目录

[1 算法介绍 3](#_Toc534190410)

[1.1 K-means算法 3](#_Toc534190411)

[1.1.1 算法介绍 3](#_Toc534190412)

[1.1.2 算法实现 3](#_Toc534190413)

[1.2 层次聚类算法 5](#_Toc534190414)

[1.2.1 算法介绍 5](#_Toc534190415)

[1.2.2 算法实现 5](#_Toc534190416)

[1.3 密度聚类算法 6](#_Toc534190417)

[1.3.1 算法介绍 6](#_Toc534190418)

[1.3.2 算法实现 7](#_Toc534190419)

[1.4 聚类性能评估 8](#_Toc534190420)

[2实验一 Plants Data Set 9](#_Toc534190421)

[2.1数据理解 9](#_Toc534190422)

[2.2数据预处理 9](#_Toc534190423)

[2.3 实验结果 10](#_Toc534190424)

[2.3.1 K-means部分 10](#_Toc534190425)

[2.3.2 层次聚类部分 12](#_Toc534190426)

[2.3.2 密度聚类部分 12](#_Toc534190427)

[2.4 实验总结 13](#_Toc534190428)

[3 实验二 Sales\_Transactions\_Dataset\_Weekly 14](#_Toc534190429)

[3.1 数据理解 14](#_Toc534190430)

[3.2 数据预处理 14](#_Toc534190431)

[3.3 实验结果 15](#_Toc534190432)

[3.3.1 K-means部分 15](#_Toc534190433)

[3.3.2 层次聚类部分 16](#_Toc534190434)

[3.3.2 密度聚类部分 17](#_Toc534190435)

[3.4 实验总结 17](#_Toc534190436)

[4 实验三 Synthetic Control Chart Time Series 18](#_Toc534190437)

[4.1 数据理解 18](#_Toc534190438)

[4.2 数据预处理 19](#_Toc534190439)

[4.3 实验结果 20](#_Toc534190440)

[4.3.1 K-means部分 20](#_Toc534190441)

[4.3.2 层次聚类部分 21](#_Toc534190442)

[4.3.3 密度聚类部分 21](#_Toc534190443)

[4 实验总结 23](#_Toc534190444)

[参考文献 **错误!未定义书签。**](#_Toc534190445)

# 1 算法介绍

## K-means算法

### 算法介绍

k-means 算法接受输入量 k ；然后将n个数据对象划分为 k个聚类以便使得所获得的聚类满足：同一聚类中的对象相似度较高；而不同聚类中的对象相似度较小。聚类相似度是利用各聚类中对象的均值所获得一个“中心对象”来进行计算的。

k-means 算法的工作过程说明如下：首先从n个数据对象任意选择 k 个对象作为初始聚类中心；而对于所剩下其它对象，则根据它们与这些聚类中心的相似度（距离），分别将它们分配给与其最相似的（聚类中心所代表的）聚类；然后再计算每个所获新聚类的聚类中心（该聚类中所有对象的均值）；不断重复这一过程直到评价函数开始收敛为止。实验中采用均方差作为标准测度函数。k个聚类具有以下特点：各聚类本身尽可能的紧凑，而各聚类之间尽可能的分开。

算法的时间复杂度上界为O(n\*k\*t)，其中n是样本数，k是聚类数，t是迭代次数。

### 算法实现

K-means.py

1. **def** NewNode(Cluster,data):                       #用于产生新的聚类中心
2. **global** num                                   #全局变量的声明
3. num=num+1
4. New\_cluster=[]
5. **for** i **in** range(len(data)):                   # 聚类
6. dis\_mix = []
7. **for** j **in** range(len(Cluster)):
8. dis\_mix.append(dis(data[i], Cluster[j]))
9. labels\_super[i] = np.argsort(dis\_mix)[0]  # 确定簇标记，并进行划分,将dis\_mix中的元素从小到大排列，提取其对应的索引
10. **print**('经过' + str(num) + '次迭代，聚类效果ARI='+str(metrics.adjusted\_rand\_score(labels\_true=labels\_true, labels\_pred=labels\_super)))
11. # plt.scatter(data[:, 0], data[:, 1], c=labels\_super)
12. # plt.show()
13. **for** i **in** range(len(Cluster)):               # 聚类中心点迭代
14. labels\_super1 = np.array(labels\_super)  # 将列表转化为矩阵
15. index = np.argwhere(labels\_super1 == i)
16. New\_cluster.append(SumM(index,data))    # 生成6个新聚类中心
17. **return** New\_cluster
19. **def** Kmeans(K,data):
20. """
21. K均值算法的具体实现，根据西瓜书上的伪代码
22. :param K: 聚类个数
23. :param data: 数据
24. :return: 每个数据的簇标记
25. """
26. # plt.scatter(data[:, 0], data[:, 1], c=labels\_super)
27. # plt.show()
28. First\_C =[]       #初始聚类点
29. C=[random.randint (0,len(data)-1)**for** \_ **in** range(K)]    #生成初始聚类中心点
30. **for** i **in** range(len(C)):
31. First\_C.append(data[C[i]])
32. New\_cluster1=NewNode(First\_C,data)
33. err=np.array(New\_cluster1) -np.array(First\_C)
34. Jud=1           #用于判断循环是否继续进行
35. **while** Jud:
36. New\_cluster2=NewNode(New\_cluster1,data)
37. err=np.array(New\_cluster2) -np.array(New\_cluster1)
38. **if** (np.array(New\_cluster2) ==np.array(New\_cluster1)).all():
39. Jud=0
40. **else**:
41. Jud=1
42. New\_cluster1=New\_cluster2
43. **print**('经过'+str(num)+'次迭代，聚类完成')
44. **print**(labels\_super)
45. **return** labels\_super

## 1.2 层次聚类算法

### 1.2.1 算法介绍

层次聚类，是一种很直观的算法。顾名思义就是要一层一层地进行聚类，可以从下而上地把小的聚类合并聚集，也可以从上而下地将大的聚类进行分割。

凝聚的层次聚类是一种自底向上的策略，首先将每个对象作为一个簇，然后合并这些原子簇为越来越大的簇，直到所有的对象都在一个簇中，或者某个终结条件被满足。

算法的时间复杂度为O(n)，其中n是样本数。

### 1.2.2 算法实现

DBscan.py

1. **def** to\_cluster(data, clusterRes, pointId, clusterId, radius, minPts):
2. """
3. 判断一个点是否是核心点，若是则将它和它邻域内的所用未分配的样本点分配给一个新类
4. 若邻域内有其他核心点，重复上一个步骤，但只处理邻域内未分配的点，并且仍然是上一个步骤的类。
5. :param data: 样本集合
6. :param clusterRes: 聚类结果
7. :param pointId:  样本Id
8. :param clusterId: 类Id
9. :param radius: 半径
10. :param minPts: 最小局部密度
11. :return:  返回是否能将点PointId分配给一个类
12. """
13. points = neighbor\_points(data, pointId, radius)
14. points = points.tolist()
16. q = queue.Queue()
17. **if** len(points) < minPts:
18. clusterRes[pointId] = NOISE
19. **return** False
20. **else**:
22. clusterRes[pointId] = clusterId        #对该点进行赋值
23. **for** point **in** points:
24. **if** clusterRes[point] == UNASSIGNED:
25. q.put(point)
26. clusterRes[point] = clusterId     #对该点的密度直达进行赋类
27. **print**(clusterRes)
28. **while** **not** q.empty():                      #寻找该点的的密度相连 对队列的使用
29. neighborRes = neighbor\_points(data, q.get(), radius)
30. **if** len(neighborRes) >= minPts:                      # 核心点
31. **for** i **in** range(len(neighborRes)):
32. resultPoint = neighborRes[i]
33. **if** clusterRes[resultPoint] == UNASSIGNED:
34. q.put(resultPoint)
35. clusterRes[resultPoint] = clusterId
36. **elif** clusterRes[clusterId] == NOISE:
37. clusterRes[resultPoint] = clusterId
38. **return** True
39. **def** dbscan(data, radius, minPts):
40. """
41. 扫描整个数据集，为每个数据集打上核心点，边界点和噪声点标签的同时为
42. 样本集聚类
43. :param data: 样本集
44. :param radius: 半径
45. :param minPts:  最小局部密度
46. :return: 返回聚类结果， 类id集合
47. """
48. clusterId = 1
49. nPoints = len(data)
50. clusterRes = [UNASSIGNED] \* 600
51. **print**(clusterRes)
52. **for** pointId **in** range(nPoints):
53. **if** clusterRes[pointId] == UNASSIGNED:
54. **if** to\_cluster(data, clusterRes, pointId, clusterId, radius, minPts):
55. clusterId = clusterId + 1
56. **return** np.asarray(clusterRes), clusterId

## 1.3 密度聚类算法

### 1.3.1 算法介绍

DBSCAN是一种基于密度的聚类算法，这类密度聚类算法一般假定类别可以通过样本分布的紧密程度决定。同一类别的样本，他们之间的紧密相连的，也就是说，在该类别任意样本周围不远处一定有同类别的样本存在。

通过将紧密相连的样本划为一类，这样就得到了一个聚类类别。通过将所有各组紧密相连的样本划为各个不同的类别，则我们就得到了最终的所有聚类类别结果。

密度聚类对领域参数比较敏感。

### 1.3.2 算法实现

AGNES.py

1. #算法模型：
2. **def** AGNES(dataset, dist, k):
3. #初始化C和M
4. C = [];M = []
5. **for** i **in** range(len(dataset)):
6. Ci = []
7. Ci.append(i)
8. C.append(Ci)
9. dis\_M=dis\_matrix(dataset)
10. q = len(dataset)
11. #合并更新
12. **while** q > k:
13. x, y, min = find\_Min(dis\_M)
14. **print**(x,y,min)
15. C[x].extend(C[y])
16. C.remove(C[y])
17. **print**(len(C))
18. dis\_M=np.zeros((len(C),len(C)))
19. **for** i **in** range(len(C)):
20. **for** j **in** range(len(C)):
21. dis\_M[i][j]=dis\_M[j][i]=dist(C[i],C[j],dataset)
22. q -= 1
23. **print**(C)
24. **for** i **in** range(k):
25. **for** j **in** C[i]:
26. labels\_super[j]=i        #对标签进行赋值，确定聚类
28. **return** C

## 1.4 聚类性能评估

轮廓系数：取值范围[-1,1]同类别的样本距离越近且不同类别样本距离越远，分数越高效果越好

Calinski系数：得到的Calinski-Harabasz分数值ss越大则聚类效果越好。

ARI系数：聚类性能评估，取值[-1,1]越接近1，性能越好

# 2实验一 Plants Data Set

## 2.1数据理解

Plants Data Set数据集包含了每一种植物(种类和科属)以及它们生长的地区。数据集中总共有69个地区，主要分布在美国和加拿大。一条数据(对应于文件中的一行)包含一种植物(或者某一科属)及其在上述69个地区中的分布情况。可以这样理解，该数据集中每一条数据包含两部分内容，如下图所示。

**植物名称(科属+名称)**

**分布区域**

图1 数据格式

例如一条数据:abronia fragrans,az,co,ks,mt,ne,nm,nd,ok,sd,tx,ut,wa,wy。其中abronia fragrans是植物名称(abronia是科属，fragrans是名称)，从az一直到wy是该植物的分布区域，采用缩写形式表示，如az代表的是美国Arizona州。植物名称和分布地区用逗号隔开，各地区之间也用逗号隔开。

## 2.2数据预处理

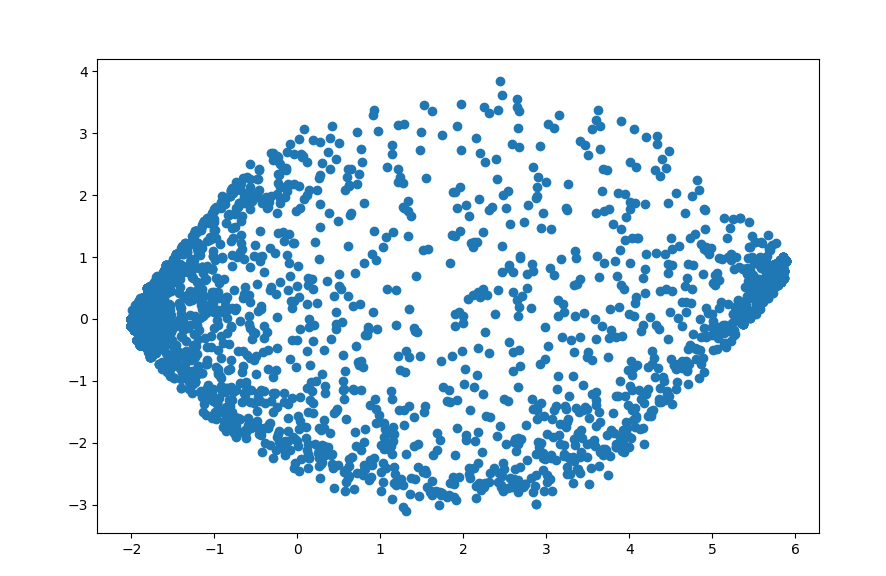
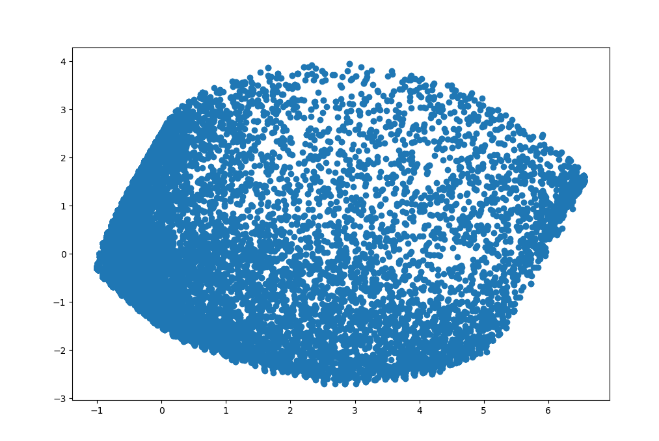
首先原数据集有30000多条数据集， 如果直接对原数据集进行聚类操作，每聚类一次所消耗的时间比较长，本实验通过预处理部分，在对结果影响不大的情况下对数据进行了缩减。例如：

➀abelmoschus,ct,dc,fl,hi,il,ky,la,md,mi,ms,nc,sc,va,pr,vi

➁abelmoschus esculentus,ct,dc,fl,il,ky,la,md,mi,ms,nc,sc,va,pr,vi

➂abelmoschus moschatus,hi,pr

上述数据中第1行给出了所有属于abelmoschus这一科属的植物的分布地区，接下来的2、3两行分别列出了属于abelmoschus科属的两种具体植物及其分布地区。我们可以看出abelmoschus的分布情况已经包含了之后两种具体植物的分布情况，并且我们通过对数据二维的分布情况进行观察，可以用科属来代替具体植物来进行聚类分析，二维分布如下：



原数据 数据预处理之后

同时该数据集由于地名是使用英文缩写来代替，这并不利于聚类，所以本实验将每条数据处理成以下数据形式。

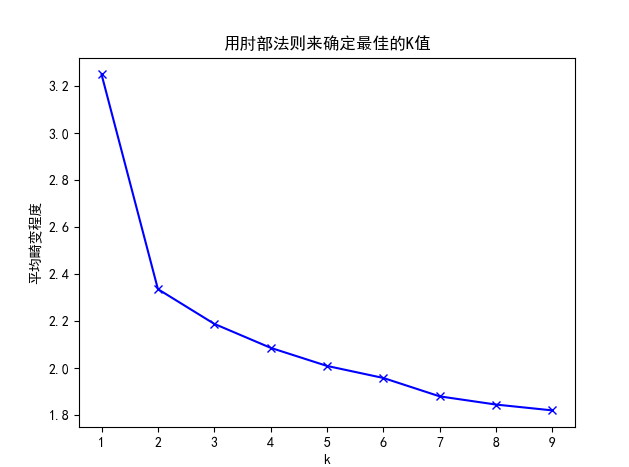
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

这是一个长度69的列表，每一位分别代表一个州，如果列表中的数字为1，则代表该植物在该州有分布，反之亦然。数据预处理之后的形式为3382\*69大小的矩阵。

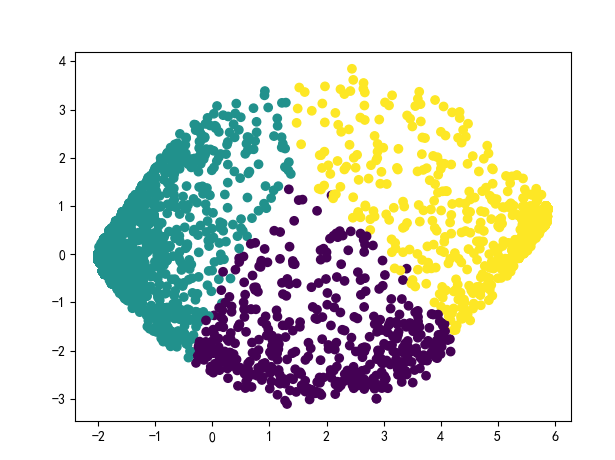
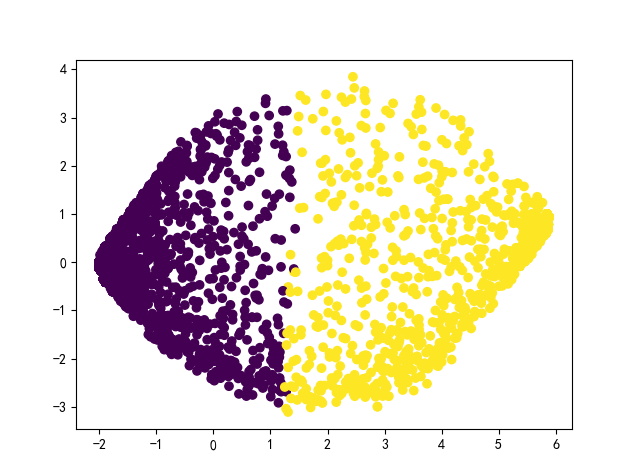
## 2.3 实验结果

### 2.3.1 K-means部分

K-means聚类部分，我们首先通过肘击原理确定最佳的K值。实验结果如下图所示：



从图中可以看出，K值从1到2是，平均畸变程度变化最大，超过2之后，平均畸变程度变化显著降低，所以肘部就是K=2。

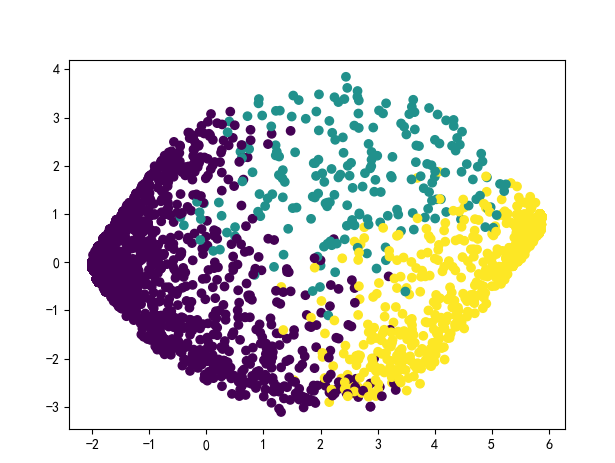
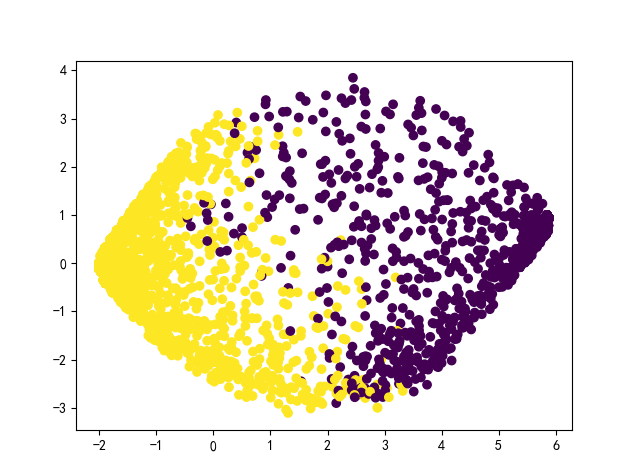


K=2 K=3

|  |  |  |  |
| --- | --- | --- | --- |
| **K** | **轮廓系数** | **Calinski系数** | **用时** |
| 2 | **0.692** | **8467.0** | **6.93s** |
| 3 | 0.647 | 7533.1 | 40.7s |
| 4 | 0.603 | 7465.6 | 38.8s |

### 2.3.2 层次聚类部分

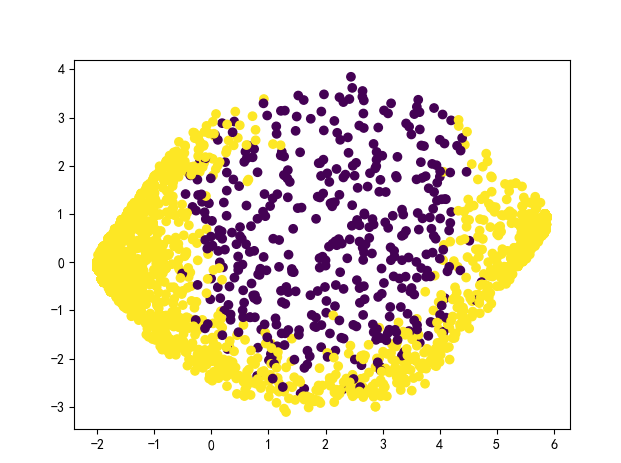
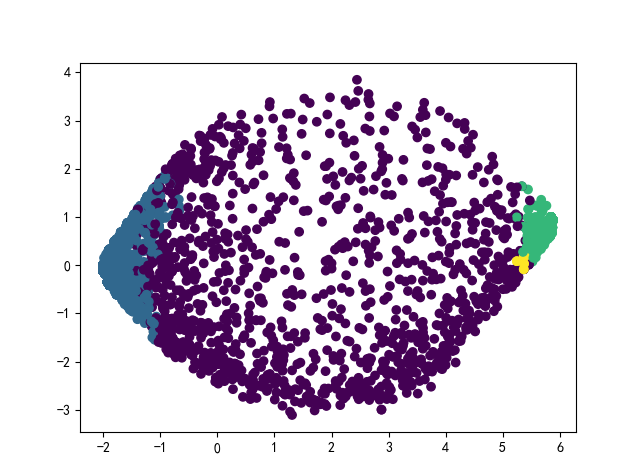
通过K-means聚类中的肘击定理我们知道该数据集最佳的K值为2，所以在层次聚类中本实验参数设置以及实验数据如下：



K=2 K=3

|  |  |  |  |
| --- | --- | --- | --- |
| **K** | **轮廓系数** | **Calinski系数** | **用时** |
| 2 | **0.666** | **7075.8** | 25.9s |
| 3 | 0.585 | 4462.6 | 15.3s |
| 4 | 0.536 | 5122.6 | **14.7s** |

### 2.3.2 密度聚类部分



2-50 3-50

密度聚类：

|  |  |  |  |
| --- | --- | --- | --- |
| **领域** | **轮廓系数** | **Calinski系数** | **用时** |
| 2-10 | 0.354 | 1620.4 | **132.3**s |
| 2-50 | **0.399** | **1996.5** | 136.9s |
| 3-50 | 0.291 | 300.6 | 150.0s |

密度聚类效果较差。

## 2.4 实验总结

对三种聚类算法在本数据上进行对比：

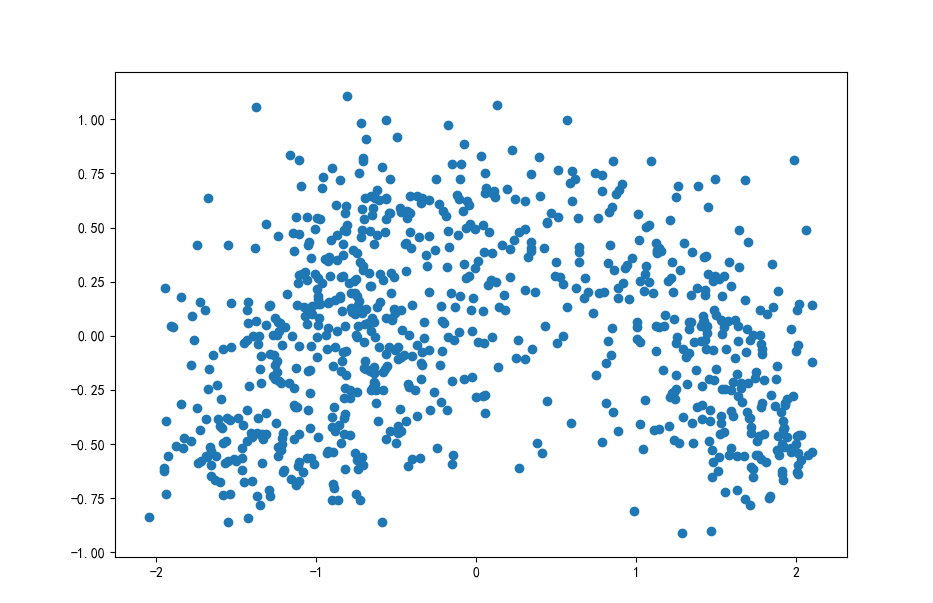
|  |  |  |  |
| --- | --- | --- | --- |
| **聚类算法** | **轮廓系数** | **Calinski系数** | **用时** |
| K-means（k=2） | **0.692** | **8467.0** | **6.93s** |
| 层次聚类（k=2） | 0.666 | 7075.8 | 25.9s |
| 密度聚类（3,50） | 0.291 | 300.6 | 150.0s |

可知，在数据较大的情况下，K-mean算法聚类的效果较好。

# 3 实验二 Sales\_Transactions\_Dataset\_Weekly

## 数据理解

Sales\_Transactions\_Dataset\_Weekly数据集是在52周之内，800多种商品每周的购买数量，同时本数据集也提供了标准化值。由于不同商品之间由于各自性质的不同，所以不能简单的通过件数来进行聚类，本实验我们采用了标准化值来进行聚类。通过聚类来找到商品之间是否存在内在的联系。首先本实验通过PCA降维将数据直观的显示在二维平面上，其分布情况如下图所示：



## 3.2 数据预处理

本实验通过标准化值进行聚类，所以我们要对实验所给的CSV文件进行预处理，处理之后的数据存储在矩阵中，之后进行聚类操作。

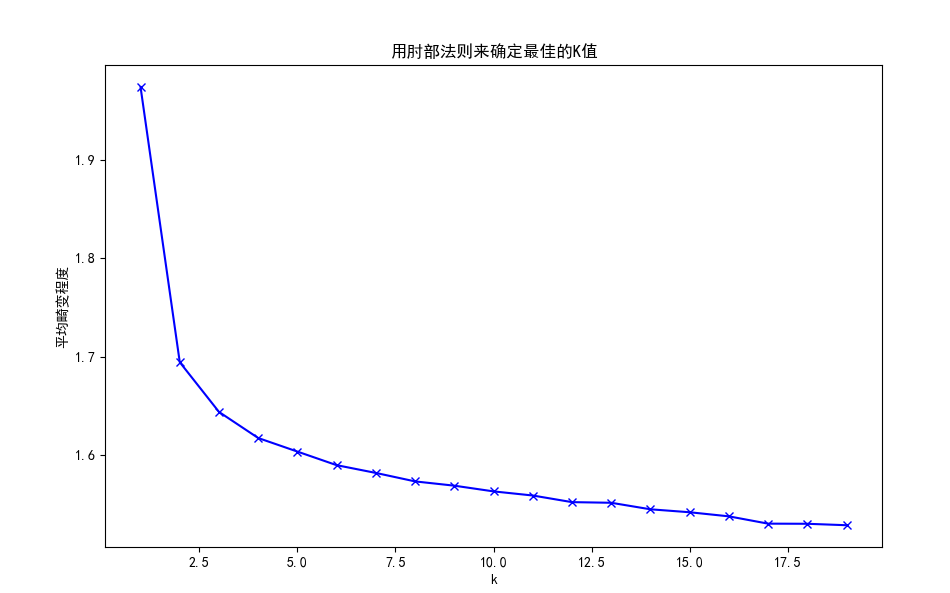
数据处理：

1. data\_path='Sales\_Transactions\_Dataset\_Weekly.csv'
2. **def** LoadTxt():
3. """
4. 数据导入
5. :return:矩阵
6. """
7. data=np.loadtxt(data\_path,delimiter=",",skiprows=1,usecols=range(55,107))
8. data=np.array(data)
9. **return** data

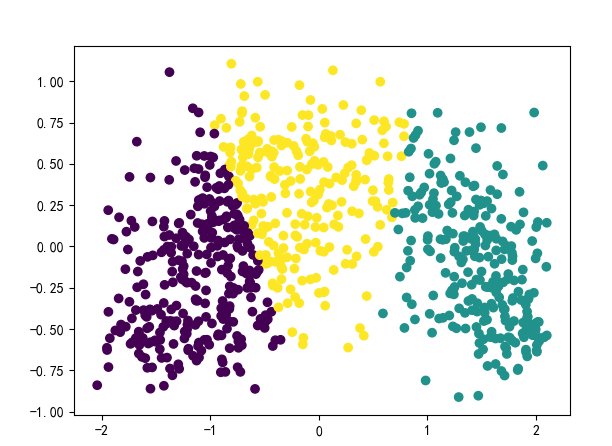
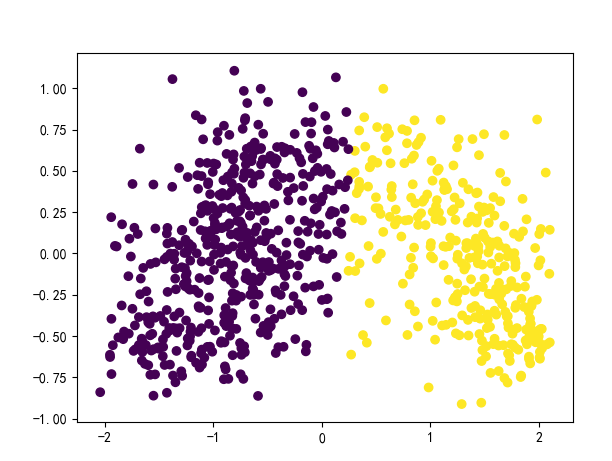
## 3.3 实验结果

### 3.3.1 K-means部分

在K-means聚类部分，我们首先通过肘击原理确定最佳的K值。实验结果如下图所示：



从图中可以看出，K值从1到2是，平均畸变程度变化最大，超过2之后，平均畸变程度变化显著降低，所以肘部就是K=2。

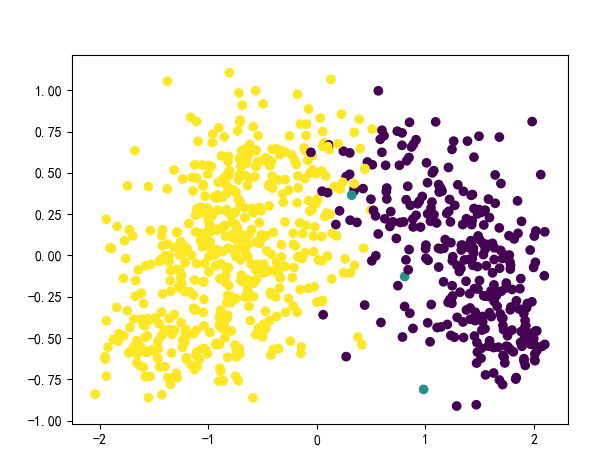
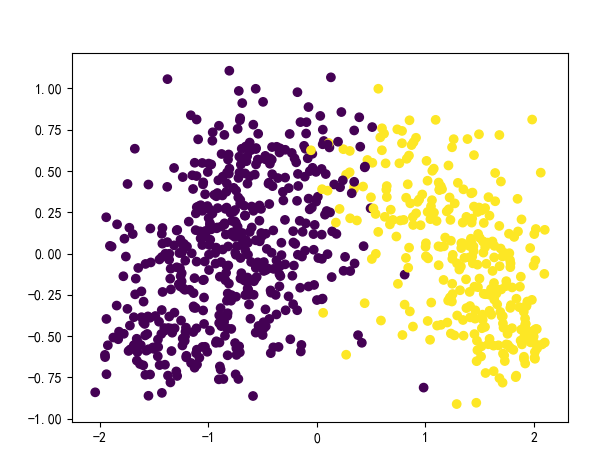


K=2 k=3

本数据集我们将采用轮廓系数、Calinski系数、用时来衡量聚类的效果：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **K** | **迭代次数** | **轮廓系数** | **Calinski系数** | **用时** |
| 2 | **7** | **0.599** | **1970.3** | **4.56s** |
| 3 | 25 | 0.457 | 1762.7 | 5.20s |
| 4 | 13 | 0.416 | 1693.7 | 6.67**s** |

### 3.3.2 层次聚类部分

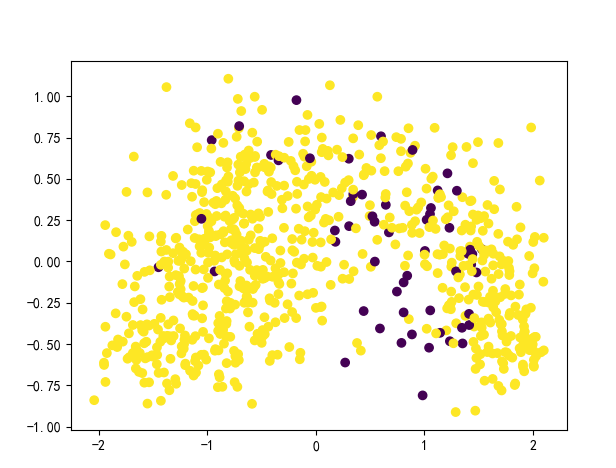
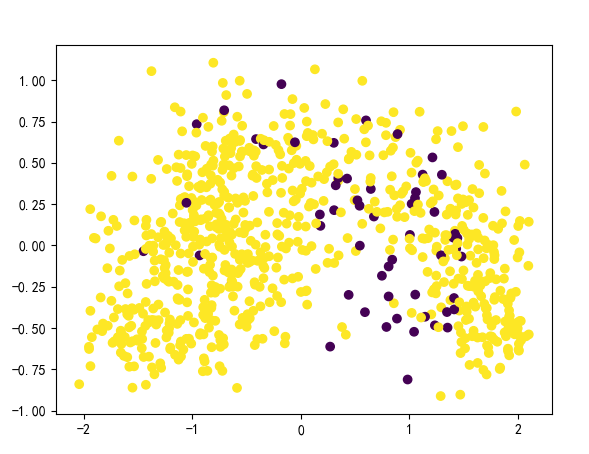


K=2 k=3

层次聚类实验数据：

|  |  |  |  |
| --- | --- | --- | --- |
| **K** | **轮廓系数** | **Calinski系数** | **用时** |
| 2 | **0.585** | **1831.9** | **2.56s** |
| 3 | 0.328 | 939.4 | 13.35s |
| 4 | 0.260 | 629.3 | 15.38s |

### 3.3.2 密度聚类部分



(2,5) (2,10)

|  |  |  |  |
| --- | --- | --- | --- |
| **领域** | **轮廓系数** | **Calinski系数** | **用时** |
| 2-5 | 0.140 | 10.84 | 14.8s |
| 2-10 | 0.140 | 10.84 | 34.5**s** |

## 3.4 实验总结

对三种聚类算法在本数据上进行对比：

|  |  |  |  |
| --- | --- | --- | --- |
| **聚类算法** | **轮廓系数** | **Calinski系数** | **用时** |
| K-means（k=2） | **0.599** | **1970.3** | 4.56s |
| 层次聚类（k=2） | 0.585 | 1831.9 | **2.56s** |
| 密度聚类（2,5） | 0.140 | 10.84 | 14.8s |

可知，在该数据集中，K-mean算法聚类的效果较好。

# 4 实验三 Synthetic Control Chart Time Series

## 4.1 数据理解

本实验采用的实验数据集为Synthetic Control Chart Time Series，这个数据集经常被用于研究时间序列聚类算法中，数据集包含600个时间序列，每个时间序列由60个时间点构成,这组数据包含6类，每个类包含100个时间序列，这6类分别代表了一种时间序列变化趋势。

1-100数据为标准时间序列

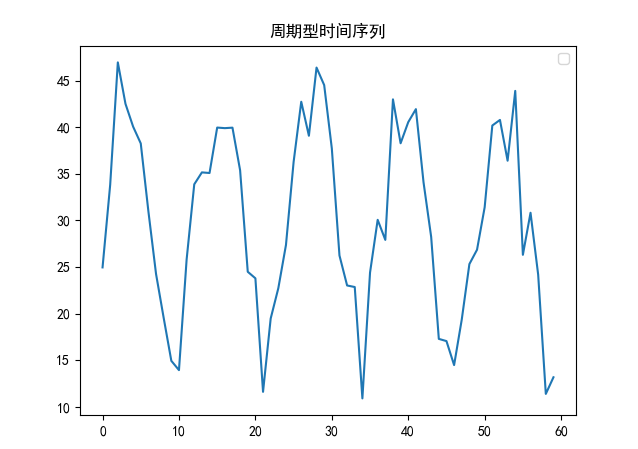
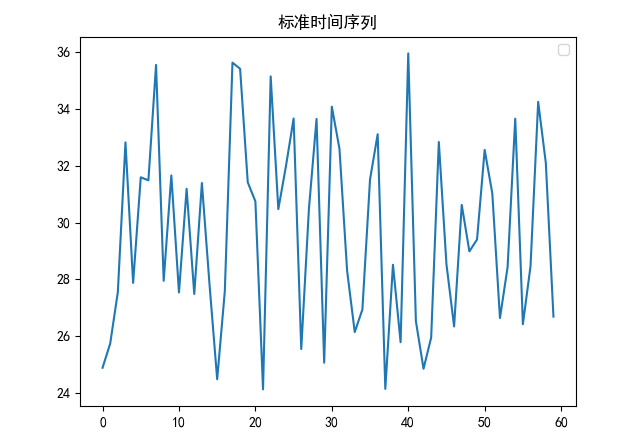
101-200数据为周期型时间序列

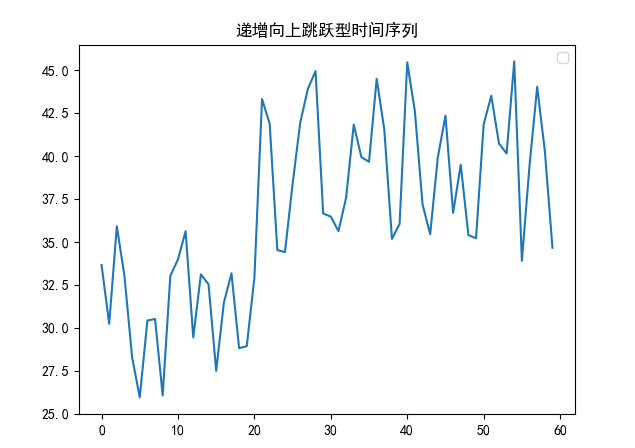
201-300数据为递增趋势的时间序列

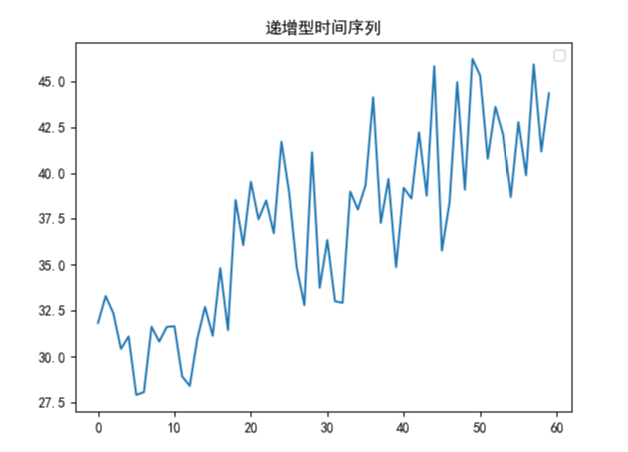
301-400数据为递减趋势的时间序列

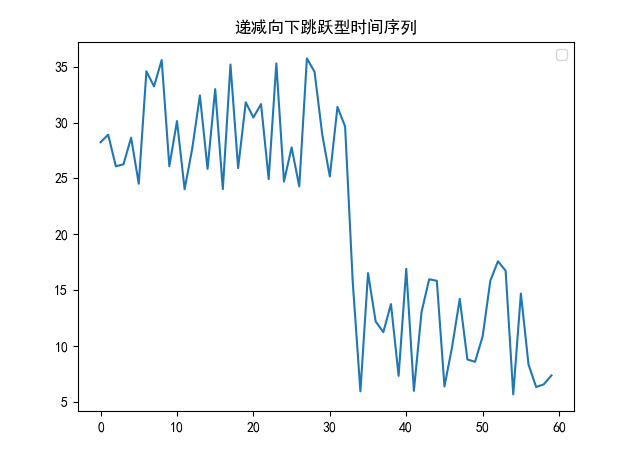
401-500数据为递增趋势时间序列，并且包含向上跳跃点

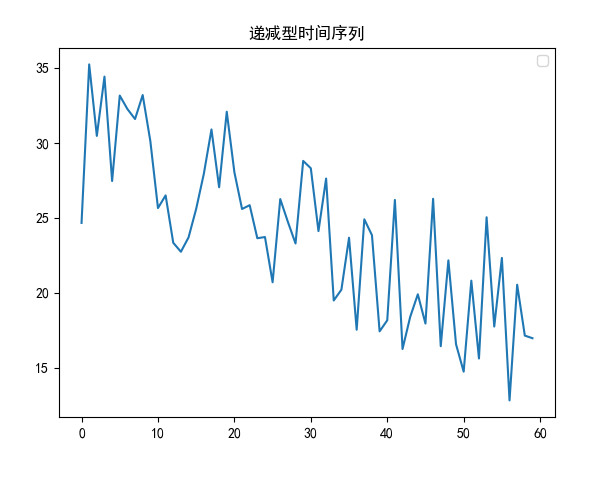
501-600数据为递减趋势时间序列，并且包含向下跳跃点



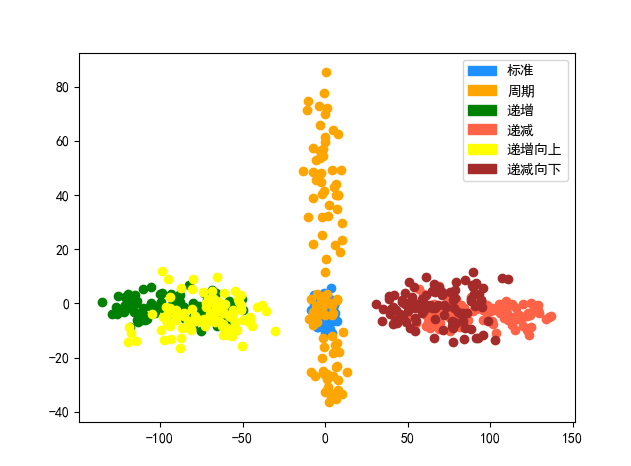








数据降维之后我们将直观的看到数据的分布情况，其中标准和周期、递增和递增向上、递减和递减向下三类之间分类比较明确，而类之间却非常接近，不易分别，以下实验就是探究聚类算法对于这几种数据的聚类效果。



## 4.2 数据预处理

由于本数据集为高维的时间序列，不能直接在二维坐标系上进行显示，也就无法直观的对数据的分布进行合理的观察，所以本实验将高维的时间序列进行了预处理，使结果更加的直观。

PCA降维：

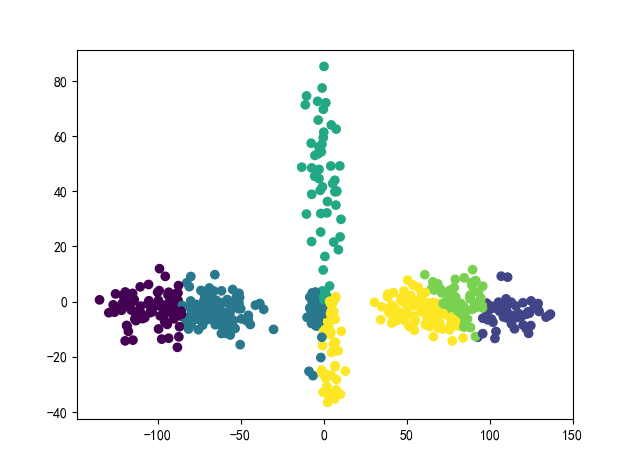
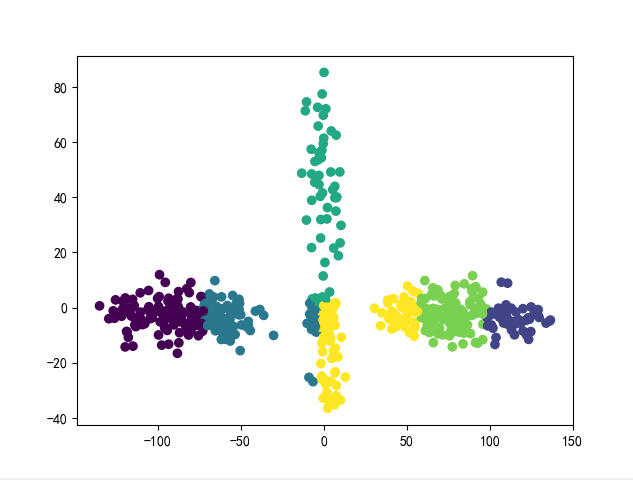
1. pca = PCA(n\_components=2)  # 进行PCA降维
2. newdata = pca.fit\_transform(data)

## 4.3 实验结果

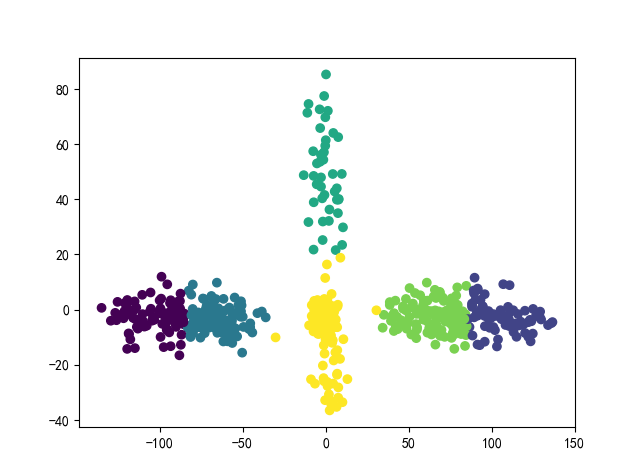
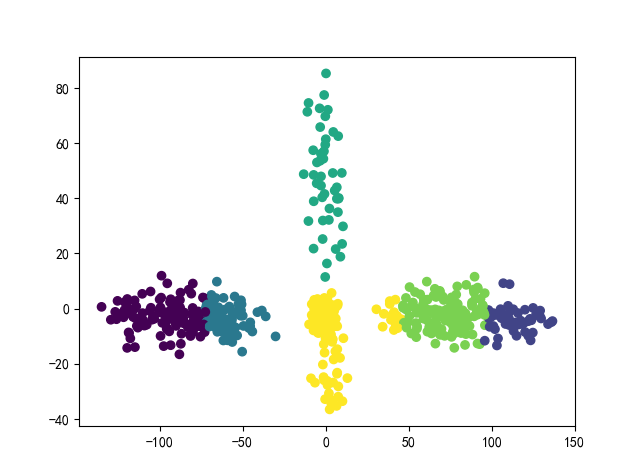
### 4.3.1 K-means部分

在K-means实验部分，本次实验经过了7次迭代，算法收敛，以下数据反映了每次迭代的ARI（ARI聚类性能评估，取值[-1,1]越接近1，性能越好）变化。通过多次实验可知K-means由于初始的聚类中心是随机的，所以每次实验的迭代次数和最后的迭代效果都是不一样的。

|  |  |
| --- | --- |
| **迭代次数** | **ARI** |
| 1 | 0.371 |
| 2 | 0.480 |
| 3 | 0.488 |
| 4 | 0.487 |
| 5 | 0.485 |
| 6 | 0.486 |
| 7 | 0.486 |

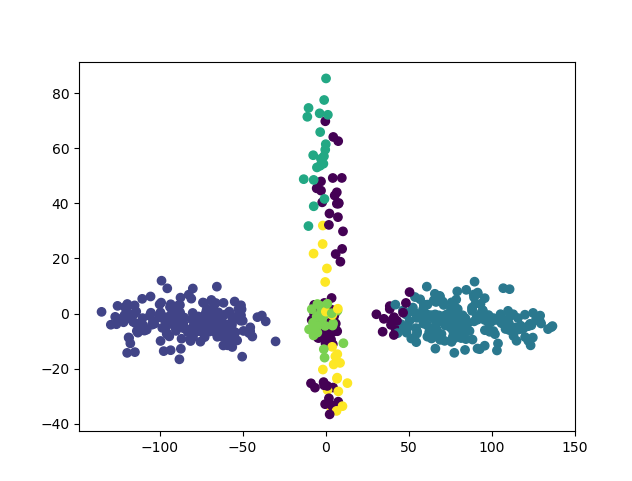
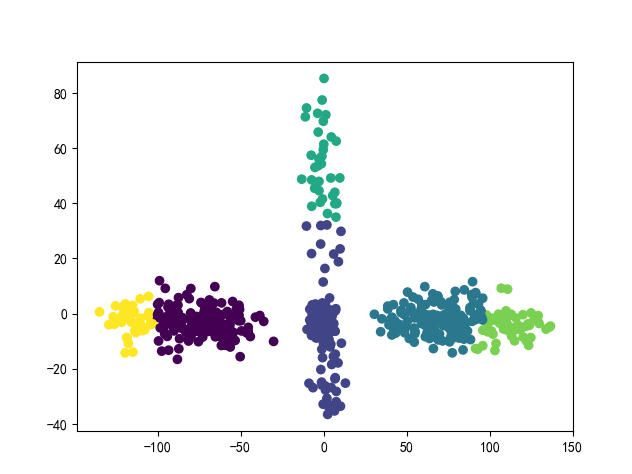
 

第1次迭代 第3次迭代



第5次迭代 第7次迭代

### 4.3.2 层次聚类部分



降维数据聚类 原始数据聚类

对降维后的数据进行聚类分析

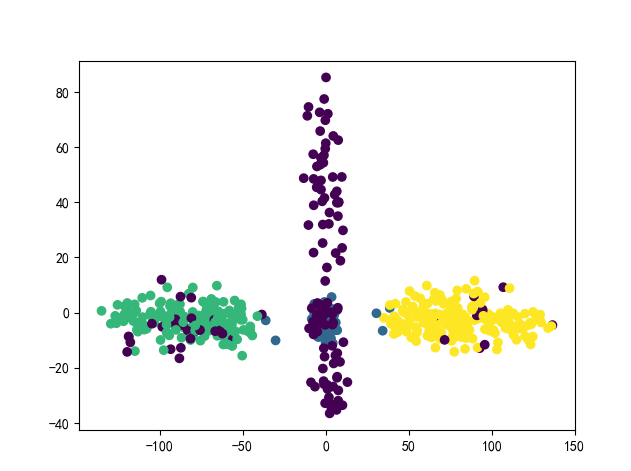
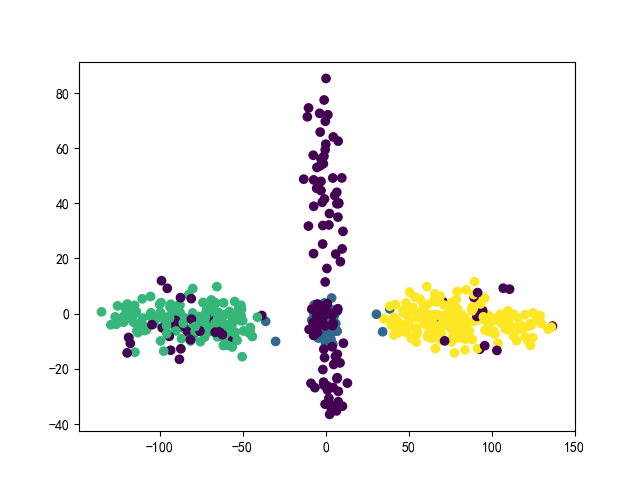
ARI:0.5361672013123161

对原始数据进行聚类分析

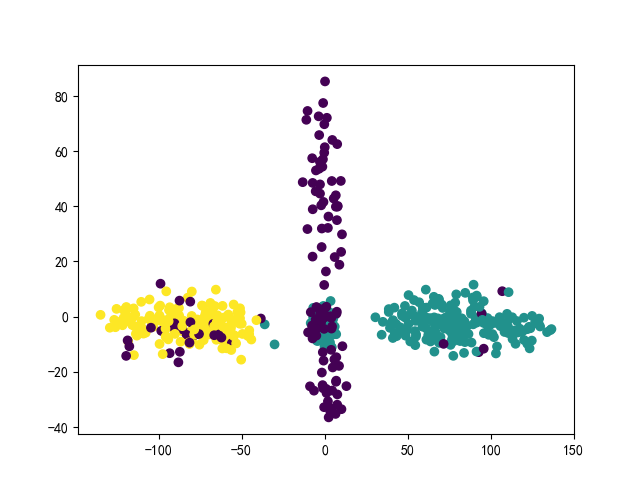
ARI:0.5373023914057349

### 4.3.3 密度聚类部分

在密度聚类部分，本实验通过多次设置领域参数，以求达到更好的聚类效果。



领域参数46.5, 50 领域参数46.9, 50



领域参数47, 50

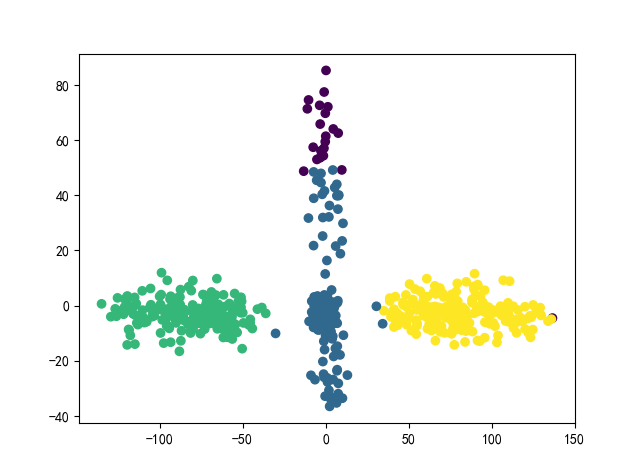
对原始数据进行聚类分析的数据：

dbscan(data, 46.5, 50) ARI:0.603315201033937

dbscan(data, 46.9, 50) ARI:0.6140718847508271

dbscan(data, 47, 50) ARI:0.429997337031337

在之后的部分我们同时也对降维数据进行了聚类，其效果如下：



领域参数20, 50

对降维后的数据进行聚类分析的数据：

dbscan(newdata, 10, 50) ARI:0.23894221391893372

dbscan(newdata, 20, 50) ARI:0.5605751861500429

dbscan(newdata, 25, 50) ARI:0.5505656168543847

## 4 实验总结

由于在本数据集中我们已经知道每一条数据集的类别所以我们可以使用ARI对数据进行聚类性能的评估，结果如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **评价标准** | **K-means** | **Agnes(层次)** | **DBscan（密度）** |
| ARI | 0.488 | 0.537 | 0.614 |

由以上实验可知，对于时间序列的数据集而言，密度聚类的效果较好，同时本实验也对PCA降维后的数据进行聚类实验，发现PCA降维后的聚类效果在三种聚类方法中结果都不如直接对原始数据进行聚类，这也反应了降维对时间序列的聚类还是存在一定影响。