



华中科技大学

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY



《数据挖掘》报告

课题名称: kMeans 与 kNN 的实现与比较

院系: 电子信息与通信学院

专业: 电子信息工程

班级: 电信 1204 班

姓名: 黄衍

学号: U201213468

1. kMeans

1.1. 原理

kMeans 算法流程如下：

24

K-Means

Algorithm 2.1 The k-means algorithm

Input: Dataset D , number clusters k

Output: Set of cluster representatives C , cluster membership vector \mathbf{m}

/* Initialize cluster representatives C */

Randomly choose k data points from D

5: Use these k points as initial set of cluster representatives C

repeat

/* Data Assignment */

Reassign points in D to closest cluster mean

Update \mathbf{m} such that m_i is cluster ID of i th point in D

10: /* Relocation of means */

Update C such that c_j is mean of points in j th cluster

until convergence of objective function $\sum_{i=1}^N (\argmin_j ||\mathbf{x}_i - \mathbf{c}_j||_2^2)$

kMeans 首先随机选取 k 个点作为聚类中心，余下的点进行择近选择，然后以聚类的均值作为新的聚类中心，作为初始迭代。此后的迭代则对每个点，对 k 个样本中心点（聚类中心）择近选择，再计算样本中心点。直到样本中心点不再变化，算法结束。

注意在实际计算时，由于样本坐标采用浮点表示，样本中心点不再变化的判断方法是判断新的样本中心与旧的样本中心相差是否小于浮点数的精度，在 Matlab 中表达如下：

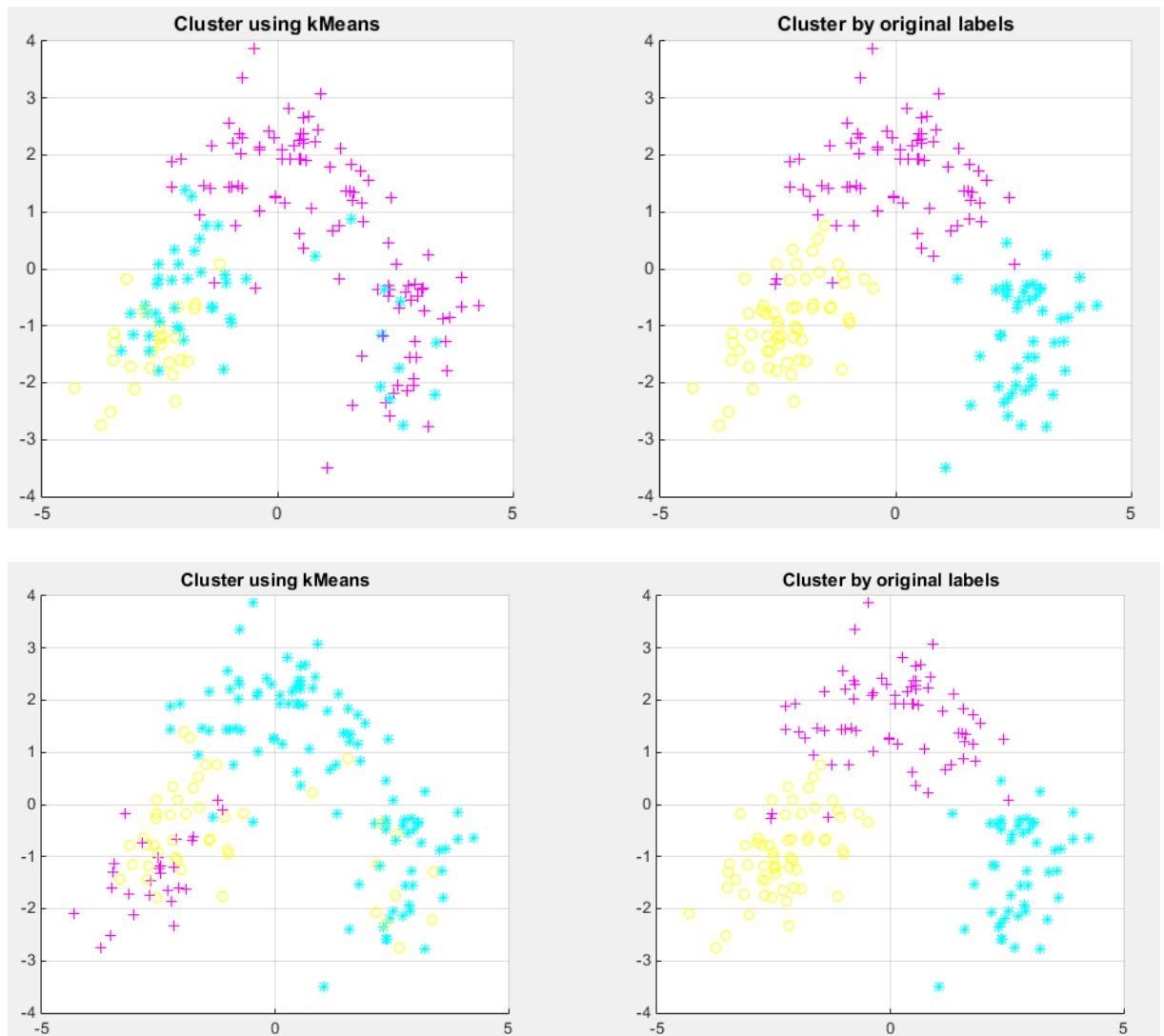
```
if abs(sum(kmeansNew - kmeans)) < eps(sum(kmeansNew))
```

kMeans 算法由于其初始化方式，带有随机性，聚类具有随机性。

1.2. 原始结果

以下是处理 UCI 的 Wine 数据的两次结果：

注意 wine 数据是 13 维的，在数据可视化之前做了 PCA 降维处理。



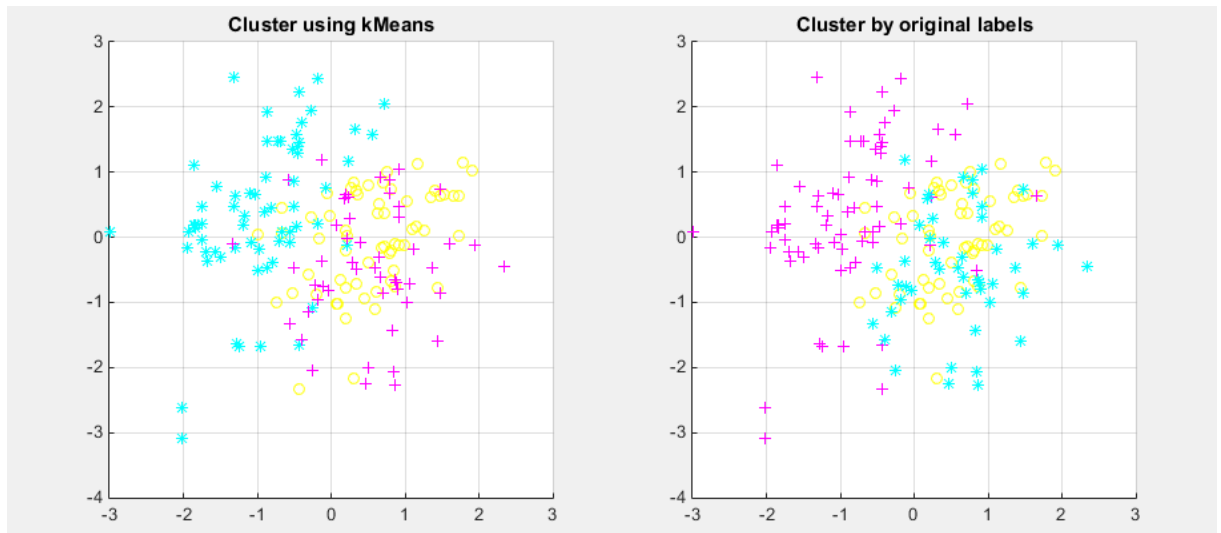
由于数据量小，聚类结果差异不明显。但是比较明显的是，由于 **kMeans** 是非监督学习，它对各个聚类的编号（Label）是随机的，如果不手动干预初始化的方式，则聚类编号极大可能与设想的编号不一致。这样，对聚类结果的解释则需要人工进行。

Wine 数据中各个维度的数据范围、方差都不同，以上结果是直接对原始数据进行 **kMeans** 得到，在实际的机器学习和数据挖掘实践中，进行诸如 **PCA** 降维、数据标准化等数据预处理有时候是很有必要的，因此，对 **Wine** 数据进行处理后再进行试验。

1.3. 数据标准化

数据标准化的意义在于，把不同范围各个属性归一化到同一个范围，使得各个属性在计算样本间距的时候具有相同的重要性（原始数据由于属性的范围不同，大范围的数据影响程度明显要高），也方便我们定义各个属性的加权重要程度，对不同的 **feature** 采取不同的权重，改进聚类结果。

下图是对 **Wine** 数据进行数据标准化并选取 5 个主成分的聚类结果：



由此可见，虽然聚类的标签编号仍然随机，但是聚类结果有了明显的改善，与原始数据聚类 Label 更加一致。同时可以看出，Wine 数据的主维度在 3 维以上，因为 2 维空间（绘图平面）不足以把各个聚类分开。

2. kNN

2.1. 原理

同为聚类算法，kNN 实质上是监督的学习，必须将数据集分为两部分——Train（训练集）和 Test（测试集），用 Train 来监督 Test 的聚类。kNN 的算法流程如下：

8.2 Description of the Algorithm

153

Algorithm 8.1 Basic kNN Algorithm

Input : D , the set of training objects, the test object, \mathbf{z} , which is a vector of attribute values, and L , the set of classes used to label the objects

Output : $c_z \in L$, the class of \mathbf{z}

foreach object $\mathbf{y} \in D$ **do**

 | Compute $d(\mathbf{z}, \mathbf{y})$, the distance between \mathbf{z} and \mathbf{y} ;

end

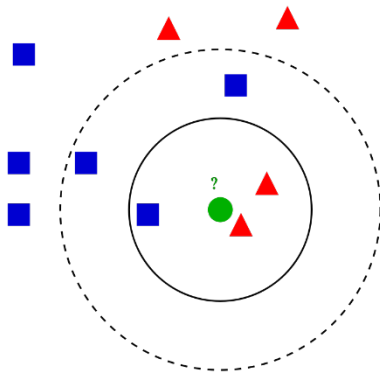
Select $N \subseteq D$, the set (neighborhood) of k closest training objects for \mathbf{z} ;

$c_z = \operatorname{argmax}_{v \in L} \sum_{y \in N} I(v = \text{class}(c_y))$;

where $I(\cdot)$ is an indicator function that returns the value 1 if its argument is true and 0 otherwise.

kNN 的实质是，根据给定 Label 的 Train 的数据，对 Test 中的每个数据，寻找最近的 kN（kN，最近邻数，一般大于 10 且取奇数，和聚类数 k 不是一个变量）个 Train 邻居。在这 kN 个最近邻居中，哪个聚类的邻居最多，该数据就属于哪一类。kNN 不需要

迭代，一次计算即可完成。

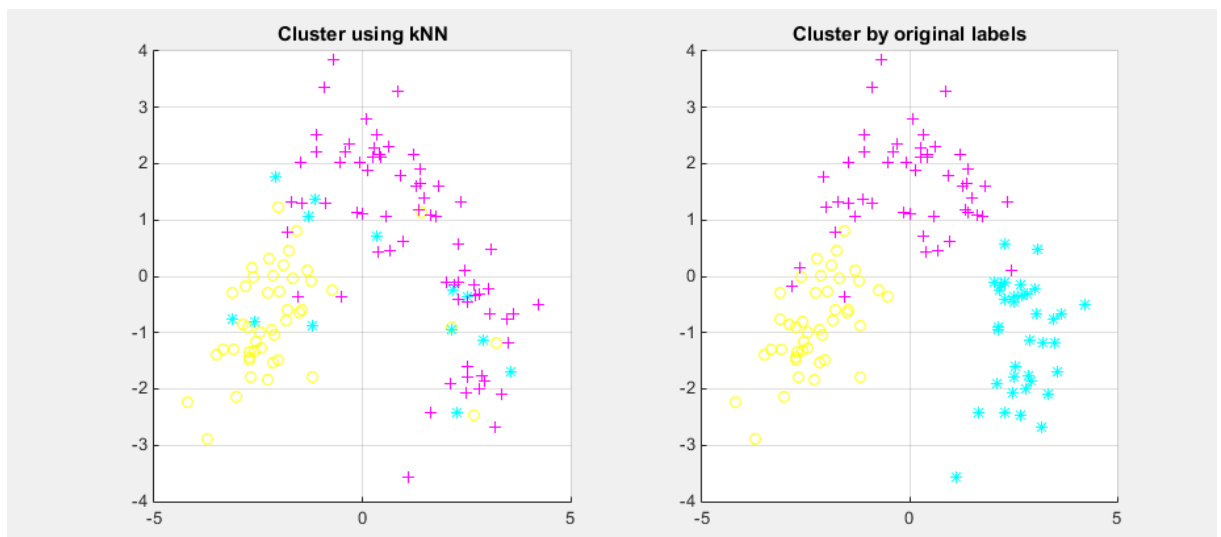


摘自维基百科[4]的一张解释 kNN 算法的经典图例。显然，kN 的设置对 kNN 的聚类效果影响很大。

一般来说，kN 越小，聚类边界越呈现齿状分布，kN 越大，聚类边界越平滑[5]。

2.2. 原始结果

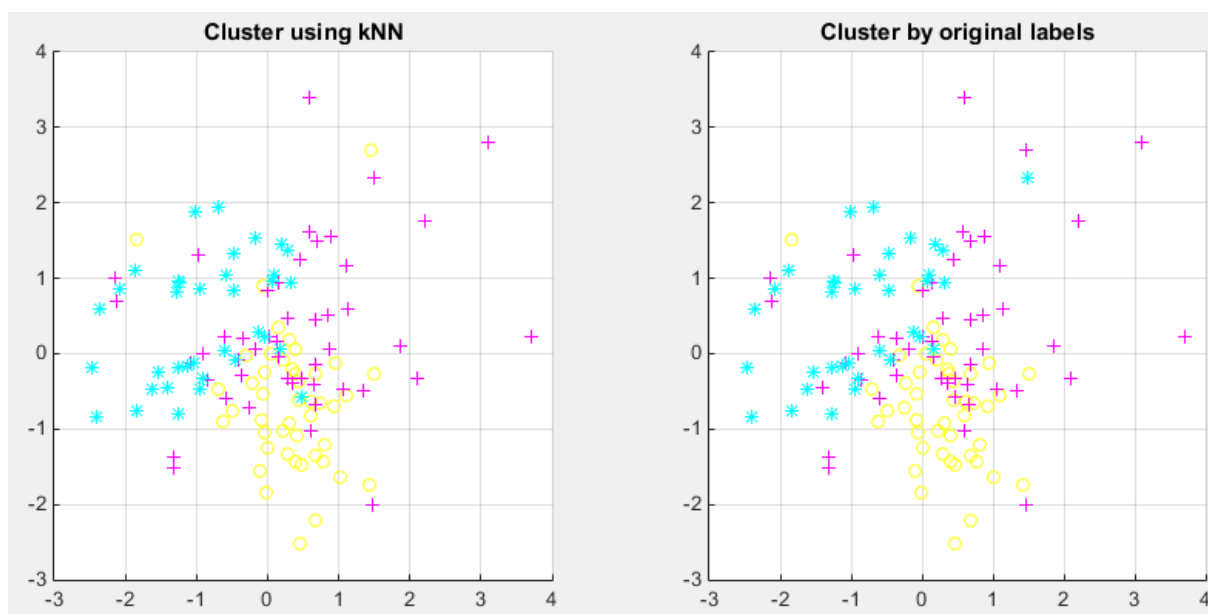
同样对 Wine 数据进行处理，它共有 178 个样本，初始化 kNN 聚类器时随机选取其中 50 个样本作为 Train（保证每个 Label 都有）。



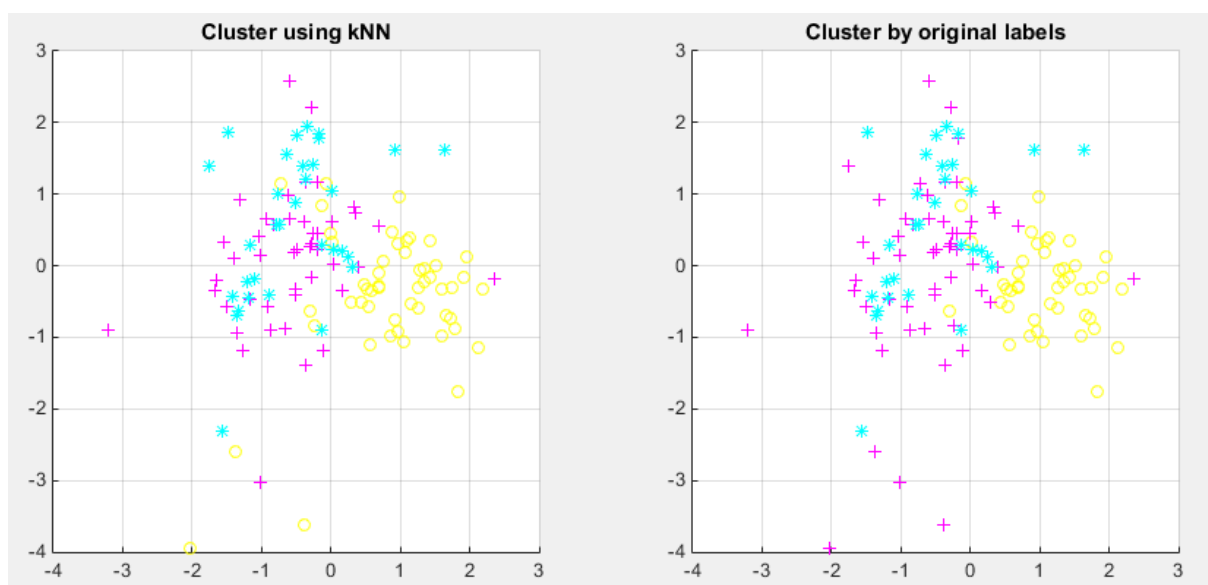
和 kMeans 一样，不进行数据预处理，结果很不理想，但是和 kMeans 比，同样不进行预处理，效果却比 kMeans 好一些。

2.3. 数据标准化和降维

同 kMeans，把 Wine 数据降到 5 维，同时取 kN=9，结果如下：

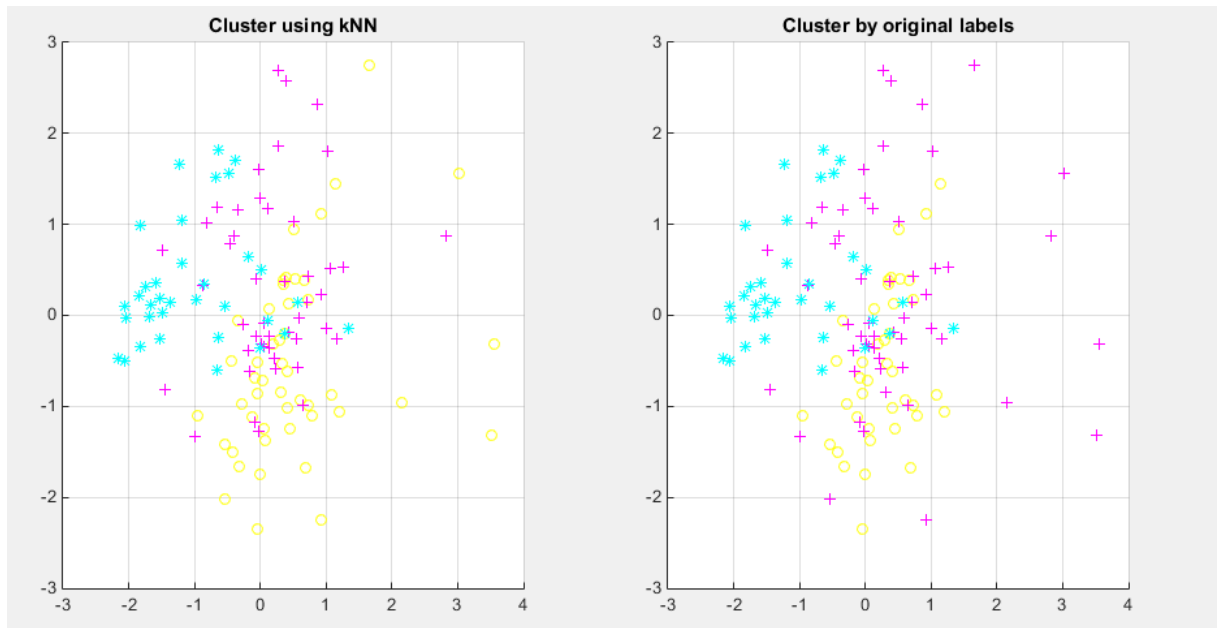


在调试过程中发现有趣的现象， $kN=10$ 的结果：

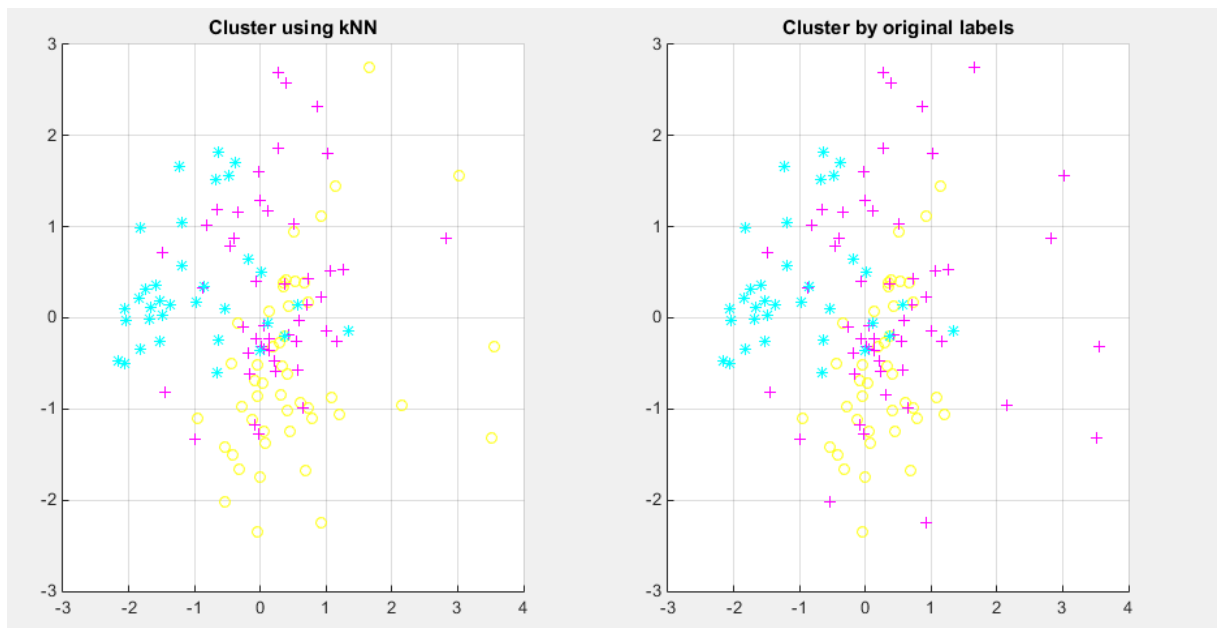


明显没有 $kN=9$ 好，错误点数比较明显。

对比 $kN=19$ ：



kN=20:



总之 kN 取奇数可能比较好，因为能以较大概率避免“投票数”相等的情况，不过实验结果是在取不同的 Train 集合得到的，只能参考，有待深入研究。

3. 聚类算法比较——kMeans 与 kNN

严格意义上讲，kNN 属于分类算法，kMeans 属于聚类算法，不过这两个概念有时候界限比较模糊，因为用聚类的方法也可以实现分类。两者最主要的区别是，kNN 是有监督的，即要有 Train 集，kNN 通过已有的 Label（完全正确的），通过最近的 kN（kNN

中 k 的含义) 个点的 Label 确定 Test 的 Label[6]。而 kMeans 是无监督的, 不需要 Train 集, 给定数据就能跑。同时 kMeans 需要反复迭代, kNN 一次完成。

从方法上说, kNN 是基于统计的 (kN 的设定也有一套统计学方法), 属于 lazy-learning, 又属于 memory-based-learning, 本质上是分类算法。kMeans 则是 Centroid-based, 并且有明显的学习过程 (迭代), 本质上是聚类算法。

应用上, kNN 适合已知 Train 集来对 Test 集做预测, 聚类数 k 隐式给定。kMeans 则适合对未知数据做预分析, 同时聚类数 k 需要人工显式给出。它们各有优缺点, kNN 需要 Train 集, 并且 kN 需要调整, 但是抗噪声能力强 (基于统计和距离)。kMeans 不需要 Train 集, 但是也要设定 k , 并且抗异常数据能力弱 (基于样本中心的)。

但是它们都有改进的方法, kNN 的 kN 有一套基于统计学的自适应设定方法, kMeans 则可以改进聚类中心的初始化方法 (kMeans++ 算法), 同时 k 值也可以利用基于聚类质量评估的方法来自动设定 (不吝计算资源的话)。

算法效率上, 取决于 kMeans 的收敛速度。从实现方法上来看, kMeans 主要时间花在迭代上, 距离计算很快, 每个样本点只要计算 k 次距离 (k 一般为个位数)。kNN 不需要迭代, 但是每个样本点对每个 Train 集样本都要计算距离。kMeans 的时间复杂度为 $O(n*k*t)$, 而 kNN 为 $O(n*n)$ 。如果对 kMeans 的迭代次数 t 做限制, 则 kMeans 明显更快一些。不过如果 kNN 做了基于 K-D 树的改进实现的话, 时间复杂度可以降低到 $O(\log N)$ [7]。

4. 基础 Apriori 算法

4.1. 原理

算法 6.2.1 Apriori。使用逐层迭代方法基于候选产生找出频繁项集。

输入:

- D : 事务数据库。
- min_sup : 最小支持度阈值。

输出: L , D 中的频繁项集。

方法:

```
(1)   $L_1 = \text{find\_frequent\_1\_itemsets}(D);$ 
(2)  for ( $k=2; L_{k-1} \neq \emptyset; k++$ ) {
(3)     $C_k = \text{apriori\_gen}(L_{k-1});$ 
(4)    for each 事务  $t \in D$  {           // 扫描  $D$ , 进行计数
(5)       $C_t = \text{subset}(C_k, t);$        // 得到  $t$  的子集, 它们是候选
```

图 6.4 挖掘布尔关联规则发现频繁项集的 Apriori 算法


```

(6)      for each 候选  $c \in C_k$ 
(7)           $c.count++$ ;
(8)      }
(9)       $L_k = \{c \in C_k \mid c.count \geq min\_sup\}$ 
(10) }
(11) return  $L = \bigcup_k L_k$ ;

procedure apriori_gen( $L_{k-1}$ : frequent( $k-1$ ) itemset)
(1)  for each 项集  $l_1 \in L_{k-1}$ 
(2)      for each 项集  $l_2 \in L_{k-1}$ 
(3)          if ( $l_1[1] = l_2[1] \wedge \dots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-2])$ ) then {
(4)               $c = l_1 \bowtie l_2$ ;           // 连接步: 产生候选
(5)              if has_infrequent_subset( $c, L_{k-1}$ ) then
(6)                  delete  $c$ ;           // 剪枝步: 删除非频繁的候选
(7)              else add  $c$  to  $C_k$ ;
(8)          }
(9)  return  $C_k$ ;

procedure has_infrequent_subset ( $c$ : candidate  $k$  itemset;  $L_{k-1}$ : frequent( $k-1$ ) itemset)
// 使用先验知识
(1)  for each ( $k-1$ ) subset  $s$  of  $c$ 
(2)      if  $s \notin L_{k-1}$  then
(3)          return TRUE;
(4)  return FALSE;
    
```

Apriori 是基于频繁项集来确定关联规则的算法。而搜寻频繁项集则利用其反单调性的特点（频繁项集的子集一定也是频繁项集，反之不一定成立；非频繁项集的超集一定也是非频繁项集）。“频繁”的定义也是设定支持度来决定的。从 C_1 候选 1 项集开始，逐步选择出满足最小支持度的频繁 k 项集 L_k 。其中两步非常重要，连接和剪枝。连接是通过取频繁 $k-1$ 项集来产生可能的频繁 k 项集。剪枝则在候选 k 项集 C_k 中剔除不可能的数据，利用了频繁项集的先验知识——如果 C_k 有一个子集 C_{k-1} 不是频繁项集，则 C_k 一定不是频繁项集，由此也看出 Apriori（先验的）算法名称的由来。

获得最大频繁项集后，再由大频繁项集开始，产生关联规则，由于大频繁项集是由小频繁项集做并集运算产生的，而算法运行时，对每种可能的频繁项集在事务集合 Transactions 中的出现次数进行了计数——即绝对支持度，故置信度很容易求得。

$$L = A \cup B$$

其中 L 为大频繁项集， A 为从 L 中任意选取的单元元素组成的子集， B 是补集。则置信度：

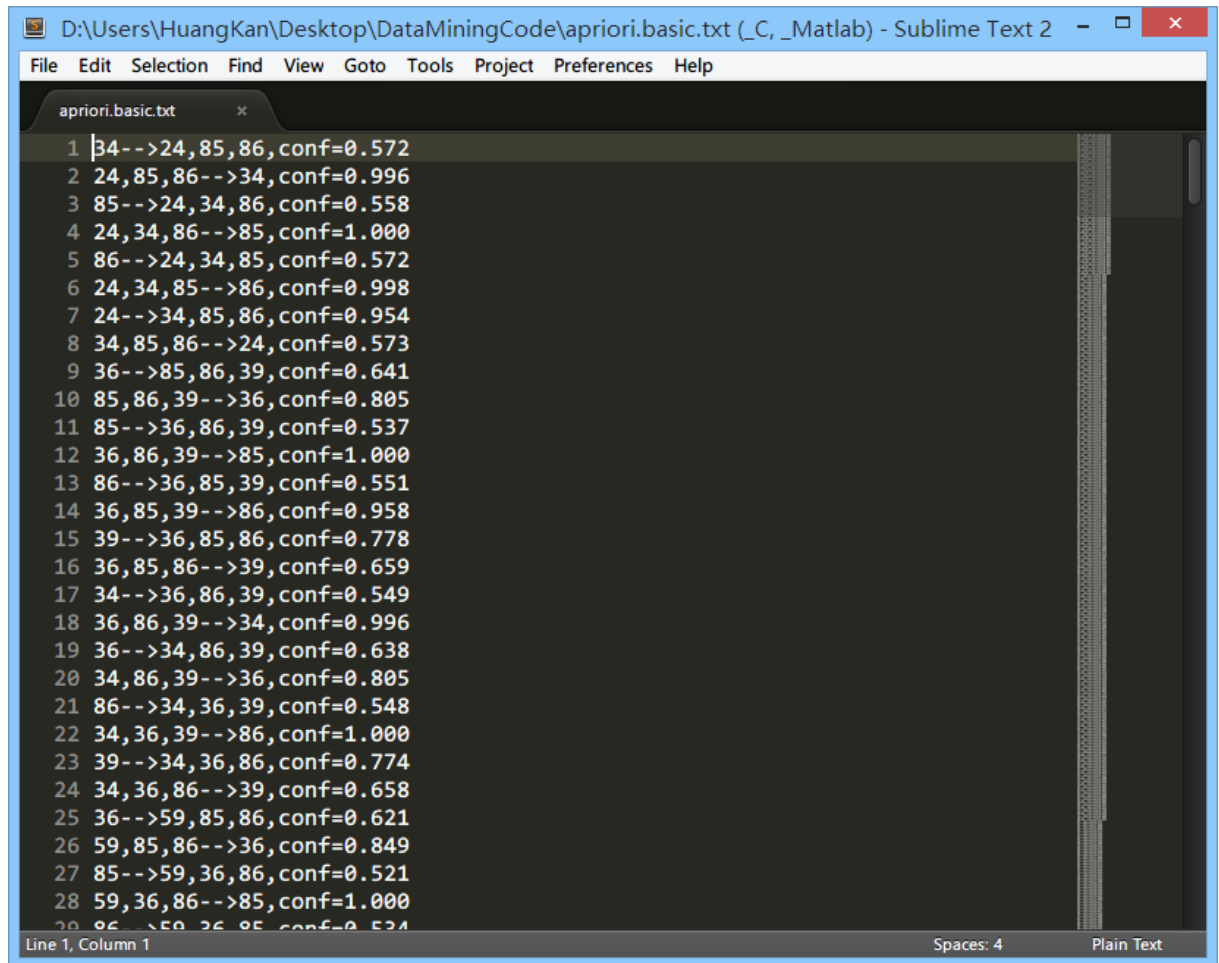
$$conf(A \Rightarrow B) = P(B \mid A) = \frac{P(AB)}{P(A)} = \frac{\sigma(L)}{\sigma(A)}$$

反之：

$$\text{conf}(B \Rightarrow A) = P(A|B) = \frac{P(AB)}{P(B)} = \frac{\sigma(L)}{\sigma(B)}$$

产生了单元素->多元素和多元素->单元素的关联规则，实际上也可以做多元素->多元素的，但是会产生更多的关联规则，时间复杂度也会更大。

以下是使用 Mushroom 数据，实现 Apriori 产生的结果，最小相对支持度设为 0.5（至少在 T 中出现一半以上）。



```

1 34-->24,85,86,conf=0.572
2 24,85,86-->34,conf=0.996
3 85-->24,34,86,conf=0.558
4 24,34,86-->85,conf=1.000
5 86-->24,34,85,conf=0.572
6 24,34,85-->86,conf=0.998
7 24-->34,85,86,conf=0.954
8 34,85,86-->24,conf=0.573
9 36-->85,86,39,conf=0.641
10 85,86,39-->36,conf=0.805
11 85-->36,86,39,conf=0.537
12 36,86,39-->85,conf=1.000
13 86-->36,85,39,conf=0.551
14 36,85,39-->86,conf=0.958
15 39-->36,85,86,conf=0.778
16 36,85,86-->39,conf=0.659
17 34-->36,86,39,conf=0.549
18 36,86,39-->34,conf=0.996
19 36-->34,86,39,conf=0.638
20 34,86,39-->36,conf=0.805
21 86-->34,36,39,conf=0.548
22 34,36,39-->86,conf=1.000
23 39-->34,36,86,conf=0.774
24 34,36,86-->39,conf=0.658
25 36-->59,85,86,conf=0.621
26 59,85,86-->36,conf=0.849
27 85-->59,36,86,conf=0.521
28 59,36,86-->85,conf=1.000
29 86-->59,36,85,conf=0.534
  
```

Mushroom 数据共 8k 多条，算法在 3.2s 内完成（包括读写文件）。得到了 298 个关联规则。

5. 带约束 Apriori 算法

带约束的 Apriori 算法即除了限制最小支持度，还限制了最小置信度，即 confidence 小于 min_conf，则该关联规则不予收录。测试程序将 min_conf 设置为 0.8。在 3s 内完成，得到了 176 个规则。

```

159 39-->90,conf=0.887
160 63-->85,conf=1.000
161 24-->85,conf=1.000
162 63-->86,conf=0.961
163 59-->90,conf=0.915
164 53-->90,conf=1.000
165 39-->36,conf=0.812
166 24-->86,conf=0.958
167 24-->34,conf=0.956
168 63-->90,conf=0.911
169 67-->86,conf=1.000
170 59-->36,conf=0.855
171 24-->90,conf=0.928
172 76-->86,conf=1.000
173 2-->85,conf=1.000
174 63-->36,conf=0.848
175 59-->63,conf=0.803
176 63-->59,conf=0.842
177

[Finished in 3.0s]

```

6. 改进 Apriori 算法

改进 Apriori 算法的方法有很多，基于 hash、事务压缩、划分、抽样等，还可以用比较复杂的数据结构来优化算法流程，如 FP-Tree 算法。我做了比较简单的，对原始数据进行抽样。

只选取原始事务集 T 中一半的元素（等距离采样），min_conf 和带约束的 Apriori 一样，设为 0.8。结果是 1.9s 就能完成，并且产生的关联规则更多了，有 185 个。应该不是程序的问题，可能与数据的分布有关系。

```

168 39-->90,conf=0.885
169 63-->85,conf=1.000
170 63-->86,conf=0.965
171 59-->90,conf=0.914
172 24-->85,conf=1.000
173 53-->90,conf=1.000
174 39-->36,conf=0.811
175 24-->86,conf=0.962
176 24-->34,conf=0.960
177 63-->90,conf=0.910
178 67-->86,conf=1.000
179 76-->86,conf=1.000
180 59-->36,conf=0.850
181 24-->90,conf=0.922
182 2-->85,conf=1.000
183 63-->36,conf=0.845
184 63-->59,conf=0.838
185 2-->34,conf=0.959
186

[Finished in 1.9s]

```

7. Apriori 算法复杂度

Apriori 算法的复杂度相对比较大，主要因为每次从 C_k 中选择 L_k ，都要对事务集整个遍历一遍。此外，如果 L_1 数量增大， C_2 将成倍增长，用基于散列的方法可以减小 L_1 到 C_2 的时间。由于多次扫描数据，Apriori 算法受事务集大小影响很大，时间复杂度接近 $O(T^2)$ ， T 为事务集大小，所以采样的方法也是非常可行的，能够很可观地减小时间，从实验结果也能看出来。

FP-growth 算法则在时间和空间上都比 Apriori 快一个量级，但是实现起来比较复杂。

8. 感想与总结

这几天都呆在自习室写程序，还是感觉自己效率太低了。代码写得不熟练，不过也和语言的选取有关。记得莫老师说过，语言只是工具，算法的思想是最重要的。

我觉得有一定道理，毕竟语言都是相通的。但是又不那么对。

因为做数模的时候用过聚类，所以算是比较熟悉。但是作业要求面向对象编程，就去查了 Mathworks 的官方资料，发现 Matlab 真的可以面向对象编程，而且资料非常详细。但是使用 Matlab 用户体验十分不好，启动慢，调试也慢，还有各种诸如矩阵维度不匹配的问题，不过可能也与我的编程习惯不太好有关。

各种语言有各自的优缺点，我认为 Matlab 不适合做算法编程，只适合建模仿真，矩阵运算和数学分析才是 Matlab 的长处，而不是 OO 编程。Matlab 的语法属于对科学家友好，但是对程序员不友好的。我甚至感觉写 Matlab 代码久了，自己编程习惯都差了。但是毕竟尝试过，也学了点东西，并且以后还要用过这东西搞科研，也不算坏。

做第二个算法的时候果断放弃 Matlab，去学 Python 的面向对象。同为脚本语言，和 Matlab 一样，Python 省去了 C++、Java 编译的麻烦，并且十分轻巧，一个文本编辑器和控制台就能写程序了。Python 的文本字符串处理能力比较强，自带的数据类型也适合做各种算法。比如 Apriori 中要做集合合并，一 Google 发现 Python 竟然有 set 类，愈发觉得 Python 的强大和自己水平还不够。可能别的语言也有，比如 C# 和 Java 的 Collections 类，但是用起来肯定没有 Python 这么方便了。Python 作为开源语言，有各路大牛开发各种库，刚看到一个图形库，效果堪比 Matlab，本打算学一学做 C4.5 的可视化的，但是时间不够也只好放弃了。配合 R 语言，Python 做数据挖掘会十分强大并且方便。

所以语言固然不是最重要的，但是选好工具还是有点必要。但是另一方面，不管用什么语言，只是实现算法是不够的。我感觉自己学得太盲目，忘记了思考。如果想当算法科学家，这种状态肯定不行。上莫老师的课的时候，有一些想法的交流，我才发现自己想法少得可怜，还是学识不够吧，做得不够好。实现算法是一回事，创新和钻研则是另外一回事儿。

最后说说对自己编程的看法，这个学期经常上知乎等程序猿网站，也耳濡目染了一些。已经大三了，到头来发现自己编程水平还远远不够。无论是习惯上、效率上、算法

知识上，感觉自己都做得不够好。虽然做数模期间学了很多算法，但是毕竟走马观花，很不熟练，加上数模基地浮躁的环境（很多人直接用网上的代码），并没有锻炼自己真正需要的能力。所以自己大三真的做的不够好。但是这学期也有点进步吧，逼着自己看编程规范文档，改掉坏习惯，也没事看看别人写的代码，挑挑毛病。

这学期因为各种原因没有好好上课，都没有 will-to-win 的斗志了，记得去年翘课也来蹭数据挖掘来着，可能是“大三病”吧，也很后悔，有时候都找不到学习的意义。最近几天又忙了起来，不管什么事情，不管有没有用，先一件件做好吧，活得充实、学得充实是很重要的。

最后，我马上大四了，特优生也毕业了，以后和莫老师交流的机会不多了，十分感谢莫老师的培养。虽然莫老师上课很多人不听，但是我喜欢这种方式，我觉得比起大部分照本宣科的课强太多了，希望老师继续保持个性，继续开拓学生的思维。

此致。

9. 附件清单

FlieName	Description
apriori.basic.txt 等	Apriori 算法关联规则结果文件
Apriori.py	Apriori 各个类
Clusterer.m	Matlab 类，集成 kMeans 和 kNN 算法
iris.data 等，wine.data 等	UCI 原始数据，用 Python 预处理后供 Matlab 读入的数据
PCA.m	MatlabPCA 类，集成降维方法
wineDataAdapter.py 等	Python 的数据预处理脚本
wineTestClusterer.m 等	Matlab 用数据测试聚类算法的主脚本

10. 参考文献

[1]The.Top.Ten.Algorithmsin.DataMining

[2]UCI 数据 Iris: <http://archive.ics.uci.edu/ml/datasets/Iris>

[3]UCI 数据 Wine: <http://archive.ics.uci.edu/ml/datasets/Wine>

- [4] 维 基 百 科 - 最 近 邻 居 法 :
<https://zh.wikipedia.org/wiki/%E6%9C%80%E8%BF%91%E9%84%B0%E5%B1%85%E6%B3%95>
- [5] KNN (K Nearest Neighbor) 算 法 的 MatLab 实 现 :
<http://blog.csdn.net/rk2900/article/details/9080821>
- [6] Kmeans 、 Kmeans++ 和 KNN 算 法 比 较 :
<http://blog.csdn.net/chlele0105/article/details/12997391>
- [7] 从 K 近邻算法、距离度量谈到 KD 树、SIFT+BBF 算法: <http://www.cnblogs.com/v-July-v/archive/2012/11/20/3125419.html>
- [8] PYTHON 风格规范: http://zh-google-styleguide.readthedocs.org/en/latest/google-python-styleguide/python_style_rules/
- [9] mushroom 数据集: <http://fimi.ua.ac.be/data/>