## mln流程

1.拿到数据集，首先要进行数据预处理，如填补空值等

2.然后划分训练集和测试集，在数据量足够的情况下，使用留出法或交叉验证法，在数据量不是很足的情况下，使用留一法或者自助法

3.在必要情况下进行数据归一化

4.利用模型和训练集进行参数求解

5.利用测试集进行模型测试

6.进行数据预测

## 数据归一化

### 作用

1.提高归一化速度

就是将数据除以该列的最大值，是该列<=1，数据归一化可以是等高线显得很圆，在梯度下降时能够较快收敛。（梯度下降时沿垂直等高线的方向，如果等高线很圆，就能很快指向圆心）

2.有可能提高精度

一些分类器需要计算样本间的距离，如果一个特征值范围非常大，那么计算距离就主要靠这个特征，可能与实际情况相悖。归一化可以有效解决这个问题。

3.试用范围

概率模型不需要归一化，因为他不关心变量的值，而是关心变量的分布和变量之间的条件概率，如决策树、rf。

而像adaboost、gbdt、xgboost、svm、lr、KNN、Kmeans之类的最优化问题需要归一化。

### 方法

#### 1.线性归一化

x’= 

使用

[**preprocessing.MinMaxScaler**](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler)

进行归一化

#### 2.标准差归一化（0均值）





使用

[**preprocessing.StandardScaler**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler)

[**preprocessing.robust\_scale**](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.robust_scale.html#sklearn.preprocessing.robust_scale)

进行归一化

其中，μ、σ分别为原始数据集的均值和方法。该种归一化方式要求原始数据的分布可以近似为高斯分布，否则归一化的效果会变得很糟糕。

或者



#### 3.非线性归一化

应用于特征分布跨度很大，采用非线性归一化。通过一些函数，将原始值进行映射。一般采用log、指数函数、正切等函数。

#### 方法比较

1、在分类、聚类算法中，需要使用距离来度量相似性的时候、或者使用PCA技术进行降维的时候，第二种方法(Z-score standardization)表现更好。

2、在不涉及距离度量、协方差计算、数据不符合正太分布的时候，可以使用第一种方法或其他归一化方法。比如图像处理中，将RGB图像转换为灰度图像后将其值限定在[0 255]的范围。

## 梯度下降GD

梯度下降主要有三种

### 1.BGD

BGD 采用整个训练集的数据来计算 cost function 对参数的梯度。

**缺点：**

**由于这种方法是在一次更新中，就对整个数据集计算梯度，所以计算起来非常慢，遇到很大量的数据集也会非常棘手，而且不能投入新数据实时更新模型。**

### 2.SGD

和 BGD 的一次用所有数据计算梯度相比，SGD 每次更新时对每个样本进行梯度更新，对于很大的数据集来说，可能会有相似的样本，这样 BGD 在计算梯度时会出现冗余，而**SGD 一次只进行一次更新，就没有冗余，而且比较快，并且可以新增样本。**

随机梯度下降是通过每个样本来迭代更新一次，如果样本量很大的情况，那么可能只用其中部分的样本，就已经将theta迭代到最优解了，对比上面的批量梯度下降，迭代一次需要用到十几万训练样本，一次迭代不可能最优，如果迭代10次的话就需要遍历训练样本10次。

**缺点是SGD的噪音较BGD要多，使得SGD并不是每次迭代都向着整体最优化方向**。**所以虽然训练速度快，但是准确度下降，并不是全局最优**。**虽然包含一定的随机性，但是从期望上来看，它是等于正确的导数的。**

**缺点：**

**SGD 因为更新比较频繁，会造成 cost function 有严重的震荡。**

**BGD 可以收敛到局部极小值，当然 SGD 的震荡可能会跳到更好的局部极小值处。**

**当我们稍微减小 learning rate，SGD 和 BGD 的收敛性是一样的。**

### 3.MBGD

**MBGD 每一次利用一小批样本，即 n 个样本进行计算，这样它可以降低参数更新时的方差，收敛更稳定，另一方面可以充分地利用深度学习库中高度优化的矩阵操作来进行更有效的梯度计算。**

**超参数设定值:  n 一般取值在 50～256**

**缺点：（两大缺点）**

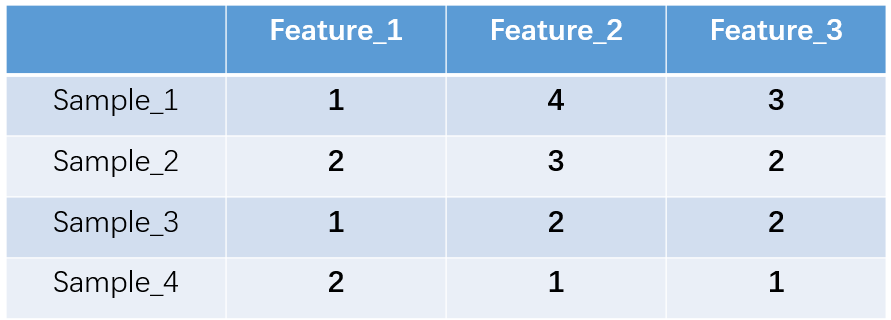
1. **不过 Mini-batch gradient descent 不能保证很好的收敛性，learning rate 如果选择的太小，收敛速度会很慢，如果太大，loss function 就会在极小值处不停地震荡甚至偏离。（有一种措施是先设定大一点的学习率，当两次迭代之间的变化低于某个阈值后，就减小 learning rate，不过这个阈值的设定需要提前写好，这样的话就不能够适应数据集的特点。）**对于非凸函数，还要避免陷于局部极小值处，或者鞍点处，因为鞍点周围的error是一样的，所有维度的梯度都接近于0，SGD 很容易被困在这里。（**会在鞍点或者局部最小点震荡跳动，因为在此点处，如果是训练集全集带入即BGD，则优化会停止不动，如果是mini-batch或者SGD，每次找到的梯度都是不同的，就会发生震荡，来回跳动。**）
2. SGD对所有参数更新时应用同样的 learning rate，如果我们的数据是稀疏的，**我们更希望对出现频率低的特征进行大一点的更新。LR会随着更新的次数逐渐变小。**

## 特征选取

### onehot

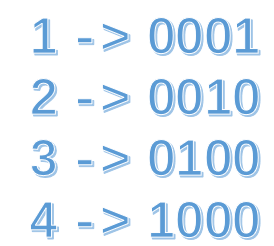
#### 1.onehot编码

什么是one-hot编码？one-hot编码，又称独热编码、一位有效编码。其方法是使用N位状态寄存器来对N个状态进行编码，每个状态都有它独立的寄存器位，并且在任意时候，其中只有一位有效。举个例子，假设我们有四个样本（行），每个样本有三个特征（列）。

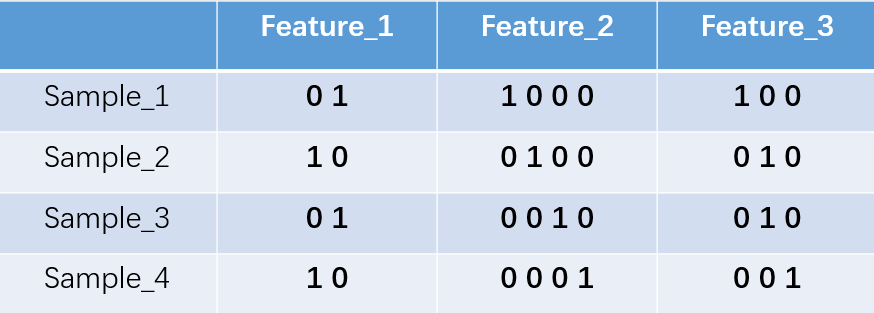


上图中我们已经对每个特征进行了普通的数字编码：我们的feature\_1有两种可能的取值，比如是男/女，这里男用1表示，女用2表示。那么one-hot编码是怎么搞的呢？我们再拿feature\_2来说明：

这里feature\_2 有4种取值（状态），我们就用4个状态位来表示这个特征，one-hot编码就是保证每个样本中的单个特征只有1位处于状态1，其他的都是0。



对于2种状态、三种状态、甚至更多状态都是这样表示，所以我们可以得到这些样本特征的新表示：



one-hot编码将每个状态位都看成一个特征。对于前两个样本我们可以得到它的特征向量分别为

#### C:\Users\Administrator\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.Word\1251096-20171030165731683-1946521226.png2 one-hot在提取文本特征上的应用

　　one hot在特征提取上属于词袋模型（bag of words）。关于如何使用one-hot抽取文本特征向量我们通过以下例子来说明。假设我们的语料库中有三段话：

　　　　我爱中国

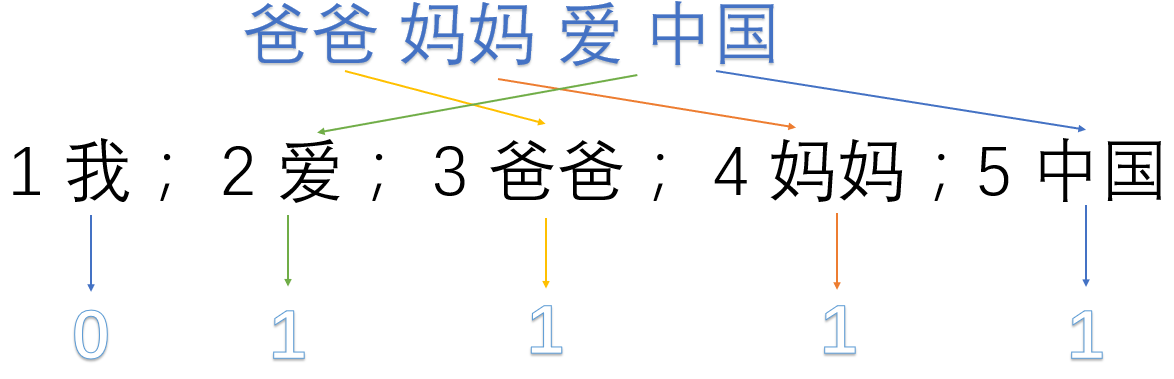
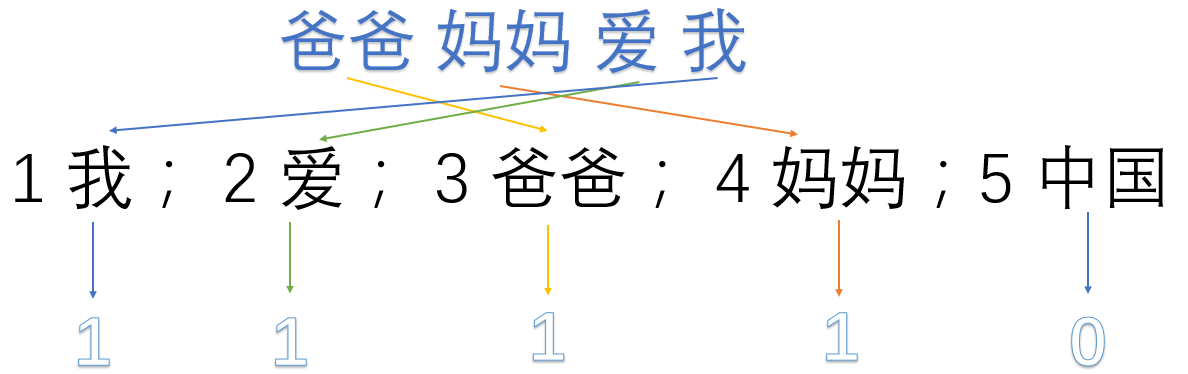
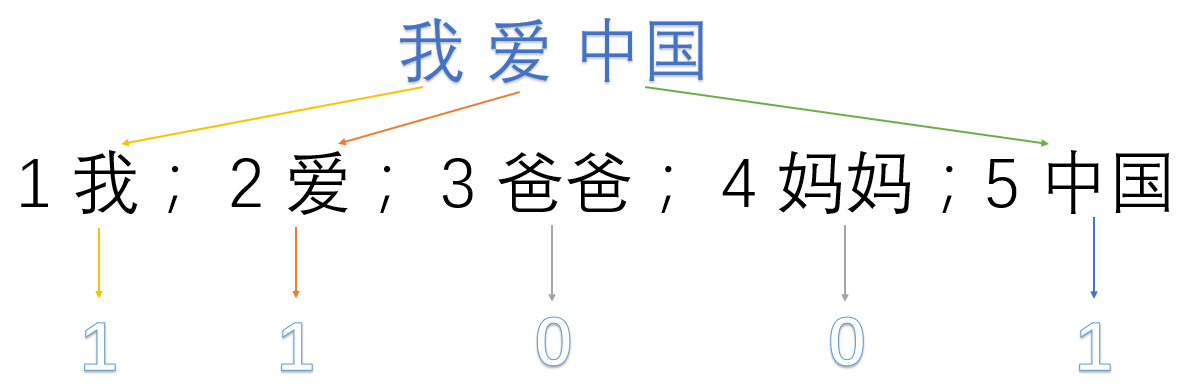
　　　　爸爸妈妈爱我

　　　　爸爸妈妈爱中国

我们首先对预料库分离并获取其中所有的词，然后对每个此进行编号：

　　　　1 我； 2 爱； 3 爸爸； 4 妈妈；5 中国

然后使用one hot对每段话提取特征向量：



因此我们得到了最终的特征向量为

　　　　我爱中国 　->　　　1，1，0，0，1

　　　　爸爸妈妈爱我　　->　　1，1，1，1，0

　　　　爸爸妈妈爱中国　　->　　0，1，1，1，1

优缺点分析

**优点：**一是解决了分类器不好处理离散数据的问题，二是在一定程度上也起到了扩充特征的作用（上面样本特征数从3扩展到了9）

**缺点：**在文本特征表示上有些缺点就非常突出了。首先，它是一个词袋模型，不考虑词与词之间的顺序（文本中词的顺序信息也是很重要的）；其次，它假设词与词相互独立（在大多数情况下，词与词是相互影响的）；最后，它得到的特征是离散稀疏的。

#### sklearn实现one hot encode

from sklearn import preprocessing

enc = preprocessing.OneHotEncoder() # 创建对象

enc.fit([[0,0,3],[1,1,0],[0,2,1],[1,0,2]]) # 拟合

array = enc.transform([[0,1,3]]).toarray() # 转化

print(array)

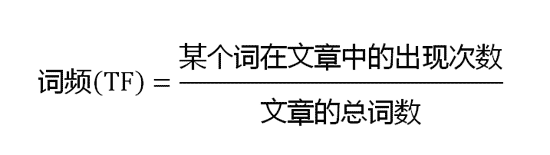
输入中，每一行为一个特征，每一列为一个输入。

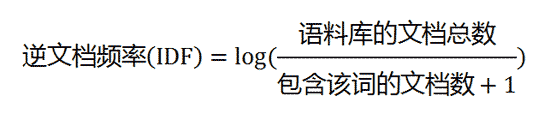
### 2. TF-IDF

#### 基本内容

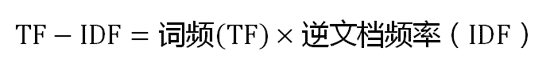
　　IF-IDF是信息检索（IR）中最常用的一种文本表示法。算法的思想也很简单，就是统计每个词出现的词频（TF），然后再为其附上一个权值参数（IDF）。举个例子：

　现在假设我们要统计一篇文档中的前10个关键词，应该怎么下手？首先想到的是统计一下文档中每个词出现的频率（TF），词频越高，这个词就越重要。但是统计完你可能会发现你得到的关键词基本都是“的”、“是”、“为”这样没有实际意义的词（停用词），这个问题怎么解决呢？你可能会想到为每个词都加一个权重，像这种”停用词“就加一个很小的权重（甚至是置为0），这个权重就是IDF。下面再来看看公式：





IF应该很容易理解就是计算词频，IDF衡量词的常见程度。为了计算IDF我们需要事先准备一个语料库用来模拟语言的使用环境，如果一个词越是常见，那么式子中分母就越大，逆文档频率就越小越接近于0。这里的分母+1是为了避免分母为0的情况出现。TF-IDF的计算公式如下：



根据公式很容易看出，TF-IDF的值与该词在文章中出现的频率成正比，与该词在整个语料库中出现的频率成反比，因此可以很好的实现提取文章中关键词的目的。

#### 优缺点分析

**优点：**简单快速，结果比较符合实际

**缺点：**单纯考虑词频，忽略了词与词的位置信息以及词与词之间的相互关系。

#### sklearn实现tfidf

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.feature\_extraction.text import TfidfTransformer

tag\_list = ['青年 吃货 唱歌',

'少年 游戏 叛逆',

'少年 吃货 足球']

vectorizer = CountVectorizer() #将文本中的词语转换为词频矩阵

X = vectorizer.fit\_transform(tag\_list) #计算个词语出现的次数

"""

word\_dict = vectorizer.vocabulary\_

{'唱歌': 2, '吃货': 1, '青年': 6, '足球': 5, '叛逆': 0, '少年': 3, '游戏': 4}

"""

transformer = TfidfTransformer()

tfidf = transformer.fit\_transform(X) #将词频矩阵X统计成TF-IDF值

print(tfidf.toarray())

<https://blog.csdn.net/m0_37324740/article/details/79411651>

将所有词都汇总在一个列表中，vectorizer.get\_feature\_names()

X返回的是一组元组，元组第一个元素表示的是输入文本所在的行，第二个元素表示该行的某个词在词汇总列表的位置。

### 特征交叉

sklearn.preprocessing.PolynomialFeatures

特征交叉就是根据需要将2个或多个特征进行融合，得出新的的特征，便于数据分析计算。

onehot其实也算是一种特征交叉。

sklearn.preprocessing.PolynomialFeatures来实现一般的特征交叉。

在建模过程中多次用到过sklearn.preprocessing.PolynomialFeatures，可以理解为专门生成多项式特征，并且多项式包含的是相互影响的特征集，比如：一个输入样本是２维的。形式如[a,b] ,则二阶多项式的特征集如下[1,a,b,a^2,ab,b^2]。

官网文档：<http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html>

参数：

degree : integer，多项式阶数，默认为2，即计算二次方

interaction\_only : boolean, default = False，如果值为true(默认是false),则会产生相互影响的特征集。如果为True，二次方就不计算a^2和b^2，即不计算自己和自己交互的项。三次方就不计算

include\_bias : boolean，是否包含偏差列。即是否包含常数项

示例：

>>> X = np.arange(6).reshape(3, 2)

>>> X

array([[0, 1],

[2, 3],

[4, 5]])

>>> poly = PolynomialFeatures(2)　＃设置多项式阶数为２，其他的默认

>>> poly.fit\_transform(X)

array([[ 1., 0., 1., 0., 0., 1.],

[ 1., 2., 3., 4., 6., 9.],

[ 1., 4., 5., 16., 20., 25.]])

>>> poly = PolynomialFeatures(interaction\_only=True)＃默认的阶数是２，同时设置交互关系为true

>>> poly.fit\_transform(X)

array([[ 1., 0., 1., 0.],

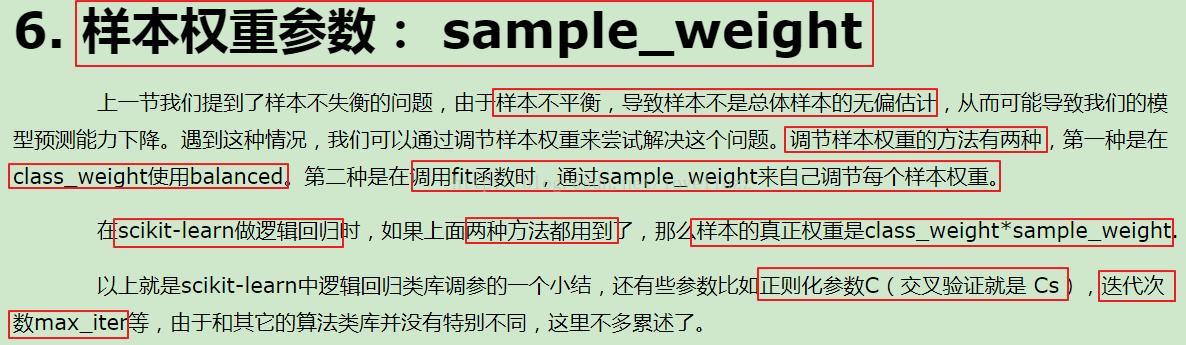
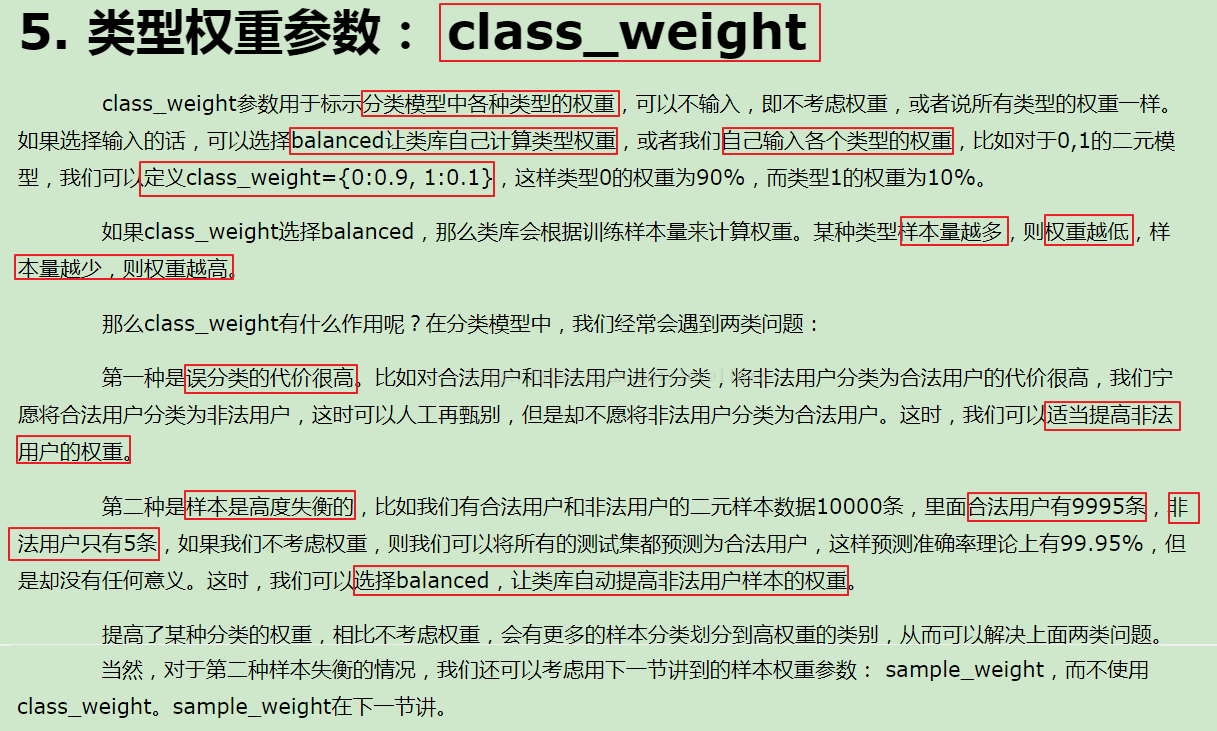
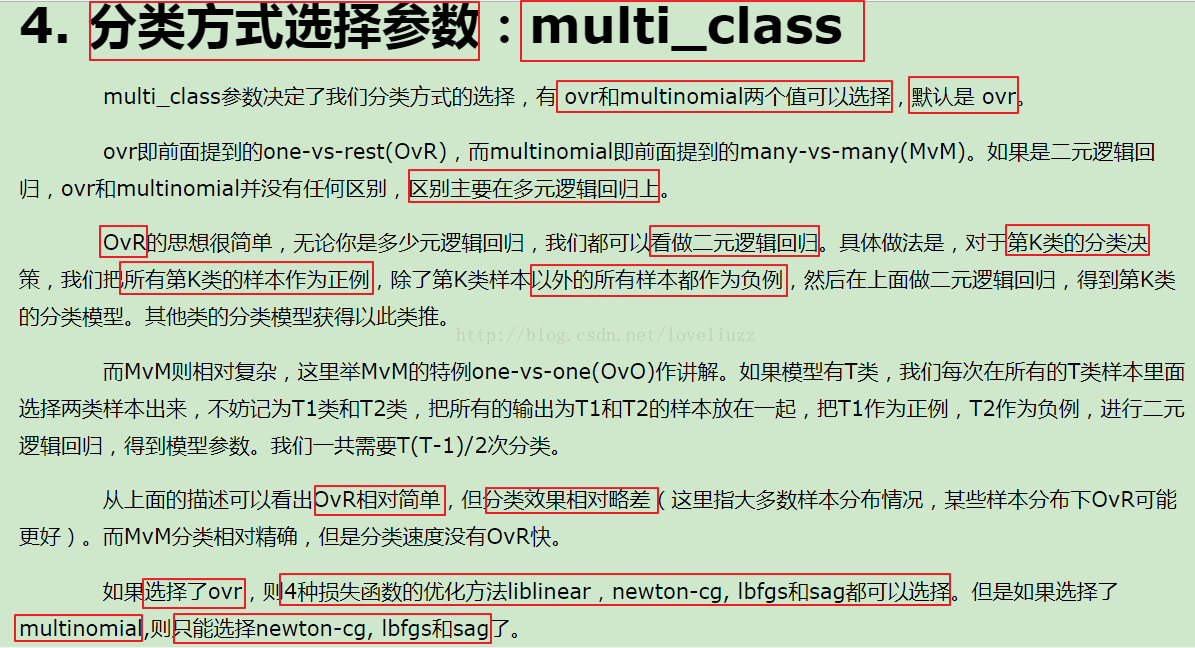
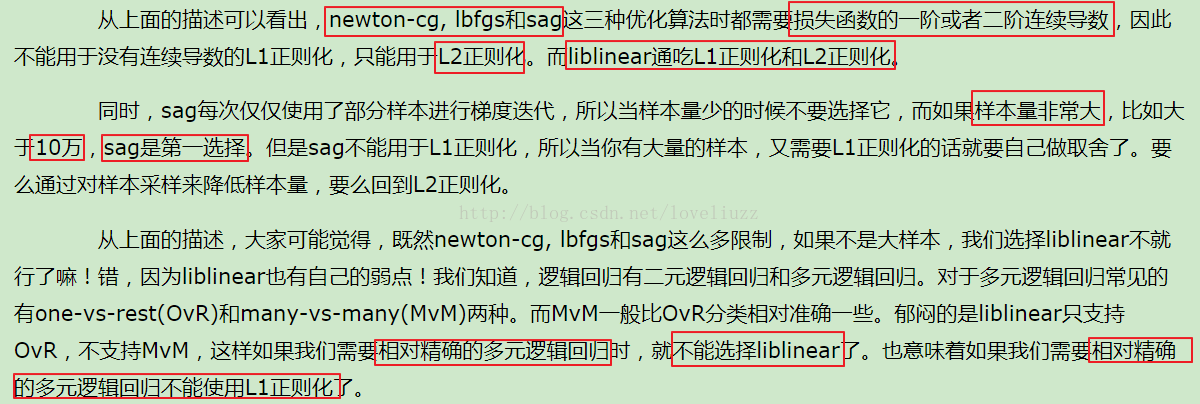
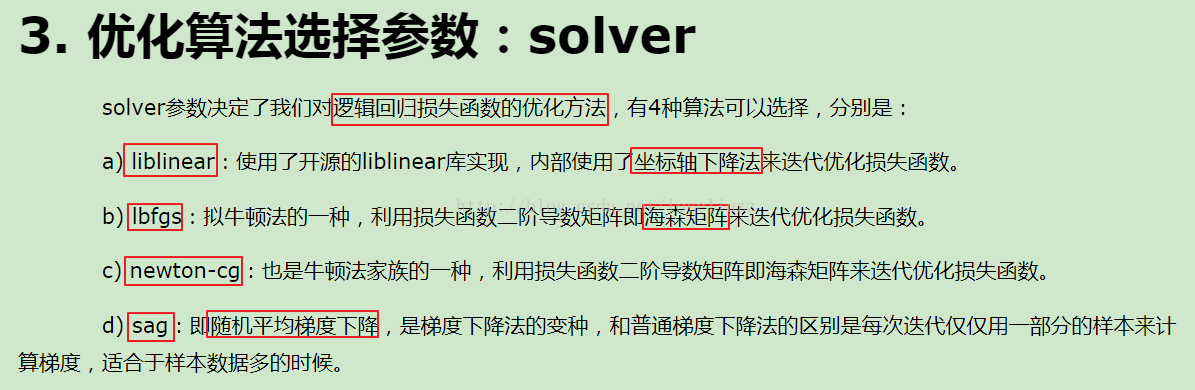
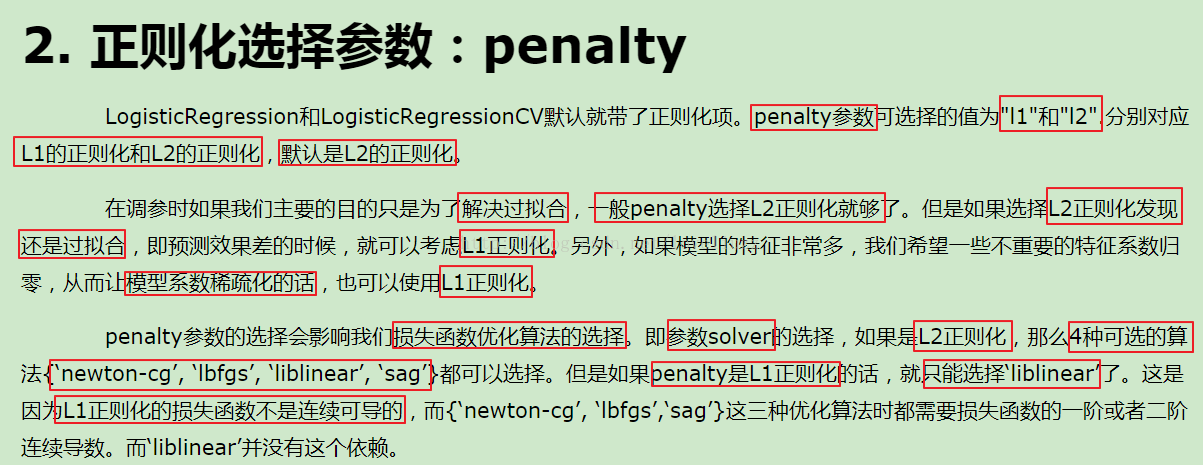
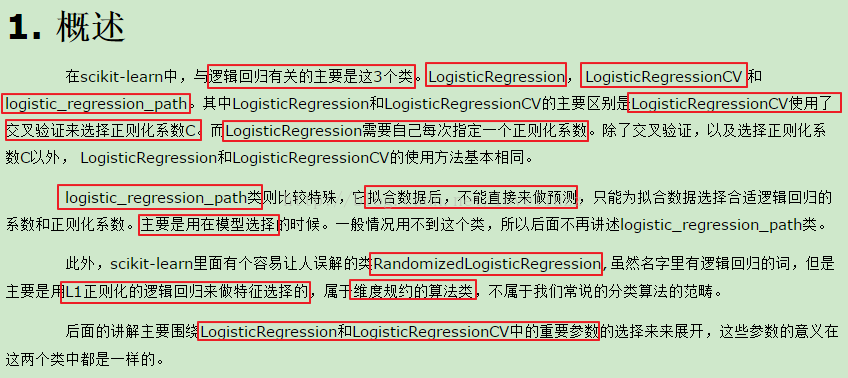
[ 1., 2., 3., 6.],

[ 1., 4., 5., 20.]])

上面的数组中，每一行是一个list。比如[0,1] 类似与上面的[a,b]。好的现在它的多项式输出矩阵就是[1,a,b,a^2,ab,b^2]。所以就是下面对应的[1,0,1,0,0,1]。现在将interaction\_only=True。这时就是只找交互作用的多项式输出矩阵。例如[a,b]的多项式交互式输出[1,a,b,ab]。不存在自己与自己交互的情况如：a^2或者b^2之类的。

## logistic回归算法LR

<https://blog.csdn.net/loveliuzz/article/details/78708359>

skilearn中的logistic回归方法：

### Logistics Regression参数名称

函数调用形式

LogisticRegression(penalty='l2',dual=False,tol=1e-4,C=1.0,fit\_intercept=True,intercept\_scaling=1,class\_weight=None,random\_state=None,solver='liblinear',max\_iter=100,multi\_class='ovr',verbose=0,warm\_start=False, n\_jobs=1)

penalty

字符串型，’l1’ or ‘l2’，默认：’l2’；正则化类型。

dual

布尔型，默认：False。当样本数>特征数时，令dual=False；用于liblinear解决器中L2正则化。

tol

浮点型，默认：1e-4；迭代终止判断的误差范围。

C

浮点型，默认：1.0；其值等于正则化强度的倒数，为正的浮点数。数值越小表示正则化越强。

fit\_intercept

布尔型，默认：True；指定是否应该向决策函数添加常量(即偏差或截距)。

intercept\_scaling

浮点型，默认为1；仅仅当solver是”liblinear”时有用。

class\_weight

默认为None；与“{class\_label: weight}”形式中的类相关联的权重。如果不给，则所有的类的权重都应该是1。

random\_state

整型，默认None；当“solver”==“sag”或“liblinear”时使用。在变换数据时使用的伪随机数生成器的种子。如果是整数, random\_state为随机数生成器使用的种子;若为RandomState实例，则random\_state为随机数生成器;如果没有，随机数生成器就是' np.random '使用的RandomState实例。

solver

{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}，默认: 'liblinear'；用于优化问题的算法。

对于小数据集来说，“liblinear”是个不错的选择，而“sag”和'saga'对于大型数据集会更快。

对于多类问题，只有'newton-cg'， 'sag'， 'saga'和'lbfgs'可以处理多项损失;“liblinear”仅限于“one-versus-rest”分类。

max\_iter

最大迭代次数，整型，默认是100；

multi\_class

字符串型，{ovr'， 'multinomial'}，默认:'ovr'；如果选择的选项是“ovr”，那么一个二进制问题适合于每个标签，否则损失最小化就是整个概率分布的多项式损失。对liblinear solver无效。

verbose

整型，默认是0；对于liblinear和lbfgs solver，verbose可以设为任意正数。

warm\_start

布尔型，默认为False；当设置为True时，重用前一个调用的解决方案以适合初始化。否则，只擦除前一个解决方案。对liblinear解码器无效。

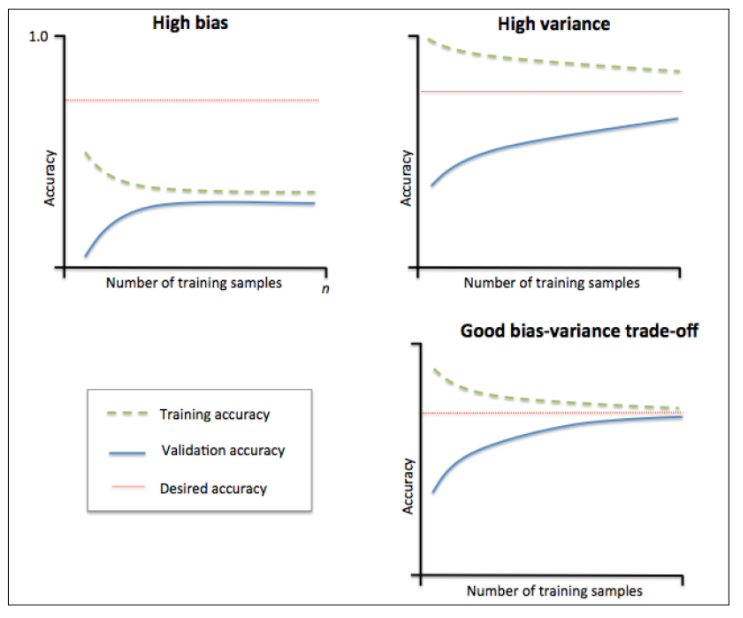
n\_jobs

整型，默认是1；如果multi\_class='ovr' ，则为在类上并行时使用的CPU核数。无论是否指定了multi\_class，当将' solver ' '设置为'liblinear'时，将忽略此参数。如果给定值为-1，则使用所有核。

## 判断过拟合和欠拟合

学习曲线是什么？

学习曲线就是通过画出不同训练集大小时训练集和交叉验证的准确率，可以看到模型在新数据上的表现，进而来判断模型是否方差偏高或偏差过高，以及增大训练集是否可以减小过拟合。



**绿色虚线**为训练集正确率，**蓝色实线**为验证集正确率（validation accuracy），红色实线为所希望的正确率。

怎么解读？

当训练集和测试集的误差收敛但却很高时，为高偏差。

左上角的偏差很高，训练集和验证集的准确率都很低，很可能是欠拟合。

我们可以增加模型参数，比如，构建更多的特征，减小正则项。

此时通过增加数据量是不起作用的。

当训练集和测试集的误差之间有大的差距时，为高方差。

当训练集的准确率比其他独立数据集上的测试结果的准确率要高时，一般都是过拟合。

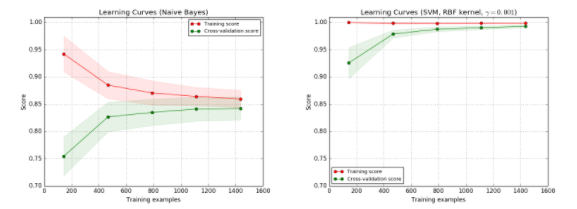
右上角方差很高，训练集和验证集的准确率相差太多，应该是过拟合。

我们可以增大训练集，降低模型复杂度，增大正则项，或者通过特征选择减少特征数。

理想情况是是找到偏差和方差都很小的情况，即收敛且误差较小。

怎么画？

在画学习曲线时，横轴为训练样本的数量，纵轴为准确率。



例如同样的问题，左图为我们用 naive Bayes 分类器时，效果不太好，分数大约收敛在 0.85，此时增加数据对效果没有帮助。

右图为 SVM（RBF kernel），训练集的准确率很高，验证集的也随着数据量增加而增加，不过因为训练集的还是高于验证集的，有点过拟合，所以还是需要增加数据量，这时增加数据会对效果有帮助。

上图的代码如下：

模型这里用 GaussianNB 和 SVC 做比较，

模型选择方法中需要用到 learning\_curve 和交叉验证方法 ShuffleSplit。

import numpy as np

import matplotlib.pyplot as plt

from sklearn.naive\_bayes import GaussianNB

from sklearn.svm import SVC

from sklearn.datasets import load\_digits

from sklearn.model\_selection import learning\_curve

from sklearn.model\_selection import ShuffleSplit

首先定义画出学习曲线的方法，

核心就是调用了 sklearn.model\_selection 的 learning\_curve，

学习曲线返回的是 train\_sizes, train\_scores, test\_scores，

画训练集的曲线时，横轴为 train\_sizes, 纵轴为 train\_scores\_mean，

画测试集的曲线时，横轴为 train\_sizes, 纵轴为 test\_scores\_mean：

def plot\_learning\_curve(estimator, title, X, y, ylim=None, cv=None,

n\_jobs=1, train\_sizes=np.linspace(.1, 1.0, 5)):

train\_sizes, train\_scores, test\_scores = learning\_curve(

estimator, X, y, cv=cv, n\_jobs=n\_jobs, train\_sizes=train\_sizes)

train\_scores\_mean = np.mean(train\_scores, axis=1)

test\_scores\_mean = np.mean(test\_scores, axis=1)

在调用 plot\_learning\_curve 时，首先定义交叉验证 cv 和学习模型 estimator。

这里交叉验证用的是 ShuffleSplit， 它首先将样例打散，并随机取 20％ 的数据作为测试集，这样取出 100 次，最后返回的是 train\_index, test\_index，就知道哪些数据是 train，哪些数据是 test。

estimator 用的是 GaussianNB，对应左图：

cv = ShuffleSplit(n\_splits=100, test\_size=0.2, random\_state=0)

estimator = GaussianNB()

plot\_learning\_curve(estimator, title, X, y, ylim=(0.7, 1.01), cv=cv, n\_jobs=4)

cv = ShuffleSplit(n\_splits=10, test\_size=0.2, random\_state=0)

estimator = SVC(gamma=0.001)

plot\_learning\_curve(estimator, title, X, y, (0.7, 1.01), cv=cv, n\_jobs=4)

## 模型评估方法及sklearn实现（选训练集和测试集）

<https://blog.csdn.net/luanpeng825485697/article/details/79836262>

### 1.留出法 hold-out

#### ①方法

将数据集D划分为两个互斥的集合，其中一个集合S作为训练集，另一个集合T作为测试集。一般当样本比较多的时候使用。

将数据集2/3 -- 4/5用于训练，剩余样本用于测试。

单次留出法结果不够稳定可靠，一般采用多次随机划分，重复试验评估后，取平均值作为评估结果。

#### ②sklearn实现

sklearn.model\_selection. train\_test\_split()

　在sklearn中我们使用sklearn.model\_selection中的train\_test\_split()来分割我们的数据集，其具体参数如下：

参数：

X：待分割的样本集中的自变量部分，通常为二维数组或矩阵的形式；

y：待分割的样本集中的因变量部分，通常为一维数组；

test\_size：用于指定验证集所占的比例，有以下几种输入类型：

　　 1.float型，0.0~1.0之间，此时传入的参数即作为验证集的比例；

　　 2.int型，此时传入的参数的绝对值即作为验证集样本的数量；

　 　3.None，这时需要另一个参数train\_size有输入才生效，此时验证集去为train\_size指定的比例或数量的补集；

　 　4.缺省时为0.25，但要注意只有在train\_size和test\_size都不输入值时缺省值才会生效；

train\_size：基本同test\_size，但缺省值为None，其实test\_size和train\_size输入一个即可；

random\_state：int型，控制随机数种子，默认为None，即纯随机（伪随机）；其实是随机数的种子。

随机数种子：其实就是该组随机数的编号，在需要重复试验的时候，保证得到一组一样的随机数。比如你每次都填1，其他参数一样的情况下你得到的随机数组是一样的。但填0或不填，每次都会不一样。

随机数的产生取决于种子，随机数和种子之间的关系遵从以下两个规则：

种子不同，产生不同的随机数；种子相同，即使实例不同也产生相同的随机数。

stratify：控制分类问题中的分层抽样，默认为None，即不进行分层抽样，当传入为数组时，则依据该数组进行分层抽样（一般传入因变量所在列）；

shuffle：bool型，用来控制是否在分割数据前打乱原数据集的顺序，默认为True，**分层抽样时即stratify存在时，该参数必须传入False；**

返回值：

依次返回训练集自变量、测试集自变量、训练集因变量、测试集因变量，因此使用该函数赋值需在等号右边采取X\_train, X\_test, y\_train, y\_test'的形式。

#### 示例

未分层情况：

from sklearn.model\_selection import train\_test\_split

from sklearn import datasets

import pandas as pd

'''载入数据'''

X,y = datasets.load\_iris(return\_X\_y=True)

'''不采取分层抽样时的数据集分割'''

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X,y,test\_size=0.3)

'''打印各个数据集的形状'''

print(X\_train.shape,X\_test.shape,y\_train.shape,y\_test.shape)

'''打印训练集中因变量的各类别数目情况'''

print(pd.value\_counts(y\_train))

'''打印验证集集中因变量的各类别数目情况'''

print(pd.value\_counts(y\_test))

分层情况：

'''采取分层抽样时的数据集分割'''

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X,y,test\_size=0.3,stratify=y)

'''打印各个数据集的形状'''

print(X\_train.shape,X\_test.shape,y\_train.shape,y\_test.shape)

'''打印训练集中因变量的各类别数目情况'''

print(pd.value\_counts(y\_train))

'''打印验证集集中因变量的各类别数目情况'''

print(pd.value\_counts(y\_test))

### 2.交叉验证法和留一法

#### ①方法

将数据集D划分为k个大小相似的互斥子集，，每个子集都尽可能保持数据分布的一致性，一般通过分层采样得到。然后每次使用k-1个自己的并集作为训练集，剩下的哪一个子集为测试集，获得k组训练/测试集，根据k个测试结果的均值。又称为‘k折交叉验证法’，k常取10。其他常见的有取5或20.

特殊的，如果样本集D较小，含有m个样本，令k=m，即每个数据为一个子集，则成为留一法。

#### sklearn实现

在sklearn.model\_selection中集成了众多用于交叉验证的方法，下面对其中常用的进行介绍：

##### cross\_val\_score()：

　　这是一个用于直接计算某个已确定参数的模型其交叉验证分数的方法，具体参数如下：

参数：

estimator：已经初始化的学习器模型；

X：自变量所在的数组；

y：因变量所在的数组；

scoring：str型，控制函数返回的模型评价指标，默认为准确率；

cv：控制交叉验证中分割样本集的策略，即k折交叉中的k，**默认是3**，即3折交叉验证，有以下多种输入形式：

　　1.int型，则输入的参数即为k；

　　2.None，则使用默认的3折；

　　3.一个生成器类型的对象，用来控制交叉验证，优点是节省内存，下面的演示中会具体介绍；

　　\*若estimator是一个分类器，则默认使用分层抽样来产生子集。

n\_jobs：int型，用来控制并行运算中使用的核心数，默认为1，即单核；特别的，设置为 -1 时开启所有核心；

函数返回值：

对应scoring指定的cv个评价指标；

下面以一个简单的小例子进行演示：

from sklearn.model\_selection import cross\_val\_score

from sklearn import datasets

from sklearn.neighbors import KNeighborsClassifier

X,y = datasets.load\_breast\_cancer(return\_X\_y=True)

clf = KNeighborsClassifier()

'''打印每次交叉验证的准确率'''

score = cross\_val\_score(clf,X,y,cv=5,scoring='accuracy')

print('accuracy:'+str(score)+'\n')

'''打印每次交叉验证的f1得分'''

score = cross\_val\_score(clf,X,y,cv=5,scoring='f1')

print('f1 score:'+str(score)+'\n')

'''打印正确率的95%置信区间'''

print(str(round(score.mean(),3))+'(+/-'+str(round(2\*score.std(),3))+')')

##### cross\_validate():

　　这个方法与cross\_val\_score()很相似，但有几处新特性：

　　1.cross\_validate()可以返回多个评价指标，这在需要一次性产生多个不同种类评分时很方便；

　　2.cross\_validate()不仅返回模型评价指标，还会返回训练花费时长、

 其具体参数如下：

参数：

estimator：已经初始化的分类器模型；

X：自变量；

y：因变量；

scoring：字符型或列表形式的多个字符型，控制产出的评价指标，可以通过在列表中写入多个评分类型来实现多指标输出；

cv：控制交叉验证的子集个数；

n\_jobs：控制并行运算利用的核心数，同cross\_val\_score()；

return\_train\_score：bool型，控制是否在得分中计算训练集回带进模型的结果；

输出：

函数输出项：字典形式的训练时间、计算得分时间、及各得分情况；

下面以一个简单的小例子进行说明：

from sklearn.model\_selection import cross\_validate

from sklearn import datasets

from sklearn.neighbors import KNeighborsClassifier

X,y = datasets.load\_breast\_cancer(return\_X\_y=True)

clf = KNeighborsClassifier()

'''定义需要输出的评价指标'''

scoring = ['accuracy','f1']

'''打印每次交叉验证的准确率'''

score = cross\_validate(clf,X,y,scoring=scoring,cv=5,return\_train\_score=True)

score

#### sklearn基于生成器的交叉验证法

sklearn中除了上述的直接完成整套交叉验证的方法外，还存在着一些基于生成器的方法，这些方法的好处是利用Python中生成器（generator）的方式，以非常节省内存的方式完成每一次的交叉验证，下面一一罗列：

##### KFold():

　　以生成器的方式产出每一次交叉验证所需的训练集与验证集，其主要参数如下：

n\_splits：int型，控制k折交叉中的k，默认是3；

shuffle：bool型，控制是否在采样前打乱原数据顺序；

random\_state：设置随机数种子，默认为None，即不固定随机水平；

下面以一个简单的小例子进行演示：

from sklearn.model\_selection import KFold

import numpy as np

X = np.random.randint(1,10,20)

kf = KFold(n\_splits=5)

for train,test in kf.split(X):

print(train,'\n',test)

##### LeaveOneOut():留一法

　　对应先前所介绍的留出法中的特例，留一法，因为其性质很固定，所以无参数需要调节，下面以一个简单的小例子进行演示：

from sklearn.model\_selection import LeaveOneOut

import numpy as np

X = np.random.randint(1,10,5)

kf = LeaveOneOut()

for train,test in kf.split(X):

print(train,'\n',test)

##### LeavePOut():

　　LeaveOneOut()的一个变种，唯一的不同就是每次留出p个而不是1个样本作为验证集，唯一的参数是p，下面是一个简单的小例子：

from sklearn.model\_selection import LeavePOut

import numpy as np

X = np.random.randint(1,10,5)

kf = LeavePOut(p=2)

for train,test in kf.split(X):

print(train,'\n',test)

##### TimeSeriesSplit():

　　在机器学习中还存在着一种叫做时间序列的数据类型，这种数据的特点是高度的自相关性，前后相邻时段的数据关联程度非常高，因此在对这种数据进行分割时不可以像其他机器学习任务那样简单随机抽样的方式采样，对时间序列数据的采样不能破坏其时段的连续型，在sklearn.model\_selection中我们使用TimeSeriesSplit()来分割时序数据，其主要参数如下：

n\_splits：int型，控制产生（训练集+验证集）的数量；

max\_train\_size：控制最大的时序数据长度；

下面是一个简单的小例子：

from sklearn.model\_selection import TimeSeriesSplit

import numpy as np

X = np.random.randint(1,10,20)

kf = TimeSeriesSplit(n\_splits=4)

for train,test in kf.split(X):

print(train,'\n',test)

### 3.自助法

给定m个样本的数据集D，对其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数据集D中，使得该样本在下次采样中仍有可能被采样到；将这个过程重复m次后，得到了有m个样本的数据集D’，这就是自助采样的结果。D’中会含有多个重复的样本，样本在m次中不被采样到的概率为 (1-m)m,取极限得到



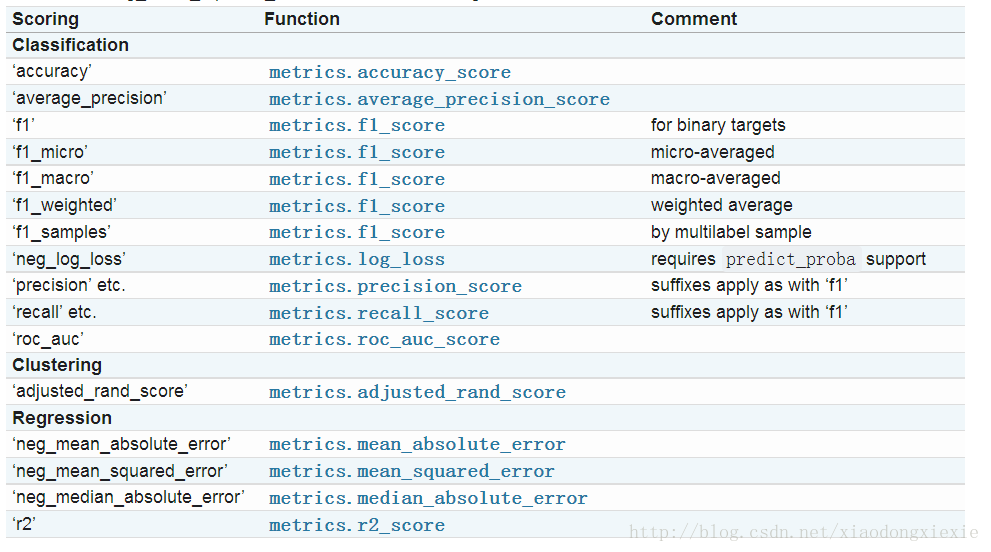
即通过自助采样，初始数据集D中约有36.8%的样本未出现在D’中，于是我们可以将D’作为测试集，D\D’作为测试集。这样，实际评估的模型与期望期望评估的模型都使用m个样本，而我们仍有1/3的样本用于测试，亦称为‘包外估计’。

自助法在数据集较小，难以有效划分训练/测试集时很有用。

在数据集足够时，留出法和交叉验证法更常用。

## sklearn.metrics中的评估方法介绍

<https://blog.csdn.net/CherDW/article/details/55813071>



### accuracy\_score

分类准确率分数是指所有分类正确的百分比。分类准确率这一衡量分类器的标准比较容易理解，但是它不能告诉你响应值的潜在分布，并且它也不能告诉你分类器犯错的类型。

形式：

sklearn.metrics.accuracy\_score(y\_true, y\_pred, normalize=True, sample\_weight=None)

normalize：默认值为True，返回正确分类的比例；如果为False，返回正确分类的样本数

### recall\_score

召回率 =提取出的正确信息条数 /样本中的信息条数。通俗地说，就是所有准确的条目有多少被检索出来了。

形式：

klearn.metrics.recall\_score(y\_true, y\_pred, labels=None, pos\_label=1,average='binary', sample\_weight=None)

参数average : string, [None, ‘micro’, ‘macro’(default), ‘samples’, ‘weighted’]

将一个二分类matrics拓展到多分类或多标签问题时，我们可以将数据看成多个二分类问题的集合，每个类都是一个二分类。接着，我们可以通过跨多个分类计算每个二分类metrics得分的均值，这在一些情况下很有用。你可以使用average参数来指定。

macro：计算二分类metrics的均值，为每个类给出相同权重的分值。当小类很重要时会出问题，因为该macro-averging方法是对性能的平均。另一方面，该方法假设所有分类都是一样重要的，因此macro-averaging方法会对小类的性能影响很大。

weighted:对于不均衡数量的类来说，计算二分类metrics的平均，通过在每个类的score上进行加权实现。

micro：给出了每个样本类以及它对整个metrics的贡献的pair（sample-weight），而非对整个类的metrics求和，它会每个类的metrics上的权重及因子进行求和，来计算整个份额。Micro-averaging方法在多标签（multilabel）问题中设置，包含多分类，此时，大类将被忽略。

samples：应用在multilabel问题上。它不会计算每个类，相反，它会在评估数据中，通过计算真实类和预测类的差异的metrics，来求平均（sample\_weight-weighted）

average：average=None将返回一个数组，它包含了每个类的得分。

### roc\_curve

ROC曲线指受试者工作特征曲线/接收器操作特性(receiver operating characteristic，ROC)曲线,是反映灵敏性和特效性连续变量的综合指标,是用构图法揭示敏感性和特异性的相互关系，它通过将连续变量设定出多个不同的临界值，从而计算出一系列敏感性和特异性。ROC曲线是根据一系列不同的二分类方式（分界值或决定阈），以真正例率（也就是灵敏度）（True Positive Rate,TPR）为纵坐标，假正例率（1-特效性）（False Positive Rate,FPR）为横坐标绘制的曲线。

ROC观察模型正确地识别正例的比例与模型错误地把负例数据识别成正例的比例之间的权衡。TPR的增加以FPR的增加为代价。ROC曲线下的面积是模型准确率的度量，AUC（Area under roccurve）。

纵坐标：真正率（True Positive Rate , TPR）或灵敏度（sensitivity）

TPR = TP /（TP + FN）  （正样本预测结果数 / 正样本实际数）

横坐标：假正率（False Positive Rate , FPR）

FPR = FP /（FP + TN） （被预测为正的负样本结果数 /负样本实际数）

形式：

sklearn.metrics.roc\_curve(y\_true,y\_score, pos\_label=None, sample\_weight=None, drop\_intermediate=True)

该函数返回这三个变量：fpr,tpr,和阈值thresholds;

这里理解thresholds:

分类器的一个重要功能“概率输出”，即表示分类器认为某个样本具有多大的概率属于正样本（或负样本）。

“Score”表示每个测试样本属于正样本的概率。

接下来，我们从高到低，依次将“Score”值作为阈值threshold，当测试样本属于正样本的概率大于或等于这个threshold时，我们认为它为正样本，否则为负样本。每次选取一个不同的threshold，我们就可以得到一组FPR和TPR，即ROC曲线上的一点。当我们将threshold设置为1和0时，分别可以得到ROC曲线上的(0,0)和(1,1)两个点。将这些(FPR,TPR)对连接起来，就得到了ROC曲线。当threshold取值越多，ROC曲线越平滑。其实，我们并不一定要得到每个测试样本是正样本的概率值，只要得到这个分类器对该测试样本的“评分值”即可（评分值并不一定在(0,1)区间）。评分越高，表示分类器越肯定地认为这个测试样本是正样本，而且同时使用各个评分值作为threshold。我认为将评分值转化为概率更易于理解一些。

### Auc

计算AUC值，其中x,y分别为数组形式，根据(xi,yi)在坐标上的点，生成的曲线，然后计算AUC值；

形式：

sklearn.metrics.auc(x, y, reorder=False)

### roc\_auc\_score

直接根据真实值（必须是二值）、预测值（可以是0/1,也可以是proba值）计算出auc值，中间过程的roc计算省略。

形式：

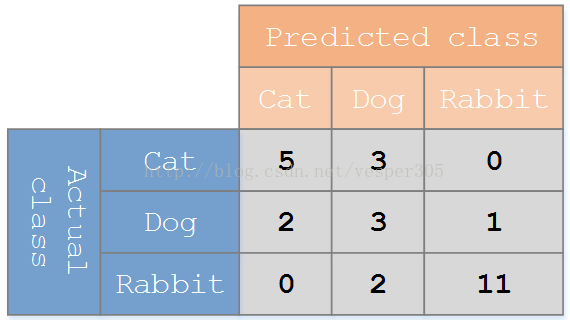
sklearn.metrics.roc\_auc\_score(y\_true,y\_score,average='macro', sample\_weight=None)

参数：**average** : string, [None, ‘micro’, ‘macro’(default), ‘samples’, ‘weighted’]

### confusion\_matrix

用一个例子来理解混淆矩阵：

假设有一个用来对猫（cats）、狗（dogs）、兔子（rabbits）进行分类的系统，混淆矩阵就是为了进一步分析性能而对该算法测试结果做出的总结。假设总共有 27 只动物：8只猫， 6条狗， 13只兔子。结果的混淆矩阵如下图：



在这个混淆矩阵中，实际有 8只猫，但是系统将其中3只预测成了狗；对于 6条狗，其中有 1条被预测成了兔子，2条被预测成了猫。从混淆矩阵中我们可以看出系统对于区分猫和狗存在一些问题，但是区分兔子和其他动物的效果还是不错的。所有正确的预测结果都在对角线上，所以从混淆矩阵中可以很方便直观的看出哪里有错误，因为他们呈现在对角线外面。

形式：

sklearn.metrics.confusion\_matrix(y\_true, y\_pred, labels=None, sample\_weight=None)

返回一个混淆矩阵；

labels：混淆矩阵的索引（如上面猫狗兔的示例），如果没有赋值，则按照y\_true, y\_pred中出现过的值排序。

## sklearn中的交叉验证（Cross-Validation）

<https://blog.csdn.net/xiaodongxiexie/article/details/71915259>

### 先导入需要的库及数据集

In [1]: import numpy as np

In [2]: from sklearn.model\_selection import train\_test\_split

In [3]: from sklearn.datasets import load\_iris

In [4]: from sklearn import svm

In [5]: iris = load\_iris()

In [6]: iris.data.shape, iris.target.shape

Out[6]: ((150, 4), (150,))

### 1.train\_test\_split（留出法）

对数据集进行**快速**打乱（分为训练集和测试集）

这里相当于对数据集进行了shuffle后按照给定的test\_size 进行数据集划分。

In [7]: X\_train, X\_test, y\_train, y\_test = train\_test\_split(

...: iris.data, iris.target, test\_size=.4, random\_state=0)

#这里是按照6:4对训练集测试集进行划分

In [8]: X\_train.shape, y\_train.shape

Out[8]: ((90, 4), (90,))

In [9]: X\_test.shape, y\_test.shape

Out[9]: ((60, 4), (60,))

In [10]: iris.data[:5]

Out[10]:

array([[ 5.1, 3.5, 1.4, 0.2],

[ 4.9, 3. , 1.4, 0.2],

[ 4.7, 3.2, 1.3, 0.2],

[ 4.6, 3.1, 1.5, 0.2],

[ 5. , 3.6, 1.4, 0.2]])

In [11]: X\_train[:5]

Out[11]:

array([[ 6. , 3.4, 4.5, 1.6],

[ 4.8, 3.1, 1.6, 0.2],

[ 5.8, 2.7, 5.1, 1.9],

[ 5.6, 2.7, 4.2, 1.3],

[ 5.6, 2.9, 3.6, 1.3]])

In [12]: clf = svm.SVC(kernel='linear', C=1).fit(X\_train, y\_train)

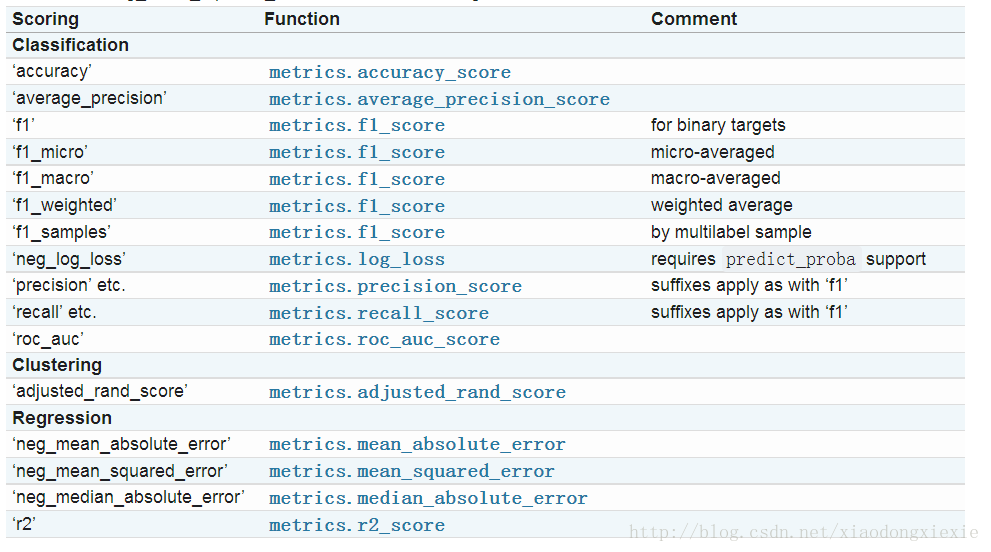
In [13]: clf.score(X\_test, y\_test)

Out[13]: 0.96666666666666667

### 2.cross\_val\_score

对数据集进行指定次数的交叉验证并为每次验证效果评测

其中，score 默认是以 scoring=’f1\_macro’进行评测的，余外针对分类或回归还有：



这需要from　sklearn import metrics ,通过在cross\_val\_score 指定参数来设定评测标准；

当cv 指定为int 类型时，默认使用KFold 或StratifiedKFold 进行数据集打乱，下面会对KFold 和StratifiedKFold 进行介绍。

In [15]: from sklearn.model\_selection import cross\_val\_score

In [16]: clf = svm.SVC(kernel='linear', C=1)

In [17]: scores = cross\_val\_score(clf, iris.data, iris.target, cv=5)

In [18]: scores

Out[18]: array([ 0.96666667, 1. , 0.96666667, 0.96666667, 1. ])

In [19]: scores.mean()

Out[19]: 0.98000000000000009

除使用默认交叉验证方式外，可以对交叉验证方式进行指定，如验证次数，训练集测试集划分比例等

In [20]: from sklearn.model\_selection import ShuffleSplit

In [21]: n\_samples = iris.data.shape[0]

In [22]: cv = ShuffleSplit(n\_splits=3, test\_size=.3, random\_state=0)

In [23]: cross\_val\_score(clf, iris.data, iris.target, cv=cv)

Out[23]: array([ 0.97777778, 0.97777778, 1. ])

在cross\_val\_score 中同样可使用pipeline 进行流水线操作

In [24]: from sklearn import preprocessing

In [25]: from sklearn.pipeline import make\_pipeline

In [26]: clf = make\_pipeline(preprocessing.StandardScaler(), svm.SVC(C=1))

In [27]: cross\_val\_score(clf, iris.data, iris.target, cv=cv)

Out[27]: array([ 0.97777778, 0.93333333, 0.95555556])

### 3.cross\_val\_predict

cross\_val\_predict 与cross\_val\_score 很相像，不过不同于返回的是评测效果，cross\_val\_predict 返回的是estimator 的分类结果（或回归值），这个对于后期模型的改善很重要，可以通过该预测输出对比实际目标值，准确定位到预测出错的地方，为我们参数优化及问题排查十分的重要。

In [28]: from sklearn.model\_selection import cross\_val\_predict

In [29]: from sklearn import metrics

In [30]: predicted = cross\_val\_predict(clf, iris.data, iris.target, cv=10)

In [31]: predicted

Out[31]:

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2,

2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2,

2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])

In [32]: metrics.accuracy\_score(iris.target, predicted)

Out[32]: 0.96666666666666667

从下面开始，返回的只是分组索引，通过循环，来进行交叉验证

### 4.KFold

K折交叉验证，这是将数据集分成K份的官方给定方案，所谓K折就是将数据集通过K次分割，使得所有数据既在训练集出现过，又在测试集出现过，当然，每次分割中不会有重叠。相当于无放回抽样。

In [33]: from sklearn.model\_selection import KFold

In [34]: X = ['a','b','c','d']

In [35]: kf = KFold(n\_splits=2)

In [36]: for train, test in kf.split(X):

...: print train, test

...: print np.array(X)[train], np.array(X)[test]

...: print '\n'

...:

[2 3] [0 1]

['c' 'd'] ['a' 'b']

[0 1] [2 3]

['a' 'b'] ['c' 'd']

### 5.LeaveOneOut

LeaveOneOut 其实就是KFold 的一个特例，因为使用次数比较多，因此独立的定义出来，完全可以通过KFold 实现。

In [37]: from sklearn.model\_selection import LeaveOneOut

In [38]: X = [1,2,3,4]

In [39]: loo = LeaveOneOut()

In [41]: for train, test in loo.split(X):

...: print train, test

...:

[1 2 3] [0]

[0 2 3] [1]

[0 1 3] [2]

[0 1 2] [3]

#使用KFold实现LeaveOneOtut

In [42]: kf = KFold(n\_splits=len(X))

In [43]: for train, test in kf.split(X):

...: print train, test

...:

[1 2 3] [0]

[0 2 3] [1]

[0 1 3] [2]

[0 1 2] [3]

### 6.LeavePOut

这个也是KFold 的一个特例，用KFold 实现起来稍麻烦些，跟LeaveOneOut 也很像。

In [44]: from sklearn.model\_selection import LeavePOut

In [45]: X = np.ones(4)

In [46]: lpo = LeavePOut(p=2)

In [47]: for train, test in lpo.split(X):

...: print train, test

...:

[2 3] [0 1]

[1 3] [0 2]

[1 2] [0 3]

[0 3] [1 2]

[0 2] [1 3]

[0 1] [2 3]

### 7.ShuffleSplit

ShuffleSplit 咋一看用法跟LeavePOut 很像，其实两者完全不一样，LeavePOut 是使得数据集经过数次分割后，所有的测试集出现的元素的集合即是完整的数据集，即无放回的抽样，而ShuffleSplit 则是有放回的抽样，只能说经过一个足够大的抽样次数后，保证测试集出现了完成的数据集的倍数。

In [48]: from sklearn.model\_selection import ShuffleSplit

In [49]: X = np.arange(5)

In [50]: ss = ShuffleSplit(n\_splits=3, test\_size=.25, random\_state=0)

In [51]: for train\_index, test\_index in ss.split(X):

...: print train\_index, test\_index

...:

[1 3 4] [2 0]

[1 4 3] [0 2]

[4 0 2] [1 3]

### 8.StratifiedKFold

这个就比较好玩了，通过指定分组，对测试集进行无放回抽样。

分层采样交叉切分，确保训练集，测试集中各类别样本的比例与原始数据集中相同

In [52]: from sklearn.model\_selection import StratifiedKFold

In [53]: X = np.ones(10)

In [54]: y = [0,0,0,0,1,1,1,1,1,1]

In [55]: skf = StratifiedKFold(n\_splits=3)

In [56]: for train, test in skf.split(X,y):

...: print train, test

...:

[2 3 6 7 8 9] [0 1 4 5]

[0 1 3 4 5 8 9] [2 6 7]

[0 1 2 4 5 6 7] [3 8 9]

### 9.GroupKFold

这个跟StratifiedKFold 比较像，不过测试集是按照一定分组进行打乱的，即先分堆，然后把这些堆打乱，每个堆里的顺序还是固定不变的。

In [57]: from sklearn.model\_selection import GroupKFold

In [58]: X = [.1, .2, 2.2, 2.4, 2.3, 4.55, 5.8, 8.8, 9, 10]

In [59]: y = ['a','b','b','b','c','c','c','d','d','d']

In [60]: groups = [1,1,1,2,2,2,3,3,3,3]

In [61]: gkf = GroupKFold(n\_splits=3)

In [62]: for train, test in gkf.split(X,y,groups=groups):

...: print train, test

...:

[0 1 2 3 4 5] [6 7 8 9]

[0 1 2 6 7 8 9] [3 4 5]

[3 4 5 6 7 8 9] [0 1 2]

### 10.LeaveOneGroupOut

这个是在GroupKFold 上的基础上混乱度又减小了，按照给定的分组方式将测试集分割下来。

In [63]: from sklearn.model\_selection import LeaveOneGroupOut

In [64]: X = [1, 5, 10, 50, 60, 70, 80]

In [65]: y = [0, 1, 1, 2, 2, 2, 2]

In [66]: groups = [1, 1, 2, 2, 3, 3, 3]

In [67]: logo = LeaveOneGroupOut()

In [68]: for train, test in logo.split(X, y, groups=groups):

...: print train, test

...:

[2 3 4 5 6] [0 1]

[0 1 4 5 6] [2 3]

[0 1 2 3] [4 5 6]

### 11.LeavePGroupsOut

这个没啥可说的，跟上面那个一样，只是一个是单组，一个是多组

from sklearn.model\_selection import LeavePGroupsOut

X = np.arange(6)

y = [1, 1, 1, 2, 2, 2]

groups = [1, 1, 2, 2, 3, 3]

lpgo = LeavePGroupsOut(n\_groups=2)

for train, test in lpgo.split(X, y, groups=groups):

print train, test

[4 5] [0 1 2 3]

[2 3] [0 1 4 5]

[0 1] [2 3 4 5]

### 12.GroupShuffleSplit

这个是有放回抽样

In [75]: from sklearn.model\_selection import GroupShuffleSplit

In [76]: X = [.1, .2, 2.2, 2.4, 2.3, 4.55, 5.8, .001]

In [77]: y = ['a', 'b','b', 'b', 'c','c', 'c', 'a']

In [78]: groups = [1,1,2,2,3,3,4,4]

In [79]: gss = GroupShuffleSplit(n\_splits=4, test\_size=.5, random\_state=0)

In [80]: for train, test in gss.split(X, y, groups=groups):

...: print train, test

...:

[0 1 2 3] [4 5 6 7]

[2 3 6 7] [0 1 4 5]

[2 3 4 5] [0 1 6 7]

[4 5 6 7] [0 1 2 3]

### 13.TimeSeriesSplit

针对时间序列的处理，防止未来数据的使用，分割时是将数据进行从前到后切割（这个说法其实不太恰当，因为切割是延续性的。。）

In [81]: from sklearn.model\_selection import TimeSeriesSplit

In [82]: X = np.array([[1,2],[3,4],[1,2],[3,4],[1,2],[3,4]])

In [83]: tscv = TimeSeriesSplit(n\_splits=3)

In [84]: for train, test in tscv.split(X):

...: print train, test

...:

[0 1 2] [3]

[0 1 2 3] [4]

[0 1 2 3 4] [5]

## 模型评估

对于数据集D，

回归任务最常用的性能度量是“均方误差”(mean squared error)



f(x)为模型函数，即通常说的h(x)

更一般的，均方误差描述为：



### 错误率与精度

分类错误率定义为：



精度定义为：



符号是成立为1，不成立为0.

更一般的，对于数据分布D和概率密度p(.)，错位率和精度分别描述为：





### 查准率与查全率与F1、Fβ

对于二分类问题：

对于数据模型验证存在以下组合：

真正例TP、假正例FP、真反例TN、假反例FN

查准率precision P=TP/(TP+FP)

查全率recall R=TP/(TP+FN) 又叫灵敏度**Sensitivity，也叫真正例率TPR（true positive rate）**

**False Negative Rate** (FNR，假负例率)  ，由混淆矩阵可以看出该指标的着眼点在于正例，意为有多少正例被错判成了负例。  
  
**True Negative Rate** (TNR，真负例率) =，又称Specificity(特异性)。Specificity衡量的是所有的负例中有多少是被正确分类了，由于类别不平衡问题中通常关注正例能否正确被识别，Specificity高则FP低，意味着很少将负例错判为正例，即该分类器对正例的判别具有“特异性”，在预测为正例的样本中很少有负例混入。  
  
**False Positive Rate** (FPR，假正例率) =  由混淆矩阵可以看出该指标的着眼点在于负例，意为有多少负例被错判成了正例。在ROC曲线中分别以TPR和FPR作为纵、横轴作图，显示出一种正例与负例之间的“博弈”，

查准率和查全率是一对矛盾，

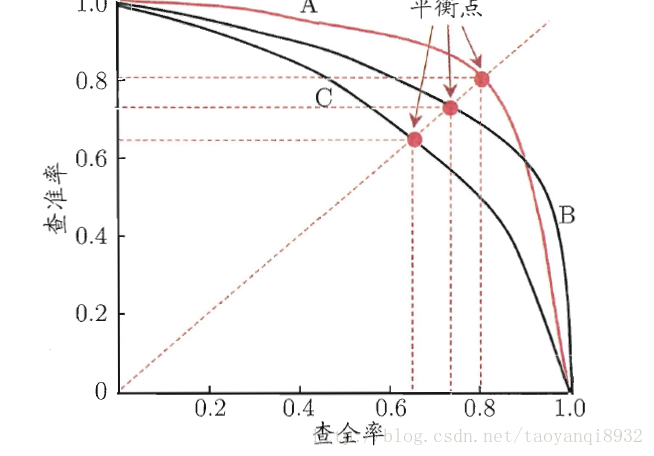
要想查的准，必然查不全（宁缺勿滥）

要想查全，必然将牺牲查准率（宁肯错杀一千，不能放过一个）

要根据实际情况来判断是需要查准还是需要查全。

在很多情况下，我们对机器预测结果进行排序，排在前面的为最可能正例样本，最后面的为最不可能为正例的样本，按此顺序，逐个把样本作为正例进行预测，则么次计算当前的查全率、查准率，以查全率为横轴，查准率为纵轴，则随着将结果判为正例的越多，查准率不断下降，查全率不断上升。

这就是P-R曲线。P为查准率，R为查全率。



在查全率上升的同时,查全率下降幅度较小，则说明模型性能较好。也就是曲线下降的越慢性能越好。

如果一个学习器的P-R曲线完全包住另一个，则可断言前者优于后者。

如果两条曲线发生了交叉，则各有优劣。但要整体比较这两条曲线的优劣，有两种方法：

BEP法，即平衡点法。分别找到两条曲线中P=R的点，该点即为平衡点，谁的平衡点值大，则可以概略的认为该学习器较好。此方法过于简化。

F1度量



TP为真正例

对于一些应用，由于对查准率P和查全率R的重视程度不同，则可以采取对F1加权的方式进行度量-- Fβ



显然，当β>1时，查全率R有更大影响，当β<1时， 查准率P有更大影响，等于1时为F1。

对于多个二分类混淆矩阵，如进行多次训练/测试，得到多个结果

或者多分类任务，每两两类别的组合得到一个结果，则综合衡量有两种方法：

一种是先各自计算P和R，然后求出平均和平均，然后进行计算。

此时称为宏查全率（macro-R）和宏查准率（macro-P），计算出的F1为宏F1 (macro-F1)

一种是计算平均TP、TP、TN、FN，然后通过、、、来计算P和R，

此时P称为‘微查准率’（micro-P）

R称为‘微查全率’（micro-R）

通过P和R计算F1,称为‘微F1’(micro-F1)

### ROC与AUC

<https://www.cnblogs.com/massquantity/p/8592091.html>

曲线上升的越快，性能越好。

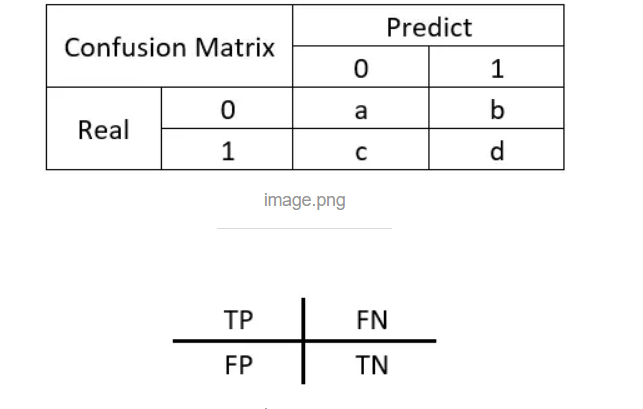
## 准确率、精确率、召回率、F1值、Roc曲线、AUC

### 混淆矩阵 confusion-matrix

* TP(True Positive): 真实为0，预测也为0
* FN(False Negative): 真实为0，预测为1
* FP(False Positive): 真实为1，预测为0
* TN(True Negative): 真实为0，预测也为0
* 混淆矩阵的API

from sklearn.metrics import confusion\_matrix

confusion\_matrix = confusion\_matrix(y\_test, y\_predict)



### 分类准确率 accuracy

* 所有样本中被预测正确的样本的比率
* 分类模型总体判断的准确率(包括了所有class的总体准确率)
* 准确率的API:

from sklearn.metrics import accuracy

accuracy = accuracy\_score(y\_test, y\_predict)

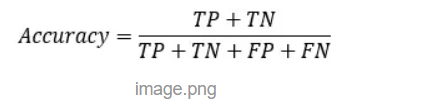


image.png

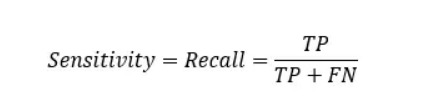
### 精确率Precision

* 预测为正类0的准确率  
  TP / ( TP + FP )

from sklearn.metrics import precision\_score

precision = precision\_score(y\_test, y\_predict)

### 召回率 recall

* 真实为0的准确率
* 
* 真实为1的准确率  
  Recall = TN/(TN+FP)
* 召回率API:

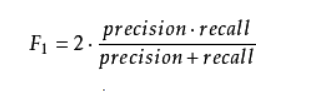
from sklearn.metrics import recall\_score

recall = recall\_score(y\_test, y\_predict)

#recall得到的是一个list，是每一类的召回率

### F1值

* 用来衡量二分类模型精确度的一种指标。它同时兼顾了分类模型的[准确率](https://baike.baidu.com/item/%E5%87%86%E7%A1%AE%E7%8E%87)和[召回率](https://baike.baidu.com/item/%E5%8F%AC%E5%9B%9E%E7%8E%87)。F1分数可以看作是模型[准确率](https://baike.baidu.com/item/%E5%87%86%E7%A1%AE%E7%8E%87)和[召回率](https://baike.baidu.com/item/%E5%8F%AC%E5%9B%9E%E7%8E%87)的一种加权平均，它的最大值是1，最小值是0。



from sklearn.metrics import f1\_score

f1\_score(y\_test, y\_predict)

### Roc曲线、AUC

#### TPR FPR

* + 样本中的真实正例类别总数即TP+FN  
    TPR即True Positive Rate，TPR = TP/(TP+FN)。
  + **TPR：真实的正例0中，被预测为正例的比例**
  + 样本中的真实反例类别总数为FP+TN  
    FPR即False Positive Rate，FPR=FP/(TN+FP)。
  + **FPR：真实的反例1中，被预测为正例的比例**
  + 理想分类器TPR=1，FPR=0

#### 截断点thresholds

机器学习算法对test样本进行预测后，可以输出各test样本对某个类别的相似度概率。比如t1是P类别的概率为0.3，一般我们认为概率低于0.5，t1就属于类别N。这里的0.5，就是”截断点”。

**总结一下，对于计算ROC，最重要的三个概念就是TPR, FPR, 截断点。**

#### ROC曲线

* ROC曲线越接近左上角，代表模型越好，即ACU接近1

from sklearn.metrics import roc\_auc\_score, auc

import matplotlib.pyplot as plt

y\_predict = model.predict(x\_test)

y\_probs = model.predict\_proba(x\_test) #模型的预测得分

fpr, tpr, thresholds = metrics.roc\_curve(y\_test,y\_probs)

roc\_auc = auc(fpr, tpr) #auc为Roc曲线下的面积

#开始画ROC曲线

plt.plot(fpr, tpr, 'b',label='AUC = %0.2f'% roc\_auc)

plt.legend(loc='lower right')

plt.plot([0,1],[0,1],'r--')

plt.xlim([-0.1,1.1])

plt.ylim([-0.1,1.1])

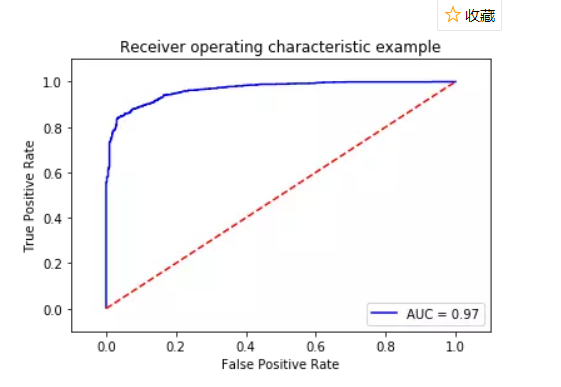
plt.xlabel('False Positive Rate') #横坐标是fpr

plt.ylabel('True Positive Rate') #纵坐标是tpr

plt.title('Receiver operating characteristic example')

plt.show()

运行结果如下图所示：



参考资料：  
1.混淆矩阵(Confusion Matrix)  
<https://www.jianshu.com/p/0fc8a0b784f1>  
2.ROC与AUC的定义与使用详解  
<https://blog.csdn.net/shenxiaoming77/article/details/72627882>