k近邻法

3.1 k近邻算法

- **原理**：给定一个训练数据集T，对新的输入实例x，在训练数据集中找到与其最近邻的k个实例，这个k个实例的多数属于某个类就把输入实例分为这个类；
  - 根据给定的距离度量，在训练集T中找出与x最近邻k个点，并把这个邻域记作$N_k(x)$；
  - 在$N_k(x)$中根据分类决策规则决定类别：

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

  其中$I(\cdot)$是指示函数，$y_i = c_j$时为1，否则为0；
- 特殊情况：$k = 1$，最近邻算法；
- k近邻算法没有*显式的学习过程*；

3.2 k近邻模型

> k近邻算法实质是对特征空间的划分，包括三要素--距离度量、k值选择和分类决策规则

3.2.1 模型

- 特征空间中，对每个训练实例点x_i，距离该点比其他店更近的所有点组成一个区域，叫做单元（cell）；

3.2.2 距离度量

- 特征空间中两个实例点的距离是其相似程度的反映；
- 特征空间一般是n维实数向量空间$\mathbb{R}^n$，x_i, x_j的L_p**距离**（$p \geq 1$）或者叫**闵可夫斯基距离**（Minkowski Distance）定义为：

$$L_p(x_i, x_j) = \left(\sum_{d=1}^{n} |x_i^{(d)} - x_j^{(d)}|^p \right)^{\frac{1}{p}} \tag{2}$$

- 当$p = 2$时，成为**欧式距离**（Euclidean distance）：

$$L_2(x_i, x_j) = \left(\sum_{d=1}^{n} |x_i^{(d)} - x_j^{(d)}|^2 \right)^{\frac{1}{2}} \tag{3}$$

- 当$p = 1$时，称为**曼哈顿距离**（Manhattan distance）：

$$L_1(x_i, x_j) = \sum_{d=1}^{n} |x_i^{(d)} - x_j^{(d)}| \tag{4}$$

- 当$p = \infty$时，成为**切比雪夫距离**（Chebyshev distance），是各个坐标距离的最大值：

$$L_\infty(x_i, x_j) = \max_d |x_i^{(d)} - x_j^{(d)}| \tag{5}$$

3.2.3 k值选择

- **较小的k值**：相当于用较小邻域的训练实例进行预测，学习的近似误差（approximation error）会减少，但估计误差（estimation error）会增大，预测结果对近邻点非常敏感；k的减少意味着模型变得复杂，容易发生过拟合；
- **较大的k值**：相当于用较大邻域的训练实例进行预测，可以减少估计误差，但是近似误差增大，较远的点也会对预测起作用，造成预测错误；k的增大意味着模

型变得简单；
- $k=N$：无论如何输入都简单地预测为训练集最多的类，过于简单，不可取；
- 通常k取一个较小值，并通过交叉验证择优；

3.2.4 分类决策准则

- **多数表决规则**（majority voting rule）：可以证明等价于经验风险最小化（误分类率）

3.3 算法实现 - kd树

> 实现k近邻算法要考虑的问题是对训练数据进行快速k近邻搜索，尤其是特征空间和训练数据容量都很大是；
>
> 最简单的线性扫描方法需要逐个计算距离，非常耗时；

3.3.1 构造kd树

- **kd树**：是二叉树，表示对n维空间的一个划分（partition）；构造kd树相当于不断地用垂直于坐标轴的超平面将n维空间切分，构成一系列n维超矩形区域；
- **构造平衡kd树算法**：
  - 输入n维空间数据集T；
  - 构造**根结点**（根节点对应于包含T的n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*中位数*为切分点，由通过切分点且与$x^{(1)}$坐标轴垂直的超平面把超矩形区域切成两个子区域；
    - 将落在切分超平面上的实例点保存为根结点；
    - 由根结点生成深度为1的左、右**子结点**：左子结点对应$x^{(1)}$坐标小于切分点的子区域，而右子结点对应另一个区域；
  - 重复：对于深度为j的结点，选择$x^{(d)}$为切分坐标轴，$d=j(\mathrm{mod}k)+1$，按照上述步骤类似的方法，生成深度为$j+1$的左右子结点；
  - 直到两个子区域没有实例点存在时为止，从而形成kd树；
- 平衡kd树（选择中位数为切分点）搜索时效率*未必最优*；

3.3.2 搜索kd树

- **搜索kd树算法**（最近邻搜索）：
  - 在kd树中找到包含目标点x的叶结点：从根结点出发，递归向下访问，若x当前维的坐标小于切分点坐标，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；
  - 以此叶结点作为"*当前最近点*"；
  - 递归向上回退，在*每个结点*上进行下列操作：
    - 如果该结点对应的实例点比"当前最近点"离x更近，则把该实例点作为"当前最近点"；
    - "当前最近点"一定存在于该结点的一个子结点对应的区域，因此需要检查该结点的另一个子结点对应的区域是否有更近的点；
    - 即检查另一个区域是否与*以目标点为球心，以目标点到"当前最近点"的距离为半径的超球体*相交；
    - 如果相交：可能该区域会存在更近的点，移动到另一个子结点，进入递归最近邻搜索；
    - 如果不想交：继续向上回退；
  - 当回到根结点时，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，那么kd树的*平均搜索复杂*为$O(\log N)$；
- kd树适用于*训练数据远大于空间维度*时（$N \gg n$），当两者接近时效率降低；