机器学习作业-FP-growth

软件 51 庞建业 2151601012

https://github.com/sherjy/Notes-RL

本次主要针对 FP-Growth 算法的实现的学习和复习

FP-growth 算法简介

FP-growth，即 Frequent Pattern Growth 模型，它是目前业界经典的频繁项集和关联规则挖掘的算法。相比于 Apriori 模型，FP-growth 模型只需要扫描数据库两次，极大得减少了数据读取次数并显著得提升了算法效率。

FP-growth 算法特点：

- 一种非常好的发现频繁项集算法。
- 基于 Apriori 算法构建,但是数据结构不同，使用叫做 FP 树的数据结构结构来存储集合。下面我们会介绍这种数据结构。

Apriori 算法需要多次扫描数据库，这就使得该算法本身不适合大数据量。本次作业的数据集是指定的，并且数据量较大，对于本次数据集非常不合适，复杂度很高，所以使用 FP-Growth 算法

已选择 1498 行，14 列

它通过构建 FP 树(即 Frequent Pattern Tree)这样的数据结构，巧妙得将数据存储在 FP 树中，只需要在构建 FP 树时扫描数据库两次，后续处理就不需要再访问数据库了。这种特性使得 FP-growth 算法比 Apriori 算法速度快。FP 树是一种前缀树，由频繁项的前缀构成，挖掘出频繁项集后，可以从频繁项集中进一步挖掘关联规则。

环境：

Ubuntu16.04

Atom+Anaconda3.6

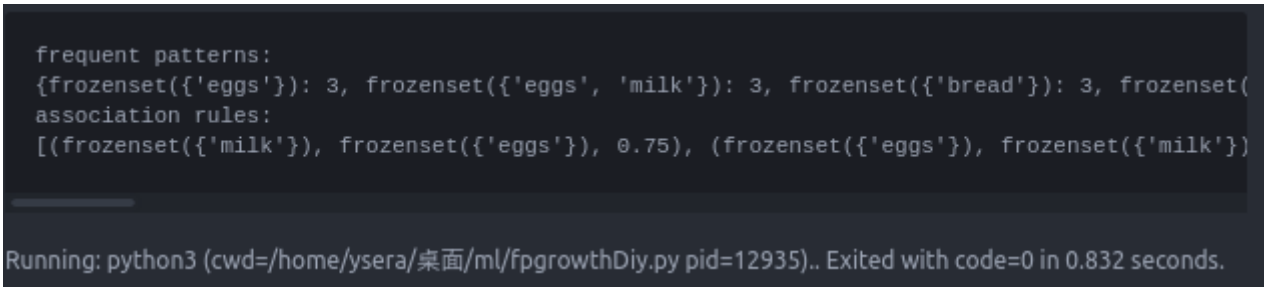
代码实现：

一、我们先来跑一个课堂上的小数据集的测试：

见文件夹中 fpgrowth_simple.py

```
def loadDataSet():
    dataSet = [['bread', 'milk', 'vegetable', 'fruit', 'eggs'],
               ['noodle', 'beef', 'pork', 'water', 'socks', 'gloves', 'shoes', 'rice'],
               ['socks', 'gloves'],
               ['bread', 'milk', 'shoes', 'socks', 'eggs'],
               ['socks', 'shoes', 'sweater', 'cap', 'milk', 'vegetable', 'gloves'],
               ['eggs', 'bread', 'milk', 'fish', 'crab', 'shrimp', 'rice']]
    return dataSet
```

对于这种数据量在两个算法的复杂度下都可行，输出为：



其中 **frequent patterns** 给出了这个小数据集的频繁项集和绝对支持度：

```
frequent patterns:
{frozenset({'eggs'}): 3,
frozenset({'eggs', 'milk'}): 3,
frozenset({'bread'}): 3,
frozenset({'eggs', 'bread'}): 3,
frozenset({'bread', 'milk'}): 3,
frozenset({'eggs', 'bread', 'milk'}): 3,
frozenset({'shoes'}): 3,
frozenset({'shoes', 'socks'}): 3,
frozenset({'gloves'}): 3,
frozenset({'gloves', 'socks'}): 3,
frozenset({'milk'}): 4,
frozenset({'socks'}): 4}
```

association rules 给出了关联规则：

```
association rules:
[(frozenset({'milk'}),
frozenset({'eggs'}), 0.75),
(frozenset({'eggs'}), frozenset({'milk'}),
1.0), (frozenset({'bread'}),
frozenset({'eggs'}), 1.0),
(frozenset({'eggs'}),
frozenset({'bread'}), 1.0),
(frozenset({'milk'}),
frozenset({'bread'}), 0.75),
(frozenset({'bread'}),
frozenset({'milk'}), 1.0),
(frozenset({'bread', 'milk'}),
frozenset({'eggs'}), 1.0),
(frozenset({'milk'}), frozenset({'eggs',
'bread'}), 0.75), (frozenset({'bread'}),
```

由于关联规则比较多，并不是随着数据项和频繁项集的增加而线性增长的，所以这里截图只放出部分

二、作业中的数据集比较大的情况 FP-growth 设计

见文件夹中 fpgrowth.py

（1）FP 树结构

首先，需要创建一个树形的数据结构，叫做 FP 树。清单 1 所示，该树结构包含结点名称 nodeName，结点元素出现频数 count，父节点 nodeParent，指向下一个相同元素的指针 nextSimilarItem，子节点集合 children。

```
14  class TreeNode:
15      def __init__(self, nodeName, count, nodeParent):
16          self.nodeName = nodeName
17          self.count = count
18          self.nodeParent = nodeParent
19          self.nextSimilarItem = None
20          self.children = {}
21
22      def increaseC(self, count):
23          self.count += count
```

（2）创建 FP 树

接着，用第一步构造出的数据结构来创建 FP 树。代码主要分为两层。第一层，扫描数据库，统计出各个元素的出现频数；第二层，扫描数据库，对每一条数据记录，将数据记录中不包含在频繁元素中的元素删除，然后将数据记录中的元素按出现频数排序。将数据记录逐条插入 FP 树中，不断更新 FP 树

```
25  def createFPTree(frozenDataSet, minSupport):
26      headPointTable = {}
27      for items in frozenDataSet:
28          for item in items:
29              headPointTable[item] = headPointTable.get(item, 0) + frozenDataSet[items]
30      headPointTable = {k:v for k,v in headPointTable.items() if v >= minSupport}
31      frequentItems = set(headPointTable.keys())
32      if len(frequentItems) == 0: return None, None
33
34      for k in headPointTable:
35          headPointTable[k] = [headPointTable[k], None]
36      fptree = TreeNode("null", 1, None)
37      for items,count in frozenDataSet.items():
38          frequentItemsInRecord = {}
```

（3）更新 FP 树

这里用到了递归的技巧。每次递归迭代中，处理数据记录中的第一个元素处理，如果该元素是 fptree 节点的子节点，则只增加该子节点的 count 树，否则，需要新创建一个 TreeNode 节点，然后将其赋给 fptree 节点的子节点，并更新头指针表关于下一个相同元素指针的信息。迭代的停止条件是当前迭代的

```
def updateFPTree(fptree, orderedFrequentItems, headPointTable, count):
    if orderedFrequentItems[0] in fptree.children:
        fptree.children[orderedFrequentItems[0]].increaseC(count)
    else:
        fptree.children[orderedFrequentItems[0]] = TreeNode(orderedFrequentItems[0], count, fptree)

        if headPointTable[orderedFrequentItems[0]][1] == None:
            headPointTable[orderedFrequentItems[0]][1] = fptree.children[orderedFrequentItems[0]]
        else:
            updateHeadPointTable(headPointTable[orderedFrequentItems[0]][1], fptree.children[orderedFrequentItems[0]])

    if(len(orderedFrequentItems) > 1):
        updateFPTree(fptree.children[orderedFrequentItems[0]], orderedFrequentItems[1::], headPointTable, count)
```

数据记录长度小于等于 1。

（4）挖掘频繁项集

开始挖掘频繁项集，这里也是递归迭代的思路。对于头指针表中的每一个元素，首先获取该元素结尾的所有前缀路径，然后将所有前缀路径作为新的数据集传入 createFPTree 函数中以创建条件 FP 树。然后对条件 FP 树对应的头指针表中的每一个元素，开始获取前缀路径，并创建新的条件 FP 树。这两步不断重复，直到条件 FP 树中只有一个元素为止。

```
def mineFPTree(headPointTable, prefix, frequentPatterns, minSupport):
    headPointItems = [v[0] for v in sorted(headPointTable.items(), key = lambda v:v[1][0])]
    if(len(headPointItems) == 0): return

    for headPointItem in headPointItems:
        newPrefix = prefix.copy()
        newPrefix.add(headPointItem)
        support = headPointTable[headPointItem][0]
        frequentPatterns[frozenset(newPrefix)] = support

        prefixPath = getPrefixPath(headPointTable, headPointItem)
        if(prefixPath != {}):
            conditionalFPtree, conditionalHeadPointTable = createFPTree(prefixPath, minSupport)
            if conditionalHeadPointTable != None:
                mineFPTree(conditionalHeadPointTable, newPrefix, frequentPatterns, minSupport)
```

（5）获取前缀路径

展示了获取前缀路径的步骤。对于每一个相同元素，通过父节点指针不断向上遍历，所得的路径就是该元素的前缀路径。

```
def getPrefixPath(headPointTable, headPointItem):
    prefixPath = {}
    beginNode = headPointTable[headPointItem][1]
    prefixs = ascendTree(beginNode)
    if((prefixs != [])):
        prefixPath[frozenset(prefixs)] = beginNode.count

    while(beginNode.nextSimilarItem != None):
        beginNode = beginNode.nextSimilarItem
        prefixs = ascendTree(beginNode)
        if (prefixs != []):
            prefixPath[frozenset(prefixs)] = beginNode.count
    return prefixPath
```

（6）挖掘关联规则

```
def rulesGenerator(frequentPatterns, minConf, rules):
    for frequentset in frequentPatterns:
        if(len(frequentset) > 1):
            getRules(frequentset,frequentset, rules, frequentPatterns, minConf)
```

展示了挖掘关联规则的代码，这里也用到了递归迭代的技巧。对于每一个频繁项集，构造所有可能的关联规则，然后对每一个关联规则计算置信度，输出置信度大于阈值的关联规则。

（7）输出结果

由于数据量太大，复杂度比较高，所以根据数据集大小设置 minSupport=100

输出结果见文件夹中 output.md

```
frequent patterns:
{frozenset({' hand soap'}): 356,
frozenset({' dishwashing
liquid/detergent', ' hand soap'}): 100,
frozenset({' flour', ' hand soap'}): 100,
frozenset({' hand soap', ' bagels'}): 101,
frozenset({' milk', ' hand soap'}): 102,
frozenset({' all- purpose', ' hand
soap'}): 103, frozenset({' waffles', '
hand soap'}): 106, frozenset({' sugar', '
hand soap'}): 106, frozenset({' mixes', '
hand soap'}): 114, frozenset({'
vegetables', ' hand soap'}): 211,
frozenset({' tortillas'}): 366,
frozenset({' dinner rolls', '
tortillas'}): 100, frozenset({' yogurt', '
tortillas'}): 102, frozenset({'
coffee/tea', ' tortillas'}): 102,
frozenset({' eggs', ' tortillas'}): 105,
```

FP-Growth 相关知识小结

（1）关联规则

关联规则是在频繁项集的基础上得到的。关联规则指由集合 A，可以在某置信度下推出集合 B。通俗来说，就是如果 A 发生了，那么 B 也很有可能会发生。举个例子，有关联规则如：{'鸡蛋', '面包'} -> {'牛奶'}，该规则的置信度是 0.9，意味着在所有买了鸡蛋和面包的客户中，有 90%的客户还买了牛奶。关联规则可以用来发现很多有趣的规律。这其中需要先阐明两个概念：支持度和置信度。

（2）支持度 Support

支持度指某频繁项集在整个数据集中的比例。假设数据集有 10 条记录，包含{'鸡蛋', '面包'}的有 5 条记录，那么{'鸡蛋', '面包'}的支持度就是 5/10 = 0.5。

（3）置信度 Confidence

置信度是针对某个关联规则定义的。有关联规则如{'鸡蛋', '面包'} -> {'牛奶'}，它的置信度计算公式为{'鸡蛋', '面包', '牛奶'}的支持度/{'鸡蛋', '面包'}的支持度。假设{'鸡蛋', '面包', '牛奶'}的支持度为 0.45，{'鸡蛋', '面包'}的支持度为 0.5，则{'鸡蛋', '面包'} -> {'牛奶'}的置信度为 0.45 / 0.5 = 0.9。

关联规则用于发现 if -> then 这样的规则，并可以给出这条规则的可信度（即置信度）。现实场景中可以用来发现很多规律，下面举个例子。在信息安全领域，需要根据已有流量数据制定规则，来判断是否触发安全报警。如规则{'数据包大'，'多个 ip 地址同时发送数据'} -> {'异常'}，该规则的置信度为 0.85。这条规则表示，当流量数据包大，并有多个 ip 地址同时向目标 ip 发送数据时，则有 85%的概率存在异常，需要触发报警。

（4）频繁项集挖掘原理

频繁项集挖掘分为构建 FP 树，和从 FP 树中挖掘频繁项集两步：

构建 FP 树

构建 FP 树时，首先统计数据集中各个元素出现的频数，将频数小于最小支持度的元素删除，然后将数据集中的各条记录按出现频数排序，剩下的这些元素称为频繁项；接着，用更新后的数据集中的每条记录

构建 FP 树，同时更新头指针表。头指针表包含所有频繁项及它们的频数，还有每个频繁项指向下一个相同元素的指针，该指针主要在挖掘 FP 树时使用。

挖掘频繁项集

得到 FP 树后，需要对每一个频繁项，逐个挖掘频繁项集。具体过程为：首先获得频繁项的前缀路径，然后将前缀路径作为新的数据集，以此构建前缀路径的条件 FP 树。然后对条件 FP 树中的每个频繁项，获得前缀路径并以此构建新的条件 FP 树。不断迭代，直到条件 FP 树中只包含一个频繁项为止。

（5）关联规则挖掘原理

关联规则挖掘首先需要对上文得到的频繁项集构建所有可能的规则，然后对每条规则逐个计算置信度，输出置信度大于最小置信度的所有规则。

FP-growth 算法优缺点:

*** 优点**：

1. 因为 FP-growth 算法只需要对数据集遍历两次，所以速度更快。
2. FP 树将集合按照支持度降序排序，不同路径如果有相同前缀路径共用存储空间，使得数据得到了压缩。
3. 不需要生成候选集。
4. 比 Apriori 更快。

*** 缺点**：

1. FP-Tree 第二次遍历会存储很多中间过程的值，会占用很多内存。
2. 构建 FP-Tree 是比较昂贵的。

*** 适用数据类型：标称型数据(离散型数据)**

参考

[1] FP-growth 模型

Peter Harrington 著《机器学习实战》

Zhang D, Zhang D, Zhang D, et al. Pfp: parallel fp-growth for query recommendation[C]// ACM Conference on Recommender Systems. ACM, 2008:107-114.

[2] An Implementation of the FP-growth Algorithm

Borgelt C. An Implementation of the FP-growth Algorithm[C]//Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations. ACM, 2005: 1-5.

[3] FP-Growth 算法

https://zhuanlan.zhihu.com/p/30194709

[4] FP-Growth 算法

https://github.com/apachecn/MachineLearning/blob/master/docs/12.使用 FP-growth 算法来高效发现频繁项集.md