面向贝爷的数据挖掘整理

# 数据仓库

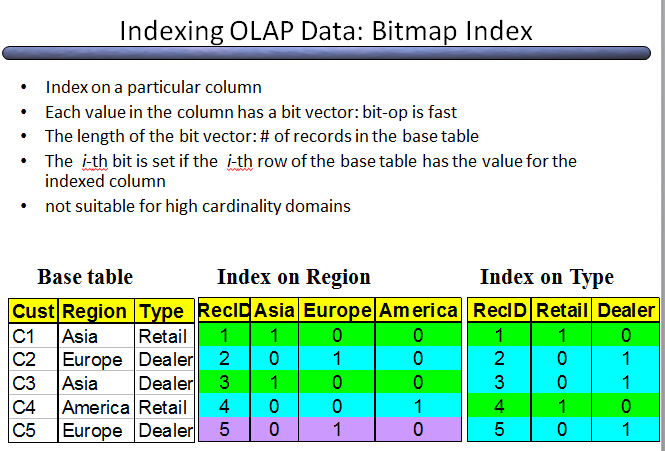
## B树索引

为什么B树索引在数据库中广泛使用，但是在数据仓库里不用？

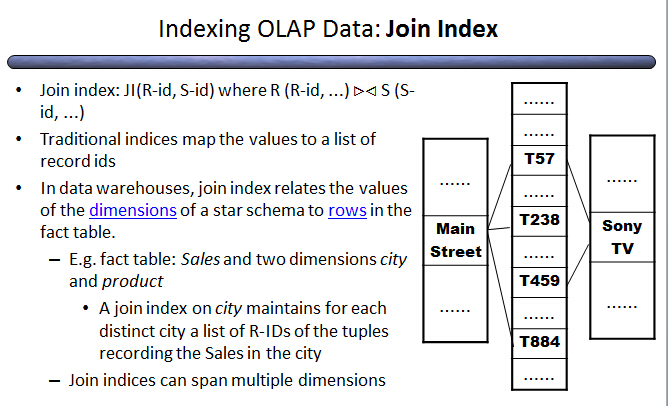
* 1. B-tree demands that the attribute must have many different values, such as itemID, customer ID, etc. （但是数据仓库里经常有如性别这样的二值字段）
  2. B-tree demands that the query should have simpler conditions and less results（但是数据仓库的查询经常要求返回一个较大的集合）
  3. The space complexity and time complexity of creating B-tree are huge（对数据仓库来说不划算）

## 位图索引

位图索引适合只有几个固定值的列，如性别、婚姻状况、行政区等等，而身份证号这种类型不适合用位图索引。



## 连接索引



# 数据预处理

## 数据清洗：针对噪声数据的分箱/分桶方法

* Binning method:
  + first sort data and partition into (equi-depth) bins（第一步，排序分箱）
  + then one can smooth by bin means, smooth by bin median, smooth by bin boundaries, etc.（第二步，用某个统计量代替，即平滑）
* Equal-width (distance) partitioning:（等距分箱）
  + It divides the range into *N* intervals of equal size: uniform grid
  + if *A* and *B* are the lowest and highest values of the attribute, the width of intervals will be: *W* = (*B*-*A*)/*N.*
  + The most straightforward
  + But outliers may dominate presentation
  + Skewed data is not handled well.
* Equal-depth (frequency) partitioning:（等比例分箱）
  + It divides the range into *N* intervals, each containing approximately same number of samples
  + Good data scaling
  + Managing categorical attributes can be tricky.

例子：

\* Sorted data for price (in dollars): 4, 8, 9, 15, 21, 21, 24, 25, 26, 28, 29, 34

\* Partition into (equi-depth) bins:

- Bin 1: 4, 8, 9, 15

- Bin 2: 21, 21, 24, 25

- Bin 3: 26, 28, 29, 34

\* Smoothing by bin means:

- Bin 1: 9, 9, 9, 9

- Bin 2: 23, 23, 23, 23

- Bin 3: 29, 29, 29, 29

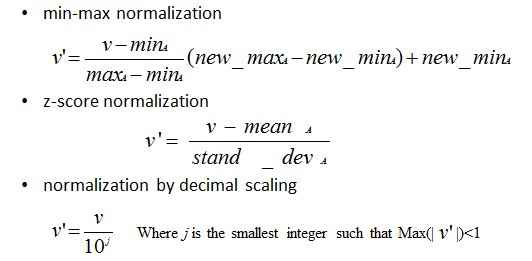
\* Smoothing by bin boundaries:

- Bin 1: 4, 4, 4, 15

- Bin 2: 21, 21, 25, 25

- Bin 3: 26, 26, 26, 34

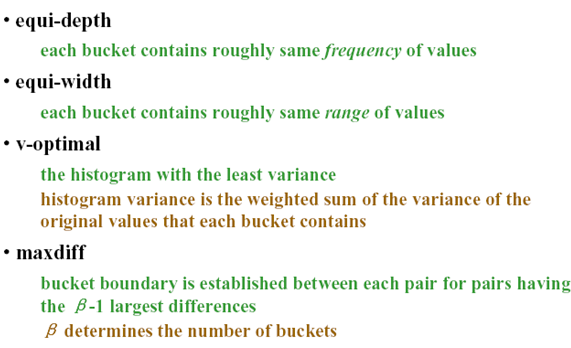
## 标准化

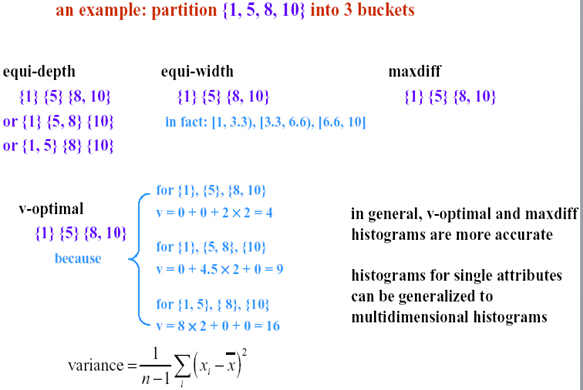


例子：

* Min-max normalization
  + An attribute: income, having values from 12000 to 98000
  + If we want to map a value 73000 to a new scope [0.0,1.0]
    - Then (73000-12000)/(98000-12000)(1.0-0)=0.716
* Z-score normalization
  + If the average of the attribute “income” is 54000, and the standard deviation is 16000
    - Then (73000-54000)/16000=1.225
* Normalization by decimal scaling
  + Given an attribute A, having values from –986 to 987, the maximum absolution is 987, so we get j=3(that is 1000)
    - -986 will be transformed to –0.986

## 离散化



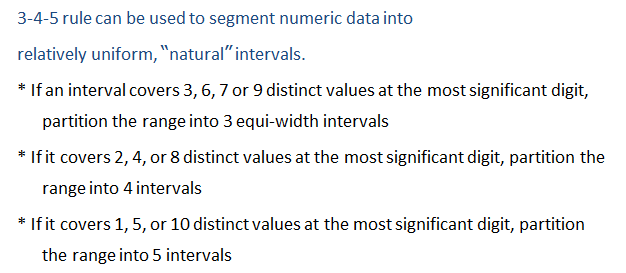


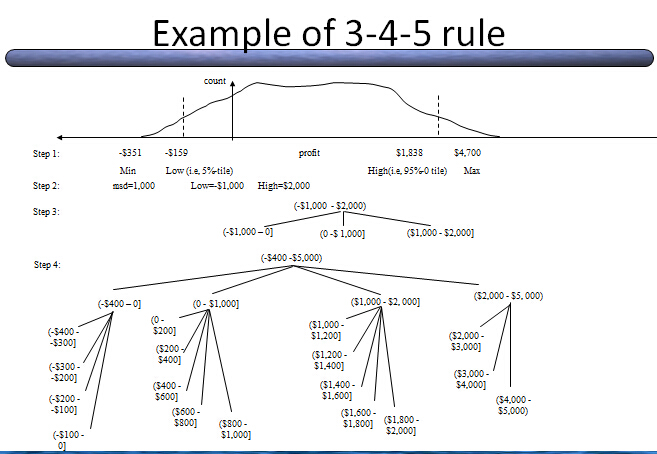
离散化的其他方法：

基于熵的离散化：

* Given a set of samples S, if S is partitioned into two intervals S1 and S2 using boundary T, the entropy after partitioning is
* The boundary that minimizes the entropy function over all possible boundaries is selected as a binary discretization.
* The process is recursively applied to partitions obtained until some stopping criterion is met, e.g.,

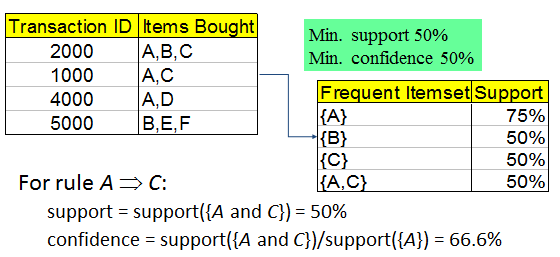
根据直观划分离散化：3-4-5规则





# 关联规则挖掘

## Apriori算法

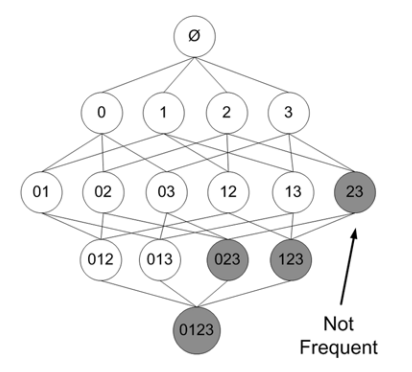


1. 关联分析的输入：一个交易（transaction）数据库，每条事务记录可以代表一次交易/或者一次购物篮的清单
2. 关联分析的两种目标：
   1. 频繁项集：指经常出现在一起的物品集合，例如{啤酒，尿布}
   2. 关联规则：形如X \Rightarrow Y的[蕴涵](http://zh.wikipedia.org/wiki/%E8%95%B4%E6%B6%B5)式，其中X \cap Y = \emptyset，例如网球拍 \Rightarrow 网球，表示买网球拍的人也很有可能会买网球

关联分析的目标就是发现这些频繁的模式。此外，关联式规则多不考虑项目的次序，而仅考虑其组合。

1. 如何定义“频繁”？
   1. 支持度(support)，只针对频繁项集定义，表示数据集中包含该记录所占的比例
   2. 置信度(confidence)，只针对关联规则定义，指包含X的事务中同时包含Y的百分比，即[条件概率](http://zh.wikipedia.org/wiki/%E6%9D%A1%E4%BB%B6%E6%A6%82%E7%8E%87)P \left ( Y | X \right )。
2. 如何发现全部的频繁项集？

首先要找到全部的项集，对于包含N种物品的数据集共有2^N-1种不同的项集，例如包含4种物品的全部项集：



然后对每个候选项集，遍历交易数据库，找出满足某种支持度的项集。

显然这种做法复杂度很高。

1. Apriori原理（针对频繁项集挖掘）

如果一个集合是频繁项集，则它的所有子集都是频繁项集。

或者用逆反命题可以说*：*

如果一个集合不是频繁项集，则它的所有超集都不是频繁项集。

后面举例来讲就是：假设集合{A}不是频繁项集，即A出现的次数小于min\_support，则它的任何超集如{A,B}出现的次数必定小于min\_support，因此其超集必定也不是频繁项集。后面一种说法更加有用，因为可以利用它来抛掉很多候选项集。

1. 使用Apriori算法发现频繁项集

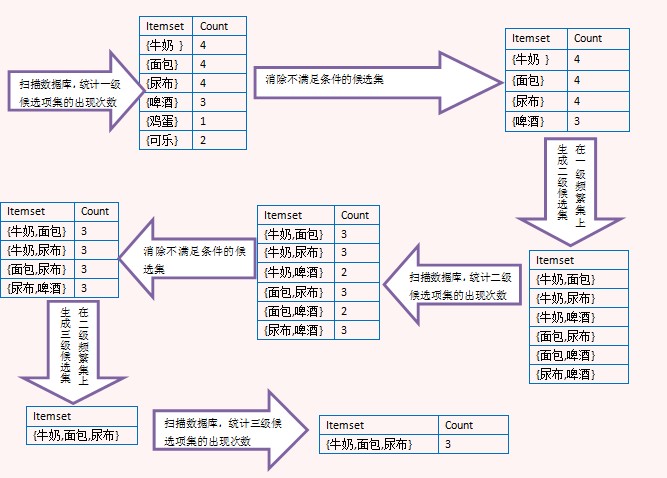
Apriori在拉丁语的意思是“来自以前”，顾名思义，这个算法每一轮的运行结果依赖于上一轮的输出。

Apriori算法的输入参数：最小支持度和数据集。

Apriori算法的执行过程：

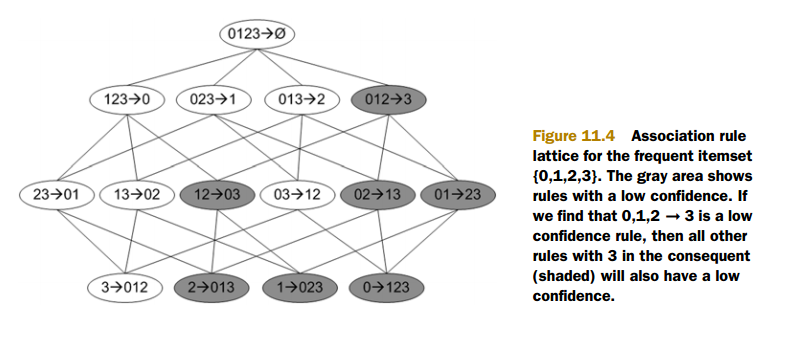
1. 首先生成所有单个物品的项集列表
2. 扫描交易数据库，查看哪些项集满足最小支持度的要求，去掉不满足最小支持度的集合
3. 对生下来的一级频繁项集进行组合以生成具有两个元素的所有候选项集
4. 在此扫描交易数据库，去掉不满足最小支持度的项集
5. …..
6. 过程重复到N为止（或者无法产生候选项集）

图示说明如下：



1. 从频繁项集产生关联规则

先看从一个频繁项集中可以产生的所有关联规则，从项集{0,1,2,3}中产生的所有关联规则如下，注意规则自上而下的生成过程：



同样这里也要考虑计算次序和筛选的问题，来减少规则数目，确保问题的可解性。

这里的Apriori原理：如果某条规则并不满足最小置信度要求，那么该规则的所有子集也就不会满足最小置信度要求。

假设最小置信度为p，且规则0,1,2🡪3并不满足最小置信度要求，即

P(0,1,2,3)/P(0,1,2)<p

那么任何左部为{0,1,2}的子集的规则也不会满足最小置信度要求，例如考虑规则1,2🡪0,3，其置信度为

P(0,1,2,3)/P(1,2) <= P(0,1,2,3)/P(0,1,2) < p

由此确定关联规则的生成算法：（输入参数：数据集和一个频繁项集）

1. 创建第一个规则列表，其中规则右部只包含一个元素
2. 对这些规则进行测试（依次计算置信度，用到的支持度数据在生成频繁项集的时候都存下来了）
3. 合并所有第一个列表中的剩余规则，创建第二个规则列表，其中规则右部包含两个元素
4. 对第二个列表中的规则进行测试
5. ……
6. 过程重复到N为止（或者无法产生新规则）

## FP-growth算法

 FP-Growth(频繁模式增长)算法是韩家炜老师在2000年提出的关联分析算法，它采取如下分治策略：将提供频繁项集的数据库压缩到一棵频繁模式树（FP-Tree），但仍保留项集关联信息；该算法和Apriori算法最大的不同有两点：第一，不产生候选集，第二，只需要两次遍历数据库，大大提高了效率。

***但是FP-Growth算法只能用来发现频繁项集，不能用来发现关联规则。***

### 算法伪代码

算法：**FP-**增长。使用FP-树，通过模式段增长，挖掘频繁模式。

**输入**：事务数据库D；最小支持度阈值min\_sup。

**输出**：频繁模式的完全集。

1． 按以下步骤构造FP-树：

(a) 扫描事务数据库D一次。收集频繁项的集合F和它们的支持度。对F按支持度降序排

序，结果为频繁项表L。

(b) 创建FP-树的根结点，以“null”标记它。对于D 中每个事务Trans，执行：

选择 Trans 中的频繁项，并按L 中的次序排序。设排序后的频繁项表为[p | P]，其

中，p 是第一个元素，而P 是剩余元素的表。调用insert\_tree([p | P], T)。该过程执行

情况如下。如果T 有子女N 使得N.item-name = p.item-name，则N 的计数增加1；否则创建一个新结点N，将其计数设置为1，链接到它的父结点T，并且通过结点链结构

将其链接到具有相同item-name 的结点。如果P 非空，递归地调用insert\_tree(P, N)。

2． FP-树的挖掘通过调用**FP\_growth**(FP\_tree, null)实现。该过程实现如下：

**procedure** **FP\_growth**(*Tree*, *a*)

**if** *Tree* 含单个路径*P***then{**

**for**路径*P*中结点的每个组合（记作*b*）**//这里non-trivial！！**

         产生模式*b* U *a*，其支持度*support* = *b*中结点的最小支持度；

**} else {**

**for each***a i* 在*Tree*的头部(按照支持度由低到高顺序进行扫描){

                  产生一个模式*b* = *a i* U *a*，其支持度*support*= *a i*.*support*；

                  构造*b*的条件模式基，然后构造*b*的条件FP-树*Tree*b；

**if***Tree*b 不为空 then

                            调用 FP\_growth (*Tree*b, *b*)；

**}**

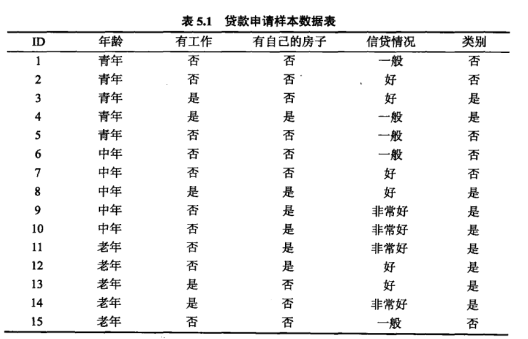
**}**

最详细的讲解：<http://blog.csdn.net/sealyao/article/details/6460578>

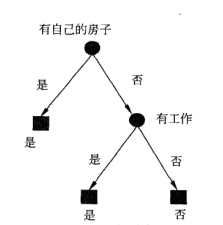
# 分类算法

## 决策树

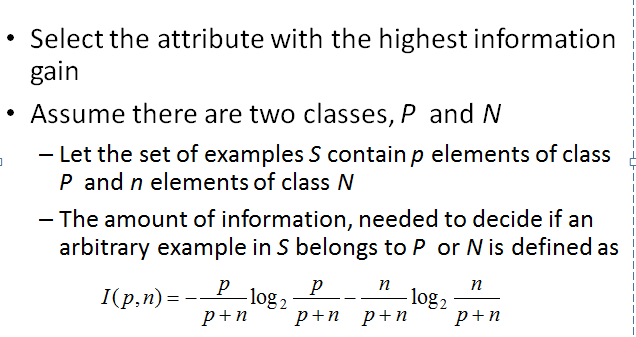
决策树的输入（也是分类问题的输入）：

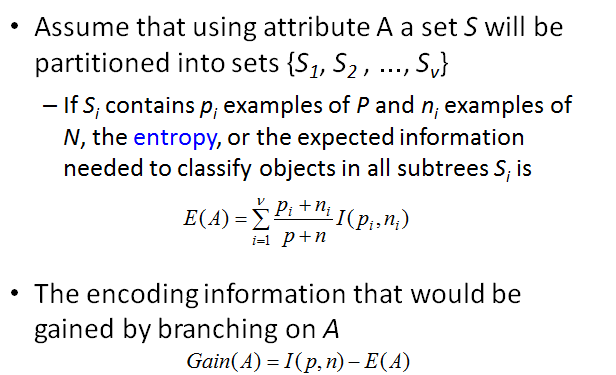


决策树的输出：

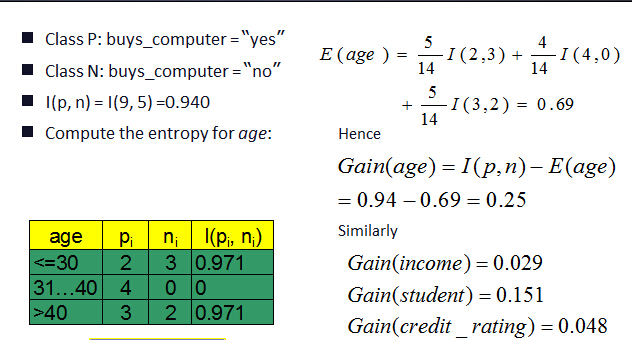


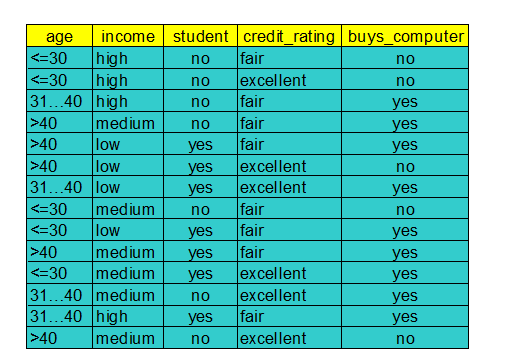
信息增益的计算（ID3算法）：



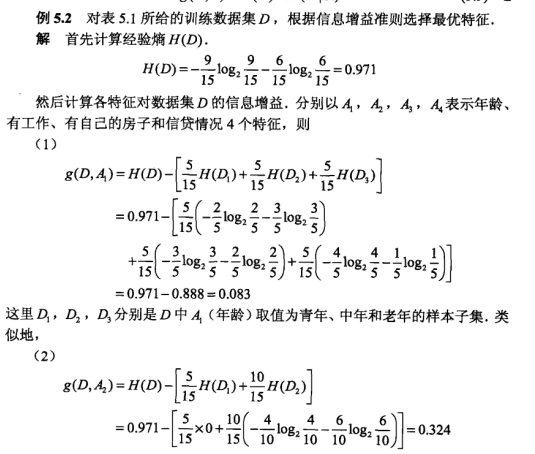


例子：





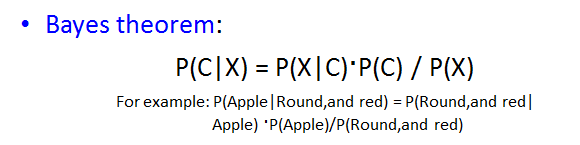
另一个例子：



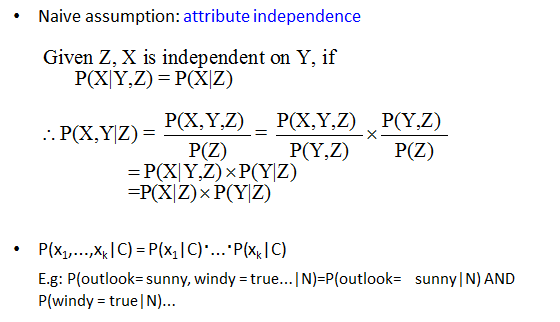
多分类的决策树

## 朴素贝叶斯

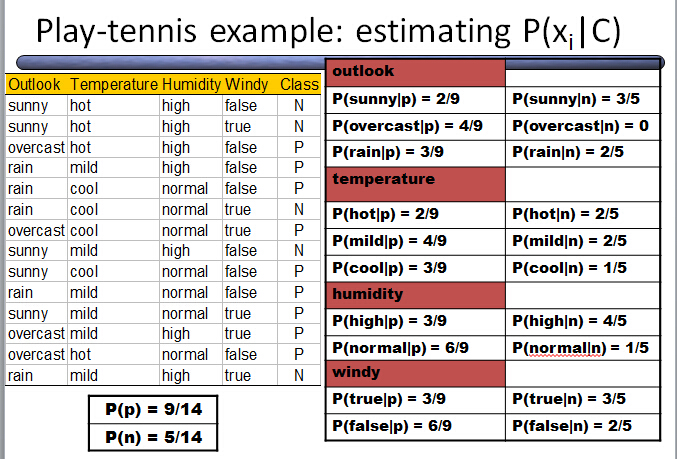
贝叶斯公式：

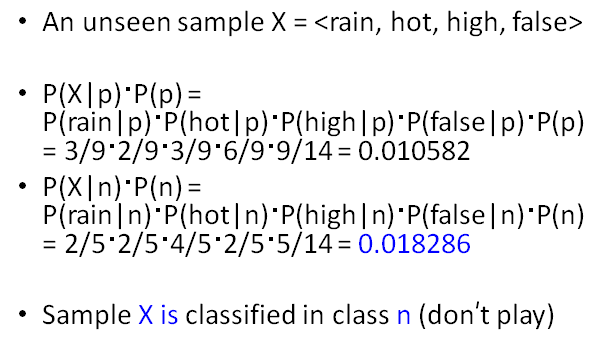


朴素贝叶斯假设：



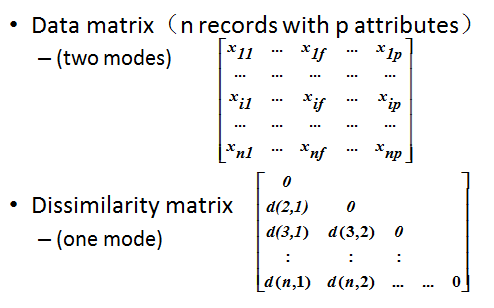
例子：





# 聚类算法

## 输入



## 按距离聚类

## K-means聚类