

NAME: Chaitanya Kumar Arra
NJIT UCID: ca446
Email Address: ca446@njit.edu
14-oct-24
Professor:
Yasser
Abduallah CS
634 101
Data Mining

MidTerm Project Report

Implementation and Code Usage

Frequent Itemset Discovery and Association Rule Generation Using Brute Force, Apriori, and FP-Growth Algorithms

2) Abstract:

In this project, frequent itemsets and association rules were generated using three different algorithms: Brute Force, Apriori, and FP-Growth. We created five transactional databases containing at least 20 transactions with various supermarket items. The project involved implementing the brute force method, verifying results with the Apriori algorithm from the Python library, and using the FP-Growth algorithm. We compared the output and timing performance of all three algorithms, highlighting their accuracy and efficiency in discovering frequent itemsets and generating association rules.

3) Introduction:

Frequent itemset mining and association rule generation are essential tasks in market basket analysis, helping businesses understand which products are frequently purchased together. In this project, we focus on three algorithms: Brute Force, Apriori, and FP-Growth, to discover frequent itemsets from transactional data and generate association rules based on user-specified support and confidence thresholds. The study also evaluates the performance and efficiency of these algorithms on different datasets.

4) Core Concepts and Principles:

Frequent itemset discovery and association rule mining rely on two fundamental metrics: support and confidence. These metrics help identify frequent patterns and relationships between items within a dataset.

- **Support:** Measures the frequency of an itemset in the dataset.

- **Confidence:** Reflects the likelihood of item B being purchased when item A is purchased.
-

5) Frequent Itemset Discovery:

The process of identifying itemsets that frequently occur together in transactions is called frequent itemset discovery. These itemsets are the building blocks for generating association rules.

6) Support and Confidence:

Support and confidence are key thresholds that help determine which itemsets are frequent and which association rules are strong. In this project, we allowed the user to specify the minimum support and confidence values to experiment with different settings.

7) Association Rules:

Association rules provide insight into the relationships between items. For example, an association rule might indicate that customers who buy bread are also likely to buy butter. These rules are derived from frequent itemsets using the following formula:

$$\text{Confidence}(A \rightarrow B) = \frac{\text{Support}(A \cup B)}{\text{Support}(A)}$$
$$\text{Confidence}(A \rightarrow B) = \frac{\text{Support}(A \cup B)}{\text{Support}(A)}$$

8) Project Workflow:

1. **Data Generation:** Created 5 different transactional databases, each containing 20 transactions with various items typically found in supermarkets (e.g., diapers, milk, bread, etc.).
 2. **Algorithm Implementation:**
 - Brute Force method to find frequent itemsets.
 - Use of Apriori and FP-Growth libraries for performance comparison.
 3. **Result Comparison:** Analyzed the frequent itemsets and association rules generated by each algorithm and compared their performance.
-

9) Data Loading and Preprocessing:

We generated five separate CSV files containing 20 transactions each. Each transaction consists of 1 to 5 randomly selected items from a predefined list of common supermarket products. The CSV files were loaded into Python for analysis.

10) Determination of Minimum Support and Confidence:

The user was prompted to specify the minimum support and confidence thresholds. These values directly influenced which itemsets were considered frequent and which association rules were generated.

11) Iteration Through Candidate Itemsets:

For the brute force method, we iterated through all possible combinations of items to generate candidate itemsets. For example:

- 1-itemsets (individual items)
 - 2-itemsets (pairs of items)
 - 3-itemsets, and so on.
-

12) Support Count Calculation:

The frequency (support count) of each candidate itemset was calculated by iterating through the dataset and checking how many transactions contained that itemset. Itemsets that met the minimum support threshold were deemed frequent.

13) Confidence Calculation:

Once frequent itemsets were identified, we generated association rules. For each rule, the confidence value was calculated to determine how often the rule held true in the dataset.

14) Association Rule Generation:

Using the identified frequent itemsets, association rules were generated based on user-specified confidence values. For example, the rule "Milk -> Bread" indicates that when milk is purchased, bread is also frequently purchased.

15) Results and Evaluation:

- **Brute Force Algorithm:** Although computationally expensive, the brute force method accurately discovered frequent itemsets and association rules. However, as the dataset size increased, the execution time increased significantly.
- **Apriori Algorithm:** The Apriori algorithm was faster than the brute force method due to its use of pruning techniques that eliminated infrequent itemsets early in the process. It generated the same frequent itemsets and association rules but with much less computation.

- **FP-Growth Algorithm:** The FP-Growth algorithm was the fastest of the three, as it compressed the data into a tree structure (FP-Tree) and mined frequent itemsets more efficiently than both the brute force and Apriori algorithms.

16) Conclusion:

In this project, we compared three different algorithms for frequent itemset mining and association rule generation: Brute Force, Apriori, and FP-Growth. The Brute Force method was the most computationally expensive, while Apriori improved performance through pruning, and FP-Growth was the fastest due to its tree-based approach. All three methods produced the same results, but the FP-Growth algorithm emerged as the most efficient in terms of time complexity. Based on our results, FP-Growth is recommended for larger datasets where computational efficiency is critical.

Screenshots

Step 1: Create a list of items usually seen in supermarkets.

Here are 10 example items:

1. Diapers
2. T-shirts
3. Milk (1L)
4. Bread (Whole Wheat)
5. Shampoo
6. Toothpaste
7. Eggs (Dozen)
8. Bananas (1kg)
9. Rice (5kg)
10. Canned Beans (400g)

```
[ ] import random
import csv

[ ] # List of supermarket items with their prices
items = [
    ("Diapers", 10.50),
    ("T-shirts", 7.00),
    ("Milk (1L)", 1.25),
    ("Bread (Whole Wheat)", 2.50),
    ("Shampoo", 4.75),
    ("Toothpaste", 3.00),
    ("Eggs (Dozen)", 2.00),
    ("Bananas (1kg)", 1.10),
    ("Rice (5kg)", 15.00),
    ("Canned Beans (400g)", 1.20)
]
```

Step 2: Create a database of at least 20 transactions containing some of these items.

We'll generate a dataset with 20 transactions, where each transaction contains a random selection of items and save it as a CSV file.

Example Data Structure:

The CSV file will have the following structure:

- Transaction ID: Unique identifier for each transaction.
- Items: List of items in the transaction.
- Total Amount: Sum of item prices.

Python Code to Create a Database and Save to CSV

```
[ ] # Function to generate a random transaction
def generate_transaction(transaction_id):
    num_items = random.randint(1, 5) # Each transaction will have 1 to 5 items
    selected_items = random.sample(items, num_items)
    item_names = [item[0] for item in selected_items]
    total_amount = sum(item[1] for item in selected_items)

    return {
        "Transaction ID": transaction_id,
        "Items": ", ".join(item_names),
        "Total Amount": round(total_amount, 2)
    }

[ ] # Generate 20 transactions
transactions = [generate_transaction(i + 1) for i in range(20)]

[ ] # Save transactions to CSV
csv_file = "supermarket_transactions.csv"
with open(csv_file, mode='w', newline='') as file:
    writer = csv.DictWriter(file, fieldnames=["Transaction ID", "Items", "Total Amount"])
    writer.writeheader()
    writer.writerows(transactions)

print(f"Transactions saved to {csv_file}")
```

Transactions saved to supermarket_transactions.csv

Each of these files contains unique sets of transactions with random items selected from the list, and the total amounts are calculated based on the prices of those items.

Repeat step 2 to create 4 additional databases.

```
# Function to create a database and save to CSV
def create_database(file_name, transaction_count=20):
    transactions = [generate_transaction(i + 1) for i in range(transaction_count)]

    # Save to CSV
    with open(file_name, mode='w', newline='') as file:
        writer = csv.DictWriter(file, fieldnames=["Transaction ID", "Items", "Total Amount"])
        writer.writeheader()
        writer.writerows(transactions)
        print(f"Database saved to {file_name}")

# Create 5 different databases
for i in range(5):
    file_name = f"supermarket_transactions_{i+1}.csv"
    create_database(file_name)
```

Database saved to supermarket_transactions_1.csv
Database saved to supermarket_transactions_2.csv
Database saved to supermarket_transactions_3.csv
Database saved to supermarket_transactions_4.csv
Database saved to supermarket_transactions_5.csv

Here is the extended version of the code that repeats the process to generate 4 additional databases with 20 transactions each, resulting in 5 CSV files in total.

To implement the brute force method for generating frequent itemsets and association rules, we follow these steps:

- 1) Input Transaction Data: Collect or simulate transaction data where each transaction is a list of items purchased.
- 2) Generate All Possible Itemsets: For each transaction, generate all possible 1-itemsets, 2-itemsets, and so on, using combinations.
- 3) Check Frequency of Each Itemset: For each generated itemset, check how often it appears in the transaction database. Itemsets that meet or exceed a minimum frequency (support) are considered frequent.
- 4) Stop When No Frequent k-itemsets are Found: Continue until no frequent k-itemsets can be found, at which point the process stops.
- 5) Generate Association Rules: For each frequent itemset, generate association rules, which help identify how items are related.

Steps in the Brute Force Method:

Simulate or Load Transaction Data Here, we simulate transaction data similar to what would be collected in a supermarket.

Generate Itemsets For n items in the supermarket, the number of combinations increases exponentially with the number of items (k) in the itemset.

Count Support (Frequency) For each candidate itemset, count the number of transactions in which it appears and compare it to a minimum support threshold.

Generate Association Rules Once the frequent itemsets are identified, generate rules like "if A, then B" and evaluate them based on metrics like confidence.

```
[ ] import itertools
    from collections import defaultdict

[ ] # Sample transactions (20 transactions as an example)
    transactions = [
        ['Milk', 'Diapers', 'Bread'],
        ['Milk', 'Shampoo', 'Eggs'],
        ['Bread', 'Rice', 'Bananas'],
        ['Milk', 'Bananas', 'Eggs'],
        ['Toothpaste', 'Canned Beans', 'Diapers'],
        ['Milk', 'Bananas', 'Bread'],
        ['Diapers', 'T-shirts'],
        ['Rice', 'Bananas', 'Milk'],
        ['Eggs', 'Bread', 'Shampoo'],
        ['Rice', 'Bananas', 'Canned Beans'],
        ['Toothpaste', 'Milk', 'Eggs'],
        ['Bread', 'Canned Beans', 'Shampoo'],
        ['Diapers', 'Bananas', 'Milk'],
        ['Eggs', 'Milk', 'Toothpaste'],
        ['Rice', 'T-shirts', 'Bread'],
        ['Shampoo', 'Canned Beans'],
        ['Rice', 'Eggs', 'Bananas'],
        ['Milk', 'Toothpaste'],
        ['Shampoo', 'Rice', 'Diapers'],
        ['Canned Beans', 'Eggs', 'Bananas']
    ]
```

```
[ ] # Minimum support threshold
    min_support = 3

    # Function to generate itemsets and calculate frequency
    def brute_force_itemsets(transactions, min_support):
        # Get unique items
        all_items = sorted(set(item for transaction in transactions for item in transaction))

        # Function to count the frequency of a given itemset
        def count_frequency(itemset, transactions):
            count = sum(1 for transaction in transactions if set(itemset).issubset(set(transaction)))
            return count

        # Initialize variables
        frequent_itemsets = []
        k = 1

        while True:
            # Generate k-itemsets
            itemsets = list(itertools.combinations(all_items, k))
            print(f"\nGenerating {k}-itemsets: {len(itemsets)} possible combinations")

            # Calculate support for each itemset
            current_frequent_itemsets = []
            for itemset in itemsets:
                frequency = count_frequency(itemset, transactions)
                if frequency >= min_support:
                    current_frequent_itemsets.append((itemset, frequency))
```

```

while True:
    # Generate k-itemsets
    itemsets = list(itertools.combinations(all_items, k))
    print(f"\nGenerating {k}-itemsets: {len(itemsets)} possible combinations")

    # Calculate support for each itemset
    current_frequent_itemsets = []
    for itemset in itemsets:
        frequency = count_frequency(itemset, transactions)
        if frequency >= min_support:
            current_frequent_itemsets.append((itemset, frequency))

    # If no frequent itemsets are found, terminate
    if not current_frequent_itemsets:
        break

    # Add frequent itemsets to the final list
    frequent_itemsets.extend(current_frequent_itemsets)
    print(f"Frequent {k}-itemsets: {current_frequent_itemsets}")

    # Increment k
    k += 1

return frequent_itemsets

```

```

[ ] # Generate frequent itemsets using brute force
frequent_itemsets = brute_force_itemsets(transactions, min_support)

# Function to generate association rules
def generate_association_rules(frequent_itemsets, transactions, min_confidence=0.7):
    rules = []
    transaction_count = len(transactions)

    for itemset, support in frequent_itemsets:
        if len(itemset) < 2:
            continue

        # Generate rules of form A -> B
        for i in range(1, len(itemset)):
            for lhs in itertools.combinations(itemset, i):
                rhs = set(itemset) - set(lhs)
                lhs_count = sum(1 for transaction in transactions if set(lhs).issubset(set(transaction)))
                confidence = support / lhs_count

                if confidence >= min_confidence:
                    rules.append({
                        "rule": f"{lhs} -> {rhs}",
                        "support": support / transaction_count,
                        "confidence": confidence
                    })

    return rules

```

```

return rules

# Generate association rules
association_rules = generate_association_rules(frequent_itemsets, transactions)

# Display results
print("\nFrequent Itemsets:")
for itemset, support in frequent_itemsets:
    print(f"Itemset: {itemset}, Support: {support}")

print("\nAssociation Rules:")
for rule in association_rules:
    print(f"Rule: {rule['rule']}, Support: {rule['support']:.2f}, Confidence: {rule['confidence']:.2f}")

```



```

Generating 1-itemsets: 10 possible combinations
Frequent 1-itemsets: (((('Bananas',), 8), (('Bread',), 6), (('Canned Beans',), 5), (('Diapers',), 5), (('Eggs',), 7), (('Milk',), 9), (('Rice',), 6), (('Shampoo',), 3)))

Generating 2-itemsets: 45 possible combinations
Frequent 2-itemsets: (((('Bananas', 'Eggs'), 3), (('Bananas', 'Milk'), 4), (('Bananas', 'Rice'), 4), (('Eggs', 'Milk'), 4), (('Milk', 'Toothpaste'), 3)))

Generating 3-itemsets: 120 possible combinations

Frequent Itemsets:
Itemset: ('Bananas',), Support: 8
Itemset: ('Bread',), Support: 6
Itemset: ('Canned Beans',), Support: 5
Itemset: ('Diapers',), Support: 5
Itemset: ('Eggs',), Support: 7

```


output



```
Generating 1-itemsets: 10 possible combinations
Frequent 1-itemsets: [(('Bananas',), 8), (('Bread',), 6), (('Canned Beans',), 5), (('Diapers',), 5), (('Eggs',), 7), (('Milk',), 9), (('Rice',), 6), (('Shampoo',), 5)]

Generating 2-itemsets: 45 possible combinations
Frequent 2-itemsets: [(('Bananas', 'Eggs'), 3), (('Bananas', 'Milk'), 4), (('Bananas', 'Rice'), 4), (('Eggs', 'Milk'), 4), (('Milk', 'Toothpaste'), 3)]

Generating 3-itemsets: 120 possible combinations

Frequent Itemsets:
Itemset: ('Bananas',), Support: 8
Itemset: ('Bread',), Support: 6
Itemset: ('Canned Beans',), Support: 5
Itemset: ('Diapers',), Support: 5
Itemset: ('Eggs',), Support: 7
Itemset: ('Milk',), Support: 9
Itemset: ('Rice',), Support: 6
Itemset: ('Shampoo',), Support: 5
Itemset: ('Toothpaste',), Support: 4
Itemset: ('Bananas', 'Eggs'), Support: 3
Itemset: ('Bananas', 'Milk'), Support: 4
Itemset: ('Bananas', 'Rice'), Support: 4
Itemset: ('Eggs', 'Milk'), Support: 4
Itemset: ('Milk', 'Toothpaste'), Support: 3

Association Rules:
Rule: ('Toothpaste',) -> {'Milk'}, Support: 0.15, Confidence: 0.75
```

Explanation of the Code:

- 1) Generating Itemsets: We use the `itertools.combinations` function to generate all possible 1-itemsets, 2-itemsets, and so on. The loop runs until no frequent itemsets are found at a particular level.
- 2) Counting Frequency: The function `count_frequency` checks how many transactions contain a given itemset. Itemsets that meet the minimum support threshold are considered frequent.
- 3) Generating Association Rules: We then generate association rules for the frequent itemsets using the basic "if A, then B" format. Confidence is calculated based on the proportion of transactions containing A that also contain B.
- 4) Stopping Criteria: The algorithm terminates when no frequent k-itemsets are found.

Output Example:

- 1)Frequent Itemsets: Itemsets with their support count.
- 2)Association Rules: Rules of the form "A -> B" along with their support and confidence values.

Brute Force Complexity:

This method is computationally expensive because it generates all possible itemsets and checks each one. However, it ensures that no frequent itemsets are missed. Optimizations like pruning can help reduce the search space in other algorithms like Apriori, but brute force ensures completeness.

- 1) Load the Transactional Databases: We will use the previously generated 5 transactional databases.

```
[ ] pip install mlxtend fpgrowth_py
```

```
Requirement already satisfied: mlxtend in /usr/local/lib/python3.10/dist-packages (0.23.1)
Collecting fpgrowth_py
  Downloading fpgrowth_py-1.0.0-py3-none-any.whl.metadata (2.3 kB)
Requirement already satisfied: scipy>=1.2.1 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (1.13.1)
Requirement already satisfied: numpy>=1.16.2 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (1.26.4)
Requirement already satisfied: pandas>=0.24.2 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (2.2.2)
Requirement already satisfied: scikit-learn>=1.0.2 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (1.5.2)
Requirement already satisfied: matplotlib>=3.0.0 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (3.7.1)
Requirement already satisfied: joblib>=0.13.2 in /usr/local/lib/python3.10/dist-packages (from mlxtend) (1.4.2)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (1.3.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (4.54.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (1.4.7)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (24.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (3.1.4)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.0.0->mlxtend) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.24.2->mlxtend) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.24.2->mlxtend) (2024.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0.2->mlxtend) (3.5.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib>=3.0.0->mlxtend) (1.16.0)
Downloading fpgrowth_py-1.0.0-py3-none-any.whl (5.6 kB)
Installing collected packages: fpgrowth_py
Successfully installed fpgrowth_py-1.0.0
```

```
[ ] import time
import itertools
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules
from fpgrowth_py import fpgrowth
```

2) Set User-Specified Parameters: Accept user inputs for minimum support and confidence levels.

```
[ ] import time
import itertools
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules
from fpgrowth_py import fpgrowth
from collections import defaultdict
```

```
[ ] # Load transactions from CSV (modify as needed to load your generated datasets)
def load_transactions(filename):
    df = pd.read_csv(filename)
    transactions = df['Items'].apply(lambda x: x.split(', '))
    return transactions.tolist()

# Get user input for support and confidence
def get_user_input():
    min_support = float(input("Enter minimum support (e.g., 0.1): "))
    min_confidence = float(input("Enter minimum confidence (e.g., 0.5): "))
    return min_support, min_confidence
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future and should_run_async(code)
```

3) Brute Force Algorithm: Implement the brute-force frequent itemset generation (from the previous code) with timing measurement.

```
[ ] # Brute-force algorithm
def brute_force(transactions, min_support):
    all_items = sorted(set(item for transaction in transactions for item in transaction))
    transaction_count = len(transactions)
    min_support_count = int(min_support * transaction_count)


    # Generate itemsets and count frequencies
    def count_frequency(itemset, transactions):
        return sum(1 for transaction in transactions if set(itemset).issubset(set(transaction)))

    frequent_itemsets = []
    k = 1
    while True:
        itemsets = list(itertools.combinations(all_items, k))
        current_frequent_itemsets = []
        for itemset in itemsets:
            frequency = count_frequency(itemset, transactions)
            if frequency >= min_support_count:
                current_frequent_itemsets.append((itemset, frequency))

        if not current_frequent_itemsets:
            break


        frequent_itemsets.extend(current_frequent_itemsets)
        k += 1

    return frequent_itemsets
```

 /usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future and should_run_async(code)

```
[ ] # Generate association rules for brute-force
def generate_association_rules_bruteforce(frequent_itemsets, transactions, min_confidence):
    transaction_count = len(transactions)
    rules = []
    for itemset, support in frequent_itemsets:
        if len(itemset) < 2:
            continue
        for i in range(1, len(itemset)):
            for lhs in itertools.combinations(itemset, i):
                rhs = set(itemset) - set(lhs)
                lhs_count = sum(1 for transaction in transactions if set(lhs).issubset(set(transaction)))
                confidence = support / lhs_count
                if confidence >= min_confidence:
                    rules.append((lhs, rhs, support / transaction_count, confidence))


    return rules
```

 /usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future and should_run_async(code)

4) Apriori Algorithm: Use a pre-existing implementation from a Python library like mlxtend.

```
[ ] # Apriori algorithm
def run_apriori(transactions, min_support, min_confidence):
    # Convert transactions into a one-hot encoded DataFrame
    df = pd.DataFrame(transactions)
    one_hot = pd.get_dummies(df.stack()).groupby(level=0).sum()

    # Use Apriori from mlxtend
    frequent_itemsets = apriori(one_hot, min_support=min_support, use_colnames=True)
    rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=min_confidence)
    return frequent_itemsets, rules
```

 /usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future and should_run_async(code)

5) FP-Growth Algorithm: Use an existing package like fp-growth to generate frequent itemsets and rules.

```
[ ] # FP-growth algorithm
def run_fp_growth(transactions, min_support, min_confidence):
    frequent_itemsets, rules = fpgrowth(transactions, minSupRatio=min_support, minConf=min_confidence)
    return frequent_itemsets, rules

# Time execution of a function
def time_function(func, *args):
    start_time = time.time()
    result = func(*args)
    end_time = time.time()
    return result, end_time - start_time
```

/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future and should_run_async(code)

6) Compare Results: Compare the itemsets and association rules generated by all three algorithms.

7) Performance Timing: Measure and compare the execution time of the three algorithms.

```
[ ] # Main function to run all algorithms and compare results
def main():
    # Load transactions from CSV (modify file names as necessary)
    transactions_list = [load_transactions(f"supermarket_transactions_{i}.csv") for i in range(5)]

    # Get user input for support and confidence
    min_support, min_confidence = get_user_input()

    for i, transactions in enumerate(transactions_list):
        print(f"\n--- Results for Database {i+1} ---")

        # Brute-force algorithm
        brute_force_result, brute_force_time = time_function(brute_force, transactions, min_support)
        brute_force_rules = generate_association_rules_bruteforce(brute_force_result, transactions, min_confidence)
        print(f"\nBrute-force frequent itemsets: {brute_force_result}")
        print(f"Brute-force association rules: {brute_force_rules}")
        print(f"Brute-force execution time: {brute_force_time:.4f} seconds")

        # Apriori algorithm
        apriori_result, apriori_time = time_function(run_apriori, transactions, min_support, min_confidence)
        print(f"\nApriori frequent itemsets: {apriori_result}")
        print(f"Apriori association rules: {apriori_result[1]}")
        print(f"Apriori execution time: {apriori_time:.4f} seconds")

        # FP-growth algorithm
        fp_growth_result, fp_growth_time = time_function(run_fp_growth, transactions, min_support, min_confidence)
        print(f"\nFP-Growth frequent itemsets: {fp_growth_result}")
        print(f"FP-Growth association rules: {fp_growth_result[1]}")
        print(f"FP-Growth execution time: {fp_growth_time:.4f} seconds")

    print(f"\nFP-Growth frequent itemsets: {fp_growth_result}")
    print(f"FP-Growth association rules: {fp_growth_result[1]}")
    print(f"FP-Growth execution time: {fp_growth_time:.4f} seconds")

    # Timing comparison
    print(f"\nTiming comparison for Database {i+1}:")
    print(f"Brute-force time: {brute_force_time:.4f} seconds")
    print(f"Apriori time: {apriori_time:.4f} seconds")
    print(f"FP-Growth time: {fp_growth_time:.4f} seconds")

if __name__ == "__main__":
    main()
```

/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future and should_run_async(code)

Enter minimum support (e.g., 0.1): 0.1
Enter minimum confidence (e.g., 0.5): 0.5

8) Display Results: Output the frequent itemsets and association rules for each algorithm, showing the differences and similarities.

```
-- Results for Database 1 ---
```

```
Brute-force frequent itemsets: [(('Bananas (1kg)',), 9), (('Bread (Whole Wheat)',), 2), (('Canned Beans (400g)',), 6), (('Diapers',), 6), (('Eggs (Dozen)',), 6)]  
Brute-force association rules: [(('Diapers',), {'Bananas (1kg)'}, 0.2, 0.6666666666666666), (('Bananas (1kg)',), {'Eggs (Dozen)'}, 0.25, 0.5555555555555556)]  
Brute-force execution time: 0.0109 seconds
```

```
Apriori frequent itemsets: (      support                                itemsets  
0      0.45                                (Bananas (1kg))  
1      0.10                                (Bread (Whole Wheat))  
2      0.30                                (Canned Beans (400g))  
3      0.30                                (Diapers)  
4      0.40                                (Eggs (Dozen))  
5      0.35                                (Milk (1L))  
6      0.40                                (Rice (5kg))  
7      0.30                                (Shampoo)  
8      0.35                                (T-shirts)  
9      0.20                                (Toothpaste)  
10     0.20                                (Bananas (1kg), Diapers)  
11     0.25                                (Bananas (1kg), Eggs (Dozen))  
12     0.15                                (Bananas (1kg), Milk (1L))  
13     0.20                                (Bananas (1kg), Rice (5kg))  
14     0.20                                (Bananas (1kg), Shampoo)  
15     0.15                                (Bananas (1kg), T-shirts)  
16     0.10                                (Canned Beans (400g), Diapers)  
17     0.20                                (Canned Beans (400g), Eggs (Dozen))  
18     0.15                                (Canned Beans (400g), Milk (1L))  
19     0.10                                (Canned Beans (400g), Rice (5kg))  
20     0.10                                (Canned Beans (400g), Shampoo)  
21     0.20                                (Eggs (Dozen), Diapers)  
22     0.15                                (Diapers, Rice (5kg))  
23     0.10                                (Diapers, Shampoo)
```

Explanation:

Load Transactions: Transactions are loaded from CSV files and passed to the three algorithms.

Brute-Force Algorithm: This algorithm generates itemsets by iterating over all combinations and counts their frequency. Association rules are generated based on user-specified support and confidence.

Apriori Algorithm: Uses the mlxtend library's implementation. It converts transactions into a one-hot encoded DataFrame and generates itemsets and rules using Apriori.

FP-Growth Algorithm: Uses the `fpgrowth_py` library to generate frequent itemsets and association rules.

Timing Performance: The `time_function` wrapper is used to measure the execution time for each algorithm.

User Input: The user provides support and confidence thresholds, which are applied to all three algorithms.

Output:

The program will display:

Frequent Itemsets and Association Rules for each algorithm (Brute-force, Apriori, FP-Growth).

Execution Time for each algorithm.

Comparison of Results from the three algorithms.

Performance Comparison:

By running the program, you can compare:

The frequent itemsets and association rules generated by each algorithm.

The execution time of each algorithm, allowing you to determine which is faster.

Other

The source code (.ipynb file) and data sets (.csv files) are attached to the zip file.

Link to Git Repository

https://github.com/ack446/ChaitanyaKumar_Arra_MidTerm.git