**推荐系统召回策略之基于kdtree的近邻快速查找算法**

姓名：岳宇轩 学号：19020011038 指导教师：王欣捷

1. **引言**

**1.1推荐系统简介**

推荐系统属于资讯过滤的一种应用。推荐系统能够将可能受喜好的资讯或实物（例如：电影、电视节目、音乐、书籍、新闻、图片、网页）推荐给使用者。比如抖音、快手等短视频平台会不停地推荐给你相关的视频，淘宝、京东会在你买完一样商品之后给你推荐类似的商品。

**1.2推荐流程简介**

一般推荐流程为：召回→粗排→精排→重排。

召回：目标是在海量的候选中找到用户可能感兴趣的内容，召回一般由多路组成，每一路会有不同的侧重点（优化目标）。

粗排：召回得到的候选集仍然很多，不能直接上复杂模型，需要再筛一轮，减少候选集数量。

精排：使用复杂模型，进行精确的预测。学术界的工作大多是在这一部分。

重排：添加人为要求，比如根据产品经理的要求做更改。

**1.3 FM召回**

我们举一个具体的例子，比如我点开一个b站的视频

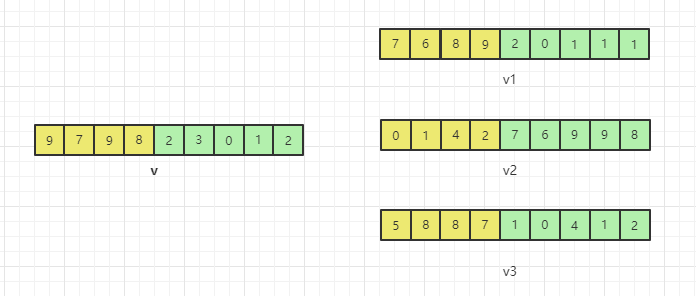


点开之后，他会给我推荐下面一些相关的视频：



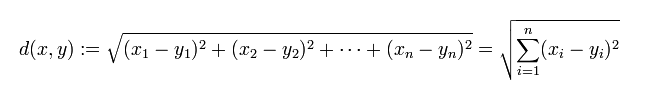
我把推荐视频中的一些信息用红框框出来了。不难发现，红框中的内容都与本视频有一定的关联性。比如第一个视频的封面和本视频的封面相似度很高，第三个视频的标题含有“国王排名”（与本视频标题关键字一致），第四个视频标题含有“烂”（与本视频标题关键字一致），第二个是bilibili番剧，很明显就是在重排阶段加入的。

如果，我们把所有视频以向量的形式示：



其中，v表示当前观看的视频，v1,v2,v3表示在侯选库中的三个视频，用标黄维度的数据表示标题信息，用标绿维度的数据表示封面的信息。那么，该如何从v1,v2,v3中选出应该推荐的视频呢？

显然，和本视频相似度越高的视频越应该被推荐。如此，则需要计算和v距离最近的向量所表示的视频，就是应该推荐的视频了。这里计算向量间距离采用欧氏距离d(x,y)：

其中，x,y分别表示待计算距离的两个向量，xi,yi表示其对应维度的数据，最终结果用d(x,y)表示。

下面我们分别计算d(v, v1), d(v, v2), d(v, v3):

|  |  |
| --- | --- |
| 向量 | 距离 |
| v1 | 4.24 |
| v2 | 19.82 |
| v3 | 5.83 |

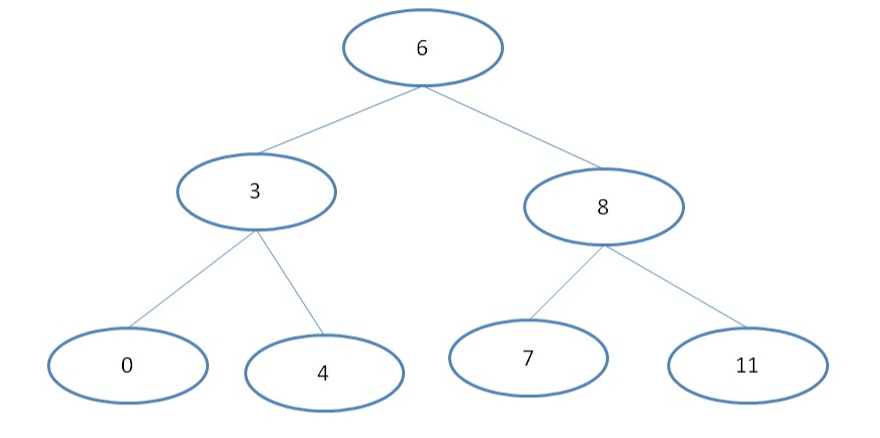
根据计算结果，d(v, v1) ＜ d(v, v3) ＜ d(v, v2)，所以最应该推荐v1对应的视频，其次v3，最后v2。

但是，该方式时间复杂度O(n)，在数据规模较大时，时间开销严重无法接受。由此引出了基于kdtree的近邻快速查找算法。

**二、基于kdtree的近邻快速查找算法**

**2.1 BST简单回顾**

在介绍kdtree之前，我们先来简单回顾一下二分查找树BST，有助于我们更好的理解kdtree。假设数组A为[0, 6, 3, 8, 7, 4, 11]，有一个元素x，我们要找到数组A中x元素，则可以把数组A建立成一个BST，结构如下图所示。我们只需要访问根节点，进行值比较来确定下一节点，如此循环往复直到访问到叶子节点为止。



比如，我要查找和7，过程如下：

·与6比，往右走

·与8比，往左走

·得到7

现在我们把问题加点难度，假设数组B为[[6, 2], [6, 3], [3, 5], [5, 0], [1, 2], [4, 9], [8, 1]]，有一个元素x，我们要找到数组B中距离x最近的元素，应该如何实现呢？答案就是kdtree啦。

**2.2 kdtree的建立**

1. 建立根节点。

2. 选取方差最大的特征作为分割特征；

3. 选择该特征的中位数作为分割点；

4. 将数据集中该特征小于中位数的传递给根节点的左儿子，大于中位数的传递给根节点的右儿子；

5. 递归执行步骤2-4，直到所有数据都被建立到KD Tree的节点上为止。

解释：

·为什么方差最大的适合作为特征呢？ 因为方差大，数据相对“分散”，选取该特征来对数据集进行分割，数据散得更“开”一些。

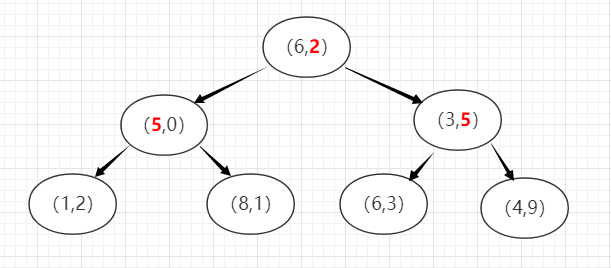
·为什么选择中位数作为分割点呢？ 因为借鉴了BST，选取中位数，让左子树和右子树的数据数量一致，便于二分查找。

具体过程如下：

①首先，我们可以根据B数组得到两个维度特征的序列：6,6,3,5,1,4,8和2,3,5,0,2,9,1.分别计算这两个维度的方差为5.23和9.14，第二个维度特征序列方差更大，取第二个序列中位数为2，有两个元素（6,2）和（1,2）的第二维都是2，任选一个作为根节点，这里选（6,2）。

②将B数组中的其他元素和（6,2）进行比较，只比较第二维的元素，小的放左边，大的放右边，可以的到左边的元素为：（5,0）、（1,2）、（8,1），右边的元素为（6,3）、（3,5）、（4,9）。

③然后在左边和右边分别重复①中的动作，即可得到如下的kdtree：



**2.3利用kdtree查找元素**

KD Tree建好之后，接下来就要利用KD Tree对元素进行查找了。查找的方式在BST的基础上又增加了一些难度，如下：

1. 从根节点开始，根据目标在分割特征中是否小于或大于当前节点，向左或向右移动。

2. 一旦算法到达叶节点，它就将节点点保存为“当前最佳”。

3. 回溯，即从叶节点再返回到根节点

4. 如果当前节点比当前最佳节点更接近，那么它就成为当前最好的。

5. 如果目标距离当前节点的父节点所在的将数据集分割为两份的**超平面**的距离更接近，说明当前节点的兄弟节点所在的子树有可能包含更近的点。因此需要对这个兄弟节点递归执行1-4步。

解释：什么是超平面？以[0, 2, 0], [1, 4, 3], [2, 6, 1]的举例：

1.用第二维特征作为分割特征，那么从三个数据点中的对应特征取出2, 4, 6，中位数是4；

2. 所以[1, 4, 3]作为分割点，将[0, 2, 0]划分到左边，[2, 6, 1]划分到右边；

3. 从立体几何的角度考虑，三维空间得用一个二维的平面才能把空间一分为二，这个平面可以用y = 4来表示；

4. 点[0, 2, 0]到超平面y = 4的距离就是 sqrt((2 - 4) ^ 2) = 2；

5. 点[2, 6, 1]到超平面y = 4的距离就是 sqrt((6 - 4) ^ 2) = 2。

比如我们要寻找和（5,4）距离最近的向量，具体过程如下：

①与根节点（6,2）比对第二维元素，4>2，走右子树

②与节点（3,5）比对右子树第二维元素，4<5，走左子树

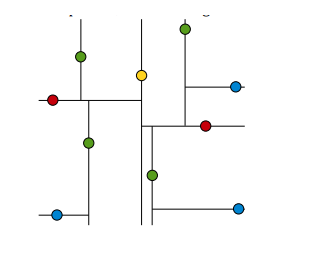
③找到叶子结点（6,3），保存（6,3）到（5,4）距离1.41为当前最佳

④回溯至（3,5），比对（3,5）到（5,4）的距离2.2大于当前最佳，不更新

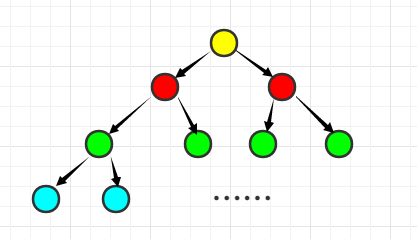
⑤计算目标（5,4）距离当前节点（3，5）的父节点（6,2）所在的将数据集分割为两份的超平面（y=2）的距离sqrt((4 - 2) ^ 2) = 2＞当前最佳，因此不对当前节点（3,5）的兄弟节点（5,0）进行递归检查。

⑥回溯至根节点（6,2），计算距离大于当前最佳，不更新。查找结束。

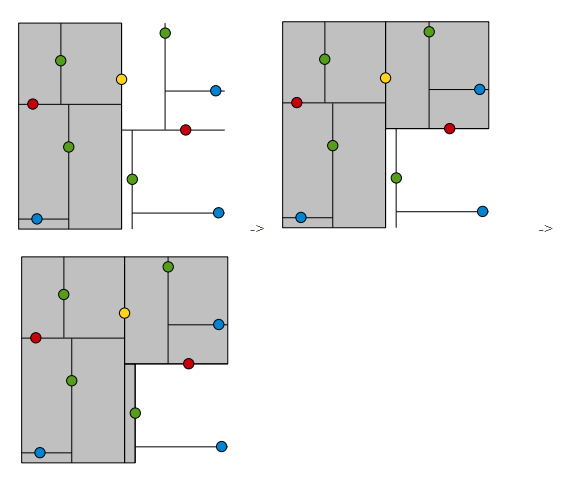
**2.4形象化理解kdtree搜索**



在上图中，黄色表示根节点，红色是黄色节点的子节点，绿色是红色节点的子节点，蓝色是绿色节点的子节点。树形结构如下图所示：



搜索过程可以参考下图理解：



首先，将查询元素和黄色节点比较，如发现在其右侧，则再跟右侧的红色节点进行比较，如发现在下侧，则再跟下侧的绿色节点比较，如发现在右侧，则再跟右侧的蓝色节点比较.....

实质上，KDTree就是超平面都垂直于轴的BSPTree。

**二、算法的应用和具体实施方式**

**2.1 物品的向量表示**

kdtree是进行向量的查找，所以需要先把带推荐的所有物品用向量形式表示。物品的特征学习是推荐系统中研究的重点问题，下面介绍两种比较简单常见的编码方式：

**2.1.1 one-hot编码**

直接举例说明：

运动特征：["足球"，"篮球"，"羽毛球"，"乒乓球"]：  
足球 → 1000  
篮球 → 0100  
羽毛球 → 0010  
乒乓球 → 0001

缺点：如果特征有多种取值情况会导致维度灾难。

**2.1.2 multi-hot编码**

运动特征：["足球"，"篮球"，"羽毛球"，"乒乓球"]：  
足球 → 0  
篮球 → 1  
羽毛球 → 2  
乒乓球 → 3

**2.1.3 Index编码**

年龄：[3,20,50,100]

3→0

20→1

50→2

100→3

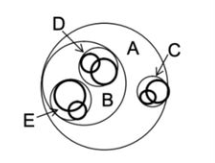
**2.2具体实施**

有了物品的向量表示后，便可直接套用kdtree的近邻快速查找算法了。

1. **总结与展望**

相比于顺序查找，kdtree的算法复杂度介于O(Log2(N))和O(N)之间，可以说是在牺牲精准度的代价下换取了查找效率。kd树在维数小于20时效率最高，一般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，几乎接近线形扫描。

为了解决高维情况下效率下降的问题，ball tree被提出。简单解释一下，就是利用三角不等式，比如我们想找q点的近邻点。假设q的半径r1，f的半径r2，q-f的距离l，如果r1+r2<l则两圆不相交，f圆内不可能存在近邻点，反之则存在并在f圆内继续搜索。即，ball tree使用超球面而不是超矩形划分区域：



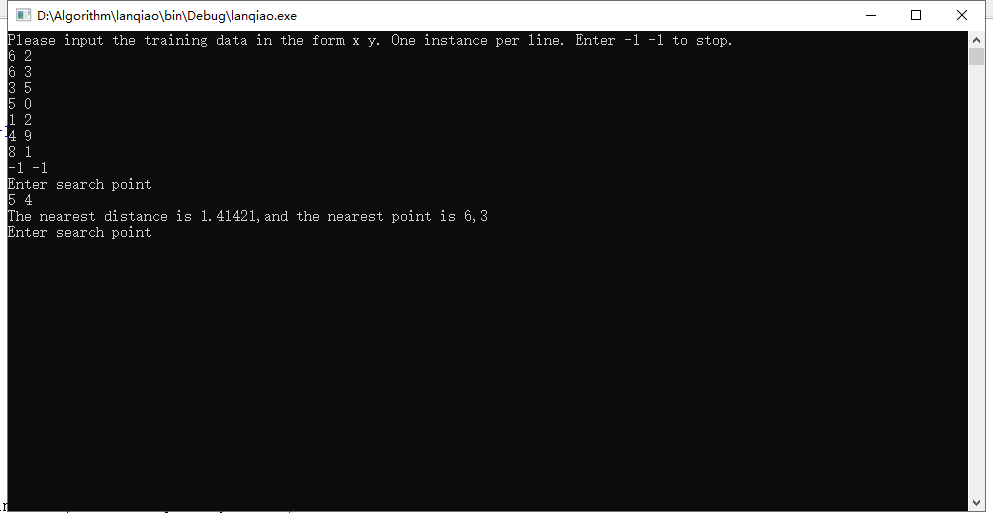
总结自己对于推荐系统的学习，发现大多都在关注精排阶段的模型，最为关注的指标乃是准确性；而在从庞大规模的数据库中进行第一轮筛选的召回阶段，其关注点除了准确性还有效率。虽然kd-tree的快速查找算法效果是最差的（HNSW>NSW>Annoy>Ball Tree>KD Tree），但它的结构是其它的基础，其舍弃精度换区效率的思想也与召回阶段的初衷一致。

展望未来，我在想：能否直接在庞大的数据集上动用复杂模型，从而省去召回阶段？如果召回阶段是必需的，那么又有没有一种方法，它既可以用于召回阶段，又可以用于粗排和精排阶段呢？那样岂不是非常优雅简洁。

1. **kdtree实现的c++代码**

以下是k-d树的c++代码实现，包括建树过程和搜索过程。算法main函数输入k-d树训练实例点，算法会完成建树操作，随后可以输入待查询的目标点，程序将会搜索K-d树找出与输入目标点最近邻的训练实例点。本程序只实现了1近邻搜索。

代码结果展示：（与2.3中例子相对应）



代码：

#include <iostream>

#include <algorithm>

#include <stack>

#include <math.h>

using namespace std;

/\*function of this program: build a 2d tree using the input training data

the input is exm\_set which contains a list of tuples (x,y)

the output is a 2d tree pointer\*/

struct data

{

double x = 0;

double y = 0;

};

struct Tnode

{

struct data dom\_elt;

int split;

struct Tnode \* left;

struct Tnode \* right;

};

bool cmp1(data a, data b){

return a.x < b.x;

}

bool cmp2(data a, data b){

return a.y < b.y;

}

bool equal(data a, data b){

if (a.x == b.x && a.y == b.y)

{

return true;

}

else{

return false;

}

}

void ChooseSplit(data exm\_set[], int size, int &split, data &SplitChoice){

/\*compute the variance on every dimension. Set split as the dismension that have the biggest

variance. Then choose the instance which is the median on this split dimension.\*/

/\*compute variance on the x,y dimension. DX=EX^2-(EX)^2\*/

double tmp1,tmp2;

tmp1 = tmp2 = 0;

for (int i = 0; i < size; ++i)

{

tmp1 += 1.0 / (double)size \* exm\_set[i].x \* exm\_set[i].x;

tmp2 += 1.0 / (double)size \* exm\_set[i].x;

}

double v1 = tmp1 - tmp2 \* tmp2; //compute variance on the x dimension

tmp1 = tmp2 = 0;

for (int i = 0; i < size; ++i)

{

tmp1 += 1.0 / (double)size \* exm\_set[i].y \* exm\_set[i].y;

tmp2 += 1.0 / (double)size \* exm\_set[i].y;

}

double v2 = tmp1 - tmp2 \* tmp2; //compute variance on the y dimension

split = v1 > v2 ? 0:1; //set the split dimension

if (split == 0)

{

sort(exm\_set,exm\_set + size, cmp1);

}

else{

sort(exm\_set,exm\_set + size, cmp2);

}

//set the split point value

SplitChoice.x = exm\_set[size / 2].x;

SplitChoice.y = exm\_set[size / 2].y;

}

Tnode\* build\_kdtree(data exm\_set[], int size, Tnode\* T){

//call function ChooseSplit to choose the split dimension and split point

if (size == 0){

return NULL;

}

else{

int split;

data dom\_elt;

ChooseSplit(exm\_set, size, split, dom\_elt);

data exm\_set\_right [100];

data exm\_set\_left [100];

int sizeleft ,sizeright;

sizeleft = sizeright = 0;

if (split == 0)

{

for (int i = 0; i < size; ++i)

{

if (!equal(exm\_set[i],dom\_elt) && exm\_set[i].x <= dom\_elt.x)

{

exm\_set\_left[sizeleft].x = exm\_set[i].x;

exm\_set\_left[sizeleft].y = exm\_set[i].y;

sizeleft++;

}

else if (!equal(exm\_set[i],dom\_elt) && exm\_set[i].x > dom\_elt.x)

{

exm\_set\_right[sizeright].x = exm\_set[i].x;

exm\_set\_right[sizeright].y = exm\_set[i].y;

sizeright++;

}

}

}

else{

for (int i = 0; i < size; ++i)

{

if (!equal(exm\_set[i],dom\_elt) && exm\_set[i].y <= dom\_elt.y)

{

exm\_set\_left[sizeleft].x = exm\_set[i].x;

exm\_set\_left[sizeleft].y = exm\_set[i].y;

sizeleft++;

}

else if (!equal(exm\_set[i],dom\_elt) && exm\_set[i].y > dom\_elt.y)

{

exm\_set\_right[sizeright].x = exm\_set[i].x;

exm\_set\_right[sizeright].y = exm\_set[i].y;

sizeright++;

}

}

}

T = new Tnode;

T->dom\_elt.x = dom\_elt.x;

T->dom\_elt.y = dom\_elt.y;

T->split = split;

T->left = build\_kdtree(exm\_set\_left, sizeleft, T->left);

T->right = build\_kdtree(exm\_set\_right, sizeright, T->right);

return T;

}

}

double Distance(data a, data b){

double tmp = (a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y);

return sqrt(tmp);

}

void searchNearest(Tnode \* Kd, data target, data &nearestpoint, double & distance){

//1. 如果Kd是空的，则设dist为无穷大返回

//2. 向下搜索直到叶子结点

stack<Tnode\*> search\_path;

Tnode\* pSearch = Kd;

data nearest;

double dist;

while(pSearch != NULL)

{

//pSearch加入到search\_path中;

search\_path.push(pSearch);

if (pSearch->split == 0)

{

if(target.x <= pSearch->dom\_elt.x) /\* 如果小于就进入左子树 \*/

{

pSearch = pSearch->left;

}

else

{

pSearch = pSearch->right;

}

}

else{

if(target.y <= pSearch->dom\_elt.y) /\* 如果小于就进入左子树 \*/

{

pSearch = pSearch->left;

}

else

{

pSearch = pSearch->right;

}

}

}

//取出search\_path最后一个赋给nearest

nearest.x = search\_path.top()->dom\_elt.x;

nearest.y = search\_path.top()->dom\_elt.y;

search\_path.pop();

dist = Distance(nearest, target);

//3. 回溯搜索路径

Tnode\* pBack;

while(search\_path.size() != 0)

{

//取出search\_path最后一个结点赋给pBack

pBack = search\_path.top();

search\_path.pop();

if(pBack->left == NULL && pBack->right == NULL) /\* 如果pBack为叶子结点 \*/

{

if( Distance(nearest, target) > Distance(pBack->dom\_elt, target) )

{

nearest = pBack->dom\_elt;

dist = Distance(pBack->dom\_elt, target);

}

}

else

{

int s = pBack->split;

if (s == 0)

{

if( fabs(pBack->dom\_elt.x - target.x) < dist) /\* 如果以target为中心的圆（球或超球），半径为dist的圆与分割超平面相交， 那么就要跳到另一边的子空间去搜索 \*/

{

if( Distance(nearest, target) > Distance(pBack->dom\_elt, target) )

{

nearest = pBack->dom\_elt;

dist = Distance(pBack->dom\_elt, target);

}

if(target.x <= pBack->dom\_elt.x) /\* 如果target位于pBack的左子空间，那么就要跳到右子空间去搜索 \*/

pSearch = pBack->right;

else

pSearch = pBack->left; /\* 如果target位于pBack的右子空间，那么就要跳到左子空间去搜索 \*/

if(pSearch != NULL)

//pSearch加入到search\_path中

search\_path.push(pSearch);

}

}

else {

if( fabs(pBack->dom\_elt.y - target.y) < dist) /\* 如果以target为中心的圆（球或超球），半径为dist的圆与分割超平面相交， 那么就要跳到另一边的子空间去搜索 \*/

{

if( Distance(nearest, target) > Distance(pBack->dom\_elt, target) )

{

nearest = pBack->dom\_elt;

dist = Distance(pBack->dom\_elt, target);

}

if(target.y <= pBack->dom\_elt.y) /\* 如果target位于pBack的左子空间，那么就要跳到右子空间去搜索 \*/

pSearch = pBack->right;

else

pSearch = pBack->left; /\* 如果target位于pBack的右子空间，那么就要跳到左子空间去搜索 \*/

if(pSearch != NULL)

// pSearch加入到search\_path中

search\_path.push(pSearch);

}

}

}

}

nearestpoint.x = nearest.x;

nearestpoint.y = nearest.y;

distance = dist;

}

int main(){

data exm\_set[100]; //assume the max training set size is 100

double x,y;

int id = 0;

cout<<"Please input the training data in the form x y. One instance per line. Enter -1 -1 to stop."<<endl;

while (cin>>x>>y){

if (x == -1)

{

break;

}

else{

exm\_set[id].x = x;

exm\_set[id].y = y;

id++;

}

}

struct Tnode \* root = NULL;

root = build\_kdtree(exm\_set, id, root);

data nearestpoint;

double distance;

data target;

cout <<"Enter search point"<<endl;

while (cin>>target.x>>target.y)

{

searchNearest(root, target, nearestpoint, distance);

cout<<"The nearest distance is "<<distance<<",and the nearest point is "<<nearestpoint.x<<","<<nearestpoint.y<<endl;

cout <<"Enter search point"<<endl;

}

}