KNN 代码说明

作者：蔡中恒
2018.06.09

一、原理讲解

原理涉及两个部分：

1、 kd 树的生成。该部分原理详见
https://www.cnblogs.com/aTianTianTianLan/articles/3902963.html

2、 KNN 具体原理，该部分详见李航《统计学习方法》第 3 章 k 近邻法。

二、算法思路

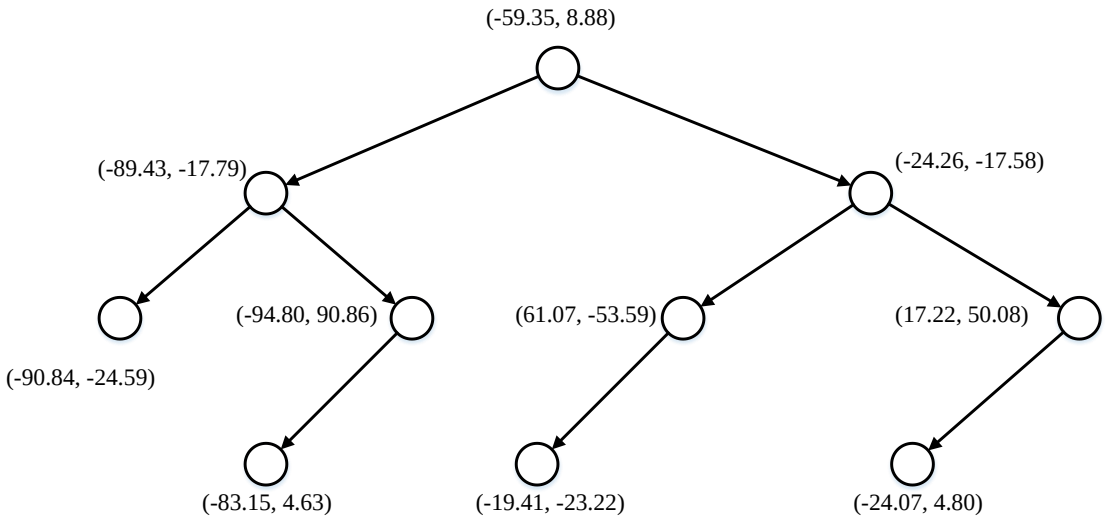
本次使用 matlab 来写 KNN 的算法原型代码。注意，虽然 matlab 有自带的 kd_tree 的函数，但这里为了让自己搞懂，所以决定自己写一个 kd tree 生成和 KNN 查找的代码。

1. kd_tree 生成算法思路

假设数据有很多维度，kd tree 的生成方法就是对每一组数据选择合适的维度，不断使用二分法，最终生成一棵树。这里，生成规则如下：

1) 对初始数据的所有维度求方差，找出方差最大的维度 dim_idx。
2) 根据方差最大的维度，将数据从小到大排序，将排序后数据的中位数设置为该节点（root.data）的值，本组数据中小于 root.data 的数据归类到左子树数据，大于 root.data 的数据归类到右子树数据。
3) 分别对左子树数据和右子树数据重复递归 1)和 2)，直到每组数据只剩 2 个或者 1 个。如果该组数据只剩 1 个，则直接赋值为该节点的 root.data，如果数据只剩 2 个，则将较小的数据归类为该节点左子树的 root.data，较大的数据直接赋值为该节点的 root.data。

这里画出一个二维数据 kd tree 的示意图。



2. KNN 搜索算法思路

在生成 kd tree 之后，对于新加入的数据（记为 A），我们需要搜索其最近的 K 个节点的值，此时需要设计 KNN 的搜索程序。算法思想如下：

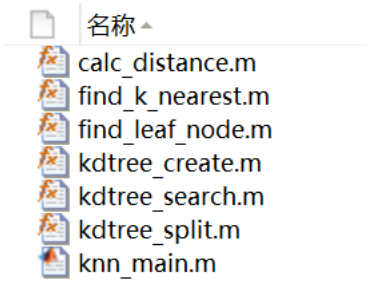
1、 先通过 kd tree 的结构，找到离 A 点最近的叶节点，标记为**初始叶节点**。并记录找到叶节点的路径过程。
   a) 对每一个节点，都判断有无左子树和右子树。
      i. 如果左右子树都没有，说明该点就是叶节点，直接返回。
      ii. 如果只存在左子树，则直接去左子树。
      iii. 如果只存在右子树，则直接去右子树。
      iv. 如果左右子树都存在，则将 A 和该节点在该节点的划分维度上进行比较，如果 A 小于等于该节点，则去左子树；否则去右子树，递归进行。
2、 接下来寻找最小的 K 个节点。从上一步找到的叶节点开始，进行路径回溯。
   a) 如果当前队列长度不满 K 个点。
      i. 将当前节点存入队列中，计算当前节点和 A 的距离；并重新按照距离，对队列中的数据进行升序排列。
   b) 如果当前队列长度已经满足 K 个点。
      i. 找到 K 个点中距离的最大的点，设该点与 A 的距离为 d_max，将当前节点与 A 的距离 d_now 和 d_max 做比较。
      ii. 如果 d_now<=d_max，则在队列中存入该节点，重新按照升序排序后，队列只取前 k 个点。
   c) 判断该节点是否还有父节点。
      i. 如果没有，说明该节点已经是根节点了。则程序终止并返回。如果还有父节点，则进行后面的步骤（判断的方法是根据前面 kd tree 的记录路径来做判断）。
      ii. 如果还存在父节点，程序继续往下执行。
   d) 判断该节点有无兄弟节点。
      i. 如果该节点没有兄弟节点，那就后退到父节点，将该节点从 kd tree 上去掉，形成新的 kd tree 和新的初始叶节点，递归调用函数，重复第 2 步。

ii. 如果该节点有兄弟节点。

1) 进行判断，将当前队列中最大的 d_max 和父节点到 A 在划分维度上的距离做比较，如果 d_max 是较小值且队列已经塞满 k 个数据，那就说明兄弟节点里面的所有数据都不属于 k 队列中，就无须搜索兄弟节点，直接回退父节点即可。否则，就要去兄弟节点搜索，将该节点从 kd tree 上去掉，形成新的 kd tree 和新的初始叶节点，递归调用函数，重复第 2 步。

三、代码讲解

代码采用 matlab 写成。



接下来依次讲解各脚本和函数的功能。

首先是 knn_main.m 代码，这是运行的主函数，包含参数设置、数据生成、图像绘制等内容。在代码最后会调用 kdtree_search 函数进行 knn 搜索，完成后将输出结果绘制成二维图像。

```
44
45 —    k_array      = kdtree_search(test_data, data_kdtree, k_num);
46
47      %% plot figure
48 —    plot(complex(k_array(:,1),k_array(:,2)),'square');
49 —    viscircles([test_data_x1,test_data_x2], k_array(end,end));
50 —    hold off;
```

对于 kdtree 的生成，这里调用 kdtree_create 函数，算法原理在上一节已经有阐述，对于 matlab 代码，这里使用结构体和递归函数来实现类似于二叉树的架构。

```
function root = kdtree_create(input_data)
%% 找出方差最大的维度
data_var                = mean(input_data.^2,1) - mean(input_data,1).*mean(input_data,1);%求每个维度的方差
[~,dim_idx]             = max(data_var);

%% 根据dim_dix所处的维度，对数据进行排序，并一分为二
tmp_data                = input_data(:,dim_idx);
[tmp_data,resort_idx]   = sort(tmp_data);
resort_data             = input_data(resort_idx,:);
data_len                = length(tmp_data);
root.dim                = dim_idx;% 将当前比较的维度编号存储下来
%% 将当前父节点赋值到root.data里面,并分为左子树和右子树，递归调用
if(data_len==1)
    root.data        = resort_data(1,:);
elseif(data_len==2)
    root.left.data   = resort_data(1,:);
    root.left.dim    = 1;
    root.data        = resort_data(2,:);
else
    mid_idx          = ceil(data_len/2);% 取中位数
    root.data        = resort_data(mid_idx,:);
    root.left        = kdtree_create(resort_data(1:mid_idx-1,:));%递归调用，生成左子树
    root.right       = kdtree_create(resort_data(mid_idx+1:end,:));%递归调用，生成右子树
end
end
```

接下来讲解 kdtree_search 函数。该函数主要作用是对 k_array 进行初始化，调用 find_leaf_node 寻找初始叶节点，然后调用 find_k_nearest 函数进行 knn 搜索。

```
function k_array = kdtree_search(test_data, kdtree, k_num, dist_mode)

if(nargin<4)
    dist_mode = 0;
end

%% 初始化个数为k的最近距离队列
k_array = [];
% k_array = []; %repmat([zeros(1,size(test_data,2)),0],k_num,1); % [train_data, distance]

%% 根据待测试的数据，首先寻找初始叶节点，搜索过程中将路径记录下来
[~,first_leaf_path] = find_leaf_node(test_data, kdtree);

%% 从初始叶节点开始，搜索过程中将路径记录下来
[~, ~, ~, k_array]  = find_k_nearest(test_data, first_leaf_path, kdtree, k_array, k_num, dist_mode);
```

find_leaf_node 函数，将新加入的 A (就是 test_data)，在 kdtree 这个结构体中依次作比较，每次比较都按照当前划分的维度(dim)来做，如果 test_data 小于当前节点在该 dim 上的值，那就去左子树；反之则去右子树。比较的过程中顺便记录下 leaf_path，0 表示这一次去了左子树，1 表示这一次去了右子树。

```
function [leaf_data, leaf_path] = find_leaf_node(test_data, kdtree)

left_flag  = isfield(kdtree,'left');
right_flag = isfield(kdtree,'right');

if(left_flag==0)&&(right_flag==0)
    % 已经是叶节点，直接返回
    leaf_data        = kdtree;
    leaf_path_parent = [];
    leaf_path_child  = [];
%     kdtree_brother   = kdtree_brother;
elseif(left_flag==1)&&(right_flag==0)
    % 只有左子树，那就直接去左子树
    leaf_path_parent               = 0; % 0: left; 1: right
    [leaf_data, leaf_path_child] = find_leaf_node(test_data, kdtree.left);
elseif(left_flag==0)&&(right_flag==1)
    % 只有右子树，那就直接去右子树
    leaf_path_parent             = 1; % 0: left; 1: right
    [leaf_data, leaf_path_child] = find_leaf_node(test_data, kdtree.right);
else
    % 左右子树都有，那就做比较，去最接近的一方
    if(test_data(kdtree.dim)<=kdtree.data(kdtree.dim))
        leaf_path_parent               = 0; % 0: left; 1: right
        [leaf_data, leaf_path_child] = find_leaf_node(test_data, kdtree.left);
    else
        leaf_path_parent               = 1; % 0: left; 1: right
        [leaf_data, leaf_path_child] = find_leaf_node(test_data, kdtree.right);
    end
end
leaf_path = [leaf_path_parent leaf_path_child];

end
```

最后是 find_k_nearest 这个函数，算法的部分在上一节已经说过了，这里只说一下和代码相关的。返回标志用的是 return_flag，一旦检测到 leaf_path 为空矩阵，就将 return_flag 设置为 1；而一旦检测到这个变量为 1，整个函数就直接返回。

```
function [data_cur_max, return_flag, kdtree_cur, k_array] = find_k_nearest(test_data, leaf_path, kdtree_cur, k_array, k_num, dist_mode)

% return_flag = 0;
k_array_flag = 0; %标记k队列是否塞满。

%% 根据当前的kd tree，寻找最近的叶节点
% 这一步使用leaf_path_search函数，目的在于根据指定的路径和kdtree，找到对应的叶节点和该叶节点的兄弟节点组成的树，以及父树
if(isempty(leaf_path)==1)
    root_data   = kdtree_cur.data;
    father_data = [];
else
    [root_data, father_data, tree_brother, tree_father] = kdtree_split(kdtree_cur, leaf_path);
end

%% 对k_array的长度进行判断
dist_cur = calc_distance(test_data, root_data, dist_mode);
if(size(k_array,1)<k_num)
    %% 队列的长度还没到k的时候，就把当前节点的值和距离直接存入队列中
    % 将该节点和对应的距离存入k_array中
    k_array = [k_array; root_data, dist_cur];
    % 对队列进行重新排序，确保队列按照距离的升序排列，最大距离的数值在队列的最后一行
    if(isempty(k_array)~=1)
        [~,sort_idx]  = sort(k_array(:,end));
        k_array       = k_array(sort_idx,:);
```

```
        data_cur_max  = k_array(end,end); % 找到队列中最大的距离
    else
        data_cur_max  = 0;
    end
else
    %% 队列的长度已经塞满，那就将当前节点和队列里的数据进行比较，将当前数据和距离塞进队列
    data_cur_max = k_array(end,end);
    k_array_flag = 1;
    if(data_cur_max>dist_cur)
        idx                     = find(k_array(:,end)>dist_cur);
        if(length(idx)>1)
            k_array(idx(2):end,:) = k_array(idx(1):end-1,:);
        else
        end
        k_array(idx(1),:)       = [root_data, dist_cur];
    else
    end
end

if(isempty(leaf_path)==1)
    return_flag = 1;
    return;
else
end
```

```
%% 判断有无兄弟节点
if(isempty(tree_brother)==1)
    % 没有兄弟节点，则直接回退到父节点，递归调用，此时将该根节点从kdtree上掉落，所以递归调用的时候kd_tree使用tree_father
    leaf_path                   = leaf_path(1:length(leaf_path)-1);
    [~, return_flag, ~, k_array]    = find_k_nearest(test_data, leaf_path, tree_father, k_array, k_num, dist_mode);
    if(return_flag==1)
        return;
    else
    end
else
    % 存在兄弟节点，则做判断，切分面为父节点
    dist_in_dim = abs(test_data(father_data.dim) - father_data.data(father_data.dim));

    if(dist_in_dim>data_cur_max)&&(k_array_flag==1)
        %% 直接回退到父节点
        leaf_path                   = leaf_path(1:length(leaf_path)-1);
        [~, return_flag, ~, k_array]    = find_k_nearest(test_data, leaf_path, tree_father, k_array, k_num, dist_mode);
        if(return_flag==1)
            return;
        else
        end
    else
```

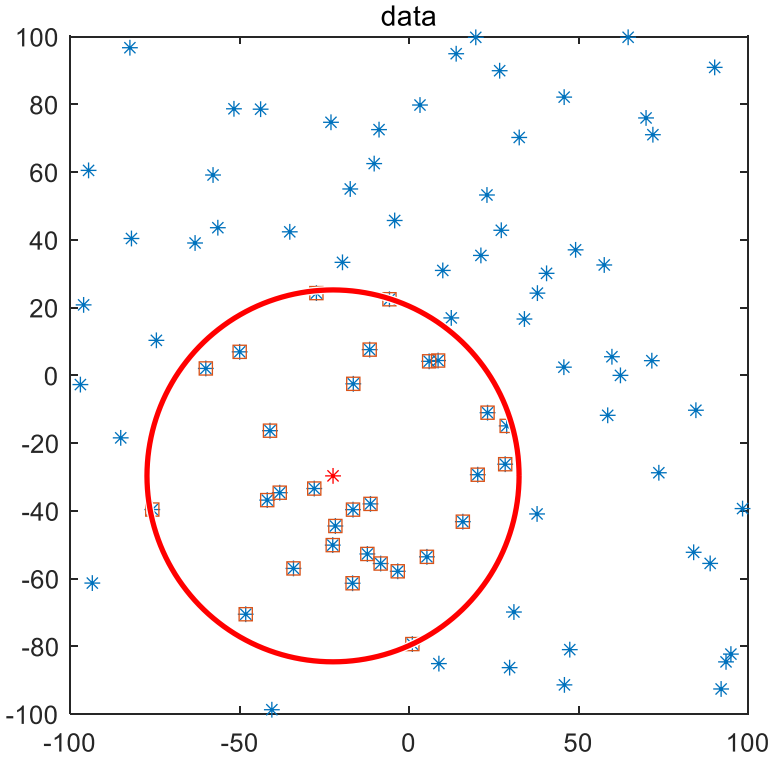
去兄弟节点搜索的时候，需要把父节点的子树和兄弟节点的子树重新拼在一起，组成一棵新的树。这种操作就相当于把已经搜索过的节点从 kdtree 上去掉，然后就可以调用递归函数了。操作中使用了 matlab 的 eval 命令，该命令可以方便地将字符串转化为成可执行的脚本语句，再配合 matlab 结构体的特性，就能写出较为简洁的代码。

```
        %% 去兄弟节点继续搜索

        % 给兄弟节点重新建立新的leaf_path，方便递归调用。
        [~,brother_leaf_path] = find_leaf_node(test_data, tree_brother);
        % 将父树的leaf_path和兄弟节点的leaf_path嫁接在一起，建立新的leaf_path
        new_leaf_path         = [leaf_path(1:end-1) double(~leaf_path(end)) brother_leaf_path];
        % 建立新的kdtree，包含父树和兄弟节点，但是不包含当前已经比较过的节点。
        father_path           = [];
        for idx=1:length(leaf_path)-1
            if(leaf_path(idx)==0)
                father_path = [father_path '.left'];
            else
                father_path = [father_path '.right'];
            end
        end
        tree_new            = tree_father;
        if(leaf_path(end)==1)
            % 兄弟节点是左子树
            eval(['tree_new' father_path '.left  = tree_brother;']);
        else
            % 兄弟节点是右子树
            eval(['tree_new' father_path '.right = tree_brother;']);
        end
```

```
        % 对兄弟节点组成的新的kdtree进行递归搜索
        [~, return_flag, ~, k_array] = find_k_nearest(test_data, new_leaf_path, tree_new, k_array, k_num, dist_mode);
        if(return_flag==1)
            return;
        else
        end
    end
end
```

最后程序的运行结果如图所示，红色的点就是 test_data，正方形的点是 k 个最近点（红色圆圈内部）。



四、后续思考

1. 原型代码对训练集使用均匀分布，这么做是为了好调试代码，实际运用的时候，训练集应该是分为几类，且维度肯定超过二维。多维空间里的 KNN 搜索，方法原理也是一样的，但是也要考虑到不同维度数据的范围肯定存在很大差别（比方说年龄和身高，数据上下限的范围就不一样），这个时候需要对数据做范围归一化，才能提高 KNN 的判断精度。
2. 在多个维度下，采用 KNN 搜索方法的时间会变得很长，理想状态下最近邻的搜索次数应该是 logN 量级，但是一旦要涉及到寻找 k 个最近点，在多维空间下搜索耗时就会很长，这个时候应该从需求出发，KNN 的目的是找到和点 A 最接近的一个分类，就算有些较近点我们不去搜索，在大部分情况下也不妨碍我们获取到较为准确的结果，所以一个可以提升运行速度的办法是限制回溯的次数。