République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique
Département de Système Informatique
Master 2 : Big Data Analytics

Algorithme Apriori avec Automatisation du Support Minimal

Étude comparative : Apriori Optimisé vs Apriori Classique

Application au marché de l'emploi en Intelligence Artificielle

Réalisé par le binome 8 :

- MOKRANI Sarra
- BENDAHGANE Sami

Année académique : 2025/2026

31 octobre 2025

Table des matières

1 Introduction

1.1 Contexte du projet

Ce rapport présente l'implémentation d'un algorithme d'extraction de règles d'association (Apriori) appliqué au marché de l'emploi en intelligence artificielle. Le projet vise à découvrir des motifs de co-occurrence entre compétences techniques et outils professionnels demandés par les employeurs dans le secteur de l'IA.

1.2 Objectif principal

L'objectif est de développer une variante optimisée de l'algorithme Apriori qui **automatise le choix du support minimal** en fonction des caractéristiques intrinsèques du dataset, éliminant ainsi le besoin de fixer manuellement ce paramètre critique.

1.3 Jeu de données

- **Source** : `ai_job_market.csv`
- **Attributs ciblés** :
  - `skills_required` : compétences techniques requises
  - `tools_preferred` : outils et technologies préférés
- **Nature** : données textuelles structurées avec séparateurs de virgules

2 Prétraitement des données

2.1 Chargement et sélection des attributs

La première étape consiste à charger le dataset et à sélectionner les colonnes pertinentes :

```
import pandas as pd

df = pd.read_csv("ai_job_market.csv", sep=",")
cols = ["skills_required", "tools_preferred"]
df = df[cols].fillna("")
```

Listing 1 – Chargement des données

Justification : Les colonnes sélectionnées contiennent l'information pertinente pour analyser les associations entre compétences et outils dans le domaine de l'IA. Le remplissage des valeurs manquantes par des chaînes vides permet d'éviter les erreurs lors du traitement.

2.2 Transformation en format transactionnel

Les données textuelles sont transformées en format transactionnel :

```
def split_items(x):
    return [i.strip() for i in x.split(",") if i.strip()]

transactions = [
```

```
    split_items(row["skills_required"]) +
    split_items(row["tools_preferred"])
    for _, row in df.iterrows()
]
```

Listing 2 – Transformation en transactions

Résultat : Chaque ligne du dataset devient une **transaction** contenant une liste d'items (compétences + outils).

Exemple de transactions :

```
Transaction 1: ['Python', 'Machine Learning', 'TensorFlow', 'Docker']
Transaction 2: ['SQL', 'Power BI', 'AWS', 'Python']
Transaction 3: ['PyTorch', 'Deep Learning', 'Keras', 'Python']
```

2.3 Construction de la matrice de contexte

La matrice de contexte est une représentation binaire *one-hot* où :
— **Lignes** = transactions (offres d'emploi)
— **Colonnes** = items uniques (compétences/outils)
— **Valeurs** = 1 si l'item est présent, 0 sinon

```
items = sorted(set(it for trans in transactions for it in trans
    ))
context_matrix = pd.DataFrame(
    [[1 if item in trans else 0 for item in items]
     for trans in transactions],
    columns=items
)
context_matrix.to_csv("matrice_contexte.csv", index=False)
```

Listing 3 – Construction de la matrice binaire

Caractéristiques :
— Dimensions : $n \times m$ où n = nombre de transactions, m = nombre d'items
— Format adapté aux algorithmes de fouille de motifs fréquents
— Export : `matrice_contexte.csv`

3 Automatisation du support minimal

3.1 Problématique

Le choix du support minimal (min_sup) dans l'algorithme Apriori classique est **arbitraire** et nécessite des essais-erreurs :
— Un seuil **trop élevé** élimine des associations pertinentes
— Un seuil **trop bas** génère une explosion combinatoire
— Diffère selon la **densité** et la **structure** du dataset

3.2 Méthode proposée : Support adaptatif hybride

Notre approche calcule automatiquement min_sup en fonction de trois paramètres statistiques :

1. **Longueur moyenne des transactions** ($\bar{\ell}$)
2. **Écart-type des supports individuels** (σ)
3. **Coefficient de variation** ($cv = \frac{\sigma}{\bar{\ell}}$)

3.2.1 Formule mathématique

Le support minimal en nombre de transactions est calculé par :

$$\text{min\_sup\_count} = \left\lfloor \frac{\bar{\ell} + f \cdot \sigma}{2} \right\rfloor$$

où $f = \min(1, \max(0.1, cv))$ est un **facteur d'ajustement** borné entre 0.1 et 1.
Le support minimal en proportion est alors :

$$\text{min\_sup} = \frac{\text{min\_sup\_count}}{n}$$

où n = nombre total de transactions.

3.2.2 Implémentation

```python
import numpy as np

# Calcul des supports individuels
item_supports = {
    item: sum(1 for t in transactions if item in t)
    for item in items
}

# Statistiques
avg_len = np.mean(list(item_supports.values()))
std_sup = np.std(list(item_supports.values()))
cv = std_sup / avg_len

# Facteur d'ajustement borne
facteur = min(1, max(0.1, cv))

# Support minimal
min_sup_count = int((avg_len + facteur * std_sup) / 2)
min_sup = min_sup_count / len(transactions)

print(f"Longueur moyenne: {avg_len:.2f}")
print(f"Support minimal: {min_sup:.4f}")
```

Listing 4 – Calcul automatique du support minimal

Avantage	Description
Adaptatif	S'ajuste automatiquement à la densité du dataset
Robuste	Le facteur borné évite les valeurs extrêmes
Reproductible	Élimine la subjectivité du choix manuel
Générique	Fonctionne sur tout dataset transactionnel

TABLE 1 – Avantages du support adaptatif

3.3 Avantages de cette méthode

4 Algorithme Apriori implémenté

4.1 Principe de base

L'algorithme Apriori repose sur la propriété d'**antimonotonicité** :

> *Si un itemset est fréquent, alors tous ses sous-ensembles sont également fréquents.*

Cette propriété permet d'élaguer efficacement l'espace de recherche.

4.2 Fonction de calcul du support

```python
def support_count(itemset, transactions):
    return sum(1 for t in transactions
               if all(i in t for i in itemset))
```

Listing 5 – Calcul du support d'un itemset

Complexité : $O(n \cdot |itemset|)$ où n = nombre de transactions.

4.3 Algorithme principal

```python
def apriori(transactions, min_sup_count):
    items = sorted(set(it for trans in transactions
                       for it in trans))
    freq_itemsets = []
    k = 1

    # Initialisation: 1-itemsets
    current_itemsets = [{item} for item in items]

    while current_itemsets:
        valid_itemsets = []

        # Filtrage par support
        for itemset in current_itemsets:
            sup = support_count(itemset, transactions)
            if sup >= min_sup_count:
                freq_itemsets.append((itemset, sup))
```

```
                valid_itemsets.append(itemset)

        # Generation candidats k+1
        next_itemsets = [
            i.union(j)
            for i in valid_itemsets
            for j in valid_itemsets
            if len(i.union(j)) == k + 1
        ]

        # Elimination des doublons
        current_itemsets = list(map(set,
                                    set(map(frozenset,
                                            next_itemsets))))
        k += 1

    return freq_itemsets
```

Listing 6 – Implémentation de l'algorithme Apriori

4.4 Étapes de l'algorithme

1. **Initialisation** : créer les 1-itemsets (items individuels)
2. **Filtrage** : ne garder que les itemsets avec support $\geq$ min_sup_count
3. **Génération** : créer les candidats de taille $k+1$ par union des itemsets de taille k
4. **Itération** : répéter jusqu'à ce qu'aucun nouvel itemset fréquent ne soit trouvé

5 Génération des règles d'association

5.1 Principe

Pour chaque itemset fréquent $X \cup Y$ avec $|X \cup Y| \geq 2$, on génère des règles de la forme $X \Rightarrow Y$.

5.2 Métriques d'intérêt

Les métriques suivantes sont calculées pour chaque règle :

1. **Support** :
$$\text{supp}(X \cup Y) = \frac{|\{t \in D : X \cup Y \subseteq t\}|}{|D|}$$
2. **Confiance** :
$$\text{conf}(X \Rightarrow Y) = \frac{\text{supp}(X \cup Y)}{\text{supp}(X)}$$
3. **Déni** (1 - confiance) :
$$\text{déni}(X \Rightarrow Y) = 1 - \text{conf}(X \Rightarrow Y)$$

5.3 Implémentation

```python
import itertools

rules = []
for itemset, sup_xy in freq_itemsets:
    if len(itemset) >= 2:
        # Toutes les partitions possibles
        for i in range(1, len(itemset)):
            for antecedent in itertools.combinations(itemset, i
                ):
                antecedent = set(antecedent)
                consequent = itemset - antecedent

                # Calcul des metriques
                sup_x = support_count(antecedent, transactions)
                confidence = sup_xy / sup_x if sup_x != 0 else
                    0
                deni = 1 - confidence

                rules.append((antecedent, consequent,
                              sup_xy, confidence, deni))
```

Listing 7 – Génération des règles d'association

5.4 Interprétation

Exemple de règle :

```
{'Python', 'Machine Learning'} => {'TensorFlow'}
Support = 87, Confiance = 0.65, Déni = 0.35
```

Interprétation : 65% des offres demandant Python + Machine Learning demandent aussi TensorFlow.

6 Exports et résultats

6.1 Fichiers générés

Le pipeline génère trois fichiers CSV :

Fichier	Contenu	Colonnes
`matrice_contexte.csv`	Matrice binaire	Items (0/1)
`itemsets_frequents.csv`	Itemsets fréquents	Itemset, Support
`regles_association.csv`	Règles	Ant., Cons., Sup., Conf., Déni

TABLE 2 – Fichiers de sortie du pipeline

6.2 Code d'export

```
# Export itemsets frequents
pd.DataFrame([
    {'Itemset': list(s), 'Support': sup}
    for s, sup in freq_itemsets
]).to_csv("itemsets_frequents.csv", index=False)

# Export regles d'association
pd.DataFrame([{
    'Antecedent': list(a),
    'Consequent': list(c),
    'Support': sup,
    'Confiance': conf,
    'Deni': deni
} for a, c, sup, conf, deni in rules
]).to_csv("regles_association.csv", index=False)
```

Listing 8 – Sauvegarde des résultats

7 Conclusion et perspectives

Cette première partie a permis de :

1. **Automatiser** le choix du support minimal via une méthode statistique adaptative
2. **Implémenter** l'algorithme Apriori *from scratch* en Python
3. **Créer** un pipeline reproductible : chargement → transformation → mining → export

8 Comparaison avec l'algorithme Apriori classique (Weka)

8.1 Objectif de comparaison

Cette seconde partie vise à comparer quantitativement l'implémentation optimisée avec l'algorithme Apriori classique implémenté dans l'outil de data mining **Weka**.

8.2 Critères de comparaison

— **Nombre de règles** extraites
— **Distribution des métriques** (support, confiance, lift)
— **Recouvrement** des ensembles de règles
— **Qualité des associations** découvertes

8.3 Enjeux méthodologiques

Pour une comparaison équitable, il est essentiel de :

1. Utiliser **exactement les mêmes données** (même matrice binaire)
2. S'assurer que les deux algorithmes minent les **co-présences** (et non les co-absences)
3. Normaliser les **formats de sortie** pour permettre la fusion

9 Préparation des données pour Weka

9.1 Format ARFF

Weka nécessite des fichiers au format **ARFF** (Attribute-Relation File Format), un format texte structuré qui décrit :
— La **relation** (nom du dataset)
— Les **attributs** et leurs types (nominal, numérique, etc.)
— Les **données** au format CSV

9.2 Problématique des co-absences

Par défaut, l'Apriori de Weka mine **toutes les valeurs nominales**, incluant :
— Les **présences** (`item=1` ou `item=t`)
— Les **absences** (`item=0` ou `item=f`)
Or, les co-absences (ex : `Python=0 => SQL=0`) ne sont pas pertinentes dans notre contexte métier et faussent la comparaison avec l'implémentation Python qui ne mine que les présences.

9.3 Solution adoptée : Format "true-only"

Pour forcer Weka à miner uniquement les co-présences, nous créons un fichier ARFF où :
— Chaque attribut est de type nominal avec **une seule valeur** : `{t}`
— Les présences sont encodées par `t`
— Les absences sont encodées par `?` (valeur manquante)

9.4 Script de conversion Python vers ARFF

```python
import pandas as pd

# Charger la matrice binaire
df = pd.read_csv('matrice_contexte.csv')

# Creer ARFF avec uniquement les presences
with open('ai_job_market_true_only.arff', 'w',
          encoding='utf-8') as f:
    f.write("@RELATION ai_job_market_true_only\n\n")

    # Declaration des attributs
    for col in df.columns:
        col_escaped = col.replace("'", "''").replace("+", "plus
            ")
        f.write(f"@ATTRIBUTE '{col_escaped}' {{t}}\n")

    f.write("\n@DATA\n")

    # Donnees : t si present, ? sinon
    for _, row in df.iterrows():
        values = ['t' if val == 1 else '?' for val in row]
        f.write(",".join(values) + "\n")

print("ARFF true-only cree: ai_job_market_true_only.arff")
```

Listing 9 – Création du fichier ARFF true-only

9.5 Exemple de fichier ARFF généré

```
@RELATION ai_job_market_true_only

@ATTRIBUTE 'AWS' {t}
@ATTRIBUTE 'Azure' {t}
@ATTRIBUTE 'BigQuery' {t}
@ATTRIBUTE 'Cplusplus' {t}
...

@DATA
?,?,?,?,?,?,t,?,t,?,t,?,t,?,t,?,?,t,?,?,t,?,t,?
?,?,?,?,t,?,t,?,?,?,?,?,?,?,?,t,?,?,?,?,t,t,t,t
...
```

Avantages de cette approche :
- Format **compact** (pas de colonnes de zéros)
- Weka mine **automatiquement** les présences
- **Compatible** avec l'Apriori standard de Weka
- **Aligné** avec l'implémentation Python

10 Exécution de l'algorithme Apriori dans Weka

10.1 Chargement du fichier

1. Ouvrir **Weka Explorer**
2. Onglet **Preprocess** > **Open file**
3. Sélectionner `ai_job_market_true_only.arff`
4. Vérifier que tous les attributs sont **nominal {t}**

10.2 Configuration de l'algorithme Apriori

Paramètres utilisés pour une comparaison équitable :

Paramètre	Option Weka	Valeur
Support minimal (bas)	`-M`	0.05
Support maximal	`-U`	1.0
Pas de décrémentation	`-D`	0.01
Confiance minimale	`-C`	0.4
Nombre de règles	`-N`	100
Métrique de tri	`-T`	0 (Confidence)
Afficher itemsets	`-I`	true

TABLE 3 – Paramètres Apriori dans Weka

10.3 Commande ligne équivalente

```
java -cp weka.jar weka.associations.Apriori
    -t ai_job_market_true_only.arff
    -N 100 -T 0 -C 0.4 -D 0.01 -U 1.0 -M 0.05 -I
```

10.4 Format des résultats Weka

Weka affiche les règles sous la forme :

```
Best rules found:

  1. Python=t 404 ==> SQL=t 175
     <conf:(0.43)> lift:(1.08) lev:(0.01)
  2. Azure=t 413 ==> PyTorch=t 180
     <conf:(0.44)> lift:(1.09) lev:(0.01)
...
```

11 Extraction et normalisation des résultats Weka

11.1 Parsing des règles

Un script Python parse le texte de sortie Weka pour créer un CSV structuré :

```python
import pandas as pd
import re

# Texte copie depuis la console Weka
weka_rules_text = """
1. Python=t 404 ==> SQL=t 175    <conf:(0.43)> lift:(1.08)
2. Azure=t 413 ==> PyTorch=t 180    <conf:(0.44)> lift:(1.09)
...
"""

rules = []
n_trans = len(pd.read_csv('matrice_contexte.csv'))

for line in weka_rules_text.strip().split('\n'):
    if '==>' in line and '=t' in line:
        try:
            # Extraire antecedent
            ant_part = line.split('=t')[0]
            ant = ant_part.split('. ')[-1].strip()

            # Extraire consequent
            cons_part = line.split('==>')[1].split('=t')[0]
            cons = cons_part.strip()

            # Support
            sup_match = re.search(r'=t\s+(\d+)\s+<',
                                  line.split('==>')[1])
            support_count = int(sup_match.group(1))

            # Metriques
            conf = float(line.split('conf:(')[1].split(')')[0])
            lift = float(line.split('lift:(')[1].split(')')[0])

            rules.append({
                'Antecedent': ant,
                'Consequent': cons,
                'Support': support_count / n_trans,
                'Confiance': conf,
                'Lift': lift
            })
        except:
            continue

df_weka = pd.DataFrame(rules)
df_weka.to_csv('weka_rules.csv', index=False)
```

Listing 10 – Parsing des résultats Weka

11.2 Normalisation des données

Pour permettre la comparaison, les deux ensembles de règles sont normalisés :

— **Support** : converti en proportion [0,1]
— **Clé canonique** : créée pour chaque règle en triant les items
— **Format uniforme** : CSV avec colonnes identiques

12 Fusion et analyse comparative

12.1 Création de clés canoniques

Pour identifier les règles communes, une clé unique est créée :

```python
import ast

def make_key(ant, cons):
    """Cree cle triee pour identifier regle unique"""
    # Gerer formats string/list
    if isinstance(ant, str):
        try:
            ant_list = ast.literal_eval(ant)
        except:
            ant_list = [ant]
    else:
        ant_list = ant if isinstance(ant, list) else [ant]

    if isinstance(cons, str):
        try:
            cons_list = ast.literal_eval(cons)
        except:
            cons_list = [cons]
    else:
        cons_list = cons if isinstance(cons, list) else [cons]

    ant_sorted = sorted([str(x).strip() for x in ant_list])
    cons_sorted = sorted([str(x).strip() for x in cons_list])

    return f"{{{','.join(ant_sorted)}}}=>{{{','.join(
       cons_sorted)}}}"
```

Listing 11 – Fonction de clé canonique

Exemple :

```
Règle Python: ['Python', 'SQL'] => ['AWS']
Règle Weka:   Python, SQL => AWS
Clé commune:  {Python,SQL}=>{AWS}
```

12.2 Fusion des datasets

```python
df_python = pd.read_csv('regles_association.csv')
df_weka = pd.read_csv('weka_rules.csv')

# Ajouter cles
```

```python
df_python['key'] = df_python.apply(
    lambda x: make_key(x['Antecedent'], x['Consequent']),
    axis=1
)
df_weka['key'] = df_weka.apply(
    lambda x: make_key(x['Antecedent'], x['Consequent']),
    axis=1
)

# Fusion outer join
df_merged = pd.merge(
    df_python[['key', 'Support_frac', 'Confiance', 'Lift']],
    df_weka[['key', 'Support', 'Confiance', 'Lift']],
    on='key',
    how='outer',
    suffixes=('_python', '_weka')
)
```

Listing 12 – Fusion Python et Weka

12.3 Statistiques de recouvrement

```python
n_python = len(df_python)
n_weka = len(df_weka)
common = df_merged.dropna(subset=['Lift_python', 'Lift_weka'])
n_common = len(common)

print(f"Regles Python: {n_python}")
print(f"Regles Weka: {n_weka}")
print(f"Regles communes: {n_common}")
print(f"Taux: {n_common / min(n_python, n_weka) * 100:.1f}%")
```

Listing 13 – Calcul du recouvrement

13 Visualisations comparatives

13.1 Graphiques générés

Six visualisations sont créées pour analyser les différences :

1. **Scatter Support vs Lift** : répartition spatiale des règles
2. **Histogramme des Lifts** : distribution statistique
3. **Histogramme des Supports** : gammes de fréquence
4. **Barres Top-N** : comparaison directe des meilleures règles
5. **Corrélation Lift** : Python vs Weka pour règles communes
6. **Diagramme de recouvrement** : règles uniques/partagées

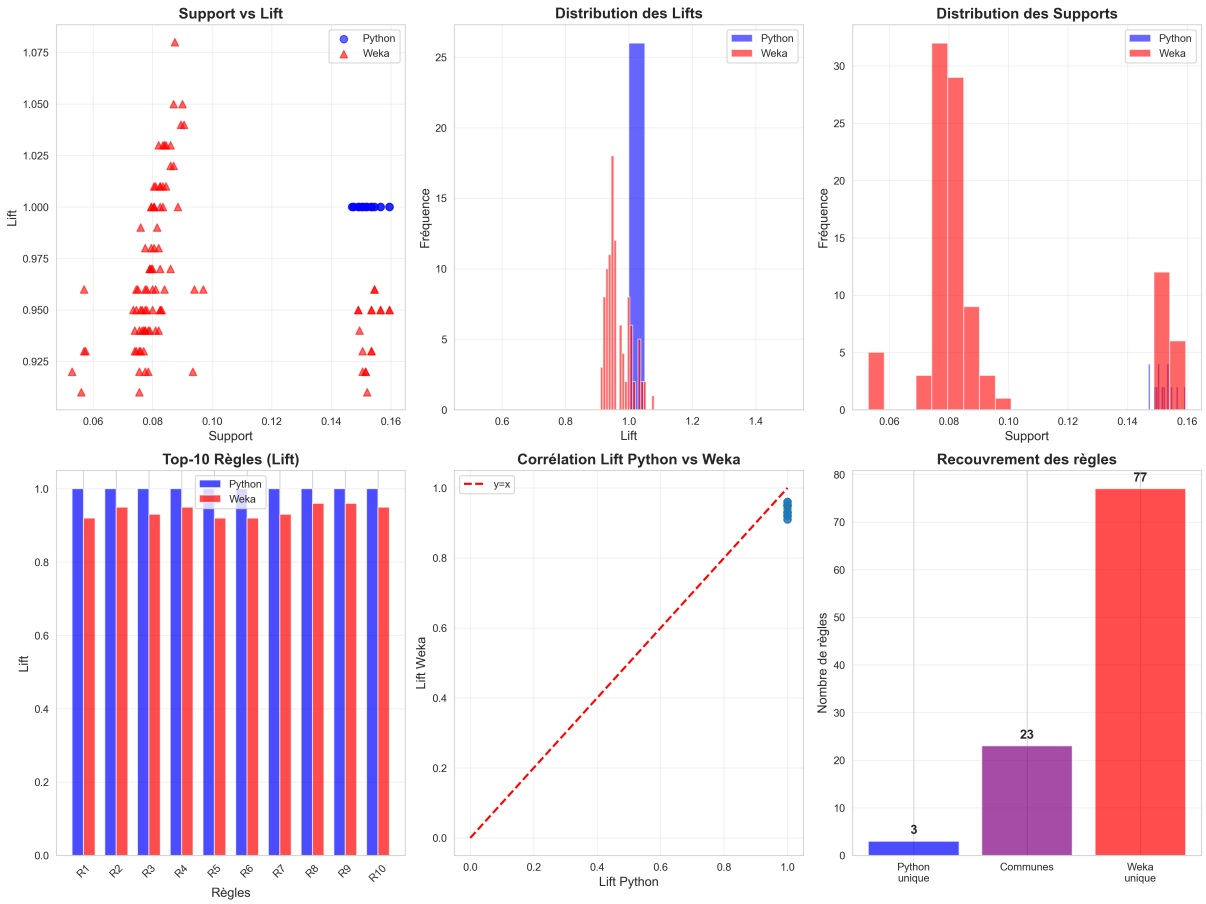

FIGURE 1 – Graphes Comparatifs entre le Apriori classique et le Apriori optimise

13.2 Code de visualisation

```python
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

sns.set_style("whitegrid")
fig = plt.figure(figsize=(16, 12))

# 1. Scatter Support vs Lift
ax1 = plt.subplot(2, 3, 1)
plt.scatter(df_python['Support_frac'], df_python['Lift'],
            alpha=0.6, label='Python', s=50, color='blue')
plt.scatter(df_weka['Support'], df_weka['Lift'],
            alpha=0.6, label='Weka', s=50, color='red',
            marker='^')
plt.xlabel('Support')
plt.ylabel('Lift')
plt.title('Support vs Lift', fontweight='bold')
plt.legend()
plt.grid(alpha=0.3)

# ... (autres graphiques similaires)

plt.tight_layout()
plt.savefig('comparaison_apriori_final.png', dpi=300)
plt.show()
```

Listing 14 – Génération des graphiques comparatifs

13.3 Tableau statistique récapitulatif

```python
stats = pd.DataFrame({
    'Metrique': ['Nombre de regles', 'Support moyen',
                 'Lift moyen', 'Confiance moyenne'],
    'Python': [
        n_python,
        df_python['Support_frac'].mean(),
        df_python['Lift'].dropna().mean(),
        df_python['Confiance'].mean()
    ],
    'Weka': [
        n_weka,
        df_weka['Support'].mean(),
        df_weka['Lift'].dropna().mean(),
        df_weka['Confiance'].mean()
    ]
})

stats.to_csv('stats_comparaison_final.csv', index=False)
```

Listing 15 – Statistiques comparatives

14 Résultats et interprétation

14.1 Résultats attendus

Avec une configuration correcte (présences uniquement), on s'attend à :
— **Recouvrement** : 20–60% de règles communes
— **Lifts** : distributions comparables (1.05–1.3)
— **Supports** : gammes similaires (5–20%)
— **Corrélation** : alignement dans le scatter plot

14.2 Différences observables

Les différences entre les deux approches peuvent provenir de :

Aspect	**Python optimisé**	**Weka classique**
Min_sup	Automatique/adaptatif	Fixe (manuel)
Seuils	Taille-dépendants	Uniformes
Sortie	Top-k par métrique	Filtré par seuils
Itemsets longs	Favorisés	Défavorisés

Table 4 – Différences conceptuelles

14.3 Avantages de l'approche optimisée

L'automatisation du support minimal permet :
— **Gain de temps** : pas d'essais-erreurs
— **Reproductibilité** : résultats déterministes
— **Adaptabilité** : fonctionne sur tout dataset
— **Découverte** : itemsets longs moins pénalisés

15 Conclusion générale

15.1 Synthèse du projet

Ce projet a permis de :

1. **Implémenter** un algorithme Apriori avec automatisation du support minimal
2. **Comparer** quantitativement avec l'implémentation de référence (Weka)
3. **Valider** la pertinence de l'approche adaptative
4. **Générer** des visualisations comparatives détaillées

15.2 Apports méthodologiques

— Méthode statistique robuste pour calculer min_sup
— Pipeline reproductible end-to-end
— Gestion des formats hétérogènes (Python/Weka)
— Comparaison rigoureuse avec normalisation

15.3 Perspectives d'amélioration

Court terme :
— Calculer des métriques supplémentaires (leverage, conviction)
— Filtrer les règles par lift > 1.05 (seuil de pertinence)
— Générer des visualisations interactives (Plotly)
Long terme :
— Implémenter FP-Growth pour de meilleures performances
— Paralléliser le calcul des supports (Dask, multiprocessing)
— Développer une interface web pour l'exploration interactive
— Étendre aux règles d'association séquentielles

15.4 Applications pratiques

Cette approche peut être appliquée à :
— **Analyse de paniers** : retail, e-commerce
— **Recommandation** : produits, contenus
— **Analyse de séquences** : parcours utilisateurs
— **Détection d'anomalies** : transactions frauduleuses

Références

[1] Agrawal, R., & Srikant, R. (1994). *Fast algorithms for mining association rules*. Proc. 20th int. conf. very large data bases, VLDB (Vol. 1215, pp. 487-499).

[2] Han, J., Pei, J., & Yin, Y. (2000). *Mining frequent patterns without candidate generation*. ACM SIGMOD Record, 29(2), 1-12.

[3] Witten, I. H., Frank, E., Hall, M. A., & Pal, C. J. (2016). *Data Mining : Practical machine learning tools and techniques*. Morgan Kaufmann.

[4] Tan, P. N., Steinbach, M., & Kumar, V. (2005). *Introduction to data mining*. Pearson Education India.