ML中相似性度量和距离的计算

前言

在机器学习中，经常需要使用距离和相似性计算的公式，在做分类时，常常需要计算不同样本之间的相似性度量(Similarity Measurement)，计算这个度量，我们通常采用的方法是计算样本之间的“距离(Distance)”。比如利用k-means进行聚类时，判断个体所属的类别，就需要使用距离计算公式得到样本距离簇心的距离，利用kNN进行分类时，也是计算个体与已知类别之间的相似性，从而判断个体的所属类别。

本文对常用的相似性度量进行了一个总结

1. 欧氏距离
2. 曼哈顿距离
3. 切比雪夫距离
4. 闵可夫斯基距离
5. 马氏距离
6. 夹角余弦
7. 汉明距离
8. 杰卡德距离 & 杰卡德相似系数
9. 相关系数 & 相关距离
10. 信息熵

1. 欧式距离(Euclidean Distance)

欧式距离是最易于理解的一种距离计算方法，也称欧几里得距离，源自欧式空间中两点的距离公式，是指在m维空间两点之间的真实距离，欧式距离在机器学习中使用的范围比较广，也比较通用，如利用k-means对二维空间内的点进行聚类。

1.1. 二维空间的欧式距离

二维空间的两点$a(x_1,y_1)$与$b(x_2,y_2)$之间的欧氏距离

$$d_{12}=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

Python实现:

```
def euclidean2(a, b):
    distance = sqrt( (a[0]-b[0])**2 + (a[1]-b[1])**2 )
    return distance
print ('a, b两点之间的欧式距离为:  ', euclidean2((1,1),(2,2)))
```

1.2. 三维空间的欧氏距离

三维空间的两点$a(x_1,y_1)$与$b(x_2,y_2)$之间的欧氏距离

$$d_{1,2}=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_i-z_2)^2}$$

Python实现:

```
def euclidean3(a, b):
    distance = sqrt( (a[0]-b[0])**2 + (a[1]-b[1])**2 + (a[2]-b[2])**2 )
    return distance
print ('a, b两点之间的欧式距离为:  ', euclidean3((1,1,1),(2,2,2)))
```

1.3. 多维空间的欧氏距离

多维空间的两点$a(x_{1,1}\cdots,x_{1n})$与$b(x_{2,1},\cdots,x_{2n})$之间的欧氏距离

$$d_{1,2}=\sqrt{\sum_{k=1}^{n}(x_{1k}-x_{2k})^2}$$

Python实现:

```python
def euclideann(a, b):
    sum = 0
    for i in range(len(a)):
        sum += (a[i]-b[i])** 2
    distance = np.sqrt(sum)
    return distance
print ('n 维空间a, b两点之间的欧式距离为:  ', euclideann((1,1,2,2),(2,2,4,4)))

def euclideann2(a, b):
    """
    不使用循环
    """
    A = np.array(a)
    B = np.array(b)
    c = (A - B) ** 2
    distance = np.sqrt(sum(c))
    return distance
print ('n 维空间a, b两点之间的欧式距离为:  ', euclideann2((1,1,2,2),(2,2,4,4)))
```

这里可以传入任意纬度空间的点

1.4. 标准化欧式距离 (Standardized Euclidean Distance)

在长方体区域进行聚类的时候，普通的距离计算公式无法满足需求，按照普通距离计算后进行聚类出的大多数是圆形区域，这时候需要采用标准化欧氏距离计算公式。

标准欧氏距离的定义

标准化欧氏距离是针对简单欧氏距离的缺点而作的一种改进方案。

标准欧氏距离的思路：既然数据各维分量的分布不一样，好吧！那我先将各个分量都“标准化”到均值、方差相等吧。

均值和方差标准化到多少呢？

这里先复习点统计学知识吧，假设样本集X的均值(mean)为m，标准差(standard deviation)为s，那么X的“标准化变量”表示为：

$$X^* = \frac{X-m}{s}$$

而且标准化变量的数学期望为0，方差为1。

因此样本集的标准化过程(standardization)用公式描述就是：

标准化后的值 = (标准化前的值 － 分量的均值) /分量的标准差

经过简单的推导就可以得到两个n维向量$a(x_{11},x_{12},\cdots,x_{1n})$与$b(x_{21},x_{22},\cdots,x_{2n})$间的标准化欧氏距离的公式：

$$d_{12} = \sqrt{\sum_{k=1}^{n}(\frac{x_{1k}-x_{2k}}{S_k})^2}$$

其中S_k是分量的标准差

标准差公式: $S = \sqrt{\frac{\sum_{i=1}^{n}(s_i-\bar{s})^2}{n}}$

如果将方差的倒数看成是一个权重，这个公式可以看成是一种**加权欧氏距离(Weighted Euclidean distance)**。

Python实现:

```
def euclideans(a, b):
    """
    标准化欧氏距离
    """
    sumnum = 0
    for i in range(len(a)):
        # 计算si 分量标准差
        avg = (a[i] - b[i]) / 2
        si = np.sqrt( (a[i]-avg)**2 + (b[i]-avg)**2 )
        sumnum += ((a[i]-b[i]) / si) **2
    distance = np.sqrt(sumnum)
    return distance
print ('a, b两点的标准化欧氏距离为:  ', euclideann2((1,2,1,2),(3,3,3,4)))
```

2. 曼哈顿距离(Manhattan Distance)

从名字就可以猜出这种距离的计算方法了。想象你在曼哈顿要从一个十字路口开车到另外一个十字路口，驾驶距离是两点间的直线距离吗？显然不是，除非你能穿越大楼。实际驾驶距离就是这个"曼哈顿距离"。而这也是曼哈顿距离名称的来源， 曼哈顿距离也称为**城市街区距离(City Block distance)**。

2.1. 二维空间的曼哈顿距离

二维空间的两点$a(x_1,y_1)$与$b(x_2,y_2)$之间的曼哈顿距离

$$d_{12} = |x_1 - x_2| + |y_1 - y_2|$$

Python实现:

```
def manhattan2(a, b):
    """
    二维空间曼哈顿距离
    """
    distance = np.abs(a[0] - b[0]) + np.abs(a[1] - b[1])
    return distance
print ('二维空间a, b两点之间的曼哈顿距离为:  ', manhattan((1,1),(2,2)))
```

2.2. 多维空间的曼哈顿距离

多维空间的两点$a(x_{1,1}\cdots,x_{1n})$与$b(x_{2,1},\cdots,x_{2n})$之间的欧氏距离

$$d_{12} = \sum_{k=1}^{n} |x_{1k} - x_{2k}|$$

Python实现:

```python
def manhattann(a, b):
    """
    n维空间曼哈顿距离
    """
    distance = 0
    for i in range(len(a)):
        distance += np.abs(a[i]-b[i])
    return distance
print ('n维空间a, b两点之间的曼哈顿距离为:  ', manhattann((1,1,2,2),(2,2,4,4)))

def manhattann2(a, b):
    """
    n维空间曼哈顿距离, 不使用循环
    """
    A = np.array(a)
    B = np.array(b)
    distance = sum(np.abs(A-B))
    return distance
print ('n维空间a, b两点之间的曼哈顿距离为:  ', manhattann2((1,1,2,2),(2,2,4,4)))
```

由于维距离计算是比较灵活的，所以也同样适合二维和三维。

3. 切比雪夫距离(Chebyshev Distance)

玩过国际象棋的都知道，国王走一步能够移动到相邻的8个方格中的任意一个。那么国王从格子(x_1, y_1)走到格子(x_2, y_2)最少需要多少步？你会发现最少步数总是$max(|x_2 - x_1|, |y_2 - y_1|)$步 。有一种类似的一种距离度量方法叫切比雪夫距离。这篇文章中**曼哈顿距离**，**欧式距离**，**明式距离**，**切比雪夫距离**的区别 给了一个很形象的解释如下：

比如，有同样两个人，在纽约准备到北京参拜天安门，同一个地点出发的话，按照欧式距离来计算，是完全一样的。

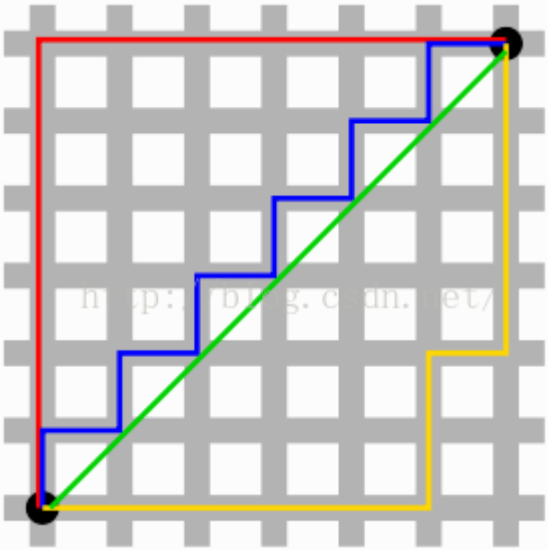
但是按照切比雪夫距离，这是完全不同的概念了。

譬如，其中一个人是土豪，另一个人是中产阶级，第一个人就能当晚直接头等舱走人，而第二个人可能就要等机票什么时候打折再去，或者选择坐船什么的。

这样来看的话，距离是不是就不一样了呢？

或者还是不清楚，我再说的详细点。

同样是这两个人，欧式距离是直接算最短距离的，而切比雪夫距离可能还得加上财力，比如第一个人财富值100，第二个只有30，虽然物理距离一样，但是所包含的内容却是不同的。



如图: 红蓝黄皆为曼哈顿距离，绿色为欧式距离

3.1. 二维切比雪夫距离

二维空间的两点$a(x_1,y_1)$与$b(x_2,y_2)$之间的切比雪夫距离

$$d_{12}=max(|x_1-x_2|,|y_1-y_2|)$$

Python实现:

```
def chebyshev2(a, b):
    """
    二维空间切比雪夫距离
    """
    distance = max(abs(a[0]-b[0]), abs(a[1]-b[1]))
    return distance
print ('二维空间a, b两点之间的欧式距离为:  ', chebyshev2((1,2),(3,4)))
```

3.2. 多维切比雪夫距离

多维空间的两点$a(x_{1,1}\cdots,x_{1n})$与$b(x_{2,1},\cdots,x_{2n})$之间的切比雪夫距离

$$d_{12}=\max_i(|x_{1i}-x_{2i}|)$$

该公式等价：

$$d_{12}=\lim_{k\to\infty}(\sum_{i=1}^{n}|x_{1i}-x_{2i}|^k)^{\frac{1}{k}}$$

(可以用放缩法和夹逼法则来证明)

Python实现:

```python
def chebyshevn(a, b):
    """
    n维空间切比雪夫距离
    """
    distance = 0
    for i in range(len(a)):
        if (abs(a[i]-b[i]) > distance):
            distance = abs(a[i]-b[i])
    return distance
print ('n维空间a, b两点之间的切比雪夫距离为: ' , chebyshevn((1,1,1,1),(3,4,3,4)))

def chebyshevn2(a, b):
    """
    n维空间切比雪夫距离, 不使用循环
    """
    distance = 0
    A = np.array(a)
    B = np.array(b)
    distance = max(abs(A-B))
    return distance

print ('n维空间a, b两点之间的切比雪夫距离为: ' , chebyshevn2((1,1,1,1),(3,4,3,4)))
```

4. 闵可夫斯基距离(Minkowski Distance)

简称为闵氏距离

闵氏距离不是一种距离，而是一组距离的定义

4.1. 定义

n维空间的两点$a(x_{1,1}\cdots,x_{1n})$与$b(x_{2,1},\cdots,x_{2n})$之间的闵可夫斯基距离定义为：

$$d_{12}=\sqrt[p]{\sum_{k=1}^{n}|x_{1k}-x_{2k}|^{p}}$$

其中p是一个变参数。

当p=1时，就是曼哈顿距离

当p=2时，就是欧氏距离

当p→∞时，就是切比雪夫距离

根据变参数的不同，闵氏距离可以表示一类的距离。

4.2. 闵可夫斯基距离缺点

闵氏距离，包括曼哈顿距离、欧氏距离和切比雪夫距离都存在明显的缺点。

举个例子：二维样本(身高,体重)，其中身高范围是150 ~ 190，体重范围是50 ~ 60，有三个样本：a(180,50)，b(190,50)，c(180,60)。那么a与b之间的闵氏距离（无论是曼哈顿距离、欧氏距离或切比雪夫距离）等于a与c之间的闵氏距离，但是身高的10cm真的等价于体重的10kg么？因此用闵氏距离来衡量这些样本间的相似度很有问题。

简单说来，闵氏距离的缺点主要有两个：(1)将各个分量的量纲(scale)，也就是"单位"当作相同的看待了。(2)没有考虑各个分量的分布（期望，方差等)可能是不同的。

Python实现:

```
def minkowski(a, b):
    """
    闵可夫斯基距离
    """
    A = np.array(a)
    B = np.array(b)
    #方法一：根据公式求解
    distance1 = np.sqrt(np.sum(np.square(A-B)))

    #方法二：根据scipy库求解
    from scipy.spatial.distance import pdist
    X = np.vstack([A,B])
    distance2 = pdist(X)[0]
    return distance1, distance2
print ('二维空间a, b两点之间的闵可夫斯基距离为：' , minkowski((1,1),(2,2))[0])
```

5. 马氏距离(Mahalanobis Distance)

有M个样本向量X_1 X_m，协方差矩阵记为S，均值记为向量μ，则其中样本向量X到u的马氏距离表示为

$$D(X) = \sqrt{(X-\mu)^T S^{-1}(X-\mu)}$$

而其中向量X_i与X_j之间的马氏距离定义为：

$$D(X_i, X_j) = \sqrt{(X_i-X_j)^T S^{-1}(X_i-X_j)}$$

若协方差矩阵是单位矩阵（各个样本向量之间独立同分布）,则公式就成了：

$$D(X_i, X_j) = \sqrt{(X_i - X_j)^T (X_i - X_j)}$$

也就是欧氏距离了。

若协方差矩阵是对角矩阵，公式变成了标准化欧氏距离。

马氏距离的优缺点：量纲(scale)无关，排除变量之间的相关性的干扰。

Python实现:

```python
def mahalanobis (a, b):
    """
    马氏距离
    """
    A = np.array(a)
    B = np.array(b)
    #马氏距离要求样本数要大于维数，否则无法求协方差矩阵
    #此处进行转置，表示10个样本，每个样本2维
    X = np.vstack([A,B])
    XT = X.T

    #方法一：根据公式求解
    S = np.cov(X)    #两个维度之间协方差矩阵
    SI = np.linalg.inv(S) #协方差矩阵的逆矩阵
    #马氏距离计算两个样本之间的距离，此处共有10个样本，两两组合，共有45个距离。
    n = XT.shape[0]
    distance1 = []
    for i in range(0, n):
        for j in range(i+1, n):
            delta = XT[i] - XT[j]
            d = np.sqrt(np.dot(np.dot(delta,SI),delta.T))
            distance1.append(d)

    #方法二：根据scipy库求解
    from scipy.spatial.distance import pdist
    distance2 = pdist(XT,'mahalanobis')
    return  distance1, distance2
print ('(1, 2), (1, 3), (2, 2), (3, 1)两两之间的闵可夫斯基距离为：' , mahalanobis((1, 1, 2, 3),(2, 3, 2, 1))[0])
```

6. 夹角余弦(Cosine)

几何中夹角余弦可用来衡量两个向量方向的差异，机器学习中借用这一概念来衡量样本向量之间的差异。

6.1. 二维空间向量的夹角余弦相似度

在二维空间中向量$A(x_1,y_1)$与向量$B(x_2,y_2)$的夹角余弦公式：

$$cos\theta = \frac{x_1x_2+y_1y_2}{\sqrt{x_1{}^2+y_1{}^2}\sqrt{x_2{}^2+y_2{}^2}}$$

Python实现:

```
def cos2(a, b):
    cos = (a[0]*b[0] + a[1]*b[1]) / (np.sqrt(a[0]**2 + a[1]**2) * np.sqrt(b[0]**2+b[1]**2))
    return cos
print ('a,b 二维夹角余弦距离: ',cos2((1,1),(2,2)))
```

6.2. 多维空间向量的夹角余弦相似度

两个n维样本点$a(x_{1,1}\cdots,x_{1n})$与$b(x_{2,1},\cdots,x_{2n})$之间的夹角余弦

可以使用类似于夹角余弦的概念来衡量这两个样本点间的相似程度。

$$cos\theta = \frac{a\cdot b}{|a||b|}$$

即:

$$cos\theta = \frac{\sum_{k=1}^{n} x_{1k}x_{2k}}{\sqrt{\sum_{k=1}^{n} x_{1k}^2}\sqrt{\sum_{k=1}^{n} x_{2k}^2}}$$

Python实现:

```
def cosn(a, b):
    """
    n维夹角余弦
    """
    sum1 = sum2 = sum3 = 0
    for i in range(len(a)):
        sum1 += a[i] * b[i]
        sum2 += a[i] ** 2
        sum3 += b[i] ** 2
    cos = sum1 / (np.sqrt(sum2) * np.sqrt(sum3))
    return cos
print ('a,b 多维夹角余弦距离: ',cosn((1,1,1,1),(2,2,2,2)))

def cosn2(a, b):
    """
    n维夹角余弦, 不使用循环
    """
    A, B = np.array(a), np.array(b)
    sum1 = sum(A * B)
    sum2 = np.sqrt(np.sum(A**2))
    sum3 = np.sqrt(np.sum(B**2))
    cos = sum1 / (sum2 * sum3)
    return cos
print ('a,b 多维夹角余弦距离: ',cosn2((1,1,1,1),(2,2,2,2)))
```

夹角余弦取值范围为(-1,1)。夹角余弦越大表示两个向量的夹角越小，夹角余弦越小表示两向量的夹角越大。当两个向量的方向重合时夹角余弦取最大值1，当两个向量的方向完全相反夹角余弦取最小值-1。

7. 汉明距离(Hamming distance)

7.1. 定义

两个等长字符串s1与s2之间的汉明距离定义为将其中一个变为另外一个所需要作的最小替换次数。例如字符串“1111”与“1001”之间的汉明距离为2。

应用：信息编码（为了增强容错性，应使得编码间的最小汉明距离尽可能大）。

Python实现:

```
def hamming(a, b):
    """
    汉明距离
    """
    sumnum = 0
    for i in range(len(a)):
        if a[i]!=b[i]:
            sumnum += 1
    return sumnum
print ('a,b 汉明距离: ',hamming((1,1,2,3),(2,2,1,3)))

def hamming2(a, b):
    """
    汉明距离, 不使用循环
    """
    matV = np.array(a) - np.array(b)
    numsum = len(np.nonzero(matV)[0])
    return numsum
print ('a,b 汉明距离: ',hamming2((1,1,2,3),(2,2,1,3)))
```

8. 杰卡德相似系数(Jaccard similarity coefficient) & 杰卡德距离(Jaccard distance)

8.1. 杰卡德相似系数

两个集合A和B的交集元素在A，B的并集中所占的比例，称为两个集合的杰卡德相似系数，用符号$J(A,B)$表示。

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

杰卡德相似系数是衡量两个集合的相似度一种指标。

Python实现:

```
def jaccard_coefficient(a, b):
    """
    杰卡德相似系数
    """
    set_a = set(a)
    set_b = set(b)
    distance = float(len(set_a & set_b)) / len(set_a | set_b)
    return distance
print ('a,b 杰卡德相似系数：', jaccard_coefficient((1,2,3),(2,3,4)))
```

8.2. 杰卡德距离

与杰卡德相似系数相反的概念是杰卡德距离(Jaccard distance)。杰卡德距离可用如下公式表示：

$$J_\delta(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

杰卡德距离用两个集合中不同元素占所有元素的比例来衡量两个集合的区分度。

Python实现:

```
def jaccard_distance(a, b):
    """
    杰卡德距离
    """
    set_a = set(a)
    set_b = set(b)
    distance = float(len(set_a | set_b) - len(set_a & set_b)) / len(set_a | set_b)
    return distance
print ('a,b 杰卡德距离: ', jaccard_coefficient((1,2,3),(2,3,4)))
```

8.3. 杰卡德相似系数与杰卡德距离的应用

可将杰卡德相似系数用在衡量样本的相似度上。

样本A与样本B是两个n维向量，而且所有维度的取值都是0或1。例如：A(0111)和B(1011)。我们将样本看成是一个集合，1表示集合包含该元素，0表示集合不包含该元素。

p ：样本A与B都是1的维度的个数

q ：样本A是1，样本B是0的维度的个数

r ：样本A是0，样本B是1的维度的个数

s ：样本A与B都是0的维度的个数

那么样本A与B的杰卡德相似系数可以表示为：

这里p+q+r可理解为A与B的并集的元素个数，而p是A与B的交集的元素个数。

而样本A与B的杰卡德距离表示为：

$$J = \frac{p}{p+q+r}$$

9. 相关系数（Correlation coefficient）与相关距离(Correlation distance)

9.1. 相关系数的定义

$$\rho_{XY} = \frac{Cov(X,Y)}{\sqrt{D(X)}\sqrt{D(Y)}} = \frac{E((X-EX)(Y-EY))}{\sqrt{D(X)}\sqrt{D(Y)}}$$

相关系数是衡量随机变量X与Y相关程度的一种方法，相关系数的取值范围是(-1,1)。相关系数的绝对值越大，则表明X与Y相关度越高。当X与Y线性相关时，相关系数取值为1（正线性相关）或-1（负线性相关）。

Python 实现：

相关系数可以利用`numpy`库中的`corrcoef`函数来计算 例如 对于矩阵`a,numpy.corrcoef(a)` 可计算行与行之间的相关系数，`numpy.corrcoef(a,rowvar=0)`用于计算各列之间的相关系数，输出为相关系数矩阵。

```
def correlation_coefficient():
    """
    相关系数
    """
    a = np.array([[1, 1, 2, 2, 3], [2, 2, 3, 3, 5], [1, 4, 2, 2, 3]])
    print ('a的行之间相关系数为：  ', np.corrcoef(a))
    print ('a的列之间相关系数为：  ', np.corrcoef(a,rowvar=0))
correlation_coefficient()
```

9.2. 相关距离的定义

$$D_{xy} = 1 - \rho_{XY}$$

Python 实现：（基于相关系数） 同样针对矩阵a

```
def correlation_distance():
    """
    相关距离
    """
    a = np.array([[1, 1, 2, 2, 3], [2, 2, 3, 3, 5], [1, 4, 2, 2, 3]])
    print ('a的行之间相关距离为：  ', np.ones(np.shape(np.corrcoef(a)),int) - np.corrcoef(a))
    print ('a的列之间相关距离为：  ', np.ones(np.shape(np.corrcoef(a,rowvar = 0)),int) - np.corrcoef(a,rowvar = 0))
correlation_distance()
```

10. 信息熵(Information Entropy)

信息熵也成香农(shanno)熵。信息熵并不属于一种相似性度量，是衡量分布的混乱程度或分散程度的一种度量。分布越分散(或者说分布越平均)，信息熵就越大。分布越有序（或者说分布越集中），信息熵就越小。

计算给定的样本集X的信息熵的公式：

$$Entropy(X) = \sum_{i=1}^{n} -p_i log_2 p_i$$

参数的含义：

n：样本集X的分类数

pi：X中第`i`类元素出现的概率

信息熵越大表明样本集S分类越分散，信息熵越小则表明样本集X分类越集中。当S中n个分类出现的概率一样大时（都是1/n），信息熵取最大值log2(n)。当X只有一个分类时，信息熵取最小值0

Python实现：

```
def calc_entropy(x):
    """
    计算信息熵
    """
    x_value_list = set([x[i] for i in range(x.shape[0])])
    ent = 0.0
    for x_value in x_value_list:
        p = float(x[x == x_value].shape[0]) / x.shape[0]
        logp = np.log2(p)
        ent -= p * logp
    return ent

def calc_condition_entropy(x, y):
    """
    计算条件信息熵
    """

    # calc ent(y|x)
    x_value_list = set([x[i] for i in range(x.shape[0])])
    ent = 0.0
    for x_value in x_value_list:
        sub_y = y[x == x_value]
        temp_ent = calc_entropy(sub_y)
        ent += (float(sub_y.shape[0]) / y.shape[0]) * temp_ent
    return ent

def calc_entropy_grap(x,y):
    """
    计算信息增益
    """
```

```
31
32      base_ent = calc_entropy(y)
33      condition_ent = calc_condition_entropy(x, y)
34      ent_grap = base_ent - condition_ent
35      return ent_grap
```

> 参考
>
> http://blog.csdn.net/gamer_gyt/article/details/75165842
>
> http://www.cnblogs.com/heaad/archive/2011/03/08/1977733.html