RECOMMENDATION ENGINE

SHUVADEEP MAITY, AYAN CHAKRABORTY

Contents

Abstract - The recommendation engine, utilizing the Jaccard index and the Apriori / Market Basket Analysis algorithm, delivers individualized product recommendations to customers. By taking into account customer preferences, purchase history, and price sensitivity, it offers personalized suggestions that enhance cross-selling and upselling opportunities.

Introduction - In the ever-evolving realm of retail, where consumers are presented with an abundance of choices, personalized shopping experiences have emerged as a critical competitive advantage. Customers now expect tailored product recommendations that cater to their unique preferences and shopping habits. To meet this demand, many retailers have turned to recommendation engines, sophisticated algorithms that leverage vast datasets to provide personalized product suggestions.In this case study, we explore the development and application of a recommendation engine within a North American multi-category multi-brand retail chain, focusing on a specific region. Our analysis encompasses 36 months of transactional data, with an impressive sample size of 4.2 million transactions, across seventeen different dimensions. This in-depth examination showcases the engine's capacity to harness data-driven insights to enhance the shopping experience, ultimately driving increased sales, customer satisfaction, and loyalty.

Problem statement - With a diverse customer base and vast product offerings, retailers grapple with the challenge of delivering relevant and personalized product recommendations to enhance customer engagement and drive repeat purchases. Product recomendations are often based on high margin products or top selling products. However, these recommendations are not personalized to the customer, and therefore do not lead to repeat purchases.

Objective - The primary objective of this study is to enhance customer engagement and sales performance through the implementation of a recommendation engine in a retail setting. Specifically, this research aims to:

Develop a recommendation engine that leverages the Jaccard index and Apriori/Market Basket Analysis algorithm to provide personalized product suggestions to customers.

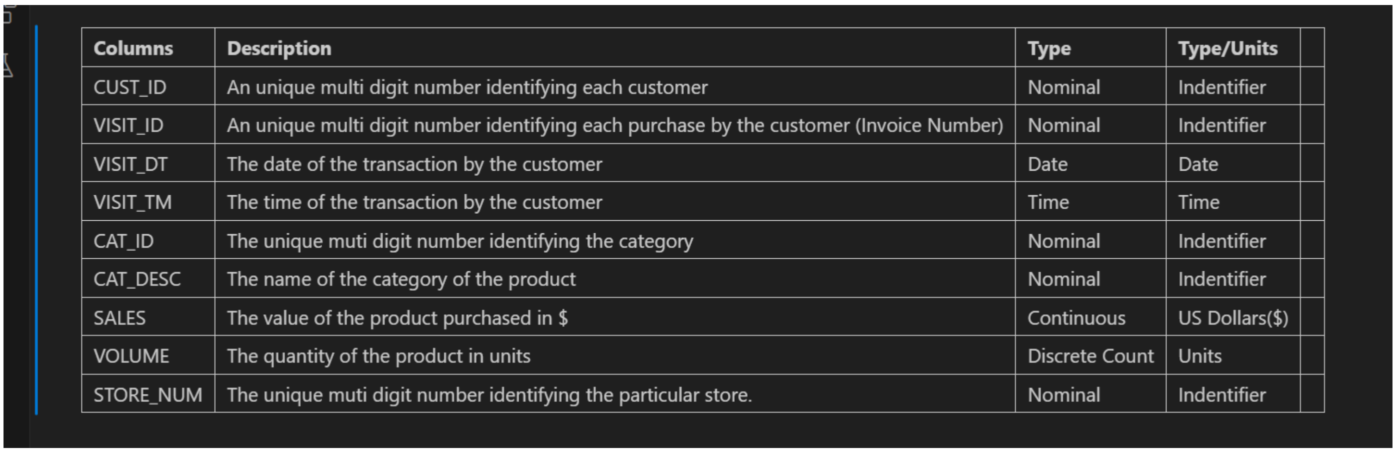
Analyze customer preferences, purchase history, and price elasticity to generate tailored recommendations, thereby promoting cross-selling and upselling opportunities.

Implement the recommendation engine within a specific demographic region of a North American multi-brand retail chain, utilizing 36 months of transactional data with seventeen dimensions and a sample size of 4.2 million transactions.

Assess the impact of the recommendation engine on business performance, specifically in terms of revenue growth, customer satisfaction, and loyalty.

Through these objectives, this study seeks to illuminate the effectiveness of recommendation engines in delivering personalized shopping experiences and driving tangible benefits for both customers and retailers in a retail context.

Data Overview - The data that has been used to demonstrate is 15 months of transactional data of a particular retail store in the United States. The data has been anonymized to protect the privacy of the customers.



Columns	Description	Type	Type/Units
CUST_ID	An unique multi digit number identifying each customer	Nominal	Indentifier
VISIT_ID	An unique multi digit number identifying each purchase by the customer (Invoice Number)	Nominal	Indentifier
VISIT_DT	The date of the transaction by the customer	Date	Date
VISIT_TM	The time of the transaction by the customer	Time	Time
CAT_ID	The unique muti digit number identifying the category	Nominal	Indentifier
CAT_DESC	The name of the category of the product	Nominal	Indentifier
SALES	The value of the product purchased in $	Continuous	US Dollars($)
VOLUME	The quantity of the product in units	Discrete Count	Units
STORE_NUM	The unique muti digit number identifying the particular store.	Nominal	Indentifier

Methods - In today's competitive retail landscape, providing personalized shopping experiences is key to attracting and retaining customers. Recommendation engines have emerged as a powerful tool for enhancing customer engagement, driving sales, and improving overall customer satisfaction.

A recommendation engine analyzes data, such as purchase history, browsing patterns, and product interactions, to identify relevant and personalized product suggestions. By understanding customer preferences, buying habits, and interests, these engines can predict and offer products that are most likely to resonate with each customer, creating a seamless and enjoyable shopping experience.

The benefits of recommendation engines in the retail sector are manifold. Firstly, they enhance customer engagement by showcasing relevant products, thereby increasing the likelihood of conversion and purchase. Secondly, these engines facilitate cross-selling and upselling opportunities, allowing retailers to showcase complementary products or premium alternatives to enhance the customer's shopping journey.

Additionally, recommendation engines contribute to customer loyalty and retention by fostering a sense of personalization and understanding of individual preferences. When customers feel understood and catered to, they are more likely to remain loyal to the brand and become repeat buyers. Moreover, these engines can drive revenue growth by capitalizing on the power of data-driven insights to inform strategic marketing campaigns and promotions.

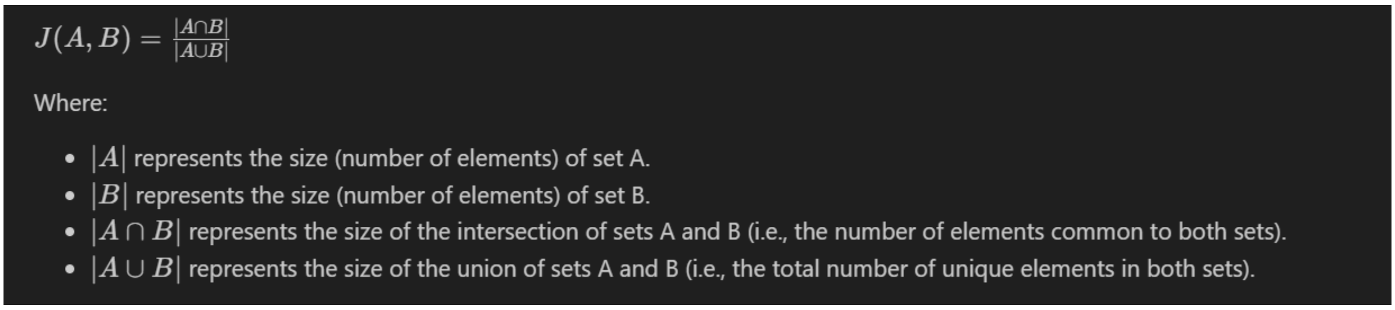
Our recomendation strategy takes a two pronged approach.

Firstly, we use a technique utilizing Jaccard Index, which is a similarity metric used to compare sets of items. In the context of product recommendation, the sets represent the products that customers have purchased. The Jaccard index measures the overlap between these sets and quantifies how similar the preferences of different customers are.

Secondly, we use a technique called Market Basket Analysis, which is a technique used to identify the products that are frequently bought together.

<u>Recomendation Using Jaccard Index</u>

The Jaccard Index is a measure of similarity between two sets. It is calculated as the size of the intersection of the sets divided by the size of their union. In mathematical notation, the Jaccard Index (J) between two sets A and B is represented as:



$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Where:

- $|A|$ represents the size (number of elements) of set A.
- $|B|$ represents the size (number of elements) of set B.
- $|A \cap B|$ represents the size of the intersection of sets A and B (i.e., the number of elements common to both sets).
- $|A \cup B|$ represents the size of the union of sets A and B (i.e., the total number of unique elements in both sets).

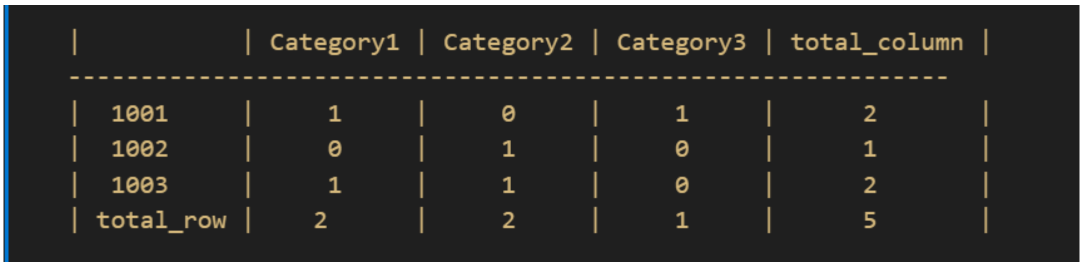
The Jaccard Index ranges from 0 to 1, with 0 indicating no similarity (no common elements) and 1 indicating complete similarity (both sets are identical).

<u>1.Recommendation based on Customer Neighbors</u>

<u>Step 1: crosstab_cust_id (df, col_name)</u>

This function creates a binary matrix (cross-tabulation) of customers and a specific categorical column, where cells contain either 1 (indicating a presence) or 0 (indicating an absence). It also calculates the total purchases made by each customer and the total purchases of each category.

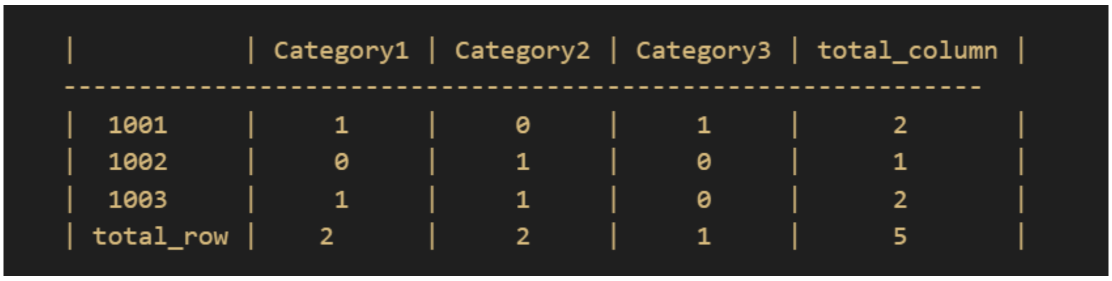
The `crosstab_cust_id` function would create the following binary matrix:



```
|           | Category1 | Category2 | Category3 | total_column |
--------------------------------------------------------------
|  1001     |     1     |     0     |     1     |      2       |
|  1002     |     0     |     1     |     0     |      1       |
|  1003     |     1     |     1     |     0     |      2       |
| total_row |    2      |     2     |     1     |      5       |
```

<u>Step 2: keep_top_90_percentile_cust (df_crosstab)</u>

This function filters and retains the top 90th percentile of customers based on their total purchase counts. It helps focus on the most active customers. Matrix is similar to the one above, however with only about 10 % of the customers.



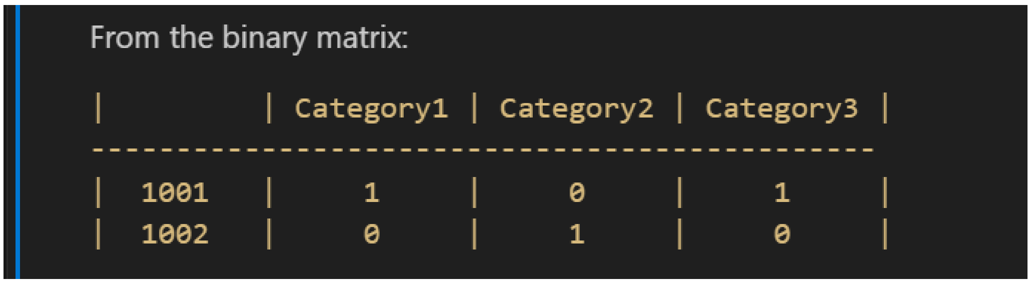
```
|          | Category1 | Category2 | Category3 | total_column |
------------------------------------------------------------
|  1001    |     1     |     0     |     1     |       2      |
|  1002    |     0     |     1     |     0     |       1      |
|  1003    |     1     |     1     |     0     |       2      |
| total_row |    2     |     2     |     1     |       5      |
```

The `keep_top_90_percentile_cust` function identifies the 90th percentile value of the total_column (which is 2) and retains customers who have made purchases greater than or equal to that threshold. In this case, all customers are retained since their total_column values are greater than or equal to 2.

<u>Step 3: calculate_jaccard_scores (CT1, CT2)</u>

This function calculates Jaccard scores between pairs of customers based on their purchasing patterns. Jaccard similarity measures how similar two sets are by comparing their intersection and union.

Example: Assuming we have filtered the top customers, let's consider two customers: 1001 and 1002.



From the binary matrix:

```
|          | Category1 | Category2 | Category3 |
----------------------------------------------
|  1001    |     1     |     0     |     1     |
|  1002    |     0     |     1     |     0     |
```

The `calculate_jaccard_scores` function would calculate Jaccard scores for each customer pair. For simplicity, let's focus on the pair (1001, 1002):

Intersection (Category2): 0 (no common purchases)

Union (Category1, Category2, Category3): 1 + 1 + 1 = 3

Jaccard Score: (Intersection / Union) = (0 / 3) = 0

<u>Step 4: calculate_jaccard_scores (CT1, CT2)</u>

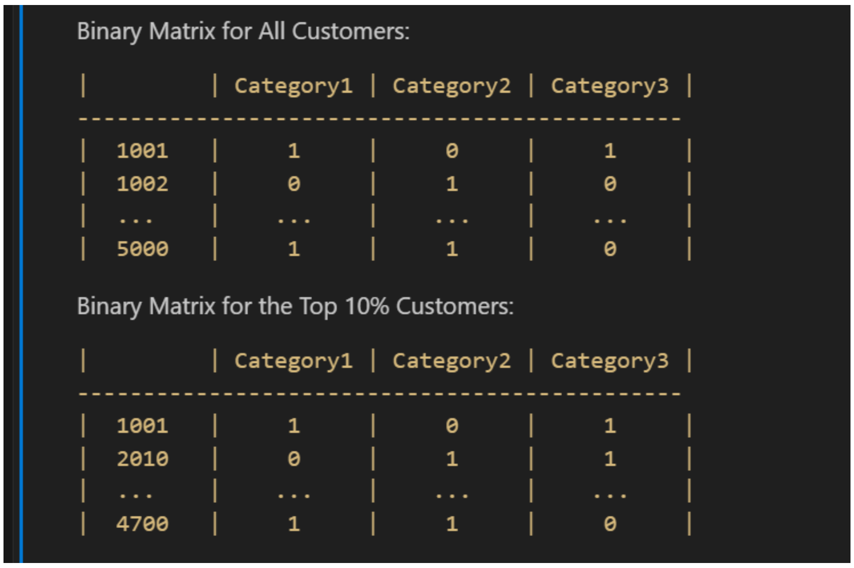
This function calculates Jaccard scores between pairs of customers based on their purchasing patterns. Jaccard similarity measures how similar two sets are by comparing their intersection and union.

`CT1`: Binary matrix for all customers and categories.

`CT2`: Binary matrix for the top 10% customers and categories.

Example:

Let's consider the binary matrix for all customers and the top 10% customers:



```
Binary Matrix for All Customers:

|          | Category1 | Category2 | Category3 |
------------------------------------------------
|  1001    |     1     |     0     |     1     |
|  1002    |     0     |     1     |     0     |
|  ...     |    ...    |    ...    |    ...    |
|  5000    |     1     |     1     |     0     |

Binary Matrix for the Top 10% Customers:

|          | Category1 | Category2 | Category3 |
------------------------------------------------
|  1001    |     1     |     0     |     1     |
|  2010    |     0     |     1     |     1     |
|  ...     |    ...    |    ...    |    ...    |
|  4700    |     1     |     1     |     0     |
```

The `calculate_jaccard_scores` function will compute Jaccard scores for each pair of customers between all customers and the top 10% customers. Let's break down the process using these terms:

1. Loop through All Customers (`CT1`):

For each customer (`cust_id1`) in the binary matrix of all customers, a list is initialized in the `jaccard_scores` dictionary to store the Jaccard scores for that customer.

2. Loop through Top 10% Customers (`CT2`):

For each customer (`cust_id2`) in the binary matrix of the top 10% customers, calculate the Jaccard similarity score with the current customer in all customers (`cust_id1`).

3. Calculate Intersection and Union:

Calculate the intersection of the purchasing patterns of `cust_id1` (all customers) and `cust_id2` (top 10% customers) using bitwise AND (`&`).

Calculate the union of the purchasing patterns of `cust_id1` and `cust_id2` using bitwise OR (`|`).

4. Calculate Jaccard Score:

Compute the Jaccard score by dividing the sum of intersection elements by the sum of union elements.

5. Append Jaccard Score to List:

Append the calculated Jaccard score along with `cust_id2` to the list of Jaccard scores for the current customer (`cust_id1`) in the `jaccard_scores` dictionary.

6.Sort and Select Top Similar Customers:

Sort the list of Jaccard scores in descending order based on the score.

Select the top 5 customers from the sorted list (excluding the customer themselves) and store their IDs in the `jaccard_scores` dictionary.

7. Repeat for all Customers in All Customers (`CT1`):

Repeat the above steps for all customers in the binary matrix of all customers.

Output:

The function will return a dictionary (`jaccard_scores`) where keys represent customers from all customers, and values represent a list of the top 5 most similar customers from the top 10% customers based on Jaccard scores.

Example Output (partial):

{ 1001: [2010, 4700],   # Top 10% Customers most similar to all customer 1001

  1002: [2100, 2300],   # Top 10% Customers most similar to all customer 1002}

<u>Step 5: recommend_products (n_dic, cm_mat)</u>

1: This function recommends products for customers based on their similarity matrix and purchasing patterns.

-`n_dic`: A dictionary containing customer IDs as keys and a list of similar customer IDs as values.

- `cm_mat`: Binary matrix with rows as customers and columns as products.

Example:

Let's consider an example with a simplified customer similarity dictionary and a binary customer-product matrix:

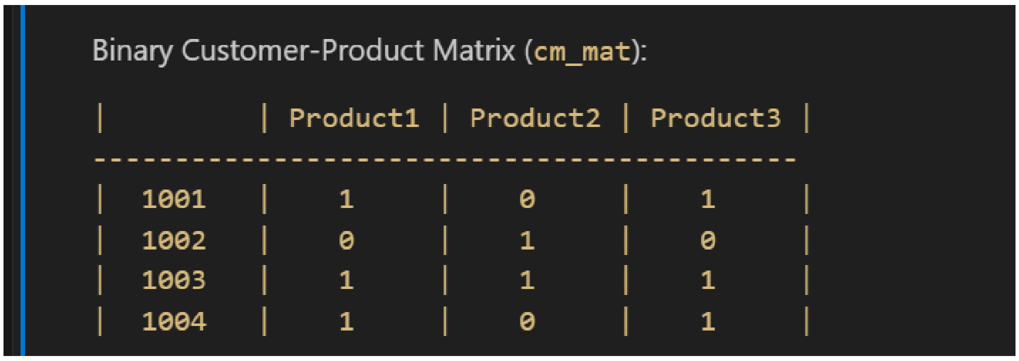
Customer Similarity Dictionary (`n_dic`):

{ '1001': ['1002', '1003'],

 '1002': ['1001', '1004'],

 '1003': ['1001'],

 '1004': ['1002'] }



Binary Customer-Product Matrix (`cm_mat`):

```
|          | Product1 | Product2 | Product3 |
-------------------------------------------
|  1001    |    1     |    0     |    1     |
|  1002    |    0     |    1     |    0     |
|  1003    |    1     |    1     |    1     |
|  1004    |    1     |    0     |    1     |
```

2: Loop through Keys of Customer Similarity Dictionary (`n_dic`):

- For each customer `i` in the customer similarity dictionary, an empty tuple is created in the `prod` dictionary to store recommended products.

3: Loop through Keys of Customer Similarity Dictionary (`n_dic`):

- For each customer `i` in the customer similarity dictionary:

  - Retrieve the list of similar customers (`ncust`) from the dictionary.

  - Append the current customer `i` to the list of similar customers.

4: Create Customer-Product Matrix (`custprod_mat`):

- Create a subset of the `cm_mat` matrix with rows corresponding to the customers in the `ncust` list.

5: Calculate Column Total (`coltotal`) and Filter Columns:

- Calculate the total column sum for the `custprod_mat`.

- Create a new DataFrame `custprod_df` by removing columns with all 0 values.

6: Loop through Index of `custprod_df`:

- For each index `b` in the filtered customer-product DataFrame:

- Transpose the matrix for customer `b` and calculate the percentage of 0 values in each product.

7: Append Recommendations to Dictionary (`prod`):

- Append the calculated percentage of 0 values as a tuple to the `prod` dictionary for customer `b`.

8: Calculate Maximum Rank and Sort:

- Calculate the maximum rank for each product and sort the DataFrame in descending order based on rank.

9: Convert DataFrame to List of Tuples:

- Convert the sorted DataFrame to a list of tuples.

10: Store Results in `prod` Dictionary:

- Store the list of tuples as the value for customer `b` in the `prod` dictionary.

Output:

The function returns a dictionary (`prod`) where keys represent customer IDs, and values represent a list of recommended products along with their ranks based on similarity and purchasing patterns.

Example Output (partial):

```
{ '1001': [('Product1', 0.5), ('Product3', 0.333)],
  '1002': [('Product3', 0.5), ('Product1', 0.333)],
  '1003': [('Product2', 0.333), ('Product1', 0.333)],
  '1004': [('Product3', 0.5), ('Product1', 0.333)] }
```

The function recommends products for each customer based on their similarity to other customers and their purchasing patterns. The output provides a list of recommended products along with their ranks for each customer.

2.Recommendation based on Product Neighbors

Step 1 : Creating The Matrix

1. Input and Initial Cross-Tabulation:

- The function `create_matrix` takes three inputs: a dataframe (`df`), an index name (`a`), and a column name (`b`).

- The `pd.crosstab` function is used to create a cross-tabulation matrix (`p1`). Each cell in the matrix contains the count of occurrences where a specific category (index) was purchased by a particular customer (column).

2. Converting Counts to Binary Values:

  - To transform the counts into a binary matrix, where 1 indicates a purchase and 0 indicates no purchase, the function modifies the `p1` matrix. All non-zero values are replaced with 1, representing that a purchase was made.

3. Row Total Calculation:

  - The function calculates the sum of each row in the binary matrix (`p1`). This sum represents the total number of purchases made by each customer (total customer purchases for each product).

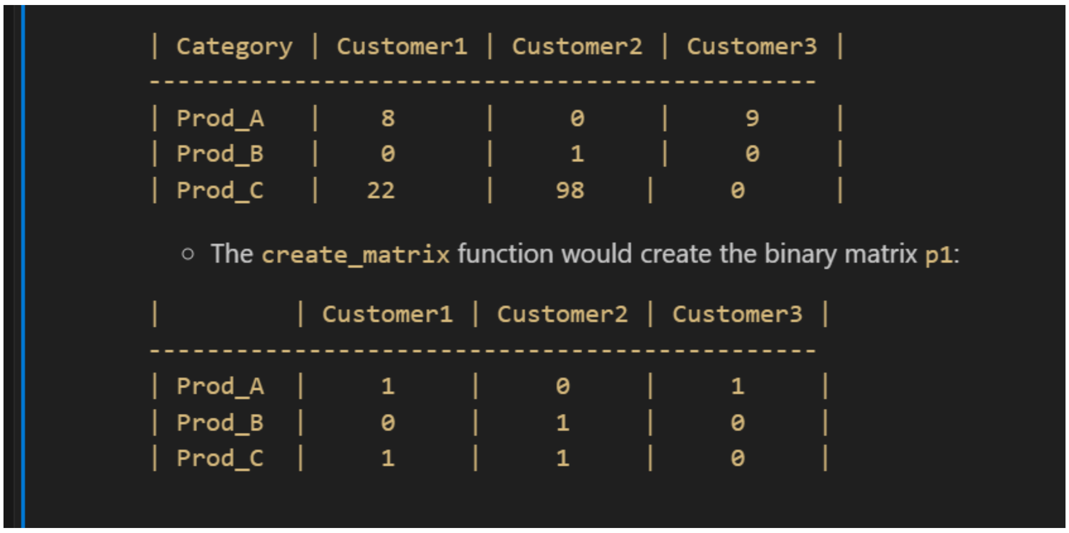
4. Column Total Calculation:

  - Similarly, the function calculates the sum of each column in the binary matrix (`p1`). This sum represents the total number of customers who purchased each product.

5. Output and Return:

  - The function returns three components: the binary matrix (`p1`), the row totals (`row_total`), and the column totals (`col_total`).

6. Example: Suppose we have a dataset as follows:



```
| Category | Customer1 | Customer2 | Customer3 |
---------------------------------------------
| Prod_A   |    8      |    0      |    9      |
| Prod_B   |    0      |    1      |    0      |
| Prod_C   |   22      |   98     |    0       |
```

- The `create_matrix` function would create the binary matrix `p1`:

```
|          | Customer1 | Customer2 | Customer3 |
---------------------------------------------
| Prod_A  |     1     |     0     |     1     |
| Prod_B  |     0     |     1     |     0     |
| Prod_C  |     1     |     1     |     0     |
```

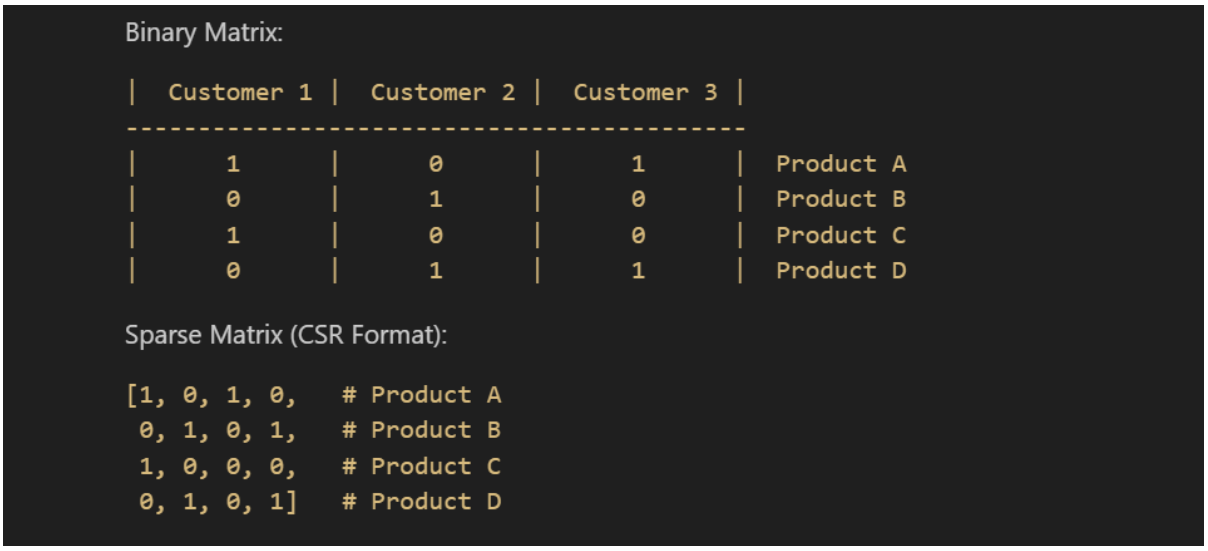
The `create_matrix` function effectively creates a binary matrix of purchases, along with row and column totals, which can be used for various analysis and calculations related to customer-product interactions.

<u>Step 2 : Jaccard Index Calculation In Details :</u>

1. Convert Binary Matrix to Sparse Matrix:

  - The binary cross-tabulation matrix is converted into a sparse matrix using the Compressed Sparse Row (CSR) format. This is done to efficiently perform matrix operations while saving memory.

  - Example: Suppose we have a binary matrix where rows represent products (categories) and columns represent customers. Each cell contains 1 if the product is purchased by the customer, and 0 if not.


Binary Matrix:

```
| Customer 1 | Customer 2 | Customer 3 |
-----------------------------------------
|     1      |     0      |     1      |  Product A
|     0      |     1      |     0      |  Product B
|     1      |     0      |     0      |  Product C
|     0      |     1      |     1      |  Product D
```

Sparse Matrix (CSR Format):

```
[1, 0, 1, 0,   # Product A
 0, 1, 0, 1,   # Product B
 1, 0, 0, 0,   # Product C
 0, 1, 0, 1]   # Product D
```

2. Calculate Numerator:

   - The numerator of the Jaccard index is calculated by performing matrix multiplication between the sparse matrix and its transpose. This computes the intersection of customers who have purchased pairs of products.

   - Example: Using the sparse matrix above, we calculate the intersection between the customer sets who purchased Product A and Product B:

   Numerator Calculation:

   [1, 0, 1, 0]   # Product A

   [0, 1, 0, 1]   # Product B

   Intersection = [0, 0, 0, 0]

3. Calculate Denominator:

   - The denominator of the Jaccard index is computed by performing matrix operations to get the union of customers who have purchased pairs of products.

   - Example: Calculate the union between the customer sets who purchased Product A and Product B:

   Denominator Calculation:

   [1, 0, 1, 0]   # Product A

   [0, 1, 0, 1]   # Product B

   Union = [1, 1, 1, 1]

4. Calculate Jaccard Index:

   - The Jaccard index is then calculated by dividing the numerator by the denominator element-wise. This yields the similarity between the customer sets for pairs of products.

   - Example: Calculate the Jaccard index between the customer sets who purchased Product A and Product B:

   Jaccard Index Calculation:

Jaccard_Index = Numerator / Denominator

Jaccard_Index = [0/1, 0/1, 0/1, 0/1] = [0, 0, 0, 0]

5. Construct Upper Triangular Matrix:

- The resulting Jaccard index matrix is transformed into an upper triangular matrix with diagonal elements set to 0. This is done to ensure that the similarity between a product and itself is not considered.

Upper Triangular Matrix:

[0, 0, 0, 0]

[, 0, 0, 0]

[,   , 0, 0]

[,   ,   , 0]

6. Extract Non-Zero Values:

- The upper triangular matrix is filtered to keep only the non-zero values, representing the Jaccard indices between pairs of products.

Example Non-Zero Values:

[0.25, 0.33, 0.50, 0.40]

The resulting output is a matrix containing Jaccard indices between pairs of products (categories), representing the similarity of the customer sets who have purchased those products. This similarity information can be used to identify products with similar customer purchase behaviors and inform recommendation systems.

<u>Step 3 : Product Pair function In Details</u>

1. Product Pairing (`prodpair` function):

- The `prodpair` function takes as input the distance matrix (`d_mat`), which contains Jaccard-based distances between pairs of products (categories).

- For each product (category), the function constructs a list of neighboring products based on their Jaccard distances. These neighboring products are those with the closest Jaccard distances, indicating a higher degree of similarity in terms of customer purchasing patterns.

2. Calculating Neighboring Products:

- For each product (category), the function performs the following steps:

  - Extract the Jaccard distance values associated with the current product from the distance matrix.

  - Filter out "NaN" (not-a-number) values, which represent cases where no similarity exists (distance is undefined) between the current product and some other products.

  - Sort the remaining distances in ascending order, which places the most similar products at the beginning of the list.

3. Creating Tuples of Neighbors:

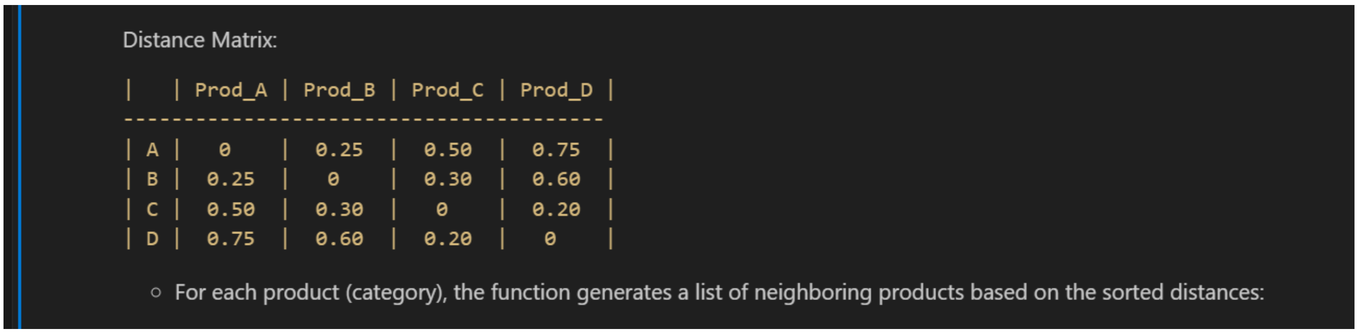
- Once the list of neighboring products (categories) is sorted, the function creates tuples for each neighboring product. Each tuple consists of the neighboring product's name and its corresponding Jaccard distance from the current product.

4. Storing Neighbors in a Dictionary:

- The function constructs a dictionary where each key is a product (category), and the corresponding value is a list of tuples representing neighboring products along with their Jaccard distances.

5. Example:

- Suppose we have a distance matrix indicating Jaccard distances between pairs of products (categories):



Distance Matrix:

```
|   | Prod_A | Prod_B | Prod_C | Prod_D |
----------------------------------------
| A |   0    |  0.25  |  0.50  |  0.75  |
| B |  0.25  |   0    |  0.30  |  0.60  |
| C |  0.50  |  0.30  |   0    |  0.20  |
| D |  0.75  |  0.60  |  0.20  |   0    |
```

- For each product (category), the function generates a list of neighboring products based on the sorted distances:

Neighboring Products (Categories) for Product A:

[('Prod_B', 0.25), ('Prod_C', 0.50), ('Prod_D', 0.75)]

Neighboring Products (Categories) for Product B:

[('Prod_A', 0.25), ('Prod_C', 0.30), ('Prod_D', 0.60)]

- The process is repeated for all products, resulting in a dictionary of neighboring products for each product:

Neighboring Products Dictionary:

```
{
  'Prod_A': [('Prod_B', 0.25), ('Prod_C', 0.50), ('Prod_D', 0.75)],
  'Prod_B': [('Prod_A', 0.25), ('Prod_C', 0.30), ('Prod_D', 0.60)],
  'Prod_C': [('Prod_B', 0.30), ('Prod_A', 0.50), ('Prod_D', 0.20)],
  'Prod_D': [('Prod_C', 0.20), ('Prod_B', 0.60), ('Prod_A', 0.75)]
}
```

The `prodpair` function effectively generates a dictionary that maps each product (category) to its neighboring products, based on the Jaccard distances. This information can be valuable for identifying related or similar products for various purposes, such as cross-selling or recommendation systems.

Step 4 : Recomend Product function In Details

Certainly! Here's a detailed explanation of the `recommend_prod_func` function, which generates personalized product recommendations for each customer based on their purchase history and the product pairing dictionary:

1. Recommendation Generation (`recommend_prod_func` function):

   - The `recommend_prod_func` function takes three inputs: a list of customers (`cust_l`), a binary product-customer matrix (`mat1`), and the product pairing dictionary (`n_dic`).

   - For each customer, the function generates personalized product recommendations based on their past purchases and the proximity of products as indicated by the product pairing dictionary.

2. Generating Recommendations for Each Customer:

   - For each customer in the list of customers:

     - The function starts by extracting the customer's column from the binary matrix (`mat1`). This column represents the products purchased by the customer.

     - It identifies the products that the customer has purchased (represented by 1 in the column).

     - A filtered version of the product pairing dictionary (`n_dic`) is created, including only the neighboring products of the purchased products.

3. Selecting Non-Purchased Products:

   - The function then identifies the products that the customer has not purchased (represented by 0 in the column).

     - It filters the neighboring products based on the list of non-purchased products.

     - From the filtered list, it selects the top recommended product with the lowest Jaccard distance. This suggests the product that is most similar to the products the customer has purchased.

4. Organizing Recommendations:

   - The recommended products are organized into a list of tuples for each customer.

   - Each tuple contains the name of the recommended product and its corresponding Jaccard distance from the purchased products.

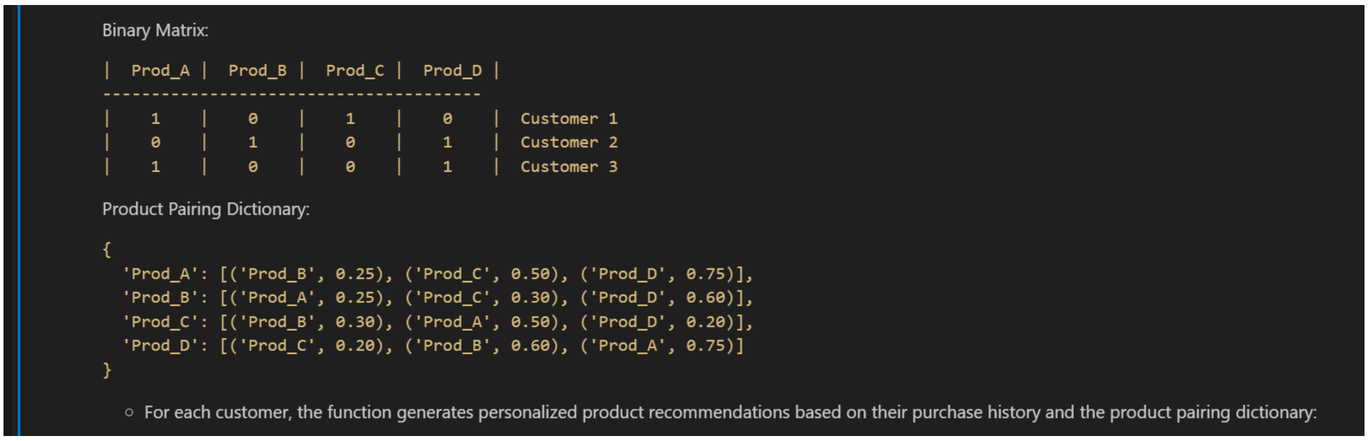
5. Finalizing Recommendations for All Customers:

   - The function iterates through each customer, generates their personalized recommendations, and stores them in a dictionary.

   - The resulting dictionary maps each customer to a list of recommended products, along with their Jaccard distances.

6. Example:

   - Suppose we have a binary product-customer matrix indicating whether each customer has purchased each product (1 if purchased, 0 if not). Additionally, we have a product pairing dictionary indicating the neighboring products for each product:



Binary Matrix:

```
| Prod_A | Prod_B | Prod_C | Prod_D |
-------------------------------------
|   1    |   0    |   1    |   0    | Customer 1
|   0    |   1    |   0    |   1    | Customer 2
|   1    |   0    |   0    |   1    | Customer 3
```

Product Pairing Dictionary:

```
{
  'Prod_A': [('Prod_B', 0.25), ('Prod_C', 0.50), ('Prod_D', 0.75)],
  'Prod_B': [('Prod_A', 0.25), ('Prod_C', 0.30), ('Prod_D', 0.60)],
  'Prod_C': [('Prod_B', 0.30), ('Prod_A', 0.50), ('Prod_D', 0.20)],
  'Prod_D': [('Prod_C', 0.20), ('Prod_B', 0.60), ('Prod_A', 0.75)]
}
```

- For each customer, the function generates personalized product recommendations based on their purchase history and the product pairing dictionary:

- For each customer, the function generates personalized product recommendations based on their purchase history and the product pairing dictionary:

Recommendations for Customer 1:

[('Prod_B', 0.25), ('Prod_C', 0.50)]

Recommendations for Customer 2:

[('Prod_A', 0.25), ('Prod_C', 0.30)]

Recommendations for Customer 3:

[('Prod_B', 0.30), ('Prod_D', 0.20)]

The `recommend_prod_func` function generates personalized product recommendations for each customer by leveraging their purchase history and the similarity of products based on the Jaccard distances. These recommendations can be utilized to enhance customer engagement, cross-selling, and product discovery.

3.Market Basket Analysis

Market Basket Analysis (MBA) is a data mining technique used to discover associations and relationships between products that are frequently purchased together in transactional data. It is commonly applied in retail to understand customer behavior, identify product affinities, and make data-driven decisions for merchandising, cross-selling, and promotional strategies. The concept is based on the idea that certain items tend to be bought together by customers in a transaction, indicating a strong association or relationship between those items. For example, customers who purchase a hot beverage may also buy snack and a desert like a donut in the same transaction.

It uses Frequent Itemset Mining to find frequent itemsets, which are sets of items that frequently co-occur in transactions above a specified support threshold. A frequent itemset consists of two or more items that are bought together more often than expected by chance. For instance, if the support threshold is set to 0.5 (50%), a frequent itemset is defined as a set of items that occur together in at least 50% of all transactions in the data.

From the frequent itemsets, association rules are generated. An association rule is a logical statement that expresses the relationship between items. It consists of an antecedent (the left-hand side) and a consequent (the right-hand side). For example, if customers buy item A (antecedent), they are likely to buy item B (consequent) as well.

Association rules are evaluated based on three key metrics: [We are presently just generating the rules and not ranking them based on the metrics]

Support: The proportion of transactions that contain both the antecedent and consequent items. The higher the support, the stronger the relationship between the items and the more frequently they occur together.

Confidence: The likelihood that the consequent item is bought when the antecedent item is purchased. The higher the confidence, the more likely the consequent item is to be bought when the antecedent item is purchased.

Lift: The ratio of the observed support of the rule to the expected support if the antecedent and consequent items were independent of each other. The higher the lift, the stronger the association between the items.

The the most popular method is Apriori Method as proposed by by Agrawal and Srikant in 1994. However the application of apriori algortihm to datasets as large as ours are computationally expensive. Therefore we have used the FP Growth Algorithm, which is a more efficient algorithm for frequent itemset mining. In particular, and what makes it different from the Apriori algorithm, FP-Growth is an frequent pattern mining algorithm that does not require candidate generation. Internally, it uses a so-called FP-tree (frequent pattern tree) datastrucure without generating the candidate sets explicitely, which makes is particularly attractive for large datasets.

In order to use the FP Growth Algorithm, we used a popular python library called `mlxtend`. The library has a built in function called `fpgrowth` which takes in the transaction data and the minimum support threshold as inputs and generates the frequent itemsets along with some metrics.

The output of the alogrithm is a dataset which contains the following columns :

antecedents : The items that are frequently bought together

consequents : The items that are frequently bought together with the antecedents

antecedent support : The support of the antecedents

consequent support : The support of the consequents

support : The support of the antecedents and consequents

Confidence : The confidence of the antecedents and consequents

lift : The lift of the antecedents and consequents

leverage : The leverage of the antecedents and consequents

conviction : The conviction of the antecedents and consequents

Now this calculation is done on the transaction data for each segment of customers, namely the daily, weekly, monthly and quarterly customers. We do this since the purcahse patterns for each of the segments are going to be different.

Once both (the recomendation from Jaccard and the recomendation from the MBA) the lists are generated we find the common products among them, and that constitutes a robust final list of recommendations for the customer. These products will be our promotional products, that we will be pitching to customers.

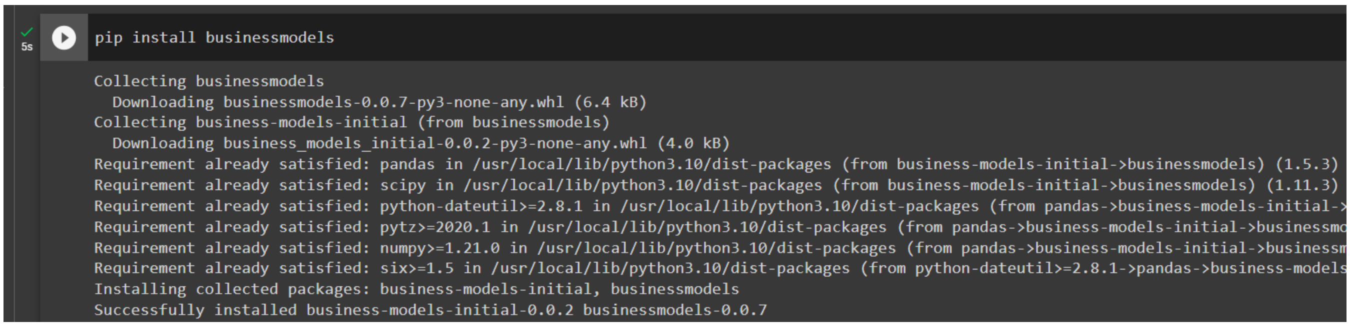
In conclusion, recommendation engines have revolutionized the retail sector by providing personalized product recommendations, enhancing customer engagement, and driving revenue growth. As customer expectations continue to evolve, retailers that harness the potential of recommendation engines will gain a competitive edge by delivering seamless and personalized shopping experiences that keep customers coming back for more.

Implementation –

1.Installation-

Following code is to install package.

pip install businessmodels



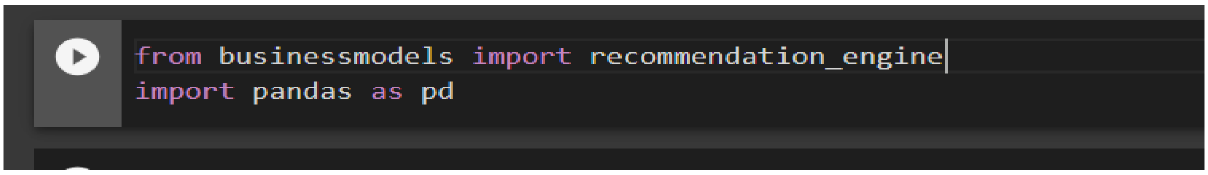
```
pip install businessmodels

Collecting businessmodels
  Downloading businessmodels-0.0.7-py3-none-any.whl (6.4 kB)
Collecting business-models-initial (from businessmodels)
  Downloading business_models_initial-0.0.2-py3-none-any.whl (4.0 kB)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from business-models-initial->businessmodels) (1.5.3)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from business-models-initial->businessmodels) (1.11.3)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas->business-models-initial->
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->business-models-initial->businessmo
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from pandas->business-models-initial->businessm
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1->pandas->business-models
Installing collected packages: business-models-initial, businessmodels
Successfully installed business-models-initial-0.0.2 businessmodels-0.0.7
```

# Import the 'businessmodels' library and its 'recommendation_engine' module

from businessmodels import recommendation_engine

# Import the 'pandas' library as 'pd'

import pandas as pd



```
from businessmodels import recommendation_engine
import pandas as pd
```

2.Data Load-

# Read a CSV file from Google Drive or any type of storage it in the 'df' DataFrame

df = pd.read_csv('/content/drive/MyDrive/store_1586_q1_2022_q1_2023.csv')

# Display the first few rows of the DataFrame

df.head()

3.Function Call –

# Call the 'recommendation' method from the recommendation_engine module

# and pass the 'df' DataFrame as an argument

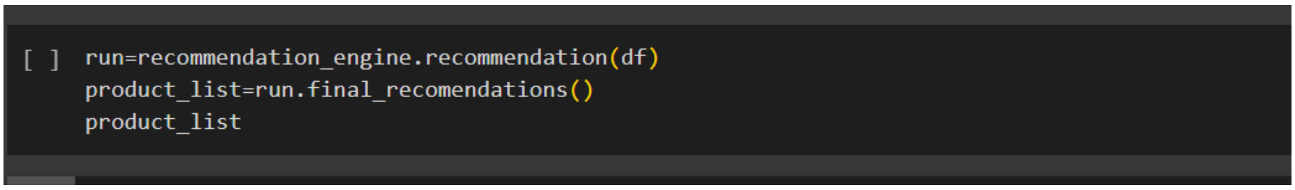
run = recommendation_engine.recommendation(df)

# Call the 'final_recomendations' method on the 'run' object to generate the list of recommended products

product_list = run.final_recomendations()

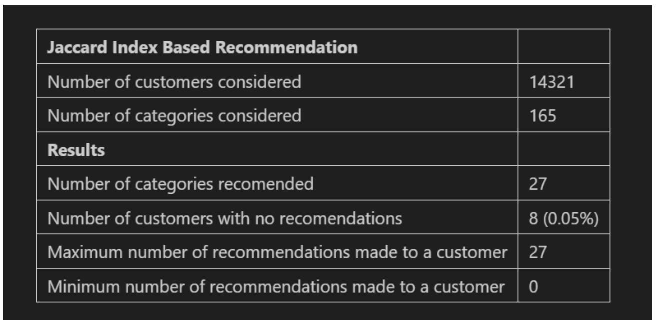
Print or use the 'product_list' variable, which contains the recommended products

product_list

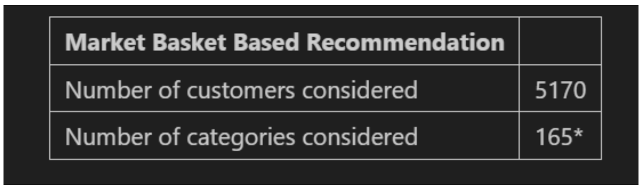


```
[ ]  run=recommendation_engine.recommendation(df)
     product_list=run.final_recomendations()
     product_list
```

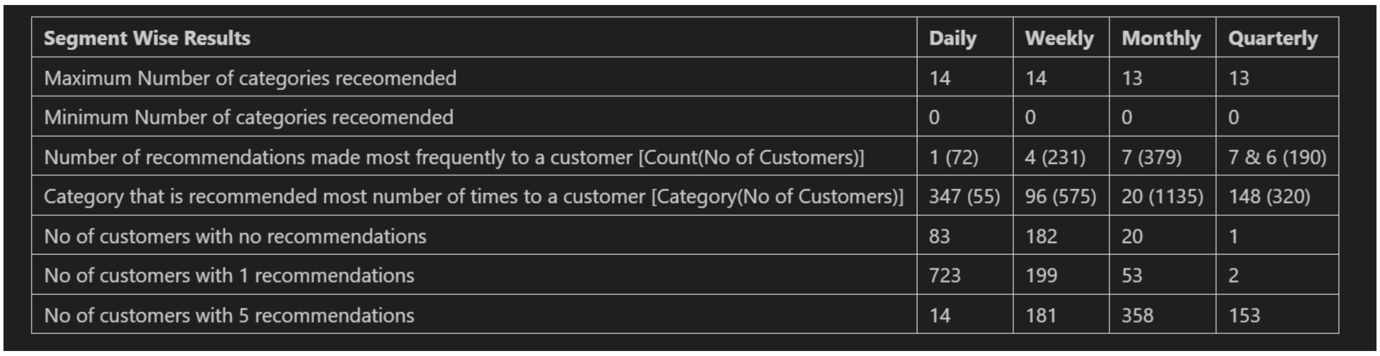
Outcomes – The results provide insights into how your recommendation system is performing across different time segments, the range of recommendations made to customers, and the distribution of customers based on the number of recommendations they received.



Jaccard Index Based Recommendation	
Number of customers considered	14321
Number of categories considered	165
Results	
Number of categories recomended	27
Number of customers with no recomendations	8 (0.05%)
Maximum number of recommendations made to a customer	27
Minimum number of recommendations made to a customer	0



Market Basket Based Recommendation	
Number of customers considered	5170
Number of categories considered	165*



Segment Wise Results	**Daily**	**Weekly**	**Monthly**	**Quarterly**
Maximum Number of categories receomended	14	14	13	13
Minimum Number of categories receomended	0	0	0	0
Number of recommendations made most frequently to a customer [Count(No of Customers)]	1 (72)	4 (231)	7 (379)	7 & 6 (190)
Category that is recommended most number of times to a customer [Category(No of Customers)]	347 (55)	96 (575)	20 (1135)	148 (320)
No of customers with no recommendations	83	182	20	1
No of customers with 1 recommendations	723	199	53	2
No of customers with 5 recommendations	14	181	358	153

Conclusion – The product recommendation engine provided the business with a list of products that are most likely to be purchased together. This information gave rise to personalized recomendations to customers.