Shiyao Jiang (ID: 112382630)
Prof. Alex Doboli
ESE 589 Learning System
11/11/2019

Project 2: FP Growth

Abstract

With the development of big data technology, the task of frequent pattern mining has become increasingly important. One of the pioneer thoughts were Apriori algorithm. However, it is not efficient enough to handle big data due to its nature of candidate elimination algorithm, which traverses the entire data set each time when executing elimination.
FP growth was born for improving frequent pattern mining. It constructs a tree to compress the data, so that frequent pattern mining only requires two times traversing in advance. In addition, this algorithm doesn't generate candidates sets, leading to a much higher efficiency.

Introduction

The key thought of FP growth is to let frequent patterns (FP) grow during the recursion along the FP tree.
Intuitively, we first start from a pattern of "null". Then from the FP tree, which contains only the items with support counts greater than minimum support, we find an item that has the minimum frequency, plug it into the tail of the pattern and check if the conditional FP tree of this new pattern has only one route. If yes, the outputs are just combinations of all the items in the tree; if no, we start from the conditional FP tree with the new pattern. The whole process could be concentrated into the image below:

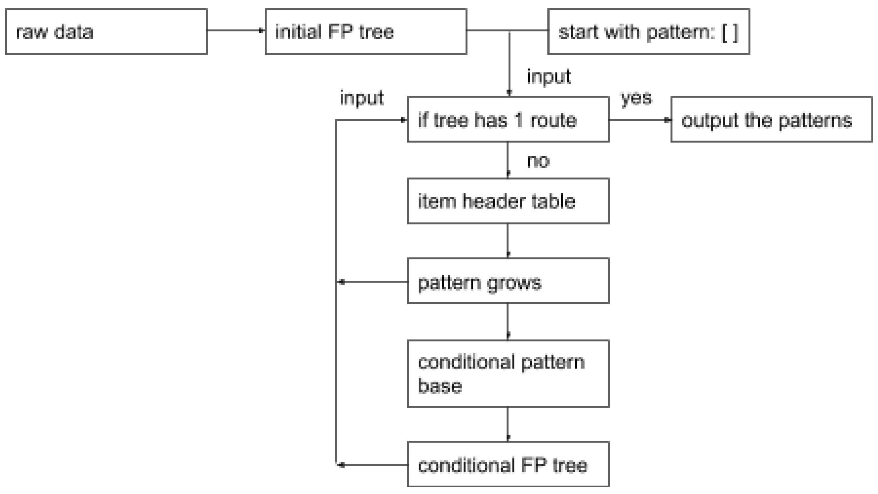


Fig. 1: FP growth flow diagram

Of course, this figure ignores a lot of details and looks rough, but it still describes the outline of FP growth. Apparently, there is no candidate in the description or the graph of FP growth, so it saves a lot of time on searching the potential patterns and pruning them.

Python Implementation

The first step was to decide the formation of every member in that flow chart.

Database/Transaction

A database used for frequent pattern mining is a set of transactions, and a transaction is a set of items. In Python language, the most easy way to store data and keep their order is storing as lists. Items, apparently, are string variables, so a transaction could be a list of strings, while a database could be a list of lists.

FP tree nodes

It must be the most important and most complicated, so it should be an individual class. As a basic design principle, for simplicity, I hope most operations to the tree should be finished by nodes, so the functions outside the tree node class would form a "shell" of the trees. The tree node class contains four variables: name as the item, value as the support count, parent and children for the connection of the tree nodes. And the nodes should be able to do four things:

- connect to each other
- search and return the specific nodes to form the node links
- check if the tree has only ONE path
- return conditional pattern base

Conditional Pattern Base

According to the textbook, conditional pattern base could be regarded as a database, so it has the same construction as the database: list of lists.

Item Header Table

This is 3*n table. Each row contains the item name, support count and the node links. The node link suppose to be a pointer navigates to one of the nodes on a tree, and the node should contain another pointer navigates to another node. But I don't think this is explicit and simple, so I just store all the corresponding nodes in a list as the third element of each row of item header table. The item header table looks like:

```
[[item_name, support_count, [*node1, *node2, *node3...]],
[item_name, support_count, [*node1, *node2, *node3...]], …]
```

Pattern
This is just a list of item names.

Modules
There are 4 files (datasets not included) in this project.

Table 1: program file description

main.py	Integration of other scripts
FPtree.py	FP tree node class
FPtreeMining.py	The FP growth algorithm itself and some additional functions
printTree.py	Customized function for visualization of the FP tree

Theoretically, only two parameters are needed in the main file in order to run the program, file name and minimum support. Datasets are in the data folder, *small.csv* contains the one used in the example in the textbook, the other two are from UCI datasets.

FP_growth()
Comparing to the pseudo code in the textbook, the *FP_growth()* function in my program has two more arguments (*L1Tuple, min_sup*) since the function is not in the main file while these arguments are necessary for sorting the item by frequency or pruning the single-branch tree.

Performance Measurement
Of course, there are some methods to test the program's performance. Here I measured the time and memory usage to evaluate its efficiency.

Results and Conclusions

As benchmark testing datasets, I chose *solar flare* and *mushroom*. The dataset used by example 6.5 in the textbook is also included, which is named as *small.csv*. Among these datasets, *flare.data1.csv* has been modified in order to increase its complexity, while others remain original.
Firstly let's test the example in the textbook. Here are just some parts of the outputs.
For the first scan, we obtain the sorted items with their support counts with *min_sup=2*:

```
L1: [('I2', 7), ('I1', 6), ('I3', 6), ('I5', 2), ('I4', 2)]
```

This is a little different from the one in the textbook, the order of the last two items is reversed. But they have the same support counts, hopefully this may not cause any negative effect. Then there comes the initial FP tree:

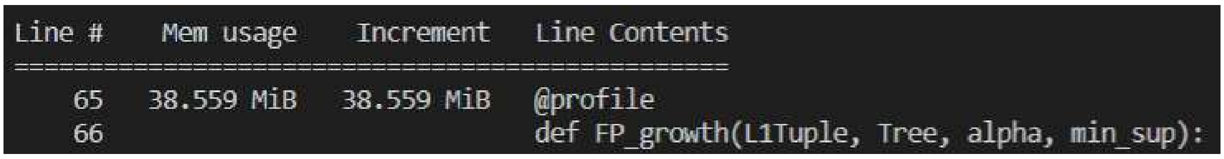
```
Initial FP tree:
                   ┌I5(1)
             ┌I1(4)┼I4(1)
             │     └I3(2)─I5(1)
       ┌I2(7)┼I4(1)
       │     └I3(2)
null(0)┤
       └I1(2)─I3(2)
```

Every node in the tree consists of two parts, the name and the support count (the number in the brackets) of the item. The root of the FP tree is *null(0)*. This tree was just the same as the one in textbook. The next step was doing FP growth by calling *FP_growth()*. Some results would be printed onto the console, such as the item header table, the conditional FP trees and the output patterns:

```
item header table:
['I2', 7, [<FPtree.FPtree object at 0x0000026218AD96D8>]]
['I1', 6, [<FPtree.FPtree object at 0x0000026218AD9710>, <FPtree.FPtree object at
0x0000026218AD9828>]]
['I3', 6, [<FPtree.FPtree object at 0x0000026218AD9898>, <FPtree.FPtree object at
0x0000026218AD97B8>, <FPtree.FPtree object at 0x0000026218AD9860>]]
['I5', 2, [<FPtree.FPtree object at 0x0000026218AD9748>, <FPtree.FPtree object at
0x0000026218AD98D0>]]
['I4', 2, [<FPtree.FPtree object at 0x0000026218AD97F0>, <FPtree.FPtree object at
0x0000026218AD9780>]]

patterns:
('I4',):2

null(0)─I2(2)─I1(1)

patterns:
('I2', 'I4'):2

patterns:
('I5',):2

null(0)─I2(2)─I1(2)─I3(1)

patterns:
('I2', 'I5'):2
```

```
('I1', 'I5'):2
('I2', 'I1', 'I5'):2

patterns:
('I3',):6

       ┌I2(4)─I1(2)
null(0)┤
       └I1(2)

item header table:
['I2', 4, [<FPtree.FPtree object at 0x0000026218AD9B70>]]
['I1', 4, [<FPtree.FPtree object at 0x0000026218AD9AC8>, <FPtree.FPtree object at
0x0000026218AD9B38>]]

patterns:
('I1', 'I3'):4

null(0)─I2(2)

patterns:
('I2', 'I1', 'I3'):2

patterns:
('I2', 'I3'):4

null(0)

patterns:
('I1',):6

null(0)─I2(4)

patterns:
('I2', 'I1'):4

patterns:
('I2',):7

null(0)
```

It’s obvious that the results showed all the patterns as in the textbook, and the conditional FP trees shown during the recursions were just the same (including those single-item pattern). Since it’s a small dataset, the time used by the whole program was very close to 0, and the memory used was nearly 40MB:



```
Line #    Mem usage    Increment   Line Contents
================================================
    65   38.559 MiB   38.559 MiB   @profile
    66                             def FP_growth(L1Tuple, Tree, alpha, min_sup):
```

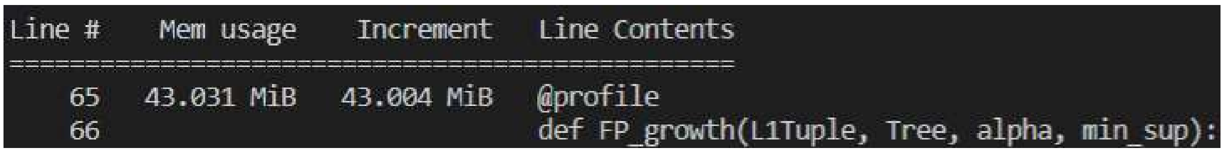
Here is another example, the *mushroom* dataset. Set *min_sup=7500* (this is a relatively large dataset), and hide the details of item header tables and conditional FP tree:

```
L1:
[('w', 22906), ('p', 22788), ('f', 21494), ('s', 14524), ('e', 13408), ('n', 12868),
('b', 12648), ('t', 8584), ('o', 8080), ('c', 8020), ('k', 7784)]

FP tree:
                                                       ┌t(256)─o(256)─c(256)─k(160)
                                                       │                       ┌c(3896)─k(2450)
                                                       │               ┌o(4712)┤
                                                       │               │       └k(450)
                                                       │       ┌t(5286)┼c(304)─k(66)
                                                       │       │       └k(222)
                                               ┌n(5914)┼b(5514)┤
                                               │       │       │      ┌c(210)─k(102)
                                               │       │       └o(228)┤
                                               │       │              └k(6)
                                               │       └o(144)─c(144)─k(60)
                                       ┌e(6042)┤
                                       │       └b(128)─t(128)─o(128)─c(128)─k(128)
                               ┌s(6330)┤
                               │       └b(288)─t(288)─o(288)─c(288)
       ┌w(7932)─p(7932)─f(7932)┤
       │                       │                                    ┌k(30)
       │                       │                             ┌o(192)┤
       │                       │                             │      └c(144)─k(144)
       │                       │                      ┌t(218)┼k(18)
       │                       │                      │      └c(8)─k(2)
       │                       │                      │      ┌c(736)─k(730)
       │                       │               ┌b(996)┼o(742)┤
       │                       │               │      │      └k(2)
       │                       │               │      └c(36)─k(36)
       │                       │       ┌n(1026)┤
       │                       │       │       └o(30)─c(30)─k(26)
       │                       └e(1602)┤
       │                               └b(576)─o(576)─c(576)─k(576)
null(0)┤
       └p(193)─f(193)─s(193)─e(193)─n(193)─b(193)─o(193)─c(193)─k(48)

patterns (shortened, 27 in total):
('b',):7695
('f', 'b'):7695
('p', 'f', 'b'):7695
('w', 'p', 'f', 'b'):7502
('w', 'f', 'b'):7502
```

```
...
('w',):7932
('p', 'f'):8125
('w', 'f'):7932
('p',):8125
('w',):7932
```

Of course, there were measurements of time and memory usage:

```
Used time (sec):
first scan: 0.039893150329589844
second scan: 0.045877456665039062
build the initial FP tree: 0.030917406082153320
FP mining: 0.002992391586303711

Used memory for FP_growth():
```



```
Line #    Mem usage    Increment   Line Contents
================================================
    65   43.031 MiB   43.004 MiB   @profile
    66                             def FP_growth(L1Tuple, Tree, alpha, min_sup):
```

It’s surprising that the memory requirement didn’t increase too much (from 38.56MB to 43.03MB), given that the *mushroom* dataset is nearly 5000 times larger than the one used in the textbook. So I can assert FP growth would not consume too much computation power even when dealing with big data.

Reference

[1] http://archive.ics.uci.edu/ml/datasets/Mushroom
[2] http://archive.ics.uci.edu/ml/datasets/solar+flare