Project Report

Project Name: FP Growth for data mining

Course Code: **CSE477**

Section: 01

Submitted by

Adnan Saif
2018-1-60-157

Md Mahmudur Rahman Limon
2018-1-60-253

Kaniz Fatama Keya
2018-1-60-114

Asef Amer
2018-2-60-068

MD Momdu Mollah
2018-2-60-070

Submitted To

Jesan Ahmed Ovi

Senior lecturer

Department of Computer Science and Engineering

Introduction: In Data Mining, Association Rule Mining is a standard and well researched technique for locating fascinating relations between variables in large databases. Association rule is used as a precursor to different Data Mining techniques like classification, clustering and prediction. To measure the performance of the Frequent Pattern (FP) growth algorithm, by comparing their capabilities in different datasets. The evaluation study shows that the FP-growth algorithm is more efficient and ascendable and has many differentiable approaches for serving results.

Here, we are analyzing chess and mushroom datasets in terms of frequent pattern (FP) growth algorithm.

FP Growth algorithm: FP growth algorithm is an efficient algorithm for producing the frequent item sets without generation of candidate item sets. It adopts a Divide and Conquer strategy and it needs two database scans to seek out the Support Count. It can mine the items by using lift, leverage and conviction by specifying a minimum threshold. Steps for the algorithm,

1. Database first scan to find frequent single item set pattern.
2. Construct header table by sorting frequent items in frequency descending order.
3. Database 2^{nd} scan to construct FP tree.
4. Constructing the conditional FP tree in the sequence of reverse order header table to generate frequent item sets.

Comparing FP growth algorithm:

FP-growth algorithm
Tree based structure
Divide and Conquer technique
2 database scan
Less memory required
Runtime is less
For large and medium datasets

Datasets Analysis: The data has to be handled efficiently to get the best outcome from the Data Mining process**.** We are analyzing chess and mushroom datasets.

Dataset name	Number of data	Number of attributes
Chess	3196	37
Mushroom	8124	23

For chess datasets

The threshold value:

[0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95]

Runtime for FP-growth:

[17.6, 7.5, 3, 2.3, 2.1, 2, 1.6, 1.2]

Runtime for:

[90, 25, 15, 13, 8, 5, 3, 2]

For mushroom datasets

The threshold value:

[0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

Runtime for FP-growth:

[14.2, 0.81, 0.22, 0.18, 0.16, 0.15, 0.14, 0.13, 0.12]

Runtime for:

[20, 10, 2, 1.9, 1.7, 1.6, 1.5, 1.4, 1.2]

Implementation:

<u>Chess Dataset</u>

```
[ ] import numpy as np
    import time
    import matplotlib.pyplot as plt
    import pandas as pd
    from mlxtend.frequent_patterns import fpgrowth
    from mlxtend.preprocessing import TransactionEncoder
    import csv
```

Here we have imported some Python libraries to make it easier to load our dataset and generate graphs. We also import it from mlxtend to use the FP growth algorithm.

```
[ ] items=[]
    with open('chess.dat', 'r') as file:
        dataset = csv.reader(file, delimiter=' ')
        for row in dataset:
            temp=[element for element in row]
            items.append(temp)
```

Here, we have loaded our chess dataset.

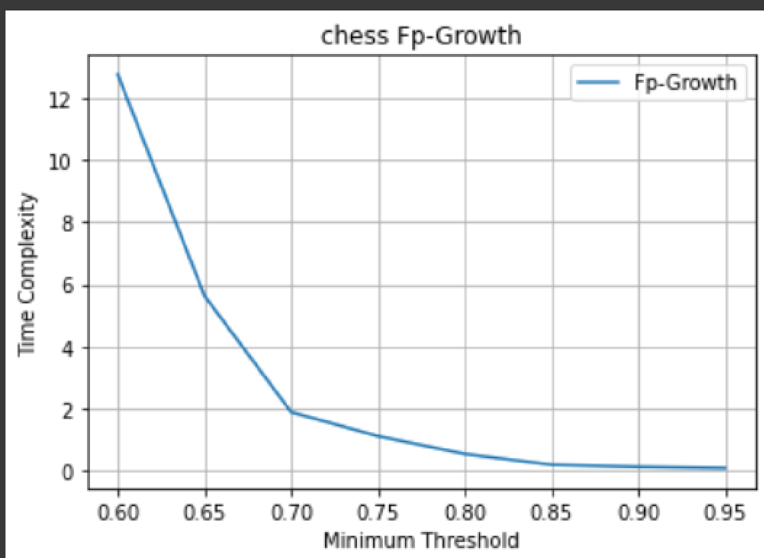
Here we build our model using the FP growth algorithm and started with mini support 0.6 and increased to 0.05 per step.

```
[ ] fpc_times = list()
    fpc_elements = list()
    minsupport = .6
    while minsupport <=1:
      start = time.process_time()
      te = TransactionEncoder()
      te_ary = te.fit(items).transform(items)
      df = pd.DataFrame(te_ary, columns=te.columns_)
      result= fpgrowth(df, min_support= minsupport, use_colnames=True)
      fpc_elements.append(minsupport)
      end = time.process_time()
      fpc_times.append(end-start)
      print (result)
      # total time taken
      print(f"Runtime of the program is {end - start}")
      minsupport +=.05
```

```
        support                  itemsets
0       1.000000                       ()
1       0.999687                     (58)
2       0.996558                     (52)
3       0.995307                     (29)
4       0.991865                     (40)
...          ...                      ...
509884  0.600751       (, 50, 62, 40, 58)
509885  0.600751        (, 50, 62, 7, 58)
509886  0.600751        (, 50, 62, 7, 40)
509887  0.600751     (50, 62, 7, 40, 58)
509888  0.600751  (, 50, 62, 7, 40, 58)

[509889 rows x 2 columns]
Runtime of the program is 12.740205250000002
        support          itemsets
0       1.000000                ()
1       0.999687              (58)
2       0.996558              (52)
3       0.995307              (29)
4       0.991865              (40)
...          ...               ...
222474  0.653630        (, 60, 11)
222475  0.653317      (60, 11, 58)
222476  0.650188      (60, 11, 52)
222477  0.653317  (, 60, 11, 58)
222478  0.650188  (, 60, 11, 52)
```

```
[ ] plt.title('chess Fp-Growth')
    plt.xlabel('Minimum Threshold')
    plt.ylabel('Time Complexity')
    plt.plot(fpc_elements, fpc_times, label ='Fp-Growth')
    plt.grid()
    plt.legend()
    plt.savefig('FpgrowthChess.pdf',bbox_inches='tight')
    plt.show()
```



Mushroom Dataset

```
import numpy as np
import time
import matplotlib.pyplot as plt
import pandas as pd
from mlxtend.frequent_patterns import fpgrowth
from mlxtend.preprocessing import TransactionEncoder
import csv
```

Here we have imported some Python libraries to make it easier to load our dataset and generate graphs. We also import it from mlxtend to use the FP growth algorithm.

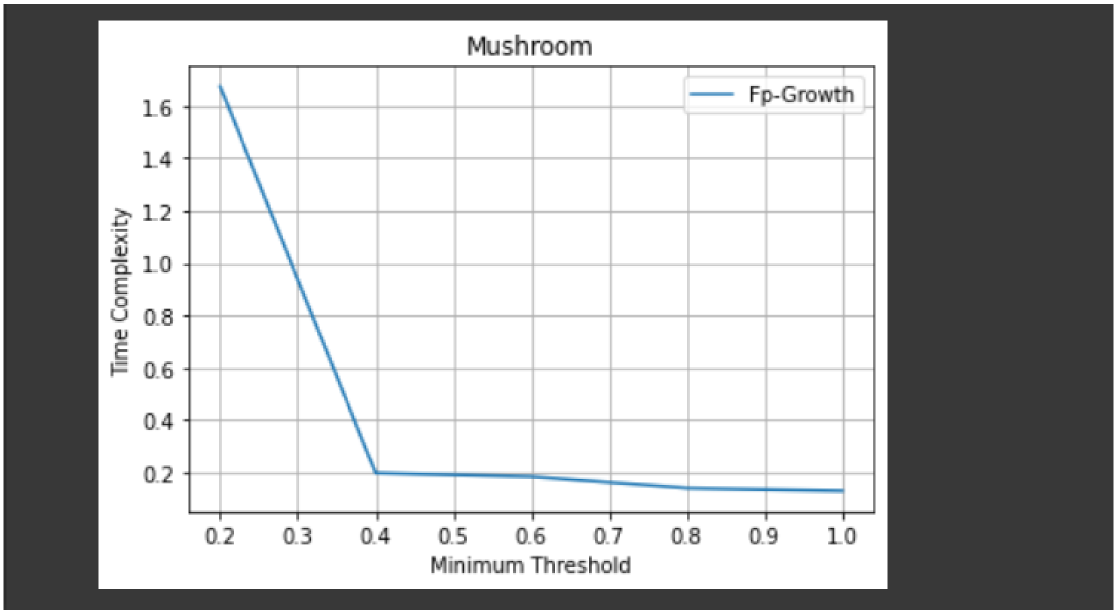
```
items=[]
with open('mushroom.dat', 'r') as file:
    dataset = csv.reader(file, delimiter=' ')
    for row in dataset:
        temp=[element for element in row]
        temp= temp[:-1]
        items.append(temp)
```

Here, we have loaded our mushroom dataset.

```
fpc_times = list()
fpc_elements = list()
minsupport = .2
while minsupport <=1:
  start = time.process_time()
  te = TransactionEncoder()
  te_ary = te.fit(items).transform(items)
  df = pd.DataFrame(te_ary, columns=te.columns_)
  result= fpgrowth(df, min_support= minsupport, use_colnames=True)
  fpc_elements.append(minsupport)
  end = time.process_time()
  fpc_times.append(end-start)
  print (result)
  # total time taken
  print(f"Runtime of the program is {end - start}")
  minsupport +=.2
```

```
         support                                         itemsets
0       1.000000                                               ()
1       1.000000                                             (85)
2       0.975382                                             (86)
3       0.974151                                             (34)
4       0.921713                                             (90)
...          ...                                              ...
107162  0.212703  (, 53, 102, 38, 36, 48, 24, 1, 86, 34, 110, 85...
107163  0.212703  (, 53, 102, 94, 36, 48, 24, 1, 86, 34, 110, 85...
107164  0.212703  (, 53, 102, 94, 36, 38, 48, 24, 1, 86, 34, 110...
107165  0.212703  (, 53, 94, 38, 36, 48, 24, 1, 86, 34, 110, 85,...
107166  0.212703  (, 53, 102, 94, 36, 38, 48, 24, 1, 86, 34, 110...

[107167 rows x 2 columns]
Runtime of the program is 1.6746577630000008
       support                      itemsets
0     1.000000                            ()
1     1.000000                          (85)
2     0.975382                          (86)
3     0.974151                          (34)
4     0.921713                          (90)
...        ...                           ...
1126  0.407681      (, 36, 39, 34, 85, 56, 90)
1127  0.407681      (, 36, 39, 86, 34, 56, 90)
1128  0.407681    (36, 39, 86, 34, 85, 56, 90)
1129  0.407681      (, 39, 86, 34, 85, 56, 90)
1130  0.407681  (, 36, 39, 86, 34, 85, 56, 90)
```



Result analysis: After observing the chess and mushroom datasets in the FP-growth algorithm, FP-growth shows a better result. Here, first we check the minimum threshold support against runtime for the datasets chess and mushroom. Because of the tree structure, the FP-growth algorithm uses less memory and works faster. It also scans the datasets 2 times for the mining process rather than taking k-1 times in. It is efficient and scalable for mining both long and short frequent patterns. It is a fundamental approach to data mining but FP-growth is definitely an improvement for the mining process.

Runtime with different minimum support:

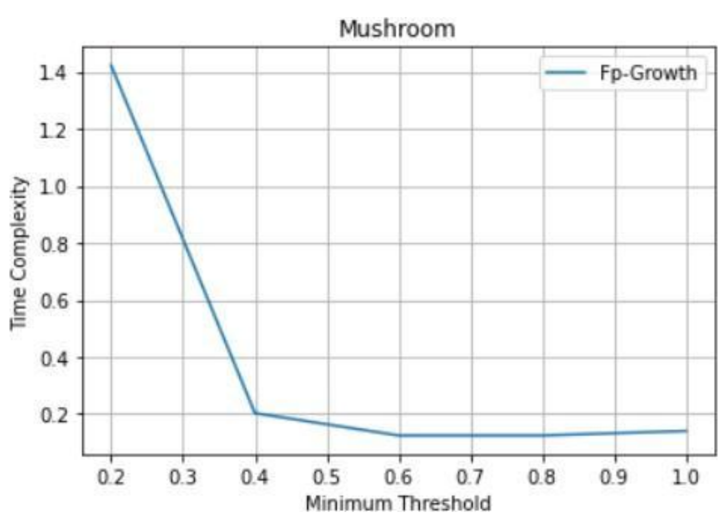


Figure 1: Execution time for various minimum thresholds for mushroom data (all transactions) in FP-growth

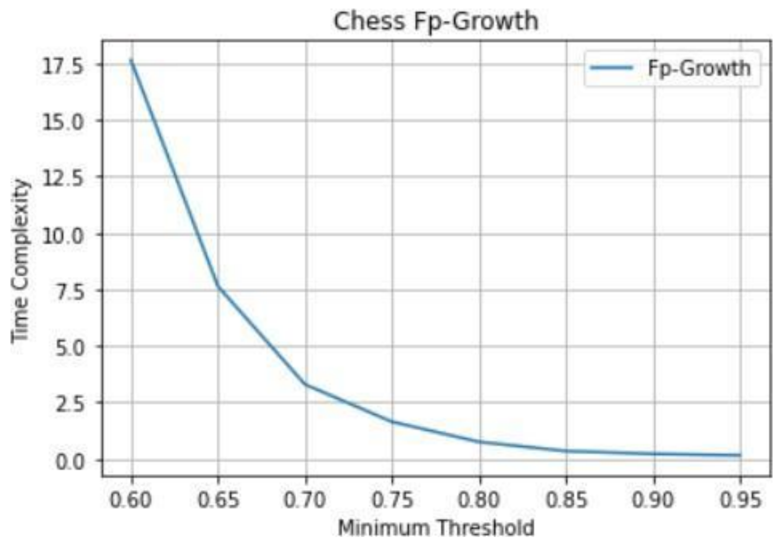


Figure 5: Execution time for various minimum thresholds for chess data (all transactions) in FP-growth

Conclusion:

After analyzing the datasets we have compared and FP-growth algorithm by their advantage disadvantage on Memory and time complexity. Frequent Pattern Mining or FP-growth algorithm can find frequent pattern efficiently. In other words we have to sort the items in frequency descending order before using it to construct the tree. We can see from the graph, sorting with descending order is always faster. And the difference between speeds is more obvious with lower support. As a result more frequently occurring items will have better chances of sharing items. Thus the performance of FP-growth algorithm is higher, than generating candidate and scan every step.