02-K最邻近算法

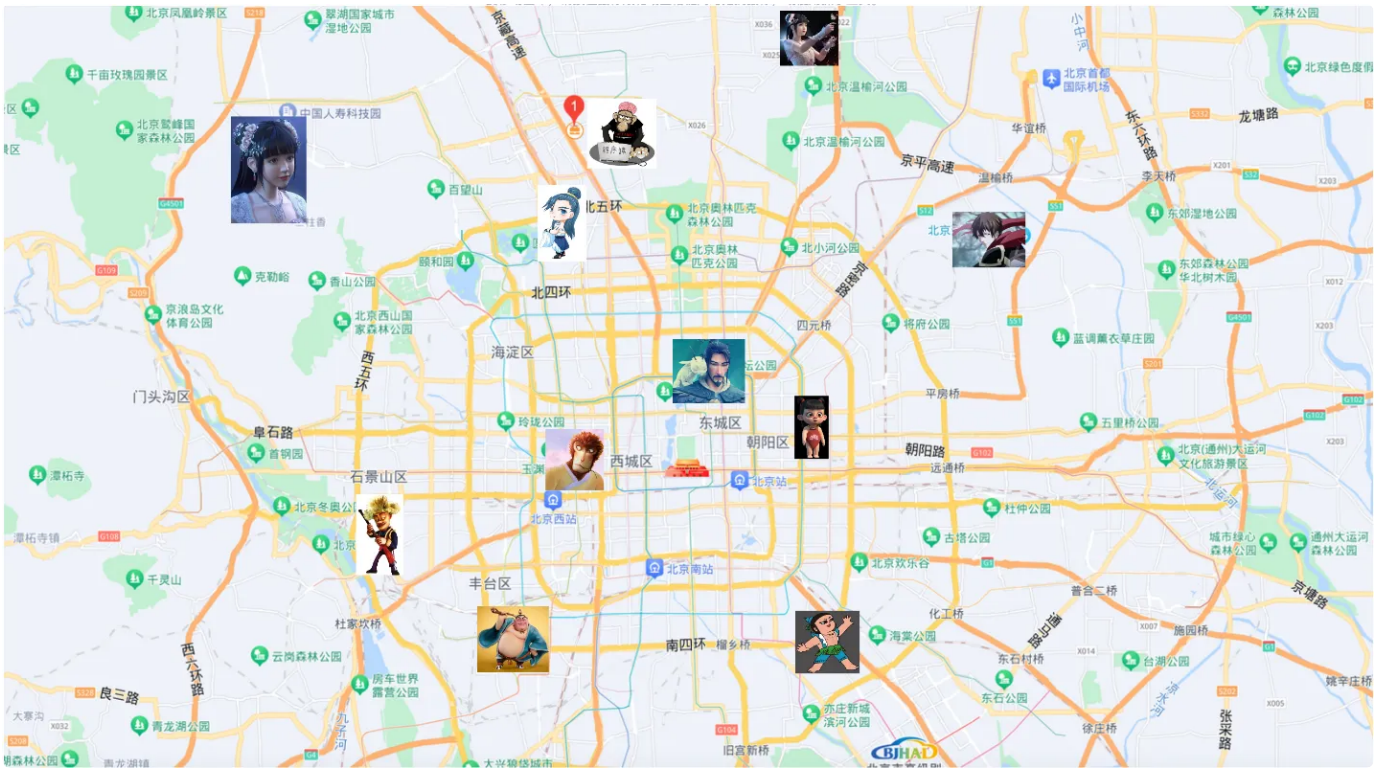
1、什么是K-近邻算法
  1.1、KNN算法介绍
  1.2、KNN核心思想
  1.3、k最近邻算法的步骤描述如下：
2、KNN 算法的关键
  2.1、K的取值
  2.2、距离的测算
3、KNN算法的一般流程
3、KNN算法的伪代码
4、使用数学公式实现 KNN

1、什么是K–近邻算法

1.1、KNN算法介绍

- KNN 算法，或者称 k最邻近算法，是 **有监督学习** 中的 **分类算法** 。它可以用于分类或回归问题，但它通常用作分类算法。
- K近邻法（K–Nearest Neighbor，KNN）是一种简单易懂的多分类方法，它也可以被用于回归运算中。
- 根据**距离**其“最近的邻居”来推断出其类别。

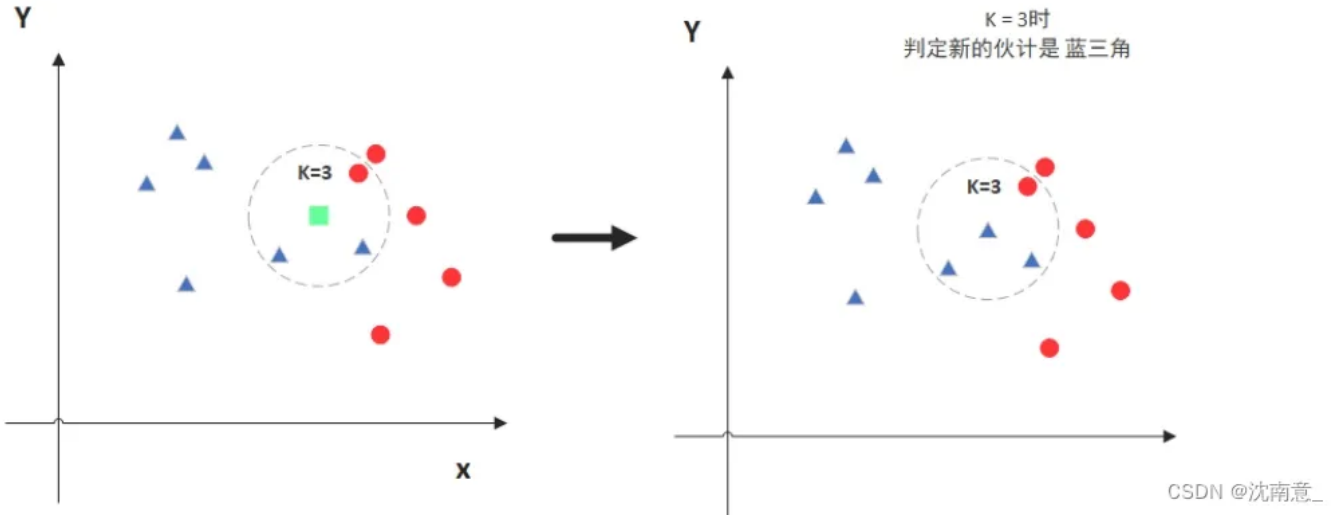
可以通过地图，判断自己在北京市的哪个区，讲解K–进行算法?



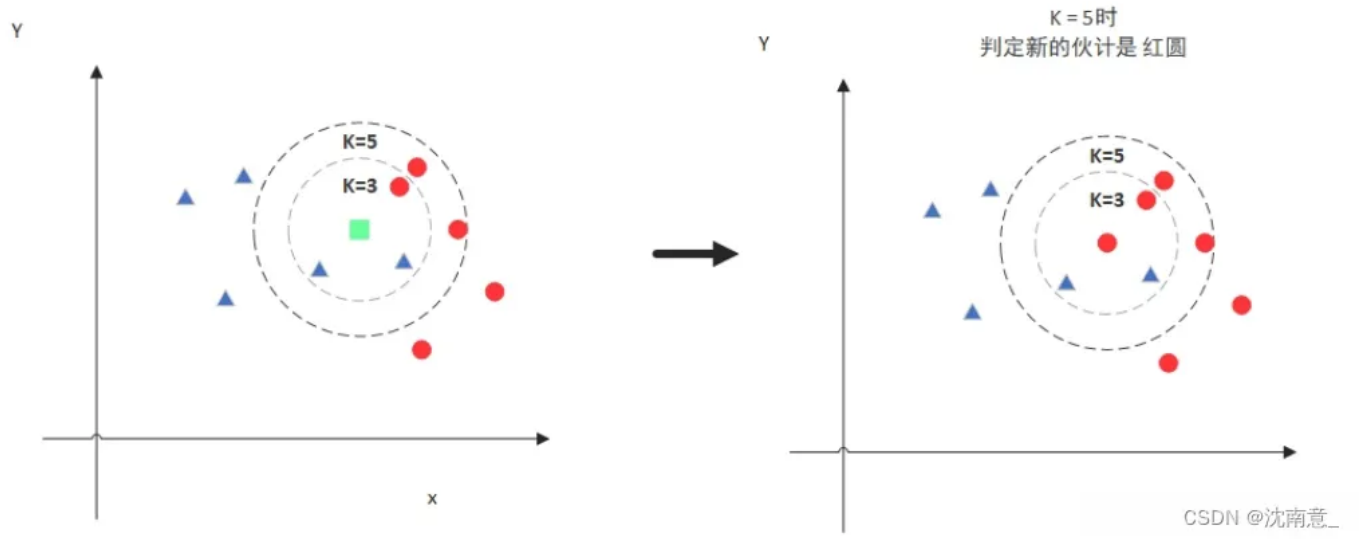
1.2、KNN核心思想

- KNN 的全称是 K Nearest（**最近的**） Neighbors（**邻居**），意思是 K 个最近的邻居。该算法用 K 个最近邻来干什么呢?
- KNN 的原理就是：**当预测一个新样本的类别时，如果一个样本在特征空间中的K个最相似（即特征空间中最临近）的样本中的大多数属于某个类别，则该样本属于某个类别。（多数投票）**

可以简单理解为：由那些离X最近的k个点来投票决定X归为哪一类。



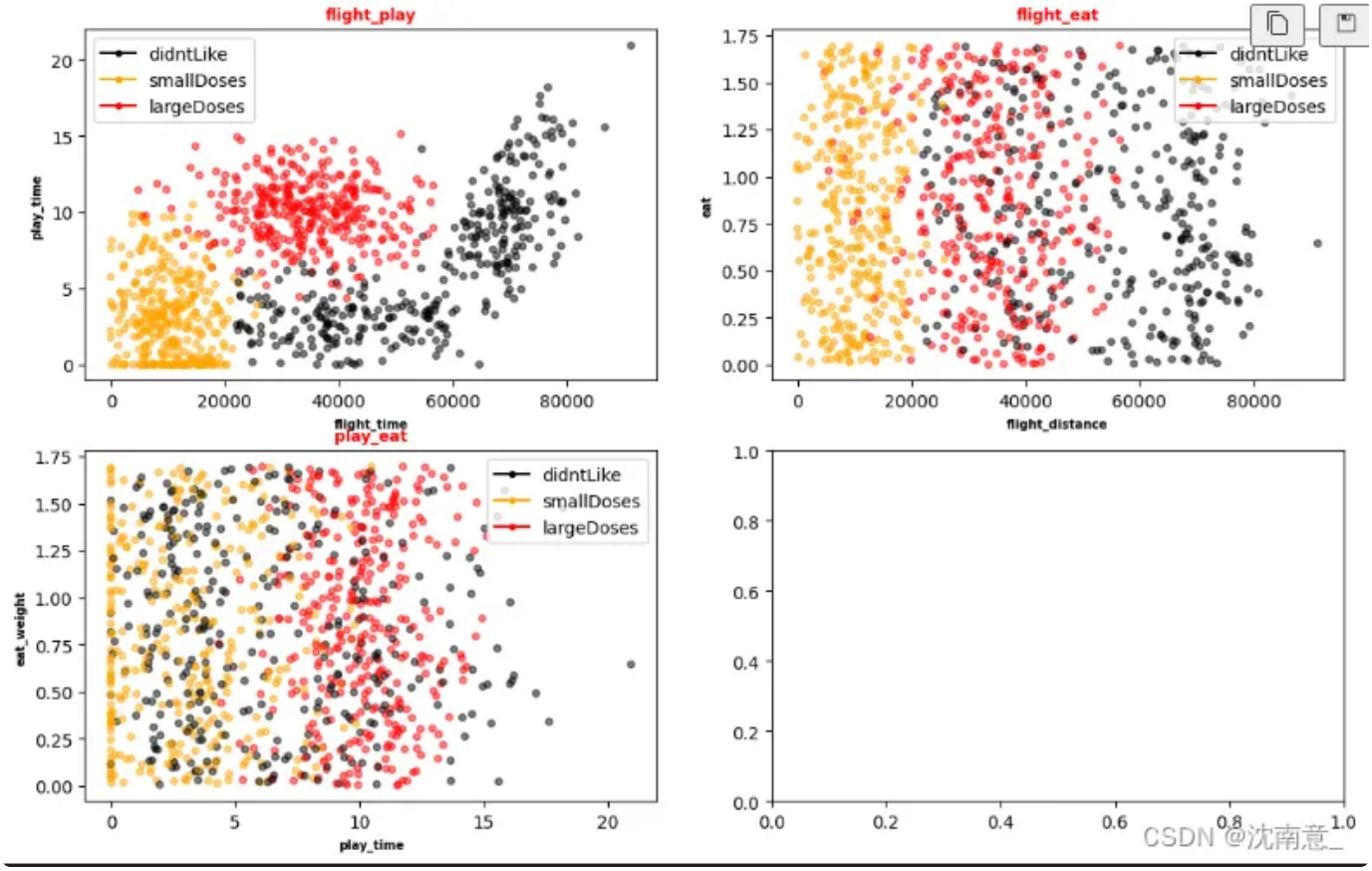
图中绿色的点就是我们要预测的那个点，假设K=3。那么KNN算法就会找到与它距离最近的三个点（这里用圆圈把它圈起来了），看看哪种类别多一些，比如这个例子中是蓝色三角形多一些，新来的绿色点就归类到蓝三角了。



但是，**当K=5的时候，判定就变成不一样了**。这次变成红圆多一些，所以新来的绿点被归类成红圆。从这个例子中，我们就能看得出K的取值是很重要的。

明白了大概原理后，我们就来说一说细节的东西吧，主要有两个：

- **1）K值的选取**
- **2）点距离的计算**



1.3、k最近邻算法的步骤描述如下：

1. 存在一个样本数据集合，也称为训练样本集，并且样本集中每个数据都存在标签，即我们知道样本集中每一个数据与所属分类的对应关系。
2. 计算待分类数据与训练集中每一个样本之间的距离。
3. 按照距离递增次序排序。
4. 找出与待分类样本距离最近的k个样本。
5. 确定前k个点所在类别的出现频率。
6. 返回前k个点所出现频率最高的类别作为当前测试点的预测分类。

2、KNN 算法的关键

2.1、K的取值

由于k值较大，预测标签比较稳定，但可能过平滑；k值较小，预测的标签比较容易受到样本的影响。往往K值较小且是奇数，避免产生占比相同

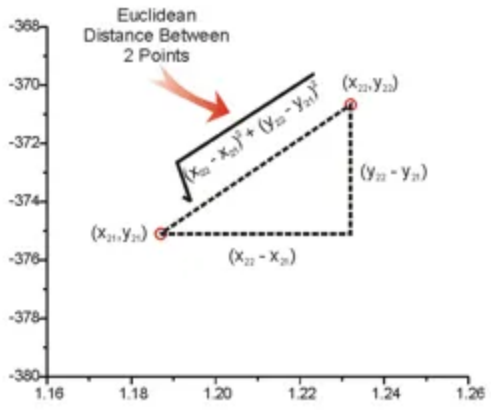
2.2、距离的测算

KNN算法通常的距离测算方式为欧式距离和曼哈顿距离，相比之下**欧氏距离会更常用**

而且 通常KNN算法中使用的是欧式距离

1） 欧式距离（L2 距离）：

这里只是简单说一下，拿二维平面为例，二维空间两个点的欧式距离计算公式如下：



$$\rho = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

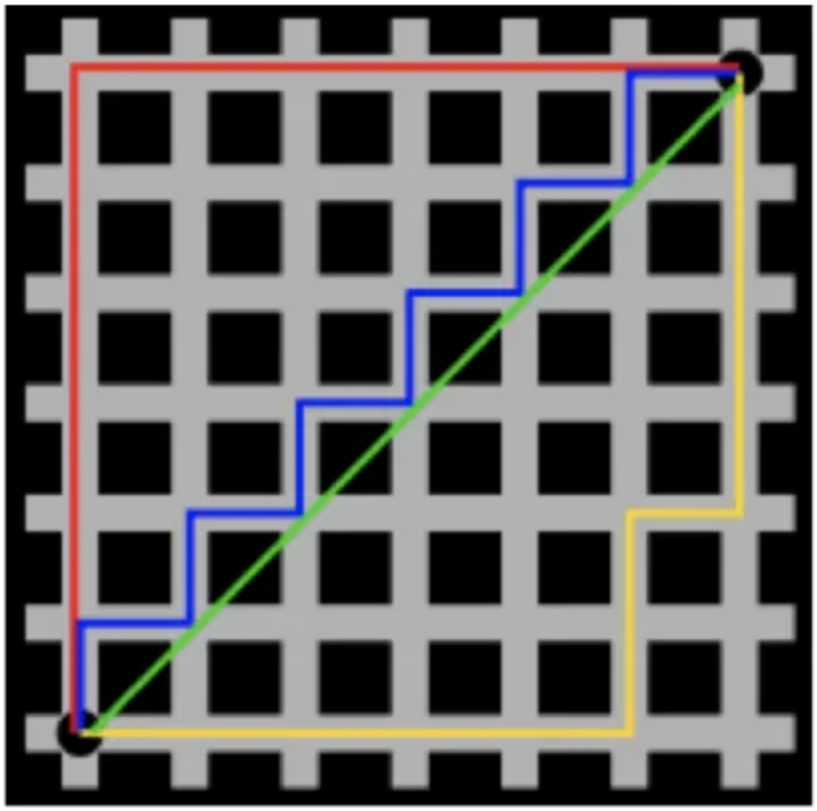
这个高中应该就有接触到的了，其实就是计算（x1,y1）和（x2,y2）的距离。拓展到多维空间，则公式变成这样：

$$d(x,y) := \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

KNN算法最简单粗暴的就是将预测点与所有点距离进行计算，然后保存并排序，选出前面K个值看看哪些类别比较多。

2）曼哈顿距离（L1 距离）：

曼哈顿距离（Manhattan distance），也称为**城市街区距离**或L1距离，是计算两点之间在网格状平面上的距离的方法。它得名于曼哈顿的街道规划，其中街道布局呈直角交叉的网格状，使得两点之间的行走距离为沿着街道的总长度。



在二维平面上，曼哈顿距离表示点(x1,y1)和点(x2,y2)之间的距离，可以用以下公式表示：

$$d = |x_2 - x_1| + |y_2 - y_1|$$

更一般地，在n维空间中，点(x1,x2,...,xn) 和点(y1,y2,...,yn) 之间的曼哈顿距离为：

$$d = \sum_{i-1}^{n} |y_i - x_i|$$

3、KNN算法的一般流程

（1）收集数据

（2）准备数据：距离计算所需要的数值，最好是结构化的数据格式

（3）分析数据

（4）测试算法：计算错误率

（5）使用算法：首先需要输入样本数据和结构化的输出结果，然后运行k-近邻算法判定输入数据分别属于哪个分类，最后应用对计算出的分类执行后续的处理。

3、KNN算法的伪代码

对未知类别属性的数据集中的每个点依次执行以下操作：

（1）计算已知类别数据集中的点与当前点之间的距离；

（2）按照距离递增次序排序；

（3）选取与当前点距离最小的k个点；

（4）确定前k个点所在类别的出现频率；

（5）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。

4、使用数学公式实现 KNN

```python
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# @Project : 机器学习
# @File    : knn_mathe.py
# @IDE     : PyCharm
# @Author  : jason
# @Date    : 2024/10/15 21:28
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

if __name__ == "__main__":
    #1. 定义数据集和测试点
    # 定义三个点集合
    point1 = [[7.7, 6.1], [3.1, 5.9], [8.6, 8.8], [9.5, 7.3], [3.9, 7.4], [5.0, 5.3], [1.0, 7.3]]
    point2 = [[0.2, 2.2], [4.5, 4.1], [0.5, 1.1], [2.7, 3.0], [4.7, 0.2], [2.9, 3.3], [7.3, 7.9]]
    point3 = [[9.2, 0.7], [9.2, 2.1], [7.3, 4.5], [8.9, 2.9], [9.5, 3.7], [7.7, 3.7], [9.4, 2.4]]

    # 合并数据集的特征
    np_train_data = np.concatenate((point1, point2, point3))
    print(f"合并之后的数据集的特征: {np_train_data}")

    # 生成对应的标签
    np_train_label = np.array([0] * len(point1) + [1] * len(point2) + [2] * len(point3))

    # 定义一个测试点坐标
    predict_point = np.array([3, 4.2])

    #2、定义K的值
    K = 3

    #-------------------------------------------------------------------------------
    #3、求距离,获取最短距离,最短距离对应的点的坐标
    # 使用numpy的广播机制去求距离
    # predict_point = np.array([1, 1])
    # np_train_data = np.array([[4, 5], [2, 3]])
    distances = np.sqrt(np.sum((predict_point - np_train_data)**2, axis=1))

    #距离排序,拿到索引
```

```
    index = np.argsort(distances)

    #取出距离最近的K个的索引
    nearest_index = index[:K]

    nearest_points = []
    nearest_distances = []
    nearest_label = []
    # 拿到距离最近的K个点的对应的坐标和距离
    for i in nearest_index:
        nearest_points.append(np_train_data[i])
        nearest_distances.append(distances[i])
        nearest_label.append(np_train_label[i])

    # print(max(nearest_label))
    counter = Counter(nearest_label)
    print("数据点的label为: ", counter.most_common()[0][0])

    #-------------------------------------------------------------------------
-------------------------
    plt.xlabel("x axis label")
    plt.ylabel("y axis label")
    plt.scatter(np_train_data[np_train_label == 0, 0], np_train_data[np_tr
ain_label == 0, 1], marker="*")
    plt.scatter(np_train_data[np_train_label == 1, 0], np_train_data[np_tr
ain_label == 1, 1], marker="^")
    plt.scatter(np_train_data[np_train_label == 2, 0], np_train_data[np_tr
ain_label == 2, 1], marker="s")

    plt.scatter(predict_point[0], predict_point[1], marker="o")

    for i in range(K):
        plt.plot([predict_point[0], nearest_points[i][0]], [predict_point[
1], nearest_points[i][1]])
        plt.annotate(f"{nearest_distances[i]:2.2f}",
                     xy=((predict_point[0] + nearest_points[i][0]) / 2, (p
redict_point[1] + nearest_points[i][1]) / 2))

    plt.show()
```