频繁项集挖掘算法：Apriori 和 FP-growth 比较

一、 实验简介

频繁模式 (frequent pattern) 是频繁出现在数据集中的模式，包括项集、子序列或子结构。在本次实验中，我们需要在具有一定规模的数据集上进行**频繁项集**的挖掘，最终生成**关联规则**。如果使用暴力遍历进行频繁项集的查找，则会花费大量时间。**Apriori 算法**是一种发现频繁项集的基本方法，在暴力检索的基础上大幅减少了时间复杂度。另一种频繁项集挖掘算法是**频繁模式增长** (Frequent-Pattern Growth, **FP-growth**)，这种方法相较于 Apriori 算法理论上进一步降低了时间复杂度。

在该实验中，我们将会用代码实现 Apriori 和 FP-growth 算法，并在数据集上进行挖掘测试。另外，我们将会把暴力遍历作为基准方法 (Baseline)，从而更客观地对 Apriori 和 FP-growth 算法的实际运行表现进行比较与评估。

a) 数据集

实验用到的数据集一共有两个：GroceryStore 数据集和 UNIX_usage 数据集。

GroceryStore 数据集记录了一家杂货店一个月的交易记录。数据以 csv 的格式存储，每行是一次交易中顾客购买的商品的集合，每个商品可以看做一个项。我们的目标是找到数据集中所有的频繁项集，即经常出现的被同时购买的商品的集合，然后找到关联规则。

UNIX_usage 数据集记录了 8 个 UNIX 使用者 2 年内使用过的命令。一次命令记录以**SOF**开始，每一行是命令中的一个字符串，最后以**EOF**结束。

b) 实验方法

Apriori 算法 Apriori 算法是 Agrawal 和 R. Srikant 于 1994 年提出的，为布尔关联规则挖掘频繁项集的原创性算法。算法使用频繁项集性质的先验知识。首先，通过扫描数据库，累计每个项的计数，并收集满足最小支持度的项，找出频繁 1 项集集合。该集合记为 L_1。然后，使用 L_1 找出频繁 2 项集的集合 L_2，再使用 L_2 找出 L_3，如此下去，直到不能再找到频繁 k 项集。

FP-growth 算法 FP-growth 算法采用了和 Apriori 算法完全不同的分治策略：首先，将代表频繁项集的数据库压缩到一颗频繁模式树（**FP 树**），该树仍然保留项集的关联信息。然后把这种压缩后的数据库划分成一组条件数据库，每个数据库关联一个频繁项或“模式段”，并分别挖掘每个条件数据库。

二、 代码实现

由于 Python 在处理数据方面的强势，本次实验的三个算法均由 Python 实现，分别

保存在 Apriori.py，FP-growth.py 和 Baseline.py 文件中。每个文件均有独立的 main 函数，在 main 函数中均包含读入数据、生成频繁项集和产生关联规则三个部分。其中只有生成频繁项集部分三个文件的实现各不相同的，读入数据和产生关联规则部分都是同样的实现。以下将会从这三个部分具体展开。

a) 数据读取

本次实验中，共使用了 2 个数据集。由于这 2 个数据集具有不同的数据储存格式，所以实现了 load_data_set1() 和 load_data_set2() 两个函数，分别用于读取 GroceryStore 数据集和 UNIX_usage 数据集。这两个函数读取数据后都会返回一个 2 维**列表（List）**data_set，存放所有的项。使用 Python 进行文件的读写是一件非常令人愉悦的事情，过程非常简单轻松，具体内容详见源代码。需要注意的是 UNIX_usage 数据集中每个**事务（transaction）**中可能会出现多个相同的项，所以在读入数据时需要先使用 set 类型，使得一个名字的项只出现一次，然后再转换成列表存放在 data_set 中。

b) 频繁项集挖掘

Baseline 算法

Baseline 算法及暴力检索的算法。其思想是每轮遍历一遍原数据集 data_set，找出 k 频繁项集，直到无法找到新的频繁项集为止。代码整体结构是每一轮调用 exhau_gen(data_set, k, min_sup) 函数，其中 k 为轮次，返回一个 k 频繁项集 L_k。最后将每轮生成的 L_k 合并成 L，即得到了全部的频繁项集集合 L。

在 exhau_gen 函数中，首先会对 data_set 中的每条记录 itemset，调用 FindkSubset(itemset, k) 函数找到其所有的长度为 k 的子集，并累计其出现次数。遍历完 data_set 后，对所有找到的 k 长子集，删去其中计数值小于最小支持度 min_sup 的，就得到 k 频繁项集 L_k。

Apriori 算法

Apriori 算法思想是每轮基于 k-1 频繁项集生成 k 频繁项集。在 main 函数中，首先会调用 find_frequent_1_items(data_set, min_sup) 函数生成频繁 1 项集集合。然后从 k=2 开始，每轮先调用 aproiri_gen(L_km1, k) 函数产生候选 k 项集集合 C_k，其中 L_km1 表示频繁 k-1 项集集合 L_{k-1}；这个过程又被称为**连接步**。再调用 pruning(C_k, data_set, min_sup)去除 C_k 中不满足最小支持度的候选，得到 L_k；这个过程又被称为**剪枝步。**一直重复直到新产生的 Lk 为空集为止。

find_frequent_1_items 函数的任务是生成频繁 1 项集字典 L1。字典 L1 的 key 属性为由单个项组成的元组，value 属性记录该项在 data_set 中出现的次数。由于 Python 中的**字典（Dictionary）**是用哈希散列的方式实现的，插入和查询速度较快，所以很适合用在这种频繁插入和查询元素的情况中。为了生成 L1，首先会对 data_set 进行一次遍历，将所有的项都转换成单个元素的**元组（Tuple）**存放到 L1 中，并进行计数（元组可以散列，而列表不行）；然后去掉 L1 中计数小于最小支持度 min_sup 的成员，L1 就成为了频繁 1 项集字典。

aproiri_gen 的任务是根据频繁 k-1 项集字典 L_km1 构造候选 k 项集列表 C_k。

设 l1 和 l2 是 L_km1 中的项集，Aporiori 算法会保证每一轮生成的项集中的项按字典序排序。如果满足 l1 和 l2 前 k-1 项相同，且最后一项 l1<l2，则连接 l1 和 l2 产生结果项集 [l1[1], l1[2], …, l1[k-1], l2[k-1]]。另外，为了检查新生成的项集是否包含不频繁的 k-1 长子集，还会调用 *has_ infrequent_subset* 函数，函数返回 False 则将项集加入到 C_k 中。

在 *pruning* 函数中，我们会再扫描一遍数据库 data_set，确定 C_k 中每个候选的计数。然后将计数不小于 min_sup 的项转换为元组加入 L_k 中，并由 L_k 中的 value 属性记录其计数值。

FP-growth 算法

FP-growth 算法核心是 FP 树的生成，以及在 FP 树上递归地进行挖掘。这两步分别在 *createFPtree* 和 *FP_growth* 函数中实现。

为了实现树结构，我们如下定义了一个树节点类 treeNode：

```
42  class treeNode:
43      def __init__(self, name, count, parent):
44          self.name = name          #item
45          self.count = count        #计数
46          self.nodeLink = None      #节点链的下一位置
47          self.parent = parent
48          self.children = {}
49
50      def add_count(self, count):
51          self.count += count
```

createFPtree 函数负责构造 FP 树，返回 FP 树的根节点指针 FPtree 和对应的项头表字典 headerTable。首先，找出所有频繁项及其计数加入 headerTable 中，然后采用如下操作扩展 headerTable 的 value 属性，使其可以存放节点链指针：

```
90      for i in headerTable:
91          headerTable[i] = [headerTable[i], None] #项ID:[支持度计数, 节点链]
```

接着创建 FP 树根节点 FPtree。对于 data_set 中的每个事务，选择其中频繁项按 headerTable 中计数降序排序。然后调用 *insert_tree* 函数将排好序的项集插入到 FP 树中。*insert_tree* 的实现在此不进行展开，具体详见源代码。

FP_growth 函数负责对 FP 树进行递归的挖掘，其具体步骤如下：

def FP_growth (FPtree，α, headerTable, L, min_sup):

(1) **for** pat **in** headerTable:

(2) 产生新模式 β=[pat]+α

(3) β加入字典 L 中，其计数等于 pat 在 headerTable 中的计数

(4) 调用 *findCondPatternBase* 函数找到 pat 在 FPtree 中的前缀路径集字典 condPats

(5) 调用 *createCondFPtree* 函数基于 cindPats 生成条件 FP 树 newFP 和新的项头表 newHT。*createCondFPtree* 函数和 *createFPtree* 函数实现方法类似，只不过前者的输入数据集类型是字典，而后者是二维列表

(6) **if** newFP≠**None**:

(7) **FP_growth** (newFP，β, newHT, L, min_sup)

在主函数中调用 FP_growth(FPtree, [], headerTable, L, min_sup)，即可得到频繁

项集字典 L。

c) 关联规则生成

经过煞费周折的频繁项集挖掘，我们已经得到了频繁项集字典集合 L，字典的 value 属性记录了 L 中每个项集的出现次数。基于频繁项集，就可以直接由他们产生满足最小支持度和最小置信度的强关联规则。

首先，对 L 中的每一个项集 freqitem，调用 FindAllSubset(freqitem)找出其所有的非空真子集(该函数同时还返回子集的补集)，对每个这样的子集 s 和补集 anti_s，我们要推出规则“s→anti_s”，则必须使其满足最小置信度。由置信度计算公式知：

$$\text{Confidence}(s \rightarrow \text{anti\_}s) = P(s|\text{anti\_}s) = \frac{\text{support\_count}(\text{freqitem})}{\text{support\_count}(s)}$$

所以只要 L[freqitem]/L[s]值不小于最小支持度 min_sup，就输出规则“s→anti_s”。

三、 方法比较

三个算法的实现方法不同，挖掘频繁项集的效率也差别较大。为了更客观地比较三种算法所花费的时间，我们引入 Python 中的 time 包，记录每种算法生成频繁项集过程中所花费的时间。首先将 GroceryStore 中的 Groceries.csv 文件作为数据集，算法所花费的时间（单位：秒）随最小支持度的变化如下图所示：

最小支持度	Baseline	Apriori	FP-growth
50	-	16.31	0.66
100	-	6.57	0.50
150	-	4.01	0.45
200	-	2.66	0.42
250	-	2.09	0.40

Groceries.csv 文件中包含了 9835 条事务，是具有一定规模的数据集。可以发现，随着最小支持度的增大，算法所花费的时间都在减小。Baseline 算法由于时间复杂度过高，导致在 5 分钟内都没有得到运行结果。Apriori 则利用先验知识大幅降低了时间复杂度，在 20 秒内就给出了频繁项集。FP-growth 则利用 FP 树，全程只需要对数据库扫描 2 次，所以仅在 1 秒内就给出了结果。

综合来看，Apriori 和 FP-growth 的效率远远优于 Baseline 方法，FP-growth 算法大约比 Apriori 算法快一个数量级。这个结论与对 FP-growth 性能相关研究得出的答案一致。

为了避免数据集选择导致的实验结果偏差，我们再来看看三个算法在 UNIX_usage 数据集上的表现。我们将 UNIX0 中的 sanitized_all.981115184025 文件作为数据集，算法所花费的时间（单位：秒）随最小支持度的变化如下图所示：

最小支持度	Baseline	Apriori	FP-growth
20	-	0.219	0.031
40	-	0.047	0.031
60	-	0.016	0.000
80	-	0.016	0.000

该数据集中只有 562 条事务，相对于 GroceryStore 比较少，算法花费的时间更少，然而 Baseline 方法还是没能在有限时间里给出结果。总体趋势与在 GroceryStore 数据集上测试的相同。所以我们得到的结论依然成立。

四、 实验结果

a) 对于三种算法

——在测试数据集上三种算法给出的频繁项集集合是一致的，所以我们可以在一定程度上保证三种算法实现的正确性。

——随着最小支持度和最小置信度的提高，算法给出的频繁项集和关联规则数量不断减少，算法花费的时间也不断减少。

——Apriori 和 FP-growth 的效率远远优于 Baseline 方法，FP-growth 算法大约比 Apriori 算法快一个数量级

b) 挖掘频繁项集方法应用

——尽管 Apriori 已经能够在 20 秒内快速给出频繁项集，但如果要用到实时搜索等对时间复杂度有严格要求领域，Apriori 的效率并不能让人满意。相较而言，能在 1 秒内给出结果的 FP-growth 算法显然更胜一筹。

——虽然 FP-growth 算法相较于 Apriori 算法快一个数量级，但是由于 Apriori 算法基于 k-1 频繁项集构造 k 频繁项集的代码架构，其算法扩展性较好，可以用于并行计算等领域，并进一步提升其性能。

五、 结果复现

本次实验的三个算法均由 Python 实现，分别保存在 Apriori.py，FP-growth.py 和 Baseline.py 文件中。三个文件均有独立的 main 函数。

最小支持度和最小置信度初始化在 main 函数开头，读入数据时可以选择调用 *load_data_set1* 函数进行 GroceryStore 数据集的读入，也可以选择调用 *load_data_set2* 函数进行 UNIX_usage 数据集的读入。具体的文件读入路径硬编码在 *load_data_set1* 和 *load_data_set2* 函数体中。

每个文件的 main 函数执行后，会在控制台依次输出以下内容：

(a) 频繁项集字典 L

(b) 生成频繁项集阶段花费的时间 Total cost

(c) 所有的强关联规则

实验用到的数据集和源代码都可以在 https://github.com/NJUaaron/FreqItemExcav 网站上找到并使用。