**第一部分 数据与画像系统**

# 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)

**第二部分 技术与算法基础**

# 大数据技术

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。

## Spark mllib基础数据类型

### 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")

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")

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

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。

### 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（Int,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。

### 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。

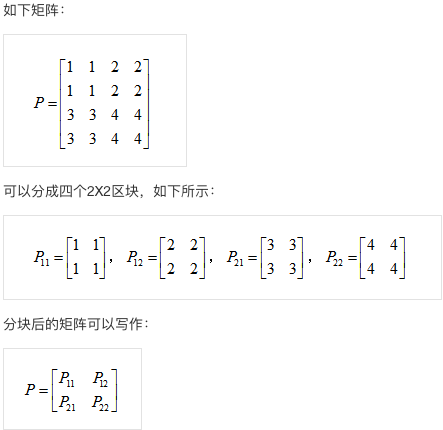
columnsSimilarities(threshold: Double)

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。

### 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。



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。

## 大规模相似度计算

（item相似度与user相似度）

数据ml-20m, movieLens2000万数据中，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。

ItemCF的相似度计算相对较快，因为item的维度有限，去重之后大概2万多个。

UserCF的相似度计算是个问题，因为用户有13万个，如果两两计算相似度，那总共就要算次相似度，如果每次4个Byte，总共需要70G的数据量。

### 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

向量化+分布式矩阵

RowMatrix

CoordinateMatrix

//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

工程上常用算法：

1. LSH

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

LSH保证高维空间相近的点映射到低维空间相近的概率很高。LSH也是一个降维的过程。

LSH的参数

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。

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。

Spark-LSH实现Item相似度计算：

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

### 业务上解决，不算或者少算

第一，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。

第二，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。

### 最佳方式

当前最佳方案： LSH + TopN，无需计算笛卡尔积，只要查找最近邻TopN即可。如果是计算笛卡尔积，时间复杂度至少 O（2^n）, 如果变成TopN查找问题，时间复杂度立马下降到O（n）。 由此，在真实推荐中，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。

由此，之前需要几个小时甚至是几天时间来完成的相似度计算问题，实际上可以通过实时检索的方式快速找到相似度对象。

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

# 经典推荐算法

## Memory\_Based 算法

### 客户向量空间模型

图片包含 文字

描述已自动生成

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

### 用户行为权重模型

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

图片包含 墙壁

描述已自动生成

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

### 协同过滤

图片包含 屏幕截图

描述已自动生成

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

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

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

## Model-Based算法

### 隐因子模型（Latent Fator Model）

图片包含 屏幕截图

描述已自动生成

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

LFM与基于邻域的方法对比：

图片包含 屏幕截图

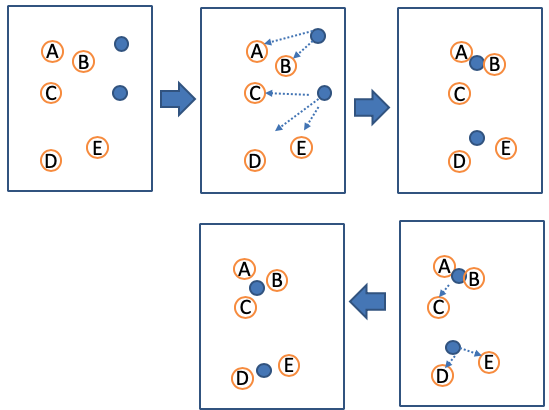
描述已自动生成

### 关联规则

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

### K-means聚类算法

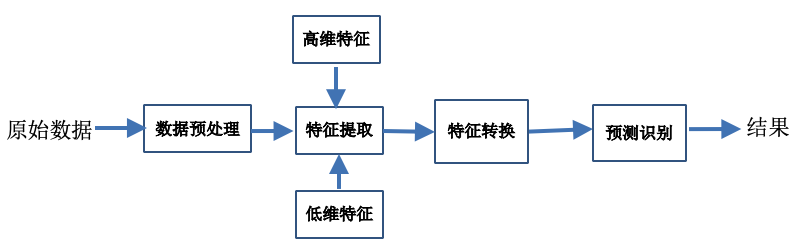
聚类算法在保险客户分群中起到了重要作用，K-means聚类算法首先会随机确定k个中心位置，位于空间中代表聚类中心的点，然后将各个数据项分配给最临近的中心点。待分配完之后，聚类中心就会移到分配给该聚类的所有节点的平均位置处，然后整个分配过程重新开始。这一过程会一直重复下去，直到分配过程不再产生变化位置。



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

# 基于CTR预估的推荐

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



## GBDT+LR实现高潜客户识别和Item排序

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

图片包含 标牌, 文字

描述已自动生成

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

## FM因式分解机

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

图片包含 物体

描述已自动生成

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

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

## 强化学习 - Clustering Of Bandit

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

图片包含 屏幕截图

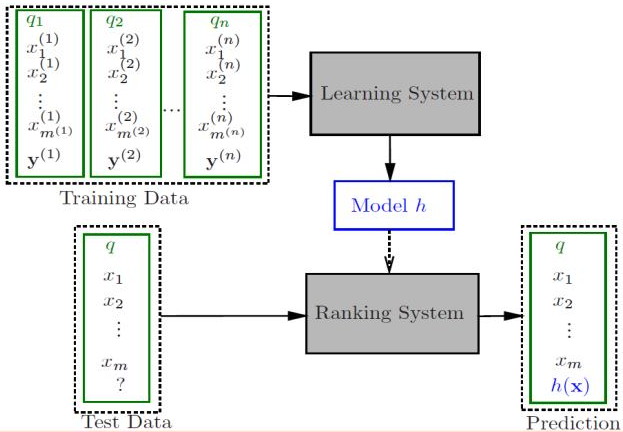
描述已自动生成

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

## 排序学习（Learning To Rank）

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

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



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

## 算法总结

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

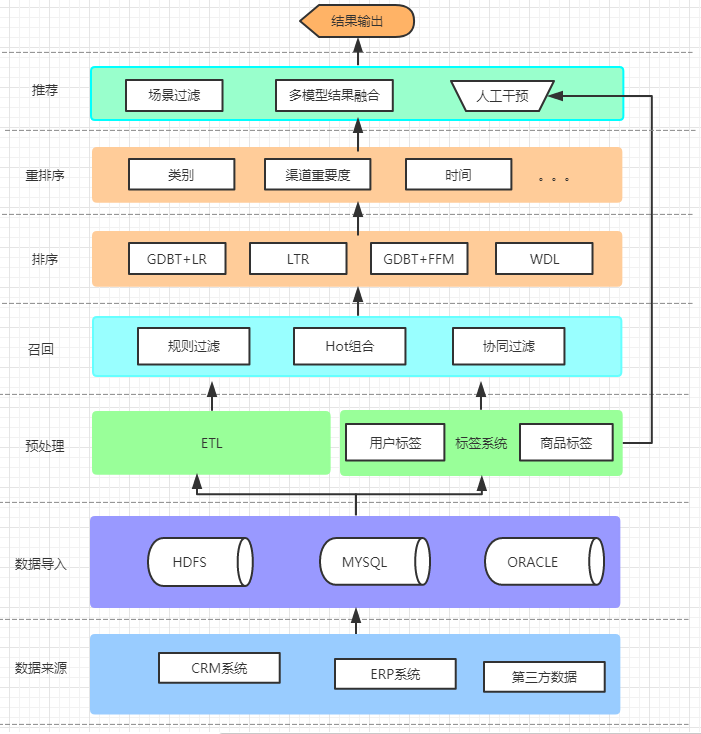
从业务角度看，Memory\_Based算法适合于显式反馈的业务场景，比如豆瓣电影和亚马逊图书，这些场景需要用户给自己接触的物品打分，而打分就直接对应用户的喜好度。但是在其他业务场景，比如金融和保险产品中，用户购买产品并不能直接得到相应的喜好度和评分，用户可能是为了别人买保险，或者只是生活需要。我们把后一种场景得到数据为隐式反馈数据。所以，通过隐式反馈数据最终只能推荐用户适合的物品，而非用户最终喜爱的物品。

**第三部分 案例**

# 电商案例

## 数据流程

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



### 召回（Recall）

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

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

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

基于地域的召回：一般用户用户冷启动，即网络行为比较少的新用户。

### 排序（Rank）

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

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

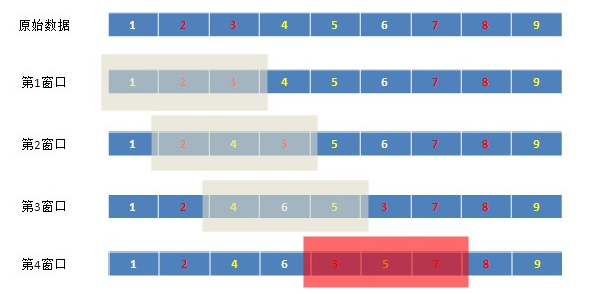
图片包含 屏幕截图

描述已自动生成

### 重排序（Rerank）

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

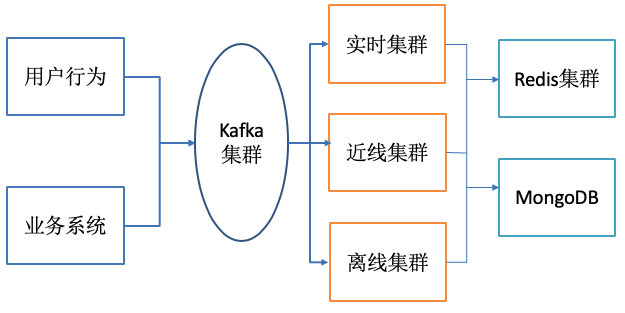
下面介绍一种基于滑动窗口的品类打散算法。模拟用户在浏览手机屏幕时的浏览窗口，通过参数设置窗口的大小，保证在窗口向后移动时，窗口内的商品如果出现同一品类，将该商品向后移出该窗口范围。



Scala代码实例：

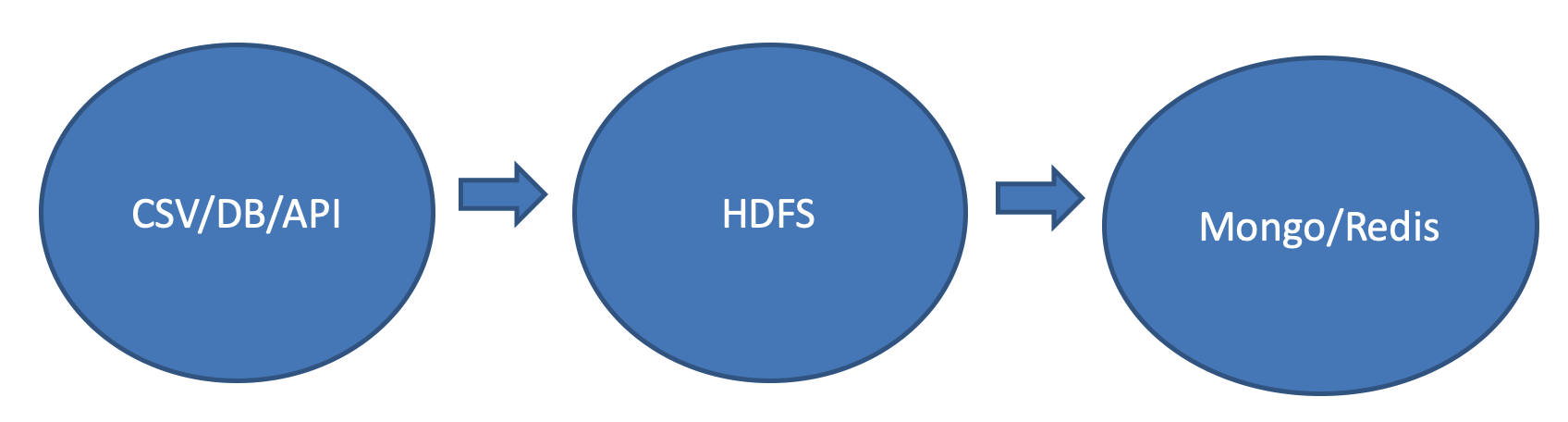
1. /\*\*
2. \* 推荐列表按品类打散
3. \* 品类窗口为w，即相邻W+1个物品不能为同一品类,default:1
4. \* 注意：调用此方法前需要保证品类足够多，否则结果会放弃重复品类的物品
5. \* @return (rank, item)
6. \*/
7. def sortByCate(ori: Array[(String, String)], W:Int=1): Set[(Int, String)] = {
8. var buffer = **new** mutable.ListBuffer[(String,String)]
9. val result = **new** mutable.ListBuffer[(Int, String)]
10. **for**(item <- ori) buffer += item
12. var windowCate = ""
13. var index = 0
14. var i = 1
15. **while** (buffer.size > 0) {
16. **if**(windowCate == buffer(index).\_2) {
17. index = index + W
18. }**else**{
19. val temp =  (i, buffer(index).\_1)
20. result += temp
21. i += 1
22. windowCate = buffer(index).\_2 //cate
23. buffer -= buffer(index)
24. index = 0
25. }
26. **if**(index >= buffer.size) buffer.clear()
27. }
28. result.toSet
29. }

## 系统架构与实现模块



### 离线计算

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



### 近线计算

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

图片包含 屏幕截图

描述已自动生成

### 实时计算

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

Spark Streaming 实时流处理，将数据流作为一张没有边界的表。

图片包含 屏幕截图

描述已自动生成

## 推荐场景

### 猜你喜欢

猜你喜欢是一种综合性推荐。



### 店铺推荐

当用户点击某个商品时，商品详情页周边和下方显示的本店铺商品推荐。

图片包含 屏幕截图

描述已自动生成

### 组合推荐

商品主页下方推荐的商品配套组合。

图片包含 屏幕截图

描述已自动生成

# 线下商超案例

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

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

## 数据流程

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

### 召回

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

### 排序

同上召回操作，接受召回结果进入排序模块。

## 推荐场景

### 复购推荐

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

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

图片包含 文字

描述已自动生成

### 新鲜推荐

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

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

通过客户分群形成的特征空间数据：

图片包含 屏幕截图

描述已自动生成

# 金融领域的推荐

保险案例为某大型银行保险电销业务。

## 基于客户细分的保险推荐

名单类型分析

图片包含 文字

描述已自动生成

信用卡分群

图片包含 屏幕截图

描述已自动生成

将银行所有的信用卡客户分成四类。

cluster-3：优质客户

cluster-2：潜力优质客户

cluster-1: 一般客户

cluster-4：劣质客户

保险客户分群

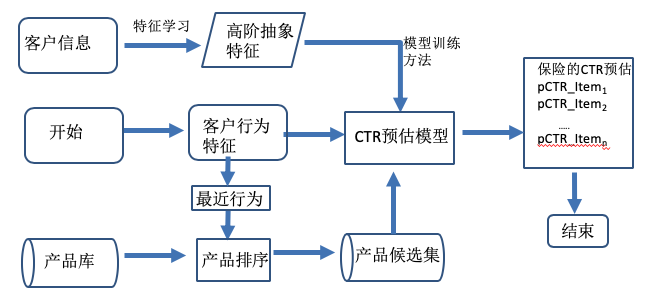


构建APP行为的回归模型

图片包含 屏幕截图

描述已自动生成

## 基于CTR预估的保险需求模型



图片包含 文字, 地图

描述已自动生成

## 集成学习下的融合模型

图片包含 标牌, 天空

描述已自动生成

# 文本推荐

新闻推荐系统

略