# CS634 – Data Mining
# Midterm Project Report

Student Name: **Mihir Kadam**

Ucid: **mk2436**

Student Email: mk2436@njit.edu

Instructor: **Dr. Yasser Abduallah**

# Introduction

This project focuses on the comparison and analysis of three major algorithms for frequent itemset mining and association rule generation: Brute-Force, Apriori, and FP-Growth. These algorithms play a crucial role in market basket analysis and transactional data mining, revealing relationships between items that are frequently purchased together.

The main objective of this project is to understand how these algorithms differ in terms of efficiency, scalability, and computational complexity when applied to retail transaction data. Each algorithm was executed on multiple store-based datasets to identify frequent itemsets, generate association rules, and measure the runtime performance under varying support and confidence thresholds.

The experiments were conducted using custom Python scripts as well as implementations from the mlxtend.frequent patterns library. The comparison aims to evaluate how optimized algorithms like Apriori and FP-Growth perform relative to the manually implemented brute-force approach.

# Datasets Overview

For this study, we utilized transaction datasets from five different retail stores: Amazon, Best-Buy, Kmart, Nike, and ShopRite. Each dataset consists of an itemset, which lists the products available in the store, and a transaction dataset, which records customer purchases.

## Amazon Dataset

The Amazon dataset contains 10 items related to programming and web development, including books on Java, C++, and HTML/CSS. The transaction dataset consists of 20 transactions representing combinations of these items purchased together.

| Transactio | Transaction |
|---|---|
| Trans1 | A Beginner's Guide,Java: The Complete Reference,Java For Dummies,A |
| Trans2 | A Beginner's Guide,Java: The Complete Reference,Java For Dummies |
| Trans3 | A Beginner's Guide,Java: The Complete Reference,Java For Dummies,A |
| Trans4 | Android Programming: The Big Nerd Ranch,Head First Java 2nd Edition |

## BestBuy Dataset

The BestBuy dataset includes 10 electronic products such as Digital Cameras, Laptops, Desktops, Printers, Flash Drives, Microsoft Office, and related accessories. It contains 20 transactions reflecting various combinations of these products purchased by customers.

| Transactio | Transaction |
|---|---|
| Trans1 | Desk Top, Printer, Flash Drive, Microsoft Office, Speakers, Anti-Virus |
| Trans2 | Lab Top, Flash Drive, Microsoft Office, Lab Top Case, Anti-Virus |
| Trans3 | Lab Top, Printer, Flash Drive, Microsoft Office, Anti-Virus, Lab Top Case, |
| Trans4 | Lab Top, Printer, Flash Drive, Anti-Virus, External Hard-Drive, Lab Top Ca |
| Trans5 | Lab Top, Flash Drive, Lab Top Case, Anti-Virus |

## Kmart Dataset

The Kmart dataset consists of 10 home and bedding items, including Quilts, Bedspreads, Decorative Pillows, Sheets, Shams, Bedding Collections, Kids Bedding, Embroidered Bedspread, Bed Skirts, and Towels. The dataset includes 20 transactions representing purchases of these Items.

| Transactio | Transaction |
|---|---|
| Trans1 | Decorative Pillows, Quilts, Embroidered Bedspread |
| Trans2 | Embroidered Bedspread, Shams, Kids Bedding, Bedding Collections, |
| Trans3 | Decorative Pillows, Quilts, Embroidered Bedspread, Shams, Kids Be |
| Trans4 | Kids Bedding, Bedding Collections, Sheets, Bedspreads, Bed Skirts |

## Nike Dataset

The Nike dataset contains 10 sportswear items, such as Running Shoes, Soccer Shoes, Socks, Swimming Shirts, Dry Fit V-Nick, Rash Guards, Sweatshirts, Hoodies, Tech Pants, and Modern Pants. The transaction dataset consists of 20 transactions capturing various combinations of these items bought by customers.

| Transactio | Transaction |
|---|---|
| Trans1 | Running Shoe, Socks, Sweatshirts, Modern Pants |
| Trans2 | Running Shoe, Socks, Sweatshirts |
| Trans3 | Running Shoe, Socks, Sweatshirts, Modern Pants |
| Trans4 | Running Shoe, Sweatshirts, Modern Pants |

## ShopRite Dataset

The ShopRite dataset includes 10 apparel items, largely overlapping with Nike's, including Dry Fit V-Nick, Hoodies, Modern Pants, Rash Guard, Running Shoes, Soccer Shoes, Socks, Swimming Shirts, Sweatshirts, and Tech Pants. It contains 25 transactions representing customer purchases of these items.

| Transaction ID | Transaction |
|---|---|
| Trans1 | Soccer Shoe, Hoodies, Socks, Running Shoe |
| Trans2 | Tech Pants, Running Shoe, Dry Fit V-Nick |
| Trans3 | Soccer Shoe, Tech Pants, Dry Fit V-Nick, Modern Pant |
| Trans4 | Soccer Shoe, Sweatshirts, Modern Pants |

These datasets were used to perform association rule mining and to analyze purchasing patterns across different stores and product categories.

# Algorithms and Implementation

The three algorithms implemented in this project differ significantly in how they search for frequent itemsets and generate association rules. Below is a detailed description of each method.

## Brute-Force Approach

The brute-force algorithm was implemented manually in Python. It systematically generates all possible combinations of items (itemsets) and calculates the support for each by scanning the dataset multiple times. If an itemset's support meets or exceeds the minimum threshold specified by the user, it is considered frequent.

Although accurate, this approach is computationally expensive due to the exponential number of itemset combinations. For larger datasets, execution time increases drastically. This method was mainly used for validation and baseline comparison.

## Apriori Algorithm

The Apriori algorithm, implemented using the mlxtend.frequent patterns.apriori function, improves on brute-force by applying a pruning strategy. It uses the property that all subsets of a frequent itemset must also be frequent. Hence, it eliminates infrequent candidates early, drastically reducing the search space.

Apriori proceeds iteratively: starting from 1-itemsets, it builds larger itemsets using only those that met the minimum support threshold in previous iterations. This reduces the number of candidate itemsets and speeds up computation.

## FP-Growth Algorithm

FP-Growth, also implemented through mlxtend.frequent patterns.fpgrowth, further optimizes the process by constructing an FP-tree — a compact data structure that stores itemset frequencies without generating candidate sets explicitly. Once the FP-tree is built, it can be mined recursively to extract frequent itemsets directly.

FP-Growth is typically faster than Apriori for larger datasets because it requires fewer scans of the database and avoids combinatorial explosion.

# Implementation and Repository Layout

The repository structure is organized as follows:

```
kadam_mihir_midtermproject/
|-- data/
|       |-- Amazon_itemset.csv
|       |-- Amazon_dataset.csv
|       |-- Bestbuy_itemset.csv
|       |-- Bestbuy_dataset.csv
|       |-- K_mart_itemset.csv
|       |-- K_mart_dataset.csv
|       |-- Nike_itemset.csv
|       |-- Nike_dataset.csv
|       |-- Shoprite_itemset.csv
|       |-- Shoprite_dataset.csv
|-- notebook/
|       |-- mainNoteBook.ipynb
|-- report/
|       |-- kadam_mihir_midtermproject.pdf
|-- README.md
|-- LICENSE
|-- main.py
|-- requirements.txt
```

The ***main.py*** file serves as the entry point of the project. It allows the user to select a dataset, specify minimum support and confidence thresholds, and then runs all three algorithms sequentially. Execution times are measured using the time module, and results such as frequent itemsets, association rules, and runtimes are displayed on the console.

The project depends on several Python libraries, including pandas, mlxtend, itertools, and tabulate. All dependencies are listed in the ***requirements.txt*** file

# Tutorial to run the Code

This section provides a step-by-step tutorial to run the project code and generate association rules for any of the provided datasets.

## Step 1: Environment Setup

Before running the code, ensure your environment is ready:
1. Install Python 3.8 or higher.
2. Install the required Python libraries by running:

*pip install -r requirements.txt*

3. Make sure all dataset CSV files are in the data/ directory of the project.

## Step 2: Run the Wrapper Code

Navigate to the project directory:

*cd kadam_mihir_midtermproject*

Execute the main script.
Windows:

*python main.py*

Linux / macOS:

*python3 main.py*

## Step 3: Interact with the Program

Follow the on-screen prompts to:

- Select one of the datasets: Amazon, BestBuy, ShopRite, Kmart, or Nike.

```
Welcome to ruleFinder!

Available stores:
1. Amazon
2. Bestbuy
3. Shoprite
4. K-mart
5. Nike
Enter the number of the store you want to analyze: 1
```

- Enter the minimum support thresholds for the association rule mining.

```
Enter the minimum support (as % between 1 and 100): 40
```

- Enter the minimum confidence thresholds for the association rule mining.

```
Enter the minimum confidence (as % between 1 and 100): 50
```

The program will automatically:

- Execute all three implemented algorithms.
- Display frequent itemsets and association rules.
- Summarize runtime comparisons across algorithms.

## Step 4: Observe Results

The program will run the Brute Force, Apriori, and FP-Growth algorithms on the selected store's data.
For each algorithm, you'll see:

1. Execution time
2. Frequent item sets found
3. Association rules generated (if any)

- **Brute Force Results**

```
Running Brute Force Algorithm...
Brute Force completed in 0.0271 seconds.


=== Brute Force Results ===

Frequent Itemsets:
```

| Itemset | Support |
|---|---|
| A Beginner's Guide | 55.00% |
| Android Programming: The Big Nerd Ranch | 65.00% |
| Java For Dummies | 65.00% |
| Java: The Complete Reference | 50.00% |
| Java For Dummies, Java: The Complete Reference | 50.00% |

Association Rules:

| Rule | Confidence | Support |
|---|---|---|
| Java For Dummies -> Java: The Complete Reference | 76.92% | 50.00% |
| Java: The Complete Reference -> Java For Dummies | 100.00% | 50.00% |

- **Arpiori Results**

```
Running Apriori Algorithm...
Apriori completed in 0.0863 seconds.


=== Apriori Results ===

Frequent Itemsets:
```

| Itemset | Support |
|---|---|
| A Beginner's Guide | 55.00% |
| Android Programming: The Big Nerd Ranch | 65.00% |
| Java For Dummies | 65.00% |
| Java: The Complete Reference | 50.00% |
| Java For Dummies, Java: The Complete Reference | 50.00% |

Association Rules:

| Rule | Confidence | Support |
|---|---|---|
| Java For Dummies -> Java: The Complete Reference | 76.92% | 50.00% |
| Java: The Complete Reference -> Java For Dummies | 100.00% | 50.00% |

- **FP-Growth Results**

```
Running FP-Growth Algorithm...
FP-Growth completed in 0.0435 seconds.


=== FP-Growth Results ===

Frequent Itemsets:
```

| Itemset | Support |
|---|---|
| Java For Dummies | 65.00% |
| Android Programming: The Big Nerd Ranch | 65.00% |
| A Beginner's Guide | 55.00% |
| Java: The Complete Reference | 50.00% |
| Java For Dummies, Java: The Complete Reference | 50.00% |

Association Rules:

| Rule | Confidence | Support |
|---|---|---|
| Java For Dummies -> Java: The Complete Reference | 76.92% | 50.00% |
| Java: The Complete Reference -> Java For Dummies | 100.00% | 50.00% |

- **Execution TIme comparison**

```
Execution Times (seconds):
+-------------+------------+
| Algorithm   |   Time (s) |
+=============+============+
| Brute Force |     0.0271 |
+-------------+------------+
| Apriori     |     0.0863 |
+-------------+------------+
| FP-Growth   |     0.0435 |
+-------------+------------+

 The fastest algorithm is: Brute Force (0.0271 seconds)
```

# Code Snippets from Jupyter

- Importing Libraries

```
import itertools
import time
import pandas as pd
from mlxtend.frequent_patterns import apriori, fpgrowth, association_rules
from tabulate import tabulate
```

- Function **calculate_support**
  This function helps us understand how popular a specific group of items (an 'itemset') is by looking at all the customer transactions. It figures out what percentage of all transactions include every single item in the itemset you're interested in.

```
def calculate_support(itemset, transactions):
    """
    Return the fraction of transactions containing the itemset.

    Args:
        itemset (tuple or list): Items to check.
        transactions (list of lists): List of transactions.

    Returns:
        float: Support value (0 to 1).
    """
    try:
        return sum(1 for t in transactions if set(itemset).issubset(set(t))) / len(transactions)
    except Exception as e:
        print(f"Error calculating support \n{e}")
```

- Function **prepare_transaction_dataframe**
  This function takes your list of customer transactions and turns it into a format that's easy for algorithms like Apriori and FP-Growth to understand.

```python
def prepare_transaction_dataframe(transactions):
    """
    Convert a list of transactions into a one-hot encoded pandas DataFrame.

    Each column represents an item, and each row represents a transaction.
    True indicates the item is present in the transaction; False otherwise.

    Args:
        transactions (list of lists): List of transactions, where each transaction is a list of items.

    Returns:
        pandas.DataFrame: One-hot encoded DataFrame of shape (num_transactions, num_items).
    """
    try:
        items = sorted(set(i for t in transactions for i in t))
        df = pd.DataFrame([[i in t for i in items] for t in transactions], columns=items).astype(bool)
        return df
    except Exception as e:
        print(f"Error loading dataframe \n{e}")
```

- Function **load_transaction_data**

  This function is responsible for reading your raw transaction and item information from CSV files. It does two main things:

1. **Reads Transactions:** It opens the transaction CSV and goes through each row. It expects items within a row to be either in separate columns or separated by commas. It collects all the items from each valid row to form a list of transactions.
2. **Creates Item Map (if available):** If you provide an itemset CSV with 'Item #' and 'Item Name' columns, this function creates a dictionary that links the item's ID (number) to its actual name.

```
def load_transaction_data(transaction_file, itemset_file):
    """
    Load transaction and itemset data from CSV files.

    Reads a transaction CSV file and an itemset CSV file. Processes the transactions
    into a list of lists, where each inner list represents a single transaction. If the
    itemset file contains 'Item #' and 'Item Name' columns, a mapping dictionary is created.

    Args:
        transaction_file (str): Path to the CSV file containing transaction data.
        itemset_file (str): Path to the CSV file containing item ID and item name mapping.

    Returns:
        tuple:
            - transactions (list of lists): Each inner list represents a transaction with item names as strings.
            - item_map (dict or None): Mapping from item IDs to item names if available, otherwise None.

    Notes:
        - Items in transaction rows can be comma-separated strings or numeric values.
        - Empty or invalid transactions are ignored.
    """
    try:
        df_trans = pd.read_csv(transaction_file)
        df_items = pd.read_csv(itemset_file)

        if 'Item #' in df_items.columns and 'Item Name' in df_items.columns:
            item_map = dict(zip(df_items['Item #'], df_items['Item Name']))
        else:
            item_map = None

        transactions = []
        for _, row in df_trans.iterrows():
            transaction = []
            for item in row:
                if isinstance(item, str):
                    items = [i.strip() for i in item.split(',') if i.strip()]
                    transaction.extend(items)
                elif pd.notna(item):
                    transaction.append(str(item))
            if transaction:
                transactions.append(transaction)
        return transactions, item_map

    except Exception as e:
        print(f"Error reading CSVs \n{e}")
        return None, None
```

- Function **brute_force_algorithm**
  This function implements the Brute Force approach to find frequent itemsets and association rules. It works by:

1. **Generating Candidates:** It creates every possible combination of items from the transactions, starting with single items and going up to combinations of all items.
2. **Calculating Support:** For each combination (itemset), it checks how many transactions contain all the items in that set and calculates the support (the percentage of transactions containing the itemset).
3. **Filtering Frequent Itemsets:** It keeps only those itemsets whose support is equal to or greater than the minimum support threshold you specified.
4. **Generating Association Rules:** From the frequent itemsets, it creates possible association rules (e.g., "if a customer buys A, they also buy B").
5. **Calculating Confidence:** For each rule, it calculates the confidence (how often itemset B appears in transactions that already contain itemset A).
6. **Filtering Rules:** It keeps only those rules whose confidence is equal to or greater than the minimum confidence threshold you specified.

```python
def brute_force_algorithm(transactions, min_support, min_confidence):
    """
    Run the Brute Force algorithm to find frequent itemsets and generate association rules.

    This function exhaustively generates all possible item combinations from the transactions,
    calculates their support, and retains those meeting the minimum support threshold.
    It then generates association rules from frequent itemsets, keeping only those with
    confidence above the specified threshold.

    Args:
        transactions (list of lists): A list where each inner list represents a transaction containing items as strings.
        min_support (float): Minimum support threshold (0 <= min_support <= 1) for an itemset to be considered frequent.
        min_confidence (float): Minimum confidence threshold (0 <= min_confidence <= 1) for association rules.

    Returns:
        tuple:
            - frequent_itemsets (list of tuples): Each tuple contains (itemset, support) for itemsets meeting the min_suppo
            - rules (list of tuples): Each tuple contains (antecedent, consequent, confidence, support) for rules meeting m
            - runtime (float): Execution time in seconds for the algorithm.

    Notes:
        - This is a computationally expensive method for large datasets, as it checks all possible item combinations.
        - If no frequent itemsets are found, returns (None, None, runtime).
        - The function prints results in a tabular format using `print_results`.
    """

    print("\nRunning Brute Force Algorithm...")
    start = time.time()
    try:
        items = sorted(set(i for t in transactions for i in t))
        frequent_itemsets = []
        size = 1

        while True:
            candidates = list(itertools.combinations(items, size))
            current = []
            for itemset in candidates:
                support = calculate_support(itemset, transactions)
                if support >= min_support:
                    current.append((itemset, support))
            if not current:
                break
            frequent_itemsets.extend(current)
            size += 1
```

```
    # Generate rules
    rules = []
    for itemset, support in frequent_itemsets:
        if len(itemset) > 1:
            for i in range(1, len(itemset)):
                for antecedent in itertools.combinations(itemset, i):
                    consequent = tuple(x for x in itemset if x not in antecedent)
                    antecedent_support = calculate_support(antecedent, transactions)
                    confidence = support / antecedent_support
                    if confidence >= min_confidence:
                        rules.append((antecedent, consequent, confidence, support))
except Exception as e:
    print(f"Error running Brute Force \n{e}")

finally:
    runtime = time.time() - start

if not frequent_itemsets:
    print("No frequent itemsets found with Brute Force.")
    return None, None, runtime

print(f"Brute Force completed in {runtime:.4f} seconds.\n")
print_results("Brute Force", frequent_itemsets, rules)
return frequent_itemsets, rules, runtime
```

- Function **apriori_algorithm**
  This function implements the Apriori algorithm using the mlxtend library. It's a more efficient method than Brute Force because it uses a key property: if an itemset is frequent, all of its subsets must also be frequent. This helps prune the search space significantly.

```python
def apriori_algorithm(transaction_df, min_support, min_confidence):
    """
    Run the Apriori algorithm using mlxtend to find frequent itemsets and generate association rules.

    This function uses the Apriori algorithm on a one-hot encoded DataFrame of transactions to
    identify frequent itemsets that meet the minimum support threshold. It then generates
    association rules from these itemsets that meet the minimum confidence threshold.

    Args:
        transaction_df (pd.DataFrame): One-hot encoded DataFrame where each column is an item and
                                       each row represents a transaction (True if item present).
        min_support (float): Minimum support threshold (0 <= min_support <= 1) for an itemset to be considered frequent.
        min_confidence (float): Minimum confidence threshold (0 <= min_confidence <= 1) for association rules.

    Returns:
        tuple:
            - frequent_itemsets (pd.DataFrame or None): DataFrame of frequent itemsets with support values.
                                                        None if no frequent itemsets are found.
            - rules (pd.DataFrame or None): DataFrame of association rules with confidence and support.
                                            None if no rules meet the threshold.
            - runtime (float): Execution time in seconds for the algorithm.

    Notes:
        - Uses mlxtend.apriori for frequent itemset mining and mlxtend.association_rules for rule generation.
        - If no frequent itemsets are found, returns (None, None, runtime).
        - The function prints results in a tabular format using `print_results`.
    """

    print("\nRunning Apriori Algorithm...")
    frequent_itemsets = None
    start = time.time()

    try:
        frequent_itemsets = apriori(transaction_df, min_support=min_support, use_colnames=True)

    except Exception as e:
        print(f"Error running Arpiori \n{e}")

    finally:
        runtime = time.time() - start

    if frequent_itemsets.empty:
        print("No frequent itemsets found with Apriori.")
        return None, None, runtime

    rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=min_confidence)
    print(f"Apriori completed in {runtime:.4f} seconds.\n")
    print_results("Apriori", frequent_itemsets, rules)
    return frequent_itemsets, rules, runtime
```

- Function **fpgrowth_algorithm**
  This function applies the FP-Growth algorithm on a one-hot encoded DataFrame of transactions to identify frequent itemsets that meet the minimum support threshold. It then generates association rules from these itemsets that meet the minimum confidence threshold.

```
def fpgrowth_algorithm(transaction_df, min_support, min_confidence):
    """
    Run the FP-Growth algorithm using mlxtend to find frequent itemsets and generate association rules.

    This function applies the FP-Growth algorithm on a one-hot encoded DataFrame of transactions
    to identify frequent itemsets that meet the minimum support threshold. It then generates
    association rules from these itemsets that meet the minimum confidence threshold.

    Args:
        transaction_df (pd.DataFrame): One-hot encoded DataFrame where each column is an item and
                                       each row represents a transaction (True if item present).
        min_support (float): Minimum support threshold (0 <= min_support <= 1) for an itemset to be considered frequent.
        min_confidence (float): Minimum confidence threshold (0 <= min_confidence <= 1) for association rules.

    Returns:
        tuple:
            - frequent_itemsets (pd.DataFrame or None): DataFrame of frequent itemsets with support values.
                                                        None if no frequent itemsets are found.
            - rules (pd.DataFrame or None): DataFrame of association rules with confidence and support.
                                            None if no rules meet the threshold.
            - runtime (float): Execution time in seconds for the algorithm.

    Notes:
        - Uses mlxtend.fpgrowth for frequent itemset mining and mlxtend.association_rules for rule generation.
        - If no frequent itemsets are found, returns (None, None, runtime).
        - The function prints results in a tabular format using `print_results`.
    """

    print("\nRunning FP-Growth Algorithm...")
    frequent_itemsets = None
    start = time.time()

    try:

        frequent_itemsets = fpgrowth(transaction_df, min_support=min_support, use_colnames=True)

    except Exception as e:
        print(f"Error running FP-Tree \n{e}")

    finally:
        runtime = time.time() - start

    if frequent_itemsets.empty:
        print("No frequent itemsets found with FP-Growth.")
        return None, None, runtime

    rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=min_confidence)
    print(f"FP-Growth completed in {runtime:.4f} seconds.\n")
    print_results("FP-Growth", frequent_itemsets, rules)
    return frequent_itemsets, rules, runtime
```

- Function **print_results**
  This function is designed to display the output of the frequent itemset mining algorithms in a clear, easy-to-read table format. It takes the results (frequent itemsets and association rules) from any of the algorithms and uses the tabulate library to create well-formatted tables.

```
def print_results(algorithm_name, frequent_itemsets, rules):
    """
    Print frequent itemsets and association rules in a tabular format.

    This function takes the results from a frequent itemset mining algorithm (either as
    a pandas DataFrame or a list of tuples) and prints them neatly using `tabulate`.
    Association rules are displayed in the format "A -> B" with confidence and support.

    Args:
        algorithm_name (str): Name of the algorithm (used in the table header).
        frequent_itemsets (pd.DataFrame or list of tuples or None): Frequent itemsets with support values.
        rules (pd.DataFrame or list of tuples or None): Association rules with confidence and support.

    Returns:
        None: This function only prints the results; it does not return any value.

    Notes:
        - If no frequent itemsets or rules are found, appropriate messages are displayed.
        - Supports both DataFrame input (from mlxtend) and list-of-tuples input (from custom algorithms).
        - Association rules are displayed in the format "antecedent -> consequent".
    """

    print(f"\n=== {algorithm_name} Results ===")

    try:

        # --- Frequent Itemsets ---
        print("\nFrequent Itemsets:")
        if isinstance(frequent_itemsets, pd.DataFrame):
            itemset_data = [
                [", ".join(list(row["itemsets"])), f"{row['support']*100:.2f}%"]
                for _, row in frequent_itemsets.iterrows()
            ]
        elif frequent_itemsets:
            itemset_data = [
                [", ".join(itemset), f"{support*100:.2f}%"]
                for itemset, support in frequent_itemsets
            ]
        else:
            itemset_data = [["No frequent itemsets found", ""]]

        print(tabulate(itemset_data, headers=["Itemset", "Support"], tablefmt="fancy_grid"))

        # --- Association Rules ---
        print("\nAssociation Rules:")
        if isinstance(rules, pd.DataFrame) and not rules.empty:
            rule_data = [
                [f"{', '.join(list(rule['antecedents']))} -> {', '.join(list(rule['consequents']))}",
                f"{rule['confidence']*100:.2f}%",
                f"{rule['support']*100:.2f}%"]
                for _, rule in rules.iterrows()
            ]
        elif rules is not None and len(rules) > 0:
            rule_data = [
                [f"{', '.join(antecedent)} -> {', '.join(consequent)}",
                f"{confidence*100:.2f}%",
                f"{support*100:.2f}%"]
                for antecedent, consequent, confidence, support in rules
            ]
        else:
            rule_data = [["No association rules found", "", ""]]

        print(tabulate(rule_data, headers=["Rule", "Confidence", "Support"], tablefmt="fancy_grid"))

    except Exception as e:
        print(f"Error printing results \n{e}")
```

- Function **main**
  This function serves as the primary control center for running the frequent itemset mining analysis. It orchestrates the entire process by:

1. Presenting Store Options: It displays a list of available stores with corresponding transaction and itemset files.
2. Getting User Input: It prompts the user to select a store and enter the desired minimum support and confidence thresholds (as percentages).
3. Loading Data: It calls load_transaction_data to load the selected transaction and itemset files.
4. Preparing Data for Algorithms: It uses prepare_transaction_dataframe to convert the raw transactions into a format suitable for the Apriori and FP-Growth algorithms.
5. Running Algorithms: It executes the brute_force_algorithm, apriori_algorithm, and fpgrowth_algorithm with the loaded data and user-defined thresholds.
6. Displaying Results: For each algorithm, it uses print_results to display the found frequent itemsets and association rules in a clear, tabular format.
7. Comparing Performance: It tracks and displays the execution time for each algorithm.
8. Identifying Fastest Algorithm: Finally, it determines and announces which algorithm completed the task the fastest.

```python
def main():
    """
    Main entry point to compare Brute Force, Apriori, and FP-Growth algorithms on transaction data.

    This function allows the user to select a store, load its transaction data, set minimum
    support and confidence thresholds, and run three frequent itemset mining algorithms.
    It prints the frequent itemsets, association rules, execution times, and identifies
    the fastest algorithm.

    Args:
        None: All inputs are collected via user prompts.

    Returns:
        None: This function only prints results; it does not return any value.

    Notes:
        - Uses `load_transaction_data` to read transactions and item mappings from CSV files.
        - Converts transactions into a one-hot encoded DataFrame for mlxtend algorithms.
        - Runs `brute_force_algorithm`, `apriori_algorithm`, and `fpgrowth_algorithm`.
        - Prints results for each algorithm using `print_results`.
        - Displays execution times and highlights the fastest algorithm.
    """

    stores = {
        "1": ("Amazon", "Amazon_Transaction.csv", "Amazon_Itemset.csv"),
        "2": ("Bestbuy", "Bestbuy_Transaction.csv", "Bestbuy_Itemset.csv"),
        "3": ("Shoprite", "Shoprite_Transaction.csv", "Shoprite_Itemset.csv"),
        "4": ("K-mart", "K_mart_Transaction.csv", "K_mart_Itemset.csv"),
        "5": ("Nike", "Nike_Transaction.csv", "Nike_Itemset.csv")
    }

    print("\nWelcome to ruleFinder!")
    print("\nAvailable stores:")
    for key, (store_name, _, _) in stores.items():
        print(f"{key}. {store_name}")

    # --- Store selection ---
    while True:
        choice = input("Enter the number of the store you want to analyze: ").strip()
        if choice in stores:
            store_name, transaction_file, itemset_file = stores[choice]
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 5.")
```

```
# --- Load data ---
transaction_file = "../data/" + transaction_file
itemset_file = "../data/" + itemset_file
transactions, item_map = load_transaction_data(transaction_file, itemset_file)
if not transactions:
    print("No valid transactions found. Please check your CSV files.")
    return

# --- User input for thresholds ---
while True:
    try:
        min_support = float(input("Enter the minimum support (as % between 1 and 100): "))
        if 1 <= min_support <= 100:
            min_support /= 100
            break
        print("Please enter a value between 1 and 100.")
    except ValueError:
        print("Invalid input. Enter a numeric value.")

while True:
    try:
        min_confidence = float(input("Enter the minimum confidence (as % between 1 and 100): "))
        if 1 <= min_confidence <= 100:
            min_confidence /= 100
            break
        print("Please enter a value between 1 and 100.")
    except ValueError:
        print("Invalid input. Enter a numeric value.")

# --- Prepare DataFrame for mLxtend algorithms ---
transaction_df = prepare_transaction_dataframe(transactions)

print(f"\nAnalyzing store: {store_name}\n{'-'*60}")

# --- Run algorithms ---
bf_itemsets, bf_rules, bf_time = brute_force_algorithm(transactions, min_support, min_confidence)
apr_itemsets, apr_rules, apr_time = apriori_algorithm(transaction_df, min_support, min_confidence)
fp_itemsets, fp_rules, fp_time = fpgrowth_algorithm(transaction_df, min_support, min_confidence)


results_table = [
    ["Brute Force", f"{bf_time:.4f}"],
    ["Apriori", f"{apr_time:.4f}"],
    ["FP-Growth", f"{fp_time:.4f}"]
]

print("\nExecution Times (seconds):")
print(tabulate(results_table, headers=["Algorithm", "Time (s)"], tablefmt="grid"))
```

```
    # --- Determine fastest algorithm ---
    times = {
        "Brute Force": bf_time,
        "Apriori": apr_time,
        "FP-Growth": fp_time
    }
    fastest = min(times, key=times.get)
    print(f"\n The fastest algorithm is: {fastest} ({times[fastest]:.4f} seconds)\n")


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

# Github Link

https://github.com/mk2436/kadam_mihir_midtermproject.git

# Conclusion

This project successfully demonstrates and compares the Brute-Force, Apriori, and FP-Growth algorithms for frequent itemset mining and association rule generation. The Brute-Force approach, while accurate, was found to be computationally intensive and unsuitable for larger datasets. Apriori offered a substantial improvement in efficiency through candidate pruning, whereas FP-Growth delivered the best performance overall due to its tree-based structure.

All three methods produced identical frequent itemsets and rules, confirming their correctness. However, in terms of execution time and scalability, FP-Growth is the preferred algorithm for practical applications involving medium to large datasets