Joseph Marquez

jvm2@njit.edu

Professor Yasser Abduallah

CS634 104

Github Link: https://github.com/jvm2NJIT/CS634-Midterm-Project

Midterm Project Report

File Overview

screenshots folder: Screenshots of code that are used in this report

Five Database#.csv files: The databases that were created for this project

part1.ipynb: My code for the creation of the 5 databases

part2.ipynb: My code for brute force

part3.ipynb: My code comparing

MidtermProject.py: The actual executable project for you to run

MidtermReport.pdf: Report of the project

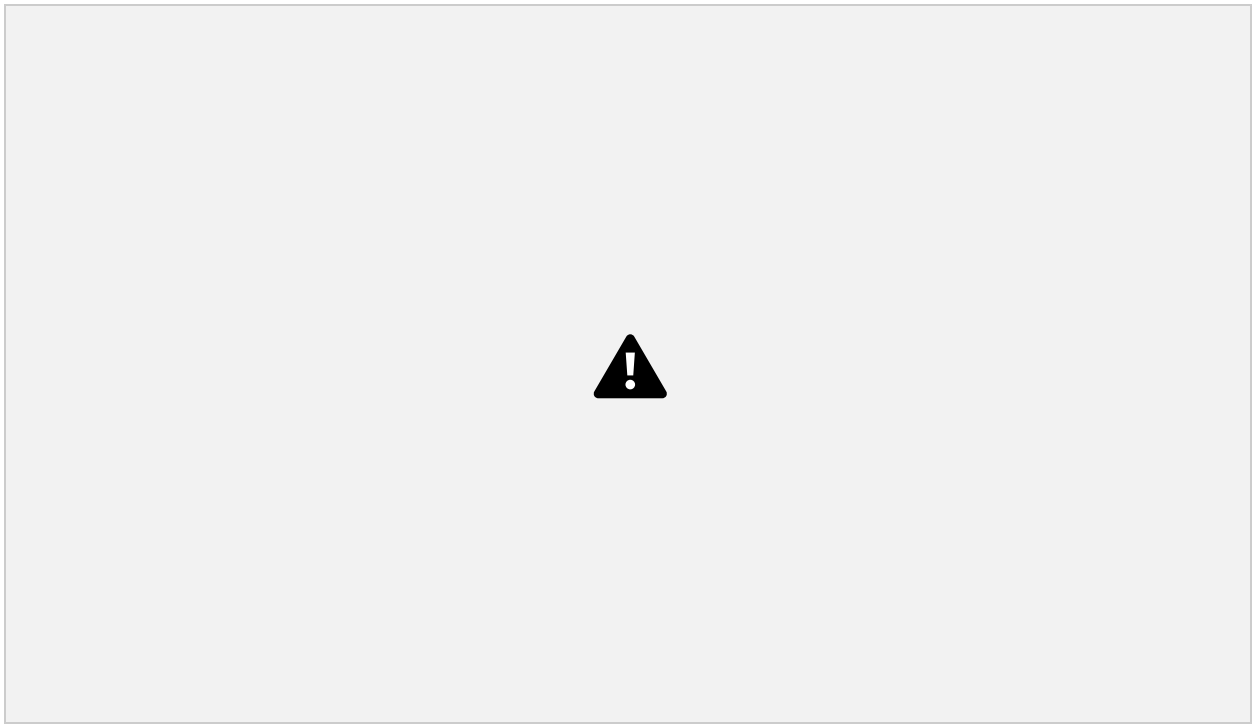
How to Run

1. Extract files
2. Run MidtermProject.py

Part 1

I decided to create my five databases myself. Every transaction in my five databases consists of ten common grocery store items. The generation of transactions was entirely random to ensure there were no biases in future parts of the project.

Here is what Database1.csv looks like:



Code to generate database:

```
def generate_database(items, filename):
    '''
    Creates database as pandas DataFrame and writes it to csv file
    Parameters:
        items (list): List of items that all customers are buying from
        filename (str): name of csv file you want to write to
    Returns:
        df (DataFrame): database of 20 transactions
    '''
    dataset = []
    for i in range(0, 20):
        j = random.randint(1, 10)
        transaction = random.sample(items, j)
        dataset.append(transaction)

    te = TransactionEncoder()
    te_ary = te.fit(dataset).transform(dataset)
    df = pd.DataFrame(te_ary, columns=te.columns_)

    # dict = {}
    # for item in items:
    #     values = [random.choice([True, False]) for i in range(20)]
    #     dict.update({item: values})
    # df = pd.DataFrame(dict)
    df.to_csv(filename)
    return(df)
```

Part 2

I then created my brute force method, which loops through every possible transaction that could be made with the ten specific items and crosschecked them with the inputted database and inputted support to determine if it was a frequent itemset. It then extracted every possible association rule that could be created from the frequent itemsets, and checked if they were association rules with the inputted confidence level.

Code to find all subsets of a set of items:

```
def findsubsets(s, k):
    '''
    Returns all k-itemsets from list s
    Parameters:
        s (list): List of items
        k (int): Size of subsets you wish to find
    Returns:
        subsets (list): List of k-subsets
    '''
    return list(itertools.combinations(s, k))
```

Code to find all possible associations from a list of items:

```
def generate_possible_rules(freq):
    '''
    Returns possible association rules from an itemset
    Parameters:
        freq (tuple): Frequent itemset
    Returns:
        possible_rules (list): List of possible rules
    '''
    n = len(freq)
    possible_rules = []
    for i in range(1, n):
        for combo in itertools.combinations(freq, i):
            tuple1 = combo
            tuple2 = tuple(item for item in freq if item not in combo)
            possible_rules.append([tuple1, tuple2])
    return possible_rules
```

Code for Brute Force:

```
def brute_force(items, filename, support, confidence):
    '''
    Uses brute force method to generate frequent items and generate association rules
    Paramaters:
        items (list): Sorted list of items
        filename (str): Filename of database you want to read from
        support (float): Desired support level for finding frequent itemsets
        confidence (float): Desired support level for finding association rules
    Returns:
        [freq_itemsets, rules, sups, cons] (list): A list of frequent items, a list of
        association rules, and their corresponding support values and confidence values
    '''
    df = pd.read_csv(filename)
    freq_itemsets = []
    supports = {}
    sups = []
    cons = []

    for i in range (1, 11):
        i_sets = findsubsets(items, i)
        for subset in i_sets:
            freq_count = 0
            for ind in df.index:
                match = True
                for item in subset:
                    if not df[item][ind]:
                        match = False
                        break
                if match:
                    freq_count = freq_count + 1

            supports.update({subset: freq_count / 20})
            if freq_count / 20 >= support:
                freq_itemsets.append(subset)
                sups.append(freq_count / 20)

    rules = []
    for freq in freq_itemsets:
        possible_rules = generate_possible_rules(freq)
        for p in possible_rules:
            X = p[0]
            Y = p[1]
            X_and_Y = tuple(sorted(X+Y))
            if supports[X_and_Y] / supports[X] >= confidence:
                rules.append(p)
                cons.append(supports[X_and_Y] / supports[X])


    return([freq_itemsets, rules, sups, cons])
```

Results from Brute Force:

Frequent itemsets for Database1.csv with support 0.5 and confidence 0.75

```
result = brute_force(items, 'Database1.csv', 0.5, 0.75)
result[0]
```

```
[('Apple',),
 ('Banana',),
 ('Eggs',),
 ('Soap',),
 ('Water',),
 ('Yogurt',),
 ('Soap', 'Water')]
```

```
result[1]
```

```
[[('Soap',), ('Water',)], [('Water',), ('Soap',)]]
```

Part 3

I simply used this part to compare my results when using my brute force method with the existing implementation of the Apriori and FP Tree methods form the mlxtend package. Using Database1.csv, support=0.35, and confidence=0.4, I found my results to be the same for all three methods.

Apriori

```
[5]  from mlxtend.frequent_patterns import apriori, association_rules, fpgrowth
```

Apriori Frequent Itemsets

```
[6]  df1 = pd.read_csv('Database1.csv')
     df1.drop(columns=df1.columns[0], axis=1, inplace=True)
     a = apriori(df1, min_support=0.5, use_colnames=True)
     a
```

	support	itemsets
0	0.50	(Apple)
1	0.55	(Banana)
2	0.65	(Eggs)
3	0.60	(Soap)
4	0.65	(Water)
5	0.55	(Yogurt)
6	0.50	(Soap, Water)

Apriori Association Rules

```
[7]  rules = association_rules(a, metric='confidence', min_threshold=0.75)
     rules[['antecedents', 'consequents', 'confidence']]
```

	antecedents	consequents	confidence
0	(Soap)	(Water)	0.833333
1	(Water)	(Soap)	0.769231

FP Tree

FP Tree Frequent Itemsets

```
[8]  f = fpgrowth(df1, min_support=0.5, use_colnames=True)
     f
```

	support	itemsets
0	0.65	(Water)
1	0.65	(Eggs)
2	0.60	(Soap)
3	0.50	(Apple)
4	0.55	(Yogurt)
5	0.55	(Banana)
6	0.50	(Soap, Water)

FP Tree Association Rules

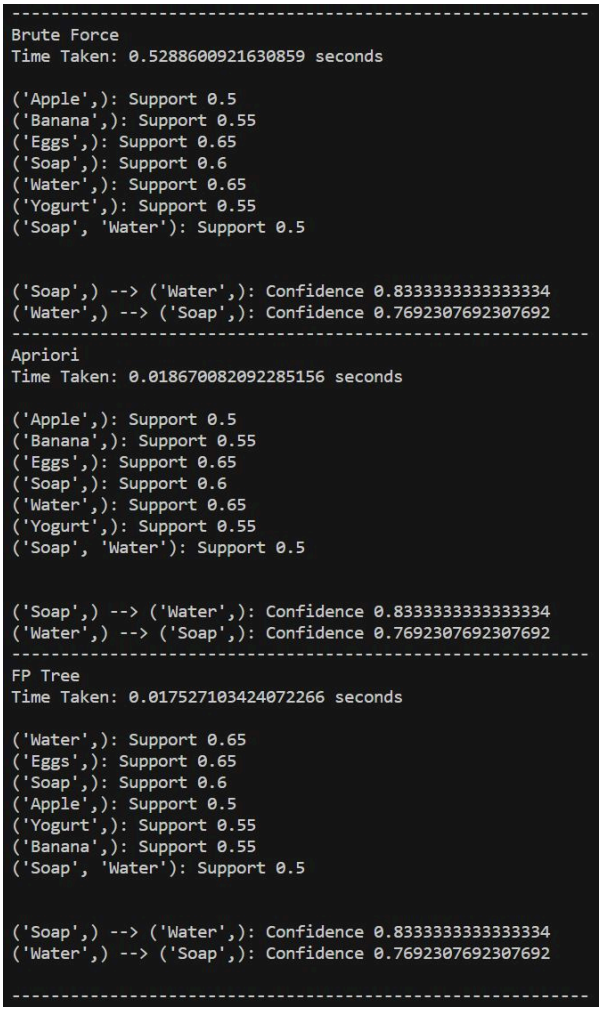
```
[9]  rules = association_rules(f, metric='confidence', min_threshold=0.75)
     rules[['antecedents', 'consequents', 'confidence']]
```

	antecedents	consequents	confidence
0	(Soap)	(Water)	0.833333
1	(Water)	(Soap)	0.769231

MidtermProject.py

Run this file in the console and input your desired database, support, and confidence. The program will output every frequent itemset and association rule according to your specifications, and then output the runtime of all three methods.

Here are some screenshots showing that the output for the three methods is the same:



```
------------------------------------------------------------
Brute Force
Time Taken: 0.5288600921630859 seconds

('Apple',): Support 0.5
('Banana',): Support 0.55
('Eggs',): Support 0.65
('Soap',): Support 0.6
('Water',): Support 0.65
('Yogurt',): Support 0.55
('Soap', 'Water'): Support 0.5


('Soap',) --> ('Water',): Confidence 0.8333333333333334
('Water',) --> ('Soap',): Confidence 0.7692307692307692
------------------------------------------------------------
Apriori
Time Taken: 0.018670082092285156 seconds

('Apple',): Support 0.5
('Banana',): Support 0.55
('Eggs',): Support 0.65
('Soap',): Support 0.6
('Water',): Support 0.65
('Yogurt',): Support 0.55
('Soap', 'Water'): Support 0.5


('Soap',) --> ('Water',): Confidence 0.8333333333333334
('Water',) --> ('Soap',): Confidence 0.7692307692307692
------------------------------------------------------------
FP Tree
Time Taken: 0.017527103424072266 seconds

('Water',): Support 0.65
('Eggs',): Support 0.65
('Soap',): Support 0.6
('Apple',): Support 0.5
('Yogurt',): Support 0.55
('Banana',): Support 0.55
('Soap', 'Water'): Support 0.5


('Soap',) --> ('Water',): Confidence 0.8333333333333334
('Water',) --> ('Soap',): Confidence 0.7692307692307692

------------------------------------------------------------
```

```
------------------------------------------------------------
Brute Force
Time Taken: 0.37562131881713867 seconds

('Eggs',): Support 0.7
('Milk',): Support 0.75
('Shampoo',): Support 0.65
('Soap',): Support 0.8
('Sugar',): Support 0.65
('Milk', 'Soap'): Support 0.65


('Milk',) --> ('Soap',): Confidence 0.8666666666666667
('Soap',) --> ('Milk',): Confidence 0.8125
------------------------------------------------------------
Apriori
Time Taken: 0.014174222946166992 seconds

('Eggs',): Support 0.7
('Milk',): Support 0.75
('Shampoo',): Support 0.65
('Soap',): Support 0.8
('Sugar',): Support 0.65
('Milk', 'Soap'): Support 0.65


('Milk',) --> ('Soap',): Confidence 0.8666666666666667
('Soap',) --> ('Milk',): Confidence 0.8125
------------------------------------------------------------
FP Tree
Time Taken: 0.008939266204833984 seconds

('Soap',): Support 0.8
('Eggs',): Support 0.7
('Sugar',): Support 0.65
('Milk',): Support 0.75
('Shampoo',): Support 0.65
('Milk', 'Soap'): Support 0.65


('Milk',) --> ('Soap',): Confidence 0.8666666666666667
('Soap',) --> ('Milk',): Confidence 0.8125

------------------------------------------------------------
```

After doing many tests using different databases, support values, and confidence values, here are my results for the runtime of the three methods.

By far, the slowest method was Brute Force. The efficiency for Apriori and FP Tree depended on the inputted support and confidence values.

In this case, FP Tree was actually slower than Apriori when both support and confidence were 0.5. When the confidence was raised, the performance for FP Tree significantly decreased.

```
Database1.csv  ---  Support: 0.5  ---  Confidence: 0.5
Runtime of the 3 methods:
Brute Force: 0.4671168327331543 seconds
Apriori: 0.0153350830078125 seconds
FP Tree: 0.07620930671691895 seconds
```

```
Database1.csv  ---  Support: 0.5  ---  Confidence: 0.8
Runtime of the 3 methods:
Brute Force: 0.483527660369873O5 seconds
Apriori: 0.015034437179565 43 seconds
FP Tree: 0.009197950363159 18 seconds
```

In most cases, however, FP Tree was consistently faster than Apriori. Here are some screenshots showing the runtime comparison using various databases, support values, and confidence values.

```
------------------------------------------------------------
Database2.csv  ---  Support: 0.4  ---  Confidence: 0.75
Runtime of the 3 methods:
Brute Force: 0.462252140045166 seconds
Apriori: 0.021493434906005 86 seconds
FP Tree: 0.014642477035522461 seconds
```

```
------------------------------------------------------------
Brute Force
Time Taken: 0.5023200511932373 seconds



------------------------------------------------------------
Apriori
Time Taken: 0.009423732757568 36 seconds



------------------------------------------------------------
FP Tree
Time Taken: 0.0010001659393310547 seconds




------------------------------------------------------------
Database1.csv  ---  Support: 0.9  ---  Confidence: 0.3
Runtime of the 3 methods:
Brute Force: 0.5023200511932373 seconds
Apriori: 0.009423732757568 36 seconds
FP Tree: 0.0010001659393310547 seconds
```

```
------------------------------------------------------------
Database3.csv  ---  Support: 0.5  ---  Confidence: 0.8
Runtime of the 3 methods:
Brute Force: 0.502882719039917 seconds
Apriori: 0.020616769790649414 seconds
FP Tree: 0.003720998764038086 seconds
```

```
------------------------------------------------------------
Database4.csv  ---  Support: 0.65  ---  Confidence: 0.8
Runtime of the 3 methods:
Brute Force: 0.375621318817138 67 seconds
Apriori: 0.014174222946166992 seconds
FP Tree: 0.008939266204833984 seconds
```

```
------------------------------------------------------------
Database5.csv  ---  Support: 0.4  ---  Confidence: 0.6
Runtime of the 3 methods:
Brute Force: 0.41196417808532715 seconds
Apriori: 0.012770175933837 89 seconds
FP Tree: 0.005632162094116211 seconds
```