## 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）**

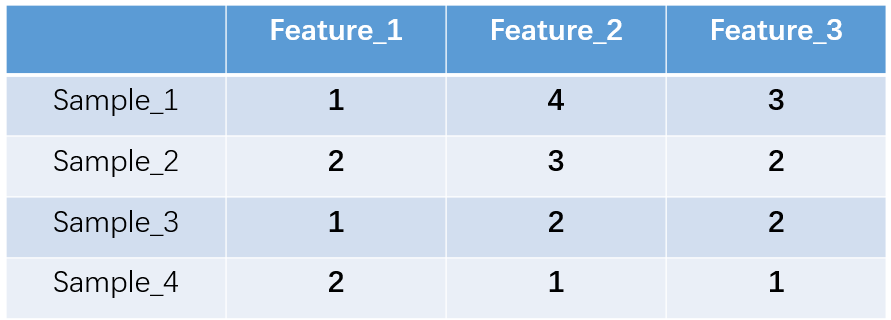
1. **不过 Mini-batch gradient descent 不能保证很好的收敛性，learning rate 如果选择的太小，收敛速度会很慢，如果太大，loss function 就会在极小值处不停地震荡甚至偏离。（有一种措施是先设定大一点的学习率，当两次迭代之间的变化低于某个阈值后，就减小 learning rate，不过这个阈值的设定需要提前写好，这样的话就不能够适应数据集的特点。）**对于非凸函数，还要避免陷于局部极小值处，或者鞍点处，因为鞍点周围的error是一样的，所有维度的梯度都接近于0，SGD 很容易被困在这里。（**会在鞍点或者局部最小点震荡跳动，因为在此点处，如果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

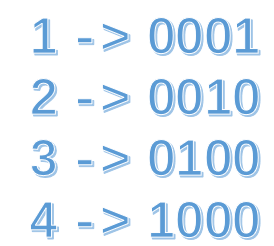
#### 1.onehot编码

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）。

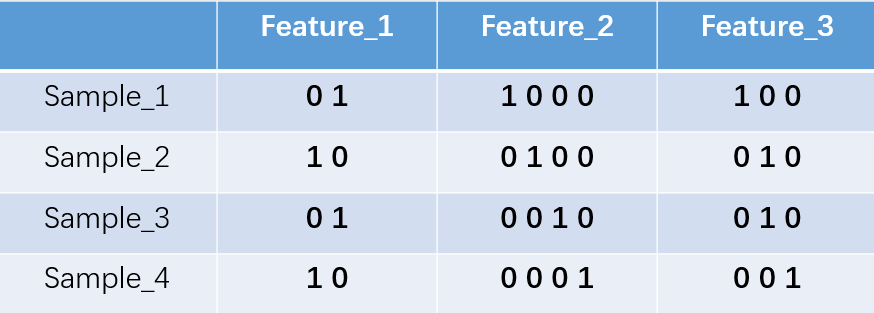


上图中我们已经对每个特征进行了普通的数字编码：我们的feature\_1有两种可能的取值，比如是男/女，这里男用1表示，女用2表示。那么one-hot编码是怎么搞的呢？我们再拿feature\_2来说明：

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

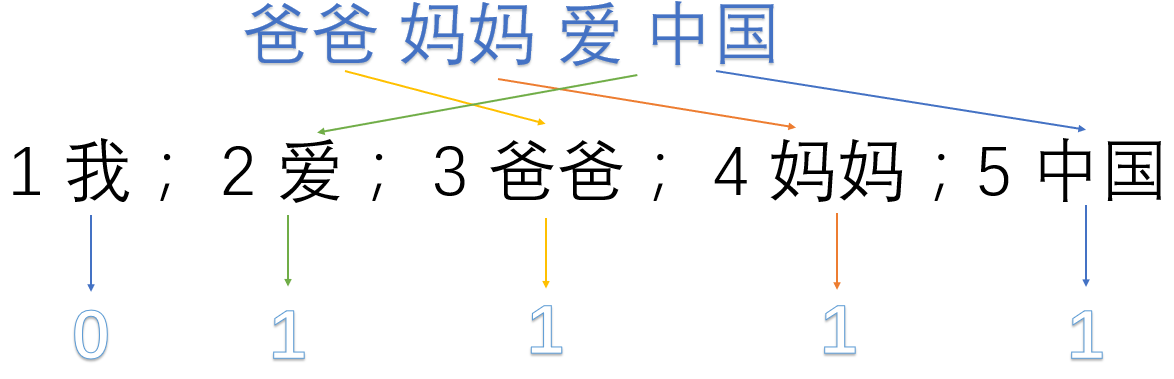
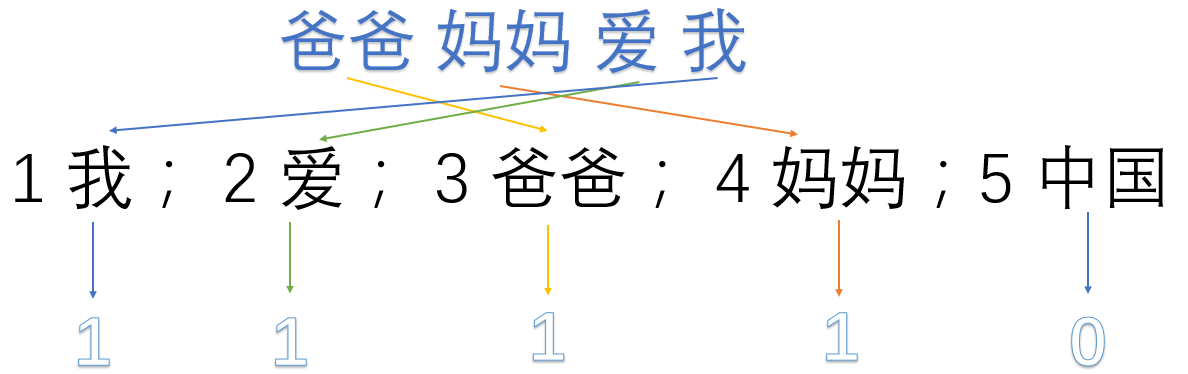
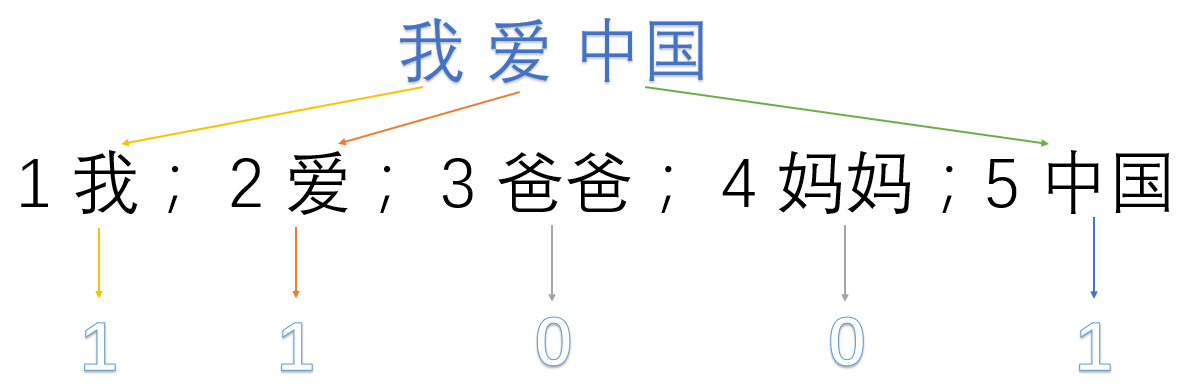
　　　　爸爸妈妈爱我

　　　　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)

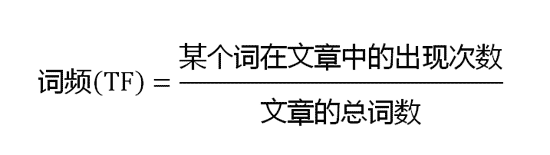
输入中，每一行为一个特征，每一列为一个输入。

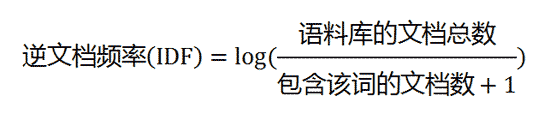
### 2. TF-IDF

#### 基本内容

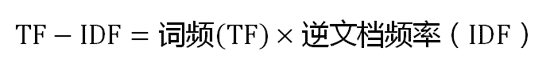
　　IF-IDF是信息检索（IR）中最常用的一种文本表示法。算法的思想也很简单，就是统计每个词出现的词频（TF），然后再为其附上一个权值参数（IDF）。举个例子：

　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

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.processing.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.],

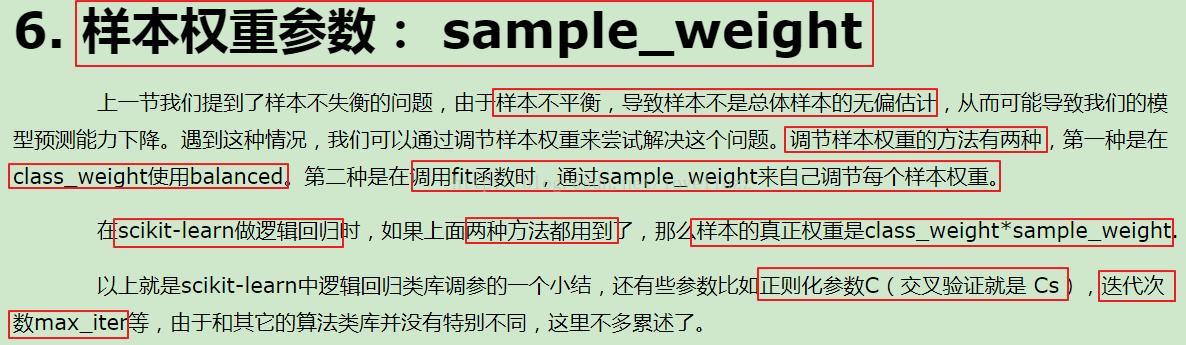
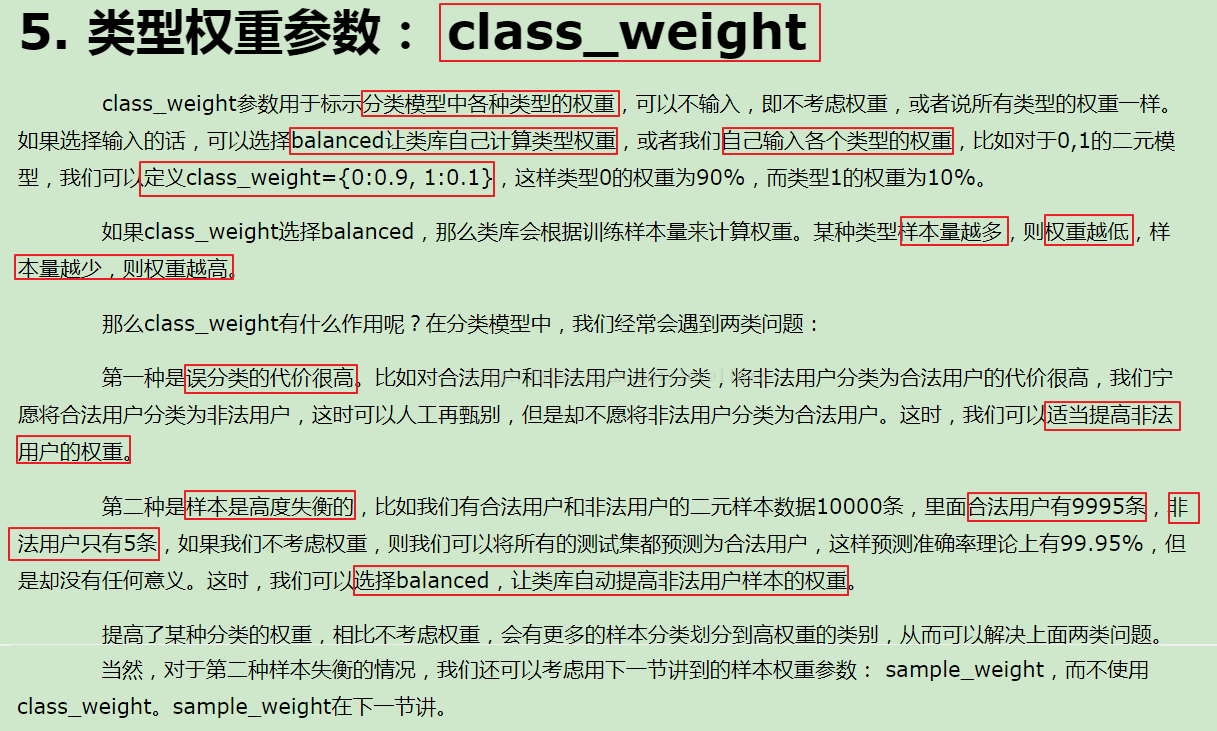
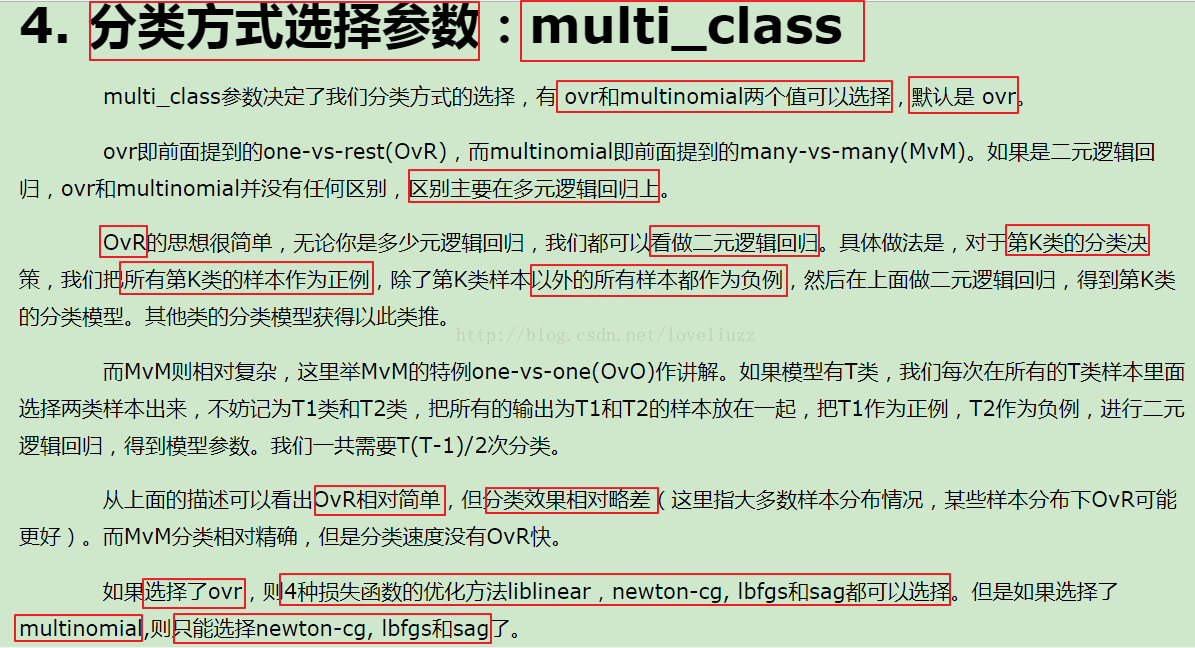
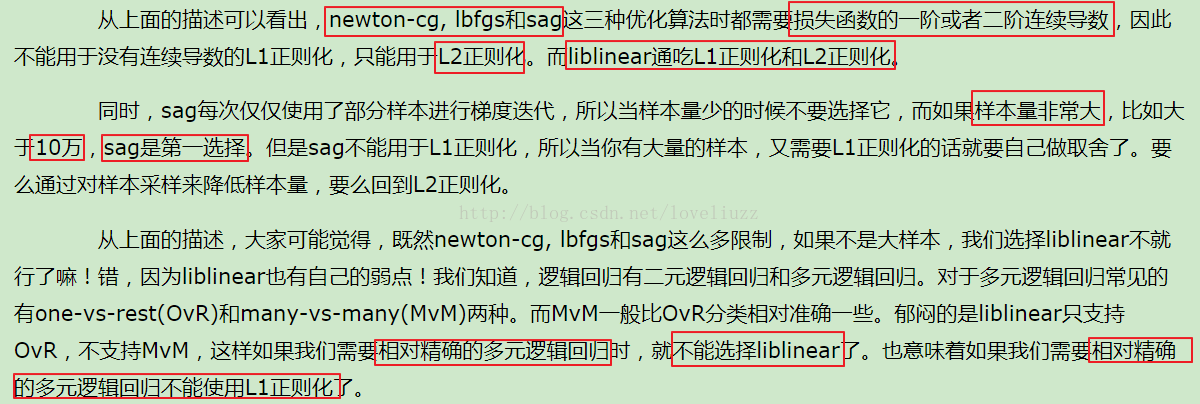
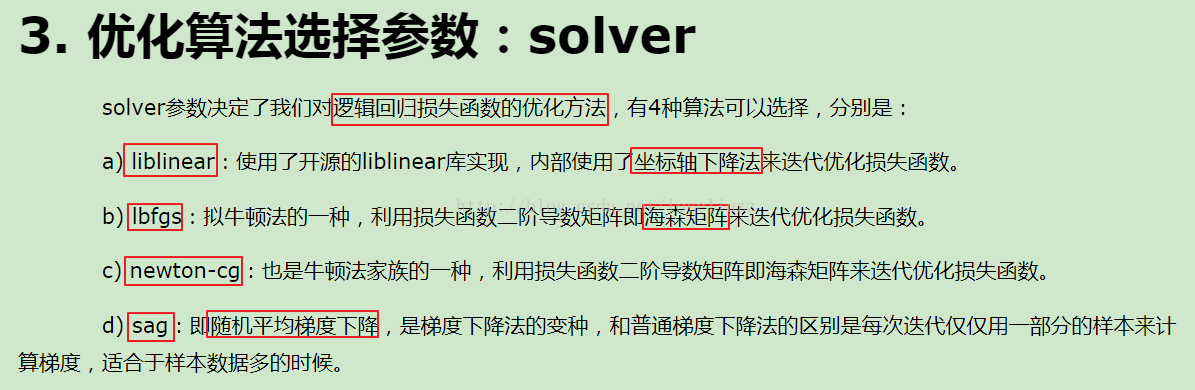
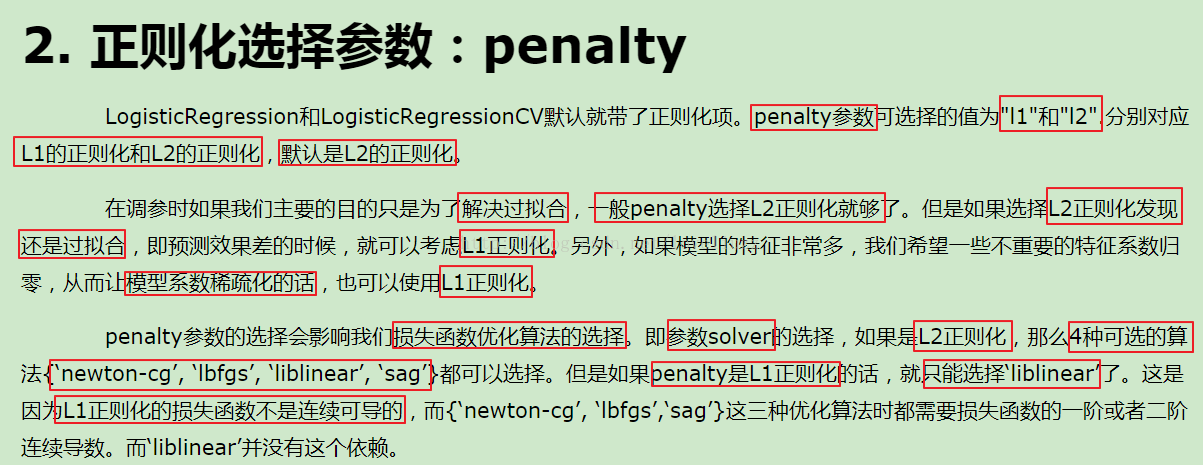
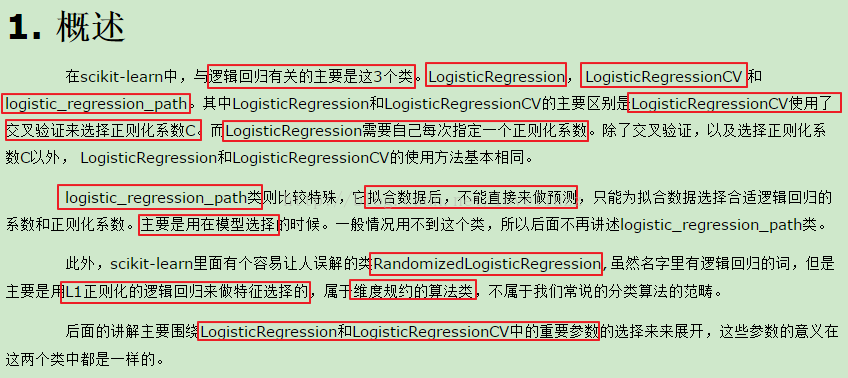
[ 1., 2., 3., 6.],

[ 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：

## 模型评估方法及sklearn实现（选训练集和测试集）

### 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）；

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。

## 模型性能度量

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P=TP/(TP+FP)

查全率R=TP/(TP+FN)

查准率和查全率是一对矛盾，

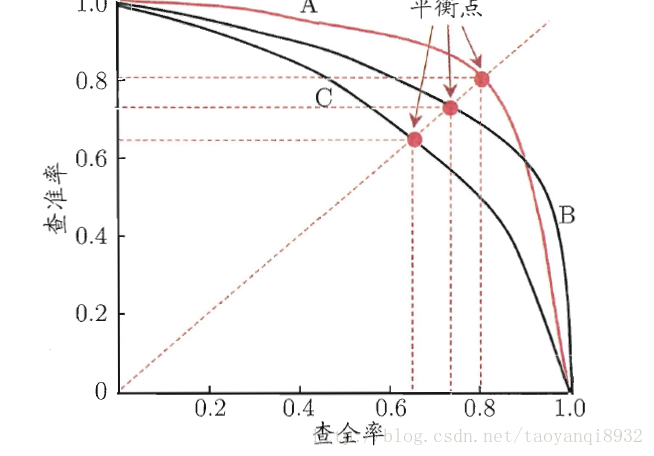
要想查的准，必然查不全（宁缺勿滥）

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。也就是曲线越平缓越好。

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