**Recommendation Model Code Explanation**

**Kernel 1**

import pandas as pd

**Code Explanation**:  
This imports the pandas library, a powerful and widely-used tool for data manipulation in Python. pandas provides the DataFrame structure, which allows for efficient data manipulation, such as filtering, merging, and aggregation.

**Kernel 2**

%pip install mlxtend

**Code Explanation**:  
This command installs the mlxtend library. mlxtend is a machine learning extension library that provides utilities for association rule mining, like Apriori and association rules generation. These functions will later be used to build a recommendation model by identifying frequently co-purchased items.

**Kernel 2 Output**



**Output Explanation**:  
The output shows that the required library (mlxtend) is already installed, along with its dependencies (numpy, scipy, pandas, etc.). This ensures that all necessary libraries for the recommendation model are available.

**Kernel 3**

orders = pd.read\_csv("/content/orders.csv")

**Code Explanation**:  
Here, the orders.csv file is loaded into a pandas DataFrame called orders. This file likely contains data related to customer orders, including information such as the order time, user ID, and order ID. The DataFrame will be used to manipulate and analyze order-related data.

**Kernel 4**

orders["day\_hour"] = [f"{day}-{hour}" for day, hour in zip(orders["order\_dow"], orders["order\_hour\_of\_day"])]

**Code Explanation**:  
This line creates a new column called day\_hour, which concatenates the order\_dow (day of the week the order was placed) and order\_hour\_of\_day (hour of the day). This feature is useful for analyzing purchasing patterns, such as peak shopping hours on specific days.

**Kernel 5**

orders["user\_day"] = [f"{user}-{day}" for user, day in zip(orders["user\_id"], orders["order\_dow"])]

**Code Explanation**:  
This line adds a new column user\_day to the orders DataFrame. It combines the user\_id (unique user identifier) and order\_dow (day of the week). This feature helps in understanding user-specific behavior by analyzing on which days certain users place orders.

**Kernel 6**

orders = orders[orders["eval\_set"]=="prior"]

**Code Explanation**:  
This filters the orders DataFrame to keep only those rows where the eval\_set column has the value "prior". This typically refers to past orders, which are used for training recommendation systems. The "prior" orders contain historical purchase data crucial for building the recommendation model.

**Kernel 7**

order\_products = pd.read\_csv("/content/order\_products\_\_prior.csv")

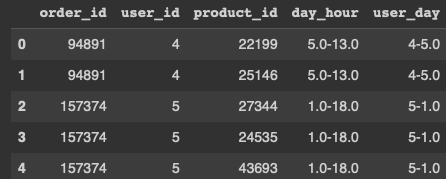
**Code Explanation**:  
This loads another dataset (order\_products\_\_prior.csv) into a DataFrame named order\_products. This file probably contains information about the products purchased in each order. It includes columns like order\_id, product\_id, and add\_to\_cart\_order.

**Kernel 8**

df = pd.merge(orders, order\_products, how="inner", on="order\_id")[["order\_id", "user\_id", "product\_id", "day\_hour", "user\_day"]]

df.head()

**Code Explanation**:  
Here, the orders DataFrame is merged with order\_products on the order\_id column, resulting in a unified dataset with both order details and product information. Only a subset of columns (order\_id, user\_id, product\_id, day\_hour, user\_day) is retained. This merged DataFrame (df) will be used to track which users bought which products and when.

****

**Output Explanation**:  
The head() function displays the first five rows of the merged DataFrame. Each row contains:

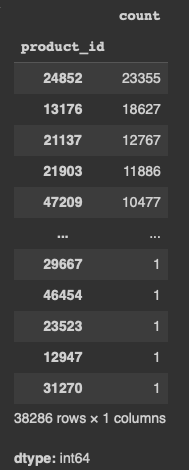
* order\_id: Unique identifier for each order.
* user\_id: Unique identifier for each user.
* product\_id: Unique identifier for each product purchased.
* day\_hour: The day and hour when the order was placed.
* user\_day: The combination of user ID and day of the week.

**Kernel 9**

df["product\_id"].value\_counts()

**Code Explanation**:  
This line counts the frequency of each product\_id in the dataset. This is essential for identifying the most frequently purchased products, which can help in understanding popular items and forming product associations.

**Output Explanation**:  
The output displays the product IDs along with their respective purchase counts. For example:



This shows that product 24852 has been purchased 23,355 times, while product 13176 has been bought 18,627 times, and so on. Understanding popular products is key in building association rules.

**Kernel 10**

df["product\_id"].value\_counts().mean()

**Code Explanation**:  
This calculates the mean number of times a product has been purchased. This value gives an overall understanding of how often, on average, products are bought.

**Output Explanation**:

41.4017917776733

This means that, on average, each product is purchased around 41 times. This helps set expectations for normal product frequency.

**Kernel 11**

import statsmodels.stats.api as sms

low\_conf, up\_conf = sms.DescrStatsW(df["product\_id"].value\_counts()).tconfint\_mean()

print(f"Lower Confidence Interval: {low\_conf:.0f}")

print(f"Upper Confidence Interval: {up\_conf:.0f}")

**Code Explanation**:  
This code computes a confidence interval for the mean product purchase count using the DescrStatsW class from the statsmodels library. Confidence intervals provide a range of values within which the true mean is likely to lie. Here, it calculates the upper and lower bounds of the interval, giving a sense of variability in the product purchase data.

**Output Explanation**:

Lower Confidence Interval: 39

Upper Confidence Interval: 44

The output shows that the average product purchase count is likely between 39 and 44. This range helps in distinguishing between frequently purchased and less popular products.

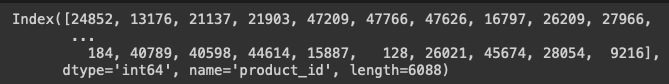
**Kernel 12**

important\_products = df["product\_id"].value\_counts()[df["product\_id"].value\_counts() > low\_conf].index

important\_products

**Code Explanation**:  
This code selects products that are purchased more frequently than the lower confidence interval bound (low\_conf), labeling them as "important products." These products will be the focus of the recommendation model, as they represent the more significant items in the dataset.

**Output Explanation**:  
The output shows a list of product IDs that are classified as important. For example:



Here, 6,088 products have been classified as important based on their purchase frequency.

**Kernel 13**

df["user\_id"].value\_counts()

**Code Explanation**:  
This counts the number of orders placed by each user (user\_id). This information is crucial for analyzing customer behaviour and identifying highly active users.

**Output Explanation**:  
The output shows the number of orders placed by each user. For example:



This indicates that user 6710 has placed 435 orders, while user 74315 has placed 391 orders, and so on.

**Kernel 14**

df = df[df["product\_id"].isin(important\_products)]

df.shape

**Code Explanation**:  
This filters the df DataFrame to retain only rows where the product\_id is in the list of important products. The .shapefunction is used to display the dimensions of the resulting DataFrame.

**Output Explanation**:

(1343642, 5)

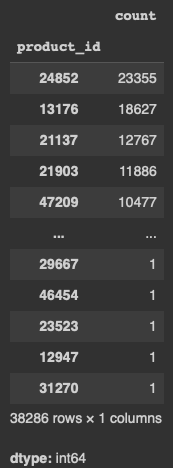
This indicates that the filtered DataFrame has 1,343,642 rows and 5 columns, meaning there are over a million transactions involving important products.

**Kernel 15**

**df["product\_id"].value\_counts()**

**Code Explanation:**This kernel counts the occurrences of each product\_id in the DataFrame, showing the most frequently purchased products. The .head() function displays the top 5 most purchased products, allowing you to focus on the items that are most popular among users.

**Output Explanation**:



In the output:

* 24852 has been purchased 23,355 times.
* 13176 has been purchased 18,627 times.
* 21137 has been purchased 12,767 times.
* 21903 has been purchased 11,886 times.
* 47209 has been purchased 10,477 times.

This step gives you insight into the top products, which can be critical for further analysis or recommendation generation, as they represent the most frequently purchased items in the dataset.

**Kernel 16**

low\_conf, up\_conf = sms.DescrStatsW(df["user\_id"].value\_counts()).tconfint\_mean()

print(f"Lower Confidence Interval: {low\_conf:.0f}")

print(f"Upper Confidence Interval: {up\_conf:.0f}")

**Code Explanation**:  
Here, the confidence interval is calculated for the number of orders per user. The results provide insight into how many orders an "average" user places.

**Output Explanation**:

Lower Confidence Interval: 21

Upper Confidence Interval: 22

The average user places between 21 and 22 orders. This can help determine which users are more active than average.

**Kernel 17**

important\_baskets = df["user\_id"].value\_counts()[df["user\_id"].value\_counts() > low\_conf].index

important\_baskets

**Code Explanation**:  
This selects users who have placed more orders than the lower bound of the confidence interval. These users are considered "important" for the analysis, and their purchase behaviour will be used to generate recommendations.

**Output Explanation**:

Index([6710, 60694, 52008, 79106, ...], dtype='int64', name='user\_id', length=20029)

The output shows 20,029 important users who have placed more than 21 orders, making them key users for the model.

**Kernel 18**

df = df[df["user\_id"].isin(important\_baskets)]

df.shape

**Code Explanation**:  
Filters the df DataFrame to retain only the rows corresponding to important users. The .shape function displays the dimensions of the resulting DataFrame.

**Output Explanation**:

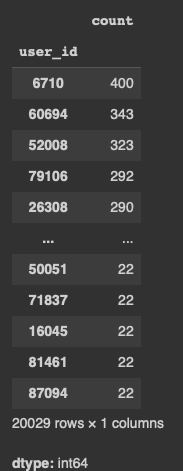
(954797, 5)

This shows that the filtered dataset contains 954,797 transactions involving important users.

**Kernel 19**

df["user\_id"].value\_counts()

**Code Explanation**:  
This counts the number of transactions per important user. This step is useful for understanding how often these key users are interacting with the platform.

****

**Output Explanation**:  
The output displays the number of transactions per important user. For example:

User 6710 has made 400 transactions, user 60694 has made 343, and so on.

**Kernel 20**

basket = df.groupby(["user\_id", "product\_id"])["order\_id"].count().unstack().notnull()

basket

**Code Explanation**:  
This code creates a binary matrix (called a "basket" or "user-item matrix") where the rows represent users, and the columns represent products. The values in the matrix are True if a user has purchased a product and False if they haven't. This matrix is crucial for collaborative filtering and market basket analysis, which helps to identify patterns in co-purchased items.

**Output Explanation**:  
The output is a large matrix, where each row corresponds to a user and each column corresponds to a product. For example:



Each True value indicates that a specific user purchased a specific product. This matrix will serve as input for frequent itemset mining.

**Kernel 21**

from mlxtend.frequent\_patterns import apriori, association\_rules

**Code Explanation**:  
This imports the apriori and association\_rules functions from the mlxtend library. These functions are essential for mining frequent itemsets and generating association rules, which help in identifying relationships between products that are often bought together.

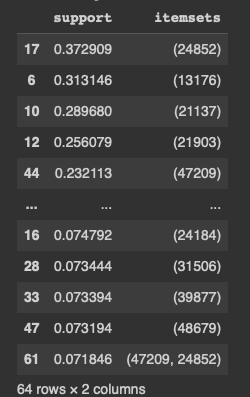
**Kernel 22**

frequent\_itemsets = apriori(basket, min\_support=0.07, use\_colnames=True)

frequent\_itemsets.sort\_values("support", ascending=False)

**Code Explanation**:  
The Apriori algorithm is applied to the user-item matrix (basket) to find frequent itemsets, with a minimum support threshold of 0.07 (i.e., items purchased by at least 7% of users). The support metric indicates the proportion of users who bought the item. After the frequent itemsets are identified, they are sorted by their support values in descending order.

**Output Explanation**:  
The output shows the frequent itemsets along with their support values. For example:



Product 24852 appears in 37.29% of the transactions, while product 13176 appears in 31.31%. These frequent itemsets will form the basis for association rule generation.

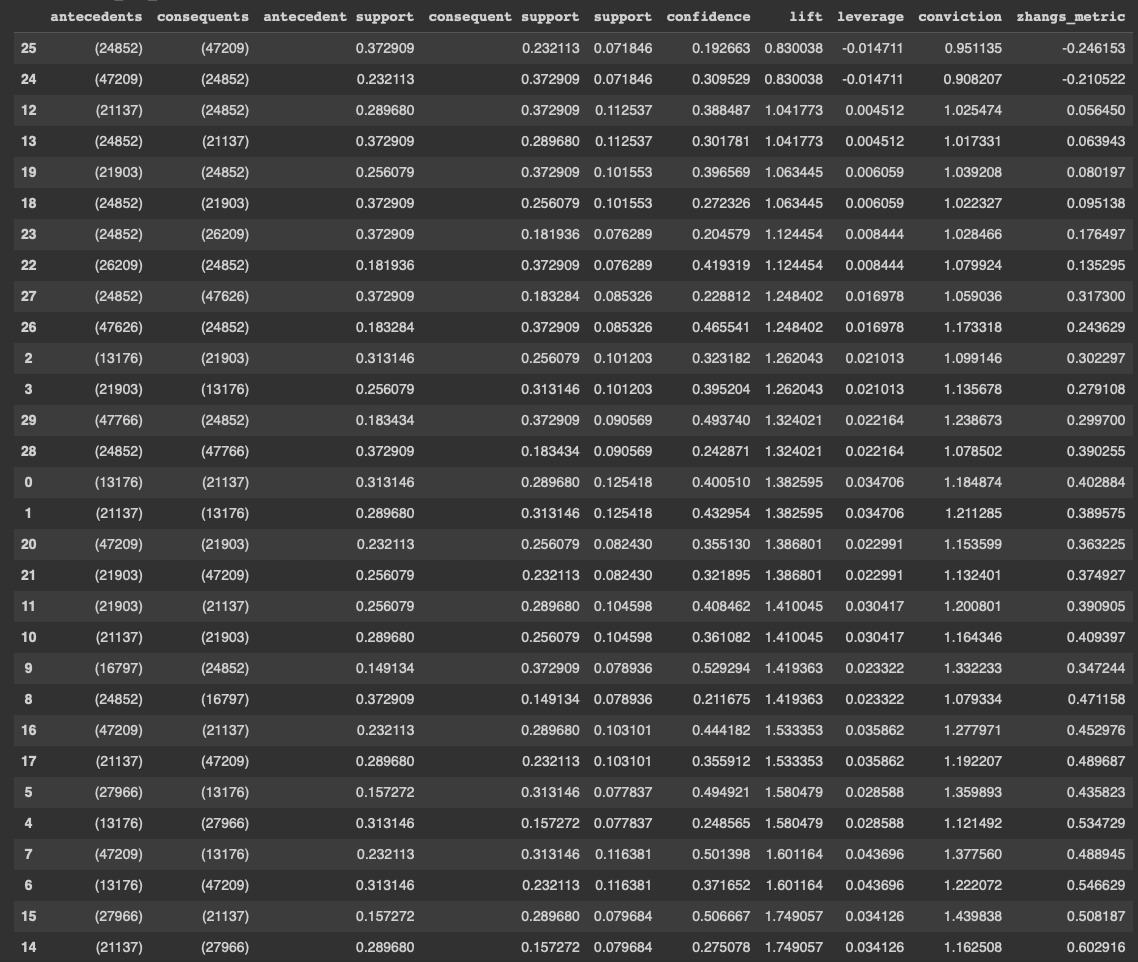
**Kernel 23**

rules = association\_rules(frequent\_itemsets, metric="support", min\_threshold=0.02)

rules.sort\_values(by="lift")

**Code Explanation**:  
The association\_rules function is used to generate association rules from the frequent itemsets. These rules help in identifying product pairs that are often bought together. The rules are sorted by the lift metric, which measures the strength of the association between two items.

**Output Explanation**:  
The output displays the association rules, including metrics like antecedents, consequents, support, confidence, and lift. For example:



This rule indicates that people who buy product 24852 (e.g., bananas) are also likely to buy product 47209 (e.g., avocados), with a confidence of 19.26% and a lift of 0.83 (where a lift > 1 indicates a strong association).

**Kernel 24**

random\_product = rules.sample(1, random\_state=100)["antecedents"].explode().iloc[0]

random\_product

**Code Explanation**:  
This line randomly selects a product from the set of antecedents (products that, if bought, are likely to result in certain consequents being purchased). The sample() function selects a random rule, and explode() unpacks the list of antecedents.

**Output Explanation**:

16797

The output shows a randomly selected product ID (16797). This product will be used as a reference for generating recommendations in the next steps.

**Kernel 25**

lime = 26209

banana = 24852

chocolatesandwichbiscuit = 1

**Code Explanation**:  
This assigns product IDs to specific items:

* lime: Product ID 26209
* banana: Product ID 24852
* chocolatesandwichbiscuit: Product ID 1

These specific product IDs will be used for testing the recommendation function.

**Kernel 26**

def arl\_recommender(rules\_df, id, rec=1):

sorted\_rules = rules\_df.sort\_values("lift", ascending=False)

recommendation\_list = []

for i, k in enumerate(sorted\_rules["antecedents"]):

for j in list(k):

if j == id :

for k in list(sorted\_rules.iloc[i]["consequents"]):

if k not in recommendation\_list:

recommendation\_list.append(k)

return recommendation\_list[0:rec]

**Code Explanation**:  
This function (arl\_recommender) recommends products based on the association rules (rules\_df). It works by:

* Sorting the rules by lift to prioritize stronger associations.
* Finding the products (consequents) associated with the input product ID (id).
* Returning the top rec recommendations based on the sorted rules.

**Kernel 27**

arl\_recommender(rules, 21137, 5)

**Code Explanation**:  
This calls the arl\_recommender function to generate recommendations for product ID 21137 (which could be, for example, organic strawberries) and requests the top 5 recommendations.

**Output Explanation**:

[27966, 47209, 21903, 13176, 24852]

The function recommends the following products based on association rules:

* 27966: Organic raspberries
* 47209: Organic Hass avocado
* 21903: Organic baby spinach
* 13176: Bag of organic bananas
* 24852: Bananas

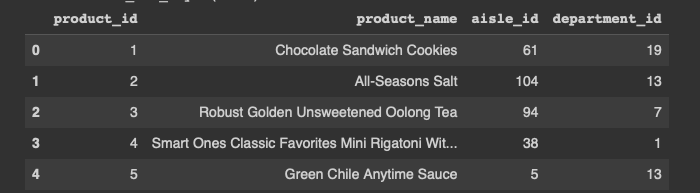
**Kernel 28**

products = pd.read\_csv("/content/products.csv")

products.head()

**Code Explanation**:  
This loads the products.csv file into a DataFrame called products, which contains product metadata such as product\_id, product\_name, aisle\_id, and department\_id. The head() function displays the first few rows of the DataFrame.

**Output Explanation**:  
The output shows the first five rows of the products DataFrame:



Each row represents a unique product with its name, aisle, and department information.

**Kernel 29**

def names\_of\_products(rules\_df, bought, recommend=5):

rec = arl\_recommender(rules\_df, bought, recommend)

name\_of\_rec = {}

bought\_name = products[products["product\_id"]==bought]["product\_name"].iloc[0]

for i in rec:

name\_of\_rec[i] = products[products["product\_id"]==i]["product\_name"].iloc[0]

recommend\_df = pd.DataFrame(name\_of\_rec.items(), columns=["product\_id", "product\_name"])

print(f"Bought: {bought\_name}\n")

return recommend\_df

**Code Explanation**:  
This function (names\_of\_products) returns the names of products recommended for a given input product. It:

* Uses the arl\_recommender function to generate product recommendations.
* Looks up the names of the recommended products and formats them in a DataFrame.
* Prints the name of the bought product and the corresponding recommendations.

**Kernel 30**

names\_of\_products(rules, 21137)

**Code Explanation**:  
This calls the names\_of\_products function to get product recommendations based on product ID 21137 (e.g., organic strawberries).

**Output Explanation**:

Bought: Organic Strawberries

product\_id product\_name

27966 Organic Raspberries

47209 Organic Hass Avocado

21903 Organic Baby Spinach

13176 Bag of Organic Bananas

24852 Banana

The function returns the names of the recommended products, showing that users who bought "Organic Strawberries" may also like "Organic Raspberries," "Organic Hass Avocado," and others.

**Kernel 31**

import joblib

joblib.dump(names\_of\_products, 'product\_recommendation\_model.joblib')

model = joblib.load('product\_recommendation\_model.joblib')

**Code Explanation**:  
This code uses joblib to save the names\_of\_products function as a file (product\_recommendation\_model.joblib). This allows the model to be reused in the future without retraining. The second line loads the saved model for use in a production environment.