Rendu DMV Guillaume COBAT Master SNS

November 12, 2023

1 Task 0: warming up

1.0.1 Comprendre la structure des données.

Dans ce script, je réalise une analyse de données en utilisant Python. Voici un aperçu des principales étapes que je suis en train de suivre :

1. **Importation des bibliothèques :** J'importe les bibliothèques NumPy, Pandas, Matplotlib, Seaborn, Random et Time pour effectuer diverses opérations de traitement de données, de visualisation et de gestion du temps.
2. **Lecture des données :** Je lis le fichier CSV appelé "order_products___train.csv" depuis un chemin spécifié.
3. **Calcul des statistiques :** J'effectue le calcul de plusieurs statistiques sur mes données, telles que le nombre de commandes différentes, le nombre de produits différents, la moyenne de produits par commande, le nombre minimal de produits par commande et le nombre maximal de produits par commande.
4. **Stockage des résultats :** Les résultats sont stockés dans un dictionnaire pour une utilisation ultérieure.
5. **Création d'un graphique :** Création d'un graphique en barre avec échelle logarithmique pour une meilleure lisibilité des résultats.

Ce script me permet d'explorer et de visualiser les caractéristiques de mes données, ce qui est essentiel pour prendre des décisions éclairées dans l'analyse de données.

[ ]:
```
import numpy as np  # Importation de la bibliothèque NumPy pour les opérations␣
↪sur les tableaux
import pandas as pd  # Importation de la bibliothèque Pandas pour la␣
↪manipulation de données en DataFrame
import matplotlib.pyplot as plt  # Importation de la bibliothèque Matplotlib␣
↪pour la création de graphiques
import seaborn as sns  # Importation de la bibliothèque Seaborn pour la␣
↪visualisation de données statistiques
import random  # Importation de la bibliothèque Random pour générer des nombres␣
↪aléatoires
import time  # Importation de la bibliothèque Time pour gérer le temps
from itertools import combinations  # Importation de la fonction combinations␣
↪de la bibliothèque itertools pour générer toutes les combinaisons possibles
```

```
from scipy.stats import kendalltau  # Importation du test de corrélation de␣
 ↪Kendall depuis la bibliothèque SciPy
import plotly.express as px  # Importation de la bibliothèque Plotly Express␣
 ↪pour la création de visualisations interactives
from skmine.itemsets import LCM  # Importation de l'algorithme LCM pour␣
 ↪l'extraction d'ensembles fréquents
import networkx as nx  # Importation de la bibliothèque NetworkX pour la␣
 ↪manipulation de graphes
from collections import Counter  # Importation de la classe Counter pour le␣
 ↪comptage d'éléments
from skmine.itemsets import SLIM  # Importation de l'algorithme SLIM pour␣
 ↪l'extraction d'ensembles fréquents
import os  # Importation de la bibliothèque os pour les opérations liées au␣
 ↪système d'exploitation
import re  # Importation de la bibliothèque re pour les expressions régulières
from operator import itemgetter  # Importation de la fonction itemgetter de la␣
 ↪bibliothèque operator pour extraire des éléments spécifiques

# La seed permet de faire en sorte que les échantillons aléatoires restent␣
 ↪toujours les mêmes, et donc ne changent pas l'analyse.
randomSeed = 123
# Définie la seed pour random
random.seed(randomSeed)
# Définie également la seed pour numpy
np.random.seed(randomSeed)

color = sns.color_palette()  # Création d'une palette de couleurs à partir de␣
 ↪Seaborn
pd.options.mode.chained_assignment = None  # Désactivation des avertissements␣
 ↪liés à l'assignation en chaîne dans Pandas
```

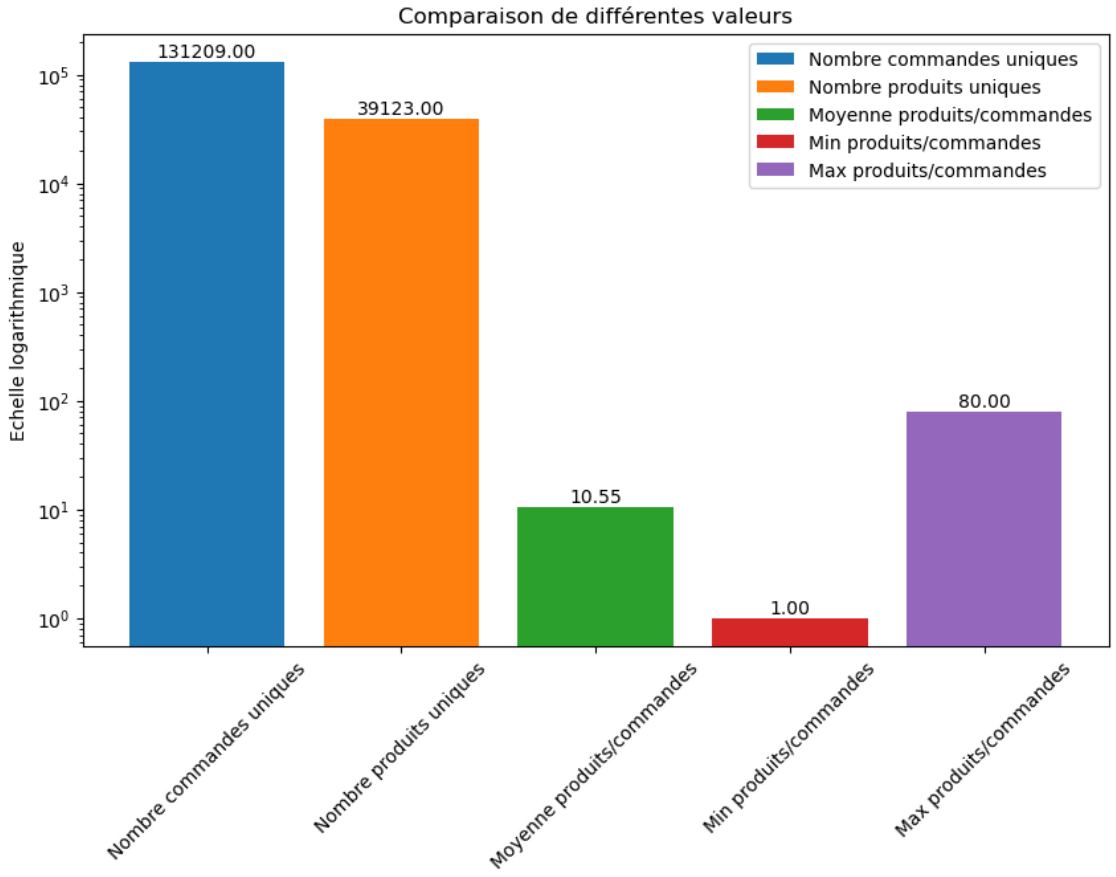
[ ]:
```
# Lecture du fichier CSV "order_products__train.csv" depuis le chemin spécifié
order_products_train_df = pd.read_csv("../datas/order_products__train.csv")

# Affichage des premières lignes du DataFrame pour examiner les données
order_products_train_df.head()
```

[ ]:

	order_id	product_id	add_to_cart_order	reordered
0	1	49302	1	1
1	1	11109	2	1
2	1	10246	3	0
3	1	49683	4	0
4	1	43633	5	1

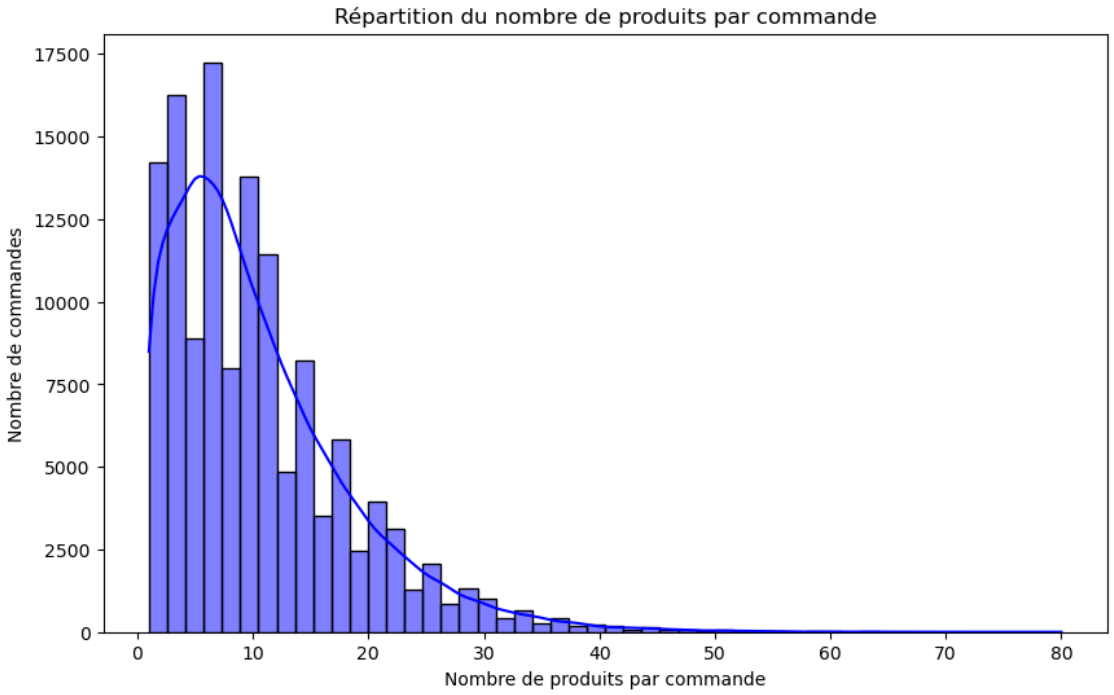
[ ]:
```
# Création d'une liste vide pour stocker les résultats
results = []
```

```
# Calcul des différentes statistiques sur l'échantillon de données
nombre_orders_differents = order_products_train_df['order_id'].nunique()
nombre_produits_differents = order_products_train_df['product_id'].nunique()
order_product_counts = order_products_train_df.
 ↪groupby('order_id')['product_id'].count()
moyenne_produits_par_order = order_product_counts.mean()
nombre_minimal_produits_par_order = order_product_counts.min()
nombre_maximal_produits_par_order = order_product_counts.max()

# Stockage des résultats dans un dictionnaire
result = {
    'Sample Size': 'Your_Sample_Size',  # Remarque : il semble manquer une␣
 ↪valeur pour la taille de l'échantillon
    'Unique Orders': nombre_orders_differents,
    'Unique Products': nombre_produits_differents,
    'Mean Products per Order': moyenne_produits_par_order,
    'Min Products per Order': nombre_minimal_produits_par_order,
    'Max Products per Order': nombre_maximal_produits_par_order
}

results.append(result)

# Création d'un DataFrame à partir des résultats
results_df = pd.DataFrame(results)

# Définition des valeurs et des étiquettes pour les barres du graphique
values = results_df.loc[0, ['Unique Orders', 'Unique Products', 'Mean Products␣
 ↪per Order', 'Min Products per Order', 'Max Products per Order']]
labels = ['Nombre commandes uniques', 'Nombre produits uniques', 'Moyenne␣
 ↪produits/commandes', 'Min produits/commandes', 'Max produits/commandes']

# Création d'un graphique avec une échelle logarithmique sur l'axe des y et des␣
 ↪annotations au-dessus de chaque barre
plt.figure(figsize=(10, 6))

for i in range(len(values)):
    plt.bar(i, values[i], label=labels[i])
    plt.text(i, values[i], f'{values[i]:.2f}', ha='center', va='bottom')

plt.yscale("log")
plt.ylabel('Echelle logarithmique')
plt.xticks(np.arange(len(labels)), labels, rotation=45)
plt.title('Comparaison de différentes valeurs')
plt.legend()

plt.show()  # Affichage du graphique
```



[ ]:
```
# Utilisation de la méthode nunique() pour compter les valeurs uniques dans la colonne "order_id"
nombre_orders_differents = order_products_train_df['order_id'].nunique()

# Utilisation de la méthode nunique() pour compter les valeurs uniques dans la colonne "product_id"
nombre_produits_differents = order_products_train_df['product_id'].nunique()

# Groupement des données par "order_id" et comptage du nombre de produits par order
order_product_counts = order_products_train_df.groupby('order_id')['product_id'].count()

# Calcul de la moyenne, du nombre minimal et du nombre maximal de produits par order
moyenne_produits_par_order = order_product_counts.mean()
nombre_minimal_produits_par_order = order_product_counts.min()
nombre_maximal_produits_par_order = order_product_counts.max()
```

```
# Tracé d'un histogramme pour visualiser la répartition du nombre de produits␣
 ↪par order
plt.figure(figsize=(10, 6))  # Définition de la taille de la figure
sns.histplot(order_product_counts, bins=50, kde=True, color='b')  # Tracé de␣
 ↪l'histogramme avec KDE
plt.title('Répartition du nombre de produits par commande')  # Titre du␣
 ↪graphique
plt.xlabel('Nombre de produits par commande')  # Étiquette de l'axe des x
plt.ylabel('Nombre de commandes')  # Étiquette de l'axe des y
plt.show()  # Affichage du graphique
```



2 Task 1: Frequent itemsets

2.1 Question 1.1:

2.1.1 Préparation des données pour l'Algorithme Apriori

Dans cette section, je vais expliquer la démarche que je suis en train de suivre pour préparer nos données avant d'appliquer l'algorithme Apriori. La préparation des données est essentielle pour que l'algorithme fonctionne efficacement et nous permette de découvrir des règles d'association utiles.

1. Tout d'abord, je commence par regrouper nos données en fonction de l'identifiant de chaque commande, ce qui correspond à la colonne 'order_id'. Cela nous permet de rassembler tous les produits inclus dans une même commande.

2. Ensuite, je transforme les produits associés à chaque commande en listes. Ainsi, pour chaque

commande, nous obtenons une liste des produits qui ont été achetés. Cette étape simplifie notre analyse, car nous pouvons maintenant voir clairement quels produits ont été inclus dans chaque commande.

3. Pour faciliter la manipulation ultérieure des données, je réinitialise l'index de nos données pour créer un nouveau DataFrame où les identifiants de commande sont remis à l'index par défaut.
4. J'extrais ensuite uniquement la colonne 'product_id', car c'est la seule information dont nous avons besoin pour l'analyse des règles d'association.
5. Enfin, je convertis la colonne 'product_id' en une liste de listes. Chaque liste correspond aux produits achetés dans une commande unique. Ce format est celui attendu par l'algorithme Apriori pour effectuer son analyse.

Cette préparation des données nous mettra en bonne position pour appliquer l'algorithme Apriori et découvrir des tendances d'achat et des combinaisons de produits fréquemment associées.

Maintenant que nos données sont prêtes, nous pouvons passer à l'application de l'algorithme Apriori.

[ ]:
```
# Regroupement des "product_id" par "order_id" et les concaténation dans une liste
apriori_product_train_dataset = order_products_train_df.groupby('order_id')['product_id'].apply(list).reset_index()

# Obtention de la liste de listes sans les colonnes
# Ce format est celui attendu par l'algorithme apriori
apriori_product_train_dataset = apriori_product_train_dataset['product_id'].tolist()

# Affichage des premières lignes du nouveau dataset
print(apriori_product_train_dataset[:5])  # Affiche les 5 premières lignes du dataset sans colonnes
```

```
[[49302, 11109, 10246, 49683, 43633, 13176, 47209, 22035], [39612, 19660, 49235,
43086, 46620, 34497, 48679, 46979], [11913, 18159, 4461, 21616, 23622, 32433,
28842, 42625, 39693], [20574, 30391, 40706, 25610, 27966, 24489, 39275], [8859,
19731, 43654, 13176, 4357, 37664, 34065, 35951, 43560, 9896, 27509, 15455,
27966, 47601, 40396, 35042, 40986, 1939, 46313, 329, 30776, 36695, 27683, 15995,
27344, 47333, 48287, 45204, 24964, 18117, 46413, 34126, 9373, 22935, 46720,
44479, 790, 18441, 45007, 20520, 7461, 26317, 3880, 36364, 32463, 41387, 31066,
17747, 25659]]
```

2.2 Question 1.2:

2.2.1 Analyse du Support des Produits

Dans cette section, j'effectue une analyse du support des produits à partir de données. Voici un résumé des principales étapes de mon code :

Étape 1 : Création d'une Liste Unique de Produits

J'extrais tous les produits de la liste de listes et les rassemble en une seule liste, ce qui me permet d'obtenir une liste unique de tous les produits.

Étape 2 : Calcul du Support de Chaque Produit

J'utilise la bibliothèque "Counter" pour compter la fréquence de chaque produit dans la liste unique. Cela me permet de déterminer le support de chaque produit, c'est-à-dire combien de fois il apparaît dans les transactions.

Étape 3 : Recherche du Support Maximum et Minimum

Je trouve le support maximum et minimum parmi tous les produits pour mieux comprendre la variabilité du support des produits.

Étape 4 : Création d'un Graphique

Je crée un graphique à barres pour afficher le support maximum et minimum sur une échelle logarithmique de l'axe des y. Cette représentation visuelle permet de mieux comparer les valeurs de support.

Étape 5 : Catégorisation du Support

Je catégorise les produits en fonction de leur support en utilisant des catégories personnalisées. Les produits sont répartis dans des groupes tels que "0-1", "1-3", "3-6", "6-10", "10-20", "20-50", "50-1312", et "1312+". Cette catégorisation permet de mieux comprendre la distribution du support des produits.

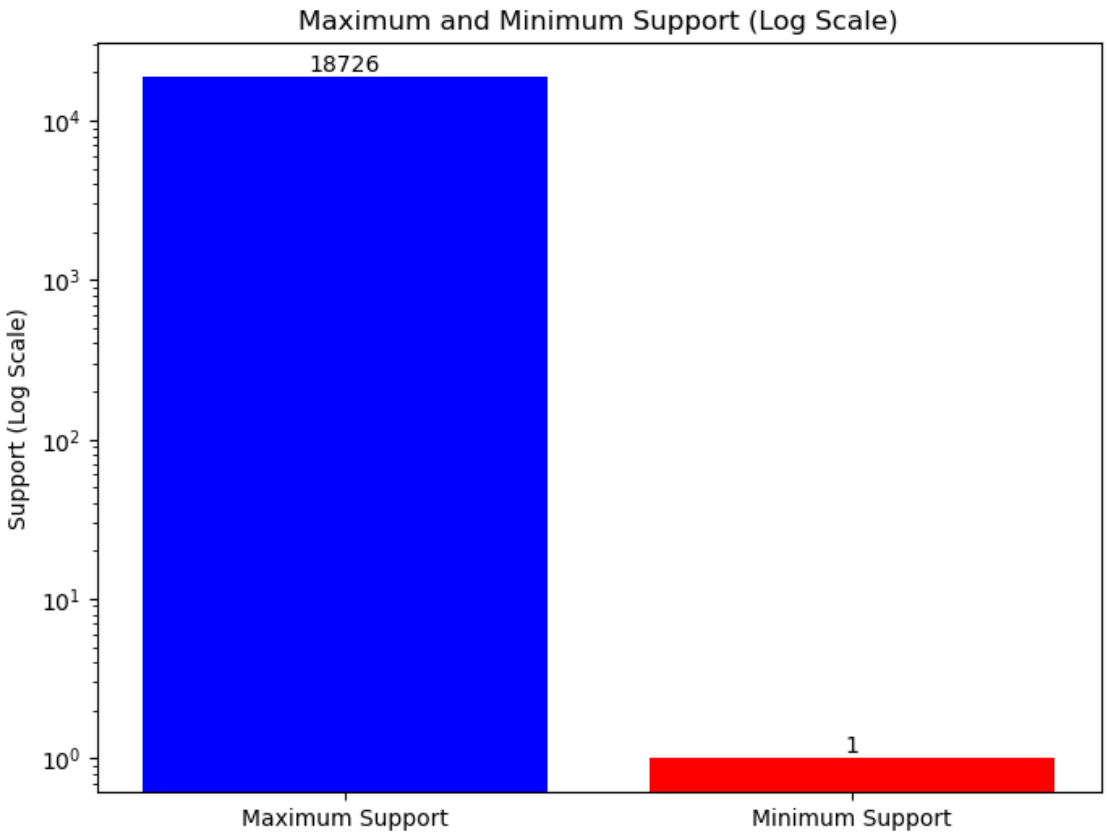
Étape 6 : Création d'un Graphique en Camembert

Je crée un graphique en camembert pour représenter la distribution du support des produits dans les différentes catégories. Chaque catégorie est affichée avec sa part relative dans l'ensemble des produits.

Ces étapes d'analyse du support des produits me permettent de mieux comprendre la popularité et la variabilité des produits dans les transactions, ce qui peut être précieux pour la prise de décision et la planification des stocks.

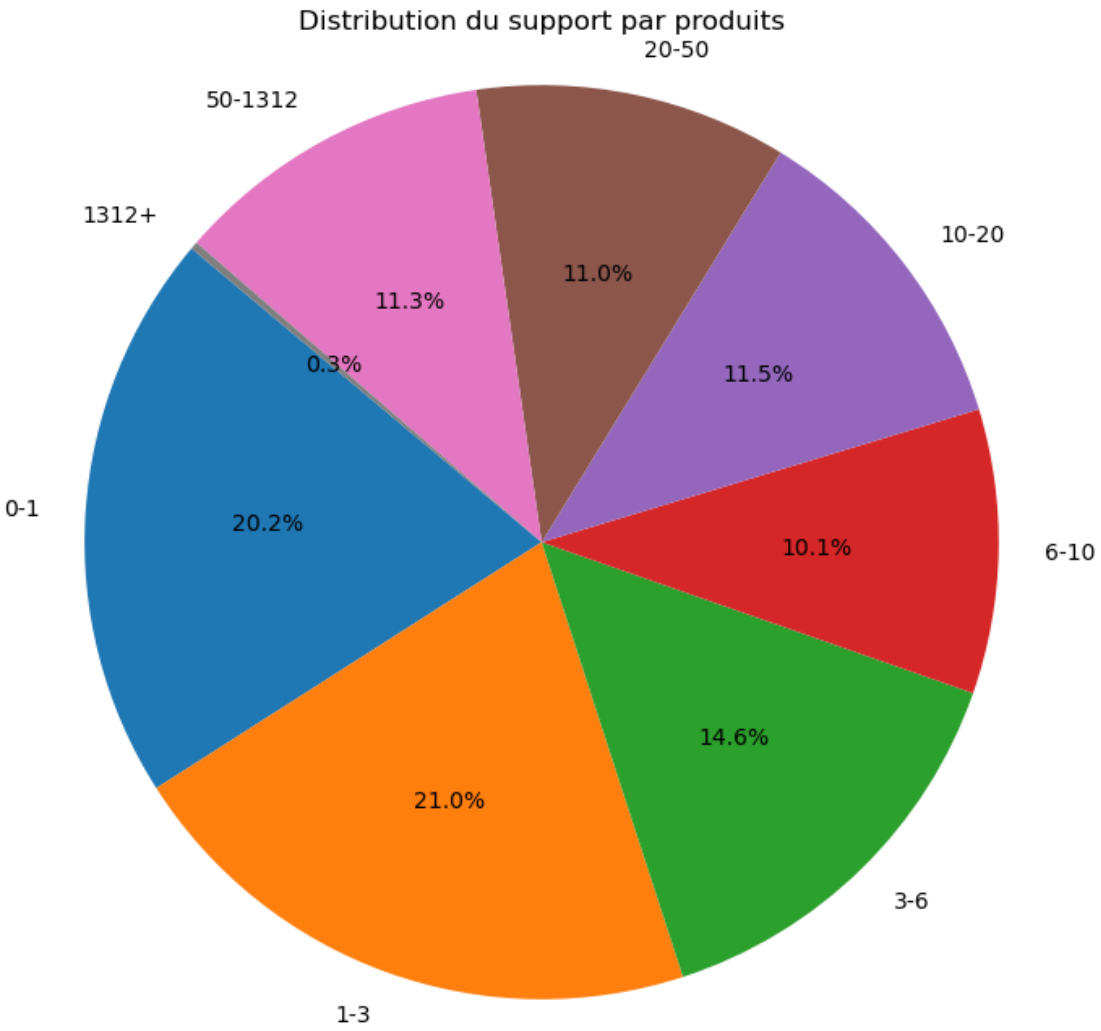
[ ]:
```
# Aplatir la liste de listes en une seule liste
all_products_list = [product for product_list in apriori_product_train_dataset␣
 ↪for product in product_list]

# Utilisation de Counter pour compter la fréquence de chaque produit
C1 = Counter(all_products_list)

# Trouve le support maximum et minimum
maximum_support = max(C1.values())
minimum_support = min(C1.values())

# Crée un graphique à barres avec une échelle logarithmique pour l'axe des y
plt.figure(figsize=(8, 6))  # Définition de la taille de la figure
plt.bar(['Maximum Support', 'Minimum Support'], [maximum_support,␣
 ↪minimum_support], color=['b', 'r'])  # Tracé des barres
plt.yscale("log")  # Utilisation d'une échelle logarithmique pour l'axe des y
```

```
# Ajout d'étiquettes aux barres
for i, v in enumerate([maximum_support, minimum_support]):
    plt.text(i, v, str(v), ha='center', va='bottom')

plt.title('Maximum and Minimum Support (Log Scale)')  # Titre du graphique
plt.ylabel('Support (Log Scale)')  # Étiquette de l'axe des y
plt.show()  # Affichage du graphique
```



[ ]:
```
nombre_orders_differents = order_products_train_df['order_id'].nunique()

last_categorie = round(nombre_orders_differents * (1/100))

# Crée des catégories de support personnalisées
support_categories = {
    "0-1": 0,
    "1-3": 0,
    "3-6": 0,
    "6-10": 0,
    "10-20": 0,
```

```
    "20-50": 0,
    "50-" + str(last_categorie): 0,
    str(last_categorie) + "+": 0,
}

# Assign products to custom support categories
for support in C1.values():
    if 0 <= support <= 1:
        support_categories["0-1"] += 1
    elif 1 <= support <= 3:
        support_categories["1-3"] += 1
    elif 3 < support <= 6:
        support_categories["3-6"] += 1
    elif 6 < support <= 10:
        support_categories["6-10"] += 1
    elif 10 < support <= 20:
        support_categories["10-20"] += 1
    elif 20 < support <= 50:
        support_categories["20-50"] += 1
    elif 50 < support <= last_categorie:
        support_categories["50-" + str(last_categorie)] += 1
    else:
        support_categories[str(last_categorie) + "+"] += 1


# Create a pie chart
plt.figure(figsize=(8, 8))
labels = support_categories.keys()
sizes = support_categories.values()
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=140)
plt.title("Distribution du support par produits")
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()
```



2.2.2 Interprétation

Ce graphique permet de voir la distribution du support par produit, d'après ce graphique, on peut se dire qu'un item est fréquent si celui-ci a un support > 1312 qui correspond à 1% de notre jeu de donnée, puisque c'est le cas pour plus de 0.3% de nos produits, ce qui est déjà assez conséquent vu la taille de notre jeu de donnée.

2.3 Question 1.3 Apriori Algorithm

[ ]:
```python
def apriori_gen(Lk, k):
    """
    Generate candidate itemsets of size k+1 from frequent itemsets of size k.

    Args:
        Lk (list): List of frequent itemsets of size k.
```

```
        k (int): Size of the frequent itemsets.

    Returns:
        list: Candidate itemsets of size k+1.
    """
    # Fonction pour générer des itemsets candidats de taille k+1 à partir de Lk
    sortedLk = list(Lk)
    sortedLk.sort()  # Trie les éléments de Lk (sortie lexicographique)

    Ckp1 = []  # Crée la liste des itemsets candidats de taille k+1

    # Boucle pour combiner les éléments de Lk ayant le même préfixe k-1
    for itemset1, itemset2 in combinations(sortedLk, 2):
        if itemset1[:-1] == itemset2[:-1]:
            Ckp1.append(itemset1 + [itemset2[-1]])

    # Élimine les éléments de Ckp1 ayant des sous-ensembles qui ne sont pas␣
 ↪dans Lk
    result = []
    for itemset in Ckp1:
        subsets = combinations(itemset, len(itemset) - 1)
        keep = True

        for s in subsets:
            s = list(s)
            if s not in Lk:
                keep = False
                break

        if keep:
            result.append(itemset)

    return result


def apriori(data, minSup):
    """
    Main Apriori function to extract frequent itemsets.

    Args:
        data (list of lists): Input transactional data.
        minSup (int): Minimum support threshold.

    Returns:
        tuple: List of frequent itemsets for each size and their corresponding␣
 ↪support.
    """
```

```
    # Fonction principale d'Apriori pour extraire les itemsets fréquents
    print("Le jeu de données contient :", len(data), "transactions")
    print("Extraction avec un support minimum de :", minSup, "(", 100 * minSup /
↪ len(data), " % )")

    flatten_db = [item for transaction in data for item in transaction]
    C1 = Counter(flatten_db)  # Compte la fréquence des 1-itemsets

    L1 = []  # Liste des 1-itemsets fréquents
    supportList = []  # Liste de support pour les itemsets fréquents

    # Élimine les 1-itemsets peu fréquents
    for item, support in C1.items():
        if support >= minSup:
            L1.append([item])
            supportList.append(support)

    results = [L1]  # Stocke les itemsets fréquents
    k = 1

    while len(results) >= k and k < 6:
        # Génère des itemsets candidats
        Ckp1 = apriori_gen(results[k - 1], k)

        Ckp1_count = [0] * len(Ckp1)

        # Parcours les données, compte un par transaction contenant l'itemset
        for transaction in data:
            st = set(transaction)
            for i, itemset in enumerate(Ckp1):
                si = set(itemset)
                if si.issubset(st):
                    Ckp1_count[i] += 1

        Lkp1 = []  # Liste des itemsets fréquents de taille k+1

        for i, itemset in enumerate(Ckp1):
            if Ckp1_count[i] >= minSup:
                Lkp1.append(itemset)
                supportList.append(Ckp1_count[i])

        # Si des itemsets sont stockés dans Lk+1, ajoutez-les au résultat final
        if Lkp1:
            results.append(Lkp1)

        k = k + 1
```

```
    # Retourne le résultat temporaire
    return results, supportList

def prettyPrintFIS(fisList, supportList):
    """
    Pretty print frequent itemsets and their support.

    Args:
        fisList (list): List of frequent itemsets.
        supportList (list): List of corresponding support values.
    """
    i = 0
    k = 1

    for Lk in fisList:
        print(k, "-itemsets")

        for itemset in Lk:
            print(itemset, " : ", supportList[i])
            i += 1

        k += 1

    print("TOTAL: ", i + 1, "itemsets")
```

2.3.1 Calcul de Similarité avec le Test de Kendall

Dans cette section, j'utilise le test de Kendall pour mesurer la similarité entre des séries de données. Le test de Kendall, également connu sous le nom de coefficient de corrélation de Kendall, est une mesure de corrélation non paramétrique utilisée pour évaluer la similarité ou l'ordre relatif entre deux ensembles de données.

Le test de Kendall compare les paires d'éléments dans les deux séries de données et mesure le nombre de concordances et de discordances entre ces paires. Plus la similarité est grande, moins il y a de discordances, et la valeur de la corrélation de Kendall est proche de 1. La valeur du test est de 0 lorsqu'il n'y a aucune corrélation entre les 2 tableaux.

Dans cette section, j'utilise le test de Kendall pour mesurer la similarité entre des séries de données. Voici un aperçu des principales étapes de mon code :

1. **Définition des Tailles :** Je définis deux tailles, `taille_comparatif` et `petite_taille_comparatif`, qui serviront à calculer les scores de similarité et les scores d'éléments communs.
2. **Fonction de Calcul de Similarité :** Je crée une fonction `calculate_similarity_with_kendall` qui calcule la similarité entre deux séries de données en utilisant le test de Kendall. Plus la distance de Kendall est proche de 1, plus la similarité est grande.
3. **Fonction de Comparaison avec une Référence :** Je crée une fonction

`compare_to_reference_with_kendall` qui compare plusieurs séries de données par rapport à une série de référence. Les séries sont alignées sur l'index de la série de référence, et des scores de similarité sont calculés pour les 10 premiers éléments et pour l'ensemble des éléments.

4. **Calcul des Éléments Communs :** Je crée une fonction `calculate_common_items` qui compte le nombre d'éléments communs entre la série de référence et une autre série.

Ces fonctions et étapes me permettent de mesurer la similarité entre les séries de données en utilisant le test de Kendall, et le nombre d'éléments commun entre les 2 listes. Le test de Kendall n'est pas idéal, et peut être un peu biasié dans notre cas, le fait de le corréler avec le nombre de valeurs identiques entre les listes permet d'avoir plus de piste pour évaluer le rapprochement entre les différents échantillon et ma liste référente.

[ ]:
```python
taille_comparatif = 100
petite_taille_comparatif = 10

def calculate_similarity_with_kendall(reference, column):
    """
    Calculates Kendall's Tau similarity between two sequences.

    Args:
        reference (list): Reference sequence.
        column (list): Sequence to compare with the reference.

    Returns:
        float: Kendall's Tau similarity score.
    """
    tau, _ = kendalltau(reference, column)
    similarity = tau  # Plus la distance de Kendall se rapproche de 1, plus la similarité est grande
    return similarity


def compare_to_reference_with_kendall(reference, dataframes):
    """
    Compares multiple sequences to a reference sequence using Kendall's Tau.

    Args:
        reference (list): Reference sequence.
        dataframes (dict): Dictionary containing DataFrame series to compare.

    Returns:
        tuple: Kendall's Tau scores for the first 10 elements, all elements, and common items percentage.
    """
    kendall_10_scores = []
    kendall_all_scores = []
    common_items_scores = []
```

```
    for df_name, df_series in dataframes.items():
        liste = df_series.iloc[0]
        # Vérifie si la liste commence par "["
        if str(liste).startswith("["):
            # Align DataFrames based on the reference index
            df_aligned = df_series.align(reference, axis=0, join='inner')[0]

            # Calcul du nombre d'items communs
            common_items = calculate_common_items(reference, df_aligned)

            # Transformation en score sur 100 pour les éléments communs
            max_items = len(reference)
            common_items_score = (common_items / max_items) * taille_comparatif

            # Calcul de la distance de Kendall pour les 10 premiers éléments
            kendall_reference_10 = reference[:petite_taille_comparatif]
            kendall_column_10 = df_aligned.tolist()[:petite_taille_comparatif]
            kendall_score_10 =␣
 ↪calculate_similarity_with_kendall(kendall_reference_10, kendall_column_10)

            # Calcul de la distance de Kendall pour l'ensemble des éléments
            kendall_score_all = calculate_similarity_with_kendall(reference,␣
 ↪df_aligned)

            kendall_10_scores.append(kendall_score_10)
            kendall_all_scores.append(kendall_score_all)
            common_items_scores.append(common_items_score)

    return kendall_10_scores, kendall_all_scores, common_items_scores


def calculate_common_items(reference, column):
    """
    Calculates the number of common items between two sequences.

    Args:
        reference (list): Reference sequence.
        column (list): Sequence to compare with the reference.

    Returns:
        int: Number of common items.
    """
    common_items = sum(1 for item in reference if item in column.tolist())
    return common_items
```

2.3.2 Évaluation de la Ressemblance entre les Données

Dans cette section, j'évalue la ressemblance entre différents échantillons de données en utilisant le test de Kendall et le nombre d'items en commun et j'analyse les résultats. Voici un aperçu des principales étapes de mon code :

Étape 1 : Boucle d'Échantillonnage et de Calcul de Similarité

Je boucle sur les différentes tailles d'échantillon, en créant des échantillons aléatoires à partir de mes données. Pour chaque échantillon, j'applique l'algorithme apriori pour extraire les itemsets fréquents (FIS) et calcule la similarité de Kendall entre les FIS de l'échantillon et ceux du jeu de données complet.

Étape 2 : Stockage des Résultats

Je stocke les scores de similarité de Kendall, le nombre d'items présents dans les échantillons par rapport aux résultats de apriori sur les données totale, et les temps d'exécution pour chaque échantillon.

Étape 3 : Comparaison des Résultats

Je compare les scores de Kendall des échantillons à ceux du jeu de données complet, en utilisant la colonne "FIS" comme référence. Les résultats sont stockés dans des tableaux pour l'analyse.

Étape 4 : Création de Graphiques

Je crée des graphiques pour visualiser les scores de similarité de Kendall, le nombre d'items présents, et les temps d'exécution en fonction de la taille de l'échantillon.

Ces étapes me permettent d'évaluer la ressemblance entre les données et de comprendre comment la taille de l'échantillon affecte les résultats de l'algorithme apriori.

[ ]:
```
# Vérifie si order_products_train_df n'est pas une liste
if not isinstance(order_products_train_df, list):

    # Groupement des données par "order_id" et concaténation des "product_id"␣
 ↪dans une liste
    order_products_train_df = order_products_train_df.
 ↪groupby('order_id')['product_id'].apply(list).reset_index()

    order_products_train_df = order_products_train_df['product_id'].tolist()

# Crée un dictionnaire pour stocker les top 100 itemsets
top_100_itemsets_dict = {}

# Tailles d'échantillons à tester (incluant la taille complète du jeu de␣
 ↪données)
sample_sizes = [int(len(order_products_train_df)), 500, 1000, 5000, 15000,␣
 ↪25000, 50000]
#sample_sizes = [int(len(order_products_train_df) / 20), 500, 1000, 5000, 10000]

isStart = True
```

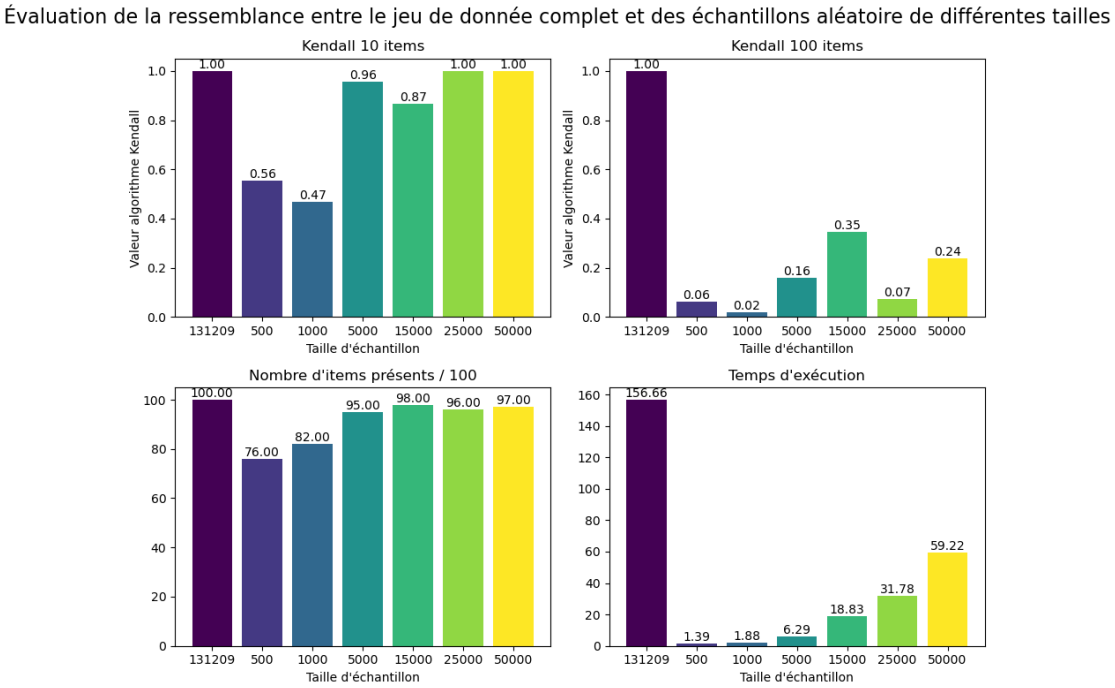
```
# Initialise des listes pour stocker les résultats
kendal_10_scores = []
kendal_100_scores = []
items_present = []
execution_times = []
# Crée un dictionnaire pour stocker les FIS_Support_tab de chaque exécution
FIS_Support_tabs_all = {}

# Boucle sur les différentes tailles d'échantillon
for sample_size in sample_sizes:

    # Crée un échantillon aléatoire de la taille spécifiée
    random_sample = random.sample(order_products_train_df, sample_size)

    # Mesure le temps d'exécution de l'algorithme apriori
    start_time = time.time()
    FIS, support = apriori(random_sample, (0.01 * sample_size))
    execution_time = time.time() - start_time
    execution_times.append(execution_time)


    # Aplatit la liste de listes
    flattened_FIS = [itemset for sublist in FIS for itemset in sublist]

    # Crée un DataFrame pandas avec une colonne "FIS" et une colonne "support"
    FIS_tab = pd.DataFrame({"FIS": flattened_FIS})
    Support_tab = pd.DataFrame({"support": support})
    FIS_Support_tab = pd.concat([FIS_tab, Support_tab], axis=1)

    # Trie les lignes par la colonne "support" en ordre décroissant
    FIS_Support_tab = FIS_Support_tab.sort_values(by="support", ascending=False)

    # Sélectionne les 100 premières lignes
    FIS_Support_tab = FIS_Support_tab.head(taille_comparatif)

    # Stocke le FIS_Support_tab dans le dictionnaire
    FIS_Support_tabs_all[sample_size] = FIS_Support_tab

    # Si c'est la première itération, enregistre les top 100 itemsets du jeu de␣
 ↪données complet
    if isStart:
        top_100_dataset_complet = FIS_Support_tab
        isStart = False
    else:
        # Enregistre les top 100 itemsets de l'échantillon actuel dans le␣
 ↪dictionnaire
```

```
        top_100_itemsets_dict[sample_size] = FIS_Support_tab

# Crée un tableau à deux colonnes avec FIS dans la première colonne et support␣
 ↪dans la deuxième colonne
concatenated_df = pd.DataFrame()
concatenated_df = pd.concat([concatenated_df, top_100_dataset_complet],␣
 ↪axis=1)# Boucle pour concaténer les DataFrames du dictionnaire

# Sélectionne la colonne "FIS" de top_100_dataset_complet comme colonne de␣
 ↪référence
reference_column = top_100_dataset_complet['FIS']
reference_column = reference_column.reset_index(drop=True)

# Crée un dictionnaire des tableaux à comparer
dataframes_to_compare = {df_name: df['FIS'] for df_name, df in␣
 ↪top_100_itemsets_dict.items()}

if len(dataframes_to_compare) < taille_comparatif:
    missing_rows = taille_comparatif - len(dataframes_to_compare)
    missing_data = pd.Series([-1] * missing_rows, name="FIS")
    missing_df = pd.DataFrame({"FIS": missing_data})
    dataframes_to_compare.update({"Missing Data": missing_df})

# Réindexe toutes les séries pour qu'elles aient le même index
for df_name, df_series in dataframes_to_compare.items():
    dataframes_to_compare[df_name] = df_series.reset_index(drop=True)

# Appelle la fonction pour comparer la colonne "FIS" de chaque tableau avec␣
 ↪décalage
results = compare_to_reference_with_kendall(reference_column,␣
 ↪dataframes_to_compare)

# Stocke les scores Kendall et le nombre d'items présents
for index, sample_size in enumerate(sample_sizes):
    if index == 0:
        kendal_10_scores.append(1.0)
        kendal_100_scores.append(1.0)
        items_present.append(taille_comparatif)


        continue  # Passe au prochain tour de boucle sans traiter le premier␣
 ↪élément
    # Traitement pour les éléments autres que le premier
    kendal_10_scores.append(results[0][index - 1])
    kendal_100_scores.append(results[1][index - 1])
    items_present.append(results[2][index - 1])
```

```
# Listes de données et titres pour chaque diagramme
data_sets = [
    {
        'data': kendal_10_scores,
        'title': "Kendall 10 items",
    },
    {
        'data': kendal_100_scores,
        'title': "Kendall 100 items",
    },
    {
        'data': items_present,
        'title': "Nombre d'items présents / 100",
    },
    {
        'data': execution_times,
        'title': "Temps d'exécution",
    }
]

# Crée un subplot de 2 lignes et 2 colonnes
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

for i, dataset in enumerate(data_sets):
    data = dataset['data']
    colors = plt.cm.viridis(np.linspace(0, 1, len(data)))  # Utilise une carte␣
 ↪de couleurs (viridis ici) pour obtenir un gradient de couleurs

    # Définit l'emplacement du graphique dans le subplot en fonction de␣
 ↪l'indice i
    row = i // 2
    col = i % 2

    ax = axes[row, col]
    ax.bar(range(len(data)), data, color=colors)
    ax.set_title(dataset['title'])

    ax.set_xticks(range(len(data)))
    ax.set_xticklabels(sample_sizes)
    ax.set_xlabel("Taille d'échantillon")
    ax.set_ylabel("Valeur algorithme Kendall")

    # Ajoute les valeurs arrondies au-dessus des barres
    for j, value in enumerate(data):
        rounded_value = round(value, 2)  # Arrondit à 2 chiffres après la␣
 ↪virgule
```

```
        ax.text(j, value, f"{rounded_value:.2f}", ha='center', va='bottom')

fig.suptitle("Évaluation de la ressemblance entre le jeu de donnée complet et␣
 ↪des échantillons aléatoire de différentes tailles", fontsize=16)

# Enlève le label de l'axe Y pour les 2 graphiques du bas
axes[1, 0].set_ylabel("")
axes[1, 1].set_ylabel("")
# Ajuste l'espacement entre les sous-graphiques pour une meilleure lisibilité
plt.tight_layout()

# Affiche les sous-graphiques
plt.show()
```

```
Le jeu de données contient : 131209 transactions
Extraction avec un support minimum de : 1312.09 ( 1.0  % )
Le jeu de données contient : 500 transactions
Extraction avec un support minimum de : 5.0 ( 1.0  % )
Le jeu de données contient : 1000 transactions
Extraction avec un support minimum de : 10.0 ( 1.0  % )
Le jeu de données contient : 5000 transactions
Extraction avec un support minimum de : 50.0 ( 1.0  % )
Le jeu de données contient : 15000 transactions
Extraction avec un support minimum de : 150.0 ( 1.0  % )
Le jeu de données contient : 25000 transactions
Extraction avec un support minimum de : 250.0 ( 1.0  % )
Le jeu de données contient : 50000 transactions
Extraction avec un support minimum de : 500.0 ( 1.0  % )
```



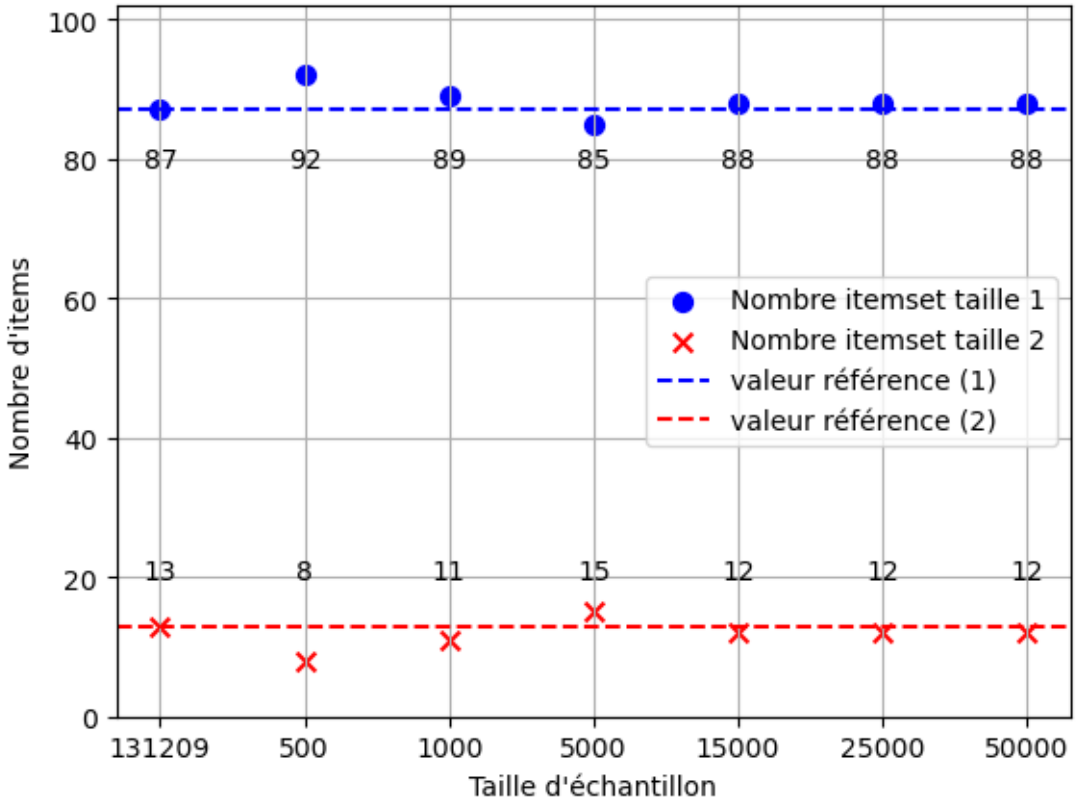
2.3.3 Interprétation

- Le premier graphique, situé en haut à gauche, met en évidence qu'à partir d'un échantillon de 5000 éléments, la majorité des 10 premiers éléments fréquents du jeu de données complet sont présents, et ce, dans un ordre similaire. Lorsque l'échantillon atteint 25000 éléments, les dix éléments les plus fréquents sont tous correctement classés.
- Le graphique supérieur droit révèle que l'algorithme de Kendall affiche des résultats relativement médiocres en termes de corrélation avec le jeu de données complet. Toutefois, une amélioration de la corrélation est observée à partir de 5000 éléments, atteignant son maximum à 15000 éléments. L'aléatoire joue ici un rôle, d'où le fait que les meilleures résultats ne soient pas forcéments observées à 50000 éléments.
- Le troisième graphique, en bas à gauche, indique qu'à partir de 5000 éléments, plus de 95 itemsets parmi les 100 itemsets fréquents se retrouvent à la fois dans le jeu de données complet et dans nos échantillons. Cette constatation est rassurante, car elle suggère que même si le positionnement exact des éléments n'est pas parfaitement corrélé avec le jeu de données complet, les éléments proches ne diffèrent probablement que de 1 ou 2 positions dans le résultat, ce qui a une assez grosse importance pour les résultats de kendall, mais beaucoup moindre pour notre analyse. En effet, les dix premiers itemsets fréquents étant bien classés, le classement des itemsets suivant est moins important.
- Enfin, le dernier graphique met en évidence l'impact significatif de la taille de l'échantillon sur le temps d'exécution. Alors que l'algorithme complet prend environ 3 minutes à s'exécuter sur l'ensemble des données, il s'exécute en moins de 20 secondes pour les échantillons contenant moins de 15000 éléments.

Vérification de la corrélation du nombre d'itemsets de taille 1 et 2 entre les différents échantillons dans le top 100

[ ]:
```
# Crée des listes pour stocker les données
sample_sizes = []
count_of_ones = []
count_of_twos = []

# Obtient toutes les valeurs de "sample size"
sample_sizes_all = list(FIS_Support_tabs_all.keys())

# Crée une séquence uniformément espacée pour l'axe des abscisses
x_values = np.arange(len(sample_sizes_all))

# Parcours le dictionnaire de DataFrames
for sample_size, df in FIS_Support_tabs_all.items():

    # Compte le nombre de listes de 1 item et 2 items
    num_single_item_lists = df['FIS'].apply(lambda x: sum(1 for itemset in x if␣
 ↪isinstance(x, list) and len(x) == 1))

    count_of_ones_value = num_single_item_lists.sum()
    count_of_twos_value = len(num_single_item_lists) - count_of_ones_value

    sample_sizes.append(sample_size)  # Ajoute la taille de l'échantillon
    count_of_ones.append(count_of_ones_value)  # Ajoute le nombre d'itemsets de␣
 ↪taille 1
    count_of_twos.append(count_of_twos_value)  # Ajoute le nombre d'itemsets de␣
 ↪taille 2


# Crée un nuage de points
plt.scatter(x_values, count_of_ones, label='Nombre itemset taille 1',␣
 ↪color='b', marker='o', s=50)
plt.scatter(x_values, count_of_twos, label='Nombre itemset taille 2',␣
 ↪color='r', marker='x', s=50)

# Ajoute les valeurs au-dessus de chaque point
for i in range(len(x_values)):
    plt.annotate(f'{count_of_ones[i]}', (x_values[i], count_of_ones[0] - 12),␣
 ↪textcoords="offset points", xytext=(0,10), ha='center')
    plt.annotate(f'{count_of_twos[i]}', (x_values[i], count_of_twos[0] + 3),␣
 ↪textcoords="offset points", xytext=(0,10), ha='center')

# Augmente la taille de l'axe des ordonnées
plt.ylim(0, max(count_of_ones + count_of_twos) + 10)
```

```
# Trace des lignes horizontales pour le premier échantillon
plt.axhline(y=count_of_ones[0], color='blue', linestyle='--', label='valeur␣
 ↪référence (1)', xmin=0, xmax=1)
plt.axhline(y=count_of_twos[0], color='red', linestyle='--', label='valeur␣
 ↪référence (2)', xmin=0, xmax=1)

# Définit les étiquettes des axes en utilisant les valeurs de "sample size"␣
 ↪comme étiquettes
plt.xticks(x_values, sample_sizes_all)
plt.xlabel('Taille d\'échantillon')
plt.ylabel('Nombre d\'items')
plt.legend()
plt.grid(True)
plt.show()
```



2.3.4 Interprétation et Conclusion

Ici, nous remarquons que la proportions des résultats d'apriori d'itemsets de taille 1 et 2 est quasiment parfaite à partir d'un échantillon de 15000 éléments aléatoires. Ainsi, après avoir pris en compte tous ces éléments, il est raisonnable de considérer qu'un échantillon aléatoire de **15000 éléments** représente de manière adéquate le jeu de données complet, offrant des gains considérables

en termes de temps de calcul.

2.3.5 Analyse des Itemsets Fréquents de 2 items

Dans cette section, j'analyse les itemsets fréquents extraits de l'échantillon de taille 15 000 en utilisant les informations sur les produits. Voici un aperçu des principales étapes de mon code :

Étape 1 : Récupération des Données de l'Échantillon

Je récupère les colonnes "FIS" (Itemsets Fréquents) et "support" pour l'échantillon de taille 15 000 à partir des résultats précédemment stockés.

Étape 2 : Préparation des Données sur les Produits

J'importe les données des fichiers "products.csv", "aisles.csv", et "departments.csv" pour obtenir des informations détaillées sur les produits, les rayons et les départements. Je crée des dictionnaires pour mapper les numéros de rayons et de départements à leurs noms respectifs.

Étape 3 : Analyse des Itemsets

Je parcours le dictionnaire contenant les itemsets et leur support. Pour chaque itemset, je filtre ceux qui contiennent plus d'un article. Ensuite, je rassemble des informations sur les produits inclus dans l'itemset, notamment leur nom, l'allée et le département auxquels ils appartiennent.

Étape 4 : Affichage des Résultats

J'affiche les informations sur les itemsets fréquents, leur support, et les produits qu'ils contiennent. Cela me permet de comprendre quels produits sont souvent achetés ensemble et dans quel contexte (rayon et département).

Ces étapes facilitent l'exploration des relations entre les produits dans les itemsets fréquents et fournissent des informations utiles pour la recommandation de produits ou la compréhension des habitudes d'achat.

[ ]:
```
for sample_size, df in FIS_Support_tabs_all.items():
    if sample_size == 15000:
        # Obtient les colonnes "FIS" et "support" pour l'échantillon de taille␣
 ↪15000
        FIS_reduce = df['FIS']
        support_reduce = df['support']

# Initialise un dictionnaire vide pour stocker les itemsets et leur support
itemset_support_dict = {}

# Parcours les données réduites pour construire le dictionnaire
for i in range(len(FIS_reduce)):
    itemset_key = tuple(FIS_reduce.iloc[i]) # Convertit l'itemset en une clé de␣
 ↪type tuple
    itemset_value = support_reduce.iloc[i] # Obtient la valeur de support␣
 ↪correspondante
    itemset_support_dict[itemset_key] = itemset_value # Stocke l'itemset et son␣
 ↪support dans le dictionnaire
```

```
[ ]: def get_all_info_from_dict(itemset_support_dict):
        """
        Extracts additional information for each itemset from the provided␣
    ↪dictionary.

        Args:
            itemset_support_dict (dict): Dictionary containing itemsets and their␣
    ↪support.

        Returns:
            list: List of dictionaries with detailed information for each itemset.
        """
        products_file = "../datas/products.csv"
        aisles_file = "../datas/aisles.csv"
        departments_file = "../datas/departments.csv"

        # Charge les données du fichier "products.csv"
        products_df = pd.read_csv(products_file)

        # Charge les données du fichier "aisles.csv"
        aisles_df = pd.read_csv(aisles_file)
        # Crée un dictionnaire pour mapper les numéros d'allées à leurs noms
        aisles_dict = dict(zip(aisles_df['aisle_id'], aisles_df['aisle']))

        # Charge les données du fichier "departments.csv"
        departments_df = pd.read_csv(departments_file)
        # Crée un dictionnaire pour mapper les numéros de départements à leurs noms
        departments_dict = dict(zip(departments_df['department_id'],␣
    ↪departments_df['department']))

        # Initialise une liste pour stocker les informations
        itemset_info_list = []

        # Parcours le dictionnaire itemset_support_dict
        for itemset, support in itemset_support_dict.items():
            # Filtre les itemsets contenant plus de 2 articles
            if len(itemset) > 1:
                itemset_info = {}
                itemset_info["itemset"] = itemset
                itemset_info["support"] = support
                itemset_info["products"] = []

                for product_id in itemset:
                    # Recherche les informations du produit dans le DataFrame des␣
    ↪produits
                    product_info = products_df[products_df['product_id'] ==␣
    ↪int(product_id)]
```

```
                if not product_info.empty:
                    product_name = product_info['product_name'].values[0]
                    aisle_id = product_info['aisle_id'].values[0]
                    department_id = product_info['department_id'].values[0]

                    # Utilise les dictionnaires pour obtenir les noms d'allées␣
 ↪et de départements
                    aisle_name = aisles_dict.get(aisle_id, "Unknown Aisle")
                    department_name = departments_dict.get(department_id,␣
 ↪"Unknown Department")

                    product_info = {
                        "product_name": product_name,
                        "aisle_name": aisle_name,
                        "department_name": department_name
                    }
                    itemset_info["products"].append(product_info)

            itemset_info_list.append(itemset_info)

    return itemset_info_list


def display_top_itemsets(results, n=5):
    """
    Displays the top N itemsets with detailed information.

    Args:
        results (list): List of dictionaries containing itemset information.
        n (int): Number of top itemsets to display.
    """
    # Trie les résultats par support dans l'ordre décroissant
    results.sort(key=lambda x: x['support'], reverse=True)

    # Affiche uniquement les n premiers itemsets avec le support le plus élevé
    for itemset_info in results[:n]:
        print(f"Itemset: {itemset_info['itemset']}")
        print(f"Support: {itemset_info['support']}")
        print("Products:")
        for product_info in itemset_info['products']:
            print(f"- Product: {product_info['product_name']}, Aisle:␣
 ↪{product_info['aisle_name']}, Department: {product_info['department_name']}")
        print("\n")

def generate_treemap(itemset_info_list):
    """
    Generates a treemap visualization based on itemset information.
```

```
    Args:
        itemset_info_list (list): List of dictionaries containing itemset information.
    """
    product_line_numbers = {}

    itemset_df = pd.DataFrame(itemset_info_list)

    for itemset_info in itemset_info_list:
        itemset = itemset_info['itemset']
        for product_info in itemset_info['products']:
            product_name = product_info['product_name']
            line_number = itemset_df[itemset_df['itemset'] == itemset].index.tolist()

            # Si le produit existe déjà dans le dictionnaire, ajoute le numéro de ligne
            if product_name in product_line_numbers:
                product_line_numbers[product_name].extend(line_number)
            else:
                product_line_numbers[product_name] = line_number

    # Crée un DataFrame avec les informations de l'itemset
    itemset_df = pd.DataFrame(itemset_info_list)

    # Crée un DataFrame vide pour stocker les informations hiérarchiques
    hierarchical_df = pd.DataFrame()

    # Crée un dictionnaire pour stocker les produits déjà ajoutés
    added_products = {}

    # Parcours la liste des informations des itemsets
    for itemset_info in itemset_info_list:
        parent = "Root"  # Racine de l'arbre hiérarchique

        for product_info in itemset_info['products']:
            product_name = product_info['product_name']

            if product_name not in added_products:
                # Utilise .loc pour filtrer le DataFrame en fonction de 'product_name' et récupère les numéros de ligne
                line_numbers = product_line_numbers[product_name]

                # Ajoute des données au DataFrame hiérarchique
                hierarchical_df = pd.concat([
                    hierarchical_df,
```

```
                    pd.DataFrame({'parent': [parent], 'category':␣
↪[product_info['department_name']], 'value': [0]}),
                    pd.DataFrame({'parent': [product_info['department_name']],␣
↪'category': [product_info['aisle_name']], 'value': [0]}),
                    pd.DataFrame({'parent': [product_info['aisle_name']],␣
↪'category': [f"{product_name} ({', '.join(map(str, line_numbers))})"],␣
↪'value': [itemset_info['support']]}),
                ])
                parent = f"{product_name} ({', '.join(map(str, line_numbers))})"
                added_products[product_name] = line_numbers
            else:
                # Le produit a déjà été ajouté, récupère ses numéros de ligne
                line_numbers = added_products[product_name]

                # Met à jour la catégorie du produit avec tous les numéros de␣
↪ligne
                category = f"{product_name} ({', '.join(map(str,␣
↪line_numbers))})"
                hierarchical_df.loc[hierarchical_df['category'] == category,␣
↪'value'] += itemset_info['support']

    # Crée un treemap avec les informations hiérarchiques
    fig = px.treemap(hierarchical_df, path=['parent', 'category'],␣
↪values='value')

    # Ajuste la mise en page du treemap
    fig.update_layout(
        title="Treemap des Itemsets",
        margin=dict(t=50, l=0, r=0, b=0)
    )

    # Affiche le treemap
    fig.show()
```

[ ]:
```
itemset_info_list = get_all_info_from_dict(itemset_support_dict)

display_top_itemsets(itemset_info_list, n=5)

# Utilisation de la fonction
generate_treemap(itemset_info_list)
```

```
Itemset: (13176, 21137)
Support: 344
Products:
- Product: Bag of Organic Bananas, Aisle: fresh fruits, Department: produce
- Product: Organic Strawberries, Aisle: fresh fruits, Department: produce
```

```
Itemset: (13176, 47209)
Support: 287
Products:
- Product: Bag of Organic Bananas, Aisle: fresh fruits, Department: produce
- Product: Organic Hass Avocado, Aisle: fresh fruits, Department: produce


Itemset: (24852, 47626)
Support: 279
Products:
- Product: Banana, Aisle: fresh fruits, Department: produce
- Product: Large Lemon, Aisle: fresh fruits, Department: produce


Itemset: (13176, 21903)
Support: 269
Products:
- Product: Bag of Organic Bananas, Aisle: fresh fruits, Department: produce
- Product: Organic Baby Spinach, Aisle: packaged vegetables fruits, Department:
produce


Itemset: (21137, 24852)
Support: 259
Products:
- Product: Organic Strawberries, Aisle: fresh fruits, Department: produce
- Product: Banana, Aisle: fresh fruits, Department: produce
```

2.3.6 Analyse du graphique

Dans cette analyse, nous examinons un graphique qui révèle des tendances intéressantes dans les habitudes d'achat alimentaire aux États-Unis. Le graphique met en évidence les produits les plus fréquemment achetés ensemble, offrant des informations précieuses sur les préférences des consommateurs.

2.3.7 Tendances générales

D'après ce graphique, nous pouvons observer que les produits les plus fréquemment achetés ensemble dans ce magasin spécialisé sont les **fruits frais** et les **légumes emballés**. Cette tendance suggère une clientèle soucieuse de sa santé, cherchant à acheter des produits frais pour leur alimentation quotidienne.

2.3.8 Identification de la population cible

En se basant sur les associations spécifiques entre les produits, nous pouvons tenter d'identifier la population cible de ce magasin spécialisé. Les associations suivantes peuvent nous donner des

indications précieuses :

1. **Bananes et fraises** : L'association fréquente entre les bananes et les fraises suggère une clientèle probablement composée de **jeunes parents** ou de familles, car ces fruits sont couramment consommés par les enfants pour le petit-déjeuner, les collations ou les repas rapides.
2. **Fraises et framboises** : La combinaison de fraises et de framboises indique également une clientèle qui apprécie les fruits rouges, ce qui peut correspondre à des **familles** ou à des **jeunes adultes** soucieux de leur alimentation.
3. **Bananes et épinards pour bébé** : Enfin, la présence des bananes et des épinards pour bébé suggère que le magasin peut attirer des **parents de jeunes enfants**, car ces produits sont souvent associés à l'alimentation infantile.

En résumé, la population cible de ce magasin spécialisé semble être principalement composée de **jeunes parents**, de **familles** et de **jeunes adultes** soucieux de leur santé et de leur alimentation. Ces clients recherchent des fruits et légumes frais pour répondre à leurs besoins nutritionnels et à ceux de leur famille. Cette information peut être utile pour le magasin dans le développement de ses offres et de sa stratégie marketing pour mieux répondre aux besoins de sa clientèle cible.

2.4 Question 1.4 a.

Extraction d'Itemsets Fréquents avec LCM

Dans cette section, j'utilise l'algorithme LCM (Lexicographic Closed itemset Miner) pour extraire des itemsets fréquents. Voici un aperçu des principales étapes de mon code :

Étape 1 : Prétraitement des Données de Commandes

Je charge un jeu de données à partir du fichier "order_products__train.csv" que j'assigne à la variable `lcm_order_products_train_df`. Ensuite, je groupe les produits par commande, les concatène dans une liste, et stocke ces listes dans un nouveau dataset appelé `lcm_product_train_dataset`. Je supprime la colonne "order_id" pour obtenir une liste de listes de produits.

Étape 2 : Extraction d'Itemsets Fréquents avec LCM

Je crée une liste `lcm_product_train_list` contenant les listes de produits sans la colonne "order_id". Ensuite, j'utilise l'algorithme LCM pour extraire des itemsets fréquents à partir de cette liste en spécifiant un seuil de support minimum de 1312.

Ces étapes me permettent d'extraire des itemsets fréquents à partir des données afin d'obtenir des informations utiles sur les combinaisons fréquentes de produits.

[ ]:
```
# Charge les données du fichier "order_products__train.csv"
lcm_order_products_train_df = pd.read_csv("../datas/order_products__train.csv")

# Groupe les produits par "order_id" et les stocke dans une liste
lcm_product_train_dataset = lcm_order_products_train_df.
  ↪groupby('order_id')['product_id'].apply(list).reset_index()
lcm_product_train_list = lcm_product_train_dataset['product_id'].tolist()
```

```
# Utilise l'algorithme LCM pour extraire des itemsets fréquents avec un support␣
 ↪minimum de 1% de la taille du jeu de donnée
minSupp = round(0.01 * (len(lcm_product_train_list)))
lcm_results = LCM(min_supp=minSupp).fit_transform(lcm_product_train_list,␣
 ↪lexicographic_order=True)

# Accède à la colonne "itemsets" et s'assure qu'elle est sous forme de tuple
itemsets = lcm_results.iloc[:, 0]
itemsets = [tuple(x) if isinstance(x, (list, tuple)) else (x,) for x in␣
 ↪itemsets]

# Accède à la colonne "support"
supports = lcm_results["support"]

# Crée un dictionnaire avec les itemsets et les supports
lcm_itemset_support_dict = dict(zip(itemsets, supports))

# Obtient des informations détaillées sur les itemsets
lcm_info_results = get_all_info_from_dict(lcm_itemset_support_dict)

#Affiche les 5 itemsets de taille 2 avec le plus gros support
display_top_itemsets(lcm_info_results, n=5)

# Génère un treemap pour visualiser les relations hiérarchiques entre les␣
 ↪itemsets
generate_treemap(lcm_info_results)
```

```
Itemset: (13176, 21137)
Support: 3074
Products:
- Product: Bag of Organic Bananas, Aisle: fresh fruits, Department: produce
- Product: Organic Strawberries, Aisle: fresh fruits, Department: produce


Itemset: (13176, 47209)
Support: 2420
Products:
- Product: Bag of Organic Bananas, Aisle: fresh fruits, Department: produce
- Product: Organic Hass Avocado, Aisle: fresh fruits, Department: produce


Itemset: (13176, 21903)
Support: 2236
Products:
- Product: Bag of Organic Bananas, Aisle: fresh fruits, Department: produce
- Product: Organic Baby Spinach, Aisle: packaged vegetables fruits, Department:
produce
```

```
Itemset: (24852, 47766)
Support: 2216
Products:
- Product: Banana, Aisle: fresh fruits, Department: produce
- Product: Organic Avocado, Aisle: fresh fruits, Department: produce


Itemset: (21137, 24852)
Support: 2174
Products:
- Product: Organic Strawberries, Aisle: fresh fruits, Department: produce
- Product: Banana, Aisle: fresh fruits, Department: produce
```

2.4.1 Analyse des Tendances d'Achat de Fruits et Légumes dans un Magasin Spécialisé aux États-Unis - Comparaison avec un Autre Algorithme

En comparant les résultats de notre analyse précédente (apriori) avec ceux de lcm, nous constatons que ce dernier a identifié des associations similaires entre les produits.

2.4.2 Associations similaires

L'algorithme a mis en évidence des relations similaires entre les produits, telles que les **bananes et fraises** ainsi que les **fraises et framboises**, **bananes et épinards** ce qui confirme les tendances de consommation que nous avons identifiées précédemment. Ces associations suggèrent une clientèle composée principalement de **jeunes parents**, de **familles**, et de **jeunes adultes** soucieux de leur santé.

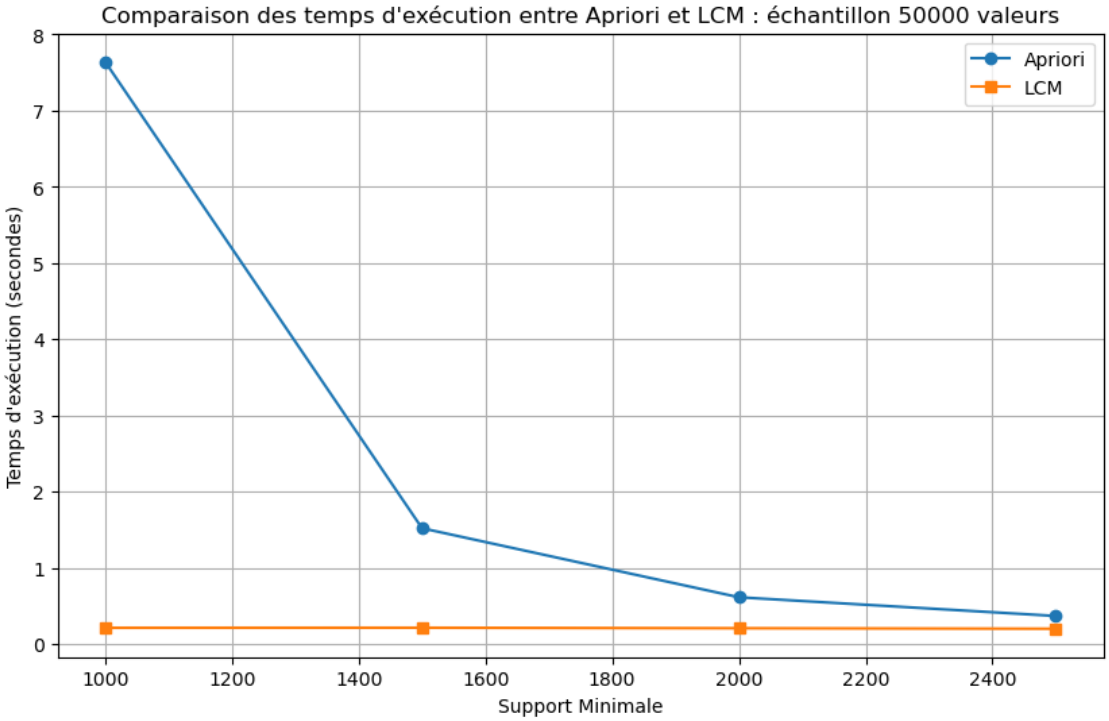
2.5 1.4 b. Test différence de temps

[ ]:
```
# Liste des valeurs de support minimaux à tester
min_support_values = [2500, 2000, 1500, 1000]

# Comme démontré précédemment, nous pouvons nous concentrer sur une portion␣
 ↪aléatoire de chaque dataset afin d'optimiser le temps d'exécution.
# Ici, nous prendrons 50000 éléments pour pouvoir observer une différence␣
 ↪significative en terme de temps d'exécution,
# sachant que plus le dataset est grand, plus la différence entre les deux␣
 ↪algorithmes sera importante.

# Regroupement des "product_id" par "order_id" et concaténation dans une liste,␣
 ↪puis sélection de 50000 valeurs aléatoires
order_products_train_df = pd.read_csv("../datas/order_products__train.csv")
apriori_product_train_dataset = order_products_train_df.
 ↪groupby('order_id')['product_id'].apply(list).reset_index()
```

```
apriori_product_train_dataset = apriori_product_train_dataset['product_id'].
 ↪tolist()
apriori_product_train_dataset = random.sample(apriori_product_train_dataset,␣
 ↪50000)

lcm_product_train_list = apriori_product_train_dataset


# Affichage de la taille des ensembles de données
print("Taille de apriori_product_train_dataset:",␣
 ↪len(apriori_product_train_dataset))
print("Taille de lcm_product_train_series:", len(lcm_product_train_list))

# Initialisation des listes pour stocker les temps d'exécution de chaque␣
 ↪algorithme
execution_times_apriori = []
execution_times_lcm = []

for min_support in min_support_values:
    # Mesure du temps d'exécution pour l'algorithme Apriori
    start_time = time.time()
    apriori_result = apriori(apriori_product_train_dataset, min_support)
    execution_time_apriori = time.time() - start_time
    execution_times_apriori.append(execution_time_apriori)

    # Mesure du temps d'exécution pour l'algorithme LCM
    start_time = time.time()
    lcm_result = LCM(min_supp = min_support).
 ↪fit_transform(lcm_product_train_list, lexicographic_order=True)
    execution_time_lcm = time.time() - start_time
    execution_times_lcm.append(execution_time_lcm)

# Trace un graphique pour comparer les temps d'exécution
plt.figure(figsize=(10, 6))
plt.plot(min_support_values, execution_times_apriori, label='Apriori',␣
 ↪marker='o')
plt.plot(min_support_values, execution_times_lcm, label='LCM', marker='s')
plt.xlabel('Support Minimale')
plt.ylabel('Temps d\'exécution (secondes)')
plt.title('Comparaison des temps d\'exécution entre Apriori et LCM :␣
 ↪échantillon 50000 valeurs')
plt.legend()
plt.grid(True)
plt.show()
```

```
Taille de apriori_product_train_dataset: 50000
Taille de lcm_product_train_series: 50000
```

```
Le jeu de données contient : 50000 transactions
Extraction avec un support minimum de : 2500 ( 5.0  % )
Le jeu de données contient : 50000 transactions
Extraction avec un support minimum de : 2000 ( 4.0  % )
Le jeu de données contient : 50000 transactions
Extraction avec un support minimum de : 1500 ( 3.0  % )
Le jeu de données contient : 50000 transactions
Extraction avec un support minimum de : 1000 ( 2.0  % )
```



2.5.1 Observations et Comparaison d'Apriori et de LCM

Dans cette section, je souhaite mettre en évidence des observations essentielles concernant les performances des algorithmes d'extraction d'itemsets fréquents, notamment Apriori et LCM, en fonction du support minimal (minsup). Ces observations sont cruciales pour comprendre les avantages et les inconvénients de chaque algorithme dans différentes situations.

2.5.2 Impact du Support Minimal sur le Temps d'Exécution

Il est clair que le choix du support minimal a un impact significatif sur le temps d'exécution des algorithmes. Voici quelques observations clés :

- **Faible Support Minimal :** Lorsque le support minimal est faible, Apriori nécessite considérablement plus de temps pour terminer son exécution par rapport à LCM. Cela est principalement dû à la nature de l'algorithme Apriori, qui explore de manière exhaustive toutes les combinaisons d'itemsets, ce qui peut devenir exponentiel en termes de temps.

- **LCM :** En revanche, LCM s'exécute presque instantanément avec un support minimal de cette taille. LCM utilise une approche basée sur la fermeture lexicographique pour extraire des itemsets fréquents, ce qui le rend plus efficace pour certaines valeurs de support minimal.

2.5.3 Limitations d'Apriori

Il est important de noter que, en raison de contraintes de temps, il n'est pas possible de tester Apriori avec un support minimal plus bas. Plus la taille de l'ensemble de données augmente, plus le temps d'exécution d'Apriori devient exponentiel et significativement plus long que celui de LCM.

2.5.4 Adaptation du Support Minimal

Ces observations soulignent l'importance de l'ajustement du support minimal en fonction des contraintes de temps et des objectifs d'analyse. Le support minimal doit être choisi avec soin pour trouver un équilibre entre la qualité des résultats et la performance. Il n'existe pas de seuil universel, et le choix dépendra des spécificités du projet et de la taille des données.

En résumé, ces observations mettent en lumière les avantages de LCM en termes de temps d'exécution.

2.6 Question 1.5

Le code effectue une analyse des données en utilisant diverses bibliothèques Python pour visualiser les relations entre les catégories de produits à partir des données extraites. Les étapes clés sont les suivantes :

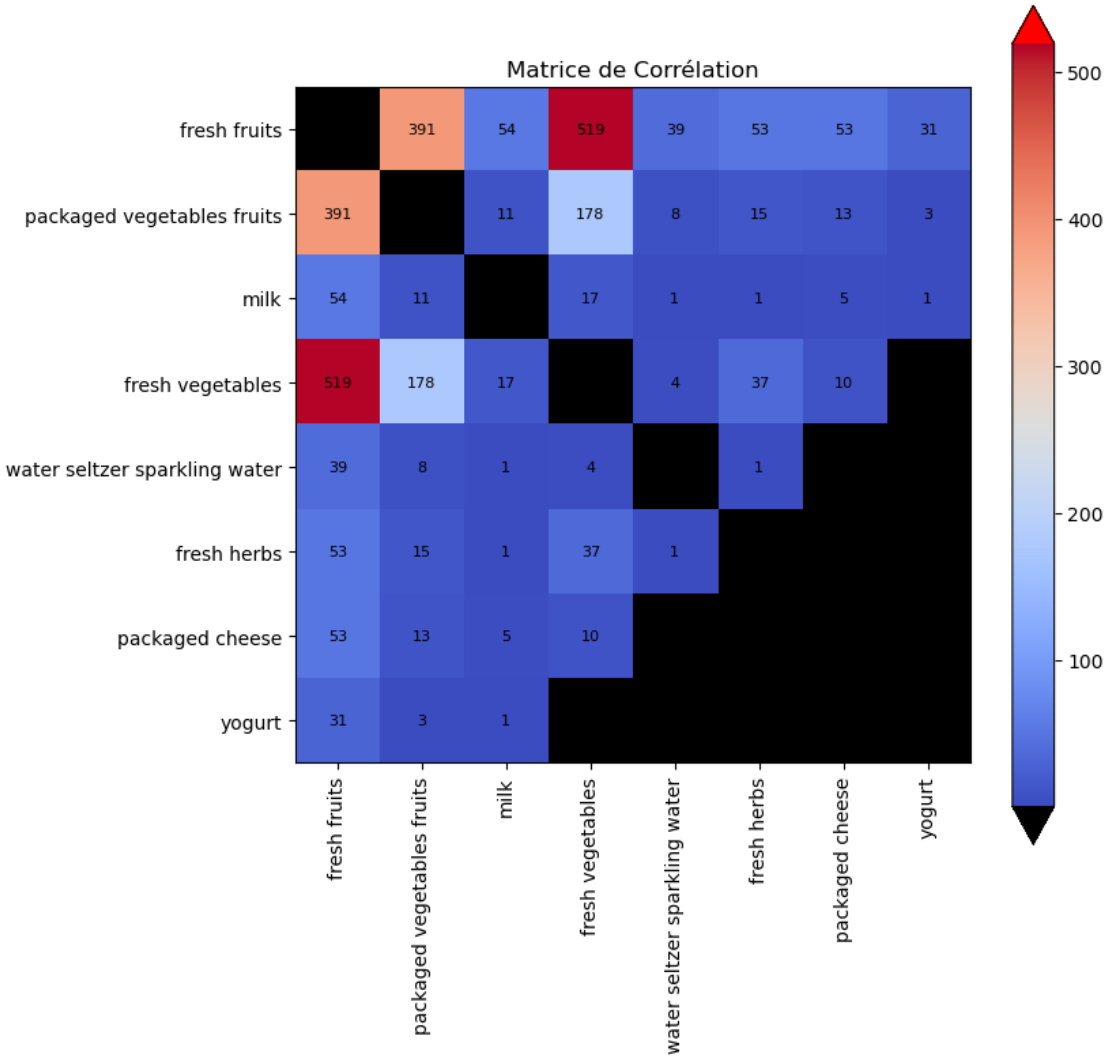
1. **Chargement des Données :** Les données sont chargées à partir d'un fichier CSV ("order_products__train.csv") et groupées par "order_id" pour former une liste de "product_id".
2. **Échantillonnage de Données :** L'algorithme LCM est appliqué à un échantillon de 5000 éléments parmi les données pour optimiser le temps de calcul.
3. **Extraction d'Itemsets Fréquents :** L'algorithme LCM est utilisé pour extraire des itemsets fréquents à partir de l'échantillon. Ces itemsets sont stockés dans la variable `lcm_results`.
4. **Création d'un Graphe Non Dirigé :** Un graphe non dirigé (`G`) est créé pour représenter les relations entre les "ailes" (catégories de produits) qui partagent des produits en commun. Des arêtes sont ajoutées entre les ailes lorsque des produits communs sont détectés.
5. **Création d'une Matrice de Corrélation :** Une matrice de corrélation est construite pour quantifier la relation entre les différentes ailes en utilisant le nombre de produits en commun. Cette matrice est ensuite affichée sous forme d'image, personnalisée avec une échelle de couleur.
6. **Personnalisation de la Colormap :** La colormap utilisée pour représenter la matrice de corrélation est ajustée pour avoir une grande variation entre 0 et la valeur maximale, avec des couleurs spécifiques pour les valeurs inférieures à 1 et les valeurs supérieures à la valeur maximale.

7. **Affichage de la Matrice de Corrélation :** La matrice de corrélation est affichée sous forme d'image avec les valeurs de corrélation indiquées dans chaque case. Une barre de couleur est ajoutée pour interpréter les valeurs de corrélation.

[ ]:
```
# Chargement des données du DataFrame depuis le fichier CSV
lcm_order_products_train_df = pd.read_csv("../datas/order_products__train.csv")

# Groupement des "product_id" par "order_id" et création d'une liste par␣
 ↪"order_id"
lcm_product_train_dataset = lcm_order_products_train_df.
 ↪groupby('order_id')['product_id'].apply(list).reset_index()

# Conversion du DataFrame en une liste de listes
lcm_product_train_list = lcm_product_train_dataset['product_id'].tolist()

# Sélection aléatoire de 5000 éléments dans la liste pour optimiser le temps de␣
 ↪calcul
lcm_product_train_list_5000 = random.sample(lcm_product_train_list, 5000)

# Utilisation de l'algorithme LCM pour extraire les itemsets fréquents (minSupp␣
 ↪adapté pour un temps d'exécution plus court)
lcm_results = LCM(min_supp=7).fit_transform(lcm_product_train_list_5000,␣
 ↪lexicographic_order=True)


# Conversion de la première colonne en tuples pour les itemsets
itemsets = lcm_results.iloc[:, 0]
itemsets = [tuple(x) if isinstance(x, (list, tuple)) else (x,) for x in␣
 ↪itemsets]

# Accès à la colonne "support" pour les supports correspondants aux itemsets
supports = lcm_results["support"]

# Création d'un dictionnaire associant les itemsets à leurs supports
lcm_itemset_support_dict = dict(zip(itemsets, supports))

# Extraction des informations complètes des itemsets
lcm_itemset_info_list = get_all_info_from_dict(lcm_itemset_support_dict)

# Création d'un graphe non dirigé pour représenter les relations entre les ailes
G = nx.Graph()

# Parcours les itemsets et ajout d'arêtes entre les ailes ayant des produits en␣
 ↪commun
for itemset in lcm_itemset_info_list:
    products = itemset["products"]
    ailes = [product["aisle_name"] for product in products]
```

```
    # Ajout d'arêtes entre les paires d'ailes dans le même itemset
    for i in range(len(ailes)):
        for j in range(i + 1, len(ailes)):
            aisle1 = ailes[i]
            aisle2 = ailes[j]
            if not G.has_edge(aisle1, aisle2):
                G.add_edge(aisle1, aisle2, weight=1)
            else:
                G[aisle1][aisle2]['weight'] += 1

# Obtention de la liste des noms d'ailes uniques
aisles = list(G.nodes)

# Création d'un tableau vide de la taille appropriée
table = np.zeros((len(aisles), len(aisles)))

# Remplissage du tableau avec les poids des arêtes pour les paires d'ailes␣
 ↪ayant des relations
for i, aisle1 in enumerate(aisles):
    for j, aisle2 in enumerate(aisles):
        if G.has_edge(aisle1, aisle2):
            table[i, j] = G[aisle1][aisle2]['weight']

# Création d'un sous-tableau avec uniquement les ailes ayant plus de 50␣
 ↪relations
filtered_indices = np.where(np.sum(table, axis=1) > 50)[0]
filtered_table = table[filtered_indices][:, filtered_indices]
filtered_aisles = [aisles[i] for i in filtered_indices]

# Création d'un nouveau graphe basé sur le sous-tableau filtré
G_filtered = nx.Graph()

for i, aisle1 in enumerate(filtered_aisles):
    for j, aisle2 in enumerate(filtered_aisles):
        if i != j and filtered_table[i, j] > 0:
            G_filtered.add_edge(aisle1, aisle2, weight=filtered_table[i, j])

# Création d'une matrice de corrélation
correlation_matrix = np.zeros((len(filtered_aisles), len(filtered_aisles)))

# Remplissage de la matrice de corrélation avec les valeurs de corrélation
for i in range(len(filtered_aisles)):
    for j in range(len(filtered_aisles)):
        if i != j:
            aisle1 = filtered_aisles[i]
            aisle2 = filtered_aisles[j]
```

```
            if G_filtered.has_edge(aisle1, aisle2):
                weight = G_filtered[aisle1][aisle2]['weight']
                correlation_matrix[i, j] = weight

# Affichage de la matrice de corrélation en tant qu'image en personnalisant␣
 ↪l'échelle de couleur
np.fill_diagonal(correlation_matrix, 0)

# Personnalisation de la colormap pour avoir une grande variation entre 0 et␣
 ↪1000, puis une variation plus faible au-delà
cmap = plt.get_cmap('coolwarm')
vmin = 1  # Définition de vmin à 1 pour que 0 relations soient en noir
vmax = correlation_matrix.max()
cmap.set_under('black')  # Couleur pour les valeurs inférieures à vmin (noir)
cmap.set_over('red')  # Couleur pour les valeurs supérieures à vmax (rouge)
norm = plt.Normalize(vmin, vmax)

plt.figure(figsize=(8, 8))
plt.imshow(correlation_matrix, cmap=cmap, norm=norm, interpolation='none')
plt.title("Matrice de Corrélation")

# Ajout du nombre de relations dans chaque case
for i in range(len(filtered_aisles)):
    for j in range(len(filtered_aisles)):
        if i != j:
            plt.text(j, i, str(int(correlation_matrix[i, j])), va='center',␣
 ↪ha='center', color='black', fontsize=8)

plt.colorbar(extend='both')  # Ajoute une barre de couleur avec des extensions␣
 ↪pour les valeurs inférieures et supérieures
plt.xticks(np.arange(len(filtered_aisles)), filtered_aisles, rotation=90)
plt.yticks(np.arange(len(filtered_aisles)), filtered_aisles)
plt.show()
```



2.6.1 Recommandations pour le Placement des Rayons en Magasin

Dans le but d'optimiser l'agencement du magasin, il est essentiel de prendre en compte les corrélations entre les catégories de produits. Plus les catégories de produits sont fortement corrélées, plus il est avantageux de les éloigner l'une de l'autre pour stimuler la découverte de nouveaux produits pour le client, et donc augmenter les profits du magasin.

Principales Recommandations :

1. **Fresh Fruits et Fresh Vegetables :** Les catégories "Fruits Frais" et "Légumes Frais" présentent une très forte corrélation, indiquant que les clients ont tendance à acheter ces produits ensemble. Pour optimiser le placement en magasin, il est recommandé de les éloigner l'une de l'autre. De plus, il serait judicieux d'analyser plus en détail quelles variétés spécifiques de fruits frais et de légumes frais sont fréquemment achetées conjointement.

2. **Packaged Vegetables Fruits et Fresh Fruits :** Une corrélation moins significative est

observée entre les catégories “Packaged Vegetables Fruits” (légumes et fruits emballés) et “Fresh Fruits” (fruits frais). Néanmoins, pour encourager les ventes croisées, il est conseillé de les placer à des endroits différents dans le magasin.

3. **Packaged Vegetables Fruits et Fresh Vegetables :** Une relation moins marquée existe entre les catégories “Packaged Vegetables Fruits” et “Fresh Vegetables” (légumes frais). Même si la corrélation est moins forte, il peut être pertinent de les éloigner légèrement l’une de l’autre afin de maximiser les ventes.

En résumé, il faudrait éloigner le plus possibles les fruits des légumes, et potentiellement éloigner un peu les légumes frais ainsi que les légumes emballés. En mettant en œuvre ces recommandations, le magasin peut augmenter les ventes en favorisant des achats complémentaires entre les catégories de produits.

2.7 Question 1.6

2.7.1 Analyse des Paires de Produits Fréquemment Achetés Ensemble

Le code vise à analyser les paires de produits fréquemment achetés ensemble, en se concentrant sur les paires de produits “fruits-légumes”, “fruits-fruits”, “légumes-légumes”. Voici un aperçu des étapes clés du code :

1. **Initialisation des Structures de Données :** Le code commence par initialiser des listes vides pour stocker les paires de produits des différentes catégories (“fruit-fruit,” “vegetable-vegetable,” et “fruit-vegetable”). De plus, des dictionnaires vides sont créés pour compter les occurrences des paires de produits dans chaque catégorie.

2. **Parcours des Itemsets et Comptage :** Le code parcourt les itemsets extraits précédemment, en vérifiant la catégorie de chaque produit dans l’itemset. Il compte le nombre de produits appartenant aux catégories “fruits” et “légumes.”

3. **Génération de Paires de Produits :** Pour les catégories “fruits” et “légumes,” le code génère toutes les paires possibles de produits. Par exemple, pour la catégorie “fruits,” toutes les combinaisons de paires de fruits sont générées.

4. **Mise à Jour des Compteurs :** Les paires de produits générées sont ensuite utilisées pour mettre à jour les compteurs correspondants. Par exemple, le compteur `fruit_fruit_pair_counts` est mis à jour avec les paires de fruits fréquemment achetées ensemble.

5. **Sélection des Meilleures Paires :** Les paires de produits sont triées par fréquence décroissante pour identifier les paires les plus fréquentes. Les 10 meilleures paires de “fruit-fruit,” “vegetable-vegetable,” et “fruit-vegetable” sont stockées dans les listes `top_fruit_fruit_pairs`, `top_vegetable_vegetable_pairs`, et `top_fruit_vegetable_pairs`, respectivement.

Le code fournit ainsi un aperçu des paires de produits les plus couramment achetées ensemble dans ces 3 catégories. Ces informations peuvent être utiles pour des recommandations de disposition des produits en magasin ou pour des analyses de marché.

[ ]:
```
# Crée des listes pour stocker les paires de produits de différentes catégories
fruit_fruit_pairs = []
```

```
vegetable_vegetable_pairs = []
fruit_vegetable_pairs = []

# Crée des dictionnaires pour compter les occurrences des paires de produits
fruit_fruit_pair_counts = Counter()
vegetable_vegetable_pair_counts = Counter()
fruit_vegetable_pair_counts = Counter()

# Parcours les itemsets et les informations
for itemset_info in lcm_itemset_info_list:
    # Compte le nombre de fruits, de légumes et d'autres produits dans l'itemset
    fruit_count = sum(1 for product_info in itemset_info['products'] if 'fresh␣
↪fruits' in product_info['aisle_name'])
    vegetable_count = sum(1 for product_info in itemset_info['products'] if␣
↪'fresh vegetables' in product_info['aisle_name'])

    if fruit_count >= 2:
        # Génère toutes les paires de fruits possibles dans cet itemset
        fruit_names = [product_info['product_name'] for product_info in␣
↪itemset_info['products'] if 'fresh fruits' in product_info['aisle_name']]
        fruit_pairs = [(product1, product2) for product1 in fruit_names for␣
↪product2 in fruit_names if product1 < product2]
        # Mets à jour le compteur de paires de fruits
        fruit_fruit_pair_counts.update(fruit_pairs)

    if vegetable_count >= 2:
        # Génère toutes les paires de légumes possibles dans cet itemset
        vegetable_names = [product_info['product_name'] for product_info in␣
↪itemset_info['products'] if 'fresh vegetables' in product_info['aisle_name']]
        vegetable_pairs = [(product1, product2) for product1 in vegetable_names␣
↪for product2 in vegetable_names if product1 < product2]
        # Mets à jour le compteur de paires de légumes
        vegetable_vegetable_pair_counts.update(vegetable_pairs)

    if fruit_count >= 1 and vegetable_count >= 1:
        # Génère toutes les paires de (fruit, légume) possibles dans cet itemset
        fruit_names = [product_info['product_name'] for product_info in␣
↪itemset_info['products'] if 'fresh fruits' in product_info['aisle_name']]
        vegetable_names = [product_info['product_name'] for product_info in␣
↪itemset_info['products'] if 'fresh vegetables' in product_info['aisle_name']]
        fruit_vegetable_pairs = [(fruit, vegetable) for fruit in fruit_names␣
↪for vegetable in vegetable_names]
        # Mets à jour le compteur de paires de (fruit, légume)
        fruit_vegetable_pair_counts.update(fruit_vegetable_pairs)

# Trie les paires de produits par fréquence décroissante
```

```
top_fruit_fruit_pairs = [pair for pair, count in fruit_fruit_pair_counts.
 ↪most_common(10)]
top_vegetable_vegetable_pairs = [pair for pair, count in␣
 ↪vegetable_vegetable_pair_counts.most_common(10)]
top_fruit_vegetable_pairs = [pair for pair, count in␣
 ↪fruit_vegetable_pair_counts.most_common(10)]
```

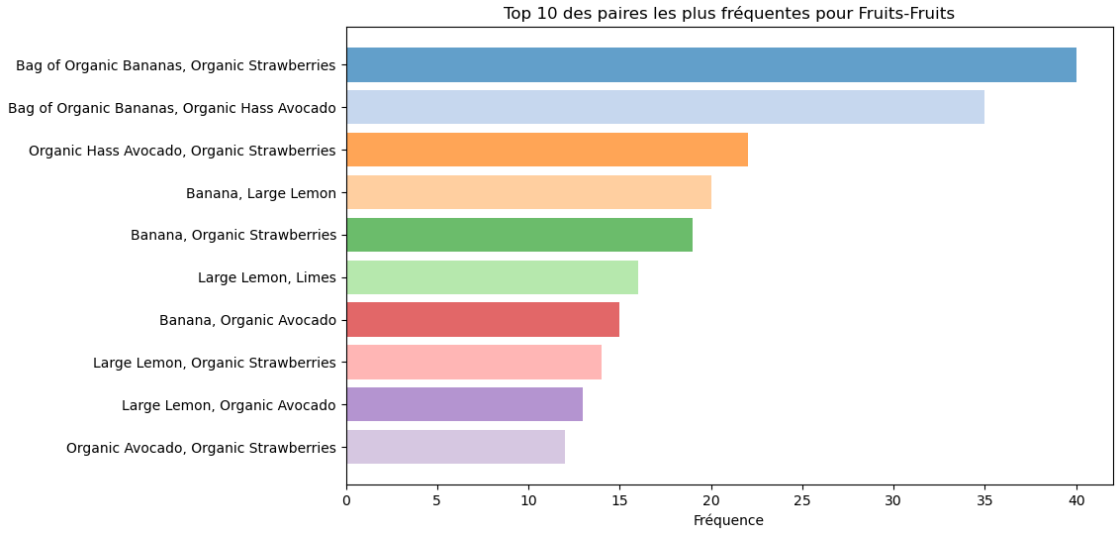
2.7.2 Analyse des Paires de Produits Fréquemment Achetés Ensemble - Visualisation

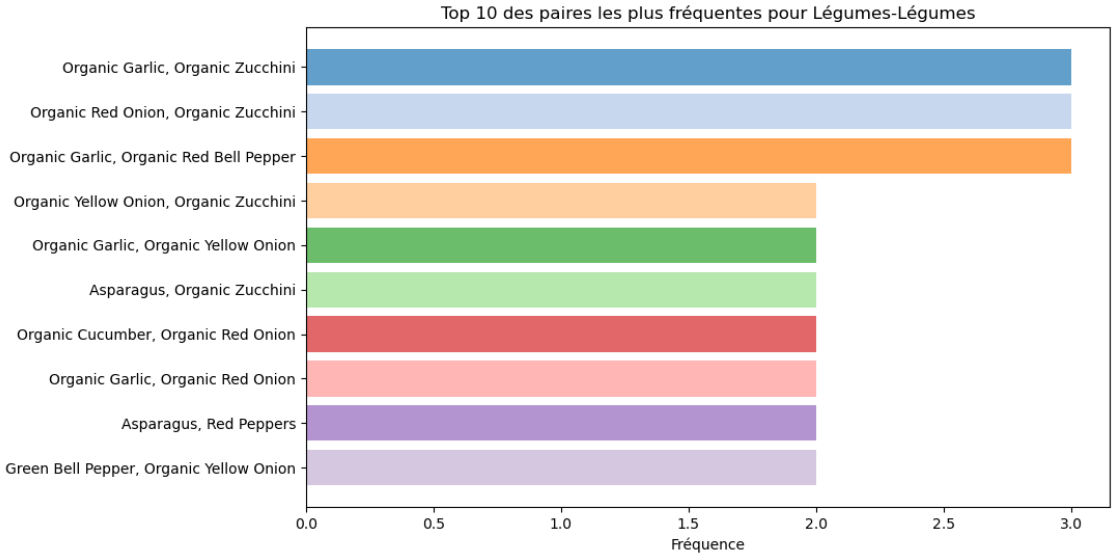
Ce code vise à visualiser les paires de produits fréquemment achetés ensemble dans différentes catégories : "Fruits-Fruits," "Légumes-Légumes," et "Fruits-Légumes." Voici comment le code génère des graphiques à barres pour chaque catégorie :

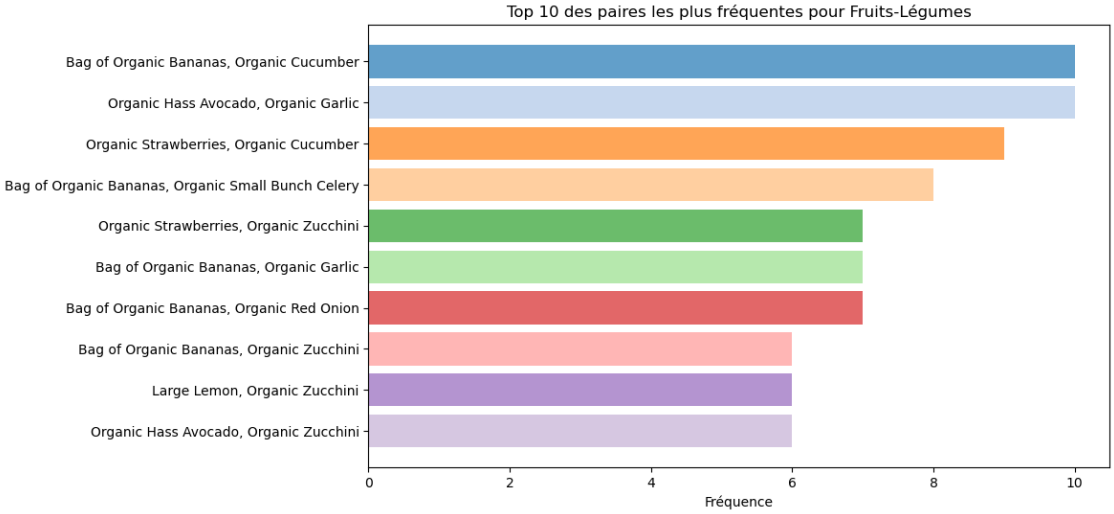
1. **Liste des Catégories :** Le code commence par définir une liste de catégories, à savoir "Fruits-Fruits," "Légumes-Légumes," et "Fruits-Légumes."
2. **Listes de Paires et Fréquences :** Les paires de produits fréquemment achetées dans chaque catégorie ("Fruits-Fruits," "Légumes-Légumes," "Fruits-Légumes") et leurs fréquences associées sont stockées dans des listes (`pair_lists` et `frequency_lists`) respectivement.
3. **Création d'une Colormap :** Une colormap est créée à l'aide de la bibliothèque Matplotlib pour garantir qu'il y a suffisamment de couleurs pour couvrir toutes les barres dans les graphiques. La colormap utilisée ici est `'tab20'`.
4. **Création des Graphiques à Barres :** Le code génère un graphique à barres pour chaque catégorie en utilisant les données de paires de produits et de fréquences. Chaque graphique est configuré avec les éléments suivants :
   - Les paires de produits sont affichées sur l'axe des ordonnées.
   - Les fréquences sont représentées par les barres horizontales.
   - Les étiquettes d'axe sont générées en combinant les noms des produits de chaque paire.
   - Les valeurs de fréquence sont affichées à droite de chaque barre.
5. **Affichage des Graphiques :** Chaque graphique en barres est affiché individuellement pour chaque catégorie. Les graphiques fournissent un aperçu visuel des paires de produits les plus fréquemment achetées ensemble dans chaque catégorie, ce qui peut être utile pour des recommandations de produits ou d'autres analyses en fonction des préférences des clients.

[ ]:
```
# Liste des catégories
categories = ["Fruits-Fruits", "Légumes-Légumes", "Fruits-Légumes"]

# Listes de paires et de fréquences pour chaque catégorie
pair_lists = [top_fruit_fruit_pairs, top_vegetable_vegetable_pairs,␣
 ↪top_fruit_vegetable_pairs]
frequency_lists = [fruit_fruit_pair_counts, vegetable_vegetable_pair_counts,␣
 ↪fruit_vegetable_pair_counts]

# Génère une colormap avec suffisamment de couleurs pour couvrir toutes les␣
 ↪barres
```

```
cmap = plt.get_cmap('tab20')  # Vous pouvez changer 'tab20' en une autre␣
 ↪colormap si vous le souhaitez

for i, category in enumerate(categories):
    pairs = pair_lists[i]
    frequencies = [frequency_lists[i][pair] for pair in pairs]

    plt.figure(figsize=(10, 6))

    plt.barh(range(len(pairs)), frequencies, color=cmap(range(len(pairs)),␣
 ↪alpha=0.7), align='center')
    plt.yticks(range(len(pairs)), [', '.join(pair) for pair in pairs])
    plt.xlabel('Fréquence')

    plt.title(f"Top {len(pairs)} des paires les plus fréquentes pour␣
 ↪{category}")
    plt.gca().invert_yaxis()
    plt.show()
```







2.7.3 Analyse des Paires de Produits Fréquemment Achetés Ensemble

1. **Tendance Globale :**
   L'observation initiale révèle une disparité significative entre les fruits et les légumes, suggérant une préférence plus marquée des consommateurs américains pour les fruits dans leur alimentation quotidienne.

2. **Fréquence des Paires de Fruits :**
   La forte fréquence des paires de fruits, en particulier la combinaison de bananes et de fraises, confirme le penchant pour des options nutritives et sucrées. Cette tendance peut être attribuée à la popularité des smoothies, où ces fruits sont couramment utilisés ensemble.

3. **Combinaison de Légumes - Ail et Oignons :**
   Les légumes fréquemment achetés ensemble, tels que l'ail et les oignons, mettent en lumière des ingrédients polyvalents souvent utilisés pour rehausser la saveur des plats. Cette observation est cohérente avec la pratique culinaire consistant à incorporer ces légumes aromatiques dans diverses préparations.

4. **Association Fraises et Concombres - Tendance Saisonnière :**
   La corrélation entre l'achat de fraises et de concombres peut s'expliquer par une tendance saisonnière. En été, lorsque les températures sont élevées, les consommateurs recherchent des options rafraîchissantes. Ces deux aliments, frais et hydratants, sont logiques dans ce contexte, suggérant une préférence pour des produits adaptés à la saison estivale.

En synthèse, cette analyse révèle des tendances distinctes dans les habitudes d'achat, avec une préférence marquée pour les fruits. Les résultats confirment également des associations logiques entre certains produits, offrant des informations utiles pour comprendre les choix alimentaires des consommateurs et orienter les recommandations de produits.

2.8 Question 1.7

2.8.1 Paramètres de SLIM

- **Pruning (`pruning=True`):**
  - **Description :** L'élagage est activé.
  - **Rôle :** Permet de retirer les éléments moins importants du modèle, et donc gagner en temps et en compression.
- **Max Time (`max_time=150`):**
  - **Description :** Limite de temps d'exécution fixée à 150 unités.
  - **Rôle :** Contrôle le temps total d'entraînement de l'algorithme, ne fonctionne pas très bien dans notre cas.
- **Singletons (`singletons=True`):**
  - **Description :** Inclut/exclut les singletons dans la transformation des données.
  - **Rôle :** Les singletons sont des éléments apparaissant seuls dans les transactions. Si activé (`True`), les singletons seront exclus de la représentation condensée pour un gain de temps et de place.
- **Lexicographic Order (`lexicographic_order=True`):**
  - **Description :** Ordonne la représentation condensée de manière lexicographique.
  - **Rôle :** Facilite l'interprétation et l'analyse des résultats en organisant la représentation de manière alphabétique ou lexicographique.

[ ]:
```
# Chargement des données du DataFrame depuis le fichier CSV
order_products_train_df = pd.read_csv("../datas/order_products__train.csv")

# Groupement des "product_id" par "order_id" et création d'une liste par␣
 ↪"order_id"
slim_product_train_dataset = order_products_train_df.
 ↪groupby('order_id')['product_id'].apply(list).reset_index()

# Conversion du DataFrame en une liste de listes
slim_product_train_list = slim_product_train_dataset['product_id'].tolist()
```

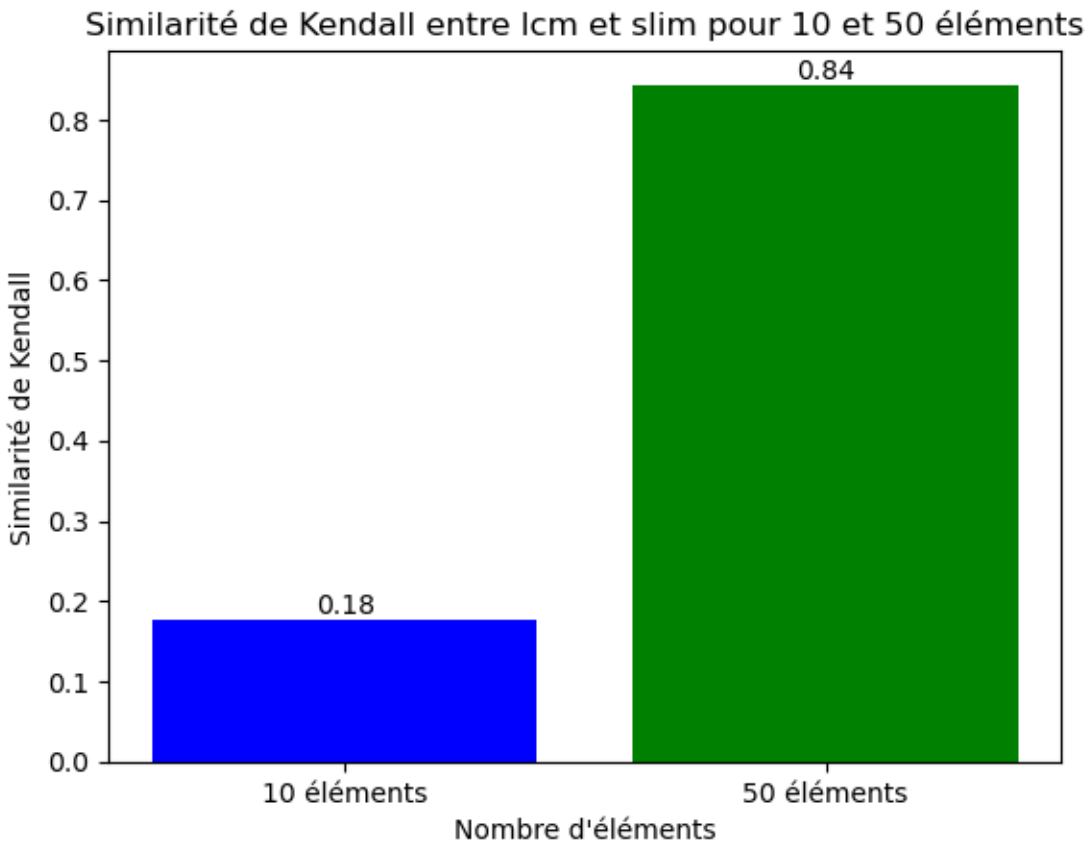
```
# Sélection aléatoire de 500 éléments dans la liste pour optimiser le temps de␣
 ↪calcul
slim_product_train_list = random.sample(slim_product_train_list, 500)
# Le compromis entre temps de calcul et précision des résultats est déterminé␣
 ↪ici.


# Création d'une instance de SLIM avec la suppression des singletons et un␣
 ↪temps d'exécution limité.
slim = SLIM(pruning=True, max_time=150)

# Entraînement de SLIM sur les données
slim.fit(slim_product_train_list)

# Transformation des données pour obtenir une représentation condensée en␣
 ↪excluant les singletons
condensed_representation = slim.transform(slim_product_train_list,␣
 ↪singletons=True, lexicographic_order=True)

# Récupération des itemsets
slim_itemsets = condensed_representation.iloc[:, 0]
slim_itemsets = [tuple(x) if isinstance(x, (list, tuple)) else (x,) for x in␣
 ↪slim_itemsets]

# Récupération des supports
slim_supports = condensed_representation["usage"]

# Création d'un dictionnaire avec les itemsets et les supports
slim_itemset_support_dict = dict(zip(slim_itemsets, slim_supports))

#----------------------LCM-----------------------------------------------------␣
 ↪

# Groupement des "product_id" par "order_id" et création d'une liste par␣
 ↪"order_id"
lcm_product_train_dataset = order_products_train_df.
 ↪groupby('order_id')['product_id'].apply(list).reset_index()
lcm_product_train_list = lcm_product_train_dataset['product_id'].tolist()
lcm_product_train_list = random.sample(lcm_product_train_list, 500)

# Utilisation de l'algorithme LCM pour extraire les itemsets fréquents
lcm_result_500 = LCM(min_supp = 5).fit_transform(lcm_product_train_list,␣
 ↪lexicographic_order=True)

# Récupération des itemsets
```

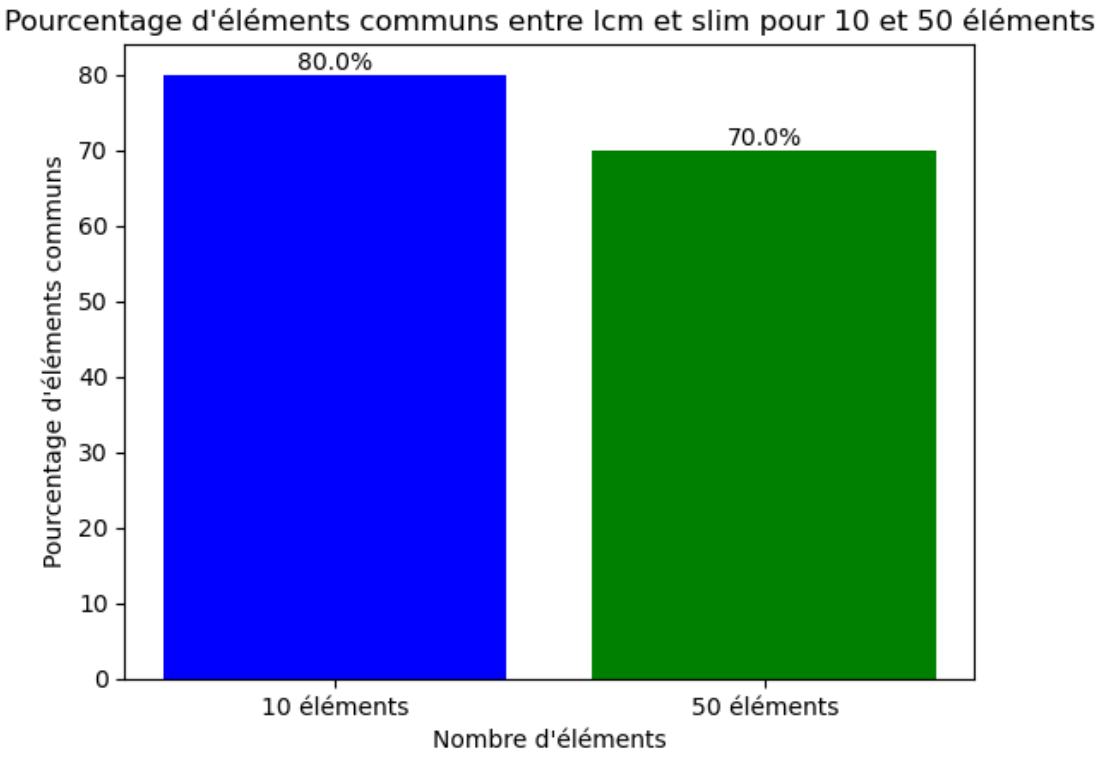
```
itemsets_500 = lcm_result_500.iloc[:, 0]
itemsets_500 = [tuple(x) if isinstance(x, (list, tuple)) else (x,) for x in␣
 ↪itemsets_500]

# Récupération des supports
supports_500 = lcm_result_500["support"]

# Création d'un dictionnaire avec les itemsets et les supports
lcm_itemset_support_dict = dict(zip(itemsets_500, supports_500))

#-------------------FONCTIONS-------------------------------------

def count_common_tuples(list1, list2):
    """
    Compte le nombre d'itemsets en commun entre deux listes.

    Args:
        list1 (list): Première liste d'itemsets.
        list2 (list): Deuxième liste d'itemsets.

    Returns:
        int: Nombre d'itemsets en commun.
    """
    # Convertion des listes de tuples en ensembles pour une recherche plus␣
 ↪efficace
    set1 = set(list1)
    set2 = set(list2)

    # Utilisation de l'opérateur d'intersection pour trouver les éléments␣
 ↪communs
    common_tuples = set1.intersection(set2)

    # Retourne la taille de l'ensemble d'éléments communs
    return len(common_tuples)


def calculate_similarity_with_kendall(reference, column):
    """
    Calcule la similarité de Kendall entre deux ensembles d'indices.

    Args:
        reference (list): Ensemble de référence.
        column (list): Ensemble à comparer avec la référence.

    Returns:
        float: Similarité de Kendall.
    """
```

```
    # Trouve la taille minimale des deux listes
    min_length = min(len(reference), len(column))

    # Tronque les listes pour qu'elles aient la même taille
    reference = reference[:min_length]
    column = column[:min_length]

    tau, _ = kendalltau(reference, column)
    similarity = 1 - tau  # Plus la distance de Kendall est faible, plus la␣
 ↪similarité est grande
    return similarity

def flatten_list_of_tuples(lst):
    """
    Aplatit une liste de tuples.

    Args:
        lst (list): Liste de tuples.

    Returns:
        list: Liste aplatie.
    """
    # Applatissement de la liste de tuples
    return [item for sublist in lst for item in sublist]

#-------------------------------------------------------------------------------------------

#-------------------------------------Comparaison des␣
 ↪résultats-----------------------------------------------------


# Trie le dictionnaire par rapport au support (itemgetter(1) fait référence à␣
 ↪la valeur du support)
sorted_dict_slim = dict(sorted(slim_itemset_support_dict.items(),␣
 ↪key=itemgetter(1), reverse=True))
sorted_dict_lcm = dict(sorted(lcm_itemset_support_dict.items(),␣
 ↪key=itemgetter(1), reverse=True))

# Sélectionne les 50 premiers éléments du dictionnaire trié
top_50_dict_slim = dict(list(sorted_dict_slim.items())[:50])

top_50_dict_slim_column = list(top_50_dict_slim.keys())
top_50_dict_lcm = dict(list(sorted_dict_lcm.items())[:50])

top_50_dict_lcm_column = list(top_50_dict_lcm.keys())

# Calcule le nombre de tuples en commun avec 50 élements
```

```
nbCommonItem_50 = count_common_tuples(top_50_dict_slim_column,␣
 ↪top_50_dict_lcm_column)

# Utilise la fonction flatten_list_of_tuples pour aplatir les listes
flattened_slim_column_50 = flatten_list_of_tuples(top_50_dict_slim_column)
flattened_lcm_column_50 = flatten_list_of_tuples(top_50_dict_lcm_column)

# Calcule la similarité de Kendall avec les listes aplaties 50 élements
similarity_50 = calculate_similarity_with_kendall(flattened_slim_column_50,␣
 ↪flattened_lcm_column_50)


# Sélectionne les 10 premiers éléments du dictionnaire trié
top_10_dict_slim = dict(list(sorted_dict_slim.items())[:10])
top_10_dict_slim_column = list(top_10_dict_slim.keys())

top_10_dict_lcm = dict(list(sorted_dict_lcm.items())[:10])
top_10_dict_lcm_column = list(top_10_dict_lcm.keys())

# Calcule le nombre de tuples en commun avec 10 élements
nbCommonItem_10 = count_common_tuples(top_10_dict_slim_column,␣
 ↪top_10_dict_lcm_column)

# Utilise la fonction flatten_list_of_tuples pour aplatir les listes
flattened_slim_column_10 = flatten_list_of_tuples(top_10_dict_slim_column)
flattened_lcm_column_10 = flatten_list_of_tuples(top_10_dict_lcm_column)

# Calcule la similarité de Kendall avec les listes aplaties : 10 élements
similarity_10 = calculate_similarity_with_kendall(flattened_slim_column_10,␣
 ↪flattened_lcm_column_10)


# Couleurs pour les barres
colors = ['blue', 'green']

# Similarité de Kendall entre lcm et slim pour 10 et 50 éléments
similarities = [similarity_10, similarity_50]

# Noms des catégories
categories = ['10 éléments', '50 éléments']

# Crée un graphique en barres avec des couleurs
plt.bar(categories, similarities, color=colors)

# Étiquettes des axes et titre
plt.xlabel('Nombre d\'éléments')
plt.ylabel('Similarité de Kendall')
```

```
plt.title('Similarité de Kendall entre lcm et slim pour 10 et 50 éléments')

# Ajoute les valeurs au-dessus des barres
for i, similarity in enumerate(similarities):
    plt.text(i, similarity, round(similarity, 2), ha='center', va='bottom')

# Affiche le graphique
plt.show()

# Nombre d'éléments communs entre LCM et SLIM en pourcentages
common_items_percentage = [nbCommonItem_10 / 10 * 100, nbCommonItem_50 / 50 *␣
 ↪100]

# Crée un graphique en barres avec des couleurs
plt.bar(categories, common_items_percentage, color=colors)

# Étiquettes des axes et titre
plt.xlabel('Nombre d\'éléments')
plt.ylabel('Pourcentage d\'éléments communs')
plt.title('Pourcentage d\'éléments communs entre lcm et slim pour 10 et 50␣
 ↪éléments')

# Ajoute les valeurs au-dessus des barres
for i, percentage in enumerate(common_items_percentage):
    plt.text(i, percentage, f"{round(percentage, 2)}%", ha='center',␣
 ↪va='bottom')

# Affiche le graphique
plt.show()
```


Similarité de Kendall entre lcm et slim pour 10 et 50 éléments
0.84
0.18
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Similarité de Kendall
10 éléments
50 éléments
Nombre d'éléments



2.8.2 Comparaison des Itemsets Fréquents entre SLIM et LCM

Les graphiques de comparaison entre les itemsets fréquents générés par SLIM et LCM révèlent des fortes similitudes.

1. **Corrélation selon l'algorithme de Kendall :**
   - L'utilisation de l'algorithme de Kendall pour évaluer la corrélation entre les itemsets fréquents de SLIM et LCM ne produit pas un très bon coefficient pour 10 éléments, cela est probablement dû à la peytite taille de l'échantillon. Pour 50 élément le coefficient est de 0.84, ce qui indique une forte similarité dans les ordres de classement des éléments.
2. **Comparaison des 10 Itemsets les Plus Fréquents :**
   - Le nombre d'élements commun atteint près de 80% pour les 10 itemsets les plus fréquents, soulignant une concordance significative dans les associations identifiées par les deux algorithmes.
   - Il est juste à noté que l'ordre de ces éléments n'est pas exacte, cela est dû à la taille de l'échantillon.
3. **Comparaison des 50 Itemsets les Plus Fréquents :**
   - Même avec une extension à 50 itemsets, la corrélation reste élevée à 70%, attestant de la robustesse de la similarité entre les résultats de SLIM et LCM.
4. **Considérations sur la Taille de l'Échantillon :**
   - Il est crucial de noter que les tests ont été effectués sur un échantillon de 500 valeurs en raison de la lenteur de l'algorithme SLIM. Cette restriction peut influencer la représentativité des itemsets fréquents, notamment dans les 10 éléments, où la corrélation

pourrait être sous-estimée.

En conclusion, bien que la comparaison soit basée sur des échantillons, la forte corrélation entre les itemsets fréquents générés par SLIM et LCM suggère une fiabilité robuste des résultats obtenus par les deux approches.

3 TASK 2 : Sequential patterns

3.0.1 Transformation de Données pour l'Algorithme PrefixSpan / CloSpan

Le code vise à préparer les données pour l'algorithme PrefixSpan / CloSpan en effectuant plusieurs étapes de transformation.

3.0.2 Chargement des Données

1. **Chargement du Fichier de Séquence :**
   - Le fichier `transactions_seq.txt` est chargé en utilisant `read_csv`, en spécifiant l'encodage latin-1 et le séparateur de tabulation.

3.0.3 Création de la Correspondance Item-Numéro

2. **Création de la Correspondance :**
   - Les éléments de la quatrième colonne (index 3) contenant des éléments séparés par des virgules sont divisés en listes.
   - Un dictionnaire `item_to_number` est créé pour mapper chaque item à un numéro unique.
   - Une correspondance est écrite dans un fichier texte `correspondance_items.txt`.
3. **Remplacement des Noms par des Numéros :**
   - La quatrième colonne est mise à jour pour remplacer les noms d'items par leurs numéros respectifs.

3.0.4 Analyse des Occurrences

4. **Analyse des Occurrences :**
   - Les occurrences de la première colonne sont comptées pour chaque item.
   - Un DataFrame `occurrences_df` est créé pour stocker les identifiants et le nombre d'apparitions.

3.0.5 Préparation des Données pour PrefixSpan

5. **Préparation des Données pour PrefixSpan :**
   - Les données sont organisées en séquences selon le format requis par PrefixSpan / CloSpan.
   - Les informations sont écrites dans un fichier `prefixSpanData.txt`.

3.0.6 Mise à Jour du Fichier de Correspondance

6. **Mise à Jour du Fichier de Correspondance :**
   - Le fichier de correspondance est lu et transformé pour être conforme au format requis par PrefixSpan / CloSpan.
   - Les nouvelles données sont écrites au début du fichier `prefixSpanData.txt`.

3.0.7 Résultats

7. **Résultats de Transformation :**
   - Un message indique que les données ont été transformées avec succès et écrites dans `prefixSpanData.txt`.

Ces étapes permettent de préparer les données pour une utilisation efficace avec l'algorithme PrefixSpan / CloSpan, en transformant les noms d'items en numéros et en organisant les séquences.

Après toutes les transformations mon jeu de données ressemble donc au format attendu ci-dessous où -1 représente la séparation entre 2 séquences d'achat pour un client, et -2 la fin de toutes les séquences d'achat d'un client :

@CONVERTED_FROM_TEXT
@ITEM=1=apple
@ITEM=2=orange
@ITEM=3=tomato
@ITEM=4=milk
@ITEM=5=bread
@ITEM=6=noodle
@ITEM=7=rice
@ITEM=-1=|
1 -1 1 2 3 -1 1 3 -1 4 -1 3 6 -1 -2
1 4 -1 3 -1 2 3 -1 1 5 -1 -2
5 6 -1 1 2 -1 4 6 -1 3 -1 2 -1 -2
5 -1 7 -1 1 6 -1 3 -1 2 -1 3 -1 -2

[ ]:
```
# Spécifie l'encodage comme étant latin-1
encoding = 'latin1'

# Spécifie le séparateur comme étant une tabulation
separator = '\t'

# Utilise `read_csv` en spécifiant l'encodage et sans spécifier de noms de␣
 ↪colonnes
sequence_data = pd.read_csv('../transactions_seq.txt', sep=separator,␣
 ↪header=None, encoding=encoding)

# Si la 4ème colonne contient des éléments séparés par des virgules, permet de␣
 ↪les diviser en listes
sequence_data.iloc[:, 3] = sequence_data.iloc[:, 3].str.split(',')

sequence_save = sequence_data

third_column = sequence_data.iloc[:, 3]  # La quatrième colonne (l'index 2)

# Initialisation du dictionnaire de correspondance
item_to_number = {}
number = 0
```

```
# Initialisation d'un ensemble pour suivre les noms déjà traités
processed_items = set()

# Nom du fichier de correspondance
file_name = "correspondance_items.txt"

# Vérifie si le fichier existe et le supprime dans le cas échéant
if os.path.exists(file_name):
    os.remove(file_name)

# Ouvre un fichier texte pour écrire la correspondance
with open(file_name, "w", encoding="utf-8") as file:
    for row in third_column:
        for item in row:
            item = item.strip()  # Supprime les espaces en surplus
            if item not in processed_items:
                processed_items.add(item)
                if item not in item_to_number:
                    item_to_number[item] = number
                    number += 1
                file.write(f"{item}: {item_to_number[item]}\n")


# Crée une nouvelle colonne pour stocker les numéros
sequence_data[3] = sequence_data[3].apply(lambda x: [item_to_number[item.
 ↪strip()] for item in x])

first_column = sequence_data.iloc[:, 0]

# Crée un dictionnaire pour stocker les occurrences dans l'ordre d'apparition
occurrences_dict = {}

for item in first_column:
    if item not in occurrences_dict:
        occurrences_dict[item] = 1
    else:
        occurrences_dict[item] += 1

# Crée un DataFrame à partir du dictionnaire

#Occurences DF contient une colonne qui correspond à l'identifiant et une␣
 ↪colonne à son nombre d'apparition. Ainsi cela me permet de connaitre le␣
 ↪nombre d'apparition d'un individu.
occurrences_df = pd.DataFrame({'Item': list(occurrences_dict.keys()), 'Count':␣
 ↪list(occurrences_dict.values())})
```

```
[ ]: nom_fichier = "prefixSpanData.txt"

if os.path.exists(nom_fichier):
    os.remove(nom_fichier)

# Ouvre le fichier texte en mode écriture
with open("prefixSpanData.txt", "w", encoding="utf-8") as file:
    cpt = 0
    # Parcours les lignes de occurrences_df
    for _, row in occurrences_df.iterrows():
        count = row['Count']  # Nombre d'occurrences

        # Supposons que 'sequence_data' est votre DataFrame
        sub_series = sequence_data.iloc[cpt:cpt+count, 3]
        # Parcours les éléments de la série
        for index, value in sub_series.items():
            # 'value' contient la valeur de la colonne 3 pour chaque ligne␣
 ↪entre les indices 8 et 15 inclus
            for element in value:
                file.write(str(element) + ' ')
            file.write("-1 ")
        file.write("-2\n")

        cpt += count

##########################################################################################

# Ouvre le fichier de correspondance des items en mode lecture
with open(nom_fichier, 'r', encoding="utf-8") as fichier:
        contenu_original = fichier.read()

# Ouvre le fichier en mode écriture
with open(nom_fichier, 'w', encoding="utf-8") as fichier_ecriture:
    # Écrit le nouveau code au début du fichier
    fichier_ecriture.write("@CONVERTED_FROM_TEXT\n")

    # Nom du fichier de correspondance des items
    nom_fichier_correspondance = "correspondance_items.txt"

    # Ouvre le fichier de correspondance en mode lecture
    with open(nom_fichier_correspondance, 'r', encoding="utf-8") as␣
 ↪fichier_correspondance:
        lignes_correspondance = fichier_correspondance.readlines()

    # Transforme les données en format requis et les stockes dans une liste
    nouvelles_lignes = []
    for ligne in lignes_correspondance:
```

```
        item, code = ligne.strip().split(": ")
        nouvelle_ligne = f'@ITEM={code}={item}'
        nouvelles_lignes.append(nouvelle_ligne)

    # Écrit les données transformées dans le fichier
    fichier_ecriture.write('\n'.join(nouvelles_lignes))

    # Écrit le contenu d'origine après les données transformées
    fichier_ecriture.write('\n')
    fichier_ecriture.write(contenu_original)

print("Données transformées et écrites avec succès dans prefixSpanData.txt.")
```

```
Données transformées et écrites avec succès dans prefixSpanData.txt.
```

3.1 Question 2.1

3.1.1 Exécution et Analyse des Résultats de l'Algorithme CloSpan

L'algorithme CloSpan a été exécuté pour extraire des itemsets fréquents, et les résultats ont été analysés pour comprendre les associations entre les éléments.

1. **Exécution de CloSpan :**
   - La commande pour exécuter CloSpan a été lancée avec succès en utilisant la bibliothèque SPMF (Sequential Pattern Mining Framework). Le code de retour 0 confirme une exécution sans erreur.
2. **Analyse des Itemsets Fréquents :**
   - Les itemsets fréquents ont été extraits du fichier de sortie généré par CloSpan. Chaque itemset est associé à son support.
3. **Sélection des Meilleurs Itemsets :**
   - Les itemsets ont été triés en fonction de leur support, et les 20 meilleurs ont été sélectionnés pour une analyse plus approfondie.
4. **Affichage des Itemsets Sélectionnés :**
   - Les 20 itemsets sélectionnés sont présentés de manière lisible, montrant les éléments inclus dans chaque itemset ainsi que leur support respectif.
5. **Visualisation Graphique :**
   - Un graphique à barres horizontal a été généré pour représenter le support des itemsets, limité aux itemsets de longueur 3 à 4. Chaque barre représente un itemset.
6. **Observations et Interprétations :**
   - Les itemsets et leurs supports offrent un aperçu des associations fréquentes entre différents éléments. L'analyse des itemsets peut fournir des informations sur les tendances et les relations dans les données sous-jacentes.

[ ]:
```
# Exécute l'algorithme CloSpan à l'aide de SPMF
commande = "java -jar spmf.jar run CloSpan prefixSpanData.txt cloSpanOutput.txt␣
 ↪1% true"
code_retour = os.system(commande)

# Vérifie si la commande s'est exécutée avec succès
```

```
if code_retour == 0:
    print("La commande s'est exécutée avec succès.")
else:
    print(f"La commande a rencontré une erreur avec le code de retour␣
 ↪{code_retour}.")
```

```
La commande s'est exécutée avec succès.
```

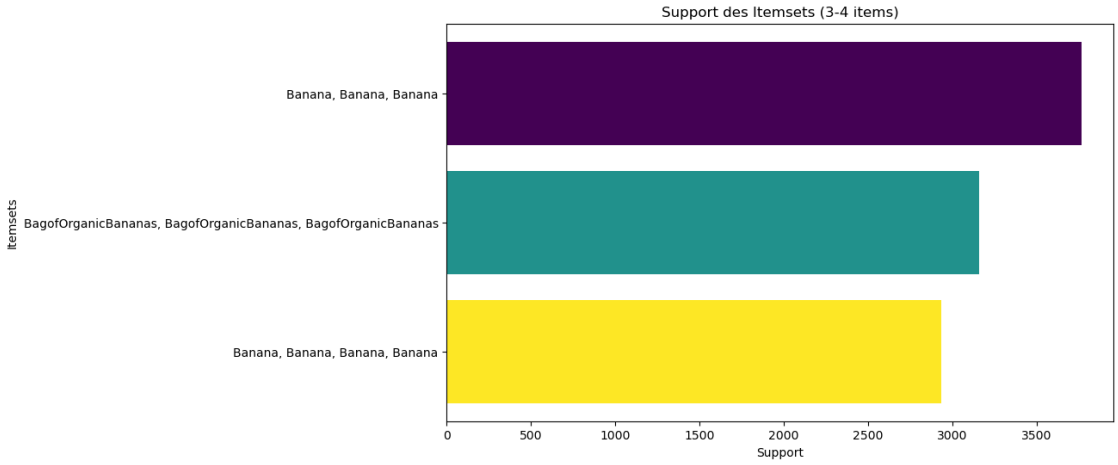
[ ]:
```
# Fonction pour extraire les top n-itemsets du fichier de sortie de CloSpan
def top_n_itemsets(file_name, n):
    """
    Extrait les top n-itemsets du fichier de sortie de CloSpan.

    Args:
        file_name (str): Nom du fichier de sortie de CloSpan.
        n (int): Nombre d'itemsets à extraire.

    Returns:
        list: Liste des n-itemsets avec leur support.
    """
    # Crée une liste pour stocker les listes d'items
    itemsets_list = []

    # Trie les n-itemsets
    trie_n_itemsets(file_name, n)

    # Ouvre le fichier original en mode lecture
    with open("cloSpanBestSupport.txt", 'r') as fichier:
        for ligne in fichier:
            # Utilise une expression régulière pour trouver tous les mots qui␣
 ↪commencent par une lettre
            mots = re.findall(r'\b(?!SUP|SID)[a-zA-Z]\w*\b', ligne)
            numeros_support = re.findall(r'#SUP: (\d+)', ligne)

            # Crée une liste pour stocker les items de chaque ligne
            itemset = []

            # Parcours tous les mots trouvés
            for mot in mots:
                itemset.append(mot)

            # Ajoute la liste d'items et le support correspondant à la liste␣
 ↪principale
            itemsets_list.append((itemset, int(numeros_support[0])))

    return itemsets_list
```

```
# Fonction pour tracer les itemsets sélectionnés en fonction de leur longueur
def plot_selected_itemsets(itemsets_list, min_length, max_length):
    """
    Trace les itemsets sélectionnés en fonction de leur longueur.

    Args:
        itemsets_list (list): Liste des itemsets avec leur support.
        min_length (int): Longueur minimale des itemsets à tracer.
        max_length (int): Longueur maximale des itemsets à tracer.
    """
    selected_itemsets = [(itemset, support) for itemset, support in␣
 ↪itemsets_list if min_length <= len(itemset) <= max_length]

    itemsets = [", ".join(itemset) for itemset, _ in selected_itemsets]
    support = [support for _, support in selected_itemsets]

    # Crée des couleurs uniques pour chaque barre
    colors = np.arange(len(itemsets))

    plt.figure(figsize=(10, 6))
    plt.barh(itemsets, support, color=plt.cm.viridis(colors / max(colors)))
    plt.xlabel("Support")
    plt.ylabel("Itemsets")
    plt.title(f"Support des Itemsets ({min_length}-{max_length} items)")
    plt.gca().invert_yaxis()
    plt.show()

#Fonction pour trier les n-itemsets en fonction de leurs support
def trie_n_itemsets(file_name, n):
    """
    Trie les n-itemsets en fonction de leurs supports et enregistre les n␣
 ↪premiers dans un nouveau fichier.

    Args:
        file_name (str): Nom du fichier de sortie de CloSpan.
        n (int): Nombre d'itemsets à extraire.
    """
    # Crée un dictionnaire pour stocker les lignes avec leurs numéros de support
    support_lines = {}

    # Ouvre le fichier en mode lecture
    with open(file_name, 'r') as fichier:
        for ligne in fichier:
            # Utilise une expression régulière pour trouver tous les numéros de␣
 ↪support dans la ligne
            numeros_support = re.findall(r'#SUP: (\d+)', ligne)
            if numeros_support:
```

```
                support = int(numeros_support[0])
                # Ajoute la ligne au dictionnaire avec son numéro de support␣
 ↪comme clé
                support_lines[support] = ligne

    # Trie les numéros de support en ordre décroissant et prends les n premiers
    top_supports = sorted(support_lines.keys(), reverse=True)[:n]

    # Ouvre un nouveau fichier en mode écriture
    with open("cloSpanBestSupport.txt", 'w') as output_file:
        # Écrit les n lignes ayant les numéros de support les plus élevés dans␣
 ↪le nouveau fichier
        for support in top_supports:
            output_file.write(support_lines[support])

#Fonction pour imprimer une liste d'une manière lisible
def print_beautiful_list(input_list):
    for item in input_list:
        item_str = ' + '.join(item[0])
        print(f"({item_str}, {item[1]})")


# Utilisation desfonctions pour extraire et afficher et les top 20 itemsets
liste_trie_itemsets = top_n_itemsets('cloSpanOutput.txt', 20)
print_beautiful_list(liste_trie_itemsets)

# Utilisation de la fonction pour tracer les itemsets sélectionnés en fonction␣
 ↪de leur longueur
plot_selected_itemsets(liste_trie_itemsets, 3, 4)
```

```
(Banana, 7088)
(BagofOrganicBananas, 6084)
(OrganicStrawberries, 5672)
(OrganicBabySpinach, 5420)
(Banana + Banana, 4998)
(LargeLemon, 4455)
(Limes, 4432)
(OrganicHassAvocado, 4223)
(BagofOrganicBananas + BagofOrganicBananas, 4209)
(OrganicAvocado, 4176)
(Strawberries, 4105)
(Banana + Banana + Banana, 3768)
(OrganicBlueberries, 3623)
(OrganicGarlic, 3376)
(OrganicYellowOnion, 3341)
(OrganicZucchini, 3226)
(BagofOrganicBananas + BagofOrganicBananas + BagofOrganicBananas, 3156)
```

```
(OrganicRaspberries, 3020)
(Banana + Banana + Banana + Banana, 2932)
(CucumberKirby, 2914)
```



3.1.2 Interprétation

Je remarque dans mes résultats que, parmi les itemsets fréquents, ceux composés de 3 ou 4 éléments sont systématiquement des bananes. Par conséquent, je souhaite approfondir l'analyse de la fréquence d'achat des bananes.

3.2 Question 2.2

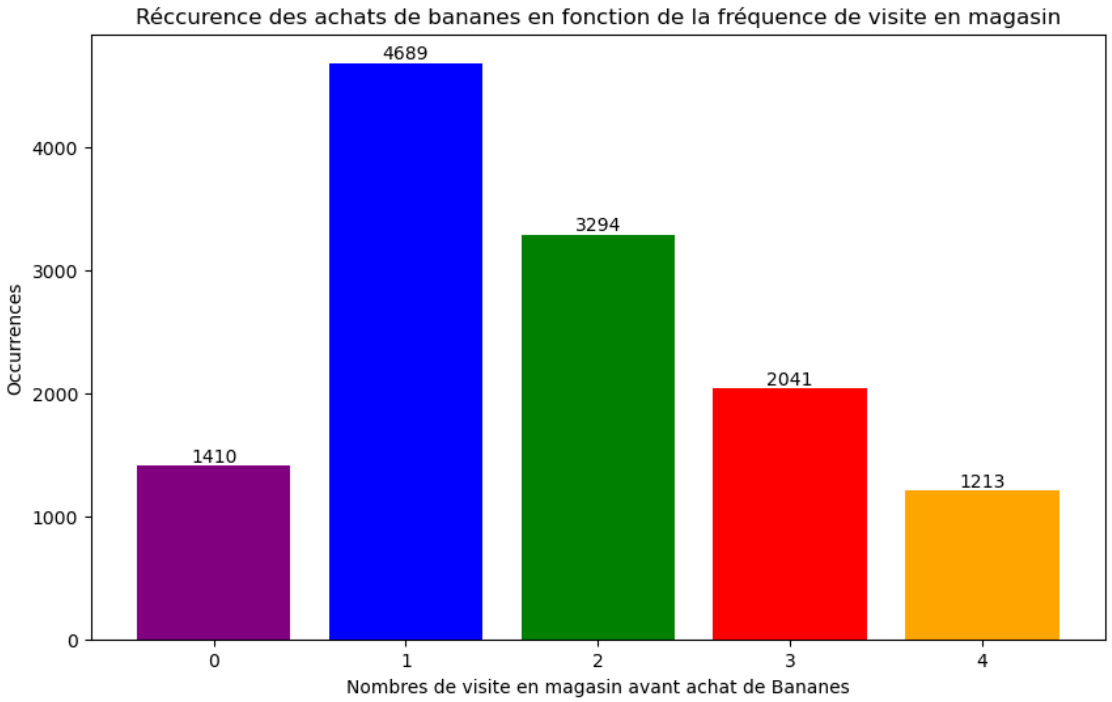
[ ]:
```python
def find_number_by_name_regex(file_path, name):
    """
    Trouve le numéro associé à un nom spécifique dans un fichier.

    Args:
        file_path (str): Chemin vers le fichier.
        name (str): Nom à rechercher.

    Returns:
        int: Numéro de ligne ou None si le nom n'est pas trouvé.
    """
    with open(file_path, 'r', encoding="utf-8") as file:
        for line_number, line in enumerate(file, 1):
            match = re.search(fr'@ITEM=(\d+)={re.escape(name)}', line)
            if match:
                return int(match.group(1))
    return None

def lire_lignes_avec_numero(nom_fichier, numero):
```

```
    """
    Lit les lignes d'un fichier qui contiennent un numéro spécifique.

    Args:
        nom_fichier (str): Nom du fichier.
        numero (str): Numéro à rechercher.

    Returns:
        list: Liste des lignes contenant le numéro.
    """
    lignes_avec_numero = []

    with open(nom_fichier, 'r') as fichier:
        lignes = fichier.readlines()
        for ligne in lignes:
            if re.search(rf" {numero} ", ligne):  # Recherche du numéro exact␣
 ↪entouré d'espaces
                lignes_avec_numero.append(ligne.strip())

    return lignes_avec_numero

def compter_minus_un_entre_occurrences(ligne, numero):
    """
    Compte le nombre d'occurrences de "-1" entre les segments d'une ligne.

    Args:
        ligne (str): Ligne à traiter.
        numero (str): Numéro à rechercher.

    Returns:
        list: Liste des compteurs pour chaque occurrence.
    """
    compteurs = []
    segments = ligne.split(f' {numero} ')

    for i in range(1, len(segments)):  # Commence à partir du deuxième segment
        compteur = segments[i - 1].count("-1")
        compteurs.append(compteur)

    compteurs = list(set(compteurs))
    return compteurs

def compter_minus_un_entre_occurrences_pour_lignes(file_path, name):
    """
    Compte le nombre d'occurrences de "-1" entre les segments pour toutes les␣
 ↪lignes associées à un nom spécifique.
```

```
    Args:
        file_path (str): Chemin vers le fichier.
        name (str): Nom à rechercher.
    """
    line_number = find_number_by_name_regex(file_path, name)
    if line_number is not None:
        lignes_avec_numero = lire_lignes_avec_numero(file_path,␣
↪str(line_number))
        compteurs = []

        for ligne in lignes_avec_numero:
            compteurs.extend(compter_minus_un_entre_occurrences(ligne,␣
↪str(line_number)))

        # Compte le nombre d'occurrences de chaque numéro
        compteurs_counts = {}
        for compteur in compteurs:
            if compteur in compteurs_counts:
                compteurs_counts[compteur] += 1
            else:
                compteurs_counts[compteur] = 1

        # Trie le dictionnaire par nombre d'occurrences
        compteurs_sorted = sorted(compteurs_counts.items(), key=lambda x: x[1],␣
↪reverse=True)

        # Extrait les valeurs triées et leurs occurrences
        compteurs, occurrences = zip(*compteurs_sorted)

        # Limite le nombre de barres à afficher
        max_bars = 5  # Affiche seulement les 5 premiers
        compteurs = compteurs[:max_bars]
        occurrences = occurrences[:max_bars]

        # Crée une liste de couleurs pour chaque barre
        colors = ['blue', 'green', 'red', 'purple', 'orange']

        # Crée le diagramme en barres
        plt.figure(figsize=(10, 6))
        plt.bar(compteurs, occurrences, color=colors)
        plt.xlabel("Nombres de visite en magasin avant achat de Bananes")
        plt.ylabel("Occurrences")
        plt.title("Réccurence des achats de bananes en fonction de la fréquence␣
↪de visite en magasin")

        # Affiche les valeurs au-dessus de chaque barre
        for i, occ in enumerate(occurrences):
```

```
            plt.text(compteurs[i], occ, str(occ), ha='center', va='bottom')

        plt.show()
    else:
        print(f"Le nom '{name}' n'a pas été trouvé dans le fichier␣
 ↪'{file_path}'.")

def afficher_toutes_valeurs_occurrences(file_path, name):
    """
    Affiche toutes les valeurs d'occurrences associées à un nom spécifique.

    Args:
        file_path (str): Chemin vers le fichier.
        name (str): Nom à rechercher.
    """
    line_number = find_number_by_name_regex(file_path, name)
    if line_number is not None:
        lignes_avec_numero = lire_lignes_avec_numero(file_path,␣
 ↪str(line_number))
        compteurs = []

        for ligne in lignes_avec_numero:
            compteurs.extend(compter_minus_un_entre_occurrences(ligne,␣
 ↪str(line_number)))

        # Compte le nombre d'occurrences de chaque compteur
        compteurs_counts = {}
        for compteur in compteurs:
            if compteur in compteurs_counts:
                compteurs_counts[compteur] += 1
            else:
                compteurs_counts[compteur] = 1

        # Trie le dictionnaire par nombre d'occurrences
        compteurs_sorted = sorted(compteurs_counts.items(), key=lambda x: x[1],␣
 ↪reverse=True)

        print("Compteurs et leurs occurrences :")
        for compteur, occurrences in compteurs_sorted:
            print(f"Compteur {compteur}: {occurrences} occurrence(s)")
    else:
        print(f"Le nom '{name}' n'a pas été trouvé dans le fichier␣
 ↪'{file_path}'.")

file_prefixSpan = "prefixSpanData.txt"
name = "Banana"
```

```
# Compte toutes la fréquence d'achat de l'item "Banana" et affiche les 5 barres␣
 ↪qui correspondent aux fréquences d'achat les plus fréquentes.
compteurs_counts =␣
 ↪compter_minus_un_entre_occurrences_pour_lignes(file_prefixSpan, name)
```



3.2.1 Interprétation

Ici, nous pouvons remarquer que chez les clients qui achètent des bananes, ils les achètent en majorité à chaque fois qu'il vont faire des courses ou toutes les 2 visites. Probablement car les bananes sont des fruits qui pourissent relativement rapidement, et donc il est compliqué d'en acheter pour une durée plus longue, si on suppose ici que les clients font leurs courses une fois par semaine.

4 TASK 3 (optional) : Discriminative patterns

4.1 Question 3.1

Mon objectfi ici va être de regarder si les personnes qui achètent de la nourriture pour bébé, sont en général les mêmes personnes qui ont des animaux. En effet, on pourrait penser que les familles installées sont normalement la population qui est la plus à même d'avoir des animaux.

[ ]:

```
# Groupement des "product_id" par "order_id" et création d'une liste par␣
 ↪"order_id"
product_train_dataset_q3 = order_products_train_df.
 ↪groupby('order_id')['product_id'].apply(list).reset_index()
product_train_list_q3 = lcm_product_train_dataset['product_id'].tolist()
products = pd.read_csv('../datas/products.csv')
departments = pd.read_csv('../datas/departments.csv')

# Fusionner les dataframes sur la colonne 'department_id' pour avoir un maximum␣
 ↪d'informations sur les produits
products_departments = pd.merge(products, departments, on='department_id')

# Filtrer les produits du département "babies" et garder uniquement la liste␣
 ↪des noms des produits qui sont dans le département babies.
babies_products = products_departments[products_departments['department'] ==␣
 ↪'babies']['product_name'].tolist()

# Spécifie l'encodage comme étant latin-1
encoding = 'latin1'
# Spécifie le séparateur comme étant une tabulation
separator = '\t'
# Utilise `read_csv` en spécifiant l'encodage et sans spécifier de noms de␣
 ↪colonnes
sequence_data = pd.read_csv('../transactions_seq.txt', sep=separator,␣
 ↪header=None, encoding=encoding)

# Renommer la première colonne
sequence_data = sequence_data.rename(columns={0: 'Numero_colonne'})
# Utiliser groupby et agg pour agréger les données
aggregated_data = sequence_data.groupby('Numero_colonne')[3].agg(lambda x: ' '.
 ↪join(x.astype(str))).reset_index()
# Renommer la colonne agrégée
aggregated_data = aggregated_data.rename(columns={3: 'Contenu_concatene'})

# Créer une expression régulière en utilisant | (OU) pour tous les produits de␣
 ↪babies_products
babies_regex = '|'.join(map(re.escape, babies_products))

# Filtrer les lignes qui comprennent au moins un produit de babies_products
filtered_babies = aggregated_data[aggregated_data['Contenu_concatene'].str.
 ↪contains(babies_regex)]

# Filtrer les lignes qui ne comprennent aucun produit de babies_products
filtered_others = aggregated_data[~aggregated_data['Contenu_concatene'].str.
 ↪contains(babies_regex)]
```

```
# Afficher les résultats
print("Découpage de mon jeu de donnée en 2 datasets")
print("Tableau avec au moins un produit qui provient du rayon bébé:")
print(len(filtered_babies))

print("Tableau avec le reste des lignes:")
print(len(filtered_others))

#---------------------------------------Babies␣
 ↪People-------------------------------------------------------------------------------

# Filtrer les lignes qui comprennent au moins un produit de babies_products
filtered_pets_babies = filtered_babies[(filtered_babies['Contenu_concatene'].
 ↪str.contains("Cat|Dog"))]
nb_babies_people_total = len(filtered_babies)
nb_pets_for_babies_people = len(filtered_pets_babies)

print("Nombre de personnes qui achètent des produits dans le rayon bébé et des␣
 ↪produits pour chat ou chien : ", nb_pets_for_babies_people)

ratio_babies_pets = round((nb_pets_for_babies_people /␣
 ↪nb_babies_people_total),2)
print("ratio bébéTotal / bébé+Animaux : ", ratio_babies_pets)

#------------------␣
 ↪-------------------------------------------------------------------------------------------------
#---------------------------------------Other␣
 ↪People-------------------------------------------------------------------------

# Filtrer les lignes qui comprennent au moins un produit de babies_products
filtered_pets_others = filtered_others[(filtered_others['Contenu_concatene'].
 ↪str.contains("Cat|Dog"))]
nb_others_people_total = len(filtered_others)
nb_pets_for_others_people = len(filtered_pets_others)

print("Nombre de personnes qui n'achètent aucun produit dans le rayon bébé mais␣
 ↪achètent des produits pour chat ou chien :", nb_pets_for_others_people)

ratio_others_pets = round((nb_pets_for_others_people /␣
 ↪nb_others_people_total),2)

print("ratio autreTotal / autre+Animaux : ", ratio_others_pets)

def calculer_taux_croissance_discriminant(support_positif, support_negatif):
    try:
        taux_croissance_discriminant = support_positif / support_negatif
```

```
        return taux_croissance_discriminant
    except ZeroDivisionError:
        # En cas de division par zéro, retourner une valeur spéciale ou gérer␣
 ↪selon vos besoins.
        return float('nan')

print("Résultat du Growth Rate entre ces 2 partitions")
print(calculer_taux_croissance_discriminant(ratio_babies_pets,␣
 ↪ratio_others_pets))
```

```
Découpage de mon jeu de donnée en 2 datasets
Tableau avec au moins un produit qui provient du rayon bébé:
1926
Tableau avec le reste des lignes:
18074
Nombre de personnes qui achètent des produits dans le rayon bébé et des produits
pour chat ou chien :  607
ratio bébéTotal / bébé+Animaux :  0.32
Nombre de personnes qui n'achètent aucun produit dans le rayon bébé mais
achètent des produits pour chat ou chien : 3066
ratio autreTotal / autre+Animaux :  0.17
Résultat du Growth Rate entre ces 2 partitions
1.8823529411764706
Nombre de personnes qui n'achètent aucun produit dans le rayon bébé mais
achètent des produits pour chat ou chien : 3066
ratio autreTotal / autre+Animaux :  0.17
Résultat du Growth Rate entre ces 2 partitions
1.8823529411764706
```

4.1.1 Interprétation

Le grosse rate étant supérieur à 1, nous pouvons en conclure que notre hypothèse est respectée et que la proportion d'individus ayant des enfants qui ont aussi des animaux est plus importante que celle sans enfants.