分类问题中的 k近邻法

杨钧尹

2023-10-31

分类问题

- 一种监督学习问题，旨在对数据分类
- 将输入数据映射到预定义的类别或标签
- 从已知的训练数据中学习一个分类模型，然后将该模型应用于新的、未知的数据，以预测其所属的类别

- 垃圾邮件过滤、金融风险评估
- 医学诊断、生物信息学
- 情感分析、客户分类
- 图像识别

kNN算法 – 提出

$$T = \{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)\}$$
$$y_i \in \mathcal{Y} = \{c_1, c_2, \cdots, c_K\}$$

$$y = \arg\max_{c_j} \sum_{x_i \in N_k(x)} I(y_i = c_j),$$

- 输入：特征向量（空间点）
- 输出：类别（可以取多类）
- 已标注的训练集

- 预测：多数表决（“近朱者赤”）
- 不具有显式的学习过程

适用范围
数数值型和标称型

优点
直观、非参数化
对异常值不敏感
支持多类别

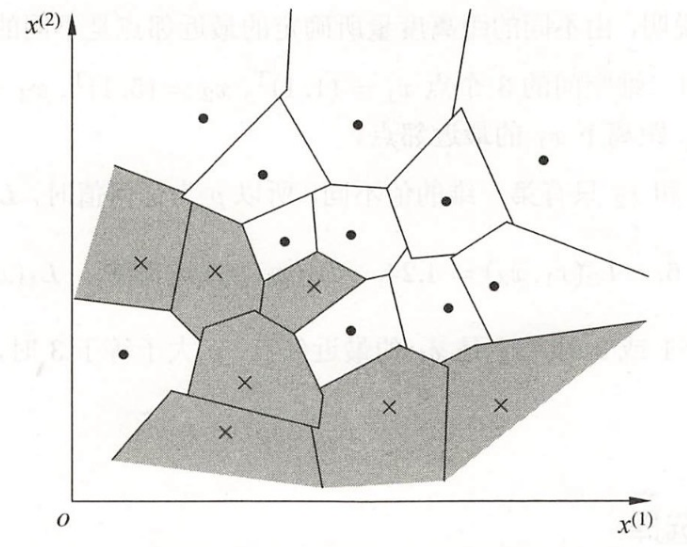
缺点
时间复杂度高
存储成本高
“维度灾难”和数据不平衡

kNN算法 – 流程

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

$$y_i\in\mathcal{Y}=\{c_1,c_2,\cdots,c_K\}$$

$$y=\arg\max_{c_j}\sum_{x_i\in N_k(x)}I(y_i=c_j),$$



（给定距离度量，k值与决策规则）

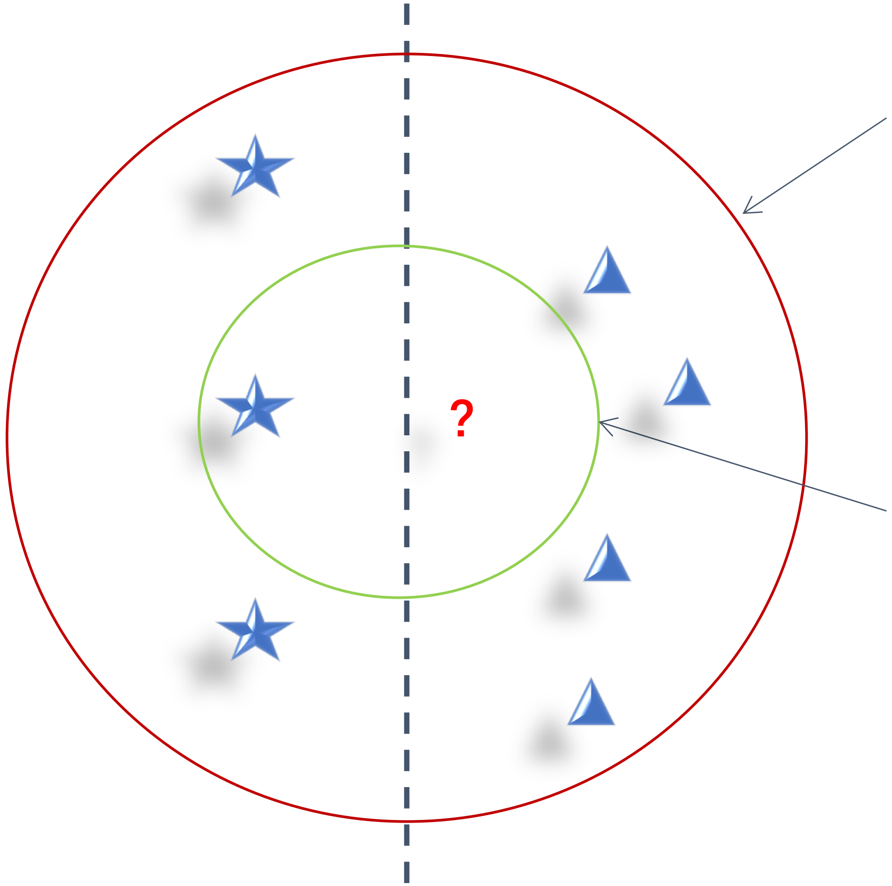
[输入训练集T]

- 在训练集 T 中找出与 x 最邻近的 k 个点，涵盖这个点的 x 的邻域记作 $N_k(a)$
- 在 $N_k(a)$中根据分类决策规则决定 x 的类别 y

基本要素
k 值选择
距离度量
决策规则

最近邻
k=1

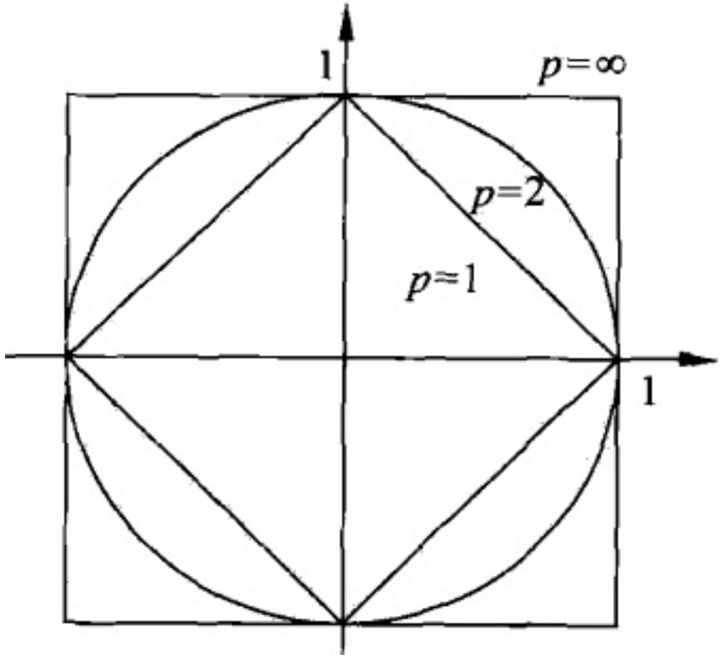
分类模型 – k值选择



偏小
近似误差减小
估计误差增大
偏大
估计误差减小
近似误差增大

交叉验证以提高泛化性能。

分类模型 – 距离度量



$$x_i = (x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(n)})^T$$

Lp距离

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

欧氏距离

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

曼哈顿距离

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

L∞距离

$$L_\infty(x_i, x_j) = \max_l |x_i^{(l)} - x_j^{(l)}|$$

度量方式不同，给定点的最近邻点的选择也可能不同。

分类模型 – 决策规则

多数表决

由输入实例的 k 个邻近的训练实例中的多数类决定输入实例的类。

- 分类函数 $f: \mathbf{R}^n \to \{c_1, c_2, \cdots, c_K\}$
- 误分类概率 $P(Y \neq f(X)) = 1 - P(Y = f(X))$

$$\frac{1}{k} \sum_{x_i \in N_k(x)} I(y_i \neq c_j) = 1 - \frac{1}{k} \sum_{x_i \in N_k(x)} I(y_i = c_j)$$

等价于经验风险最小化。

分类模型 – 基本流程

1. 对未知类别的数据集中的每个点：
   计算已知类别数据集众多点与当前点之间的距离；
   按照距离递增次序排序。
2. 选取与当前点距离最小的k个点：
   选定前k个点所在类别的出现频率
   返回前k个点出现频率最高的类别作为当前点的预测分类
3. 重复步骤，完成对所有点的预测分类

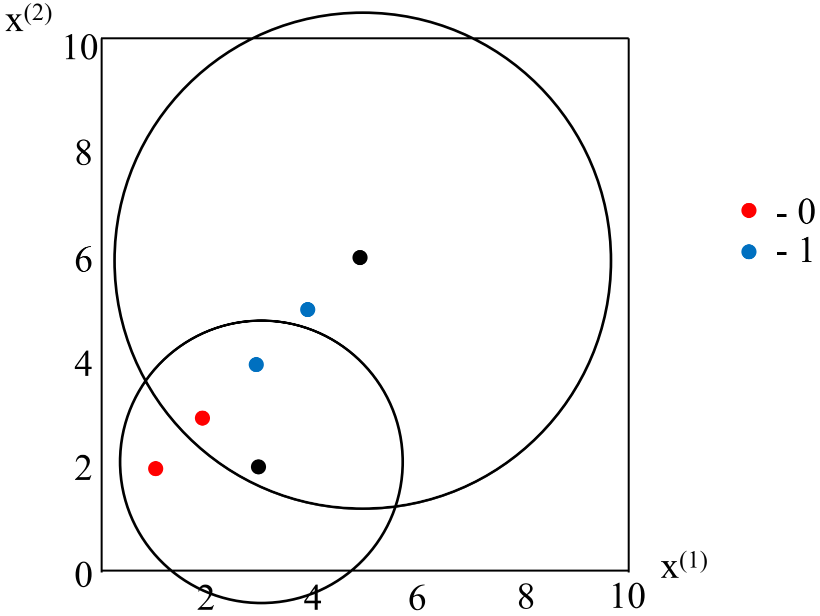
示例 – kNN分类算法的Python实现

```
import numpy as np
from collections import Counter

class KNN:
    def __init__(self, k=3):
        self.k = k
    def fit(self, X, y):
        self.X_train = X
        self.y_train = y
    def euclidean_distance(self, x1, x2):
        return np.sqrt(np.sum((x1 - x2) ** 2))
    def predict(self, X):
        y_pred = [self._predict(x) for x in X]
        return np.array(y_pred)
    def _predict(self, x):
        distances = [self.euclidean_distance(x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]
```

示例 – kNN分类算法的Python实现

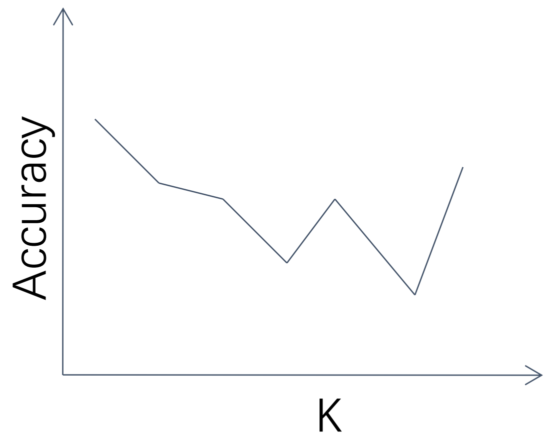
```
# 一个简单的例子:
X_train = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])
y_train = np.array([0, 0, 1, 1])
X_test = np.array([[5, 6], [3, 2]])
clf = KNN(k=3)
clf.fit(X_train, y_train)
predictions = clf.predict(X_test)
print(predictions)
```

[5, 6] -> 1
[3, 2] -> 0



挑战

- 前置处理：特征的选择
- 模型
  - 合适的度量函数
  - 合适的K值
  - 降低训练和预测的复杂度



改进 – kd树

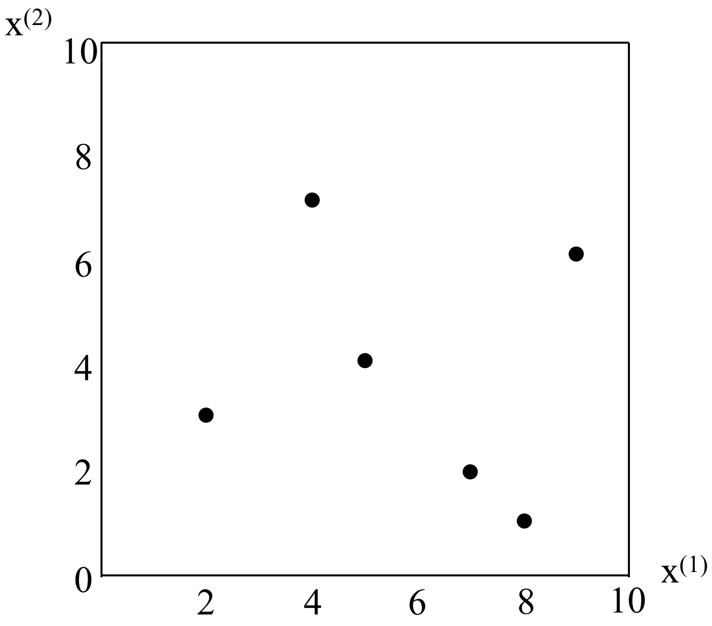
一种二叉树数据结构，用于优化搜索算法。

优势：
- 降低搜索维度
- 提高搜索效率
- 更少的存储需求
- 支持范围搜索

可能因数据的特定分布而表现不佳。

kd树 – 构造

$$T = \{(2,3)^{\mathrm{T}}, (5,4)^{\mathrm{T}}, (9,6)^{\mathrm{T}}, (4,7)^{\mathrm{T}}, (8,1)^{\mathrm{T}}, (7,2)^{\mathrm{T}}\}$$



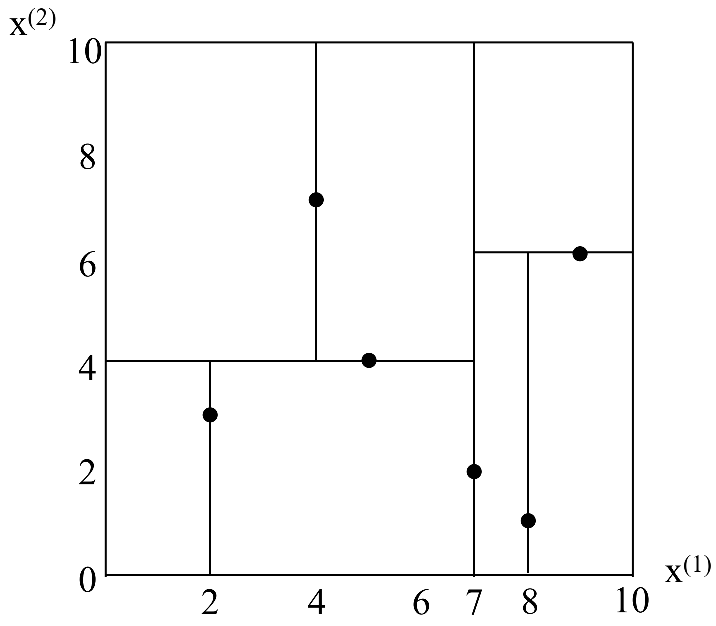
1. 构造根结点，使根结点对应于 k 维空间中包含所有实例点的超矩形区域。
2. 递归（生成子结点）：
    1. 选择坐标轴和切分点，确定一个超平面
    2. 将当前超矩形区域切分为左右两个子区域
    3. 直到子区域内没有实例时终止。
3. 实例保存在相应的结点上。

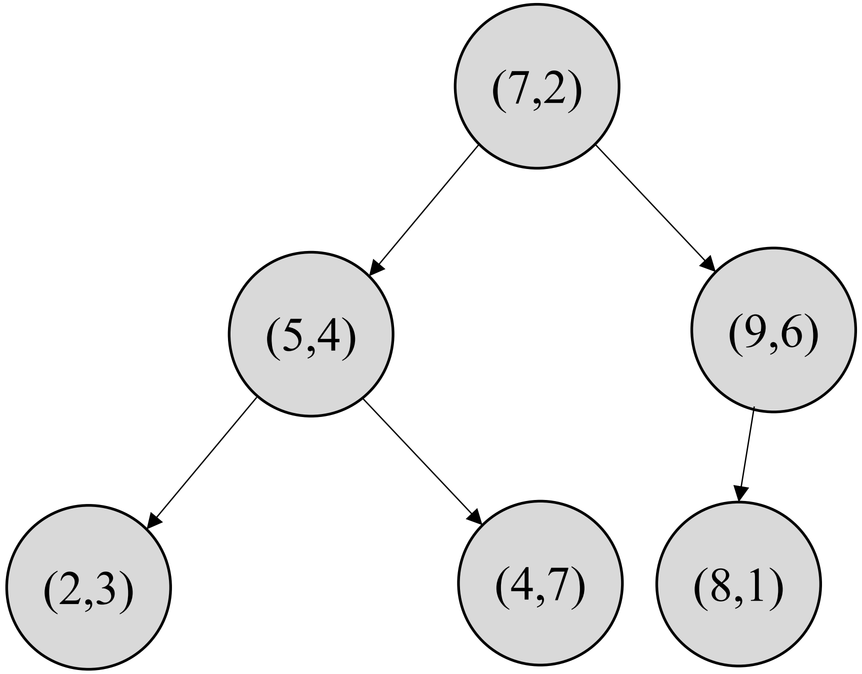
如何选择：

- 空间切分参照：坐标轴
- 切分点的选择：中位数

kd树 – 构造

$$T=\{(2,3)^{\mathrm{T}},(5,4)^{\mathrm{T}},(9,6)^{\mathrm{T}},(4,7)^{\mathrm{T}},(8,1)^{\mathrm{T}},(7,2)^{\mathrm{T}}\}$$

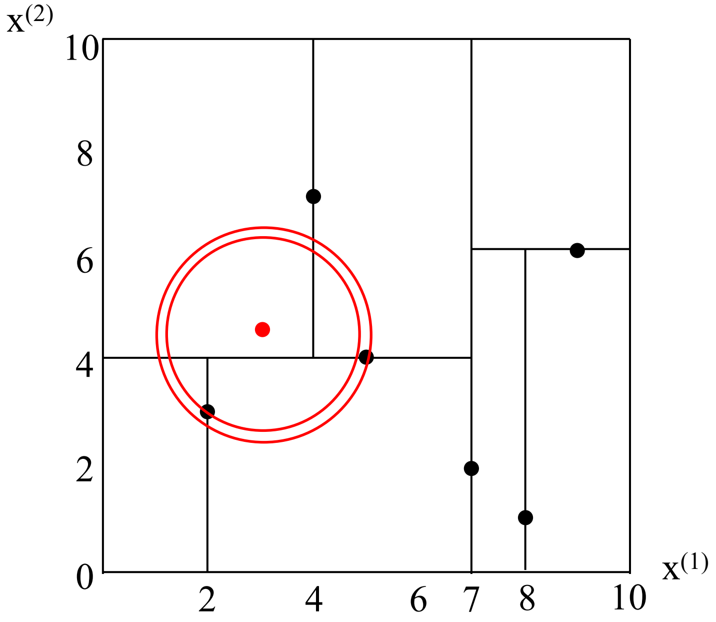


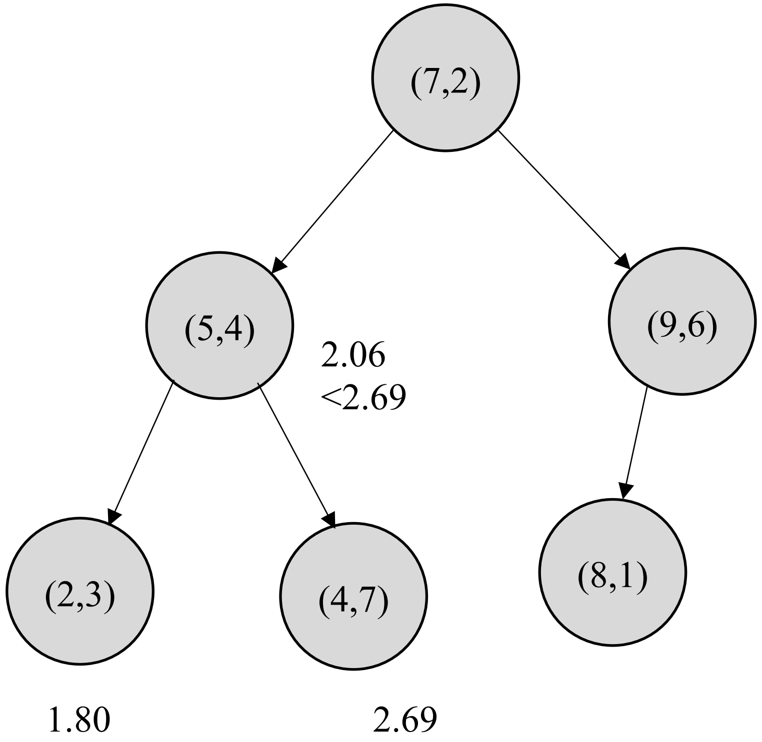


kd树 – 搜索

$$T=\{(2,3)^{\mathrm{T}},(5,4)^{\mathrm{T}},(9,6)^{\mathrm{T}},(4,7)^{\mathrm{T}},(8,1)^{\mathrm{T}},(7,2)^{\mathrm{T}}\}$$

$$x=(3,4.5)^T$$

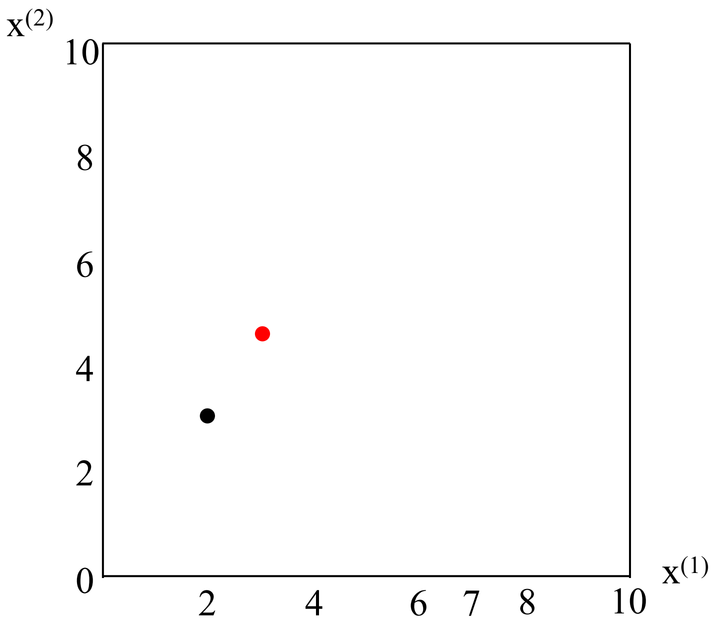


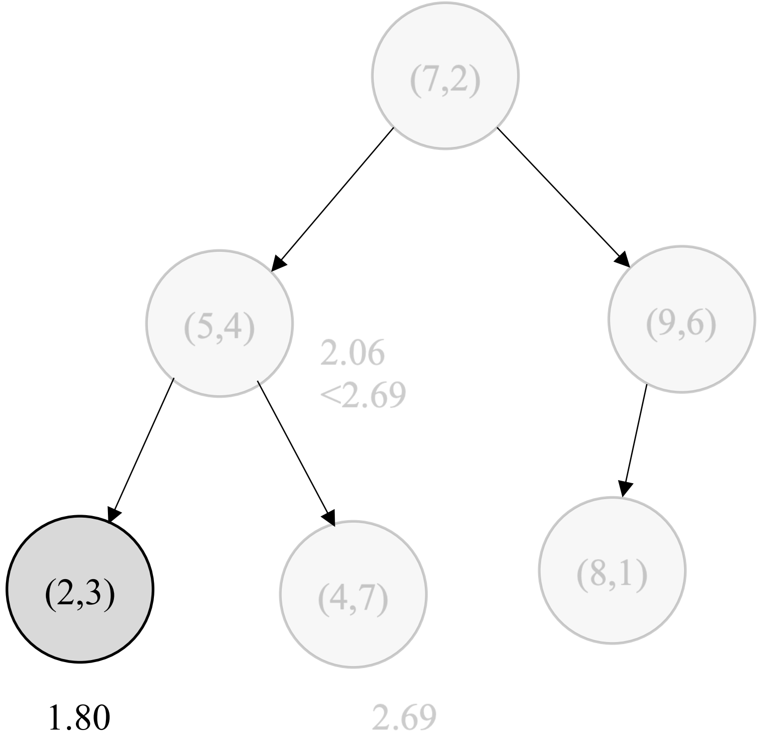


kd树 – 搜索

$$T = \{(2,3)^{\mathrm{T}}, (5,4)^{\mathrm{T}}, (9,6)^{\mathrm{T}}, (4,7)^{\mathrm{T}}, (8,1)^{\mathrm{T}}, (7,2)^{\mathrm{T}}\}$$

$$x = (3, 4.5)^T$$





kd树 – 改进后的算法

[输入] 已构造的 kd 树，目标点 x;
[输出] x 的 k 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. 构建“当前 k 近邻点集”，将该叶结点插入“当前 k 近邻点集”，并计算该结点到目标点 x 的距离。
3. 递归地向上回退，在每个结点进行以下操作:
    (a) 如果“当前 k 近邻点集”的元素数量 $<k$，则将该结点插入“当前 k 近邻点集”，并计算该结点到目标点 x 的距离;
    (b) 如果“当前 k 近邻点集”的元素数量 $=k$，但该结点到目标点 x 的距离小于“当前 k 近邻点集”中最远点到目标点 x 的距离，则将该结点插入“当前 k 近邻点集”，并删除原先的最远点。
    (c) 检查另一子结点对应的区域是否与以目标点 x 为球心、以目标点 x 与“当前 k 近邻点集”中最远点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离目标点更近的点，移动到另一个子结点接着，递归地进行 k 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(若此时“当前 k 近邻点集”中的元素不足 k 个，则需要访问另一半树的结点)。

最后的“当前 k 近邻点集”中的 k 个点即为 x 的 k 近邻点。

kd树 – Python实现

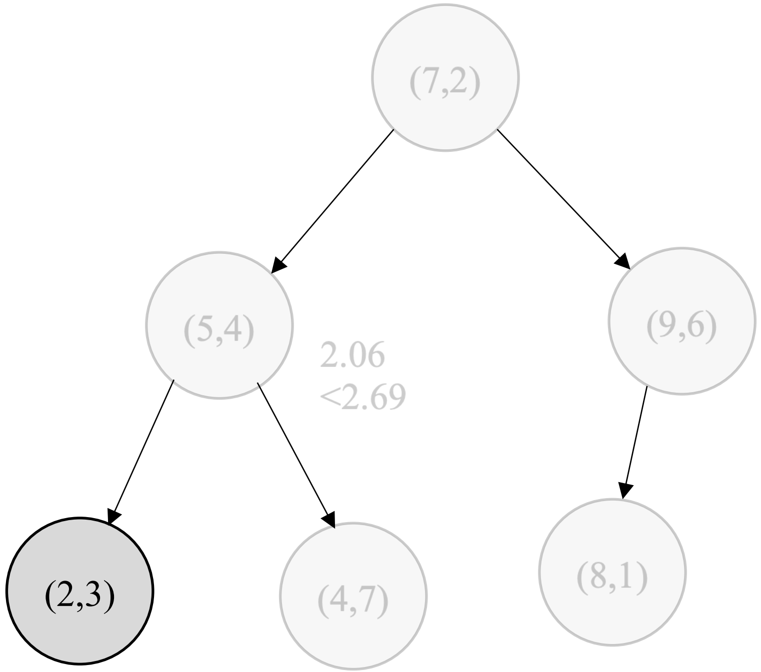
```
class Node:
    def __init__(self, data, left = None, right = None) -> None:
        self.val = data
        self.left = left
        self.right = right

class KdTree:
    def __init__(self, k) -> None:
        self.k = k

def create_Tree(self, dataset, depth):
        if not dataset:
            return None
        mid_index = len(dataset) // 2
        axis = depth % self.k
        sort_dataset = sorted(dataset, key=(lambda x: x[axis]))
        mid_data = sort_dataset[mid_index]
        cur_node = Node(mid_data)
        left_data = sort_dataset[:mid_index]
        right_data = sort_dataset[mid_index+1:]
        cur_node.left = self.create_Tree(left_data, depth + 1)
        cur_node.right = self.create_Tree(right_data, depth + 1)
        return cur_node
```

```
def search(self, tree, new_data):
    self.near_point = None
    self.near_val = None
    def dfs(node, depth):
        if not node:
            return
        axis = depth % self.k
        if new_data[axis] < node.val[axis]:
            dfs(node.left, depth + 1)
        else:
            dfs(node.right, depth + 1)
        dist = self.distance(new_data, node.val)
        if not self.near_val or dist < self.near_val:
            self.near_val = dist
            self.near_point = node.val

        if abs(new_data[axis] - node.val[axis]) <= self.near_val:
            if new_data[axis] < node.val[axis]:
                dfs(node.right, depth + 1)
            else:
                dfs(node.left, depth + 1)
    dfs(tree, 0)
    return self.near_point

def distance(self, point_1, point_2):
    res = 0
    for i in range(self.k):
        res += (point_1[i]-point_2[i]) ** 2
    return res ** 0.5
```

kd树 – Python实现

```
data_set = [[2,3],[5,4],[9,6],[4,7],[8,1],[7,2]]
new_data = [1,5]
k = len(data_set[0])
kd_tree = KdTree(k)
our_tree = kd_tree.create_Tree(data_set, 0)
predict = kd_tree.search(our_tree, new_data)
print(predict)
```

[1, 5] -> [2, 3]



改进 – 马氏距离

由P.C. Mahalanobis提出；
基于样本分布的一种距离测量。

- 考虑特征之间的相关性
- 对数据的缩放不敏感
- 考虑协方差结构
- 适用于异常值和噪声数据

广泛用于分类和聚类分析。

一组向量$\{\vec{X}_1,\vec{X}_2,\vec{X}_3,\dots,\vec{X}_n\}$，其中，

$$\vec{X}=\{x_1,x_2,x_3,\dots,x_m\}$$

其均值为$\vec{\mu}=\{\mu_1,\mu_2,\mu_3,\dots,\mu_m\}$；

协方差矩阵为Σ，其中

$$\Sigma_{ij}=cov(x_i,x_j)$$

单向量的马氏距离定义为：

$$\mathrm{MD}(\vec{X})=\sqrt{(\vec{X}-\vec{\mu})^T\Sigma^{-1}(\vec{X}-\vec{\mu})}$$

向量间的马氏距离定义为：

$$\mathrm{MD}(\vec{X},\vec{Y})=\sqrt{(\vec{X}-\vec{Y})^T\Sigma^{-1}(\vec{X}-\vec{Y})}$$

一组向量：**{3,4},{5,6},{2,2},{8,4}**

均值：**$\vec{\mu}$={4.5,4}**

协方差矩阵：

$$\Sigma=\begin{bmatrix}7 & 2\\ 2 & 2.667\end{bmatrix},\Sigma^{-1}=\begin{bmatrix}0.18 & -0.13\\ -0.13 & 0.48\end{bmatrix}$$

可以计算**{3,4}**和**{5,6}**之间的距离为：

$$\mathrm{MD}=\sqrt{(-2,-2)^T\Sigma^{-1}(-2,-2)}=1.2$$

分类问题中的
k近邻法

杨钧尹
2023-10-31