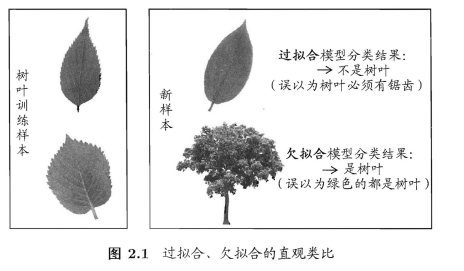
# 4 模型评估与选择

## 4.1经验误差与过拟合

通常我们把分类错误的样本数占样本总数的比例称为“**错误率**”（errorrate），即如果在m个样本中有a个样本分类错误则**错误率E=a/m**；相应的，**1-a/m称为“精度”**(accuracy)，即“精度=1-错误率”，更一般地，我们把学习器的实际**预测输出与样本的真实输出之间的差异称为“误差”**（error），学习器在训练集上的误差称为“**训练误差**”(trainingerror)或“**经验误差**”(empiricalerror)，在新样本上的误称为“**泛化误差**”(generalizationerror)。显然，我们希望得到泛化误差小的学习器。然而，我们事先并不知道新样本是什么样，实际能做的是努力使经验误差最小化。在很多情况下，我们可以学得一个经验误差很小、在训练集上表现很好的学习器，例如甚至对所有**训练样本都分类正确，即分类错误率为零，分类精度为100%，但这是不是我们想要的学习器呢**？遗憾的是，这样的学习器在多数情况下都不好。

我们实际希望的，是在新样本上能表现得很好的学习器。为了达到这个目的，应该**从训练样本中尽可能学出适用于所有潜在样本的“普遍规律”，这样才能在遇到新样本时做出正确的判别然而当学习器把训练样本学得“太好”了的时候很可能已经把训练样本自身的一些特点当作了所有潜在样本都会具有的一般性质**，这样就会导致泛化性能下降，这种现象在机器学习中称为“**过拟合**”(overfitting)与“过拟合”相对的是“**欠拟合**”(underftting)，这是指对训练样本的一般性质尚未学好。图2.1给出了关于过拟合与欠拟合的一个便于直观理解的类比。

有多种因素可能导致过拟合，其中最常见的情况是由于**学习能力过于强大以至于把训练样本所包含的不太一般的特性都学到了**，而欠拟合则通常是由于学习能力低下而造成的。欠拟合比较容易克服，例如在决策树学习中扩展分支、在神经网络学习中增加训练轮数等，而过拟合则很麻烦。在后面的学习中我们将看到，过拟合是机器学习面临的关键障碍，各类学习算法都必然带有一些针对过拟合的措施；然而必须认识到，**过拟合是无法彻底避免的**，我们所能做的只是“**缓解**”，或者说减小其风险。关于这一点，可大致这样理解：机器学习面临的问题通常是NP难甚至更难而有效的学习算法必然是在多项式时间内运行完成，若可彻底避免过拟合，则通过经验误差最小化就能获最优解，这就意味着我们构造性地证明了



“P=NP”；因此只要相信“PNP”，过拟合就不可避免。

在计算机科学中，P 和 NP 是两种复杂性类别，涉及解决问题的难度和效率。P 类问题指的是能够在多项式时间内解决的问题，也就是说，存在一个有效算法可以在合理的时间内找到问题的解。而 NP 类问题则是指可以在多项式时间内验证解的问题，但尚未找到有效算法能够在多项式时间内解决这类问题。

“P=NP” 问题的陈述是在询问是否所有可以在多项式时间内验证解的问题都可以在多项式时间内解决。换句话说，这个问题探讨了是否存在一种方法，让NP类问题的解可以在与问题规模成多项式关系的时间内找到。

如果“P=NP” 成立，那么意味着NP类问题的解法与验证解一样容易，这将会对密码学、优化问题等领域产生深远影响。然而，迄今为止，尚未找到证据证明“P=NP”，也未能证明“P≠NP”。这个问题依然是计算机科学中的一个重大谜题

在现实任务中，我们往往有多种学习算法可供选择，甚至对同一个学习算法，当使用不同的参数配置时，也会产生不同的模型，那么，我们该选用哪一个学习算法、使用哪一种参数配置呢？这就是机器学习中的“**模型选择**”(modelselection)问题理想的解决方案当然是对候选模型的泛化误差进行评估，然后**选择泛化误差最小的那个模型**，然而如上面所讨论的，我们无法直接获得泛化误差，而训练误差又由于过拟合现象的存在而不适合作为标准，那么，在现实中如何进行模型评估与选择呢？

Python实例代码：

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  from sklearn.pipeline import Pipeline  from sklearn.preprocessing import PolynomialFeatures  from sklearn.linear\_model import LinearRegression  from sklearn.model\_selection import train\_test\_split  from sklearn.metrics import mean\_squared\_error  # 生成模拟数据集  np.random.seed(0)  X = np.linspace(-3, 3, 100)  y = 1 + 2\*X - 0.5\*X\*\*2 + np.random.normal(0, 1, 100)  # 将数据集划分为训练集和测试集  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  # 创建多项式回归模型  degrees = [1, 4, 15] # 不同多项式的次数  plt.figure(figsize=(12, 6))  for i in range(len(degrees)):  ax = plt.subplot(1, len(degrees), i + 1)  plt.setp(ax, xticks=(), yticks=())    polynomial\_features = PolynomialFeatures(degree=degrees[i], include\_bias=False)  linear\_regression = LinearRegression()  pipeline = Pipeline([("polynomial\_features", polynomial\_features), ("linear\_regression", linear\_regression)])  pipeline.fit(X\_train[:, np.newaxis], y\_train)    # 在训练集上拟合模型  y\_train\_predicted = pipeline.predict(X\_train[:, np.newaxis])    # 在测试集上评估模型  y\_test\_predicted = pipeline.predict(X\_test[:, np.newaxis])    # 绘制拟合曲线  plt.scatter(X\_train, y\_train, label="Training points")  plt.scatter(X\_test, y\_test, label="Test points")  plt.plot(X, pipeline.predict(X[:, np.newaxis]), label="Model")  plt.xlabel("x")  plt.ylabel("y")  plt.title(f"Degree {degrees[i]}\nTrain MSE: {mean\_squared\_error(y\_train, y\_train\_predicted):.2f}, Test MSE: {mean\_squared\_error(y\_test, y\_test\_predicted):.2f}")  plt.legend(loc="best")  plt.tight\_layout()  plt.show() |

这个示例使用了多项式回归模型，并分别尝试了三种不同次数的多项式拟合。在训练集和测试集上计算了均方误差（MSE），并绘制了不同次数多项式的拟合曲线。通常，当模型的复杂度过高时（例如，多项式次数过高），模型会过度拟合训练集的数据，导致在测试集上表现不佳。

Python示例代码（欠拟合）：

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  from sklearn.pipeline import Pipeline  from sklearn.linear\_model import LinearRegression  from sklearn.model\_selection import train\_test\_split  from sklearn.metrics import mean\_squared\_error  # 生成模拟数据集  np.random.seed(0)  X = np.linspace(-3, 3, 100)  y = 2\*X + np.random.normal(0, 1, 100)  # 将数据集划分为训练集和测试集  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  # 创建线性模型  linear\_model = LinearRegression()  # 在训练集上拟合模型  linear\_model.fit(X\_train[:, np.newaxis], y\_train)  # 在训练集和测试集上评估模型  y\_train\_predicted = linear\_model.predict(X\_train[:, np.newaxis])  y\_test\_predicted = linear\_model.predict(X\_test[:, np.newaxis])  # 计算均方误差（MSE）  train\_mse = mean\_squared\_error(y\_train, y\_train\_predicted)  test\_mse = mean\_squared\_error(y\_test, y\_test\_predicted)  # 绘制拟合曲线和数据散点图  plt.figure(figsize=(8, 6))  plt.scatter(X\_train, y\_train, label="Training data")  plt.scatter(X\_test, y\_test, label="Test data")  plt.plot(X, linear\_model.predict(X[:, np.newaxis]), color='red', label="Model")  plt.xlabel("x")  plt.ylabel("y")  plt.title(f"Linear Model\nTrain MSE: {train\_mse:.2f}, Test MSE: {test\_mse:.2f}")  plt.legend()  plt.show() |

## 4.2评估方法

通常，我们可通过实验测试来对学习器的泛化误差进行评估并进而做出选择。为此，需使用一个“**测试集”(testingset)来测试学习器对新样本的判别能力**，然后以测试集上的“**测试误差**”(testingerror)作为**泛化误差的近似通常我们假设测试样本也是从样本真实分布中独立同分布采样而得**。但需注意的是，测试集应该尽可能与训练集**互斥**，即测试样本尽量不在训练集中出现、未在训练过程中使用过。

测试样本为什么要尽可能不出现在训练集中呢？为理解这一点，不妨考虑这样一个场景:老师出了10道习题供同学们练习，考试时老师又用同样的这10道题作为试题，这个考试成绩能否有效反映出同学们学得好不好呢？答案是否定的，可能有的同学只会做这10道题却能得高分回到我们的问题上来，我们希望得到**泛化性能强**的模型，好比是希望同学们对课程学得很好、获得了对所学知识“**举一反三**”的能力:**训练样本相当于给同学们练习的习题，测试过程则相当于考试显然若测试样本被用作训练了，则得到的将是过于“乐观”的估计结果。**

可是我们只有一个包含m个样例的数据集D={(x1，y1)，(x2，y2)，…，(xm，ym)}既要训练，又要测试，怎样才能做到呢？答案是:通过对D进行适当的处理，从中产生出训练集S和测试集下面介绍几种常见的做法

## 4.2.1留出法

“**留出法**”(hold-out)直接将数据集D划分为两个互斥的集合其中一个集合作为训练集S，另一个作为测试集T，即D=。在S上训练出模型后，用T来评估其测试误差，作为对泛化误差的估计。

以二分类任务为例，假定D包含1000个样本将其划分为**S包含700个样本**，**T包含300个样本**用S进行训练后如果模型在T上有90个样本分类错误，那么其错误率为**(90/300)x100%=30%**相应的**精度为1-30%=70%**。

需注意的是，训练/测试集的划分要尽可能保持数据分布的一致性，避免因数据划分过程引入额外的偏差而对最终结果产生影响，例如在分类任务中至少要保持样本的类别比例相似如果从采样(sampling)的角度来看待数据集的划分过程则**保留类别比例的采样方式通常称为“分层采样”**(stratifiedsampling)例如通过对D进行分层采样而获得含70%样本的训练集S和含30%样本的测试集T若D包含500个正例500个反**例则分层采样得到的S应包含350个正例、350个反例，而则包含150个正例和150个反例**；若S、T中样本类别比例差别很大则误差估计将由于训练/测试数据分布的差异而产生偏差。

另一个需注意的问题是即便在给定训练/测试集的样本比例后，仍存在多种划分方式对初始数据集D进行分割例如在上面的例子中可以把D中的样本排序，然后把**前350个正例放到训练集中也可以把最后350个正例放到训练集**中，······这些不同的划分将导致**不同的训练/测试集**，相应的，模型评估的结果也会有差别因此单次使用留出法得到的估计结果往往不够稳定可靠在使用留出法时，一般要采用**若干次随机划分、重复进行实验评估后取平均值作为留出法的评估结果**。例如进行**100次随机划分每次产生一个训练/测试集用于实验评估**，100次后就得到100个结果而留出法返回的则是这100个结果的平均。

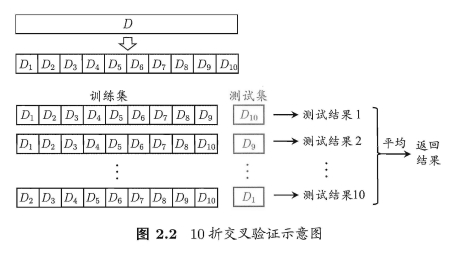
此外，我们希望评估的是用D训练出的模型的性能，但留出法需划分训练/测试集，这就会导致一个窘境:**若令训练集S包含绝大多数样本，则训练出的模型可能更接近于用D训练出的模型但由于T比较小评估结果可能不够稳定准确**:若令测试集T多包含一些样本则训练集S与D差别更大了被评估的模型与用D训练出的模型相比可能有较大差别从而降低了评估结果的保真性(fidelity)。这个问题没有完美的解决方案，常见做法是将大约**2/3~4/5的样本**用于训练剩余样本用于测试。

Python实例代码：

|  |
| --- |
| from sklearn.model\_selection import train\_test\_split  import numpy as np  # 假设有一个特征集和对应的标签  features = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])  labels = np.array([0, 1, 0, 1, 1])  # 定义测试集的比例，例如这里设为 0.2，即 20% 的数据作为测试集  test\_size = 0.2  # 划分数据集为训练集和测试集  X\_train, X\_test, y\_train, y\_test = train\_test\_split(features, labels, test\_size=test\_size, random\_state=42)  # 输出训练集和测试集的大小  print("训练集特征集大小:", X\_train.shape)  print("训练集标签大小:", y\_train.shape)  print("测试集特征集大小:", X\_test.shape)  print("测试集标签大小:", y\_test.shape) |

## 4.2.2交叉验证法

“交叉验证法”(crossvalidation)先将**数据集D划分为k个大小**相似的互**斥子集**，即，每个子集Di都尽可能保持数据分布的一致性，即从D中通过分层采样得到。然后，每次用k-1个子集的并集作为训练集，余下的那个子集作为测集；这样就可获得**k组训练/测试集**，从而可进行**k次训练和测试**，最终返回的是这个**测试结果的均值**。显然，交叉验证法评估结果的稳定性和保真性在很大程度上取决于的取值，为强调这一点，通常把交叉验证法称为“**折交叉验证**”(k-foldcrossvalidation)k最常用的取值是10此时称为10折交叉验证其他常用的值有5、20等图2.2给出了10折交叉验证的示意图。



与留出法相似，将**数据集D划分为个子集同样存在多种划分方式为减小因样本划分不同而引入的差别**，k折交叉验证通常要随机使用不同的划分重复p次，最终的评估结果是这p次折交验证结果的均值例如常见的有“10次10折交验证”。

Python示例代码：

|  |
| --- |
| from sklearn.model\_selection import KFold  import numpy as np  # 假设有一个特征集和对应的标签  features = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])  labels = np.array([0, 1, 0, 1, 1])  # 定义 k 值（这里设定为 3 折交叉验证）  k = 3  # 初始化 k 折交叉验证对象  kf = KFold(n\_splits=k)  # 执行 k 折交叉验证  fold = 1  for train\_index, test\_index in kf.split(features):  print(f"Fold {fold}:")  print("Train Index:", train\_index)  print("Test Index:", test\_index)    # 使用索引从特征集和标签中获取训练集和测试集  X\_train, X\_test = features[train\_index], features[test\_index]  y\_train, y\_test = labels[train\_index], labels[test\_index]    # 在这里可以添加模型训练和评估的代码  # 例如，使用 X\_train 和 y\_train 进行模型训练，使用 X\_test 进行模型测试  fold += 1 |

假定数据集D中包含m个样本，若令k=m则得到了交叉验证法的一个特例:**留一法(Leave-One-Out，简称LOO)显然留一法不受随机样本划分方式的影响，因为m个样本只有唯一的方式划分为m个一每个子集包含个样本**；留一法使用的**训练集与初始数据集**相比只少了一个样本，这就使得在绝大多数情况下，留一法中被**实际评估的模型与期望评估的用D训练出的模型很相似**.因此留一法的评估结果往往被认为比较准确然而，留一法也有其缺陷:**在数据集比较大时训练个模型的计算开销可能是难以忍受**的(例如数据集包含1百万个样本则需训练1百万个模型)而这还是在未考虑算法调参的情况下另外留一法的估计结果也未必永远比其他评估方法准确；“没有免费的午餐”定理对实验评估方法同样适用。

## 4.2.3自助法

我们希望评估的是用D训练出的模型但在留出法和交叉验证法中，由于保留了一部分样本用于测试，**因此实际评估的模型所使用的训练集比D小这必然会引入一些因训练样本规模不同而导致的估计偏差，**留一法受训练样本规模变化的影响较小，但计算复杂度又太高了，**有没有什么办法可以减少训练样本规模不同造成的影响，同时还能比较高效地进行实验估计呢**？

“自助法”(bootstrapping)是一个比较好的解决方案它直接以自助采样法(bootstrapsampling)为基础[EfronandTibshirani，1993给定包含m个样本的数据集D我们对它进行采样产生数据集**D、**：每次随机从D中挑选一个样本，将其拷贝放入**D、**，然后再将该样本放回初始数据集D中，使得该样本在下次采样时仍有可能被采到；这个过程重复执行m次后我们就得到了包含m个样本的数据集D、，这就是自助采样的结果。

假设你有一个箱子，里面装了一些彩球，每个彩球代表数据集中的一个样本。现在，你想要创建一个和原始数据集一样大的新数据集，但是你希望这个新数据集是通过抽样得到的，而且抽样时每个样本都有可能被选中，即被抽中后会放回原始数据集，可能再次被选中。

首先，你从箱子中随机抽取一个彩球（一个样本），记录下它的数据，并把这个彩球放回箱子里。这个动作代表了从原始数据集中抽取一个样本，并将它放入新的数据集。

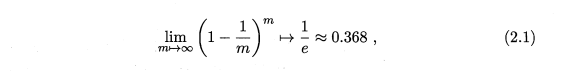
然后，你再次从箱子中随机抽取一个彩球，记录它的数据，并把这个彩球放回。这个过程不断重复，直到你重复了和原始数据集一样多次的抽样次数，也就是m次。

最后，你就得到了一个新的数据集D'，这个数据集和原始数据集D一样大，但是它是通过自助法得到的，即通过从原始数据集中有放回地抽取样本得到的。

Python示例代码：

|  |
| --- |
| import random  # 原始数据集  original\_data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] # 这里假设原始数据集是一个简单的列表  # 执行自助法抽样的次数  num\_samples = 10 # 例如，我们想要抽样10次  bootstrapped\_data = [] # 用于存储自助法得到的样本集合  for \_ in range(num\_samples):  # 随机选择一个样本放回原始数据集中  bootstrap\_sample = random.choice(original\_data)  bootstrapped\_data.append(bootstrap\_sample)  print("原始数据集:", original\_data)  print("自助法抽样得到的数据集:", bootstrapped\_data) |

显然D中有一部分样本会在D中多次出现，而另一部分样本不出现，可以做一个简单的估计，样本在m次采样中始终不被采到的概率是(1-1/m)m,取极限得到



即通过自助采样，初始数据集D中约有36.8%的出现在采样数据集**D、**中，于是我们可将**D、**用作训练集，D/**D、**用作测试集；这样实际评估的模型与期望评估的模型都使用m个练本而我们仍有数据总量约1/3的、没在训练集中出现的样本用于测试这样的测试结果亦称“包外估计”(out-ofbagestimate)。

自助法在数据集较小、难以有效划分训练/测试集时很有用；此外，自助法能从初始数据集中产生多个不同的训练集，这对集成学习等方法有很大的好处然而，自助法产生的数据集改变了初始数据集的分布，这会引入估计偏差，因此，在初始数据量足够时，留出法和交叉验证法更常用一些。

## 4.2.4调参与最终模型

大多数学习算法都有些参数(parameter)需要设定，参数配置不同，学得模型的性能往往有显著差别。因此，在进行模型评估与选择时，除了要对适用学习算法进行选择还需对算法参数进行设定，这就是通常所说的“参数调节”或简称“调参”(parametertuning)。

读者可能马上想到，调参和算法选择没什么本质区别:对每种参数配置都训练出模型然后把对应最好模型的参数作为结果这样的考虑基本是正确的但有一点需注意:学习算法的很多参数是在实数范围内取值，因此对每种参数配置都训练出模型来是不可行的现实中常用的做法，是对每个参数选定一个范围和变化步长，例如在0，0.2范围内以0.05为步长则实际要评估的候选参数值有5个最终是从这5个候选值中产生选定值显然这样选定的参数值往往不是“最佳”值，但这是在计算开销和性能估计之间进行折中的结果，通过这个折中，学习过程才变得可行，事实上，即便在进行这样的折中后，调参往往仍很困难可以简单估算一下:假定算法有3个参数每个参数仅考虑5个候选值这样对每一组训练/测试集就有53=125个模型需考察很多强大的学习算法有大量参数需设定，这将导致极大的调参工程量，以至于在不少应用任务中参数调得好不好往往对最终模型性能有关键性影响。

给定包含m个样本的数据集D，在模型评估与选择过程中由于需要留出部分数据进行评估测试，事实上我们只使用了一部分数据训练模型。因此，在模型选择完成后，学习算法和参数配置已选定，此时应该用数据集D重新训练模型这个模型在训练过程中使用了所有m个样本这才是我们最终提交给用户的模型。

另外，需注意的是，我们通常把学得模型在实际使用中遇到的数据称为测试数据，为了加以区分，模型评估与选择中用于评估测试的数据集常称为“验证集”(validationset)例如在研究对比不同算法的泛化性能时我们用测试集上的判别效果来估计模型在实际使用时的泛化能力，而把训练数据另外划分为训练集和验证集，基于验证集上的性能来进行模型选择和调参。

Python示例代码：

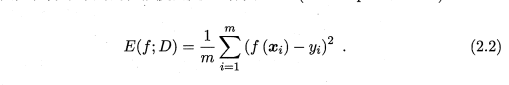
|  |
| --- |
| from sklearn.preprocessing import PolynomialFeatures  **from sklearn.metrics import mean\_squared\_error**  **# 不同多项式的次数**  **degrees = [1, 2, 3, 4, 5, 10]**  **# 初始化均方误差列表**  **train\_mse\_list = []**  **test\_mse\_list = []**  **plt.figure(figsize=(12, 6))**  **for i, degree in enumerate(degrees):**  **polynomial\_features = PolynomialFeatures(degree=degree, include\_bias=False)**  **linear\_regression = LinearRegression()**  **pipeline = Pipeline([("polynomial\_features", polynomial\_features), ("linear\_regression", linear\_regression)])**  **pipeline.fit(X\_train[:, np.newaxis], y\_train)**    **# 在训练集和测试集上评估模型**  **y\_train\_predicted = pipeline.predict(X\_train[:, np.newaxis])**  **y\_test\_predicted = pipeline.predict(X\_test[:, np.newaxis])**    **# 计算均方误差**  **train\_mse = mean\_squared\_error(y\_train, y\_train\_predicted)**  **test\_mse = mean\_squared\_error(y\_test, y\_test\_predicted)**    **train\_mse\_list.append(train\_mse)**  **test\_mse\_list.append(test\_mse)**    **# 绘制拟合曲线**  **plt.subplot(2, 3, i+1)**  **plt.scatter(X\_train, y\_train, label="Training data")**  **plt.scatter(X\_test, y\_test, label="Test data")**  **plt.plot(X, pipeline.predict(X[:, np.newaxis]), color='red', label="Model")**  **plt.title(f"Degree {degree}\nTrain MSE: {train\_mse:.2f}, Test MSE: {test\_mse:.2f}")**  **plt.xlabel("x")**  **plt.ylabel("y")**  **plt.legend()**  **plt.tight\_layout()**  **plt.show()**  **# 输出不同多项式次数下的均方误差**  **for degree, train\_mse, test\_mse in zip(degrees, train\_mse\_list, test\_mse\_list):**  **print(f"Degree {degree}: Train MSE = {train\_mse:.2f}, Test MSE = {test\_mse:.2f}")** |

## 4.3性能度量

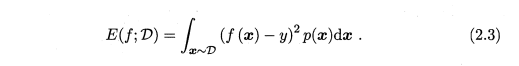
对学习器的泛化性能进行评估，不仅需要有效可行的实验估计方法，还需要有衡量模型泛化能力的评价标准这就是性能度量(performancemeasure)。性能度量反映了任务需求，在对比不同模型的能力时，使用不同的性能度量往往会导致不同的评判结果；这意味着模型的“好坏”是相对的，什么样的模型是好的。不仅取决于算法和数据，还决定于任务需求。

在预测任务中给定样例集，其中yi是示例x2的真实标记要评估学习器的性能就要把学习器预测结果f(x)与真实标记y进行比较。

回归任务最常用的性能度量是“均方误差”(meansquarederror)

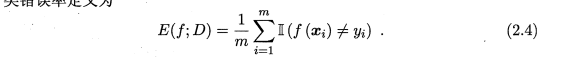


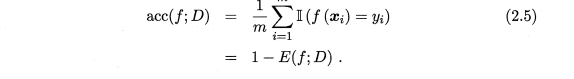
更一般的，对于数据分布D和概率密度函数p()，均方误差可描述为



## 4.3.1错误率与精度

本章开头提到了错误率和精度，这是分类任务中最常用的两种性能度量既适用于二分类任务，也适用于多分类任务。错误率是分类错误的样本数占样本总数的比例，精度则是分类正确的样本数占样本总数的比例对样例集D分类错误率定义为

精度则定义为

更一般的，对于数据分布D和概率密度函数p（）错率与精度可分别述为

2.3.2查准率查全率F1

错误率和精度虽常用，但并不能满足所有任务需求。以西瓜问题为例，假定瓜农拉来一车西瓜，我们用训练好的模型对这些西瓜进行判别，显然，错误率衡量了有多少比例的瓜被判别错误。但是若我们关心的是“挑出的西瓜中有多少比例是好瓜”，或者“所有好瓜中有多少比例被挑了出来”，那么错误率显然就不够用了，这时需要使用其他的性能度量。

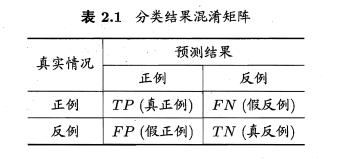
类似的需求在信息检索、Web搜索等应用中经常出现，例如在信息检索中，我们经常会关心“检索出的信息中有多少比例是用户感兴趣的”“用户感兴趣的信息中有多少被检索出来了”“查准率”(precision)与“查全率”(recall)是更为适用于此类需求的性能度量。

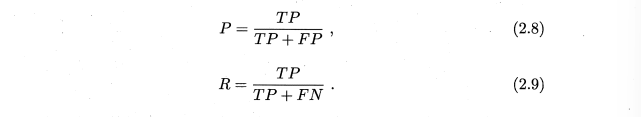
**例如：假设一个不太准确的验钞机，真钞的会被存起来，但有时候验钞机也会出现故障，把真钞拦住，假钞存起来，那么查准率和查全率的公式如下：**

**查准率=存起来的真钞/（存起来的真钞+存起来的假钞）**

**查全率=存起来的真钞/（存起来的真钞+误拦住的真钞）**

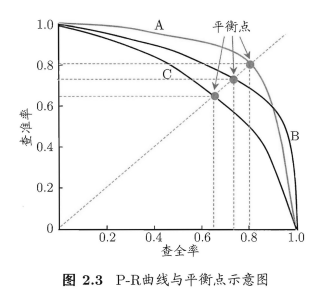
对于二分类问题，可将样例根据其真实类别与学习器预测类别的组合划分为真正例(truepositive)、假正例(falsepositive)、真反例(truenegative)、假反例(falsenegative)四种情形令TP、FP、TN、FN分别表示其对应的样例数，则显然有TP+FP+TN+FN=样例总数。分类结果的“混矩阵”(confusionmatrix)如表2.1所示



查准率P与查全率R分别定义为

查准率和查全率是一对矛盾的度量。一般来说，查准率高时，查全率往往偏低；而查全率高时查准率往往偏低例如若希望将好瓜尽可能多地选出来则可通过增加选瓜的数量来实现，如果将所有西瓜都选上，那么所有的好瓜也必然都被选上了，但这样查准率就会较低；若希望选出的瓜中好瓜比例尽可能高，则可只挑选最有把握的瓜，但这样就难免会漏掉不少好瓜，使得查全率较低。通常只有在一些简单任务中，才可能使查全率和查准率都很高。

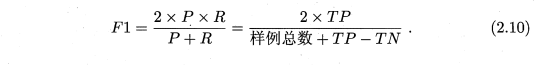
在很多情形下，我们可根据学习器的预测结果对样例进行排序，排在前面的是学习器认为“最可能”是正例的样本，排在最后的则是学习器认为“最不可能”是正例的样本，按此顺序逐个把样本作为正例进行预测，则每次可以计算出当前的查全率、查准率。以查准率为纵轴、查全率为横轴作图，就得到了查准率查全率曲线简称“PR曲线”，显示该曲线的图称为“P-R图”。图2.3给出了一个示意图。



P-R图直观地显示出学习器在样本总体上的查全率、查准率在进行比较时，若一个学习器的P-R曲线被另一个学习器的曲线完全“包住”，则可断言后者的性能优于前者，例如图2.3中学习器A的性能优于学习器C；如果两个学习器的P-R曲线发生了交叉，例如图2.3中的A与B则难以一般性地断言两者孰优孰劣，只能在具体的查准率或查全率条件下进行比较。然而，在很多情形下，人们往往仍希望把学习器A与B比出个高低这时一个比较合理的判据是比较P-R曲线下面积的大小，它在一定程度上表征了学习器在查准率和查全率上取得相对“双高”的比例，但这个值不太容易估算，因此人们设计了一些综合考虑查准率、查全率的性能度量。

“平衡点”(Break-EventPoint，简称BEP)就是这样一个度量，它是“查准率=查全率”时的取值例如图2.3中学习器C的BEP是0。4而基于BEP的比较，可认为学习器A优于B。

但BEP还是过于简化了些更常用的是F1度量：



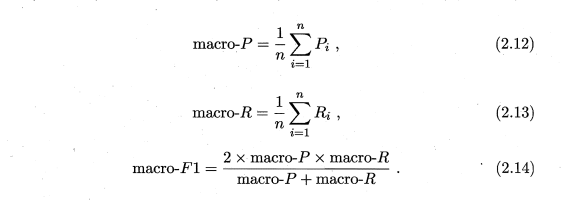
在一些应用中，对查准率和查全率的重视程度有所不同例如在商品推荐系统中，为了尽可能少打扰用户，更希望推荐内容确是用户感兴趣的，此时查准率更重要:而在逃犯信息检索系统中，更希望尽可能少漏掉逃犯，此时查全率更重要F1度量的一般形式——Fβ能让我们表达出对查准率/查全率的不同偏好，它定义为



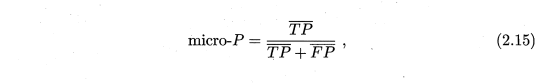
其中β>0度量了查全率对查准率的相对重要性[VanRijsbergen，1979]。β=1时退化为标准的F1>；β>1查全率会更有影响。

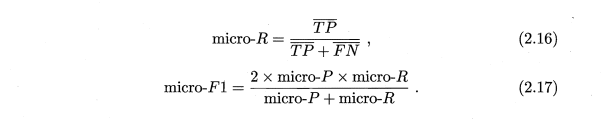
很多时候我们有多个二分类混淆矩阵，例如进行多次训练/测试，每次得到个混矩阵:或是在多个数据集上进行训练/测试希望估计算法的“全局”性能；甚或是执行多分类任务，每两两类别的组合都对应一个混淆矩阵；……总之，我们希望在n个二分类混淆矩阵上综合考察查准率和查全率。

一种直接的做法是先在各混淆矩阵上分别计算出查准率和查全率记为（p1，r2）（p2，r2），……(pn，rn)再计算平均值，这样就得到“宏查准率”(macro-P)、“宏查全率”(macro-R)，以及相应的“宏F1”(macro-F1):



还可先将各混淆矩阵的对应元素进行平均，得到TP、FP、TN、FN的平均值分别记为TP、FP、TN、FN再基于这些平均值计算出“微查准率”(micro-P)、“微查全率”(micro-R)和“微F1”(micro-F1):



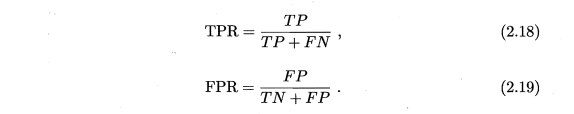


## 4.3.3 ROC与AUC

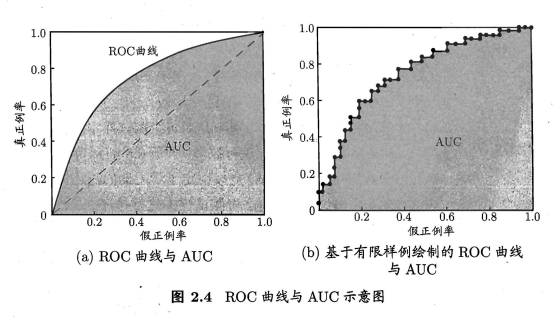
很多学习器是为测试样本产生一个实值或概率预测，然后将这个预测值与一个分类阈值(threshold)进行比较，若大于阙值则分为正类，否则为反类例如，神经网络在一般情形下是对每个测试样本预测出一个0.0，1之间的实值然后将这个值与0.5进行比较大于0.5则判为正例否则为反例这个实值或概率预测结果的好坏，直接决定了学习器的泛化能力。实际上，根据这个实值或概率预测结果，我们可将测试样本进行排序，“最可能”是正例的排在最前面“最不可能”是正例的排在最后面这样分类过程就相当于在这个排序中以某个“截断点”(cutpoint)将样本分为两部分前一部分判作正例，后一部分则判作反例。

在不同的应用任务中，我们可根据任务需求来采用不同的截断点，例如若我们更重视“查准率”，则可选择排序中靠前的位置进行截断；若更重视“查全率”，则可选择靠后的位置进行截断。因此，排序本身的质量好坏，体现了综合考虑学习器在不同任务下的“期望泛化性能”的好坏，或者说，“一般情况下”泛化性能的好坏ROC曲线则是从这个角度出发来研究学习器泛化性能的有力工具。

ROC全称是“受试者工作特征”(ReceiverOperatingCharacteristic)曲线，它源于“二战”中用于敌机检测的雷达信号分析技术，二十世纪六七+年代开始被用于一些心理学、医学检测应用中，此后被引入机器学习领域Spackman，19891与2.3.2节中介绍的P-R曲相似我们根据学习器的预测结果对样例进行排序，按此顺序逐个把样本作为正例进行预测，每次计算出两个重要量的值，分别以它们为横、纵标作图，就得到了“ROC曲线”与P-R曲线使用查准率、查全率为纵、横轴不同ROC线的轴是“真正例率”(TruePositiveRate简称TPR)横轴是“假正例率”(FalsePositiveRate，简称FPR)基于表21中的符号，两者分别定义为



显示ROC曲线的图称为“ROC图”图2.4(a)给出了一个示意图，显然对角线对应于“随机猜测”模型，而点(0，1)则对应于将所有正例排在所有反例之前的“理想模型”。



ROC曲线：

ROC曲线是用于衡量分类模型在不同阈值下的表现的一种图示方式。它展示了真正例率（True Positive Rate，又称为召回率）和假正例率（False Positive Rate）之间的关系。在ROC曲线上，横轴表示假正例率（FPR），纵轴表示真正例率（TPR）。通俗地说，它展示了当我们改变分类器的阈值时，真正例率和假正例率的变化情况。

AUC：

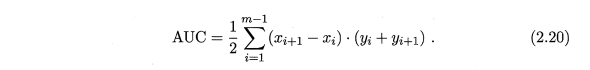
AUC则是ROC曲线下的面积，即ROC曲线下的积分值。AUC提供了一个单一的度量值，用于比较不同模型在各种阈值下的整体性能。AUC的取值范围在0到1之间，值越接近1表示模型性能越好，它代表了模型正确地预测正例样本排在负例样本前面的概率。

通俗地说，ROC曲线可以帮助我们了解模型在不同阈值下的表现，而AUC则是对整个ROC曲线的一个总结，用于比较模型的整体性能。如果一个模型的AUC值较高，说明在各种阈值下，该模型识别正负样本的能力较为稳健。

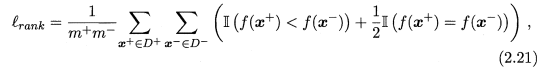
现实任务中通常是利用有限个测试样例来绘制ROC图此时仅能获得有限个(真正例率假正例率)标对无法产生图2.4(a)中的光滑ROC曲线只能绘制出如图2.4(b)所的近似ROC曲线图过程很简单:给定m个正例和m个反例，根据学习器预测结果对样例进行排序，然后把分类阈设为最大即把所有样例均预测为反例此时真正例率和假正例率均为0标(0，0)处标记一个点。然后，将分类阅值依次设为每个样例的预测值，即依次将每个样例划分为正例。设前一个标记点坐标为当前若为(x,y)真正例，则对应标记点的坐标为；当前为假正则对应标记点的为然后用线段连接相邻点即得。

进行学习器的比较时，与P-R图相似若一个学习器的ROC曲线被另一个学习器的曲线完全“包住”，则可断言后者的性能优于前者；若两个学习器的ROC曲线发生交叉，则难以一般性地断言两者孰优孰劣。此时如果一定要进行比较则较为合理的判据是比较ROC曲线下的面积即AUC(AreaUnderROCCurve)，如图2.4所示

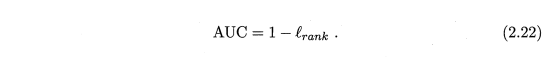
从定义可知AUC可通过对ROC曲线下各部分的面积求和而得假定ROC曲线是由标为的点按序连接而形成参见图2.4(b)，则AUC可估算为



形式化地看，AUC考虑的是样本预测的排序质量，因此它与排序误差有紧密联系给定m+个正例和m-个反例令D和D分别表示正、反例集合则排序“损失”(loss)定义为



即考虑每一对正、反例，若正例的预测值小于反例，则记一个“罚分”，若相等，则记0。5个“罚分”容易看出，rank对应的是ROC曲线之上的面积:若一个正例在ROC曲线上对应记点的为()则恰是排在其之前的反例所占的比例，即假正例率。因此有



## 4.4模型验证指标

常见的模型验证衡量指标和对应的sklearn方法如下表所示：

|  |  |  |
| --- | --- | --- |
| 指标 | 描述 | 方法 |
| Accuracy | 准确度 | from sklearn.metrics import accuracy\_score |
| Precision | 查准率 | from sklearn.metrics import precision\_score |
| Recall | 查全率 | from sklearn.metrics import import\_recall\_score |
| F1 | F1值 | from sklearn.metrics import f1\_score |
| Classification Report | 分类报告 | from sklearn.metrics import classification\_report |
| Confusion Matrix | 混淆矩阵 | from sklearn.metrics import confusion\_matrix |
| ROC | ROC曲线 | from sklearn.metrics import roc\_curve |
| AUC | ROC曲线下的面积 | from sklearn.metrics import auc |

4.4.1 准确度是预测正确的数(包括正样本和负样本)所占的数的比例

例如：矩阵X表示特征值，向量Y表示分类标签。

|  |
| --- |
| import numpy as np  X=np.random.random((10,5))  y=np.array(["M","M","F","F","M","F","M","M","F","F"])  X[X<0.7] = 0  X |

对矩阵X特征值进行二值化处理

|  |
| --- |
| from sklearn.preprocessing import Binarizer  binarizer=Binarizer(threshold=0.0).fit(X)  binarizer\_X=binarizer.transform(X)  binarizer\_X |

提示：对矩阵 X 的特征值进行二值化处理意味着将这些特征值转换为二进制值，通常是将它们映射为 0 和 1。总的来说，二值化特征值可能是为了简化数据、改变特征的表示方式、或者帮助模型更好地处理特征信息。然而，是否对特征值进行二值化处理应该根据具体情况和问题需求来确定，并且需要谨慎考虑其可能对数据和模型带来的影响

对Y标记进行类条件编码：

|  |
| --- |
| from sklearn.preprocessing import LabelEncoder  enc=LabelEncoder()  enc\_y=enc.fit\_transform(y)  enc\_y |

将数据切分为训练集和测试集

|  |
| --- |
| from sklearn.model\_selection import train\_test\_split  X\_train,X\_test,y\_train,y\_test=train\_test\_split(binarizer\_X,enc\_y,random\_state=0)  X\_train |

利用KNN模型对数据进行拟合

|  |
| --- |
| from sklearn.neighbors import KNeighborsClassifier  from sklearn import neighbors  knn=neighbors.KNeighborsClassifier(n\_neighbors=5)  knn.fit(X\_train,y\_train) |

提示：KNeighborsClassifier(algorithm='auto',leaf\_size=30,metric='minkowski',metric\_params=None,n\_jobs=None,n\_neighbors=5,p=2,weights='uniform')

K 最近邻分类器（KNeighborsClassifier）在 scikit-learn 中有许多参数，其中一些是默认设置的。这里列出一些常见的默认参数及其含义：

n\_neighbors：默认值为 5。它表示在进行分类时考虑的最近邻居的数量。对于给定样本，算法会找到离它最近的 n\_neighbors 个样本，并根据它们的标签进行分类决策。

weights：默认值为 'uniform'。用于预测的近邻的权重函数。如果设置为 'uniform'，则所有邻居的权重相等。如果设置为 'distance'，则权重与距离成反比，即越近的邻居对预测的影响越大。

algorithm：默认值为 'auto'。用于计算最近邻居的算法。可以是 'auto'、'ball\_tree'、'kd\_tree' 或 'brute'。通常情况下，'auto' 会根据训练数据的类型和情况选择最合适的算法。

leaf\_size：默认值为 30。它影响了使用 'ball\_tree' 或 'kd\_tree' 时的叶子节点的大小。这个参数会影响树构建的速度和存储空间。

p：默认值为 2。它表示在计算距离时所采用的距离度量。当 p = 1 时，使用曼哈顿距离（L1 距离），当 p = 2 时，使用欧式距离（L2 距离），其他数值表示使用闵可夫斯基距离。

metric：默认值为 'minkowski'。用于计算距离的度量标准。通常配合参数 p 使用，当 p = 2 时，'minkowski' 距离等同于欧式距离。

metric\_params：默认值为 None。用于度量标准的其他关键字参数。这些默认参数可以通过创建 KNeighborsClassifier 对象后进行更改，以适应特定的问题和数据集。这些参数的选择通常取决于数据的性质以及算法的性能需求。

**对测试数据集进行预测**

|  |
| --- |
| y\_pred=knn.predict(X\_test)  print("预测标签:",y\_pred)  print("实际标签:",y\_test) |

查准率和查全率预测：

|  |
| --- |
| from sklearn.metrics import precision\_score,recall\_score  print("查准率(precision):",precision\_score(y\_test,y\_pred))  print("查全率(recall)",recall\_score(y\_test,y\_pred)) |

F1值预测：

|  |
| --- |
| from sklearn.metrics import f1\_score  print("F1 score:",f1\_score(y\_test,y\_pred)) |

分类报告测试

metrics模块的分类报告方法，综合提供了查准率、查全率和F1值三种评估指标，示例代码如下：

|  |
| --- |
| from sklearn.metrics import classification\_report  print(classification\_report(y\_test,y\_pred)) |