K近邻法(k-nearest neighbor,k-NN)

k近邻法

k近邻算法定义：定义：给定一个训练数据集，对新输入的实例，在训练数据集中找到与该实例最邻近的k个实例，这k个实例的多数属于某个类，就把该输入实例分为这个类

一种基本分类、回归的监督算法

懒惰学习(lazy learning)：没有显示的训练过程

k近邻算法三要素

距离度量

闵科夫斯基距离(Minkowski Distance) -- Lp距离

$$L_p(x_i,x_j)=\left(\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}|^p\right)^{\frac{1}{p}}$$

欧式距离(p = 2)

$$L_2(x_i,x_j)=\left(\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}|^2\right)^{\frac{1}{2}}$$

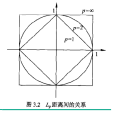

图 3.2 L_p距离为1时关系

曼哈顿距离(p = 1)

切比雪夫距离(p = 无穷)

$$D_{\text{Chebyshev}}(p,q):=\max_i(|p_i-q_i|)$$

$$\lim_{k\to\infty}\left(\sum_{i=1}^{n}|p_i-q_i|^k\right)^{1/k}$$

k值选择

选择较小：学习的近似误差(approximation error)会减小，学习的估计误差(estimation error)会增大

与输入实例较近的训练实例对预测结果起作用

k值减小则整体模型变复杂，容易过拟合

选择较大：减小学习的估计误差，学习的近似误差会增大

输入实例较远的训练实例对预测结果起作用

k值增大则整体模型变简单

k值一般取一个比较小的值

采用交叉验证法选取最优k值

分类决策规则：多数表决规则(majority voting rule)

多数表决规则等价于经验风险最小化

算法 3.1（k 近邻法）

输入：训练数据集

$$T=\{(x_1,y_1),(x_2,y_2),\cdots,(x_N,y_N)\}$$

其中，$x_i\in\mathcal{X}\subseteq\mathbf{R}^n$ 为实例的特征向量，$y_i\in\mathcal{Y}=\{c_1,c_2,\cdots,c_K\}$ 为实例的类别，$i=1,2,\cdots,N$；实例特征向量 x；

输出：实例 x 所属的类 y。

（1）根据给定的距离度量，在训练集 T 中找出与 x 最邻近的 k 个点，涵盖这 k 个点的 x 的邻域记作 $N_k(x)$；

（2）在 $N_k(x)$ 中根据分类决策规则（如多数表决）决定 x 的类别 y：

$$y=\arg\max_{c_j}\sum_{x_i\in N_k(x)}I(y_i=c_j),\quad i=1,2,\cdots,N;\quad j=1,2,\cdots,K \tag{3.1}$$

式 (3.1) 中，I 为指示函数，即当 $y_i=c_j$ 时 I 为 1，否则 I 为 0。 ■

k 近邻法的特殊情况是 $k=1$ 的情形，称为最近邻算法。对于输入的实例点（特征向量）x，最近邻法将训练数据集中与 x 最邻近点的类作为 x 的类。

用kd树的最近邻搜索

算法 3.3（用 kd 树的最近邻搜索）

输入：已构造的 kd 树；目标点 x；

输出：x 的最近邻。

（1）在 kd 树中找出包含目标点 x 的叶结点：从根结点出发，递归地向下访问 kd 树。若目标点 x 当前维的坐标小于切分点的坐标，则移动到左子结点，否则移动到右子结点。直到子结点为叶结点为止。

（2）以此叶结点为“当前最近点”。

（3）递归地向上回退，在每个结点进行以下操作：

（a）如果该结点保存的实例点比当前最近点距离目标点更近，则以该实例点为“当前最近点”。

（b）当前最近点一定存在于该结点一个子结点对应的区域。检查该子结点的父结点的另一子结点对应的区域是否有更近的点。具体地，检查另一子结点对应的区域是否与以目标点为球心、以目标点与“当前最近点”间的距离为半径的超球体相交。

如果相交，可能在另一个子结点对应的区域内存在距目标点更近的点，移动到另一个子结点。接着，递归地进行最近邻搜索；

如果不相交，向上回退。

（4）当回退到根结点时，搜索结束。最后的“当前最近点”即为 x 的最近邻点。 ■

如果实例点是随机分布的，kd 树搜索的平均计算复杂度是 $O(\log N)$，这里 N 是训练实例数。kd 树更适用于训练实例数远大于空间维数时的 k 近邻搜索。当空间维数接近训练实例数时，它的效率会迅速下降，几乎接近线性扫描。

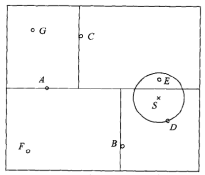

图 3.5 通过 kd 树搜索最近邻

构造平衡kd树

算法 3.2（构造平衡 kd 树）

输入：k 维空间数据集 $T=\{x_1,x_2,\cdots,x_N\}$，

其中 $x_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(k)})^{\mathrm{T}}$，$i=1,2,\cdots,N$；

输出：kd 树。

（1）开始：构造根结点，根结点对应于包含 T 的 k 维空间的超矩形区域。

选择 $x^{(1)}$ 为坐标轴，以 T 中所有实例的 $x^{(1)}$ 坐标的中位数为切分点，将根结点对应的超矩形区域切分为两个子区域。切分由通过切分点并与坐标轴 $x^{(1)}$ 垂直的超平面实现。

由根结点生成深度为 1 的左、右子结点：左子结点对应坐标 $x^{(1)}$ 小于切分点的子区域，右子结点对应于坐标 $x^{(1)}$ 大于切分点的子区域。

将落在切分超平面上的实例点保存在根结点。

（2）重复：对深度为 j 的结点，选择 $x^{(l)}$ 为切分的坐标轴，$l=j(\mathrm{mod}\,k)+1$，以该结点的区域中所有实例的 $x^{(l)}$ 坐标的中位数为切分点，将该结点对应的超矩形区域切分为两个子区域。切分由通过切分点并与坐标轴 $x^{(l)}$ 垂直的超平面实现。

由该结点生成深度为 $j+1$ 的左、右子结点：左子结点对应坐标 $x^{(l)}$ 小于切分点的子区域，右子结点对应坐标 $x^{(l)}$ 大于切分点的子区域。

将落在切分超平面上的实例点保存在该结点。

（3）直到两个子区域没有实例存在时停止。从而形成 kd 树的区域划分。 ■

伪代码

```
算法：构建k-d树（createKDTree）
输入：数据点集Data-set和其所在的空间Range
输出：Kd，类型为k-d tree
1.If Data-set为空，则返回空的k-d tree
2.调用节点生成程序：
    （1）确定split域：对于所有描述子数据（特征矢量），统计它们在每个维上的数据方差。假设每条数据记录为64维，可计算64个方差。挑选出最大值，对应的维就是split域的值。数据方差大表明沿该坐标轴方向上的数据分散得比较开，在这个方向上进行数据分割有较好的分辨率；
    （2）确定Node-data域：数据点集Data-set按其第split域的值排序。位于正中间的那个数据点被选为Node-data。此时新的Data-set' = Data-set \ Node-data（除去其中Node-data这一点）。
3.dataleft = {d属于Data-set' && d[split] ≤ Node-data[split]}
    Left_Range = {Range && dataleft}
  dataright = {d属于Data-set' && d[split] > Node-data[split]}
    Right_Range = {Range && dataright}
4.left = 由（dataleft，Left_Range）建立的k-d tree，即递归调用createKDTree（dataleft，Left_Range）。并设置left的parent域为Kd；
    right = 由（dataright，Right_Range）建立的k-d tree，即调用createKDTree（dataleft，Left_Range）。并设置right的parent域为Kd。
```

伪代码

```
算法：k-d树最邻近查找
输入：Kd，     //k-d tree类型
    target  //查询数据点
输出：nearest，  //最邻近数据点
    dist      //最邻近数据点和查询点间的距离
1. If Kd为NULL，则设dist为infinite并返回
2. //进行二叉查找，生成搜索路径
   Kd_point = &Kd;                          //Kd-point中保存k-d tree根节点地址
   nearest = Kd_point -> Node-data;   //初始化最近邻点
   while（Kd_point）
      push（Kd_point）到search_path中；  //search_path是一个堆栈结构，存储着搜索路径节点指针
 /*** If Dist（nearest，target） > Dist（Kd_point -> Node-data，target）
         nearest  = Kd_point -> Node-data;     //更新最近邻点
         Max_dist = Dist(Kd_point，target)；  //更新最近邻点与查询点间的距离  ***/
      s = Kd_point -> split;                        //确定待分割的方向
      If target[s] <= Kd_point -> Node-data[s]     //进行二叉查找
         Kd_point = Kd_point -> left;
      else
         Kd_point = Kd_point ->right;
   nearest = search_path中最后一个叶子节点；  //注意：二叉搜索时不比计算选择搜索路径中的最邻近点，这部分已被注释
   Max_dist = Dist（nearest，target）；      //直接取最后叶子节点作为回溯前的初始最近邻点

3. //回溯查找
   while（search_path != NULL）
      back_point = 从search_path取出一个节点指针；    //从search_path堆栈弹栈
      s = back_point -> split;                    //确定分割方向
      If Dist（target[s]，back_point -> Node-data[s]）  < Max_dist   //判断还需进入的子空间
         If target[s] <= back_point -> Node-data[s]
            Kd_point = back_point -> right;   //如果target位于左子空间，就应进入右子空间
         else
            Kd_point = back_point -> left;    //如果target位于右子空间，就应进入左子空间
         将Kd_point压入search_path堆栈；
      If Dist（nearest，target）  > Dist（Kd_Point -> Node-data，target）
         nearest  = Kd_point -> Node-data;                //更新最近邻点
         Min_dist = Dist（Kd_point -> Node-data,target）；   //更新最近邻点与查询点间的距离
```