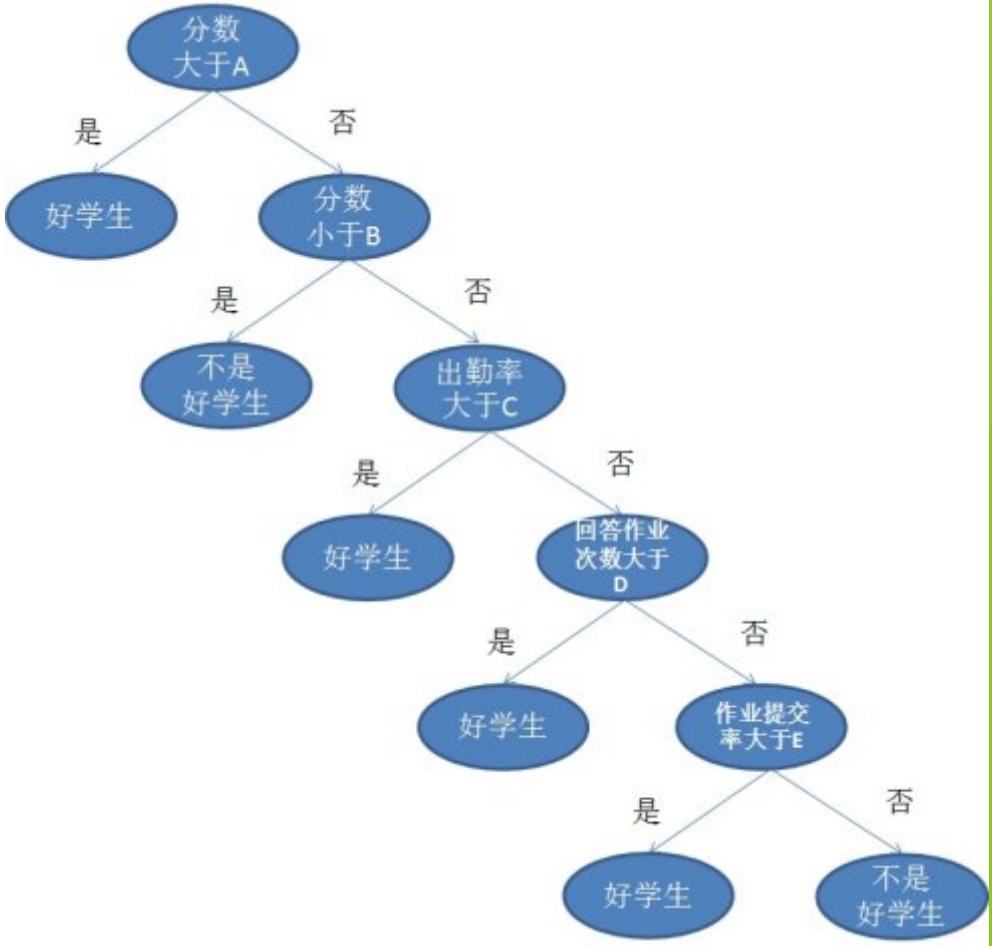
决策树和集成学习

Guangrui Qian

决策树模型 - 示例

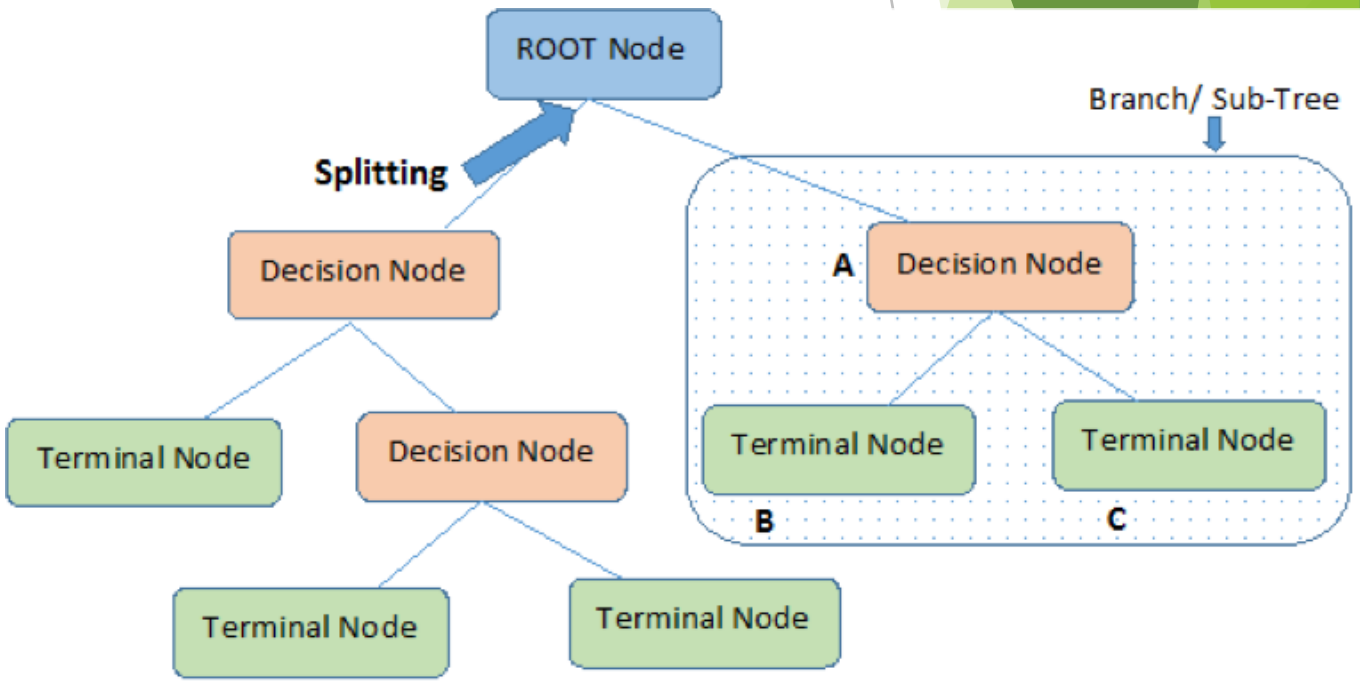
学生编号	分数	出勤率	回答问题次数	作业提交率	分类:是否好学生
1	99	80%	5	90%	是
2	89	100%	6	100%	是
3	69	100%	7	100%	否
4	50	60%	8	70%	否
5	95	70%	9	80%	否
6	98	60%	10	80%	是
7	92	65%	11	100%	是
8	91	80%	12	85%	是
9	85	80%	13	95%	是
10	85	91%	14	98%	是



- 决策树是一种机器学习的方法。决策树的生成算法有ID3, C4.5和C5.0等。决策树是一种树形结构，其中每个内部节点表示一个属性上的判断，每个分支代表一个判断结果的输出，最后每个叶节点代表一种分类结果。

决策树相关的重要概念

- 根结点(Root Node)：它表示整个样本集合，并且该节点可以进一步划分成两个或多个子集。
- 拆分(Splitting)：表示将一个结点拆分成多个子集的过程。
- 决策结点(Decision Node)：当一个子结点进一步被拆分成多个子节点时，这个子节点就叫做决策结点。
- 叶子结点(Leaf/Terminal Node)：无法再拆分的结点被称为叶子结点。
- 剪枝(Pruning)：移除决策树中子结点的过程就叫做剪枝，跟拆分过程相反。
- 分支/子树(Branch/Sub-Tree)：一棵决策树的一部分就叫做分支或子树。
- 父结点和子结点(Parent and Child Node)：一个结点被拆分成多个子节点，这个结点就叫做父节点；其拆分后的子结点也叫做子结点。

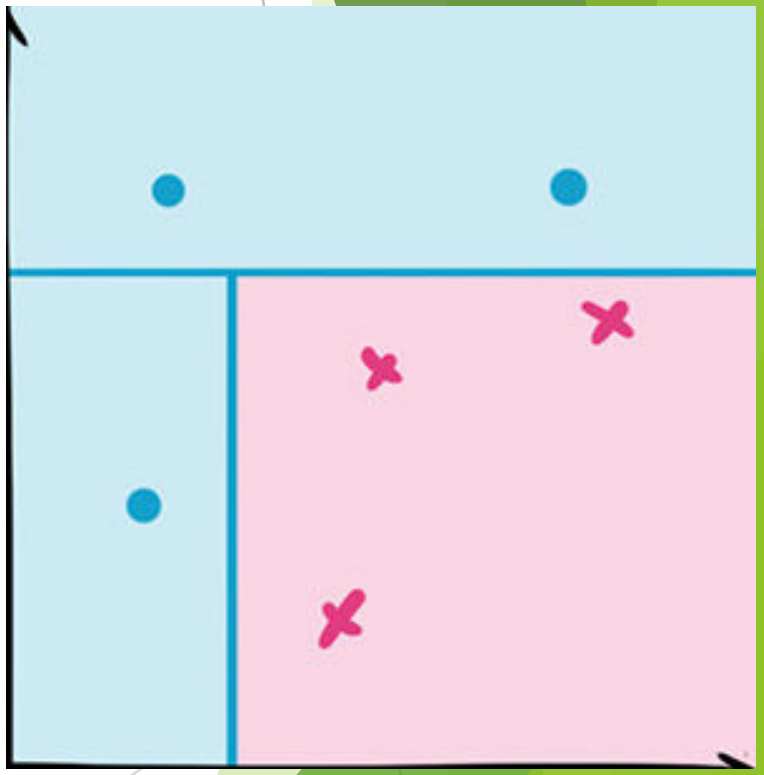


决策树

- 决策树是一种依托决策而建立起来的一种树。在机器学习中，决策树是一种预测模型，代表的是一种对象属性与对象值之间的一种映射关系，每一个节点代表某个对象，树中的每一个分叉路径代表某个可能的属性值，而每一个叶子节点则对应从根节点到该叶子节点所经历的路径所表示的对象的值。

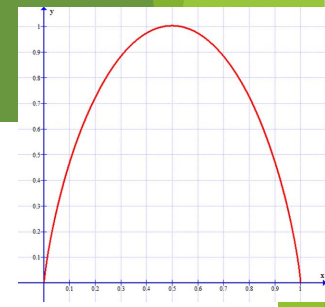
- 决策树仅有单一输出，如果有多个输出，可以分别建立独立的决策树以处理不同的输出。

- 决策树的特点是它总是在沿着特征做切分。随着层层递进，这个划分会越来越细。



信息熵

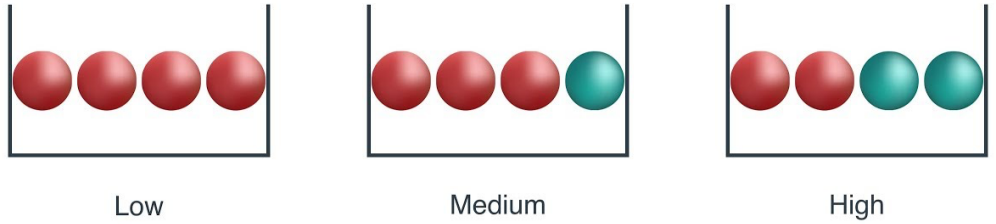
- 在信息论中，熵是接收的每条消息中包含的信息的平均量，又被称为信息熵、信源熵、平均自信息量。这里，“消息”代表来自分布或数据流中的事件、样本或特征。熵的单位通常为比特，但也用Sh、nat、Hart计量，取决于定义用到对数的底。



Entropy

熵： $H(Y) = -\sum_{i=1}^{K} p_i \log p_i$

Gini： $Gini(Y) = \sum_{k=1}^{K} p_k(1-p_k) = 1 - \sum_{k=1}^{K} p_k^2$



- 物理中的”熵“一种直观的定义就是表征一个系统的混乱程度，越混乱，熵值越大，越有序，则熵值越低。
- 在信息熵中，符号越多信息熵越大，也就是单个符号提供的信息越多。符号频率越均匀，信息熵越高。
- 例如语言文字越”混乱“，信息量则越大。语言间关联小，字与字之间出现的频率差距不大，每个符号都很关键，都不能丢，所以符号信息量大

如果一个符号系统只有1个符号，那么这个符号系统什么信息都不能传递，单个字符能传递的信息是0。如果一种文字只有一个字母a，一a到底，那么这种语言真的啥信息能不能传递。

信息增益和信息增益率

- 熵是对事件结果不确定性的度量，但在知道有些条件时，不确定性会变小。信息增益是知道了某个条件后，事件的不确定性下降的程度。
- 信息增益（Info-Gain）指的就是熵的减少量
- 与熵一样，基尼系数表征的也是事件的不确定性，将熵定义式中的 “$-\log P_i$”替换为 $1-P_i$ 就是基尼系数。

熵： $H(Y) = -\sum_{i=1}^{K} p_i \log p_i$

Gini： $Gini(Y) = \sum_{k=1}^{K} p_k(1-p_k) = 1 - \sum_{k=1}^{K} p_k^2$

假设变量 $X = \{x_1, x_2 \ldots x_i \ldots x_n\}$，其中每个元素对应的概率（比例）为 $P = \{p_1, p_2 \ldots p_i \ldots p_n\}$，则对应熵的计算公式如下：

$$E(X) = -\sum_{i=1}^{n} p_i log_2(p_i)$$

$$IGain(S, A) = E(S) - E(A)$$

上面公式中S和A分别代表操作前后的数据划分状态。

信息增益

$$Info = -\sum_{v \in value(A)} \frac{num(S_v)}{num(S)} log_2 \frac{num(S_v)}{num(S)},$$

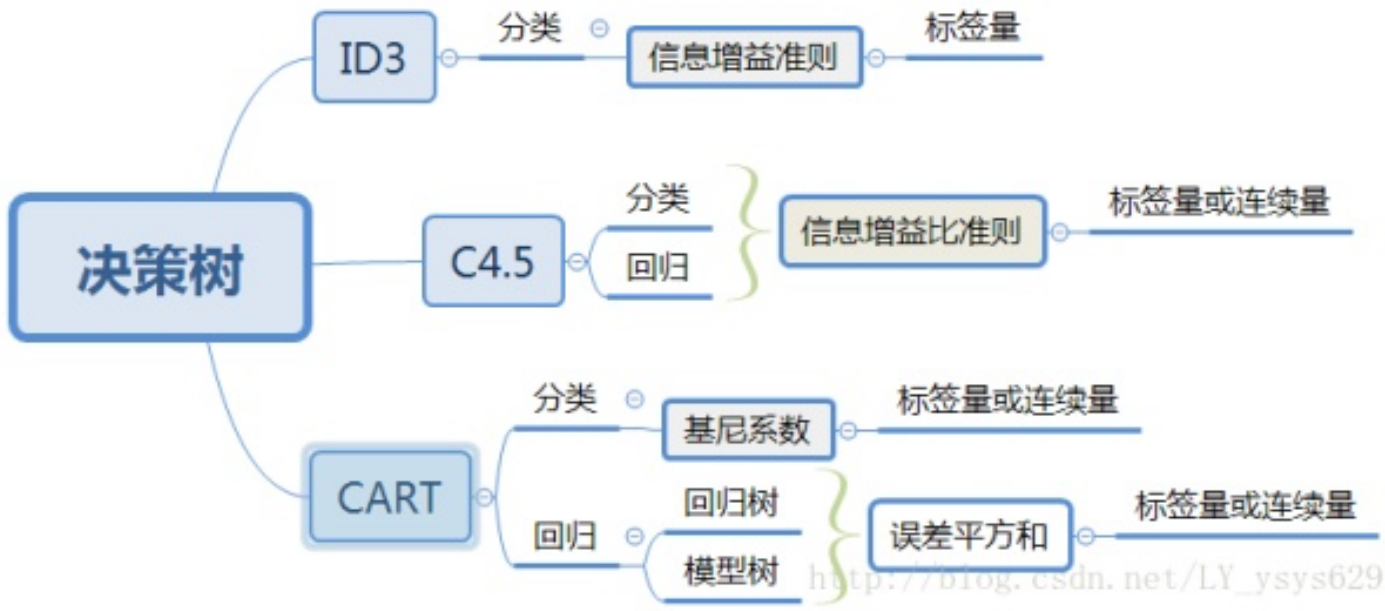
这里Info为划分行为带来的信息，信息增益率如下计算：

$$Gain - ratio = \frac{IGain(S, A)}{Info}$$

信息增益率

3种典型的决策树算法

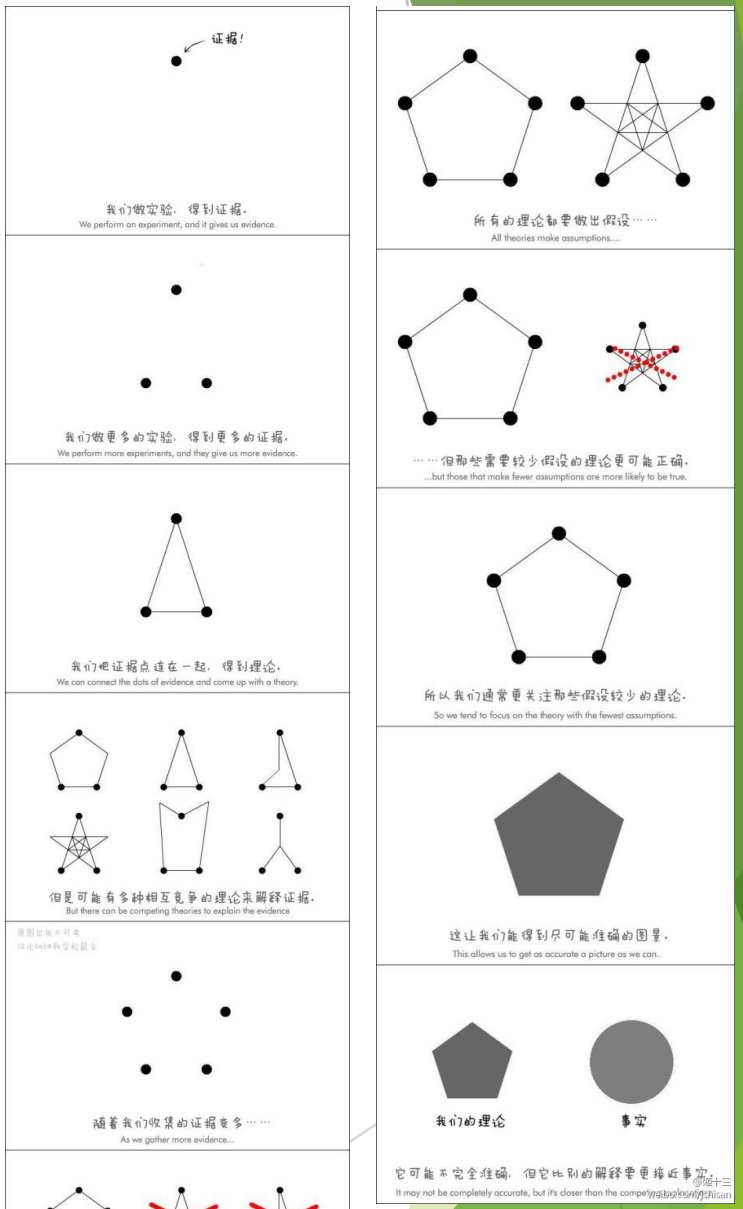
- **ID3 算法：**
  - ID3 是最早提出的决策树算法，他就是利用信息增益来选择特征的。
- **C4.5 算法：**
  - ID3 的改进版，他不是直接使用信息增益，而是引入“信息增益比”指标作为特征的选择依据。
- **CART (Classification and Regression Tree)：**
  - 这种算法即可以用于分类，也可以用于回归问题。CART 算法使用了基尼系数取代了信息熵模型。



http://blog.csdn.net/LY_ysys629

ID3算法介绍

- ID3算法(即Iterative Dichotomiser 3，迭代二叉树3代)是决策树的一种，是Ross Quinlan发明的一种决策树算法。
- ID3是基于奥卡姆剃刀原理的，即用尽量用较少的东西做更多的事。越是小型的决策树越优于大的决策树，尽管如此，也不总是生成最小的树型结构，而是一个启发式算法。
- ID3算法的核心思想就是以信息增益来度量属性的选择，选择分裂后信息增益最大的属性进行分裂。该算法采用自顶向下的贪婪搜索遍历可能的决策空间。
- **信息增益大，则越适合用来分类**
- **所有的叶子结点都是纯的划分过程中止**



奥卡姆剃刀原理:

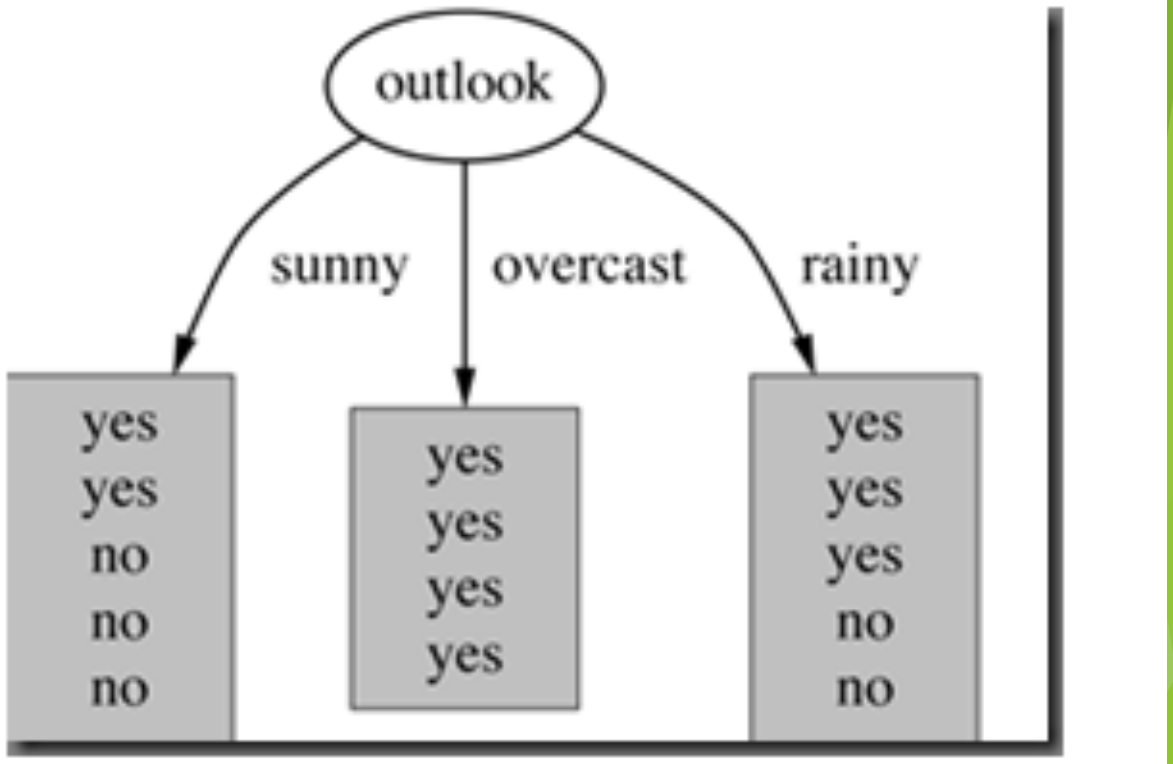
如无必要，勿增实体

信息增益计算 - 1

表1 天气预报数据集例子

Outlook	Temperature	Humidity	Windy	Play?
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rain	mild	high	false	yes
rain	cool	normal	false	yes
rain	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rain	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rain	mild	high	true	no

分类前

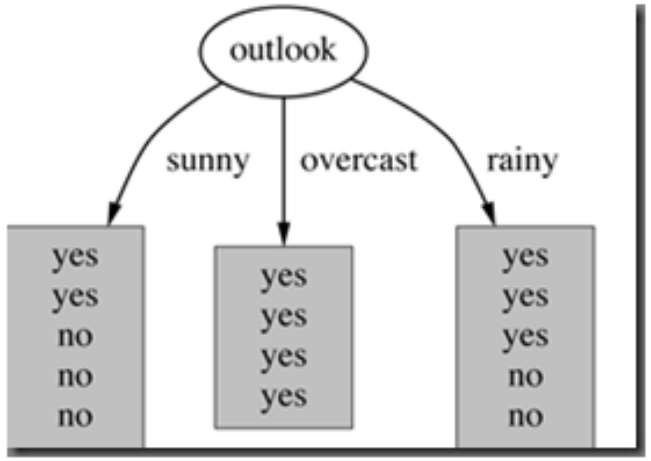


分类后

信息增益计算 - 2

表1 天气预报数据集例子

Outlook	Temperature	Humidity	Windy	Play?
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rain	mild	high	false	yes
rain	cool	normal	false	yes
rain	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rain	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rain	mild	high	true	no



分类前

$$Entropy(S) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0.940286$$

分类后

各个分支的信息熵

$$Entropy(sunny) = -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0.970951$$

$$Entropy(overcast) = -\frac{4}{4}\log_2\frac{4}{4} - 0\cdot\log_2 0 = 0$$

$$Entropy(rainy) = -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} = 0.970951$$

划分后的信息熵
(条件熵)

$$Entropy(S|T) = \frac{5}{14}\cdot 0.970951 + \frac{4}{14}\cdot 0 + \frac{5}{14}\cdot 0.970951 = 0.693536$$

信息增益

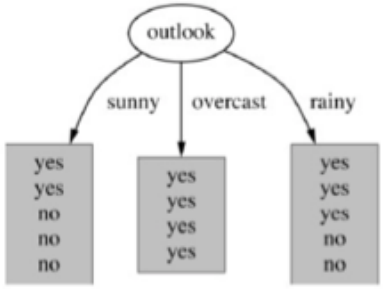
$$IG(T) = Entropy(S) - Entropy(S|T) = 0.24675$$

决策树实际案例

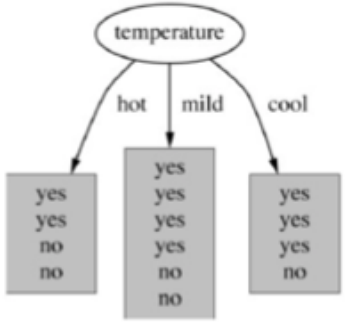
表1 天气预报数据集例子

Outlook	Temperature	Humidity	Windy	Play?
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rain	mild	high	false	yes
rain	cool	normal	false	yes
rain	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rain	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rain	mild	high	true	no

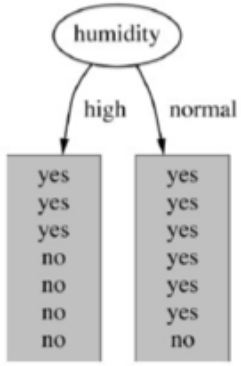
1.基于天气的划分



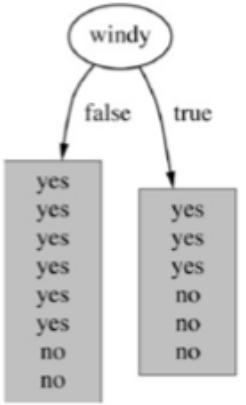
2.基于温度的划分



3.基于湿度的划分



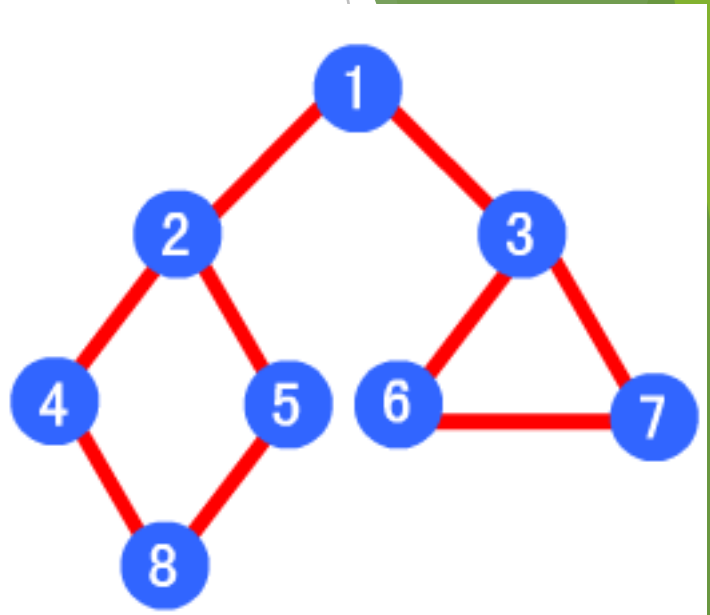
4.基于有风的划分



深度优先、广度优先

- 深度优先遍历，从初始访问结点出发，我们知道初始访问结点可能有多个邻接结点，深度优先遍历的策略就是首先访问第一个邻接结点，然后再以这个被访问的邻接结点作为初始结点，访问它的第一个邻接结点。总结起来可以这样说：每次都在访问完当前结点后首先访问当前结点的第一个邻接结点。

- 类似于一个分层搜索的过程，广度优先遍历需要使用一个队列以保持访问过的结点的顺序，以便按这个顺序来访问这些结点的邻接结点。



深度优先遍历：

1->2->4->8->5->3->6->7

广度优先遍历：

1->2->3->4->5->6->7->8

决策树ID3算法的不足

- ID3没有考虑连续特征，连续值无法在ID3运用。这大大限制了ID3的用途。
- ID3采用信息增益大的特征优先建立决策树的节点。很快就被人发现，在相同条件下，取值比较多的特征比取值少的特征信息增益大。比如一个变量有2个值，各为1/2，另一个变量为3个值，各为1/3，其实他们都是完全不确定的变量，但是取3个值的比取2个值的信息增益大。该问题需要校正。
- ID3算法对于缺失值的情况没有做考虑
- 没有考虑过拟合的问题

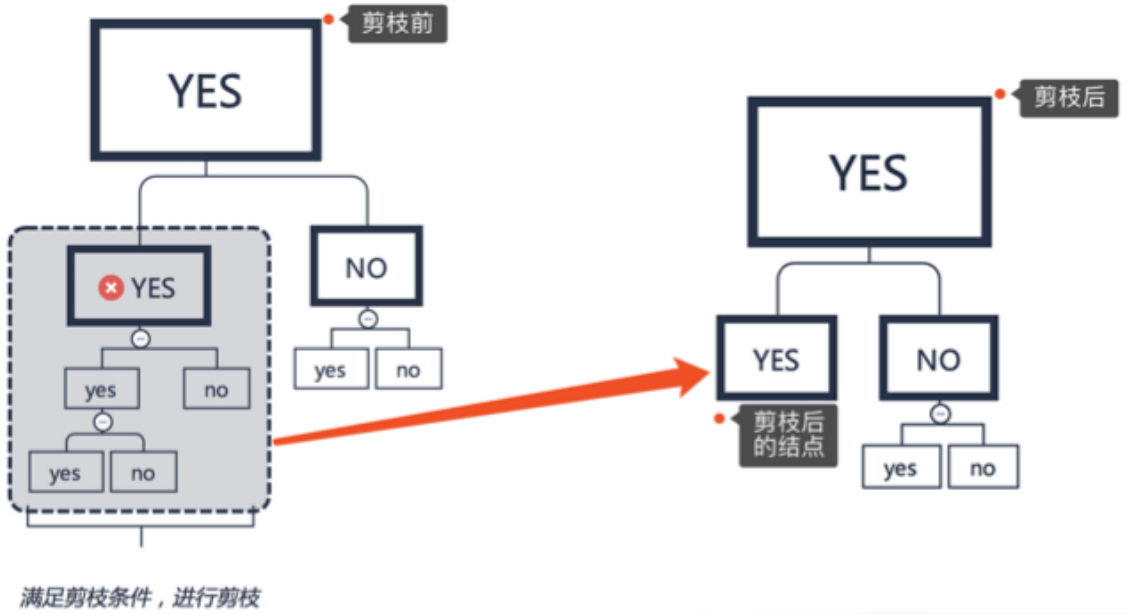
若某一列数据没有重复，ID3算法倾向于把每个数据自成一类，此时

$$E(A) = \sum_{i=1} \frac{1}{n} log_2(1) = 0 ,$$

这样E(A)为最小，IGain(S,A)最大，程序会倾向于选择这种划分，这样划分效果极差。

决策树C4.5算法

- C4.5的思路是将连续的特征离散化。比如m个样本的连续特征A有m个，从小到大排列为a1,a2,...,ama1,a2,...,am,则C4.5取相邻两样本值的平均数，一共取得m-1个划分点。
- 信息增益作为标准容易偏向于取值较多的特征的问题。特征数越多的特征对应的特征熵越大，它作为分母，可以校正信息增益容易偏向于取值较多的特征的问题。
- 第三个缺失值处理的问题：
  - 在样本某些特征缺失的情况下选择划分的属性：
  - 选定了划分属性，对于在该属性上缺失特征的样本的处理
- C4.5引入了正则化系数进行初步的剪枝。



$$Info = -\sum_{v \in value(A)} \frac{num(S_v)}{num(S)} log_2 \frac{num(S_v)}{num(S)},$$

这里Info为划分行为带来的信息，信息增益率如下计算：

$$Gain - ratio = \frac{IGain(S, A)}{Info}$$

决策树C4.5算法缺失值处理

在样本某些特征缺失的情况下选择划分的属性：

- 对于某一个有缺失特征值的特征A，C4.5的思路是将数据分成两部分，对每个样本设置一个权重（初始可以都为1），然后划分数据，一部分是有特征值A的数据D1，另一部分是没有特征A的数据D2。然后对于没有缺失特征A的数据集D1来和对应的A特征的各个特征值一起计算加权重后的信息增益比，最后乘上一个系数，这个系数是无特征A缺失的样本加权后所占加权总样本的比例。

选定了划分属性，对于在该属性上缺失特征的样本的处理

- 可以将缺失特征的样本同时划分入所有的子节点，不过将该样本的权重按各个子节点样本的数量比例来分配。比如缺失特征A的样本a之前权重为1，特征A有3个特征值A1,A2,A3。 3个特征值对应的无缺失A特征的样本个数为2,3,4.则a同时划分入A1，A2，A3。对应权重调节为2/9,3/9, 4/9。

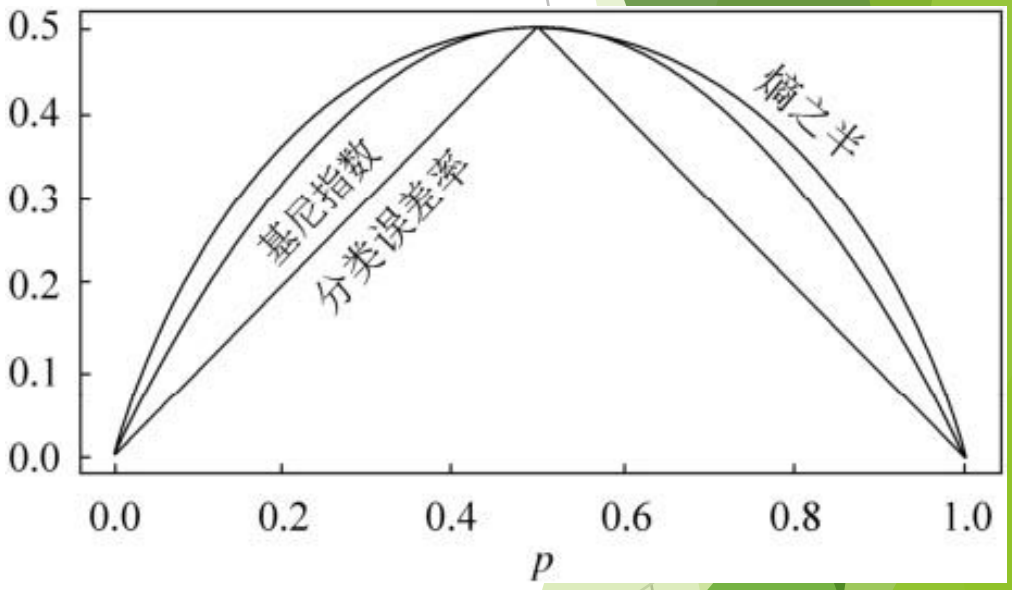
决策树C4.5算法不足

- 由于决策树算法非常容易过拟合，因此对于生成的决策树必须要进行剪枝。剪枝的算法有非常多，C4.5的剪枝方法有优化的空间。思路主要是两种，一种是预剪枝，即在生成决策树的时候就决定是否剪枝。另一个是后剪枝，即先生成决策树，再通过交叉验证来剪枝。
- C4.5生成的是多叉树，即一个父节点可以有多个节点。很多时候，在计算机中二叉树模型会比多叉树运算效率高。如果采用二叉树，可以提高效率。
- C4.5只能用于分类，如果能将决策树用于回归的话可以扩大它的使用范围。
- C4.5由于使用了熵模型，里面有大量的耗时的对数运算,如果是连续值还有大量的排序运算。如果能够加以模型简化可以减少运算强度但又不牺牲太多准确性的话，那就更好了。

CART分类树算法

- CART: Classification and Regression Trees
- CART分类树算法使用基尼系数来代替信息增益比，降低计算复杂度。基尼系数代表了模型的不纯度，基尼系数越小，则不纯度越低，特征越好。这和信息增益(比)是相反的。
- CART分类树算法每次仅仅对某个特征的值进行二分，而不是多分，这样CART分类树算法建立起来的是二叉树，而不是多叉树。这样一可以进一步简化基尼系数的计算，二可以建立一个更加优雅的二叉树模型。
- 对于CART分类树连续值的处理问题，其思想和C4.5是相同的，都是将连续的特征离散化。唯一的区别在于在选择划分点时的度量方式不同，C4.5使用的是信息增益比，则CART分类树使用的是基尼系数。 对于CART分类树离散值的处理问题，采用的思路是不停的二分离散特征。

CART分类树采用的是用基尼系数的大小来度量特征的各个划分点的优劣情况



对于二类分类，基尼系数和熵之半的曲线

熵： $H(Y) = -\sum_{i=1}^{K} p_i \log p_i$

Gini： $Gini(Y) = \sum_{k=1}^{K} p_k(1-p_k) = 1 - \sum_{k=1}^{K} p_k^2$

CART回归树

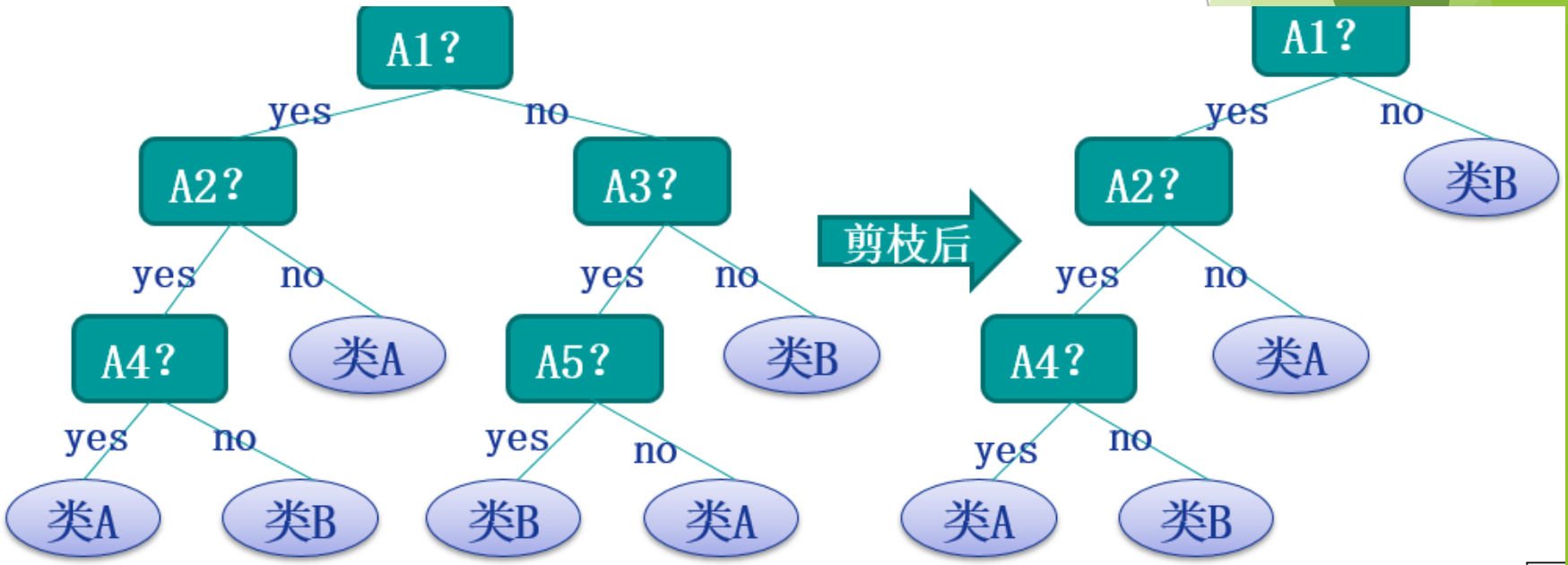
- 如果样本输出是离散值，那么这是一颗分类树。如果果样本输出是连续值，那么那么这是一颗回归树。
- CART回归树和CART分类树的建立和预测的区别主要有下面两点:
  - 连续值的处理方法不同
  - 决策树建立后做预测的方式不同
- 对于回归模型，我们使用了常见的和方差的度量方式，CART回归树的度量目标是，对于任意划分特征A，对应的任意划分点s两边划分成的数据集D1和D2，求出使D1和D2各自集合的均方差最小，同时D1和D2的均方差之和最小所对应的特征和特征值划分点:

$$\underbrace{min}_{A,s}\left[\underbrace{min}_{c_1}\sum_{x_i \in D_1(A,s)}(y_i - c_1)^2 + \underbrace{min}_{c_2}\sum_{x_i \in D_2(A,s)}(y_i - c_2)^2\right]$$

其中，c_1为D1数据集的样本输出均值，c_2为D2数据集的样本输出均值。

CART树算法的剪枝

- 由于决策时算法很容易对训练集过拟合，而导致泛化能力差，为了解决这个问题，我们需要对CART树进行剪枝，即类似于线性回归的正则化，来增加决策树的返回能力。
- CART采用的办法是后剪枝法，即先生成决策树，然后产生所有可能的剪枝后的CART树，然后使用交叉验证来检验各种剪枝的效果，选择泛化能力最好的剪枝策略。
- CART树的剪枝算法可以概括为两步，
  - 从原始决策树生成各种剪枝效果的决策树，
  - 用交叉验证来检验剪枝后的预测能力，选择泛化预测能力最好的剪枝后的数作为最终的CART树。



CART决策树算法损失函数度量

$$C_\alpha(T) = \sum_{t=1}^{|T|} N_t H_t(T) + \alpha|T|$$

咱们来看看决策树损失函数的定义。其中第一部分 $N_t H_t(T)$ 就是事物的不确定性次数。我们都知道存在噪声的情况下，模型将趋于复杂。在决策树中也就是 $|T|$ 的数值越大。再来看看损失函数中的 $\alpha|T|$，假设参数 α 已知的，那么复杂的模型，所带来的结果就是 $\alpha|T|$ 也将增大。且决策树存在过拟合现象，那么为了使得损失函数减小，有两种办法：

1. 降低第一部分的不确定次数，但我们知道这是不可能的了，因为降低不确定次数的办法是再找寻一个特征，或者找寻特征的最优切分点。这在生成决策时就已经决定了，无法改变。
2. 进行剪枝操作，这是可以的。剪枝最明显地变化就是叶结点个数变少。假设是一个三叉树，那么一次剪枝它的叶结点数减少2个。变化量为 2α，有了这变化量，我们就可以用来求解最优决策子树了。

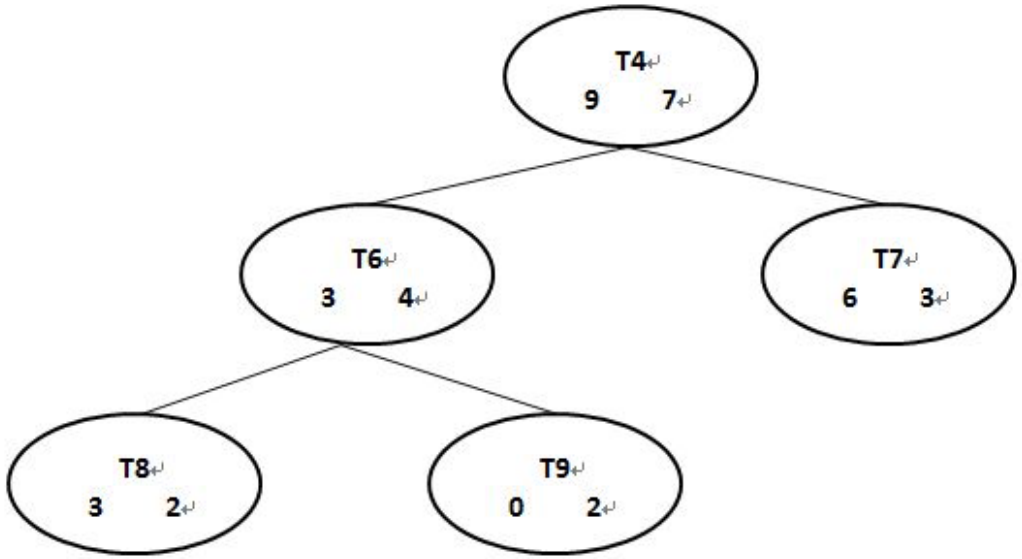
$$C_\alpha(T_t) = C(T_t) + \alpha|T_t|$$

为正则化参数，这和线性回归的正则化一样。C(Tt)为训练数据的预测误差 。|Tt|是子树T的叶子节点的数量

这公式是最初的决策树损失函数变化而来的！
它是其中一个子结点'吞并'它的子树，所得到'叶结点后'的损失函数。

剪枝方法

- Reduced-Error Pruning（REP, 错误率降低剪枝）
- Pesimistic-Error Pruning（PEP, 悲观错误剪枝）
- Cost-Complexity Pruning （CCP，代价复杂度剪枝）
- EBP(Error-Based Pruning)（基于错误的剪枝）



拿T6叶节点来说：

剪掉T6验证集准确度91%

不剪去T6验证集准确度90%

剪枝后准确率更高（错误率降低），所以剪枝！

悲观剪枝方法

- 悲观错误剪枝法是根据剪枝前后的错误率来判定子树的修剪。该方法引入了统计学上连续修正的概念弥补REP中的缺陷，在评价子树的训练错误公式中添加了一个常数，假定每个叶子结点都自动对实例的某个部分进行错误的分类。
- 把一颗子树（具有多个叶子节点）的分类用一个叶子节点来替代的话，在训练集上的误判率肯定是上升的，但是在新数据上不一定。于是我们需要把子树的误判计算加上一个经验性的惩罚因子。对于一颗叶子节点，它覆盖了N个样本，其中有E个错误，那么该叶子节点的错误率为(E+0.5)/N。这个0.5就是惩罚因子，那么一颗子树，它有L个叶子节点，
- 一颗子树虽然具有多个子节点，但由于加上了惩罚因子，所以子树的误判率计算未必占到便宜。剪枝后内部节点变成了叶子节点，其误判个数J也需要加上一个惩罚因子，变成J+0.5。那么子树是否可以被剪枝就取决于剪枝后的错误J+0.5在的标准误差内。对于样本的误差率e，我们可以根据经验把它估计成各种各样的分布模型，比如是二项式分布，比如是正态分布。

悲观剪枝是自上而下的对训练集建立的树。

$$e'(t) \le e'(T_t) + S_e(e'(T_t))$$

成立时，T_t 被剪枝。

上式中：

$$e'(t) = e(t) + 0.5$$

$$e'(T_t) = \Sigma e(i) + \frac{N_t}{2}$$

$$S_e(e'(t)) = \sqrt{\frac{e'(t) * (n_t - e'(t))}{n_t}}$$

e(t)为结点t出的误差；i为覆盖 T_t 的叶子结点； N_t 为 T_t 的叶子树； n_t 为 T_t 的数据数据量；

树中每个节点有两个数字，左边的代表正确，右边代表错误

CART算法不足

- 应该大家有注意到，无论是ID3, C4.5还是CART,在做特征选择的时候都是选择最优的一个特征来做分类决策，但是大多数，分类决策不应该是由某一个特征决定的，而是应该由一组特征决定的。这样绝息到的决策树更加准确。这个决策树叫做多变量决策树(multi-variate decision tree)。在选择最优特征的时候，多变量决策树不是选择某一个最优特征，而是选择最优的一个特征线性组合来做决策。

- 如果样本发生一点点的改动，就会导致树结构的剧烈改变。这个可以通过集成学习里面的随机森林之类的方法解决。

决策树预测的代价
$O(\log_2 m)$, m为样本数

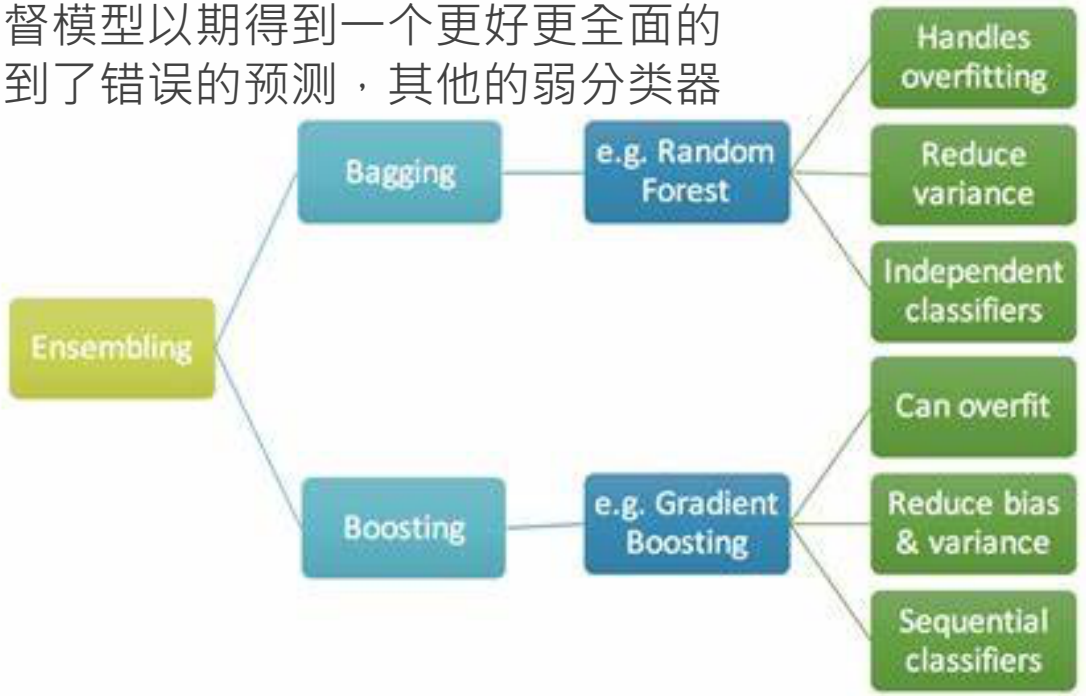
决策树算法总结

算法	支持模型	树结构	特征选择	连续值处理	缺失值处理	剪枝
ID3	分类	多叉树	信息增益	不支持	不支持	不支持
C4.5	分类	多叉树	信息增益比	支持	支持	支持
CART	分类，回归	二叉树	基尼系数，均方差	支持	支持	支持

http://blog.csdn.net/LY_ysys629

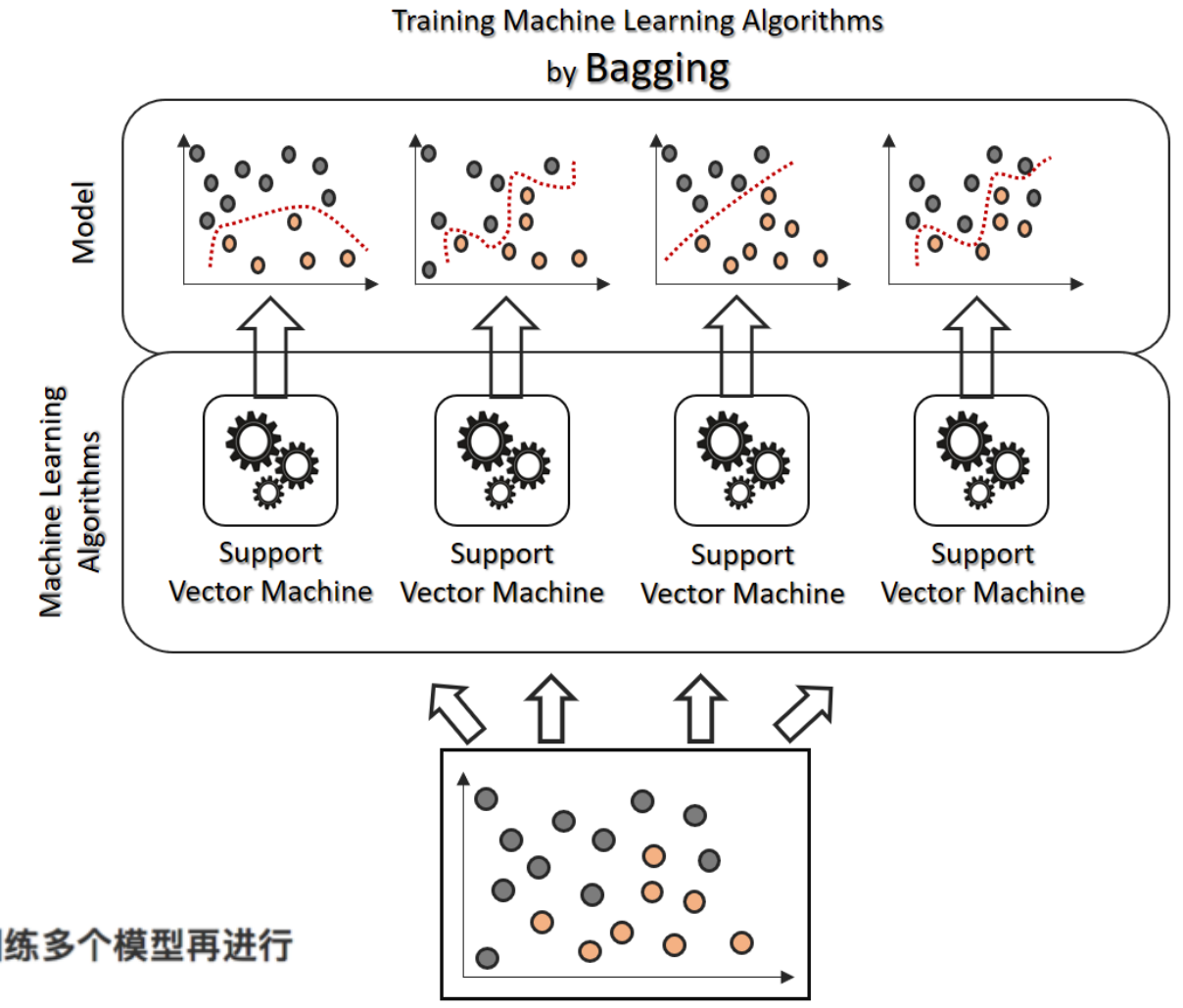
集成学习

- 在机器学习和统计学习中, Ensemble Learning(集成学习)是一种将多种学习算法组合在一起以取得更好表现的一种方法。与 Statistical Ensemble(统计总体，通常是无限的)不同，机器学习下的Ensemble 主要是指有限的模型相互组合，而且可以有很多不同的结构。相关的概念有多模型系统、Committee Learning、Modular systems、多分类器系统等等。
- 在机器学习的有监督学习算法中，我们的目标是学习出一个稳定的且在各个方面表现都较好的模型，但实际情况往往不这么理想，有时我们只能得到多个有偏好的模型（弱监督模型，在某些方面表现的比较好）。集成学习就是组合这里的多个弱监督模型以期得到一个更好更全面的强监督模型，集成学习潜在的思想是即便某一个弱分类器得到了错误的预测，其他的弱分类器也可以将错误纠正回来。
- 集成算法种类：Bagging，Boosting以及Stacking 和其他



集成学习特点 - 1

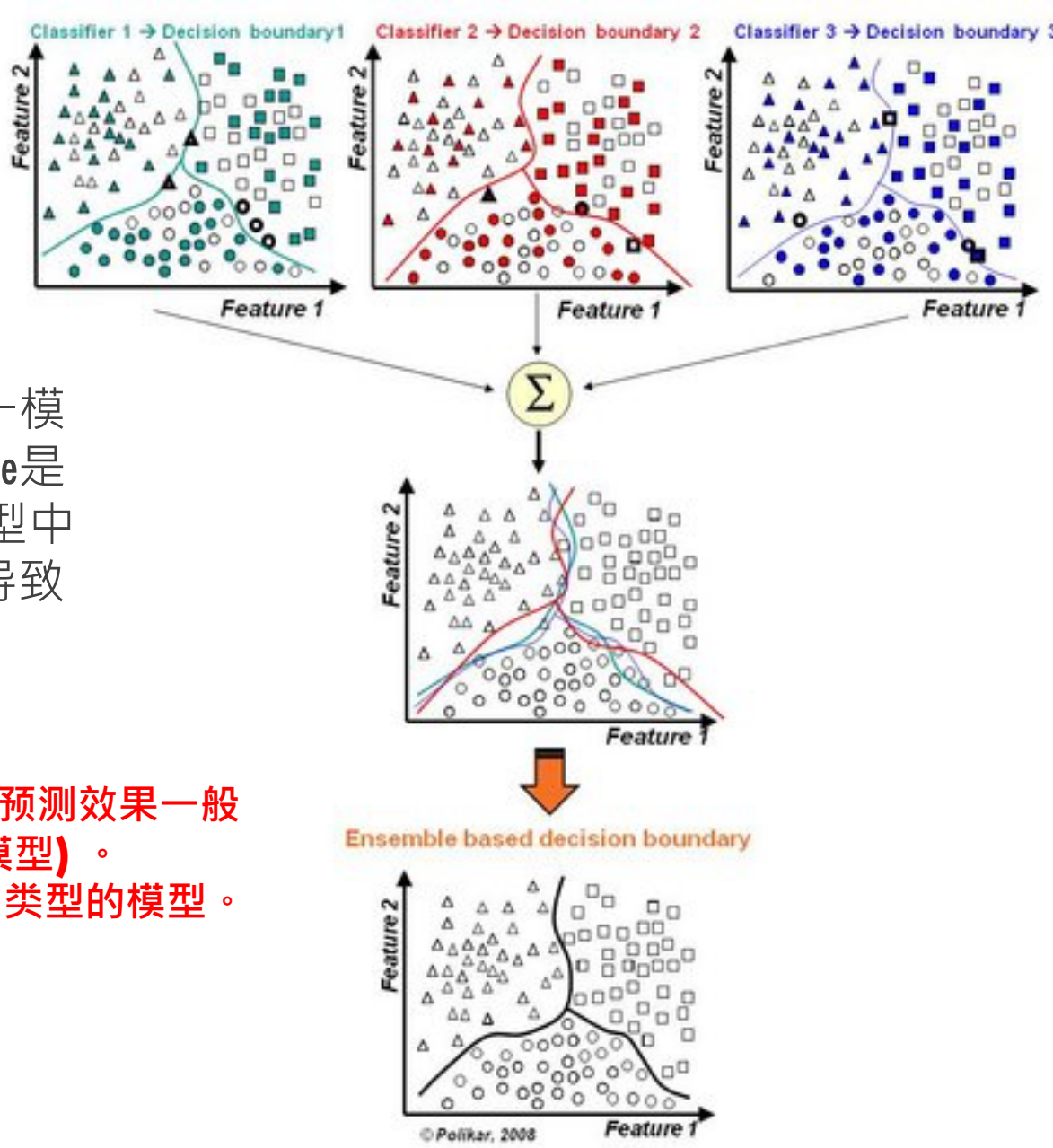
- Ensemble方法对于大量数据和不充分数据都有很好的效果。一些简单模型数据量太大而很难训练，或者只能学习到一部分，而Ensemble方法可以有策略的将数据集划分成一些小数据集，分别进行训练，之后根据一些策略进行组合。相反，如果数据量很少，可以使用bootstrap进行抽样，得到多个数据集，分别进行训练后再组合



- **数据集大：划分成多个小数据集，学习多个模型进行组合**
- **数据集小：利用Bootstrap方法进行抽样，得到多个数据集，分别训练多个模型再进行组合**

集成学习特点 - 2

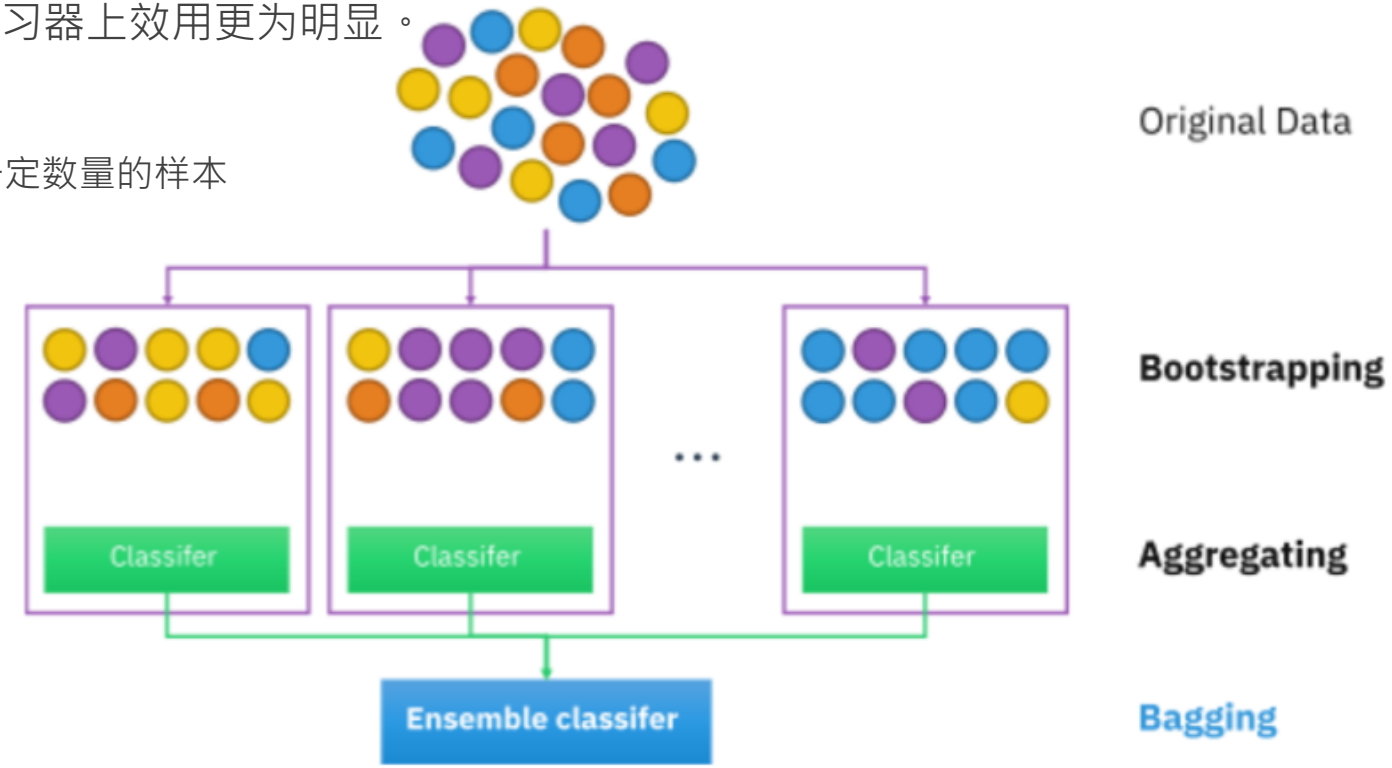
- 使用Ensemble的方法在评估测试的时候，相比于单一模型，需要更多的计算。因此，有时候也认为Ensemble是用更多的计算来弥补弱模型。 Ensemble方法导致模型中的每个参数所包含的信息量比单一模型少很多，导致太多的冗余



Ensemble的方法本质就是组合许多弱模型(weak learners，预测效果一般的模型) 以得到一个强模型(strong learner，预测效果好的模型) 。
Ensemble中组合的模型可以是同一类的模型，也可以是不同类型的模型。

Bagging

- bootstrap也称为自助法，它是一种**有放回的抽样方法**，目的为了得到统计量的分布以及置信区间。Bagging是bootstrap aggregating的简写。
- 在Bagging方法中，利用bootstrap方法从整体数据集中采取有放回抽样得到N个数据集，在每个数据集上学习出一个模型，最后的预测结果利用N个模型的输出得到，具体地：分类问题采用N个模型预测投票的方式，回归问题采用N个模型预测平均的方式。
- Bagging要求“不稳定”（不稳定是指数据集的小的变动能够使得分类结果的显著的变动）的分类方法。Bagging主要关注降低方差，因此它在不剪枝的决策树、神经网络等学习器上效用更为明显。
- Bagging方法具体步骤：
  - 采用重抽样方法（有放回抽样）从原始样本中抽取一定数量的样本
  - 根据抽出的样本计算想要得到的统计量T
  - 重复上述N次（一般大于1000），得到N个统计量T
  - 根据这N个统计量，即可计算出统计量的置信区间
- 典型Bagging集成算法：随机森林



Bagging算法描述和调用

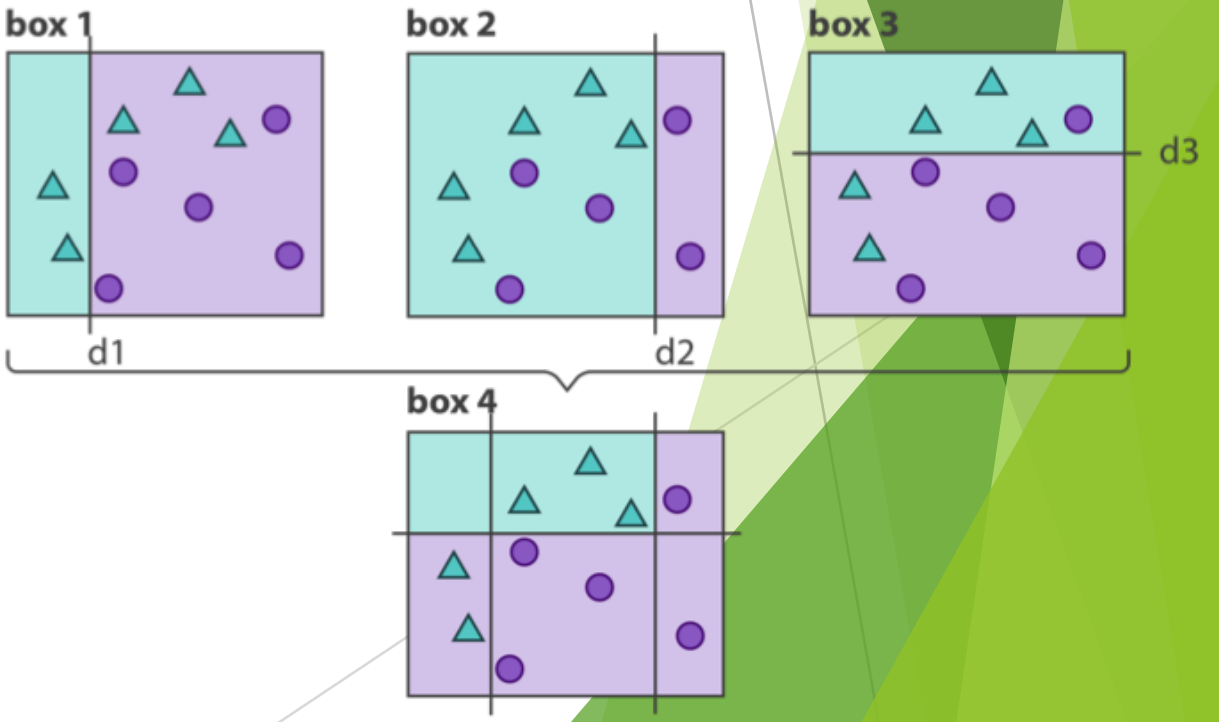
```
模型生成
        令n为训练数据的实例数量
        对于t次循环中的每一次
                从训练数据中采样n个实例
                将学习应用于所采样本
                保存结果模型
分类
        对于t个模型的每一个
                使用模型对实例进行预测
        返回被预测次数最多的一个
```

使用scikit-learn测试bagging方法

```
>>> from sklearn.ensemble import BaggingClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> bagging = BaggingClassifier(KNeighborsClassifier(),
...                             max_samples=0.5, max_features=0.5)
```

Boosting

- Boosting是一种框架算法，用来提高弱分类器准确度的方法，这种方法通过构造一个预测函数序列，然后以一定的方式将他们组合成为一个准确度较高的预测函数。
- Boosting方法可以用来减小监督学习中偏差的机器学习算法。主要也是学习一系列弱分类器，并将其组合为一个强分类器。
- Boosting通过不断的建立新模型而新模型更强调上一个模型中被错误分类的样本，再将这些模型组合起来的方法。在一些例子中，boosting要比bagging有更好的准确率，但是也更容易过拟合。
- Boosting主要关注降低偏差，因此Boosting能基于泛化性能相当弱的学习器构建出很强的集成；
- Boosting典型算法：AdaBoost（Adaptive boosting）算法，GBDT（Gradient Boost Decision Tree)



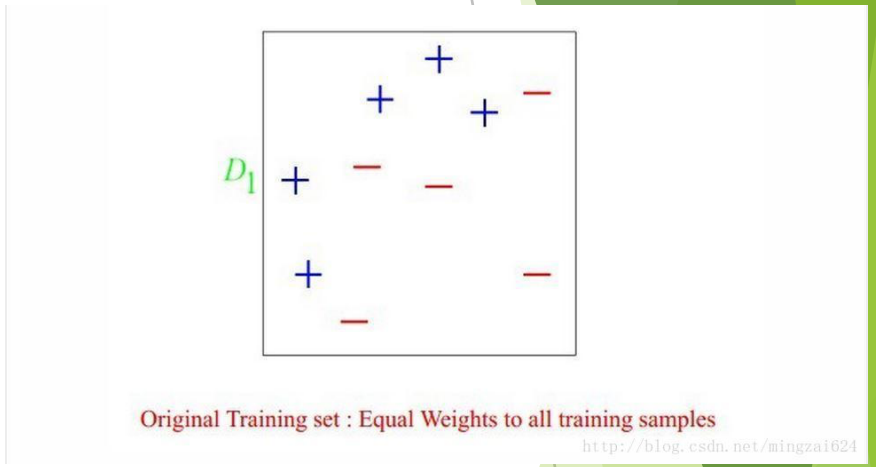
Boosting算法描述

- boosting: 其中主要的是Adaptive boosting。
- 自适应boosting方法计算时，初始化时对每一个训练例赋相等的权重1/N，然后用该学算法对训练集训练t轮，每次训练后，对训练失败的训练例赋以较大的权重，也就是让学习算法在后续的学习中集中对比较难的训练例进行学习，从而得到一个预测函数序列$h_1,\cdots,h_m$, 其中h_i也有一定的权重，预测效果好的预测函数权重较大，反之较小。
- 最终的预测函数H对分类问题采用有权重的投票方式，对回归问题采用加权平均的方法对新示例进行判别。

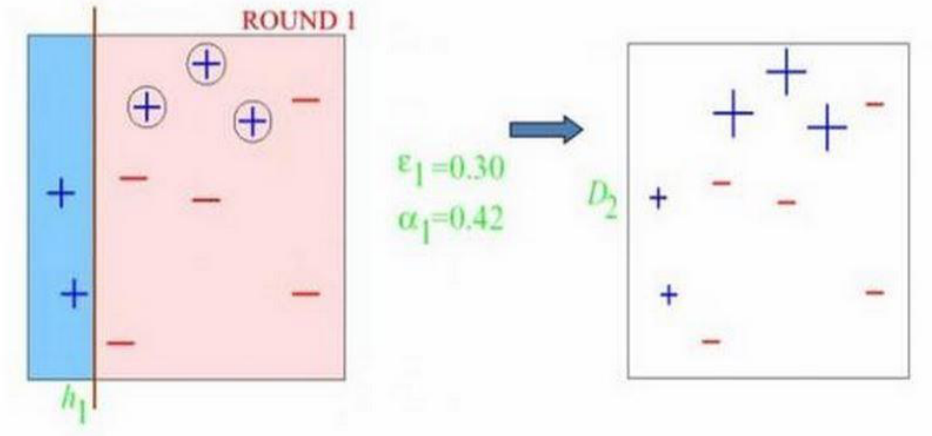
```
模型生成
        赋予每个训练实例相同的权值
        t次循环中的每一次:
                将学习算法应用于加了权的数据集上并保存结果模型
                计算模型在加了权的数据上的误差e并保存这个误差
                结果e等于0或者大于等于0.5:
                        终止模型
                对于数据集中的每个实例:
                        如果模型将实例正确分类
                                将实例的权值乘以e/(1-e)
                将所有的实例权重进行正常化
分类
        赋予所有类权重为0
        对于t（或小于t）个模型中的每一个:
                给模型预测的类加权 -log(e/(1-e))
        返回权重最高的类
```

AdaBoosting算法示意 - 1

Step 1: 算法开始前，需要将每个样本的权重初始化为**1/m,**这样一开始每个样本都是等概率的分布，每个分类器都会公正对待。



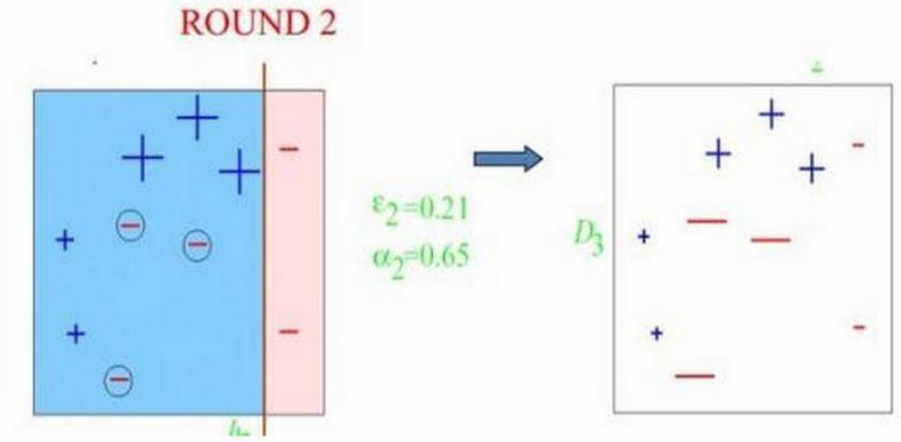
Round 1: 因为样本权重都一样，所以分类器开始划分，根据自己分类器的情况，只和分类器有关。划分之后发现分错了三个**"+"**号，那么这些分错的样本，在给下一个类器的时候权重就得到提高,也就是会影响到下次取训练样本的分布，就是提醒下一个分类器，**"**诶！你注意这几个小子，我上次栽在他们手里了！”



http://blog.csdn.net/mingzai624

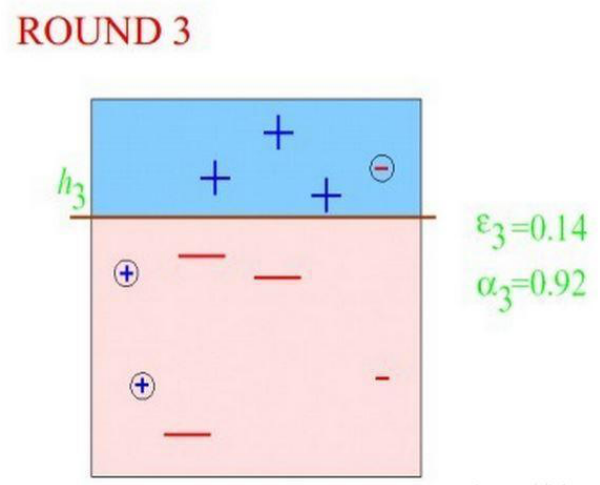
AdaBoosting算法示意 - 2

Round 2: 第二代分类器信誓旦旦的对上一代分类器说"我知道了，大哥！我一定睁大眼睛好好分着三个玩意！"ok，这次三个上次分错的都被分出来了，但是并不是全部正确，这次又栽倒在左下角三个"-"上了，然后临死前，第二代分类器对下一代分类器说"这次我和上一代分类器已经把他们摸得差不多了，你再稍微注意下左下角那三个小子，也别忘了上面那三个(一代错分的那三个"+")！"



http://blog.csdn.net/mingzai624

Round 3: 有了上面两位大哥的提醒，第三代分类器表示，我差不多都知道上次大哥们都错哪了，我只要小心这几个，应该没什么问题！只要把他们弄错的我给整对了，然后把我们收集的信息一对，这不就行了么！ok，第三代分类器不负众望，成功分对上面两代分类器重点关注的对象，至于分错的那几个小的，以前大哥们都分对了，我们坐下来核对一下就行了！

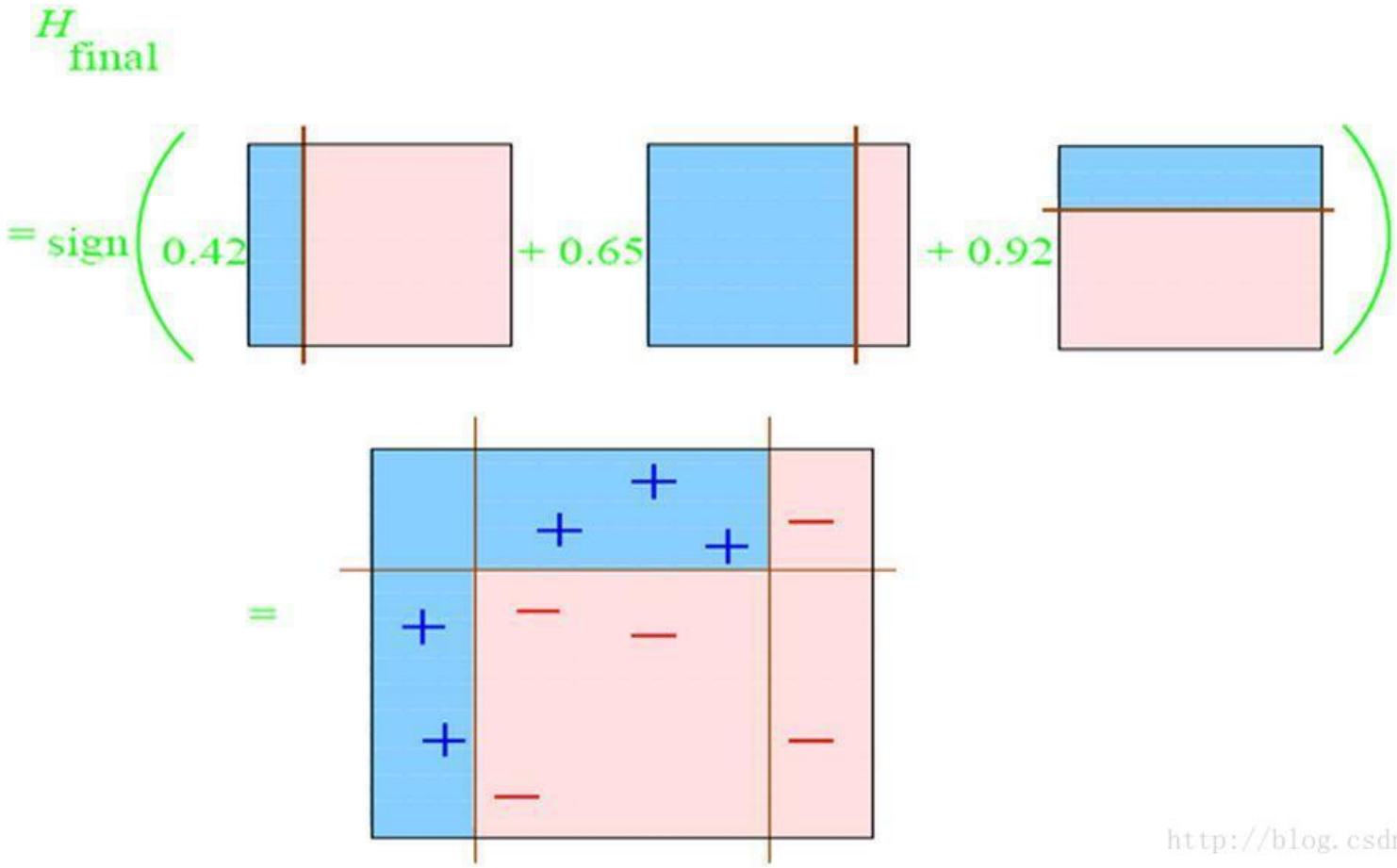


http://blog.csdn.net/mingzai624

AdaBoosting算法示意 - 3

Finally:

最后，三个分类器坐下来，各自谈了谈心得，分配了下权重。

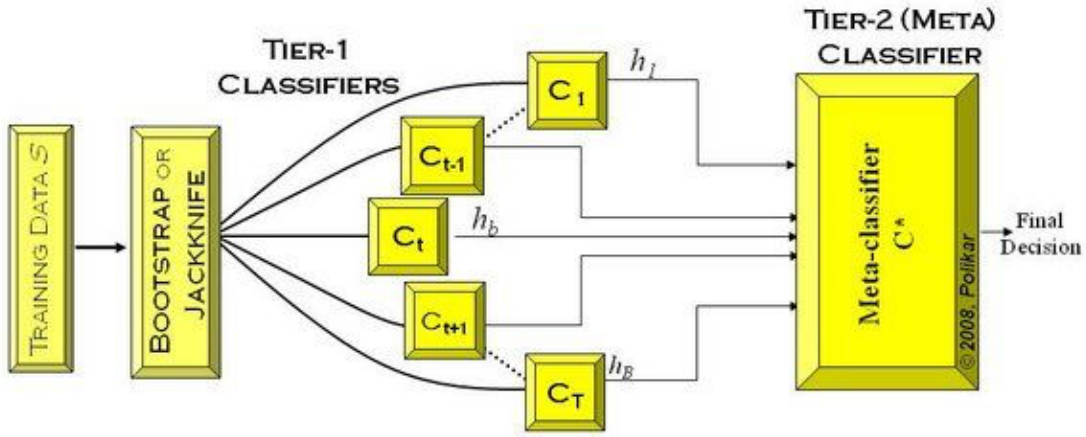


http://blog.csdn.net/mingzai62

Stacking方法

- Stacking方法是指训练一个模型用于组合其他各个模型。首先我们先训练多个不同的模型，然后把之前训练的各个模型的输出为输入来训练一个模型，以得到一个最终的输出。理论上，Stacking可以表示上面提到的两种Ensemble方法，只要我们采用合适的模型组合策略即可。但在实际中，我们通常使用logistic回归作为组合策略。

- Stacking 方法比任何单一模型的效果都要好，而且不仅成功应用在了监督式学习中，也成功应用在了非监督式(概率密度估计)学习中。



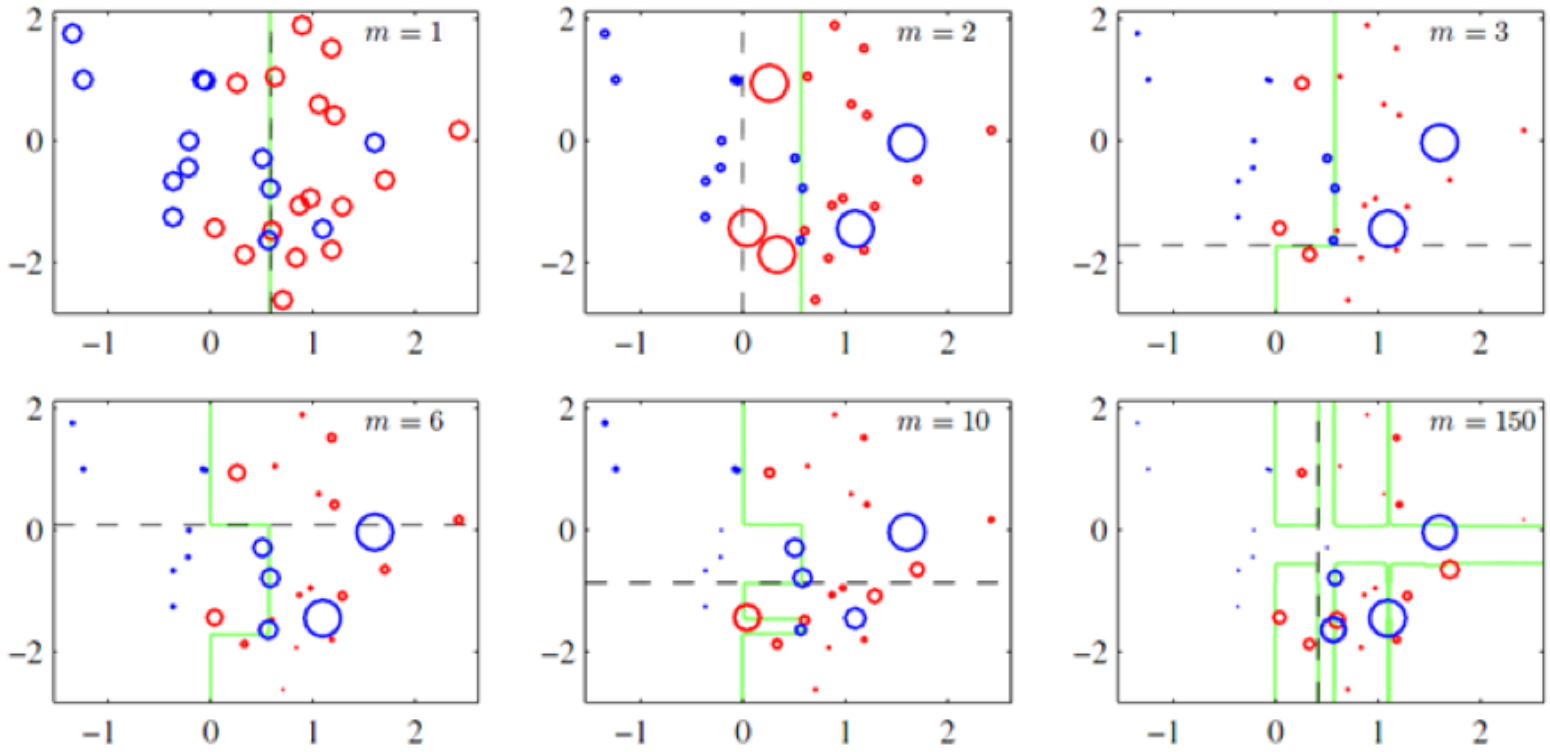
先在整个训练数据集上通过bootstrap抽样得到各个训练集合，得到一系列分类模型，称之为Tier 1分类器（可以采用交叉验证的方式学习），然后将输出用于训练 Tier 2 分类器。

集成算法组合特征

- Ensemble内的各个模型不仅仅可以是同一个模型根据训练集合的随机子集进行训练（得到不同的参数），也可以不同的模型进行组合、甚至可以是针对不同的特征子集进行训练。之后各个模型可以通过不同的策略进行组合。
- 不同的结果输出，组合的情况是不同的，主要可以分为三种情况：
  - Abstract-level:各个模型只输出一个目标类别，如猫、狗和人的图像识别中，仅输出人；
  - Rank-level:各个模型是输出目标类别的一个排序，如猫、狗和人的图像识别中，输出人-狗-猫；
  - measurement-level:各个模型输出的是目标类别的概率估计或一些相关的信念值，如猫、狗和人的图像识别中，输出0.7人-0.2狗-0.1猫；
- Ensemble组合时可以采用多种方式，比如根据均值或者加权等等。
  - Algebraic combiners(代数组合器)是非训练得到的组合器，这里通常用于有数值输出的情况。一般使用最小值、最大值、求和、均值、求积、中位数等等，进行最终的决策。
  - Voting based methods（基于投票），一般用于离散的情况，比较常见的是按众数决策。
  - 按照 Weighted majority voting，即各个模型的结果有不同的权重，加权得到最终的结果，这里的权重可以通过学习得到。

Bagging与Boosting - 1

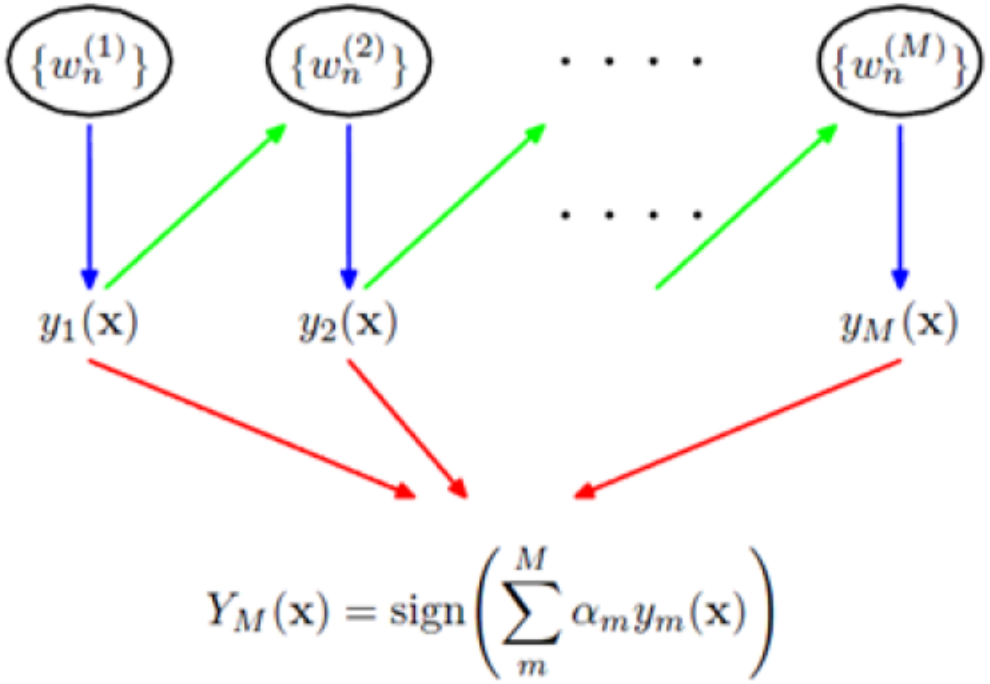
- Bagging和Boosting采用的都是采样-学习-组合的方式
- 二者的主要区别是取样方式不同。bagging采用均匀取样，而Boosting根据错误率来取样，因此boosting的分类精度要优于Bagging。
- bagging的训练集的选择是**随机的**，各轮训练集之间相互独立，而boostIng的各轮训练集的选择与前面各轮的学习结果**有关**
- bagging的各个预测函数没有权重，而boosting是有权重的；bagging的各个预测函数可以并行生成，而boosting的各个预测函数只能顺序生成。



绿色的线表示目前取得的模型（模型是由前m次得到的模型合并得到的），虚线表示当前这次模型。每次分类的时候，会更关注分错的数据，上图中，红色和蓝色的点就是数据，点越大表示权重越高，看看右下角的图片，当m=150的时候，获取的模型已经几乎能够将红色和蓝色的点区分开了。

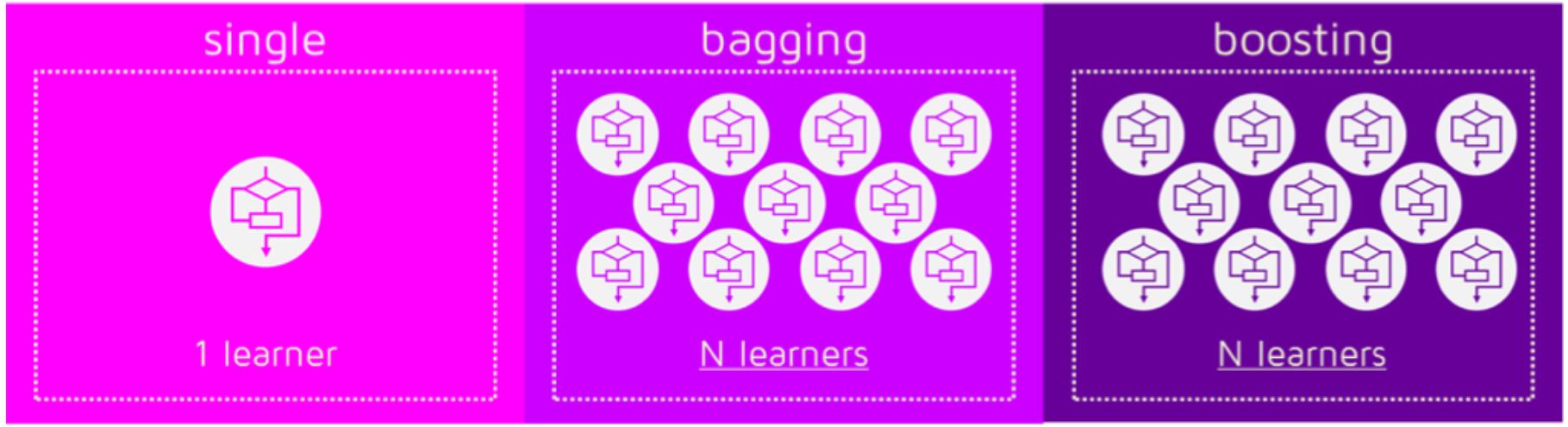
Bagging与Boosting - 2

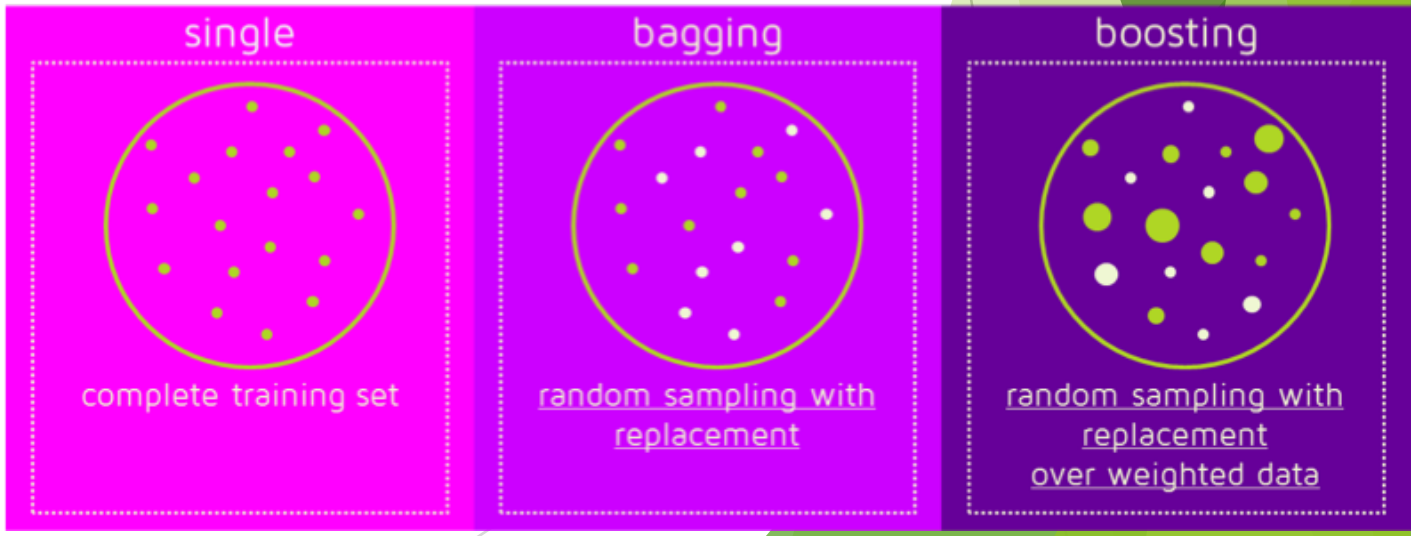
- Bagging和Boosting在细节上有一些不同，主要在于：
  - Bagging中每个训练集互不相关，也就是每个基分类器互不相关，而Bagging中预测函数是均匀平等的，但在Boosting中预测函数是加权的。对于象神经网络这样极为耗时的学习方法。bagging可通过并行训练节省大量时间开销。
  - Boosting中训练集要在上一轮的结果上进行调整，也使得其不能并行计算
- 从算法来看，Bagging关注的是多个基模型的投票组合，保证了模型的稳定，因而每一个基模型就要相对复杂一些以降低偏差（比如每一棵决策树都很深）；而Boosting采用的策略是在每一次学习中都减少上一轮的偏差，因而在保证了偏差的基础上就要将每一个基分类器简化使得方差更小。



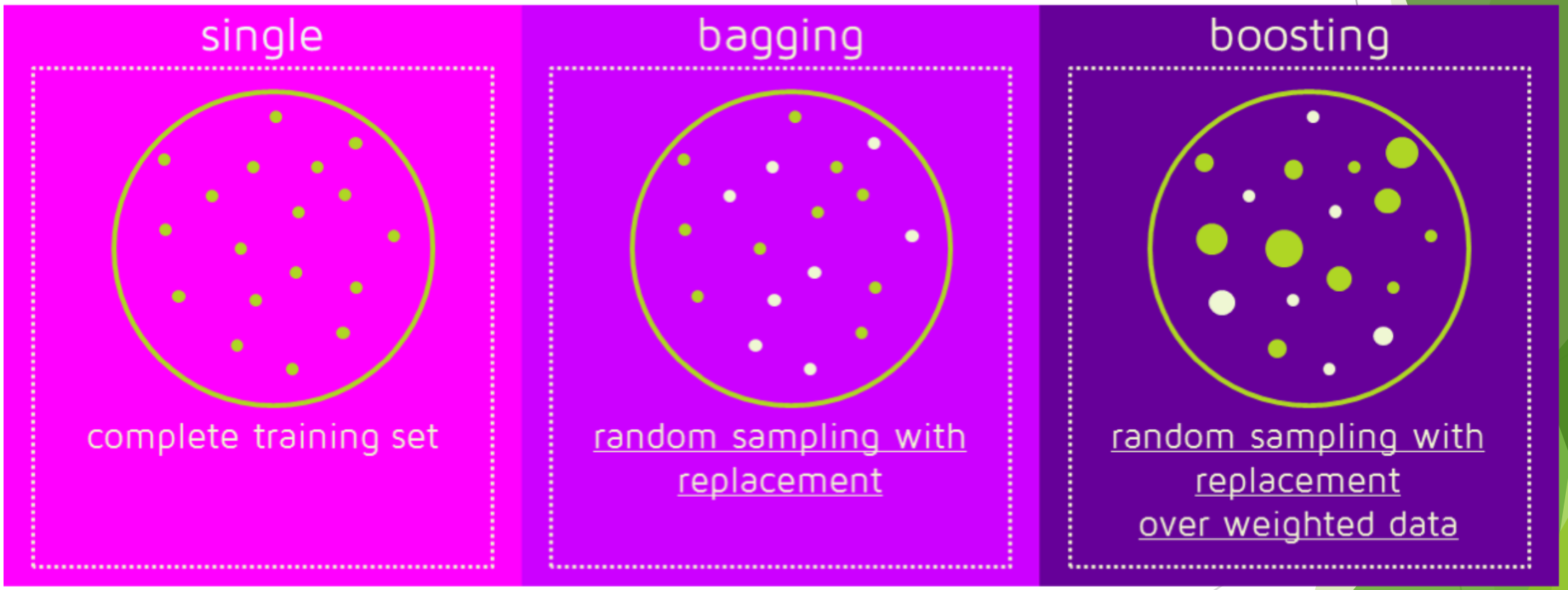
boosting算法
通过加权的方式组合成一个最终的模型$Y_M(x)$

Bagging与Boosting – 图片示意1

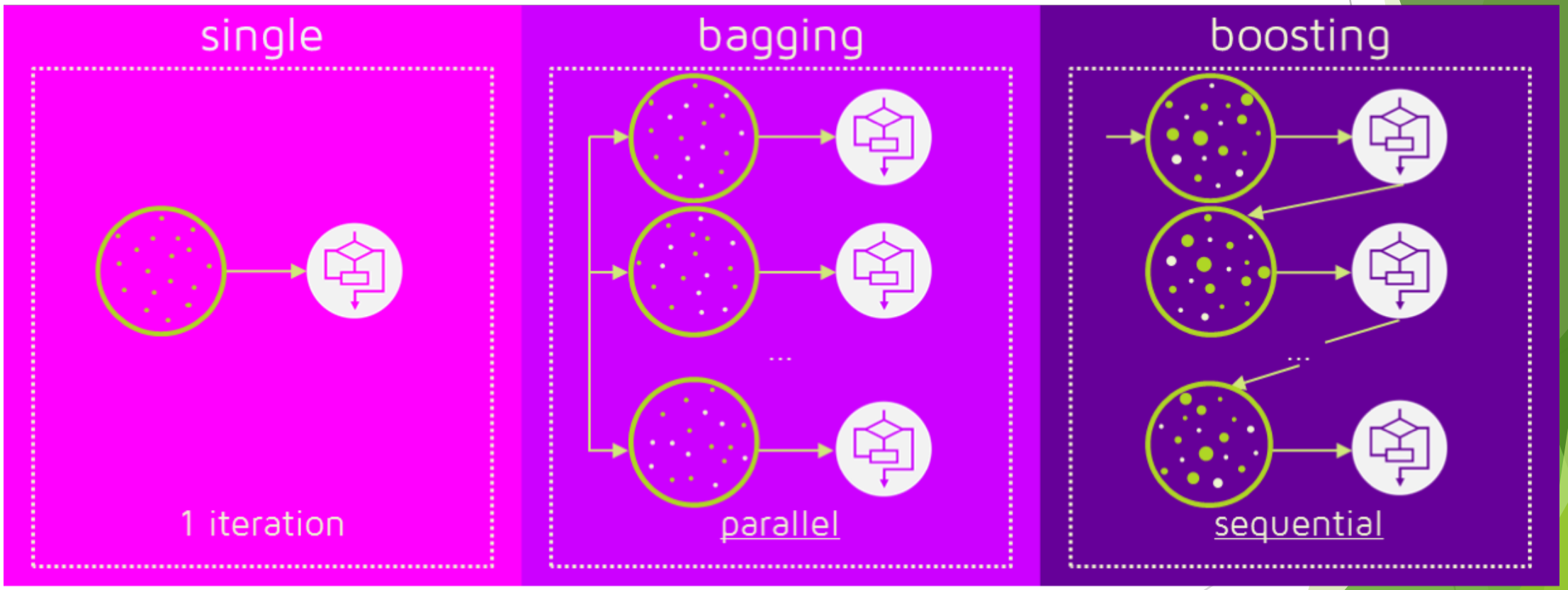




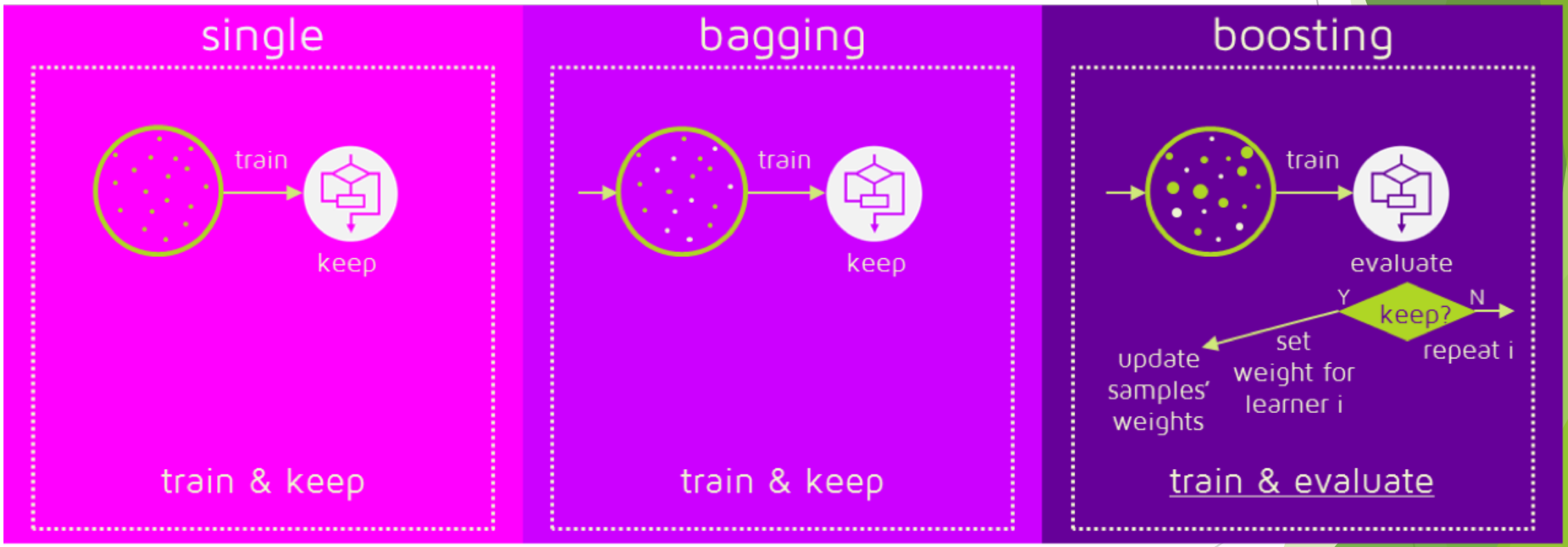
Bagging与Boosting – 图片示意1



Bagging与Boosting – 图片示意3



Bagging与Boosting – 图片示意4



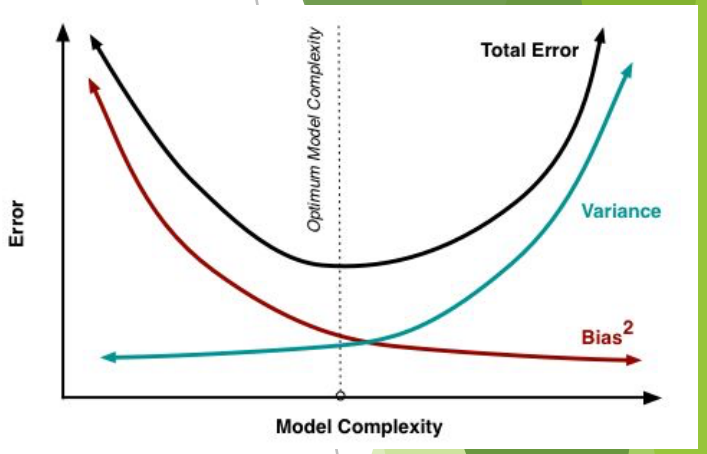
Bagging降低variance

Bagging对样本重采样，对每一重采样得到的子样本集训练一个模型，最后取平均。由于子样本集的相似性以及使用的是同种模型，因此各模型有近似相等的bias和variance（事实上，各模型的分布也近似相同，但不独立）。由于 $E[\frac{\sum X_i}{n}] = E[X_i]$，所以bagging后的bias和单个子模型的接近，一般来说不能显著降低bias。另一方面，若各子模型独立，则有 $Var(\frac{\sum X_i}{n}) = \frac{Var(X_i)}{n}$，此时可以显著降低variance。若各子模型完全相同，则 $Var(\frac{\sum X_i}{n}) = Var(X_i)$

，此时不会降低variance。bagging方法得到的各子模型是有一定相关性的，属于上面两个极端状况的中间态，因此可以一定程度降低variance。为了进一步降低variance，Random forest通过随机选取变量子集做拟合的方式de-correlated了各子模型（树），使得variance进一步降低。

（用公式可以一目了然：设有i.d.的n个随机变量，方差记为 σ^2，两两变量之间的相关性为 ρ，则 $\frac{\sum X_i}{n}$ 的方差为 $\rho * \sigma^2 + (1-\rho) * \sigma^2/n$

，bagging降低的是第二项，random forest是同时降低两项。



Bias(偏差)，Error(误差)
Variance(方差)

$$(a+b+c+d)^2 = a^2+b^2+c^2+d^2+2*(ab+ac+ad+bc+bd+cd)$$

由排列组合得出，此项个数为：$m(m-1)/2$

$$var(X) = \frac{\sum_{i=1}^{n}(X_i-\bar{X})(X_i-\bar{X})}{n-1}$$

$$cov(X,Y) = \frac{\sum_{i=1}^{n}(X_i-\bar{X})(Y_i-\bar{Y})}{n-1}$$

$$\rho = \frac{Cov(X,Y)}{\sigma_X \sigma_Y}$$

boosting减少bias

boosting从优化角度来看，是用forward-stagewise这种贪心法去最小化损失函数 $L(y, \sum_i a_i f_i(x))$ 。例如，常见的AdaBoost即等价于用这种方法最小化exponential loss：$L(y, f(x)) = exp(-yf(x))$ 。所谓forward-stagewise，就是在迭代的第n步，求解新的子模型f(x)及步长a（或者叫组合系数），来最小化 $L(y, f_{n-1}(x) + af(x))$ ，这里 $f_{n-1}(x)$ 是前n-1步得到的子模型的和。因此boosting是在sequential地最小化损失函数，其bias自然逐步下降。但由于是采取这种sequential、adaptive的策略，各子模型之间是强相关的，于是子模型之和并不能显著降低variance。所以说boosting主要还是靠降低bias来提升预测精度。

Boosting 是一种将弱分离器 $f_i(x)$ 组合起来形成强分类器 $F(x)$ 的算法框架。
一般而言，Boosting算法有三个要素[1]:

1）函数模型：Boosting的函数模型是叠加型的，即 $F(x) = \sum_{i=1}^{k} f_i(x;\theta_i)$ ；

2）目标函数：选定某种损失函数 $E\{F(x)\} = E\{\sum_{i=1}^{k} f_i(x;\theta_i)\}$ 作为优化目标；

3）优化算法：贪婪地逐步优化，即 $\theta_m^* = \arg\min_{\theta_m} E\{\sum_{i=1}^{m-1} f_i(x;\theta_i^*) + f_m(x;\theta_m)\}$ 。

将上述框架中的 $f_i(x)$ 选为决策树，将 $E\{F(x)\}$ 选为指数损失函数，就可以得到AdaBoost算法。也就是说AdaBoost是Boosting算法框架中的一种实现。当然也有别的实现，比如LogitBoost算法等。

集成算法的意义

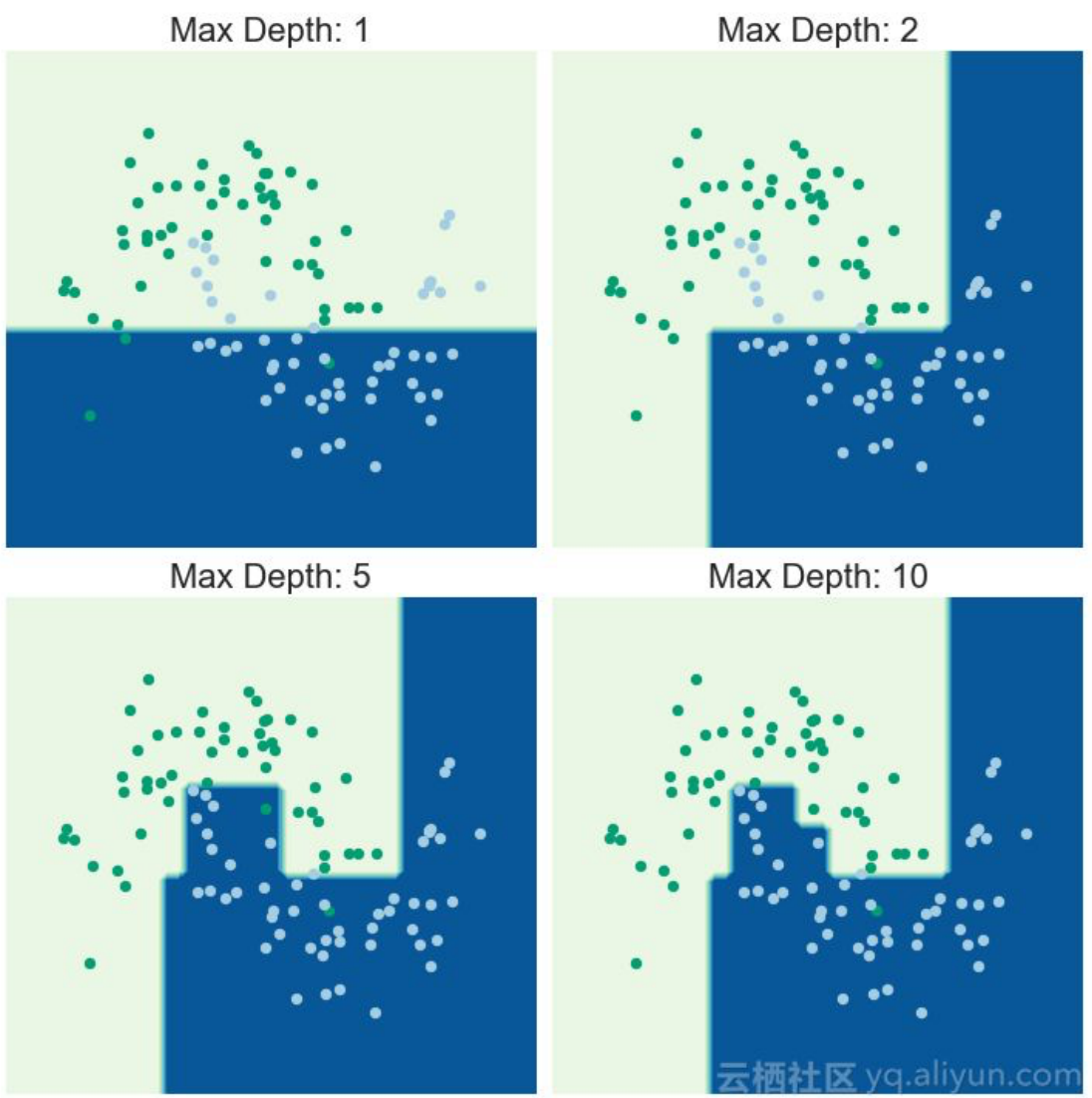
- 泛化误差的框架目的是为了让我们理解误差来源，从而改进模型来降低误差，并非给出了普遍意义的真理。
- 集成的思想还是植根于统计学，是基于对随机变量的理解而出发的。通过注入随机性而产生差异，通过平均（或者其他合并手段如加权）来降低方差，从而（期望）提升总体模型的表现能力。
- 从实际角度来看，集成学习不总能提高模型表现，有时甚至会有更差的效果。因此不管到底是降方差还是降偏差，谨慎使用才是最重要的。

决策树和随机森林

- Random Forest： 随机森林，顾名思义，是用随机的方式建立一个森林，森林里面有很多的决策树组成，随机森林的每一棵决策树之间是没有关联的。

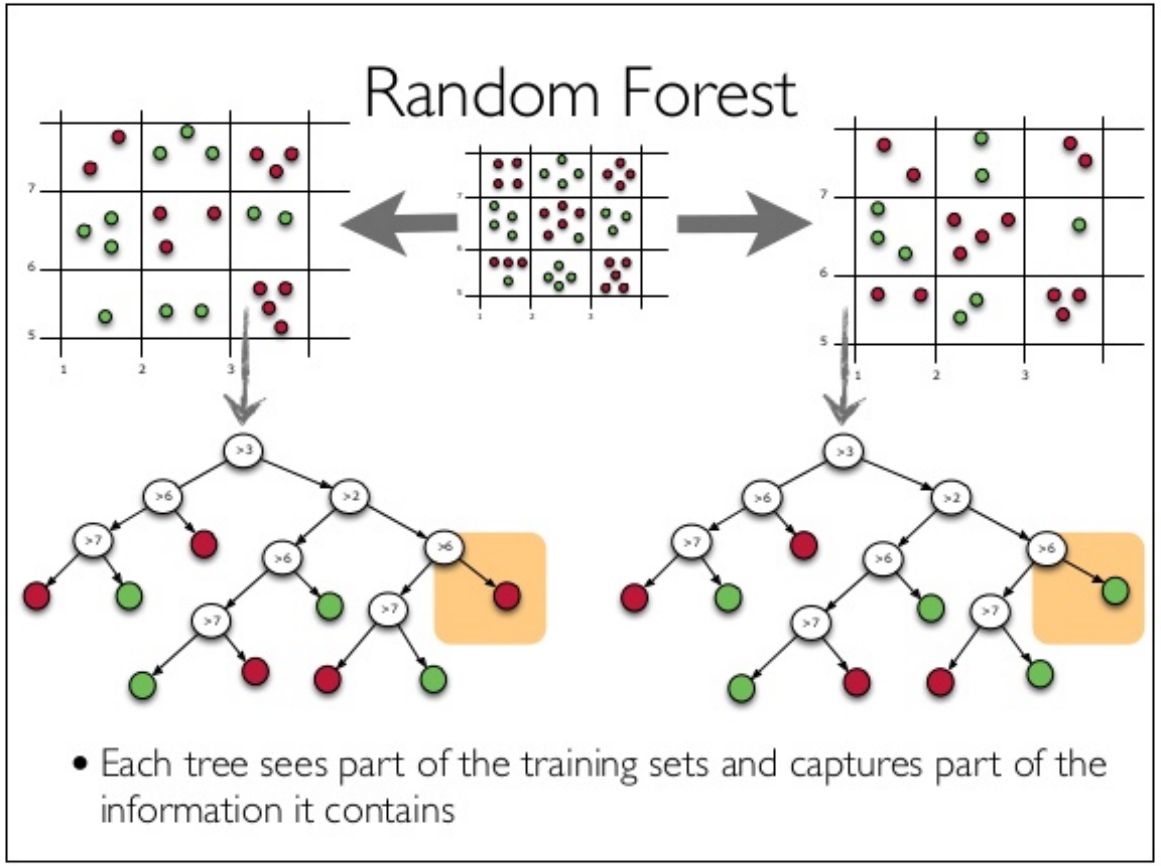
- 在建立每一棵决策树的过程中，有两点需要注意—采样与完全分裂。两个随机采样过程在random forest的主要特征：
  - 对输入的数据要进行行和列的采样
  - 然后进行列采样，从M个feature中，选择m个(m << M)

- 一般很多的决策树算法都一个重要的步骤—剪枝，但随机森林不这样做，由于之前的两个随机采样的过程保证了随机性，所以就算不剪枝，也不会出现over-fitting。 按这种算法得到的随机森林中的每一棵都是很弱的，但是大家组合起来就很厉害了。



随机森林

- 随机森林简单地来说就是用随机的方式建立一个森林，森林由很多的决策树组成，随机森林的每一棵决策树之间是没有关联的。
- 在我们学习每一棵决策树的时候就需要用到Bootstrap方法。在随机森林中，有两个随机采样的过程：对输入数据的行（数据的数量）与列（数据的特征）都进行采样。对于行采样，采用有放回的方式，若有N个数据，则采样出N个数据（可能有重复），这样在训练的时候每一棵树都不是全部的样本，相对而言不容易出现overfitting；接着进行列采样从M个feature中选择出m个（m<<M）。最近进行决策树的学习。
- 预测的时候，随机森林中的每一棵树的都对输入进行预测，最后进行投票，哪个类别多，输入样本就属于哪个类别。这就相当于前面说的，每一个分类器（每一棵树）都比较弱，但组合到一起（投票）就比较强了。



随机森林中使用决策树的优点

- 在数据集上表现良好，两个随机性的引入，使得随机森林不容易陷入过拟合
- 在当前的很多数据集上，相对其他算法有着很大的优势，两个随机性的引入，使得随机森林具有很好的抗噪声能力
- 它能够处理很高维度（feature很多）的数据，并且不用做特征选择，对数据集的适应能力强：既能处理离散型数据，也能处理连续型数据，数据集无需规范化
- 可生成一个Proximities=（pij）矩阵，用于度量样本之间的相似性： pij=aij/N, aij表示样本i和j出现在随机森林中同一个叶子结点的次数，N随机森林中树的颗数
- 在创建随机森林的时候，对generlization error使用的是无偏估计
- 训练速度快，可以得到变量重要性排序（两种：基于OOB误分率的增加量和基于分裂时的GINI下降量
- 在训练过程中，能够检测到feature间的互相影响
- 容易做成并行化方法
- 实现比较简单

无偏 → 有放回地抽样

构建随机森林的关键问题就是如何选择最优的m，要解决这个问题主要依据计算袋外错误率oob error（out-of-bag error）

随机森林使用

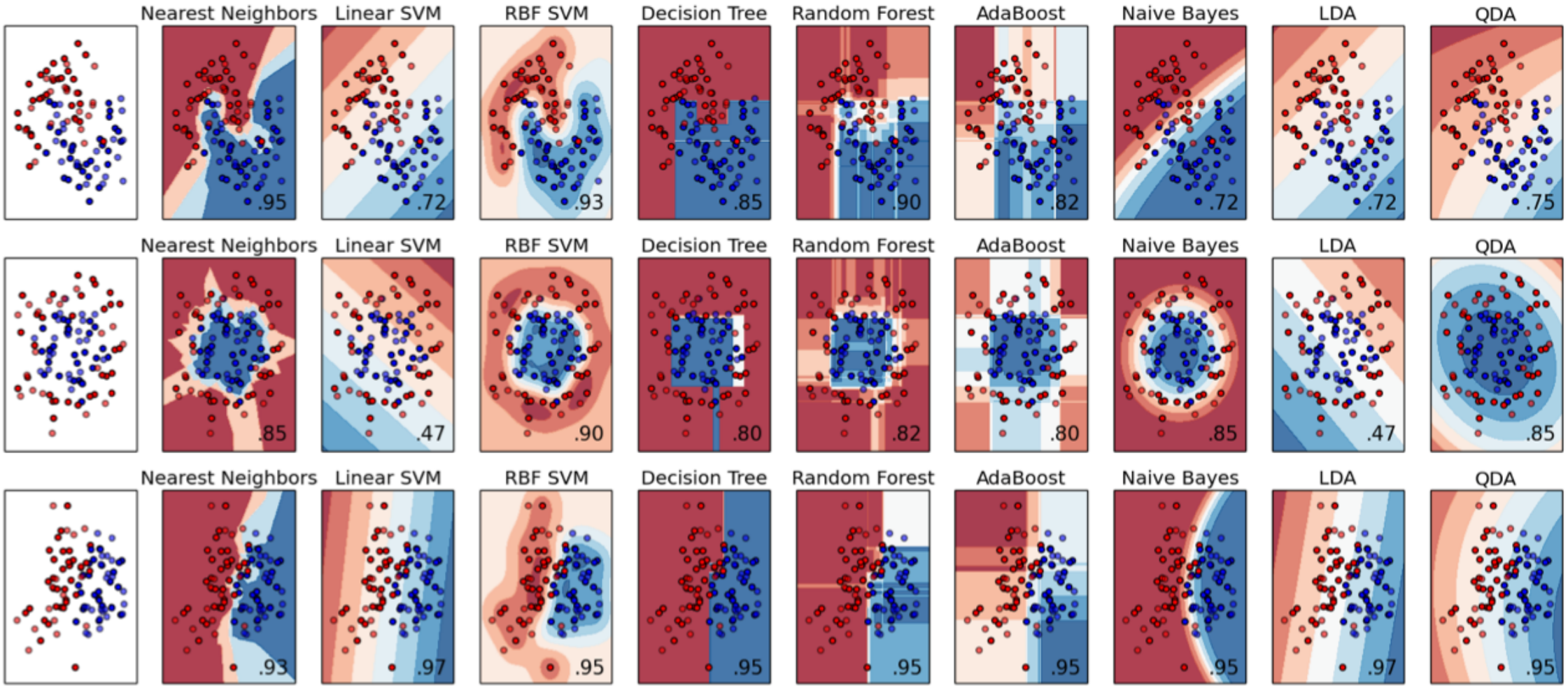
使用scikit-learn测试随机森林算法

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = RandomForestClassifier(n_estimators=10)
>>> clf = clf.fit(X, Y)
```

随机森林特点

- 优点
  - 不容易出现过拟合，因为选择训练样本的时候就不是全部样本。
  - 可以既可以处理属性为离散值的量，比如ID3算法来构造树，也可以处理属性为连续值的量，比如C4.5算法来构造树。
  - 对于高维数据集的处理能力令人兴奋，它可以处理成千上万的输入变量，并确定最重要的变量，因此被认为是一个不错的降维方法。此外，该模型能够输出变量的重要性程度，这是一个非常便利的功能。
  - 分类不平衡的情况时，随机森林能够提供平衡数据集误差的有效方法
- 缺点
  - 随机森林在解决回归问题时并没有像它在分类中表现的那么好，这是因为它并不能给出一个连续型的输出。当进行回归时，随机森林不能够作出超越训练集数据范围的预测，这可能导致在对某些还有特定噪声的数据进行建模时出现过度拟合。
  - 对于许多统计建模者来说，随机森林给人的感觉像是一个黑盒子—你几乎无法控制模型内部的运行，只能在不同的参数和随机种子之间进行尝试。

各种算法效果对比



重点对比一下决策树和随机森林对样本空间的分割：

1）从准确率上可以看出，随机森林在这三个测试集上都要优于单棵决策树，**90%>85%**，**82%>80%**，**95%=95%**；

2）从特征空间上直观地可以看出，随机森林比决策树拥有更强的分割能力（非线性拟合能力）。

Adaboost算法描述

```
模型生成
        训练数据中的每个样本，并赋予一个权重，构成权重向量D，初始值为1/N
        t次循环中的每一次:
                在训练数据上训练弱分类器并计算分类器的错误率e
                如果e等于0或者大于等于用户指定的阈值:
                        终止模型，break
                重新调整每个样本的权重，其中alpha=0.5*ln((1-e)/e)
                对权重向量D进行更新，正确分类的样本的权重降低而错误分类的样本权重值升高
                对于数据集中的每个样例:
                        如果某个样本正确分类:
                                权重改为D^(t+1)_i = D^(t)_i * e^(-a)/Sum(D)
                        如果某个样本错误分类:
                                权重改为D^(t+1)_i = D^(t)_i * e^(a)/Sum(D)
分类
        赋予所有类权重为0
        对于t（或小于t）个模型（基分类器）中的每一个:
                给模型预测的类加权 -log(e/(1-e))
        返回权重最高的类
```

Adaboost算法调用

使用scikit-learn测试adaboost算法

```
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.datasets import load_iris
>>> from sklearn.ensemble import AdaBoostClassifier
>>> iris = load_iris()
>>> clf = AdaBoostClassifier(n_estimators=100)
>>> scores = cross_val_score(clf, iris.data, iris.target)
>>> scores.mean()
0.9...
```

Adaboost特点

- 优点
  - 可以使用各种方法构造子分类器，Adaboost算法提供的是框架
  - 简单，不用做特征筛选
- 缺点
  - adaboost对于噪音数据和异常数据是十分敏感的。Boosting方法本身对噪声点异常点很敏感，因此在每次迭代时候会给噪声点较大的权重，这不是我们系统所期望的。
  - 运行速度慢，凡是涉及迭代的基本上都无法采用并行计算，Adaboost是一种"串行"算法. 所以GBDT(Gradient Boosting Decision Tree)也非常慢。

Gradient Boosting原理

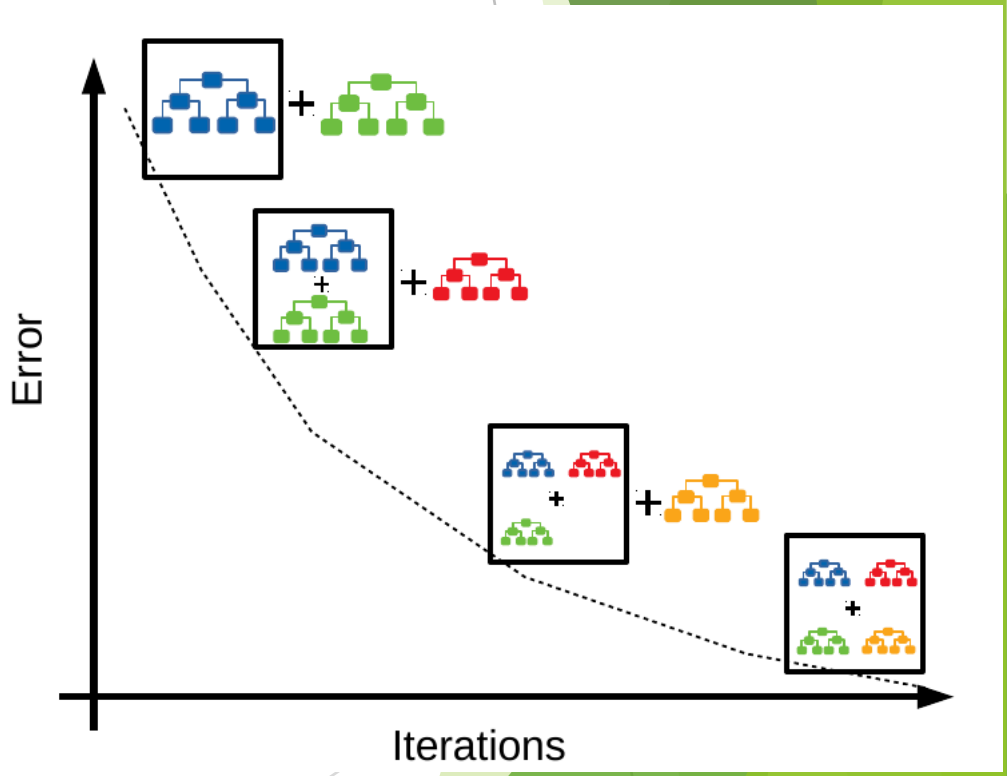
- 与其他Boosting方法一样，Gradient Boosting通过迭代将弱分类器合并成一个强分类器的方法。
- 对于标准的(x_i,y_i)训练集，Gradient Boosting在迭代到第mm次时，可能会得到一个分类能力不是很强的f_m模型，但是他下次迭代并不改变f_m，而是生成一个这样的新模型：

$$f_{m+1} = f_m + G_{m+1}$$

- 现在假如训练完G_{m+1}可以得到完美的f_{m+1}，那么也就是有：

$$G_{m+1} = y - f_m$$

- 其中$y-f_m$表示上一次迭代得到分类器预测结果与真实结果的差值，我们一般称之为的残差(residual)，因此也可以理解为Gradient Boosting在每一轮训练新的分类器将会拟合残差进行最优化



Gradient Boosting算法 - 1

假设现有训练集$\{(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)\}$，其中$x$是特征向量，$y$是相应的训练目标，在给定相应的损失函数$L(y, f(x))$,我们的目标是找到一个近似的函数$f(x)$使得与真实函数$f^*(x)$的损失最小期望值最接近:

$$f^* = \underset{f}{argmin} E_{x,y}\left[L(y, f(x))\right]$$

Gradient Boosting 方法假设y是一个实值，而$f(x)$近似目标函数是一个弱分类器$G_m(x)$加权求和的形式

$$f(x) = \sum_{m=1}^{M} \gamma_m G_m(x) + const$$

根据经验风险最小化的原则，近似函数$f(x)$将会尝试对训练集上的平均损失函数进行最小化，它从一个常量函数进行起步，并且通过贪心的方式进行逐步优化

$$f_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$$

$$f_m(x) = f_{m-1}(x) + \underset{G \in H}{argmin} \sum_{i=1}^{n} L\left(y_i, f_{m-1}(x_i) + G_m(x_i)\right)$$

Gradient Boosting算法 - 2

然而，对于L为任意的损失函数时,在选择每一步最佳的$G_m(x_i)$时将会很难优化。
这里使用最速下降法(steepest descent)来解决这个问题
,在使用这种方法时，对于损失函数$L(y, G)$不要将其看做一个函数，而是将其看做通过函数得到的值的向量$G(x_1), G(x_2) \dots G(x_n)$,那么这样的话我们就可以将模型的式子写成如下的等式:

$$f_m(x) = f_{m-1}(x) - \gamma_m \sum_{i=1}^{n} \nabla_G L\left(y_i, f_{m-1}(x_i)\right)$$

$$\gamma_m = \underset{\gamma}{argmin} \sum_{i=1}^{n} L\left(y_i, f_{m-1}(x_i) - \gamma \frac{\partial L(y_i, f_{m-1}(x_i))}{\partial G(x_i)}\right)$$

上面第一个式子表示根据梯度的负方向进行更新,第二个式子表明了γ使用线性搜索进行计算。

Gradient Boosting算法 - 3

Procedure:

1. 使用一个常量进行模型的初始化

$$f_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$$

2. 循环$m \in \{1 \ldots . M\}$

   1. 计算残差

$$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x_i)=f_{m-1}(x_i)} \qquad i = 1 \ldots n$$

   2. 使用训练集$\{(x_i, r_{im})\}$对弱分类器$G_m(x)$进行拟合
   3. 通过线性搜索进行乘子γ_m的计算

$$\gamma_m = \underset{\gamma}{argmin} \sum_{i=1}^{n} L\left(y_i, f_{m-1}(x_i) + \gamma_m G_m(x_i)\right)$$

   4. 进行模型的更新:

$$f_m(x) = f_{m-1}(x) + \gamma_m G_m(x)$$

3. 输出最终的模型$f_M(x)$

Input:

- 训练数据集$T = \{(x_1, y_1), (x_2, y_2) \ldots (x_N, y_N)\}$
- 可导的损失函数:$L(y, f(x))$
- 迭代的次数:M

Output:

- 最终模型$f(x)$

提升树(Gradient tree boosting)

- 提升树(Gradient tree boosting)故名思议就是使用决策树(一般使用CART树)来作为弱分类器.
- 提升树在第mm步迭代时将会使用决策树$G_m(x)$来集合残差，现在假设这棵树有J个叶子节点，则决策树将会将空间划分为JJ个不相交的区域$R_{1m},R_{2m}...R_{3m}d$，以及每个区域都是预测一个常量值.
- 则树模型$G_m(x)$对于特征x的输入将可以写成

$$G_m(x) = \sum_{j=1}^{J} b_{jm} I(x \in R_{jm})$$

GBDT正则化

- 调整树的个数
  - 树的个数MM越多，过拟合的情况可能越为严重，这里树的个数一般使用交叉验证的误差来调整确定
- Shrinkage(学习率)

Shrinkage 又称学习率，是指在 Gradient Boosting 训练时不训练全部的残差，而是:

$$f_m(x) = f_{m-1}(x) + v \cdot \gamma_m G_m(x) \quad 0 < v \leq 1$$

经验表明较小的学习率($v < 0.1$)将会取得较为明显的正则化效果，但是学习率太小会导致训练次数增加..

- Stochastic gradient boosting
  - 随机梯度提升法，表示每一轮迭代时并不是拿所有的数据进行训练，所以按无放回的随机取一定的比率□进行训练，这里的0.5<□<0.8将会取得较为不错的正则化效果，同时随机取样本进行训练还能加快模型的训练速度，
- 叶子节点的数量
  - 一般这个叶子节点的数量不宜太多（其实可以理解为节点数越多，模型复杂度越高）
- 使用惩罚项: 可以L2之类的惩罚项

Scikit-learn GradientBoostingClassifier调参数

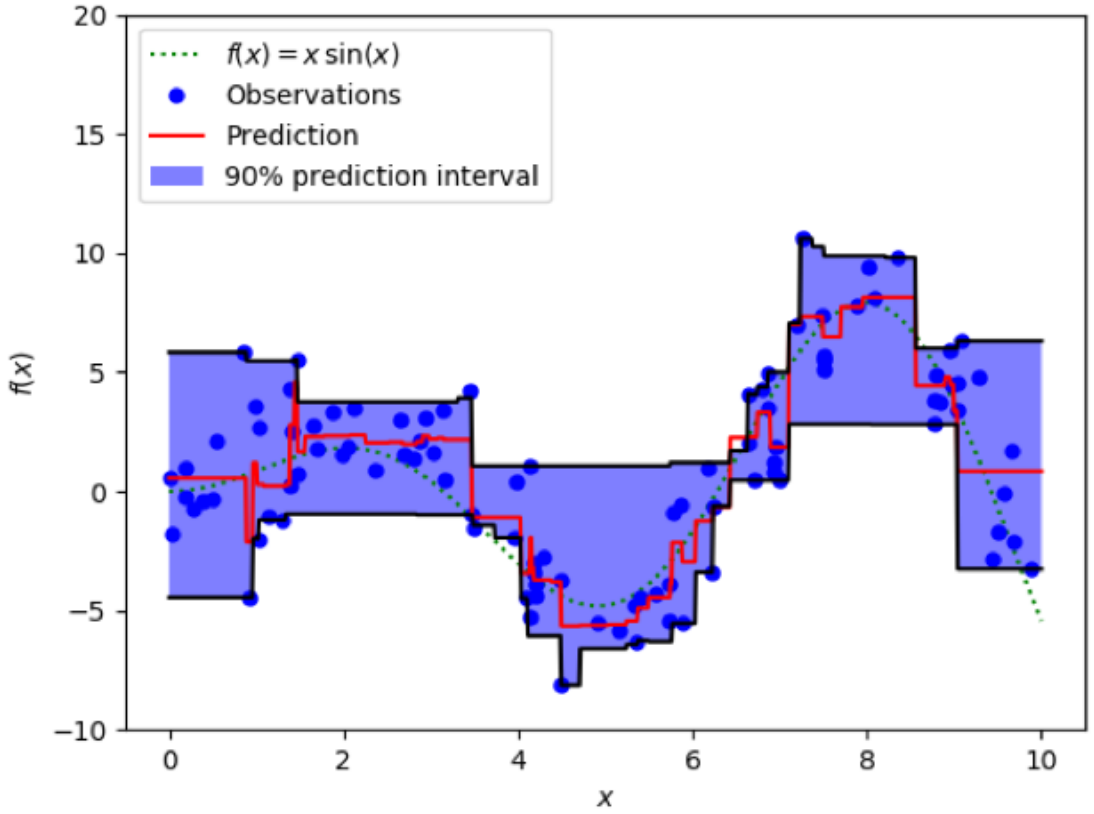
GBDT类库boosting框架参数

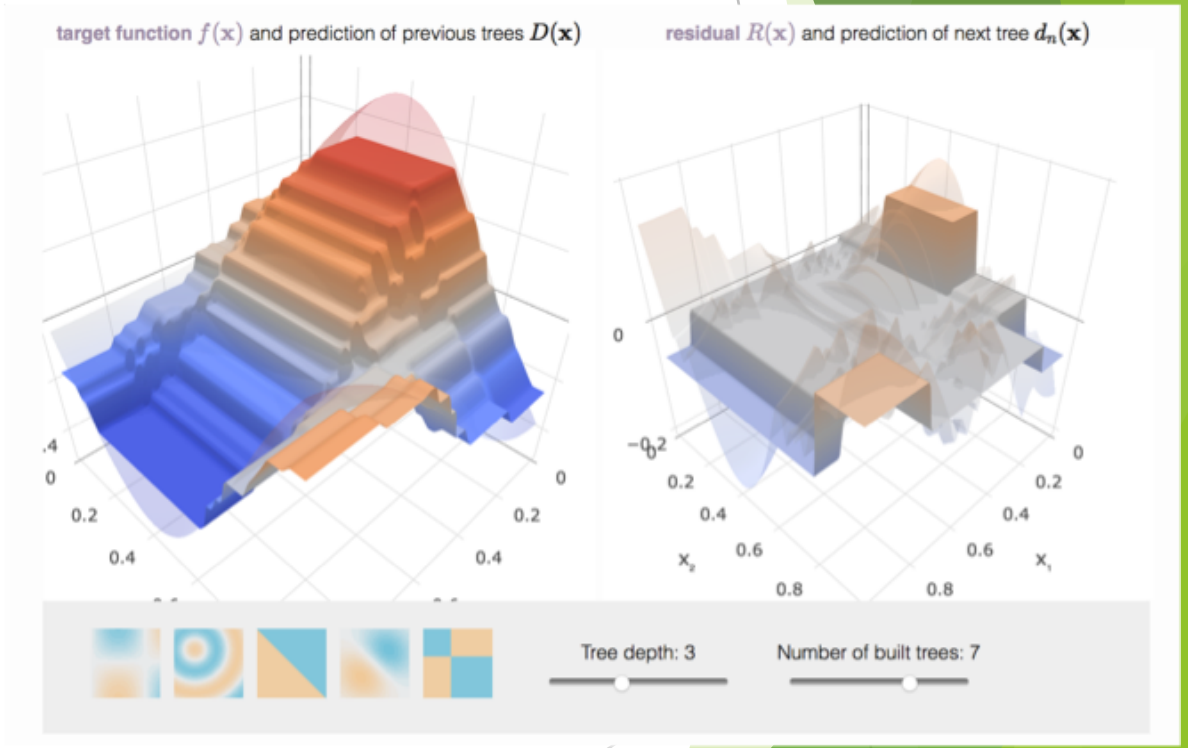
- n_estimators: 也就是弱学习器的最大迭代次数，或者说最大的弱学习器的个数。一般来说n_estimators太小，容易欠拟合，n_estimators太大，又容易过拟合，一般选择一个适中的数值。默认是100。在实际调参的过程中，我们常常将n_estimators和下面介绍的参数learning_rate一起考虑。
- learning_rate: 即每个弱学习器的权重缩减系数
- subsample: 即我们在原理篇的正则化章节讲到的子采样，取值为(0,1]。注意这里的子采样和随机森林不一样，随机森林使用的是放回抽样，而这里是不放回抽样。如果取值为1，则全部样本都使用，等于没有使用子采样。如果取值小于1，则只有一部分样本会去做GBDT的决策树拟合。

GBDT类库弱学习器参数

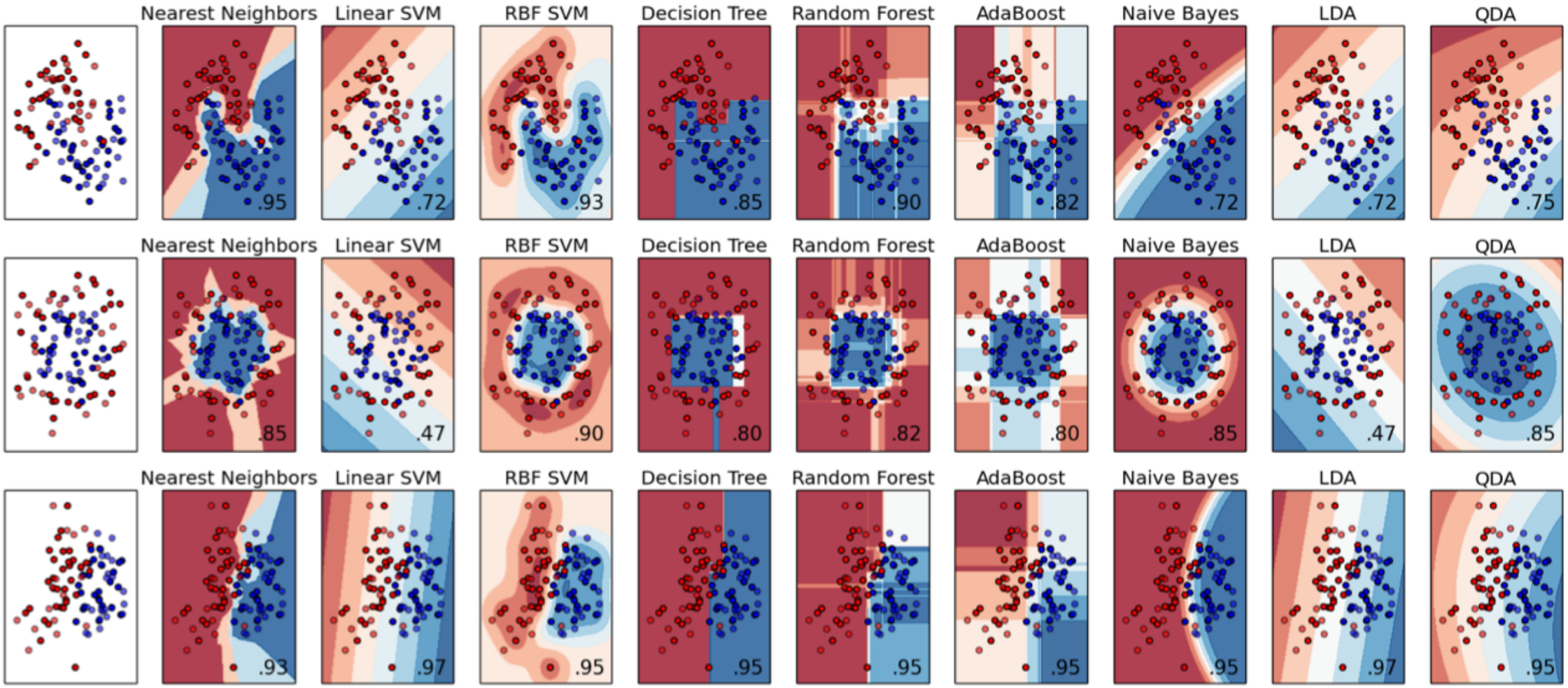
- max_features划分时考虑的最大特征数(数据类型多) 。一般来说，如果样本特征数不多，比如小于50，我们用默认的"None"就可以
- max_depth 决策树最大深度。 常用的可以取值10-100之间。
- max_leaf_nodes 最大叶子节点数。 通过限制最大叶子节点数，可以防止过拟合，默认是"None”，即不限制最大的叶子节点数。如果加了限制，算法会建立在最大叶子节点数内最优的决策树。如果特征不多，可以不考虑这个值，但是如果特征分成多的话，可以加以限制，具体的值可以通过交叉验证得到。

Gradient Boosting Regression回归





各种算法效果对比



重点对比一下决策树和随机森林对样本空间的分割：

1）从准确率上可以看出，随机森林在这三个测试集上都要优于单棵决策树，**90%>85%**，**82%>80%**，**95%=95%**；

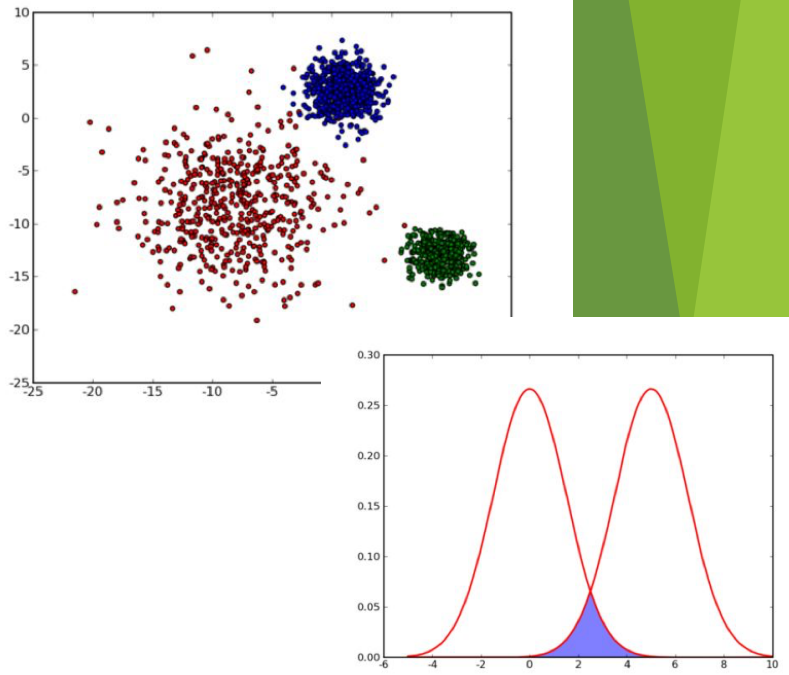
2）从特征空间上直观地可以看出，随机森林比决策树拥有更强的分割能力（非线性拟合能力）。

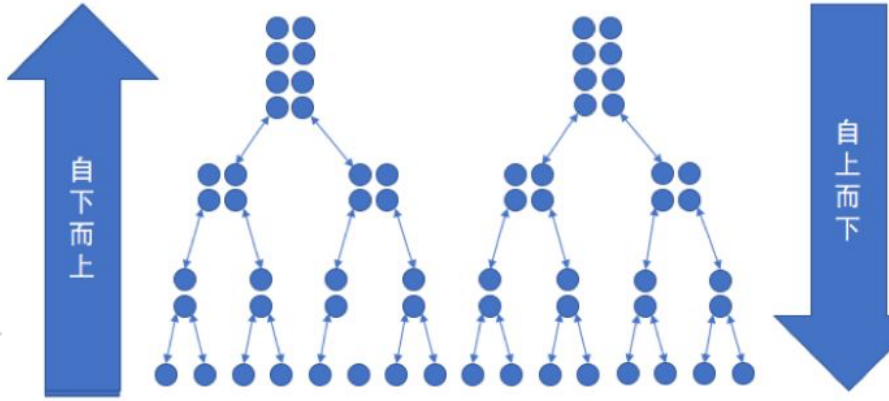
非监督学习

Guangrui Qian

无监督核心聚类算法

- **K-means**
  - 在众多聚类方法中，K-means属于最简单的一类。其大致思想就是把数据分为多个堆，每个堆就是一类。每个堆都有一个聚类中心（学习的结果就是获得这k个聚类中心），这个中心就是这个类中所有数据的均值，而这个堆中所有的点到该类的聚类中心都小于到其他类的聚类中心（分类的过程就是将未知数据对这k个聚类中心进行比较的过程，离谁近就是谁）
- **高斯混合模型**
  - GMM和K-means很相似，区别仅在于GMM中，我们采用的是概率模型P(Y|X)，也就是我们通过未知数据X可以获得Y取值的一个概率分布，我们训练后模型得到的输出不是一个具体的值，而是一系列值的概率。
  - 用GMM的优点是投影后样本点不是得到一个确定的分类标记，而是得到每个类的概率，这是一个重要信息。GMM每一步迭代的计算量比较大，大于k-means。GMM的求解办法基于EM算法，因此有可能陷入局部极值，这与初始值的选取十分相关。
- **层次聚类**
  - 把每个样本归为一类（初始化），计算每两个类之间的距离，也就是样本与样本之间的相似度；
  - 寻找各个类之间最近的两个类，把他们归为一类（这样类的总数就少了一个）；
  - 重新计算新生成的这个类与各个旧类之间的相似度；
  - 重复2和3直到所有样本点都归为一类，结束。

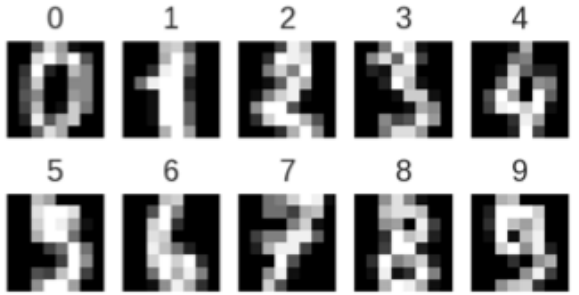




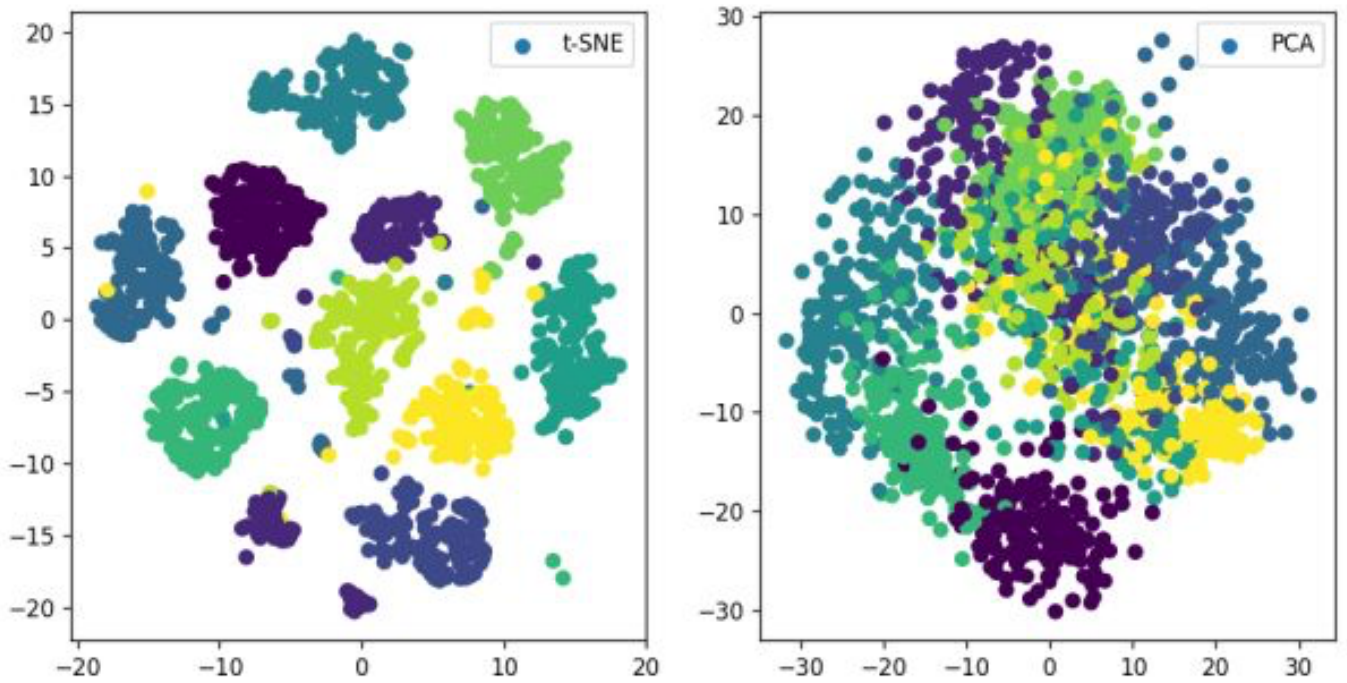
非监督学习 – t分布随机邻域嵌入

- t-SNE: t-distributed stochastic neighbor embedding
- t-SNE 是用于降维的一种机器学习算法， 是一种非线性降维算法，非常适用于高维数据降维到2维或者3维，进行可视化。相对于PCA来说，t-SNE可以说是一种更高有效的方法。
- t-SNE是一种降维算法，目的就是把X(原始高维数据)转换成Z(指定低维度的数据)。t-SNE首先将距离转换为条件概率来表达点与点之间的相似度，距离通过欧式距离算得。

9



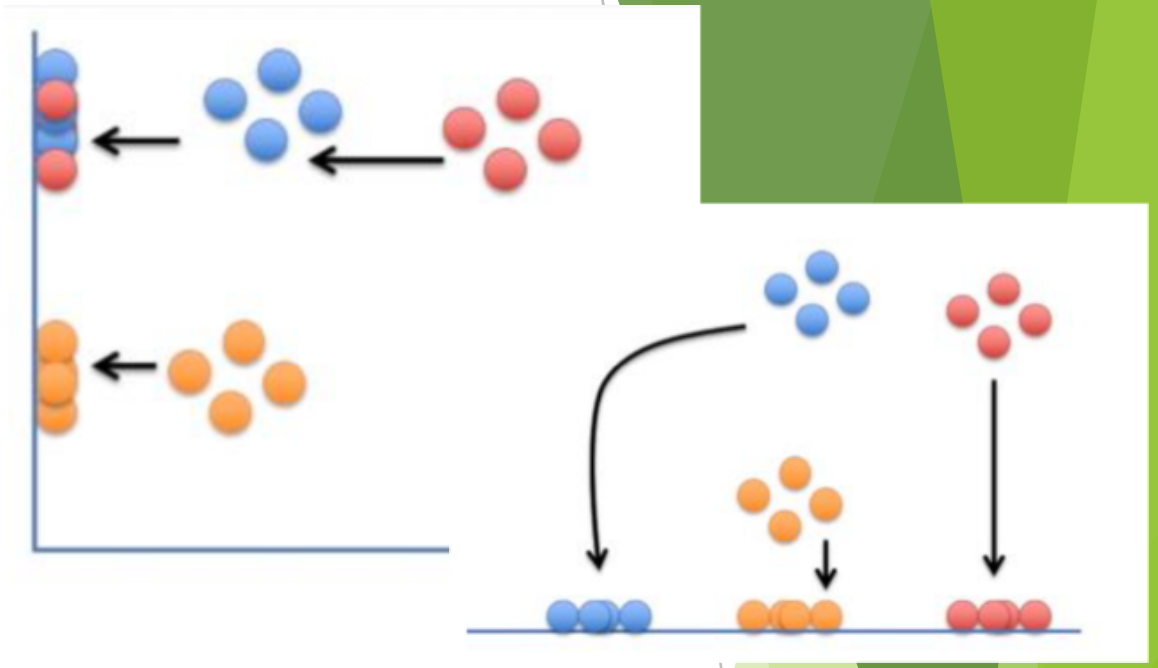
t-SNE vs PCA



从实际效果可以看出PCA降到二维后基本混到一起来，很难进行区分。而t-SNE的效果非常的不错。

t-SNE基本原理

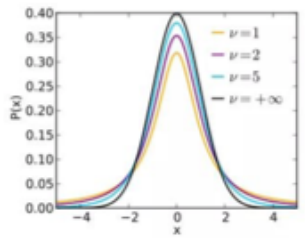
- SNE是通过仿射(affinitie)变换将数据点映射到概率分布上，主要包括两个步骤：
  - SNE构建一个高维对象之间的概率分布，使得相似的对象有更高的概率被选择，而不相似的对象有较低的概率被选择。
  - SNE在低维s空间里在构建这些点的概率分布，使得这两个概率分布之间尽可能的相似。
- 不能通过训练得到一些东西之后再用于其它数据，t-SNE只能单独的对数据做操作

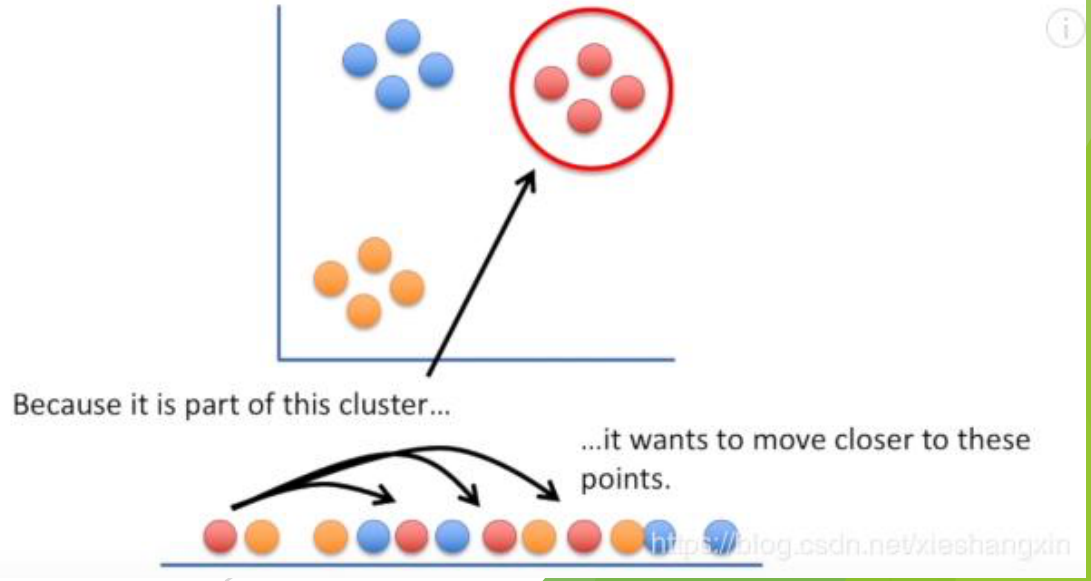


★t-分布的概率密度函数（probability density function，PDF）形式为：

$$f(t)=\frac{\Gamma(\frac{v+1}{2})}{\sqrt{v\pi}\Gamma(\frac{v}{2})}(1+\frac{t^2}{v})^{-\frac{v+1}{2}}$$

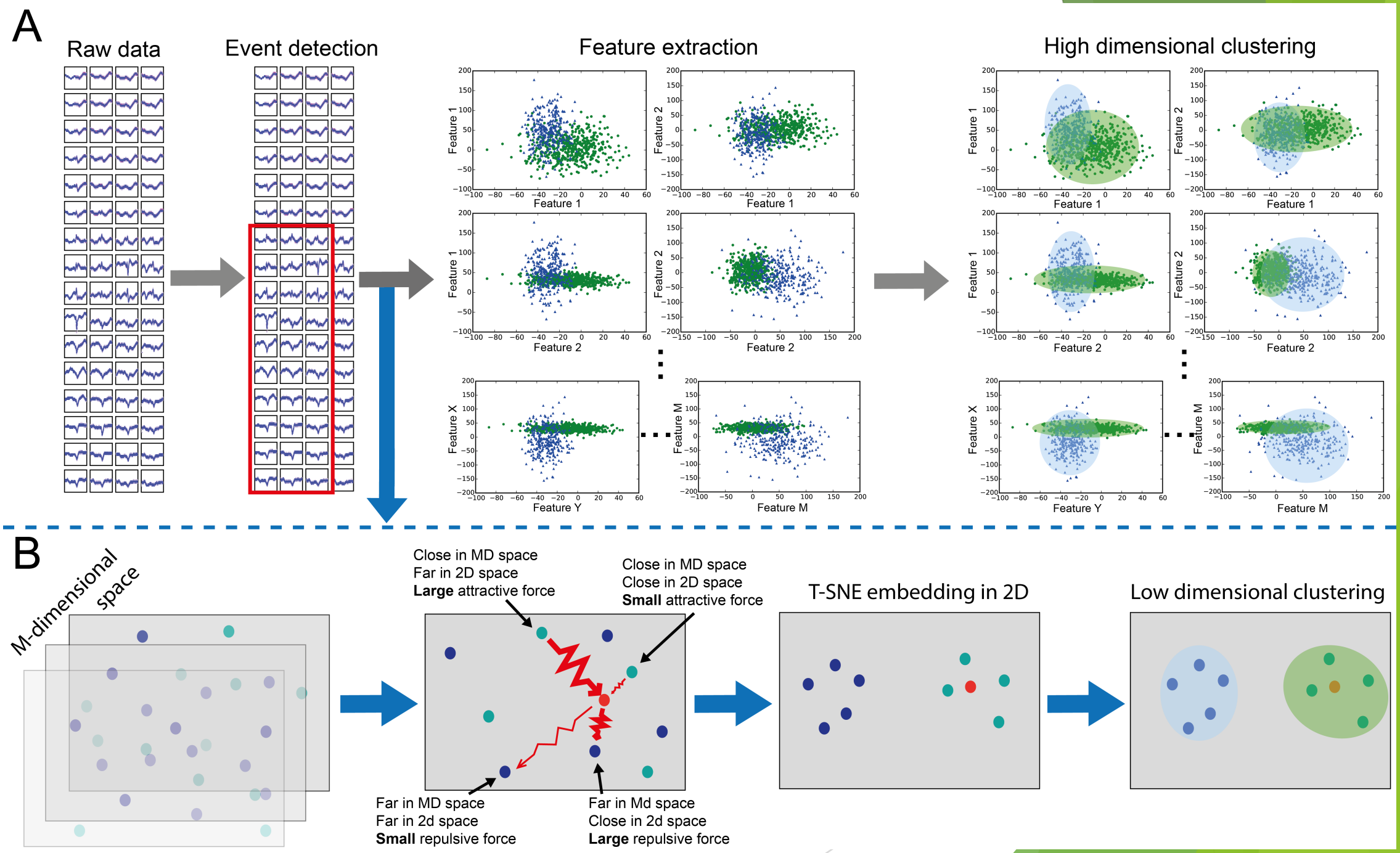
其中 v 是自由度。





https://blog.csdn.net/xieshangxin

t-SNE基本原理


A
Raw data
Event detection
Feature extraction
High dimensional clustering
Feature 1
Feature 2
Feature X
Feature Y
Feature M
B
M-dimensional space
Close in MD space
Far in 2D space
Large attractive force
Close in MD space
Close in 2D space
Small attractive force
T-SNE embedding in 2D
Low dimensional clustering
Far in MD space
Far in 2d space
Small repulsive force
Far in Md space
Close in 2d space
Large repulsive force

K邻近(KNN)分类

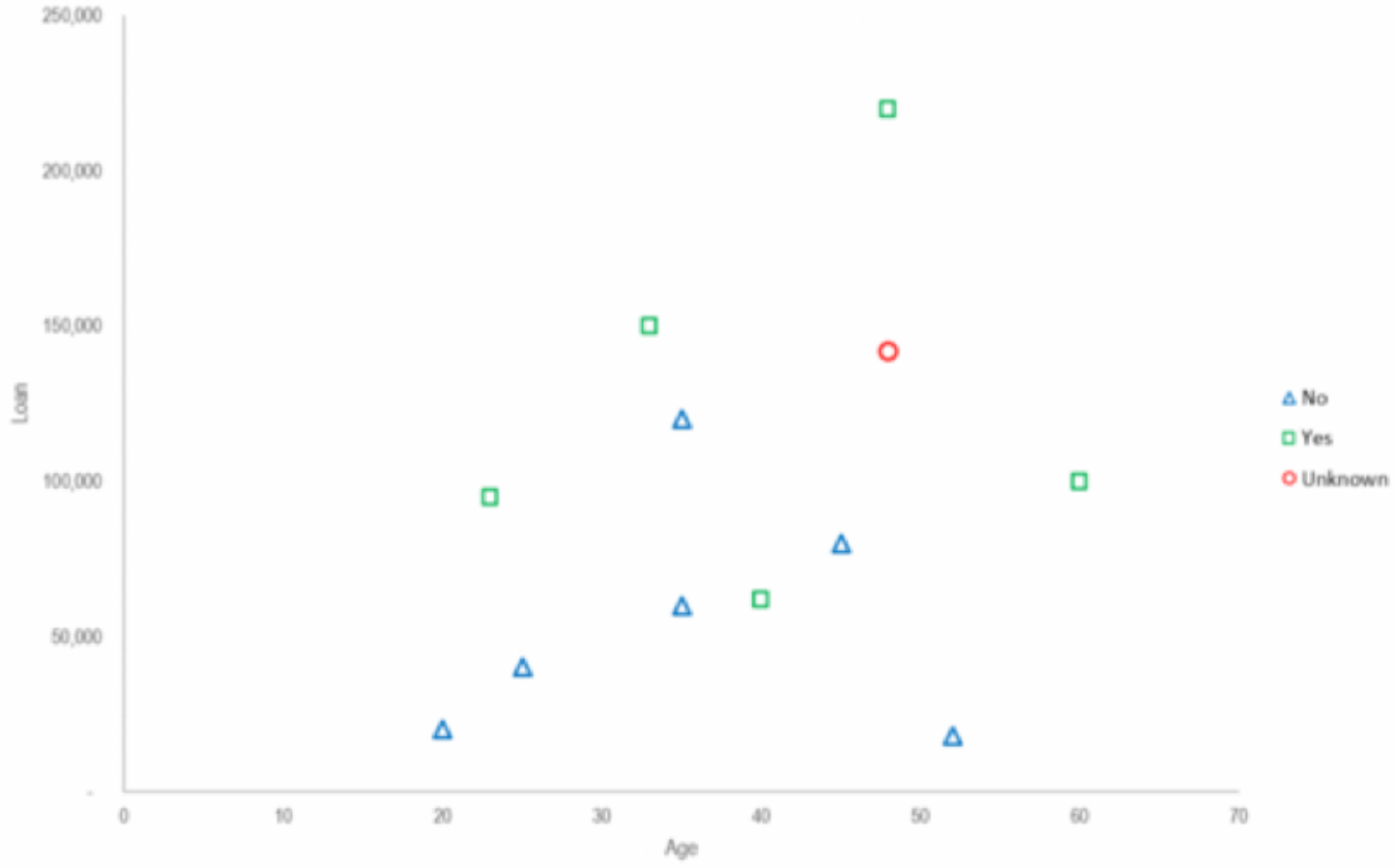
- 常用于无监督学习。
- K邻近算法(k-NearestNeighbor)简称KNN，是分类算法中的一种。KNN通过计算新数据与历史样本数据中不同类别数据点间的距离对新数据进行分类。简单来说就是通过与新数据点最邻近的K个数据点来对新数据进行分类和预测。
- KNN算法流程：
  - 计算已知类别数据集中的点与当前点之间的距离；
  - 按照距离递增次序排序；
  - 选取与当前点距离最小的k个点；
  - 确定前k个点所在类别的出现频率；
  - 返回前k个点出现频率最高的类别作为当前点的预测分类。

KNN方法实例

- 右图是一组贷款用户还款情况的样本数据，其中包含了贷款用户的年龄，贷款金额和是否还款。我们将基于这些历史的贷款样本数据，通过KNN算法对新的贷款用户进行还款预测。

Age	Loan	Default
25	40,000	N
35	60,000	N
45	80,000	N
20	20,000	N
35	120,000	N
52	18,000	N
23	95,000	Y
40	62,000	Y
60	100,000	Y
48	220,000	Y
33	150,000	Y

KNN方法实例



绿色的方块表示已经还款的用户，蓝色的三角表示未还款的用户，
红色的圆圈代表新产生的数据（48岁用户贷款142000元）。

KNN对象距离计算

- 在KNN中，通过计算对象间距离来作为各个对象之间的非相似性指标，避免了对象之间的匹配问题，在这里距离一般使用欧氏距离或曼哈顿距离：

欧式距离： $$d(x,y)=\sqrt{\sum_{k=1}^{n}(x_k-y_k)^2}$$

曼哈顿距离： $$d(x,y)=\sqrt{\sum_{k=1}^{n}|x_k-y_k|}$$

原始数据0-1标准化

Age	Loan	Default	Distance
25	40,000	N	102000
35	60,000	N	82000
45	80,000	N	62000
20	20,000	N	122000
35	120,000	N	22000
52	18,000	N	124000
23	95,000	Y	47000
40	62,000	Y	80000
60	100,000	Y	42000
48	220,000	Y	78000
33	150,000	Y	8000
48	142000	?	

← 1

数据未0-1标准化

Age	Loan	Default	Distance
0.125	0.109	N	0.76525
0.375	0.208	N	0.52001
0.625	0.307	N	0.31596
0	0.010	N	0.92454
0.375	0.505	N	0.34276
0.8	-	N	0.62195
0.075	0.381	Y	0.6669
0.5	0.218	Y	0.44367
1	0.406	Y	0.36501
0.7	1.000	Y	0.38614
0.325	0.653	Y	0.37709
0.7	0.61386	?	

← 1

数据0-1标准化

$$X_s = \frac{X - Min}{Max - Min}$$

原始数据0-1标准化

选择并调整K值

- 无论是对原始数据的距离计算和分类还是标准化后的分类。我们都是以距离最近的数据点分类来表示的新数据类别。只用一个数据点(K值)进行分类的准确性并不高，并且可能会被数据中的噪声影响而产生错误。一般情况下选择多个K值比只选择一个K值要更加准确，并且可以避免数据中噪声的干扰。最优的K值一般应该在3-10个之间。

Age	Loan	Default	Distance	
0.125	0.109	N	0.76525	
0.375	0.208	N	0.52001	
0.625	0.307	N	0.31596	1
0	0.010	N	0.92454	
0.375	0.505	N	0.34276	2
0.8	-	N	0.62195	
0.075	0.381	Y	0.6669	
0.5	0.218	Y	0.44367	6
1	0.406	Y	0.36501	3
0.7	1.000	Y	0.38614	5
0.325	0.653	Y	0.37709	4

这里我们选择了K=6，6个数据点中4个为Y，2个为N。代表还款的Y出现的概率(0.67)要更高一些，所以新的数据点应该被分类为Y。当新数据的结果发生时，我们会将结果与现在的分类和预测进行对比，以调整和优化K值的选择。

scikit-learn调用KNN

- from sklearn import neighbors

```
neighbors.KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm=
'auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n-jobs=1
)
```

谢谢