Homework 1 - Frequent Pattern Analysis

Name: < insert name here >

Remember that you are encouraged to discuss the problems with your instructors and classmates, but **you must write all code and solutions on your own**.

The rules to be followed for the assignment are:

- Do **NOT** load additional packages beyond what we've shared in the cells below.
- Some problems with code may be autograded. If we provide a function or class API **do not** change it.
- Do not change the location of the data or data directory. Use only relative paths to access the data.

In [1]:
```
import argparse
import pandas as pd
import numpy as np
import random
import pickle
from pathlib import Path
from collections import defaultdict
```

[10 points] Problem 1 - Apriori Implementation

A sample dataset has been provided to you in the './data/dataset.pickle' path. Here are the attributes for the dataset. Use this dataset to test your functions.

- Dataset should load the transactions in the form of a python dictionary where each key holds the transaction id and the value is a python list of the items purchased in that transaction.
- An example transaction will have the following structure. If items A, C, D, F are purchased in transaction T3, this would appear as follows in the dictionary.

```
transactions = {
   "T3": ["A", "C", "D", "F"]
}
```

Note:

- A sample dataset to test your code has been provided in the location "./data/dataset.pickle". Please maintain this as it would be necessary while grading.
- ge the variable names of the returned values.

- After calculating each of those values, assign them to the corresponding value that is being returned.
- If you are encountering any errors while loading the dataset, the following lines of code should help. Please delete the cells before submitting, to reduce any potential autograder issues.

```
!pip install pickle5

import pickle5 as pickle
```

In [2]:
```
import itertools

def findsubsets(s, n):

#   A helper function that you can use to list of all subsets of size n. D
o not make any changes to this code block.
#   Input:
#       1. s - A python list of items
#       2. n - Size of each subset
#   Output:
#       1. subsets - A python list containing the subsets of size n.

    subsets = list(sorted((itertools.combinations(s,n))))
    return subsets
```

In [3]:
```
def items_from_frequent_itemsets(frequent_itemset):

#   A helper function that you can use to get the sorted items from the fr
equent itemsets. Do not make any changes
#   to this code block
#   Input:
#       1. Frequent Itemsets
#   Output:
#       1. Sorted list of items

    items = list()
    for keys in frequent_itemset.keys():
        for item in list(keys):
            items.append(item)
    return sorted(list(set(items)))
```

In [4]:
```
def generate_frequent_itemsets(dataset, support, items, n=1, frequent_item
s={}):

#   Input:
#       1. dataset - A python dictionary containing the transactions.
#       2. support - A floating point variable representing the min_suppor
t value for the set of transactions.
#       3. items - A python list representing all the items that are part
of all the transactions.
#       4. n - An integer variable representing what frequent item pairs t
o generate.
#       5. frequent_items - A dictionary representing k-1 frequent sets.
#   Output:
#       1. frequent_itemsets - A dictionary representing the frequent item
```

```
sets and their corresponding support counts.


    #dataset is a dict with {transaction: [items]}.
    # your code here:

    if n == 1:
        # your code here:
        items_counts = dict()
        #print("items_counts:", items_counts)
        for i in items:
            temp_i = {i}
            #print("temp_i:", temp_i)
            for j in dataset.items():
                if temp_i.issubset(set(j[1])):
                    #print("temp_i.issubset: + j", temp_i, j)
                    if i in items_counts:
                        items_counts[i] += 1
                        #print("items_counts after iteration:", items_coun
ts)
                    else:
                        items_counts[i] = 1
            for (key, value) in items_counts.items():
                if value / len(dataset) >= support:
                    frequent_items[key] = value
        return frequent_items


    else:
        frequent_items2 = {}
        # your code here
        temp_comb = {}
        l1 = list(sorted(itertools.combinations(items, n))) #all possible
combinations of n from items

        #1st step: go through each transactions and check if
        for i in l1:
            count = 0
            for key, value in dataset.items():
                #temp_comb[i] = {}
                if(all(x in value for x in i)):

                    count += 1
                temp_comb[i] = count
            for key, value in temp_comb.items():
                if value / len(dataset) >= support:
                    frequent_items2[key] = value

        return frequent_items2
```

In [5]:
```
# This cell has hidden test cases that will run after you submit your assi
gnment.
```

In [6]:
```
import unittest

class TestX(unittest.TestCase):
```

```
    def setUp(self):
        self.min_support = 0.5
        self.items = ['A', 'B', 'C', 'D', 'E']
        self.dataset = dict()
        self.dataset["T1"] = ['A', 'B', 'D']
        self.dataset["T2"] = ['A', 'B', 'E']
        self.dataset["T3"] = ['B', 'C', 'D']
        self.dataset["T4"] = ['B', 'D', 'E']
        self.dataset["T5"] = ['A', 'B', 'C', 'D']

    def test0(self):
        frequent_1_itemsets = generate_frequent_itemsets(self.dataset, sel
f.min_support, self.items)
        print (frequent_1_itemsets)
        frequent_1_itemsets_solution = dict()
        frequent_1_itemsets_solution['A'] = 3
        frequent_1_itemsets_solution['B'] = 5
        frequent_1_itemsets_solution['D'] = 4

        print ("Test 1: frequent 1 itemsets")
        assert frequent_1_itemsets == frequent_1_itemsets_solution

        frequent_2_itemsets = generate_frequent_itemsets(self.dataset, sel
f.min_support, self.items, 2, frequent_1_itemsets)
        print (frequent_2_itemsets)
        frequent_2_itemsets_solution = dict()
        frequent_2_itemsets_solution[('A', 'B')] = 3
        frequent_2_itemsets_solution[('B', 'D')] = 4

        print ("Test 1: frequent 2 itemsets")
        assert frequent_2_itemsets == frequent_2_itemsets_solution

        frequent_3_itemsets = generate_frequent_itemsets(self.dataset, sel
f.min_support, self.items, 3, frequent_2_itemsets)
        print (frequent_3_itemsets)
        frequent_3_itemsets_solution = dict()

        print ("Test 1: frequent 3 itemsets")
        assert frequent_3_itemsets == frequent_3_itemsets_solution


tests = TestX()
tests_to_run = unittest.TestLoader().loadTestsFromModule(tests)
unittest.TextTestRunner().run(tests_to_run)
```

```
.
```

```
{'A': 3, 'B': 5, 'D': 4}
Test 1: frequent 1 itemsets
{('A', 'B'): 3, ('B', 'D'): 4}
Test 1: frequent 2 itemsets
{}
Test 1: frequent 3 itemsets
```

```
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Out[6]: <unittest.runner.TextTestResult run=1 errors=0 failures=0>

[10 points] Problem 2 - FP-Growth Implementation

A sample dataset has been provided to you in the './data/dataset.pickle' path. Here are the attributes for the dataset. Use this dataset to test your functions.

- Dataset should load the transactions in the form of a python dictionary where each key holds the transaction id and the value is a python list of the items purchased in that transaction.
- An example transaction will have the following structure. If items A, C, D, F are purchased in transaction T3, this would appear as follows in the dictionary.

```
transactions = {
   "T3": ["A", "C", "D", "F"]
}
```

Note:

- A sample dataset to test your code has been provided in the location "./data/dataset.pickle". Please maintain this as it would be necessary while grading.
- Do not change the variable names of the returned values.
- After calculating each of those values, assign them to the corresponding value that is being returned.

In [7]:
```
def item_support(dataset, min_support):

#   A helper function that returns the support count of each item in the d
ataset. The dictionary is further sorted
#   based on maximum support of each item and pruned based on min_support.
#   Input:
#       1. dataset - A python dictionary containing the transactions.
#       2. items - A python list representing all the items that are part
of all the transactions.
#       3. min_support - A floating point variable representing the min_su
pport value for the set of transactions.
#   Output:
#       1. support_dict - A dictionary representing the support count of e
ach item in the dataset.

    len_transactions = len(dataset)
    support_dict = dict()
    items_counts = dict()
    items = []
    for key, value in dataset.items():
        items.append(value)

    items = [item for sublist in items for item in sublist]
    items = (set(items))

    for i,j in enumerate(items):
        for k in j:
```

```
            temp_k = {k}
            for l in dataset.items():
                if temp_k.issubset(set(l[1])):
                    if k in items_counts:
                        items_counts[k] += 1
                    else:
                        items_counts[k] = 1
            print(" count is :", items_counts)

        # your code here

    support_dict = items_counts



    sorted_support = dict(sorted(support_dict.items(), key=lambda item: it
em[1], reverse=True))
    pruned_support = {key:val for key, val in sorted_support.items() if va
l/len_transactions >= min_support}

    return support_dict
```

In [8]:
```
# This cell has hidden test cases that will run after you submit your assi
gnment.
```

In [9]:
```
def reorder_transactions(dataset, min_support):

#   A helper function that reorders the transaction items based on maximum
 support count. It is important that you finish
#   the code in the previous cells since this function makes use of the su
pport count dictionary calculated above.
#   Input:
#       1. dataset - A python dictionary containing the transactions.
#       2. items - A python list representing all the items that are part
of all the transactions.
#       3. min_support - A floating point variable representing the min_su
pport value for the set of transactions.
#   Output:
#       1. updated_dataset - A dictionary representing the transaction ite
ms in sorted order of their support counts.

    #pruned_support = item_support(dataset, min_support)
    updated_dataset = dict()
    len_transactions = len(dataset)

    support_dict = item_support(dataset, min_support)
    sorted_support = dict(sorted(support_dict.items(), key=lambda item: it
em[1], reverse=True))
    pruned_support = {key:val for key, val in sorted_support.items() if va
l/len_transactions >= min_support}


    # dataset1 will have all items.
    # 1. Scan dataset1 and pruned_supp.
    # 2. Go through each transaction and keep only the items in both prune
d and dataset1.
```

```
    # Attempt 1. a) Store pruned_supp keys in temp_list.
    #            b) Check membership for each item.
    #            c) If yes, keep and move to front. If no, remove.
    #new_dataset = {}
    for trans, items in dataset.items():
        temp = []
        for item in pruned_support.keys():
            if set(item).issubset(items):
                temp.append(item)
            else:
                pass
        dataset[trans] = temp
    updated_dataset = dataset




    return updated_dataset
```

In [ ]:

In [10]:
```
from collections import defaultdict
from itertools import chain, combinations

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)

def build_fp_tree(updated_dataset):

#   Input:
#       1. updated_dataset - A python dictionary containing the updated se
t of transactions based on the pruned support dictionary.
#   Output:
#       1. fp_tree - A dictionary representing the fp_tree. Each node shou
ld have a count and children attribute.
#
#   HINT:
#       1. Loop over each transaction in the dataset and make an update to
 the fp_tree dictionary.
#       2. For each loop iteration store a pointer to the previously visit
ed node and update it's children in the next pass.
#       3. Update the root pointer when you start processing items in each
 transaction.
```

```
#       4. Reset the root pointer for each transaction.
#
#   Sample Tree Output:
#   {'Y': {'count': 3, 'children': {'V': {'count': 1, 'children': {}}}},
#    'X': {'count': 2, 'children': {'R': {'count': 1, 'children': {'F': {'
count': 1, 'children': {}}}}}}}
    #frequency = generate_frequent_itemsets(updated_dataset, 0.5)

    #fp_tree = dict({ 'A': {'count': 8, 'children': {'B': {'count':6, 'chi
ldren': { 'C': {'count':2, 'children': {}},
    #                                                                  '
D': {'count': 4, 'children': {
    #
   'C': {'count':1, 'children': {}}}}}},
    #                                        'D': {'count':1, 'children':
{'C': {'count': 1, 'children': {}}}}},
    #           'B': {'count':1, 'children': {'D':{'count':1, 'children':
{'C':{'count':1, 'children':{}}}}}}})
    #data = []
    #for trans, items in updated_dataset.items():
    #    data.append(items)
    data = updated_dataset.values()
    print(data)
    def fpgrowth(itemSetList, minSupRatio, minConf):
        frequency = getFrequencyFromList(itemSetList)
        minSup = len(itemSetList) * minSupRatio
        fpTree, headerTable = constructTree(itemSetList, frequency, minSup
)
        if(fpTree == None):
            print('No frequent item set')
        else:
            freqItems = []
            mineTree(headerTable, minSup, set(), freqItems)
            rules = associationRule(freqItems, itemSetList, minConf)
            return freqItems, rules
    def getFromFile(data):
        itemSetList = []
        frequency = []

        for line in data:
            line = list(filter(None, line))
            itemSetList.append(line)
            frequency.append(1)
        return itemSetList, frequency
    itemSetList, frequency = getFromFile(data)
    print(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)
        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=Tr
ue)
            # Traverse from root to leaf, update tree with given item
            currentNode = fpTree
            for item in itemSet:
                currentNode = updateTree(item, currentNode, headerTable, f
requency[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 getFrequencyFromList(itemSetList):
        frequency = [1 for i in range(len(itemSetList))]
        return frequency
    def mineTree(headerTable, minSup, preFix, freqItemList):
    # Sort the items with frequency and create a list
        sortedItemList = [item[0] for item in sorted(list(headerTable.item
s()), key=lambda p:p[1][0])]
    # Start with the lowest frequency
        for item in sortedItemList:
            # Pattern growth is achieved by the concatenation of suffix pa
ttern with frequent patterns generated from conditional FP-tree
            newFreqSet = preFix.copy()
            newFreqSet.add(item)
```

```
            freqItemList.append(newFreqSet)
            # Find all prefix path, constrcut conditional pattern base
            conditionalPattBase, frequency = findPrefixPath(item, headerTa
ble)
            # Construct conditonal FP Tree with conditional pattern base
            conditionalTree, newHeaderTable = constructTree(conditionalPat
tBase, frequency, minSup)
            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, le
n(s)))

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

    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)
)
                if(confidence > minConf):
                    rules.append([set(s), set(itemSet.difference(s)), conf
idence])
        return rules
```

```
    frequentItemSet, rules = fpgrowth(data, minSupRatio = 0.5, minConf = 0
.5)
    fp_tree = constructTree(data, frequency, 0.5)
    return fp_tree
```

In [ ]: