Machine Learning Lab (CS353) Report Format
Frequent Pattern (FP) Growth Algorithm

Team Members:

1. Name: Akshat Nambiar, Reg no.: 181CO204
2. Name: Mohammed Rushad, Reg no.: 181CO232

1. FP Growth – INTRODUCTION

For Data Mining, an important approach is finding frequently occurring patterns in large datasets or association rule mining. But this task can be computationally very expensive, especially if there exist a large number of patterns in the dataset. It is used as an analytical process that finds frequent patterns or associations from data sets.

The FP-Growth Algorithm, proposed in 2000, is an efficient and scalable method for mining the frequent patterns sets completely by pattern fragment growth. This is done using an extended prefix-tree structure for storing compressed and important information about the frequent patterns and is named the Frequent-Pattern Tree, FP-Tree. It was shown that this method, outperforms other methods that were popular at the time, for mining frequent patterns like the Apriori Algorithm and the TreeProjection Algorithm. It was also proved that FP-Growth has better performance than other methods, including Eclat and Relim.

The Apriori Algorithm had a major drawback; using Apriori required multiple scans of the database to check the support count of each item and itemsets. When the database is huge, this costs a significant amount of disk I/O and computing power. Therefore the FP-Growth algorithm was created to overcome this drwback. It only scans the database twice and uses the FP-tree to store all the information. The root represents null and each node represents an item, while the association of the nodes is the itemsets with the order maintained while forming the tree. Once an FP-tree is constructed, we can use a recursive divide-and-conquer approach to mine the frequent itemsets.

The FP-Tree

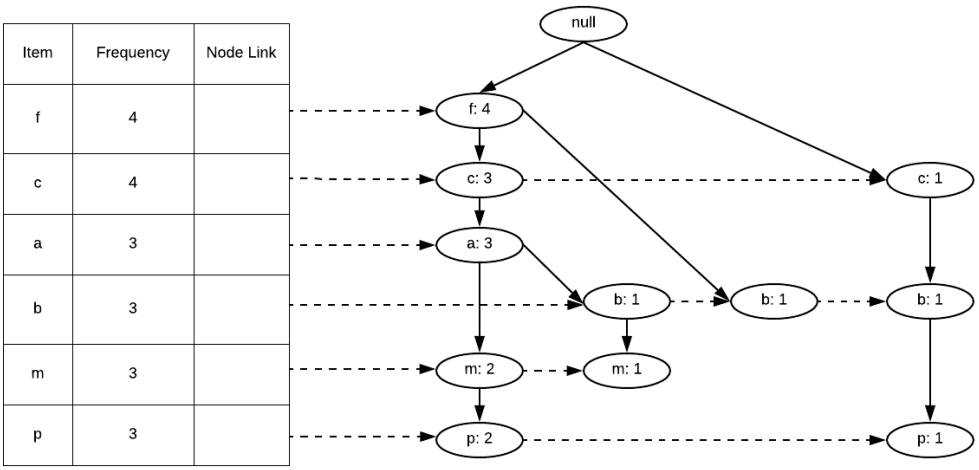
The FP-tree is a compact structure that stores quantitative information about frequent patterns in a dataset.

The general structure is as follows:

- The root node labeled as “null” that has a set of item-prefix subtrees as its children and a frequent-item-header table.
- All nodes in the item-prefix subtree have the following three fields:
    - Item-name: Name of item is represented by the node.
    - Count: Number of transactions represented by the portion of the path that reaches the node.
    - Node-link: It is the link to the next node in the FP-tree that has same item-name, or null if there is none.

- Each entry in the frequent-item-header table consists of two fields:
    - Item-name: Same as the node.
    - Head of the node-link: It is a pointer to the first node in the FP-tree having the item-name.

Example:



2. FP-Growth Algorithm

The FP Growth Algorithm consists of two parts:

1. The FP-Tree construction from the dataset.
2. The actual FP Growth algorithm using the FP-Tree.

FP-Tree Construction:

- Scan the transaction database DB once. Collect F, the set of frequent items, and the support of each frequent item. Sort F in descending order of support as FList i.e. the list of frequent items.
- Make the root of an FP-tree, T, and label it as “null”. For each transaction *Trans* in DB do the following:
    - Select the frequent items in *Trans* and sort them according to the order of FList. Let the sorted frequent-item list in *Trans* be [p | P], where p is the first element and P is the remaining list. Call insertTree([p | P], T).
    - The function insertTree([p | P], T) is performed as follows:
        - If T has a child N such that N.item-name = p.item-name, then increment N ’s count by 1; else create a new node N , with its count initialized to 1, its parent link linked to T , and its node-link linked to the nodes with the same item-name via the node-link structure. If P is nonempty, call insertTree(P, N) recursively.

FP-Growth Algorithm:

After constructing the FP-Tree we need to mine it to find the complete set of frequent patterns using the algorithm below that takes the tree and minimum support threshold as input:

FP-Growth(Tree, a)

-If Tree contains a single prefix path then

1. let P be the single prefix-path part of Tree;
2. let Q be the multipath part with the top branching node replaced by a null root;
3. for each combination (denoted as B) of the nodes in the path P do
4. generate pattern B ∪ a with support = minimum support of nodes in B;
5. let frequent pattern set(P) be the set of patterns so generated;

-else let Q be Tree:

-for each item ai in Q do

1. generate pattern B = ai ∪ a with support = ai.support;
2. construct B's conditional pattern-base and then ß's conditional FP-tree Tree B.
3. If Tree B is non-empty then
4. call FP-growth(Tree B , B);
5. let frequent pattern set(Q) be the set of patterns so generated;

-return (frequent pattern set(P) ∪ frequent pattern set(Q) ∪ (frequent pattern set(P) × frequent pattern set(Q)))

3. PROS & CONS of FP-Growth

The FP Growth algorithm was developed to compensate for the shortcomings of the Apriori algorithms. Some of the advantages of the FP-Growth are:

1. The algorithm only needs to scan the database twice, while Apriori algorithm needs to scan the database after each iteration.
2. Low memory utilisation as it stores the database compactly in memory.
3. It is scalable and efficient for both long and short frequent pattern mining.
4. Faster than Apriori as it does not need to implement the pairing of the items.

However, there are a few shortcomings of the FP-Growth algorithm, such as:

1. It is more difficult and heavier to implement as compared to Apriori.
2. A large database may not fit in a shared memory.
3. Expensive to implement.

4. APPLICATIONS of FP-Growth

The FP Growth Algorithm is used in a variety of applications. Some of the are:

- Basket Data Analysis
- Cross Marketing
- Catalog Design
- Sales Campaign Analysis
- Web Log Analysis
- DNA Sequencing Analysis

5. FP-Growth NUMERICAL ANALYSIS

5.1 DATASET DESCRIPTION

The dataset used for analysis is a randomly generated transaction table as follows, for some items A, B, C, D, E, F:

Transaction ID	Items
1	A B D E F
2	A B D E
3	B C E
4	A B C E
5	B C D
6	A B C D E

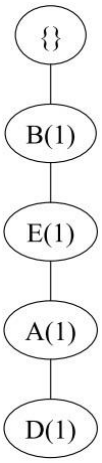
5.2 Numerical Solution and Analysis

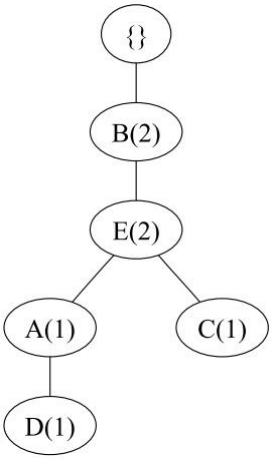
Step1: Create Frequency list for supports of the items, given minimum support = 3.

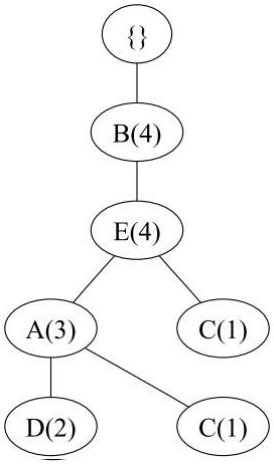
A – 4, B – 6, C – 4, D – 4, E – 5, F – 1

Step2: Construct FP-Tree, iteratively

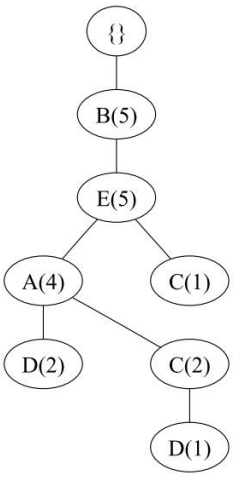
Transaction 1 Transaction 2 Transaction 3



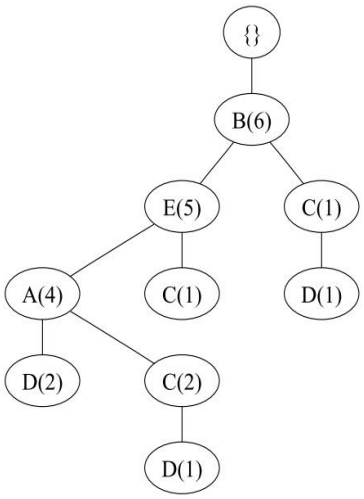




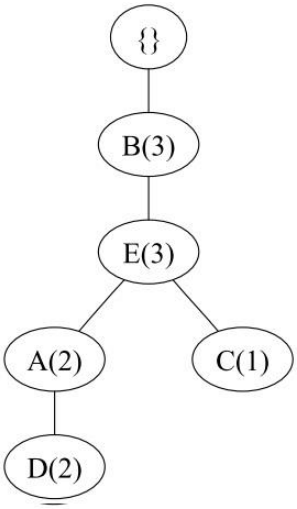
Transaction 5



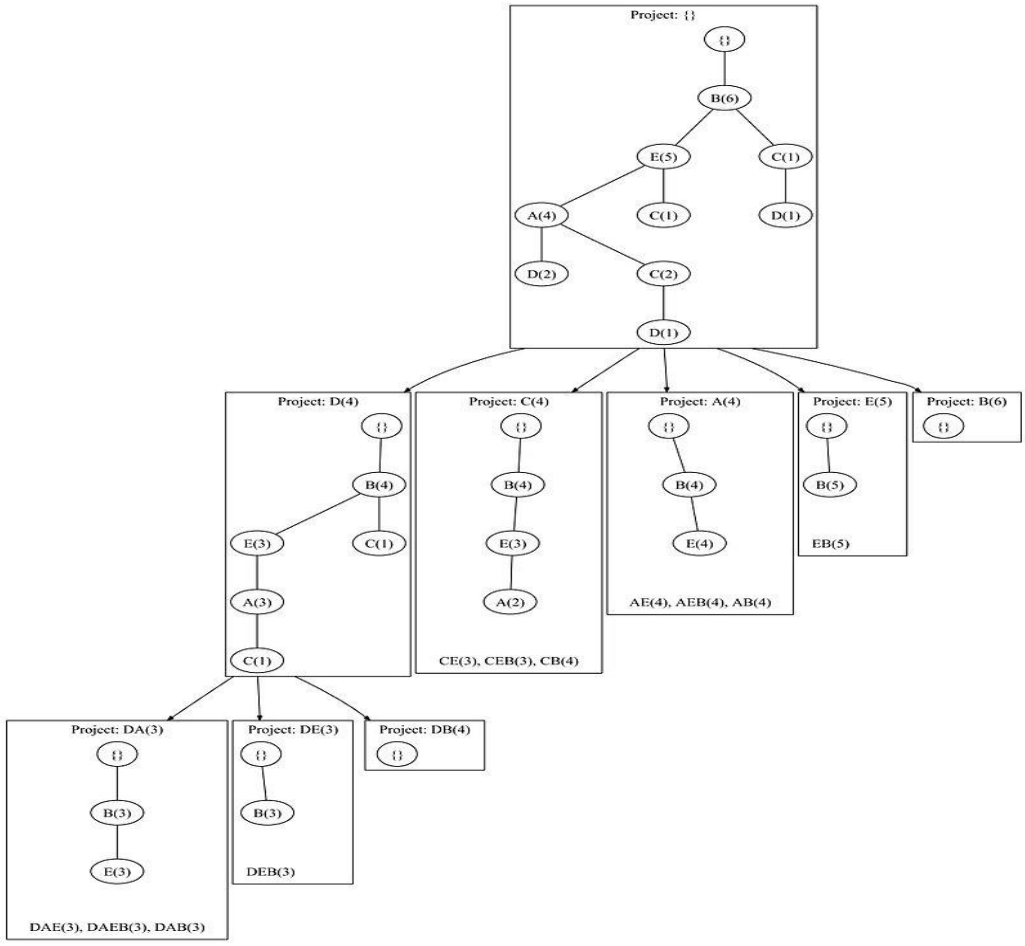
Transaction 5



Transaction 6



Final Tree



Therefore, the following frequent patterns are generated:

A, AE, AB, ABE, D, AD, DE, BD, ADE, ABD, BDE, ABDE, C, CE, BC, BCE, E, BE, B for a minimum support of 3.

6.IMPLEMENTATION

6.1 Python-FP-Growth Implementation without Libraries

Code:

```
import time

class TreeNode:
    def __init__(self, Node_name,counter,parentNode):
        self.name = Node_name
        self.count = counter
        self.nodeLink = None
        self.parent = parentNode
        self.children = {}

    def increment_counter(self, counter):
        self.count += counter
```

```
def create_FPTree(dataset, minSupport):
    HeaderTable = {}
    for transaction in dataset:
        for item in transaction:
            HeaderTable[item] = HeaderTable.get(item,0) + dataset[transaction]
    for k in list(HeaderTable):
        if HeaderTable[k] < minSupport:
            del(HeaderTable[k])

    frequent_itemset = set(HeaderTable.keys())

    if len(frequent_itemset) == 0:
        return None, None

    for k in HeaderTable:
        HeaderTable[k] = [HeaderTable[k], None]

    retTree = TreeNode('Null Set',1,None)
    for itemset,count in dataset.items():
        frequent_transaction = {}
```

```
        for item in itemset:
            if item in frequent_itemset:
                frequent_transaction[item] = HeaderTable[item][0]
        if len(frequent_transaction) > 0:
            #to get ordered itemsets form transactions
            ordered_itemset = [v[0] for v in sorted(frequent_transaction.items(), key=lambda p: p[1], reverse=True)]
            #to update the FPTree
            updateTree(ordered_itemset, retTree, HeaderTable, count)
    return retTree, HeaderTable

def updateTree(itemset, FPTree, HeaderTable, count):
    if itemset[0] in FPTree.children:
        FPTree.children[itemset[0]].increment_counter(count)
    else:
        FPTree.children[itemset[0]] = TreeNode(itemset[0], count, FPTree)

        if HeaderTable[itemset[0]][1] == None:
            HeaderTable[itemset[0]][1] = FPTree.children[itemset[0]]
        else:
            update_NodeLink(HeaderTable[itemset[0]][1], FPTree.children[itemset[0]])

    if len(itemset) > 1:
        updateTree(itemset[1::], FPTree.children[itemset[0]], HeaderTable, count)
```

```
def update_NodeLink(Test_Node, Target_Node):
    while (Test_Node.nodeLink != None):
        Test_Node = Test_Node.nodeLink

    Test_Node.nodeLink = Target_Node

def FPTree_uptransveral(leaf_Node, prefixPath):
 if leaf_Node.parent != None:
    prefixPath.append(leaf_Node.name)
    FPTree_uptransveral(leaf_Node.parent, prefixPath)

def find_prefix_path(basePat, TreeNode):
 Conditional_patterns_base = {}

 while TreeNode != None:
    prefixPath = []
    FPTree_uptransveral(TreeNode, prefixPath)
    if len(prefixPath) > 1:
        Conditional_patterns_base[frozenset(prefixPath[1:])] = TreeNode.count
    TreeNode = TreeNode.nodeLink

 return Conditional_patterns_base
```

```
def Mine_Tree(FPTree, HeaderTable, minSupport, prefix, frequent_itemset):
    bigL = [v[0] for v in sorted(HeaderTable.items(),key=lambda p: p[1][0])]
    for basePat in bigL:
        new_frequentset = prefix.copy()
        new_frequentset.add(basePat)
        frequent_itemset.append(new_frequentset)
        Conditional_pattern_bases = find_prefix_path(basePat, HeaderTable[basePat][1])
        Conditional_FPTree, Conditional_header = create_FPTree(Conditional_pattern_bases,minSupport)

        if Conditional_header != None:
            Mine_Tree(Conditional_FPTree, Conditional_header, minSupport, new_frequentset, frequent_itemset)

def Load_data(filename):
    with open(filename) as f:
        content = f.readlines()

    content = [x.strip() for x in content]
    Transaction = []

    for i in range(0, len(content)):
        Transaction.append(content[i].split())

    return Transaction
```

```
def create_initialset(dataset):
    retDict = {}
    for trans in dataset:
        retDict[frozenset(trans)] = 1
    return retDict

print("Enter the filename:")
filename = input()
print("Enter the minimum support count:")
min_Support = int(input())
initSet = create_initialset(Load_data(filename))
FPtree, HeaderTable = create_FPTree(initSet, min_Support)
frequent_itemset = []
Mine_Tree(FPtree, HeaderTable, min_Support, set([]), frequent_itemset)

print("The patterns generated are:")
for f in frequent_itemset:
  print(f)
```

Output:

```
The patterns generated are:

('A',)
('A', 'E')
('A', 'B')
('A', 'B', 'E')
('D',)
('A', 'D')
('D', 'E')
('B', 'D')
('A', 'D', 'E')
('A', 'B', 'D')
('B', 'D', 'E')
('A', 'B', 'D', 'E')
('C',)
('C', 'E')
('B', 'C')
('B', 'C', 'E')
('E',)
('B', 'E')
('B',)
```

Therefore, the following frequent patterns are generated without libraries are:

A, AE, AB, ABE, D, AD, DE, BD, ADE, ABD, BDE, ABDE, C, CE, BC, BCE, E, BE, B for a minimum support of 3.

6.2 Python- FP-Growth Implementation with Libraries

Code:

```
import numpy as np
import pandas as pd

!pip install pyfpgrowth

import pyfpgrowth

transactions = [['F','B','A','E','D'], ['B','C','E'],
['A','B','D','E'],
['A','B','C','E'],
['A','B','C','D','E'],
['B','C','D']]

patterns = pyfpgrowth.find_frequent_patterns(transactions, 3)
print("The patterns generated are:\n")
for p in patterns:
  print(p)
```

Output:

```
Requirement already satisfied: pyfpgrowth in /usr/local/lib/python3.7/dist-packages (1.0)
The patterns generated are:

('A',)
('A', 'E')
('A', 'B')
('A', 'B', 'E')
('D',)
('A', 'D')
('D', 'E')
('B', 'D')
('A', 'D', 'E')
('A', 'B', 'D')
('B', 'D', 'E')
('A', 'B', 'D', 'E')
('C',)
('C', 'E')
('B', 'C')
('B', 'C', 'E')
('E',)
('B', 'E')
('B',)
```

Therefore, the following frequent patterns are generated using Python Library pyfpgrowth:

A, AE, AB, ABE, D, AD, DE, BD, ADE, ABD, BDE, ABDE, C, CE, BC, BCE, E, BE, B

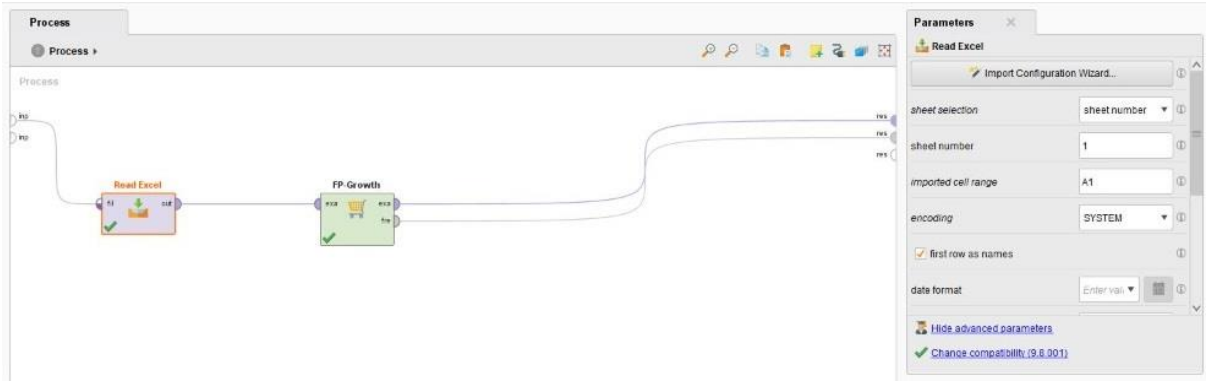
for a minimum support of 3.

6.3 Rapidminer Implementation

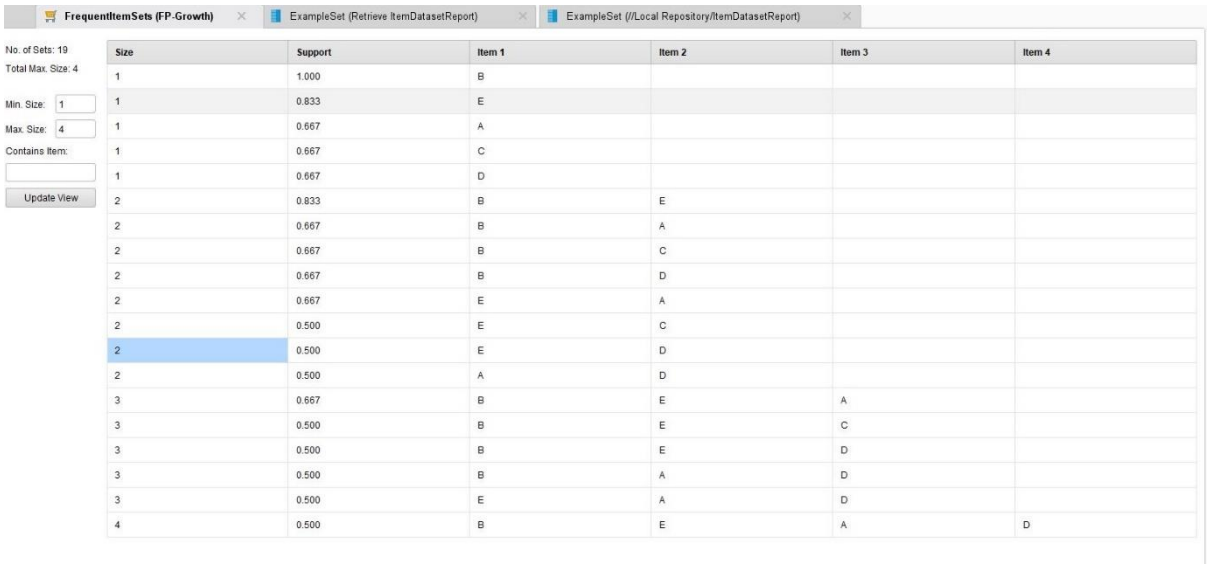
1.Initial Data

Row No.	Transaction ...	ITEMSet
1	id1	F|B|A|E|D
2	id2	B|C|E
3	id3	A|B|D|E
4	id4	A|B|C|E
5	id5	A|B|C|D|E
6	id6	B|C|D

2. Model Design and Parameters



3.Output: Frequent Patterns Generated



Size	Support	Item 1	Item 2	Item 3	Item 4
1	1.000	B			
1	0.833	E			
1	0.667	A			
1	0.667	C			
1	0.667	D			
2	0.833	B	E		
2	0.667	B	A		
2	0.667	B	C		
2	0.667	B	D		
2	0.667	E	A		
2	0.500	E	C		
2	0.500	E	D		
2	0.500	A	D		
3	0.667	B	E	A	
3	0.500	B	E	C	
3	0.500	B	E	D	
3	0.500	B	A	D	
3	0.500	E	A	D	
4	0.500	B	E	A	D

Hence, the following frequent patterns are generated for RapidMiner are:

A, AE, AB, ABE, D, AD, DE, BD, ADE, ABD, BDE, ABDE, C, CE, BC, BCE, E, BE, B for a minimum support of 3.

7.References

1. WikiBooks: https://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Frequent_Pattern_Mining/The_FP-Growth_Algorithm#:~:text=The%20FP%2DGrowth%20Algorithm%2C%20proposed,tree%20(FP%2Dtree).
2. https://www.mygreatlearning.com/blog/understanding-fp-growth-algorithm/
3. https://www.geeksforgeeks.org/ml-frequent-pattern-growth-algorithm/