中山大学数据科学与计算机学院

移动信息工程专业-数据仓库与数据挖掘

本科生实验报告

教学班级	1515	专业（方向）	软件工程
学号	15352334	姓名	吴佳卫

一、 实验题目

聚类算法

二、 实验内容

1. 算法原理

K-means

k-means 算法的核心思想是每一次迭代都最小化所有样本到其对应的聚类中心的欧氏距离和，也可以理解为 k-means 算法的目的是将每一个样本都分配到与其欧氏距离最小的类中心对应的类中。

在为一个样本点 i 分配其对应的类 c(i)时，我们遵从的公式是：

$$c(i) = \arg\min_j \|x_i - \mu_j\|^2$$

其中μ_j指代的是聚类 j 的中心点。

每次迭代分配完所有样本点后，由于原有的聚类已经改变，因此需要更新所有类的类中心：

$$\mu_j = \frac{\sum_{i=1}^{m} x_i | i \epsilon c_j}{m}$$

其中 m 指的是聚类 j 中样本点的个数。

重复以上步骤直到收敛或者达到了指定的迭代次数。

DBSCAN

DBSCAN 基于的基本思想是：一个聚类可以由其中的任何核心对象唯一确定。算法要求一个聚类内对于每一个点，其周围一定空间内的点的数目不小于一个阈值。

与 K-means 算法不同的是，DBSACN 并没有一个显式的目标函数，而是一个类似启发式的算法。

DBSCAN 算法由一下三个核心操作组成：

1. 检查数据集中每点的 Eps 邻域来搜索簇, 如果点 p 的 Eps 邻域包含的点多于 MinPts 个，则创建一个以 p 为核心对象的簇；
2. DBSCAN 迭代地聚集从这些核心对象直接密度可达的对象；
3. 当没有新的点添加到任何簇时，该过程结束；

谱聚类

谱聚类算法是基于图论的算法，其主要的思想是把空间内的所有数据点连接起来，连接的强度与点与点之间的距离成反比。随后进行切图的工作，强度越低的连接（即距离较远的点）会被有限切割开。最终的目的是切图后不同的子图间边权重和尽可能的低，而子图内的边权重和尽可能的高，以此达到了聚类的目的。

在有数据点生成图的时候有两种可行的方式：

1. KNN 即 K 临近法，如果两个点互为 k 临近那么就认为这两个点有一条边相连接。这种方法也有一个弱化的版本，也就是主要求一个点在另一个的 k 临近中就可以判断相连。
2. 全连接法，非常简单粗暴。直接将所有的点都两两链接，也就是说所有点之间的权重值都是大于零的。

当我们得到图后就可以计算其对应的 W 矩阵和 D 矩阵了，两个矩阵的定义分别如下：

$$W_{ij} = \exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})$$

$$D_{i=\sum_{j=1}^{n} w_{ij}}$$

需要注意的是由于我使用的是全连接法，因此生成 W 矩阵式使用到的是高斯相似度的计算公式。至于 D 就直接把每个点的权值相加就可以了。

然后生成标准化拉普拉斯矩阵

$$L_{sym} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$$

为了计算方便这条公式是可以继续化简的：

$$L_{sym} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$$

$$= I - D^{-\frac{1}{2}} (D - L) D^{-\frac{1}{2}}$$

$$= I - \left(D^{\frac{1}{2}} + D^{-\frac{1}{2}} L\right) D^{-\frac{1}{2}}$$

$$= I - I + D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$$

$$= D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$$

为了避免最小切图导致的切图效果不佳，我们需要对每个子图的规模做出限定。讨论到割图的部分，一共有两种割图的方法即 RadioCut 和 NormalCut。两种准则的区别是对应目标函数的不同。

$$\min_{F \in R^{N \times K}} Tr(F^T L F) \quad 与 \quad \min_{F \in R^{N \times K}} Tr(F^T L_{sym} F)$$

在使用以上算法流程解出 F 后，再使用类似 K-means 的算法进行聚类，也就是可理解为前面的算法都是一个特征提取的过程

DenPeak

DensityPeak 是一个很有趣的聚类算法，与 DBSCAN 有些相似。DensityPeak 算法基于的基本原理是聚类中心周围都是密度比其低的点，同时这些点距离该聚类中心的距离相比于其他聚类中心最近。

我们定义了局部密度ρ和距离δ，定义分别如下：

$$\rho_i = \sum_j X(d_{ij} - d_c)$$

$$\delta_i = \min_{j:j>\rho_i}(d_{ij})$$

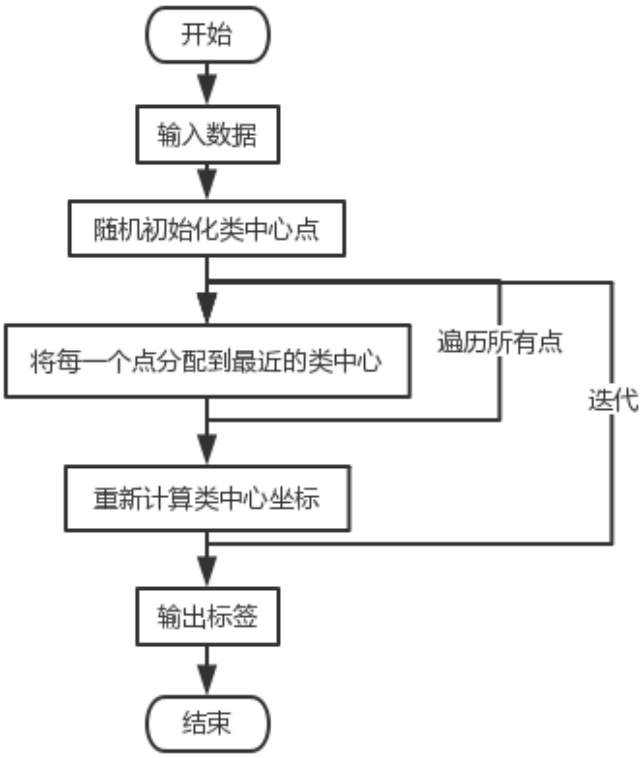
其中$X(n) = \begin{cases} 1, & x < 0 \\ 0, & x \geq 0 \end{cases}$，$d_c$是阶段距离，定义为所有点的相互距离中由小到大排列占总数 2%的位置距离数值。同时可以看出距离针对的是比当前点 i 密度高且距离最近的点。

聚类中心可以理解为ρ和δ都相对较大的点，可以通过给定ρ和δ的阈值来筛选中心点。

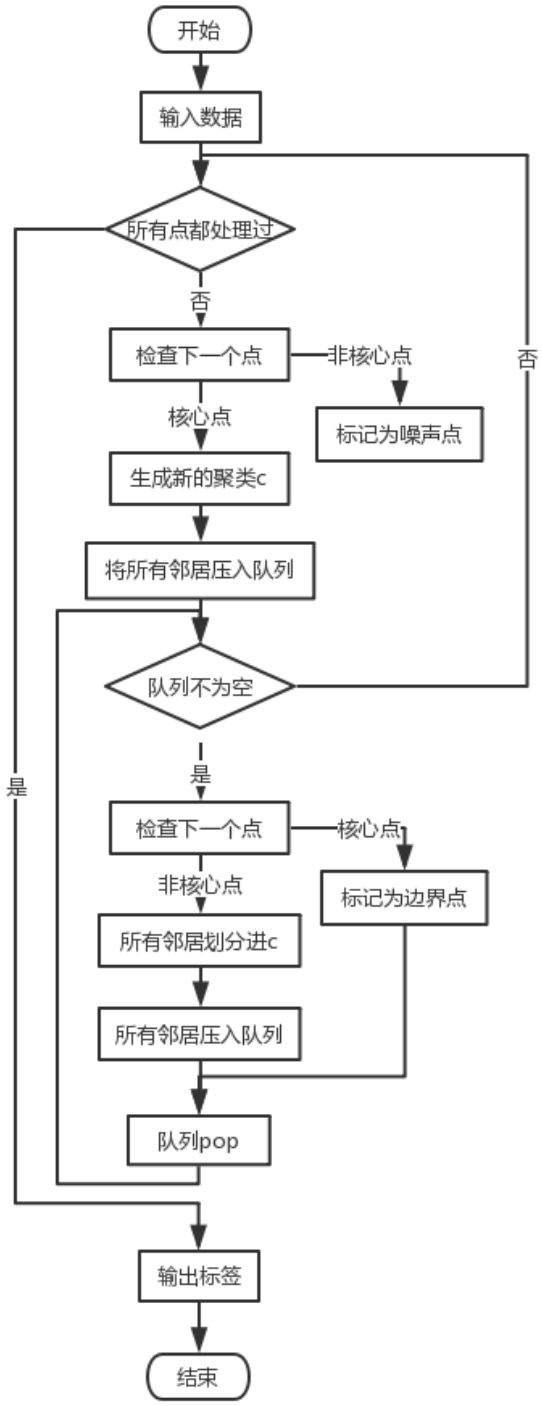
剩余点的指派遵从以下法则：某一个点的标签应该与与其最近且密度更高的点一致。

2. 伪代码

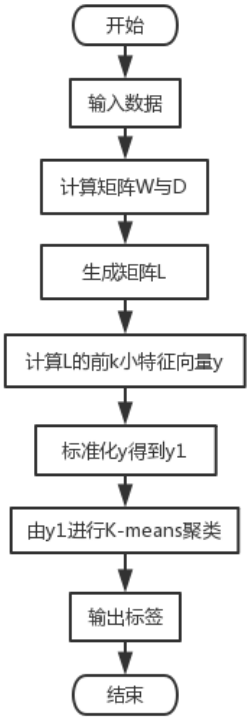
K-means



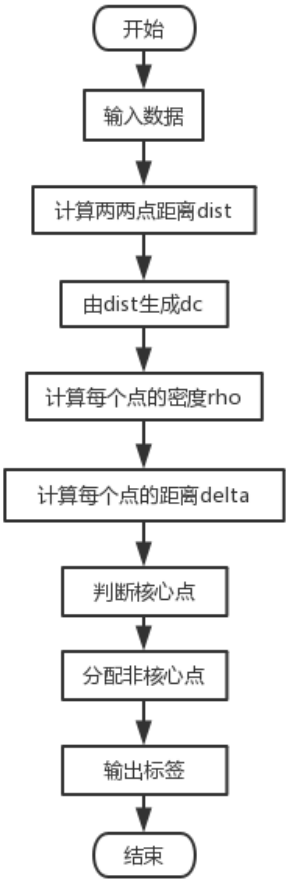
DBSCAN


开始
输入数据
所有点都处理过
否
检查下一个点
非核心点
标记为噪声点
核心点
生成新的聚类c
将所有邻居压入队列
队列不为空
是
检查下一个点
核心点
标记为边界点
非核心点
所有邻居划分进c
所有邻居压入队列
队列pop
是
否
输出标签
结束

谱聚类


开始
输入数据
计算矩阵W与D
生成矩阵L
计算L的前k小特征向量y
标准化y得到y1
由y1进行K-means聚类
输出标签
结束

DenPeak


开始
输入数据
计算两两点距离dist
由dist生成dc
计算每个点的密度rho
计算每个点的距离delta
判断核心点
分配非核心点
输出标签
结束

3. 关键代码截图

K-means

Kmeans 的实现比较简单，主要的核心只做了两个操作。第一个是根据欧氏距离将所有的点分别到最近的聚类中心，第二个是重新计算所有聚类的中心坐标。

```
void KMeans()
{
    for(int i = 0; i < V0.size(); i ++)
    {
        int n_point = -1;
        double min_dis = 999999;
        for(int j = 0; j < C0.size(); j ++)
        {
            point center;
            center.x = C0[j].center_x;
            center.y = C0[j].center_y;
            double tmp_dis = cal_dis(center, V0[i]);
            if(tmp_dis < min_dis)
            {
                min_dis = tmp_dis;
                n_point = j;
            }
        }
        if(n_point == -1) cout << "ERRROR" << endl;
        C0[n_point].V.push_back(V0[i]);
        V0[i].flag = n_point;
    }
    for(int i = 0; i < C0.size() ; i ++)
    {
        C0[i].update_center();
    }
}
```

更新的过程也是一个简单的求几何中心的过程。

```
void update_center()
{
    double sum_x = 0;
    double sum_y = 0;
    for(int i = 0; i < V.size(); i ++)
    {
        sum_x += V[i].x;
        sum_y += V[i].y;
    }
    sum_x /= V.size();
    sum_y /= V.size();
    center_x = sum_x;
    center_y = sum_y;
};
```

DBSCAN

对于 DBSCAN 算法很重要的一个函数就是检查一个点是否是中心点。

```
bool check_core(point p)
{
    int cnt = 0;
    for(int i = 0; i < P.size(); i ++)
    {
        double tmp_dis = pow((p.x - P[i].x), 2) + pow((p.y - P[i].y), 2);
        double tmp_ep = pow(epsion, 2);
        if(tmp_dis <= tmp_ep && tmp_dis > 0)
        {
            cnt ++;
        }
    }
    if(cnt >= density) return true;
    else return false;
}
```

还有一个比较关键的步骤就是队列 Q 的设置，当我们将一个点划分入聚类时需要将他的邻居点都压入一个队列 Q。

```
for(int j = 0; j < P.size(); j ++)
{
    double tmp_dis = pow((P[j].x - P[i].x), 2) + pow((P[j].y - P[i].y), 2);
    double tmp_ep = pow(epsion, 2);
    if(tmp_dis <= tmp_ep && tmp_dis > 0)
    {
        Q.push(P[j]);
        tmp_c.p_set.push_back(P[j]);
        P[j].flag = 2;
    }
}
```

然后就是针对 Q 内的点进行遍历操作，同样也涉及上面的操作直至队列清空。这里可以理解为一个深度优先搜索的过程。

```
while(! Q.empty())
{
    if(check_core(Q.front()))
    {
        P[Q.front().id].flag = 1;
        for(int k = 0; k < P.size(); k ++)
        {
            if(P[k].flag == -1)
            {
                double tmp_dis = pow((P[k].x - Q.front().x), 2) + pow((P[k].y - Q.front().y), 2);
                double tmp_ep = pow(epsion, 2);
                {
                    if(tmp_dis <= tmp_ep && tmp_dis > 0)
                    {
                        Q.push(P[k]);
                        tmp_c.p_set.push_back(P[k]);
                        P[k].flag = 2;
                    }
                }
            }
        }
    }
    else
    {
        P[Q.front().id].flag = 2;
    }
    Q.pop();
}
```

谱聚类

对于谱聚类算法而言，最重要的自然是生成所有的矩阵。

```
for(int i = 0; i < size; i ++)
{
    W[i][i] = 0;
    for(int j = i + 1; j < size; j ++)
    {
        W[i][j] = cal_dis(P[i], P[j]);
        W[j][i] = W[i][j];
    }
}
```

```
for(int i = 0; i < size; i ++)
{
    double tmp_w = 0;
    for(int j = 0; j < size; j ++)
    {
        tmp_w += W[i][j];
    }
    D[i] = tmp_w;
}
```

```
for(int i = 0; i < size; i ++)
{
    L[i][i] = D[i] - W[i][i];
    for(int j = i + 1; j < size; j ++)
    {
        L[i][j] = 0 - W[i][j];
        L[j][i] = L[i][j];
    }
}
```

```
for(int i = 0; i < size; i ++)
{
    for(int j = 0; j < size; j ++) L0[i][j] = L[i][j] / sqrt(D[i] * D[j]);
}
```

由于我是用的语言是 c++，因此矩阵的特征值特征向量计算是要使用第三方库 Eigen 的。将 Eigen 返回的特征值和特征向量存储到 vector 中就可以进行进一步的操作。

```
Matrix<double, Dynamic, Dynamic> U;
U.resize(size, size);
for(int i = 0; i < size; i ++)
{
    for(int j = 0; j < size; j ++) U(i, j) = L0[i][j];
}
cout << "Matrix Assignment Success!" << endl;
EigenSolver<MatrixXd> es(U);
eigen_val.resize(size);
eigen_vec.resize(size);
for(int i = 0; i < size; i ++) eigen_vec[i].resize(size);
y.resize(k);
for(int i = 0; i < k; i ++) y[i].resize(size);

for(int i = 0; i < size; i ++)
{
    eigen_val[i] = es.eigenvalues()[i].real();
}
cout << "Matrix Eigen Values Success!" << endl;
Matrix<complex<double>, Dynamic, Dynamic> m = es.eigenvectors();
for(int i = 0; i < size; i ++)
{
    for(int j = 0; j < size ; j ++)
    {
        eigen_vec[i][j] = m.row(i)[j].real();
    }
}
cout << "Matrix Eigen Vector Success!" << endl;
```

当完成上述的 Generate 过程后，直接接入 K-means 进行聚类。

```
int main()
{
    Init();
    Input(input_file);
    Generate();
    KM_Rand_Init();
    for(int i = 0; i < iteration; i ++)
    {
        KMeans();
        if(i % 100 == 0) cout << i << endl;
    }
    Output(output_file);
}
```

DenPeak

Density peak 聚类方法也涉及了许多需要计算的参数。

```
class Cluster
{
    public:
        Cluster(string filename);
        int fclust();
        void getdist();
        void getdc();
        void getrho();
        void getdelta();
        void assign();
        void Output(string filename);
    private:
        vector<point> data;
        double dc;
        vector<vector<double> > t_dist;
        double t_neighbor_rate;
        double maxdist;

        vector<double> t_rho;
        vector<node> t_orderrho;
        vector<double> t_delta;

        vector<int> t_neighbor;
        double t_maxrho;
        double t_minrho;

        int t_clusterNum;
        vector<int> cl;
        vector<int> icenter_of_class;
};
```

生成 dc。

```
void Cluster::getdc()
{
    vector<double> alldist;
    for(int i = 0; i < t_dist.size() - 1; i ++)
    {
        for(int j = i + 1; j < t_dist.size(); j ++)
        {
            alldist.push_back(t_dist[i][j]);
        }
    }
    sort(alldist.begin(), alldist.end());
    dc = alldist[static_cast<int>(alldist.size()) * t_neighbor_rate];
    maxdist = alldist[alldist.size() - 1];
}
```

计算点的密度，这里使用了高斯相似度的计算公式。

```
void Cluster::getrho()
{
    for(int i = 0; i < t_dist.size() - 1; i ++)
    {
        for(int j = i + 1; j < t_dist.size(); j ++)
        {
            double distij = t_dist[i][j];
            t_rho[i] = t_rho[i] + exp(-1 * (pow(t_dist[i][j] / dc, 2)));
            t_rho[j] = t_rho[j] + exp(-1 * (pow(t_dist[i][j] / dc, 2)));
        }
    }

    for(int i = 0; i < t_rho.size(); i ++)
    {
        t_orderrho[i].id = i;
        t_orderrho[i].rho = t_rho[i];
    }
    sort(t_orderrho.begin(), t_orderrho.end(), comp);
}
```

以及计算每一个点距离，注意这里的距离指的是距离密度高于它的点的最小距离。

```
void Cluster::getdelta()
{
    t_delta.resize(data.size(), 0);
    t_neighbor.resize(data.size(), -1);
    t_delta[t_orderrho[0].id] = -1.0;
    t_neighbor[t_orderrho[0].id] = -1;

    for (int i = 0; i < data.size(); i ++)
    {
        t_delta[t_orderrho[i].id] = maxdist;
        for(int j = 0; j < i; j ++)
        {
            if(t_dist[t_orderrho[i].id][t_orderrho[j].id] < t_delta[t_orderrho[i].id])
            {
                t_delta[t_orderrho[i].id] = t_dist[t_orderrho[i].id][t_orderrho[j].id];
                t_neighbor[t_orderrho[i].id] = t_orderrho[j].id;
            }
        }
    }
    t_delta[t_orderrho[0].id] = maxdist;
}
```

我们设定阈值为 1/2 的最大密度值与 1/8 的最大距离值，然后根据这个标准筛选出中心点。

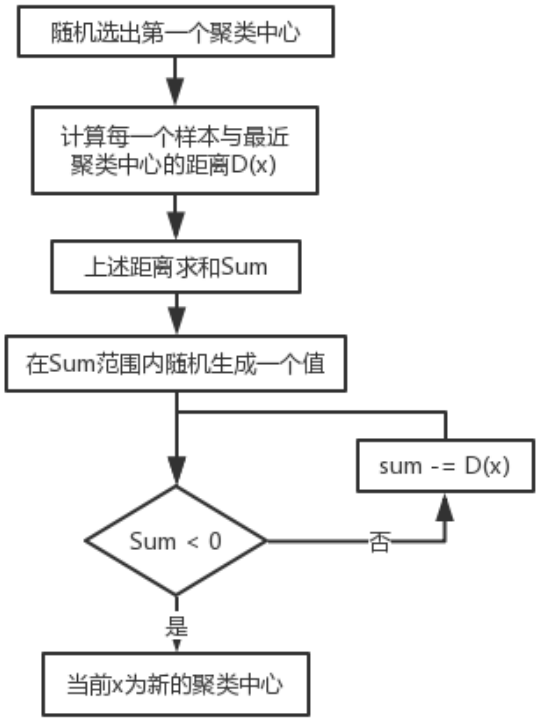
```
double t_rhoTH = t_maxrho / 2;
double t_deltaTH = maxdist / 8;

t_clusterNum = 0;
cl.resize(data.size(), -1);
icenter_of_class.clear();
for(int i = 0; i < data.size(); i ++)
{
    if (t_rho[i] > t_rhoTH && t_delta[i] > t_deltaTH)
    {
        cl[i] = t_clusterNum;
        icenter_of_class.push_back(i);
        t_clusterNum++;
    }
}
```

对于其余的点聚类的法则是寻找其最近且密度更大的邻居点，两个点所属的聚类应该一致。

```
for(int i = 0; i < data.size(); i ++)
{
    if(cl[t_orderrho[i].id] == -1)
    {
        cl[t_orderrho[i].id] = cl[t_neighbor[t_orderrho[i].id]];
    }
}


for(int i = 0;i < data.size(); i ++)
{
    data[i].label = cl[i];
}
for(int i = 0;i < icenter_of_class.size(); i ++)
{
    int centerlDofclsl = icenter_of_class[i];
    data[centerlDofclsl].center = true;
}
```

4. 创新点&优化

1. 在实现 K-means 算法的时候，实现了 K-means++的功能。也就是对于初始聚类中心的赋值遵从以下的原则：每次尽量选择离已确定聚类中心较远的随机点作为新的聚类中心，算法的流程如下：



3. 在编写谱聚类的程序时通过自己化简一些公式来避免了 C++内低效的矩阵计算：

$$L_{sym} = I - D^{-\frac{1}{2}}WD^{-\frac{1}{2}}$$

$$= I - D^{-\frac{1}{2}}(D - L)D^{-\frac{1}{2}}$$

$$= I - \left(D^{\frac{1}{2}} + D^{-\frac{1}{2}}L\right)D^{-\frac{1}{2}}$$

$$= I - I + D^{-\frac{1}{2}}LD^{-\frac{1}{2}}$$

$$= D^{-\frac{1}{2}}LD^{-\frac{1}{2}}$$

$$= \frac{L_{ij}}{\sqrt{d_i \times d_j}}$$

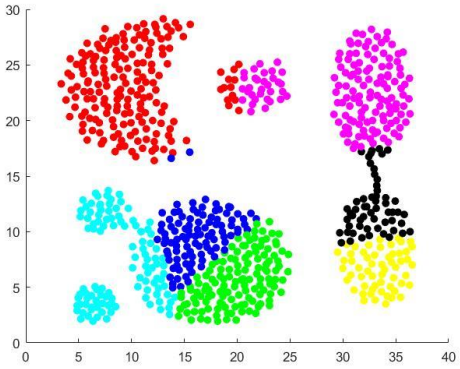
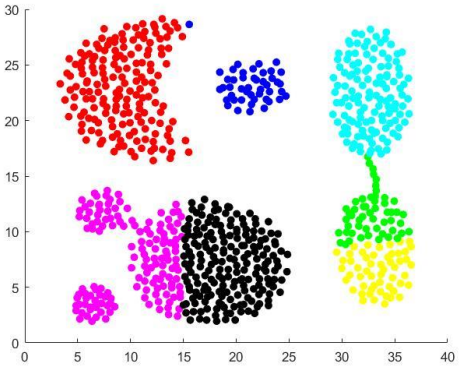
三、 实验结果及分析

1. 实验结果展示示例

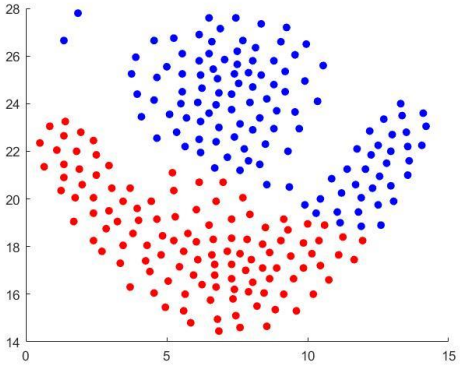
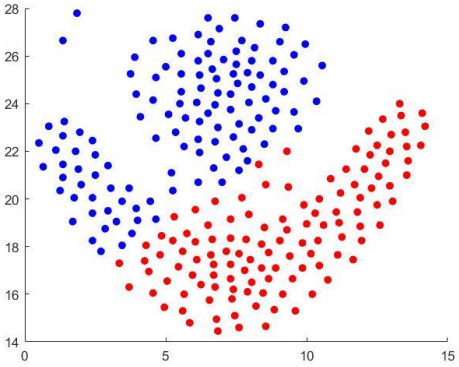
K-means

从理论上来分析，K-means 算法的效果是非常依赖于其随机初始化的过程的，因此没一个样本我都是用 K-means 算法分析了两次

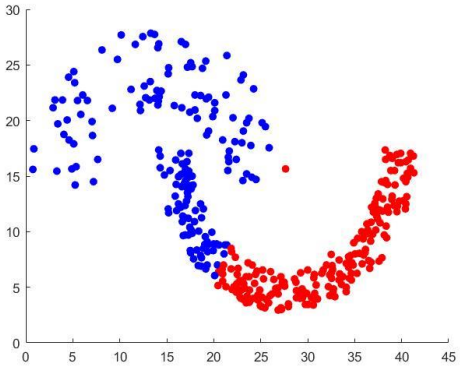
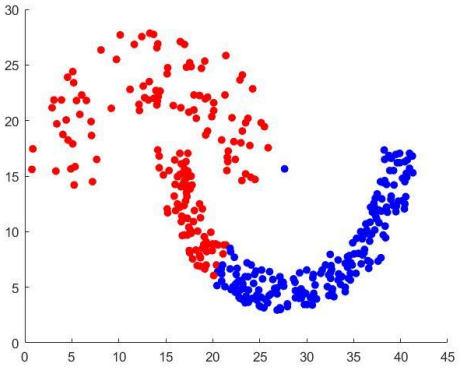
1. Aggregation_cluster=7

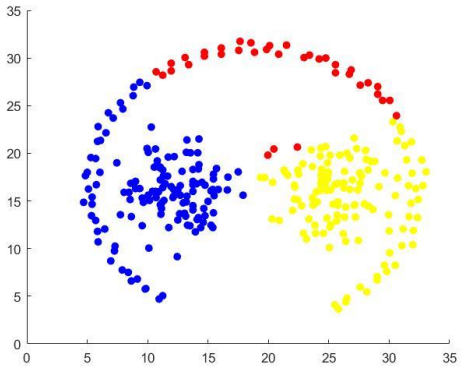
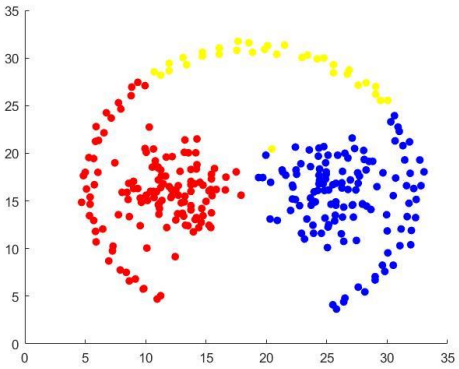
2. flame_cluster=2

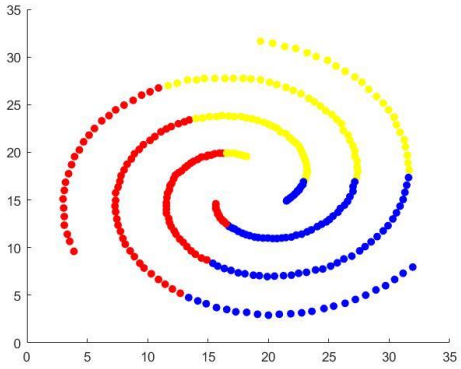
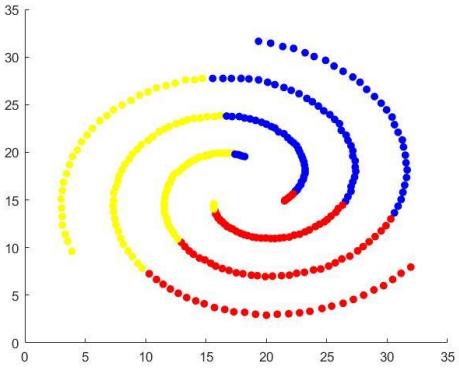
3. Jain_cluster=2

4. Pathbased_cluster=3

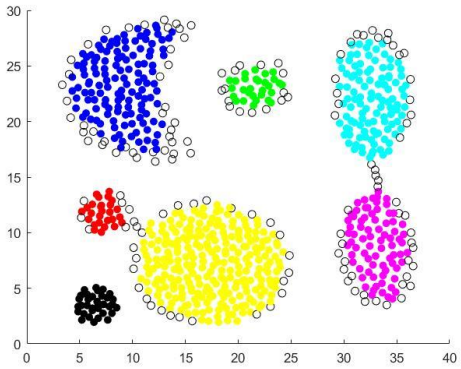



5. Spiral_cluster=3

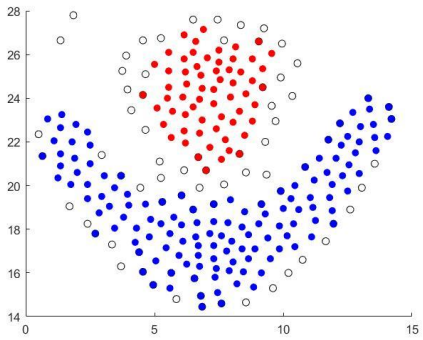



DBSCAN

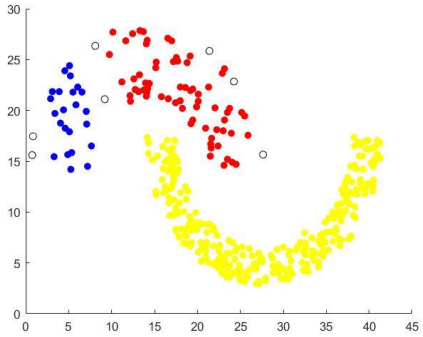
1. Aggregation_cluster=7



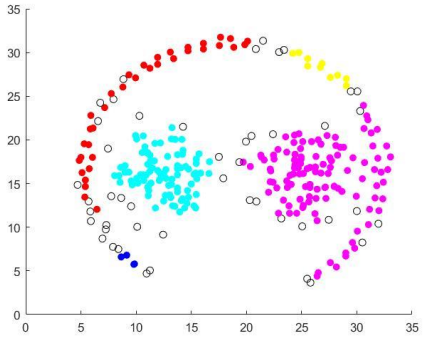
2. flame_cluster=2



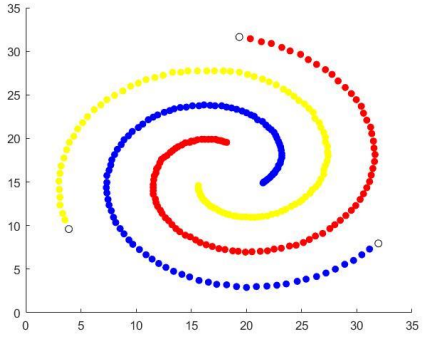
3. Jain_cluster=2



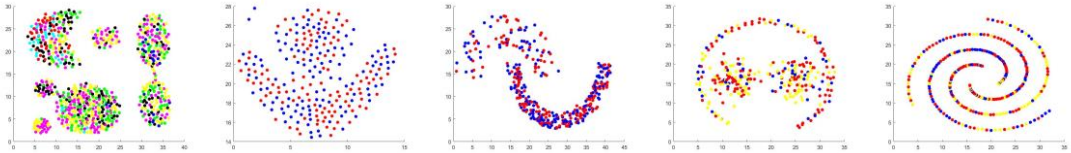
4. Pathbased_cluster=3



5. Spiral_cluster=3



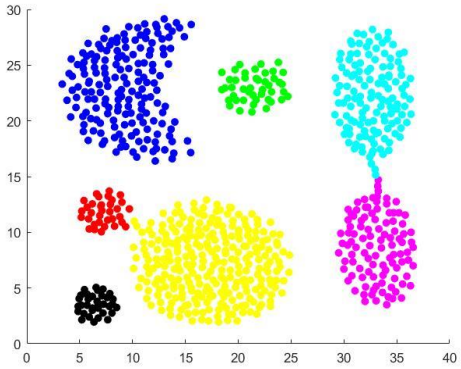
谱聚类



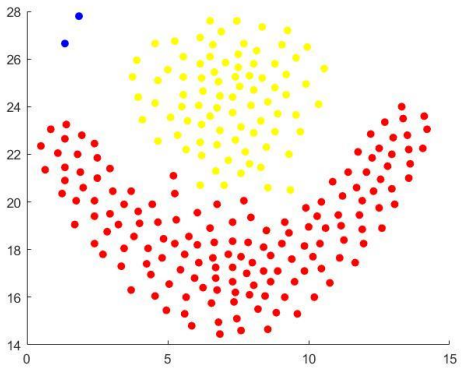
显然什么地方出了错的，但是很不幸的是我并没有找到所以也不好意思放大图来分析了。谱聚类是很好的算法，可惜我不是很好的程序员。

DenPeak

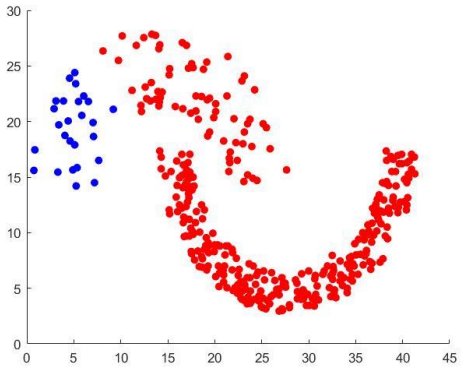
1. Aggregation_cluster=7



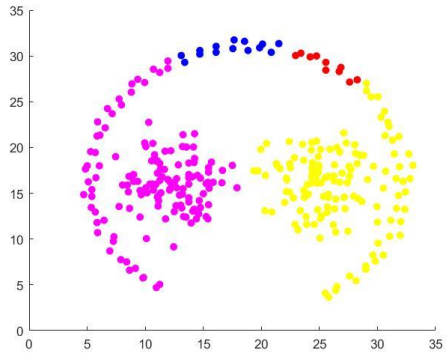
2. flame_cluster=2



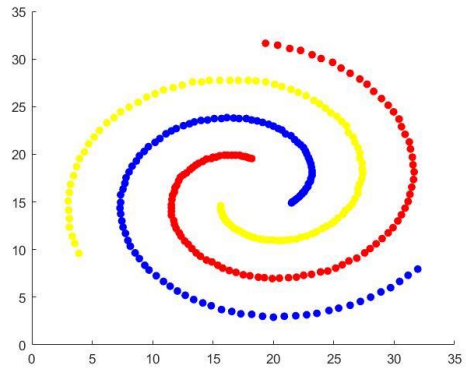
3. Jain_cluster=2



4. Pathbased_cluster=3



5. Spiral_cluster=3



手写数据集 Mfeat

如何处理这个高维的数据集的确十分棘手，于是我尝试着在 K-means 上做一些修改，直接读入归一化后的所有特征进行高纬度的聚类。

下图是归一化特征数据的代码。

```
void norm()
{
    double max,min;
    for(int i = 0; i < 649; i ++)
    {
        max = V0[0].ch[i];
        min = max;
        for(int j = 0 ; j < 2000; j ++)
        {
            if(V0[j].ch[i] < min) min = V0[j].ch[i];
            if(V0[j].ch[i] > max) max = V0[j].ch[i];
        }
        double tmp_num = max - min;
        for(int j = 0 ; j < 2000; j ++)
        {
            V0[j].ch[i] = (V0[j].ch[i] - min) * 100 / tmp_num;
        }
    }
}
```

除此以外 K-means 的距离也修正为高纬度的欧氏距离。

考虑到针对这个数据集要求聚类结果有 10 个维度加上我也初步假设手写数据集的特征是线性可分的，所以打算尝试实现高维的 K-means。

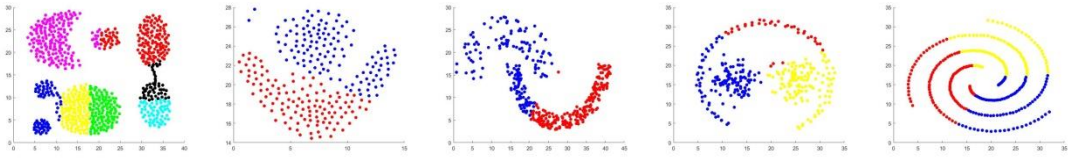
非常可惜的是效果也并不好…

```
>> ClusteringMeasure(classid, KMeans_Mfeat)

ans =

    0.1005    0.0005    0.1005
```

2. 评测指标展示即分析

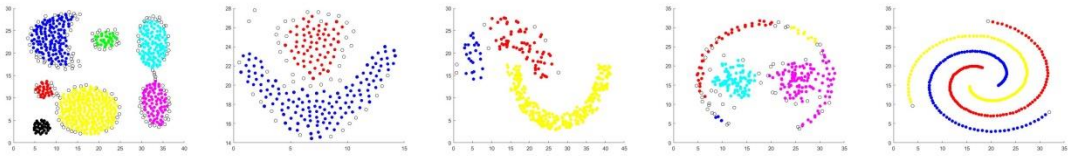
在这里我们分别进行横向和纵向的比较，从而得出结论。

首先看 K-means 的结果，如下图：



我们已经知道 K-means 针对的是线性可分的数据集，由于这次给出的五个热工数据集都是非线性可分的，因此可以说 K-means 算法的表现并不是很好，同时在之前我们也发现了 K-means 划分聚类的结果是和类中心的初始化有很大的关系的，因此 K-means 的结果也并不是很稳定。但是值得一提的是由于 K-means 算法制定了 k 的大小，因此在划分聚类的时候会严格按照我们设定的 k 来划分，省去了调参的需要并且在聚类的图形一致的情况下一般都可以完成比较不错的划分。

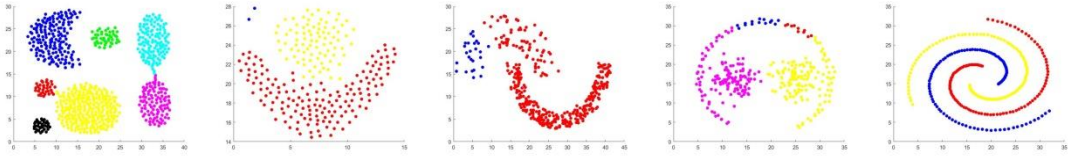
然后是 DBSCAN 的运行结果：



这部分的 DBSCAN 算法是没有任何优化的，可以看出在五个数据集的表现都比较能让人满意，特别是前两个和最后一个数据集，都向我们展示了 DBSCAN 作为基于密度的聚类，在处理非线性可分数据集时强大的功能。但是我们在实际使用中是可以发现，DBSCAN 的结果非常依赖参数的设置，不同的参数可以使 DBSCAN 有不同的敏感性。因此针对每一个数据集我们都要反复的调节两个参数。同时我们也留意到 DBSCAN 要求数据集的密度分布大致相同。也就是说如果不同聚类的内部密度有着比较大的差异时（如第三个数据集），DBSCAN 的表现并不是很好，并且是没有办法通过修正参数来达到想要的效果的。同时 DBSCAN 的聚类个数也是由算法决定的。

同时 DBSCAN 算法是会产生噪音点的，并且初始的算法是没有办法处理他们的。一种可行的解决方案是将噪音点归入与其距离最近的簇中。

Density Peak 算法的运行结果如下：



可以说 Density Peak 是最让我惊喜的算法了。第一个原因是这个算法机会只要考虑一个参数，也就是计算 dc 时使用到的最大距离百分比的值，通过调整这个值我们可以决定在划分剩余点的时候算法对距离的敏感程度。第二个原因是聚类的效果真的特别好，不仅达到了 DBSCAN 的精度，同时也自动处理了噪音点的问题。

同时从理论上来说，Density Peak 算法应该是可以适应不同密度的聚类的，也就是算法是允许聚类有一个密度的梯度的。这个性质从一定程度上可以解决 DBSCAN 没办法很好的处理不同密度的聚类的问题。但是很致命的是由于 Density Peak 在划分非中心点时遵从的是划分入密度比当前点大并且最近的聚类中，因此上图的第 3、4 个例子我们可以明显的看出，在同一个聚类密度由高变低再变高的情况发生时，Density Peak 算法会将他们切断成两个聚类。不过这个问题已经算是 Density Peak 算法原理上的不足了。

最后纵向的来分析我们实现的聚类算法，他们之间并没有办法区分孰优孰劣。更重要的还是遇到了什么样具体的问题就使用与之相对的具体的方法。可以说在划分线性可分数据及上 K-means 是完全可以胜任的，但是遇到不规则数据集就会使用到 DBSCAN 或者 Density Peak，而这两者在遇到不同密度的聚类是也有着不同的表现。