Projet d’apprentissage

Fouille d’itemsets

Groupe CGKS :
KOVIAZINA Aleksandra
GAYET Tiphaine
SELVAKUMAR Mathusan
CHENG Jewin

Enseignant : LAZAAR Nadjib
Cours : Projet Apprentissage

Date de rendu : 21/05/2025

Introduction

La fouille de motifs fréquents (ou *frequent itemset mining*) est une tâche essentielle en data mining. Elle consiste à identifier des ensembles d'items apparaissant fréquemment dans une base de transactions. Cette approche est largement utilisée dans des domaines tels que la recommandation de produits, la détection de comportements ou l'analyse de données biologiques. De nombreux algorithmes ont été développés pour répondre à cette problématique, chacun ayant ses forces et ses limitations en fonction des caractéristiques des données ou des contraintes imposées.

Dans ce projet, deux approches complémentaires ont été explorées :

- **SPMF**, une bibliothèque Java contenant une large collection d'algorithmes classiques de fouille de motifs, comme FP-Growth, LCM, Zart, etc.
- **ChocoMining**, une bibliothèque fondée sur un *solver* de satisfaction de contraintes (CSP), permettant une fouille déclarative par l'ajout de contraintes paramétrables.

L'objectif principal de ce projet est de développer une application web permettant à un utilisateur de soumettre une requête de fouille sur un dataset, et de recevoir automatiquement les résultats produits par l'algorithme le plus adapté. Ce choix d'algorithme repose sur un modèle de classification supervisée entraîné à partir de benchmarks expérimentaux.

Ainsi, l'application sélectionne dynamiquement entre **ChocoMining** et **SPMF**, exécute l'algorithme recommandé, et affiche les motifs extraits de manière lisible et interactive.

Le projet a été structuré en deux grandes phases :

1. **Étude comparative expérimentale** des algorithmes ChocoMining et SPMF sur plusieurs jeux de données. Cette phase a permis de générer un grand nombre de requêtes et d'en extraire des métriques (temps d'exécution, nombre de motifs, etc.), en vue d'entraîner un modèle de classification automatique.

2. **Développement Full Stack**, intégrant à la fois le backend (dont le rôle est de charger le modèle, choisir l'algorithme adapté et exécuter la fouille) et le frontend (interface utilisateur permettant de formuler les requêtes et de visualiser les résultats renvoyés par le backend).

Voici une liste des technologies utilisées :

- **Java** avec **Maven** pour la structuration du projet backend et l'intégration des bibliothèques SPMF et ChocoSolver.
- **Spring Boot** pour l'implémentation de l'API REST.
- **Python** avec **scikit-learn** pour l'entraînement du modèle de classification, exporté sous la forme d'un fichier `.pkl` et **flask** pour le service de prédiction sur une requête.
- **React** pour le développement de l'interface utilisateur web, avec gestion dynamique des formulaires et visualisation des motifs extraits.

Cette architecture permet une communication fluide entre les différentes couches de l'application.

La première partie de ce rapport est consacrée à l'étude expérimentale, à l'intégration des algorithmes et à la construction du modèle prédictif. La seconde partie décrit le développement de l'application web et les choix techniques associés. Enfin, une analyse comparative des résultats permet d'évaluer la pertinence de notre approche et d'envisager des pistes d'amélioration.

1. Étude comparative et modélisation

1.1. Première Implémentation : Exécution des 9 Requêtes sur les Deux Librairies de DataMining

ChocoMiner
La structure du solveur étant relativement simple à comprendre, il a surtout été nécessaire de bien définir les contraintes pour chaque requête. Toutefois, dans le cas de la génération de motifs rares, une adaptation de l'algorithme a été nécessaire. En effet, SPMF ne propose pas de solution correspondant exactement à la définition des motifs rares présentée en cours.
Selon l'algorithme de SPMF, tout motif rare de taille ≥ 2 doit obligatoirement inclure au moins un item rare de taille 1.

SPMF
Pour l'implémentation avec SPMF, il a été nécessaire de sélectionner un algorithme adapté à chaque requête, en fonction de ses contraintes spécifiques. Nous n'avons pas mené d'étude comparative approfondie entre tous les algorithmes possibles.

Nous expliquons ci-dessous le raisonnement ayant guidé le choix de l'algorithme pour chaque requête:

- **Motifs fréquents - LCMFreq et Motifs fermés - LCM**

Plusieurs algorithmes sont disponibles pour résoudre ces types de problèmes. Nous avons choisi de partir avec LCM et LCMFreq, car ce sont les algorithmes abordés en cours mais aussi gagnants de la compétition de FIMI 2004 pour le minage de motif fréquent et clos. Nous étions déjà familiers avec son implémentation, ce qui a facilité son intégration et son adaptation.

- **Motifs maximaux - FPMax**

Dans ce cas, seuls deux algorithmes étaient réellement adaptés au problème. Nous avons choisi de tester FPMax car d'après la documentation cet algo est plus efficace. De plus, son implémentation est particulièrement claire, ce qui a facilité son utilisation.

- **Motifs rares - RPGrowth**

Nous avons choisi cet algorithme car c'était le seul réellement adapté au problème, les autres comme AprioriInverse et CORI ajoutent des contraintes qui ne sont pas forcément voulu dans notre cas. Toutefois, comme mentionné précédemment, une particularité de cet algorithme est que les motifs rares qui sont minés au obligatoirement un des subset de taille 1 qui est rare. Cela est probablement dû à l'implémentation basé sur des arbres. La modification du code côté SPMF étant trop complexe, nous avons préféré adapter l'implémentation de ChocoMining, afin de maintenir une cohérence entre les deux approches.

- **Motifs Générateurs - AlgoZart**

Les algorithmes de SPMF qui traitent la requête de motifs générateurs sur des itemsets sont au nombre de trois : DefMe, Pascal et Zart. Nous avons choisi Zart pour sa facilité d'implémentation comparé aux deux autres algorithmes, en effet l'algorithme DefMe utilise des structures d'itemsets avec une implémentation différentes (les bitsets) des autres algorithmes. L'algorithme Pascal utilise aussi un itemset avec une implémentation particulière mais la récupération difficile des résultats nous a conduit à ignorer cet algorithme.
On a également décidé d'ignorer l'ensemble vide comme motif générateur car il est pas très informatif et pour garder la cohérence avec l'implémentation de la requête avec choco-mining.

- **Motifs minimaux - AprioriRare**

Nous avons choisi cet algorithme car c'était le seul réellement adapté au problème.

- **Motifs clos avec taille comprise entre X et Y, Inclusion des éléments, Exclusion des éléments - LCM**

Nous n'avons pas identifié d'algorithme adapté pour ce cas particulier. Par conséquent, nous avons opté pour un traitement manuel des motifs clos générés par LCM. On a laissé la possibilité de choisir le support de fréquence à fournir à l'algorithme LCM pour permettre un minimum de flexibilité sur ce paramètre. Les motifs sont ensuite filtrés manuellement en ajoutant à une nouvelle collection les motifs respectant les conditions de la requête.

1.2. Benchmarking : description des datasets FIMI utilisés, protocole de test.

Pour évaluer les performances de notre approche, nous avons réalisé un benchmarking à l'aide de neuf jeux de données FIMI présentant des métadonnées variées. Une description détaillée de ces jeux de données est présentée dans le tableau ci dessus:

Data Set	Nb. of items	Nb. of transactions	Density
anneal.dat	90	812	47.78%
chess.dat	76	3196	50%
connect.dat	130	67557	33.85%
pumsb.dat	2114	49046	3.55%
heart-cleveland.dat	96	296	47.92%
eisen.dat	9839	2464	0.37%
mushroom.dat	113	8124	19.47%
iris.dat	16	150	37.5%
contextPasquier99.dat	5	5	64%

Afin de garantir une comparaison équitable, nous avons investi un temps conséquent dans l'analyse et la compréhension approfondie des codes sources des deux bibliothèques utilisées : ChocoMining et SPMF. Pour chacune, nous avons mesuré le temps total d'exécution, incluant la création du modèle ainsi que l'ajout des contraintes, car ces étapes seraient exécutées à chaque requête dans un environnement de production.

Nous avons fixé un temps limité (timeout) de 15 secondes pour chaque requête. Chaque test a été lancé pour les 9 requêtes sur les 9 jeux de données, chacun avec quatre seuils de fréquence minimale : 0.2, 0.4, 0.6 et 0.8, soit un total de 324 requêtes par algorithme. Le temps maximal théorique pour la génération complète des benchmarks s'élève donc à 1 heure et 21 minutes par algorithme. Tous les tests ont été exécutés sur la même machine, afin de garantir la comparabilité des résultats entre les différentes approches.

Dans ChocoMining, le respect du timeout est relativement simple à observer, car le solveur retourne les solutions de manière incrémentale. Nous nous sommes assurés

que chaque algorithme ait le temps de générer au moins une solution avant d'être interrompu.

Avec SPMF, l'implémentation du timeout a nécessité une intervention plus rigoureuse. Nous avons dû modifier le code source de six algorithmes : AlgoLCMFreq, AlgoLCM, AlgoFPMax, AlgoRPGrowth, AlgoZart, et AlgoAprioriRare. Pour tous ces algorithmes, une variable globale endTime a été ajoutée afin de contrôler la durée d'exécution.

Les algorithmes de SPMF peuvent être classés en deux catégories:

- Algorithmes itératifs (AlgoZart, AlgoAprioriRare) : un contrôle du timeout a été intégré à chaque itération de la boucle principale de génération de motifs.
- Algorithmes récursifs (AlgoLCMFreq, AlgoLCM, AlgoFPMax, AlgoRPGrowth) : la gestion du timeout a été intégrée dans chaque appel récursif. Cela a nécessité la modification des signatures des fonctions afin de permettre la remontée d'un signal d'arrêt via les valeurs de retour, permettant ainsi d'interrompre proprement la génération dès que le délai est dépassé.

1.3. Labellisation des performances

Tout d'abord, nous avons réalisé une analyse des écarts sur les requêtes ayant abouti pour les deux algorithmes. Cette étape a permis d'identifier et de corriger plusieurs points importants afin d'assurer l'équité et la cohérence des résultats.

Les ajustements suivants ont été apportés suite à cette analyse:

- Exclusion de l'ensemble vide dans les résultats, afin d'harmoniser les sorties entre SPMF et ChocoMining ;
- Adaptation du choix des algorithmes SPMF, en sélectionnant ceux les plus appropriés et comparables à la méthode utilisée dans ChocoMining;
- Harmonisation du calcul du minimum de support (minSupport), afin qu'il soit défini de manière équivalente pour les deux implémentations, quelles que soient les caractéristiques des jeux de données.
- Adaptation de certains comportements côté choco pour reproduire ceux de spmf (RPGrowth)

Suite à ces corrections, l'ensemble des benchmarks a été régénéré afin d'obtenir des résultats fiables et comparables sur des bases de données cohérentes.

Le script Python **study/add_classes.py** a pour objectif de générer un fichier **classes.csv** contenant des informations statistiques et comparatives sur l'exécution de requêtes de fouille de motifs fréquents, à partir de ChocoMining et SPMF.

Le fichier final contient, pour chaque requête :

- des métadonnées issues des fichiers de données .dat (nombre d'items uniques, nombre de transactions, densité)(voir ligne 5-32) ;

- des mesures de performance (temps d'exécution et nombre de motifs extraits) issues des résultats respectifs des outils ChocoMining (**choco.csv**) et SPMF (**spmf.csv**) ;

- une classe, dont la valeur permet d'identifier quel outil a mieux performé selon les critères définis.

Pour chaque requête, la classe est déterminée selon deux scénarios :

Cas 1 : les deux algorithmes terminent dans le délai imparti (voir ligne 60-61)

Si les deux durées d'exécution sont inférieures à 15 000 millisecondes alors on suppose que le nombre de motifs générés sont les même :

- L'algorithme le plus rapide est considéré comme le meilleur
- La classe est donc fixée à :
    - **0** si ChocoMining est plus rapide,
    - **1** si SPMF est plus rapide.

Cas 2 : au moins un des algorithmes dépasse le délai de 15 secondes(voir ligne 62-63)

Si l'un ou les deux algorithmes dépassent la limite : l'algorithme qui a généré le plus grand nombre de motifs est considéré comme plus performant.

Cette approche vise à prendre en compte à la fois l'efficacité (temps) et l'exhaustivité (quantité de résultats), afin de refléter une évaluation plus juste dans les cas où la contrainte de temps impacte le comportement des algorithmes

1.4. Le modèle de classification et le sélectionneur de librairie

La classification vise à déployer un service pour prédire automatiquement la librairie optimale (ChocoMining ou SPMF) pour une requête de fouille donné, en se basant sur :

- **Métadonnées** des datasets: **densité, nombre d'items, nombre de transactions** (calculé avec la fonction *statistics* de *dataprep.py*)
- Paramètres de requête : **fréquence**, **type de requête**, **nom du dataset**

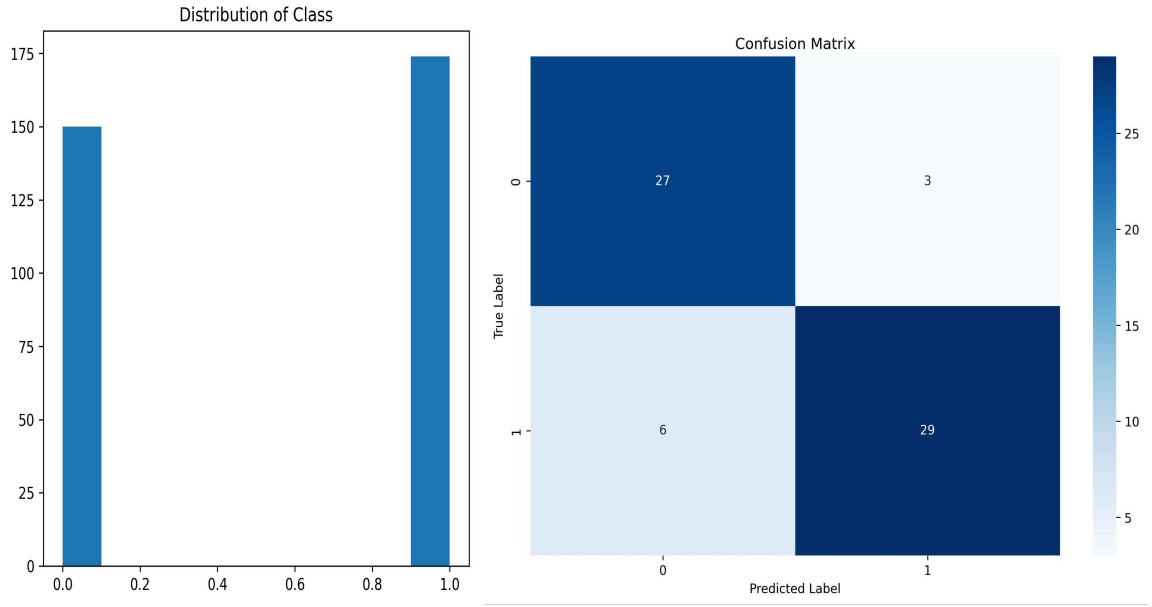
- Performances historiques : **benchmark** des 9 requêtes sur 36 combinaisons (fréquence*requête*dataset) (fichier classes.csv)

Pipeline de Classification (main.py : lignes 18-59)**:**

- L'entrée est le dataset classes.csv
- Préparation des features : one hot encoding (colonnes Query, File), standardisation (Frequency, Nbitems, Nbtransactions, Density), séparation du label Class (dataprep.py : lignes 88-137)
- Découpage train et test (80/20) et stratifié selon le label
- Entraînement et tuning à partir d'un GridSearchCV sur un RandomForest (paramètres de l'entrainement pipeline_utils.py : lignes 56-132)

On sauvegarde la pipeline entière dans le dossier pipeline avec les graphiques.

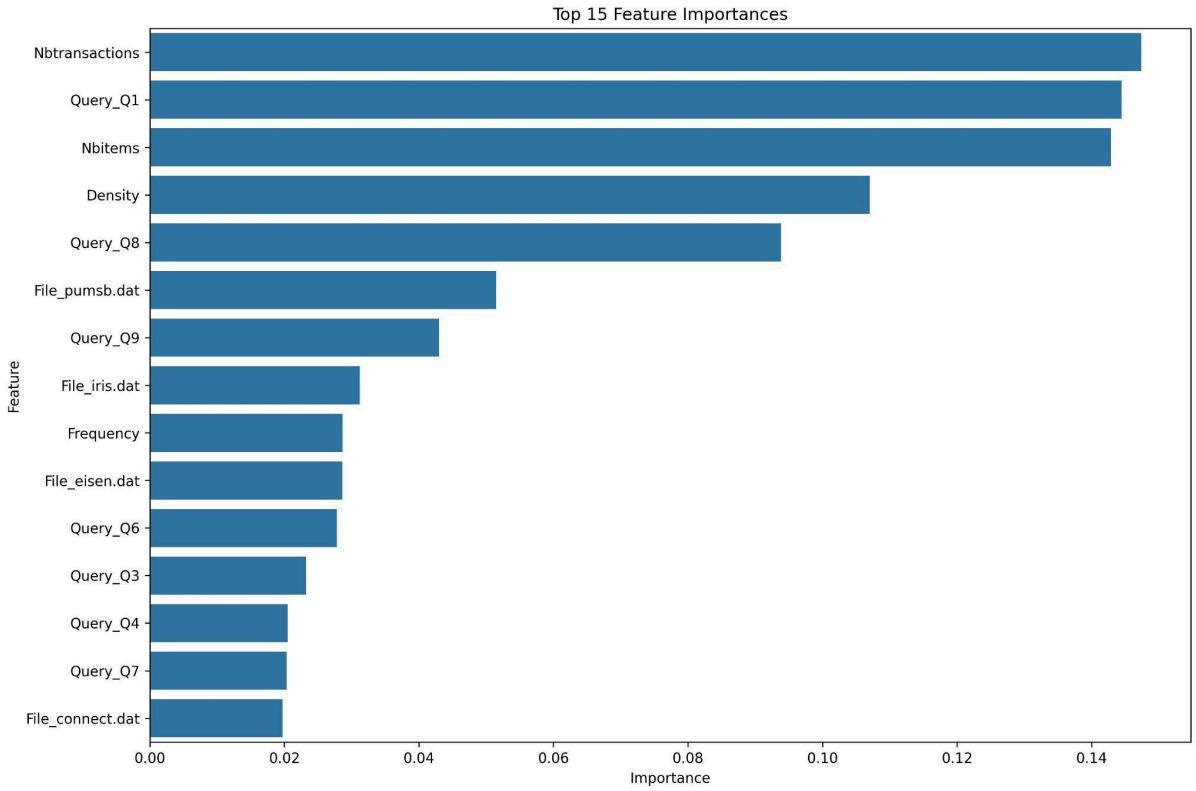
Analyses et évaluations :



Les deux graphes ci-dessus représentent respectivement la distribution du label (dataprep.py: lignes 107-121) Class (0 pour Choco et 1 pour Spmf) et la matrice de confusion des prédictions (pipeline_utils.py: lignes 120-152).

On peut remarquer que Spmf est un peu plus dominant que Choco en termes de labellisation par temps/nombre de motifs, signe que les algos spécialisés de spmf sont plus performants que l'algo déclaratif de choco.

Pour ce qui est de la matrice de confusion, les prédictions sont relativement correctes (on obtient ~85% d'accuracy) et on peut voir un peu plus d'erreur en terme de fausse prédiction pour choco plutôt que spmf.



On peut voir sur ce graphique les features (pipeline_utils.py: lignes 178-229) qui ont le plus joué dans la classification, on voit que les 3 métadonnées : nombre de transactions, nombre d'item et densité, sont très déterminants dans la classification.

Limitations et Améliorations :

On a pu constater au niveau des features que la requête fréquente (Q1) est plus déterminante que la densité et le nombre d'item or ce n'est pas forcément la feature qu'on veut dépendre pour choisir entre la librairie Spmf et Choco.

Le faible nombre de features ne joue pas en faveur de l'algorithme de prédiction.

L'entraînement a été uniquement fait sur certains algos de spmf or avec un benchmark plus affiné avec différents algorithmes de spmf cela pourrait augmenter la prédominance de spmf.

L'optimisation peut être améliorée avec un tuner comme Optuna et d'autres algorithmes de machine learning comme XGBoost pourrait augmenter les performances.

1.5. Etude sur les algorithmes de recherche pour ChocoMining

L'intégration des algorithmes dans SPMF repose sur des implémentations spécifiques à chaque type de problème. À l'inverse, ChocoMining utilise une approche générique : les contraintes sont ajoutées dynamiquement, mais la logique de résolution reste inchangée.

Pour cette raison, nous avons étudié l'impact du paramètre .**setSearch()** de Choco, qui permet de définir la stratégie de recherche. Choco construit un arbre de recherche binaire et utilise par défaut une recherche en profondeur. Or, les heuristiques de recherche influencent fortement les performances. Nous avons donc testé quatre stratégies supplémentaires dans notre version de ChocoMining afin d'évaluer leur impact sur le temps de résolution et la qualité des résultats.

- **inputOrderLBSearch**

C'est une stratégie de choix des variables et valeurs utilisée dans une recherche en profondeur. Recherche par défaut choisit les variables dans un ordre défini par solver. inputOrderLBSearch prends les variables dans l'ordre d'entrée et pour chaque variable, nous essayons la plus petite valeur d'abord.

- **inputOrderUBSearch**

Cette stratégie prend les variables dans l'ordre d'entrée et pour chaque variable, nous essayons la grande valeur d'abord.

- **minDomLBSearch**

Cette stratégie choisit la variable la plus restreinte (petit domaine) et essaie sa plus petite valeur. C'est une stratégie plus intelligente que l'ordre d'entrée, car elle tente de réduire l'espace de recherche plus vite.

- **minDomUBSearch**

Choisis la variable non instanciée qui a le domaine le plus petit (comme dans minDomLBSearch), mais essaie sa valeur la plus grande (upper bound) en premier.

Nous avons sélectionné, pour chaque requête, une stratégie de recherche spécifique à tester. Il est important de souligner que ces expérimentations ont été menées sous des contraintes de temps strictes. Ainsi, bien que plusieurs pistes d'optimisation aient été identifiées, nous n'avons pas pu explorer l'ensemble des alternatives possibles. Les résultats seront présentés requête par requête dans la suite de ce rapport.

Pour chaque question, 36 requêtes ont été exécutées (correspondant à 4 seuils de fréquence : 0.2, 0.4, 0.6, 0.8, appliqués à 9 fichiers différents). Un tableau récapitulatif est présenté ci-dessous. Il indique :

- la stratégie de recherche choisie pour chaque requête,
- le nombre de requêtes ayant donné de meilleurs résultats avec la stratégie par défaut(Nb def),
- le nombre de requêtes ayant donné de meilleurs résultats avec cette stratégie par rapport à la stratégie par défaut (Nb strat),
- le nombre de cas sans changement de performance (Nb equal)

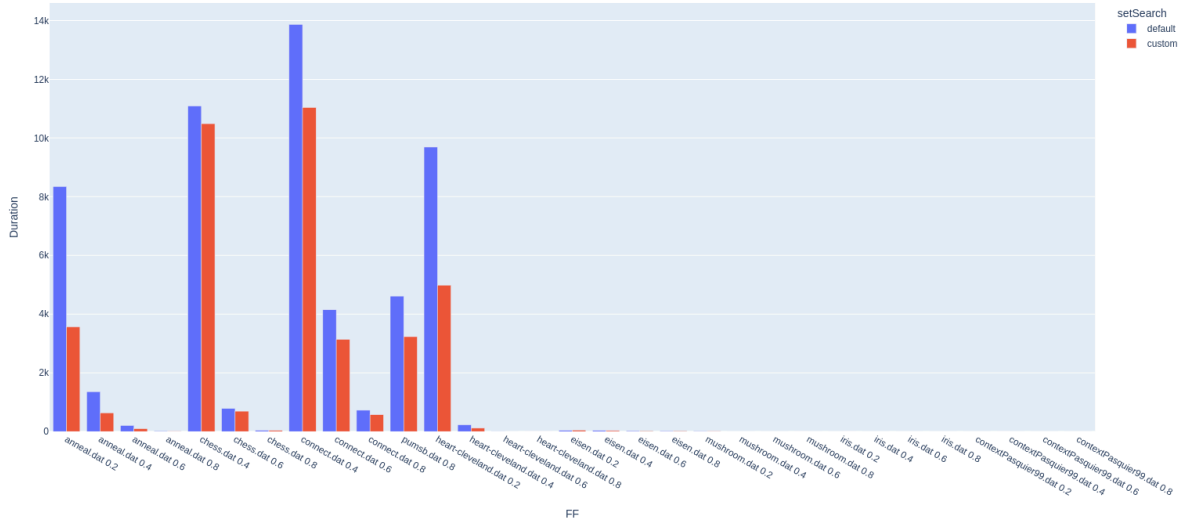
N	Question	Stratégie choisie	Explication	Nb def	Nb strat	Nb equal
1	Motifs fréquents	inputOrderLBSearch	On veut explorer toutes les combinaisons d'items, de manière ordonnée. Donc on choisit une stratégie exhaustive pour voir si l'ordre a une importance.	4	21	11
2	Motifs fermés	minDomUBSearch	Les motifs fermés sont souvent inclus plusieurs items.En assignant d'abord les bornes supérieures, on explore rapidement les motifs denses. La stratégie aide à pruner tôt si la fermeture n'est pas respectée.	5	25	6
3	Motifs maximaux	inputOrderUBSearch	Nous avons besoin d'une approche "gloutonne" vers les motifs larges pour l'exploration rapide des plus grands motifs.	7	19	10
4	Motifs rares	minDomLBSearch	Nous privilégions l'exploration de petits motifs ou ceux qui n'ont pas beaucoup d'items en commun avec les transactions	1	30	5
5	Motifs générateurs	inputOrderLBSearch	La stratégie favorise l'exploration rapide de motifs petits mais fréquents, et surtout non redondant	8	18	10
6	Motifs minimaux	inputOrderLBSearch	Nous cherchons d'abord les plus petits motifs.	18	11	7
7	Taille comprise entre X et Y	minDomLBSearch	Le choix de la variable avec le plus petit domaine permet de forcer les décisions sur les variables les plus contraintes.	1	26	9
8	Inclusion des éléments	inputOrderLBSearch	Une recherche exhaustive.	11	14	11
9	Exclusion des éléments	inputOrderLBSearch	Une recherche exhaustive.	3	23	10

Les résultats ne sont pas toujours concluants pour toutes les requêtes. Par exemple, pour la requête 6, la stratégie de recherche par défaut a donné de meilleures performances. Nous avons donc décidé de ne pas modifier la stratégie dans ce cas. De même, pour la requête 8, les améliorations apportées par la nouvelle stratégie sont trop faibles pour justifier un changement. Il serait nécessaire de poursuivre les tests pour identifier une stratégie plus adaptée.

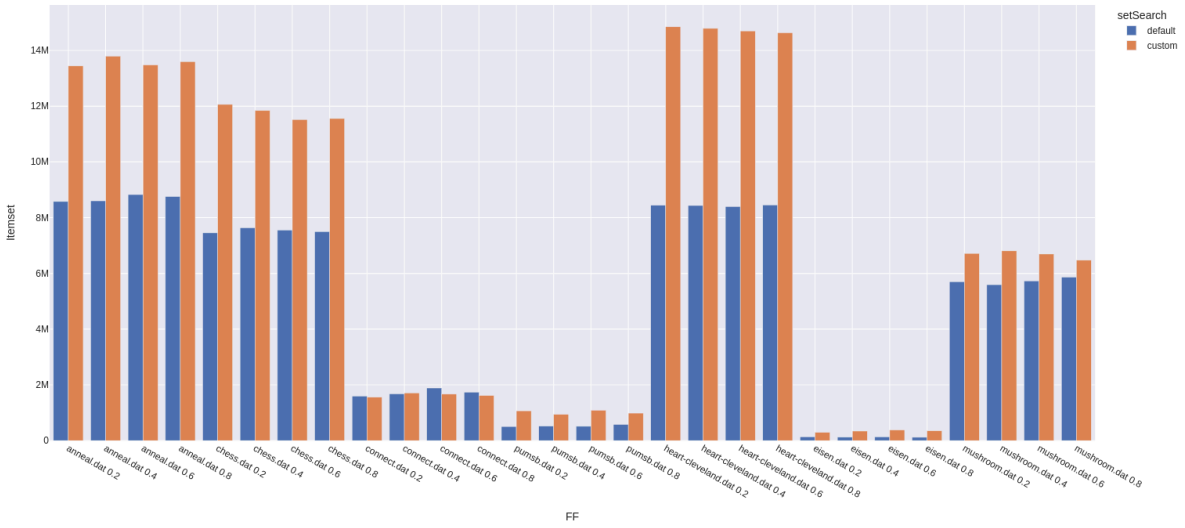
En revanche, pour la majorité des requêtes, le choix de stratégie a permis d'améliorer les performances. Les résultats moins satisfaisants pour les requêtes 6 et 8 sont probablement liés à un mauvais choix de stratégie, faute de temps pour tester plus d'options.

Enfin, comme le KPI utilisé ne reflète pas toujours bien les différences de performance, nous avons préparé une analyse graphique plus visuelle. Cette étude repose sur 2 types de cas:

- Les deux algorithmes terminent dans le délai imparti, on regarde la différence dans le temps d'exécution (Question 7)



- Au moins un des algorithmes dépasse le délai de 15 secondes, on regarde le nombre de patterns générés (Question 4)

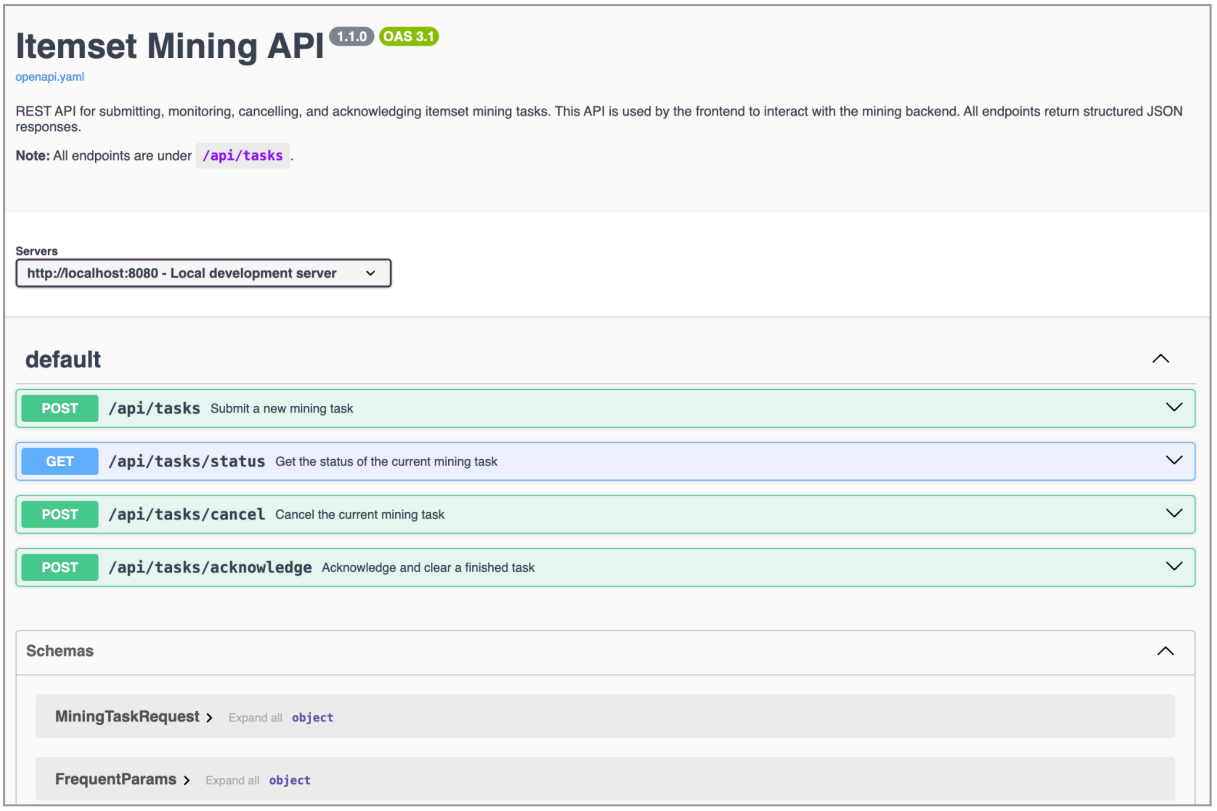


Certaines améliorations sont envisageables pour affiner cette étude. Par exemple, il serait pertinent de tester l'ensemble des stratégies de recherche sur toutes les requêtes, afin d'obtenir une vue d'ensemble plus complète de leur efficacité. De plus, l'ajout d'indicateurs de performance plus précis, tels que le temps gagné ou perdu par rapport à la stratégie par défaut, permettrait une analyse plus fine des résultats. Enfin, des travaux plus approfondis pourraient être menés, notamment en explorant la conception de stratégies de recherche personnalisées adaptées aux caractéristiques spécifiques des requêtes ou des jeux de données.

2. Développement de l'application web

Notre application web constitue l'interface entre les utilisateurs et les puissants algorithmes de fouille de données intégrés dans notre système.
L'architecture globale de notre application repose sur une séparation claire entre le frontend, une application React, et le backend, codé en Java, reliés par une API REST assurée par Spring Boot.

SwaggerUI (`make docs`) est utilisé pour fournir une documentation visuelle conforme à **OpenAPI** et permettre un test interactif des endpoints. Cela permettra surtout de faciliter l'exploration de l'API des utilisateurs avancés.



2.1. Backend

2.1.1. Structure du projet

L'arborescence du backend est organisée pour séparer clairement les responsabilités et faciliter la maintenance. Le code source principal se trouve dans `src/`.
Voici un extrait de l'arborescence :

```
src/
├── main/
│   ├── java/
│   │   └── com/
│   │       └── github/
│   │       └── cgks/
│   │           ├── Miner.java             → Interface pour les mineurs de données
│   │           ├── MiningController.java  → Gestionnaire du API REST
│   │           ├── MiningRequest.java     → Structure de données pour les requêtes
│   │           ├── MiningResult.java      → Structure de données pour les résultats
│   │           ├── MiningEngine.java      → Orchestrateur d'exécution des requêtes
│   │           ├── MiningSelector.java    → Sélection dynamique de l'algorithme
│   │           ├── choco/
│   │           │   └── ChocoMiner.java    → Implémentation ChocoMining
│   │           └── spmf/
│   │               └── SpmfMiner.java     → Implémentation SPMF
```

```
|   ├── python/
|   |     └── classifier_api.py      → API Flask pour exposer le modèle de classification
|   └── resources/
|         ├── datasets/              → Jeux de données utilisés pour les tests
|         └── model/                 → Modèle de classification sauvegardé (.pkl)
└── test/    → tests unitaires
```

Modules principaux :

- **MiningEngine** : point d'entrée du backend, orchestre le traitement des requêtes de fouille.
- **MiningSelector** : sélectionne dynamiquement l'algorithme à utiliser selon la prédiction du modèle.
- **ChocoMiner** et **SpmfMiner** : implémentent l'interface **Miner** pour intégrer respectivement Choco Mining et SPMF.

2.1.2. Intégration des algorithmes

Le backend intègre deux moteurs de fouille :

- **SPMF** (librairie Java d'algorithmes de fouille de motifs)
- **Choco Mining** (basé sur le solveur de contraintes Choco)

Chaque moteur est encapsulé dans une classe dédiée :

- **com.github.cgks.spmf.SpmfMiner**
- **com.github.cgks.choco.ChocoMiner**

Chacune de ces classes implémente l'interface **Miner**, qui définit les 9 méthodes d'extraction étudiées.

Par exemple, la méthode `extractClosed` dans **ChocoMiner** (lignes 163-192) montre comment le solveur Choco est utilisé pour extraire les itemsets fermés à partir d'une requête POST avec les contraintes appropriées et en itérant sur les solutions. On peut comparer la même méthode mais implémenté avec Spmf (lignes 109-132)

2.1.3. Sélection dynamique via le modèle

Le backend utilise un modèle de classification pour choisir dynamiquement l'algorithme de fouille le plus adapté à la requête utilisateur.
L'architecture de sélection dynamique s'organise autour d'un **service de prédiction** développé en Python (Flask), qui interagit avec notre sélectionneur **MiningSelector** en Java.

Le choix du modèle se déroule en 3 temps (lignes 80-112) :

- **Extraction des métadonnées de la requête utilisateur** issue du front (type de requête, dataset utilisé, support) et envoie de la requête au format JSON(lignes 83-101)
- **Enrichissement de la requête** avec des métadonnées et le modèle prédit la famille d'algorithmes la plus pertinente sous forme binaire. (lignes 39-63)
- **MiningSelector** choisit l'implémentation en fonction de la prédiction (**SpmfMiner** ou **ChocoMiner**). (lignes 104-107)

MiningEngine choisit ensuite la méthode (`extractFrequent`, `extractClosed`, etc.) à appeler en fonction de la requête.
L'utilisateur a également la possibilité de sélectionner explicitement la famille d'algorithmes auquel cas le choix est fait directement sans appel à l'API de prédiction. (lignes 58-70)

Avantages :

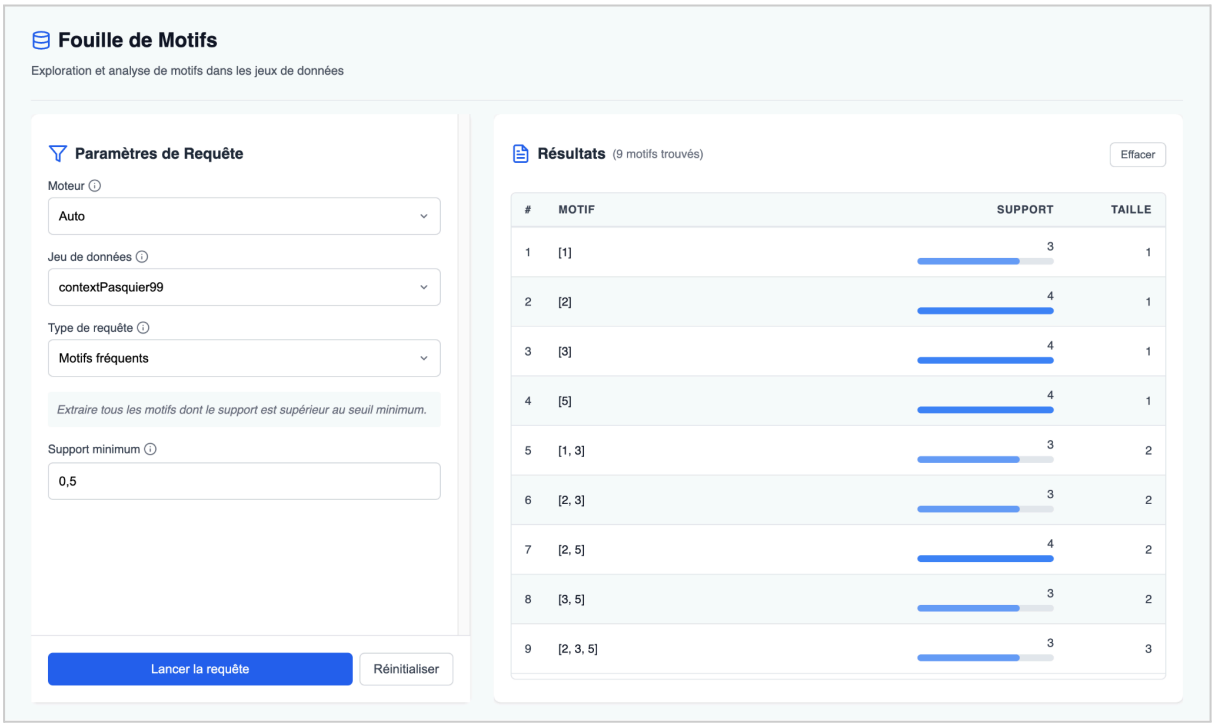
- Flexibilité: adaptation facile à différents jeux de données et types de requêtes
- Évolutivité : possibilité d'intégrer de nouveaux algorithmes sans modifier l'architecture

2.1.4. Bonnes pratiques

Le code backend respecte plusieurs bonnes pratiques :

- Principes SOLID : chaque classe a une responsabilité claire (Single Responsibility), l'interface **Miner** permet l'extension facile (Open/Closed), etc.
- Javadoc : chaque classe et méthode publique est documentée (voir par exemple la Javadoc de **ChocoMiner**).
- Gestion des erreurs : des exceptions spécifiques (**ParameterException**, **DatabaseException**, **MiningException**) sont utilisées pour distinguer les types d'erreurs et fournir des messages explicites.
- Modularité : l'architecture permet d'ajouter facilement de nouveaux algorithmes ou moteurs de fouille en implémentant l'interface **Miner**.

2.2. Frontend



2.2.1. Structure et composants

Le frontend est développé avec **React**.
L'arborescence typique du frontend se trouve dans motif-mining-app :

```
frontend/
└── motif-mining-app/
    └── src/
        ├── App.js                    → Composant principal de l'application React
        ├── index.css                 → Feuille de style globale
        ├── constants.js              → Définition des constantes globales
        ├── services/
        │   └── ApiService.js         → Fonctions pour interagir avec l'API backend
        ├── hooks/
        │   ├── useFormValidation.js  → Hook pour la validation des formulaires
        │   └── useContentHeight.js   → Hook pour gérer la hauteur de contenu
        └── components/
            ├── form/
            │   └── DataForm.jsx      → Formulaire de sélection des paramètres
            ├── results/
            │   └── ResultsTable.jsx  → Composant d'affichage des résultats (tableaux)
            └── ui/
                ├── Alert.jsx         → Composant d'alerte personnalisée
                └── ErrorBoundary.jsx → Composant de gestion des erreurs React
```

Composants clés :

- **App.js** : composant racine, gère l'état global, la logique de soumission, le polling, l'affichage des résultats et des alertes.
- **components/form/DataForm.jsx** : formulaire principal pour la sélection du moteur, du dataset, du type de requête et des paramètres.
- **components/results/ResultsTable.jsx** : affichage des résultats sous forme de tableau, avec gestion du chargement et du reset.
- **components/ui/Alert.jsx** : affichage des messages d'alerte (succès, erreur, info).
- **components/ui/ErrorBoundary.jsx** : gestion des erreurs d'exécution côté React.
- **hooks/useFormValidation.js** : validation dynamique des champs du formulaire.
- **services/ApiService.js** : fonctions pour communiquer avec l'API backend (soumission de tâche, polling, annulation…).

2.2.2. Communication avec le backend

La communication avec le backend se fait via une API REST.
Les appels sont centralisés dans src/services/**ApiService.js** :

- `submitTask(engine, dataset, query, params)` : envoie une requête POST pour lancer une fouille.
- `getTaskStatus()` : récupère l'état courant de la tâche (polling).
- `cancelTask()` : annule une tâche en cours.
- `acknowledgeTask()` : notifie le backend que le frontend a bien reçu le résultat.

Les classes **MiningRequest** et **MiningResult** permettent de sérialiser et désérialiser les messages communiqués entre le frontend et le backend.

Exemple de requête **MiningRequest** construite à partir du formulaire, envoyée au backend :

```
{
  “engine”: “auto”,
  "datasetPath": "datasets/contextPasquier99.dat",
  "query": "frequent"
  "params": { "minSupport": "0.2" }
}
```

Exemple de réponse **MiningResult** construite par le backend, renvoyée au frontend :

```
{
  "id": "59864991-8bcd-4ae3-b28e-91fde8093209",
  "status": "COMPLETED",
  "parameters": {
    "engine": "SPMF",
    "dataset": "/data/contextPasquier99.dat",
    "queryType": "frequent",
    "params": {
      "minSupport": "0.5"
    }
  },
  "result": [
    {
      "pattern": [1],
      "freq": 3
    },
    ...
    {
      "pattern": [2,3,5],
      "freq": 3
    }
  ],
  "error": null,
  "cancellationRequested": false
}
```

Concernant la gestion des erreurs et des retours utilisateur :

- Les erreurs de validation sont affichées via le composant Alert.
- Les erreurs réseau ou backend sont capturées et affichées à l'utilisateur.
- Les états de chargement, d'annulation et de complétion sont gérés dans l'état React et reflétés dans l'UI (boutons désactivés, messages, etc.).

2.2.3. Visualisation des résultats

Les résultats sont affichés dans le composant **ResultsTable.jsx** :

- **Tableau** : chaque motif (itemset) et son support sont affichés dans un tableau réactif.
- **Gestion du chargement** : un indicateur de chargement s'affiche pendant l'exécution de la tâche.
- **Reset** : possibilité de réinitialiser les résultats et le formulaire.

Pour des visualisations plus avancées (graphiques, heatmaps), le projet peut facilement intégrer des bibliothèques comme Recharts ou Chart.js dans de futurs développements.

Les messages d'alerte (succès, erreur, annulation) sont affichés en haut de l'interface via le composant Alert. Les erreurs inattendues côté React sont capturées par **ErrorBoundary** et affichées de façon conviviale.

2.3. Gestion asynchrone et utilisation du protocole "handshake"

2.3.1. Deux tâches principales

- **Tâche de statut (`/api/tasks/status`)**
  Elle sert à maintenir un dialogue fluide avec le frontend, répondant à ses requêtes régulières de mise à jour. Ce mécanisme garantit une interface réactive, même pendant les traitements intensifs.

- **Tâche de minage**
  Cette tâche effectue l'itemset mining de façon isolée. Une seule tâche de ce type est exécutée à un moment donné pour éviter les conflits et préserver les ressources.

Initialement, le backend fonctionnait de manière synchrone, ce qui bloquait complètement toute autre interaction avec le frontend lors d'une tâche de minage longue. En séparant les responsabilités sur deux tâches indépendantes, le backend reste disponible et réactif.

2.3.2. Protocole de synchronisation "handshake" pour la réception des résultats

Lorsque le frontend récupérait les résultats via un `POST`, le backend supprimait immédiatement les données associées à la tâche. Si le frontend ne "réussissait" pas à écouter à temps, les résultats étaient perdus.

Solution mise en place
Un protocole de type "handshake" a été introduit :

1. Le frontend récupère les résultats via un `GET`.

2. Il envoie ensuite un `POST` pour confirmer qu'il a bien reçu les résultats.
3. Ce n'est qu'après cette confirmation que le backend libère la mémoire et se prépare pour une future tâche.

Ce mécanisme garantit la fiabilité de la transmission des résultats et permet une nouvelle exécution dans des conditions sûres.

Bilan

Bien que l'application web offre une solution convaincante pour la fouille de motifs fréquents – avec neuf types de requêtes possibles via ChocoMining et SPMF, une sélection dynamique des librairies par un modèle de classification, un backend Java/Spring Boot intégrant les bibliothèques, un module Python/Flask pour la décision, un frontend React interagissant via API REST, et une gestion asynchrone assurée par un protocole de type "handshake" – plusieurs axes d'amélioration peuvent être envisagés:

- La sélection dynamique entre SPMF et ChocoMining repose actuellement sur un modèle de classification supervisée. Celui-ci gagnerait à être enrichi avec plus de caractéristiques descriptives des requêtes et des algorithmes d'apprentissage plus performants, afin d'affiner le choix de la bibliothèque selon le contexte.
- La gestion des résultats très volumineux (plus de 10 000 itemsets) pourrait être améliorée par un traitement par lots ou une pagination côté serveur. Cela limiterait les surcharges de mémoire, fluidifierait l'affichage côté frontend et sécuriserait la transmission des résultats, notamment en cas de connexions instables.
- Enfin, des optimisations internes de ChocoMining restent à explorer, en testant différentes stratégies de recherche selon les types de données et contraintes ainsi qu'une étude plus approfondie pour déterminer les algorithmes SPMF les plus efficaces en fonction des paramètres. En parallèle, une visualisation plus interactive (graphiques, filtres dynamiques) permettrait de mieux exploiter les résultats extraits et d'améliorer l'expérience utilisateur globale.