菜菜的机器学习sklearn第十期

sklearn中的朴素贝叶斯

1 概述

1.1 真正的概率分类器

在许多分类算法应用中，特征和标签之间的关系并非是决定性的。比如说，我们想预测一个人究竟是否会在泰坦尼克号海难中生存下来，那我们可以建一棵决策树来学习我们的训练集。在训练中，其中一个人的特征为：30岁，男，普通舱，他最后在泰坦尼克号海难中去世了。当我们测试的时候，我们发现有另一个人的特征也为：30岁，男，普通舱。基于在训练集中的学习，我们的决策树必然会给这个人打上标签：去世。然而这个人的真实情况一定是去世了吗？并非如此。

也许这个人是心脏病患者，得到了上救生艇的优先权。又有可能，这个人就是挤上了救生艇，活了下来。对分类算法来说，基于训练的经验，这个人“很有可能”是没有活下来，但算法永远也无法确定”这个人一定没有活下来“。即便这个人最后真的没有活下来，算法也无法确定基于训练数据给出的判断，是否真的解释了这个人没有存活下来的真实情况。这就是说，**算法得出的结论，永远不是100%确定的，更多的是判断出了一种“样本的标签更可能是某类的可能性”，而非一种“确定”。**我们通过某些规定，比如说，在决策树的叶子节点上占比较多的标签，就是叶子节点上所有样本的标签，来强行让算法为我们返回一个固定结果。但许多时候，我们也希望能够理解算法判断出的可能性本身。

每种算法使用不同的指标来衡量这种可能性。比如说，决策树使用的就是叶子节点上占比较多的标签所占的比例（接口predict_proba调用），逻辑回归使用的是sigmoid函数压缩后的似然（接口predict_proba调用），而SVM使用的是样本点到决策边界的距离（接口decision_function调用）。但这些指标的本质，其实都是一种“类概率”的表示，我们可以通过归一化或sigmoid函数将这些指标压缩到0~1之间，让他们表示我们的模型对预测的结果究竟有多大的把握（置信度）。但无论如何，我们都希望使用真正的概率来衡量可能性，因此就有了真正的概率算法：朴素贝叶斯。

朴素贝叶斯是一种直接衡量标签和特征之间的概率关系的有监督学习算法，是一种专注分类的算法。朴素贝叶斯的算法根源就是基于概率论和数理统计的贝叶斯理论，因此它是根正苗红的概率模型。接下来，我们就来认识一下这个简单快速的概率算法。

1.2 朴素贝叶斯是如何工作的

朴素贝叶斯被认为是最简单的分类算法之一。首先，我们需要了解一些概率论的基本理论。假设有两个随机变量X和Y，他们分别可以取值为x和y。有这两个随机变量，我们可以定义两种概率：

关键概念：联合概率与条件概率
联合概率：“X取值为x”和“Y取值为y”两个事件同时发生的概率，表示为$P(X=x,Y=y)$ **条件概率**：在”X取值为x“的前提下，”Y取值为y“的概率，表示为$P(Y=y\|X=x)$

举个例子，我们让X为"气温"，Y为“七星瓢虫冬眠”，则X和Y可能的取值分为别x和y，其中x = {0,1}，0表示没有下降到0度以下，1表示下降到了0度以下。y = {0,1}，其中0表示否，1表示是。

两个事件分别发生的概率就为：

$P(X=1)$ = 50%，则是说明，气温下降到0度以下的可能性为50%，则$P(X=0)=1-P(X=1)$ = 50%。

$P(Y=1)$ = 70%，则是说明，七星瓢虫会冬眠的可能性为70%，则$P(Y=0)=1-P(Y=1)$ = 30%。

则这两个事件的联合概率为$P(X=1,\ Y=1)$，这个概率代表了气温下降到0度以下和七星瓢虫去冬眠这两件事情**同时，独立**发生的概率。

而两个事件之间的条件概率为$P(Y=1|X=1)$，这个概率代表了，当气温下降到0度以下这个条件被满足之后，七星瓢虫会去冬眠的概率。也就是说，气温下降到0度以下，一定程度上影响了七星瓢虫去冬眠这个事件。

在概率论中，我们可以证明，两个事件的联合概率等于这两个事件任意条件概率 * 这个条件事件本身的概率。

$$P(X=1,Y=1) = P(Y=1|X=1)*P(X=1) = P(X=1|Y=1)*P(Y=1)$$

简单一些，则可以将上面的式子写成：

$$P(X,Y) = P(Y|X)*P(X) = P(X|Y)*P(Y)$$

由上面的式子，我们可以得到贝叶斯理论等式：

$$P(Y|X) = \frac{P(X|Y)*P(Y)}{P(X)}$$

而这个式子，就是我们一切贝叶斯算法的根源理论。我们可以把我们的特征X当成是我们的条件事件，而我们要求解的标签Y当成是我们被满足条件后会被影响的结果，而两者之间的概率关系就是$P(Y|X)$，这个概率在机器学习中，被我们称之为是标签的后验概率（posterior probability），即是说我们先知道了条件，再去求解结果。而标签Y在没有任何条件限制下取值为某个值的概率，被我们写作$P(Y)$，与后验概率相反，这是完全没有任何条件限制的，标签的先验概率（prior probability）。而我们的$P(X|Y)$被称为"类的条件概率"，表示当Y的取值固定的时候，X为某个值的概率。那现在，有趣的事情就出现了。

1.2.1 瓢虫冬眠：理解P(Y|X)

假设，我们依然让X是"气温"，这就是我们的特征，Y是"七星瓢虫冬眠"，就是我们的标签。现在，我们建模的目的是，预测七星瓢虫是否会冬眠。在许多教材和博客里，大家会非常自然地开始说，我们现在求的就是我们的$P(Y|X)$，然后根据贝叶斯理论等式开始做各种计算和分析。现在请问大家，我写作$P(Y|X)$的这个概率，代表了什么呢？更具体一点，这个表达，可以代表多少种概率呢？

P(Y|X)代表了多少种情况的概率?

$P(Y=1|X=1)$ 气温0度以下的条件下，七星瓢虫冬眠的概率
$P(Y=1|X=0)$ 气温0度以上的条件下，七星瓢虫冬眠的概率
$P(Y=0|X=1)$ 气温0度以下的条件下，七星瓢虫没有冬眠的概率
$P(Y=0|X=0)$ 气温0度以上的条件下，七星瓢虫没有冬眠的概率

数学中的第一个步骤，也就是最重要的事情，就是定义清晰。其实在数学中，$P(Y|X)$还真的就代表了全部的可能性，而不是单一的概率本身。现在我们的Y有两种取值，而X也有两种取值，就让概率$P(Y|X)$的定义变得很模糊，排列组合之后竟然有4种可能。在机器学习当中，一个特征X下取值可能远远不止两种，标签也可能是多分类的，还会有多个特征，排列组合以下，到底求解的$P(Y|X)$是什么东西，是一个太让人感到混淆的点。同理，$P(Y)$随着标签中分类的个数，可以有不同的取值。$P(X|Y)$也是一样。在这里，我们来为大家澄清：

机器学习中的简写$P(Y)$，通常表示标签取到少数类的概率，少数类往往使用正样本表示，也就是$P(Y=1)$，本质就是所有样本中标签为1的样本所占的比例。如果没有样本不均衡问题，则必须在求解的时候明确，你的Y的取值是什么。

而$P(Y|X)$**是对于任意一个样本而言**，如果这个样本的特征X的取值为1，则表示求解$P(Y=1|X=1)$。如果这个样本下的特征X取值为0，则表示求解$P(Y=1|X=0)$。也就是说，$P(Y|X)$是具体到每一个样本上的，究竟求什么概率，由样本本身的特征的取值决定。每个样本的$P(Y|X)$如果大于阈值0.5，则认为样本是少数类（正样本，1），如果这个样本的$P(Y|X)$小于阈值0.5，则认为样本是多数类（负样本，0或者-1）。如果没有具体的样本，只是说明例子，则必须明确$P(Y|X)$中X的取值。

在机器学习当中，对每一个样本，我们不可能只有一个特征X，而是会存在着包含n个特征的取值的特征向量$\boldsymbol{X}$。因此机器学习中的后验概率，被写作$P(Y|\boldsymbol{X})$，其中$\boldsymbol{X}$中包含样本在n个特征X_i上的分别的取值x_i，由此可i以表示为$\boldsymbol{X}=\{X_1=x_1,X_2=x_2,....X_n=x_n\}$。

因此，我们有：

一个样本上，所有特征取值下的概率：

每一个向量都是一个样本上的特征向量

$$P(\boldsymbol{X})$$
$$P(\boldsymbol{x})$$
$$P(X_1=x_1,X_2=x_2,\dots,X_n=x_n)$$
$$P(X_1,X_2,\dots,X_n)$$
$$P(x_1,x_2,\dots,x_n)$$

一个样本上，一个特征所取值下的概率：

是对于单独元素来说，没有向量存在

$$P(X_i=x_i)$$
$$P(X_i)$$
$$P(x_i)$$

一个特征上，所有样本下所取得的概率：

每一个向量都是一个特征下的样本向量

$$P(\boldsymbol{X_i})$$
$$P(i)$$

虽然写法不同，但其实都包含折同样的含义。以此为基础，机器学习中，**对每一个样本**我们有：

$$P(Y=1|\boldsymbol{X})=\frac{P(\boldsymbol{X}|Y=1)*P(Y=1)}{P(\boldsymbol{X})}=\frac{P(x_1,x_2....x_n|Y=1)*P(Y=1)}{P(x_1,x_2....x_n)}$$

对于分子而言，$P(Y=1)$就是少数类占总样本量的比例，$P(\boldsymbol{X}|Y=1)$则需要稍微复杂一点的过程来求解。假设我们只有两个特征X_1，X_2，由联合概率公式，我们可以有如下证明：

$$\begin{aligned}P(X_1,X_2|Y=1)&=\frac{P(X_1,X_2,Y=1)}{P(Y=1)}\\&=\frac{P(X_1,X_2,Y=1)}{P(X_2,Y=1)}*\frac{P(X_2,Y=1)}{P(Y=1)}\\&=P(X_1|X_2,Y=1)*P(X_2|Y=1)\end{aligned}$$

假设X_1和X_2之是有条件独立的：

$$=P(X_1|Y=1)*P(X_2|Y=1)$$

是一种若推广到n个X上，则有：

$$P(\boldsymbol{X}|Y=1)=\prod_{i=1}^{n}P(X_i=x_i|Y=1)$$

这个式子证明，在Y=1的条件下，多个特征的取值被同时取到的概率，就等于Y=1的条件下，多个特征的取值被分别取到的概率相乘。其中，假设X_1与X_2是有条件独立则可以让公式$P(X_1|X_2,Y=1)=P(X_1|Y=1)$，这是在假设$X_2$是一个对$X_1$在某个条件下的取值完全无影响的变量。

比如说，温度(X_1)与观察到的瓢虫的数目(X_2)之间的关系。有人可能会说，温度在零下的时候，观察到的瓢虫数目往往很少。这种关系的存在可以通过一个中间因素：瓢虫会冬眠(Y)来解释。冬天瓢虫都冬眠了，自然观察不到很多瓢虫出来活动。也就是说，如果瓢虫冬眠的属性是固定的($Y=1$)，那么观察到的温度和瓢虫出没数目之间关系就会消失，因此无论是否还存在着"观察到的瓢虫的数目"这样的因素，我们都可以判断这只瓢虫到底会不会冬眠。这种情况下，我们就说，温度与观察到的瓢虫的数目，是条件独立的。

假设特征之间是有条件独立的，可以解决众多问题，也简化了很多计算过程，这是朴素贝叶斯被称为"朴素"的理由。因此，贝叶斯在特征之间有较多相关性的数据集上表现不佳，而现实中的数据多多少少都会有一些相关性，所以贝叶斯的分类效力在分类算法中不算特别强大。同时，一些影响特征本身的相关性的降维算法，比如PCA和SVD，和贝叶斯连用效果也会不佳。但无论如何，有了这个式子，我们就可以求解出我们的分子了。

接下来，来看看我们贝叶斯理论等式的分母$P(\boldsymbol{X})$。我们可以使用全概率公式来求解$P(\boldsymbol{X})$：

$$P(\boldsymbol{X})=\sum_{i=1}^{m}P(y_i)*P(\boldsymbol{X}|Y_i)$$

其中m代表标签的种类，也就是说，对于二分类而言我们有：

$$P(\boldsymbol{X})=P(Y=1)*P(\boldsymbol{X}|Y=1)+P(Y=0)*P(\boldsymbol{X}|Y=0)$$

基于这个方程，我们可以来求解一个非常简单的例子下的后验概率。

索引	温度（X_1）	瓢虫的年龄（X_2）	瓢虫冬眠（Y）
0	零下	10天	是
1	零下	20天	是
2	零上	10天	否
3	零下	一个月	是
4	零下	20天	否
5	零上	两个月	否
6	零下	一个月	否
7	零下	两个月	是
8	零上	一个月	否
9	零上	10天	否
10	零下	20天	否

现在，我们希望预测零下的时候，年龄为20天的瓢虫，是否会冬眠。

$$P(Y=1|X_1=\text{零下},X_2=20\text{天})=\frac{P(x_1,x_2|Y=1)*P(Y=1)}{P(x_1,x_2)}$$

$$P(\text{冬眠}|\text{零下},20\text{天})=\frac{P(\text{零下},20\text{天}|\text{冬眠})*P(\text{冬眠})}{P(\text{零下},20\text{天})}$$

$$=\frac{P(\text{零下}|\text{冬眠})*P(20\text{天}|\text{冬眠})*P(\text{冬眠})}{P(\text{冬眠})*P(\text{零下},20\text{天}|\text{冬眠})+P(\text{不冬眠})*P(\text{零下},20\text{天}|\text{不冬眠})}$$

对于分子我们可以求得：

$$P(\text{冬眠})=\frac{4}{11},\quad P(\text{零下}|\text{冬眠})=\frac{4}{4}=1,\ \ P(20\text{天}|\text{冬眠})=\frac{1}{4}$$

对于分母我们可以求得：

$$P(\text{零下},20\text{天}|\text{冬眠})=\frac{1}{4},\ P(\text{不冬眠})=\frac{7}{11},\ P(\text{零下},20\text{天}|\text{不冬眠})=\frac{2}{7}$$

所以我们的，温度为零下的时候， 生活了20天的瓢虫，会冬眠的概率为：

$$\frac{4/11*1*1/4}{4/11\ *1/4\ +\ 7/11\ *2/7}=\frac{1/11}{3/11}=0.33$$

设定阈值为0.5，假设大于0.5的就被认为是会冬眠，小于0.5的就被认为是不会冬眠。根据我们的计算，我们认为一个在零下条件下，年龄为20天的瓢虫，是不会冬眠的。这就完成了一次预测。但是这样，有趣的地方又来了。刚才的预测过程是没有问题的。但我们总是好奇，这个过程如何对应sklearn当中的fit和predict呢？这个决策过程中，我们的训练集和我的测试集分别在哪里？以及，算法建模建模，我的模型在哪里呢？

1.2.2 贝叶斯的性质与最大后验估计

在过去的许多个星期内，我们学习的分类算法总是有一个特点：这些算法先从训练集中学习，获取某种信息来建立模型，然后用模型去对测试集进行预测。比如逻辑回归，我们要先从训练集中获取让损失函数最小的参数，然后用参数建立模型，再对测试集进行预测。在比如支持向量机，我们要先从训练集中获取让边际最大的决策边界，然后用决策边界对测试集进行预测。相同的流程在决策树，随机森林中也出现，我们在fit的时候必然已经构造好了能够让对测试集进行判断的模型。而朴素贝叶斯，似乎没有这个过程。

我给了大家一张有标签的表，然后提出说，我要预测零下的时候，年龄为20天的瓢虫，会冬眠的概率，然后我们就顺理成章地算了出来。没有利用训练集求解某个模型的过程，也没有训练完毕了我们来做测试的过程，而是直接对有标签的数据提出要求，就可以得到预测结果了。

这说明，**朴素贝叶斯是一个不建模的算法。**以往我们学的不建模算法，比如KMeans，比如PCA，都是无监督学习，而朴素贝叶斯是第一个有监督的，不建模的分类算法。在我们刚才举的例子中，有标签的表格就是我们的训练集，而我提出的要求“零下的时候，年龄为20天的瓢虫”就是没有标签的测试集。我们认为，训练集和测试集都来自于同一个不可获得的大样本下，并且这个大样本下的各种属性所表现出来的规律应当是一致的，因此训练集上计算出来的各种概率，可以直接放到测试集上来使用。即便不建模，也可以完成分类。

但实际中，贝叶斯的决策过程并没有我们给出的例子这么简单。

$$P(Y=1|\boldsymbol{X}) = \frac{P(Y=1)*\prod_{i=1}^{n}P(x_i|Y=1)}{P(\boldsymbol{X})}$$

对于这个式子来说，从训练集中求解$P(Y=1)$很容易，但$P(\boldsymbol{X})$和$P(x_i|Y=1)$这一部分就没有这么容易了。在我们的例子中，我们通过全概率公式来求解分母，两个特征就求解了四项概率。随着特征数目的逐渐变多，分母上的计算两会成指数级增长，而分子中的$P(x_i|Y=1)$也越来越难计算。

不过幸运的是，对于同一个样本来说，在二分类状况下我们可以有：

$$P(Y=1|\boldsymbol{X}) = \frac{P(Y=1)*\prod_{i=1}^{n}P(x_i|Y=1)}{P(\boldsymbol{X})}$$
$$P(Y=0|\boldsymbol{X}) = \frac{P(Y=0)*\prod_{i=1}^{n}P(x_i|Y=0)}{P(\boldsymbol{X})}$$

并且：

$$P(Y=1|\boldsymbol{X}) + P(Y=0|\boldsymbol{X}) = 1$$

在分类的时候，我们选择$P(Y=1|\boldsymbol{X})$和$P(Y=0|\boldsymbol{X})$中较大的一个所对应的Y的取值，作为这个样本的分类。在比较两个类别的时候，两个概率计算的分母是一致的，因此我们可以不用计算分母，只考虑分子的大小。当我们分别计算出分子的大小之后，就可以通过让两个分子相加，来获得分母的值，以此来避免计算一个样本上所有特征下的概率$P(\boldsymbol{X})$。这个过程，被我们称为“最大后验估计”（MAP）。在最大后验估计中，我们只需要求解分子，主要是求解一个样本下每个特征取值下的概率$P(x_i|Y=y_i)$，再求连乘便能够获得相应的概率。

在现实中，要求解分子也会有各种各样的问题。比如说，测试集中出现的某种概率组合，是训练集中从未出现的状况，这种时候就会出现某一个概率为0的情况，贝叶斯概率的分子就会为0。还有，现实中的大多数标签还是连续型变量，要处理连续型变量的概率，就不是单纯的数样本个数的占比的问题了。接下来我们就来看看，如何对连续型特征求解概率。

1.2.3 汉堡称重：连续型变量的概率估计

要处理连续型变量，我们可以有两种方法。第一种是把连续型变量分成j个箱，把连续型强行变成分类型变量。我们分箱后，将每个箱中的均值$\bar{x}_i$当作一个特征X_i上的取值，然后我们计算箱j中$Y=1$所占的比例，就是我们的$P(x_i|Y=1)$。这个过程的主要问题是，箱子不能太大也不能太小，如果箱子太大，就失去了分箱的基本意义，如果箱子太小，可能每个箱子里就没有足够的样本来帮助我们计算$P(x_i|Y)$，因此我们必须要适当地衡量我们的分箱效果。

但其实，我们没有必要这样做，因为我们可以直接通过概率论中来计算连续型变量的概率分布。在分类型变量的情况中，比如掷骰子的情况，我们有且仅有六种可能的结果1~6，并且每种结果的可能性为1/6。此时每个基本的随机事件发生的概率都是相等的，所以我们可以使用$\frac{1}{N}$来表示有N个基本随机事件可以发生的情况。

基于此，我们来思考一个简单的问题：汉堡王向客户承诺说他们的汉堡至少是100g一个，但如果我们去汉堡王买个汉堡，我们可以预料到它肯定不是标准的100g。设我们的汉堡重量为特征Xi，100g就是我们的取值x_i，那我买到一个汉堡是100g的概率$P(100g|Y)$是多少呢？如果我们买n个汉堡，很可能n个汉堡都不一样重，只要我们称重足够精确，100.000001g和100.00002g就可以是不一致的。这种情况下我们可以买无限个汉堡，可能得到无限个重量，可以有无限个基本随机事件的发生，所以我们有：

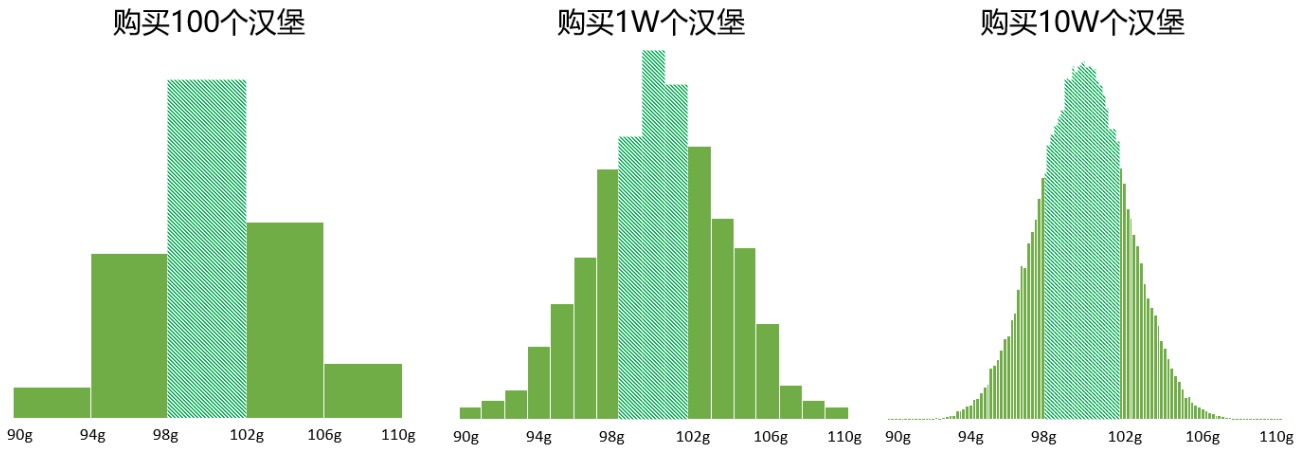
$$P(100g|Y) = \lim_{N\to\infty}\frac{1}{N} = 0$$

即买到的汉堡刚好是100g概率为0。当一个特征下有无数种可能发生的事件时，这个特征的取值就是连续型的，比如我们现在的特征"汉堡的重量"。从上面的例子可以看得出，**当特征为连续型时，随机取到某一个事件发生的概率就为0。**

那换一个问题，我们随机买一个汉堡，汉堡的重量在98g~102g之间的概率是多少？即是说，我们现在求解概率$P(98g < x < 102g)$。那我们现在随机购买100个汉堡，称重后记下所有重量在98g~102g之间的汉堡个数，假设为m，则就有：

$$P(98g < x < 102g) = \frac{m}{100}$$

如果我们基于100个汉堡绘制直方图，并规定每4g为一个区间，横坐标为汉堡的重量的分布，纵坐标为这个区间上汉堡的个数。



那我们可以看到最左边的图。m就是中间的浅绿色区间中所对应的纵坐标轴。则对于我们的概率，我们可以变换为：

$$P(98g < x < 102g) = \frac{m * 4}{100 * 4} = \frac{\text{浅绿色区间的面积}}{\text{直方图中所有区间的面积}}$$

如果我们购买一万个汉堡并绘制直方图（如中间的图），将直方图上的区间缩小，$P(98g < x < 102g)$依然是所有浅绿色区域的面积除以所有柱状图的面积，可以看见现在我们的直方图变得更加平滑，看起来更像一座山了。那假设我们购买10W个，或者无限个汉堡，则我们可以想象我们的直方图最终会变成仿佛一条曲线，而我们的汉堡重量的概率$P(98g < x < 102g)$依然是所有浅绿色柱子的面积除以曲线下所有柱状图的总面积。当购买无数个汉堡的时候形成的则条曲线就叫做概率密度曲线（probability density function，PDF）。

一条曲线下的面积，就是这条曲线所代表的函数的积分。如果我们定义曲线可以用函数$f(x)$来表示的话，我们整条曲线下的面积就是：

$$\int_{-\infty}^{+\infty} f(x)dx$$

其中dx是$f(x)$在x上的微分。在某些特定的$f(x)$下，我们可以证明，上述积分等于1。总面积是1，这说明一个连续型特征X的取值x取到某个区间$[\boldsymbol{x_i}, \boldsymbol{x_i} + \boldsymbol{\epsilon}]$之内的概率就为这个区间上概率密度曲线下的面积，所以我们的特征X_i在区间$[x_i, x_i + \epsilon]$中取值的概率可以表示为：

$$\begin{aligned} P(x_i < x < x_i + \epsilon) &= \int_{x_i}^{x_i+\epsilon} f(x)dx \\ &\approx f(x_i) * \epsilon \end{aligned}$$

非常幸运的是，在我们后验概率的计算过程中，我们可以将常量ϵ抵消掉，然后我们就可以利用$f(x_i)$的某种变化来估计我们的$P(x_i|Y)$了。**现在，我们就将求解连续型变量下某个点取值的概率问题，转化成了求解一个函数$f(x)$在点x_i上的取值的问题。**那接下来只要找到我们的$f(x)$，我们就可以求解出不同的条件概率了。

在现实中，我们往往假设我们的$f(x)$是满足某种统计学中的分布的，最常见的就是高斯分布（正太分布，像我们购买汉堡的例子），常用的还有伯努利分布，多项式分布。这些分布对应着不同的贝叶斯算法，其实他们的本质都是相同的，只不过他们计算之中的$f(x)$不同。每个$f(x)$都对应着一系列需要我们去估计的参数，因此在贝叶斯中，我们的fit过程其实是在估计对应分布的参数，predict过程是在该参数下的分布中去进行概率预测。

1.3 sklearn中的朴素贝叶斯

Sklearn基于这些分布以及这些分布上的概率估计的改进，为我们提供了四个朴素贝叶斯的分类器。

类	含义
naive_bayes.BernoulliNB	伯努利分布下的朴素贝叶斯
naive_bayes.GaussianNB	高斯分布下的朴素贝叶斯
naive_bayes.MultinomialNB	多项式分布下的朴素贝叶斯
naive_bayes.ComplementNB	补集朴素贝叶斯
linear_model.BayesianRidge	贝叶斯岭回归，在参数估计过程中使用贝叶斯回归技术来包括正则化参数

虽然朴素贝叶斯使用了过于简化的假设，这个分类器在许多实际情况中都运行良好，著名的是文档分类和垃圾邮件过滤。而且由于贝叶斯是从概率角度进行估计，它所需要的样本量比较少，极端情况下甚至我们可以使用1%的数据作为训练集，依然可以得到很好的拟合效果。当然，如果样本量少于特征数目，贝叶斯的效果就会被削弱。

与SVM和随机森林相比，朴素贝叶斯运行速度更快，因为求解$P(X_i|Y)$本质是在每个特征上单独对概率进行计算，然后再求乘积，所以每个特征上的计算可以是独立并且并行的，因此贝叶斯的计算速度比较快。不过相对的，贝叶斯的运行效果不是那么好，所以贝叶斯的接口调用的predict_proba其实也不是总指向真正的分类结果，这一点需要注意。

2 不同分布下的贝叶斯

2.1 高斯朴素贝叶斯GaussianNB

2.1.1 认识高斯朴素贝叶斯

class `sklearn.naive_bayes.GaussianNB`(*priors=None*, *var_smoothing=1e-09*)

高斯朴素贝叶斯，通过假设$P(x_i|Y)$是服从高斯分布（也就是正态分布），来估计每个特征下每个类别上的条件概率。对于每个特征下的取值，高斯朴素贝叶斯有如下公式：

$$P(x_i|Y) = f(x_i;\mu_y,\sigma_y) * \epsilon$$
$$= \frac{1}{\sqrt{2\pi\sigma_y^2}} exp(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2})$$

对于任意一个Y的取值，贝叶斯都以求解最大化的$P(x_i|Y)$为目标，这样我们才能够比较在不同标签下我们的样本究竟更靠近哪一个取值。以最大化$P(x_i|Y)$为目标，高斯朴素贝叶斯会为我们求解公式中的参数σ_y和μ_y。求解出参数后，带入一个x_i的值，就能够得到一个$P(x_i|Y)$的概率取值。

这个类包含两个参数：

参数	含义
prior	可输入任何类数组结构，形状为（n_classes，） 表示类的先验概率。如果指定，则不根据数据调整先验，如果不指定，则自行根据数据计算先验概率$P(Y)$。
var_smoothing	浮点数，可不填（默认值= 1e-9） 在估计方差时，为了追求估计的稳定性，将所有特征的方差中最大的方差以某个比例添加到估计的方差中。这个比例，由var_smoothing参数控制。

但在实例化的时候，我们不需要对高斯朴素贝叶斯类输入任何的参数，调用的接口也全部sklearn中比较标准的一些搭配，可以说是一个非常轻量级的类，操作非常容易。但过于简单也意味着贝叶斯没有太多的参数可以调整，因此贝叶斯算法的成长空间并不是太大，如果贝叶斯算法的效果不是太理想，我们一般都会考虑换模型。

无论如何，先来进行一次预测试试看吧：

1. **展示我所使用的设备以及各个库的版本**

在这里我们来使用watermask这个便利的模块来帮助我们，这是一个能够帮助我们一行代码查看设备和库的版本的模块。如果没有watermask的你可能需要在cmd中运行pip来安装。也可以直接使用魔法命令%%cmd作为一个cell的开头来帮助我们在jupyter lab中安装你的watermark。

```
%%cmd
pip install watermark

#在这里必须分开cell，魔法命令必须是一个cell的第一部分内容
#注意load_ext这个命令只能够执行一次，再执行就会报错，要求用reload命令
%load_ext watermark

%watermark -a "TsaiTsai" -d -v -m -p numpy,pandas,matplotlib,scipy,sklearn
```

2. **导入需要的库和数据**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.naive_bayes import GaussianNB
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
X, y = digits.data, digits.target

Xtrain,Xtest,Ytrain,Ytest = train_test_split(X,y,test_size=0.3,random_state=420)
```

3. **建模，探索建模结果**

```
gnb = GaussianNB().fit(Xtrain,Ytrain)

#查看分数
acc_score = gnb.score(Xtest,Ytest)

acc_score

#查看预测结果
Y_pred = gnb.predict(Xtest)

#查看预测的概率结果
prob = gnb.predict_proba(Xtest)

prob.shape

prob.shape #每一列对应一个标签下的概率

prob[1,:].sum() #每一行的和都是一

prob.sum(axis=1)
```

4. **使用混淆矩阵来查看贝叶斯的分类结果**

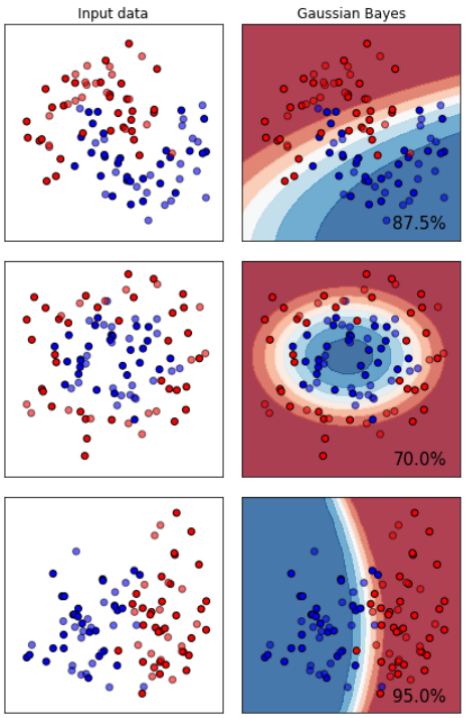
```
from sklearn.metrics import confusion_matrix as CM
CM(Ytest,Y_pred)

#注意，ROC曲线是不能用于多分类的。多分类状况下最佳的模型评估指标是混淆矩阵和整体的准确度
```

2.1.2 探索贝叶斯：高斯朴素贝叶斯擅长的数据集

那高斯普斯贝叶斯擅长什么样的数据集呢？我们还是使用常用的三种数据分布：月亮型，环形数据以及二分型数据。**注意这段代码曾经在决策树中详细讲解过，在SVM中也有非常类似的代码，核心就是构建分类器然后画决策边界，只不过更换了需要验证的模型而已，因此在这里就不对这段代码是如何实现的进行赘述了。**需要了解的小伙伴可以去查看第一章决策树完整版：决策树在合成数据集上的表现，或者查看SVM第一期完整版中，SVC在不同数据集上的表现。

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.naive_bayes import GaussianNB

h = .02

names = ["Multinomial","Gaussian","Bernoulli","Complement"]

classifiers = [MultinomialNB(),GaussianNB(),BernoulliNB(),ComplementNB()]

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           random_state=1, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable
            ]

figure = plt.figure(figsize=(6, 9))
i = 1

for ds_index, ds in enumerate(datasets):
    X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.4,
random_state=42)
    x1_min, x1_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    x2_min, x2_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    array1,array2 = np.meshgrid(np.arange(x1_min, x1_max, 0.2),
                        np.arange(x2_min, x2_max, 0.2))
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = plt.subplot(len(datasets), 2, i)
    if ds_index == 0:
        ax.set_title("Input data")
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
               cmap=cm_bright,edgecolors='k')
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test,
               cmap=cm_bright, alpha=0.6,edgecolors='k')
    ax.set_xlim(array1.min(), array1.max())
    ax.set_ylim(array2.min(), array2.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1
    ax = plt.subplot(len(datasets),2,i)
```

```
    clf = GaussianNB().fit(X_train, y_train)
    score = clf.score(X_test, y_test)

    Z = clf.predict_proba(np.c_[array1.ravel(),array2.ravel()])[:, 1]
    Z = Z.reshape(array1.shape)
    ax.contourf(array1, array2, Z, cmap=cm, alpha=.8)

    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
               edgecolors='k')
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
               edgecolors='k', alpha=0.6)

    ax.set_xlim(array1.min(), array1.max())
    ax.set_ylim(array2.min(), array2.max())
    ax.set_xticks(())
    ax.set_yticks(())

    if ds_index == 0:
        ax.set_title("Gaussian Bayes")

    ax.text(array1.max() - .3, array2.min() + .3, ('{:.1f}%'.format(score*100)),
            size=15, horizontalalignment='right')
    i += 1

plt.tight_layout()
plt.show()
```



从图上来看，高斯贝叶斯属于比较特殊的一类分类器，其分类效果在二分数据和月亮型数据上表现优秀，但是环形数据不太擅长。我们之前学过的模型中，许多线性模型比如逻辑回归，线性SVM等等，在线性数据集上会绘制直线决策边界，因此难以对月亮型和环形数据进行区分，但高斯朴素贝叶斯的决策边界是曲线，可以是环形也可以是弧线，所以尽管贝叶斯本身更加擅长线性可分的二分数据，但朴素贝叶斯在环形数据和月亮型数据上也可以有远远胜过其他线性模型的表现。

2.1.3 探索贝叶斯：高斯朴素贝叶斯的拟合效果与运算速度

我们已经了解高斯朴素贝叶斯属于分类效果不算顶尖的模型，但我们依然好奇，这个算法在拟合的时候还有哪些特性呢？比如说我们了解，决策树是天生过拟合的模型，而支持向量机是不调参数的情况下就非常接近极限的模型。我们希望通过绘制高斯朴素贝叶斯的学习曲线与分类树，随机森林和支持向量机的学习曲线的对比，来探索高斯朴素贝叶斯算法在拟合上的性质。过去绘制学习曲线都是以算法类的某个参数的取值为横坐标，今天我们来使用sklearn中自带的绘制学习曲线的类learning_curve，在这个类中执行交叉验证并从中获得不同样本量下的训练和测试的准确度。

1. **首先导入需要的模块和库**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.tree import DecisionTreeClassifier as DTC
from sklearn.linear_model import LogisticRegression as LR
from sklearn.datasets import load_digits
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit
from time import time
import datetime
```

2. **定义绘制学习曲线的函数**

```
def plot_learning_curve(estimator,title, X, y,
                        ax, #选择子图
                        ylim=None, #设置纵坐标的取值范围
                        cv=None, #交叉验证
                        n_jobs=None #设定索要使用的线程
                       ):
    train_sizes, train_scores, test_scores = learning_curve(estimator, X, y
                                                            ,cv=cv,n_jobs=n_jobs)
    ax.set_title(title)
    if ylim is not None:
        ax.set_ylim(*ylim)
    ax.set_xlabel("Training examples")
    ax.set_ylabel("Score")
    ax.grid() #显示网格作为背景，不是必须
    ax.plot(train_sizes, np.mean(train_scores, axis=1), 'o-'
            , color="r",label="Training score")
    ax.plot(train_sizes, np.mean(test_scores, axis=1), 'o-'
            , color="g",label="Test score")
    ax.legend(loc="best")
    return ax
```

3. **导入数据，定义循环**

```
digits = load_digits()
X, y = digits.data, digits.target

X.shape

X #是一个稀疏矩阵

title = ["Naive Bayes","DecisionTree","SVM, RBF kernel","RandomForest","Logistic"]
model = [GaussianNB(),DTC(),SVC(gamma=0.001)
         ,RFC(n_estimators=50),LR(C=.1,solver="lbfgs")]
cv = ShuffleSplit(n_splits=50, test_size=0.2, random_state=0)
```

4. **进入循环，绘制学习曲线**

```
fig, axes = plt.subplots(1,5,figsize=(30,6))
for ind,title_,estimator in zip(range(len(title)),title,model):
    times = time()
    plot_learning_curve(estimator, title_, X, y,
                        ax=axes[ind], ylim = [0.7, 1.05],n_jobs=4, cv=cv)
    print("{}:{}".format(title_,datetime.datetime.fromtimestamp(time()-
times).strftime("%M:%S:%f")))
plt.show()
```

三个模型表现出的状态非常有意思。

我们首先返回的结果是**各个算法的运行时间。**可以看到，决策树和贝叶斯不相伯仲（如果你没有发现这个结果，那么可以多运行几次，你会发现贝叶斯和决策树的运行时间逐渐变得差不多）。决策树之所以能够运行非常快速是因为sklearn中的分类树在选择特征时有所"偷懒"，没有计算全部特征的信息熵而是随机选择了一部分特征来进行计算，因此速度快可以理解，但我们知道决策树的运算效率随着样本量逐渐增大会越来越慢，但朴素贝叶斯却可以在很少的样本上获得不错的结果，因此我们可以预料，随着样本量的逐渐增大贝叶斯会逐渐变得比决策树更快。朴素贝叶斯计算速度远远胜过SVM，随机森林这样复杂的模型，逻辑回归的运行受到最大迭代次数的强烈影响和输入数据的影响（逻辑回归一般在线性数据上运行都比较快，但在这里应该是受到了稀疏矩阵的影响）。因此在运算时间上，朴素贝叶斯还是十分有优势的。

紧接着，我们来看一下每个算法在**训练集上的拟合。**手写数字数据集是一个较为简单的数据集，决策树，森林，SVC和逻辑回归都成功拟合了100%的准确率，但贝叶斯的最高训练准确率都没有超过95%，这也应证了我们最开始说的，朴素贝叶斯的分类效果其实不如其他分类器，贝叶斯天生学习能力比较弱。并且我们注意到，随着训练样本量的逐渐增大，其他模型的训练拟合都保持在100%的水平，但贝叶斯的训练准确率却逐渐下降，这证明样本量越大，贝叶斯需要学习的东西越多，对训练集的拟合程度也越来越差。反而比较少量的样本可以让贝叶斯有较高的训练准确率。

再来看看**过拟合问题。**首先一眼看到，所有模型在样本量很少的时候都是出于过拟合状态的（训练集上表现好，测试集上表现糟糕），但随着样本的逐渐增多，过拟合问题都逐渐消失了，不过每个模型的处理手段不同。比较强大的分类器们，比如SVM，随机森林和逻辑回归，是依靠快速升高模型在测试集上的表现来减轻过拟合问题。相对的，决策树虽然也是通过提高模型在测试集上的表现来减轻过拟合，但随着训练样本的增加，模型在测试集上的表现善生却非常缓慢。朴素贝叶斯独树一帜，是依赖训练集上的准确率下降，测试集上的准确率上升来逐渐解决过拟合问题。

接下来，看看每个算法在**测试集上的拟合结果，即泛化误差的大小。**随着训练样本数量的上升，所有模型的测试表现都上升了，但贝叶斯和决策树在测试集上的表现远远不如SVM，随机森林和逻辑回归。SVM在训练数据量增大到1500个样本左右的时候，测试集上的表现已经非常接近100%，而随机森林和逻辑回归的表现也在95%以上，而决策树和朴素贝叶斯还徘徊在85%左右。但这两个模型所面临的情况十分不同：决策树虽然测试结果不高，但是却依然具

有潜力，因为它的过拟合现象非常严重，我们可以通过减枝来让决策树的测试结果逼近训练结果。然而贝叶斯的过拟合现象在训练样本达到1500左右的时候已经几乎不存在了，训练集上的分数和测试集上的分数非常接近，只有在非常少的时候测试集上的分数才能够比训练集上的结果更高，所以我们基本可以判断，85%左右就是贝叶斯在这个数据集上的极限了。可以预测到，如果我们进行调参，那决策树最后应该可以达到90%左右的预测准确率，但贝叶斯却几乎没有潜力了。

在这个对比之下，我们可以看出：贝叶斯是速度很快，但分类效果一般，并且初次训练之后的结果就很接近算法极限的算法，几乎没有调参的余地。也就是说，如果我们追求对概率的预测，并且希望越准确越好，那我们应该先选择逻辑回归。如果数据十分复杂，或者是稀疏矩阵，那我们坚定地使用贝叶斯。如果我们分类的目标不是要追求对概率的预测，那我们完全可以先试试看高斯朴素贝叶斯的效果（反正它运算很快速，还不需要太多的样本），如果效果很不错，我们就很幸运地得到了一个表现优秀又快速的模型。如果我们没有得到比较好的结果，那我们完全可以选择再更换成更加复杂的模型。

2.2 概率类模型的评估指标

混淆矩阵和精确性可以帮助我们了解贝叶斯的分类结果。然而，我们选择贝叶斯进行分类，大多数时候都不是为了单单追求效果，而是希望看到预测的相关概率。这种概率给出预测的可信度，所以对于概率类模型，我们希望能够由其他的模型评估指标来帮助我们判断，模型在“概率预测”这项工作上，完成得如何。接下来，我们就来看看概率模型独有的评估指标。

2.2.1 布里尔分数Brier Score

概率预测的准确程度被称为“校准程度”，是衡量算法预测出的概率和真实结果的差异的一种方式。一种比较常用的指标叫做布里尔分数，它被计算为是概率预测相对于测试样本的均方误差，表示为：

$$Brier\ Score = \frac{1}{N}\sum_{i=1}^{n}(p_i - o_i)^2$$

其中N是样本数量，p_i为朴素贝叶斯预测出的概率，o_i是样本所对应的真实结果，只能取到0或者1，如果事件发生则为1，如果不发生则为0。这个指标衡量了我们的概率距离真实标签结果的差异，其实看起来非常像是均方误差。**布里尔分数的范围是从0到1，分数越高则预测结果越差劲，校准程度越差，因此布里尔分数越接近0越好。**由于它的本质也是在衡量一种损失，所以在sklearn当中，布里尔得分被命名为brier_score_loss。我们可以从模块metrics中导入这个分数来衡量我们的模型评估结果：

```
from sklearn.metrics import brier_score_loss

#注意，第一个参数是真实标签，第二个参数是预测出的概率值
#在二分类情况下，接口predict_proba会返回两列，但SVC的接口decision_function却只会返回一列
#要随时注意，使用了怎样的概率分类器，以辨别查找置信度的接口，以及这些接口的结构
brier_score_loss(Ytest, prob[:,1], pos_label=1)
#我们的pos_label与prob中的索引一致，就可以查看这个类别下的布里尔分数是多少
```

布里尔分数可以用于任何可以使用predict_proba接口调用概率的模型，我们来探索一下在我们的手写数字数据集上，逻辑回归，SVC和我们的高斯朴素贝叶斯的效果如何：

```
from sklearn.metrics import brier_score_loss
brier_score_loss(Ytest,prob[:,8],pos_label=8)
```

```
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression as LR

logi = LR(C=1., solver='lbfgs',max_iter=3000,multi_class="auto").fit(Xtrain,Ytrain)
svc = SVC(kernel = "linear",gamma=1).fit(Xtrain,Ytrain)

brier_score_loss(Ytest,logi.predict_proba(Xtest)[:,1],pos_label=1)

#由于SVC的置信度并不是概率，为了可比性，我们需要将SVC的置信度“距离”归一化，压缩到[0,1]之间
svc_prob = (svc.decision_function(Xtest) -
svc.decision_function(Xtest).min())/(svc.decision_function(Xtest).max() -
svc.decision_function(Xtest).min())

brier_score_loss(Ytest,svc_prob[:,1],pos_label=1)
```

如果将每个分类器每个标签类别下的布里尔分数可视化：

```
import pandas as pd
name = ["Bayes","Logistic","SVC"]
color = ["red","black","orange"]

df = pd.DataFrame(index=range(10),columns=name)
for i in range(10):
    df.loc[i,name[0]] = brier_score_loss(Ytest,prob[:,i],pos_label=i)
    df.loc[i,name[1]] = brier_score_loss(Ytest,logi.predict_proba(Xtest)
[:,i],pos_label=i)
    df.loc[i,name[2]] = brier_score_loss(Ytest,svc_prob[:,i],pos_label=i)
for i in range(df.shape[1]):
    plt.plot(range(10),df.iloc[:,i],c=color[i])
plt.legend()
plt.show()

df
```

可以观察到，逻辑回归的布里尔分数有着压倒性优势，SVC的效果明显弱于贝叶斯和逻辑回归（如同我们之前在SVC的讲解中说明过的一样，SVC是强行利用sigmoid函数来压缩概率，因此SVC产出的概率结果并不那么可靠）。贝叶斯位于逻辑回归和SVC之间，效果也不错，但比起逻辑回归，还是不够精确和稳定。

2.2.2 对数似然函数Log Loss

另一种常用的概率损失衡量是对数损失（log_loss），又叫做对数似然，逻辑损失或者交叉熵损失，它是多元逻辑回归以及一些拓展算法，比如神经网络中使用的损失函数。它被定义为，对于一个给定的概率分类器，在预测概率为条件的情况下，真实概率发生的可能性的负对数（如何得到这个损失函数的证明过程和推导过程在逻辑回归的章节中有完整得呈现）。**由于是损失，因此对数似然函数的取值越小，则证明概率估计越准确，模型越理想。**值得注意得是，对数损失只能用于评估分类型模型。

对于一个样本，如果样本的真实标签y_{true}在{0,1}中取值，并且这个样本在类别1下的概率估计为y_{pred}，则这个样本所对应的对数损失是：

$$-logP(y_{true}|y_{pred}) = -(y_{true} * log(y_{pred}) + (1 - y_{true}) * log(1 - y_{pred}))$$

和我们逻辑回归的损失函数一模一样：

$$J(\theta) = -\sum_{i=1}^{m}(y_i * log(y_\theta(x_i)) \ + \ (1-y_i) * log(1-y_\theta(x_i)))$$

只不过在逻辑回归的损失函数中，我们的真实标签是由y_i表示，预测值（概率估计）是由$y_\theta(x_i)$来表示，仅仅是表示方式的不同。注意，这里的log表示以e为底的自然对数。

在sklearn中，我们可以从metrics模块中导入我们的对数似然函数：

```
from sklearn.metrics import log_loss

log_loss(Ytest,prob)
log_loss(Ytest,logi.predict_proba(Xtest))
log_loss(Ytest,svc_prob)
```

第一个参数是真实标签，第二个参数是我们预测的概率。如果我们使用shift tab来查看log_loss的参数，会发现第二个参数写着y_pred，这会让人误解为这是我们的预测标签。由于log_loss是专门用于产出概率的算法的，因此它假设我们预测出的y就是以概率形式呈现，但在sklearn当中，我们的y_pred往往是已经根据概率归类后的类别{0，1，2}，真正的概率必须要以接口predict_proba来调用，千万避免混淆。

注意到，**我们用log_loss得出的结论和我们使用布里尔分数得出的结论不一致**：当使用布里尔分数作为评判标准的时候，SVC的估计效果是最差的，逻辑回归和贝叶斯的结果相接近。而使用对数似然的时候，虽然依然是逻辑回归最强大，但贝叶斯却没有SVC的效果好。为什么会有这样的不同呢？

因为逻辑回归和SVC都是以最优化为目的来求解模型，然后进行分类的算法。而朴素贝叶斯中，却没有最优化的过程。对数似然函数直接指向模型最优化的方向，甚至就是逻辑回归的损失函数本身，因此在逻辑回归和SVC上表现得更好。

那什么时候使用对数似然，什么时候使用布里尔分数？

在现实应用中，对数似然函数是概率类模型评估的黄金指标，往往是我们评估概率类模型的优先选择。但是它也有一些缺点，首先它没有界，不像布里尔分数有上限，可以作为模型效果的参考。其次，它的解释性不如布里尔分数，很难与非技术人员去交流对数似然存在的可靠性和必要性。第三，它在以最优化为目标的模型上明显表现更好。而且，它还有一些数学上的问题，比如不能接受为0或1的概率，否则的话对数似然就会取到极限值（考虑以e为底的自然对数在取到0或1的时候的情况）。所以因此通常来说，我们有以下使用规则：

需求	优先使用对数似然	优先使用布里尔分数
衡量模型	要对比多个模型，或者衡量模型的不同变化	衡量单一模型的表现
可解释性	机器学习和深度学习之间的行家交流，学术论文	商业报告，老板开会，业务模型的衡量
最优化指向	逻辑回归，SVC	朴素贝叶斯
数学问题	概率只能无限接近于0或1，无法取到0或1	概率可以取到0或1，比如树，随机森林

回到我们的贝叶斯来看，如果贝叶斯的模型效果不如其他模型，而我们又不想更换模型，那怎么办呢？如果以精确度为指标来调整参数，贝叶斯估计是无法拯救了——不同于SVC和逻辑回归，贝叶斯的原理简单，根本没有什么可用的参数。但是产出概率的算法有自己的调节方式，就是**调节概率的校准程度**。校准程度越高，模型对概率的预测越准确，算法在做判断时就越有自信，模型就会更稳定。如果我们追求模型在概率预测上必须尽量贴近真实概率，那我们就可以使用可靠性曲线来调节概率的校准程度。

2.2.3 可靠性曲线Reliability Curve

可靠性曲线（reliability curve），又叫做概率校准曲线（probability calibration curve），可靠性图（reliability diagrams），这是一条以预测概率为横坐标，真实标签为纵坐标的曲线。我们希望预测概率和真实值越接近越好，最好两者相等，因此**一个模型/算法的概率校准曲线越靠近对角线越好。**校准曲线因此也是我们的模型评估指标之一。和布里尔分数相似，概率校准曲线是对于标签的某一类来说的，因此一类标签就会有一条曲线，或者我们可以使用一个多类标签下的平均来表示一整个模型的概率校准曲线。**但通常来说，曲线用于二分类的情况最多**，大家如果感兴趣可以自行探索多分类的情况。

根据这个思路，我们来绘制一条曲线试试看。

1. **导入需要的库和模块**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification as mc
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression as LR
from sklearn.metrics import brier_score_loss
from sklearn.model_selection import train_test_split
```

2. **创建数据集**

```
X, y = mc(n_samples=100000,n_features=20 #总共20个特征
          ,n_classes=2 #标签为2分类
          ,n_informative=2 #其中两个代表较多信息
          ,n_redundant=10 #10个都是冗余特征
          ,random_state=42)

#样本量足够大，因此使用1%的样本作为训练集
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, y
                                                ,test_size=0.99
                                                ,random_state=42)
Xtrain

np.unique(Ytrain)
```

3. **建立模型，绘制图像**

```
gnb = GaussianNB()
gnb.fit(Xtrain,Ytrain)
y_pred = gnb.predict(Xtest)
prob_pos = gnb.predict_proba(Xtest)[:,1] #我们的预测概率 - 横坐标
#Ytest - 我们的真实标签 - 横坐标

#在我们的横纵表坐标上，概率是由顺序的（由小到大），为了让图形规整一些，我们要先对预测概率和真实标签按照预测概率进行一个排序，这一点我们通过DataFrame来实现

df = pd.DataFrame({"ytrue":Ytest[:500],"probability":prob_pos[:500]})
```

```
df

df = df.sort_values(by="probability")
df.index = range(df.shape[0])

df

#紧接着我们就可以画图了
fig = plt.figure()
ax1 = plt.subplot()
ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated") #得做一条对角线来对比呀
ax1.plot(df["probability"],df["ytrue"],"s-",label="%s (%1.3f)" % ("Bayes", clf_score))
ax1.set_ylabel("True label")
ax1.set_xlabel("predcited probability")
ax1.set_ylim([-0.05, 1.05])
ax1.legend()
plt.show()
```

这个图像看起来非常可怕，完全不止所云！为什么存在这么多上下穿梭的直线？反应快的小伙伴可能很快就发现了，我们是按照预测概率的顺序进行排序的，而预测概率从0开始到1的过程中，真实取值不断在0和1之间变化，而我们是绘制折线图，因此无数个纵坐标分布在0和1的被链接起来了，所以看起来如此混乱。

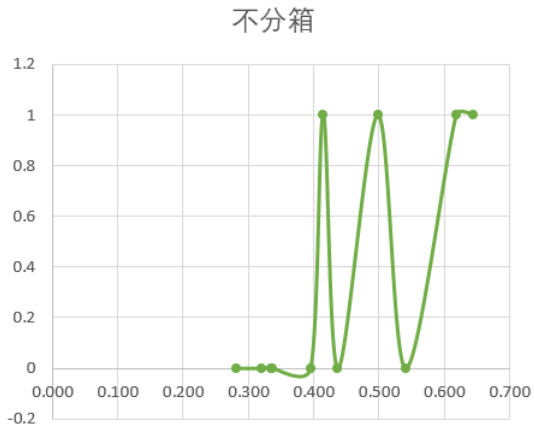
那我们换成散点图来试试看呢？

```
fig = plt.figure()
ax1 = plt.subplot()
ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
ax1.scatter(df["probability"],df["ytrue"],s=10)
ax1.set_ylabel("True label")
ax1.set_xlabel("predcited probability")
ax1.set_ylim([-0.05, 1.05])
ax1.legend()
plt.show()
```

可以看到，由于真实标签是0和1，所以所有的点都在y=1和y=0这两条直线上分布，这完全不是我们希望看到的图像。回想一下我们的可靠性曲线的横纵坐标：横坐标是预测概率，而纵坐标是真实值，我们希望预测概率很靠近真实值，那我们的真实取值必然也需要是一个概率才可以，如果使用真实标签，那我们绘制出来的图像完全是没有意义的。但是，我们去哪里寻找真实值的概率呢？这是不可能找到的——如果我们能够找到真实的概率，那我们何必还用算法来估计概率呢，直接去获取真实的概率不就好了么？所以真实概率在现实中是不可获得的。但是，我们可以获得类概率的指标来帮助我们进行校准。**一个简单的做法是，将数据进行分箱，然后规定每个箱子中真实的少数类所占的比例为这个箱上的真实概率trueproba，这个箱子中预测概率的均值为这个箱子的预测概率predproba，然后以trueproba为纵坐标，predproba为横坐标，来绘制我们的可靠性曲线。**

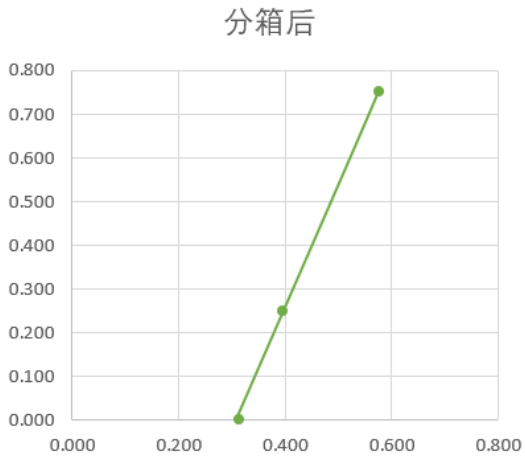
举个例子，来看下面这张表，这是一组数据不分箱时表现出来的图像：

标签为1的概率	真实标签
0.645	1
0.618	1
0.541	0
0.499	1
0.437	0
0.415	1
0.395	0
0.338	0
0.335	0
0.321	0
0.282	0



再来看看分箱之后的图像：

标签为1的概率	真实标签	箱编号	预测概率箱内平均	真实标签箱内平均
0.645	1	1	0.576	0.750
0.618	1			
0.541	0			
0.499	1			
0.437	0	2	0.396	0.250
0.415	1			
0.395	0			
0.338	0			
0.335	0	3	0.313	0.000
0.321	0			
0.282	0			



可见，分箱之后样本点的特征被聚合到了一起，曲线明显变得单调且平滑。这种分箱操作本质相当于是一种平滑，在sklearn中，这样的做法可以通过绘制可靠性曲线的类**calibration_curve**来实现。和ROC曲线类似，类calibration_curve可以帮助我们获取我们的横纵坐标，然后使用matplotlib来绘制图像。该类有如下参数：

参数	**含义**
y_true	真实标签
y_prob	预测返回的，正类别下的概率值或置信度
normalize	布尔值，默认False 是否将y_prob中输入的内容归一化到[0,1]之间，比如说，当y_prob并不是真正的概率的时候可以使用。如果这是为True，则会将y_prob中最小的值归一化为0，最大值归一化为1。
n_bins	整数值，表示分箱的个数。如果箱数很大，则需要更多的数据。
返回	**含义**
trueproba	可靠性曲线的纵坐标，结构为(n_bins,)，是每个箱子中少数类(Y=1)的占比
predproba	可靠性曲线的横坐标，结构为(n_bins,)，是每个箱子中概率的均值

4. **使用可靠性曲线的类在贝叶斯上绘制一条校准曲线**

```
from sklearn.calibration import calibration_curve

#从类calibiration_curve中获取横坐标和纵坐标
trueproba, predproba = calibration_curve(Ytest, prob_pos
                                         ,n_bins=10 #输入希望分箱的个数
                                        )

fig = plt.figure()
ax1 = plt.subplot()
ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
ax1.plot(predproba, trueproba,"s-",label="%s (%1.3f)" % ("Bayes", clf_score))
ax1.set_ylabel("True probability for class 1")
ax1.set_xlabel("Mean predcited probability")
ax1.set_ylim([-0.05, 1.05])
ax1.legend()
plt.show()
```

5. **不同的n_bins取值下曲线如何改变?**

```
fig, axes = plt.subplots(1,3,figsize=(18,4))
for ind,i in enumerate([3,10,100]):
    ax = axes[ind]
    ax.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
    trueproba, predproba = calibration_curve(Ytest, prob_pos,n_bins=i)
    ax.plot(predproba, trueproba,"s-",label="n_bins = {}".format(i))
    ax1.set_ylabel("True probability for class 1")
    ax1.set_xlabel("Mean predcited probability")
    ax1.set_ylim([-0.05, 1.05])
    ax.legend()
plt.show()
```

很明显可以看出，n_bins越大，箱子越多，概率校准曲线就越精确，但是太过精确的曲线不够平滑，无法和我们希望的完美概率密度曲线相比较。n_bins越小，箱子越少，概率校准曲线就越粗糙，虽然靠近完美概率密度曲线，但是无法真实地展现模型概率预测地结果。因此我们需要取一个既不是太大，也不是太小的箱子个数，让概率校准曲线既不是太精确，也不是太粗糙，而是一条相对平滑，又可以反应出模型对概率预测的趋势的曲线。通常来说，建议先试试看箱子数等于10的情况。箱子的数目越大，所需要的样本量也越多，否则曲线就会太过精确。

6. **建立更多模型**

```
name = ["GaussianBayes","Logistic","SVC"]

gnb = GaussianNB()
logi = LR(C=1., solver='lbfgs',max_iter=3000,multi_class="auto")
svc = SVC(kernel = "linear",gamma=1)
```

7. **建立循环，绘制多个模型的概率校准曲线**

```
fig, ax1 = plt.subplots(figsize=(8,6))
ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
```

```
for clf, name_ in zip([gnb,logi,svc],name):
    clf.fit(Xtrain,Ytrain)
    y_pred = clf.predict(Xtest)
    #hasattr(obj,name)：查看一个类obj中是否存在名字为name的接口，存在则返回True
    if hasattr(clf, "predict_proba"):
        prob_pos = clf.predict_proba(Xtest)[:,1]
    else:  # use decision function
        prob_pos = clf.decision_function(Xtest)
        prob_pos = (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())
    #返回布里尔分数
    clf_score = brier_score_loss(Ytest, prob_pos, pos_label=y.max())
    trueproba, predproba = calibration_curve(Ytest, prob_pos,n_bins=10)
    ax1.plot(predproba, trueproba,"s-",label="%s (%1.3f)" % (name_, clf_score))

ax1.set_ylabel("True probability for class 1")
ax1.set_xlabel("Mean predcited probability")
ax1.set_ylim([-0.05, 1.05])
ax1.legend()
ax1.set_title('Calibration plots  (reliability curve)')
plt.show()
```

从图像的结果来看，我们可以明显看出，逻辑回归的概率估计是最接近完美的概率校准曲线，所以逻辑虎归的效果最完美。相对的，高斯朴素贝叶斯和支持向量机分类器的结果都比较糟糕。支持向量机呈现类似于sigmoid函数的形状，而高斯朴素贝叶斯呈现和Sigmoid函数相反的形状。

对于贝叶斯，如果概率校准曲线呈现sigmoid函数的镜像的情况，则说明数据集中的特征不是相互条件独立的。贝叶斯原理中的"朴素"原则：特征相互条件独立原则被违反了（这其实是我们自己的设定，我们设定了10个冗余特征，这些特征就是噪音，他们之间不可能完全独立），因此贝叶斯的表现不够好。

而支持向量机的概率校准曲线效果其实是典型的置信度不足的分类器(under-confident classifier)的表现：**大量的样本点集中在决策边界的附近，因此许多样本点的置信度靠近0.5左右，即便决策边界能够将样本点判断正确，模型本身对这个结果也不是非常确信的。**相对的，离决策边界很远的点的置信度就会很高，因为它很大可能性上不会被判断错误。支持向量机在面对混合度较高的数据的时候，有着天生的置信度不足的缺点。

2.2.4 预测概率的直方图

我们可以通过绘制直方图来查看模型的预测概率的分布。直方图是以样本的预测概率分箱后的结果为横坐标，每个箱中的样本数量为纵坐标的一个图像。注意，这里的分箱和我们在可靠性曲线中的分箱不同，这里的分箱是将预测概率均匀分为一个个的区间，与之前可靠性曲线中为了平滑的分箱完全是两码事。我们来绘制一下我们的直方图：

```
fig, ax2 = plt.subplots(figsize=(8,6))

for clf, name_ in zip([gnb,logi,svc],name):
    clf.fit(Xtrain,Ytrain)
    y_pred = clf.predict(Xtest)
    #hasattr(obj,name)：查看一个类obj中是否存在名字为name的接口，存在则返回True
    if hasattr(clf, "predict_proba"):
        prob_pos = clf.predict_proba(Xtest)[:,1]
    else:  # use decision function
        prob_pos = clf.decision_function(Xtest)
```

```
        prob_pos = (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())
    ax2.hist(prob_pos
             ,bins=10
             ,label=name_
             ,histtype="step" #设置直方图为透明
             ,lw=2 #设置直方图每个柱子描边的粗细
            )

ax2.set_ylabel("Distribution of probability")
ax2.set_xlabel("Mean predicted probability")
ax2.set_xlim([-0.05, 1.05])
ax2.set_xticks([0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1])
ax2.legend(loc=9)
plt.show()
```

可以看到，高斯贝叶斯的概率分布是两边非常高，中间非常低，几乎90%以上的样本都在0和1的附近，可以说是置信度最高的算法，但是贝叶斯的布里尔分数却不如逻辑回归，这证明贝叶斯中在0和1附近的样本中有一部分是被分错的。支持向量贝叶斯完全相反，明显是中间高，两边低，类似于正态分布的状况，证明了我们刚才所说的，大部分样本都在决策边界附近，置信度都徘徊在0.5左右的情况。而逻辑回归位于高斯朴素贝叶斯和支持向量机的中间，即没有太多的样本过度靠近0和1，也没有形成像支持向量机那样的正态分布。一个比较健康的正样本的概率分布，就是逻辑回归的直方图显示出来的样子。

> **避免混淆：概率密度曲线和概率分布直方图**
>
> 大家也许还记得我们说过，我们是假设样本的概率分布为高斯分布，然后使用高斯的方程来估计连续型变量的概率。怎么现在我们绘制出的概率分布结果中，高斯普斯贝叶斯的概率分布反而完全不是高斯分布了呢？
> 注意，千万不要把**概率密度曲线**和**概率分布直方图**混淆。
> 在称重汉堡的时候所绘制的曲线，是概率密度曲线，横坐标是样本的取值，纵坐标是落在这个样本取值区间中的样本个数，衡量的是每个X的取值区间之内有多少样本。服从高斯分布的是X的取值上的样本分布。
> 现在我们的概率分布直方图，横坐标是概率的取值[0,1]，纵坐标是落在这个概率取值范围中的样本的个数，衡量的是每个概率取值区间之内有多少样本。这个分布，是没有任何假设的。

我们已经得知了朴素贝叶斯和SVC预测概率的效果各方面都不如逻辑回归，那在这种情况下，我们如何来帮助模型或者算法，让他们对自己的预测更有信心，置信度更高呢？我们可以**使用等近似回归来矫正概率算法。**

2.2.5 校准可靠性曲线

等近似回归有两种回归可以使用，一种是基于Platt的Sigmoid模型的参数校准方法，一种是基于等渗回归（isotonic calibration）的非参数的校准方法。概率校准应该发生在测试集上，必须是模型未曾见过的数据。在数学上，使用这两种方式来对概率进行校准的原理十分复杂，而此过程我们在sklearn中无法进行干涉，大家不必过于去深究，如果希望深入研究利用回归校准概率的细节，可以查看sklearn中的如下案例。这是一个基于鸢尾花数据集的，三分类数据上的概率校准过程，大家如果感兴趣可以仔细研究：

https://scikit-learn.org/stable/auto_examples/calibration/plot_calibration_multiclass.html#sphx-glr-auto-examples-calibration-plot-calibration-multiclass-py

在这里，我主要来为大家展示如果使用sklearn中的概率校正类CalibratedClassifierCV来对二分类情况下的数据集进行概率校正。

class `sklearn.calibration.CalibratedClassifierCV` (*base_estimator=None*, *method='sigmoid'*, *cv='warn'*)

这是一个带交叉验证的概率校准类，它使用交叉验证生成器，对交叉验证中的每一份数据，它都在训练样本上进行模型参数估计，在测试样本上进行概率校准，然后为我们返回最佳的一组参数估计和校准结果。每一份数据的预测概率会被求解平均。**注意，类CalibratedClassifierCV**没有接口decision_function，要查看这个类下校准过后的模型生成的概率，必须调用predict_proba接口。

base_estimator

需要校准其输出决策功能的分类器，必须存在predict_proba或decision_function接口。 如果参数cv = prefit，分类器必须已经拟合数据完毕。

cv

整数，确定交叉验证的策略。可能输入是：

- None，表示使用默认的3折交叉验证
- 任意整数，指定折数

  对于输入整数和None的情况下来说，如果时二分类，则自动使用类sklearn.model_selection.StratifiedKFold进行折数分割。如果y是连续型变量，则使用sklearn.model_selection.KFold进行分割。
- 已经使用其他类建好的交叉验证模式或生成器cv
- 可迭代的，已经分割完毕的测试集和训练集索引数组
- 输入"prefit"，则假设已经在分类器上拟合完毕数据。在这种模式下，使用者必须手动确定用来拟合分类器的数据与即将倍校准的数据没有交集

在版本0.20中更改：在0.22版本中输入“None”，将由使用3折交叉验证改为5折交叉验证

method

进行概率校准的方法，可输入"sigmoid"或者"isotonic"

- 输入'sigmoid'，使用基于Platt的Sigmoid模型来进行校准
- 输入'isotonic'，使用等渗回归来进行校准

当校准的样本量太少（比如，小于等于1000个测试样本）的时候，不建议使用等渗回归，因为它倾向于过拟合。样本量过少时请使用sigmoids，即Platt校准。

我们依然来使用之前建立的数据集。

1. **包装函数**

首先，我们将之前绘制可靠性曲线和直方图的代码包装成函数。考虑函数的参数为：模型，模型的名字，数据，和需要分箱的个数。我们在这里将直方图和可靠性曲线打包在同一个函数中，让他们并排显示。

```
def plot_calib(models,name,Xtrain,Xtest,Ytrain,Ytest,n_bins=10):

    import matplotlib.pyplot as plt
    from sklearn.metrics import brier_score_loss
    from sklearn.calibration import calibration_curve
```

```
    fig, (ax1, ax2) = plt.subplots(1, 2,figsize=(20,6))
    ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")

    for clf, name_ in zip(models,name):
        clf.fit(Xtrain,Ytrain)
        y_pred = clf.predict(Xtest)
        #hasattr(obj,name)：查看一个类obj中是否存在名字为name的接口，存在则返回True
        if hasattr(clf, "predict_proba"):
            prob_pos = clf.predict_proba(Xtest)[:,1]
        else:  # use decision function
            prob_pos = clf.decision_function(Xtest)
            prob_pos = (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())
        #返回布里尔分数
        clf_score = brier_score_loss(Ytest, prob_pos, pos_label=y.max())
        trueproba, predproba = calibration_curve(Ytest, prob_pos,n_bins=n_bins)
        ax1.plot(predproba, trueproba,"s-",label="%s (%1.3f)" % (name_, clf_score))
        ax2.hist(prob_pos, range=(0, 1), bins=n_bins, label=name_,histtype="step",lw=2)

    ax2.set_ylabel("Distribution of probability")
    ax2.set_xlabel("Mean predicted probability")
    ax2.set_xlim([-0.05, 1.05])
    ax2.legend(loc=9)
    ax2.set_title("Distribution of probablity")
    ax1.set_ylabel("True probability for class 1")
    ax1.set_xlabel("Mean predcited probability")
    ax1.set_ylim([-0.05, 1.05])
    ax1.legend()
    ax1.set_title('Calibration plots(reliability curve)')
    plt.show()
```

2. **设实例化模型，设定模型的名字**

```
from sklearn.calibration import CalibratedClassifierCV

name = ["GaussianBayes","Logistic","Bayes+isotonic","Bayes+sigmoid"]

gnb = GaussianNB()

models = [gnb
          ,LR(C=1., solver='lbfgs',max_iter=3000,multi_class="auto")
        #定义两种校准方式
          ,CalibratedClassifierCV(gnb, cv=2, method='isotonic')
          ,CalibratedClassifierCV(gnb, cv=2, method='sigmoid')]
```

3. **基于函数进行绘图**

```
plot_calib(models,name,Xtrain,Xtest,Ytrain,Ytest)
```

从校正朴素贝叶斯的结果来看，Isotonic等渗校正大大改善了曲线的形状，几乎让贝叶斯的效果与逻辑回归持平，并且布里尔分数也下降到了0.098，比逻辑回归还低一个点。Sigmoid校准的方式也对曲线进行了稍稍的改善，不过效果不明显。从直方图来看，Isotonic校正让高斯朴素贝叶斯的效果接近逻辑回归，而Sigmoid校正后的结果依然和原本的高斯朴素贝叶斯更相近。可见，当数据的特征之间不是相互条件独立的时候，使用Isotonic方式来校准概率曲线，可以得到不错的结果，让模型在预测上更加谦虚。

4. **基于校准结果查看精确性的变化**

```
gnb = GaussianNB().fit(Xtrain,Ytrain)
gnb.score(Xtest,Ytest)

brier_score_loss(Ytest,gnb.predict_proba(Xtest)[:,1],pos_label = 1)

gnbisotonic = CalibratedClassifierCV(gnb, cv=2, method='isotonic').fit(Xtrain,Ytrain)
gnbisotonic.score(Xtest,Ytest)

brier_score_loss(Ytest,gnbisotonic.predict_proba(Xtest)[:,1],pos_label = 1)
```

可以看出，校准概率后，布里尔分数明显变小了，但整体的准确率却略有下降，这证明算法在校准之后，尽管对概率的预测更准确了，但模型的判断力略有降低。来思考一下：布里尔分数衡量模型概率预测的准确率，布里尔分数越低，代表模型的概率越接近真实概率，当进行概率校准后，本来标签是1的样本的概率应该会更接近1，而标签本来是0的样本应该会更接近0，没有理由布里尔分数提升了，模型的判断准确率居然下降了。但从我们的结果来看，模型的准确率和概率预测的正确性并不是完全一致的，为什么会这样呢？

对于不同的概率类模型，原因是不同的。对于SVC，决策树这样的模型来说，概率不是真正的概率，而更偏向于是一个“置信度”，这些模型也不是依赖于概率预测来进行分类（决策树依赖于树权而SVC依赖于决策边界），因此对于这些模型，可能存在着类别1下的概率为0.4但样本依然被分类为1的情况，这种情况代表着——模型很没有信心认为这个样本是1，但是还是坚持把这个样本的标签分类为1了。这种时候，概率校准可能会向着更加错误的方向调整（比如把概率为0.4的点调节得更接近0，导致模型最终判断错误），因此出现布里尔分数可能会显示和精确性相反的趋势。

而对于朴素贝叶斯这样的模型，却是另一种情况。注意在朴素贝叶斯中，我们有各种各样的假设，除了我们的“朴素”假设，还有我们对概率分布的假设（比如说高斯），这些假设使得我们的贝叶斯得出的概率估计其实是有偏估计，也就是说，这种概率估计其实不是那么准确和严肃。我们通过校准，让模型的预测概率更贴近于真实概率，本质是在统计学上让算法更加贴近我们对整体样本状况的估计，这样的一种校准在一组数据集上可能表现出让准确率上升，也可能表现出让准确率下降，这取决于我们的测试集有多贴近我们估计的真实样本的面貌。这一系列有偏估计使得我们在概率校准中可能出现布里尔分数和准确度的趋势相反的情况。

当然，可能还有更多更深层的原因，比如概率校准过程中的数学细节如何影响了我们的校准，类calibration_curve中是如何分箱，如何通过真实标签和预测值来生成校准曲线使用的横纵坐标的，这些过程中也可能有着让布里尔分数和准确率向两个方向移动的过程。

在现实中，**当两者相悖的时候，请务必以准确率为标准。**但是这不代表说布里尔分数和概率校准曲线就无效了。概率类模型几乎没有参数可以调整，除了换模型之外，鲜有更好的方式帮助我们提升模型的表现，概率校准是难得的可以帮助我们针对概率提升模型的方法。

5. **试试看对于SVC，哪种校准更有效呢？**

```
name_svc = ["SVC","Logistic","SVC+isotonic","SVC+sigmoid"]

svc = SVC(kernel = "linear",gamma=1)

models_svc = [svc
              ,LR(C=1., solver='lbfgs',max_iter=3000,multi_class="auto")
              #依然定义两种校准方式
              ,CalibratedClassifierCV(svc, cv=2, method='isotonic')
              ,CalibratedClassifierCV(svc, cv=2, method='sigmoid')]

plot_calib(models_svc,name_svc,Xtrain,Xtest,Ytrain,Ytest)
```

可以看出，对于SVC，sigmoid和isotonic的校准效果都非常不错，无论是从校准曲线来看还是从概率分布图来看，两种校准都让SVC的结果接近逻辑回归，其中sigmoid更加有效。来看看不同的SVC下的精确度结果（对于这一段代码，大家完全可以把它包括在原有的绘图函数中）：

```
name_svc = ["SVC","SVC+isotonic","SVC+sigmoid"]

svc = SVC(kernel = "linear",gamma=1)

models_svc = [svc
              ,CalibratedClassifierCV(svc, cv=2, method='isotonic')
              ,CalibratedClassifierCV(svc, cv=2, method='sigmoid')]

for clf, name in zip(models_svc,name_svc):
    clf.fit(Xtrain,Ytrain)
    y_pred = clf.predict(Xtest)
    if hasattr(clf, "predict_proba"):
        prob_pos = clf.predict_proba(Xtest)[:, 1]
    else:
        prob_pos = clf.decision_function(Xtest)
        prob_pos = (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())
    clf_score = brier_score_loss(Ytest, prob_pos, pos_label=y.max())
    score = clf.score(Xtest,Ytest)
    print("{}:".format(name))
    print("\tBrier:{:.4f}".format(clf_score))
    print("\tAccuracy:{:.4f}".format(score))
```

可以看到，对于SVC来说，两种校正都改善了准确率和布里尔分数。可见，概率校正对于SVC非常有效。这也说明，**概率校正对于原本的可靠性曲线是形容Sigmoid形状的曲线的算法比较有效。**

在现实中，我们可以选择调节模型的方向，我们不一定要追求最高的准确率或者追求概率拟合最好，我们可以根据自己的需求来调整模型。当然，对于概率类模型来说，由于可以调节的参数甚少，所以我们更倾向于追求概率拟合，并使用概率校准的方式来调节模型。如果你的确希望追求更高的准确率和Recall，可以考虑使用天生就非常准确的概率类模型逻辑回归，也可以考虑使用除了概率校准之外还有很多其他参数可调的支持向量机分类器。

2.3 多项式朴素贝叶斯以及其变化

2.3.1 多项式朴素贝叶斯MultinomialNB

多项式贝叶斯可能是除了高斯之外，最为人所知的贝叶斯算法了。它也是基于原始的贝叶斯理论，但假设概率分布是服从一个简单多项式分布。多项式分布来源于统计学中的多项式实验，这种实验可以具体解释为：实验包括n次重复试验，每项试验都有不同的可能结果。在任何给定的试验中，特定结果发生的概率是不变的。

举个例子，比如说一个特征矩阵$\boldsymbol{X}$表示投掷硬币的结果，则得到正面的概率为$P(X = \text{正面}|Y)$ = 0.5，得到反面的概率为$P(X = \text{反面}|Y)$ = 0.5，只有这两种可能并且两种结果互不干涉，并且两个随机事件的概率加和为1，这就是一个二项分布。这种情况下，适合于多项式朴素贝叶斯的特征矩阵应该长这样：

测试编号	X_1：**出现正面**	X_2：**出现反面**
0	0	1
1	1	0
2	1	0
3	0	1

假设另一个特征X'表示投掷骰子的结果，则i就可以在[1,2,3,4,5,6]中取值，六种结果互不干涉，且只要样本量足够大，概率都为$1/6$，这就是一个多项分布。多项分布的特征矩阵应该长这样：

测试编号	**出现1**	**出现2**	**出现3**	**出现4**	**出现5**	**出现6**
0	1	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	1	0	0	0
…						
m	0	0	0	0	0	1

可以看出：

1. **多项式分布擅长的是分类型变量**，在其原理假设中，$P(x_i|Y)$的概率是离散的，并且不同x_i下的$P(x_i|Y)$相互独立，互不影响。虽然sklearn中的多项式分布也可以处理连续型变量，但现实中，如果我们真的想要处理连续型变量，我们应当使用高斯朴素贝叶斯。
2. 多项式实验中的实验结果都很具体，它所涉及的特征往往是次数，频率，计数，出现与否这样的概念，这些概念都是离散的正整数，因此**sklearn中的多项式朴素贝叶斯不接受负值的输入。**

由于这样的特性，多项式朴素贝叶斯的特征矩阵经常是稀疏矩阵（不一定总是稀疏矩阵），并且它经常被用于文本分类。我们可以使用著名的**TF-IDF向量技术**，也可以使用常见并且简单的**单词计数向量**手段与贝叶斯配合使用。这两种手段都属于常见的文本特征提取的方法，可以很简单地通过sklearn来实现，在案例中我们会来详细讲到。

从数学的角度来看，在一种标签类别$Y = c$下，我们有一组分别对应特征的参数向量$\boldsymbol{\theta_c} = (\boldsymbol{\theta_{c1}}, \boldsymbol{\theta_{c2}}, \ldots, \boldsymbol{\theta_{cn}})$，其中n表示特征的总数。一个$\boldsymbol{\theta_{ci}}$表示这个标签类别下的第$i$个特征所对应的参数。这个参数被我们定义为：

$$\boldsymbol{\theta_{ci}} = \frac{\text{特征}X_i\text{在}Y = c\text{这个分类下的所有样本的取值总和}}{\text{所有特征在}Y = c\text{这个分类下的所有样本的取值总和}}$$

记作$P(\boldsymbol{X_i}|Y=c)$，表示当Y=c这个条件固定的时候，一组样本在X_i这个特征上的取值被取到的概率。**注意，我们在高斯朴素贝叶斯中求解的概率$P(x_i|Y)$是对于一个样本来说，而我们现在求解的$P(\boldsymbol{X_i}|Y=c)$是对于一个特征X_i来说的概率。**

对于一个在标签类别$Y=c$下，结构为(m, n)的特征矩阵来说，我们有：

$$\boldsymbol{X_y}=\begin{bmatrix} x_{11} & x_{12} & x_{13} & \cdots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \cdots & x_{2n} \\ x_{31} & x_{32} & x_{33} & \cdots & x_{3n} \\ & & \cdots & & \\ x_{m1} & x_{m2} & x_{m3} & \cdots & x_{mn} \end{bmatrix}$$

其中每个x_{ji}都是特征X_i发生的次数。基于这些理解，我们通过平滑后的最大似然估计来求解参数θ_y：

$$\theta_{ci}=\frac{\sum_{y_j=c}x_{ji}+\alpha}{\sum_{i=1}^{n}\sum_{y_j=c}x_{ji}+\alpha n}$$

对于每个特征，$\sum_{y_j=c}x_{ji}$是特征X_i下所有标签为c的样本的特征取值之和，其实就是特征矩阵中每一列的和。$\sum_{i=1}^{n}\sum_{y_j=c}x_{ji}$是所有标签类别为c的样本上，所有特征的取值之和，其实就是特征矩阵$\boldsymbol{X_y}$中所有元素的和。α被称为平滑系数，**我们令$\alpha>0$来防止训练数据中出现过的一些词汇没有出现在测试集中导致的0概率，以避免让参数θ为0的情况。**如果我们将α设置为1，则这个平滑叫做拉普拉斯平滑，如果α小于1，则我们把它叫做利德斯通平滑。两种平滑都属于自然语言处理中比较常用的用来平滑分类数据的统计手段。

在sklearn中，用来执行多项式朴素贝叶斯的类MultinomialNB包含如下的参数和属性：

class `sklearn.naive_bayes.MultinomialNB`(*alpha=1.0*, *fit_prior=True*, *class_prior=None*)

参数
alpha : 浮点数, 可不填 (默认为1.0) 拉普拉斯或利德斯通平滑的参数α，如果设置为0则表示完全没有平滑选项。但是需要注意的是，平滑相当于人为给概率加上一些噪音，因此α设置得越大，多项式朴素贝叶斯的精确性会越低（虽然影响不是非常大），布里尔分数也会逐渐升高。
fit_prior : 布尔值, 可不填 (默认为True) 是否学习先验概率$P(Y=c)$。如果设置为false，则不使用先验概率，而使用统一先验概率（uniform prior），即认为每个标签类出现的概率是$\frac{1}{n_classes}$。
class_prior：形似数组的结构，结构为(n_classes,)，可不填（默认为None) 类的先验概率$P(Y=c)$。如果没有给出具体的先验概率则自动根据数据来进行计算。

通常我们在实例化多项式朴素贝叶斯的时候，我们会让所有的参数保持默认。先来简单建一个多项式朴素贝叶斯的例子试试看：

1. **导入需要的模块和库**

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_blobs
from sklearn.metrics import brier_score_loss
```

2. **建立数据集**

```
class_1 = 500
class_2 = 500 #两个类别分别设定500个样本
centers = [[0.0, 0.0], [2.0, 2.0]] #设定两个类别的中心
clusters_std = [0.5, 0.5] #设定两个类别的方差
X, y = make_blobs(n_samples=[class_1, class_2],
                  centers=centers,
                  cluster_std=clusters_std,
                  random_state=0, shuffle=False)

Xtrain, Xtest, Ytrain, Ytest = train_test_split(X,y
                                                ,test_size=0.3
                                                ,random_state=420)
```

3. **归一化，确保输入的矩阵不带有负数**

```
#先归一化，保证输入多项式朴素贝叶斯的特征矩阵中不带有负数
mms = MinMaxScaler().fit(Xtrain)
Xtrain_ = mms.transform(Xtrain)
Xtest_ = mms.transform(Xtest)
```

4. **建立一个多项式朴素贝叶斯分类器吧**

```
mnb = MultinomialNB().fit(Xtrain_, Ytrain)

#重要属性：调用根据数据获取的，每个标签类的对数先验概率log(P(Y))
#由于概率永远是在[0,1]之间，因此对数先验概率返回的永远是负值
mnb.class_log_prior_

np.unique(Ytrain)

(Ytrain == 1).sum()/Ytrain.shape[0]

mnb.class_log_prior_.shape

#可以使用np.exp来查看真正的概率值
np.exp(mnb.class_log_prior_)

#重要属性：返回一个固定标签类别下的每个特征的对数概率log(P(Xi|y))
mnb.feature_log_prob_

mnb.feature_log_prob_.shape
```

```
#重要属性：在fit时每个标签类别下包含的样本数。当fit接口中的sample_weight被设置时，该接口返回的值也会受到加权的影响
mnb.class_count_

mnb.class_count_.shape
```

5. **那分类器的效果如何呢？**

```
#一些传统的接口
mnb.predict(Xtest_)

mnb.predict_proba(Xtest_)

mnb.score(Xtest_,Ytest)

brier_score_loss(Ytest,mnb.predict_proba(Xtest_)[:,1],pos_label=1)
```

7. **效果不太理想，思考一下多项式贝叶斯的性质，我们能够做点什么呢？**

```
#来试试看把Xtiain转换成分类型数据吧
#注意我们的Xtrain没有经过归一化，因为做哑变量之后自然所有的数据就不会又负数了
from sklearn.preprocessing import KBinsDiscretizer
kbs = KBinsDiscretizer(n_bins=10, encode='onehot').fit(Xtrain)
Xtrain_ = kbs.transform(Xtrain)
Xtest_ = kbs.transform(Xtest)

mnb = MultinomialNB().fit(Xtrain_, Ytrain)

mnb.score(Xtest_,Ytest)

brier_score_loss(Ytest,mnb.predict_proba(Xtest_)[:,1],pos_label=1)
```

可以看出，多项式朴素贝叶斯的基本操作和代码都非常简单。同样的数据，如果采用哑变量方式的分箱处理，多项式贝叶斯的效果会突飞猛进。作为在文本分类中大放异彩的算法，我们将会在案例中来详细讲解多项式贝叶斯的使用，并为大家介绍文本分类的更多细节。

2.3.2 伯努利朴素贝叶斯BernoulliNB

多项式朴素贝叶斯可同时处理二项分布（抛硬币）和多项分布（掷骰子），其中二项分布又叫做伯努利分布，它是一种现实中常见，并且拥有很多优越数学性质的分布。因此，既然有着多项式朴素贝叶斯，我们自然也就又专门用来处理二项分布的朴素贝叶斯：伯努利朴素贝叶斯。

伯努利贝叶斯类BernoulliN假设数据服从多元伯努利分布，并在此基础上应用朴素贝叶斯的训练和分类过程。多元伯努利分布简单来说，就是数据集中可以存在多个特征，但每个特征都是二分类的，可以以布尔变量表示，也可以表示为{0，1}或者{-1，1}等任意二分类组合。因此，这个类要求将样本转换为二分类特征向量，如果数据本身不是二分类的，那可以使用类中专门用来二值化的参数binarize来改变数据。

伯努利朴素贝叶斯与多项式朴素贝叶斯非常相似，都常用于处理文本分类数据。但由于伯努利朴素贝叶斯是处理二项分布，所以它更加在意的是"存在与否"，而不是"出现多少次"这样的次数或频率，这是伯努利贝叶斯与多项式贝叶斯的根本性不同。在文本分类的情况下，伯努利朴素贝叶斯可以使用单词出现向量（而不是单词计数向量）来训练分类器。文档较短的数据集上，伯努利朴素贝叶斯的效果会更加好。如果时间允许，建议两种模型都试试看。

来看看伯努利朴素贝叶斯类的参数：

class `sklearn.naive_bayes.BernoulliNB`(*alpha=1.0*, *binarize=0.0*, *fit_prior=True*, *class_prior=None*)

伯努利朴素贝叶斯
alpha : 浮点数, 可不填 (默认为1.0) 拉普拉斯或利德斯通平滑的参数α，如果设置为0则表示完全没有平滑选项。但是需要注意的是，平滑相当于人为给概率加上一些噪音，因此α设置得越大，多项式朴素贝叶斯的精确性会越低（虽然影响不是非常大），布里尔分数也会逐渐升高。
binarize : 浮点数或None，可不填，默认为0 将特征二值化的阈值，如果设定为None，则会假定说特征已经被二值化完毕
fit_prior : 布尔值, 可不填 (默认为True) 是否学习先验概率$P(Y=c)$。如果设置为false，则不使用先验概率，而使用统一先验概率（uniform prior），即认为每个标签类出现的概率是$\frac{1}{n_classes}$。
class_prior：形似数组的结构，结构为(n_classes,)，可不填（默认为None) 类的先验概率$P(Y=c)$。如果没有给出具体的先验概率则自动根据数据来进行计算。

在sklearn中，伯努利朴素贝叶斯的实现也非常简单：

```
from sklearn.naive_bayes import BernoulliNB

#普通来说我们应该使用二值化的类sklearn.preprocessing.Binarizer来将特征一个个二值化
#然而这样效率过低，因此我们选择归一化之后直接设置一个阈值

mms = MinMaxScaler().fit(Xtrain)
Xtrain_ = mms.transform(Xtrain)
Xtest_ = mms.transform(Xtest)

#不设置二值化
bnl_ = BernoulliNB().fit(Xtrain_, Ytrain)
bnl_.score(Xtest_,Ytest)
brier_score_loss(Ytest,bnl_.predict_proba(Xtest_)[:,1],pos_label=1)

#设置二值化阈值为0.5
bnl = BernoulliNB(binarize=0.5).fit(Xtrain_, Ytrain)
bnl.score(Xtest_,Ytest)
brier_score_loss(Ytest,bnl.predict_proba(Xtest_)[:,1],pos_label=1)
```

和多项式贝叶斯一样，伯努利贝叶斯的结果也受到数据结构非常大的影响。因此，根据数据的模样选择贝叶斯，是贝叶斯模型选择中十分重要的一点。

2.3.3 探索贝叶斯：贝叶斯的样本不均衡问题

接下来，我们来探讨一个分类算法永远都逃不过的核心问题：样本不平衡。贝叶斯由于分类效力不算太好，因此对样本不平衡极为敏感，我们接下来就来看一看样本不平衡如何影响了贝叶斯。

1. **导入需要的模块，建立样本不平衡的数据集**

```
from sklearn.naive_bayes import MultinomialNB, GaussianNB, BernoulliNB
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_blobs
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.metrics import brier_score_loss as BS,recall_score,roc_auc_score as AUC

class_1 = 50000 #多数类为50000个样本
class_2 = 500 #少数类为500个样本
centers = [[0.0, 0.0], [5.0, 5.0]] #设定两个类别的中心
clusters_std = [3, 1] #设定两个类别的方差
X, y = make_blobs(n_samples=[class_1, class_2],
                  centers=centers,
                  cluster_std=clusters_std,
                  random_state=0, shuffle=False)
X.shape

np.unique(y)
```

2. **查看所有贝叶斯在样本不平衡数据集上的表现**

```
name = ["Multinomial","Gaussian","Bernoulli"]
models = [MultinomialNB(),GaussianNB(),BernoulliNB()]

for name,clf in zip(name,models):
    Xtrain, Xtest, Ytrain, Ytest = train_test_split(X,y
                                                    ,test_size=0.3
                                                    ,random_state=420)
    if name != "Gaussian":
        kbs = KBinsDiscretizer(n_bins=10, encode='onehot').fit(Xtrain)
        Xtrain = kbs.transform(Xtrain)
        Xtest = kbs.transform(Xtest)

    clf.fit(Xtrain,Ytrain)
    y_pred = clf.predict(Xtest)
    proba = clf.predict_proba(Xtest)[:,1]
    score = clf.score(Xtest,Ytest)
    print(name)
    print("\tBrier:{:.3f}".format(BS(Ytest,proba,pos_label=1)))
    print("\tAccuracy:{:.3f}".format(score))
    print("\tRecall:{:.3f}".format(recall_score(Ytest,y_pred)))
    print("\tAUC:{:.3f}".format(AUC(Ytest,proba)))
```

从结果上来看，多项式朴素贝叶斯判断出了所有的多数类样本，但放弃了全部的少数类样本，受到样本不均衡问题影响最严重。高斯比多项式在少数类的判断上更加成功一些，至少得到了43.8%的recall。伯努利贝叶斯虽然整体的准确度和布里尔分数不如多项式和高斯朴素贝叶斯和，但至少成功捕捉出了77.1%的少数类。可见，伯努利贝叶斯最能够忍受样本不均衡问题。

可是，伯努利贝叶斯只能用于处理二项分布数据，在现实中，强行将所有的数据都二值化不会永远得到好结果，在我们有多个特征的时候，我们更需要一个个去判断究竟二值化的阈值该取多少才能够让算法的效果优秀。这样做无疑是非常低效的。那如果我们的目标是捕捉少数类，我们应该怎么办呢？高斯朴素贝叶斯的效果虽然比多项式好，但是也没有好到可以用来帮助我们捕捉少数类的程度——43.8%，还不如抛硬币的结果。因此，孜孜不倦的统计学家们改进了朴素贝叶斯算法，修正了包括无法处理样本不平衡在内的传统朴素贝叶斯的众多缺点，得到了新兴贝叶斯算法：补集朴素贝叶斯。

2.3.4 改进多项式朴素贝叶斯：补集朴素贝叶斯ComplementNB

补集朴素贝叶斯（complement naive Bayes，CNB）算法是标准多项式朴素贝叶斯算法的改进。CNB的发明小组创造出CNB的初衷是为了解决贝叶斯中的"朴素"假设带来的各种问题，他们希望能够创造出数学方法以逃避朴素贝叶斯中的朴素假设，让算法能够不去关心所有特征之间是否是条件独立的。以此为基础，他们创造出了能够解决样本不平衡问题，并且能够一定程度上忽略朴素假设的补集朴素贝叶斯。在实验中，CNB的参数估计已经被证明比普通多项式朴素贝叶斯更稳定，并且它特别适合于样本不平衡的数据集。有时候，CNB在文本分类任务上的表现有时能够优于多项式朴素贝叶斯，因此现在补集朴素贝叶斯也开始逐渐流行。

关于补集朴素贝叶斯具体是如何逃避了我们的朴素假设，或者如何让我们的样本不均衡问题得到了改善，背后有深刻的数学原理和复杂的数学证明过程，大家如果感兴趣可以参阅这篇论文：

- Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003). Tackling the poor assumptions of naive bayes text classifiers. In ICML (Vol. 3, pp. 616-623).

简单来说，CNB使用来自每个标签类别的补集的概率，并以此来计算每个特征的权重。

$$\hat{\theta}_{i,y\neq c} = \frac{\alpha_i + \sum_{y_j \neq c} x_{ij}}{\alpha_i n + \sum_{i,y\neq c} \sum_{i=1}^{n} x_{ij}}$$

其中j表示每个样本，x_{ij}表示在样本j上对于特征i的下的取值，在文本分类中通常是计数的值或者是TF-IDF值。α是像标准多项式朴素贝叶斯中一样的平滑系数。可以看出，这个看似复杂的公式其实很简单，$\sum_{y_j \neq c} x_{ij}$其实指的就是，一个特征i下，所有标签类别不等于c值的样本的特征取值之和。而$\sum_{i,y\neq c} \sum_{i=1}^{n} x_{ij}$其实就是，所有特征下，素有标签类别不等于c值得样本的特征取值之和。其实就是多项式分布的逆向思路。

$$w_{ci} = log\ \hat{\theta}_{i,y\neq c}$$

或者我们可以选择：

$$w_{ci} = \frac{log\ \hat{\theta}_{i,y\neq c}}{\sum_j |log\ \hat{\theta}_{i,y\neq c}|}$$

对于这个概率，我们对它取对数后得到权重。我们还可以选择除以它的L2范式，以解决了在多项式分布中，特征取值比较多的样本（比如说比较长的文档）支配参数估计的情况。很多时候我们的特征矩阵是稀疏矩阵，但也不排除在有一些随机事件中，可以一次在两个特征中取值的情况。如果一个样本下的很多个随机事件可以同时发生，并且互不干涉，那么这个样本上可能有很多个特征下都有取值——文本分类中就有很多这样的情况。比如说我们可以有如下特征矩阵：

索引	X1	X2
0	1	1
1	0	1

这种状况下，索引为0的样本就会在参数估计中占更多的权重。

更甚，如果一个样本下的很多个随机事件同时发生，还在一次实验中发生了多次，那这个样本在参数估计中也会占有更大的权重。

索引	X1	X2
0	5	1
1	0	1

基于这个权重，补充朴素贝叶斯中一个样本的预测规则为：

$$p(Y \neq c|\boldsymbol{X}) = \arg\min_{c} \sum_{i} x_i w_{ci}$$

即我们求解出的最小补集概率所对应的标签就是样本的标签，因为$Y \neq c$的概率越小，则意味着$Y = c$的概率越大，所以样本属于标签类别c。

在sklearn中，补集朴素贝叶斯由类ComplementNB完成，它包含的参数和多项式贝叶斯也非常相似：

class `sklearn.naive_bayes.ComplementNB`(*alpha=1.0*, *fit_prior=True*, *class_prior=None*, *norm=False*)

补集朴素贝叶斯
alpha : 浮点数, 可不填 (默认为1.0) 拉普拉斯或利德斯通平滑的参数α，如果设置为0则表示完全没有平滑选项。但是需要注意的是，平滑相当于人为给概率加上一些噪音，因此α设置得越大，多项式朴素贝叶斯的精确性会越低（虽然影响不是非常大），布里尔分数也会逐渐升高。
norm : 布尔值，可不填，默认False 在计算权重的时候是否适用L2范式来规范权重的大小。默认不进行规范，即不跟从补集朴素贝叶斯算法的全部内容，如果希望进行规范，请设置为True。
fit_prior : 布尔值, 可不填 (默认为True) 是否学习先验概率$P(Y = c)$。如果设置为false，则不使用先验概率，而使用统一先验概率（uniform prior），即认为每个标签类出现的概率是$\frac{1}{n_classes}$。
class_prior：形似数组的结构，结构为(n_classes,)，可不填（默认为None) 类的先验概率$P(Y = c)$。如果没有给出具体的先验概率则自动根据数据来进行计算。

那来看看，补集朴素贝叶斯在不平衡样本上的表现吧，同时我们来计算一下每种贝叶斯的计算速度：

```
from sklearn.naive_bayes import ComplementNB
from time import time
import datetime
```

```
name = ["Multinomial","Gaussian","Bernoulli","Complement"]
models = [MultinomialNB(),GaussianNB(),BernoulliNB(),ComplementNB()]


for name,clf in zip(name,models):
    times = time()
    Xtrain, Xtest, Ytrain, Ytest = train_test_split(X,y
                                                    ,test_size=0.3
                                                    ,random_state=420)
    #预处理
    if name != "Gaussian":
        kbs = KBinsDiscretizer(n_bins=10, encode='onehot').fit(Xtrain)
        Xtrain = kbs.transform(Xtrain)
        Xtest = kbs.transform(Xtest)

    clf.fit(Xtrain,Ytrain)
    y_pred = clf.predict(Xtest)
    proba = clf.predict_proba(Xtest)[:,1]
    score = clf.score(Xtest,Ytest)
    print(name)
    print("\tBrier:{:.3f}".format(BS(Ytest,proba,pos_label=1)))
    print("\tAccuracy:{:.3f}".format(score))
    print("\tRecall:{:.3f}".format(recall_score(Ytest,y_pred)))
    print("\tAUC:{:.3f}".format(AUC(Ytest,proba)))
    print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))
```

可以发现，补集朴素贝叶斯牺牲了部分整体的精确度和布里尔指数，但是得到了十分高的召回率Recall，捕捉出了98.7%的少数类，并且在此基础上维持了和原本的多项式朴素贝叶斯一致的AUC分数。和其他的贝叶斯算法比起来，我们的补集朴素贝叶斯的运行速度也十分优秀。如果我们的目标是捕捉少数类，那我们毫无疑问会希望选择补集朴素贝叶斯作为我们的算法。

3 案例：贝叶斯分类器做文本分类

文本分类是现代机器学习应用中的一大模块，更是自然语言处理的基础之一。我们可以通过将文字数据处理成数字数据，然后使用贝叶斯来帮助我们判断一段话，或者一篇文章中的主题分类，感情倾向，甚至文章体裁。现在，绝大多数社交媒体数据的自动化采集，都是依靠首先将文本编码成数字，然后按分类结果采集需要的信息。虽然现在自然语言处理领域大部分由深度学习所控制，贝叶斯分类器依然是文本分类中的一颗明珠。现在，我们就来学习一下，贝叶斯分类器是怎样实现文本分类的。

3.1 文本编码技术简介

3.1.1 单词计数向量

在开始分类之前，我们必须先将文本编码成数字。一种常用的方法是单词计数向量。在这种技术中，一个样本可以包含一段话或一篇文章，这个样本中如果出现了10个单词，就会有10个特征(n=10)，每个特征X_i代表一个单词，特征的取值x_i表示这个单词在这个样本中总共出现了几次，**是一个离散的，代表次数的，正整数。**

在sklearn当中，单词计数向量计数可以通过feature_extraction.text模块中的CountVectorizer类实现，来看一个简单的例子：

```
sample = ["Machine learning is fascinating, it is wonderful"
          ,"Machine learning is a sensational techonology"
          ,"Elsa is a popular character"]

from sklearn.feature_extraction.text import CountVectorizer

vec = CountVectorizer()

X = vec.fit_transform(sample)

X

#使用接口get_feature_names()调用每个列的名称

import pandas as pd
#注意稀疏矩阵是无法输入pandas的
CVresult = pd.DataFrame(X.toarray(),columns = vec.get_feature_names())

CVresult
```

从这个编码结果，我们可以发现两个问题。

首先，来回忆一下我们多项式朴素贝叶斯的计算公式：

$$\theta_{ci} = \frac{\sum_{y_j=c} x_{ji} + \alpha}{\sum_{i=1}^{n} \sum_{y_j=c} x_{ji} + \alpha n}$$

如果我们将每一列加和，除以整个特征矩阵的和，就是每一列对应的概率θ_i。由于是将x_{ji}进行加和，对于一个在很多个特征下都有值的样本来说，这个样本在对θ_{ci}的贡献就会比其他的样本更大。对于句子特别长的样本而言，这个样本对θ_i的影响是巨大的。因此补集朴素贝叶斯让每个特征的权重除以自己的L2范式，就是为了避免这种情况发生。

第二个问题，观察我们的矩阵，会发现"is"这个单词出现了四次，那经过计算，这个单词出现的概率就会最大，但其实它对我们的语义并没有什么影响（除非我们希望判断的是，文章描述的是过去的事件还是现在发生的事件）。可以遇见，如果使用单词计数向量，可能会导致一部分常用词（比如中文中的"的"）频繁出现在我们的矩阵中并且占有很高的权重，对分类来说，这明显是对算法的一种误导。为了解决这个问题，比起使用次数，我们使用单词在句子中所占的比例来编码我们的单词，这就是我们著名的TF-IDF方法。

3.1.2 TF-IDF

TF-IDF全称term frequency-inverse document frequency，词频逆文档频率，是通过单词在文档中出现的频率来衡量其权重，也就是说，IDF的大小与一个词的常见程度成反比，这个词越常见，编码后为它设置的权重会倾向于越小，以此来压制频繁出现的一些无意义的词。在sklearn当中，我们使用feature_extraction.text中类TfidfVectorizer来执行这种编码。

```
from sklearn.feature_extraction.text import TfidfVectorizer as TFIDF
vec = TFIDF()

X = vec.fit_transform(sample)

X
```

```
#同样使用接口get_feature_names()调用每个列的名称
TFIDFresult = pd.DataFrame(X.toarray(),columns=vec.get_feature_names())

TFIDFresult

#使用TF-IDF编码之后，出现得多的单词的权重被降低了么？

CVresult.sum(axis=0)/CVresult.sum(axis=0).sum()

TFIDFresult.sum(axis=0) / TFIDFresult.sum(axis=0).sum()
```

在之后的例子中，我们都会使用TF-IDF的编码方式。

3.2 探索文本数据

在现实中，文本数据的处理是十分耗时耗力的，尤其是不规则的长文本的处理方式，绝对不是一两句话能够说明白的，因此在这里我们将使用的数据集是sklearn中自带的文本数据集fetch_20newsgroup。这个数据集是20个网络新闻组的语料库，其中包含约2万篇新闻，全部以英文显示，如果大家希望使用中文则处理过程会更加困难，会需要自己加载中文的语料库。在这个例子中，主要目的是为大家展示贝叶斯的用法和效果，因此我们就使用英文的语料库。

```
from sklearn.datasets import fetch_20newsgroups

#初次使用这个数据集的时候，会在实例化的时候开始下载
data = fetch_20newsgroups()

#通常我们使用data来查看data里面到底包含了什么内容，但由于fetch_20newsgourps这个类加载出的数据巨大，数据结构中混杂很多文字，因此很难去看清

#不同类型的新闻
data.target_names

#其实fetch_20newsgroups也是一个类，既然是类，应该就有可以调用的参数
#面对简单数据集，我们往往在实例化的过程中什么都不写，但是现在data中数据量太多，不方便探索
#因此我们需要来看看我们的类fetch_20newsgroups都有什么样的参数可以帮助我们
```

`sklearn.datasets.fetch_20newsgroups`(*data_home=None*, *subset='train'*, *categories=None*, *shuffle=True*, *random_state=42*, *remove=()*, *download_if_missing=True*)

在这之中，我们来认识几个比较重要的参数：

fetch_20newsgroups 参数列表
subset : 选择类中包含的数据子集 输入"train"表示选择训练集，“test"表示输入测试集，”all"表示加载所有的数据
categories : 可输入None或者数据所在的目录 选择一个子集下，不同类型或不同内容的数据所在的目录。如果不输入默认None，则会加载全部的目录。
download_if_missing：可选，默认是True 如果发现本地数据不全，是否自动进行下载
shuffle : 布尔值，可不填，表示是否打乱样本顺序 对于假设样本之间互相独立并且服从相同分布的算法或模型（比如随机梯度下降）来说可能很重要。

现在我们就可以直接通过参数来提取我们希望得到的数据集了。

```
import numpy as np
import pandas as pd

categories = ["sci.space" #科学技术 - 太空
              ,"rec.sport.hockey" #运动 - 曲棍球
              ,"talk.politics.guns" #政治 - 枪支问题
              ,"talk.politics.mideast"] #政治 - 中东问题

train = fetch_20newsgroups(subset="train",categories = categories)
test = fetch_20newsgroups(subset="test",categories = categories)

train
#可以观察到，里面依然是类字典结构，我们可以通过使用键的方式来提取内容

train.target_names

#查看总共有多少篇文章存在
len(train.data)

#随意提取一篇文章来看看
train.data[0]

#查看一下我们的标签
np.unique(train.target)

len(train.target)

#是否存在样本不平衡问题?
for i in [1,2,3]:
    print(i,(train.target == i).sum()/len(train.target))
```

3.3 使用TF-IDF将文本数据编码

```
from sklearn.feature_extraction.text import TfidfVectorizer as TFIDF
```

```
Xtrain = train.data
Xtest = test.data
Ytrain = train.target
Ytest = test.target

tfidf = TFIDF().fit(Xtrain)
Xtrain_ = tfidf.transform(Xtrain)
Xtest_ = tfidf.transform(Xtest)

Xtrain_

tosee = pd.DataFrame(Xtrain_.toarray(),columns=tfidf.get_feature_names())

tosee.head()

tosee.shape
```

3.4 在贝叶斯上分别建模，查看结果

```
from sklearn.naive_bayes import MultinomialNB, ComplementNB, BernoulliNB
from sklearn.metrics import brier_score_loss as BS

name = ["Multinomial","Complement","Bournulli"]
#注意高斯朴素贝叶斯不接受稀疏矩阵
models = [MultinomialNB(),ComplementNB(),BernoulliNB()]

for name,clf in zip(name,models):
    clf.fit(Xtrain_,Ytrain)
    y_pred = clf.predict(Xtest_)
    proba = clf.predict_proba(Xtest_)
    score = clf.score(Xtest_,Ytest)
    print(name)

    #4个不同的标签取值下的布里尔分数
    Bscore = []
    for i in range(len(np.unique(Ytrain))):
        bs = BS(Ytest,proba[:,i],pos_label=i)
        Bscore.append(bs)
        print("\tBrier under {}:{:.3f}".format(train.target_names[i],bs))

    print("\tAverage Brier:{:.3f}".format(np.mean(Bscore)))
    print("\tAccuracy:{:.3f}".format(score))
    print("\n")
```

从结果上来看，两种贝叶斯的效果都很不错。虽然补集贝叶斯的布里尔分数更高，但它的精确度更高。我们可以使用概率校准来试试看能否让模型进一步突破：

```
from sklearn.calibration import CalibratedClassifierCV
```

```
name = ["Multinomial"
        ,"Multinomial + Isotonic"
        ,"Multinomial + Sigmoid"
        ,"Complement"
        ,"Complement + Isotonic"
        ,"Complement + Sigmoid"
        ,"Bernoulli"
        ,"Bernoulli + Isotonic"
        ,"Bernoulli + Sigmoid"]

models = [MultinomialNB()
          ,CalibratedClassifierCV(MultinomialNB(), cv=2, method='isotonic')
          ,CalibratedClassifierCV(MultinomialNB(), cv=2, method='sigmoid')
          ,ComplementNB()
          ,CalibratedClassifierCV(ComplementNB(), cv=2, method='isotonic')
          ,CalibratedClassifierCV(ComplementNB(), cv=2, method='sigmoid')
          ,BernoulliNB()
          ,CalibratedClassifierCV(BernoulliNB(), cv=2, method='isotonic')
          ,CalibratedClassifierCV(BernoulliNB(), cv=2, method='sigmoid')
         ]

for name,clf in zip(name,models):
    clf.fit(Xtrain_,Ytrain)
    y_pred = clf.predict(Xtest_)
    proba = clf.predict_proba(Xtest_)
    score = clf.score(Xtest_,Ytest)
    print(name)
    Bscore = []
    for i in range(len(np.unique(Ytrain))):
        bs = BS(Ytest,proba[:,i],pos_label=i)
        Bscore.append(bs)
        print("\tBrier under {}:{:.3f}".format(train.target_names[i],bs))
    print("\tAverage Brier:{:.3f}".format(np.mean(Bscore)))
    print("\tAccuracy:{:.3f}".format(score))
    print("\n")
```

可以观察到，多项式分布下无论如何调整，算法的效果都不如补集朴素贝叶斯来得好。因此我们在分类的时候，应该选择补集朴素贝叶斯。对于补集朴素贝叶斯来说，使用Sigmoid进行概率校准的模型综合最优秀：准确率最高，对数损失和布里尔分数都在0.1以下，可以说是非常理想的模型了。

对于机器学习而言，朴素贝叶斯也许不是最常用的分类算法，但作为概率预测算法中唯一一个真正依赖概率来进行计算，并且简单快捷的算法，朴素贝叶斯还是常常被人们提起。并且，朴素贝叶斯在文本分类上的效果的确非常优秀。由此可见，只要我们能够提供足够的数据，合理利用高维数据进行训练，朴素贝叶斯就可以为我们提供意想不到的效果。