**大数据电商用户画像项目**

**第六部分◎教案**

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

# 第6章：挖掘类标签之机器学习理论基础

# 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。

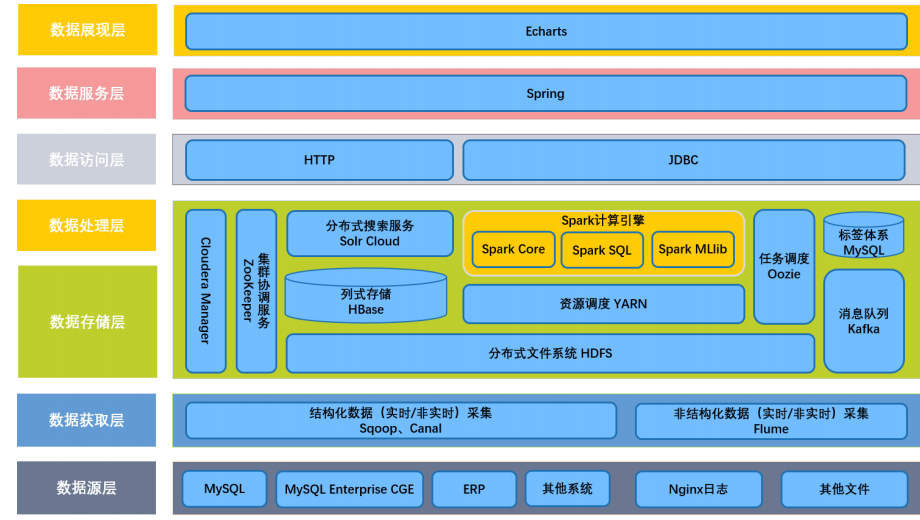
4.标签查询

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。

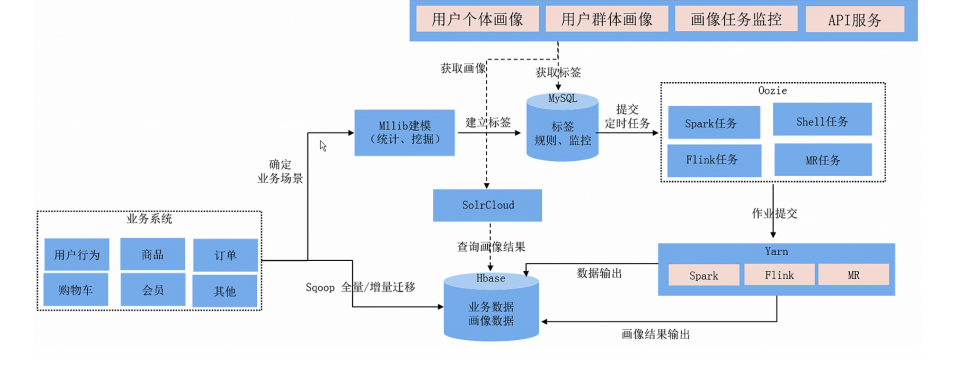
5.系统设置

系统设置模块主要包括用户管理、权限管理。

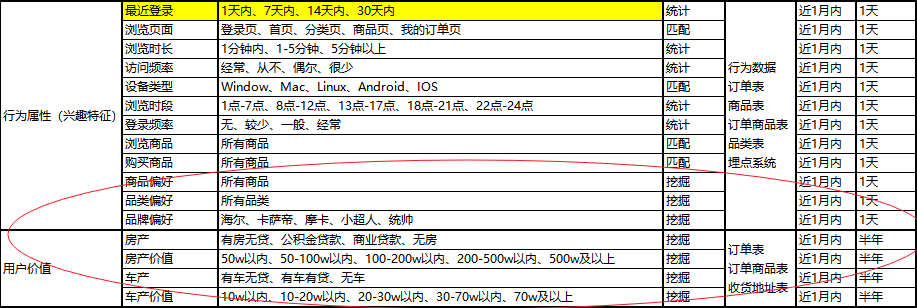
## 技术架构



## 项目功能架构



# 挖掘类标签的举例



## 1.1 SparkMllib的决策树的原理及实战

### 1.1.1决策树的引入：

有的同学可能在大学学习过一门课程叫《数据结构》，里面有一个重要的结构就是“树”，和现实生活中的树一样，树的主要由四部分[树根](https://baike.so.com/doc/6831821-7049020.html" \t "https://baike.so.com/doc/_blank)、[树干](https://baike.so.com/doc/6002536-6215513.html" \t "https://baike.so.com/doc/_blank)、[树枝](https://baike.so.com/doc/1378276-1456997.html" \t "https://baike.so.com/doc/_blank)、[树叶](https://baike.so.com/doc/5408467-5646454.html" \t "https://baike.so.com/doc/_blank)组成，今天的决策树也是一种树结构，大家学习的时候可以想象现实生活中的树来来理解。

决策树算法是一种监督学习算法，英文是Decision tree。

决策树思想的来源非常朴素，试想每个人的大脑都有类似于if-else这样的逻辑判断，这其中的if表示的是条件，if之后的then就是一种选择或决策。程序设计中的条件分支结构就是if-then结构，最早的决策树就是利用这类结构分割数据的一种分类学习方法。

比如：你母亲要给你介绍男朋友，是这么来对话的：

女儿：多大年纪了？

母亲：26。

女儿：长的帅不帅？

母亲：挺帅的。

女儿：收入高不？

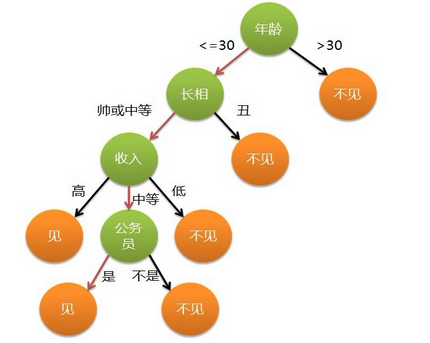
母亲：不算很高，中等情况。

女儿：是公务员不？

母亲：是，在税务局上班呢。

女儿：那好，我去见见。

于是你在脑袋里面就有了下面这张图：



作为女孩的你在决策过程就是典型的分类树决策。相当于通过年龄、长相、收入和是否公务员对将男人分为两个类别：见和不见。

**有了上面这个认识，我们就通过之前比较熟悉电商例子开始了解决策树算法。**

### 1.2.1.电商业务场景：

在商业的数据挖掘中，不同的消费行为顾客特征的提炼和表述极为重要。

我们模拟前期大家做过的电商平台项目中，模拟顾客及其消费行为数据共1024条，包括用户是否购买某种产品（0购买，1不购买），年龄（青年0，中年1，老年2），收入（高0，中1，低2），学生（是1，否0），信誉（良0，优1）。

我们模拟收集了如表1-1的部分用户购买数据，建立了一张统一的调查表，统计几个月的销售数据。我们这里模拟通过从电商平台中对下表中潜在的客户进行分析，并根据得到的一些特征用于销售人员制定销售策略等工作。

总结为两个问题：

(1) 如何对客户进行分类？

(2) 如何根据分类的依据，给出销售人员的指导意见？

数据如下：

表 1-1 用户购买统计表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 统计用户计数 | 年龄 | 收入 | 学生 | 信誉 | 是否购买 |
| 64 | 青 | 高 | 否 | 良 | 不买 |
| 64 | 青 | 高 | 否 | 优 | 不买 |
| 128 | 中 | 高 | 否 | 良 | 买 |
| 60 | 老 | 中 | 否 | 良 | 买 |
| 64 | 老 | 低 | 是 | 良 | 买 |
| 64 | 老 | 低 | 是 | 优 | 不买 |
| 64 | 中 | 低 | 是 | 优 | 买 |
| 128 | 青 | 中 | 否 | 良 | 不买 |
| 64 | 青 | 低 | 是 | 良 | 买 |
| 132 | 老 | 中 | 是 | 良 | 买 |
| 64 | 青 | 中 | 是 | 优 | 买 |
| 32 | 中 | 中 | 否 | 优 | 买 |
| 32 | 中 | 高 | 是 | 良 | 买 |
| 64 | 老 | 中 | 否 | 优 | 不买 |

**分析问题：**

1.首先搞清楚数据集的内容

2.数据集：行--样本(15)、列---特征个数(4)

3.特征：(4)

年龄---青年、中年、老年

收入----高、中、低

学生----是 or 否

信誉----优 良

4.类标签：(1)

最后一列：买 or 不买 （二分类问题）

5.训练集(训练模型) + 测试集(测试模型的优劣)

**回答问题：**

1.如何对客户进行分类？

答：根据数据集中收集的用户的特征信息

2.如何根据分类依据，给出销售指导意见呢？

答：给出这样的销售意见：

中年人通常会无条件购买

青年人中如果是学生一般会购买

老年人中信誉好的常常会购买

### 1.2.2基于规则的建树

1. 对表格数据的分析

从第一列中可以看到这张表格，共15行，每行表示列特征不同取值的统计人数。**第二列是年龄特征**，取值有三个：老、中、青。**第三列是收入**：取值是高、中、低。**第四列是学生**，取值有是获否。**第五列为信誉**，取值有两个，为优和良。最后一列为**销售的结果**，也就是分类标签，取值为买或不买。

1. 问题分析：

那么对于任意给定特征值的一个客户（也就是测试样例），算法需要帮助公司将这位客户归类，即预测这位客户是属于“购买”商品的一类还是“不购买”商品的一类并且给出判断依据。

1. 首先，我们根据示例**手工**实现一颗决策树。

**分析规则：（制定规则）**

a.既然是一棵树，肯定是由根节点、叶子结点和内部节点组成。

b.(根节点选取)如果从一颗空决策树开始，任意选择第一个特征就是根节点。

c.(叶子结点的判断)如果我们按照条件进行划分，若划分的到一个子集为空或子集中所有样本已经归为同一类的别标签，则该子集就是叶子结点，否则这些子集就对应于决策树中的内部节点。

d.(内部节点需要继续划分)如果是内部节点，就需要选择一个新的类别标签继续对该子集进行划分，直到所有子集为叶子节点。

**根据规则进行划分：** 根据上述树的规则我们对示例数据集进行划分。

1. 我们选年龄为根节点，该特征值取值为：老、中、青。根据年龄特征将所有样本进行划分，并作为决策树的第一层。

（1）年龄 = 青，是否购买：不买、买

（2）年龄 = 中，是否购买：买

（3）年龄=老，是否购买：不买、买

也就是当年龄为中年，是否购买都一致地变为买，此时的中年就成为决策树的叶子节点。而当年龄为青年和老年时候，是否购买有两个选择，可以继续分解。

如下表1-2：按照年龄分开数据集：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 统计用户计数 | 年龄 | 收入 | 学生 | 信誉 | 是否购买 |
| 64 | 青 | 高 | 否 | 良 | 不买 |
| 64 | 青 | 高 | 否 | 优 | 不买 |
| 128 | 青 | 中 | 否 | 良 | 不买 |
| 64 | 青 | 低 | 是 | 良 | 买 |
| 64 | 青 | 中 | 是 | 优 | 买 |
| 128 | 中 | 高 | 否 | 良 | 买 |
| 64 | 中 | 低 | 是 | 优 | 买 |
| 32 | 中 | 中 | 否 | 优 | 买 |
| 32 | 中 | 高 | 是 | 良 | 买 |
| 60 | 老 | 中 | 否 | 良 | 买 |
| 64 | 老 | 低 | 是 | 良 | 买 |
| 64 | 老 | 低 | 是 | 优 | 不买 |
| 132 | 老 | 中 | 是 | 良 | 买 |
| 64 | 老 | 中 | 否 | 优 | 不买 |

1. 接着，年龄为“青年”、“老年”需要接着划分。对数据集中“青年”选项的行数据继续选择第二个特征，如：收入，根据收入得到如下表1-3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 统计用户计数 | 年龄 | 收入 | 学生 | 信誉 | 是否购买 |
| 64 | 青 | 高 | 否 | 良 | 不买 |
| 64 | 青 | 高 | 否 | 优 | 不买 |
| 128 | 青 | 中 | 否 | 良 | 不买 |
| 64 | 青 | 中 | 是 | 优 | 买 |
| 64 | 青 | 低 | 是 | 良 | 买 |

1. 根据上表分析，高收入和低收入的特征值只有一个类别标签---买，将其作为叶子节点，然后继续划分中等以下收入的下一个特征---学生，如下表1-4：根据是学生将不会购买此商品，不是学生将会购买此商品。到此，年龄为“青年”的全部归类。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 统计用户计数 | 年龄 | 收入 | 学生 | 信誉 | 是否购买 |
| 128 | 青 | 中 | 否 | 良 | 不买 |
| 64 | 青 | 中 | 是 | 优 | 买 |

1. 青年划分完之后继续标签为老年的划分，如我们选择划分的顺序为信誉---收入---学生----计数，如此，当选择信誉作为下一级划分数据集的特征时，一次性就将信誉为优的类别标签划分为不买，信誉为良的类别标签划分为买。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 统计用户计数 | 年龄 | 收入 | 学生 | 信誉 | 是否购买 |
| 60 | 老 | 中 | 否 | 良 | 买 |
| 64 | 老 | 低 | 是 | 良 | 买 |
| 64 | 老 | 中 | 是 | 良 | 买 |
| 132 | 老 | 低 | 是 | 优 | 不买 |
| 64 | 老 | 中 | 否 | 优 | 不买 |

根据以上几部，我们可以得到一颗完整的决策树，我们为了判断方便将所有购买商品的叶子节点放在树的右侧，不购买商品的叶子结点放在树的左侧。

a.从定性的角度画出决策树：



b.从定量的角度画出决策树：（根据表格进行量化）

为什么需要从定量角度来分析呢？

答：这样会更精确的分析用户的特征信息，给出销售人员更准确的数据信息。



### 1.2.4构建决策树的三个步骤

构建决策树包括三个步骤：

**特征选择：**选取有较强分类能力的特征。

**决策树生成：**典型的算法有ID3和C4.5，它们生成决策树过程相似，ID3是采用信息增益作为特征选择度量，而C4.5采用信息增益比率。

**决策树剪枝：**剪枝原因是决策树生成算法生成的树对训练数据的预测很准确，但是对于未知数据分类很差，这就产生了过拟合的现象。

详细分析过程如下：

注意：决策树的构建直到没有特征可选或者信息增益很小时停止，这就导致了构建的决策树模型过于复杂，而复杂的模型是在训练数据上建立的，所以对于测试集往往造成分类不准确，这就是过拟合。

发生过拟合是由于决策树太复杂，解决办法是控制模型的复杂度，对于决策树来说就是简化模型，称为“剪枝”。

决策树剪枝的过程就是从已生成的决策树上裁掉一些子树或者叶节点。剪枝的目标是通过极小化决策树的整体损失函数来实现的。

下面我们从决策树目标入手，讲一下如何构建决策树模型。

决策树的目标：给定训练集，其中为输入实例，n为样本个数（），为类标记，i=1,2.....N;N为样本容量。构建决策树的目标是，根据给定的训练数据集学习一个决策树模型。

构建决策树通常是递归地选择最优特征，并根据该特征对训练数据进行分割，步骤如下：

构建根节点，所有的样本都位于根节点。

选择一个最优的特征。通过该特征将训练数据集分割成子集，确保各个子集有最好的分类，但要考虑以下两种情况：

若子集能够被“较好地”分类，则构建叶节点，并将子集划分到对应的叶节点中去。

若某个子集不能被“较好地”分类，则对该子集继续划分。

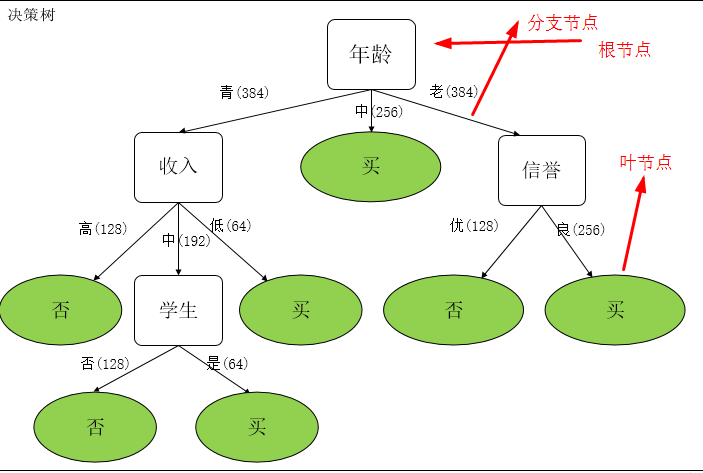
递归直到所有训练样本都被很好的分类，或没有合适的特征为止。对于是否较好的分类，我们借助“信息增益”、“信息增益率”、“基尼系数”、“分类错误率”等指标衡量。

通过上述步骤生成的决策树对训练样本有很好的分类能力，但是我们需要的是对未知样本的分类能力。因此通常需要对已生成的决策树进行剪枝，从而使得决策树有很好的泛化能力。剪枝的过程就是去掉过于细分的叶节点，从而提高泛化能力。

### 1.2.5.决策树的基本概念和算法

#### （1）什么是决策树/判定树（decision tree)?

①决策树是一个类似于流程图的树结构：其中，每个内部结点表示一个特征或属性，而每个树叶结点代表一个分类。树的最顶层是根结点。使用决策树分类时就是将实例分配到叶节点的类中。该叶节点所属的类就是该节点的分类。（通过下图理解）



② 决策树是机器学习中分类方法中的一个重要算法

③构造决策树的基本算法

* ID3算法：使用信息增益进行特征选择
* C4.5算法：使用信息增益率进行特征选择，克服了信息增益选择特征的时候偏向于特征个数较多的不足。
* CART算法：CART全称为分类回归树，既可以用于分类，也可以用于预测。利用CART构建回归树用到树的**剪枝**技术，用于防止树的过拟合。

通过以下表格，我们通过观察并得到了一颗决策树，那么我们如何根据ID3算法得到我们优化过的决策树呢？这就需要**首先引入熵的概念**。



表1 用户购买统计表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 统计用户计数 | 年龄 | 收入 | 学生 | 信誉 | 是否购买 |
| 64 | 青 | 高 | 否 | 良 | 不买 |
| 64 | 青 | 高 | 否 | 优 | 不买 |
| 128 | 中 | 高 | 否 | 良 | 买 |
| 60 | 老 | 中 | 否 | 良 | 买 |
| 64 | 老 | 低 | 是 | 良 | 买 |
| 64 | 老 | 低 | 是 | 优 | 不买 |
| 64 | 中 | 低 | 是 | 优 | 买 |
| 128 | 青 | 中 | 否 | 良 | 不买 |
| 64 | 青 | 低 | 是 | 良 | 买 |
| 132 | 老 | 中 | 是 | 良 | 买 |
| 64 | 青 | 中 | 是 | 优 | 买 |
| 32 | 中 | 中 | 否 | 优 | 买 |
| 32 | 中 | 高 | 是 | 良 | 买 |
| 64 | 老 | 中 | 否 | 优 | 不买 |

#### （2）熵（entropy）概念：

①**信息如何度量？**

 1948年，香农提出了 “信息熵(entropy)”的概念

 一条信息的信息量大小和它的不确定性有直接的关系，要搞清楚一件非常非常不确定的事情，或者是我们一无所知的事情，需要了解大量信息==>（也就是说）信息量的度量就等于不确定性的多少

（使用）比特(bit)来衡量信息的多少

例子：对一个有32支球队的世界杯争夺冠军，熵为？

其中p1,p2……p32分别是32支球队争冠的概率。如果当32支球队夺冠概率相同时，对应的熵是最大的，等于5bit。（分析如何得出5这个数值的。）

注意：实质上每个队夺冠的几率不是相等的。当概率不相等的时候算出的熵都比5小。

②**为什么要有“熵”的概念？为什么是这样一个公式？**

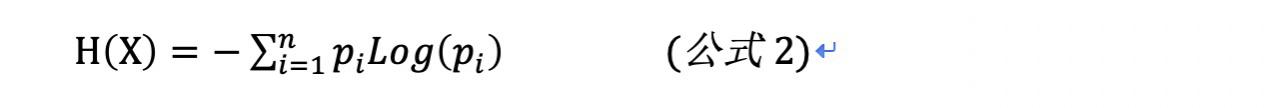
既然决策树有根节点、内部节点和叶子节点之分，那么究竟那个属性做根节点，那个属性又做下一级分类的分支节点(内部节点)?

香农在他的<<信息论>>中借用德国物理学家发明的“熵”的概念提出了“信息熵”的概念。在物理学中“熵”通常是指物体能量的分布更加均匀，而香农首先定义了“信息”的概念：信息就是对不确定性的消除。因此，“信息熵”就表示事物不确定性的度量标准，可以根据数学中的概率计算，出现的概率就大，出现的机会就多，不确定性就小（信息熵小）。

通常多个独立事件所产生的不确定性等于各自不确定性之和，也就是每个事件的发生与不发生是相互独立的，这在数学上称为可加性，而满足这样可加性的函数就是我们在高中阶段所学习过的对数函数(log函数)。由此得到第一个不确定性函数公式：



  如果对一个随机事件有n种取值：X1,X2,X3…，对应的概率p1,p2…..pn,则该事件的不确定性应该是单个取值的不确定性的期望E，也叫作信息熵，即：



上述若对数的底数取值为2，就是我们平常所说的信息单位bit(比特)。

总结：变量的不确定性越大，熵也就越大。熵越小，信息的纯度越高。

有了以上基础之后，我们就可以学习ID3算法，从而构建我们的决策树，从而达到分类目的。

### 1.2.6决策树归纳算法 （ID3）

* 1970-1980， J.Ross. Quinlan, ID3算法
* 目的：选择属性判断结点
* 引入“信息增益”：信息获取量(Information Gain)：
* Gain(A) = Info(D)-Info\_A(D)
  + 注：整个数据集信息熵与当前节点（划分节点）信息熵的差
* 通过某一个节点来作为分支节点来分类获取了多少信息

**（1）信息增益概念的含义和本质？**

**含义：**划分数据集前后信息发生的变化。

**本质:** 衡量给定属性能否更完美、没有误差地区分开训练数据集的能力。

我们使用信息增益来确定决策树分支的划分依据，它也是决策树分支上整个数据集信息熵与当前节点（划分节点）信息熵的差值，通常用Gain(A)表示：

 也就是当前节点信息熵E(A)越小，不确定性也越小，整体数据集的信息熵（I(parent)）一般是保持不变的，从而当前节点的信息增益就越大，因此通常选择信息熵小的，信息增益大的值作为根节点。

**（2）ID3算法详解：**

ID3算法起源于CLS概念学习系统，以信息熵的下降速度为选取测试属性的标准，即在每个节点选取尚未被用来划分的具有最高信息增益的属性作为划分标准，继续这个过程，直到生成决策树能完美分类训练样例。

输入：样本的集合S，属性集合A

输出：ID3决策树

* 如果所有种类的属性都处理完毕，则返回。否则，执行步骤B.
* 计算出信息增益最大的属性a，将该属性作为一个节点。如果仅凭借a就可以对样本分类，则返回。否则执行步骤C。
* 对属性a的每个可能取值v执行以下操作：

1.将v的样本作为s的一个子集Sv

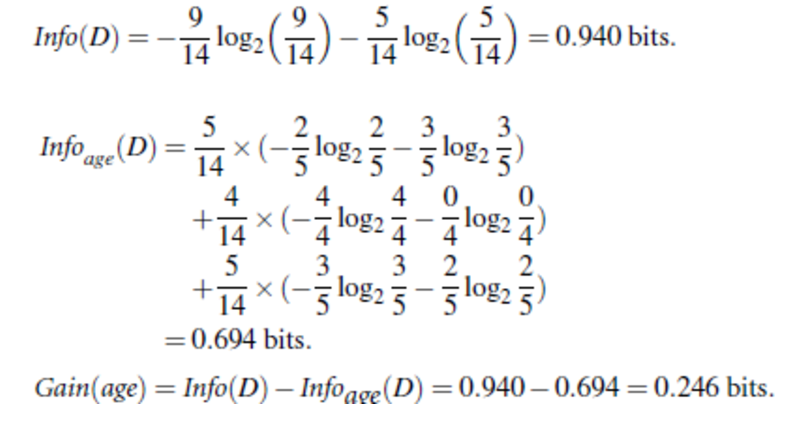
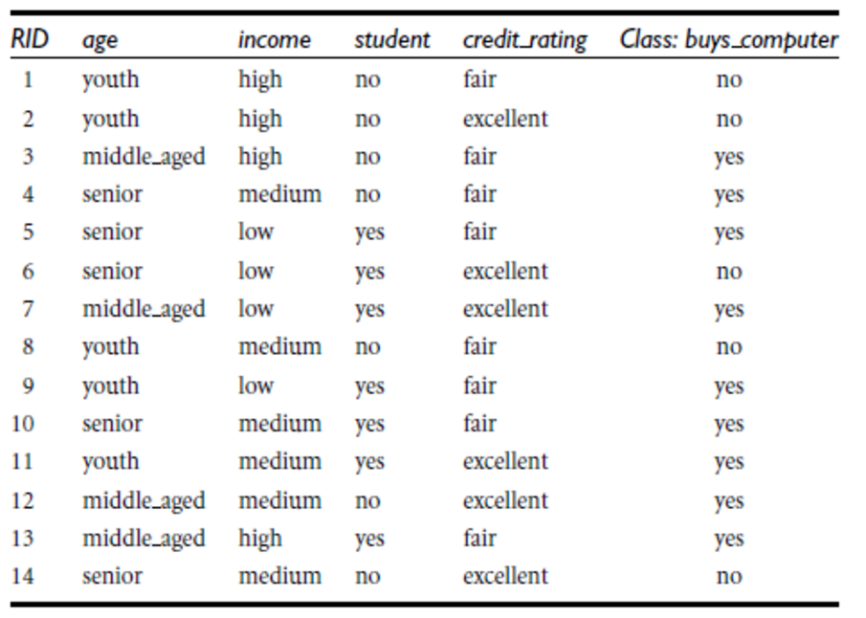
2.生成剩余属性集合AT=A-{a}

3.以样本集合Sv和属性集合AT为输入，递归执行ID3算法。

在了解了信息熵和熵的计算之后，我们通过简单例子计算这两个重要的变量。

### 1.2.7信息熵和信息增益的计算示例（ID3关键步骤）

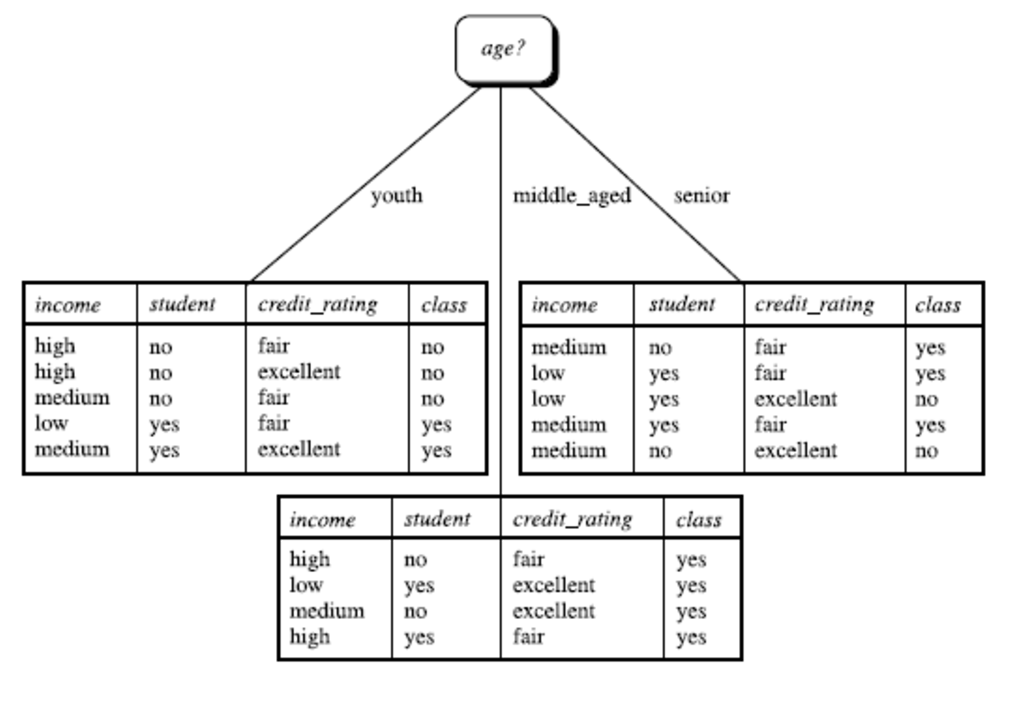
有了以上理论基础我们可以接下来通过一个小例子完整计算信息熵和信息增益，从而通过ID3算法构造决策树。



注：整个数据集信息熵与当前节点（划分节点）信息熵的差，就是信息增益

类似，Gain(income) = 0.029, Gain(student) = 0.151, Gain(credit\_rating)=0.048

所以，选择age作为第一个根节点



重复上述过程。

**算法步骤：(和前面的算法步骤理解其一即可)**

* 树以代表训练样本的单个结点开始（步骤1）。
* 如果样本都在同一个类，则该结点成为树叶，并用该类标号（步骤2 和3）。
* 否则，算法使用称为信息增益的基于熵的度量作为启发信息，选择能够最好地将样本分类的属性（步骤6）。该属性成为该结点的“测试”或“判定”属性（步骤7）。在算法的该版本中，
* 所有的属性都是分类的，即离散值。连续属性必须离散化。
* 对测试属性的每个已知的值，创建一个分枝，并据此划分样本（步骤8-10）。
* 算法使用同样的过程，递归地形成每个划分上的样本判定树。一旦一个属性出现在一个结点上，就不必在该结点的任何后代上考虑它（步骤13）。

**递归划分步骤仅当下列条件之一成立停止：**

(a) 给定结点的所有样本属于同一类（步骤2 和3）。

(b) 没有剩余属性可以用来进一步划分样本（步骤4）。在此情况下，使用多数表决（上述步骤5）。

这涉及将给定的结点转换成树叶，并用样本中的多数所在的类标记它。替换了可以存放结点样本的类分布。

(c) 分枝：test\_attribute = a ，没有样本（步骤11）。在这种情况下，以 samples 中的多数类创建一个树叶（步骤12）

定义一个最大不纯度，如果最大不纯度下降小于a时，就将该节点看做是叶子节点。(或可理解为纯度达到阈值a)

通过这个小例子大家了解了通过ID3算法构建树的过程，我们在后面还会利用熵和信息熵计算我们电商的例子，接下来我们了解下除了ID3算法，还有哪几种算法可以来构建决策树。

### 1.2.8决策树的其他优化算法：

**（1）C4.5算法**

C4.5和ID3（1986年 Quinlan提出的）算法类似，只有在以下几个点作了改进。

* 用信息增益率来选择属性，克服用信息增益选择时候偏向选择取值多属性不足。（了解）

**特征A对数据集D的信息增益：Gain(A) = Info(D)-Info\_A(D)**

**信息增益率：Gainr(A) =Gain(A) /H(A) ：其中H(A)为A的熵**

**如何理解：**

由于在分类问题困难时候，也就是在训练数据集经验熵(总体的Info(D)值)大的时候，信息增益值会偏大(Info(D)值偏大)，如果训练数据集经验熵小的时候，信息增益会偏小，这时候也可以通过信息增益率来矫正这一问题。

**1.经验熵(就是前面提到的信息熵，也就是信息量的大小问题)**

**2.在树构造过程中进行剪支。**

**3.能够完成连续属性的离散化处理。**

**4.能够对不完整数据进行处理。**

**（2）CART**

**Classification and Regression Trees (CART)**: (L. Breiman, J. Friedman, R. Olshen, C. Stone)

共同点：都是贪心算法，自上而下(Top-down approach)

区别：属性选择度量方法不同

C4.5 （gain ratio)，CART(gini index)，ID3 (Information Gain)

关于以上几点解释：

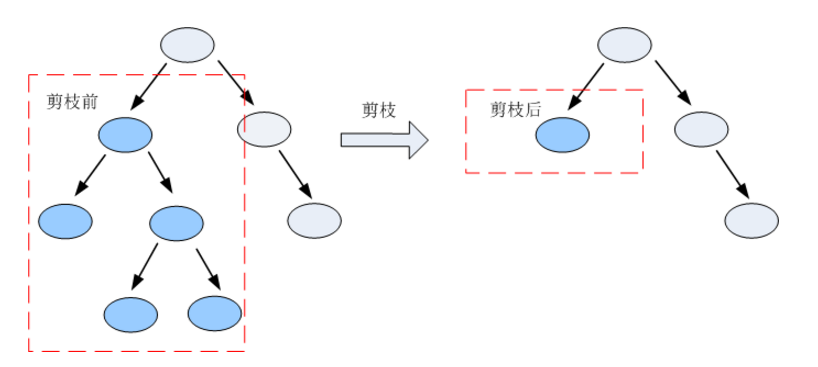
**贪心算法：**顾名思义，贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。

无论哪种方法构建树，因为树的构建过程是递归形式的，所以有可能出现树的过拟合情况，因此，我们对树过拟合情况有一种“树剪枝”技术，避免树由于过度生长而出现的过拟合现象。接下来我们讲解树的两种剪枝技术。

### 1.2.9树剪枝叶(避免overfitting过拟合)

**问题1：什么是剪枝？**

剪枝是**指将一颗子树的子节点全部删掉，利用叶子节点替换子树**(实质上是后剪枝技术)，也可以（假定当前对以root为根的子树进行剪枝）只保留根节点本身而删除所有的叶子，以下图为例：



**问题2：为什么要进行树的剪枝？**

决策树是充分考虑了所有的数据点而生成的复杂树，有可能出现过拟合的情况，决策树越复杂，过拟合的程度会越高。

考虑极端的情况：如果我们令所有的叶子节点都只含有一个数据点，那么我们能够保证所有的训练数据都能准确分类，但是很有可能得到高的预测误差，原因是将训练数据中所有的噪声数据都”准确划分”了，强化了噪声数据的作用。

而剪枝修剪分裂前后分类误差相差不大的子树，能够降低决策树的复杂度，降低过拟合出现的概率。

**关键步骤解释：**

因为决策树的构建过程是一个递归的过层，所以必须确定停止条件，否则过程将不会停止，树会不停生长。通过我们前面的例子我们可以当一个节点下面的所有记录都属于同一类，或者当所有记录属性都具有相同的值时停止，但是这样往往会使得树的节点过多，导致过度拟合的问题。

过度拟合是指直接生成的完全决策树对训练样本的特征描述的“过于精确”，无法实现对新样本进行合理的分许，所以这种情况我们构建的树不是一颗最佳的决策树。

所以，为了避免过拟合，我们引入剪枝技术。

除了剪枝技术我们还有一种解决方法：当前结点中的记录数低于一个最小阈值就停止分裂，采用多数表决的方法决定叶节点的分类。

**问题3：怎样剪枝？**

两种方案：先剪枝和后剪枝

* 先剪枝说白了就是提前结束决策树的增长，跟上述决策树停止生长的方法一样。
* 后剪枝是指在决策树生长完成之后再进行剪枝的过程。

接下来我们深入理解下这两种剪枝技术。

#### （1）先剪枝（预剪枝）

先剪枝是对决策树停止标准的修改。

在ID3算法中，节点的分割一直到节点中的实例属于同一类别的时候才停止。对于包含较少实例的节点，很有可能被分割为单一实例的节点。为了避免这种情况，我们给出一个阈值，当一个节点分割导致的最大的不纯度下降小于a时，就把该节点看作是叶子结点。该方法选择的阈值对决策树的构建有很大的影响。

（1）当阈值a选择的过大的时候，节点的不纯度依然很高就停止了分裂。此时的树由于生长不足，导致决策树过小，分类的错误率过高。因此需要调整a参数的合理范围之内的值。

（2）当阈值a选择的过小的时候，比如a接近0，节点的分割过程近似于原始的分割过程。

由此可见，预剪枝方法虽然简单，但在实际应用中对a的选择存在很大的主观性。精确的给出a的值也是相当困难。

#### （2）后剪枝

后剪枝是从一个充分生长的树中，按照自低向上的方式修剪掉多余的分支，有两种方法：

（1）用新的叶子结点替换子树，该叶子结点的类标号由子树记录中的多数类决定。

（2）用子树中最常用的分支替代子树。

通常计算前后预期分类错误率，如果修剪导致预期分类错误率变大，则放弃修剪，保留相应的各个分支，否则就将相应的节点分支修剪删除。

通常采用后剪枝技术是最小的错误剪枝(MEP)技术，即在产生一系列的经过修剪后的决策树候选之后，利用一个独立的测试数据集，对这些经过修剪之后的决策树的分类准确性进行评价，保留下预期分类错误率最小的（修剪后）决策树。

除了最小错误剪枝技术外，还有悲观错误剪枝(MEP)和代价复杂度剪枝(CCP)

#### （3）剪枝技术对比

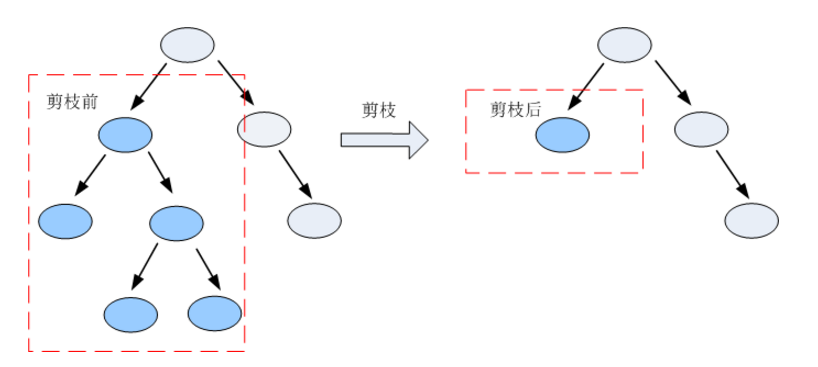
预剪枝使得决策树的很多分支没有展开，这不仅仅降低了过拟合的风险，还显著减少了决策树的训练时间开销和测试时间开销。但另一方面，有些分支的当前划分虽不能提升泛化性能，甚至会导致泛化性能降低，但在其基础上进行的后续划分却有可能导致性能的显著提高。同时，预见值决策树也带来了欠拟合的风险。

后剪枝的决策树通常比预剪枝的决策树保留了更多的分支。一般情况下，后剪枝决策树的欠拟合风险很小，泛化性能往往优于预剪枝决策树。但后剪枝过程是在生成完全决策树之后进行的，并且要自底向上地对树中所有非叶子节点进行逐一考察，因此在训练时间开销比未剪枝的决策树和预剪枝的决策树都要大得多。

**接下来对决策树算法的特点进行总结：**

### 1.2.10剪枝算法的数学原理（了解）

剪枝总体思路：



由完全树T0开始，剪枝部分节点得到T1，再次剪枝部分节点得到T2...直到仅剩树根的树Tk；在验证数据集上对这K个树分别评价，选择损失函数最小的树的树Ta。

#### （1）剪枝系数

根据原损失函数：



叶节点越多，决策树越复杂，损失越大，修正：



当时，未剪枝的决策树损失最小（全树）；

当时，单根结点的决策树损失最小（单根）

**注：参考回归问题损失函数定义理解：**



现在考虑以r为根的子树剪枝，且剪枝后只保留r本身而删掉所有的叶子：

剪枝后的损失函数：

剪枝前的损失函数：

剪枝前后要求损失函数下降或至少相等，这里取剪枝前后相等得到剪枝系数a：



#### （2）剪枝算法

对于给定的决策树T0

* 计算所有内部节点的剪枝系数a
* (a越大，表示单根，就会越安全，a越小表示未剪枝)
* 查找最小剪枝系数a的结点，剪枝得决策树Tk
* 重复以上步骤，直到决策树TK只有1个结点
* 得到决策树序列T0、T1、T2、T3..............Tk
* 使用验证样本集选择最优子树，可以使用损失函数作为评价最优子树的标准



### 1.2.11决策树算法的特点

（1） 决策树的优点：

* 直观，便于理解，小规模数据集有效
* 执行效率高，执行只需要一次构建，可反复使用

（2）决策树的缺点：

* 处理连续变量不好，较难预测连续字段
* 类别较多时，错误增加的比较快
* 对于时间序列数据需要做很多的预处理
* 可规模性一般
* 实际分类的时候只能根据一个字段进行

### 1.2.12决策树算法处理连续值

连续值的处理：由于连续属性的可取值数目不在有限，因此，不能直接根据连续属性的取值对结点进行划分。此时，我们采用连续值离散化的技术，最简单的是采用二分法（将样本的属性取值从大到小排序，找一个划分点将样本集分成两个子集，大于划分点的集合是决策树的一个分支，小于划分点的是决策树的一个分支）对连续属性进行处理，这也是C4.5算法采用的策略。

### 1.2.13电商实例ID3算法实例详解

表1-5：按照年龄分开数据集：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 统计用户计数 | 年龄 | 收入 | 学生 | 信誉 | 是否购买 |
| 64 | 青 | 高 | 否 | 良 | 不买 |
| 64 | 青 | 高 | 否 | 优 | 不买 |
| 128 | 青 | 中 | 否 | 良 | 不买 |
| 64 | 青 | 低 | 是 | 良 | 买 |
| 64 | 青 | 中 | 是 | 优 | 买 |
| 128 | 中 | 高 | 否 | 良 | 买 |
| 64 | 中 | 低 | 是 | 优 | 买 |
| 32 | 中 | 中 | 否 | 优 | 买 |
| 32 | 中 | 高 | 是 | 良 | 买 |
| 60 | 老 | 中 | 否 | 良 | 买 |
| 64 | 老 | 低 | 是 | 良 | 买 |
| 64 | 老 | 低 | 是 | 优 | 不买 |
| 132 | 老 | 中 | 是 | 良 | 买 |
| 64 | 老 | 中 | 否 | 优 | 不买 |

#### （1）ID3算法步骤演示

有了上面的基础之后，通过最上面的例子实现ID3算法的生成过程。

①**计算对给定样本分类所需的信息熵：**

类别标签S被分为两类：买or不买。其中：

S1（买）=640；S2（不买）=384；S=S1+S2=1024;

S1的概率p1=640/1024=0.625; S2的概率p2=384/1024=0.375

**I(S1,S2)=I(640,384)=-p1log(p1) -p2log(p2)=0.9544**

②计算每个特征的信息熵及信息增益

1. 计算“年龄”特征的信息熵：年龄分为三组：青年（0）、中年（1）、老年（2）
   1. 青年占总样本的概率

**P(0)**=384/1024=0.375

S1(买)=128 p1=128/384

S2(不买)=256 p2=256/384

**I1(S1,S2)=I(128,256)=-p1log(p1) -p2log(p2)=0.9183**

* 1. 中年占总样本的概率

**P(1)**=256/1024=0.25

S1(买)=256 p1=256/256=1

S2(不买)=0 p2=0/384=0

**I2(S1,S2)=I(256,0)=-p1log(p1) -p2log(p2)=0**

* 1. 老年占总样本的概率

**P(2)**=384/1024=0.375

S1(买)=257 p1=257/384

S2(不买)=127 p2=127/384

**I3(S1,S2)=I(257,127)=-p1log(p1) -p2log(p2)=0.9157**

则年龄的信息熵为：

E（年龄）= **P(0)\* I1(S1,S2)+ P(1)\* I2(S1,S2)+ P(2)\* I3(S1,S2)**

**=0.375\*0.9183+0.25\*0+0375\*0.9157 =0.6877**

则年龄的信息增益为：

G（年龄）=**0.9544-0.6877=0.2667**

1. 计算“学生”特征的信息熵

E（学生）=0.7811

G（学生）=0.9544-0.7811=0.1733

1. 计算“收入”“特征的熵

E（收入）=0.9361

G（收入）=0.9544-0.9361=0.0183

1. 计算“信誉”“特征的熵

E（信誉）=0.9048

G（信誉）=0.9544-0.9048=0.0496

③从所有列的特征中选出信息增益最大的那个作为根节点或内部节点

* 根据上述信息增益的大小，选定年龄列G=0.2667来划分数据集。

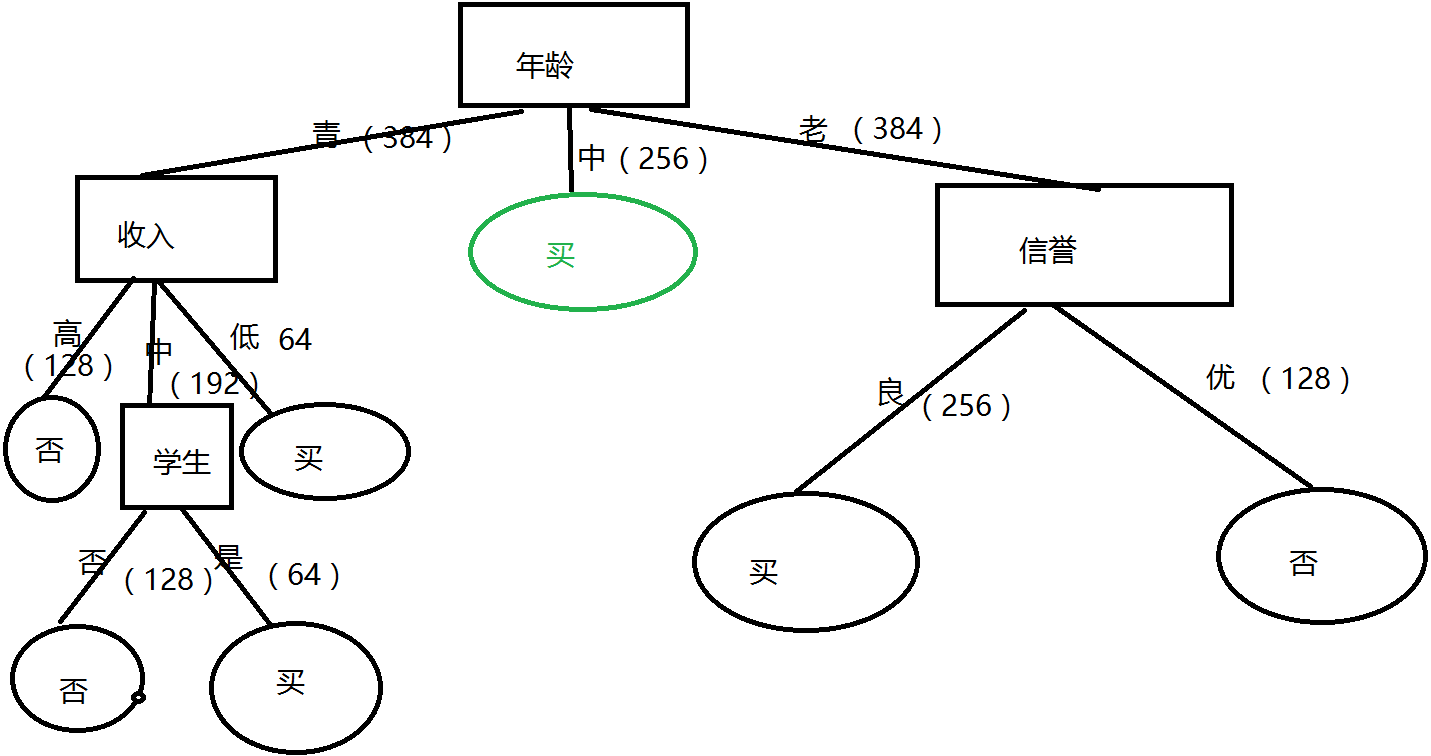
④根据节点的不同取值将数据集拆分为若干子集，删除当前的特征列，在计算剩余列的特征信息熵。如果有信息增益，就重复第二步直至划分结束。

* 首次划分后，青年和老年内含有多个标签，所以可以继续划分，中年选项就剩下一个标签就称作为叶子结点。

⑤划分结束标志为：子集中只有一个类别标签，停止划分。

**根据信息增益，选择出根节点，依次划分得到一颗决策树，大家可以对比下这颗决策树和之前的决策树的差别在哪里？**

**基于规则的决策树如下：**



#### （2）基于模型的决策树

按照上述结果产生一个决策树如下图：

**（解释）**这张图明显是比我们根据规则手画的决策树优化了，根据年龄判断了直接就看是否为学生就可以达到分类目的。无需根据年龄🡪收入🡪学生-🡪是否购买来判断。

**（总结）**根据信息熵和信息增益得到的决策树和之前利用自己制定规则得到的决策树，有明显的不同，因此，利用信息熵和信息增益可以得到分类效果更优的决策树。



## 1.2.项目案例：构建SparkML决策树分类模型项目实战

（数据加载-数据探索-数据预处理-建立模型-模型评估-模型优化-项目总结）

### 1.2.1SparkMllib决策树算法参数详解

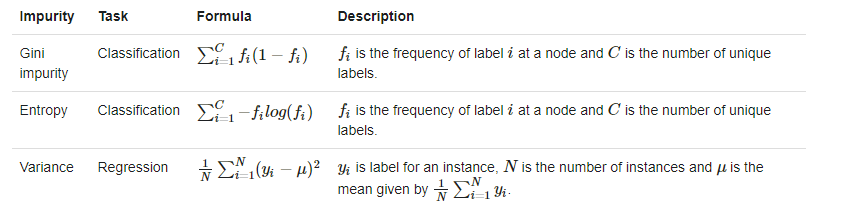
[决策树](http://en.wikipedia.org/wiki/Decision_tree_learning)及其集合是分类和回归的机器学习任务的流行方法。决策树被广泛使用，因为它们易于解释，处理分类特征，扩展到多类分类设置，不需要特征缩放，并且能够捕获非线性和特征交互。诸如随机森林和增强的树集合算法是分类和回归任务的最佳表现者。

spark.ml实现支持使用连续和分类特征的二进制和多类分类以及回归的决策树。该实现按行分区数据，允许分布式训练数百万甚至数十亿实例。

用户可以在[MLlib决策树指南中](http://spark.apache.org/docs/2.2.0/mllib-decision-tree.html)找到有关决策树算法的更多信息。此API与[原始MLlib Decision Tree API](http://spark.apache.org/docs/2.2.0/mllib-decision-tree.html)之间的主要区别是：

* + 支持ML管道
  + 决策树的分离用于分类与回归
  + 使用DataFrame元数据来区分连续和分类功能
  + 决策树的Pipelines API提供了比原始API更多的功能。

特别是，对于分类，用户可以获得每个类的预测概率（也称为类条件概率）; 对于回归，用户可以获得有偏差的预测样本方差。



#### （1）节点分裂

**连续特征**

对于单机实现中的小型数据集，每个连续特征的拆分候选项通常是该特征的唯一值。某些实现对特征值进行排序，然后使用有序的唯一值作为快速树计算的分割候选。

对于大型分布式数据集，排序特征值是非常昂贵的。SparkMllib该实现通过对数据的采样部分**执行分位数计算来计算一组近似的分割候选。有序拆分创建“箱”**，并且可以使用maxBins参数指定此类箱的最大数量。

请注意，bin的数量不能大于实例样本数（由于默认值为32，因此很少见）。如果不满足条件，树算法会自动减少容器的数量。maxBins

**分类特征**

对于具有可能值（类别）的分类特征，可以提出分割候选特征。对于二进制（0/1）分类和回归，我们可以通过按平均标签对分类特征值进行排序来减少分割候选特征的数量。例如，对于具有三个类别A，B和C的一个分类特征的二元分类问题，其标记1的对应比例为0.2,0.6和0.4，分类特征按A，C，B排序。两个分割候选者是A | C，B和A，C | B，其中| 表示分裂。

在多类分类中，尽可能使用所有可能的分割。当大于参数时，我们使用类似于二进制分类和回归的方法的（启发式）方法。该分类的特征值由杂质排序，并将得到的分裂的候选人被认为。

#### （2）停止规则

当满足以下条件之一时，在节点处停止递归树构造：

1. 节点深度等于maxDepth训练参数。
2. 没有分割候选特征导致信息增益大于minInfoGain。
3. 没有拆分候选特征进一步生成子节点，每个子节点至少具有minInstancesPerNode训练实例。

#### （3）参数

我们通过讨论各种参数，包括一些使用决策树的指南。下面大致按照重要性递减的顺序列出参数。

这些参数描述了您要解决的问题和数据集。它们应该被指定，不需要调整。

* algo：决策树的类型，Classification或者Regression。
* numClasses：类的数量（Classification仅用于）。
* categoricalFeaturesInfo：**指定哪些特征是分类的，以及每个特征可以采用的分类值。**这是从特征索引到特征（类别数）的映射。
  + 例如，Map(0 -> 2, 4 -> 10)指定特征0是二进制（取值0或1），该特征4有10个类别（值{0, 1, ..., 9}）。需要注意的是特征是从0下标开始的：特征0和4是第1个元素和实例的特征向量的第5元素。
  + 请注意，不必指定categoricalFeaturesInfo。该算法仍将运行并可获得合理的结果。但是，如果正确指定了分类功能，性能会更好。

#### （4）停止标准

这些参数确定树何时停止构建（添加新节点）。调整这些参数时，请小心验证测试数据，以避免过度拟合。

* + maxDepth：限定决策树的最大可能深度。但由于其它终止条件或者是被剪枝的缘故，最终的决策树的深度可能要比maxDepth小。树的最大深度。更深的树木有更好的表现（更高的准确度），但它们的训练更耗时，而且更容易过度拟合。
  + minInstancesPerNode：如果某个节点的样本数量小于该值，则该节点将不再被分叉。（设置阈值）

实际上要想获得一个适当的阈值是相当困难的。高阈值可能导致过分简化的树，而低阈值可能简化不够对于要进一步拆分的节点，其每个子节点必须至少接收此数量的训练实例。这通常与[RandomForest一起](http://spark.apache.org/docs/2.2.0/api/scala/index.html" \l "org.apache.spark.mllib.tree.RandomForest$)使用，因为它们通常比单个树更深地训练。

* + minInfoGain：最小信息增益（设置阈值），小于该值将不带继续分叉;对于要进一步拆分的节点，拆分必须至少改善这一点（就信息增益而言）。

#### （5）可调参数

可以调整这些参数。在调整时要小心验证保持的测试数据，以避免过度拟合。

* maxBins：离散连续特征时使用的分箱数。
  + 增加maxBins允许算法考虑更多分割候选者并进行细粒度分割决策。但是，它也增加了计算和通信。
  + 请注意，maxBins参数必须至少为任何分类功能的最大类别数。M
* maxMemoryInMB：用于收集足够统计信息的内存量。
  + 保守地选择默认值为256 MB，以允许决策算法在大多数情况下工作。maxMemoryInMB通过允许更少的数据传递，增加可以导致更快的训练（如果存储器可用）。但是，maxMemoryInMB由于每次迭代的通信量可以成比例，因此增长的回报可能会增加maxMemoryInMB。
  + 实现细节：为了加快处理速度，决策树算法收集有关要拆分的节点组的统计信息（而不是一次收集1个节点）。可以在一个组中处理的节点数由内存要求（每个功能不同）确定。该maxMemoryInMB参数指定每个工作程序可用于这些统计信息的内存限制（以兆字节为单位）。
* subsamplingRate：用于学习决策树的训练数据的采样比率。此参数与树的训练集合（使用[RandomForest](http://spark.apache.org/docs/2.2.0/api/scala/index.html" \l "org.apache.spark.mllib.tree.RandomForest$)和[GradientBoostedTrees](http://spark.apache.org/docs/2.2.0/api/scala/index.html" \l "org.apache.spark.mllib.tree.GradientBoostedTrees)）最相关，其中对原始数据进行子采样可能很有用。对于训练单个决策树，该参数不太有用，因为训练实例的数量通常不是主要约束。
* impurity：用于在候选分割之间进行选择的杂质度量（如上所述）。此度量必须与algo参数匹配。

#### （6）缓存和检查点

MLlib 1.2增加了几个功能，可以扩展到更大（更深）的树和树集合。当maxDepth设置为大时，打开节点ID缓存和检查点可能很有用。当设置为大时，这些参数对[RandomForest](http://spark.apache.org/docs/2.2.0/api/scala/index.html" \l "org.apache.spark.mllib.tree.RandomForest$)也很有用numTrees。

* useNodeIdCache：如果将此设置为true，则算法将避免在每次迭代时将当前模型（树或树）传递给执行程序。

这对于深树（加速计算工作）和大型随机森林（减少每次迭代的通信）非常有用。

实现细节：默认情况下，算法将当前模型传递给执行程序，以便执行程序可以将训练实例与树节点进行匹配。打开此设置后，算法将缓存此信息。

节点ID缓存生成一系列RDD（每次迭代1次）。这种长谱系可能会导致性能问题，但检查点中间RDD可以缓解这些问题。请注意，检查点仅在useNodeIdCache设置为true 时适用。

* checkpointDir：检查点节点ID缓存RDD的目录。
* checkpointInterval：检查点节点ID缓存RDD的频率。设置太低会导致写入HDFS带来额外开销; 如果执行程序失败并且需要重新计算RDD，则将此设置得太高会导致问题。

#### （7）缩放

计算在训练实例的数量，特征的数量和maxBins参数中近似线性地缩放。通常在特征数量和数量上近似线性缩放maxBins。

实现的算法读取稀疏和密集数据。但是，它没有针对稀疏输入进行优化。

#### （8）输入和输出

这里列出输入和输出（预测）列类型。所有输出列都是可选的; 要排除输出列，请将其对应的Param设置为空字符串。

**Input Columns**

|  |  |  |  |
| --- | --- | --- | --- |
| Param name | Type(s) | Default | Description |
| labelCol | Double | "label" | Label to predict |
| featuresCol | Vector | "features" | Feature vector |

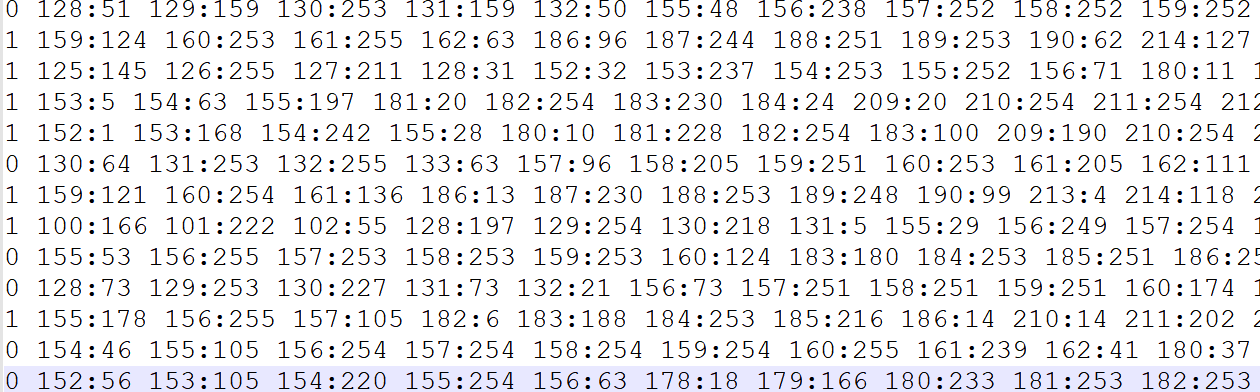
**Output Columns**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Param name | Type(s) | Default | Description | Notes |
| predictionCol | Double | "prediction" | Predicted label |  |
| rawPredictionCol | Vector | "rawPrediction" | Vector of length # classes, with the counts of training instance labels at the tree node which makes the prediction | Classification only |
| probabilityCol | Vector | "probability" | Vector of length # classes equal to rawPrediction normalized to a multinomial distribution | Classification only |
| varianceCol | Double |  | The biased sample variance of prediction | Regression only |

### 1.2.2SparkMl决策树案例实战libsvm数据集

**数据：**

**Label index1：value1 index2：value2 index3：value3**



**代码：**

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.classification.DecisionTreeClassificationModel

import org.apache.spark.ml.classification.DecisionTreeClassifier

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}

// Load the data stored in LIBSVM format as a DataFrame.

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

// Index labels, adding metadata to the label column.

// Fit on whole dataset to include all labels in index.

val labelIndexer = new StringIndexer()

.setInputCol("label")

.setOutputCol("indexedLabel")

.fit(data)

// Automatically identify categorical features, and index them.

val featureIndexer = new VectorIndexer()

.setInputCol("features")

.setOutputCol("indexedFeatures")

.setMaxCategories(4) // features with > 4 distinct values are treated as continuous.

.fit(data)

// Split the data into training and test sets (30% held out for testing).

val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))

// Train a DecisionTree model.

val dt = new DecisionTreeClassifier()

.setLabelCol("indexedLabel")

.setFeaturesCol("indexedFeatures")

// Convert indexed labels back to original labels.

val labelConverter = new IndexToString()

.setInputCol("prediction")

.setOutputCol("predictedLabel")

.setLabels(labelIndexer.labels)

// Chain indexers and tree in a Pipeline.

val pipeline = new Pipeline()

.setStages(Array(labelIndexer, featureIndexer, dt, labelConverter))

// Train model. This also runs the indexers.

val model = pipeline.fit(trainingData)

// Make predictions.

val predictions = model.transform(testData)

// Select example rows to display.

predictions.select("predictedLabel", "label", "features").show(5)

// Select (prediction, true label) and compute test error.

val evaluator = new MulticlassClassificationEvaluator()

.setLabelCol("indexedLabel")

.setPredictionCol("prediction")

.setMetricName("accuracy")

val accuracy = evaluator.evaluate(predictions)

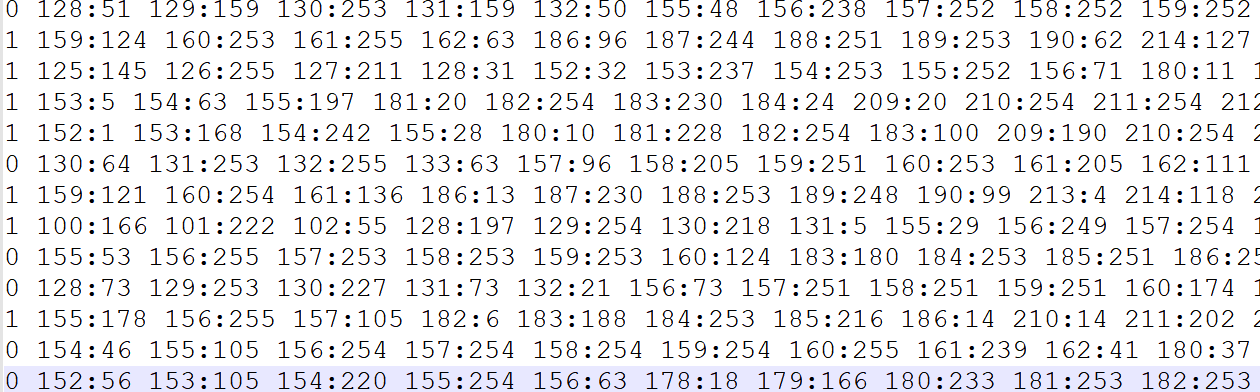
println("Test Error = " + (1.0 - accuracy))

val treeModel = model.stages(2).asInstanceOf[DecisionTreeClassificationModel]

println("Learned classification tree model:\n" + treeModel.toDebugString)

### 1.2.3SparkMllib决策树案例实战libsvm数据

**数据：**



**代码：**

import org.apache.spark.mllib.tree.DecisionTree

import org.apache.spark.mllib.tree.model.DecisionTreeModel

import org.apache.spark.mllib.util.MLUtils

// Load and parse the data file.

val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

// Split the data into training and test sets (30% held out for testing)

val splits = data.randomSplit(Array(0.7, 0.3))

val (trainingData, testData) = (splits(0), splits(1))

// Train a DecisionTree model.

// Empty categoricalFeaturesInfo indicates all features are continuous.

val numClasses = 2

val categoricalFeaturesInfo = Map[Int, Int]()

val impurity = "gini"

val maxDepth = 5

val maxBins = 32

val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,

impurity, maxDepth, maxBins)

// Evaluate model on test instances and compute test error

val labelAndPreds = testData.map { point =>

val prediction = model.predict(point.features)

(point.label, prediction)

}

val testErr = labelAndPreds.filter(r => r.\_1 != r.\_2).count().toDouble / testData.count()

println("Test Error = " + testErr)

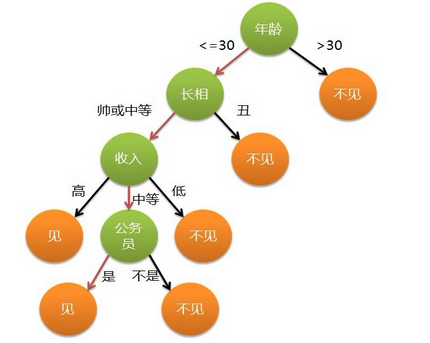
println("Learned classification tree model:\n" + model.toDebugString)

// Save and load model

model.save(sc, "target/tmp/myDecisionTreeClassificationModel")

val sameModel = DecisionTreeModel.load(sc, "target/tmp/myDecisionTreeClassificationModel")

### 1.2.4决策树案例实战相亲数据集



#### （1）数据集

字段说明：是否见面, 年龄，是否帅，收入(1 高 2 中等 0 少)，是否公务员

0,32 1 1 0

0,25 1 2 0

1,29 1 2 1

1,24 1 1 0

0,31 1 1 0

1,35 1 2 1

0,30 0 1 0

0,31 1 1 0

1,30 1 2 1

1,21 1 1 0

0,21 1 2 0

1,21 1 2 1

0,29 0 2 1

0,29 1 0 1

0,29 0 2 1

1,30 1 1 0

测试数据集

0,32 1 2 0

1,27 1 1 1

1,29 1 1 0

1,25 1 2 1

0,23 0 2 1

#### （2）Spark-MLlib决策树应用代码

代码如下：

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

import org.apache.spark.mllib.feature.HashingTF

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

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

import org.apache.spark.mllib.tree.DecisionTree

import org.apache.spark.mllib.util.MLUtils

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

/\*\*

\* **决策树分类**

\*/

object TreeDemo {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("DecisionTree").setMaster("local")

val sc = new SparkContext(conf)

Logger.getRootLogger.setLevel(Level.WARN)

//训练数据

val data1 = sc.textFile("data/Tree1.txt")

//测试数据

val data2 = sc.textFile("data/Tree2.txt")

//转换成向量

val tree1 = data1.map { line =>

val parts = line.split(',')

LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(\_.toDouble)))

}

val tree2 = data2.map { line =>

val parts = line.split(',')

LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(\_.toDouble)))

}

//赋值

val (trainingData, testData) = (tree1, tree2)

//分类

val numClasses = 2

val categoricalFeaturesInfo = Map[Int, Int]()

val impurity = "gini"

//最大深度

val maxDepth = 5

//最大分支

val maxBins = 32

//模型训练

val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,

impurity, maxDepth, maxBins)

//模型预测

val labelAndPreds = testData.map { point =>

val prediction = model.predict(point.features)

(point.label, prediction)

}

//测试值与真实值对比

val print\_predict = labelAndPreds.take(15)

println("label" + "\t" + "prediction")

for (i <- 0 to print\_predict.length - 1) {

println(print\_predict(i).\_1 + "\t" + print\_predict(i).\_2)

}

//树的错误率

val testErr = labelAndPreds.filter(r => r.\_1 != r.\_2).count.toDouble / testData.count()

println("Test Error = " + testErr)

//打印树的判断值

println("Learned classification tree model:\n" + model.toDebugString)

}

}

3、测试结果：

label prediction

0.0 0.0

1.0 1.0

1.0 1.0

1.0 1.0

0.0 0.0

Test Error = 0.0

Learned classification tree model:

可见真实值与预测值一致，Error为0

打印决策树的分支值，这里最大深度为 5 ，对应的树结构：

Learned classification tree model:

DecisionTreeModel classifier of depth 4 with 11 nodes

If (feature 1 <= 0.0)

Predict: 0.0

Else (feature 1 > 0.0)

If (feature 3 <= 0.0)

If (feature 0 <= 30.0)

If (feature 2 <= 1.0)

Predict: 1.0

Else (feature 2 > 1.0)

Predict: 0.0

Else (feature 0 > 30.0)

Predict: 0.0

Else (feature 3 > 0.0)

If (feature 2 <= 0.0)

Predict: 0.0

Else (feature 2 > 0.0)

Predict: 1.0

### 1.2.5SparkMllib决策树案例实战Iris数据集

数据描述：

iris以鸢尾花的特征作为数据来源，常用在分类操作中。该数据集由3种不同类型的鸢尾花的50个样本数据构成。其中的一个种类与另外两个种类是线性可分离的，后两个种类是非线性可分离的。

该数据集包含了5个属性：

Sepal.Length（花萼长度），单位是cm;

Sepal.Width（花萼宽度），单位是cm;

Petal.Length（花瓣长度），单位是cm;

Petal.Width（花瓣宽度），单位是cm;

Class|Species种类：Iris Setosa（山鸢尾）、Iris Versicolour（杂色鸢尾），以及Iris Virginica（维吉尼亚鸢尾）。

数据：

sepal\_length sepal\_width petal\_length petal\_width class

5.1 3.5 1.4 0.2 Iris-setosa

4.9 3 1.4 0.2 Iris-setosa

4.7 3.2 1.3 0.2 Iris-setosa

4.6 3.1 1.5 0.2 Iris-setosa

5 3.6 1.4 0.2 Iris-setosa

5.4 3.9 1.7 0.4 Iris-setosa

基于sparkmllib实现方案：

package com.test.spark.sparkmllib\_helloworld.iris\_ml\_1

import org.apache.spark.mllib.tree.DecisionTree

import org.apache.spark.mllib.tree.model.DecisionTreeModel

import org.apache.spark.mllib.util.MLUtils

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext.\_

import org.apache.spark.SparkConf

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

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

import org.apache.spark.rdd.RDD

object DecisionTree2 {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("iris.Example01").setMaster("local")

val sc = new SparkContext(conf)

val lines = sc.textFile("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/sparkmllib\_helloworld/iris\_ml\_1/iris.csv")

val rdd: RDD[LabeledPoint] = lines filter (line => !line.isEmpty) map { line =>

val v = line.split(",")

LabeledPoint(v(4) match {

case "Iris-setosa" => 0.0

case "Iris-versicolor" => 1.0

case "Iris-virginica" => 2.0

case \_ => -1.0 // should never occur

}, Vectors.dense(v(0).toDouble, v(1).toDouble, v(2).toDouble, v(3).toDouble))

}

val Array(data, test) = rdd.randomSplit(Array(0.8, 0.2))

// Count

println("#data = " + data.count)

println("#test = " + test.count)

// Train a DecisionTree model.

// Empty categoricalFeaturesInfo indicates all features are continuous.

val numClasses = 3

val categoricalFeaturesInfo = Map[Int, Int]()

val impurity = "gini"

val maxDepth = 5

val maxBins = 32

val model = DecisionTree.trainClassifier(data, numClasses, categoricalFeaturesInfo,

impurity, maxDepth, maxBins)

test.foreach {

element =>

println(element.label + "=" + model.predict(element.features))

}

println("Learned classification tree model:\n" + model.toDebugString)

// Evaluate model on test instances and compute test error

val labelAndPreds = test.map { point =>

val prediction = model.predict(point.features)

(point.label, prediction)

}

//println(labelAndPreds)

val testErr = labelAndPreds.filter(r => r.\_1 != r.\_2).count().toDouble / test.count()

println(s"Test Error = $testErr")

println(s"Learned classification tree model:\n ${model.toDebugString}")

// Save and load model

model.save(sc, "/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/sparkmllib\_helloworld/iris\_ml\_1/tmp/myDecisionTreeClassificationModel")

val sameModel = DecisionTreeModel.load(sc, "/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/sparkmllib\_helloworld/iris\_ml\_1/tmp/myDecisionTreeClassificationModel")

sc.stop()

}

}

### 1.2.6SparkMl决策树案例实战Iris数据集

数据描述：

iris以鸢尾花的特征作为数据来源，常用在分类操作中。该数据集由3种不同类型的鸢尾花的50个样本数据构成。其中的一个种类与另外两个种类是线性可分离的，后两个种类是非线性可分离的。

该数据集包含了5个属性：

Sepal.Length（花萼长度），单位是cm;

Sepal.Width（花萼宽度），单位是cm;

Petal.Length（花瓣长度），单位是cm;

Petal.Width（花瓣宽度），单位是cm;

Class|Species种类：Iris Setosa（山鸢尾）、Iris Versicolour（杂色鸢尾），以及Iris Virginica（维吉尼亚鸢尾）。

数据：

sepal\_length sepal\_width petal\_length petal\_width class

5.1 3.5 1.4 0.2 Iris-setosa

4.9 3 1.4 0.2 Iris-setosa

4.7 3.2 1.3 0.2 Iris-setosa

4.6 3.1 1.5 0.2 Iris-setosa

5 3.6 1.4 0.2 Iris-setosa

5.4 3.9 1.7 0.4 Iris-setosa

基于sparkml实现方案，代码部分：

package com.test.spark.sparkmllib\_helloworld.iris\_ml\_2

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.classification.DecisionTreeClassifier

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorAssembler}

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

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

object IrisSpark {

case class IrisData(SepalWidth: Double, SepalLength: Double, PetalLength: Double, PetalWidth: Double, Name: String)

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

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

val sc = new SparkContext(conf)

val sparkSession = SparkSession.builder

.config(conf = conf)

.appName("IrisSpark")

.getOrCreate()

import sparkSession.implicits.\_

val path = "/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/sparkmllib\_helloworld/iris\_ml\_2/iris.csv"

val data: DataFrame = sparkSession.read

.option("header","true")

.option("inferSchema", "true")

.csv(path)

data.printSchema()

val Array(train, test) = data.randomSplit(Array(.8,.2))

val featVec = new VectorAssembler()

.setInputCols(Array("SepalLength", "SepalWidth", "PetalLength", "PetalWidth"))

.setOutputCol("features")

// val trainFeatures = assembler.transform(dataOne)

// val testFeatures = assembler.transform(dataTwo)

//need LabelIndexer

val nameIndexer = new StringIndexer()

.setInputCol("Name")

.setOutputCol("nameIndex")

.fit(data)

println(nameIndexer)

val predConverter = new IndexToString()

.setInputCol("prediction")

.setOutputCol("predLabel")

.setLabels(nameIndexer.labels)

val tree = new DecisionTreeClassifier()

.setLabelCol("nameIndex")

.setFeaturesCol("features")

val pipeline = new Pipeline()

.setStages(Array(nameIndexer, featVec, tree, predConverter))

val model = pipeline.fit(train)

val preds = model.transform(test)

preds.select("Name", "predLabel").show(10)

val evaluator = new MulticlassClassificationEvaluator()

.setLabelCol("nameIndex")

.setPredictionCol("prediction")

.setMetricName("accuracy")

val accuracy = evaluator.evaluate(preds)

println("accuracy: " + (accuracy \* 100) + "%")

preds.printSchema()

val notDataFrame = data.as[IrisData].rdd

println(notDataFrame.map(\_.SepalWidth).mean())

//notDataFrame.map(\_.)

//dataOne.join(dataTwo, dataOne("Name"), "outer")

//(data.groupBy("Name").mean("SepalWidth", "SepalLength")).show()

}

}

## 1.3SparkMliib高级模块Pipeline原理及实战

#### 1.3.1Pipeline入门

从Spark1.2版本之后引入了ML Pipeline，经过多个版本的发展，SparkMl克服了Mllib在处理复杂机器学习问题的一些不足，如工作比较复杂，流程不够清晰等，向用户提供基于DataFrame之上的更高层次的API库，以方便的构建复杂的机器学习工作流式应用，使得整个机器学习构建过程更加简单、高效和规范。

一个Pipeline在结构上会包含一个或多个Stage，每一个Stage都会完成一个任务，如数据处理、数据转换、模型训练、参数设置或数据预测等，其中两个Stage为Transformer和Estimator。

Transformer主要是用于操作一个DataFrame数据并生成另一个DataFrame数据，比如决策树模型，一个特征提取器都可以抽象为一个Transformer。

Estimator则主要是用来做模型拟合，用来生成一个Transformer。这些Stage有序组成一个Pipeline。

#### 1.3.2DataFrame组件

import org.apache.spark.sql.SparkSession

object dataframeTest1 {

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

//sparksql入口点

val spark = SparkSession

.builder()

.appName("Spark SQL basic example")

.config("spark.some.config.option", "some-value")

.master("local[2]")

.getOrCreate()

// For implicit conversions like converting RDDs to DataFrames

import spark.implicits.\_

val df1 = spark.read.option("header", true).format("csv").load("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/sparkmllib\_helloworld/Pipeline\_ml\_1/customer.txt")

// Displays the content of the DataFrame to stdout

df1.show()

//转换字符类型

val df2 = df1.select(

df1("name").cast("String"),

df1("age").cast("Double"),

df1("gender").cast("String"))

df2.printSchema()

df2.createOrReplaceTempView("customer") // createOrReplaceGlobalTempView

df2.select("name").filter($"age" > 25).show()

val cust1 = spark.sql("select \* from customer where age between 30 and 50")

cust1.limit(5).show()

val cust2 = spark.sql("select \* from customer where gender like 'M'")

cust2.limit(5).show()

}

}

#### 1.3.3Pipeline组件

* **DataFrame**:ML API使用这个来自Spark SQL的概念作为ML dataset，可以保存多种数据类型。比如：使用不同的列存储文本、特征向量、真实标签和预测结果。**Spark ML采用[SchemaRDD](http://spark.apache.org/docs/1.2.0/api/scala/index.html" \l "org.apache.spark.sql.SchemaRDD)Spark SQL，以便在统一的Dataset概念下支持各种数据类型。**
* **Transformer**:转化器，**包括特征变化和学习模型两部分。**这是个是指一个算法将一个DataFrame transform成另一个DataFrame。也就是训练好的模型。比如：一个ML模型就是一个Transformer能够将一个特征数据的DataFrame转成预测结果的DataFrame。
* **Estimator**:估计器，**是指一个操作DataFrame产生Transformer的算法。**比如：一个学习算法就是一个Estimator，可以在一个DataFrame上训练得到一个模型。
  + 估计器通过调用fit方法，接收一个DataFrame产生一个模型，这个模型就是一个Transformer。比如逻辑回归就是一个调用fit方法训练后产生的一个逻辑回归模型。
* **Pipeline**:一个Pipeline链是将多个Transformer和Estimator组合在一起组成一个ML workflow。
* [Param](http://spark.apache.org/docs/1.2.0/ml-guide.html" \l "parameters)：所有Transformers和Estimators现在共享一个用于指定参数的通用API。

Parameter:所有的Transformer和Estimator共享一个公共的说明参数的API。

保存或加载管道 通常情况下，将模型或管道保存到磁盘供以后使用是值得的。模型的导入导出功能在spark1.6的时候加入了pipeline API。大多数基础transformers和基本ML models都支持。

可以将训练好的pipeline输出到磁盘保存起来

model.write.overwrite().save("/opt/spark-logistic-regression-model")

#### 1.3.4Pipeline原理

参考：[http://spark.apache.org/docs/2.2.0/ml-guide.html](http://spark.apache.org/docs/1.2.0/ml-guide.html)

在机器学习中，通常运行一系列算法来处理和训练数据。例如，简单的文本文档处理工作流程可能包括几个阶段：

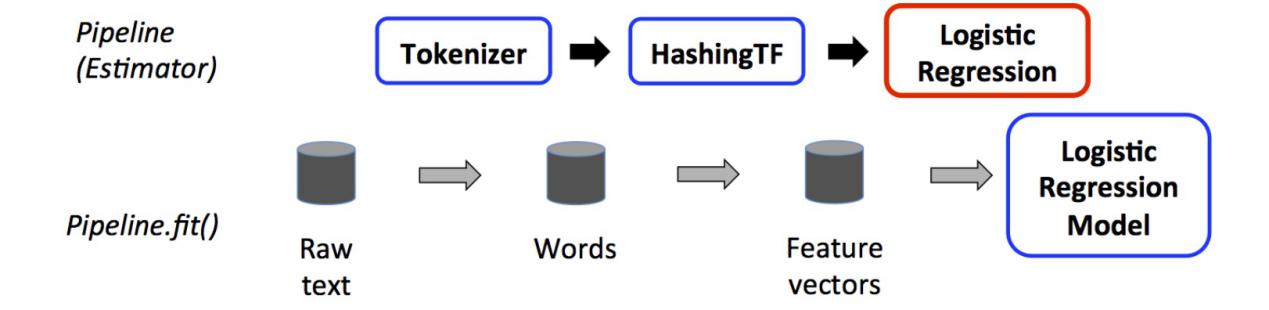
* 将每个文档的文本拆分为单词。
* 将每个文档的单词转换为数字特征向量。
* 使用特征向量和标签学习预测模型。

Spark ML表示这样的工作流程[Pipeline](http://spark.apache.org/docs/1.2.0/api/scala/index.html" \l "org.apache.spark.ml.Pipeline)，它由一系列以特定顺序运行的[PipelineStages](http://spark.apache.org/docs/1.2.0/api/scala/index.html" \l "org.apache.spark.ml.PipelineStage)（Transformers和Estimators）组成。

这个究竟是怎么运作？

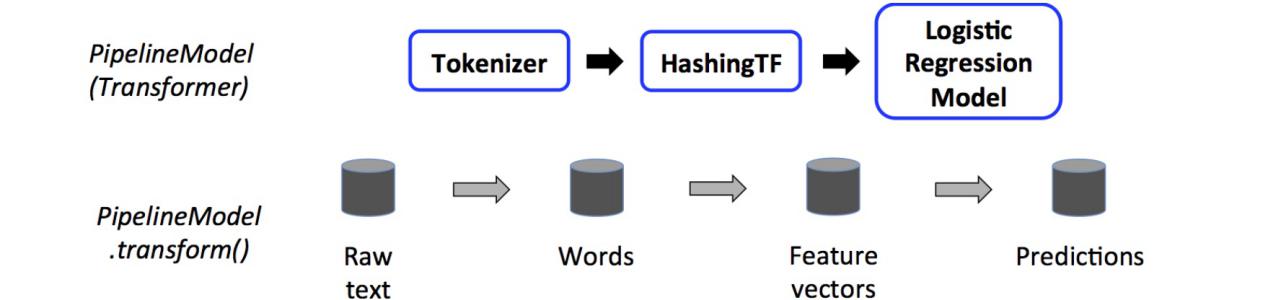
一个Pipeline被指定为阶段序列，每个阶段是一个Transformer或一个 Estimator。这些阶段按顺序运行，输入数据集在通过每个阶段时进行修改。对于Transformer阶段，transform()在数据集上调用该方法。对于Estimator阶段，fit()方法被调用，以产生Transformer（它成为的部分PipelineModel或Pipeline），以及Transformer的transform()方法被称为数据集上。

我们为简单的文本文档工作流说明了这一点。下图是针对a的训练时间使用情况Pipeline。



上面，顶行表示Pipeline三个阶段。前两个（Tokenizer和HashingTF）是Transformers（蓝色），第三个（LogisticRegression）是Estimator（红色）。底行表示流经管道的数据，其中柱面表示SchemaRDDs。Pipeline.fit()在**原始文本文档和标签**的原始数据集上调用该方法。该Tokenizer.transform()方法将原始文本文档拆分为单词，将带有单词的新列添加到数据集中。该HashingTF.transform()方法将单词列转换为要素向量，将具有这些向量的新列添加到数据集。现在，因为LogisticRegression是一个Estimator，Pipeline第一次调用LogisticRegression.fit()生产一个LogisticRegressionModel。如果Pipeline有更多的阶段，它会称之为LogisticRegressionModelstransform() 在将数据集传递到下一阶段之前对数据集的方法。

一个Pipeline是一个Estimator。因此，之后Pipeline的fit()方法运行时，它产生一个PipelineModel其实是Transformer。这PipelineModel是在测试时使用的 ; 如下图：



在上图中，它PipelineModel具有与原始相同的阶段数Pipeline，但原始中的所有Estimators Pipeline都变为Transformers。当PipelineModel的transform()方法用在测试数据集，该数据通过顺序给传递Pipeline。**每个阶段的transform()方法都会更新数据集并将其传递到下一个阶段。**

Pipelines和PipelineModels有助于确保训练和测试数据经过相同的功能处理步骤。

**细节**

* + DAG Pipeline：一个Pipeline的级被指定为一个有序阵列。这里给出的例子都是线性Pipelines，即Pipelines，其中每个阶段使用前一阶段产生的数据。Pipeline只要数据流图形成有向无环图（DAG），就可以创建非线性PIpelines。目前，此图基于每个阶段的输入和输出列名称（通常指定为参数）隐式指定。如果Pipeline形成DAG，则必须按拓扑顺序指定阶段。
  + 运行时检查：由于Pipelines可以对具有不同类型的数据集进行操作，因此它们不能使用编译时类型检查。 Pipelines和PipelineModels代替在实际运行之前进行运行时检查Pipeline。此类型检查是使用数据集模式完成的，该模式是对列中数据类型的描述SchemaRDD。

**参数：**

Spark ML Estimator和Transformers使用统一的API来指定参数。

一个[Param](http://spark.apache.org/docs/1.2.0/api/scala/index.html" \l "org.apache.spark.ml.param.Param)是带有自包含文档的命名参数。一个[ParamMap](http://spark.apache.org/docs/1.2.0/api/scala/index.html" \l "org.apache.spark.ml.param.ParamMap)是一组（参数，值）对。

**将参数传递给算法有两种主要方法：**

设置实例的参数。例如，如果lr是实例LogisticRegression，则可以调用lr.setMaxIter(10)以lr.fit()最多使用10次迭代。此API类似于MLlib中使用的API。

传递ParamMap到fit()或transform()。在任何参数ParamMap将覆盖以前通过setter方法指定的参数。

参数属于Estimators和Transformers的特定实例。例如，如果我们有两个LogisticRegression实例lr1和lr2，然后我们可以建立一个ParamMap与两个maxIter指定的参数：ParamMap(lr1.maxIter -> 10, lr2.maxIter -> 20)。

#### 1.3.5PipeLine实践

##### （1）Estimator，Transformer和Param

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

import org.apache.spark.ml.classification.LogisticRegression

import org.apache.spark.ml.param.ParamMap

import org.apache.spark.ml.classification.LogisticRegression

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

import org.apache.spark.ml.param.ParamMap

import org.apache.spark.sql.Row

// Prepare training data from a list of (label, features) tuples.

val training = spark.createDataFrame(Seq(

(1.0, Vectors.dense(0.0, 1.1, 0.1)),

(0.0, Vectors.dense(2.0, 1.0, -1.0)),

(0.0, Vectors.dense(2.0, 1.3, 1.0)),

(1.0, Vectors.dense(0.0, 1.2, -0.5))

)).toDF("label", "features")

// Create a LogisticRegression instance. This instance is an Estimator.

val lr = new LogisticRegression()

// Print out the parameters, documentation, and any default values.

println("LogisticRegression parameters:\n" + lr.explainParams() + "\n")

// We may set parameters using setter methods.

lr.setMaxIter(10)

.setRegParam(0.01)

// Learn a LogisticRegression model. This uses the parameters stored in lr.

val model1 = lr.fit(training)

// Since model1 is a Model (i.e., a Transformer produced by an Estimator),

// we can view the parameters it used during fit().

// This prints the parameter (name: value) pairs, where names are unique IDs for this

// LogisticRegression instance.

println("Model 1 was fit using parameters: " + model1.parent.extractParamMap)

// We may alternatively specify parameters using a ParamMap,

// which supports several methods for specifying parameters.

val paramMap = ParamMap(lr.maxIter -> 20)

.put(lr.maxIter, 30) // Specify 1 Param. This overwrites the original maxIter.

.put(lr.regParam -> 0.1, lr.threshold -> 0.55) // Specify multiple Params.

// One can also combine ParamMaps.

val paramMap2 = ParamMap(lr.probabilityCol -> "myProbability") // Change output column name.

val paramMapCombined = paramMap ++ paramMap2

// Now learn a new model using the paramMapCombined parameters.

// paramMapCombined overrides all parameters set earlier via lr.set\* methods.

val model2 = lr.fit(training, paramMapCombined)

println("Model 2 was fit using parameters: " + model2.parent.extractParamMap)

// Prepare test data.

val test = spark.createDataFrame(Seq(

(1.0, Vectors.dense(-1.0, 1.5, 1.3)),

(0.0, Vectors.dense(3.0, 2.0, -0.1)),

(1.0, Vectors.dense(0.0, 2.2, -1.5))

)).toDF("label", "features")

// Make predictions on test data using the Transformer.transform() method.

// LogisticRegression.transform will only use the 'features' column.

// Note that model2.transform() outputs a 'myProbability' column instead of the usual

// 'probability' column since we renamed the lr.probabilityCol parameter previously.

model2.transform(test)

.select("features", "label", "myProbability", "prediction")

.collect()

.foreach { case Row(features: Vector, label: Double, prob: Vector, prediction: Double) =>

println(s"($features, $label) -> prob=$prob, prediction=$prediction") }

##### （2）PIpeline文本处理

import org.apache.spark.ml.{Pipeline, PipelineModel}

import org.apache.spark.ml.classification.LogisticRegression

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

import org.apache.spark.ml.linalg.Vector

import org.apache.spark.sql.Row

// Prepare training documents from a list of (id, text, label) tuples.

val training = spark.createDataFrame(Seq(

(0L, "a b c d e spark", 1.0),

(1L, "b d", 0.0),

(2L, "spark f g h", 1.0),

(3L, "hadoop mapreduce", 0.0)

)).toDF("id", "text", "label")

// Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.

val tokenizer = new Tokenizer()

.setInputCol("text")

.setOutputCol("words")

val hashingTF = new HashingTF()

.setNumFeatures(1000)

.setInputCol(tokenizer.getOutputCol)

.setOutputCol("features")

val lr = new LogisticRegression()

.setMaxIter(10)

.setRegParam(0.001)

val pipeline = new Pipeline()

.setStages(Array(tokenizer, hashingTF, lr))

// Fit the pipeline to training documents.

val model = pipeline.fit(training)

// Now we can optionally save the fitted pipeline to disk

model.write.overwrite().save("/tmp/spark-logistic-regression-model")

// We can also save this unfit pipeline to disk

pipeline.write.overwrite().save("/tmp/unfit-lr-model")

// And load it back in during production

val sameModel = PipelineModel.load("/tmp/spark-logistic-regression-model")

// Prepare test documents, which are unlabeled (id, text) tuples.

val test = spark.createDataFrame(Seq(

(4L, "spark i j k"),

(5L, "l m n"),

(6L, "spark hadoop spark"),

(7L, "apache hadoop")

)).toDF("id", "text")

// Make predictions on test documents.

model.transform(test)

.select("id", "text", "probability", "prediction")

.collect()

.foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double) =>

println(s"($id, $text) --> prob=$prob, prediction=$prediction")

}

##### （3）交叉验证

ML中的一项重要任务是模型选择，或使用数据来查找给定任务的最佳模型或参数。这也称为调参。 Pipeline有助于模型选择，方便Pipeline一次调整整个模型，而不是Pipeline单独调整每个部分。

目前，spark.ml支持使用CrossValidator类的模型选择，该类采用Estimator一组ParamMaps和一组Evaluator。 CrossValidator首先将数据集拆分为一组折叠，用作单独的训练和测试数据集; 例如，将一个数据集切分为3个（训练，测试）数据集子集，每个数据集对使用2/3的数据进行训练，1/3进行测试。 遍历一组数据。对于每一个，它训练给定的参数并使用给定的该参数评估模型，ParamMap中产生的参数比较之后产生最好的评价指标（平均值的倍）被选择作为最佳模型。

下面讲解如何使用CrossValidator从GridSearch中进行选择。为了帮助构造参数网格，我们使用该ParamGridBuilder实用程序。

请注意，通过参数网格进行交叉验证非常昂贵。例如，在下面的示例中，参数网格具有3个值hashingTF.numFeatures和2个值lr.regParam，而且CrossValidator使用2个折叠。这会成倍增加到正在训练的不同模型。在实际设置中，通常可以尝试更多参数并使用更多折叠（并且是常见的）。换句话说，使用可能非常昂贵。然而，它也是一种成熟的方法，用于选择比启发式手动调整更具统计学特征的参数。得到(3×2)×2=12k个不同模型，一般训练模型的时候用的3则交叉验证和10则交叉验证较多。

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.classification.LogisticRegression

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

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

import org.apache.spark.ml.linalg.Vector

import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}

import org.apache.spark.sql.Row

// Prepare training data from a list of (id, text, label) tuples.

val training = spark.createDataFrame(Seq(

(0L, "a b c d e spark", 1.0),

(1L, "b d", 0.0),

(2L, "spark f g h", 1.0),

(3L, "hadoop mapreduce", 0.0),

(4L, "b spark who", 1.0),

(5L, "g d a y", 0.0),

(6L, "spark fly", 1.0),

(7L, "was mapreduce", 0.0),

(8L, "e spark program", 1.0),

(9L, "a e c l", 0.0),

(10L, "spark compile", 1.0),

(11L, "hadoop software", 0.0)

)).toDF("id", "text", "label")

// Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.

val tokenizer = new Tokenizer()

.setInputCol("text")

.setOutputCol("words")

val hashingTF = new HashingTF()

.setInputCol(tokenizer.getOutputCol)

.setOutputCol("features")

val lr = new LogisticRegression()

.setMaxIter(10)

val pipeline = new Pipeline()

.setStages(Array(tokenizer, hashingTF, lr))

// We use a ParamGridBuilder to construct a grid of parameters to search over.

// With 3 values for hashingTF.numFeatures and 2 values for lr.regParam,

// this grid will have 3 x 2 = 6 parameter settings for CrossValidator to choose from.

val paramGrid = new ParamGridBuilder()

.addGrid(hashingTF.numFeatures, Array(10, 100, 1000))

.addGrid(lr.regParam, Array(0.1, 0.01))

.build()

// We now treat the Pipeline as an Estimator, wrapping it in a CrossValidator instance.

// This will allow us to jointly choose parameters for all Pipeline stages.

// A CrossValidator requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.

// Note that the evaluator here is a BinaryClassificationEvaluator and its default metric

// is areaUnderROC.

val cv = new CrossValidator()

.setEstimator(pipeline)

.setEvaluator(new BinaryClassificationEvaluator)

.setEstimatorParamMaps(paramGrid)

.setNumFolds(2) // Use 3+ in practice

// Run cross-validation, and choose the best set of parameters.

val cvModel = cv.fit(training)

// Prepare test documents, which are unlabeled (id, text) tuples.

val test = spark.createDataFrame(Seq(

(4L, "spark i j k"),

(5L, "l m n"),

(6L, "mapreduce spark"),

(7L, "apache hadoop")

)).toDF("id", "text")

// Make predictions on test documents. cvModel uses the best model found (lrModel).

cvModel.transform(test)

.select("id", "text", "probability", "prediction")

.collect()

.foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double) =>

println(s"($id, $text) --> prob=$prob, prediction=$prediction")

}

## 1.4 SparkMllib的模型选择和优化

#### 1.4.1模型选择

**ML中的一个重要任务是模型选择**，或使用数据找到给定任务的最佳模型或参数。这也叫**调优**。可以针对个体估算器（如Logistic回归）或包括多个算法，特征化和其他步骤的整个管道完成调整。用户可以一次调整整个流水线，而不是单独调整管道中的每个元素。

MLlib支持使用 CrossValidator和TrainValidationSplit等工具进行模型选择。这些工具需要以下项目：

* Estimator（估算器）：算法或管道调整
* ParamMaps集：可供选择的参数，有时称为“**参数网格**”进行搜索
* Evaluator（评估者）：衡量拟合模型对延伸测试数据有多好的度量

这些模型选择工具的工作如下：

* + 他们将输入数据分成单独的训练和测试数据集。
  + 对于每个（训练，测试）对，他们一组ParamMaps集合进行迭代；
  + 对于每个ParamMap，它们使用这些参数适合Estimator，获得拟合的Model，并使用Evaluator评估Model的性能。

他们选择由最佳性能参数组合生成的模型。

评估者可以是回归问题的RegressionEvaluator，二进制数据的BinaryClassificationEvaluator或多类问题的MulticlassClassificationEvaluator。用于选择最佳ParamMap的默认度量可以被这些评估器中的每一个的setMetricName方法覆盖。

#### 1.4.2交叉验证

CrossValidator首先将数据集分成一组折叠，这些折叠用作单独的训练和测试数据集。 例如，k = 3倍，CrossValidator将生成3个（训练，测试）数据集对，每个数据集使用2/3的数据进行训练，1/3进行测试。为了评估一个特定的ParamMap，CrossValidator通过在3个不同的（训练，测试）数据集对上拟合Estimator来计算3个模型的平均评估度量。在确定最佳ParamMap之后，CrossValidator最终使用最好的ParamMap和整个数据集重新拟合Estimator。

案例：通过交叉-验证进行模型选择，如何使用CrossValidator从参数网格中进行选择。请注意，通过参数网格的交叉验证是昂贵的。 例如，在下面的示例中，参数网格具有3个值，用于hashingTF.numFeatures，2个值用于lr.regParam，CrossValidator使用2折交叉验证。 得到模型一共 (3×2)×2 = 12。一般k=3或10较为常用。

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.classification.LogisticRegression

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

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

import org.apache.spark.ml.linalg.Vector

import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}

import org.apache.spark.sql.Row

// Prepare training data from a list of (id, text, label) tuples.

val training = spark.createDataFrame(Seq(

(0L, "a b c d e spark", 1.0),

(1L, "b d", 0.0),

(2L, "spark f g h", 1.0),

(3L, "hadoop mapreduce", 0.0),

(4L, "b spark who", 1.0),

(5L, "g d a y", 0.0),

(6L, "spark fly", 1.0),

(7L, "was mapreduce", 0.0),

(8L, "e spark program", 1.0),

(9L, "a e c l", 0.0),

(10L, "spark compile", 1.0),

(11L, "hadoop software", 0.0)

)).toDF("id", "text", "label")

// Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.

val tokenizer = new Tokenizer()

.setInputCol("text")

.setOutputCol("words")

val hashingTF = new HashingTF()

.setInputCol(tokenizer.getOutputCol)

.setOutputCol("features")

val lr = new LogisticRegression()

.setMaxIter(10)

val pipeline = new Pipeline()

.setStages(Array(tokenizer, hashingTF, lr))

// We use a ParamGridBuilder to construct a grid of parameters to search over.

// With 3 values for hashingTF.numFeatures and 2 values for lr.regParam,

// this grid will have 3 x 2 = 6 parameter settings for CrossValidator to choose from.

val paramGrid = new ParamGridBuilder()

.addGrid(hashingTF.numFeatures, Array(10, 100, 1000))

.addGrid(lr.regParam, Array(0.1, 0.01))

.build()

// We now treat the Pipeline as an Estimator, wrapping it in a CrossValidator instance.

// This will allow us to jointly choose parameters for all Pipeline stages.

// A CrossValidator requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.

// Note that the evaluator here is a BinaryClassificationEvaluator and its default metric

// is areaUnderROC.

val cv = new CrossValidator()

.setEstimator(pipeline)

.setEvaluator(new BinaryClassificationEvaluator)

.setEstimatorParamMaps(paramGrid)

.setNumFolds(2) // Use 3+ in practice

// Run cross-validation, and choose the best set of parameters.

val cvModel = cv.fit(training)

// Prepare test documents, which are unlabeled (id, text) tuples.

val test = spark.createDataFrame(Seq(

(4L, "spark i j k"),

(5L, "l m n"),

(6L, "mapreduce spark"),

(7L, "apache hadoop")

)).toDF("id", "text")

// Make predictions on test documents. cvModel uses the best model found (lrModel).

cvModel.transform(test)

.select("id", "text", "probability", "prediction")

.collect()

.foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double) =>

println(s"($id, $text) --> prob=$prob, prediction=$prediction")

}

#### 1.4.3训练验证方法切分

除了CrossValidator Spark，还提供了用于超参数调整的TrainValidationSplit。 TrainValidationSplit仅对参数每个组合进行一次评估，而在CrossValidator的情况下，则不是k次。 因此，它较便宜，但在训练数据集不够大时不会产生可靠的结果。与CrossValidator不同，TrainValidationSplit创建一个（训练，测试）数据集对。 它使用trainRatio参数将数据集分成这两个部分。 例如，trainRatio = 0.75，TrainValidationSplit将生成训练和测试数据集对，其中75％的数据用于训练，25％用于验证。

像CrossValidator一样，TrainValidationSplit最终适合使用最好的ParamMap和整个数据集的Estimator。

代码：

import org.apache.spark.ml.evaluation.RegressionEvaluator

import org.apache.spark.ml.evaluation.RegressionEvaluator

import org.apache.spark.ml.regression.LinearRegression

import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

// Prepare training and test data.

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

val Array(training, test) = data.randomSplit(Array(0.9, 0.1), seed = 12345)

val lr = new LinearRegression()

.setMaxIter(10)

// We use a ParamGridBuilder to construct a grid of parameters to search over.

// TrainValidationSplit will try all combinations of values and determine best model using

// the evaluator.

val paramGrid = new ParamGridBuilder()

.addGrid(lr.regParam, Array(0.1, 0.01))

.addGrid(lr.fitIntercept)

.addGrid(lr.elasticNetParam, Array(0.0, 0.5, 1.0))

.build()

// In this case the estimator is simply the linear regression.

// A TrainValidationSplit requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.

val trainValidationSplit = new TrainValidationSplit()

.setEstimator(lr)

.setEvaluator(new RegressionEvaluator)

.setEstimatorParamMaps(paramGrid)

// 80% of the data will be used for training and the remaining 20% for validation.

.setTrainRatio(0.8)

// Run train validation split, and choose the best set of parameters.

val model = trainValidationSplit.fit(training)

// Make predictions on test data. model is the model with combination of parameters

// that performed best.

model.transform(test).select("features", "label", "prediction") .show()

#### 1.4.4自定义模型选择

案例分析：

package cn.itcast.sparkmllib\_day03\_classification  
  
import org.apache.spark.SparkConf  
import org.apache.spark.ml.Pipeline  
import org.apache.spark.ml.classification.DecisionTreeClassifier  
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator  
import org.apache.spark.ml.feature.{StringIndexer, VectorAssembler}  
import org.apache.spark.ml.param.ParamMap  
import org.apache.spark.ml.tuning.{CrossValidator, CrossValidatorModel, ParamGridBuilder, TrainValidationSplit}  
import org.apache.spark.sql.{DataFrame, Dataset, Row, SparkSession}  
  
/\*\*  
 \* DESC: 构建机器学习模型  
 \* Complete data processing and modeling process steps:  
 \* 构建机器学习算法的流程  
 \* \*  
 \*- 1-准备数据  
 \*- 2-数据处理  
 \*- 3-特征工程  
 \*- 特征提取  
 \*- 特征转换  
 \*- 特征选择  
 \*- 4-数据集的拆分  
 \*- 5-准备算法------分类算法  
 \*- 6-训练模型  
 \*- 7-模型预测分析  
 \*- 8-模型的校验  
 \*- 9-模型的保存  
 \*/  
object sparkmllib\_irisModel\_pipeline\_trainvalidationsplit {  
 def main(args: Array[String]): Unit = {  
 // \*- 0-准备环境  
 val conf: SparkConf = new SparkConf().setAppName("sparkmllib\_irisModel").setMaster("local[\*]")  
 val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()  
 spark.sparkContext.setLogLevel("WARN")  
 // \*- 1-准备数据  
 val path = "D:\\BigData\\Workspace\\SparkMachineLearningTest\\SparkMllib\_BigData30\\src\\main\\resources\\iris.csv"  
 val data: DataFrame = spark.read.format("csv").option("header", "true").option("inferSchema", "true").load(path)  
 data.printSchema()  
 // \*- 2-数据处理  
 // \*- 3-特征工程  
 // \*- 特征提取  
 // \*- 特征转换  
 //3-1-StringIndexer转换模块：  
 //.setInputCol("class") 设置输入列，该列一定是数据集中本身存在的需要处理的特征列，如class特征需要转化为index信息  
 //.setOutputCol("indexedclass") 设置输出列，自己可以设定名字  
 val strIndex: StringIndexer = new StringIndexer().setInputCol("class").setOutputCol("indexedclass")  
 //3-2VectorAssemble  
 //为什么需要做特征组合的转换？原因是因为上述的4个特征信息是独立的，而建立模型需要整合为一个样本含有四个特征列的形式  
 //.setInputCols()需要设置Array数组代表的不同的特征字段  
 //解析：def setInputCols(value: Array[String]): this.type = set(inputCols, value)  
 // .setOutputCol() 自己设定的，显示的是将几个特征整合为一个特征向量的过程  
 val vec: VectorAssembler = new VectorAssembler()  
 .setInputCols(Array("sepal\_length", "sepal\_width", "petal\_length", "petal\_width")).setOutputCol("features")  
 // \*- 特征选择  
 // \*- 4-数据集的拆分  
 // seed随机数种子----保证每次随机切分结果的可重复性  
 val Array(trainingSet, testSet): Array[Dataset[Row]] = data.randomSplit(Array(0.8, 0.2), seed = 123L)  
 // val data = vecResult.randomSplit(Array(0.8, 0.2), seed = 123L)  
 // trainingSet=data(0)  
 // testSet=data(1)  
 // \*- 5-准备算法------分类算法  
 // .setFeaturesCol() 特征列，需要给定string类型的特征列  
 // .setLabelCol() 标签列，需要给定过的是数据中的label列  
 // .setImpurity() 不纯度的度量，entropy和gini系数的参数  
 // .setMaxDepth() 树的深度，默认5  
 // .setMaxBins() 分箱的个数，默认32  
 // .setMinInfoGain() 最小的不纯度，默认0  
 // .setMinInstancesPerNode() 每个节点含有的样本的个数最小的阈值，设定默认值是1  
 val dtc = new DecisionTreeClassifier()  
 .setFeaturesCol("features")  
 .setLabelCol("indexedclass")  
 .setImpurity("entropy")  
 .setMaxDepth(5)  
 .setMaxBins(32)  
 .setMinInfoGain(0.0)  
 .setMinInstancesPerNode(1)  
 .setPredictionCol("prces")  
 // \*- 6-训练模型  
 val pipeline: Pipeline = new Pipeline().setStages(Array(strIndex, vec, dtc))  
 //网格搜索指定超参数  
 val param: Array[ParamMap] = new ParamGridBuilder().addGrid(dtc.impurity, Array("gini", "entropy")).addGrid(dtc.maxDepth, Array(4, 5, 6, 7)).build()  
 // \*- 7-模型的校验  
 //.setLabelCol() 标签列  
 // .setPredictionCol() 预测列  
 // .setMetricName("accuracy") 参数列表  
 val evaluator: MulticlassClassificationEvaluator = new MulticlassClassificationEvaluator()  
 .setLabelCol("indexedclass")  
 .setPredictionCol("prces")  
 .setMetricName("accuracy")  
 //8-交叉验证  
 // .setEstimator() 设置学习器或具体的算法  
 // .setEvaluator() 设置校验方式---分类问题如何检验  
 // .setNumFolds() 几折交叉验证  
 // .setEstimatorParamMaps() 指定的不同的超参数  
 val crossvaliator = new TrainValidationSplit().setEstimator(pipeline).setEvaluator(evaluator).setEstimatorParamMaps(param).setTrainRatio(0.8)  
 val crossModel = crossvaliator.fit(trainingSet)  
 crossModel.transform(testSet)  
 // \*- 7-通过crossvalia方法输出参数  
 val extractParamMap: ParamMap = crossModel.extractParamMap()  
 println(extractParamMap)  
 // {  
 // tvs\_cab848d53081-estimator: pipeline\_8a77b264e2d1,  
 // tvs\_cab848d53081-estimatorParamMaps: [Lorg.apache.spark.ml.param.ParamMap;@78494a59,  
 // tvs\_cab848d53081-evaluator: mcEval\_709a58fd527e,  
 // tvs\_cab848d53081-seed: -1772833110,  
 // tvs\_cab848d53081-trainRatio: 0.8  
 // }  
 import org.apache.spark.ml.PipelineModel  
 crossModel  
 .  
 val pipeModel = crossModel.bestModel.asInstanceOf[PipelineModel]  
 println(pipeModel.stages(2).extractParamMap)  
 /\* {  
 dtc\_e77eebb80fef-cacheNodeIds: false,  
 dtc\_e77eebb80fef-checkpointInterval: 10,  
 dtc\_e77eebb80fef-featuresCol: features,  
 dtc\_e77eebb80fef-impurity: gini,  
 dtc\_e77eebb80fef-labelCol: indexedclass,  
 dtc\_e77eebb80fef-maxBins: 32,  
 dtc\_e77eebb80fef-maxDepth: 4,  
 dtc\_e77eebb80fef-maxMemoryInMB: 256,  
 dtc\_e77eebb80fef-minInfoGain: 0.0,  
 dtc\_e77eebb80fef-minInstancesPerNode: 1,  
 dtc\_e77eebb80fef-predictionCol: prces,  
 dtc\_e77eebb80fef-probabilityCol: probability,  
 dtc\_e77eebb80fef-rawPredictionCol: rawPrediction,  
 dtc\_e77eebb80fef-seed: 159147643  
 }\*/  
 }  
}

# 3.用户购物性别模型-USG

## 1.1USG模型引入

USG(User Shopping Gender)

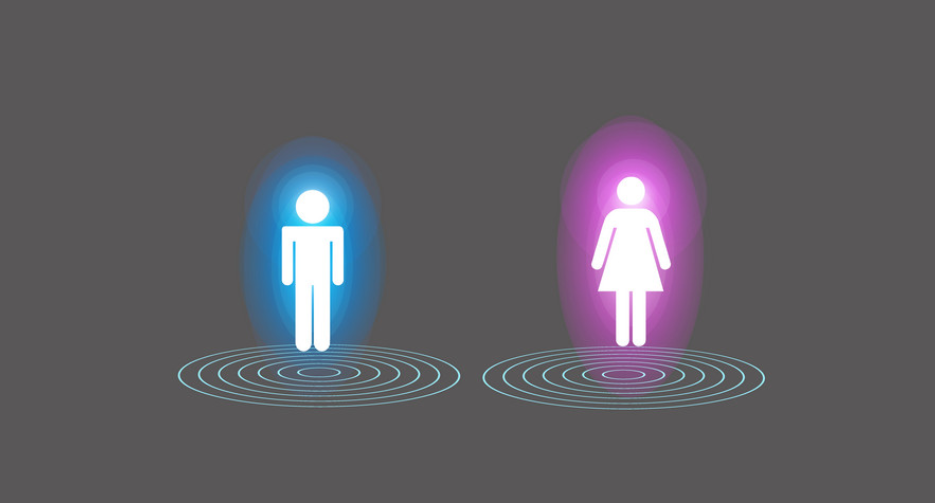
### 1.1.2AI驱动的电商用户模型：性别属性是这样确定的

首先带领大家了解一下，如何通过大数据来确定用户的真实性别。

我们经常谈论的用户精细化运营，到底是什么？简单来讲，就是将网站的每个用户标签化，制作一个属于他自己的网络身份证。

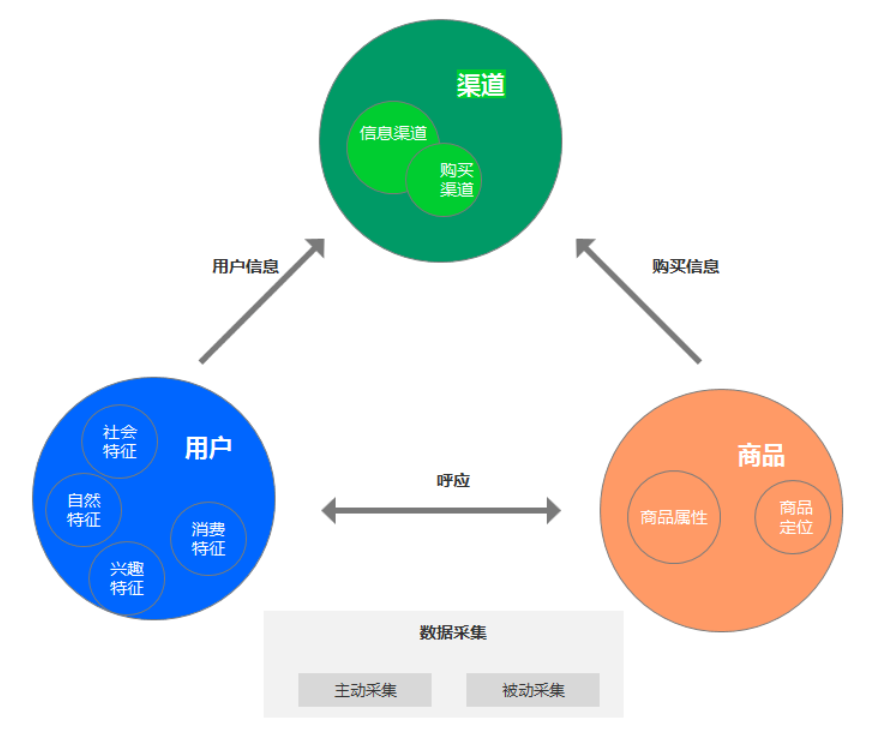
然后，运用人员通过身份证来确定活动的投放人群，圈定人群范围，更为精准的用户培养和管理。当然，身份证最基本的信息就是姓名，年龄和性别，与现实不同的是，网络上用户填写的资料不一定完全准确，还需要进行进一步的确认和评估。

确定性别这件事很重要，简单举个栗子，比如店铺想推荐新品的Bra，如果粗糙的全部投放人群或者投放到不准确性别的人群，那后果可想而知了。下面来介绍一下具体的识别思路



#### （1）用户画像需要的数据

用户平时在电商网站的购物行为、浏览行为、搜索行为，以及订单购买情况都会被记录在案，探查其消费能力，兴趣等。数据归类后，一般来讲，可以通过三类数据对用户进行分群和定义。



1、用户信息

社会特征：马克思的人性观把人分为社会属性和自然属性。社会特征主要指的是人在社会上的阶级属性，当然也包括服从性、依赖性或者自觉性等，这是人类发展的必然的基本要求。

自然特征：也可以说成是人的生物性，通常来讲可以是食欲，物欲或者购买欲，自我保存能力。但不同人会有不同的自然特征，比如学习能力和逻辑思维等。

兴趣特征：对于电商来讲，主要是对某件商品，某个品牌或者品类的兴趣程度，如加购、浏览、收藏、搜索和下单行为。

消费特征：消费能力的评估，消费倾向的评估，能够判断用户的消费层级，是高消费力还是低消费力。

2、商品

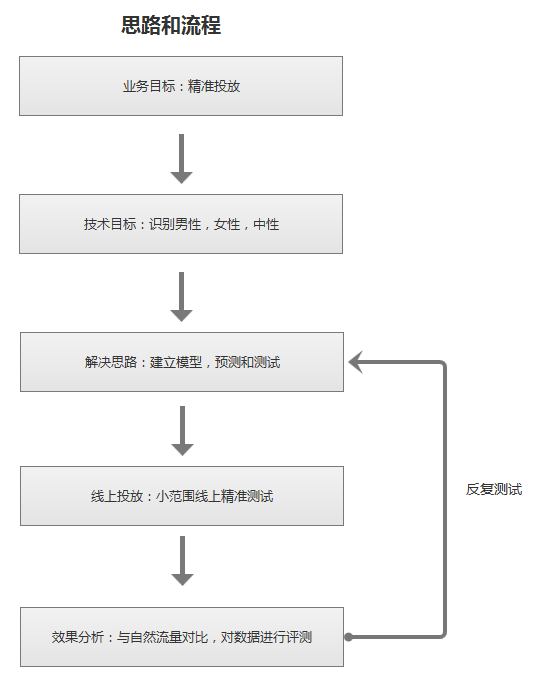
商品属性：基本信息，品类，颜色尺码型号等。

商品定位：商品层级，是否为高中低端，商品类型倾向于哪类客户，区域或者其他的特征。

最后通过以上的信息来获取用户信息，判断其具体的画像特征，然后得到类似于酱紫的网络身份证。



通常，拿到数据后，我们会将每个环节进行拆解，落实到具体的行动策略上。大体可以根据以下流程进行模型的预估：



业务目标： 精准投放：针对已有产品，寻找某性别偏好的精准人群进行广告投放。

技术目标： 对用户购物性别识别：男性，女性，中性

解决思路：选择一种分类算法，建立机器学习算法模型，对模型进行应用

线上投放：对得到的数据进行小范围内的测试投放，初期不宜过大扩大投放范围

效果分析：对投放的用户进行数据分析，评估数据的准确性。若不够完美，则需要重新建模和测试

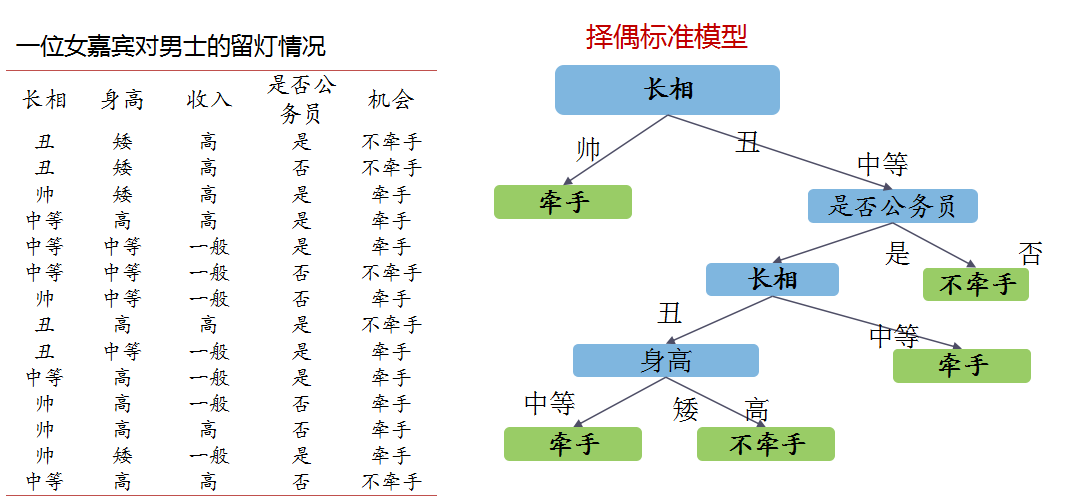
#### （2）如何理解建模过程

重点来了，虽然能够通过用户的行为、购买和兴趣数据，了解用户的基本信息，但是仍然不清楚如何建模？用什么语言建模？

其实，购物性别的区分使用的是spark，但是机器学习算法也有很多分类，包含逻辑回归，线性支*持向量机，朴素贝叶斯模型和决策树。那么，又该如何选择呢？*

*其中，决策树的优点较多，主要是其变量处理灵活，不要求相互独立。可处理大维度的数据，不用预先对模型的特征有所了解。对于表达复杂的非线性模式和特征的相互关系，模型相对容易理解和解释。看起来决策树的方法最适合区分性别特征了，所以决定用决策树进行尝试。*

*什么是决策树？简单来讲，是通过训练数据来构建一棵用于分类的树，从而对未知数据进行高效分类。可以从下面的图了解决策树的工作原理。*



构造决策树的步骤为：

通过训练数据来构建一棵用于分类的树，从而对未知数据进行高效分类。

以上步骤中，能够得出一个结论，在构建决策树的过程中，最重要的是如何找到最好的分割点。决策树值得注意的问题是过拟合问题，整个算法必须解决「如何停止分割」和「如何选择分割」两个关键问题。

最简单的做法就是设定树的深度或枝叶的最少样本量。但是，过少的样本量又不具有代表性，所以一般情况，可以使用交叉验证的方法。交叉验证就是可以使用一部分数据用于模型的训练，另一部分数据可以用来评估模型的性能。业内常用的划分方法是讲样本进行50/50分，60/40分或者80/20分。

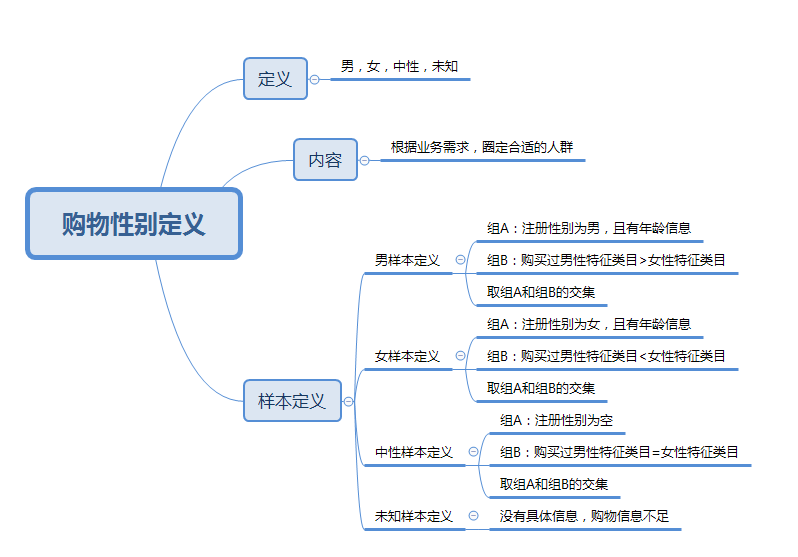
#### （3）模型确立过程

在建模前期，首要考虑的事情就是先确定指标，以及对样本的定义。购物性别指的是什么？通过哪些数据来确定购物性别，样本的准确性，如何验证数据的可信度等。

#### （4）购物性别的定义

先看下图，具体的逻辑可从图中查看。一般来讲，用户填写的资料不一定真实，我们对他/她的性别数据持怀疑态度，所以，就需要其他数据进行辅助证明其性别。

订单数据能够真实反映用户的购买心态，预测购买行为，并且能够通过购买商品的所属类别，判断用户的购买倾向，最后得到性别特征类目。不过本文就不展开探讨甄别特征类目的区分方法了。



根据数据结果，最终，确认了购物性别的定义。分为：

购物性别男：N月购买的男性特征类目子下单数> N月购买的女性特征类目子下单数

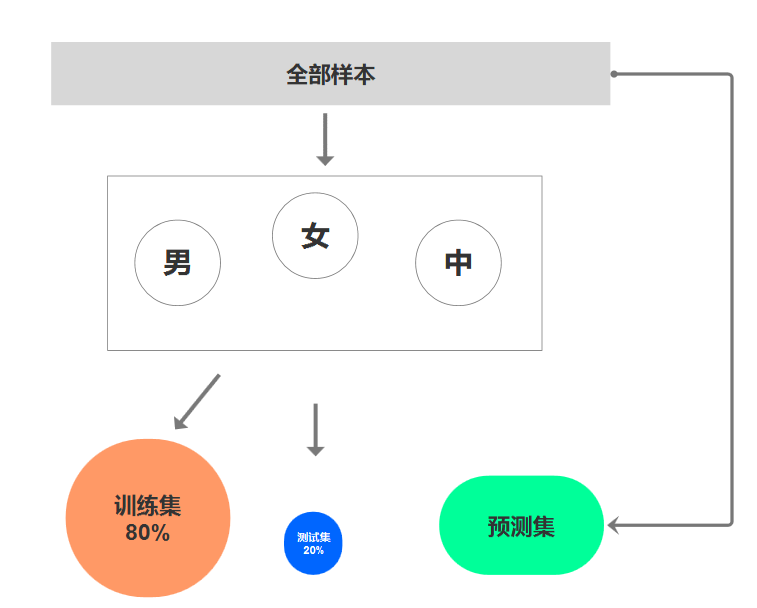
购物性别女：N月购买的男性特征类目子下单数<N月购买的女性特征类目子下单数

购物性别中性：未下单男女特征类目

N需要具体根据业务场景来定。

#### （5）建模数据准备过程

本节是具体的操作过程，模型的实操阶段。一般来讲，不同模型的训练其实大体雷同。从技术上来讲，各家算法大多使用sparkmllib，不同点是所运算的模型都是针对于场景来定的。



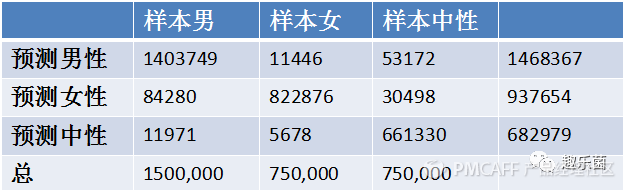
在全部样本中，取80%的数据用于训练模型

在全部样本中，取20%的数据用户数据测试

这种方式可以更好的根据数据的规模，提高模型的准确性

#### （6）模型效果分析

行业内当前采用数据挖掘、机器学习和推荐系统中的评测指标—准确率(accuracy)、精确率(Precision)、召回率(Recall)，准确率是应用最广的数据指标，也很清晰易懂，以男性为例



准确率=命中的男性用户数量/所有预测男性数量，一般来讲，准确率可以评估模型的质量，他是很直观的数据评价，但并不是说准确度越高，算法越好。

召回率=命中的男性用户数量/所有男性数量，反**映了被正确判定的正例占总的正例的比重。**

模型建立完后，需根据模型的结果与预期的对比，进行调优。

#### （7）最后要说的

购物性别定义对于用户精准营销十分重要，疑难杂症，对症下药，才能出现更好的疗效。

对于新手来说，初期一定是对模型性能及效果分析不是很熟练，可先用小数据量进行测试， 走通全流程 建表要规范，方便后期批量删除，因为建模是个反复的过程。

根据各类参数的评估结果，以及人工经验选定的模型参数，建立模型。值得注意的是，决策树的深度不要过深，以防止过拟合的问题

## 1.2决策树分类算法详解

决策树是最经典的机器学习模型之一。它的预测结果容易理解， 易于向业务部门解释，

预测速度快，可以处理类别型数据和连续型数据。在机器学习的数据挖掘类求职面试中，决策树是面试官最喜欢的面试题之一。

### 1.2.1算法原理回顾

构建决策树三要素回顾。

### 1.2.2代码快速回顾

package cn.itcast.up.ml

import org.apache.spark.ml.{Pipeline, PipelineModel}

import org.apache.spark.ml.classification.{DecisionTreeClassificationModel, DecisionTreeClassifier}

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.{IndexToString, StringIndexer, StringIndexerModel, VectorAssembler}

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

import org.apache.spark.sql.types.DoubleType

object IrisDecisionTree {

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 source: DataFrame = spark.read

.csv("file:///D:\\data\\spark\\ml\\iris.csv")

.toDF("Sepal\_Length", "Sepal\_Width", "Petal\_Length", "Petal\_Width", "Species")

.select(

'Sepal\_Length cast DoubleType,

'Sepal\_Width cast DoubleType,

'Petal\_Length cast DoubleType,

'Petal\_Width cast DoubleType,

'Species)

source.show(false)

/\*

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

|Sepal\_Length|Sepal\_Width|Petal\_Length|Petal\_Width|Species |

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

|5.1 |3.5 |1.4 |0.2 |Iris-setosa|

|4.9 |3.0 |1.4 |0.2 |Iris-setosa|

|4.7 |3.2 |1.3 |0.2 |Iris-setosa|

|4.6 |3.1 |1.5 |0.2 |Iris-setosa|

|5.0 |3.6 |1.4 |0.2 |Iris-setosa|

|5.4 |3.9 |1.7 |0.4 |Iris-setosa|

|4.6 |3.4 |1.4 |0.3 |Iris-setosa|

|5.0 |3.4 |1.5 |0.2 |Iris-setosa|

|4.4 |2.9 |1.4 |0.2 |Iris-setosa|

|4.9 |3.1 |1.5 |0.1 |Iris-setosa|

|5.4 |3.7 |1.5 |0.2 |Iris-setosa|

|4.8 |3.4 |1.6 |0.2 |Iris-setosa|

|4.8 |3.0 |1.4 |0.1 |Iris-setosa|

|4.3 |3.0 |1.1 |0.1 |Iris-setosa|

|5.8 |4.0 |1.2 |0.2 |Iris-setosa|

|5.7 |4.4 |1.5 |0.4 |Iris-setosa|

|5.4 |3.9 |1.3 |0.4 |Iris-setosa|

|5.1 |3.5 |1.4 |0.3 |Iris-setosa|

|5.7 |3.8 |1.7 |0.3 |Iris-setosa|

|5.1 |3.8 |1.5 |0.3 |Iris-setosa|

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

only showing top 20 rows

\*/

//2. 处理标签

val labelIndexer: StringIndexerModel = new StringIndexer()

.setInputCol("Species")

.setOutputCol("label")

.fit(source)

//3. 处理特征

val featuresAssembler: VectorAssembler = new VectorAssembler()

.setInputCols(Array("Sepal\_Width", "Petal\_Length", "Sepal\_Length", "Petal\_Width"))

.setOutputCol("features")

**//4. 分类器**

val decisionTreeClassifier: DecisionTreeClassifier = new DecisionTreeClassifier()

.setFeaturesCol("features")

.setPredictionCol("prediction")

.setMaxDepth(5)

**//5. 还原标签**

val labelConverter: IndexToString = new IndexToString()

.setInputCol("label")

.setOutputCol("species\_converted")

**//6. 划分训练集、测试集**

val Array(trainingData, testData) = source.randomSplit(Array(0.8, 0.2))

**//7. 使用 Pipeline 计算**

val pipeline: Pipeline = new Pipeline().setStages(Array(labelIndexer, featuresAssembler, decisionTreeClassifier, labelConverter))

val model: PipelineModel = pipeline.fit(trainingData)

val resultDF: DataFrame = model.transform(testData)

resultDF.show(false)

/\*

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

|Sepal\_Length|Sepal\_Width|Petal\_Length|Petal\_Width|Species |label|features |rawPrediction |probability |prediction|species\_converted|

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

|4.9 |2.4 |3.3 |1.0 |Iris-versicolor|1.0 |[2.4,3.3,4.9,1.0]|[0.0,0.0,1.0] |[0.0,0.0,1.0]|2.0 |Iris-versicolor |

|4.9 |3.0 |1.4 |0.2 |Iris-setosa |0.0 |[3.0,1.4,4.9,0.2]|[43.0,0.0,0.0]|[1.0,0.0,0.0]|0.0 |Iris-setosa |

|4.9 |3.1 |1.5 |0.1 |Iris-setosa |0.0 |[3.1,1.5,4.9,0.1]|[43.0,0.0,0.0]|[1.0,0.0,0.0]|0.0 |Iris-setosa |

|5.0 |2.0 |3.5 |1.0 |Iris-versicolor|1.0 |[2.0,3.5,5.0,1.0]|[0.0,40.0,0.0]|[0.0,1.0,0.0]|1.0 |Iris-versicolor |

|5.0 |3.5 |1.3 |0.3 |Iris-setosa |0.0 |[3.5,1.3,5.0,0.3]|[43.0,0.0,0.0]|[1.0,0.0,0.0]|0.0 |Iris-setosa |

|5.1 |2.5 |3.0 |1.1 |Iris-versicolor|1.0 |[2.5,3.0,5.1,1.1]|[0.0,40.0,0.0]|[0.0,1.0,0.0]|1.0 |Iris-versicolor |

|5.2 |3.4 |1.4 |0.2 |Iris-setosa |0.0 |[3.4,1.4,5.2,0.2]|[43.0,0.0,0.0]|[1.0,0.0,0.0]|0.0 |Iris-setosa |

|5.2 |3.5 |1.5 |0.2 |Iris-setosa |0.0 |[3.5,1.5,5.2,0.2]|[43.0,0.0,0.0]|[1.0,0.0,0.0]|0.0 |Iris-setosa |

|5.2 |4.1 |1.5 |0.1 |Iris-setosa |0.0 |[4.1,1.5,5.2,0.1]|[43.0,0.0,0.0]|[1.0,0.0,0.0]|0.0 |Iris-setosa |

|5.4 |3.4 |1.7 |0.2 |Iris-setosa |0.0 |[3.4,1.7,5.4,0.2]|[43.0,0.0,0.0]|[1.0,0.0,0.0]|0.0 |Iris-setosa |

|5.5 |2.3 |4.0 |1.3 |Iris-versicolor|1.0 |[2.3,4.0,5.5,1.3]|[0.0,40.0,0.0]|[0.0,1.0,0.0]|1.0 |Iris-versicolor |

|5.5 |2.4 |3.7 |1.0 |Iris-versicolor|1.0 |[2.4,3.7,5.5,1.0]|[0.0,40.0,0.0]|[0.0,1.0,0.0]|1.0 |Iris-versicolor |

|5.5 |2.6 |4.4 |1.2 |Iris-versicolor|1.0 |[2.6,4.4,5.5,1.2]|[0.0,40.0,0.0]|[0.0,1.0,0.0]|1.0 |Iris-versicolor |

|6.0 |2.2 |5.0 |1.5 |Iris-virginica |2.0 |[2.2,5.0,6.0,1.5]|[0.0,40.0,0.0]|[0.0,1.0,0.0]|1.0 |Iris-virginica |

|6.0 |2.7 |5.1 |1.6 |Iris-versicolor|1.0 |[2.7,5.1,6.0,1.6]|[0.0,0.0,3.0] |[0.0,0.0,1.0]|2.0 |Iris-versicolor |

|6.1 |2.8 |4.0 |1.3 |Iris-versicolor|1.0 |[2.8,4.0,6.1,1.3]|[0.0,40.0,0.0]|[0.0,1.0,0.0]|1.0 |Iris-versicolor |

|6.3 |3.3 |6.0 |2.5 |Iris-virginica |2.0 |[3.3,6.0,6.3,2.5]|[0.0,0.0,38.0]|[0.0,0.0,1.0]|2.0 |Iris-virginica |

|6.4 |3.1 |5.5 |1.8 |Iris-virginica |2.0 |[3.1,5.5,6.4,1.8]|[0.0,0.0,38.0]|[0.0,0.0,1.0]|2.0 |Iris-virginica |

|6.7 |3.3 |5.7 |2.1 |Iris-virginica |2.0 |[3.3,5.7,6.7,2.1]|[0.0,0.0,38.0]|[0.0,0.0,1.0]|2.0 |Iris-virginica |

|6.9 |3.1 |4.9 |1.5 |Iris-versicolor|1.0 |[3.1,4.9,6.9,1.5]|[0.0,40.0,0.0]|[0.0,1.0,0.0]|1.0 |Iris-versicolor |

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

only showing top 20 rows

\*/

**//8. 验证结果**

val evalutor: MulticlassClassificationEvaluator = new MulticlassClassificationEvaluator()

.setLabelCol("label")

.setPredictionCol("prediction")

.setMetricName("accuracy")

val acc: Double = evalutor.evaluate(resultDF)

println(s"ERROR Ratio : ${1 - acc}")

/\*

ERROR Ratio : 0.13636363636363635

\*/

**//9.查看决策树**

val treeModel: DecisionTreeClassificationModel = model.stages(2).asInstanceOf[DecisionTreeClassificationModel]

println("Learned DecisionTreeClassificationModel:\n" + treeModel.toDebugString)

/\*

Learned DecisionTreeClassificationModel:

DecisionTreeClassificationModel (uid=dtc\_0f9b93f0b062) of depth 4 with 13 nodes

If (feature 1 <= 1.9)

Predict: 0.0

Else (feature 1 > 1.9)

If (feature 3 <= 1.6)

If (feature 1 <= 4.9)

Predict: 1.0

Else (feature 1 > 4.9)

If (feature 0 <= 2.6)

Predict: 2.0

Else (feature 0 > 2.6)

Predict: 1.0

Else (feature 3 > 1.6)

If (feature 1 <= 4.8)

If (feature 0 <= 3.0)

Predict: 2.0

Else (feature 0 > 3.0)

Predict: 1.0

Else (feature 1 > 4.8)

Predict: 2.0

\*/

}

}

## 1.3模型标签开发

在标签管理平台新建标签（业务标签和属性标签），构建标签模型，预测用户标签值，结合标签规则，给每个用户打标签。

### 1.3.1新建标签

新建业务（4级）标签：用户购物性别标签，相关字段信息如下：

标签名称：用户购物性别

标签分类：电商-某商城-行为属性

更新周期：1周

业务含义：用户购物性别分为男、女及中性（未知）

标签规则：

inType=hbase

zkHosts=bigdata-cdh01.itcast.cncong

zkPort=2181

hbaseTable=tbl\_goods

family=detail

selectFieldNames=cordersn,ogcolor,producttype

程序入口：

cn.itcast.tags.models.ml.UsgModel

算法名称：

DecisionTree

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【男】

标签名称：男

标签含义：用户购物性别为男

标签规则：0

2）、属性值【女】

标签名称：女

标签含义：用户购物性别为女

标签规则：1

3）、属性值【中性】

标签名称：中性

标签含义：用户购物性别未知，可能是男，也肯能时女

标签规则：-1

新建标签（业务标签和属性标签）插入SQL语句如下：

INSERT INTO `tbl\_basic\_tag` VALUES ('381', '用户购物性别', null,

'inType=hbase\nzkHosts=bigdatacdh01.

itcast.cn\nzkPort=2181\nhbaseTable=tbl\_goods\nfamily=detail\nsele

ctFieldNames=cordersn,ogcolor,producttype', null, '4', '316', '2019-12-

09 17:52:44', '2019-12-09 17:52:44', null, null);

INSERT INTO `tbl\_model` VALUES ('11', '381', 'DecisionTree',

'cn.itcast.tags.models.ml.UsgModel', 'hdfs://bigdatacdh01.

itcast.cn:8020/apps/temp/jars/8cec197c-617c-40db-8150-

8a3dd3734a80.jar', '2,2019-12-09 08:00:00,2029-12-09 08:00:00', '2019-

12-09 17:52:44', '2019-12-09 17:52:44', '4', '--driver-memory 512m --

executor-memory 512m --num-executors 1 --executor-cores 1');

INSERT INTO `tbl\_basic\_tag` VALUES ('382', '男', null, '0', null, '5',

'381', '2019-12-09 17:53:21', '2019-12-09 17:53:21', null, null);

INSERT INTO `tbl\_basic\_tag` VALUES ('383', '女', null, '1', null, '5',

'381', '2019-12-09 17:53:27', '2019-12-09 17:53:27', null, null);

### 1.3.2模型开发

继承基类AbstractModel ，实现标签计算方法doTag 。

1、商品表数据中字段值：

3597241 column=detail:cordersn, timestamp=1575267957780,

value=jd\_14091818005983607

3597241 column=detail:ogcolor, timestamp=1575267957780,

value=\xE7\x99\xBD\x89\xB2

3597241 column=detail:producttype, timestamp=1575267957780,

value=\xE7xA4\xE7\xAE\xB1

2、订单表数据中字段值：

需要将订单商品中订单号cordersn关联到订单表中订单号ordersn，获取到对应的会员

ID：memberid。

10 column=detail:memberid, timestamp=1575729720218, value=13823431

10 column=detail:ordersn, timestamp=1575729720218,

value=gome\_792756751164275

#### 主函数编写

object USGTest1 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 = 56  
  
 /\*\*  
 \* 开始计算  
 \* @param fiveDF MySQL中的5级规则 id,rule  
 \* inType=HBase##zkHosts=192.168.10.20##zkPort=2181##hbaseTable=tbl\_goods##family=detail##selectFields=cOrderSn,ogColor,productType  
 \* @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 = {  
 println("fiveDF show result:")  
 fiveDF.show(10)  
 println("hbaseDF show result:")  
 hbaseDF.show(10)  
 val goodsDF = hbaseDF

#### （1）订单表数据读取

读取订单表数据代码如下所示：

val spark: SparkSession = businessDF.sparkSession

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

import spark.implicits.\_

// 1. 获取订单表数据tbl\_orders，与订单商品表数据关联获取会员ID

import cn.itcast.up.bean.HBaseMeta

val ordersDF: DataFrame = spark.read  
 .format("cn.itcast.up.tools.HBaseSource")  
 .option(HBaseMeta.ZKHOSTS, "bd001")  
 .option(HBaseMeta.ZKPORT, "2181")  
 .option(HBaseMeta.HBASETABLE, "tbl\_orders")  
 .option(HBaseMeta.FAMILY, "detail")  
 .option(HBaseMeta.SELECTFIELDS, "memberId,orderSn")  
 .load()  
//ordersDF.show(10,false)  
//ordersDF.printSchema()

#### **（2）字典表数据**

获取订单商品表数据以后，需要将颜色color和商品product转换为对应ID，需要读取维度表数据，为了简化开发，直接进行判断映射，如下所示：

//2.特征选取(实际中可以选取商品的众多属性再进行主成份分析PCA降维,我们这里考虑到时间原因,只选取2个)  
//颜色ID应该来源于字典表,这里简化处理  
val color: Column = functions  
 .when('ogColor.equalTo("银色"), 1)  
 .when('ogColor.equalTo("香槟金色"), 2)  
 .when('ogColor.equalTo("黑色"), 3)  
 .when('ogColor.equalTo("白色"), 4)  
 .when('ogColor.equalTo("梦境极光【卡其金】"), 5)  
 .when('ogColor.equalTo("梦境极光【布朗灰】"), 6)  
 .when('ogColor.equalTo("粉色"), 7)  
 .when('ogColor.equalTo("金属灰"), 8)  
 .when('ogColor.equalTo("金色"), 9)  
 .when('ogColor.equalTo("乐享金"), 10)  
 .when('ogColor.equalTo("布鲁钢"), 11)  
 .when('ogColor.equalTo("月光银"), 12)  
 .when('ogColor.equalTo("时尚光谱【浅金棕】"), 13)  
 .when('ogColor.equalTo("香槟色"), 14)  
 .when('ogColor.equalTo("香槟金"), 15)  
 .when('ogColor.equalTo("灰色"), 16)  
 .when('ogColor.equalTo("樱花粉"), 17)  
 .when('ogColor.equalTo("蓝色"), 18)  
 .when('ogColor.equalTo("金属银"), 19)  
 .when('ogColor.equalTo("玫瑰金"), 20)  
 .otherwise(0)  
 .alias("color")  
//类型ID应该来源于字典表,这里简化处理  
val productType: Column = functions  
 .when('productType.equalTo("4K电视"), 9)  
 .when('productType.equalTo("Haier/海尔冰箱"), 10)  
 .when('productType.equalTo("Haier/海尔冰箱"), 11)  
 .when('productType.equalTo("LED电视"), 12)  
 .when('productType.equalTo("Leader/统帅冰箱"), 13)  
 .when('productType.equalTo("冰吧"), 14)  
 .when('productType.equalTo("冷柜"), 15)  
 .when('productType.equalTo("净水机"), 16)  
 .when('productType.equalTo("前置过滤器"), 17)  
 .when('productType.equalTo("取暖电器"), 18)  
 .when('productType.equalTo("吸尘器/除螨仪"), 19)  
 .when('productType.equalTo("嵌入式厨电"), 20)  
 .when('productType.equalTo("微波炉"), 21)  
 .when('productType.equalTo("挂烫机"), 22)  
 .when('productType.equalTo("料理机"), 23)  
 .when('productType.equalTo("智能电视"), 24)  
 .when('productType.equalTo("波轮洗衣机"), 25)  
 .when('productType.equalTo("滤芯"), 26)  
 .when('productType.equalTo("烟灶套系"), 27)  
 .when('productType.equalTo("烤箱"), 28)  
 .when('productType.equalTo("燃气灶"), 29)  
 .when('productType.equalTo("燃气热水器"), 30)  
 .when('productType.equalTo("电水壶/热水瓶"), 31)  
 .when('productType.equalTo("电热水器"), 32)  
 .when('productType.equalTo("电磁炉"), 33)  
 .when('productType.equalTo("电风扇"), 34)  
 .when('productType.equalTo("电饭煲"), 35)  
 .when('productType.equalTo("破壁机"), 36)  
 .when('productType.equalTo("空气净化器"), 37)  
 .otherwise(0)  
 .alias("productType")

#### （3）数据标注

数据标注：根据运营的前期的统计分析对数据进行性别类别的标注，目标是根据运营则

标注的部分数据,进行决策树模型的训练，以后来了新的数据,模型就可以判断出购物性别是男还是女。购买商品某些颜色为女性用户，作为标注商品性别的条件：

val label: Column = functions  
 .when('ogColor.equalTo("樱花粉")  
 .or('ogColor.equalTo("白色"))  
 .or('ogColor.equalTo("香槟色"))  
 .or('ogColor.equalTo("香槟金"))  
 .or('productType.equalTo("料理机"))  
 .or('productType.equalTo("挂烫机"))  
 .or('productType.equalTo("吸尘器/除螨仪")), 1) //女  
 .otherwise(0)//男  
 .alias("gender")//决策树预测label

#### （3）数据合并

订单数据与商品数据关联，获取数据集（标注后），代码如下：

//4.将数据表进行合并  
 //最终需要找到用户和用户所购买的所有商品,进行训练,找到商品和性别之间的关系  
 val source = goodsDF.select('cOrderSn as "orderSn", color, productType, label)  
 .join(ordersDF, "orderSn")  
 .select('memberId as "userId", 'color, 'productType, 'gender)  
 //source.show(10,false)  
/\*  
+---------+-----+-----------+------+  
|userId |color|productType|gender|  
+---------+-----+-----------+------+  
|13823535 |16 |0 |0 |  
|13823535 |1 |24 |0 |  
|13823535 |7 |30 |0 |  
|13823391 |10 |14 |0 |  
|4034493 |9 |12 |0 |  
|13823683 |8 |17 |0 |  
|62 |9 |15 |0 |  
|4035201 |8 |12 |0 |  
|13823449 |10 |0 |0 |  
|138230919|12 |15 |0 |  
+---------+-----+-----------+------+  
 \*/

#### 特征工程

//机器学习部分  
//1.类别处理  
val stringIndexerModel: StringIndexerModel = new StringIndexer()  
.setInputCol("gender")  
.setOutputCol("label").fit(source)  
  
//2.特征向量化  
val vectorAssembler: VectorAssembler = new VectorAssembler()  
 .setInputCols(Array("color", "productType"))  
 .setOutputCol("features")  
val featureDF: DataFrame = vectorAssembler.transform(source)  
  
//3.对特征进行索引,大于3个不同的值的特征被视为连续特征  
//VectorIndexer是对数据集特征向量中的类别(离散值)特征(index categorical features categorical features)进行编号。  
//它能够自动判断那些特征是离散值型的特征，并对他们进行编号，具体做法是通过设置一个maxCategories，  
//特征向量中某一个特征不重复取值个数小于maxCategories，则被重新编号为0～K（K<=maxCategories-1）。  
//某一个特征不重复取值个数大于maxCategories，则该特征视为连续值，不会重新编号（不会发生任何改变）  
//主要作用：提高决策树或随机森林等ML方法的分类效果  
val featureVectorIndexer: VectorIndexerModel = new VectorIndexer()  
 .setInputCol("features")  
 .setOutputCol("featureIndexed")  
 .setMaxCategories(3)  
 .fit(featureDF)

#### 构建模型

//4.构建分类器  
val decisionTreeClassification: DecisionTreeClassifier = new DecisionTreeClassifier()  
 .setFeaturesCol("featureIndexed")  
 .setPredictionCol("predict")  
 .setMaxDepth(5)  
 //.setImpurity("gini")

#### 模型预测及评价

//5.类别还原  
val indexToString: IndexToString = new IndexToString()  
 .setInputCol("label")  
 .setOutputCol("gender\_converted")  
  
//6.数据集划分  
val Array(trainingData, testData) = source.randomSplit(Array(0.8,0.2))  
  
//7.构建Pipeline并训练模型  
val pipeline: Pipeline = new Pipeline().setStages(Array(stringIndexerModel,vectorAssembler,featureVectorIndexer,decisionTreeClassification,indexToString))  
val pipelineModel: PipelineModel = pipeline.fit(trainingData)  
  
//8.预测  
val testReusltDF: DataFrame = pipelineModel.transform(testData)  
val trainingReusltDF: DataFrame = pipelineModel.transform(trainingData)  
  
//9.决策过程输出  
val classificationModel: DecisionTreeClassificationModel = pipelineModel.stages(3).asInstanceOf[DecisionTreeClassificationModel]  
println("决策过程如下:\n"+classificationModel.toDebugString)

#### 模型评价

//10.评价  
this.evaluateACCAndAUC(trainingReusltDF,testReusltDF)

调用函数

def evaluateACCAndAUC(predictTrainDF: DataFrame,predictTestDF: DataFrame): Unit = {  
 // 1. ACC  
 val accEvaluator = new MulticlassClassificationEvaluator()  
 .setPredictionCol("predict")  
 .setLabelCol("label")  
 .setMetricName("accuracy")//精准度  
  
 val trainAcc: Double = accEvaluator.evaluate(predictTrainDF)  
 val testAcc: Double = accEvaluator.evaluate(predictTestDF)  
 println(s"训练集上的 ACC 是 : $trainAcc")  
 println(s"测试集上的 ACC 是 : $testAcc")  
 //训练集上的 ACC 是 : 0.9659155462980497  
 //测试集上的 ACC 是 : 0.9666924864446166  
  
 // 2. AUC  
 val trainRdd: RDD[(Double, Double)] = predictTrainDF.select("label", "predict").rdd  
 .map(row => (row.getAs[Double](0), row.getAs[Double](1)))  
 val testRdd: RDD[(Double, Double)] = predictTestDF.select("label", "predict").rdd  
 .map(row => (row.getAs[Double](0), row.getAs[Double](1)))  
  
 val trainAUC: Double = new BinaryClassificationMetrics(trainRdd).areaUnderROC()  
 val testAUC: Double = new BinaryClassificationMetrics(testRdd).areaUnderROC()  
 println(s"训练集上的 AUC 是 : $trainAUC")  
 println(s"测试集上的 AUC 是 : $testAUC")  
 //训练集上的 AUC 是 : 0.955163396745856  
 //测试集上的 AUC 是 : 0.955110198789974  
  
}

#### 返回结果并保存

//11.返回结果并保存

选择userId，判断prediction预测为男或女，根据userId进行聚合，使用count，sum进行统计

val allResult: Dataset[Row] = testReusltDF.union(trainingReusltDF)  
 val tempDF: DataFrame = allResult.select('userId,  
 when('predict === 0, 1).otherwise(0).as("male"), //计算每个用户所有订单中的男性商品的订单数  
 when('predict === 1, 1).otherwise(0).as("female")) //计算每个用户所有订单中的女性商品的订单数  
 .groupBy('userId)  
 .agg(  
 count('userId) cast DoubleType as "total", //总共预测多少次  
 sum('male) cast DoubleType as "maleCount", //预测为男的次数  
 sum('female) cast DoubleType as "femaleCount") //预测为女的次数  
 .select('userId, 'total, 'maleCount, 'femaleCount)  
 tempDF.show(10,false)  
/\*  
+---------+-----+---------+-----------+  
|userId |total|maleCount|femaleCount|  
+---------+-----+---------+-----------+  
|4033473 |13.0 |12.0 |1.0 |  
|13823083 |17.0 |13.0 |4.0 |  
|13823681 |3.0 |3.0 |0.0 |  
|138230919|5.0 |3.0 |2.0 |  
|13822725 |7.0 |6.0 |1.0 |  
|4033575 |9.0 |8.0 |1.0 |  
|4034191 |6.0 |5.0 |1.0 |  
|13823153 |9.0 |6.0 |3.0 |  
|13822841 |3.0 |3.0 |0.0 |  
|13823431 |8.0 |4.0 |4.0 |  
+---------+-----+---------+-----------+  
 \*/

#### 标签预测

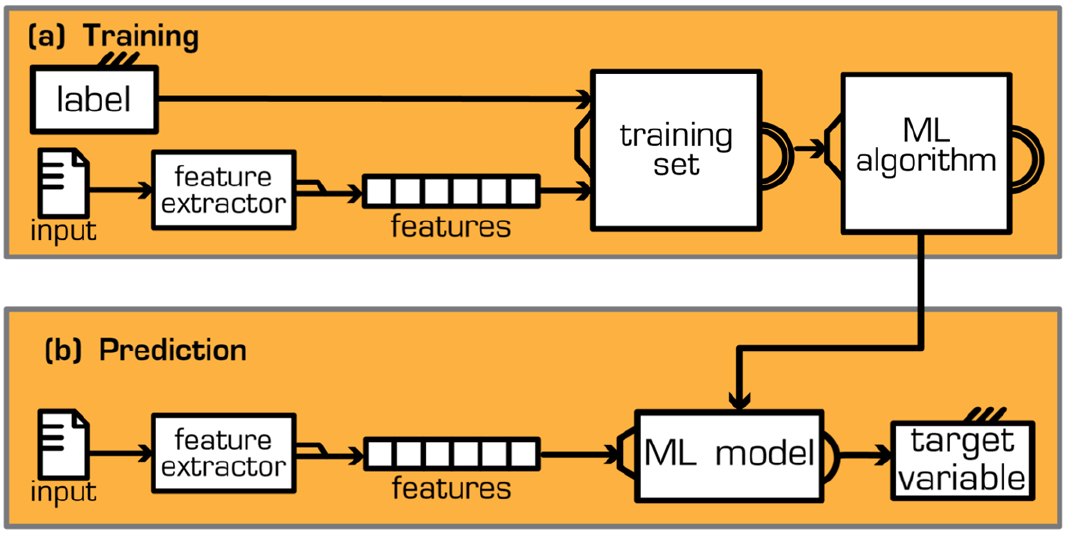
//12.上面我们就得到了用户在该网站被预测购物性别的总次数,以及被预测为男和女各多少次  
 //接下来我们要结合该用户所有的预测结果真正的给该用户打上购物性别标签  
 //预测规则A:每个订单的男性商品>=80%则认定为该订单的用户为男，或女商品比例达到80%则认定为该订单的用户为女；  
 //由于是家电产品，一个订单中通常只有一个商品。调整规则A为规则B：  
 //预测规则B:计算每个用户近半年内所有订单中的男性商品超过60%则认定该用户为男，或近半年内所有订单中的女性品超过60%则认定该用户为女  
 //那么现在我们要做的事情可以简化为:根据该用户的total/maleCount/femaleCount判断用户应该被打上哪个标签  
 //12.1 将5级标签转为map方便后面使用  
 /\*  
+---+----+  
|id |rule|  
+---+----+  
|57 |0 |  
|58 |1 |  
|59 |-1 |  
+---+----+  
 \*/  
 //Map[rule, tagId]  
 val ruleMap: Map[String, Long] = fiveDF.collect().map(row=>(row.getString(1),row.getLong(0))).toMap  
  
  
 val getGenderTag = udf((total:Double,maleCount:Double,femaleCount:Double)=>{  
 //预测规则B:计算每个用户近半年内所有订单中的男性商品超过60%则认定该用户为男，或近半年内所有订单中的女性品超过60%则认定该用户为女  
 val maleRate: Double = maleCount / total  
 val femaleRate: Double = femaleCount/ total  
 if (maleRate>= 0.6){  
 ruleMap("0")  
 }else if(femaleRate>=0.6){  
 ruleMap("1")  
 }else{  
 ruleMap("-1")  
 }  
 })

#### 自定义UDF完成tag转换

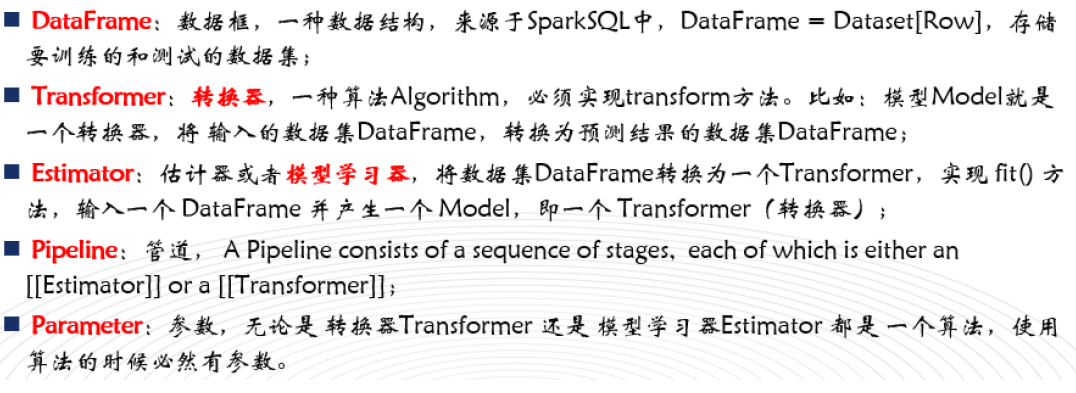
//12.2使用自定义UDF完成tag转换  
val newDF: DataFrame = tempDF.select('userId,getGenderTag('total,'maleCount,'femaleCount).as("tagIds"))  
newDF.show(10,false)  
  
~~println(new Date().toLocaleString)//需要20多分钟~~  
newDF

#### （10）Pipeline构建模型

使用数据集，构建决策树分类模型，依据购买的商品和颜色预测其性别，为后续判断用户购物性别做依据。由于需要对数据集进行特征工程，所以使用Pipeline管道进行封装，获取管道模型PipelineModel。



Pipeline管道模型中相关概念如下：



封装函数代码，其中涉及分类模型评估，具体代码如下：

/\*\*

\* 使用决策树分类算法训练模型，返回PipelineModel模型

\*

\* @return

\*/

def trainPipelineModel(dataframe: DataFrame): PipelineModel = {

// 数据划分为训练数据集和测试数据集

val Array(trainingDF, testingDF) =

dataframe.randomSplit(Array(0.8, 0.2), seed = 123)

// a. 特征向量化

val assembler = new VectorAssembler()

.setInputCols(Array("color", "product"))

.setOutputCol("features")

// b. 类别特征进行索引

val vectorIndexer = new VectorIndexer()

.setInputCol("features")

.setOutputCol("indexer\_features")

.setMaxCategories(30)

// c. 构建决策树分类器

val dtc: DecisionTreeClassifier = new DecisionTreeClassifier()

.setFeaturesCol("indexer\_features")

.setLabelCol("label")

.setPredictionCol("prediction")

.setMaxDepth(5) // 树的深度

.setMaxBins(32) // 树的叶子数目

.setImpurity("gini") // 基尼系数

// d. 构建Pipeline管道流实例对象

val pipeline: Pipeline = new Pipeline().setStages(

Array(assembler, vectorIndexer, dtc)

)

// e. 训练模型

val pipelineModel: PipelineModel = pipeline.fit(trainingDF)

// 训练数据预测评估

//println(s"training accuracy =

${modelEvaluate(pipelineModel.transform(trainingDF))}")

// f. 模型评估

val predictionDF: DataFrame =

pipelineModel.transform(testingDF)

//predictionDF.show(100, truncate = false)

//println(s"accuracy = ${modelEvaluate(predictionDF)}")

// 返回模型

pipelineModel

}

/\*\*

\* 模型评估，返回计算分类指标值

\*

\* @param dataframe 预测结果的数据集

\* @return

\*/

def modelEvaluate(dataframe: DataFrame): Double = {

// a. 构建多分类分类器

val evaluator = new MulticlassClassificationEvaluator()

.setLabelCol("label")

.setPredictionCol("prediction")

// 指标名称，支持：f1、weightedPrecision、weightedRecall、

accuracy

.setMetricName("accuracy")

// b. 计算评估指标

val accuracy: Double = evaluator.evaluate(dataframe)

// c. 返回指标

accuracy

}

### 1.3.3打标签

获取模型后，预测每个订单商品数据中购物性别，后期结合属性标签规则rule进行打标

签，代码如下：

// 5. 构建模型（PipelineModel）

val pipelineModel: PipelineModel =

trainPipelineModel(genderDF)

// 6. 模型预测标签

val predictionDF: DataFrame =

pipelineModel.transform(genderDF)

//val predictionDF = genderDF.select($"userId",

$"label".as("prediction"))

//predictionDF.printSchema()

//predictionDF.show(10, truncate = false)

// 属性标签数据中规则rule

val rulesMap: Map[String, Long] = basicTagDF

.filter($"level".equalTo(5))

.select($"rule", $"id".as("tagId"))

.as[(String, Long)]

.rdd

.collectAsMap().toMap

// 7. 统计每个会员购买商品的性别次数，最终确定用户购物性别

val usgDF: DataFrame = predictionDF

.select(

$"userId", //

when($"prediction" === 0, 1)

.otherwise(0).as("male"), // 男

when($"prediction" === 1, 1)

.otherwise(0).as("female") // 女

)

.groupBy($"userId")

.agg(

count($"userId").as("total"), // 用户购物次数

sum($"male").as("maleTotal"), //预测为男性的次数

sum($"female").as("femaleTotal") //预测为女性的次数

)

依据每个用户购买商品中属于男性商品和女性商品占比，确定购物性，计算规则如下：

计算每个用户近半年内所有订单中的男性商品超过60%则认定该用户为男，或近半年内所有

订单中的女性品超过60%则认定该用户为女。

自定义UDF函数，实现上述功能，代码如下：

// 自定义UDF函数

/\*

计算每个用户近半年内所有订单中的男性商品超过60%则认定该用户为男，

或近半年内所有订单中的女性品超过60%则认定该用户为女

\*/

val gender\_tag: UserDefinedFunction = udf(

(total: Long, maleCount: Double, femaleCount: Double) => {

val maleRate: Double = maleCount / total

val femaleRate: Double = femaleCount / total

if (maleRate >= 0.6){

rulesMap("0")

}else if(femaleRate >= 0.6){

rulesMap("1")

}else{

rulesMap("-1")

}

}

)

// 根据该用户的total/maleCount/femaleCount判断用户应该被打上哪个标签

val modelDF = usgDF.select(

$"userId", //

gender\_tag($"total", $"maleTotal",

$"femaleTotal").as("tagId")

)

//modelDF.printSchema()

//modelDF.show(200, truncate = false)

### 1.3.4未使用pipeline构建代码

**package** cn.itcast.up.ml.mltest32  
  
**import** cn.itcast.up.base.BaseModel  
**import** cn.itcast.up.bean.HBaseMeta  
**import** org.apache.spark.ml.classification.DecisionTreeClassifier  
**import** org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorAssembler, VectorIndexer}  
**import** org.apache.spark.sql.{Column, DataFrame, functions}  
  
*/\*\*  
 \* BaseModel 使用基础的算法达到USG模型的创建  
 \*/***object** USGModelTest1 **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 = 56  
  
 */\*\*  
 \* 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 = {  
 **import** *spark*.implicits.\_  
 *//1.加载数据  
 //fiveDF.show(10,false)  
 //fiveDF.printSchema()  
 /\*  
 +---+----+  
|id |rule|  
+---+----+  
|57 |0 |  
|58 |1 |  
|59 |-1 |  
+---+----+  
 \*/* **val** goodsDF: DataFrame = hbaseDF  
 *//goodsDF.show(10,false)  
 //goodsDF.printSchema()  
 /\*  
 +----------------------+---------+-----------+  
|cOrderSn |ogColor |productType|  
+----------------------+---------+-----------+  
|jd\_14091818005983607 |白色 |烤箱 |  
|jd\_14091317283357943 |香槟金 |冰吧 |  
|jd\_14092012560709235 |香槟金色 |净水机 |  
|rrs\_15234137 |梦境极光【布朗灰】|烤箱 |  
|suning\_790750687478116|梦境极光【卡其金】|4K电视 |  
|rsq\_805093707860210 |黑色 |烟灶套系 |  
|jd\_14090910361908941 |黑色 |智能电视 |  
|jd\_14091823464864679 |香槟金色 |燃气灶 |  
|jd\_14091817311906413 |银色 |滤芯 |  
|suning\_804226647488814|玫瑰金 |电饭煲 |  
+----------------------+---------+-----------+  
only showing top 10 rows  
 \*/* **val** ordersDF: DataFrame = *spark*.read  
 .format(**"cn.itcast.up.tools.HBaseSource"**)  
 .option(HBaseMeta.*ZKHOSTS*, **"bd001"**)  
 .option(HBaseMeta.*ZKPORT*, **"2181"**)  
 .option(HBaseMeta.*HBASETABLE*, **"tbl\_orders"**)  
 .option(HBaseMeta.*FAMILY*, **"detail"**)  
 .option(HBaseMeta.*SELECTFIELDS*, **"memberId,orderSn"**)  
 .load()  
 *//ordersDF.show(10,false)  
 //ordersDF.printSchema()  
 /\*  
 +---------+-------------------+  
|memberId |orderSn |  
+---------+-------------------+  
|13823431 |ts\_792756751164275 |  
|4035167 |D14090106121770839 |  
|4035291 |D14090112394810659 |  
|4035041 |fx\_787749561729045 |  
|13823285 |D14092120154435903 |  
|4034219 |D14092120155620305 |  
|138230939|top\_810791455519102|  
|4035083 |D14092120161884409 |  
|138230935|D14092120162313538 |  
|13823231 |D14092120162378713 |  
+---------+-------------------+  
 \*/  
 //2.特征选取(实际中可以选取商品的众多属性再进行主成份分析PCA降维,我们这里考虑到时间原因,只选取2个)  
 //颜色ID应该来源于字典表,这里简化处理* **val** color: Column = functions  
 .*when*(**'ogColor**.equalTo(**"银色"**), 1)  
 .when(**'ogColor**.equalTo(**"香槟金色"**), 2)  
 .when(**'ogColor**.equalTo(**"黑色"**), 3)  
 .when(**'ogColor**.equalTo(**"白色"**), 4)  
 .when(**'ogColor**.equalTo(**"梦境极光【卡其金】"**), 5)  
 .when(**'ogColor**.equalTo(**"梦境极光【布朗灰】"**), 6)  
 .when(**'ogColor**.equalTo(**"粉色"**), 7)  
 .when(**'ogColor**.equalTo(**"金属灰"**), 8)  
 .when(**'ogColor**.equalTo(**"金色"**), 9)  
 .when(**'ogColor**.equalTo(**"乐享金"**), 10)  
 .when(**'ogColor**.equalTo(**"布鲁钢"**), 11)  
 .when(**'ogColor**.equalTo(**"月光银"**), 12)  
 .when(**'ogColor**.equalTo(**"时尚光谱【浅金棕】"**), 13)  
 .when(**'ogColor**.equalTo(**"香槟色"**), 14)  
 .when(**'ogColor**.equalTo(**"香槟金"**), 15)  
 .when(**'ogColor**.equalTo(**"灰色"**), 16)  
 .when(**'ogColor**.equalTo(**"樱花粉"**), 17)  
 .when(**'ogColor**.equalTo(**"蓝色"**), 18)  
 .when(**'ogColor**.equalTo(**"金属银"**), 19)  
 .when(**'ogColor**.equalTo(**"玫瑰金"**), 20)  
 .otherwise(0)  
 .alias(**"color"**)  
 *//类型ID应该来源于字典表,这里简化处理* **val** productType: Column = functions  
 .*when*(**'productType**.equalTo(**"4K电视"**), 9)  
 .when(**'productType**.equalTo(**"Haier/海尔冰箱"**), 10)  
 .when(**'productType**.equalTo(**"Haier/海尔冰箱"**), 11)  
 .when(**'productType**.equalTo(**"LED电视"**), 12)  
 .when(**'productType**.equalTo(**"Leader/统帅冰箱"**), 13)  
 .when(**'productType**.equalTo(**"冰吧"**), 14)  
 .when(**'productType**.equalTo(**"冷柜"**), 15)  
 .when(**'productType**.equalTo(**"净水机"**), 16)  
 .when(**'productType**.equalTo(**"前置过滤器"**), 17)  
 .when(**'productType**.equalTo(**"取暖电器"**), 18)  
 .when(**'productType**.equalTo(**"吸尘器/除螨仪"**), 19)  
 .when(**'productType**.equalTo(**"嵌入式厨电"**), 20)  
 .when(**'productType**.equalTo(**"微波炉"**), 21)  
 .when(**'productType**.equalTo(**"挂烫机"**), 22)  
 .when(**'productType**.equalTo(**"料理机"**), 23)  
 .when(**'productType**.equalTo(**"智能电视"**), 24)  
 .when(**'productType**.equalTo(**"波轮洗衣机"**), 25)  
 .when(**'productType**.equalTo(**"滤芯"**), 26)  
 .when(**'productType**.equalTo(**"烟灶套系"**), 27)  
 .when(**'productType**.equalTo(**"烤箱"**), 28)  
 .when(**'productType**.equalTo(**"燃气灶"**), 29)  
 .when(**'productType**.equalTo(**"燃气热水器"**), 30)  
 .when(**'productType**.equalTo(**"电水壶/热水瓶"**), 31)  
 .when(**'productType**.equalTo(**"电热水器"**), 32)  
 .when(**'productType**.equalTo(**"电磁炉"**), 33)  
 .when(**'productType**.equalTo(**"电风扇"**), 34)  
 .when(**'productType**.equalTo(**"电饭煲"**), 35)  
 .when(**'productType**.equalTo(**"破壁机"**), 36)  
 .when(**'productType**.equalTo(**"空气净化器"**), 37)  
 .otherwise(0)  
 .alias(**"productType"**)  
  
 *//3.数据标注(根据运营的前期的统计分析对数据进行性别类别的标注)  
 //目标是根据运营规则标注的部分数据,进行决策树模型的训练,以后来了新的数据,模型就可以判断出购物性别是男还是女* **val** label: Column = functions  
 .*when*(**'ogColor**.equalTo(**"樱花粉"**)  
 .or(**'ogColor**.equalTo(**"白色"**))  
 .or(**'ogColor**.equalTo(**"香槟色"**))  
 .or(**'ogColor**.equalTo(**"香槟金"**))  
 .or(**'productType**.equalTo(**"料理机"**))  
 .or(**'productType**.equalTo(**"挂烫机"**))  
 .or(**'productType**.equalTo(**"吸尘器/除螨仪"**)), 1) *//女* .otherwise(0) *//男* .alias(**"gender"**) *//决策树预测label  
  
 //4-join两个数据源---构成了基本的数据源  
 // |cOrderSn |ogColor |productType|  
 // |memberId |orderSn |* **val** source = goodsDF.select(**'cOrderSn** as (**"orderSn"**), color, productType, label)  
 .join(ordersDF, **"orderSn"**)  
 .select(**'memberId** as (**"userid"**), **'color**, **'productType**, **'gender**)  
 source.show(10, **false**)  
 *// +---------+-----+-----------+------+  
 // |userid |color|productType|gender|  
 // +---------+-----+-----------+------+  
 // |13823535 |16 |0 |0 |  
 // |13823535 |1 |24 |0 |  
 // |13823535 |7 |30 |0 |  
 // |13823391 |10 |14 |0 |  
 // |4034493 |9 |12 |0 |  
 // |13823683 |8 |17 |0 |  
 // |62 |9 |15 |0 |  
 // |4035201 |8 |12 |0 |  
 // |13823449 |10 |0 |0 |  
 // |138230919|12 |15 |0 |  
 // +---------+-----+-----------+------+  
 // only showing top 10 rows* source.printSchema()  
 *// root  
 // |-- userid: string (nullable = true)  
 // |-- color: integer (nullable = false)  
 // |-- productType: integer (nullable = false)  
 // |-- gender: integer (nullable = false)  
 //5-特征工程  
 //- StringIndexer对gender进行处理 0 1----> 0 1* **val** stringIndexer = **new** StringIndexer()  
 .setInputCol(**"gender"**)  
 .setOutputCol(**"classlabel"**)  
 **val** stringIndexerModel = stringIndexer.fit(source)  
 **val** sourceStrResult = stringIndexerModel.transform(source)  
 *//- VectorAssemble进行特征组合----color+productType* **val** assembler = **new** VectorAssembler().setInputCols(*Array*(**"color"**, **"productType"**)).setOutputCol(**"features"**)  
 **val** vecResult = assembler.transform(sourceStrResult)  
 *//- VectorIndexer进行类别值的索引* **val** vectorIndexer = **new** VectorIndexer()  
 .setInputCol(**"features"**)  
 .setOutputCol(**"index\_features"**)  
 .setMaxCategories(3)  
 **val** indexerModel = vectorIndexer.fit(vecResult)  
 **val** indexerResult = indexerModel.transform(vecResult)  
 *//6- 构建模型  
 // - 新建决策树算法* **val** treeClassifier = **new** DecisionTreeClassifier()  
 .setImpurity(**"entropy"**)  
 .setLabelCol(**"classlabel"**)  
 .setFeaturesCol(**"index\_features"**)  
 .setPredictionCol(**"prces"**)  
 **val** model = treeClassifier.fit(indexerResult)  
 **val** resultS = model.transform(indexerResult)  
 *//- IndexTOString将转化后的值进行翻转* **val** indexToString = **new** IndexToString().setLabels(stringIndexerModel.labels)  
 .setInputCol(**"prces"**).setOutputCol(**"before\_class\_label"**)  
 indexToString.transform(resultS).show(**false**)  
 *//7- 模型预测  
 // - 模型对于测试集数据预测  
 // +---------+-----+-----------+------+----------+-----------+--------------+-------------+---------------------------------------+-----+------------------+  
 // |userid |color|productType|gender|classlabel|features |index\_features|rawPrediction|probability |prces|before\_class\_label|  
 // +---------+-----+-----------+------+----------+-----------+--------------+-------------+---------------------------------------+-----+------------------+  
 // |13823535 |16 |0 |0 |0.0 |[16.0,0.0] |[16.0,0.0] |[157.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |13823535 |1 |24 |0 |0.0 |[1.0,24.0] |[1.0,24.0] |[334.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |13823535 |7 |30 |0 |0.0 |[7.0,30.0] |[7.0,30.0] |[1065.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |13823391 |10 |14 |0 |0.0 |[10.0,14.0]|[10.0,14.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |4034493 |9 |12 |0 |0.0 |[9.0,12.0] |[9.0,12.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |13823683 |8 |17 |0 |0.0 |[8.0,17.0] |[8.0,17.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |62 |9 |15 |0 |0.0 |[9.0,15.0] |[9.0,15.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |4035201 |8 |12 |0 |0.0 |[8.0,12.0] |[8.0,12.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |13823449 |10 |0 |0 |0.0 |[10.0,0.0] |[10.0,0.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |138230919|12 |15 |0 |0.0 |[12.0,15.0]|[12.0,15.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |80 |9 |28 |0 |0.0 |[9.0,28.0] |[9.0,28.0] |[1065.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |4035113 |6 |13 |0 |0.0 |[6.0,13.0] |[6.0,13.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |4035123 |12 |0 |0 |0.0 |[12.0,0.0] |[12.0,0.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |4035123 |1 |0 |0 |0.0 |[1.0,0.0] |[1.0,0.0] |[485.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |23 |8 |13 |0 |0.0 |[8.0,13.0] |[8.0,13.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |4035247 |16 |36 |0 |0.0 |[16.0,36.0]|[16.0,36.0] |[170.0,31.0] |[0.845771144278607,0.15422885572139303]|0.0 |0 |  
 // |13823563 |18 |27 |0 |0.0 |[18.0,27.0]|[18.0,27.0] |[395.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |91 |5 |10 |0 |0.0 |[5.0,10.0] |[5.0,10.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |91 |7 |10 |0 |0.0 |[7.0,10.0] |[7.0,10.0] |[1483.0,0.0] |[1.0,0.0] |0.0 |0 |  
 // |91 |15 |0 |1 |1.0 |[15.0,0.0] |[15.0,0.0] |[0.0,678.0] |[0.0,1.0] |1.0 |1 |  
 // +---------+-----+-----------+------+----------+-----------+--------------+-------------+---------------------------------------+-----+------------------+* **null** }  
}

### 1.3.5 Pipeline构建完整代码

上述综合完整代码如下：

**package** cn.itcast.up.ml  
**import** java.util.Date  
  
**import** cn.itcast.up.base.BaseModel  
**import** cn.itcast.up.bean.HBaseMeta  
**import** org.apache.spark.ml.{Pipeline, PipelineModel}  
**import** org.apache.spark.ml.classification.{DecisionTreeClassificationModel, DecisionTreeClassifier}  
**import** org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator  
**import** org.apache.spark.ml.feature.\_  
**import** org.apache.spark.mllib.evaluation.BinaryClassificationMetrics  
**import** org.apache.spark.rdd.RDD  
**import** org.apache.spark.sql.\_  
**import** org.apache.spark.sql.types.DoubleType  
  
*/\*\*  
 \* Author itcast  
 \* Date 2019/11/6 11:36  
 \* Desc 使用决策树完成用户购物性别模型  
 \*/***object** USGModel **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 = 56  
  
 */\*\*  
 \* 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 = {  
 **import** org.apache.spark.sql.functions.\_  
 **import** *spark*.implicits.\_  
 *//1.加载数据  
 //fiveDF.show(10,false)  
 //fiveDF.printSchema()  
 /\*  
 +---+----+  
|id |rule|  
+---+----+  
|57 |0 |  
|58 |1 |  
|59 |-1 |  
+---+----+  
 \*/* **val** goodsDF: DataFrame = hbaseDF  
 *//goodsDF.show(10,false)  
 //goodsDF.printSchema()  
 /\*  
 +----------------------+---------+-----------+  
|cOrderSn |ogColor |productType|  
+----------------------+---------+-----------+  
|jd\_14091818005983607 |白色 |烤箱 |  
|jd\_14091317283357943 |香槟金 |冰吧 |  
|jd\_14092012560709235 |香槟金色 |净水机 |  
|rrs\_15234137 |梦境极光【布朗灰】|烤箱 |  
|suning\_790750687478116|梦境极光【卡其金】|4K电视 |  
|rsq\_805093707860210 |黑色 |烟灶套系 |  
|jd\_14090910361908941 |黑色 |智能电视 |  
|jd\_14091823464864679 |香槟金色 |燃气灶 |  
|jd\_14091817311906413 |银色 |滤芯 |  
|suning\_804226647488814|玫瑰金 |电饭煲 |  
+----------------------+---------+-----------+  
only showing top 10 rows  
 \*/* **val** ordersDF: DataFrame = *spark*.read  
 .format(**"cn.itcast.up.tools.HBaseSource"**)  
 .option(HBaseMeta.*ZKHOSTS*, **"bd001"**)  
 .option(HBaseMeta.*ZKPORT*, **"2181"**)  
 .option(HBaseMeta.*HBASETABLE*, **"tbl\_orders"**)  
 .option(HBaseMeta.*FAMILY*, **"detail"**)  
 .option(HBaseMeta.*SELECTFIELDS*, **"memberId,orderSn"**)  
 .load()  
 *//ordersDF.show(10,false)  
 //ordersDF.printSchema()  
 /\*  
 +---------+-------------------+  
|memberId |orderSn |  
+---------+-------------------+  
|13823431 |ts\_792756751164275 |  
|4035167 |D14090106121770839 |  
|4035291 |D14090112394810659 |  
|4035041 |fx\_787749561729045 |  
|13823285 |D14092120154435903 |  
|4034219 |D14092120155620305 |  
|138230939|top\_810791455519102|  
|4035083 |D14092120161884409 |  
|138230935|D14092120162313538 |  
|13823231 |D14092120162378713 |  
+---------+-------------------+  
 \*/  
  
 //2.特征选取(实际中可以选取商品的众多属性再进行主成份分析PCA降维,我们这里考虑到时间原因,只选取2个)  
 //颜色ID应该来源于字典表,这里简化处理* **val** color: Column = functions  
 .*when*(**'ogColor**.equalTo(**"银色"**), 1)  
 .when(**'ogColor**.equalTo(**"香槟金色"**), 2)  
 .when(**'ogColor**.equalTo(**"黑色"**), 3)  
 .when(**'ogColor**.equalTo(**"白色"**), 4)  
 .when(**'ogColor**.equalTo(**"梦境极光【卡其金】"**), 5)  
 .when(**'ogColor**.equalTo(**"梦境极光【布朗灰】"**), 6)  
 .when(**'ogColor**.equalTo(**"粉色"**), 7)  
 .when(**'ogColor**.equalTo(**"金属灰"**), 8)  
 .when(**'ogColor**.equalTo(**"金色"**), 9)  
 .when(**'ogColor**.equalTo(**"乐享金"**), 10)  
 .when(**'ogColor**.equalTo(**"布鲁钢"**), 11)  
 .when(**'ogColor**.equalTo(**"月光银"**), 12)  
 .when(**'ogColor**.equalTo(**"时尚光谱【浅金棕】"**), 13)  
 .when(**'ogColor**.equalTo(**"香槟色"**), 14)  
 .when(**'ogColor**.equalTo(**"香槟金"**), 15)  
 .when(**'ogColor**.equalTo(**"灰色"**), 16)  
 .when(**'ogColor**.equalTo(**"樱花粉"**), 17)  
 .when(**'ogColor**.equalTo(**"蓝色"**), 18)  
 .when(**'ogColor**.equalTo(**"金属银"**), 19)  
 .when(**'ogColor**.equalTo(**"玫瑰金"**), 20)  
 .otherwise(0)  
 .alias(**"color"**)  
 *//类型ID应该来源于字典表,这里简化处理* **val** productType: Column = functions  
 .*when*(**'productType**.equalTo(**"4K电视"**), 9)  
 .when(**'productType**.equalTo(**"Haier/海尔冰箱"**), 10)  
 .when(**'productType**.equalTo(**"Haier/海尔冰箱"**), 11)  
 .when(**'productType**.equalTo(**"LED电视"**), 12)  
 .when(**'productType**.equalTo(**"Leader/统帅冰箱"**), 13)  
 .when(**'productType**.equalTo(**"冰吧"**), 14)  
 .when(**'productType**.equalTo(**"冷柜"**), 15)  
 .when(**'productType**.equalTo(**"净水机"**), 16)  
 .when(**'productType**.equalTo(**"前置过滤器"**), 17)  
 .when(**'productType**.equalTo(**"取暖电器"**), 18)  
 .when(**'productType**.equalTo(**"吸尘器/除螨仪"**), 19)  
 .when(**'productType**.equalTo(**"嵌入式厨电"**), 20)  
 .when(**'productType**.equalTo(**"微波炉"**), 21)  
 .when(**'productType**.equalTo(**"挂烫机"**), 22)  
 .when(**'productType**.equalTo(**"料理机"**), 23)  
 .when(**'productType**.equalTo(**"智能电视"**), 24)  
 .when(**'productType**.equalTo(**"波轮洗衣机"**), 25)  
 .when(**'productType**.equalTo(**"滤芯"**), 26)  
 .when(**'productType**.equalTo(**"烟灶套系"**), 27)  
 .when(**'productType**.equalTo(**"烤箱"**), 28)  
 .when(**'productType**.equalTo(**"燃气灶"**), 29)  
 .when(**'productType**.equalTo(**"燃气热水器"**), 30)  
 .when(**'productType**.equalTo(**"电水壶/热水瓶"**), 31)  
 .when(**'productType**.equalTo(**"电热水器"**), 32)  
 .when(**'productType**.equalTo(**"电磁炉"**), 33)  
 .when(**'productType**.equalTo(**"电风扇"**), 34)  
 .when(**'productType**.equalTo(**"电饭煲"**), 35)  
 .when(**'productType**.equalTo(**"破壁机"**), 36)  
 .when(**'productType**.equalTo(**"空气净化器"**), 37)  
 .otherwise(0)  
 .alias(**"productType"**)  
  
 *//3.数据标注(根据运营的前期的统计分析对数据进行性别类别的标注)  
 //目标是根据运营规则标注的部分数据,进行决策树模型的训练,以后来了新的数据,模型就可以判断出购物性别是男还是女* **val** label: Column = functions  
 .*when*(**'ogColor**.equalTo(**"樱花粉"**)  
 .or(**'ogColor**.equalTo(**"白色"**))  
 .or(**'ogColor**.equalTo(**"香槟色"**))  
 .or(**'ogColor**.equalTo(**"香槟金"**))  
 .or(**'productType**.equalTo(**"料理机"**))  
 .or(**'productType**.equalTo(**"挂烫机"**))  
 .or(**'productType**.equalTo(**"吸尘器/除螨仪"**)), 1) *//女* .otherwise(0)*//男* .alias(**"gender"**)*//决策树预测label  
  
 //4.将数据表进行合并  
 //最终需要找到用户和用户所购买的所有商品,进行训练,找到商品和性别之间的关系* **val** source = goodsDF.select(**'cOrderSn** as **"orderSn"**, color, productType, label)  
 .join(ordersDF, **"orderSn"**)  
 .select(**'memberId** as **"userId"**, **'color**, **'productType**, **'gender**)  
 *//source.show(10,false)  
/\*  
+---------+-----+-----------+------+  
|userId |color|productType|gender|  
+---------+-----+-----------+------+  
|13823535 |16 |0 |0 |  
|13823535 |1 |24 |0 |  
|13823535 |7 |30 |0 |  
|13823391 |10 |14 |0 |  
|4034493 |9 |12 |0 |  
|13823683 |8 |17 |0 |  
|62 |9 |15 |0 |  
|4035201 |8 |12 |0 |  
|13823449 |10 |0 |0 |  
|138230919|12 |15 |0 |  
+---------+-----+-----------+------+  
 \*/  
 //机器学习部分  
 //1.类别处理* **val** stringIndexerModel: StringIndexerModel = **new** StringIndexer()  
 .setInputCol(**"gender"**)  
 .setOutputCol(**"label"**).fit(source)  
  
 *//2.特征向量化* **val** vectorAssembler: VectorAssembler = **new** VectorAssembler()  
 .setInputCols(*Array*(**"color"**, **"productType"**))  
 .setOutputCol(**"features"**)  
 **val** featureDF: DataFrame = vectorAssembler.transform(source)  
  
 *//3.对特征进行索引,大于3个不同的值的特征被视为连续特征  
 //VectorIndexer是对数据集特征向量中的类别(离散值)特征(index categorical features categorical features)进行编号。  
 //它能够自动判断那些特征是离散值型的特征，并对他们进行编号，具体做法是通过设置一个maxCategories，  
 //特征向量中某一个特征不重复取值个数小于maxCategories，则被重新编号为0～K（K<=maxCategories-1）。  
 //某一个特征不重复取值个数大于maxCategories，则该特征视为连续值，不会重新编号（不会发生任何改变）  
 //主要作用：提高决策树或随机森林等ML方法的分类效果* **val** featureVectorIndexer: VectorIndexerModel = **new** VectorIndexer()  
 .setInputCol(**"features"**)  
 .setOutputCol(**"featureIndexed"**)  
 .setMaxCategories(3)  
 .fit(featureDF)  
  
 *//4.构建分类器* **val** decisionTreeClassification: DecisionTreeClassifier = **new** DecisionTreeClassifier()  
 .setFeaturesCol(**"featureIndexed"**)  
 .setPredictionCol(**"predict"**)  
 .setMaxDepth(5)  
 *//.setImpurity("gini")  
  
 //5.类别还原* **val** indexToString: IndexToString = **new** IndexToString()  
 .setInputCol(**"label"**)  
 .setOutputCol(**"gender\_converted"**)  
  
 *//6.数据集划分* **val** *Array*(trainingData, testData) = source.randomSplit(*Array*(0.8,0.2))  
  
 *//7.构建Pipeline并训练模型* **val** pipeline: Pipeline = **new** Pipeline().setStages(*Array*(stringIndexerModel,vectorAssembler,featureVectorIndexer,decisionTreeClassification,indexToString))  
 **val** pipelineModel: PipelineModel = pipeline.fit(trainingData)  
  
 *//8.预测* **val** testReusltDF: DataFrame = pipelineModel.transform(testData)  
 **val** trainingReusltDF: DataFrame = pipelineModel.transform(trainingData)  
  
 *//9.决策过程输出* **val** classificationModel: DecisionTreeClassificationModel = pipelineModel.stages(3).asInstanceOf[DecisionTreeClassificationModel]  
 *println*(**"决策过程如下:\n"**+classificationModel.toDebugString)  
  
 *//10.评价* **this**.*evaluateACCAndAUC*(trainingReusltDF,testReusltDF)  
  
  
 *//11.返回结果并保存* **val** allResult: Dataset[Row] = testReusltDF.union(trainingReusltDF)  
 **val** tempDF: DataFrame = allResult.select(**'userId**,  
 *when*(**'predict** === 0, 1).otherwise(0).as(**"male"**), *//计算每个用户所有订单中的男性商品的订单数  
 when*(**'predict** === 1, 1).otherwise(0).as(**"female"**)) *//计算每个用户所有订单中的女性商品的订单数* .groupBy(**'userId**)  
 .agg(  
 *count*(**'userId**) cast DoubleType as **"total"**, *//总共预测多少次  
 sum*(**'male**) cast DoubleType as **"maleCount"**, *//预测为男的次数  
 sum*(**'female**) cast DoubleType as **"femaleCount"**) *//预测为女的次数* .select(**'userId**, **'total**, **'maleCount**, **'femaleCount**)  
 tempDF.show(10,**false**)  
*/\*  
+---------+-----+---------+-----------+  
|userId |total|maleCount|femaleCount|  
+---------+-----+---------+-----------+  
|4033473 |13.0 |12.0 |1.0 |  
|13823083 |17.0 |13.0 |4.0 |  
|13823681 |3.0 |3.0 |0.0 |  
|138230919|5.0 |3.0 |2.0 |  
|13822725 |7.0 |6.0 |1.0 |  
|4033575 |9.0 |8.0 |1.0 |  
|4034191 |6.0 |5.0 |1.0 |  
|13823153 |9.0 |6.0 |3.0 |  
|13822841 |3.0 |3.0 |0.0 |  
|13823431 |8.0 |4.0 |4.0 |  
+---------+-----+---------+-----------+  
 \*/  
  
 //12.上面我们就得到了用户在该网站被预测购物性别的总次数,以及被预测为男和女各多少次  
 //接下来我们要结合该用户所有的预测结果真正的给该用户打上购物性别标签  
 //预测规则A:每个订单的男性商品>=80%则认定为该订单的用户为男，或女商品比例达到80%则认定为该订单的用户为女；  
 //由于是家电产品，一个订单中通常只有一个商品。调整规则A为规则B：  
 //预测规则B:计算每个用户近半年内所有订单中的男性商品超过60%则认定该用户为男，或近半年内所有订单中的女性品超过60%则认定该用户为女  
 //那么现在我们要做的事情可以简化为:根据该用户的total/maleCount/femaleCount判断用户应该被打上哪个标签  
 //12.1 将5级标签转为map方便后面使用  
 /\*  
+---+----+  
|id |rule|  
+---+----+  
|57 |0 |  
|58 |1 |  
|59 |-1 |  
+---+----+  
 \*/  
 //Map[rule, tagId]* **val** ruleMap: Map[String, Long] = fiveDF.collect().map(row=>(row.getString(1),row.getLong(0))).toMap  
  
  
 **val** getGenderTag = *udf*((total:Double,maleCount:Double,femaleCount:Double)=>{  
 *//预测规则B:计算每个用户近半年内所有订单中的男性商品超过60%则认定该用户为男，或近半年内所有订单中的女性品超过60%则认定该用户为女* **val** maleRate: Double = maleCount / total  
 **val** femaleRate: Double = femaleCount/ total  
 **if** (maleRate>= 0.6){  
 ruleMap(**"0"**)  
 }**else if**(femaleRate>=0.6){  
 ruleMap(**"1"**)  
 }**else**{  
 ruleMap(**"-1"**)  
 }  
 })  
  
 *//12.2使用自定义UDF完成tag转换* **val** newDF: DataFrame = tempDF.select(**'userId**,getGenderTag(**'total**,**'maleCount**,**'femaleCount**).as(**"tagIds"**))  
 newDF.show(10,**false**)  
  
 *println*(**new** Date().toLocaleString)*//需要20多分钟* newDF  
 }  
 */\*\*  
 \** **@param predictTestDF** *\** **@param predictTrainDF** *\*/* **def** evaluateACCAndAUC(predictTrainDF: DataFrame,predictTestDF: DataFrame): Unit = {  
 *// 1. ACC* **val** accEvaluator = **new** MulticlassClassificationEvaluator()  
 .setPredictionCol(**"predict"**)  
 .setLabelCol(**"label"**)  
 .setMetricName(**"accuracy"**)*//精准度* **val** trainAcc: Double = accEvaluator.evaluate(predictTrainDF)  
 **val** testAcc: Double = accEvaluator.evaluate(predictTestDF)  
 *println*(**s"训练集上的 ACC 是 : $**trainAcc**"**)  
 *println*(**s"测试集上的 ACC 是 : $**testAcc**"**)  
 *//训练集上的 ACC 是 : 0.9659155462980497  
 //测试集上的 ACC 是 : 0.9666924864446166  
  
 // 2. AUC* **val** trainRdd: RDD[(Double, Double)] = predictTrainDF.select(**"label"**, **"predict"**).*rdd* .map(row => (row.getAs[Double](0), row.getAs[Double](1)))  
 **val** testRdd: RDD[(Double, Double)] = predictTestDF.select(**"label"**, **"predict"**).*rdd* .map(row => (row.getAs[Double](0), row.getAs[Double](1)))  
  
 **val** trainAUC: Double = **new** BinaryClassificationMetrics(trainRdd).areaUnderROC()  
 **val** testAUC: Double = **new** BinaryClassificationMetrics(testRdd).areaUnderROC()  
 *println*(**s"训练集上的 AUC 是 : $**trainAUC**"**)  
 *println*(**s"测试集上的 AUC 是 : $**testAUC**"**)  
 *//训练集上的 AUC 是 : 0.955163396745856  
 //测试集上的 AUC 是 : 0.955110198789974* }  
}

## 1.4模型调优

使用决策树算法训练模型时，可以调整相关参数，结合训练验证或交叉验证，获取最佳模型。

获取最佳模型代码如下：

训练验证案例文档：

http://spark.apache.org/docs/2.2.0/ml-tuning.html#train-validationsplit

/\*\*

\* 调整算法超参数，找出最优模型

\* @param dataframe 数据集

\* @return

\*/

def trainBestModel(dataframe: DataFrame): PipelineModel = {

// a. 特征向量化

val assembler = new VectorAssembler()

.setInputCols(Array("color", "product"))

.setOutputCol("features")

// b. 类别特征进行索引

val vectorIndexer = new VectorIndexer()

.setInputCol("features")

.setOutputCol("indexer\_features")

.setMaxCategories(30)

// c. 构建决策树分类器

val dtc: DecisionTreeClassifier = new DecisionTreeClassifier()

.setFeaturesCol("indexer\_features")

.setLabelCol("label")

.setPredictionCol("prediction")

.setMaxDepth(5) // 树的深度

.setMaxBins(32) // 树的叶子数目

.setImpurity("gini") // 基尼系数

// d. 构建Pipeline管道流实例对象

val pipeline: Pipeline = new Pipeline().setStages(

Array(assembler, vectorIndexer, dtc)

)

// e. 构建参数网格，设置超参数的值

val paramGrid: Array[ParamMap] = new ParamGridBuilder()

.addGrid(dtc.maxDepth, Array(5, 10))

.addGrid(dtc.impurity, Array("gini", "entropy"))

.addGrid(dtc.maxBins, Array(32, 63))

.build()

// f. 多分类评估器

val evaluator = new MulticlassClassificationEvaluator()

.setLabelCol("label")

.setPredictionCol("prediction")

// 指标名称，支持：f1、weightedPrecision、weightedRecall、

accuracy

.setMetricName("accuracy")

// g. 训练验证

val trainValidationSplit = new TrainValidationSplit()

.setEstimator(pipeline)

.setEvaluator(evaluator)

.setEstimatorParamMaps(paramGrid)

// 80% of the data will be used for training and the

remaining 20% for validation.

.setTrainRatio(0.8)

// h. 训练模型

val model: TrainValidationSplitModel =

trainValidationSplit.fit(dataframe)

// i. 获取最佳模型返回

model.bestModel.asInstanceOf[PipelineModel]

}

**修改获取模型的代码**

// 5. 构建模型（PipelineModel）

val pipelineModel: PipelineModel = trainBestModel(genderDF)

val dtcModel = pipelineModel.stages(2)

.asInstanceOf[DecisionTreeClassificationModel]

println(dtcModel.toDebugString)

## 1.5项目代码实现(未使用pipeline)

package cn.itcast.up.ml

import java.util.Date

import cn.itcast.up.base.BaseModel2

import cn.itcast.up.bean.HBaseMeta

import org.apache.spark.ml.classification.{DecisionTreeClassificationModel, DecisionTreeClassifier}

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.\_

import org.apache.spark.ml.{Pipeline, PipelineModel}

import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics

import org.apache.spark.rdd.RDD

import org.apache.spark.sql.types.DoubleType

import org.apache.spark.sql.{Column, DataFrame, functions}

/\*\*

\* Author itcast

\* Date 2019/10/29 17:49

\* Desc

\*/

object USGModel 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 = 56

/\*\*

\* 开始计算

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

\*hbaseTable=tbl\_goods##family=detail##selectFields=cOrderSn,ogColor,productType

\* @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 = {

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

import spark.implicits.\_

//fiveDF.show()

//fiveDF.printSchema()

/\*

+---+----+

| id|rule|

+---+----+

| 57| 0|

| 58| 1|

| 59| -1|

+---+----+

\*/

val ordersDF: DataFrame = spark.read

.format("cn.itcast.up.tools.HBaseSource")

.option(HBaseMeta.ZKHOSTS, "bd001")

.option(HBaseMeta.ZKPORT, "2181")

.option(HBaseMeta.HBASETABLE, "tbl\_orders")

.option(HBaseMeta.FAMILY, "detail")

.option(HBaseMeta.SELECTFIELDS, "memberId,orderSn")

.load()

//ordersDF.show(10)

/\*

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

| memberId| orderSn|

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

| 13823431| ts\_792756751164275|

| 4035167| D14090106121770839|

| 4035291| D14090112394810659|

| 4035041| fx\_787749561729045|

| 13823285| D14092120154435903|

| 4034219| D14092120155620305|

|138230939|top\_810791455519102|

| 4035083| D14092120161884409|

|138230935| D14092120162313538|

| 13823231| D14092120162378713|

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

\*/

val goodsDF: DataFrame = spark.read

.format("cn.itcast.up.tools.HBaseSource")

.option(HBaseMeta.ZKHOSTS, "bd001")

.option(HBaseMeta.ZKPORT, "2181")

.option(HBaseMeta.HBASETABLE, "tbl\_goods")

.option(HBaseMeta.FAMILY, "detail")

.option(HBaseMeta.SELECTFIELDS, "cOrderSn,ogColor,productType")//如果有更多的特征,可能需要 PCA降维

.load()

//goodsDF.show(10)

/\*

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

| cOrderSn| ogColor|productType|

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

|jd\_14091818005983607| 白色| 烤箱|

|jd\_14091317283357943| 香槟金| 冰吧|

|jd\_14092012560709235| 香槟金色| 净水机|

| rrs\_15234137|梦境极光【布朗灰】| 烤箱|

|suning\_7907506874...|梦境极光【卡其金】| 4K电视|

| rsq\_805093707860210| 黑色| 烟灶套系|

|jd\_14090910361908941| 黑色| 智能电视|

|jd\_14091823464864679| 香槟金色| 燃气灶|

|jd\_14091817311906413| 银色| 滤芯|

|suning\_8042266474...| 玫瑰金| 电饭煲|

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

\*/

//颜色ID应该来源于字典表,这里简化处理

val color: Column = functions

.when('ogColor.equalTo("银色"), 1)

.when('ogColor.equalTo("香槟金色"), 2)

.when('ogColor.equalTo("黑色"), 3)

.when('ogColor.equalTo("白色"), 4)

.when('ogColor.equalTo("梦境极光【卡其金】"), 5)

.when('ogColor.equalTo("梦境极光【布朗灰】"), 6)

.when('ogColor.equalTo("粉色"), 7)

.when('ogColor.equalTo("金属灰"), 8)

.when('ogColor.equalTo("金色"), 9)

.when('ogColor.equalTo("乐享金"), 10)

.when('ogColor.equalTo("布鲁钢"), 11)

.when('ogColor.equalTo("月光银"), 12)

.when('ogColor.equalTo("时尚光谱【浅金棕】"), 13)

.when('ogColor.equalTo("香槟色"), 14)

.when('ogColor.equalTo("香槟金"), 15)

.when('ogColor.equalTo("灰色"), 16)

.when('ogColor.equalTo("樱花粉"), 17)

.when('ogColor.equalTo("蓝色"), 18)

.when('ogColor.equalTo("金属银"), 19)

.when('ogColor.equalTo("玫瑰金"), 20)

.otherwise(0)

.alias("color")

//类型ID应该来源于字典表,这里简化处理

val productType: Column = functions

.when('productType.equalTo("4K电视"), 9)

.when('productType.equalTo("Haier/海尔冰箱"), 10)

.when('productType.equalTo("Haier/海尔冰箱"), 11)

.when('productType.equalTo("LED电视"), 12)

.when('productType.equalTo("Leader/统帅冰箱"), 13)

.when('productType.equalTo("冰吧"), 14)

.when('productType.equalTo("冷柜"), 15)

.when('productType.equalTo("净水机"), 16)

.when('productType.equalTo("前置过滤器"), 17)

.when('productType.equalTo("取暖电器"), 18)

.when('productType.equalTo("吸尘器/除螨仪"), 19)

.when('productType.equalTo("嵌入式厨电"), 20)

.when('productType.equalTo("微波炉"), 21)

.when('productType.equalTo("挂烫机"), 22)

.when('productType.equalTo("料理机"), 23)

.when('productType.equalTo("智能电视"), 24)

.when('productType.equalTo("波轮洗衣机"), 25)

.when('productType.equalTo("滤芯"), 26)

.when('productType.equalTo("烟灶套系"), 27)

.when('productType.equalTo("烤箱"), 28)

.when('productType.equalTo("燃气灶"), 29)

.when('productType.equalTo("燃气热水器"), 30)

.when('productType.equalTo("电水壶/热水瓶"), 31)

.when('productType.equalTo("电热水器"), 32)

.when('productType.equalTo("电磁炉"), 33)

.when('productType.equalTo("电风扇"), 34)

.when('productType.equalTo("电饭煲"), 35)

.when('productType.equalTo("破壁机"), 36)

.when('productType.equalTo("空气净化器"), 37)

.otherwise(0)

.alias("productType")

//使用运营的统计数据对数据进行标注

//训练的目的就是从已标注数据中找到规律,以后新来了一条数据,就可以进行预测

val label: Column = functions

.when('ogColor.equalTo("樱花粉")

.or('ogColor.equalTo("白色"))

.or('ogColor.equalTo("香槟色"))

.or('ogColor.equalTo("香槟金"))

.or('productType.equalTo("料理机"))

.or('productType.equalTo("挂烫机"))

.or('productType.equalTo("吸尘器/除螨仪")), 1) //女

.otherwise(0)//男

.alias("gender")//决策树预测label

//最终需要找到用户和用户所购买的所有商品,进行训练,找到商品和性别之间的关系

val source = goodsDF.select('cOrderSn as "orderSn", color, productType, label)

.join(ordersDF, "orderSn")

.select('memberId as "userId", 'color, 'productType, 'gender)

//source.show(10)

/\*

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

| userId|color|productType|gender|

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

| 13823535| 16| 0| 0|

| 13823535| 1| 24| 0|

| 13823535| 7| 30| 0|

| 13823391| 10| 14| 0|

| 4034493| 9| 12| 0|

| 13823683| 8| 17| 0|

| 62| 9| 15| 0|

| 4035201| 8| 12| 0|

| 13823449| 10| 0| 0|

|138230919| 12| 15| 0|

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

\*/

//机器学习部分

//https://www.cnblogs.com/itboys/p/8312894.html

//1.处理label,将元数据添加到标签列中

val labelIndexer: StringIndexerModel = new StringIndexer()

.setInputCol("gender")

.setOutputCol("label")

.fit(source)

//2.处理features,将多个列合并为向量列的特征变换器

val featureVectorAssembler: VectorAssembler = new VectorAssembler()

.setInputCols(Array("color", "productType"))

.setOutputCol("features")

val featureDF: DataFrame = featureVectorAssembler.transform(source)

//featureDF.show(10)

/\*

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

| userId|color|productType|gender| features|

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

| 13823535| 16| 0| 0| [16.0,0.0]|

| 13823535| 1| 24| 0| [1.0,24.0]|

| 13823535| 7| 30| 0| [7.0,30.0]|

| 13823391| 10| 14| 0|[10.0,14.0]|

| 4034493| 9| 12| 0| [9.0,12.0]|

| 13823683| 8| 17| 0| [8.0,17.0]|

| 62| 9| 15| 0| [9.0,15.0]|

| 4035201| 8| 12| 0| [8.0,12.0]|

| 13823449| 10| 0| 0| [10.0,0.0]|

|138230919| 12| 15| 0|[12.0,15.0]|

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

\*/

//对特征进行索引,大于3个不同的值的特征被视为连续特征

//VectorIndexer是对数据集特征向量中的类别(离散值)特征(index categorical features categorical features)进行编号。

//它能够自动判断那些特征是离散值型的特征，并对他们进行编号，具体做法是通过设置一个maxCategories，

//特征向量中某一个特征不重复取值个数小于maxCategories，则被重新编号为0～K（K<=maxCategories-1）。

//某一个特征不重复取值个数大于maxCategories，则该特征视为连续值，不会重新编号（不会发生任何改变）

//主要作用：提高决策树或随机森林等ML方法的分类效果

val featureVectorIndexer: VectorIndexerModel = new VectorIndexer()

.setInputCol("features")

.setOutputCol("featureIndexed")

.setMaxCategories(3)

.fit(featureDF)

//3.创建决策树

val decisionTreeClassifier: DecisionTreeClassifier = new DecisionTreeClassifier()

.setFeaturesCol("featureIndexed")

.setPredictionCol("predict")

.setImpurity("gini") //Gini不纯度

.setMaxDepth(5) //树的最大深度

.setMaxBins(5)//离散化连续特征的最大划分数

//4.还原label

val labelConverter: IndexToString = new IndexToString()

.setInputCol("label")

.setOutputCol("labelConverted")

.setLabels(labelIndexer.labels)

//5.划分训练集/测试集

val Array(traiData,testData) = source.randomSplit(Array(0.8,0.2))

//6.使用Pipeline串联

val pipeline: Pipeline = new Pipeline()

.setStages(Array(labelIndexer,featureVectorAssembler,featureVectorIndexer,decisionTreeClassifier,labelConverter))

val model: PipelineModel = pipeline.fit(traiData)

//7.预测

val predictTestDF: DataFrame = model.transform(testData)

val predictTrainDF: DataFrame = model.transform(traiData)

//predictTestDF.show(10,false)

//predictTrainDF.show(10,false)

/\*

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

|userId |color|productType|gender|label|features |featureIndexed|rawPrediction|probability |predict|labelConverted|

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

|13823349|1 |36 |0 |0.0 |[1.0,36.0] |[1.0,36.0] |[132.0,46.0] |[0.7415730337078652,0.25842696629213485]|0.0 |0 |

|13823535|1 |24 |0 |0.0 |[1.0,24.0] |[1.0,24.0] |[132.0,88.0] |[0.6,0.4] |0.0 |0 |

|13823611|2 |21 |0 |0.0 |[2.0,21.0] |[2.0,21.0] |[116.0,89.0] |[0.5658536585365853,0.43414634146341463]|0.0 |0 |

|4035201 |8 |12 |0 |0.0 |[8.0,12.0] |[8.0,12.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|62 |9 |15 |0 |0.0 |[9.0,15.0] |[9.0,15.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|91 |5 |10 |0 |0.0 |[5.0,10.0] |[5.0,10.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|91 |15 |0 |1 |1.0 |[15.0,0.0] |[15.0,0.0] |[135.0,131.0]|[0.5075187969924813,0.4924812030075188] |0.0 |1 |

|13823337|20 |19 |1 |1.0 |[20.0,19.0]|[20.0,19.0] |[96.0,105.0] |[0.47761194029850745,0.5223880597014925]|1.0 |1 |

|4033567 |16 |27 |0 |0.0 |[16.0,27.0]|[16.0,27.0] |[122.0,137.0]|[0.47104247104247104,0.528957528957529] |1.0 |0 |

|4034793 |7 |0 |0 |0.0 |[7.0,0.0] |[7.0,0.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

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

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

|userId |color|productType|gender|label|features |featureIndexed|rawPrediction|probability |predict|labelConverted|

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

|138230919|12 |15 |0 |0.0 |[12.0,15.0]|[12.0,15.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|13823391 |10 |14 |0 |0.0 |[10.0,14.0]|[10.0,14.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|13823449 |10 |0 |0 |0.0 |[10.0,0.0] |[10.0,0.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|13823535 |7 |30 |0 |0.0 |[7.0,30.0] |[7.0,30.0] |[353.0,51.0] |[0.8737623762376238,0.12623762376237624]|0.0 |0 |

|13823535 |16 |0 |0 |0.0 |[16.0,0.0] |[16.0,0.0] |[135.0,131.0]|[0.5075187969924813,0.4924812030075188] |0.0 |0 |

|13823563 |18 |27 |0 |0.0 |[18.0,27.0]|[18.0,27.0] |[150.0,69.0] |[0.684931506849315,0.3150684931506849] |0.0 |0 |

|13823611 |2 |32 |0 |0.0 |[2.0,32.0] |[2.0,32.0] |[132.0,46.0] |[0.7415730337078652,0.25842696629213485]|0.0 |0 |

|13823611 |6 |13 |0 |0.0 |[6.0,13.0] |[6.0,13.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|13823611 |11 |21 |0 |0.0 |[11.0,21.0]|[11.0,21.0] |[139.0,49.0] |[0.7393617021276596,0.26063829787234044]|0.0 |0 |

|13823683 |8 |17 |0 |0.0 |[8.0,17.0] |[8.0,17.0] |[154.0,56.0] |[0.7333333333333333,0.26666666666666666]|0.0 |0 |

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

\*/

//8.查看损失

evaluateAUC(predictTrainDF,predictTestDF)

//9.查看决策树

val treeModel: DecisionTreeClassificationModel = model.stages(3).asInstanceOf[DecisionTreeClassificationModel]

println("Learned classification tree model:\n" + treeModel.toDebugString)

/\*

Learned classification tree model:

DecisionTreeClassificationModel (uid=dtc\_256b176ce735) of depth 5 with 39 nodes

If (feature 0 <= 12.0)

If (feature 0 <= 4.0)

If (feature 1 <= 14.0)

If (feature 1 <= 0.0)

Predict: 0.0

Else (feature 1 > 0.0)

Predict: 0.0

Else (feature 1 > 14.0)

If (feature 1 <= 30.0)

If (feature 1 <= 22.0)

Predict: 0.0

Else (feature 1 > 22.0)

Predict: 0.0

Else (feature 1 > 30.0)

Predict: 0.0

Else (feature 0 > 4.0)

If (feature 1 <= 14.0)

Predict: 0.0

Else (feature 1 > 14.0)

If (feature 1 <= 22.0)

If (feature 0 <= 8.0)

Predict: 0.0

Else (feature 0 > 8.0)

Predict: 0.0

Else (feature 1 > 22.0)

If (feature 1 <= 30.0)

Predict: 0.0

Else (feature 1 > 30.0)

Predict: 0.0

Else (feature 0 > 12.0)

If (feature 0 <= 16.0)

If (feature 1 <= 14.0)

If (feature 1 <= 0.0)

Predict: 0.0

Else (feature 1 > 0.0)

Predict: 0.0

Else (feature 1 > 14.0)

If (feature 1 <= 22.0)

Predict: 1.0

Else (feature 1 > 22.0)

If (feature 1 <= 30.0)

Predict: 0.0

Else (feature 1 > 30.0)

Predict: 1.0

Else (feature 0 > 16.0)

If (feature 1 <= 14.0)

If (feature 1 <= 0.0)

Predict: 0.0

Else (feature 1 > 0.0)

Predict: 0.0

Else (feature 1 > 14.0)

If (feature 1 <= 22.0)

Predict: 0.0

Else (feature 1 > 22.0)

If (feature 1 <= 30.0)

Predict: 0.0

Else (feature 1 > 30.0)

Predict: 0.0

\*/

//10.预测用户的性别, 分组聚合

//预测规则A:每个订单的男性商品>=80%则认定为该订单的用户为男，或女商品比例达到80%则认定为该订单的用户为女；

//由于是家电产品，一个订单中通常只有一个商品。调整规则A为规则B：

//预测规则B:计算每个用户近半年内所有订单中的男性商品超过60%则认定该用户为男，或近半年内所有订单中的女性品超过60%则认定该用户为女

val ruleMap: Map[String, Long] = fiveDF.collect().map(row=>(row.getString(1),row.getLong(0))).toMap

println(ruleMap)

//Map(0 -> 57, 1 -> 58, -1 -> 59)

val getGenderTagUDF = udf((maleCount: Double, femaleCount: Double, total: Double)=> {

val maleRatio = maleCount / total

val femaleRatio = femaleCount / total

if (maleRatio >= 0.6) {

ruleMap("0") //男

}

if (femaleRatio >= 0.6) {

ruleMap("1")

}

ruleMap("-1")

})

val tempDF: DataFrame = predictTestDF.union(predictTrainDF)

.select('userId,

when('predict === 0, 1).otherwise(0).as("male"), //计算每个用户所有订单中的男性商品的订单数

when('predict === 1, 1).otherwise(0).as("female")) //计算每个用户所有订单中的女性商品的订单数

.groupBy('userId)

.agg(

count('userId) cast DoubleType as "total",

sum('male) cast DoubleType as "maleCount",

sum('female) cast DoubleType as "femaleCount")

tempDF.show(20,false)

/\*

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

|userId |total|maleCount|femaleCount|

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

|138230919|5.0 |3.0 |2.0 |

|13823083 |17.0 |16.0 |1.0 |

|4033473 |13.0 |13.0 |0.0 |

|13823681 |3.0 |3.0 |0.0 |

|13822725 |7.0 |7.0 |0.0 |

|4034923 |6.0 |5.0 |1.0 |

|13823431 |8.0 |7.0 |1.0 |

|13823153 |9.0 |7.0 |2.0 |

|4034191 |6.0 |4.0 |2.0 |

|4033575 |9.0 |9.0 |0.0 |

|13822841 |3.0 |3.0 |0.0 |

|4033483 |5.0 |4.0 |1.0 |

|4033348 |9.0 |8.0 |1.0 |

|4035131 |6.0 |4.0 |2.0 |

|4034761 |11.0 |9.0 |2.0 |

|13823077 |4.0 |3.0 |1.0 |

|138230937|8.0 |7.0 |1.0 |

|13822847 |3.0 |3.0 |0.0 |

|4035219 |13.0 |11.0 |2.0 |

|51 |3.0 |3.0 |0.0 |

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

\*/

val newDF: DataFrame = tempDF.select('userId, getGenderTagUDF('maleCount, 'femaleCount, 'total) as "tagIds")

newDF.show(10)

/\*

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

| userId|tagIds|

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

| 4033473| 59|

| 13823681| 59|

|138230919| 59|

| 13823083| 59|

| 13822725| 59|

| 4033575| 59|

| 4034923| 59|

| 4034191| 59|

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

\*/

println(new Date().toLocaleString)//需要20多分钟

newDF

}

/\*\*

\* @param predictTestDF

\* @param predictTrainDF

\*/

def evaluateAUC(predictTrainDF: DataFrame,predictTestDF: DataFrame): Unit = {

// 1. ACC

val accEvaluator = new MulticlassClassificationEvaluator()

.setPredictionCol("predict")

.setLabelCol("label")

.setMetricName("accuracy")//精准度

val trainAcc: Double = accEvaluator.evaluate(predictTrainDF)

val testAcc: Double = accEvaluator.evaluate(predictTestDF)

println(s"训练集上的 ACC 是 : $trainAcc")

println(s"测试集上的 ACC 是 : $testAcc")

//训练集上的 ACC 是 : 0.7512278050623347

//测试集上的 ACC 是 : 0.7660406885758998

// 2. AUC

val trainRdd: RDD[(Double, Double)] = predictTrainDF.select("label", "predict").rdd

.map(row => (row.getAs[Double](0), row.getAs[Double](1)))

val testRdd: RDD[(Double, Double)] = predictTestDF.select("label", "predict").rdd

.map(row => (row.getAs[Double](0), row.getAs[Double](1)))

val trainAUC: Double = new BinaryClassificationMetrics(trainRdd).areaUnderROC()

val testAUC: Double = new BinaryClassificationMetrics(testRdd).areaUnderROC()

println(s"训练集上的 AUC 是 : $trainAUC")

println(s"测试集上的 AUC 是 : $testAUC")

//训练集上的 AUC 是 : 0.6591635864480606

//测试集上的 AUC 是 : 0.7046995800897444

}

}

## 1.6Pipeline的项目代码实现

package cn.itcast.up.ml

import java.util.Date

import cn.itcast.up.base.BaseModel2

import cn.itcast.up.bean.HBaseMeta

import org.apache.spark.ml.classification.{DecisionTreeClassificationModel, DecisionTreeClassifier}

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.\_

import org.apache.spark.ml.{Pipeline, PipelineModel}

import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics

import org.apache.spark.rdd.RDD

import org.apache.spark.sql.types.DoubleType

import org.apache.spark.sql.{Column, DataFrame, functions}

/\*\*

\* Author itcast

\* Date 2019/10/29 17:49

\* Desc

\*/

object USGModel 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 = 56

/\*\*

\* 开始计算

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

\*hbaseTable=tbl\_goods##family=detail##selectFields=cOrderSn,ogColor,productType

\* @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 = {

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

import spark.implicits.\_

//fiveDF.show()

//fiveDF.printSchema()

/\*

+---+----+

| id|rule|

+---+----+

| 57| 0|

| 58| 1|

| 59| -1|

+---+----+

\*/

val ordersDF: DataFrame = spark.read

.format("cn.itcast.up.tools.HBaseSource")

.option(HBaseMeta.ZKHOSTS, "bd001")

.option(HBaseMeta.ZKPORT, "2181")

.option(HBaseMeta.HBASETABLE, "tbl\_orders")

.option(HBaseMeta.FAMILY, "detail")

.option(HBaseMeta.SELECTFIELDS, "memberId,orderSn")

.load()

//ordersDF.show(10)

/\*

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

| memberId| orderSn|

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

| 13823431| ts\_792756751164275|

| 4035167| D14090106121770839|

| 4035291| D14090112394810659|

| 4035041| fx\_787749561729045|

| 13823285| D14092120154435903|

| 4034219| D14092120155620305|

|138230939|top\_810791455519102|

| 4035083| D14092120161884409|

|138230935| D14092120162313538|

| 13823231| D14092120162378713|

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

\*/

val goodsDF: DataFrame = spark.read

.format("cn.itcast.up.tools.HBaseSource")

.option(HBaseMeta.ZKHOSTS, "bd001")

.option(HBaseMeta.ZKPORT, "2181")

.option(HBaseMeta.HBASETABLE, "tbl\_goods")

.option(HBaseMeta.FAMILY, "detail")

.option(HBaseMeta.SELECTFIELDS, "cOrderSn,ogColor,productType")//如果有更多的特征,可能需要 PCA降维

.load()

//goodsDF.show(10)

/\*

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

| cOrderSn| ogColor|productType|

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

|jd\_14091818005983607| 白色| 烤箱|

|jd\_14091317283357943| 香槟金| 冰吧|

|jd\_14092012560709235| 香槟金色| 净水机|

| rrs\_15234137|梦境极光【布朗灰】| 烤箱|

|suning\_7907506874...|梦境极光【卡其金】| 4K电视|

| rsq\_805093707860210| 黑色| 烟灶套系|

|jd\_14090910361908941| 黑色| 智能电视|

|jd\_14091823464864679| 香槟金色| 燃气灶|

|jd\_14091817311906413| 银色| 滤芯|

|suning\_8042266474...| 玫瑰金| 电饭煲|

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

\*/

//颜色ID应该来源于字典表,这里简化处理

val color: Column = functions

.when('ogColor.equalTo("银色"), 1)

.when('ogColor.equalTo("香槟金色"), 2)

.when('ogColor.equalTo("黑色"), 3)

.when('ogColor.equalTo("白色"), 4)

.when('ogColor.equalTo("梦境极光【卡其金】"), 5)

.when('ogColor.equalTo("梦境极光【布朗灰】"), 6)

.when('ogColor.equalTo("粉色"), 7)

.when('ogColor.equalTo("金属灰"), 8)

.when('ogColor.equalTo("金色"), 9)

.when('ogColor.equalTo("乐享金"), 10)

.when('ogColor.equalTo("布鲁钢"), 11)

.when('ogColor.equalTo("月光银"), 12)

.when('ogColor.equalTo("时尚光谱【浅金棕】"), 13)

.when('ogColor.equalTo("香槟色"), 14)

.when('ogColor.equalTo("香槟金"), 15)

.when('ogColor.equalTo("灰色"), 16)

.when('ogColor.equalTo("樱花粉"), 17)

.when('ogColor.equalTo("蓝色"), 18)

.when('ogColor.equalTo("金属银"), 19)

.when('ogColor.equalTo("玫瑰金"), 20)

.otherwise(0)

.alias("color")

//类型ID应该来源于字典表,这里简化处理

val productType: Column = functions

.when('productType.equalTo("4K电视"), 9)

.when('productType.equalTo("Haier/海尔冰箱"), 10)

.when('productType.equalTo("Haier/海尔冰箱"), 11)

.when('productType.equalTo("LED电视"), 12)

.when('productType.equalTo("Leader/统帅冰箱"), 13)

.when('productType.equalTo("冰吧"), 14)

.when('productType.equalTo("冷柜"), 15)

.when('productType.equalTo("净水机"), 16)

.when('productType.equalTo("前置过滤器"), 17)

.when('productType.equalTo("取暖电器"), 18)

.when('productType.equalTo("吸尘器/除螨仪"), 19)

.when('productType.equalTo("嵌入式厨电"), 20)

.when('productType.equalTo("微波炉"), 21)

.when('productType.equalTo("挂烫机"), 22)

.when('productType.equalTo("料理机"), 23)

.when('productType.equalTo("智能电视"), 24)

.when('productType.equalTo("波轮洗衣机"), 25)

.when('productType.equalTo("滤芯"), 26)

.when('productType.equalTo("烟灶套系"), 27)

.when('productType.equalTo("烤箱"), 28)

.when('productType.equalTo("燃气灶"), 29)

.when('productType.equalTo("燃气热水器"), 30)

.when('productType.equalTo("电水壶/热水瓶"), 31)

.when('productType.equalTo("电热水器"), 32)

.when('productType.equalTo("电磁炉"), 33)

.when('productType.equalTo("电风扇"), 34)

.when('productType.equalTo("电饭煲"), 35)

.when('productType.equalTo("破壁机"), 36)

.when('productType.equalTo("空气净化器"), 37)

.otherwise(0)

.alias("productType")

//使用运营的统计数据对数据进行标注

//训练的目的就是从已标注数据中找到规律,以后新来了一条数据,就可以进行预测

val label: Column = functions

.when('ogColor.equalTo("樱花粉")

.or('ogColor.equalTo("白色"))

.or('ogColor.equalTo("香槟色"))

.or('ogColor.equalTo("香槟金"))

.or('productType.equalTo("料理机"))

.or('productType.equalTo("挂烫机"))

.or('productType.equalTo("吸尘器/除螨仪")), 1) //女

.otherwise(0)//男

.alias("gender")//决策树预测label

//最终需要找到用户和用户所购买的所有商品,进行训练,找到商品和性别之间的关系

val source = goodsDF.select('cOrderSn as "orderSn", color, productType, label)

.join(ordersDF, "orderSn")

.select('memberId as "userId", 'color, 'productType, 'gender)

//source.show(10)

/\*

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

| userId|color|productType|gender|

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

| 13823535| 16| 0| 0|

| 13823535| 1| 24| 0|

| 13823535| 7| 30| 0|

| 13823391| 10| 14| 0|

| 4034493| 9| 12| 0|

| 13823683| 8| 17| 0|

| 62| 9| 15| 0|

| 4035201| 8| 12| 0|

| 13823449| 10| 0| 0|

|138230919| 12| 15| 0|

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

\*/

//机器学习部分

//https://www.cnblogs.com/itboys/p/8312894.html

//1.处理label,将元数据添加到标签列中

val labelIndexer: StringIndexerModel = new StringIndexer()

.setInputCol("gender")

.setOutputCol("label")

.fit(source)

//2.处理features,将多个列合并为向量列的特征变换器

val featureVectorAssembler: VectorAssembler = new VectorAssembler()

.setInputCols(Array("color", "productType"))

.setOutputCol("features")

val featureDF: DataFrame = featureVectorAssembler.transform(source)

//featureDF.show(10)

/\*

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

| userId|color|productType|gender| features|

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

| 13823535| 16| 0| 0| [16.0,0.0]|

| 13823535| 1| 24| 0| [1.0,24.0]|

| 13823535| 7| 30| 0| [7.0,30.0]|

| 13823391| 10| 14| 0|[10.0,14.0]|

| 4034493| 9| 12| 0| [9.0,12.0]|

| 13823683| 8| 17| 0| [8.0,17.0]|

| 62| 9| 15| 0| [9.0,15.0]|

| 4035201| 8| 12| 0| [8.0,12.0]|

| 13823449| 10| 0| 0| [10.0,0.0]|

|138230919| 12| 15| 0|[12.0,15.0]|

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

\*/

//对特征进行索引,大于3个不同的值的特征被视为连续特征

//VectorIndexer是对数据集特征向量中的类别(离散值)特征(index categorical features categorical features)进行编号。

//它能够自动判断那些特征是离散值型的特征，并对他们进行编号，具体做法是通过设置一个maxCategories，

//特征向量中某一个特征不重复取值个数小于maxCategories，则被重新编号为0～K（K<=maxCategories-1）。

//某一个特征不重复取值个数大于maxCategories，则该特征视为连续值，不会重新编号（不会发生任何改变）

//主要作用：提高决策树或随机森林等ML方法的分类效果

val featureVectorIndexer: VectorIndexerModel = new VectorIndexer()

.setInputCol("features")

.setOutputCol("featureIndexed")

.setMaxCategories(3)

.fit(featureDF)

//3.创建决策树

val decisionTreeClassifier: DecisionTreeClassifier = new DecisionTreeClassifier()

.setFeaturesCol("featureIndexed")

.setPredictionCol("predict")

.setImpurity("gini") //Gini不纯度

.setMaxDepth(5) //树的最大深度

.setMaxBins(5)//离散化连续特征的最大划分数

//4.还原label

val labelConverter: IndexToString = new IndexToString()

.setInputCol("label")

.setOutputCol("labelConverted")

.setLabels(labelIndexer.labels)

//5.划分训练集/测试集

val Array(traiData,testData) = source.randomSplit(Array(0.8,0.2))

//6.使用Pipeline串联

val pipeline: Pipeline = new Pipeline()

.setStages(Array(labelIndexer,featureVectorAssembler,featureVectorIndexer,decisionTreeClassifier,labelConverter))

val model: PipelineModel = pipeline.fit(traiData)

//7.预测

val predictTestDF: DataFrame = model.transform(testData)

val predictTrainDF: DataFrame = model.transform(traiData)

//predictTestDF.show(10,false)

//predictTrainDF.show(10,false)

/\*

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

|userId |color|productType|gender|label|features |featureIndexed|rawPrediction|probability |predict|labelConverted|

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

|13823349|1 |36 |0 |0.0 |[1.0,36.0] |[1.0,36.0] |[132.0,46.0] |[0.7415730337078652,0.25842696629213485]|0.0 |0 |

|13823535|1 |24 |0 |0.0 |[1.0,24.0] |[1.0,24.0] |[132.0,88.0] |[0.6,0.4] |0.0 |0 |

|13823611|2 |21 |0 |0.0 |[2.0,21.0] |[2.0,21.0] |[116.0,89.0] |[0.5658536585365853,0.43414634146341463]|0.0 |0 |

|4035201 |8 |12 |0 |0.0 |[8.0,12.0] |[8.0,12.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|62 |9 |15 |0 |0.0 |[9.0,15.0] |[9.0,15.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|91 |5 |10 |0 |0.0 |[5.0,10.0] |[5.0,10.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|91 |15 |0 |1 |1.0 |[15.0,0.0] |[15.0,0.0] |[135.0,131.0]|[0.5075187969924813,0.4924812030075188] |0.0 |1 |

|13823337|20 |19 |1 |1.0 |[20.0,19.0]|[20.0,19.0] |[96.0,105.0] |[0.47761194029850745,0.5223880597014925]|1.0 |1 |

|4033567 |16 |27 |0 |0.0 |[16.0,27.0]|[16.0,27.0] |[122.0,137.0]|[0.47104247104247104,0.528957528957529] |1.0 |0 |

|4034793 |7 |0 |0 |0.0 |[7.0,0.0] |[7.0,0.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

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

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

|userId |color|productType|gender|label|features |featureIndexed|rawPrediction|probability |predict|labelConverted|

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

|138230919|12 |15 |0 |0.0 |[12.0,15.0]|[12.0,15.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|13823391 |10 |14 |0 |0.0 |[10.0,14.0]|[10.0,14.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|13823449 |10 |0 |0 |0.0 |[10.0,0.0] |[10.0,0.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|13823535 |7 |30 |0 |0.0 |[7.0,30.0] |[7.0,30.0] |[353.0,51.0] |[0.8737623762376238,0.12623762376237624]|0.0 |0 |

|13823535 |16 |0 |0 |0.0 |[16.0,0.0] |[16.0,0.0] |[135.0,131.0]|[0.5075187969924813,0.4924812030075188] |0.0 |0 |

|13823563 |18 |27 |0 |0.0 |[18.0,27.0]|[18.0,27.0] |[150.0,69.0] |[0.684931506849315,0.3150684931506849] |0.0 |0 |

|13823611 |2 |32 |0 |0.0 |[2.0,32.0] |[2.0,32.0] |[132.0,46.0] |[0.7415730337078652,0.25842696629213485]|0.0 |0 |

|13823611 |6 |13 |0 |0.0 |[6.0,13.0] |[6.0,13.0] |[834.0,0.0] |[1.0,0.0] |0.0 |0 |

|13823611 |11 |21 |0 |0.0 |[11.0,21.0]|[11.0,21.0] |[139.0,49.0] |[0.7393617021276596,0.26063829787234044]|0.0 |0 |

|13823683 |8 |17 |0 |0.0 |[8.0,17.0] |[8.0,17.0] |[154.0,56.0] |[0.7333333333333333,0.26666666666666666]|0.0 |0 |

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

\*/

//8.查看损失

evaluateAUC(predictTrainDF,predictTestDF)

//9.查看决策树

val treeModel: DecisionTreeClassificationModel = model.stages(3).asInstanceOf[DecisionTreeClassificationModel]

println("Learned classification tree model:\n" + treeModel.toDebugString)

/\*

Learned classification tree model:

DecisionTreeClassificationModel (uid=dtc\_256b176ce735) of depth 5 with 39 nodes

If (feature 0 <= 12.0)

If (feature 0 <= 4.0)

If (feature 1 <= 14.0)

If (feature 1 <= 0.0)

Predict: 0.0

Else (feature 1 > 0.0)

Predict: 0.0

Else (feature 1 > 14.0)

If (feature 1 <= 30.0)

If (feature 1 <= 22.0)

Predict: 0.0

Else (feature 1 > 22.0)

Predict: 0.0

Else (feature 1 > 30.0)

Predict: 0.0

Else (feature 0 > 4.0)

If (feature 1 <= 14.0)

Predict: 0.0

Else (feature 1 > 14.0)

If (feature 1 <= 22.0)

If (feature 0 <= 8.0)

Predict: 0.0

Else (feature 0 > 8.0)

Predict: 0.0

Else (feature 1 > 22.0)

If (feature 1 <= 30.0)

Predict: 0.0

Else (feature 1 > 30.0)

Predict: 0.0

Else (feature 0 > 12.0)

If (feature 0 <= 16.0)

If (feature 1 <= 14.0)

If (feature 1 <= 0.0)

Predict: 0.0

Else (feature 1 > 0.0)

Predict: 0.0

Else (feature 1 > 14.0)

If (feature 1 <= 22.0)

Predict: 1.0

Else (feature 1 > 22.0)

If (feature 1 <= 30.0)

Predict: 0.0

Else (feature 1 > 30.0)

Predict: 1.0

Else (feature 0 > 16.0)

If (feature 1 <= 14.0)

If (feature 1 <= 0.0)

Predict: 0.0

Else (feature 1 > 0.0)

Predict: 0.0

Else (feature 1 > 14.0)

If (feature 1 <= 22.0)

Predict: 0.0

Else (feature 1 > 22.0)

If (feature 1 <= 30.0)

Predict: 0.0

Else (feature 1 > 30.0)

Predict: 0.0

\*/

//10.预测用户的性别, 分组聚合

//预测规则A:每个订单的男性商品>=80%则认定为该订单的用户为男，或女商品比例达到80%则认定为该订单的用户为女；

//由于是家电产品，一个订单中通常只有一个商品。调整规则A为规则B：

//预测规则B:计算每个用户近半年内所有订单中的男性商品超过60%则认定该用户为男，或近半年内所有订单中的女性品超过60%则认定该用户为女

val ruleMap: Map[String, Long] = fiveDF.collect().map(row=>(row.getString(1),row.getLong(0))).toMap

println(ruleMap)

//Map(0 -> 57, 1 -> 58, -1 -> 59)

val getGenderTagUDF = udf((maleCount: Double, femaleCount: Double, total: Double)=> {

val maleRatio = maleCount / total

val femaleRatio = femaleCount / total

if (maleRatio >= 0.6) {

ruleMap("0") //男

}

if (femaleRatio >= 0.6) {

ruleMap("1")

}

ruleMap("-1")

})

val tempDF: DataFrame = predictTestDF.union(predictTrainDF)

.select('userId,

when('predict === 0, 1).otherwise(0).as("male"), //计算每个用户所有订单中的男性商品的订单数

when('predict === 1, 1).otherwise(0).as("female")) //计算每个用户所有订单中的女性商品的订单数

.groupBy('userId)

.agg(

count('userId) cast DoubleType as "total",

sum('male) cast DoubleType as "maleCount",

sum('female) cast DoubleType as "femaleCount")

tempDF.show(20,false)

/\*

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

|userId |total|maleCount|femaleCount|

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

|138230919|5.0 |3.0 |2.0 |

|13823083 |17.0 |16.0 |1.0 |

|4033473 |13.0 |13.0 |0.0 |

|13823681 |3.0 |3.0 |0.0 |

|13822725 |7.0 |7.0 |0.0 |

|4034923 |6.0 |5.0 |1.0 |

|13823431 |8.0 |7.0 |1.0 |

|13823153 |9.0 |7.0 |2.0 |

|4034191 |6.0 |4.0 |2.0 |

|4033575 |9.0 |9.0 |0.0 |

|13822841 |3.0 |3.0 |0.0 |

|4033483 |5.0 |4.0 |1.0 |

|4033348 |9.0 |8.0 |1.0 |

|4035131 |6.0 |4.0 |2.0 |

|4034761 |11.0 |9.0 |2.0 |

|13823077 |4.0 |3.0 |1.0 |

|138230937|8.0 |7.0 |1.0 |

|13822847 |3.0 |3.0 |0.0 |

|4035219 |13.0 |11.0 |2.0 |

|51 |3.0 |3.0 |0.0 |

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

\*/

val newDF: DataFrame = tempDF.select('userId, getGenderTagUDF('maleCount, 'femaleCount, 'total) as "tagIds")

newDF.show(10)

/\*

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

| userId|tagIds|

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

| 4033473| 59|

| 13823681| 59|

|138230919| 59|

| 13823083| 59|

| 13822725| 59|

| 4033575| 59|

| 4034923| 59|

| 4034191| 59|

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

\*/

println(new Date().toLocaleString)//需要20多分钟

newDF

}

/\*\*

\* @param predictTestDF

\* @param predictTrainDF

\*/

def evaluateAUC(predictTrainDF: DataFrame,predictTestDF: DataFrame): Unit = {

// 1. ACC

val accEvaluator = new MulticlassClassificationEvaluator()

.setPredictionCol("predict")

.setLabelCol("label")

.setMetricName("accuracy")//精准度

val trainAcc: Double = accEvaluator.evaluate(predictTrainDF)

val testAcc: Double = accEvaluator.evaluate(predictTestDF)

println(s"训练集上的 ACC 是 : $trainAcc")

println(s"测试集上的 ACC 是 : $testAcc")

//训练集上的 ACC 是 : 0.7512278050623347

//测试集上的 ACC 是 : 0.7660406885758998

// 2. AUC

val trainRdd: RDD[(Double, Double)] = predictTrainDF.select("label", "predict").rdd

.map(row => (row.getAs[Double](0), row.getAs[Double](1)))

val testRdd: RDD[(Double, Double)] = predictTestDF.select("label", "predict").rdd

.map(row => (row.getAs[Double](0), row.getAs[Double](1)))

val trainAUC: Double = new BinaryClassificationMetrics(trainRdd).areaUnderROC()

val testAUC: Double = new BinaryClassificationMetrics(testRdd).areaUnderROC()

println(s"训练集上的 AUC 是 : $trainAUC")

println(s"测试集上的 AUC 是 : $testAUC")

//训练集上的 AUC 是 : 0.6591635864480606

//测试集上的 AUC 是 : 0.7046995800897444

}

}

# 4.模型进阶

### 4.1.SparkMllib的集成学习的原理及实战

#### 4.1.1 Cart树

Cart模型是一种决策树模型，它即可以用于分类，也可以用于回归，其学习算法分为下面两步：

1. 决策树生成：用训练数据生成决策树，生成树尽可能大
2. 决策树剪枝：基于损失函数最小化的剪枝，用验证数据对生成的数据进行剪枝。

分类和回归树模型采用不同的最优化策略。Cart回归树使用平方误差最小化策略，Cart分类生成树采用的基尼指数最小化策略。

Scikit-learn中有两类决策树，他们均采用优化的Cart决策树算法。一个是DecisionTreeClassifier一个是DecisionTreeRegressor回归。

##### 4.1.1.1Cart树介绍

分类回归树(CART,Classification And Regression Tree)[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/u014568921/article/details/_blank)是一种决策树分类方法。

它采用一种二分递归分割的技术，分割方法采用基于最小距离的基尼指数估计函数，将当前的样本集分为两个子样本集，使得生成的的每个非叶子节点都有两个分支。因此，CART算法生成的决策树是结构简洁的二叉树。

**其核心思想**与ID3和C4.5相同，主要的不同处在于CART在每一个节点上都采用二分法，即每个节点都只能有两个子节点，最后构成的是二叉树。

划分方法：剪枝

**Cart算法步骤：**

1. 决策树的生成：基于训练数据集生成决策树，生成的决策树要尽量大。
2. 决策树的剪枝：用验证数据集对以生成的树进行剪枝并选择最优子树。这时用损失函数最小作为剪枝标准。

##### 4.1.1.2 Cart树生成

决策树的生成就是递归地构建二叉决策树的过程，对回归树用平方误差最小准则，对分类树用基尼指数最小化原则，进行特征选择，生成二叉树。

##### 4.1.1.3回归树原理

**回归树的原理：**

如果目标是连续变量，则是Regression Tree回归树。CART树是二叉树，不像多叉树那样形成过多的数据碎片。

对于连续变量X（x1…xn）如何处理？

首先将值排序，分别取其两相邻值的平均值点作为分隔点，将树一分成左枝和右枝，不断扫描，进而判断最佳分割点。特征值大于分裂值就走左子树，或者就走右子树。

**下面从数学层面做推导：**

假设X与Y分别为输入和输出变量，并且Y是连续变量，给定训练数据集：



考虑如何生成回归树。

一个回归树对用着一个特征空间的一个划分及在划分单元上的是输出值。假设已输入特征空间划分为M个单元R1，R2.....Rn,并且在每个单元上有一个固定的输出类别Cm，于是我们把回归树表示为：



损失函数：



表示回归树对于训练数据的预测误差，用平方误差最小的准则求解每个单元上的最有输出值。我们考虑特征空间的第m个单元上的的最优值，它是上的所有输入实例对应输出的的均值：（连续值通常的最优值都是取均值）



问题是怎么样对输入空间进行划分，我们采用启发式的方法；选择第j个变量和它去的值s，作为切分变量(splitting variable)和切分点(splitting point)，并定义两个区域：



然后我们需要寻求切分变量j和最有切分点s，求解如下方程：



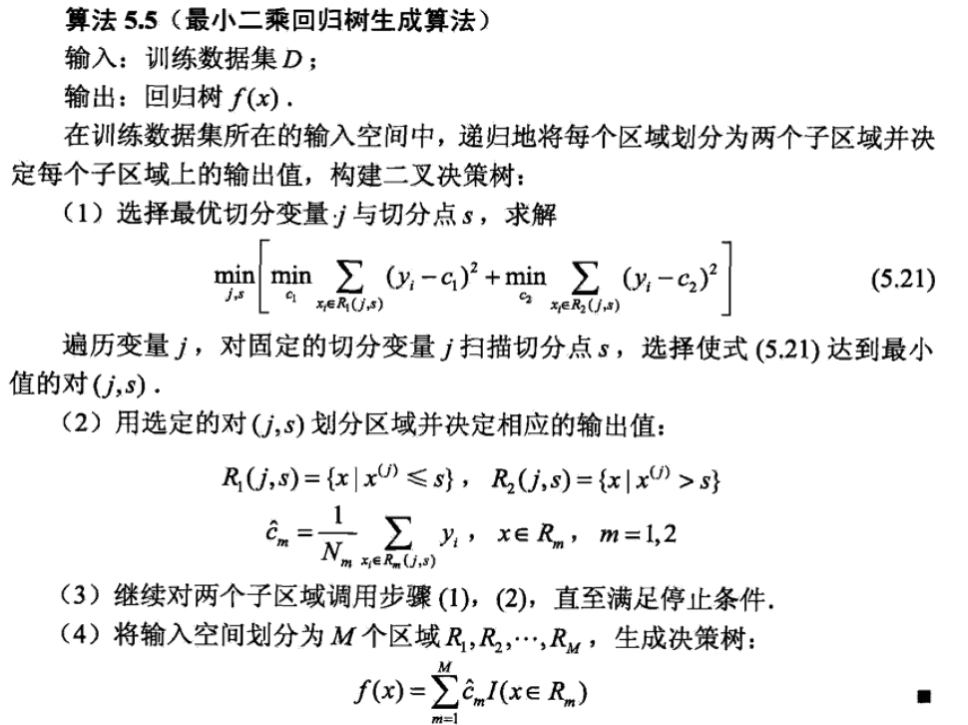
对固定输入变量j可以寻找最优切分点s：



遍历所有的输入变量，找到最优的切分变量j，构成一个对(j,s),依次将输入空间划分为两个区域。

接着对每个区域重复上述过程，直到满足停止条件为主。这样就生成了一颗回归树，也称为最小二乘回归树(least square regression tree)。

##### 4.1.1.4回归树算法总结



##### 4.1.1.5分类树原理

如果目标变量是离散变量，则是classfication Tree分类树。

分类树是使用树结构算法将数据分成离散类的方法。

**（1） 分类树两个关键点：**

➀将训练样本进行递归地划分自变量空间进行建树

➁用验证数据进行剪枝。

**（2）对于离散变量X（x1…xn）处理：**

分别取X变量各值的不同组合，将其分到树的左枝或右枝，并对不同组合而产生的树，进行评判，找出最佳组合。如果只有两个取值，直接根据这两个值就可以划分树。取值多于两个的情况就复杂一些了，如变量年纪，其值有“少年”、“中年”、“老年”，则分别生产{少年，中年}和{老年}，{上年、老年}和{中年}，{中年，老年}和{少年}，这三种组合，最后评判对目标区分最佳的组合。因为CART二分的特性，当训练数据具有两个以上的类别，CART需考虑将目标类别合并成两个超类别，这个过程称为双化。这里可以说一个公式,n个属性，可以分出(2^n-2)/2种情况。

1. **例子**



上例是属性有8个，每个属性又有多个离散的值可取。在决策树的每一个节点上我们可以按任一个属性的任一个值进行划分。比如最开始我们按：

1）表面覆盖为毛发和非毛发

2）表面覆盖为鳞片和非鳞片

3）体温为恒温和非恒温

要产生树的左右两个孩子，按哪种划分最好呢？一般我们采用GINI指数，作为划分标准。总体内包含的类别越杂乱，GINI指数就越大（跟熵的概念很相似）。

分类树用基尼指数选择最优特征，同时决定该特征的最优二值切分点。

##### 4.1.1.6变量和最佳切分点选择原则

树的生长，总的原则是，让枝比树更纯，而度量原则是根据不纯对指标来衡量，对于分类树，则用GINI指标、Twoing指标、Order Twoing等；如果是回归树则用，最小平方残差、最小绝对残差等指标衡量。

（1）利用基尼指数求解划分

**GINI指数**

分类问题中，假设有k个类，样本点属于第i类的概率为pi，则基尼指数定义为：



体温为恒温时包含哺乳类5个、鸟类2个，体温为非恒温时包含爬行类3个、鱼类3个、两栖类2个。



体温为**恒温**时包含哺乳类5个、鸟类2个，则：



体温为**非恒温**时包含爬行类3个、鱼类3个、两栖类2个,则：



（2）集合的基尼指数

如果样本集合D根据特征A是否取某一可能值a被分割成D1和D2两部分，则在特征A的条件下，集合D的基尼增益定义为：



如果按照“体温为恒温和非恒温”进行划分的话，我们得到GINI的增益：



集合的基尼指数表示集合D的不确定性，基尼指数值越大，样本属于某类的不确定性也就越大，这点与熵相似。我们总希望获得更多信息，减少不确定性。因此，最好的选取特征划分就是使得集合的基尼指数GINI最小的划分。

GINI指数补充：介于0~1之间的数，0-完全相等，1-完全不等。总体内包含的类别越杂乱，GINI指数就越大（和熵的概念类似）。

（3）最小平方残差——针对连续目标（用于回归树中）

IMG_256

其思想是，让组内方差最小，对应组间方差最大，这样的两组数据，也即树分裂的左枝和右枝差异化最大。

通过以上不纯度指标，分别计算每个变量的各种切分/组合情况，找出该变量的最佳值组合/切分点；再比较各个变量的最佳值组合/切分点，最终找出最佳变量和该变量的最佳值组合/切分点。

（3）剪枝

当CART树划分得太细时，会对噪声数据产生过拟合作用。因此我们要通过剪枝来解决。剪枝又分为前剪枝和后剪枝。

前剪枝是指在构造树的过程中就知道哪些节点可以剪掉，于是干脆不对这些节点进行分裂。后剪枝是指构造出完整的决策树之后再来考查哪些子树可以剪掉。

CART剪枝算法从“完全生长”的决策树的底端剪去一些子树，使决策树变小(模型变简单)，从而能够对未知数据有更准确的预测。

CART剪枝算法由两步组成：首先从生成算法产生的决策树T0底端开始不断剪枝，直到T0的根节点，形成一个子树序列；然后通过交叉验证法在独立的验证数据集上对子树序列进行测试，从中选择最优子树。

（4）误差率增益值

CART树中的每一个非叶子节点的表面误差率增益值α(误差增加的速率，越小越好)



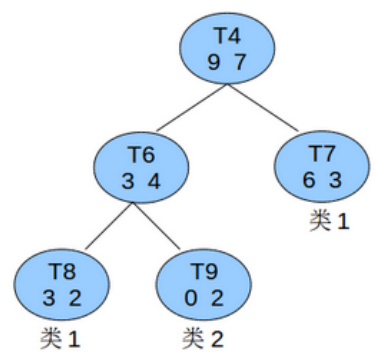
是是子树中包含的叶子节点个数。

是节点t的误差代价，如果该节点被剪枝：，r(t)是节点t的误差率；p(t)是节点t上的数据占所有数据的比例；

 是子树Tt的误差代价，如果该节点不被剪枝。它等于子树Tt上所有叶子节点的误差代价之和。

**例子讲解：**

有个非叶子节点t4如图所示：



已知所有的数据总共有60条，则节点t4的节点误差代价为：



注意:叶子节点的类定义为覆盖的样本占多数的类，即分正确的为多数，分错的为少数。

子树误差代价为：



以t4为根节点的子树上叶子节点有3个，最终：



找到α值最小的非叶子节点，令其左右孩子为空，即该节点成为叶子节点，即剪枝。

##### 4.1.2分类树算法步骤

CART分类树生成算法：

输入：训练数据集D，停止计算的条件；

输出：CART决策树

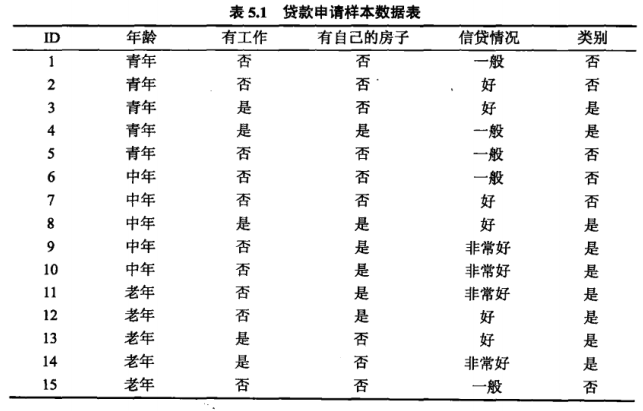
1. 根据训练数据集，从根节点开始，递归地对每个节点进行以下操作，构建二叉决策树：
2. 设结点的训练数据集为D，计算现有特征对该数据集的基尼指数，此时对每一个特征A，对其可能取得每个值a，根据样本点对A=a的测试为“是”或“否”将D分割成D1和D2两部分，利用集合的基尼指数公式计算A=a的基尼指数。



1. 在所有可能的特征A以及他们的所有可能的切分点a中，选择基尼指数最小的特征及其对应的切分点作为最优特征与最优切分点。依照最优特征和最优切分点，从现结点生成两个子节点，将训练数据集依特征分配到两个子节点中去。
2. 对两个子节点递归地调用（1），（2），直到满足停止条件。
3. 生成CART决策树。

##### 4.1.3分类决策树的例子

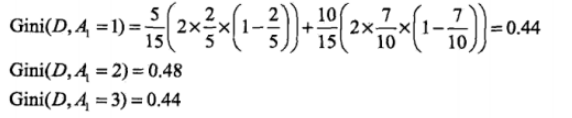
数据集：



应用上述数据集，应用CART算法生成决策树。

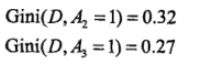
分析：首先计算各特征的基尼指数，选择最优特征以及其最优切分点。我们以A1,A2,A3,A4表示**年龄、有工作、有自己的房子和信贷**4个特征，并以1，2，3表示年龄的值为青年、中年、老年，以1、2表示有工作和有自己的房子的值为是和否，以1、2、3表示信贷情况的值为非常好、好和一般。

求特征A1的基尼指数：



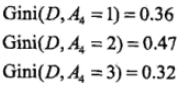
优于Gini(D,A1=1)和Gini(D,A1=3)相等，且最小，所以A1=1和A1=3都可以选作最优切分点。

求特征A2和A3的基尼指数：



由于A2和A3只有一个切分点，所以他们就是最优切分点。

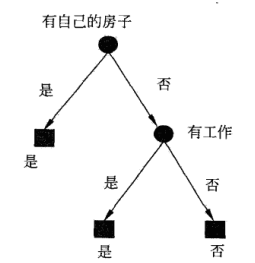
求解特征4的基尼指数：



Gini(D,A4=3)最小，所以A4=3为A4的最优切分点。

在A1，A2，A3，A4几个特征中，Gini(D,A3=1)=0.27最小，所以**选择A3为最优特征**，A3=1为其最优切分点。于是根节点生成两个子节点，一个是叶节点，对另一个结点继续使用以上方法在A1，A2，A3中选择最优特征及其最优切分点，结果是A2=1，依次计算得知，所有结点都是叶节点。

该例子按照CART算法和ID3算法生成的决策树完全一致的。下图我们只拿A3和A2作为最优划分来切分数据集。



##### 4.1.4Cart树总结

创建分类树递归过程中，CART每次都选择当前数据集中具有最小Gini信息增益的特征作为结点划分决策树。ID3算法和C4.5算法虽然在对训练样本集的学习中可以尽可能多地挖掘信息，但其生成的决策树分支、规模较大，CART算法的二分法可以简化决策树的规模，提高生成决策树的效率。对于连续特征，CART也是采取和C4.5同样的方法处理。为了避免过拟合(Overfitting)，CART决策树需要剪枝（后剪枝）。预测过程当然也就十分简单，根据产生的决策树模型，延伸匹配特征值到最后的叶子节点即得到预测的类别。

### 4.2集成学习分类--串行和并行学习算法

集成学习算法一般分为：bagging、boosting和Stacking。

随机森林是集成模型中的一种，常言道：“一个篱笆三个桩，一个好汉三个帮”。集成分类模型便是综合考量多个分类器的预测结果，从而做出决策，集成学习分两种：

1. 利用相同的训练数据同时搭建多个独立的分类模型，然后通过投票的方式，以少数服从多数的原则做出最终的分类决策。今天学习的随机森林就是这种方式，即在相同训练数据上同时搭建多颗决策树。

在决策树中学到过一颗标准的决策树是根据每维特征对预测结果的影响程度进行排序，进而决定不同特征从上到下构建分裂节点的顺序；如果这里还按照这种方式随机森林会因为这一策略影响而构建的所有树都一致，从而丧失了多样性。因此随机森林在构建的过程中，每一颗决策树都会放弃这一个固定的算法，转而随机选取特征。

（2）按一定的次序搭建多个分类模型。这些模型之间彼此存在依赖关系。一般后一个模型的加入都需要对现有的集成模型有一定贡献，进而不断提高更新过后的集成模型性能，并借助多个弱分类器搭建出强分类器。代表行的有Bossting（AdaBoost）算法。该算法与第一种的随机森林主要区别在于每一颗决策树在生成的过程中都会尽可能降低模型在训练集上的拟合或训练误差。

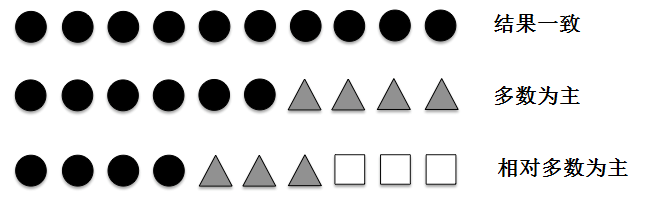
##### 4.2.1集成学习关键要素

俗话说：“三个臭皮匠赛过诸葛亮”。

当使用某一种分类器不能使我们达到很好的效果的时候，我们不妨设想将这些分类效果不好的分类器组合一下，再去看看效果是否有提升，这就是集成学习的思想。

集成算法（Ensemble Learning）思想：通过构建并结合多个学习器来完成学习任务，有时候我们也叫作“多分类器系统”。如下图C1，C2都是某一种个体的分类器，我们采用某种策略将他们组合起来。

我们有10个分类器，多数表决如下图，其中三角形、正方形或圆分别代表一个类别。



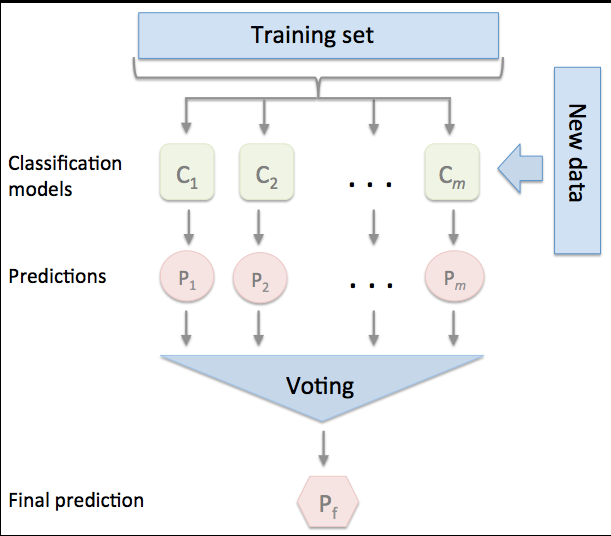
个体的分类器由一个现有的学习算法从训练数据产生，例如：C4.5决策树算法，BP神经网络算法等。我们一般把个体分类器全部为同种的分类器称为“同质的”，如全部为决策树模型。同质的集成学习中的个体学习器称为“基学习器（base learner）”，相应的算法称为基学习算法。反之，集成中包含不同种的学习器，我们称之为“异质”的，异质集成中的个体学习器包含不同的学习算法组成的，这种情况下的学习器称为“组合学习器”。相信这些名词大家能够了解。



图1： 集成学习的系统示意图

集成学习通过将多个学习器组合，常获得比单一学习器显著优越的泛化性能。这对弱学习器特别明显，这里的弱学习器我们一般会使用决策树，BP神经网络和逻辑斯特回归，有时候SVM也可以作为个体学习器。

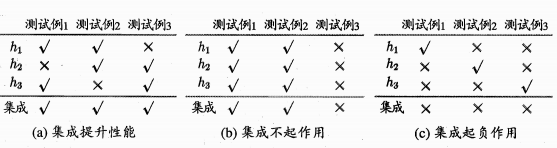
下图是使用多数投票法通用集成方法的概念：



##### 4.2.2集成学习器性能评估

一般经验中如果把好坏不等的东西掺到一起，通常结果会比最坏的好一些，比最好的坏一些。集成学习把多个学习器结合起来，如何获得比最好的单一学习器更好的性能呢？

考虑一个例子：二分类问题中，假定三个分类器在三个测试样本上表现，如下图所示。打对勾的表示正确分类，打叉号的表示分类错误。集成学习的结果通过投票法voting产生。即少数服从多数。第一个图中每个分类器有66.6%的精度，但集成学习却达到了100%。第二个图中三个分类器没有差别，但是集成之后性能却没有什么提高。第三幅图中每个分类器的精度都只有33.3%，集成学习的结果更糟糕。



这个例子我们可以总结出：要获得好的集成，个体学习器应有一定的**“准确性”**，即学习器不能太坏，并且要有“多样性”，即学习器之间具有**差异**。

### 4.3Bagging算法

##### 4.3.1Bootstrap sampling自助采样

之前的课程已经讲过模型的评估方法中有留一法（将数据集划分为两个互不相交的集合，一个做测试集，一个做训练集）和交叉验证方法（将数据分成k个大小相似互不相交的子集，每次使用k-1个子集做训练集，剩下的一个子集做测试集，以此循环进行k次的训练和测试，最后返回k次测试结果的均值。）。但是上述两种方法中都保留了一部分样本用于测试，所以实际模型所使用的训练集比源数据都要小，因此就会引入一些因训练样本规模不同而导致的估计偏差。另外一方面留一法受训练样本影响较小，但是计算复杂度又太高。因此为了解决减少训练样本规模不同造成的影响，同时还能比较高效地进行测试集的评估。自助法就是很好的解决方案。

boostrap抽样

在样本集D（样本数为m）内有放回的抽样，抽取数为m，每次抽取的概率相等为1/m，可能重复抽取。我们做一个简单的估计，样本m次采样中适中不被采样的概率为

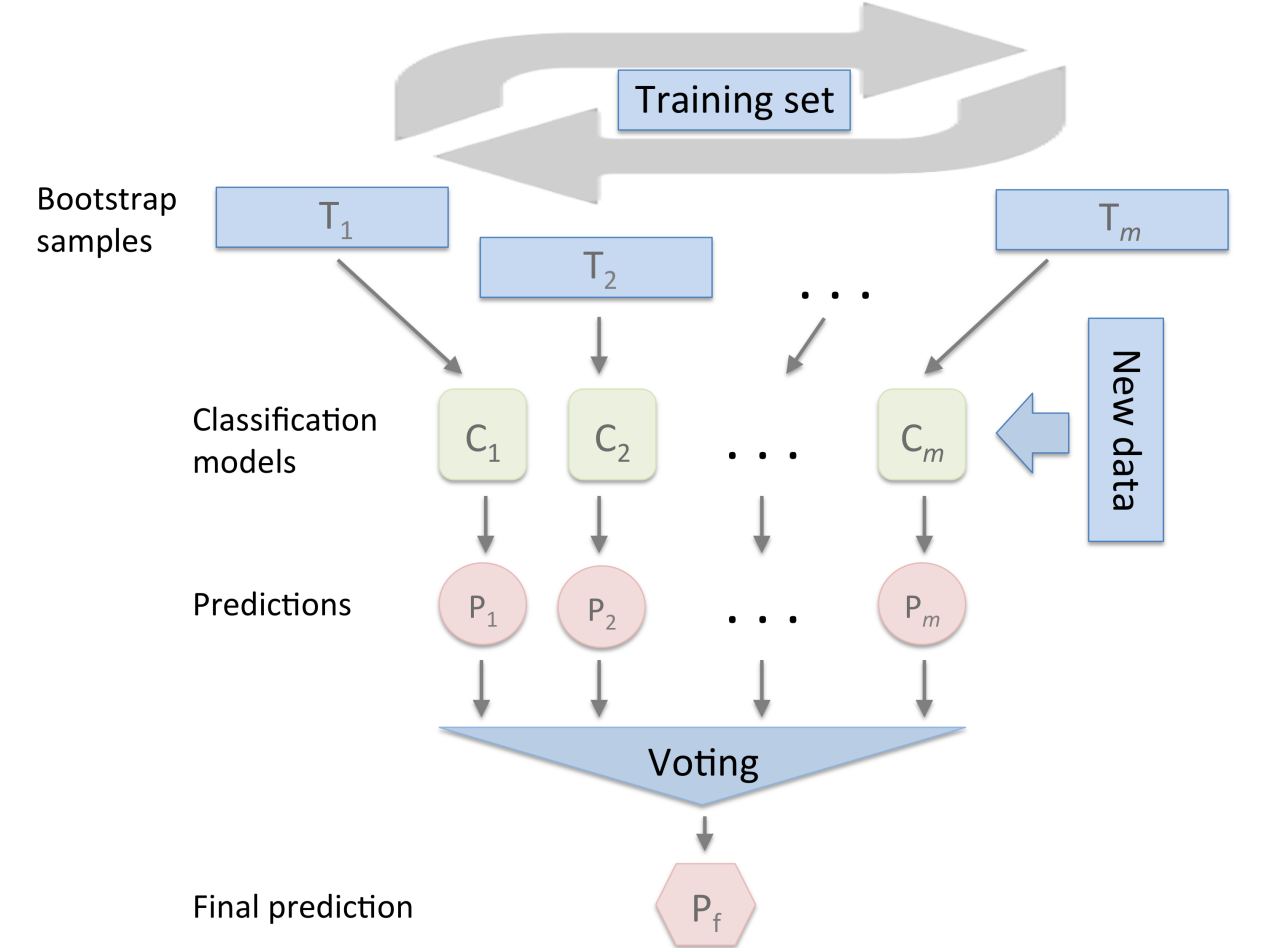


取极限得到原数据集D中36.8%的样本没有出现在采样数据集D1中。我们可以使用D1作为训练集，D-D1作为测试集。这样实际评估的模型与期望的模型都使用m个训练样本，而我们仍有数据总量的1/3的，没有在训练集中出现的样本用于测试。术语“包外估计”可以解释上述过程。



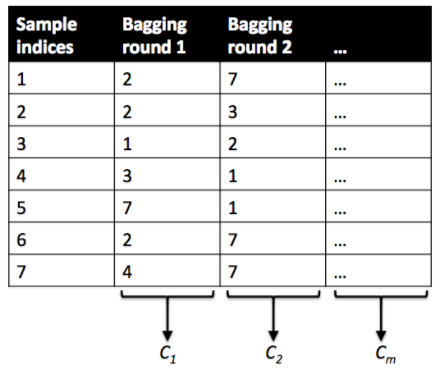
##### 4.3.2 Bagging算法

此算法没有使用相同的训练集拟合集成分类器中的单个分类器，而是采用了Bootstrap有放回抽样选取训练数据集。



下图有7个不同的训练样例，每一轮的bagging循环中，样本数据均可放回随机抽样，每个bootstrip抽样都是被用于分类器C的训练，这就是一颗典型的未剪枝的决策树。

随机森林是bagging的特例，它使用了随机特征子集去拟合单颗决策树。



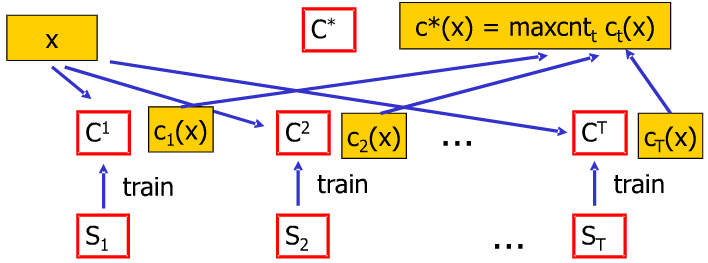
##### 4.3.3算法流程



Bagging基本流程：通过上述自助采样，采出T个含m个训练样本的采样集，然后基于每个采样集训练出一个基学习器，在将这些基学习器进行组合。

在对预测输出进行结合的时候，Bagging通常对分类任务使用简单投票法，对回归任务进行简单的平均法。但是如果投票个数一致，则最简单的做法是随机选择一个类别，当然也可以进一步考察学习器投票的置信度来确定最终的分类。

##### 4.3.4算法图解分析



基本分类器可以是决策树，逻辑回归等基分类器。

对于稳定性不好的分类器很实用，通过多数投票，减小了泛化误差，而对于稳定的分类器，集成效果并不明显。

##### 4.3.5 Bagging性能

（1）Bagging是一个很高效的集成学习算法

（2）Bagging与下面讲的AdaBoost只适用于二分类不同，它能不经修改地用于多分类、回归任务。

（3）自助bootstrap采样过程还给Bagging带来了另一个优点：由于每个基学习器只使用了初始训练集中约63.2%的样本，剩下的约36.8%样本可用作验证集来泛化性能进行“包外样本评估（即：不同于训练数据的样本）”。

（4）从偏差-方差分解角度看，Bagging主要关注降低方差，因此他在不剪枝决策树、神经网络等易受样本扰动的学习器上效果更为明显。

##### 4.3.6Bagging算法总结

Bagging算法首先采用M轮自助采样法，获得M个包含N个训练样本的采样集。然后，基于这些采样集训练出一个基学习器。最后将这M个基学习器进行组合。组合策略为：

* 分类任务采用简单投票法：即每个基学习器一票
* 回归问题使用简单平均法：即每个基学习器的预测值取平均值。

### 4.4随机森林算法详解

随机森林就是建立很多决策树，组成一个决策树的“森林”，通过多棵树投票来进行决策。这种方法能够有效地提高对新样本的分类准确度。

随机森林在以决策树为基学习器构建Bagging集成（样本的随机选取）的基础上，进一步在决策树的训练过程中引入随机属性选择。具体来说，传统决策树在选择划分属性时是在当前节点的属性集合（假设有d个属性）中选择一个最优属性；而在RF随机森林中，对基决策树的每个节点，先从该节点的属性集合中随机选择一个包含K个属性的子集，然后在从这个子集中选择一个最优属性用于划分。K=d就是传统决策树，K=1则是随机选取一个属性用于划分，一般情况。

##### 4.4.1随机森林的步骤

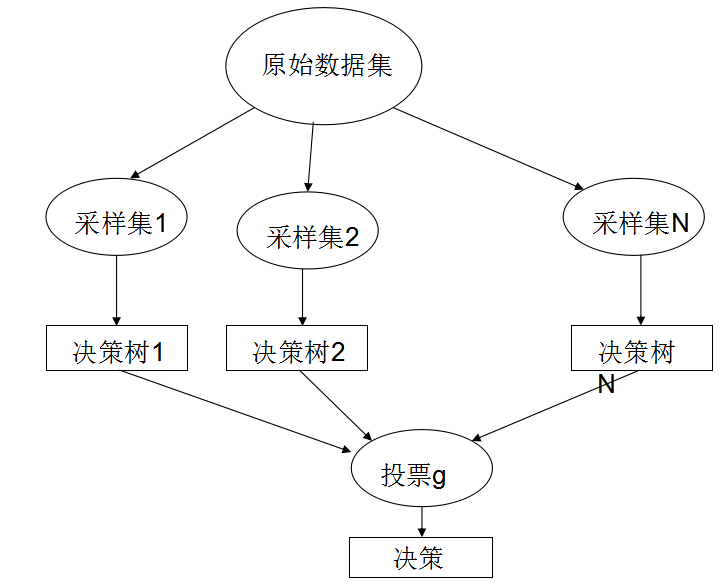
首先，对样本数据进行有放回的抽样，得到多个样本集。具体来讲就是每次从原来的N个训练样本中有放回地随机抽取N个样本(包括可能重复样本)。

然后，从候选的特征中随机抽取m个特征，作为当前节点下决策的备选特征，从这些特征中选择最好地划分训练样本的特征。用每个样本集作为训练样本构造决策树。单个决策树在产生样本集和确定特征后，使用CART算法计算，不剪枝。

最后，得到所需数目的决策树后，随机森林方法对这些树的输出进行投票，以得票最多的类作为随机森林的决策。

说明：

（1）随机森林的方法即对训练样本进行了采样，又对特征进行了采样，充分保证了所构建的每个树之间的独立性，使得投票结果更准确。



（2）随机森林的随机性体现在每棵树的训练样本是随机的，树中每个节点的分裂属性也是随机选择的。有了这2个随机因素，即使每棵决策树没有进行剪枝，随机森林也不会产生过拟合的现象。

随机森林中有两个可控制参数：

* + 森林中树的数量（一般选取值较大）
  + 抽取的属性值m的大小。

##### 4.4.2随机森林的特点总结

随机森林简单、容易实现、计算开销小，被誉为“代表集成学习计数水平的方法”。可以看出随机森林只是对Bagging做了很小的改动。Bagging的多样性只是体现在样本的随机性，随机森林的基学习器的多样性不仅来自于样本的随机性，还来自于属性的随机性。所及森林随着学习器数目的增加，随机森林通常会收敛到更低的泛化误差。

（1）分类结果更加准确

（2）可以处理高维度的属性，并且不用做特征选择

（3）即使有很大部分数据遗失，仍可以维持高准确度

（4）学习过程快速

（5）在训练完成后，能够给出哪些属性比较重要

（6）容易实现并行化计算

（7）在训练过程中，能够检测到属性之间的相互影响