**Bruno Garofalo - Midterm**

**Apriori and Brute force algorithm**

*Github repository link:* [*https://github.com/BrunoGarofalo/DataMining-Apriori*](https://github.com/BrunoGarofalo/DataMining-Apriori)

**Content:**

1. Model structure
2. Model outputs
3. Results
4. Summary
5. **Model structure:**
   1. Packages
   2. Item lists
   3. Model functions
   4. Model execution
6. **Packages: Python packages required to run the model**

#standard Python packages

import pandas as pd

import numpy as np

import random

#import package to perform combination calculations

from itertools import combinations

from itertools import permutations

#Apriori packages

from mlxtend.preprocessing import TransactionEncoder

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

#package to silence warning messages

import warnings

# Suppress all warnings

warnings.filterwarnings("ignore")

#rime package to measure model computation time

import time

#package to analyze the file location and check if the transaction datasets already exist in the local drive

import os

#package to save and read transaction datasets on local drive

import pickle

**Item lists: lists of 20 best-selling items from 5 different stores: Amazon, Target, Dick’s, ShopRite, Costco**

#Item lists

shoprite\_top\_selling = ["Milk","Eggs","Bread","Rice","Pasta","Chicken","Beef","Oranges","Spinach","Cereal","Yogurt","Cheese","Butter","Cooking oil","Beans","Canned tomatoes",

    "Peanuts","Jam","Frozen vegetables","Pizza","Ice cream","Chips","Coffee","Tea","Sugar","Salt","Pepper","Dish soap","Toilet paper","Shampoo"]

amazon\_top\_selling = ["Smartphone","Laptop","Headphones","Books","Video games","Fitness tracker","Coffee maker","Bluetooth speaker","Portable charger","Smartwatch",

    "Kitchen knife set","Electric toothbrush","Yoga mat","Air fryer","Wireless earbuds","Instant pot","External hard drive","Resistance bands","Digital camera",

    "Tablet","LED TV","Home security camera","Water bottle","Electric kettle","Printer","Air purifier","Smart thermostat","Dumbbells","Robot vacuum","Car phone mount"]

costco\_top\_selling\_items = ["Toilet paper","Rotisserie chicken","Kirkland Signature batteries","Laundry detergent","Paper towels","Water bottles","Fresh produce",

    "Snack foods","Kitchen appliances","Wine","Frozen foods","Office supplies","Clothing","Electronics","Outdoor furniture","Mattresses","Jewelry","Home appliances",

    "Tools","Books","Pet supplies","Tires","Baby products","Furniture","Health and beauty products","Cookware","Cleaning supplies","Gardening supplies","School supplies",

    "Home decor"]

dicks\_top\_selling\_items = ["Athletic shoes","Sports apparel","Fitness equipment","Outdoor gear","Hiking boots","Running shoes","Exercise clothing","Basketball",

    "Football","Baseball equipment","Golf clubs","Tennis racquets","Camping gear","Fishing equipment","Bicycles","Swimming gear","Yoga equipment","Soccer equipment",

    "Hunting gear","Skiing equipment","Snowboarding gear","Gym bags","Water bottles","Fitness trackers","Gym accessories","Gymnastics equipment","Skateboarding gear",

    "Inline skates","Scooters","Surfing gear"]

target\_top\_selling\_items = ["Household essentials","Groceries","Electronics","Clothing","Home decor","Furniture","Toys","Baby products","Beauty products",

    "Healthcare products","Pet supplies","Outdoor furniture","Kitchen appliances","Bedding","Bath towels","School supplies","Office supplies","Craft supplies",

    "Books","Games","Cookware","Cleaning supplies","Storage solutions","Party supplies","Gardening supplies","Sports equipment","Fitness equipment","Bikes",

    "Skateboards","Scooters"]

1. **Model functions**

***Data generation:***

* **Create\_db():** this function generates 20 random transactions for each store and saves them on the user local drive as a pickle file. Prior to creating a new pickle file, the function verifies that the transaction datasets are not already present in the user folder. In that case the new transaction datasets generation is not executed. This is done to prevent that the datasets change every time that the model is executed

"""-----------------------DATA GENERATION FUNCTIONS-------------------------------"""

#function to generate random datasets with 20 transactions each

def create\_db(max\_items, n\_transactions):

#Check if the datasets have already been created, if not move on otherwise stop the function

#I do not want to keep creating datasets every time that I relaunch the model

if os.path.exists('amazon.pkl') and os.path.exists('costco.pkl') and os.path.exists('dicks.pkl') \

and os.path.exists('target.pkl') and os.path.exists('shoprite.pkl'):

print("Pickle files already exist. Skipping new transactions dataset creation.")

return

#Lists of transactions

amazon = []

target = []

dicks = []

costco = []

shoprite = []

#Store and item lists

store\_list = [["Amazon", amazon\_top\_selling], ["Costco",costco\_top\_selling\_items ], ["Dicks", dicks\_top\_selling\_items], ["Target", target\_top\_selling\_items],

["ShopRite", shoprite\_top\_selling]]

#Generate transactions for each store

for store, items in store\_list:

for \_ in range(n\_transactions):

transaction\_items = random.sample(items, min(np.random.randint(1, max\_items), len(items)))

if store == "Amazon":

amazon.append(transaction\_items)

elif store == "Costco":

costco.append(transaction\_items)

elif store == "Dicks":

dicks.append(transaction\_items)

elif store == "Target":

target.append(transaction\_items)

elif store == "ShopRite":

shoprite.append(transaction\_items)

# Save datasets to pickle files on local drive

with open('amazon.pkl', 'wb') as f:

pickle.dump(amazon, f)

with open('costco.pkl', 'wb') as f:

pickle.dump(costco, f)

with open('dicks.pkl', 'wb') as f:

pickle.dump(dicks, f)

with open('target.pkl', 'wb') as f:

pickle.dump(target, f)

with open('shoprite.pkl', 'wb') as f:

pickle.dump(shoprite, f)

* **get\_pickle():** This function retrieves the transaction dataset pickle files from the user folder

#function to read datasets from pickle files on local drive

def get\_pickle():

with open('amazon.pkl', 'rb') as f:

amazon = pickle.load(f)

with open('costco.pkl', 'rb') as f:

costco = pickle.load(f)

with open('dicks.pkl', 'rb') as f:

dicks = pickle.load(f)

with open('target.pkl', 'rb') as f:

target = pickle.load(f)

with open('shoprite.pkl', 'rb') as f:

shoprite = pickle.load(f)

return amazon, costco, dicks, target, shoprite

***Apriori algorithm:***

* **apriori\_function():** executes the Apriori algorithm for the selected store transaction dataset. The algorithm uses the minimum threshold to find the frequent itemsets and the minimum confidence to find the association rules

"""---------------------------------APRIORI ALGO--------------------------------------------"""

#Function for Apriori algorithm

def apriori\_function(dataset, min\_support, min\_confidence):

# Transform the dataset into a binary format suitable for Apriori

te = TransactionEncoder()

te\_ary = te.fit(dataset).transform(dataset)

df = pd.DataFrame(te\_ary, columns=te.columns\_)

# Apply Apriori algorithm to find frequent itemsets

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

if len(frequent\_itemsets) == 0:

print("No frequent itemsets found")

return

else:

# Generate association rules

rules = association\_rules(frequent\_itemsets, metric="confidence", min\_threshold=min\_confidence)

frequent\_itemsets\_count = len(frequent\_itemsets)

rules\_count = len(rules)

# Display results

print("Frequent Itemsets:")

display(frequent\_itemsets)

print("\nAssociation Rules:")

display(rules)

return rules\_count, frequent\_itemsets\_count

***Brute force algorithm:***

* **get\_store():** this function prompts the user to select one store by entering an integer between 1 and 5, or all stores by entering 6. Any other entry such as decimals or string won’t be accepted

"""-----------------------------------------------------BRUTE FORCE FUNCTIONS----------------------------------------------------------"""

# Define a function to take in the store input from the user

def get\_store():

while True:

store\_name = input("Enter a store number: 1. Amazon, 2. Costco, 3. Target, 4. Dick's, 5. ShopRite, 6. All: ")

try:

store\_num = int(store\_name)

if 1 <= store\_num <= 6:

if store\_num == 1:

return "amazon"

elif store\_num == 2:

return "costco"

elif store\_num == 3:

return "target"

elif store\_num == 4:

return "dicks"

elif store\_num == 5:

return "shoprite"

elif store\_num == 6:

return "all"

else:

print("Invalid store number. Please enter a number between 1 and 6.")

except ValueError:

print("Invalid input. Please enter a valid integer number.")

* **get\_min\_support\_confidence():** this function prompts the user to enter a decimal value for the minimum support and minimum confidence. The acceptable range is greater than zero or less than or equal to one. Any other numeric value or string won’t be accepted

#function to get the minimum support and minimum confidence values from the suer

def get\_min\_support\_confidence():

while True:

min\_support = input("Enter a decimal minimum support value greater than 0 and less or equal than 1: ")

min\_confidence = input("Enter a decimal confidence support value greater than 0 and less or equal than 1: ")

try:

min\_support = float(min\_support)

min\_confidence = float(min\_confidence)

if 0 < min\_support <= 1 and 0 < min\_confidence <= 1:

return min\_support, min\_confidence

else:

print("Invalid values. Please enter decimal numbers between 0 and 1.")

except ValueError:

print("Invalid input. Please enter a valid decimal number.")

* **itemset\_gen():** this function uses the itemset size n and the store top selling list passed to the combinations function to generate all possible itemsets of size n from the store top selling list. The function returns the n itemset

#generate itemset of size n for selected store

def itemset\_gen(n, store\_top\_selling):

itemset = list(combinations(store\_top\_selling, n))

return itemset

* **support\_calc():** this function is used to find the new frequent itemsets and add them to the support\_val\_dict that stores all of the frequent itemset values. For each item in the itemset, the algorithm counts the number of times the itemset appears in the store transactions and then calculates the support by dividing the count over the total number of transactions. If the support value is greater or equal than the minimum support, then the frequent itemset is added to the dictionary and the variable new\_frequent\_itemset is updated to True.

#Support calculation for an itemset of any size

def support\_calc(itemset, store\_transactions, min\_support):

support\_val\_dict = {}

new\_frequent\_itemset = False

#Calculate the number of times the itemset appears in the transactions dataset

for item in itemset:

final\_support = 0

support\_val = 0

for transaction in store\_transactions:

if all(word in transaction for word in item):

support\_val += 1 #increment itemset count by one if the itemset is in the transaction

#calculate the support value for the itemset

final\_support = support\_val/len(store\_transactions)

#if the support value is greater or equal than the minimum support, join the items and add the new

#frequent itemset to the support val dictionary

if final\_support >= min\_support:

key\_ = ",".join(item)

support\_val\_dict[key\_] = final\_support

#if a new frequent itemset was added, change new\_frequent\_itemset to True

new\_frequent\_itemset = True

return support\_val\_dict, new\_frequent\_itemset

* **permutation\_support():** this function calculates the support for a frequent itemset permutation and saves it to the permutation\_support\_dict

#function to get the support for the frequent item permutations

#for every frequent itemset, find all possible permutations

def permutation\_support(itemset, frequent\_itemset):

permutation\_support\_dict = {}

itemset\_split = itemset.split(",")

itemset\_len = len(itemset\_split)

itemset\_permutations = list(permutations(itemset\_split, itemset\_len))

support\_val = frequent\_itemset[itemset]

for item in itemset\_permutations:

item = ",".join(item)

permutation\_support\_dict[item] = support\_val

return permutation\_support\_dict

* **perm\_support\_to\_frequent\_itemset():** calculates the support for all frequent itemset permutations

#frequent\_itemset\_w\_permutations: get the support value for all frequent itemset permutations

def perm\_support\_to\_freq\_itemset\_dict(frequent\_itemset):

temp\_dict = {}

for item in frequent\_itemset:

item\_split = item.split(",")

if len(item\_split)>=2:

temp\_dict.update(permutation\_support(item, frequent\_itemset))

return temp\_dict

* **association \_rule():** for each frequent itemset and all of their possible permutations, this function splits the itemset and then for each left and right side finds the support value, then calculates the confidence value. If the confidence value is >= than the minimum confidence value, then the permutation is saved to the association\_ruler\_dict with the key formatted as left-side -> right-side. This dictionary is then converted to a Pandas dataframe, the columns of the df are renamed. The left and right sides are saved to their respective columns in the dataframe. Finally, all duplicate subsets in the left and right columns of the df are dropped to ensure that the association rules are not double-counted

#function to extract the association rules

def association\_rules\_funct(frequent\_itemset\_w\_permut\_dict, min\_confidence):

#initialize dictionary that will store the association rules

association\_rules\_dict = {}

for item in frequent\_itemset\_w\_permut\_dict.keys():

#split the itemset into individual words

item\_split = item.split(",")

#get the length and the individual items in each itemset

item\_len = len(item\_split)

#for each itemset calculate the association rules

if item\_len >=2:

for i in range(1, item\_len):

left\_side = ",".join(item\_split[0:i])

#this will format the association rule so that it will look like A-> B

key\_ = left\_side+" -> "+ ",".join(item\_split[i:item\_len])

left\_side\_support = frequent\_itemset\_w\_permut\_dict[left\_side]

full\_support = frequent\_itemset\_w\_permut\_dict[item]

confidence = full\_support/left\_side\_support

#if the itemset confidence is greater than the minimum confidence, then add itemset to association rule dictionary

if confidence >= min\_confidence:

association\_rules\_dict[key\_] = [round(full\_support, 3), round(confidence,3)]

#Convert the association rule dictionary to a dataframe so that results can be displayed in a clean format

results\_df = pd.DataFrame(association\_rules\_dict).T

results\_df.reset\_index(inplace=True)

results\_df = results\_df.rename(columns={0: "Support", 1: "Confidence", "index": "Association rule"})

left\_list = []

right\_list = []

#Add the left and right sides of the association rules to the left and right list

for i in results\_df["Association rule"].to\_list():

left\_, right\_ = i.split("->")

left\_list.append(left\_)

right\_list.append(right\_)

#add the left and right lists to the results dataframe

results\_df["left"] = left\_list

results\_df["right"] = right\_list

# Apply the sorting function to each element of the DataFrame's 'left' and 'right' columns

results\_df[['left', 'right']] = results\_df[['left', 'right']].applymap(sort\_row)

# Convert the sorted lists to tuples to make them hashable

results\_df['left'] = results\_df['left'].apply(tuple)

results\_df['right'] = results\_df['right'].apply(tuple)

# Drop duplicate association rules

results\_df.drop\_duplicates(subset=['left', 'right'], inplace=True)

#Reset the dataframe index

results\_df.reset\_index(drop=True, inplace=True)

results\_df.drop(columns=["left", "right"], inplace=True)

return results\_df

**Part D: master function for the model execution**

The last part of the model uses all functions presented above to output the results of the Apriori and Brute Force sequentially

The first block of the code creates the dataset and/or retrieves the data from the local drive.

def model\_run():

#initialize lists where result counts and computation times will be stored

apriori\_counts = []

brute\_counts = []

computation\_time = []

"""============================DATA CREATION"""

#create 5 databases with 20 transactions each; new data won't be generated if there exist pickle files in the local drive

create\_db(max\_items = 16, n\_transactions=20)

#retrieve transaction data from pickle files

amazon, costco, dicks, target, shoprite = get\_pickle()

#Store and item dictionary of lists

store\_dictionary = {"amazon": [amazon\_top\_selling, amazon], "costco": [costco\_top\_selling\_items, costco], "dicks": [dicks\_top\_selling\_items, dicks], "target": [target\_top\_selling\_items, target],"shoprite": [shoprite\_top\_selling, shoprite]}

The second block of the code, gets the user inputs (store, minimum support, minimum confidence) and create the corresponding variables:

* **Store\_selection:** the list of selected stores, either all or a single store

"""============================INPUT RETRIEVAL FROM USER"""

#get the store name from the user

store\_input = get\_store()

print(f"Your selected store is {store\_input}")

#get the minimum support and confidence value from the user

min\_support, min\_confidence = get\_min\_support\_confidence()

print(f"Your selected minimum support is {min\_support}")

print(f"Your selected minimum confidence is {min\_confidence}")

store\_selection = []

if store\_input=="all":

store\_selection = ['amazon', 'target', 'costco', 'dicks', 'shoprite']

else:

store\_selection = [store\_input]

The next block of code is a for loop that cycles through all stores in the selected\_store list and assigns the store transaction data to the store\_transactions variable. If only one store was selected, then the model will return only the results for that store.

for store\_name in store\_selection:

#Get the top selling list for the selected store

store\_top\_selling = store\_dictionary[store\_name][0]

#Get the transaction list for the selected store

store\_transactions = store\_dictionary[store\_name][1]

**Apriori algorithm:** at this point the Apriori function is executed and the results are displayed as a dataframe. The time variables allow to calculate the execution time for this block of code

print(f"STORE = {store\_name}=============================APRIORI ALGORITHM EXECUTION============================")

# Start measuring time

start\_time = time.time()

apriori\_rules\_count, apriori\_frequent\_itemsets\_count = apriori\_function(store\_transactions, min\_support, min\_confidence)

#Save Apriori counts for frequent itemsets and association rules

apriori\_counts.append([apriori\_rules\_count, apriori\_frequent\_itemsets\_count])

# Stop measuring time

end\_time = time.time()

# Calculate and display the elapsed time

elapsed\_time\_apriori = round(end\_time - start\_time,3)

print(f"Code computation time of Apriori algorithm: {elapsed\_time\_apriori} seconds")

**Brute force algorithm:** in the next block the Brute force code is executed. Inside a while loop, the itemset of size k is extracted from the store\_top\_selling item list, then for each itemset the support is calculated. If the itemset is frequent, then it is added to the dictionary with the frequent itemsets. If new frequent itemsets were found, then the value of k is increased by 1 and the code loops over, otherwise the while loop will break. The frequent itemsets are then displayed as a dataframe.

print(f"\n\n\nSTORE = {store\_name}=============================BRUTE FORCE ALGORITHM EXECUTION=========================")

# Start measuring time

start\_time = time.time()

# frequent\_itemset = {}

itemset = []

k = 1 #itemset size

final\_frequent\_set = {}

#This while loops find the new itemsets of size k and calculates if there are new frequent itemsets

while True:

print(f"Computing {k} frequent itemsets")

#generate all itemsets of size k

itemset = itemset\_gen(k, store\_top\_selling)

# itemset.extend(itemset\_list)

#calculate the support for all n-item-sets

#dictionary that will contain the support values for the itemsets

frequent\_set, new\_frequent\_event = support\_calc(itemset, store\_transactions, min\_support)

# new\_freq\_itemset = freq\_item(support\_val\_dict, min\_support)

#dictionary that contains the frequent itemsets

#If there are new itemsets, add new frequent itemsets to dictionary

#and increase the itemset size k

if new\_frequent\_event == True:

final\_frequent\_set.update(frequent\_set)

k +=1

else:

break

frequent\_itemset\_df = pd.DataFrame.from\_dict(final\_frequent\_set, orient='index', columns=['Support'])

frequent\_itemset\_df.reset\_index(inplace=True)

frequent\_itemset\_df.rename(columns={"index": "Itemset"}, inplace=True)

display(frequent\_itemset\_df)

The last block of code calculates the support for all frequent itemsets permutations and saves the results into the frequent\_itemset\_w\_permut\_dict; then this dictionary is passed to the association\_rules\_funct() so that all association rules for all permutations can be calculated. Duplicates will be dropped within this function. The association\_rules\_funct() outputs the results\_df which contains the association rules in a dataframe format. Finally, the computation time of the Brute force code is displayed

#initialize a new dictionary that will contain the frequent itemsets and their support values,

#as well as the itemset permutations and their support values

frequent\_itemset\_w\_permut\_dict = {}

frequent\_itemset\_w\_permut\_dict.update(final\_frequent\_set)

frequent\_itemset\_w\_permut\_dict.update(perm\_support\_to\_freq\_itemset\_dict(final\_frequent\_set))

print("calculating associations rules")

#get the association rules

results\_df = association\_rules\_funct(frequent\_itemset\_w\_permut\_dict, min\_confidence)

if results\_df.empty:

print(f"the {store\_name} dataset contains no association rules")

else:

print(f"the {store\_name} dataset contains {len(results\_df)} association rules")

display(results\_df)

# Stop measuring time

end\_time = time.time()

#Save Brute force counts for frequent itemsets and association rules

brute\_rules\_count = results\_df.shape[0]

brute\_frequent\_itemsets\_count = frequent\_itemset\_df.shape[0]

brute\_counts.append([brute\_rules\_count, brute\_frequent\_itemsets\_count])

# Calculate and display the elapsed time

elapsed\_time\_brute = round(end\_time - start\_time,3)

#save computation times to list

computation\_time.append([elapsed\_time\_apriori, elapsed\_time\_brute])

print(f"Code computation time of Brute Force algorithm: {elapsed\_time\_brute} seconds")

return brute\_counts, apriori\_counts, store\_selection, computation\_time

**d) Model execution:**

In the last block of code the entire model is executed in a sequential manner.The model\_run() master function will prompt the user to enter the following inputs:

* **Dataset** number, from 1 to 6 where 6 means all datasets
* **Minimum support**
* **Minimum confidence**

Then the entire code will be executed and the function will return:

* **Brute\_counts:** count of association rules and frequent itemsets found by the Brute Force for each dataset analyzed
* **Apriori\_counts**: count of association rules and frequent itemsets found by the Apriori for each dataset analyzed
* **Store\_selection**: the list of stores selected by the user
* **Computation\_time**: computation times for Apriori and Brute force for each dataset analyzed

The counts and elapsed time are then saved in a dataframe and new feature, Speed\_comparison, is calculated as: brute elapsed time / apriori elapsed time. This will show how much faster the Apriori is in comparison to the Brute force. At the end the means for the Brute force and Apriori elapsed times are computed and printed

#Function that runs the entire model

brute\_counts, apriori\_counts, store\_selection, computation\_time = model\_run()

#Return the side by side comparison of Apriori and Brute force algorithms

print("\n\n=============================RESULTS COMPARISON=============================")

results\_summary\_df = pd.DataFrame()

results\_summary\_df['Store name'] = store\_selection

results\_summary\_df['Apriori time (sec)'] = [x[0] for x in computation\_time]

results\_summary\_df['Brute force time (sec)'] = [x[1] for x in computation\_time]

results\_summary\_df['Speed comparison'] = (results\_summary\_df['Brute force time (sec)']/results\_summary\_df['Apriori time (sec)'])

results\_summary\_df['Apriori association rules'] = [x[0] for x in apriori\_counts]

results\_summary\_df['Apriori freq itemsets'] = [x[1] for x in apriori\_counts]

results\_summary\_df['Brute force association rules'] = [x[0] for x in brute\_counts]

results\_summary\_df['Brute force freq itemsets'] = [x[1] for x in brute\_counts]

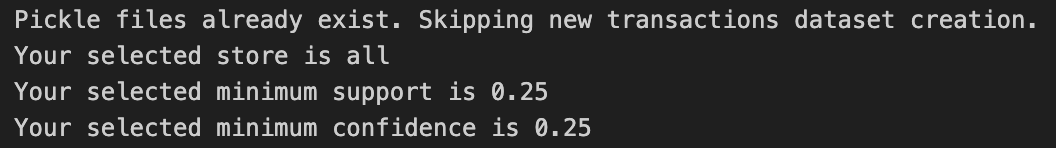
display(results\_summary\_df)

print(f"Apriori mean time = {round(results\_summary\_df['Apriori time (sec)'].mean(),3)} | Brute force mean time = {round(results\_summary\_df['Brute force time (sec)'].mean(),3)} | Speed comparison mean = {round(results\_summary\_df['Speed comparison'].mean(),3)}")

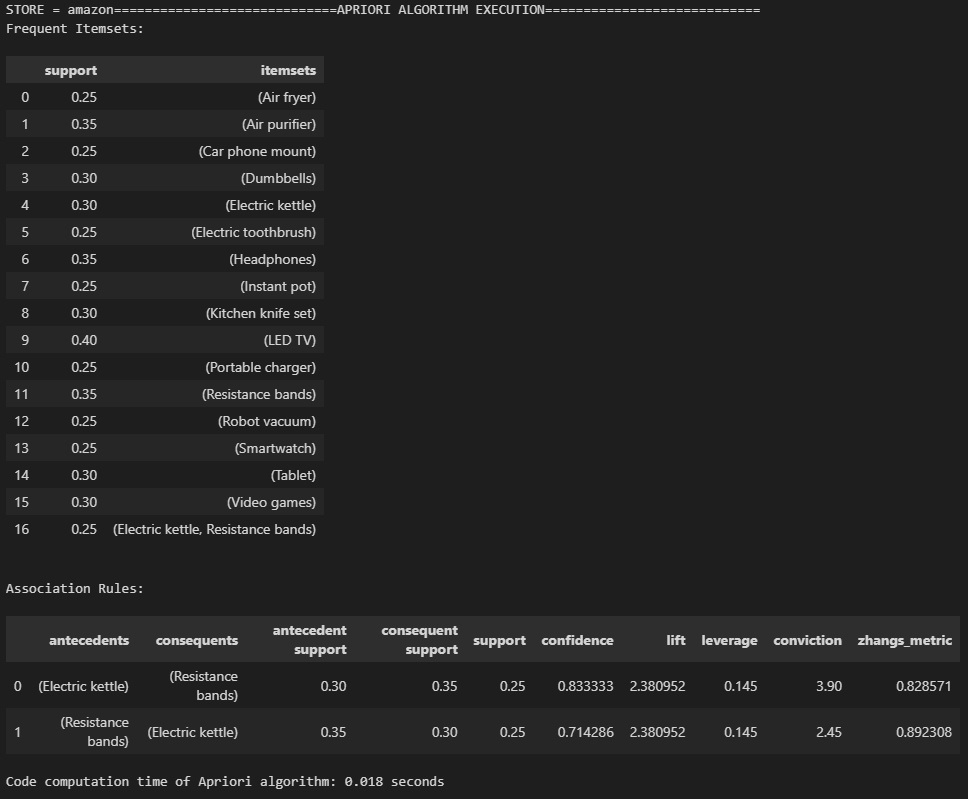
1. **Model Outputs:**

This line will show the store name and the algorithm that computed the results:

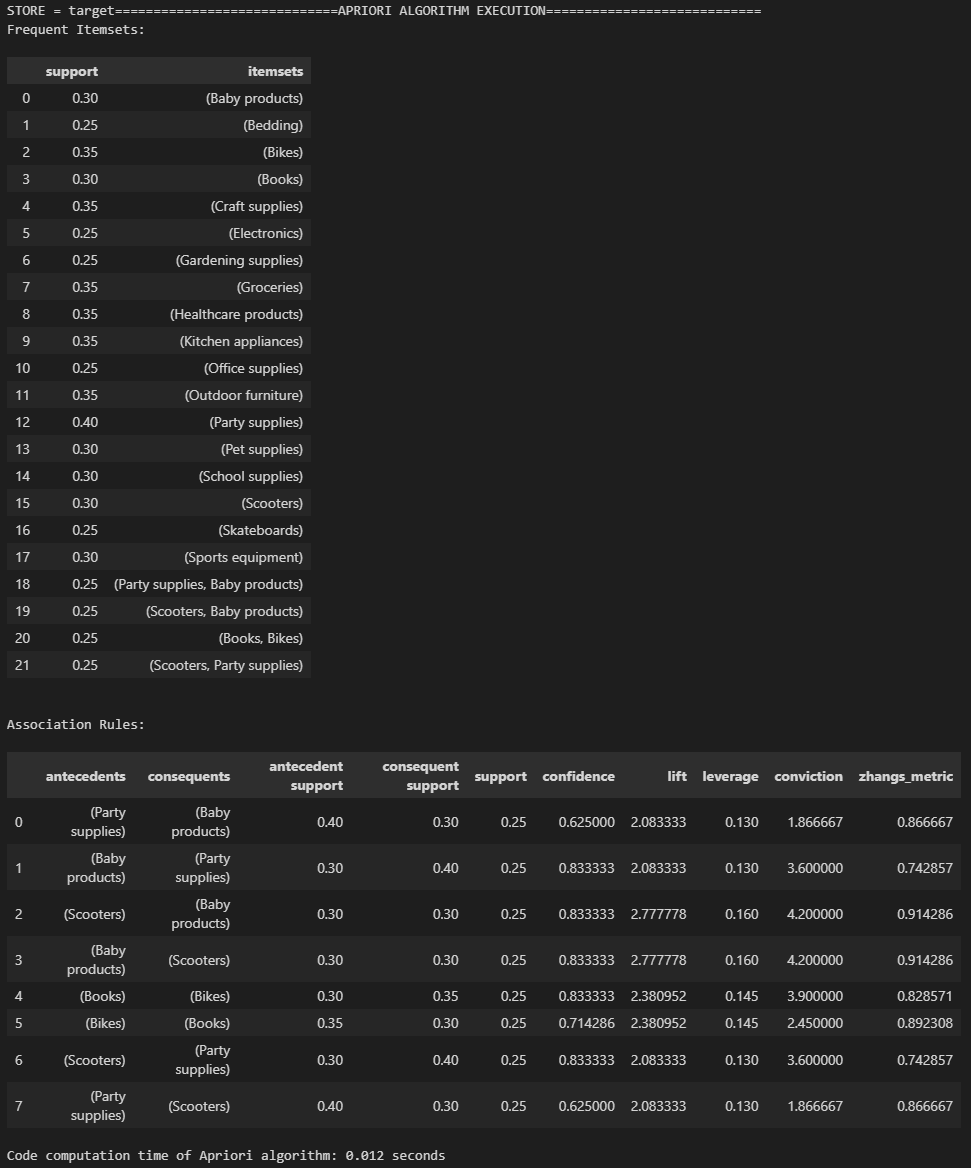
**STORE = amazon===================APRIORI ALGORITHM EXECUTION============================**



**Amazon:**

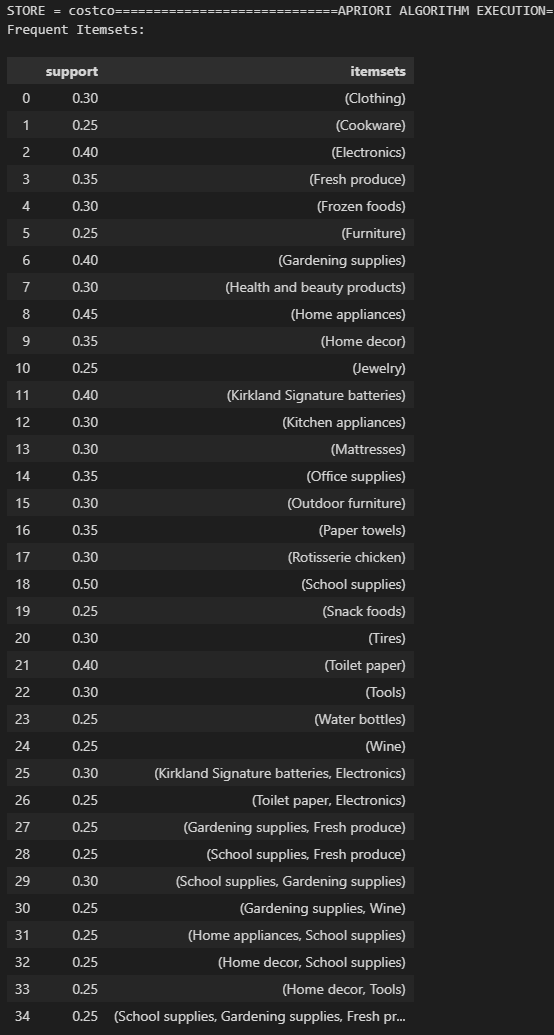


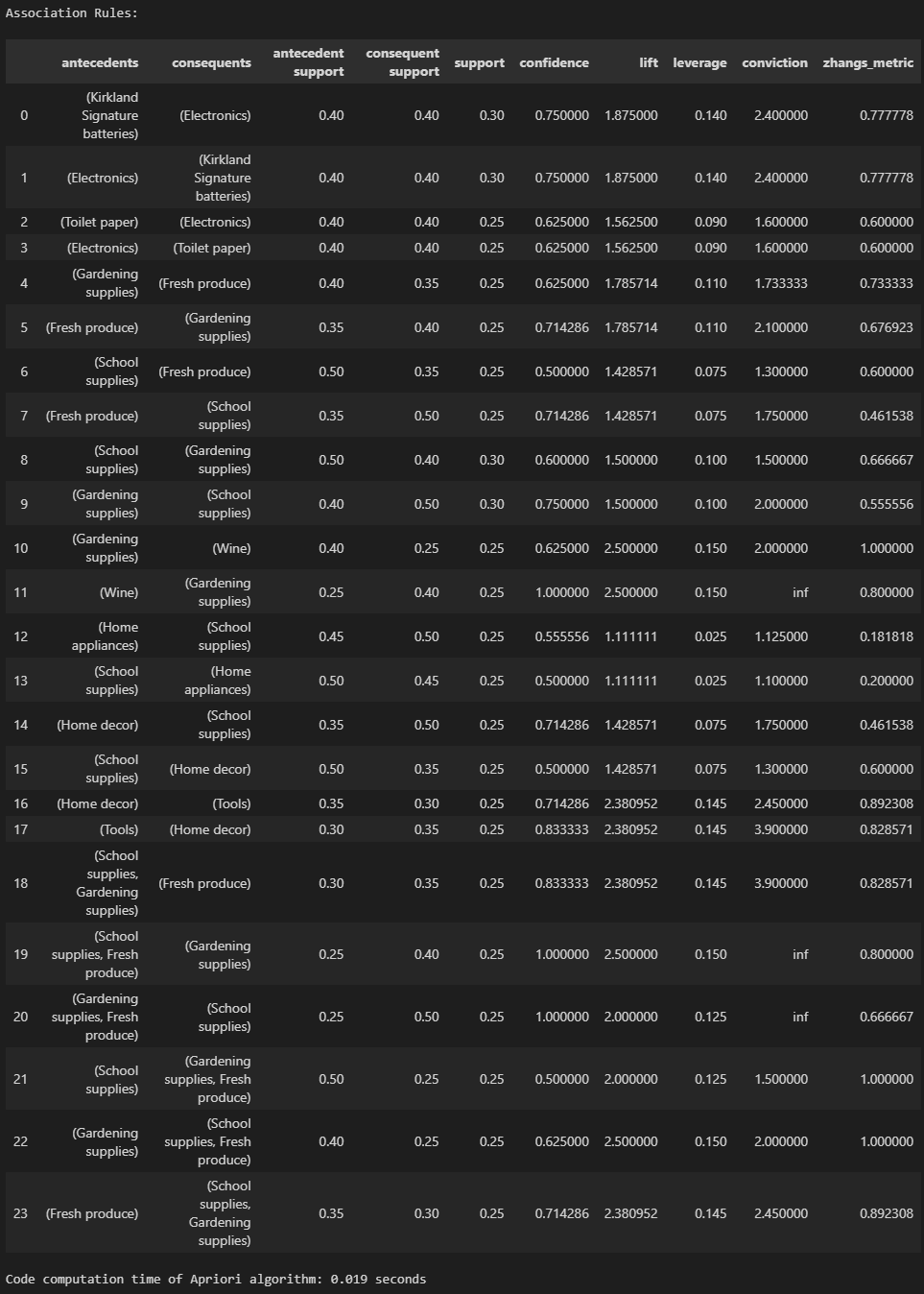


**Target**:

****

**Costco:**

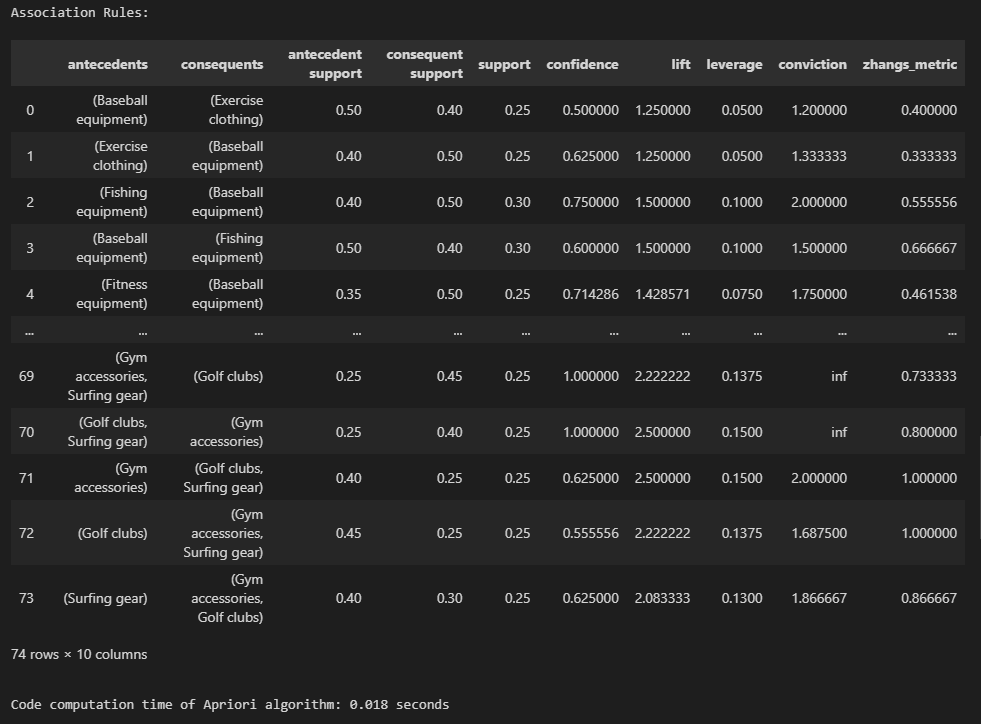






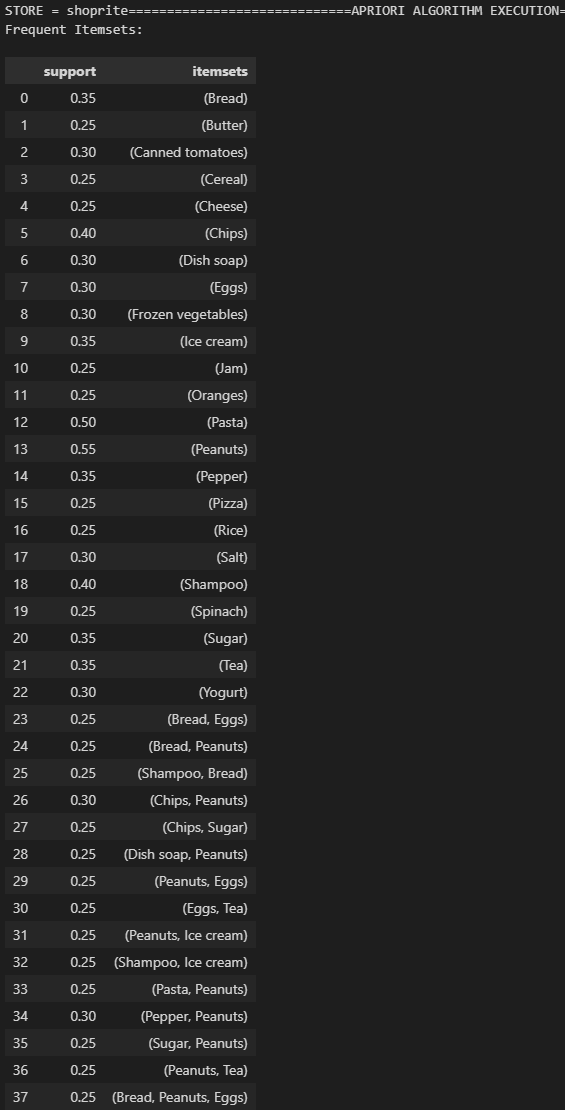
**Dick’s:**

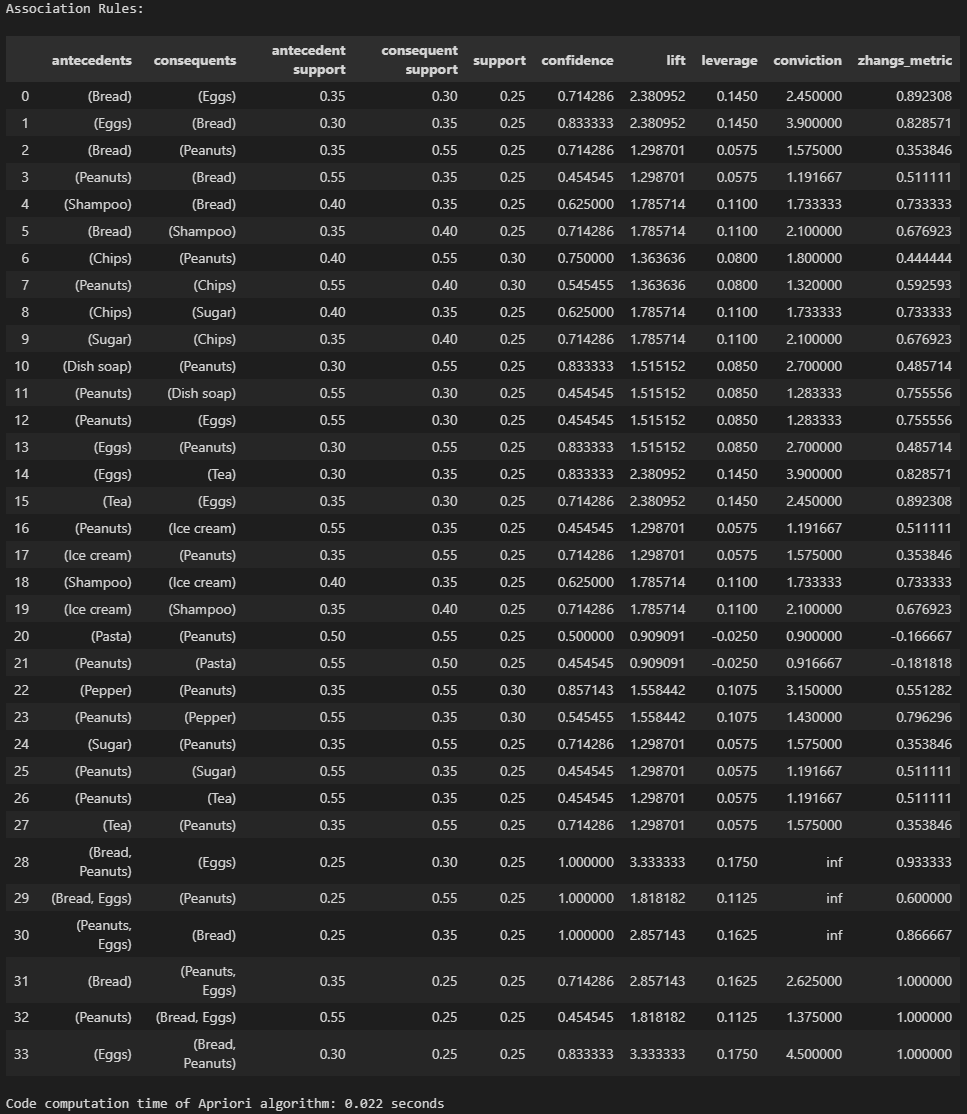


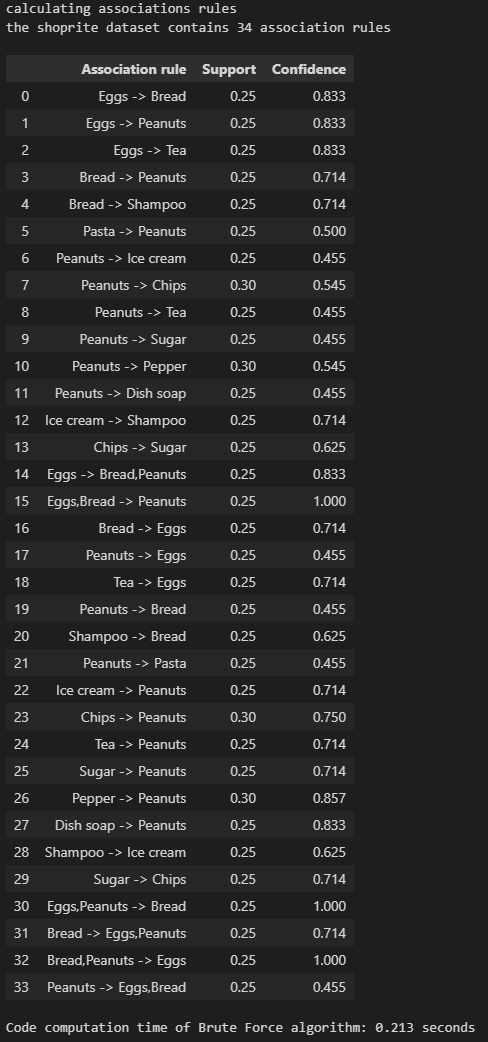




**SHOPRITE**:







1. **Results: 6.All datasets, minimum support = 0.25, minimum confidence = 0.25**



The values in the Speed Comparison column represent the ratio between the Brute Force elapsed time and the Apriori elapsed time. As we can see on average the Apriori algorithm is 7.5 times faster than the algorithm. As expected, when the number of frequent itemsets increases, the computation time increases as well. For example, from the table above, we can see that the number of frequent itemsets in the Dick’s dataset is 58, and the computation time of the Brute force was about 7 times longer than that of the Target dataset that has only 22 frequent itemsets, less than 50% of those of the Dick’s dataset. Despite the computation time difference, both algorithms returned the same identical results. In fact, as shown in the table, I was able to achieve a perfect match for both association rules and frequent itemsets for each database analyzed.

1. **Summary**

In conclusion, I built a model that utilizes the Apriori algorithm and the Brute Force algorithm to calculate the frequent itemsets and association rules for the dataset selected, based on the minimum support value and minimum confidence values entered by the user. Both algorithms returned the same identical results, but in terms of performance the Apriori algorithm is between 2 and 12 times faster than the Brute Force, depending on the transactions complexity and number of frequent itemsets found.