Comparison of Priori and Brute Force Algorithms for Frequent Items Mining

Midterm Project

Aurel Sahiti

UCID: as4579

as4579@njit.edu

In [2]:

```
import random
import pandas as pd
from itertools import combinations
import time
```

Introduction

This document details the implementation and comparison of two algorithms: the Priori algorithm and the Brute Force method for mining frequent item sets and generating association rules from transaction data. We will explore how both methods work and compare their performance based on their execution times.

Creating Transaction Data

In [3]:

```
# Create 30 items from supermarkets
items = [
    'milk', 'bread', 'butter', 'eggs', 'cheese', 'chicken', 'beef', 'fish',
    'apples', 'bananas', 'oranges', 'grapes', 'cereal', 'pasta', 'rice',
    'tomatoes', 'onions', 'potatoes', 'broccoli', 'carrots', 'shampoo',
    'floss', 'toothpaste', 'detergent', 'cleaning_wipes', 'diapers',
    'napkins', 'paper_towels', 'dish_soap', 'toilet_paper'
]

# Create a function that will generate random transactions
def generate_transactions(num_transactions=20):
    transactions = []
    for _ in range(num_transactions):
        # Randomly choose a subset of items for each transaction (1-10 items)
        num_items = random.randint(1, 10)
        transaction = random.sample(items, num_items)
        transactions.append(transaction)
    return transactions

# Generate 5 databases with 20 transactions each
databases = [generate_transactions() for _ in range(5)]
```

- We started by defining 30 items commonly purchased in supermarkets.

- The “generate_transactions” function generates a list of transactions, where each transaction contains anywhere from 1 up to 10 items.

- In total, 5 databases were created with 20 transactions in each.

Creating Functions

In [4]:

```
# Calculate support for a given itemset
def calculate_support(itemset, transactions):
    count = sum(1 for transaction in transactions if itemset.issubset(transaction))
    return count / len(transactions)

# Filter itemsets by support
def filter_itemsets_by_support(itemsets, transactions, min_support):
    filtered_itemsets = {}
    for itemset in itemsets:
        support = calculate_support(itemset, transactions)
        if support >= min_support:
            filtered_itemsets[itemset] = support
    return filtered_itemsets

# Generate candidate itemsets of size 1
def get_itemsets_size_1(transactions):
    itemset = set()
    for transaction in transactions:
        for item in transaction:
            itemset.add(frozenset([item]))
    return itemset

# Generate candidate itemsets of larger size by combining frequent itemsets
def generate_candidate_itemsets(frequent_itemsets, k):
    candidates = set()
    frequent_itemsets_list = list(frequent_itemsets)

    for i in range(len(frequent_itemsets_list)):
        for j in range(i + 1, len(frequent_itemsets_list)):
            l1 = list(frequent_itemsets_list[i])
            l2 = list(frequent_itemsets_list[j])
            l1.sort()
            l2.sort()
            if l1[:k - 2] == l2[:k - 2]:  # Only combine if first k-2 items are the same
                candidates.add(frozenset(frequent_itemsets_list[i] | frequent_itemsets_list[j]))

    return candidates

# Generate association rules
def generate_association_rules(frequent_itemsets, transactions, min_confidence):
    rules = []

    for itemset in frequent_itemsets:
        if len(itemset) < 2:
            continue
        for consequence in itemset:
            antecedent = itemset - frozenset([consequence])
            if len(antecedent) == 0:
                continue
            # Calculate confidence
            support_itemset = calculate_support(itemset, transactions)
            support_antecedent = calculate_support(antecedent, transactions)
            confidence = support_itemset / support_antecedent if support_antecedent > 0 else 0

            if confidence >= min_confidence:
                rules.append((antecedent, frozenset([consequence]), confidence))

    return rules
```

- Support is calculated as the fraction of transactions containing the given item-set.

- The “calculate_support” function is used in both algorithms to check if an item-set is frequent (if its support is above the specified minimum threshold).

- The “filter_itemsets_by_support” function loops through each item-set in the input “itemsets”. For each item-set, it calculates its support using the “calculate_support” function. If the support is greater than or equal to the specified “min_support”, the item-set is added to the “filtered_itemsets” dictionary along with its support value. Finally, the function returns the filtered_itemsets dictionary, which contains only itemsets that meet the minimum support criteria.

- The “get_itemsets_size_1” function is created to generate the initial set of 1-itemsets (combinations of single items) from a list of transactions.

- The use of the “frozenset” ensures that the itemsets are immutable, which is useful when storing these itemsets as keys in a dictionary for support calculations or when further processing them to generate larger itemsets.

- The “generate_candidate_itemsets” function generates candidate itemsets of size “k” by combining frequent itemsets of size k-1.

- The “generate_association_rules” function generates association rules from frequent item-sets by calculating the confidence of each rule (based on the support of the item-set and its antecedent).

- It also checks if the confidence meets the specified minimum threshold and stores the rules that satisfy that condition.

Apriori Algorithm Implementation

In [5]:

```
# Main Apriori algorithm
def apriori(transactions, items, min_support=0.2, min_confidence=0.6):
    # Step 1: Generate itemsets of size 1
    candidate_itemsets = get_itemsets_size_1(transactions)

    # Step 2: Filter itemsets by support
    frequent_itemsets = filter_itemsets_by_support(candidate_itemsets, transactions, min_support)

    all_frequent_itemsets = dict(frequent_itemsets)

    k = 2
    while frequent_itemsets:
        # Step 3: Generate candidate itemsets of size k
        candidate_itemsets = generate_candidate_itemsets(frequent_itemsets, k)

        # Step 4: Filter itemsets by support
        frequent_itemsets = filter_itemsets_by_support(candidate_itemsets, transactions, min_support)

        all_frequent_itemsets.update(frequent_itemsets)

        k += 1

    # Step 5: Generate association rules
    rules = generate_association_rules(all_frequent_itemsets.keys(), transactions, min_confidence)

    return all_frequent_itemsets, rules
```

In [6]:

```
# Apriori results
def apriori_results(databases, min_support, min_confidence, items):
    for i, transactions in enumerate(databases):
        print(f"\nDatabase {i + 1}:")

        # Apriori
        start_time = time.time()
        frequent_itemsets_apriori, rules_apriori = apriori(transactions, items, min_support, min_confidence)
        apriori_time = time.time() - start_time

        # Output frequent itemsets from Apriori
        print("\nApriori Frequent Itemsets:")
        for itemset, support in frequent_itemsets_apriori.items():
            print(f"Itemset: {set(itemset)}, Support: {support:.4f}")

        # Output association rules from Apriori
        print("\nApriori Association Rules:")
        for antecedent, consequent, confidence in rules_apriori:
            print(f"Rule: {set(antecedent)} -> {set(consequent)}, Confidence: {confidence:.4f}")


# Set minimum support and confidence
min_support = 0.2
min_confidence = 0.6

# Compare the two algorithms on all 5 databases
apriori_results(databases, min_support, min_confidence, items)
```

```
Database 1:

Apriori Frequent Itemsets:
Itemset: {'rice'}, Support: 0.2000
Itemset: {'butter'}, Support: 0.2000
Itemset: {'chicken'}, Support: 0.2000
Itemset: {'cleaning_wipes'}, Support: 0.3000
Itemset: {'onions'}, Support: 0.2000
Itemset: {'eggs'}, Support: 0.2000
Itemset: {'carrots'}, Support: 0.2500
Itemset: {'detergent'}, Support: 0.3500
Itemset: {'grapes'}, Support: 0.3500
Itemset: {'milk'}, Support: 0.2000
Itemset: {'bananas'}, Support: 0.3000
Itemset: {'oranges'}, Support: 0.2000
Itemset: {'floss'}, Support: 0.2500
Itemset: {'broccoli'}, Support: 0.2500
Itemset: {'dish_soap'}, Support: 0.2000
Itemset: {'tomatoes'}, Support: 0.3000
Itemset: {'detergent', 'grapes'}, Support: 0.2500
Itemset: {'chicken', 'grapes'}, Support: 0.2000
Itemset: {'cleaning_wipes', 'milk'}, Support: 0.2000

Apriori Association Rules:
Rule: {'grapes'} -> {'detergent'}, Confidence: 0.7143
Rule: {'detergent'} -> {'grapes'}, Confidence: 0.7143
Rule: {'chicken'} -> {'grapes'}, Confidence: 1.0000
Rule: {'milk'} -> {'cleaning_wipes'}, Confidence: 1.0000
Rule: {'cleaning_wipes'} -> {'milk'}, Confidence: 0.6667

Database 2:

Apriori Frequent Itemsets:
Itemset: {'beef'}, Support: 0.2000
Itemset: {'cereal'}, Support: 0.2000
Itemset: {'apples'}, Support: 0.2500
Itemset: {'onions'}, Support: 0.2000
Itemset: {'eggs'}, Support: 0.4000
Itemset: {'bread'}, Support: 0.3000
Itemset: {'detergent'}, Support: 0.2000
Itemset: {'oranges'}, Support: 0.2500
Itemset: {'broccoli'}, Support: 0.3000
Itemset: {'napkins'}, Support: 0.2500
Itemset: {'tomatoes'}, Support: 0.2000
Itemset: {'shampoo'}, Support: 0.2500
Itemset: {'toilet_paper'}, Support: 0.2000

Apriori Association Rules:

Database 3:

Apriori Frequent Itemsets:
Itemset: {'rice'}, Support: 0.2000
Itemset: {'butter'}, Support: 0.2500
Itemset: {'cleaning_wipes'}, Support: 0.2500
Itemset: {'cereal'}, Support: 0.2000
Itemset: {'diapers'}, Support: 0.2000
Itemset: {'apples'}, Support: 0.3000
Itemset: {'onions'}, Support: 0.2500
Itemset: {'eggs'}, Support: 0.3500
Itemset: {'pasta'}, Support: 0.2500
Itemset: {'detergent'}, Support: 0.2000
Itemset: {'paper_towels'}, Support: 0.2000
Itemset: {'grapes'}, Support: 0.3500
Itemset: {'milk'}, Support: 0.2000
Itemset: {'bananas'}, Support: 0.3500
Itemset: {'toothpaste'}, Support: 0.2000
Itemset: {'oranges'}, Support: 0.2000
Itemset: {'broccoli'}, Support: 0.2000
Itemset: {'napkins'}, Support: 0.2500
Itemset: {'fish'}, Support: 0.2500
Itemset: {'potatoes'}, Support: 0.2500
Itemset: {'toilet_paper'}, Support: 0.2500
Itemset: {'diapers', 'bananas'}, Support: 0.2000
Itemset: {'grapes', 'bananas'}, Support: 0.2000

Apriori Association Rules:
Rule: {'diapers'} -> {'bananas'}, Confidence: 1.0000

Database 4:

Apriori Frequent Itemsets:
Itemset: {'beef'}, Support: 0.3000
Itemset: {'butter'}, Support: 0.2000
Itemset: {'cereal'}, Support: 0.2000
Itemset: {'diapers'}, Support: 0.2500
Itemset: {'apples'}, Support: 0.3000
Itemset: {'pasta'}, Support: 0.2000
Itemset: {'grapes'}, Support: 0.3000
Itemset: {'bananas'}, Support: 0.2000
Itemset: {'toothpaste'}, Support: 0.2500
Itemset: {'floss'}, Support: 0.2500
Itemset: {'napkins'}, Support: 0.3000
Itemset: {'fish'}, Support: 0.2500
Itemset: {'potatoes'}, Support: 0.2500
Itemset: {'toilet_paper'}, Support: 0.2000

Apriori Association Rules:

Database 5:

Apriori Frequent Itemsets:
Itemset: {'butter'}, Support: 0.2000
Itemset: {'cleaning_wipes'}, Support: 0.3500
Itemset: {'diapers'}, Support: 0.2000
Itemset: {'apples'}, Support: 0.4000
Itemset: {'onions'}, Support: 0.3500
Itemset: {'eggs'}, Support: 0.2500
Itemset: {'carrots'}, Support: 0.3000
Itemset: {'bread'}, Support: 0.2000
Itemset: {'paper_towels'}, Support: 0.2000
Itemset: {'milk'}, Support: 0.3000
Itemset: {'bananas'}, Support: 0.2500
Itemset: {'toothpaste'}, Support: 0.3000
Itemset: {'oranges'}, Support: 0.2500
Itemset: {'floss'}, Support: 0.3000
Itemset: {'broccoli'}, Support: 0.4000
Itemset: {'napkins'}, Support: 0.2000
Itemset: {'dish_soap'}, Support: 0.2500
Itemset: {'shampoo'}, Support: 0.3000
Itemset: {'potatoes'}, Support: 0.2500
Itemset: {'broccoli', 'dish_soap'}, Support: 0.2000
Itemset: {'shampoo', 'potatoes'}, Support: 0.2000
Itemset: {'apples', 'broccoli'}, Support: 0.2000
Itemset: {'broccoli', 'milk'}, Support: 0.2000
Itemset: {'apples', 'toothpaste'}, Support: 0.2000
Itemset: {'broccoli', 'eggs'}, Support: 0.2000
Itemset: {'cleaning_wipes', 'milk'}, Support: 0.2000
Itemset: {'shampoo', 'onions'}, Support: 0.2000

Apriori Association Rules:
Rule: {'dish_soap'} -> {'broccoli'}, Confidence: 0.8000
Rule: {'potatoes'} -> {'shampoo'}, Confidence: 0.8000
Rule: {'shampoo'} -> {'potatoes'}, Confidence: 0.6667
Rule: {'milk'} -> {'broccoli'}, Confidence: 0.6667
Rule: {'toothpaste'} -> {'apples'}, Confidence: 0.6667
Rule: {'eggs'} -> {'broccoli'}, Confidence: 0.8000
Rule: {'milk'} -> {'cleaning_wipes'}, Confidence: 0.6667
Rule: {'shampoo'} -> {'onions'}, Confidence: 0.6667
```

- The Apriori algorithm works by generating candidate itemsets of size 1, then iteratively combining frequent itemsets to form larger itemsets.

- After generating itemsets, it filters them based on the minimum support.

- Once all frequent itemsets are found, it generates association rules by calculating their confidence and comparing it to the minimum confidence threshold.

Brute Force Method Implementation

In [7]:

```
# Part 2: Brute Force Method
# Brute force method for finding frequent itemsets
def brute_force(transactions, min_support=0.2, min_confidence=0.6):
    transactions = list(map(set, transactions))  # Convert transactions to sets
    all_frequent_itemsets = {}  # To store all frequent itemsets
    k = 1  # Start with 1-itemsets

    while True:
        # Step 1: Generate all possible k-itemsets
        items_in_transactions = set(item for transaction in transactions for item in transaction)
        candidate_itemsets = list(combinations(items_in_transactions, k))

        # Step 2: Filter itemsets by support
        frequent_itemsets = {}
        for itemset in candidate_itemsets:
            itemset = frozenset(itemset)
            support = calculate_support(itemset, transactions)
            if support >= min_support:
                frequent_itemsets[itemset] = support

        # Step 3: Terminate if no frequent itemsets found
        if not frequent_itemsets:
            break

        # Add to all frequent itemsets
        all_frequent_itemsets.update(frequent_itemsets)
        k += 1  # Increase the size of the itemset


    return all_frequent_itemsets
```

In [8]:

```
# Brute Force results
def brute_results(databases, min_support, min_confidence, items):
    for i, transactions in enumerate(databases):
        print(f"\nDatabase {i + 1}:")

        # Brute Force
        start_time = time.time()
        frequent_itemsets_brute = brute_force(transactions, min_support)
        brute_time = time.time() - start_time

        # Output frequent itemsets from Brute
        print("\nBrute Frequent Itemsets:")
        for itemset, support in frequent_itemsets_brute.items():
            print(f"Itemset: {set(itemset)}, Support: {support:.4f}")

        # Generate and output association rules from Brute
        print("\nBrute Force Association Rules:")
        rules_bruteforce = generate_association_rules(frequent_itemsets_brute.keys(), transactions, min_confidence)
        for antecedent, consequent, confidence in rules_bruteforce:
            print(f"Rule: {set(antecedent)} -> {set(consequent)}, Confidence: {confidence:.4f}")


# Set minimum support and confidence
min_support = 0.2
min_confidence = 0.6

# Compare the two algorithms on all 5 databases
brute_results(databases, min_support, min_confidence, items)
```

```
Database 1:

Brute Frequent Itemsets:
Itemset: {'detergent'}, Support: 0.3500
Itemset: {'carrots'}, Support: 0.2500
Itemset: {'grapes'}, Support: 0.3500
Itemset: {'butter'}, Support: 0.2000
Itemset: {'floss'}, Support: 0.2500
Itemset: {'eggs'}, Support: 0.2000
Itemset: {'tomatoes'}, Support: 0.3000
Itemset: {'chicken'}, Support: 0.2000
Itemset: {'rice'}, Support: 0.2000
Itemset: {'broccoli'}, Support: 0.2500
Itemset: {'onions'}, Support: 0.2000
Itemset: {'dish_soap'}, Support: 0.2000
Itemset: {'cleaning_wipes'}, Support: 0.3000
Itemset: {'oranges'}, Support: 0.2000
Itemset: {'bananas'}, Support: 0.3000
Itemset: {'milk'}, Support: 0.2000
Itemset: {'detergent', 'grapes'}, Support: 0.2500
Itemset: {'chicken', 'grapes'}, Support: 0.2000
Itemset: {'cleaning_wipes', 'milk'}, Support: 0.2000

Brute Force Association Rules:
Rule: {'grapes'} -> {'detergent'}, Confidence: 0.7143
Rule: {'detergent'} -> {'grapes'}, Confidence: 0.7143
Rule: {'chicken'} -> {'grapes'}, Confidence: 1.0000
Rule: {'milk'} -> {'cleaning_wipes'}, Confidence: 1.0000
Rule: {'cleaning_wipes'} -> {'milk'}, Confidence: 0.6667

Database 2:

Brute Frequent Itemsets:
Itemset: {'apples'}, Support: 0.2500
Itemset: {'detergent'}, Support: 0.2000
Itemset: {'cereal'}, Support: 0.2000
Itemset: {'eggs'}, Support: 0.4000
Itemset: {'tomatoes'}, Support: 0.2000
Itemset: {'beef'}, Support: 0.2000
Itemset: {'toilet_paper'}, Support: 0.2000
Itemset: {'broccoli'}, Support: 0.3000
Itemset: {'onions'}, Support: 0.2000
Itemset: {'napkins'}, Support: 0.2500
Itemset: {'shampoo'}, Support: 0.2500
Itemset: {'oranges'}, Support: 0.2500
Itemset: {'bread'}, Support: 0.3000

Brute Force Association Rules:

Database 3:

Brute Frequent Itemsets:
Itemset: {'detergent'}, Support: 0.2000
Itemset: {'apples'}, Support: 0.3000
Itemset: {'potatoes'}, Support: 0.2500
Itemset: {'grapes'}, Support: 0.3500
Itemset: {'butter'}, Support: 0.2500
Itemset: {'toothpaste'}, Support: 0.2000
Itemset: {'cereal'}, Support: 0.2000
Itemset: {'eggs'}, Support: 0.3500
Itemset: {'pasta'}, Support: 0.2500
Itemset: {'toilet_paper'}, Support: 0.2500
Itemset: {'paper_towels'}, Support: 0.2000
Itemset: {'fish'}, Support: 0.2500
Itemset: {'rice'}, Support: 0.2000
Itemset: {'broccoli'}, Support: 0.2000
Itemset: {'onions'}, Support: 0.2500
Itemset: {'napkins'}, Support: 0.2500
Itemset: {'cleaning_wipes'}, Support: 0.2500
Itemset: {'oranges'}, Support: 0.2000
Itemset: {'diapers'}, Support: 0.2000
Itemset: {'bananas'}, Support: 0.3500
Itemset: {'milk'}, Support: 0.2000
Itemset: {'grapes', 'bananas'}, Support: 0.2000
Itemset: {'diapers', 'bananas'}, Support: 0.2000

Brute Force Association Rules:
Rule: {'diapers'} -> {'bananas'}, Confidence: 1.0000

Database 4:

Brute Frequent Itemsets:
Itemset: {'apples'}, Support: 0.3000
Itemset: {'potatoes'}, Support: 0.2500
Itemset: {'grapes'}, Support: 0.3000
Itemset: {'butter'}, Support: 0.2000
Itemset: {'floss'}, Support: 0.2500
Itemset: {'toothpaste'}, Support: 0.2500
Itemset: {'cereal'}, Support: 0.2000
Itemset: {'pasta'}, Support: 0.2000
Itemset: {'beef'}, Support: 0.3000
Itemset: {'toilet_paper'}, Support: 0.2000
Itemset: {'fish'}, Support: 0.2500
Itemset: {'napkins'}, Support: 0.3000
Itemset: {'diapers'}, Support: 0.2500
Itemset: {'bananas'}, Support: 0.2000

Brute Force Association Rules:

Database 5:

Brute Frequent Itemsets:
Itemset: {'apples'}, Support: 0.4000
Itemset: {'potatoes'}, Support: 0.2500
Itemset: {'carrots'}, Support: 0.3000
Itemset: {'butter'}, Support: 0.2000
Itemset: {'floss'}, Support: 0.3000
Itemset: {'toothpaste'}, Support: 0.3000
Itemset: {'eggs'}, Support: 0.2500
Itemset: {'paper_towels'}, Support: 0.2000
Itemset: {'broccoli'}, Support: 0.4000
Itemset: {'onions'}, Support: 0.3500
Itemset: {'dish_soap'}, Support: 0.2500
Itemset: {'napkins'}, Support: 0.2000
Itemset: {'shampoo'}, Support: 0.3000
Itemset: {'cleaning_wipes'}, Support: 0.3500
Itemset: {'oranges'}, Support: 0.2500
Itemset: {'bread'}, Support: 0.2000
Itemset: {'diapers'}, Support: 0.2000
Itemset: {'bananas'}, Support: 0.2500
Itemset: {'milk'}, Support: 0.3000
Itemset: {'apples', 'toothpaste'}, Support: 0.2000
Itemset: {'apples', 'broccoli'}, Support: 0.2000
Itemset: {'potatoes', 'shampoo'}, Support: 0.2000
Itemset: {'broccoli', 'eggs'}, Support: 0.2000
Itemset: {'broccoli', 'dish_soap'}, Support: 0.2000
Itemset: {'broccoli', 'milk'}, Support: 0.2000
Itemset: {'shampoo', 'onions'}, Support: 0.2000
Itemset: {'cleaning_wipes', 'milk'}, Support: 0.2000

Brute Force Association Rules:
Rule: {'toothpaste'} -> {'apples'}, Confidence: 0.6667
Rule: {'shampoo'} -> {'potatoes'}, Confidence: 0.6667
Rule: {'potatoes'} -> {'shampoo'}, Confidence: 0.8000
Rule: {'eggs'} -> {'broccoli'}, Confidence: 0.8000
Rule: {'dish_soap'} -> {'broccoli'}, Confidence: 0.8000
Rule: {'milk'} -> {'broccoli'}, Confidence: 0.6667
Rule: {'shampoo'} -> {'onions'}, Confidence: 0.6667
Rule: {'milk'} -> {'cleaning_wipes'}, Confidence: 0.6667
```

- The Brute Force method generates all possible itemsets of increasing size (starting from size 1).

- It filters itemsets based on support but does not use any optimizations (like Apriori) to limit the number of candidate itemsets. This results in a much higher computational cost.

- The following function takes the results and prints for each database.

Comparison of Algorithms

In [9]:

```
# Comparing brute force and apriori
def compare_algorithms(databases, min_support, min_confidence, items):
    for i, transactions in enumerate(databases):
        print(f"\nComparison for Database {i + 1}:")

        # Apriori method
        start_time = time.time()
        frequent_itemsets_apriori, rules_apriori = apriori(transactions, items, min_support, min_confidence)
        apriori_time = time.time() - start_time
        print(f"Apriori Association Rules Count: {len(rules_apriori)}, Time: {apriori_time:.6f} seconds")

        # Brute force method
        start_time = time.time()
        frequent_itemsets_bruteforce = brute_force(transactions, min_support)
        rules_bruteforce = generate_association_rules(frequent_itemsets_bruteforce.keys(), transactions, min_confidence)
        bruteforce_time = time.time() - start_time
        print(f"Brute Force Association Rules Count: {len(rules_bruteforce)}, Time: {bruteforce_time:.6f} seconds")

# Set minimum support and confidence for comparison
min_support = 0.2
min_confidence = 0.6

# Compare the two algorithms (Apriori and Brute) on all 5 databases
compare_algorithms(databases, min_support, min_confidence, items)
```

```
Comparison for Database 1:
Apriori Association Rules Count: 5, Time: 0.001103 seconds
Brute Force Association Rules Count: 5, Time: 0.011115 seconds

Comparison for Database 2:
Apriori Association Rules Count: 0, Time: 0.000779 seconds
Brute Force Association Rules Count: 0, Time: 0.000789 seconds

Comparison for Database 3:
Apriori Association Rules Count: 1, Time: 0.001988 seconds
Brute Force Association Rules Count: 1, Time: 0.009172 seconds

Comparison for Database 4:
Apriori Association Rules Count: 0, Time: 0.000917 seconds
Brute Force Association Rules Count: 0, Time: 0.000914 seconds

Comparison for Database 5:
Apriori Association Rules Count: 8, Time: 0.001774 seconds
Brute Force Association Rules Count: 8, Time: 0.006785 seconds
```

- This function compares the Apriori and Brute Force methods on multiple databases.

- For each database, it prints the number of association rules generated and the time taken by each algorithm.

- The output shows how Apriori is faster while generating the same number of association rules as the Brute Force method.

Conclusion

In this comparison, both the Apriori algorithm and the Brute Force method generate the same number of association rules for each database, as they are both designed to find all rules that meet the specified minimum support and confidence thresholds. However, the Apriori algorithm significantly outperforms the Brute Force method in terms of execution time due to its optimization of candidate generation.