

CS634_Fall2024_jd849

October 12, 2024

```
[347]: from IPython.display import Image
```

1 CS634 Midterm Project

Name: Jaysheel Dodia UCID: jd849 MailID: jd849@njit.edu Date: 10/12/2024 (MM/DD/YYYY)
Instructor: Dr. Yasser Abdullallah Class: CS634

1.1 Abstract

In this project, I explore the Apriori Algorithm to uncover associations in retail transactions. I created a brute force implementation, put it through several dataset tests, and evaluated its effectiveness against pre-installed Python libraries. Important metrics were employed to assess frequent itemsets and association rules, including confidence and support. The results show how successful the algorithm is at retail data mining.

1.2 Introduction

This project presents an exploration of the Apriori Algorithm, a foundational method in data mining, to identify associations within retail transaction data. The objective is to assess the algorithm's effectiveness and efficiency by implementing it alongside various data mining concepts and techniques. Through the design and development of custom data mining tools, I have created a tailored model for extracting valuable insights from transaction data. This project aims to improve our understanding of customer behavior and purchasing patterns to make informed business and retail decisions. This report focuses on two key aspects:

1. The implementation of the Apriori algorithm and Association Rule Mining using a brute force approach in Python.
2. Verification of the implementation using the MLXTend library to compare the results of the custom implementation with an optimized library-based version.
3. Note the time difference between brute force algorithm and prebuilt libraries
4. Implement FP Tree using the Prebuilt optimized libraries and compare the execution time with that of the Apriori algorithm with both methods Brute force and library-based.

1.3 Frequent Itemsets

Frequent Itemsets are the sets of items that appear together frequently in a transaction dataset. Finding all groupings of itemsets that satisfy a specific threshold of occurrence (support) within the dataset is the aim of frequent itemset mining.

1.4 Association Rule Mining

Association Rules between elements in big databases can be found through mining. It is employed to identify robust rules that, given the presence of one item, are likely to predict the occurrence of another. The most well-known instance of this is the “market basket analysis,” in which retailers examine sales data to determine which products are frequently purchased in tandem. Steps: 1. Identify frequent Itemsets 2. Generate Candidate Association Rules 3. Calculate Confidence for each rule 4. Prune rules that don’t meet minimum confidence 5. Output the Association Rules

1.5 FP-Tree and FP-Growth

FP-Tree (Frequent Pattern Tree) is a data structure used in the FP-Growth algorithm, an efficient method of mining frequent itemsets without generating candidate sets (unlike Apriori). The FP-Growth algorithm is based on the use of a Frequent Pattern Tree (FP-Tree), a compact structure that stores the database in a way that preserves itemset association. Unlike the Apriori algorithm, which generates candidate itemsets explicitly and requires multiple database passes, FP-Growth scans the database only twice and avoids generating unnecessary candidates.

1.6 FP-Growth vs. Apriori

- Efficiency: FP-Growth is significantly faster than Apriori for large datasets because it avoids generating candidate itemsets explicitly and reduces the number of database scans to two.
- Memory: FP-Growth uses an FP-Tree to store transactions compactly, whereas Apriori requires storage of many intermediate candidate itemsets.
- Implementation: Although FP-Growth is more efficient, it is more complex to implement from scratch than Apriori, due to the tree-building and recursive mining processes.

1.7 Brute Force

The Brute Force Apriori algorithm is association rule mining to discover frequent itemsets within a dataset. It works on the principle of generating all the possible combinations of items and then filtering them based upon the specified minimum support threshold.

Steps: 1. Define the Minimum Support. 2. Generate Candidate Set Items of size $k=1$. 3. Calculate Support. 4. Prune items that do not meet the minimum support requirement. 5. Repeat steps 2-4 and increment k until no more frequent items can be generated.

1.8 Using Prebuilt Libraries

MLxtend is a Python library that provides implementations for data science and machine learning. It includes the Apriori and FP-Growth algorithms, along with functionality to generate association rules.

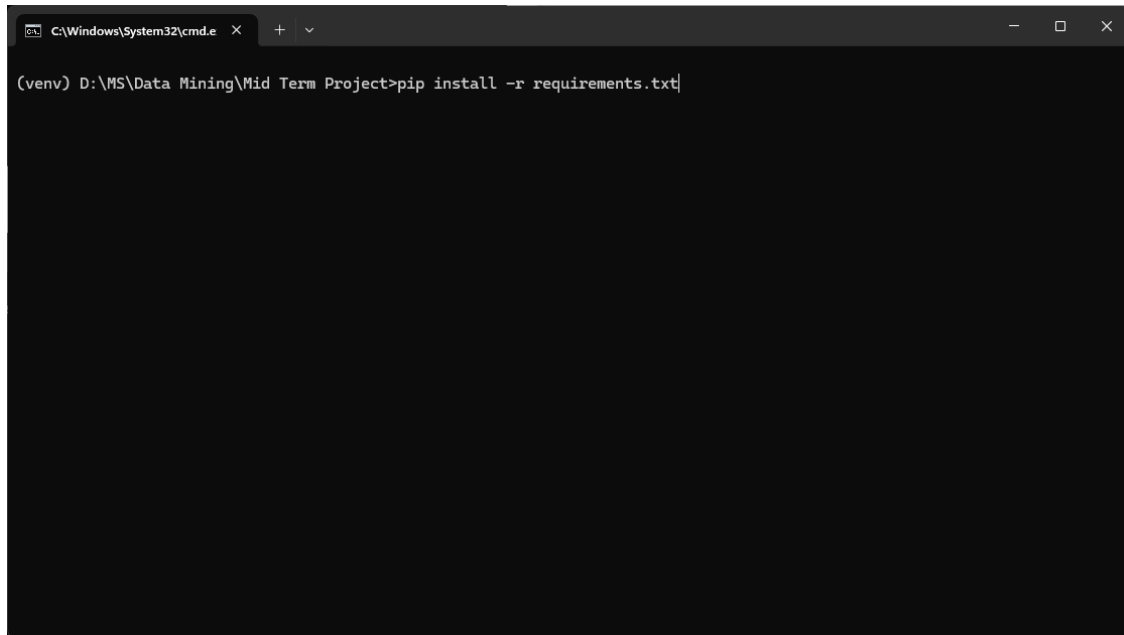
1.9 Prerequisite

1. First we clone our github repository [Github Repository](#).
2. Now before executing the python code we need to update our dependencies by writing the command `pip install -r requirements.txt` (Refer screenshot 1)
3. Once that is done we can directly execute our Python file (Refer screenshot 2 & 3)

1.9.1 Screenshot 1

[348]: Image("screenshots/1.png")

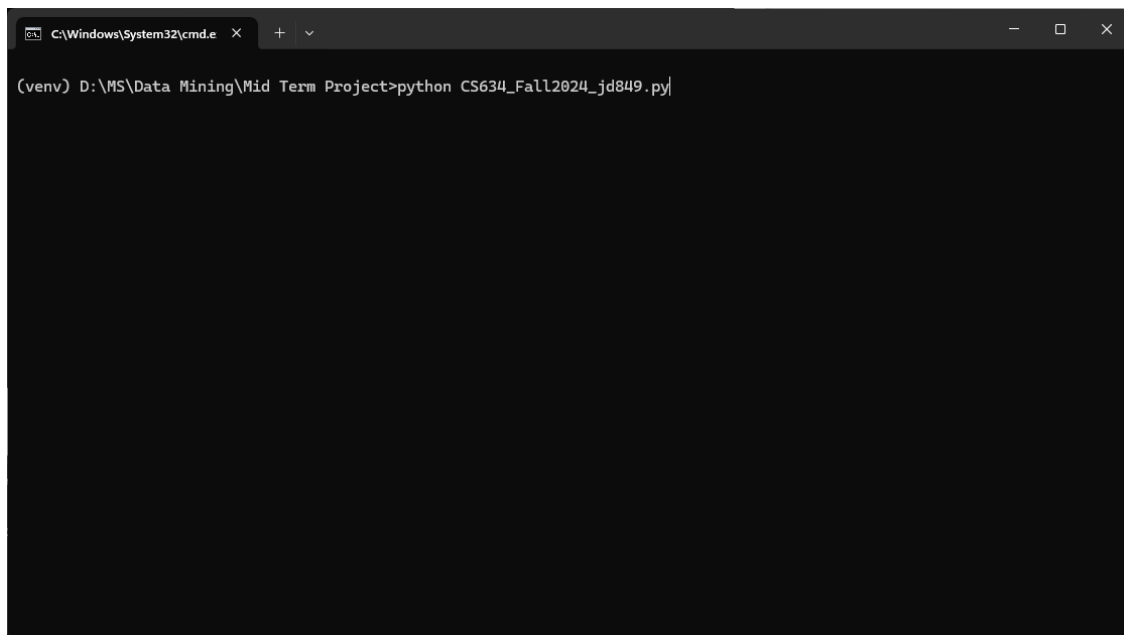
[348]:



1.10 Screenshot 2

[349]: Image("screenshots/2.png")

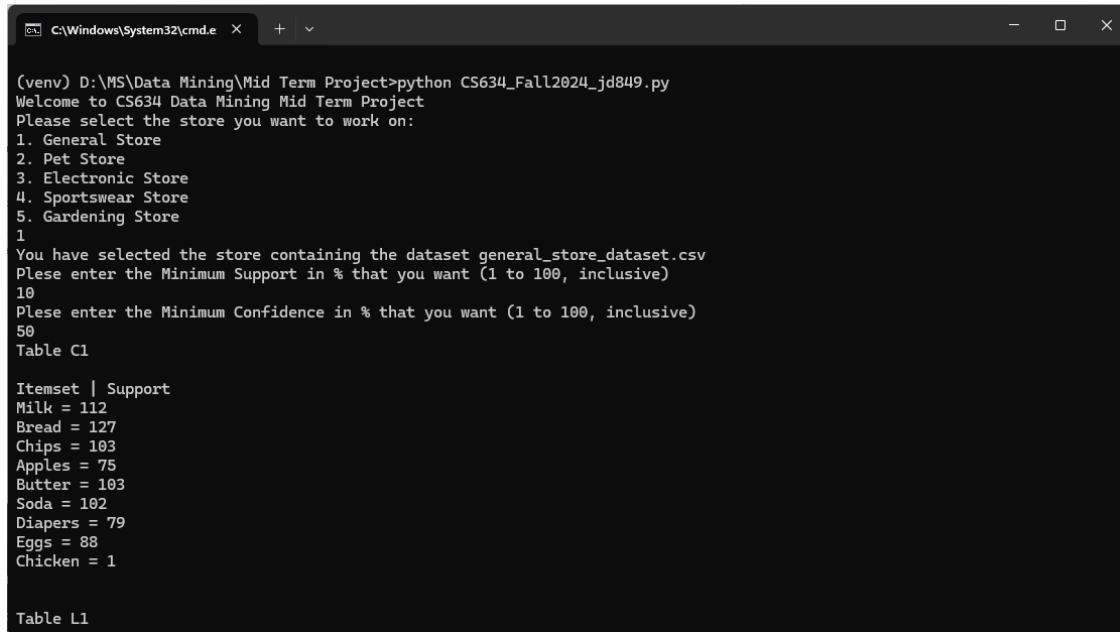
[349]:



1.11 Screenshot 3

```
[350]: Image("screenshots/3.png")
```

[350]:



```
(venv) D:\MS\Data Mining\Mid Term Project>python CS634_Fall2024_jd849.py
Welcome to CS634 Data Mining Mid Term Project
Please select the store you want to work on:
1. General Store
2. Pet Store
3. Electronic Store
4. Sportswear Store
5. Gardening Store
1
You have selected the store containing the dataset general_store_dataset.csv
Please enter the Minimum Support in % that you want (1 to 100, inclusive)
10
Please enter the Minimum Confidence in % that you want (1 to 100, inclusive)
50
Table C1

Itemset | Support
Milk = 112
Bread = 127
Chips = 103
Apples = 75
Butter = 103
Soda = 102
Diapers = 79
Eggs = 88
Chicken = 1

Table L1
```

1.11.1 Importing the libraries

```
[351]: import time
import pandas as pd
from itertools import combinations, permutations
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules, fpgrowth
```

1.11.2 Datastore Selection

- Prompt the users to select a store.
- There are 5 different datasets, each containing 200 transactions included in the `datasets` folder.
- This dataset is generated using a python script called `create_dataset.py` to generate a large dataset (hundreds or thousands of transactions, but for now we kept 200) with more number of transactions for our project. This python script is included along with the code.
- Implemented a flow control for correct input and discard any invalid input so our program doesn't crash.

```
[352]: datasets = {
    1: "general_store_dataset.csv",
```

```

2: "pet_store_dataset.csv",
3: "electronic_store_dataset.csv",
4: "sportswear_store_dataset.csv",
5: "gardening_dataset.csv"
}
dataset = None
print("Welcome to CS634 Data Mining Mid Term Project")
print("Please select the store you want to work on:")
print("1. General Store")
print("2. Pet Store")
print("3. Electronic Store")
print("4. Sportswear Store")
print("5. Gardening Store")

# User input
while not dataset:
    dataset = input()
    # Checking for valid input
    if not dataset.isnumeric():
        print("Enter a valid number from 1 to 5, inclusive")
        dataset = None
        continue
    dataset = int(dataset)
    # Checking for valid input
    if dataset < 1 or dataset > 5:
        dataset = None
        print("Please enter a valid number from 1 to 5, inclusive.")
    else:
        # valid input
        print(f"You have selected the store containing the dataset_
↪{datasets[dataset]}")

# load the dataset
filename = f"dataset/{datasets.get(dataset)}"
df = pd.read_csv(filename)

```

Welcome to CS634 Data Mining Mid Term Project

Please select the store you want to work on:

1. General Store
2. Pet Store
3. Electronic Store
4. Sportswear Store
5. Gardening Store

1

You have selected the store containing the dataset general_store_dataset.csv

1.11.3 Minimum Support and Minimum Confidence

Defining the minimum support and confidence by taking user input. - We will take an input from the user about the Support and Confidence percent that he wants - We then multiply the minimum support by the number of transactions in the dataset which gives us the minimum frequency of an item which the items in the itemset need to satisfy, to be considered as a frequent item or an itemset.

```
[353]: MIN_SUPPORT = None
MIN_CONFIDENCE = None

# Taking input for Minimum Support
print("Plese enter the Minimum Support in % that you want (1 to 100,␣
↪inclusive)")
while not MIN_SUPPORT:
    MIN_SUPPORT = input()
    if not MIN_SUPPORT.isnumeric():
        print("Please enter a valid number from 1 to 100, inclusive")
        MIN_SUPPORT = None
        continue
    MIN_SUPPORT = int(MIN_SUPPORT)
    if MIN_SUPPORT < 1 or MIN_SUPPORT > 100:
        MIN_SUPPORT = None
        print("Please enter a valid number from 1 to 100, inclusive.")
    else:
        MIN_SUPPORT/=100

# Taking input for Minimum Confidence
print("Plese enter the Minimum Confidence in % that you want (1 to 100,␣
↪inclusive)")
while not MIN_CONFIDENCE:
    MIN_CONFIDENCE = input()
    if not MIN_CONFIDENCE.isnumeric():
        print("Enter a valid number from 1 to 100, inclusive")
        MIN_CONFIDENCE = None
        continue
    MIN_CONFIDENCE = int(MIN_CONFIDENCE)
    if MIN_CONFIDENCE < 1 or MIN_CONFIDENCE > 100:
        MIN_CONFIDENCE = None
        print("Please enter a valid number from 1 to 100, inclusive.")
    else:
        MIN_CONFIDENCE/=100

# Setting the minimum frequency that an itemset should have
MIN_SUPPORT_COUNT = df.shape[0]*MIN_SUPPORT
```

Plese enter the Minimum Support in % that you want (1 to 100, inclusive)

100

Please enter the Minimum Confidence in % that you want (1 to 100, inclusive)

100

1.11.4 Brute Force Function for Apriori Algorithm

The Apriori algorithm finds frequent itemsets by progressively generating larger item combinations. I used a brute-force method for itemset generation, iterating from 1-itemsets to k-itemsets. We implement the apriori algorithm. It takes the dataframe and the minimum support as argument and calculates the number of frequent itemsets and stores in the array `frequent_item_sets`. It follows the following steps: 1. First we calculate the frequent_item_sets for $k = 1$ and store them in frequent_item_sets while maintaining the value of `li` which is our intermediate set. 2. We filter all the itemsets which do not satisfy our criteria of minimum support. 3. This way we keep only the itemsets that match our criteria and then calculate the Frequent Itemsets for all the succeeding values of k . 4. This way we are able to dynamically generate all the Frequent Itemsets for all the values of k starting from $k = 2$. At the end we just print all the Frequent Itemset found at each step and print their corresponding support.

Keeping a count of the rules found by each of our methods to evaluate and compare later

```
[354]: brute_force_rules_count = 0
brute_force_fp_count = 0
library_rules_count = 0
library_fp_count = 0
library_fptree_count = 0
```

```
[355]: def brute_force(df, min_support):

    global brute_force_fp_count
    k = 1
    c1 = {}
    # itemsets which are frequent
    frequent_item_sets = []

    # keeping a record of the frequent itemsets along with their support
    support_counts = {}

    # generate candidate itemset for k = 1
    for i in range(len(df['ITEM_SET'].values)):
        for item in df['ITEM_SET'].values[i].split(','):
            c1[item] = c1.get(item, 0) + 1

    # generate intermediate itemset for k = 1 which satisfies the Minimum
    ↪ support criteria
    li = [key for key, value in c1.items() if value >= min_support]
    frequent_item_sets += li

    # Print the table c1
    print("Table C1")
```

```

print()
print("Itemset | Support")
for key, value in c1.items():
    print(f"{key} = {value}")
print()
print()

# Print the table l1
print("Table L1")
print()
print("Itemset | Support")
for item in li:
    print(f"{item} = {c1[item]}")

print()
print()

# keeping a track of frequent itemsets and their respective supports
for item in li:
    support_counts[(item,)] = c1[item]

k = 2
rows = df['ITEM_SET'].values

# proceeding ahead from k = 2
while True:
    # Combination
    comb = combinations(li, k)
    ci = {}
    # Count combination
    for item in comb:
        for row in rows:
            add = True
            for i in range(k):
                if isinstance(item[i], tuple):
                    # if any item is not in the row, then it is not a
                    ↪ frequent_itemset
                    if item[i][0] not in row or item[i][1] not in row:
                        add = False
                        break
            elif item[i] not in row:
                add = False
                break
            if add:
                if ci.get(item):
                    ci[item] += 1
                else:

```



```

        ci[item] = 1

    li = []
    print(f"Table C{k}")
    print()
    print(f"Itemset | Support")

    for key, value in ci.items():
        print(f"{key} = {value}")
    for key, value in ci.items():
        if value >= min_support:
            li.append(key)

    print()
    print()
    print(f"Table L{k}")
    print()
    print(f"Itemset | Support")
    for item in li:
        print(f"{item} = {ci[item]}")
    print()
    print()

    # if the intermediate itemset is empty that means no further itemsets
    ↪are found and we exit
    if len(li) == 0:
        break
    else:
        frequent_item_sets += li
        support_counts.update({item: ci[item] for item in li})
        li = set([j for i in li for j in i])
        k+=1

    # count of number of items in the frequen itemsets
    brute_force_fp_count = len(frequent_item_sets)

    print(f"The total number of frequent itemsets are {brute_force_fp_count}")
    return frequent_item_sets, support_counts

```

1.11.5 Candidate Set Generation Function

We implement the function `generate_candidate_set` to create intermediate candidate set by permutating the items in the frequent itemsets. In this way we can generate all the permutations of the frequent itemsets for association rule mining.

```
[356]: def generate_candidat_set(frequent_item_sets):
    # Generate permutation for each itemset in the frequent itemsets array
    candidate_set = [perm for item in frequent_item_sets if isinstance(item,
    ↪tuple) for perm in permutations(item)]
    return candidate_set
```

1.11.6 Association Rule Mining Function

Implementing the function to mine Association Rules, provided the frequent itemsets minimum confidence. Here we first taken the dataframe, frequent itemsets and minimum confidence as arguments to the function. Then we follow the following steps to find the confidence of each rule: 1. Calculate the number of transactions containing Antecedent and Consequent (consider it as part1) 2. Calculate the number of transactions containing only Antecedent (consider it as part2) 3. Calculate the confidence of a Frequent Itemset by dividing the number of transactions containing both (part1) by the number of transactions containing only Antecedent (part2) 4. Keep only the transactions which satisfy the criteria of minimum confidence for each transaction and filter the rest 5. Print the transactions which satisfy our criteria. These are the association rules.

```
[357]: def association_rule_mining(df, frequent_itemsets, min_confidence):

    # keeping a count of the number of association rules generated by brute_
    ↪force
    global brute_force_rules_count

    rows = df['ITEM_SET'].values

    association_rules = {}

    # traversing through all the itemsets from the list of frequent itemsets
    for itemset in frequent_itemsets:
        if isinstance(itemset, tuple) and len(itemset) > 1:
            for k in range(1, len(itemset)):
                antecedents = combinations(itemset, k)

                for antecedent in antecedents:
                    # if X -> then Y, here X is antecedent and Y is consequent
                    antecedent = set(antecedent)
                    consequent = set(itemset) - antecedent

                    antecedent_count = 0
                    consequent_count = 0

                    # we traverse through all the rows and check if the values_
                    ↪are present or not
                    for row in rows:
                        transaction = set(row.split(','))
```

```

# if antecedent is present increment the antecedent_
↪count
    if antecedent.issubset(transaction):
        antecedent_count += 1

# if the consequent is present increment the_
↪consequent count
    if consequent.issubset(transaction):
        consequent_count += 1

# we check if antecedent is greater than 0, to avoid_
↪division by 0
    if antecedent_count > 0:

# calculate confidence by dividing the consequent_count_
↪by antecedent count
        confidence = consequent_count / antecedent_count

# Only store the rule if they are greater than the_
↪minimum threshold
        if confidence >= min_confidence:
            rule = (tuple(sorted(antecedent)),_
↪tuple(sorted(consequent)))
                    association_rules[rule] = confidence

print()
print("Final Association Rules by Brute Force Method")
print()
brute_force_rules_count = len(association_rules)
print(f"The total number of association rule formed are:_
↪{brute_force_rules_count}")
print()
i = 1
# Printing all the associatoin rules and their corresponding confidence_
↪values
for key, val in association_rules.items():
    if val >= min_confidence:
        key = list(key)
        consequent = key[-1]
        antecedent = key[:-1]
        print(f"Rule {i}")
        print(f"{antecedent} --> {consequent[0]}")
        print(f"Confidence = {val:.2f}")
        print()
        i += 1

```

1.11.7 Executing Brute Force

Executing the apriori algorithm, generating candidate sets and executing association_rule_mining function to find the time comparison between each function and find out Frequent Patterns and Association Rules from our Dataset.

```
[358]: # Frequent Itemsets Mining using Brute Force
start_time = time.time()
frequent_item_sets, support_counts = brute_force(df, MIN_SUPPORT_COUNT)
end_time = time.time()

# Time taken by brute force apriori algorithm to run
brute_force_apriori_time = end_time - start_time

# Generating candidate set by permutating the frequent itemsets
candidate_set = generate_candidate_set(frequent_item_sets)

# Association Rule Mining on our candidate set
start_time = time.time()
association_rules = association_rule_mining(df, candidate_set, MIN_CONFIDENCE)
end_time = time.time()

# Time taken by Brute force association rule mining to run
brute_force_associationrules_time = end_time - start_time
```

Table C1

Itemset	Support
Milk	112
Bread	127
Chips	103
Apples	75
Butter	103
Soda	102
Diapers	79
Eggs	88
Chicken	1

Table L1

Itemset	Support
---------	---------

Table C2

Itemset	Support
---------	---------

Table L2

Itemset | Support

The total number of frequent itemsets are 0

Final Association Rules by Brute Force Method

The total number of association rule formed are: 0

```
[359]: print(f"Brute Force Apriori algorithm time: {brute_force_apriori_time:.4f}\n↪seconds")
print()
print(f"Brute Force Association Rule Mining time:\n↪{brute_force_associationrules_time:.4f} seconds")
print()
print()
```

Brute Force Apriori algorithm time: 0.0010 seconds

Brute Force Association Rule Mining time: 0.0000 seconds

1.12 Frequent Pattern and Association Rule Mining using Python Libraries

```
[360]: from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules, fpgrowth
```

Read the dataset perform preprocessing

```
[361]: df = pd.read_csv(filename)
data = df['ITEM_SET'].apply(lambda x: x.split(',')).tolist()
```

```
[362]: te = TransactionEncoder()
encoded_data = te.fit(data).transform(data)
df_encoded = pd.DataFrame(encoded_data, columns=te.columns_)
```

1.12.1 Frequent Itemset Mining using mlxtend Apriori Algorithm

Implementing the function to print the Frequent Itemsets that are found from the Apriori Algorithm.

```
[363]: def print_frequent_itemsets_apriori(frequent_itemsets):
    global library_fp_count
    # Get the maximum size of the itemsets present
    max_level = frequent_itemsets['length'].max()
```

```

library_fp_count = frequent_itemsets.shape[0]
if isinstance(max_level, float):
    print("No frequent itemsets")
else:
    for level in range(1, max_level + 1):
        print(f"Table C{level}")
        print("\nItemset | Count")
        level_itemsets = frequent_itemsets[frequent_itemsets['length'] ==
↪level]

        for index, row in level_itemsets.iterrows():
            itemset = tuple(row['itemsets'])
            count = row['support'] * len(df_encoded)
            print(f"{itemset} = {int(count)}")
        print()

```

We time the start time and end time to determine the total time taken by the algorithm to run. Execute the apriori algorithm that we imported from mlxtend library and note the time taken

```

[364]: # Executing the apriori algorithm by MLXTend library and recording the time
start_time = time.time()
frequent_itemsets_apriori = apriori(df_encoded, min_support=MIN_SUPPORT,
↪use_colnames=True)
frequent_itemsets_apriori['length'] = frequent_itemsets_apriori['itemsets'].
↪apply(lambda x: len(x))
print_frequent_itemsets_apriori(frequent_itemsets_apriori)
end_time = time.time()

# storing the time
mlxtend_apriori_time = end_time-start_time

print()
print(f"Using mlxtend Apriori algorithm time: {mlxtend_apriori_time:.4f}
↪seconds")

```

No frequent itemsets

Using mlxtend Apriori algorithm time: 0.0024 seconds

1.12.2 Association Rule Mining using mlxtend Association Rule Algorithm

Implement the `print_association_rules` function to print all the association rules mined using the mlxtend library. Here we use the `association_rules` function from the mlxtend library to find the association rules, which are mined from the frequent item sets that we found earlier.

```

[365]: def print_association_rules(rules):
        global library_rules_count
        print("Final Association Rules by using the Library\n")
        library_rules_count = rules.shape[0]

```

```

for index, rule in rules.iterrows():
    antecedents = list(rule['antecedents'])
    consequents = list(rule['consequents'])[0]
    confidence = rule['confidence']
    print(f"Rule {index + 1}")
    print(f"{antecedents} --> {consequents}")
    print(f"Confidence = {confidence:.2f}\n")

```

Executing the `association_rules` function from `mlxtend` and noting our timetaken for finding the rules and printing them

```

[366]: if frequent_itemsets_apriori.shape[0] != 0:
        start_time = time.time()
        rules = association_rules(frequent_itemsets_apriori, metric="confidence",
        ↪min_threshold=MIN_CONFIDENCE)
        print_association_rules(rules)
        end_time = time.time()
        mlxtend_association_rules_time = end_time - start_time
        print(f"mlxtend Association Rule Mining time: {end_time - start_time:.4f},
        ↪seconds")
    else:
        mlxtend_association_rules_time = 0
        print("No association rules")

```

No association rules

1.13 Frequent Pattern Mining using mlxtend FP Growth Algorithm from mlxtend Library

Implement the print function for Frequent Item sets mined using the FP Tree algorithm from MLXTend Library

```

[367]: def print_fp_growth_itemsets(frequent_itemsets_fp):
        global library_fptree_count
        library_fptree_count = frequent_itemsets_fp.shape[0]
        print("\nFrequent Itemsets using FP-Growth:")
        print("Itemset | Support")
        library_fptree_count = frequent_itemsets_fp.shape[0]
        if library_fptree_count == 0:
            print("No frequent Itemsets found")
            return
        for index, row in frequent_itemsets_fp.iterrows():
            itemset = tuple(row['itemsets'])
            support = row['support']
            print(f"{itemset} = {support:.2f}")

```

Executing and finding frequent items using FP Tree algorithm

```
[368]: start_time = time.time()

frequent_itemsets_fp = fpgrowth(df_encoded, min_support=MIN_SUPPORT,
    ↪use_colnames=True)
print_fp_growth_itemsets(frequent_itemsets_fp)

end_time = time.time()

mlxtend_fptree_time = end_time-start_time

print(f"FP-Growth time: {end_time - start_time:.4f} seconds")
```

Frequent Itemsets using FP-Growth:

Itemset | Support

No frequent Itemsets found

FP-Growth time: 0.0018 seconds

Executing the `association_rules` function from `mlx_tend` and noting our timetaken for finding the rules and printing them

```
[369]: print()
print()
```

1.14 Evaluation and Comparison of our Algorithm

```
[370]: print()
print('-----')
print("Time taken by all the algorithms:")
print()
```

```
-----
Time taken by all the algorithms:
```

```
[371]: data = [
    ['Brute Force Apriori', brute_force_apriori_time],
    ['Brute Force Association Rule', brute_force_associationrules_time],
    ['MLXTend Apriori', mlxtend_apriori_time],
    ['MLXTend FP Growth', mlxtend_fptree_time],
    ['MLXTend Association Rule', mlxtend_association_rules_time],
]

time_df = pd.DataFrame(data, columns=['Algorithm', 'Time (seconds)'])
print(time_df)
```


	Algorithm	Time (seconds)
0	Brute Force Apriori	0.001000
1	Brute Force Association Rule	0.000000
2	MLXTend Apriori	0.002373
3	MLXTend FP Growth	0.001786
4	MLXTend Association Rule	0.002617

```
[372]: print()
print('-----')
print("Number of Frequent Itemsets Mined:")
print()
```

```
-----
Number of Frequent Itemsets Mined:
```

```
[373]: data = [
    ['Brute Force Apriori', brute_force_fp_count],
    ['MLXTend Apriori', library_fp_count],
    ['MLXTend FP Growth', library_fptree_count],
]

time_df = pd.DataFrame(data, columns=['Algorithm', 'Frequent Patterns Count'])
print(time_df)
```

	Algorithm	Frequent Patterns Count
0	Brute Force Apriori	0
1	MLXTend Apriori	0
2	MLXTend FP Growth	0

```
[374]: print()
print('-----')
print("Number of Association Rules Mined:")
print()
```

```
-----
Number of Association Rules Mined:
```

```
[375]: data = [
    ['Brute Force Association Rule Count', brute_force_rules_count],
    ['MLXTend Association Rule Count', library_rules_count],
]

time_df = pd.DataFrame(data, columns=['Algorithm', 'Rule Count'])
print(time_df)
```

	Algorithm	Rule Count
--	-----------	------------

0	Brute Force Association Rule Count	0
1	MLXTend Association Rule Count	0

1.15 Observations

1. We can see that we have successfully created Apriori algorithm and Association Rule Mining algorithm using Brute Force approach which yields correct output
2. We have also implemented the both the algorithms and FP-Growth algorithm using MLXTend library and validated our data.
3. The prebuilt library are optimized hence they can be seen performing better than our implemented brute force approach.
4. We have also implemented the FP-Growth algorithm using the MLXTend library and it can be observed that it works faster compared to Apriori algorithm even when both of them are optimized.

1.16 Conclusion

In conclusion, we can say that our project successfully demonstrates the concepts of Apriori Algorithm, Association Rule Mining and methods. We have successfully implemented the Apriori algorithm from scratch to mine Association Rules from different datasets. Our brute force method works exceptionally well while taking the user defined parameters from input. In this way we are able to reveal patterns from our dataset and mine association rules which can help in decision making in the industry. We can conclude from our observations that FP-Growth algorithm is faster and more efficient at Mining Frequent Patterns than Apriori algorithm.

1.17 Github Repository Link

jd849@njit.edu -> [Github Repository Link](#)

1.18 Screenshots of Implementation in Jupyter Notebook

The below image is a sample of our entire General Store dataset

```
[376]: Image("screenshots/10.png")
```

```
[376]:
```

TID	ITEM_SET
1	Milk,Bread,Chips,Apples
2	Milk,Bread,Butter,Soda
3	Apples,Bread,Butter,Soda
4	Butter,Bread,Chips,Soda
5	Diapers,Apples,Butter,Chips
6	Diapers,Milk,Bread,Butter
7	Butter,Apples,Chips,Milk
8	Apples,Eggs,Butter,Soda
9	Milk,Bread,Eggs,Chips
10	Apples,Butter,Milk,Chips
11	Apples,Bread,Chips,Soda

Prompt the user to choose the store for our dataset

[377]: `Image("screenshots/11.png")`

[377]:

```
[428]: datasets = {
    1: "general_store_dataset.csv",
    2: "pet_store_dataset.csv",
    3: "electronic_store_dataset.csv",
    4: "sportswear_store_dataset.csv",
    5: "gardening_dataset.csv"
}
dataset = None
print("Welcome to CS634 Data Mining Mid Term Project")
print("Please select the store you want to work on:")
print("1. General Store")
print("2. Pet Store")
print("3. Electronic Store")
print("4. Sportswear Store")
print("5. Gardening Store")

# User input
while not dataset:
    dataset = input()
    # Checking for valid input
    if not dataset.isnumeric():
        print("Enter a valid number from 1 to 5, inclusive")
        dataset = None
        continue
    dataset = int(dataset)
    # Checking for valid input
    if dataset < 1 or dataset > 5:
        dataset = None
        print("Please enter a valid number from 1 to 5, inclusive.")
    else:
        # valid input
        print(f"You have selected the store containing the dataset {datasets[dataset]}")

# Load the dataset
filename = f"{dataset}/{datasets.get(dataset)}"
df = pd.read_csv(filename)

Welcome to CS634 Data Mining Mid Term Project
Please select the store you want to work on:
1. General Store
2. Pet Store
3. Electronic Store
4. Sportswear Store
5. Gardening Store
1
You have selected the store containing the dataset general_store_dataset.csv
```

Keeping the count of rules and frequent patterns mined

```
[378]: Image("screenshots/13.png")
```

[378]:

```
[430]: brute_force_rules_count = 0
brute_force_fp_count = 0
library_rules_count = 0
library_fp_count = 0
library_fptree_count = 0
```

Prompt the user for input of Minimum Support Percentage and Minimum Confidence Percentage

```
[379]: Image("screenshots/12.png")
```

[379]:

```
[429]: MIN_SUPPORT = None
MIN_CONFIDENCE = None

# Taking input for Minimum Support
print("Plese enter the Minimum Support in % that you want (1 to 100, inclusive)")
while not MIN_SUPPORT:
    MIN_SUPPORT = input()
    if not MIN_SUPPORT.isnumeric():
        print("Please enter a valid number from 1 to 100, inclusive")
        MIN_SUPPORT = None
        continue
    MIN_SUPPORT = int(MIN_SUPPORT)
    if MIN_SUPPORT < 1 or MIN_SUPPORT > 100:
        MIN_SUPPORT = None
        print("Please enter a valid number from 1 to 100, inclusive.")
    else:
        MIN_SUPPORT/=100

# Taking input for Minimum Confidence
print("Plese enter the Minimum Confidence in % that you want (1 to 100, inclusive)")
while not MIN_CONFIDENCE:
    MIN_CONFIDENCE = input()
    if not MIN_CONFIDENCE.isnumeric():
        print("Enter a valid number from 1 to 100, inclusive")
        MIN_CONFIDENCE = None
        continue
    MIN_CONFIDENCE = int(MIN_CONFIDENCE)
    if MIN_CONFIDENCE < 1 or MIN_CONFIDENCE > 100:
        MIN_CONFIDENCE = None
        print("Please enter a valid number from 1 to 100, inclusive.")
    else:
        MIN_CONFIDENCE/=100

# Setting the minimum frequency that an itemset should have
MIN_SUPPORT_COUNT = df.shape[0]*MIN_SUPPORT

Plese enter the Minimum Support in % that you want (1 to 100, inclusive)
10
Plese enter the Minimum Confidence in % that you want (1 to 100, inclusive)
50
```

Defining the brute force apriori algorithm

```
[380]: Image("screenshots/14.png")
```

[380]:

```
[431]: def brute_force(df, min_support):

    global brute_force_fp_count
    k = 1
    c1 = {}
    # itemsets which are frequent
    frequent_item_sets = []

    # keeping a record of the frequent itemsets along with their support
    support_counts = {}

    # generate candidate itemset for k = 1
    for i in range(len(df['ITEM_SET'].values)):
        for item in df['ITEM_SET'].values[i].split(','):
            c1[item] = c1.get(item, 0) + 1

    # generate intermediate itemset for k = 1 which satisfies the Minimum support criteria
    li = [key for key, value in c1.items() if value >= min_support]
    frequent_item_sets += li

    # Print the table c1
    print("Table C1")
    print()
    print("Itemset | Support")
    for key, value in c1.items():
        print(f"{key} = {value}")
    print()
    print()

    # Print the table l1
    print("Table L1")
    print()
    print("Itemset | Support")
    for item in li:
        print(f"{item} = {c1[item]}")

    print()
    print()

    # keeping a track of frequent itemsets and their respective supports
    for item in li:
        support_counts[{item,}] = c1[item]

    k = 2
    rows = df['ITEM_SET'].values

    # proceeding ahead from k = 2
    while True:
        # Combination
        comb = combinations(li, k)
        ci = {}
        # Count combination
        for item in comb:
            for row in rows:
                add = True
                for i in range(k):
                    if isinstance(item[i], tuple):
                        # if any item is not in the row, then it is not a frequent_itemset
                        if item[i][0] not in row or item[i][1] not in row:
```

Continuation of the above brute force code

```
[381]: Image("screenshots/15.png")
```

```
[381]:
```

```

        if isinstance(item[i], tuple):
            # if any item is not in the row, then it is not a frequent_itemset
            if item[i][0] not in row or item[i][1] not in row:
                add = False
                break
            elif item[i] not in row:
                add = False
                break
        if add:
            if ci.get(item):
                ci[item] += 1
            else:
                ci[item] = 1

    li = []
    print(f"Table C{k}")
    print()
    print(f"Itemset | Support")

    for key, value in ci.items():
        print(f"{key} = {value}")
    for key, value in ci.items():
        if value >= min_support:
            li.append(key)

    print()
    print()
    print(f"Table L{k}")
    print()
    print(f"Itemset | Support")
    for item in li:
        print(f"{item} = {ci[item]}")
    print()
    print()

    # if the intermediate itemset is empty that means no further itemsets are found and we exit
    if len(li) == 0:
        break
    else:
        frequent_item_sets += li
        support_counts.update({item: ci[item] for item in li})
        li = set([j for i in li for j in i])
        k+=1

    # count of number of items in the frequen itemsets
    brute_force_fp_count = len(frequent_item_sets)

    print(f"The total number of frequent itemsets are {brute_force_fp_count}")
    return frequent_item_sets, support_counts

```

Function to generate candidate set which consists of permuted frequent itemsets

[382]: `Image("screenshots/16.png")`

[382]:

```

[432]: def generate_candidat_set(frequent_item_sets):
        # Generate permutation for each itemset in the frequent itemsets array
        candidate_set = [perm for item in frequent_item_sets if isinstance(item, tuple) for perm in permutations(item)]
        return candidate_set

```

Defining the association_rule_mining algorithm using brute force

[383]: `Image("screenshots/17.png")`

[383]:

```
[433]: def association_rule_mining(df, frequent_itemsets, min_confidence):

    # keeping a count of the number of association rules generated by brute force
    global brute_force_rules_count

    rows = df['ITEM_SET'].values

    association_rules = {}

    # traversing through all the itemsets from the list of frequent itemsets
    for itemset in frequent_itemsets:
        if isinstance(itemset, tuple) and len(itemset) > 1:
            for k in range(1, len(itemset)):
                antecedents = combinations(itemset, k)

                for antecedent in antecedents:
                    # if X -> then Y, here X is antecedent and Y is consequent
                    antecedent = set(antecedent)
                    consequent = set(itemset) - antecedent

                    antecedent_count = 0
                    consequent_count = 0

                    # we traverse through all the rows and check if the values are present or not
                    for row in rows:
                        transaction = set(row.split(','))

                        # if antecedent is present increment the antecedent count
                        if antecedent.issubset(transaction):
                            antecedent_count += 1

                        # if the consequent is present increment the consequent count
                        if consequent.issubset(transaction):
                            consequent_count += 1

                    # we check if antecedent is greater than 0, to avoid division by 0
                    if antecedent_count > 0:

                        # calculate confidence by dividing the consequent_count by antecedent count
                        confidence = consequent_count / antecedent_count

                        # Only store the rule if they are greater than the minimum threshold
                        if confidence >= min_confidence:
                            rule = (tuple(sorted(antecedent)), tuple(sorted(consequent)))
                            association_rules[rule] = confidence

    print()
    print("Final Association Rules by Brute Force Method")
    print()
    brute_force_rules_count = len(association_rules)
    print(f"The total number of association rule formed are: {brute_force_rules_count}")
    print()
    i = 1
    # Printing all the associatoin rules and their corresponding confidence values
    for key, val in association_rules.items():
        if val >= min_confidence:
            key = list(key)
            consequent = key[-1]
            antecedent = key[:-1]
            print(f"Rule {i}")
            print(f"{antecedent} --> {consequent[0]}")

            rule = (tuple(sorted(antecedent)), tuple(sorted(consequent)))
            association_rules[rule] = confidence

    print()
    print("Final Association Rules by Brute Force Method")
    print()
    brute_force_rules_count = len(association_rules)
    print(f"The total number of association rule formed are: {brute_force_rules_count}")
    print()
    i = 1
    # Printing all the associatoin rules and their corresponding confidence values
    for key, val in association_rules.items():
        if val >= min_confidence:
            key = list(key)
            consequent = key[-1]
            antecedent = key[:-1]
            print(f"Rule {i}")
            print(f"{antecedent} --> {consequent[0]}")
            print(f"Confidence = {val:.2f}")
            print()
            i += 1
```

[384]: # Continuation of the above
Image("screenshots/18.png")

```
[384]: rule = (tuple(sorted(antecedent)), tuple(sorted(consequent)))
        association_rules[rule] = confidence

    print()
    print("Final Association Rules by Brute Force Method")
    print()
    brute_force_rules_count = len(association_rules)
    print(f"The total number of association rule formed are: {brute_force_rules_count}")
    print()
    i = 1
    # Printing all the associatoin rules and their corresponding confidence values
    for key, val in association_rules.items():
        if val >= min_confidence:
            key = list(key)
            consequent = key[-1]
            antecedent = key[:-1]
            print(f"Rule {i}")
            print(f"{antecedent} --> {consequent[0]}")
            print(f"Confidence = {val:.2f}")
            print()
            i += 1
```

1.18.1 Executing our brute force algorithm and recording the time

[385]: `Image("screenshots/19.png")`

[385]:

```
[434]: # Frequent Itemsets Mining using Brute Force
start_time = time.time()
frequent_item_sets, support_counts = brute_force(df, MIN_SUPPORT_COUNT)
end_time = time.time()

# Time taken by brute force apriori algorithm to run
brute_force_apriori_time = end_time - start_time

# Generating candidate set by permutating the frequent itemsets
candidate_set = generate_candidate_set(frequent_item_sets)

# Association Rule Mining on our candidate set
start_time = time.time()
association_rules = association_rule_mining(df, candidate_set, MIN_CONFIDENCE)
end_time = time.time()

# Time taken by Brute force association rule mining to run
brute_force_associationrules_time = end_time - start_time
```

Table C1

Itemset	Support
Milk = 112	
Bread = 127	
Chips = 103	
Apples = 75	
Butter = 103	
Soda = 102	
Diapers = 79	
Eggs = 88	
Chicken = 1	

Table L1

Itemset	Support
Milk = 112	

1.18.2 Executing prebuilt library for association rule mining and frequent pattern mining

Defining the function for printing Apriori Algorithm and executing it

[386]: `Image("screenshots/20.png")`

[386]:

Frequent Pattern and Association Rule Mining using Python Libraries

```
[436]: from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules, fpgrowth
```

Read the dataset perform preprocessing

```
[437]: df = pd.read_csv(filename)
data = df['ITEM_SET'].apply(lambda x: x.split(',')).tolist()
```

```
[438]: te = TransactionEncoder()
encoded_data = te.fit(data).transform(data)
df_encoded = pd.DataFrame(encoded_data, columns=te.columns_)
```

Frequent Itemset Mining using mlxtend Apriori Algorithm

Implementing the function to print the Frequent Itemsets that are found from the Apriori Algorithm.

```
[439]: def print_frequent_itemsets_apriori(frequent_itemsets):
    global library_fp_count
    # Get the maximum size of the itemsets present
    max_level = frequent_itemsets['length'].max()
    library_fp_count = frequent_itemsets.shape[0]

    # print the frequent itemset and their count
    for level in range(1, max_level + 1):
        print(f"Table C{level}")
        print("\nItemset | Count")
        level_itemsets = frequent_itemsets[frequent_itemsets['length'] == level]
        for index, row in level_itemsets.iterrows():
            itemset = tuple(row['itemsets'])
            count = row['support'] * len(df_encoded)
            print(f"{itemset} = {int(count)}")
        print()
```

We time the start time and end time to determine the total time taken by the algorithm to run.

Execute the apriori algorithm that we imported from mlxtend library and note the time taken

```
[440]: # Executing the apriori algorithm by MLXTend library and recording the time
start_time = time.time()
frequent_itemsets_apriori = apriori(df_encoded, min_support=MIN_SUPPORT, use_colnames=True)
frequent_itemsets_apriori['length'] = frequent_itemsets_apriori['itemsets'].apply(lambda x: len(x))
print_frequent_itemsets_apriori(frequent_itemsets_apriori)
end_time = time.time()

# storing the time
mlxtend_apriori_time = end_time - start_time

print()
print(f"Using mlxtend Apriori algorithm time: {mlxtend_apriori_time:.4f} seconds")
```

Table C1

Itemset	Count
('Apples',)	75
('Bread',)	127
('Butter',)	103
('Chips',)	103
('Diapers',)	79
('Eggs',)	88
('Milk',)	112
('Soda',)	102

Implementing the function to print association rules using mlxtend library and executing it

```
[387]: Image("screenshots/21.png")
```

[387]:

Association Rule Mining using mlxtend Association Rule Algorithm

Implement the `print_association_rules` function to print all the association rules mined using the mlxtend library. Here we use the `association_rules` function from the mlxtend library to find the association rules, which are mined from the frequent item sets that we found earlier.

```
[441]: def print_association_rules(rules):
        global library_rules_count
        print("Final Association Rules by using the Library\n")
        library_rules_count = rules.shape[0]
        for index, rule in rules.iterrows():
            antecedents = list(rule['antecedents'])
            consequents = list(rule['consequents'])
            confidence = rule['confidence']
            print(f"Rule {index + 1}")
            print(f"{antecedents} --> {consequents}")
            print(f"Confidence = {confidence:.2f}\n")
```

Executing the `association_rules` function from `mlxtend` and noting our timetaken for finding the rules and printing them

```
[442]: start_time = time.time()

rules = association_rules(frequent_itemsets_apriori, metric="confidence", min_threshold=MIN_CONFIDENCE)
print_association_rules(rules)

end_time = time.time()

mlxtend_association_rules_time = end_time - start_time
print(f"mlxtend Association Rule Mining time: {end_time - start_time:.4f} seconds")

Final Association Rules by using the Library

Rule 1
['Apples'] --> Chips
Confidence = 0.56

Rule 2
['Bread'] --> Butter
Confidence = 0.50

Rule 3
['Butter'] --> Bread
Confidence = 0.62
```

Implementing the function to print frequent itemsets using FP-Tree algorithm and then running it

[388]: `Image("screenshots/22.png")`

[388]:

Frequent Pattern Mining using mlxtend FP Growth Algorithm from mlxtend Library

Implement the print function for Frequent Item sets mined using the FP Tree algorithm from MLXTend Library

```
[443]: def print_fp_growth_itemsets(frequent_itemsets_fp):
        global library_fptree_count
        library_fptree_count = frequent_itemsets_fp.shape[0]
        print("\nFrequent Itemsets using FP-Growth:")
        print("Itemset | Support")
        library_fptree_count = frequent_itemsets_fp.shape[0]
        for index, row in frequent_itemsets_fp.iterrows():
            itemset = tuple(row['itemsets'])
            support = row['support']
            print(f"{itemset} = {support:.2f}")
```

Executing and finding frequent items using FP Tree algorithm

```
[444]: start_time = time.time()

frequent_itemsets_fp = fpgrowth(df_encoded, min_support=MIN_SUPPORT, use_colnames=True)
print_fp_growth_itemsets(frequent_itemsets_fp)

end_time = time.time()

mlxtend_fptree_time = end_time - start_time

print(f"FP-Growth time: {end_time - start_time:.4f} seconds")

Frequent Itemsets using FP-Growth:
Itemset | Support
('Bread',) = 0.64
('Milk',) = 0.56
('Chips',) = 0.52
('Apples',) = 0.38
.. .. .
```

1.18.3 Performance evaluation and comparison

[389]: `Image("screenshots/23.png")`

[389]:

▼ Evaluation and Comparison of our Algorithm

```
[446]: print()
print('-----')
print("Time taken by all the algorithms:")
print()
```

Time taken by all the algorithms:

```
[447]: data = [
    ['Brute Force Apriori', brute_force_apriori_time],
    ['Brute Force Association Rule', brute_force_associationrules_time],
    ['MLXTend Apriori', mlxtend_apriori_time],
    ['MLXTend FP Growth', mlxtend_fptree_time],
    ['MLXTend Association Rule', mlxtend_association_rules_time],
]

time_df = pd.DataFrame(data, columns=['Algorithm', 'Time (seconds)'])
print(time_df)
```

	Algorithm	Time (seconds)
0	Brute Force Apriori	0.010994
1	Brute Force Association Rule	0.036136
2	MLXTend Apriori	0.012779
3	MLXTend FP Growth	0.002898
4	MLXTend Association Rule	0.003664

```
[448]: print()
print('-----')
print("Number of Frequent Patterns Mined:")
print()
```

Number of Frequent Patterns Mined:

```
[449]: data = [
    ['Brute Force Apriori', brute_force_fp_count],
    ['MLXTend Apriori', library_fp_count],
    ['MLXTend FP Growth', library_fptree_count],
]

time_df = pd.DataFrame(data, columns=['Algorithm', 'Frequent Patterns Count'])
print(time_df)
```

	Algorithm	Frequent Patterns Count
0	Brute Force Apriori	41
1	MLXTend Apriori	41
2	MLXTend FP Growth	41

[390]: `Image("screenshots/24.png")`

[390]:

```
[450]: print()
print('-----')
print("Number of Association Rules Mined:")
print()

-----
Number of Association Rules Mined:

[451]: data = [
    ['Brute Force Association Rule Count', brute_force_rules_count],
    ['MLXTend Association Rule Count', library_rules_count],
]

time_df = pd.DataFrame(data, columns=['Algorithm', 'Rule Count'])
print(time_df)
```

	Algorithm	Rule Count
0	Brute Force Association Rule Count	19
1	MLXTend Association Rule Count	19

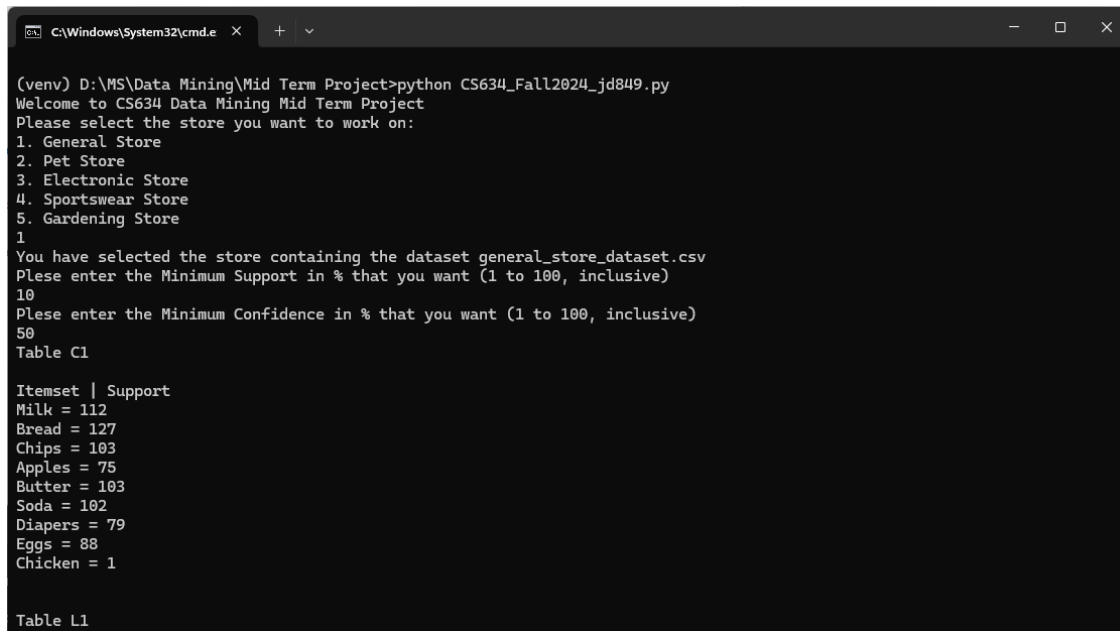
1.19 Screenshots of Execution of Python file

1.19.1 Running on the dataset General Store

- Minimum support = 10%
- Minimum confidence = 50%

[391]: Image("screenshots/3.png")

[391]:



```
C:\Windows\System32\cmd.exe X + v

(env) D:\MS\Data Mining\Mid Term Project>python CS634_Fall2024_jd849.py
Welcome to CS634 Data Mining Mid Term Project
Please select the store you want to work on:
1. General Store
2. Pet Store
3. Electronic Store
4. Sportswear Store
5. Gardening Store
1
You have selected the store containing the dataset general_store_dataset.csv
Please enter the Minimum Support in % that you want (1 to 100, inclusive)
10
Please enter the Minimum Confidence in % that you want (1 to 100, inclusive)
50
Table C1

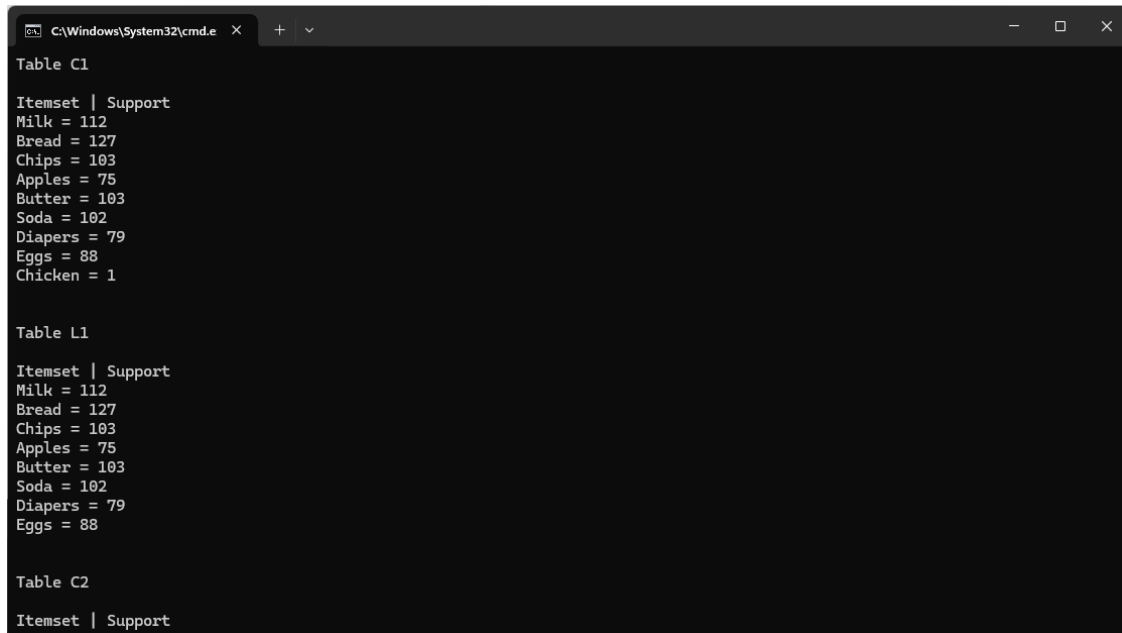
Itemset | Support
Milk = 112
Bread = 127
Chips = 103
Apples = 75
Butter = 103
Soda = 102
Diapers = 79
Eggs = 88
Chicken = 1

Table L1
```

Candidate set vs the frequently mined items

[392]: Image("screenshots/6.png")

[392]:



```
C:\Windows\System32\cmd.e x + v

Table C1

Itemset | Support
Milk = 112
Bread = 127
Chips = 103
Apples = 75
Butter = 103
Soda = 102
Diapers = 79
Eggs = 88
Chicken = 1

Table L1

Itemset | Support
Milk = 112
Bread = 127
Chips = 103
Apples = 75
Butter = 103
Soda = 102
Diapers = 79
Eggs = 88

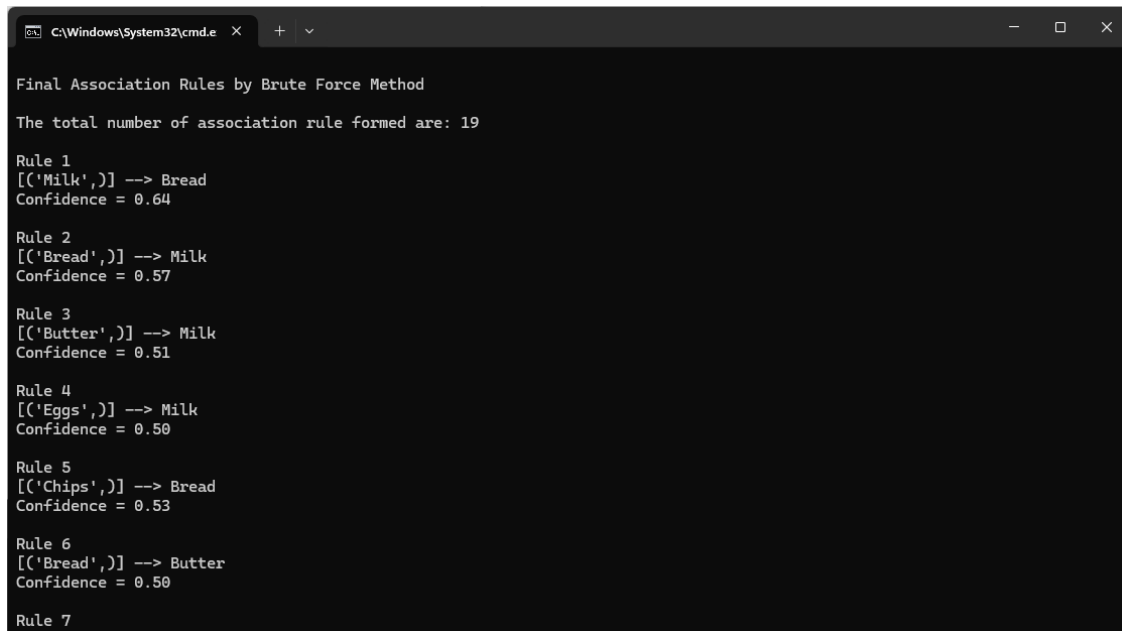
Table C2

Itemset | Support
```

Association rules generated by brute force method

[393]: Image("screenshots/7.png")

[393]:



```
C:\Windows\System32\cmd.e x + v

Final Association Rules by Brute Force Method

The total number of association rule formed are: 19

Rule 1
['Milk',)] --> Bread
Confidence = 0.64

Rule 2
['Bread',)] --> Milk
Confidence = 0.57

Rule 3
['Butter',)] --> Milk
Confidence = 0.51

Rule 4
['Eggs',)] --> Milk
Confidence = 0.50

Rule 5
['Chips',)] --> Bread
Confidence = 0.53

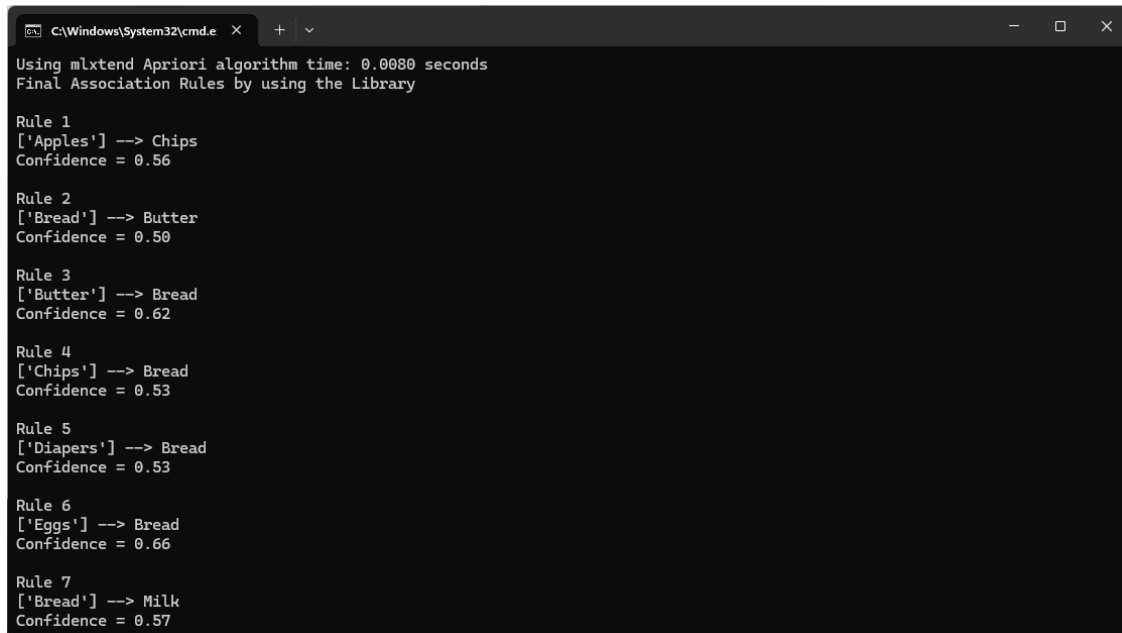
Rule 6
['Bread',)] --> Butter
Confidence = 0.50

Rule 7
```

Association rules generated by mlxtend library

[394]: Image("screenshots/8.png")

[394]:



```
C:\Windows\System32\cmd.exe
Using mlxtend Apriori algorithm time: 0.0080 seconds
Final Association Rules by using the Library

Rule 1
['Apples'] --> Chips
Confidence = 0.56

Rule 2
['Bread'] --> Butter
Confidence = 0.50

Rule 3
['Butter'] --> Bread
Confidence = 0.62

Rule 4
['Chips'] --> Bread
Confidence = 0.53

Rule 5
['Diapers'] --> Bread
Confidence = 0.53

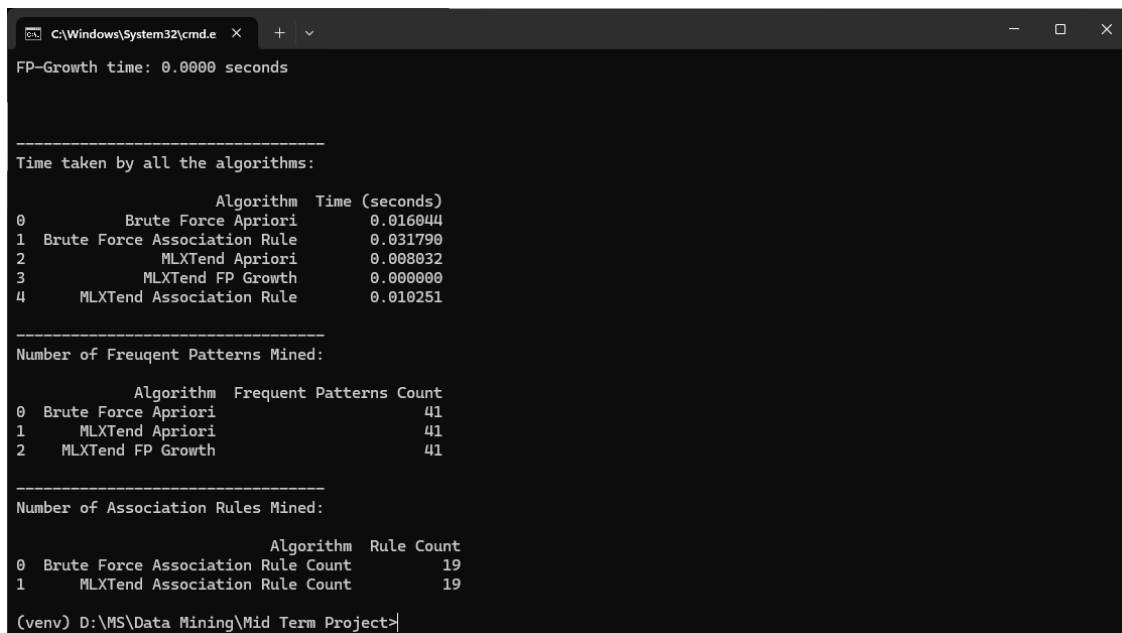
Rule 6
['Eggs'] --> Bread
Confidence = 0.66

Rule 7
['Bread'] --> Milk
Confidence = 0.57
```

Comparison between brute force and mlxtend library

[395]: Image("screenshots/9.png")

[395]:



```
C:\Windows\System32\cmd.exe
FP-Growth time: 0.0000 seconds

-----
Time taken by all the algorithms:

Algorithm      Time (seconds)
0  Brute Force Apriori      0.016044
1  Brute Force Association Rule 0.031790
2  MLXTend Apriori          0.008032
3  MLXTend FP Growth        0.000000
4  MLXTend Association Rule  0.010251

-----
Number of Frequent Patterns Mined:

Algorithm      Frequent Patterns Count
0  Brute Force Apriori      41
1  MLXTend Apriori          41
2  MLXTend FP Growth        41

-----
Number of Association Rules Mined:

Algorithm      Rule Count
0  Brute Force Association Rule Count 19
1  MLXTend Association Rule Count      19

(venv) D:\MS\Data Mining\Mid Term Project>|
```

1.20 *Output*

The output of executions show us that the association rules formed by Brute Force Method and that by using mlxtend library are the same. When compared the count of Frequent Itemsets and that of the Association Rules is the same for both the methods. This way we upon comparison, we find that the association rules generated by brute force are verifiable by the ones generated by the built-in package.

1.21 *Other*

The source code (.py file) and data sets (.csv files) will be attached to the zip file. *Link to GitHub repository* https://github.com/jaysheeldodianjit/CS634_MidtermProject