kd-tree笔记

Honghui Wang

May 31, 2015

1 Kd-树概念

Kd-树其实是K-dimension tree的缩写，是对数据点在k维空间中划分的一种数据结构。其实，Kd-树是一种平衡二叉树。

举一示例:

假设有六个二维数据点= $\{(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)\}$，数据点位于二维空间中。为了能有效的找到最近邻，Kd-树采用分而治之的思想，即将整个空间划分为几个小部分。六个二维数据点生成的Kd-树如图1所示。

k-d树算法可以分为两大部分，一部分是有关k-d树本身这种数据结构建立的算法，另一部分是在建立的k-d树上如何进行最邻近查找的算法。

1.1 kd树的构建

Kd-树是一个二叉树，每个节点表示的是一个空间范围。表1表示的是Kd-树中每个节点中主要包含的数据结构。

Range域表示的是节点包含的空间范围。

Node-data域就是数据集中的某一个n维数据点。分割超面是通过数据点Node-Data并垂直于轴split的平面，分割超面将整个空间分割成两个子空间。

令split域的值为i，如果空间Range中某个数据点的第i维数据小于Node-Data[i]，那么，它就属于该节点空间的左子空间，否则就属于右子空间。

Left，Right域分别表示由左子空间和右子空间空的数据点构成的Kd-树。

从上面对k-d树节点的数据类型的描述可以看出构建k-d树是一个逐级展开的递归过程。下面是给出的是构建k-d树的伪码。

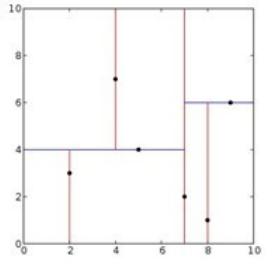

Figure 1: 2D对应的kd的平面划分

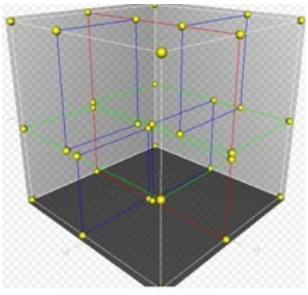

Figure 2: 3D对应的kd的平面划分

Table 1: kd树节点

域名	数据类型	描述
Node-data	数据矢量	数据集中某个数据点，是n维矢量（这里也就是k维）
Range	空间矢量	该节点所代表的空间范围
split	整数	垂直于分割超平面的方向轴序号
Left	k-d树	由位于该节点分割超平面左子空间内所有数据点所构成的k-d树
Right	k-d树	由位于该节点分割超平面右子空间内所有数据点所构成的k-d树
parent	k-d树	父节点

1.2 构建k-d树的算法实现

算法：构建k-d树(createKDTree)

输入：数据点集Data-set和其所在的空间Range

输出：Kd，类型为k-d tree

1、If Data-set为空，则返回空的k-d tree

2、调用节点生成程序：

（1）确定split域：对于所有描述子数据（特征矢量），统计它们在每个维上的数据方差。以SURF特征为例，描述子为64维，可计算64个方差。挑选出最大值，对应的维就是split域的值。数据方差大表明沿该坐标轴方向上的数据分散得比较开，在这个方向上进行数据分割有较好的分辨率；

（2）确定Node-data域：数据点集Data-set按其第split域的值排序。位于正中间的那个数据点被选为Node-data。此时新的Data-set' = Data-set\Node-data（除去其中Node-data这一点）。

3、dataleft = {d属于Data-set' && $d[split] \leq Node - data[split]$}

Left_Range = {Range && dataleft} dataright = {d属于Data-set' && $d[split] > Node - data[split]$}

Right_Range = {Range && dataright}

4.left = 由(dataleft，Left_Range)建立的k-d tree，即递归调用createKDTree(dataleft，Left_Range)。并设置left的parent域为Kd;

right = 由(dataright，Right_Range)建立的k-d tree，即调用createKDTree(dataright，Right_Range)。并设置right的parent域为Kd。

1.3 构建k-d树算法举例

从上述举的实例来看，过程如下：

（1）确定：split 域=x，6个数据点在x,y 维度上的数据方差为39,28.63.在x轴方向上的方差大，所以split域值为x。

（2）确定：Node-Data=（7,2），根据x维上的值将数据排序，6个数据的中值为7，所以node-data域为数据点（7,2）。这样该节点的分割超面就是通过（7,2）并垂直于：split=x轴的直线x=7.

(3) 确定：左子空间和右子空间，分割超面x=7将整个空间分为两部分。x_j=7为左子空间，包含节点（2,3），（5,4），（4，7），另一部分为右子空间。包含节点（9,6），（8,1）

这个构建过程是一个递归过程。重复上述过程，直至只包含一个节点。

如算法所述，k-d树的构建是一个递归的过程。然后对左子空间和右子空间内的数据重复根节点的过程就可以得到下一级子节点（5,4）和（9,6）（也就是左右子空间的'根'节点），同时将空间和数据集进一步细分。如此反复直到空间中只包含一个数据点，如图3所示。最后生成的k-d树如图4所示。

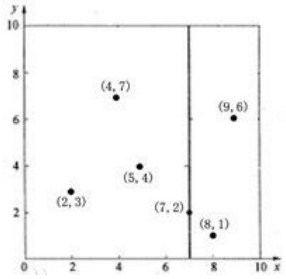

Figure 3:

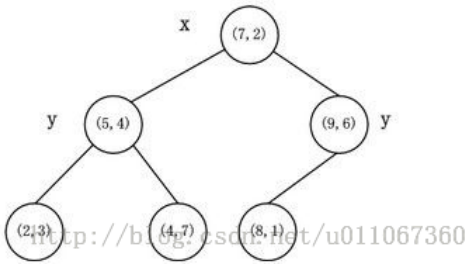

Figure 4:

2 二、构建完kd树之后，如今进行最近邻搜索呢?

2.1 KD树的查找算法:

在k-d树中进行数据的查找也是特征匹配的重要环节，其目的是检索在k-d树中与查询点距离最近的数据点。

这里先以一个简单的实例来描述最邻近查找的基本思路。

2.2 例一：查询的点（2.1,3.1）（较简单）。

1、如图3所示，星号表示要查询的点（2.1,3.1）。通过二叉搜索，顺着搜索路径很快就能找到最邻近的近似点，也就是叶子节点（2,3）。

2、而找到的叶子节点并不一定就是最邻近的，最邻近肯定距离查询点更近，应该位于以查询点为圆心且通过叶子节点的圆域内。

3、为了找到真正的最近邻，还需要进行'回溯'操作:

算法沿搜索路径反向查找是否有距离查询点更近的数据点。

此例中先从（7,2）点开始进行二叉查找，然后到达（5,4），最后到达（2,3），此时搜索路径中的节点为{（7,2），（5,4），（2,3）}

首先以（2,3）作为当前最近邻点，计算其到查询点（2.1,3.1）的距离为0.1414，

然后回溯到其父节点（5,4），并判断在该父节点的其他子节点空间中是否有距离查询点更近的数据点。以（2.1,3.1）为圆心，以0.1414为半径画圆，如图5所示。发现该圆并不和超平面y = 4交割，因此不用进入（5,4）节点右子空间中去搜索。

4、最后，再回溯到（7,2），以（2.1,3.1）为圆心，以0.1414为半径的圆更不会与x=7超平面交割，因此不用进入（7,2）右子空间进行查找。至此，搜索路径中的节点已经全部回溯完，结束整个搜索，返回最近邻点（2,3），最近距离为0.1414。

2.3 例二：查找点为（2，4.5）（叫复杂一点）。

一个复杂点了例子如查找点为（2，4.5）。

1、同样先进行二叉查找，先从（7,2）查找到（5,4）节点，在进行查找时是由y=4为分割超平面的，由于查找点为y值为4.5，因此进入右子空间查找到（4,7），形成搜索路径{（7,2），（5,4），（4,7）}

2、取（4,7）为当前最近邻点，计算其与目标查找点的距离为3.202。然后回溯到（5,4），计算其与查找点之间的距离为3.041。

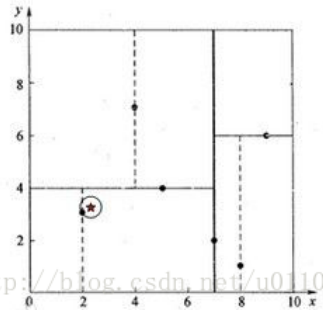

Figure 5:

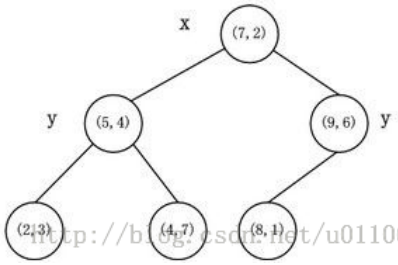

Figure 6:

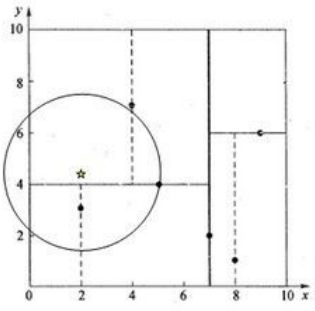

Figure 7:

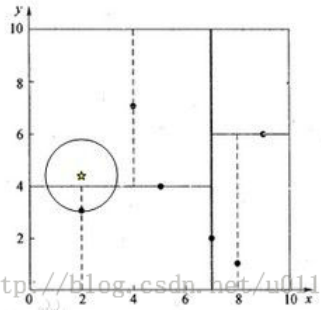

Figure 8:

（（4,7）与目标查找点的距离为3.202，而（5,4）与查找点之间的距离为3.041，所以（5,4）为查询点的最近点；

3、以（2，4.5）为圆心，以3.041为半径作圆，如图4所示。可见该圆和y = 4超平面交割，所以需要进入（5,4）左子空间进行查找。此时需将（2,3）节点加入搜索路径中得{（7,2），（2,3）}。

4、回溯至（2,3）叶子节点，（2,3）距离（2,4.5）比（5,4）要近，所以最近邻点更新为（2，3），最近距离更新为1.5。

5、回溯至（7,2），以（2,4.5）为圆心1.5为半径作圆，并不和x = 7分割超平面交割，如图8所示。

至此，搜索路径回溯完。返回最近邻点（2,3），最近距离1.5。

2.4 k-d树查询算法的简要说明：

从root节点开始，DFS搜索直到叶子节点，同时在stack中顺序存储已经访问的节点。

如果搜索到叶子节点，当前的叶子节点被设为最近邻节点。

然后通过stack回溯:

如果当前点的距离比最近邻点距离近，更新最近邻节点.

然后检查以最近距离为半径的圆是否和父节点的超平面相交.

如果相交，则必须到父节点的另外一侧，用同样的DFS搜索法，开始检查最近邻节点。

如果不相交，则继续往上回溯，而父节点的另一侧子节点都被淘汰，不再考虑的范围中.

当搜索回到root节点时，搜索完成，得到最近邻节点。

当然涉及到KD树的操作还有插入和删除等，但是k近邻算法主要就是用到查找元素，这里就不再写了。