

k 近邻法

一、k 近邻算法

1. k 近邻法 (k-Nearest Neighbor:kNN) 是一种基本的分类与回归方法。

- 分类问题：对新的样本，根据其 k 个最近邻的训练样本的类别，通过多数表决等方式进行预测。
- 回归问题：对新的样本，根据其 k 个最近邻的训练样本标签值的均值作为预测值。

2. k 近邻法不具有显式的学习过程，它是直接预测。它是“惰性学习”(lazy learning)的著名代表。

- 它实际上利用训练数据集对特征向量空间进行划分，并且作为其分类的“模型”。
- 这类学习技术在训练阶段仅仅将样本保存起来，训练时间开销为零，等到收到测试样本后再进行处理。

那些在训练阶段就对样本进行学习处理的方法称作“急切学习”(eager learning)。

3. k 近邻法是个非参数学习算法，它没有任何参数 (k 是超参数，而不是需要学习的参数)。

- k 近邻模型具有非常高的容量，这使得它在训练样本数量较大时能获得较高的精度。
- 它的缺点有：
 - 计算成本很高。因为需要构建一个 $N \times N$ 的距离矩阵，其计算量为 $O(N^2)$ ，其中 N 为训练样本的数量。
当数据集是几十亿个样本时，计算量是不可接受的。
 - 在训练集较小时，泛化能力很差，非常容易陷入过拟合。
 - 无法判断特征的重要性。

4. k 近邻法的三要素：

- k 值选择。
- 距离度量。
- 决策规则。

1.1 k 值选择

1. 当 $k = 1$ 时的 k 近邻算法称为最近邻算法，此时将训练集中与 \mathbf{x} 最近的点的类别作为 \mathbf{x} 的分类。

2. k 值的选择会对 k 近邻法的结果产生重大影响。

- 若 k 值较小，则相当于用较小的邻域中的训练样本进行预测，“学习”的偏差减小。

只有与输入样本较近的训练样本才会对预测起作用，预测结果会对近邻的样本点非常敏感。

若近邻的训练样本点刚好是噪声，则预测会出错。即： k 值的减小意味着模型整体变复杂，易发生过拟合。

- 优点：减少“学习”的偏差。
- 缺点：增大“学习”的方差（即波动较大）。
- 若 k 值较大，则相当于用较大的邻域中的训练样本进行预测。

这时输入样本较远的训练样本也会对预测起作用，使预测偏离预期的结果。

即： k 值增大意味着模型整体变简单。

- 优点：减少“学习”的方差（即波动较小）。

- 缺点：增大"学习"的偏差。

3. 应用中， k 值一般取一个较小的数值。通常采用交叉验证法来选取最优的 k 值。

1.2 距离度量

1. 特征空间中两个样本点的距离是两个样本点的相似程度的反映。

k 近邻模型的特征空间一般是 n 维实数向量空间 \mathbb{R}^n ， k 其距离一般为欧氏距离，也可以是一般的 L_p 距离：

$$L_p(\vec{x}_i, \vec{x}_j) = \left(\sum_{l=1}^n |x_{i,l} - x_{j,l}|^p \right)^{1/p}, \quad p \geq 1$$

$$\vec{x}_i, \vec{x}_j \in \mathcal{X} = \mathbb{R}^n; \quad \vec{x}_i = (x_{i,1}, x_{i,2}, \dots, x_{i,n})^T$$

- 当 $p = 2$ 时，为欧氏距离： $L_2(\vec{x}_i, \vec{x}_j) = (\sum_{l=1}^n |x_{i,l} - x_{j,l}|^2)^{1/2}$
- 当 $p = 1$ 时，为曼哈顿距离： $L_1(\vec{x}_i, \vec{x}_j) = \sum_{l=1}^n |x_{i,l} - x_{j,l}|$
- 当 $p = \infty$ 时，为各维度距离中的最大值： $L_\infty(\vec{x}_i, \vec{x}_j) = \max_l |x_{i,l} - x_{j,l}|$

2. 不同的距离度量所确定的最近邻点是不同的。

1.3 决策规则

1.3.1 分类决策规则

1. 分类决策通常采用多数表决，也可以基于距离的远近进行加权投票：距离越近的样本权重越大。

2. 多数表决等价于经验风险最小化。

设分类的损失函数为 0-1 损失函数，分类函数为 $f: \mathbb{R}^n \rightarrow \{c_1, c_2, \dots, c_K\}$ 。

给定样本 $\vec{x} \in \mathcal{X}$ ，其最邻近的 k 个训练点构成集合 $\mathcal{N}_k(\vec{x})$ 。设涵盖 $\mathcal{N}_k(\vec{x})$ 区域的类别为 c_m （这是待求的未知量，但是它肯定是 c_1, c_2, \dots, c_K 之一），则损失函数为：

$$L = \frac{1}{k} \sum_{\vec{x}_i \in \mathcal{N}_k(\vec{x})} I(\tilde{y}_i \neq c_m) = 1 - \frac{1}{k} \sum_{\vec{x}_i \in \mathcal{N}_k(\vec{x})} I(\tilde{y}_i = c_m)$$

L 就是训练数据的经验风险。要使经验风险最小，则使得 $\sum_{\vec{x}_i \in \mathcal{N}_k(\vec{x})} I(\tilde{y}_i = c_m)$ 最大。即多数表决：

$$c_m = \arg \max_{c_m} \sum_{\vec{x}_i \in \mathcal{N}_k(\vec{x})} I(\tilde{y}_i = c_m)。$$

1.3.2 回归决策规则

1. 回归决策通常采用均值回归，也可以基于距离的远近进行加权投票：距离越近的样本权重越大。

2. 均值回归等价于经验风险最小化。

设回归的损失函数为均方误差。给定样本 $\vec{x} \in \mathcal{X}$ ，其最邻近的 k 个训练点构成集合 $\mathcal{N}_k(\vec{x})$ 。设涵盖 $\mathcal{N}_k(\vec{x})$ 区域的输出为 \hat{y} ，则损失函数为：

$$L = \frac{1}{k} \sum_{\vec{x}_i \in \mathcal{N}_k(\vec{x})} (\tilde{y}_i - \hat{y})^2$$

L 就是训练数据的经验风险。要使经验风险最小，则有： $\hat{y} = \frac{1}{k} \sum_{\vec{x}_i \in \mathcal{N}_k(\vec{x})} \tilde{y}_i$ 。即：均值回归。

1.4 k 近邻算法

1. k 近邻法的分类算法：

- 输入：

- 训练数据集 $\mathbb{D} = \{(\vec{x}_1, \tilde{y}_1), (\vec{x}_2, \tilde{y}_2), \dots, (\vec{x}_N, \tilde{y}_N)\}, \vec{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n, \tilde{y}_i \in \mathcal{Y} = \{c_1, c_2, \dots, c_K\}$
 - 给定样本 \vec{x}
 - 输出：样本 \vec{x} 所属的类别 y
 - 步骤：
 - 根据给定的距离度量，在 \mathbb{D} 中寻找与 \vec{x} 最近邻的 k 个点。定义涵盖这 k 个点的 \vec{x} 的邻域记作 $\mathcal{N}_k(\vec{x})$ 。
 - 从 $\mathcal{N}_k(\vec{x})$ 中，根据分类决策规则（如多数表决）决定 \vec{x} 的类别 y ：

$$y = \arg \max_{c_m} \sum_{\vec{x}_i \in \mathcal{N}_k(\vec{x})} I(\tilde{y}_i = c_m)。$$
2. k 近邻法的回归算法：
- 输入：
 - 训练数据集 $\mathbb{D} = \{(\vec{x}_1, \tilde{y}_1), (\vec{x}_2, \tilde{y}_2), \dots, (\vec{x}_N, \tilde{y}_N)\}, \vec{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n, \tilde{y}_i \in \mathcal{Y} \subseteq \mathbb{R}$
 - 给定样本 \vec{x}
 - 输出：样本 \vec{x} 的输出 y
 - 步骤：
 - 根据给定的距离度量，在 \mathbb{D} 中寻找与 \vec{x} 最近邻的 k 个点。定义涵盖这 k 个点的 \vec{x} 的邻域记作 $\mathcal{N}_k(\vec{x})$ 。
 - 从 $\mathcal{N}_k(\vec{x})$ 中，根据回归决策规则（如均值回归）决定 \vec{x} 的输出 y ： $y = \frac{1}{k} \sum_{\vec{x}_i \in \mathcal{N}_k(\vec{x})} \tilde{y}_i。$

二、kd树

1. 实现 k 近邻法时，主要考虑的问题是：如何对训练数据进行快速 k 近邻搜索。
2. 最简单的实现方法：线性扫描。此时要计算输入样本与每个训练样本的距离。
当训练集很大时，计算非常耗时。解决办法是：使用 kd 树来提高 k 近邻搜索的效率。
3. kd 树是一种对 k 维空间中的样本点进行存储以便对其进行快速检索的树型数据结构。
它是二叉树，表示对 k 维空间的一个划分。
4. 构造 kd 树的过程相当于不断的用垂直于坐标轴的超平面将 k 维空间切分的过程。
 kd 树的每个结点对应于一个 k 维超矩形区域。

2.1 kd树构建算法

1. 平衡 kd 树构建算法：
 - 输入： k 维空间样本集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}, \vec{x}_i \in \mathcal{X} \subseteq \mathbb{R}^k$
 - 输出： kd 树
 - 算法步骤：
 - 构造根结点。根结点对应于包含 \mathbb{D} 的 k 维超矩形。
选择 x_1 为轴，以 \mathbb{D} 中所有样本的 x_1 坐标的中位数 x_1^* 为切分点，将根结点的超矩形切分为两个子区域，切分产生深度为 1 的左、右子结点。切分超平面为： $x_1 = x_1^*$ 。
 - 左子结点对应于坐标 $x_1 < x_1^*$ 的子区域。
 - 右子结点对应于坐标 $x_1 > x_1^*$ 的子区域。
 - 落在切分超平面上的点($x_1 = x_1^*$)保存在根结点。
 - 对深度为 j 的结点，选择 x_l 为切分的坐标轴继续切分， $l = j \pmod k + 1$ 。本次切分之后，树的深度为 $j + 1$ 。

这里取模而不是 $l = j + 1$ ，因为树的深度可以超过维度 k 。此时切分轴又重复回到 x_l ，轮转坐标轴进行切分。

- 直到所有结点的两个子域中没有样本存在时，切分停止。此时形成 kd 树的区域划分。

2.2 kd 树搜索算法

1. kd 树最近邻搜索算法（ k 近邻搜索以此类推）：

◦ 输入：

- 已构造的 kd 树
- 测试点 \vec{x}

◦ 输出： \vec{x} 的最近邻测试点

◦ 步骤：

- 在 kd 树中找到包含测试点 \vec{x} 的叶结点：从根结点出发，递归向下访问 kd 树：
 - 若测试点 \vec{x} 当前维度的坐标小于切分点的坐标，则查找当前结点的左子结点。
 - 若测试点 \vec{x} 当前维度的坐标大于切分点的坐标，则查找当前结点的右子结点。

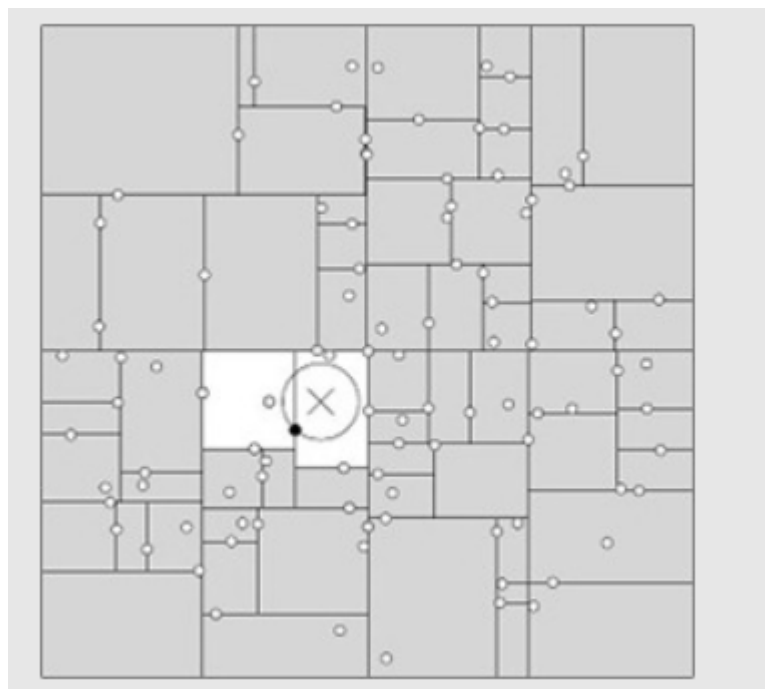
在访问过程中记录下访问的各结点的顺序，存放在先进后出队列 `Queue` 中，以便于后面的回退。

- 以此叶结点为“当前最近子结点” \vec{x}_{nst} 。真实最近点一定在 \vec{x} 与“当前最近点”构成的超球体内， \vec{x} 为球心。
- 设当前考察的结点为 \vec{x}_i ，从 `Queue` 中弹出结点，设为 \vec{x}_{inew} （每次回退都是退到 kd 树的父结点）。
 - 若 \vec{x}_{inew} 比 \vec{x}_{nst} 离 \vec{x} 更近，则更新： $\vec{x}_{nst} = \vec{x}_{inew}$ 。
 - 考察结点 \vec{x}_{inew} 所在的超平面与以 \vec{x} 为球心、以 \vec{x} 到当前最近点 \vec{x}_{nst} 的距离为半径的超球体是否相交：
 - 若相交：
 - 若 \vec{x}_i 是 \vec{x}_{inew} 的左子结点，则进入 \vec{x}_{inew} 的右子结点，然后先进行向下搜索并更新队列 `Queue`，然后向上回退。
 - 若 \vec{x}_i 是 \vec{x}_{inew} 的右子结点，则进入 \vec{x}_{inew} 的左子结点，然后先进行向下搜索并更新队列 `Queue`，然后向上回退。
 - 若不相交，则直接回退。
- 当回退到根结点时，搜索结束。最后的“当前最近点”即为 \vec{x} 的最近邻点。

2. kd 树搜索的平均计算复杂度为 $O(\log N)$ ， N 为训练集大小。

kd 树适合 $N \gg k$ 的情形，当 N 与 维度 k 接近时效率会迅速下降。

3. 通常最近邻搜索只需要检测几个叶结点即可：



但是如果样本点的分布比较糟糕时，需要几乎遍历所有的结点：

