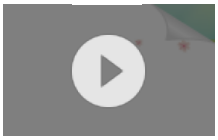
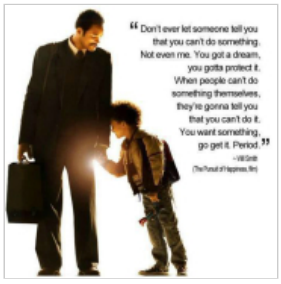


小虫不会飞_的博客

http://blog.sina.com.cn/yancong2011 ［订阅］［手机订阅］



小虫不会飞_

微博

加好友 发纸条

写留言 加关注

转载：Kd-Tree算法原理和开源实现代码

(2013-11-03 20:03:15) 转 载 ▼

标签： kd-tree bbf it 分类： 数据挖掘

Kd-Tree算法原理和开源实现代码

本文介绍一种用于高维空间中的快速最近邻和近似最近邻查找技术——Kd-Tree（Kd树）。Kd-Tree，即K-dimensional tree，是一种高维索引树形数据结构，常用于在大规模的高维数据空间进行最近邻查找(Nearest Neighbor)和近似最近邻查找(Approximate Nearest Neighbor)，例如图像检索和识别中的高维图像特征向量的K近邻查找与匹配。本文首先介绍Kd-Tree的基本原理，然后对基于BBF的近似查找方法进行介绍，最后给出一些参考文献和开源实现代码。

一、Kd-tree

Kd-Tree，即K-dimensional tree，是一棵二叉树，树中存储的是一些K维数据。在一个K维数据集合上构建一棵Kd-Tree代表了对该K维数据集合构成的K维空间的一个划分，即树中的每个结点就对应了一个K维的超矩形区域（Hyperrectangle）。

在介绍Kd-tree的相关算法前，我们先回顾一下二叉查找树（Binary Search Tree）的相关概念和算法。

二叉查找树（Binary Search Tree，BST），是具有如下性质的二叉树（来自wiki）：

1）若它的左子树不为空，则左子树上所有结点的值均小于它的根结点的值；

2）若它的右子树不为空，则右子树上所有结点的值均大于它的根结点的值；

3）它的左、右子树也分别为二叉排序树；

例如，图1中是一棵二叉查找树，其满足BST的性质。

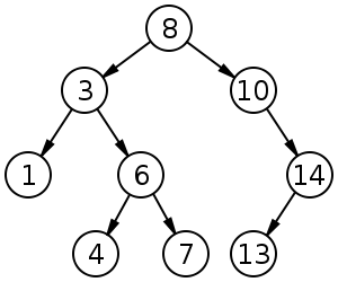


图1 二叉查找树（来源：Wiki）

给定一个1维数据集合，怎样构建一棵BST树呢？根据BST的性质就可以创建，即将数据点一个一个插入到BST树中，插入后的树仍然是BST树，即根结点的左子树中所有结点的值均小于根结点的值，而根结点的右子树中所有结点的值均大于根结点的值。

将一个1维数据集用一棵BST树存储后，当我们想要查询某个数据是否位于该数据集合中时，只需要将查询数据与结点值进行比较然后选择对应的子树继续往下查找即可，查找的平均时间复杂度

为：O(logN)，最坏的情况下是O(N)。

如果我们要处理的对象集合是一个K维空间中的数据集，那么是否也可以构建一棵类似于1维空间中的二叉查找树呢？答案是肯定的，只不过推广到K维空间后，创建二叉树和查询二叉树的算法会有一些相应的变化（后面会介绍到两者的区别），这就是下面我们要介绍的Kd-tree算法。

怎样构造一棵Kd-tree？

对于Kd-tree这样一棵二叉树，我们首先需要确定怎样划分左子树和右子树，即一个K维数据是依据什么被划分到左子树或右子树的。

在构造1维BST树时，一个1维数据根据其与树的根结点和中间结点进行大小比较的结果来决定是划分到左子树还是右子树，同理，我们也可以按照这样的方式，将一个K维数据与Kd-tree的根结点和中间结点进行比较，只不过不是对K维数据进行整体的比较，而是选择某一个维度Di，然后比较两个K维数在该维度Di上的大小关系，即每次选择一个维度Di来对K维数据进行划分，相当于用一个垂直于该维度Di的超平面将K维数据空间一分为二，平面一边的所有K维数据在Di维度上的值小于平面另一边的所有K维数据对应维度上的值。也就是说，我们每选择一个维度进行如上的划分，就会将K维数据空间划分为两个部分，如果我们继续分别对这两个子K维空间进行如上的划分，又会得到新的子空间，对新的子空间又继续划分，重复以上过程直到每个子空间都不能再划分为止。以上就是构造Kd-Tree的过程，上述过程中涉及到两个重要的问题：1）每次对子空间的划分时，怎样确定在哪个维度上进行划分；2）在某个维度上进行划分时，怎样确保在这一维度上的划分得到的两个子集合的数量尽量相等，即左子树和右子树中的结点个数尽量相等。

问题1： 每次对子空间的划分时，怎样确定在哪个维度上进行划分？

最简单的方法就是轮着来，即如果这次选择了在第i维上进行数据划分，那下一次就在第j(j≠i)维上进行划分，例如：j = (i mod k) + 1。想象一下我们切豆腐时，先是竖着切一刀，切成两半后，再横着来一刀，就得到了很小的方块豆腐。

可是“轮着来”的方法是否可以很好地解决问题呢？再次想象一下，我们现在要切的是一根木条，按照“轮着来”的方法先是竖着切一刀，木条一分为二，干净利落，接下来就是再横着切一刀，这个时候就有点考验刀法了，如果木条的直径（横截面）较大，还可以下手，如果直径较小，就没法往下切了。因此，如果K维数据的分布像上面的豆腐一样，“轮着来”的切分方法是可以奏效，但是如果K维度上数据的分布像木条一样，“轮着来”就不好用了。因此，还需要想想其他的切法。

如果一个K维数据集合的分布像木条一样，那就是说明这K维数据在木条较长方向代表的维度上，这些数据的分布散得比较开，数学上来说，就是这些数据在该维度上的方差（invariance）比较大，换句话说，正因为这些数据在该维度上分散的比较开，我们就更容易在这个维度上将它们划分开，因此，这就引出了我们选择维度的另一种方法：最大方差法（max invarince），即每次我们选择维度进行划分时，都选择具有最大方差维度。

问题2：在某个维度上进行划分时，怎样确保在这一维度上的划分得到的两个子集合的数量尽量相等，即左子树和右子树中的结点个数尽量相等？

假设当前我们按照最大方差法选择了在维度i上进行K维数据集S的划分，此时我们需要在维度i上将K维数据集合S划分为两个子集合A和B，子集合A中的数据在维度i上的值都小于子集合B中。首先考虑最简单的划分法，即选择第一个数作为比较对象（即划分轴，pivot），S中剩余的其他所有K维数据都跟该pivot在维度i上进行比较，如果小于pivot则划A集合，大于则划入B集合。把A集合和B集合分别看做是左子树和右子树，那么我们在构造一个二叉树的时候，当然是希望它是一棵尽量平衡的树，即左右子树中的结点个数相差不大。而A集合和B集合中数据的个数显然跟pivot值有关，因为它们是跟pivot比较后才被划分到相应的集合中去的。好了，现在的问题就是确定pivot了。给定一个数组，怎样才能得到两个子数组，这两个数组包含的元素个数差不多且其中一个子数组中的元素值都小于另一个子数组呢？方法很简单，找到数组中的中值（即中位数，median），然后将数组中所有元素与中值进行比较，就可以得到上述两个子数组。同样，在维度i上进行划分时，pivot就选择该维度i上所有数据的中值，这样得到的两个子集合数据个数就基本相同了。

解决了上面两个重要的问题后，就得到了Kd-Tree的构造算法了。

Kd-Tree的构建算法：

（1）在K维数据集合中选择具有最大方差的维度k，然后在该维度上选择中值m为pivot对该数据集合进行划分，得到两个子集合；同时创建一个树结点node，用于存储；

（2）对两个子集合重复（1）步骤的过程，直至所有子集合都不能再划分为止；如果某个子集合不能再划分时，则将该子集合中的数据保存到叶子结点（leaf node）。

以上就是创建Kd-Tree的算法。下面给出一个简单例子。

给定二维数据集合：(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)，利用上述算法构建一棵Kd-tree。左图是Kd-tree对应二维数据集合的一个空间划分，右图是构建的一棵Kd-tree。

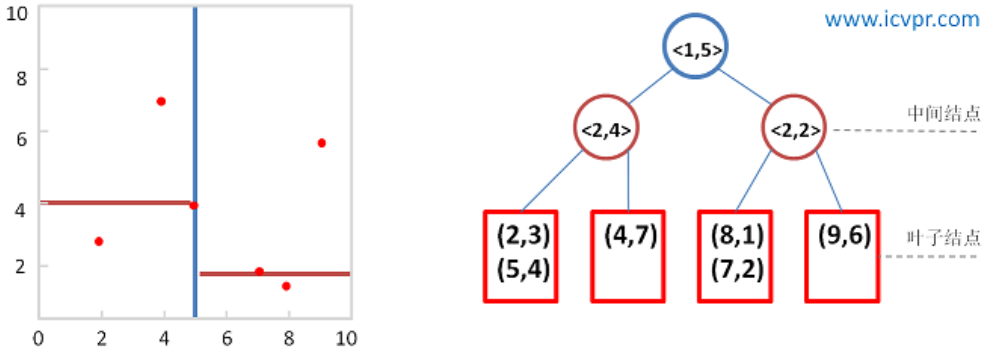


图2 构建的kd-tree

其中圆圈代表了中间结点(k, m)，而红色矩形代表了叶子结点。

Kd-Tree与一维二叉查找树之间的区别：

二叉查找树：数据存放在树中的每个结点（根结点、中间结点、叶子结点）中；

Kd-Tree：数据只存放在叶子结点，而根结点和中间结点存放一些空间划分信息（例如划分维度、划分值）；

构建好一棵Kd-Tree后，下面给出利用Kd-Tree进行最近邻查找的算法：

（1）将查询数据Q从根结点开始，按照Q与各个结点的比较结果向下访问Kd-Tree，直至达到叶子结点。

其中Q与结点的比较指的是将Q对应于结点中的k维度上的值与m进行比较，若Q(k) < m，则访问左子树，否则访问右子树。达到叶子结点时，计算Q与叶子结点上保存的数据之间的距离，记录下最小距离对应的数据点，记为当前“最近邻点”Pcur和最小距离Dcur。

（2）进行回溯（Backtracking）操作，该操作是为了找到离Q更近的“最近邻点”。即判断未被访问过的分支里是否还有离Q更近的点，它们之间的距离小于Dcur。

如果Q与其父结点下的未被访问过的分支之间的距离小于Dcur，则认为该分支中存在离P更近的数据，进入该结点，进行（1）步骤一样的查找过程，如果找到更近的数据点，则更新为当前的“最近邻点”Pcur，并更新Dcur。

如果Q与其父结点下的未被访问过的分支之间的距离大于Dcur，则说明该分支内不存在与Q更近的点。

回溯的判断过程是从下往上进行的，直到回溯到根结点时已经不存在与P更近的分支为止。

怎样判断未被访问过的树分支Branch里是否还有离Q更近的点？

从几何空间上来看，就是判断以Q为中心center和以Dcur为半径Radius的超球面（Hypersphere）与树分支Branch代表的超矩形（Hyperrectangle）之间是否相交。

在实现中，我们可以有两种方式来求Q与树分支Branch之间的距离。第一种是在构造树的过程中，就记录下每个子树中包含的所有数据在该子树对应的维度k上的边界参数[min, max]；第二种是在构造树的过程中，记录下每个子树所在的分割维度k和分割值m，（k, m），Q与子树的距离则为|Q(k) - m|。

以上就是Kd-tree的构造过程和基于Kd-Tree的最近邻查找过程。

下面用一个简单的例子来演示基于Kd-Tree的最近邻查找的过程。

数据点集合：(2,3), (4,7), (5,4), (9,6), (8,1), (7,2) 。

已建好的Kd-Tree：

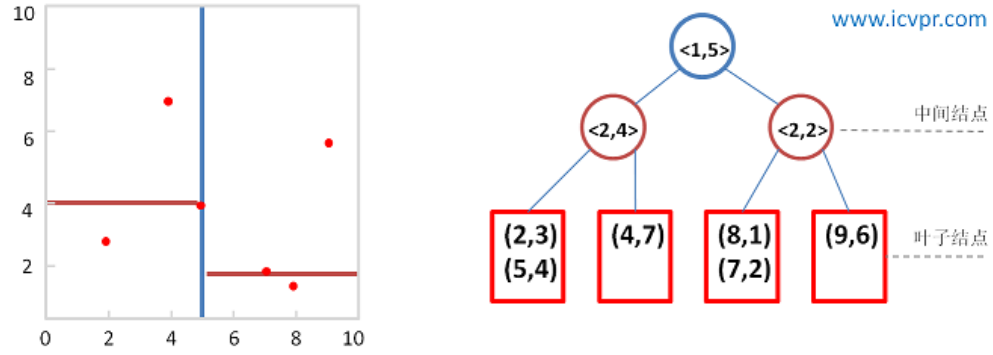


图3 构建的kd-tree

其中，左图中红色点表示数据集合中的所有点。

查询点：(8, 3)（在左图中用茶色菱形点表示）

第一次查询：

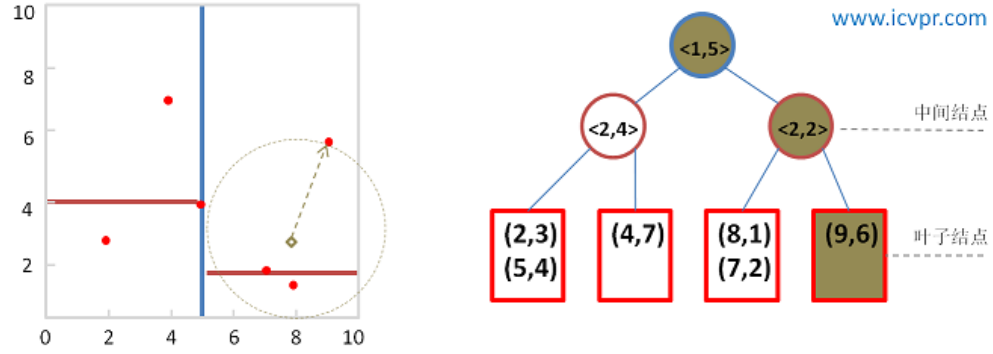
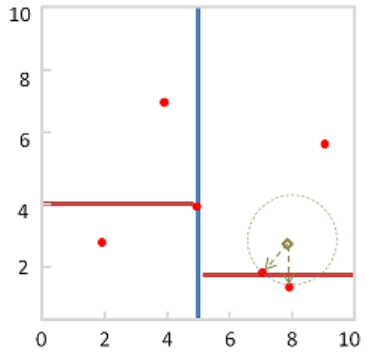


图4 第一次查询的kd-tree

当前最近邻点：(9, 6)，最近邻距离：sqrt(10)，

且在未被选择的树分支中存在于Q更近的点（如茶色圈圈内的两个红色点）

回溯：



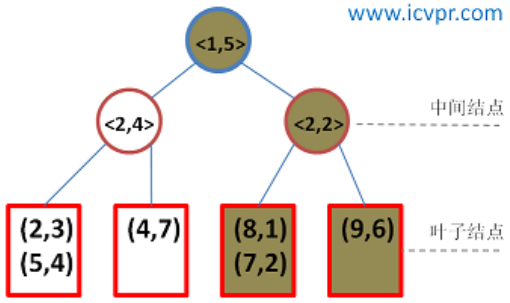


图5 回溯kd-tree

当前最近邻点： (8, 1)和(7, 2) ， 最近邻距离： sqrt(2)

最后，查询点(8, 3)的近似最近邻点为(8, 1)和(7, 2) 。

二、Kd-tree with BBF

上一节介绍的Kd-tree在维度较小时（例如：K≤30 ），算法的查找效率很高，然而当Kd-tree用于对高维数据（例如：K≥100 ）进行索引和查找时，就面临着维数灾难（curse of dimension ）问题，查找效率会随着维度的增加而迅速下降。通常，实际应用中，我们常常处理的数据都具有高维的特点，例如在图像检索和识别中，每张图像通常用一个几百维的向量来表示，每个特征点的局部特征用一个高维向量来表征（例如：128维的SIFT特征 ）。因此，为了能够让Kd-tree满足对高维数据的索引，Jeffrey S. Beis和David G. Lowe提出了一种改进算法——Kd-tree with BBF（ Best Bin First ），该算法能够实现近似K近邻的快速搜索，在保证一定查找精度的前提下使得查找速度较快。

在介绍BBF算法前，我们先来看一下原始Kd-tree是**为什么在低维空间中有效而到了高维空间后查找效率就会下降**。在原始kd-tree的最近邻查找算法中（第一节中介绍的算法 ），为了能够找到查询点Q在数据集合中的最近邻点，有一个重要的操作步骤：回溯，该步骤是在未被访问过的且与Q的超球面相交的子树分支中查找可能存在的最近邻点。随着维度K的增大，与Q的超球面相交的超矩形（子树分支所在的区域 ）就会增加，这就意味着需要回溯判断的树分支就会更多，从而算法的查找效率便会下降很大。

一个很自然的思路是：既然kd-tree算法在高维空间中是由于过多的回溯次数导致算法查找效率下降的话，我们就可以限制查找时进行回溯的次数上限，从而避免查找效率下降。这样做有两个问题需要解决：1 ）最大回溯次数怎么确定？2 ）怎样保证在最大回溯次数内找到的最近邻比较接近真实最近邻，即查找准确度不能下降太大。

问题1 ）：最大回溯次数怎么确定？

最大回溯次数一般人为设定，通常根据在数据集上的实验结果进行调整。

问题2 ）：怎样保证在最大回溯次数内找到的最近邻比较接近真实最近邻，即查找准确度不能下降太大？

限制回溯次数后，如果我们还是按照原来的回溯方法挨个地进行访问的话，那很显然最后的查找结果的精度就很大程度上取决于数据的分布和回溯次数了。挨个访问的方法的问题在于认为每个待回溯的树分支中存在最近邻的概率是一样的，所以它对所有的待回溯树分支一视同仁。实际上，在这些待回溯树分支中，有些树分支存在最近邻的可能性比其他树分支要高，因为树分支离Q点之间的距离或相交程度是不一样的，离Q更近的树分支存在Q的最近邻的可能性更高。因此，我们需要区别对待每个待回溯的树分支，即采用某种优先级顺序来访问这些待回溯树分支，使得在有限的回溯次数中找到Q的最近邻的可能性很高。我们要介绍的BBF算法正是基于这样的解决思路，下面我们介绍BBF查找算法。

基于BBF的Kd-Tree近似最近邻查找算法

已知：

Q：查询数据； KT：已建好的Kd-Tree；

1. 查找Q的当前最近邻点P

1）从KT的根结点开始，将Q与中间结点node(k,m)进行比较，根据比较结果选择某个树分支Branch（或称为Bin）；并将未被选择的另一个树分支（Unexplored Branch）所在的树中位置和它跟Q之间的距离一起保存到一个优先级队列中Queue；

2）按照步骤1）的过程，对树分支Branch进行如上比较和选择，直至访问到叶子结点，然后计算Q与叶子结点中保存的数据之间的距离，并记录下最小距离D以及对应的数据P。

注：

A、Q与中间结点node(k,m)的比较过程：如果Q(k) > m则选择右子树，否则选择左子树。

B、优先级队列：按照距离从小到大的顺序排列。

C、叶子结点：每个叶子结点中保存的数据的个数可能是一个或多个。

2. 基于BBF的回溯

已知：最大回溯次数BTmax

1）如果当前回溯的次数小于BTmax，且Queue不为空，则进行如下操作：

从Queue中取出最小距离对应的Branch，然后按照1.1步骤访问该Branch直至达到叶子结点；计算Q与叶子结点中各个数据间距离，如果有比D更小的值，则将该值赋给D，该数据则被认为是Q的当前近似最近邻点；

2）重复1）步骤，直到回溯次数大于BTmax或Queue为空时，查找结束，此时得到的数据P和距离D就是Q的近似最近邻点和它们之间的距离。

下面用一个简单的例子来演示基于Kd-Tree+BBF的近似最近邻查找的过程。

数据点集合：(2,3), (4,7), (5,4), (9,6), (8,1), (7,2) 。

已建好的Kd-Tree：

图6 构建的kd-tree

基于BBF的查找的过程：

查询点Q： (5.5, 5)

第一遍查询：

图7 第一次查询的kd-tree

当前最近邻点： (9, 6) ， 最近邻距离： sqrt(13.25)，
同时将未被选择的树分支的位置和与Q的距离记录到优先级队列中。

BBF回溯：
从优先级队列里选择距离Q最近的未被选择树分支进行回溯。

图8 利用BBF方法回溯kd-tree

当前最近邻点： (4, 7) ， 最近邻距离： sqrt(6.25)

继续从优先级队列里选择距离Q最近的未被选择树分支进行回溯。

图9 利用BBF方法回溯kd-tree

当前最近邻点： (5, 4) ， 最近邻距离： sqrt(1.25)

最后，查询点(5.5, 5)的近似最近邻点为(5, 4) 。

三、参考文献

Paper

[1] Multidimensional binary search trees used for associative searching

[2] Shape indexing using approximate nearest-neighbour search in high-dimensional spaces

Tutorial

[1] An introductory tutorial on kd trees

[2] Nearest-Neighbor Methods in Learning and Vision: Theory and Practice

Website

[1] wiki: http://en.wikipedia.org/wiki/K-d_tree

Code

[1] OpenCV FLANN

[2] VLFeat

[3] FLANN

[4] KD-Tree Implementation in Java and C#

[5] C/C++

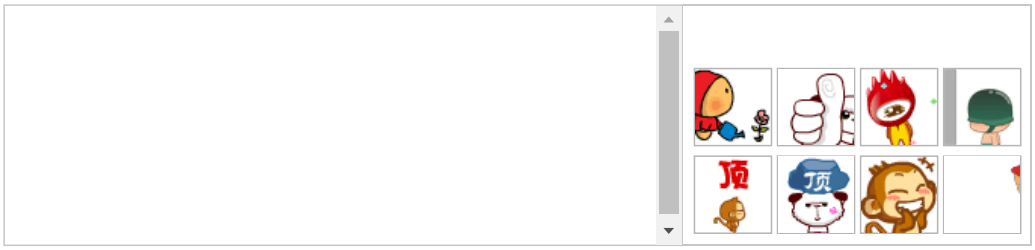
http://code.google.com/p/kdtree/

https://github.com/sdeming/kdtree

出处：http://www.icvpr.com/kd-tree-tutorial-and-code/

前一篇：opencv中使用SurfFeatureDetector,BruteForceMatcher

后一篇：距离汇总



登录名： 密码： 找回密码 注册 记住登录状态

分享到微博 评论并转载此博文

按住左边滑块，拖动完成上方拼图

发评论

〈 前一篇
opencv中使用SurfFeatureDetector,BruteForceMatcher

后一篇 〉
距离汇总