ASSOCIATION RULE MINING USING MARKET BASKET ANALYSIS

Bachelor Of Technology

In

Computer Science And Engineering

By

G.Neelima (16JG1A0533) J.Pavani (16JG1A041)

D.Vineela (16JG1A0526) D.Poornima (16JG1A0524)

CONTENT

ABSTRACT

Different people have different choices and different buying habits as well. To fulfil the various demands of different people based on their buying behaviour and taste, retailers (and the Government) have to choose some sample goods or services to know the buying behaviour and cost of living of an economy. This is when market basket analysis comes into the picture. This is an imaginary basket used by the vendors and Government to fulfil the customer demand. Market basket analysis is done by the retailers to check the correlation of two or more items that the customers are likely to buy.

The Objective of this project is to find what items are frequently bought together by the customer. The rules and the acquired frequent items sets can help in effective sales and marketing.

INTRODUCTION

1.1 Market basket:

The market basket is a list of some fixed items that are used to track the inflation and overall price movements of a specific market in an economy. A market basket contains some sample goods and services to know the inflation. In this technique, the inflation rate is the change in the cost of living of two baskets. Market basket and its contents are supposed to change every time to calculate inflation in an efficient and effective manner.

1.2 Market basket analysis:

Market basket analysis is a method or technique of data analysis for retail and marketing purpose. The idea behind market basket analysis has emerged from customers who are buying and adding different products to their shopping cart or in a market basket. MBA (Market Business Analysis) is used to uncover what items are frequently brought together by the customer. Market basket analysis leads to effective sales and marketing. Market basket analysis measures the co-occurrence of products and services. Market basket analysis is only considered when there are many transactions in which each transaction has two or more items. It is also used to predict future purchase decision of a customer.

LITERATURE SURVEY

Within the field of machine learning ,there are two main types of tasks:
1.Supervised Learning
2.Unsupervised Learning

Supervised Learning: Learning from the know label data to create a model then predicting target class for the given input data. Supervised learning is where you have input variables (x) and an output variable (Y) and you use an algorithm to learn the mapping function from the input to the output.

$$Y = f(X)$$

The goal is to approximate the mapping function so well that when you have new input data (x) that you can predict the output variables (Y) for that data. Supervised learning problems can be further grouped into regression and classification problems.

- **Classification**: A classification problem is when the output variable is a category, such as “red” or “blue” or “disease” and “no disease”.
- **Regression**: A regression problem is when the output variable is a real value, such as “dollars” or “weight”.

Unsupervised Learning: Learning from the unlabeled data to differentiating the given input data. The goal for unsupervised learning is to model the underlying structure or distribution in the data in order to learn more about the data. Unsupervised learning problems can be further grouped into clustering and association problems

- **Clustering**: A clustering problem is where you want to discover the inherent groupings in the data, such as grouping customers by purchasing behavior.
- **Association**: An association rule learning problem is where you want to discover rules that describe large portions of your data, such as people that buy X also tend to buy Y.

Apriori:

- It is given by R. Agrawal and R. Srikant in 1994 for finding frequent itemsets in a dataset for boolean association rule.
- It uses prior knowledge of frequent itemset properties.
- We apply an iterative approach or level-wise search where k-frequent itemsets are used to find k+1 itemsets.
- To improve the efficiency of level-wise generation of frequent itemsets, an important property is used called ***Apriori property*** which helps by reducing the search space.
- Apriori Property: All subsets of a frequent itemset must be frequent (Apriori propertry). If an itemset is infrequent, all its supersets will be infrequent.
- There are multiple rules possible even from a very small database, so in order to select the interesting ones, we use constraints on various measures of interest and significance.
- Some of the metrics are:
    - **Support**: The support of an itemset X, supp(X) is the proportion of transaction in the database in which the item X appears. It signifies the popularity of an itemset.

$$supp(X) = \frac{Number\ of\ \text{transaction in which X appears}}{Total\ number\ of\ transactions}$$

    - **Confidence**: It signifies the likelihood of item Y being purchased when item X is purchased.

$$conf(X \to Y) = \frac{supp(XUY)}{supp(X)}$$

    - Other metrics are lift , conviction.

The Apriori Algorithm:

It is an iterative process involving two steps:

1. **The join step**: To find L_k (set of k-itemsets that have support count not less than the minimum support), a set of **candidate** *k*-itemsets is generated by joining L_{k-1} with itself. This set of candidates is denoted C_k.

2. **The prune step**: *Ck* is a superset of L_k, that is, its members may or may not be frequent, but all of the frequent *k*-itemsets are included in C_k. A database scan to determine the count of each candidate in *Ck* would result in the determination of L_k. To reduce the size of C_k, the Apriori property is used as follows. Any (*k* -1)-itemset that is not frequent cannot be a subset of a frequent *k*-itemset. Hence, if any (k-1)-subset of a candidate *k*-itemset is not in L_{k-1}, then the candidate cannot be frequent either and so can be removed from *Ck*.

The steps 1 and 2 are repeated till we get can no longer produce candidate sets(C_k).

Generating Association Rules from Frequent Itemsets:

Once the frequent itemsets from transactions in a database *D* have been found, it is straightforward to generate strong association rules from them (where *strong* association rules satisfy both minimum support and minimum confidence). This can be done using the below equation :

$$confidence\ (A => B) = P\left(\frac{B}{A}\right) = \frac{support\_count(A\ U\ B)}{support\_count(A)}$$

The conditional probability is expressed in terms of itemset support count, where *support_count*(*A* U *B*) is the number of transactions containing the itemsets *A U B*, and *support_count*(*A*) is the number of transactions containing the itemset *A*. Based on this equation, association rules can be generated as follows:

- For each frequent itemset *l*, generate all nonempty subsets of *l*.
- For every nonempty subset *s* of *l*, output the rule "*s=>(l - s)*" if $\frac{support\_count(l)}{support\_count(s)} \geq$ *min conf*, where *min conf* is the minimum confidence threshold.

Because the rules are generated from frequent itemsets, each one automatically satisfies the minimum support.

REQUIREMENTS

1. SOFTWARE:

- Operating System : Windows 8.1
- Application : Spyder 3.7
- Graphical User Interface (GUI) : Tkinter in python
- IDE : Spyder 3.7

2. HARDWARE:

- Processor : Intel(R) Core(TM) i5-7200U CPU @2.5GHz 2.71GHz

IMPLEMENTATION

1. **FRONT END**:

```
import tkinter as tk
import apriori as apri
import numpy as np

dic = {}
dec = {}

def isValidFile(name):
    l=name.split(".")
    if len(l)!=2:
        return 0
    if l[1]!='csv' and l[1]!='arff' and l[1]!='xls':
        return 0
    return 1

def getstr(l):
    if isinstance(l,int):
        return "[ "+dec[l]+" ]"
    t = "["
    for item in range(len(l)):
        t += str(dec[l[item]])
        if len(l)==1 or item == len(l)-1 :
            break
        t += ", "
    t += "]"
    return t

def create_win(note,title,font):
    window = tk.Tk()
    window.title(title)
    n = tk.Label(window, text=note,wraplength=210,
pady=10,bg="white",font=("Helvetica",font)).grid(row=0,padx=
50,pady=50)
```

```
    window.mainloop()

def show_freq_items(frequent_items):
    window = tk.Tk()
    window.title("FREQUENT ITEMSETS")
    text = "\n"
    count = 0
    freq_list = tk.Listbox(window, width=100,height=30)
    scrollbar1 = tk.Scrollbar(window)
    scrollbar1.pack(side=tk.RIGHT, fill=tk.Y)
    for i in frequent_items:
        text = ""
        for j in i:
            freq_list.insert(tk.END, getstr(j))
            count += 1
    tk.Label(window, text="THE FREQUENT ITEM SETS
ARE:" + str(count), bg="blue",
width=90,pady=5,font=("Helvetica", 10)).pack()
    freq_list.config(yscrollcommand=scrollbar1.set)
    freq_list.pack()
    scrollbar1.config(command=freq_list.yview)
    tk.Button(window, width=90, padx=50, pady=5,
text="EXIT", command=window.destroy).pack()
    window.mainloop()

def show_association_rules(rules):
    window = tk.Tk()
    window.title("ASSOCIATION RULES")
    tk.Label(window, text="THE NUMBER OF ASSOCIATION
RULES ARE:" +
str(len(rules)),bg="blue",width=90,pady=5,font=("Helvetica",10
)).pack()
    list_box=tk.Listbox(window,width=100,height=30)
    scrollbar2 = tk.Scrollbar(window)
    scrollbar2.pack(side=tk.RIGHT, fill=tk.Y)
    for rule in rules:
```

```
        text="( "+getstr(rule.antecedent)+"-
>"+getstr(rule.concequent)+" )  s:"+str(round(rule.support,5))+"
c:"+str(round(rule.confidence,5))
        list_box.insert(tk.END,text)
    list_box.config(yscrollcommand=scrollbar2.set)
    list_box.pack()
    scrollbar2.config(command=list_box.yview)
    tk.Button(window,padx=50,width=90, pady=5, text="EXIT",
command=window.destroy).pack()
    window.mainloop()

def final_results_window(rules,freq_items):
    window=tk.Tk()
    window.title("FINAL RESULTS")
    tk.Button(window, width=10, padx=50, pady=5,
text="SHOW ASSOCIATION
RULES",command=lambda:show_association_rules(rules)).grid
(row=0,padx=100, pady=10)
    tk.Button(window, width=10, padx=50, pady=5,
text="SHOW FREQUENT ITEMSETS",
command=lambda:show_freq_items(freq_items)).grid(row=1,pa
dx=100, pady=10)
    tk.Button(window, width=10, padx=50, pady=5,
text="EXIT",
command=window.destroy).grid(row=3,padx=100, pady=10)
    window.mainloop()


def apply_algo():
    try:
        filename = algo_name.get()
        a = thre_supp.get()
        b = thre_conf.get()
        window.destroy()
        min_sup=float(a)
        min_conf=float(b)
        print("Received the file name:",filename)
```

```
    if isValidFile(filename):
        data = []
        """OPENING THE FILE"""
        f = open(filename)
        data_txt = f.read().split("\n")
        for i in data_txt:
            data.append(i.split(","))

        """AND GETTING NUMERIC DATA"""
        if type(data[0][0]).__name__=='str':
            new_list = list()
            for i in data:
                l = []
                for j in i:
                    if j not in dic:
                        dic[j] = len(dic)
                        dec[len(dec)]=j
                    l.append(dic[j])
                new_list.append(l)
            data=new_list

        """CONVERTING NUMERIC DATA TO NUMPY
ARRAY"""
        transactions = np.array(data)
        apriori = apri.Apriori(min_sup=min_sup,
min_conf=min_conf)
        create_win("PROCESSESING\n
STARTED","STATUS",16)
        rules = apriori.generate_rules(transactions)
        create_win("THE GIVEN FILE\n HAS BEEN
\nPROCESSED","STATUS",16)
        frequent_items=apriori.freq_itemsets
        final_results_window(rules,frequent_items)

    else:
      create_win("Invalid file name","ERROR FOUND",30)
```

```
    except Exception as e:
          create_win("File does'nt exist","ERROR FOUND",30)

window = tk.Tk()
window.title("APRIORI ALGORITHM")
window.geometry("350x140")
tk.Label(window,text="File
name",padx=30,pady=5).grid(row=0,column=0)
algo_name =tk.Entry(window,bg="white",width=20)
algo_name.grid(row=0,column=1)
tk.Label(window,padx=30,pady=5,text="Threshold
support").grid(row=1,column=0)
thre_supp =tk.Entry(window,bg="white",width=20)
thre_supp.grid(row=1,column=1)
tk.Label(window,padx=30,pady=5,text="Threshold
confidence").grid(row=2,column=0)
thre_conf =tk.Entry(window,bg="white",width=20)
thre_conf.grid(row=2,column=1)
tk.Button(window,width=5,padx=30,pady=5,text="RUN",comm
and=apply_algo).grid(row=3,column=0)
tk.Button(window,width=5,padx=30,pady=5,text="EXIT",com
mand=window.destroy).grid(row=3,column=1)
window.mainloop()
```

2. BACKEND

(APRIORI ALGORITHM IMPLEMENTATION)

```
from __future__ import division, print_function
import numpy as np
import itertools

class Rule():
    def __init__(self, antecedent, concequent, confidence,
support):
        self.antecedent = antecedent
        self.concequent = concequent
```

```
        self.confidence = confidence
        self.support = support


class Apriori():
    """A method for determining frequent itemsets in a
transactional database and also for generating rules for those
itemsets.
    Parameters:
    -----------
    min_sup: float
        The minimum fraction of transactions an itemets needs to
occur in to be deemed frequent
    min_conf: float
        The minimum fraction of times the antecedent needs to
imply the concequent to justify rule
    """
    def __init__(self, min_sup=0.3, min_conf=0.81):

        """


        :rtype:
        """
        self.min_sup = min_sup
        self.min_conf = min_conf
        self.freq_itemsets = None       # List of freqeuent itemsets
        self.transactions = None        # List of transactions

    def _calculate_support(self, itemset):
        count = 0
        for transaction in self.transactions:
            if self._transaction_contains_items(transaction, itemset):
                count += 1
        support = count / len(self.transactions)
        return support

    # Prunes the candidates that are not frequent
```

```
    # => returns list with only frequent itemsets
    def _get_frequent_itemsets(self, candidates):
        frequent = []
        # Find frequent items
        for itemset in candidates:
            support = self._calculate_support(itemset)
            if support >= self.min_sup:
                frequent.append(itemset)
        return frequent

    # True or false depending on the candidate has any
    # subset with size k - 1 that is not in the frequent
    # itemset
    def _has_infrequent_itemsets(self, candidate):
        k = len(candidate)
        # Find all combinations of size k-1 in candidate
        # E.g [1,2,3] => [[1,2],[1,3],[2,3]]
        subsets = list(itertools.combinations(candidate, k - 1))
        for t in subsets:
            # t - is tuple. If size == 1 get the element
            subset = list(t) if len(t) > 1 else t[0]
            if not subset in self.freq_itemsets[-1]:
                return True
        return False

    # Joins the elements in the frequent itemset and prunes
    # resulting sets if they contain subsets that have been
determined
    # to be infrequent.
    def _generate_candidates(self, freq_itemset):
        candidates = []
        for itemset1 in freq_itemset:
            for itemset2 in freq_itemset:
                # Valid if every element but the last are the same
                # and the last element in itemset1 is smaller than the
last
                # in itemset2
```

```
            valid = False
            single_item = isinstance(itemset1, int)
            if single_item and itemset1 < itemset2:
                valid = True
            elif not single_item and np.array_equal(itemset1[:-1],
itemset2[:-1]) and itemset1[-1] < itemset2[-1]:
                valid = True

            if valid:
                # JOIN: Add the last element in itemset2 to
itemset1 to
                # create a new candidate
                if single_item:
                    candidate = [itemset1, itemset2]
                else:
                    candidate = itemset1 + [itemset2[-1]]
                # PRUNE: Check if any subset of candidate have
been determined
                # to be infrequent
                infrequent =
self._has_infrequent_itemsets(candidate)
                if not infrequent:
                    candidates.append(candidate)
        return candidates

    # True or false depending on each item in the itemset is
    # in the transaction
    def _transaction_contains_items(self, transaction, items):
        # If items is in fact only one item
        if isinstance(items,str) or isinstance(items,int):
            return items in transaction
        # Iterate through list of items and make sure that
        # all items are in the transaction
        for item in items:
            if not item in transaction:
                return False
        return True
```

```
    # Returns the set of frequent itemsets in the list of
transactions
    def find_frequent_itemsets(self, transactions):
        self.transactions = transactions
        # Get all unique items in the transactions
        unique_items = set(item for transaction in self.transactions
for item in transaction)
        # Get the frequent items
        self.freq_itemsets =
[self._get_frequent_itemsets(unique_items)]
        while(True):
            # Generate new candidates from last added frequent
itemsets
            candidates =
self._generate_candidates(self.freq_itemsets[-1])
            # Get the frequent itemsets among those candidates
            frequent_itemsets =
self._get_frequent_itemsets(candidates)

            # If there are no frequent itemsets we're done
            if not frequent_itemsets:
                break

            # Add them to the total list of frequent itemsets and start
over
            self.freq_itemsets.append(frequent_itemsets)

        # Flatten the array and return every frequent itemset
        frequent_itemsets = [itemset for sublist in
self.freq_itemsets for itemset in sublist]
        return frequent_itemsets

    # Recursive function which returns the rules where
confidence >= min_confidence
    # Starts with large itemset and recursively explores rules for
subsets
```

```
    def _rules_from_itemset(self, initial_itemset, itemset):
        rules = []
        k = len(itemset)
        # Get all combinations of sub-itemsets of size k - 1 from
itemset
        # E.g [1,2,3] => [[1,2],[1,3],[2,3]]
        subsets = list(itertools.combinations(itemset, k - 1))
        support = self._calculate_support(initial_itemset)
        for antecedent in subsets:
            # itertools.combinations returns tuples => convert to list
            antecedent = list(antecedent)
            antecedent_support =
self._calculate_support(antecedent)
            # Calculate the confidence as sup(A and B) / sup(B), if
antecedent
            # is B in an itemset of A and B
            confidence = float("{0:.2f}".format(support /
antecedent_support))
            if confidence >= self.min_conf:
                # The concequent is the initial_itemset except for
antecedent
                concequent = [itemset for itemset in initial_itemset if
not itemset in antecedent]
                # If single item => get item
                if len(antecedent) == 1:
                    antecedent = antecedent[0]
                if len(concequent) == 1:
                    concequent = concequent[0]
                # Create new rule
                rule =
Rule(antecedent=antecedent,concequent=concequent,confidence
=confidence, support=support)
                rules.append(rule)
                # If there are subsets that could result in rules
                # recursively add rules from subsets
                if k - 1 > 1:
```

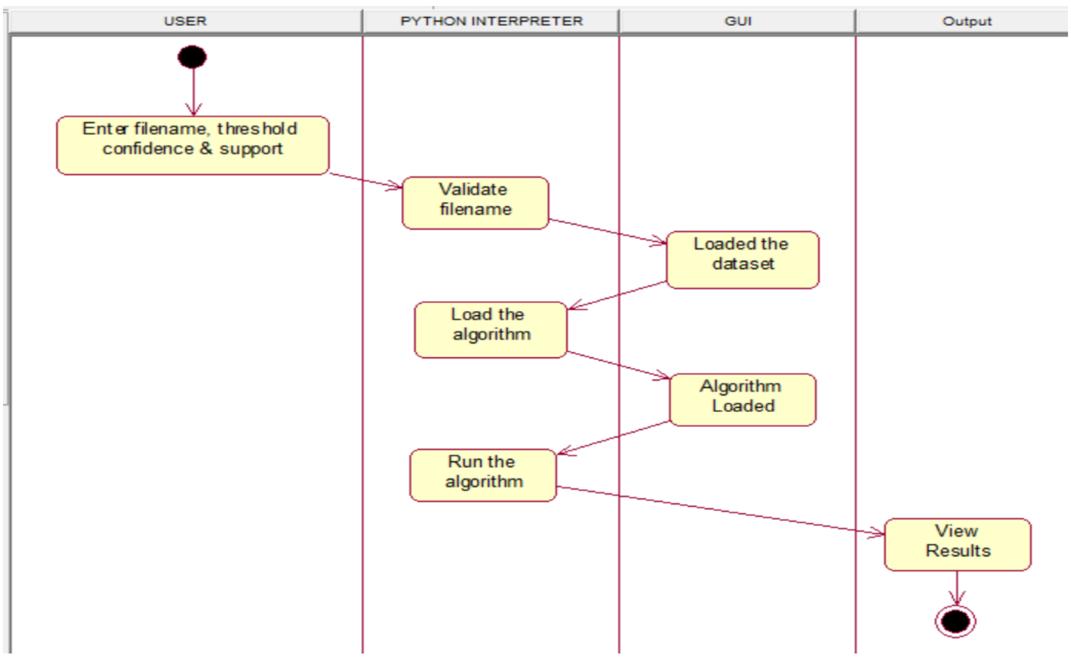
```
                rules += self._rules_from_itemset(initial_itemset,
antecedent)
        return rules

    def generate_rules(self, transactions):
        self.transactions = transactions
        frequent_itemsets =
self.find_frequent_itemsets(transactions)
        # Only consider itemsets of size >= 2 items
        frequent_itemsets = [itemset for itemset in
frequent_itemsets if not isinstance(itemset, int) or
isinstance(itemset,str)]
        rules = []
        for itemset in frequent_itemsets:
            rules += self._rules_from_itemset(itemset, itemset)
        # Remove empty values
        return rules
```

UML DIAGRAM

We use **Activity Diagrams** to illustrate the flow of control in a system and refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram. An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.

UML models basically three types of diagrams, namely, structure diagrams, interaction diagrams, and behavior diagrams. An activity diagram is a **behavioural diagram** i.e. it depicts the behavior of a system.

An activity diagram portrays the control flow from a start point to a finish point showing the various decision paths that exist while the activity is being executed. We can depict both sequential processing and concurrent processing of activities using an activity diagram. They are used in business and process modelling where their primary use is to depict the dynamic aspects of a system.

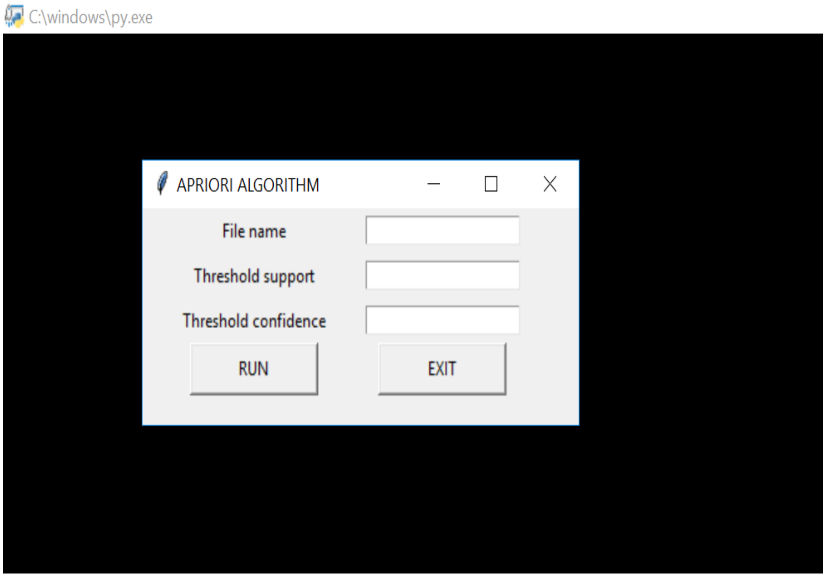


DATASET

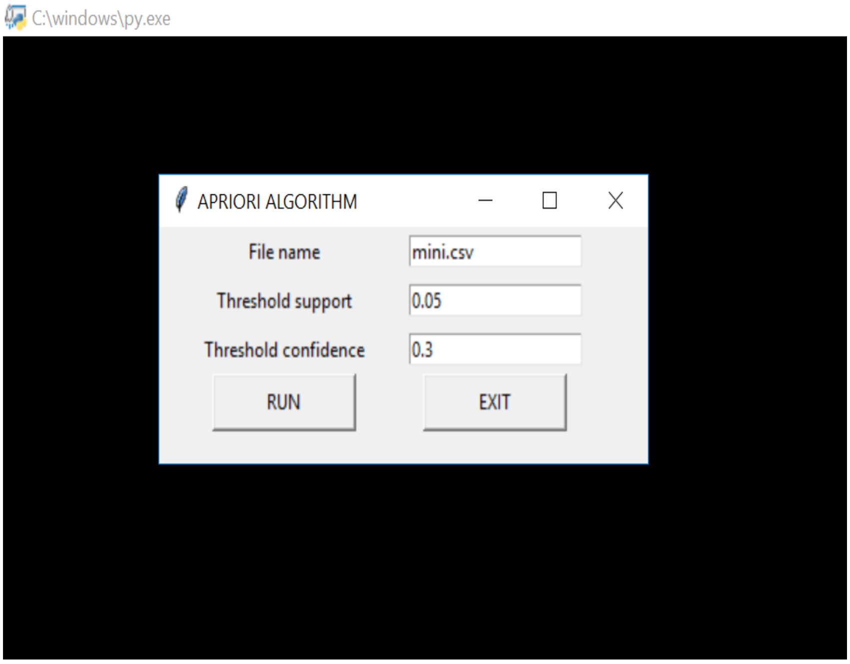
```
citrus fruit,semi-finished bread,margarine,ready soups
tropical fruit,yogurt,coffee
whole milk
pip fruit,yogurt,cream cheese,meat spreads
other vegetables,whole milk,condensed milk,long life bakery product
whole milk,butter,yogurt,rice,abrasive cleaner
rolls/buns
other vegetables,UHT-milk,rolls/buns,bottled beer,liquor (appetizer)
potted plants
whole milk,cereals
tropical fruit,other vegetables,white bread,bottled water,chocolate
citrus fruit,tropical fruit,whole milk,butter,curd,yogurt,flour,bottled water,dishes
beef
frankfurter,rolls/buns,soda
chicken,tropical fruit
butter,sugar,fruit/vegetable juice,newspapers
fruit/vegetable juice
packaged fruit/vegetables
chocolate
specialty bar
other vegetables
butter milk,pastry
whole milk
tropical fruit,cream cheese,processed cheese,detergent,newspapers
tropical fruit,root vegetables,other vegetables,frozen dessert,rolls/buns,flour,sweet spreads,salty snack,waffles,candy,bathroom cleaner
bottled water,canned beer
yogurt
sausage,rolls/buns,soda,chocolate
other vegetables
brown bread,soda,fruit/vegetable juice,canned beer,newspapers,shopping bags
yogurt,beverages,bottled water,specialty bar
hamburger meat,other vegetables,rolls/buns,spices,bottled water,hygiene articles,napkins
root vegetables,other vegetables,whole milk,beverages,sugar
pork,berries,other vegetables,whole milk,whipped/sour cream,artif. sweetener,soda,abrasive cleaner
beef,grapes,detergent
pastry,soda
```

ANALYSIS

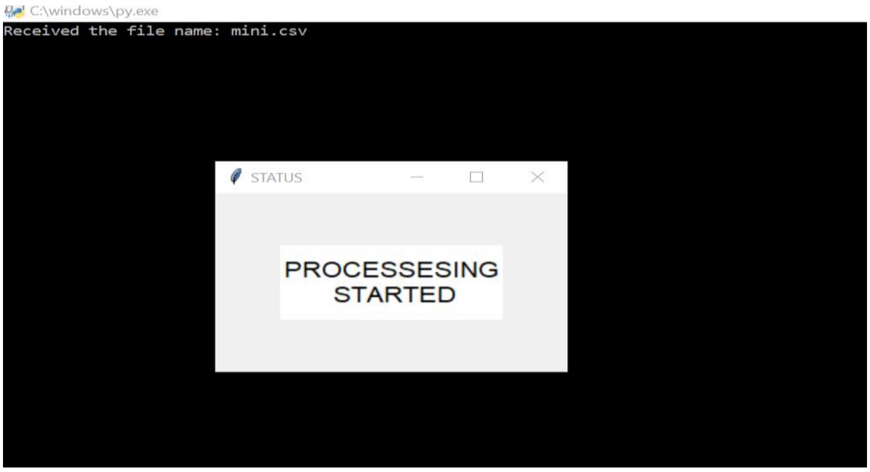
1. Run the program



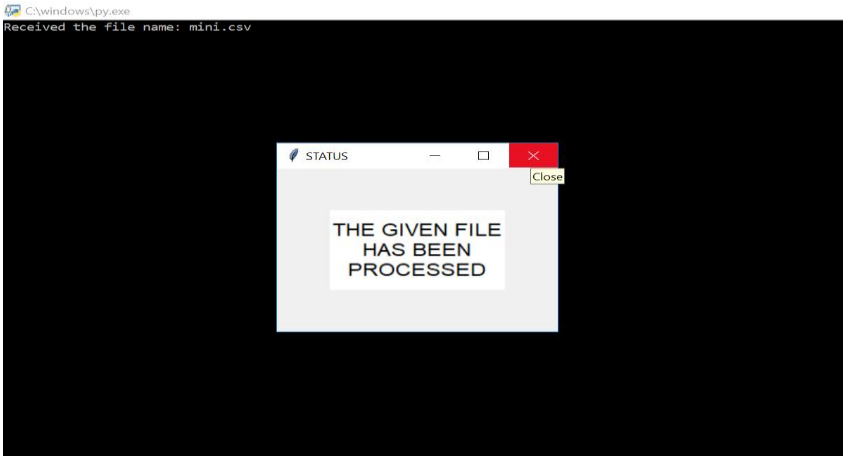
2. Enter the filename along with minimum support and minimum confidence.



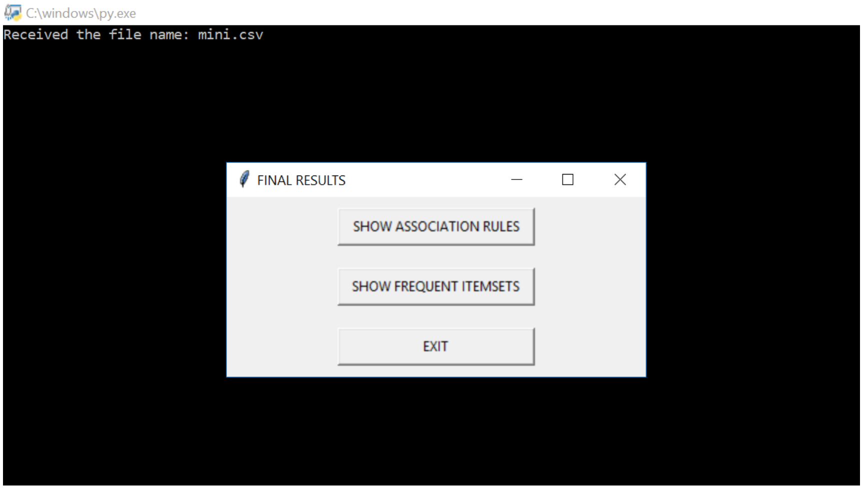
3. Processing started.



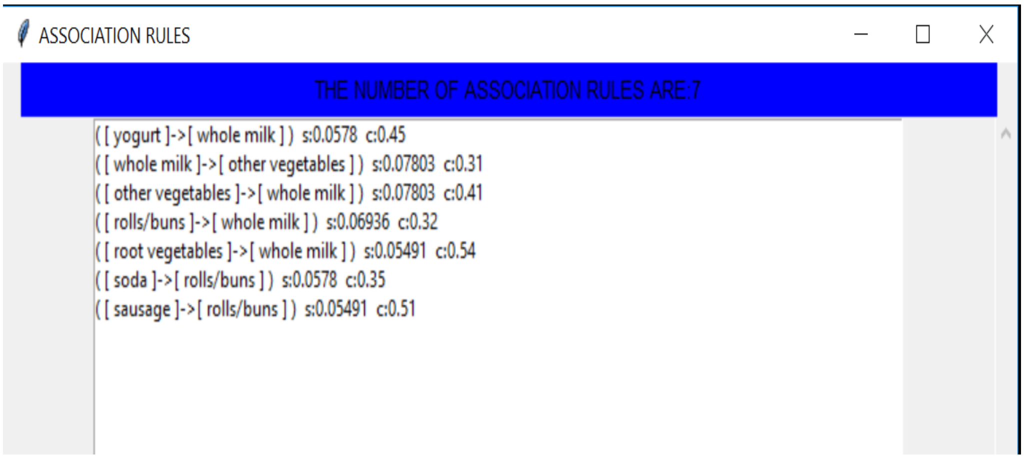
4. The file has been processed.



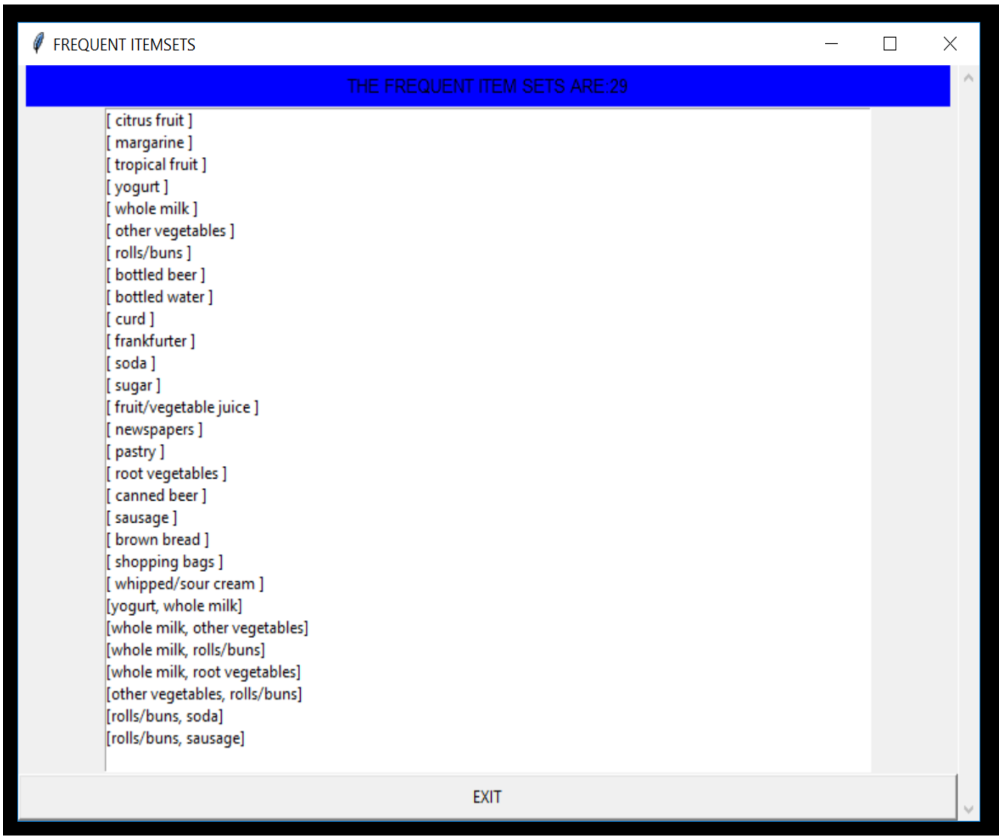
5. The frequent itemsets and association rules can be viewed.



6. The frequent Itemsets are



7. The association rules obtained are



CONCLUSION

In data mining, association rules are useful for analyzing and predicting customer behaviour. They can be helpful in store layout, marketing messages, maintain inventory, content placement and also in recommendation engines.

The market basket analysis has many applications like cross selling, product placement, affinity promotion, fraud detection, understanding customer behaviour.

The efficiency of apriori can be improved using hash-based itemset counting, transaction reduction, partitioning, sampling and dynamic itemset counting.

REFERENCES

[1] https://www.newgenapps.com/blog/application-of-market-basket-analysis
[2].http://www.sci.csueastbay.edu/~esuess/classes/Statistics_6620/Presentations/ml13/groceries.csv
[3].https://www3.cs.stonybrook.edu/~cse634/lecture_notes/07apriori.pdf
[4]. https://www.tutorialspoint.com/uml/uml_activity_diagram.html