Association Rule Mining Project

Apriori Algorithm

Association rule mining finds interesting associations and relationships among large sets of data items. This rule shows how frequently a itemset occurs in a transaction. Given a set of transactions, the goal is to find rules that will predict the occurrence of an item based on the occurrences of other items in the transaction.

Transaction Reduction

A transaction that does not contain any frequent k-itemsets cannot contain any frequent (k+1)-itemsets. Therefore, such a transaction can be marked or removed from further consideration. Implemented the below algorithm from scratch per the paper. Refer transaction_reduction.py for implementation.

```
Algorithm 1 - TR-RC for FIM
Min_sup. : Minimum support count
Step 1: Begin
Step 2: Read BAM
Step 3:
        Generate the set of frequent 1  itemset
        Add RC column //
k:=2;
while (L_{k-1} ≠ ø) do
begin
        for each k itemset
                compute sup_count
                if  sup_count >= min_sup then
                 L_k := All candidates in C_k with   minimum support ;
                end if
        end for
k := k + 1;
end  Answer := U_k L_k
Step 4:   End.
```

Rules are generated for toy data as below with minimum support of 2 and confidence of 30%.

```
printRules(valid_rules)
```

```
i2 66.66666666666666
i3 66.66666666666666
i4 66.66666666666666
i2,i3 100.0
i2,i4 100.0
i3,i4 100.0
i2,i3,i4 100.0
Rules generated with  min_sup = 5 and  min_conf = 30.0
i2   ==>    i4 & i3
i3   ==>    i4 & i2
i4   ==>    i2 & i3
i2,i3   ==>    i4
i2,i4   ==>    i3
i3,i4   ==>    i2
i2,i3,i4   ==>    i2 & i3 & i4
```

Hash based Technique

When scanning each transaction in the database to generate the frequent 1-itemsets, L1, we can generate all the 2-itemsets for each transaction, hash (i.e., map) them into the different buckets of a hash table structure, and increase the corresponding bucket counts. A 2-itemset with a corresponding bucket count in the hash table that is below the support threshold cannot be frequent and thus should be removed from the candidate set.

We used itemset of size 2 for frequent set generation using the below hash function.

$$H(x,y) = ((\text{Order of first})*10 + (\text{Order of second})) \bmod 7$$

Refer hashing.py for implementation details.

Rule generation using WEKA

Data Source:

We used transactions data available at **SPFM** website http://www.philippe-fournier-viger.com/spmf/index.php **.** SPMF is an open-source software and data mining library written in Java, specialized in pattern mining (the discovery of patterns in data)

Library Used: WEKA library is used for rule generation using SPFM data.

Script: SPFM2WEKA parser is written to convert data in SPFM format to WEKA format. Refer the script convertspfm2weka.py

Example#1: - SIGN data

A dataset of sign language utterance containing approximately 800 sequences and 267 items. The original dataset file in another format can be obtained here with more details on this dataset.

```
%%sh
wget http://www.philippe-fournier-viger.com/spmf/datasets/SIGN.txt
```

```
--2020-10-27 05:54:05--  http://www.philippe-fournier-viger.com/spmf/datasets/SIGN.txt
Resolving www.philippe-fournier-viger.com (www.philippe-fournier-viger.com)... 74.208.236.167
Connecting to www.philippe-fournier-viger.com (www.philippe-fournier-viger.com)|74.208.236.167|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 236727 (231K) [text/plain]
Saving to: 'SIGN.txt'

     0K .......... .......... .......... .......... .......... 21%  134K 1s
    50K .......... .......... .......... .......... .......... 43%  268K 1s
   100K .......... .......... .......... .......... .......... 64%  494K 0s
   150K .......... .......... .......... .......... .......... 86%  590K 0s
   200K .......... .......... .......... .                    100%  810K=0.8s

2020-10-27 05:54:06 (295 KB/s) - 'SIGN.txt' saved [236727/236727]
```

This raw data is converted to WEKA format, where Binary matrix of transactions and items is created.

	10	100	101	102	103	104	105	106	107	108	109	11	110	111	112	113	114	115	116	117	118	119	12	120	121	122
0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	t	?	?	?	t	?	?	?	?	?
1	t	?	?	?	?	?	?	?	?	?	?	?	?	?	?	t	?	?	?	t	t	?	?	?	?	?
2	t	?	?	?	?	?	?	?	?	?	?	?	?	?	?	t	t	?	?	t	t	?	t	?	?	?
3	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	t	t	?	?	t	t	?	t	?	?	?
4	?	?	?	?	?	?	?	?	?	?	?	t	?	?	?	t	t	?	?	t	t	?	?	?	?	?

Here each cell[i,j] represents if the transaction ‘i’ has item ‘j’ . Presence is represented as ‘t’ and absence as ‘?’.

Rules:

Below Rules generated in the WEKA tool with given support and confidence.

Associator output

```
=== Associator model (full training set) ===


Apriori
=======

Minimum support: 0.7 (511 instances)
Minimum metric <confidence>: 0.9
Number of cycles performed: 6

Generated sets of large itemsets:

Size of set of large itemsets L(1): 10

Size of set of large itemsets L(2): 13

Best rules found:

 1. 18=t 552 ==> 17=t 516    <conf:(0.93)> lift:(1.05) lev:(0.03) [22] conv:(1.59)
 2. 117=t 575 ==> 143=t 529    <conf:(0.92)> lift:(1.07) lev:(0.05) [35] conv:(1.74)
 3. 26=t 601 ==> 253=t 548    <conf:(0.91)> lift:(1) lev:(0) [1] conv:(1.01)
 4. 35=t 595 ==> 253=t 541    <conf:(0.91)> lift:(1) lev:(-0) [0] conv:(0.98)
 5. 143=t 626 ==> 253=t 569    <conf:(0.91)> lift:(1) lev:(-0) [0] conv:(0.98)
 6. 35=t 595 ==> 17=t 539    <conf:(0.91)> lift:(1.01) lev:(0.01) [7] conv:(1.12)
 7. 117=t 575 ==> 253=t 519    <conf:(0.9)> lift:(0.99) lev:(-0.01) [-4] conv:(0.91)
 8. 17=t 652 ==> 253=t 588    <conf:(0.9)> lift:(0.99) lev:(-0.01) [-5] conv:(0.91)
 9. 117=t 575 ==> 17=t 518    <conf:(0.9)> lift:(1.01) lev:(0.01) [4] conv:(1.06)
10. 26=t 601 ==> 17=t 541    <conf:(0.9)> lift:(1.01) lev:(0.01) [4] conv:(1.05)
```

FP-Growth Algorithm

Apriori Algorithm has slow performance and has below drawbacks

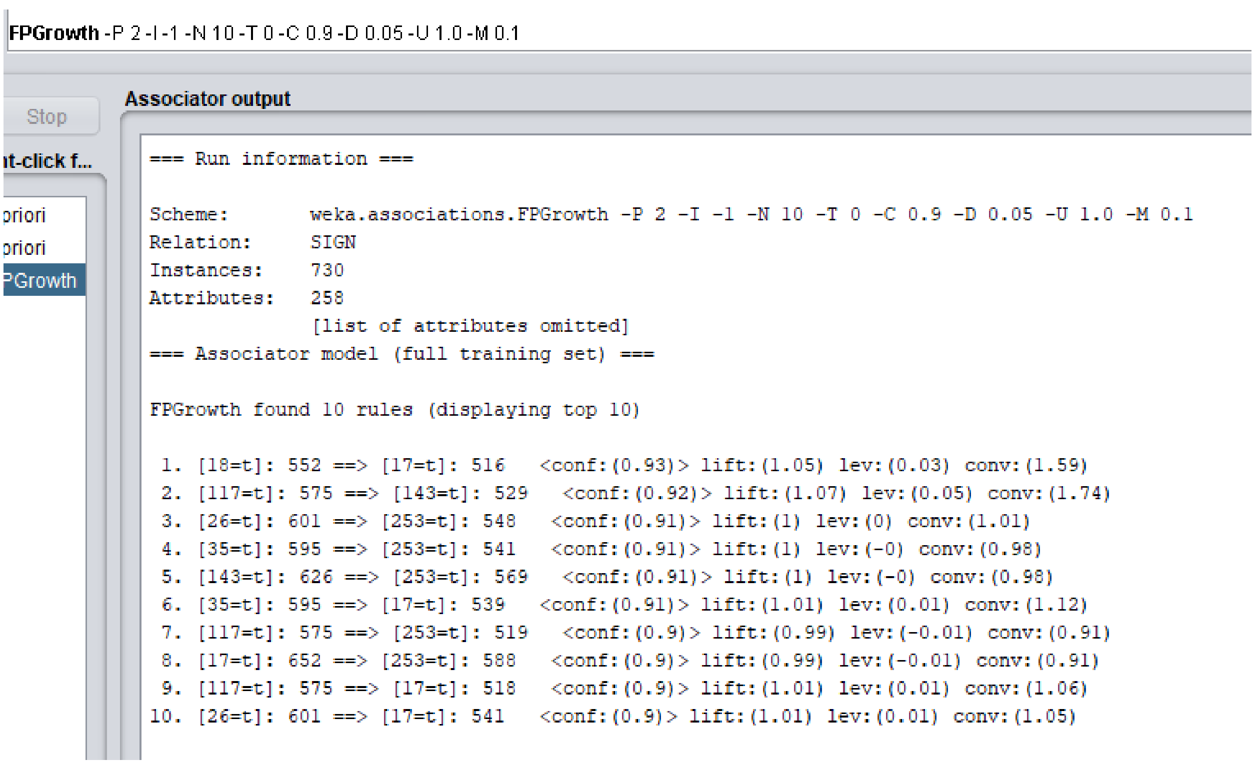
- At each step, candidate sets must be built.
- To build the candidate sets, the algorithm must repeatedly scan the database.

These two properties inevitably make the algorithm slower. To overcome these redundant steps, a new association-rule mining algorithm was developed named Frequent Pattern Growth Algorithm. It overcomes the disadvantages of the Apriori algorithm by storing all the transactions in a Tree Data Structure.

Rule generation using WEKA

Rules:

Below Rules generated in the WEKA tool with given support and confidence.




```
=== Run information ===

Scheme:       weka.associations.FPGrowth -P 2 -I -1 -N 10 -T 0 -C 0.9 -D 0.05 -U 1.0 -M 0.1
Relation:     SIGN
Instances:    730
Attributes:   258
              [list of attributes omitted]
=== Associator model (full training set) ===

FPGrowth found 10 rules (displaying top 10)

 1. [18=t]: 552 ==> [17=t]: 516   <conf:(0.93)> lift:(1.05) lev:(0.03) conv:(1.59)
 2. [117=t]: 575 ==> [143=t]: 529   <conf:(0.92)> lift:(1.07) lev:(0.05) conv:(1.74)
 3. [26=t]: 601 ==> [253=t]: 548   <conf:(0.91)> lift:(1) lev:(0) conv:(1.01)
 4. [35=t]: 595 ==> [253=t]: 541   <conf:(0.91)> lift:(1) lev:(-0) conv:(0.98)
 5. [143=t]: 626 ==> [253=t]: 569   <conf:(0.91)> lift:(1) lev:(-0) conv:(0.98)
 6. [35=t]: 595 ==> [17=t]: 539   <conf:(0.91)> lift:(1.01) lev:(0.01) conv:(1.12)
 7. [117=t]: 575 ==> [253=t]: 519   <conf:(0.9)> lift:(0.99) lev:(-0.01) conv:(0.91)
 8. [17=t]: 652 ==> [253=t]: 588   <conf:(0.9)> lift:(0.99) lev:(-0.01) conv:(0.91)
 9. [117=t]: 575 ==> [17=t]: 518   <conf:(0.9)> lift:(1.01) lev:(0.01) conv:(1.06)
10. [26=t]: 601 ==> [17=t]: 541   <conf:(0.9)> lift:(1.01) lev:(0.01) conv:(1.05)
```

Rule generation using implementation from scratch

FP growth Algorithm:

FP algorithm is implemented from scratch in python. Generated Condition FP trees using Bottom UP and Top Down approach:

Example 1:

```
test_data = [['I1','I2','I5'],
             ['I2','I3','I4'],
             ['I3','I4'],
             ['I1','I2','I3','I4']]
```

Bottom up Approach:
Condtional FPTree Root on I1 : ['Null 1', ['I2 2']]
Condtional FPTree Root on I4 : ['Null 1', ['I3 3', ['I2 2']]]
Condtional FPTree Root on I3 : ['Null 1', ['I2 2']]

Top Down Approach:
Condtional FPTree Root on I1: None
Condtional FPTree Root on I4: ['I1:1']
Condtional FPTree Root on I3: [['I1:1'], 'I4:2']
Condtional FPTree Root on I2: [[['I1:1'], 'I4:2'], 'I3:2', 'I1:1']

Example 2:

```
test_data = [['I1','I2','I5'],
             ['I2','I4'],
             ['I2','I3'],
             ['I1','I2','I4'],
             ['I1','I3'],
             ['I2','I3'],
             ['I1','I3'],
             ['I1','I2','I3','I5'],
             ['I1','I2','I3']]
```

Bottom up Approach:
Condtional FPTree Root on I1:['Null 1', ['I2 2']]
Condtional FPTree Root on I4:['Null 1', ['I3 3', ['I2 2']]]
Condtional FPTree Root on I3:['Null 1', ['I2 2']]

Top Down Approach:
Condtional FPTree Root on I1: None
Condtional FPTree Root on I4: ['I1:1']

Condtional FPTree Root on I3: [['I1:1'], 'I4:2']
Condtional FPTree Root on I2: [[['I1:1'], 'I4:2'], 'I3:2', 'I1:1']

Example 3:

```
test_data = [['A','B'],
             ['B','C','D'],
             ['A','C','D','E'],
             ['A','D','E'],
             ['A','B','C']]
```

Bottom up Approach:
Condtional FPTree Root on E: ['Null 1', ['A 2', ['D 2']]]
Condtional FPTree Root on D: ['Null 1', ['C 1'], ['A 2', ['C 1']]]
Condtional FPTree Root on C: ['Null 1', ['B 1'], ['A 2', ['B 1']]]
Condtional FPTree Root on B: ['Null 1', ['A 2']]

Top Down Approach:
Condtional FPTree Root on E : None
Condtional FPTree Root on D : ['E:1']
Condtional FPTree Root on C : [['E:1'], 'D:1']
Condtional FPTree Root on B : ['C:1']
Condtional FPTree Root on A : [['E:1'], 'D:1', [['E:1'], 'D:1'], 'C:1', ['C:1'], 'B:2']

Optimization and Comparison of performance:
Below is the approach followed in implementation of Fp-growth algorithm Traditional one (Bottom up approach):

- 1: Find the ordered frequent items. For items with the same frequency, order is given based on the alphabetical order.
- 2: Develop the FP-tree from the above data
- 3: From FP-tree, deduce the FP-conditional tree for each item
- 4: Identify the frequent patterns

In top bottom approach: Traversed the item from top to bottom (until leaf node). For finding the Conditional FP patterns, we need to traverse the tree for all its children from top to bottom for each branch. Also, while developing FP tree, each node is linked to its another instance if they are placed in another branch. So, for traversing the children, multiple branches are traversed and selected only required items, whereas from Bottom to Top approach, where we do traverse from leaf to root, only one time traversal is fine.

Bonus

This part is implemented using library. For detailed steps refer bonus_using_library.py

```
In [12]: runfile('F:/DA/bonus.py', wdir='F:/DA')
      I2     I3     I4     I1     I2     I3
0   True   True   True   True  False  False
1   True  False   True   True  False  False
2   True  False  False   True  False  False
3  False   True   True  False   True  False
4  False   True  False  False   True  False
5  False  False   True  False  False   True
6  False  False   True  False   True  False
Time to find frequent itemset
--- 0.004002332687377393 seconds ---
[0.71428571 0.42857143 0.14285714 0.28571429]
Time to find Close frequent itemset
--- 0.003999233245849609 seconds ---
Time to find Max frequent itemset
--- 0.004000186920166016 seconds ---
```

Comparative case study

We used SIGN.txt from SPFM library to compare the timing using Apriori and FP_Growth algorithms.

Below table indicates the result at various support and confidence levels. Results of the experiments are attached in SIGN_Timings.txt

Data	Algorithm	support	confidenence	
SIGN.TXT	Apriori	25	60	543 ms
	FP_Growth	25	60	1156 ms
	Apriori	50	50	55 ms
	FP_Growth	50	50	29 ms
	Apriori	60	60	16 ms
	FP_Growth	60	60	41 ms
	Apriori	75	75	8 ms
	FP_Growth	75	75	10 ms

Results of the experiments are attached in Mushroom_Timings.txt . In this case FP_Growth is faster than Apriori.

Mushroom.TXT	Apriori	60	60	60 ms
	FP_Growth	60	60	48 ms
	Apriori	40	70	129 ms
	FP_Growth	40	70	58 ms
	Apriori	20	40	13634 ms
	FP_Growth	20	40	86 ms

Results of the experiments are attached in chess_Timings.txt . In this case FP_Growth is faster than Apriori by 1000x times.

Chess.TXT	Apriori	80	40	2024 ms
	FP_Growth	80	40	41 ms
	Apriori	75	50	4934 ms
	FP_Growth	75	50	47 ms
	Apriori	75	50	4934 ms
	FP_Growth	75	50	47 ms