第 3 章

CHAPTER 3

贝叶斯分类器

贝叶斯分类器是一个相当宽泛的定义，它背后的数学理论根据是相当出名的贝叶斯决策论(Bayesian Decision Theory)。贝叶斯决策论和传统的统计学理论有着区别，其中最不可调和的就是它们各自关于概率的定义。因此，使用了贝叶斯决策论作为基石的贝叶斯分类器，在各个 Python 算法所导出的分类器中也算是比较标新立异的存在。

3.1 贝叶斯学派

贝叶斯分类器是各种分类器中分类错误概率最小或者在预先给定代价的情况下平均风险最小的分类器。它的设计方法是一种最基本的统计分类方法。其分类原理是通过某对象的先验概率，利用贝叶斯公式计算出其后验概率，即该对象属于某一类的概率，选择具有最大后验概率的类作为该对象所属的类。

为了深刻理解贝叶斯分类器，需要先对贝叶斯学派及其决策理论有一个大致的认识。

3.1.1 贝叶斯学派论述

贝叶斯学派强调概率的“主观性”，这一点和传统的、我们比较熟悉的频率学派不同。主要表现在：

- 频率学派强调频率的“自然属性”，认为应该使用事件在重复实验中发生的频率作为其发生的概率的估计。
- 贝叶斯学派不强调事件的“客观随机性”，认为仅仅只是“观察者”不知道事件的结果。换句话说，贝叶斯学派认为：事件之所以具有随机性。仅仅是因为“观察者”的知识不完备，对于“知情者”来说，该事件其实不具备随机性。随机性的根源不在于事件，而在于“观察者”对该事件的知识状态。

举个例子：假设有一个人抛了一枚质地均匀的硬币到地上并迅速将其踩在脚底，而在他面前从近到远坐了三个人。他本人看到了硬币是正面朝上的，而其他三个人也多多少少看到了一些信息，但显然坐得越远，看得就越模糊。频率学派会认为，该硬币是正是反，各自

的概率都应该是50%；但是贝叶斯学派会认为，对抛硬币的人来说，硬币是正面的概率就是100%，对离他最近的人来说可能是80%，对离他最远的人来说可能是50%。

所以相比起把模型参数固定、注重样本的随机性的频率学派而言，贝叶斯学派将样本视为固定的，把模型的参数视为关键。在上面这个例子中，样本就是抛出去的那枚硬币，模型的参数就是每个人从中获得的“信息”。对于频率学派而言，每个人获得的“信息”不应该有不同，所以自然会根据“均匀硬币抛出下面的概率为50%”这个“样本的信息”来导出“硬币是正面的概率为50%”这个结论。但是对贝叶斯学派而言，硬币抛出去就抛出去了，问题的关键在于模型的参数，亦即“观察者”从中获得的信息，所以会导致“对于抛硬币的人而言，硬币是正面的概率为100%”这一类的结论。

3.1.2　贝叶斯决策论

在大致知道贝叶斯学派的思想后，就可以介绍贝叶斯决策论了。这里不可避免地要涉及概率论和数理统计的相关定义和知识，幸运的是，它们都是比较基础且直观的部分，无须太多的数学背景就可以知道它们的含义。

- 行动空间 A，它是某项实际工作中可能采取的各种“行动”所构成的集合。

注意：贝叶斯学派注意的是模型参数，所以通常而言我们想要做出的“行动”是“决策模型的参数”。因此我们通常会将行动空间取为参数空间，亦即 $A=\theta$。

- 决策 $\delta(\widetilde{\boldsymbol{X}})$，它是样本空间 X 到行动空间 A 的一个映射。换句话说，对于一个单一的样本 $\widetilde{\boldsymbol{X}}(\widetilde{\boldsymbol{X}}\in X)$，决策函数可以利用它得到 A 中的一个行动。

注意：这里的样本 $\widetilde{\boldsymbol{X}}$ 通常是高维的随机向量：$\widetilde{\boldsymbol{X}}=(x_1,x_2,\cdots,x_N)^{\mathrm{T}}$；尤其需要分清的是，这个 $\widetilde{\boldsymbol{X}}$ 其实是一般意义上的“训练集”，x_i 才是一般意义上的“样本”。

- 损失函数 $L(\theta,a)=LL(\theta,\delta(\widetilde{\boldsymbol{X}}))$，它表示参数 $\theta(\theta\in\Theta,\Theta$ 是参数空间)时采取行动 $a(a\in A)$所引起的损失。
- 决策风险 $R(\theta,\delta)$，它是损失函数的期望：$R(\theta,\delta)=EL(\theta,\delta(\widetilde{\boldsymbol{X}}))$。
- 先验分布：描述了参数 θ 在已知样本 $\widetilde{\boldsymbol{X}}$ 中的分布。
- 平均风险 $\rho(\delta)$，它定义为决策风险 $R(\theta,\delta)$在先验分布下的期望：

$$\rho(\delta)=E_{\xi}R(\theta,\delta)$$

- 贝叶斯决策 δ^*，它满足：

$$\rho(\delta^*)=\inf_{\delta}\rho(\delta)$$

换句话说，贝叶斯决策 δ^* 是在某个先验分布下使得平均风险最小的决策。

寻找一般意义下的贝叶斯决策是相当不易的数学问题，为简洁起见，需要结合具体的算法来推导相应的贝叶斯决策。

3.1.3 贝叶斯原理

贝叶斯决策论在相关概率已知的情况下利用误判损失来选择最优的类别分类。

“风险”(误判损失)= 原本为 c_j 的样本误分类成 c_i 产生的期望损失(如下式,概率乘以损失为期望损失):

$$R(c_i \mid X)=\sum_{j=1}^{N}\lambda_{ij}P(c_j \mid X)$$

为了最小总体风险,只需在每个样本上选择能够使条件风险 $R(c|X)$最小的类别标记。

$$h^*(x)=\underset{c\cap y}{\operatorname{argmin}}R(c \mid X)$$

h^* 称为贝叶斯最优分类器,与之对应的总体风险为贝叶斯风险,当 λ 等于 1 时,最优贝叶斯分类器是使后验概率 $P(c|X)$最大。

利用贝叶斯判定准则来最小化决策风险,首先要获得后验概率 $P(c|X)$,机器学习则是基于有限的训练样本集尽可能准确地估计出后验概率 $P(c|X)$。通常有两种模型:

(1) 判别式模型。通过直接建模 $P(c|X)$来预测(决策树、BP 神经网络、支持向量机)。

(2) 生成式模型。通过对联合概率模型 $P(X,c)$进行建模,然后再获得 $P(c|X)$。

$$P(c \mid X)=\frac{P(X,c)}{P(X)}=\frac{P(c)P(X \mid c)}{P(X)}$$

$P(c)$是类“先验”概述,$P(X|c)$是样本 X 相对于类标记条件概率,或称似然。似然函数的定义为:对同一个似然函数,如果存在一个参数值,使得它的函数值达到最大,那么这个值就是最为“合理”的参数值。

$P(c)$代表样本空间中各类样本所占的比例,根据大数定理,当训练集包含充足的独立同分布样本时,可通过各类样本出现的频率进行估计。而 $P(X|c)$涉及关于所有属性的联合概率,无法根据样本出现的频率进行估计。

【例 3-1】 Python 实现简单贝叶斯分类。

```
import numpy
#先生成原始数据
n1 = 200;n2 = 40;m = 4;h = 6
#n1 代表总样品数,n2 是测试样品数,m 是种类数,h 是特征数

S1 = numpy.zeros((200,7))
#最后一列代表类别
S1[0:30,0:6] = numpy.random.randn(30,6)
S1[0:30,6] = 1
S1[30:60,0:6] = numpy.random.randn(30,6) * 3 + 10
S1[30:60,6] = 2
S1[60:140,0:6] = numpy.random.randn(80,6) * 5 + 20
S1[60:140,6] = 3
S1[140:,0:6] = numpy.random.randn(60,6) * 10 + 30
S1[140:,6] =
```

```
S2 = numpy.zeros((40,6))
S2[0:10,:] = numpy.random.randn(10,6)
S2[10:20,:] = numpy.random.randn(10,6) * 3 + 10
S2[20:30,:] = numpy.random.randn(10,6) * 5 + 20
S2[30:,:] = numpy.random.randn(10,6) * 10 + 30
#si 为样本,X 为 test 数据
#单个数据在模板空间 Si 中出现的概率
def beiyesi(X,Si):
 S = numpy.cov(Si.T)
 S = numpy.mat(S);m,n = Si.shape
 a = 1/((2 * numpy.pi) ** (n/2) * numpy.linalg.det(S) ** 0.5)
 X = X - numpy.mean(Si,axis = 0)
 ans = numpy.dot(X,S.I)
 ans = numpy.dot(ans,X.T)
 p = numpy.log(a * (numpy.exp( - 0.5 * ans)))
 return p
def many_beiyesi(X,Si):
 #多个数据在模板空间 Si 中出现的概率
 a,b = X.shape
 P = [  beiyesi(X[i,:  ],Si) for i in range(a)]
 return P
def final(X,S,m):
 li = []
 w = []
 a,b = X.shape
 n1,h = S.shape
 h = h - 1
 #P[i,j]代表第 j 个数据在第 i 个模板空间中出现的概率
 P = numpy.zeros((m,a))
 Q = numpy.zeros((m,a))
 #Q[i,j]代表第 j 个数据属于第 i 个模板空间的概率
 for i in range(m):
     li.append(numpy.argwhere(S == i + 1)[:,0])
     w.append(len(li[i]))
     Si = S[li[i],0:h]
     p = many_beiyesi(X,Si)
     for j in range(a):
         P[i,j] = p[j]
 for i in range(m):
     w[i] = float(w[i])/sum(w)
 for i in range(m):
     for j in range(a):
         Q[i,j] = P[i,j] * w[i]/numpy.dot(w,P[:,j])
 return P,Q
P,Q = final(S2,S1,m)
```

运行程序,输出如下:

```
test.py:36: RuntimeWarning: divide by zero encountered in log
  p = numpy.log(a * (numpy.exp( - 0.5 * ans)))
```

```
test.py:64: RuntimeWarning: invalid value encountered in double_scalars
  Q[i,j] = P[i,j] * w[i]/numpy.dot(w,P[:,j])
```

3.2 参数估计

无论是贝叶斯学派还是频率学派，一个无法回避的问题就是如何从已有的样本中获取信息并据此估计目标模型的参数。比较有名的“频率近似概率”其实就是(基于大数定律的)相当合理的估计之一。

3.2.1 似然函数

在数理统计学中，似然函数是一种关于统计模型参数的函数，表示模型参数中的似然性。“似然性”与“或然性”或“概率”意思相近，都是指某种事件发生的可能性，但是在统计学中，“似然性”和“或然性”或“概率”又有明确的区分。概率用于在已知一些参数的情况下，预测接下来的观测所得到的结果，而似然性则是用于在已知某些观测所得到的结果时，对有关事物的性质的参数进行估计。

例如，在已知某个参数 θ 的情况下事件 X 会发生的概率写作：

$$p(X \mid \theta) = \frac{p(X,\theta)}{p(\theta)}$$

根据贝叶斯定理，有：

$$p(X \mid \theta) = \frac{p(X \mid \theta)p(\theta)}{p(X)}$$

因此，可以反过来构造表示似然性的方法：已知有事件 X 发生，运用似然函数 $L(\theta|X)$估计参数 Y 的可能性。

3.2.2 极大似然估计原理

频率学派认为已知一个分布，虽然不知道分布的具体参数，但是却客观上存在固定的参数值。因此可以通过一些准则来确定参数值。这里介绍的极大似然估计就是一种根据采样来估计概率分布参数的经典方法。

最大似然估计会寻找关于 θ 的最可能的值(即，在所有可能的 θ 取值中，寻找一个值使这个采样的“可能性”最大化，相当于是利用概率密度函数参数去拟合采样的结果)。现在的工作就是最大化似然函数：

$$p(\theta \mid X) = \frac{p(X \mid \theta)p(\theta)}{p(X)}$$

根据大数定律，当训练集包含充足的独立同分布样本时，$p(X)$可以通过各类样本出现的频率来进行估计。在这里，已知样本之后，就可以估算出 $p(X)$的值，并将其当作固定值

处理。

现在需要根据所有可能 θ 的取值，选取一个让 $p(\theta|X)$ 最大化的值。这里还是以正态分布为例。为了简化运算，我们对似然函数取对数。最大化一个似然函数同最大化它的自然对数是等价的，因为自然对数 log 是一个连续且在似然函数的值域内严格递增的上凸函数。

3.2.3　极大似然估计(ML 估计)

如果将模型描述成一个概率模型，那么一个自然的想法是希望得到的模型参数 θ 能够使得在训练集 $\widetilde{\boldsymbol{X}}$ 作为输入时、模型输出的概率达到极大。这里就有一个似然函数的概念，它能够输出 $\widetilde{\boldsymbol{X}}=(x_1,x_2,\cdots,x_N)^{\mathrm{T}}$ 在模型参数 θ 下的概率：

$$p(\widetilde{\boldsymbol{X}}\mid\theta)=\prod_{i=1}^{N}p(x_i\mid\theta)$$

我们希望找到的 $\hat{\theta}$，就是使得似然函数在 $\widetilde{\boldsymbol{X}}$ 作为输入时达到极大的参数。

$$\hat{\theta}=\underset{\theta}{\operatorname{argmax}}p(\widetilde{\boldsymbol{X}}\mid\theta)=\underset{\theta}{\operatorname{argmax}}\prod_{i=1}^{N}p(x_i\mid\theta)$$

举个例子：假设一个暗箱中有白球、黑球共两个，显然不知道具体的颜色分布情况，但是知道这两个球是完全一样的。现在有放回地从箱子里抽了两个球，发现两次抽出来的结果是一黑一白，那么该如何估计箱子里面球的颜色？从直观上来说，似乎箱子中也是一黑一白比较合理，下面就来说明“一黑一白”这个估计就是极大似然估计。

在这个问题中，模型的参数 θ 可以设为从暗箱中抽出黑球的概率，样本 x_i 可以描述为第 i 次取出的球是否是黑球；如果是就是取 1，否则取 0。这样，似然函数就可以描述为：

$$p(\widetilde{\boldsymbol{X}}\mid\theta)=\theta^{x_1+x_2}(1-\theta)^{2-x_1-x_2}$$

直接对它求极大值(虽然可行但是)不太方便，通常的做法是将似然函数取对数之后再进行极大值的求解：

$$\ln p(\widetilde{\boldsymbol{X}}\mid\theta)=(x_1+x_2)\ln\theta+2-x_1-x_2\ln(1-\theta)$$

$$\Rightarrow\frac{\partial\ln p}{\partial\theta}=\frac{x_1+x_2}{\theta}-\frac{2-x_1-x_2}{1-\theta}$$

从而可知：

$$\frac{\partial\ln p}{\partial\theta}=0\Rightarrow\theta=\frac{x_1+x_2}{\theta}$$

由于 $x_1+x_2=1$，所以得 $\hat{\theta}=0.5$，亦即应该估计从暗箱中抽出黑球的概率是 50%；既然暗箱中的两个球完全一样，我们应该估计暗箱中的颜色分布为一黑一白。

从以上的讨论可以看出，极大似然估计视待估计参数为一个未知但固定的量，不考虑“观察者”的影响(亦即不考虑先验知识的影响)，是传统的频率学派的做法。

【例 3-2】 利用 Python 对正态分布的数据进行最大似然估计。

```
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
mu = 30                                                    # 分配的平均值
sigma = 2                                                  # 分布的标准偏差
x = mu + sigma * np.random.randn(10000)
def mle(x):
    """
极大似然估计
    :param x:
    :return:
    """
    u = np.mean(x)
    return u, np.sqrt(np.dot(x - u, (x - u).T) / x.shape[0])
print(mle(x))
num_bins = 100
plt.hist(x, num_bins)
plt.show()
```

运行程序,效果如图 3-1 所示。

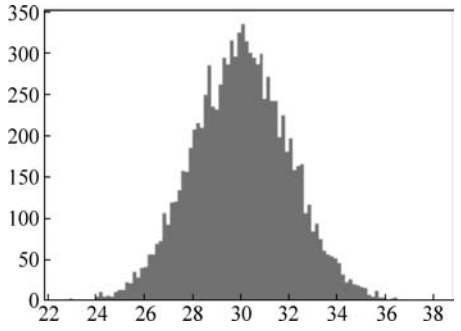

图 3-1 最大似然估计

3.2.4 极大后验概率估计(MAP 估计)

相比起极大似然估计,极大后验概率估计是更贴合贝叶斯学派思想的做法。事实上,甚至也有不少人直接称其为“贝叶斯估计”。

在讨论 MAP 估计之前,有必要先知道何为后验概率 $p(\theta|\tilde{\boldsymbol{X}})$。它可以理解为参数 θ 在训练集 $\tilde{\boldsymbol{X}}$ 下所谓的“真实的出现概率”,能够利用参数的先验概率 $p(\theta)$、样本的先验概率 $p(\tilde{\boldsymbol{X}})$和条件概率 $p(\tilde{\boldsymbol{X}} \mid \theta)=\prod_{i=1}^{N} p(x_i \mid \theta)$ 通过贝叶斯公式导出。

而 MAP 估计的核心思想,就是将待估参数 θ 看成是一个随机变量,从而引入了极大似

然估计中没有引入的、参数 θ 的先验分布。MAP 估计 $\hat{\theta}_{\mathrm{MAP}}$ 的定义为：

$$\hat{\theta}_{\mathrm{MAP}}=\underset{\theta}{\operatorname{argmax}}p(\theta\mid\widetilde{\boldsymbol{X}})=\underset{\theta}{\operatorname{argmax}}p(\theta)\prod_{i=1}^{N}p(x_i\mid\theta)$$

同样，为了计算简便，通常对此式取对数：

$$\hat{\theta}_{\mathrm{MAP}}=\underset{\theta}{\operatorname{argmax}}\ln p(\theta\mid\widetilde{\boldsymbol{X}})=\underset{\theta}{\operatorname{argmax}}\left[\ln p(\theta)+\sum_{i=1}^{N}\ln p(x_i\mid\theta)\right]$$

可以看到，从形式上，极大后验概率估计只比极大似然估计多了 $\ln p(\theta)$ 这一项，但它们背后的思想相当不同。下面在具体讨论朴素贝叶斯算法时将会看到：朴素贝叶斯在估计参数时选用了极大似然估计法，但是在做决策时则选用了 MAP 估计。

和极大似然估计相比，MAP 估计的一个显著优势在于它可以引入所谓的“先验知识”，这正是贝叶斯学派的精髓。当然这个优势同时也伴随着劣势：它要求我们对模型参数有相对较好的认知，否则会相当大地影响到结果的合理性。

既然先验分布如此重要，那么对于先验分布是否有比较合理的选取方法呢？事实上，如何确定先验分布这个问题，正是贝叶斯统计中最困难、最具有争议性却又必须解决的问题。虽然这个问题确实有许多现代的研究成果，但遗憾的是，尚未能有一个较完善的理论和普适的方法。

所选择的参数 θ 的先验分布，应该与由它和训练集确定的后验分布属同一类型。

此时先验分布又叫共轭先验分布。这里面所谓的“同一类型”其实又是难有恰当定义的概念，但是可以直观地理解为：概率性质相似的所有分布归为“同一类型”。比如，所有的正态分布都是“同一类型”的。

3.3　朴素贝叶斯

在朴素贝叶斯这个名字中，“朴素”二字对应着“独立性假设”这一个朴素的假设，“贝叶斯”则对应“后验概率最大化”这一贝叶斯学派的思想。

3.3.1　基本框架

朴素贝叶斯算法一个非常重要的基本假设称为独立性假设，其大致叙述如下：

如果样本空间 X 是 n 维的，那么对 $\forall x=(x^{(1)},x^{(2)},\cdots,x^{(n)})^{\mathrm{T}}\in X$，假设 $x^{(i)}$ 是由随机变量 $X^{(i)}$ 生成的，且 $X^{(1)},X^{(2)},\cdots,X^{(n)}$ 之间在各种意义下相互独立。

在朴素贝叶斯算法思想下，一般来说会衍生出以下 3 种不同的模型。

- 离散型朴素贝叶斯(MultinomialNB)：所有维度的特征都是离散型随机变量。
- 连续型朴素贝叶斯(GaussianNB)：所有维度的特征都是连续型随机变量。
- 混合型朴素贝叶斯(MergedNB)：各个维度的特征有离散型也有连续型。

由浅入深，我们先用离散型朴素贝叶斯来说明一些普适性的概念，连续型和混合型的相关定义是类似的。

- 朴素贝叶斯的模型参数即是类别的选择空间（假设一共有 K 类：$c_1,c_2,\cdots,c_K$）：

$$\Theta=\{y=c_1,y=c_2,\cdots,y=c_K\}$$

- 朴素贝叶斯总的参数空间 $\widetilde{\Theta}$ 本应包括模型参数的先验概率 $p(\theta_k)=p(y=c_k)$、样本空间在模型参数下的条件概率 $p(X|\theta_k)=p(X|y=c_k)$ 和样本空间本身的概率 $p(X)$。

但由于我们采取样本空间的子集 $\widetilde{X}$ 作为训练集，所以在给定的 $\widetilde{X}$ 下，$p(X)=p(\widetilde{X})$ 是常数，因此可以把它从参数空间中删去。换句话说，我们关心的只有模型参数的先验概率和样本空间在模型参数下的条件概率：

$$\widetilde{\Theta}=\{p(\theta),p(X\mid\theta):\theta\in\Theta\}$$

- 行动空间 A 是朴素贝叶斯总的参数空间 $\widetilde{\Theta}$。
- 决策就是后验概率最大化：

$$\delta(\widetilde{X})=\hat{\theta}=\underset{\widetilde{\theta}\in\widetilde{\Theta}}{\operatorname{argmax}}p(\widetilde{\theta}\mid\widetilde{X})$$

在 $\hat{\theta}$ 确定后，模型的决策就可以具体写成（这一步用到了独立性假设）：

$$\begin{aligned}f(x^*)&=\underset{c_k}{\operatorname{argmax}}\hat{p}(c_k\mid X=x^*)\\&=\underset{c_k}{\operatorname{argmax}}\hat{p}(y=c_k)\prod_{j=1}^{N}\hat{p}(X^{(j)}=x^{*(j)}\mid y=c_k)\end{aligned}$$

- 损失函数会随模型的不同而不同。在离散型朴素贝叶斯中，损失函数就是比较简单的 0-1 损失函数：

$$L(\theta,\delta(\widetilde{X}))=\sum_{i=1}^{N}\widetilde{L}(y_i,f(x_i))=\sum_{i=1}^{N}I(y_i\neq f(x_i))$$

这里的 I 是示性函数，它满足：

$$I(y_i\neq f(x_i))=\begin{cases}1, & y_i\neq f(x_i)\\0, & y_i=f(x_i)\end{cases}$$

从上述定义出发，可以利用两种参数估计方法导出离散型计算朴素贝叶斯的算法。下面介绍其算法。

输入：训练数据集 $D=\{(x_1,y_1),(x_2,y_2),\cdots,(x_N,y_N)\}$。

过程（利用 ML 估计导出模型的具体参数）：

（1）计算先验概率 $p(y=c_k)$ 的极大似然估计

$$\hat{p}(y=c_k)=\frac{\sum_{i=1}^{N}I(y_i=c_k)}{N},\quad k=1,2,\cdots,K$$

(2) 计算条件概率 $p(X^{(j)}=a_{ji}|y=c_k)$的极大似然估计(设每一个单独输入的 n 维向量 $\boldsymbol{x}$ 的 j 维特征 $x^{(j)}$ 可能的取值集合为$\{a_{ji},a_{j(i+1)},\cdots,a_{js_j}\}$):

$$p(X^{(j)}=a_{ji}\mid y=c_k)=\frac{\sum_{i=1}^{N}I(x_i^{(j)}=a_{jl},y_i=c_k)}{\sum_{i=1}^{N}I(y_i=c_k)}$$

输出(利用 MAP 估计进行决策):朴素贝叶斯模型,能够估计数据 $\boldsymbol{x}^*=(x^{*(1)},x^{*(2)},\cdots,x^{*(n)})^{\mathrm{T}}$ 的类别:

$$y=f(\boldsymbol{x}^*)=\underset{c_k}{\operatorname{argmax}}\hat{p}(y=c_k)\prod_{j=1}^{n}\hat{p}(X^{(j)}=x^{*(j)}\mid y=c_k)$$

由上述算法可以清晰地梳理出朴素贝叶斯算法背后的数学思想:

- 使用极大似然估计导出模型的具体参数(先验概率、条件概率)。
- 使用极大后验概率估计出作为模型的决策(输出使得数据后验概率最大化的类别)。

【例 3-3】 在一个简单、虚拟的数据集上应用离散型朴素贝叶斯算法以加深对算法的理解。该数据集如表 3-1 所示。

表 3-1　气球数据集 1.0

颜　色	大　小	测试人员	测试动作	结　果
黄色	小	成人	用手打	不爆炸
黄色	小	成人	用脚踩	爆炸
黄色	小	小孩	用手打	不爆炸
黄色	小	小孩	用脚踩	不爆炸
黄色	大	成人	用手打	爆炸
黄色	大	成人	用脚踩	爆炸
黄色	大	小孩	用手打	不爆炸
黄色	大	小孩	用脚踩	爆炸
紫色	小	成人	用手打	不爆炸
紫色	小	小孩	用手打	不爆炸
紫色	大	成人	用脚踩	爆炸
紫色	大	小孩	用脚踩	爆炸

该数据集的电子版本可参见 https://github.com/cr=arefree0910/MachineLearning/blob/master/_Data/balloon1.0.txt。我们想预测的是样本:

紫色	小	小孩	用脚踩

所导致的结果。容易观察到的是,气球的颜色对结果不起丝毫影响,所以在算法中该项特征可以直接去掉。因此从直观上来说,以样本所导致的结果应该是“不爆炸”,我们用离散型朴素贝叶斯算法来看看是否确实如此。首先需要计算类别的先验概率,易得:

$$p(\text{不爆炸})=p(\text{爆炸})=0.5$$

也可得到类别的先验概率对决策不起作用。继而需要依次求出第2、3、4个特征(大小、测试人员、测试动作)的条件概率,它们才是决定新样本所属类别的关键。易得:

$$p(\text{小气球}\mid\text{不爆炸})=\frac{5}{6},\quad p(\text{大气球}\mid\text{不爆炸})=\frac{1}{6}$$

$$p(\text{小气球}\mid\text{爆炸})=\frac{1}{6},\quad p(\text{大气球}\mid\text{不爆炸})=\frac{1}{6}$$

$$p(\text{成人}\mid\text{不爆炸})=\frac{1}{3},\quad p(\text{小孩}\mid\text{不爆炸})=\frac{2}{3}$$

$$p(\text{成人}\mid\text{爆炸})=\frac{2}{3},\quad p(\text{小孩}\mid\text{爆炸})=\frac{1}{3}$$

$$p(\text{用手打}\mid\text{不爆炸})=\frac{5}{6},\quad p(\text{用脚踩}\mid\text{不爆炸})=\frac{1}{6}$$

$$p(\text{用手打}\mid\text{爆炸})=\frac{1}{6},\quad p(\text{用脚踩}\mid\text{爆炸})=\frac{5}{6}$$

那么在条件"紫色小气球、小孩用脚踩"下,知(注意可以忽略颜色和先验概率):

$$\hat{p}(\text{不爆炸})=p(\text{小气球}\mid\text{不爆炸})\times p(\text{小孩}\mid\text{不爆炸})\times p(\text{用脚踩}\mid\text{爆炸})=\frac{5}{54}$$

$$\hat{p}(\text{爆炸})=p(\text{小气球}\mid\text{爆炸})\times p(\text{小孩}\mid\text{爆炸})\times p(\text{用脚踩}\mid\text{爆炸})=\frac{5}{108}$$

所以确定认为,给定样本所导致的结果是"不爆炸"。

需要指出的是,该算法存在一个问题:如果训练集中某个类别 c_k 的数据没有涵盖第 j 维特征的第1个取值,相应估计的条件概率 $\hat{p}(X^{(j)}=a_{jl}\mid y=c_k)$ 就是0,从而导致模型可能会在测试集上的分类产生误差。解决这个问题的办法是在各个估计中加入平滑项(也有这种做法就是叫贝叶斯估计的说法),其过程为:

(1) 计算先验概率。

$$p_\lambda(y=c_k)=\frac{\sum_{i=1}^{N}I(y_i=c_k)+\lambda}{N+K\lambda},\quad k=1,2,\cdots,K$$

(2) 计算条件概率。

$$p_\lambda(X^{(j)}=a_{jl}\mid y=c_k)=\frac{\sum_{i=1}^{N}I(x_i^{(j)}a_{jl},y_i=c_k)+\lambda}{\sum_{i=1}^{N}I(y_i=c_k)+S_j\lambda}$$

$\lambda=0$ 时就是极大似然估计,$\lambda=1$ 时则叫作拉普拉斯平滑(Laplace Smoothing)。拉普拉斯平滑是常见的做法,在实现中也会默认使用它。将气球数据集1.0稍做变动以彰显加入平滑项的重要性。新数据集如表3-2所示。

表 3-2　气球数据集 1.5

颜　色	大　小	测试人员	测试动作	结　果
黄色	小	成人	用手打	不爆炸
黄色	小	成人	用脚踩	爆炸
黄色	小	小孩	用手打	不爆炸
黄色	小	小孩	用脚踩	爆炸
黄色	小	小孩	用脚踩	爆炸
黄色	小	小孩	用脚踩	爆炸
黄色	大	成人	用手打	爆炸
黄色	大	成人	用脚踩	爆炸
黄色	大	小孩	用手打	不爆炸
紫色	小	成人	用手打	不爆炸
紫色	小	小孩	用手打	不爆炸
紫色	大	小孩	用手打	不爆炸

该数据集的电子版本可以参见 https://github.com/cr=arefree0910/MachineLearning/blob/master/_Data/balloon1.5.txt。可以看到，这个数据集是“不太均衡”的：它对样本“黄色小气球，小孩用脚踩”重复进行了 3 次实验，而对所有紫色气球样本实验的结果都是“不爆炸”。如果此时想预测“紫色小气球，小孩用脚踩”的结果，虽然从直观上来说应该是“爆炸”，但我们会发现，此时由于

$$p(\text{用脚踩}\mid\text{不爆炸})=p(\text{紫色}\mid\text{爆炸})=0$$

所以会直接导致

$$\hat{p}(\text{不爆炸})=\hat{p}(\text{爆炸})=0$$

从而只能随机进行决策。此时加入平滑项就显得比较重要了，由拉普拉斯平滑，可知（注意类别的先验概率仍然不造成影响）：

$$p(\text{黄色}\mid\text{不爆炸})=\frac{3+1}{6+2},\quad p(\text{紫色}\mid\text{不爆炸})=\frac{3+1}{6+2}$$

$$p(\text{黄色}\mid\text{爆炸})=\frac{1+1}{6+2},\quad p(\text{紫色}\mid\text{爆炸})=\frac{0+1}{6+2}$$

$$p(\text{小气球}\mid\text{不爆炸})=\frac{4+1}{6+2},\quad p(\text{大气球}\mid\text{不爆炸})=\frac{2+1}{6+2}$$

$$p(\text{小气球}\mid\text{爆炸})=\frac{4+1}{6+2},\quad p(\text{大气球}\mid\text{爆炸})=\frac{2+1}{6+2}$$

$$p(\text{成人}\mid\text{不爆炸})=\frac{2+1}{6+2},\quad p(\text{小孩}\mid\text{不爆炸})=\frac{4+1}{6+2}$$

$$p(\text{成人}\mid\text{爆炸})=\frac{3+1}{6+2},\quad p(\text{小孩}\mid\text{爆炸})=\frac{3+1}{6+2}$$

$$p(\text{用手打}\mid\text{不爆炸})=\frac{6+1}{6+2},\quad p(\text{用脚踩}\mid\text{不爆炸})=\frac{0+1}{6+2}$$

$$p(\text{用手打}\mid\text{爆炸})=\frac{1+1}{6+2},\quad p(\text{用脚踩}\mid\text{爆炸})=\frac{5+1}{6+2}$$

从而可算得：

$$\hat{p}(\text{不爆炸})=\frac{25}{1024},\quad \hat{p}(\text{爆炸})=\frac{25}{512}$$

因此，我们确实应该认为给定样本所导致的结果是“爆炸”。

接着我们来看看如何进行 3 种模型的实现。考虑到代码重用和可拓展性，需要搭建一个基本架构，它应该定义好 3 种模型都会用到的通用功能，例如：

- 定义获取训练集里类别先验概率的函数；
- 将核心训练步骤以外的训练步骤进行定义，其中核心训练步骤需要训练出一个决策函数，该决策函数能够输出给定数据的后验概率；
- 利用决策函数定义预测函数和评估函数。

我们先来看看这个架构的基本框架：

```
import numpy as np
#定义好贝叶斯模型的基类,方便以后的拓展
class NaiveBayes(ClassifierBase):
 """
 self._x,self._y: 记录训练集的变量
 self._data: 核心数组,存储实际使用的条件概率的相关信息
 self._func: 模型核心——决策函数,能够根据输入的 x、y 输出对应的后验概率
 self._n_possibilities: 记录各个维度特征取值个数的数组:[S1,S2,…,SN]
 self._labelled_x: 记录按类别分开后的输入数据的数组
 self._label_zip: 记录类别相关信息的数组,视具体算法,定义会有所不同
 self._cat_counter: 核心数组,记录第 i 类数据的个数(cat 是 category 的缩写)
 self._con_counter: 核心数组,用于记录数据条件概率的原始极大似然估计
 self._label_dic: 核心字典,用于记录数值化类别时的转换关系
 self._feat_dice: 核心字典,用于记录数值化各维度特征(feat)时的转换关系
 """
   NaiveBayesTiming = Timing()
   def __init__(self, **kwargs):
        super(NaiveBayes, self).__init__(**kwargs)
        self._x = self._y = None
        self._data = self._func = None
        self._n_possibilities = None
        self._labelled_x = self._label_zip = None
        self._cat_counter = self._con_counter = None
        self.label_dict = self._feat_dicts = None
        self._params["lb"] = kwargs.get("lb", 1)
        #重载_getitem_运算符以避免定义大量 property
        def_getitem_(self,item):
            if isinstance(item,str):
                return getattr(self,"_" + item)
        #留下抽象方法让子类定义,这里的 tar_idx 参数和 self._tar_idx 的意义一致
        def feed_data(self,x,y,sample_weight = None):
            pass
```

```
        #留下抽象方法让子类定义,这里的 sample_weight 参数代表着样本权重
        def feed_sample_weight(self,sample_weight = None):
            pass
```

注意：让模型支持输入样本权重，更多的是为了使模型能够应用在提升方法中。这里只说一个直观理解：样本权重体现了各个样本的“重要性”。

上面这些代码定义的基本框架会在本书很多算法中出现，对于相同的结构，不会再进行详尽的相关注释。同样，即使是在接下来介绍的朴素贝叶斯相关算法的实现中，也有不少是具有普适性的。

```
#定义具有普适性的训练函数
@NaiveBayesTiming.timeit(level = 2, prefix = "[API] ")
def fit(self, x = None, y = None, sample_weight = None, lb = None):
    if sample_weight is None:
        sample_weight = self._params["sample_weight"]
    if lb is None:
        lb = self._params["lb"]
    #如果没有传入 x,y,那么就用传入的 x,y 的初始化模型
    if x is not None and y is not None:
        self.feed_data(x, y, sample_weight)
    #调用核心算法得到决策函数
    self._func = self._fit(lb)
#留下抽象核心算法让子类定义
def _fit(self, lb):
    pass
```

以上是模型训练相关的过程，下面就是模型的预测和评估过程。由浅入深，我们先进行“朴素的”实现。

```
#定义预测单一样本的函数
#参数 get_raw_result 控制该函数是输出预测的类别还是输出相应的后验概率
#get_raw_result = False 则输出类别,get_raw_result = True 则输出后验概率
    @NaiveBayesTiming.timeit(level = 1, prefix = "[API] ")
    def predict_one(self, x, get_raw_result = False):
        #在进行预测之前,要先把新的输入数据数值化
        #如果输入的是 Numpy 数组,要先将它转换成 Python 的数组
        #这是因为 Python 数组在数值化这个操作上要更快
            if type(x) is np.ndarray:
                x = x.tolist()
            #否则,复制数组
            else:
                x = x[:]
            #调用相关方法进行数值化,该方法随具体模型的不同而不同
            x = self._transfer_x(x)
            m_arg, m_probability = 0, 0
            #遍历各类别、找到能使后验概率最大化的类别
            for i in range(len(self._cat_counter)):
                p = self._func(x, i)
                if p > m_probability:
```

```
                m_arg, m_probability = i, p
            if not get_raw_result:
                return self.label_dict[m_arg]
            return m_probability
    #定义预测多样本的函数,本质是不断调用上面定义的 predict_one 函数
        @NaiveBayesTiming.timeit(level=3, prefix="[API] ")
        def predict(self, x, get_raw_result=False, **kwargs):
            return np.array([self.predict_one(xx, get_raw_result) for xx in x])
    #定义能对新数据进行评估的方法,这里暂以简单地输出准确率作为演示
        def evalute(self, x,y):
        y_pred=self.predict(x)
        print("Acc:{:12.6}%".format(100*np.sum(y_pred==y)/len(y)))
```

注意：之所以称上述实现是“朴素的”，是因为预测单一样本的函数只是在算法没有向量化时的一个临时产物。在算法完成向量化后，模型就能进行批量预测，该函数就可以删去了。

3.3.2 朴素贝叶斯分类算法实现二分类

朴素贝叶斯是一种有监督的分类算法，可以进行二分类或者多分类。

【例 3-4】 一个数据集实例如表 3-3 所示。现在有一个新的样本，$X=$(年龄：<=30，收入：中，是否学生：是，信誉：中)，目标是利用朴素贝叶斯分类来进行分类。

表 3-3 数据集实例

编号	描述属性				类别属性
	年龄	收入	学生	信誉	购买计算机
1	≤30	高	否	中	否
2	≤30	高	否	优	否
3	31～40	高	否	中	是
4	>40	中	否	中	是
5	>40	低	是	中	是
6	>40	低	是	优	否
7	31～40	低	是	优	是
8	≤30	中	否	中	否
9	≤30	低	是	中	是
10	>40	中	是	中	是
11	≤30	中	是	优	是
12	31～40	中	否	优	是
13	31～40	高	是	中	是
14	>40	中	否	优	否

假设类别为 $C(c_1=$是，$c_2=$否)，我们的目标是求出 $p(c_1|X)$和 $p(c_2|X)$，比较谁更大，就将 X 分为某个类。

可以将这个实例中的描述属性和类别属性与公式对应起来，然后计算。

描述属性 $A=\{A_1,A_2,A_3,A_4\}=\{$年龄，收入，学生与否，信誉$\}$

A_1 取值分别为 $a_{11}=$'<=30'，$a_{12}=$'30～40'，$a_{13}=$'>40'

A_2 取值分别为 $a_{21}=$'低'，$a_{22}=$'中'，$a_{23}=$'高'

A_3 取值分别为 $a_{31}=$'是'，$a_{32}=$'否'

A_4 取值分别为 $a_{41}=$'中'，$a_{42}=$'优'

类别属性 C：$c_1=$'是'，$c_2=$'否'

样本可以表示为 $X=(a_{11},a_{22},a_{31},a_{41})$，待求的概率分别是

$$p(c_1X)=p(c_1)p(a_{11}\mid c_1)p(a_{22}\mid c_1)p(a_{31}\mid c_1)p(a_{41}\mid c_1)$$

$$p(c_2X)=p(c_2)p(a_{11}\mid c_2)p(a_{22}\mid c_2)p(a_{31}\mid c_2)p(a_{41}\mid c_2)$$

计算过程：

$$p(c_1)=\frac{9}{14},p(a_{11}\mid c_1)=\frac{2}{9},p(a_{22}\mid c_1)=\frac{4}{9},p(a_{31}\mid c_1)=\frac{6}{9},p(a_{41}\mid c_1)=\frac{6}{9}$$

$$p(c_2)=\frac{5}{14},p(a_{11}\mid c_2)=\frac{3}{5},p(a_{22}\mid c_2)=\frac{2}{5},p(a_{31}\mid c_2)=\frac{1}{5},p(a_{41}\mid c_2)=\frac{2}{5}$$

所以，

$$p(c_1X)=\frac{9}{14}\times\frac{2}{9}\times\frac{4}{9}\times\frac{5}{9}\times\frac{5}{9}\approx 0.02821$$

$$p(c_2X)=\frac{5}{14}\times\frac{3}{5}\times\frac{2}{5}\times\frac{1}{5}\times\frac{2}{5}\approx 0.006857$$

由 $p(c_1X)>p(c_2X)$可知，样本 X 将被分类为$c_1=$'是'，会购买计算机。

利用 Python 编写上述实例对应的代码为：

```
#针对"买计算机"实例进行朴素贝叶斯分类
if __name__ == '__main__':
    #描述属性分别用数字替换
    #年龄, <=30 -->0, 31～40 -->1, >40 -->2
    #收入, '低'-->0, '中'-->1, '高'-->2
    #是否学生, '是'-->0, '否'-->1
    #信誉: '中'-->0, '优'-->1
    #类别属性用数字替换
    #是否购买计算机是-->0, 否-->1
    MAP = [{'<=30': 0, '31～40': 1, '>40': 2},
           {'低': 0, '中': 1, '高': 2},
           {'是': 0, '否': 1},
           {'中': 0, '优': 1},
           {'是': 0, '否': 1}]
    #训练样本
    train_samples = ["<=30 高否中否",
                     "<=30 高否优否",
                     "31～40 高否中是",
                     ">40 中否中是",
                     ">40 低是中是",
```

```
                    "> 40 低是优否",
                    "31～40 低是优是",
                    "<= 30 中否中否",
                    "<= 30 低是中是",
                    "> 40 中是中是",
                    "<= 30 中是优是",
                    "31～40 中否优是",
                    "31～40 高是中是",
                    "> 40 中否优否"]
#下面步骤将文字转化为对应数字
train_samples = [sample.split(' ') for sample in train_samples]
#print(train_samples)
#exit()
train_samples = [[MAP[i][attr] for i, attr in enumerate(sample)]for sample in train_samples]
#print(train_samples)
#待分类样本
X = '<= 30 中是中'
X = [MAP[i][attr] for i, attr in enumerate(X.split(' '))]
#训练样本数量
n_sample = len(train_samples)
#单个样本的维度：描述属性和类别属性个数
dim_sample = len(train_samples[0])
#计算每个属性有哪些取值
attr = []
for i in range(0, dim_sample):
    attr.append([])
for sample in train_samples:
    for i in range(0, dim_sample):
        if sample[i] not in attr[i]:
            attr[i].append(sample[i])
#每个属性取值的个数
n_attr = [len(attr) for attr in attr]
#记录不同类别的样本个数
n_c = []
for i in range(0, n_attr[dim_sample - 1]):
    n_c.append(0)
#计算不同类别的样本个数
for sample in train_samples:
    n_c[sample[dim_sample - 1]] += 1
#计算不同类别样本所占概率
p_c = [n_cx / sum(n_c) for n_cx in n_c]
#print(p_c)
#将用户按照类别分类
samples_at_c = {}
for c in attr[dim_sample - 1]:
    samples_at_c[c] = []
for sample in train_samples:
    samples_at_c[sample[dim_sample - 1]].append(sample)
#记录每个类别的训练样本中,取待分类样本的某个属性值的样本个数
n_attr_X = {}
```

```
for c in attr[dim_sample - 1]:
    n_attr_X[c] = []
    for j in range(0, dim_sample - 1):
        n_attr_X[c].append(0)
#计算每个类别的训练样本中,取待分类样本的某个属性值的样本个数
for c, samples_at_cx in zip(samples_at_c.keys(), samples_at_c.values()):
    for sample in samples_at_cx:
        for i in range(0, dim_sample - 1):
            if X[i] == sample[i]:
                n_attr_X[c][i] += 1
#字典转化为 list
n_attr_X = list(n_attr_X.values())
#print(n_attr_X)
#存储最终的概率
result_p = []
for i in range(0, n_attr[dim_sample - 1]):
    result_p.append(p_c[i])
#计算概率
for i in range(0, n_attr[dim_sample - 1]):
    n_attr_X[i] = [x/n_c[i] for x in n_attr_X[i]]
    for x in n_attr_X[i]:
        result_p[i] * = x
print('概率分别为', result_p)
#找到概率最大对应的那个类别,就是预测样本的分类情况
predict_class = result_p.index(max(result_p))
print(predict_class)
```

运行程序,输出如下:

```
概率分别为 [0.0011757789535567313, 0.16457142857142862]
1
```

输出结果表明:样本被分为第一类,即会购买计算机。对应的概率与手动计算的结果相同。

3.3.3　贝叶斯算法实现垃圾邮件分类

实例所讲解的是如何通过 Python 将文本读取,并且将每一个文本生成对应的词向量并返回。实例的背景是对 30 封邮件(包含 15 封正常邮件、15 封垃圾邮件)通过贝叶斯算法进行分类。

主要分为如下几个部分:

(1) 读取所有邮件;

(2) 建立词汇表;

(3) 生成每封邮件对应的词向量(词集模型);

(4) 用 sklearn 中的朴素贝叶斯算法进行分类;

(5) 生成性能评估报告。

下面先介绍需要用到的功能函数。思路：用所给的文本建立一个词汇表，就是将用所有出现的单词构成一个不重复的集合，即不含同一个单词。

```
def createVocabList(dataSet):
    vocabSet = set([])                          #创建空集
    for document in dataSet:
        vocabSet = vocabSet | set(document)  #两个并集
    return list(vocabSet)
postingList = [['my', 'dog', 'dog','has']]
print(createVocabList(postingList))
>>>['has', 'my', 'dog']
```

将所有的大写字母转换成小写字母，并且去掉长度小于两个字符的单词：

```
def textParse(bigString):                       #输入一个大字符串,输出是单词列表
    import re
    listOfTokens = re.split(r'\W*', bigString)
    return [tok.lower() for tok in listOfTokens if len(tok) > 2]
                                                #去掉长度小于两个字符的单词,2可以自己调节
s = 'i Love YYUU'
print(textParse(s))
>> ['love', 'yyuu']
```

构建词向量有两种方式：第一种是用文本中出现的单词，同词汇表向量进行对比，如果出现在词汇表中，则对应位置为1，反之为0。这种方式只管有无出现，不管出现次数，称为词集模型(set-of-words model)；第二种是同时统计出现次数，称为词袋模型(bag-of-words model)。

```
def setOfWords2Vec(vocabList, inputSet):
    returnVec = [0]*len(vocabList)
    for word in inputSet:
        if word in vocabList:
            returnVec[vocabList.index(word)] = 1
        else: print( "the word: %s is not in my Vocabulary!" % word)
    return returnVec
vocabulary = ['wo','do','like','what','go']
text = ['do','go','what','do']
print(setOfWords2Vec(vocabulary,text))
>> [0, 1, 0, 1, 1]
def bagOfWords2Vec(vocabList, inputSet):
    returnVec = [0]*len(vocabList)
    for word in inputSet:
        if word in vocabList:
            returnVec[vocabList.index(word)] += 1
        else: print("the word: %s is not in my Vocabulary!" % word)
    return returnVec
vocabulary = ['wo','do','like','what','go']
text = ['do','go','what','do']
print(setOfWords2Vec(vocabulary,text))
>> [0, 2, 0, 1, 1]
```

将上面 3 个函数写在一起；下面的操作方式只是针对本例，但是只要稍做修改同样能够适用于其他场合。

```
def createVocabList(dataSet):                    #建立词汇表
    vocabSet = set([])                           #创建空集
    for document in dataSet:
        vocabSet = vocabSet | set(document)      #两个并集
    return list(vocabSet)
def setOfWords2Vec(vocabList, inputSet):         #建立词向量
    returnVec = [0] * len(vocabList)
    for word in inputSet:
        if word in vocabList:
            returnVec[vocabList.index(word)] = 1
        else: print("the word: %s is not in my Vocabulary!" % word)
    return returnVec
def textParse(bigString):                        #输入一个大字符串,输出是单词列表
    import re
    listOfTokens = re.split(r'\W*', bigString)
    return [tok.lower() for tok in listOfTokens if len(tok) > 2]
def preProcessing():
    docList=[]; classList = []; fullText =[]
    for i in range(1,26):
        wordList = textParse(open('email/spam/%d.txt' % i).read())
        docList.append(wordList)                 #读取文本
        classList.append(1)                      #读取每个文本的标签
        wordList = textParse(open('email/ham/%d.txt' % i).read())
        docList.append(wordList)
        classList.append(0)
    vocabList = createVocabList(docList)#create vocabulary       #生成词向表
    data = []
    target = classList
    for docIndex in range(30):                   #本例一共有 30 个文本
        data.append(setOfWords2Vec(vocabList,docList[docIndex]))生成词向量
    return data,target                           #返回处理好的词向量和标签
```

对数据进行训练并预测：

```
import textProcess as tp
from sklearn.naive_bayes import MultinomialNB
from sklearn.cross_validation import train_test_split
from sklearn.metrics import classification_report
data,target= tp.preProcessing()
X_train,X_test,y_train,y_test = train_test_split(data,target,test_size=0.25)
mnb = MultinomialNB()
mnb.fit(X_train,y_train)
y_pre = mnb.predict((X_test))
print (y_pre)                                    #预测结果
print (y_test)                                   #实际结果
print ('The accuracy of Naive Bayes Classifier is',mnb.score(X_test,y_test))
print (classification_report(y_test,y_pre))
```

3.3.4 MultinomialNB 的实现

对于离散型朴素贝叶斯模型的实现，由于核心算法都是在进行“计数”工作，所以问题的关键就转换为如何进行计数。幸运的是，Numpy 中的一个方法——bincount 就是专门用来计数的，它能够非常快速地数出一个数组中各个数字出现的频率；而且由于它是 Numpy 自带的方法，其速度比 Python 标准库 collections 中的计数器 Counter 还要快上许多。不幸的是，该方法有两个缺点：

- 只能处理非负整数型的数组；
- 向量中的最大值即为返回的数组的长度，换句话说，如果用 bincount 方法对一个长度为 1、元素为 1000 的数组计数，返回的结果就是 999 个 0 加 1 个 1。

所以在做数据预处理时就要充分考虑到这两点，具体代码为：

```
#导入基本架构 Basic
from b_NaiveBayes.Original.Basic import *
class MultinomialNB(NaiveBayes):
    #定义预处理数据的方法
    def feed_data(self, x, y, sample_weight = None):
      #分情况将输入向量 x 进行转置
      if isinstance(x,list):
            features = map(list,zip( * x))
      else:
            features = x.T
      #利用 Python 中内置的高级数据结构——集合，获取各个维度的特征和类别
      #为了利用 bincount 方法来优化算法，将所有特征从 0 开始数值化
      #注意：需要将数值化过程中的转换关系记录成字典，否则无法对新数据进行判断
      features = [set(feat) for feat in features]
      feat_dics = [{_1:i for i,_1 in enumerate(feats)} for feats in features]
      label_dic = {_1:i for i, _1 in enumerate(set(y))}
      #利用转换字典更新训练集
      x = np.array([  [  feat_dics[i][_1] for i,_1 in enumerate(sample)]  for sample in features])
      y = np.array([  label_dic[yy] for yy in y])
      #利用 Numpy 中的 bincount 方法，获得各类别的数据的个数
          cat_counter = np.bincount(y)
          #记录各维度特征的取值个数
          n_possibilities = [len(feats) for feats in features]
          #获取各类别数据的下标
          labels = [y == value for value in range(len(cat_counter))]
          #利用下标获取记录按类别分开后的输入数据的数组
          labelled_x = [x[ci].T for ci in labels]
          #更新模型的各个属性
          self._x, self._y = x, y
          self._labelled_x, self._label_zip = labelled_x, list(zip(labels, labelled_x))
          self._cat_counter, self._feat_dicts, self._n_possibilities = cat_counter, feat_
dicts, n_possibilities
          self.label_dict = label_dict
          #调用处理样本权重的函数，以更新记录条件概率的数组
```

```
            self.feed_sample_weight(sample_weight)
        #定义处理样本权重的函数
        def feed_sample_weight(self, sample_weight = None):
            self._con_counter = []
            #利用 Numpy 的 bincount 方法获取带权重的条件概率的极大似然估计
            for dim, p in enumerate(self._n_possibilities):
                if sample_weight is None:
                    self._con_counter.append([
                        np.bincount(xx[dim], minlength = p) for xx in self._labelled_x])
                else:
                    local_weights = sample_weight * len(sample_weight)
                    self._con_counter.append([
                        np.bincount(xx[dim], weights = local_weights[label], minlength = p)
                        for label, xx in self._label_zip])
```

注意：这样做确实会让训练过程加速很多，但是同时也会使预测过程的速度下降一些（因为预测时要先将输入数据数值化）；视具体情况的不同，数据预处理部分的实现可以有所不同。

下面的核心函数就变为调用与整合数据预处理时记录下来的信息的过程：

```
#定义核心训练函数
def _fit(self, lb):
    n_dim = len(self._n_possibilities)
    n_category = len(self._cat_counter)
    p_category = self.get_prior_probability(lb)
    #data 即为存储加了平滑项后的条件概率的数组
    data = [[] for _ in range(n_dim)]
    for dim, n_possibilities in enumerate(self._n_possibilities):
        data[dim] = [
            [(self._con_counter[dim][c][p] + lb) / (self._cat_counter[c] + lb * n_possibilities)
             for p in range(n_possibilities)] for c in range(n_category)]
    self._data = [np.asarray(dim_info) for dim_info in data]
    #利用 data 生成决策函数
    def func(input_x, tar_category):
        rs = 1
        #遍历各个维度，利用 data 和条件独立性假设计算联合条件概率
        for d, xx in enumerate(input_x):
            rs *= data[d][tar_category][xx]
            #利用先验概率和联合条件概率计算后验概率
        return rs * p_category[tar_category]
  #返回决策函数
    return func
#定义数值化数据的函数
def _transfer_x(self, x):
    for j, char in enumerate(x):
        x[j] = self._feat_dicts[j][char]
    return x
```

至此，第一个通用的朴素贝叶斯模型就完全搭建完毕了，可以用之前的气球数据集

1.0、1.5来简单地评估我们的模型。首先要定义一个能够将文件的数据转化为Python数组的类：

```
def get_dataset(name, path, n_train = None, tar_idx = None, shuffle = True,
                quantize = False, quantized = False, one_hot = False, ** kwargs):
    x = []
    #将编码设为utf8以便读入中文等特殊字符
    with open(path, "r", encoding = "utf8") as file:
        if DataUtil.is_naive(name):
            #如果是气球数据集的话,直接依逗号分隔数据即可
            if "balloon" in name:
                for sample in file:
                    x.append(sample.strip().split(","))
#默认打乱数据
    if shuffle:
        np.random.shuffle(x)
    #默认类别在最后一列
    tar_idx = -1 if tar_idx is None else tar_idx
    y = np.array([xx.pop(tar_idx) for xx in x])
    x = np.array(x)
    #默认全部都是训练样本
    if quantized:
        return x, y
#如果传入了训练样本数,则依之将数据集切分为训练集和测试集
    return(x[:n_train], y[:n_train]), (x[n_train:], y[n_train:])
```

以下为MultinomialNB的评估代码：

```
if __name__ == '__main__':
    #导入标准库time以计时,导入DataUtil类以获取数据
    from Util import DataUtil
    import time
    #遍历1.0、1.5两个版本的气球数据集
    for dataset in ("balloon1.0", "balloon1.5"):
     #读入数据
        _x, _y = DataUtil.get_dataset(dataset, "../../_Data/{}.txt".format(dataset))
        #实例化模型并进行训练,同时记录整个过程花费的时间
        learning_time = time.time()
        nb = MultinomialNB()
        nb.fit(_x, _y)
        learning_time = time.time() - learning_time
        print("=" * 30)
        print(dataset)
        print("-" * 30)
        #评估模型的表现,同时记录评估过程花费的时间
        estimation_time = time.time()
        nb.evaluate(_x, _y)
        estimation_time = time.time() - estimation_time
        #将记录下来的耗时输出
        print(
```

```
            "Model building  : {:12.6} s\n"
            "Estimation      : {:12.6} s\n"
            "Total           : {:12.6} s".format(
                learning_time, estimation_time,
                learning_time + estimation_time
            )
        )
```

运行以上代码，得到结果：

```
balloo1.0
Acc:      100.0 %
Model building  :   0.0s
Estimation      :   0.0s
Total           :   0.0s
--------------------------------------------------
balloo1.5
Acc:      91.6667 %
Model building  :   0.0s
Estimation      :   0.0s
Total           :   0.0s
```

由于数据量太少，所以建模和评估的过程耗费的时间已是可以忽略不计的程度。气球数据集 1.5 是“不太均衡”的数据集，所以朴素贝叶斯在其上的表现会比较差。

仅仅在一个虚构的数据集上进行评估可能不太有说服力，下面通过自带的 digits 数据来评估我们的模型。

【例 3-5】 通过 digits 数据来评估模型。

(1) 导入必要的库。

```
from sklearn import datasets,cross_validation,naive_bayes
import numpy as np
import matplotlib.pyplot as plt
```

(2) 显示 Digit Dataset 数据集。

```
def show_digits():
    digits = datasets.load_digits()
    fig = plt.figure()
    print("vector from images 0:",digits.data[0])
    for i in range(25):
        ax = fig.add_subplot(5,5,i + 1)
        ax.imshow(digits.images[i],cmap = plt.cm.gray_r,interpolation = 'nearest')
    plt.show()          #效果如图 3-2 所示
show_digits()
vector from images 0: [ 0.   0.   5. 13.   9.   1.   0.   0.   0.   0. 13. 15. 10. 15.   5.   0.   0.   3.
 15.   2.   0. 11.   8.   0.   0.   4. 12.   0.   0.   8.   8.   0.   0.   5.   8.   0.
  0.   9.   8.   0.   0.   4. 11.   0.   1. 12.   7.   0.   0.   2. 14.   5. 10. 12.
  0.   0.   0.   0.   6. 13. 10.   0.   0.   0.]
```

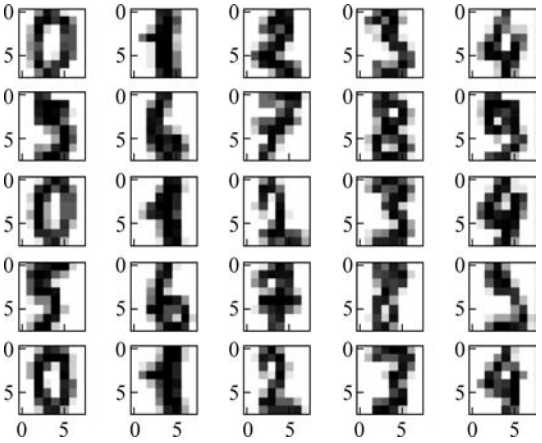

图 3-2　Digit Dataset 数据集

（3）加载数据。

```
def load_data():
    digits = datasets.load_digits()
    return cross_validation.train_test_split(digits.data,digits.target,test_size = 0.25,
random_state = 0)
```

（4）测试多项式贝叶斯分类器。

```
def test_MultinomialNB(*data):
    X_train,X_test,y_train,y_test = data
    cls = naive_bayes.MultinomialNB()
    cls.fit(X_train,y_train)
    print('Training Score: %.2f' % cls.score(X_train,y_train))
    print('Testing Score: %.2f' % cls.score(X_test,y_test))
X_train,X_test,y_train,y_test = load_data()
test_MultinomialNB(X_train,X_test,y_train,y_test)
```

（5）检验不同的 a 对多项式贝叶斯分类器的预测性能的影响。

```
def test_MultinomialNB_alpha(*data):
    X_train,X_test,y_train,y_test = data
    alphas = np.logspace(-2,5,num = 200)
    train_scores = []
    test_scores = []
    for alpha in alphas:
        cls = naive_bayes.MultinomialNB(alpha = alpha)
        cls.fit(X_train,y_train)
        train_scores.append(cls.score(X_train,y_train))
        test_scores.append(cls.score(X_test,y_test))
    #绘图
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.plot(alphas,train_scores,label = "Training Score")
```

```
    ax.plot(alphas,test_scores,label = "Testing Score")
    ax.set_xlabel(r" $ alpha $ ")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.0)
    ax.set_title("MultinomialNB")
    ax.set_xscale("log")
    plt.show()        #效果如图 3-3 所示
X_train,X_test,y_train,y_test = load_data()
test_MultinomialNB_alpha(X_train,X_test,y_train,y_test)
Training Score:0.91
Testing Score:0.91
```

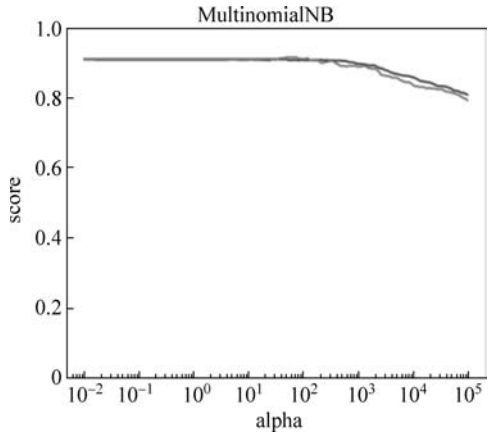

图 3-3　MultinomialNB 预测性能曲线图

3.3.5　GaussianNB 的实现

有了实现离散朴素贝叶斯的经验，就可以触类旁通地实现连续朴素贝叶斯模型了。

处理连续型变量有一个最直观的方法：使用小区间切割，直接使其离散化。由于这种方法较难控制小区间的大小，而且对训练集质量的要求比较高，所以选用第二种方法：假设该变量服从正态分布（或称高斯分布，Gaussian Distribution），再利用极大似然估计来计算该变量的“条件概率”（仅展示和离散型算法中不同的部分），具体过程为：

（1）与离散型算法一致。

（2）计算“条件概率”$p(X^{(j)}=a_{jl}\mid y=c_k)$：

$$p(X^{(j)}=a_{jl}\mid y=c_k)=\frac{1}{\sqrt{2\pi}\sigma_{jk}}\mathrm{e}^{-\frac{(a_{jl}-\mu_{jk})^2}{2\sigma_{jk}^2}}$$

这里有两个参数 μ_{jk}、σ_{jk}，它们可以用极大似然估计法定义：

$$\hat{\mu}_{jk}=\frac{1}{N_k}\sum_{i=1}^{N}x_i^{(j)}I(y_i=c_k)$$

$$\sigma_{jk}^2 = \frac{1}{N_k}\sum_{i=1}^{N}(x_i^{(j)} - \mu_{jk})^2 I(y_i = c_k)$$

注意：这里的“条件概率”其实是“条件概率密度”，真正的条件概率其实是0（因为连续型变量单点概率为0）。这样做的合理性涉及比较深的概率论知识，此处不介绍。

所以在实现GaussianNB之前，需要先实现一个能够计算正态分布密度和进行正态分布极大似然估计的类：

```
import numpy as np
from math import pi, exp
#记录 sqrt(2π),避免该项的重复运算
sqrt_pi = (2 * pi) ** 0.5
class NBFunctions:
    #定义正态分布的密度函数
    @staticmethod
    def gaussian(x, mu, sigma):
        return exp(-(x - mu) ** 2 / (2 * sigma ** 2)) / (sqrt_pi * sigma)
     #定义进行极大似然估计的函数
     #它能返回一个存储着计算条件概率密度的函数的列表
    @staticmethod
    def gaussian_maximum_likelihood(labelled_x, n_category, dim):
        mu = [np.sum(
            labelled_x[c][dim]) / len(labelled_x[c][dim]) for c in range(n_category)]
        sigma = [np.sum(
            (labelled_x[c][dim] - mu[c]) ** 2)/len(labelled_x[c][dim]) for c in range(n_category)]
    #利用极大似然估计得到的 μ 和 σ,定义生成计算条件概率密度的函数 func
        def func(_c):
            def sub(x):
                return NBFunctions.gaussian(x, mu[_c], sigma[_c])
            return sub
    #利用 func 返回目标列表
        return [func(_c = c) for c in range(n_category)]
```

由于算法中只有条件概率相关的定义变了，所以只需要将相关的函数重新定义即可。此外，由于输入数据肯定是数值数据，所以数据预处理会简单不少（至少不用因为要对输入进行特殊的数值化处理而记录其转换字典了）。考虑到MultinomialNB处的注释基本上把框架的思想都说明清楚了，因此在接下来的GaussianNB的代码实现中会适当减少注释。

【例3-6】 连续型朴素贝叶斯的实现。

```
from b_NaiveBayes.Original.Basic import *
class GaussianNB(NaiveBayes):
    GaussianNBTiming = Timing()
    def feed_data(self, x, y, sample_weight = None):
        #简单地调用 Python 自带的 float 方法将输入数据数值化
        x = np.array([list(map(lambda c: float(c), sample)) for sample in x])
        #数值化类别向量
        labels = list(set(y))
        label_dict = {label: i for i, label in enumerate(labels)}
```

```
        y = np.array([label_dict[yy] for yy in y])
        cat_counter = np.bincount(y)
        labels = [y == value for value in range(len(cat_counter))]
        labelled_x = [x[label].T for label in labels]
        #更新模型的各个属性
        self._x, self._y = x.T, y
        self._labelled_x, self._label_zip = labelled_x, labels
        self._cat_counter, self.label_dict = cat_counter, {i: l for l, i in label_dict.items()}
        self.feed_sample_weight(sample_weight)
```

可以看到,数据预处理这一步确实要轻松很多。接着只需要再定义训练用的代码就可以了,它们和 MultinomialNB 中的实现大同小异:

```
    #定义处理样本权重的函数
def feed_sample_weight(self, sample_weight = None):
    if sample_weight is not None:
        local_weights = sample_weight * len(sample_weight)
        for i, label in enumerate(self._label_zip):
            self._labelled_x[i] * = local_weights[label]
@GaussianNBTiming.timeit(level = 1, prefix = "[Core] ")
def _fit(self, lb):
    n_category = len(self._cat_counter)
    p_category = self.get_prior_probability(lb)
    #利用极大似然估计获得计算条件概率的函数,使用数组变量 data 进行存储
    data = [
        NBFunctions.gaussian_maximum_likelihood(
            self._labelled_x, n_category, dim) for dim in range(len(self._x))]
    self._data = data
    def func(input_x, tar_category):
        rs = 1
        for d, xx in enumerate(input_x):
        #由于 data 中存储的是函数,所以需要调用它来进行条件概率的计算
            rs * = data[d][tar_category](xx)
        return rs * p_category[tar_category]
    return func
```

由于数据本身就是数值的,所以数据转换函数只需直接返回输入值即可:

```
@staticmethod
def _transfer_x(x):
return x
```

至此,连续型朴素贝叶斯模型就搭建完毕,运行程序,输出如下:

```
Acc:     48.3333 %
Acc:     47.8343 %
Model building  :   0.0551443 s
Estimation      :   0.715473 s
Total           :   0.770618 s
```

可看到,建模的速度比 MultionialNB 要快,但是预测的速度非常慢(MultinomialNB 比

它快四五倍)。这是因为 GaussianNB 在预测时要进行大量正态分布密度的计算,而我们还没有进行算法的向量化。

连续型朴素贝叶斯同样能够进行和离散型朴素贝叶斯类似的可视化,下面用自带的 digits 数据来评估我们的模型。

【例 3-7】 利用高斯朴素贝叶斯模型实现光学字符识别。

光学字符识别问题:手写数字识别。简单地说,这个问题包括图像中字符的定位和识别两部分。为了演示方便,我们选择使用 Scikit-Learn 中自带的手写数字数据集。

(1) 加载并可视化手写数字。

首先用 Scikit-Learn 的数据获取接口加载数据,并简单统计一下:

```
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
>>> digits.images.shape
(1797, 8, 8)
>>> import matplotlib.pyplot as plt
>>> fig,axes = plt.subplots(10,10, figsize=(8, 8),subplot_kw={'xticks':[], 'yticks':[]},
gridspec_kw=dict(hspace=0.1, wspace=0.1))
    >>> for i, ax in enumerate(axes.flat):
     ... ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
     ...ax.text(0.05, 0.05, str(digits.target[i]),transform=ax.transAxes, color='green')
    >>> plt.show()
```

这份图像数据是一个三维矩阵:共有 1797 个样本,每张图像都是 8×8 像素。对前 100 张图进行可视化,如图 3-4 所示。

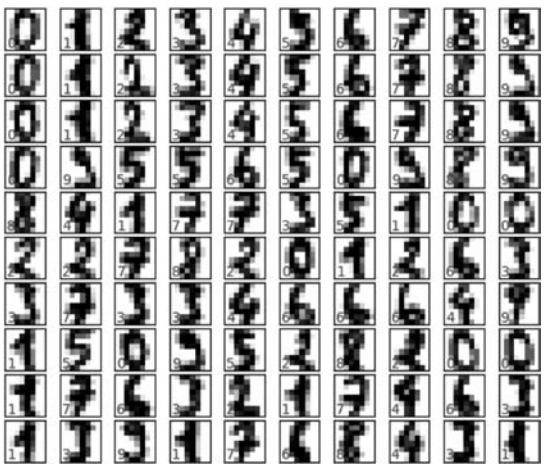

图 3-4 原始光学字符

为了在 Scikit-Learn 中使用数据,需要一个维度为[n_samples, n_features]的二维特征矩阵——可以将每个样本图像的所有像素都作为特征,也就是将每个数字的 8×8 像素平铺成长度为 64 的一维数组。另外,还需要一个目标数组,用来表示每个数字的真实值(标签)。

这两份数据已经放在手写数字数据集的 data 与 target 属性中，直接使用即可：

```
>>> X = digits.data
>>> X.shape
(1797, 64)
>>> y = digits.target
>>> y.shape
(1797,)
```

(2) 无监督学习：降维。

虽然想对具有 64 维参数空间的样本进行可视化，但是在如此高维度的空间中进行可视化十分困难。因此，需要借助无监督学习方法将维度降到二维。在此试试流形学习算法中的 Isomap 算法对数据进行降维：

```
>>> from sklearn.manifold import Isomap
>>> iso = Isomap(n_components = 2)
>>> iso.fit(digits.data)
Isomap(eigen_solver = 'auto', max_iter = None, n_components = 2, n_jobs = 1,
    n_neighbors = 5, neighbors_algorithm = 'auto', path_method = 'auto', tol = 0)
>>> data_projected = iso.transform(digits.data)
>>> data_projected.shape
(1797, 2)
```

现在数据已经投影到二维，把数据画出来，看看从结构中能发现什么：

```
>>> plt.scatter(data_projected[:, 0], data_projected[:, 1], c = digits.target,edgecolor = 'none',
alpha = 0.5,cmap = plt.cm.get_cmap('Spectral',10))
>>> plt.colorbar(label = 'digit label', ticks = range(10))
>>> plt.clim( - 0.5, 9.5)
>>> plt.show()            #效果如图 3-5 所示
```

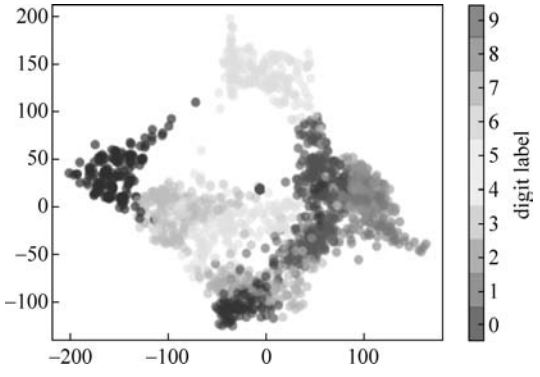

图 3-5　二维投影散点图

这幅图呈现出了非常直观的效果，让我们知道数字在 64 维空间中的分离(可识别)程度。虽然有些瑕疵，但从总体上看，各个数字在参数空间中的分离程度还是令人满意的。这其实告诉我们：用一个非常简单的有监督分类算法就可以完成任务。下面来演示一下。

(3) 数字分类。

我们需要找到一个分类算法,对手写数字进行分类。先将数据分成训练集和测试集,然后用高斯朴素贝叶斯模型来拟合:

```
>>> from sklearn.model_selection import train_test_split
>>> Xtrain,Xtest,ytrain,ytest = train_test_split(X,y,random_state = 0)
>>> from sklearn.naive_bayes import GaussianNB
>>> model = GaussianNB()
>>> model.fit(Xtrain,ytrain)
GaussianNB(priors = None)
>>> y_model = model.predict(Xtest)
```

模型预测已经完成,现在用模型在训练集中的正确识别样本量与总训练样本量进行对比,获得模型的准确率:

```
>>> from sklearn.metrics import accuracy_score
>>> accuracy_score(ytest,y_model)
0.8333333333333334
```

可以看出,通过一个非常简单的模型,数字识别率就可以达到 80% 以上。但仅依靠这个指标,我们无法知道模型哪里做得不够好,解决这个问题的办法就是用混淆矩阵(confusion matrix)。可以用 Scikit-Learn 计算混淆矩阵,然后用 Seaborn 画出来:

```
>>> from sklearn.metrics import confusion_matrix
>>> mat = confusion_matrix(ytest,y_model)
>>> import seaborn as sns
>>> sns.heatmap(mat, square = True, annot = True, cbar = False)
<matplotlib.axes._subplots.AxesSubplot object at 0x000001636AC15438>
>>> plt.xlabel('predicted value')
Text(0.5,64.2268,'predicted value')
>>> plt.ylabel('true value')
Text(633.841,0.5,'true value')
>>> plt.show()                    #效果如图 3-6 所示
```

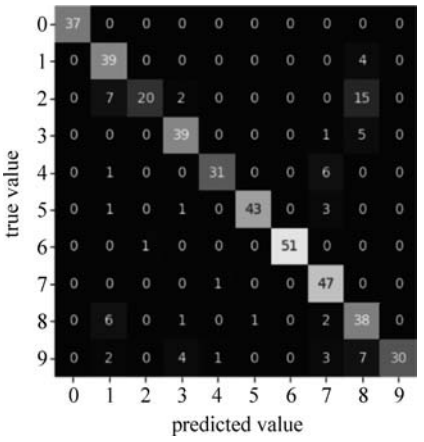

图 3-6 数字识别模型图

从图 3-6 可以看出，误判的主要原因在于许多数字 2 被误判成了数字 1 或数字 8。另一种显示模型特征的直观方式是将样本画出来，然后把预测标签放在左下角，用绿色表示预测正确，用红色表示预测错误：

```
>>> fig, axes = plt.subplots(10, 10, figsize = (8, 8),subplot_kw = {'xticks':[], 'yticks':[]},
gridspec_kw = dict(hspace = 0.1, wspace = 0.1))
>>> test_images = Xtest.reshape( - 1,8,8)
>>> for i, ax in enumerate(axes.flat):
 ax.imshow(test_images[i], cmap = 'binary', interpolation = 'nearest')
 ax.text(0.05, 0.05, str(y_model[i]),transform = ax.transAxes,color = 'green' if (ytest[i] ==
y_model[i]) else 'red')
>>> plt.show()                    # 效果如图 3-7 所示
```

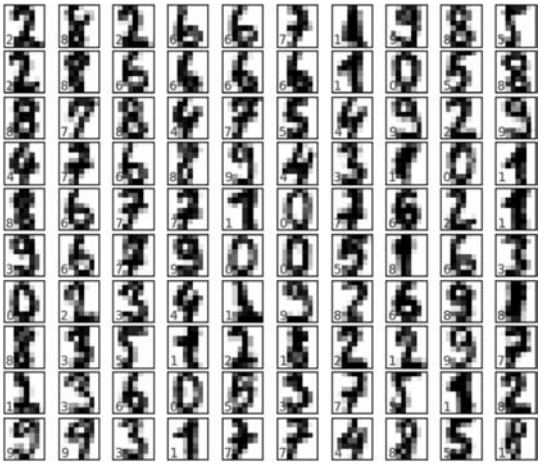

图 3-7　光学字符识别结果

彩色图片
图 3-7

3.3.6　MergedNB 的实现

混合型贝叶斯算法主要有两种提示：

- 用某种分布的密度函数算出训练集中各个样本连续型特征相应维度的密度之后，根据这些密度的情况将该维度离散化，最后再训练离散型朴素贝叶斯模型。
- 直接结合离散型朴素贝叶斯和连续型朴素贝叶斯：

$$y=f(x^{*})=\underset{c_k}{\operatorname{argmax}}p(y=c_k)\prod_{j\in S_1}p(X^{(j)}=x^{*(j)}=c_k)\prod_{j\in S_2}p(X^{(j)}=x^{*(j)}=c_k)$$

其中，S_1 和 S_2 代表离散、连续维度的集合，条件概率由 3.3.1 节及 3.3.5 节的算法给出。

可以直观看出，第二种提法可能会比第一种提法要“激进”一些，因为如果某个连续型维度采用的分布特别“大起大落”，那么该维度可能会直接“主导”整个决策。但是考虑到实现的简洁和直观，我们还演示了第二种提法的实现。

可以对气球数据集 1.0 稍做变动，将“气球大小”这个特征改为“气球直径”，然后再手动做一次分类，以加深对混合型朴素贝叶斯算法的理解。新数据集如表 3-4 所示。

表 3-4 气球数据集 2.0

颜 色	直 径	测试人员	测试动作	结 果
黄色	10	成人	用手打	不爆炸
黄色	15	成人	用脚踩	爆炸
黄色	9	小孩	用手打	不爆炸
黄色	9	小孩	用脚踩	不爆炸
黄色	19	成人	用手打	爆炸
黄色	21	成人	用脚踩	爆炸
黄色	16	小孩	用手打	不爆炸
黄色	22	小孩	用脚踩	爆炸
紫色	10	成人	用手打	不爆炸
紫色	12	小孩	用手打	不爆炸
紫色	22	成人	用脚踩	爆炸
紫色	21	小孩	用脚踩	爆炸

该数据集的电子版本可参见 https://github.com/carefree0910/MachineLearning/blob/master/_Data/balloon2.0.txt。我们需要预测的是样本：

紫色	10	小孩	用脚踩

除了“大小”变成“直径”，其余特征都一点未变，所以只需再计算直径的条件概率（密度）。由 GaussianNB 的算法可知：

$$\hat{\mu}_{\text{不爆炸}}=\frac{10+9+9+16+10+12}{6}=11$$

$$\hat{\mu}_{\text{爆炸}}=\frac{15+19+21+22+22+21}{6}=20$$

$$\hat{\sigma}_{\text{不爆炸}}=\frac{1}{6}[(10-\hat{\mu}_{\text{不爆炸}})^2+\cdots+(12-\hat{\mu}_{\text{不爆炸}})^2]=6$$

$$\hat{\sigma}_{\text{爆炸}}=\frac{1}{6}[(15-\hat{\mu}_{\text{爆炸}})^2+\cdots+(21-\hat{\mu}_{\text{爆炸}})^2]=6$$

从而，

$$\hat{p}(\text{不爆炸})=\frac{1}{\sqrt{2\pi}\hat{\sigma}_{\text{不爆炸}}}e^{-\frac{((10-\hat{\mu}_{\text{不爆炸}})^2)}{2\hat{\sigma}^2_{\text{不爆炸}}}}\times p(\text{小孩}\mid\text{不爆炸})\times p(\text{用脚踩}\mid\text{不爆炸})\approx 0.0073$$

$$\hat{p}(\text{爆炸})=\frac{1}{\sqrt{2\pi}\hat{\sigma}_{\text{爆炸}}}e^{-\frac{((10-\hat{\mu}_{\text{爆炸}})^2)}{2\hat{\sigma}^2_{\text{爆炸}}}}\times p(\text{小孩}\mid\text{爆炸})\times p(\text{用脚踩}\mid\text{爆炸})\approx 0.0046$$

因此应认为给定样本所导致的结果是“不爆炸”，这和直观感受大体相符。接着看一下具体如何实现：

```
from b_NaiveBayes.Original.Basic import *
from b_NaiveBayes.Original.MultinomialNB import MultinomialNB
from b_NaiveBayes.Original.GaussianNB import GaussianNB

class MergedNB(NaiveBayes):
    MergedNBTiming = Timing()
    """
```

初始化结构

```
    self._whether_discrete:记录各个维度的变量是否是离散型变量
    self._whether_continuous:记录各个维度的变量是否是连续型变量
    self._multinomial,self._gaussian:离散型、连续型朴素贝叶斯模型
    """
    def __init__(self, **kwargs):
        self._multinomial, self._gaussian = MultinomialNB(), GaussianNB()

        if wc is None:
            self._whether_discrete = self._whether_continuous = None
        else:
            self._whether_continuous = np.asarray(wc)
            self._whether_discrete = ~self._whether_continuous
```

对模型的训练进行实现，代码为：

```
  #分别利用 MultinomialNB 和 GaussianNB 的数据预处理方法进行数据处理
  def feed_data(self, x, y, sample_weight=None):
      if sample_weight is not None:
          sample_weight = np.asarray(sample_weight)
      x, y, wc, features, feat_dicts, label_dict = DataUtil.quantize_data(
          x, y, wc=self._whether_continuous, separate=True)
      #如果没有指定哪些维度连续,则用 quantize_data 中朴素的方法判定哪些维度连续
      if self._whether_continuous is None:
        #通过 Numpy 中对逻辑非的支持进行快速运算
          self._whether_continuous = wc
          self._whether_discrete = ~self._whether_continuous
      self.label_dict = label_dict
      discrete_x, continuous_x = x
      cat_counter = np.bincount(y)
      self._cat_counter = cat_counter
      labels = [y == value for value in range(len(cat_counter))]
    #训练离散型朴素贝叶斯
      labelled_x = [discrete_x[ci].T for ci in labels]
      self._multinomial._x, self._multinomial._y = x, y
       self._multinomial._labelled_x, self._multinomial._label_zip = labelled_x, list(zip
(labels, labelled_x))
      self._multinomial._cat_counter = cat_counter
      self._multinomial._feat_dicts = [dic for i, dic in enumerate(feat_dicts) if self._whether_
discrete[i]]
      self._multinomial._n_possibilities = [len(feats) for i, feats in enumerate(features)
                                            if self._whether_discrete[i]]
      self._multinomial.label_dict = label_dict
```

```
        #训练连续型朴素贝叶斯
        labelled_x = [continuous_x[label].T for label in labels]
        self._gaussian._x, self._gaussian._y = continuous_x.T, y
        self._gaussian._labelled_x, self._gaussian._label_zip = labelled_x, labels
        self._gaussian._cat_counter, self._gaussian.label_dict = cat_counter, label_dict
        #处理样本权重
        self.feed_sample_weight(sample_weight)
    #分别利用 MultinomialNB 和 GaussianNB 处理样本权重的方法来处理样本权重
    def feed_sample_weight(self, sample_weight = None):
        self._multinomial.feed_sample_weight(sample_weight)
        self._gaussian.feed_sample_weight(sample_weight)
    #分别利用 MultinomialNB 和 GaussianNB 的训练函数来进行训练
    def _fit(self, lb):
        self._multinomial.fit()
        self._gaussian.fit()
        p_category = self._multinomial.get_prior_probability(lb)
        discrete_func, continuous_func = self._multinomial["func"], self._gaussian["func"]
    #将 MultinomialNB 和 GaussianNB 的决策函数直接合成 MergedNB 的决策函数
    #由于这两个决策函数都乘了先验概率,所以需要除掉一个先验概率
        def func(input_x, tar_category):
            input_x = np.asarray(input_x)
            return discrete_func(
        input_x[self._whether_discrete].astype(np.int), tar_category) * continuous_func(
            input_x[self._whether_continuous], tar_category) / p_category[tar_category]
        return func
```

上述实现有一个显而易见的可以优化的地方：我们一共在代码中重复计算了 3 次先验概率，但其实只计算一次就可以了，考虑到这一点不是性能瓶颈，为了代码的连贯性和可读性，就没有进行这个优化。数据转换函数则相对而言要复杂一些，因为需要跳过连续维度，将离散维度挑出来进行数值化：

```
    #实现转换混合型数据的方法,要注意利用 MultinomialNB 的相应变量
    def _transfer_x(self, x):
        feat_dicts = self._multinomial["feat_dicts"]
        idx = 0
        for d, discrete in enumerate(self._whether_discrete):
            #如果是连续维度,直接调用 float 方法将其转为浮点数
            if not discrete:
                x[d] = float(x[d])
            else:
                #如果是离散维度,利用转换字典进行数值化
                x[d] = feat_dicts[idx][x[d]]
            if discrete:
                idx += 1
        return x
```

至此，混合型朴素贝叶斯模型就搭建完毕了。

3.3.7 BernoulliNB 分类器实现

BernoulliNB 是伯努利贝叶斯分类器，它假设特征的条件概率分布满足二项分布：

$$P(X^{(j)} \mid y=c_k)=pX^{(j)}+(1-p)(1-X^{(j)})$$

其中,要求特征的取值为 $X^{(j)}\in\{0,1\}$,且 $P(X^{(j)}=1|y=c_k)=p$。

【例 3-8】 用自带的 digits 数据来评估 BernoulliNB 分类器。

(1) 载入必要的库。

```
from sklearn import datasets,cross_validation,naive_bayes
import numpy as np
import matplotlib.pyplot as plt
```

(2) 观察 digit Dataset。

```
def show_digits():
    digits = datasets.load_digits()
    fig = plt.figure()
    print('vector from  images 0:',digits.data[0])
    for i in range(25):
         ax = fig.add_subplot(5,5,i+1)
         ax.imshow(digits.images[i],cmap=plt.cm.gray_r,interpolation='nearest')
    plt.show()
show_digits()
vector from  images 0:[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.  0.  0.  3.
 15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.  8.  0.
  0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 10. 12.
  0.  0.  0.  0.  6. 13. 10.  0.  0.  0.]
```

(3) 加载数据集。

```
def load_data():
    digits = datasets.load_digits()
    return cross_validation.train_test_split(digits.data,digits.target,test_size=0.25,
random_state=0)
```

(4) 伯努利贝叶斯分类器。

```
def test_BernoulliNB(*data):
    X_train,X_test,y_train,y_test = data
    cls = naive_bayes.BernoulliNB()
    cls.fit(X_train,y_train)
    print("Training score:%.2f" % cls.score(X_train,y_train))
    print("Testing score:%.2f" %cls.score(X_test,y_test))
X_train,X_test,y_train,y_test = load_data()
test_BernoulliNB(X_train,X_test,y_train,y_test)
Training score:0.87
Testing score:0.85
```

(5) 检验不同的 a 对伯努利贝叶斯分类器的预测性能的影响。

```
def test_BernoulliNB_alpha(*data):
    X_train,X_test,y_train,y_test = data
    alphas = np.logspace(-2,5,num=200)
    train_scores = []
```

```
    test_scores = []
    for alpha in alphas:
        cls = naive_bayes.BernoulliNB(alpha = alpha)
        cls.fit(X_train,y_train)
        train_scores.append(cls.score(X_train,y_train))
        test_scores.append(cls.score(X_test,y_test))
```

(6) 绘图。

```
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.plot(alphas,train_scores,label = 'Training Score')
    ax.plot(alphas,test_scores,label = 'Testing Score')
    ax.set_xlabel(r'$\alpha$')
    ax.set_ylabel('score')
    ax.set_ylim(0,1.0)
    ax.set_title('BernoulliNB')
    ax.set_xscale('log')
    ax.legend(loc = 'best')
    plt.show()                    #效果如图 3-8 所示
X_train, X_test, y_train, y_test = load_data()
test_BernoulliNB_alpha(X_train, X_test, y_train, y_test)
```

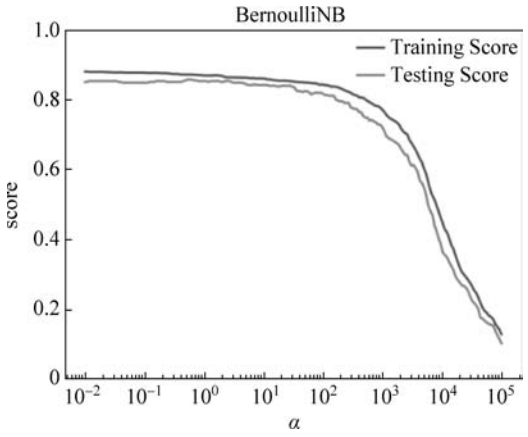

图 3-8 BernoulliNB 预测性能曲线图

(7) 考虑 binarize 参数对伯努利贝叶斯分类器的影响。

```
def test_BernoulliNB_binarize(*data):
    X_train, X_test, y_train, y_test = data
    min_x = min(np.min(X_train.ravel()),np.min(X_test.ravel())) - 0.1
    max_x = max(np.max(X_train.ravel()),np.max(X_test.ravel())) + 0.1
    binarizes = np.linspace(min_x,max_x,endpoint = True,num = 100)
    train_scores = []
    test_scores = []
    for binarize in binarizes:
        cls = naive_bayes.BernoulliNB(binarize = binarize)
```

```
            cls.fit(X_train,y_train)
            train_scores.append(cls.score(X_train,y_train))
            test_scores.append(cls.score(X_test,y_test))
        #绘图
        fig = plt.figure()
        ax = fig.add_subplot(1,1,1)
        ax.plot(binarizes,train_scores,label = 'Training Score')
        ax.plot(binarizes,test_scores,label = 'Testing Score')
        ax.set_xlabel('binarize')
        ax.set_ylabel('score')
        ax.set_ylim(0,1.0)
        ax.set_xlim(min_x - 1,max_x + 1)
        ax.set_title('BernoulliNB')
        ax.legend(loc = 'best')
        plt.show()                #效果如图 3-9 所示
X_train, X_test, y_train, y_test = load_data()
test_BernoulliNB_binarize(X_train, X_test, y_train, y_test)
```

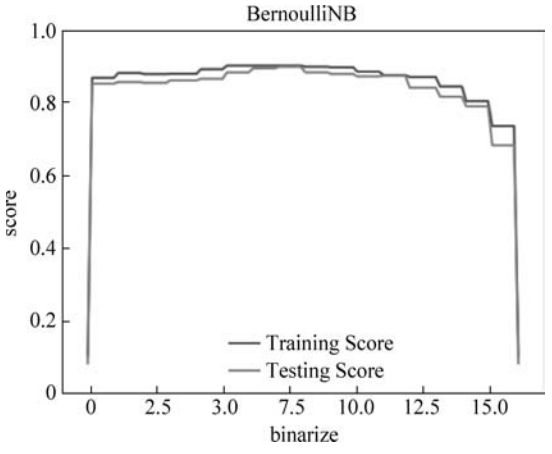

图 3-9　binarize 参数对 BernoulliNB 的影响

3.4　半朴素贝叶斯

朴素贝叶斯导出的分类器只是贝叶斯分类器中的一小类，它所做的独立性假设在绝大多数的情况下都显得太强，现实任务中这个假设往往难以成立。为了做出改进，人们尝试在不过于增加模型复杂度的前提下，将独立性假设进行各种弱化。在由此衍生出来的模型中，比较经典的就是半朴素贝叶斯(Semi-Naive Bayes)模型和贝叶斯网模型(Bayesian Network)两种。

由于提出条件独立性假设的原因正是联合概率难以求解，所以在弱化假设的时候同样应该避免引入过多的联合概率，这也正是半朴素贝叶斯的基本想法。比较常见的半朴素贝叶斯算法有如下 3 种。

3.4.1 ODE 算法

顾名思义，ODE 算法（One-Dependent Estimator，独依赖估计）各个维度的特征至多依赖一个其他维度的特征。从公式上来说，它在描述条件概率时会多出一个条件：

$$p(c_k \mid X=x)=p(y=c_k)\prod_{i=1}^{n}p(X^{(j)}=x^{(j)} \mid Y=c_k, X^{(\mathrm{pa}_j)}=x^{(\mathrm{pa}_j)})$$

这里的 pa_j 代表维度 j 所“独依赖”的维度。

3.4.2 SPODE 算法

SPODE 算法（Super-Parent ODE，超父独依赖估计）是 ODE 算法的一个特例。在该算法中，所有维度的特征都独依赖于同一个维度的特征，这个被共同依赖的特征就叫作“超父”（Super-Parent）。如果它的维度是第 pa 维，知：

$$p(c_k \mid X=x)=p(y=c_k)\prod_{i=1}^{n}p(X^{(j)}=x^{(j)} \mid Y=c_k, X^{(\mathrm{pa})}=x^{(\mathrm{pa})})$$

一般而言，会选择通过交叉验证来选择超父。

3.4.3 AODE 算法

AODE 算法（Averaged One-Dependent Estimator，集成独依赖估计）的背后有提升方法的思想。AODE 算法会利用 SPODE 算法并尝试将许多个训练后的、有足够的训练数据量支撑的 SPODE 模型集成在一起来构建最终的模型。一般来说，AODE 会以所有维度的特征作为超父训练 n 个 SPODE 模型，然后线性组合出最终的模型。

3.5 贝叶斯网

贝叶斯网又称“信念网”（Belief Network），比起朴素贝叶斯，它背后还蕴含了图论的思想。贝叶斯网有许多奇妙的性质，详细讨论的不可避免地要使用到图论的术语，这里仅仅对其做一个直观的介绍。

贝叶斯网络既然带了“网”字，它的结构自然可以直观地看作一张网络，其中，网络的节点就是单一样本的各个维度上的随机变量 $X^{(1)}, X^{(2)}, \cdots, X^{(n)}$，连接节点的边就是节点之间的依赖关系。

注意：贝叶斯网络一般要求这些边是“有方向的”，同时整张网络中不能出现“环”。无向的贝叶斯网络通常是由有向贝叶斯网络无向化得到的，此时它被称为 moral graph（除了把所有有向边改成无向边以外，moral graph 还需要将有向网络中不相互独立的随机变量之间连上一条无向边，细节略），基于它能够非常直观，故可迅速地看出变量间的条件独立性。

显然，有了代表各个维度随机变量的节点和代表这些节点之间依赖关系的边之后，各个

随机变量之间的条件依赖关系都可以通过这张网络表示出来。类似的在条件随机场中也有用到，可以说是一个适用范围非常泛的思想。

贝叶斯网络的学习在网络结构已经确定的情况下相对简单，其思想和朴素贝叶斯类似：只需要对训练集相应的条件进行“计数”即可，所以贝叶斯网的学习任务主要归结于如何找到最恰当的网络结构。常见的做法是定义一个用来打分的函数并基于该函数通过某种搜索手段来决定结构。如同很多最优化算法一样，在所有可能的结构空间中搜索最优结构是一个 NP 完全问题，无法在合理的时间内求解，所以一般会使用替代的方法求近似最优解。常见的方法有两种：一种是贪心法，比如，先确定一个初始的网络结构并从该结构出发，每次增添一条边、删去一条边或调整一条边的方向，期望通过这些手段能够使评分函数的值变大；另一种是直接限定假设空间，比如，假设要求的贝叶斯网络一定是一个树形结构。

与学习方法相比，贝叶斯网的决策方法并不简单。虽说最理想的情况是直接根据贝叶斯网络的结构所决定的联合概率密度来计算后验概率，但是这样的计算被证明是 NP 完全问题。换句话说，只要贝叶斯网络稍微复杂一点，这种精确的计算就无法在合理的时间内完成。所以同样要借助近似法求解，一种常见的做法是吉布斯采样(Gibbs Sampling)，它的定义涉及马尔可夫链的相关知识，这里不展开介绍。

3.6　习题

1. 在数理统计学中，_____是一种关于统计模型中的参数的函数，表示模型参数中的_____。

2. 贝叶斯学派强调概率的“主观性”与传统的频率学派不同有什么不同？

3. 和极大似然估计相比，MAP 有一个显著的优势，是什么？

4. 在朴素贝叶斯算法下衍生出 3 种不同的模型，是哪 3 种？

5. 利用高斯贝叶斯分类器对自带的 digits 数据集进行训练与测试。