(a) Implement Apriori and FP-growth algorithm. Cite any sources helpful to you for implementing the algorithms.

Implementing Apriori Algorithm

In [ ]:
```
# importing libraries
import pandas as pd
import numpy as np

from itertools import combinations
```

In [ ]:
```
# creating transaction array
transactions = []

# reading data from retail.dat file line-wise and storing each line as transaction in transaction array
with open('/Users/quasar/Downloads/Courses/Data Mining/retail.dat') as f:
    for line in f.readlines():
        transaction = line.strip().split(' ')
        transaction = np.asarray(transaction, dtype='int64')
        transactions.append(transaction)
```

In [ ]:
```
# choosing subset of transactions array to reduce compute time
transactions[:10]
```

Out[ ]:
```
[array([  9,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]),
 array([30, 31, 32]),
 array([33, 34, 35]),
 array([36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46]),
 array([38, 39, 47, 48]),
 array([38, 39, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58]),
 array([32, 41, 59, 60, 61, 62]),
 array([  3, 39, 48]),
 array([43, 44, 45, 46, 47, 48]),
 array([32, 63])]
```

In [ ]:
```
# create the 1-item candidate,
# Function creates a frozenset for each unique item and stores them in a list

def create_candidate_1(X):
        c1 = []
        for transaction in X:
                for t in transaction:
                        t = frozenset([t])
                        if t not in c1:
                                c1.append(t)
        return c1
```

In [ ]:
```
# Transaction data is input to the function and the minimum support threshold to obtain the frequent itemset.
# Functions store the support for each itemset, which will be used in the rule generation step

# the candidate sets for the 1-item is different
# create them independently from others

def apriori(X, min_support):
        c1 = create_candidate_1(X)
        freq_item, item_support_dict = create_freq_item(X, c1, min_support)
        freq_items = [freq_item]

        k=2
        while len(freq_items[k]) > 0:
                freq_item = freq_items[k]
                ck = create_candidate_k(freq_item, k)
                freq_item, item_support = create_freq_item(X, ck, min_support)
                freq_items.append(freq_item)
                item_support_dict.update(item_support)
                k += 1

        return freq_items, item_support_dict
```

In [ ]:
```
# filters the candidate with the specified minimum support
# loop through the transaction and compute the count for each candidate (item)
# if the support of an item is greater than the min_support, then it is considered as frequent

def create_freq_item(X, ck, min_support = 0.2):
        item_count = {}
        for transaction in X:
                for item in ck:
                        if item.issubset(transaction):
                                if item not in item_count:
                                        item_count[item] = 1
                                else:
                                        item_count[item] += 1

        n_row = len(X)
        freq_item = []
        item_support = {}

        for item in item_count:
                support = item_count[item] / n_row
                if support >= min_support:
                        freq_item.append(item)

                item_support[item] = support

        return freq_item, item_support
```

In [ ]:
```
# create the list of k-item candidate

def create_candidate_k(freq_item, k):
        ck = []
        # for generating candidate of size two (2-itemset)
        if k == 2:
                for f1, f2 in combinations(freq_item, 2):
                        item = f1 | f2 # union of two sets
                        ck.append(item)

        else:
                for f1, f2 in combinations(freq_item, 2):
                        # if the two (k-1)-item sets has k common elements then they will be unioned to be
                        # the (k+1)- item candidate
                        intersection = f1 & f2
                        if len(intersection) == k:
                                item = f1 | f2
                                if item not in ck:
                                        ck.append(item)


        return ck
```

In [ ]:
```
freq_items, item_support_dict =  apriori(transactions[:500], min_support = 0.2)
freq_items
```

Out[ ]:
```
[[frozenset({39}), frozenset({29}), frozenset({41}), frozenset({48})],
 [frozenset({39, 48})],
 []]
```

In [ ]:
```
# create the association rules, the rules will be a list.
# each element is a tuple of size 4, containing rules' left hand side, right hand side, confidence

def create_rules(freq_items, item_support_dict, min_confidence):
        association_rules = []

        # for the list that stores the frequent items, loop through
        # the second element to the one before the last to generate the rules
        # because the last one will be an empty list. It's the stopping criteria
        # for the frequent itemset generating process and the first one are all
        # single element frequent itemset, which can't perform the set
        # operation X ==> Y -> Z

        for idx, freq_item in enumerate(freq_items[1:(len(freq_items) - 1)]):
                for freq_set in freq_item:

                        # start with creating rules for single item on
                        # the right hand side
                        subsets = [frozenset({item}) for item in freq_set]
                        rules, right_hand_side = compute_conf(freq_items, item_support_dict,
                                                              freq_set, subsets, min_confidence)
                        association_rules.extend(rules)

                        # starting from 3-itemset, loop through each length item
                        # to create the rules, as for the while loop condition,
                        # e.g. suppose you start with a 3-itemset {2, 3, 5} then the
                        # while loop condition will stop when the right hand side's
                        # item is of length 2, e.g. {{2, 3}, {3, 5} }, since this
                        # will be merged into 3 itemset, making the left hand side
                        # null when computing the confidence
                        if idx != 0:
                                k = 2
                                while len(right_hand_side[0]) < len(freq_set) - 1:
                                        ck = create_candidate_k(right_hand_side, k + 1)
                                        rules, right_hand_side = compute_conf(freq_items, item_support_dict,
                                                                              freq_set, ck, min_confidence)
                                        association_rules.extend(rules)
                                        k += 1

        return association_rules
```

In [ ]:
```
#create the rules and returns the rules info and the rules's right hand side
# (used for generating the next round of rules)
# if it surpasses the minimum confidence threshold

def compute_conf(freq_items, item_support_dict, freq_set, subsets, min_confidence):
        rules = []
        right_hand_side = []

        for rhs in subsets:
                # create the left hand side of the rule
                # and add the rules if it's greater than
                # the confidence threshold
                lhs = freq_set - rhs
                conf = item_support_dict[freq_set] / item_support_dict[lhs]
                if conf >= min_confidence:
                        lift = conf / item_support_dict[rhs]
                        rules_info = lhs, rhs, conf, lift
                        rules.append(rules_info)
                        right_hand_side.append(rhs)

        return rules, right_hand_side
```

In [ ]:
```
association_rules = create_rules(freq_items, item_support_dict, min_confidence = 0.5)
association_rules
```

Out[ ]:
```
[(frozenset({39}), frozenset({48}), 0.5747508030564784, 1.2549144732156853),
 (frozenset({48}), frozenset({39}), 0.7554585152838427, 1.2549144732156855)]
```

Implementing FP algorithm

In [ ]:
```
# class of FP tree node
class Tree_Node:
        def __init__(self, node_name, counter, parent_node):
                self.name = node_name
                self.count = counter
                self.node_link = None
                self.parent = parent_node
                self.children = []

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

In [ ]:
```
'''
def create_FP_tree(dataset, min_support):
        header_table = {}
        for transaction in dataset:
                for item in transaction:
                        header_table[item] = header_table.get(item, 0) + dataset[transaction]
        for k in list(header_table):
                if header_table[k] < len(dataset) * min_support:
                        del(header_table[k])

        frequent_itemset = set(header_table.keys())

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

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

        retTree = Tree_Node('Null Set', 1, None)

        for itemset, count in dataset.items():
                frequent_transaction = ()
                #print(dataset.items())
                #print(itemset, count)
                for item in itemset:
                        if item in frequent_itemset:
                                frequent_transaction[item] = header_table[item][0]
                                #print(frequent_transaction)



create_FP_tree(initSet, min_support=0.1)
'''';
```

In [ ]:
```
# To create header_table and ordered itemsets for FP Tree
def create_FP_tree(dataset, min_support):
        header_table = {}
        for transaction in dataset:
                for item in transaction:
                        header_table[item] = header_table.get(item, 0) + dataset[transaction]
        for k in list(header_table):
                if header_table[k] < len(dataset) * min_support:
                        del(header_table[k])

        frequent_itemset = set(header_table.keys())

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

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

        retTree = Tree_Node('Null Set', 1, None)

        for itemset, count in dataset.items():
                frequent_transaction = ()
                for item in itemset:
                        if item in frequent_itemset:
                                frequent_transaction[item] = header_table[item][0]
                if len(frequent_transaction) > 0:
                        # sorted frequent transaction from transactions
                        ordered_itemset = [v[0] for v in sorted(frequent_transaction.items(), key=lambda p: p[1], reverse=True)]
                        # to update the FP Tree
                        update_tree(ordered_itemset, retTree, header_table, count)
        return retTree, header_table
```

In [ ]:
```
# to update the FP Tree using ordered itemsets
def update_tree(itemset, FP_tree, header_table, count):
        if itemset[0] in FP_tree.children:
                FP_tree.children[itemset[0]].increment_counter(count)
        else:
                FP_tree.children[itemset[0]] = Tree_Node(itemset[0], count, FP_tree)

                if header_table[itemset[0]][1] == None:
                        header_table[itemset[0]][1] = FP_tree.children[itemset[0]]
                else:
                        update_Node_link(header_table[itemset[0]][1], FP_tree.children[itemset[0]])

        if len(itemset) > 1:
                update_tree(itemset[1:], FP_tree.children[itemset[0]], header_table, count)
```

In [ ]:
```
#To update the link of node in FP tree
def update_Node_link(test_node, target_node):
        while(test_node.node_link != None):
                test_node = test_node.node_link

        test_node.node_link = target_node
```

In [ ]:
```
# to transverse FP_tree in upward direction
def FP_tree_uptraversal(leaf_node, prefix_Path):
        if leaf_node.parent != None:
                prefix_Path.append(leaf_node.name)
                FP_tree_uptraversal(leaf_node.parent, prefix_Path)
```

In [ ]:
```
# to find conditional Pattern base
def find_prefix_path(base_Pattern, Tree_Node):
        conditional_patterns_base = {}

        while Tree_Node != None:
                prefix_Path = []
                FP_tree_uptraversal(Tree_Node, prefix_Path)
                if len(prefix_Path) > 1:
                        conditional_patterns_base[frozenset(prefix_Path[1:])] = Tree_Node.count
                Tree_Node = Tree_Node.node_link

        return conditional_patterns_base
```

In [ ]:
```
#function to mine recursively conditional patterns base and conditional FP tree
def Mine_tree(FPTree, header_table, min_support, prefix, frequent_itemsets):
        big1 = [v[0] for v in sorted(header_table.items(), key=lambda p: p[1][0])]
        for base_Pattern in big1:
                new_frequentset = prefix.copy()
                new_frequentset.add(base_Pattern)
                #add frequent itemset to final list of frequent itemsets
                frequent_itemset.append(new_frequentset)
                #get all conditional patterb bases for item or itemsets
                Conditional_patterns_bases = find_prefix_path(base_Pattern, header_table[base_Pattern][1])
                #call FP_tree construction to make conditional FP Tree
                Conditional_FP_tree, Conditional_header = create_FP_tree(Conditional_patterns_bases,min_support)

                if Conditional_header != None:
                        Mine_tree(Conditional_FP_tree, Conditional_header, min_support, new_frequentset, frequent_itemset)
```

In [ ]:
```
#function to load file and return lists of Transactions
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
```

In [ ]:
```
#To convert initial transaction into frozenset
def create_initialset(dataset):
        retDict = {}
        for trans in dataset:
                retDict[frozenset(trans)] = 1
        return retDict
```

In [ ]:
```
min_support = 0.2

filename = '/Users/quasar/Downloads/Courses/Data Mining/retail.dat'
initSet = create_initialset(load_data(filename)[:10])
FP_tree, header_table = create_FP_tree(initSet, min_support)
print(FP_tree, header_table)
```

```
<__main__.Tree_Node object at 0x10e89aad0> {'3': [2, <__main__.Tree_Node object at 0x1250d1d50>], '32': [3, <__main__.Tree_Node object at 0x125bd1b50>], '39': [4, <__main__.Tree_Node object at 0x125b05760>], '38': [3, <__main__.Tree_Node object at 0x10f73ed60>], '41': [2, <__main__.Tree_Node object at 0x10f73ee70>], '48': [3, <__main__.Tree_Node object at 0x10f73ed60>]}
```

In [ ]:
```
frequent_itemset = []

#call function to mine all frequent itemsets
Mine_tree(FP_tree, header_table, min_support, set([]), frequent_itemset)
print(frequent_itemset)
```

```
[{'3'}, {'48', '3'}, {'3', '39'}, {'48', '3', '39'}, {'41'}, {'38', '41'}, {'41', '39'}, {'38', '41', '39'}, {'32', '41'}, {'32'}, {'38'}, {'38', '48'}, {'38', '48', '39'}, {'38', '39'}, {'48'}, {'48', '39'}, {'39'}]
```

(b) Modify the algorithms to achieve the same task (preferably with some improvement) . Clearly mention the difference in the modified algorithm.

Implementation Apriori using Hashtree