Xiaoning Guo

Frequent Itemsets with Adult Census Data

Using Apriori and FP-Growth

Contents

HOW TO RUN

Requirements: Python 3.6 +, Anaconda installed and Path Environment Set

Paste all the files into the current working directory. Run the following command:

```
C:\Users\Xiaoning Guo\Desktop>python Miniproject.py adult.data 20000
```

DATA CLEANING CHOICES

The provided adult census dataset is not a list of itemsets, thus it is necessary to convert it into something of such nature. Continuous data needs to converted to categorical data

Since *age* is continuous, I binned it according to the criteria:

‘Young’(less than 30), ‘Middle-aged’ (between 30 and 55), ‘Old’ (greater than 55).

I treated *fnlwgt* as a feature and binned it according to the criteria:

‘fnlwgt_low’ (less than 50000), ‘fnlwgt_low’ (between 50000 and 100000), ‘fnlwgt_high’ (greater than 100000).

I binned *capital gains* into two categories:

'gain_zero' (no capital gains) and 'gain_positive' (greater than 0). I realized most these attributes were either 0 or positive.

I binned *capital losses* into two categories:

'loss_zero' (no capital gains) and 'loss_positive' (greater than 0). I realized most these attributes were either 0 or positive.

This also distinguishes one 0 from the other (capital loss zero or capital gain zero).

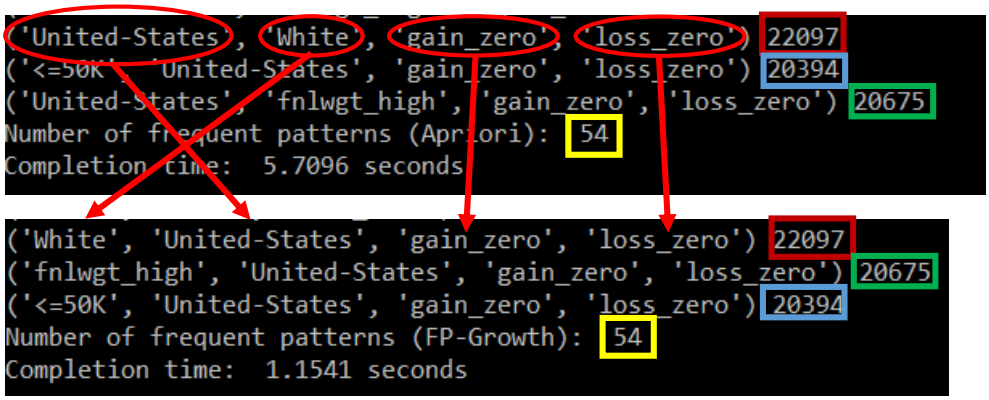
I binned *hours per week* into two categories:

'full-time" (>= 40 hours per week) and 'part-time' (< 40 hours per week). This is the most logical way to do it as it is represented in real life.

PROOF THAT IT WORKS

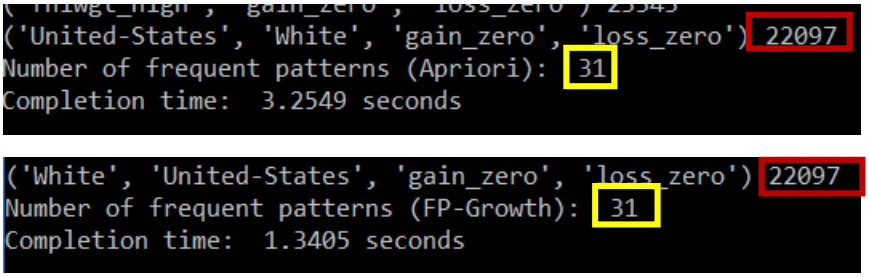
Using snippets from the output:

```
C:\Users\Xiaoning Guo\Desktop>python Miniproject.py adult.data 20000
```



Both algorithms yielded the same number of frequent patterns, all frequencies are >= 20000, all frequencies for the same itemset are the same. All frequent patterns are the same (the order might be rearranged, but they're still the same itemsets).

```
C:\Users\Xiaoning Guo\Desktop>python Miniproject.py adult.data 22000
```



Again, same story with a different support threshold. Therefore, it isn't just getting lucky.

```
C:\Users\Xiaoning Guo\Desktop>python Miniproject.py test.data 3 showtree
```

```
('M',) 3
('O',) 3
('K',) 5
('E',) 4
('Y',) 3
('K', 'M') 3
('K', 'O') 3
('E', 'O') 3
('E', 'K') 4
('K', 'Y') 3
('E', 'K', 'O') 3
Number of frequent patterns (Apriori): 11
Completion time:  0.02 seconds
```

```
('K',) 5
('E',) 4
('M',) 3
('O',) 3
('Y',) 3
('E', 'K') 4
('M', 'K') 3
('O', 'K') 3
('O', 'E') 3
('Y', 'K') 3
('O', 'E', 'K') 3
Number of frequent patterns (FP-Growth): 11
Completion time:  0.0055 seconds
```

```
('K',) 5
       ('E',) 4
              ('M',) 2
                     ('O',) 1
                            ('Y',) 1
              ('O',) 2
                     ('Y',) 1
       ('M',) 1
              ('Y',) 1
```

```
M, O, N, K, E, Y
D, O, N, K, E, Y
M, A, K, E, ?, ?
M, U, C, K, Y, ?
C, O, 0, K, I, E
```

On the test.data, we see that it matches the answers from question 6.6. The printed FP-tree also matches the one we constructed for that question.

EXPECTED OUTPUT

C:\Users\Xiaoning Guo\Desktop>python Miniproject.py adult.data 28000

```
C:\Users\Xiaoning Guo\Desktop>python Miniproject.py adult.data 28000
('loss_zero',) 31042
('United-States',) 29170
('gain_zero',) 29849
('gain_zero', 'loss_zero') 28330
Number of frequent patterns (Apriori):  4
Completion time:  0.7399 seconds

('loss_zero',) 31042
('gain_zero',) 29849
('United-States',) 29170
('gain_zero', 'loss_zero') 28330
Number of frequent patterns (FP-Growth):  4
Completion time:  0.8023 seconds
```

Standard usage should run the apriori algorithm on the selected dataset and then run the fp-growth algorithm right after. It should then print out all the frequent itemsets with their respective frequencies (having support greater than or equal to the specified support count).

C:\Users\Xiaoning Guo\Desktop>python Miniproject.py test.data 3 showtree

```
('K',) 5
       ('E',) 4
              ('M',) 2
                     ('O',) 1
                            ('Y',) 1
              ('O',) 2
                     ('Y',) 1
       ('M',) 1
              ('Y',) 1
```

The ‘showtree’ parameter prints out the fp-tree structure with their counts and node values. This comes after the standard output.

C:\Users\Xiaoning Guo\Desktop>python Miniproject.py test.data 3 fp-only

```
C:\Users\Xiaoning Guo\Desktop>python Miniproject.py adult.data 28000 fp-only
('loss_zero',) 31042
('gain_zero',) 29849
('United-States',) 29170
('gain_zero', 'loss_zero') 28330
Number of frequent patterns (FP-Growth):  4
Completion time:  0.7678 seconds
```

The ‘fp-only’ parameter means to use only the fp-growth algorithm. This comes in handy when the apriori algorithm runs very slowly on low minimum support counts.

PROGRAM DESIGN

Everything is written in Miniproject.py and works perfectly as intended. The data is loaded in via Pandas DataFrame and binned using column operations. The Apriori Algorithm is written exactly as it is from the textbook. However, the FP-Growth one was very tricky. I created my own tree data structure which was similar to a linked list except there is a list of children. The tree nodes also stored a bunch of information like the count, the item itself, its parent and its children. As for the header table, I used an ordered dictionary by descending counts where the key is the given item and the value is the node linked list. Since Python did not have its own linked list package, I wrote one myself. I used these for the header table which links up to each tree node in the FP structure. The header table makes it really convenient to generate new trees as it can find exactly all occurrences of the given a_i as opposed to searching through the entire dataset every time.

COMPARISON OF ALGORITHMS

A **brute force approach** is very impractical as it requires the generation and checking of 2^n number of items. An exponential runtime is VERY expensive.

The **Apriori Algorithm** works similar to brute forcing but with pruning by a special property. It takes advantage of the fact that if a subset of an itemset is not frequent then it too is infrequent. This prunes down the number of possible combinations that can come after each itemset building step. Starting with all single items, it prunes it by removing all items that do not meet the minimum support count and then builds a new candidate set with the surviving items. The following prune step check to see if any k-1 subset is infrequent. This process repeats until no new itemsets can be formed. Note that this process is very computationally expensive as it loops through the candidate lists multiple times, forming new combinations and checking all subsets. The looping of combinations over and over again is very expensive.

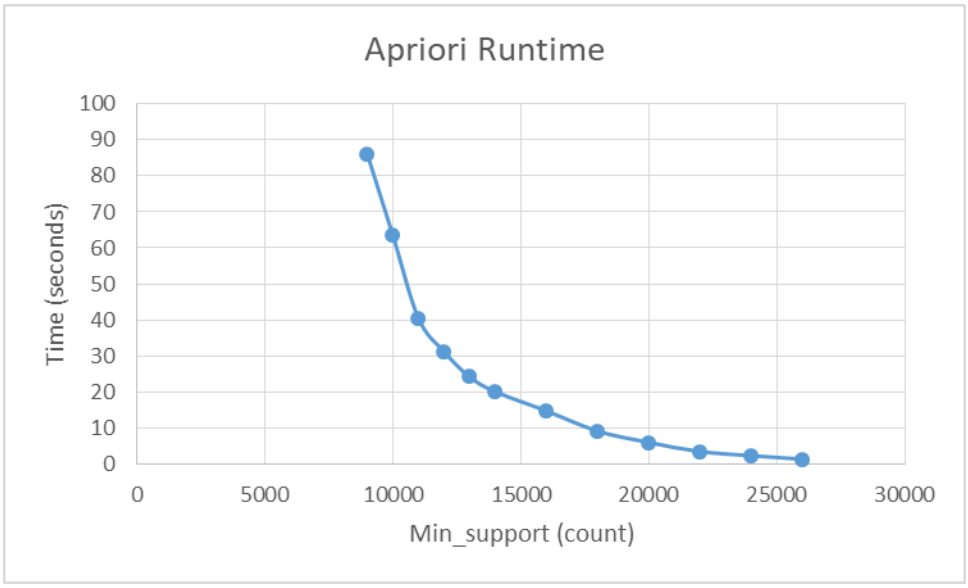
The **FP-Growth Algorithm** is a two-part algorithm that is preferred over the Apriori Algorithm as it uses a divide-and-conquer approach. First, it begins the same way as the Apriori Algorithm and finds all frequent size 1 itemsets. Then it constructs a tree from the entire dataset which is essentially a condensed version with no information lost. Instead of mining the dataset directly, we can then mine the tree efficiently. By using a combination of data structures like trees, linked lists and hashmaps, the runtimes for retrieval and insertion are O(1). Thus, the construction of the tree itself is not very expensive. The actual mining algorithm builds trees

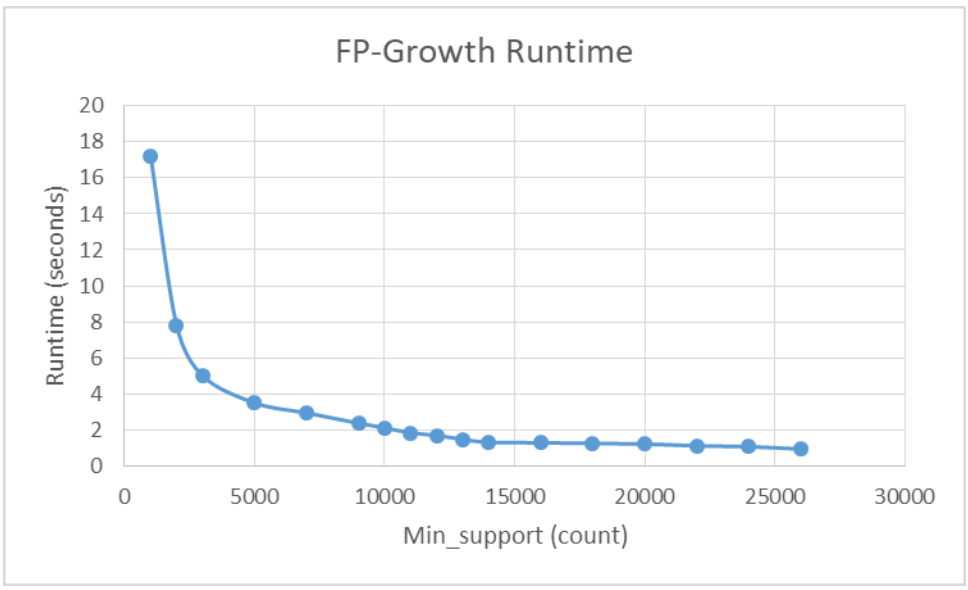
recursively until single paths are found. Then all combinations that form from the single paths are the frequent patterns with the count of its last node. The tree structures and linked lists from the header table allow for very fast traversal, thus it can find frequent patterns much faster.

As shown below, we see that FP-Growth is magnitudes faster as we lower the minimum support count. We see that the runtimes for FP-growth grow a lot slower than Apriori:

Runtime Table

Support/Algorithm	Apriori	FP-Growth
1000	I will be dead by then	17.237s
2000	Longer than I can wait	7.806s
3000	Longer than I can wait	5.001s
5000	Longer than I can wait	3.491s
7000	Longer than I can wait	2.934s
9000	86.197s	2.364s
10000	63.583s	2.108s
11000	40.484s	1.827s
12000	31.178s	1.679s
13000	24.164s	1.466s
14000	20.034s	1.303s
16000	14.679s	1.273s
18000	9.0727s	1.237s
20000	5.917s	1.206s
22000	3.383s	1.104s
24000	2.242s	1.054s
26000	1.255s	0.925s


Apriori Runtime
Time (seconds)
100
90
80
70
60
50
40
30
20
10
0
0
5000
10000
15000
20000
25000
30000
Min_support (count)


FP-Growth Runtime
Runtime (seconds)
20
18
16
14
12
10
8
6
4
2
0
0
5000
10000
15000
20000
25000
30000
Min_support (count)