1、设总体X~U(a,b),a,b未知，x1,x2,……,xk是来自X的样本值, 求a,b的最大似然估计量。

x_i都服从如下概率密度函数

$f(x)=\frac{1}{b-a}(a\le x\le b)$,在其他区间为0

似然函数$L=\left(\frac{1}{b-a}\right)^k$，$\ln L=-k\ln\left(\frac{1}{b-a}\right)$,

对$\ln L$求偏导$\frac{\partial \ln L}{\partial a}=\frac{k}{b-a}$　　$\frac{\partial \ln L}{\partial b}=\frac{-k}{b-a}$,两个偏导都无法为0，因此开始分析$L$

要使L最大，须$b-a$最小，则b取最小值，a取最大值，

x_i都服从均匀分布$U(a,b)$,故，a最大取$x_{\min}$,b最小取$x_{\max}$

综上，$\hat{a}=x_{\min}$　　$\hat{b}=x_{\max}$

2、一个框子里装有4个西红柿（蔬菜、红色）、5个茄子（蔬菜、紫色）、7个苹果（水果、红色）和5个梨（水果、黄色），采用决策树进行分类。如果用第一个特性（水果/蔬菜）来分类，则分类后的信息增益是多少？如果采用颜色进行分来，则分类后的信息增益是多少？

原集合每种东西的个数为[4,5,7,5],表示4个西红柿、5个茄子、7个苹果和5个梨组成的集合

根据信息熵的定义，其信息熵是1.9699000226227195

若按水果/蔬菜分类

分成两个子集，[7,5]和[4,5]

两个子集的基数、信息熵分别是

12　　0.9798687566511528

9　　0.9910760598382222

$$\sum_{v\in T}\frac{|P_v|}{|P|}\cdot Entropy(P_v)$$

由此公式，加权熵值为

$$\frac{12}{21}\times 0.9798687566511528+\frac{9}{21}\times 0.9910760598382222$$

值为0.5599250038006587+0.4247468827878095=0.9846718865884682

信息增量为1.9699000226227195-0.9846718865884682=**0.9852281360342513**

若按颜色来分

则分三个子集，个数分别是[4,7],[5],[5]

由信息熵的定义，两个个数为5的集合，即5个茄子组成的集合，5个梨组成的集合，是确定的，信息熵为0

[4,7]集合的基数是11，熵为0.9456603046006402

故信息熵变为 $\frac{11}{21}\times 0.9456603046006402$，

值为0.4953458735238095

信息增量为1.9699000226227195-0.495345873523809=**1.4745541490989105**

3、编程实现4.5和4.6小节中的入侵检测例题。

4.5节

例 4-*（**入侵检测**）通过计算机的 5 个外在特征来判断计算机是否受到入侵。给定表 3 的入侵检测数据集，建立朴素贝叶斯模型，判断以下计算机的外部特征 **X** 和 **Y** 是否为入侵：

入侵/正常	时延	响应	流量	行为	内存
X	中	慢	异常	正常	不变
Y	高	快	正常	异常	不变

表 3 入侵检测数据集

入侵/正常	时延	响应	流量	行为	内存
正常	高	快	异常	正常	不变
正常	中	快	正常	异常	不变
正常	低	中	未知	正常	增大
正常	高	慢	正常	正常	减小
正常	中	中	未知	正常	增大
正常	低	快	正常	异常	不变
入侵	高	中	异常	正常	增大
入侵	中	慢	正常	异常	增大

入侵	中	慢	正常	异常	不变
入侵	高	慢	异常	正常	不变
入侵	低	中	未知	异常	增大

对于计算机的外部特征 **X**，根据朴素贝叶斯推理，它属于**正常**的概率为：

既然要编程实现，那就不导入python的现成的包，

核心计算公式：$C_{NB}(x) = \arg\max_{c \in Y} \quad P(c) \prod_{i=1}^{d} P(x_i|c)$

程序中概率均采用拉普拉斯修正后的概率 $P(x_i|c) = \frac{|D_{c,x_i}| + 1}{|D_c| + N_i}$

将属性，标签都数值化，按例题中给出的数据：

```
# 网络时延分为:高、中、低 -> 1, 2, 3
# 响应速度分为:快、中、慢 -> 1, 2, 3
# 流量异常分为:异常、正常、未知 -> 1, 2, 3
# 行为异常分为:异常、正常 -> 1, 2
# 存储增大分为:增大、减小、不变 -> 1, 2, 3
X = np.array([[1,1,1,2,3],
[2,1,2,1,3],
[3,2,3,2,1],
[1,3,2,2,2],
[2,2,3,2,1],
[3,1,2,1,3],
[1,2,1,2,1],
[2,3,2,1,1],
[2,3,2,1,3],
[2,3,1,2,3],
[3,2,3,1,1]]);
# 标签：1:正常， 2:入侵
Y = np.array([1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2])
```

仿照GaussianNB ，自己设计了一个类

```
class My_Module:
    def __init__(self, X, Y):#样本
        self.x = X#属性矩阵
        self.y = Y#类别
        self.all_classes = list(set(Y))#所有类的集合,在本例题中就是{1, 2},为了让其可下标索引，又转换成list,相当于对Y做了一次去重操作
        self.times_classes = []#给出的样本中，每个类出现的频率(概率)
        self.attributes = []#内容是集合，集合装着每个属性可能的取值。

        #开始计算pr_classes
        for c in self.all_classes:
            count = 0
            for item in Y:
                if item == c:
                    count += 1
            self.times_classes.append(count)

        #开始计算attributes
        row, column = X.shape
        for index in range(0, column):
            possible_value = set(X[:, index])
            self.attributes.append(possible_value)

    def predict(self, vector):#给一个属性向量,预测其概率
        result = []
        for i in range(0, len(self.all_classes)):#对每一个类都求概率
```

```
            cls = self.all_classes[i]#当前算的是哪个类的概率

            fr_cls = self.times_classes[i]#当前类出现的频率
            buf = fr_cls / len(self.y)#P(c)
            frequency = [0] * len(vector) # 每个元素都是 样本中 类=cls 且 属性与vector
相同 出现的频率,即 数学表达式中的|Dc,Xi|，算出所有的

            #计算frequency
            for j in range(0, len(self.y)):
                if self.y[j] == cls:
                    sample = self.x[j]
                    for k in range(0, len(sample)):
                        if sample[k] == vector[k]:
                            frequency[k] += 1

            #现在对frequency处理，得到p(xi|c)们,并与buf相乘
            for i in range(0, len(frequency)):
                num = frequency[i] + 1#分子
                den = fr_cls + len(self.attributes[i])#分母
                pr = num / den
                buf *= pr

            result.append('为%d类的未归一化概率是%lf'%(cls, buf))
        print(result)
```

运行结果：

```
if __name__ == '__main__':
    NB = My_Module(X,Y)
    NB.predict([2,3,1,2,3])
    NB.predict([1,1,2,1,3])

['为1类的未归一化概率是0.002494', '为2类的未归一化概率是0.006849']
['为1类的未归一化概率是0.005986', '为2类的未归一化概率是0.001141']
```

比较这些概率，程序给出的结果是，X是2类的概率比较大，Y是1类的概率比较大。

也就是X是入侵，Y是正常

4.6节

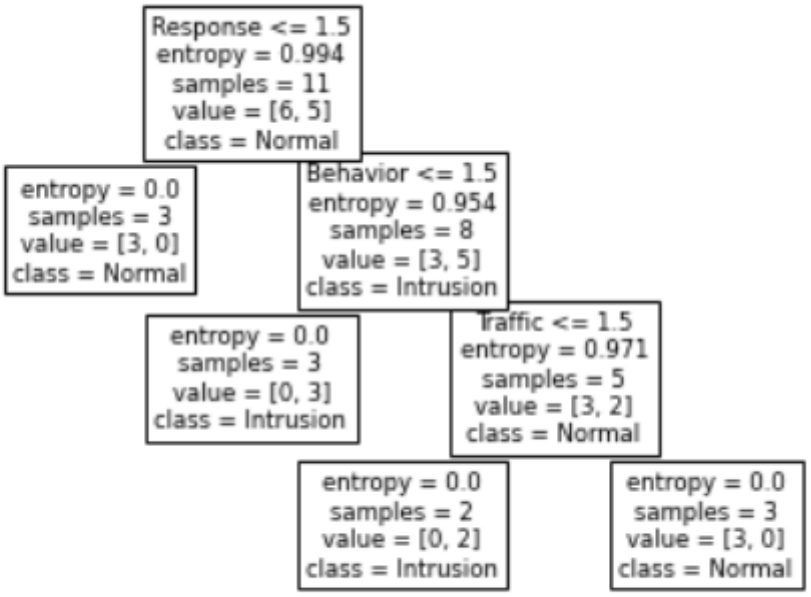


图 3、决策树

#	时延 (DELAY)	响应 (RESPONSE)	流量 (TRAFFIC)	行为 (BEHAVIOR)	内存 (MEMORY)	正常/入侵 （NORMAL/INTRUSION）
1	高	快	异常	正常	不变	正常
2	中	快	正常	异常	不变	正常
3	低	中	未知	正常	增大	正常
4	高	慢	正常	正常	减小	正常
5	中	中	未知	正常	增大	正常
6	低	快	正常	异常	不变	正常
7	高	中	异常	正常	增大	入侵
8	中	慢	正常	异常	增大	入侵
9	中	慢	正常	异常	不变	入侵
10	高	慢	异常	正常	不变	入侵
11	低	中	未知	异常	增大	入侵

22

参考资料给的应该有些许问题，如果第一次按Behavior分，

6个行为正常的中，有4个正常，2入侵

5个行为异常的中，有2个正常，3入侵。

分出来的应该是[4,2],[2,3]

而参考资料写[3,0]和[3,5]

```
class Node:
    def __init__(self, set_of_index=None, set_of_counter=None, entropy=None):
        self.set_of_index = set_of_index
```

```
        self.set_of_counter = set_of_counter
        self.entropy = entropy
        self.child = None
        self.division_choice = None

    def __str__(self):
        return str(self.set_of_index) + '  ' + str(self.set_of_counter) + '  ' +
str(self.entropy)

    def display(self):
        print(self,end='  ')
        print('分割方案', self.division_choice)
        if self.child:
            for i in range(0, len(self.child)):
                print('第', i, '个子集', self.child[i])
            print()
            for child in self.child:
                child.display()
```

决策树也是树，于是写了一个树类，有一个重载字符串，和一个用于遍历树的函数

一个封装了所有功能的类，叫做My_Module.

上面的树类依赖于这个类

```
class My_Module:
    def __init__(self, X, Y):#样本
        self.X = X#属性矩阵
        self.Y = list(Y)
        self.class_names = list(set(Y))#所有类的集合,在本例题中就是{1, 2},为了让其可下标
索引，又转换成list,相当于对Y做了一次去重操作
        self.features = []#内容是集合，集合装着每个属性可能的取值。
        self.DecisionTree = None
        #开始计算features
        row, column = X.shape
        for index in range(0, column):
            possible_value = set(X[:, index])
            self.features.append(possible_value)

        #开始计算entropy
        buf = []
        for cls in self.class_names:
            frequency = self.Y.count(cls)
            buf.append(frequency)
        self.entropy = cal_entropy(buf)

    def classify(self, feature, node):#对样本集合（或子集）分一次类,feature下标是Index
        index = feature
        possible_value = list(self.features[index])
        subsets = []
        #开始计算子集
        #子集做初始化
        for i in range(0, len(possible_value)):
```

```
            buf = []
            subsets.append(buf)
        #subsets就会长这样:[[],[],[]]
        #下面开始遍历一次样本，按属性分类。
        for i in node.set_of_index:
            sample = self.X[i]
            index1 = possible_value.index(sample[index])
            subsets[index1].append(i)

        #print('按下标为:', index, '的属性分类，共分为', len(possible_value), '个子集',
'含有不同类的样本个数分别为',subsets)

        return subsets

    def create_node(self, set_of_index):#由下标的集合 算出节点所需的信息
        if not set_of_index:
            return None
        buf = [self.Y[index] for index in set_of_index]
        diversity = list(set(buf))
        set_of_counter = [0] * len(diversity)
        for cls in buf:
            index = diversity.index(cls)
            set_of_counter[index] += 1

        entropy = cal_entropy(set_of_counter)

        return Node(set_of_index, set_of_counter, entropy)

    def decide(self, node):#对一个决策树节点，进行下一步决策,决定以什么feature为分类
        if node.entropy == 0:
            return -1

        container = []#装各个分类下的节点
        #开始计算所有决策
        for feature_index in range(0, len(self.features)):
            subsets = self.classify(feature=feature_index,node=node)
            buf = []
            for set_of_index in subsets:
                nd = self.create_node(set_of_index=set_of_index)
                if nd:
                    buf.append(nd)
            container.append(buf)

        #开始分析每个决策导致的信息增量，比较之
        gains = []
        card = len(node.set_of_index)
        for division in container:
            new_entropy = 0
            for n in division:#n是node对象
                n_card = len(n.set_of_index)
                new_entropy += (n_card/card) * n.entropy
            #print('_____________________________________________________')
            gains.append(node.entropy - new_entropy)

        max_gain = max(gains)
        choice = gains.index(max_gain)
```

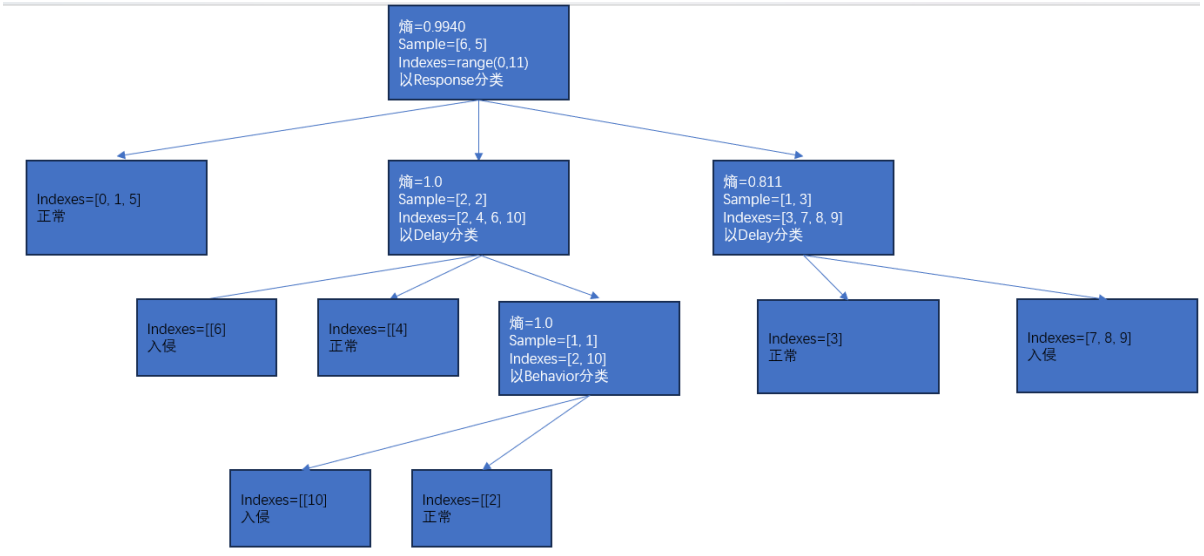
```
        #print(gains)
        #print('节点', node, '的最佳分类法是按下标为', choice, '的属性分类')

        return choice, container[choice]#返回决策，及其分割



    def create_decision_tree(self):
        tree = self.create_node(set_of_index=range(0, len(self.Y)))
        self.divide(node=tree)
        return tree


    def divide(self, node):
        buf = self.decide(node=node)
        if buf == -1:
            return None
        else:
            node.division_choice, node.child = buf
            for child in node.child:
                self.divide(node=child)
```

```
import math
import numpy as np
# 网络时延分为:高、中、低 -> 1, 2, 3
# 响应速度分为:快、中、慢 -> 1, 2, 3
# 流量异常分为:异常、正常、未知 -> 1, 2, 3
# 行为异常分为:异常、正常 -> 1, 2
# 存储增大分为:增大、减小、不变 -> 1, 2, 3
# 标签: 1:正常， 2:入侵
X = np.array([[1,1,1,2,3],
[2,1,2,1,3],
[3,2,3,2,1],
[1,3,2,2,2],
[2,2,3,2,1],
[3,1,2,1,3],
[1,2,1,2,1],
[2,3,2,1,1],
[2,3,2,1,3],
[2,3,1,2,3],
[3,2,3,1,1]])
# 标签: 1:正常， 2:入侵
Y = np.array([1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2])

if __name__ == '__main__':
    M = My_Module(X, Y)
    tree = M.create_decision_tree()
    tree.display()
```

运行结果很长,把树的遍历结果展示了出来

```
range(0, 11)  [6, 5]  0.9940302114769565  分割方案 1
第 0 个子集 [0, 1, 5]  [3]  0.0
第 1 个子集 [2, 4, 6, 10]  [2, 2]  1.0
第 2 个子集 [3, 7, 8, 9]  [1, 3]  0.8112781244591328

[0, 1, 5]  [3]  0.0  分割方案 None
[2, 4, 6, 10]  [2, 2]  1.0  分割方案 0
第 0 个子集 [6]  [1]  0.0
第 1 个子集 [4]  [1]  0.0
第 2 个子集 [2, 10]  [1, 1]  1.0

[6]  [1]  0.0  分割方案 None
[4]  [1]  0.0  分割方案 None
[2, 10]  [1, 1]  1.0  分割方案 3
第 0 个子集 [10]  [1]  0.0
第 1 个子集 [2]  [1]  0.0

[10]  [1]  0.0  分割方案 None
[2]  [1]  0.0  分割方案 None
[3, 7, 8, 9]  [1, 3]  0.8112781244591328  分割方案 0
第 0 个子集 [3]  [1]  0.0
第 1 个子集 [7, 8, 9]  [3]  0.0

[3]  [1]  0.0  分割方案 None
[7, 8, 9]  [3]  0.0  分割方案 None

进程已结束,退出代码0
```

树的图形表示



4、证明差分隐私具有后处理不变性。

差分隐私的后处理不变性指的是在对已经通过差分隐私机制保护的数据进行一些合法的后续处理时，保持差分隐私的性质不受影响。

换句话说，如果对差分隐私保护的数据进行了一些操作，比如加噪声、扰动等，那么对处理后的数据进行分析不应该泄露更多的隐私信息，即使攻击者知道了原始数据的差分隐私保护程度和处理过程。

设函数A_1满足$\varepsilon - DP$定义，即

$\forall t \in A_1$的值域，都满足$\frac{P[A_1(D)=t]}{P[A_1(D')=t]} \le e^{\varepsilon}$

任意函数A_2，后处理不变，指的是$A_2(A_1())$依然满足$\varepsilon - DP$

证明：

概率的乘法法则，$P[A_2(A_1(D))=t] = \sum_{temp \in A_1\text{的值域}} P[A_1(D)=temp] \times P[A_2(temp)=t]$

由于A_1满足$\varepsilon - DP$，由定义，$P[A_1(D)=temp] \le e^{\varepsilon} P[A_1(D')=t]$，代入上式

故$P[A_2(A_1(D))=t] \le e^{\varepsilon} \sum_{temp \in A_1\text{的值域}} P[A_1(D')=temp] \times P[A_2(temp)=t]$

这一不等式的右边也就是$e^{\varepsilon} P[A_2(A_1(D'))=t]$

所以得到了$\frac{P[A_2(A_1(D))=t]}{P[A_2(A_1(D'))=t]} \le e^{\varepsilon}$

5、把随机响应技术的掷硬币方式改为：首先掷一枚不均匀的硬币，如果是反面，请如实回应；如果是正面，那么再掷第二枚均匀的硬币，如果正面回答“是”，如果是反面则回答“否”。重新估计艾滋病患者的比例。

设这不均匀的硬币抛出正面的概率为ε

设人群中，患与不患艾滋病的概率向量(是，否) =（π,1-π）

如果抛出反面,概率为1-ε，那受访者回答的就是真实的概率向量（π,1-π）

如果抛出正面后，均匀硬币抛出正面，概率为$\frac{\varepsilon}{2}$，受访者一定回答是，概率向量为(1,0)

如果抛出正面后，均匀硬币抛出反面，概率为$\frac{\varepsilon}{2}$，受访者一定回答否，概率向量为(0,1)

故理论上此样本的概率向量为$(1-\varepsilon)(\pi,1-\pi)+\frac{\varepsilon}{2}(1,0)+\frac{\varepsilon}{2}(0,1)=(\pi-\varepsilon\pi+\frac{\varepsilon}{2},\ 1-\pi-\frac{\varepsilon}{2}+\varepsilon\pi)$

如果统计显示概率向量是(a,b),那么在ε已知的情况下，可以通过

$\pi-\varepsilon\pi+\frac{\varepsilon}{2}=a$　　或者　　$1-\pi-\frac{\varepsilon}{2}+\varepsilon\pi=b$　　一元一次方程解出艾滋病患者的比例π