**East West University**

**Course title: Data Mining**

**Course code: CSE477**

**Fall 2023**

**Section: 2**

**“Mini Project Report”**



**"** **Finding frequent patterns (Apriori) on Streaming Data Using incremental-window based mining"**

**Submitted To:**

**Dr. Md Mostofa Kamal Rasel**

Assistant Professor,

Department of Computer Science and Engineering

East West University, Dhaka

**Submitted By:**

| **Name** | **ID** | **Section** |
| --- | --- | --- |
| **Md Ashraful Hasan** | **2019-1-60-219** | **2** |
| **Ashraful Reza Tanjil** | **2019-1-60-221** | **2** |
| **Fahim Md Adud** | **2019-1-60-223** | **2** |
| **Md. Iftekhar Hossain** | **2020-1-60-106** | **2** |

Department of Computer Science and Engineering

East West University, Dhaka

Date of Submission: 25/ 12 / 2023

**Title: Finding Frequent Patterns (Apriori) on Streaming Data Using incremental-window based mining**

# **Project Description:**

This mini project focuses on window-based mining using the Apriori algorithm in Python for frequent pattern mining. Apriori is a fast way to find frequent patterns in big datasets. This method requires candidate generation. This project aims to develop a Python program that takes a Streaming dataset as input and mines the frequent patterns using the Apriori algorithm.

This project has a few important steps:

1. Installation:

The parameters min\_support and window\_size are set when the IncrementalApriori class is created.

It keeps a sliding window of transactions and a list of frequently used itemsets.

2. Candidate Identification:

The generate\_candidates function generates candidate itemsets from the current window's transactions.

3. Counting Support:

The count\_support method counts the amount of support for each candidate itemset in the current transaction window.

4. Frequently Updated Itemset:

The update\_frequent\_itemsets method modifies the set of often occurring itemsets depending on the support counts and the minimum support threshold.

5. Sliding Glass Door:

If the window size exceeds the set limit, the slide\_window method refreshes the window by adding a new transaction and removing the oldest transaction.

6. Mining Apriori Incremental:

The mine\_incremental\_apriori technique combines the operations of moving the window, producing candidates, counting support, and updating frequent itemsets into a single step.

After processing the new transaction, it returns the current set of frequent itemsets.

7. Reading Datasets:

The read\_binary\_dataset method reads and converts a binary dataset into a DataFrame.

8. Assessment:

The main loop loops over various combinations of min\_support and window\_size.

It processes several datasets with streaming transactions for each combination.

Every tenth iteration, it assesses the algorithm's performance against a reference set (ground truth) using metrics such as Jaccard Similarity, Precision, and Recall.

Paper Review :

1. The paper elegantly illustrates the concept of a sliding window in managing continuous data streams, likening it to a port handling incoming cargo. It effectively conveys the challenge of processing and storing data arriving persistently, emphasizing the necessity of timely data processing akin to managing cargo efficiently within limited space. This analogy provides a vivid understanding of the sliding window's role in handling streaming data within constrained environments.(Mining frequent itemsets from streaming transaction data using genetic algorithms)
2. The paper's exploration of sliding window configurations in the context of the FUFP-Tree algorithm is insightful. It illuminates how altering the number of sliding windows impacts runtime and the efficient construction of the FUFP-Tree. This analysis highlights the algorithm's adaptability to handle varying data loads, emphasizing its potential for real-time processing in streaming data environments.( Efficient streaming data association rule mining)

# **Dataset Description:**

The data set was generated by a data set generator. The data set generation system was taught by our course instructor. Binary stream data was generated by the data generator.

We create 4 dataset

1st:

Number of transactions is set to 100

Transaction length is set to 10

The percentage of items in each transaction is set to 5%

2nd:

Number of transactions is set to 200

Transaction length is set to 10

The percentage of items in each transaction is set to 5%

3rd:

Number of transactions is set to 1000

Transaction length is set to 10

The percentage of items in each transaction is set to 10%

4th:

Number of transactions is set to 2000

Transaction length is set to 10

The percentage of items in each transaction is set to 10%

# **Dataset pre-processing:**

The data set was generated by us. Binary streaming data was used.

# **Implementation:**

The Apriori algorithm was implemented in Python language. Jupyted Notebook software is used for this. The configuration of the computer is Intel i5 6th gaming CPU, 8GB DDR5 (5200MHz) RAM, RTX 1060 GDDR5 GPU, and Windows 10 operating system.

# **Result Analysis:**

The algorithm's results are saved in the results list, which includes parameters, dataset information, iteration, frequent item sets, and evaluation measures.

In short, the algorithm is an incremental version of the Apriori algorithm that discovers frequent item sets quickly in a streaming scenario, updating its findings as new transactions arrive. The evaluation metrics show how well the algorithm performs in terms of accuracy and completeness when compared to a reference set.

# **Discussion:**

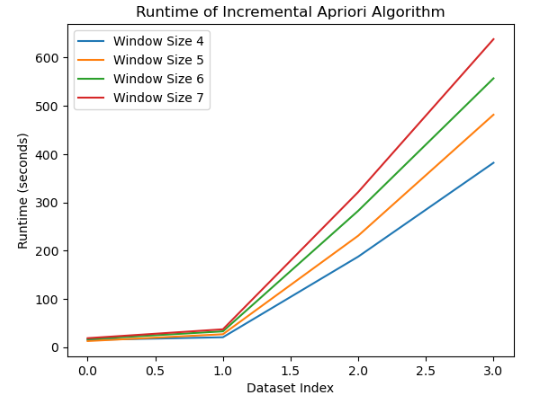


Fig: 1 Data set index and their run time.

The comparison of the line charts reveals the following differences between the windows considering the Apriori algorithms:

1. Incremental Processing:

• Suitable for instances with streaming data in which transactions are received incrementally.

• Effectively refreshes the frequently used item sets when new transactions arrive.

2. Memory Efficiency: Maintains a sliding window of transactions, which can save memory as compared to maintaining the whole transaction history.

3. Adaptability to Varying Support Thresholds:

• Can adjust to changes in the minimum support threshold during different iterations.

• The algorithm is relatively easy to understand and implement.

# **Limitations:**

**1. High Computation Cost:**

The approach may have high computational costs, particularly when dealing with a large number of transactions and candidate item sets.

**2. Window Size Sensitivity:**

The algorithm's performance is sensitive to the window size selected, and an incorrect window size may reduce its effectiveness.

**3. No Transaction Deletions Handled:**

The method concentrates on additions to the window but does not handle deletions. In some programs, data must be withdrawn from the window.

# **Suggestions:**

1. **Lack of User-Friendly Interface:** This code is more of a script than a user-friendly application. In practice, users may prefer tools with graphical interfaces or command-line options for better usability.
2. **Performance Optimization:** While the code works correctly, there may be opportunities to optimize it further for better performance, especially when dealing with large datasets.
3. **Efficiency and Scalability:** Although the Apriori algorithm is generally efficient for frequent pattern mining, the code may not be optimized for large or complex datasets.

# **Code:**

import pandas as pd

import numpy as np

from collections import Counter

class IncrementalApriori:

def \_\_init\_\_(self, min\_support, window\_size):

self.min\_support = min\_support

self.window\_size = window\_size

self.window = []

self.frequent\_itemsets = set()

def generate\_candidates(self, transactions):

candidates = set()

for transaction in transactions:

for item in transaction:

candidates.add(frozenset([item])) # Each item is a candidate

return candidates

def count\_support(self, transactions, candidates):

support\_counts = Counter()

for transaction in transactions:

for candidate in candidates:

if candidate.issubset(transaction):

support\_counts[candidate] += 1

return support\_counts

def update\_frequent\_itemsets(self, support\_counts):

frequent\_itemsets = set()

for itemset, count in support\_counts.items():

if count >= self.min\_support:

frequent\_itemsets.add(itemset)

return frequent\_itemsets

def slide\_window(self, new\_transaction):

self.window.append(new\_transaction)

if len(self.window) > self.window\_size:

self.window.pop(0)

def mine\_incremental\_apriori(self, new\_transaction):

self.slide\_window(new\_transaction)

candidates = self.generate\_candidates(self.window)

support\_counts = self.count\_support(self.window, candidates)

self.frequent\_itemsets = self.update\_frequent\_itemsets(support\_counts)

return self.frequent\_itemsets

# Function to read the binary dataset and convert it into a DataFrame

def read\_binary\_dataset(filename):

with open(filename, 'rb') as f:

data = np.fromfile(f, dtype=np.uint32)

index = 0

N = len(data)

transactions = []

while index < N:

tran\_id = data[index]

items\_count = data[index + 1]

s\_index = index + 2

e\_index = s\_index + items\_count

items = data[s\_index:e\_index]

index = e\_index

transactions.append(items)

encoder = TransactionEncoder()

transactions\_e = encoder.fit(transactions).transform(transactions)

df = pd.DataFrame(transactions\_e, columns=encoder.columns\_)

return df

# Placeholder function, replace with actual reference data

def get\_reference\_itemsets\_for\_iteration(iteration):

# This function should return the reference set of itemsets for the given iteration

# Replace this with your actual reference data or ground truth

return set()

# Additional dataset filenames

dataset\_filenames = [

'G:\\D\_Generator\\db\_1000000\_50\_10.data',

'G:\\D\_Generator\\db\_2000000\_50\_10.data',

'G:\\D\_Generator\\db\_10000000\_100\_10.data',

'G:\\D\_Generator\\db\_20000000\_100\_10.data',

]

# Additional window sizes

window\_sizes = [4, 5, 6, 7]

# Create a list to store the results

results = []

for min\_support in [2]: # You can adjust the minimum support if needed

for window\_size in window\_sizes:

# Initialize IncrementalApriori for each combination of parameters

apriori = IncrementalApriori(min\_support, window\_size)

# Process each dataset

for dataset\_filename in dataset\_filenames:

stream\_data = read\_binary\_dataset(dataset\_filename)

# Simulate streaming data

for i, transaction in enumerate(stream\_data.values, 1):

frequent\_itemsets = apriori.mine\_incremental\_apriori(transaction)

if i % 10 == 0:

# Evaluate accuracy measures using a reference set if available

reference\_itemsets = get\_reference\_itemsets\_for\_iteration(i)

# Calculate Jaccard Similarity

common\_itemsets = frequent\_itemsets.intersection(reference\_itemsets)

total\_unique\_itemsets = frequent\_itemsets.union(reference\_itemsets)

jaccard\_similarity = len(common\_itemsets) / len(total\_unique\_itemsets)

# Calculate Precision and Recall

precision = len(common\_itemsets) / len(frequent\_itemsets) if frequent\_itemsets else 0

recall = len(common\_itemsets) / len(reference\_itemsets) if reference\_itemsets else 0

# Print or store results as needed

results.append({

'min\_support': min\_support,

'window\_size': window\_size,

'dataset': dataset\_filename,

'iteration': i,

'frequent\_itemsets': frequent\_itemsets,

'jaccard\_similarity': jaccard\_similarity,

'precision': precision,

'recall': recall

})

Windows:

A screenshot of a computer code

Description automatically generated

For window 4

A screenshot of a computer code

Description automatically generated

For window 6

A screenshot of a computer code

Description automatically generated

For window 7