# MTL782 ASSIGNMENT

*Prasoon Bajpai - 2019MT60824*
*Sparsh Chaudhri - 2019MT10765*
*Rajdeep Das - 2019MT10718*

## 1. UCI MACHINE LEARNING REPOSITORY (ADULT DATABASE) : COMPARISON OF CLASSIFICATION TECHNIQUES

### 1.1. Description of Data

This data was extracted from the census bureau database found at `http://www.census.gov/ftp/pub/DES/www/welcome.html`.

- Ronny Kohavi and Barry Becker, Data Mining and Visualization Silicon Graphics
- Split into train-test using MLC++ GenCVFiles (2/3, 1/3 random)
- 48842 instances, mix of continuous and discrete (train=32561, test=16281)
- 45222 if instances with unknown values are removed (train=30162, test=15060)
- Duplicate or conflicting instances : 6
- Probability for the label '>50K' : 23.93% / 24.78% (without unknowns)
- Probability for the label '≤50K' : 76.07% / 75.22% (without unknowns)

***Prediction task is to determine whether a person makes over 50K a year.***
Attribute information:

- **age**: continuous.
- **workclass**: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov,
- Without-pay, Never-worked.
- **fnlwgt**: continuous.
- **education**: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool.
- education-num: continuous.
- **marital-status**: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse.
- **occupation**: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
- **relationship**: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
- **race**: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
- **sex**: Female, Male.
- **capital-gain**: continuous.
- **capital-loss**: continuous.
- **hours-per-week**: continuous.
- **native-country**: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, TrinadadTobago, Peru, Hong, Holand-Netherlands.

| Approach | Accuracy |
|---|---|
| Naive Bayes | 0.776 |
| Random Forest | 0.851 |
| KNN | 0.7913793103448274 |
| Decision Tree | 0.8110079575596817 |

## 1.2. Benefits obtained from Data Mining

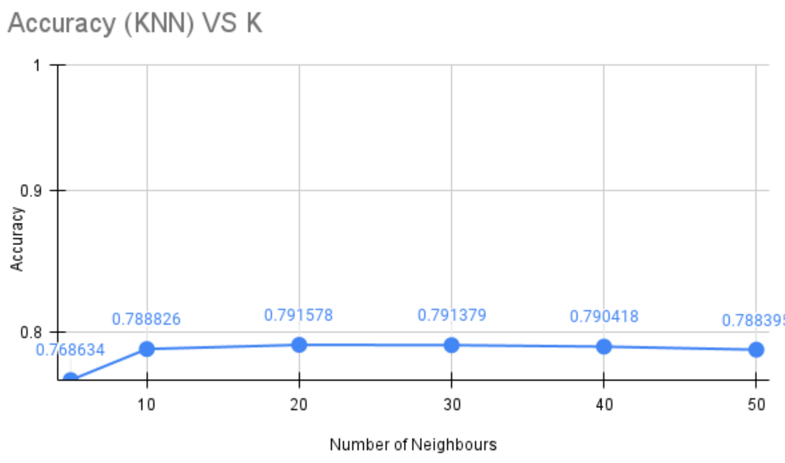
- Application of data mining on this dataset gives an effective tool to the census authorities to predict the annual income of a person based on his information.
- It helps in controlling the resources required to collect annual income information by achieving the task through random sampling.

## 1.3. Data Quality Issues and Response

- There are many examples with values for some attributes missing.
- 48842 instances, mix of continuous and discrete (train=32561, test=16281), 45222 if instances with unknown values are removed (train=30162, test=15060)
- Duplicate or conflicting instances : 6
- While applying various models, all instances that had any values missing were **removed** from the dataset.
- All the ordinal attributes were one hot encoded so as to treat them as numerical attributes. This led to an increase to 104 attributes for the dataset.

## 1.4. Applying different classifiers

- 10-fold cross validation was employed to account for the relative average accuracy of the four methods of classification.
- Hypertuning for **K-NN Classifier**[1] revealed that the maximum accuracy was obtained for value of K = 30. For **Naives Bayes** Classifer, `CategoricalNB`[1] implementation was used. It assumes that each feature, which is described by the index i, has its own categorical distribution. For **Random Forest** classifier, the value of $max - depth$ parameter was set to **None** and *entropy* criteria was used.



## 2. REFERENCES

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

# Decision Tree Classifier ¶

In [1]:

```
import numpy as np
import pandas as pd
```

In [2]:

```
TRAIN_ADDRESS = "Train.csv"
df = pd.read_csv(TRAIN_ADDRESS, header = None)
X = pd.DataFrame(df, columns = df.columns, index = [i for i in range(df.shape[0])])
Y = pd.DataFrame(df, columns = [df.columns[-1]], index = [i for i in range(df.shape[
X = X[X.columns[:-1]]
```

In [3]:

```
checkX = []
checkY = []
for ind, rows in X.iterrows():
    if (" ?" in list(rows)):
        checkX.append(X.index[ind])
        checkY.append(Y.index[ind])
X = X.drop(checkX)
Y = Y.drop(checkY)
```

One Hot Encoding Nominal Attributes

In [4]:

```
X = pd.get_dummies(X,
prefix=None,
prefix_sep="_",
dummy_na=False,
columns=[1,3,5,6,7,8,9,13],
sparse=False,
drop_first=False,
dtype=None)
```

In [5]:

```
X
```

Out[5]:

| | 0 | 2 | 4 | 10 | 11 | 12 | 1_ Federal-gov | 1_ Local-gov | 1_ Private | 1_ Self-emp-inc | ... | 13_ Portugal | 13_ Puerto-Rico |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 39 | 77516 | 13 | 2174 | 0 | 40 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **1** | 50 | 83311 | 13 | 0 | 0 | 13 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **2** | 38 | 215646 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **3** | 53 | 234721 | 7 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **4** | 28 | 338409 | 13 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **32556** | 27 | 257302 | 12 | 0 | 0 | 38 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32557** | 40 | 154374 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32558** | 58 | 151910 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32559** | 22 | 201490 | 9 | 0 | 0 | 20 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32560** | 52 | 287927 | 9 | 15024 | 0 | 40 | 0 | 0 | 0 | 1 | ... | 0 | 0 |

30162 rows × 104 columns

In [6]:

```
Y = pd.get_dummies(Y,
prefix=None,
prefix_sep="_",
dummy_na=False,
columns = None,
sparse=False,
drop_first=True,
dtype=None)
```

In [7]:

```
from sklearn import tree
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, Y)
```

In [8]:

```
k = 10
accuracy = 0
size = X.shape[0]//k
for i in range(k):
    TEST = X[i*size:(i+1)*size]
    TEST_Y = Y[i*size:(i+1)*size].to_numpy()
    TRAIN = pd.concat([X[0:i*size],X[(i+1)*size:]])
    TRAIN_Y = pd.concat([Y[0:i*size],Y[(i+1)*size:]])
    clf = tree.DecisionTreeClassifier()
    clf = clf.fit(TRAIN, TRAIN_Y)
    output = clf.predict(TEST)
    correct = 0
    for i in range(TEST.shape[0]):
        if(output[i] == TEST_Y[i]): correct += 1
    accuracy += ((correct/TEST.shape[0])/k)
print("FINAL AVERAGE ACCURACY AFTER",k,"-fold cross validation =",accuracy)
```

```
FINAL AVERAGE ACCURACY AFTER 10 -fold cross validation = 0.81104111405
83554
```

FINAL AVERAGE ACCURACY AFTER 10 -fold cross validation = 0.8110411140583554

In [ ]:

```

```

# KNN Classifier

In [1]:

```
import numpy as np
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
```

In [2]:

```
TRAIN_ADDRESS = "Train.csv"
df = pd.read_csv(TRAIN_ADDRESS, header = None)
X = pd.DataFrame(df, columns = df.columns, index = [i for i in range(df.shape[0])])
Y = pd.DataFrame(df, columns = [df.columns[-1]], index = [i for i in range(df.shape[
X = X[X.columns[:-1]]
```

In [3]:

```
checkX = []
checkY = []
for ind, rows in X.iterrows():
    if (" ?" in list(rows)):
        checkX.append(X.index[ind])
        checkY.append(Y.index[ind])
X = X.drop(checkX)
Y = Y.drop(checkY)
```

One Hot Encoding Nominal Attributes

In [4]:

```
X = pd.get_dummies(X,
prefix=None,
prefix_sep="_",
dummy_na=False,
columns=[1,3,5,6,7,8,9,13],
sparse=False,
drop_first=False,
dtype=None)
```

In [5]:

```
X
```

Out[5]:

| | 0 | 2 | 4 | 10 | 11 | 12 | 1_ Federal-gov | 1_ Local-gov | 1_ Private | 1_ Self-emp-inc | ... | 13_ Portugal | 13_ Puerto-Rico |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 39 | 77516 | 13 | 2174 | 0 | 40 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **1** | 50 | 83311 | 13 | 0 | 0 | 13 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **2** | 38 | 215646 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **3** | 53 | 234721 | 7 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **4** | 28 | 338409 | 13 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **32556** | 27 | 257302 | 12 | 0 | 0 | 38 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32557** | 40 | 154374 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32558** | 58 | 151910 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32559** | 22 | 201490 | 9 | 0 | 0 | 20 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32560** | 52 | 287927 | 9 | 15024 | 0 | 40 | 0 | 0 | 0 | 1 | ... | 0 | 0 |

30162 rows × 104 columns

In [6]:

```
Y = pd.get_dummies(Y,
prefix=None,
prefix_sep="_",
dummy_na=False,
columns = None,
sparse=False,
drop_first=True,
dtype=None)
```

In [13]:

```
k = 10
accuracy = 0
size = X.shape[0]//k
for i in range(k):
    TEST = X[i*size:(i+1)*size]
    TEST_Y = Y[i*size:(i+1)*size].to_numpy()
    TRAIN = pd.concat([X[0:i*size],X[(i+1)*size:]])
    TRAIN_Y = pd.concat([Y[0:i*size],Y[(i+1)*size:]])
    clf = KNeighborsClassifier(n_neighbors=30)
    clf = clf.fit(TRAIN, TRAIN_Y.to_numpy().reshape((TRAIN_Y.shape[0],)))
    output = clf.predict(TEST)
    correct = 0
    for i in range(TEST.shape[0]):
        if(output[i] == TEST_Y[i]): correct += 1
    accuracy += ((correct/TEST.shape[0])/k)
print("FINAL AVERAGE ACCURACY AFTER",k,"-fold cross validation =",accuracy)
```

```
FINAL AVERAGE ACCURACY AFTER 10 -fold cross validation = 0.79137931034
48274
```

FINAL AVERAGE ACCURACY AFTER 10 -fold cross validation = 0.7913793103448274

In [ ]:

# Random Forest Classifier

In [1]:

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
```

In [2]:

```
TRAIN_ADDRESS = "Train.csv"
df = pd.read_csv(TRAIN_ADDRESS, header = None)
X = pd.DataFrame(df, columns = df.columns, index = [i for i in range(df.shape[0])])
Y = pd.DataFrame(df, columns = [df.columns[-1]], index = [i for i in range(df.shape[
X = X[X.columns[:-1]]
```

In [3]:

```
checkX = []
checkY = []
for ind, rows in X.iterrows():
    if (" ?" in list(rows)):
        checkX.append(X.index[ind])
        checkY.append(Y.index[ind])
X = X.drop(checkX)
Y = Y.drop(checkY)
```

One Hot Encoding Nominal Attributes

In [4]:

```
X = pd.get_dummies(X,
prefix=None,
prefix_sep="_",
dummy_na=False,
columns=[1,3,5,6,7,8,9,13],
sparse=False,
drop_first=False,
dtype=None)
```

In [5]:

```
X
```

Out[5]:

| | 0 | 2 | 4 | 10 | 11 | 12 | 1_ Federal-gov | 1_ Local-gov | 1_ Private | 1_ Self-emp-inc | ... | 13_ Portugal | 13_ Puerto-Rico |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 39 | 77516 | 13 | 2174 | 0 | 40 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **1** | 50 | 83311 | 13 | 0 | 0 | 13 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **2** | 38 | 215646 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **3** | 53 | 234721 | 7 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **4** | 28 | 338409 | 13 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **32556** | 27 | 257302 | 12 | 0 | 0 | 38 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32557** | 40 | 154374 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32558** | 58 | 151910 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32559** | 22 | 201490 | 9 | 0 | 0 | 20 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32560** | 52 | 287927 | 9 | 15024 | 0 | 40 | 0 | 0 | 0 | 1 | ... | 0 | 0 |

30162 rows × 104 columns

In [6]:

```
Y = pd.get_dummies(Y,
prefix=None,
prefix_sep="_",
dummy_na=False,
columns = None,
sparse=False,
drop_first=True,
dtype=None)
```

In [7]:

```
k = 10
accuracy = 0
size = X.shape[0]//k
for i in range(k):
    TEST = X[i*size:(i+1)*size]
    TEST_Y = Y[i*size:(i+1)*size].to_numpy()
    TRAIN = pd.concat([X[0:i*size],X[(i+1)*size:]])
    TRAIN_Y = pd.concat([Y[0:i*size],Y[(i+1)*size:]])
    clf = RandomForestClassifier(max_depth=None, random_state=0, criterion = "entrop
    clf = clf.fit(TRAIN, TRAIN_Y.to_numpy().reshape((TRAIN_Y.shape[0],)))
    output = clf.predict(TEST)
    correct = 0
    for i in range(TEST.shape[0]):
        if(output[i] == TEST_Y[i]): correct += 1
    accuracy += ((correct/TEST.shape[0])/k)
print("FINAL AVERAGE ACCURACY AFTER",k,"-fold cross validation =",accuracy)
```

```
FINAL AVERAGE ACCURACY AFTER 10 -fold cross validation = 0.85049734748
01059
```

FINAL AVERAGE ACCURACY AFTER 10 -fold cross validation = 0.8504973474801059

# Naive Bayes Classifier ¶

In [1]:

```
import numpy as np
import pandas as pd
from sklearn.naive_bayes import MultinomialNB
```

In [2]:

```
TRAIN_ADDRESS = "Train.csv"
df = pd.read_csv(TRAIN_ADDRESS, header = None)
X = pd.DataFrame(df, columns = df.columns, index = [i for i in range(df.shape[0])])
Y = pd.DataFrame(df, columns = [df.columns[-1]], index = [i for i in range(df.shape[
X = X[X.columns[:-1]]
```

In [3]:

```
checkX = []
checkY = []
for ind, rows in X.iterrows():
    if (" ?" in list(rows)):
        checkX.append(X.index[ind])
        checkY.append(Y.index[ind])
X = X.drop(checkX)
Y = Y.drop(checkY)
```

One Hot Encoding Nominal Attributes

In [4]:

```
X = pd.get_dummies(X,
prefix=None,
prefix_sep="_",
dummy_na=False,
columns=[1,3,5,6,7,8,9,13],
sparse=False,
drop_first=False,
dtype=None)
```

In [5]:

```
X
```

Out[5]:

| | 0 | 2 | 4 | 10 | 11 | 12 | 1_Federal-gov | 1_Local-gov | 1_Private | 1_Self-emp-inc | ... | 13_Portugal | 13_Puerto-Rico |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 39 | 77516 | 13 | 2174 | 0 | 40 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **1** | 50 | 83311 | 13 | 0 | 0 | 13 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **2** | 38 | 215646 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **3** | 53 | 234721 | 7 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **4** | 28 | 338409 | 13 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **32556** | 27 | 257302 | 12 | 0 | 0 | 38 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32557** | 40 | 154374 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32558** | 58 | 151910 | 9 | 0 | 0 | 40 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32559** | 22 | 201490 | 9 | 0 | 0 | 20 | 0 | 0 | 1 | 0 | ... | 0 | 0 |
| **32560** | 52 | 287927 | 9 | 15024 | 0 | 40 | 0 | 0 | 0 | 1 | ... | 0 | 0 |

30162 rows × 104 columns

In [6]:

```
Y = pd.get_dummies(Y,
prefix=None,
prefix_sep="_",
dummy_na=False,
columns = None,
sparse=False,
drop_first=True,
dtype=None)
```

In [7]:

```
k = 10
accuracy = 0
size = X.shape[0]//k
for i in range(k):
    TEST = X[i*size:(i+1)*size]
    TEST_Y = Y[i*size:(i+1)*size].to_numpy()
    TRAIN = pd.concat([X[0:i*size],X[(i+1)*size:]])
    TRAIN_Y = pd.concat([Y[0:i*size],Y[(i+1)*size:]])
    clf = MultinomialNB()
    clf = clf.fit(TRAIN, TRAIN_Y.to_numpy().reshape((TRAIN_Y.shape[0],)))
    output = clf.predict(TEST)
    correct = 0
    for i in range(TEST.shape[0]):
        if(output[i] == TEST_Y[i]): correct += 1
    accuracy += ((correct/TEST.shape[0])/k)
print("FINAL AVERAGE ACCURACY AFTER",k,"-fold cross validation =",accuracy)
```

```
FINAL AVERAGE ACCURACY AFTER 10 -fold cross validation = 0.77616047745
35809
```

FINAL AVERAGE ACCURACY AFTER 10 -fold cross validation = 0.7761604774535809

In [ ]:

```

```

## Q2 Part (a)

### *Apriori:*

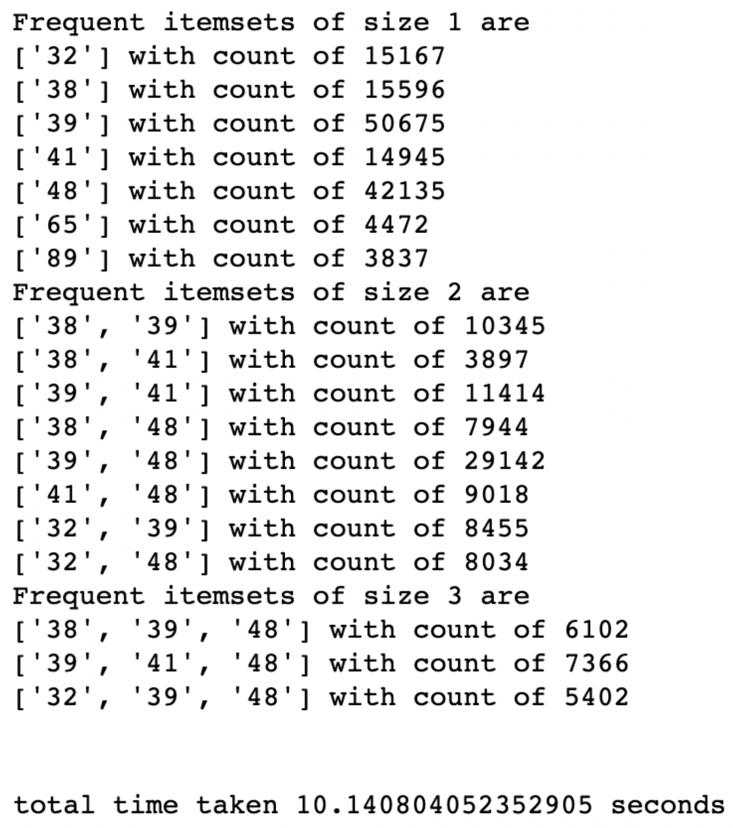
**References:**

1. https://www.geeksforgeeks.org/apriori-algorithm/ was a useful resource to gain conceptual clarity that helped in the development of the code.

***Apriori rudimentary version:***

**Performance:**

In this model if we set the minimum support for frequent itemsets as 0.04 then we get the following output in about 10 seconds:



```
Frequent itemsets of size 1 are
['32'] with count of 15167
['38'] with count of 15596
['39'] with count of 50675
['41'] with count of 14945
['48'] with count of 42135
['65'] with count of 4472
['89'] with count of 3837
Frequent itemsets of size 2 are
['38', '39'] with count of 10345
['38', '41'] with count of 3897
['39', '41'] with count of 11414
['38', '48'] with count of 7944
['39', '48'] with count of 29142
['41', '48'] with count of 9018
['32', '39'] with count of 8455
['32', '48'] with count of 8034
Frequent itemsets of size 3 are
['38', '39', '48'] with count of 6102
['39', '41', '48'] with count of 7366
['32', '39', '48'] with count of 5402


total time taken 10.140804052352905 seconds
```

```
Association rule ['38'] -> ['39'] with confidence 0.6633111054116441
Association rule ['41'] -> ['39'] with confidence 0.7637336901973905
Association rule ['38'] -> ['48'] with confidence 0.5093613747114645
Association rule ['39'] -> ['48'] with confidence 0.5750764676862358
Association rule ['48'] -> ['39'] with confidence 0.6916340334638661
Association rule ['41'] -> ['48'] with confidence 0.603412512546002
Association rule ['32'] -> ['39'] with confidence 0.5574602755983386
Association rule ['32'] -> ['48'] with confidence 0.5297026438979363
Association rule ['38'] -> ['39', '48'] with confidence 0.39125416773531674
Association rule ['38', '39'] -> ['48'] with confidence 0.5898501691638472
Association rule ['38', '48'] -> ['39'] with confidence 0.7681268882175226
Association rule ['41'] -> ['39', '48'] with confidence 0.4928738708598193
Association rule ['39', '41'] -> ['48'] with confidence 0.6453478184685474
Association rule ['41', '48'] -> ['39'] with confidence 0.8168108227988468
Association rule ['32'] -> ['39', '48'] with confidence 0.35616799630777346
Association rule ['32', '39'] -> ['48'] with confidence 0.6389118864577173
Association rule ['32', '48'] -> ['39'] with confidence 0.6723923325865073
```

In [244]:

```
import math
import time
import copy
```

In [245]:

```
#start timer
start_time = time.time()
```

In [246]:

```
#read from dat file
with open(r"retail.dat") as datFile:
    lines=[data.strip().split() for data in datFile]

transactions=[['T'+str(i),lines[i]] for i in range(len(lines))]
```

In [247]:

```
#total number of transactions
print('total number of transactions are',len(transactions))
```

```
total number of transactions are 88162
```

In [248]:

```
#number of items
items=set()
for itemset in transactions:
  for item in itemset[1]:
    items.add(item)
items=list(items)
print('total number of distinct items',len(items))
```

```
total number of distinct items 16470
```

In [249]:

```
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 0.5158648490905762 seconds ---
```

In [250]:

```
#support
support=0.04 #4%
#required minimum support count for selection
supcount=math.ceil(support*len(transactions))
print(supcount)
```

```
3527
```

In [251]:

```
#in case direct support count needs to be constrained instead of support
#supcount=3000
```

In [252]:

```
#generating 1 item frequentsets
L=[]
L1=[]
dic={}
counts=[]
c=[]
for itemset in transactions:
  for x in itemset[1]:
    if x in dic.keys():
        dic[x]+=1
    else:
        dic[x]=1
for k,v in dic.items():
  if v>=supcount:
    L1.append([k])
    c.append(v)
L.append(L1)
counts.append(c)
print(L1)
```

```
[['32'], ['38'], ['39'], ['41'], ['48'], ['65'], ['89']]
```

In [253]:

```
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 0.7243030071258545 seconds ---
```

In [254]:

```
#Cdash=transactions[:]
Cdash=copy.deepcopy(transactions)
```

In [255]:

```

for x in Cdash:
  lis=[[t] for t in x[1]]
  x[1]=lis

```

In [256]:

```python
def gen_candidate(L):
  #L of the form [[1,2,3],[2,3,4],[1,2,4]]

  #number of elements in candidates of round
  k=len(L[0])+1

  C=[]
  for i in range(len(L)):
    for j in range(i+1,len(L)):

      possible_candidate=list( set(L[i]) | set(L[j]) )
      possible_candidate.sort()

      if len(possible_candidate)!=k:
        continue
      elif possible_candidate in C:
        continue
      else:
        true_candidate=True

        for x in possible_candidate:
          test_candidate=copy.deepcopy(possible_candidate)
          test_candidate.remove(x)

          if test_candidate not in L:
            true_candidate=False
            break

        if true_candidate:
          C.append(possible_candidate)

  return C
```

In [257]:

```
for k in range(len(items)+2):
  if len(L[k])==0:
    L.remove([])
    break
  else:
    lis=[]
    C=gen_candidate(L[k])

    #print(C)
    dic={}
    Cdash2=copy.deepcopy(Cdash)

    for itemset1 in Cdash2:
      itemset1[1]=[]

    for i in range(len(Cdash)):
    #for i in range(1):
      itemset=Cdash[i]
      #print(itemset)
      for candidate in C:
        willbe=True
        dummy=copy.deepcopy(candidate)
        #print(dummy)
        dummy.remove(dummy[0])
        #print(dummy)
        if dummy not in itemset[1]:
          willbe=False
        if willbe:
          dummy=copy.deepcopy(candidate)
          dummy.remove(dummy[1])
          if dummy not in itemset[1]:
            willbe=False
        if willbe:
          Cdash2[i][1].append(candidate)
          candi=tuple(candidate)
          if candi in dic.keys():
            dic[candi]+=1
          else:
            dic[candi]=1

    #print(Cdash2)
    Cdash3=[]
    for itemset in Cdash2:
      if itemset[1]!=[]:
        Cdash3.append(itemset)
    Cdash=Cdash3
    c=[]
    for k,v in dic.items():
      if v>=supcount:
        a=list(k)
        a.sort()
        lis.append(a)
        c.append(v)
    counts.append(c)
    L.append(lis)

    #L has candidates in Ck+1 with support ≥min_sup
for i in range(len(L)):
  print('Frequent itemsets of size',i+1,'are')
  for x in range(len(L[i])):
    print(L[i][x],'with count of',counts[i][x])
```

```
Frequent itemsets of size 1 are
['32'] with count of 15167
['38'] with count of 15596
['39'] with count of 50675
['41'] with count of 14945
['48'] with count of 42135
['65'] with count of 4472
['89'] with count of 3837
Frequent itemsets of size 2 are
['38', '39'] with count of 10345
['38', '41'] with count of 3897
['39', '41'] with count of 11414
['38', '48'] with count of 7944
['39', '48'] with count of 29142
['41', '48'] with count of 9018
['32', '39'] with count of 8455
['32', '48'] with count of 8034
Frequent itemsets of size 3 are
['38', '39', '48'] with count of 6102
['39', '41', '48'] with count of 7366
['32', '39', '48'] with count of 5402
```

In [258]:

```
print("--- total time taken %s seconds ---" % (time.time() - start_time))
```

```
--- total time taken 10.472259998321533 seconds ---
```

In [ ]:

## Q2 Part (a)

### *FP Tree:*

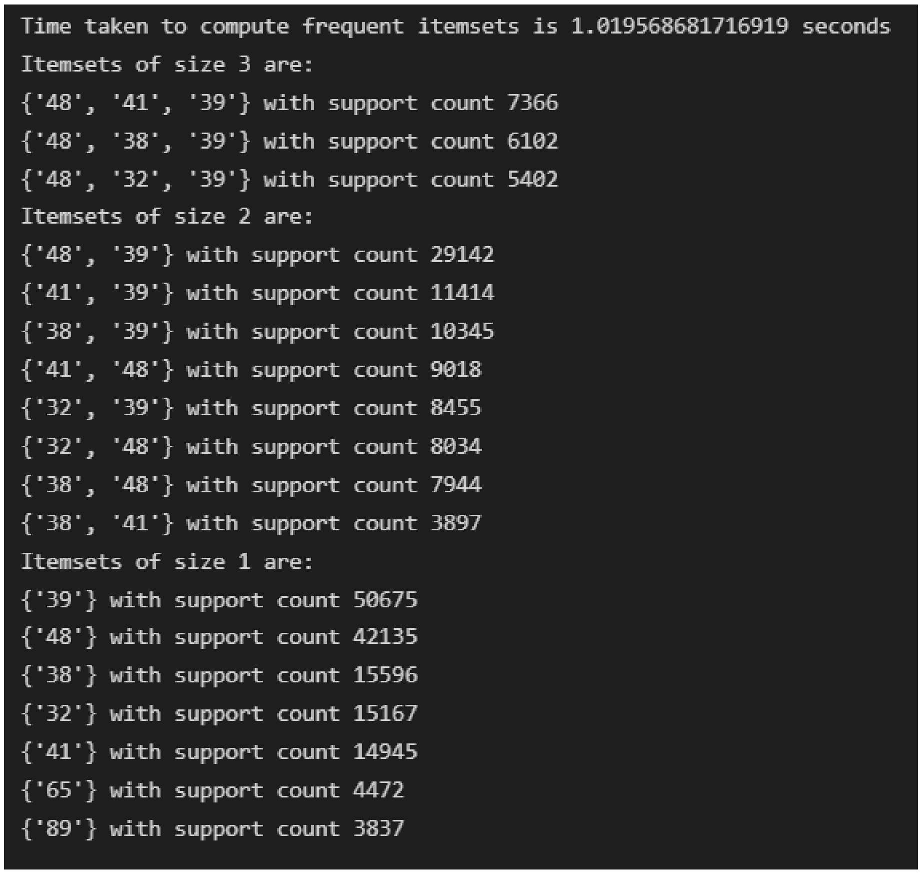
**References:**

1. Frequent Pattern (FP) Growth Algorithm In Data Mining (softwaretestinghelp.com)
2. Introduction Guide To FP-Tree Algorithm (analyticsindiamag.com)
3. ML | Frequent Pattern Growth Algorithm - GeeksforGeeks

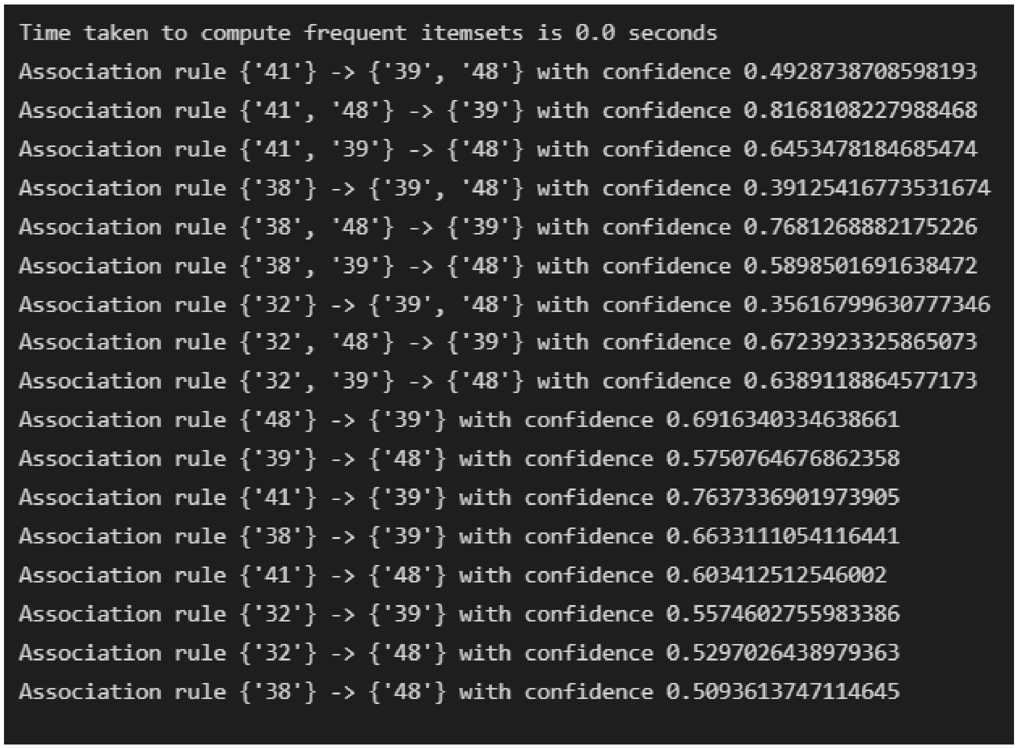
***FP Tree rudimentary version:***

**Performance:**

In this model if we set the minimum support for frequent itemsets as 0.04 then we get the following output in about 1.02 seconds:



```
Time taken to compute frequent itemsets is 1.019568681716919 seconds
Itemsets of size 3 are:
{'48', '41', '39'} with support count 7366
{'48', '38', '39'} with support count 6102
{'48', '32', '39'} with support count 5402
Itemsets of size 2 are:
{'48', '39'} with support count 29142
{'41', '39'} with support count 11414
{'38', '39'} with support count 10345
{'41', '48'} with support count 9018
{'32', '39'} with support count 8455
{'32', '48'} with support count 8034
{'38', '48'} with support count 7944
{'38', '41'} with support count 3897
Itemsets of size 1 are:
{'39'} with support count 50675
{'48'} with support count 42135
{'38'} with support count 15596
{'32'} with support count 15167
{'41'} with support count 14945
{'65'} with support count 4472
{'89'} with support count 3837
```



```
Time taken to compute frequent itemsets is 0.0 seconds
Association rule {'41'} -> {'39', '48'} with confidence 0.4928738708598193
Association rule {'41', '48'} -> {'39'} with confidence 0.8168108227988468
Association rule {'41', '39'} -> {'48'} with confidence 0.6453478184685474
Association rule {'38'} -> {'39', '48'} with confidence 0.39125416773531674
Association rule {'38', '48'} -> {'39'} with confidence 0.7681268882175226
Association rule {'38', '39'} -> {'48'} with confidence 0.5898501691638472
Association rule {'32'} -> {'39', '48'} with confidence 0.35616799630777346
Association rule {'32', '48'} -> {'39'} with confidence 0.6723923325865073
Association rule {'32', '39'} -> {'48'} with confidence 0.6389118864577173
Association rule {'48'} -> {'39'} with confidence 0.6916340334638661
Association rule {'39'} -> {'48'} with confidence 0.5750764676862358
Association rule {'41'} -> {'39'} with confidence 0.7637336901973905
Association rule {'38'} -> {'39'} with confidence 0.6633111054116441
Association rule {'41'} -> {'48'} with confidence 0.603412512546002
Association rule {'32'} -> {'39'} with confidence 0.5574602755983386
Association rule {'32'} -> {'48'} with confidence 0.5297026438979363
Association rule {'38'} -> {'48'} with confidence 0.5093613747114645
```

In [1]:

```
from itertools import combinations
import time
```

In [2]:

```
f = open("retail.dat", 'r')
lines = f.readlines()
data = []
for line in lines:
    data.append(line.split())
```

In [3]:

```
class Node:
    def __init__(self, item, item_count=0, next_node = None, parent=None):
        self.item=item
        self.item_count=item_count
        self.parent=parent
        self.children={}
        self.next_node = next_node
```

In [4]:

```
class fptree:
    def __init__(self, data, minsup):
        self.data=data
        self.minsup=minsup
        self.root= Node(item="Null", item_count=1)
        self.sup_items={}
        self.header_table = {}
        self.make_tree(data)


    def make_tree(self, data):
        if data is None: return None

        for transaction in data:
            for item in transaction:
                if item in self.sup_items.keys():
                    self.sup_items[item]+=1
                else:
                    self.sup_items[item]=1

        for item in self.sup_items.copy().keys():
            if(self.sup_items[item]<self.minsup):
                del self.sup_items[item]
            else: self.header_table[item]=None

        self.sup_items = dict(sorted(self.sup_items.items(), key=lambda x: (-x[1],x[

        for transaction in data:
            trans_minsup_items=[]
            for item in transaction:
                if item in self.sup_items.keys():
                    trans_minsup_items.append(item)

            if len(trans_minsup_items)==0:pass

            trans_minsup_items = sorted(trans_minsup_items, key = lambda k: self.sup
            curr_node = self.root

            for item in trans_minsup_items:
                if item in curr_node.children.keys():
                    curr_node.children[item].item_count +=1
                    curr_node=curr_node.children[item]
                else:
                    curr_node.children[item] = Node(item=item,item_count=1, parent=c
                    curr_node=curr_node.children[item]

                    temp_node = self.header_table[item]
                    if temp_node is None:
                        self.header_table[item] = curr_node
                    else:
                        while temp_node.next_node is not None:
                            temp_node = temp_node.next_node
                        temp_node.next_node = curr_node

    def make_cond_tree(self, item):
        if self.header_table[item] is None:
            return None

        cond_trans_data=[]
        adj_node = self.header_table[item]
        while adj_node is not None:
            cond_trans=[]
            parent_node = adj_node.parent
            while parent_node.parent is not None:
                cond_trans.append(parent_node.item)
                parent_node=parent_node.parent
            cond_trans = cond_trans[::-1]
            for i in range(adj_node.item_count):
                cond_trans_data.append(cond_trans)
            adj_node = adj_node.next_node
        return fptree(cond_trans_data, self.minsup)

    def gen_freq_set(self, parentset=None):
        if len(self.sup_items)==0:
            return None
        freqs=[]
        rev_items = list(self.sup_items.keys())[::-1]
        for item in rev_items:
            freq_set=[set(), 0]
            if(parentset==None):
                freq_set[0]={item,}
            else:
                freq_set[0]= {item}.union(parentset[0])
            freq_set[1] = self.sup_items[item]
            freqs.append(freq_set)
            item_cond_tree= self.make_cond_tree(item)
            cond_items = item_cond_tree.gen_freq_set(freq_set)
            if cond_items is not None:
                for items in cond_items:
                    freqs.append(items)
        return freqs
```

In [5]:

```
min_sup_ratio = 0.04
min_cond_ratio = 0.3
min_sup= min_sup_ratio * len(data)

start_time_pat = time.time()

fp_tree = fptree(data, min_sup)
frequentitemset = fp_tree.gen_freq_set()

frequentitemset=sorted(frequentitemset,key = lambda k: -k[1] )
frequentitemset = sorted(frequentitemset, key = lambda x: len(x[0]), reverse=True)

end_time_pat = time.time()

print("Time taken to compute frequent itemsets is %s seconds" % (end_time_pat - star

curr_set_size = len(frequentitemset[0][0])
print("Itemsets of size", curr_set_size, "are: ")
for item in frequentitemset:
    if curr_set_size > len(item[0]):
        curr_set_size = len(item[0])
        print("Itemsets of size", curr_set_size, "are: ")
    print(item[0], "with support count", item[1])
```

```
Time taken to compute frequent itemsets is 1.019568681716919 seconds
Itemsets of size 3 are:
{'48', '41', '39'} with support count 7366
{'48', '38', '39'} with support count 6102
{'48', '32', '39'} with support count 5402
Itemsets of size 2 are:
{'48', '39'} with support count 29142
{'41', '39'} with support count 11414
{'38', '39'} with support count 10345
{'41', '48'} with support count 9018
{'32', '39'} with support count 8455
{'32', '48'} with support count 8034
{'38', '48'} with support count 7944
{'38', '41'} with support count 3897
Itemsets of size 1 are:
{'39'} with support count 50675
{'48'} with support count 42135
{'38'} with support count 15596
{'32'} with support count 15167
{'41'} with support count 14945
{'65'} with support count 4472
{'89'} with support count 3837
```

In [6]:

```
start_time_assoc = time.time()

frequentdict = {}
for item in frequentitemset:
    frequentdict[tuple(sorted(item[0]))] = item[1]

assoc_rules = []

for item in frequentitemset:
  if len(item[0])<2: break
  s=item[0]
  s_tuple = tuple(sorted(s))
  for i in range(1,len(s)):
    a = list(combinations(s, i))
    for t in a:
      t_set=set(t)
      t_s_diff = s.difference(t_set)
      t_tuple = tuple(sorted(t))
      cond=frequentdict[s_tuple]/frequentdict[t_tuple]
      if cond>min_cond_ratio:
        assoc_rules.append([t_set, t_s_diff, cond])

end_time_assoc = time.time()

print("Time taken to compute frequent itemsets is %s seconds" % (end_time_assoc - st


for rule in assoc_rules:
          print('Association rule',rule[0],'->',rule[1],'with confidence',rule[2])
```

```
Time taken to compute frequent itemsets is 0.0 seconds
Association rule {'41'} -> {'39', '48'} with confidence 0.492873870859
8193
Association rule {'41', '48'} -> {'39'} with confidence 0.816810822798
8468
Association rule {'41', '39'} -> {'48'} with confidence 0.645347818468
5474
Association rule {'38'} -> {'39', '48'} with confidence 0.391254167735
31674
Association rule {'38', '48'} -> {'39'} with confidence 0.768126888217
5226
Association rule {'38', '39'} -> {'48'} with confidence 0.589850169163
8472
Association rule {'32'} -> {'39', '48'} with confidence 0.356167996307
77346
Association rule {'32', '48'} -> {'39'} with confidence 0.672392332586
5073
Association rule {'32', '39'} -> {'48'} with confidence 0.638911886457
7173
Association rule {'48'} -> {'39'} with confidence 0.6916340334638661
Association rule {'39'} -> {'48'} with confidence 0.5750764676862358
Association rule {'41'} -> {'39'} with confidence 0.7637336901973905
Association rule {'38'} -> {'39'} with confidence 0.6633111054116441
Association rule {'41'} -> {'48'} with confidence 0.603412512546002
Association rule {'32'} -> {'39'} with confidence 0.5574602755983386
Association rule {'32'} -> {'48'} with confidence 0.5297026438979363
Association rule {'38'} -> {'48'} with confidence 0.5093613747114645
```

In [ ]:

```
```

## Q2 Part (b)

### *Apriori:*

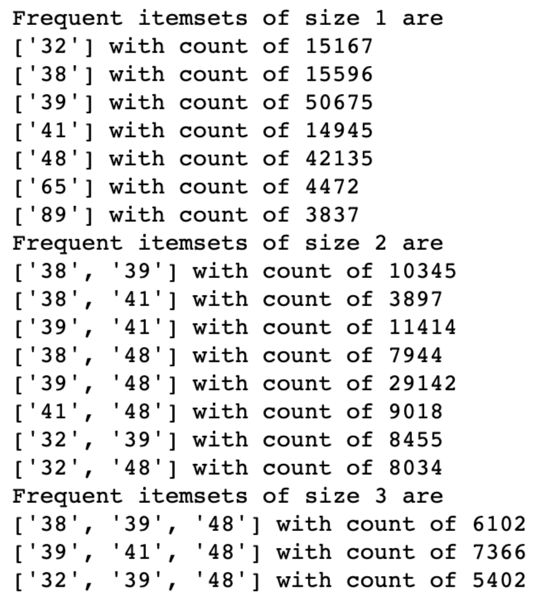
**References:**

1. https://www.geeksforgeeks.org/apriori-algorithm/ was a useful resource to gain conceptual clarity that helped in the development of the code.

***Apriori improved version:***

**Improvements made:**
According to the given dataset at http://fimi.ua.ac.be/data/retail.dat even at 1 percent support, the dataset was giving at most up to 4-item frequent itemsets. Due to such a low number, copying the dataset for every iteration (inspired by aprioriTID) seemed futile so that was removed from the original model thereby also saving memory space. This reinforces the importance of understanding the data in a data mining problem. Also items in the itemsets were hashed and candidates were searched in this hashset. This reflects the importance of data structures. This too improved the performance of the algorithm by a huge factor. Apart from this we also added the functionality of generating association rules from the frequent itemsets.

**Performance:**
In this model if we set the minimum support for frequent itemsets as 0.04 and the minimum confidence for association rules as 0.3 then we get the following output in about 2 seconds:



```
Frequent itemsets of size 1 are
['32'] with count of 15167
['38'] with count of 15596
['39'] with count of 50675
['41'] with count of 14945
['48'] with count of 42135
['65'] with count of 4472
['89'] with count of 3837
Frequent itemsets of size 2 are
['38', '39'] with count of 10345
['38', '41'] with count of 3897
['39', '41'] with count of 11414
['38', '48'] with count of 7944
['39', '48'] with count of 29142
['41', '48'] with count of 9018
['32', '39'] with count of 8455
['32', '48'] with count of 8034
Frequent itemsets of size 3 are
['38', '39', '48'] with count of 6102
['39', '41', '48'] with count of 7366
['32', '39', '48'] with count of 5402
```

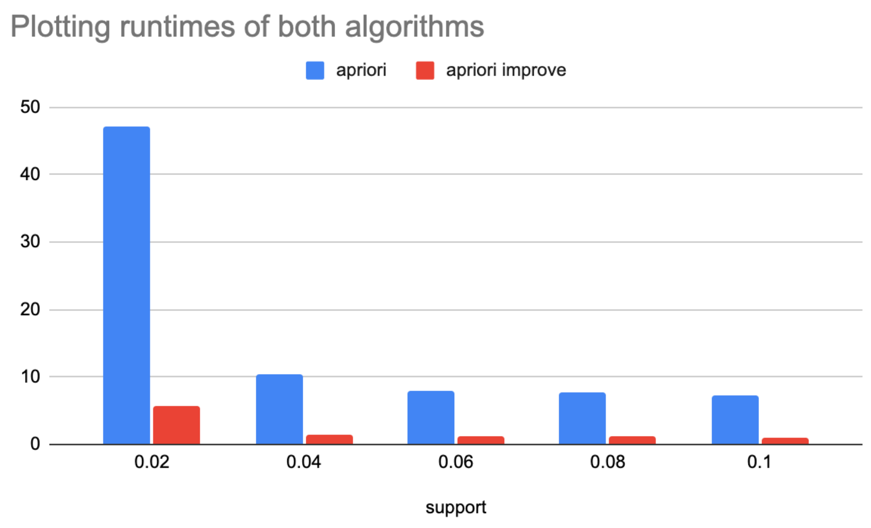
```
Association rule ['38'] -> ['39'] with confidence 0.6633111054116441
Association rule ['41'] -> ['39'] with confidence 0.7637336901973905
Association rule ['38'] -> ['48'] with confidence 0.5093613747114645
Association rule ['39'] -> ['48'] with confidence 0.5750764676862358
Association rule ['48'] -> ['39'] with confidence 0.6916340334638661
Association rule ['41'] -> ['48'] with confidence 0.603412512546002
Association rule ['32'] -> ['39'] with confidence 0.5574602755983386
Association rule ['32'] -> ['48'] with confidence 0.5297026438979363
Association rule ['38'] -> ['39', '48'] with confidence 0.39125416773531674
Association rule ['38', '39'] -> ['48'] with confidence 0.5898501691638472
Association rule ['38', '48'] -> ['39'] with confidence 0.7681268882175226
Association rule ['41'] -> ['39', '48'] with confidence 0.4928738708598193
Association rule ['39', '41'] -> ['48'] with confidence 0.6453478184685474
Association rule ['41', '48'] -> ['39'] with confidence 0.8168108227988468
Association rule ['32'] -> ['39', '48'] with confidence 0.35616799630777346
Association rule ['32', '39'] -> ['48'] with confidence 0.6389118864577173
Association rule ['32', '48'] -> ['39'] with confidence 0.6723923325865073

total time taken 1.465137004852295
```

***Increase in efficiency due to improvements:***

As we can see the change is quite significant, with about **85.6% reduction** in total time taken. So the modifications work well.

- A graph to compare both algorithms easily (time measured in seconds)

| support | apriori | apriori improve |
|---|---|---|
| 0.02 | 47.08041 | 5.789883852 |
| 0.04 | 10.34487987 | 1.489167929 |
| 0.06 | 7.974373102 | 1.225999928 |
| 0.08 | 7.816525221 | 1.154891968 |
| 0.1 | 7.378671646 | 1.042146921 |



In [265]:

```
import math
import time
import copy
from itertools import combinations
```

In [266]:

```
#start timer
start_time = time.time()
```

In [267]:

```
#read from dat file
with open(r"retail.dat") as datFile:
    lines=[data.strip().split() for data in datFile]

transactions=[['T'+str(i),lines[i]] for i in range(len(lines))]
```

In [268]:

```
#total number of transactions
print('total number of transactions are',len(transactions))
```

```
total number of transactions are 88162
```

In [269]:

```
#number of items
items=set()
for itemset in transactions:
  for item in itemset[1]:
    items.add(item)
items=list(items)
print('total number of distinct items',len(items))
#print(items)
```

```
total number of distinct items 16470
```

In [270]:

```
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 0.5476198196411133 seconds ---
```

In [271]:

```
#support
support=0.04 #4%
#required minimum support count for selection
supcount=math.ceil(support*len(transactions))
print('minimum support count is',supcount)
```

```
minimum support count is 3527
```

In [272]:

```
#in case direct support count needs to be constrained instead of support
#supcount=3000
```

In [273]:

```
#generating 1 item frequentsets
L=[]
L1=[]
D={}
dic={}
counts=[]
c=[]
for itemset in transactions:
  for x in itemset[1]:
    if x in dic.keys():
        dic[x]+=1
    else:
        dic[x]=1
for k,v in dic.items():
  if v>=supcount:
    L1.append([k])
    c.append(v)
    D[tuple([k])]=v
L.append(L1)
counts.append(c)
print('Frequent 1 item itemsets are as follows')
print(L1)
#print(counts)
#print(D)
```

```
Frequent 1 item itemsets are as follows
[['32'], ['38'], ['39'], ['41'], ['48'], ['65'], ['89']]
```

In [274]:

```
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 0.7474570274353027 seconds ---
```

In [275]:

```
#function to generate candidates C

def gen_candidate(L):
  #L of the form [[1,2,3],[2,3,4],[1,2,4]]

  #number of elements in candidates of round
  k=len(L[0])+1
  #candidate list
  C=[]
  for i in range(len(L)):
    for j in range(i+1,len(L)):

      #checking size of union
      possible_candidate=list( set(L[i]) | set(L[j]) )
      possible_candidate.sort()

      if len(possible_candidate)!=k:
        continue
      elif possible_candidate in C:
        continue
      else:
        true_candidate=True

        for x in possible_candidate:
          test_candidate=copy.deepcopy(possible_candidate)
          test_candidate.remove(x)

          if test_candidate not in L:
            true_candidate=False
            break

        if true_candidate:
          C.append(possible_candidate)

  return C
```

In [276]:

```
#loop for generating all itemsets
for k in range(len(items)+2):
  if len(L[k])==0:
    L.remove([])
    break
  else:
    lis=[]
    C=gen_candidate(L[k])
    #print(C)
    dic={}
    for itemset in transactions:
      supm=set(itemset[1])
      for candidate in C:
        #candi=tuple(candidate)
        candi=set(candidate)
        result =  candi.issubset(supm)
        #result =  all(e in itemset[1] for e in candidate)

        if result==True:
          candi=tuple(candidate)
          if candi in dic.keys():
            dic[candi]+=1
          else:
            dic[candi]=1
    c=[]
    for k,v in dic.items():
      if v>=supcount:
        a=list(k)
        a.sort()
        lis.append(a)
        c.append(v)
        D[tuple(k)]=v
    L.append(lis)
    counts.append(c)

for i in range(len(L)):
  print('Frequent itemsets of size',i+1,'are')
  for x in range(len(L[i])):
    print(L[i][x],'with count of',counts[i][x])
```

```
Frequent itemsets of size 1 are
['32'] with count of 15167
['38'] with count of 15596
['39'] with count of 50675
['41'] with count of 14945
['48'] with count of 42135
['65'] with count of 4472
['89'] with count of 3837
Frequent itemsets of size 2 are
['38', '39'] with count of 10345
['38', '41'] with count of 3897
['39', '41'] with count of 11414
['38', '48'] with count of 7944
['39', '48'] with count of 29142
['41', '48'] with count of 9018
['32', '39'] with count of 8455
['32', '48'] with count of 8034
Frequent itemsets of size 3 are
```

```
['38', '39', '48'] with count of 6102
['39', '41', '48'] with count of 7366
['32', '39', '48'] with count of 5402
```

In [277]:

```
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 1.5522758960723877 seconds ---
```

In [278]:

```
confidence=0.3
```

In [279]:

```
for lis in L:
  for itemset in lis:
    #print('itemset',itemset)
    s=set(itemset)
    #print(tuple(s))
    num=D[tuple(sorted(tuple(s)))]
    #print(num)
    for i in range(1,len(itemset)):
      a = list(combinations(itemset, i))
      a=[list(x) for x in a]
      #print(a)
      for t in a:
        b=set(t)
        c=s.difference(b)
        c=list(c)
        #print('combo',t,c)
        para=num/D[tuple(sorted(tuple(t)))]
        if para>confidence:
          print('Association rule',t,'->',c,'with confidence',para)
```

```
Association rule ['38'] -> ['39'] with confidence 0.6633111054116441
Association rule ['41'] -> ['39'] with confidence 0.7637336901973905
Association rule ['38'] -> ['48'] with confidence 0.5093613747114645
Association rule ['39'] -> ['48'] with confidence 0.5750764676862358
Association rule ['48'] -> ['39'] with confidence 0.6916340334638661
Association rule ['41'] -> ['48'] with confidence 0.603412512546002
Association rule ['32'] -> ['39'] with confidence 0.5574602755983386
Association rule ['32'] -> ['48'] with confidence 0.5297026438979363
Association rule ['38'] -> ['39', '48'] with confidence 0.391254167735
31674
Association rule ['38', '39'] -> ['48'] with confidence 0.589850169163
8472
Association rule ['38', '48'] -> ['39'] with confidence 0.768126888217
5226
Association rule ['41'] -> ['39', '48'] with confidence 0.492873870859
8193
Association rule ['39', '41'] -> ['48'] with confidence 0.645347818468
5474
Association rule ['41', '48'] -> ['39'] with confidence 0.816810822798
8468
Association rule ['32'] -> ['39', '48'] with confidence 0.356167996307
77346
Association rule ['32', '39'] -> ['48'] with confidence 0.638911886457
7173
Association rule ['32', '48'] -> ['39'] with confidence 0.672392332586
5073
```

In [280]:

```
print("--- total time taken %s seconds ---" % (time.time() - start_time))
```

```
--- total time taken 1.561579942703247 seconds ---
```

In [ ]:

```
```

## Q2 Part (b)

### *FP Tree:*

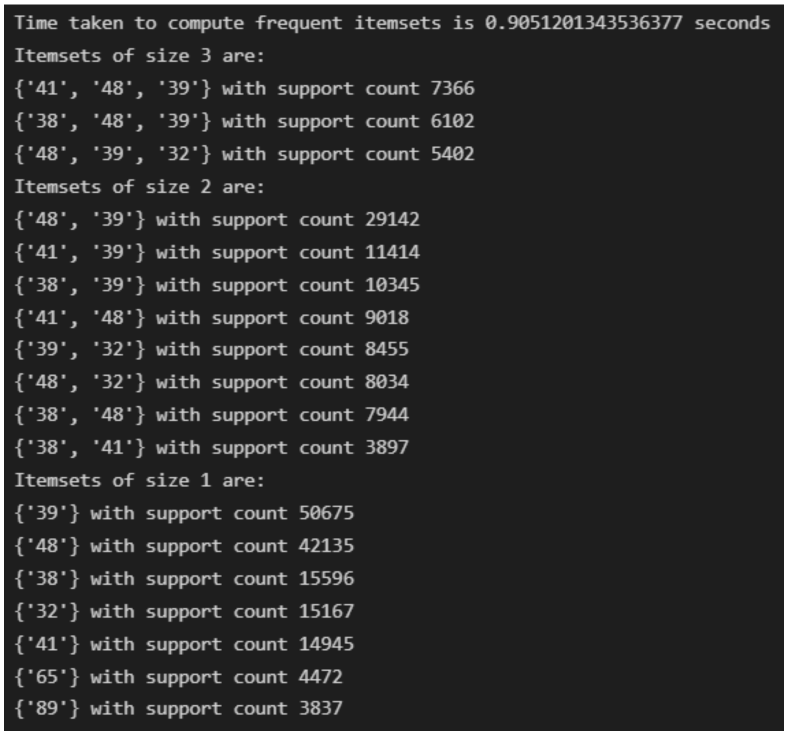
**References:**

1. [Frequent Pattern (FP) Growth Algorithm In Data Mining (softwaretestinghelp.com)]
2. [Introduction Guide To FP-Tree Algorithm (analyticsindiamag.com)]
3. [ML | Frequent Pattern Growth Algorithm - GeeksforGeeks]

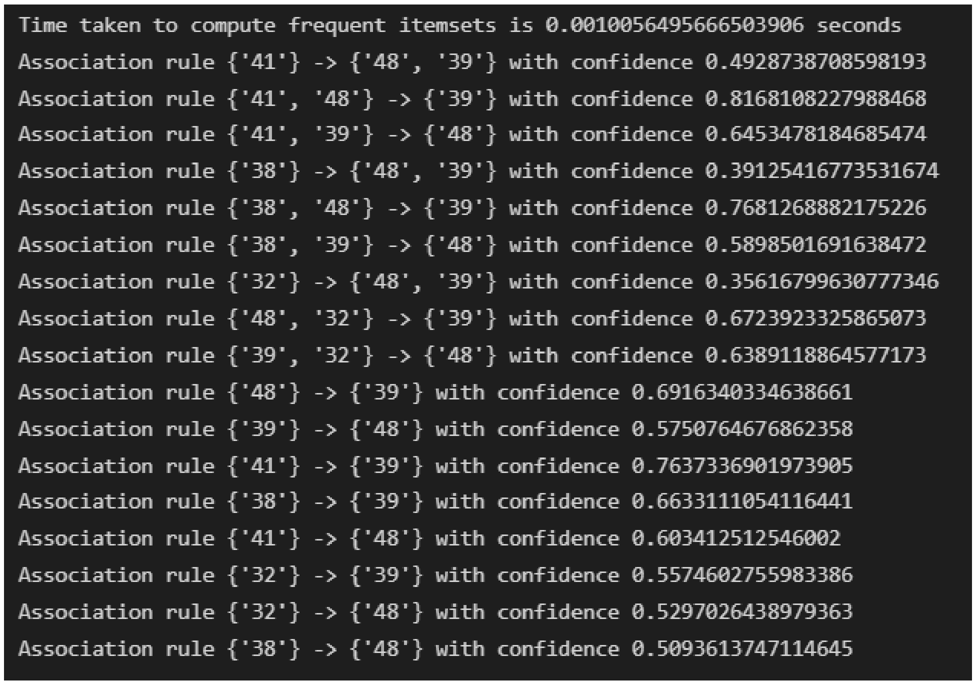
***FP Tree improved version:***

**Improvements made:**
In the original FP tree, we have linked one node to another node, and the starting node is stored in a header table. Due to this, whenever a new node has to be added, we need to traverse to the end of the list of nodes starting from the header node each time. Thus, this can be improved. We have improved this by storing all the nodes of a given item in an adjacency list. Due to this, we can simply append the nodes which are created to the end of the list, reducing the time complexity of the operation from O(n) to O(1), where n is the number of nodes corresponding to the item.

**Performance:**
In this model if we set the minimum support for frequent itemsets as 0.04 and the minimum confidence for association rules as 0.3 then we get the following output in about 0.9 seconds:



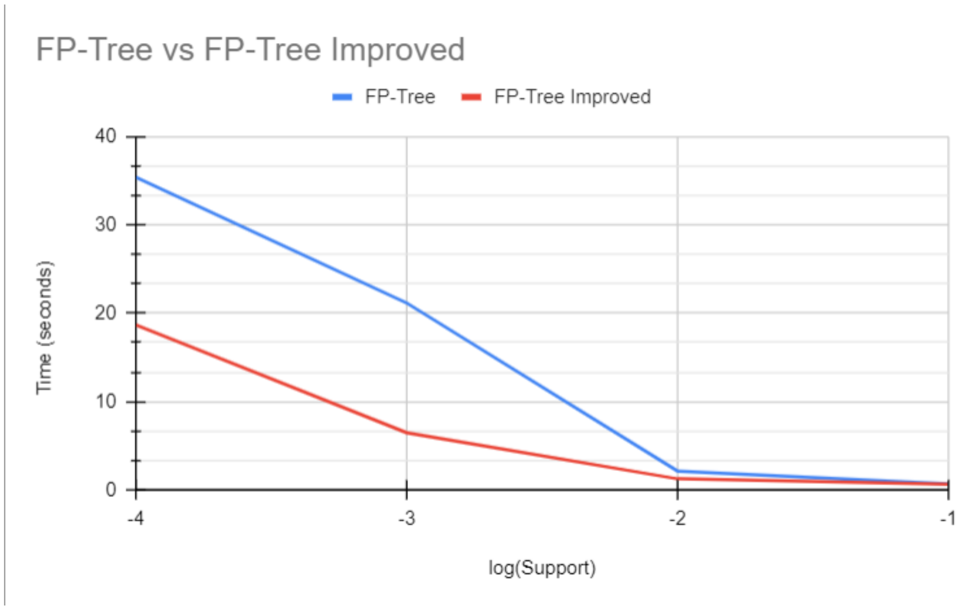
```
Time taken to compute frequent itemsets is 0.9051201343536377 seconds
Itemsets of size 3 are:
{'41', '48', '39'} with support count 7366
{'38', '48', '39'} with support count 6102
{'48', '39', '32'} with support count 5402
Itemsets of size 2 are:
{'48', '39'} with support count 29142
{'41', '39'} with support count 11414
{'38', '39'} with support count 10345
{'41', '48'} with support count 9018
{'39', '32'} with support count 8455
{'48', '32'} with support count 8034
{'38', '48'} with support count 7944
{'38', '41'} with support count 3897
Itemsets of size 1 are:
{'39'} with support count 50675
{'48'} with support count 42135
{'38'} with support count 15596
{'32'} with support count 15167
{'41'} with support count 14945
{'65'} with support count 4472
{'89'} with support count 3837
```

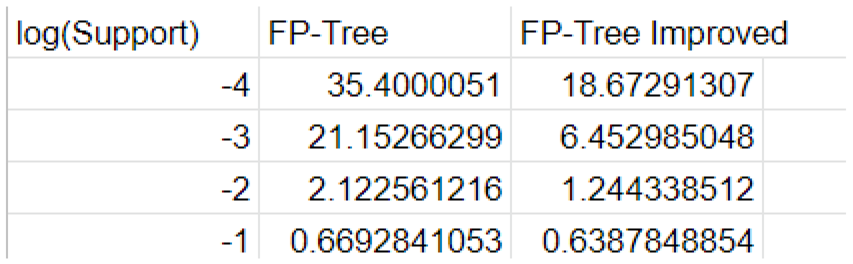


```
Time taken to compute frequent itemsets is 0.0010056495666503906 seconds
Association rule {'41'} -> {'48', '39'} with confidence 0.4928738708598193
Association rule {'41', '48'} -> {'39'} with confidence 0.8168108227988468
Association rule {'41', '39'} -> {'48'} with confidence 0.6453478184685474
Association rule {'38'} -> {'48', '39'} with confidence 0.39125416773531674
Association rule {'38', '48'} -> {'39'} with confidence 0.7681268882175226
Association rule {'38', '39'} -> {'48'} with confidence 0.5898501691638472
Association rule {'32'} -> {'48', '39'} with confidence 0.35616799630777346
Association rule {'48', '32'} -> {'39'} with confidence 0.6723923325865073
Association rule {'39', '32'} -> {'48'} with confidence 0.6389118864577173
Association rule {'48'} -> {'39'} with confidence 0.6916340334638661
Association rule {'39'} -> {'48'} with confidence 0.5750764676862358
Association rule {'41'} -> {'39'} with confidence 0.7637336901973905
Association rule {'38'} -> {'39'} with confidence 0.6633111054116441
Association rule {'41'} -> {'48'} with confidence 0.603412512546002
Association rule {'32'} -> {'39'} with confidence 0.5574602755983386
Association rule {'32'} -> {'48'} with confidence 0.5297026438979363
Association rule {'38'} -> {'48'} with confidence 0.5093613747114645
```

***Increase in efficiency due to improvements:***

As we can see the change is quite significant, especially for lower support values we are able to get much faster times with the improved version using adjacency list.





| log(Support) | FP-Tree | FP-Tree Improved |
|---|---|---|
| -4 | 35.4000051 | 18.67291307 |
| -3 | 21.15266299 | 6.452985048 |
| -2 | 2.122561216 | 1.244338512 |
| -1 | 0.6692841053 | 0.6387848854 |

In [1]:

```
import time
from itertools import combinations
```

In [2]:

```
f = open("retail.dat", 'r')
lines = f.readlines()
data = []
for line in lines:
    data.append(line.split())
```

In [3]:

```
class Node:
    def __init__(self, item, item_count=0, parent=None):
        self.item=item
        self.item_count=item_count
        self.parent=parent
        self.children={}
```

In [4]:

```
class fptree:
    def __init__(self, data, minsup):
        self.data=data
        self.minsup=minsup

        self.root= Node(item="Null", item_count=1)
        self.sup_items={}
        self.adj_link={}

        self.make_tree(data)


    def make_tree(self, data):
        if data is None: return None

        for transaction in data:
            for item in transaction:
                if item in self.sup_items.keys():
                    self.sup_items[item]+=1
                else:
                    self.sup_items[item]=1

        for item in self.sup_items.copy().keys():
            if(self.sup_items[item]<self.minsup):
                del self.sup_items[item]
            else: self.adj_link[item]=[]

        self.sup_items = dict(sorted(self.sup_items.items(), key=lambda x: (-x[1],x[

        for transaction in data:
            trans_minsup_items=[]
            for item in transaction:
                if item in self.sup_items.keys():
                    trans_minsup_items.append(item)

            if len(trans_minsup_items)==0:pass

            trans_minsup_items = sorted(trans_minsup_items, key = lambda k: self.sup
            curr_node = self.root

            for item in trans_minsup_items:
                if item in curr_node.children.keys():
                    curr_node.children[item].item_count +=1
                    curr_node=curr_node.children[item]
                else:
                    curr_node.children[item] = Node(item=item,item_count=1,parent=cu
                    curr_node=curr_node.children[item]
                    self.adj_link[item].append(curr_node)


    def make_cond_tree(self, item):
        if len(self.adj_link[item])==0:
            return None

        cond_trans_data=[]
        for adj_node in self.adj_link[item]:
            cond_trans=[]
            parent_node = adj_node.parent
            while parent_node.parent is not None:
                cond_trans.append(parent_node.item)
                parent_node=parent_node.parent
            cond_trans = cond_trans[::-1]
            for i in range(adj_node.item_count):
                cond_trans_data.append(cond_trans)
        return fptree(cond_trans_data, self.minsup)

    def gen_freq_set(self, parentset=None):
        if len(self.sup_items)==0:
            return None
        freqs=[]
        rev_items = list(self.sup_items.keys())[::-1]
        for item in rev_items:
            freq_set=[set(), 0]
            if(parentset==None):
                freq_set[0]={item,}
            else:
                freq_set[0]= {item}.union(parentset[0])
            freq_set[1] = self.sup_items[item]
            freqs.append(freq_set)
            item_cond_tree= self.make_cond_tree(item)
            cond_items = item_cond_tree.gen_freq_set(freq_set)
            if cond_items is not None:
                for items in cond_items:
                    freqs.append(items)
        return freqs
```

In [5]:

```
min_sup_ratio = 0.04
min_cond_ratio = 0.3
min_sup= min_sup_ratio * len(data)

start_time_pat = time.time()

fp_tree = fptree(data, min_sup)
frequentitemset = fp_tree.gen_freq_set()

frequentitemset=sorted(frequentitemset,key = lambda k: -k[1] )
frequentitemset = sorted(frequentitemset, key = lambda x: len(x[0]), reverse=True)

end_time_pat = time.time()

print("Time taken to compute frequent itemsets is %s seconds" % (end_time_pat - star

curr_set_size = len(frequentitemset[0][0])
print("Itemsets of size", curr_set_size, "are: ")
for item in frequentitemset:
    if curr_set_size > len(item[0]):
        curr_set_size = len(item[0])
        print("Itemsets of size", curr_set_size, "are: ")
    print(item[0], "with support count", item[1])
```

```
Time taken to compute frequent itemsets is 0.9051201343536377 seconds
Itemsets of size 3 are:
{'41', '48', '39'} with support count 7366
{'38', '48', '39'} with support count 6102
{'48', '39', '32'} with support count 5402
Itemsets of size 2 are:
{'48', '39'} with support count 29142
{'41', '39'} with support count 11414
{'38', '39'} with support count 10345
{'41', '48'} with support count 9018
{'39', '32'} with support count 8455
{'48', '32'} with support count 8034
{'38', '48'} with support count 7944
{'38', '41'} with support count 3897
Itemsets of size 1 are:
{'39'} with support count 50675
{'48'} with support count 42135
{'38'} with support count 15596
{'32'} with support count 15167
{'41'} with support count 14945
{'65'} with support count 4472
{'89'} with support count 3837
```

In [6]:

```
start_time_assoc = time.time()

frequentdict = {}
for item in frequentitemset:
    frequentdict[tuple(sorted(item[0]))] = item[1]

assoc_rules = []

for item in frequentitemset:
  if len(item[0])<2: break
  s=item[0]
  s_tuple = tuple(sorted(s))
  for i in range(1,len(s)):
    a = list(combinations(s, i))
    for t in a:
      t_set=set(t)
      t_s_diff = s.difference(t_set)
      t_tuple = tuple(sorted(t))
      cond=frequentdict[s_tuple]/frequentdict[t_tuple]
      if cond>min_cond_ratio:
        assoc_rules.append([t_set, t_s_diff, cond])

end_time_assoc = time.time()

print("Time taken to compute frequent itemsets is %s seconds" % (end_time_assoc - st


for rule in assoc_rules:
          print('Association rule',rule[0],'->',rule[1],'with confidence',rule[2])
```

```
Time taken to compute frequent itemsets is 0.0010056495666503906 secon
ds
Association rule {'41'} -> {'48', '39'} with confidence 0.492873870859
8193
Association rule {'41', '48'} -> {'39'} with confidence 0.816810822798
8468
Association rule {'41', '39'} -> {'48'} with confidence 0.645347818468
5474
Association rule {'38'} -> {'48', '39'} with confidence 0.391254167735
31674
Association rule {'38', '48'} -> {'39'} with confidence 0.768126888217
5226
Association rule {'38', '39'} -> {'48'} with confidence 0.589850169163
8472
Association rule {'32'} -> {'48', '39'} with confidence 0.356167996307
77346
Association rule {'48', '32'} -> {'39'} with confidence 0.672392332586
5073
Association rule {'39', '32'} -> {'48'} with confidence 0.638911886457
7173
Association rule {'48'} -> {'39'} with confidence 0.6916340334638661
Association rule {'39'} -> {'48'} with confidence 0.5750764676862358
Association rule {'41'} -> {'39'} with confidence 0.7637336901973905
Association rule {'38'} -> {'39'} with confidence 0.6633111054116441
Association rule {'41'} -> {'48'} with confidence 0.603412512546002
Association rule {'32'} -> {'39'} with confidence 0.5574602755983386
Association rule {'32'} -> {'48'} with confidence 0.5297026438979363
Association rule {'38'} -> {'48'} with confidence 0.5093613747114645
```

In [ ]:

```

```

In [ ]:

```

```