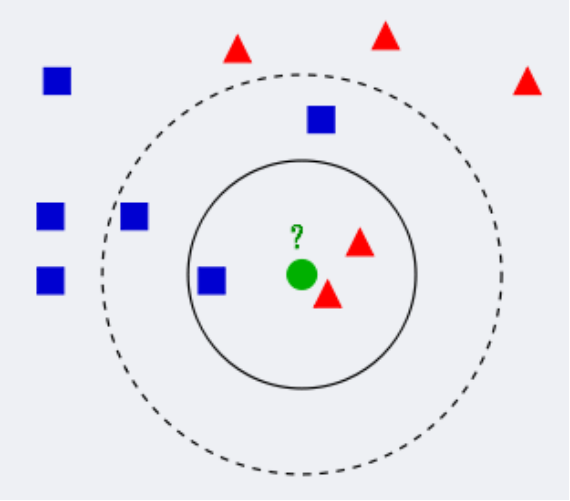
1.k近邻算法

1.1 什么是k近邻算法

k近邻，k-nearest neighbor algorithm,简称KNN。k=1时，算法变成最近邻算法。给定一个训练数据集，对新的输入实例，在训练数据集中找到与该实例最邻近的k个实例，这k个实例的多数属于某个类，就把该输入实例分类到这个类中。



1.2 近邻的距离度量表示方法

特征空间中两个实例点的距离可以反应出两个实例点之间的相似程度。k近邻模型的特征空间一般是n维实数向量空间，使用的距离可以是欧式距离，也可以是其他距离。几个距离度量：

1. 欧式距离

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

2. 曼哈顿距离

$$d_{12} = |x_1-x_2| + |y_1-y_2|$$

3. 马氏距离

有M个样本向量X1~Xm，协方差矩阵记为S，均值记为向量μ，则其中样本向量X到u的马氏距离表示为：

$$D(X) = \sqrt{(X-\mu)^T S^{-1}(X-\mu)}$$

（协方差矩阵中每个元素是各个矢量元素之间的协方差Cov(X,Y)，Cov(X,Y) = E{ [X-E(X)] [Y-E(Y)]}，其中E为数学期望）

而其中向量Xi与Xj之间的马氏距离定义为：

$$D(X_i,X_j) = \sqrt{(X_i-X_j)^T S^{-1}(X_i-X_j)}$$

若协方差矩阵是单位矩阵（各个样本向量之间独立同分布），则公式就成了：

$$D(X_i,X_j) = \sqrt{(X_i-X_j)^T (X_i-X_j)}$$

1.3 k值的选择

k值得选择对算法结果会产生重大影响：

- 如果选择较小的k值，就相当于用较小的领域中的训练实例进行预测，“学习”近似误差会减小，只有与输入实例较近或相似的训练实例才会对预测结果起作用，与此同时带来的问题是，估计误差会增大。换句话说，k值得减小意味着整体模型变得复杂，容易发生过拟合
- 如果选择较大的k值，就相当于用较大领域中的训练实例进行预测，其优点是可以减少学习的估计误差，但缺点是学习的近似误差会增大。这时候，与输入实例较远（不相似的）训练实例也会对预测起作用，使预测发生错误，且k值得增大意味着整体的模型变得简单
- 如果k=N，则完全不足取，因为此时无论输入实例是什么，都只是简单的预测它属于在训练实例中最多的类，模型过于简单，忽略了训练实例中大量有用信息。

- 在实际应用中，一般取一个比较小的数值，通常采用交叉验证法来选取最优的k值

2. K近邻算法的实现-kd树

2.1 背景

之前blog内曾经介绍过SIFT特征匹配算法，特征点匹配和数据库查、图像检索本质上是同一个问题，都可以归结为一个通过距离函数在高维矢量之间进行相似性检索的问题，如何快速而准确地找到查询点的近邻，不少人提出了很多高维空间索引结构和近似查询的算法。

一般说来，索引结构中相似性查询有两种基本的方式：

一种是范围查询，范围查询时给定查询点和查询距离阈值，从数据集中查找所有与查询点距离小于阈值的数据

另一种是K近邻查询，就是给定查询点及正整数K，从数据集中找到距离查询点最近的K个数据，当K=1时，它就是最近邻查询。

同样，针对特征点匹配也有两种方法：

- 最容易的办法就是线性扫描，也就是我们常说的穷举搜索，依次计算样本集E中每个样本到输入实例点的距离，然后抽取出计算出来的最小距离的点即为最近邻点。此种办法简单直白，但当样本集或训练集很大时，它的缺点就立马暴露出来了，举个例子，在物体识别的问题中，可能有数千个甚至数万个SIFT特征点，而去一一计算这成千上万的特征点与输入实例点的距离，明显是不足取的。
- 另外一种，就是构建数据索引，因为实际数据一般都会呈现簇状的聚类形态，因此我们想到建立数据索引，然后再进行快速匹配。索引树是一种树结构索引方法，其基本思想是对搜索空间进行层次划分。根据划分的空间是否有混叠可以分为Clipping和Overlapping两种。前者划分空间没有重叠，**其代表就是k-d树**；后者划分空间相互有交叠，其代表为R树。

2.2 什么是kd树

Kd树是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，主要应用于多维空间关键数据的搜索（如：范围搜索和最近邻搜索）。本质上说，kd树就是一种平衡二叉树。

2.3 kd树的构建

算法：构建 Kd-树

输入：数据点集 Data-set 和其所在的空间：Range

输出：Kd，类型为 Kd-tree

1. If Data-set 是空的，则返回空的 Kd-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 为例，描述子为 64 维，可计算出 64 个方差。挑选出方差中的最大值，对应的维就是:split 域的值。数据方差最大表明沿该坐标轴方向上数据点分散得比较开，这个方向上进行数据分割可以获得最好的分辨率；

（2） 确定 Node-data 域：数据点集 Data-set 按其第 split 维的值排序，位于正中间的那个数据点被选为 Node-data，Data-set'=Data-set\Node-data，

3. $dataleft = \{d \in Data-set' \&\& d[:split] \leq Node-data[:split]\}$

   Left_Range={Range && dataleft}

   $dataright = \{d \in Data-set' \&\& d[:split] > Node-data[:split]\}$

   Right_Range={Range && dataright}

4. :left= 由(dataleft,Left_Range)建立的 Kd-tree

   设置:*left* 的 *parent* 域（父节点）为 Kd

   :right=由 (*dataright,Right_Range*)建立的 Kd-tree

设置:*right* 的 *parent* 域（父节点）为 Kd

假设有6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内。 6个二维数据点{(2,3)，(5,4)，(9,6)，(4,7)，(8,1)，(7,2)}构建kd树的具体步骤为：

1. 确定：split域=x。具体是：6个数据点在x，y维度上的数据方差分别为39，28.63，所以在x轴上方差更大，故split域值为x；
2. 确定：Node-data = （7,2）。具体是：根据x维上的值将数据排序，6个数据的中值(所谓中值，即中间大小的值)为7，所以Node-data域位数据点（7,2）。这样，该节点的分割超平面就是通过（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。具体是：分割超平面x=7将整个空间分为两部分：x<=7的部分为左子空间，包含3个节点={(2,3),(5,4),(4,7)}；另一部分为右子空间，包含2个节点={(9,6)，(8,1)}；

每个非叶节点可以想象为一个分割超平面，用垂直于坐标轴的超平面将空间分为两个部分，这样递归的从根节点不停的划分，直到没有实例为止。经典的构造k-d tree的规则如下：

1. 随着树的深度增加，循环的选取坐标轴，作为分割超平面的法向量。对于3-d tree来说，根节点选取x轴，根节点的孩子选取y轴，根节点的孙子选取z轴，根节点的曾孙子选取x轴，这样循环下去。
2. 每次均为所有对应实例的中位数的实例作为切分点，切分点作为父节点，左右两侧为划分的作为左右两子树。

对于n个实例的k维数据来说，建立kd-tree的时间复杂度为O(k*n*logn)。

2.4 kd树的插入

元素插入到一个K-D树的方法和二叉检索树类似。本质上，在偶数层比较x坐标值，而在奇数层比较y坐标值。当我们到达了树的底部，（也就是当一个空指针出现），我们也就找到了结点将要插入的位置。生成的K-D树的形状依赖于结点插入时的顺序。给定N个点，其中一个结点插入和检索的平均代价是O(log2N)。

2.5 kd树的最近邻搜索算法

算法 3.3（用 *kd* 树的最近邻搜索）

输入：已构造的 *kd* 树；目标点 x；

输出：x 的最近邻.

（1）在 *kd* 树中找出包含目标点 x 的叶结点：从根结点出发，递归地向下访问 *kd* 树. 若目标点 x 当前维的坐标小于切分点的坐标，则移动到左子结点，否则移动到右子结点. 直到子结点为叶结点为止.

（2）以此叶结点为“当前最近点”.

（3）递归地向上回退，在每个结点进行以下操作：

（a）如果该结点保存的实例点比当前最近点距离目标点更近，则以该实例点为“当前最近点”.

（b）当前最近点一定存在于该结点一个子结点对应的区域. 检查该子结点的父结点的另一子结点对应的区域是否有更近的点. 具体地，检查另一子结点对应

的区域是否与以目标点为球心、以目标点与“当前最近点”间的距离为半径的超球体相交.

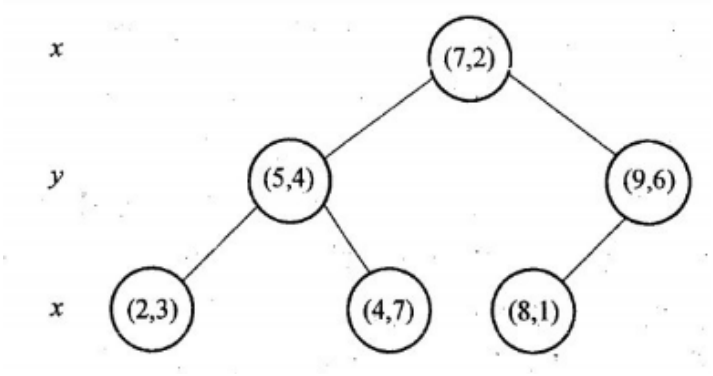
如果相交，可能在另一个子结点对应的区域内存在距目标点更近的点，移动到另一个子结点．接着，递归地进行最近邻搜索；

如果不相交，向上回退.

（4）当回退到根结点时，搜索结束．最后的“当前最近点”即为 x 的最近邻点．■

如果实例点是随机分布的，kd 树搜索的平均计算复杂度是 $O(\log N)$，这里 N 是训练实例数．kd 树更适用于训练实例数远大于空间维数时的 k 近邻搜索．当空间维数接近训练实例数时，它的效率会迅速下降，几乎接近线性扫描.

举例：

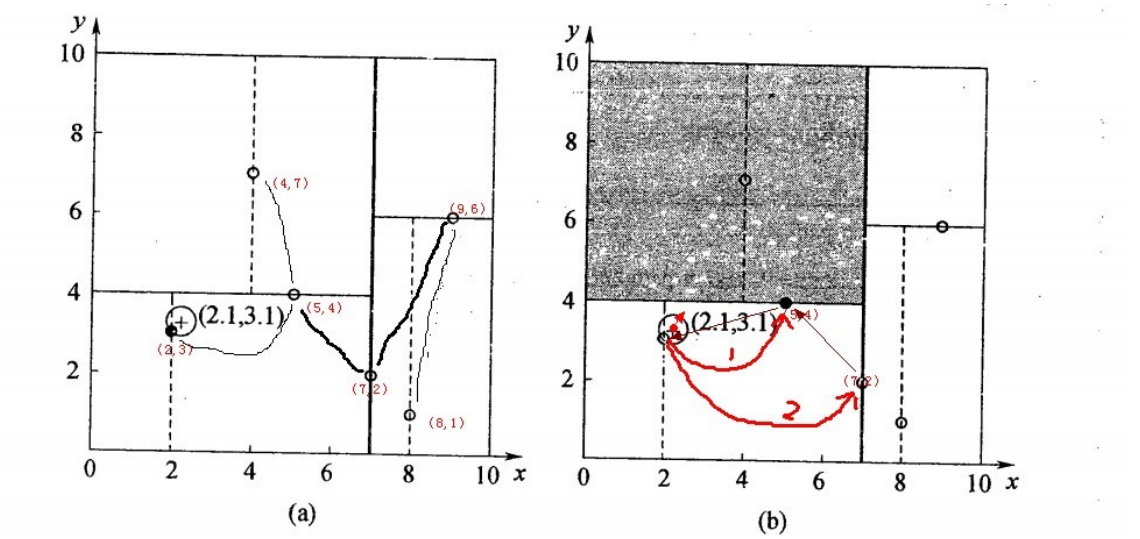


查找点（2. 1, 3. 1）

1. 二叉树搜索：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，

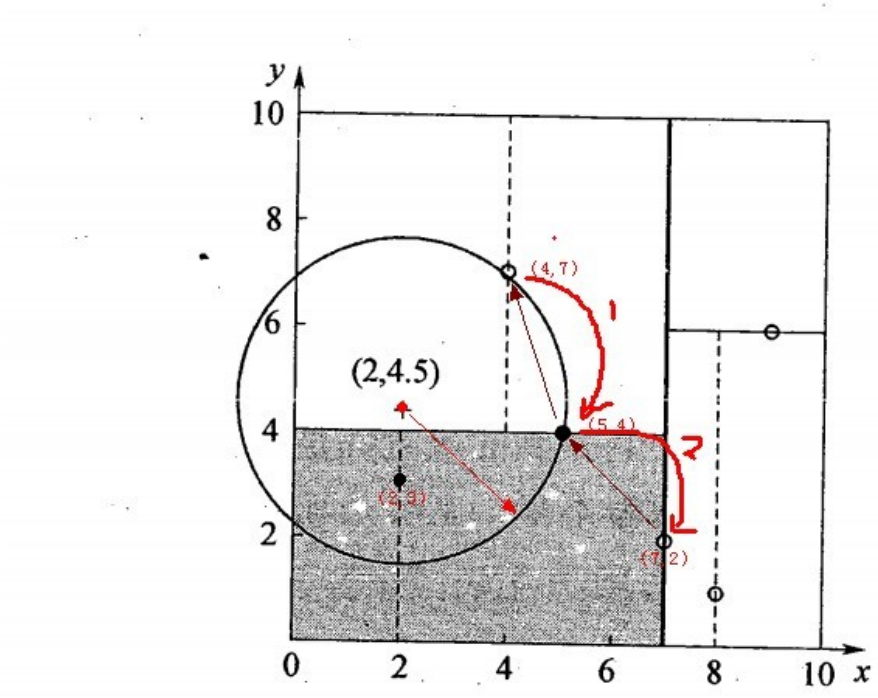
2. 回溯查找：在得到（2,3）为查询点的最近点之后，回溯到其父节点（5,4），并判断在该父节点的其他子节点空间中是否有距离查询点更近的数据点。以（2.1,3.1）为圆心，以0.1414为半径画圆，如下图所示。发现该圆并不和超平面y = 4交割，因此不用进入（5,4）节点右子空间中(图中灰色区域)去搜索；

3. 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。



查找点（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)>，但（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；
2. 以（2，4.5）为圆心，以3.041为半径作圆，如下图所示。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，也就是将（2,3）节点加入搜索路径中得<(7,2)，(2,3)>；于是接着搜索至（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；
3. 回溯查找至（5,4），直到最后回溯到根结点（7,2）的时候，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，如下图所示。至此，搜索路径回溯完，返回最近邻点（2,3），最近距离1.5。



2.6 kd树紧邻搜索算法的改进：BBF算法

如果实例点是随机分布的，那么kd树搜索的平均计算复杂度是O（logN），这里的N是训练实例树。所以说，kd树**更适用于训练实例数远大于空间维数**时的k近邻搜索，当空间维数接近训练实例数时，它的效率会迅速下降，一降降到“解放前”：线性扫描的速度。

也正因为上述k最近邻搜索算法的第4个步骤中的所述：**“回退到根结点时，搜索结束”**，每个最近邻点的查询比较完成过程最终都要回退到根结点而结束，而导致了许多不必要回溯访问和比较到的结点，这些多余的损耗在高维度数据查找的时候，搜索效率将变得相当之地下，那有什么办法可以改进这个原始的kd树最近邻搜索算法呢？

从上述标准的kd树查询过程可以看出其搜索过程中的“回溯”是由**“查询路径”**决定的，并没有考虑查询路径上一些数据点本身的一些性质。一个简单的改进思路就是**将“查询路径”上的结点进行排序**，如按各自分割超平面（也称bin）与查询点的距离排序，也就是说，回溯检查总是从优先级最高（Best Bin）的树结点开始。

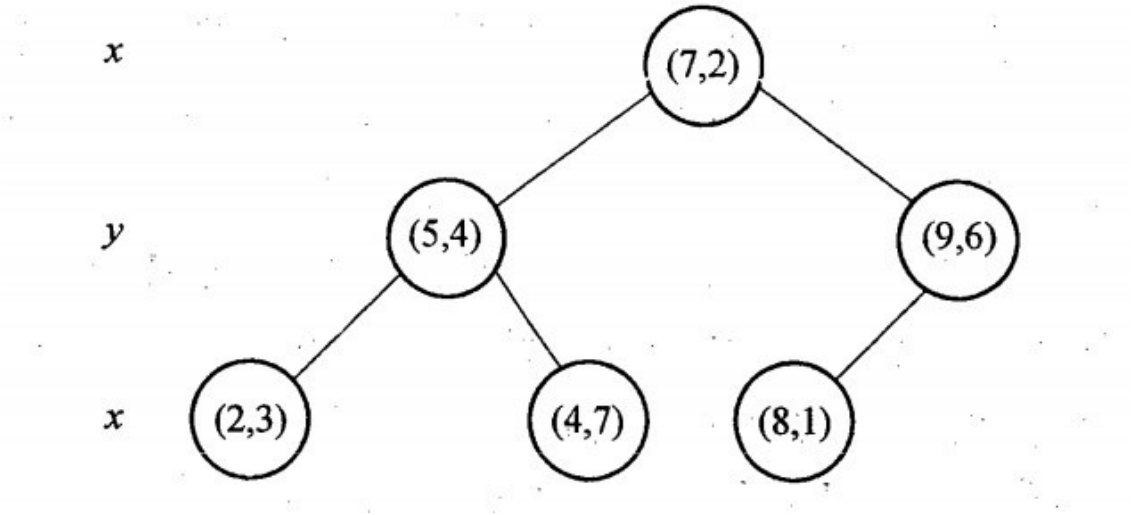
BBF（Best-Bin-First）查询算法，它是由发明sift算法的David Lowe在1997的一篇文章中针对**高维数据**提出的一种近似算法，此算法能确保**优先检索包含最近邻点可能性较高的空间**，此外，BBF机制还设置了一个运行超时限定。采用了BBF查询机制后，kd树便可以有效的扩展到高维数据集上。

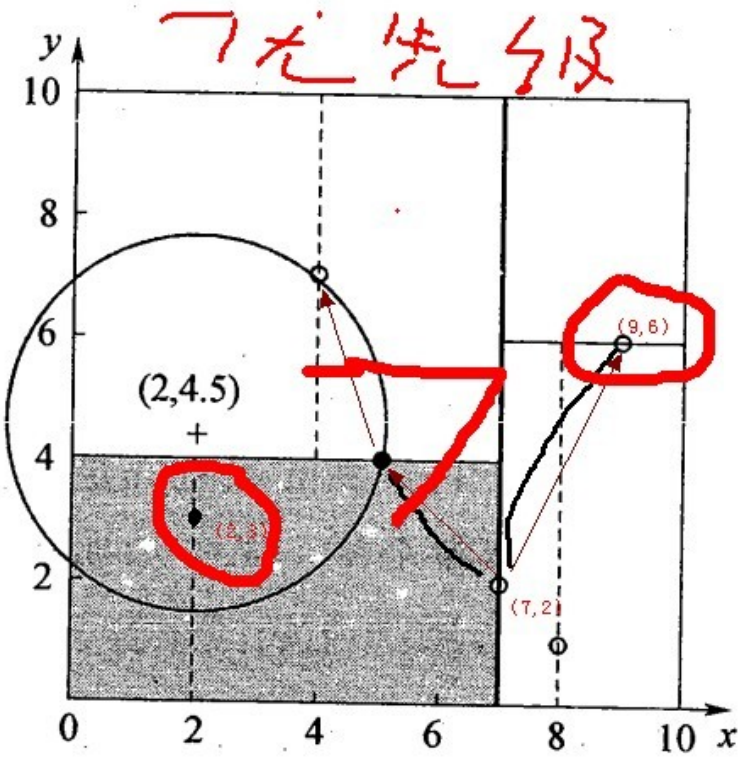
举例：查询（2,4.5)

1. 将（7,2）压人优先队列中;

2. 提取优先队列中的（7,2），由于（2,4.5）位于（7,2）分割超平面的左侧，所以检索其左子结点（5,4）。同时，根据BBF机制”搜索左/右子树，就把对应这一层的兄弟结点即右/左结点存进队列”，将其（5,4）对应的兄弟结点即右子结点（9,6）压人优先队列中，此时优先队列为{（9,6）}，最佳点为（7,2）；然后一直检索到叶子结点（4,7），此时优先队列为{（2,3），（9,6）}，“最佳点”则为（5,4）；

3. 提取优先级最高的结点（2,3），重复步骤2，直到优先队列为空。





3. kd树的应用：SIFT+KD_BBF搜索算法

https://blog.csdn.net/v_july_v/article/details/8203674