Midterm Project Report

Implementing Brute-Force Approach for Mining Association Rules and Comparing it with the Apriori Principle Approach from a Python Library

Name: *Benyamin Plaksienko*
NJIT UCID: *bp446*
Email Address: *bp446@njit.edu*
Professor: Yasser Abdullah
Course: CS 634 Data Mining

Abstract

In this project, I explore association rule mining and frequent itemset mining by implementing a brute-force approach from scratch and comparing it to an existing Apriori implementation. The brute-force method, developed using only Python's standard csv library, processes pre-generated transaction datasets to identify frequent itemsets and generate association rules. To benchmark its performance, I use the mlxtend library's Apriori algorithm for the same task. By comparing both approaches, I analyze their efficiency and validate whether my custom brute-force implementation produces correct results.

Introduction

Data mining is a powerful technique used to uncover hidden patterns and associations within large datasets. By identifying these patterns, businesses and organizations can make informed decisions and gain valuable insights. One of the core techniques in data mining is *association rule mining*, which aims to discover relationships between variables in a dataset. For example, in a retail context, association rule mining can help determine which products are frequently purchased together, leading to improved marketing strategies, better product placements, and personalized recommendations.

In this project, the focus is on the *Apriori algorithm*, a widely used method for association rule mining. The Apriori algorithm works by finding frequent itemsets — sets of items that appear together in transactions — and then generating association rules based on these itemsets. While the Apriori algorithm is well-established, there are alternative methods for performing association rule mining. One such method is the *Brute-Force approach*, which involves exhaustively checking all possible itemset combinations to determine frequent itemsets and generate association rules.

The main goal of this project was to **implement the Brute-Force approach for mining association rules** and compare its performance with the *Apriori principle approach* from a Python library. Specifically, the project aimed to explore:

- The differences in **execution time** between the Brute-Force approach and the Apriori algorithm under varying parameters (such as *support* and *confidence* thresholds).
- The **accuracy** of the results, comparing the frequent itemsets and association rules produced by both methods.
- The **efficiency** and **scalability** of the approaches when tested on different datasets and parameter values.

The project involved applying these methods to a custom dataset associated with retail stores, allowing for the generation of frequent itemsets and association rules. By comparing the two approaches, I was able to evaluate their strengths and limitations, focusing on their performance in real-world data mining tasks.

Project Workflow

1. Data Generation

In my code, I am generating random transactions for various stores, each with a list of items available for purchase. The process involves the following steps:

Defining the Data:

I created a dictionary `stores` where each store is associated with a list of products they sell. For example, `"Amazon"` has books and programming guides, `"Best_Buy"` has electronics, `"K-Mart"` sells bedding items, and `"Nike"` has clothing and shoes. I also defined a `store_seeds` dictionary where each store is assigned a unique seed value to ensure that the random transactions generated are consistent for each store when the code is run multiple times.

Generating Transactions:

I defined the function `generate_transactions()`, which creates a list of random transactions based on the store's available items. The function accepts the list of items, a seed value (to ensure the randomness is consistent across runs), and other parameters like the number of transactions to generate and the minimum and maximum number of items per transaction.

Inside the function, I set the random seed, then looped through and created a specified number of transactions. Each transaction randomly selects between `min_items` and `max_items` items from the list, ensuring the number of items in a transaction is random but within the defined range.

Writing Transactions to CSV:

For each store, I call `generate_transactions()` with the store's item list and seed value to generate the transactions. The transactions are saved into a CSV file with the store name in the format `store_transactions.csv`. The file contains two columns: `Transaction ID` and `Transaction`. Each row represents a transaction with an ID and a list of items that were randomly selected for that transaction.

Output:

After generating the transactions for each store, I print a message to the console indicating that the transactions have been successfully generated for that store.

2. Data Loading and Preprocessing

In my code, I am extracting transactions and their corresponding items from a CSV file. The process involves the following steps:

Reading the CSV File:

I defined a function `extract_transactions_and_items()` that reads a CSV file containing transactions. The function accepts the CSV file as an input and opens it using

Python's built-in `csv.reader`. The file is opened with UTF-8 encoding to handle any special characters, and errors are replaced to prevent issues with malformed data.

Extracting Transaction Data:

Within the function, I first initialize two variables:

- `transactions`: A list that will store each transaction's items.
- `item_set`: A set that will store all unique items encountered across the transactions.

I use `csv.reader` to iterate over the rows in the CSV file. The first row (which contains column headers) is skipped using `next(reader)`.

For each subsequent row, I extract the list of items from the second column (index 1). The items are separated by commas, so I split the string by commas, strip any leading or trailing whitespace, and convert the result into a set to ensure there are no duplicate items within a single transaction. These items are then added to the `transactions` list.

Updating the Item Set:

After processing the items for each transaction, I update the `item_set` with the new items encountered in the current transaction. Using a set ensures that only unique items are retained across all transactions.

Returning the Results:

Finally, the function returns two values:

- `transactions`: A list of sets, where each set represents a transaction and contains the unique items purchased.
- `item_set`: A list of all unique items encountered in all transactions, ensuring there are no duplicates.

Usage

- `Brute-Force Approach`: I originally made this for the Brute-Force Approach, and thus there is no more preprocessing needed
- `Apriori Principle Approach`: In the code, the TransactionEncoder transforms the list of transactions into a one-hot encoded format, where each transaction is represented by a binary vector indicating the presence or absence of items, and then the result is stored in a pandas DataFrame.

3. Iteration Through Candidate Itemsets

In my code, I am using an iterative approach to identify frequent itemsets based on their support values. The process involves generating candidate itemsets, calculating their support, and updating the list of frequent itemsets. The process is as follows:

Brute Force Approach:

The function `brute_force_approach()` implements a brute force approach to generate and check itemsets of increasing size. The steps are as follows:

- `all_frequent_itemsets = {}`: A dictionary `all_frequent_itemsets` is initialized to store the frequent itemsets found during the iteration process.

The iteration proceeds through itemsets of increasing size (from 1-itemsets to k-itemsets) using the following steps:

- `for k in range(1, len(items_list) + 1)`: A loop starts from 1-itemsets to k-itemsets, where k is the current size of the itemsets being considered.
- `current_itemsets = generate_itemsets(items_list, k)`: For each size k, the function `generate_itemsets()` generates all possible combinations of k-itemsets from the list of items.

Generating Itemsets:

The function `generate_itemsets()` is used to generate all possible itemsets of size k from the list of items:

- `generate_combinations(items, 0, [], k, itemsets)`: This recursive helper function `generate_combinations()` generates combinations of size k from the list of items, which are stored in `itemsets`.
- The function returns the generated itemsets, which are combinations of items of size k.

Support Calculation and Filtering:

Once the candidate itemsets are generated, the support for each itemset is calculated using the `calculate_support()` function:

- `support = calculate_support(transactions_list, itemset)`: The support for each itemset is calculated by checking how frequently it appears in the transactions.
- `if support >= min_support`: If the support of an itemset meets the minimum support threshold, it is added to the `new_frequent` dictionary.
- `new_frequent[frozenset(itemset)] = support`: The frequent itemsets are stored in a dictionary, where each itemset is represented as a `frozenset` to ensure immutability and hashability.

Breaking the Loop:

If no new frequent itemsets are found for a given size k, the loop breaks, indicating that larger itemsets will not be frequent either:

- `if not new_frequent`: If `new_frequent` is empty, meaning no frequent itemsets were found for this size, the loop breaks without generating $(k + 1)$-itemsets.

Updating the Itemsets:

If frequent itemsets are found, they are added to the `all_frequent_itemsets` dictionary:

- `all_frequent_itemsets.update(new_frequent)`: The frequent itemsets found in the current iteration are added to the overall dictionary of frequent itemsets.

Result:

This iterative approach continues until no more frequent itemsets are found. The result is a dictionary of all frequent itemsets that meet the minimum support threshold.

Key Functions:

- `generate_itemsets()`: Generates all possible itemsets of a given size k from the list of items.
- `generate_combinations()`: A recursive helper function that generates combinations of size k and appends them to `itemsets`.

4. Minimum Support

In my code, I am calculating the support of an itemset in a list of transactions. The process involves the following steps:

Defining the Itemset:

The function `calculate_support()` accepts two inputs:

- `transactions`: A list of transactions, where each transaction is a set of items.
- `itemset`: A list or set of items for which the support is being calculated.

The first step in the function is to convert the `itemset` into a set using `itemset = set(itemset)` to ensure that the input is in the proper format.

Counting Transactions Containing the Itemset:

Next, the function counts how many transactions contain all the items in the itemset. This is done by iterating over each transaction and checking if the itemset is a subset of the transaction. The expression `itemset.issubset(transaction)` returns `True` if the itemset is contained within the transaction.

- `count = sum(1 for transaction in transactions if itemset.issubset(transaction))`: This line counts the number of transactions where the itemset is a subset by summing 1 for each such transaction.

Calculating the Support:

The support is then calculated by dividing the count of transactions containing the itemset by the total number of transactions. This is done by the following line:

- `return count / len(transactions) if len(transactions) > 0 else 0`: If the number of transactions is greater than 0, the function returns the support as the ratio of transactions containing the itemset to the total number of transactions. If there are no transactions, the function returns 0 to avoid division by zero.

Result:

The function returns the support of the given itemset, which is the proportion of transactions in which the itemset appears.

5. Association Rule Generation

In this section, I generate association rules from the frequent itemsets. The process works as follows:

1. I begin by calling the function `generate_association_rules()`, passing the frequent itemsets, the transactions, and the minimum confidence threshold as arguments.

2. Inside `generate_association_rules()`, I loop through each frequent itemset with more than one item. For each itemset, I generate all possible subsets (antecedents) using the `generate_all_subsets()` function.

3. For each subset (antecedent), I compute the consequent by subtracting the antecedent from the original itemset. The consequent represents the items that are associated with the antecedent.

4. I then calculate the confidence of the rule by dividing the support of the whole itemset (antecedent + consequent) by the support of the antecedent. The support values are fetched from the list of frequent itemsets.

5. If the confidence of the rule meets or exceeds the minimum confidence threshold, I store the rule, including the antecedent, consequent, support, and confidence values.

6. The `generate_association_rules()` function returns a list of all valid association rules.

Additionally, the function `generate_all_subsets()` is responsible for generating all non-empty subsets of an itemset using recursion, which is necessary for finding the antecedents of the rules.

Key Functions:

- `generate_association_rules()`: Generates association rules based on the frequent itemsets and confidence threshold.
- `generate_all_subsets()`: Generates all non-empty subsets of an itemset, used to identify antecedents.

Requirements

- **Python Version: 3.8.20**
- **Conda Version: 24.11.3**
- **Python Standard Libraries:**
  - `random` - For generating random test data.
  - `csv` - For handling CSV files.
  - `time` - For performance measurement.
- **Data Mining and Apriori Algorithm:**

  - `mlxtend` - Provides functions for Apriori and association rule mining.
    - Install using: `pip install mlxtend`
  - `pandas` - For data manipulation and preprocessing.
  - `mlxtend.frequent_patterns` - Includes the Apriori and association rule functions.
  - `mlxtend.preprocessing.TransactionEncoder` - Used for transforming transaction data.

- **Visualization and Testing:**
  - `matplotlib.pyplot` - For plotting graphs.
  - `numpy` - For numerical computations, and creating quick arrays.

Running the Code

The Python code consists of two main scripts:

- **Generate_Data.py** - This script generates the required transaction data.
- **Benyamin_Plaksienko_Midterm_Project.py** - The main script that executes the Apriori Algorithm.

Steps to Run the Code:

1. If the transaction data has not been generated yet, first run:

```
python Generate_Data.py
```

2. Once the data is generated, or if it already exists, run:

```
python Benyamin_Plaksienko_Midterm_Project.py
```

3. When using Jupyter Notebook, run the following command to increase the data rate limit and prevent output issues with large datasets:

```
jupyter notebook --ServerApp.iopub_data_rate_limit=10000000
```

   Reason: This increases the data rate limit for the notebook server, preventing issues with large output data, such as frequent itemsets and association rules exceeding the default data transmission limit.

Conclusion

float The comparison of Brute Force and Apriori algorithms for association rule mining reveals significant differences in their performance across varying minimum support and confidence thresholds.

The results from Figure 1 show that the Brute Force algorithm maintains a relatively stable execution time across different support values, with minor fluctuations. In contrast, the Apriori algorithm exhibits significantly higher execution times at low support values but becomes much more efficient as the support threshold increases. This behavior highlights that Apriori is computationally expensive when generating frequent itemsets at low support but significantly outperforms Brute Force when the support threshold is increased.

Similarly, Figure 2 illustrates the execution time behavior concerning minimum confidence levels. The Brute Force algorithm maintains a nearly constant execution time, whereas the Apriori algorithm starts with significantly higher execution times at low confidence values but improves as confidence increases. This indicates that Apriori requires more computation at lower confidence levels but benefits from higher confidence thresholds.

Key Observations:

- **Brute Force has a stable but higher execution time,** making it inefficient and less scalable compared to Apriori.
- **Apriori is the preferred algorithm for large datasets,** where tuning support and confidence values can enhance computational efficiency.

In general, the Apriori algorithm demonstrates greater scalability than the Brute Force approach for mining association rules, especially when support and confidence thresholds are optimized. This advantage becomes more pronounced when dealing with higher support and confidence values. In such cases, the Apriori algorithm can avoid unnecessary checks by pruning non-promising itemsets early in the process. These pruning steps significantly reduce the number of candidate itemsets considered, thus improving efficiency. On the other hand, the Brute Force method lacks such pruning capabilities and ends up performing redundant checks, even for itemsets that ultimately do not meet the user-defined thresholds, making it less efficient under these conditions. These pruning checks work the opposite with lower support and confidence values, because Apriori algorithm ends up doing redundant checks.

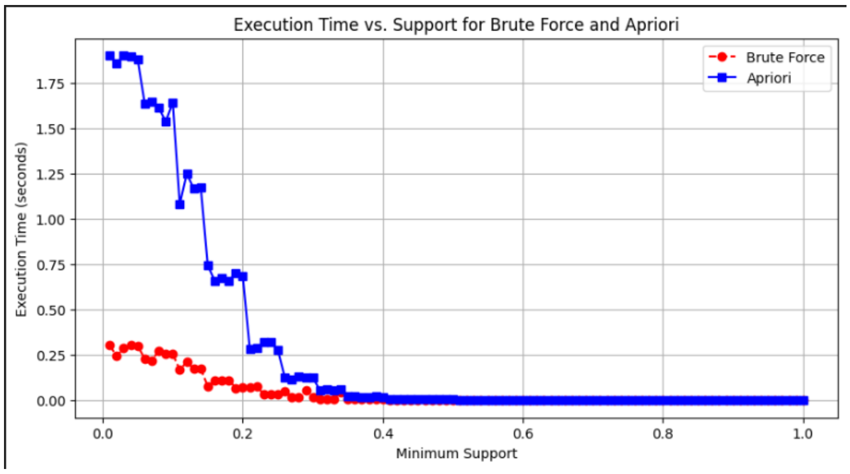


Figure 1: Execution Time vs. Support for Brute Force and Apriori

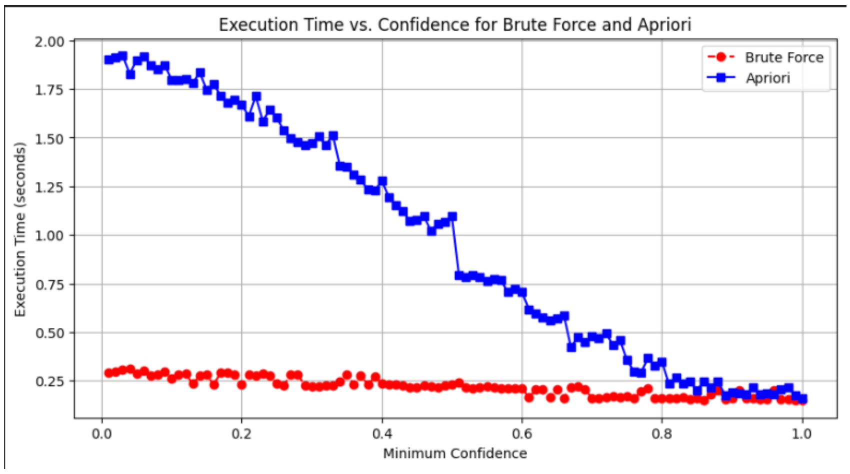
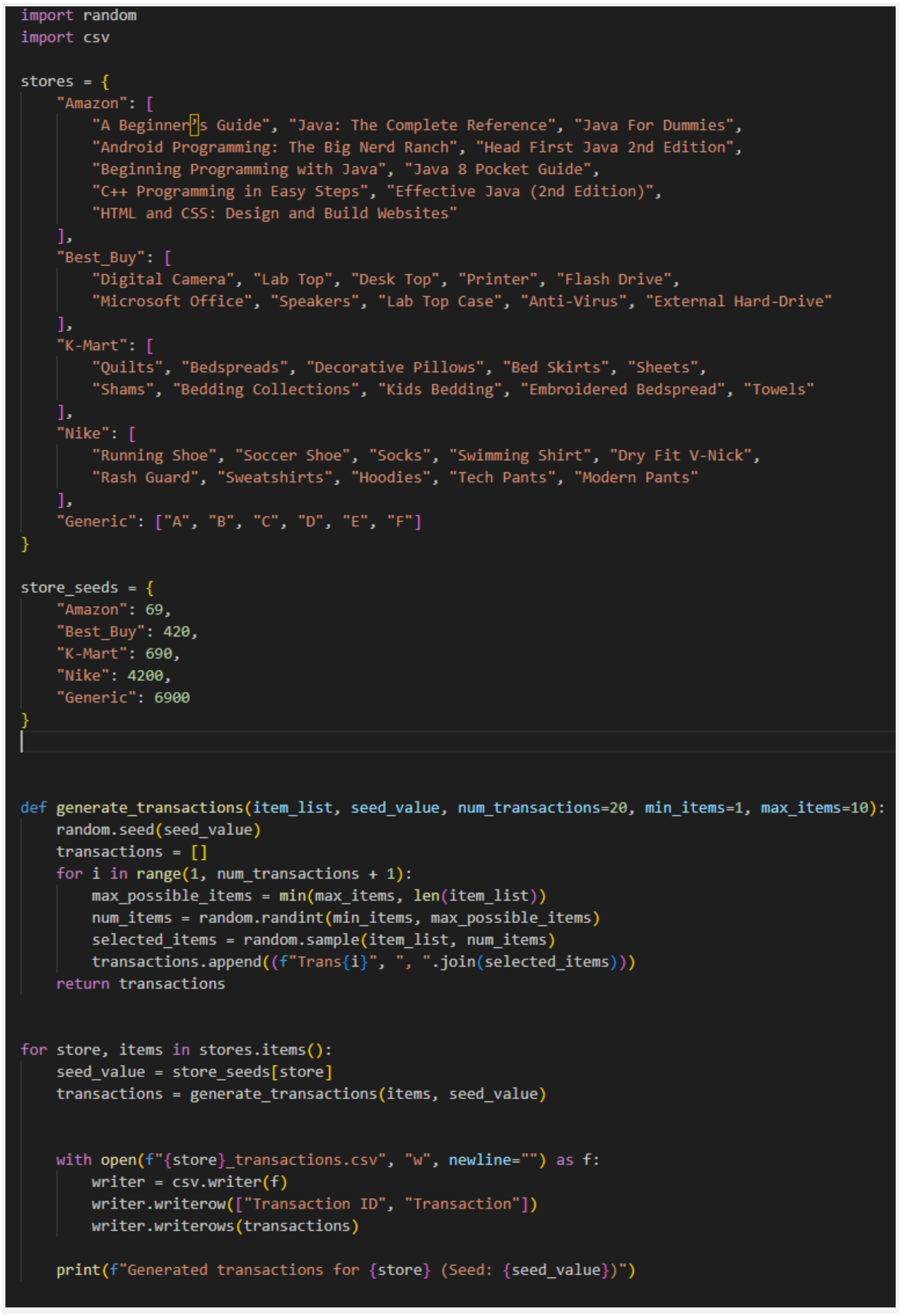


Figure 2: Execution Time vs. Confidence for Brute Force and Apriori

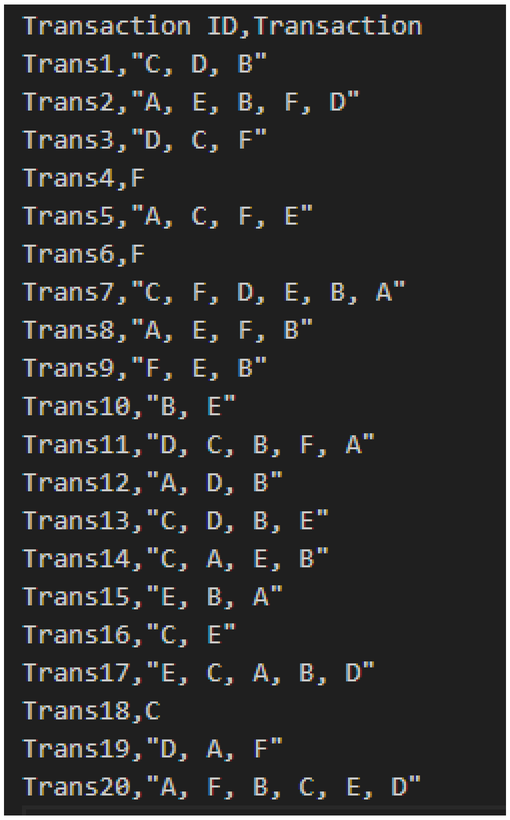
Below are screenshots of the code from python file:

Code to generate transaction data as a csv



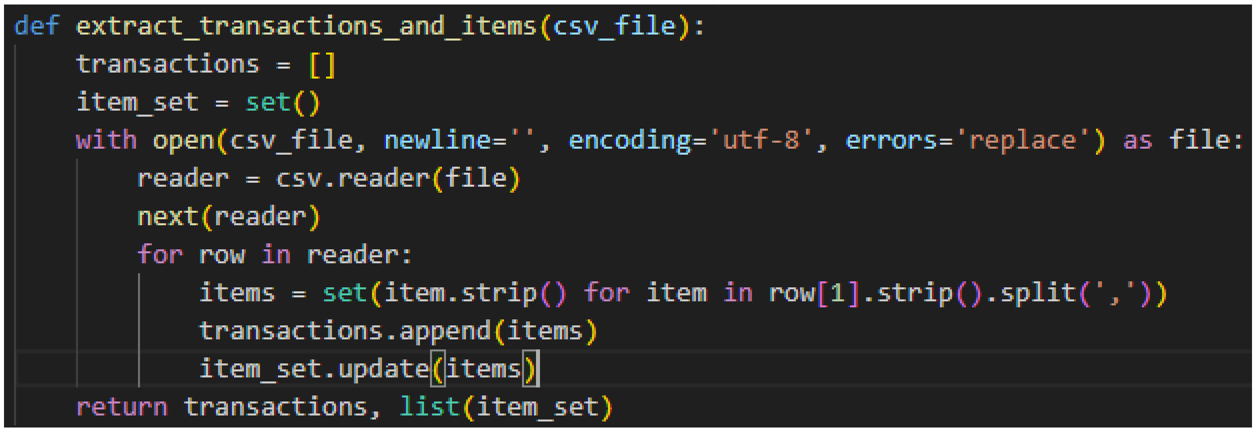
```python
import random
import csv

stores = {
    "Amazon": [
        "A Beginner's Guide", "Java: The Complete Reference", "Java For Dummies",
        "Android Programming: The Big Nerd Ranch", "Head First Java 2nd Edition",
        "Beginning Programming with Java", "Java 8 Pocket Guide",
        "C++ Programming in Easy Steps", "Effective Java (2nd Edition)",
        "HTML and CSS: Design and Build Websites"
    ],
    "Best_Buy": [
        "Digital Camera", "Lab Top", "Desk Top", "Printer", "Flash Drive",
        "Microsoft Office", "Speakers", "Lab Top Case", "Anti-Virus", "External Hard-Drive"
    ],
    "K-Mart": [
        "Quilts", "Bedspreads", "Decorative Pillows", "Bed Skirts", "Sheets",
        "Shams", "Bedding Collections", "Kids Bedding", "Embroidered Bedspread", "Towels"
    ],
    "Nike": [
        "Running Shoe", "Soccer Shoe", "Socks", "Swimming Shirt", "Dry Fit V-Nick",
        "Rash Guard", "Sweatshirts", "Hoodies", "Tech Pants", "Modern Pants"
    ],
    "Generic": ["A", "B", "C", "D", "E", "F"]
}

store_seeds = {
    "Amazon": 69,
    "Best_Buy": 420,
    "K-Mart": 690,
    "Nike": 4200,
    "Generic": 6900
}


def generate_transactions(item_list, seed_value, num_transactions=20, min_items=1, max_items=10):
    random.seed(seed_value)
    transactions = []
    for i in range(1, num_transactions + 1):
        max_possible_items = min(max_items, len(item_list))
        num_items = random.randint(min_items, max_possible_items)
        selected_items = random.sample(item_list, num_items)
        transactions.append((f"Trans{i}", ", ".join(selected_items)))
    return transactions


for store, items in stores.items():
    seed_value = store_seeds[store]
    transactions = generate_transactions(items, seed_value)


    with open(f"{store}_transactions.csv", "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["Transaction ID", "Transaction"])
        writer.writerows(transactions)

    print(f"Generated transactions for {store} (Seed: {seed_value})")
```

Here are what the csv files format look like
Name is {store}_transactions.CSV



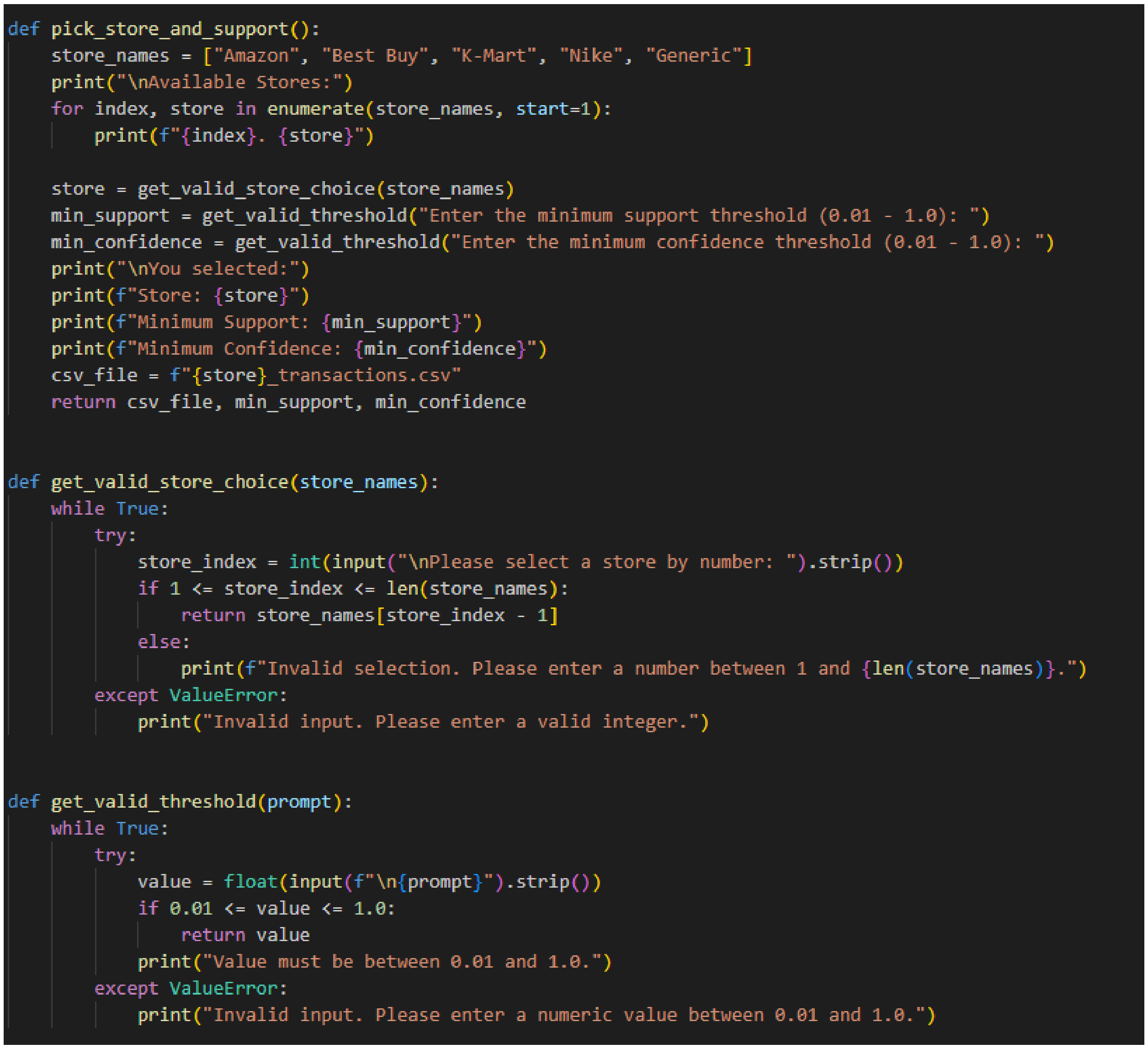
```
Transaction ID,Transaction
Trans1,"C, D, B"
Trans2,"A, E, B, F, D"
Trans3,"D, C, F"
Trans4,F
Trans5,"A, C, F, E"
Trans6,F
Trans7,"C, F, D, E, B, A"
Trans8,"A, E, F, B"
Trans9,"F, E, B"
Trans10,"B, E"
Trans11,"D, C, B, F, A"
Trans12,"A, D, B"
Trans13,"C, D, B, E"
Trans14,"C, A, E, B"
Trans15,"E, B, A"
Trans16,"C, E"
Trans17,"E, C, A, B, D"
Trans18,C
Trans19,"D, A, F"
Trans20,"A, F, B, C, E, D"
```

Code to preprocess and extract transactions from csv. Returns a list of sets for transactions and a list of items



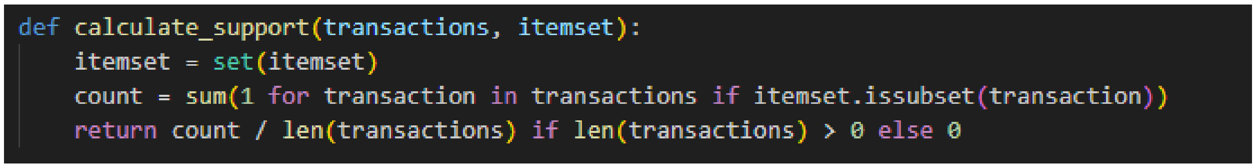
```python
def extract_transactions_and_items(csv_file):
    transactions = []
    item_set = set()
    with open(csv_file, newline='', encoding='utf-8', errors='replace') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            items = set(item.strip() for item in row[1].strip().split(','))
            transactions.append(items)
            item_set.update(items)
    return transactions, list(item_set)
```

Prompts to choose which store you want,min confidence, and min support, and handles errors such as incorrect input.



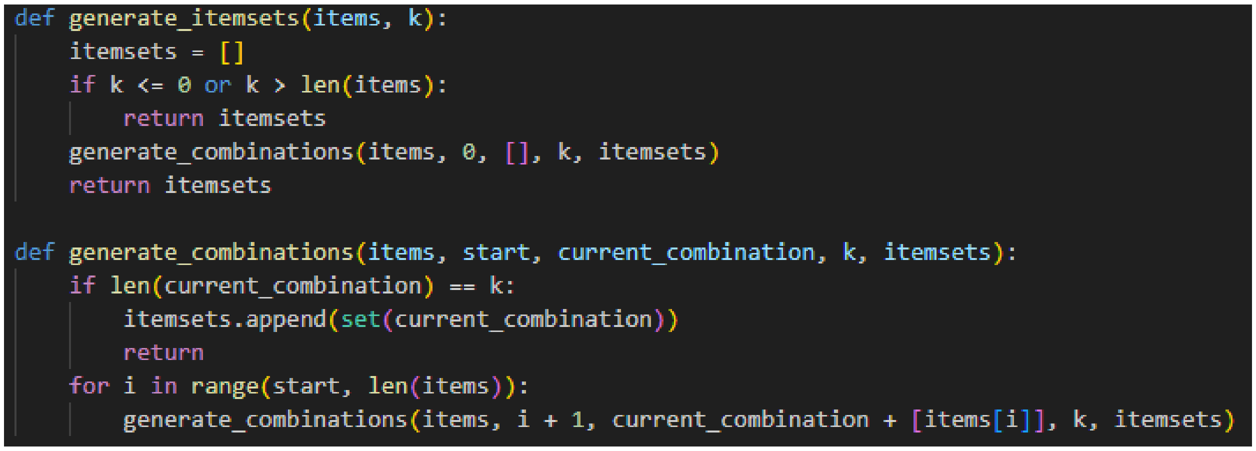
```python
def pick_store_and_support():
    store_names = ["Amazon", "Best Buy", "K-Mart", "Nike", "Generic"]
    print("\nAvailable Stores:")
    for index, store in enumerate(store_names, start=1):
        print(f"{index}. {store}")

    store = get_valid_store_choice(store_names)
    min_support = get_valid_threshold("Enter the minimum support threshold (0.01 - 1.0): ")
    min_confidence = get_valid_threshold("Enter the minimum confidence threshold (0.01 - 1.0): ")
    print("\nYou selected:")
    print(f"Store: {store}")
    print(f"Minimum Support: {min_support}")
    print(f"Minimum Confidence: {min_confidence}")
    csv_file = f"{store}_transactions.csv"
    return csv_file, min_support, min_confidence


def get_valid_store_choice(store_names):
    while True:
        try:
            store_index = int(input("\nPlease select a store by number: ").strip())
            if 1 <= store_index <= len(store_names):
                return store_names[store_index - 1]
            else:
                print(f"Invalid selection. Please enter a number between 1 and {len(store_names)}.")
        except ValueError:
            print("Invalid input. Please enter a valid integer.")


def get_valid_threshold(prompt):
    while True:
        try:
            value = float(input(f"\n{prompt}").strip())
            if 0.01 <= value <= 1.0:
                return value
            print("Value must be between 0.01 and 1.0.")
        except ValueError:
            print("Invalid input. Please enter a numeric value between 0.01 and 1.0.")
```

Calculating support

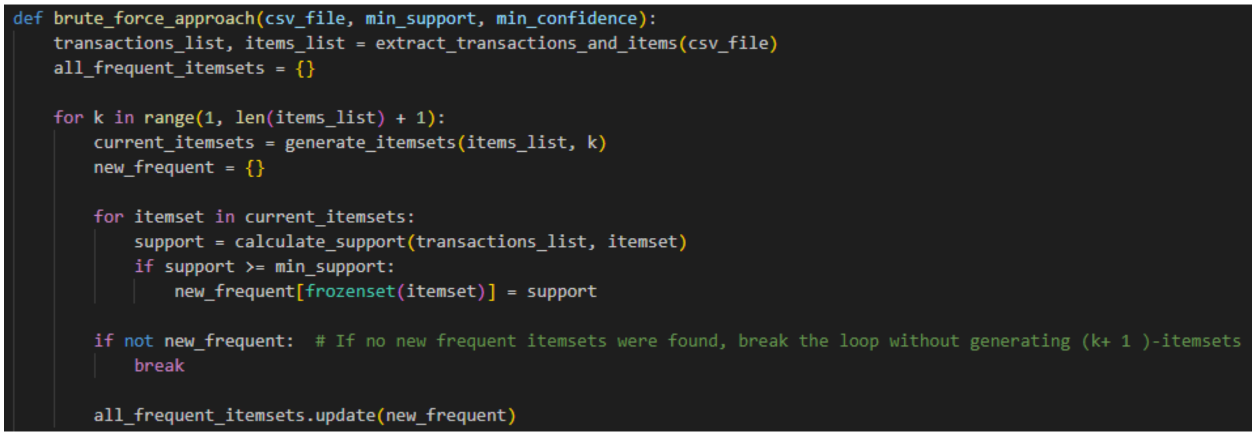


```python
def calculate_support(transactions, itemset):
    itemset = set(itemset)
    count = sum(1 for transaction in transactions if itemset.issubset(transaction))
    return count / len(transactions) if len(transactions) > 0 else 0
```

Finding Frequent Itemsets and checking support

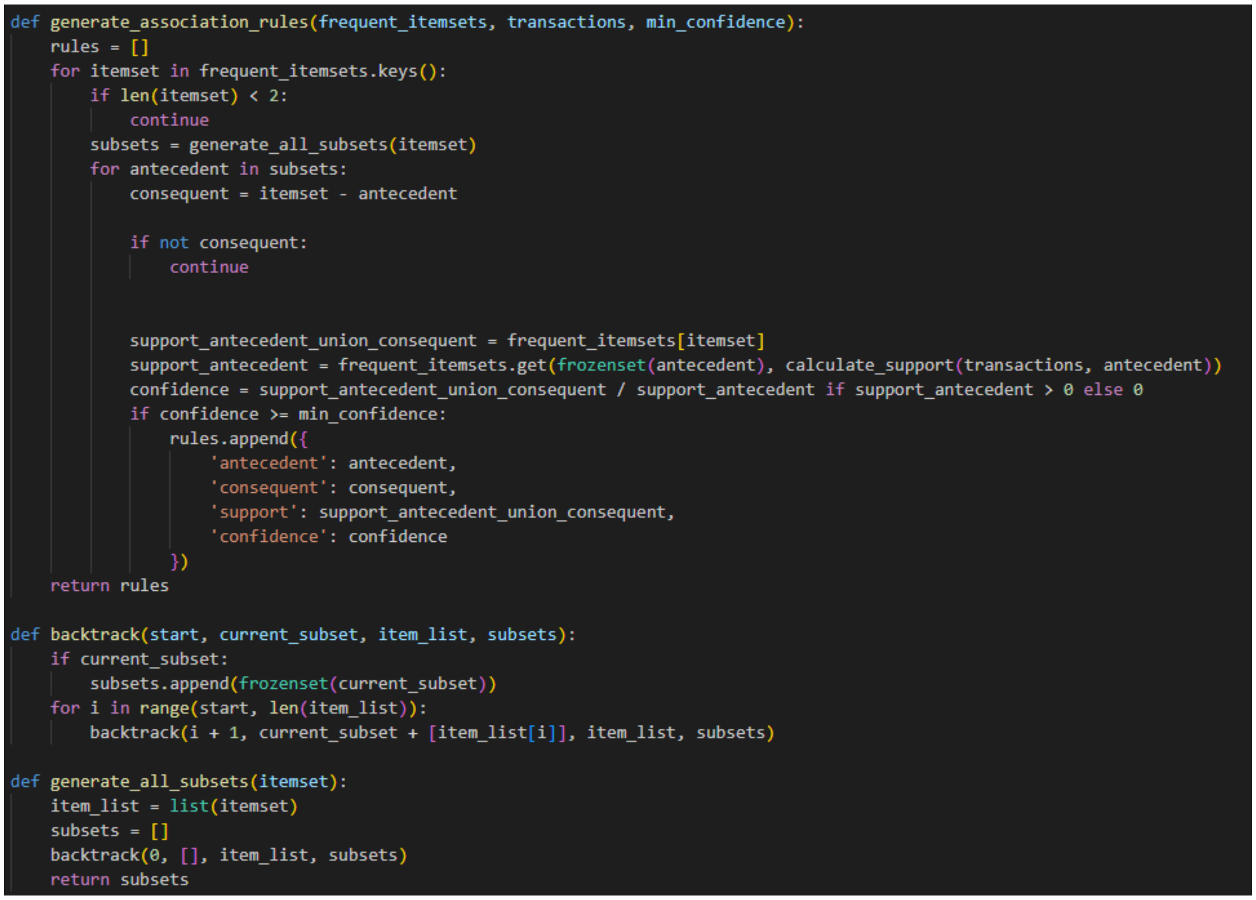


```python
def generate_itemsets(items, k):
    itemsets = []
    if k <= 0 or k > len(items):
        return itemsets
    generate_combinations(items, 0, [], k, itemsets)
    return itemsets

def generate_combinations(items, start, current_combination, k, itemsets):
    if len(current_combination) == k:
        itemsets.append(set(current_combination))
        return
    for i in range(start, len(items)):
        generate_combinations(items, i + 1, current_combination + [items[i]], k, itemsets)
```

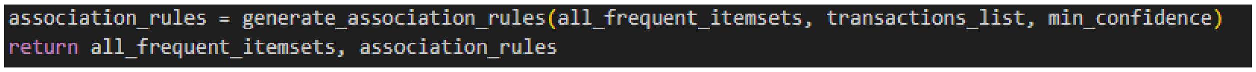


```python
def brute_force_approach(csv_file, min_support, min_confidence):
    transactions_list, items_list = extract_transactions_and_items(csv_file)
    all_frequent_itemsets = {}

    for k in range(1, len(items_list) + 1):
        current_itemsets = generate_itemsets(items_list, k)
        new_frequent = {}

        for itemset in current_itemsets:
            support = calculate_support(transactions_list, itemset)
            if support >= min_support:
                new_frequent[frozenset(itemset)] = support

        if not new_frequent:  # If no new frequent itemsets were found, break the loop without generating (k+ 1 )-itemsets
            break

        all_frequent_itemsets.update(new_frequent)
```

Generate association rules and calculate confidence and filter association rules, and return for brute force and call for association rule generation

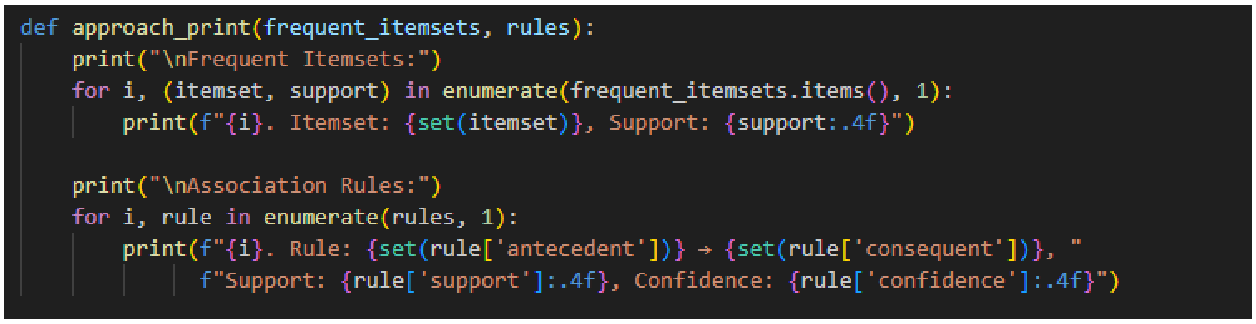


```python
def generate_association_rules(frequent_itemsets, transactions, min_confidence):
    rules = []
    for itemset in frequent_itemsets.keys():
        if len(itemset) < 2:
            continue
        subsets = generate_all_subsets(itemset)
        for antecedent in subsets:
            consequent = itemset - antecedent

            if not consequent:
                continue


            support_antecedent_union_consequent = frequent_itemsets[itemset]
            support_antecedent = frequent_itemsets.get(frozenset(antecedent), calculate_support(transactions, antecedent))
            confidence = support_antecedent_union_consequent / support_antecedent if support_antecedent > 0 else 0
            if confidence >= min_confidence:
                rules.append({
                    'antecedent': antecedent,
                    'consequent': consequent,
                    'support': support_antecedent_union_consequent,
                    'confidence': confidence
                })
    return rules

def backtrack(start, current_subset, item_list, subsets):
    if current_subset:
        subsets.append(frozenset(current_subset))
    for i in range(start, len(item_list)):
        backtrack(i + 1, current_subset + [item_list[i]], item_list, subsets)

def generate_all_subsets(itemset):
    item_list = list(itemset)
    subsets = []
    backtrack(0, [], item_list, subsets)
    return subsets
```



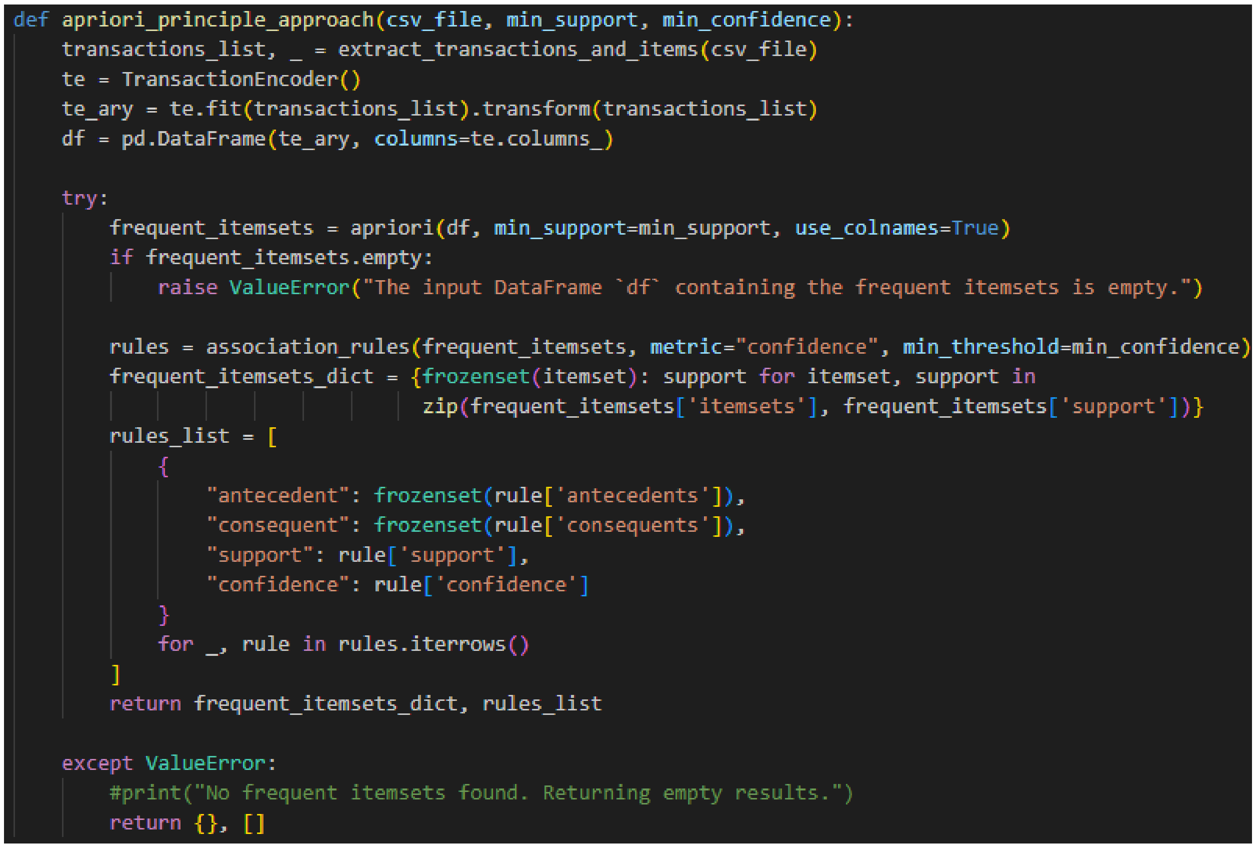
```python
association_rules = generate_association_rules(all_frequent_itemsets, transactions_list, min_confidence)
return all_frequent_itemsets, association_rules
```

Printing the data nicely for both approaches



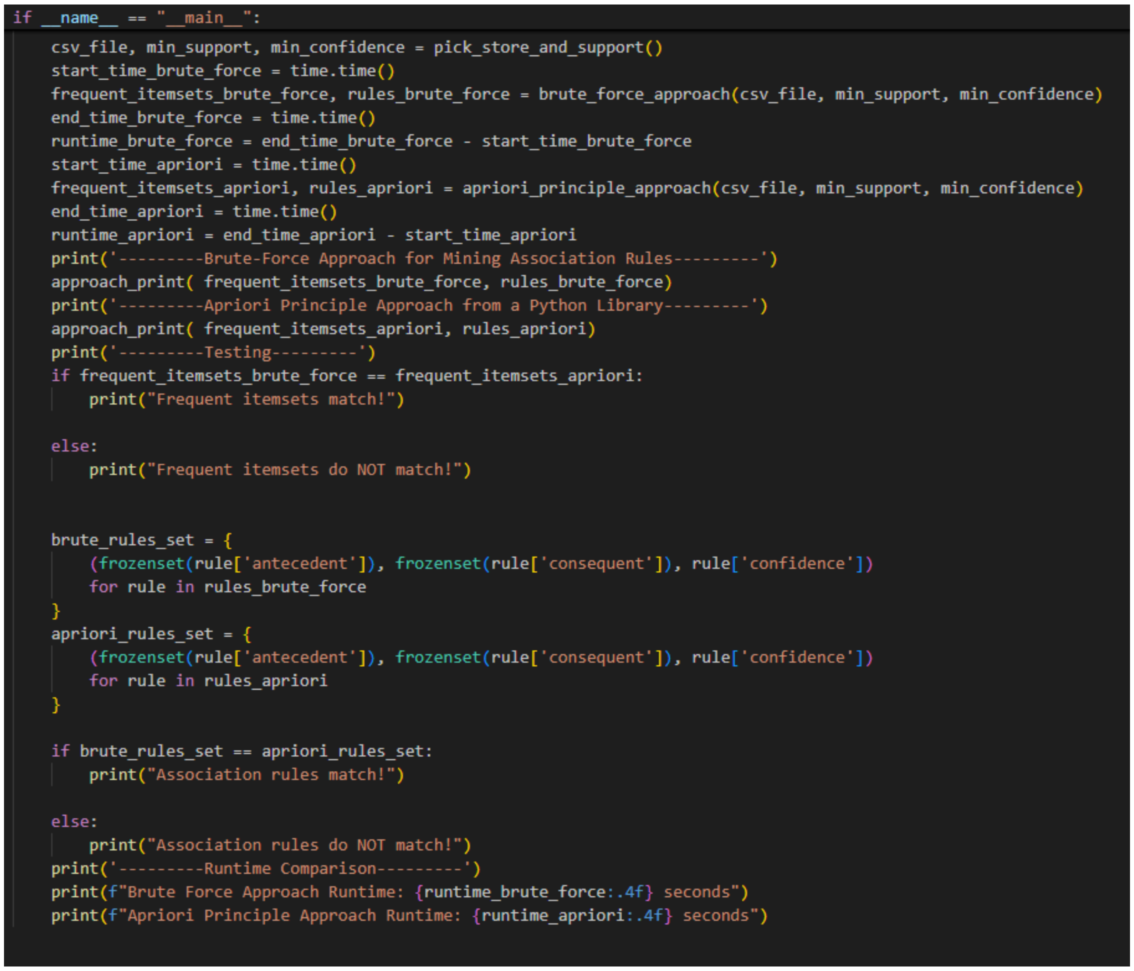
```python
def approach_print(frequent_itemsets, rules):
    print("\nFrequent Itemsets:")
    for i, (itemset, support) in enumerate(frequent_itemsets.items(), 1):
        print(f"{i}. Itemset: {set(itemset)}, Support: {support:.4f}")

    print("\nAssociation Rules:")
    for i, rule in enumerate(rules, 1):
        print(f"{i}. Rule: {set(rule['antecedent'])} -> {set(rule['consequent'])}, "
              f"Support: {rule['support']:.4f}, Confidence: {rule['confidence']:.4f}")
```

Apriori principle approach using existing library/package



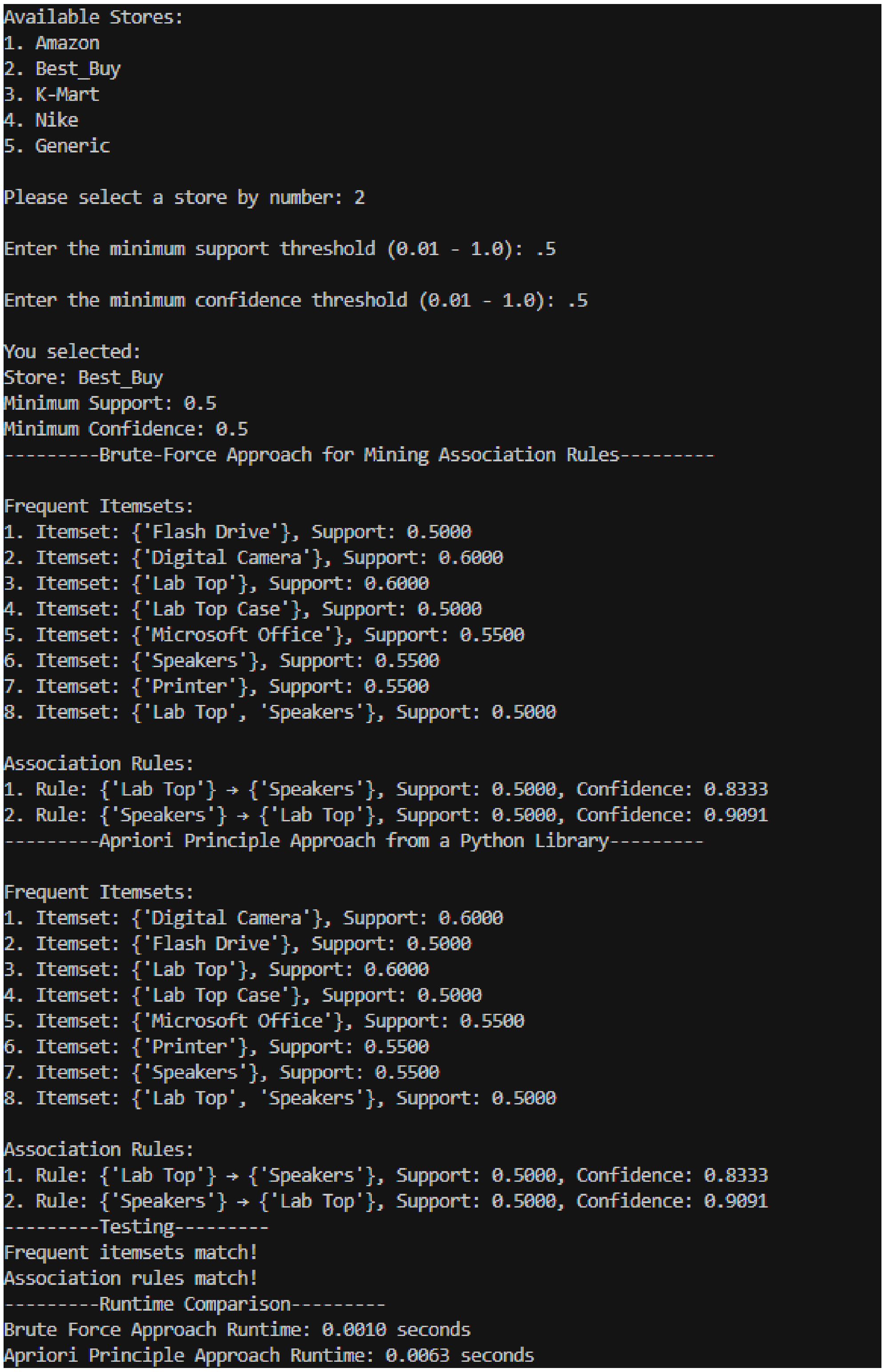
```python
def apriori_principle_approach(csv_file, min_support, min_confidence):
    transactions_list, _ = extract_transactions_and_items(csv_file)
    te = TransactionEncoder()
    te_ary = te.fit(transactions_list).transform(transactions_list)
    df = pd.DataFrame(te_ary, columns=te.columns_)

    try:
        frequent_itemsets = apriori(df, min_support=min_support, use_colnames=True)
        if frequent_itemsets.empty:
            raise ValueError("The input DataFrame `df` containing the frequent itemsets is empty.")

        rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=min_confidence)
        frequent_itemsets_dict = {frozenset(itemset): support for itemset, support in
                                  zip(frequent_itemsets['itemsets'], frequent_itemsets['support'])}
        rules_list = [
            {
                "antecedent": frozenset(rule['antecedents']),
                "consequent": frozenset(rule['consequents']),
                "support": rule['support'],
                "confidence": rule['confidence']
            }
            for _, rule in rules.iterrows()
        ]
        return frequent_itemsets_dict, rules_list

    except ValueError:
        #print("No frequent itemsets found. Returning empty results.")
        return {}, []
```

When Running python file it runs both algorithms after prompt, and it check both the algorithms run time, and if they match this is code for that



```python
if __name__ == "__main__":
    csv_file, min_support, min_confidence = pick_store_and_support()
    start_time_brute_force = time.time()
    frequent_itemsets_brute_force, rules_brute_force = brute_force_approach(csv_file, min_support, min_confidence)
    end_time_brute_force = time.time()
    runtime_brute_force = end_time_brute_force - start_time_brute_force
    start_time_apriori = time.time()
    frequent_itemsets_apriori, rules_apriori = apriori_principle_approach(csv_file, min_support, min_confidence)
    end_time_apriori = time.time()
    runtime_apriori = end_time_apriori - start_time_apriori
    print('---------Brute-Force Approach for Mining Association Rules---------')
    approach_print( frequent_itemsets_brute_force, rules_brute_force)
    print('---------Apriori Principle Approach from a Python Library---------')
    approach_print( frequent_itemsets_apriori, rules_apriori)
    print('---------Testing---------')
    if frequent_itemsets_brute_force == frequent_itemsets_apriori:
        print("Frequent itemsets match!")

    else:
        print("Frequent itemsets do NOT match!")


    brute_rules_set = {
        (frozenset(rule['antecedent']), frozenset(rule['consequent']), rule['confidence'])
        for rule in rules_brute_force
    }
    apriori_rules_set = {
        (frozenset(rule['antecedent']), frozenset(rule['consequent']), rule['confidence'])
        for rule in rules_apriori
    }

    if brute_rules_set == apriori_rules_set:
        print("Association rules match!")

    else:
        print("Association rules do NOT match!")
    print('---------Runtime Comparison---------')
    print(f"Brute Force Approach Runtime: {runtime_brute_force:.4f} seconds")
    print(f"Apriori Principle Approach Runtime: {runtime_apriori:.4f} seconds")
```

This is how it looks in terminal when running code



```
Available Stores:
1. Amazon
2. Best_Buy
3. K-Mart
4. Nike
5. Generic

Please select a store by number: 2

Enter the minimum support threshold (0.01 - 1.0): .5

Enter the minimum confidence threshold (0.01 - 1.0): .5

You selected:
Store: Best_Buy
Minimum Support: 0.5
Minimum Confidence: 0.5
---------Brute-Force Approach for Mining Association Rules---------

Frequent Itemsets:
1. Itemset: {'Flash Drive'}, Support: 0.5000
2. Itemset: {'Digital Camera'}, Support: 0.6000
3. Itemset: {'Lab Top'}, Support: 0.6000
4. Itemset: {'Lab Top Case'}, Support: 0.5000
5. Itemset: {'Microsoft Office'}, Support: 0.5500
6. Itemset: {'Speakers'}, Support: 0.5500
7. Itemset: {'Printer'}, Support: 0.5500
8. Itemset: {'Lab Top', 'Speakers'}, Support: 0.5000

Association Rules:
1. Rule: {'Lab Top'} → {'Speakers'}, Support: 0.5000, Confidence: 0.8333
2. Rule: {'Speakers'} → {'Lab Top'}, Support: 0.5000, Confidence: 0.9091
---------Apriori Principle Approach from a Python Library---------

Frequent Itemsets:
1. Itemset: {'Digital Camera'}, Support: 0.6000
2. Itemset: {'Flash Drive'}, Support: 0.5000
3. Itemset: {'Lab Top'}, Support: 0.6000
4. Itemset: {'Lab Top Case'}, Support: 0.5000
5. Itemset: {'Microsoft Office'}, Support: 0.5500
6. Itemset: {'Printer'}, Support: 0.5500
7. Itemset: {'Speakers'}, Support: 0.5500
8. Itemset: {'Lab Top', 'Speakers'}, Support: 0.5000

Association Rules:
1. Rule: {'Lab Top'} → {'Speakers'}, Support: 0.5000, Confidence: 0.8333
2. Rule: {'Speakers'} → {'Lab Top'}, Support: 0.5000, Confidence: 0.9091
---------Testing---------
Frequent itemsets match!
Association rules match!
---------Runtime Comparison---------
Brute Force Approach Runtime: 0.0010 seconds
Apriori Principle Approach Runtime: 0.0063 seconds
```

The source code (.py file) and data sets (.csv files) will be attached to the zip file.

Link to Git Repository:https://github.com/BenyaminPlaksienkoNJIT/CS-634-Data-Mining-Midterm-Term-Project