**大数据电商用户画像项目**

**第七部分◎教案**

|  |  |  |
| --- | --- | --- |
| **课程教案版本** | **日期** | **备注** |
| **V1.0** | **20190515** |  |
|  |  |  |

# 第7章：挖掘类标签之聚类算法基础实战

# 项目架构再梳理

## 项目描述

企业级360°全方位用户画像是基于电商平台进行设计和开发，是面向注册会员的偏好、行为习惯和 人口属性的画像还原，同时也包括对商品信息的画像还原。提供用户喜好和商品特征帮助营销平台提升营销的精准度，也方便个性化推荐系统快速准确的为每个用户推荐相关的商品。



1.标签体系

标签体系模块基于本体论建立，包括基础标签和组合标签两个子模块。基础标签维护了人口属性、商业属性、行为属性和用户价值4个方面。组合标签是在基础标签的基础上按TGI创建的，用于反映目标群体在特定研究范围的强势和弱势。

2.标签引擎

标签引擎模块是用户画像运转的核心实现，维护标签的具体实现逻辑，包括标签依赖的数据源、规则和模型的信息。同时提供每一个标签引擎运行状态的可视化监控。当Super User登录时会出现新增标签及其引擎的审核。

3.画像模块

画像模块是对实体（用户/物品）信息全貌可视化的精准还原，包括对个体的全方位还原和群体的全方位还原。

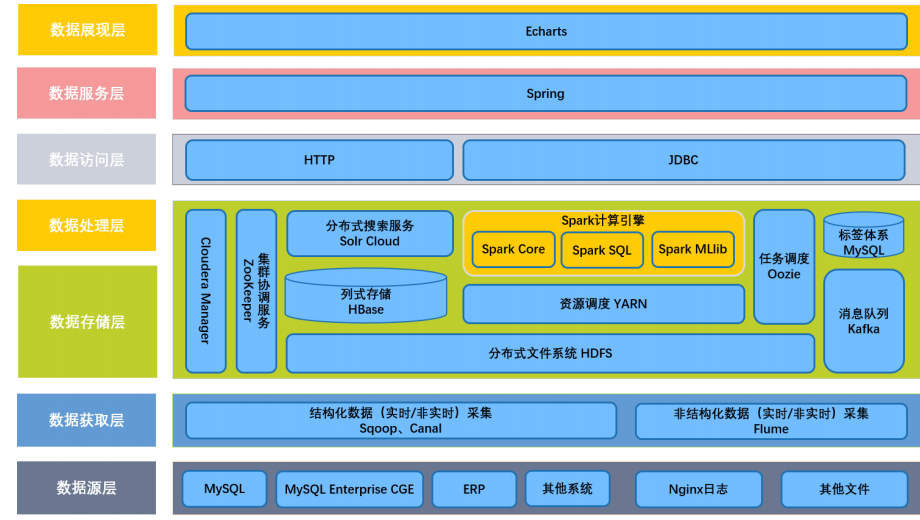
4.标签查询

标签查询模块是查看标签所覆盖的实体，用于查看标签体系中所有标签包含的商品详细信息和经过 脱敏后的用户信息。

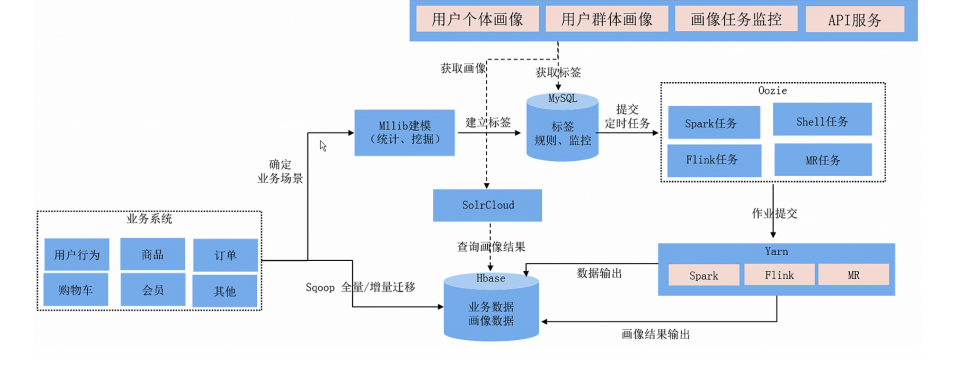
5.系统设置

系统设置模块主要包括用户管理、权限管理。

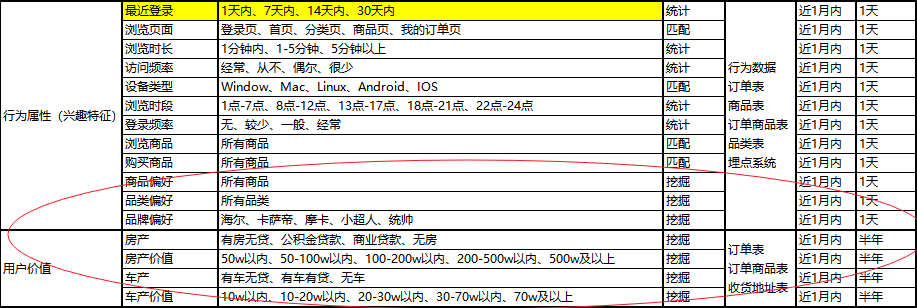
## 技术架构



## 项目功能架构



## 挖掘类标签的举例



**Spark机器学习聚类算法基础**

**教案**

|  |  |  |
| --- | --- | --- |
| **课程教案版本** | **日期** | **备注** |
| **V1.0** | **20190515** |  |
|  |  |  |

# 2.SparkMliib构建聚类算法

学习目标：

聚类算法分类

KMeans算法理论基础及改进思路

SparkMllib聚类算法实战

## SparkMllib聚类

聚类是一种无监督（无类标签）学习技术（包括聚类、属性约减的PCA），可以在事先不知道正确结果（即无类标信息或预期输出值）的情况下，发现数据本身所蕴含的结构等信息。

**聚类的目标**是发现数据中自然形成的分组，使得每个簇内样本的相似性大于其他簇内样本的相似性。

**聚类本质**上是一种分组方法，分组的标准是组内对象相似度尽可能高，而组间对象的相似度尽可能低。

可将聚类理解为：**将对象的集合进行分组的过程。**

### 2.1.SparkMllib聚类原理详解

#### 2.1.1聚类方法案例引入

1. 两个人在话筒前面同时说话，录音后发现这两个人的声音混杂在一起。因此我们需要区分两个人的声音，这时候我们依靠声音的频谱不同对声音进行不同的区分，当然如果实现并不知道每个人声音的特点的话，那么需要利用人们的声音数据进行训练，这就可以利用监督学习方式来实现。
2. 我们试想下面场景：不同客户的有不同的特点和需求，针对每个客户的特点选定不同的销售策略。我们可以对每个客户选取不同的特征，之后进行静态分类，即聚类。

**聚类在商业的应用**中还有如下，按照不同主题对文档、音乐、电影进行分组，或基于常见的购买行为，发现有相同的兴趣爱好顾客，并在此构建推荐引擎。

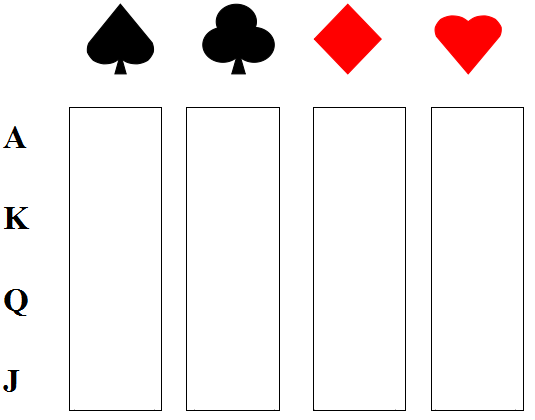
聚类就是要挖掘数据蕴含的相似性的结构信息。

下面通过一个例子，简单的说明相似性的结构信息。对于下面的16张扑克牌，如何将他们分组呢？



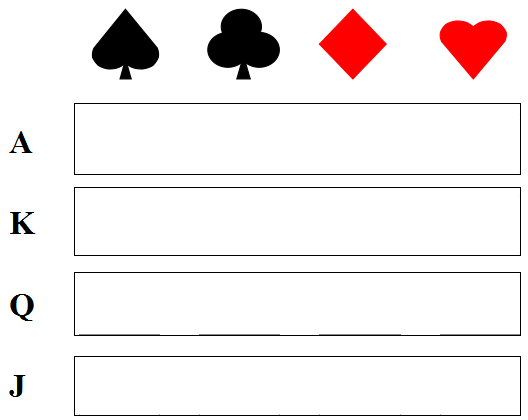
（1）同花色一组：

分为四组，每组里面花色相同，组与组之间花色不同



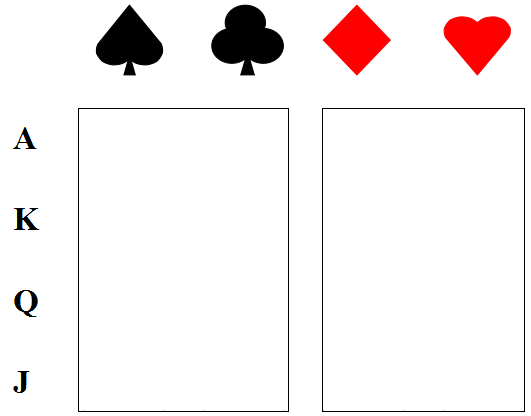
（2）同字符符号一组：

分为四组，字符符号相同为一组



（3）同颜色一组：

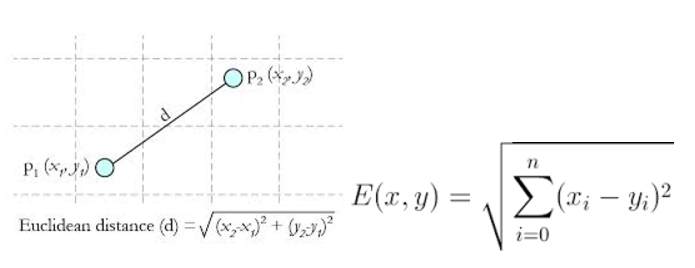
分为两组，字符颜色相同为一组



总结：无论选择哪种划分方法，关键在于我们怎样定义并度量“相似性”。

#### 2.1.2距离的度量（相似性度量）

**1. Euclidean Distance 定义**



      其他距离衡量：余弦值（cos）, 相关度 （correlation）, 曼哈顿距离 （Manhattan distance）

曼哈顿距离：纽约曼哈顿的街区比较平整，我们从P1起点到P2起点，首先横向跨越三个街区，在纵向跨越2个街区就达到P2点了。

**2. 其他距离度量的补充**

数序中的定义为：

n维特征空间向量， 的 距离（闵可夫斯基距离）定义为



* 欧几里得距离： 
* 曼哈顿距离：城市街区距离
* 上确界距离：对象属性之间的最大距离

**3. 余弦距离**

余弦相似度也常被称作为余弦距离。定义如下：



接下来通过例子来理解余弦距离

X=（3,2,0,5,0,0,0,2,0,0）

Y=（1,0,0,0,0,0,0,1,0,2）

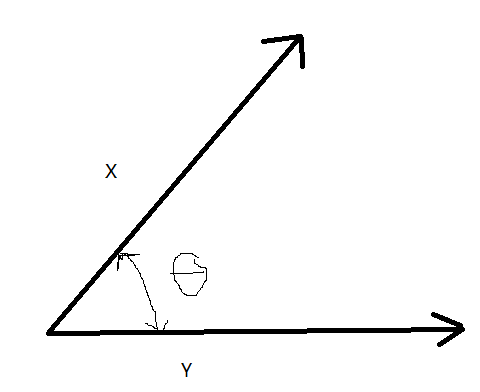
X\*Y=3\*1+2\*0+0\*0+5\*0+0\*0+0\*0+0\*0+2\*1+0\*0+0\*2=5

||X||=sqrt(3\*3+2\*2+0\*0+5\*5+0\*0+0\*0+0\*0+2\*2+0\*0+0\*0)=6.48

||Y||=sqrt(1\*1+0\*0+0\*0+0\*0+0\*0+0\*0+0\*0+1\*1+0\*0+2\*2)=2.45

Cos(x,y)=0.31

如下图所示，如果余弦相似度为1，则x与y之间的夹角为0度，此时的两个向量是完全一样的。如果余弦相似度为0，则x与y之间的夹角为90度，此时可以说两个向量不包含相同的词。



**4. 简单匹配系数**

简单匹配系数是最简单的二元属性的对象之间的相似性度量，设x和y是两个数据对象，都由n个二元属性组成。这样的两个对象的比较可生成四个量：



简单匹配数SMC定义为：

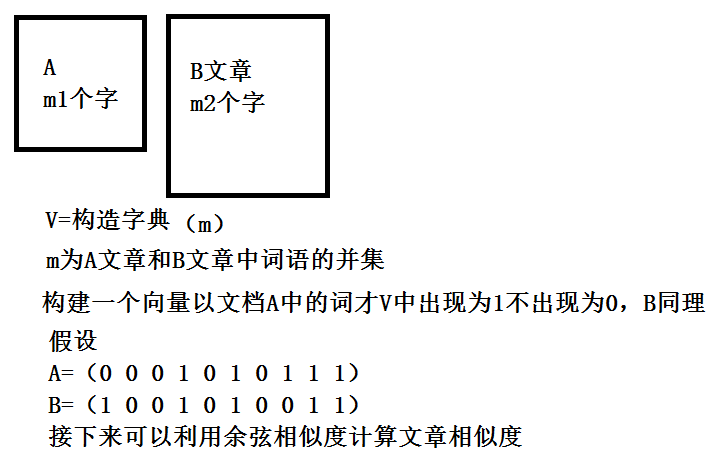


当然也可以度量值不匹配的匹配系数：



通过以下例子分析：

两篇文档的相似度求解



以X和Y为例

X=(1,0,0,0,0,0,0,0,0,0)

Y=(0,0,0,0,0,0,1,0,0,1)

f 01=2(x取0且y取1)， f 10=1(x取1且y取0)

f 00=7(x取0且y取0)， f 11=0(x取1且y取1)



注意：大家对**海明距离**也要有一点了解，他指的是两个码字的对应比特取值不同的比特数称为这两个码字的海明距离。如x=(1,0,0)，y=(1,1,1),从x到y的海明距离即为2（不相似的字有多少个），相当于后两个数值均要从0->1。

**5. Jaccard系数**



两个集合之间的相似性可以使用Jaccard来度量。

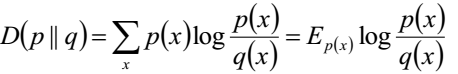
当分析购买商品数据时，由于未被顾客购买的商品数量远大于被其购买的商品数，像上述SMC的计算方式将会判断所有的事务都是类似的。所以引出了Jaccard系数来处理仅包含非对称的二元属性的对象。



上述例子使用jaccard度量的结果如下：

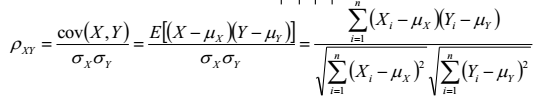


**6.相对熵（KL距离）**



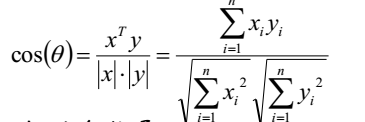
D(p||q)=0 表示p=q，极度相关

**7.Pearson相关系数**

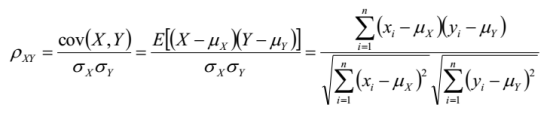


**8.余弦相似度与Pearson相关系数**

余弦定理：



这两个向量的相关系数是：



相关系数即将x、y坐标向量各自平移到原点后的夹角余弦

这即解释了为何文档间求距离使用夹角余弦，因为这一物理量表征了文档去均值化后的随机向量间的相关关系。

有很多的计算相似度方法，不局限在上面：

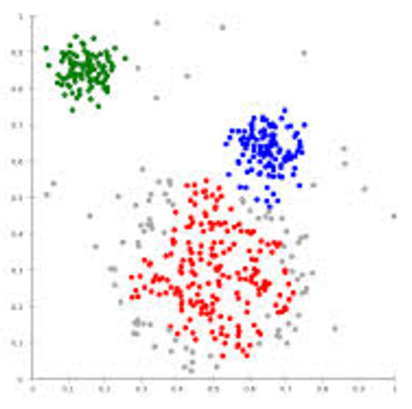
K(x,y)=exp(-|x-y|\*\*2) RBF 高斯径向基函数求解相似度

#### 2.1.3Kmean算法原理

##### （1）非监督中的Kmean算法分类

聚类(clustering) 属于非监督学习 (unsupervised learning),无类别标记(class label).

观察下图，相同类别的通过属性之间的相似性聚集在一起，算法中并未涉及类别标记的问题。



##### K-means 算法详解

K-几个聚类中心 Mean-均值，每次迭代的时候使用均值方式迭代

 Clustering 中的经典算法，数据挖掘十大经典算法之一

 算法接受参数 k ；然后将事先输入的n个数据对象划分为 k个聚类以便使得所获得的聚类满足：同一聚类中的对象相似度较高；而不同聚类中的对象相似度较小。

**算法思想：**

以空间中k个点为中心进行聚类，对最靠近他们的对象归类。通过迭代的方法，逐次更新各聚类中心的值，直至得到最好的聚类结果

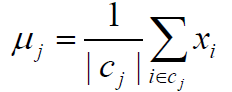
**算法描述：**

（1）随机适当选择c个类的初始中心；

（2）在第k次迭代中，对任意一个样本，求其到c各中心的距离，将该样本归到距离最短的中心所在的类；

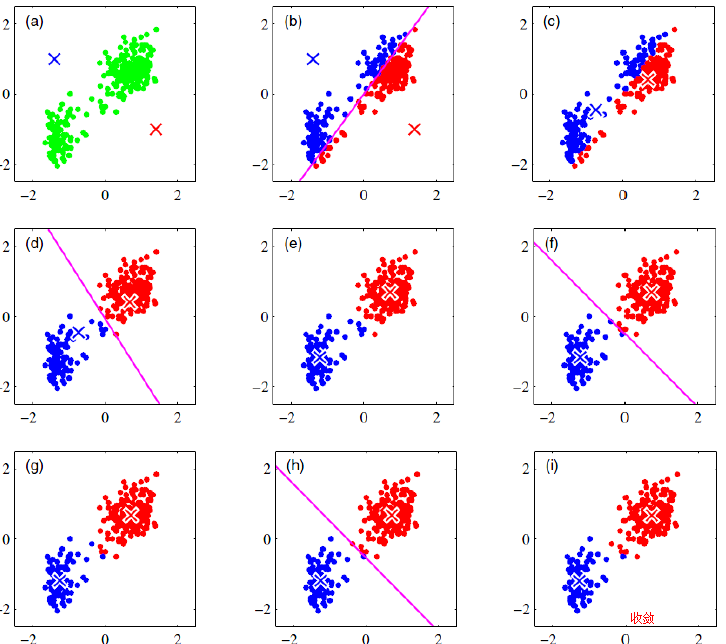


（3）利用均值等方法更新该类的中心值；



（4）对于所有的c个聚类中心，如果利用（2）（3）的迭代法更新后，聚类中心的值保持不变，则迭代结束，否则继续迭代。

终止条件：迭代次数/簇中心变化率/最小平方误差MSE



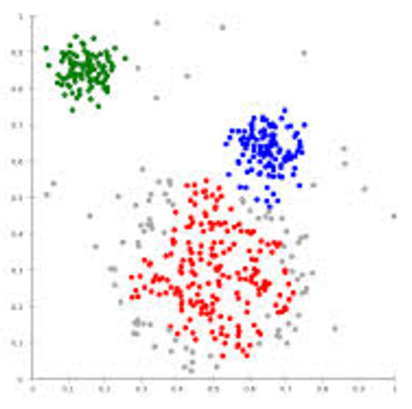
**算法核心步骤：**

**计算距离---归类-----计算均值---距离----归类----均值----距离--.........**

（1）求解第i个样本xi到达第j个聚类中心的距离，选择j最小的值赋值给第i个样本的标记值。

（2）对于属于第j个样本的所有的xj求均值，作为第j个样本的新的聚类中心，聚类中心改变了，重新计算标记信息，标记变了重新计算距离。

**算法流程：**



        输入：k, data[n];

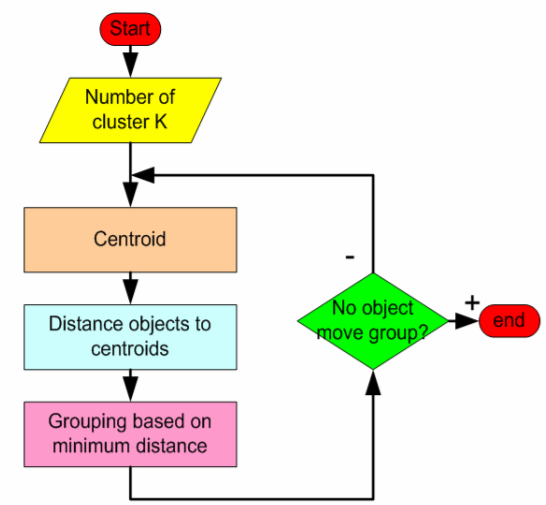
       （1） 选择k个初始中心点，例如c[0]=data[0],…c[k-1]=data[k-1];

       （2） 对于data[0]….data[n], 分别与c[0]…c[k-1]比较，假定与c[i]差值最少，就标记为i;

       （3） 对于所有标记为i点，重新计算c[i]={ 所有标记为i的data[j]之和}/标记为i的个数；

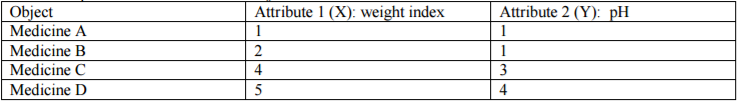
      （4） 重复(2)(3),直到所有c[i]值的变化小于给定阈值。

##### （3）KMeans算法举例

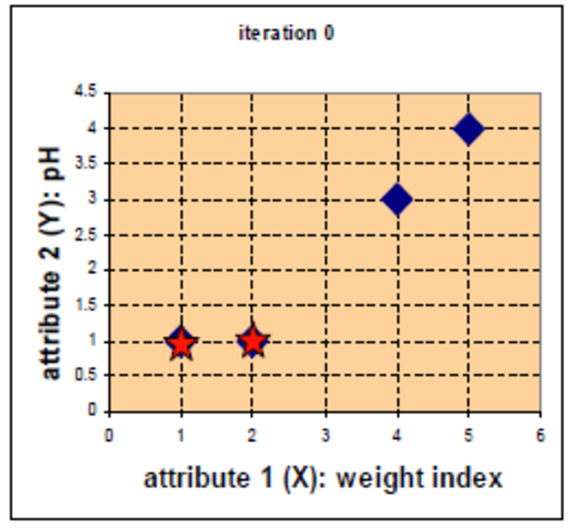
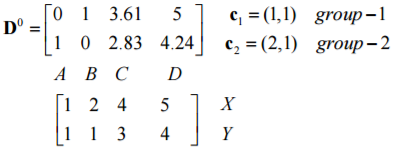


通过下面的例子理解上面的算法迭代过程。

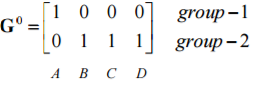
示例：假设我们有4个对象作为训练数据点，每个对象有2个属性。 每个属性表示对象的一个维度的坐标。



每个药物代表一个具有两个特征（X，Y）的点，我们可以将其在二维坐标系中表示，如下图所示。

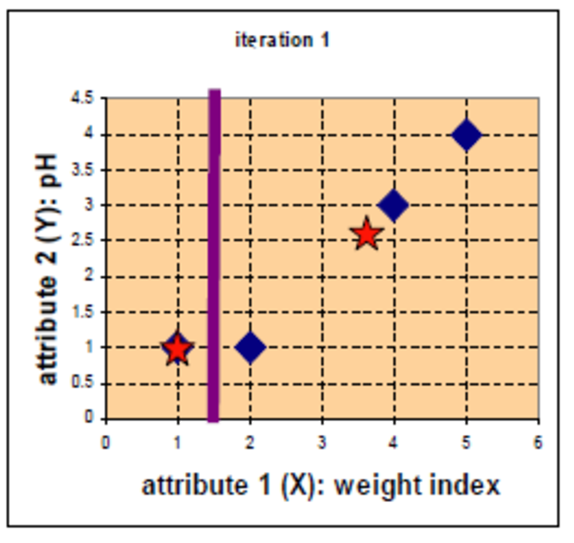
 

1. 质心的初始值：假设我们使用药物A和药物B作为第一重心。 让c1和 c2表示质心的坐标，则c1 =(1,1)和c2 =(2,1)
2. 对象 - 质心距离：我们计算每个对象的聚类质心之间的距离。右侧D(0)
3. 距离矩阵中的每列都表示对象。（右侧图） 距离矩阵的第一行对应于每个物体与第一重心的距离，第二行是距离是每个对象与第二个重心的距离。 例如，药物三c3=(4,3)距离第一个质心c1=(1,1)的距离为sqrt((4-1)^2+(3-1)^2)=3.61;到第二个质心c2 =(2,1)的距离sqrt((4-2)^2+(3-1)^2)=2.83。
4. 对象聚类：我们根据最小距离分配每个对象。 因此，药A是分配到第1组，药物B至组2，药物C至组2和药物D至组2当且仅当对象被分配给该组时，下面的组矩阵的元素是1表示属于哪一类。

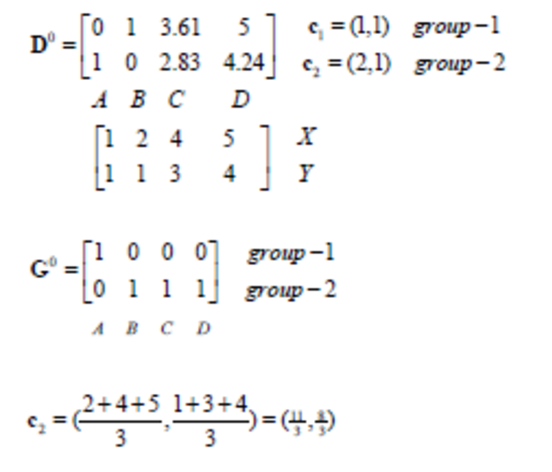


1. 迭代1次，确定质心：知道每个组的成员，现在我们计算新的基于这些新成员的每个组的质心。 第1组只有一个成员质心保持在c1=（1,1）。 组2现在有三个成员，因此重心是平均数三位成员之间的协调。

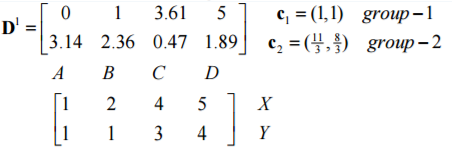




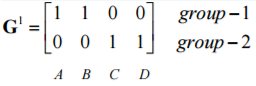
备注：



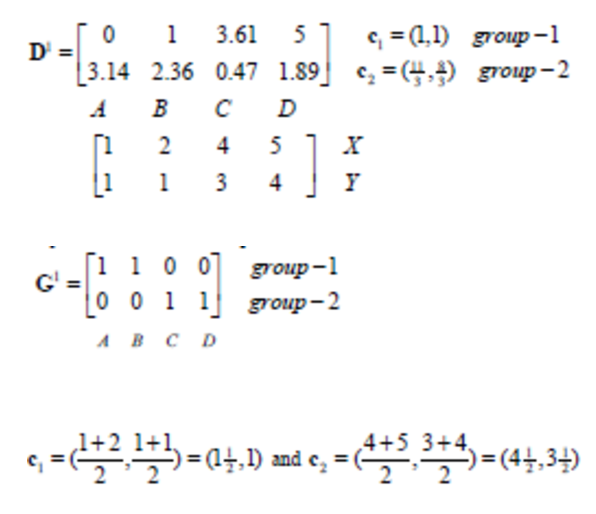
1. 迭代1，对象-质心距离：下一步是计算所有对象的距离新的质心。与步骤2类似，我们在迭代1中有距离矩阵



1. 迭代1，对象聚类：与步骤3类似，我们基于最小值分配每个对象距离。 基于新的距离矩阵，我们将药物B移动到组1，而所有其他对象仍然存在。 组矩阵如下所示：



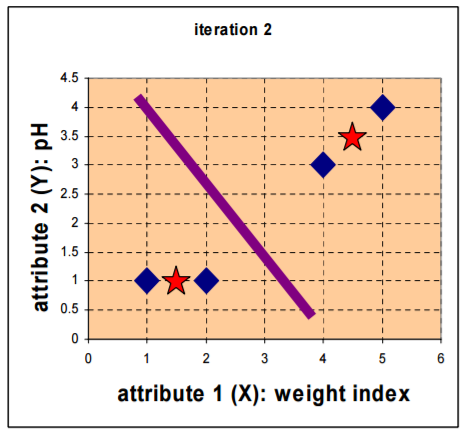
备注：



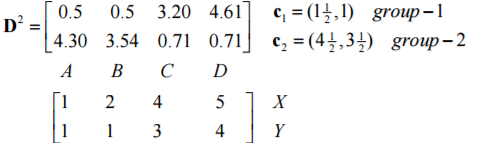
1. 迭代2，确定质心：现在我们基于以前迭代的聚类重复步骤（5）来计算新的质心坐标。 组1和组2都有两个成员，因此

新的质心是

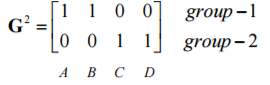




1. 迭代2，对象 - 质心距离：再次重复步骤2，我们有新的距离矩阵迭代2为：

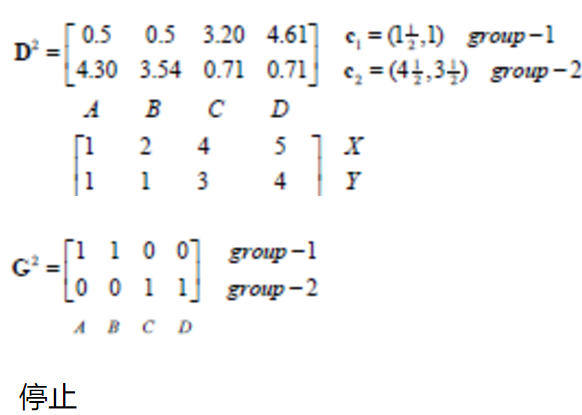


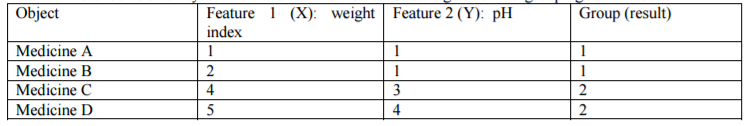
1. 迭代2，对象聚类：再次，我们根据最小距离分配每个对象。



我们得到的结果。 比较上一次迭代的分组和这个迭代显示对象不再移动组。 因此，k均值聚类的计算具有达到稳定，不再需要迭代。 我们得到最终的分组作为结果。

备注:





**总结：**

简单来说，K均值是一种基于属性/特征将对象分类或分组为K的算法组数。 K是正整数。 通过最小化平方和来完成分组数据与对应的集群中心之间的距离。 因此，K均值聚类的目的是对数据进行分类。

一开始，我们确定簇K的数量和假设这些聚类的质心或中心。 我们可以把任何随机选择对象作为初始质心或者第一个K对象（第一行数据）也可以作为初始质心。

那么K均值算法将会执行以下三个步骤，直到收敛

迭代直到稳定（=没有对象移动到其他分组）：

1.确定质心坐标

2.确定每个物体与重心（中心，或选取点）的距离

3.根据最小距离对对象进行分组（找到最近的质心）

##### （4）K-Mean性能评价指标（误差函数）

一种度量k-means算法的聚类效果的指标是误差平方和SSE（Sum of Squared-Error）。



|  |  |
| --- | --- |
| 符号 | 描述 |
| x | 样本 |
| Ci | 第i个簇 |
| ci | 簇Ci的质心（该簇的均值） |
| mi | 第i个簇的样本个数 |
| K | 簇的个数 |

SSE表示**数据样本与它所属的簇中心之间的距离（差异度）平方之和**。直观的来说，SSE越小，表示数据点越接近它们的中心，聚类效果越好。因为对误差取了平方，更加重视那些远离中心的点。

可证明：使簇的SSE最小的质心是均值





在聚类算法的性能评价中，还有下面会介绍的ARI指数（Adjusted Rand Index），结果介于[0,1]之间，值越靠近1性能越好。

##### （5）K-Mean算法特点

优点：速度快，简单

对处理大数据集，该算法保持可伸缩性和高效率

当簇近似为高斯分布时，它的效果较好。

缺点：最终结果跟初始点选择相关，容易陷入局部最优，需直到k值

k均值算法中k是实现给定的，这个k值的选定是非常难估计的。

k均值的聚类算法需要不断地进行样本分类调整，不断地计算调整后的新的聚类中心，当数据量大的时候，算法开销很大。

k均值是求得局部最优解的算法，所以对于初始化时选取的k个聚类的中心比较敏感，不同点的中心选取策略可能带来不同的聚类结果。比如实际分类5类的情况却只进行了3均值的聚类。

对噪声点和孤立点数据敏感。

KMeans一般是其他聚类方法的基础算法，如谱聚类。

##### （6）对K-Means算法思考

（1）均值的影响

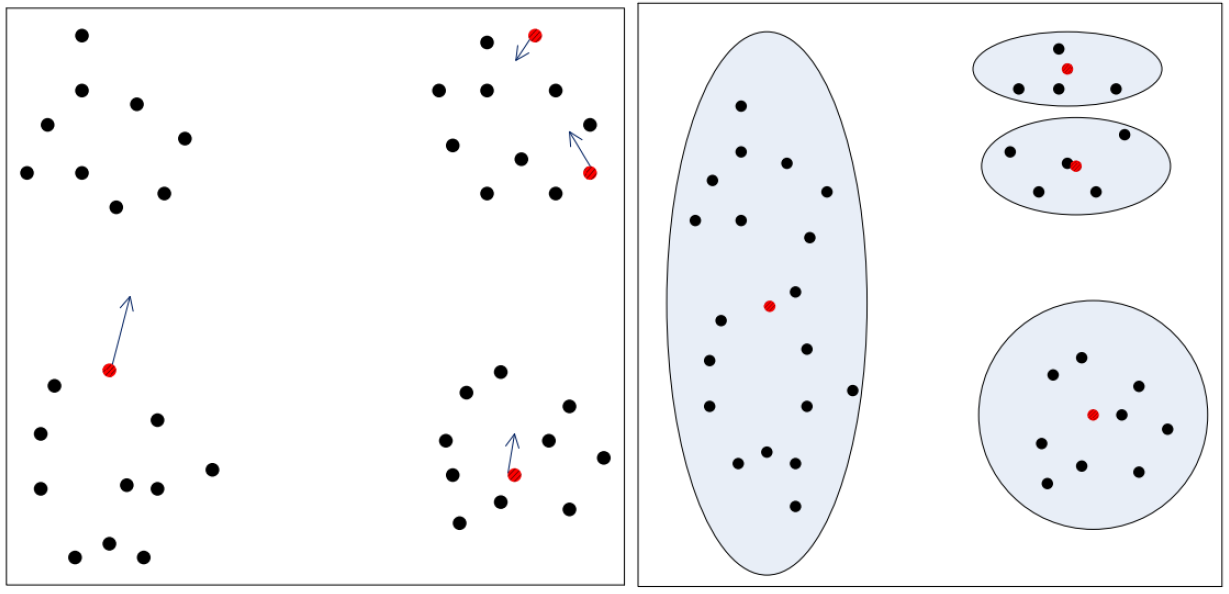
K-Means将簇中所有的点均值作为新质心，若簇中含有异常点，将导致均值偏离严重，如下面数据

（1，2，3，4，100）均值22，距离大多数点1，2，3距离较远

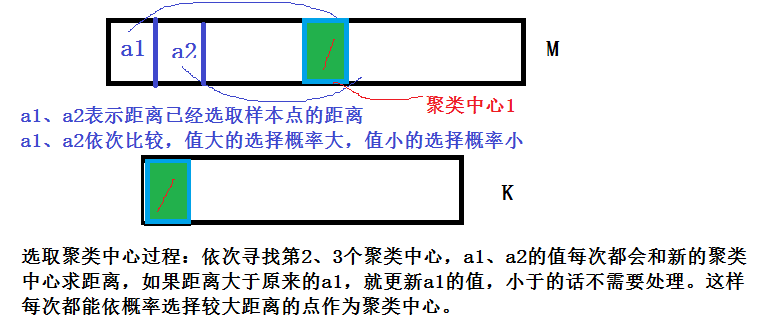
改为求数组的中位数3，在该实例中比较稳妥，这也是下面的K-Mediods聚类。

（2）初值的选定

目的：让聚类中心离得远一些

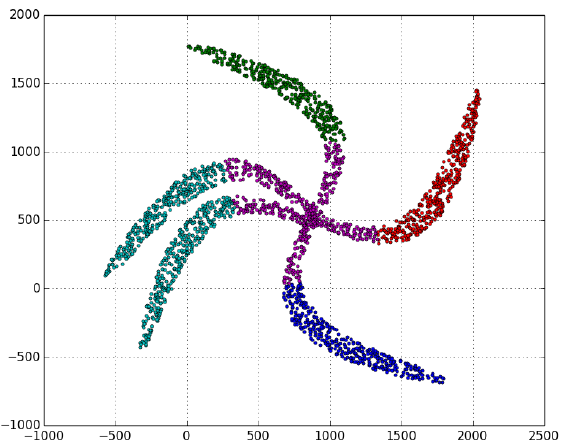
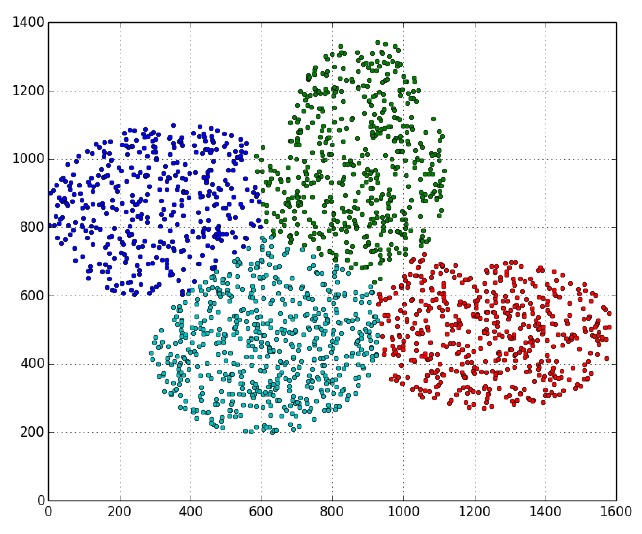


**方案：（kmeans++---以距离作为权值来选择初值，一般无法收敛全局最优解）**



（3）K-Means假设

K-Means本质上认为样本服从混合高斯分布GMM（如左图），每一个高斯分布的方差都相等，这是K均值的假设前提，如果样本点不能很好的满足GMM，聚类的效果会下降（如右图）。

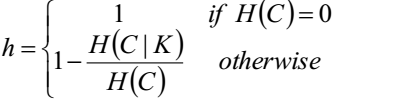


#### 2.1.4聚类结果相关衡量指标

常用的聚类指标如下：

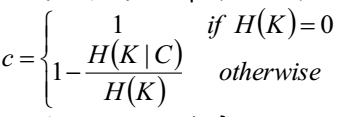
* 均一性

一个簇只包含一个样本，则满足均一性



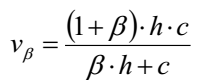
* 完整性

同类别的样本被分配到相同簇中，则满足完整性。

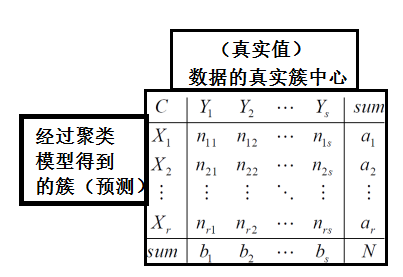


* V-measure

均一性和完整性的加权平均



假设聚类结果数据矩阵如下：定义ARI,AMI和轮廓系数



* ARI

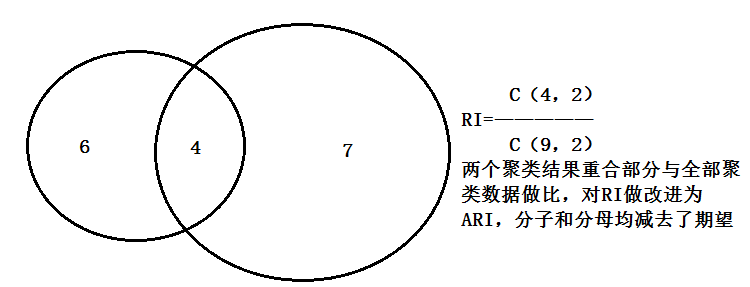
数据集S共有N个元素，两个聚类结果分别是



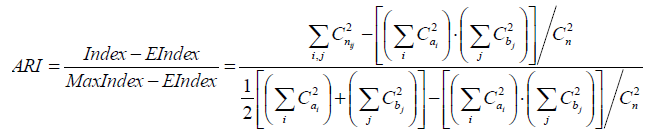
其中X和Y元素的个数为：



RI的简单理解



则ARI系数为



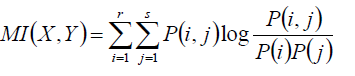
其中

该系数越近于1聚类结果越接近与真实结果，越接近于0结果越差。前提是需要给定真实的结果。

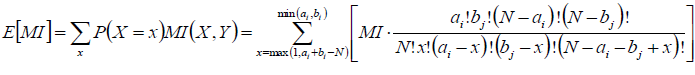
* AMI

类比与ARI，AMI使用信息熵作为衡量聚类效果的标准，也需要提前给定真实的聚类结果。

* 互信息：



X服从超几何分布，求互信息期望



借鉴ARI有AMI



* 轮廓系数silhouette\_score

计算样本i到同簇其他样本的平均距离ai，ai越小说明样本i越应该被聚类到该簇。将ai称之为样本i的簇内不相似度。

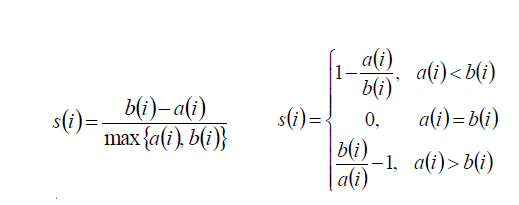
簇C中所有的样本的ai的均值称为簇c的簇不相似度。

计算样本i到其他簇其他样本的平均距离bi，bi越大说明样本i越不属于其他簇。将bi称之为样本i的簇间不相似度。



轮廓系数s(i)定义为：

S(i)越接近1，则说明样本i聚类合理（看公式中当ai=0（簇内不相似度为0），bi等于无穷大的时候（簇间的不相似度为无穷大，簇间极其不相似），结果为1）；s接近-1，则说明样本i更应该分类到另外的簇上；若s(i)为0则说明样本i在两个簇的边界上。



所有的样本的si的均值称为聚类结果的轮廓系数，是该聚类是否合理、有效的度量。区别于ARI\AMI是不需要真实的簇数据。

**Spark中引用指标方法：**

**（1）WSSSE：该方法是集中平方误差和指标，它通过对计算数据集中所有的点到簇中心点的距离的平方和来衡量聚类的效果。一般来说簇内的点距离中心的距离越小，说明聚类的效果越好。但是在实际使用过程中，必须还要考虑聚类结果的可解析性，比如如果考虑极限情况，当簇的个数和数据集的大小一样时，每个点都是聚类中心，每个类都只有一个点，此时簇内距离平方和为0，但是这样的聚类效果显然没有意义。WSSSE适合于基于距离的聚类算法，如K-Means的效果评估，不适合DBSCAN这类基于密度的聚类算法。**

**（2）Purity**

纯度Purity评价方法定义pij为聚类的成员属于类别的概率。假设聚类数目为k，整个样本数据集的大小为m，则聚类划分的Purity的计算公式为：



该评价方法简单，其在类平衡的数据集上可以使用，但是如果数据集不平衡的数据集上使用则会有一定的舞蹈作用。

#### 2.1.5聚类算法性能分析与选择

如何评估聚类算法的性能，特别是应用在没有标注类别的数据集上的时候，针对不同的数据特点，大致有两种方式

1）如果被用来评估的数据本身带有正确的类别信息，那么使用ARI，该指标和分类指标的准确率Accurary一致，同时也兼顾到类簇无法和分类标记一一对应的问题。

2）如果被用于评估的数据没有所属类别，那么我们习惯使用轮廓系数来度量聚类结果的质量。轮廓系数同时兼顾了聚类的凝聚度和分离度，用于评估聚类的效果并且取值范围为[-1,1]。轮廓系数越大，表示聚类效果越好。计算步骤如下：

（1）对于已经聚类数据中第i个样本，计算与其同一个类簇内的所有其他样本距离的平均值，记作，用于量化簇内的凝聚度

（2）选取外的一个簇b，计算与簇b中所有样本的平均距离，遍历所有其他簇，找到最近的这个平均距离，记作，用于量化簇之间的分离度

（3）对于样本，轮廓系数为

（4）最后对所有样本X求出平均值即为当前聚类结果的整体轮廓系数。

由轮廓系数的计算公式，不难发现，如果，说明与其簇内元素的平均距离大于最近的其他簇，表示效果聚类不好；

如果趋于0，或者足够大，那么趋近于1，说明聚类效果比较好。

### 2.2SparkMLlib实战KMeans聚类算法

#### 2.2.1SparkMllib-KMean原理及实战

SparkMLLIb中实现的Kmeans是通过**KMeans++**实现的。

接下来分析KMneans++思路， k-means++算法选择初始选择聚类中心的基本思想就是：**初始的聚类中心之间的相互距离要尽可能的远。**

* 从输入的数据点集合中随机选择一个点作为第一个聚类中心
* 对于数据集中的每一个点x，计算它与最近聚类中心(指已选择的聚类中心)的距离D(x)
* 选择一个新的数据点作为新的聚类中心，选择的原则是：D(x)较大的点，被选取作为聚类中心的概率较大
* 重复2和3直到k个聚类中心被选出来
* 利用这k个初始的聚类中心来运行标准的k-means算法

接下来继续SparkMLLIb中**实现KMeans的参数**：

K-means是最常用的聚类算法之一，它将数据点聚类为预定义数量的聚类。该spark.mllib实现包括一个名为kmeans ||的k-means ++方法的并行变体 。实现中包含以下参数：

* k是所需簇的数量。
* maxIterations是要运行的最大迭代次数。
* initializationMode指定随机初始化或通过k-means ++初始化。
* runs是运行k-means算法的次数（k-means不保证找到全局最优解，并且当在给定数据集上多次运行时，算法返回最佳聚类结果）。
* initializationSteps确定k-means ||中的步数。
* epsilon确定我们认为k-means已收敛的距离阈值。
* initialModel是一组用于初始化的可选集群中心。如果提供此参数，则仅执行一次运行。

下面，首先加载和解析数据，使用该 KMeans对象将数据聚类为两个集群。将期望的簇的数量传递给算法。然后在平方误差的集合和（**WSSSE**）内计算。算法可以通过增加k来减少此错误度量。实际上，最优k通常是WSSSE图中存在“肘(elbow)”的k。

代码：

import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}

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

// Load and parse the data

val data = sc.textFile("data/mllib/kmeans\_data.txt")

val parsedData = data.map(s => Vectors.dense(s.split(' ').map(\_.toDouble))).cache()

// Cluster the data into two classes using KMeans

val numClusters = 2

val numIterations = 20

val clusters = KMeans.train(parsedData, numClusters, numIterations)

// Evaluate clustering by computing Within Set Sum of Squared Errors

val WSSSE = clusters.computeCost(parsedData)

println("Within Set Sum of Squared Errors = " + WSSSE)

// Save and load model

clusters.save(sc, "target/org/apache/spark/KMeansExample/KMeansModel")

val sameModel = KMeansModel.load(sc, "target/org/apache/spark/KMeansExample/KMeansModel")

#### 2.2.2SparkMl-KMean原理及实战

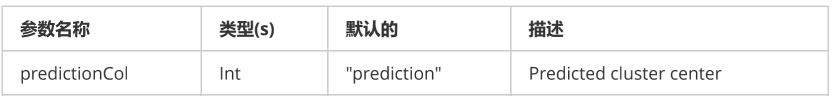
k-means 是其中一个最常用的聚类算法，它将数据点聚成一个预定义的簇数。MLlib实现包括一个并行化变体k-means++方法称为kmeans.

KMeans被实现为Estimator并生成一个KMeansModel作为基本模型。

输入列：



输出列：



代码如下：

import org.apache.spark.ml.clustering.KMeans

// Loads data.

val dataset = spark.read.format("libsvm").load("data/mllib/sample\_kmeans\_data.txt")

// Trains a k-means model.

val kmeans = new KMeans().setK(2).setSeed(1L)

val model = kmeans.fit(dataset)

// Evaluate clustering by computing Within Set Sum of Squared Errors.

val WSSSE = model.computeCost(dataset)

println(s"Within Set Sum of Squared Errors = $WSSSE")

// Shows the result.

println("Cluster Centers: ")

model.clusterCenters.foreach(println)

#### 2.2.3(实战) K-means药物聚类代码实战

数据格式：

Weightindex PH值

1.0 1.0  
 2.0 1.0  
 4.0 3.0  
 5.0 4.0

代码如下：

import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}

import org.apache.spark.mllib.linalg.{Vector, Vectors}

import org.apache.spark.rdd.RDD

import org.apache.spark.{SparkConf, SparkContext}

/\*\*

\* 目的-四个样本的药品的聚类

\* 1-准备SPark环境

\* 2-准备数据

\* 3-准备算法

\* 4-建立模型

\* 5-模型预测分析

\* 6-打印数据的聚类中心

\*/

object SparkMllibMediem {

def main(args: Array[String]): Unit = {

// \* 1-准备SPark环境

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkMllibMediem")

val sc = new SparkContext(conf)

// \* 2-准备数据

val datapath = "D:\\BigData\\Workspace\\SparkMllibLesson\\src\\main\\scala\\sparkmllib\_part4\\data\\medium.txt"

val tranData: RDD[Vector] = sc.textFile(datapath).map(x=>Vectors.dense(x.split(" ").map(\_.toDouble))).cache()

// \* 3-准备算法

// \* 4-建立模型

val model: KMeansModel = KMeans.train(tranData,2,3)

// \* 5-模型预测分析

// \* 6-打印数据的聚类中心

println(model.clusterCenters.mkString(","))//[1.5,1.0],[4.5,3.5]

println(model.computeCost(tranData))//1.5

}

}

#### 2.2.4(实战) K-means经纬度聚类代码实战

Spark MLlib K-means 算法的实现在初始聚类点的选择上，借鉴了 K-means++ 实现。K-means++ 算法在初始点选择上遵循一个基本原则: 初始聚类中心点相互之间的距离应该尽可能的远。基本步骤如下:

第一步，从数据集 X 中随机选择一个点作为第一个初始点。

第二步，计算数据集中所有点与最新选择的中心点的距离 D(x)。

第三步，选择下一个中心点，使得IMG_256最大。

第四部，重复 (二),(三) 步过程，直到 K 个初始点选择完成。

Spark的Kmeans算法参数：

参数的含义解释如下：

* + k 表示期望的聚类的个数。
  + maxInterations 表示方法单次运行最大的迭代次数。
  + runs 表示算法被运行的次数。K-means 算法不保证能返回全局最优的聚类结果，所以在目标数据集上多次跑 K-means 算法，有助于返回最佳聚类结果。
  + initializationMode 表示初始聚类中心点的选择方式, 目前支持随机选择或者 K-means||方式。默认是 K-means||。
  + initializationSteps表示 K-means||方法中的部数。
  + epsilon 表示 K-means 算法迭代收敛的阀值。
  + seed 表示集群初始化时的随机种子。

通常应用时，我们都会先调用 KMeans.train 方法对数据集进行聚类训练，这个方法会返回 KMeansModel 类实例，然后我们也可以使用 KMeansModel.predict 方法对新的数据点进行所属聚类的预测，这是非常实用的功能。

（1）数据集

数据格式以及意义：

111,30.655325,104.072573,173749

111,30.655346,104.072363,173828

111,30.655377,104.120252,124057

111,30.655439,104.088812,142016

列一：出租车ID

列二：经度

列三：纬度

列四：时间（例如：142016表示14点20分16秒）

（2）代码  
 通过分析出租车数据，然后使用KMeans对经纬度进行聚类，然后按照（类别，时间）进行分类，再统计每个类别每个时段的次数。

步骤：

1.整理数据，分割成训练数据和测试数据，且使其符合KMeans模型训练的格式

2.使用训练好的模型对测试数据进行预测，然后对结果以（类别，小时时间 ）进行count统计，结果为每个类别每个小时的总次数。

代码部分：

import org.apache.spark.ml.clustering.KMeans

import org.apache.spark.ml.feature.VectorAssembler

import org.apache.spark.sql.SparkSession

import org.apache.spark.sql.types.{StructField, StructType}

import org.apache.spark.sql.types.\_

import org.apache.spark.sql.functions.\_

object Tax1 {

def main(arg:Array[String]):Unit={

val spark = SparkSession.builder().appName("Taxi1").master("local[\*]").getOrCreate()

//为读取的数据创建schema

val taxiSchema = StructType(Array(

StructField("id",IntegerType,true),

StructField("tid",DoubleType,true),

StructField("lat",DoubleType,true),

StructField("time",StringType,true)

))

val path = "/home/enche/data/taxi.csv"

val data = spark.read.schema(taxiSchema).csv(path)

//将tid和lat转换成训练使用的Vector类型

val assembler = new VectorAssembler()

val tid\_lat = Array("tid","lat")

assembler.setInputCols(tid\_lat).setOutputCol("feature").transform(data)

//按照8：2的比例随即分割数据，分别用于训练和测试

val splitRait = Array(0.8, 0.2)

val Array(train, test) = data.randomSplit(splitRait)

//建立Kmeans，设置类别数为10

val kmeans = new KMeans().setK(10).setFeaturesCol("feature").setPredictionCol("prediction")

//模型训练

val model = kmeans.fit(train)

//使用模型预测 测试数据

val testResult = model.transform(test)

//导入隐式转换，不然$"time"会出现错误 $ not e member of StringContext

import spark.implicits.\_

val time\_prediction = testResult.select(substring($"time", 0, 2).alias("hour"), $"prediction")

time\_prediction.groupBy("hour","prediction").agg(("prediction","count")).orderBy(desc("count")).filter(row=>row.getAs(0)==15).take(10)

}

}

#### 2.2.5Iris数据的不同K值选择方法

package sparkmllib\_day05\_clustring  
  
import org.apache.spark.ml.clustering.{KMeans, KMeansModel}  
import org.apache.spark.ml.feature.{MinMaxScaler, MinMaxScalerModel, VectorAssembler}  
import org.apache.spark.sql.{DataFrame, Dataset, Row, SparkSession}  
  
/\*\*  
 \* DESC:   
 \* Complete data processing and modeling process steps:  
 \*  
 \*/  
object iris\_kmeans {  
 def main(args: Array[String]): Unit = {  
 // \* 1-准备环境  
 val spark: SparkSession = SparkSession.builder().appName("SparkMLLibCarKMeansAnalysis").master("local[\*]").getOrCreate()  
 spark.sparkContext.setLogLevel("WARN")  
 // \* 2-准备数据  
 val datapath = "D:\\BigData\\Workspace\\SparkMachineLearningTest\\SparkMllib\_FeatureEngineering\\src\\main\\scala\\cn\\itcast\\iris.csv"  
 val data1: DataFrame = spark.read.format("csv").option("header", "true").load(datapath)  
 data1.show(false)  
 data1.printSchema()  
  
 val data = data1.select(data1("sepal\_length").cast("Double"),  
 data1("sepal\_width").cast("Double"),  
 data1("petal\_length").cast("Double"),  
 data1("petal\_width").cast("Double"))  
  
  
 data.printSchema()  
 //sepal\_length,sepal\_width,petal\_length,petal\_width,class  
 // \* 3-特征工程  
 //1-将经度和纬度数据整合为一起  
 val vectran: VectorAssembler = new VectorAssembler().setInputCols(Array("sepal\_length", "sepal\_width", "petal\_length", "petal\_width")).setOutputCol("features")  
 val datatrans1: DataFrame = vectran.transform(data)  
  
  
 // 3-特征工程---最大值最小化的处理------【0,1】区间  
 val sclaer: MinMaxScaler = new MinMaxScaler().setInputCol("features").setOutputCol("scaledfeatures")  
 val scalerModel: MinMaxScalerModel = sclaer.fit(datatrans1)  
 val datatrans: DataFrame = scalerModel.transform(datatrans1)  
  
  
 val Array(trainset, testset): Array[Dataset[Row]] = datatrans.randomSplit(Array(0.8, 0.2), 123L)  
 // \* 4-建模实战  
 //loss很高  
 // val kmeans: KMeans = new KMeans().setFeaturesCol("features").setPredictionCol("predictions").setK(3)  
 //loss降低  
// val kmeans: KMeans = new KMeans().setFeaturesCol("scaledfeatures").setPredictionCol("predictions").setK(3)  
// val kmeanModel: KMeansModel = kmeans.fit(trainset)  
  
  
 val ks:Array[Int] = Array(2,3,4)  
 ks.foreach(  
 cluster =>{  
 val kmeans: KMeans = new KMeans().setFeaturesCol("scaledfeatures").setPredictionCol("predictions")  
 .setK(cluster)  
 val kmeanModel: KMeansModel = kmeans.fit(trainset)  
 // \* 5-预测分析  
 val testResult: DataFrame = kmeanModel.transform(testset)  
 println("wssse，when cluster=", cluster,kmeanModel.computeCost(testResult)) //(wssse,2.5189564869286865)  
 println(kmeanModel.clusterCenters.mkString(","))  
 }  
 )  
  
  
  
// \* 5-预测分析  
// val testResult: DataFrame = kmeanModel.transform(testset)  
// testResult.show()  
  
// testResult.groupBy("petal\_length", "predictions").  
// agg(("predictions", "count")).show()  
  
// \* 6-模型校验分析-wssse打印结果等  
// println("wssse", kmeanModel.computeCost(testResult)) //(wssse,2.5189564869286865)  
// println(kmeanModel.clusterCenters.mkString(","))  
 }  
}

### 2.3其他聚类算法

K-Means算法问题：依赖初始聚类中心选择，均值的影响及KMeans对数据的要求。

#### 2.3.1聚类的其他算法

**基于原型的聚类：**假设聚类结构能通过一组原型刻画，常用的K-Means和高斯混合聚类。K-Means假定数据来源于同样本同方差，GMM聚类假定数据来源于多个高斯分布中。

**基于密度的聚类：**假定聚类结构能够通过样本分布的紧密程度来确定。常见的是DBSCAN，通过一组领域参数来描绘样本分布紧密程度。

**基于层次聚类：**在不同层次上对数据集进行划分，形成树装的聚类结构，AglomerativeNesting(AGNES)是一种常见的层次聚类算法。

#### 2.3.2k-medoids聚类算法

K-medoids选择数据集中有代表性的样本（而不是均值）来表示整体簇，即选取靠近中心点（medoid）的那个样本代表整个簇。

**k-medoids特点**

该算法是k-mean的改进，由于**k均值对于噪声和孤立点数据是敏感的**，为了修改这种敏感性，将k-means中的平均值作为参照点，修改为聚类中位置最中心的对象，即中心点。

与kmeans区别在于：

中心点选取：k-medoids中心点选取限制**在当前簇（分组）中所包含的数据点的集合中**。也就是与kmeans的差别就在于一个是数据样本的平均值，一个是样本数据的中位数，这两者区别在于平均数的取值是连续空间中的任意值，后者只能在样本给定的点中选择。

这样做的优势之处：

a，k-medoids不容易受到由于误差之类的原因产生的离群点影响

b，k均值要求数据只能落在二维的欧式空间中，但并不是所有的数据都可以满足这样的需求，特别是类别型的变量用欧氏距离是无法适用的。

k-means 和 k-medoids 之间的差异就类似于一个数据样本的均值 (mean) 和中位数 之间的差异：前者的取值范围可以是连续空间中的任意值，而后者只能在给样本给定的那些点里面选。那么，这样做的好处是什么呢？

 一个最直接的理由就是 k-means 对数据的要求太高了，它使用欧氏距离描述数据点之间的差异 (dissimilarity) ，从而可以直接通过求均值来计算中心点。这要求数据点处在一个欧氏空间之中。

然而并不是所有的数据都能满足这样的要求，对于数值类型的特征，比如身高，可以很自然地用这样的方式来处理，但是类别 (categorical) 类型的特征就不行了。 因为k-medoids，类中心点的选择，是选取该类中的一个样本，

存在的问题：

a，k-medoids也会陷入局部最优解。

b，k-medoids选取初值点需要枚举每个点，并要求他到所有点的距离之和，复杂度为O(N2)。Kmeans仅计算一个平均值，复杂度为O(N)。

#### 2.3.3层次聚类

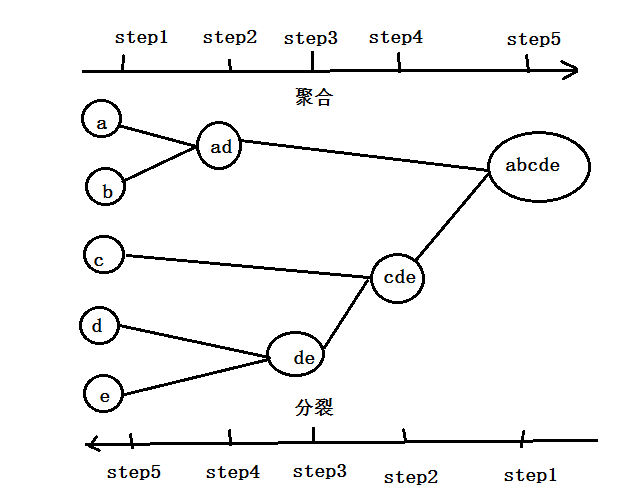
##### （1）基础知识

层次聚类也叫**连通聚类方法**，有两个基本方法：**自顶而下**和**自底而上**。自顶而下将所有样本看做是同一簇，然后进行分裂。自底而上将所有样本看做不同的簇，然后进行凝聚。这种聚类的中心思想是：离观测点较近的点相比离观测点较远的点更可能是一类，即相似的类簇和为新的类簇。

上述的两种方法（kmeans和kmedoids）均需要手动选取初始的类中心，层次聚类是一种树状的聚类算法，通过一种层次架构的方式，反复将数据进行分类和聚合，已形成一个层次序列的聚类问题。根据层次的分解如果形成，层次的基本方法可以分为：凝聚和分裂。

凝聚是有底向上构造树的方法，将每一个对象作为一个类，然后合并这些原子聚类为越来越大的聚类，直到所有的对象都在一个聚类中，或者满足某个终结条件（达到聚类数目）。

分裂是自顶向下方法，它的策略为将所有的对象置于一个聚类中，然后逐渐细分为越来越小的聚类，直到每个对象自成一聚类，或者达到某个结束条件。



**【补充】**

 对于层次聚类常用的距离度量方法主要有最大距离、最小距离、平均距离等。一个算法使用最大距离度量距离的时候，称为**最远邻聚类算法**。一个算法使用最小距离度量距离的时候，称为**邻聚类算法**。使用最小距离度量的聚合增长算法也称为最小生成树算法。当最近族的距离超过某个阀值时算法停止，称为**全连接算法**。**平均距离**是对最大最小距离度量的折中，可以有效克服噪音和奇异点的影响。

下面是代表性的算法。

**名字 特点**

CURE 采用抽样技术先对数据集D随机抽取样本，再采用分区技术对样本进行分区，然后对每个分区局部聚类，最后对局部聚类进行全局聚类

ROCK 也采用了随机抽样技术，该算法在计算两个对象的相似度时，同时考虑了周围对象的影响

CHEMALOEN（变色龙算法） 首先由数据集构造成一个K-最近邻图Gk ,再通过一个图的划分算法将图Gk 划分成大量的子图,每个子图代表一个初始子簇,最后用一个凝聚的层次聚类算法反复合并子簇，找到真正的结果簇

SBAC SBAC算法则在计算对象间相似度时，考虑了属性特征对于体现对象本质的重要程度，对于更能体现对象本质的属性赋予较高的权值

BIRCH BIRCH算法利用树结构对数据集进行处理，叶结点存储一个聚类，用中心和半径表示，顺序处理每一个对象，并把它划分到距离最近的结点，该算法也可以作为其他聚类算法的预处理过程

BUBBLE BUBBLE算法则把BIRCH算法的中心和半径概念推广到普通的距离空间

BUBBLE-FM BUBBLE-FM算法通过减少距离的计算次数，提高了BUBBLE算法的效率

**如：BIRCH算法描述**

* BIRCH算法考虑I/O开销。
* 一次扫描能够产生一个基本聚类，多次扫描能够改善聚类结果。
* 它是一个增量的聚类方法，对于数据的聚类决策是基于已经处理过的数据点，而不是全部样本空间，因此能够提高计算速度。
* 需要使用半径或者直接控制聚类的边界。

##### （2）层次聚类算法原理

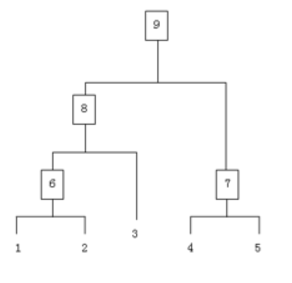
假设有N个待聚类的样本，对于层次聚类来说，步骤：

 1、（初始化）把每个样本归为一类，计算每两个类之间的距离，也就是样本与样本之间的相似度；

 2、寻找各个类之间最近的两个类，把他们归为一类（这样类的总数就少了一个）；

 3、重新计算新生成的这个类与各个旧类之间的相似度；

 4、重复2和3直到所有样本点都归为一类，结束



整个聚类过程其实是建立了一棵树，在建立的过程中，可以通过在第二步上设置一个阈值，**当最近的两个类的距离大于这个阈值，则认为迭代可以终止**。另外关键的一步就是第三步，如何判断两个类之间的相似度有不少种方法。**这里介绍三种相似度计算方法**：

①**SingleLinkage-最小距离**

又叫做 nearest-neighbor ，就是取两个类中距离最近的两个样本的距离作为这两个集合的距离，也就是说，**最近两个样本之间的距离越小，这两个类之间的相似度就越大**。容易造成一种叫做 Chaining 的效果，两个 cluster 明明从“大局”上离得比较远，但是由于其中个别的点距离比较近就被合并了，并且这样合并之后 Chaining 效应会进一步扩大，最后会得到比较松散的 cluster 。

②**CompleteLinkage-最大距离**

这个则完全是 Single Linkage 的反面极端，**取两个集合中距离最远的两个点的距离作为两个集合的距离**。其效果也是刚好相反的，限制非常大，两个 cluster 即使已经很接近了，但是只要有不配合的点存在，就顽固到底，老死不相合并，也是不太好的办法。这两种相似度的定义方法的共同问题就是指考虑了某个有特点的数据，而没有考虑类内数据的整体特点。

③**Average-linkage--平均距离**

这种方法就是把两个集合中的点两两的距离全部放在一起求一个平均值，相对也能得到合适一点的结果。

④**两距离的中值**

Average-linkage的一个变种就是取两两距离的中值，与取均值相比更加能够解除个别偏离样本对结果的干扰。

##### （3）算法总结：

基于原型的聚类---层次聚类

优势1：它能够使得我们绘制出树状图（基于二叉层次聚类的可视化）

优势2：不需要事先指定簇数量

层次聚类中有两种方法：**凝聚层次聚类和分裂层次聚类**

在分裂层次聚类中，首先把所有样本看作是在同一个簇中，然后迭代划分为更小的簇，知道每个簇中只包含一个样本

凝聚层次中，最初将每个样本都看做是一个簇，重复地将最近的一对簇进行合并，直到所有样本都在一个簇中心为止

凝聚层次聚类中，判断簇间距离的两个标准方法是单链接和全连接

* + 单链接：计算每一对簇中最相似两个样本的距离，并合并距离最近的两个样本所属簇
  + 全连接：通过比较找到分布于两个簇中最不相似的样本（距离最远的样本），进而完成簇的合并

# 这里着重掌握基于全连接的凝聚层次聚类

步骤如下：

1）计算得到所有样本间的距离矩阵

2）将每个数据点看作是一个单独的簇

3）基于最不相似（距离最远）样本的距离，合并两个最接近的簇

4）更新相似矩阵（样本间相似矩阵）

5）重复2到4，直到所有样本都合并到一个簇为止

#### 2.3.4基于密度的聚类

指导思想：只要样本的密度大于一个设定的阈值，则将该样本添加到最近的簇中。

##### （1）基础算法

聚类除了上述kmeans的聚类属于**基于原型的聚类**，kmedoids是**基于划分的聚类**，另外还有层次聚类和基于密度的聚类。基于原型意味着每个簇都对应一个模型，它可以是一些具有连续型特征的相似点的中心点（平均值），或者是相似点的众数---出现频率比较高的点。

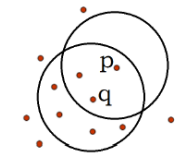
**基于密度的聚类**是基于距离的聚类的变型。纯粹的基于距离的聚类只能发现球状的分组，而在发现任意形状的的簇上遇到了困难。主要思想是：**只要邻近区域的密度（对象或数据点的数目）超出了一个阈值，就继续聚类。**也就是说，对给定类中的每个数据点，在一个给定范围的区域中必须至少包含某个数据的点。这样的方法可以用来过滤“噪声”孤立点的数据，发现任意形状的簇。

DBSCAN的几个概念：

**对象的E领域：**给定对象在半径E内的区域。

**核心对象：**对于给定的数目m，如果一个对象的E领域至少包含m个对象，则称该对象为核心对象。

**直接密度可达：**给定一个对象集合D，如果p是在q的E领域内，而q是一个核心对象，我们说对象p从对象q出发是直接密度可达的。如下图：

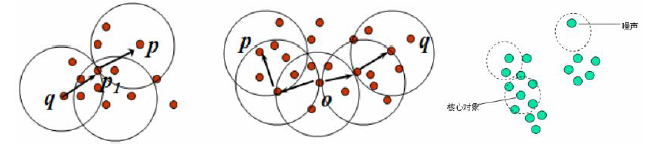


**密度可达：**如果存在一个对象链p1p2....pn,p1=q,pn=p，p(i+1)是从pi关于E和m直接密度可达的，则对象p是从对象q关于E和m密度可达的。如下图1。

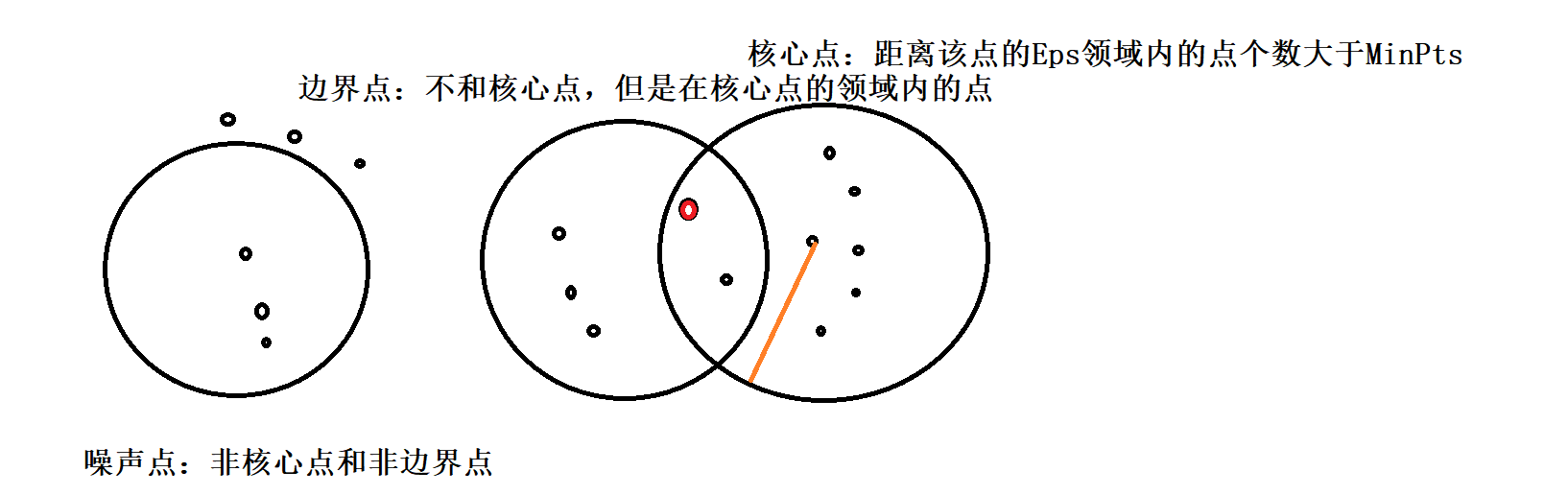
**密度相连：**如果集合D中存在一个对象o，使得对象p和q是从o关于E和m密度可达的，那么对象p和q是关于E和m密度相连的。如下图2.

簇：一个基于密度的簇是最大的密度项链对象的集合。

噪声：不包含在任何簇中的对象成为噪声。



对比理解



##### （2）DBSCAN算法

算法思想：

1. 从数据中抽出一个未处理的节点。
2. 统计该节点周围半径E范围内的节点数量，如果小于MinPts，则该点为**噪声点**。回到步骤1继续处理。
3. 如果抽出的节点其最小半径E范围内节点个数大于MinPts，则该点称为一个新簇的**核心点**，将所有从该点密度可达的点加入到当前新簇中，如果该点**密度可达**节点中已经有已经生成的簇的核心，则将两个簇合并为新的簇，新簇核心点为原来合并簇的核心点的集合。
4. 重复算法过程直到所有节点被处理。

**算法流程：（重点记忆）**

* 如果一个点p的E领域内包含多于m个对象，则创建一个p作为核心对象的新簇；
* 寻找并合并核心对象直接密度可达的对象
* 没有新点可以更新簇时，算法结束。

**由上述算法可得：**

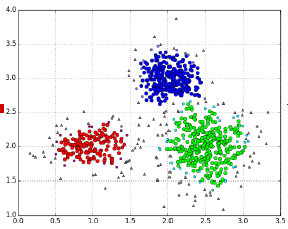
* 每个簇至少包含一个核心对象；
* 非核心对象可以是簇的一部分，构成了簇的边缘。
* 包含过少对象的簇被认为是噪声。

DBSCAN算法最大的特点是**抗噪声干扰，可以发现任意形状的特性**。

DBSCAN将簇定义为密度相连的点的最大集合，能够把足够高密度的趋于划分为簇，并可在有“噪声”的数据中发现任意形状的聚类。

**如下：给定m=5，也就是一个点领域r内至少5个点才会被认为是核心点。**三角形点代表噪声点，最外侧的是簇的边缘，每个簇至少有一个核心对象。

当m值取的越大，就会有越多的点会由核心点变为非核心点，甚至为噪声点。



##### （3）算法总结

基于密度的聚类-DBSCAN

1. 将多有的点定义为核心点、边界点和噪声点
2. 删除噪声点
3. 为距离控制在Eps之内的所有的核心点连在一起(密度可达)
4. 每组连通的核心点形成新的簇
5. 将每个边界点指派到一个与之关联的核心点的簇中

DBSCAN的算法与输入的参数MinPts数量有关系的，DBSCAN算法需要不停调整差参数的影响

DBSCAN算法的优点是能够发现任意形状的簇，不需要指定簇的个数，但是缺点是高纬度空间中，密度的定义是比较困难的

适用场景：基于地理位置信息的聚类中使用效果很好。

#### 2.3.5高斯混合聚类GMM

GMM（Gauss Mixture Model）是一种组合模型，它用多个高斯密度函数的线性加权和表示分布的概率。其中每个函数称之为一个组件。各加权系数通过迭代式期望最大方法(EM-expectation-maximization)或后验概率最大法(MAP,maximum a posteriori estimation)在训练数据集上训练得出。

##### （1）Spark实现GMM

**SparkML使用EM算法实现GMM。**

**其参数如下：**

* **K：期望的类簇个数**
* **converagenceTol：收敛阈值，当相邻两次迭代的损失差小于该值的时候，认为模型已经收敛，训练完成。**
* **maxIerations:最大迭代次数，一致未收敛，最多迭代训练多少次。**
* **InitModel：可选参数，EM初始化的方法，若未指定，则会随机从数据创建一个点来训练。**

【补充】期望-最大化(Expectation-Maximization, EM) 方法步骤：

**1.根据给定的K值，初始化K个多元高斯分布以及其权重；**

**2.根据贝叶斯定理，估计每个样本由每个成分生成的后验概率；(EM方法中的E步)**

**3.根据均值，协方差的定义以及2步求出的后验概率，更新均值向量、协方差矩阵和权重；（EM方法的M步）**

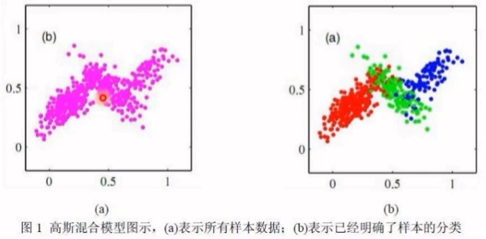
**重复2~3步，直到似然函数增加值已小于收敛阈值，或达到最大迭代次数**

##### （2）GMM原理详细解析

高斯混合聚类就是通过概率模型表示聚类原型。和K均值都属于**基于原型的聚**类。（假设聚类结构能通过一组原型或模型刻画）

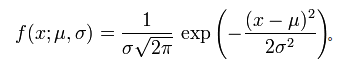
高斯混合模型（Gauss Mixture Model）

* 是一个概率模型
* 它假定所有的数据点**都是从若干个参数位置的单高斯分布的混合分布中产生的**
* 在上述假定下，GMM希望能从原始数据中学习到混合分布中每个单高斯分布的参数以及他们之间的混合权重比例，从而得到原始数据的生成式模型。



有几种不同的方法用来说明一个随机变量。最直观的方法是[概率密度函数](http://zh.wikipedia.org/wiki/%E6%A6%82%E7%8E%87%E5%AF%86%E5%BA%A6%E5%87%BD%E6%95%B0" \o "概率密度函数" \t "http://blog.csdn.net/rns521/article/details/_blank)，这种方法能够表示随机变量每个取值有多大的可能性。

正态分布的概率密度函数



正态分布的[概率密度函数](http://zh.wikipedia.org/wiki/%E6%A6%82%E7%8E%87%E5%AF%86%E5%BA%A6%E5%87%BD%E6%95%B0" \o "概率密度函数" \t "http://blog.csdn.net/rns521/article/details/_blank)均值为μ ,[方差](http://zh.wikipedia.org/wiki/%E6%96%B9%E5%B7%AE" \o "方差" \t "http://blog.csdn.net/rns521/article/details/_blank)为σ2 (或[标准差](http://zh.wikipedia.org/wiki/%E6%A8%99%E6%BA%96%E5%B7%AE" \o "标准差" \t "http://blog.csdn.net/rns521/article/details/_blank)σ)是[高斯函数](http://zh.wikipedia.org/wiki/%E9%AB%98%E6%96%AF%E5%87%BD%E6%95%B8" \o "高斯函数" \t "http://blog.csdn.net/rns521/article/details/_blank)的一个实例：

**高斯混合模型定义如下：**





#### （3）总结

因此，高斯混合聚类的原理为：如果样本x最有可能是Z=k产生的，则可将该样本划归到簇Ck中，即**通过最大后验概率确定样本所属的聚类。**



当搭建好了高斯混合模型框架之后，最后留下一个问题就是如何求解高斯混合分布的参数，由于涉及隐变量Z（意思是来源于第几个高斯分布），**需要采用EM算法求解**。

注意：Python的sklearn实现有两类，均在sklearn.mixture下，其中sklearn.mixture.BayesianGaussianMixture使用的是变分推理算法估计GMM中的参数μ和σ，sklearn.mixture.GaussianMixture使用EM最大化算法求解参数。

**SparkMl使用EM算法实现GMM。**

**总结**

若已知高斯混合分布，则**高斯混合聚类的原理是**：***如果样本x(i)最有可能是Z=k产生的则可将该样本划归到簇C(k)。即通过最大的后验概率确定样本所属聚类***。

注：聚类的ARI指标：ARI是Adjusted Rand Index简称，该指标取值介于[0,1]之间，这些值越靠近1说明聚类性能越好。Sklearn实现了ARI指标计算。

#### 2.3.6总结及聚类算法选择

##### （1）总结

在实际的应用中，对于给定的数据集，往往不太确定使用哪种算法是最为合适的，特别是面对难以或者无法进行可视化处理的高维数据集。另外，一个好的聚类算法并不是仅仅依赖于算法及其超参数的调整，相反，选择合适的距离度量标准和专业领域知识在实验设定的应用可能很有用。

##### （2）如何选择各种聚类方法？

如果提前可判定数据属于高斯分布，则选择K均值聚类。

如果样本数据不多，使用k均值效果不好，或者数据分布不属于高斯分布，就可以用密度聚类，密度聚类可以发现任意形状的簇。

如果样本本身具有很好的层次性，就可以选择层次聚类。

也可以在层次聚类中加入密度聚类来实现聚类。

|  |  |  |
| --- | --- | --- |
| 模型 | 关键参数 | 使用场景 |
| KMeans | 簇的数量 | 通用聚类的方法，用于均匀的簇大小，簇的数量不多的情况 |
| DBSCAN算法(发现任意聚类的形状算法) | c和MinPts | 用于不均匀的簇大小以及非平均的集合结构 |
| AgglomerativeClustering算法 | 簇的数量，链接类型 | 用于簇的数量较多，有链接约束情形 |
| GMM算法 | 一些均值，方差，Z(来源第几个高斯分布) | 用于平坦的集合结构，对密度估计很合适。 |

### 2.4SparkMllib聚类实战

#### 2.4.1SparkMllib-BisectingKMean原理及实战

BisectingKMeans是一种分层聚类使用一个分裂的(或“自上而下”)的方法：所有观测开始在一个集群,将递归地执行沿着层次结构。平分k - means通常可以比常规的k - means要快得多,但是它通常会产生不同的集群。

BisectingKMeans 被实现为一个Estomtor 并生成一个 BisectingKMeansModel 基本模型。

import org.apache.spark.ml.clustering.BisectingKMeans

// Loads data.

val dataset = spark.read.format("libsvm").load("data/mllib/sample\_kmeans\_data.txt")

// Trains a bisecting k-means model.

val bkm = new BisectingKMeans().setK(2).setSeed(1)

val model = bkm.fit(dataset)

// Evaluate clustering.

val cost = model.computeCost(dataset)

println(s"Within Set Sum of Squared Errors = $cost")

// Shows the result.

println("Cluster Centers: ")

val centers = model.clusterCenters

centers.foreach(println)

#### 2.4.2SparkMllib-GMM原理及实战

GMM（Gauss Mixture Model）是一种组合模型，他用多个高斯密度函数的线性加权和表示分布的概率。其中每个函数称之为一个组件。各加权系数通过迭代式期望最大方法(EM-expectation-maximization)或后验概率最大法(MAP,maximum a posteriori estimation)在训练数据集上训练得出。

**SparkML使用EM算法实现GMM。**

**其参数如下：**

**K：期望的类簇个数**

**converagenceTol：收敛阈值，当相邻两次迭代的损失差小于该值的时候，认为模型已经收敛，训练完成。**

**maxIerations:最大迭代次数，一致未收敛，最多迭代训练多少次。**

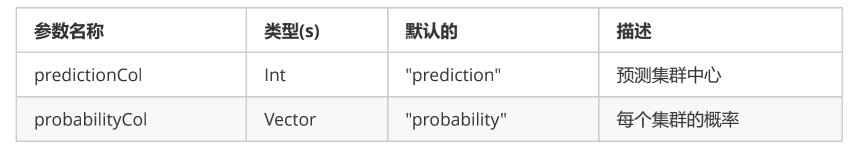
**InitModel：可选参数，EM初始化的方法，若未指定，则会随机从数据创建一个点来训练。**

高斯混合模型表示复合分布，其中点从k个高斯子分布中的一个绘出，每个具有其自使用期望最大化算法来给出给定一组样本的最大似然模型。GaussianMixture作为估计器实现，并生成GaussianMixtureModel 作为基本模型。

输入列：



输出列：



代码：

import org.apache.spark.ml.clustering.GaussianMixture

// Loads data

val dataset = spark.read.format("libsvm").load("data/mllib/sample\_kmeans\_data.txt")

// Trains Gaussian Mixture Model

val gmm = new GaussianMixture()

.setK(2)

val model = gmm.fit(dataset)

// output parameters of mixture model model

for (i <- 0 until model.getK) {

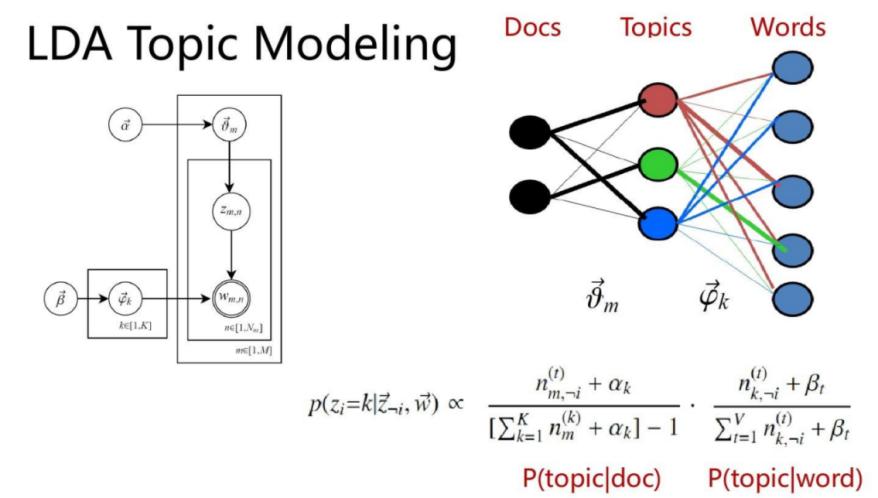
println(s"Gaussian $i:\nweight=${model.weights(i)}\n" +

s"mu=${model.gaussians(i).mean}\nsigma=\n${model.gaussians(i).cov}\n")

}

#### 2.4.3SparkMllib-LDA原理及实战

首先，了解LDA算法思想和用途：



* LDA建模的过程

     LDA模型中一篇文档生成的方式：如上图

       (1)从狄利克雷分布α中取样生成文档i的主题分布θi

       (2)从主题的多项式分布θi中取样生成文档i第j个词的主题zi,j

       (3)从狄利克雷分布β中取样生成主题zi,j的词语分布ϕ(zi,j)

       (4)从词语的多项式分布ϕ(zi,j)中采样最终生成词语wi,j

* LDA建模算法

      接下来要去考虑，怎么去计算这两个矩阵θi和ϕ(zi,j)，怎么去优化的问题？。Spark采用的两种优化算法：

      （1）EMLDAOptimizer 通过在likelihood函数上计算最大期望EM，提供较全面的结果。

      （2）OnlineLDAOptimizer 通过在小批量数据上迭代采样实现online变分推断，比较节省内存。在线变分预测是一种训练LDA模型的技术，它以小批次增量式地处理数据。由于每次处理一小批数据，我们可以轻易地将其扩展应用到大数据集上。MLlib按照 Hoffman论文里最初提出的算法实现了一种在线变分学习算法。

* + SPARK中可选参数

    （1）K：主题数量（或者说聚簇中心数量）

    （2）maxIterations：EM算法的最大迭代次数，设置足够大的迭代次数非常重要，前期的迭代返回一些无用的（极其相似的）话题，但是继续迭代多次后结果明显改善。我们注意到这对EM算法尤其有效。，至少需要设置20次的迭代，50-100次是更合理的设置，取决于你的数据集。

    （3）docConcentration（Dirichlet分布的参数α)：文档在主题上分布的先验参数（超参数α)。当前必须大于1，值越大，推断出的分布越平滑。默认为-1，自动设置。

   （4）topicConcentration（Dirichlet分布的参数β)：主题在单词上的先验分布参数。当前必须大于1，值越大，推断出的分布越平滑。默认为-1，自动设置。

   （5）checkpointInterval：检查点间隔。maxIterations很大的时候，检查点可以帮助减少shuffle文件大小并且可以帮助故障恢复。

LDA实现为一个支持EMLDAOptimizer和OnlineLDAOptimizer的估计器，并生成一个LDAModel作为基本模型。 如果需要，可以将EMLDAOptimizer生成的LDAModel转换为DistributedLDAModel。

代码：

import org.apache.spark.ml.clustering.LDA

// Loads data.

val dataset = spark.read.format("libsvm")

.load("data/mllib/sample\_lda\_libsvm\_data.txt")

// Trains a LDA model.

// val lda = new LDA()

// .setK(10)//k: 主题数，或者聚类中心数 >1

// .setMaxIter(10)// MaxIterations：最大迭代次数 >= 0

// .setCheckpointInterval(1) //迭代计算时检查点的间隔 set checkpoint interval (>= 1) or disable checkpoint (-1)

// .setDocConcentration(0.1) //文章分布的超参数(Dirichlet分布的参数)，必需>1.0

// .setTopicConcentration(0.1)//主题分布的超参数(Dirichlet分布的参数)，必需>1.0

// .setOptimizer("online") //默认 online 优化计算方法，目前支持"em", "online"

val lda = new LDA().setK(10).setMaxIter(10)

val model = lda.fit(dataset)

val ll = model.logLikelihood(dataset)

val lp = model.logPerplexity(dataset)

println(s"The lower bound on the log likelihood of the entire corpus: $ll")

println(s"The upper bound on perplexity: $lp")

// Describe topics.

val topics = model.describeTopics(3)

println("The topics described by their top-weighted terms:")

topics.show(false)

// Shows the result.

val transformed = model.transform(dataset)

transformed.show(false)

代码注释讲解：

import org.apache.spark.sql.SparkSession

import org.apache.log4j.{Level, Logger}

import org.apache.spark.ml.clustering.LDA

object myClusters {

def main(args:Array[String]){

//屏蔽日志

Logger.getLogger("org.apache.spark").setLevel(Level.ERROR)

Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

val warehouseLocation = "/Java/Spark/spark-warehouse"

val spark=SparkSession

.builder()

.appName("myClusters")

.master("local[4]")

.config("spark.sql.warehouse.dir",warehouseLocation)

.getOrCreate();

val dataset\_lpa=spark.read.format("libsvm")

.load("/spark-2.0.0-bin-hadoop2.6/data/mllib/sample\_lda\_libsvm\_data.txt")

//------------------------------------1 模型训练-----------------------------------------

/\*\*

\* k: 主题数，或者聚类中心数

\* DocConcentration：文章分布的超参数(Dirichlet分布的参数)，EM必需>1.0，值越大，推断出的分布越平滑

\* TopicConcentration：主题分布的超参数(Dirichlet分布的参数)，EM必需>1.0，值越大，推断出的分布越平滑

\* MaxIterations：迭代次数，需充分迭代，至少20次以上

\* setSeed：随机种子

\* CheckpointInterval：迭代计算时检查点的间隔

\* Optimizer：优化计算方法，目前支持"em", "online" ，em方法更占内存，迭代次数多内存可能不够会抛出stack异常

\*/

val lda=new LDA()

.setK(3)

.setTopicConcentration(3)

.setDocConcentration(3)

.setOptimizer("online")

.setCheckpointInterval(10)

.setMaxIter(100)

val model=lda.fit(dataset\_lpa)

/\*\*生成的model不仅存储了推断的主题，还包括模型的评价方法。\*/

//---------------------------------2 模型评价-------------------------------------

//模型的评价指标：ogLikelihood，logPerplexity

//（1）根据训练集的模型分布计算的log likelihood，越大越好。

val ll = model.logLikelihood(dataset\_lpa)

//（2）Perplexity评估，越小越好

val lp = model.logPerplexity(dataset\_lpa)

println(s"The lower bound on the log likelihood of the entire corpus: $ll")

println(s"The upper bound bound on perplexity: $lp")

//---------------------------------3 模型及描述------------------------------

//模型通过describeTopics、topicsMatrix来描述

//（1）描述各个主题最终的前maxTermsPerTopic个词语（最重要的词向量）及其权重

val topics=model.describeTopics(maxTermsPerTopic=2)

println("The topics described by their top-weighted terms:")

topics.show(false)

/\*\*主题 主题包含最重要的词语序号 各词语的权重

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

|topic|termIndices |termWeights |

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

|0 |[5, 4, 0, 1] |[0.21169509638828377, 0.19142090510443274]|

|1 |[5, 6, 1, 2] |[0.12521929515791688, 0.10175547561034966]|

|2 |[3, 10, 6, 9]|[0.19885345685860667, 0.18794498802657686]|

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

\*/

//（2） topicsMatrix: 主题-词分布，相当于phi。

val topicsMat=model.topicsMatrix

println("topicsMatrix")

println(topicsMat.toString())

/\*\*topicsMatrix

12.992380082908886 0.5654447550856024 16.438154549631257

10.552480038361052 0.6367807085306598 19.81281695100224

2.204054885551135 0.597153999004713 6.979803589429554

\*

\*/

//-----------------------------------4 对语料的主题进行聚类---------------------

val topicsProb=model.transform(dataset\_lpa)

topicsProb.select("label", "topicDistribution")show(false)

/\*\* label是文档序号 文档中各主题的权重

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

|label|topicDistribution |

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

|0.0 |[0.523730754859981,0.006564444943344147,0.46970480019667477] |

|1.0 |[0.7825074858166653,0.011001204994496623,0.206491309188838] |

|2.0 |[0.2085069748527087,0.005698459472719417,0.785794565674572] |

\*/

}

}

# 3.客户价值模型-RFM

## 3.1RFM模型引入

比如电商网站要做一次营销活动，需要针对不同价值的客户群体进行分群，对于高价值的用户推荐手表，珠宝等高端商品，对于低价值用户推荐打折促销的廉价商品，当然还有以下这些问题都是需要考虑的：

1.谁是我的最佳客户？

2.谁即将要成为流失客户？

3.谁将有潜力成为有价值的客户

4.哪些客户能够留存？

5.哪些客户会对你目前对活动有所反应？

那么最终的问题是如何对客户进行分群，即如何建立客户的价值模型呢？

在传统企业和电商众多的客户细分模型中，RFM模型是被广泛提到和使用的。

RFM模型是衡量当前用户价值和客户潜在价值的重要工具和手段。

RFM是Rencency（最近一次消费），Frequency（消费频率）、Monetary（消费金额），三个指标首字母组合，如图所示：



一般情况下RFM模型可以说明下列几个事实：

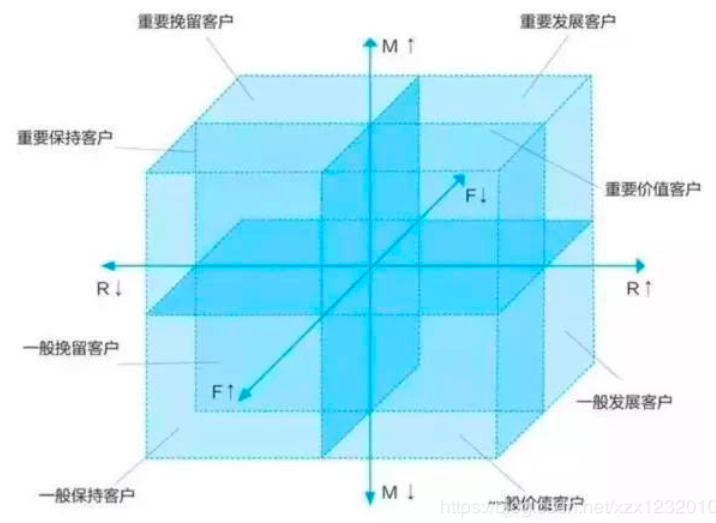
1.最近购买的时间越近，用户对产品促销互动越大

2.客户购买的频率越高，客户就品牌的满意度就越大

3.货币价值将高消费客户和低消费客户区分开来

如图所示，根据RFM模型，就可以统计在某一段时间内，用户最近的消费间隔，消费次数和消费金额，再根据使用k-means算法对用户进行聚类分群

注意一点，不仅仅可以局限于这三个数据字段，还可以根据业务需求，加入其他字段，进行调整模型。



问题:我们可以根据RFM模型计算出所有用的RFM值形成一个二维表:

userid, R值, F值, M值

1 2019-11-01 5 10000

2 2019-10-01 4 800

对于以上数据的量纲不一致(单位不统一),所以要对数据进行归一化

如何归一化?---需要自定义归一化的规则!即建立一个评分标准?

如何建立评分标准?----根据运营/产品的经验,做一个标准

R: 1-3天=5分，4-6天=4分，7-9天=3分，10-15天=2分，大于16天=1分

F: ≥200=5分，150-199=4分，100-149=3分，50-99=2分，1-49=1分

M: ≥20w=5分，10-19w=4分，5-9w=3分，1-4w=2分，<1w=1分

根据上面的打分规则就可以对数据进行自定义的归一化,得到如下类似结果:

userid, R值, F值, M值

1 5 1 2

2 1 1 1

那么现在数据已经归一化了,如何对数据进行分类?--肯定不能简单的将数据直接丢到三维坐标系,因为坐标系的原点不好确定,且三维坐标系只能分为8类

所以应该使用算法进行分类(聚类)--让算法自动学习用户之间的相似度,然后相似度高的用户,自动聚成一类,最后完成聚类的划分

那么到这里我们的目标就很清晰了!

首先对所有用户的最近一次消费时间/总共消费次数/总共消费金额进行统计

再进行归一化(运营/产品提供的打分规则)

再使用算法进行聚类(K-Means)

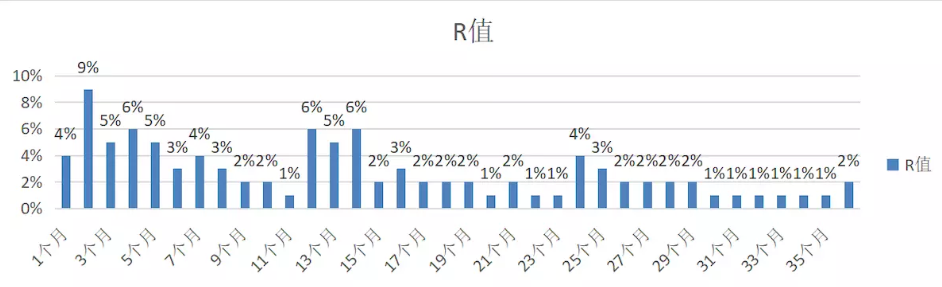
根据聚类结果给用户打Tag(标签)

### 3.1.1RFM详解

#### （1）R值：最近一次消费（Recency）

消费指的是客户在店铺消费最近一次和上一次的时间间隔，理论上R值越小的客户是价值越高的客户，即对店铺的回购几次最有可能产生回应。目前网购便利，顾客已经有了更多的购买选择和更低的购买成本，去除地域的限制因素，客户非常容易流失，因此CRM操盘手想要提高回购率和留存率，需要时刻警惕R值。

如下图，某零食网店用户最近一次消费R值分布图：

1、客户R值呈规律性的“波浪形”分布，时间越长，波浪越小；

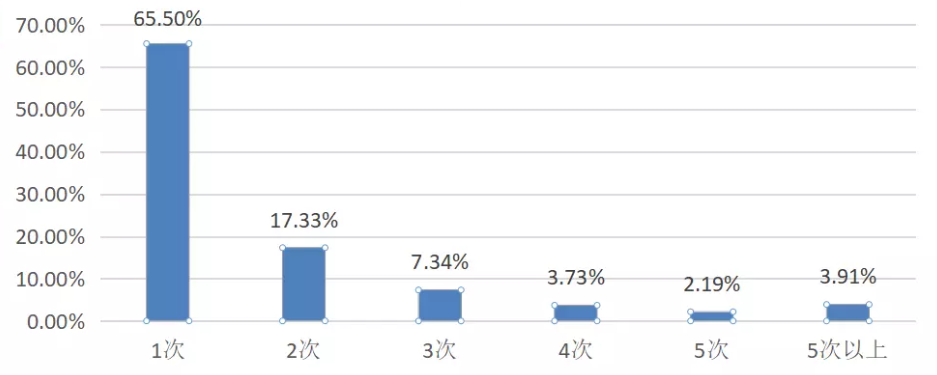
2、最近一年内用户占比50%（真的很巧）；

数据分析：这个数据根据向行业内专业人员请教，已经是比较理想了的。说明每引入2个客户，就有一位用户在持续购买。说明店铺复购做的比较好。

#### （2）F值：消费频率（Frequency）

消费频率是客户在固定时间内的购买次数（一般是1年）。但是如果实操中实际店铺由于受品类宽度的原因，比如卖3C产品，耐用品等即使是忠实粉丝用户也很难在1年内购买多次。所以，有些店铺在运营RFM模型时，会把F值的时间范围去掉，替换成累计购买次数。

如下图，某零食网店用户购买频次图（如1个客户在1天内购买多笔订单，则自动合并为1笔订单）：



1、购买1次（新客户）占比为65.5%，产生重复购买（老客户）的占比为34.4%；

2、购买3次及以上（成熟客户）的占比为17%，购买5次及以上（忠实客户）的占比为6%。

数据分析：影响复购的核心因素是商品，因此复购不适合做跨类目比较。比如食品类目和美妆类目：食品是属于“半标品”，产品的标品化程度越高，客户背叛的难度就越小，越难形成忠实用户；但是相对美妆，食品又属于易耗品，消耗周期短，购买频率高，相对容易产生重复购买，因此跨类目复购并不具有可比性。

#### （3）M值：消费金额（Monetary）

M值是RFM模型中相对于R值和F值最难使用，但最具有价值的指标。大家熟知的“二八定律”（又名“帕雷托法则”）曾作出过这样的解释：公司80%的收入来自于20%的用户。

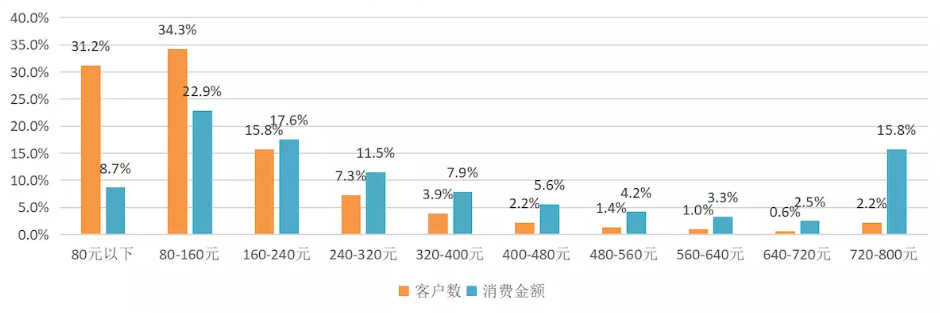
这个数据我在自己所从事的公司总都得到过验证！可能有些店铺不会那么精确，一般也会在30%客户贡献70%收入，或者40%贡献60%收入。

理论上M值和F值是一样的，都带有时间范围，指的是一段时间（通常是1年）内的消费金额，在工作中我认为对于一般店铺的类目而言，产品的价格带都是比较单一的，比如：同一品牌美妆类，价格浮动范围基本在某个特定消费群的可接受范围内，加上单一品类购买频次不高，所以对于一般店铺而言，M值对客户细分的作用相对较弱。

所以我认为用店铺的累计购买金额和平均客单价替代传统的M值能更好的体现客户消费金额的差异。

教大家一个特别简单的累积金额划分方法：将1/2的客单价作为累积消费金额的分段，比如客单价是300元，则按照150元进行累计消费金额分段，得出十个分段。

现以国内某知名化妆品店铺举例，店铺平均客单为160元，因此以80元作为间隔将累积消费金额分段，从表中可以很明显发现，累计消费160元以下用户占比为65.5%（近2/3），贡献的店铺收入比例只占31.6%（近1/3），具体如下：



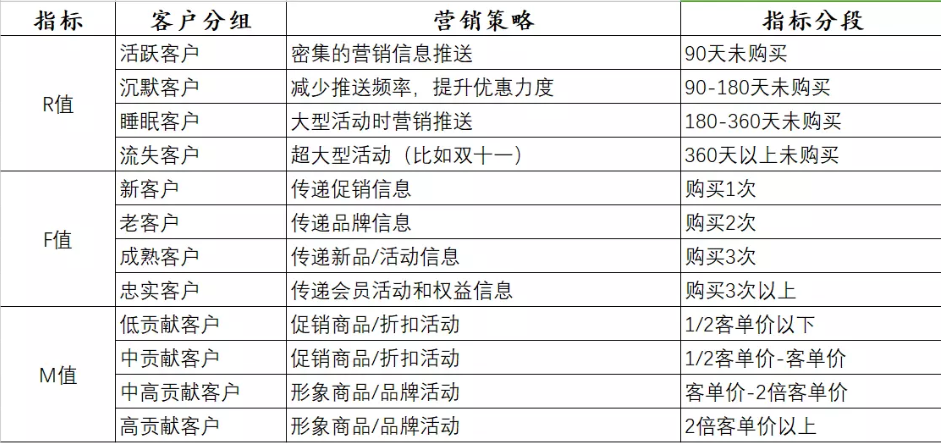
### 3.1.2基于RFM模型的实践应用

主要有两种方法来分析RFM模型的结果：用基于RFM模型的划分标准来进行客户细分，用基于RFM模型的客户评分来进行客户细分。

#### （1）基于RFM模型进行客户细分

CRM实操时可以选择RFM模型中的1-3个指标进行客户细分，如下表所示。切记细分指标需要在自己可操控的合理范围内，并非越多越好，一旦用户细分群组过多，一来会给自己的营销方案执行带来较大的难度，而来可能会遗漏用户群或者对同个用户造成多次打扰。

最终选择多少个指标有两个参考标准：店铺的客户基数，店铺的商品和客户结构。



店铺的客户基数：在店铺客户一定的情况下选择的维度越多，细分出来每一组的用户越少。对于店铺基数不大（5万以下客户数）的店铺而言，选择1-2个维度进行细分即可。对于客户超过50万的大卖家而言可以选择2-3个指标。

店铺的商品和客户结构：如果在店铺的商品层次比较单一，客单价差异幅度不大的情况下，购买频次（F值）和消费金额（M值）高度相关的情况下，可以只选择比较容易操作的购买频次（F值）代替消费金额（M值）。对于刚刚开店还没形成客户粘性的店铺，则可以放弃购买频次（F值），直接用最后一次消费（R值）或者消费金额（M值）。

#### （2）通过RFM模型评分后输出目标用户

除了直接用RFM模型对用户进行分组之外，还有一种常见的方法是利用RFM模型的三个属性对客户进行打分，通过打分确定每个用户的质量，最终筛选出自己的目标用户。

RFM模型评分主要有三个部分：

1、确定RFM三个指标的分段和每个分段的分值；

2、计算每个客户RFM三个指标的得分；

3、计算每个客户的总得分，并且根据总得分筛选出优质的客户

比如，实操的过程中一般每个指标分为3-5段，其中R值可以根据开店以来的时间和产品的回购周期来判定，F值根据现有店铺的平均购买频次，M值可参考上文客单价的分段指标。

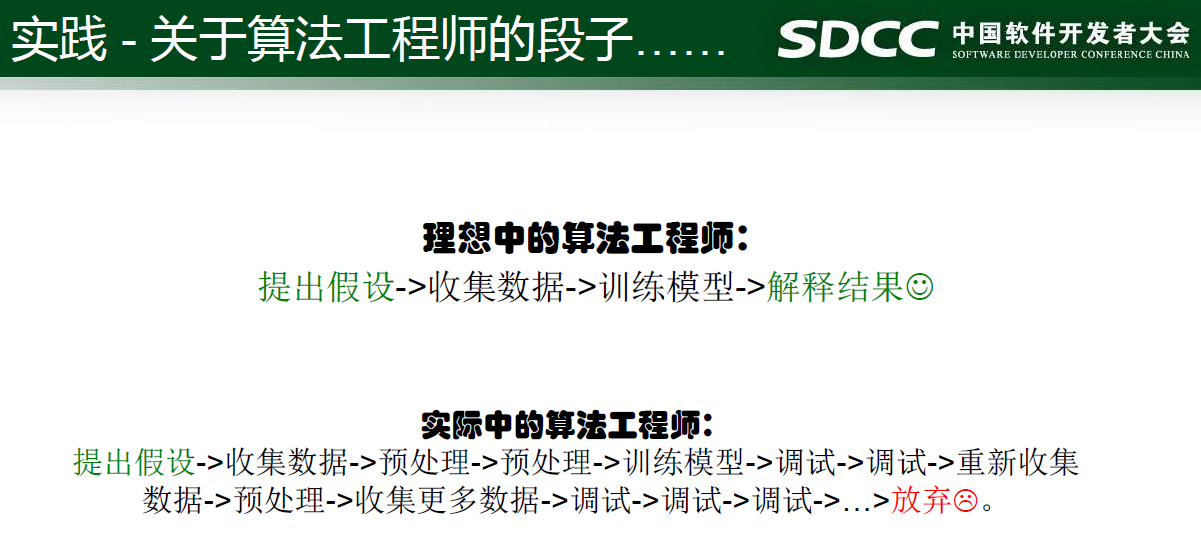
举个例子：



确认RFM的分段和对应分段的分值之后，就可以按照用户情况对应进行打分。

这个时候可能有人会对此产生质疑，我如何验证这个给予的分值就是合理的呢？一般使用经验值或用算法模型进行验证。

**小Tips：**

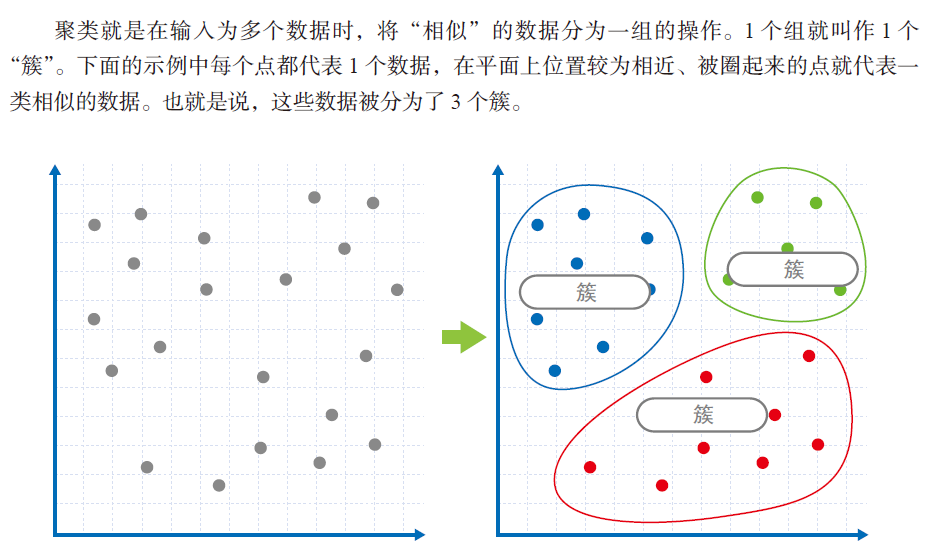


## 3.2KMeans聚类算法详解

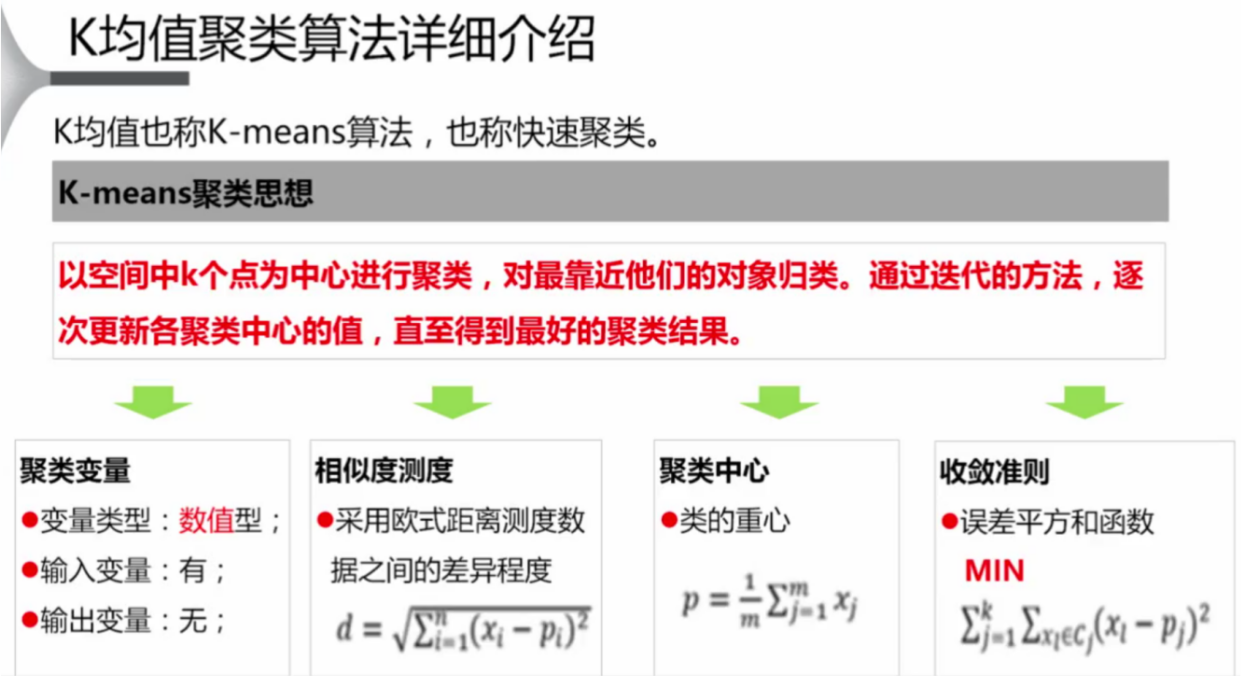
### 3.2.1算法原理

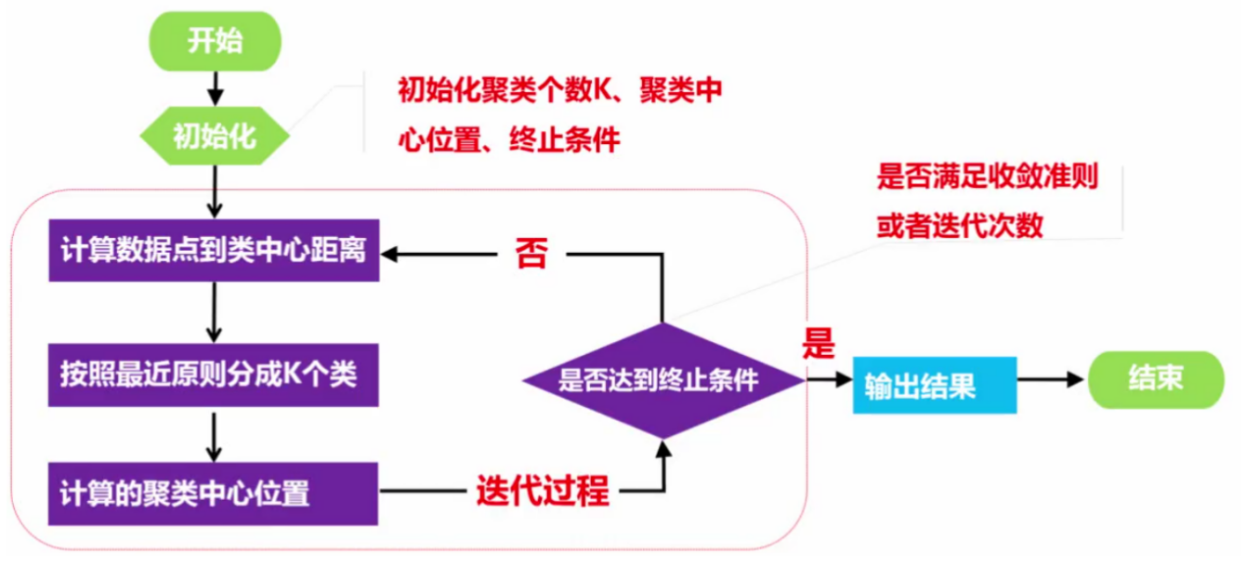
<http://shabal.in/visuals/kmeans/3.html>

<https://www.naftaliharris.com/blog/visualizing-k-means-clustering/>









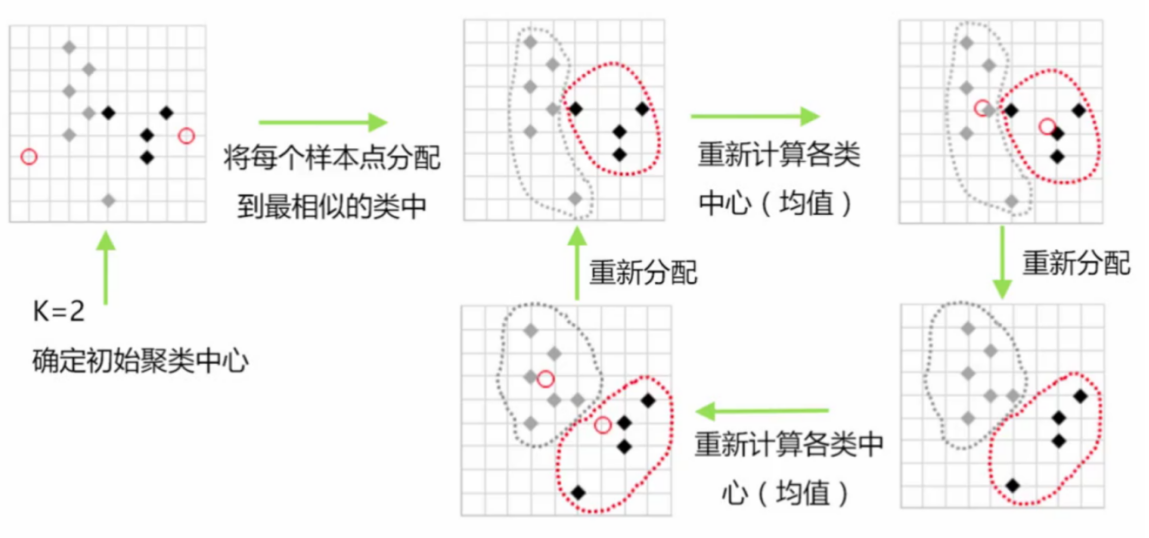
计算步骤

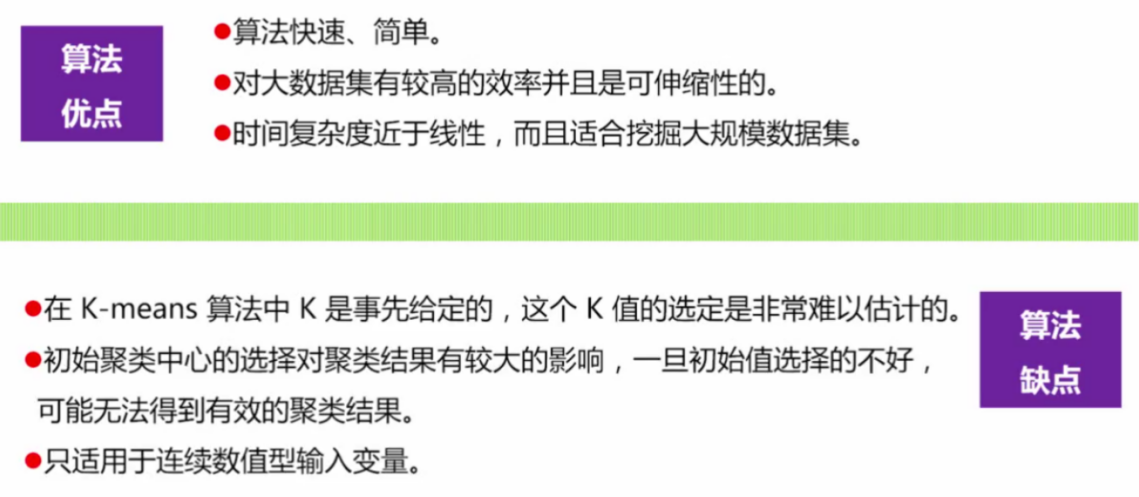
选择 K 个点作为初始聚类中心

计算其他的点到中心点的距离, 进行聚类, 使用欧式距离

重新计算每个聚类的中心点, 再次聚类

直到中心点不再变化, 或者达到迭代次数





### 3.2.2快速体验

#### （1）数据集

IRIS数据集由Fisher在1936年整理的一个经典数据集，在统计学习和机器学习领域都经常被用作示例。

数据集内包含 3 类共 150 条记录，每类各 50 个数据，每条数据包含4个特征，都是浮点数，单位为厘米

Sepal.Length（花萼长度）

Sepal.Width（花萼宽度）

Petal.Length（花瓣长度）

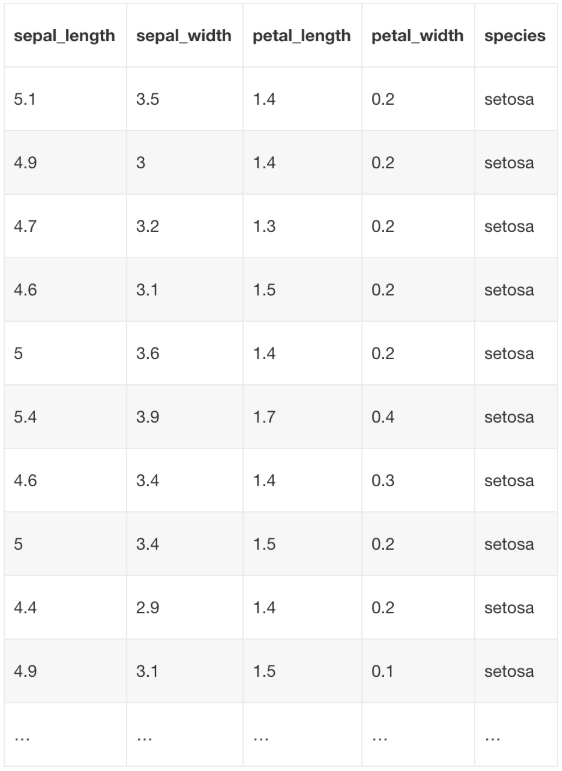
Petal.Width（花瓣宽度））

目标值为鸢尾花的分类:

Iris Setosa（山鸢尾）

Iris Versicolour（杂色鸢尾）

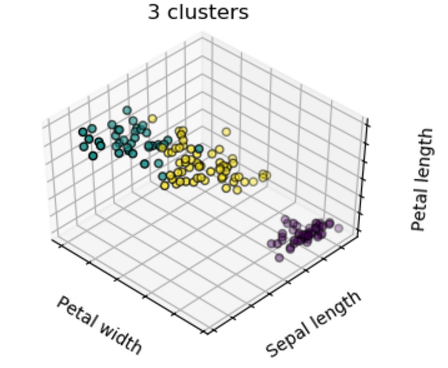
Iris Virginica（维吉尼亚鸢尾）





<https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html#sklearn.datasets.load_iris>

<https://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_iris.html#sphx-glr-auto-examples-cluster-plot-cluster-iris-py>



#### （2）代码演示

package cn.itcast.up.ml

import org.apache.spark.ml.clustering.{KMeans, KMeansModel}

import org.apache.spark.ml.feature.{MinMaxScaler, MinMaxScalerModel}

import org.apache.spark.sql.{DataFrame, SparkSession}

//使用KMeans聚类算法处理鸢尾花数据集

//https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load\_iris.html#sklearn.datasets.load\_iris

//libsvm数据格式:<label> <index1>:<value1> <index2>:<value2> ...

//数据归一化

//https://blog.csdn.net/m0\_37870649/article/details/79535339

//https://www.zhihu.com/question/28641663?sort=created

object IrisCluster {

def main(args: Array[String]): Unit = {

val spark = SparkSession.builder()

.appName("IrisDecisionTree")

.master("local[\*]")

.getOrCreate()

spark.sparkContext.setLogLevel("WARN")

import spark.implicits.\_

//1.数据读取

val irisDF: DataFrame = spark.read.format("libsvm").load("file:///D:\\data\\spark\\ml\\iris\_kmeans.txt")

irisDF.show(false)

irisDF.printSchema()

/\*

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

|label|features |

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

|1.0 |(4,[0,1,2,3],[5.1,3.5,1.4,0.2])|

|1.0 |(4,[0,1,2,3],[4.9,3.0,1.4,0.2])|

|1.0 |(4,[0,1,2,3],[4.7,3.2,1.3,0.2])|

|1.0 |(4,[0,1,2,3],[4.6,3.1,1.5,0.2])|

|1.0 |(4,[0,1,2,3],[5.0,3.6,1.4,0.2])|

|1.0 |(4,[0,1,2,3],[5.4,3.9,1.7,0.4])|

|1.0 |(4,[0,1,2,3],[4.6,3.4,1.4,0.3])|

|1.0 |(4,[0,1,2,3],[5.0,3.4,1.5,0.2])|

|1.0 |(4,[0,1,2,3],[4.4,2.9,1.4,0.2])|

|1.0 |(4,[0,1,2,3],[4.9,3.1,1.5,0.1])|

|1.0 |(4,[0,1,2,3],[5.4,3.7,1.5,0.2])|

|1.0 |(4,[0,1,2,3],[4.8,3.4,1.6,0.2])|

|1.0 |(4,[0,1,2,3],[4.8,3.0,1.4,0.1])|

|1.0 |(4,[0,1,2,3],[4.3,3.0,1.1,0.1])|

|1.0 |(4,[0,1,2,3],[5.8,4.0,1.2,0.2])|

|1.0 |(4,[0,1,2,3],[5.7,4.4,1.5,0.4])|

|1.0 |(4,[0,1,2,3],[5.4,3.9,1.3,0.4])|

|1.0 |(4,[0,1,2,3],[5.1,3.5,1.4,0.3])|

|1.0 |(4,[0,1,2,3],[5.7,3.8,1.7,0.3])|

|1.0 |(4,[0,1,2,3],[5.1,3.8,1.5,0.3])|

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

only showing top 20 rows

root

|-- label: double (nullable = true)

|-- features: vector (nullable = true)

\*/

//2.数据归一化

//MinMaxScaler,区间缩放，基于最大最小值，将数据转换到0,1区间上的,可以提升模型收敛速度，提升模型精度

//X\_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))

//X\_scaled = X\_std \* (max - min) + min

val scaler: MinMaxScaler = new MinMaxScaler().setInputCol("features").setOutputCol("scaledFeatures")

//生成MinMaxScalerModel

val scalerModel: MinMaxScalerModel = scaler.fit(irisDF)

//将每个特征重新缩放到范围[最小，最大]

val scaledData: DataFrame = scalerModel.transform(irisDF)

scaledData.show(false)

/\*

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

|label|features |scaledFeatures |

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

|1.0 |(4,[0,1,2,3],[5.1,3.5,1.4,0.2])|[0.22222222222222213,0.6249999999999999,0.06779661016949151,0.04166666666666667] |

|1.0 |(4,[0,1,2,3],[4.9,3.0,1.4,0.2])|[0.1666666666666668,0.41666666666666663,0.06779661016949151,0.04166666666666667] |

|1.0 |(4,[0,1,2,3],[4.7,3.2,1.3,0.2])|[0.11111111111111119,0.5,0.05084745762711865,0.04166666666666667] |

|1.0 |(4,[0,1,2,3],[4.6,3.1,1.5,0.2])|[0.08333333333333327,0.4583333333333333,0.0847457627118644,0.04166666666666667] |

|1.0 |(4,[0,1,2,3],[5.0,3.6,1.4,0.2])|[0.19444444444444448,0.6666666666666666,0.06779661016949151,0.04166666666666667] |

|1.0 |(4,[0,1,2,3],[5.4,3.9,1.7,0.4])|[0.30555555555555564,0.7916666666666665,0.11864406779661016,0.12500000000000003] |

|1.0 |(4,[0,1,2,3],[4.6,3.4,1.4,0.3])|[0.08333333333333327,0.5833333333333333,0.06779661016949151,0.08333333333333333] |

|1.0 |(4,[0,1,2,3],[5.0,3.4,1.5,0.2])|[0.19444444444444448,0.5833333333333333,0.0847457627118644,0.04166666666666667] |

|1.0 |(4,[0,1,2,3],[4.4,2.9,1.4,0.2])|[0.027777777777777922,0.3749999999999999,0.06779661016949151,0.04166666666666667]|

|1.0 |(4,[0,1,2,3],[4.9,3.1,1.5,0.1])|[0.1666666666666668,0.4583333333333333,0.0847457627118644,0.0] |

|1.0 |(4,[0,1,2,3],[5.4,3.7,1.5,0.2])|[0.30555555555555564,0.7083333333333333,0.0847457627118644,0.04166666666666667] |

|1.0 |(4,[0,1,2,3],[4.8,3.4,1.6,0.2])|[0.13888888888888887,0.5833333333333333,0.1016949152542373,0.04166666666666667] |

|1.0 |(4,[0,1,2,3],[4.8,3.0,1.4,0.1])|[0.13888888888888887,0.41666666666666663,0.06779661016949151,0.0] |

|1.0 |(4,[0,1,2,3],[4.3,3.0,1.1,0.1])|[0.0,0.41666666666666663,0.016949152542372895,0.0] |

|1.0 |(4,[0,1,2,3],[5.8,4.0,1.2,0.2])|[0.41666666666666663,0.8333333333333333,0.033898305084745756,0.04166666666666667]|

|1.0 |(4,[0,1,2,3],[5.7,4.4,1.5,0.4])|[0.38888888888888895,1.0,0.0847457627118644,0.12500000000000003] |

|1.0 |(4,[0,1,2,3],[5.4,3.9,1.3,0.4])|[0.30555555555555564,0.7916666666666665,0.05084745762711865,0.12500000000000003] |

|1.0 |(4,[0,1,2,3],[5.1,3.5,1.4,0.3])|[0.22222222222222213,0.6249999999999999,0.06779661016949151,0.08333333333333333] |

|1.0 |(4,[0,1,2,3],[5.7,3.8,1.7,0.3])|[0.38888888888888895,0.7499999999999998,0.11864406779661016,0.08333333333333333] |

|1.0 |(4,[0,1,2,3],[5.1,3.8,1.5,0.3])|[0.22222222222222213,0.7499999999999998,0.0847457627118644,0.08333333333333333] |

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

only showing top 20 rows

\*/

//3.训练

val model: KMeansModel = new KMeans()

.setK(3)

.setSeed(10) //可重复的随机种子

.setFeaturesCol("scaledFeatures")

.setPredictionCol("prediction")//预测结果列

.fit(scaledData)

//4.预测

val result: DataFrame = model.transform(scaledData)

result.show(false)

/\*

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

|label|features |scaledFeatures |prediction|

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

|1.0 |(4,[0,1,2,3],[5.1,3.5,1.4,0.2])|[0.22222222222222213,0.6249999999999999,0.06779661016949151,0.04166666666666667] |0 |

|1.0 |(4,[0,1,2,3],[4.9,3.0,1.4,0.2])|[0.1666666666666668,0.41666666666666663,0.06779661016949151,0.04166666666666667] |0 |

|1.0 |(4,[0,1,2,3],[4.7,3.2,1.3,0.2])|[0.11111111111111119,0.5,0.05084745762711865,0.04166666666666667] |0 |

|1.0 |(4,[0,1,2,3],[4.6,3.1,1.5,0.2])|[0.08333333333333327,0.4583333333333333,0.0847457627118644,0.04166666666666667] |0 |

|1.0 |(4,[0,1,2,3],[5.0,3.6,1.4,0.2])|[0.19444444444444448,0.6666666666666666,0.06779661016949151,0.04166666666666667] |0 |

|1.0 |(4,[0,1,2,3],[5.4,3.9,1.7,0.4])|[0.30555555555555564,0.7916666666666665,0.11864406779661016,0.12500000000000003] |0 |

|1.0 |(4,[0,1,2,3],[4.6,3.4,1.4,0.3])|[0.08333333333333327,0.5833333333333333,0.06779661016949151,0.08333333333333333] |0 |

|1.0 |(4,[0,1,2,3],[5.0,3.4,1.5,0.2])|[0.19444444444444448,0.5833333333333333,0.0847457627118644,0.04166666666666667] |0 |

|1.0 |(4,[0,1,2,3],[4.4,2.9,1.4,0.2])|[0.027777777777777922,0.3749999999999999,0.06779661016949151,0.04166666666666667]|0 |

|1.0 |(4,[0,1,2,3],[4.9,3.1,1.5,0.1])|[0.1666666666666668,0.4583333333333333,0.0847457627118644,0.0] |0 |

|1.0 |(4,[0,1,2,3],[5.4,3.7,1.5,0.2])|[0.30555555555555564,0.7083333333333333,0.0847457627118644,0.04166666666666667] |0 |

|1.0 |(4,[0,1,2,3],[4.8,3.4,1.6,0.2])|[0.13888888888888887,0.5833333333333333,0.1016949152542373,0.04166666666666667] |0 |

|1.0 |(4,[0,1,2,3],[4.8,3.0,1.4,0.1])|[0.13888888888888887,0.41666666666666663,0.06779661016949151,0.0] |0 |

|1.0 |(4,[0,1,2,3],[4.3,3.0,1.1,0.1])|[0.0,0.41666666666666663,0.016949152542372895,0.0] |0 |

|1.0 |(4,[0,1,2,3],[5.8,4.0,1.2,0.2])|[0.41666666666666663,0.8333333333333333,0.033898305084745756,0.04166666666666667]|0 |

|1.0 |(4,[0,1,2,3],[5.7,4.4,1.5,0.4])|[0.38888888888888895,1.0,0.0847457627118644,0.12500000000000003] |0 |

|1.0 |(4,[0,1,2,3],[5.4,3.9,1.3,0.4])|[0.30555555555555564,0.7916666666666665,0.05084745762711865,0.12500000000000003] |0 |

|1.0 |(4,[0,1,2,3],[5.1,3.5,1.4,0.3])|[0.22222222222222213,0.6249999999999999,0.06779661016949151,0.08333333333333333] |0 |

|1.0 |(4,[0,1,2,3],[5.7,3.8,1.7,0.3])|[0.38888888888888895,0.7499999999999998,0.11864406779661016,0.08333333333333333] |0 |

|1.0 |(4,[0,1,2,3],[5.1,3.8,1.5,0.3])|[0.22222222222222213,0.7499999999999998,0.0847457627118644,0.08333333333333333] |0 |

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

only showing top 20 rows

\*/

//5.查看各类别数量

result.groupBy($"label", $"prediction").count().show()

/\*

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

|label|prediction|count|

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

| 2.0| 1| 47|

| 1.0| 0| 50|

| 2.0| 2| 3|

| 3.0| 1| 14|

| 3.0| 2| 36|

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

\*/

spark.stop()

}

}

## 3.3项目代码实战

package cn.itcast.up.ml

import cn.itcast.up.base.BaseModel2

import org.apache.spark.ml.clustering.{KMeans, KMeansModel}

import org.apache.spark.ml.feature.VectorAssembler

import org.apache.spark.sql.\_

import scala.collection.immutable

/\*\*

\* Author itcast

\* Date 2019/10/28 9:22

\* Desc 客户价值模型-RFM:

\* Rencency:最近一次消费,最后一次订单距今时间

\* Frequency:消费频率,订单总数量

\* Monetary:消费金额,订单总金额

\* https://blog.csdn.net/liam08/article/details/79663018

\*/

object RFMModel extends BaseModel2{

def main(args: Array[String]): Unit = {

execute()

}

/\*\*

\* 获取标签id(即模型id,该方法应该在编写不同模型时进行实现)

\* @return

\*/

override def getTagID(): Int = 37

/\*\*

\* 开始计算

\* inType=HBase##zkHosts=192.168.10.20##zkPort=2181##

\* hbaseTable=tbl\_orders##family=detail##selectFields=memberId,orderSn,orderAmount,finishTime

\* @param fiveDF MySQL中的5级规则 id,rule

\* @param hbaseDF 根据selectFields查询出来的HBase中的数据

\* @return userid,tagIds

\*/

override def compute(fiveDF: DataFrame, hbaseDF: DataFrame): DataFrame = {

//fiveDF.show()

//fiveDF.printSchema()

//hbaseDF.show(10)

//hbaseDF.printSchema()

/\*

+---+----+

| id|rule|

+---+----+

| 38| 1|

| 39| 2|

| 40| 3|

| 41| 4|

| 42| 5|

| 43| 6|

| 44| 7|

+---+----+

root

|-- id: long (nullable = false)

|-- rule: string (nullable = true)

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

| memberId| orderSn|orderAmount|finishTime|

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

| 13823431| ts\_792756751164275| 2479.45|1564415022|

| 4035167| D14090106121770839| 2449.00|1565687310|

| 4035291| D14090112394810659| 1099.42|1564681801|

| 4035041| fx\_787749561729045| 1999.00|1565799378|

| 13823285| D14092120154435903| 2488.00|1565062072|

| 4034219| D14092120155620305| 3449.00|1563601306|

|138230939|top\_810791455519102| 1649.00|1565509622|

| 4035083| D14092120161884409| 7.00|1565731851|

|138230935| D14092120162313538| 1299.00|1565382991|

| 13823231| D14092120162378713| 499.00|1565677650|

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

only showing top 10 rows

root

|-- memberId: string (nullable = true)

|-- orderSn: string (nullable = true)

|-- orderAmount: string (nullable = true)

|-- finishTime: string (nullable = true)

\*/

import spark.implicits.\_

import scala.collection.JavaConversions.\_

import org.apache.spark.sql.functions.\_

//0.定义常量字符串,避免后续拼写错误

val recencyStr = "recency"

val frequencyStr = "frequency"

val monetaryStr = "monetary"

val featureStr = "feature"

val predictStr = "predict"

//1.按用户id进行聚合获取客户RFM

//客户价值模型-RFM:

//Rencency:最近一次消费,最后一次订单距今时间

//Frequency:消费频率,订单总数量

//Monetary:消费金额,订单总金额

//https://blog.csdn.net/liam08/article/details/79663018

val recencyAggColumn: Column = functions.datediff(date\_sub(current\_timestamp(),60), from\_unixtime(max('finishTime))) as recencyStr

val frequencyAggColumn: Column = functions.count('orderSn) as frequencyStr

val monetaryAggColumn: Column = functions.sum('orderAmount) as monetaryStr

val RFMResult = hbaseDF

.groupBy('memberId)

.agg(recencyAggColumn, frequencyAggColumn, monetaryAggColumn)

//RFMResult.show(10)

//RFMResult.printSchema()

//RFMResult.groupBy(recencyStr).count().show()

/\*

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

| memberId|recency|frequency| monetary|

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

| 13822725| 12| 116| 179298.34|

| 13823083| 12| 132| 233524.17|

|138230919| 12| 125|240061.56999999998|

| 13823681| 12| 108| 169746.1|

| 4033473| 12| 142| 251930.92|

| 13822841| 12| 113| 205931.91|

| 13823153| 12| 133| 250698.57|

| 13823431| 12| 122| 180858.22|

| 4033348| 12| 145|240173.78999999998|

| 4033483| 12| 110|157811.09999999998|

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

only showing top 10 rows

root

|-- memberId: string (nullable = true)

|-- recency: integer (nullable = true)

|-- frequency: long (nullable = false)

|-- monetary: double (nullable = true)

\*/

//2.为RFM打分

//R: 1-3天=5分，4-6天=4分，7-9天=3分，10-15天=2分，大于16天=1分

//F: ≥200=5分，150-199=4分，100-149=3分，50-99=2分，1-49=1分

//M: ≥20w=5分，10-19w=4分，5-9w=3分，1-4w=2分，<1w=1分

val recencyScore: Column = functions.when((col(recencyStr) >= 1) && (col(recencyStr) <= 3), 5)

.when((col(recencyStr) >= 4) && (col(recencyStr) <= 6), 4)

.when((col(recencyStr) >= 7) && (col(recencyStr) <= 9), 3)

.when((col(recencyStr) >= 10) && (col(recencyStr) <= 15), 2)

.when(col(recencyStr) >= 16, 1)

.as(recencyStr)

val frequencyScore: Column = functions.when(col(frequencyStr) >= 200, 5)

.when((col(frequencyStr) >= 150) && (col(frequencyStr) <= 199), 4)

.when((col(frequencyStr) >= 100) && (col(frequencyStr) <= 149), 3)

.when((col(frequencyStr) >= 50) && (col(frequencyStr) <= 99), 2)

.when((col(frequencyStr) >= 1) && (col(frequencyStr) <= 49), 1)

.as(frequencyStr)

val monetaryScore: Column = functions.when(col(monetaryStr) >= 200000, 5)

.when(col(monetaryStr).between(100000, 199999), 4)

.when(col(monetaryStr).between(50000, 99999), 3)

.when(col(monetaryStr).between(10000, 49999), 2)

.when(col(monetaryStr) <= 9999, 1)

.as(monetaryStr)

val RFMScoreResult: DataFrame = RFMResult.select('memberId, recencyScore, frequencyScore, monetaryScore)

//RFMScoreResult.show(10)

/\*

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

| memberId|recency|frequency|monetary|

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

| 13822725| 2| 3| 4|

| 13823083| 2| 3| 5|

|138230919| 2| 3| 5|

| 13823681| 2| 3| 4|

| 4033473| 2| 3| 5|

| 13822841| 2| 3| 5|

| 13823153| 2| 3| 5|

| 13823431| 2| 3| 4|

| 4033348| 2| 3| 5|

| 4033483| 2| 3| 4|

| 4033575| 2| 3| 5|

| 4034191| 2| 3| 5|

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

only showing top 200 rows

\*/

//3.聚类

//为方便后续模型进行特征输入，需要部分列的数据转换为特征向量，并统一命名，VectorAssembler类就可以完成这一任务。

//VectorAssembler是一个transformer，将多列数据转化为单列的向量列

val vectorDF: DataFrame = new VectorAssembler()

.setInputCols(Array(recencyStr, frequencyStr, monetaryStr))

.setOutputCol(featureStr)

.transform(RFMScoreResult)

//vectorDF.show(10)

/\*

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

| memberId|recency|frequency|monetary| feature|

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

| 13822725| 2| 3| 4|[2.0,3.0,4.0]|

| 13823083| 2| 3| 5|[2.0,3.0,5.0]|

|138230919| 2| 3| 5|[2.0,3.0,5.0]|

| 13823681| 2| 3| 4|[2.0,3.0,4.0]|

| 4033473| 2| 3| 5|[2.0,3.0,5.0]|

| 13822841| 2| 3| 5|[2.0,3.0,5.0]|

| 13823153| 2| 3| 5|[2.0,3.0,5.0]|

| 13823431| 2| 3| 4|[2.0,3.0,4.0]|

| 4033348| 2| 3| 5|[2.0,3.0,5.0]|

| 4033483| 2| 3| 4|[2.0,3.0,4.0]|

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

\*/

val kMeans: KMeans = new KMeans()

.setK(7)

.setSeed(10) //可重复的随机种子

.setMaxIter(2)//最大迭代次数

.setFeaturesCol(featureStr) //特征列

.setPredictionCol(predictStr)//预测结果列

//4.训练模型

val model: KMeansModel = kMeans.fit(vectorDF)

model.save("/model/RFMModel/")

//val model = KMeansModel.load("/model/RFMModel/")

//5.预测

val result: DataFrame = model.transform(vectorDF)

//result.show(10)

/\*

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

| memberId|recency|frequency|monetary| feature|predict|

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

| 13822725| 2| 3| 4|[2.0,3.0,4.0]| 1|

| 13823083| 2| 3| 5|[2.0,3.0,5.0]| 0|

|138230919| 2| 3| 5|[2.0,3.0,5.0]| 0|

| 13823681| 2| 3| 4|[2.0,3.0,4.0]| 1|

| 4033473| 2| 3| 5|[2.0,3.0,5.0]| 0|

| 13822841| 2| 3| 5|[2.0,3.0,5.0]| 0|

| 13823153| 2| 3| 5|[2.0,3.0,5.0]| 0|

| 13823431| 2| 3| 4|[2.0,3.0,4.0]| 1|

| 4033348| 2| 3| 5|[2.0,3.0,5.0]| 0|

| 4033483| 2| 3| 4|[2.0,3.0,4.0]| 1|

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

\*/

//6.测试时看下聚类效果

val ds: Dataset[Row] = result

.groupBy(predictStr)

.agg(max(col(recencyStr) + col(frequencyStr) + col(monetaryStr)), min(col(recencyStr) + col(frequencyStr) + col(monetaryStr)))

.sort(col(predictStr).asc)

//ds.show()

/\*

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

|predict|max(((recency + frequency) + monetary))|min(((recency + frequency) + monetary))|

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

| 0| 10| 9|

| 1| 9| 8|

| 2| 4| 3|

| 3| 6| 6|

| 4| 7| 7|

| 5| 12| 11|

| 6| 5| 5|

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

\*/

//问题: 每一个簇的ID是无序的,但是我们将分类簇和rule进行对应的时候,需要有序

//7.按质心排序,质心大,该类用户价值大

//[(质心id, 质心值)]

val center: immutable.IndexedSeq[(Int, Double)] = for(i <- model.clusterCenters.indices) yield (i, model.clusterCenters(i).toArray.sum)

val sortedCenter: immutable.IndexedSeq[(Int, Double)] = center.sortBy(\_.\_2).reverse

//sortedCenter.foreach(println)

/\*

(5,11.038461538461538)

(0,10.0)

(1,9.0)

(6,8.0)

(3,5.333333333333333)

(2,4.0)

(4,3.0)

\*/

//[(质心id, rule值)]

val centerIdAndRule: immutable.IndexedSeq[(Int, Int)] = for(i <- sortedCenter.indices) yield (sortedCenter(i).\_1, i+1)

//centerIdAndRule.foreach(println)

/\*

(5,1)

(0,2)

(1,3)

(6,4)

(3,5)

(2,6)

(4,7)

\*/

val predictRuleDF: DataFrame = centerIdAndRule.toDF(predictStr,"rule")

//predictRuleDF.show()

/\*

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

|predict|rule|

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

| 5| 1|

| 0| 2|

| 1| 3|

| 6| 4|

| 3| 5|

| 2| 6|

| 4| 7|

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

\*/

//8.将rule和5级规则进行匹配

val ruleTagDF: DataFrame = fiveDF.as[(Long,String)].map(t=>(t.\_2,t.\_1)).toDF("rule","tag")

ruleTagDF.show()

/\*

+----+---+

|rule|tag|

+----+---+

| 1| 38|

| 2| 39|

| 3| 40|

| 4| 41|

| 5| 42|

| 6| 43|

| 7| 44|

+----+---+

\*/

val predictRuleTagDF: DataFrame = predictRuleDF.join(ruleTagDF,"rule")

predictRuleTagDF.show()

/\*

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

|rule|predict|tag|

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

| 1| 5| 38|

| 2| 0| 39|

| 3| 1| 40|

| 4| 6| 41|

| 5| 3| 42|

| 6| 2| 43|

| 7| 4| 44|

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

\*/

//Map[predict, tag]

val map: Map[Long, Long] = predictRuleTagDF.as[(String,Long,Long)].map(t=>(t.\_2,t.\_3)).collect().toMap

val predict2tag = functions.udf((predict:Long)=>{

val tag = map(predict)

tag

})

//8.predict->tag

val newDF: DataFrame = result.select($"memberId".as("userId"),predict2tag($"predict").as("tagIds"))

newDF.show(10)

/\*

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

| userId|tagIds|

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

| 13822725| 40|

| 13823083| 39|

|138230919| 39|

| 13823681| 40|

| 4033473| 39|

| 13822841| 39|

| 13823153| 39|

| 13823431| 40|

| 4033348| 39|

| 4033483| 40|

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

\*/

newDF

}

}

# 4.标签模型开发

## 4.1新建标签

**新建业务（4级）标签：客户价值标签，相关字段信息如下：**

标签名称：客户价值

标签分类：电商-某商城-商业属性

更新周期：1天

业务含义：客户价值分为7类（依据产品或业务需求划分）

标签规则：

inType=hbase

zkHosts=bigdata-cdh01.itcast.cn

zkPort=2181

hbaseTable=tbl\_tag\_orders

family=detail

selectFieldNames=memberid,ordersn,orderamount,finishtime

程序入口：

cn.itcast.tags.models.ml.RfmModel

算法名称：

KMEANS

算法引擎：

tags-model\_2.11.jar

模型参数：

--driver-memory 512m --executor-memory 512m --num-executors 1 --

executor-cores 1

**新建属性（5级）标签，相关字段信息如下：**

1）、属性值【超高价值】

标签名称：超高价值

标签含义：客户类型为：超高价值

标签规则：1

2）、属性值【高价值】

标签名称：高价值

标签含义：客户类型为：高价值

标签规则：2

3）、属性值【中上价值】

标签名称：中上价值

标签含义：客户类型为：中上价值

标签规则：3

4）、属性值【中价值】

标签名称：中价值

标签含义：客户类型为：中价值

标签规则：4

5）、属性值【中下价值】

标签名称：中下价值

标签含义：客户类型为：中下价值

标签规则：5

6）、属性值【低价值】

标签名称：低价值

标签含义：客户类型为：低价值

标签规则：6

7）、属性值【超低价值】

标签名称：超低价值

标签含义：客户类型为：超低价值

标签规则：7

属性标签插入SQL语句：

INSERT INTO `tbl\_basic\_tag` VALUES ('363', '超高价值', null, '1', null,

'5', '362', '2019-12-07 17:12:15', '2019-12-07 17:12:15', null, null);

INSERT INTO `tbl\_basic\_tag` VALUES ('364', '高价值', null, '2', null,

'5', '362', '2019-12-07 17:12:24', '2019-12-07 17:12:24', null, null);

INSERT INTO `tbl\_basic\_tag` VALUES ('365', '中上价值', null, '3', null,

'5', '362', '2019-12-07 17:12:31', '2019-12-07 17:12:31', null, null);

INSERT INTO `tbl\_basic\_tag` VALUES ('366', '中价值', null, '4', null,

'5', '362', '2019-12-07 17:12:40', '2019-12-07 17:12:40', null, null);

INSERT INTO `tbl\_basic\_tag` VALUES ('367', '中下价值', null, '5', null,

'5', '362', '2019-12-07 17:12:48', '2019-12-07 17:12:48', null, null);

INSERT INTO `tbl\_basic\_tag` VALUES ('368', '低价值', null, '6', null,

'5', '362', '2019-12-07 17:12:57', '2019-12-07 17:12:57', null, null);

INSERT INTO `tbl\_basic\_tag` VALUES ('369', '超低价值', null, '7', null,

'5', '362', '2019-12-07 17:13:05', '2019-12-07 17:13:05', null, null);

## 4.2模型开发

继承基类AbstractModel ，实现标签计算方法doTag ，涉及到订单表数据中字段值：

10 column=detail:memberid, timestamp=1574240145469, value=13823431

10 column=detail:ordersn, timestamp=1574240145469,value=gome\_792756751164275

10 column=detail:orderamount, timestamp=1574240145469,value=2479.45

10 column=detail:finishtime, timestamp=1574240145469,value=1564415022

针对RFM统计数值使用规则打分，具体打分规则如下所述：

R: 1-3天=5分，4-6天=4分，7-9天=3分，10-15天=2分，大于16天=1分

F: ≥200=5分，150-199=4分，100-149=3分，50-99=2分，1-49=1分

M: ≥20w=5分，10-19w=4分，5-9w=3分，1-4w=2分，<1w=1分

**标签实现思路分析：**

1）、从订单数据获取字段值，计算每个用户RFM值

R: 最后一次消费时间 距离 今天 的天数

越小越好

F: 消费次数（订单次数）

越大越好

M：消费金额（所有订单总的消费金额）

越大越好

2）、按照规则，给RFM值打分Score

R: 1-3天=5分，4-6天=4分，7-9天=3分，10-15天=2分，大于16天=1分

F: ≥200=5分，150-199=4分，100-149=3分，50-99=2分，1-49=1分

M: ≥20w=5分，10-19w=4分，5-9w=3分，1-4w=2分，<1w=1分

3）、将RFM数据使用KMeans算法聚类（K=7个）

RFM转换为特征features，传入到KMeans算法，训练获取模型KMeansModel

4）、从KMeans中获取出每个用户属于簇

0,1,2,3,4,5,6

每个簇的向量Vector中所有特征之和越大客户价值越高

按照上述思路，依次代码实现说明如下：

### 0-主函数写法

**object** RFMModel **extends** BaseModel{  
 **def** main(args: Array[String]): Unit = {  
 execute()  
 }  
 */\*\*  
 \* 获取标签id(即模型id,该方法应该在编写不同模型时进行实现)  
 \** **@return** *\*/* **override def** getTagID(): Int = 37  
  
 */\*\*  
 \* 开始计算  
 \** **@param fiveDF** *MySQL中的5级规则 id,rule  
 \** **@param hbaseDF** *根据selectFields查询出来的HBase中的数据  
 \* @return userid,tagIds*

*inType=HBase##zkHosts=192.168.10.20*

*##zkPort=2181*

*##hbaseTable=tbl\_orders*

*##family=detail*

*##selectFields=memberId,orderSn,orderAmount,finishTime*

*\*/* **override def** compute(fiveDF: DataFrame, hbaseDF: DataFrame): DataFrame = {  
 *//fiveDF.show(10)  
 //fiveDF.printSchema()  
 /\*  
+---+----+  
| id|rule|  
+---+----+  
| 38| 1|  
| 39| 2|  
| 40| 3|  
| 41| 4|  
| 42| 5|  
| 43| 6|  
| 44| 7|  
+---+----+  
/\**

*客户价值*

*超高价值 1*

*高价值 2*

*中上价值 3*

*中价值 4*

*中下价值 5*

*低价值 6*

*超低价值 7*

*\*/*

*root  
 |-- id: long (nullable = false)  
 |-- rule: string (nullable = true)*

*//hbaseDF.show(10)  
 // hbaseDF.printSchema()  
+---------+-------------------+-----------+----------+  
| memberId| orderSn|orderAmount|finishTime|  
+---------+-------------------+-----------+----------+  
| 13823431| ts\_792756751164275| 2479.45|1564415022|  
| 4035167| D14090106121770839| 2449.00|1565687310|  
| 4035291| D14090112394810659| 1099.42|1564681801|  
| 4035041| fx\_787749561729045| 1999.00|1565799378|  
| 13823285| D14092120154435903| 2488.00|1565062072|  
| 4034219| D14092120155620305| 3449.00|1563601306|  
|138230939|top\_810791455519102| 1649.00|1565509622|  
| 4035083| D14092120161884409| 7.00|1565731851|  
|138230935| D14092120162313538| 1299.00|1565382991|  
| 13823231| D14092120162378713| 499.00|1565677650|  
+---------+-------------------+-----------+----------+  
only showing top 10 rows  
  
root  
 |-- memberId: string (nullable = true)  
 |-- orderSn: string (nullable = true)  
 |-- orderAmount: string (nullable = true)  
 |-- finishTime: string (nullable = true)  
 \*/*

### 第一、从业务数据中计算R、F、M值

*//0.定义常量字符串,避免后续拼写错误* **val** recencyStr = **"recency"  
 val** frequencyStr = **"frequency"  
 val** monetaryStr = **"monetary"  
 val** featureStr = **"feature"  
 val** predictStr = **"predict"** *//https://blog.csdn.net/liam08/article/details/79663018* **import** org.apache.spark.sql.functions.\_  
 **import** *spark*.implicits.\_  
  
 *//1.统计每个用户的RFM  
 //Rencency:最近一次消费,最后一次订单距今时间  
 //Frequency:消费频率,订单总数量  
 //Monetary:消费金额,订单总金额  
  
 //max("finishTime"))求每个用户的最近一次的订单时间  
 //from\_unixtime()将字符串时间戳转为时间对象  
 //datediff(结束时间,开始时间)获取两个时间的时间差--天数* **val** rencencyAggColumn: Column = functions.*datediff*(*date\_sub*(*current\_timestamp*(),60),*from\_unixtime*(*max*(**"finishTime"**))) as recencyStr  
 **val** frequencyAggColumn: Column = functions.*count*(**"orderSn"**) as frequencyStr  
 **val** monetaryAggColumn: Column = functions.*sum*(**"orderAmount"**) as monetaryStr

*//根据用户id的memberId求解三个RFM的值*  
 **val** tempDF: DataFrame = hbaseDF  
 .groupBy(**"memberId"**)  
 .agg(rencencyAggColumn, frequencyAggColumn, monetaryAggColumn)  
 *//tempDF.show(10,false)  
  
 /\*  
+---------+-------+---------+------------------+  
|memberId |recency|frequency|monetary |  
+---------+-------+---------+------------------+  
|13822725 |17 |116 |179298.34 |  
|13823083 |17 |132 |233524.17 |  
|138230919|17 |125 |240061.56999999998|  
|13823681 |17 |108 |169746.1 |  
|4033473 |17 |142 |251930.92 |  
|13822841 |17 |113 |205931.91 |  
|13823153 |17 |133 |250698.57 |  
|13823431 |17 |122 |180858.22 |  
|4033348 |17 |145 |240173.78999999998|  
|4033483 |17 |110 |157811.09999999998|  
+---------+-------+---------+------------------+  
only showing top 10 rows  
 \*/*

### 第二、按照规则，给计算RFM值打分Score

*//2.数据打分(相当于归一化)  
 //R: 1-3天=5分，4-6天=4分，7-9天=3分，10-15天=2分，大于16天=1分  
 //F: ≥200=5分，150-199=4分，100-149=3分，50-99=2分，1-49=1分  
 //M: ≥20w=5分，10-19w=4分，5-9w=3分，1-4w=2分，<1w=1分* **val** recencyScore: Column = functions.*when*((*col*(recencyStr) >= 1) && (*col*(recencyStr) <= 3), 5)  
 .when((*col*(recencyStr) >= 4) && (*col*(recencyStr) <= 6), 4)  
 .when((*col*(recencyStr) >= 7) && (*col*(recencyStr) <= 9), 3)  
 .when((*col*(recencyStr) >= 10) && (*col*(recencyStr) <= 15), 2)  
 .when(*col*(recencyStr) >= 16, 1)  
 .as(recencyStr)  
  
 **val** frequencyScore: Column = functions.*when*(*col*(frequencyStr) >= 200, 5)  
 .when((*col*(frequencyStr) >= 150) && (*col*(frequencyStr) <= 199), 4)  
 .when((*col*(frequencyStr) >= 100) && (*col*(frequencyStr) <= 149), 3)  
 .when((*col*(frequencyStr) >= 50) && (*col*(frequencyStr) <= 99), 2)  
 .when((*col*(frequencyStr) >= 1) && (*col*(frequencyStr) <= 49), 1)  
 .as(frequencyStr)  
  
 **val** monetaryScore: Column = functions.*when*(*col*(monetaryStr) >= 200000, 5)  
 .when(*col*(monetaryStr).between(100000, 199999), 4)  
 .when(*col*(monetaryStr).between(50000, 99999), 3)  
 .when(*col*(monetaryStr).between(10000, 49999), 2)  
 .when(*col*(monetaryStr) <= 9999, 1)  
 .as(monetaryStr)  
  
  
 **val** RFMScoreDF: DataFrame = tempDF.select(**$"memberId"**,recencyScore,frequencyScore,monetaryScore)  
 *//RFMScoreDF.show(10,false)  
 /\*  
+---------+-------+---------+--------+  
|memberId |recency|frequency|monetary|  
+---------+-------+---------+--------+  
|13822725 |1 |3 |4 |  
|13823083 |1 |3 |5 |  
|138230919|1 |3 |5 |  
|13823681 |1 |3 |4 |  
|4033473 |1 |3 |5 |  
|13822841 |1 |3 |5 |  
|13823153 |1 |3 |5 |  
|13823431 |1 |3 |4 |  
|4033348 |1 |3 |5 |  
|4033483 |1 |3 |4 |  
+---------+-------+---------+--------+  
 \*/*

### 第三、使用聚类算法KMeans训练模型与预测

*//再使用SparkMLlib中的归一化工具进行数据缩放到[0,1]之间也可以  
  
 //3.我们在使用机器学习算法的时候要的是向量,所以先对特征进行向量化表示  
 //vectorAssembler可以将多列数据转换成一列特征的向量表示(特征向量)* **val** vectorAssembler: VectorAssembler = **new** VectorAssembler()  
 .setInputCols(*Array*(recencyStr, frequencyStr, monetaryStr))  
 .setOutputCol(featureStr)  
 **val** VectorDF: DataFrame = vectorAssembler.transform(RFMScoreDF)  
 *//VectorDF.show(10,false)  
/\*  
+---------+-------+---------+--------+-------------+  
|memberId |recency|frequency|monetary|feature |  
+---------+-------+---------+--------+-------------+  
|13822725 |1 |3 |4 |[1.0,3.0,4.0]|  
|13823083 |1 |3 |5 |[1.0,3.0,5.0]|  
|138230919|1 |3 |5 |[1.0,3.0,5.0]|  
|13823681 |1 |3 |4 |[1.0,3.0,4.0]|  
|4033473 |1 |3 |5 |[1.0,3.0,5.0]|  
|13822841 |1 |3 |5 |[1.0,3.0,5.0]|  
|13823153 |1 |3 |5 |[1.0,3.0,5.0]|  
|13823431 |1 |3 |4 |[1.0,3.0,4.0]|  
|4033348 |1 |3 |5 |[1.0,3.0,5.0]|  
|4033483 |1 |3 |4 |[1.0,3.0,4.0]|  
+---------+-------+---------+--------+-------------+  
 \*/  
  
 //4.聚类  
 //训练* **val** model: KMeansModel = **new** KMeans()  
 .setFeaturesCol(featureStr)  
 .setPredictionCol(predictStr)  
 .setK(7) *//实际业务运营要求分为7类* .setMaxIter(10) *//最大迭代次数* .setSeed(10) *//可重复的随子种子* .fit(VectorDF)  
 *//预测* **val** result: DataFrame = model.transform(VectorDF)  
 *//result.show(10,false)  
 /\*  
 +---------+-------+---------+--------+-------------+-------+  
|memberId |recency|frequency|monetary|feature |predict|  
+---------+-------+---------+--------+-------------+-------+  
|13822725 |1 |3 |4 |[1.0,3.0,4.0]|1 |  
|13823083 |1 |3 |5 |[1.0,3.0,5.0]|0 |  
|138230919|1 |3 |5 |[1.0,3.0,5.0]|0 |  
|13823681 |1 |3 |4 |[1.0,3.0,4.0]|1 |  
|4033473 |1 |3 |5 |[1.0,3.0,5.0]|0 |  
|13822841 |1 |3 |5 |[1.0,3.0,5.0]|0 |  
|13823153 |1 |3 |5 |[1.0,3.0,5.0]|0 |  
|13823431 |1 |3 |4 |[1.0,3.0,4.0]|1 |  
|4033348 |1 |3 |5 |[1.0,3.0,5.0]|0 |  
|4033483 |1 |3 |4 |[1.0,3.0,4.0]|1 |  
+---------+-------+---------+--------+-------------+-------+  
only showing top 10 rows  
 \*/*

### 第四、获取聚类类簇中心点，进行数据转换处理

*//5.测试时看下聚类效果--每个聚类组或簇中的最大的rfm值相加和最小的rfm值相加，在按照预测值进行排序，可以看到7个类别，从高价值到低价值的排序* **val** ds: Dataset[Row] = result  
 .groupBy(predictStr)  
 .agg(*max*(*col*(recencyStr) + *col*(frequencyStr) + *col*(monetaryStr)), *min*(*col*(recencyStr) + *col*(frequencyStr) + *col*(monetaryStr)))  
 .sort(*col*(predictStr).asc)  
 *//ds.show()  
 /\*  
 +-------+---------------------------+--------------------+  
|predict|max(((recency + frequency) + monetary))|min(((recency + frequency) + monetary))|+-------+------------+  
| 0| 9| 9|  
| 1| 8| 8|  
| 2| 3| 3|  
| 3| 7| 7|  
| 4| 11| 10|  
| 5| 5| 4|  
| 6| 8| 7|  
+-------+---------+  
 \*/  
  
 //6.获取聚类编号和聚类中心  
 //接下来我们要对****聚类之后的用户打Tag****,不能仅仅根据聚类的序号打Tag  
 //因为****聚类序号并不能代表客户的价值等级,和我们5级规则并不对应*** *//所以****我们应该求出每个聚类的聚类中心,排序,聚类中心的值越大说明该用户群体价值越高*** *//model.clusterCenters.indices获取所有聚类中心的索引编号  
 //model.clusterCenters(i)根据索引取聚类中心  
 //model.clusterCenters(i).toArray.sum求该聚类中心的RFM的和*

*//聚类中心是7个点，分别是由recencyStr, frequencyStr, monetaryStr组成  
 //[(聚类索引/编号, 聚类中心的RFM的和)]* **val** indexAndRFM: immutable.IndexedSeq[(Int, Double)] =

**for**(i <- model.clusterCenters.indices) **yield** (i,model.clusterCenters(i).toArray.sum)  
  *//和上面不同的写法，结果一致*  
 **val** tuples: immutable.IndexedSeq[(Int, Double)] = model.clusterCenters.indices.map(i=>(i,model.clusterCenters(i).toArray.sum))  
  
 *//indexAndRFM.foreach(println)  
 /\*  
(0,9.0)  
(1,8.0)  
(2,3.0)  
(3,7.0)  
(4,10.038461538461538)  
(5,4.333333333333333)  
(6,7.5)  
  
 \*/  
 //println("==================")  
 //根据聚类中心的RFM的和[(聚类索引/编号, 聚类中心的RFM的和)]*sortBy*排序* **val** sortedIndexAndRFM: immutable.IndexedSeq[(Int, Double)] = indexAndRFM.sortBy(\_.\_2).reverse  
 *//sortedIndexAndRFM.foreach(println)  
/\*  
(4,10.038461538461538)  
(0,9.0)  
(1,8.0)  
(6,7.5)  
(3,7.0)  
(5,4.333333333333333)  
(2,3.0)  
 \*/*

### 第五、类簇点分类数据与标签数据合并，进行打标签

*//7.将上面的排好序的聚类编号和聚类中心与5级规则分别转化为DataSet进行对应，在转化为RDD的repartition进行用zip拉链操作，最后进行选择predict|tagIds  
 /\*  
5级规则fiveDF:  
+---+----+  
| id|rule|  
+---+----+  
| 38| 1|  
| 39| 2|  
| 40| 3|  
| 41| 4|  
| 42| 5|  
| 43| 6|  
| 44| 7|  
+---+----+  
 \*/  
 //目标:  
 /\*  
+-------+---  
predict| id|  
+------+----  
| 4的聚类中心| 38标签Id|  
| 0| 39|  
| 1| 40|  
| 6| 41|  
| 3| 42|  
| 5| 43|  
| 2| 44|  
+--------+--  
 \*/* **val** indexAndRFMDS: Dataset[(Int, Double)] = sortedIndexAndRFM.toDS()  
 **val** fiveDS: Dataset[(Long, String)] = fiveDF.as[(Long,String)]  
 **val** tempRDD: RDD[((Int, Double), (Long, String))] = indexAndRFMDS.*rdd*.repartition(1).zip(fiveDS.*rdd*.repartition(1))  
 *//tempRDD.collect().foreach(println)  
/\*  
((4,10.038461538461538),(38,1))  
((0,9.0),(39,2))  
((1,8.0),(40,3))  
((6,7.5),(41,4))  
((3,7.0),(42,5))  
((5,4.333333333333333),(43,6))  
((2,3.0),(44,7))  
 \*/*

*//选择tempRDD的预测值和标签Id* **val** ruleDF: DataFrame = tempRDD.map(t=>(t.\_1.\_1,t.\_2.\_1)).toDF(**"predict"**,**"tagIds"**)  
 *//ruleDF.show()  
/\*  
+-------+------+  
|predict|tagIds|  
+-------+------+  
| 4| 38|  
| 0| 39|  
| 1| 40|  
| 6| 41|  
| 3| 42|  
| 5| 43|  
| 2| 44|  
+-------+------+  
 \*/*

### 第六、写入Hbase

*//8将ruleDF首先转化为DataSet转化为map类型，使用rdd的collectAsMap操作，在使用udf转换，比如预测为predict=4代表的是38号标签* **val** ruleMap: collection.Map[Int, Long] = ruleDF.as[(Int,Long)].*rdd*.collectAsMap()  
  
 **val** predict2Tag = *udf*((predict:Int)=>{  
 ruleMap(predict)  
 })  
*//使用经过Kmeans聚类的result结果选择memberId'predict并作为一个newDF的datafrmae写入hbase*  
 **val** newDF: DataFrame = result.select(**$"memberId"**.as(**"userId"**),predict2Tag(**'predict**).as(**"tagIds"**))  
 newDF.show()  
 *//get 'test', '13822725'  
 /\*  
 +---------+------+  
| userId|tagIds|  
+---------+------+  
| 13822725| 40|  
| 13823083| 39|  
|138230919| 39|  
| 13823681| 40|  
| 4033473| 39|  
| 13822841| 39|  
| 13823153| 39|  
| 13823431| 40|  
| 4033348| 39|  
| 4033483| 40|  
| 4033575| 39|  
| 4034191| 39|  
| 4034923| 40|  
| 13823077| 39|  
|138230937| 40|  
| 4034761| 39|  
| 4035131| 40|  
| 13822847| 39|  
|138230911| 39|  
| 4034221| 39|  
+---------+------+  
 \*/* newDF*//返回dataframe写入hbase的test中*

## 4.3标签模型完整代码

**package** cn.itcast.up.ml  
  
  
**import** cn.itcast.up.base.BaseModel  
**import** org.apache.spark.ml.clustering.{KMeans, KMeansModel}  
**import** org.apache.spark.ml.feature.VectorAssembler  
**import** org.apache.spark.rdd.RDD  
**import** org.apache.spark.sql.\_  
  
**import** scala.collection.immutable  
  
*/\*\*  
 \* Author itcast  
 \* Date 2019/11/2 15:45  
 \* Desc  
 \*/***object** RFMModel **extends** BaseModel{  
 **def** main(args: Array[String]): Unit = {  
 execute()  
 }  
 */\*\*  
 \* 获取标签id(即模型id,该方法应该在编写不同模型时进行实现)  
 \** **@return** *\*/* **override def** getTagID(): Int = 37  
  
 */\*\*  
 \* 开始计算  
 \** **@param fiveDF** *MySQL中的5级规则 id,rule  
 \** **@param hbaseDF** *根据selectFields查询出来的HBase中的数据  
 \** **@return** *userid,tagIds  
 \*/* **override def** compute(fiveDF: DataFrame, hbaseDF: DataFrame): DataFrame = {  
 *//fiveDF.show(10)  
 //fiveDF.printSchema()  
//  
 //hbaseDF.show(10)  
 // hbaseDF.printSchema()  
 /\*  
+---+----+  
| id|rule|  
+---+----+  
| 38| 1|  
| 39| 2|  
| 40| 3|  
| 41| 4|  
| 42| 5|  
| 43| 6|  
| 44| 7|  
+---+----+  
  
root  
 |-- id: long (nullable = false)  
 |-- rule: string (nullable = true)  
  
+---------+-------------------+-----------+----------+  
| memberId| orderSn|orderAmount|finishTime|  
+---------+-------------------+-----------+----------+  
| 13823431| ts\_792756751164275| 2479.45|1564415022|  
| 4035167| D14090106121770839| 2449.00|1565687310|  
| 4035291| D14090112394810659| 1099.42|1564681801|  
| 4035041| fx\_787749561729045| 1999.00|1565799378|  
| 13823285| D14092120154435903| 2488.00|1565062072|  
| 4034219| D14092120155620305| 3449.00|1563601306|  
|138230939|top\_810791455519102| 1649.00|1565509622|  
| 4035083| D14092120161884409| 7.00|1565731851|  
|138230935| D14092120162313538| 1299.00|1565382991|  
| 13823231| D14092120162378713| 499.00|1565677650|  
+---------+-------------------+-----------+----------+  
only showing top 10 rows  
  
root  
 |-- memberId: string (nullable = true)  
 |-- orderSn: string (nullable = true)  
 |-- orderAmount: string (nullable = true)  
 |-- finishTime: string (nullable = true)  
 \*/  
  
 //0.定义常量字符串,避免后续拼写错误* **val** recencyStr = **"recency"  
 val** frequencyStr = **"frequency"  
 val** monetaryStr = **"monetary"  
 val** featureStr = **"feature"  
 val** predictStr = **"predict"** *//https://blog.csdn.net/liam08/article/details/79663018* **import** org.apache.spark.sql.functions.\_  
 **import** *spark*.implicits.\_  
  
 *//1.统计每个用户的RFM  
 //Rencency:最近一次消费,最后一次订单距今时间  
 //Frequency:消费频率,订单总数量  
 //Monetary:消费金额,订单总金额  
  
 //max("finishTime"))求每个用户的最近一次的订单时间  
 //from\_unixtime()将字符串时间戳转为时间对象  
 //datediff(结束时间,开始时间)获取两个时间的时间差--天数* **val** rencencyAggColumn: Column = functions.*datediff*(*date\_sub*(*current\_timestamp*(),60),*from\_unixtime*(*max*(**"finishTime"**))) as recencyStr  
 **val** frequencyAggColumn: Column = functions.*count*(**"orderSn"**) as frequencyStr  
 **val** monetaryAggColumn: Column = functions.*sum*(**"orderAmount"**) as monetaryStr  
  
 **val** tempDF: DataFrame = hbaseDF  
 .groupBy(**"memberId"**)  
 .agg(rencencyAggColumn, frequencyAggColumn, monetaryAggColumn)  
 *//tempDF.show(10,false)  
  
 /\*  
+---------+-------+---------+------------------+  
|memberId |recency|frequency|monetary |  
+---------+-------+---------+------------------+  
|13822725 |17 |116 |179298.34 |  
|13823083 |17 |132 |233524.17 |  
|138230919|17 |125 |240061.56999999998|  
|13823681 |17 |108 |169746.1 |  
|4033473 |17 |142 |251930.92 |  
|13822841 |17 |113 |205931.91 |  
|13823153 |17 |133 |250698.57 |  
|13823431 |17 |122 |180858.22 |  
|4033348 |17 |145 |240173.78999999998|  
|4033483 |17 |110 |157811.09999999998|  
+---------+-------+---------+------------------+  
only showing top 10 rows  
 \*/  
  
 //2.数据打分(相当于归一化)  
 //R: 1-3天=5分，4-6天=4分，7-9天=3分，10-15天=2分，大于16天=1分  
 //F: ≥200=5分，150-199=4分，100-149=3分，50-99=2分，1-49=1分  
 //M: ≥20w=5分，10-19w=4分，5-9w=3分，1-4w=2分，<1w=1分* **val** recencyScore: Column = functions.*when*((*col*(recencyStr) >= 1) && (*col*(recencyStr) <= 3), 5)  
 .when((*col*(recencyStr) >= 4) && (*col*(recencyStr) <= 6), 4)  
 .when((*col*(recencyStr) >= 7) && (*col*(recencyStr) <= 9), 3)  
 .when((*col*(recencyStr) >= 10) && (*col*(recencyStr) <= 15), 2)  
 .when(*col*(recencyStr) >= 16, 1)  
 .as(recencyStr)  
  
 **val** frequencyScore: Column = functions.*when*(*col*(frequencyStr) >= 200, 5)  
 .when((*col*(frequencyStr) >= 150) && (*col*(frequencyStr) <= 199), 4)  
 .when((*col*(frequencyStr) >= 100) && (*col*(frequencyStr) <= 149), 3)  
 .when((*col*(frequencyStr) >= 50) && (*col*(frequencyStr) <= 99), 2)  
 .when((*col*(frequencyStr) >= 1) && (*col*(frequencyStr) <= 49), 1)  
 .as(frequencyStr)  
  
 **val** monetaryScore: Column = functions.*when*(*col*(monetaryStr) >= 200000, 5)  
 .when(*col*(monetaryStr).between(100000, 199999), 4)  
 .when(*col*(monetaryStr).between(50000, 99999), 3)  
 .when(*col*(monetaryStr).between(10000, 49999), 2)  
 .when(*col*(monetaryStr) <= 9999, 1)  
 .as(monetaryStr)  
  
  
 **val** RFMScoreDF: DataFrame = tempDF.select(**$"memberId"**,recencyScore,frequencyScore,monetaryScore)  
 *//RFMScoreDF.show(10,false)  
 /\*  
+---------+-------+---------+--------+  
|memberId |recency|frequency|monetary|  
+---------+-------+---------+--------+  
|13822725 |1 |3 |4 |  
|13823083 |1 |3 |5 |  
|138230919|1 |3 |5 |  
|13823681 |1 |3 |4 |  
|4033473 |1 |3 |5 |  
|13822841 |1 |3 |5 |  
|13823153 |1 |3 |5 |  
|13823431 |1 |3 |4 |  
|4033348 |1 |3 |5 |  
|4033483 |1 |3 |4 |  
+---------+-------+---------+--------+  
 \*/  
 //再使用SparkMLlib中的归一化工具进行数据缩放到[0,1]之间也可以  
  
  
 //3.我们在使用机器学习算法的时候要的是向量,所以先对特征进行向量化表示  
 //vectorAssembler可以将多列数据转换成一列特征的向量表示(特征向量)* **val** vectorAssembler: VectorAssembler = **new** VectorAssembler()  
 .setInputCols(*Array*(recencyStr, frequencyStr, monetaryStr))  
 .setOutputCol(featureStr)  
 **val** VectorDF: DataFrame = vectorAssembler.transform(RFMScoreDF)  
 *//VectorDF.show(10,false)  
/\*  
+---------+-------+---------+--------+-------------+  
|memberId |recency|frequency|monetary|feature |  
+---------+-------+---------+--------+-------------+  
|13822725 |1 |3 |4 |[1.0,3.0,4.0]|  
|13823083 |1 |3 |5 |[1.0,3.0,5.0]|  
|138230919|1 |3 |5 |[1.0,3.0,5.0]|  
|13823681 |1 |3 |4 |[1.0,3.0,4.0]|  
|4033473 |1 |3 |5 |[1.0,3.0,5.0]|  
|13822841 |1 |3 |5 |[1.0,3.0,5.0]|  
|13823153 |1 |3 |5 |[1.0,3.0,5.0]|  
|13823431 |1 |3 |4 |[1.0,3.0,4.0]|  
|4033348 |1 |3 |5 |[1.0,3.0,5.0]|  
|4033483 |1 |3 |4 |[1.0,3.0,4.0]|  
+---------+-------+---------+--------+-------------+  
 \*/  
  
 //4.聚类  
 //训练* **val** model: KMeansModel = **new** KMeans()  
 .setFeaturesCol(featureStr)  
 .setPredictionCol(predictStr)  
 .setK(7) *//实际业务运营要求分为7类* .setMaxIter(10) *//最大迭代次数* .setSeed(10) *//可重复的随子种子* .fit(VectorDF)  
 *//预测* **val** result: DataFrame = model.transform(VectorDF)  
 *//result.show(10,false)  
 /\*  
 +---------+-------+---------+--------+-------------+-------+  
|memberId |recency|frequency|monetary|feature |predict|  
+---------+-------+---------+--------+-------------+-------+  
|13822725 |1 |3 |4 |[1.0,3.0,4.0]|1 |  
|13823083 |1 |3 |5 |[1.0,3.0,5.0]|0 |  
|138230919|1 |3 |5 |[1.0,3.0,5.0]|0 |  
|13823681 |1 |3 |4 |[1.0,3.0,4.0]|1 |  
|4033473 |1 |3 |5 |[1.0,3.0,5.0]|0 |  
|13822841 |1 |3 |5 |[1.0,3.0,5.0]|0 |  
|13823153 |1 |3 |5 |[1.0,3.0,5.0]|0 |  
|13823431 |1 |3 |4 |[1.0,3.0,4.0]|1 |  
|4033348 |1 |3 |5 |[1.0,3.0,5.0]|0 |  
|4033483 |1 |3 |4 |[1.0,3.0,4.0]|1 |  
+---------+-------+---------+--------+-------------+-------+  
only showing top 10 rows  
 \*/  
  
 //5.测试时看下聚类效果* **val** ds: Dataset[Row] = result  
 .groupBy(predictStr)  
 .agg(*max*(*col*(recencyStr) + *col*(frequencyStr) + *col*(monetaryStr)), *min*(*col*(recencyStr) + *col*(frequencyStr) + *col*(monetaryStr)))  
 .sort(*col*(predictStr).asc)  
 *//ds.show()  
 /\*  
 +-------+---------------------------------------+---------------------------------------+  
|predict|max(((recency + frequency) + monetary))|min(((recency + frequency) + monetary))|  
+-------+---------------------------------------+---------------------------------------+  
| 0| 9| 9|  
| 1| 8| 8|  
| 2| 3| 3|  
| 3| 7| 7|  
| 4| 11| 10|  
| 5| 5| 4|  
| 6| 8| 7|  
+-------+---------------------------------------+---------------------------------------+  
 \*/  
  
 //6.获取聚类编号和聚类中心  
 //接下来我们要对聚类之后的用户打Tag,不能仅仅根据聚类的序号打Tag  
 //因为聚类序号并不能代表客户的价值等级,和我们5级规则并不对应  
 //所以我们应该求出每个聚类的聚类中心,排序,聚类中心的值越大说明该用户群体价值越高  
  
 //model.clusterCenters.indices获取所有聚类中心的索引编号  
 //model.clusterCenters(i)根据索引取聚类中心  
 //model.clusterCenters(i).toArray.sum求该聚类中心的RFM的和  
 //[(聚类索引/编号, 聚类中心的RFM的和)]* **val** indexAndRFM: immutable.IndexedSeq[(Int, Double)] = **for**(i <- model.clusterCenters.indices) **yield** (i,model.clusterCenters(i).toArray.sum)  
  
 **val** tuples: immutable.IndexedSeq[(Int, Double)] = model.clusterCenters.indices.map(i=>(i,model.clusterCenters(i).toArray.sum))  
  
 *//indexAndRFM.foreach(println)  
 /\*  
(0,9.0)  
(1,8.0)  
(2,3.0)  
(3,7.0)  
(4,10.038461538461538)  
(5,4.333333333333333)  
(6,7.5)  
  
 \*/  
 //println("==================")  
 //根据聚类中心的RFM的和排好序的[(聚类索引/编号, 聚类中心的RFM的和)]* **val** sortedIndexAndRFM: immutable.IndexedSeq[(Int, Double)] = indexAndRFM.sortBy(\_.\_2).reverse  
 *//sortedIndexAndRFM.foreach(println)  
/\*  
(4,10.038461538461538)  
(0,9.0)  
(1,8.0)  
(6,7.5)  
(3,7.0)  
(5,4.333333333333333)  
(2,3.0)  
 \*/  
 //7.将上面的排好序的聚类编号和聚类中心与5级规则进行对应  
 /\*  
5级规则fiveDF:  
+---+----+  
| id|rule|  
+---+----+  
| 38| 1|  
| 39| 2|  
| 40| 3|  
| 41| 4|  
| 42| 5|  
| 43| 6|  
| 44| 7|  
+---+----+  
 \*/  
 //目标:  
 /\*  
+-------+---  
predict| id|  
+------+----  
| 4| 38|  
| 0| 39|  
| 1| 40|  
| 6| 41|  
| 3| 42|  
| 5| 43|  
| 2| 44|  
+--------+--  
 \*/* **val** indexAndRFMDS: Dataset[(Int, Double)] = sortedIndexAndRFM.toDS()  
 **val** fiveDS: Dataset[(Long, String)] = fiveDF.as[(Long,String)]  
 **val** tempRDD: RDD[((Int, Double), (Long, String))] = indexAndRFMDS.*rdd*.repartition(1).zip(fiveDS.*rdd*.repartition(1))  
 *//tempRDD.collect().foreach(println)  
/\*  
((4,10.038461538461538),(38,1))  
((0,9.0),(39,2))  
((1,8.0),(40,3))  
((6,7.5),(41,4))  
((3,7.0),(42,5))  
((5,4.333333333333333),(43,6))  
((2,3.0),(44,7))  
 \*/* **val** ruleDF: DataFrame = tempRDD.map(t=>(t.\_1.\_1,t.\_2.\_1)).toDF(**"predict"**,**"tagIds"**)  
 *//ruleDF.show()  
/\*  
+-------+------+  
|predict|tagIds|  
+-------+------+  
| 4| 38|  
| 0| 39|  
| 1| 40|  
| 6| 41|  
| 3| 42|  
| 5| 43|  
| 2| 44|  
+-------+------+  
 \*/* **val** ruleMap: collection.Map[Int, Long] = ruleDF.as[(Int,Long)].*rdd*.collectAsMap()  
  
 **val** predict2Tag = *udf*((predict:Int)=>{  
 ruleMap(predict)  
 })  
  
 **val** newDF: DataFrame = result.select(**$"memberId"**.as(**"userId"**),predict2Tag(**'predict**).as(**"tagIds"**))  
 newDF.show()  
 *//get 'test', '13822725'  
 /\*  
 +---------+------+  
| userId|tagIds|  
+---------+------+  
| 13822725| 40|  
| 13823083| 39|  
|138230919| 39|  
| 13823681| 40|  
| 4033473| 39|  
| 13822841| 39|  
| 13823153| 39|  
| 13823431| 40|  
| 4033348| 39|  
| 4033483| 40|  
| 4033575| 39|  
| 4034191| 39|  
| 4034923| 40|  
| 13823077| 39|  
|138230937| 40|  
| 4034761| 39|  
| 4035131| 40|  
| 13822847| 39|  
|138230911| 39|  
| 4034221| 39|  
+---------+------+  
 \*/* newDF  
 }  
}

## 4.4标签模型调优完整代码如下

#### 调优1

*//* ***TODO：模型调优方式一：特征值归一化，使用最小最大值归一化****// val scalerModel: MinMaxScalerModel = new MinMaxScaler()  
// .setInputCol("features")  
// .setOutputCol("scaledFeatures")  
// .setMin(0.0).setMax(1.0)  
// .fit(vecDF)  
// val rfmFeaturesDF: DataFrame = scalerModel.transform(vecDF)*

#### 调优2

*//* ***TODO：模型调优方式二：调整算法超参数 -> MaxIter 最大迭代次数, 使用训练验证模式完成****// val maxIters: Array[Int] = Array(5, 10)  
// val models: Array[(Double, KMeansModel, Int)] = maxIters.map { maxIter  
// =>  
// // 5. 使用KMeans算法应用数据训练模式  
// val kMeans: KMeans = new KMeans()  
// .setFeaturesCol("features")  
// .setPredictionCol("prediction")  
// .setK(7) // 设置聚类的类簇个数  
// .setMaxIter(maxIter)  
// .setSeed(31) // 实际项目中，需要设置值  
// // 6. 训练模式  
// val model: KMeansModel = kMeans.fit(vecDF)  
// // 模型评估指标WSSSE  
// val ssse = model.computeCost(vecDF)  
// // 返回三元组  
// (ssse, model, maxIter)  
// }  
//models.foreach(println)  
// 获取最佳模型  
// val (\_, bestModel, \_) = models.minBy(tuple => tuple.\_1)  
// val KMenasModel=bestModel*

#### 调优3

*//* ***TODO：模型调优方式三：调整算法超参数 -> K值选择 肘部法, 使用训练验证模式完成****// val ks: Array[Int] = Array(2, 3, 4)  
// ks.foreach(  
// cluster => {  
// val kmeans: KMeans = new KMeans().setFeaturesCol("features").setPredictionCol("predictions")  
// .setK(cluster)  
// val kmeanModel: KMeansModel = kmeans.fit(vecDF)  
// // \* 5-预测分析  
// val testResult: DataFrame = kmeanModel.transform(vecDF)  
// println("wssse，when cluster=", cluster, kmeanModel.computeCost(testResult)) //(wssse,2.5189564869286865)  
// println(kmeanModel.clusterCenters.mkString(","))  
// }  
// )*

#### 调优4

*//* ***TODO：模型调优方式三：同时调整算法超参数 ->k和迭代次数*val** ks: Array[Int] = *Array*(2, 3, 4)  
**val** iters: Array[Int] = *Array*(2, 3, 4)  
ks.foreach(  
 cluster => {  
 iters.foreach(  
 iter => {  
 **val** kmeans: KMeans = **new** KMeans().setFeaturesCol(featureStr).setPredictionCol(**"predictions"**)  
 .setK(cluster).setMaxIter(iter)  
 **val** kmeanModel: KMeansModel = kmeans.fit(vecDF)  
 *// \* 5-预测分析* **val** testResult: DataFrame = kmeanModel.transform(vecDF)  
 *println*(**"wssse，when cluster="**, cluster,iter, kmeanModel.computeCost(testResult)) *//(wssse,2.5189564869286865)  
 println*(kmeanModel.clusterCenters.mkString(**","**))  
 }  
 )

#### 完成代码

**package** cn.itcast.up.mltest  
  
**import** cn.itcast.up.base.BaseModel  
**import** org.apache.spark.ml.clustering.{KMeans, KMeansModel}  
**import** org.apache.spark.ml.feature.{MinMaxScaler, MinMaxScalerModel, VectorAssembler}  
**import** org.apache.spark.rdd.RDD  
**import** org.apache.spark.sql.expressions.UserDefinedFunction  
**import** org.apache.spark.sql.{Column, DataFrame, Dataset, functions}  
  
**import** scala.collection.immutable  
  
**object** rfm011 **extends** BaseModel {  
 **def** main(args: Array[String]): Unit = {  
 execute()  
 }  
  
 */\*\*  
 \* 获取标签id(即模型id,该方法应该在编写不同模型时进行实现)  
 \*  
 \** **@return** *\*/* **override def** getTagID(): Int = 37  
  
 */\*\*  
 \* 开始计算  
 \*  
 \** **@param fiveDF** *MySQL中的5级规则 id,rule  
 \** **@param hbaseDF** *根据selectFields查询出来的HBase中的数据  
 \** **@return** *userid,tagIds  
 \*/* **override def** compute(fiveDF: DataFrame, hbaseDF: DataFrame): DataFrame = {  
 *// fiveDF.show()  
 // hbaseDF.show()  
 //0.定义常量字符串,避免后续拼写错误* **val** recencyStr = **"recency"  
 val** frequencyStr = **"frequency"  
 val** monetaryStr = **"monetary"  
 val** featureStr = **"feature"  
 val** predictStr = **"predict"** *//1.统计每个用户的RFM  
 //Rencency:最近一次消费,最后一次订单距今时间  
 //Frequency:消费频率,订单总数量  
 //Monetary:消费金额,订单总金额  
  
 //max("finishTime"))求每个用户的最近一次的订单时间  
 //from\_unixtime()将字符串时间戳转为时间对象  
 //datediff(结束时间,开始时间)获取两个时间的时间差--天数  
 // +---------+-------------------+-----------+----------+  
 // |memberId |orderSn |orderAmount|finishTime|  
 // +---------+-------------------+-----------+----------+  
 // |13823431 |ts\_792756751164275 |2479.45 |1564415022|  
 // |4035167 |D14090106121770839 |2449.00 |1565687310|* **import** org.apache.spark.sql.functions.\_  
 **val** recencyAggColumn: Column = functions.*datediff*(*date\_sub*(*current\_timestamp*(), 60), *from\_unixtime*(*max*(**"finishTime"**))) as recencyStr  
 **val** frencyAggColumns: Column = functions.*count*(**"orderSn"**) as frequencyStr  
 **val** moneyAggColumns: Column = functions.*sum*(**"orderAmount"**) as monetaryStr  
  
 **val** tempDF: DataFrame = hbaseDF  
 .groupBy(**"memberId"**)  
 .agg(  
 recencyAggColumn, frencyAggColumns, moneyAggColumns  
 )  
 tempDF.show(10, **false**)  
 *//2.数据打分(相当于归一化)  
 //R: 1-3天=5分，4-6天=4分，7-9天=3分，10-15天=2分，大于16天=1分  
 //F: ≥200=5分，150-199=4分，100-149=3分，50-99=2分，1-49=1分  
 //M: ≥20w=5分，10-19w=4分，5-9w=3分，1-4w=2分，<1w=1分* **val** recenyScore: Column = functions  
 .*when*((*col*(recencyStr) >= 1) && (*col*(recencyStr) <= 3), 5)  
 .when((*col*(recencyStr) >= 4) && (*col*(recencyStr) <= 6), 4)  
 .when((*col*(recencyStr) >= 7) && (*col*(recencyStr) <= 9), 3)  
 .when((*col*(recencyStr) >= 10) && (*col*(recencyStr) <= 15), 2)  
 .when((*col*(recencyStr) > 16), 1)  
 .as(recencyStr)  
  
 **val** frequencyScore: Column = functions.*when*(*col*(frequencyStr) >= 200, 5)  
 .when((*col*(frequencyStr) >= 150) && (*col*(frequencyStr) <= 199), 4)  
 .when((*col*(frequencyStr) >= 100) && (*col*(frequencyStr) <= 149), 3)  
 .when((*col*(frequencyStr) >= 50) && (*col*(frequencyStr) <= 99), 2)  
 .when((*col*(frequencyStr) >= 1) && (*col*(frequencyStr) <= 49), 1)  
 .as(frequencyStr)  
  
 **val** monetaryScore: Column = functions.*when*(*col*(monetaryStr) >= 200000, 5)  
 .when(*col*(monetaryStr).between(100000, 199999), 4)  
 .when(*col*(monetaryStr).between(50000, 99999), 3)  
 .when(*col*(monetaryStr).between(10000, 49999), 2)  
 .when(*col*(monetaryStr) <= 9999, 1)  
 .as(monetaryStr)  
 **import** *spark*.implicits.\_  
 **val** RFMScore: DataFrame = tempDF.select(**$"memberId"**, recenyScore, frequencyScore, monetaryScore)  
 RFMScore.show(10, **false**)  
 *//聚类算法实战* **val** vectorAssembler: VectorAssembler = **new** VectorAssembler().setInputCols(*Array*(recencyStr, frequencyStr, monetaryStr))  
 .setOutputCol(featureStr)  
 **val** vecDF: DataFrame = vectorAssembler.transform(RFMScore)  
  
 *//* ***TODO：模型调优方式一：特征值归一化，使用最小最大值归一化*** *// val scalerModel: MinMaxScalerModel = new MinMaxScaler()  
 // .setInputCol("features")  
 // .setOutputCol("scaledFeatures")  
 // .setMin(0.0).setMax(1.0)  
 // .fit(vecDF)  
 // val rfmFeaturesDF: DataFrame = scalerModel.transform(vecDF)  
  
 //* ***TODO：模型调优方式二：调整算法超参数 -> MaxIter 最大迭代次数, 使用训练验证模式完成*** *// val maxIters: Array[Int] = Array(5, 10)  
 // val models: Array[(Double, KMeansModel, Int)] = maxIters.map { maxIter  
 // =>  
 // // 5. 使用KMeans算法应用数据训练模式  
 // val kMeans: KMeans = new KMeans()  
 // .setFeaturesCol("features")  
 // .setPredictionCol("prediction")  
 // .setK(7) // 设置聚类的类簇个数  
 // .setMaxIter(maxIter)  
 // .setSeed(31) // 实际项目中，需要设置值  
 // // 6. 训练模式  
 // val model: KMeansModel = kMeans.fit(vecDF)  
 // // 模型评估指标WSSSE  
 // val ssse = model.computeCost(vecDF)  
 // // 返回三元组  
 // (ssse, model, maxIter)  
 // }  
 //models.foreach(println)  
 // 获取最佳模型  
 // val (\_, bestModel, \_) = models.minBy(tuple => tuple.\_1)  
 // val KMenasModel=bestModel  
 //* ***TODO：模型调优方式三：调整算法超参数 -> K值选择 最大迭代次数, 使用训练验证模式完成*** *// val ks: Array[Int] = Array(2, 3, 4)  
 // ks.foreach(  
 // cluster => {  
 // val kmeans: KMeans = new KMeans().setFeaturesCol("features").setPredictionCol("predictions")  
 // .setK(cluster)  
 // val kmeanModel: KMeansModel = kmeans.fit(vecDF)  
 // // \* 5-预测分析  
 // val testResult: DataFrame = kmeanModel.transform(vecDF)  
 // println("wssse，when cluster=", cluster, kmeanModel.computeCost(testResult)) //(wssse,2.5189564869286865)  
 // println(kmeanModel.clusterCenters.mkString(","))  
 // }  
 // )  
 //* ***TODO：模型调优方式三：同时调整算法超参数 ->k和迭代次数* val** ks: Array[Int] = *Array*(2, 3, 4)  
 **val** iters: Array[Int] = *Array*(2, 3, 4)  
 ks.foreach(  
 cluster => {  
 iters.foreach(  
 iter => {  
 **val** kmeans: KMeans = **new** KMeans().setFeaturesCol(featureStr).setPredictionCol(**"predictions"**)  
 .setK(cluster).setMaxIter(iter)  
 **val** kmeanModel: KMeansModel = kmeans.fit(vecDF)  
 *// \* 5-预测分析* **val** testResult: DataFrame = kmeanModel.transform(vecDF)  
 *println*(**"wssse，when cluster="**, cluster,iter, kmeanModel.computeCost(testResult)) *//(wssse,2.5189564869286865)  
 println*(kmeanModel.clusterCenters.mkString(**","**))  
 }  
 )  
 }  
 )  
*// //使用最终模型预测  
// val kMeans: KMeans = new KMeans()  
// .setK(7)  
// .setFeaturesCol(featureStr)  
// .setMaxIter(10)  
// .setPredictionCol(predictStr)  
// val model: KMeansModel = kMeans.fit(vecDF)  
// val resultDF: DataFrame = model.transform(vecDF)  
// println(model.computeCost(vecDF))  
// resultDF.show(10, false)  
// //结合规则--获取聚类中心  
// val indexAndRFM: immutable.IndexedSeq[(Int, Double)] = model.clusterCenters.indices.map(x => (x, model.clusterCenters(x).toArray.sum))  
// println("================================")  
// indexAndRFM.foreach(println(\_))  
// //根据聚类中心的RFM的和[(聚类索引/编号, 聚类中心的RFM的和)]sortBy排序  
// val sortIndexAndRFM: immutable.IndexedSeq[(Int, Double)] = indexAndRFM.sortBy(\_.\_2).reverse  
// val DS1: Dataset[(Int, Double)] = sortIndexAndRFM.toDS()  
// val fiveDS: Dataset[(Long, String)] = fiveDF.as[(Long, String)]  
// val tempRDD: RDD[((Int, Double), (Long, String))] = DS1.rdd.repartition(1).zip(fiveDS.rdd.repartition(1))  
// tempRDD.collect().foreach(println(\_))  
//  
// val ruleDF: DataFrame = tempRDD.map(x => (x.\_1.\_1, x.\_2.\_1)).toDF("predict", "tagIds")  
// //写入Hbase中  
// ////将ruleDF转化为map类型，使用rdd的collectAsMap操作，在使用udf转换，比如预测为predict=4代表的是38号标签  
// val ruleMap: collection.Map[Long, String] = ruleDF.as[(Long, String)].rdd.collectAsMap()  
//  
// val predict2tag: UserDefinedFunction = udf((predict: Int) => {  
// ruleMap(predict)  
// })  
// val newDF: DataFrame = resultDF.select($"memberId".as("userId"), predict2tag('predict).as("tagIds"))  
// newDF.show(10, false)* **null** }  
}

## 4.5模型调优方法

机器学习中算法模型调优，主要两个方面考虑：特征数据和算法超参数。

1）、特征数据

特征值选择、个数

特征值转换：正则化、标准化及归一化

2）、算法超参数

每个算法都有不同的超参数，每个超参数的不同值都会影响最终训练的模型

3）、注意：

注意模型的过拟合overfitting和欠拟合underfitting

### （1）特征归一化

使用最小最大值归一化MinMaxScaler 特征数据，演示代码如下：

// TODO：模型调优方式一：特征值归一化，使用最小最大值归一化

val scalerModel: MinMaxScalerModel = new MinMaxScaler()

.setInputCol("features")

.setOutputCol("scaledFeatures")

.setMin(0.0).setMax(1.0)

.fit(rfmRawFeaturesDF)

val rfmFeaturesDF: DataFrame =

scalerModel.transform(rfmRawFeaturesDF)

### （2）调整超参数

在SparkMLlib中提供针对模型调优的方法： TrainValidationSplit 和

CrossValidator，需要准备：数据、算法和评估器，通过网格参数ParamGid 封装算法不同

参数的不同值，文档说明如下：

文档：http://spark.apache.org/docs/1 2.2.0/ml-tuning.html

但是聚类算法KMeans没有提供转换模型评估器Evaluator ，所以自己编写代码完成不同超参数训练模型，获取最佳模型代码，具体如下所示：

// TODO：模型调优方式二：调整算法超参数 -> MaxIter 最大迭代次数, 使用训练验证模

式完成

val maxIters: Array[Int] = Array(5, 10)

val models: Array[(Double, KMeansModel, Int)] = maxIters.map{ maxIter

=>

// 5. 使用KMeans算法应用数据训练模式

val kMeans: KMeans = new KMeans()

.setFeaturesCol("features")

.setPredictionCol("prediction")

.setK(7) // 设置聚类的类簇个数

.setMaxIter(maxIter)

.setSeed(31) // 实际项目中，需要设置值

// 6. 训练模式

val model: KMeansModel = kMeans.fit(rfmFeaturesDF)

// 模型评估指标WSSSE

val ssse = model.computeCost(rfmFeaturesDF)

// 返回三元组

(ssse, model, maxIter)

}

//models.foreach(println)

// 获取最佳模型

val (\_, bestModel, \_) = models.minBy(tuple => tuple.\_1)

## 4.6保存和加载模型

实际开发标签时，如果是挖掘类型标签，往往先开发算法模型，获取最佳模型以后，将模型保存，进行构建标签时，先加载保存（比如从HDFS文件系统），再依据算法模型预测获取标签值，依据标签属性值打标签。

在SparkMLlib中每个算法模型都提供保存save和加载load模型的方法，如下截图所示：

/\*\*

\* Trait for classes that provide `MLWriter`.

\*/

@Since("1.6.0")

trait MLWritable {

/\*\*

\* Returns an `MLWriter` instance for this ML instance.

\*/

@Since("1.6.0")

def write: MLWriter

/\*\*

\* Saves this ML instance to the input path, a shortcut of

`write.save(path)`.

\*/

@Since("1.6.0")

@throws[IOException]("If the input path already exists but overwrite

is not enabled.")

def save(path: String): Unit = write.save(path)

}

//////////////////////////////////////////////////////////////////////

///

/\*\*

\* Trait for objects that provide `MLReader`.

\*

\* @tparam T ML instance type

\*/

@Since("1.6.0")

trait MLReadable[T] {

/\*\*

\* Returns an `MLReader` instance for this class.

\*/

@Since("1.6.0")

def read: MLReader[T]

/\*\*

\* Reads an ML instance from the input path, a shortcut of

`read.load(path)`.

\*

\* @note Implementing classes should override this to be Javafriendly.

\*/

@Since("1.6.0")

def load(path: String): T = read.load(path)

}

结合RfmModel标签模型开发，先判断模型是否存在，如果存在加载模型；如不存在训练模型并保存，代码：

// TODO: 获取RFM标签对应KMeans模型

var kMeansModel: KMeansModel = null

// 模型保存路径

val modelPath = ModelConfig.MODEL\_BASE\_PATH +

s"/${this.getClass.getSimpleName.toLowerCase}"

if(HdfsUtils.exists(modelPath)){

kMeansModel = KMeansModel.load(modelPath)

}else{

// TODO：模型调优方式二：调整算法超参数 -> MaxIter 最大迭代次数,

使用训练验证模式完成

val maxIters: Array[Int] = Array(5, 10)

val models: Array[(Double, KMeansModel, Int)] =

maxIters.map{ maxIter =>

// 5. 使用KMeans算法应用数据训练模式

val kMeans: KMeans = new KMeans()

.setFeaturesCol("features")

.setPredictionCol("prediction")

.setK(7) // 设置聚类的类簇个数

.setMaxIter(maxIter)

.setSeed(31) // 实际项目中，需要设置值

// 6. 训练模式

val model: KMeansModel = kMeans.fit(rfmFeaturesDF)

// 模型评估指标WSSSE

val ssse = model.computeCost(rfmFeaturesDF)

// 返回三元组

(ssse, model, maxIter)

}

//models.foreach(println)

// 获取最佳模型

val (\_, bestModel, \_) = models.minBy(tuple => tuple.\_1)

// TODO：当训练完成以后，获取最佳模型，需要保存

bestModel.save(modelPath)

// 返回

kMeansModel = bestModel

}

其中需要判断HDFS上模型保存路径是否存在，工具类HdfsUtils ，代码如下：

package cn.itcast.tags.utils

import cn.itcast.tags.models.ModelConfig

import org.apache.hadoop.conf.Configuration

import org.apache.hadoop.fs.{FileSystem, Path}

/\*\*

\* 操作HDFS文件系统工具类

\*/

object HdfsUtils {

/\*\*

\* 判断HDFS上路径是否存在

\* @param path

\* @return

\*/

def exists(path: String): Boolean = {

// a. 读取HDFS配置信息

val conf = new Configuration()

conf.set("fs.defaultFS", ModelConfig.DEFAULT\_FS)

// b. 获取文件系统

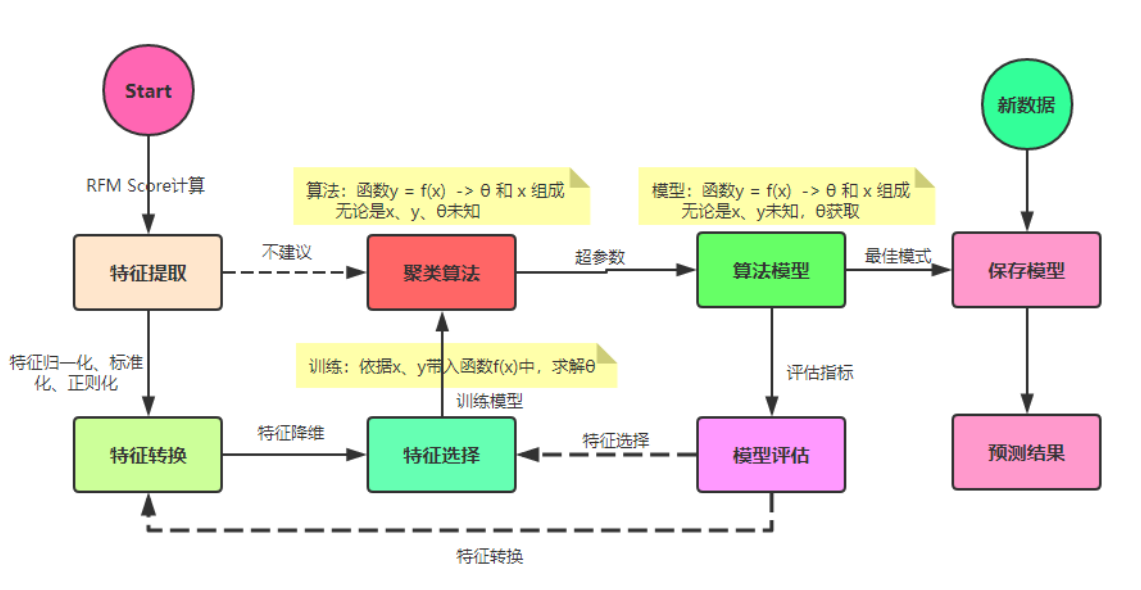
val dfs = FileSystem.get(conf)

// c. 判断路径是否存在

dfs.exists(new Path(path))

}

}



# 5.总结