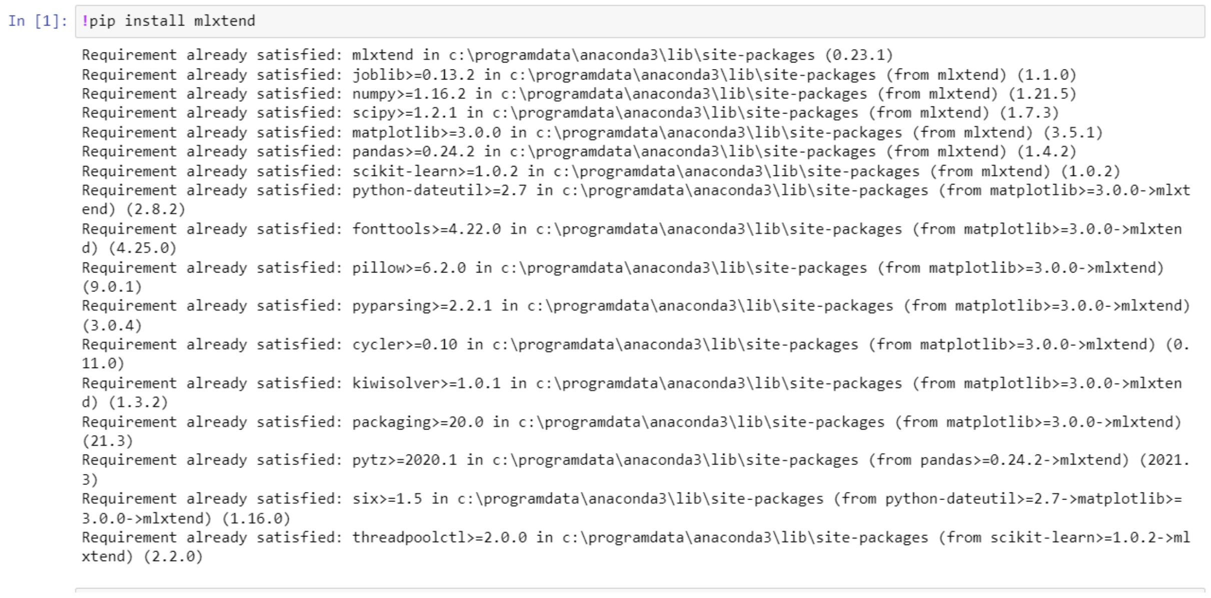
Association Mining

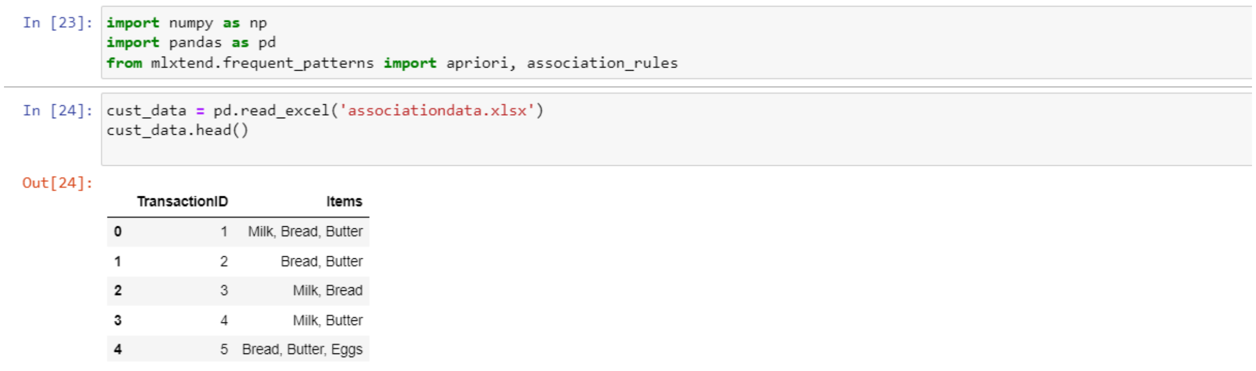
Association mining, in the context of data mining, is a method for discovering interesting relationships and patterns within large datasets. It identifies how often certain items or events co-occur, revealing underlying connections and correlations. This technique helps businesses and researchers gain insights into data, enabling them to make informed decisions, optimize strategies, and predict future trends. Association mining is essential for understanding complex data relationships and improving various applications, such as marketing, inventory management, and customer behavior analysis.

For implementing association mining we will use dataset that contains "Transaction ID" and "Items" which contain combination of Milk, Bread, Butter and Eggs.



```
In [1]: !pip install mlxtend

        Requirement already satisfied: mlxtend in c:\programdata\anaconda3\lib\site-packages (0.23.1)
        Requirement already satisfied: joblib>=0.13.2 in c:\programdata\anaconda3\lib\site-packages (from mlxtend) (1.1.0)
        Requirement already satisfied: numpy>=1.16.2 in c:\programdata\anaconda3\lib\site-packages (from mlxtend) (1.21.5)
        Requirement already satisfied: scipy>=1.2.1 in c:\programdata\anaconda3\lib\site-packages (from mlxtend) (1.7.3)
        Requirement already satisfied: matplotlib>=3.0.0 in c:\programdata\anaconda3\lib\site-packages (from mlxtend) (3.5.1)
        Requirement already satisfied: pandas>=0.24.2 in c:\programdata\anaconda3\lib\site-packages (from mlxtend) (1.4.2)
        Requirement already satisfied: scikit-learn>=1.0.2 in c:\programdata\anaconda3\lib\site-packages (from mlxtend) (1.0.2)
        Requirement already satisfied: python-dateutil>=2.7 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.0.0->mlxt
        end) (2.8.2)
        Requirement already satisfied: fonttools>=4.22.0 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.0.0->mlxten
        d) (4.25.0)
        Requirement already satisfied: pillow>=6.2.0 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.0.0->mlxtend)
        (9.0.1)
        Requirement already satisfied: pyparsing>=2.2.1 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.0.0->mlxtend)
        (3.0.4)
        Requirement already satisfied: cycler>=0.10 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.0.0->mlxtend) (0.
        11.0)
        Requirement already satisfied: kiwisolver>=1.0.1 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.0.0->mlxten
        d) (1.3.2)
        Requirement already satisfied: packaging>=20.0 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.0.0->mlxtend)
        (21.3)
        Requirement already satisfied: pytz>=2020.1 in c:\programdata\anaconda3\lib\site-packages (from pandas>=0.24.2->mlxtend) (2021.
        3)
        Requirement already satisfied: six>=1.5 in c:\programdata\anaconda3\lib\site-packages (from python-dateutil>=2.7->matplotlib>=
        3.0.0->mlxtend) (1.16.0)
        Requirement already satisfied: threadpoolctl>=2.0.0 in c:\programdata\anaconda3\lib\site-packages (from scikit-learn>=1.0.2->ml
        xtend) (2.2.0)
```



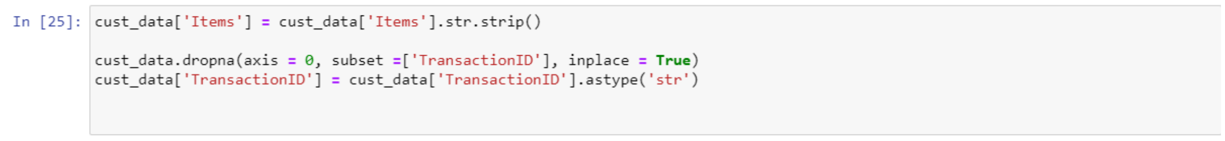
```
In [23]: import numpy as np
         import pandas as pd
         from mlxtend.frequent_patterns import apriori, association_rules
```

```
In [24]: cust_data = pd.read_excel('associationdata.xlsx')
         cust_data.head()
```

Out[24]:

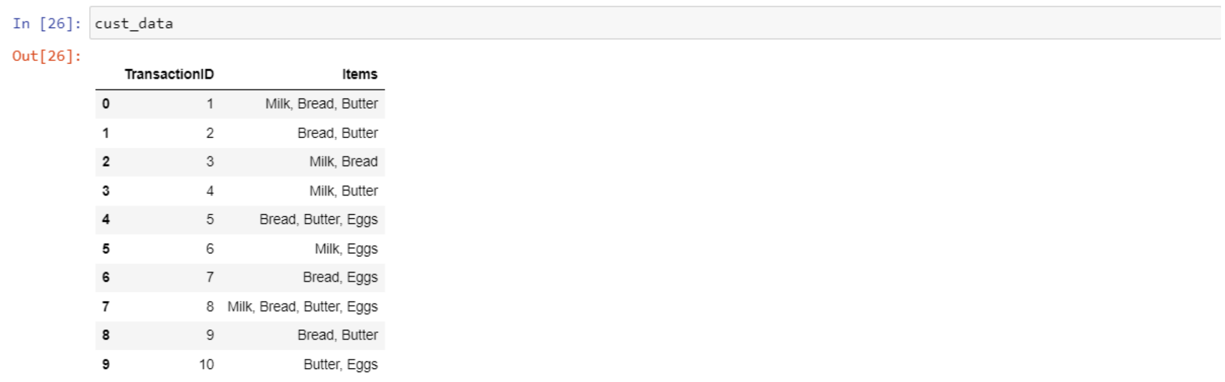
	TransactionID	Items
0	1	Milk, Bread, Butter
1	2	Bread, Butter
2	3	Milk, Bread
3	4	Milk, Butter
4	5	Bread, Butter, Eggs

Now we will perform some data cleaning and preparation on the cust_data DataFrame. First, we remove any whitespace characters from the strings in the 'Items' column, ensuring that item names are clean and consistent. Next, we drop any rows in the DataFrame that have missing values in the 'TransactionID' column, ensuring that all transactions have an associated ID. Finally, we convert the data type of the 'TransactionID' column to string, which can be useful for consistency, if some IDs were originally numbers and others were strings. Overall, these steps clean and prepare the cust_data DataFrame for further analysis.



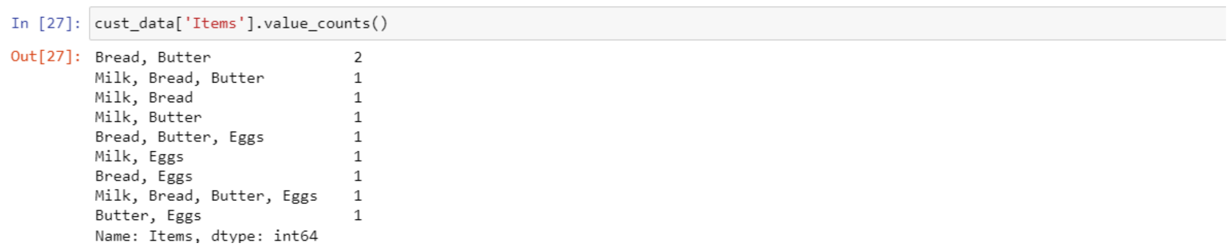
```
In [25]: cust_data['Items'] = cust_data['Items'].str.strip()

         cust_data.dropna(axis = 0, subset =['TransactionID'], inplace = True)
         cust_data['TransactionID'] = cust_data['TransactionID'].astype('str')
```



```
In [26]: cust_data
Out[26]:
```

	TransactionID	Items
0	1	Milk, Bread, Butter
1	2	Bread, Butter
2	3	Milk, Bread
3	4	Milk, Butter
4	5	Bread, Butter, Eggs
5	6	Milk, Eggs
6	7	Bread, Eggs
7	8	Milk, Bread, Butter, Eggs
8	9	Bread, Butter
9	10	Butter, Eggs



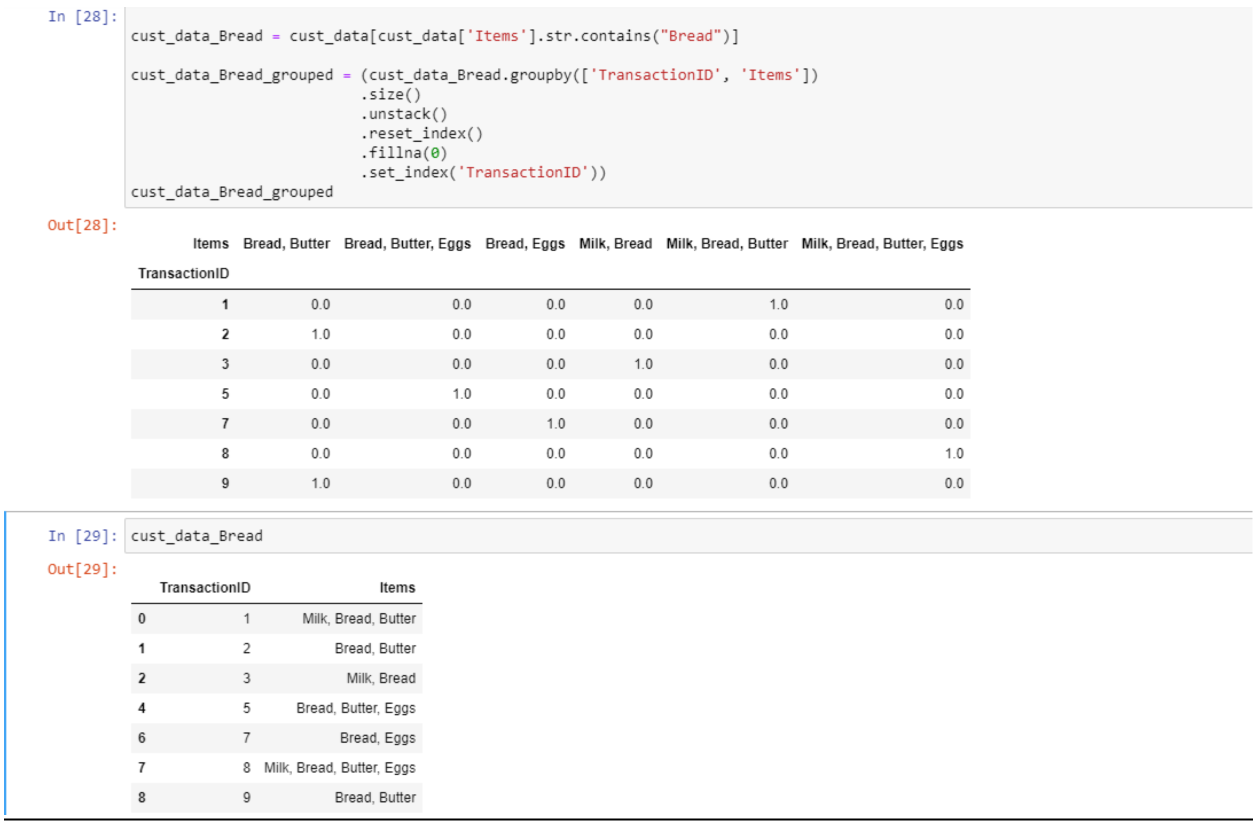
```
In [27]: cust_data['Items'].value_counts()

Out[27]: Bread, Butter                2
         Milk, Bread, Butter          1
         Milk, Bread                  1
         Milk, Butter                 1
         Bread, Butter, Eggs          1
         Milk, Eggs                   1
         Bread, Eggs                  1
         Milk, Bread, Butter, Eggs    1
         Butter, Eggs                 1
         Name: Items, dtype: int64
```

The figure helps to identify how many times each combination of items appears in the transactions. It provides a summary of the frequency of each item set purchased by customers, which is useful for understanding common purchasing patterns.

Filtering and Grouping Transactions Containing Bread

First, we filter the cust_data DataFrame to include only those transactions where the 'Items' column contains the word "Bread". This filtered data is stored in the cust_data_ Bread DataFrame. Next, we group this filtered DataFrame by 'TransactionID' and 'Items', counting the occurrences of each unique combination. The result is then transformed with the unstack method, which makes the table to have unique items as columns. We reset the index to make 'TransactionID' a column again and fill any missing values with zeros, ensuring a complete table. Finally, we set 'TransactionID' as the index of the DataFrame again, resulting in cust_data_Bread_grouped, which provides a structured view of transactions containing " Bread " and the corresponding items in those transactions.



In [28]:
```
cust_data_Bread = cust_data[cust_data['Items'].str.contains("Bread")]

cust_data_Bread_grouped = (cust_data_Bread.groupby(['TransactionID', 'Items'])
                           .size()
                           .unstack()
                           .reset_index()
                           .fillna(0)
                           .set_index('TransactionID'))
cust_data_Bread_grouped
```

Out[28]:

Items	Bread, Butter	Bread, Butter, Eggs	Bread, Eggs	Milk, Bread	Milk, Bread, Butter	Milk, Bread, Butter, Eggs
TransactionID						
1	0.0	0.0	0.0	0.0	1.0	0.0
2	1.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	1.0	0.0	0.0
5	0.0	1.0	0.0	0.0	0.0	0.0
7	0.0	0.0	1.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0	0.0	1.0
9	1.0	0.0	0.0	0.0	0.0	0.0

In [29]:
```
cust_data_Bread
```

Out[29]:

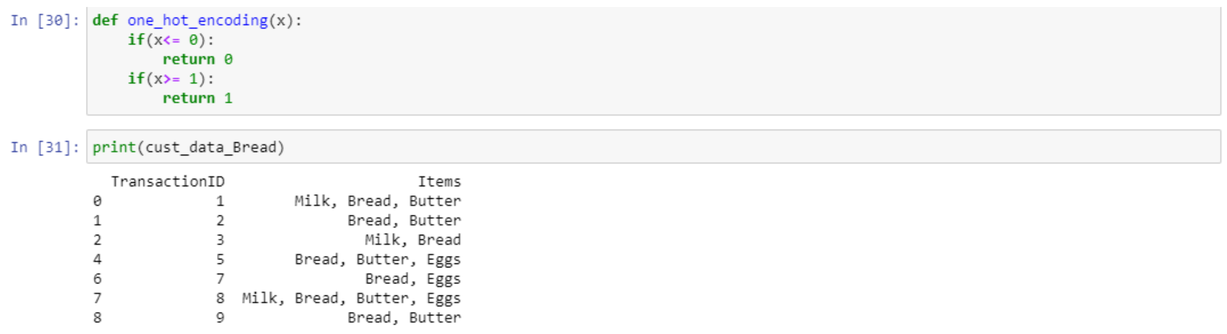
	TransactionID	Items
0	1	Milk, Bread, Butter
1	2	Bread, Butter
2	3	Milk, Bread
4	5	Bread, Butter, Eggs
6	7	Bread, Eggs
7	8	Milk, Bread, Butter, Eggs
8	9	Bread, Butter

This code filters the dataset to include only transactions containing "Milk" and then organizes these transactions into a clear, structured format, showing the presence of each item within those transactions. This allows for easy analysis of purchasing patterns involving milk.

Now we encode a list of customer data based on purchased items. First, we will define a function one_hot_encoding(x), which converts values into binary (0 or 1). If the input x is less than or equal to 0, it returns 0; if x is greater than or equal to 1, it returns 1.

Next, we will define a function encode_items(item_string) that takes a string of purchased items as input (item_string). It defines a list of items (items = ['Milk', 'Bread', 'Butter', 'Eggs']) and returns a list where each item is replaced by 1 if it appears in the item_string, and 0 if it does not. This function is applied to each row in a DataFrame column named 'Items' (cust_data_ Bread['Items']), resulting in a DataFrame (cust_data_Bread_encoded) where each row represents whether a customer bought each of the specified items ('Milk', 'Bread', 'Butter', 'Eggs'). The columns in this DataFrame correspond to these items, and each cell contains either 1 (indicating the customer bought that item) or 0 (indicating they did not).

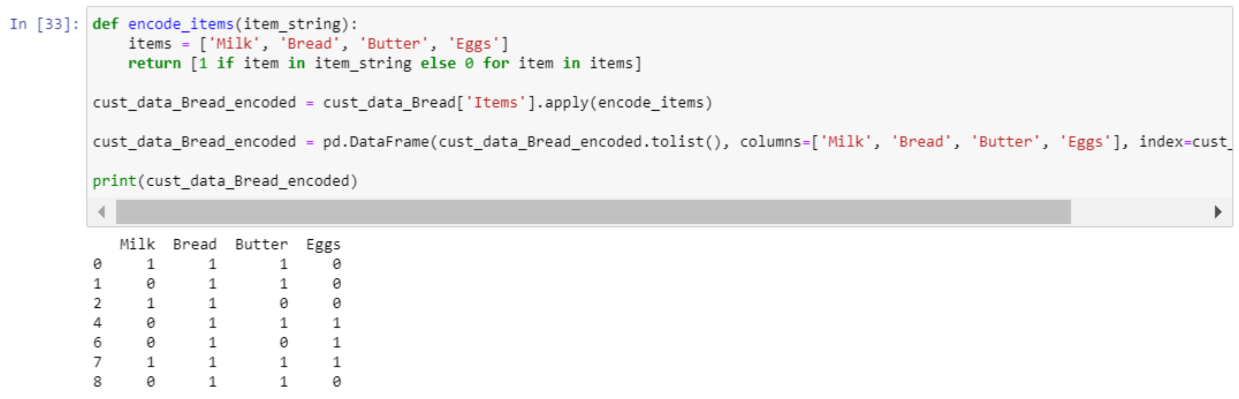
Finally, cust_data_Bread_encoded is printed to show the encoded representation of customer purchase data based on the items bought.



```
In [30]: def one_hot_encoding(x):
             if(x<= 0):
                 return 0
             if(x>= 1):
                 return 1
```

```
In [31]: print(cust_data_Bread)
```

```
   TransactionID                       Items
0              1         Milk, Bread, Butter
1              2               Bread, Butter
2              3                 Milk, Bread
4              5         Bread, Butter, Eggs
6              7                 Bread, Eggs
7              8   Milk, Bread, Butter, Eggs
8              9               Bread, Butter
```



```
In [33]: def encode_items(item_string):
             items = ['Milk', 'Bread', 'Butter', 'Eggs']
             return [1 if item in item_string else 0 for item in items]

         cust_data_Bread_encoded = cust_data_Bread['Items'].apply(encode_items)

         cust_data_Bread_encoded = pd.DataFrame(cust_data_Bread_encoded.tolist(), columns=['Milk', 'Bread', 'Butter', 'Eggs'], index=cust_

         print(cust_data_Bread_encoded)
```

```
   Milk  Bread  Butter  Eggs
0     1      1       1     0
1     0      1       1     0
2     1      1       0     0
4     0      1       1     1
6     0      1       0     1
7     1      1       1     1
8     0      1       1     0
```

Mining Association Rules from Customer Purchase Data

We demonstrates how to mine association rules from customer purchase data using the Apriori algorithm. Initially, the apriori function from mlxtend.frequent_patterns is used to find frequent itemsets in cust_data_Bread_encoded, which represents customer purchases encoded as binary values for items like 'Milk', 'Bread', 'Butter', and 'Eggs'. The parameter min_support=0.2 sets the minimum support threshold for itemsets to be considered frequent. These itemsets are then used to generate association rules using association_rules, focusing on rules with a confidence of at least 70%. The results (rules) are printed to show which items are commonly purchased together.

Then we proceed to experiment with different min_support values (0.01, 0.02, 0.05) to explore how varying support thresholds impact the number of association rules discovered. For each min_sup value, Apriori is applied again to find frequent itemsets (frq_items), and association rules are generated based on the lift metric (metric="lift"). The results display the number of rules found for each min_sup, illustrating how adjusting support thresholds can influence the granularity and number of discovered association rules.

```
In [34]: from mlxtend.frequent_patterns import apriori, association_rules

frequent_itemsets = apriori(cust_data_Bread_encoded, min_support=0.2, use_colnames=True)

rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.7)

print(rules)
```

```
      antecedents consequents  antecedent support  consequent support  \
0          (Milk)     (Bread)            0.428571            1.000000
1        (Butter)     (Bread)            0.714286            1.000000
2         (Bread)    (Butter)            1.000000            0.714286
3          (Eggs)     (Bread)            0.428571            1.000000
4  (Butter, Milk)     (Bread)            0.285714            1.000000
5  (Butter, Eggs)     (Bread)            0.285714            1.000000

    support  confidence  lift  leverage  conviction  zhangs_metric
0  0.428571    1.000000   1.0       0.0         inf            0.0
1  0.714286    1.000000   1.0       0.0         inf            0.0
2  0.714286    0.714286   1.0       0.0         1.0            0.0
3  0.428571    1.000000   1.0       0.0         inf            0.0
4  0.285714    1.000000   1.0       0.0         inf            0.0
5  0.285714    1.000000   1.0       0.0         inf            0.0
```

```
C:\ProgramData\Anaconda3\lib\site-packages\mlxtend\frequent_patterns\fpcommon.py:109: DeprecationWarning: DataFrames with non-b
ool types result in worse computationalperformance and their support might be discontinued in the future.Please use a DataFrame
with bool type
  warnings.warn(
```

```
In [35]: # Experiment with different min_support values
for min_sup in [0.01, 0.02, 0.05]:
    frq_items = apriori(cust_data_Bread_encoded, min_support=min_sup, use_colnames=True)
    rules = association_rules(frq_items, metric="lift", min_threshold=1)
    print(f"Number of rules with min_support {min_sup}: {len(rules)}")
```

```
Number of rules with min_support 0.01: 32
Number of rules with min_support 0.02: 32
Number of rules with min_support 0.05: 32
```

```
C:\ProgramData\Anaconda3\lib\site-packages\mlxtend\frequent_patterns\fpcommon.py:109: DeprecationWarning: DataFrames with non-b
ool types result in worse computationalperformance and their support might be discontinued in the future.Please use a DataFrame
with bool type
  warnings.warn(
C:\ProgramData\Anaconda3\lib\site-packages\mlxtend\frequent_patterns\fpcommon.py:109: DeprecationWarning: DataFrames with non-b
ool types result in worse computationalperformance and their support might be discontinued in the future.Please use a DataFrame
with bool type
  warnings.warn(
C:\ProgramData\Anaconda3\lib\site-packages\mlxtend\frequent_patterns\fpcommon.py:109: DeprecationWarning: DataFrames with non-b
ool types result in worse computationalperformance and their support might be discontinued in the future.Please use a DataFrame
with bool type
  warnings.warn(
```

Conclusion

Throughout the steps, we've processed customer purchase data to uncover patterns and associations using the Apriori algorithm. Initially, we encoded customer transactions into binary form to indicate which items were purchased. We then applied the Apriori algorithm to identify frequent itemsets, which represent combinations of items bought together frequently by customers. Using these itemsets, we generated association rules that highlight relationships between different products based on customer buying patterns. By experimenting with different support thresholds, we observed how adjusting this parameter affects the number and specificity of association rules discovered, providing insights into customer behavior and potential product correlations that can inform marketing strategies and product placements.