```
import numpy as np
from collections import defaultdict
from itertools import chain, combinations


datas = {
    'T100':['I1','I2','I5'],
    'T200':['I2','I4'],
    'T300':['I2','I3'],
    'T400':['I1','I2','I4'],
    'T500':['I1','I3'],
    'T600':['I2','I3'],
    'T700':['I1','I3'],
    'T800':['I1','I2','I3','I5'],
    'T900':['I1','I2','I3'],
}

class Node:
    def __init__(self, itemName, frequency, parentNode):
        self.itemName = itemName
        self.count = frequency
        self.parent = parentNode
        self.children = {}
        self.next = None

    def increment(self, frequency):
        self.count += frequency

    def display(self, ind=1):
        print('  ' * ind, self.itemName, ' ', self.count)
        for child in list(self.children.values()):
            child.display(ind+1)

order = ['I' + str(i) for i in range(1, 6)]

def getFromFile():
  itemSetList=[]
  frequency=[]
  for keys in datas.keys():
    unique_datas = list(np.unique(datas[keys]))
    unique_datas.sort(key=lambda x: order.index(x))
    itemSetList.append(unique_datas)
    frequency.append(1)
  return itemSetList,frequency


def constructTree(itemSetList, frequency, minSup):
    headerTable = defaultdict(int)
    # Counting frequency and create header table
    for idx, itemSet in enumerate(itemSetList):
        for item in itemSet:
            headerTable[item] += frequency[idx]

    # Deleting items below minSup
```

```
        headerTable = dict((item, sup) for item, sup in headerTable.items() if sup >= minSup)
        # print(headerTable)
        if(len(headerTable) == 0):
            return None, None

        # HeaderTable column [Item: [frequency, headNode]]
        for item in headerTable:
            headerTable[item] = [headerTable[item], None]

        # Init Null head node
        fpTree = Node('Null', 1, None)
        # Update FP tree for each cleaned and sorted itemSet
        for idx, itemSet in enumerate(itemSetList):
            itemSet = [item for item in itemSet if item in headerTable]
            itemSet.sort(key=lambda item: headerTable[item][0], reverse=True)
            # Traverse from root to leaf, update tree with given item
            currentNode = fpTree
            for item in itemSet:
                currentNode = updateTree(item, currentNode, headerTable, frequency[idx])

        return fpTree, headerTable

    def updateTree(item, treeNode, headerTable, frequency):
        if item in treeNode.children:
            # If the item already exists, increment the count
            treeNode.children[item].increment(frequency)
        else:
            # Create a new branch
            newItemNode = Node(item, frequency, treeNode)
            treeNode.children[item] = newItemNode
            # Link the new branch to header table
            updateHeaderTable(item, newItemNode, headerTable)

        return treeNode.children[item]

    def updateHeaderTable(item, targetNode, headerTable):
        if(headerTable[item][1] == None):
            headerTable[item][1] = targetNode
        else:
            currentNode = headerTable[item][1]
            # Traverse to the last node then link it to the target
            while currentNode.next != None:
                currentNode = currentNode.next
            currentNode.next = targetNode

    def newTree(itemSetList, frequency, minSup):
        headerTable = defaultdict(int)
        # Counting frequency and create header table
        for idx, itemSet in enumerate(itemSetList):
            for item in itemSet:
                headerTable[item] += frequency[idx]

        # Deleting items below minSup
        headerTable = dict((item, sup) for item, sup in headerTable.items() if sup > minSup)
        # print(headerTable)
        if(len(headerTable) == 0):
```

```
        if(len(headerTable) == 0):
            return None, None

        # HeaderTable column [Item: [frequency, headNode]]
        for item in headerTable:
            headerTable[item] = [headerTable[item], None]

        # Init Null head node
        fpTree = Node('Null', 1, None)
        # Update FP tree for each cleaned and sorted itemSet
        for idx, itemSet in enumerate(itemSetList):
            itemSet = [item for item in itemSet if item in headerTable]
            itemSet.sort(key=lambda item: headerTable[item][0], reverse=True)
            # Traverse from root to leaf, update tree with given item
            currentNode = fpTree
            for item in itemSet:
                currentNode = updateTree(item, currentNode, headerTable, frequency[idx])

        return fpTree, headerTable


    def mineTree(headerTable, minSup, preFix, freqItemList):
        # Sort the items with frequency and create a list
        sortedItemList = [item[0] for item in sorted(list(headerTable.items()), key=lambda p:p
        # Start with the lowest frequency
        for item in sortedItemList:
            # Pattern growth is achieved by the concatenation of suffix pattern with frequent
            newFreqSet = preFix.copy()
            newFreqSet.add(item)
            freqItemList.append(newFreqSet)
            # Find all prefix path, constrcut conditional pattern base
            conditionalPattBase, frequency = findPrefixPath(item, headerTable)
            if len(conditionalPattBase)>0:
              print(f"{conditionalPattBase} => {frequency}")

            # Construct conditonal FP Tree with conditional pattern base
            conditionalTree, newHeaderTable = constructTree(conditionalPattBase, frequency, mi
            if newHeaderTable != None:
                # Mining recursively on the tree
                mineTree(newHeaderTable, minSup,
                            newFreqSet, freqItemList)

    def findPrefixPath(basePat, headerTable):
        # First node in linked list
        treeNode = headerTable[basePat][1]
        condPats = []
        frequency = []
        while treeNode != None:
            prefixPath = []
            # From leaf node all the way to root
            ascendFPtree(treeNode, prefixPath)
            if len(prefixPath) > 1:
                # Storing the prefix path and it's corresponding count
                condPats.append(prefixPath[1:])
                frequency.append(treeNode.count)
```

```
            # Go to next node
            treeNode = treeNode.next
        return condPats, frequency

    def ascendFPtree(node, prefixPath):
        if node.parent != None:
            prefixPath.append(node.itemName)
            ascendFPtree(node.parent, prefixPath)


    def powerset(s):
        return chain.from_iterable(combinations(s, r) for r in range(1, len(s)))

    def getSupport(testSet, itemSetList):
        count = 0
        for itemSet in itemSetList:
            if(set(testSet).issubset(itemSet)):
                count += 1
        return count

    def tentitiveRule(freqItemSet, itemSetList, minConf):
        rules = []
        for itemSet in freqItemSet:
            subsets = powerset(itemSet)
            itemSetSup = getSupport(itemSet, itemSetList)
            for s in subsets:
                  confidence = float(itemSetSup / getSupport(s, itemSetList))
                  rules.append([set(s), set(itemSet.difference(s)), confidence])
        return rules

    def associationRule(freqItemSet, itemSetList, minConf):
        rules = []
        for itemSet in freqItemSet:
            subsets = powerset(itemSet)
            itemSetSup = getSupport(itemSet, itemSetList)
            for s in subsets:
                confidence = float(itemSetSup / getSupport(s, itemSetList))*100
                if(confidence > ((minConf)*100)):
                    rules.append([set(s), set(itemSet.difference(s)), confidence])
        return rules


    def fpgrowthFromFile(minSup, minConf):
        itemSetList, frequency = getFromFile()
        fpTree, headerTable = constructTree(itemSetList, frequency, minSup)
        freqItems = []
        print("Conditional Pattern Base")
        mineTree(headerTable, minSup, set(), freqItems)

        print("\n FP Growth Tree is given below")
        fpTree.display()

        tentitive_rules = tentitiveRule(freqItems,itemSetList,minConf)
        association_rules = associationRule(freqItems, itemSetList, minConf)
        return freqItems, tentitive_rules,association_rules
```

```
    # return freqItems

minSup=float(input("Enter min Support: "))
print("value of minimum support is",minSup)

minConf=float(input("Enter min Conf: "))
print("value of minimum confidence is",minConf)

freqItems, tentitive_rules, association_rules=fpgrowthFromFile(minSup,minConf)
```

```
 Enter min Support: 2
 value of minimum support is 2.0
 Enter min Conf: 0.4
 value of minimum confidence is 0.4
 Conditional Pattern Base
 [['I1', 'I2'], ['I3', 'I1', 'I2']] => [1, 1]
 [['I1']] => [2]
 [['I2'], ['I1', 'I2']] => [1, 1]
 [['I2']] => [4]
 [['I2'], ['I1'], ['I1', 'I2']] => [2, 2, 2]
 [['I1']] => [2]

  FP Growth Tree is given below
    Null   1
      I2   7
        I1   4
          I5   1
          I4   1
          I3   2
            I5   1
        I4   1
        I3   2
      I1   2
        I3   2
```

```
def fpgrowthFromFile(minSup):
    itemSetList, frequency = getFromFile()
    fpTree, headerTable = constructTree(itemSetList, frequency, minSup)
    #fpTree, headerTable = fpgrowthFromFile(minSup)
    freqItems = []
    print("Conditional Pattern Base:")
    mineTree(headerTable, minSup, set(), freqItems)

fpgrowthFromFile(2)
```

```
 Conditional Pattern Base:
 [['I1', 'I2'], ['I3', 'I1', 'I2']] => [1, 1]
 [['I1']] => [2]
 [['I2'], ['I1', 'I2']] => [1, 1]
 [['I2']] => [4]
 [['I2'], ['I1'], ['I1', 'I2']] => [2, 2, 2]
 [['I1']] => [2]
```

```
print("\nFreq Patters is given below")
for item in freqItems:
  print(item)
```

```
    Freq Patters is given below
    {'I5'}
    {'I1', 'I5'}
    {'I5', 'I2'}
    {'I1', 'I5', 'I2'}
    {'I4'}
    {'I2', 'I4'}
    {'I1'}
    {'I1', 'I2'}
    {'I3'}
    {'I3', 'I2'}
    {'I3', 'I1', 'I2'}
    {'I3', 'I1'}
    {'I2'}
```

```
print("\nTentitive rules are given below")
for rule in tentitive_rules:
  print(rule)
```

```
    Tentitive rules are given below
    [{'I1'}, {'I5'}, 0.3333333333333333]
    [{'I5'}, {'I1'}, 1.0]
    [{'I5'}, {'I2'}, 1.0]
    [{'I2'}, {'I5'}, 0.2857142857142857]
    [{'I1'}, {'I5', 'I2'}, 0.3333333333333333]
    [{'I5'}, {'I1', 'I2'}, 1.0]
    [{'I2'}, {'I1', 'I5'}, 0.2857142857142857]
    [{'I1', 'I5'}, {'I2'}, 1.0]
    [{'I1', 'I2'}, {'I5'}, 0.5]
    [{'I5', 'I2'}, {'I1'}, 1.0]
    [{'I2'}, {'I4'}, 0.2857142857142857]
    [{'I4'}, {'I2'}, 1.0]
    [{'I1'}, {'I2'}, 0.6666666666666666]
    [{'I2'}, {'I1'}, 0.5714285714285714]
    [{'I3'}, {'I2'}, 0.6666666666666666]
    [{'I2'}, {'I3'}, 0.5714285714285714]
    [{'I3'}, {'I1', 'I2'}, 0.3333333333333333]
    [{'I1'}, {'I3', 'I2'}, 0.3333333333333333]
    [{'I2'}, {'I3', 'I1'}, 0.2857142857142857]
    [{'I3', 'I1'}, {'I2'}, 0.5]
    [{'I3', 'I2'}, {'I1'}, 0.5]
    [{'I1', 'I2'}, {'I3'}, 0.5]
    [{'I3'}, {'I1'}, 0.6666666666666666]
    [{'I1'}, {'I3'}, 0.6666666666666666]
```

```
print("\nAssociation Rules are given below")
for rule in association_rules:
  print(rule)
```

```
    Association Rules are given below
    [{'I5'}, {'I1'}, 100.0]
    [{'I5'}, {'I2'}, 100.0]
    [{'I5'}, {'I1', 'I2'}, 100.0]
    [{'I1', 'I5'}, {'I2'}, 100.0]
    [{'I1', 'I2'}, {'I5'}, 50.0]
```

```
[{'I5', 'I2'}, {'I1'}, 100.0]
[{'I4'}, {'I2'}, 100.0]
[{'I1'}, {'I2'}, 66.66666666666666]
[{'I2'}, {'I1'}, 57.14285714285714]
[{'I3'}, {'I2'}, 66.66666666666666]
[{'I2'}, {'I3'}, 57.14285714285714]
[{'I3', 'I1'}, {'I2'}, 50.0]
[{'I3', 'I2'}, {'I1'}, 50.0]
[{'I1', 'I2'}, {'I3'}, 50.0]
[{'I3'}, {'I1'}, 66.66666666666666]
[{'I1'}, {'I3'}, 66.66666666666666]
```

+ Code + Text

✓ 0s completed at 22:22