CSOE18 – Big Data Analytics

Name:-	Kshitij Kanade
Roll No:-	107121045
Date:-	08/10/2023
Batch:-	2025
Dept:-	EEE Sec A

Assignment - 1

Frequent Pattern Mining

Aim/Objective:-

The objective of this project is to learn the concepts of Frequent Pattern Mining and to implement it for a real-time dataset without using any Python libraries.

External Links:-

WebPage Report

Readme.txt - a detailed guide to run the code in the given file.

Categories.txt - Dataset.

Colab file - Contains the code implementations of the project.

Drive folder - Contains All required files.

Summary of Results:-

The real-time implementation of Frequent Pattern Mining using the Apriori Algorithm in Python applied to the provided dataset without relying on any built-in libraries was successfully executed and the output was rigorously verified. Three output files, named "patterns_1.txt," "patterns_all.txt," and "patterns_close.txt," were generated with precision. These files contain Frequent Itemsets of Length One, All Frequent Itemsets, and Closed Frequent Itemsets, each presented in descending order of Category: Counts.

In addition to the given dataset, my algorithm also demonstrated excellent performance when applied to other datasets on platforms such as Kaggle, including Market Basket Analysis and Groceries Association Rule datasets. Although some adjustments were needed to adapt to these different datasets, the underlying logic of the algorithm remained consistent. The references to these datasets are included for transparency, as I drew upon them to develop my project. The key distinction lies in the fact that I wrote the code from scratch, relying solely on Python without

utilizing any external libraries. This approach allowed for a deeper understanding of the algorithm and its inner workings, reinforcing my proficiency in Python.

Algorithmic Optimizations Implemented:-

Custom Functions for Efficiency:

I. I designed specialized file-handling functions to eliminate repetitive code, enhancing readability and maintainability.
   - These functions facilitated reading data from files, writing tuples to files, and printing the contents of files.

II. The utilization of Merge Sort significantly improved sorting efficiency.

III. Itemset Generation:
   - I employed count and filter mechanisms for itemset management, enhancing overall performance.
   - I successfully generated candidate itemsets of a specified length.
   - By efficiently calculating counts from the data for a given dictionary, I minimized redundancy.
   - I optimized the mining process for closed frequent itemsets.

Use of Appropriate Data Types:

- I leveraged Python data types such as *frozenset*, which prioritizes the elements within a set without regard to order. This was valuable for maintaining consistency in itemset representation.
- In addition to frozensets, I effectively harnessed dictionaries and lists, including lists of lists of lists, to streamline data manipulation and analysis.

The GET Method:

- In the conventional approach, the dataset is scanned once to create keys and then scanned again to update their values, which can be less efficient.
- Python's "GET Method" efficiently initializes dictionary keys with a default value of 0, a departure from the conventional two-step approach.

- The "GET Method" simplifies this process by allowing simultaneous key initialization and value updates, resulting in a more streamlined and optimized workflow.

Use of Attributes/Methods Data Types:

- Various data type methods and attributes, including "issubset," "append," "isinstance," and "union," were applied to streamline code execution.

Experiences & Lessons:-

The implementation of an algorithm in Python that could be distilled into just four lines of code initially appeared straightforward. However, as the project progressed, it evolved into a profound exploration of the language's intricacies and the inner workings of the Apriori Algorithm. It underscored the pivotal role of built-in libraries in simplifying complex tasks. This journey proved to be a valuable lesson as it deepened my comprehension of both the algorithm and the Python language itself. Notably, the execution time, which initially required 5-6 hours based on a brute force approach, was progressively refined, optimized, and reduced to a **mere 25 seconds whereas the library approach takes at least 40 seconds.**

Further Optimizations:-

- The **"generate_itemsets_of_length_n"** function could be further optimized using advanced concepts, such as *Dynamic Programming with recursive calls*, to efficiently construct itemsets of higher lengths.
- Implementing a main function that orchestrates the sequential generation and updating of frequent itemsets using a single recursive function would enhance overall code structure and performance.