statistics format ((user,item),(sim,sim\*rating)),,,,  
 //ratingsInverse.join(similarities) fromating as (user,((item,rating),(user2,similar)))  
 val statistics = ratingsInverse.join(similarities)  
 .map(x => ((x.\_2.\_2.\_1, x.\_2.\_1.\_1), (x.\_2.\_2.\_2, x.\_2.\_1.\_2 \* x.\_2.\_2.\_2)))  
 // predictResult fromat ((user,item),predict)  
 val predictResult = statistics.  
 reduceByKey((x, y) =>  
 ((x.\_1 + y.\_1), (x.\_2 + y.\_2))).map(x => (x.\_1, x.\_2.\_2 / x.\_2.\_1))  
 val filterItem = trainData.map(x => ((x.\_1, x.\_2), Double.*NaN*))  
 val totalScore = predictResult ++ filterItem  
 val finalResult = totalScore.reduceByKey(\_ + \_)  
 .filter(x => !(x.\_2 equals (Double.*NaN*)))  
 .map(x => (x.\_1.\_1, x.\_1.\_2, x.\_2))  
 .groupBy(x => x.\_1)  
 .flatMap(x => (x.\_2.toList.sortWith((x, y) => x.\_3 > y.\_3).take(recom\_num)))  
 *logger*.info("uBased recom example: " + finalResult.first)  
 *logger*.info("uBased recom count: " + finalResult.count)  
 finalResult  
}

#### 代码目录与输入输出

**recall/iSimMatchByRating 相似物品计算**

i\_sim\_result 结果保存磁盘并入库

输入：/xiaoi/recommend/data/etl/train\_implicit

输出：/xiaoi/recommend/data/recall/i\_sim\_result

**iSimByRating相似用户计算**

u\_sim\_result 结果保存磁盘并入库

作为协同过滤提供的候选集

输入：/xiaoi/recommend/data/etl/train\_implicit

输出：/xiaoi/recommend/data/recall/u\_sim\_result

**iRelatedMatch相关物品计算**

i\_related\_result 结果保存磁盘并入库

模块化提供：

1. 相关sku，提供组合推荐

2. 相关品类，如啤酒和尿布

**matchMerge综合所有召回结果**

1. 读取召回文件夹下的所有候选集

2. 去重，过滤黑名单

协同过滤目前只获取itembased的结果

输入：/xiaoi/recommend/data/recall/

字段： user,item,value

输出：

复购 /xiaoi/recommend/data/recall/reorder

新鲜 /xiaoi/recommend/data/recall/fresh

### 特征计算与特征选择

零售场景目前涉及的字段 UID,SLDAT,VIPNO,PRODNO,PLUNAME,DPTNO,DPTNAME,BNDNO,BNDNAME,QTY,AMT,VIP\_GENDER,VIP\_BIRTHDAY,VIP\_CREATED\_DATE

特征计算所需：

UID,SLDAT,VIPNO,PRODNO,DPTNO,QTY,AMT

特征选择

1. 反例的特征没有 ui和order特征

2. 基于item的推荐系统，反例组合过于庞大，xgboost 单机无法进行，尝试拆分

3. 可以先尝试category的推荐，看看效果是否比LFM推荐的好

**新的特征（包含之前的部分特征）**

user：

u\_new\_item\_ratio 购买新鲜商品所占的比例

u\_new\_category\_ratio 购买新鲜品类的比例

u\_new\_m\_category\_ratio 购买新鲜中类的比例

u\_new\_b\_category\_ratio 购买新鲜大类的比例

u\_time\_new\_item 多长时间购买新商品

u\_turnover\_rate 用户在验证集中购买的转化率，用户购买的商品个数/用户预测集的商品个数

u\_diff\_hours\_times 用户的每两次购买之间的平均时间差，用小时表示

u\_b\_count\_in\_n（n=1/3/6） 用户在观察日前n天的行为总数计数（不同时间粒度：最近1天/3天/6天）

item：

i\_hot\_index 商品的热门指数

i\_hot\_in\_small\_category\_index, i\_hot\_in\_middle\_category\_index, i\_hot\_in\_big\_category\_index 商品在改小类/中类/大类中的热门指数

i\_small\_category 小类

i\_middle\_category 中类 （不需要）

i\_big\_category 大类 （不需要）

i\_c\_u\_count\_in\_n 类别在考察日前n天的用户总数计数 反映了item\_category的热度（用户覆盖性）

i\_c\_b\_count\_in\_n 类别在考察日前n天的行为总数计数 反映了item\_category的热度（用户停留性）

i\_u\_count\_in\_n 商品在考察日前n天的用户总数计数 反映了item\_id的热度（用户覆盖性） 3

i\_b\_count\_in\_n 商品在考察日前n天的行为总数计数 反映了item\_id的热度（用户停留性） 3

i\_bi\_count\_in\_n 商品在考察日前n天的各项行为计数 反映了item\_id的热度（用户操作吸引），折射出item\_id产生的购买习惯特点 12

i\_b4\_rate 商品的点击购买转化率 反映了商品的购买决策操作特点 1

i\_b4\_diff\_hours 商品的点击购买平均时差 反映了商品的购买决策时间特点 1

ui特征： 只有复购模型使用，以及线上电商场景使用

发现：将类别、性别等类别特征转换成one-hot编码，准确率反而降低。可能， Tree-Model 不太需要one-hot 编码

参考：<https://www.deeplearn.me/1625.html>

## 零售商品排序

**排序 rank**

A 复购

B 新鲜

LTR排序的结果为分数，而不是概率值。

C 热门

D 相似商品

E 综合模型（多模型融合）

整体式混合

特征组合

特征补充

并行式混合

交叉式混合

其中每个分类采用一种单独的推荐系统。

加权式混合

多个推荐系统结果的加权和。

切换式混合

流水线式混合

串联混合

分级混合

简单电商混合推荐设计：

结果加权（以下所有权重 \* 时间衰减系数）：

1. 主动搜索的权重最高

2. 进入购买订单，未付款情况权重第二。

3. 加购物车（未购买）权重第三。

4. 查看评论，说明有购买意愿

5. 收藏

LR再做一次回归 ：

相关特征的计算，根据用户行为判断其对 热门-复购-新鲜 物品的反应情况。

人工干预（群包匹配）

人工根据用户标签和商品标签批量加入排序列表。

**注意点：**

1. 综合模型是归并多个推荐器的推荐结果，按业务规则进行合并，需要考虑一定的多样性。

2. 每个模型计算得到的<推荐值>的归一化问题。热门模型得到的是权重值，复购模型得到的是概率值，新鲜回归模型得到的是评分值。

3. 目前实现简单的加权融合，考虑两个指标confidence=1-rank/K （K取50或20）， importance = 1 （也就是说默认weight都为1）

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。

**python/freshRank.py 新鲜商品排序**

保存磁盘并入库 /xiaoi/recommend/code/src/main/python/fresh\_rank.py

输入：/xiaoi/recommend/data/recall/fresh

输出：/xiaoi/recommend/data/rank/fresh

**rank/rankMerge融合模型排序**

读取并入库

merge\_model\_result

输入：/xiaoi/recommend/data/rank/reorder

/xiaoi/recommend/data/rank/fresh

输出：/xiaoi/recommend/data/rank/merge

**重排序 rerank**

A 品类交替/打散

B 品牌交替/打散

C 强制营销活动（粒度为排名）

**结果入库 database**

各个模型的结果以及最终融合结果都要保存。一般情况下，所有数据都要先放在MongoDB的保存，对于那些请求频繁的用户或者商品可以存放在Redis缓存数据库。

**rank/rankInsertDB 结果入库**

复购和新鲜的推荐结果列表入库

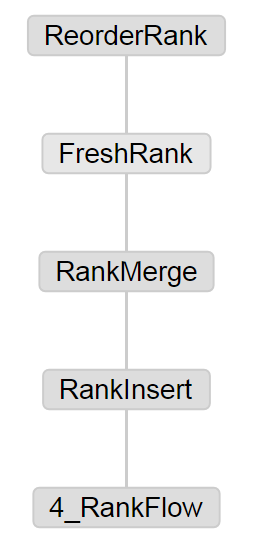
reorder /xiaoi/recommend/data/rank/reorder/prediction\_result.csv

fresh /xiaoi/recommend/data/rank/fresh/

black\_list

white\_list

排序模块总体azkaban Flow流程：



同上召回操作，接受召回结果进入排序模块，分模块排序之后整体融合，最后入库。

# 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

## Spark GBDT+LR预测高潜客户

基于银行客户的基础信息、账户信息、交易行为、APP浏览行为等等，预测潜在的购买人群。

Adult数据集是UCI机器学习库中数据集，也是机器学习实践的一个很好的入门示例。该数据集记录了若干个人信息和个人年收入。个人年收入与多种因素相关，从直觉上讲，可能受到个人教育程度、年龄、性别、职业等等因素影响。

数据集中一共包含14列属性字段，包含个人的人口统计信息和其他信息。1列类别字段，表示为收入，分为两类：<=50K和>=50K。通过机器学习方法，我们可以根据个人信息预测某个人收入水平的可能性。

数据集下载：<http://www.cs.toronto.edu/~delve/data/adult/desc.html>

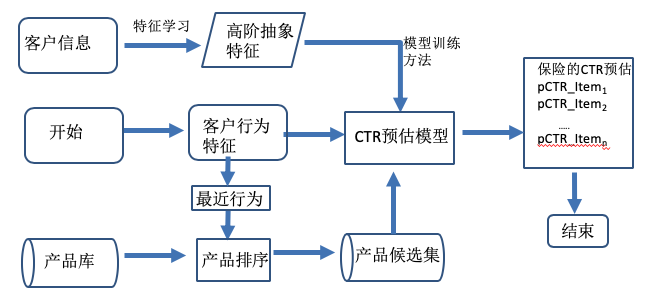
图片包含 文字

描述已自动生成

个人属性包含6个连续属性和8个离散属性。

我们发现数据集中包含7%的缺失值。

## 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://itindex.net/detail/58622-spark-%E7%89%B9%E5%BE%81%E6%8F%90%E5%8F%96-%E9%80%89%E6%8B%A9> [↑](#footnote-ref-1)
2. <http://www.zmonster.me/2015/11/15/text_similarity_survey.html> [↑](#footnote-ref-2)
3. 是一种在图形平面上，有多个[节点](https://zh.wikipedia.org/wiki/%E7%AF%80%E9%BB%9E)的[路径](https://zh.wikipedia.org/wiki/%E8%B7%AF%E5%BE%84)，求出最低通过[成本](https://zh.wikipedia.org/wiki/%E6%88%90%E6%9C%AC)的[算法](https://zh.wikipedia.org/wiki/%E7%AE%97%E6%B3%95)。常用于游戏中的NPC的移动计算，或[网络游戏](https://zh.wikipedia.org/wiki/%E7%BD%91%E7%BB%9C%E6%B8%B8%E6%88%8F)的BOT的移动计算上。 [↑](#footnote-ref-3)
4. <https://yongyuan.name/blog/ann-search.html> [↑](#footnote-ref-4)
5. 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-5)
6. 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-6)
7. <https://github.com/nmslib/hnswlib> [↑](#footnote-ref-7)
8. <https://github.com/microsoft/SPTAG> [↑](#footnote-ref-8)
9. <http://www.zmonster.me/2015/11/15/text_similarity_survey.html> [↑](#footnote-ref-9)
10. 推荐系统原理与实践 213 [↑](#footnote-ref-10)
11. <https://blog.csdn.net/sinat_22594309/article/details/86576757> [↑](#footnote-ref-11)
12. <https://www.cnblogs.com/bjwu/p/9358777.html> [↑](#footnote-ref-12)
13. <https://www.csie.ntu.edu.tw/~cjlin/libmf/> [↑](#footnote-ref-13)
14. Spark mllib edition2 [↑](#footnote-ref-14)
15. <https://www.cnblogs.com/code2one/p/10366183.html> [↑](#footnote-ref-15)
16. <https://zhuanlan.zhihu.com/p/42123341> [↑](#footnote-ref-16)
17. <https://xgboost.readthedocs.io/en/latest/jvm/xgboost4j_spark_tutorial.html> [↑](#footnote-ref-17)
18. <https://www.cnblogs.com/code2one/p/10343134.html> [↑](#footnote-ref-18)
19. <http://www.voidcn.com/article/p-efnalpyr-sn.html>

    <https://zhuanlan.zhihu.com/p/28067278> [↑](#footnote-ref-19)
20. <https://github.com/Snailclimb/JavaGuide/blob/master/docs/database/Redis/Redis.md> [↑](#footnote-ref-20)
21. 按实际情况删减修改 [↑](#footnote-ref-21)
22. <https://www.cnblogs.com/code2one/p/9927026.html#4288560> [↑](#footnote-ref-22)
23. <https://zhuanlan.zhihu.com/p/73853438> [↑](#footnote-ref-23)
24. <https://blog.csdn.net/qq_39521554/article/details/79884525> [↑](#footnote-ref-24)