目录

[**任务一、Apriori算法的编程实现** -1-](#_Toc118921250)

[**一、算法简介** -1-](#_Toc118921251)

[**二、算法内容** -1-](#_Toc118921252)

[**三、算法结果与分析** -2-](#_Toc118921253)

[**四、小结与心得体会** -2-](#_Toc118921254)

[**任务二、KNN算法的编程实现** -3-](#_Toc118921255)

[**一、算法简介** -3-](#_Toc118921256)

[**二、算法内容** -3-](#_Toc118921257)

[**三、算法结果与分析** -3-](#_Toc118921258)

[**四、小结与心得体会** -3-](#_Toc118921259)

[**任务三、ID3算法的编程实现** -4-](#_Toc118921260)

[**一、算法简介** -4-](#_Toc118921261)

[**二、算法内容** -4-](#_Toc118921262)

[**三、算法结果与分析** -6-](#_Toc118921263)

[**四、小结与心得体会** -6-](#_Toc118921264)

[**任务四、朴素贝叶斯算法的编程实现** -7-](#_Toc118921265)

[**一、算法简介** -7-](#_Toc118921266)

[**二、算法内容** -7-](#_Toc118921267)

[**三、算法结果与分析** -8-](#_Toc118921268)

[**四、小结与心得体会** -8-](#_Toc118921269)

[**任务五、K-Means算法的编程实现** -9-](#_Toc118921270)

[**一、算法简介** -9-](#_Toc118921271)

[**二、算法内容** -9-](#_Toc118921272)

[**三、算法结果与分析** -11-](#_Toc118921273)

[**四、小结与心得体会** -11-](#_Toc118921274)

[**任务六、Agnes算法的编程实现** -12-](#_Toc118921275)

[**一、算法简介** -12-](#_Toc118921276)

[**二、算法内容** -12-](#_Toc118921277)

[**三、算法结果与分析** -13-](#_Toc118921278)

[**四、小结与心得体会** -13-](#_Toc118921279)

[**任务七、DBSCAN算法的编程实现** -14-](#_Toc118921280)

[**一、算法简介** -14-](#_Toc118921281)

[**二、算法内容** -14-](#_Toc118921282)

[**三、算法结果与分析** -15-](#_Toc118921283)

[**四、小结与心得体会** -15-](#_Toc118921284)

**任务一、Apriori算法的编程实现**

1. **算法简介**

Apriori算法是常用的用于挖掘出数据关联规则的算法，通过关联分析寻找关联规则，可以科学地发现数据集中的隐匿关系，用于相关业务的决策支持。Apriori算法核心思想是首先设定最小支持度和置信度，假如存在一条规则能够满足支持度和置信度大于设定的值，该规则即为强关联规则。Apriori 算法是数据挖掘关联规则的频繁项集算法，主要是通过产生频繁项集和产生规则两个阶段实现挖掘。

1. **算法内容**

算法流程

* 扫描数据集，得到所有出现过的数据，作为候选1项集。
* 挖掘频繁k项集。
  + 扫描计算候选k项集的支持度。
  + 剪枝去掉候选k项集中支持度低于最小支持度α的数据集，得到频繁k项集。如果频繁k项集为空，则返回频繁k-1项集的集合作为算法结果，算法结束。如果得到的频繁k项集只有一项，则直接返回频繁k项集的集合作为算法结果，算法结束。
  + 基于频繁k项集，连接生成候选k+1项集。
* 利用步骤2，迭代得到k=k+1项集结果。

关键代码

def lk\_2\_ck\_plus\_1(lk):

lk\_list = list(lk)

ck\_plus\_1 = set()

lk\_size = len(lk)

if lk\_size > 1:

k = len(lk\_list[0])

for i, j in itertools.combinations(range(lk\_size), 2):

t = lk\_list[i] | lk\_list[j]

if len(t) == k + 1:

ck\_plus\_1.add(t)

return ck\_plus\_1

def get\_all\_L(data\_set, min\_support):

c1 = build\_c1(data\_set)

no\_frequent\_set = []

l1 = ck\_2\_lk(data\_set, ck=c1, min\_support=min\_support, no\_frequent\_set=no\_frequent\_set)

L = l1

Lk = l1

while len(Lk) > 0:

lk\_key\_list = list(Lk.keys())

ck\_plus\_1 = lk\_2\_ck\_plus\_1(lk\_key\_list)

Lk = ck\_2\_lk(data\_set, ck\_plus\_1, min\_support, no\_frequent\_set)

if len(Lk) > 0:

L.update(Lk)

else:

break

return L

def get\_maximum\_frequent\_itemset(L):

max\_L = {}

for i in L:

flag = 1

for j in L:

if len(i) >= len(j):

continue

if i.issubset(j):

flag = 0

break

if flag == 1:

max\_L[i] = L[i]

return max\_L

if \_\_name\_\_ == '\_\_main\_\_':

file\_path = "data/data\_in\_book.txt"

min\_support = 0.4

min\_confidence = 0.6

is\_maximum\_frequent\_itemset = 0

data\_set = load\_data(file\_path)

data\_set, index\_2\_str = data\_2\_index(data\_set)

L = get\_all\_L(data\_set, min\_support=min\_support)

if is\_maximum\_frequent\_itemset:

max\_L = get\_maximum\_frequent\_itemset(L)

result = rules\_from\_L(max\_L, L, min\_confidence=min\_confidence)

else:

result = rules\_from\_L(L, L, min\_confidence=min\_confidence)

result = return\_2\_str(index\_2\_str, result)

1. **算法结果与分析**

计算出的结果（输出最大频繁项目集的强关联规则）

序号 Ik Xm-1 置信度 支持度 规则

1 BCE E 67 % 40 % E → BC

2 BCE CE 100 % 40 % CE → B

3 BCE BE 67 % 40 % BE → C

4 ABCD D 67 % 40 % D → ABC

5 ABCD A 67 % 40 % A → BCD

6 ABCD CD 100 % 40 % CD → AB

7 ABCD BD 67 % 40 % BD → AC

8 ABCD AD 100 % 40 % AD → BC

9 ABCD AC 67 % 40 % AC → BD

10 ABCD AB 67 % 40 % AB → CD

11 ABCD BCD 100 % 40 % BCD → A

12 ABCD ACD 100 % 40 % ACD → B

13 ABCD ABD 100 % 40 % ABD → C

14 ABCD ABC 67 % 40 % ABC → D

1. **小结与心得体会**

Apriori算法简单，易于实现。但是它也有自己的缺点，数据集很大的时会出现下面两个问题。

1. 需要多次扫描数据集
2. 可能会产生庞大的候选集

**任务二、KNN算法的编程实现**

1. **算法简介**

k近邻法（k-nearest neighbor）是一种基本分类与回归方法。k值的选择、距离的度量及分类决策规则是k近邻法的三个基本元素。

k近邻法不具有显式的学习过程，也就是没有一个学习好的模型。

1. **算法内容**

关键代码：

def my\_knn(data\_normed, sample, labels, k=3):

new\_data\_normed = tile(sample, (data\_normed.shape[0], 1)) - data\_normed

print(tile(sample, (data\_normed.shape[0], 1)))

double\_matrix = new\_data\_normed \*\* 2

double\_distance = double\_matrix.sum(axis=1)

sqrt\_distance = double\_distance \*\* 0.5

new\_matrix = pd.DataFrame()

new\_matrix["distance"] = sqrt\_distance

new\_matrix["label"] = labels

new\_matrix = new\_matrix.sort\_values(by=["distance"], ascending=True)

final\_matrix = new\_matrix.iloc[:k, :]

label\_dict = {}

for i in set(labels):

label\_dict[i] = 0

for i in range(k):

label\_dict[final\_matrix.iloc[i]["label"]] += 1

print(label\_dict)

sorted\_label = sorted(label\_dict.items(), key=operator.itemgetter(1), reverse=True)

return sorted\_label[0][0]

if \_\_name\_\_ == '\_\_main\_\_':

file\_path = "./dataset/bookTestSet.txt"

header = ["性别", "身高", "类别"]

sample = [1, 1.5]

data, target = read\_data(file\_path, header)

data\_normed, sample = to\_normalize(data, header, sample)

print(data\_normed) # 打印0,1归一化后的矩阵信息

label = my\_knn(data\_normed, sample, target, k=5)

print("KNN结果是：", label)

1. **算法结果与分析**

运行结果

{'矮': 3, '高': 0, '中等': 2}

KNN结果是： 矮

1. **小结与心得体会**

优点：精度高，对异常值不敏感，无数据输入假定

缺点：计算复杂度高，空间复杂度高

适用数据范围：数值型和标称型

**任务三、ID3算法的编程实现**

1. **算法简介**

ID3作为一种经典的决策树算法，是基于信息熵来选择最佳的测试属性，其选择了当前样本集中具有最大信息增益值的属性作为测试属性。

样本集的划分则依据了测试属性的取值进行，测试属性有多少种取值就能划分出多少的子样本集；同时决策树上与该样本集相应的节点长出新的叶子节点。

ID3算法根据信息论理论，采用划分后样本集的不确定性作为衡量划分样本子集的好坏程度，用“信息增益值”度量不确定性——信息增益值越大，不确定性就更小，这就促使我们找到一个好的非叶子节点来进行划分。

1. **算法内容**

关键代码

def calcShannonEnt(dataSet):

numEntires = len(dataSet) # 返回数据集的行数

labelCounts = {} # 保存每个标签(Label)出现次数的字典

for featVec in dataSet: # 对每组特征向量进行统计

currentLabel = featVec[-1] # 提取标签(Label)信息

if currentLabel not in labelCounts.keys(): # 如果标签(Label)没有放入统计次数的字典,添加进去

labelCounts[currentLabel] = 0

labelCounts[currentLabel] += 1 # Label计数

shannonEnt = 0.0 # 经验熵(香农熵)

for key in labelCounts: # 计算香农熵

prob = float(labelCounts[key]) / numEntires # 选择该标签(Label)的概率

shannonEnt -= prob \* log(prob, 2) # 利用公式计算

return shannonEnt # 返回经验熵(香农熵)

def splitDataSet(dataSet, axis, value):

retDataSet = [] # 创建返回的数据集列表

for featVec in dataSet: # 遍历数据集

if featVec[axis] == value:

reducedFeatVec = featVec[:axis] # 去掉axis特征

reducedFeatVec.extend(featVec[axis + 1:]) # 将符合条件的添加到返回的数据集

retDataSet.append(reducedFeatVec)

return retDataSet # 返回划分后的数据集

def chooseBestFeatureToSplit(dataSet):

numFeatures = len(dataSet[0]) - 1 # 特征数量

baseEntropy = calcShannonEnt(dataSet) # 计算数据集的香农熵

bestInfoGain = 0.0 # 信息增益

bestFeature = -1 # 最优特征的索引值

for i in range(numFeatures): # 遍历所有特征

# 获取dataSet的第i个所有特征

featList = [example[i] for example in dataSet]

uniqueVals = set(featList) # 创建set集合{},元素不可重复

newEntropy = 0.0 # 经验条件熵

for value in uniqueVals: # 计算信息增益

subDataSet = splitDataSet(dataSet, i, value) # subDataSet划分后的子集

prob = len(subDataSet) / float(len(dataSet)) # 计算子集的概率

newEntropy += prob \* calcShannonEnt(subDataSet) # 根据公式计算经验条件熵

infoGain = baseEntropy - newEntropy # 信息增益

print("第%d个特征的增益为%.3f" % (i, infoGain)) # 打印每个特征的信息增益

if (infoGain > bestInfoGain): # 计算信息增益

bestInfoGain = infoGain # 更新信息增益，找到最大的信息增益

bestFeature = i # 记录信息增益最大的特征的索引值

return bestFeature # 返回信息增益最大的特征的索引值

def createTree(dataSet, labels, featLabels):

classList = [example[-1] for example in dataSet] # 取分类标签(是否放贷:yes or no)

if classList.count(classList[0]) == len(classList): # 如果类别完全相同则停止继续划分

return classList[0]

if len(dataSet[0]) == 1: # 遍历完所有特征时返回出现次数最多的类标签

return majorityCnt(classList)

bestFeat = chooseBestFeatureToSplit(dataSet) # 选择最优特征

bestFeatLabel = labels[bestFeat] # 最优特征的标签

featLabels.append(bestFeatLabel)

myTree = {bestFeatLabel: {}} # 根据最优特征的标签生成树

del (labels[bestFeat]) # 删除已经使用特征标签

featValues = [example[bestFeat] for example in dataSet]

uniqueVals = set(featValues) # 去掉重复的属性值

for value in uniqueVals:

subLabels = labels[:]

myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, value), subLabels, featLabels)

return myTree

def classify(inputTree, featLabels, testVec):

firstStr = next(iter(inputTree)) # 获取决策树结点

secondDict = inputTree[firstStr] # 下一个字典

featIndex = featLabels.index(firstStr)

for key in secondDict.keys():

if testVec[featIndex] == key:

if type(secondDict[key]).\_\_name\_\_ == 'dict':

classLabel = classify(secondDict[key], featLabels, testVec)

else:

classLabel = secondDict[key]

return classLabel

def createDataSet(file\_path):

labels = []

dataSet = []

i = 1

with open(file\_path, encoding="utf-8") as f:

for line in f:

line = line.strip("\n")

if i == 1:

labels = line.split(',')

else:

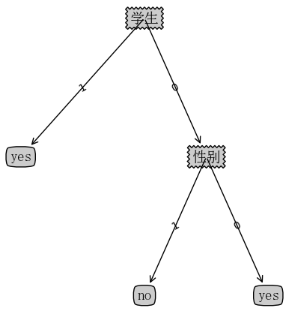
dataSet.append(line.split(','))

i = i + 1

return dataSet, labels # 返回数据集和分类属性

1. **算法结果与分析**

运行结果截图



1. **小结与心得体会**

D3算法十分简单，核心是根据“最大信息熵增益”原则选择划分当前数据集的最好特征，信息熵是信息论里面的概念，是信息的度量方式，不确定度越大或者说越混乱，熵就越大。在建立决策树的过程中，根据特征属性划分数据，使得原本“混乱”的数据的熵(混乱度)减少，按照不同特征划分数据熵减少的程度会不一样。在ID3中选择熵减少程度最大的特征来划分数据（贪心），也就是“最大信息熵增益”原则。

ID3算法缺点:

（1）只能处理分类属性的数据，不能处理连续的数据；

（2）划分过程会由于子集规模过小而造成统计特征不充分而停止；

（3）ID3算法不能增量的接受训练集，每增加一次实例就抛弃原有的决策树，重新构造新的决策树，开销很大。

（4）ID3算法在选择根节点和各内部节点中的分支属性时，采用信息增益作为评价标准。信息增益的缺点是倾向于选择取值较多的属性，而不是最优的属性，这样就有可能得到局部最优解而失去全局最优解;在搜索过程中无回溯；在有些情况下这类属性可能不会提供太多有价值的信息。

**任务四、朴素贝叶斯算法的编程实现**

1. **算法简介**

朴素贝叶斯法是基于贝叶斯定理与特征条件独立假设的分类方法。朴素贝叶斯分类器(Naive Bayes Classifier,或 NBC)发源于古典数学理论，有着坚实的数学基础，以及稳定的分类效率。

1. **算法内容**

关键代码

def typeCount(typeList, t):

cnt = 0

for tL in typeList:

if tL == t:

cnt += 1

return cnt

# 计算Y=-1或1的条件下，X等于某值 个数

def featCount(dataSet, i, feat, y):

cnt = 0

# print(i, feat, y)

for row in dataSet:

if row[i] == feat and row[-1] == y:

cnt += 1

return cnt

def calcBayes(dataSet, X):

lenDataSet = len(dataSet)

typeList = [row[-1] for row in dataSet]

typeSet = set(typeList) # 类别集合

print(typeList, typeSet)

pList = [] # 记录预计 各类类别 概率

for t in typeSet:

print('-' \* 50)

yNum = typeCount(typeList, t) # 计算yi的个数

print(f'{t} num =', yNum)

py = yNum / lenDataSet

print(f'P(Y = {t}) =', py)

pSum = py

# 对每个特征分量计数

for i in range(len(X)):

xiNum = featCount(dataSet, i, X[i], t) # 统计Y条件下 Xi取相应特征 的数量

print(f'特征{X[i]} num =', xiNum)

# 条件概率P{X = xi | Y = yi}

pxy = xiNum / yNum

print(f'条件概率 =', pxy)

pSum \*= pxy

print(f'P(X|Y= {t})P(Y= {t}) =', pSum)

pList.append(pSum)

# print(pList)

return pList, typeSet

# 就是找最大的概率，记录下标

def predict(pList, typeList):

for i in range(len(pList)):

if pList[i] == max(pList):

print('\*' \* 50)

print(f'预测类为 = {typeList[i]}')

return

def createDataSet(file\_path):

labels = []

dataSet = []

i = 1

with open(file\_path, encoding="utf-8") as f:

for line in f:

line = line.strip("\n")

if i == 1:

labels = line.split(',')

else:

dataSet.append(line.split(','))

i = i + 1

return dataSet, labels # 返回数据集和分类属性

1. **算法结果与分析**

运行结果

['0', '0', '1', '1', '1', '0', '1', '0', '1', '1', '1', '1', '1', '0'] {'0', '1'}

--------------------------------------------------

0 num = 5

P(Y = 0) = 0.35714285714285715

特征0 num = 3

条件概率 = 0.6

特征1 num = 2

条件概率 = 0.4

特征1 num = 1

条件概率 = 0.2

特征0 num = 2

条件概率 = 0.4

P(X|Y= 0)P(Y= 0) = 0.006857142857142858

--------------------------------------------------

1 num = 9

P(Y = 1) = 0.6428571428571429

特征0 num = 2

条件概率 = 0.2222222222222222

特征1 num = 4

条件概率 = 0.4444444444444444

特征1 num = 6

条件概率 = 0.6666666666666666

特征0 num = 6

条件概率 = 0.6666666666666666

P(X|Y= 1)P(Y= 1) = 0.02821869488536155

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

预测类 为 = 1

1. **小结与心得体会**

(1)朴素贝叶斯模型发源于古典数学理论，分类效率比较稳定。

(2)对小规模的数据表现很好，能够用于多分类任务的处理，适合增量式训练，尤其是在数据量超出内存的情况下，能够一批批的去增量训练。

(3)算法简单，对缺失数据不太敏感。

**任务五、K-Means算法的编程实现**

1. **算法简介**

kMeans算法是最常用的聚类算法,该算法的主要作用是将相似的样本自动归到一个类别中。

1. **算法内容**

关键代码

struct group {

vector<double> avg;//簇每个属性的平均值

vector<int> ids;//簇的成员id

void cal\_avg(int field\_num) {

for (int i = 1; i < field\_num; i++)

{

double ans = 0;

for (int j = 0; j < ids.size(); j++)

{

ans += sample[ids[j]-1][i];

}

avg.push\_back(ans / ids.size());

}

}

};

vector<group> get\_new\_groups(vector<group>& old\_groups, bool &flag) {

vector<group> new\_groups;

for (int i = 0; i < old\_groups.size(); i++)

{

group t;

new\_groups.push\_back(t);

}

for (int i = 0; i < row; i++)

{

int min\_distance = INT\_MAX;

int target\_groups=-1;

for (int k = 0; k < old\_groups.size(); k++)

{

int t\_distance = 0;

//序号不计算

for (int j = 1; j < column; j++)

{

//欧氏距离

t\_distance += pow(abs(sample[i][j] - old\_groups[k].avg[j - 1]), 2);

}

if (min\_distance > t\_distance)

{

min\_distance = t\_distance;

target\_groups = k;

}

}

new\_groups[target\_groups].ids.push\_back(sample[i][0]);

}

for (int i = 0; i < new\_groups.size(); i++) {

new\_groups[i].cal\_avg(column);

}

for (int i = 0; i < new\_groups.size(); i++) {

int len = new\_groups[i].avg.size(),j;

for (j = 0; j < len; j++)

{

if (new\_groups[i].avg[j]!=old\_groups[i].avg[j])

{

flag = false;

break;

}

}

if (j== len)

{

flag = true;

}

else

{

flag = false;

break;

}

}

old\_groups.clear();

return new\_groups;

}

int main()

{

string path = "data.csv";//书P178

ifstream inFile(path, ios::in);

if (!inFile)

{

cout << "打开文件失败！" << endl;

exit(1);

}

string line;

string field;

while (getline(inFile, line))//getline(inFile, line)表示按行读取CSV文件中的数据

{

sample.push\_back(vector<int>());

string field;

istringstream sin(line); //将整行字符串line读入到字符串流sin中

while (getline(sin, field, ',')) {//将字符串流sin中的字符读入到field字符串中，以逗号为分隔符

sample[row].push\_back(atoi(field.c\_str()));

}

row++;

}

inFile.close();

column = sample[0].size();

for (int i = 0; i < row; i++)

{

for (int j = 0; j < sample[i].size(); j++)

{

cout << sample[i][j] << " ";

}

cout << endl;

}

int k;

cout << "请输入要分多少组：\n";

cin >> k;

while (k > row) {

cout << "组数不合法";

cin >> k;

}

vector<group> old\_groups;

set<int> initial;//初始各个簇随机放一个id [1,row]

srand(time(0));

while (initial.size()!=k)

{

int id = rand() % row + 1;

initial.insert(id);

}

for (set<int>::iterator i = initial.begin(); i!=initial.end(); i++)

{

group t;

t.ids.push\_back(\*i);

t.cal\_avg(column);

old\_groups.push\_back(t);

}

print\_head(old\_groups);

int id = 1;

print\_content(old\_groups, id);

bool flag = false;

while (!flag)

{

old\_groups = get\_new\_groups(old\_groups,flag);

print\_content(old\_groups, id);

}

return 0;

}

1. **算法结果与分析**

运行结果截图



1. **小结与心得体会**

K-Means的主要优点：

1）原理简单，容易实现

2）可解释度较强

K-Means的主要缺点：

1）K值很难确定

2）局部最优

3）对噪音和异常点敏感

4）需样本存在均值（限定数据种类）

5）聚类效果依赖于聚类中心的初始化

6）对于非凸数据集或类别规模差异太大的数据效果不好

**任务六、Agnes算法的编程实现**

1. **算法简介**

“AGNES(AGglomerative NESting)算法是凝聚的层次聚类方法。AGNES最初将每个对象作为一个簇，然后这些簇根据某些准则被一步一步地合并。

1. **算法内容**

关键代码

# -\*- coding:utf-8 -\*-

import math

import pylab as pl

# 计算欧几里得距离,a,b分别为两个元组

def dist(a, b):

return math.sqrt(math.pow(a[0] - b[0], 2) + math.pow(a[1] - b[1], 2))

# dist\_min

def dist\_min(Ci, Cj):

return min(dist(i, j) for i in Ci for j in Cj)

# dist\_max

def dist\_max(Ci, Cj):

return max(dist(i, j) for i in Ci for j in Cj)

# dist\_avg

def dist\_avg(Ci, Cj):

return sum(dist(i, j) for i in Ci for j in Cj) / (len(Ci) \* len(Cj))

# 找到距离最小的下标

def find\_Min(M):

min = 1000

x = 0

y = 0

for i in range(len(M)):

for j in range(len(M[i])):

if i != j and M[i][j] < min:

min = M[i][j]

x = i

y = j

return (x, y, min)

# 算法模型：

def AGNES(dataset, dist, k):

# 初始化C和M

C = [] # 初始簇

M = []

for i in dataset:

Ci = []

Ci.append(i)

C.append(Ci)

for i in C:

Mi = []

for j in C:

Mi.append(dist(i, j))

M.append(Mi)

q = len(dataset)

# 合并更新

while q > k:

x, y, min = find\_Min(M)

C[x].extend(C[y])

C.remove(C[y])

M = []

for i in C:

Mi = []

for j in C:

Mi.append(dist(i, j))

M.append(Mi)

q -= 1

return C

def load\_data(file\_path):

dataset = []

with open(file\_path, encoding="utf-8") as f:

for line in f:

line = line.strip("\n")

t = []

for i in line.split(','):

t.append(float(i))

dataset.append(t)

return dataset

if \_\_name\_\_ == "\_\_main\_\_":

file\_path = "./dataset/dataset.csv"

category = 2

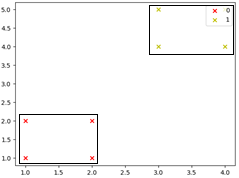
dataset = load\_data(file\_path)

C = AGNES(dataset, dist\_max, category)

draw(C)

1. **算法结果与分析**

运行结果截图



1. **小结与心得体会**

AGNES算法比较简单，但经常会遇到合并点选择的困难。如果在某一步没有很好地选择出合并点，很可能导致低质量的聚类结果。而且此算法没有良好的可伸缩性，算法复杂度较高。

**任务七、DBSCAN算法的编程实现**

1. **算法简介**

DBSCAN算法是一种基于密度的聚类算法,聚类的时候不需要预先指定簇的个数,最终簇的个数不确定。

1. **算法内容**

关键代码

import matplotlib.pyplot as plt

import random

import numpy as np

import math

list\_1 = []

list\_2 = []

def loadDataSet(fileName, splitChar='\t'):

dataSet = []

with open(fileName) as fr:

for line in fr.readlines():

curline = line.strip().split(splitChar)

fltline = list(map(float, curline)) # 将字符串转换成浮点数

dataSet.append(fltline)

return dataSet

def dist(t1, t2):

dis = math.sqrt((np.power((t1[0] - t2[0]), 2) + np.power((t1[1] - t2[1]), 2)))

# print("两点之间的距离为："+str(dis))

return dis

def dbscan(Data, Eps, MinPts):

num = len(Data) # 点的个数

# print("点的个数："+str(num))

unvisited = [i for i in range(num)] # 没有访问到的点的列表

# print(unvisited)

visited = [] # 已经访问的点的列表

C = [-1 for i in range(num)]

k = -1

while len(unvisited) > 0:

p = random.choice(unvisited)

unvisited.remove(p)

visited.append(p)

N = []

for i in range(num):

# 计算epsilon邻域中的对象的个数

if (dist(Data[i], Data[p]) <= Eps): # and (i!=p):

N.append(i)

if len(N) >= MinPts:

k = k + 1 # 簇编号加一

C[p] = k

# 对于p的epsilon邻域中的每个对象pi，将其加入p的簇中

for pi in N:

if pi in unvisited:

unvisited.remove(pi)

visited.append(pi)

# 找到核心对象p的邻域中的对象pi

# M是位于pi的邻域中的核心对象点的列表

M = []

for j in range(num):

if (dist(Data[j], Data[pi]) <= Eps): # and (j!=pi):

M.append(j)

if len(M) >= MinPts:

for t in M:

if t not in N:

N.append(t)

# 若pi不属于任何簇，C[pi] == -1说明C中第pi个值没有改动

if C[pi] == -1:

C[pi] = k

# 如果p的epsilon邻域中的对象数小于指定阈值，说明p是一个噪声点

else:

C[p] = -1

return C

def dataSet2(dataSet, Eps, MinPts):

C = dbscan(dataSet, Eps, MinPts)

print(C)

x = []

y = []

for data in dataSet:

x.append(data[0])

y.append(data[1])

plt.figure(figsize=(8, 6), dpi=80)

plt.scatter(x, y, c=C, marker='o')

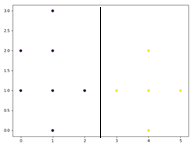
plt.show()

if \_\_name\_\_ == '\_\_main\_\_':

dataSet2(loadDataSet('book\_p191.txt', splitChar=','), 1, 4)

1. **算法结果与分析**

运行结果截图



1. **小结与心得体会**

优点

①可以对任意形状的稠密数据集进行聚类

②可以在聚类的同时发现异常点，对数据集中的异常点不敏感

③不需要指定簇数，并且多次实验结果往往是相同的

缺点

①如果样本集的密度不均匀、聚类间距差相差很大时，聚类质量较差，这时用DBSCAN聚类一般不适合

②调参相较于K-Means复杂一点，不同参数组合对聚类效果有很大影响

③由于传统的欧式距离不能很好处理高纬数据，所以对于距离定义较难给出好的解决方案

④DBSCAN算法适合处理不同簇的密度相对比较均匀的情况，当不同簇的密度变化很大时会出现一些问题