

## 第一章

### 1. 偏差与方差：

模型在训练集上的误差来源主要来自于偏差，在测试集上误差来源主要来自于方差。

欠拟合是一种高偏差的情况。过拟合是一种低偏差，高方差的情况。

- 偏差：预计值的期望与真实值之间的差距；
- 方差：预测值的离散程度，也就是离其期望值的距离。

	训练集	测试集	偏差	方差
欠拟合	80%	79%	20%	1%
过拟合	99%	80%	1%	19%

注意这个表格的数量关系：所说的“偏差、方差”表示的是模型在测试集上的预测结果偏离真实值的程度（偏差）和预测值的离散程度（方差）。

因此      测试集误差=偏差+方差      训练集误差=偏差

（如第一行：） 测试集误差=1-79%=21%

其中偏差=1-80%=20%

方差=21%-20%=1%

### 2. 验证集与交叉验证：

#### 为什么需要验证集

在机器学习中，通常需要评估若干候选模型的表现并从中选择模型。这一过程称为模型选择。  
可供选择的候选模型可以是有着不同超参数的同类模型（验证集的作用就是辅助进行超参数的选择，从而影响模型的选择，注意是超参数的选择而不是可学习参数，可学习参数是通过训练集误差反向传播学习到的）。以神经网络为例，我们可以选择隐藏层的个数，学习率大小和激活函数。为了得到有效的模型，我们通常要在模型选择上下一番功夫。从严格意义上讲，测试集只能在所有超参数和模型参数选定后使用一次。不可以使用测试数据选择模型，如调参。由于无法从训练误差估计泛化误差，因此也不应只依赖训练数据选择模型。鉴于此，我们可以预留一部分在训练数据集和测试数据集以外的数据来进行模型选择。这部分数据被称为验证数据集，简称验证集。

#### k 折交叉验证

由于验证数据集不参与模型训练，当训练数据不够用时，预留大量的验证数据显得太奢侈。一种改善的方法是 K 折交叉验证。在 K 折交叉验证中，我们把原始训练数据集分割成 K 个不重合的子数据集，然后我们做 K 次模型训练和验证。每一次，我们使用一个子数据集

验证模型，并使用其它  $K-1$  个子数据集来训练模型。在这  $K$  次训练和验证中，每次用来验证模型的子数据集都不同。最后，我们对这  $K$  次训练误差和验证误差分别求平均。

(对于每一组超参数，都进行  $k$  折交叉验证，每一组超参数会得到  $k$  个正确率得分，然后取平均得到这组超参数对应的平均性能得分，然后对于不同组超参数，选出平均性能得分最高的那种作为最佳超参数组合)

### 3. 集成学习

在机器学习的有监督学习算法中，我们的目标是学习出一个稳定的且在各个方面表现都较好的模型，但实际情况往往不这么理想，有时我们只能得到多个有偏好的模型（弱监督模型，在某些方面表现的比较好）。集成学习就是组合这里的多个弱监督模型以期得到一个更好更全面的强监督模型。集成学习潜在的思想是即便某一个弱分类器得到了错误的预测，其他的弱分类器也可以将错误纠正回来。集成方法是将几种机器学习技术组合成一个预测模型的元算法，以达到减小方差、偏差或改进预测的效果。

### 4. 自助法 (bootstrap)

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。

### 5. 回归问题评价指标：

R-square：

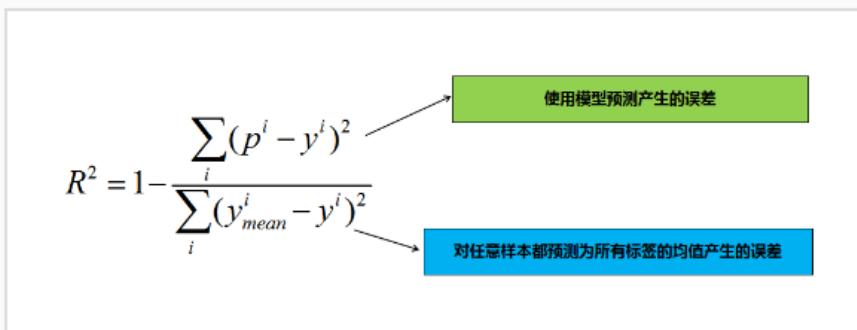
## R-Squared

上面的几种衡量标准针对不同的模型会有不同的值。比如说预测房价 那么误差单位就是万元。数子可能是 3, 4, 5 之类的。那么预测身高就可能是 0.1, 0.6 之类的。没有什么可读性，到底多少才算好呢？不知道，那要根据模型的应用场景来。看看分类算法的衡量标准就是正确率，而正确率又在 0~1 之间，最高百分之百。最低 0。如果是负数，则考虑非线性相关。很直观，而且不同模型一样的。那么线性回归有没有这样的衡量标准呢？

R-Squared 就是这么一个指标，公式如下：

$$R^2 = 1 - \frac{\sum_i (p^i - y^i)^2}{\sum_i (y_{mean}^i - y^i)^2}$$

其中  $y_{mean}$  表示所有测试样本标签值的均值。为什么这个指标会有刚刚我们提到的性能呢？我们分析下公式：



其实分子表示的是模型预测时产生的误差，分母表示的是对任意样本都预测为所有标签均值时产生的误差，由此可知：

1.  $R^2 \leq 1$ ，当我们的模型不犯任何错误时，取最大值 1。
2. 当我们的模型性能跟基模型性能相同时，取 0。
3. 如果为负数，则说明我们训练出来的模型还不如基准模型，此时，很有可能我们的数据不存在任何线性关系。

R-square 指标越大越好

MAE:

## MAE

MAE (平均绝对误差)，公式如下：

$$\frac{1}{m} \sum_{i=1}^m |y^i - p^i|$$

MSE & RMSE:

### MSE

MSE (Mean Squared Error) 叫做均方误差, 公式如下:

$$\frac{1}{m} \sum_{i=1}^m (y^i - p^i)^2$$

其中  $y^i$  表示第  $i$  个样本的真实标签,  $p^i$  表示模型对第  $i$  个样本的预测标签。线性回归的目的就是让损失函数最小。那么模型训练出来了, 我们在测试集上用损失函数来评估模型就行了。

### RMSE

RMSE (Root Mean Squared Error) 均方根误差, 公式如下:

$$\sqrt{\frac{1}{m} \sum_{i=1}^m (y^i - p^i)^2}$$

RMSE 其实就是 MSE 开个根号。有什么意义呢? 其其实质是一样的。只不过用于数据更好的描述。

相比 MSE 指标, MAE 对噪声数据不敏感, 因为 MAE 是 L1 距离, MSE 是 L2 距离, 对于外点更敏感。

## 6. ROC 与 AUC

### ROC曲线

ROC曲线( Receiver Operating Characteristic Curve )描述的 TPR ( True Positive Rate ) 与 FPR ( False Positive Rate ) 之间关系的曲线。

TPR 与 FPR 的计算公式如下:

$$TPR = \frac{TP}{TP + FN}$$
$$FPR = \frac{FP}{FP + TN}$$

其中 TPR 的计算公式您可能有点眼熟, 没错! 就是召回率的计算公式。也就是说 TPR 就是召回率。所以 TPR 描述的是模型预测 Positive 并且预测正确的数量占真实类别为 Positive 样本的比例。而 FPR 描述的模型预测 Positive 并且预测错了的数量占真实类别为 Negative 样本的比例。

当模型的 TPR 越高 FPR 也会越高, TPR 越低 FPR 也会越低。这与精准率和召回率之间的关系刚好相反。并且, 模型的分类阈值一但改变, 就有一组对应的 TPR 与 FPR。若将

FPR 作为横轴， TPR 作为纵轴，将上面的表格以折线图的形式画出来就是 ROC 曲线。ROC

曲线与横轴所围成的面积越大，模型的分类性能就越高。而 ROC 曲线 的面积称为 AUC。

### AUC

很明显模型的 AUC 越高，模型的二分类性能就越强。AUC 的计算公式如下：

$$AUC = \frac{\sum_{i \in \text{positive class}} rank_i - \frac{M(M+1)}{2}}{M * N}$$

其中 M 为真实类别为 Positive 的样本数量，N 为真实类别为 Negtive 的样本数量。ranki 代表了真实类别为 Positive 的样本点预测概率从小到大排序后，该预测概率排在第几。

举个例子，现有预测概率与真实类别的表格如下所示（其中 0 表示 Negtive， 1 表示 Positive）：

编号	预测概率	真实类别
1	0.1	0
2	0.4	0
3	0.3	1
4	0.8	1

想要得到公式中的 rank，就需要将预测概率从小到大排序，排序后如下：

编号	预测概率	真实类别
1	0.1	0
3	0.3	1
2	0.4	0
4	0.8	1

排序后的表格中，真实类别为 Positive 只有编号为 3 和编号为 4 的数据，并且编号为 3 的数据排在第 2， 编号为 4 的数据排在第 4。所以 rank=[2, 4]。又因表格中真是类别为 Positive 的数据有 2 条， Negtive 的数据有 2 条。因此 M 为2， N 为2。所以根据 AUC 的计算公式可知：

$$AUC = \frac{(2+4)-\frac{2(2+1)}{2}}{2*2} = 0.75。$$

## 第三章

### 1. 线性回归

使用的损失函数：均方根误差 (RMSE)

下列关于线性回归分析中的残差（预测值减去真实值）说法正确的是？

- A、残差均值总是为零
- B、残差均值总是小于零
- C、残差均值总是大于零
- D、以上说法都不对

解释：因为 Loss 函数对  $w$  和  $b$  求导的结果要为 0，把求导结果写出来，如下所示，就得到了残差均值为 0 的证明。

## ● 简单回归模型

**方法一：**求导解方程       $\text{Loss} = \sum_{i=1}^N (y'_i - y_i)^2 = \sum_{i=1}^N (y'_i - (wx_i + b))^2$

对于凸优化问题，求最小化可以通过求导数实现

对  $w$  和  $b$  分别求导令其值为 0：

$$\begin{pmatrix} \sum_{i=1}^N -2x_i(y'_i - (wx_i + b)) \\ \sum_{i=1}^N -2(y'_i - (wx_i + b)) \end{pmatrix} = 0$$

---

### 方法一：求导解方程

$$\begin{pmatrix} \sum_{i=1}^N -2x_i(y'_i - (wx_i + b)) \\ \sum_{i=1}^N -2(y'_i - (wx_i + b)) \end{pmatrix} = 0 \rightarrow \bar{x} = \frac{\sum_{i=1}^N x_i}{N} \quad \bar{y} = \frac{\sum_{i=1}^N y_i}{N}$$

$$w = \frac{\sum_{i=1}^N x_i y_i - N \bar{x} \bar{y}}{\sum_{i=1}^N x_i^2 - N \bar{x}^2}$$

$$b = \frac{\bar{y}(\sum_{i=1}^N x_i^2) - \bar{x}(\sum_{i=1}^N x_i y_i)}{\sum_{i=1}^N x_i^2 - N \bar{x}^2}$$

## 2. 线性回归的解

## 线性回归的正规方程解

对线性回归模型，假设训练集中  $m$  个训练样本，每个训练样本中有  $n$  个特征，可以使用矩阵的表示方法，预测函数可以写为：

$$Y = \theta X$$

其损失函数可以表示为

$$(Y - \theta X)^T (Y - \theta X)$$

其中，标签  $Y$  为  $m \times 1$  的矩阵，训练特征  $X$  为  $m \times (n+1)$  的矩阵，回归系数  $\theta$  为  $(n+1) \times 1$  的矩阵，对  $\theta$  求导，并令其导数等于 0，可以得到  $X^T(Y - \theta X) = 0$ 。所以，最优解为：

$$\theta = (X^T X)^{-1} X^T Y$$

这个就是正规方程解，我们可以通过最优方程解直接求得我们所需要的参数。

## 方法二：梯度下降

使用偏差平方和作为损失函数      求梯度       $\frac{\partial C}{\partial w} = -2x_i(y'_i - (wx_i + b))$       更新       $w := w - \alpha \frac{\partial C}{\partial w}$

$C = (y'_i - y_i)^2 = (y'_i - (wx_i + b))^2$        $\frac{\partial C}{\partial b} = -2(y'_i - (wx_i + b))$        $b := b - \alpha \frac{\partial C}{\partial b}$

## 多元线性回归模型

属性越多越有可能过拟合训练集合，导致模型在测试时表现不佳，怎么办？

正则化 (regularization) : 对模型参数施加 “惩罚”

$$C = (y'_i - y_i)^2 = (y'_i - (w x_i + b))^2 \rightarrow C = (y'_i - (w x_i + b))^2 + \alpha \|w\|_2$$

使  $C$  减小，正则项也会被 “惩罚”

若线性回归方程得到多个解，下面哪些方法能够解决此问题？

- A、获取更多的训练样本
- B、选取样本有效的特征，使样本数量大于特征数
- C、加入正则化项
- D、不考虑偏置项  $b$

线性回归（矩阵表达）：

$$y = w_0x_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n$$

其中  $x_0$  等于 1。

$$Y = \theta X$$

$$\theta = (w_0, w_1, \dots, w_n)$$

$$X = (1, x_1, \dots, x_n)$$

的差异，然后将问题转化为最优化损失函数。既然损失函数是用来衡量真实值与预测值之间的差异那么很多人自然而然的想到了用所有真实值与预测值的差的绝对值来表示损失函数。不过带绝对值的函数不容易求导，所以采用 MSE (均方误差)作为损失函数，公式如下：

$$loss = \frac{1}{m} \sum_{i=1}^m (y^i - p^i)^2$$

### 线性回归的正规方程解

对线性回归模型，假设训练集中  $m$  个训练样本，每个训练样本中有  $n$  个特征，可以使用矩阵的表示方法，预测函数可以写为：

$$Y = \theta X$$

其损失函数可以表示为

$$(Y - \theta X)^T (Y - \theta X)$$

其中，标签  $y$  为  $m \times 1$  的矩阵，训练特征  $x$  为  $m \times (n+1)$  的矩阵，回归系数  $\theta$  为  $(n+1) \times 1$  的矩阵，对  $\theta$  求导，并令其导数等于 0，可以得到  $X^T(Y - \theta X) = 0$ 。所以，最优解为：

$$\theta = (X^T X)^{-1} X^T Y$$

## 逻辑回归

### 1. 概念理解

当一看到“回归”这两个字，可能会认为逻辑回归是一种解决回归问题的算法，然而逻辑回归是通过回归的思想来解决**二分类**问题的算法。

那么问题来了，回归的算法怎样解决分类问题呢？其实很简单，逻辑回归是将样本特征和样本所属类别的概率联系在一起，假设现在已经训练好了一个逻辑回归的模型为  $f(x)$ ，模型的输出是样本  $x$  的标签是 1 的概率，则该模型可以表示， $\hat{p} = f(x)$ 。若得到了样本  $x$  属于标签 1 的概率后，很自然的就能想到当  $\hat{p} > 0.5$  时  $x$  属于标签 1，否则属于标签 0。所以就有

$$\hat{y} = \begin{cases} 0 & \hat{p} < 0.5 \\ 1 & \hat{p} > 0.5 \end{cases}$$

**由于概率是 0 到 1 的实数，所以逻辑回归若只需要计算出样本所属标签的概率就是一种回归算法，若需要计算出样本所属标签，则就是一种二分类算法。**

那么逻辑回归中样本所属标签的概率怎样计算呢？其实和线性回归有关系，学习了线性回归的同学肯定知道线性回归无非就是训练出一组参数  $W^T$  和  $b$  来拟合样本数据，线性回归的输出为  $\hat{y} = W^T x + b$ 。不过  $\hat{y}$  的值域是  $(-\infty, +\infty)$ ，如果能够将值域为  $(-\infty, +\infty)$  的实数转换成  $(0, 1)$  的概率值的话问题就解决了。要解决这个问题很自然地就能想到将线性回归的输出作为输入，输入到另一个函数中，这个函数能够进行转换工作，假设函数为  $\sigma$ ，转换后的概率为  $\hat{p}$ ，则逻辑回归在预测时可以看成  $\hat{p} = \sigma(W^T x + b)$ 。 $\sigma$  其实就是接下来要介绍的 sigmoid 函数。

根据上一关中所学习到的知识，我们已经知道了逻辑回归计算出的样本所属类别的概率  $\hat{p} = \sigma(W^T x + b)$ ，样本所属列表的判定条件为

$$\hat{y} = \begin{cases} 0 & \hat{p} < 0.5 \\ 1 & \hat{p} > 0.5 \end{cases}$$

**逻辑回归本质上就是通过回归模型，得到一个负无穷到正无穷的预测值，在通过 sigmoid 函数映射到  $(0,1)$  的值，这个值就表示概率意义，在通过设定一个阈值，如 0.5，**

之前得到的正样本预测概率大于 0.5 就分类为正样本，否则分类为负样本，这就是逻辑回归解决二分类问题的思路。

## 2. 损失函数

当然逻辑回归的损失函数不仅仅与  $\hat{p}$  有关，它还与真实类别有关。假设现在有两种情况，情况A：现在有个样本的真实类别是 0，但是模型预测出来该样本是类别 1 的概率是 0.7（也就是说类别 0 的概率为 0.3）；情况B：现在有个样本的真实类别是 0，但是模型预测出来该样本是类别 1 的概率是 0.6（也就是说类别 0 的概率为 0.4）；请你思考 2 秒钟，AB两种情况哪种情况的误差更大？很显然，情况A的误差更大！因为情况A中模型认为样本是类别 0 的可能性只有 30%，而B有 40%。

假设现在又有两种情况，**情况A**：现在有个样本的真实类别是 0，但是模型预测出来该样本是类别 1 的概率是 0.7（也就是说类别 0 的概率为 0.3）；**情况B**：现在有个样本的真实类别是 1，但是模型预测出来该样本是类别 1 的概率是 0.3（也就是说类别 0 的概率为 0.7）；请你再思考 2 秒钟，AB两种情况哪种情况的误差更大？很显然，**一样大！**

所以逻辑回归的损失函数如下，其中  $cost$  表示损失函数的值， $y$  表示样本的真实类别：

$$cost = -y \log(\hat{p}) - (1 - y) \log(1 - \hat{p})$$

$cost = -y \log(\hat{p}) - (1 - y) \log(1 - \hat{p})$  是一个样本的损失计算公式，但是在一般情况下需要计算的是  $m$  条样本数据的平均损失值，所以损失函数的最终形态如下，其中  $m$  表示数据集中样本的数量， $i$  表示数据集中第  $i$  个样本：

$$cost = -\frac{1}{m} \sum_{i=0}^m y^{(i)} \log(\hat{p}^{(i)}) - (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})$$

知道了逻辑回归的损失函数之后，逻辑回归的训练流程就很明显了，就是寻找一组合适的  $W^T$  和  $b$ ，使得损失值最小。找到这组参数后模型就确定下来了。

逻辑回归的优点有哪些?

- A、需要事先对数据的分布做假设
- B、可以得到“类别”的真正的概率预测
- C、可以用闭式解求解
- D、可以用现有的数值优化算法求解

sigmoid 函数(对数几率函数)相对于单位阶跃函数有哪些好处?

- A、sigmoid函数可微分
- B、sigmoid函数处处连续
- C、sigmoid函数不是单调的
- D、sigmoid函数最多计算二阶导

## 梯度下降

### 1. 简单代码实现:

```
def gradient_descent(initial_theta, eta=0.05, n_iters=1000, epsilon=1e-8):
    """
    梯度下降
    :param initial_theta: 参数初始值, 类型为 float
    :param eta: 学习率, 类型为 float
    :param n_iters: 训练轮数, 类型为 int
    :param epsilon: 容忍误差范围, 类型为 float
    :return: 训练后得到的参数
    """

    # 请在此添加实现代码 #
    #***** Begin *****#
    theta = initial_theta
    for i in range(n_iters):
        new_theta = theta - eta * 2 * (theta - 3)
        if abs(new_theta - theta) < epsilon: # 收敛了
            break;
        theta = new_theta
    return new_theta
```

## 逻辑回归解决实际问题

### 构建逻辑回归模型

由数据集可以知道，每一个样本有 30 个特征和 1 个标签，而我们要做的事就是通过这 30 个特征来分析细胞是良性还是恶性(其中标签  $y=0$  表示是良性， $y=1$  表示是恶性)。逻辑回归算法正好是一个二分类模型，我们可以构建一个逻辑回归模型，来对癌细胞进行识别。模型如下：

$$z = b + w_1x_1 + w_2x_2 + \dots + w_nx_n$$
$$y = \frac{1}{1 + e^{-z}}$$

其中  $x_i$  表示第  $i$  个特征， $w_i$  表示第  $i$  个特征对应的权重， $b$  表示偏置。

为了方便，我们稍微将模型进行变换：

$$z = w_0x_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n$$

其中  $x_0$  等于 1。

$$Z = \theta \cdot X$$

$$\theta = (w_0, w_1, \dots, w_n)$$

$$X = (1, x_1, \dots, x_n)$$

$$y = \frac{1}{1 + e^{-\theta \cdot X}}$$

我们将一个样本输入模型，如果预测值大于等于 0.5 则判定为 1 类别，如果小于 0.5 则判定为 0 类别。

在上一节中，我们知道要使用梯度下降算法首先要知道损失函数对参数的梯度，即损失函数对每个参数的偏导，求解步骤如下：

$$\text{loss} = -y \ln a - (1 - y) \ln(1 - a)$$
$$\frac{\partial \text{loss}}{\partial w} = \frac{\partial \text{loss}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$
$$\frac{\partial \text{loss}}{\partial a} = -\frac{y}{a} - \frac{1-y}{1-a}(-1) = \frac{a-y}{a(1-a)}$$
$$\frac{\partial a}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2} = a(1-a)$$
$$\frac{\partial z}{\partial w} = x$$
$$\frac{\partial \text{loss}}{\partial w} = (a-y)x$$

其中  $a$  为预测值， $y$  为真实值。

于是，在逻辑回归中的梯度下降公式如下：

$$w_i = w_i - \eta(a - y)x_i$$

代码实现：

```
def fit(x,y,eta=1e-3,n_iters=10000):
    ...
    训练逻辑回归模型
    :param x: 训练集特征数据, 类型为 ndarray
    :param y: 训练集标签, 类型为 ndarray
    :param eta: 学习率, 类型为 float
    :param n_iters: 训练轮数, 类型为 int
    :return: 模型参数, 类型为 ndarray
    ...

    # 请在此添加实现代码 #
    #***** Begin ****#
    theta=np.zeros(x.shape[1])
    for i in range(n_iters):
        z=x@theta
        a=sigmoid(z)
        grad=x.T @ (a-y)
        theta-=eta*grad
    return theta
    #***** End *****#
```

### 使用逻辑回归进行手写数字识别(sklearn)

1. sklearn 中的逻辑回归已经实现了 OVR 多分类，因此可以直接用来进行多分类。注意为了提高准确率，训练数据和测试数据都要先进行零均值标准化，注意标准化是怎么用 sklearn 实现的。

```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

def digit_predict(train_image, train_label, test_image):
    ...
    实现功能：训练模型并输出预测结果
    :param train_sample: 包含多条训练样本的样本集, 类型为
ndarray, shape 为[-1, 8, 8]
    :param train_label: 包含多条训练样本标签的标签集, 类型为
ndarray
    :param test_sample: 包含多条测试样本的测试集, 类型为 ndarray
```

```

:return: test_sample 对应的预测标签
''

#***** Begin ****#
train_data=train_image.reshape(-1,8*8)
test_data=test_image.reshape(-1,8*8)
#标准化数据
scaler=StandardScaler()
train_data=scaler.fit_transform(train_data)
test_data=scaler.transform(test_data)

lr=LogisticRegression(solver='lbfgs',max_iter=10,C=10)
lr.fit(train_data,train_label)
res=lr.predict(test_data)
return res;
#***** End ****#

```

## 多层感知机

### 1. 简单代码实现：

比如说，输入的特征值分别是青绿，蜷缩，浊响对应特征向量为 (0,0,0)。感知机模型会将每一个特征值  $x_i$  乘以一个对应的权重  $w_i$ ，再加上一个偏置  $b$ ，所得到的值如果大于等于 0，则判断为 +1 类别，即为好瓜，如果得到的值小于 0，则判断为 -1 类别，即不是好瓜。数学模型如下：

$$f(x) = \text{sign}(w_1x_1 + w_2x_2 + \dots + w_nx_n + b)$$

$$\text{sign} = \begin{cases} -1 & x < 0 \\ +1 & x \geq 0 \end{cases}$$

其中  $x_i$  为第  $i$  个特征值， $w_i$  为第  $i$  个特征所对应的权重， $b$  为偏置。

感知机采用的损失函数最初采用的是误分类点到决策边界的距离：

$$\frac{|w \cdot x + b|}{\|w\|}$$

而对于误分类的数据 $(x_i, y_i)$ 来说：

$$\begin{cases} y_i = +1 & w \cdot x_i + b < 0 \\ y_i = -1 & w \cdot x_i + b \geq 0 \end{cases}$$

所以， $|w \cdot x_i + b| = -y_i(w \cdot x_i + b)$

又因为：

- $\frac{1}{\|w\|}$  不影响  $-y(w \cdot x + b)$  正负的判断，我们只需要判断  $-y(w \cdot x + b)$  的正负来判断分类的正确与否。  
所以  $1/\|w\|$  对感知机学习算法的中间过程可有可无。
- $\frac{1}{\|w\|}$  不影响感知机学习算法的最终结果。因为感知机学习算法最终的终止条件是所有的输入都被正确分类，即不存在误分类的点。则此时损失函数为 0。对应于  $\frac{-y(w \cdot x + b)}{\|w\|}$ ，即分子为 0。则可以看出  $\frac{1}{\|w\|}$  对最终结果也无影响。所以最后采用的损失函数为：

$$L(w, b) = -\sum_{x_i \in M} y_i(w \cdot x_i + b)$$

其中  $M$  为误分类点的集合

感知机只针对误分类的点对参数进行更新，算法流程如下：

1. 选取初始值  $w_0, b_0$ ；
2. 在训练集中选取数据  $x_i, y_i$ ；
3. 如果  $y_i(w \cdot x_i + b) \leq 0$ ，即这个点为误分类点，则进行如下更新( $\eta$ 为学习率，为0到1之间的值)：  
 $w = w - \eta \frac{\partial l(w, b)}{\partial w} = w + \eta y_i x_i$   
 $b = b - \eta \frac{\partial l(w, b)}{\partial b} = b + \eta y_i$
4. 重复 2, 3 直到训练集中没有误分类点

```
import numpy as np
#构建感知机算法
class Perceptron(object):
    def __init__(self, learning_rate = 0.01, max_iter = 200):
        self.lr = learning_rate
        self.max_iter = max_iter
    def fit(self, data, label):
        ...
        input: data(ndarray): 训练数据特征
        label(ndarray): 训练数据标签
        output:w(ndarray): 训练好的权重
        b(ndarray): 训练好的偏置
        ...
    #编写感知机训练方法， w 为权重， b 为偏置
```

```

self.w = np.array([1.]*data.shape[1]) #特征维
self.b = np.array([1.])
***** Begin *****
for i in range(self.max_iter):
    for x_i,y_i in zip(data,label): #关键！拿出一个样本
出来
        if y_i*(self.w @ x_i+ self.b)<=0: #误分类
            self.w+=self.lr * y_i*x_i
            self.b+=self.lr * y_i
***** End *****
def predict(self, data):
    ...
    input: data(ndarray): 测试数据特征
    output: predict(ndarray): 预测标签
    ...
    ***** Begin *****
    y = data @ self.w + self.b
    predict=np.where(y>=0,1,-1)
    ***** End *****
    return predict

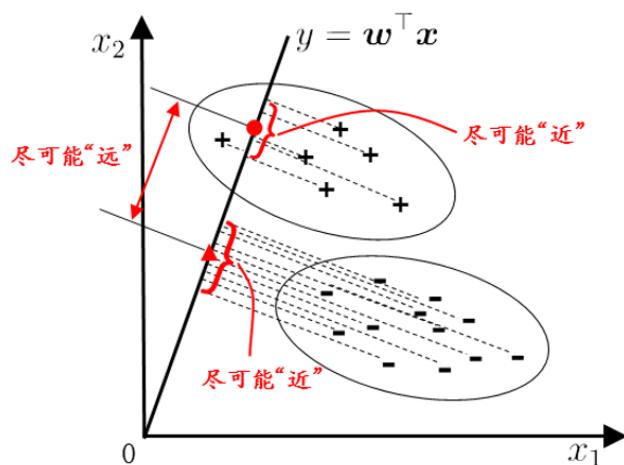
```

## LDA

### 1. 原理

#### 线性判别分析算法思想

LDA 的思想分析非常朴素：给定训练样本集，设法将样本投影到一条直线上，使得同类样本的投影点尽可能接近、异类样本点的投影点尽可能远离。在对新样本进行分类时，将其投影到同样的这条直线上，再根据投影点的位置来确定样本的类别。示意图如下：



用一句话来概括 LDA 思想就是：**投影后类内方差最小，类间方差最大。**

## 二类线性判别分析算法原理

假设我们的数据集  $D = (x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$ , 其中任意样本  $x_i$  为  $n$  维向量,  $y_i \in (0, 1)$ 。我们定义  $N_j (j = 0, 1)$  为第  $j$  类样本的个数,  $X_j (j = 0, 1)$  为第  $j$  类样本的集合, 而  $u_j (j = 0, 1)$  为第  $j$  类样本的均值向量, 定义  $\sum_j (j = 0, 1)$  为第  $j$  类样本的协方差矩阵 (**严格说是缺少分母部分的协方差矩阵**)。

$u_j$  的表达式为：

$$u_j = \frac{1}{N_j} \sum_{x \in X_j} x (j = 0, 1)$$

$\sum_j$  的表达式为：

$$\sum_j = \sum_{x \in X_j} (x - u_j)(x - u_j)^T (j = 0, 1)$$

由于是两类数据, 因此我们只需要将数据投影到一条直线上即可。假设我们的投影直线是向量  $w$ , 则对任意一个样本  $x_i$ , 它在直线上的投影为  $w^T x_i$ , 对于我们的两个类别的中心点  $u_0, u_1$ , 在直线上的投影为  $w^T u_0, w^T u_1$ 。由于 LDA 需要让不同类别的数据的类别中心之间的距离尽可能的大, 也就是我们要最大化  $\|w^T u_0 - w^T u_1\|_2^2$ , 同时我们希望同一种类别数据的投影点尽可能的接近, 也就是要同类样本投影点的协方差  $w^T \sum_0 w, w^T \sum_1 w$  尽可能的小, 即最小化  $w^T \sum_0 w + w^T \sum_1 w$ , 综上所述, 我们的优化目标为最大化:

$$J = \frac{\|w^T u_0 - w^T u_1\|_2^2}{w^T \sum_0 w + w^T \sum_1 w} = \frac{w^T (u_0 - u_1)(u_0 - u_1)^T w}{w^T (\sum_0 + \sum_1) w}$$

定义类内散度矩阵:

$$S_w = \sum_0 + \sum_1 = \sum_{x \in X_0} (x - u_0)(x - u_0)^T + \sum_{x \in X_1} (x - u_1)(x - u_1)^T$$

以及类间散度矩阵:

$$S_b = (u_0 - u_1)(u_0 - u_1)^T$$

则优化目标可重写为:

$$J = \frac{w^T S_b w}{w^T S_w w}$$

这就是 LDA 欲最大化目标, 即  $S_b, S_w$  的**广义瑞利商**。

根据广义瑞利商的性质, 矩阵  $S_w^{-1} S_b$  的最大特征值即为  $J$  的最大值, 矩阵  $S_w^{-1} S_b$  的最大特征向量即为  $w$ 。

$$w = S_w^{-1}(u_0 - u_1)$$

### 线性判别分析算法流程

输入：数据集  $D = (x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$ , 其中任意样本  $x_i$  为  $n$  维向量,  $y_i \in (0, 1)$ 。  
输出：降维后的数据集  $\hat{D}$ 。

算法流程如下：

1. 划分出第一类样本与第二类样本
2. 获取第一类样本与第二类样本中心点
3. 计算第一类样本与第二类样本协方差矩阵
4. 计算类内散度矩阵
5. 计算  $w$
6. 计算新样本集

## 2. 代码实现：

```
import numpy as np
from numpy.linalg import inv
def lda(X, y):
    ...
    input:X(ndarray):待处理数据
    y(ndarray):待处理数据标签, 标签分别为 0 和 1
    output:X_new(ndarray):处理后的数据
    ...

    ***** Begin *****
    # 划分出第一类样本与第二类样本
    x0 = X[y == 0]
    x1 = X[y == 1]

    # 获取第一类样本与第二类样本中心点
    mean0 = np.mean(x0, axis=0)
    mean1 = np.mean(x1, axis=0)

    # 计算第一类样本与第二类样本协方差矩阵
    cov0 = np.dot((x0-mean0).T,(x0-mean0))
    cov1 = np.dot((x1-mean1).T,(x1-mean1))

    # 计算类内散度矩阵
    S_w = cov0 + cov1

    # 计算 w
    w = inv(S_w).dot((mean0 - mean1).reshape(len(mean0),1))

    # 计算新样本集
```

```
X_new = X.dot(w)
***** End ****#
return X_new
```

注意：其中  $X$  维度是(100,2)，即 100 个样本，每个样本有 2 个特征 (x,y 坐标) 注意计算协方差矩阵时要自己根据公式写，调用 `np.cov()` 结果错误；注意计算  $w$  时  $\text{mean0}-\text{mean1}$  要 `reshape` 成一列，即原本  $x_0$  是(50,2)的，50 个样本，每个样本两个特征， $\text{mean0}$  是(2,)的，即  $1 \times 2$ ，1 行 2 列的，现在要 `reshape` 成 1 列，即将原本第 1 列的特征竖直拼接到第 0 列的后面，变成 1 列。

使用 `sklearn` 实现 LDA：

`LinearDiscriminantAnalysis` 的使用代码如下：

```
1. lda = LinearDiscriminantAnalysis(n_components=1)
2. lda.fit(X,y)
3. X_new = lda.transform(X)
```

- `n_components`: 即我们进行 LDA 降维时降到的维数。在降维时需要输入这个参数。需要注意的是, `n_components` 值的范围是 1 到 类别数-1 之间的值。

LinearDiscriminantAnalysis 类中的 `fit` 函数用于训练模型, `fit` 函数有两个向量输入:

- `X` : 大小为\*\*[样本数量,特征数量]\*\*的 ndarray, 存放训练样本;
- `Y` : 值为整型, 大小为\*\*[样本数量]\*\*的 ndarray, 存放训练样本的标签值。

LinearDiscriminantAnalysis 类中的 `transform` 函数用于降维, 返回降维后的数据, `transform` 函数有一个向量输入:

- `X` : 大小为\*\*[样本数量,特征数量]\*\*的 ndarray, 存放需降维的样本。

```
from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis

def lda(x,y):
    """
    input:x(ndarray):待处理数据
    y(ndarray):待处理数据标签
    output:x_new(ndarray):降维后数据
    """

    #***** Begin ****#
    lda=LinearDiscriminantAnalysis()
    lda.fit(x,y)
    x_new=lda.transform(x)
    #***** End *****#
    return x_new
```

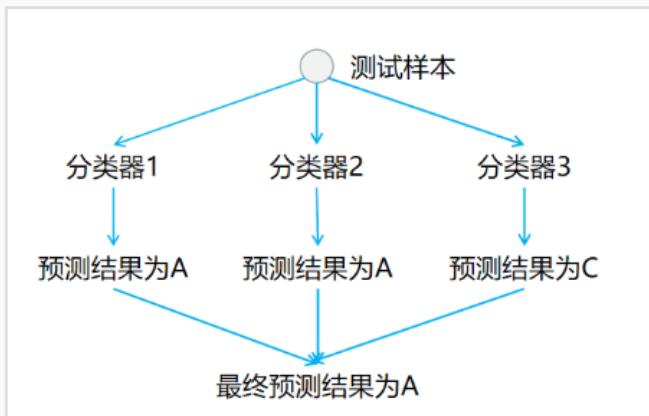
## OvO 多分类

### 1. 原理

如果想要使用逻辑回归算法来解决这种 3 分类问题，可以使用 ovo。ovo ( one vs one )是使用二分类算法来解决多分类问题的一种策略。从字面意思可以看出它的核心思想就是**一对一**。所谓的“一”，指的是类别。而“对”指的是从训练集中划分不同的两个类别的组合来训练出多个分类器。

划分的规则很简单，就是组合 $C_n^2$ ，其中 n 表示训练集中类别的数量，在这个例子中为 3 )。如下图所示(其中每一个矩形框代表一种划分)：

在预测阶段，只需要将测试样本分别扔给训练阶段训练好的 3 个分类器进行预测，最后将 3 个分类器预测出的结果进行投票统计，票数最高的结果为预测结果。如下图所示：



OvO 算法的核心思想就是：类别之间两两训练一个二分类器，当测试数据来的时候，让测试数据通过所有的二分类器，每个二分类器将测试数据分类为 2 个类之一，分类为哪个类，哪个类的投票数就+1，最后得票数最多的类就是最后的多分类预测结果。

## 2. 代码

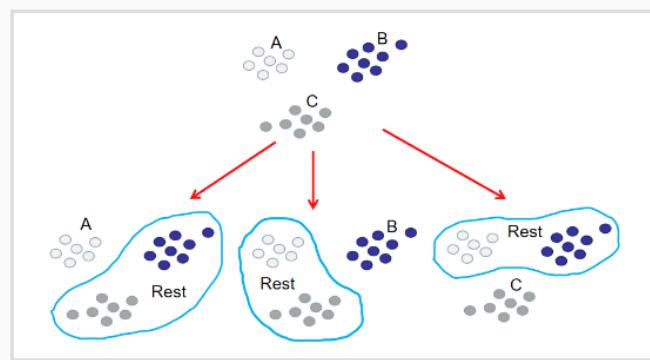
```
1. class OvO(object):
2.     def __init__(self):
3.         # 用于保存训练时各种模型的 list
4.         self.models = []
5.
6.
7.     def fit(self, train_datas, train_labels):
8.         ...
9.         OvO 的训练阶段，将模型保存到 self.models 中
10.        :param train_datas: 训练集数据，类型为 ndarray
11.        :param train_labels: 训练集标签，标签值为 0,1,2 之类的整数，
12.        类型为 ndarray, shape 为(-1,)
13.        :return:None
14.
15.        #***** Begin *****
16.        unique_labels=list(set(train_labels))
17.        for i in range(len(unique_labels)):
18.            for j in range(i+1,len(unique_labels)):
19.                #两两类别(i 类和j 类)训练一个二分类器
20.                data=train_datas[(train_labels==unique_labels[i])
21.                | (train_labels==unique_labels[j])]
22.                label=train_labels[(train_labels==unique_labels[i]
23.                ) | (train_labels==unique_labels[j])]
24.                lr=tiny_logistic_regression()
25.                lr.fit(data,label)
26.                self.models.append(lr)
27.        #***** End *****
28.
29.    def predict(self,test_datas):
30.
31.        def _predict(models, test_datas):
32.            ...
33.            OvO 的预测阶段
34.            :param test_datas: 测试集数据，类型为 ndarray
35.            :return: 预测结果，类型为 ndarray
36.            ...
37.            #***** Begin *****
38.            test_datas=np.reshape(test_datas,(1,-1)) #要先转为1
行的行向量
```

```
38.         vote={}
39.         for model in models:
40.             #type(model.predict(test_datas))=np.ndarray
41.             #例如: model.predict(test_datas)=[1], 即预测出的类
42.             别为1
43.             #故pred=model.predict(test_datas)[0]=1, 把其中的1
44.             #这个数字取出来
44.             pred=model.predict(test_datas)[0] #pred 是一个数
45.             字, 表示预测的类别, 如1
45.             if pred not in vote:
46.                 vote[pred]=1
46.             else:
47.                 vote[pred]+=1
48.             vote=sorted(vote.items(),key=lambda x:x[1],reverse=True)
49.             return vote[0][0]
50.             **** End ****
51.             predict=[]
52.             for data in test_datas:
53.                 predict.append(_predict(self.models,data))
54.             return np.array(predict)
```

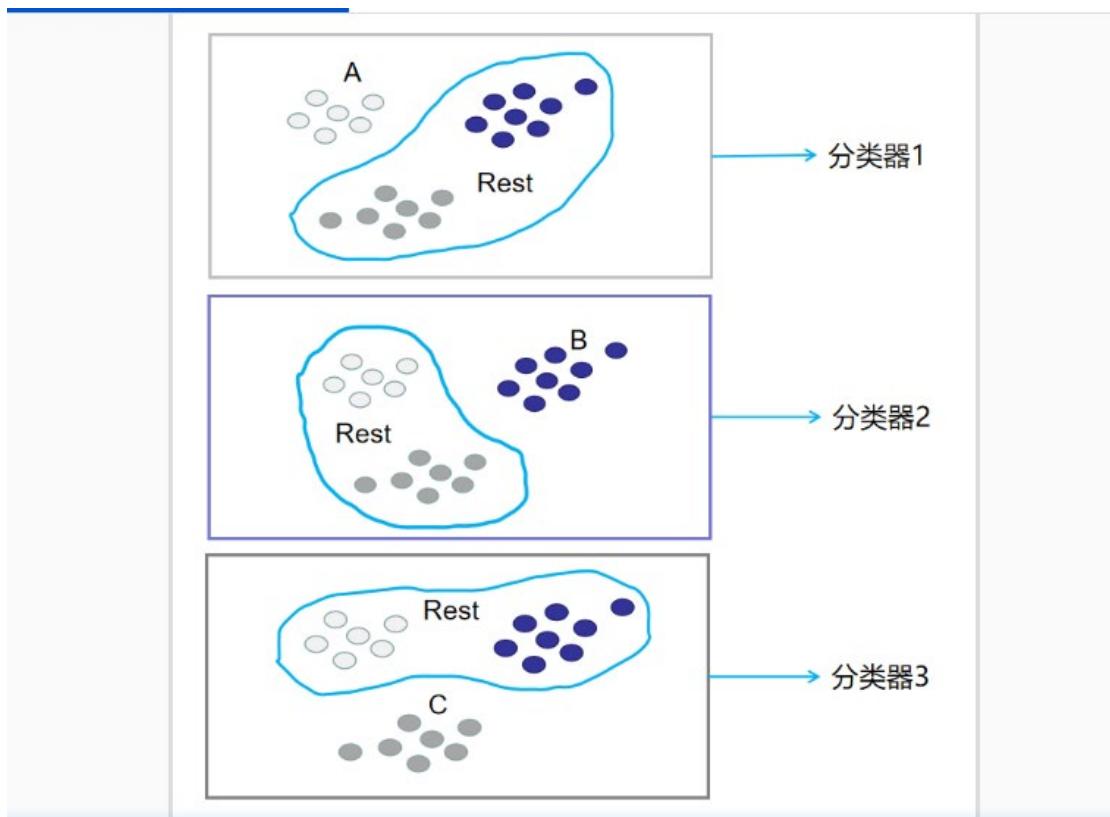
## OvR 多分类

### 1. 原理

如果想要使用逻辑回归算法来解决这种 3 分类问题，可以使用 OvR。OvR (One Vs Rest) 是使用二分类算法来解决多分类问题的一种策略。从字面意思可以看出它的核心思想就是**一对剩余**。一对剩余的意思是当要对  $n$  种类别的样本进行分类时，分别取一种样本作为一类，将剩余的所有类型的样本看做另一类，这样就形成了  $n$  个二分类问题。所以和 ovo 一样，在训练阶段需要进行划分。



分别用这 3 种划分，划分出来的训练集训练二分类分类器，就能得到 3 个分类器。此时训练阶段已经完毕。如下图所示：



在预测阶段，只需要将测试样本分别扔给训练阶段训练好的 3 个分类器进行预测，最后选概率最高的类别作为最终结果。如下图所示：



## 2. 代码

```

class OvR(object):
    def __init__(self):
        # 用于保存训练时各种模型的 list
        self.models = []
        # 用于保存 models 中对应的正例的真实标签
        # 例如第 1 个模型的正例是 2，则 real_label[0]=2
        self.real_label = []
  
```

```

def fit(self, train_datas, train_labels):
    ...
    OvR 的训练阶段，将模型保存到 self.models 中
    :param train_datas: 训练集数据，类型为 ndarray
    :param train_labels: 训练集标签，类型为 ndarray, shape 为
    (-1, )
    :return:None
    ...

    #***** Begin *****
    for i in np.unique(train_labels):
        model=tiny_logistic_regression()
        model.fit(train_datas,train_labels==i) #train_labels==i 为布尔数组，变成 OvR 二分类问题，标签为 i 的就是正类，为 True，其他不是 i 的就是负类，为 False，这一步只是把标签二值化，变为 1（正类，即类别 i 的）和 0（负类，即类别非 i 的），但还是把所有的数据和标签都输入到 model 中去 fit 的
        self.models.append(model)
        self.real_label.append(i)
    #***** End *****

def predict(self, test_datas):
    ...
    OvR 的预测阶段
    :param test_datas: 测试集数据，类型为 ndarray
    :return: 预测结果，类型为 ndarray
    ...

    #***** Begin *****
    predict_result=[]
    #test_datas.shape=(30,4) 测试数据共 30 个样本，每个样本 4 个特征
    for model in self.models:
        predict_result.append(model.predict_proba(test_datas))
    predict_result=np.array(predict_result)
    #predict_result.shape=(3,30), 有 3 个分类器，测试数据共 30 个样本
    predict_result=predict_result.T #(30,3)
    result=np.argmax(predict_result, axis=1) #(30,) 对每个样本，通过 3 个分类器，取出得到概率最高的那个分类器，得到的是每个样本，概率最高的是由哪一个分类器提供的
    #result.shape=(30,)

```

```
    result=[self.real_label[i] for i in result] #找到分类器  
对应的正类是哪一类  
    return result  
  
#***** End *****#
```

## KNN

### 1. 原理

#### kNN 算法的优缺点

从算法流程中可以看出，kNN 算法的优点有：

- 原理简单，实现简单；
- 天生支持多分类，不像其他二分类算法在进行多分类时要使用 OvO、OvR 的策略。

缺点也很明显：

- 当数据量比较大或者数据的特征比较多时，预测过程的时间效率太低。

### 2. 代码

```
class kNNClassifier(object):  
    def __init__(self, k):  
        ...  
        初始化函数  
        :param k:kNN 算法中的 k  
        ...  
        self.k = k  
        # 用来存放训练数据，类型为 ndarray  
        self.train_feature = None  
        # 用来存放训练标签，类型为 ndarray
```

```
self.train_label = None

def fit(self, feature, label):
    ...
    kNN 算法的训练过程
    :param feature: 训练集数据, 类型为 ndarray
    :param label: 训练集标签, 类型为 ndarray
    :return: 无返回
    ...

#***** Begin ****#
self.train_feature=feature
self.train_label=label
#***** End *****#

def predict(self, feature):
    ...
    kNN 算法的预测过程
    :param feature: 测试集数据, 类型为 ndarray
    :return: 预测结果, 类型为 ndarray 或 list
    ...

#***** Begin ****#
result=[]
for i in range(len(feature)):
    #feature.shape=(38,4) 38 个测试样本, 特征有 4 个
    #计算测试数据与训练数据的欧式距离
    distance=np.sum((self.train_feature-
feature[i])**2, axis=1)**0.5
        #self.train_feature.shape=(112,4) 训练集有 112 个样
本, 每个样本特征有 4 个
        #distance.shape=(112,), 表示当前测试样本 i 和 112 个训练
样本的距离
    #对距离排序, 取出前 k 个
    index=np.argsort(distance)[:self.k]
    #取出前 k 个标签
    label=self.train_label[index]
    #找到 k 个标签中出现次数最多的标签
    result.append(np.argmax(np.bincount(label)))
return np.array(result)
```

关键在于：for 循环遍历测试样本，对于每一个测试样本，都要计算它与所有的训练样本之间的距离，然后找出前 k 个最小的距离，通过这 k 个训练样本的类别投票，票数最高的就是该测试样本的预测类别。

KNN 是懒惰学习，训练阶段只需要把训练样本保存到 self 里面，什么都不用算，测试时才要算距离，时间开销都花在测试阶段了。

### 3. KNN-sklearn 版

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler

def classification(train_feature, train_label, test_feature):
    ...
    对 test_feature 进行红酒分类
    :param train_feature: 训练集数据, 类型为 ndarray
    :param train_label: 训练集标签, 类型为 ndarray
    :param test_feature: 测试集数据, 类型为 ndarray
    :return: 测试集数据的分类结果
    ...

    #***** Begin *****
    scalar=StandardScaler()
    train_feature2=scalar.fit_transform(train_feature)
    test_feature2=scalar.transform(test_feature)
    knn=KNeighborsClassifier()
    knn.fit(train_feature2,train_label)
    return knn.predict(test_feature2)
```

## StandardScaler的使用

由于数据中有些特征的标准差比较大，例如 Proline 的标准差大约为 314。如果现在用 kNN 算法来对这样的数据进行分类的话，kNN 算法会认为最后一个特征比较重要。因为假设有两个样本的最后一个特征值分别为 1 和 100，那么这两个样本之间的距离可能就被这最后一个特征决定了。这样就很有可能会影响 kNN 算法的准确度。为了解决这种问题，我们可以对数据进行标准化。

标准化的手段有很多，而最为常用的就是 Z Score 标准化。Z Score 标准化通过删除平均值和缩放到单位方差来标准化特征，并将标准化的结果的均值变成 0，标准差为 1。

```
1. from sklearn.preprocessing import StandardScaler  
2.  
3. data = [[0, 0], [0, 0], [1, 1], [1, 1]]  
4.  
5. # 实例化StandardScaler对象  
6. scaler = StandardScaler()  
7.  
8. # 用data的均值和标准差来进行标准化，并将结果保存到after_scaler  
9. after_scaler = scaler.fit_transform(data)  
10.  
11. # 用刚刚的StandardScaler对象来进行归一化  
12. after_scaler2 = scaler.transform([[2, 2]])  
13.  
14. print(after_scaler)  
15. print(after_scaler2)
```

## KNeighborsClassifier的使用

想要使用 sklearn 中使用 kNN 算法进行分类，只需要如下的代码（其中 train\_feature、train\_label 和 test\_feature 分别表示训练集数据、训练集标签和测试集数据）：

```
1. from sklearn.neighbors import KNeighborsClassifier  
2.  
3. #生成K近邻分类器  
4. clf=KNeighborsClassifier()  
5. #训练分类器  
6. clf.fit(train_feature, train_label)  
7. #进行预测  
8. predict_result=clf.predict(test_feature)
```

KNeighborsClassifier() 的构造函数包含一些参数的设定。比较常用的参数有以下几个：

- n\_neighbors：即 kNN 算法中的 K 值，为一整数，默认为 5；
- metric：距离函数。参数可以为字符串（预设好的距离函数）或者是 callable 对象。默认值为闵可夫斯基距离；
- p：当 metric 为闵可夫斯基距离公式时可用，为一整数，默认值为 2，也就是欧式距离。

## 决策树

那么如何构造出一棵好的决策树呢？其实构造决策树时会遵循一个指标，有的是按照信息增益来构建，如**ID3算法**；有的是信息增益率来构建，如**C4.5算法**；有的是按照基尼系数来构建的，如**CART算法**。但不管是使用哪种构建算法，决策树的构建过程通常都是一个递归选择最优特征，并根据特征对训练集进行分割，使得对各个子数据集有一个最好的分类的过程。

2、下列说法错误的是？

- A、从树的根节点开始，根据特征的值一步一步走到叶子节点的过程是决策树做决策的过程
- B、决策树只能是一棵二叉树
- C、根节点所代表的特征是最优特征

## 信息熵与信息增益

从机器学习的角度来看，信息熵表示的是信息量的期望值。如果数据集中的数据需要被分成多个类别，则信息量  $I(x_i)$  的定义如下(其中  $x_i$  表示多个类别中的第  $i$  个类别， $p(x_i)$  数据集中类别为  $x_i$  的数据在数据集中出现的概率表示)：

$$I(X_i) = -\log_2 p(x_i)$$

由于信息熵是信息量的期望值，所以信息熵  $H(X)$  的定义如下(其中  $n$  为数据集中类别的数量)：

$$H(X) = -\sum_{i=1}^n p(x_i) \log_2 p(x_i)$$

从这个公式也可以看出，如果概率是 0 或者是 1 的时候，熵就是 0 （因为这种情况下随机变量的不确定性是最低的）。那如果概率是 0.5，也就是五五开的时候，此时熵达到最大，也就是 1。

（就像扔硬币，你永远都猜不透你下次扔到的是正面还是反面，所以它的不确定性非常高）。所以呢，**熵越大，不确定性就越高**。

## 条件熵

在实际的场景中，我们可能需要研究数据集中某个特征等于某个值时的信息熵等于多少，这个时候就需要用到**条件熵**。条件熵  $H(Y|X)$  表示特征 X 为某个值的条件下，类别为 Y 的熵。条件熵的计算公式如下：

$$H(Y|X) = \sum_{i=1}^n p_i H(Y|X = x_i)$$

当然条件熵的性质也和熵的性质一样，概率越确定，条件熵就越小，概率越五五开，条件熵就越大。

## 信息增益

现在已经知道了什么是熵，什么是条件熵。接下来就可以看看什么是信息增益了。所谓的信息增益就是表示我已知条件  $X$  后能得到信息  $Y$  的不确定性的减少程度。

就好比，我在玩读心术。你心里想一件东西，我来猜。我已开始什么都没问你，我要猜的话，肯定是瞎猜。这个时候我的熵就非常高。然后我接下来我会去试着问你是非题，当我问了是非题之后，我就能减小猜测你心中想到的东西的范围，这样其实也就是减小了我的熵。那么我熵的减小程度就是我的信息增益。

所以信息增益如果套上机器学习的话就是，如果把特征 A 对训练集 D 的信息增益记为  $g(D, A)$  的话，那么  $g(D, A)$  的计算公式就是：

$$g(D, A) = H(D) - H(D, A)$$

那信息增益算出来之后有什么意义呢？回到读心术的问题，为了我能更加准确的猜出你心中所想，我肯定是问的问题越好就能猜得越准！换句话来说我肯定是要想出一个信息增益最大（**减少不确定性程度最高**）的问题来问你。其实 ID3 算法也是这么想的。ID3 算法的思想是从训练集 D 中计算每个特征的信息增益，然后看哪个最大就选哪个作为当前结点。然后继续重复刚刚的步骤来构建决策树。

## 2. 代码

```
1. import numpy as np
2.
3.
4. def calcInfoGain(feature, label, index):
5.     """
6.     计算信息增益
7.     :param feature: 测试用例中字典里的 feature，类型为 ndarray
8.     :param label: 测试用例中字典里的 label，类型为 ndarray
9.     :param index: 测试用例中字典里的 index，即 feature 部分特征列的索引。该索引指的是 feature 中第几个特征，如 index:0 表示使用第一个特征来计算信息增益。
10.    :return: 信息增益，类型 float
11.    """
12.
13.    #***** Begin *****
14.    # 计算信息熵
15.    def calcEntropy(label):
16.        entropy = 0
17.        label_set = set(label)
18.        for l in label_set:
19.            p = np.sum(label == l) / len(label)
```

```
20.         entropy -= p * np.log2(p)
21.     return entropy
22.
23. # 计算条件熵
24. def calcCondEntropy(feature, label, index):
25.     cond_entropy = 0
26.     feature_set = set(feature[:, index])
27.     for f in feature_set:
28.         mask = feature[:, index] == f
29.         cond_entropy += np.sum(mask) / len(label) * calcEntropy(label[mask])
30.     return cond_entropy
31.
32. # 计算信息增益
33. info_gain = calcEntropy(label) - calcCondEntropy(feature, label, index)
34. return info_gain
35. ***** End *****
36.
37.feature=np.array([[0, 1], [1, 0], [1, 2], [0, 0], [1, 1]])
38.label=np.array([0, 1, 0, 0, 1])
39.print(calcInfoGain(feature, label, 0))
```

## ID3 决策树

### 1. 原理

ID3 算法伪代码：

```

1. #假设数据集为D, 标签集为A, 需要构造的决策树为tree
2. def ID3(D, A):
3.     if D中所有的标签都相同:
4.         return 标签
5.     if 样本中只有一个特征或者所有样本的特征都一样:
6.         对D中所有的标签进行计数
7.         return 计数最高的标签
8.
9.     计算所有特征的信息增益
10.    选出增益最大的特征作为最佳特征(best_feature)
11.    将best_feature作为tree的根结点
12.    得到best_feature在数据集中所有出现过的值的集合(value_set)
13.    for value in value_set:
14.        从D中筛选出best_feature=value的子数据集(sub_feature)
15.        从A中筛选出best_feature=value的子标签集(sub_label)
16.        #递归构造tree
17.        tree[best_feature][value] = ID3(sub_feature, sub_label)
18.    return tree

```

因此使用决策树进行预测的伪代码也比较简单，伪代码如下：

```

1. #tree表示决策树, feature表示测试数据
2. def predict(tree, feature):
3.     if tree是叶子结点:
4.         return tree
5.     根据feature中的特征值走入tree中对应的分支
6.     if 分支依然是棵树:
7.         result = predict(分支, feature)
8.     return result

```

## 2. 代码

```

1. import numpy as np
2. class DecisionTree(object):
3.     def __init__(self):
4.         #决策树模型
5.         self.tree = {}
6.     def calcInfoGain(self, feature, label, index):
7.         ...
8.         计算信息增益
9.         :param feature: 测试用例中字典里的 feature, 类型为 ndarray
10.        :param label: 测试用例中字典里的 label, 类型为 ndarray
11.        :param index: 测试用例中字典里的 index, 即 feature 部分特征列的索引。该索引指的是 feature 中第几个特征, 如 index:0 表示使用第一个特征来计算信息增益。

```

```
12.         :return:信息增益, 类型 float
13.         ...
14.     # 计算熵
15.     def calcInfoEntropy(label):
16.         ...
17.         计算信息熵
18.         :param label:数据集中的标签, 类型为 ndarray
19.         :return:信息熵, 类型 float
20.         ...
21.         label_set = set(label)
22.         result = 0
23.         for l in label_set:
24.             count = 0
25.             for j in range(len(label)):
26.                 if label[j] == l:
27.                     count += 1
28.             # 计算标签在数据集中出现的概率
29.             p = count / len(label)
30.             # 计算熵
31.             result -= p * np.log2(p)
32.         return result
33.     # 计算条件熵
34.     def calcHDA(feature, label, index, value):
35.         ...
36.         计算信息熵
37.         :param feature:数据集中的特征, 类型为 ndarray
38.         :param label:数据集中的标签, 类型为 ndarray
39.         :param index:需要使用的特征列索引, 类型为 int
40.         :param value:index 所表示的特征列中需要考察的特征值, 类型
    为 int
41.         :return:信息熵, 类型 float
42.         ...
43.         count = 0
44.         # sub_feature 和 sub_label 表示根据特征列和特征值分割出的
    子数据集中的特征和标签
45.         sub_feature = []
46.         sub_label = []
47.         for i in range(len(feature)):
48.             if feature[i][index] == value:
49.                 count += 1
50.                 sub_feature.append(feature[i])
51.                 sub_label.append(label[i])
52.         pH = count / len(feature)
53.         e = calcInfoEntropy(sub_label)
```

```
54.         return pHA * e
55.     base_e = calcInfoEntropy(label)
56.     f = np.array(feature)
57.     # 得到指定特征列的值的集合
58.     f_set = set(f[:, index])
59.     sum_HDA = 0
60.     # 计算条件熵
61.     for value in f_set:
62.         sum_HDA += calcHDA(feature, label, index, value)
63.     # 计算信息增益
64.     return base_e - sum_HDA
65. # 获得信息增益最高的特征
66. def getBestFeature(self, feature, label):
67.     max_infogain = 0
68.     best_feature = 0
69.     for i in range(len(feature[0])):
70.         infogain = self.calcInfoGain(feature, label, i)
71.         if infogain > max_infogain:
72.             max_infogain = infogain
73.             best_feature = i
74.     return best_feature
75. def createTree(self, feature, label):
76.     # 样本里都是同一个label 没必要继续分叉了
77.     if len(set(label)) == 1:
78.         return label[0]
79.     # 样本中只有一个特征或者所有样本的特征都一样的话就看哪个label
80.     # 的票数高
81.     if len(feature[0]) == 1 or len(np.unique(feature, axis=0))
82.     ) == 1:
83.         vote = {}
84.         for l in label:
85.             if l in vote.keys():
86.                 vote[l] += 1
87.             else:
88.                 vote[l] = 1
89.         max_count = 0
90.         vote_label = None
91.         for k, v in vote.items():
92.             if v > max_count:
93.                 max_count = v
94.                 vote_label = k
95.         return vote_label
96.     # 根据信息增益拿到特征的索引
97.     best_feature = self.getBestFeature(feature, label)
```

```
96.         tree = {best_feature: {}}
97.         f = np.array(feature)
98.         # 拿到bestfeature 的所有特征值
99.         f_set = set(f[:, best_feature])
100.        # 构建对应特征值的子样本集 sub_feature, sub_label
101.        for v in f_set:
102.            sub_feature = []
103.            sub_label = []
104.            for i in range(len(feature)):
105.                if feature[i][best_feature] == v:
106.                    sub_feature.append(feature[i])
107.                    sub_label.append(label[i])
108.            # 递归构建决策树
109.            tree[best_feature][v] = self.createTree(sub_feature,
110.                sub_label)
111.        return tree
112.    def fit(self, feature, label):
113.        """
114.        :param feature: 训练集数据, 类型为 ndarray
115.        :param label: 训练集标签, 类型为 ndarray
116.        :return: None
117.        """
118.        self.tree=self.createTree(feature,label)
119.        """
120.    def predict(self, feature):
121.        """
122.        :param feature: 测试集数据, 类型为 ndarray
123.        :return: 预测结果, 如 np.array([0, 1, 2, 2, 1, 0])
124.        """
125.        """
126.        #如果 tree 是叶子结点, 返回叶子结点的值
127.        def predictOne(tree, feature): #利用递归不断走到叶子结点确
定Label 的过程
128.            if not isinstance(tree, dict): #tree 不是字典, 说明是
叶子结点, 直接返回叶子结点的值
129.                return tree
130.            for k, v in tree.items(): #k 是特征索引, v 是特征值
131.                return predictOne(v[feature[k]], feature) #v[fea
ture[k]] 表示根据特征值找到下一个子树
132.
133.            result = []
134.            for f in feature: #对于每一个测试集样本, 预测其 Label
135.                result.append(predictOne(self.tree, f))
```

```
136.         return np.array(result)
137.
138.         #***** End *****#
```

### 解释:

- 假设有一个简单的决策树，`tree = {0: {1: {1: 0, 0: 1}, 0: 1}}`。
  - 最外层字典的键是0，表示第一个特征用于分割数据。
  - 第一个特征取值为1时，进入下一个子树：`{1: {1: 0, 0: 1}}`。
  - 在这个子树中，键是1，表示第二个特征用于进一步分割数据。
  - 第二个特征取值为1时，分类结果为0（叶子节点）。
  - 第二个特征取值为0时，分类结果为1（叶子节点）。

### 3. 解析字典的键和值:

- `k` 是当前节点的特征索引。该索引表示用来分割数据的特征。
- `v` 是一个字典，表示当前特征索引下的不同取值及其对应的子树。

## 工作流程示例：

假设有一个简单的决策树：

python

复制代码

```
tree = {
    0: {
        1: {
            1: 0,
            0: 1
        },
        0: 1
    }
}
```



假设要预测一个样本 `feature = [1, 0]`：

### 1. 第一层：

- `tree` 是字典，进入循环。
- `k` 是 `0`，表示第一个特征。
- `v` 是 `{1: {1: 0, 0: 1}, 0: 1}`。
- 取 `feature[0]`，即 `1`。
- 递归调用 `predictOne`，参数为 `v[1]` 和 `feature`，即 `predictOne({1: 0, 0: 1}, [1, 0])`。

## 2. 第二层:

- `tree` 是 `{1: 0, 0: 1}`。
- `k` 是 `1`，表示第二个特征。
- `v` 是 `{1: 0, 0: 1}`。
- 取 `feature[1]`，即 `0`。
- 递归调用 `predictOne`，参数为 `v[0]` 和 `feature`，即 `predictOne(1, [1, 0])`。

## 3. 叶子节点:

- `tree` 是 `1`，不是字典，返回 `1`。

最终，`predictOne` 返回 `1`，表示样本 `[1, 0]` 的预测标签是  
`1`。

## 信息增益率

信息增益率的数学定义为如下，其中  $D$  表示数据集， $a$  表示数据集中的某一列， $Gain(D, a)$  表示  $D$  中  $a$  的信息增益， $V$  表示  $a$  这一列中取值的集合， $v$  表示  $V$  中的某种取值， $|D|$  表示  $D$  中样本的数量， $|D^v|$  表示  $D$  中  $a$  这一列中值等于  $v$  的数量。

$$Gain\_ratio(D, a) = \frac{Gain(D, a)}{- \sum_{v=1}^V \frac{|D^v|}{|D|} \log_2 \frac{|D^v|}{|D|}}$$

从公式可以看出，信息增益率很好算，只是用信息增益除以另一个分母，该分母通常称为**固有值**。举个例子，还是使用**第二关**中提到过的数据集，第一列是编号，第二列是性别，第三列是活跃度，第四列是客户是否流失的标签（ $0$  表示未流失， $1$  表示流失）。

```
1. def calcInfoGainRatio(feature, label, index):
2.     ...
3.     计算信息增益率
4.     :param feature: 测试用例中字典里的 feature, 类型为 ndarray
5.     :param label: 测试用例中字典里的 label, 类型为 ndarray
6.     :param index: 测试用例中字典里的 index, 即 feature 部分特征列的索引。该索引指的是 feature 中第几个特征, 如 index:0 表示使用第一个特征来计算信息增益。
7.     :return: 信息增益率, 类型 float
8.     ...
9.
10.    #***** Begin *****
11.    V=set(feature[:,index]) #该属性的不同的可取值集合
12.    D=len(label) #所有样本个数
13.    g=0
14.    for v in V: #遍历每一个该属性的可取值
15.        #np.sum(feature[:,index]==v)可以统计出该属性取值为v的样本总共有多少个, 即D_v 的值
16.        D_v_D=np.sum(feature[:,index]==v)/D #D_v/D
17.        g-=D_v_D*np.log2(D_v_D)
18.
19.    InfoGain=calcInfoGain(feature,label,index)
20.    return InfoGain/g
21.    #***** End *****
```

## Gini 指数

## 基尼系数

在 ID3 算法中我们使用了信息增益来选择特征，信息增益大的优先选择。在 C4.5 算法中，采用了信息增益率来选择特征，以减少信息增益容易选择特征值多的特征的问题。但是无论是 ID3 还是 C4.5，都是基于信息论的熵模型的，这里面会涉及大量的对数运算。能不能简化模型同时也不至于完全丢失熵模型的优点呢？当然有！那就是**基尼系数**！

CART 算法使用**基尼系数**来代替信息增益率，基尼系数代表了模型的不纯度，基尼系数越小，则不纯度越低，特征越好。这和信息增益与信息增益率是相反的(它们都是越大越好)。

基尼系数的数学定义为如下，其中  $D$  表示数据集， $p_k$  表示  $D$  中第  $k$  个类别在  $D$  中所占比例。

$$Gini(D) = 1 - \sum_{k=1}^{|y|} p_k^2$$

从表格可以看出， $D$ 中总共有2个类别，设类别为0的比例为 $p_1$ ，则有 $p_1 = \frac{10}{15}$ 。设类别为1的比例为 $p_2$ ，则有 $p_2 = \frac{5}{15}$ 。根据基尼系数的公式可知 $Gini(D) = 1 - (p_1^2 + p_2^2) = 0.4444$ 。

上面是基于数据集  $D$  的基尼系数的计算方法，那么基于数据集  $D$  与特征  $a$  的基尼系数怎样计算呢？其实和信息增益率的套路差不多。计算公式如下：

$$Gini(D, a) = \sum_{v=1}^V \frac{|D^v|}{|D|} Gini(D^v)$$

还是以用户流失的数据为例，现在算一算性别的基尼系数。设性别男为 $v = 1$ ，性别女为 $v = 2$ 则有 $|D| = 15$ ,  $|D^1| = 8$ ,  $|D^2| = 7$ ,  $Gini(D^1) = 0.46875$ ,  $Gini(D^2) = 0.40816$ 。所以 $Gini(D, a) = 0.44048$ 。

```
1. import numpy as np
2.
3. def calcGini(feature, label, index):
4.     ...
5.     计算基尼系数
6.     :param feature: 测试用例中字典里的 feature, 类型为 ndarray
7.     :param label: 测试用例中字典里的 label, 类型为 ndarray
8.     :param index: 测试用例中字典里的 index, 即 feature 部分特征列的索引。该索引指的是 feature 中第几个特征, 如 index:0 表示使用第一个特征来计算信息增益。
9.     :return: 基尼系数, 类型 float
10.    ...
11.
12.    #***** Begin *****
13.    def calGini(label):
14.        label_set=set(label)
15.        n=len(label)
16.        sum_p=0
17.        for l in label_set:
18.            p=np.sum(label==l)/n
```

```
19.         sum_p+=p**2
20.     return 1-sum_p
21.
22.     feature_set=set(feature[:,index]) #当前属性的可取值
23.     n=len(label)
24.     res=0
25.     for f in feature_set:
26.         D_v_D=np.sum(feature[:,index]==f)/n #D_v/D
27.         mask = feature[:,index]==f #若当前属性取值为f，则mask 对应
    的值为True，去label列表里面就能把它们对应的标签取出来放到Gini 的计算
    函数中求
28.         res+=D_v_D*calGini(label[mask])
29.     return res
30. #***** End *****#
```

## 决策树剪枝

从上图可以看出，预剪枝能够降低决策树的复杂度。这种预剪枝处理属于贪心思想，但是贪心有一定的缺陷，就是可能当前划分会降低泛化性能，但在其基础上进行的后续划分却有可能导致性能显著提高。所以有可能会导致决策树出现欠拟合的情况。

后剪枝是先从训练集生成一棵完整的决策树，然后自底向上地对非叶结点进行考察，若将该结点对应的子树替换为叶结点能够带来决策树泛化性能提升，则将该子树替换为叶结点。

后剪枝的思路很直接，对于决策树中的每一个非叶子结点的子树，我们尝试着把它替换成一个叶子结点，该叶子结点的类别我们用子树所覆盖训练样本中存在最多的那个类来代替，这样就产生了一个简化决策树，然后比较这两个决策树在测试数据集中的表现，如果简化决策树在验证数据集中的准确率有所提高，那么该子树就可以替换成叶子结点。该算法以 bottom-up 的方式遍历所有的子树，直至没有任何子树可以替使得测试数据集的表现得以改进时，算法就可以终止。

从后剪枝的流程可以看出，后剪枝是从全局的角度来看待要不要剪枝，所以造成欠拟合现象的可能性比较小。但由于后剪枝需要先生成完整的决策树，然后再剪枝，所以下剪枝的训练时间开销更高。

```
1.     def fit(self, train_feature, train_label, val_feature, val_la  
bel):  
2.         ...  
3.         :param train_feature:训练集数据, 类型为 ndarray  
4.         :param train_label:训练集标签, 类型为 ndarray  
5.         :param val_feature:验证集数据, 类型为 ndarray  
6.         :param val_label:验证集标签, 类型为 ndarray  
7.         :return: None  
8.         ...  
9.         #***** Begin *****#  
10.  
11.        #先构建一棵完整的树  
12.        self.tree=self.createTree(train_feature,train_label)  
13.        #再进行后剪枝  
14.        self.post_cut(val_feature,val_label)  
15.        #***** End *****#  
16.    def predict(self, feature):  
17.        ...  
18.        :param feature:测试集数据, 类型为 ndarray  
19.        :return:预测结果, 如 np.array([0, 1, 2, 2, 1, 0])  
20.        ...  
21.        #***** Begin *****#  
22.  
23.        # 单个样本分类  
24.        def classify(tree, feature):  
25.            if not isinstance(tree, dict):  
26.                return tree  
27.            t_index, t_value = list(tree.items())[0] #取出根节点的  
特征索引和特征值  
28.            f_value = feature[t_index] #取出当前样本的特征值  
29.            if isinstance(t_value, dict): #如果根节点的值是字典,  
说明还有子树  
30.                classLabel = classify(tree[t_index][f_value], fea  
ture) #递归调用  
31.                return classLabel  
32.            else:
```

```
33.             return t_value #如果根节点的值不是字典，说明是叶子  
结点，直接返回叶子结点的值  
34.  
35.         # 对每个样本进行预测  
36.         return np.array([classify(self.tree, f) for f in feature]  
)
```

## 决策树鸢尾花分类-sklearn 版

```
1. ***** Begin *****#  
2. from sklearn.tree import DecisionTreeClassifier  
3. import numpy as np  
4. import pandas as pd  
5.  
6. #.as_matrix()可以将DataFrame 转换成 ndarray 类型, 这样才能正常的使用  
fit 和predict  
7. X_train=pd.read_csv('./step7/train_data.csv').as_matrix()  
8. Y_train=pd.read_csv('./step7/train_label.csv').as_matrix()  
9. X_test=pd.read_csv('./step7/test_data.csv').as_matrix()  
10.  
11.clf=DecisionTreeClassifier()  
12.clf.fit(X_train,Y_train)  
13.res=clf.predict(X_test)  
14.  
15.df=pd.DataFrame(res,columns=['target'])  
16.df.to_csv('./step7/predict.csv',index=False)  
17.***** End *****#
```

## 支持向量机

### 基本思想

支持向量机的思想认为，一条决策边界它如果有很好的泛化性，它需要满足一下以下两个条件：

1. 能够很好的将样本划分
2. 离最近的样本点最远

## 间隔与支持向量

在样本空间中，决策边界可以通过如下线性方程来描述：

$$w^T x + b = 0$$

其中  $w = (w_1, w_2, \dots, w_d)$  为法向量，决定了决策边界的方  
向。  $b$  为位移项，决定了决策边界与原点之间的距离。显然，决策  
边界可被法向量和位移确定，我们将其表示为  $(w, b)$ 。样本空间中  
的任意一个点  $x$ ，到决策边界  $(w, b)$  的距离可写为：

$$r = \frac{|w^T x + b|}{\|w\|}$$

假设决策边界  $(w, b)$  能够将训练样本正确分类，即对于任何一个样  
本点  $(x_i, y_i)$ ，若它为正类，即  $y_i = +1$  时， $w^T x + b \geq +1$ 。若它  
为负类，即  $y_i = -1$  时， $w^T x + b \leq -1$ 。

如图中，距离最近的几个点使两个不等式的等号成立，它们就被称为**支持向量**，即图中两条黄色的线。两个异类支持向量到超平面的  
距离之和为：

$$r = \frac{2}{\|w\|}$$

它被称为**间隔**，即蓝线的长度。欲找到具有“最大间隔”的决策边  
界，即黑色的线，也就是要找到能够同时满足如下式子的  $w$  与  $b$ ：

### SMO 算法

参考：

[https://blog.csdn.net/Cyril\\_KI/article/details/107779454](https://blog.csdn.net/Cyril_KI/article/details/107779454)

## 朴素贝叶斯

从  $1, 2, \dots, 15$  中小明和小红两人各任取一个数字，现已知小明取到的数字是 5 的倍数，请问小明取到的数大于小红取到的数的概率是多少？

- A、 $7/14$
- B、 $8/14$
- C、 $9/14$
- D、 $10/14$

**解释：**

### 小明取到5

当小明取到5时，小红可以取到的数是1到15之间的任意一个，其中小红取到的数小于5的有4个（1, 2, 3, 4）。

因此，小明取到5且大于小红的数的概率是：

$$\frac{4}{14}$$

### 小明取到10

当小明取到10时，小红可以取到的数是1到15之间的任意一个，其中小红取到的数小于10的有9个（1, 2, 3, 4, 5, 6, 7, 8, 9）。

因此，小明取到10且大于小红的数的概率是：

$$\frac{9}{14}$$

### 小明取到15

当小明取到15时，小红可以取到的数是1到15之间的任意一个，其中小红取到的数小于15的有14个 (1, 2, 3, ..., 14)。

因此，小明取到15且大于小红的数的概率是：

$$\frac{14}{14} = 1$$

### 步骤3: 计算总概率

小明取到5、10、15的概率是均等的，每个是 $\frac{1}{3}$ 。因此，总概率是这三种情况下概率的期望值：

$$P(\text{小明取到的数大于小红取到的数}) = \frac{1}{3} \left( \frac{4}{14} + \frac{9}{14} + \frac{14}{14} \right)$$

## 全概率公式

当为了达到某种目的，但是达到目的有很多种方式，如果想知道通过所有方式能够达到目的的概率是多少的话，就需要用到**全概率公式**（上面的例子就是这种情况！）。全概率公式的定义如下：

若事件 $B_1, B_2, \dots, B_n$ 两两互不相容，并且其概率和为 1。那么对于任意一个事件 c 都满足：

$$P(C) = P(B_1)P(C|B_1) + \dots + P(B_n)P(C|B_n) = \sum_{i=1}^n P$$

引例中小明选择哪条路去公司的概率是**两两互不相容的**（只能选其中一条路去公司），\*\*并且和为 1 \*\*。所以小明不迟到的概率可以通过全概率公式来计算，而引例中的计算过程就是用的全概率公式。

贝叶斯公式定义如下，其中 $A$ 表示已经发生的事件， $B_i$ 为导致事件 $A$ 发生的第*i*个原因：

$$P(B_i|A) = \frac{P(A|B_i)P(B_i)}{\sum_{i=1}^n P(A|B_i)P(B_i)}$$

贝叶斯公式看起来比较复杂，其实非常简单，分子部分是**乘法定理**，分母部分是**全概率公式**（分母等于 $P(A)$ ）。

如果我们对贝叶斯公式进行一个简单的数学变换（两边同时乘以分母，再两边同时除以 $P(B_i)$ ）。就能够得到如下公式：

$$P(A|B_i) = \frac{P(B_i|A)P(A)}{P(B_i)}$$

这个公式是朴素贝叶斯分类算法的核心数学公式，至于为什么，下

- 1、 对以往数据分析结果表明，当机器调整得良好时，产品的合格率为 98%，而当机器发生某种故障时，产品的合格率为 55%。每天早上机器开动时，机器调整得良好的概率为 95%。计算已知某日早上第一件产品是合格时，机器调整得良好的概率是多少？
- A、0.94
  - B、0.95
  - C、0.96
  - D、0.97

计算过程： $(0.98*0.95)/(0.98*0.95+0.05*0.55)$

**(贝叶斯公式)**

- 2、一批产品共 8 件，其中正品 6 件，次品 2 件。现不放回地从中取产品两次，每次一件，求第二次取得正品的概率。
- A、1/4
  - B、1/2
  - C、3/4
  - D、1

计算过程： $6/8*5/7 + 2/8*6/7$

**(全概率公式，一正二正+一次二正)**

## 朴素贝叶斯+平滑代码

```
1. def fit(self, feature, label):
2.     ...
3.     对模型进行训练，需要将各种概率分别保存在 self.label_prob 和
4.     self.condition_prob 中
5.     :param feature: 训练数据集所有特征组成的 ndarray
6.     :param label: 训练数据集中所有标签组成的 ndarray
7.     :return: 无返回
8.     ...
9.     #***** Begin *****
10.    #计算先验概率
11.    for l in np.unique(label):
12.        self.label_prob[l]=(np.sum(label==l)+1)/(len(label)+1
13.        en(set(label))) #求该类别的先验概率，加上平滑
13.        self.condition_prob[l]={} #创建空字典
14.
15.    #计算类条件概率
16.    for l in np.unique(label):
17.        for index,f in enumerate(feature.T): #feature.T 之后每
18.            行是一个特征，f 如[1,2,3,4,5]，表示所有样本的某一个特征取值
19.            self.condition_prob[l][index]={} #创建空字典
20.            for ff in np.unique(f): #对于每一个特征的取值，统计
21.                该类别下该特征取值的概率
22.                self.condition_prob[l][index][ff]=(np.sum((la
23.                bel==l) & (f==ff))+1)/(np.sum(label==l)+len(set(f))) #加上平滑，f
24.                表示某特征的取值，len(set(f)) 表示该特征的可取值个数
25.    #***** End *****
26. def predict(self, feature):
27.     ...
28.     对数据进行预测，返回预测结果
29.     :param feature: 测试数据集所有特征组成的 ndarray
30.     :return:
31.     ...
32.     # ***** Begin *****
33.     res=[]
34.     for f in feature: #对于每一个样本
35.         predict_label=None #维护一个最大概率的类别
36.         max_prob=-1
```

```

33.         for k,l in self.label_prob.items(): #k 是类别, l 是先验
概率
34.             conditon_prob=1
35.             for index,ff in enumerate(f):
36.                 conditon_prob*=self.condition_prob[k][index][
ff] #计算条件概率, 连乘
37.             prior_prob=l #该类的先验概率值
38.             prob=conditon_prob*prior_prob #先验概率*条件概率
39.             if prob>max_prob: #维护所有类别中最大的概率, 以及对
应的类别, 这个最大的概率的类别就是预测的类别
40.                 max_prob=prob
41.                 predict_label=k
42.             res.append(predict_label) #将当前这个样本的预测类别加入
到结果中
43.
44.         return np.array(res)
45.
46.     #***** End ****#

```

## 神经网络

### 1. sigmoid 函数的问题

`sigmoid` 函数在  $x$  过大或过小时, 函数变化非常小, 即梯度非常接近 0, 随着神经网络的加深, 在使用梯度下降方法的时候, 由于梯度接近 0, 参数更新接近 0, 网络开始学不到东西, 即梯度消失。所以现在常使用 `relu` 激活函数, 函数公式如下:

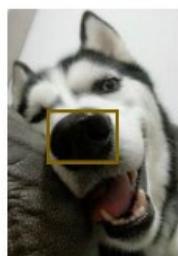
$$relu(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

`relu` 函数在  $x$  大于等于  $0$  时，输出的是  $x$  本身，函数变化率恒为  $1$ ，这样就避免了梯度消失的情况。当  $x$  小于  $0$  时，输出为  $0$ ，即神经元不被激活，这样也符合人脑内并不是所有神经元同时被激活的情况。并且，`relu` 函数计算成本非常低，所以在神经网络过深时常使用 `relu` 函数作为激活函数。

## Pytorch 卷积神经网络

**卷积神经网络**是一种具有局部连接、权重共享等特性的深层前馈神经网络。想要识别图像中的物体，就需要提取出比较好的特征，该特征应能很好地描述想要识别的物体。所以物体的特征提取是一项非常重要的工作。而图像中物体的特征以下几种特点：

1. 物体的特征可能只占图像中的一小部分。比如下图中狗的鼻子只是图像中很小的一部分。
2. 同样的特征可能出现在不同图像中的不同位置，比如下图中狗的鼻子在两幅图中出现的位置不同。



3. 缩放图像的大小对物体特征的影响可能不大，比如下图是缩小后的图，但依然能很清楚的辨认出狗的鼻子。



而卷积神经网络中的**卷积与池化**操作能够较好地抓住物体特征的以上 3 种特点。

## 卷积

卷积说白了就是有一个**卷积核**(其实就是一个带权值的滑动窗口)在图像上从左到右，从上到下地扫描，每次扫描的时候都会将卷积核里的值所构成的矩阵与图像被卷积核覆盖的像素值矩阵做**内积**。整个过程如下图所示，其中黄色方框代表卷积核，绿色部分代表单通道图像，红色部分代表卷积计算后的结果，通常称为特征图：

1	0	1
0	1	0
1	0	1

当这个卷积核卷积的时候就会在3行3列的小范围内计算出图像中几乎所有的 3 行 3 列子图像与卷积核的相似程度（也就是内积的计算结果）。相似程度越高说明该区域中的像素值与卷积核越相似。（上图的特征图中值为 4 的位置所对应到的源图像子区域中像素值的分布与卷积核值的分布最为接近）这也就说明了卷积在提取特征时能够考虑到特征可能只占图像的一小部分，以及同样的特征可能出现在不同的图像中不同的位置这两个特点。

**PS：卷积核的值是怎么确定下来的？很明显是训练出来的！**

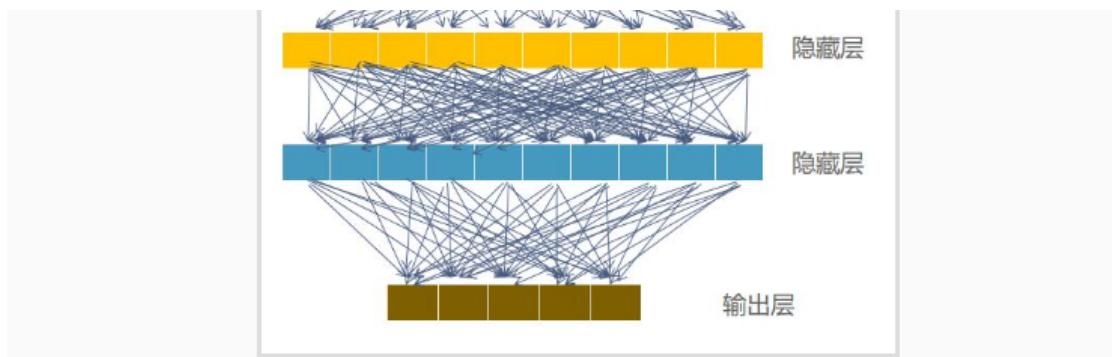
## 池化

池化就是将输入图像进行缩小，减少像素信息，只保留重要信息。池化的操作也很简单，通常情况下，池化区域是 2 行 2 列的大小，然后按一定规则转换成相应的值，例如最常用的最大池化( max pooling )。最大池化保留了每一小块内的最大值，也就是相当于保留了这一块最佳的匹配结果。举个例子，如下图中图像是 4 行 4 列的，池化区域是 2 行 2 列的，所以最终池化后的特征图是 2 行 2 列的。图像中粉色区域最大的值是 6，所以池化后特征图中粉色位置

从上图可以看出，最大池化不仅仅缩小了图像的大小，减少后续卷积的计算量，而且保留了最佳的特征（如果图像是经过卷积后的特征图）。也就相当于把图缩小了，但主要特征还在，这就考虑到了缩放图像的大小对物体的特征影响可能不大的特点。

## 全连接网络

卷积与池化能够很好的提取图像中物体的特征，当提取好特征之后就可以着手开始使用全连接网络来进行分类了。全连接网络的大致结构如下：



其中输入层通常指的是对图像进行卷积，池化等计算之后并进行扁平后的特征图。隐藏层中每个方块代表一个神经元，每一个神经元可以看成是一个很简单的线性分类器和激活函数的组合。输出层中神经元的数量一般为标签类别的数量，激活函数为 softmax（因为将该图像是猫或者狗的得分进行概率化）。因此我们可以讲全连接网络理解成很多个简单的分类器的组合，来构建成一个非常强大的分类器。

需要指出的几个地方：

1. class CNN需要继承Module
2. 需要调用父类的构造方法：super(CNN, self).\_\_init\_\_()
3. 在Pytorch中激活函数Relu也算是一层layer
4. 需要实现forward()方法，用于网络的前向传播，而反向传播只需要调用Variable.backward()即可。

定义好模型后还要构建优化器与损失函数：

torch.optim 是一个实现了各种优化算法的库。使用方法如下：

### 训练模型

在定义好模型后，就可以根据反向传播计算出来的梯度，对模型参数进行更新，在pytorch中实现部分代码如下：

1. #将梯度清零
2. optimizer.zero\_grad()
3. #对损失函数进行反向传播
4. loss.backward()
5. #训练
6. optimizer.step()

### 保存模型

在 pytorch 中使用 torch.save 保存模型，有两种方法，第一种：  
保存整个模型和参数，方法如下：

1. torch.save(model, PATH)

第二种为**官方推荐**，只保存模型的参数，方法如下：

1. torch.save(model.state\_dict(), PATH)

## 加载模型

对应两种保存模型的方法，加载模型也有两种方法，第一种如下：

```
1. model = torch.load(PATH)
```

第二种：

```
1. #CNN()为你搭建的模型  
2. model = CNN()  
3. model.load_state_dict(torch.load(PATH))
```

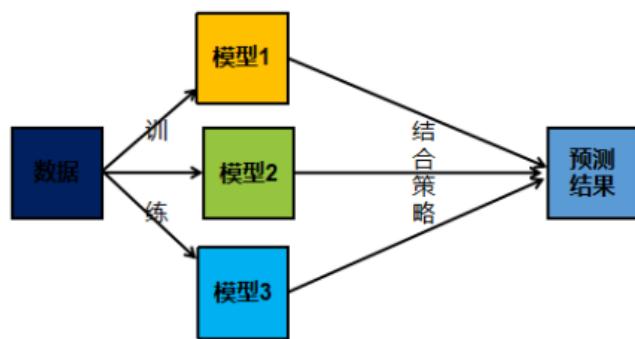
如果要对加载的模型进行测试，需将模型切换为验证模式

如果要对加载的模型进行测试，需将模型切换为验证模式

```
1. model.eval()
```

## 集成学习

集成学习方法是一种常用的机器学习方法，分为 bagging 与 boosting 两种方法，应用十分广泛。集成学习基本思想是：对于一个复杂的学习任务，我们首先构造多个简单的学习模型，然后再把这些简单模型组合成一个高效的学习模型。实际上，就是“**三个臭皮匠顶个诸葛亮**”的道理。



集成学习采取投票的方式来综合多个简单模型的结果，按 bagging 投票思想，如下面例子：

	A	B
模型1	1	0
模型2	0	1
模型3	0	1
模型4	1	0
模型5	0	1

假设一共训练了 5 个简单模型，每个模型对分类结果预测如上图，则最终预测结果为：

A:2

B:3

3>2

结果为 B

不过在有的时候，每个模型对分类结果的确定性不一样，即有的对分类结果非常肯定，有的不是很肯定，说明每个模型投的一票应该是有相应的权重来衡量这一票的重要性。就像在歌手比赛中，每个观众投的票记 1 分，而专家投票记 10 分。按 boosting 投票思想，如下例：

	A	B
模型1	90%	10%
模型2	40%	60%
模型3	30%	70%
模型4	80%	20%
模型5	20%	80%

$$A: (0.9+0.4+0.3+0.8+0.2)/5=0.52$$

$$B: (0.1+0.6+0.7+0.2+0.8)/5=0.48$$

$$0.52 > 0.48$$

结果为 A

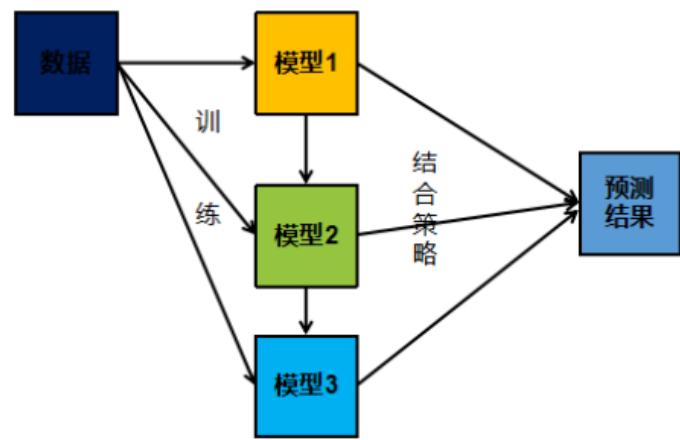
## Boosting

**提升方法**基于这样一种思想：对于一个复杂任务来说，将多个专家的判断进行适当的综合所得出的判断，要比其中任何一个专家单独的判断好。

历史上，Kearns 和 Valiant 首先提出了**强可学习**和**弱可学习**的概念。指出：在 PAC 学习的框架中，一个概念，如果存在一个多项式的学习算法能够学习它，并且正确率很高，那么就称这个概念是强可学习的；一个概念，如果存在一个多项式的学习算法能够学习它，学习的正确率仅比随机猜测略好，那么就称这个概念是弱可学习的。非常有趣的是 Schapire 后来证明强可学习与弱可学习是等价的，也就是说，在 PAC 学习的框架下，一个概念是强可学习的充分必要条件是这个概念是弱可学习的。

这样一来，问题便成为，在学习中，如果已经发现了**弱学习算法**，那么能否将它**提升为强学习算法**。大家知道，发现弱学习算法通常要比发现强学习算法容易得多。那么如何具体实施提升，便成为开发提升方法时所要解决的问题。

与 bagging 不同，boosting 采用的是一个串行训练的方法。首先，它训练出一个**弱分类器**，然后在此基础上，再训练出一个稍好点的**弱分类器**，以此类推，不断的训练出多个弱分类器，最终再将这些分类器相结合，这就是 boosting 的基本思想，流程如下图：



可以看出，子模型之间存在强依赖关系，必须串行生成。boosting是利用不同模型的相加，构成一个更好的模型，求取模型一般都采用序列化方法，后面的模型依据前面的模型。

现在有一份数据，你随机的将数据分成了  $n$  份，然后同时训练  $n$  个子模型，再将模型最后相结合得到一个强学习器，这属于 boosting 方法吗？

- A、是
  - B、不是
  - C、不确定

**Boosting** 是串行的，不能同时训练

对于一个二分类问题，假如现在训练了 500 个子模型，每个模型权重大小一样。若每个子模型正确率为 51%，则整体正确率为多少？若把每个子模型正确率提升到 60%，则整体正确率为多少？

- A、51%,60%
  - B、60%,90%
  - C、65.7%,99.99%
  - D、65.7%,90%

## 情况 1: 每个子模型的正确率为 51%

假设每个模型的正确率是 51%，错误率是 49%。我们有 500 个子模型，最终的预测结果是所有子模型中预测为正例最多的那个。

要计算最终模型正确率，我们需要计算在多数投票中预测正确的概率。对于 500 个模型中的多数投票，需要超过一半（即超过 250 个）模型预测正确。因此，我们需要计算在 500 次试验中，至少 251 次正确的概率。

这个问题可以通过二项分布来解决：

$$P(X \geq 251)$$

其中  $X$  是 500 次试验中成功（正确预测）的次数，成功的概率是 0.51。

python

复制代码

```
from scipy.stats import binom

# 每个模型正确率
p = 0.51
# 总模型数
n = 500
# 至少需要正确的次数
k = 251

# 计算 P(X >= 251)
probability_51 = 1 - binom.cdf(k - 1, n, p)
probability_51
```

本题关键在于：要有至少一遍以上的分类器分类正确，最终集成的模型才分类正确（因为是投票的），所以本质上是一个二项分布，每个分类器的正确率为 51%，要求  $P(x \geq 251)$  的这个累积分布函数，如果是连续变量的分布的话就是从 251 到正无穷的积分，但是二项分布是离散分布，所以是连加，等于  $1 - P(x < 251)$ ，可调用函数求负无穷到

250 的概率累加值。

## Adaboost

### Adaboost算法原理

对提升方法来说，有两个问题需要回答：**一是在每一轮如何改变训练数据的权值或概率分布；二是如何将弱分类器组合成一个强分类器。** 关于第 1 个问题，AdaBoost的做法是，**提高那些被前一轮弱分类器错误分类样本的权值，而降低那些被正确分类样本的权值。** 这样一来，那些没有得到正确分类的数据，由于其权值的加大而受到后一轮的弱分类器的更大关注。于是，分类问题被一系列的弱分类器“分而治之”。至于第 2 个问题，即弱分类器的组合，AdaBoost采取**加权多数表决的方法，加大分类误差率小的弱分类器的权值，使其在表决中起较大的作用，减小分类误差率大的弱分类器的权值，使其在表决中起较小的作用。**

### Adaboost算法流程

AdaBoost 是 AdaptiveBoost 的缩写，表明该算法是具有适应性的提升算法。

算法的步骤如下：

- 1.给每个训练样本( $x_1, x_2, \dots, x_N$ )分配权重，初始权重 $w_1$ 均为 $1/N$ ；
- 2.针对带有权值的样本进行训练，得到模型 $G_m$  (初始模型为 $G_1$ )；
- 3.计算模型 $G_m$ 的误分率：

$$e_m = \sum_i^N w_i I(y_i \neq G_M(X_i))$$

其中：

$$I(y_i \neq G_M(X_i))$$

为指示函数，表示括号内成立时函数值为 1，否则为 0。

4.计算模型 $G_m$ 的系数：

$$\alpha_m = \frac{1}{2} \log \left[ \frac{1 - e_m}{e_m} \right]$$

5.根据误分率 $e$ 和当前权重向量 $w_m$ 更新权重向量：

$$w_{m+1,i} = \frac{w_m}{z_m} \exp(-\alpha_m y_i G_m(x_i))$$

其中 $Z_m$ 为规范化因子：

# Adaboost-sklearn

## AdaBoostClassifier

AdaBoostClassifier 的构造函数中有四个常用的参数可以设置：

- algorithm : 这个参数只有 AdaBoostClassifier 有。主要原因是 scikit-learn 实现了两种 Adaboost 分类算法， SAMME 和 SAMME.R。两者的主要区别是弱学习器权重的度量， SAMME.R 使用了概率度量的连续值，迭代一般比 SAMME 快，因此 AdaBoostClassifier 的默认算法 algorithm 的值也是 SAMME.R；
- n\_estimators : 弱学习器的最大迭代次数。一般来说 n\_estimators 太小，容易欠拟合，n\_estimators 太大，又容易过拟合，一般选择一个适中的数值。默认是 50；
- learning\_rate : AdaBoostClassifier 和 AdaBoostRegressor 都有，即每个弱学习器的权重缩减系数 v， 默认为 1.0；
- base\_estimator : 弱分类学习器或者弱回归学习器。理论上可以选择任何一个分类或者回归学习器，不过需要支持样本权重。我们常用的一般是 CART 决策树或者神经网络 MLP。

## Adaboost 代码：

```
1. class AdaBoost:  
2.     ...  
3.     input:n_estimators(int):迭代轮数  
4.             learning_rate(float):弱分类器权重缩减系数  
5.     ...  
6.     def __init__(self, n_estimators=50, learning_rate=1.0):  
7.         self.clf_num = n_estimators # 弱分类器数目  
8.         self.learning_rate = learning_rate  
9.     def init_args(self, datasets, labels):  
10.        self.X = datasets
```

```

11.         self.Y = labels
12.         self.M, self.N = datasets.shape # M 为样本数, N 为特征数
13.         # 弱分类器数目和集合
14.         self.clf_sets = []
15.         # 初始化weights
16.         self.weights = [1.0/self.M]*self.M # 初始化样本权重系数,
[1.0/self.M]*self.M 表示生成一个长度为self.M 的列表, 每个元素都是
1.0/self.M, 即复制样本个数份
17.         # G(x)系数 alpha
18.         self.alpha = []
19.         #***** Begin *****
20.     def _G(self, features, labels, weights): # 弱分类器
21.         """
22.             input:features(ndarray):数据特征
23.                         labels(ndarray):数据标签
24.                         weights(ndarray):样本权重系数
25.                         ...
26.             m = len(features)
27.             error = 100000.0
28.             best_v = 0.0
29.             #单维features
30.             features_min = min(features)
31.             features_max = max(features)
32.             n_step = (features_max - features_min + self.learning_rate)/self.learning_rate #计算步长,即特征值的范围/学习率
33.             direct, compare_array = None, None #初始化方向和分类结果
34.             for i in range(1, int(n_step)): #对于每个特征值
35.                 v = features_min + self.learning_rate * i #计算阈值
36.                 if v not in features: #如果阈值不在特征值中
37.                     #误分类计算
38.                     compare_array_positive = np.array([1 if features[
k] > v else -1 for k in range(m)]) #对于每个样本, 如果特征大于阈
值, 标签为1, 否则为-1
39.                     weight_error_positive = sum([weights[k] for k in
range(m) if compare_array_positive[k] != labels[k]]) #把所有分类
错的权重都加起来
40.                     compare_array_negative = np.array([
-1 if features[k] > v else 1 for k in range(m)])
41.                     weight_error_negative = sum([weights[k] for k in
range(m) if compare_array_negative[k] != labels[k]])
42.                     if weight_error_positive < weight_error_negative:
#选择误差小的, 正类别误差小, 则模型输出结果为正类别, 否则为负类别
43.                         weight_error = weight_error_positive
44.                         _compare_array = compare_array_positive

```

```
45.                 direct = 'positive'
46.             else:
47.                 weight_error = weight_error_negative
48.                 _compare_array = compare_array_negative
49.                 direct = 'negative'
50.
51.             if weight_error < error: #选择误差最小的
52.                 error = weight_error #更新误差
53.                 compare_array = _compare_array #更新分类结果
54.                 best_v = v #更新阈值
55.             return best_v, direct, error, compare_array
56.
57.     # 计算alpha
58.     def _alpha(self, error):
59.         return 0.5 * np.log((1 - error) / error)
60.
61.     # 规范化因子
62.     def _Z(self, weights, a, clf):
63.         #weights:样本权重系数,a:G(x)系数,即alpha,clf:弱分类器,即
64.         # G(x)
65.         return sum([weights[i] * np.exp(-
66.             1 * a * self.Y[i] * clf[i]) for i in range(self.M)]) #对于每个样
67.         #本, 计算权值*exp(-1*alpha*y*g(x))
68.     # 权值更新
69.     def _w(self, a, clf, Z):
70.         for i in range(self.M): #对于每个样本
71.             self.weights[i] = self.weights[i] * np.exp(-
72.                 1 * a * self.Y[i] * clf[i]) / Z #样本分布更新公式
73.
74.     # G(x)的线性组合
75.     def G(self, x, v, direct):
76.         #x:样本特征, v:特征阈值, direct:正负类别
77.         if direct == 'positive':
78.             return 1 if x > v else -1
79.         else:
80.             return -1 if x > v else 1
81.
82.     def fit(self, X, y):
83.         ...
84.         X(ndarray):训练数据
85.         y(ndarray):训练标签
86.         ...
87.
88.         self.init_args(X, y)
```

```
85.         for epoch in range(self.clf_num): #对于每个弱分类器
86.             best_clf_error, best_v, clf_result = 100000, None, No
87.             ne
88.             # 根据特征维度, 选择误差最小的
89.             for j in range(self.N): #对于每个特征
90.                 features = self.X[:, j]
91.                 #分类阈值, 分类误差, 分类结果
92.                 v, direct, error, compare_array = self._G(feature
93. s, self.Y, self.weights) #获得弱分类器
94.                 if error < best_clf_error:
95.                     best_clf_error = error #更新最小误差
96.                     best_v = v #更新最佳阈值
97.                     final_direct = direct #更新最佳方向
98.                     clf_result = compare_array #更新最佳分类
99.             结果
100.            axis = j #更新最佳特征
101.            if best_clf_error == 0: #如果误差为0, 即分类正确,
102.                跳出循环
103.                break
104.            # 计算G(x)系数a
105.            a = self._alpha(best_clf_error)
106.            self.alpha.append(a)
107.            # 记录分类器
108.            self.clf_sets.append((axis, best_v, final_direct))
109.            #记录弱分类器, 包括特征, 阈值, 方向
110.            # 规范化因子
111.            Z = self._Z(self.weights, a, clf_result)
112.            # 权值更新
113.            self._w(a, clf_result, Z)
114.            def predict(self, data):
115.                ...
116.                input:data(ndarray):单个样本
117.                output:预测为正样本返回+1, 负样本返回-1
118.                ...
119.                res = 0
120.                for i in range(len(self.clf_sets)):
121.                    axis, clf_v, direct = self.clf_sets[i] #获取弱分类
122.                    器, 包括特征, 阈值, 方向
123.                    f_input = data[axis] #获取特征值
```

```

122.             res += self.alpha[i] * self.G(f_input, clf_v, direct
    )    #计算G(x)的线性组合
123.         return 1 if res > 0 else -1
124.
125.     #***** End *****

```

## Bagging 算法

**Bagging 方法的核心思想就是三个臭皮匠顶个诸葛亮。**如果使用 Bagging 解决分类问题，就是将多个分类器的结果整合起来进行投票，选取票数最高的结果作为最终结果。如果使用 Bagging 解决回归问题，就将多个回归器的结果加起来然后求平均，将平均值作为最终结果。

那么 Bagging 方法如此有效呢，举个例子。狼人杀我相信大家都玩过，在天黑之前，村民们都要根据当天所发生的事和别人的发现来投票决定谁可能是狼人。

如果我们将每个村民看成是一个分类器，那么每个村民的任务就是二分类，假设  $h_i(x)$  表示第  $i$  个村民认为  $x$  是不是狼人（-1 代表不是狼人，1 代表是狼人）， $f(x)$  表示  $x$  真正的身份（是不是狼人）， $\epsilon$  表示为村民判断错误的错误率。则有  $P(h_i(x) \neq f(x)) = \epsilon$ 。

根据狼人杀的规则，村民们需要投票决定天黑前谁是狼人，也就是说如果有超过半数的村民投票时猜对了，那么这一轮就猜对了。那么假设现在有  $T$  个村民， $H(x)$  表示投票后最终的结果，则有  $H(x) = \text{sign}(\sum_{i=1}^T h_i(x))$ 。

现在假设每个村民都是有主见的人，对于谁是狼人都有自己的想法，那么他们的错误率也是相互独立的。那么根据 Hoeffding 不等式 可知， $H(x)$  的错误率为：

$$P(H(x) \neq f(x)) = \sum_{k=0}^{T/2} C_T^k (1-\epsilon)^k \epsilon^{T-k} \leq \exp\left(-\frac{1}{2} T (1-2\epsilon)^2\right)$$

根据上式可知，如果 5 个村民，每个村民的错误率为 0.33，那么投票的错误率为 0.749；如果 20 个村民，每个村民的错误率为 0.33，那么投票的错误率为 0.315；如果 50 个村民，每个村民的错误率为 0.33，那么投票的错误率为 0.056；如果 100 个村民，每个村民的错误率为 0.33，那么投票的错误率为 0.003。从结果可以看出，村民的数量越大，那么投票后犯错的错误率就越小。这也是 Bagging 性能强的原因之一。

## Bagging 方法如何训练与预测

### 训练

Bagging 在训练时的特点就是随机有放回采样和并行。

- **随机有放回采样：**假设训练数据集有  $m$  条样本数据，每次从这  $m$  条数据中随机取一条数据放入采样集，然后将其返回，让下一次采样有机会仍然能被采样。然后重复  $m$  次，就能得到拥有  $m$  条数据的采样集，该采样集作为 Bagging 的众多分类器中的一个作为训练数据集。假设有  $T$  个分类器（随便什么分类器），那么就重复  $T$  此随机有放回采样，构建出  $T$  个采样集分别作为  $T$  个分类器的训练数据集。
- **并行：**假设有 10 个分类器，在 Boosting 中，1 号分类器训练完成之后才能开始 2 号分类器的训练，而在 Bagging 中，分类器可以同时进行训练，当所有分类器训练完成之后，整个 Bagging 的训练过程就结束了。

Bagging 训练过程如下图所示：

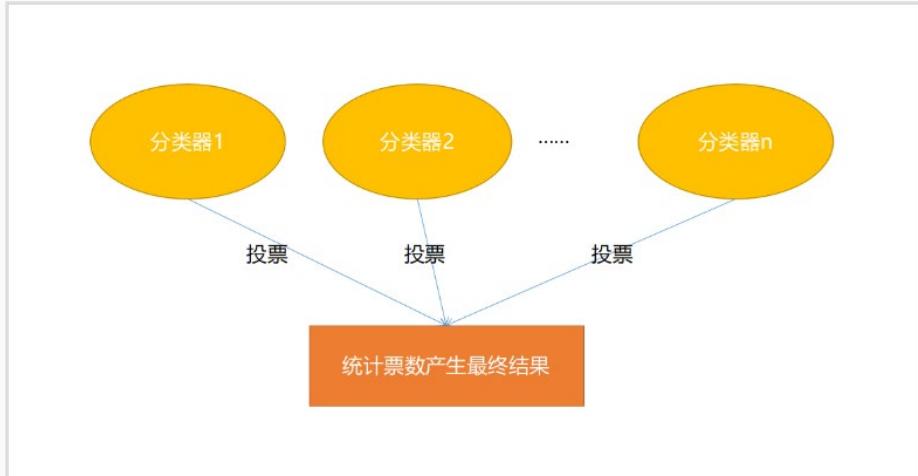


## 预测

Bagging 在预测时非常简单，就是投票！比如现在有 5 个分类器，有 3 个分类器认为当前样本属于 A 类，1 个分类器认为属于 B 类，1 个分类器认为属于 C 类，那么 Bagging 的结果会是 A 类（因为 A 类的票数最高）。

Bagging 预测过程如下图所示：

Bagging 预测过程如下图所示：



Bagging 代码，决策树为基学习器：

```
1. class BaggingClassifier():
2.     def __init__(self, n_model=10):
3.         ...
4.     初始化函数
5.         ...
6.     # 分类器的数量，默认为 10
7.     self.n_model = n_model
8.     # 用于保存模型的列表，训练好分类器后将对象 append 进去即可
9.     self.models = []
10.
11.
12.     def fit(self, feature, label):
13.         ...
14.         训练模型，请记得将模型保存至 self.models
15.         :param feature: 训练集数据，类型为 ndarray
16.         :param label: 训练集标签，类型为 ndarray
17.         :return: None
18.         ...
19.
```

```

20.         #***** Begin *****
21.         for i in range(self.n_model):
22.             #随机选择数据
23.             index = np.random.choice(range(len(feature)),len(feature),replace=True)
24.             #构建决策树
25.             clf = DecisionTreeClassifier()
26.             clf.fit(feature[index],label[index])
27.             self.models.append(clf)
28.         #***** End *****
29.
30.
31.     def predict(self, feature):
32.         ...
33.         :param feature: 测试集数据, 类型为 ndarray
34.         :return: 预测结果, 类型为 ndarray, 如
35.             np.array([0, 1, 2, 2, 1, 0])
36.         ...
37.         #***** Begin *****
38.         #feature.shape=(15,4), 15 个样本, 每个样本4 个特征
39.         predictions=np.array([model.predict(feature) for model in
40.             self.models])
41.         #predictions.shape(10,15) 10 个分类器, 15 个样本, 每一行是一
42.         #个分类器对于所有15 个测试样本的分类结果
43.         res=np.array(
44.             [np.bincount(sample).argmax() for sample in np.array(
45.                 predictions.T)])
46.             ) #现在要转置使得每行表示一个样本, 遍历每个样本, 求出所有10
47.             #个分类器对其类别投票的最高票数的那个类别, 就是这个样本最终的预测类别了
48.
49.     return res
50.     #***** End *****

```

1. 代码重点在于自助采样法的实现,

```
index = np.random.choice(range(len(feature)),len(feature),replace=True)
```

2. 这句代码采样出了 len(feature) 个下标, 这些下标可能有重复, 这就模拟了自助采样法中有放回采样, 采样之后仍有几率在下一次采样中被采样到的过程。

# 随机森林算法

## 随机森林的训练流程

随机森林是 Bagging 的一种扩展变体，随机森林的训练过程相对与 Bagging 的训练过程的改变有：

- 基学习器：Bagging 的基学习器可以是**任意学习器**，而随机森林则是以**决策树作为基学习器**。
- 随机属性选择：假设原始训练数据集有 10 个特征，从这 10 个特征中随机选取 k 个特征构成训练数据子集，然后将这个子集作为训练集扔给决策树去训练。其中 k 的取值一般为  $\log_2(\text{特征数量})$ 。

这样的改动通常会使得**随机森林具有更加强的泛化性**，因为每一棵决策树的训练数据集是随机的，而且训练数据集中的特征也是随机抽取的。如果每一棵决策树模型的差异比较大，那么就很容易能够解决决策树容易过拟合的问题。

随机森林训练过程伪代码如下：

```
1. #假设数据集为D, 标签集为A, 需要构造的决策树为tree
2. def fit(D, A):
3.     models = []
4.     for i in range(决策树的数量):
5.         有放回的随机采样数据, 得到数据集sample_D和标签sample_A
6.         从采样到的数据中随机抽取K个特征构成训练集sub_D
7.         构建决策树tree
8.         tree.fit(sub_D, sample_A)
9.         models.append(tree)
10.    return models
```

## 随机森林的预测流程

随机森林的预测流程与 Bagging 的预测流程基本一致，如果是回归，就将结果基学习器的预测结果全部加起来算平均；如果是分类，就投票，票数最多的结果作为最终结果。**但需要注意的是，在预测时所用到的特征必须与训练模型时所用到的特征保持一致。**例如，第 3 棵决策树在训练时用到了训练集的第 2, 5, 8 这 3 个特征。那么在预测时也要用第 2, 5, 8 这 3 个特征所组成的测试集传给第 3 棵决策树进行预测。

## K-Means

### 1. 距离度量

```
1. def distance(x,y,p=2):
2.     ...
3.     input:x(ndarray):第一个样本的坐标
4.             y(ndarray):第二个样本的坐标
5.             p(int):等于 1 时为曼哈顿距离，等于 2 时为欧氏距离
6.     output:distance(float):x 到 y 的距离
7.     ...
8.     #***** Begin *****
9.     if p==1:
10.         distance=np.sum(abs(x-y))
11.     if p==2:
12.         distance=np.sqrt(np.sum((x-y)**2))
13.     return distance
14.     #***** End *****#
```

### 2. 计算质心

```
1. import numpy as np
2. #计算样本间距离
3. def distance(x, y, p=2):
4.     ...
5.     input:x(ndarray):第一个样本的坐标
6.             y(ndarray):第二个样本的坐标
7.             p(int):等于 1 时为曼哈顿距离，等于 2 时为欧氏距离
8.     output:distance(float):x 到 y 的距离
9.     ...
```

```

10.     #***** Begin *****
11.     if p==1:
12.         distance=np.sum(abs(x-y))
13.     elif p==2:
14.         distance=np.sqrt(np.sum((x-y)**2))
15.     return distance
16.     #***** End *****
17.
18.#计算质心
19.def cal_Cmass(data):
20.
21.    input:data(ndarray):数据样本
22.    output:mass(ndarray):数据样本质心
23.
24.    #***** Begin *****
25.    Cmass=np.mean(data,axis=0)
26.
27.    #***** End *****
28.    return Cmass
29.
30.#计算每个样本到质心的距离，并按照从小到大的顺序排列
31.def sorted_list(data,Cmass):
32.
33.    input:data(ndarray):数据样本
34.            Cmass(ndarray):数据样本质心
35.    output:dis_list(list):排好序的样本到质心距离
36.
37.    #***** Begin *****
38.    dis_list=[]
39.    for i in range(data.shape[0]): #遍历每个样本
40.        dis=distance(data[i],Cmass)
41.        dis_list.append(dis)
42.    dis_list.sort() #对距离进行排序
43.    #***** End *****
44.    return dis_list

```

### 3. Kmeans 算法：

```

1. import numpy as np
2.
3. # 计算一个样本与数据集中所有样本的欧氏距离的平方
4. def euclidean_distance(one_sample, X):
5.     one_sample = one_sample.reshape(1, -1)

```

```
6.     distances = np.power(np.tile(one_sample, (X.shape[0], 1)) - X
    , 2).sum(axis=1)
7.     return distances
8.
9. class Kmeans():
10.     """Kmeans 聚类算法.
11.     Parameters:
12.     -----
13.     k: int
14.         聚类的数目.
15.     max_iterations: int
16.         最大迭代次数.
17.     varepsilon: float
18.         判断是否收敛, 如果上一次的所有 k 个聚类中心与本次的所有 k 个聚类
    中心的差都小于 varepsilon,
19.         则说明算法已经收敛
20.     """
21.     def __init__(self, k=2, max_iterations=500, varepsilon=0.0001
    ):
22.         self.k = k
23.         self.max_iterations = max_iterations
24.         self.varepsilon = varepsilon
25.         np.random.seed(1)
26.         #***** Begin *****
27.         # 从所有样本中随机选取 self.k 样本作为初始的聚类中心
28.         def init_random_centroids(self, X):
29.             n_samples, n_features = X.shape
30.             centroids = np.zeros((self.k, n_features)) # 初始化聚类中
    心
31.             for i in range(self.k):
32.                 centroid = X[np.random.choice(range(n_samples))] # 随
    机选择一个样本作为聚类中心 np.random.choice 不写 size 参数默认返回 1 个随
    机数
33.                 centroids[i] = centroid
34.             return centroids
35.
36.         # 返回距离该样本最近的一个中心索引[0, self.k)
37.         def closest_centroid(self, sample, centroids):
38.             distances = euclidean_distance(sample, centroids) # 计算该
    样本与所有聚类中心的距离
39.             closest_i = np.argmin(distances) # 返回最小值的索引
40.             return closest_i
41.
42.         # 将所有样本进行归类, 归类规则就是将该样本归类到与其最近的中心
```

```
43.     def create_clusters(self, centroids, X):
44.         n_samples = X.shape[0]
45.         clusters = [[] for _ in range(self.k)] # 初始化聚类结果
46.         for sample_i, sample in enumerate(X):
47.             centroid_i = self._closest_centroid(sample, centroids
48. ) # 找到该样本最近的聚类中心
49.             clusters[centroid_i].append(sample_i) # 第 i 个聚类中心
50.             拥有的样本
51.         return clusters
52.
53.     # 对中心进行更新
54.     def update_centroids(self, clusters, X):
55.         n_features = X.shape[1]
56.         centroids = np.zeros((self.k, n_features))
57.         for i, cluster in enumerate(clusters): # cluster 是一个列
58.             表, 里面存放的是属于第 i 个聚类的样本的索引
59.             centroid = np.mean(X[cluster], axis=0) # 计算第 i 个聚
60.             群类新的聚类中心
61.             centroids[i] = centroid # 更新第 i 个聚类的聚类中心
62.         return centroids
63.
64.
65.     # 将所有样本进行归类, 其所在的类别的索引就是其类别标签
66.     def get_cluster_labels(self, clusters, X):
67.         y_pred = np.zeros(X.shape[0])
68.         for cluster_i, cluster in enumerate(clusters): # cluster
69.             是一个列表, 里面存放的是属于第 i 个聚类的样本的索引
70.             for sample_i in cluster: # 遍历第 i 个聚类中的所有样本
71.                 y_pred[sample_i] = cluster_i # 将第 i 个聚类中的所
72.                 有样本的类别标签都设置为 i
73.         return y_pred
74.
75.     # 对整个数据集 X 进行 Kmeans 聚类, 返回其聚类的标签
76.     def predict(self, X):
77.         # 从所有样本中随机选取 self.k 样本作为初始的聚类中心
78.         centroids = self.init_random_centroids(X)
79.         # 迭代, 直到算法收敛(上一次的聚类中心和这一次的聚类中心几乎重
80.         合)或者达到最大迭代次数
81.         for _ in range(self.max_iterations):
82.             # 将所有进行归类, 归类规则就是将该样本归类到与其最近的中心
83.             clusters = self.create_clusters(centroids, X)
84.             # 计算新的聚类中心
85.             prev_centroids = centroids
86.             centroids = self.update_centroids(clusters, X)
```

```

80.
81.          # 如果聚类中心几乎没有变化, 说明算法已经收敛, 退出迭代
82.          diff = centroids - prev_centroids
83.          #只要有一个聚类中心的变化小于varepsilon, 就认为算法已经收
84.          #敛, 退出迭代
85.          if diff.any() < self.varepsilon: #.any()表示只要有一
86.              break
87.          return self.get_cluster_labels(clusters, X)
88.      **** End ****#

```

## Kmeans-sklearn

```

1. #encoding=utf8
2. from sklearn.cluster import KMeans
3.
4. def kmeans_cluster(data):
5.     ...
6.     input:data(ndarray):样本数据
7.     output:result(ndarray):聚类结果
8.     ...
9.     **** Begin ****#
10.    kmeans=KMeans(n_clusters=3,random_state=888)
11.    kmeans.fit(data) #fit就是在求合适的聚类中心的位置的过程
12.    result=kmeans.predict(data)
13.    **** End ****#
14.    return result

```

## DBSCAN

```

1. #encoding=utf8
2. import numpy as np
3. import random
4. #寻找样本点j的eps邻域内的点
5. def findNeighbor(j,X,eps):
6.     N=[]
7.     for p in range(X.shape[0]): #找到所有领域内对象
8.         temp=np.sqrt(np.sum(np.square(X[j]-X[p]))) #欧氏距离
9.         if(temp<=eps):

```

```
10.         N.append(p)
11.     return N
12.
13. #dbscan 算法
14. def dbscan(X,eps,min_Pts):
15.     """
16.         input:X(ndarray):样本数据
17.             eps(float):eps 邻域半径
18.             min_Pts(int):eps 邻域内最少点个数
19.         output:cluster(list):聚类结果
20.     """
21.     #***** Begin *****
22.     main_point=[] #核心点集
23.     for sample_i in range(X.shape[0]):
24.         N=findNeighbor(sample_i,X,eps)
25.         if len(N)>=min_Pts:
26.             main_point.append(sample_i)
27.
28.     k = 0 # 聚类簇数
29.     cluster = [-1] * X.shape[0] # 聚类结果
30.
31.     while len(main_point) > 0:
32.         i = random.choice(main_point) #在核心点集中随机选取一个核心对象
33.         cluster[i] = k #将核心对象标记为当前簇
34.
35.         #找到核心点的所有密度可达点的关键在于使用队列queue, 将邻域内的对象加入队列, 使得下次取出队列中的对象时, 可以继续找到邻居的邻居, 即找到所有密度可达点
36.         #思想有点类似广度优先BFS
37.         queue = [i]
38.         while len(queue) > 0:
39.             q = queue.pop(0) #取出队列中的第一个对象
40.             neighbor = findNeighbor(q, X, eps) #找到q 的eps 邻域内的对象
41.             for p in neighbor: #遍历q 的邻域内对象
42.                 if cluster[p] == -1: #如果该对象还没有被聚类
43.                     cluster[p] = k #将该对象标记为当前簇
44.                     queue.append(p) #将该对象加入队列
45.                 if len(neighbor) >= min_Pts: #如果q 是核心对象
46.                     main_point.remove(q) #将q 从核心对象集中移除
47.
48.         k += 1 # 聚类簇数加1
49.
```

```
50.      #***** End ****#
51.      return cluster
52.
53.#测试
54.data = np.array([[1,2],[2,2],[2,3],[8,7],[8,8],[25,80]])
55.eps=3
56.min_Pts=2
57.result=dbSCAN(data,eps,min_Pts)
58.print(result)
59.#输出: [0, 0, 0, 1, 1, -1] #其中-1 表示噪声点, 最终聚类簇数为2
```

## DBSCAN-sklearn

### DBSCAN

DBSCAN 的构造函数中有两个常用的参数可以设置：

- `eps` : `eps` 邻域半径大小；
- `min_samples` : 即 `Minpts`, `eps` 邻域内样本最少数目；  
和 `sklearn` 中其他聚类器一样, `DBSCAN` 不允许对新的数据进行预测, `DBSCAN` 类中的 `fit_predict` 函数用于训练模型并获取聚类结果, `fit_predict` 函数有一个向量输入:
- `x` : 大小为 `[样本数量,特征数量]` 的 `ndarray` , 存放训练样本。

DBSCAN 的使用代码如下：

```
1. from sklearn.cluster import DBSCAN
2. dbSCAN = DBSCAN(eps=0.5,min_samples =10)
3. result = dbSCAN.fit_predict(x)
```

代码：

```
1. #encoding=utf8
2. from sklearn.cluster import DBSCAN
```

```

3. def data_cluster(data):
4.     ...
5.     input: data(ndarray) :数据
6.     output: result(ndarray):聚类结果
7.     ...
8.     #***** Begin *****
9.     db=DBSCAN(eps=0.5,min_samples=10)
10.    res=db.fit_predict(data)
11.    return res
12.    #***** End *****
13.

```

## 层次聚类

### 1. 簇间距离计算

假设给定簇 $C_i$ 与 $C_j$ ,  $|C_i|, |C_j|$ 分别表示簇 i 与簇 j 中样本的数量, 则平均距离为:

$$d_{min} = \frac{1}{|C_i||C_j|} \sum_{x \in C_i} \sum_{z \in C_j} dist(x, z)$$

```

1. import numpy as np
2.
3.
4. def calc_min_dist(cluster1, cluster2):
5.     ...
6.     计算簇间最小距离
7.     :param cluster1:簇 1 中的样本数据, 类型为 ndarray
8.     :param cluster2:簇 2 中的样本数据, 类型为 ndarray
9.     :return:簇 1 与簇 2 之间的最小距离
10.    ...
11.
12.    #***** Begin *****
13.    min_dist=np.inf
14.    for i in range(len(cluster1)):
15.        for j in range(len(cluster2)):
16.            dist=np.linalg.norm(cluster1[i]-cluster2[j])
17.            if dist<min_dist:
18.                min_dist=dist

```

```
19.     return min_dist
20.     #***** End *****
21.
22.
23.def calc_max_dist(cluster1, cluster2):
24.    ...
25.    计算簇间最大距离
26.    :param cluster1:簇 1 中的样本数据, 类型为 ndarray
27.    :param cluster2:簇 2 中的样本数据, 类型为 ndarray
28.    :return:簇 1 与簇 2 之间的最大距离
29.    ...
30.
31.    #***** Begin *****
32.    max_dist=-np.inf
33.    for i in range(len(cluster1)):
34.        for j in range(len(cluster2)):
35.            dist=np.linalg.norm(cluster1[i]-cluster2[j])
36.            if dist>max_dist:
37.                max_dist=dist
38.    return max_dist
39.    #***** End *****
40.
41.
42.def calc_avg_dist(cluster1, cluster2):
43.    ...
44.    计算簇间平均距离
45.    :param cluster1:簇 1 中的样本数据, 类型为 ndarray
46.    :param cluster2:簇 2 中的样本数据, 类型为 ndarray
47.    :return:簇 1 与簇 2 之间的平均距离
48.    ...
49.
50.    #***** Begin *****
51.    sum_dist=0
52.    for i in range(len(cluster1)):
53.        for j in range(len(cluster2)):
54.            sum_dist+=np.linalg.norm(cluster1[i]-cluster2[j])
55.    return sum_dist / (len(cluster1)*len(cluster2))
56.    #***** End *****
```

## 2. 层次聚类算法:

AGNES 伪代码如下：

```
1. #假设数据集为D, 想要聚成的簇的数量为k
2. def AGNES(D, k):
3.     #C为聚类结果
4.     C = []
5.     #将每个样本看成一个簇
6.     for d in D:
7.         C.append(d)
8.
9.     #C中簇的数量
10.    q=len(C)
11.    while q > k:
12.        寻找距离最小的两个簇a和b
13.        将a和b合并, 并修改c
14.        q = len(C)
15.    return C
```

```
1. import numpy as np
2.
3. def calc_max_dist(cluster1, cluster2):
4.     """
5.     计算簇间最大距离
6.     :param cluster1: 簇 1 中的样本数据, 类型为 ndarray
7.     :param cluster2: 簇 2 中的样本数据, 类型为 ndarray
8.     :return: 簇 1 与簇 2 之间的最大距离
9.     """
10.    max_dist=-np.inf
11.    for i in range(len(cluster1)):
12.        for j in range(len(cluster2)):
13.            dist=np.linalg.norm(cluster1[i]-cluster2[j])
14.            if dist>max_dist:
15.                max_dist=dist
16.    return max_dist
17.
18.
19. def AGNES(feature, k):
20.     """
21.     AGNES 聚类并返回聚类结果, 量化距离时请使用簇间最大欧氏距离
```

```

22.    假设数据集为`[1, 2], [10, 11], [1, 3]`, 那么聚类结果可能为
23.        `[[1, 2], [1, 3]], [[10, 11]]`
24.    :param feature:数据集, 类型为 ndarray
25.    :param k:表示想要将数据聚成`k`类, 类型为`int`、
26.    :return:聚类结果, 类型为 list
27.    ...
28.    #***** Begin *****
29.    C=[]
30.    for d in feature:
31.        C.append(np.array([d]))      #初始化: 每个样本作为一个簇
32.
33.
34.    q=len(C)
35.    while q>k:
36.        min_dist=np.inf
37.        for i in range(len(C)):
38.            for j in range(i+1,len(C)):
39.                dist=calc_max_dist(C[i],C[j])
40.                if dist<min_dist:
41.                    min_dist=dist
42.                    merge_i=i
43.                    merge_j=j
44.        C[merge_i]=np.vstack((C[merge_i],C[merge_j]))  #合并簇, 竖
直拼接是因为一行代表一个样本
45.        C.pop(merge_j)
46.        q-=1
47.    return C
48.    #***** End *****
49.
50.#测试
51.feature = np.array([[1, 2], [10, 11], [1, 3]])
52.k = 2
53.print(AGNES(feature, k))
54.#输出[array([1, 2]), array([1, 3])], [array([10, 11])]]
```

### 3. 层次聚类-sklearn

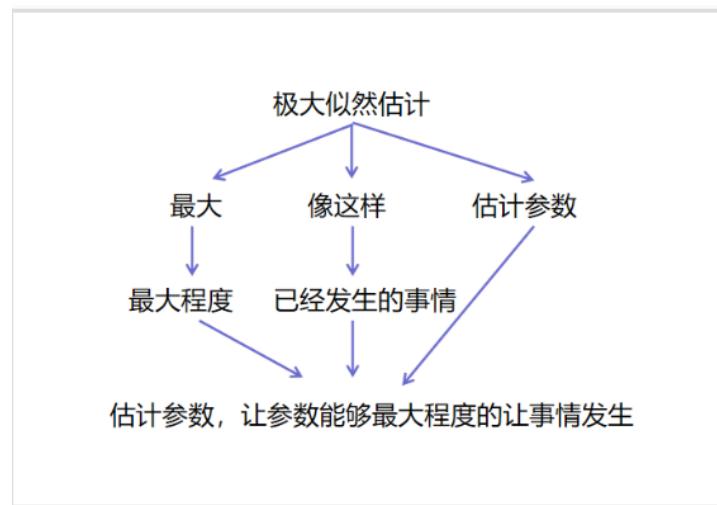
```

1. #encoding=utf8
2. from sklearn.cluster import AgglomerativeClustering
3. from sklearn.preprocessing import StandardScaler
4. def Agglomerative_cluster(data):
5.     ...
6.     对红酒数据进行聚类
```

```
7.     :param data: 数据集, 类型为 ndarray
8.     :return: 聚类结果, 类型为 ndarray
9.     ...
10.
11.    **** Begin ****#
12.    scaler=StandardScaler()
13.    data=scaler.fit_transform(data)
14.
15.    agnes=AgglomerativeClustering(n_clusters=3)
16.    res=agnes.fit_predict(data)
17.    return res
18.    **** End ****#
19.
```

## EM 算法

### 1. 极大似然估计



1、似然函数 $L(\theta|x)$ 中是已知量的是?

A、x

B、 $\theta$

```

1. import numpy as np
2. from scipy import stats
3.
4.
5. def em_single(init_values, observations):
6.     """
7.         模拟抛掷硬币实验并估计在一次迭代中，硬币 A 与硬币 B 正面朝上的概率
8.         :param init_values:硬币 A 与硬币 B 正面朝上的概率的初始值，类型为
9.             list，如[0.2, 0.7]代表硬币 A 正面朝上的概率为 0.2，硬币 B 正面朝上的概率
10.            为 0.7。
11.        :param observations:抛掷硬币的实验结果记录，类型为 list。
12.        :return:将估计出来的硬币 A 和硬币 B 正面朝上的概率组成 list 返回。如
13.            [0.4, 0.6]表示你认为硬币 A 正面朝上的概率为 0.4，硬币 B 正面朝上的概率为
14.            0.6。
15.    """
16.
17.    # **** Begin ****#
18.    # 初始化
19.    p_A, p_B = init_values
20.    # E step
21.    heads_A, tails_A = 0, 0      # 硬币A 正反面次数
22.    heads_B, tails_B = 0, 0      # 硬币B 正反面次数
23.    # 计算硬币A 和硬币B 产生当前观测数据的概率
24.    weight_A = stats.binom.pmf(num_heads, len_observation, p_
25.        A) # 二项分布概率质量函数，返回硬币A 产生当前观测数据的概率, num_heads
26.        为正面次数, len_observation 为抛掷次数, p_A 为硬币A 正面朝上的概率
27.    weight_B = stats.binom.pmf(num_heads, len_observation, p_
28.        B)
29.    # 归一化概率(每一轮都要归一化，因为每一轮的权重和不为1)
30.    norm_weight_A = weight_A / (weight_A + weight_B)
31.    norm_weight_B = weight_B / (weight_A + weight_B)
32.    # 计算硬币A 和硬币B 产生当前观测数据的概率之比
33.    heads_A += norm_weight_A * num_heads      #求期望
34.    tails_A += norm_weight_A * num_tails
35.    heads_B += norm_weight_B * num_heads
36.    tails_B += norm_weight_B * num_tails
37.    # M step
38.    p_A = heads_A / (heads_A + tails_A)      # 更新硬币A 正面朝上的
39.    概率

```

```
36.     p_B = heads_B / (heads_B + tails_B)      # 更新硬币B 正面朝上的
概率
37.     return [p_A, p_B]
38.     #***** End ****#
39.
40.#测试
41.init_values = [0.2, 0.7]
42.observations = [[1, 1, 0, 1, 0], [0, 0, 1, 1, 0], [1, 0, 0, 0, 0]
, [1, 0, 0, 1, 1], [0, 1, 1, 0, 0]]
43.print(em_single(init_values, observations)) #输出
[0.34654779620869536, 0.5287058763827479]
44.
45.
46.import numpy as np
47.from scipy import stats
48.
49.
50.def em_single(init_values, observations):
51.    """
52.    模拟抛掷硬币实验并估计在一次迭代中，硬币A与硬币B正面朝上的概率。
请不要修改！！
53.    :param init_values:硬币A与硬币B正面朝上的概率的初始值，类型为
list，如[0.2, 0.7]代表硬币A正面朝上的概率为0.2，硬币B正面朝上的概率
为0.7。
54.    :param observations:抛掷硬币的实验结果记录，类型为list。
55.    :return:将估计出来的硬币A和硬币B正面朝上的概率组成list返回。如
[0.4, 0.6]表示你认为硬币A正面朝上的概率为0.4，硬币B正面朝上的概率为
0.6。
56.    """
57.    observations = np.array(observations)
58.    counts = {'A': {'H': 0, 'T': 0}, 'B': {'H': 0, 'T': 0}}
59.    theta_A = init_values[0]
60.    theta_B = init_values[1]
61.    # E step
62.    for observation in observations:
63.        len_observation = len(observation)
64.        num_heads = observation.sum()
65.        num_tails = len_observation - num_heads
66.        # 两个二项分布
67.        contribution_A = stats.binom.pmf(num_heads, len_observation,
theta_A)
68.        contribution_B = stats.binom.pmf(num_heads, len_observation,
theta_B)
```

```

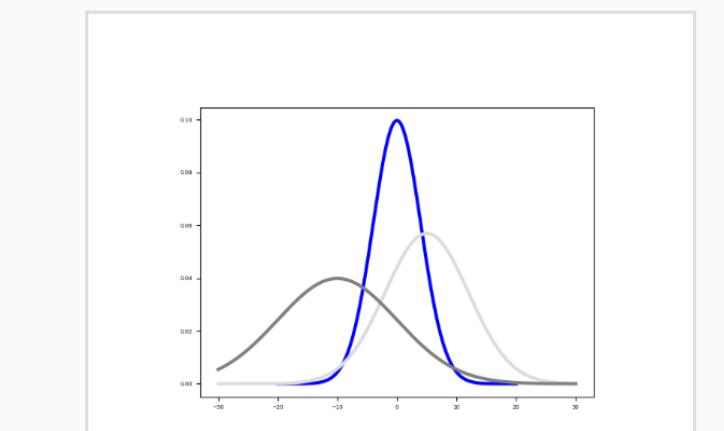
69.         weight_A = contribution_A / (contribution_A + contribution_B)
70.         weight_B = contribution_B / (contribution_A + contribution_B)
71.         # 更新在当前参数下A、B 硬币产生的正反面次数
72.         counts['A']['H'] += weight_A * num_heads
73.         counts['A']['T'] += weight_A * num_tails
74.         counts['B']['H'] += weight_B * num_heads
75.         counts['B']['T'] += weight_B * num_tails
76.     # M step
77.     new_theta_A = counts['A']['H'] / (counts['A']['H'] + counts['A']['T'])
78.     new_theta_B = counts['B']['H'] / (counts['B']['H'] + counts['B']['T'])
79.     return [new_theta_A, new_theta_B]
80.
81.#EM 算法的迭代过程
82.def em(observations, thetas, tol=1e-4, iterations=100):
83.    """
84.        模拟抛掷硬币实验并使用 EM 算法估计硬币 A 与硬币 B 正面朝上的概率。
85.        :param observations: 抛掷硬币的实验结果记录, 类型为 list。
86.        :param thetas: 硬币 A 与硬币 B 正面朝上的概率的初始值, 类型为 list,
87.            如[0.2, 0.7]代表硬币 A 正面朝上的概率为 0.2, 硬币 B 正面朝上的概率为
88.            0.7。
89.        :param tol: 差异容忍度, 即当 EM 算法估计出来的参数 theta 不怎么变化
90.            时, 可以提前跳出循环。例如容忍度为 1e-4, 则表示若这次迭代的估计结果与上
91.            一次迭代的估计结果之间的 L1 距离小于 1e-4 则跳出循环。为了正确的评测, 请
92.            不要修改该值。
93.        :param iterations: EM 算法的最大迭代次数。为了正确的评测, 请不要修
94.            改该值。
95.        :return: 将估计出来的硬币 A 和硬币 B 正面朝上的概率组成 list 或者
96.            ndarray 返回。如[0.4, 0.6]表示你认为硬币 A 正面朝上的概率为 0.4, 硬币 B
97.            正面朝上的概率为 0.6。
98.
99.    """
100.   ****Begin ****#
101.   theta = thetas
102.   for i in range(iterations):
103.       new_theta = em_single(theta, observations)
104.       if sum(abs(np.array(new_theta) - np.array(theta))) < tol:
105.           break
106.       theta = new_theta
107.   ****End ****#

```

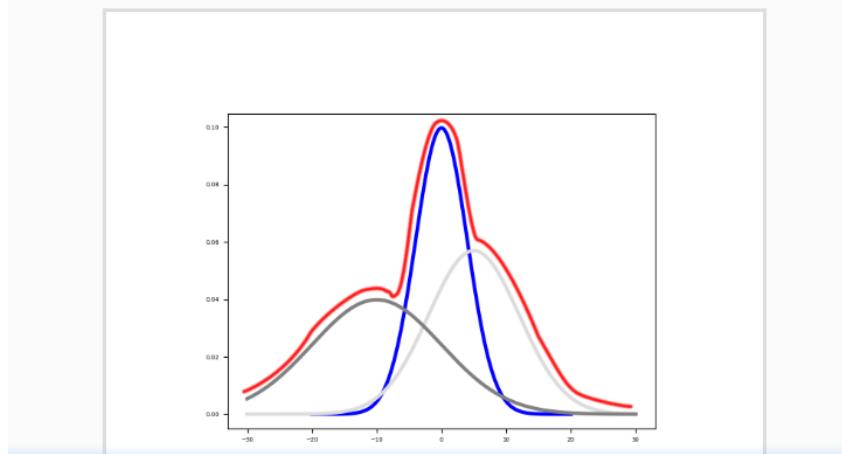
## 高斯混合聚类

### 1. 高斯混合分布

这 3 个高斯分布可能如下图所示：



如果仅仅想用一个分布来描述这 3 个高斯分布的话，我们下意识地可能会觉得如果我的分布如下图中红色部分所示，就相当于用一个分布描述了 3 个高斯分布。



上图中红色的分布其实就是**高斯混合分布**。高斯混合分布其实就是多个高斯分布的带权线性加和。例如上图中红色的分布等于  $0.2 * \text{高斯分布 A} + 0.5 * \text{高斯分布 B} + 0.3 * \text{高斯分布 C}$ 。

现在我们对高斯混合分布有了一定的感官上的认识，下面我们给出高斯混合分布的数学定义。设  $p_M(x)$  为样本  $x$  所服从的概率密度函数(高斯混合分布的概率密度函数)，则有：

$$p_M(x) = \sum_{i=1}^k \alpha_i * p(x|\mu_i, \Sigma_i)$$

并有：

$$\sum_{i=1}^k \alpha_i = 1, \alpha_i > 0$$

其中  $\alpha_i$  表示第  $i$  个高斯分布的系数， $p(x|\mu_i, \Sigma_i)$  为均值向量为  $\mu_i$ ，协方差矩阵为  $\Sigma_i$  的高斯分布。

所以在使用高斯混合聚类时，对于样本的产生过程有这样的假设：

- 首先，根据  $\alpha_1, \alpha_2, \dots, \alpha_k$  定义的先验分布选择高斯分布，其中  $\alpha_i$  为选择第  $i$  个高斯分布的概率(这也是所有  $\alpha$  的和为 1 的原因)。
- 然后，根据被选择的混合成分的概率密度函数进行采样，从而生成相应的样本。

### 高斯混合聚类的核心思想

现在对样本集  $D = \{x_1, x_2, \dots, x_m\}$  使用高斯混合聚类划分成  $k$  个簇。高斯混合聚类会怎样想呢？其实很简单，如果能分别计算出  $x_1$  属于 1 号簇的概率， $x_1$  属于 2 号簇的概率， $\dots$ ， $x_k$  属于  $k$  号簇的概率。接着将概率最大的簇作为聚类结果就好了。同样，样本集中的其他样本也如法炮制，就能实现对样本集的聚类。

那么接下来的问题就是怎样计算这个概率？

想要计算这个概率，可以令随机变量  $z_j \in \{1, 2, \dots, k\}$  表示样本  $x_i$  是从 1 到  $k$  这  $k$  个高斯分布中的哪个高斯分布通过采样所得到的(**假如  $z_1 = 2$  表示  $x_1$  这个样本属于第 2 个高斯分布，也就是说  $x_1$  这个样本属于 2 号簇**)。

有了 $z_j$ 这个随机变量后，就可以使用贝叶斯公式将 $p_M(z_j = i|x_j)$ (即 $x_j$ 属于第*i*个高斯分布的概率)计算出来了。

$$p_M(z_j = i|x_j) = \frac{P(z_j = i) * pM(x_j|z_j = i)}{p_M(x_j)} = \frac{\alpha_i * p(x_j|\mu_i, \Sigma_i)}{\sum_{l=1}^k \alpha_l * p(x_j|\mu_l, \Sigma_l)}$$

为了方便描述，我们不妨将 $p_M(z_j = i|x_j)$ 记成 $\tau_{ji}$ 。所以当高斯混合分布已知时，高斯混合聚类将会把样本集 $D$ 划分为 $k$ 个簇，每个样本 $x_j$ 的簇标记 $\lambda_j$ 以如下方式确定：

$$\lambda_j = argmax_{i \in \{1,2,\dots,k\}} \tau_{ji}$$

1、下列说法正确的是？

- A、 $\alpha$ 的和为1
- B、高斯混合分布是由多个高斯分布组成的
- C、对数据可视化后发现数据大致可以分为4个簇，则高斯混合分布的k可以设置成4
- D、高斯混合聚类使用距离来刻画聚类原型

最后总结一下高斯混合聚类算法的流程。

- 初始化参数；
- EM 算法更新参数；
- 根据高斯混合分布确定簇的划分。

## 聚类与图像分割

那么图像分割怎样和聚类扯上关系呢？很简单，我们知道一副彩色图像是由多个像素点组成的。若把每个像素点看成是一个样本的话，我们就可以通过聚类的方式给每个像素点打上簇标记（比如 0,1,2）。然后再根据簇标记将像素点设置成想要的颜色（比如簇标记为 0 的设置成黄色，簇标记为 1 的设置成绿色，簇标记为 2 的设置成蓝色），然后可视化出来就能得到图像分割的效果。

那么怎样将一副图像转换成我们通常想要的类似表格的数据呢？也很简单，一副彩色图像是由多个像素点组成的，而一个像素点一般是由 R, G, B 三个通道组成的。那么我们可以把每一个像素点看成是数据集中的一一个样本，每个样本包含 3 个特征（RGB 三个通道的值）。所以如果一副彩色图的高是 200，宽是 300。则该图可以看成是一个 60000 行，3 列的数据集。有了我们喜闻乐见的表格数

据后，就可以将数据传递给聚类算法进行聚类了。

如果你对数字图像处理中的阈值化感兴趣，可以 [点这里](#) 学习怎样用另一种思来实现图像分割。

### 图像的基础操作

我们可以使用 PIL 来对图像进行操作。如果你想读取一幅图或者保存一副图，代码如下：

```
1. from PIL import Image  
2.  
3. # 读取road.jpg到im变量中  
4. im = Image.open('road.jpg')  
5.  
6. # 将im保存为new_road.jpg  
7. im.save('new_road.jpg')
```

值得注意的是，当使用 open 函数读取到图像后，我们不能直接将图像传给 sklearn 的聚类算法接口。因为此时的 im 不是 ndarray 或者 list，而且 im 的形状不是喜闻乐见的表格的形状。所以需要进行转换，代码如下：

```
1. import numpy as np
2.
3. # 将im转换成ndarray
4. img = np.array(im)
5.
6. # 将img变形为[-1, 3]的shape，并保存至img_reshape
7. img_reshape = img.reshape(-1, 3)
```

当聚类算法给出结果后，我们需要根据聚类结果给图像上色，代码如下：

```
1. # pred为聚类算法的预测结果，将簇为0的点设置成红色，簇为1的点设置成蓝色
2. img[pred == 0, :] = [255, 0, 0]
3. img[pred == 1, :] = [0, 0, 255]
```

由于 img 为 ndarray，我们需要将其转成 Image 类型才能使用 save 函数实现保存图片的功能。转换代码如下：

```
1. im = Image.fromarray(img.astype('uint8'))
```

## 如何使用 GaussianMixture

GaussianMixture 是 sklearn 提供的高斯混合聚类的一个类，该类的构造函数中可以根据实际需要设置很多参数。但常用的参数是 n\_components 和 max\_iter。其中：

- n\_components：想要聚成几个簇，类型为 int；
- max\_iter：迭代次数，类型为 int。

使用 GaussianMixture 进行聚类很简单，fit-predict 大法就完事了。代码如下：

```

1. # 实例化一个将数据聚成2个簇的高斯混合聚类器
2. gmm = GaussianMixture(2)
3.
4. # 将数据传给fit函数, fit函数会计算出各个高斯分布的参数和响应系数
5. gmm.fit(img_reshape)
6.
7. # 对数据进行聚类, 簇标记为0或1(因为gmm对象想要聚成2个簇)
8. pred = gmm.predict(img_reshape)
9.
10. # 图省事可以这样, 相当于调用了fit之后调用predict
11. pred = gmm.fit_predict(img_reshape)

```

## 聚类性能度量指标

### 1. 外部指标

想要计算上述指标来度量聚类的性能, 首先需要计算出 $a, c, d, e$ 。假设数据集 $E = \{x_1, x_2, \dots, x_m\}$ 。通过聚类模型给出的簇划分 $C = \{C_1, C_2, \dots, C_k\}$ , 参考模型给出的簇划分为 $D = \{D_1, D_2, \dots, D_s\}$ 。 $\lambda$ 与 $\lambda^*$ 分别表示 $C$ 与 $D$ 对应的簇标记, 则有:

$$a = |\{(x_i, x_j) | \lambda_i = \lambda_j, \lambda_i^* = \lambda_j^*, i < j\}|$$

$$b = |\{(x_i, x_j) | \lambda_i = \lambda_j, \lambda_i^* \neq \lambda_j^*, i < j\}|$$

$$c = |\{(x_i, x_j) | \lambda_i \neq \lambda_j, \lambda_i^* = \lambda_j^*, i < j\}|$$

$$d = |\{(x_i, x_j) | \lambda_i \neq \lambda_j, \lambda_i^* \neq \lambda_j^*, i < j\}|$$

举个例子, 参考模型给出的簇与聚类模型给出的簇划分如下:

举个例子，参考模型给出的簇与聚类模型给出的簇划分如下：

编号	参考簇	聚类簇
1	0	0
2	0	0
3	0	1
4	1	1
5	1	2
6	1	2

---

那么满足 $a$ 的样本对为(1, 2)(因为1号样本与2号样本的参考簇都为0，聚类簇都为0)，(5, 6)(因为5号样本与6号样本的参考簇都为1，聚类簇都为2)。总共有2个样本对满足 $a$ ，因此 $a = 2$ 。

满足 $b$ 的样本对为(3, 4)(因为3号样本与4号样本的参考簇不同，但聚类簇都为1)。总共有1个样本对满足 $b$ ，因此 $b = 1$ 。

那么满足 $c$ 的样本对为(1, 3)(因为1号样本与3号样本的聚类簇不同，但参考簇都为0)，(2, 3)(因为2号样本与3号样本的聚类簇不同，但参考簇都为0)，(4, 5)(因为4号样本与5号样本的聚类簇不同，但参考簇都为1)，(4, 6)(因为4号样本与6号样本的聚类簇不同，但参考簇都为1)。总共有4个样本对满足 $c$ ，因此 $c = 4$ 。

满足 $d$ 的样本对为(1, 4)(因为1号样本与4号样本的参考簇不同, 聚类簇也不同), (1, 5)(因为1号样本与5号样本的参考簇不同, 聚类簇也不同), (1, 6)(因为1号样本与6号样本的参考簇不同, 聚类簇也不同), (2, 4)(因为2号样本与4号样本的参考簇不同, 聚类簇也不同), (2, 5)(因为2号样本与5号样本的参考簇不同, 聚类簇也不同), (2, 6)(因为2号样本与6号样本的参考簇不同, 聚类簇也不同), (3, 5)(因为3号样本与5号样本的参考簇不同, 聚类簇也不同), (3, 6)(因为3号样本与6号样本的参考簇不同, 聚类簇也不同)。总共有8个样本对满足 $d$ , 因此 $d = 8$ 。

### JC系数

**JC系数**根据上面所提到的 $a, b, c$ 来计算, 并且值域为 $[0, 1]$ , 值越大说明聚类性能越好, 公式如下:

$$JC = \frac{a}{a + b + c}$$

因此刚刚的例子中,  $JC = \frac{2}{2+1+4} = \frac{2}{7}$

### FM指数

**FM指数**根据上面所提到的 $a, b, c$ 来计算, 并且值域为 $[0, 1]$ , 值越大说明聚类性能越好, 公式如下:

---

$$FMI = \sqrt{\frac{a}{a+b} * \frac{a}{a+c}}$$

因此刚刚的例子中， $FMI = \sqrt{\frac{2}{2+1} * \frac{2}{2+4}} = \sqrt{\frac{4}{18}}$

### Rand指数

**Rand指数**根据上面所提到的 $a$ 和 $d$ 来计算，并且值域为 $[0, 1]$ ，值越大说明聚类性能越好，假设 $m$ 为样本数量，公式如下：

$$RandI = \frac{2(a+d)}{m(m-1)}$$

因此刚刚的例子中， $RandI = \frac{2*(2+8)}{6*(6-1)} = \frac{2}{3}$ 。

## 2. 内部指标

### DB指数

DB 指数又称 DBI，计算公式如下：

$$DBI = \frac{1}{k} \sum_{i=1}^k \max\left(\frac{\text{avg}(C_i) + \text{avg}(C_j)}{d_c(\mu_i, \mu_j)}, i \neq j\right)$$

公式中的表达式其实很好理解，其中 $k$ 代表聚类有多少个簇， $\mu_i$ 代表第 $i$ 个簇的中心点， $\text{avg}(C_i)$ 代表 $C_i$ 第 $i$ 个簇中所有数据与第 $i$ 个簇的中心点的平均距离。 $d_c(\mu_i, \mu_j)$ 代表第 $i$ 个簇的中心点与第 $j$ 个簇的中心点的距离。

举个例子，现在有6条西瓜数据 $\{x_1, x_2, \dots, x_6\}$ ，这些数据已经聚类成了2个簇。

编号	体积	重量	簇
1	3	4	1
2	6	9	2
3	2	3	1
4	3	4	1
5	7	10	2
6	8	11	2

从表格可以看出：

$$k = 2$$

$$\mu_1 = \left( \frac{(3+2+3)}{3}, \frac{(4+3+4)}{3} \right) = (2.67, 3.67)$$

$$\mu_2 = \left( \frac{(6+7+8)}{3}, \frac{(9+10+11)}{3} \right) = (7, 10)$$

$$d_c(\mu_1, \mu_2) = \sqrt{(2.67 - 7)^2 + (3.67 - 10)^2} = 7.67391$$

$$avg(C_1) = (\sqrt{(3-2.67)^2 + (4-3.67)^2} + \sqrt{(2-2.67)^2 + (3-3.67)^2} + \sqrt{(3-2.67)^2 + (4-3.67)^2})/3 = 0.628539$$

$$avg(C_2) = (\sqrt{(6-7)^2 + (9-10)^2} + \sqrt{(7-7)^2 + (10-10)^2} + \sqrt{(8-7)^2 + (11-10)^2})/3 = 0.94281$$

因此有：

$$DBI = \frac{1}{k} \sum_{i=1}^k \max\left(\frac{avg(C_i) + avg(C_j)}{d_c(\mu_i, \mu_j)}\right) = 0.204765$$

DB 指数越小就越意味着簇内距离越小同时簇间距离越大，也就是说DB 指数越小越好。

#### Dunn指数

Dunn 指数又称 DI，计算公式如下：

$$DI = \min_{1 \leq i \leq k} \left\{ \min_{i \neq j} \left( \frac{d_{min}(C_i, C_j)}{\max_{1 \leq l \leq k} diam(C_l)} \right) \right\}$$

公式中的表达式其实很好理解，其中 $k$ 代表聚类有多少个簇， $d_{min}(C_i, C_j)$ 代表第 $i$ 个簇中的样本与第 $j$ 个簇中的样本之间的最短距离， $diam(C_l)$ 代表第 $l$ 个簇中相距最近的样本之间的距离。

还是这个例子，现在有 6 条西瓜数据 $\{x_1, x_2, \dots, x_6\}$ ，这些数据已经聚类成了 2 个簇。

编号	体积	重量	簇
1	3	4	1
2	6	9	2
3	2	3	1
4	3	4	1
5	7	10	2
6	8	11	2

从表格可以看出：

$$k = 2$$

$$d_{min}(C_1, C_2) = \sqrt{(3 - 6)^2 + (4 - 9)^2} = 5.831$$

$$diam(C_1) = \sqrt{(3 - 2)^2 + (4 - 2)^2} = 1.414$$

$$diam(C_2) = \sqrt{(6 - 8)^2 + (9 - 11)^2} = 2.828$$

因此有：

$$DI = \min_{1 \leq i \leq k} \left\{ \min_{i \neq j} \left( \frac{d_{min}(C_i, C_j)}{\max_{1 \leq l \leq k} diam(C_l)} \right) \right\} = 2.061553$$

\*\* Dunn 指数越大意味着簇内距离越小同时簇间距离越大，也就是说 Dunn 指数 越大越好。\*\*

## 降维算法

这次我提供的信息比上面个两次都多（这次有 10 个特征），但是您可能将阿拉斯加误判成哈士奇。因为您可能看到长得像狼和比较二就认为是哈士奇了，也就是发生了**过拟合**的现象。这也说明了不是说数据的特征数量越多，我们的机器学习算法的效果就越强。当数据的特征数量变大时，可能会造成机器学习算法的模型变得非常复杂，**从而导致过拟合**。而且如果我所提供的特征数量越多，比如有 10000 个特征，那么模型训练过程中的时间成本会越大。

所以**维数灾难**通常是指对于已知样本数目，存在一个特征数目的最大值，当实际使用的特征数目超过这个最大值时，机器学习算法的性能不是得到改善，而是退化。

降维的好处在于：1. 减少特征个数，降低时间开销；2. 减少没用的特征，防止模型因为特征数目太多而导致过拟合

2、下列说法错误的是

- A、降维能够减小训练的时间复杂度
- B、降维能够减小预测的时间复杂度
- C、维数灾难不会引起过拟合
- D、根据原始数据挖掘出新特征后，特征数量较多，可能会引发维数灾难

## 降维

既然维数太大可能引发维数灾难，那么如果能有算法能够自动地帮我们把重要性比较高的特征维度保留下来，把其他的维度过滤掉就好了。那这个过程我们称之为**降维**。

从维数灾难的概念出发，我们就能知道**降维的作用了**。

- 降低机器学习算法的时间复杂度；
- 节省了提取不必要特征的开销；
- 缓解因为维数灾难所造成的过拟合现象。

## PCA

### PCA与降维

降维的方法有很多，而最为常用的就是**PCA(主成分分析)**。PCA 是将数据从原来的坐标系转换到新的坐标系，新的坐标系的选择是由数据本身决定的。第一个新坐标轴选择的是原始数据中**方差最大的方向**，第二个新坐标轴的选择和第一个坐标轴**正交且方差最大的方向**。然后该过程一直重复，重复次数为原始数据中的特征数量。**最后会发现大部分方差都包含在最前面几个新坐标轴中，因此可以忽略剩下的坐标轴，从而达到降维的目的。**

## PCA的算法流程

PCA 在降维时，需要指定将维度降至多少维，假设降至 k 维，则 PCA 的算法流程如下：

1. demean；
2. 计算数据的协方差矩阵；
3. 计算协方差矩阵的特征值与特征向量；
4. 按照特征值，将特征向量从大到小进行排序；
5. 选取前 k 个特征向量作为转换矩阵；
6. demean 后的数据与转换矩阵做矩阵乘法获得降维后的数据。

### 1. demean

demean 又称为零均值化，意思是将数据中每个维度上的均值变成 0。那为什么要这样做呢？PCA 实质上是找方差最大的方向，而方差的公式如下(其中 $\mu$ 为均值)：

$$Var(x) = \frac{1}{n} \sum_{i=1}^n (x - \mu)^2$$

如果将均值变成 0，那么方差计算起来就更加方便，如下：

$$Var(x) = \frac{1}{n} \sum_{i=1}^n (x)^2$$

在 numpy 中想要 demean 很简单，代码如下：

```
1. import numpy as np
2.
3. #计算样本各个维度的均值
4. u = np.mean(data, axis=0)
5. #demean
6. after_demean = data - u
```

## 2. 协方差矩阵

协方差描述的是两个特征之间的相关性，当协方差为正时，两个特征呈正相关关系（同增同减）；当协方差为负时，两个特征呈负相关关系（一增一减）；当协方差为0时，两个特征之间没有任何相关关系。

协方差的数学定义如下(假设样本有  $x$  和  $y$  两种特征，而  $X$  就是包含所有样本的  $x$  特征的集合， $Y$  就是包含所有样本的  $y$  特征的集合)：

$$cov(X, Y) = \frac{\sum_{i=1}^n (x_i - \mu_x) \sum_{i=1}^n (y_i - \mu_y)}{n - 1}$$

如果在算协方差之前做了 `demean` 操作，那么公式则为：

$$cov(X, Y) = \frac{\sum_{i=1}^n x_i \sum_{i=1}^n y_i}{n - 1}$$

假设样本只有  $X$  和  $Y$  这两个特征，现在把  $X$  与  $X$ ， $X$  与  $Y$ ， $Y$  与  $X$ ， $Y$  与  $Y$  的协方差组成矩阵，那么就构成了协方差矩阵。而协方差矩阵反应的就是特征与特征之间的相关关系。

	$X$	$Y$
$X$	$cov(X, X)$	$cov(X, Y)$
$Y$	$cov(Y, X)$	$cov(Y, Y)$

NumPy 提供了计算协方差矩阵的函数 `cov`，示例代码如下：

```
1. import numpy as np
2.
3. # 计算after_demean的协方差矩阵
4. # after_demean的行数为样本个数，列数为特征个数
5. # 由于cov函数的输入希望是行代表特征，列代表数据的矩阵，所以要转置
6. cov = np.cov(after_demean.T)
```

### 3. 特征值与特征向量

特征值与特征向量的数学定义：如果向量  $v$  与矩阵  $A$  满足  $Av=\lambda v$ ，则称向量  $v$  是矩阵  $A$  的一个特征向量， $\lambda$  是相应的特征值。

因为协方差矩阵为方阵，所以我们可以计算协方差矩阵的特征向量和特征值。其实这里的特征值从某种意义上来说体现了方差的大小，特征值越大方差就越大。而特征值所对应的特征向量就代表将原始数据进行坐标轴转换之后的数据。

numpy 为我们提供了计算特征值与特征向量的接口 `eig`，示例代码如下：

```
1. import numpy as np
2.
3. #eig函数为计算特征值与特征向量的函数
4. #cov为矩阵, value为特征值, vector为特征向量
5. value, vector = np.linalg.eig(cov)
```

因此，PCA 算法伪代码如下：

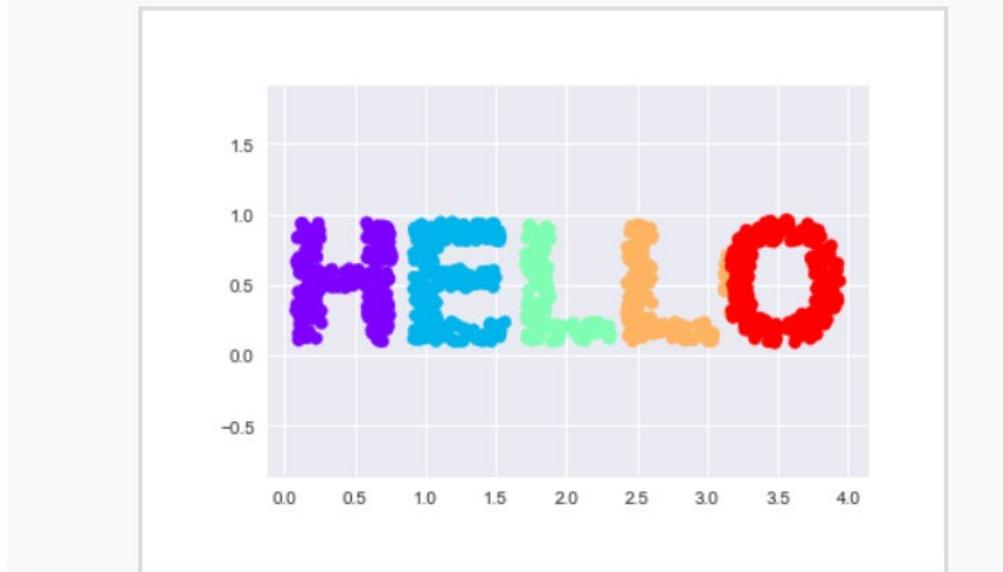
```
1. #假设数据集为D, PCA后的特征数量为k
2. def pca(D, k):
3.     after_demean=demean(D)
4.     计算after_demean的协方差矩阵cov
5.     value, vector = eig(cov)
6.     根据特征值value将特征向量vector降序排序
7.     筛选出前k个特征向量组成映射矩阵P
8.     after_demean和P做矩阵乘法得到result
9.     return result
```

## 多维缩放

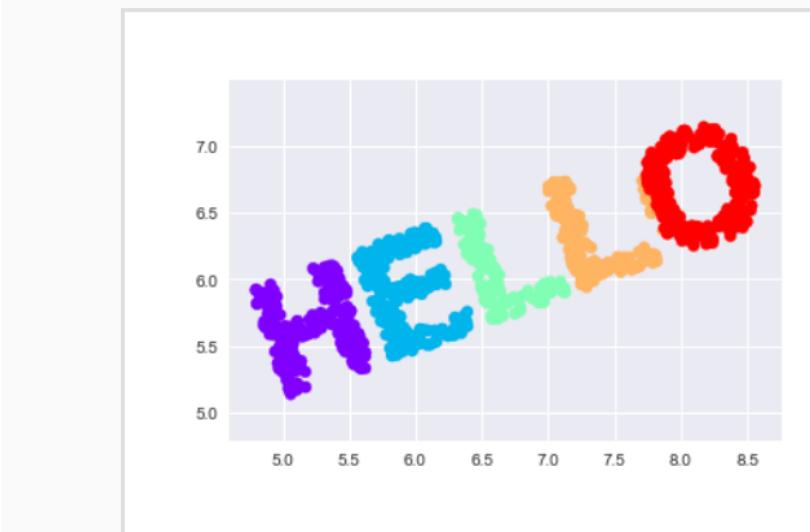
### 1. MDS 算法思想

#### MDS算法思想

假如，一份数据可视化后结果如下：



则将数据进行旋转，数据的特征值发生改变，但是每个点与数据中其他点的距离并没有发生改变：



所以， MDS 算法认为，在数据样本中，每个样本的每个特征值并不是数据间关系的必要特征，真正的基础特征是每个点与数据集中其他点的**距离**。

假设存在距离矩阵，它的第  $i$  行第  $j$  列为  $dist_{ij}$  表示样本  $i$  到样本  $j$  之间的距离，我们要通过样本的坐标计算出距离矩阵非常简单。但是，反过来，我们想通过距离矩阵还原出每个样本的坐标就很困难了，而 MDS 算法就是用来解决这个问题的。它可以将一个数据集的距离矩阵还原成一个  $D$  维坐标来表示数据集。

### MDS算法推导

假设存在  $m$  个样本，在原始空间中的距离矩阵  $D \in R^{m \times m}$ ，其第  $i$  行  $j$  列为样本  $i$  到样本  $j$  之间的距离。目标是获得样本在  $d'$  维空间中的欧式距离等于原空间的欧氏距离，即：

$$\|z_i - z_j\| = dist_{ij}$$

令  $B = Z^T Z \in R^{m \times m}$ ，其中  $B$  为降维后的内积矩阵， $b_{ij} = z_i^T z_j$ ，则：

$$dist_{ij}^2 = \|z_i - z_j\|^2 = \|z_i\|^2 + \|z_j\|^2 - 2z_i^T z_j$$

令降维后的样本被中心化，则：

令降维后的样本被中心化，则：

$$\sum_{i=1}^m dist_{ij}^2 = \text{tr}(B) + mb_{jj}$$

$$\sum_{j=1}^m dist_{ij}^2 = \text{tr}(B) + mb_{ii}$$

$$\sum_{i=1}^m \sum_{j=1}^m dist_{ij}^2 = 2m\text{tr}(B)$$

令

$$dist_i^2 = \frac{1}{m} \sum_{j=1}^m dist_{ij}^2 \dots (1)$$

$$dist_{\cdot j}^2 = \frac{1}{m} \sum_{i=1}^m dist_{ij}^2 \dots (2)$$

$$dist_{\cdot \cdot}^2 = \frac{1}{m^2} \sum_{i=1}^m \sum_{j=1}^m dist_{ij}^2 \dots (3)$$

则

$$b_{ij} = -\frac{1}{2}(dist_{ij}^2 - dist_i^2 - dist_{\cdot j}^2 + dist_{\cdot \cdot}^2)$$

由此即可通过降维前后保持不变的**距离矩阵求取内积矩阵**。

再将**内积矩阵**做特征分解，取  $d$  个最大的特征值所构成的对角矩阵  $\Lambda$ ，并求相应的特征向量矩阵  $V$ ，最后计算出  $Z$ ：

$$Z = V \Lambda^{\frac{1}{2}}$$

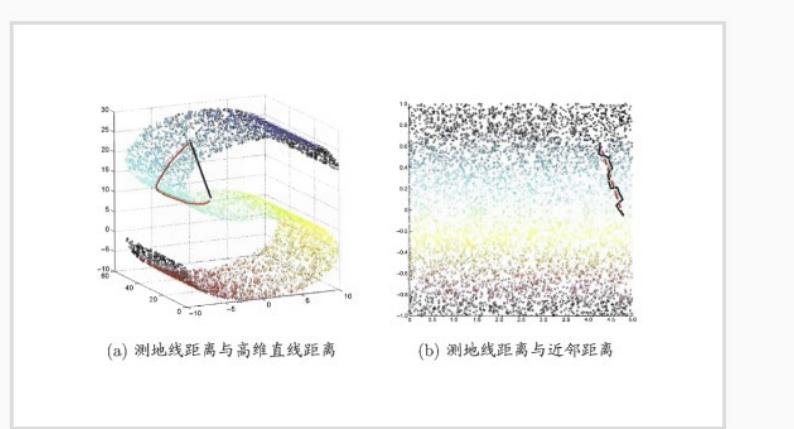
### MDS算法流程

1. 计算式子1,2,3
2. 计算矩阵B
3. 对矩阵B进行特征分解
4. 取d个最大特征值所构成的对角矩阵，并求相应的特征向量矩阵
5. 计算矩阵Z

## 2. 等度量映射

### Isomap算法思想

Isomap 通过“改造一种原本适用于欧氏空间的算法”，达到了“将流形映射到一个欧氏空间”的目的。Isomap 所改造的这个方法是 MDS，它的目的就是使得降维之后的点两两之间的距离尽量不变。只是 MDS 是针对欧氏空间设计的，对于距离的计算也是使用欧氏距离来完成的。如果数据分布在一个流形上的话，欧氏距离就不适用了，如下图：



我们可以发现，测地线距离才是两点之间的本真距离。而 Isomap 算法就将 MDS 算法中所用欧式距离，改为了测地线距离，再采用 MDS 的方法对数据进行降维。

## Isomap算法推导

Isomap 算法将样本与在其邻域内样本之间距离设为**欧式距离**，与其它样本距离设为**无穷大**，如此获得**距离矩阵**。后续推导同 MDS 算法。

1. 对每个样本找出其最近的  $k$  个样本(包括样本本身)。
2. 计算式子1,2,3
3. 计算矩阵  $B$
4. 对矩阵  $B$  进行特征分解
5. 取  $d$  个最大特征值所构成的对角矩阵，并求相应的特征向量矩阵
6. 计算矩阵  $Z$

## LLE (局部线性嵌入)

### LLE算法思想

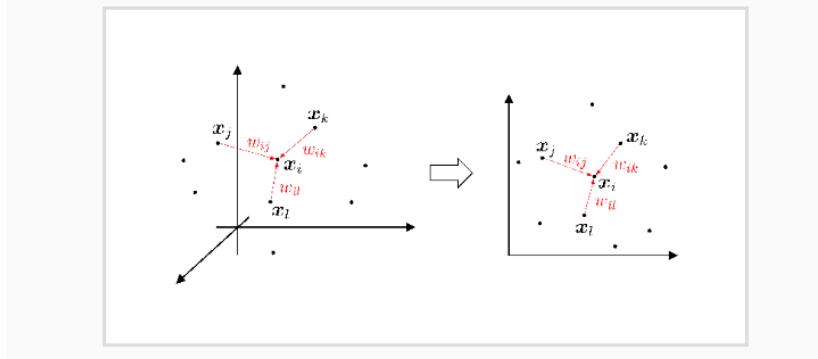
LLE 首先假设数据在较小的局部是线性的，也就是说，某一个数据可以由它邻域中的几个样本来线性表示。比如我们有一个样本 $x_i$ ，我们在它的原始高维邻域里用 K-近邻 思想找到和它最近的三个样本 $x_j, x_k, x_l$ 。然后我们假设 $x_i$ 可以由 $x_j, x_k, x_l$ 线性表示，即：

$$x_i = w_{ij}x_j + w_{ik}x_k + w_{il}x_l$$

在我们通过 LLE 降维后，我们希望 $x_i$ 在低维空间对应的投影 $z_i$ 和 $x_j, x_k, x_l$ 对应的投影 $z_j, z_k, z_l$ 也尽量保持同样的线性关系，即：

$$z_i = w_{ij}z_j + w_{ik}z_k + w_{il}z_l$$

也就是说，投影前后线性关系的权重系数是尽量不变或者最小改变的，如下图：



### LLE算法流程

1. 对每个样本确定其邻域。
2. 求得矩阵C及其逆。
3. 求得权重的值，对不属于邻域内的权重令其等于0。
4. 求得M。
5. 对M进行特征分解。
6. 求得M第2小到第d+1小的特征值对应的特征向量。
7. 求得Z。