NAME:	Rajat Pednekar
NJIT UCID:	rp2348@njit.edu
EMAIL ADDRESS:	rajatpednekar07@gmail.com
PROFESSOR:	Yasser Abduallah
LECTURE:	CS 634 Data Mining

Data Mining: Midterm Project Report

Topic: A Comparative Study of Brute-Force, Apriori, and FP-Growth Algorithms.

➢ Abstract

This project compares three algorithms for identifying frequent itemsets and generating association rules: brute-force, Apriori, and FP-Growth. Five transactional datasets containing supermarket items were created and analyzed using these methods. The brute-force approach manually generates all possible itemsets, while Apriori and FP-Growth utilize more efficient techniques available in Python libraries. The algorithms are evaluated based on accuracy and execution time, with varying support and confidence thresholds. Results demonstrate the benefits of optimized algorithms in frequent itemset mining

➢ Introduction

This project details the implementation of three algorithms—Brute-Force, Apriori, and FP-Growth—for mining frequent itemsets and generating association rules from transactional data. The purpose is to compare the efficiency and effectiveness of each algorithm in generating frequent itemsets and rules while measuring their execution time under various support and confidence thresholds.

➢ Project Overview

The project is divided into three main parts:

1. Data preparation
2. Brute Force method implementation
3. Algorithm comparison (Brute Force, Apriori, and FP-Growth)

1. Data Preparation

Five different datasets representing transactions from various stores (E.g., Amazon, Nike, Best Buy, Kmart, General) are used for analysis. Each dataset contains a set of transactions with items typically found in supermarkets. The transactional data is stored in CSV files and is manually curated to ensure a deterministic nature.

2. Brute Force Method Implementation

The brute-force method enumerates all possible itemsets and calculates their support. For each dataset:

- **Itemsets of increasing size** are generated (1-itemsets, 2-itemsets, etc.).
- The **support** of each itemset is calculated.
- Itemsets that meet the user-specified support threshold are deemed frequent.

This method is exhaustive but can become computationally expensive as the size of the dataset and itemsets increase.

3. Algorithm Comparison

- **The Apriori algorithm**, implemented using the mlxtend library, follows a more optimized approach by pruning the search space:
  - It uses frequent (k-1)-itemsets to generate k-itemsets.
  - Only itemsets meeting the support threshold are expanded, reducing the number of candidates sets.
- **The FP-Growth algorithm**, also implemented via mlxtend, uses a frequent pattern tree (FP-tree) to compactly represent the dataset and avoid repeated scanning:
  - It builds an FP-tree to store item frequencies.
  - The tree is mined recursively to generate frequent itemsets.
- Compared results from Brute Force, Apriori, and FP-Growth.
- Analyzed timing performance for all three algorithms.
- Generated association rules for each of the 5 transactional databases.

➢ Core Concepts and Principles:

1. **Frequent Itemset discovery**: All the Algorithm revolves around discovering frequent itemsets, i.e., sets of items that frequently co-occur in transactions. These itemsets provide insights into customer purchase behavior and preferences.
2. **Support and Confidence:** Two key metrics in data mining are support and confidence. Support measures how frequently an item or itemset occurs, while confidence assesses the likelihood of items being purchased together. These metrics guide our analysis.
3. **Association Rules:** By determining strong association rules, I identify which items are commonly purchased together. These rules are instrumental for optimizing sales strategies, such as recommendations.

➢ Project Workflow

The project follows a structured workflow to ensure the implementation and comparison of the three algorithms—Brute-Force, Apriori, and FP-Growth—on multiple datasets. The steps are as follows:

1. **Dataset Preparation:**
   a. Five transactional datasets are created, each containing at least 20 transactions.
   b. The datasets are stored in CSV format, and each transaction contains a combination of items typically found in supermarkets.
2. **Data Loading and Preprocessing:**
   a. The user selects a dataset, which is read into the program.
   b. Transactions are cleaned and prepared for analysis by ensuring that items are correctly categorized, and missing or malformed data is handled.
3. **User Input:**
   a. The user is prompted to input a minimum support threshold (as a percentage) and a minimum confidence threshold for generating association rules.
   b. The user specifies the thresholds once, and the values are reused across all three algorithms.

4. **Algorithm Execution:**
   a. Brute-Force Algorithm: The brute-force method is executed first. It generates all possible itemsets of increasing size and checks their support against the user-specified threshold.

b. Apriori Algorithm: Next, the Apriori algorithm from the mlxtend library is applied to prune infrequent itemsets and generate frequent itemsets.

c. FP-Growth Algorithm: Finally, the FP-Growth algorithm from the mlxtend library is executed to build an FP-tree and mine frequent itemsets efficiently.

5. **Association Rule Generation:**
   a. After generating frequent itemsets, each algorithm produces association rules based on the user-defined confidence threshold.
   b. The rules are printed along with their support, confidence, and lift values.
6. **Performance Measurement:**
   a. The execution time for each algorithm is recorded and compared.
   b. The results are displayed, showing which algorithm performed the fastest on each dataset.
7. **Result Output:**
   a. The association rules and frequent itemsets are presented for each dataset, along with the execution times.
   b. The user is informed of the best-performing algorithm in terms of speed.

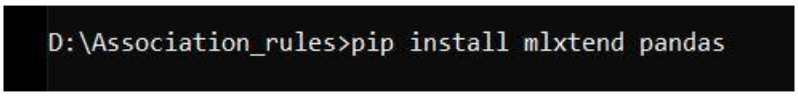
➢ Tutorial to run the .py file in your device.

✓ Prerequisites

1. Make sure the Datasets (CSV files) are in same directory.
2. The Naming convention should be same as it is in the zip file/ folder provided in the github.
3. The General Store dataset is small, making it suitable for testing with low support values. For larger datasets, the Brute Force algorithm may take significantly longer to execute, though it will eventually complete.
4. Make sure the python environment is set up beforehand.

Step 1: Set up the Environments.

1. Ensure you have installed Python on your system.
2. Install the required Libraries by running:



```
D:\Association_rules>pip install mlxtend pandas
```

Step 2: Prepare the data files.

Make sure you have the following CSV files for each store in the same directory and with same naming convention as the script:

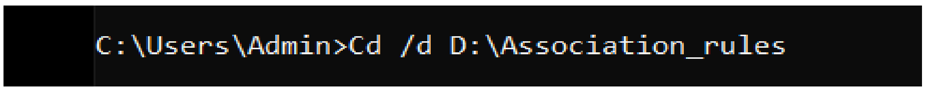
- Amazon_Transaction.csv and Amazon_Itemset.csv
- Bestbuy_Transaction.csv and Bestbuy_Itemset.csv
- General_Transaction.csv and General_Itemset.csv
- K_mart_Transaction.csv and K_mart_Itemset.csv
- Nike_Transaction.csv and Nike_Itemset.csv

Note: *The Dataset is the same as professor provided in the canvas.*

Step 3: Run the program.

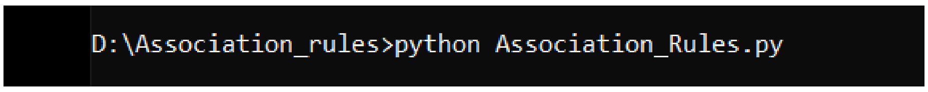
1. Check if the Python file is saved as, e.g., Association_Rules.py.
2. Open a terminal or command prompt.
3. Navigate to the directory containing the Python file and CSV files.

a. Type Command: Cd /d “Path of the file”



```
C:\Users\Admin>Cd /d D:\Association_rules
```

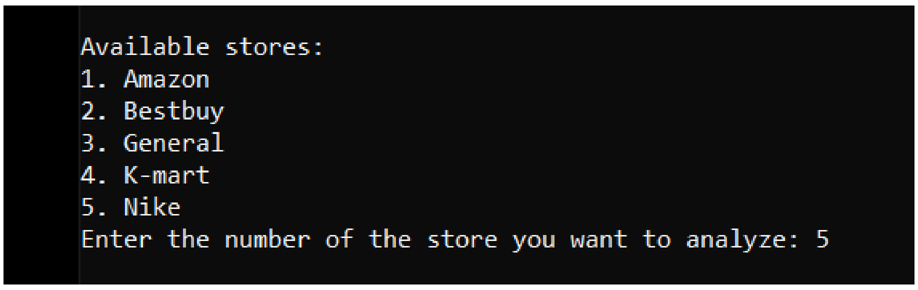
4. Run the script by entering:



```
D:\Association_rules>python Association_Rules.py
```

Step 4: Interact with the program.

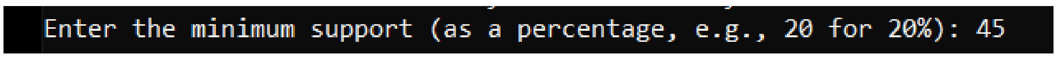
1. The program will display a list of available stores:
2. Enter the number corresponding to the store you want to analyse (1-5).



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

3. Next, you'll be prompted to enter the minimum support:

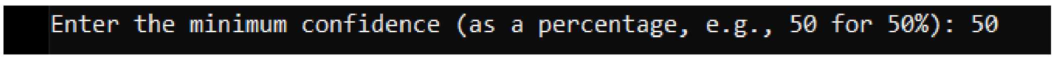
- Enter a number between 0 and 100 (Describing percentages)



```
Enter the minimum support (as a percentage, e.g., 20 for 20%): 45
```

4. Then, enter the minimum confidence:

- Enter a number between 0 and 100 (Describing percentages).



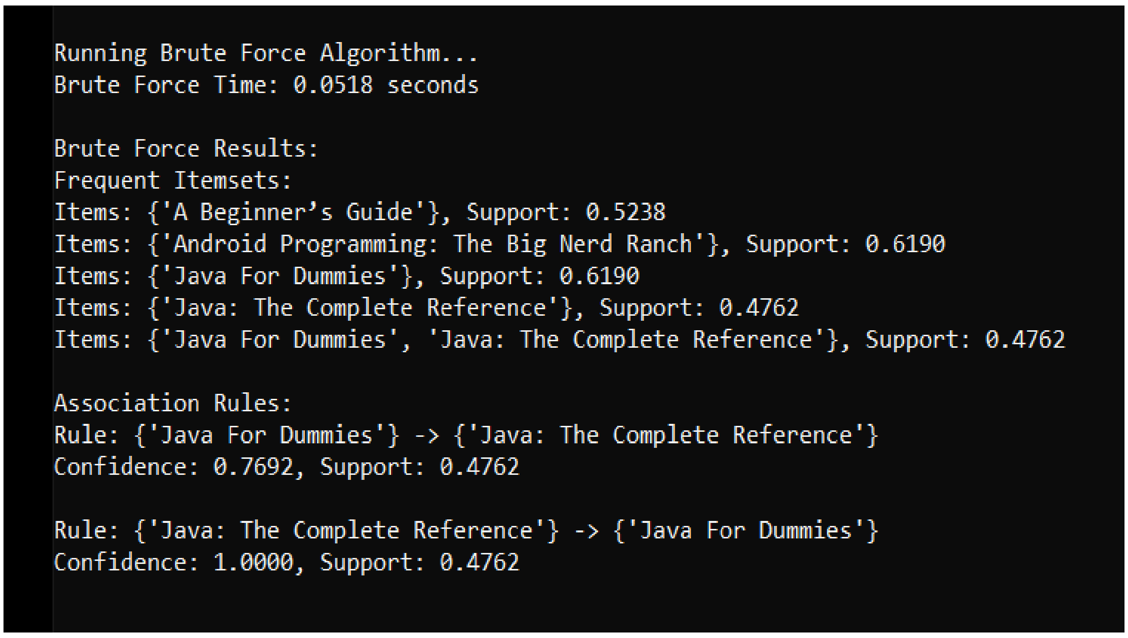
```
Enter the minimum confidence (as a percentage, e.g., 50 for 50%): 50
```

Tutorial Note: *The General Store dataset is small, making it suitable for testing with low support values. For larger datasets, the Brute Force algorithm may take significantly longer to execute, though it will eventually complete.*

Step 5: Review the results

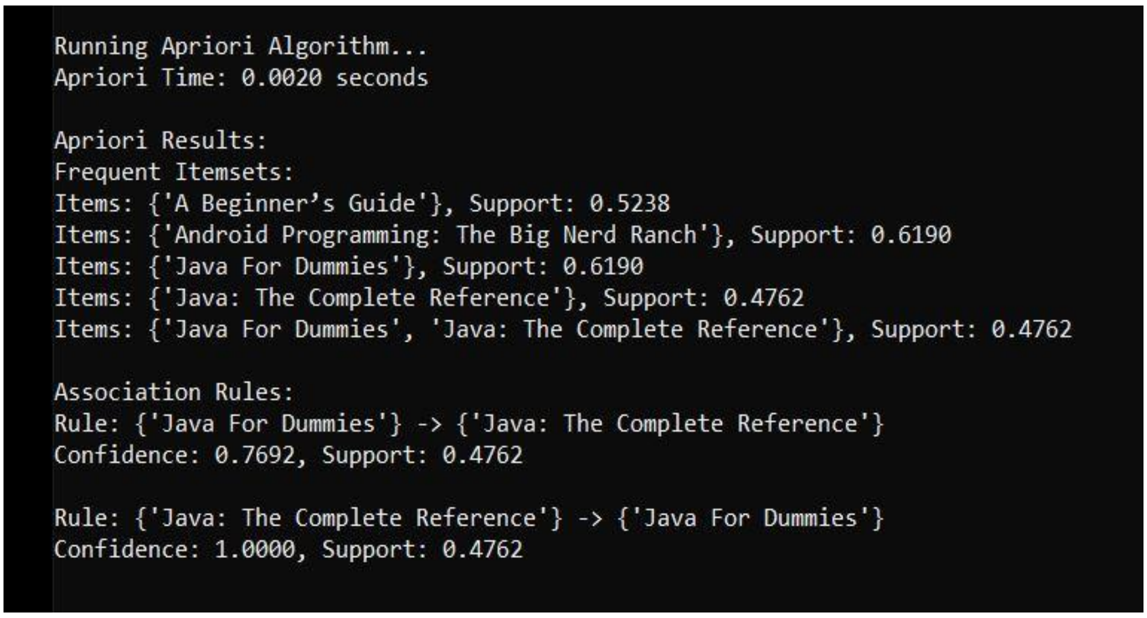
1. The program will run the Brute Force, Apriori, and FP-Growth algorithms on the selected store's data.
2. For each algorithm, you'll see:
   a. Execution time
   b. Frequent item sets found
   c. Association rules generated (if any)

- **Result for Brute Force.**



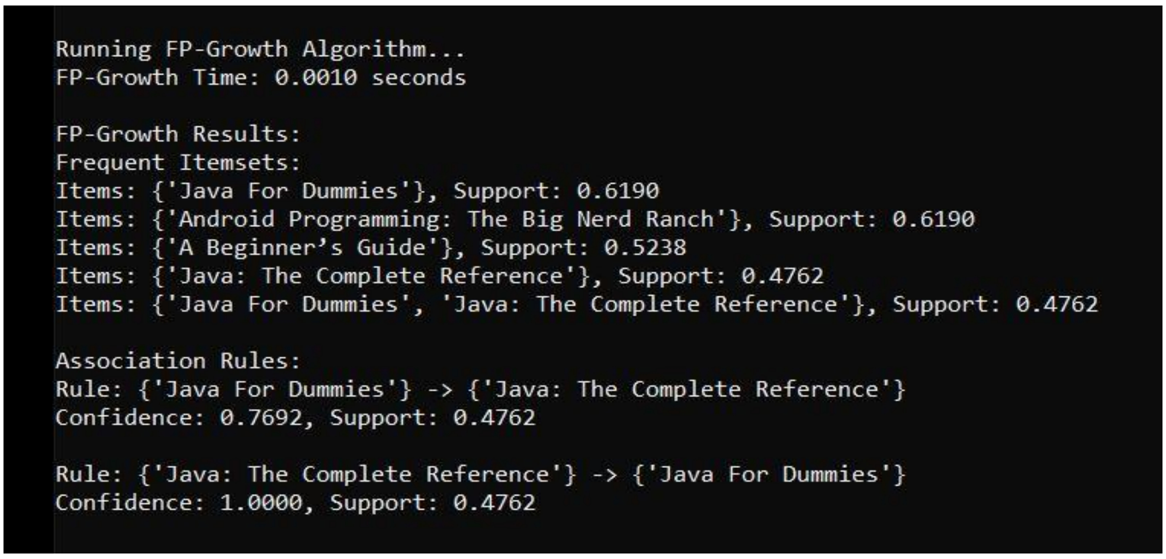
```
Running Brute Force Algorithm...
Brute Force Time: 0.0518 seconds

Brute Force Results:
Frequent Itemsets:
Items: {'A Beginner’s Guide'}, Support: 0.5238
Items: {'Android Programming: The Big Nerd Ranch'}, Support: 0.6190
Items: {'Java For Dummies'}, Support: 0.6190
Items: {'Java: The Complete Reference'}, Support: 0.4762
Items: {'Java For Dummies', 'Java: The Complete Reference'}, Support: 0.4762

Association Rules:
Rule: {'Java For Dummies'} -> {'Java: The Complete Reference'}
Confidence: 0.7692, Support: 0.4762

Rule: {'Java: The Complete Reference'} -> {'Java For Dummies'}
Confidence: 1.0000, Support: 0.4762
```

- **Result for Apriori Algorithm**



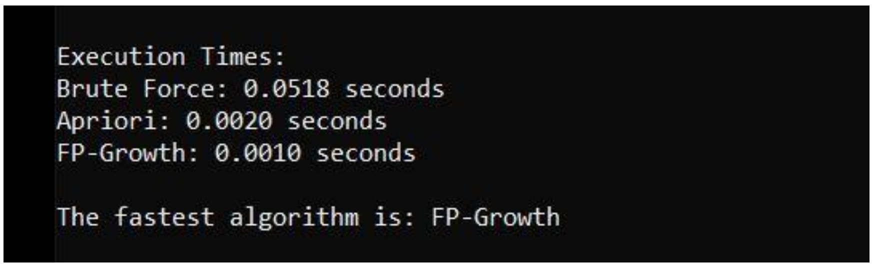
```
Running Apriori Algorithm...
Apriori Time: 0.0020 seconds

Apriori Results:
Frequent Itemsets:
Items: {'A Beginner’s Guide'}, Support: 0.5238
Items: {'Android Programming: The Big Nerd Ranch'}, Support: 0.6190
Items: {'Java For Dummies'}, Support: 0.6190
Items: {'Java: The Complete Reference'}, Support: 0.4762
Items: {'Java For Dummies', 'Java: The Complete Reference'}, Support: 0.4762

Association Rules:
Rule: {'Java For Dummies'} -> {'Java: The Complete Reference'}
Confidence: 0.7692, Support: 0.4762

Rule: {'Java: The Complete Reference'} -> {'Java For Dummies'}
Confidence: 1.0000, Support: 0.4762
```

- **Result for FP-Growth Algorithm**



```
Running FP-Growth Algorithm...
FP-Growth Time: 0.0010 seconds

FP-Growth Results:
Frequent Itemsets:
Items: {'Java For Dummies'}, Support: 0.6190
Items: {'Android Programming: The Big Nerd Ranch'}, Support: 0.6190
Items: {'A Beginner’s Guide'}, Support: 0.5238
Items: {'Java: The Complete Reference'}, Support: 0.4762
Items: {'Java For Dummies', 'Java: The Complete Reference'}, Support: 0.4762

Association Rules:
Rule: {'Java For Dummies'} -> {'Java: The Complete Reference'}
Confidence: 0.7692, Support: 0.4762

Rule: {'Java: The Complete Reference'} -> {'Java For Dummies'}
Confidence: 1.0000, Support: 0.4762
```

3. At the end, the program will display a summary of execution times and the fastest algorithm:



```
Execution Times:
Brute Force: 0.0518 seconds
Apriori: 0.0020 seconds
FP-Growth: 0.0010 seconds

The fastest algorithm is: FP-Growth
```

Results and Analysis

- **Brute-Force**: Accurate but inefficient for large datasets due to the exponential increase in itemset combinations.
- **Apriori**: Significant reduction in computation time by pruning infrequent itemsets early.
- **FP-Growth**: The most efficient algorithm for larger datasets due to its compressed tree structure.

In all cases, the results for frequent itemsets and association rules were consistent across the three algorithms, but the FP-Growth algorithm performed the fastest.

Conclusion

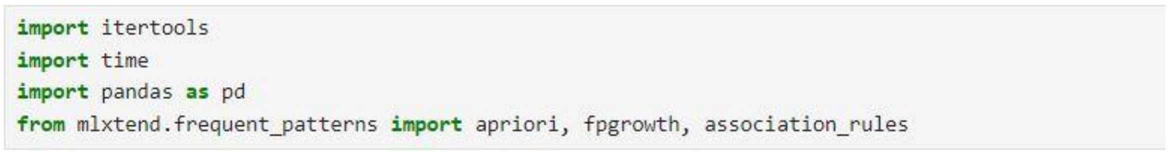
This project demonstrates the trade-offs between the brute-force, Apriori, and FP-Growth algorithms in frequent itemset mining. While brute-force guarantees accurate results, its computational cost is prohibitive for larger datasets. Apriori and FP-Growth provide more scalable alternatives, with FP-Growth offering the best performance overall.

Implemented Code Screenshots

1. Import Statements

This cell imports the necessary libraries:

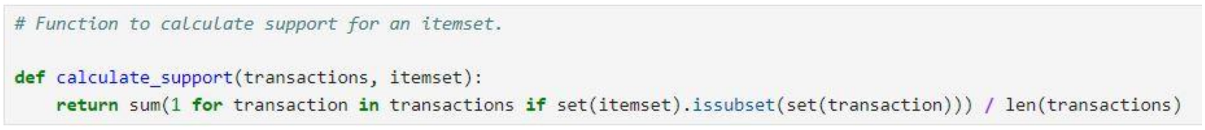
- itertools for generating combinations.
- time for measuring execution time.
- pandas for data manipulation and analysis.
- apriori, fpgrowth, and association_rules from mlxtend.frequent_patterns for implementing the Apriori and FP-Growth algorithms.



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

2. Calculate Support Function

This function calculates the support for a given itemset in a set of transactions. Support is the fraction of transactions that contain the itemset. It's a crucial metric in association rule mining.

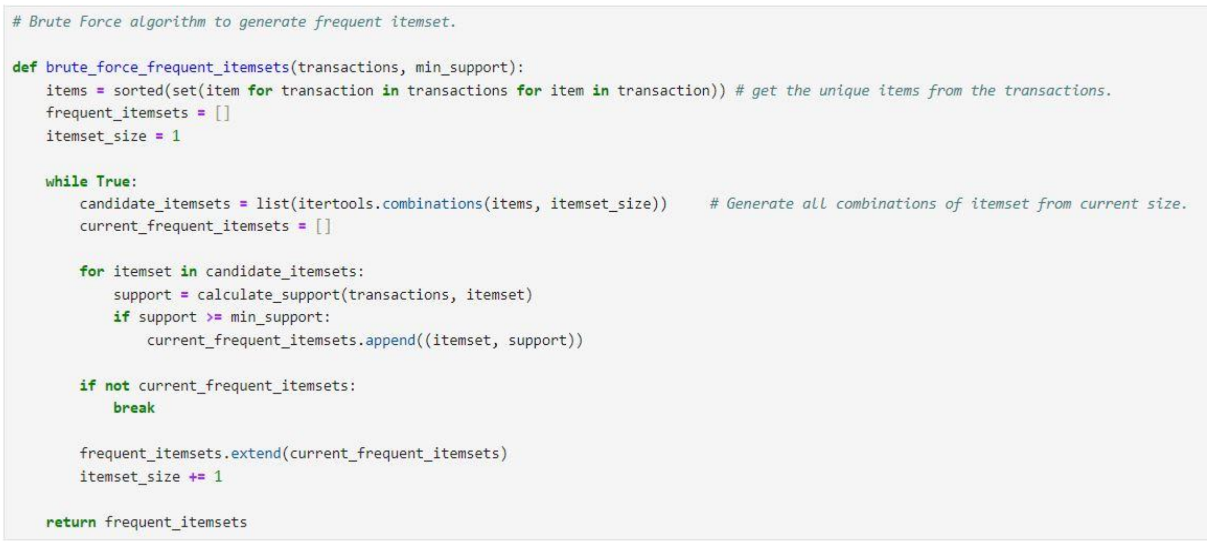


```python
# Function to calculate support for an itemset.

def calculate_support(transactions, itemset):
    return sum(1 for transaction in transactions if set(itemset).issubset(set(transaction))) / len(transactions)
```

3. **Brute Force Frequent Itemsets Function**

   This function implements a brute force approach to find the frequent itemsets.

- It starts with individual items and progressively increases the itemset size.
- For each size, it generates all possible combinations of items.
- It calculates the support for each combination and keeps those that meet the minimum support threshold.
- The process continues until no new frequent itemsets are found.

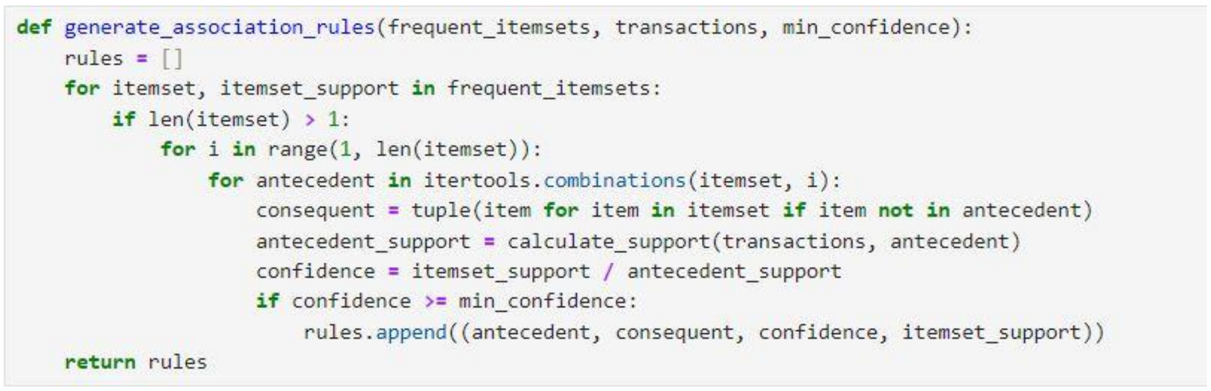


```python
# Brute Force algorithm to generate frequent itemset.

def brute_force_frequent_itemsets(transactions, min_support):
    items = sorted(set(item for transaction in transactions for item in transaction)) # get the unique items from the transactions.
    frequent_itemsets = []
    itemset_size = 1

    while True:
        candidate_itemsets = list(itertools.combinations(items, itemset_size))     # Generate all combinations of itemset from current size.
        current_frequent_itemsets = []

        for itemset in candidate_itemsets:
            support = calculate_support(transactions, itemset)
            if support >= min_support:
                current_frequent_itemsets.append((itemset, support))

        if not current_frequent_itemsets:
            break

        frequent_itemsets.extend(current_frequent_itemsets)
        itemset_size += 1

    return frequent_itemsets
```

4. **Generate Association Rules Function**

   This function generates association rules from the frequent itemsets:

- For each frequent itemset with more than one item, it generates all possible antecedent-consequent pairs.
- It calculates the confidence for each rule.
- Rules that meet the minimum confidence threshold are kept and returns a list of rules with their antecedents, consequents, confidence and support.

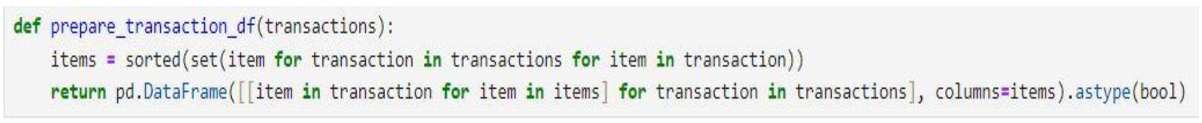


```python
def generate_association_rules(frequent_itemsets, transactions, min_confidence):
    rules = []
    for itemset, itemset_support in frequent_itemsets:
        if len(itemset) > 1:
            for i in range(1, len(itemset)):
                for antecedent in itertools.combinations(itemset, i):
                    consequent = tuple(item for item in itemset if item not in antecedent)
                    antecedent_support = calculate_support(transactions, antecedent)
                    confidence = itemset_support / antecedent_support
                    if confidence >= min_confidence:
                        rules.append((antecedent, consequent, confidence, itemset_support))
    return rules
```

5. **Prepare Transaction DataFrame Function**

   This function prepares the transaction data for use with the Apriori and FP-Growth algorithms from mlxtend:

- It creates a list of all unique items across all transactions.
- t then creates a boolean DataFrame where each row represents a transaction and each column represents an item.
- A True value indicates the presence of an item in a transaction, while False indicates its absence.
- This format is required for the mlxtend implementation of Apriori and FP-Growth algorithms.

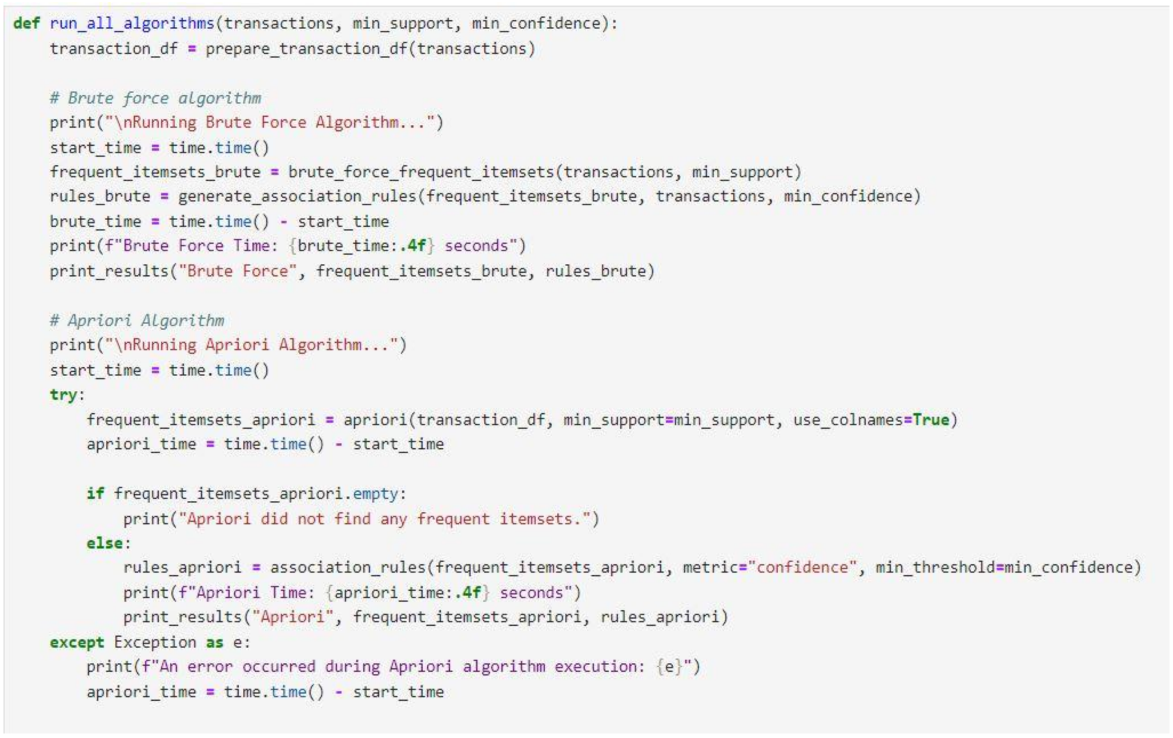


```python
def prepare_transaction_df(transactions):
    items = sorted(set(item for transaction in transactions for item in transaction))
    return pd.DataFrame([[item in transaction for item in items] for transaction in transactions], columns=items).astype(bool)
```

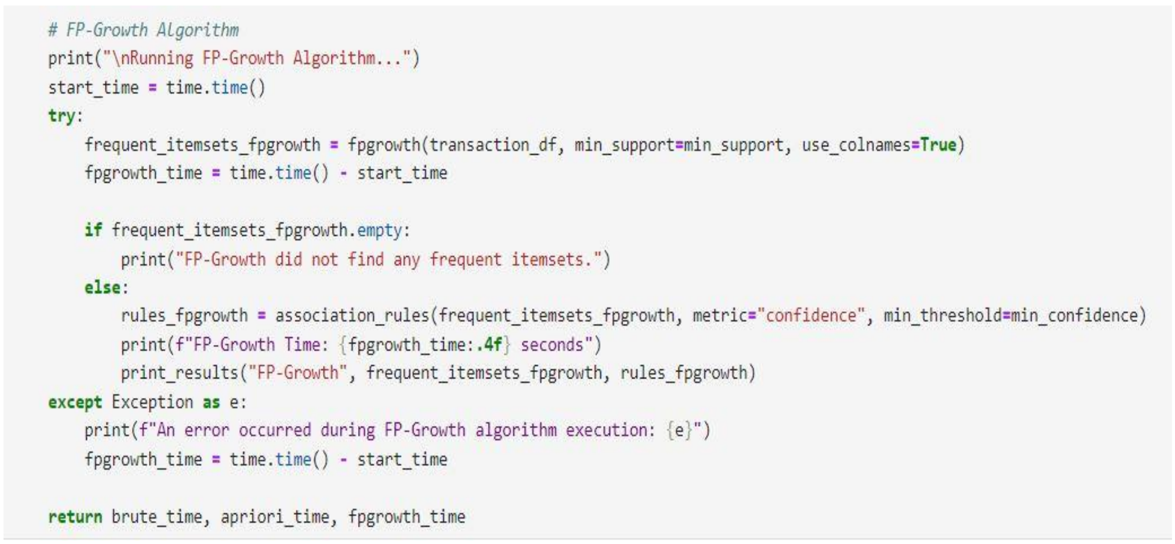
6. **Run All Algorithms Function**

   This function runs all three algorithms (Brute Force, Apriori, and FP-Growth) on the given transactions:

- It first prepares the transaction data in the required format.
- For each algorithm, it:
- Measures the execution time
- Finds frequent itemsets
- Generates association rules
- Prints the results
- It handles potential errors in the Apriori and FP-Growth algorithms.
- Finally, it returns the execution times for all three algorithms.



```python
def run_all_algorithms(transactions, min_support, min_confidence):
    transaction_df = prepare_transaction_df(transactions)

    # Brute force algorithm
    print("\nRunning Brute Force Algorithm...")
    start_time = time.time()
    frequent_itemsets_brute = brute_force_frequent_itemsets(transactions, min_support)
    rules_brute = generate_association_rules(frequent_itemsets_brute, transactions, min_confidence)
    brute_time = time.time() - start_time
    print(f"Brute Force Time: {brute_time:.4f} seconds")
    print_results("Brute Force", frequent_itemsets_brute, rules_brute)

    # Apriori Algorithm
    print("\nRunning Apriori Algorithm...")
    start_time = time.time()
    try:
        frequent_itemsets_apriori = apriori(transaction_df, min_support=min_support, use_colnames=True)
        apriori_time = time.time() - start_time

        if frequent_itemsets_apriori.empty:
            print("Apriori did not find any frequent itemsets.")
        else:
            rules_apriori = association_rules(frequent_itemsets_apriori, metric="confidence", min_threshold=min_confidence)
            print(f"Apriori Time: {apriori_time:.4f} seconds")
            print_results("Apriori", frequent_itemsets_apriori, rules_apriori)
    except Exception as e:
        print(f"An error occurred during Apriori algorithm execution: {e}")
        apriori_time = time.time() - start_time
```



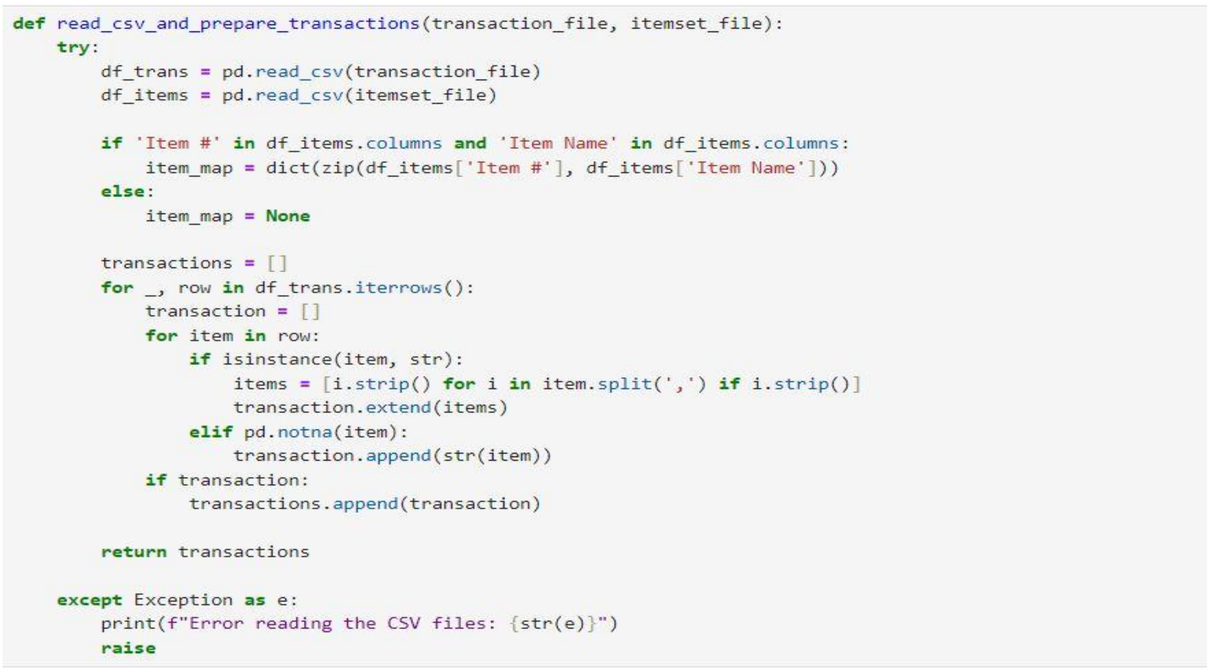
```python
    # FP-Growth Algorithm
    print("\nRunning FP-Growth Algorithm...")
    start_time = time.time()
    try:
        frequent_itemsets_fpgrowth = fpgrowth(transaction_df, min_support=min_support, use_colnames=True)
        fpgrowth_time = time.time() - start_time

        if frequent_itemsets_fpgrowth.empty:
            print("FP-Growth did not find any frequent itemsets.")
        else:
            rules_fpgrowth = association_rules(frequent_itemsets_fpgrowth, metric="confidence", min_threshold=min_confidence)
            print(f"FP-Growth Time: {fpgrowth_time:.4f} seconds")
            print_results("FP-Growth", frequent_itemsets_fpgrowth, rules_fpgrowth)
    except Exception as e:
        print(f"An error occurred during FP-Growth algorithm execution: {e}")
        fpgrowth_time = time.time() - start_time

    return brute_time, apriori_time, fpgrowth_time
```

7. **Read CSV and Prepare Transactions Function**

   This function reads transaction and itemset data from CSV files and prepares it for analysis:

- It reads both the transaction and itemset CSV files.
- It creates a mapping of item numbers to item names if available.
- It processes each transaction, handling both comma-separated strings and individual items.
- It returns a list of transactions, where each transaction is a list of items.

***Note:** Make sure the Datasets are kept in the same directory with the same naming conventions.*

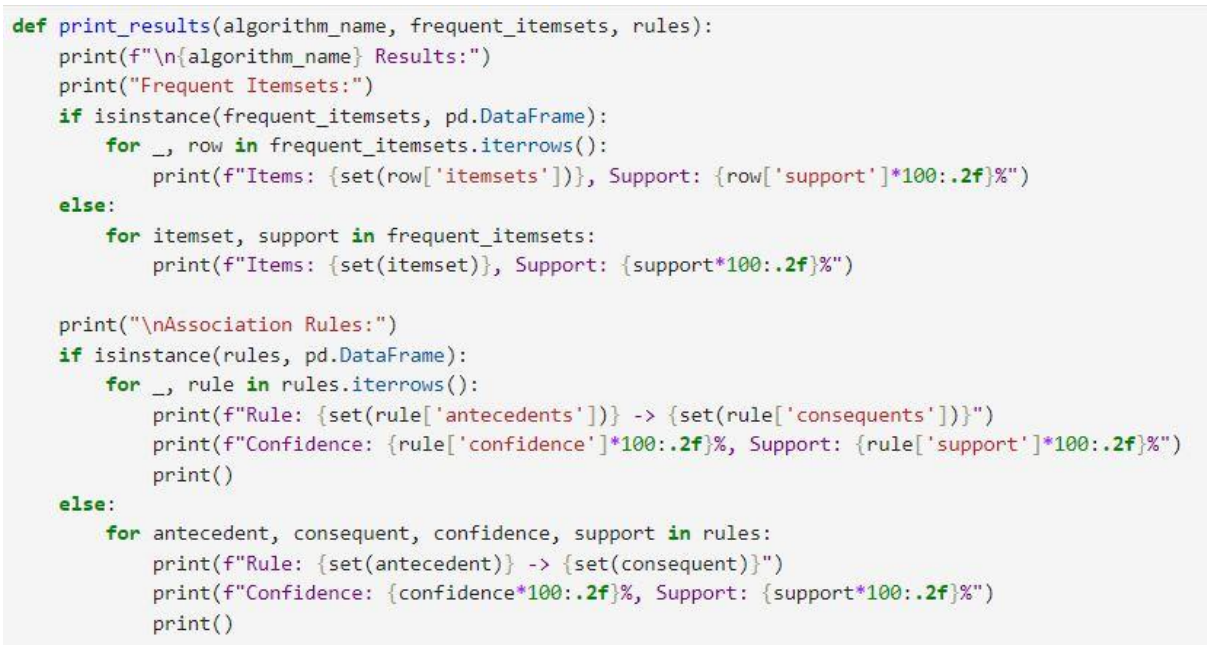


```python
def read_csv_and_prepare_transactions(transaction_file, itemset_file):
    try:
        df_trans = pd.read_csv(transaction_file)
        df_items = pd.read_csv(itemset_file)

        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

    except Exception as e:
        print(f"Error reading the CSV files: {str(e)}")
        raise
```

8. **Print Results Function**

   This function prints the results of the algorithms in a readable format:

- It first prints the frequent itemsets with their support values.
- Then it prints the association rules, showing the antecedent, consequent, confidence, and support for each rule.
- It handles both DataFrame and list formats, accommodating the different output formats of the algorithms.

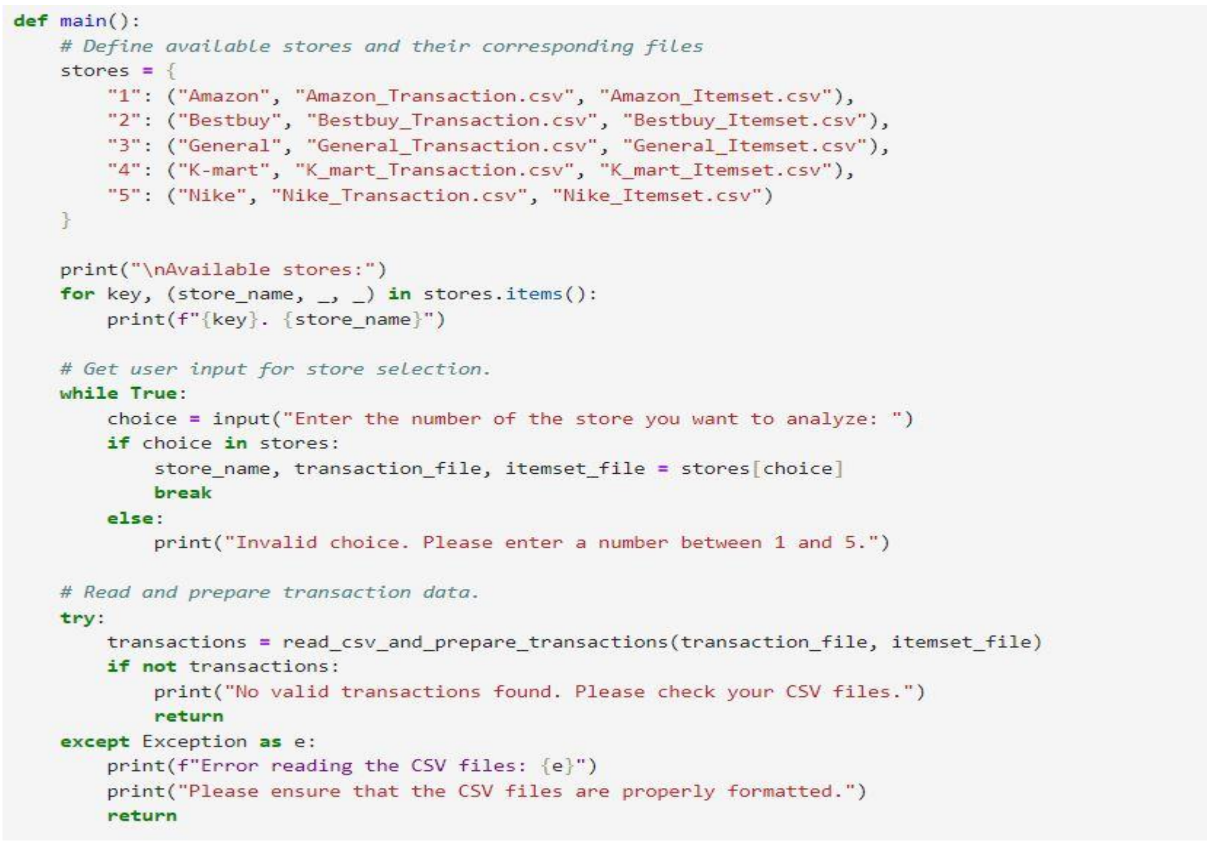


```python
def print_results(algorithm_name, frequent_itemsets, rules):
    print(f"\n{algorithm_name} Results:")
    print("Frequent Itemsets:")
    if isinstance(frequent_itemsets, pd.DataFrame):
        for _, row in frequent_itemsets.iterrows():
            print(f"Items: {set(row['itemsets'])}, Support: {row['support']*100:.2f}%")
    else:
        for itemset, support in frequent_itemsets:
            print(f"Items: {set(itemset)}, Support: {support*100:.2f}%")

    print("\nAssociation Rules:")
    if isinstance(rules, pd.DataFrame):
        for _, rule in rules.iterrows():
            print(f"Rule: {set(rule['antecedents'])} -> {set(rule['consequents'])}")
            print(f"Confidence: {rule['confidence']*100:.2f}%, Support: {rule['support']*100:.2f}%")
            print()
    else:
        for antecedent, consequent, confidence, support in rules:
            print(f"Rule: {set(antecedent)} -> {set(consequent)}")
            print(f"Confidence: {confidence*100:.2f}%, Support: {support*100:.2f}%")
            print()
```

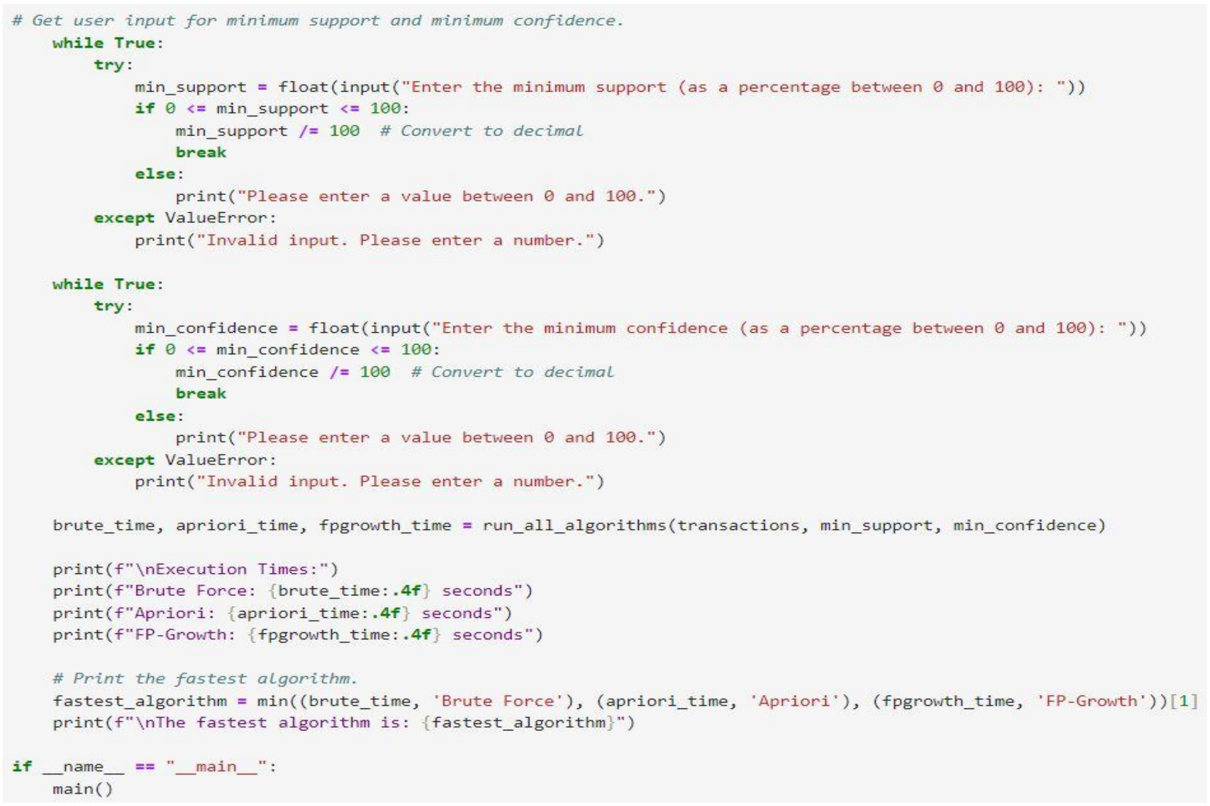
9. **Main Function**
   This is the main function that orchestrates the entire process:

- It defines a dictionary of available stores and their corresponding CSV files.
- It prompts the user to select a store for analysis.
- It reads and prepares the transaction data for the selected store.
- It prompts the user for minimum support and confidence thresholds.
- It runs all three algorithms on the data and measures their execution times.
- Finally, it prints the execution times and identifies the fastest algorithm.

Tutorial Note: The General Store is small dataset so if you want to try low support values, try on that dataset otherwise the brute force takes time to execute but it works.



```
def main():
    # Define available stores and their corresponding files
    stores = {
        "1": ("Amazon", "Amazon_Transaction.csv", "Amazon_Itemset.csv"),
        "2": ("Bestbuy", "Bestbuy_Transaction.csv", "Bestbuy_Itemset.csv"),
        "3": ("General", "General_Transaction.csv", "General_Itemset.csv"),
        "4": ("K-mart", "K_mart_Transaction.csv", "K_mart_Itemset.csv"),
        "5": ("Nike", "Nike_Transaction.csv", "Nike_Itemset.csv")
    }

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

    # Get user input for store selection.
    while True:
        choice = input("Enter the number of the store you want to analyze: ")
        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.")

    # Read and prepare transaction data.
    try:
        transactions = read_csv_and_prepare_transactions(transaction_file, itemset_file)
        if not transactions:
            print("No valid transactions found. Please check your CSV files.")
            return
    except Exception as e:
        print(f"Error reading the CSV files: {e}")
        print("Please ensure that the CSV files are properly formatted.")
        return
```



```
# Get user input for minimum support and minimum confidence.
    while True:
        try:
            min_support = float(input("Enter the minimum support (as a percentage between 0 and 100): "))
            if 0 <= min_support <= 100:
                min_support /= 100  # Convert to decimal
                break
            else:
                print("Please enter a value between 0 and 100.")
        except ValueError:
            print("Invalid input. Please enter a number.")

    while True:
        try:
            min_confidence = float(input("Enter the minimum confidence (as a percentage between 0 and 100): "))
            if 0 <= min_confidence <= 100:
                min_confidence /= 100  # Convert to decimal
                break
            else:
                print("Please enter a value between 0 and 100.")
        except ValueError:
            print("Invalid input. Please enter a number.")

    brute_time, apriori_time, fpgrowth_time = run_all_algorithms(transactions, min_support, min_confidence)

    print(f"\nExecution Times:")
    print(f"Brute Force: {brute_time:.4f} seconds")
    print(f"Apriori: {apriori_time:.4f} seconds")
    print(f"FP-Growth: {fpgrowth_time:.4f} seconds")

    # Print the fastest algorithm.
    fastest_algorithm = min((brute_time, 'Brute Force'), (apriori_time, 'Apriori'), (fpgrowth_time, 'FP-Growth'))[1]
    print(f"\nThe fastest algorithm is: {fastest_algorithm}")

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

- **Referral Links**

  https://github.com/Rajat-njit/Association_Rules_Data_Mining_Mid-Term.git