第7章

k近邻学习

k近邻（k-Nearest Neighbor）学习简称kNN。它既可以作为分类方法，又可以作为回归方法。kNN几乎是最简单直白的机器学习算法，但在处理很多问题时非常有效。

7.1 kNN学习

7.1.1 kNN学习模型

kNN的基本思想简单直观：在处理某些问题时，我们认为两个实例在特征空间中的距离反映了它们之间的相似程度，距离越近则越相似。那么，对于一个输入实例 $\boldsymbol{x}$ 的类别或目标值，可根据训练集中与其距离最近的一些实例（最相似的实例）的类别或目标值进行推断。

假设数据集 $\boldsymbol{D}$ 为训练集，kNN对输入实例 $\boldsymbol{x}$ 进行预测的算法可描述为：

（1）根据某种距离度量方法（通常为欧式距离），找到 $\boldsymbol{D}$ 中与$\boldsymbol{x}$距离最近的 k 个实例。

（2）根据最近的k个实例的类别或目标值，对 $\boldsymbol{x}$ 的类别或目标值进行预测：

- 对于分类问题使用“投票法”，即取 k 个实例中出现最多的类标记作为 $\boldsymbol{x}$ 的预测结果。
- 对于回归问题使用“平均法”，即取 k 个实例的目标值的平均值作为 $\boldsymbol{x}$ 的预测结果。

本章我们主要讨论使用kNN处理分类问题。kNN分类本质上可视为根据训练数据和k值将特征空间划分成一个个小区域，确定每个区域内所有的实例点所属的类别。换句话说，对于给定的训练集和k值，一个输入实例 $\boldsymbol{x}$ 的类标记由它在特征空间的位置唯一确定。举一个例子，在极端情况 k=1 时（1NN也称为最近邻学习），1NN分类器会将输入实例 $\boldsymbol{x}$ 的类标记预测为与其最近的训练实例的类标记。图7-1清晰地展示了1NN分类器是如何根据训练集数据划分特征空间的。

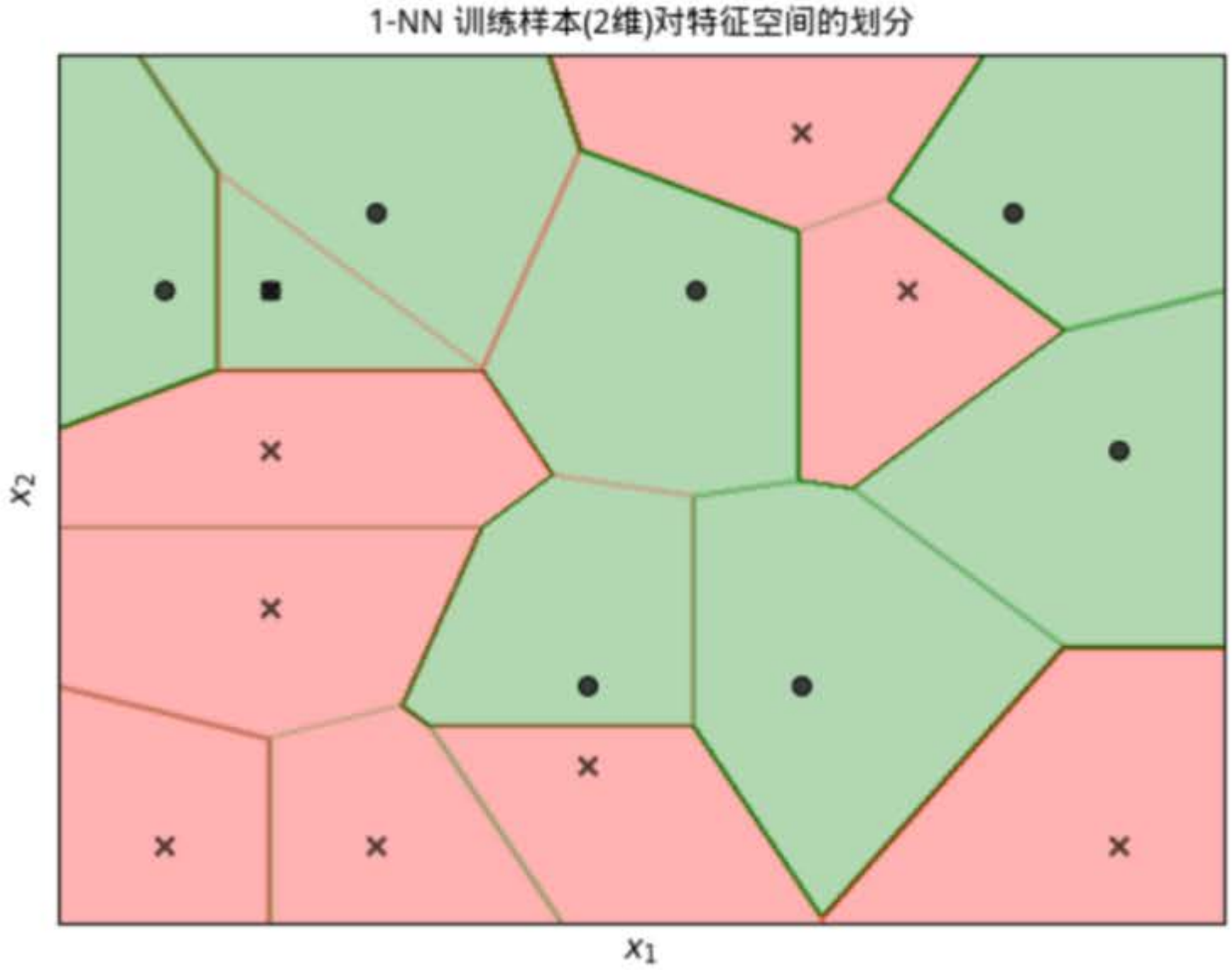

图 7-1

另外，kNN学习并没有显式的训练过程，或者说训练过程仅是把训练数据保存起来，只有在对一个输入实例 $\boldsymbol{x}$ 预测时模型才根据训练数据做出处理，这种方式称为“惰性学习”（Lazy Learning）。

7.1.2 距离的度量

在kNN学习中，假定特征空间中两个点（实例）的距离可以反映它们的相似程度。

特征空间 $\mathbf{R}^n$ 中两点距离的度量有多种方式，我们最熟悉的一种就是中学几何中学习的欧式距离，在KNN学习中通常也使用欧式距离。

$\mathbf{R}^n$ 中两个实例 x_i, x_j，它们的欧式距离定义为：

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

其中 $x^{(l)}$ 表示实例 x 的第 1 个特征。

在某些任务中，可能使用其他距离度量方式效果更佳，比如使用曼哈顿距离。x_i, x_j 的曼哈顿距离定义为：

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

实际上，以上两种距离可看作 $p=2$ 和 $p=1$ 时的闵可夫斯基距离。x_i, x_j的闵可夫斯基距离定义为：

$$d(x_i, x_j) = \left(\sum_{k}^{n} |x_k^{(i)} - x_k^{(j)}|^p \right)^{\frac{1}{p}}$$

在实际应用中计算距离还需注意：在计算距离之前，通常应对各特征数据进行归一化处理，从而消除因各特征尺寸（或值）不同对距离计算造成的影响。举一个简单的例子：a、b、c 三人以cm和kg为单位的身高体重数据分别为（174, 78）、（177, 72）、（184, 80）。

分别计算a与b、c的距离（欧式距离）：

```
1.  >>> import numpy as np
2.  >>> X = np.array([[174., 78.], [177.,  72.], [184.,  80.]])
3.  >>> a, b, c = X
4.  >>> dist_ab = np.sum((a - b) ** 2) ** 0.5
5.  >>> dist_ab
6.  6.708203932499369
7.  >>> dist_ac = np.sum((a - c) ** 2) ** 0.5
8.  >>> dist_ac
9.  10.198039027185569
10. >>> dist_ab > dist_ac
11. False
```

计算结果表明a与b更近。如果身高数据的单位不是 cm 而是 m，再来计算a与b、c的距离：

```
>>> X2 = np.array([[1.74,  78], [1.77,  72.], [1.84,  80.]])
>>> a2, b2, c2 = X2
>>> dist_ab2 = np.sum((a2 - b2) ** 2) ** 0.5
>>> dist_ab2
6.000074999531256
>>> dist_ac2 = np.sum((a2 - c2) ** 2) ** 0.5
>>> dist_ac2
2.0024984394500787
>>> dist_ab2 > dist_ac2
True
```

可以看到，仅改变了计量单位，a变成与c更近了。道理很简单，此时身高的数值远比体重的数值小，在计算距离时身高的影响就很小，这对尺寸较小的特征就“不太公平”。为了消除计量单位的影响，应将各特征的取值范围调整到一个统一的尺寸，归一化处理就是将各特征的数值都映射到 $[0,1]$ 之间。对于某样本 x_i 的第 1 个特征，归一化转换公式如下：

$$x_{i\_Norm}^{(l)} = \frac{x_i^{(l)} - x_{min}^{(l)}}{x_{max}^{(l)} - x_{min}^{(l)}}$$

下面分别对上面例子中的数据进行归一化处理：

```
>>> X_norm = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
>>> X_norm
array([[0.  , 0.75],
       [0.3 , 0.  ],
       [1.  , 1.  ]])
>>> X2_norm = (X2 - X2.min(axis=0)) / (X2.max(axis=0) -
X2.min(axis=0))
>>> X2_norm
array([[0.  , 0.75],
       [0.3 , 0.  ],
       [1.  , 1.  ]])
>>> X_norm == X2_norm
array([[ True,  True],
```

```
13.         [ True,  True],
14.         [ True,  True]])
```

可以看出，某一特征无论如何选取计量单位，其归一化后的结果都是一致的。以归一化的数据再去计算距离，这样就对各特征“公平”了。使用归一化处理就能“公平”地对待每一个特征，但是，这并不意味着各个特征所携带的信息对于模型做分类是同等重要的，至少应该由我们确定哪些特征重要，而不是由其计量单位来决定。

7.1.3 k 值的选择

k值的选择对kNN模型的预测结果有很大的影响。以图7-2为例，当k=3时，预测结果为“三角”；当k=5时，预测结果为“方块”。

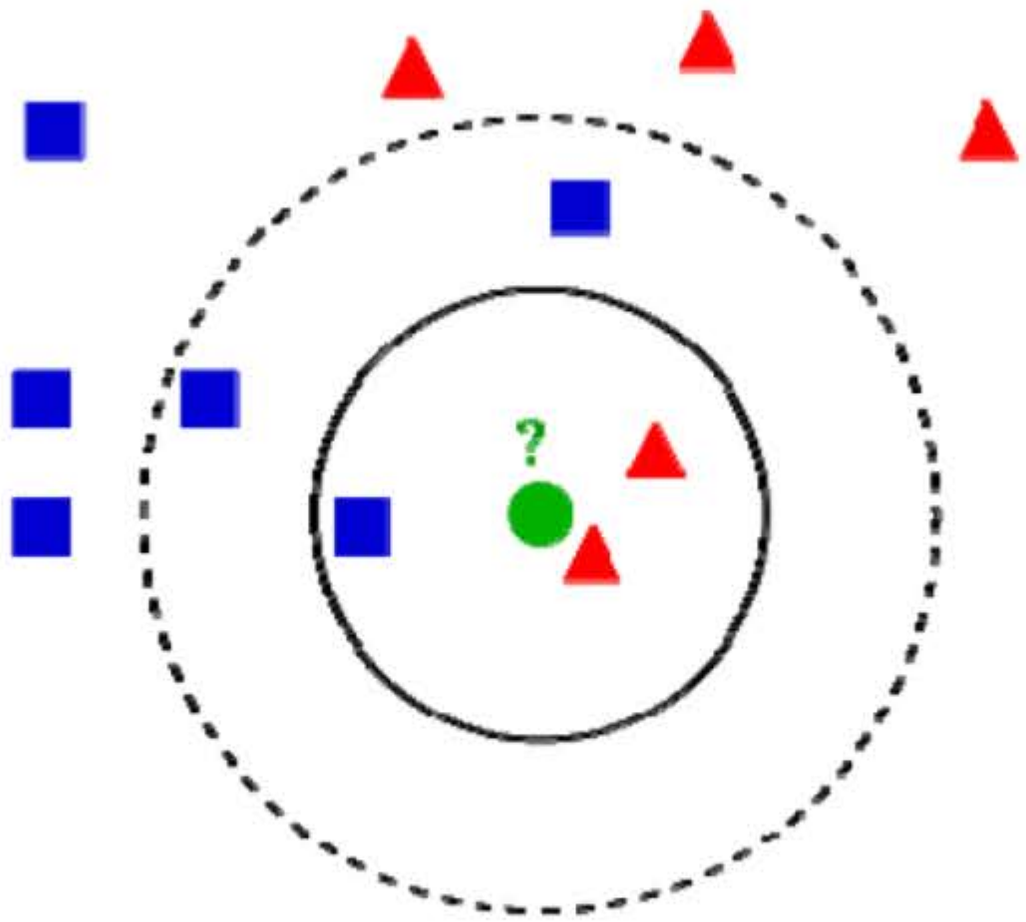

图 7-2

k值较小时，只有测试点周围很少的几个训练实例对预测有贡献，此时近似误差小，而估计误差大。预测结果对近邻的训练实例非常敏感，容易发生过度拟合；k值较大时，测试点周围较大范围内的训练实例都对预测有贡献，此时近似误差大，而估计误差小，虽然不容易发生过度拟合，但预测受到较远距离（不相似）训练实例的影响，导致预测发生错误。k值越小模型越复杂，k值越大模型越简单。在极端情况下，$k = m$ 时（m为训练集 $\boldsymbol{D}$ 的容量），对于任何输入实例的预测结果都为训练集中出现最多的类标记，此时模型过于简单，完全忽略了训练实例中包含的有用信息。

在实际应用中，k值一般选取一个比较小的数值（3,4,5,...）。通常可采用交叉验证法，在几个较小k值中选择最优的。

7.2 kNN的一种实现：k-d树

因为kNN模型非常简单，大家很快便可以想出一种最简单的kNN分类器的实现方法：

（1）对于一个输入实例，计算它和训练集中每一个实例的距离。
（2）根据距离寻找到最近k个邻居。
（3）根据邻居的类别对输入实例的类别进行预测。

以上搜索最近k个邻居的方式称作线性扫描，这种方法虽然实现起来简单，但缺点是需要计算输入实例和每一个训练实例间的距离，如果训练集容量非常大，计算则要耗费大量时间，以至于不可行。本节我们介绍一种常用的实现kNN分类器的方法：k-d树（k-dimensional tree，即k维树）。在寻找最近k个邻居时，k-d树搜索可以大大减少计算距离的次数，从而显著提高搜索效率。

k-d树是存储训练集实例的二叉树，树中每一个节点存储一个训练实例，一个节点对应特征空间中的一个k维超矩形区域（请注意：k-d 树中的“k”实际上指实例的特征数n，而kNN中的“k”指k个邻居）。互为兄弟的两个节点，它们对应的k维超矩形区域是紧挨在一起的，这两个超矩形区域是以父节点中实例的第 l 个特征 $x^{(l)}$ 作为切分点的，用特征空间中垂直于第 l 个轴的超平面，对父节点对应的超矩形区域进行切分得到的，这里的 l 由父节点深度 j 决定，计算式为 $l = j \pmod{n}$。可以想象，k-d树中各层节点依次循环使用各特征进行切分，最终便把整个特征空间切分成了一个个小的超矩形区域。

7.2.1 构造k-d树

设训练集 $D \in \mathbf{R}^n$，k-d树的根节点深度为1。构造k-d树的递归算法如下：

（1）算法输入参数为当前训练集T和当前所创建树（或子树）根节点的深度 j。
（2）根据 j 选择第 l 个特征 $x^{(l)}$ 作为切分特征，其中 $l = j \pmod{n}$。
（3）以 T 中所有实例第 l 个特征的中位数作为切分点：

- 切分点对应的实例 x_{mid} 存入当前所创建树（或子树）的根节点。
- 以第 l 个特征小于中位数的实例构成的集合 T_1 和子树根节点深度 $j+1$ 为参数，递归调用该算法构造左子树。

- 以第 l 个特征大于中位数的实例构成的集合 T_2 和子树根节点深度 $j+1$ 为参数，递归调用该算法构造右子树。

（4）返回当前所创建树（或子树）的根节点。

以上算法描述可能有些抽象，下面我们通过一个例子演示k-d树的构造过程。

假设训练集 $D \in \mathbf{R}^2$，其中有6个实例：

$$(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)$$

首先建立根节点，其深度为1，选择第1个特征进行切分。将实例根据第1个特征排序：

$$(2,3),(4,7),(5,4),(7,2),(8,1),(9,6)$$

使用中位数作为切分点，第1个特征的中位数是7，相应节点为 $(7,2)$，因此：

- 将 $(7,2)$ 存入根节点。
- 将 $(2,3),(4,7),(5,4)$ 划分到左子树。
- 将 $(8,1),(9,6)$ 划分到右子树。

继续构造 $(7,2)$ 的左子树，建立左子树根节点，其深度为2，选择第2个特征进行切分。将实例根据第2个特征排序：

$$(2,3),(5,4),(4,7)$$

第2个特征的中位数是4，相应节点为 $(5,4)$，因此：

- 将$(5,4)$ 存入$(7,2)$ 的左儿子节点。
- 将 $(2,3)$ 划分到左子树。
- 将 $(4,7)$ 划分到右子树。

继续构造 $(5,4)$ 的左右子树，此时左右子树都仅剩一个实例，因此：

- 将$(2,3)$ 存入$(5,4)$ 的左儿子节点。
- 将$(4,7)$ 存入$(5,4)$ 的右儿子节点。

构造 $(7,2)$ 的右子树的过程与左子树相同，不再赘述。

最终构造出的k-d树以及相应特征空间的切分情况如图7-3和图7-4所示。

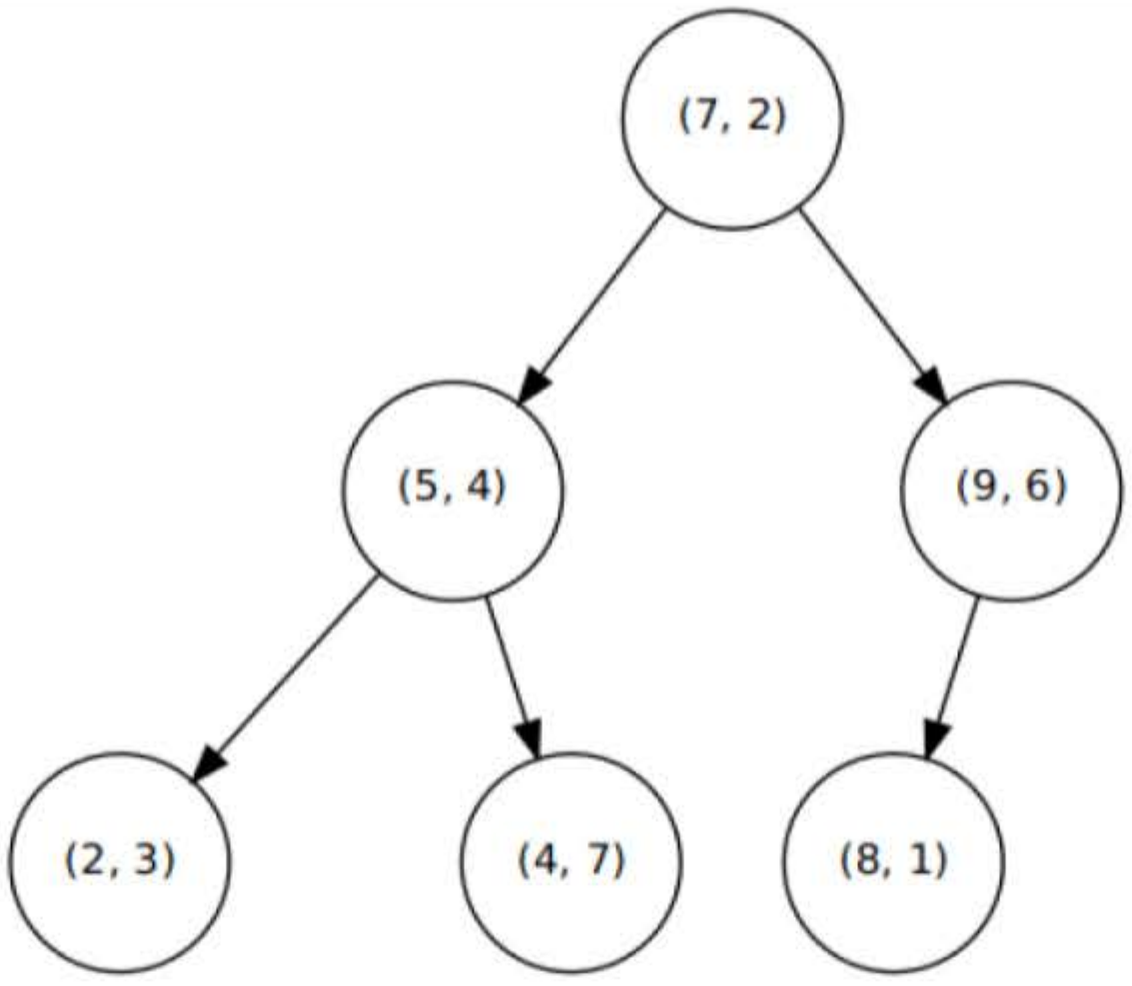

图 7-3

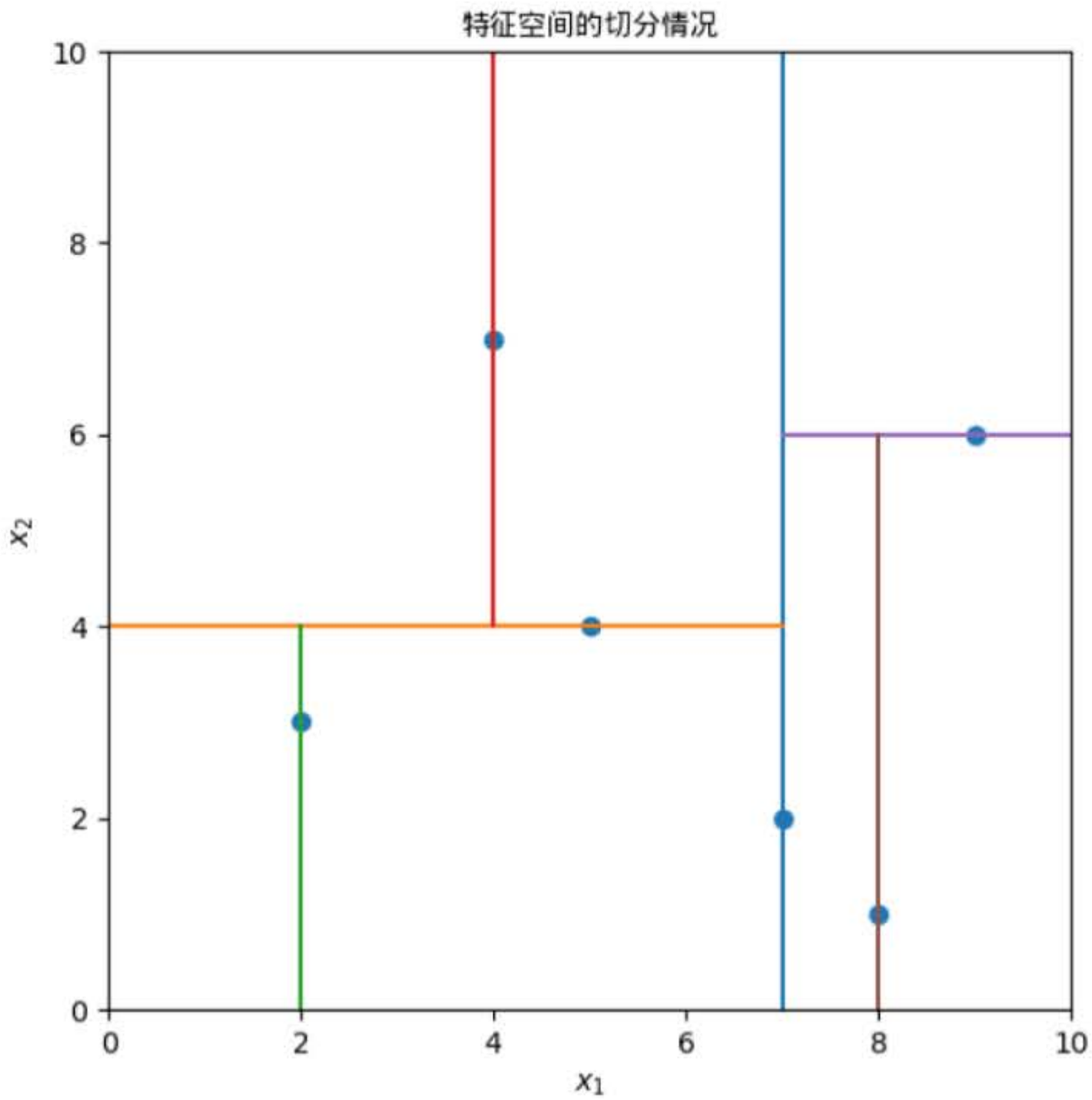

图 7-4

7.2.2 搜索 k-d 树

下面介绍如何在已有k-d树中搜索与给定的输入实例最近的k个邻居。k-d树搜索算法有些复杂，我们先以简单的最近邻（1NN）k-d树搜索为例进行讲解，再推广到k近邻（kNN）k-d树搜索。

之前我们提到，k-d树将特征空间划分成一个个小的超矩形区域，其中叶节点对应最小切分区域。显然给定的输入实例必然位于某个叶节点对应的区域内，k-d搜索算法首先要找到这个叶节点，并设这个叶节点为当前最近邻居，输入实例与叶节点的距离 r 为当前最近距离。可想而知，更近的邻居一定在以输入实例点为球心、r 为半径的超球体内。这个超球体可能不位于叶节点对应的超矩形区域内部，而与其他超矩形区域相交，这种情况下可能有更近的邻居存在于相交的超矩形区域内。因此，接下来搜索算法回退到父节点，在父节点对应的超矩形区域内进行搜索（更近的邻居可能是父节点，也可能位于兄弟节点对应的区域内），搜索完成后更新当前最近邻居以及当前最近距离 r。之后继续回退到当前节点的父节点并搜索相应区域，直到回退到k-d树的根节点，整个特征空间搜索完毕，最近邻居便找到了。下面给出算法的具体细节。

最近邻k-d树搜索的递归算法如下：

（1）算法输入参数为输入实例 x、树（或子树）根节点 $root$、当前最近邻居 x_{nn} 以及当前最近的距离 r_{nn}。

（2）从根节点出发，递归向下访问k-d树：

- 设当前访问节点中实例为 x_{node}，切分特征为第 l 个特征。
- 若 $x^{(l)} \leqslant x^{(l)}_{node}$，则移动到当前访问节点的左儿子节点。
- 若 $x^{(l)} > x^{(l)}_{node}$，则移动到当前访问节点的右儿子节点。

一直到达某个叶节点才停止。

（3）计算输入实例 x 到叶节点中实例的距离 r。若$r < r_{nn}$，则更新x_{nn}和r_{nn}。

（4）递归向根节点回退，每次搜索父节点对应的区域：

- 计算输入实例 x 到父节点中实例的距离r。若$r < r_{nn}$，则更新x_{nn} 和r_{nn}。
- 判断以 x 为球心，r_{nn}为半径的超球体是否与兄弟节点对应的区域相交。若相交，则以兄弟节点为根，递归调用最近邻搜索算法，尝试更新x_{nn} 和r_{nn}。

（5）回退到根节点算法就结束了，返回 x_{nn} 和 r_{nn}。

调用以上最近邻k-d树搜索算法时，可将输入实例 x 传给参数 x，k-d树的根传给参数 $root$，None传给参数 x_{nn}，$+\infty$ 传给参数 r_{nn}。

下面还是通过一个例子来演示最近邻k-d树的搜索过程。回顾7.2.1小节的例子中创建的k-d树，假设现在我们想在该树中找到输入实例 $(6,1)$ 的最近邻居。

对照图7-5想象以下搜索过程。

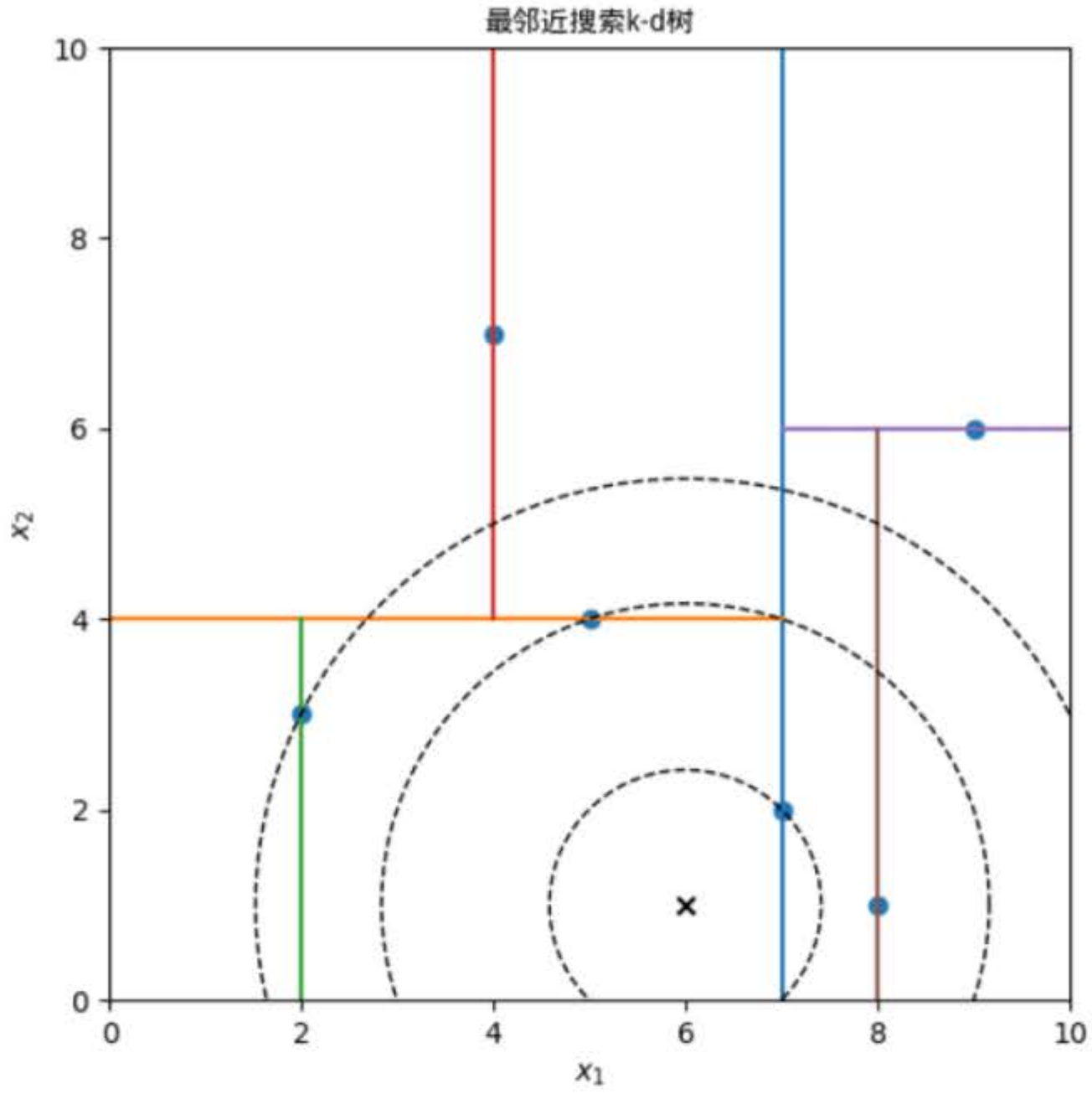

图 7-5

- 从图 7-5 中看出，输入实例$(6,1)$ 位于叶节点 $(2,3)$ 对应的区域内，因此经过搜索算法第（2）步后，当前节点为$(2,3)$，并设当前最近邻居为$(2,3)$，超球体为图 7-5 中最大的圆。
- 接着回退到$(2,3)$ 的父节点 $(5,4)$。从图 7-5 中看出，输入实例到$(5,4)$ 的距离比到$(2,3)$ 更近，因此当前最近邻居更新为$(5,4)$，超球体更新为图 7-5 中次大的圆。
- 次大的圆与$(5,4)$ 另一个子节点 $(4,7)$（$(2,3)$ 的兄弟节点）对应的区域相交，因此递归调用搜索算法搜索以$(4,7)$ 为根的子树（搜索过程省略），但并未找到更近的邻居。
- 继续回退到$(5,4)$ 的父节点 $(7,2)$。从图 7-5 中看出，输入实例到$(7,2)$ 的距离比到$(5,4)$ 更近，因此当前最近邻居更新为$(7,2)$，超球体更新为图 7-5 中最小的圆。

- 最小的圆与$(7,2)$另一个子节点$(9,6)$（$(5,4)$的兄弟节点）对应的区域相交，因此递归调用搜索算法搜索以$(9,6)$为根的子树（搜索过程省略），但并未找到更近的邻居。
- 已回退到根节点，算法结束，最终最近邻居为$(7,2)$。

相信读者已了解了最邻近k-d树搜索，在其基础上推广到k邻近k-d树搜索也很容易，只需使用一个容量为k的最大堆（Max Heap）存储k个最近邻居以及相应的k个距离，其中距离为键（Key）。搜索过程可视为在k-d树中搜索比最大堆中最远邻居更近的邻居，相关细节包括：

- 初始化堆时，可使用 k 个 $+\infty$ 作为距离将堆填满。
- 搜索过程中遇到新训练实例则计算距离，然后以距离为键（Key）执行一次入堆出堆的操作。如果距离足够小，则可从堆中挤出最远邻居，而新训练实例入堆成为 k 个最近邻居之一。整个搜索过程中堆的容量始终保持为 k。
- 最大堆的根节点存储了堆中最远邻居和最大距离。
- 超球体半径始终为堆根节点中存储的最大距离。

7.3 算法实现

7.3.1 线性扫描版本

首先，我们实现一个简单的线性扫描版本的kNN分类器，代码如下：

```
1.  import numpy as np
2.
3.  class KNN:
4.      def __init__(self, k_neighbors=5):
5.          # 保存最近邻居数 k
6.          self.k_neighbors = k_neighbors
7.
8.      def train(self, X_train, y_train):
9.          '''训练'''
10.
11.         # 仅保存训练集合
12.         self.X_train = X_train
13.         self.y_train = y_train
14.
```

```
    def _predict_one(self, x):
        '''对单个实例进行预测'''

        # 计算到每个训练实例的距离
        d = np.linalg.norm(x - self.X_train, axis=1)

        # 获得距离最近 k 个邻居的索引
        idx = np.argpartition(d, self.k_neighbors)
[:self.k_neighbors]

        # 根据索引得到每个测试样本 k_neighbors 个邻居的 y 值
        y_neighbors = self.y_train[idx]

        # 投票法:
        # 1.统计 k 个邻居中各类别出现的个数
        counts = np.bincount(y_neighbors)
        # 2.返回最频繁出现的类别
        return np.argmax(counts)

    def predict(self, X):
        '''预测'''

        # 对 X 中每个实例依次调用_predict_one 方法进行预测
        return np.apply_along_axis(self._predict_one, axis=1, arr=X)
```

上述代码简要说明如下（详细内容参看代码注释）。

- __init__()方法：保存了用户输入的 k 值。
- train()方法：训练模型。之前已经说过 kNN 属于“惰性学习”，该方法仅保存训练数据。
- _predict()方法：对单个实例进行预测。先计算输入实例与每一个训练实例的距离，然后找到最近 k 个邻居，最后统计出邻居中出现次数最多的类别，并作为返回值返回。
- predict()方法：预测。对$\boldsymbol{X}$中每个实例，内部调用_predict_one 方法进行预测。

7.3.2 k-d 树版本

接下来，我们实现一个k-d树版本的kNN分类器，代码如下：

```
1.  import numpy as np
2.  from queue import deque
3.  import heapq
4.
5.  class KDTree:
6.      def __init__(self, k_neighbors=5):
7.          # 保存最近邻居数 k
8.          self.k_neighbors = k_neighbors
9.
10.     def _node_depth(self, i):
11.         '''计算节点深度'''
12.
13.         t = np.log2(i + 2)
14.         return int(t) + (0 if t.is_integer() else 1)
15.
16.     def _kd_tree_build(self, X):
17.         '''构造 k-d 树算法'''
18.
19.         m, n = X.shape
20.         tree_depth = self._node_depth(m - 1)
21.         M = 2 ** tree_depth - 1
22.
23.         # 节点由两个索引构成:
24.         # [0]实例索引，[1]切分特征索引
25.         tree = np.zeros((M, 2), dtype=np.int)
26.         tree[:, 0] = -1
27.
28.         # 使用队列按树的层级和顺序创建 KD-Tree
29.         indices = np.arange(m)
30.         queue = deque([[0, 0, indices]])
31.         while queue:
32.             # 队列中弹出一项包括:
33.             # 树节点索引，切分特征的索引，当前区域所有的实例索引
34.             i, l, indices = queue.popleft()
```

```
35.             # 以实例第 l 个特征中的位数作为切分点进行切分
36.             k = indices.size // 2
37.             indices = indices[np.argpartition(X[indices, l], k)]
38.             # 保存切分点实例到当前节点
39.             tree[i, 0] = indices[k]
40.             tree[i, 1] = l
41.
42.             # 循环使用下一特征作为切分特征
43.             l = (l + 1) % n
44.             # 将切分点左右区域的节点划分到左右子树：将实例索引入队，创建左右子树
45.             li, ri = 2 * i + 1, 2 * i + 2
46.             if indices.size > 1:
47.                 queue.append([li, l, indices[:k]])
48.                 if indices.size > 2:
49.                     queue.append([ri, l, indices[k+1:]])
50.
51.         # 返回树及树的深度
52.         return tree, tree_depth
53.
54.     def _kd_tree_search(self, x, root, X, res_heap):
55.         '''搜索 k-d 树的递归算法，将最近的 k 个邻居存入最大堆'''
56.
57.         i = root
58.         idx = self.tree[i, 0]
59.         # 判断节点是否存在，若不存在，则返回
60.         if idx < 0:
61.             return
62.
63.         # 获取当前 root 节点深度
64.         depth = self._node_depth(i)
65.         # 移动到 x 所在最小超矩形区域相应的叶节点
66.         for _ in range(self.tree_depth - depth):
67.             s = X[idx]
68.             # 获取当前节点切分特征的索引
69.             l = self.tree[i, 1]
70.             # 根据当前节点切分特征的值，选择移动到左儿子或右儿子节点
71.             if x[l] <= s[l]:
72.                 i = i * 2 + 1
```

```
            else:
                i = i * 2 + 2
            idx = self.tree[i, 0]

        if idx > 0:
            # 计算到叶节点中实例的距离
            s = X[idx]
            d = np.linalg.norm(x - s)

            # 执行入堆出堆的操作，更新当前 k 个最近邻居和最近距离
            heapq.heappushpop(res_heap, (-d, idx))

        while i > root:
            # 计算到父节点中实例的距离， 并更新当前最近距离
            parent_i = (i - 1) // 2
            parent_idx = self.tree[parent_i, 0]
            parent_s = X[parent_idx]
            d = np.linalg.norm(x - parent_s)

            # 执行入堆出堆的操作，更新当前 k 个最近邻居和最近距离
            heapq.heappushpop(res_heap, (-d, parent_idx))

            # 获取切分特征的索引
            l = self.tree[parent_i, 1]
            # 获取超球体半径
            r = -res_heap[0][0]
            # 判断超球体(x, r)是否与兄弟节点区域相交
            if np.abs(x[l] - parent_s[l]) < r:
                # 获取兄弟节点的树索引
                sibling_i = (i + 1) if i % 2 else (i - 1)
                # 递归搜索兄弟子树
                self._kd_tree_search(x, sibling_i, X, res_heap)

            # 递归向根节点回退
            i = parent_i

    def train(self, X_train, y_train):
        '''训练'''
```

```
111.
112.          # 保存训练集
113.          self.X_train = X_train
114.          self.y_train = y_train
115.
116.          # 构造 k-d 树，保存树及树的深度
117.          self.tree, self.tree_depth = self._kd_tree_build (X_train)
118.
119.      def _predict_one(self, x):
120.          '''对单个实例进行预测'''
121.
122.          # 创建存储 k 个最近邻居索引的最大堆
123.          # 注意：标准库中的 heapq 实现的是最小堆，以距离的负数作为键则
                等价于最大堆
124.          res_heap = [(-np.inf, -1)] * self.k_neighbors
125.          # 从根开始搜索 kd tree，将最近的 k 个邻居存入堆
126.          self._kd_tree_search(x, 0, self.X_train, res_heap)
127.          # 获取 k 个邻居的索引
128.          indices = [idx for _, idx in res_heap]
129.
130.          # 投票法:
131.          # 1.统计 k 个邻居中各类别出现的个数
132.          counts = np.bincount(self.y_train[indices])
133.          # 2.返回最频繁出现的类别
134.          return np.argmax(counts)
135.
136.      def predict(self, X):
137.          '''预测'''
138.
139.          # 对 X 中每个实例依次调用_predict_one 方法进行预测
140.          return np.apply_along_axis(self._predict_one, axis=1,
arr=X)
```

上述代码简要说明如下（详细内容参看代码注释）。

- __init__()方法：保存了用户输入的 k 值。
- _kd_tree_build()方法：构造 k-d 树算法。使用队列按树的层级和顺序依次构造树节点。

- _kd_tree_search()方法：搜索 k-d 树的递归算法。
- train()方法：训练模型。调用_kd_tree_build 方法构造 k-d 树。
- _predict()方法：对单个实例进行预测。调用_kd_tree_search 方法获取最近 k 个邻居，然后统计出邻居中出现次数最多的类别，并作为返回值返回。
- predict()方法：预测。对$\boldsymbol{X}$中每个实例，内部调用_predict_one 方法进行预测。

7.4 项目实战

最后，我们来做一个kNN的实战项目：使用kNN分类器（k-d树版本）判断乳腺肿瘤是否为良性，如表7-1所示。

表7-1　乳腺肿瘤数据集

（http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29）

列号	列名	特征 / 类标记	可取值
1	ID	-	-
2	Class	类标记	M, B
3	radius (mean)	特征	实数
4	texture (mean)	特征	实数
5	perimeter (mean)	特征	实数
6	area (mean)	特征	实数
7	smoothness (mean)	特征	实数
8	compactness (mean)	特征	实数
9	concavity (mean)	特征	实数
10	concave points (mean)	特征	实数
11	symmetry (mean)	特征	实数
12	fractal dimension (mean)	特征	实数
…	…	…	…
30	concave points (worst)	特征	实数
31	symmetry (worst)	特征	实数
32	fractal dimension (worst)	特征	实数

数据集中有569条数据，其中每一条包含30项关于肿瘤的医学检查数据和1个肿瘤诊断结果（良性/恶性）。

读者可使用任意方式将数据集文件wdbc.data下载到本地。该文件所在的URL为：https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data

7.4.1 准备数据

调用Numpy的genfromtxt函数加载数据集：

```
>>> import numpy as np
>>> X = np.genfromtxt('wdbc.data', delimiter=',', usecols=range(2, 32))
>>> X
array([[1.799e+01, 1.038e+01, 1.228e+02, ..., 2.654e-01, 4.601e-01,
        1.189e-01],
       [2.057e+01, 1.777e+01, 1.329e+02, ..., 1.860e-01, 2.750e-01,
        8.902e-02],
       [1.969e+01, 2.125e+01, 1.300e+02, ..., 2.430e-01, 3.613e-01,
        8.758e-02],
       ...,
       [1.660e+01, 2.808e+01, 1.083e+02, ..., 1.418e-01, 2.218e-01,
        7.820e-02],
       [2.060e+01, 2.933e+01, 1.401e+02, ..., 2.650e-01, 4.087e-01,
        1.240e-01],
       [7.760e+00, 2.454e+01, 4.792e+01, ..., 0.000e+00, 2.871e-01,
        7.039e-02]])
>>> y = np.genfromtxt('wdbc.data', delimiter=',', usecols=1, dtype=np.str)
>>> y
array(['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M',
       'M', 'M', 'M', 'M', 'M', 'M', 'B', 'B', 'B', 'M', 'M', 'M', 'M',
       'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'B', 'M', 'M',
       'M', 'M', 'M', 'M', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M',
       'M', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'M', 'B', 'M',
        ...,
       'B', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'B', 'M', 'M', 'B', 'B',
       'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
       'M', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
```

```
    28.       'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
    29.       'B', 'B', 'B', 'M', 'M', 'M', 'M', 'M', 'M', 'B'], dtype='<U1')
```

目前y中是字符类标记，转换为整数类型（int）的类标记：

```
    1.  >>> y = np.where(y == 'B', 1, 0)
    2.  >>> y
    3.  array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
1, 1,
    4.         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0,
    5.         0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0,
0,
    6.         1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0,
0,
    7.         1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0,
1,
    8.          ...,
    9.         1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
1,
    10.        1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1,
1,
    11.        1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1,
1,
    12.        1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1,
    13.        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1])
```

数据准备完毕。

7.4.2 模型训练与测试

KDTree只有一个超参数，即邻居个数k。先以k=3来创建模型：

```
    1.  >>> from kd_tree import KDTree
    2.  >>> clf = KDTree(3)
```

然后，调用sklearn中的train_test_split函数将数据集切分为训练集和测试集（比例为7:3）：

```
1.  >>> from sklearn.model_selection import train_test_split
2.  >>> X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3)
```

接下来，训练模型：

```
1.  >>> clf.train(X_train, y_train)
```

使用已训练好的模型对测试集中的实例进行预测，并调用sklearn中的accuracy_score函数计算预测的准确率：

```
1.  >>> from sklearn.metrics import accuracy_score
2.  >>>
3.  >>> y_pred = clf.predict(X_test)
4.  >>> accuracy = accuracy_score(y_test, y_pred)
5.  >>> accuracy
6.  0.9298245614035088
```

单次测试一下，预测的准确率为92.98%. 再进行多次（50次）反复测试，观察平均的预测准确率：

```
1.  >>> def test(X, y, k):
2.  ...     X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3)
3.  ...
4.  ...     clf = KDTree(k)
5.  ...     clf.train(X_train, y_train)
6.  ...     y_pred = clf.predict(X_test)
7.  ...     accuracy = accuracy_score(y_test, y_pred)
8.  ...
9.  ...     return accuracy
10. ...
11. >>> accuracy_mean = np.mean([test(X, y, 3) for _ in range(50)])
12. >>> accuracy_mean
13. 0.9278362573099415
```

50次测试的平均预测准确率为92.78%。请注意，在以上测试中，我们并未对X的各特征进行归一化处理，这有可能导致预测准确率偏低。

下面调用sklearn中的MinMaxScaler函数对X的各特征进行归一化处理，再进行50次测试，观察平均的预测准确率：

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> def test(X, y, k):
...     X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3)
...
...     mms = MinMaxScaler()
...     X_train_norm = mms.fit_transform(X_train)
...     X_test_norm  = mms.transform(X_test)
...
...     clf = KDTree(k)
...     clf.train(X_train_norm, y_train)
...     y_pred = clf.predict(X_test_norm)
...     accuracy = accuracy_score(y_test, y_pred)
...
...     return accuracy
...
>>> accuracy_mean = np.mean([test(X, y, 3) for _ in range(50)])
>>> accuracy_mean
0.9659649122807017
```

可以看到，对X的各特征进行归一化处理后，预测的准确率提升到了96.60%，性能还是不错的。

我们再来考察取不同k值（20以内的奇数）时，预测准确率的变化情况：

```
>>> K = list(range(1, 20, 2))
>>> K
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> K = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> acc_arry = [[test(X, y, k) for _ in range(50)] for k in K]
>>> np.mean(acc_arry, axis=1)
array([0.95578947, 0.96444444, 0.96654971, 0.96526316, 0.9677193 ,
       0.96327485, 0.96187135, 0.96502924, 0.9574269 , 0.95988304])
```

根据以上结果绘制曲线，如图7-6所示。

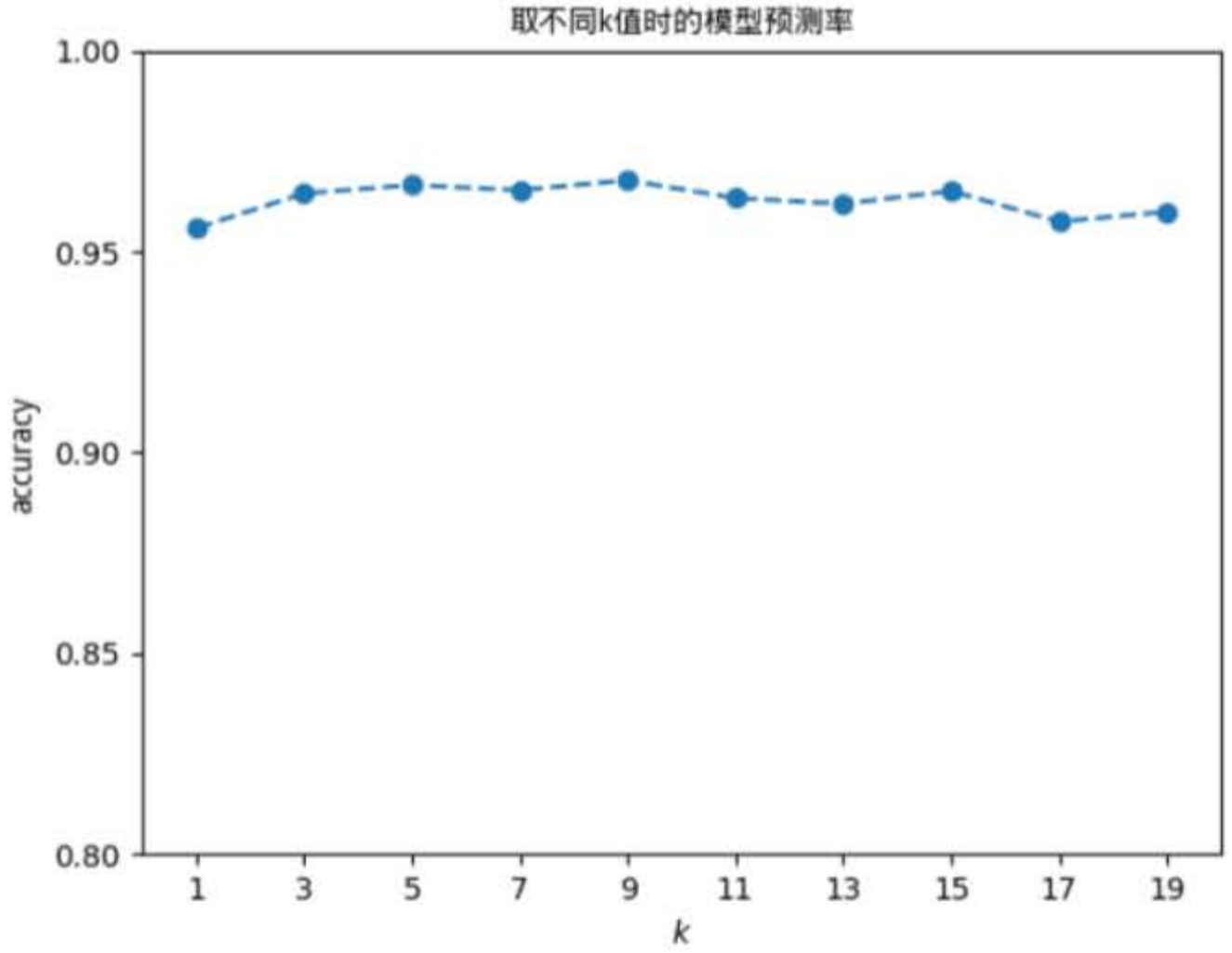

图 7-6

可以发现，对于当前这个分类问题取不同 k 值对模型性能的影响不大，k 取20以内的任意奇数时准确率都超过了95%，但都没能超过97%。

至此，我们使用kNN分类器判断肿瘤是否为良性的项目就完成了。