**Spark机器学习分类算法基础**

**教案**

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

# SparkMliib构建分类算法

学习目标：

* 决策树算法原理
* 随机森林原理
* 如何利用决策树算法、随机森林算法、GBDT算法建模
* XGBOOST+Spark的XGBOOST4J应用

## 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？

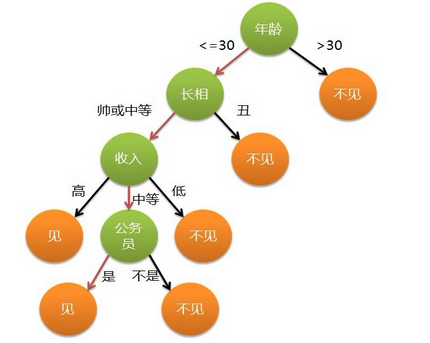
母亲：不算很高，中等情况。

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。

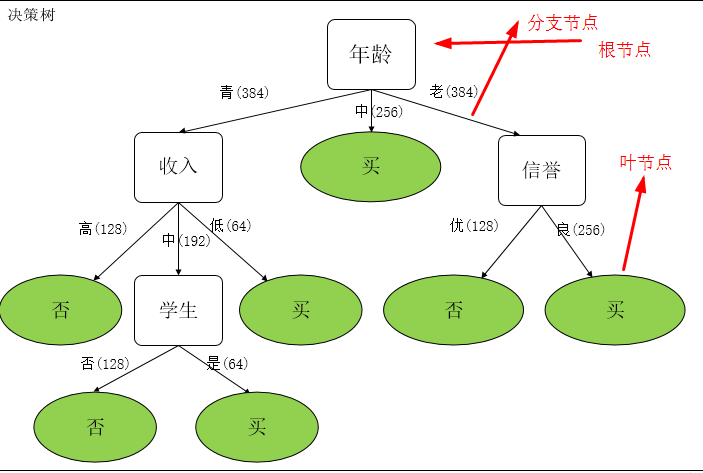
递归直到所有训练样本都被很好的分类，或没有合适的特征为止。对于是否较好的分类，我们借助“信息增益”、“信息增益率”、“基尼系数”、“分类错误率”等指标衡量。

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？**

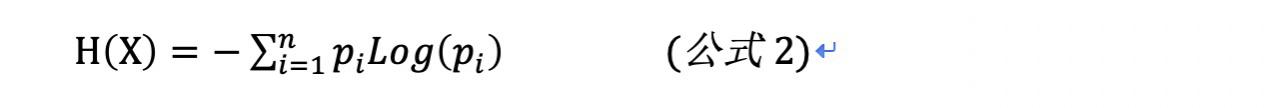
既然决策树有根节点、内部节点和叶子节点之分，那么究竟那个属性做根节点，那个属性又做下一级分类的分支节点(内部节点)?

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

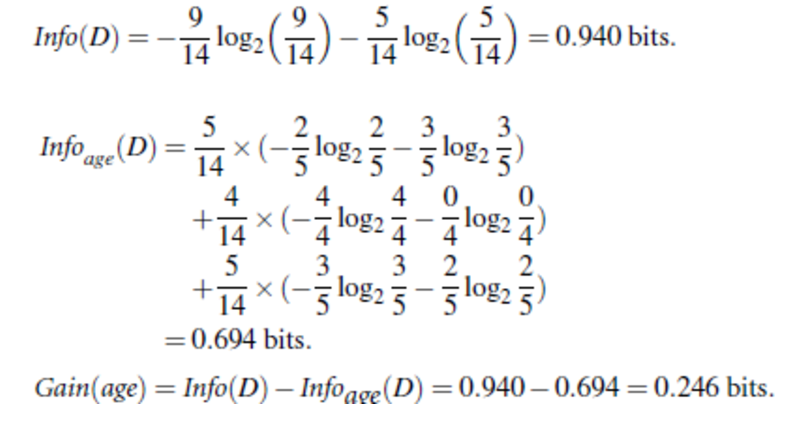
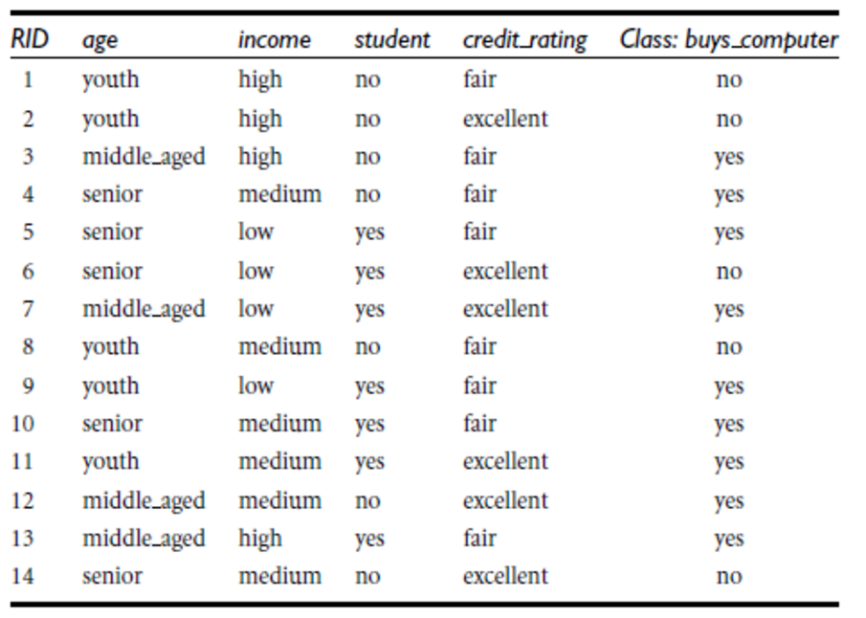
2.生成剩余属性集合AT=A-{a}

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）

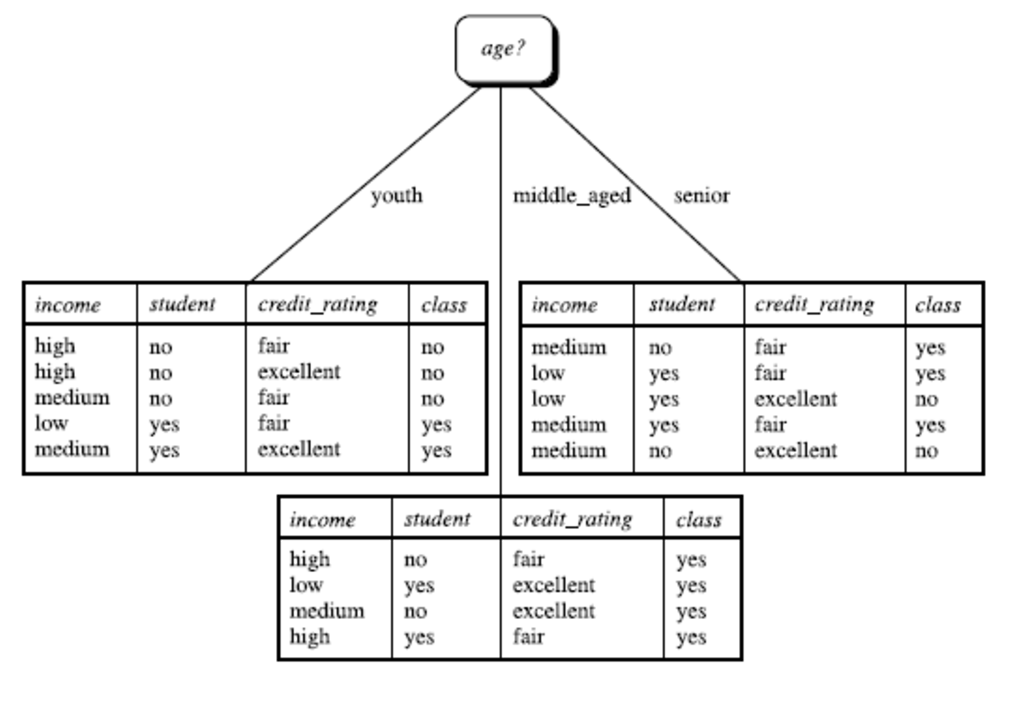
有了以上理论基础我们可以接下来通过一个小例子完整计算信息熵和信息增益，从而通过ID3算法构造决策树。



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：

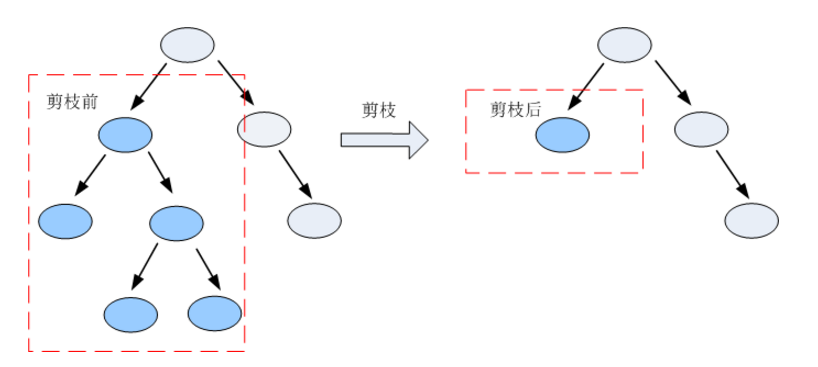
**贪心算法：**顾名思义，贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。

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

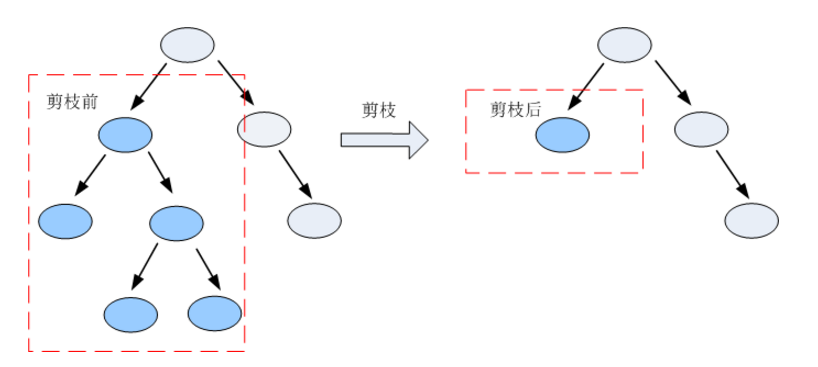
预剪枝使得决策树的很多分支没有展开，这不仅仅降低了过拟合的风险，还显著减少了决策树的训练时间开销和测试时间开销。但另一方面，有些分支的当前划分虽不能提升泛化性能，甚至会导致泛化性能降低，但在其基础上进行的后续划分却有可能导致性能的显著提高。同时，预见值决策树也带来了欠拟合的风险。

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。

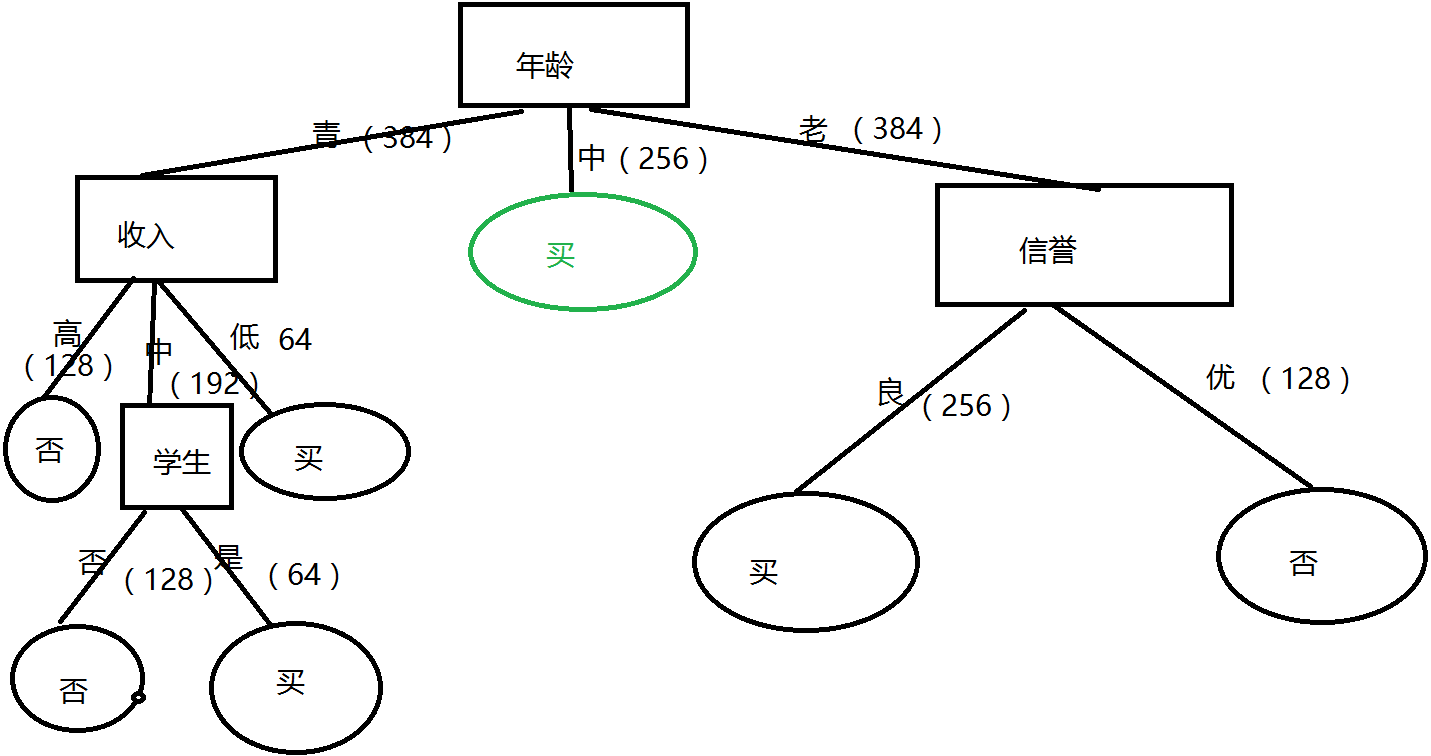
④根据节点的不同取值将数据集拆分为若干子集，删除当前的特征列，在计算剩余列的特征信息熵。如果有信息增益，就重复第二步直至划分结束。

* 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

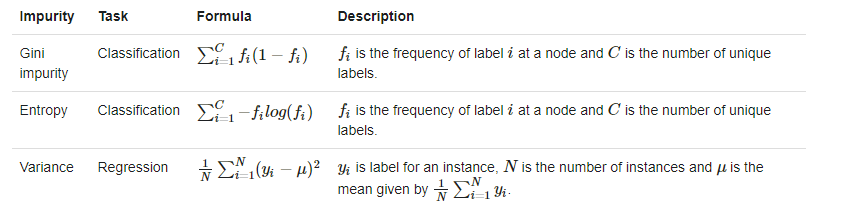
[决策树](http://en.wikipedia.org/wiki/Decision_tree_learning)及其集合是分类和回归的机器学习任务的流行方法。决策树被广泛使用，因为它们易于解释，处理分类特征，扩展到多类分类设置，不需要特征缩放，并且能够捕获非线性和特征交互。诸如随机森林和增强的树集合算法是分类和回归任务的最佳表现者。

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 |

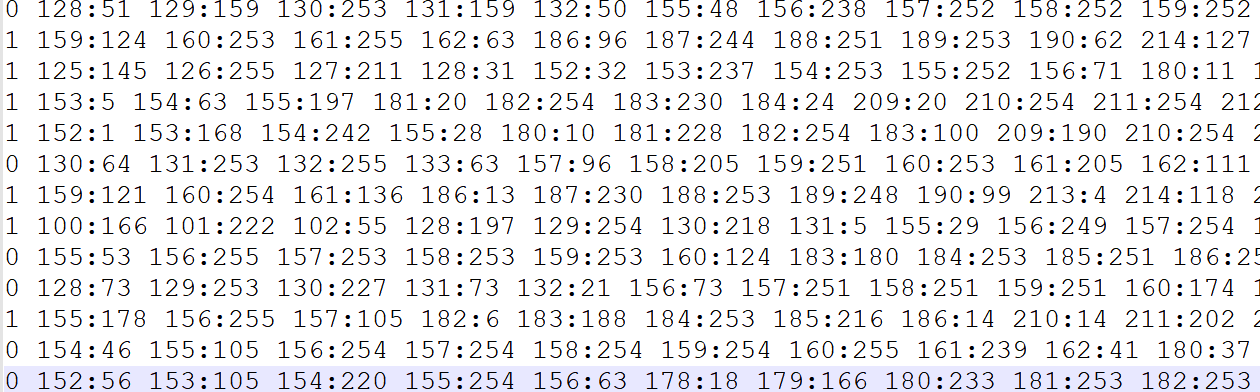
**Output Columns**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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)

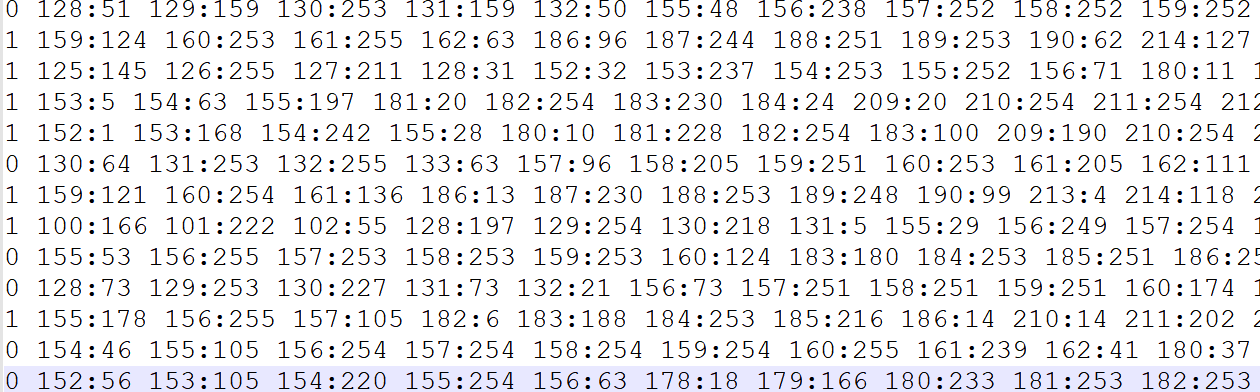
println("Test Error = " + (1.0 - accuracy))

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)

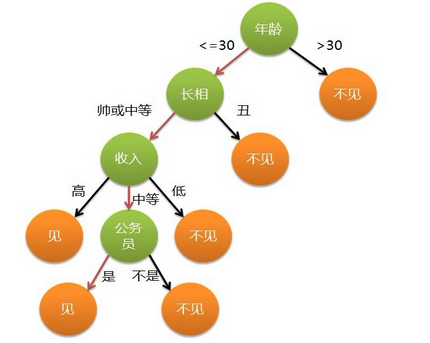
println("Learned classification tree model:\n" + model.toDebugString)

// 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()

}

}

## 1.3.SparkMllib的集成学习的原理及实战

### 1.3.1 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。

#### （1）Cart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。

【思考】

对于多个离散取值如何处理？双化

将多个离散取值分为任意两个取值的集合。如：青年，中年，老年可以划分为{青年、非青年}、{老年、非老年}、{中年、非中年}

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。

**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。

#### （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。

### 1.3.2回归树原理

#### （1）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。

**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）



【疑问】为什么是平均值？

证明：**均方误差的目标函数最优值（取定值）是均值**



同时， **绝对误差的目标函数的最优解是中位数**

* **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个变量和它取的值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)。

#### （2）回归树算法总结

算法（最小二乘回归树生成算法）：

输入：训练数据集D

输出：回归树f(x)

在训练数据集所在的输入空间中，递归地将每个区域划分为两个子区域并决定每个子区域上的输出值，构建二叉决策树：

（1）选择最优切分变量j和切分点s：求解：



遍历变量j，对固定的切分变量j扫描切分点s，选择上式达到最小值的对(j,s).

1. 用选定的对(j,s)划分区域并决定相应的输出值：





（3）继续对两个子区域调用步骤(1),(2)，直至满足停止条件：

（4）将输入空间划分为M个区域R1,,R2......RM，生成决策树：

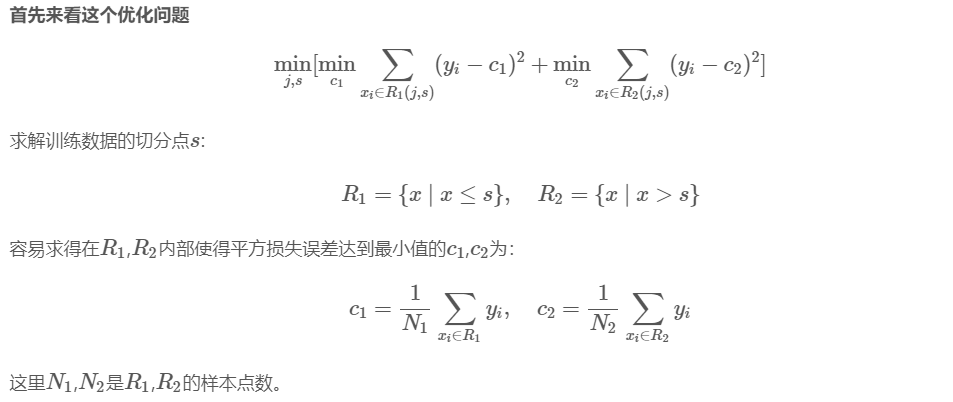


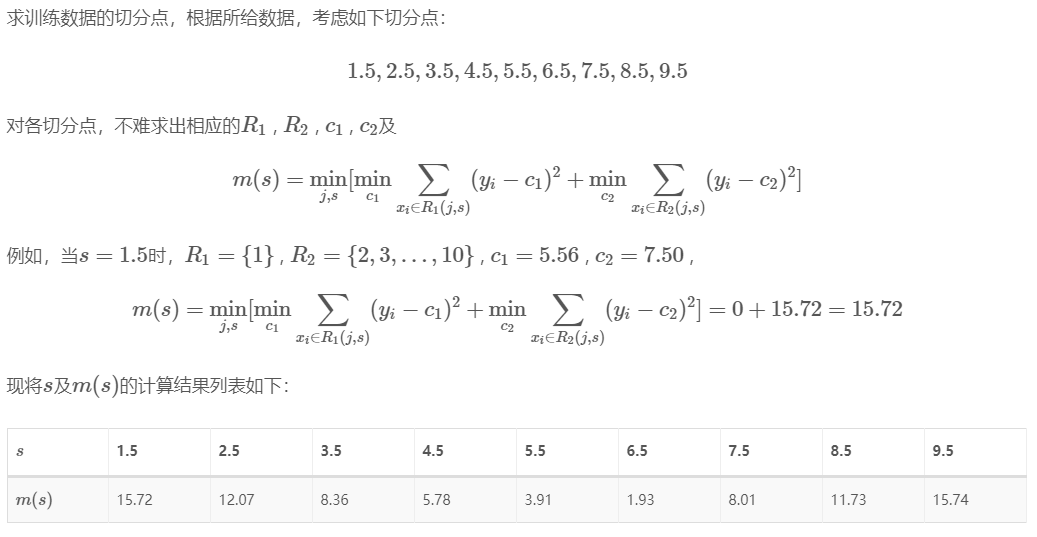
#### （3）回归树案例

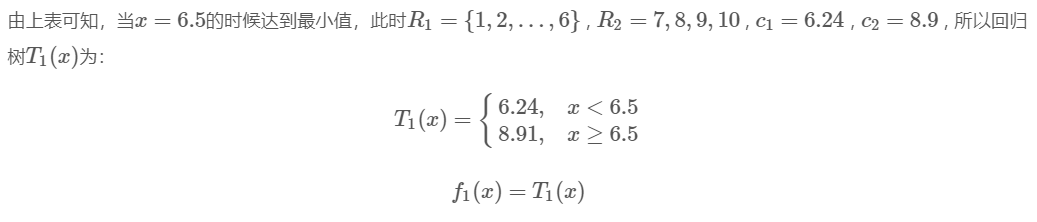
上述算法原理的不容易理解，下面举个例子来说明。

训练数据见下表，x的取值范围为区间[0.5,10.5],y的取值范围为区间[5.0,10.0],学习这个回归问题的最小二叉回归树。

| **X** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| y | 5.56 | 5.70 | 5.91 | 6.40 | 6.80 | 7.05 | 8.9 | 8.70 | 9.00 | 9.05 |







### 1.3.3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}，{少年、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：

* 表面覆盖为毛发和非毛发
* 表面覆盖为鳞片和非鳞片
* 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。

#### （1）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）。

#### （2）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。

①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)



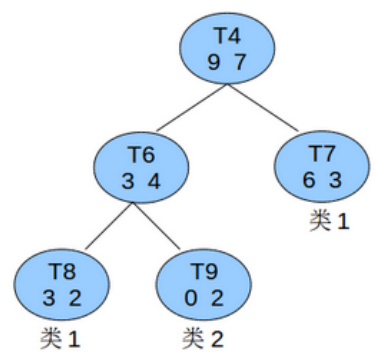
是是子树中包含的叶子节点个数。

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。

#### （3）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

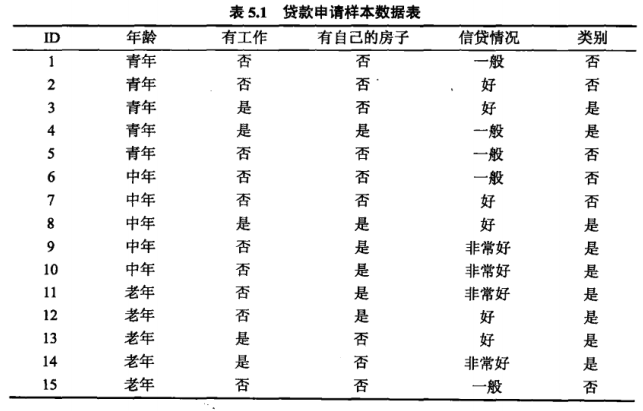
1. 根据训练数据集，从根节点开始，递归地对每个节点进行以下操作，构建二叉决策树：
2. 设结点的训练数据集为D，计算现有特征对该数据集的基尼指数，此时对每一个特征A，对其可能取得每个值a，根据样本点对A=a的测试为“是”或“否”将D分割成D1和D2两部分，利用集合的基尼指数公式计算A=a的基尼指数。



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）分类决策树的例子

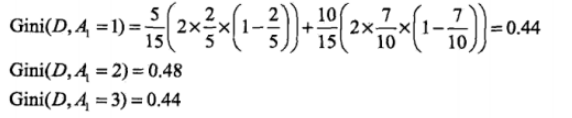
数据集：



应用上述数据集，应用CART算法生成决策树。

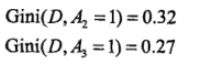
分析：首先计算各特征的基尼指数，选择最优特征以及其最优切分点。我们以A1,A2,A3,A4表示**年龄、有工作、有自己的房子和信贷**4个特征，并以1，2，3表示年龄的值为青年、中年、老年，以1、2表示有工作和有自己的房子的值为是和否，以1、2、3表示信贷情况的值为非常好、好和一般。

求特征A1的基尼指数：



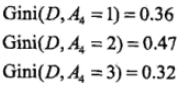
优于Gini(D,A1=1)和Gini(D,A1=3)相等，且最小，所以A1=1和A1=3都可以选作最优切分点。

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。

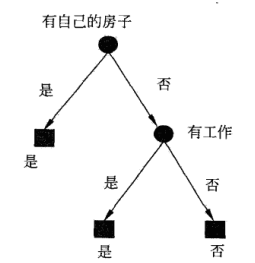
求解特征4的基尼指数：



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。



#### （5）Cart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。

## 1.4集成学习分类的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。

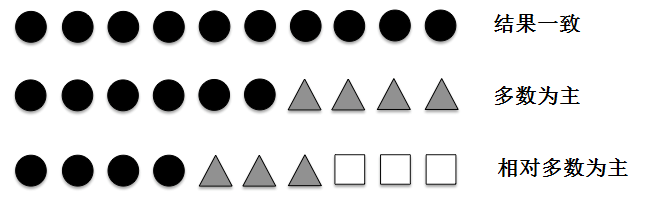
### 1.4.1集成学习关键要素

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。



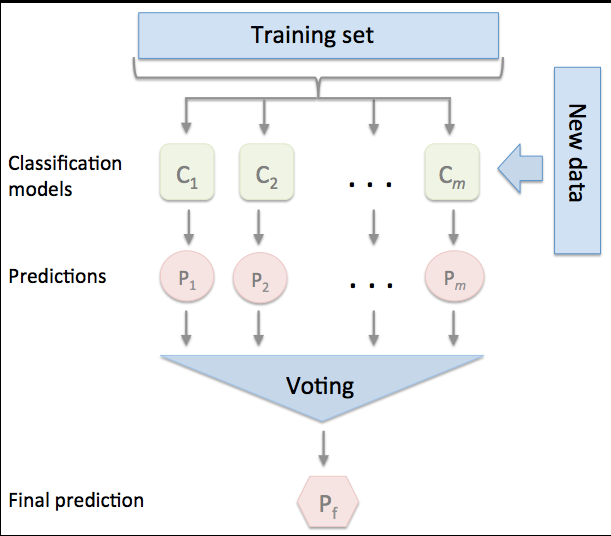
个体的分类器由一个现有的学习算法从训练数据产生，例如：C4.5决策树算法，BP神经网络算法等。我们一般把个体分类器全部为同种的分类器称为“同质的”，如全部为决策树模型。同质的集成学习中的个体学习器称为“基学习器（base learner）”，相应的算法称为基学习算法。反之，集成中包含不同种的学习器，我们称之为“异质”的，异质集成中的个体学习器包含不同的学习算法组成的，这种情况下的学习器称为“组合学习器”。相信这些名词大家能够了解。



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。

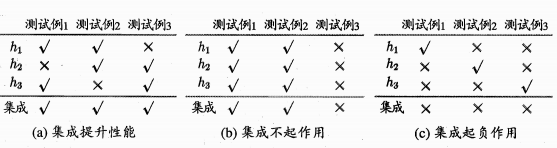
下图是使用多数投票法通用集成方法的概念：



### 1.4.2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**。

## 1.5集成学习分类之Bagging算法

### 1.5.1Bootstrap 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

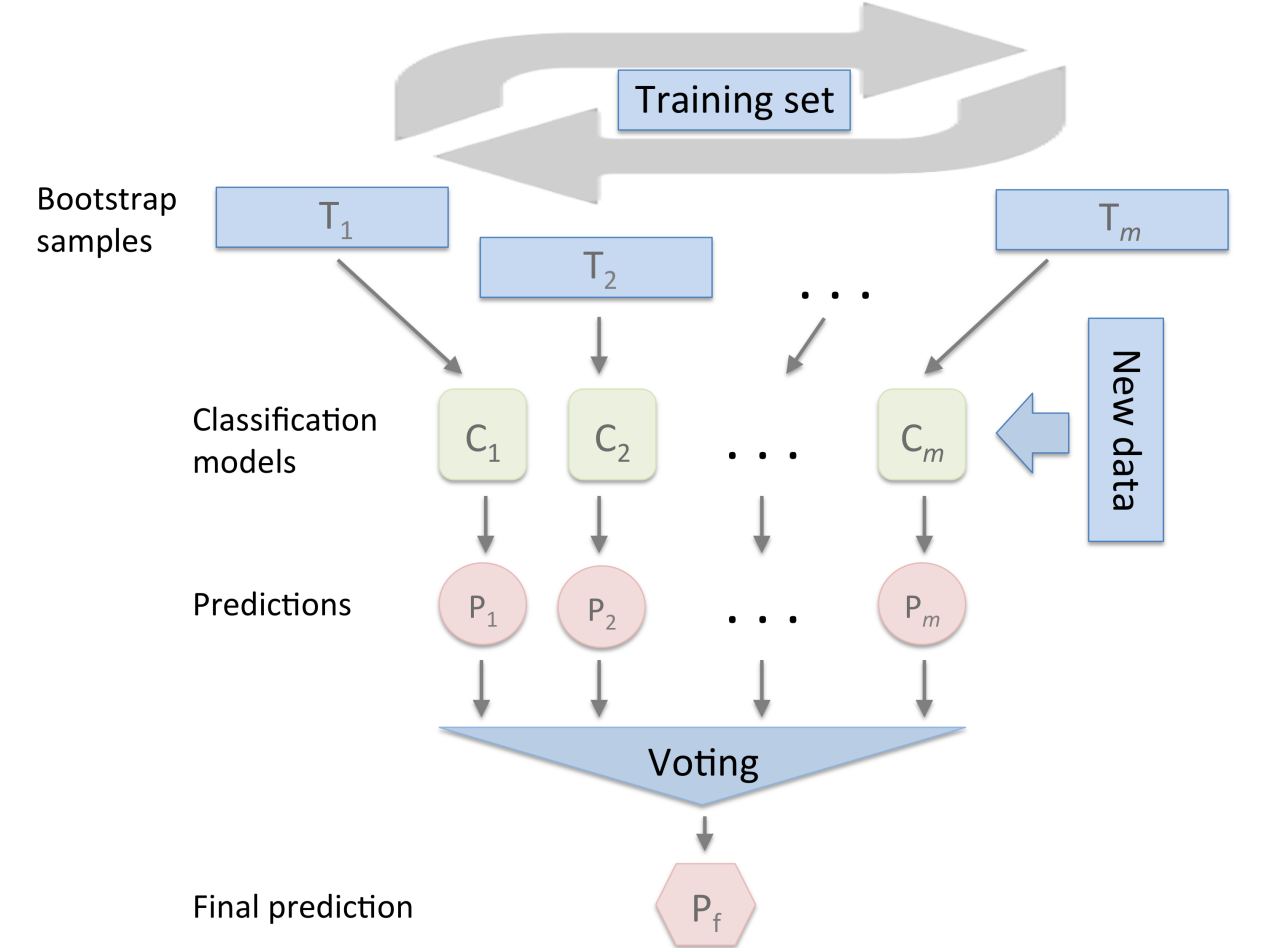


取极限得到原数据集D中36.8%的样本没有出现在采样数据集D1中。我们可以使用D1作为训练集，D-D1作为测试集。这样实际评估的模型与期望的模型都使用m个训练样本，而我们仍有数据总量的1/3的，没有在训练集中出现的样本用于测试。术语“包外估计”可以解释上述过程。



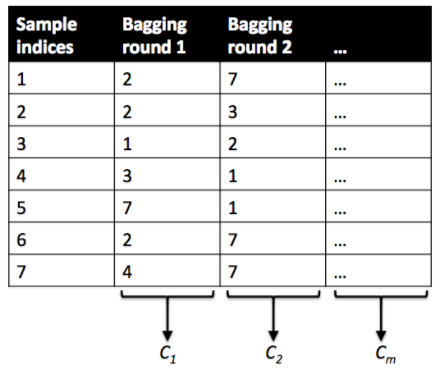
### 1.5.2 Bagging算法

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。



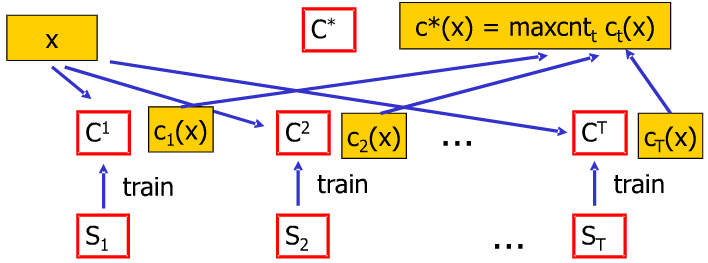
### 1.5.3算法流程



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。

### 1.5.4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。

### 1.5.5Bagging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。

### 1.5.5Bagging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。

## 1.6集成学习分类之随机森林算法详解

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。

### 1.6.1集成学习分类之随机森林的步骤

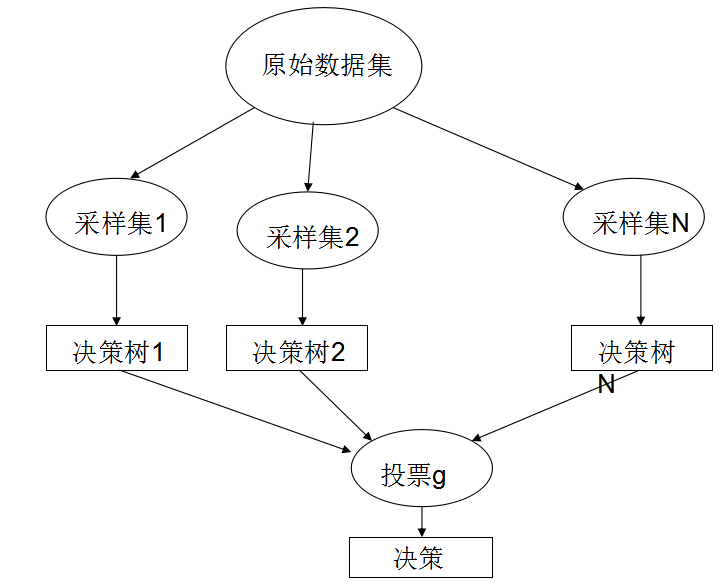
首先，对样本数据进行有放回的抽样，得到多个样本集。具体来讲就是每次从原来的N个训练样本中有放回地随机抽取N个样本(包括可能重复样本)。

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。

### 1.6.2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

### 1.6.3SparkMllib中随机森林参数详解

[随机森林](http://en.wikipedia.org/wiki/Random_forest)是[决策树的](http://spark.apache.org/docs/2.2.0/mllib-decision-tree.html)集合。随机森林是用于分类和回归的最成功的机器学习模型之一。它们结合了许多决策树，以降低过度拟合的风险。与决策树一样，随机森林处理分类特征，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。

spark.mllib支持使用连续和分类特征的随机森林进行二元和多类分类以及回归。 spark.mllib使用现有的[决策树](http://spark.apache.org/docs/2.2.0/mllib-decision-tree.html)实现实现随机森林。

#### （1）基本算法

随机森林分别训练一组决策树，因此训练可以并行完成。该算法将随机性注入训练过程，以使每个决策树略有不同。组合每棵树的预测可以减少预测的方差，从而提高测试数据的性能。

#### （2）训练

训练过程的随机性包括：

* 在每次迭代时对原始数据集进行子采样以获得不同的训练集。
* 考虑在每个树节点处拆分的不同随机特征子集。

除了这些随机化之外，决策树训练的方式与单个决策树的方式相同。

#### （3）预测

要对新实例进行预测，随机林必须聚合来自其决策树集的预测。对于分类和回归，这种聚合的方式不同。

* *分类*：多数投票。每棵树的预测都算作一个类的投票。预计该标签是获得最多选票的类别。
* *回归*：平均。每棵树预测一个真实的价值。预测标签是树预测的平均值。

#### （4）参数

这里通过讨论各种参数，包括一些使用随机森林的指南。这里省略了一些决策树参数。

这里提到的前两个参数是最重要的，调整它们通常可以提高性能：

* **numTrees：森林中的树木数量。**
  + - 增加树的数量将减少预测的方差，从而提高模型的测试时间准确性。
    - 训练时间在树木数量上大致线性增加。
* **maxDepth：森林中每棵树的最大深度。**
  + - 增加深度使模型更具表现力和强大。然而，深树需要更长的时间训练，也更容易过度拟合。
    - 通常，使用随机森林时比使用单个决策树训练更深的树是可以接受的。一棵树比随机森林更容易过度拟合（因为在森林中平均多棵树的方差减少）。

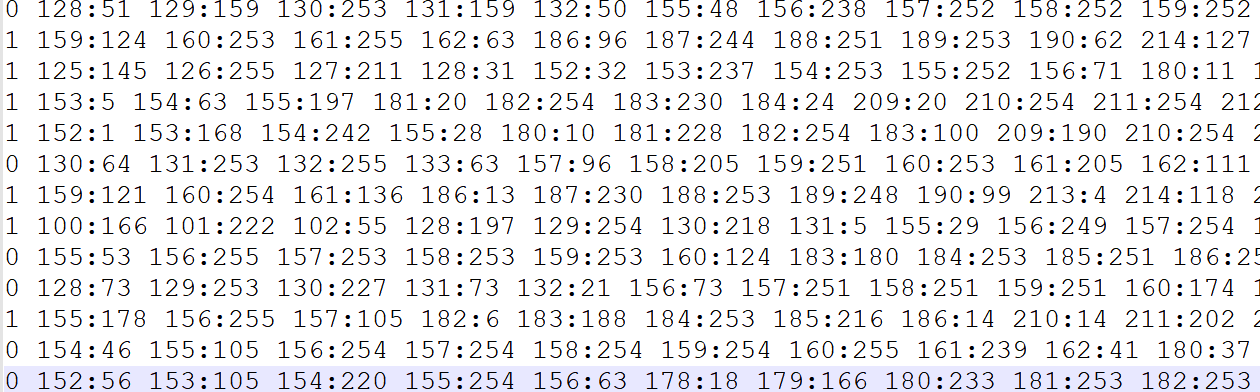
接下来的两个参数通常不需要调整。但是，他们可以通过调整来加速培训。

* subsamplingRate：此参数指定用于训练林中每棵树的数据集的大小，作为原始数据集大小的一部分。建议使用默认值（1.0），但减少此分数可以加快训练速度。
* featureSubsetStrategy：要用作每个树节点处拆分的候选项的特征数。该数字被指定为特征总数的分数或函数。减少这个数字会加快培训速度，但如果太低，有时会影响性能。

### 1.6.4随机森林建模实战

#### （1）SparkMl代码实战

**数据：**



**代码：**

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.classification.{RandomForestClassificationModel, RandomForestClassifier}

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

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

// Load and parse the data file, converting it to 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.

// Set maxCategories so features with > 4 distinct values are treated as continuous.

val featureIndexer = new VectorIndexer()

.setInputCol("features")

.setOutputCol("indexedFeatures")

.setMaxCategories(4)

.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 RandomForest model.

val rf = new RandomForestClassifier()

.setLabelCol("indexedLabel")

.setFeaturesCol("indexedFeatures")

.setNumTrees(10)

// Convert indexed labels back to original labels.

val labelConverter = new IndexToString()

.setInputCol("prediction")

.setOutputCol("predictedLabel")

.setLabels(labelIndexer.labels)

// Chain indexers and forest in a Pipeline.

val pipeline = new Pipeline()

.setStages(Array(labelIndexer, featureIndexer, rf, 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)

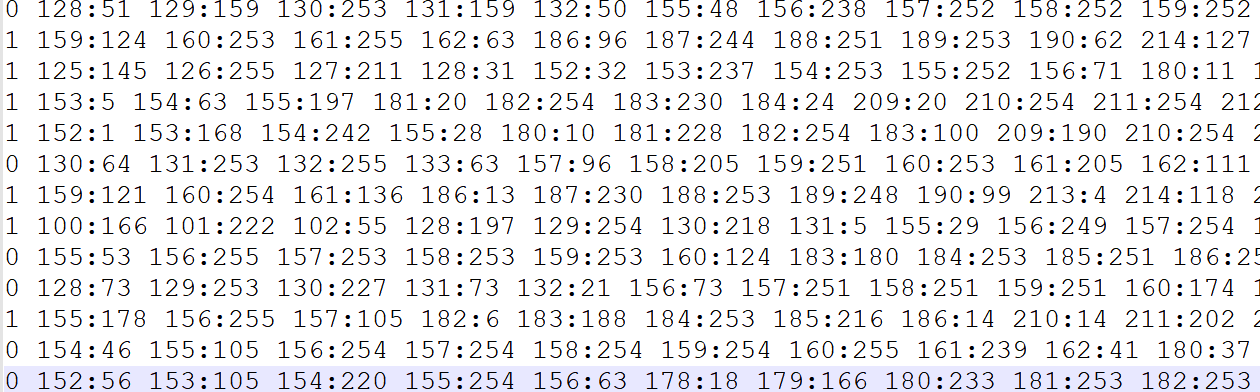
println("Test Error = " + (1.0 - accuracy))

val rfModel = model.stages(2).asInstanceOf[RandomForestClassificationModel]

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

#### （2）SparkMlllib代码实战

**数据：**



**代码：**

import org.apache.spark.mllib.tree.RandomForest

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

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 RandomForest model.

// Empty categoricalFeaturesInfo indicates all features are continuous.

val numClasses = 2

val categoricalFeaturesInfo = Map[Int, Int]()

val numTrees = 3 // Use more in practice.

val featureSubsetStrategy = "auto" // Let the algorithm choose.

val impurity = "gini"

val maxDepth = 4

val maxBins = 32

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

numTrees, featureSubsetStrategy, 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 forest model:\n" + model.toDebugString)

// Save and load model

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

val sameModel = RandomForestModel.load(sc, "target/tmp/myRandomForestClassificationModel")

## 1.7Boosting框架简介

### 1.7.1算法机制

Boosting是一族可将弱学习器升为强学习器算法。这类算法的工作机制类似：

1.先从初始训练集训练出一个基学习器

2.在根据基学习器的表现对训练样本分布进行调整，使得先前基学习器做错的训练样本在后续得到最大的关注。

3.然后基于调整后的样本分布来训练下一个基学习器；

4.如此重复进行，直至基学习器数目达到实现指定的值T为止。

5.再将这T个基学习器进行加权结合得到集成学习器。

Boosting算法的著名代表就是Adaboost算法。

因此，对于Boosting算法，存在两个问题：

1. 在每一轮中如何调整训练集，使训练的弱分类器得以进行；（调整样本权值）

2. 如何将各个弱分类器联合起来形成强分类器。 （调整模型权值）

### 1.7.2提升的概念强化

提升是一个机器学习技术，可以用于回归和分类问题，它每一步产生一个弱预测模型（如决策树）并加权累加到总模型中；如果每一步的弱预测模型生成都是依据损失函数的梯度方向，则称之为梯度提升（GradientBoosting）。

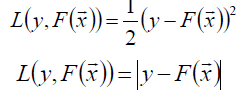
梯度提升算法首先给定一个目标损失函数，它的定义域是所有可行的弱函数集合（基函数）；提升算法通过迭代的选择一个负梯度方向上的基函数来逐渐逼近局部极小值。这种在函数域的梯度提升观点对机器学习有很大影响。

提升的理论意义：如果一个问题存在弱分类器，则可以通过提升的办法得到强分类器。

**提升算法框架：**

给定输入向量(x1,y1),(x2,y2).....(xn,yn),目标是找到近似函数F(x)，使得损失函数L(y,F(x))的损失值最小。

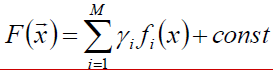
L(y,F(x))常用的两种定义：



假定最优函数为F\*(x),即



提升算法就是假定F(x)是一族基函数fi(x)的加权和



**推导补充：**

**均方误差的目标函数最优值（取定值）是均值**

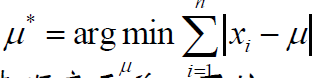


**绝对误差的目标函数的最优解是中位数**

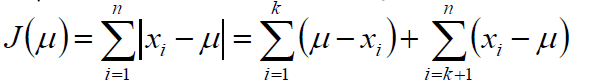
**推导：**给定样本



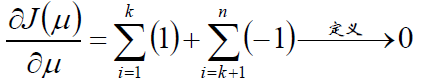
计算



为了便于推导，由于样本顺序无关，不妨假定样本是递增排序的，则



也就是有一部分x是大于u的，有一部分是小于u均值的。



从而，前k个样本数目与后k-1个样本数目相同，即u是中位数。

## 1.8集成学习分类之Adaboost算法原理及实战

Adaptive Boosting称之为自适应boosting，利用该学习器可以提高学习器的性能。

### 1.8.1AdaBoost简介

AdaBoost自适应提升学习算法和Boosting考虑的点一样

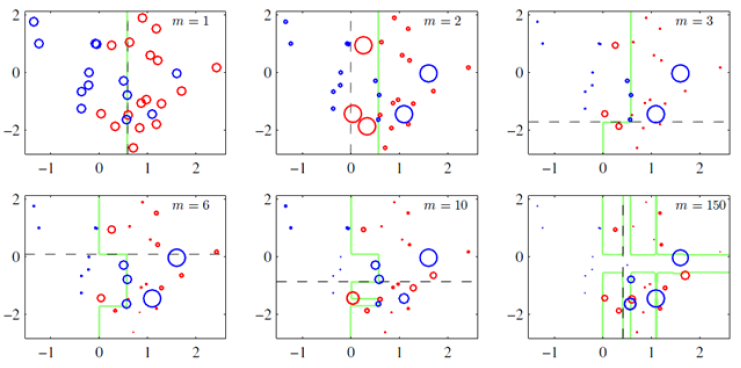
Adaboost自适应在于：“关注”被错分的样本，“器重”性能好的弱分类器：**（观察下图）**

（1）不同的训练集--->调整样本权重

（2）“关注”--->增加错分样本权重

（3）“器重”--->好的分类器权重大

（4） 样本权重间接影响分类器权重



AdaBoost算法的两个核心步骤：

* + 权值调整：AdaBoost算法提高那些被前一轮基分类器错误分类样本的权值，而降低那些被正确分类样本的权值。从而使得那些没有得到正确分类的样本，由于权值的加大而受到后一轮基分类器的更大关注。
  + 基分类器组合：AdaBoost采用加权多数表决的方法。
* ~加大分类误差率较小的弱分类器的权值，使得它在表决中起到较大的作用。
* ~减小分类误差率较大的弱分类器的权值，使得它在表决中起较小的作用。

### 1.8.2AdaBoost特点

AdaBoost把多个不同的弱分类算法，用一种非随机的方式组合起来，表现出惊人的性能。

1，可以使用各种方法构建子分类器,Adaboost算法提供的是框架；

2，子分类器容易构造；

3，速度快，且基本不用调参数；

4，泛化错误率低。

### 1.8.3AdaBoost步骤

**Adaboost迭代算法有3步：**

1.初始化训练数据的权值分布：假设有N个样本，每个样本赋予相同权值1/N。

2.训练弱分类器：本轮训练中，若某样本分错，则提高它的权值，相反分类正确的样本被降低权值。然后，权值更新过的全体样本被用于训练下一个分类器，使得下一个分类器更关注权重大的难分样本。多次迭代，训练多个弱分类器。

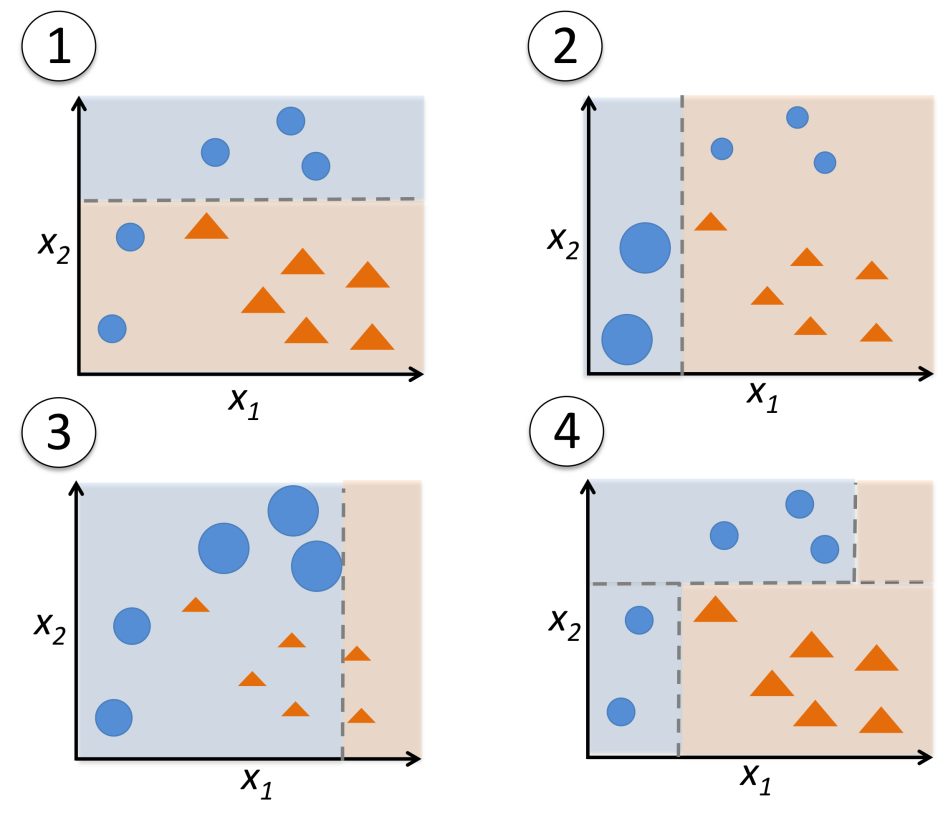
3.加权组合弱分类器：加大分类误差率小的弱分类器的权重，使其在最终表决中起较大作用，而降低分类误差率大的弱分类器的权重，使其在最终表决中起较小作用。

**详细步骤举例：**

AdaBoost和Boosting不同，AdaBoost使用了整个训练集来训练学习器，其中每个训练样本在每次迭代中都会重新被赋值一个权重，在上一弱学习器错误的基础上进行学习进而构建一个更加强大的分类器。下图1代表了一个2类别问题的训练集，其中所有的样本都被赋予了相同的权重，基于此训练集我们得到了一颗单层决策树，图中的虚线表示，它尝试通过最小代价函数（或决策树的不纯度）划分两类样本（三角形或圆形）。

在下一轮中，图2，我们为前面无分类的样本（圆形）赋予了更大的权重，此外，我们还降低了被正确分类的权重，图2错误的划分了圆形类的三个样本。被分错的样本在第三轮Bossting中，被赋予了更大的权重。

假定AdaBoost只包含了3轮的Boosting过程没我们就能够用加权投票方式将不同重采样训练子集上得到的三个弱学习器进行组合，也就是图4（加权组合弱分类器）。



### 1.8.3AdaBoost数学定义

考虑二分类问题，问题，数据集如下



1.初始化训练数据的权值分布



2.用m=1,2,...,M表示迭代次数

2.1使用具有权值分布的训练数据集学习得到基本分类器（KNN、决策树、SVM等）



2.2计算在训练数据集上的分类误差率

（不相等返回1，相等返回0）

注：（1）所有误分类点的权重之和。权重越大的误差分类点，其在误差率中占比越大。

（2）若算法终止，构建失败。（后面提到）

2.3计算的相关系数

（em做错的样本，1-em是做对的样本）

该系数表示在最终分类器中的重要性。它是em的单调减函数（说明误差越小的基本分类器，其重要性越高）。系数大于0也要求em<1/2

2.4更新训练数据集的权值分布





这里，Zm是规范化因子，它使成为一个概率分布



构建基本分类器的线性组合



这里实现了M个基本分类器的加权表决。系数表示基本分类器的重要性，它是em的单调减函数（说明误差越小的基本分类器，其重要性越高）

**4.得到最终分类器**



### 1.8.3Adaboost关键步骤解析

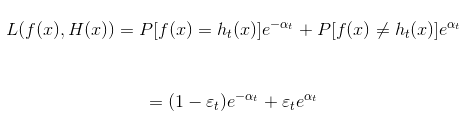
首先我们明确，若****算法终止，构建失败。

构建损失函数并求解am：这里用的是**指数损失函数**（exponential loss function）



其中f(x)是正确的分类，等于-1或者1，H(x)是分类器的分类结果，等于-1或者1。





所以对该式子求的偏导，得，并令其等于0，得

 其中

现在考虑要提高的那些被前一轮弱分类器错误分类样本的权值，而降低那些被正确分类样本的权值，就要有，因此。

另外：因此如果在第m次迭代过程中，发现基学习器的误差率，算法终止。因为不满足算法的条件：提高那些被前一轮弱分类器错分类样本的权值，而降低那些被正确分类样本的权值。

综上：当，

也就是错误率小于50%，根据系数am计算公式得到am>0，也就是说凡是错误率小于0.5我们都能得到一个大于0的am，即分类错误率em越小，我们就赋予更高的am权值，反之分类错误率em越大，我们就赋予am更低的权值，即降低权值。

### 1.8.4算法举例

下表给出了10个训练数据，假设弱分类器由x<v或x>v产生，其中阈值v使得该分类器在训练数据集上分类误差最低，这里v=3.0。试用Adaboost算法学习一个弱分类器。x是一维的样本数据，y为类别标签1/-1，x<=3是+1类，否则-1类。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 样本索引 | x | y | 权重值 | Y\_pred(x<=3.0) | 正确？ | 更新后的权值 |
| 1 | 1.0 | 1 | 0.1 | 1 | 是 | 0.072 |
| 2 | 2.0 | 1 | 0.1 | 1 | 是 | 0.072 |
| 3 | 3.0 | 1 | 0.1 | 1 | 是 | 0.072 |
| 4 | 4.0 | -1 | 0.1 | -1 | 是 | 0.072 |
| 5 | 5.0 | -1 | 0.1 | -1 | 是 | 0.072 |
| 6 | 6.0 | -1 | 0.1 | -1 | 是 | 0.072 |
| 7 | 7.0 | 1 | 0.1 | -1 | 是 | 0.167 |
| 8 | 8.0 | 1 | 0.1 | -1 | 是 | 0.167 |
| 9 | 9.0 | 1 | 0.1 | -1 | 是 | 0.167 |
| 10 | 10.0 | -1 | 0.1 | -1 | 是 | 0.167 |

表中第一列为样本序号，第二列为样本的特征值，第三列为类标签（+1类和-1类），第四列为初始权重，权重值相等均为1/10=0.1，划分标准为x<=0.3第5列存储了预测类标y1。基于权重更新规则，最后一列存储了更新后的权值。

* 首先计算上述步骤中的权重错误率：



* 计算Gm(x)的相关系数：



* 得到相关系数后，根据公式计算更新权重向量



我们首先计算除了Zm的式子：

这里计算被正确分类的样本，预测类标向量y1和真实类标本身y相同，乘积符号为正，本身为正，则加一个负号之后第i个权重的值会被降低：



如果yi预测类标错误，则第i个权重将按照如下方式更新，对分错的样本权重将会增加：



* 完成上述w权重更新之后在除以Zm进行权重的归一化。

，这里面有7个分对的，3个分错的。

* 最后一步：

由于分正确的样本其对应权重在下一轮boosting中将从初始的0.1降低到0.666/0.914=0.072，分错的样本，其对应权重从0.1提升到0.153/0.914=0.167，权重得到提升。

### 1.8.5算法总结及推广到多分类

方法：加大前一轮分错样本权值，减小分对样本权值。而在下一轮迭代中，

总是选取让误差率最低的阈值来设计基本分类器，所以误差率e不断降低，在最终的加权投票时的基分类器的权重也相应的增加。

接下来讨论多维数据的单层决策树分类器构建方法

前面的例子：给定10个一维样本数据及类别。

1.选某个阈值进行分类（即单节点决策树）：阈值v的一边取+1，另一边取-1，并计算分类误差率；

2.多个不同阈值下分类后，选分类误差率最小的阈值作为本轮的基分类器。

推广到多维数据：

例子中的一维样本数据相当于多维数据中的某个特征向量。

对于多维训练数据，构建单节点决策树时，需要遍历每个特征，最终选择使分类误差率e最小的阈值作为最终分类决策函数。

## 1.9GBDT梯度提升决策树算法原理及实战

### 1.9.1提升树的原理

（1）提升方法采用加法模型(即基函数的线性组合)与前向分布算法。以决策树为基函数的提升方法提升方法称为提升树。Boosting tree，对分类问题，决策树是二叉决策树；对回归问题，决策树是二叉回归树。

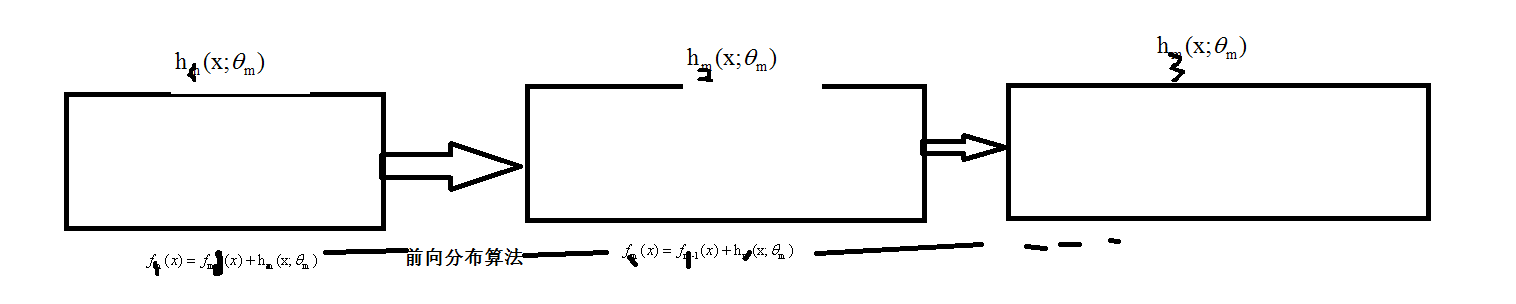
* 提升树模型可以表示为决策树(为基本分类器)的加法模型：



* 提升树算法采用前向分布算法。令为当前模型，则第m步模型为：



其中初始化提升树



通过损失函数最小化来确定下一棵决策树参数：



如果使用不同的损失函数，则得到两种不同的提升树算法（预测值为y1，真实值为y）：

（1）回归问题：通常使用平方误差损失函数：



（2）分类问题：通常使用指数损失函数：



* 最后得到损失函数



由于树的线性组合可以很好拟合训练数据，即使数据中的输入和输出之间的关系很复杂也是如此，所以提升树是一个高功能的学习算法。



### 1.9.2提升树算法

回归树算法:

输入：训练数据集

输出：提升树

算法步骤：

（1）初始化

（2）对于m=1,2,3......M

\*\*计算残差

\*\*拟合残差学习一颗回归树，得到。

\*\*更新

（3）得到提升树

### 1.9.3GBDT&GBRT

提升是一个机器学习技术，可以用于回归和分类问题，它每一步产生一个弱预测模型，如决策树，并加权累加到总模型中了如果每一步的弱预测模型生成都是依据损失函数的梯度方向，则称之为梯度提升。

梯度提升算法首先给定一个目标损失函数，它的定义域是所有可行的弱函数集合(基函数)；提升算法通过迭代的选择一个负梯度方向上的基函数来逐渐逼近局部极小值。

提升的理论意义：如果一个问题存在弱分类器，则可以通过提升的方式得到强分类器。

***在提升树的基础上，利用梯度下降法，求解目标函数或者损失函数最优解的情况，就称之为梯度提升算法。（理解）***

提升树中，如果损失函数是平方损失函数和指数损失函数时，由于这两种函数求导数跟简单，所以求解下面的参数最优解就比较简单：



如果损失函数是一般函数，该优化问题往往比较难求得，Freidman提出了梯度提升算法来解决该最优值求解问题----利用损失函数的负梯度（梯度下降法）在当前模型的值作为提升树算法中残差的近似值，拟合一颗决策树。在回归问题中，这称之为梯度提升回归树GBRT，在分类问题中，称之为梯度提升决策树GBDT。

**梯度提升算法步骤如下：**

输入：训练数据集

输出：回归树

算法步骤：

初始化：

1.初始化f0



对于m=1,2.....M，m指的是串行分类器的个数

对于i=1,2......N， i指的是求残差是yi和f(xi)中的i值

**2.这里计算的是损失负梯度在当前模型的值，将它作为残差的估计。**



B.对拟合一颗回归树，得到第m棵树的叶节点区域

C.计算在每个区域上的输出值，



其中c指的是上面公式中的h(x)决策树或回归树。

D.更新

，该决策树将x所属的叶节点的输出作为x的预测值。

得到回归树



上述梯度提升树算法，在回归问题中，这称之为梯度提升回归树GBRT，在分类问题中，称之为梯度提升决策树GBDT。

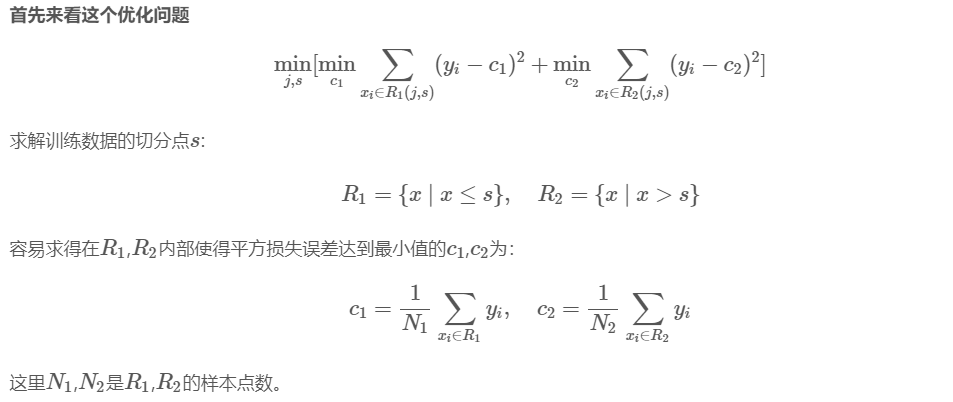
### 1.9.4GBDT案例

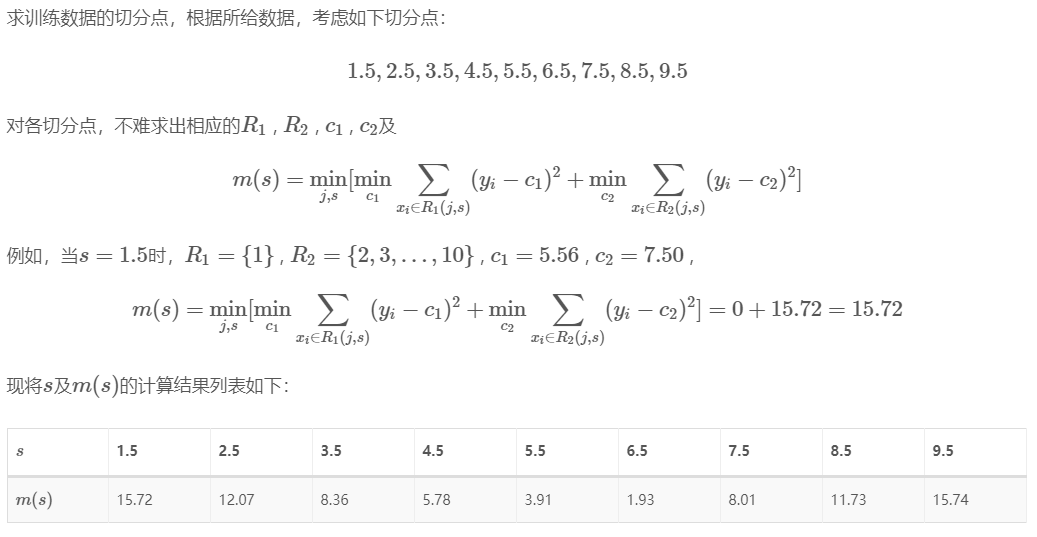
在Cart树基础上继续利用前向分布算法结合加法模型，构建提升决策树模型：

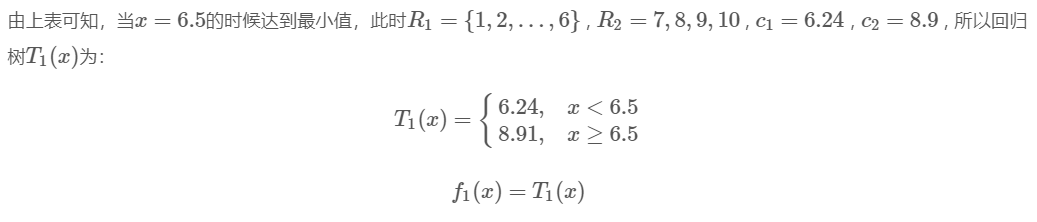
GBDT：获得一颗二叉树后，利用残差，再在完整的数据集上生成一颗二叉树，最终将多颗二叉树加权累加组成一个最终的函数。

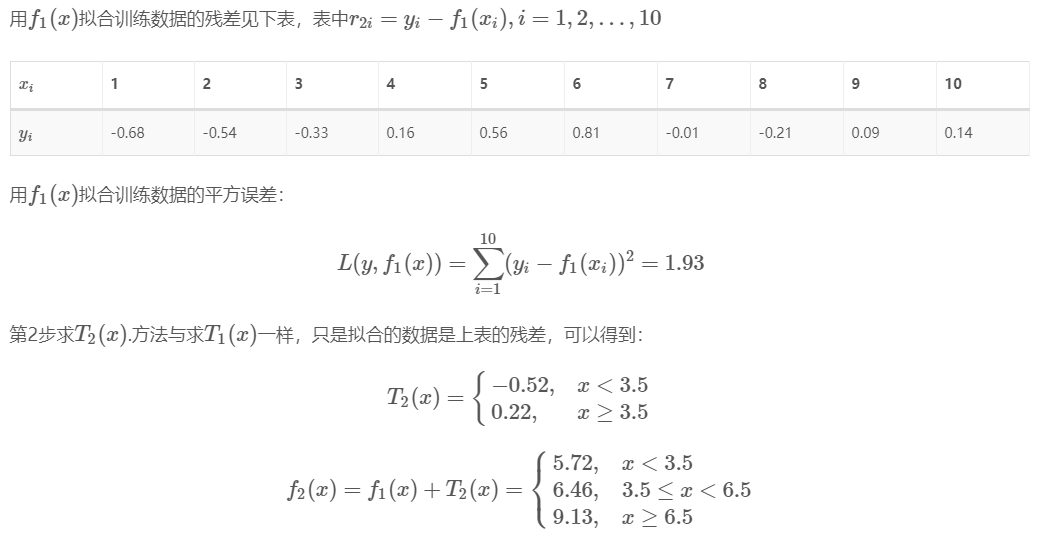
在回归树案例中的训练数据，x的取值范围为区间[0.5,10.5],y的取值范围为区间[5.0,10.0],学习这个回归问题的最小二叉回归树。

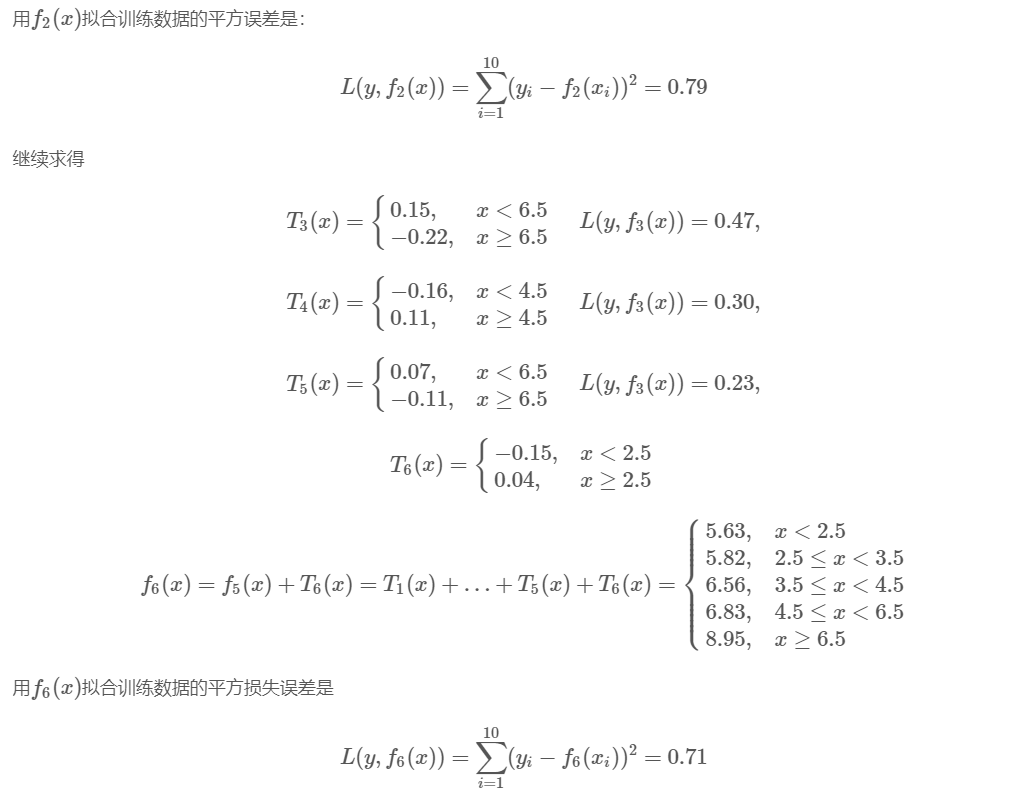
| **X** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| y | 5.56 | 5.70 | 5.91 | 6.40 | 6.80 | 7.05 | 8.9 | 8.70 | 9.00 | 9.05 |











假设此时已经满足误差要求，那么f(x)=f6(x)即为所求的回归树。

### 1.9.5集成算法多样性

集成学习中，个体学习器多样性越大越好。通常为了增大个体学习器的多样性，在学习过程中引入随机性。常用的方法包括：对数据样本进行扰动、对输入属性进行扰动、对算法参数进行扰动。

#### （1）数据样本扰动

给定数据集，可以使用采样法从中产生出不同的数据子集。然后在利用不同的数据子集训练出不同的个体学习器。

该方法简单有效，使用广泛。

1. 数据样本扰动对于“不稳定学习器”很有效。“不稳定学习器”是这样一类学习器：训练样本稍加变化就会导致学习器有显著的变动，如决策树和神经网络等。
2. 数据样本扰动对于“稳定学习器”无效。“稳定学习器”是这样一类学习器：学习器对于数据样本的扰动不敏感，如线性学习器、支持向量机、朴素贝叶斯、K近邻学习器等。

如Bagging算法就是利用Bootstrip抽样完成对数据样本的自助采样。

#### （2）输入属性的扰动

训练样本通常由一组属性描述，可以基于这些属性的不同组合产生不同的数据子集，然后在利用这些数据子集训练出不同的个体学习器。

1. 若数据包含了大量冗余的属性，则输入属性扰动效果较好。此时不仅训练出了多样性大的个体，还会因为属性数量的减少而大幅节省时间开销。同时由于冗余属性多，即使减少一些属性，训练个体学习器也不会很差。
2. 若数据值包含少量属性，则不宜采用输入属性扰动法。

#### （3）算法参数的扰动

通常可以通过随机设置不用的参数，比如对模型参数加入小范围的随机扰动，从而产生差别较大的个体学习器。

在使用交叉验证法（GridSearch网格搜索）来确定基学习器的参数时，实际上就是用不同的参数训练出来了多个学习器，然后从中挑选出效果最好的学习器。集成学习相当于将所有这些学习器利用起来了。

随机森林学习器就结合了数据样本的扰动及输入属性的扰动。

### 1.9.6SparkMllib的GBDT参数详解

[梯度提升树（GBT）](http://en.wikipedia.org/wiki/Gradient_boosting) 是[决策树的](http://spark.apache.org/docs/2.2.0/ml-classification-regression.html" \l "decision-trees)集合。GBDT迭代地训练决策树以最小化损失函数。该spark.ml实现支持使用连续和分类特征进行二分类和回归的GBT。

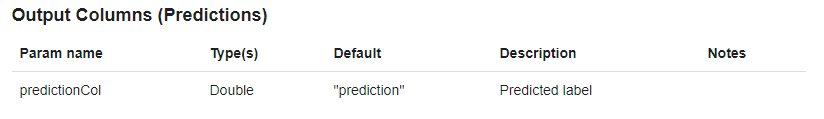
#### （1）输入和输出

我们在此列出输入和输出（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。

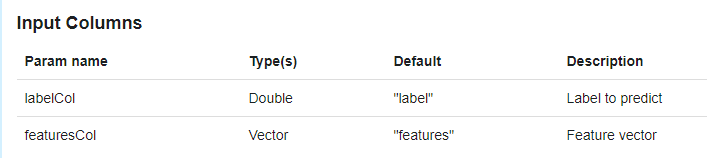
#### （2）输入输出列

请注意，GBTClassifier目前仅支持二进制标签。

**输出列（预测）**



**输入列**



将来，GBTClassifier还会输出rawPrediction和的列probability，就像那样RandomForestClassifier。

#### （3）SparkMllib中梯度树（GBT）详解

[梯度提升树（GBT）](http://en.wikipedia.org/wiki/Gradient_boosting) 是[决策树的](http://spark.apache.org/docs/2.2.0/mllib-decision-tree.html)集合。GBT迭代地训练决策树以最小化损失函数。与决策树一样，GBT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。

spark.mllib支持GBT用于二进制分类和回归，使用连续和分类功能。 spark.mllib使用现有的[决策树](http://spark.apache.org/docs/2.2.0/mllib-decision-tree.html)实现来实现GBT 。

注意：**GBT尚不支持多类分类**。对于多类问题，请使用 [决策树](http://spark.apache.org/docs/2.2.0/mllib-decision-tree.html)或[随机森林](http://spark.apache.org/docs/2.2.0/mllib-ensembles.html" \l "Random-Forest)。

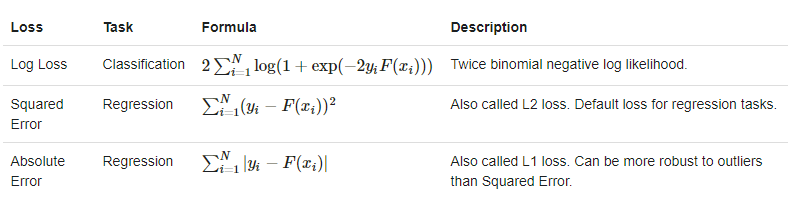
#### （4）基本算法

梯度增强迭代地训练一系列决策树。在每次迭代时，算法使用当前集合来预测每个训练实例的标签，然后将预测与真实标签进行比较。重新标记数据集以更加强调预测错误的训练实例。因此，在下一次迭代中，决策树将帮助纠正先前的错误。

重新标记实例的具体机制由损失函数定义（下面讨论）。每次迭代，GBT进一步减少训练数据的这种损失函数。

#### （5）损失

下表列出了GBT目前支持的损失。请注意，每种损失都适用于分类或回归，而不是两者。



#### （6）参数

通过讨论各种参数包括一些使用GBT的指南。这里省略了一些决策树参数，因为[决策树指南](http://spark.apache.org/docs/2.2.0/mllib-decision-tree.html)中对这些参数进行了介绍。

* loss：有关损失及其对任务的适用性（分类与回归）的信息，请参阅上面的部分。根据数据集的不同，不同的损失可能会产生明显不同的结果。
* numIterations：这设置了整体中树的数量。每次迭代都会生成一棵树。增加此数字可使模型更具表现力，从而提高训练数据的准确性。但是，如果测试时间过长，则测试时间精度可能会受到影响。
* learningRate：不需要调整此参数。如果算法行为看起来不稳定，则降低此值可以提高稳定性。
* algo：使用树[策略]参数设置算法或任务（分类与回归）。

#### （7）训练时验证

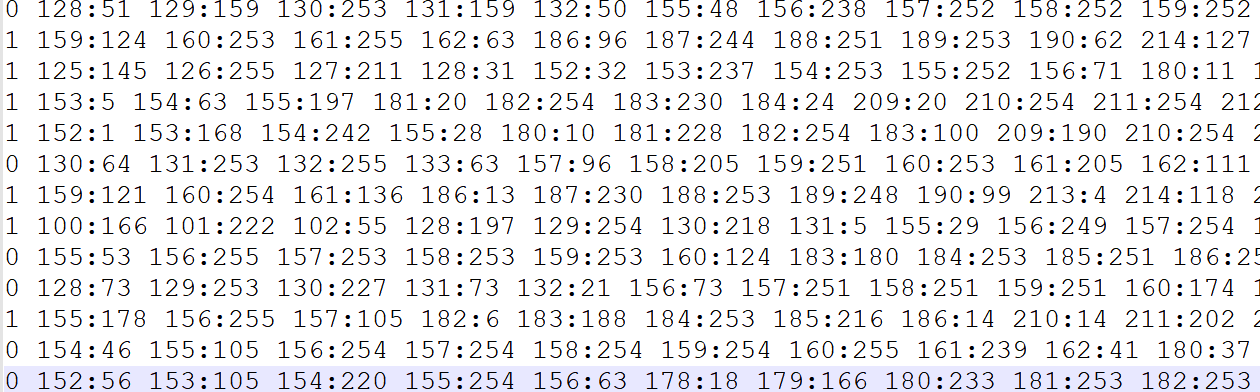
当训练更多树木时，梯度增强容易产生过度拟合。为了防止过度拟合，在训练时进行验证很有用。提供了runWithValidation方法以使用此选项。它需要一对RDD作为参数，第一个是训练数据集，第二个是验证数据集。

当验证错误的改进不超过某个阈值（由validationTol参数提供BoostingStrategy）时，停止训练。实际上，验证错误最初会降低，之后会增加。可能存在验证错误不会单调变化的情况，建议用户设置足够大的阈值，并使用evaluateEachIteration （每次迭代产生错误或损失）检查**验证曲线**以调整迭代次数。

### 1.9.7GBDT建模实战

#### （1）SparkMl代码实战

**数据：**



**代码：**

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.classification.{GBTClassificationModel, GBTClassifier}

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

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

// Load and parse the data file, converting it to 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.

// Set maxCategories so features with > 4 distinct values are treated as continuous.

val featureIndexer = new VectorIndexer()

.setInputCol("features")

.setOutputCol("indexedFeatures")

.setMaxCategories(4)

.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 GBT model.

val gbt = new GBTClassifier()

.setLabelCol("indexedLabel")

.setFeaturesCol("indexedFeatures")

.setMaxIter(10)

// Convert indexed labels back to original labels.

val labelConverter = new IndexToString()

.setInputCol("prediction")

.setOutputCol("predictedLabel")

.setLabels(labelIndexer.labels)

// Chain indexers and GBT in a Pipeline.

val pipeline = new Pipeline()

.setStages(Array(labelIndexer, featureIndexer, gbt, 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)

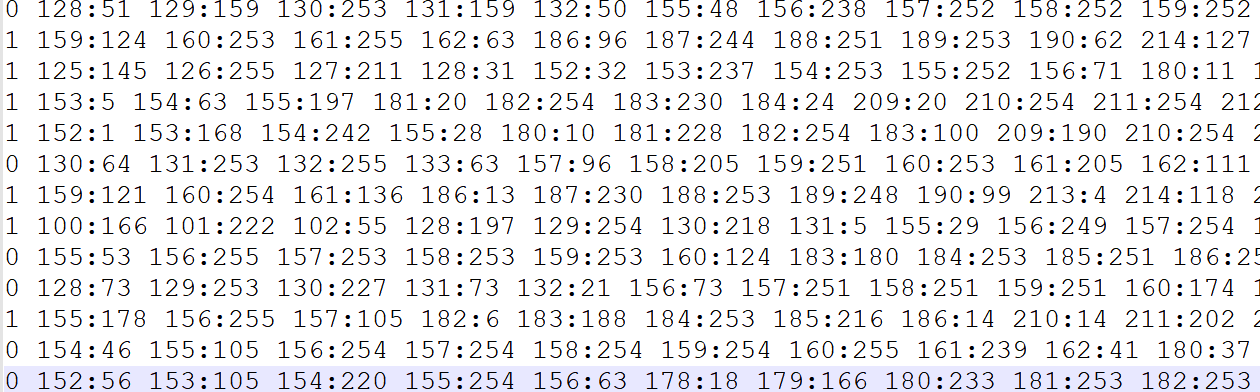
println("Test Error = " + (1.0 - accuracy))

val gbtModel = model.stages(2).asInstanceOf[GBTClassificationModel]

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

#### （2）SparkMlllib代码实战

**数据：**



**代码：**

import org.apache.spark.mllib.tree.GradientBoostedTrees

import org.apache.spark.mllib.tree.configuration.BoostingStrategy

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

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 GradientBoostedTrees model.

// The defaultParams for Classification use LogLoss by default.

val boostingStrategy = BoostingStrategy.defaultParams("Classification")

boostingStrategy.numIterations = 3 // Note: Use more iterations in practice.

boostingStrategy.treeStrategy.numClasses = 2

boostingStrategy.treeStrategy.maxDepth = 5

// Empty categoricalFeaturesInfo indicates all features are continuous.

boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int, Int]()

val model = GradientBoostedTrees.train(trainingData, boostingStrategy)

// 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 GBT model:\n" + model.toDebugString)

// Save and load model

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

val sameModel = GradientBoostedTreesModel.load(sc,

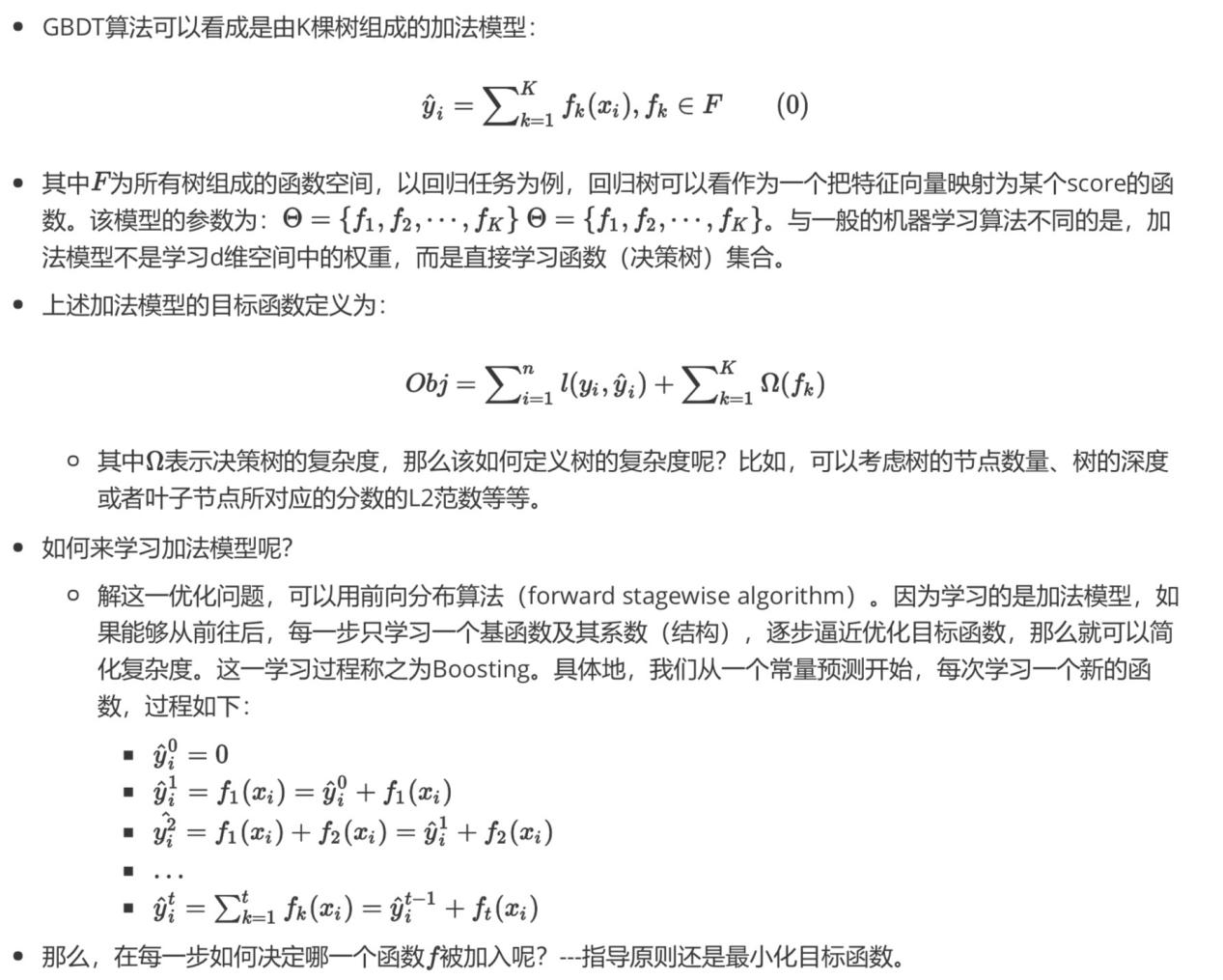
"target/tmp/myGradientBoostingClassificationModel")

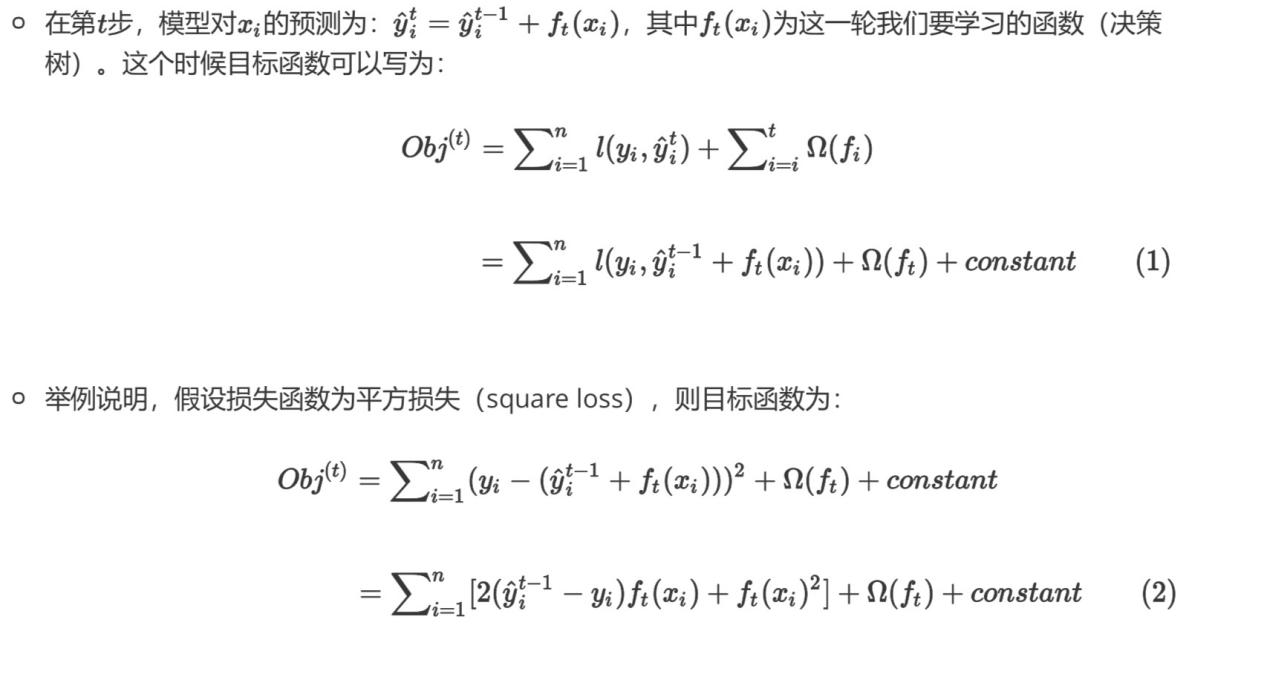
## 1.10XGBOOST极限梯度提升树原理及实战

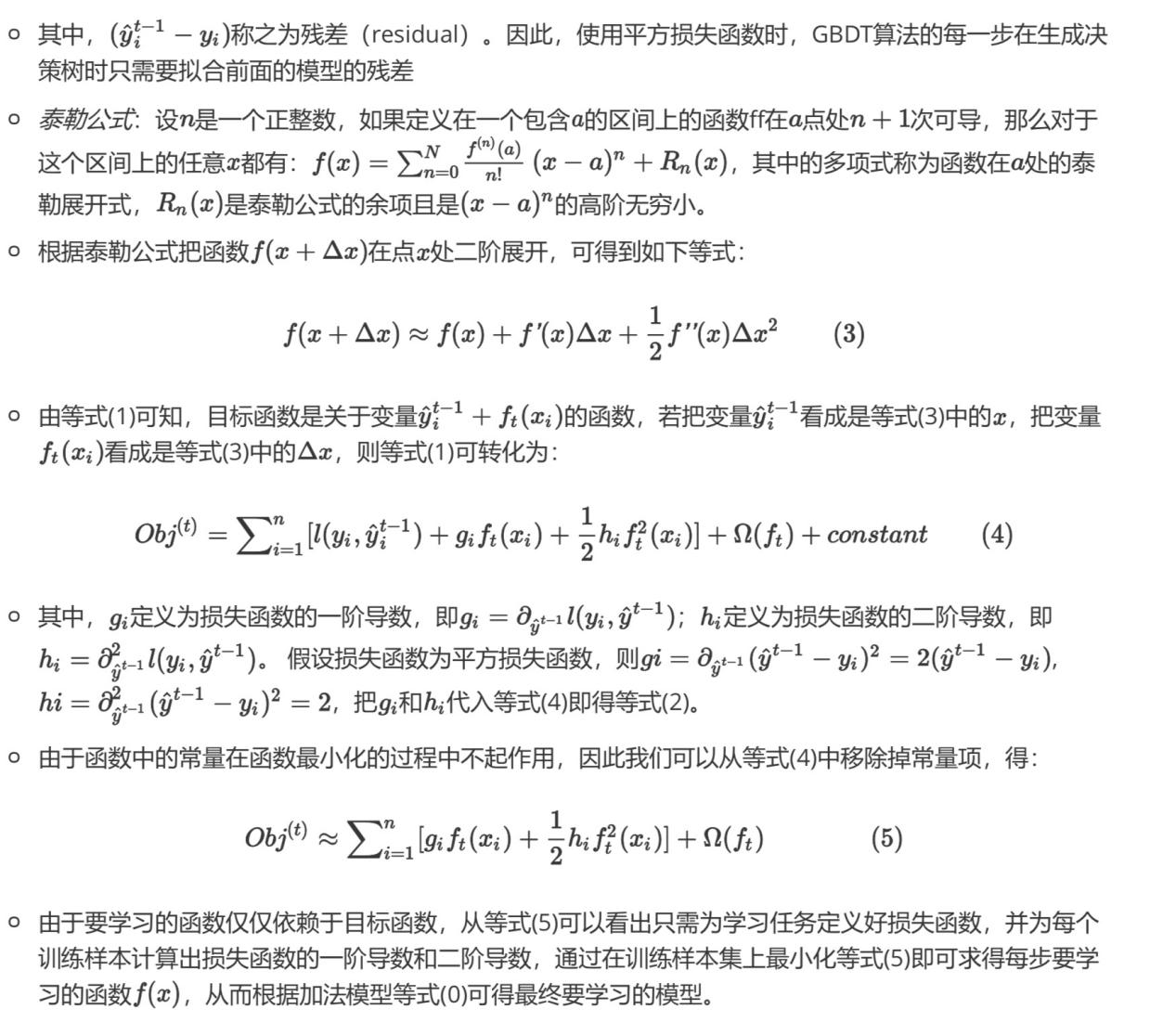
该部分详细步骤参考XGBOOST原理笔记与推导，该算法是互联网公司常用算法模型，务必掌握。

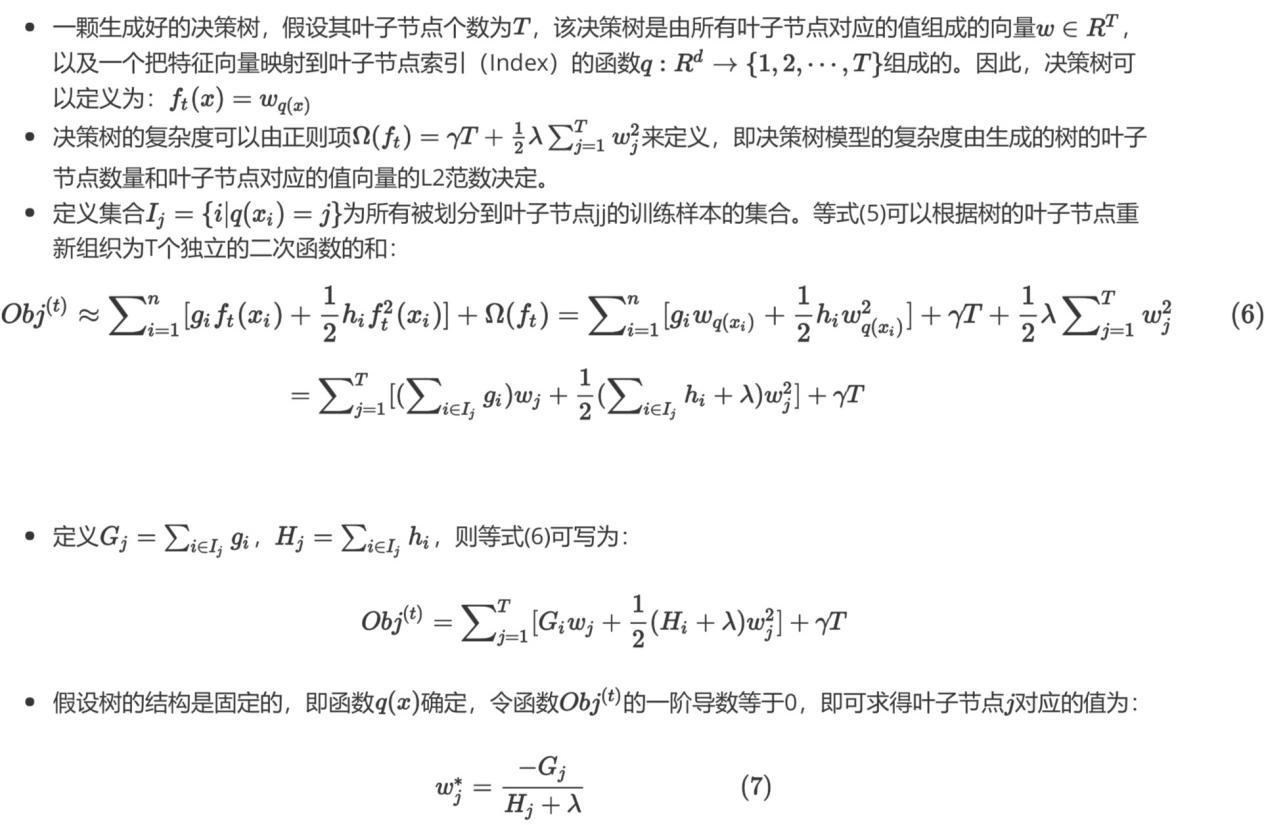
官方doc: https://xgboost.readthedocs.io/en/latest/tutorials/model.html

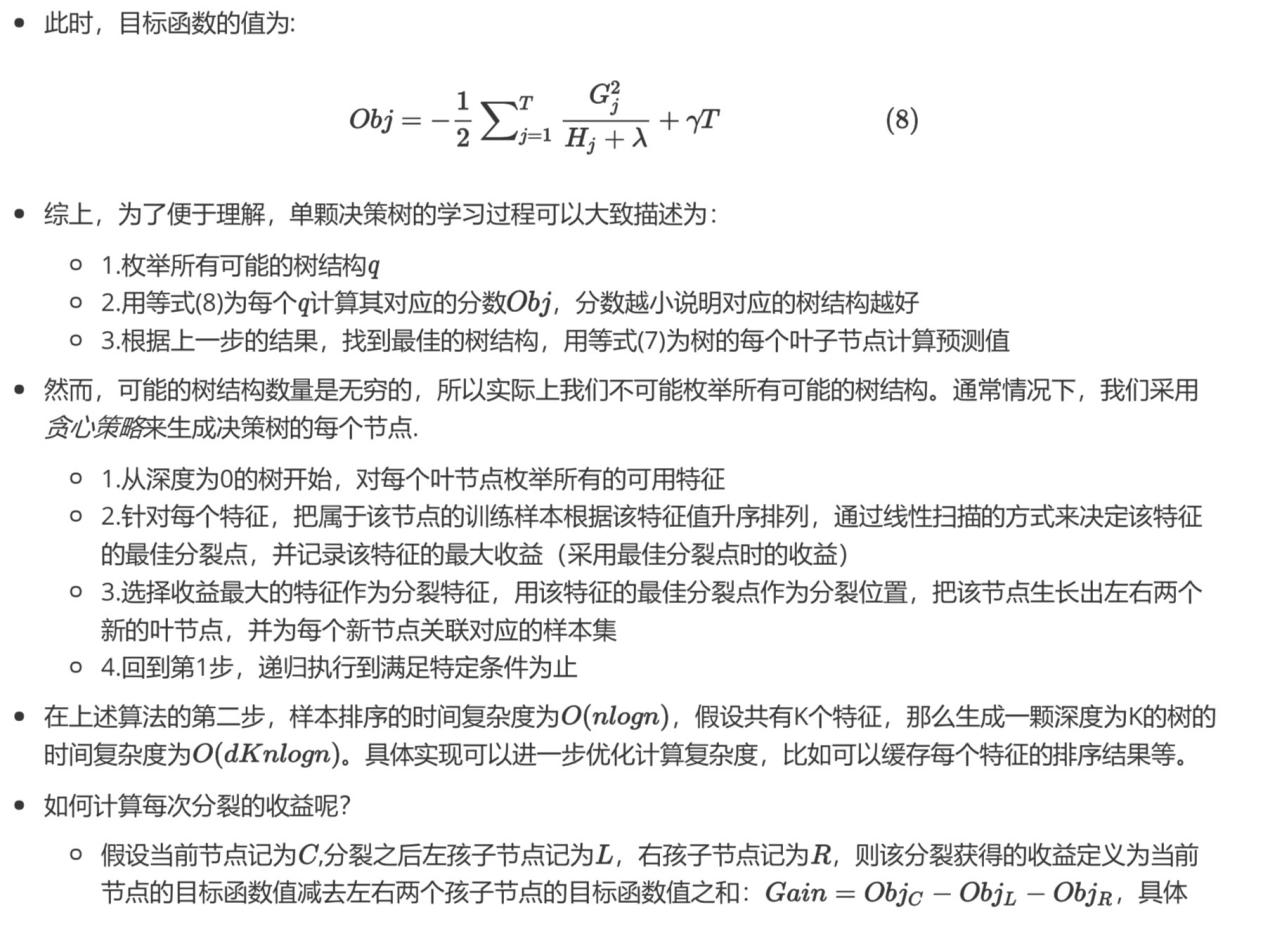
### 1.10.1XGBOOST原理

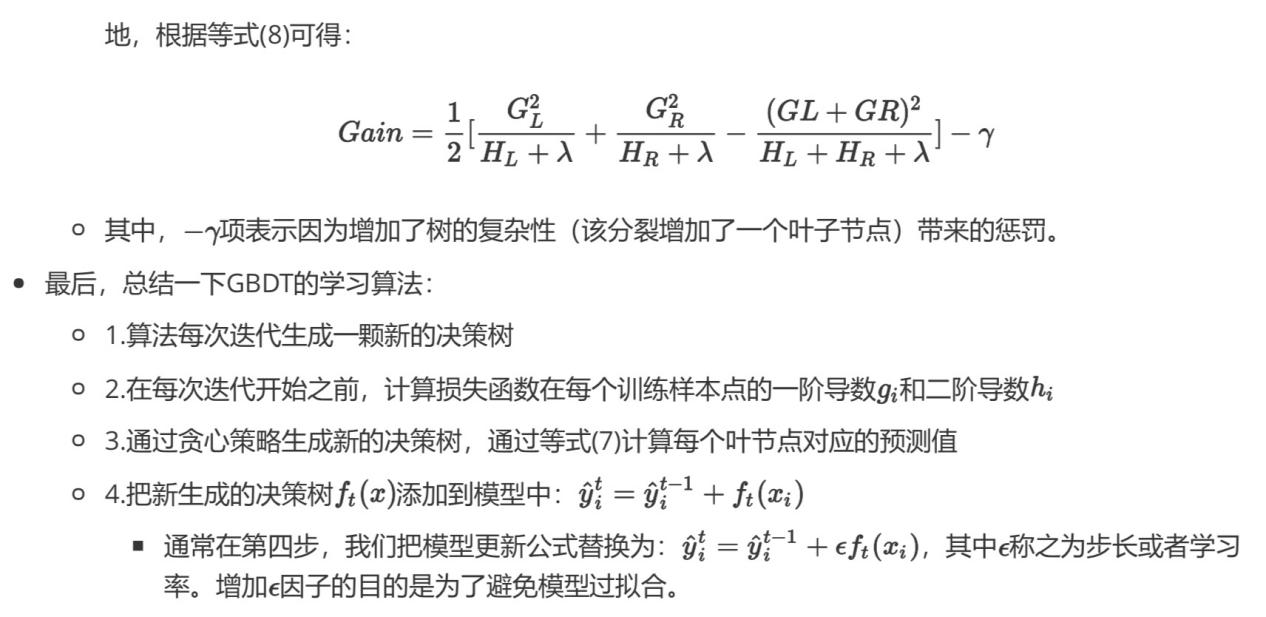






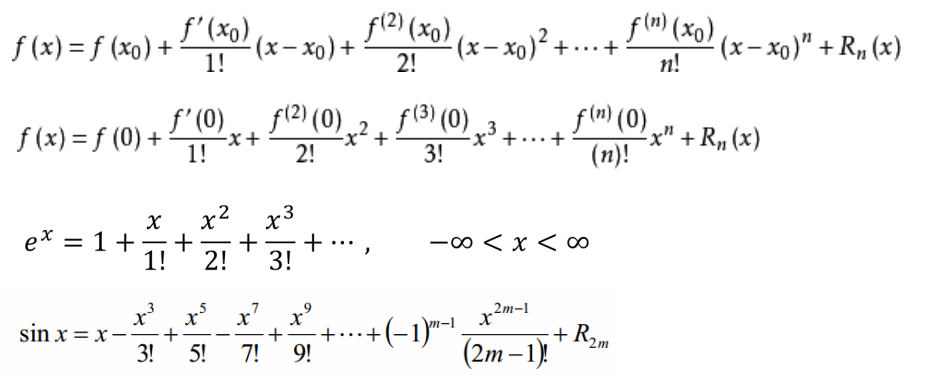




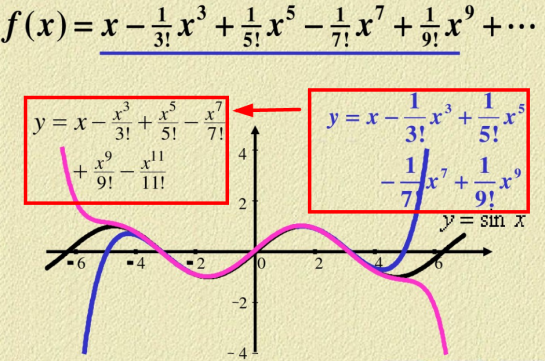
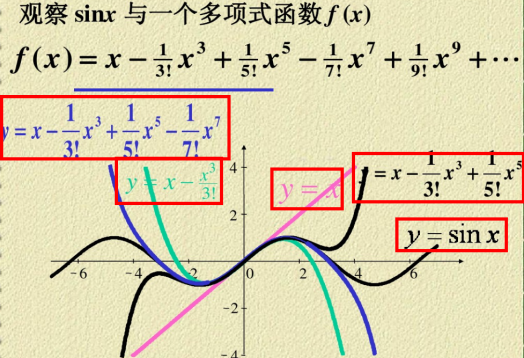


【泰勒公式补充】

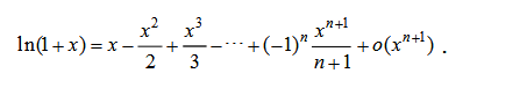
泰勒公式以及泰勒公式在x=0处的展开：



观察泰勒公式近似求解-5阶近似及右图的9阶近似

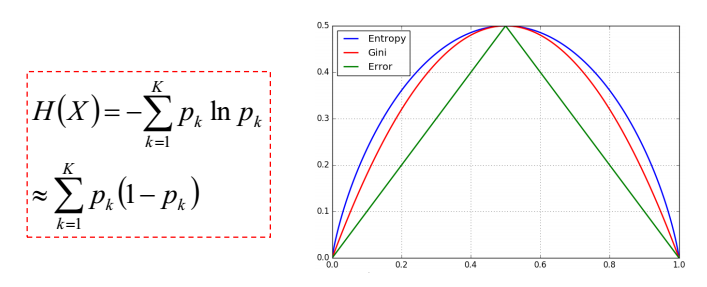


接下来分析ln(1+x)的泰勒展开式



在决策树算法中的Gini系数即为通过泰勒的一阶展开实现Gini系数公式的求解：

将f(x)=-ln(x)在x=1处一阶展开，忽略高阶无穷小，得到f(x)=1-x（约等于）



### 1.10.2XGBOOST4J(XGBOOST+Spark结合) XGBoost4J-Spark是一个旨在通过将XGBoost安装到Apache Spark的MLLIB框架来无缝集成XGBoost和Apache Spark的项目。通过集成用户不仅可以使用XGBoost的高性能算法实现，还可以利用Spark强大的数据处理引擎：

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

这里使用XGBoost4J-Spark构建机器学习管道的端到端流程。

* 使用Spark预处理数据以适应XGBoost / XGBoost4J-Spark的数据接口
* 使用XGBoost4J-Spark训练XGBoost模型
* 使用Spark提供XGBoost模型（预测）
* 使用XGBoost4J-Spark构建机器学习管道
* 在生产环境中运行XGBoost4J-Spark

在进入如何使用XGBoost4J-Spark之前，我们将简要介绍如何使用XGBoost4J-Spark构建机器学习应用程序。我们需要引用Maven Central中的依赖项。

您可以在您的添加中添加以下依赖项pom.xml。

<dependency>

<groupId>ml.dmlc</groupId>

<artifactId>xgboost4j-spark</artifactId>

<version>0.8</version>

</dependency>

默认情况下，使用dmlc-core中的跟踪器来驱动XGBoost4J-Spark的训练。它需要Python 2.7+。我们还有一个实验性的Scala版本跟踪器，可以通过传递参数启用tracker\_conf。

### 1.10.3数据准备

如前所述，XGBoost4J-Spark无缝集成了Spark和XGBoost。通过集成，用户可以使用方便，强大的数据处理框架Spark在训练/测试数据集上应用各种类型的转换。

这里使用Iris数据集作为示例来展示如何使用Spark来转换原始数据集并使其适合XGBoost的数据接口。

Iris数据集以CSV格式提供。每个实例包含4个特征，“萼片长度”，“萼片宽度”，“花瓣长度”和“花瓣宽度”。此外，它包含“class”列，它基本上是具有三个可能值的标签：“Iris Setosa”，“Iris Versicolour”和“Iris Virginica”。

使用Spark的内置读取器读取数据集

数据转换的第一件事是将数据集加载为Spark的结构化数据抽象DataFrame。

import org.apache.spark.sql.SparkSession

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

val spark = SparkSession.builder().getOrCreate()

val schema = new StructType(Array(

StructField("sepal length", DoubleType, true),

StructField("sepal width", DoubleType, true),

StructField("petal length", DoubleType, true),

StructField("petal width", DoubleType, true),

StructField("class", StringType, true)))

val rawInput = spark.read.schema(schema).csv("input\_path")

在第一行，我们创建了一个SparkSession实例，它是使用DataFrame的任何Spark程序的入口。该schema变量定义了包装Iris数据的DataFrame的模式。使用这个显式设置的模式，我们可以定义列的名称及其类型; 否则列名称将是Spark派生的默认名称，例如\_col0等等。最后，我们可以使用Spark的内置csv读取器将Iris csv文件作为名为DataFrame的文件加载rawInput。

注意：Spark还包含许多其他格式的内置阅读器。最新版本的Spark支持CSV，JSON，Parquet和LIBSVM。

转换原始iris数据集，为了使Iris数据集能够被XGBoost识别，我们需要

* + 1.将String-typed标签（即“class”）转换为Double-typed标签。
  + 2.将特征列组合为矢量以适合Spark ML框架的数据接口。

要将String-typed标签转换为Double，我们可以使用Spark的内置功能转换器StringIndexer。

import org.apache.spark.ml.feature.StringIndexer

val stringIndexer = new StringIndexer().

setInputCol("class").

setOutputCol("classIndex").

fit(rawInput)

val labelTransformed = stringIndexer.transform(rawInput).drop("class")

使用新创建的StringIndexer实例：

* 1.我们设置输入列，即包含String类型标签的列
* 2.我们设置输出列，即列包含Double-typed标签。
* 3.然后我们fit使用输入DataFrame进行StringIndex的rawInput，这样Spark内部可以获得不同值的总数等信息。

现在我们有一个StringIndexer，可以应用于我们的输入DataFrame。要执行StringIndexer的转换逻辑，我们transform输入DataFrame的rawInput并保持简洁的DataFrame，我们删除列“class”，只保留特征列和转换的Double-typed标签列（在上面代码片段的最后一行）。

该fit和transform在MLLIB两个关键操作。基本上，fit生成一个“转换器”，例如StringIndexer，并且每个变换器transform在DataFrame上应用方法以添加包含变换特征/标签或预测结果等的新列。

类似地，我们可以使用另一个变换器VectorAssembler来组合特征列“萼片长度”，“萼片宽度”，“花瓣长度”和“花瓣宽度”作为矢量。

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

val vectorAssembler = new VectorAssembler().

setInputCols(Array("sepal length", "sepal width", "petal length", "petal width")).

setOutputCol("features")

val xgbInput = vectorAssembler.transform(labelTransformed).select("features", "classIndex")

现在，我们有一个只包含两列的DataFrame，“features”包含向量表示的“sepal length”，“sepal width”，“petal length”和“petal width”以及“classIndex”，它们具有Double-typed标签。像这样的DataFrame（包含向量表示的功能和数字标签）可以直接提供给XGBoost4J-Spark的训练引擎。

### 1.10.4模型训练

XGBoost支持回归和分类。虽然我们在本教程中使用Iris数据集来展示我们如何使用XGBoost / XGBoost4J-Spark来解决多类别分类问题，但回归中的用法与分类非常相似。

要训练XGBoost模型进行分类，我们需要首先声明XGBoostClassifier：

import ml.dmlc.xgboost4j.scala.spark.XGBoostClassifier

val xgbParam = Map("eta" -> 0.1f,

"max\_depth" -> 2,

"objective" -> "multi:softprob",

"num\_class" -> 3,

"num\_round" -> 100,

"num\_workers" -> 2)

val xgbClassifier = new XGBoostClassifier(xgbParam).

setFeaturesCol("features").

setLabelCol("classIndex")

可以在此处找到用于训练XGBoost模型的可用参数。在XGBoost4J-Spark中，我们不仅支持默认的参数集，还支持这些参数的驼峰式CAMEL变体，以保持与Spark的MLLIB参数一致。

具体来说，此页面中的每个参数都在XGBoost4J-Spark中使用驼峰大小写的等效形式。例如，要max\_depth为每个树设置，您可以传递参数，就像我们在上面的代码片段中所做的那样（max\_depth包含在Map中），或者您可以通过XGBoostClassifer中的setter来完成：

val xgbClassifier = new XGBoostClassifier().

setFeaturesCol("features").

setLabelCol("classIndex")

xgbClassifier.setMaxDepth(2)

在我们设置XGBoostClassifier参数和feature / label列之后，我们可以通过使用输入DataFrame拟合XGBoostClassifier来构建变换器XGBoostClassificationModel。该fit操作基本上是训练过程，然后可以将生成的模型用于预测。

val xgbClassificationModel = xgbClassifier.fit(xgbInput)

### 1.10.5earlystop

早期停止是防止不必要的训练迭代的特征。通过指定num\_early\_stopping\_rounds或直接调用setNumEarlyStoppingRoundsXGBoostClassifier或XGBoostRegressor，如果评估指标远离最佳迭代和早期停止训练迭代，我们可以定义轮数。

除此之外num\_early\_stopping\_rounds，您还需要定义maximize\_evaluation\_metrics或调用setMaximizeEvaluationMetrics以指定是否要最大化或最小化培训中的度量标准。

例如，我们需要最大化评估指标（设置maximize\_evaluation\_metrics为true），并设置num\_early\_stopping\_rounds为5.第10次迭代的评估指标是迄今为止的最大值。在以下迭代中，如果没有大于第10次迭代的评估度量（最佳值），则训练将在第15次迭代时提前停止。

### 1.10.6使用评估集进行训练

您还可以在使用多个评估数据集进行培训期间监控模型的性能。通过指定eval\_sets或调用setEvalSetsXGBoostClassifier或XGBoostRegressor，您可以传入多个评估数据集，这些数据集的类型为从String到DataFrame的Map。

### 1.10.7模型预测

XGBoost4j-Spark支持两种模型服务方式：批量预测和单实例预测。

#### （1）批量预测

当我们得到一个模型，XGBoostClassificationModel或XGBoostRegressionModel时，它需要一个DataFrame，读取包含特征向量的列，预测每个特征向量，并默认输出带有以下列的新DataFrame：

* XGBoostClassificationModel将为每个可能的标签输出marginins（rawPredictionCol），probabilities（probabilityCol）和最终预测标签（predictionCol）。
* XGBoostRegressionModel将输出预测标签（predictionCol）。

批量预测期望用户以DataFrame的形式传递测试集。XGBoost4J-Spark为DataFrame的每个分区启动XGBoost工作程序以进行并行预测，并批量生成整个DataFrame的预测结果。

val xgbClassificationModel = xgbClassifier.fit(xgbInput)

val results = xgbClassificationModel.transform(testSet)

使用上面的代码片段，我们得到一个结果DataFrame，结果包含边距，每个类的概率以及每个实例的预测

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

| features|classIndex| rawPrediction| probability|prediction|

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

|[5.1,3.5,1.4,0.2]| 0.0|[3.45569849014282...|[0.99579632282257...| 0.0|

|[4.9,3.0,1.4,0.2]| 0.0|[3.45569849014282...|[0.99618089199066...| 0.0|

|[4.7,3.2,1.3,0.2]| 0.0|[3.45569849014282...|[0.99643349647521...| 0.0|

|[4.6,3.1,1.5,0.2]| 0.0|[3.45569849014282...|[0.99636095762252...| 0.0|

|[5.0,3.6,1.4,0.2]| 0.0|[3.45569849014282...|[0.99579632282257...| 0.0|

|[5.4,3.9,1.7,0.4]| 0.0|[3.45569849014282...|[0.99428516626358...| 0.0|

|[4.6,3.4,1.4,0.3]| 0.0|[3.45569849014282...|[0.99643349647521...| 0.0|

|[5.0,3.4,1.5,0.2]| 0.0|[3.45569849014282...|[0.99579632282257...| 0.0|

|[4.4,2.9,1.4,0.2]| 0.0|[3.45569849014282...|[0.99618089199066...| 0.0|

|[4.9,3.1,1.5,0.1]| 0.0|[3.45569849014282...|[0.99636095762252...| 0.0|

|[5.4,3.7,1.5,0.2]| 0.0|[3.45569849014282...|[0.99428516626358...| 0.0|

|[4.8,3.4,1.6,0.2]| 0.0|[3.45569849014282...|[0.99643349647521...| 0.0|

|[4.8,3.0,1.4,0.1]| 0.0|[3.45569849014282...|[0.99618089199066...| 0.0|

|[4.3,3.0,1.1,0.1]| 0.0|[3.45569849014282...|[0.99618089199066...| 0.0|

|[5.8,4.0,1.2,0.2]| 0.0|[3.45569849014282...|[0.97809928655624...| 0.0|

|[5.7,4.4,1.5,0.4]| 0.0|[3.45569849014282...|[0.97809928655624...| 0.0|

|[5.4,3.9,1.3,0.4]| 0.0|[3.45569849014282...|[0.99428516626358...| 0.0|

|[5.1,3.5,1.4,0.3]| 0.0|[3.45569849014282...|[0.99579632282257...| 0.0|

|[5.7,3.8,1.7,0.3]| 0.0|[3.45569849014282...|[0.97809928655624...| 0.0|

|[5.1,3.8,1.5,0.3]| 0.0|[3.45569849014282...|[0.99579632282257...| 0.0|

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

#### （2）单实例预测

XGBoostClassificationModel或XGBoostRegressionModel支持对单个实例进行预测。它接受单个Vector作为特征，并输出预测标签。

但是，由于XGBoost的内部开销，单实例预测的开销很高，请谨慎使用！

val features = xgbInput.head().getAs[Vector]("features")

val result = xgbClassificationModel.predict(features)

#### （3）模型持久性

模型和管道持久性

数据科学家生成ML模型并将其交给工程团队，以便在生产环境中进行部署。相反，数据科学家可以使用经过训练的模型，例如作为基线，在整个数据探索过程中。因此，支持模型持久性以使模型在使用场景和编程语言中可用是非常重要的。

XGBoost4j-Spark支持保存和加载XGBoostClassifier / XGBoostClassificationModel和XGBoostRegressor / XGBoostRegressionModel。它还支持保存和加载包含这些估算器和模型的ML管道。

我们可以将XGBoostClassificationModel保存到文件系统：

val xgbClassificationModelPath = "/tmp/xgbClassificationModel"

xgbClassificationModel.write.overwrite().save(xgbClassificationModelPath)

然后在另一个会话中加载模型：

import ml.dmlc.xgboost4j.scala.spark.XGBoostClassificationModel

val xgbClassificationModel2 = XGBoostClassificationModel.load(xgbClassificationModelPath)

xgbClassificationModel2.transform(xgbInput)

接下来阐述ML管道保存和加载。

#### （4）与XGBoost的其他绑定交互

在使用XGBoost4j-Spark在大规模数据集上训练模型之后，有时我们希望在单个机器中进行模型服务或将其与其他单个节点库集成以进行进一步处理。XGBoost4j-Spark通过以下方式支持导出模型到本地：

val nativeModelPath = "/tmp/nativeModel"

xgbClassificationModel.nativeBooster.saveModel(nativeModelPath)

然后我们可以用单节点Python XGBoost加载这个模型：

import xgboost as xgb

bst = xgb.Booster({'nthread': 4})

bst.load\_model(nativeModelPath)

注意：

使用HDFS和S3导出带有nativeBooster.saveModel()的模型

当与其他语言绑定交互时，XGBoost还支持从本地文件系统以外的文件系统中保存模型和加载模型。你可以用前缀路径中使用HDFS和S3 hdfs://和s3://分别。但是，对于此功能，您必须执行以下操作之一：

使用此处描述的步骤构建XGBoost4J-Spark ，但打开USE\_HDFS（或USE\_S3等在同一位置）。使用此方法，您可以通过将“nativeModelPath”替换为HDFS路径来重用上述代码示例。

但是，如果使用USE\_HDFS等进行构建，则必须确保将所涉及的共享对象文件（例如libhdfs.so）放入群集的LIBRARY\_PATH中。要避免复杂的群集环境配置，请选择其他选项。

使用HDFS，S3等的绑定来传递模型文件。以下是步骤（以HDFS为例）：

创建一个新文件：

val outputStream = fs.create("hdfs\_path")

其中“fs”是Hadoop 中org.apache.hadoop.fs.FileSystem类的一个实例。

将返回的OutputStream在第一步中传递给nativeBooster.saveModel()：

xgbClassificationModel.nativeBooster.saveModel(outputStream)

从HDFS下载其他语言的文件，并使用预先构建的（不需要libhdfs.so）版本的XGBoost加载。（函数“download\_from\_hdfs”是由用户实现的辅助函数）

import xgboost as xgb

bst = xgb.Booster({'nthread': 4})

local\_path = download\_from\_hdfs("hdfs\_path")

bst.load\_model(local\_path)

**注意：**

XGBoost4J-Spark与其他绑定之间的一致性问题

XGBoost4J-Spark与XGBoost的其他语言绑定之间存在一致性问题。

当用户使用Spark以LIBSVM格式加载训练/测试数据时，使用以下代码片段：

spark.read.format("libsvm").load("trainingset\_libsvm")

Spark假设数据集使用的是基于1的索引（特征索引用1表示）。但是，在使用XGBoost的其他绑定进行预测时（例如XGBoost的Python API），XGBoost假定数据集默认使用基于0的索引（特征索引从0开始）。它为使用Spark训练模型但在XGBoost的其他绑定中以相同格式预测数据集的用户造成了陷阱。解决方案是在使用例如Python API预测之前将数据集转换为基于0的索引。indexing\_mode=1在使用DMatirx加载时附加到文件路径。例如在Python中：

xgb.DMatrix('test.libsvm?indexing\_mode=1')

### 1.10.8使用XGBoost4J-Spark构建ML管道

#### （1）基本ML管道

Spark ML管道可以将多个算法或函数组合到一个管道中。它涵盖了从特征提取，转换，选择到模型训练和预测。XGBoost4j-Spark可以无缝地将XGBoost嵌入到这样的管道中。以下示例显示如何构建由Spark MLlib特征转换器和XGBoostClassifier估计器组成的此类管道。

这里使用Iris数据集和rawInputDataFrame。首先，我们需要将数据集拆分为训练和测试数据集。

val Array(training, test) = rawInput.randomSplit(Array(0.8, 0.2), 123)

在建立了ML管道，包括4个阶段：

* 将所有要素组合到单个向量列中。
* 从字符串标签到索引双标签。
* 使用XGBoostClassifier训练分类模型。
* 将索引的双标签转换回原始字符串标签。

我们已经在前面的部分中展示了前三个步骤，最后一步是使用新的变换器IndexToString完成的：

val labelConverter = new IndexToString()

.setInputCol("prediction")

.setOutputCol("realLabel")

.setLabels(stringIndexer.labels)

我们需要将这些步骤组织为Spark ML框架中的管道，并评估整个管道以获取PipelineModel：

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

import org.apache.spark.ml.Pipeline

val pipeline = new Pipeline()

.setStages(Array(assembler, stringIndexer, booster, labelConverter))

val model = pipeline.fit(training)

在我们获得PipelineModel之后，我们可以对测试数据集进行预测并评估模型的准确性。

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

val prediction = model.transform(test)

val evaluator = new MulticlassClassificationEvaluator()

val accuracy = evaluator.evaluate(prediction)

#### （2）具有超参数调整的pipeline

最大化XGBoost功能的最关键操作是为模型选择最佳参数。手动调整参数是一个繁琐且耗费人力的过程。使用最新版本的XGBoost4J-Spark，我们可以利用Spark模型选择工具自动完成此过程。

接下来了使用CrossValidation和MulticlassClassificationEvaluator搜索两个XGBoost参数的最佳组合的代码片段，max\_depth以及eta。（请参阅XGBoost参数。）选择产生MulticlassClassificationEvaluator定义的最大精度的模型，并使用该模型生成测试集的预测。

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

import org.apache.spark.ml.PipelineModel

import ml.dmlc.xgboost4j.scala.spark.XGBoostClassificationModel

val paramGrid = new ParamGridBuilder()

.addGrid(booster.maxDepth, Array(3, 8))

.addGrid(booster.eta, Array(0.2, 0.6))

.build()

val cv = new CrossValidator()

.setEstimator(pipeline)

.setEvaluator(evaluator)

.setEstimatorParamMaps(paramGrid)

.setNumFolds(3)

val cvModel = cv.fit(training)

val bestModel = cvModel.bestModel.asInstanceOf[PipelineModel].stages(2)

.asInstanceOf[XGBoostClassificationModel]

bestModel.extractParamMap()

#### （3）运行XGBoost4J-Spark在生产

XGBoost4J-Spark是使XGBoost更容易进入生产环境的最重要步骤之一。这里了解在生产中运行XGBoost4J-Spark的三个关键功能。

**并行/分布训练**

大规模的训练数据集是生产环境中最重要的特征之一。为了确保XGBoost中的训练与数据大小成比例，XGBoost4J-Spark桥接了Spark的分布式/并行处理框架和XGBoost的并行/分布式训练机制。

在XGBoost4J-Spark中，每个XGBoost工作程序都由Spark任务包装，Spark的内存空间中的训练数据集以透明的方式提供给XGBoost工作者。

在我们构建XGBoostClassifier的代码片段中，我们设置参数num\_workers（或numWorkers）。此参数控制在训练XGBoostClassificationModel时我们希望拥有多少并行工作程序。

**注意**

**关于OpenMP优化：**

默认情况下，我们为每个XGBoost工作者分配一个核心。因此，每个XGBoost工作程序中的OpenMP优化都不会生效，并且通过同时运行多个工作程序（即Spark任务）来实现训练的并行化。

如果您确实需要OpenMP优化，则必须这样做nthread在创建XGBoostClassifier / XGBoostRegressor时设置为大于1的值spark.task.cpus在Spark中设置与之相同的值nthread

#### （4）关键点补充

XGBoost使用AllReduce。算法在训练期间同步所有的统计数据，例如直方图值。因此，XGBoost4J-Spark要求在训练运行之前所有核心都可用。nthread \* numWorkers

在许多用户共享同一群集的生产环境中，很难保证您的XGBoost4J-Spark应用程序可以为每次运行获取所有请求的资源。默认情况下，XGBoost中的通信层将在需要更多资源时阻止整个应用程序。这个过程通常会带来不必要的资源浪费，因为它可以保留现成资源并尝试提出更多要求 此外，这通常是默默地发生，并没有引起用户的注意。

XGBoost4J-Spark允许用户设置超时阈值，以便从群集中声明资源。如果应用程序在此时间段内无法获得足够的资源，则应用程序将失败而不是浪费资源来挂起。要启用此功能，您可以使用XGBoostClassifier / XGBoostRegressor进行设置：

xgbClassifier.setTimeoutRequestWorkers(60000L)

或通过在timeout\_request\_workers在xgbParamMap建XGBoostClassifier时：

val xgbParam = Map("eta" -> 0.1f,

"max\_depth" -> 2,

"objective" -> "multi:softprob",

"num\_class" -> 3,

"num\_round" -> 100,

"num\_workers" -> 2,

"timeout\_request\_workers" -> 60000L)

val xgbClassifier = new XGBoostClassifier(xgbParam).

setFeaturesCol("features").

setLabelCol("classIndex")

如果XGBoost4J-Spark无法获得足够的资源来运行两个XGBoost工作程序，则应用程序将失败。用户可以使用外部机制来监控应用程序的状态，并获得此类情况的通知。

#### （5）训练期间的检查点Checkpoint

瞬态故障也常见于生产环境中。为简化XGBoost的设计，如果任何分布式工作程序失败，我们将停止训练。但是，如果经过很长一段时间后训练失败，那将是对资源的极大浪费。

我们支持在训练期间创建检查点，以便更有效地从故障中恢复。要启用此功能，您可以设置我们为每个检查点构建的迭代次数setCheckpointInterval以及检查点的位置setCheckpointPath：

xgbClassifier.setCheckpointInterval(2)

xgbClassifier.setCheckpointPath("/checkpoint\_path")

一种等效的方法是在XGBoostClassifier的构造函数中传入参数：

val xgbParam = Map("eta" -> 0.1f,

"max\_depth" -> 2,

"objective" -> "multi:softprob",

"num\_class" -> 3,

"num\_round" -> 100,

"num\_workers" -> 2,

"checkpoint\_path" -> "/checkpoints\_path",

"checkpoint\_interval" -> 2)

val xgbClassifier = new XGBoostClassifier(xgbParam).

setFeaturesCol("features").

setLabelCol("classIndex")

如果在这100轮中训练失败，则下一轮训练将从读取最新检查点文件/checkpoints\_path开始，并在检查点建立时从迭代开始，直到下一次失败或指定的100轮。

### 1.10.9 Xgboost+Spark代码实战

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

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

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

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

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

import org.apache.spark.sql.SparkSession

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

import ml.dmlc.xgboost4j.scala.spark.{XGBoostClassifier, XGBoostClassificationModel}

// this example works with Iris dataset (https://archive.ics.uci.edu/ml/datasets/iris)

object SparkMLlibPipeline {

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

    if (args.length != 3) {

      println("Usage: SparkMLlibPipeline input\_path native\_model\_path pipeline\_model\_path")

      sys.exit(1)

    }

    val inputPath = args(0)

    val nativeModelPath = args(1)

    val pipelineModelPath = args(2)

    val spark = SparkSession

      .builder()

      .appName("XGBoost4J-Spark Pipeline Example")

      .getOrCreate()

    // Load dataset

    val schema = new StructType(Array(

      StructField("sepal length", DoubleType, true),

      StructField("sepal width", DoubleType, true),

      StructField("petal length", DoubleType, true),

      StructField("petal width", DoubleType, true),

      StructField("class", StringType, true)))

    val rawInput = spark.read.schema(schema).csv(inputPath)

    // Split training and test dataset

    val Array(training, test) = rawInput.randomSplit(Array(0.8, 0.2), 123)

    // Build ML pipeline, it includes 4 stages:

    // 1, Assemble all features into a single vector column.

    // 2, From string label to indexed double label.

    // 3, Use XGBoostClassifier to train classification model.

    // 4, Convert indexed double label back to original string label.

    val assembler = new VectorAssembler()

      .setInputCols(Array("sepal length", "sepal width", "petal length", "petal width"))

      .setOutputCol("features")

    val labelIndexer = new StringIndexer()

      .setInputCol("class")

      .setOutputCol("classIndex")

      .fit(training)

    val booster = new XGBoostClassifier(

      Map("eta" -> 0.1f,

        "max\_depth" -> 2,

        "objective" -> "multi:softprob",

        "num\_class" -> 3,

        "num\_round" -> 100,

        "num\_workers" -> 2

      )

    )

    booster.setFeaturesCol("features")

    booster.setLabelCol("classIndex")

    val labelConverter = new IndexToString()

      .setInputCol("prediction")

      .setOutputCol("realLabel")

      .setLabels(labelIndexer.labels)

    val pipeline = new Pipeline()

      .setStages(Array(assembler, labelIndexer, booster, labelConverter))

    val model = pipeline.fit(training)

    // Batch prediction

    val prediction = model.transform(test)

    prediction.show(false)

    // Model evaluation

    val evaluator = new MulticlassClassificationEvaluator()

    evaluator.setLabelCol("classIndex")

    evaluator.setPredictionCol("prediction")

    val accuracy = evaluator.evaluate(prediction)

    println("The model accuracy is : " + accuracy)

    // Tune model using cross validation

    val paramGrid = new ParamGridBuilder()

      .addGrid(booster.maxDepth, Array(3, 8))

      .addGrid(booster.eta, Array(0.2, 0.6))

      .build()

    val cv = new CrossValidator()

      .setEstimator(pipeline)

      .setEvaluator(evaluator)

      .setEstimatorParamMaps(paramGrid)

      .setNumFolds(3)

    val cvModel = cv.fit(training)

    val bestModel = cvModel.bestModel.asInstanceOf[PipelineModel].stages(2)

      .asInstanceOf[XGBoostClassificationModel]

    println("The params of best XGBoostClassification model : " +

      bestModel.extractParamMap())

    println("The training summary of best XGBoostClassificationModel : " +

      bestModel.summary)

    // Export the XGBoostClassificationModel as local XGBoost model,

    // then you can load it back in local Python environment.

    bestModel.nativeBooster.saveModel(nativeModelPath)

    // ML pipeline persistence

    model.write.overwrite().save(pipelineModelPath)

    // Load a saved model and serving

    val model2 = PipelineModel.load(pipelineModelPath)

    model2.transform(test).show(false)

  }

}

### 1.10.10 Xgboost+Flink代码实战

import ml.dmlc.xgboost4j.scala.flink.XGBoost

import org.apache.flink.api.scala.{ExecutionEnvironment, \_}

import org.apache.flink.ml.MLUtils

object DistTrainWithFlink {

def main(args: Array[String]) {

val env: ExecutionEnvironment = ExecutionEnvironment.getExecutionEnvironment

// read trainining data

val trainData =

MLUtils.readLibSVM(env, "/path/to/data/agaricus.txt.train")

val testData = MLUtils.readLibSVM(env, "/path/to/data/agaricus.txt.test")

// define parameters

val paramMap = List(

"eta" -> 0.1,

"max\_depth" -> 2,

"objective" -> "binary:logistic").toMap

// number of iterations

val round = 2

// train the model

val model = XGBoost.train(trainData, paramMap, round)

val predTest = model.predict(testData.map{x => x.vector})

model.saveModelAsHadoopFile("file:///path/to/xgboost.model")

}

}

## 1.11项目案例：构建SparkML集成学习分类模型项目实战

（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）

### 1.11.1SparkMllib实战Iris数据分类模型

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

代码实战如下：

package com.test.spark.sparkmllib\_helloworld.iris\_ml\_3

/\*

General Approach to Analyse ML Problem

Stage(1) - Getting Raw data.

Stage(2) - Data Exploration

1. Quantitative Analysis

a. Variable Identification.

b. Distribution Analysis on Continuous variables.

i. Outlier detection.

ii. Skewness detection.

c. Null Detection on all variables.

2. Feature Identification - i.e. identifying features that influence Target Variable.

a. Categorical-Target variable graphical Analysis.

b. Continuous-Target variable graphical Analysis.

c. Hypothesis Testing.

d. Correlation.

3. Feature Munging

a. Null Removal

c. Outlier Removal

Stage(3) - Setting Pipeline Stages

i. Feature Extractors

ii. Problem Solver

Stage(4) - Dataset Splitting [(Training), (Validation) and (Testing)]

Stage(5) - Problem Solver Preparation.

i. Setting up Pipeline/Estimator.

ii. Setting up Parameter grid.

iii. Setting up Evaluator.

Stage(6) - Hyperparameter Tuning using (Training) and (Validation).

Stage(7) - Best Model Testing using (Testing).

Stage(8) - Best Model Evaluation.

Stage(9) - Saving/Showing Results to Repository

i. Testing Results.

ii. Evaluation Results.

iii. The best model.

Note:- To ensure expected results, print Schemas for Data set generated in between stages.

\*/

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

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

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

import org.apache.spark.ml.stat.Correlation

import org.apache.spark.ml.classification.{RandomForestClassifier, RandomForestClassificationModel}

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

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

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

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

object IrisClassification {

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

// Comment this to view all Log4j messages

Logger.getLogger("org").setLevel(Level.OFF)

Logger.getLogger("akka").setLevel(Level.OFF)

//--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Stage(1) - Getting Raw data.

val sqlContext: SparkSession = SparkSession.builder().appName("Iris Dataset Solver").master("local[\*]").getOrCreate()

val rawDF: DataFrame = sqlContext.read.format("csv").option("header", "true").option("inferSchema", "true").load("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/sparkmllib\_helloworld/iris\_ml\_3/iris.csv").toDF("sepal\_length", "sepal\_width", "petal\_length", "petal\_width", "class")

rawDF.printSchema()

rawDF.show(10, truncate = false)

//--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Stage(2) - Data Exploration

// 1. Quantitative Analysis

// a. Variable Identification.

// Continuous : sepal\_length,sepal\_width,petal\_length,petal\_width

// Categorical : class

rawDF.describe().show(false)

// b. Distribution Analysis on Continuous variables.

// i. Outlier detection.

// Almost no outliers in any feature

// ii. Skewness detection.

// sepal\_length - Slightly right skewed + Not Normal Distribution

// sepal\_width - No Skewness + Normal Distribution

// petal\_length and petal\_width - Not Normal Distribution

// c. Null Detection on all variables.

for (column <- rawDF.columns)

println(f"Null values in $column%s : ${rawDF.filter(rawDF(column) === null || rawDF(column).isNull || rawDF(column).isNaN).count()}%s")

// 2. Feature Identification - i.e. identifying features that influence Target Variable.

// a. Categorical-Target variable graphical Analysis.

// b. Continuous-Target variable graphical Analysis.

// c. Hypothesis Testing.

// d. Correlation.

val vectorAssembler: VectorAssembler = new VectorAssembler().setInputCols(Array("sepal\_length", "sepal\_width", "petal\_length", "petal\_width")).setOutputCol("feature")

val corrDF: DataFrame = vectorAssembler.transform(rawDF)

Correlation.corr(corrDF, "feature", "pearson").show(false)

/\*

Conclusion :

1. sepal\_length variates slightly -ve with sepal\_width but very +ve with petal\_length,petal\_width

2. sepal\_width variates -ve with petal\_length,petal\_width.

3. petal\_length variates very +ve with petal\_width.

Inference : petal\_length,petal\_width variates very +ve with sepal\_length but -ve with sepal\_width, hence all features can decide type of Iris flower

\*/

// 3. Feature Munging

// a. Null Removal - No Nulls

// b. Outlier Removal - No Outliers

val filteredDF: DataFrame = rawDF

filteredDF.printSchema()

//--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Stage(3) - Setting Pipeline Stages

// i. Feature Extractors

val classToIndex: StringIndexer = new StringIndexer().setInputCol("class").setOutputCol("indexedClass").setHandleInvalid("keep")

val featureAssembler: VectorAssembler = new VectorAssembler().setInputCols(Array("sepal\_length", "sepal\_width", "petal\_length", "petal\_width")).setOutputCol("feature")

val indexToClass: IndexToString = new IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").setLabels(classToIndex.fit(filteredDF).labels)

// ii. Problem Solver

val problemSolver: RandomForestClassifier = new RandomForestClassifier().setLabelCol("indexedClass").setFeaturesCol("feature").setPredictionCol("prediction")

//--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Stage(4) - Dataset Splitting [(Training), (Validation) and (Testing)]

val splitDF: Array[DataFrame] = filteredDF.randomSplit(Array(0.5, 0.3, 0.2), seed = 11L)

val (trainingDF: DataFrame, validationDF: DataFrame, testingDF: DataFrame) = (splitDF(0), splitDF(1), splitDF(2))

//--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Stage(5) - Problem Solver Preparation.

// i. Setting up Pipeline/Estimator.

val pipeline: Pipeline = new Pipeline().setStages(Array(classToIndex, featureAssembler, problemSolver, indexToClass))

// ii. Setting up Parameter grid.

val paramGrid: Array[ParamMap] = new ParamGridBuilder().addGrid(problemSolver.impurity, Array("gini", "entropy")).addGrid(problemSolver.maxDepth, Array(4, 5)).addGrid(problemSolver.numTrees, Array(5, 6)).build()

// iii. Setting up Evaluator.

val evaluator: MulticlassClassificationEvaluator = new MulticlassClassificationEvaluator().setLabelCol("indexedClass").setPredictionCol("prediction").setMetricName("accuracy")

//--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Stage(6) - Hyperparameter Tuning using (Training) and (Validation).

val crossValidator: CrossValidator = new CrossValidator().setEstimator(pipeline).setEstimatorParamMaps(paramGrid).setEvaluator(evaluator).setNumFolds(4)

val crossValidatorModel: CrossValidatorModel = crossValidator.fit(trainingDF)

val crossValidatorDF: DataFrame = crossValidatorModel.transform(validationDF)

crossValidatorDF.printSchema()

//--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Stage(7) - Best Model Testing using (Testing).

val bestPipelineModel: PipelineModel = crossValidatorModel.bestModel.asInstanceOf[PipelineModel]

val evaluationTestingDF: DataFrame = bestPipelineModel.transform(testingDF)

//--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Stage(8) - Best Model Evaluation.

val accuracyValidation: Double = evaluator.evaluate(crossValidatorDF)

val accuracyTesting: Double = evaluator.evaluate(evaluationTestingDF)

//--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Stage(9) - Saving/Showing Results to Repository

// i. Testing Results.

evaluationTestingDF.show(false)

// ii. Evaluation Results.

println(f"Model Accuracy over Validation set : ${accuracyValidation \* 100}%2.2f %%")

println(f"Model Accuracy over Testing set : ${accuracyTesting \* 100}%2.2f %%")

// iii. The best model.

val bestRFModel: RandomForestClassificationModel = bestPipelineModel.stages(2).asInstanceOf[RandomForestClassificationModel]

println("The best RFClassification model properties")

println(s"1.Impurity : ${bestRFModel.getImpurity}")

println(f"2.Number of tress : ${bestRFModel.getNumTrees}%d")

println(f"3.Max Depth : ${bestRFModel.getMaxDepth}%d")

println(s"4.Model Description : \n${bestRFModel.toDebugString}")

}

}

### 1.11.2SparkMllib实战Titanic数据分类模型

数据描述

|变量 | 定义| 说明|  
 | :-------------: |:-------------:| : -----:|  
 |survival | 是否幸存 | 0 = No, 1 = Yes | label  
 |pclass| 船舱等级| 1 = 1st, 2 = 2nd, 3 = 3rd |  
 |sex | 性别| |  
 |Age | 年龄| |  
 |sibsp| 有多少兄弟姐妹一起乘船 | |  
 |parch|有 多少父母与儿子一起乘船 | |  
 |ticket| 船票编号| |  
 |fare | 船票价格| |  
 |cabin| 舱数量| |  
 |embarked| 上船地点| C = Cherbourg, Q = Queenstown, S = Southampton|  
数据：

PassengerId,Pclass,Name,Sex,Age,SibSp,Parch,Ticket,Fare,Cabin,Embarked

892,3,"Kelly, Mr. James",male,34.5,0,0,330911,7.8292,,Q

893,3,"Wilkes, Mrs. James (Ellen Needs)",female,47,1,0,363272,7,,S

894,2,"Myles, Mr. Thomas Francis",male,62,0,0,240276,9.6875,,Q

895,3,"Wirz, Mr. Albert",male,27,0,0,315154,8.6625,,S

896,3,"Hirvonen, Mrs. Alexander (Helga E Lindqvist)",female,22,1,1,3101298,12.2875,,S

897,3,"Svensson, Mr. Johan Cervin",male,14,0,0,7538,9.225,,S

898,3,"Connolly, Miss. Kate",female,30,0,0,330972,7.6292,,Q

899,2,"Caldwell, Mr. Albert Francis",male,26,1,1,248738,29,,S

900,3,"Abrahim, Mrs. Joseph (Sophie Halaut Easu)",female,18,0,0,2657,7.2292,,C

901,3,"Davies, Mr. John Samuel",male,21,2,0,A/4 48871,24.15,,S

902,3,"Ilieff, Mr. Ylio",male,,0,0,349220,7.8958,,S

903,1,"Jones, Mr. Charles Cresson",male,46,0,0,694,26,,S

904,1,"Snyder, Mrs. John Pillsbury (Nelle Stevenson)",female,23,1,0,21228,82.2667,B45,S

905,2,"Howard, Mr. Benjamin",male,63,1,0,24065,26,,S

906,1,"Chaffee, Mrs. Herbert Fuller (Carrie Constance Toogood)",female,47,1,0,W.E.P. 5734,61.175,E31,S

907,2,"del Carlo, Mrs. Sebastiano (Argenia Genovesi)",fem

代码：

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

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.classification.{RandomForestClassificationModel, RandomForestClassifier}

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

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

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

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

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

import org.apache.spark.sql.{SparkSession, \_}

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

object TitanicMain {

val conf = new SparkConf()

.setAppName("Titan")

.setMaster("local[\*]")

.set("spark.sql.shuffle.partitions", "2")

val sc = new SparkContext(conf)

val sqlContext = SparkSession.builder.config(conf).getOrCreate()

def computeAverageAge(training: DataFrame, test: DataFrame) = {

training.select("Age")

.union(test.select("Age"))

.agg(avg("Age"))

.head().getDouble(0)

}

def main(args: Array[String]) {

var training = sqlContext.read.format("com.databricks.spark.csv")

.option("header", "true")

.option("delimiter", ",")

.option("inferSchema", "true")

.load("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/sparkmllib\_helloworld/titanic\_ml\_1/train.csv")

var test = sqlContext.read.format("com.databricks.spark.csv")

.option("header", "true")

.option("delimiter", ",")

.option("inferSchema", "true")

.load("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/sparkmllib\_helloworld/titanic\_ml\_1/test.csv")

//schema

println(test.schema)

// clean age

val averageAge = computeAverageAge(training, test)

training = training.na.fill(averageAge, Array("Age"))

test = test.na.fill(averageAge, Array("Age"))

// clean Embarked

training = training.na.fill("S", Array("Embarked"))

test = test.na.fill("S", Array("Embarked"))

// clean null Fare

training = training.na.drop("any", Array("Fare"))

test = test.na.fill(8.05, Array("Fare"))

// add title

val nameToTitle: String => String = \_.replaceAll(""".\*, |\..\*""", "") match {

case "Miss" => "Miss"

case "Mr" => "Mr"

case "Mrs" => "Mrs"

case \_ => "RareTitle"

}

training = training.withColumn("title", udf(nameToTitle).apply(training("Name")))

test = test.withColumn("title", udf(nameToTitle).apply(test("Name")))

// Add mother or not

val mother = when(expr("Sex == 'female' AND Age > 18.0 AND Parch > 1"), 1).otherwise(0)

training = training.withColumn("mother", mother)

test = test.withColumn("mother", mother)

// Index Sex

val sexIndexer = new StringIndexer()

.setInputCol("Sex")

.setOutputCol("SexIndex")

.fit(training)

// Index Embarked

val embarkedIndexer = new StringIndexer()

.setInputCol("Embarked")

.setOutputCol("EmbarkedIndex")

.fit(training)

// Title Embarked

val titleIndexer = new StringIndexer()

.setInputCol("title")

.setOutputCol("titleIndex")

.fit(training)

// Index Survived as the Label

val labelIndexer = new StringIndexer()

.setInputCol("Survived")

.setOutputCol("Label")

.fit(training)

val features: Array[String] = Array("Pclass", "Age", "Fare", "SexIndex", "EmbarkedIndex", "titleIndex", "mother")

// Features

val assembler = new VectorAssembler()

.setInputCols(features)

.setOutputCol("Features")

val randomForest = new RandomForestClassifier()

.setLabelCol("Label")

.setFeaturesCol("Features")

val pipeline = new Pipeline().setStages(

Array(sexIndexer,

embarkedIndexer,

titleIndexer,

labelIndexer,

assembler,

randomForest

)

)

// train the model

val model = pipeline.fit(training)

// display the features importance in the RandomForest algo

val importances: Vector = model.stages(5).asInstanceOf[RandomForestClassificationModel].featureImportances

println("feature importances:")

println(features.mkString(", "))

println(importances)

// Use the model to make predictions

val predictions = model.transform(test)

predictions.selectExpr("PassengerId", "cast(prediction as int) Survived")

.repartition(1)

.write

.format("com.databricks.spark.csv")

.option("header", "true")

.save("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/sparkmllib\_helloworld/titanic\_ml\_1/predictions.csv")

predictions.show()

// With cross validation

val params = new ParamGridBuilder()

.addGrid(randomForest.maxDepth, Array(4, 8, 12))

.addGrid(randomForest.numTrees, Array(15, 30, 50))

.addGrid(randomForest.maxBins, Array(16, 32, 64))

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

.build()

val evaluator = new BinaryClassificationEvaluator()

.setLabelCol("Label")

.setRawPredictionCol("prediction")

.setMetricName("areaUnderPR")

val cv = new CrossValidator()

.setEstimator(pipeline)

.setEvaluator(evaluator)

.setEstimatorParamMaps(pa rams)

.setNumFolds(10)

val crossValidatorModel = cv.fit(training)

val predictionsCV = crossValidatorModel.transform(test)

predictionsCV.selectExpr("PassengerId", "cast(prediction as int) Survived")

.repartition(1)

.write

.format("com.databricks.spark.csv")

.option("header", "true")

.save("/Users/zhao-chj/develop/IdeaProjects/SparkTest/src/main/scala/com/test/spark/sparkmllib\_helloworld/titanic\_ml\_1/predictionsCV.csv")

predictionsCV.show()

}

}

### 1.11.3Xgboost4J代码实战

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

Xgboost代码实战：

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

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

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

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

import org.apache.spark.sql.SparkSession

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

import ml.dmlc.xgboost4j.scala.spark.{XGBoostClassifier, XGBoostClassificationModel}

// this example works with Iris dataset (https://archive.ics.uci.edu/ml/datasets/iris)

object SparkMLlibPipeline {

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

    if (args.length != 3) {

      println("Usage: SparkMLlibPipeline input\_path native\_model\_path pipeline\_model\_path")

      sys.exit(1)

    }

    val inputPath = args(0)

    val nativeModelPath = args(1)

    val pipelineModelPath = args(2)

    val spark = SparkSession

      .builder()

      .appName("XGBoost4J-Spark Pipeline Example")

      .getOrCreate()

    // Load dataset

    val schema = new StructType(Array(

      StructField("sepal length", DoubleType, true),

      StructField("sepal width", DoubleType, true),

      StructField("petal length", DoubleType, true),

      StructField("petal width", DoubleType, true),

      StructField("class", StringType, true)))

    val rawInput = spark.read.schema(schema).csv(inputPath)

    // Split training and test dataset

    val Array(training, test) = rawInput.randomSplit(Array(0.8, 0.2), 123)

    // Build ML pipeline, it includes 4 stages:

    // 1, Assemble all features into a single vector column.

    // 2, From string label to indexed double label.

    // 3, Use XGBoostClassifier to train classification model.

    // 4, Convert indexed double label back to original string label.

    val assembler = new VectorAssembler()

      .setInputCols(Array("sepal length", "sepal width", "petal length", "petal width"))

      .setOutputCol("features")

    val labelIndexer = new StringIndexer()

      .setInputCol("class")

      .setOutputCol("classIndex")

      .fit(training)

    val booster = new XGBoostClassifier(

      Map("eta" -> 0.1f,

        "max\_depth" -> 2,

        "objective" -> "multi:softprob",

        "num\_class" -> 3,

        "num\_round" -> 100,

        "num\_workers" -> 2

      )

    )

    booster.setFeaturesCol("features")

    booster.setLabelCol("classIndex")

    val labelConverter = new IndexToString()

      .setInputCol("prediction")

      .setOutputCol("realLabel")

      .setLabels(labelIndexer.labels)

    val pipeline = new Pipeline()

      .setStages(Array(assembler, labelIndexer, booster, labelConverter))

    val model = pipeline.fit(training)

    // Batch prediction

    val prediction = model.transform(test)

    prediction.show(false)

    // Model evaluation

    val evaluator = new MulticlassClassificationEvaluator()

    evaluator.setLabelCol("classIndex")

    evaluator.setPredictionCol("prediction")

    val accuracy = evaluator.evaluate(prediction)

    println("The model accuracy is : " + accuracy)

    // Tune model using cross validation

    val paramGrid = new ParamGridBuilder()

      .addGrid(booster.maxDepth, Array(3, 8))

      .addGrid(booster.eta, Array(0.2, 0.6))

      .build()

    val cv = new CrossValidator()

      .setEstimator(pipeline)

      .setEvaluator(evaluator)

      .setEstimatorParamMaps(paramGrid)

      .setNumFolds(3)

    val cvModel = cv.fit(training)

    val bestModel = cvModel.bestModel.asInstanceOf[PipelineModel].stages(2)

      .asInstanceOf[XGBoostClassificationModel]

    println("The params of best XGBoostClassification model : " +

      bestModel.extractParamMap())

    println("The training summary of best XGBoostClassificationModel : " +

      bestModel.summary)

    // Export the XGBoostClassificationModel as local XGBoost model,

    // then you can load it back in local Python environment.

    bestModel.nativeBooster.saveModel(nativeModelPath)

    // ML pipeline persistence

    model.write.overwrite().save(pipelineModelPath)

    // Load a saved model and serving

    val model2 = PipelineModel.load(pipelineModelPath)

    model2.transform(test).show(false)

  }

}