Frequent Pattern Mining: Comparative Analysis

PROJECT REPORT

Mini Project of Data Mining (DSCC 440-2)

MS Data Science

By
Aradhya Mathur

(Implementation project) Using a programming language that you are familiar with, such as C++ or Java, implement three frequent itemset mining algorithms introduced in this chapter: (1) Apriori [AS94b], (2) FP-growth [HPY00], and (3) Eclat [Zak00] (mining using the vertical data format). Compare the performance of each algorithm with various kinds of large data sets. Write a report to analyze the situations (e.g., data size, data distribution, minimal support threshold setting, and pattern density) where one algorithm may perform better than the others, and state why.

Implementation project #1 (not HW#4)

6.7 (1) & (2)*, plus one improvement of your choice for 6.7(1)
Due 10/06

Use the UCI Adult Census Dataset to test your code

http://archive.ics.uci.edu/ml/datasets/Adult

Important note: you can use the open-source code as a reference, but you should implement the algorithms independently. The point of the assignment is for you to know how the algorithms are implemented, not just how to run them. It would be easy to detect the latter, e.g. if more than one of you uses the same code.

UNIVERSITY of ROCHESTER

Fall Semester 2022

INDEX

ABSTRACT

Frequent pattern mining searches for recurrent relationships in a certain data set. In frequent mining typically the interesting associations and correlations between itemsets in transactional and relational databases are determined.

Support: Support of 10% means that 10% of all the transactions under analysis show that mobile and mobile cover are purchased together.

Confidence: A confidence of 60% means that 60% of the customers who purchased a mobile and mobile cover bought screen guard.

Support(A -> B) = Sup_count(A ∪ B)

Confidence(A -> B) = Sup_count(A ∪ B) / Sup_count(A)

A strong rule always satisfies both minimum support and minimum confidence.

Sup_count(A): Total transactions in which A appears.

Closed Itemset: An itemset in which none of its direct supersets have support count same as itemset is called closed itemset.

K- Itemset: The itemset in which there are K items.

To conclude it can be said that an itemset is frequent if its support count is greater than minimum support count set by users or domain experts.

PROBLEM STATEMENT:

(Implementation project) Using a programming language that you are familiar with, such as Python, implement (1) Apriori, (2) FP-growth, and (3) Improved Apriori. Compare the performance of each algorithm with UCI Adult dataset. Analyse why one algorithm may perform better than the others.

OBJECTIVES

Implementing Apriori, Improved Apriori and FP Growth algorithms. Comparing performance of each other based on time taken to implement algorithms at different minimum support.

DATASET DESCRIPTION

Adult dataset is census income dataset which based on census data predicts whether income exceeds $50K per year.

In this dataset there are 32561 rows and 15 columns.

Attributes in the dataset are:

1) Age: continuous values.
2) Workclass: Self-emp-not-inc, Federal-gov, Without-pay, etc.
3) fnlwgt: continuous values.
4) Education: Bachelors, HS-grad, Assoc-acdm, etc.
5) Education-num: continuous values.
6) Marital-status: Married-civ-spouse, Separated, Married-AF-spouse, etc.
7) Occupation: Exec-managerial, Machine-op-inspct, Armed-Forces, etc.
8) Relationship: Husband, Unmarried, etc.
9) Race: White, Amer-Indian-Eskimo, Black, etc.
10) Sex: Female or Male.
11) Capital-gain: continuous values.
12) Capital-loss: continuous values.
13) Hours-per-week: continuous values.
14) Native-country: United-States, India, China, France, Holand-Netherlands etc.
15) Income: >50K or <=50K.

In this data we can determine the pattern of which kind of people lie in income range>50K and <=50K. For example, after mining we obtain a frequent pattern {Female, United-States, Full-Time, >50K} pattern, we can say that a person who is a Female, lives in United-States and works Full-Time earns >50K.

There are some continuous attributes and rest are categorical. Our primary job is to convert continuous attributes into categorical.

PRE-PROCESSING:

Step 1) Reading data and adding column names

df = pd.read_csv('adult.data', sep=",", header = None , na_values = "?")

In this we observed there was no header.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K

Fig 1: Raw Data

In order to add headers (column names), the following process was done:

df.columns = ['age', 'workclass','fnlwgt','education_num', 'education', 'marital_status', 'occupation', 'relationship', 'race', 'sex', 'capital_gain', 'capital_loss','hours_per_week','native_country', 'class']

	age	workclass	fnlwgt	education	education_num	marital_status	occupation	relationship	race	sex	capital_gain	capital_loss	hours_per_week	nativ
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	Un
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	Un
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	Un
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	Un
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	

Fig 2: Added Columns

Step 2) Converting continuous data into categorical data

We use binning method to convert continuous data into categorical data

df['Age'] = pd.cut(x=df['age'], bins=[0, 18, 30, 50, 100], labels=['Underage', 'Young', 'Adult','Elderly'])

Bins	**Labels**
0-18	Underage
18-30	Young
30-50	Adult
50-100	Elderly

Similarly, for hours_per_week

df['Hours_per_Week'] = pd.cut(x=df['hours_per_week'], bins=[0, 20, 40, 100], labels=['Part-Time', 'Full-Time', 'Overtime'])

Bins	**Labels**
0-20	Part-Time
20-40	Full-Time
40-100	Overtime

Age	Hours_per_Week
Adult	Full-Time
Adult	Part-Time
Adult	Full-Time
Elderly	Full-Time
Young	Full-Time

Fig 3: New Age and Hours_per_Week Column

This is what new Age and Hours_per_Week columns look.

Step 3) Dropping continuous age and hours_per_week column as we created categorical ones.

Dropped fnlwgt and education number because they were random and due to inability to convert into categorical.

Dropped capital_gain and capital_loss as they have large number of 0 values.

df = df.drop(['age','fnlwgt','education_num','hours_per_week','capital_loss', 'capital_gain'], axis = 1)

Before dropping

ucation_num	marital_status	occupation	relationship	race	sex	capital_gain	capital_loss	hours_per_week	native_country	class	Age	Hours_per_Week
13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K	Adult	Full-Time
13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K	Adult	Part-Time
9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K	Adult	Full-Time
7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K	Elderly	Full-Time
13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K	Young	Full-Time

Fig 4: Before Dropping Columns

After dropping

```
df = df.drop(['age','fnlwgt','education_num','hours_per_week','capital_loss', 'capital_gain'], axis = 1)
df.head()
```

	workclass	education	marital_status	occupation	relationship	race	sex	native_country	class	Age	Hours_per_Week
0	State-gov	Bachelors	Never-married	Adm-clerical	Not-in-family	White	Male	United-States	<=50K	Adult	Full-Time
1	Self-emp-not-inc	Bachelors	Married-civ-spouse	Exec-managerial	Husband	White	Male	United-States	<=50K	Adult	Part-Time
2	Private	HS-grad	Divorced	Handlers-cleaners	Not-in-family	White	Male	United-States	<=50K	Adult	Full-Time
3	Private	11th	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	United-States	<=50K	Elderly	Full-Time
4	Private	Bachelors	Married-civ-spouse	Prof-specialty	Wife	Black	Female	Cuba	<=50K	Young	Full-Time

Fig 5: After Dropping Columns

Step 4) Removing whitespace

df = df.applymap(lambda space: space.strip() if type(space) is str else space)

ALGORITHMS

Apriori Algorithm:

```
#We will start the timer for Apriori Algorithm here
start_time = timeit.default_timer()
# Earlier we cleaned the dataframe. Now we will be using dataframe.values.tolist() to load the
dataset.
def load_df():
    return df.values.tolist()
#First step is generating Candidate1.
def gen_cand1(itemset):
    CANDIDATE1 = []
    for i in itemset:
        for j in i:
            if not [j] in CANDIDATE1:
                CANDIDATE1.append([j])
    return list(map(frozenset, CANDIDATE1))
#Set in unhashable therefore we use frozenset which is nothing but immutable version of python set
object.
itemset = load_df()
CANDIDATE1 = gen_cand1(itemset)
# Scanning Databse
def database_scan(D, Ck, min_sup):
    sup_count = {}
    for t in D:
        for i in Ck:
            if i.issubset(t):
                if not i in sup_count: sup_count[i]=1
                else: sup_count[i] += 1
    total_items = float(len(D))
    r_list = []
    sup_data = {}
    for key in sup_count:
        support = sup_count[key]/total_items
        if support >= min_sup:
            r_list.insert(0,key)
        sup_data[key] = support
    return r_list, sup_data

#In this step we use our knowledge of support count.
#If support of an item is greater than min_support, we insert that item and store it.

# Generating Apriori
def generate_apriori(L_k, k):
    C = []
    Lk_length = len(L_k)
    for l in range(Lk_length):
        for i in range(l+1, Lk_length):
            l1 = list(L_k[l])[:k-2]
```

```
            l2 = list(L_k[i])[:k-2]
            l1.sort()
            l2.sort()
            if l1==l2:
                C.append(L_k[l] | L_k[i])
    return C

# Apriori function in which we specify min_sup and obtain frequent itemsets corresponding to it.
def apriori(itemset, min_sup = 0.12):
    CANDIDATE1 = gen_cand1(itemset)
    D = list(map(set, itemset))
    L1, sup_data = database_scan(D, CANDIDATE1, min_sup)
    L = [L1]
    k = 2
    while (len(L[k-2]) > 0):
        Ck = generate_apriori(L[k-2], k)
        L_k, supK = database_scan(D, Ck, min_sup)
        sup_data.update(supK)
        L.append(L_k)
        k += 1
    return L, sup_data
L,sdata = apriori(itemset)
new_list = []
for index in range(len(L)):
    print("\nL{} ".format(index))
    print("Number of patterns={} \n".format(len(L[index])))
    apriori_freq_pattern = [list(x) for x in L[index]]
    print(apriori_freq_pattern)
```

Output:

For min_sup = 0.4

```
L0
Number of patterns=9

[['Private'], ['Husband'], ['Married-civ-spouse'], ['Full-Time'], ['Adult'], ['<=50K'], ['United-States'], ['Male'], ['White']]

L1
Number of patterns=21

[['White', 'Private'], ['Male', 'Private'], ['United-States', 'Private'], ['Private', '<=50K'], ['Full-Time', 'Private'], ['Whi
te', 'Married-civ-spouse'], ['Male', 'Married-civ-spouse'], ['United-States', 'Married-civ-spouse'], ['Husband', 'Male'], ['Hus
band', 'Married-civ-spouse'], ['Male', 'White'], ['United-States', 'White'], ['Male', 'United-States'], ['White', '<=50K'], ['M
ale', '<=50K'], ['United-States', '<=50K'], ['White', 'Adult'], ['United-States', 'Adult'], ['Full-Time', 'White'], ['Full-Tim
e', 'United-States'], ['Full-Time', '<=50K']]

L2
Number of patterns=13

[['United-States', 'Private', '<=50K'], ['United-States', 'White', 'Private'], ['Male', 'United-States', 'Private'], ['Male',
'White', 'Private'], ['White', 'Private', '<=50K'], ['Husband', 'Male', 'Married-civ-spouse'], ['Full-Time', 'United-States',
'<=50K'], ['Full-Time', 'White', '<=50K'], ['Full-Time', 'United-States', 'White'], ['Male', 'United-States', '<=50K'], ['Unite
d-States', 'White', '<=50K'], ['Male', 'White', '<=50K'], ['Male', 'United-States', 'White']]

L3
Number of patterns=1

[['United-States', 'White', 'Private', '<=50K']]

L4
Number of patterns=0

[]
```

Fig 6: Output for min_sup = 0.4

Improved Apriori Algorithm

```
#Taking a random sample containing 60% of original data
df = df.sample(frac=0.6)
df.describe()
#We will start the timer for Apriori Algorithm here
start_time = timeit.default_timer()
# Earlier we cleaned the dataframe. Now we will be using dataframe.values.tolist() to load the
dataset.
def load_df():
    return df.values.tolist()
#First step is generating Candidate1.
def gen_cand1(itemset):
    CANDIDATE1 = []
    for i in itemset:
        for j in i:
            if not [j] in CANDIDATE1:
                CANDIDATE1.append([j])
    return list(map(frozenset, CANDIDATE1))
#Set in unhashable therefore we use frozenset which is nothing but immutable version of python set
object.

itemset = load_df()
CANDIDATE1 = gen_cand1(itemset)

# Scanning Databse
def database_scan(D, Ck, min_sup):
    sup_count = {}
    for t in D:
        for i in Ck:
            if i.issubset(t):
                if not i in sup_count: sup_count[i]=1
                else: sup_count[i] += 1
    total_items = float(len(D))
    r_list = []
    sup_data = {}
    for key in sup_count:
        support = sup_count[key]/total_items
        if support >= min_sup:
            r_list.insert(0,key)
        sup_data[key] = support
    return r_list, sup_data

#In this step we use our knowledge of support count.
#If support of an item is greater than min_support, we insert that item and store it.

# Generating Apriori
def generate_apriori(L_k, k):
    C = []
    Lk_length = len(L_k)
```

```
    for l in range(Lk_length):
        for i in range(l+1, Lk_length):
            l1 = list(L_k[l])[:k-2]
            l2 = list(L_k[i])[:k-2]
            l1.sort()
            l2.sort()
            if l1==l2:
                C.append(L_k[l] | L_k[i])
    return C

# Apriori function in which we specify min_sup and obtain frequent itemsets corresponding to it.
def apriori(itemset, min_sup = 0.12):
    CANDIDATE1 = gen_cand1(itemset)
    D = list(map(set, itemset))
    L1, sup_data = database_scan(D, CANDIDATE1, min_sup)
    L = [L1]
    k = 2
    while (len(L[k-2]) > 0):
        Ck = generate_apriori(L[k-2], k)
        L_k, supK = database_scan(D, Ck, min_sup)
        sup_data.update(supK)
        L.append(L_k)
        k += 1
    return L, sup_data
L,sdata = apriori(itemset)
new_list = []
for index in range(len(L)):
    print("\nL{} ".format(index))
    print("Number of patterns={} \n".format(len(L[index])))
    apriori_freq_pattern = [list(x) for x in L[index]]
    print(apriori_freq_pattern)
```

Output:

For min_sup = 0.4

```
L0
Number of patterns=9

[['<=50K'], ['Full-Time'], ['Adult'], ['United-States'], ['Male'], ['White'], ['Husband'], ['Married-civ-spouse'], ['Private']]

L1
Number of patterns=21

[['Male', '<=50K'], ['Private', '<=50K'], ['White', '<=50K'], ['United-States', '<=50K'], ['Full-Time', '<=50K'], ['Private',
'Full-Time'], ['White', 'Full-Time'], ['United-States', 'Full-Time'], ['Husband', 'Married-civ-spouse'], ['Private', 'White'],
['Married-civ-spouse', 'White'], ['Male', 'Private'], ['Married-civ-spouse', 'Male'], ['Husband', 'Male'], ['Male', 'White'],
['United-States', 'Private'], ['United-States', 'Married-civ-spouse'], ['United-States', 'White'], ['United-States', 'Male'],
['White', 'Adult'], ['United-States', 'Adult']]

L2
Number of patterns=12

[['United-States', 'Male', '<=50K'], ['United-States', 'White', '<=50K'], ['United-States', 'Private', '<=50K'], ['United-State
s', 'Full-Time', '<=50K'], ['White', 'Full-Time', '<=50K'], ['Private', 'White', '<=50K'], ['United-States', 'White', 'Full-Tim
e'], ['United-States', 'Male', 'White'], ['United-States', 'Male', 'Private'], ['United-States', 'Private', 'White'], ['Male',
'White', 'Private'], ['Male', 'Husband', 'Married-civ-spouse']]

L3
Number of patterns=1

[['United-States', 'White', '<=50K', 'Private']]

L4
Number of patterns=0

[]
```

Fig 7: Output for min_sup = 0.4

FP-Growth Algorithm

```
#Here we start timing
start_time = timeit.default_timer()

#Defining class for Tree
class Tree(object):

    def __init__(self, value, count, parent):
        self.value = value
        self.count = count
        self.parent = parent
        self.link = None
        self.child = []
#function for getting child
    def getting_child(self, value):
        for n in self.child:
            if n.value == value:
                return n
        return None
#function for adding child
    def adding_child(self, value):
        a_child = Tree(value, 1, self)
        self.child.append(a_child)
        return a_child
#Defining class for Building FP Growth Tree
class Build_FPGrowth_Tree(object):
    def __init__(self, trans, threshold, r_value, r_count):
        self.frequent = self.get_freq_items(trans, threshold)
        self.headers = self.gen_header(self.frequent)
        self.root = self.gen_fp_tree(trans, r_value, r_count, self.frequent, self.headers)

#Function for getting frequent items
    def get_freq_items(self, trans, threshold):
        freq_item = {}
        for t in trans:
            for i in t:
                if i in freq_item:
                    freq_item[i] += 1
                else:
                    freq_item[i] = 1
        for key in list(freq_item.keys()):
            if freq_item[key] < threshold:
                del freq_item[key]
        return freq_item
```

```
    def gen_header(self, freq):
        h_table = {}
        for key in freq.keys():
            h_table[key] = None
        return h_table

#Function for building FP Tree
    def gen_fp_tree(self, trans, r_value, r_count, freq, heads):
        r = Tree(r_value, r_count, None)
        for t in trans:
            sort_item = [x for x in t if x in freq]
            sort_item.sort(key=lambda x: freq[x], reverse=True)
            if len(sort_item) > 0:
                self.node_insert(sort_item, r, heads)
        return r
    #Function for inserting tree
    def node_insert(self, items, node, h):
        f = items[0]
        c = node.getting_child(f)
        if c is not None:
            c.count += 1
        else:
            c = node.adding_child(f)
            if h[f] is None:
                h[f] = c
            else:
                cur = h[f]
                while cur.link is not None:
                    cur = cur.link
                cur.link = c
        rem_items = items[1:] # Recurrsive calling
        if len(rem_items) > 0:
            self.node_insert(rem_items, c, h)

    def tree_path(self, n):
        child_num = len(n.child)
        if child_num > 1:
            return False
        elif child_num == 0:
            return True
        else:
            return True and self.tree_path(n.child[0])

    def pattern_mining(self, t_hold): #Pattern Mining
        if self.tree_path(self.root):
            return self.pattern_generation()
```

```
        else:
            return self.zpattern(self.subtrees_mining(t_hold))

        #Conditional tree
    def zpattern(self, pattern):
        i = self.root.value
        if i is not None:
            pattern_2 = {}
            for key in pattern.keys():
                pattern_2[tuple(sorted(list(key) + [i]))] = pattern[key]
            return pattern_2
        return pattern

#Pattern Generation
    def pattern_generation(self):
        pat = {}
        i = self.frequent.keys()
        if self.root.value is None:
            s_value = []
        else:
            s_value = [self.root.value]
            pat[tuple(s_value)] = self.root.count
        for j in range(1, len(i) + 1):
            for k in itertools.combinations(i, j):
                p = tuple(sorted(list(k) + s_value))
                pat[p] =  min([self.frequent[x] for x in k])
        return pat

    def subtrees_mining(self, threshold):
        fre_pat = {}
        m_order = sorted(self.frequent.keys(), key=lambda x: self.frequent[x])
        for each_item in m_order:
            s = []
            cond_tree = []
            n = self.headers[each_item]
            #Getting count of an item
            while n is not None:
                s.append(n)
                n = n.link
            for i in s:
                freq = i.count
                path_tree = []
                parent = i.parent
                while parent.parent is not None:
                    path_tree.append(parent.value)
                    parent = parent.parent
```

```
                for i in range(freq):
                    cond_tree.append(path_tree)
                    #Constructing subtree with frequent patterns
            stree = Build_FPGrowth_Tree(cond_tree, threshold,each_item,
self.frequent[each_item])
            stree_pat = stree.pattern_mining(threshold)
            # Adding patterns generated in subtree to the main tree
            for p in stree_pat.keys():
                if p in fre_pat:
                    fre_pat[p] += stree_pat[p]
                else:
                    fre_pat[p] = stree_pat[p]
        return fre_pat
#Getting Frequent patterns
def fp_growth_freq_patterns(data, sup_threshold):
    tree = Build_FPGrowth_Tree(data, sup_threshold, None, None)
    return tree.pattern_mining(sup_threshold)
#Defining minimum support
min_sup = 0.12
x = min_sup*32561
print("((Pattern) , Support Count) are:- ")
fp_freq_itemsets = fp_growth_freq_patterns(df, x)
fpgrowth_freq_itemsets = list(fp_freq_itemsets.items())
end_time = timeit.default_timer()
fpgrowth_freq_itemsets
```

Output: This is not complete output, just a snapshot of first few frequent patterns
For min_sup = 0.4

```
min_sup = 0.4
x = min_sup*32561
print("((Pattern) , Support Count) are:- ")
fp_freq_itemsets = fp_growth_freq_patterns(df, x)
fpgrowth_freq_itemsets = list(fp_freq_itemsets.items())
end_time = timeit.default_timer()
fpgrowth_freq_itemsets

((Pattern) , Support Count) are:-

[(('Husband', 'Married-civ-spouse'), 13184),
 (('Husband', 'Male', 'Married-civ-spouse'), 13183),
 (('Husband', 'Male'), 13192),
 (('Male', 'Married-civ-spouse'), 13319),
 (('Married-civ-spouse', 'United-States'), 13368),
 (('Married-civ-spouse', 'White'), 13410),
 (('Adult', 'White'), 13194),
 (('Adult', 'United-States'), 13887),
 (('Full-Time', 'Private'), 14465),
 (('<=50K', 'Full-Time', 'White'), 13200),
 (('<=50K', 'Full-Time', 'United-States'), 14315),
 (('Full-Time', 'White'), 16535),
 (('Full-Time', 'United-States', 'White'), 15063),
 (('Full-Time', 'United-States'), 17734),
 (('Male', 'Private', 'White'), 13123),
 (('Male', 'Private', 'United-States'), 13209),
 (('<=50K', 'Male', 'White'), 13085),
 (('<=50K', 'Male', 'United-States'), 13389),
 (('Male', 'White'), 19174),
 (('Male', 'United-States', 'White'), 17653),
 (('Male', 'United-States'), 19488),
 (('<=50K', 'Private', 'White'), 14872),
 (('<=50K', 'Private', 'United-States', 'White'), 13452),
```

Fig 8: Output for min_sup = 0.4

Libraries Used

Pandas, NumPy and Timeit

Results:

Using the following code to get time taken by each algorithm at different minimum support

start_time = timeit.default_timer()
end_time = timeit.default_timer()
total_time = end_time - start_time
total_time

Comparison Table:

Min_sup	Time taken by Apriori in secs	Time taken by improved Apriori in secs	Time taken by FP Growth in secs
0.05	19.27	15.34	14.04
0.08	13.47	10.33	9.89
0.12	7.93	7.37	6.87

Comparing all the algorithms for different minimum support:

For Minimum Support = 0.05

Apriori

```
for index in range(len(L)):
    print("\nL{} ".format(index))
    print("Number of patterns={} \n".format(len(L[index])))
    apriori_freq_pattern = [list(x) for x in L[index]]
    print(apriori_freq_pattern)
```

```
K'], ['HS-grad', 'United-States', 'White', 'Full-Time', 'Male', 'Private', '<=50K'], ['United-States', 'Adult', 'White', 'Ful
l-Time', 'Male', 'Private', '<=50K'], ['Husband', 'United-States', 'Adult', 'White', 'Married-civ-spouse', 'Male', '<=50K']]

L7
Number of patterns=9

[['HS-grad', 'Husband', 'United-States', 'White', 'Married-civ-spouse', 'Male', 'Private', '<=50K'], ['Husband', 'United-Stat
es', 'Adult', 'White', 'Full-Time', 'Married-civ-spouse', 'Male', '<=50K'], ['Husband', 'United-States', 'White', 'Full-Tim
e', 'Married-civ-spouse', 'Male', 'Private', '<=50K'], ['Husband', 'United-States', 'Overtime', '>50K', 'White', 'Married-civ
-spouse', 'Male', 'Private'], ['Husband', 'United-States', 'Overtime', '>50K', 'Adult', 'White', 'Married-civ-spouse', 'Mal
e'], ['Husband', 'United-States', 'Adult', 'White', 'Married-civ-spouse', 'Male', 'Private', '<=50K'], ['Husband', 'United-St
ates', 'Overtime', 'Adult', 'White', 'Married-civ-spouse', 'Male', 'Private'], ['Husband', 'United-States', 'Adult', '>50K',
'White', 'Married-civ-spouse', 'Male', 'Private'], ['Husband', 'United-States', 'Adult', 'White', 'Full-Time', 'Married-civ-s
pouse', 'Male', 'Private']]

L8
Number of patterns=0

[]
```

```
end_time = timeit.default_timer() #ending timer
total_time = end_time - start_time
total_time
```

```
19.27536420000024
```

Fig 8: Apriori Output for min_sup = 0.05

Improved Apriori

```python
for index in range(len(L)):
    print("\nL{} ".format(index))
    print("Number of patterns={} \n".format(len(L[index])))
    improved_apriori_freq_pattern = [list(x) for x in L[index]]
    print(improved_apriori_freq_pattern)
```

```
vate', '>50K', 'White', 'Overtime', 'Married-civ-spouse'], ['Husband', 'Male', 'Private', 'Adult', 'White', 'Overtime', 'Marr
ied-civ-spouse'], ['Husband', 'Male', 'Private', '>50K', 'White', 'Overtime', 'Married-civ-spouse']]

L7
Number of patterns=9

[['United-States', 'Husband', 'Married-civ-spouse', 'Full-Time', 'Adult', 'White', '<=50K', 'Male'], ['United-States', 'Husba
nd', 'Male', 'Private', 'Full-Time', 'White', '<=50K', 'Married-civ-spouse'], ['United-States', 'Husband', 'Male', 'Private',
'Full-Time', 'Adult', 'White', 'Married-civ-spouse'], ['United-States', 'Husband', 'Male', 'Private', 'White', '<=50K', 'Marr
ied-civ-spouse', 'HS-grad'], ['United-States', 'Husband', 'Male', 'Private', 'Adult', 'White', '<=50K', 'Married-civ-spous
e'], ['United-States', 'Husband', 'Male', 'Private', 'Adult', 'White', 'Overtime', 'Married-civ-spouse'], ['United-States',
'Husband', 'Male', 'Private', '>50K', 'White', 'Overtime', 'Married-civ-spouse'], ['United-States', 'Husband', 'Married-civ-s
pouse', '>50K', 'Adult', 'Overtime', 'White', 'Male'], ['United-States', 'Husband', 'Male', 'Private', '>50K', 'Adult', 'Whit
e', 'Married-civ-spouse']]

L8
Number of patterns=0

[]
```

```python
end_time = timeit.default_timer()
total_time = end_time - start_time
total_time
```

```
15.345461100000193
```

Fig 9: Improved Apriori Output for min_sup = 0.05

FP Tree

```python
min_sup = 0.05
x = min_sup*32561
print("((Pattern) , Support Count) are:- ")
fp_freq_itemsets = fp_growth_freq_patterns(df, x)
fpgrowth_freq_itemsets = list(fp_freq_itemsets.items())
end_time = timeit.default_timer()
fpgrowth_freq_itemsets
```

```
((Pattern) , Support Count) are:-

[(('Masters',), 1723),
 (('Machine-op-inspct', 'United-States'), 1687),
 (('<=50K', 'Machine-op-inspct'), 1752),
 (('<=50K', 'Machine-op-inspct', 'Private'), 1688),
 (('Machine-op-inspct', 'Private'), 1913),
 (('Local-gov', 'White'), 1720),
 (('Local-gov', 'United-States', 'White'), 1632),
 (('Local-gov', 'United-States'), 1956),
 (('Married-civ-spouse', 'Self-emp-not-inc'), 1680),
 (('<=50K', 'Self-emp-not-inc', 'United-States'), 1654),
 (('<=50K', 'Self-emp-not-inc', 'White'), 1671),
 (('Male', 'Self-emp-not-inc', 'United-States'), 1956),
 (('Male', 'Self-emp-not-inc', 'United-States', 'White'), 1860),
 (('Male', 'Self-emp-not-inc', 'White'), 1985),
 (('Self-emp-not-inc', 'United-States'), 2313),
 (('Self-emp-not-inc', 'United-States', 'White'), 2190),
 (('Self-emp-not-inc', 'White'), 2342),
 (('Part-Time', 'Private', 'United-States'), 1665)
```

```python
total_time = end_time - start_time
total_time
```

```
14.046819700000015
```

Fig 10: FP Growth Output for min_sup = 0.05

For minimum support = 0.08

Apriori

```
for index in range(len(L)):
    print("\nL{} ".format(index))
    print("Number of patterns={} \n".format(len(L[index])))
    apriori_freq_pattern = [list(x) for x in L[index]]
    print(apriori_freq_pattern)
```

```
L6
Number of patterns=14

[['Husband', 'White', 'Full-Time', 'Married-civ-spouse', 'Male', 'Private', '<=50K'], ['Husband', 'United-States', 'White',
'Full-Time', 'Married-civ-spouse', 'Male', '<=50K'], ['Husband', 'United-States', 'White', 'Married-civ-spouse', 'Male', 'Pri
vate', '<=50K'], ['Husband', 'United-States', 'Overtime', 'White', 'Married-civ-spouse', 'Male', 'Private'], ['Husband', 'Uni
ted-States', 'Overtime', 'Adult', 'White', 'Married-civ-spouse', 'Male'], ['Young', 'United-States', 'Never-married', 'Whit
e', 'Full-Time', 'Private', '<=50K'], ['Husband', 'United-States', 'White', 'Full-Time', 'Married-civ-spouse', 'Male', 'Priva
te'], ['Husband', 'United-States', 'Adult', 'White', 'Full-Time', 'Married-civ-spouse', 'Male'], ['Husband', 'United-States',
'>50K', 'White', 'Married-civ-spouse', 'Male', 'Private'], ['Husband', 'United-States', 'Adult', '>50K', 'White', 'Married-ci
v-spouse', 'Male'], ['Husband', 'United-States', 'Adult', 'White', 'Married-civ-spouse', 'Male', 'Private'], ['Husband', 'Uni
ted-States', 'Overtime', '>50K', 'White', 'Married-civ-spouse', 'Male'], ['Husband', 'United-States', 'Full-Time', 'Married-c
iv-spouse', 'Male', 'Private', '<=50K'], ['Husband', 'United-States', 'Adult', 'White', 'Married-civ-spouse', 'Male', '<=50
K']]

L7
Number of patterns=0

[]
```

```
end_time = timeit.default_timer() #ending timer
total_time = end_time - start_time
total_time
```

```
13.4794423000003
```

Fig 11: Apriori Output for min_sup = 0.08

Improved Apriori

```
for index in range(len(L)):
    print("\nL{} ".format(index))
    print("Number of patterns={} \n".format(len(L[index])))
    improved_apriori_freq_pattern = [list(x) for x in L[index]]
    print(improved_apriori_freq_pattern)
```

```
L6
Number of patterns=14

[['United-States', 'Husband', 'Male', 'Full-Time', 'White', '<=50K', 'Married-civ-spouse'], ['United-States', 'Husband', 'Mal
e', 'Private', 'Full-Time', '<=50K', 'Married-civ-spouse'], ['Husband', 'Male', 'Private', 'Full-Time', 'White', '<=50K', 'Ma
rried-civ-spouse'], ['United-States', 'Husband', 'Male', 'Full-Time', 'Adult', 'White', 'Married-civ-spouse'], ['United-State
s', 'Husband', 'Male', 'Adult', 'White', '<=50K', 'Married-civ-spouse'], ['United-States', 'Husband', 'Male', 'Private', 'Whi
te', '<=50K', 'Married-civ-spouse'], ['United-States', 'Full-Time', 'Young', 'White', 'Never-married', '<=50K', 'Private'],
['United-States', 'Husband', 'Male', 'Private', 'Full-Time', 'White', 'Married-civ-spouse'], ['United-States', 'Husband', 'Ma
le', '>50K', 'Adult', 'White', 'Married-civ-spouse'], ['United-States', 'Husband', 'Male', '>50K', 'White', 'Overtime', 'Marr
ied-civ-spouse'], ['United-States', 'Husband', 'Male', 'Adult', 'White', 'Overtime', 'Married-civ-spouse'], ['United-States',
'Husband', 'Male', 'Private', 'Overtime', 'White', 'Married-civ-spouse'], ['United-States', 'Husband', 'Male', 'Private', '>5
0K', 'White', 'Married-civ-spouse'], ['United-States', 'Husband', 'Male', 'Private', 'Adult', 'White', 'Married-civ-spouse']]

L7
Number of patterns=0

[]
```

```
end_time = timeit.default_timer()
total_time = end_time - start_time
total_time
```

```
10.33158370000001
```

Fig 12: Imprroved Apriori Output for min_sup = 0.08

FP Growth

```
min_sup = 0.08
x = min_sup*32561
print("((Pattern) , Support Count) are:- ")
fp_freq_itemsets = fp_growth_freq_patterns(df, x)
fpgrowth_freq_itemsets = list(fp_freq_itemsets.items())
end_time = timeit.default_timer()
fpgrowth_freq_itemsets
```

```
((Pattern) , Support Count) are:-

[(('Part-Time', 'United-States'), 2680),
 (('<=50K', 'Part-Time'), 2733),
 (('<=50K', 'Black'), 2737),
 (('Black', 'United-States'), 2832),
 (('Other-service', 'Private'), 2740),
 (('<=50K', 'Other-service', 'Private'), 2640),
 (('Other-service', 'United-States'), 2777),
 (('<=50K', 'Other-service', 'United-States'), 2671),
 (('<=50K', 'Other-service'), 3158),
 (('Female', 'Unmarried'), 2654),
 (('United-States', 'Unmarried'), 3033),
 (('<=50K', 'United-States', 'Unmarried'), 2837),
 (('<=50K', 'Unmarried'), 3228),
 (('<=50K', 'Sales'), 2667),
 (('Private', 'Sales'), 2942),
 (('Private', 'Sales', 'United-States'), 2734),
 (('Sales', 'White'), 3237),
 (('Sales', 'United-States', 'White'), 3065)
```

```
total_time = end_time - start_time
total_time
```

```
9.89017530000001
```

Fig 13: FP Growth Output for min_sup = 0.08

For minimum support = 0.12

Apriori

```
for index in range(len(L)):
    print("\nL{} ".format(index))
    print("Number of patterns={} \n".format(len(L[index])))
    apriori_freq_pattern = [list(x) for x in L[index]]
    print(apriori_freq_pattern)
```

```
L0
Number of patterns=24

[['Craft-repair'], ['Own-child'], ['Some-college'], ['Overtime'], ['>50K'], ['Young'], ['Female'], ['Prof-specialty'], ['Elde
rly'], ['Divorced'], ['HS-grad'], ['Private'], ['Husband'], ['Exec-managerial'], ['Married-civ-spouse'], ['Full-Time'], ['Adu
lt'], ['<=50K'], ['United-States'], ['Male'], ['White'], ['Not-in-family'], ['Never-married'], ['Bachelors']]

L1
Number of patterns=106

[['Some-college', 'White'], ['Full-Time', 'Some-college'], ['Some-college', '<=50K'], ['<=50K', 'Overtime'], ['Young', 'Never
-married'], ['Young', 'White'], ['Young', 'United-States'], ['Own-child', 'Never-married'], ['Own-child', 'White'], ['Own-chi
ld', 'United-States'], ['Own-child', '<=50K'], ['Male', 'Young'], ['Male', 'Some-college'], ['Some-college', 'United-State
s'], ['Some-college', 'Private'], ['Private', 'Never-married'], ['Female', 'Never-married'], ['>50K', 'Adult'], ['>50K', 'Pri
vate'], ['Adult', 'Overtime'], ['Private', 'Overtime'], ['HS-grad', 'Married-civ-spouse'], ['HS-grad', 'Husband'], ['Elderl
y', 'White'], ['>50K', 'White'], ['>50K', 'Male'], ['>50K', 'United-States'], ['>50K', 'Married-civ-spouse'], ['>50K', 'Husba
nd'], ['White', 'Overtime'], ['Male', 'Overtime'], ['United-States', 'Overtime'], ['Married-civ-spouse', 'Overtime'], ['Husba
nd', 'Overtime'], ['White', 'Female'], ['United-States', 'Female'], ['Female', 'Adult'], ['Female', '<=50K'], ['Full-Time',
```

```
end_time = timeit.default_timer() #ending timer
total_time = end_time - start_time
total_time
```

```
7.937175000000025
```

Fig 14: Apriori Output for min_sup = 0.12

Improved Apriori

```
for index in range(len(L)):
    print("\nL{} ".format(index))
    print("Number of patterns={} \n".format(len(L[index])))
    improved_apriori_freq_pattern = [list(x) for x in L[index]]
    print(improved_apriori_freq_pattern)
```

```
L0
Number of patterns=24

[['Some-college'], ['Bachelors'], ['Not-in-family'], ['Craft-repair'], ['Divorced'], ['Young'], ['Female'], ['Own-child'],
['Never-married'], ['HS-grad'], ['Exec-managerial'], ['Elderly'], ['<=50K'], ['Overtime'], ['Private'], ['Full-Time'], ['Adul
t'], ['>50K'], ['United-States'], ['Male'], ['White'], ['Husband'], ['Prof-specialty'], ['Married-civ-spouse']]

L1
Number of patterns=107

[['Some-college', 'Full-Time'], ['Some-college', 'Male'], ['Some-college', 'White'], ['United-States', 'Some-college'], ['Som
e-college', 'Private'], ['Some-college', '<=50K'], ['Female', 'Adult'], ['Bachelors', 'White'], ['United-States', 'Bachelor
s'], ['Male', 'Never-married'], ['Full-Time', 'Never-married'], ['Male', 'Young'], ['White', 'Not-in-family'], ['Male', 'Not-
in-family'], ['United-States', 'Not-in-family'], ['Not-in-family', 'Full-Time'], ['Private', 'Not-in-family'], ['Not-in-famil
y', '<=50K'], ['Not-in-family', 'Never-married'], ['Female', 'Full-Time'], ['Full-Time', 'Young'], ['Overtime', '<=50K'], ['U
nited-States', 'Divorced'], ['Divorced', '<=50K'], ['White', 'Never-married'], ['United-States', 'Never-married'], ['Privat
e', 'Never-married'], ['<=50K', 'Never-married'], ['Own-child', 'White'], ['Own-child', 'United-States'], ['Own-child', 'Priv
ate'], ['Own-child', '<=50K'], ['Own-child', 'Never-married'], ['Female', 'White'], ['United-States', 'Female'], ['Female',
```

```
end_time = timeit.default_timer()
total_time = end_time - start_time
total_time
```

```
7.377294800000527
```

Fig 15: Improved Apriori Output for min_sup = 0.12

FP Growth

```
min_sup = 0.12
x = min_sup*32561
print("((Pattern) , Support Count) are:- ")
fp_freq_itemsets = fp_growth_freq_patterns(df, x)
fpgrowth_freq_itemsets = list(fp_freq_itemsets.items())
end_time = timeit.default_timer()
fpgrowth_freq_itemsets
```

```
((Pattern) , Support Count) are:-

[(('Exec-managerial',), 4066),
 (('Craft-repair',), 4099),
 (('Prof-specialty',), 4140),
 (('?',), 4262),
 (('<=50K', 'Divorced'), 3980),
 (('Divorced', 'United-States'), 4162),
 (('Own-child', 'United-States', 'White'), 4022),
 (('<=50K', 'Own-child', 'United-States', 'White'), 3966),
 (('<=50K', 'Own-child', 'White'), 4196),
 (('Never-married', 'Own-child', 'United-States'), 4163),
 (('<=50K', 'Never-married', 'Own-child', 'United-States'), 4134),
 (('<=50K', 'Never-married', 'Own-child'), 4451),
 (('Own-child', 'United-States'), 4691),
 (('<=50K', 'Own-child', 'United-States'), 4632),
 (('<=50K', 'Own-child'), 5001),
 (('Bachelors', 'White'), 4682),
 (('Bachelors', 'United-States', 'White'), 4380),
```

```
total_time = end_time - start_time
total_time
```

```
6.8724927000002936
```

Fig 16: FP Growth Output for min_sup = 0.12

Conclusion

It is evident improved Apriori algorithm is better than the traditional Apriori algorithm as during the comparison we found time required to execute improved algorithm is less than the time required to execute traditional algorithm. Also, it is quite clear that FP Growth is the fastest algorithm out of all.

For the improvement, I used sampling technique and found out that the sample taken gives all the frequent patterns efficiently. Minimum support is lowered because of less number of rows. Only one scan is required as it covered all the frequent patterns as compared to normal Apriori algorithm.

L6 is same for Apriori and Improved Apriori algorithm

```
for index in range(len(L)):
    print("\nL{} ".format(index))
    print("Number of patterns={} \n".format(len(L[index])))
    apriori_freq_pattern = [list(x) for x in L[index]]
    print(apriori_freq_pattern)
```

```
use', 'Male', 'Private'], ['Husband', 'United-States', 'Adult', 'White', 'Male', 'Private'], ['Husband', 'United-States', 'Ad
ult', 'White', 'Married-civ-spouse', 'Private'], ['Husband', 'Adult', 'White', 'Married-civ-spouse', 'Male', 'Private'], ['Hu
sband', 'United-States', 'White', 'Full-Time', 'Married-civ-spouse', 'Male'], ['Husband', 'White', 'Full-Time', 'Married-civ-
spouse', 'Male', 'Private'], ['Husband', 'United-States', 'White', 'Married-civ-spouse', 'Male', 'Private'], ['Husband', 'Uni
ted-States', 'Adult', 'Married-civ-spouse', 'Male', 'Private'], ['Husband', 'United-States', '>50K', 'White', 'Married-civ-sp
ouse', 'Male'], ['Husband', 'United-States', 'Overtime', 'White', 'Married-civ-spouse', 'Male'], ['Husband', 'United-States',
'Full-Time', 'Married-civ-spouse', 'Male', 'Private'], ['Husband', 'United-States', 'Married-civ-spouse', 'Male', 'Private',
'<=50K'], ['United-States', 'White', 'Full-Time', 'Male', 'Private', '<=50K'], ['Husband', 'United-States', 'White', 'Married
-civ-spouse', 'Male', '<=50K'], ['Husband', 'United-States', 'Adult', 'White', 'Married-civ-spouse', 'Male']]

L6
Number of patterns=1

[['Husband', 'United-States', 'Adult', 'White', 'Married-civ-spouse', 'Male', 'Private']]

L7
Number of patterns=0

[]
```

```
end_time = timeit.default_timer() #ending timer
total_time = end_time - start_time
total_time
```

```
7.937175000000025
```

Fig 17: Apriori Output for min_sup = 0.12

```
for index in range(len(L)):
    print("\nL{} ".format(index))
    print("Number of patterns={} \n".format(len(L[index])))
    improved_apriori_freq_pattern = [list(x) for x in L[index]]
    print(improved_apriori_freq_pattern)
```

```
ime', 'White', '<=50K', 'Private'], ['United-States', 'Male', 'Married-civ-spouse', 'Full-Time', 'White', 'Private'], ['Marri
ed-civ-spouse', 'Husband', 'Full-Time', 'White', '<=50K', 'Male'], ['United-States', 'Husband', 'Married-civ-spouse', 'Whit
e', '<=50K', 'Male'], ['Married-civ-spouse', 'Husband', 'Private', 'Adult', 'White', 'Male'], ['United-States', 'Husband', 'M
ale', 'Married-civ-spouse', 'White', 'Private'], ['United-States', 'Husband', 'Married-civ-spouse', 'Adult', 'White', 'Privat
e'], ['United-States', 'Husband', 'Male', 'Adult', 'White', 'Private'], ['United-States', 'Husband', 'Married-civ-spouse', 'P
rivate', 'Adult', 'Male'], ['United-States', 'Married-civ-spouse', 'Private', 'Adult', 'White', 'Male'], ['United-States', 'H
usband', 'Married-civ-spouse', 'White', 'Overtime', 'Male'], ['United-States', 'Husband', 'Married-civ-spouse', 'Adult', 'Whi
te', 'Male'], ['United-States', 'Husband', 'Married-civ-spouse', 'Full-Time', 'White', 'Male'], ['United-States', 'Husband',
'Married-civ-spouse', '>50K', 'White', 'Male']]

L6
Number of patterns=1

[['United-States', 'Husband', 'Married-civ-spouse', 'Male', 'Adult', 'White', 'Private']]

L7
Number of patterns=0

[]
```

```
end_time = timeit.default_timer()
total_time = end_time - start_time
total_time
```

```
7.377294800000527
```

Fig 18: Improved Apriori Output for min_sup = 0.12

Reference

[1] Yu Cheng, Ying Xiong. Research and Improvement of Apriori Algorithm for Association Rules. 2010 2nd International Workshop on Intelligent Systems and Applications

[2] Jiawei Han, Michelin Kamber, Jian Pei. Data Mining Concepts and Techniques, 248-264.

[3] https://notebook.community/aleph314/K2/Data%20Mining/Association%20Rules/assoc_mining_problems

[4] Reynaldo John Tristan Mahinay Jr., Franz Stewart Dizon, Stephen Kyle Farinas and Harry Pardo. Learning of High Dengue Incidence with Clustering and FP-Growth Algorithm using WHO Historical Data. *3rd IEEE International Conference on Agents (ICA 2018)*

[5] Jiao Yabing. Research of an Improved Apriori Algorithm in Data Mining Association Rules. International Journal of Computer and Communication Engineering, Vol. 2, No. 1, January 2013

[6] https://towardsdatascience.com/understand-and-build-fp-growth-algorithm-in-python-d8b989bab342

[7] https://www.geeksforgeeks.org/ml-frequent-pattern-growth-algorithm/