# bi分析报告

1. 情景分析

本题要求在前两题的基础上，利用sldat这一时间字段，设计Sequential FP算法，进行频繁集挖掘。

1. Sequential FP说明

我使用的是PrefixSpan算法，它是对FP-Growth算法提出的改进。PrefixSpan算法的全称是Prefix-Projected Pattern Growth，即前缀投影的模式挖掘。

PrefixSpan算法的目标是挖掘出满足最小支持度的频繁序列。那么怎么去挖掘出所有满足要求的频繁序列呢，回忆Aprior算法，它是从频繁1项集出发，一步一步的挖掘2项集，直到最大的K项集。PrefixSpan算法也类似，它从长度为1的前缀开始挖掘序列模式，搜索对应的投影数据库得到长度为1的前缀对应的频繁序列，然后递归的挖掘长度为2的前缀所对应的频繁序列，。。。以此类推，一直递归到不能挖掘到更长的前缀挖掘为止。

PrefixSpan算法的流程：

输入：序列数据集S和支持度阈值α

　　输出：所有满足支持度要求的频繁序列集

1. 找出所有长度为1的前缀和对应的投影数据库
2. 对长度为1的前缀进行计数，将支持度低于阈值α的前缀对应的项从数据集S删除，同时得到所有的频繁1项序列，i=1
3. 对于每个长度为i满足支持度要求的前缀进行递归挖掘：
4. 找出前缀所对应的投影数据库。如果投影数据库为空，则递归返回。
5. 统计对应投影数据库中各项的支持度计数。如果所有项的支持度计数都低于阈值α，则递归返回。
6. 将满足支持度计数的各个单项和当前的前缀进行合并，得到若干新的前缀。
7. 令i=i+1，前缀为合并单项后的各个前缀，分别递归执行第3步。

PrefixSpan算法由于不用产生候选序列，且投影数据库缩小的很快，内存消耗比较稳定，作频繁序列模式挖掘的时候效果很高。PrefixSpan运行时最大的消耗在递归的构造投影数据库。如果序列数据集较大，项数种类较多时，算法运行速度会有明显下降。

1. 代码说明

sequentialFP.py文件是将序列FP算法集成的一个模块文件，将所有的过程都封装到了函数fpGrouth()中，只需传入数据字典、最小阀值两个参数即可，返回是当前阀值下的频繁集。

bi\_model.py是将数据处理、算法调用等函数集成的一个模块，只需要调用fp\_pluno()、fp\_dptno()、fp\_bndno()三个函数即可。

bi\_pluno.py、bi\_dptno.py、bi\_bndno.py分别是对三种item进行频繁集挖掘的代码，主要完成数据的整理和调用FP算法。

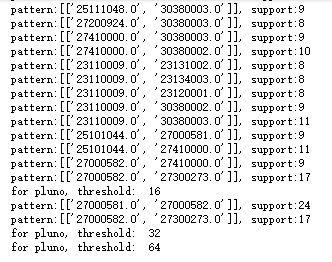
考虑到这是对购物记录做频繁项集的挖掘，我只将含有两个以上item的项集作为输出，只含有一个item的项集并没有什么意义。

1. 运行结果

由于作业要求分别取阀值为2，4，8，16，32，64，所以程序的输出比较长，无法一次截取，所以分别取得到的结果的一部分进行分析：

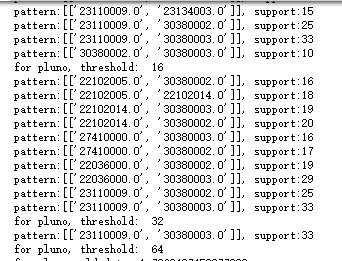
* 1. Pluno商品

对于旧数据trade，有：



得到的结果和使用一般FP-Growth是算法是效果一样不佳。

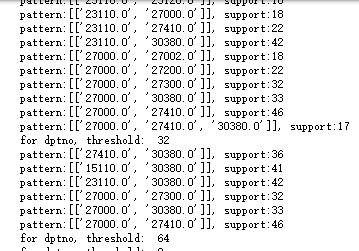
对于新数据trade\_new，有：



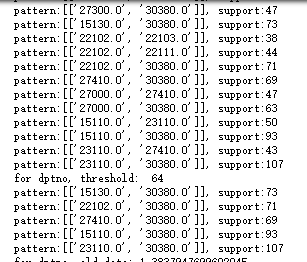
得到的结果和ai的结果差不多，可能的原因是在ai题做数据预处理的时候，以及按照时间将数据进行过一次排序了。

* 1. Dptno商品类型

旧数据：



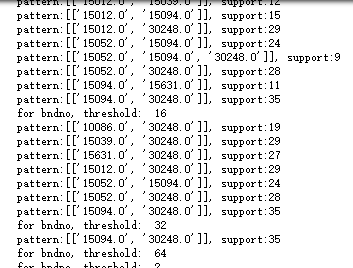
对于新数据：



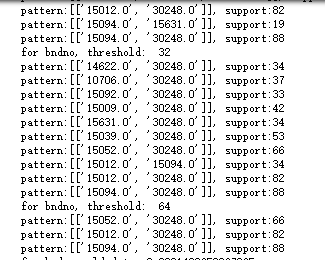
对于dptno得到的结果也和ai一样。

* 1. Bndno品牌：

旧数据：

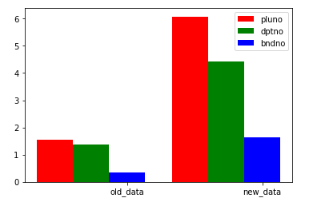
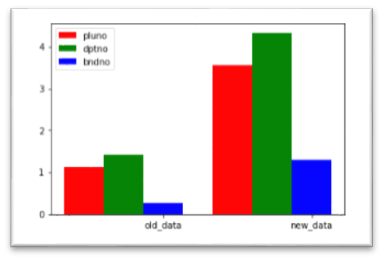


新数据：



和pluno,dptno的结果不同，bndno新旧数据的结果都和ai不同，相对还要比ai题的结果要好。

* 1. 性能对比



左图是ai的结果，右图是本题的结果。对于pluno 和dptno，FP-growth和Sequential FP的效果一样，而且两者花费的时间基本上不差太多。对于bndno，两个算法的结果都比较好，Sequential FP的结果稍微好一点。