《机器学习》课程系列

k 近邻*

武汉纺织大学数学与计算机学院

杜小勤

2020/06/19

Contents

---

*本系列文档属于讲义性质，仅用于学习目的。Last updated on: June 20, 2020。

1 概述

k 近邻 (k-Nearest Neighbor, kNN) 是一种基本的监督学习方法，既可以用于分类，也可以用于回归，于 1968 年由 Cover 和 Hart 提出。它也是一种懒惰学习 (Lazy Learning) 方法——几乎没有显式的训练过程——在训练阶段所做的工作仅仅是将训练数据集以某种结构进行保存。

而对于新样本点，则依据某种距离度量从保存的训练数据集中找出与之最"相邻"的 k 个训练样本点，然后基于这 k 个训练样本点共同确定新样本点的类别或回归输出。通常，对于分类问题，可以使用投票法或加权投票法；对于回归问题，可以使用平均法或加权平均法。

对于 kNN 方法，有如下几个重要问题：

- 距离度量的选择；
- k 值的选择；
- 决策规则；
- 存取训练数据集的数据结构；

2 kNN 模型

kNN 模型有三个基本要素，即距离度量、k 值的选择以及决策规则。它们共同决定了 kNN 算法的性能。

kNN 算法简洁明了，非常直观。下面，直接给出 kNN 算法。

算法 **2.1** (kNN 算法)

Input:

Data structure: for example, pre-store $T=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\dots,(\boldsymbol{x}_N,y_N)\}$ in kd tree

k Value: k

New sample: $\boldsymbol{x}$

Output:

Class or regression for $\boldsymbol{x}$: y

Algorithm:

1. find $N_k(\boldsymbol{x})$: k-nearest neighbors in kd tree for $\boldsymbol{x}$
2. for classification:

   $$y=\arg\max_{C_j}\{|C_j|\}=\arg\max_{C_j}\left\{\sum_{\boldsymbol{x}_i\in N_k(\boldsymbol{x})} I(C_j=y_i)\right\},$$

   $j=1,2\dots,K$, K: numbers of class

   for regression:

   $$y=\frac{1}{|N_k(\boldsymbol{x})|}\sum_{\boldsymbol{x}_i\in N_k(\boldsymbol{x})} y_i$$

当 $k=1$ 时，被称为最近邻算法。

2.1 距离度量

距离度量用来反映特征空间中 2 个实例点的距离或相似程度，也被称为相似性度量——在某种度量准则下，2 个实例点距离越小，它们就越相似；反之，距离越大，越不相似。

距离度量的常用准则是 L_p 距离 (L_p Distance) 或 Minkowski 距离 (Minkowski Distance)，它比欧式距离更加具有一般性。

设特征空间 $\mathcal{X}$ 中有 2 个实例点 $\boldsymbol{x}_i$ 和 $\boldsymbol{x}_j$，其 L_p 距离被定义为：

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

其中，n 表示实例点 $\boldsymbol{x}$ 的维度，$p\geqslant 1$。具体地，常用的距离准则有：

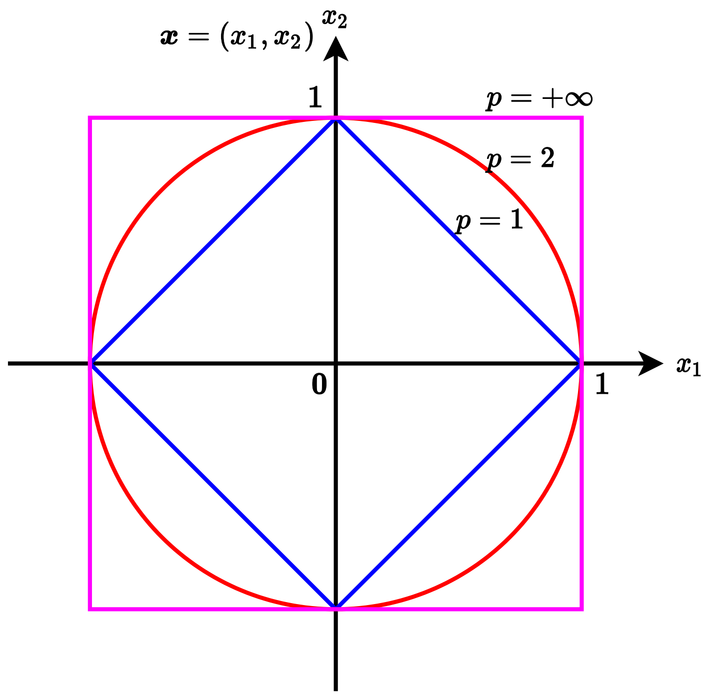


图 2-1: $L_p(\boldsymbol{x},\mathbf{0})=1$ 的点 $\boldsymbol{x}$ 所组成的图形：$p=1$、$p=2$、$p=+\infty$

- $p=1$：

$$L_1(\boldsymbol{x}_i,\boldsymbol{x}_j)=\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}| \tag{2}$$

  被称为曼哈顿距离 (Manhattan Distance)。

- $p=2$：

$$L_2(\boldsymbol{x}_i,\boldsymbol{x}_j)=\left(\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}|^2\right)^{\frac{1}{2}} \tag{3}$$

  被称为欧式距离 (Euclidean Distance)。

- $p=+\infty$：

$$L_{+\infty}(\boldsymbol{x}_i,\boldsymbol{x}_j)=\max_{l}|x_i^{(l)}-x_j^{(l)}| \tag{4}$$

  上式表明，该度量准则将坐标距离中的最大值作为距离。

图2-1展示了上述三种 p 值所对应的图形：二维空间中，$L_p(\boldsymbol{x},\mathbf{0})=1$ 的点 $\boldsymbol{x}$ 所组成的图形。

2.2 k 值

对于 kNN 模型而言，无论是分类问题，还是回归问题，都需要新实例点 $\boldsymbol{x}$ 邻域内 k 个训练样本点参与决策。因此，k 值大小将对 kNN 算法的效果产生直接的影响。

较小的 k 值，即 k 邻域范围较小，新实例点 $\boldsymbol{x}$ 邻域内参与决策的训练样本点数量较少。但是，这些训练样本点与 $\boldsymbol{x}$ 较接近或较相似，它们做出的共同决策往往更准确[1]。从另一方面来讲，k 值越小，决策模型越复杂[2]，越容易发生过拟合。此外，当训练样本点包含噪声时，决策结果对噪声较为敏感，易受到噪声的干扰。

另一方面，较大的 k 值，即 k 邻域范围较大，新实例点 $\boldsymbol{x}$ 邻域内参与决策的训练样本点数量较多。这意味着，离 $\boldsymbol{x}$ 较远或不相似的训练样本点，也参与决策，增加了预测错误的可能性[3]。从另一方面来讲，k 值越大，决策模型越简单，越容易发生欠拟合。一个极端的情形是，$k = N$，其中 N 表示训练集中样本点的个数。显然，该决策模型过于简单，依靠所有的样本点[4]进行投票决策，完全忽略了训练实例样本点中包含的大量有用信息。这种方法，是不可取的。

在实际应用中，从较小的 k 值开始，且以一定的步长，计算出若干个 k 值，最后通过交叉验证来选取最佳的 k 值。

2.3 决策规则

对于分类问题，kNN 模型使用多数表决的策略，确定新实例点 $\boldsymbol{x}$ 的类别输出 y。具体而言，设 $\boldsymbol{x}$ 的邻域 $N_k(\boldsymbol{x})$ 内，C^* 为训练样本点数量最多的类别，即：

$$C^* = \arg\max_{C_j} \{|C_j|\} = \arg\max_{C_j} \left\{ \sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} I(C_j = y_i) \right\} \quad j = 1, 2, \ldots, K \tag{5}$$

其中 K 表示类别数目。

因此，根据“少数服从多数”原则或多数表决规则 (Majority Voting Rule)，可

[1]即近似误差 (Approximation Error) 或偏差 (Bias) 较小。

[2]这意味着 k 值变小时，模型的预测精度变高，模型变得更复杂，即假设空间 $\mathcal{H}$ 变大，而估计误差 (Estimation Error) 或方差 (Variance) 与 $|\mathcal{H}|$ 的大小是成正比的，因而估计误差或方差也将变大。实际上，降低近似误差与估计误差的目标是矛盾的、有冲突的，需要在两者之间取得某种平衡或折中，该问题被称为“偏差与方差的平衡”(Tradeoff between Bias and Variance) 问题。

[3]即近似误差 (偏差) 变大，但是估计误差 (方差) 变小。

[4]无论这些样本点同新实例点 $\boldsymbol{x}$ 是相似的，还是不相似的，统统参与决策。

令 $\boldsymbol{x}$ 的输出为 $y = C^*$。为什么要使用多数表决规则来确定 $\boldsymbol{x}$ 的实际输出 y 呢？下面，分析一下其中的原因。

对于分类问题而言，假设损失函数为 0-1 损失函数：

$$L(y, f(\boldsymbol{x})) = \begin{cases} 1 & y \neq f(\boldsymbol{x}) \\ 0 & y = f(\boldsymbol{x}) \end{cases} \tag{6}$$

其中，$f(\boldsymbol{x})$ 为分类决策函数，此处为 kNN 模型。分类的期望风险函数为：

$$\begin{aligned} R_{exp}(f) &= \mathbb{E}_{P(\boldsymbol{x},y)}\left[L(y, f(\boldsymbol{x}))\right] \quad \Rightarrow \\ R_{exp}(f) &= \mathbb{E}_{P(\boldsymbol{x})} \sum_{y} L(y, f(\boldsymbol{x})) P(y|\boldsymbol{x}) \end{aligned} \tag{7}$$

利用数据样本点之间的独立同分布假设，为了使期望风险最小化，只需对每个样本点 $\boldsymbol{x}$ 逐个最小化，于是得到：

$$\begin{aligned} f^*(\boldsymbol{x}) &= \arg\min_f \sum_y L(y, f(\boldsymbol{x})) P(y|\boldsymbol{x}) \quad \Rightarrow \\ f^*(\boldsymbol{x}) &= \arg\min_f \sum_y I\{y \neq f(\boldsymbol{x})\} P(y|\boldsymbol{x}) \quad \Rightarrow \\ f^*(\boldsymbol{x}) &= \arg\min_f \sum_{y \neq f(\boldsymbol{x})} P(y|\boldsymbol{x}) \quad \Rightarrow \\ f^*(\boldsymbol{x}) &= \arg\min_f 1 - P_{y=f(\boldsymbol{x})}(y|\boldsymbol{x}) \quad \Rightarrow \\ f^*(\boldsymbol{x}) &= \arg\max_f P\left(y = f(\boldsymbol{x})|\boldsymbol{x}\right) \end{aligned} \tag{8}$$

上式的最后一行表明，当 $y = f^*(\boldsymbol{x}) = \arg\max\limits_f P(y|\boldsymbol{x})$ 时，即取后验概率最大的类别作为 $\boldsymbol{x}$ 的输出类别时，期望风险取得最小值。

对于 kNN 模型，如何才能取得后验概率 $P\left(y|\boldsymbol{x}\right)$ 的最大值呢？显然，在给定训练数据集的情况下，对于新实例点 $\boldsymbol{x}$ 的邻域 $N_k(\boldsymbol{x})$ 而言，可以将该邻域内训练样本点数量最多的类别 C^* 作为 y 的输出值。此时，(经验) 误分类率或经验风险将取得最小值[5]：

$$\frac{1}{k} \sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} I(y_i \neq C^*) = 1 - \frac{1}{k} \sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} I(y_i = C^*) \tag{10}$$

[5]注意，经验误分类率、经验风险 (Empirical Risk) 或经验损失 (Empirical Loss) 是模型 $f(\boldsymbol{x})$ 关于训练数据集的平均损失，记作 R_{emp}：

$$R_{emp}(f) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(\boldsymbol{x}_i)) \tag{9}$$

由此可以看出，分类问题中的多数表决规则等价于经验风险最小化。

对于回归问题而言，需要确定新实例点 $\boldsymbol{x}$ 在邻域 $N_k(\boldsymbol{x})$ 内的最佳输出值 C^*。这是数据的拟合问题，一般使用平方误差损失最小化的原则：

$$C^* = \arg\min_{C, \boldsymbol{x}_i \in N_k(\boldsymbol{x})} \frac{1}{2} \sum_{i=1}^{|N_k(\boldsymbol{x})|} (y_i - C)^2 \tag{11}$$

其中 $|N_k(\boldsymbol{x})|$ 表示邻域 $N_k(\boldsymbol{x})$ 内训练样本点的个数。令目标函数关于 C 的导数为 0，得到：

$$-\sum_{i=1}^{|N_k(\boldsymbol{x})|} (y_i - C^*) = 0 \quad \Rightarrow \quad C^* = \frac{1}{|N_k(\boldsymbol{x})|} \sum_{i=1}^{|N_k(\boldsymbol{x})|} y_i \tag{12}$$

上式意义非常明确，新实例点 $\boldsymbol{x}$ 在邻域 $N_k(\boldsymbol{x})$ 内的最佳输出值 C^*，就是该邻域内所有样本实例点输出值 y 的均值。因此，回归问题中的平均值规则也等价于经验风险最小化。

3 kd 树

在 kNN 模型中，对于每个新实例点 $\boldsymbol{x}$，为了求解其类别或得到具体的输出数值，需要能够高效地执行邻域 $N_k(\boldsymbol{x})$ 内训练样本点的搜索工作，尤其当训练样本点的数量较多、维度较大时。因此，如何高效地存取训练样本点，是 kNN 模型实现的关键问题。

下面，介绍 kd 树 (kd Tree) 技术。需要注意的是，此处的 kd 表示样本点 $\boldsymbol{x}$ 的维度为 k 或特征分量的个数为 k；k 的意义与 kNN 中的 k 不一样：后者的 k 表示邻域 $N_k(\boldsymbol{x})$ 中训练样本点的个数。

3.1 kd 树的生成

kd 树是一种二叉树。在初始时，只有一个根节点，且所有的训练样本点都集中于根节点。然后，选择 $\boldsymbol{x}$ 的一个特征分量或坐标轴[6]，并考虑所有样本点在该坐标轴上的值，选择一个最佳切分点[7]，将当前数据集一分为二。从二叉树的角度看，

[6]正如 2 维或 3 维笛卡尔坐标系那样，可以将每个特征分量当作一个坐标轴。“坐标轴”概念，使得描述更加形象具体些。坐标轴可以从第 1 个分量开始，且依据 kd 树的实际生成情况，可以反复使用。

[7]为了使得生成的 kd 树是平衡的，可以考虑将坐标轴上的中位数 (Median) 作为切分点。中位数指的是，按顺序排列的一组数据中居于中间位置的数，它可以将数值集合划分为相等的上下两

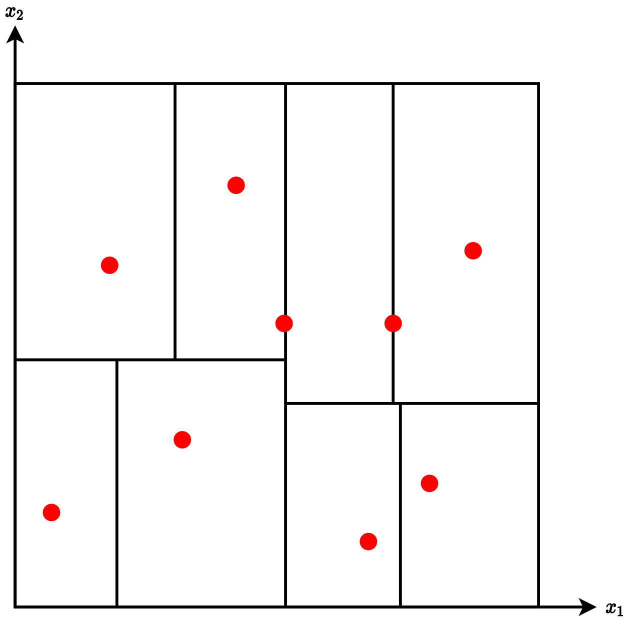


图 3-2: kd 树的特征空间划分

该过程生成了左右两个分支节点：在该坐标轴上，左子树所有样本点的值小于切分点的值，而右子树所有样本点的值大于切分点的值。接着，针对每个分支节点，选取下一个坐标轴，并重复上述过程，直至没有实例点可划分或满足一定条件为止。在上述递归过程中，所有的实例点将被保存在非叶子节点或叶子节点上：保存在非叶子节点上的实例点，其相应的坐标轴取值刚好等于切分点的值。

实际上，kd 树的构造或生成，相当于对训练数据集进行 k 维空间上的一个划分。图3-2展示了 $k=2$ 维时，特征空间的一个划分。

下面，给出平衡 kd 树的生成算法。

算法 3.1 （平衡 kd 树的生成算法）

Input:

Train Dataset: $D=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\ldots,(\boldsymbol{x}_N,y_N)\}$

Output:

kd Tree: $root$

Algorithm:

```
struct KDNode(sample, feaute_index, left, right):
    Sample: sample
```

部分；一般取中间位置的一个数或最中间的 2 个数的平均值。

```
    Feature_Index: feature_index
    Left: left
    Right: right

class KDTree(dataset):
    n_features = dataset.sample.n_features
    root = Build(dataset, 0)

    def Build(dataset, feature_index):
        if dataset is empty:
            return None
        Sort dataset by dataset[feature_index]
        split_pos = len(dataset)/2
        return KDNode(dataset[split_pos], feature_index,
                Build(dataset[:split_pos],
                (feature_index+1)%n_features,
                Build(dataset[split_pos+1:],
                (feature_index+1)%n_features))
```

3.2 kd 树的搜索

给定一个新实例点 $\boldsymbol{x}$，需要在 kd 树上搜索出 k 个邻近的训练样本点[8]。其基本过程是：首先，在 kd 树上，自根节点开始，依据节点的转移条件选择某一分支，从上往下，一直找到包含目标点 $\boldsymbol{x}$ 的叶子节点；然后，从该叶子节点开始，依次向上回溯，在所有遇到的节点中，查找与目标点 $\boldsymbol{x}$ 最近的 k 个训练样本点，并以到 $\boldsymbol{x}$ 的距离按从小到大的顺序对 k 个样本点进行排序，放入队列中；以第 k 个训练样本点与 $\boldsymbol{x}$ 的 (最远) 距离作半径，形成一个超球体；此后，在超球体范围内，查找更近的训练样本点，如果发现新的训练样本点，则将其插入到队列的合适位置，并更新超球体的半径。上述过程反复执行，直至 kd 树搜索完毕。在此过程中，如果发现节点与超球体不相交，则直接忽略掉该节点，不再进一步访问，因而大大提高了搜索效率。

如图3-3所示，展示了 $k=3$ 时，实例点 $\boldsymbol{x}$ 的超球体示意图。可以看出，当前超球体与几个不同层次的节点相交。

下面给出 kd 树上的 k 近邻搜索算法。

算法 **3.2** (kd 树上的 k 近邻搜索算法)

[8]注意，此处的 k，表示实例点 $\boldsymbol{x}$ 的 k 个近邻，即 kNN 中的 k。

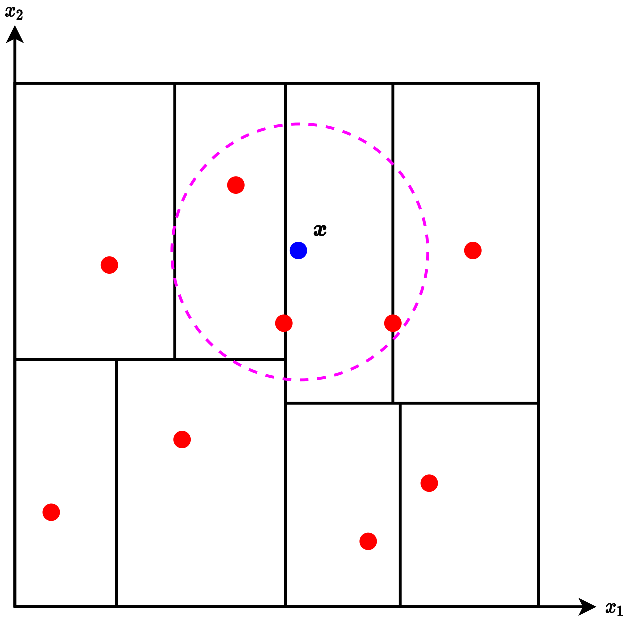


图 3-3: kd 树上的 k 近邻搜索

Input:

kd Tree: *root*

New Sample: $\boldsymbol{x}$ or *source*

k Nearest Neighbors: k-*targets*, empty queue

Output:

k Nearest Neighbors: k-*targets* with k train samples

Algorithm:

```
def FindKNearest(kdnode, source, target, k-targets, radius):
    if kdnode == None:
        return None, +∞

    pivot = kdnode.Sample
    if source[kdnode.feature_index] <= pivot[kdnode.feature_index]:
        forward = kdnode.left
        backward = kdnode.right
    else:
        forward = kdnode.right
        backward = kdnode.left

    forward_result = FindKNearest(forward, source, target, k-targets,
    radius)
    if (k-targets is full and forward_result[1] < radius) or
    (k-targets is not full and forward_result[0] is not None):
        target = forward_result[0]
        insert target into k-targets
        radius = dist(source, k-targets[-1])

    to_pivot = dist(source, pivot)
    if k-targets is full and radius < to_pivot:
        return target, radius
```

```
    target = pivot
    insert target into k-targets
    radius = dist(source, k-targets[-1])

    backward_result = FindKNearest(backward, source, target,
    k-targets, radius)
    if (k-targets is full and backward_result[1] < radius) or
    (k-targets is not full and backward_result[0] is not None):
        target = backward_result[0]
        insert target into k-targets
        radius = dist(source, k-targets[-1])

    return target, radius

def KNearest(source):
    k-targets = an empty queue
    FindKNearest(root, source, None, k-targets, +∞)
    return k-targets
```

4 kNN 实验

相似性度量——L_p 距离（L_p Distance）或 Minkowski 距离（Minkowski Distance）：

- p = 1 曼哈顿距离
- p = 2 欧氏距离
- p = ∞ 闵氏距离

In [1]:
```
import math
```

In [2]:
```
def distance(a, b, p = 2):
    if p == float('inf'):
        maxi = float('-inf')
        for i in range(len(a)):
            maxi = max(maxi, abs(a[i] - b[i]))
        return maxi
    else:
        sum = 0
        for i in range(len(a)):
            sum += math.pow(abs(a[i] - b[i]), p)
        return math.pow(sum, 1/p)
```

In [3]:
```
x1 = [1, 1]
x2 = [5, 1]
x3 = [4, 4]
print(distance(x1, x2, p = 1))
print(distance(x1, x2, p = 2))
print(distance(x1, x2, p = 3))
print(distance(x1, x2, p = 4))
print(distance(x1, x2, p = float('inf')))
print()
print(distance(x1, x3, p = 1))
print(distance(x1, x3, p = 2))
print(distance(x1, x3, p = 3))
print(distance(x1, x3, p = 4))
print(distance(x1, x3, p = float('inf')))
```

```
4.0
4.0
3.9999999999999996
4.0
4

6.0
4.242640687119285
3.7797631496846193
3.5676213450081633
3
```

使用 numpy.linalg.norm 函数计算范数

```
In [4]: import numpy as np
```

```
In [5]: print(np.linalg.norm(np.array(x1) - np.array(x2), ord = 1))
        print(np.linalg.norm(np.array(x1) - np.array(x2), ord = 2))
        print(np.linalg.norm(np.array(x1) - np.array(x2), ord = 3))
        print(np.linalg.norm(np.array(x1) - np.array(x2), ord = 4))
        print(np.linalg.norm(np.array(x1) - np.array(x2), ord = float('inf')))
        print()
        print(np.linalg.norm(np.array(x1) - np.array(x3), ord = 1))
        print(np.linalg.norm(np.array(x1) - np.array(x3), ord = 2))
        print(np.linalg.norm(np.array(x1) - np.array(x3), ord = 3))
        print(np.linalg.norm(np.array(x1) - np.array(x3), ord = 4))
        print(np.linalg.norm(np.array(x1) - np.array(x3), ord = float('inf')))
```

```
4.0
4.0
4.0
4.0
4.0

6.0
4.24264068712
3.77976314968
3.56762134501
3.0
```

载入鸢尾花数据集

```
In [6]: import pandas as pd
        from sklearn.datasets import load_iris
        import matplotlib.pyplot as plt
        %matplotlib inline
        %config InlineBackend.figure_format = 'svg'

        np.random.seed()
```

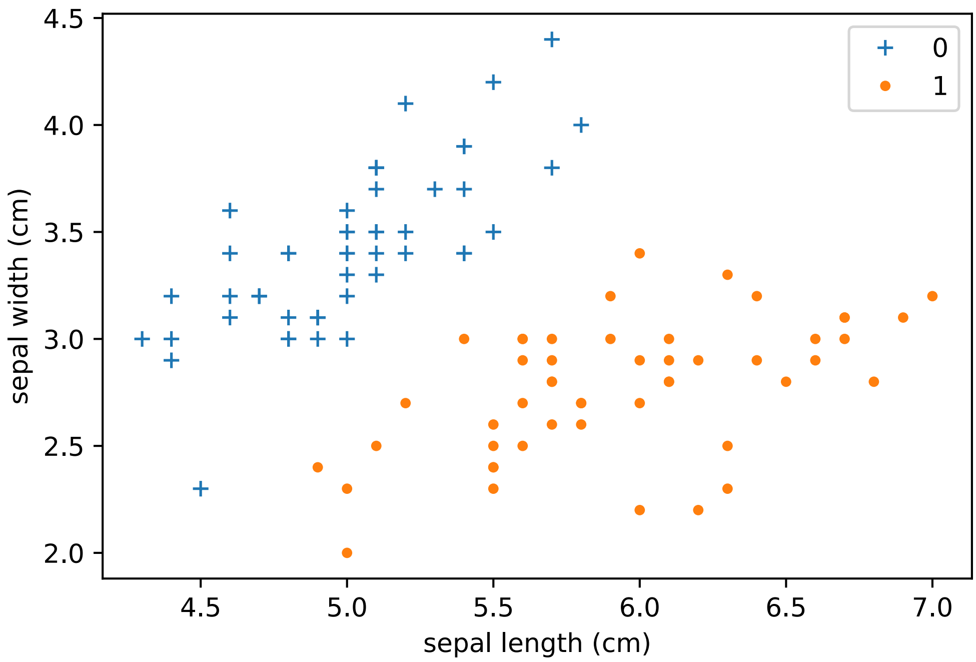
```
In [7]: iris = load_iris()
        iris_df = pd.DataFrame(iris.data, columns = iris.feature_names)
        iris_df['label'] = iris.target

        # 取出第 0 列、第 1 列、最后一列，前 100 条记录
        data = np.array(iris_df.iloc[:100, [0, 1, -1]])
        X, Y = data[:, :-1], data[:, -1]
        print(data.shape)
```

```
(100, 3)
```

绘制训练数据集

```
In [8]: plt.plot(data[:50, 0], data[:50, 1], '+', label='0')
        plt.plot(data[50:100, 0], data[50:100, 1], '.', label='1')
        plt.xlabel('sepal length (cm)')
        plt.ylabel('sepal width (cm)')
        plt.legend()
        plt.savefig('KNN_OUTPUT1.pdf', bbox_inches='tight')
```



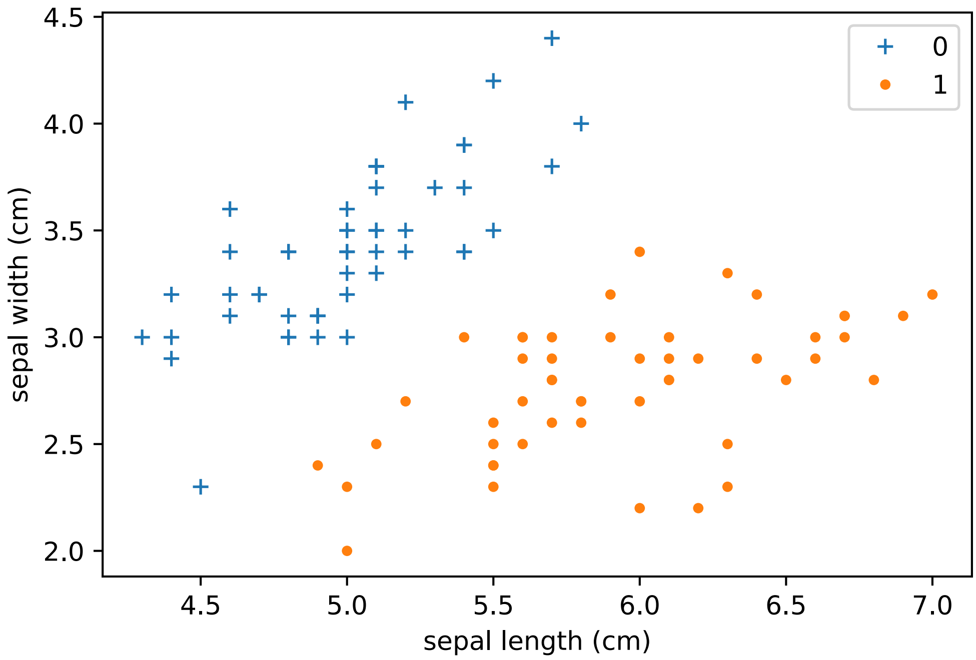
保存数据集，以后可以直接使用

```
In [9]: np.savez_compressed('iris.npz', data = data, X = X, Y = Y)
```

测试刚才保存的训练数据集

```
In [10]: iris_npz = np.load('iris.npz')
         data = iris_npz['data']
         X = iris_npz['X']
         Y = iris_npz['Y']
```

```
In [11]: plt.plot(data[:50, 0], data[:50, 1], '+', label='0')
         plt.plot(data[50:100, 0], data[50:100, 1], '.', label='1')
         plt.xlabel('sepal length (cm)')
         plt.ylabel('sepal width (cm)')
         plt.legend()
         plt.savefig('KNN_OUTPUT2.pdf', bbox_inches='tight')
```



kd 树的生成

```
In [12]: class KDNode:
             def __init__(self, sample, feature_index, left, right):
                 self.sample = sample
                 self.feature_index = feature_index
                 self.left = left
                 self.right = right
```

```
In [13]: class KDTree:
             def __init__(self, dataset):
                 self.n_features = dataset.shape[1] - 1 # 最后一列是标签
                 self.root = self.Build(dataset, 0)

             def Build(self, dataset, feature_index):
                 if len(dataset) == 0:
                     return None
                 dataset = sorted(dataset, key = lambda x: x[feature_index])
                 split_pos = len(dataset) // 2
                 return KDNode(dataset[split_pos], feature_index,\
                         self.Build(dataset[:split_pos],
                         (feature_index + 1) % self.n_features),\
                         self.Build(dataset[split_pos+1:],
                         (feature_index + 1) % self.n_features))

             # 参数：KDTree 树节点 KDNode、源数据点、KDTree 中的最近数据点、最近距离
```

```
    # 返回：KDTree 中的最近数据点、最近距离
    def _FindNearest(self, kdnode, source, target, mindist):
        if kdnode == None: # 无节点可以访问
            return None, float('inf') # 分别表示数据点、距离

        # 当前节点
        pivot = kdnode.sample
        # 左子树
        if source[kdnode.feature_index] <= pivot[kdnode.feature_index]:
            forward = kdnode.left
            backward = kdnode.right
        else: # 右子树
            forward = kdnode.right
            backward = kdnode.left

        # 往子树递归访问
        forward_result = self._FindNearest(forward, source, target, mindist)
        if forward_result[1] < mindist: # 发现目前最近距离
            target = forward_result[0]
            mindist = forward_result[1]

        # -1，去除标签数据
        to_pivot = np.linalg.norm(source[:-1] - pivot[:-1], ord = 2)
        # pivot 离 source 更远，不可能找到更近的点，不访问另一分支 backward,
        # 直接返回
        if mindist < to_pivot:
            return target, mindist

        # 发现更近的数据点，更新数据
        target = pivot
        mindist = to_pivot

        # 继续在另一分支 backward 上搜索，有可能找到更近的点
        backward_result = self._FindNearest(backward, source, target, mindist)
        if backward_result[1] < mindist: # 发现目前最近距离
            target = backward_result[0]
            mindist = backward_result[1]

        return target, mindist

    def Nearest(self, source):
        result = self._FindNearest(self.root, source, None, float('inf'))
        return result[0] # 返回 KDTree 中的最近数据点
```

测试《统计学习方法》第 2 版，李航，P55 实例
生成 KDTree

```
In [14]: # 最后一列是标签
```

```
example_data = np.array([[2, 3, 0],[5, 4, 0],[9, 6, 0],[4, 7, 1],
 [8, 1, 1],[7, 2, 1]])
example_kdtree = KDTree(example_data)
```

按先根遍历打印 KDTree

```
In [15]: def print_kdtree(root):
             if root == None:
                 return
             print(root.sample)
             print_kdtree(root.left)
             print_kdtree(root.right)
         print_kdtree(example_kdtree.root)
```

```
[7 2 1]
[5 4 0]
[2 3 0]
[4 7 1]
[9 6 0]
[8 1 1]
```

在 KDTree 树中搜索

```
In [16]: source = np.array([2, 16, 0])
         print('Nearst:', example_kdtree.Nearest(source))
```

```
Nearst: [4 7 1]
```

按穷举法搜索

```
In [17]: print('Nearst:', example_data[np.argmin([np.linalg.norm(item[:-1]
         -source[:-1], ord = 2) for item in example_data])])
```

```
Nearst: [4 7 1]
```

测试随机生成的数据

```
In [18]: from time import clock
         from random import random

         K = 6 # 维度 5, 最后 1 维，假定为标签
         N = 400000
         random_data = np.array([[random() for k in range(K)] for n in range(N)])
         t0 = clock()
         random_kdtree = KDTree(random_data)
         t1 = clock()
         print ("KDTree building time: ", t1-t0, "s")
```

```
        source = [random() for k in range(K)]
        t0 = clock()
        print('Nearst:', random_kdtree.Nearest(source)) # 在 KDTree 树中搜索
        t1 = clock()
        print ("KDTree searching time: ", t1-t0, "s")
        t0 = clock()
        print('Nearst:', random_data[np.argmin([np.linalg.norm(item[:-1]
         -source[:-1], ord = 2) for item in random_data])]) # 按穷举法搜索
        t1 = clock()
        print ("One-by-one searching time: ", t1-t0, "s")
```

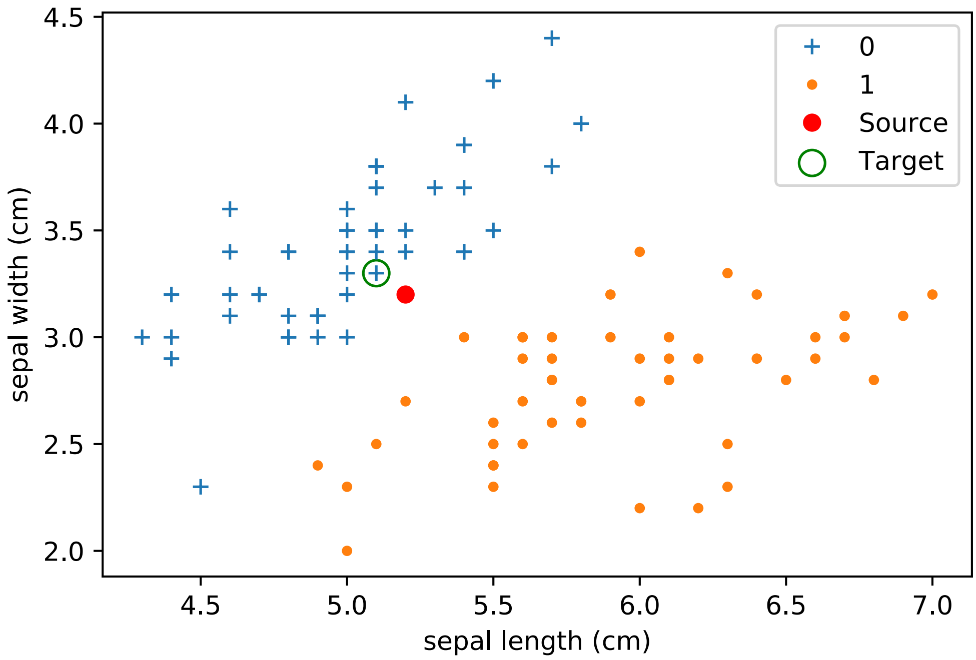
```
KDTree building time:  6.3251292875099425 s
Nearst: [ 0.90124998  0.71193777  0.87953213  0.89937157  0.18528211  0.35873041]
KDTree searching time:  0.0014760656600749655 s
Nearst: [ 0.90124998  0.71193777  0.87953213  0.89937157  0.18528211  0.35873041]
One-by-one searching time:  2.867321243509271 s
```

测试鸢尾花数据集

```
In [19]: iris_kdtree = KDTree(data)
         source = np.array([5, 4, 0])
         print('Nearst:', iris_kdtree.Nearest(source)) # 在 KDTree 树中搜索
         print('Nearst:', data[np.argmin([np.linalg.norm(item[:-1]
          -source[:-1], ord = 2) for item in data])]) # 按穷举法搜索
```

```
Nearst: [ 5.1  3.8  0. ]
Nearst: [ 5.1  3.8  0. ]
```

```
In [20]: source = np.array([5.2, 3.2, 0])
         target = iris_kdtree.Nearest(source)
         plt.plot(data[:50, 0], data[:50, 1], '+', label='0')
         plt.plot(data[50:100, 0], data[50:100, 1], '.', label='1')
         plt.plot(source[0], source[1], 'ro', label='Source')
         plt.scatter(target[0], target[1], color = '', marker = 'o',
          edgecolors = 'g', s = 100, label='Target')
         plt.xlabel('sepal length (cm)')
         plt.ylabel('sepal width (cm)')
         plt.legend()
         plt.savefig('KNN_OUTPUT3.pdf', bbox_inches='tight')
```



sklearn 的 kNN 分类器
sklearn.neighbors.KNeighborsClassifier:

- n_neighbors: 临近点个数
- p: 距离度量
- algorithm: 实现算法，可选 {'auto', 'ball_tree', 'kd_tree', 'brute'}
- weights: 确定近邻的权重

```
In [21]: from sklearn.neighbors import KNeighborsClassifier
         from sklearn.model_selection import train_test_split
```

鸢尾花数据集的 80% 用作训练数据，20% 用作测试数据

```
In [22]: X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2)

In [23]: clf = KNeighborsClassifier()
         clf.fit(X_train, Y_train)

Out[23]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                    weights='uniform')

In [24]: clf.score(X_test, Y_test)

Out[24]: 1.0
```

5 参考文献

1. 李航。《统计学习方法》，清华大学出版社，2019 年 5 月第 2 版。
2. 周志华。《机器学习》，清华大学出版社，2016 年 1 月第 1 版。
3. Christopher M. Bishop. Pattern Recognition and Machine Learning. Springer, 2006.
4. Kevin P. Murphy. Machine Learning: A Probabilistic Perspective, The MIT Press, 2012.
5. https://davidrosenberg.github.io/ml2015/docs/5b.bias-variance.pdf.
6. https://www.zhihu.com/question/60793482.
7. https://www.colorado.edu/amath/sites/default/files/attached-files/k-d_trees_and_knn_searches.pdf.
8. https://github.com/wzyonggege/statistical-learning-method.