Stony Brook University

ESE 589: Learning Systems for Engineering Applications

Project 1

Sushanth Reddy Gurram #115668498
Mohammadreza Bakhtiari #115843646

Instructor: Prof. Alex Doboli
Department of Electrical and Computer Engineering

October 2023

Table of Contents

Part 1 - FP Growth Algorithm

I. Introduction

FP-Growth (Frequent Pattern Growth) is an algorithm used for mining frequent patterns in large datasets, especially in the context of association rule mining. It's particularly effective for handling large transactional datasets, such as those found in retail, market basket analysis, web usage mining, and more. FP-Growth aims to discover frequent itemsets, which are subsets of items that frequently appear together in the dataset.

There are other algorithms for mining frequent patterns such as **Apriori** algorithm and **ECLAT** algorithm. Apriori algorithm mines frequent patterns by scanning the full transactional database multiple times which increases the time complexity exponentially(2^n time complexity!). To counter the performance of Apriori algorithm in terms of speed, there is another famous algorithm known as ECLAT algorithm which does not involve repeated scanning of the transactional database to compute individual items support values, hence faster!. But, when run on large datasets, ECLAT uses up a lot of memory space.

FP-Growth algorithm addresses these challenges by mining frequent patterns without generating candidate rule at every step and also it avoids scanning transactional databases multiple times. It is also efficient in memory usage since it stores the items present in transaction in a graph data structure(FP-Tree) thus better performance in terms of both time and memory usage.

This project focuses on implementing the FP-Growth algorithm on different transactional datasets and evaluating its performance.

II. Implementation

II.1 Data Preproprocessing

Python is used as the primary coding language for this project.

Before using the given transactional data, it is important that the transactional data does not contain any abnormal values which affect the mined frequent patterns. The abnormal values are of different types such as "?", empty values, empty transactions. These abnormal cases are handled before using the transactional data for mining frequent patterns such as:

i) For items with "?" :- These are removed from the items using character matching.

ii) For empty values :- For categorical data, these are filled with the item's **mode** in the transaction data. For numerical values, they are filled with the item's **mean** in the transaction data.

iii) For empty transactions :- Empty transactions do not contribute to the frequent patterns, hence they are completely removed.

Once the transactional data is preprocessed, the data is ready for frequent pattern mining.

II.2 Data Structures

Following is the list of different data structures used for implementing the algorithm

1. For transactional data :-
   a. Each transaction present in the transactional data is stored as a **list**.
   b. All such lists are stored in a list, forming a list of lists for transactional data.
   c. List is chosen for its easy traversal operation.
2. For FP Tree node :-
   a. A class is created with following attributes :-

      i. tree

      ii. item

      iii. count = 1

      iv. parent

      v. children = {}

      vi. next_pointer = None

   b. Count of an FP Tree node is initilized to 1 and children attribute stores the children nodes of the current item's FP tree node.
   c. Next_pointer points to the the same item in the FP tree node when present in another branch of the same tree.
   d. Parent points to the parent of the current item's FP tree node.
   e. Tree points to the current item's FP tree root.
3. For FP Tree :-
   a. A class is created with following attributes :-

      i. root = NULL

      ii. header= {}

   b. Root is the current FP tree's root.
   c. Header is a dictionary(key-value pair) which stores the items and their support values of the present in the current FP tree and links to the items it is connected to.

The code is attached to the doc at the end of the report as Appendix.

Detailed steps involved in the Algorithm:

I. Preparing the Transactional Dataset
   a. Given dataset is preprocessed(inorder to remove abnormal values).
   b. Each **transaction** is stored as a list of items in another list **transactions.**
   c. Preprocessed transactional dataset is scanned once and respective frequencies(support values) for each item is stored in a dictionary **itemset** as key-value pair.
   d. Given the support values of each item, the itemset is sorted in descending order of support values.
   e. Each transaction in transactions is now ordered in the same order of itemset.

II. Creating FP tree

a. A class **FPTree** is created(with attributes shown in Data Structures section of the report). An empty fp_tree is created using **FPTree** class.
b. Each item in the transaction is added to the fp_tree as a node with default values as shown in the Data Structures section of the report.
c. While adding the nodes to the fp_tree, corresponding item is updated in the **header(**route table) as well with its support value from itemset and the item in header is linked to the same item in fp_tree.

III. Creating Conditional Pattern Base

a. For each conditional_pattern_base call, the item with the lowest support value in current iteration is taken from header and all the parent paths corresponding to that item are obtained through bottom-up traversal of the created fp_tree.
b. A new conditional_fp_tree is created using FPTree class.
c. Nodes corresponding to the parent_paths are added to this conditional_fp_tree.
d. After adding the nodes to the conditional_fp_tree, a check is performed to determine whether the created conditional_fp_tree has a single branch or not.
e. If only one branch exists and the items present in the parent path have support values greater than minimum_support, a frequent pattern is generated->traverse the single branch of FP Tree to generate frequent pattern.
f. Else if more than one branch exist, a recursive call is made to conditional_pattern_base until a tree with single branch is created.

II.3 Pseudo Code for FP-Growth Algorithm

```
processed_transactional_data = data_preprocessing(transactional_data)


FPGrowth(processed_transactional_data,minimum_support):
        return generate_frequent_itemsets(processed_transactional_data,minimum_support)


generate_frequent_itemsets(data, minimum_support):
        frequency_dictionary = count of items in data
        sort frequency_dictionary in decreasing order
        sort data in order of frequency_dictionary
        fp_tree = FP_Tree()
        ∀ transaction in data:
                fp_tree.append(transaction)
        return conditional_pattern_base(fp_tree,[],minimum_support)


conditional_pattern_base(fp_tree,itemlist_retrieved,minimum_support):
   header_item, corresponding_nodes = fetch nodes(fp_tree)
   for each header_item:
        current_support_value = sum(corresponding_nodes support values)
        if current_support_value ⩾ minimum_support and header_item not in itemlist_retrived:
```

```
frequent_itemset = header_item + itemlist_retrieved
yield frequent_itemset, minimum_support
parents_path = fetch_parent_paths(header_item)
conditional_fp_tree = fp_tree()
conditional_fp_tree.append(nodes in parents_path)
for each frequent_item_sets ∊ conditional_pattern_base(
                               conditional_fp_tree,frequent_itemsets,minimum_support):
    yield frequent_item_sets
```

II.4 Execution of FP-Growth Algorithm

To test the implemented FP-Growth algorithm, we used 15 different datasets from UCI benchmark datasets and one small sample dataset to validate our implementation. Each dataset comprises categorical, numerical, or mixed type data. Upon inspecting the datasets, we found that certain attributes have the same value (e.g., 0) for many transactions. When performing the FP-Growth algorithm directly on the transactional dataset, the total number of mined frequent patterns is less than the actual required number of frequent patterns. To counter this, we appended each item present in the transactional data with its attribute name as part of our data preprocessing. This makes it more robust and understandable.

II.4.0 Validating FP-Growth on Sample Dataset:

For validating implemented FP-Growth algorithm, we have taken a small sample dataset as shown below,

```
k,e,m,o,y
k,e,o,y
k,e,m
k,m,y
k,e,o
```

where each letter is a short form for the items bought in a supermarket. Their description is as follows:

i. k = Ketchup

ii. e = Eggs

iii. m = Mangoes

iv. o = Oil

v. y = Yogurt

Upon solving for frequent patterns for above mentioned dataset on paper, the frequent itemsets are **{k,y :3},{k,o :3},{e,o :3}, {e,k,o :3}, {k,m :3}, {e,k :4}** for a minimum support of 3.

When the same dataset is executed on the implemented FP-growth algorithm, the outputs are as follows:

"['k', 'e']": 4, "['k', 'm']":3, "['k', 'o']":3, "['e', 'o']":3, "['k', 'e', 'o']":3, "['k', 'y']":3

The algorithm returns the same set of frequent patterns for a give dataset. This validates our implementation of the algorithm. Now generating results on 15 benchmark datasets.

II.4.1 Dataset-1 : Abalone

This dataset contains 4177 transactions with 9 features each. For the given dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
100	8575200	409.996	59
600	8429056	410.491	5
1000	8554248	359.303	3
2000	8442368	414.517	0
2500	8443752	403.346	0

II.4.2 Dataset-2 : Adult

This dataset contains 48842 transactions with 15 features each. For the given dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
100	122647056	56508.183	419068
700	48745136	12099.33	31605
1750	46543216	4268.543	7805
5000	43709960	2581.798	1184
10000	41575480	2162.864	228

II.4.3 Dataset-3 : Air-Quality

This dataset contains 9358 transactions with 16 features each. For the given dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined

100	42390448	4504.19	33917
700	35783864	1360.623	15
1750	35783328	1318.016	1
5000	35782480	1341.364	1
6000	35782480	1397.984	1

II.4.4 Dataset-4 : Balance-Scale

This dataset contains 625 transactions with 5 features each. For above dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
10	700880	76.023	405
50	740640	35.016	46
100	694624	35.074	22
200	256312	10.01	2
400	268568	8.224	0

II.4.5 Dataset-5 : Breast-Cancer

This dataset contains 286 transactions with 10 features each.For above dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table:

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
10	2051376	539.685	4041
30	973280	145.311	655
50	948096	58.854	237
70	682096	33.734	117
150	409040	13.421	15

II.4.6 Dataset-6 : Computer-Hardware

This dataset contains 209 transactions with 10 features each. For above dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
10	599400	30.112	86
20	622896	17.182	24
30	563616	17.406	11
50	519008	12.204	3
75	513656	13.249	1

II.4.7 Dataset-7 : Glass-Identification

This dataset contains 214 transactions with 10 features each.For above dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
10	653856	18.351	44
30	638312	17.089	14
50	637664	14.736	7
70	642568	14.156	6
150	623216	373.358	1

II.4.8 Dataset-8 : Iris

This dataset contains 150 transactions with 5 features each.For above dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
10	165320	6.002	23

30	128224	3.001	3
50	140208	4.0	3
70	137824	3.0	0
100	137008	4.0	0

II.4.9 Dataset-9 : Liver-Disorder

This dataset contains 345 transactions with 7 features each. For above dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
10	616856	32.968	112
30	532224	11.002	12
50	496000	11.013	6
70	515448	10.417	3
150	499816	11.947	1

II.4.10 Dataset-10 : Metro-Traffic

This dataset contains 48204 transactions with 11 features each. For above dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
100	37123192	1874.299	1795
700	36936360	1897.35	687
1750	36792424	1858.552	371
5000	36769032	2062.513	127
10000	36740720	1890.746	79

II.4.11 Dataset-11 : Online-Retail

This dataset contains 541909 transactions with 8 features each. For above dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
1000	533704440	22472.491	888
10000	531351672	24853.274	75
50000	531231088	24097.077	12
200000	531219488	23471.667	1
500000	531227864	22634.152	0

II.4.12 Dataset-12 : Tic-Tac-Toe

This dataset contains 958 transactions with 9 features each. For given dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
10	16818848	3676.093	19849
100	4649648	373.173	293
300	2263352	134.908	21
600	1481016	43.591	1
800	1494584	34.774	0

II.4.13 Dataset-13 : Congressional Voting Records

This dataset contains 435 transactions with 17 features each. For above dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
10	196497312	105884.677	985586

30	56983384	32305.74	214205
50	18950832	12976.213	63186
150	3882736	398.924	370
300	1105832	24.518	0

II.4.14 Dataset-14 : Wine

This dataset contains 178 transactions with 14 features each. For above dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
10	817032	28.767	12
30	790072	18.774	3
50	785648	19.322	2
70	799952	16.84	1
150	799696	15.842	0

II.4.15 Dataset-15 : Zoo Animals

This dataset contains 101 transactions with 18 features each. For given dataset, the number of mined frequent patterns, time taken for executing the code and memory utilized are summarized as shown below.

Summary Table :

Minimum Support	Memory Used (Bytes)	Execution Time (milli seconds)	Number of frequent patterns mined
10	53632048	18258.724	283134
30	3532392	819.464	11612
50	485672	29.228	193
70	283728	6.939	23
90	209152	4.541	1

Part 2 - Modified FP-Growth Algorithm for Grid Networks of Embedded Sensors

I. Introduction

In this part, we want to discuss the adaptation of the FP-Growth algorithm for mining frequent item sets to operate efficiently in grid networks of embedded sensors, as proposed in the paper "Linear programming-based optimization for robust data modeling in a distributed sensing platform". Our objective is to extend the applicability of the FP-Growth algorithm to these grid-based sensor networks, enabling them to discover valuable patterns and insights in the collected data.

II. Implementation

II.1 Description

In the grid network, sensor nodes collect data locally. We have introduced mechanisms for data collection, enabling each sensor node to aggregate data within its vicinity efficiently. Also, we need to develop methods for the distributed construction of the FP-Tree, where each sensor node builds a local FP-Tree from its collected data. Then, frequent item set mining is executed in a distributed manner, where sensor nodes share information and combine local frequent item sets to construct global patterns.

Now, let's take a look at the schematic of our implementation:

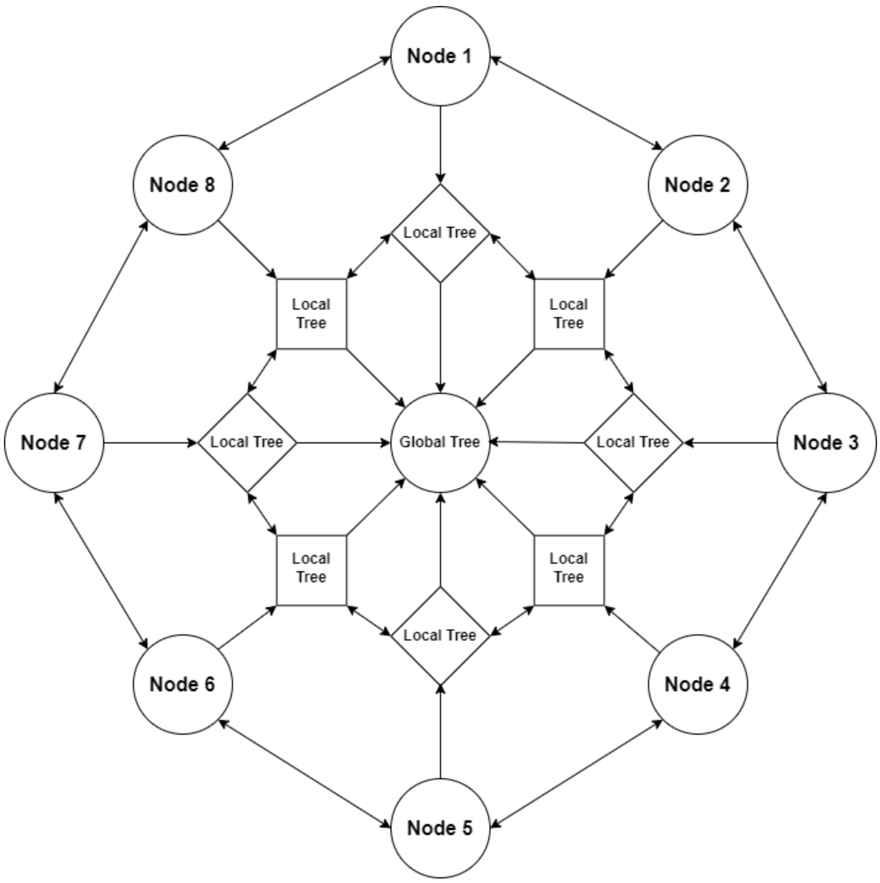

Figure 4: Simple schematic of implementation of FP-Growth for Grid Networks of Embedded Sensors

Note that Figure 4 is only a simple schematic and based on the connection of nodes we can have more complex and different implementations. For more complex and large Networks we can define a new structure, using intermediary nodes. These nodes serve as a bridge or connection between the local and global aspects of our network, helping to facilitate communication and data transfer between them. So, we can have multiple intermediary nodes and each of them has multiple nodes. A simple structure is shown below:

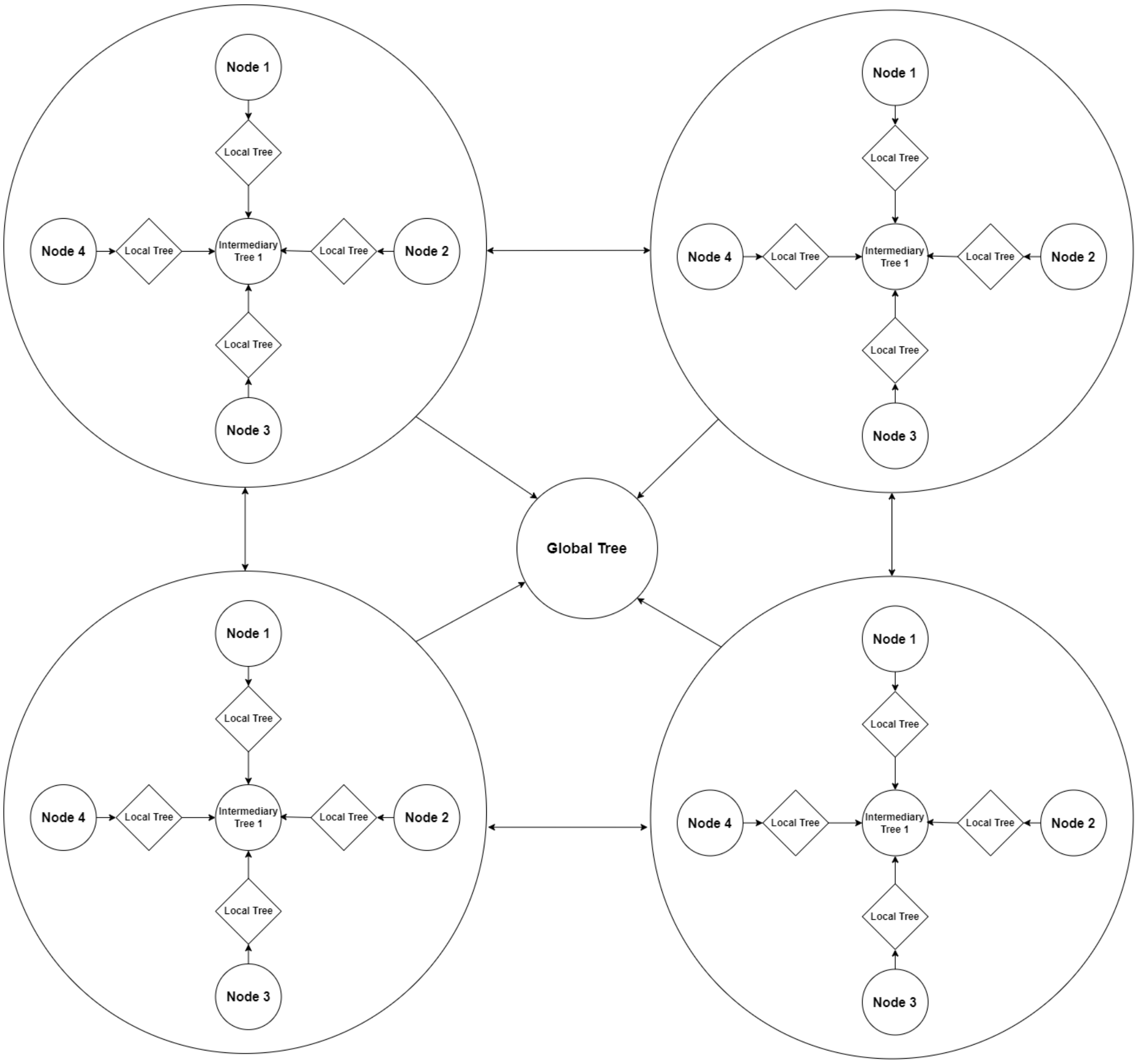

Figure 5: A more complex structure for our Network using intermediary nodes

II.2 Pseudo Code for Modified FP-Growth Algorithm for Grid Networks of Embedded Sensors

```
FPGrowth_GridNetworksofEmbeddedSensors(Data, Minimum_Support):
 ∀ Node in Network:
        Local_Data = Collect.Data.Locally(Data.Node)
        Processed_Local_Data = Preprocess.Data.Locally(Local_Data)
        Local_FPTree = Initialize.Local.FPTree(Processed_Local_Data)
```

```
        Locals_FPTree.append(Local_FPTree)
        Frequent_ItemSets = Mine.Local.Frequent.ItemSets(Local_FPTree, Minimum_Support)
        Locals_Frequent_ItemSets.append(Frequent_ItemSets)

 # Share local frequent item sets with neighboring nodes
 Shared_ItemSets = Share.Frequent.ItemSets(Locals_Frequent_ItemSets)
 # Merge local FP-Trees to construct a global FP-Tree
 Shared_FPTrees = Merge.Local.FPTrees(Locals_FPTree)

 # Perform global frequent item set mining
Return (Mine.Global.Frequent.ItemSets(Shared_ItemSets, Shared_FPTrees, Minimum_Support))
```

Appendix

Our Implentation of FP-Growth Algorithm consists of 4 files :- main.py, fp_tree_node.py, fp_growth.py and plot_utils.py.

main.py

```
import os
import numpy as np
import matplotlib.pylab as plt
import csv
import logging
import time
import psutil
import tracemalloc
from fp_tree_node import *
from fp_growth import *
from plot_utils import *

# Setting up a logger
logger = logging.getLogger()  # Initiating a logger
logging.basicConfig(filename="logs.log",
                    format='%(asctime)s %(message)s',
                    filemode='w')
logger.setLevel(logging.DEBUG)


# To begin with FP growth Algorithm, first we need to fetch the transactional data

transactions = []  # Initiliazing a list to store the transactions from csv file

# For "Adults" dataset(https://archive.ics.uci.edu/dataset/2/adult) fetched from UCI benchmark datasets
```

```
input_datasets_path = [
    "datasets/Abalone/abalone.csv",
    "datasets/Adult/adult.csv",
    "datasets/Air-Quality/air-quality.csv",
    "datasets/Balance-scale/balance-scale.csv",
    "datasets/Breast-Cancer/breast_cancer.csv",
    "datasets/Comp-Hardware/comp-hardware.csv",
    "datasets/Glass/glass.csv",
    "datasets/Iris/iris.csv",
    "datasets/Liver-Disorder/liver-disorder.csv",
    "datasets/Metro-Traffic/metro-traffic.csv",
    "datasets/Online-Retail/online-retail.csv",
    "datasets/Sample/sample.csv",
    "datasets/Tic-Tac-Toe/tic-tac-toe.csv",
    "datasets/Voting/voting.csv",
    "datasets/Wine/wine.csv",
    "datasets/Zoo/zoo.csv"
]


# List of minimum support valeus to test FP Growth Algorithm
minimum_support_abalone = [100,600,1000,2000,2500]
minimum_support_adult = [100,700,1750,5000,10000]
minimum_support_airquality = [100,700,1750,5000,6000]
minimum_support_balancescale = [10,50,100,200,400]
minimum_support_breastcancer = [10,30,50,70,150]
minimum_support_computer_hardware = [10,20,30,50,75]
minimum_support_glass = [10,30,50,70,150]
minimum_support_iris = [10,30,50,70,100]
minimum_support_liverdisorder = [10,30,50,70,150]
```

```
minimum_support_metro_traffic = [100,700,1750,5000,10000]

minimum_support_onlineretail = [1000,10000,50000,200000,500000]

minimum_support_tictactoe = [10,100,300,600,800]

minimum_support_voting = [10,30,50,150,300]

minimum_support_wine = [10,30,50,70,150]

minimum_support_zoo = [10,30,50,70,90]

minimum_support_sample = [3]

minimum_supports                                                                =
[minimum_support_abalone,minimum_support_adult,minimum_support_airqua
lity,minimum_support_balancescale,minimum_support_breastcancer,


minimum_support_computer_hardware,minimum_support_glass,minimum_suppo
rt_iris,minimum_support_liverdisorder,minimum_support_metro_traffic,


minimum_support_onlineretail,minimum_support_sample,minimum_support_t
ictactoe,minimum_support_voting,minimum_support_wine,

                    minimum_support_zoo]


count = 0

for input_dataset_path in input_datasets_path:

    minimum_support = minimum_supports[count]

    dataset_name = input_dataset_path.split("/")

    input_dataset_name = dataset_name[1]

    output_path = "outputs/" + input_dataset_name

    try:

        os.makedirs(output_path)

    except:

        pass


    transactions = []

     # Reading input CSV dataset and converting it into transactional
data

    logger.info("---------------Reading Input Data---------------")
```

```
    try:
        with open(input_dataset_path) as input_data:
            for row in csv.reader(input_data):
                transactions.append(row)
    except Exception as e:
        logger.error(e)
                    logger.info("---------------Transactional     Data
formed---------------")


    """
     Upon manual inspection of the input data, there are several data
entry  points  with  unwanted  characters  such  as  "?"  and  single
quotes('"')
       For  missing  data  points,  data  points  are  stored  as  either
""(empty values) or "?" in the dataset which bring no value.
       Hence  it  crucial  to  clean  the  data  before  using  FP  Growth
algorith to mine frequent patterns.


    """
                      logger.info("---------------Performing      Data
Cleaning---------------")
    try:
        for item in transactions:
            item = list(filter(lambda x: x != '', item))
            for i in item:
                if '?' in i:
                    item.remove(i)
                if '"' in i:
                    i = i.replace('"', '')
    except Exception as e:
        logger.error(e)
```

```
                        logger.info("---------------Data      Cleaning
Complete---------------")


        #  Finding  maximum  number  of  attributes  present  in  the
transactional dataset


    maximum_number_of_attributes = 0

    for transaction in transactions:

        if maximum_number_of_attributes < len(transaction):

            maximum_number_of_attributes = len(transaction)


     logger.info("Maximum number of attributes present in the dataset
: %s",

                maximum_number_of_attributes)




      #  Since  we  are  running  the  algorithm  multiple  times,  we  are
storing the time taken for the algorithm to complete and memory used
for each threshold value for analysis

    time_taken = []

    memory_used = []

    total_number_of_frequent_itemsets = []

    for threshold in minimum_support:

        frequent_itemsets = []

        tracemalloc.start()

        start_time = time.time()

         for itemset, support in mine_frequent_itemsets(transactions,
threshold):

            frequent_itemsets.append((itemset, support))


        end_time = time.time()

        memory_used.append(tracemalloc.get_traced_memory()[0])
```

```
        tracemalloc.stop()

        execution_time = end_time - start_time
            time_taken.append(round(execution_time*1000,3))  #  Time  in
micro seconds rounded off to 3 digits
        number_of_frequent_itemsets = len(frequent_itemsets)

total_number_of_frequent_itemsets.append(number_of_frequent_itemsets)

                    answer   =   sorted(frequent_itemsets,key=   lambda
x:x[1],reverse=True)
                                                                      with
open(f"{output_path}/Frequent_Patterns_{threshold}.csv","w") as f:
            write = csv.writer(f)
            write.writerows(answer)

    # Plotting execution time against several minimum support values
    x_axis = np.arange(len(minimum_support))

plt.bar(x_axis,time_taken,color="maroon",tick_label=minimum_support)
    xlabel = "Minimum Support Values"
    ylabel = "Execution Time(milli-seconds)"
    addlabels(x_axis,time_taken)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.savefig(f"{output_path}/{ylabel}.png")
    plt.clf()

    # Plotting memory usage against several minimum support values
    x_axis = np.arange(len(minimum_support))

plt.bar(x_axis,memory_used,color="green",tick_label=minimum_support)
    xlabel = "Minimum Support Values"
```

```
    ylabel = "Memory Usage(Bytes)"
    # addlabels(x_axis,memory_used)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.savefig(f"{output_path}/{ylabel}.png")
    plt.clf()

     # Plotting number of frequent patterns generated against several
minimum support values
    x_axis = np.arange(len(minimum_support))

plt.bar(x_axis,total_number_of_frequent_itemsets,color="blue",tick_la
bel=minimum_support)
    xlabel = "Minimum Support Values"
    ylabel = "Number of frequent patterns mined"
    addlabels(x_axis,total_number_of_frequent_itemsets)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.savefig(f"{output_path}/{ylabel}.png")
    plt.clf()

    with open(f"{output_path}/Memory_taken.csv","w") as f:
        write = csv.writer(f)
        write.writerow(memory_used)

    with open(f"{output_path}/Execution_time.csv","w") as f:
        write = csv.writer(f)
        write.writerow(time_taken)

    count += 1
```

fp_growth.py

```
import logging
import pandas as pd
import numpy as np
import csv
from fp_tree_node import *
from fp_growth import *
from collections import defaultdict

# Setting up a logger
logger = logging.getLogger()  # Initiating a logger
logging.basicConfig(filename="logs.log",
                    format='%(asctime)s %(message)s',
                    filemode='w')
logger.setLevel(logging.DEBUG)

def conditional_pattern_base(fp_tree, itemlist_retrieved,minimum_support):
    """
    Algo for conditional pattern base

    """
     # Fetching items from header table and the corresponding nodes that are linked to the item fetched from header table
    for header_item, corresponding_nodes in fp_tree.fetch_items():
            current_support_value = sum(node.count for node in corresponding_nodes)

         # Checking whether the current support value of the node is greater or equal to the minimum support
```

```
        # Also Need to check whether the item is already retrieved or
not
         if current_support_value >= minimum_support and header_item
not in itemlist_retrieved:
            frequent_itemsets = [header_item] + itemlist_retrieved
            # Returning frequent_itemsets
            yield (frequent_itemsets,current_support_value)

                # Now, we need to traverse through the nodes of the
present itemset
            path_of_parents = fp_tree.fetch_parent_paths(header_item)

              # Once the path of parents are fetched for the present
item, we need to create a new conditional FP tree based on the nodes
present in the path of parents
            conditional_fp_tree = FPTree()
            conditional_item = None

            # We have to add nodes to this conditional FP tree
            for current_parent_path in path_of_parents:
                # Setting the last item in the current parent path as
the conditional item. Conditional item should not be None
                if conditional_item is None:
                    conditional_item = current_parent_path[-1].item

                                      conditional_fp_tree_iterator  =
conditional_fp_tree.get_root

                # Now using this iterator we traverse the conditional
FP tree
                for node in current_parent_path:
```

```
                # Checking whether the node is already present in the conditional FP Tree
                present_item = conditional_fp_tree_iterator.find_node(node.item)

                # If the node is already present in the conditional FP Tree we move the iterator to the present_item and repeat the process till the present_item is None
                if present_item:
                    conditional_fp_tree_iterator = present_item
                    continue

                # If the node is not present in the FP tree, a new node is created and added to the FP tree
                else:

                    # Initializing the count of this new node as 0

                    count_of_new_node = 0
                    if node.item == conditional_item:
                        count_of_new_node = node.count

                    # Creating this new node using fp_tree_node class
                    present_item = fp_tree_node(conditional_fp_tree,node.item,count_of_new_node)
                    conditional_fp_tree_iterator.add_node(present_item)

                    # Now this node has to be updated in the header table
                    conditional_fp_tree.add_to_header_table(present_item)
```

```
                    # After adding the node to the Header table,
initiliazing the conditional fp tree iterator to the present item
node
                conditional_fp_tree_iterator = present_item

            # Once the conditional fp tree is generated, mining for
frequent items starts
                                    for    current_parent_path    in
conditional_fp_tree.fetch_parent_paths(conditional_item):
            current_node_in_parent_path = current_parent_path[-1]
                                    current_support_value    =
current_node_in_parent_path.count
            for node in reversed(current_parent_path[:-1]):
                node.count += current_support_value

                                    for    frequent_item_sets    in
conditional_pattern_base(conditional_fp_tree,frequent_itemsets,minimu
m_support):
            yield frequent_item_sets



def mine_frequent_itemsets(transactions,threshold):

    # Initializing a default dictionary which assigns a value 0 if
that key is not present in the dict
    itemset = defaultdict(lambda:0)

    # Calculating count of attributes from transactional data and
storing them in itemset
    for transaction in transactions:
        for attribute in transaction:
            if attribute is not None:
                itemset[attribute] += 1
```

```
        logger.info("---------------Count of Items stored in a Dictionary---------------")
    # Sorting the itemsets in decreasing order of their count
    try:
        itemset=dict(sorted(itemset.items(), key=lambda item:item[1],reverse=True))
    except Exception as e:
        logger.error(e)

    logger.info("---------------Itemset sorted in decreasing order of their count---------------")

    # Now we have to rearrange the items present in each transaction in the same way the itemsets are arranged
    try:
        for index, transaction in enumerate(transactions):
            transactions[index] = sorted(transaction,key=lambda x:itemset[x],reverse=True)
    except Exception as e:
        logger.error(e)

        logger.info("---------------Items in each transaction are sorted---------------")

    # It is time to build the FP Tree using the items present in the ordered transaction in transactional data
    fp_tree = FPTree()
    try:
        for transaction in transactions:
            if len(transaction) != 0:
                fp_tree.add_items(transaction)
    except Exception as e:
```

```
        return e

    logger.info("---------------FP Tree created---------------")

     # Now we have to mine the frequent patterns from the FP tree that
was created using the items from the transaction data

                                for        frequent_itemsets        in
conditional_pattern_base(fp_tree,[],threshold):
        yield frequent_itemsets
```

fp_tree_node.py

```
from collections import namedtuple

class fp_tree_node(object):
    def __init__(self,tree,item,count=1):
        """
         -This is a constructor class used to define the node in an FP
Tree.
        -Parameters passed:
            -item : Name of the item present in a transaction.
            -count : Number of occurences of that item.
        -Definitions:
            -parent : Link to parent of the current node .
            -children : Link to children of the current node.
              -next_pointer : Link to the same item present in another
branch.
```

```
        """
        self.tree=tree
        self.item=item # data point is item
        self.count=count
        self.parent=None
        self.children={}
        self.next_pointer=None

    def find_node(self,item):
        """
        Function to find the node in which the item is present.
        -Returns corresponding node of the item passed.
        -Parameters Passed:
            -item: An item is passed to find it in the child branch.
        """
        try:
            return self.children[item]
        except:
            return None

    @property
    def get_children(self):
        """
        Function to get the item from FP Tree node
        """
        return tuple(self.children.values())

    def print_node(self):
        """
        Function to print FP tree nodes and its values
        """
```

```
        print(self.item, self.count)
        for child in self.children.keys():
            self.children[child].print_node()

    def print_leaves(self):
        """
        Function to print FP tree leaves and its values
        """
        if len(self.children.keys()) == 0:
            print(self.item, self.count)

        for child in self.children.keys():
            self.children[child].print_leaves()

    def add_node(self,node):
        """
        Function to add a node to a parent
        -Parameters passed:
            -item : An item is passed as a node
        """
        if node.item not in self.children:
            self.children[node.item] = node
            node.parent = self

    @property
    def check_root(self):
        """
        Function to check for branches in an FP Tree
        """
        return self.item is None and self.count is None
```

```
class FPTree(object):

    def __init__(self):

        """

        1. Creating a root node for the FP tree with NULL value.

        2. Creating an empty header table

        """

        self.root = fp_tree_node(self, None, None)

        self.header = {}


    def print_tree(self):

        """

        Function to print nodes present in the FP Tree

        """

        root_node = self.root

        root_node.print_leaves()


    def fetch_nodes(self, item):

        """

         Function to fetch the nodes from the FP tree of the item in
header table

        """

        # Accessing first node from the header table

        node = self.header[item][0]

        if node is None:

            return KeyError

        while node is not None:

            yield node

            node = node.next_pointer


    def add_items(self, transaction):

        """
```

```
        Funtion to add items present in a transaction to the FP tree
        Parameters passed:
            -transaction : ordered transaction
        Functionality:
            1. Take each item from the transaction.
            2. Check whether the item exists in the FP tree
                a. If exists -> Increment count of the node by 1.
                  b. Does not exist -> Create a new node of the item
and add the item with the count of 1 to the FP tree.

        """
        current_node = self.root
        for item in transaction:
            next_node = current_node.find_node(item)
            if next_node is None:
                next_node = fp_tree_node(self,item)
                current_node.add_node(next_node)
                self.add_to_header_table(next_node)
            else:
                next_node.count += 1
            current_node = next_node


    def fetch_items(self):
        """
             Function to return items in the header table and its
corresponding nodes
         Using yield instead of return to use fetch_items function as
a generator so as to store local variables(not executing the function
with a return)
        """
        for item in self.header.keys():
```

```
            yield item, self.fetch_nodes(item)

    def fetch_parent_paths(self, item):
        """
        Function to fetch the parent paths of the present item.
        """
           parent_paths = []  # Use a list to store multiple parent
paths
        for node in self.fetch_nodes(item):
            current_parent_path_of_present_node = []
            while node and not node.check_root:
                current_parent_path_of_present_node.append(node)
                node = node.parent

            # Reverse the parent path to get the correct order
            current_parent_path_of_present_node.reverse()
            parent_paths.append(current_parent_path_of_present_node)

        return parent_paths  # Return a list of parent paths

    Item_Track = namedtuple("Item_Track", "begin end")
    def add_to_header_table(self,present_item):
        """
         Function to add items to the header table which are connected
to the FP Tree
        Paramters passed:
            -present_item: Present node in the FP Tree
        """
        present_node = present_item.item
        try:
            # Path to the present_item
```

```
            present_path = self.header[present_item.item]
            present_path[1].next_pointer = present_item
                                    self.header[present_item.item]   =
self.Item_Track(present_path[0], present_item)
        except:
                                    self.header[present_item.item]   =
self.Item_Track(present_item, present_item)

    @property
    def get_root(self):
        return self.root
```

plot_utils.py

```
import matplotlib.pyplot as plt

def addlabels(x,y):
    for i in range(len(x)):
        plt.text(i,y[i],y[i])
```