**CS634-Data Mining**

**Midterm Project Report**

**Student Name: Rajeev Kumar Alahari**

**Email: ra833@njit.edu**

**Course: CS634-Data Mining**

**Instructor: Dr. Yasser Abduallah**

**1 Introduction**

This report focuses on implementing and comparing three key methods for frequent itemset mining and association rule learning, a core concept in data mining. These methods aim to uncover hidden relationships between items in transaction datasets to support decision-making and recommendation systems.

**Brute Force:**

The brute-force approach was implemented from scratch to demonstrate the basic working principle of frequent itemset mining. It systematically generates all possible item combinations and checks their frequency in the dataset. Though simple and educational, it is computationally expensive and inefficient for large datasets.

**Apriori (mlxtend):**

The Apriori algorithm was implemented using the `mlxtend` Python library, which efficiently prunes unpromising itemsets based on the Apriori property. It uses a level-wise search, reducing the number of candidate sets and improving performance compared to brute force. However, it can still be slow with very dense or large datasets.

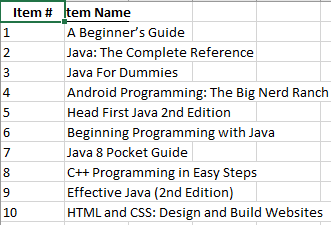
**FP-Growth (mlxtend):**

The FP-Growth algorithm, also implemented using `mlxtend`, overcomes Apriori’s candidate generation limitation by compressing transactions into a compact FP-Tree structure. It recursively extracts frequent itemsets from the tree, making it significantly faster and more scalable. This method is ideal for large-scale data mining applications.

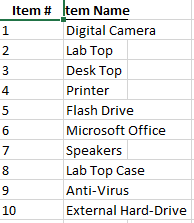
**2 Dataset Creation**

**2.1 Items:**

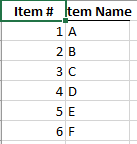
**Amazon Items:**



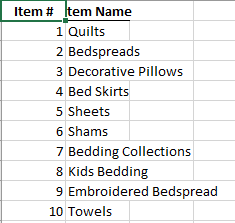
**BestBuy items:**



**Generic Items:**



**Kmart Items:**

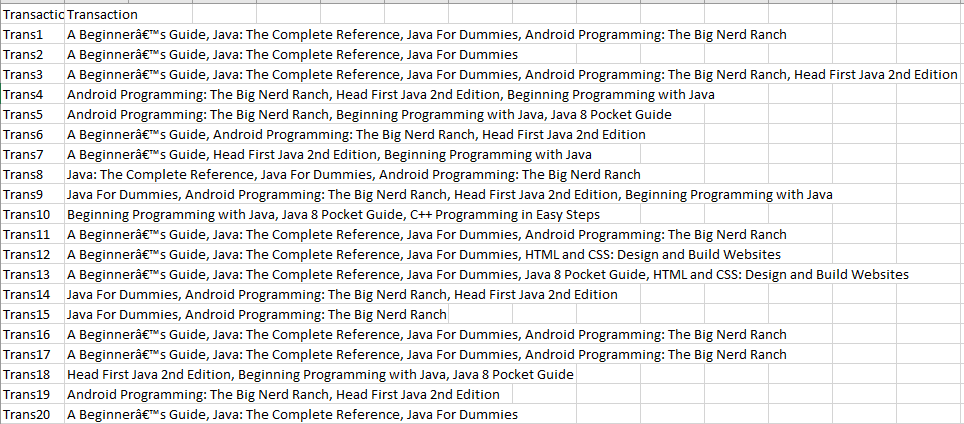


**Nike Items:**

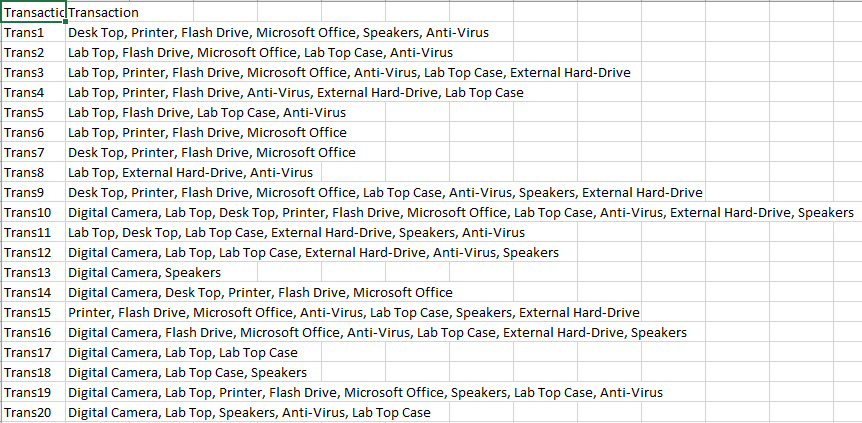


**2.2 Transactions:**

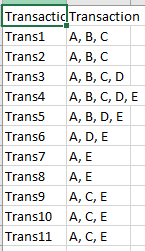
**Amazon Transactions:**



**BestBuy Transactions**



**Generic Transactions**



**Kmart Transactions**



**Nike Transactions**



**2.3 Dataset notes:**

I have extracted the professor’s dataset and converted to csv format and saved those files.

**3 Brute Force Algorithm**

**3.1 Method:**

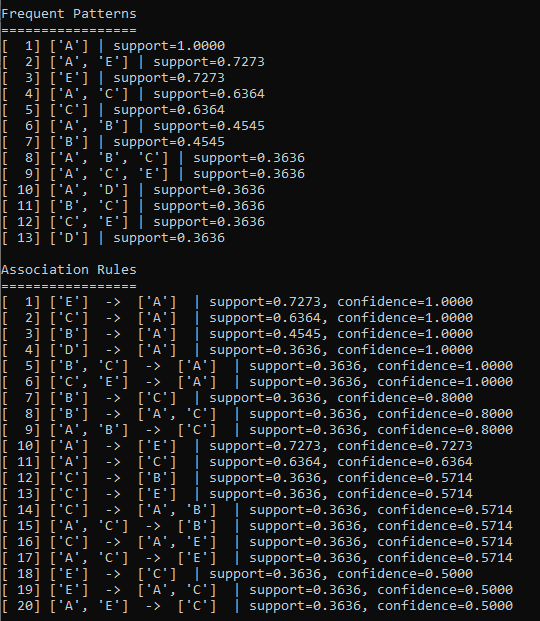
1. The brute-force method systematically generates all possible combinations of items from the dataset.
2. It then checks each combination’s occurrence across all transactions to calculate its support value.
3. Only itemsets meeting the minimum support threshold are considered frequent.
4. After finding frequent itemsets, association rules are generated by computing confidence for all possible splits.
5. Though conceptually simple, this approach becomes computationally infeasible for large datasets due to exponential growth of combinations.

**3.2 Run:**

**Dataset:** Generic\_Transactions.csv

**Parameters**: Support=0.3, Confidence=0.5

**Output:**



**4 Apriori and FP-Growth**

**4.1 Apriori Algorithm**

1. Apriori improves efficiency by applying the Apriori principle — “if an itemset is frequent, all its subsets must be frequent.”
2. It begins by finding frequent 1-itemsets, then iteratively extends them to k-itemsets using only previously frequent ones.
3. At each iteration, infrequent itemsets are pruned early, significantly reducing the search space.
4. The algorithm scans the database multiple times to calculate supports and identify qualifying itemsets.
5. Finally, association rules are derived from the frequent itemsets that meet both support and confidence thresholds.

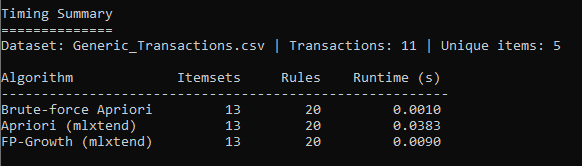
**Results:** Same as Brute Force.

**4.2 FP-Growth Algorithm**

1. FP-Growth eliminates candidate generation by using an efficient tree-based structure (FP-Tree) to store transactions.
2. It first scans the database to identify frequent items and orders them by descending frequency.
3. Each transaction is then inserted into the FP-Tree, sharing common prefixes to compress the dataset.
4. The algorithm recursively mines conditional FP-Trees to discover frequent patterns directly.
5. As it avoids multiple database scans, FP-Growth achieves much faster performance on large and dense datasets.

**Results:** Same as Brute Force.

**4.3 Timing Comparison**



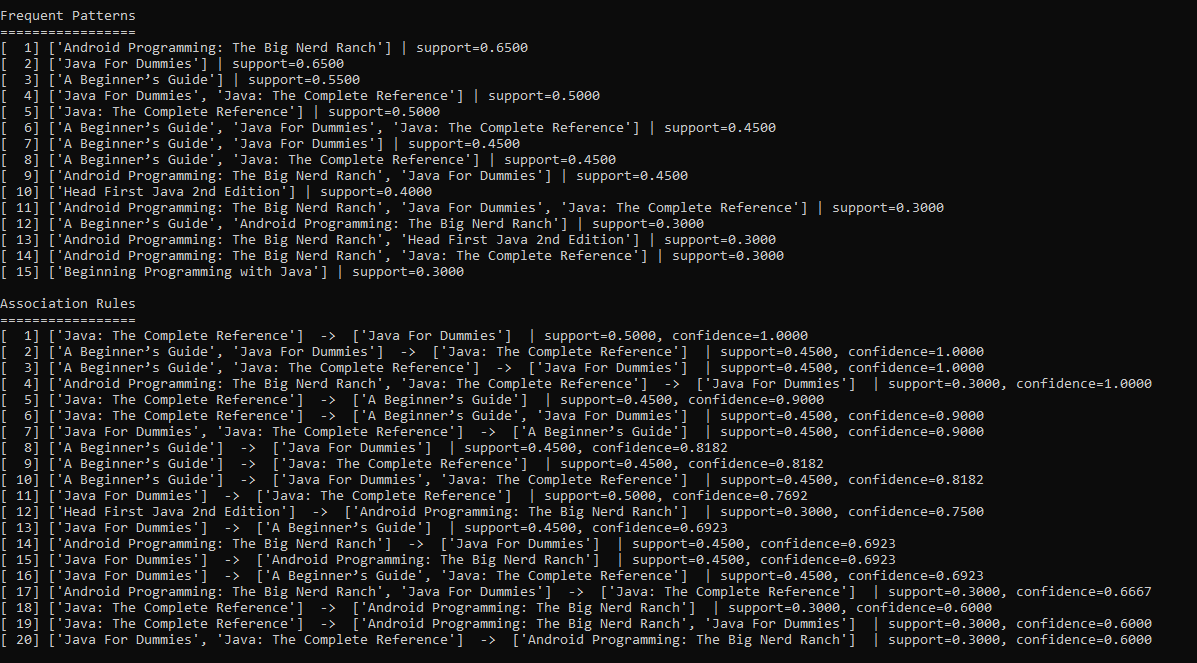
Based on the results we can say that brute force here runs quicker than the apriori and

Fp-growth because here the dataset we are dealing with is very small in size.

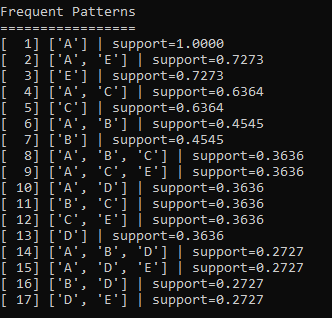
For large datasets the mlxtend versions of algorithms tends to perform better with good scalability.

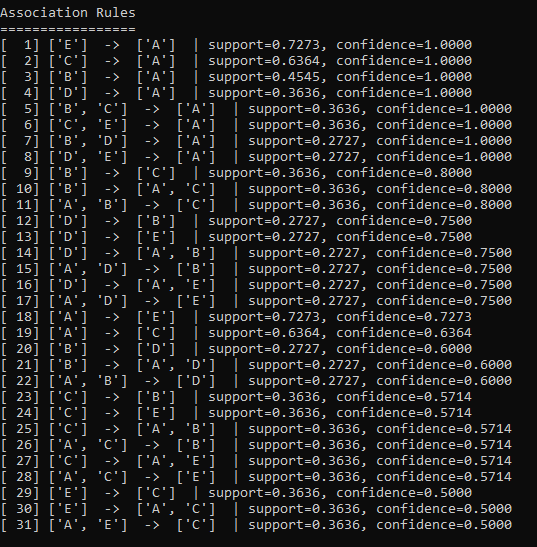
**5 Multiple Parameters**

**- Support = 0.3, Confidence = 0.6 for Amazon Dataset**

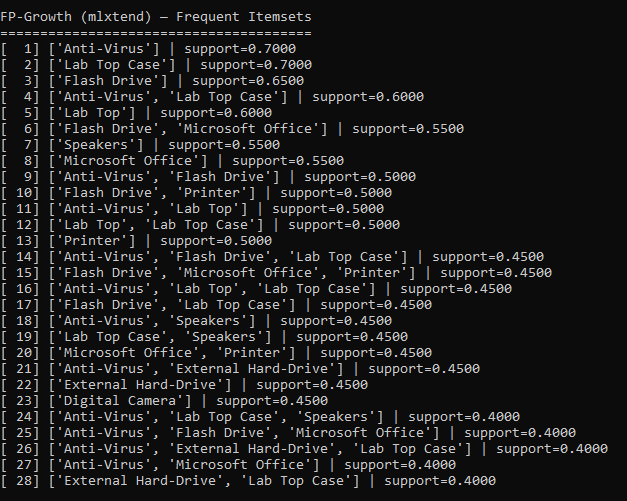


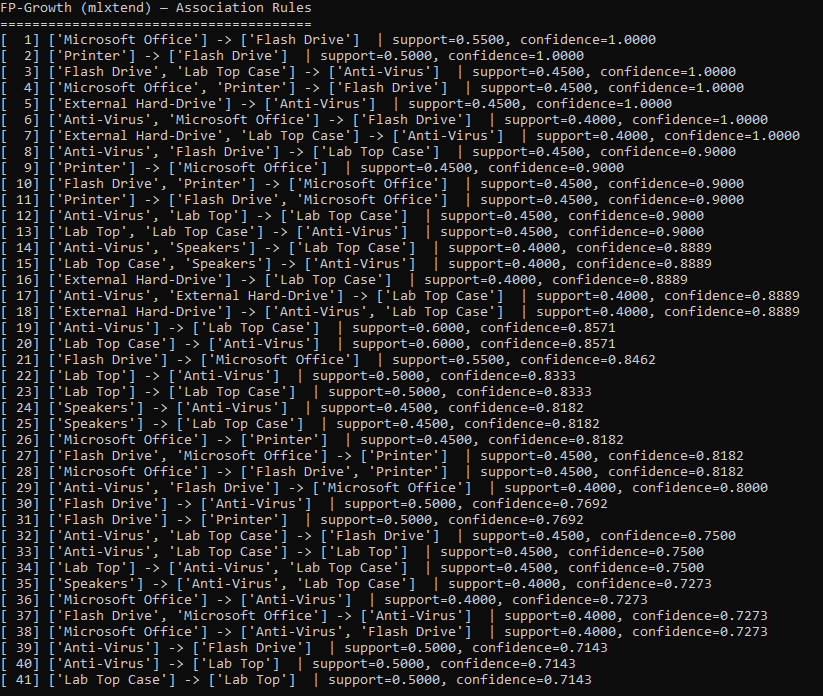
**- Support = 0.2, Confidence = 0.5 for Generic Dataset**





**- Support = 0.4, Confidence = 0.7 for BestBuy Dataset**





**6 GitHub Repository**

**Link:**

**7 How to run the code**

**7.1 Install Requirements:**

1. The basic requirements needed are pandas and mlxtend.

pip install pandas

pip install mlxtend

(OPTIONAL)

1. If you face any issues with requirements I have frozen the requirements in my machine
2. You can install them by changing directory to alahari\_rajeevkumar\_midtermproject folder and run

pip install -r requirements.txt

1. If you don’t want to install requirements in our entire system and want to create a virtual environment and install requirements in it and use it for the program execution.

python -m venv venv

venv\Scripts\activate

this creates virtual environment and activates it in windows.

**7.2 Run Options**

**PYTHON FILE**

1. Extract the file:

Unzip “alahari\_rajeevkumar\_midtermproject.zip” to a folder on your computer.

1. Open Command Prompt (Windows) or Terminal (macOS/Linux).
2. Change into the src folder
3. Run the program:

Windows: python code.py

macOS/Linux: python3 code.py

1. Main menu options:

Enter 1–5 to select a dataset (each number is a different dataset).

Enter 0 to Exit.

1. After selecting a dataset:

You will be asked for support (e.g., 0.30) and confidence (e.g., 0.40).

The program generates frequent itemsets and association rules using:

Brute Force

Apriori (mlxtend)

FP-Growth (mlxtend)

It also prints timing comparisons across all three approaches.

1. Run again or exit:

When prompted, type:

y → returns to the home page (run again)

n → exits the program

**IPYNB FILE**

1. Open your notebook tool (Jupyter notebook).
2. Change into the notebook folder or open the notebook from there:
3. Run the notebook cell by cell from top to bottom.
4. Behavior is the same as the Python CLI:

You’ll get prompts for support and confidence.

Frequent itemsets, association rules, and timing comparisons are displayed.

**RUN FROM GITHUB**

1. Clone the repository:

git clone "repo name"

1. Enter the project folder:
2. Follow the same steps as above:

For python file: cd into src and run python code.py (or python3 on macOS/Linux).

For Notebook: cd into notebook and run the notebook cell by cell.

**8 Conclusion**

This project demonstrated and compared three core approaches to frequent itemset mining and association rule learning Brute Force (from scratch), Apriori, and FP-Growth (via `mlxtend`)—across multiple retail-style datasets. Brute Force served as a transparent baseline, showing the mechanics of exhaustive candidate generation and why the search space grows exponentially. Apriori improved practicality by pruning with the downward-closure property, while FP-Growth avoided candidate generation entirely through an FP-Tree, yielding the most scalable path for larger or denser datasets. On our relatively small datasets, all three approaches produced consistent frequent itemsets and rules, with timing differences modest; however, the algorithms’ theoretical strengths indicate that Apriori and especially FP-Growth are the preferred choices for real-world, large-scale mining.

From an engineering standpoint, care was taken to make the command-line workflow reliable and user-friendly. Defensive programming techniques include strict input validation for the dataset menu (accepting only 0–5, with clear re-prompts), numeric checks for support and confidence (enforcing valid floats in the [0, 1] range), and loop-based prompts that gracefully handle mistakes without crashing. Additional safeguards—such as try/except blocks, file/path existence checks, and clean exit options—ensure that the program fails safely, guides the user to correct inputs, and supports repeated runs from a single session. Together, these design choices make the implementation both pedagogically clear and robust enough for exploratory analysis.