k 均值聚类

1 算法

对于 N 个样点构成的集合 $\{x_i,\ldots,x_N\}$，将其划分为 K 个子集 $\{C_1,\ldots,C_K\}$，这就是聚类分析。所划分的每个子集称为种类。

对于任意样点 x_i，若将其分配至 C_k，分配过程可表示为 $k=A(x_i)$。要保证 x_i 会被正确的分配到 C_k 中，而不是被分配至其他种类，需要有一种方法来判断 x_i 与 C_k 中的样点更接近，还是与其他种类中的样点更接近，亦即需要有一种方法来衡量任意两个样点 x_i 与 x_j 的相异度 $d(x_i,x_j)$——$d(x_i,x_j)$ 的值越大，表示样点 x_i 与 x_j 越相异，反之则表示二者越相似。若样点 x_i 与 C_k 中所有样点的的相异度的总和小于其他种类，便可断定 x_i 应该归属于种类 C_k。

将 $\{x_i,\ldots,x_N\}$ 划分为 $\{C_1,\ldots,C_K\}$ 的过程，本质上是令函数

$$W\big(\{C_1,\ldots,C_K\}\big)=\frac{1}{2}\sum_{k=1}^{K}\sum_{A(x_i)=k}\sum_{A(x_j)=k}d(x_i,x_j) \tag{1}$$

最小化，亦即令每个种类内的样点之间的相异度之和最小。

最简单的聚类分析方法是仅定义 $d(x_i,x_j)$，而不去考虑 $A(x_i)$。例如，可通过枚举各种可能的种类——聚类分析问题的可行解，将其代入式 (1)，选出能够最小化式 (1) 的可行解作为最优解。

不幸的是，聚类分析问题的可行解数量往往是天文数字。若将 $\{x_i,\ldots,x_N\}$ 划分为 K 个种类，聚类问题的可行解其数量为：

$$S(N,K)=\frac{1}{K!}\sum_{k=1}^{K}(-1)^{K-k}\binom{K}{k}k^N \tag{2}$$

对于现实中的聚类分析问题而言，N 的值通常很大。即使 N 取不是太大的值，譬如 30，而 K 值为 2，可行解也有 5 亿 3 千多万个，并且其中大部分可行解不符合聚类目标。通过枚举所有可行解来获取最优结果的方法显然不适于数量较多的样点集合的聚类分析。

k 均值算法提供了一种 $A(x_i)$ 的定义，即

$$A(x_i) = \arg\min_k d(x_i, m_k) \tag{3}$$

其中，m_k 为 C_k 的中心——C_k 中所有样点的均值点。这个公式的含义是，容纳 x_i 的种类 C_k，其中心与 x_i 的相异度必须小于其他种类。

由于 $\{C_1, \ldots, C_K\}$ 中每个种类初始时均为空集，要让式 (3) 能够发挥作用，必须为它提供一组初始的种类中心。k 均值算法将 $\{x_i, \ldots, x_N\}$ 中随机选取的 K 个样点分别初始种类中心，然后基于式 (3) 将 $\{x_i, \ldots, x_N\}$ 中的每个样点分配至 $\{C_1, \ldots, C_K\}$ 中的相应的种类。

在完成上述 $\{C_1, \ldots, C_K\}$ 的初始化工作之后，k 均值算法反复执行以下过程

由 $\{C_1, \ldots, C_K\}$ 产生 K 个均值点 $\{m_1, \ldots, m_K\}$；
将 $\{C_1, \ldots, C_K\}$ 中的每个种类清空；
for x_i in $\{x_i, \ldots, x_N\}$ {
    $k^* \leftarrow \arg\min_k d(x_i, m_k)$；
    将 x_i 添加到 C_{k^*} 中；
}

直至 $\{C_1, \ldots, C_K\}$ 的每个种类的中心不再发生变化时为止。最终所得 $\{C_1, \ldots, C_K\}$ 便是聚类分析结果。

2 初始化

`km_init` 函数实现了 k 均值聚类初始化过程：从样点集合中选取指定数量的样点作为初始种类中心，然后初始化种类树，亦即对样点集合进行首次分类。

```
@ agn_km.c #
<km-init>
static AgnTree *km_init(AgnArray *points, size_t K)
{
        size_t *indices = generate_indices(K, 0, points->n - 1);
        # 初始化种类中心 -> init_centers @ <3>
        # 初始化种类树 -> class_tree @ <3>
        agn_array_free(init_centers);
        free(indices);
        return class_tree;
}
```

km_init 函数的返回结果是种类树，它是 AgnTree 的一个实例。种类树的结构分为三层，根结点，亦即第一层结点，不存放数据；第二层结点表示种类，存放的数据为种类中心。第三层结点存放分配到相应种类的样点序号。

初始种类中心的样点，是从 points 中随机选取的：

```
@ 初始化种类中心 -> init_centers #
AgnArray *init_centers = agn_array_alloc(K);
for (size_t i = 0; i < K; i++) {
        init_centers->body[i] = agn_point_copy(points->body[indices[i]]);
}
=> agn_km.c <2>
```

种类中心初始化过程中所用的 indices 是 $[0, n-1]$ 内 K 个不重复的随机值，由 generate_indices 生成：

```
@ agn_km.c # <km-init> ^+
static bool index_recurring(size_t index, size_t *indices, size_t n)
{
        for (size_t i = 0; i < n; i++) {
                if (indices[i] == index) return true;
        }
        return false;
}

static size_t *generate_indices(size_t K, size_t from, size_t to)
{
        size_t *indices = malloc(K * sizeof(size_t));
        srand(time(NULL));
        for (size_t i = 0; i < K; i++) {
                size_t index;
                do {
                        index = rand() % to + from;
                } while (index_recurring(index, indices, i));
                indices[i] = index;
        }
        return indices;
}
```

获取初始种类中心后，便可初始化种类树——将样点集合记录于树的根结点，将初始种类中心存入树的第二层结点，然后基于初始种类中心，将样点分配至树的第三层结点：

```
@ 初始化种类树 -> class_tree #
AgnTree  *class_tree = agn_tree_alloc();
for (size_t i = 0; i < K; i++) {
```

```
        agn_tree_prepend(class_tree, init_centers->body[i]);
}
assign_points(class_tree, points);
=> agn_km.c <2>
```

assign_points 函数用于样点分配：

```
@ agn_km.c # <km-init> ^+
static void assign_points(AgnTree *class_tree, AgnArray *points)
{
        for (size_t i = 0; i < points->n; i++) {
                AgnTree *t = class_tree->lower;
                for (AgnTree *it = t->next; it != NULL; it = it->next) {
                        if (agn_point_cmp(it->data,
                                          t->data,
                                          points->body[i]) <= 0) {
                                t = it;
                        }
                }
                size_t *id = malloc(sizeof(size_t));
                *id = i;
                agn_tree_prepend(t, id);
        }
}
```

对于两个不同的种类中心，agn_point_cmp 能够比较出它们中哪一个距距样点更近，这样便可为当前样点选择距其最近的种类中心，从而将样点加入相应的种类。

3 聚类

agn_km_classify 函数实现了样点数据的 k 均值聚类过程：

```
@ agn_km.c # +
<agn-km-classify>
AgnTree *agn_km_classify(AgnArray *points, size_t K)
{
        AgnTree *class_tree = km_init(points, K);
        while (!class_tree_stablized(class_tree, points)) {
                # 样点重新分类 @ <6>
        }
        return class_tree;
}
```

辅助函数 class_tree_stablized 用于判断种类树是否处于稳定状态类：

```
@ agn_km.c # <agn-km-classify> ^+
static bool class_tree_stablized(AgnTree *class_tree, AgnArray *points)
{
        # 生成新旧种类中心集合 -> centers 与 new_centers @ <5>
        # 判断 centers 与 new_centers 是否相同 -> stablized @ <6>
        for (size_t i = 0; i < centers->n; i++) {
                agn_point_free(centers->body[i]);
        }
        agn_array_free(centers);
        agn_array_free(new_centers);
        return stablized;
}
```

种类树第 2 层结点中存储的各个种类的中心并非当前种类的中心，而是上一代种类的中心。将当前的种类中心与种类树第 2 层结点中存储的上一代种类的中心进行比较，若二者相同，便意味着种类树已经达到稳定状态。上一代种类中心的提取，当前的种类中心的计算以及种类树中第二层结点所存储的数据更新等运算，可以揉合到同一个种类树第二层结点的遍历过程中，即：

```
@ 生成新旧种类中心集合 -> centers 与 new_centers #
AgnArray *centers = agn_array_alloc(0);
AgnArray *new_centers = agn_array_alloc(0);
agn_tree_foreach(class_tree, class) {
        agn_array_append(centers, agn_point_copy(class->data), NULL);
        if (class->n > 0) {
                AgnArray *o_points = agn_array_alloc(0);
                agn_tree_foreach(class, it) {
                        agn_array_append(o_points,
                                         points->body[*(size_t *)(it->data)],
                                         NULL);
                }
                void *new_center = agn_points_center(o_points);
                agn_point_free(class->data);
                class->data = new_center;
                agn_array_append(new_centers, new_center, NULL);
                agn_array_free(o_points);
        } else {
                agn_array_append(new_centers, class->data, NULL);
        }
}
=> agn_km.c <5>
```

需要注意，在 k 均值聚类过程中，有时会出现某个种类没有被分配到样点，导致种类为空，此时只需保持种类树第二层中相应的结点数据不变即可。

下面这段代码用于判断新旧种类中心是否相同，若它们完全相同，则意味着种类已达到稳定状态，否则意味着种类还不稳定。

```
@ 判断 centers 与 new_centers 是否相同 -> stablized #
bool stablized = true;
for (size_t i = 0; i < centers->n; i++) {
        if (!agn_point_eq(centers->body[i], new_centers->body[i])) {
                stablized = false;
                break;
        }
}
=> agn_km.c <5>
```

若种类尚不稳定，便需要清空所有已分配给它们的样点，然后重新向它们分配样点：

```
@ 样点重新分类 #
empty_classes(class_tree);
assign_points(class_tree, points);
=> agn_km.c <4>
```

其中，empty_classes 函数的定义如下：

```
@ agn_km.c # <agn-km-classify> ^+
static void empty_classes(AgnTree *class_tree)
{
        agn_tree_foreach(class_tree, class) {
                AgnTree *member = class->lower;
                while (member) {
                        AgnTree *next_member = member->next;
                        free(member->data);
                        agn_tree_free(member);
                        member = next_member;
                }
                class->lower = NULL;
        }
}
```

agn_km_classes_free 用于释放 agn_km_classify 返回的种类树：

```
@ agn_km.c # +
void agn_km_classes_free(AgnTree *class_tree)
{
```

```
        agn_tree_foreach(class_tree, class) {
                agn_tree_foreach(class, it) free(it->data);
                agn_point_free(class->data);
        }
        agn_tree_free(class_tree);
}
```

4 测试

```
@ km_test.c #
int main(int argc, char **argv)
{
        size_t K = atoi(argv[1]);
        AgnArray *points = agn_points_load_into_array(argv[2], 3);
        if (K > points->n) {
                fprintf(stderr, "Error: I can not know how to classify!");
                exit(EXIT_FAILURE);
        }
        AgnTree *class_tree = agn_km_classify(points, K);
        # 输出种类数据 @ <7>
        agn_km_classes_free(class_tree);
        agn_point_array_free(points);
        return 0;
}
```

样点数据的文件名与种类数量 K 分别通过程序的命令行参数 `argv[1]` 与 `argv[2]` 设定。对于聚类结果的输出，若所读取的样点数据文件名为 foo（建议文件名不要有后缀），那么种类数据会输出到 foo-1.asc, foo-2.asc, ⋯, foo-K.asc 等文件中，实现这一过程的代码如下：

```
@ 输出种类数据 #
size_t k = 1;
agn_tree_foreach(class_tree, class) {
        char of_name[BUFSIZ];
        sprintf(of_name, "%s-%ld.asc", argv[2], k);
        FILE *fp = fopen(of_name, "w");
        agn_tree_foreach(class, it) {
                agn_point_print(fp, points->body[*(size_t *)(it->data)], " ");
                fprintf(fp, "\n");
        }
        fclose(fp);
        k++;
}
```

```
=> km_test.c <7>
```

要构造完备的测试环境，还需要增设 agn_km.h，agn_point.h 等头文件，并且在各个.c 文件中载入它们。这部分琐碎的工作见本文附录部分。现在假设测试程序相关的源码均已备好，使用 Bash 脚本 gen-km-src.sh（其内容见附录部分）从相关的 zero 源文件中提取它们：

```
$ mkdir km-test
$ cd km-test
$ wget http://192.168.0.7:8080/meta-doc/agn_km.zero -O agn_km.zero
$ zero -m night -e "gen-km-src.sh" agn_km.zero
$ sh ./gen-km-src.sh
```

获得所有源码文件之后，使用以下命令编译将其编译为可执行文件 km-test：

```
$ gcc -std=c11 -Wall -pedantic -g -I./ \
  agn_array.c agn_list.c agn_tree.c agn_km.c agn_point.c km_test.c -o km-test
```

在 valgrind 环境中执行 km-test：

```
$ valgrind --leak-check=full ./km-test 5 foo
```

若上述命令在当前目录下生成了 foo-1.asc, ⋯, foo-5.asc，并且它们如下图 **1** 所示——可以用 FreeCAD 可视化这些文件，这意味着测试程序运行过程是正确的。测试数据 foo 包含了 4969 个三维样点，在 CPU 主频为 2.5GHZ 的机器上，聚类时间约为 0.2 秒。

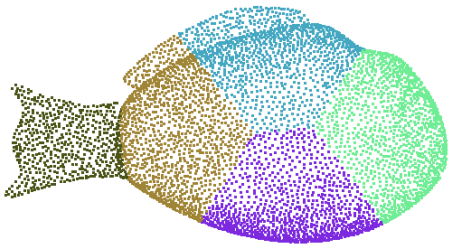


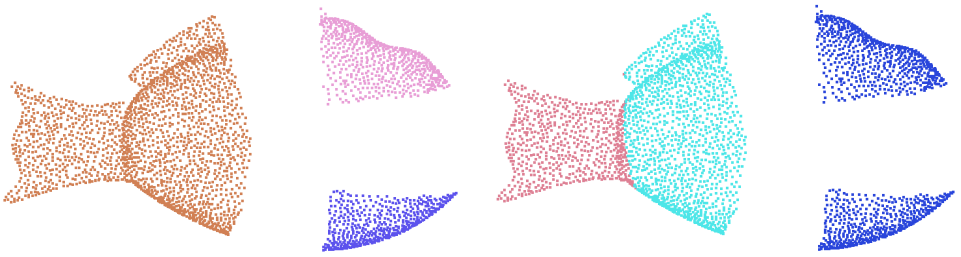
图 1　三维样点数据用 k 均值聚类算法划分为 5 个种类的结果

本文首页中的插图，是一份龙的样点数据（样点数量 150 万）采用上述测试程序划分为 50 类的结果，在 CPU 主频 2.5 GHZ 的机器上，用时约为 10 分钟。

_若要提高聚类过程的计算效率，可以对 `calsses_stablized` 函数的「@ 生成新旧种类中心集合 -> centers 与 new_centers #」部分进行优化，因为这部分代码中需要用 $O(n)$ 的时间获得各个种类所包含的点数。优化方法是在分类树的第 2 层结点中增加一个变量，用它记录种类所包含的点数。

5 讨论

对于种类界限非常明确的样点数据，这个程序也有可能会得到图 2(b) 所示的结果。不知道你怎么看待这个结果，我觉得它不如图 2(a) 所示的结果。之所以会出现这样的结果，并非是因为我写的程序有问题，这是 k 均值算法自身的问题——k 均值聚类算法的计算结果不一定是最优分类结果。



(a) 合理的聚类结果　　(b) 不合理的聚类结果

图 2　两种聚类结果

在现实中，种类与种类的边界并不像 k 均值聚类算法中那样清晰。有些种类之间存在交集，对于交集中的样点，很难判定它们应该归属于哪个种类，这时即使 k 均值聚类算法能够得到最优解，但这个解未必是我们想要的，例如图 3 所示的分类结果。之所以会出现不合理的分类结果，其根源在于它认为种类的均值点可以表征这个种类本身。事实上，只有符合高斯独立同分布而且样点数量相当的种类，其均值点才是有意义的。

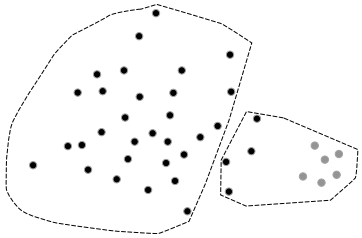


图 3　不正确的 2 均值聚类结果

曾思考过很久，试验过好多种改进 k 均值聚类算法的方案，但是这些方案很快就被我推翻了。我的建议是，不要试图脱离具体问题领域而去研究如何改进 k 均值聚类算法以产生合理的分类结果，徒劳无益。

6 附录

```
@ agn_km.h #
#ifndef AGN_KM_H
#define AGN_KM_H
#include <agn_tree.h>
#include <agn_point.h>

@ agn_km.h # +
AgnTree *agn_km_classify(AgnArray *points, size_t K);
void agn_km_classes_free(AgnTree *class_tree);
#endif

@ agn_km.c # ^+
#include <stdlib.h>
#include <float.h>
#include <stdbool.h>
#include <time.h>
#include "agn_km.h"

@ km_test.c # ^+
#include <stdio.h>
#include <stdlib.h>
#include <error.h>
#include "agn_km.h"
```

```
@ gen-km-src.sh #
#!/bin/bash
LOCATION="http://192.168.0.7:8080"
for i in array list tree ; do
    wget $LOCATION/meta-doc/agn_${i}.zero -O agn_${i}.zero
    zero -m night -e "agn_${i}.h, agn_${i}.c" agn_${i}.zero
done

wget $LOCATION/meta-doc/agn_km.zero -O agn_km.zero
POINTS_KM="agn_km.h, agn_km.c, km_test.c"
zero -m night -e "$POINTS_KM" -o "$POINTS_KM" agn_km.zero

wget $LOCATION/meta-doc/agn_m4_macros.zero -O agn_m4_macros.zero
zero -e "agn_m4_macros.m4, agn-m4.sh" agn_m4_macros.zero
sed -i '/[[:blank:]]*#line.*/d' ./agn-m4.sh
chmod +x agn-m4.sh

wget $LOCATION/meta-doc/agn_point.zero -O agn_point.zero
for i in h c ; do ./agn-m4.sh agn_point.zero agn_point.${i}_T ; done

wget $LOCATION/files/foo -O foo
```

7 后记

虽然所有的代码都是自己写的，但是有时也会脑子抽风，犯一些愚蠢但是又很难发现的错误。譬如，第 **3** 节的 `empty_classes` 函数，一开始我将其定义为：

```
static void empty_classes(AgnTree *class_tree)
{
        agn_tree_foreach(class_tree, class) {
                agn_tree_foreach(class, member) {
                        agn_tree_free(member);
                }
                class->lower = NULL;
        }
}
```

结果所得测试程序 km-test，直接运行它不会出错，并且能够得到正确的样点数据聚类结果，但是拿到 valgrind 环境中运行，检测结果如下：

```
==25725== Invalid read of size 8
==25725==    at 0x401AE3: empty_classes (km.zero:249)
==25725==    by 0x401B50: agn_km_classify (km.zero:239)
==25725==    by 0x402283: main (km.zero:481)
==25725==  Address 0x54b4138 is 8 bytes inside a block of size 32 free'd
==25725==    at 0x4C2D250: free (in /...path.../vgpreload_memcheck-amd64-linux.so)
==25725==    by 0x401296: agn_tree_free (agn_tree.zero:69)
==25725==    by 0x401ADE: empty_classes (km.zero:250)
==25725==    by 0x401B50: agn_km_classify (km.zero:239)
==25725==    by 0x402283: main (km.zero:481)
==25725==  Block was alloc'd at
==25725==    at 0x4C2BFE0: malloc (in /...path.../vgpreload_memcheck-amd64-linux.so)
==25725==    by 0x401201: agn_tree_alloc (agn_tree.zero:47)
==25725==    by 0x4012B0: agn_tree_prepend (agn_tree.zero:86)
==25725==    by 0x4016C4: assign_points (km.zero:156)
==25725==    by 0x401801: km_init (km.zero:138)
==25725==    by 0x401B3E: agn_km_classify (km.zero:172)
==25725==    by 0x402283: main (km.zero:481)
```

将相关代码看了很多遍，不得其解。午饭没吃踏实。下午猛不丁想到，犯了个很低级的错误，不应该在遍历一棵树或列表的过程去删除结点。这种行为类似于边过河边拆桥。

由于 `agn_tree_foreach` 是一个宏：

```
#define agn_tree_foreach(tree, it) \
        for (AgnTree *it = tree->lower; it != NULL; it = it->next)
```

因此上述代码中用于释放树结点的迭代过程的代码：

```
agn_tree_foreach(class, member) {
        agn_tree_free(member);
}
```

会被展开为：

```
for (AgnTree *member = class->lower; member != NULL; member = member->next) {
        agn_tree_free(member);
}
```

在循环过程中，释放 member 所指向的 AgnTree 实例之后，member 便成为悬挂指针，它的值并没有变，亦即它依然指向原来它所指向的内存区域。这块内存区域可能依然保持着原来的 AgnTree 实例数据，这便是直接运行 km-test 之所以没有出错的原因，而 valgrind 犀利的看到了隐患。

一边过河一边拆桥，也不是不可以。上面代码的做法类似于拆掉前面尚未走到的桥段，没掉进河里，全凭侥幸。拆毁的桥段欲坠未坠，我的脚踏了上去，然后侥幸的走了过去……正确的做法是拆身后的已走过去的桥段，亦即：

```
agn_tree_foreach(class_tree, class) {
        AgnTree *member = class->lower;
        while (member) {
                AgnTree *next_member = member->next;
                agn_tree_free(member);
                member = next_member;
        }
        class->lower = NULL;
}
```

看到这里，也许你就会明白，为什么 C/C++ 这样的语言吓走了很多人，他们去热烈拥抱 Java、C#、go 之类的语言了，还有很多人认为 Python 可以拯救世界。