East West University

Fall 2023

Course Title: Data Mining

Course code: CSE 477

Final Project - FP-Growth Algorithm

Submitted To:

Course Instructor: Dr. Md Mostofa Kamal Rasel

Designation: Associate Professor, Department of Computer Science and Engineering, East West University

Submitted By:

Name	ID
Md Rezaul Hossain	2020-2-60-165
Fakiha Rahman Soha	2020-2-60-136
Tajkiratul Abida Ananna	2020-2-60-082
Md Farhad Billah	2020-2-60-213

Introduction

Frequent Pattern Growth, or FPGrowth, is a strong and effective technique used in machine learning and data mining to find patterns in huge datasets. FPGrowth, created in 2000 by Jiawei Han, Jian Pei, and Yiwen Yin, is especially well-liked in association rule mining, where determining relationships and associations between objects in a transactional database is the main objective. Some of the shortcomings of conventional algorithms for mining frequent patterns, including the Apriori algorithm, are intended to be addressed by the FPGrowth algorithm. FPGrowth is noteworthy for its capacity to build the FP-tree, or frequent pattern tree, a compact data structure. The tree model facilitates the mining of frequently occurring itemsets in an efficient and scalable manner, eliminating the requirement for the computationally costly generation of candidate sets.

The technique employs a divide-and-conquer tactic, extracting common patterns from this condensed structure after iteratively partitioning the dataset to create the FP-tree. FPGrowth greatly lowers the computational overhead associated with frequent pattern mining by doing away with the requirement for multiple database scans and candidate generation, which makes it ideal for managing big and complicated datasets. Applications for FPGrowth can be found in many fields, such as network intrusion detection, bioinformatics, and market basket analysis. Its capacity to handle high-dimensional data efficiently and scale makes it a useful tool for deriving insights and patterns from a variety of datasets, advancing knowledge discovery in contexts with large amounts of data.

The FP-Growth Algorithm

Building an FP-tree (Frequent Pattern tree) and then mining frequent patterns from this tree are the two steps in the FPGrowth (Frequent Pattern Growth) algorithm. The steps of the FPGrowth algorithm are as follows in detail:

1. **Scan the Database:** Go through the dataset to identify the support of each item (the frequency of occurrence). This step is usually called the "first pass" or "initial scan."

2. **Sort Items by Support**: Sort the items in descending order of support. This ensures that the FP-tree construction starts with the most frequent items, optimizing the tree-building process.
3. **Construct the FP-Tree**:
    a. Create the root of the FP tree and scan the dataset again. For each transaction, insert the items into the tree while maintaining the item order based on support.
    b. Incrementally build the tree by adding branches and nodes for each transaction. Nodes represent items, and edges represent the order and frequency of item occurrence.
4. **Create Conditional Pattern Bases**: For each frequent item in the FP-tree, create a conditional pattern base. This involves extracting the paths in the tree that end with the target item and removing the target item itself. These paths form the basis for constructing conditional FP-trees.
5. **Mine Frequent Patterns Recursively**: For each frequent item in the FP-tree, mine frequent patterns recursively using the corresponding conditional pattern base. This process involves building a conditional FP-tree for each frequent item and repeating the steps from 1 to 4 on these smaller datasets.
6. **Combine Frequent Patterns**: Combine the frequent patterns obtained from the conditional FP-tree mining step to form the complete set of frequent patterns for the entire dataset.

Implementation

The FP-Growth algorithm has been implemented on two large datasets in this project and compared the results. These datasets are the chess dataset and the mushroom dataset. These files contain lines of integer numbers. Each line in these files is a transaction and a transaction is recorded in the form of “item_1 item_2…..”, where each item is represented by an integer.

In this implementation first, the nodes of the tree have to be created. The given code blocks are responsible for creating the nodes with the intended indentation.

```
[ ]
    class Node:
      def __init__(self, *args, **kwargs):
        self.pattern = kwargs['pattern']
        self.frequency = kwargs['frequency']
        self.parent = kwargs['parent']
        self.children = []

      def __str__(self):
        self.show()
        return ''

      def show(self):
        if self.pattern:
          print(self.pattern, end='')
          print(' : ', end='')
          print(self.frequency)
        else:
          print('root')

        if self.children:
          for child in self.children[:-1]:
            self.__elegant_print(child, '', '├─')

          self.__elegant_print(self.children[-1], '', '└─')



      def __elegant_print(self, node, prefix, symbol):
        print(prefix + symbol + ' ', end='')
        print(node.pattern, end='')
        print(" : ", end='')
        print(node.frequency)

        if node.children:
          for child in node.children[:-1]:
            self.__elegant_print(child, prefix + "|\t" if symbol == '├─' else prefix + ' \t', '├─')

          self.__elegant_print(node.children[-1], prefix + "|\t" if symbol == '├─' else prefix + ' \t', '└─')

      def add_children(self, child):
        self.children.append(child)
```

The next step is to create the class for **FP-Growth** that will take the path of the dataset as an argument and create a root using the **Node** class. The following code block defines the FP-Growth class and initializes the parameters.

```
class FPgrowth:
  def __init__(self, *args, **kwargs):
    self.path = kwargs['path']
    self.root = Node(pattern=(), frequency=None, parent=None)
    self.header, self.order, self.total = self.__header_table()
    self.__build_tree()
```

A **_header_table()** method has been used to create a dictionary sorted by item frequency in descending order, a list of item names sorted by frequency and the total number of transactions. This table will be used in the construction of the FP-tree.

```
def __custom_sort_key(self, item):
  if item in self.order:
      return self.order.index(item)
  else:
      return len(self.order)


def __header_table(self):
  temp = defaultdict(int)
  count = 0
  with open(self.path, 'r') as f:
    for line in f:
      count += 1
      transactions = sorted(line.split())
      for i in transactions:
        temp[i] += 1

  return dict(sorted(temp.items(), key=lambda x:x[1],reverse = True)[::-1]), [i[0] for i in sorted(temp.items(), key=lambda x:x[1], reverse=True)], count
```

After that with the help of **_build_tree()** method the tree has been constructed. At first the dataset file has been opened to read line by line with the help of a for loop. The inner loop processes each item in the transaction, sorted based on a custom sorting key (self.__custom_sort_key).

```
def __build_tree(self):
  # print('Transactions:')
  with open(self.path, 'r') as f:
    for line in f:
      # print(line, end='')
      current = self.root

      for i in sorted(line.split(), key=self.__custom_sort_key):
        found = False

        for child in current.children:
          if child.pattern == i:
            child.frequency += 1
            current = child
            found = True
            break

        if not found:
          temp = Node(pattern=i, frequency=1, parent=current)
          current.add_children(temp)
          current = temp

  return self.root
```

Then it checks if the current item is already a child of the current node in the FP-tree. If yes, it increments the frequency of the existing node and moves to that node.If the item is not a child of the current node, it creates a new node with the item's pattern, sets the frequency to 1, and adds it as a child to the current node. The current node is then updated to the newly created node and then it returns the root of the tree.

The next step is to mine the tree with the help of the **mine()** method which takes the frequency threshold as a percentage. It initializes a default dict to store frequent patterns and their counts. This dictionary will be populated during the mining process. After that, it calculates the threshold value from the percentage given in the function. Then prints the number of transactions and the frequency threshold. Then it sets the timer and mines the frequent patterns using _get_frequent_patterns() function and prints the computational time.

```
def mine(self, *args, **kwargs):
  self.frequent_patterns = defaultdict(int)
  self.threshold = ceil(self.total*kwargs['threshold'])
  print(f"total transactions {self.total}")
  print(f"threshold {self.threshold}")

  start = time()
  self.__get_frequent_patterns(self.header, self.root)
  end = time()

  print()
  print(f"completed in {end-start} secs")
```

The **_get_frequent_patterns()** function is responsible for effectively exploiting the entire FP-Tree to identify all frequent items set based on the given threshold value. It iterates through the header for each item to check if the frequency meets the given threshold value. If the frequency is sufficient, it adds the pattern with the current item to the dictionary of frequent patterns as well as its frequency. After that using _get_conditional_trasaction() all the conditional transaction of the current item has been obtained. If there are conditional transactions a conditional FP-Tree has been built and recursively calls the _get_frequent_patterns() function with the conditional header and tree.

```
def __get_conditional_transactions(self, tree, item):
  queue = [tree]
  path_list = []

  while queue:
    current = queue.pop(0)
    for child in current.children:
      if child.pattern == item:
        path = self.__get_path(current,[])
        if path:
          for i in range(child.frequency):
            path_list.append(path)
      else:
        queue.append(child)

  return path_list
```

The **_get_path()** method is a recursive function that is used to obtain the path from a node in the FP-tree back to the root. The path represents the sequence of itemsets that lead from the current node to the root of the tree. The **_get_conditional_header()** method takes a set of transactions as input and generates a conditional header based on the frequency of items in those transactions.

```
def __get_path(self, tree, path):
  if not tree.parent:
    return path

  path.append(tree.pattern)
  return self.__get_path(tree.parent, path)


def __get_conditional_header(self, transactions):
  temp = defaultdict(int)
  for line in transactions:
    for i in line:
      temp[i] += 1

  return dict(sorted(temp.items(), key=lambda x:x[1],reverse = True)[::-1])
```

The function **__get_conditional_tree()** takes a set of transactions and a root node as input, where each transaction is a sequence of items. It iterates through each transaction, updating a conditional header to keep track of the frequency of items. Simultaneously, it constructs a conditional FP-tree by traversing the tree structure, updating existing nodes when an item is encountered, and creating new nodes if necessary. The conditional header reflects the frequencies of items in the context of these transactions. Ultimately, the function returns a tuple containing the sorted conditional header and the root of the conditional FP-tree.

This part initializes a pointer to the root of the conditional tree and processes each item in the transaction. It maintains a conditional header, updating item frequencies as it traverses the transactions. The code checks whether each item is already a child of the current node in the conditional tree. If it is, the code increments the frequency of the existing node and moves to that node. If the item is not a child, a new node is created with the item's pattern, its frequency is set to 1, and it is added as a child to the current node. The process repeats for all transactions, resulting in a conditional FP-tree that encapsulates patterns relevant to a specific item and a corresponding conditional header reflecting the item frequencies in these patterns.

```
def __get_conditional_tree(self, transactions, root):
  temp_header = defaultdict(int)
  for line in transactions:
    current = root
    # print(line)
    for i in sorted(line, key=self.__custom_sort_key):
      # print(type(i))
      temp_header[i] += 1
      found = False

      for child in current.children:
        if child.pattern == i:
          child.frequency += 1
          current = child
          found = True
          break

      if not found:
        temp = Node(pattern=i, frequency=1, parent=current)
        current.add_children(temp)
        current = temp

  return dict(sorted(temp_header.items(), key=lambda x:x[1],reverse = True)[::-1]), root
```

The **count_frequent_pattern()** method organizes and prints the counts of frequent patterns based on their lengths. It uses a defaultdict to categorize patterns by length and then prints the lengths along with the corresponding counts. Here "sorted_patterns" is a defaultdict where keys represent the lengths of frequent patterns, and values are lists containing tuples of sorted patterns and their frequencies. The code iterates over each frequent pattern. For each pattern, it appends a tuple (sorted pattern, frequency) to the list associated with the pattern's length in sorted_patterns. After processing all frequent patterns, it iterates over the lengths of patterns in ascending order. For each length, it prints the length and the count of patterns of that length.

```
def count_frequent_pattern(self):
  print()
  print(f"lengthwise frequent pattern count for threshold {self.threshold}")
  sorted_patterns = defaultdict(list)
  for pattern in self.frequent_patterns:
    sorted_patterns[len(pattern)].append((tuple(sorted(pattern)), self.frequent_patterns[pattern]))

  for i in sorted(sorted_patterns):
    print(f"length {i} : {len(sorted_patterns[i])}")
```

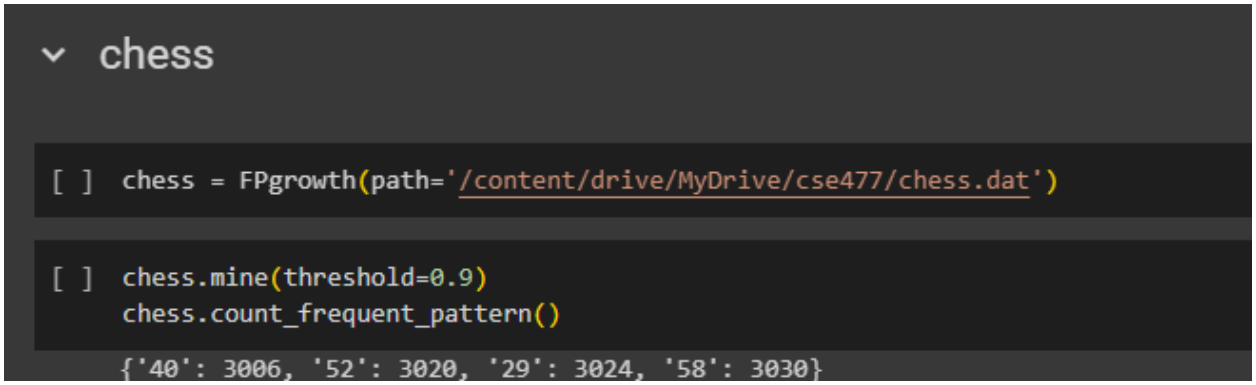
The **show_frequent_patterns()** method organizes and displays information about frequent patterns based on their lengths. It uses a defaultdict to categorize patterns by length, where keys represent the lengths, and values are lists of tuples containing sorted patterns and their corresponding frequencies. If no specific lengths are provided as arguments **args**, the method iterates over all lengths, sorts the patterns within each length category, and prints each pattern along with its frequency. However, if specific lengths are specified as arguments, the method only prints patterns and frequencies for those lengths. This functionality aids in presenting a clear and organized representation of frequent patterns, allowing users to inspect and analyze patterns of interest.

```
def show_frequent_patterns(self, *args, **kwargs):
  sorted_patterns = defaultdict(list)
  for pattern in self.frequent_patterns:
    sorted_patterns[len(pattern)].append((tuple(sorted(pattern)), self.frequent_patterns[pattern]))

  if not args:
    for i in sorted_patterns:
      sorted_patterns[i].sort()
      for pattern, frequency in sorted_patterns[i]:
        print(pattern, end=' : ')
        print(frequency)
  else:
    for i in args:
      for pattern, frequency in sorted_patterns[i]:
        print(pattern, end=' : ')
        print(frequency)
```

That is the end of the FPtree class.

After that the dataset has been loaded to an object that has been created with the help of FPtree class and has been called two methods to mine the pattern and count the frequent pattern. The code snippet is given below.



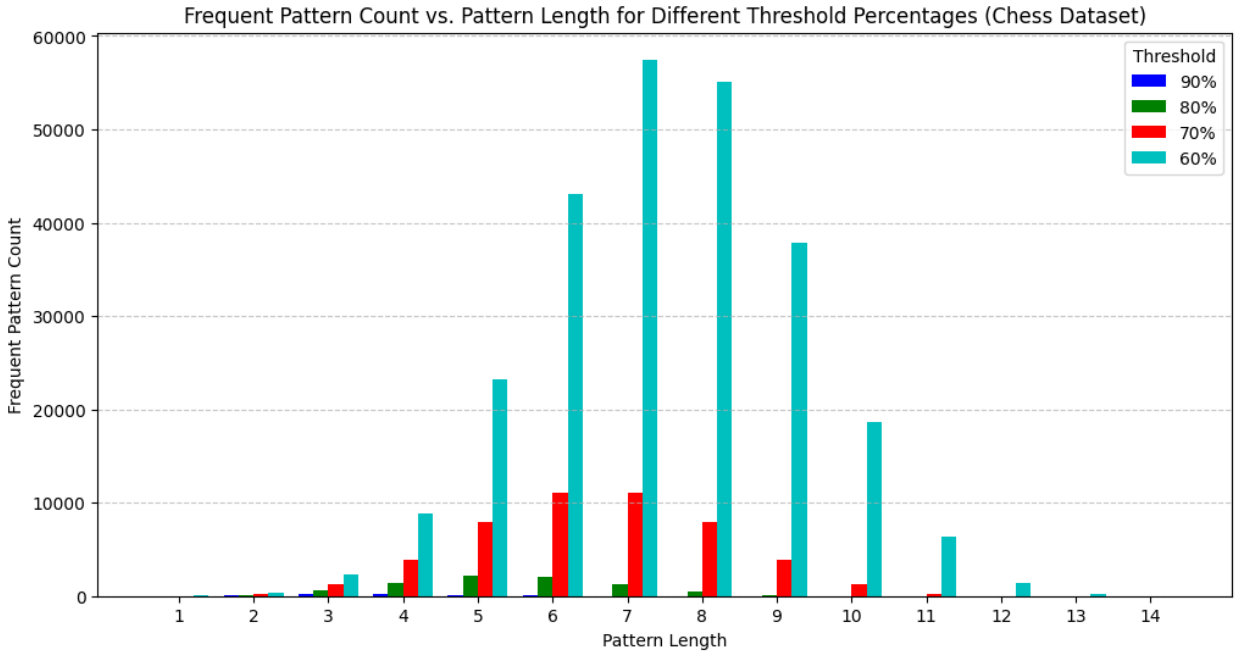
This process has been repeated multiple times with threshold values of 0.9, 0.8, 0.7 and 0.6. Also the algorithm has been applied to the mushroom dataset with the same threshold values. To analyze the performance visually some graphical representations have been done as well using matplotlib library. The outcome of the algorithm has been discussed in the following section of the report.

Output

The Chess Dataset

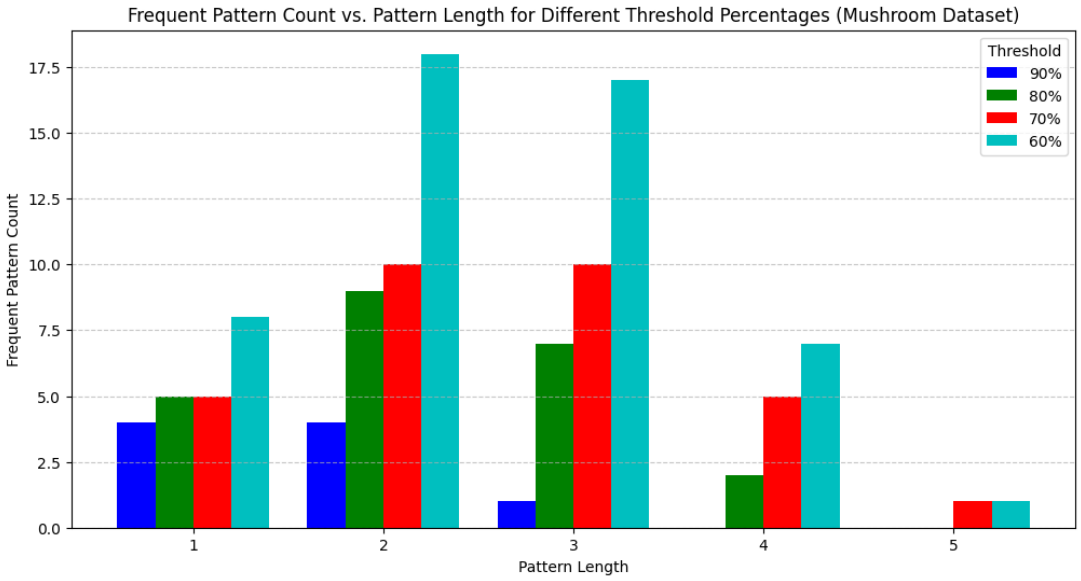
Threshold 0.9 : Completed in 1.845860242843628 secs lengthwise frequent pattern count for threshold 2877 length 1 : 13 length 2 : 68 length 3 : 167 length 4 : 203 length 5 : 128 length 6 : 39 length 7 : 4	**Threshold 0.8 :** completed in 25.662269592285156 secs lengthwise frequent pattern count for threshold 2557 length 1 : 19 length 2 : 141 length 3 : 566 length 4 : 1383 length 5 : 2130 length 6 : 2104 length 7 : 1314 length 8 : 481 length 9 : 85 length 10 : 4
Threshold 0.7 : completed in 127.77382493019104 secs lengthwise frequent pattern count for threshold 2238 length 1 : 24 length 2 : 238 length 3 : 1237 length 4 : 3857 length 5 : 7891 length 6 : 11125 length 7 : 11113	**Threshold 0.6 :** completed in 595.7428267002106 secs lengthwise frequent pattern count for threshold 1918 length 1 : 34 length 2 : 389 length 3 : 2325 length 4 : 8831 length 5 : 23155 length 6 : 43106

length 8 : 7916 length 9 : 3895 length 10 : 1216 length 11 : 204 length 12 : 14 length 13 : 1	length 7 : 57479 length 8 : 55062 length 9 : 37876 length 10 : 18607 length 11 : 6419 length 12 : 1466 length 13 : 187 length 14 : 8

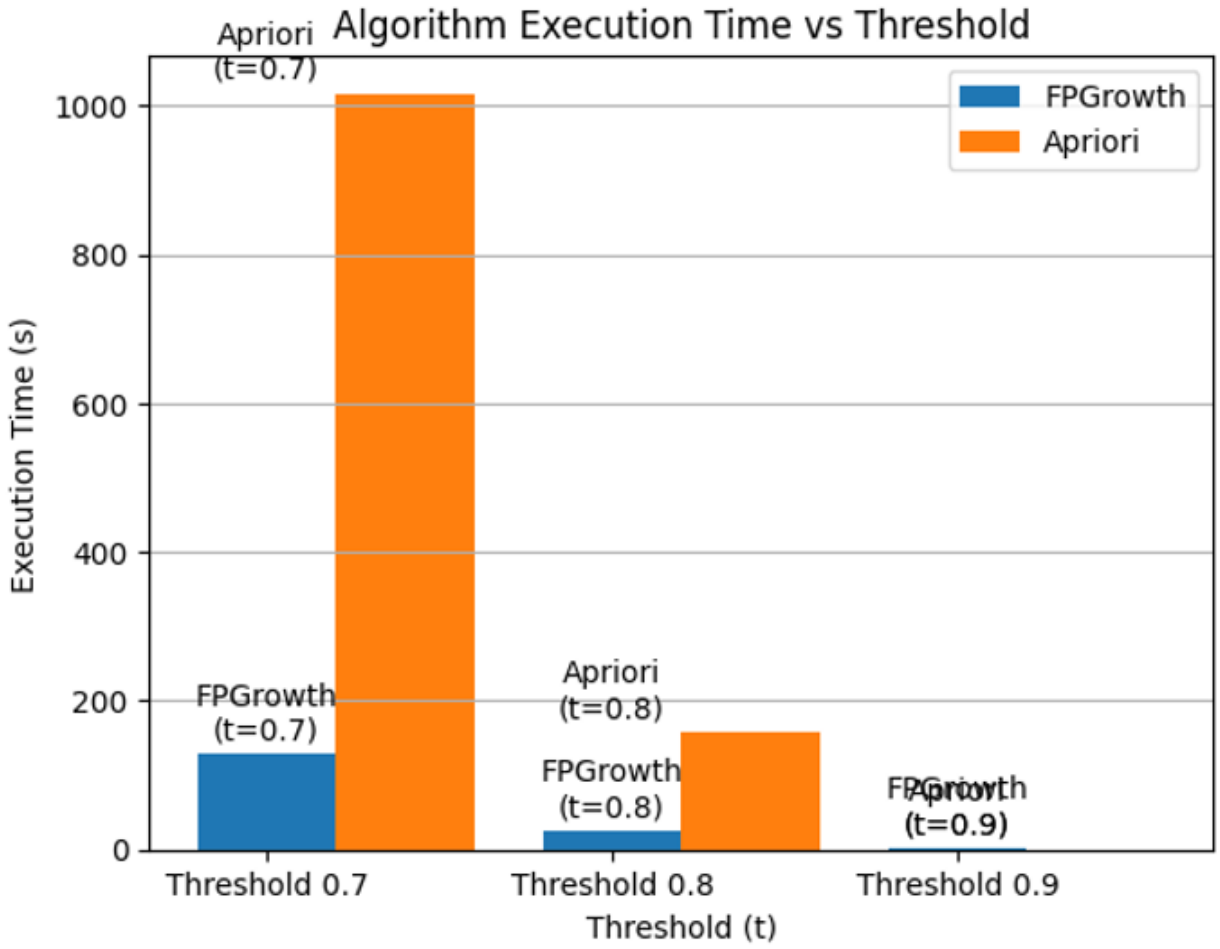


The Mushroom Dataset

Threshold 0.9 : completed in 0.051717281341552734 secs lengthwise frequent pattern count for threshold 7312 length 1 : 4 length 2 : 4 length 3 : 1	**Threshold 0.8 :** completed in 0.1360759735107422 secs lengthwise frequent pattern count for threshold 6500 length 1 : 5 length 2 : 9 length 3 : 7 length 4 : 2
Threshold 0.7 : completed in 0.171217679977417 secs lengthwise frequent pattern count for threshold 5687 length 1 : 5 length 2 : 10 length 3 : 10 length 4 : 5 length 5 : 1	**Threshold 0.6 :** completed in 0.3181943893432617 secs lengthwise frequent pattern count for threshold 4875 length 1 : 8 length 2 : 18 length 3 : 17 length 4 : 7 length 5 : 1



Comparison of FPgwoth and Apriori algorithm:



Here we can see the FP-Growth algorithm performs better compared to the Apriori algorithm for the same dataset with the same threshold value.