

Assignment-3 Report

Team - 30

Dataset Explanation for Market Basket Analysis:

We used three main CSV files from the “Instacart Market Basket Analysis” dataset, each representing a different aspect of customer purchasing behavior.

1) **orders.csv** - *Customer Purchase History*

Purpose: This file contains details about every order placed by each user. It describes who ordered, when, and in which sequence the order occurred.

| Column | Description |
|------------------------|---|
| order_id | Unique ID assigned to each order. |
| user_id | ID of the user who placed that order. |
| eval_set | Indicates the type of order: 'prior', 'train', or 'test'. |
| order_number | Sequential number of the order for that user (1 = first order). |
| order_dow | Day of week the order was placed (0=Sunday, 6=Saturday). |
| order_hour_of_day | Hour of the day (0–23) when the order was placed. |
| days_since_prior_order | Number of days since the user’s previous order (NaN for first order). |

eval_set in **orders.csv**

The column `eval_set` in the file `orders.csv` tells us which group (or subset) each order belongs to.

There are three possible values in this column:

'prior', 'train', and 'test'.

These are evaluation splits — used to separate the dataset into past orders, training orders, and testing orders for model validation.

For each user:

- 'prior' orders = past behavior
- 'train' order = what model learns from
- 'test' order = final test to check accuracy

2) **order_products__train.csv** - *Products Purchased per Order*

Purpose: This file lists which products were included in each order.

It links `order_id` from `orders.csv` to specific `product_ids` from `products.csv`.

| Column | Description |
|--------------------------------|---|
| <code>order_id</code> | The order this product belongs to. |
| <code>product_id</code> | The specific product purchased in that order. |
| <code>add_to_cart_order</code> | The sequence in which the item was added to the cart. |
| <code>reordered</code> | 1 if the user has bought this product before, else 0. |

3) **products.csv** - *Product Catalog*

Purpose: This is a reference file that provides human-readable names and categories for each `product_id`.

| Column | Description |
|----------------------------|---|
| <code>product_id</code> | Unique ID of each product. |
| <code>product_name</code> | Descriptive product name. |
| <code>aisle_id</code> | Aisle (in the store) where the product is located. |
| <code>department_id</code> | Department category (e.g., dairy, beverages, snacks). |

Data Processing Steps:

1. Loading the Dataset

- **Files Loaded:**
 - **orders.csv:** Contains order metadata, including `order_id` and `eval_set` (train/test).
 - **order_products__train.csv:** Contains product IDs for orders in the training set.
 - **products.csv:** Maps `product_id` to `product_name`.

2. Filtering Training Orders

- **Objective:** Extract only the orders marked as `train` in the `orders.csv` file.
- **Steps:**
 1. Filter rows in `orders` where `eval_set == 'train'`.
 2. Extract the `order_id` values for these rows.
 3. Convert the `order_id` values into a set for faster filtering.

```
train_order_ids = set(orders[orders['eval_set'] ==  
'train']['order_id'])
```

3. Filtering Product Data for Training Orders

- **Objective:** Keep only the rows in order_products__train.csv that correspond to the training orders.
- **Steps:**
 1. Use the train_order_ids set to filter rows in order_products.
 2. This ensures that only products from training orders are retained.

```
op_train =  
order_products[order_products['order_id'].isin(train_order_ids)]
```

4. Grouping Products into Transactions

- **Objective:** Group products by order_id to create a list of transactions.
- **Steps:**
 1. Group the op_train dataframe by order_id.
 2. Aggregate the product_id values into lists for each order.
 3. Convert the grouped data into a list of transactions.

```
transactions =  
op_train.groupby('order_id')['product_id'].apply(list).tolist()
```

5. Removing Duplicate Products Within Transactions

- **Objective:** Ensure that each product appears only once in a transaction while preserving the order.
- **Steps:**
 1. Use dict.fromkeys() to remove duplicates while maintaining the order of items in each transaction.

```
transactions = [list(dict.fromkeys(tx)) for tx in  
transactions]
```

6. Splitting Data into Training and Testing Sets

- **Objective:** Split the transactions into training (80%) and testing (20%) subsets.
- **Steps:**
 1. Use `train_test_split` from `sklearn` to split the transactions list.
 2. Set `test_size=0.2` to allocate 20% of the data to the test set.
 3. Use `random_state=42` for reproducibility.

```
train_tx, test_tx = train_test_split(transactions,  
test_size=0.2, random_state=42)
```

Algorithm Selection: FP-Growth vs Apriori

Selected Algorithm: **FP-Growth**

Justification for Choosing FP-Growth Over Apriori:

1. Computational Efficiency

- FP-Growth: Requires only 2 database scans – one to count item frequencies and another to build the FP-tree.
- Apriori: Requires multiple scans ($k+1$ scans for k -itemsets), which is computationally expensive for large datasets.
- Impact: With this grocery dataset containing thousands of transactions, FP-Growth significantly reduces I/O overhead.

2. Memory Usage and Data Structure

- FP-Growth: Uses a compact FP-tree structure that stores the entire database in a compressed form.
- Apriori: Generates and stores all candidate itemsets explicitly in memory.
- Advantage: FP-tree eliminates redundant information and shares common prefixes, leading to better memory utilization.

3. Candidate Generation

- FP-Growth: Employs a divide-and-conquer approach with no explicit candidate generation.

- Apriori: Uses a generate-and-test approach, creating numerous candidate itemsets that may not be frequent.
- Benefit: FP-Growth avoids expensive candidate generation, directly mining frequent patterns.

4. Scalability with Dataset Size

- FP-Growth: Performance scales linearly with database size due to the tree-based approach.
- Apriori: Performance degrades exponentially with increasing dataset size and number of frequent items.
- Real-world Impact: Our Instacart dataset with 131,209 transactions benefits significantly from FP-Growth's scalability.

5. Pattern Length Independence

- FP-Growth: Efficiently handles both short and long frequent patterns.
- Apriori: Struggles with long patterns due to exponential candidate set growth.
- Relevance: Grocery shopping often involves varied basket sizes, making FP-Growth more suitable.

6. Implementation Complexity vs Performance Trade-off

- FP-Growth: More complex to implement but provides substantial performance gains.
- Apriori: Simpler conceptually but inefficient for real-world applications.
- Decision: The performance benefits justify the implementation complexity for our market basket analysis.

7. Specific Advantages for Our Use Case

- Large Transaction Volume: 131,209 training transactions favor FP-Growth's efficiency.
- Real-time Requirements: Recommendation systems need fast rule mining, making FP-Growth ideal.
- Memory Constraints: FP-tree's compact representation is crucial for large-scale deployment.

Conclusion:

FP-Growth's superior computational efficiency, memory optimization, and scalability make it the optimal choice for our grocery recommendation system.

While Apriori offers conceptual simplicity, FP-Growth's performance advantages are essential for handling real-world e-commerce datasets and supporting practical recommendation applications.

Algorithm Overview:

1. FP-Tree Node (**FPNode**)

- Each node represents one item in a transaction.
- Attributes:
 - item: Name of the item (e.g., 'milk').
 - count: Number of transactions that pass through this node.
 - parent: Pointer to the parent node in the tree.
 - children: Dictionary mapping child items → FPNode.
 - node_link: Pointer to the next node in the tree with the same item (used to traverse all occurrences of an item efficiently).
- Methods:
 - increment(count=1): Adds to the count when the item appears in another transaction along the same path.

Purpose: Represents each element in the FP-tree compactly, allowing hierarchical aggregation of transactions.

2. FP-Tree Construction (**FPTree**)

Input:

- transactions: List of transactions, each is a list of items.
- min_support_count: Minimum number of times an item must appear to be considered frequent.

Step 1: Count item frequencies

- Iterate over all transactions.

- Count the number of times each item occurs.
- Only items with count \geq min_support_count are considered frequent.

Step 2: Build header table

- Stores each frequent item:
 - 'count': total frequency
 - 'head': pointer to the first node in the FP-tree for this item
- Header table allows fast access to all nodes for any frequent item.

Step 3: Insert transactions

- For each transaction:
 1. Filter items to keep only frequent items.
 2. Sort by descending frequency (most frequent items first) → ensures compact tree.
 3. Recursively insert items into the tree (_insert_transaction):
 - If the node already exists as a child, increment its count.
 - Otherwise, create a new node and link it via node_link in the header table.

Purpose: Creates a compressed FP-tree, sharing common prefixes among transactions.

3. Conditional Pattern Base (**get_prefix_paths**)

- For a given item:
 1. Traverse all nodes for that item using node_link.
 2. For each node, collect the path from that node to root (excluding root).
 3. Store the path along with the node count.

Purpose: This produces the conditional pattern base needed to recursively mine frequent itemsets for that item.

4. Mining Frequent Itemsets (**fp_growth**)

Steps:

1. Start with the full FP-tree.
2. Sort items by increasing frequency.
3. For each item:
 - Form a new frequent itemset by combining it with the current suffix.
 - Add it to the list of frequent_itemsets.
 - Build a conditional FP-tree for this item using its prefix paths.
 - Recursively mine the conditional tree.

Purpose: Finds all frequent itemsets without generating candidate sets, making FP-Growth more efficient than Apriori.

5. Generate Association Rules (**generate_association_rules_CORRECT**)

Inputs:

- frequent_itemsets, transactions, min_confidence

Steps:

1. Build a dictionary of itemset \rightarrow support count.
2. For each frequent itemset of size ≥ 2 :
 - Generate rules with single-item antecedents ($A \rightarrow B$).
 - Compute:
 - $\text{confidence} = \text{support}(A \cup B) / \text{support}(A)$
 - $\text{lift} = \text{support}(A \cup B) / (\text{support}(A) * \text{support}(B))$
3. Only keep rules that meet min_confidence.

Purpose: Extract meaningful rules with strong associations.

6. Applying FP-Growth

- Choose thresholds:
 - $\text{min_support} = 0.003 \rightarrow$ filters very rare items.
 - $\text{min_confidence} = 0.06 \rightarrow$ ensures rules have reasonable strength.
- Mine frequent itemsets and generate rules.
- Filter rules to single-item antecedents for simplicity.

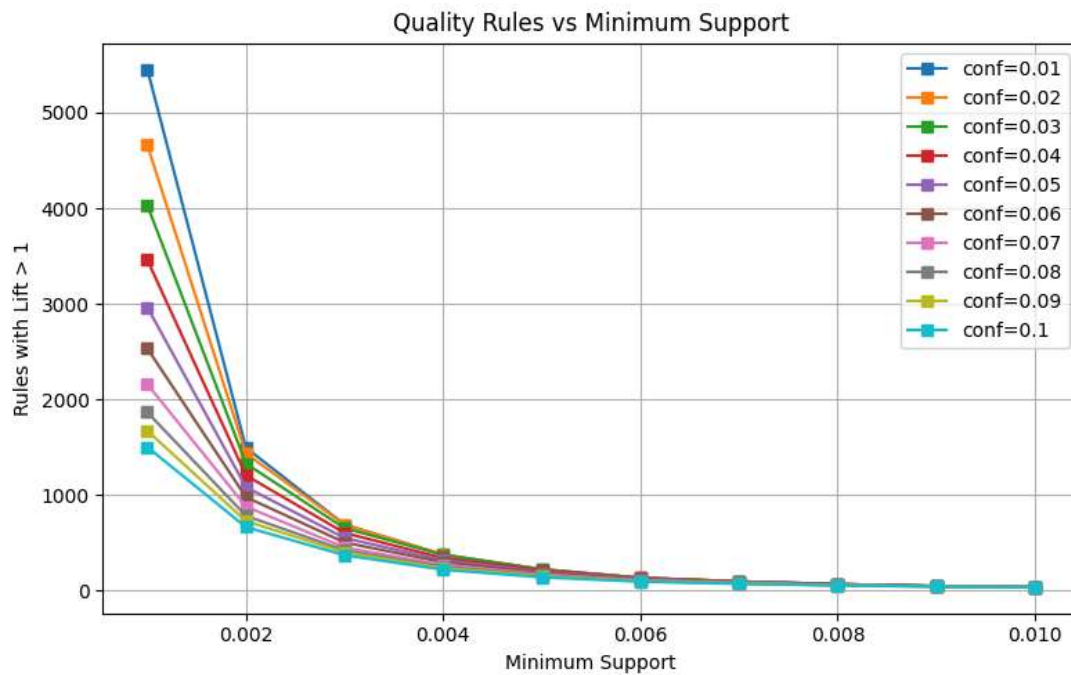
Validation:

- Small test transactions are used to verify correctness, ensuring:
 - All expected itemsets are found.
 - No unexpected itemsets are generated.

Summary

1. Input: Transactions and thresholds (min_support , min_confidence).
2. Count frequent items \rightarrow Filter infrequent items.
3. Build FP-tree: Insert transactions in frequency order, linking nodes with `node_link`.
4. Mine tree recursively:
 - Create conditional pattern bases.
 - Build conditional FP-trees.
 - Extract all frequent itemsets.
5. Generate rules:
 - For each frequent itemset, compute confidence and lift.
 - Keep rules above min_confidence .

- Single-item antecedents for clarity.
- Test on known datasets for correctness.

[illegible]

To determine suitable thresholds for association rule mining, we experimented with multiple combinations of minimum support (minsup) and minimum confidence (minconf) values in the ranges **[0.001, 0.01]** and **[0.01, 0.10]**, respectively. As expected, lowering these thresholds resulted in a very large number of rules, many of which were weak or redundant, while higher thresholds significantly reduced the number of meaningful associations.

From the experimental results, we observed that:

- At **minsup = 0.001**, more than **5000 rules** were generated, making interpretation and practical usage difficult.
- As **minsup** increased, the total number of rules decreased sharply, leading to a more concise and interpretable rule set while still capturing significant patterns.
- At **minsup = 0.003**, the number of rules stabilized around **500–600**, with the majority having a **lift greater than 1**, indicating strong associations between products.
- When comparing confidence thresholds, **minconf = 0.05** and **minconf = 0.06** both produced a manageable number of rules, but the **quality rules ratio** (percentage of rules with lift > 1) was slightly higher for **minconf = 0.06 (99.6%)** compared to **minconf = 0.05 (99.4%)**, suggesting marginally stronger and more reliable rules at this confidence level.

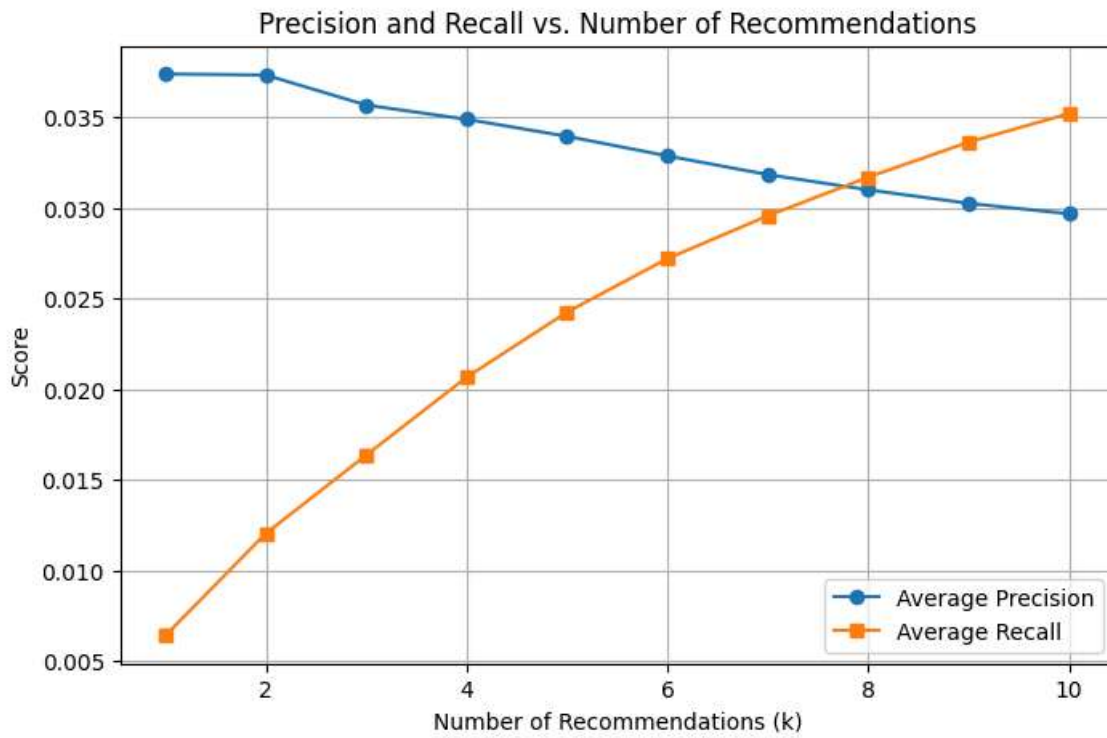
Hence, after balancing rule quantity, strength (lift > 1), and quality ratio, the final chosen thresholds are:

Minimum Support (minsup) = 0.003

Minimum Confidence (minconf) = 0.06

This combination provides an optimal balance between rule interpretability and association strength, producing a compact set of high-quality, reliable rules suitable for downstream tasks such as product recommendation and evaluation in the market basket analysis.

Plot of Precision and Recall vs Number of Recommendations:



Key Observations:

1. Precision Trend:

- Precision generally **decreases** as the number of recommendations (k) increases.
- This is expected because as more recommendations are made, the likelihood of including irrelevant items increases, reducing the proportion of relevant recommendations.

2. Recall Trend:

- Recall generally **increases** as the number of recommendations (k) increases.
- This is because recommending more items increases the chances of covering more relevant items from the ground truth.

3. Trade-off Between Precision and Recall: There is a clear trade-off between precision and recall:

- At lower values of k, precision is higher but recall is lower.
- At higher values of k, recall improves, but precision drops.

4. Convergence:

- Both precision and recall tend to stabilize as k approaches 10, indicating that increasing k beyond this range may not significantly impact the metrics.

Interpretation of Results

- **Precision:**

- Precision measures the proportion of recommended items that are relevant.
- The decreasing trend suggests that the recommendation system is more accurate when fewer items are recommended. However, as k increases, the system starts recommending less relevant items, diluting precision.

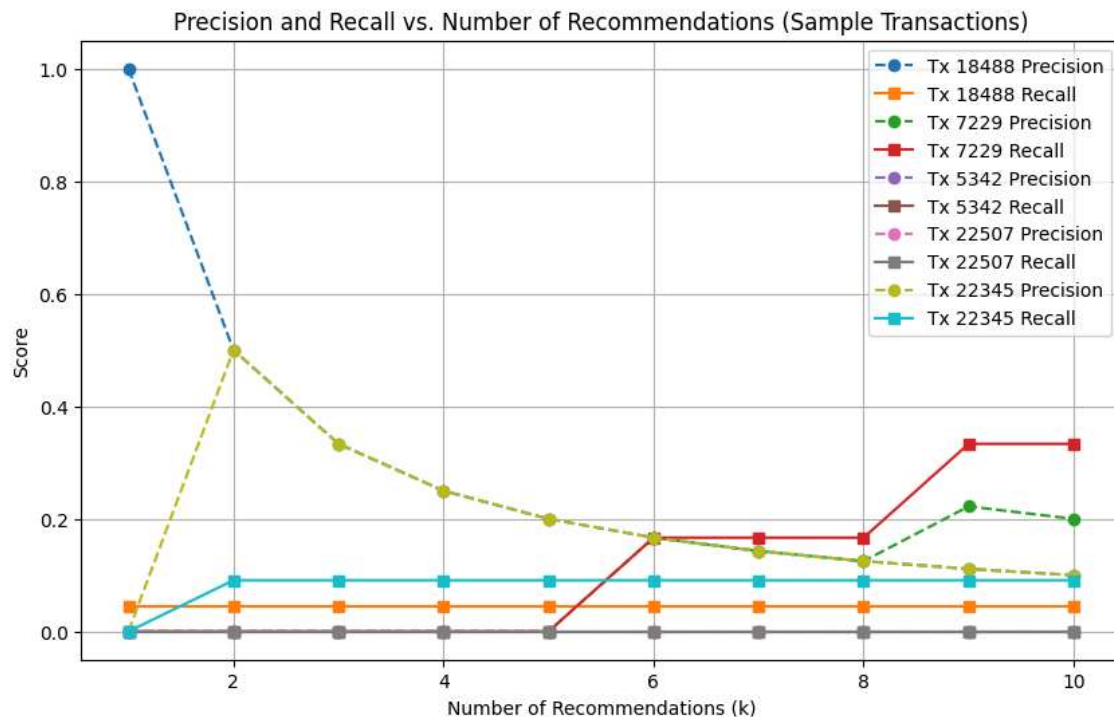
- **Recall:**

- Recall measures the proportion of relevant items that are successfully recommended.
- The increasing trend indicates that the system is able to capture more relevant items as the number of recommendations increases. However, this comes at the cost of including irrelevant items, as seen in the drop in precision.

- **Optimal k :** The choice of k depends on the application:

- If **precision** is more critical (e.g., recommending highly relevant items in a limited space), a smaller k (e.g., 3–5) may be ideal.
- If **recall** is more critical (e.g., ensuring all relevant items are recommended), a larger k (e.g., 8–10) may be preferred.

Plot of Precision and Recall vs. Number of Recommendations (Sample Transactions)



The plot shows the Precision and Recall scores for five randomly selected test transactions as the number of recommendations (k) increases from 1 to 10. Each transaction is represented by two curves: one for Precision and one for Recall.

Key Observations

1. Precision:

- Precision starts high for smaller values of k (e.g., k=1) and decreases as k increases.
- This is because, with fewer recommendations, the system is more likely to recommend highly relevant items. As k increases, the likelihood of recommending irrelevant items grows, reducing precision.

2. Recall:

- Recall generally increases as k increases.
- This is expected because recommending more items increases the chances of covering more relevant items from the ground

truth.

3. Transaction-Specific Trends:

- Some transactions (e.g., **Tx 18488**) show a sharp drop in precision and recall after $k=1$, indicating that the system struggles to recommend relevant items beyond the top recommendation.
- Other transactions (e.g., **Tx 7229**) show a steady increase in recall as k increases, suggesting that the system captures more relevant items with larger recommendation lists.

4. Variability Across Transactions:

- Precision and recall scores vary significantly across transactions, highlighting the dependency of recommendation performance on the specific input basket and ground truth.

Interpretation of Results

1. Precision vs. Recall Trade-off:

- The plot demonstrates the inherent trade-off between precision and recall:
 - **Precision** is higher for smaller k values, as the system focuses on recommending the most relevant items.
 - **Recall** improves with larger k values, as the system captures more relevant items from the ground truth.

2. Optimal Number of Recommendations (k):

- The choice of k depends on the application:
 - For scenarios where users prioritize highly relevant recommendations (e.g., e-commerce), smaller k values are ideal.
 - For applications where missing relevant items is critical (e.g., medical diagnosis), larger k values are preferred.

3. Performance Variability:

- The system performs inconsistently across transactions, as seen in the varying precision and recall curves. This suggests that the recommendation rules may not generalize well across all test cases.

Comment on Transactions with Zero Scores

Transactions **Tx 7229**, **Tx 5342**, and **Tx 22507** show zero scores for **Precision** and **Recall** for certain or all values of k .

Reasons for Zero Scores

- Input items may not appear as antecedents in learned rules
- Transaction Data is huge
- Sparse shopping patterns not well-represented in training data

Real-world shopping behavior:

- Grocery shopping patterns are highly individual and sparse
- Association rules capture common patterns but miss unique combinations

Insights from Specific Transactions

1. Tx 7229:

- Precision and Recall are zero for smaller values of k but improve slightly for larger k values. This suggests that the system starts recommending relevant items only when the recommendation list is expanded, indicating weak rule coverage for this transaction.

2. Tx 5342 and Tx 22507:

- Both transactions have zero scores across all k values, indicating that the system completely failed to recommend relevant items. This could be due to:
 - Ground truth items being rare or absent in the frequent itemsets.
 - Input basket items not triggering any rules.

3. Tx 22345:

- Precision and Recall are non-zero for larger k values, but the scores are relatively low. This suggests that the system is able to recommend some relevant items, but the overlap with the ground truth is limited.

Suggestions for Improvement

1. Expand Rule Coverage:

- Increase the diversity of association rules by lowering the minimum support threshold during rule mining. This can help

1. Banana: 18726 purchases
2. Bag of Organic Bananas: 15480 purchases
3. Organic Strawberries: 10894 purchases
4. Organic Baby Spinach: 9784 purchases
5. Large Lemon: 8135 purchases