快速 k-means 聚类算法

分析与实现

薛理立（006146）赵晟(006145)，刘芳（006151）

摘要

k-means 是一种十分流行的聚类技术，在许多方面有着广泛的应用。K-means 算法解决的问题是：给定 d 维空间的 n 个数据点和一个整数 k，寻找 k 个中心点，使得每个数据点到最近中心的平方和最小。而 Lloyd 算法则是 k-means 聚类的一种通行的算法。本文提出了一个简单而快速的 Lloyd 算法的实现，称之为过滤算法。该算法易于实现，和以前的传统 Lloyd 算法的实现截然不同的是该算法巧妙的为数据点建立 kd_tree 而不是为中心点建立 kd-tree. 因此大大提高了聚类的效率。

介绍

聚类问题来源于许多实际的应用，如数据挖掘、图象压缩、模式识别、矢量编码等。而 k-means 是一个重要的聚类问题。

k-means 问题

在 d 维欧氏空间里，给定 n 个数据点，和一个整数 k，求出这样的 k 个中心点，使得给定的 n 个数据点到离各自最近的中心点的距离和最小。

k-means 问题和许多其它的聚类问题密切相关，如欧几里得 k-medicans 问题，该问题要求最小化距离和；k_center 问题，该问题要求最小化最大距离。

由于 k-means 算法有多种变种，为清晰起见我们称为 Lloyd 算法。

Lloyd 算法

```
输入：k 个初始中心点，设每个中心点为 Ci
      Vi：以 Ci 为最近邻的数据点的集合。
输出：k 个中心点
[   While(收敛条件未满足)
    {
        for(I=0;I<k;I++)
        {
            求 Vi 集合的质心,用 Vi 集合的质心代替 Ci；
        }
        for(I=0;I<k;I++)
        {
            在新 C 下更新 Vi；
        }
    }
    返回 k 个中心点位置
]
```

该算法最后可能会收敛到一个局部最小点，因为每个局部最小点对应于一个 vorono 图的 configuration。Lloyd 算法可以和一个全局算法结合起来使用，将 Lloyd 算法作为一个后

处理阶段来提高结果的精度。

由于 Lloyd 算法简单、适应性强，因此 Lloyd 算法十分流行并广泛应用于统计分析。

所有 k-means 算法的一个共同的问题是耗时过多而提高 Lloyd 算法的效率的最常用的方法是在每次递归的初始阶段对中心点进行预处理。

在本文中我们提出了一个简单而高效的 Lloyd 算法的实现，称之为**过滤算法**。过滤算法和其它算法的截然不同之处在于它在预处理阶段为数据点而不是中心点建立了一棵 kd-tree。这正是该算法的出众之处。其优点是

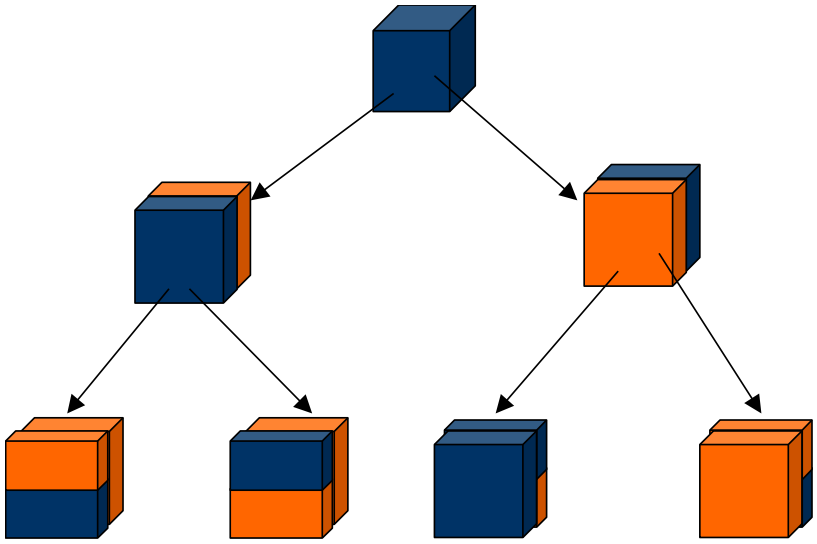
- 由于在计算过程中数据点没有变化，因此在每次迭代的初始阶段没必要对 kd-tree 进行更新。这就节省了时间。
- 一般来说数据点的个数远大于查询点的个数，因此处理速度大大提高。
- 以数据点为中心建立 kd-tree 可以迅速地排除一些不可能的中心，提高算法的效率。

三种 Lloyd 算法的实现

在描述算法之前现介绍一下 kd-tree。

给定空间中 n 个点的点集 S，先构造出包含这些点的超立方体，使用平行于某一个坐标轴的超平面（其方程为 Xi = const）将此超立方体一分为二，使得超平面两边的点的数目大致相等。递归的对左右两个新超立方体进行同样的操作，直到超立方体里的数目小于等于一个数。很显然，这是分治思想设计的一种方法。

KD-Tree 可以用二叉树来存储。KD-Tree 具有查找方便的特点。



3 维的 KD-Tree 示意图

k-means 的算法有很多种，基本上都是基于 Lloyd 算法进行的改进。Lloyd 算法的瓶颈在于寻找最近邻(Nearest neighbor)和收敛速度(convergence speed)。根据目前找到的文献来看主要有：

- Brute-force k-means algorithms

  所谓 Brute-force，就是在 Lloyd 算法中寻找最近邻问题时采用最原始的方法。计算出点到每个中心的距离，从中选出最小值和其对应的中心来。这种求最近邻算法的复杂度是 O(k)。那么 Lloyd 整个一步循环复杂度是 O(k*n)。此算法简单，易于实现，效率最低。

- Building a nearest neighbor search structure for the centers

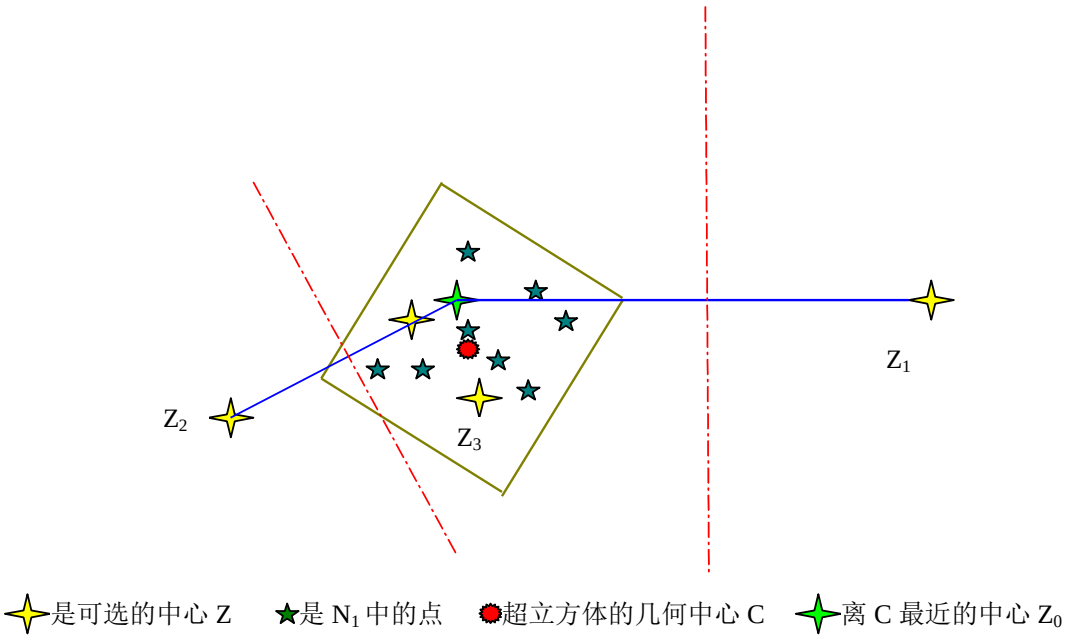
Kd-Tree 是一种描述空间点集的数据结构。先对中心集合作一个 Kd-Tree 结构，用这个结构来查找一个定点的最近邻，以降低复杂度。

此算法中创建 kd-tree 需要一点时间，所以中心数较少时，没有明显的效果，反而出现不如 brute 的方法效率高的结果（参看 2， 1 6 色聚类的情形）。但如果 k 值较大，其优势就上来了。（参看 2 5 6 色聚类的情形）

- Building a nearest neighbor search structure for the points.

这是本文提出的算法。Lloyd 算法的每一步循环，其目的是为了找到新的替代中心，以达到收敛的效果。为此先求出了每个中心关联的点集 P(i)，这也相当于将 n 个点进行了一下分类，即将每个点分配到与它最近的中心所在的类去。基于这种角度，最容易想到的是对每个点逐个分类，即一个重数为 n 的循环。前 2 种方法都是这样的。

如果换了一个角度来思考问题。不是对每个点逐一分类，而是整体上来考虑。注意到一开始不知道 n 个点各自属于那一个中心。但它们所属的中心集合一定是当前 k 个中心的集合的子集。不妨形式化一下，有一个三元组（NP, KP, CP）。NP 是待分类的 m 个点(在根结点 m=n)，KP 是这 m 个点属于的中心的集合，CP 是我们的输入中心集合。KP 应该是 CP 的子集。我们称集合 CP 为可选的中心集合。问题转化为这样，初始的三元组分别是 n 个点，KP 未知，CP 为 k 个中心的集合，要求 KP，在求 KP 的同时可以得到 n 个点各自所属的中心。用分治法来考虑问题，设这 n 点可以分为两部分，集合表示为 N1 和 N2。解子问题(N1, KP, CP)和(N2, KP, CP)，如此递归下去，直到(Ni, KP, CP)。Ni 中只有一个点，这里求 KP 没别的办法，只有 Brute-force 的办法。要使分治法有真正的效果，就必须使得划分的子问题的难度要小于原来的问题。一个可能的降低问题难度的地方对于子问题(N1, KP, CP)利用某种算法，减少集合 CP 中的中心的数目。即过滤掉某些显而易见的元素，它们不可能是 N1 中的点的中心。这样 N1 中的点就不能是随意给定的，它们应该是符合某种分布规律的点集。对于这样的点集，必须设法找到某种过滤算法，来过滤 CP 中的中心点。KD-Tree 正是这样的一种结构，通过对空间的划分，产生很多树节点，每个树节点里的点都是位于一个超立方体中的。下面说明对于这样的局限于一个超立方体中的点集，有一种简单的算法，可以快速的过滤掉 CP 中一些不可能的中心点。以二维的情况为例，如下图所示



对于 N1 中的点先求出包含它们的超立方体的的几何中心 C，找出 CP 中离 C 最近的中心 Z0。再 CP 中考虑除 Z0 外的每个中心 Z。对于超立方体中的任意点 X，如果 X 到 Z 的距离都大于 X 到 Z0 的距离，则显然 Z 不会是点集 N1 中的点的中心。（因为 N1 中的点都在超立方体的内部）。如图中的 Z1 就是这样应该过滤的中心。而对于 Z2，Z3，由于超立方体中的存在点 X ，X 到 Z 的距离都小于 X 到 Z0 的距离，没有充足的理由过滤掉它们。

给定(Z, Z0, HyperBox), 判断超立方体里是否存在点 X，X 到 Z 的距离都小于 X 到 Z0 的距离，用下面的算法实现。事实上不需要判断所有的点，过 Z 和 Z0 的中点，可以做一个超平面，它上面的点到 Z 和 Z0 的距离相等。故此只需要判断超立方体的顶点是否有和 Z 在超平面的同一侧的。如果有，表示存在需求的点 X，否则不存在。

ZZ0 就是所作超平面的法线向量。设 Z 为(x1,x2,…,xd)，Z0 为(y1, y2,…,yd)，则法线向量为(x1 – y1, x2 – y2 ,…, xd- yd)。超平面方程为 f(X1, X2, …, Xd) =

（x1 – y1）(X1 – (x1 + y1) / 2) + (x2 – y2) (X2 – (x2 + y2) / 2) + …+(xd – yd) (Xd – (xd + yd) / 2) = 0

Z 的坐标代入上式左边，f(x1, x2, …, xd) > 0，如果存在某顶点 f(D1) > 0，则表示 D 和 Z 在超平面的同一侧。 超立方体的顶点有 2^d 个，其每一维坐标可以为有两个值选择。只需要计算 f(D)的最大值 Max，如果 Max 大于 0，则有 D1 满足 f(D1)>0

算法代码如下：

```
//返回 TRUE,如果 box 中存在点到 z 的距离比到 z0 的距离近
BOOL Closer(ANNpoint z, ANNpoint z0, ANNpoint lbox, ANNpoint ubox, int dim)
{
    ANNdist dd = annBoxDistance(z, lbox, ubox, dim);
    if (dd == 0)    return TRUE;
//if dd > 0. maybe z0 inside or outside box
    ANNcoord mind;
    mind = 0;
    for (int i = 0; i < dim; i++)
    {
        ANNcoord dz, x ,y , a , b;
        dz = (z[i] - z0[i]);
        x = (lbox[i] - ( z0[i] + z[i]) / 2);
        y = (ubox[i] - ( z0[i] + z[i]) / 2);
        a = dz * x;
        b = dz * y;
        if ( a > b)mind += a;
        else       mind += b;
    }
    if (mind > 0)  return TRUE;
    else return FALSE;
}
```

在进行这样的过滤后，得到新的 CP1，继续划分 N1 为 N3, N4，再类似的计算子问题(N3, KP, CP1)和(N4, KP, CP1)。但是经过的过滤，可能 CP1 中只剩下一个中心 K。这时候不需要继续划分 N1，因为此时的 KP 已经求出，KP=CP1，即 K 就是 N1 中所有点的中心。

下面详细描述 Lloyd 算法在这种方案下的实现：

1. 对 n 个 point 建立一棵 KD-Tree，树的节点有两种类型，一种是 Split-Node，也就是树的内部节点，一种是 Leaf-Node，是树的叶子节点。 Split-Node 里保存的数据有一个超立方体各维坐标的上下界，一个切分超平面的切分位置，本立方体里的数据点的索引的集合，即集合 NP（之所以用索引是为了节约内存），可能的中心点的索引，即 CP。特别的，根结点里存储了所有点的坐标，放在一个数组里。上面所说的索引就是对这个数据的索引。对于中心点也是类似的处理。 Leaf-Node 里仅保存了一个点的索引(NP 只有一个元素)，可能的中心索引的集合（即 CP）。给定初始中心点。
2. 如果没有满足收敛条件，就执行 2，3，否则跳到 5
3. 从根结点开始，调用下面的 Filter 算法。也就是前面所说的分治法，在 KD-Tree 这个数据结构上的实现。

```
void ANNFilterkd_tree::FilterNode(ANNkd_node *node, ANNidxArray pCand, int n_cand)
{
    if (!node || node == KD_TRIVIAL) return;
    if (node->ann_nodetype() == SPLITNODE)                    //是内部节点吗？
    {
        int i;
        ANNkd_split *split = (ANNkd_split *)node;
        for (i = 0; i < dim; i++)
            cellcenter[i] = (lbox[i] + ubox[i]) / 2;        //计算超立方体的几何中心 C
        ANNdist dist = MAXDOUBLE;
        int minz;
        for (i = 0; i < n_cand; i++)
        {
            ANNdist dd;
            if ((dd = annDist(dim, centerArray[pCand[i]], cellcenter)) < dist)
            {
                minz = i;
                dist = dd;
            }
            //在候选中心 CP 里找到距 C 最近的中心 Z0，其索引放在 minz 里

        if (node->n_size < n_cand)
        {
            delete []node->pCandArray;
            node->pCandArray = new ANNidx[n_cand];
            node->n_size = n_cand;
        }
        int k = 0;
        for ( i = 0; i < n_cand; i++)
        {
            if (i != minz)                //对于不是 Z0 的 Z，看看是否需要过滤掉
            {
                if  (Closer(centerArray[pCand[i]],  centerArray[pCand[minz]],  lbox, ubox, dim))
                {
                    node->pCandArray[k] = pCand[i];
                    k++;
                }
            }
```

```
            else
            {
                node->pCandArray[k] = pCand[minz];
                k++;
            }
        }
        node->n_cand = k;
        if ( k == 1)                //只剩下一个中心 Z0，返回上一层调用
        {
            for (int i = 0; i < dim; i++)
                npc[node->pCandArray[0]][i] += split->centerid[i];
            ccount[node->pCandArray[0]] += split->n_points;

        }
        else
        {
            lbox[split->cut_dim] = split->cd_bnds[LO];
            ubox[split->cut_dim] = split->cut_val;
            FilterNode(split->child[LO], node->pCandArray, node->n_cand);       //
对左孩子递归调用 Filter
            lbox[split->cut_dim] = split->cut_val;
            ubox[split->cut_dim] = split->cd_bnds[HI];
            FilterNode(split->child[HI], node->pCandArray, node->n_cand); //对右
孩子递归调用 Filter
            lbox[split->cut_dim] = split->cd_bnds[LO];
            ubox[split->cut_dim] = split->cd_bnds[HI];
        }
    }
    else
    {
    //是叶子节点，包含一个点 P
        ANNkd_leaf *leaf = (ANNkd_leaf *)node;
        ANNdist d = MAXDOUBLE;
        int minz;
        minz = 0;

    //用 Brute-Force 方法，从候选中心中找到离 P 最近的中心
        for (int i = 0; i < n_cand; i++)
        {
            ANNdist dd;
            int n = leaf->n_pts;
            dd = annDist(dim, centerArray[pCand[i]], pts[leaf->bkt[0]]);
            if (dd < d)
            {
                minz = i;
                d = dd;
            }
        }
        for (i = 0; i < dim; i++)
            npc[pCand[minz]][i] += pts[leaf->bkt[0]][i];
        ccount[pCand[minz]] += 1;
    }
}
```

4. 更新中心点，返回 2
5. 算法结束。

此算法需要对 point 节点作 kD-tree,一般 point 的数目都是很多的，好在在我们考虑得情况中 point 数不变，故只需要创建一次就可以了。算法执行的结果表明本算法确实是优于前两种算法的。该算法效率很高，但缺点也是明显的，内存耗费太多。

本程序的运行结果参看附录。可以看到

- K-Means 之类的聚类方法来解决 Color Quantization 不是很适合，效率低下，而且最后的结果还不是很好。只是用来检验过滤算法的效果。
- 原因是 k-means 算法受初值选取影响，另外直接用中心的色彩代替原来的点，不加以抖动之类的处理，视觉上的效果自然很差。

动态演示 K-Means 聚类

在 K-Means 这个程序里实现的主要功能有

- 用户自定义数据点和指定初始中心
- 控制 Lloyd 算法的一些参数。
- 显示聚类的结果。
- 数据点的存取

通过运行这个程序可以看到：

- K-Means 聚类的初始中心很重要，选择的中心点不同，可能导致不同的聚类结果，甚至 Lloyd 算法会失败。
- 对于不同的点的分布，K-Means 给出的结果有好有坏，总的来看，K-Means 适合做 Compat 点集的聚类。

K-Means 实验结果

原始图象

24bit 彩色图象
宽 199
长 164
点数 32636
颜色数目 18373

聚成 2 色

Brute force(2.002000s)

Center KD-Tree(3.444000s)

Point KDT-Tree(1.483000s)

聚成 16 色


Brute force(33.958000s)


Center KD-Tree(35.311000s)


Point KDT-Tree(3.936000s)

聚成 256 色


Brute force(423.729000s)


Center KD-Tree(48.170000s)


Point KDT-Tree(21.260000s)

参考文献(Reference)

[1] Cluster Analysis

[2] Sergios theodoridis and Konstantinos Koutroumbas. Pattern recognition.

[3] Menahem Friedan and Abraham Kandel. Introduction to pattern recognition.

[4] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. Piatko an d R. Silverman, and A. Y. Wu. The analysis of a simple k-means clustering algorithm. Technical Report CAR-TR-937, Center for Automation Research, University of Maryland, College Park, MD, January 2000.