CS634 – Data Mining
Midterm Project Report

Name: Kenil Pravinbhai Avaiya

NJIT UCID: ka683

Email Address: ka683@njit.edu

Date: 10-17-2025

Professor: Dr. Yasser Abdullah

Course: CS-634 101 Data Mining

Title

Frequent item set data mining for diverse transactional data sets from different retailers.

Abstract

The objective of this project is to discover association rules (A -> B, C) and frequent itemset from various transaction datasets using data mining techniques like Apriori and FP-Growth tree. Furthermore, a brute force approach, which I have developed from scratch, will also be employed to accomplish the same task.

The goal is to allow users to select a dataset and the chosen algorithm. After making the selection, data mining will be performed on that dataset, resulting in the printing of the generated association rules along with the time taken to generate them.

Introduction

Retailers are increasingly adopting frequent item set data mining as a popular technique. By analyzing transactional datasets, retailers can uncover core relational relationships and co-occurrence patterns among frequently purchased items. This project aims to explore and compare three distinct algorithms for generating association rules from transaction datasets.

1. **Brute Force Algorithm** – A manually implemented method that systematically checks all possible item combinations to find frequent itemset.
2. **Apriori Algorithm** – An algorithm that optimizes the Apriori property to minimize unnecessary computations by pruning infrequent itemset.

3. **FP-Growth Algorithm** – A tree base approach that efficiently discovers frequent patterns without candidate generation.

To implement the algorithm, multiple transactional datasets from various retailers, including Apple and Amazon, were used. Each algorithm was evaluated using user-defined thresholds for minimum support and confidence, allowing for a comparative analysis of their performance and rule generation efficiency.

Important Concepts, terminology and Principles

Support: It eliminates rare patterns (statistical reliability) based on the frequency of items appearing in the dataset.

Confidence: It eliminates unreliable associations (predictive strength) based on the likelihood of an item being purchased given the presence of another item. In other words, it measures how often the rule is likely to be true.

Frequent Itemset: Itemset that satisfy the minimum support requirement are considered frequent itemset. These are sets of items that appear frequently together in transactions.

Association Rules: Association rules are like connections between things that happen together a lot. They're usually written as A->B, which means if A happens, B is probably going to happen too.

Project Structure and Workflow

1. Preparation of Datasets

This project involved generating five distinct datasets using Kaggle, along with a midterm project example PDF. Each dataset contained approximately ten unique items and was based on those items, resulting in twenty-five transactions. These datasets represent general purchase records from popular retailers that potential buyers might be interested in, including Amazon, Apple, Best Buy, Costco, and Nike. During the project, the program prompted users to select one of these datasets for analysis and specify minimum support and confidence values (thresholds). These parameters play a crucial role in identifying association rules during the mining process.

2. Environment and Installing required libraries and modules

To run the project file, we need some python packages:

- Pandas: use to handle dataset, load the data, as well as clean the data and ready for next analysis step.

- Apyori: to implement apriori data mining
- Mlxtend: to implement FP growth technique.

In my files, if you try to run main.ipynb file. I already define the required libraires, so you do not need to install any package just run the cell and you ready to go.

```
%pip install apyori
%pip install pandas
%pip install mlxtend
```

If you are trying to run the main.py file, then followed bellowed the instruction.

1. **First Install Python if you do not have**

To run python file, we need to install python in our system.

- **Windows:** Download the installer from python.org and add Python to your PATH during installation.
- **Mac:** Python is usually pre-installed; for the latest version, use python.org or brew install python.

2. **Set Up a Virtual Environment**

   To establish a virtual environment, locate the folder containing all the necessary files. Then, open the terminal within that folder and proceed with the following steps.

- Windows Command Prompt:
  python -m venv env # create a python environment with name env
  venv\Scripts\activate # activate the python environment

- Mac / Linux Terminal:
  python3 -m venv env # create a python environment with name env
  source venv/bin/activate # activate the python environment

3. **Install Required Python Packages**

   For your file, install these libraries:
   - Pandas
   - apyori
   - mlxtend
   - tabulate

   Run in your command line or terminal (in the activated environment):
   pip install pandas apyori mlxtend tabulate

4. **Prepare Your Data Files**
   Make sure your dataset CSV files (such as amazon.csv, apple.csv, etc.) are in the same directory as your script or update the script with their full paths.

5. **Run the Python Script**
   Run your main file in your terminal.
   Windows:
   python main.py
   Mac:
   python main.py

3. Code Structure and Project Workflow Overview

This project is organized to allow users to perform association rule mining on transaction datasets from multiple retail stores (Amazon, Apple, Best Buy, Costco, Nike) using different algorithms.

Workflow:

- Store & Dataset Selection:
  The program presents a list of stores to the user. The user selects a store, and the corresponding dataset (CSV file) is loaded.

- Preprocessing:
  Transactions with empty or null entries are removed from the dataset. Each transaction string is split and converted into a list of items (e.g., 'Tv, Usb C cable, Ps5' → ['Tv, ' Usb C cable', 'Ps5']).

- Parameter Input:
  The user is prompted to enter minimum support and confidence thresholds (usually expressed as percentages). These values are converted and used as input parameters for the mining algorithms.

```python
print("Hi there!!\n")

user_input = int(input("Enter the store number you want to move on:: \n1. Amazon \n2. Apple \n3. Best Buy \n4. Costco \n5. Nike \n"))

datasets = ('Amazon', 'Apple', 'Best Buy', 'Costco', 'Nike')

if user_input < 1 or user_input > len(datasets):
    print("You select invalid value!!, retry")
    quit()

def select_store(store):
    data = ''
    if store == 1:
        data = 'amazon.csv'
    elif store == 2:
        data = 'apple.csv'
    elif store == 3:
        data = 'bestbuy.csv'
    elif store == 4:
        data = 'costco.csv'
    elif store == 5:
        data = 'nike.csv'

    return data

print(f"You select {select_store(user_input)}")
df = pd.read_csv(select_store(user_input))
try:
    support_input = int(input("\nEnter the minimum support value between 1 and 100 :: "))
    if support_input < 1 or support_input > 100:
        raise ValueError("Invalid support value.")

    confidence_input = int(input("Enter the confidence level between 1 and 100 :: "))
    if confidence_input < 1 or confidence_input > 100:
        raise ValueError("Invalid confidence value.")

except ValueError as e:
    print(f"Error: {e}. Please retry...")
    quit()
print(f"\nEnter support value: {support_input} \nEnter confidence value: {confidence_input}")

#now we remove all null or empty transaction row from the data set and also split transtion into list. For example: if out transction is A,B,C
df = df[df['Transaction'].apply(lambda x: x.strip() != '' )]
transactions = df['Transaction'].apply(lambda x: [item.strip() for item in x.split(',')]).tolist()
```

Code responsible for taking input from the user.

- Algorithm Selection & Execution:
  Ask the user to choose an algorithm based on their choice.

    - Brute Force Algorithm:
      A brute force algorithm checks every possible option or combination to find the correct solution. It checks all possible answers one by one, without using any shortcut or smart method. This way, it makes sure that all correct answers are found.
      Brute force is easy to understand and use, but it can take a lot of time and computer power when the data set or problem is large.

Brute Force Algorithm

```python
def brute_force(transactions, support, confidence):
    total_tran = len(transactions)
    support_count = support * total_tran

    freq_item = {}
    main_itemsets = {}
    association_rules = []

    # Generate frequency set
    for elem in transactions:
        for length in range(1, len(elem) + 1):
            for i in combinations(elem, length):
                i = tuple(sorted(i))  # this avoids duplicates like {'A', 'B'} and {'B', 'A'}
                if i in freq_item:
                    freq_item[i] += 1
                else:
                    freq_item[i] = 1

    # Filtering frequent itemsets with minimum support
    for itemset, count in freq_item.items():
        if count >= support_count:
            main_itemsets[itemset] = count

    # Generate association rules
    for itemset in main_itemsets:
        if len(itemset) > 1:
            for i in range(1, len(itemset)):
                for left in combinations(itemset, i):
                    right = tuple(sorted(set(itemset) - set(left)))  # A -> B
                    union = tuple(sorted(left + right))  # A ∪ B

                    support_union = main_itemsets.get(union, 0)
                    support_left = main_itemsets.get(left, 0)

                    if support_left > 0:
                        support_confidence = support_union / support_left
                        if support_confidence >= confidence:
                            association_rules.append((left, right, support_confidence))

    # Generating Association rules for output
    final_list = []
    for A, B, confi in association_rules:
        ans_str = f"{{{', '.join(A)}}} → {{{', '.join(B)}}}, Confidence: {confi * 100:.2f}%"
        final_list.append(ans_str)

    return final_list
```

Code represents implementation of brute force.

- Apriori Algorithm (apyori library):
  The Apriori algorithm finds frequent item sets in a unique way by taking combinations that meet the minimum support value condition. It starts with small item sets and gradually builds larger ones using only the frequent items. This helps reduce extra work and makes the process faster.
  The algorithm is often used with libraries like apyori to create association rules from transaction data.

Apriori Algorithm

```python
def apriori_algo(transactions, support, confidence):
    # Implement the apriori algorithm
    ans = list(apriori(transactions, min_support=support, min_confidence=confidence))

    # Generating Association rules for output
    final_list = []
    for elem in ans:
        for stat in elem.ordered_statistics:
            left = ', '.join(stat.items_base)
            right = ', '.join(stat.items_add)
            confidence = stat.confidence * 100
            support = elem.support * 100
            if left:
                ans_str = f"{{{left}}} → {{{right}}}, Confidence: {confidence:.2f}%"
                final_list.append(ans_str)

    return final_list
```

Code represents implementation of Apriori Algorithm.

- FP-Tree Algorithm (mlxtend library):
  The FP-Growth algorithm is a fast method that uses a tree structure to find frequent item sets. It stores all transactions in an **FP-tree**, which helps it find common patterns without creating too many possible combinations. By exploring this tree step by step, FP-Growth can quickly find large item sets that meet the minimum support value.
  In our project, I used the **mlxtend** python library to find frequent item sets and association rules from the transaction data.

FP Tree Algoritham

```python
def fp_growth_tree_algo(transactions, support, confidence):

    # Use TransactionEncoder to Encode the transactions
    encoder = TransactionEncoder()
    encoder_array = encoder.fit(transactions).transform(transactions)
    encoder_dataframe = pd.DataFrame(encoder_array, columns=encoder.columns_)

    # geting frequent itemsets using fpgrowth
    frequent_itemsets = fpgrowth(encoder_dataframe, min_support=support, use_colnames=True)

    if frequent_itemsets.empty:
        print("Don't have frequent itemsets for support")
        return []

    # Generate the association rules
    association_rules_df = association_rules(frequent_itemsets, metric="confidence", min_threshold=confidence)

    if association_rules_df.empty:
        print("Don't have any association rules for itemsets")

    # rules for output
    final_list = []
    for index, row in association_rules_df.iterrows():
        antecedents = ', '.join(row['antecedents'])
        consequents = ', '.join(row['consequents'])
        confidence = row['confidence'] * 100  # Convert confidence to percentage
        and_str = f"{{{antecedents}}} → {{{consequents}}}, Confidence: {confidence:.2f}%"
        final_list.append(and_str)

    return final_list
```

Code represents implementation of FP-Growth tree Algorithm.

- All Algorithms with Execution Time Comparison:
  I also provide an option to select all algorithms simultaneously and execute all three algorithms on the same data. This allows me to compare their execution times and highlight any performance differences.

- Output:
  The discovered association rules are displayed in a human-readable format (e.g., **{Mac} → {Kindle}, Confidence: 92.31%**).
  When the user selects all algorithms to execute, the execution times and rule outputs are presented. The faster algorithm that takes the minimum time is also displayed, along with the total time taken by the respected algorithm table. Simply compare the times to determine the most efficient algorithm.

My Output when I run the code:
I select the Best Buy data set for running and also select all algorithms for running with the minimum support of 50 and the minimum confidence value of 60.

```
Hi there!!

Enter the store number you want to move on::
1. Amazon
2. Apple
3. Best Buy
4. Costco
5. Nike
 3
You select bestbuy.csv

Enter the minimum support value between 1 and 100 ::  50
Enter the confidence level between 1 and 100 ::  60

Enter support value: 50
Enter confidence value: 60
```

Selecting data set and providing support and confidence value

```
Select Algorithm of your choise::
1. Brute Force Algoritham
2. Apriori Algorithm
3. FP Tree Algoritham
4. For All Algorithms
 4

Running Brute Force Algorithm...

--------------------------------
1. {1TB SSD} → {Adobe}, Confidence: 86.67%
2. {Adobe} → {1TB SSD}, Confidence: 100.00%
3. {Anti-Virus} → {Laptop case}, Confidence: 86.67%
4. {Laptop case} → {Anti-Virus}, Confidence: 86.67%
Brute Force time taken: 0.0679250 seconds

Running Apriori Algorithm...
--------------------------------
1. {1TB SSD} → {Adobe}, Confidence: 86.67%
2. {Adobe} → {1TB SSD}, Confidence: 100.00%
3. {Anti-Virus} → {Laptop case}, Confidence: 86.67%
4. {Laptop case} → {Anti-Virus}, Confidence: 86.67%

Apriori time taken: 0.0001900 seconds

Running FP Tree Algorithm...
--------------------------------
1. {Anti-Virus} → {Laptop case}, Confidence: 86.67%
2. {Laptop case} → {Anti-Virus}, Confidence: 86.67%
3. {1TB SSD} → {Adobe}, Confidence: 86.67%
4. {Adobe} → {1TB SSD}, Confidence: 100.00%

FP Tree time taken: 0.0057804 seconds
```

Algorithm	Time
Brute Force	0.067925
Apriori	0.00019
FP-Growth	0.0057804

```
Fastest Algorithm: Apriori with a time of 0.0001900 seconds
```

Selecting algorithm and getting result.

Fastest Algorithm: Apriori library with a time of **0.0009 seconds**

- Brute Force: **0.067925 seconds**
- Apriori: **0.0001356 seconds**
- FP Tree: **0.0057804 seconds**

Additionally, I observed that when the dataset contained many transactions, the Apriori library algorithm exhibited the most efficient execution time. Conversely, with a reduced number of transactions (approximately 10) and fewer items, the Brute Force algorithm occasionally demonstrated the best performance.

Results and Evaluation

Based on our results, the **Apriori algorithm** worked the fastest, followed by the **Brute Force algorithm**. The **FP-Growth Tree algorithm** was the slowest, however in some cases the rime taken by **Fp-Growth tree algorithm** is smaller than **Brute Force algorithm**, but it still gave correct results. Even though their speeds were different, all three algorithms produced accurate and reliable outcomes in our project.

Conclusion

This project presents the implementation and comparative analysis of three widely used algorithms for association rule mining: Brute Force, Apriori, and FP-Tree. While the Brute Force algorithm provides simplicity, it suffers from efficiency limitations. In contrast, the Apriori and FP-Tree algorithms exhibit enhanced efficiency and scalability, making them suitable for handling larger datasets.

GitHub Repository and Jupyter Notebook

The complete source code, datasets, Python file and a Jupyter notebook documenting the process are available on my Github repository. The repository link is provided below:
Note: I used my personal GitHub account (instead of an NJIT email account).
The repository link is provided below:
https://github.com/KenilAvaiyaa/Avaiya_Kenil_Midtermproj/tree/main

Appendices

- Datasets: Five custom retail transaction files (Amazon, Apple, Best Buy, Costco, Nike) in .csv format, each with 10 items across 25 transactions, used for algorithm comparison.
- Source Files: Python script (main.py) and Jupyter notebook (notebook.ipynb) containing all model code and workflow steps.
- Installation Steps: Environment setup and required commands for dependencies (pandas, apyori, mlxtend, tabulate, etc.).
- Algorithm Outputs: Sample results such as discovered rules ({Mac} → {Kindle}, Confidence: 92.31%) and execution times (e.g., Apriori: 0.0001356 sec, FP-Growth: 0.0057804 sec, Brute Force: 0.067925 sec) for the chosen minimum support and confidence.
- Repository Link: Project files and documentation hosted athttps://github.com/KenilAvaiyaa/Avaiya_Kenil_Midtermproj/tree/main

References

- Apyori library: https://www.geeksforgeeks.org/machine-learning/apriori-algorithm/
- Mlxtend: https://rasbt.github.io/mlxtend/
- https://www.geeksforgeeks.org/machine-learning/frequent-pattern-growth-algorithm/
- https://mobigaurav.medium.com/association-rule-mining-apriori-principle-and-brute-force-f6d11c908505