RAPPORT TP FOUILLE DE DONNÉES

Thème: Amélioration de l'algorithme de Apriori

Réalisé par:

TAREB Selma 191931089198

ADLAOUI Anis 181831052692

Section: M1 IV

Année universitaire: 2023/2024

Introduction

La fouille de données est la technique qui consiste à examiner une grande structure de données pour trouver des modèles, des tendances, des informations cachées qui ne seraient pas possibles en utilisant des techniques plus simples basées sur des requêtes.

Il utilise des algorithmes mathématiques sophistiqués pour classer, diviser, segmenter l'ensemble des données, les prétraiter si nécessaire et évaluer la possibilité d'événements futurs.

parmi les algorithmes les plus connus de la fouille de données est l’algorithme a priori.
Notre projet se concentre sur l'amélioration de cet algorithme afin d'adresser ces limitations et d'optimiser son fonctionnement dans des environnements de données modernes. Notre objectif principal est d'explorer diverses techniques et stratégies pour renforcer les performances de l'algorithme, en particulier dans des scénarios où les ensembles de données sont volumineux et hétérogènes.

Algorithme Apriori

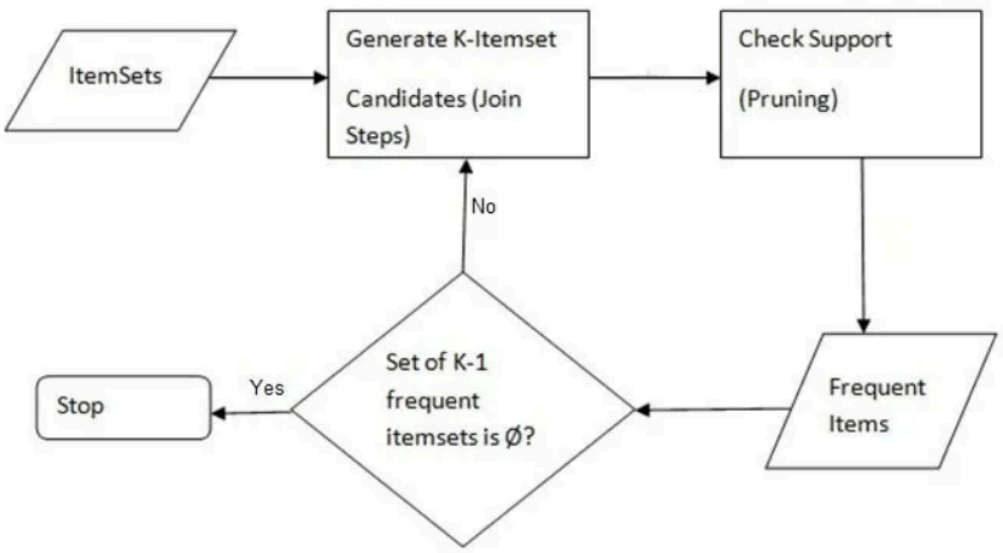
1- Définition

L'algorithme Apriori est axé sur la génération d'ensembles d'éléments fréquents, un sous-ensemble d'éléments qui apparaissent ensemble dans un ensemble de données avec une fréquence supérieure à un seuil défini par l'utilisateur. Cette approche est plus efficace qu'une méthode de force brute, qui examine tous les ensembles possibles, et elle réduit considérablement la complexité des calculs.

2- Comment fonctionne Apriori ?

- Génération de candidats :
  En commençant par les 1-itemsets (éléments individuels), l'algorithme génère des candidats itemsets de longueur croissante. Par exemple, après avoir établi les 1-itemsets fréquents, il construit des 2-itemsets candidats à partir de ceux-ci.

- Élagage basé sur le seuil de support :
  Avec un seuil de soutien défini (par exemple, 20 %), l'algorithme parcourt ensuite l'ensemble de données pour calculer le soutien de chaque ensemble candidat, en écartant ceux qui se situent en dessous du seuil. Ce processus se poursuit de manière itérative, en créant des ensembles plus importants à partir des ensembles fréquents identifiés à l'étape précédente.



Avantages et les limites de Apriori

1- Avantages :

- Facile à comprendre et à mettre en œuvre, adapté même aux débutants en exploration de données.
- Performant pour extraire des règles d'association sur des ensembles de données de taille modérée.
- Permet d'identifier rapidement les éléments fréquents dans une base de données transactionnelle.
- Peut être amélioré pour gérer de grands ensembles de données en utilisant des techniques comme le pruning et le parallélisme.
- Utilisé dans le marketing, la bioinformatique, la surveillance des réseaux, etc., pour découvrir des modèles utiles.
- Repose sur une base théorique robuste en théorie des ensembles et en statistiques

2- Limites :

- Sensibilité à la taille des données: L'algorithme peut être inefficace avec de grandes quantités de données, nécessitant beaucoup de mémoire et de temps de traitement.
- Il peut nécessiter plusieurs itérations sur les données pour extraire les ensembles d'articles fréquents, ce qui peut être coûteux en termes de temps de calcul.
- Il peut avoir du mal à trouver des règles significatives dans des ensembles de données où les éléments fréquents sont rares ou lorsque les données sont déséquilibrées.
- Il peut produire un grand nombre de règles d'association non pertinentes ou triviales, nécessitant un post-traitement pour filtrer les résultats.
- Il génère par défaut des règles d'une taille spécifique, ce qui peut limiter sa capacité à découvrir des modèles plus complexes dans les données.

Implémentation de l’algorithme apriori sans amélioration

Dataset 1 :

La dataset contient les achats des items individuels au sein d’une supérette
https://drive.google.com/file/d/1Pf6834FViN_A4rpnn386draLO-XwWjio/view?usp=sharing

Dataset 2 (plus grande) :

La dataset contient des achats de plusieur item regroupé dans une liste dans la colonnes “product”
https://drive.google.com/file/d/15w7wjz7qJPU3PiEBM_El6DNo53EjVzCG/view?usp=sharing

generate_candidate_itemsets
cette fonction génère les ensembles d'éléments candidate de la taille k+1 a partir des ensembles d'éléments fréquents de taille ‘k’, elle boucle sur les éléments si les éléments k-1 premiers éléments fréquent sont égaux alors il sont combiné

```python
def generate_candidate_itemsets(itemset, k):
    candidate_itemsets = []
    for i in range(len(itemset)):
        for j in range(i + 1, len(itemset)):
            if itemset[i][:-1] == itemset[j][:-1]:
                candidate = sorted(set(itemset[i]) | set(itemset[j]))
                if len(candidate) == k and '' not in candidate:
                    if not has_infrequent_subset(candidate, itemset):
                        candidate_itemsets.append(candidate)
                        #hash_table.insert(candidate)
    return candidate_itemsets
```

has_infrequent_itemset
vérifie si un ensemble d'éléments possède des sous-ensembles peu fréquent

```python
def has_infrequent_subset(itemset, prev_itemsets):
    subsets = itertools.combinations(itemset, len(itemset) - 1)
    for subset in subsets:
        if list(subset) not in prev_itemsets:
            return True
    return False
```

prune_itemsets
utilisé pour filtrer les ensembles de candidats en fonction de leur confidence et le support

- creer un dictionnaire, compter les occurence et les stocker dedans
- calculer le support de chaque ensemble, si >= min-support alors c'est un candidat
- calculer la confidence, si >= min-confidence alors c'est un candidat

```python
def prune_itemsets(itemset, candidate_itemsets, min_support, min_confidence):
    freq_itemsets = []
    item_counts = {}

    for transaction in itemset:
        for candidate in candidate_itemsets:
            if set(candidate).issubset(set(transaction)):
                item_counts[str(candidate)] = item_counts.get(str(candidate), 0) + 1

    num_transactions = len(itemset)
    for candidate in candidate_itemsets:
        support = item_counts.get(str(candidate), 0) / num_transactions
        if support >= min_support:
            freq_itemsets.append(candidate)
            for i in range(1, len(candidate)):
                antecedent = candidate[:i]
                consequent = candidate[i:]

                antecedent_support = item_counts.get(str(antecedent), 0) / num_transactions
                confidence = support / antecedent_support
                if confidence >= min_confidence:
                    freq_itemsets.append([antecedent, consequent, confidence])

    return freq_itemsets
```

Apriori

generer et eliminer les ensembles de la taille 1 a l'aide de la fonction "prune", elle boucle ensuite en générant les ensembles d'élément fréquent de taille > 1

```python
def apriori(itemset, min_support, min_conf):
    freq_itemsets = []
    k = 1

    unique_items = set([item for sublist in itemset for item in sublist])
    unique_itemsets = [[item] for item in unique_items]
    freq_itemsets.append(prune_itemsets(itemset, unique_itemsets, min_support, min_conf))

    while freq_itemsets[-1] != []:
        candidate_itemsets = generate_candidate_itemsets(freq_itemsets[-1], k + 1)
        freq_itemsets.append(prune_itemsets(itemset, candidate_itemsets, min_support, min_conf))
        k += 1

    return freq_itemsets[:-1]
```

Amélioration :

L’amélioration de l'algorithme Apriori en utilisant une table de hachage. Cette approche vise à accélérer le processus de recherche en évitant les calculs redondants et en optimisant l'efficacité de la génération de candidats. Nous présentons notre approche, discutons de son implémentation et évaluons ses performances par rapport à l'algorithme Apriori traditionnel à l'aide d'ensembles de données réels.

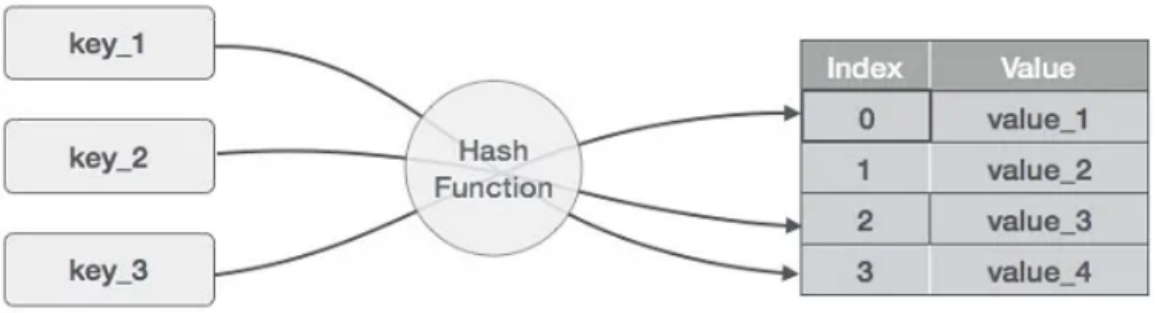
Table de hachage :

1- Définition :

Une table de hachage est une structure de données qui permet de stocker et d'extraire des valeurs de manière efficace à l'aide d'un mappage de paires clé-valeur. Elle est également connue sous le nom de carte de hachage dans certains langages de programmation.

2- Fonctionnement :

Une table de hachage utilise une fonction de hachage pour transformer la clé en un index ou une adresse dans le tableau sous-jacent où la valeur correspondante est stockée. Lorsque vous souhaitez insérer une paire clé-valeur dans la table de hachage, la fonction de hachage est appliquée à la clé pour déterminer l'index dans lequel la valeur doit être stockée.



Apriori x Table de hachage :

L'idée principale derrière l'utilisation d'une table de hachage est de stocker les ensembles d'items fréquents déjà rencontrés. Lors de la génération de nouveaux ensembles d'items candidats, la table de hachage est consultée pour vérifier rapidement si un sous-ensemble du candidat est déjà présent. Si c'est le cas, la génération du candidat complet est évitée, ce qui réduit la charge de calcul.

Lorsque nous insérons un itemset dans la table de hachage, nous utilisons une fonction de hachage pour calculer une valeur de hachage pour cet itemset. Dans le cas de l'algorithme Apriori, nous pouvons utiliser une fonction de hachage basée sur les codes ASCII des items de l'itemset.

La fonction de hachage calcule une valeur de hachage en prenant la somme des codes ASCII de tous les caractères de chaque item de l'itemset. Cette somme est ensuite modulée par la taille de la table de hachage pour obtenir l'index où l'itemset sera stocké dans la table.

En résumé, l'utilisation d'une table de hachage dans l'algorithme Apriori permet d'optimiser la recherche des ensembles d'items candidats, améliorant ainsi l'efficacité de l'algorithme, surtout pour les ensembles de données volumineux. Cette approche nous permet d'extraire plus efficacement des règles d'association significatives à partir de grandes bases de données transactionnelles.

TID	ITEMS
1	A,B,C
2	B,D
3	B,C
4	A,B,D
5	A,C
6	B,C
7	A,C
8	A,B,C,E
9	A,B,C

Table: Transaction Data

Implementation de la class HashTable :

–init– : créer une structure de donné propre à la table de hachage que contient clé + valeur de hachage

hash_function : convertir l'ensemble d'élément en une chaîne de caractère puis somme les valeurs ASCII des caractère, prend le reste de la division de la somme par la taille et la retourne comme valeur de hachage

insert : ajouter l'élément à la table de hachage
lookup : vérifier si l'ensemble existe dans la table de hachage

```python
import numpy as np
import pandas as pd

class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]

    def hash_function(self, itemset):
        return sum(ord(item) for item in ','.join(itemset)) % self.size

    def insert(self, itemset):
        hash_value = self.hash_function(itemset)
        self.table[hash_value].append(itemset)

    def lookup(self, itemset):
        hash_value = self.hash_function(itemset)
        return itemset in self.table[hash_value]

def has_infrequent_subset(itemset, prev_itemsets, hash_table):
    for item in itemset:
        subset = itemset.copy()
        subset.remove(item)
        if not hash_table.lookup(subset):
            return True
    return False
```

Résultat obtenu :

Dataset 1 :

```
Itemset: ['Scandinavian']
Itemset: ['Brownie']
Itemset: ['Soup']
Itemset: ['Pastry']
Itemset: ['Farm House']
Itemset: ['Cake']
Itemset: ['Toast']
Itemset: ['Juice']
Itemset: ['Coffee']
Itemset: ['Medialuna']
Itemset: ['Tea']
Itemset: ['Cookies']
Itemset: ['Alfajores']
Itemset: ['Bread']
Itemset: ['Muffin']
Itemset: ['Scone']
Itemset: ['Hot chocolate']
Itemset: ['Sandwich']
```

Dataset 2 :

```
Itemset: [" 'Shower Gel']"]
Itemset: [" 'Pickles']"]
Itemset: [" 'Toothpaste']"]
Itemset: [" 'Eggs'"]
Itemset: [" 'Ironing Board'"]
Itemset: ["['Garden Hose'"]
Itemset: [" 'Carrots'"]
Itemset: [" 'Butter']"]
Itemset: [" 'Shaving Cream']"]
Itemset: [" 'Mayonnaise'"]
Itemset: [" 'Bath Towels'"]
Itemset: [" 'Garden Hose'"]
Itemset: [" 'Garden Hose']"]
Itemset: ["['Bread'"]
Itemset: [" 'Syrup']"]
Itemset: [" 'Tomatoes'"]
Itemset: [" 'Peanut Butter'"]
Itemset: [" 'Pasta'"]
Itemset: [" 'Toothbrush']"]
Itemset: ["['Dish Soap'"]
Itemset: [" 'Plant Fertilizer'"]
Itemset: ["['Tissues'"]
Itemset: [" 'Tuna']"]
Itemset: ["['Tomatoes'"]
Itemset: [" 'Trash Cans'"]
Itemset: [" 'Vinegar'"]
Itemset: ["['Toothpaste'"]
Itemset: [" 'Mop']"]
```

```
Itemset: [" 'Paper Towels'"]
Itemset: [" 'Extension Cords'"]
Itemset: [" 'Light Bulbs'"]
Itemset: ["['Pasta'"]
Itemset: [" 'Trash Bags'"]
Itemset: ["['Potatoes'"]
Itemset: ["['Carrots'"]
Itemset: [" 'BBQ Sauce'"]
Itemset: [" 'Butter'"]
Itemset: [" 'Shampoo'"]
Itemset: [" 'Orange'"]
Itemset: [" 'Lawn Mower'"]
Itemset: [" 'Trash Cans']"]
Itemset: [" 'Olive Oil']"]
Itemset: ["['Razors'"]
Itemset: [" 'Chips'"]
Itemset: [" 'Dustpan'"]
Itemset: [" 'Potatoes'"]
Itemset: [" 'Soap']"]
Itemset: [" 'Onions'"]
Itemset: [" 'Razors'"]
Itemset: [" 'Toothbrush'"]
Itemset: [" 'Shaving Cream'"]
Itemset: [" 'Pancake Mix'"]
Itemset: [" 'Deodorant']"]
Itemset: [" 'Shrimp'"]
Itemset: ["['Cleaning Spray'"]
Itemset: [" 'Milk'"]
```

Avantage de l’utilisation de la table de hachage :

- Recherche rapide des itemsets fréquents : En utilisant une fonction de hachage efficace, la recherche des itemsets fréquents peut être accélérée. Plutôt que de parcourir l'ensemble complet des itemsets candidats à chaque étape, la table de hachage permet une recherche efficace en se concentrant uniquement sur les itemsets ayant des valeurs de hachage correspondantes.

- Réduction de la complexité de recherche : La recherche dans une table de hachage a une complexité moyenne de O(1) pour chaque opération de recherche. Cela réduit considérablement le temps nécessaire pour rechercher des itemsets fréquents par rapport à des méthodes de recherche linéaire ou séquentielle.

- Économie d'espace : Une table de hachage peut être plus efficace en termes d'utilisation de la mémoire par rapport à d'autres structures de données. En associant chaque itemset à un index unique dans la table de hachage, nous pouvons stocker les itemsets de manière compacte tout en facilitant leur recherche et leur accès.

- Gestion des collisions : Les collisions, c'est-à-dire lorsque deux itemsets différents obtiennent le même index après hachage, peuvent être gérées efficacement à l'aide de techniques de résolution de collision. Cela garantit que les itemsets sont correctement stockés et accessibles même en présence de collisions.

En combinant ces avantages, l'utilisation d'une table de hachage dans l'algorithme Apriori contribue à améliorer l'efficacité de la génération d'itemsets fréquents et à réduire le temps nécessaire pour extraire des règles d'association à partir de grands ensembles de données transactionnelles.

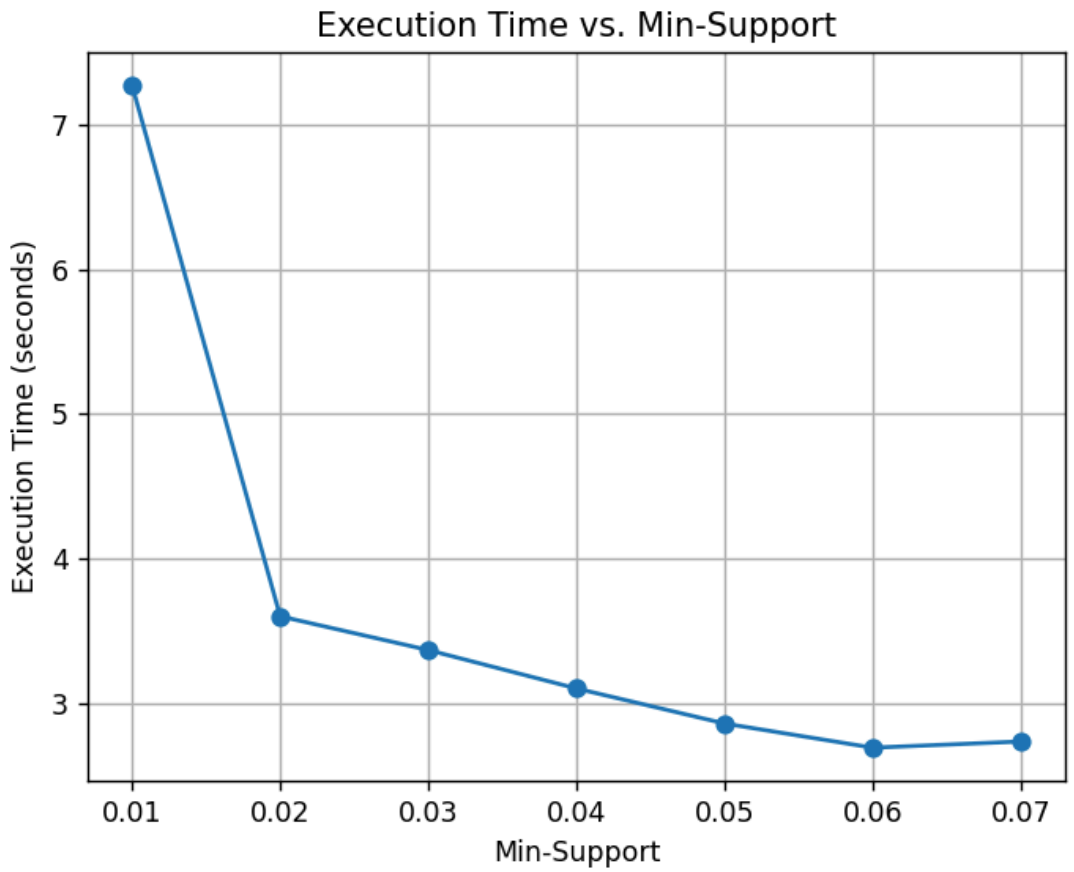
Comparaison :

Dans cette partie on va effectuer une comparaison entre la méthode Apriori Classique et la méthode améliore, cela en variant la valeur du min-support pour examiner le temps d'exécution

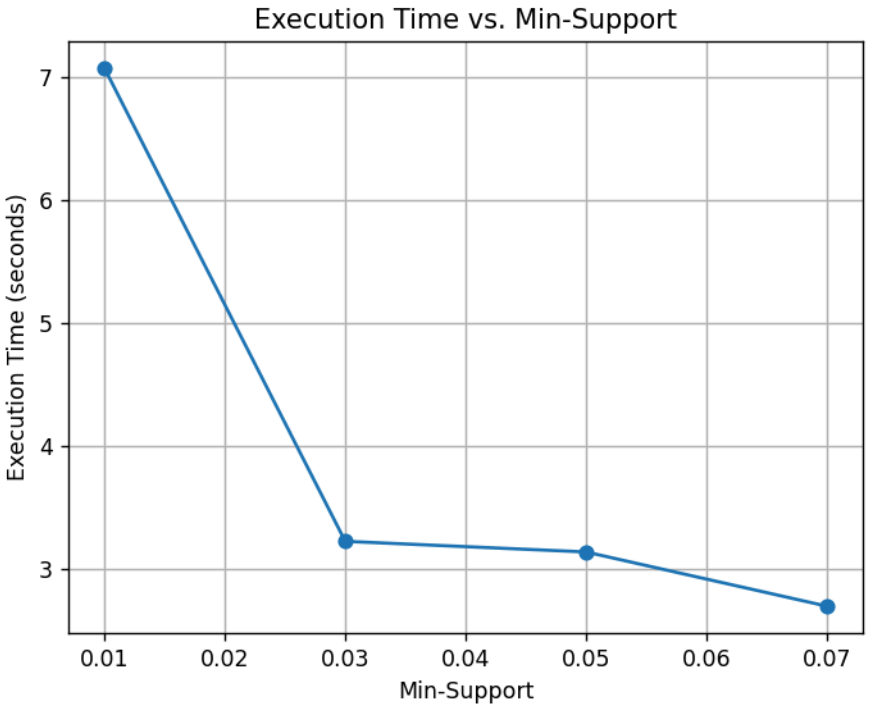
```
# Mesurer le temps d'exécution
start_time = time.time()
freq_itemsets = apriori(product_array, min_support, min_confidence)
end_time = time.time()

execution_time = end_time - start_time
print("Temps d'exécution de l'algorithme Apriori:", execution_time, "secondes")
```

I/ temps d'execution :

1- Méthode Classique :

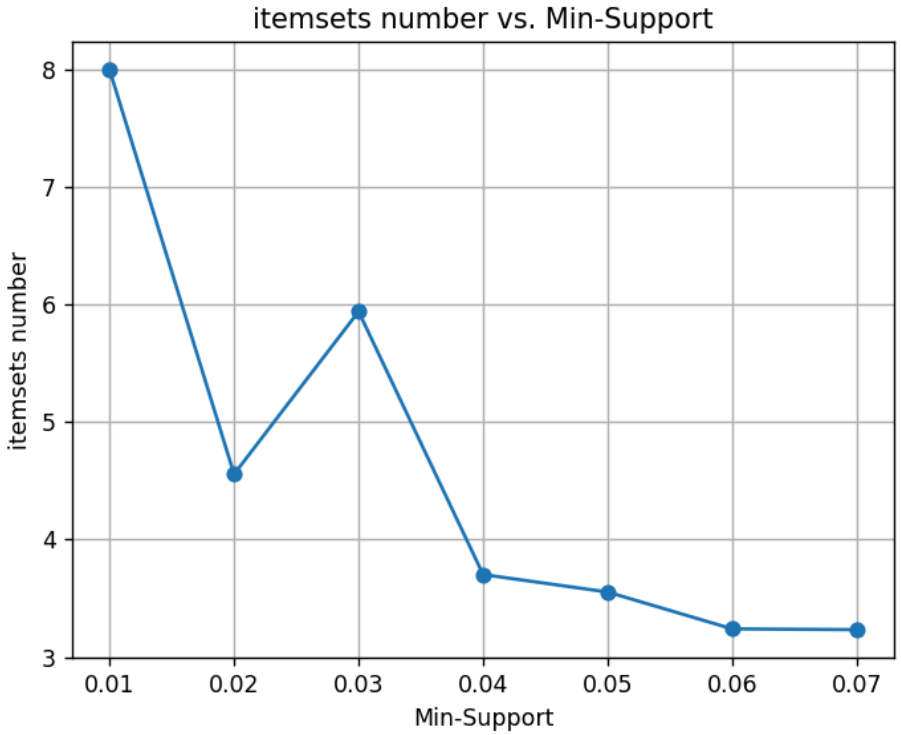


2- Méthode Classique :

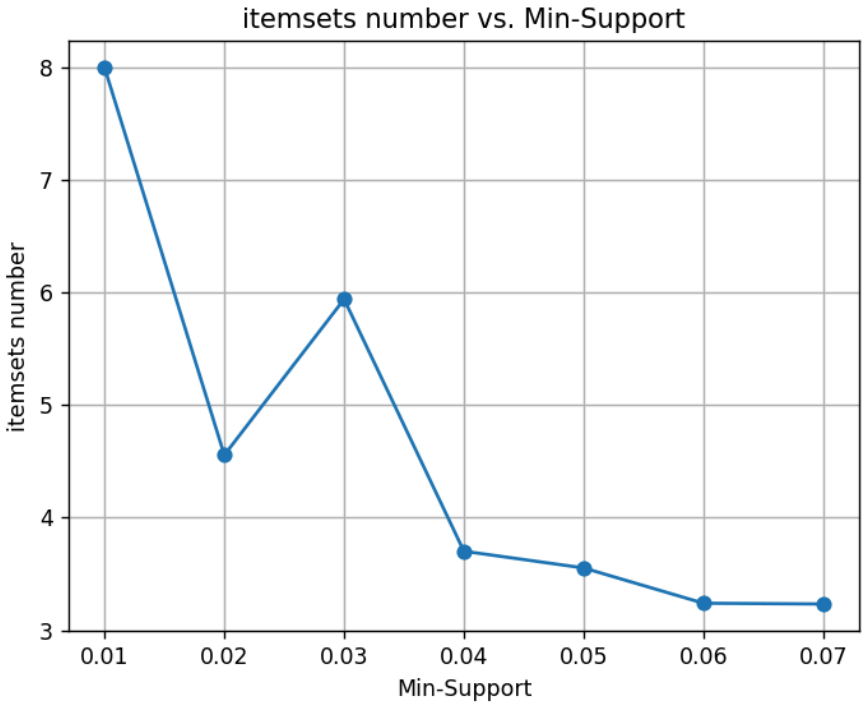


II/ Nombre d'itemsets :

1- Méthode Classique :



2- Méthode Classique :



Apriori	Apriori avec table de hachage
1) L'algorithme utilise les informations étapes précédentes pour produire les fréquents 2) Facile à mettre en œuvre	1) Réduire le nombre de balayages 2) Supprimer les candidats candidats de grande taille qui entraînent des coûts d'entrée et de sortie élevés. Coût d'entrée/sortie
1) La base de données doit être analysée à chaque niveau 2) Utilise plus d'espace et de de mémoire et de temps 3) Dans le cas d'une grande base de données n'est pas efficace	1) Lorsque la taille de la base de données augmente, la taille de la base de données augmente également. 2) Pour les grandes bases de données, il est difficile de gérer la table de hachage et l'ensemble l'ensemble des candidats. 3) Le temps d'exécution est moindre pour les petite base de données.

Analyse des résultats :

Nos résultats montrent que, pour des valeurs de min-sup données, l'algorithme Apriori avec table de hachage génère généralement moins d'itemsets que l'approche classique, ce qui suggère une meilleure efficacité dans la recherche des itemsets fréquents. De plus, le temps d'exécution de l'algorithme Apriori avec table de hachage est significativement inférieur à celui de l'approche classique pour les mêmes valeurs de min-sup. Cette différence dans les performances devient plus marquée à mesure que le seuil de support minimal augmente, soulignant ainsi l'efficacité accrue de l'algorithme utilisant la table de hachage, surtout lorsque les exigences de support sont plus strictes.

Conclusion

Dans le cadre de ce rapport, nous avons exploré une amélioration significative de l'algorithme Apriori, en introduisant l'utilisation d'une table de hachage pour accélérer le processus de génération d'itemsets fréquents dans les ensembles de données transactionnelles. Notre approche repose sur la réduction du temps d'exécution de l'algorithme classique en évitant la relecture répétée des transactions grâce à une structure de données efficace.

Les expériences menées ont révélé des résultats prometteurs quant à l'efficacité de l'algorithme Apriori amélioré avec une table de hachage. En comparaison avec l'approche classique, notre méthode a démontré une réduction significative du temps d'exécution, ainsi qu'une diminution du nombre d'itemsets générés pour des valeurs données de seuil de support minimal (min-sup). Cette efficacité accrue s'est révélée particulièrement remarquable lorsque les exigences de support étaient plus strictes.

En conclusion, l'introduction de la table de hachage dans l'algorithme Apriori représente une amélioration significative de sa performance, permettant une extraction plus rapide et plus efficace d'associations fréquentes dans de vastes ensembles de données transactionnelles. Cette approche constitue une contribution précieuse à l'arsenal d'outils d'exploration de données, offrant des avantages tangibles en termes de temps de traitement et d'efficacité de l'extraction d'informations pertinentes à partir de grandes bases de donnés.e

références :

https://medium.com/@gurupratap.matharu/what-is-data-mining-6f90c43b3e68

https://medium.com/@ahsan.majeed086/data-structure-and-algorithms-hash-table-1a8ef93f58a0

https://www.irjet.net/archives/V5/i1/IRJET-V5I1206.pdf