关联规则挖掘任务

1 数据认知

本次任务所提供的数据集为“groceries.csv”和“处方数据.xls”。

从数据集的描述上我们可以得知“groceries”是从真实世界杂货店的一个月操作收集到的数据集，我们可以通过关联规则挖掘算法分析得到商品间的关系，进而提供促销建议。

而“处方数据”则是与处方相关的信息，我们可以通过关联规则挖掘得知不同药品间的关系。其中处方 ID 与药品用量在本次任务中为多余的信息，在预处理时我们需要将其去除。

2 数据预处理

观察到两个数据集存储的数据格式不同，分别作了不同的预处理。

1) groceries

groceries.csv 中每项由所购买商品的名称构成，直接对其进行读取即可。

```
file_name = 'data/groceries.csv'

# 读取‘groceries.csv’
def loadDataSet1():
    dataSet = []
    with open(file_name) as file_obj:
        for content in file_obj:
            content = content.rstrip().split(',')
            dataSet.append(content)
        return dataSet
```

2) 处方数据

本次任务选择的语言为 python，为了方便在 python 中读取数据，首先将“处方数据.xls”用 Excel 表格将说明性文字删除，并**另存为 csv 后缀名文件**。

处方数据中我们所需的信息仅为每个处方包含的药品名称，需要对**多余的数据进行删除处理**。

```
file2_name = 'data/处方数据.csv'

# 读取‘处方数据.csv’
def loadDataSet2():
  dataSet = []
  with open(file2_name) as file2_obj:
    for content in file2_obj:
      content = content.rstrip().split(',')[1]
```

```
        content = content.split(':;')
        itemList = []
        for item in content:
            item = item.split(':')[0]
            itemList.append(item)
        if itemList[0] != 'NULL':
            dataSet.append(itemList[:len(itemList) - 1])
        return dataSet
```

通过对数据进行预处理，我们实现了数据清洗任务，可以在算法中直接调用 loadDataSet1()和 loadDataSet2()函数来获得标准化的数据集。

3 算法模型

常用的关联规则挖掘算法为 Apriori 算法和 FP-growth 算法。

3.1 Apriori 算法

在关联分析中，频繁项集的挖掘最常用到的就是 Apriori 算法。Apriori 算法是一种挖掘关联规则的频繁项集算法，其核心是基于两阶段频集思想的递推算法。

Apriori 原理：

如果某个项集是频繁的，那么它的所有子集也是频繁的。如果某一个项集是非频繁的，那么它的所有超集也是非频繁的。所以在得知某些项集是非频繁之后，不需要计算该集合的超集，有效地避免项集数目的指数增长，从而在合理时间内计算出频繁项集。

主要算法步骤：

连接：利用已经找到的 k 个项的频繁项集，通过两两连接得出 k+1 项候选集。

剪枝：找出候选集中的非频繁项集，剪枝去掉，避免项集数目的指数增长。

详细步骤及代码实现：

1) 获取候选 1 项集：

```
# 获取候选 1 项集，dataSet 为事务集
def findCandidate1(dataSet):
    candidate1 = []
    for line in dataSet:
        for item in line:
            if not [item] in candidate1:
                candidate1.append([item])
    candidate1.sort()
    return list(map(frozenset, candidate1)) # 返回 list，每个元素为 frozenset
```

2) 连接：

```
# 通过频繁 k 项集生成候选项集 k+1。
def candidateGenerate(ListK, k):
```

```
    ListK = list(map(frozenset, ListK))
    frequentList = []
    lenListK = len(ListK)
    for i in range(lenListK):
        for j in range(i + 1, lenListK):
            list1 = list(ListK[i])[:k - 2]
            list2 = list(ListK[j])[:k - 2]
            list1.sort()
            list2.sort()
            if list1 == list2:
                frequentList.append(ListK[i] | ListK[j])  # 取出并比较两个集合的前 k-1 个元素，若相同则连接
    return frequentList
```

3) 减枝：

```
# 找出候选集中的频繁项集
def scanDataSet(dataSet, candidateK, minSupport):
    countDictionary = {}  # 候选项计数字典
    for line in dataSet:
        for candidate in candidateK:
            if candidate.issubset(line):
                countDictionary[candidate] = countDictionary.get(candidate, 0) + 1  # 计数字典
    itemNum = float(len(dataSet))
    frequentList = []
    supportDictionary = {}
    for key in countDictionary:
        support = countDictionary[key] / itemNum
        if support >= minSupport:
            frequentList.insert(0, key)  # 将频繁项集插入返回列表的首部
            supportDictionary[key] = support
    return frequentList, supportDictionary  # 返回频繁项集及其支持度
```

4) 递归：

```
# 获取事务集中的所有的频繁项集
def apriori(dataSet, minSupport):
    candidate1 = findCandidate1(dataSet)  # 从事务集中获取候选 1 项集
    dataSet = list(map(set, dataSet))  # 将事务集转化为 list，每个元素为 set
    frequentList1, supportDictionary = scanDataSet(dataSet, candidate1, minSupport)  # 获取频繁 1 项集和对应的支持度
    frequentList = [frequentList1]  # 初始化频繁集列表
    k = 1
    while len(frequentList[k - 1]) > 0:
        candidateK = candidateGenerate(frequentList[k - 1], k + 1)  # 根据 K 频繁项集生成 K+1 候选项集
        frequentListK, supportK = scanDataSet(dataSet, candidateK, minSupport)  # 得到 K+1 频繁项集及其支持度
```

```
    frequentList.append(frequentListK)  #  添加新频繁项集
    supportDictionary.update(supportK)  #  更新支持度
    k += 1
        return frequentList, supportDictionary
```

这样，我们便可以得到频繁集及其支持度，最后通过输出函数进行处理获得其有效关联规则。

5) 输出“有趣”规则：

```
# ResultOutput
def outputResult(out_file_name, supportDictionary, minConfidence):
  with open(out_file_name, 'w') as f:
    for key in supportDictionary:
      result = []
      if len(key) > 1:
        for item in key:
          result.append(item)
        print("FrequentSet:" + str(result))
        f.write("FrequentSet:" + str(result) + '\n')
        supportKey = supportDictionary.get(key)
        print("Support: " + str(format(supportKey, '.5f')))
        f.write("Support: " + str(format(supportKey, '.5f')) + '\n')

        for item in result:
          supportItem = supportDictionary.get(frozenset([item]))
          confidence = supportKey / supportItem
          if confidence >= minConfidence:
            tempList = result.copy()
            tempList.remove(item)
            print(str([item]) + '->' + str(tempList), end='  ')
            print("Confidence: " + str(format(confidence, '.5f')))
            f.write(str([item]) + '->' + str(tempList) + '  ')
            f.write("Confidence: " + str(format(confidence, '.5f')) + '\n')
        print('')
            f.write('\n')
```

在本文测试部分将给出主函数和输出结果。

Apriori 算法局限：

Apriori 算法是一种先产生候选项集再检验是否频繁的“产生-测试”的方法，这种方法有种弊端：当数据集很大的时候，需要不断扫描数据集造成运行效率很低。所以结合 Hash（散列）和事务压缩等技巧，我们可以对 Apriori 算法进行改进，以提升其处理效率。

3.2 Apriori 改进算法 —— 基于 Hash 的 Apriori 算法

基于散列优化的 Apriori 算法，主要用于缩小候选集项集个数。

Hash 改进思想：

Hash 算法生效于 Apriori 算法的剪枝过程中。在第 k 次扫描时，生成每个事务的 k+1 项集，代入一个 Hash 函数中，生成一个 Hash 表，同时记录每个桶中元素个数。

当生成 k+1 候选集时，将 k 项频繁集自然连接产生的结果代入上述 Hash 函数若落的哈希桶中，若计数小于最小支持阈值，则该元素必定不为频繁项集，可以将之过滤不加入 Ck+1 中。

主要改进步骤及代码：

1） 设计哈希函数

```
# 哈希函数
def hashFunc(x, y, order):
  return (131*(order[x]) + order[y]) % 77
```

2) 减枝步骤中进行哈希桶映射

```
# 在第 k 次扫描时，生成每个事务的 k+1 项集，代入一个 Hash 函数中，生成一个 Hash 表，同时记录每个桶中元素个数。
def scanDataSet(dataSet, candidateK, minSupport):
  countDictionary = {} # 候选项计数字典
  hashTable = {}
  order = {}     # 候选项映射字典
  i=1
  for candidate in candidateK:
    order[candidate] = order.get(candidate, 0) + i
    i += 1

  for line in dataSet:
    candidateLine = []
    for candidate in candidateK:
      if candidate.issubset(line):
        candidateLine.append(candidate)
        countDictionary[candidate] = countDictionary.get(candidate, 0) + 1 # 计数字典

    if len(candidateLine) > 1:
      for i in range(len(candidateLine)):
        for j in range(i + 1, len(candidateLine)):
          hashValue = hashFunc(candidateLine[i], candidateLine[j], order)
          hashTable[hashValue] = hashTable.get(hashValue, 0) + 1          # 哈希桶映射

  itemNum = float(len(dataSet))
  frequentList = []
  supportDictionary = {}
  for key in countDictionary:
```

```
    support = countDictionary[key] / itemNum
    if support >= minSupport:
      frequentList.insert(0, key)  # 将频繁项集插入返回列表的首部
      supportDictionary[key] = support
  return frequentList, supportDictionary, hashTable, order  # 返回频繁项集及其支持度
```

3) 在生成候选集的过程中对不满足最小支持阈值的项进行过滤

```
# 通过频繁 k 项集生成候选项集 k+1。
def candidateGenerate(ListK, k, hashTable, minSup, order):
  ListK = list(map(frozenset, ListK))
  candidateList = []
  lenListK = len(ListK)
  for i in range(lenListK):
    for j in range(i + 1, lenListK):
      L1 = list(ListK[i])[:k - 2]
      L2 = list(ListK[j])[:k - 2]
      L1.sort()
      L2.sort()
      if L1 == L2:
        hashValue = hashFunc(ListK[i], ListK[j], order)   # 在这里对不满足条件的候选项进行过滤
        candidate = ListK[i] | ListK[j]
        if hashTable[hashValue] >= minSup:
          candidateList.append(candidate)
  return candidateList
```

4) 在递归函数中传字典参数

```
# 获取事务集中的所有的频繁项集
def aprioriHash(dataSet, minSupport):
  candidate1 = findCandidate1(dataSet)  # 从事务集中获取候选 1 项集
  minSup = 0.6*len(dataSet)
  dataSet = list(map(set, dataSet))  # 将事务集转化为 list，每个元素为 set
  frequentList1, supportDictionary, hashTable, order = scanDataSet(dataSet, candidate1, minSupport)  # 获取频繁 1 项集和对应的支持度
  frequentList = [frequentList1]  # 初始化频繁集列表
  k = 1
  while len(frequentList[k - 1]) > 0:
    candidateK, count = candidateGenerate(frequentList[k - 1], k + 1, hashTable, minSup, order)  # 根据 K 频繁项集生成 K+1 候选项集
    frequentListK, supportK, hashTable, order = scanDataSet(dataSet, candidateK, minSupport)  # 得到 K+1 频繁项集及其支持度
    frequentList.append(frequentListK)  # 添加新频繁项集
    supportDictionary.update(supportK)  # 更新支持度
    k += 1
  return frequentList, supportDictionary
```

3.3 Apriori 改进算法——基于事务缩减的 Apriori 算法

事务缩减即通过减少数据库中需要考虑事务的数目来提升时间效率。

事务缩减改进思想：

如果一个事务中含有 k+1 频繁项集，那么该项目集中的每一个元素，至少都在 k 项候选集中出现至少 k 次，当其存在于某一个(k+1) 频繁项集的时候，k 频繁项集中必须包含该 k+1 项目集中的(k+1)个 k 频繁项集。

实现细节及代码：

事务缩减主要发生在候选集产生阶段：

```
# 通过频繁 k 项集生成候选项集 k+1。
def candidateGenerate(ListK, k, hashTable, minSup, order, supportDictionary, lenOfData):
  ListK = list(map(frozenset, ListK))
  for item in ListK:
    for element in item:
      element = frozenset([element])
      numOfElement = supportDictionary[element]*lenOfData
      if numOfElement <= k:
        ListK.remove(item)    # 若对于候选项集 k+1 中元素而言少于 k+1 个，则从 List 去掉。
        continue
  candidateList = []
  lenListK = len(ListK)
  count = 0
  for i in range(lenListK):
    for j in range(i + 1, lenListK):
      L1 = list(ListK[i])[:k - 2]
      L2 = list(ListK[j])[:k - 2]
      L1.sort()
      L2.sort()
      if L1 == L2:
        hashValue = hashFunc(ListK[i], ListK[j], order)
        candidate = ListK[i] | ListK[j]

        if hashTable[hashValue] >= minSup:
          candidateList.append(candidate)
        else:
          print("remove", end=" ")
          print(candidate)
      return candidateList, count
```

3.4 FP-growth 算法

FP-growth 算法很好的解决了 Apriori 算法需要多次扫描数据集才能找到频繁项集的问题。

FP-growth 原理：

它的思路是把数据集中的事务映射到一棵 FP-Tree 上面，再根据这棵树找出频繁项集。FP-Tree 的构建过程只需要扫描两次数据集，这样可以大幅减少需要的时间。

主要算法步骤：

1) 构建 FP 树：对数据集扫描两次，第一次对所有元素项出现次数进行计数，第二遍的扫描只考虑那些频繁元素。

2) 从 FP 树中挖掘频繁项集：思想与 Apriori 算法大致一样，从单元素项集开始，逐步构建更大的集合，但不需要用到原始数据集。

详细步骤及代码实现：

1) FP 树数据类型：

```
# FP 树节点类
class treeNode:
    def __init__(self, nameValue, times, parentNode):
        self.name = nameValue  #  节点名称
        self.count = times  #  出现次数
        self.nodeBother = None  #  指向下一个同级节点的指针
        self.parent = parentNode  #  指向父节点的指针，在构造时初始化为给定值
        self.children = {}  #  指向子节点的字典，键为子节点的元素名，值为指向子节点的指针

    def addCount(self, times):
        self.count += times
```

2) 两次扫描数据集，创建 FP 树：

```
#  创建 FP 树
def createTree(dataSet, minSup):
    #  第一次遍历数据集，创建头指针表
    headerTable = {}
    for line in dataSet:
        for item in line:
            headerTable[item] = headerTable.get(item, 0) + dataSet[line]
    #  移除不满足最小支持度的元素项
    keys = list(headerTable.keys())
    for item in keys:
        if headerTable[item] < minSup:
            del (headerTable[item])
    freqItemSet = set(headerTable.keys())
    if len(freqItemSet) == 0:
        return None, None   #  若不存在任何频繁项集，返回空

    for item in headerTable:
```

```
    headerTable[item] = [headerTable[item], None]     # 在头指针表中添加用于存放指向兄弟节点指针
  rootTree = treeNode('Root', 0, None) # 根节点

  # 第二次遍历数据集，创建 FP 树
  for data, count in dataSet.items():
    countDictionary = {} # 记录频繁 1 项集的全局频率，用于排序
    for item in data:
      if item in freqItemSet: # 只考虑频繁项
        countDictionary[item] = headerTable[item][0]
    if len(countDictionary) > 0:
      orderedItems = [v[0] for v in sorted(countDictionary.items(), key=lambda p: p[1], reverse=True)] #
排序
      updateTree(orderedItems, rootTree, headerTable, count) # 更新 FP 树
  return rootTree, headerTable
```

其中所用函数 updateTree()为：

```
# 根据排序后的频繁项更新 FP 树
def updateTree(items, node, headerTable, count):
  if items[0] in node.children:
    node.children[items[0]].addCount(count) # 有该项时计数值+1
  else:
    node.children[items[0]] = treeNode(items[0], count, node) # 没有时则创建一个新节点
    if headerTable[items[0]][1] is None: # 如果是第一次出现，则在头指针表中增加对该节点的指向
      headerTable[items[0]][1] = node.children[items[0]]
    else:
      updateHeader(headerTable[items[0]][1], node.children[items[0]])
  if len(items) > 1:
    # 对剩下的元素项迭代调用 updateTree 函数
    updateTree(items[1::], node.children[items[0]], headerTable, count)

# 更新头指针块
def updateHeader(nodeToTest, targetNode):
  while nodeToTest.nodeBother is not None:
    nodeToTest = nodeToTest.nodeBother
      nodeToTest.nodeBothe
```

3) 挖掘频繁项：

```
# 递归查找频繁项集
# freqItemList 请传入一个空列表（[]），将用来储存生成的频繁项集。
def findFreqItem(headerTable, minSup, preFix, freqItemList):
  # 对频繁项按出现的数量进行排序
  sorted_headerTable = sorted(headerTable.items(), key=lambda p: p[1][0]) # 返回重新排序的列表
  freqSet = [v[0] for v in sorted_headerTable] # 获取频繁项
```

```
  for base in freqSet:
    newFreqSet = preFix.copy() # 新的频繁项集
    newFreqSet.add(base) # 当前前缀添加一个新元素
    freqItemList.append(newFreqSet) # 所有的频繁项集列表
    condPattBases = findPrefixPath(headerTable[base][1]) # 获取条件模式基
    rootTree, head = createTree(condPattBases, minSup) # 创建条件 FP 树
    if head is not None:
        findFreqItem(head, minSup, newFreqSet, freqItemList) # 递归直到不再有元素
```

其中所用函数 findPrefixPath()为：

```
# 返回条件模式基，用一个字典表示，键为前缀路径，值为计数值。
def findPrefixPath(treeNode):
  condPaths = {} # 存储条件模式基
  while treeNode is not None:
    prefixPath = [] # 用于存储前缀路径
    ascendTree(treeNode, prefixPath) # 生成前缀路径
    if len(prefixPath) > 1:
      condPats[frozenset(prefixPath[1:])] = treeNode.count # 出现的数量就是当前叶子节点的数量
    treeNode = treeNode.nodeBother # 遍历下一个相同元素
  return condPaths

# 递归添加其父节点。
def ascendTree(leafNode, prefixPath):
  if leafNode.parent is not None:
    prefixPath.append(leafNode.name)
     # prefixPath 就是一条从 treeNode 到根节点的路径，但不包含根节点
     ascendTree(leafNode.parent, prefixPath)
```

这样 freqItemList 便会返回所查找的频繁项集。

4) 获取频繁项集支持度：

```
# 获取频繁集支持度
def getSupportData(freqItems,dataSet)
  freqItems = list(map(frozenset, freqItems))
  countDictionary = {} # 记录每个候选项的个数
  for line in dataSet:
    for item in freqItems:
      if item.issubset(line):
        countDictionary[item] = countDictionary.get(item, 0) + 1 # 计数字典
  supportDictionary = {}
  for key in countDictionary:
    supportDictionary[key] = countDictionary[key] / len(dataSet)
   return supportDictionary
```

5) fpTree 算法，对之前步骤进行合并：

```
# fpTree 算法
def fpTree(dataSet, minSupport):
    minSup = int(len(dataSet) * minSupport)
    initSet = normDataSet(dataSet)  # 转化为符合格式的事务集
    rootFPtree, headerFPtree = createTree(initSet, minSup)  # 初始化 FP 树，得到头指针表

    preFix = set([])  # 用于存储前缀
    freqItems = []  # 用于存储频繁项集
    findFreqItem(headerFPtree, minSup, preFix, freqItems)  # 获取频繁项集
    supportDictionary = getSupportData(freqItems, dataSet)
     return freqItems, supportDictionary
```

接下来调用和 Apriori 算法一样的输出函数 outputResult()即可，代码已在 3.1 给出。

4 实验测试

本文分别在任务所提供的杂货店数据集和处方数据集上对上述算法进行了测试。

4.1 主函数

所用的主函数可以统一为：

```
if __name__ == '__main__':
    # 默认最小支持度和最小置信度
    minSupport = 0.04
    minConfidence = 0.3

    # 加载数据集 1，即 groceries，可替换
    dataSet = loadDataSet1()

    # 调用算法 fpTree，可替换
    frequentList, supportDictionary = fpTree(dataSet, minSupport)

     # 输出文件名，可替换
    out_file_name = "fp_result.txt"
     outputResult(out_file_name, supportDictionary, minConfidence)
```

4.2 输出结果

采用不同的关联规则挖掘算法，其输出结果不会发生变化，所以本文给出每个数据集上一种算法的输出结果。

1) Groceries 挖掘结果

设置最小支持度为 0.04，最小置信度为 0.3，结果未输出单项频繁集。

```
FrequentSet:['other vegetables', 'whole milk']
Support: 0.07483
```

```
['other vegetables']->['whole milk']  Confidence: 0.38676

FrequentSet:['yogurt', 'whole milk']
Support: 0.05602
['yogurt']->['whole milk']  Confidence: 0.40160

FrequentSet:['rolls/buns', 'other vegetables']
Support: 0.04260

FrequentSet:['whole milk', 'tropical fruit']
Support: 0.04230
['tropical fruit']->['whole milk']  Confidence: 0.40310

FrequentSet:['root vegetables', 'other vegetables']
Support: 0.04738
['root vegetables']->['other vegetables']  Confidence: 0.43470

FrequentSet:['root vegetables', 'whole milk']
Support: 0.04891
['root vegetables']->['whole milk']  Confidence: 0.44869

FrequentSet:['rolls/buns', 'whole milk']
Support: 0.05663
['rolls/buns']->['whole milk']  Confidence: 0.30790

FrequentSet:['other vegetables', 'yogurt']
Support: 0.04342
['yogurt']->['other vegetables']  Confidence: 0.31122
```

从中我们可以发现一些有趣的规则，例如乳制品与蔬菜水果经常被一起购买。

2) 处方数据挖掘结果（节选）

设置最小支持度为 0.06，最小置信度为 0.3，由于输出过多，本文仅给出了部分结果，完整结果在附件的文档中给出。

```
FrequentSet:['14197', '15190']
Support: 0.12657
['14197']->['15190']  Confidence: 0.59730
['15190']->['14197']  Confidence: 0.38342

FrequentSet:['14197', '14984']
Support: 0.09201
['14197']->['14984']  Confidence: 0.43422

FrequentSet:['14197', '15204']
Support: 0.06239
```

['15204']->['14197'] Confidence: 0.37624

FrequentSet:['14197', '14717']
Support: 0.06578
['14197']->['14717'] Confidence: 0.31044
['14717']->['14197'] Confidence: 0.70479

FrequentSet:['15180', '14922', '14984']
Support: 0.06786
['15180']->['14922', '14984'] Confidence: 0.38116
['14922']->['15180', '14984'] Confidence: 0.50133

FrequentSet:['14984', '14132', '15180']
Support: 0.06461
['15180']->['14984', '14132'] Confidence: 0.36288

4.3 算法时间效率对比

本文在处方数据集上以相同参数对比并分析了不同算法的时间效率。

1) Apriori 算法

Name	Call Count	Time (ms) ▼	Own Time (ms)
main.py	1	23381 100.0%	22 0.1%
apriori	1	22444 96.0%	100 0.4%
scanDataSet	4	19377 82.9%	11114 47.5%
<method 'issubset' of 'frozense	94534338	8047 34.4%	8047 34.4%
findCandidate1	1	2962 12.7%	2962 12.7%
loadDataSet2	1	669 2.9%	403 1.7%
_find_and_load_unlocked	147	240 1.0%	0 0.0%
_find_and_load	147	240 1.0%	1 0.0%
_load_unlocked	140	239 1.0%	0 0.0%
exec_module	124	238 1.0%	0 0.0%
_call_with_frames_removed	202	237 1.0%	0 0.0%
Apriori.py	1	236 1.0%	0 0.0%
__init__.py	1	231 1.0%	0 0.0%
<method 'get' of 'dict' objects	1827201	215 0.9%	215 0.9%
<method 'split' of 'str' objects:	801412	198 0.8%	198 0.8%

Apriori 算法共耗时 23381 ms，我们可以注意到 scanDataSet 函数占据了总时长的 82.9%，这说明扫描数据集的开销是非常大的。

2) Apriori 算法改进——基于 Hash 的 Apriori 算法

Name	Call Count	Time (ms)	Own Time (ms)
scanDataSet	4	40380 90.6%	23678 53.1%
<method 'issubset' of 'frozens	97070754	8513 19.1%	8513 19.1%
hashFunc	22855606	5756 12.9%	5756 12.9%
findCandidate1	1	3156 7.1%	3156 7.1%
<method 'get' of 'dict' objects	24726198	2117 4.8%	2117 4.8%
loadDataSet2	1	670 1.5%	403 0.9%
<method 'append' of 'list' obje	2622377	220 0.5%	220 0.5%
<method 'split' of 'str' objects	801412	199 0.4%	199 0.4%
<built-in method builtins.len>	2239954	152 0.3%	152 0.3%
aprioriHash	1	43644 97.9%	104 0.2%
<built-in method nt.stat>	639	80 0.2%	80 0.2%
main.py	1	44562 100.0%	22 0.0%
get_data	124	21 0.0%	16 0.0%

我们从测试中发现，基于 Hash 的改进算法不仅没有提高时间效率，反倒大幅增加了运行时间。从代码调试的过程中我们发现，对于本任务所提供的实验数据而言，hash 算法并不能达到我们想要的减少候选集的效果，原因在于本数据适合的支持度太低，基本上 2 项候选集映射的所有哈希桶都远远超过了所设置的阈值。而 hash 所做的额外工作至少包含了一次扫描，以及 hash 函数映射，这些都增加了时间开销。经过分析，我们认为本任务所用数据集可能不适合 hash 算法。

3) Apriori 算法改进——基于事务压缩的 Apriori 算法

Name	Call Count	Time (ms)	Own Time (ms)
main.py	1	23077 100.0%	22 0.1%
aprioriReduce2	1	22162 96.0%	98 0.4%
scanDataSet	4	19116 82.8%	11024 47.8%
<method 'issubset' of 'frozens	96542334	7898 34.2%	7898 34.2%
findCandidate1	1	2944 12.8%	2944 12.8%
loadDataSet2	1	661 2.9%	397 1.7%
_find_and_load_unlocked	148	226 1.0%	0 0.0%
_find_and_load	148	226 1.0%	1 0.0%
_load_unlocked	141	224 1.0%	0 0.0%
exec_module	125	223 1.0%	0 0.0%
_call_with_frames_removed	205	221 1.0%	0 0.0%
Apriori.py	1	218 0.9%	0 0.0%
__init__.py	1	214 0.9%	0 0.0%
<method 'split' of 'str' objects	801412	196 0.8%	196 0.8%
<method 'get' of 'dict' objects	1859272	193 0.8%	193 0.8%
_handle_fromlist	834	163 0.7%	1 0.0%
<built-in method builtins.__imp	316	163 0.7%	0 0.0%

经过多次实验，我们发现事务压缩算法较原先的算法而言，在处方数据集上有轻微的提升效果，但并不显著，可能与数据特征有关联。

4) FP-Growth 算法

Name	Call Count	Time (ms) ▼	Own Time (ms)
test.py	1	4944 100.0%	12 0.2%
fpTree	1	4017 81.2%	0 0.0%
createTree	46	2311 46.7%	281 5.7%
updateTree	166992	1833 37.1%	222 4.5%
updateHeader	20658	1554 31.4%	1554 31.4%
getSupportData	1	1471 29.8%	389 7.9%
<method 'issubset' of 'frozenset' obje	2377890	1046 21.2%	1046 21.2%
loadDataSet2	1	688 13.9%	415 8.4%
_load_unlocked	138	223 4.5%	0 0.0%
_find_and_load_unlocked	145	223 4.5%	0 0.0%
_find_and_load	145	223 4.5%	1 0.0%
exec_module	122	223 4.5%	0 0.0%

FP-Growth 算法共耗时 4944 ms，近似为 Apriori 算法耗时的五分之一，也就是说时间效率上提升了五倍。

5 总结

通过本次数据挖掘任务，熟悉了数据挖掘的流程，并且掌握了关联规则挖掘中常用的 Apriori 算法和 FP-Growth 算法。在实验的过程中能够发现，真实的数据挖掘任务不仅需要能够使用合适的算法处理数据，还需要对数据有洞察力才能够获取数据背后隐藏的信息。