DATA MINING WITH FP-GROWTH ALGORITHM

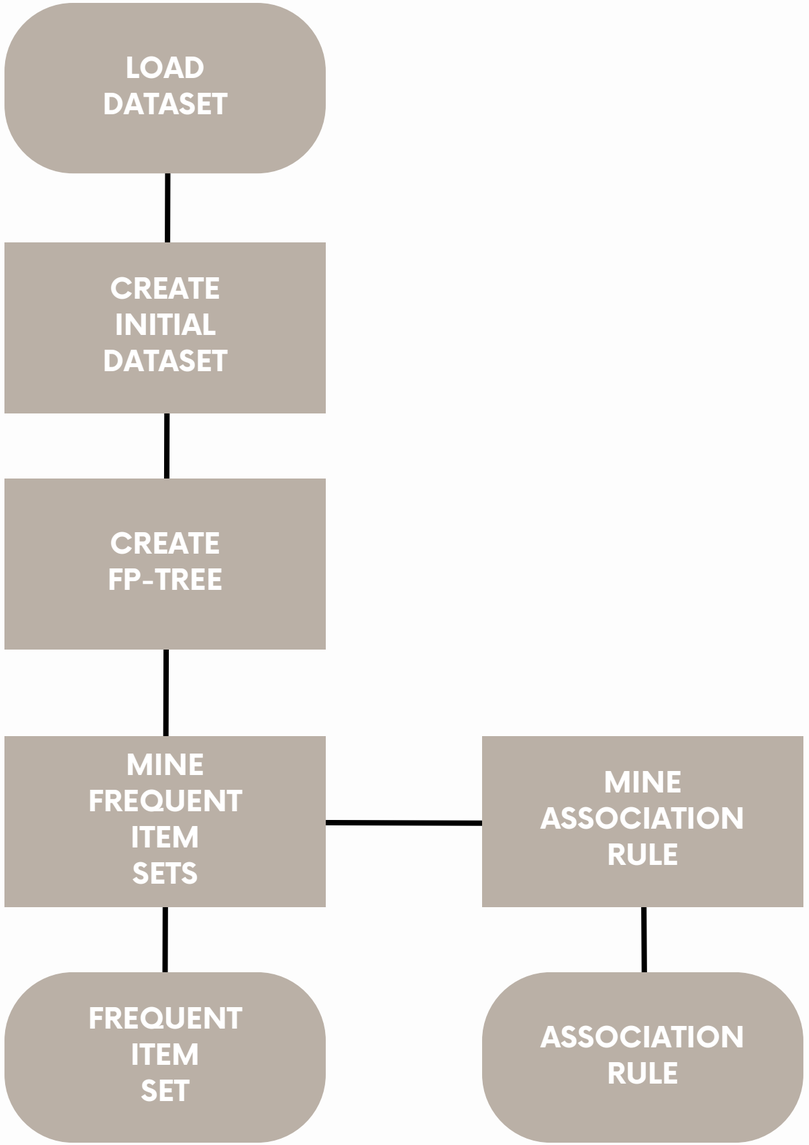
TABLE OF CONTENTS

INTRODUCTION & FLOWCHART:

FP-Growth 演算法是一種用於資料探勘中頻繁項集發現的有效方法。 它是由Jian Pei，Jiawei Han和Runying Mao在2000年的論文中首次提出的。 該演算法主要應用於事務資料分析、關聯規則挖掘以及資料探勘領域的其他相關應用。

FP-Growth 演算法的核心思想是使用一種叫做"FP樹（Frequent Pattern Tree）"的緊湊資料結構來儲存頻繁項集資訊。 這個資料結構能夠大大減少需要遍歷的搜尋空間，從而提高演算法的執行效率。

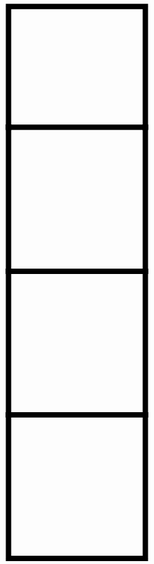
用Python時作此演算法能運用到許多內建的資料結構。



DATA STRUCTURE:

InitSet:
dtype:dict{key:frozenset({itemSet}),value:出現次數}

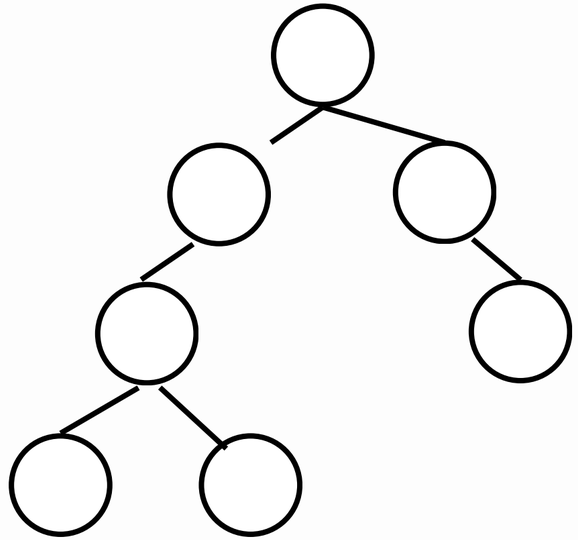
HeaderTable



dtype : dict{key:元素,value:[出現次數,第一個元素節點物件]}

self.name 節點元素名稱，在構造時初始化爲給定值
self.count 出現次數，在構造時初始化爲給定值
self.nodeLink 指向下一個相似節點的指針，默認爲None
self.parent 指向父節點的指針，在構造時初始化爲給定值
self.children 指向子節點的字典，以子節點的元素名稱爲鍵，指向子節點的指針爲值，初始化爲空字典

FP-Tree



前綴路徑prefixPath:
dtype:list

條件模式基condPats:
dtype:dict{key:前綴路徑,value:節點的計數值}

MODULE:

import time :
time.time():用以計算總體時間與分項時間

from itertools import chain, combinations:
chain.from_iterable():接受一個iterable物件作為參數，返回由iterable物件所有元素的扁平化iterable物件
combinations():尋找所有可能的組合

STEP:

```
dataSet = loadData("mushroom.dat")
```

```
freqItemDict = fpGrowth(dataSet, minSup
```

```
def fpGrowth(dataSet, minSup):
    initSet = createInitSet(dataSet)
    fpTree, headerTable = createTree(initSet, minSup)
    freqItemDict = {}
    mineTree(fpTree, headerTable, minSup, set([]), freqItemDict)
    return freqItemDict
```

```
rules = associationRule(freqItemDict, minConf)
```

GENERATE DATASET:

```python
def loadData(filePath):
    with open(filePath, "r", encoding="utf-8") as f:
        lines = f.readlines()
        dataSet = [list(map(int, line.split())) for line in lines]
    return dataSet
```

(載入檔案)

```python
def createInitSet(dataSet):
    dataDict = {}
    for itemSet in dataSet:
        if frozenset(itemSet) not in dataDict:
            dataDict[frozenset(itemSet)] = 1
        else:
            dataDict[frozenset(itemSet)] += 1
    return dataDict
```

(生成初始資料集)

CREATE FP-TREE STRUCTURE:

```
class treeNode:
    def __init__(self, nameValue, numOccur, parentNode):
        self.name = nameValue
        self.count = numOccur
        self.nodeLink = None
        self.parent = parentNode
        self.children = {}

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

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

CREATE FP-TREE:

```python
def createTree(dataDict, minSup):
    headerTable = {}
    for itemSet in dataDict:
        for item in itemSet :
            headerTable[item] = headerTable.get(item, 0) + dataDict[itemSet]
    keysToRemove = [item for item in headerTable if headerTable[item] < minSup]
    for item in keysToRemove:
        del headerTable[item]
    freqItemSet = set(headerTable.keys())
    if len(freqItemSet) == 0:
        return None, None
    for item in headerTable:
        headerTable[item] = [headerTable[item], None]
    fpTree = treeNode("Null", 1, None)
    for itemSet, count in dataDict.items():
        localD = {}
        for item in itemSet:
            if item in freqItemSet:
                localD[item] = headerTable[item][0]
        if len(localD) > 0:
            orderedItems = [v[0] for v in sorted(localD.items(), key=lambda p:(p[1],int(p[0])), reverse=True)]
            updateTree(orderedItems, fpTree, headerTable, count)
    return fpTree, headerTable
```

```python
def updateTree(items, inTree, headerTable, count):
    if items[0] in inTree.children:
        inTree.children[items[0]].increment(count)
    else:
        inTree.children[items[0]] = treeNode(items[0], count, inTree)
        if headerTable[items[0]][1] == None:
            headerTable[items[0]][1] = inTree.children[items[0]]
        else:
            updateHeader(headerTable[items[0]][1], inTree.children[items[0]])

    if len(items) > 1:
        updateTree(items[1::], inTree.children[items[0]], headerTable, count)
```

```python
def updateHeader(nodeToTest, targetNode):
    while (nodeToTest.nodeLink != None):
        nodeToTest = nodeToTest.nodeLink
    nodeToTest.nodeLink = targetNode
```

(更新HeaderTable)

(遞迴)

RECURSIVE SEARCH FOR FREQUENT ITEM SETS:

```python
def mineTree(inTree, headerTable, minSup, preFix, freqItemDict):
    bigL = [v[0] for v in sorted(headerTable.items(), key=lambda p: str(p[1]))]
    for basePat in bigL:
        newFreqSet = preFix.copy()
        newFreqSet.add(basePat)
        if len(newFreqSet) > 10:
            continue
        if frozenset(newFreqSet) not in freqItemDict:
            freqItemDict[frozenset(newFreqSet)] =  headerTable[basePat][0]
        else:
            freqItemDict[frozenset(newFreqSet)] += headerTable[basePat][0]
        condPattBases = findPrefixPath(basePat, headerTable[basePat][1])
        condTree, headTab= createTree(condPattBases, minSup)
        if headTab != None:
            mineTree(condTree, headTab, minSup, newFreqSet, freqItemDict)
```

```python
def ascendTree(leafNode, prefixPath):
    if leafNode.parent != None:
        prefixPath.append(leafNode.name)
        ascendTree(leafNode.parent, prefixPath)
```

```python
def findPrefixPath(basePat, treeNode):
    condPats = {}
    while treeNode != None:
        prefixPath = []
        ascendTree(treeNode, prefixPath)
        if len(prefixPath) > 1:
            condPats[frozenset(prefixPath[1:])] = treeNode.count
        treeNode = treeNode.nodeLink
    return condPats
```

(生成條件模式基condPats)

ASSOCIATION RULE MINING:

```python
def associationRule(freqItemDict, minConf):
    rules = 0
    for freqItem, freqItemSup in freqItemDict.items():
        subsets = powerset(freqItem)
        for s in subsets:
            s = frozenset(s)
            if len(s) > 0 and s != frozenset(freqItem):
                conf = freqItemSup / freqItemDict.get(s, 1)
                if conf >= minConf:
                    rules += 1
    return rules
```

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

(生成一個集合中的所有非空子集)

RESULT:

```
Frequent Item Sets :
|L^1|=56
|L^2|=763
|L^3|=4593
|L^4|=16150
|L^5|=38800
|L^6|=69835
|L^7|=98846
|L^8|=111786
|L^9|=100660
|L^10|=71342
Number of association rules that meet the conditions : 70382966
Total Execution Time : 62.89203715324402 seconds.
i. frequent item set mining : 3.3795313835144043 seconds.
ii. association rule mining : 59.512505769729614 seconds.
```

HARDWARE SPECIFICATIONS:

CPU : 12th Gen Intel(R) Core(TM) i5-12400F 2.50 GHz
RAM : 16GB