姓名：向建宇

学号：1801210687

1.用K\_Means对产品归类：数据集（Sales\_Transactions\_Dataset\_Weekly）

**算法原理：**

聚类是一种无监督的学习，它将相似的对象归到同一簇中，类似全自动分类。簇内的对象越相似，聚类的效果越好。K-均值聚类是每个类别簇都是采用簇中所含值的均值计算而成。

下面是K\_Means的算法过程：

|  |
| --- |
| 当任意一个点的簇分配结果发生改变时：  对数据集中的每个点：  对每个质心：  计算质心与数据点之间的距离  将数据点分配到距离其最近的簇  对每一个簇，计算簇中所有点的均值并将均值作为质心。 |

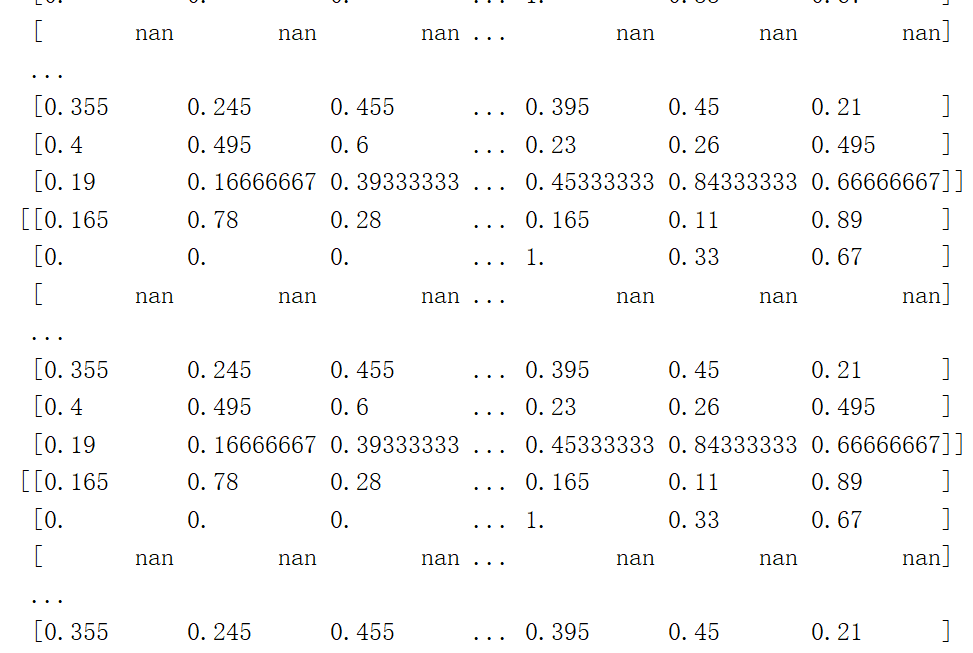
从上面可知，K\_Means通过不停的迭代，来划分K个类，而K值得选取将会很大成度上影响分类的结果，又因为k个初始点随机选取，所以结果也不稳定。

**思路分析：（具体步骤解释请看代码注释）**

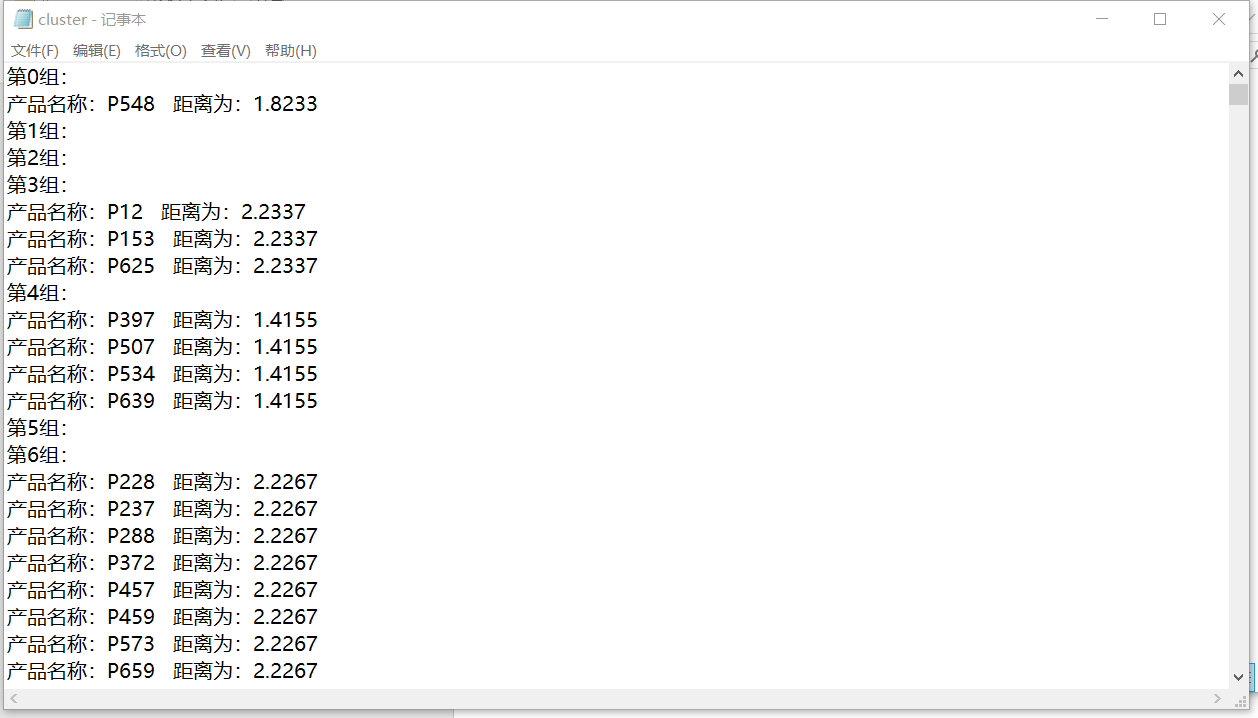
首先创建三个基本功能函数：加载数据、计算距离、初始化k个中心三个函数，以供主要函数kMeans(dataSet, k, distMeas=distEclud, createCent=randCent)调用，其中传入4个参数，分别是预处理的数据集，K的值，参看论文可知，K的取值与数据的大小D和预期每个簇包含的数据条数V有关，考虑到是寻找相似时间段的产品聚类，再加上论文的处理方式（k=D/V），我在本次实验也采取了上面的方法来取K值。在传入参数之后，kMeans算法采用“计算质心-分配-重新计算质心”反复迭代的方式，直到所有点的分配结果不再改变。

**结果截图：**

我将每次迭代的中心点打印出来：



并将产品的聚类结果写到了txt文件里以供查看：（其中的距离即为到簇中心点的距离）



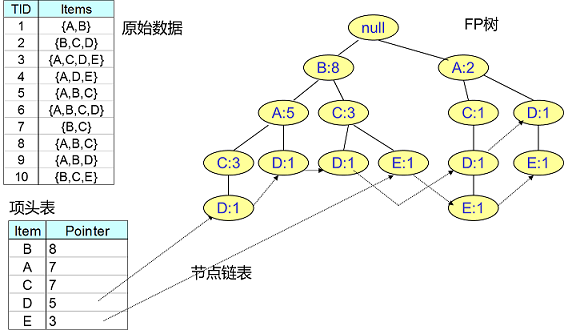
完整代码请见附录1

2.用fp\_tree找商品之间的关联性：数据集（retail）

**算法原理：**

根据论文及生活常识，我们可以发现很多商品都具有一定的关联性，而fp\_tree是找出这种关联性的一个比较好的办法。它用了比较巧妙的办法，大大缩短了时间复杂度

为了减少I/O次数，FP Tree算法引入了一些数据结构来临时存储数据。这个数据结构包括三部分，如下图所示：



然后再根据项头表找到所有的条件模式基，即关联的程度和关系

**思路分析：（具体步骤解释请看代码注释）**

1. 循环遍历两次数据集，第一次对所有产品进行计数，统计每个产品出现的个数，并把未达到阈值的产品删除，第二次遍历时，根据第一次遍历结果得到的产品数量排名来将每一条数据进行排序。

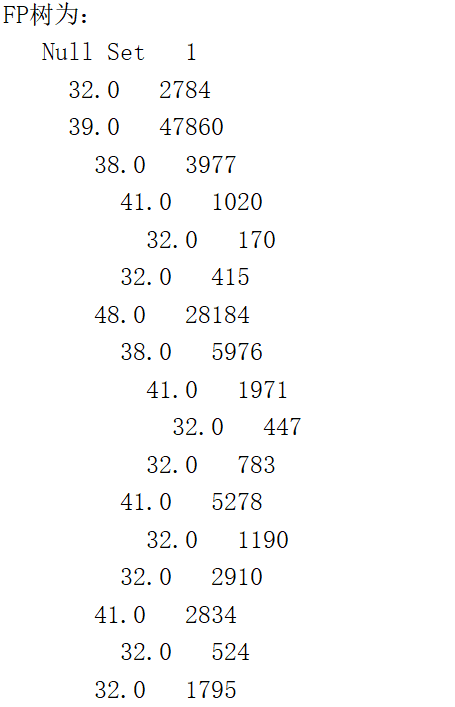
2. 根据第二次排序的结果，循环遍历生成一颗fp树，每一个结点记录了它的名字和计数。

3. 通过项头表，从小往上得到每一个的item的子树路径。

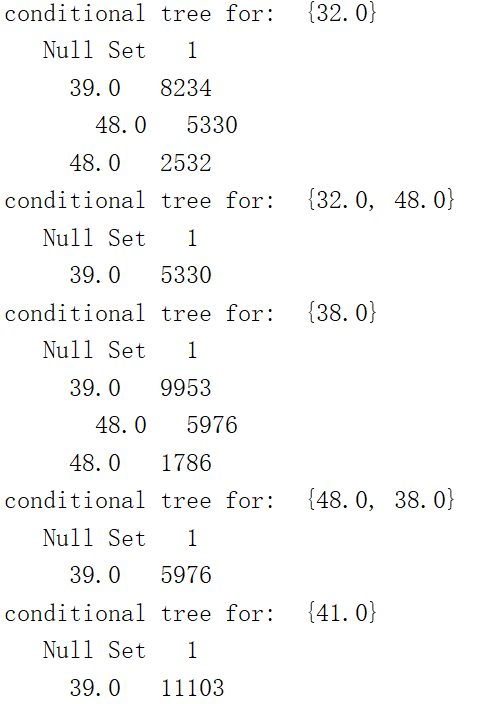
4. 通过子树路径得到他们的条件模式基。

**结果截图：**

生成的fp树为：



条件模式基为：



说明：如第一个32.0，则产品(32,48) 在一行数据共同出现5330次，(32,39) 在一行数据共同出现8234

第二个32.0，48.0，则产品（32，48，39）在一行数据一共出现5330次

完整代码见附录2

附录1：

|  |
| --- |
| *''' k Means Clustering for Finding Similar Time Series in Sales Transaction Data '''* **from** numpy **import** \* **import** pandas **as** pd **from** pandas **import** DataFrame,Series  *#加载数据* **def** loadDataSet(fileName): *# general function to parse tab -delimited floats* fr = pd.read\_csv(fileName)  *#切分数据* examDf = DataFrame(fr)  new\_examDf1 = examDf.ix[:, :1]  new\_examDf2 = examDf.ix[:, 55:]  result = new\_examDf1.join(new\_examDf2)  dataset = result.ix[ :, 1:].values   **return** dataset  *#计算距离* **def** distEclud(vecA, vecB):  **return** sqrt(sum(power(vecA - vecB, 2))) *# la.norm(vecA-vecB)  #初始化k个点* **def** randCent(dataSet, k):  n = shape(dataSet)[1]  centroids = mat(zeros((k, n))) *# create centroid mat* **for** j **in** range(n): *# create random cluster centers, within bounds of each dimension* minJ = min(dataSet[:, j])  rangeJ = float(max(dataSet[:, j]) - minJ)  centroids[:, j] = mat(minJ + rangeJ \* random.rand(k, 1))  **return** centroids  *#普通Kmeans* **def** kMeans(dataSet, k, distMeas=distEclud, createCent=randCent):  m = shape(dataSet)[0]  clusterAssment = mat(zeros((m, 2))) *# create mat to assign data points  # to a centroid, also holds SE of each point* centroids = createCent(dataSet, k)  clusterChanged = **True** i = 0  **while** (clusterChanged **or** i < 100):  clusterChanged = **False  for** i **in** range(m): *# for each data point assign it to the closest centroid* minDist = inf;  minIndex = -1  **for** j **in** range(k):  distJI = distMeas(centroids[j, :], dataSet[i, :])  **if** distJI < minDist:  minDist = distJI;  minIndex = j  **if** clusterAssment[i, 0] != minIndex: clusterChanged = **True** clusterAssment[i, :] = minIndex, minDist \*\* 2  print(centroids)  **for** cent **in** range(k): *# recalculate centroids* ptsInClust = dataSet[nonzero(clusterAssment[:, 0].A == cent)[0]] *# get all the point in this cluster* centroids[cent, :] = mean(ptsInClust, axis=0) *# assign centroid to mean* i += 1  **return** centroids, clusterAssment  *#二分K-均值算法* **def** biKmeans(dataSet, k, distMeas=distEclud):  m = shape(dataSet)[0]  clusterAssment = mat(zeros((m, 2)))  centroid0 = mean(dataSet, axis=0).tolist()[0]  centList = [centroid0] *# create a list with one centroid* **for** j **in** range(m): *# calc initial Error* clusterAssment[j, 1] = distMeas(mat(centroid0), dataSet[j, :]) \*\* 2  **while** (len(centList) < k):  lowestSSE = inf  **for** i **in** range(len(centList)):  ptsInCurrCluster = dataSet[nonzero(clusterAssment[:, 0].A == i)[0],  :] *# get the data points currently in cluster i* centroidMat, splitClustAss = kMeans(ptsInCurrCluster, 2, distMeas)  sseSplit = sum(splitClustAss[:, 1]) *# compare the SSE to the currrent minimum* sseNotSplit = sum(clusterAssment[nonzero(clusterAssment[:, 0].A != i)[0], 1])  print(**"sseSplit, and notSplit: "**, sseSplit, sseNotSplit)  **if** (sseSplit + sseNotSplit) < lowestSSE:  bestCentToSplit = i  bestNewCents = centroidMat  bestClustAss = splitClustAss.copy()  lowestSSE = sseSplit + sseNotSplit  bestClustAss[nonzero(bestClustAss[:, 0].A == 1)[0], 0] = len(centList) *# change 1 to 3,4, or whatever* bestClustAss[nonzero(bestClustAss[:, 0].A == 0)[0], 0] = bestCentToSplit  print(**'the bestCentToSplit is: '**, bestCentToSplit)  print(**'the len of bestClustAss is: '**, len(bestClustAss))  centList[bestCentToSplit] = bestNewCents[0, :].tolist()[0] *# replace a centroid with two best centroids* centList.append(bestNewCents[1, :].tolist()[0])  clusterAssment[nonzero(clusterAssment[:, 0].A == bestCentToSplit)[0],  :] = bestClustAss *# reassign new clusters, and SSE* **return** mat(centList), clusterAssment  *#测试kmeans* **def** test\_Kmeans():  date = loadDataSet(**'Sales\_Transactions\_Dataset\_Weekly.csv'**)  *#v为每个簇包含的产品数，d为总共产品数，k为总共的簇的个数* v = 4;d = len(date)  k = int(d/v)  centroids, clusterAssment = kMeans(date, k,distEclud,randCent)   file = open(**'cluster.txt'**, **'w'**, encoding=**'utf-8'**)  *#写入数据到txt文本* **for** i **in** range(k):  file.write(**"第{}组："**.format(i)+**"\n"**)  **for** j **in** range(len(clusterAssment)):  **if**(clusterAssment[j,0] == i):  file.write(**"产品名称：P{} 距离为：{:.4f}"**.format(j,clusterAssment[i,1]) +**"\n"**)  **if** \_\_name\_\_ == **'\_\_main\_\_'**:  test\_Kmeans() |

附录2：

|  |
| --- |
| *''' 注：用来处理retail数据集，modify 2018-10-28 '''* **from** numpy **import** \* **import** pandas **as** pd **from** pandas **import** DataFrame,Series  *#定义树节点* **class** treeNode:  **def** \_\_init\_\_(self, nameValue, numOccur, parentNode):  self.name = nameValue  self.count = numOccur  self.nodeLink = **None** self.parent = parentNode *# needs to be updated* self.children = {}   **def** inc(self, numOccur):  self.count += numOccur   **def** disp(self, ind=1):  print(**' '** \* ind, self.name, **' '**, self.count)  **for** child **in** self.children.values():  child.disp(ind + 1)  *#创建树* **def** createTree(dataSet, minSup=1): *# create FP-tree from dataset but don't mine* headerTable = {}  *# go over dataSet twice* **for** trans **in** dataSet: *# first pass counts frequency of occurance* **for** item **in** trans:  headerTable[item] = headerTable.get(item, 0) + dataSet[trans]  headerTableCopy = headerTable.copy()  **for** k **in** headerTableCopy.keys(): *# remove items not meeting minSup* **if** headerTable[k] < minSup:  **del** (headerTable[k])  freqItemSet = set(headerTable.keys())  *# print 'freqItemSet: ',freqItemSet* **if** len(freqItemSet) == 0: **return None**, **None** *# if no items meet min support -->get out* **for** k **in** headerTable:  headerTable[k] = [headerTable[k], **None**] *# reformat headerTable to use Node link  # print 'headerTable: ',headerTable* retTree = treeNode(**'Null Set'**, 1, **None**) *# create tree* **for** tranSet, count **in** dataSet.items(): *# go through dataset 2nd time* localD = {}  **for** item **in** tranSet: *# put transaction items in order* **if** item **in** freqItemSet:  localD[item] = headerTable[item][0]  **if** len(localD) > 0:  orderedItems = [v[0] **for** v **in** sorted(localD.items(), key=**lambda** p: p[1], reverse=**True**)]  updateTree(orderedItems, retTree, headerTable, count) *# populate tree with ordered freq itemset* **return** retTree, headerTable *# return tree and header table* **def** updateTree(items, inTree, headerTable, count):  **if** items[0] **in** inTree.children: *# check if orderedItems[0] in retTree.children* inTree.children[items[0]].inc(count) *# incrament count* **else**: *# add items[0] to inTree.children* inTree.children[items[0]] = treeNode(items[0], count, inTree)  **if** headerTable[items[0]][1] == **None**: *# update header table* headerTable[items[0]][1] = inTree.children[items[0]]  **else**:  updateHeader(headerTable[items[0]][1], inTree.children[items[0]])  **if** len(items) > 1: *# call updateTree() with remaining ordered items* updateTree(items[1::], inTree.children[items[0]], headerTable, count)   **def** updateHeader(nodeToTest, targetNode): *# this version does not use recursion* **while** (nodeToTest.nodeLink != **None**): *# Do not use recursion to traverse a linked list!* nodeToTest = nodeToTest.nodeLink  nodeToTest.nodeLink = targetNode   **def** ascendTree(leafNode, prefixPath): *# ascends from leaf node to root* **if** leafNode.parent != **None**:  prefixPath.append(leafNode.name)  ascendTree(leafNode.parent, prefixPath)   **def** findPrefixPath(basePat, treeNode): *# treeNode comes from header table* condPats = {}  **while** treeNode != **None**:  prefixPath = []  ascendTree(treeNode, prefixPath)  **if** len(prefixPath) > 1:  condPats[frozenset(prefixPath[1:])] = treeNode.count  treeNode = treeNode.nodeLink  **return** condPats   **def** mineTree(inTree, headerTable, minSup, preFix, freqItemList):  bigL = [v[0] **for** v **in** sorted(headerTable.items(), key=**lambda** p: p[0])] *# (sort header table)* **for** basePat **in** bigL: *# start from bottom of header table* newFreqSet = preFix.copy()  newFreqSet.add(basePat)  *# print 'finalFrequent Item: ',newFreqSet #append to set* freqItemList.append(newFreqSet)  condPattBases = findPrefixPath(basePat, headerTable[basePat][1])  *# print 'condPattBases :',basePat, condPattBases  # 2. construct cond FP-tree from cond. pattern base* myCondTree, myHead = createTree(condPattBases, minSup)  *# print 'head from conditional tree: ', myHead* **if** myHead != **None**: *# 3. mine cond. FP-tree* print(**'conditional tree for: '**, newFreqSet)  myCondTree.disp(1)  mineTree(myCondTree, myHead, minSup, newFreqSet, freqItemList)  *#测试数据* **def** loadSimpDat():  simpDat = [[**'r'**, **'z'**, **'h'**, **'j'**, **'p'**],  [**'z'**, **'y'**, **'x'**, **'w'**, **'v'**, **'u'**, **'t'**, **'s'**],  [**'z'**],  [**'r'**, **'x'**, **'n'**, **'o'**, **'s'**],  [**'y'**, **'r'**, **'x'**, **'z'**, **'q'**, **'t'**, **'p'**],  [**'y'**, **'z'**, **'x'**, **'e'**, **'q'**, **'s'**, **'t'**, **'m'**]]  **return** simpDat  *#将每一条数据记录初始化* **def** createInitSet(dataSet):  retDict = {}  **for** trans **in** dataSet:  retDict[frozenset(trans)] = 1  **return** retDict *#加载，预处理数据* **def** loadDat(fileName):  usecols = []  **for** i **in** range(30):  usecols.append(i)  fr = pd.read\_csv(fileName, usecols = usecols)  *# 切分数据* examDf = DataFrame(fr)  examDf = examDf.fillna(-1)  examDf = examDf.values  *#去掉excel中的空* re\_data = []  **for** arry **in** examDf:  arry\_T = []  **for** elem **in** arry:  **if**(elem == -1):  **break  else**:  arry\_T.append(elem)  re\_data.append(arry\_T) **return** re\_data   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  file = **'retail.csv'** simDat = loadDat(file)  *#simDat = loadSimpDat()* initSet = createInitSet(simDat)  myFPtree, myHeaderTab = createTree(initSet, 5000) *#阈值取了5000，即找5000以上的强关联性* freqItems = []  mineTree(myFPtree, myHeaderTab, 5000, set([]), freqItems)  **for** x **in** freqItems:  print  x  print(**"FP树为："**)  myFPtree.disp() |