Second Home Assignment

Made by Group 3:

- Diogo Araújo - fc60997 -2H
- João Braz - fc60419 - 2H
- Joel Oliveira - fc59442 - 14H

For the second home home assignment, we completed two objetives regarding itemsets (frequent, closed and maximal) and their associated rules. These will be explained further in their sections.

In [ ]:
```
import pandas as pd
import numpy as np
import pickle
import matplotlib.pyplot as plt

from time import time
from pyfim import pyeclat
from PD_freqitems import freqitemsets
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import (apriori, association_rules, fpgrowth, fpm
```

Load and Analyze Data

In this section, we loaded and analyzed the data obtained for the home assignment.

This data is divided into two separate files, these being the "products.txt" and the "order_products.pickle" files. For the products file, we can find all products that can be ordered as well as what isle they can be found in and what department they belong to (their ids). In the orders file, we can find the orders (transactions) that took place which are the bought products.

In [ ]:
```
#Read product names and IDs
lines=open("products.txt", "rt", encoding="utf8").readlines()

#we subtract 1 because the pids start at 1, the first 0 is never filled
#therefore the product with pid {pid} is at index {pid}-1
products=[0]* (len(lines)-1)
for lin in lines[1:]:
    pid, pname, aid, did=lin.strip().split("\t")
    products[int(pid) - 1]=pname

#read transactions
orders=pickle.load(open("order_products.pickle", "rb"))

#check products on order 6:
print("Products for Sixth Order:")
for prod in orders[6]: print(products[prod-1])
```

```
Products for Sixth Order:
Cleanse
Dryer Sheets Geranium Scent
Clean Day Lavender Scent Room Freshener Spray
```

In [ ]:
```python
#Create functions to support creation of itemsets
def index_to_product(index):
    return products[index]
def sequence_of_index_to_products(sequence):
    return [index_to_product(i) for i in sequence]

#Print out sizes of files
print("Length of Orders, Products, Combination of Files (Respectively): " + str(
```

```
Length of Orders, Products, Combination of Files (Respectively): 3214874 | 49688
| 159740659312
```

Objective 1 - Analyze the itemset/rules generation procedure

For the first objective, we analyzed the candidate methods discussed in class, which are Apriori, FP-Growth, ECLAT and Naive-Bayes, and identified a good approach based on their performance up to a threshold level of support. This also allowed us to define a good support threshold for analysis for that same approach.

In [ ]:
```python
#Create a list of all orders
order_list = orders.values()
order_list = [list(map(lambda x: x-1, order)) for order in order_list]

#Create an encoder in order to make the orders into a sparse matrix
encoder = TransactionEncoder().fit(order_list)
binary_orders = encoder.transform(order_list, sparse=True)
binary_orders = pd.DataFrame.sparse.from_spmatrix(binary_orders, columns=encoder
```

In [ ]:
```python
#Create the columns for our results dataset
results = pd.DataFrame(columns = ["threshold", "n_itemsets", "apriori",
                                  "fp-growth", "eclat", "naive"])

#threshold values to explore
thresholds = [0.03, 0.02, 0.01, 0.009, 0.007, 0.005, 0.003, 0.001,]
#buffer to store time results of each fqi function;
results_list = [[np.nan for _ in range(len(thresholds))] for __ in range(4)]
#max number of iterations for each fqi function
#(crash limits found or values where the methods have beend surpassed)
caps = [3, float('inf'), 7, 6 ]

#Create functions and data which we will use for the performance analysis
func = [apriori, fpgrowth, pyeclat, freqitemsets]
data = [binary_orders, order_list]
```

In [ ]:
```
# performed each algorithm once at a time because of memory limitations ( 16GB )
#index for the data source in the list {data}
data_index = 1
#index for the fqi function in the list {func}
func_index = 3
for i,thresh in enumerate(thresholds):
    if i>caps[func_index]:
        break
    start = time()
    fi = func[func_index](
        data[data_index],
        thresh
    )
    stop = time() - start
    results_list[func_index][i] = stop
    results.loc[i] = {
        "threshold": thresh,
        "n_itemsets": len(fi),
        "apriori": results_list[0][i],
        "fp-growth": results_list[1][i],
        "eclat": results_list[2][i],
        "naive": results_list[3][i]
    }
```

When running the previous code, we could verify that the apriori algorithm consumes a lot more memory than any of the other 3 algorithms. With the other methods, we can set the threshold value to a much lower value past 0.009. When using apriori, any value below that would crash the program because of insufficient memory.

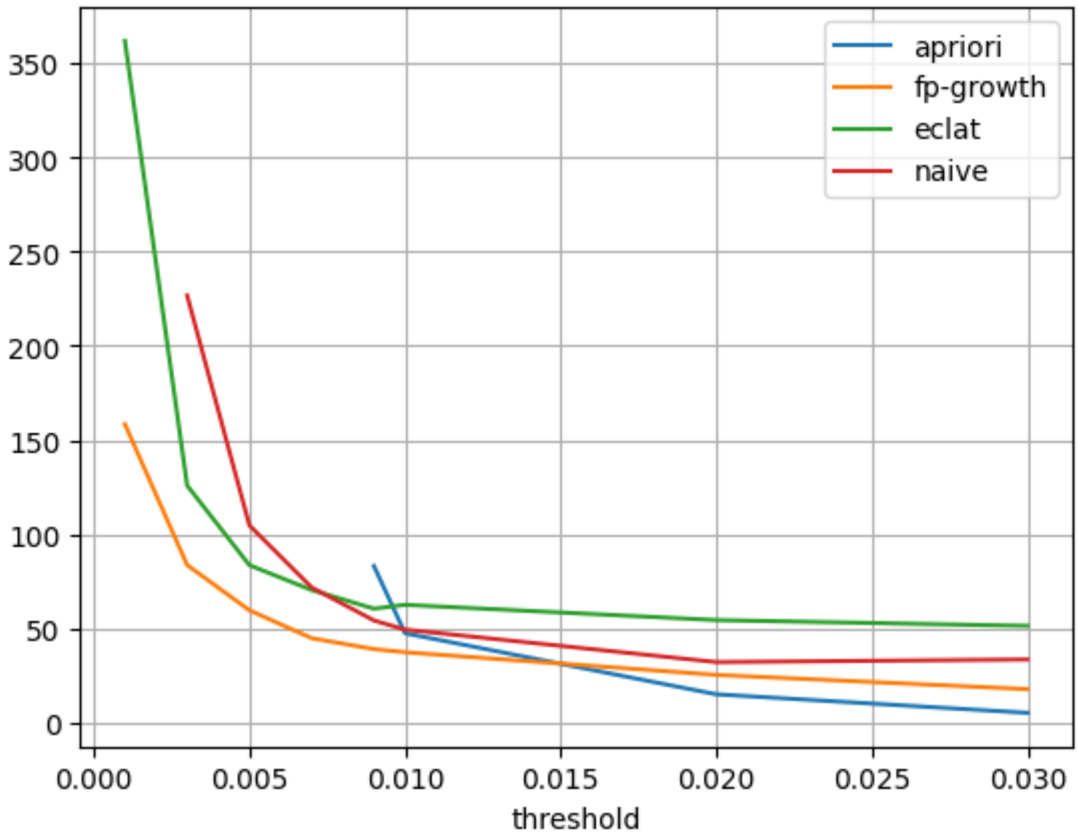
As we can see from the next plot, the *Apriori* approach is the one that has the better results for a larger threshold. However, the time it takes rapidly surpasses all the other algorithms and as we said consumes more memory than the others too. Because of this, it is not a good option and we did not explore it beyond the support value of 0.009 since the memory consumption was too high.

The *Eclat* approach started with a performance worse than the '*Naive*' approach. However, as we lowered the threshold it (*Eclat*) surpassed it's (*Naive*) performance.

The *FPGrowth* approach has a much better trend and is therefore better than *Eclat* and '*Naive*', being only surpassed by apriori when the threshold value is higher. Therefore, we found that *FPGrowth* was the best option. Because of this, we then exploited it's minimum threshold given the limits of our machine.

In [ ]:
```
#Plot the results of the performance analysis
results.plot(x="threshold", y=["apriori", "fp-growth", "eclat", "naive"], grid=T
```

Out[ ]:
```
<Axes: xlabel='threshold'>
```



From the code seen below, we see that with the threshold at 0.01% the memory surpassed 16 GB. Therefore, we will use the threshold values of 0.02%

In [ ]:
```
#Test minimum threshold for FPGrowth
threshold=[0.0005, 0.0002, 0.0002, 0.0001]
for thresh in threshold:
    fi = fpgrowth(binary_orders, thresh)
```

Objective 2 - Identify the most relevant rules

For the second objective, we generated our frequent itemsets, their maximal and closed itemsets and their associated rules.

In [ ]:
```
#Generate all availble itemsets (based on approach and support)
thresh=0.0002
fi = fpgrowth(binary_orders, thresh)
```

In [ ]:
```
#Create function to print out rules
def print_rule(rules, index):
    for item in sequence_of_index_to_products(rules.loc[index, "antecedents"]):
        print(item)
    print("==>")
    for item in sequence_of_index_to_products(rules.loc[index, "consequents"]):
        print(item)

#Create rules based on itemset with a threshold (confidence) value of 60%
rules = association_rules(fi, metric="confidence", min_threshold=0.6)

#Filter the rules once more
promissing_rules = rules[ rules.lift>=rules.lift.quantile(0.75) ]
```

In [ ]:
```
promissing_rules.sort_values(by="lift", ascending=False).iloc[:5]
```

Out[ ]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	li
80	(6505, 13265)	(44777)	0.000381	0.000707	0.000239	0.625612	885.23896
79	(44777, 6505)	(13265)	0.000306	0.000889	0.000239	0.779472	877.10983
78	(44777)	(13265)	0.000707	0.000889	0.000467	0.661092	743.90130
82	(37746, 4806)	(41798)	0.000326	0.001308	0.000205	0.627863	479.90944
71	(38304, 48209, 31629)	(15980)	0.000298	0.001467	0.000204	0.685089	467.02155

There a set of rules that stand out, when looking from the *lift* statistic.

From the *__first__* and *__second__, *we can see that they were built from the same* set. *The* __second__* rule has a higher *conviction* value with almost the same *lift* which makes it stand out between the two. The *__third__* rule is built from a subset of the previous one. In it, we can see that the *support* from this rule is almost 2 times the *support* of the *__second__* rule.

It seems logical that product 44777 is highly related to product 13265. This rule also seems very promising.

In [ ]:
```
index = [80, 79, 78, 82, 71]
for i, idx in enumerate(index, start=1):
    print(f"--- Association Rule #{i} --- Lift = \{promissing_rules.loc[idx, 'li
    print_rule(promissing_rules, idx)
    print()
```

```
--- Association Rule #1 --- Lift = 885.24 ---
Snacks Sharp Cheddar Sticks Cheese
Manchego
==>
Giant Chocolate Cookies & Cream Ice Cream Bars

--- Association Rule #2 --- Lift = 877.11 ---
Giant Chocolate Cookies & Cream Ice Cream Bars
Snacks Sharp Cheddar Sticks Cheese
==>
Manchego

--- Association Rule #3 --- Lift = 743.90 ---
Giant Chocolate Cookies & Cream Ice Cream Bars
==>
Manchego

--- Association Rule #4 --- Lift = 479.91 ---
Green Writing Gel
Breaded Chicken Patties
==>
Unscented Glycerine Soap

--- Association Rule #5 --- Lift = 467.02 ---
Extra Sweet Iced Tea
Whole Wheat Blueberry Fig Bars
Pepperoncini Potato Chips
==>
Unsweetened Soymilk
```

(This analysis was made with the help of ChatGPT)
Rule #1 / #2 / #3 - *Manchego* can be used as a desert, or has a snack. *Giant Chocolate Cookies & Ice Cream Bars* is also a desert, or a mid-afternoon snack. This items are likely bought for gathering events of some kind.
Rule #4 - We can't find any logical relation for this rule.
Rule #5 - The items themselves don't seem to have a clear relation. Maybe someone that is trying to start a diet or a change in their meals but still need

We came across alot of difficulty in order to interpret these results. Even in our comments, we can't actually know if they are accurate without more proper studies.

For this section, we obtained the Maximal and Closed Itemsets for the same level of support used in the previous sections. These are:

- Closed itemsets are a subset of the all the frequent itemsets.
- Maximal itemsets are a subset of the Closed itemsets.

In [ ]:
```
def get_closed_itemset(fi):
    set_size = fi.itemsets.apply(len)
    min_set_size, max_set_size = set_size.min(), set_size.max()
    closed_fi = pd.DataFrame(columns=["support", "itemsets"])
    for size in range(min_set_size, max_set_size+1):
        sets = fi[set_size==size]
        super_sets = fi[set_size==size+1]
        for i in sets.index:
            row = sets.loc[i]
            itemset = row.itemsets
            matching_supersets = super_sets[super_sets.itemsets.apply(lambda ite
            if (len(matching_supersets)==0) or \
            (matching_supersets.support!=row.support).all():
                closed_fi.loc[i] = row
    return closed_fi

def get_maximal_itemset(fi):
    set_size = fi.itemsets.apply(len)
    min_set_size, max_set_size = set_size.min(), set_size.max()
    max_fi = pd.DataFrame(columns=["support", "itemsets"])
    for size in range(min_set_size, max_set_size+1):
        sets = fi[set_size==size]
        super_sets = fi[set_size==size+1]
        for i in sets.index:
            row = sets.loc[i]
            itemset = row.itemsets
            matching_supersets = super_sets[super_sets.itemsets.apply(lambda ite
            if len(matching_supersets)==0:
                max_fi.loc[i] = row
    return max_fi
```

In [ ]:
```
fi_closed = get_closed_itemset(fi)
fi.shape, fi_closed.shape
```

Out[ ]: ((39492, 2), (39492, 2))

In [ ]:
```
fi_max = get_maximal_itemset(fi_closed)
fi_closed.shape, fi_max.shape
```

Out[ ]: ((39492, 2), (32597, 2))

From what can be seen in the previous code, the *frequent itemsets* generated for the support value threshold of 0.02% were already closed itemsets (same shape). Because of this, the most relevant rules for the close itemset have already been generated.

In [ ]:
```
#Create a filter of the rules for the itemset
def filter_rules_by_itemset(rules, fi):
    rule_filter = [False for _ in range(len(rules))]
    sets_in_rules = list(map(lambda ant, con: ant|con, rules["antecedents"], rul
    for i, set_ in enumerate(sets_in_rules):
        if  (fi.itemsets.apply(lambda itemset: set_==itemset)).any():
            rule_filter[i]=True
    return rules[rule_filter]
```

```
In [ ]: #Print out the amount of rules
print("Total Nº of rules =", rules.shape[0])
max_itemset_rules = filter_rules_by_itemset(rules, fi_max)
print("Maximal Itemset Nº of Rules =", max_itemset_rules.shape[0])
#Get the maximum itemset rules with a filter on their lift
promissing_max_itemset_rules = max_itemset_rules[max_itemset_rules.lift > max_it
```

```
Total Nº of rules = 85
Maximal Itemset Nº of Rules = 52
```

```
In [ ]: index = [80, 79, 82, 71, 84]
for i, idx in enumerate(index, start=1):
    print(f"--- Association Rule #{i} --- Lift = \{promissing_max_itemset_rules.
    print_rule(promissing_max_itemset_rules, idx)
    print()
```

```
--- Association Rule #1 --- Lift = 885.24 ---
Snacks Sharp Cheddar Sticks Cheese
Manchego
==>
Giant Chocolate Cookies & Cream Ice Cream Bars

--- Association Rule #2 --- Lift = 877.11 ---
Giant Chocolate Cookies & Cream Ice Cream Bars
Snacks Sharp Cheddar Sticks Cheese
==>
Manchego

--- Association Rule #3 --- Lift = 479.91 ---
Green Writing Gel
Breaded Chicken Patties
==>
Unscented Glycerine Soap

--- Association Rule #4 --- Lift = 467.02 ---
Extra Sweet Iced Tea
Whole Wheat Blueberry Fig Bars
Pepperoncini Potato Chips
==>
Unsweetened Soymilk

--- Association Rule #5 --- Lift = 457.70 ---
Organic Strawberry Smoothie
Soft-Picks  - 40 CT
==>
Distilled Water
```

The top 4 rules had been commented previously, therefore we wont be commenting further for these and only for the fifth rule.

(With the help of ***ChatGPT**) Rule #5** - Distilled water is used majorly for cleaning. Soft picks are used for dental cleaning. Smoothies are items.