Name: Arjun Reddy
NJIT UCID: ar2752
Email Address: ar2752@njit.edu
Date: 3/10/2024
Professor: Yasser Abduallah
CS 634: Data Mining

Midterm Project Report

Abstract
This project embarks on an in-depth exploration of retail data mining by implementing and comparing three distinct approaches: a custom brute force method, the Apriori algorithm, and the FP-Growth algorithm. It innovates in its comprehensive analysis of transaction data, aiming to discover patterns and associations that shed light on customer purchasing behaviors. By juxtaposing the traditional brute force method against more sophisticated algorithms, the project evaluates their relative effectiveness, efficiency, and practicality in real-world retail applications. This comparative study not only benchmarks the performance of these algorithms but also contributes to the broader understanding of how data mining can be optimized for actionable business insights.

Introduction

Data mining emerges as an influential technique for detecting unseen relationships and patterns buried within extensive datasets. This project is centered around the application of the Apriori Algorithm, an established method for the discovery of association rules, especially in retail environments. We aim to explicate the primary data mining concepts and methodologies implemented in our investigation.

At the heart of the Apriori Algorithm is its capacity to establish connections among items. Initiating this process required identifying items that frequently appear across a multitude of transactions. Upon pinpointing these frequently occurring items, it becomes necessary to calculate their support values according to a threshold defined by the user. Subsequently, items that do not satisfy the minimum support criteria are pruned away. Distinguished by its systematic approach to exhaustively search for frequent item-sets and to formulate association rules, the Apriori Algorithm incrementally increases the breadth of item-sets, filtering out those that do not meet the minimum support requirement.

Libraries/ Modules Used

pip install mlxtend
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, fpgrowth, association_rules

Project Workflow

- **Data Loading and Preprocessing**: The initial phase involves loading transaction data from a retail dataset, where each transaction is represented by a list of purchased items. Preprocessing steps are undertaken to ensure data integrity, including filtering unique items and organizing them according to a predefined hierarchy.
- **Minimum Support and Confidence Determination**: This critical stage involves gathering user inputs to set the minimum thresholds for support and confidence levels, essential for sifting through the data to identify significant patterns.
- **Iterative Candidate Item-set Generation**: Employing the Apriori algorithm, the project iteratively generates candidate item-sets of increasing size, from individual items to larger combinations, using a systematic approach to explore all potential item-set permutations.
- **Support Count Calculation**: For each candidate item-set, we calculate its support by determining the frequency of its occurrence across all transactions. Only those item-sets meeting the minimum support threshold proceed to the next analysis phase.
- **Confidence Evaluation**: The confidence of potential association rules is assessed to gauge the strength of item associations, ensuring that only the most reliable and meaningful patterns are considered for rule generation.
- **Association Rule Extraction**: The final step involves extracting association rules that meet both the support and confidence criteria, revealing insightful patterns about items frequently purchased together.

Brute Force Method

A novel aspect of this project is the brute force method for frequent item-set discovery. This approach meticulously examines every possible item-set combination from the transaction data to identify those exceeding a user-

defined support threshold. This exhaustive method ensures comprehensive analysis at the cost of increased computational effort.

Transaction Encoder

Key to processing transaction data for Apriori and FP-Growth algorithms is the Transaction Encoder. This component transforms transaction lists into a binary matrix representation, facilitating efficient analysis by these algorithms. This preprocessing step is crucial for converting raw transaction data into a format suitable for mining algorithms.

Apriori and FP-Growth Algorithms
Following the brute force and preprocessing stages, the project employs the Apriori and FP-Growth algorithms. These methods offer optimized pathways to identify frequent item-sets without examining every possible combination, significantly reducing computational load compared to the brute force approach.

Core Concepts and Principles

Frequent Item-set Discovery: Identifying commonly purchased item combinations to understand customer purchasing behavior.

Support and Confidence Metrics: Utilizing these metrics to measure the frequency of item-sets and the likelihood of item co-occurrence.

Association Rule Learning: Generating rules that predict the presence of an item based on the presence of other items in a transaction.
Project Workflow

Data Preparation: Transaction data is loaded and preprocessed, converting item lists into a format suitable for analysis.
Brute Force Item-set Discovery: All possible item-sets are examined to find those meeting the minimum support criterion

Transaction Encoding: Raw data is transformed into a binary matrix, enabling efficient analysis by Apriori and FP-Growth algorithms.

Algorithmic Analysis: Both Apriori and FP-Growth algorithms are applied to identify frequent item-sets and generate association rules

Rule Generation and Validation: Association rules are derived using combinatorics and compared across methods to ensure consistency and accuracy.

Results and Evaluation:
The effectiveness of the Apriori and FP-Growth algorithms is assessed through a analysis of the association rules generated, considering factors such as execution time, and set equality. This evaluation not only benchmarks the algorithms against each other but also against existing implementations, offering a clear perspective on their relative merits and limitations in the context of retail data mining.

Conclusion:
Overall, this project underscores the vital role of the Apriori algorithm and associated data mining techniques such as FP-Growth in uncovering patterns within retail transaction data. The structured workflow, from data preprocessing to the iterative generation of item-sets and rules, demonstrates the simple yet powerful nature of these algorithms.

db_1 Snapshot

Items
Charmin Ultra Soft Toilet Paper, Dawn Dish Soap
Glad Trash Bags, Bounty Paper Towels, Tide Laundry Detergent
Dawn Dish Soap, Charmin Ultra Soft Toilet Paper
Charmin Ultra Soft Toilet Paper
Bounty Paper Towels, Tide Laundry Detergent, Dawn Dish Soap, Glad Trash Bags, Charmin Ultra Soft Toilet Paper
Glad Trash Bags, Bounty Paper Towels, Tide Laundry Detergent, Dawn Dish Soap
Bounty Paper Towels, Dawn Dish Soap, Charmin Ultra Soft Toilet Paper, Glad Trash Bags
Glad Trash Bags, Bounty Paper Towels, Dawn Dish Soap
Dawn Dish Soap, Charmin Ultra Soft Toilet Paper
Tide Laundry Detergent, Charmin Ultra Soft Toilet Paper, Bounty Paper Towels
Charmin Ultra Soft Toilet Paper
Dawn Dish Soap, Bounty Paper Towels, Charmin Ultra Soft Toilet Paper
Dawn Dish Soap
Dawn Dish Soap, Glad Trash Bags
Bounty Paper Towels, Glad Trash Bags, Tide Laundry Detergent, Charmin Ultra Soft Toilet Paper, Dawn Dish Soap
Charmin Ultra Soft Toilet Paper, Bounty Paper Towels, Glad Trash Bags, Dawn Dish Soap, Tide Laundry Detergent
Bounty Paper Towels, Glad Trash Bags, Tide Laundry Detergent, Charmin Ultra Soft Toilet Paper
Tide Laundry Detergent, Dawn Dish Soap, Charmin Ultra Soft Toilet Paper

db_2 Snapshot

Items
Pantene Shampoo, Gillette Razors, Neutrogena Moisturizer, Crest Toothpaste
Dove Soap, Neutrogena Moisturizer
Crest Toothpaste, Dove Soap, Pantene Shampoo, Gillette Razors
Neutrogena Moisturizer, Pantene Shampoo, Dove Soap, Crest Toothpaste, Gillette Razors
Gillette Razors
Dove Soap, Neutrogena Moisturizer, Pantene Shampoo, Gillette Razors
Pantene Shampoo, Neutrogena Moisturizer, Crest Toothpaste
Neutrogena Moisturizer
Pantene Shampoo, Neutrogena Moisturizer, Crest Toothpaste
Dove Soap, Neutrogena Moisturizer, Crest Toothpaste
Dove Soap, Crest Toothpaste
Crest Toothpaste, Pantene Shampoo
Dove Soap, Pantene Shampoo, Crest Toothpaste, Neutrogena Moisturizer
Pantene Shampoo, Neutrogena Moisturizer, Gillette Razors, Dove Soap, Crest Toothpaste
Dove Soap, Gillette Razors, Neutrogena Moisturizer, Crest Toothpaste
Crest Toothpaste, Pantene Shampoo, Gillette Razors
Gillette Razors, Dove Soap, Neutrogena Moisturizer, Crest Toothpaste, Pantene Shampoo
Neutrogena Moisturizer
Dove Soap, Crest Toothpaste, Neutrogena Moisturizer, Pantene Shampoo, Gillette Razors
Neutrogena Moisturizer, Gillette Razors, Pantene Shampoo, Crest Toothpaste
Pantene Shampoo
Pantene Shampoo, Neutrogena Moisturizer, Gillette Razors, Crest Toothpaste
Crest Toothpaste, Dove Soap, Neutrogena Moisturizer, Pantene Shampoo
Neutrogena Moisturizer
Gillette Razors
Dove Soap, Gillette Razors, Crest Toothpaste, Pantene Shampoo, Neutrogena Moisturizer
Pantene Shampoo, Gillette Razors

Screenshots

(db_1: Stop & Shop transactions, db_2: Walgreen transactions)

(Prompts user to select database and input minimum support and confidence)

```
mapping = {1:'db_1.csv',2:'db_2.csv',3:'db_3.csv',4:'db_4.csv',5:'db_5.csv'}
file_path = mapping[database]
while True:
    min_support_input = input("Enter minimum support as a percentage ranging 1 - 100 % (e.g., 2% -> 2): ")
    try:
        min_support = float(min_support_input) / 100
        if 0 < min_support <= 1:
            break
        else:
            print("Error: Minimum support must be between 1 and 100.")
    except ValueError:
        print("Error: Please enter a valid number.")

    # Prompt for minimum confidence with validation
while True:
    min_confidence_input = input("Enter minimum confidence as a percentage ranging 1 - 100 % (e.g., 2% -> 2): ")
    try:
        min_confidence = float(min_confidence_input) / 100
        if 0 < min_confidence <= 1:
            break
        else:
            print("Error: Minimum confidence must be between 1 and 100.")
    except ValueError:
        print("Error: Please enter a valid number.")
```

```
Here are databases you can explore....
1) Shop & Stop
2) Walgreens
3) Kroger
4) Walmart
5) CVS
Pick a database by number:  2
Enter minimum support as a percentage ranging 1 - 100 % (e.g., 2% -> 2):  32
Enter minimum confidence as a percentage ranging 1 - 100 % (e.g., 2% -> 2):  12
```

```
def load_transactions_from_csv(file_path):
    df = pd.read_csv(file_path)
    transactions = df['Items'].apply(lambda x: set(x.split(', '))).tolist()
    return transactions
transactions = load_transactions_from_csv(file_path)
transactions_list = [list(t) for t in transactions]
```

(Load Transactions from file)

(Iterative algorithm to generate frequent item sets in brute force fashion)

```
def find_frequent_itemsets_bruteforce(transactions, min_support):
    all_items = set(item for transaction in transactions for item in transaction)
    total_transactions = len(transactions)
    max_itemset_size = max([len(transaction) for transaction in transactions])
    def calc_support(itemset):
        count = sum(1 for trans in transactions if itemset.issubset(trans))
        return count / total_transactions

    current_itemsets = [frozenset([item]) for item in all_items]
    frequent_itemsets = []
    k = 1

    while current_itemsets:
        new_frequent_itemsets = [itemset for itemset in current_itemsets if calc_support(itemset) >= min_support]
        if not new_frequent_itemsets:
            break
        frequent_itemsets.extend(new_frequent_itemsets)
        if k < max_itemset_size:
            next_level_itemsets = set()
            for i in range(len(new_frequent_itemsets)):
                for j in range(i + 1, len(new_frequent_itemsets)):
                    union_set = new_frequent_itemsets[i].union(new_frequent_itemsets[j])
                    if len(union_set) == k + 1:
                        next_level_itemsets.add(union_set)
            current_itemsets = list(next_level_itemsets)
        else:
            break
        k += 1

    return frequent_itemsets
```

(Iterative algorithm to generate rule sets in brute force fashion)

```
def generate_rules(frequent_itemsets, transactions, min_confidence):
    rules = []

    def calc_support(itemset):
        return sum(1 for trans in transactions if itemset.issubset(trans)) / len(transactions)

    def calc_confidence(antecedent, consequent):
        antecedent_consequent = antecedent.union(consequent)
        return calc_support(antecedent_consequent) / calc_support(antecedent)

    for itemset in frequent_itemsets:
        for antecedent in chain.from_iterable(combinations(itemset, r) for r in range(1, len(itemset))):
            antecedent_set = frozenset(antecedent)
            consequent_set = itemset - antecedent_set
            confidence = calc_confidence(antecedent_set, consequent_set)
            if confidence >= min_confidence:
                rules.append((antecedent_set, consequent_set, calc_support(itemset), confidence))
    return rules
```

(Encode transactions in TransactionEncoder() object to handle mlxtend library algorithms FP-Growth and Apriori)

```
def run_algorithm(transactions, min_support, min_confidence, algorithm='apriori'):
    te = TransactionEncoder()
    te_ary = te.fit(transactions).transform(transactions)
    df = pd.DataFrame(te_ary, columns=te.columns_)
    start_time = time.time()
    if algorithm == 'apriori':
        frequent_itemsets = apriori(df, min_support=min_support, use_colnames=True)
    else:
        frequent_itemsets = fpgrowth(df, min_support=min_support, use_colnames=True)
    rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=min_confidence)
    print(f"\n{algorithm.capitalize()} Execution Time: {time.time() - start_time} seconds")
    return rules

def print_rules(rules, title):
    print(f"\n{title}:")
    for rule in rules:
        antecedent, consequent, support, confidence = rule
        # Convert frozensets to sorted lists and then to strings for nicer printing
        antecedent_str = ', '.join(sorted(antecedent))
        consequent_str = ', '.join(sorted(consequent))
        print(f"Rule: {{{antecedent_str}}} => {{{consequent_str}}}, Support: {support:.2f}, Confidence: {confidence:.2f}")

def format_mlxtend_rules(rules_df):
    formatted_rules = []
    for _, row in rules_df.iterrows():
        antecedent = frozenset(row['antecedents'])
        consequent = frozenset(row['consequents'])
        support = row['support']
        confidence = row['confidence']
        formatted_rules.append((antecedent, consequent, support, confidence))
    return formatted_rules
```

(Checks for rule equivalence and formats rules being printed for each database)

```
def rules_to_set(rules):
    """Convert rules into a set of immutable elements for comparison."""
    return set(
        (frozenset(antecedent), frozenset(consequent), round(support, 4), round(confidence, 4))
        for antecedent, consequent, support, confidence in rules
    )

def validate_rules_equivalence(brute_force_rules, mlxtend_rules_df):
    """
    Validates that the sets of rules from the Brute Force method and the MLxtend method
    (Apriori or FP-Growth) are exactly equivalent, without considering the order of the
    rules.

    :param brute_force_rules: Rules generated by the brute force method.
    :param mlxtend_rules_df: Rules generated by the MLxtend method (DataFrame format).
    :return: True if the sets of rules are exactly equivalent, False otherwise.
    """

    # Convert brute_force_rules to set for easy comparison
    brute_force_set = rules_to_set(brute_force_rules)

    # Convert mlxtend_rules_df DataFrame to a set of tuples
    mlxtend_set = set()
    for _, row in mlxtend_rules_df.iterrows():
        antecedent = frozenset(row['antecedents'])
        consequent = frozenset(row['consequents'])
        support = round(row['support'], 4)  # Assuming support is a column in the DataFrame
        confidence = round(row['confidence'], 4)  # Assuming confidence is a column
        mlxtend_set.add((antecedent, consequent, support, confidence))

    # Directly compare the sets
    return brute_force_set == mlxtend_set
```

```
4]: # Execute Brute Force Method
start_time = time.time()
frequent_itemsets = find_frequent_itemsets_bruteforce(transactions, min_support)
print(frequent_itemsets, "Frequwnt itemsets from Brute-Force")
brute_force_rules = generate_rules(frequent_itemsets, transactions, min_confidence)
end_time = time.time()
print(f"\nBrute Force Execution Time: {end_time - start_time} seconds")
print_rules(brute_force_rules, "Brute Force Rules")
```

```
:   # Execute Apriori
start_time = time.time()
apriori_rules_df = run_algorithm(transactions_list, min_support, min_confidence, 'apriori')
apriori_rules = format_mlxtend_rules(apriori_rules_df)
end_time = time.time()
print(f"Apriori Execution Time: {end_time - start_time} seconds")
print_rules(apriori_rules, "Apriori Rules")
```

```
:  # Execute FP-Growth
start_time = time.time()
fp_growth_rules_df = run_algorithm(transactions_list, min_support, min_confidence, 'fpgrowth')
fp_growth_rules = format_mlxtend_rules(fp_growth_rules_df)
end_time = time.time()
print(f"FP-Growth Execution Time: {end_time - start_time} seconds")
print_rules(fp_growth_rules, "FP-Growth Rules")
```

(Verified Custom Brute-force method results with mlxtend algorithms)

```
:  are_apriori_rules_equivalent = validate_rules_equivalence(brute_force_rules, apriori_rules_df)
print(f"Brute Force and Apriori rules are equivalent: {are_apriori_rules_equivalent}")

# Validate equivalence of Brute Force and FP-Growth rules
are_fp_growth_rules_equivalent = validate_rules_equivalence(brute_force_rules, fp_growth_rules_df)
print(f"Brute Force and FP-Growth rules are equivalent: {are_fp_growth_rules_equivalent}")
```

```
Brute Force and Apriori rules are equivalent: True
Brute Force and FP-Growth rules are equivalent: True
```

Results (From terminal) **(Association rules/ Execution Times)**

Based on this example the brute force method actually executed the fasted on this input. The brute force rule sets were identical to the rule sets generated by the Growth and their Apriori

```
Here are databases you can explore....
1) Stop & Shop
2) Walgreens
3) Kroger
4) Walmart
5) CVS
Pick a database by number: 3
Kroger Chosen!
Enter minimum support as a percentage ranging 1 - 100 % (e.g., 2% -> 2): 36
Enter minimum confidence as a percentage ranging 1 - 100 % (e.g., 2% -> 2): 23
Transactions from this database

[{'Kraft Macaroni & Cheese', 'Haagen-Dazs Ice Cream'}, {"Campbell's Soup", 'Green Giant Frozen Vegetables', 'Haagen-Dazs Ice Cream'}, {'Kr
aft Macaroni & Cheese', "Campbell's Soup"}, {'Kraft Macaroni & Cheese', 'Green Giant Frozen Vegetables', 'Haagen-Dazs Ice Cream', "Campbel
l's Soup"}, {"Campbell's Soup"}, {'Chiquita Bananas'}, {'Green Giant Frozen Vegetables', 'Haagen-Dazs Ice Cream', 'Kraft Macaroni & Cheese
', "Campbell's Soup", 'Chiquita Bananas'}, {'Kraft Macaroni & Cheese', "Campbell's Soup", 'Chiquita Bananas'}, {"Campbell's Soup", 'Green
Giant Frozen Vegetables'}, {'Kraft Macaroni & Cheese', "Campbell's Soup", 'Chiquita Bananas'}, {'Green Giant Frozen Vegetables', 'Haagen-D
azs Ice Cream', 'Kraft Macaroni & Cheese', "Campbell's Soup", 'Chiquita Bananas'}, {'Green Giant Frozen Vegetables', 'Haagen-Dazs Ice Crea
m', 'Kraft Macaroni & Cheese', "Campbell's Soup", 'Chiquita Bananas'}, {"Campbell's Soup"}, {'Kraft Macaroni & Cheese', 'Chiquita Bananas'
}, {'Kraft Macaroni & Cheese', 'Green Giant Frozen Vegetables'}, {'Kraft Macaroni & Cheese'}, {'Green Giant Frozen Vegetables', 'Chiquita
Bananas', 'Haagen-Dazs Ice Cream', "Campbell's Soup"}, {'Kraft Macaroni & Cheese', "Campbell's Soup"}, {'Kraft Macaroni & Cheese', 'Haagen
-Dazs Ice Cream'}, {'Kraft Macaroni & Cheese', "Campbell's Soup", 'Haagen-Dazs Ice Cream'}, {'Kraft Macaroni & Cheese', 'Green Giant Froze
n Vegetables', 'Chiquita Bananas', 'Haagen-Dazs Ice Cream'}]
Frequent item sets from Brute Force Method

[('Green Giant Frozen Vegetables',), ('Haagen-Dazs Ice Cream',), ('Kraft Macaroni & Cheese',), ("Campbell's Soup",), ('Chiquita Bananas',)
, ('Kraft Macaroni & Cheese', 'Haagen-Dazs Ice Cream'), ('Kraft Macaroni & Cheese', "Campbell's Soup")]

Brute Force Execution Time: 0.0002460479736328125 seconds

Brute Force Rules:
Rule: {Kraft Macaroni & Cheese} => {Haagen-Dazs Ice Cream}, Support: 0.38, Confidence: 0.53
Rule: {Haagen-Dazs Ice Cream} => {Kraft Macaroni & Cheese}, Support: 0.38, Confidence: 0.80
Rule: {Kraft Macaroni & Cheese} => {Campbell's Soup}, Support: 0.43, Confidence: 0.60
Rule: {Campbell's Soup} => {Kraft Macaroni & Cheese}, Support: 0.43, Confidence: 0.64

Apriori Execution Time: 0.006217002868652344 seconds
Apriori Execution Time: 0.006792306900024414 seconds

Apriori Rules:
Rule: {Kraft Macaroni & Cheese} => {Campbell's Soup}, Support: 0.43, Confidence: 0.60
Rule: {Campbell's Soup} => {Kraft Macaroni & Cheese}, Support: 0.43, Confidence: 0.64
Rule: {Kraft Macaroni & Cheese} => {Haagen-Dazs Ice Cream}, Support: 0.38, Confidence: 0.53
Rule: {Haagen-Dazs Ice Cream} => {Kraft Macaroni & Cheese}, Support: 0.38, Confidence: 0.80

Fpgrowth Execution Time: 0.0020668506622314453 seconds
FP-Growth Execution Time: 0.0024309158325195312 seconds

FP-Growth Rules:
Rule: {Kraft Macaroni & Cheese} => {Haagen-Dazs Ice Cream}, Support: 0.38, Confidence: 0.53
Rule: {Haagen-Dazs Ice Cream} => {Kraft Macaroni & Cheese}, Support: 0.38, Confidence: 0.80
Rule: {Kraft Macaroni & Cheese} => {Campbell's Soup}, Support: 0.43, Confidence: 0.60
Rule: {Campbell's Soup} => {Kraft Macaroni & Cheese}, Support: 0.43, Confidence: 0.64
Brute Force and Apriori rules are equivalent: True
Brute Force and FP-Growth rules are equivalent: True
```

implementation in the mlxtend library. While the brute force execution time was faster in this example, this was not always the case on other user defined inputs. However, with respect to the rule sets generated by the Growth and their Apriori implementation in the mlxtend library, every all rule sets were pairwise equivalent on all user defined inputs.

Steps to Run:

(assuming python and pip is installed)
> cd data_mining
> pip install mlxtend
> pip install pandas
(Run Script) > python project_notebook.py

Other
The source code (project_notebook.py file and data sets (.csv files0 will be attached to zip file

Link To Git Repository

https://github.com/ar2752/data_mining