Classification Association Rules under WEKA

UNIVERSITÉ MOHAMMED V

Faculté des Sciences Rabat

TOUIJER OUSSAMA
Master (IAO)
Faculté des Sciences Rabats
Rabat, MAROC
oussama.touijer@um5r.ac.ma

ASSAL ANASS
Master (IAO)
Faculté des Sciences Rabat
Rabat, MAROC
anass.assal@um5r.ac.ma

***Résumé*—Cet article explore l'utilisation des règles d'association pour la classification (Classification Association Rules - CAR) en fouille de données en implémentant l'algorithme Apriori dans WEKA. L'étude se concentre sur l'extraction de règles d'association à partir d'un ensemble de transactions et leur utilisation pour la classification.**

Les performances de ce classificateur basé sur les règles sont comparées à celles d'un classificateur standard K-Nearest Neighbors (KNN). Les résultats expérimentaux sur le jeu de données supermarket.

arff montrent que le classificateur basé sur CAR surpasse KNN en termes d'exactitude (79,95%) et de F1-score (72,90%), démontrant ainsi l'efficacité de l'approche basée sur les règles d'association.

***Index Terms*—Règles d'association, Classification Association Rules, Apriori, Fouille de données, WEKA, K-Nearest Neighbors,**

I. Introduction

L'extraction de règles d'association est une technique essentielle en fouille de données, permettant d'identifier des relations fréquentes entre différents attributs dans un ensemble de transactions. Cette approche est largement utilisée dans des domaines tels que le commerce, la santé et la finance pour analyser les comportements et faire des prédictions.

Dans le cadre de la Classification Association Rules (CAR), les règles d'association ne se limitent pas à la découverte de relations entre les éléments, mais sont également exploitées pour prédire une classe cible. L'algorithme Apriori, implémenté dans WEKA, est un outil puissant pour générer ces règles et construire un classificateur basé sur leur pertinence.

Document rédigé dans le cadre d'un projet finale de module BI & DM. Au terme de ce travail, nous tenons à exprimer notre profonde gratitude et nos sincères remerciements à votre encadrant, le Professeur **BEN KHALIFA Mohammed**.

L'objectif de cette étude est de comparer l'efficacité d'un classificateur basé sur les règles d'association avec un classificateur standard K-Nearest Neighbors (KNN). Pour cela, nous avons appliqué ces deux approches sur le jeu de données supermarket.arff, qui contient des transactions d'achats en supermarché. Nous analysons les performances des deux modèles en utilisant des métriques standard telles que l'exactitude, la précision, le rappel et le F1-score.

Les résultats obtenus montrent que l'approche basée sur les règles d'association surpasse KNN en termes de performance, mettant en évidence la pertinence de cette méthode dans le cadre de la classification.

II. Algorithme Apriori pour l'extraction des règles d'association et classification

L'algorithme Apriori est une méthode d'exploration de données utilisée pour extraire des règles d'association à partir d'un ensemble de transactions. Il est souvent appliqué à des bases de données transactionnelles pour découvrir des relations intéressantes entre les éléments. Dans le cadre de la Classification Association Rules (CAR), l'algorithme est utilisé pour générer des règles qui prédisent une classe spécifique.

A. Algorithme Apriori pour la Classification Association Rules (CAR)

1) Principe de fonctionnement: L'algorithme Apriori repose sur le concept de fréquence des items et utilise une approche itérative basée sur le principe "un sous-ensemble fréquent d'un itemset doit être fréquent".

- **Génération des itemsets fréquents :**
  - On commence par identifier les éléments fréquents dans la base de données en fonction d'un seuil minimum de support.
  - Ensuite, on combine ces éléments pour former des itemsets de plus grande taille.
  - Ce processus continue jusqu'à ce qu'il n'y ait plus d'itemsets fréquents.

— **Génération des règles d'association :**
  — À partir des itemsets fréquents, des règles de la forme $X \rightarrow Y$ sont générées.
  — Ces règles doivent respecter un seuil minimum de confiance.
— **Pruning et sélection des règles les plus pertinentes :**
  — Seules les règles ayant un bon support, confiance et lift sont conservées.
  — Une technique supplémentaire appelée "Rule Ordering" est appliquée pour classer les règles en fonction de leur pertinence.
— **Utilisation des règles pour la classification :**
  — Lorsqu'un nouvel exemple est présenté, il est classé en fonction des règles trouvées.
  — Si plusieurs règles s'appliquent, une stratégie de vote peut être utilisée.

2) Paramètres d'Apriori dans WEKA: L'implémentation d'Apriori dans WEKA propose plusieurs paramètres réglables pour influencer la qualité des règles générées :

— **MinSupport (-N) :**
  — Définit le seuil minimum de support pour qu'un itemset soit considéré comme fréquent.
  — Exemple : MinSupport = 0.1 signifie que l'itemset doit apparaître dans au moins 10% des transactions.
— **MinConfidence (-C) :**
  — Seuil de confiance pour la génération des règles.
  — Exemple : MinConfidence = 0.6 signifie que les règles doivent avoir une probabilité d'au moins 60% d'être correctes.
— **UpperBoundMinSupport (-U) :**
  — Fixe une limite supérieure pour le support minimum.
  — Cela permet d'éviter d'avoir trop de règles.
— **LowerBoundMinSupport (-L) :**
  — Fixe une limite inférieure du support minimum.
— **Metric Type (-M) :**
  — Permet de choisir la métrique d'évaluation des règles (Lift, Leverage, etc.).
— **NumRules (-N) :**
  — Définit le nombre maximal de règles à générer.
— **Pruning (-P) :**
  — Permet d'éliminer les règles peu pertinentes en fonction des critères de support et de confiance.

3) Exemple d'implémentation dans WEKA:

— Importer le dataset dans WEKA.
— Sélectionner l'onglet "Associate".
— Choisir l'algorithme "Apriori".
— Configurer les paramètres :
  — MinSupport : 0.1
  — MinConfidence : 0.6
  — Nombre max de règles : 10
— Lancer l'algorithme et examiner les règles générées.

III. Algorithme K-Nearest Neighbors (KNN)

L'algorithme KNN (K-Nearest Neighbors) est un algorithme de classification basé sur la proximité entre les instances.

A. Principe de fonctionnement

KNN est un algorithme non paramétrique basé sur la notion de distance :

— **Définition du voisinage :**
  — Lorsqu'un nouvel échantillon doit être classé, l'algorithme recherche ses K voisins les plus proches dans l'ensemble d'apprentissage.
— **Calcul de la distance :**
  — La distance entre les points est calculée selon différentes métriques :
    — **Distance Euclidienne :**
$$d(A,B) = \sqrt{\sum (A_i - B_i)^2}$$
    — **Distance de Manhattan :**
$$d(A,B) = \sum |A_i - B_i|$$
    — **Distance de Minkowski :** (généralisation des deux précédentes).
— **Vote majoritaire :**
  — La classe est attribuée en fonction de la majorité des classes des voisins.

B. Paramètres de KNN dans WEKA

Dans WEKA, KNN est implémenté sous le nom IBk. Voici les principaux paramètres configurables :

— **Nombre de voisins (K) :**
  — Définit combien de voisins sont considérés pour la classification.
  — Valeurs typiques : K = 3 ou K = 5.
— **Type de distance :**
  — Euclidean (par défaut)
  — Manhattan (utile pour des données avec des distributions différentes).
— **Pondération des voisins :**
  — Uniform (vote simple)
  — Inverse Distance Weighting (les plus proches ont plus d'influence).
— **Stratégie de gestion des valeurs manquantes :**
  — Ignore
  — Remplacement par la moyenne ou médiane.

C. Exemple d'implémentation dans WEKA

— Importer le dataset dans WEKA.
— Aller dans l'onglet "Classify".
— Choisir l'algorithme "IBk" (KNN).
— Configurer les paramètres :
  — K = 5
  — Distance = Euclidean
  — Weighting = None
— Lancer la classification et analyser les performances.

IV. Utilisation de l'algorithme Apriori pour l'extraction des règles d'association et la classification

A. Comment utiliser l'algorithme Apriori

L'algorithme Apriori est un algorithme de fouille de données utilisé pour découvrir des relations entre des variables dans une base de données transactionnelle. Il est utilisé pour générer des règles d'association, qui peuvent ensuite être exploitées pour construire un classificateur.

1) Étapes principales d'Apriori:

- **Génération des itemsets fréquents :** Apriori parcourt les transactions pour trouver les ensembles d'items fréquents en fonction d'un seuil minimum de support.
- **Génération des règles d'association :** À partir des itemsets fréquents, Apriori génère des règles ayant une confiance minimale requise.
- **Sélection des meilleures règles :** Les règles sont triées en fonction de leur support, confiance et lift pour garder les plus pertinentes.
- **Utilisation des règles pour la classification :** Les règles sélectionnées sont utilisées pour classer de nouvelles instances.

B. Définition des paramètres d'Apriori

Dans WEKA, Apriori est configuré avec plusieurs paramètres. Voici la signification des paramètres utilisés dans cette expérience :

```
Apriori -N 30 -T 0.9 -D 0.05 -U 1.0 -M 0.1
        -1.0 -A -c -1
```

1) Explication des paramètres:

- **-N 30 :** Nombre maximal de règles à générer (30 règles).
- **-T 0.9 :** Seuil de confiance minimal (90%).
- **-D 0.05 :** Diminution du support minimum à chaque itération (5%).
- **-U 1.0 :** Valeur maximale du support (100%).
- **-M 0.1 :** Support minimal pour qu'un itemset soit considéré comme fréquent (10%).
- **-S -1.0 :** Définit la stratégie de support (valeur négative pour auto-ajustement).
- **-A :** Active la génération de règles de classification.
- **-c -1 :** Spécifie que la dernière colonne du dataset est la variable cible (classe à prédire).

C. Sélection des meilleures règles d'association

Les règles générées doivent être filtrées pour garder uniquement les plus pertinentes. Les critères de sélection sont :

- **Support élevé :** La règle doit apparaître fréquemment dans les transactions.
- **Confiance élevée :** La probabilité que la conclusion de la règle soit vraie lorsqu'on observe la condition doit être haute ($\geq 90\%$).
- **Lift élevé :** Mesure la pertinence de la règle par rapport à une distribution aléatoire.

1) Exemples de règles sélectionnées:

```
1. bread and cake=t, frozen foods=t,
pet foods=t, tissues-paper prd=t,
cheese=t  ==> total=high (conf: 0.91)

2. biscuits=t, frozen foods=t,
pet foods=t,
tissues-paper prd=t,
margarine=t  ==> total=high (conf: 0.91)
```

Ces règles indiquent que certains groupes de produits sont fortement corrélés avec un total d'achat élevé.

V. Génération d'un classificateur basé sur les règles

Un classificateur à règles est un modèle de classification qui prend des décisions en appliquant des règles d'association dérivées de l'algorithme Apriori. Il compare une nouvelle instance avec un ensemble de règles préétablies pour attribuer une classe.

A. Définition des Règles

Les règles sont définies sous la forme :

```
{
    'conditions': {'attribut1': 'valeur',
    'attribut2': 'valeur', ...},
    'class': 'high'
}
```

Elles représentent des relations fréquentes trouvées dans le dataset. Si un client achète certains articles ensemble, la règle peut prédire que son total d'achat sera élevé (high).

```
{
    'conditions': {'bread and cake': 't',
    'frozen foods': 't',
    'pet foods': 't',
    'tissues-paper prd': 't',
    'cheese': 't'},
    'class': 'high'
}
```

Interprétation : Si un client achète du pain, des aliments surgelés, des aliments pour animaux, du papier hygiénique et du fromage, alors son total d'achat sera élevé.

B. Création du Classificateur

La classe `RuleClassifier` est définie comme suit :

```
class RuleClassifier:
"Classificateur a regles avec suivi d'utilisation"

    def __init__(self, rules: List[Dict]):
        self.rules = rules
        self.rule_usage = {str(r['conditions']):
        0 for r in rules}
```

— `self.rules` : Stocke les règles d'association sélectionnées.
— `self.rule_usage` : Suivi de l'utilisation des règles pour voir lesquelles sont les plus utilisées.

C. Prédiction avec le Classificateur

La fonction `predict` applique les règles à une instance d'entrée :

```
def predict(self, instance: Dict) -> str:
"Effectue une prediction avec suivi des regles"

for rule in self.rules:
 if all(instance.get(k) == v for k, v in rule['
    conditions'].items()):
       self.rule_usage[str(rule['
       conditions'])] += 1

         return rule['class']
    return 'low'
```

Explication :

— Elle parcourt toutes les règles une par une.
— Elle vérifie si toutes les conditions d'une règle sont satisfaites par l'instance.
— Si une règle correspond, elle retourne la classe associée (ex. high).
— Sinon, la classe par défaut est low (faible total d'achat).

D. Chargement des Données

On charge les données à partir d'un fichier CSV :

```
def load_dataset(file_path: str) ->
    List[Dict[str, str]]:
    """Charge et nettoie le dataset"""
    try:
        with open(file_path, 'r',
        encoding='utf-8') as f:
            return [{k.strip(" '\""):
            v.strip()
            for k, v in row.items()} for
            row in csv.DictReader(f)]
    except FileNotFoundError:
        print(f"Erreur: Fichier
        '{file_path}' introuvable!")
        exit()
```

Cette fonction :

— Lit le fichier CSV.
— Nettoie les données en enlevant les espaces et caractères superflus.
— Retourne une liste de dictionnaires.

E. Séparation des Données en Entraînement et Test

On divise les données en données d'entraînement et données de test :

```
def split_data(data: List[Dict],
  test_size: float = 0.3) ->
    Tuple[List, List]:
"Divise les donn es de mani re al atoire"
    shuffled = data.copy()
    random.shuffle(shuffled)
    split_idx = int(len(shuffled) *
    (1 - test_size))

    return shuffled[:split_idx],
    shuffled[split_idx:]
```

Explication :

— Mélange les données aléatoirement.
— Sépare 70% des données pour l'entraînement et 30% pour le test.

F. Validation Croisée

On utilise la validation croisée k-fold pour tester le modèle plusieurs fois :

```
def cross_validate(data: List[Dict],
classifier:

RuleClassifier, k: int = 10) -> Dict:
    """Validation crois e k-fold"""
    data_copy = data.copy()
    random.shuffle(data_copy)
    fold_size = len(data_copy) // k
    all_preds, all_actuals = [], []

    for i in range(k):
        test_start = i * fold_size
        test_end = (i + 1) * fold_size
        if i < k - 1 else len(data_copy)

        test_set = data_copy[test_start:
        test_end]

        preds = [classifier.predict(x)
        for x in test_set]
        actuals = [x['total']
        for x in test_set]

        all_preds.extend(preds)
        all_actuals.extend(actuals)

    return calculate_metrics(all_preds,
    all_actuals)
```

Explication :

— On divise les données en 10 groupes (k=10).
— À chaque tour, un groupe est utilisé pour tester et les 9 autres pour entraîner.
— On répète 10 fois et on moyenne les résultats.

G. Évaluation du Modèle

On calcule la performance avec des métriques standards :

```
def calculate_metrics(predictions:
  List[str],
actuals: List[str]) -> Dict:
 "Calcule les m triques de performance"
    cm = {'TP': 0, 'FP': 0, 'TN': 0,
```

```
        'FN': 0}

for pred, true in zip(predictions,
    actuals):

 if pred == 'high' and true ==
 'high': cm['TP'] += 1

 elif pred == 'high' and true ==
 'low': cm['FP'] += 1

 elif pred == 'low' and true ==
 'low': cm['TN'] += 1

 else: cm['FN'] += 1

    total = len(predictions)
    precision = cm['TP'] / (cm['TP'] + cm['FP'])
    if (cm['TP'] + cm['FP']) > 0 else 0
    recall = cm['TP'] / (cm['TP'] + cm['FN'])
    if (cm['TP'] + cm['FN']) > 0 else 0

    return {

        'accuracy': (cm['TP'] + cm['TN']) /
                    total * 100,

        'precision': precision * 100,

        'recall': recall * 100,

        'f1': 2 * (precision * recall) /
        (precision + recall) * 100
        if (precision + recall) > 0 else 0,

        'confusion_matrix': cm,

        'total': total
    }
```

Explication :

- **Vrai Positif (TP) :** Le modèle prédit high et c'est correct.
- **Faux Positif (FP) :** Le modèle prédit high mais c'est faux.
- **Vrai Négatif (TN) :** Le modèle prédit low et c'est correct.
- **Faux Négatif (FN) :** Le modèle prédit low mais c'est faux.

On calcule :

- **Précision (precision) :** Corrects parmi les prédictions positives.
- **Rappel (recall) :** Corrects parmi les vrais cas positifs.
- **F1-score :** Moyenne harmonique entre précision et rappel.
- **Accuracy :** Taux de bonnes prédictions.

H. Étapes du classificateur

Le classificateur fonctionne en 3 étapes :

- Génération des règles (Apriori).
- Prédiction d'une classe (high ou low) en testant les règles.
- Évaluation des performances avec des métriques standards.

VI. Utilisation de l'algorithme K-Nearest Neighbors (KNN) pour la classification dans WEKA

L'algorithme KNN (K-Nearest Neighbors) est un classificateur basé sur la proximité des exemples dans l'espace des caractéristiques. Voici une explication détaillée pour l'utiliser, définir ses paramètres et classifier des instances.

A. Comment utiliser l'algorithme KNN (IBk) dans WEKA ?

Dans WEKA, IBk est l'implémentation du KNN. Voici les étapes pour l'utiliser :

- Ouvrir WEKA et aller dans l'onglet **Classify**.
- Charger le dataset en cliquant sur **Open file** et en sélectionnant un fichier `.arff` ou un fichier compatible.
- Sélectionner le classificateur :
  - Cliquer sur **Choose ¿ Lazy ¿ IBk**.
- Configurer les paramètres (voir section suivante).
- Définir l'option de test :
  - **Use training set** : Utilise les mêmes données pour l'entraînement et le test.
  - **Supplied test set** : Charge un autre fichier comme jeu de test.
  - **Cross-validation** (ex : 18 folds) : Divise les données en 18 sousensembles et entraîne le modèle sur 17 tout en testant sur le 18eme (et ainsi de suite).
  - **Percentage split** (ex : 40%) : Utilise 40% des données pour le test et le reste pour l'entraînement.
- Lancer la classification en cliquant sur **Start**.
- Analyser les résultats :
  - Précision, rappel, matrice de confusion...

B. Comment définir les paramètres de l'algorithme ?

Voici une explication des paramètres de IBk utilisés dans votre configuration :

- **K-nearest neighbours classifier (KNN = 7)** : L'algorithme recherche les 7 voisins les plus proches pour classer une nouvelle instance.
- **batchSize = 100** : Nombre d'instances traitées à la fois.
- **crossValidate = False** : Désactive la validation croisée automatique.
- **distanceWeighting = Weight by 1/distance** : Attribue un poids aux voisins en fonction de l'inverse de la distance.
- **nearestNeighbourSearchAlgorithm = LinearNNSearch** : Utilise la recherche linéaire pour trouver les voisins.
- **distance metric = EuclideanDistance -R first-last** : Utilise la distance euclidienne pour mesurer la proximité.

— **windowSize = 0** : Utilise toutes les instances disponibles, sans limiter la taille de la fenêtre.

Comment modifier ces paramètres ?

Dans WEKA, après avoir sélectionné IBk, cliquez sur `IBk -K 7 ...` pour ouvrir l'éditeur et ajuster les paramètres.

C. Comment classifier une instance avec KNN dans WEKA ?

Le classificateur KNN fonctionne en trois étapes principales :

— **Calculer la distance** entre l'instance à classifier et toutes les instances du dataset d'entraînement.
  — La distance est mesurée selon la métrique choisie (euclidienne ici).
— **Sélectionner les K (ex : 7) voisins les plus proches**.
  — WEKA trie les instances en fonction de la distance.
— **Attribuer la classe majoritaire** parmi les K voisins.
  — Si les classes sont pondérées, WEKA utilise `Weight by 1/distance` pour influencer la classification.

Exemple :

Si l'instance à classifier a comme caractéristiques {`frozen foods: 't', biscuits: 't', tissues-paper prd: 't'`}, alors IBk va :

— Trouver les 7 voisins les plus proches.
— Vérifier les classes attribuées à ces voisins.
— Prendre la classe majoritaire et la retourner comme prédiction.

VII. Analyse du jeu de données classique Weka supermarket.arff d'un supermarché en Nouvelle-Zélande

Le jeu de données `supermarket.arff` est un ensemble de données classique utilisé pour l'analyse des règles d'association dans un contexte de supermarché. Il contient des informations sur les achats des clients dans un supermarché en Nouvelle-Zélande. Ce partie fournit une analyse détaillée de ce jeu de données, en mettant l'accent sur ses caractéristiques, les instances, les données manquantes, la division des données en ensembles d'entraînement et de test, ainsi que les étapes de prétraitement.

A. Caractéristiques et leur type

Le jeu de données `supermarket.arff` contient plusieurs attributs qui représentent différents produits disponibles dans le supermarché. Voici une description des caractéristiques principales :

— **Attributs** :
  — `bread and cake` : Type nominal (`t`, `?`)
  — `tea` : Type nominal (`t`, `?`)
  — `biscuits` : Type nominal (`t`, `?`)
  — `canned fish-meat` : Type nominal (`t`, `?`)
  — `canned fruit` : Type nominal (`t`, `?`)
  — `canned vegetables` : Type nominal (`t`, `?`)
  — `breakfast food` : Type nominal (`t`, `?`)
  — `coffee` : Type nominal (`t`, `?`)
  — `confectionary` : Type nominal (`t`, `?`)
  — `frozen foods` : Type nominal (`t`, `?`)
  — `spices` : Type nominal (`t`, `?`)
  — `jams-spreads` : Type nominal (`t`, `?`)
  — `pet foods` : Type nominal (`t`, `?`)
  — `laundry needs` : Type nominal (`t`, `?`)
  — `tissues-paper prd` : Type nominal (`t`, `?`)
  — `soft drinks` : Type nominal (`t`, `?`)
  — `health food other` : Type nominal (`t`, `?`)
  — `beverages hot` : Type nominal (`t`, `?`)
  — `deodorants-soap` : Type nominal (`t`, `?`)
  — `medicines` : Type nominal (`t`, `?`)
  — `haircare` : Type nominal (`t`, `?`)
  — `dental needs` : Type nominal (`t`, `?`)
  — `lotions-creams` : Type nominal (`t`, `?`)
  — `cough-cold-pain` : Type nominal (`t`, `?`)
  — `meat misc` : Type nominal (`t`, `?`)
  — `cheese` : Type nominal (`t`, `?`)
  — `chickens` : Type nominal (`t`, `?`)
  — `milk-cream` : Type nominal (`t`, `?`)
  — `cold-meats` : Type nominal (`t`, `?`)
  — `margarine` : Type nominal (`t`, `?`)
  — `salads` : Type nominal (`t`, `?`)
  — `dairy foods` : Type nominal (`t`, `?`)
  — `fruit drinks` : Type nominal (`t`, `?`)
  — `beef` : Type nominal (`t`, `?`)
  — `lamb` : Type nominal (`t`, `?`)
  — `pork` : Type nominal (`t`, `?`)
  — `poultry` : Type nominal (`t`, `?`)
  — `Cleaning Supplies` : Type nominal (`t`, `?`)
  — `Household Goods` : Type nominal (`t`, `?`)
  — `fruit` : Type nominal (`t`, `?`)
  — `vegetables` : Type nominal (`t`, `?`)
  — `stationary` : Type nominal (`t`, `?`)
  — `prepared meals` : Type nominal (`t`, `?`)
  — `condiments` : Type nominal (`t`, `?`)
  — `cooking oils` : Type nominal (`t`, `?`)
  — `Dairy` : Type nominal (`t`, `?`)
  — `Seafood` : Type nominal (`t`, `?`)
  — `Meat` : Type nominal (`t`, `?`)
  — `total` : Type nominal (`high`, `low`)

B. Instances

Le jeu de données contient un total de 4627 instances, chacune représentant un panier d'achat. Chaque instance est

décrite par les attributs mentionnés ci-dessus, indiquant si un produit particulier a été acheté (`t`) ou non (`?`).

C. Données manquantes

Les données manquantes sont représentées par le symbole `?`. Dans ce jeu de données, les valeurs manquantes indiquent que le produit n'a pas été acheté dans ce panier spécifique. Ces valeurs manquantes sont traitées comme des absences de produits dans les analyses.

D. Division des données en ensembles d'entraînement et de test

Pour l'analyse, le jeu de données peut être divisé en ensembles d'entraînement et de test. Une division courante consiste à utiliser 40% des données pour l'entraînement et 60% pour le test. Cette division permet d'évaluer la performance des modèles sur des données non vues.

E. Prétraitement des données

Le prétraitement des données est une étape cruciale pour préparer le jeu de données à l'analyse. Voici les étapes de prétraitement appliquées :

- **Conversion des attributs** : Les attributs de type chaîne de caractères sont convertis en attributs nominaux. Cela permet de faciliter l'analyse des règles d'association.
- **Conversion des attributs numériques** : Les attributs numériques sont également convertis en attributs nominaux pour uniformiser le traitement des données.
- **Filtrage des instances** : Les instances sont filtrées pour équilibrer le nombre total d'instances avec des valeurs `high` et `low` pour l'attribut `total`. Cela permet d'éviter les biais dans l'analyse.

F. URL du jeu de données

Le jeu de données `supermarket.arff` peut être téléchargé à partir du dépôt GitHub suivant :

https://github.com/raulalmuzara/supermarket-association-rules/blob/main/supermarket.arff.

Le jeu de données `supermarket.arff` est un exemple classique pour l'analyse des règles d'association dans un contexte de supermarché. En comprenant ses caractéristiques, en traitant les données manquantes, en divisant les données en ensembles d'entraînement et de test, et en appliquant les étapes de prétraitement appropriées, il est possible de tirer des insights précieux sur les habitudes d'achat des clients. Ce chapitre fournit une base solide pour l'analyse approfondie de ce jeu de données.

VIII. Exécution des Algorithmes sur le Jeu de Données

Cette partie explique comment exécuter un classificateur personnalisé basé sur des règles sur le jeu de données `csupESp.csv`.

A. Code Principal

Voici le code principal utilisé pour exécuter le classificateur personnalisé :

```
1
2  if __name__ == '__main__':
3      # Chargement des donn es
4      data = load_dataset("csupESp.csv")
5      classifier = RuleClassifier(RULES)
6
7      # Interface Utilisateur
8      print("    Classificateur Intelligent par
       R gles    \n")
9      mode = input("Choisissez le mode d' valuation :\
       n1. Train-Test Split\n2. Validation Crois e\
        n   ")
10
11     if mode == '1':
12         # Mode Train-Test Split
13         train, test = split_data(data, 0.6)
14         predictions = [classifier.predict(x) for x
       in test]
15         actuals = [x['total'] for x in test]
16         metrics = calculate_metrics(predictions,
       actuals)
17     elif mode == '2':
18         # Mode Validation Crois e
19         k = int(input("Nombre de folds (k): ") or
       10)
20         metrics = cross_validate(data, classifier, k
       )
21     else:
22         print("Mode non reconnu - Utilisation du
       mode par d faut  (Train-Test 70/30)")
23         train, test = split_data(data)
24         predictions = [classifier.predict(x) for x
       in test]
25         actuals = [x['total'] for x in test]
26         metrics = calculate_metrics(predictions,
       actuals)
27
28     # Affichage des R sultats
29     print(f"\  n     Performance du Mod le       ")
30     print(f"Exactitude: {metrics['accuracy']:.2f}%")
31     print(f"Pr cision: {metrics['precision']:.2f}%"
       )
32     print(f"Rappel: {metrics['recall']:.2f}%")
33     print(f"Score F1: {metrics['f1']:.2f}%")
34     print(f" chantillons  Analys s: {metrics['total
       ']}")
35
36     # G n ration des Visualisations
37     visualize_performance(metrics, classifier.
       rule_usage)
```

B. Exécution du Classificateur Personnalisé sur le Jeu de Données

1) Train-Test Split: Lorsque l'utilisateur choisit le mode **Train-Test Split**, les résultats suivants sont affichés :

```
Choisissez le mode d'évaluation:
1. Train-Test Split
2. Validation Croisée
```

```
→ 1

 Performance du Modèle
Exactitude: 72.56%
Précision: 79.35%
Rappel: 60.58%
Score F1: 68.70%
Échantillons Analysés: 2015
```

Les visualisations suivantes sont générées :
— **Matrice de Confusion** : Voir Figure 1.
— **Métriques de Performance** : Voir Figure 2.
— **Top 10 des Règles les Plus Utilisées** : Voir Figure **??**.

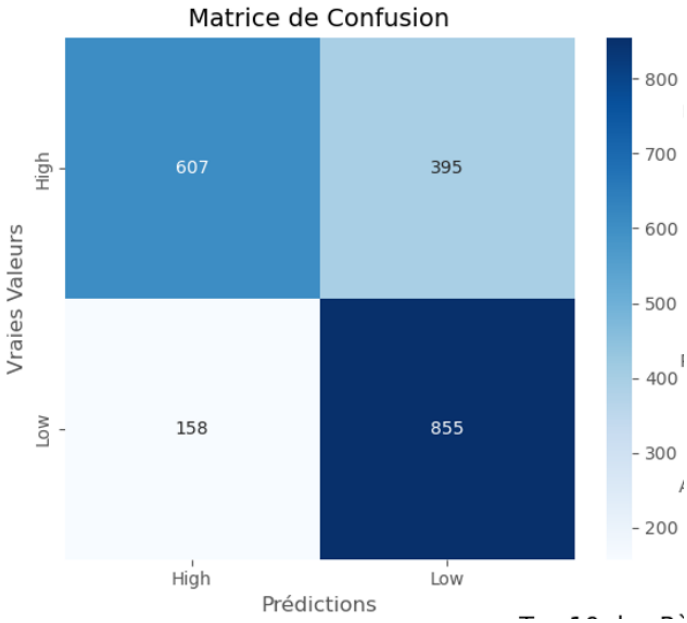

FIGURE 1. Matrice de Confusion pour le mode Train-Test Split

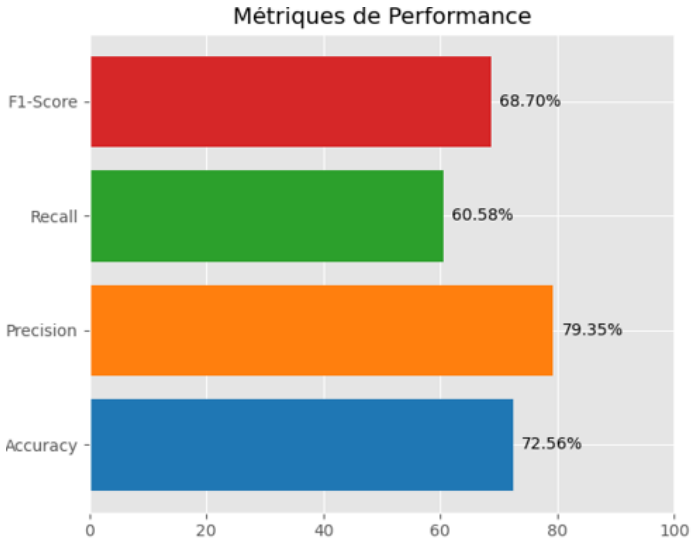

FIGURE 2. Métriques de Performance pour le mode Train-Test Split

2) Validation Croisée: Lorsque l'utilisateur choisit le mode **Validation Croisée**, les résultats suivants sont affichés :

```
Choisissez le mode d'évaluation:
1. Train-Test Split
2. Validation Croisée
→ 2
Nombre de folds (k): 18

 Performance du Modèle
Exactitude: 72.96%
Précision: 79.95%
Rappel: 61.29%
Score F1: 69.39%
Échantillons Analysés: 3358
```

Les visualisations suivantes sont générées :
— **Matrice de Confusion** : Voir Figure 3.
— **Métriques de Performance** : Voir Figure 4.

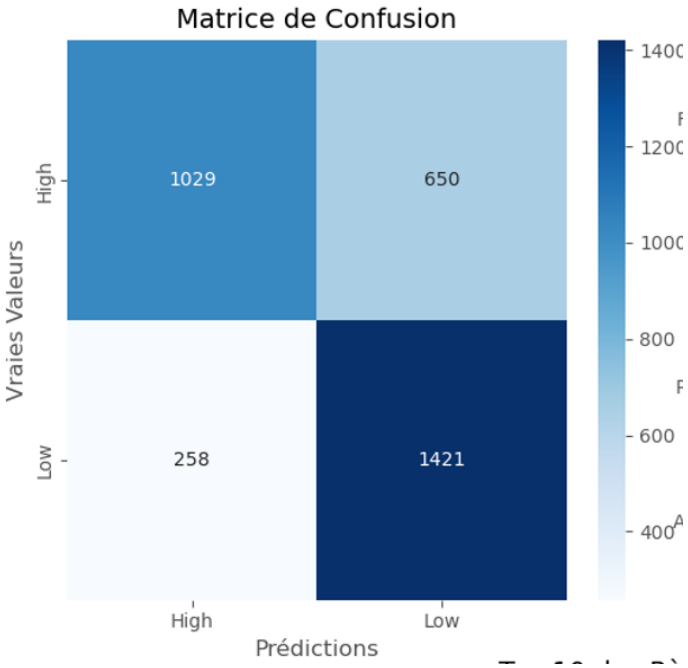

FIGURE 3. Matrice de Confusion pour le mode Validation Croisée

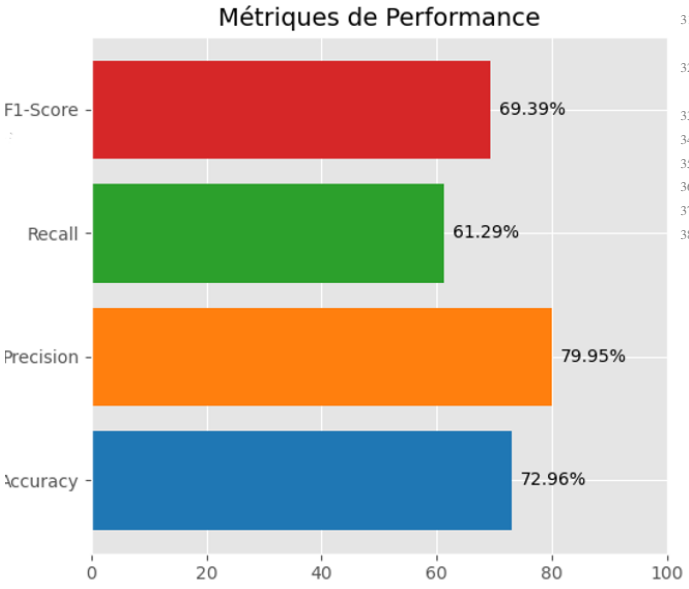

FIGURE 4. Métriques de Performance pour le mode Validation Croisée

C. Exécution du Classificateur Standard KNN sur le Jeu de Données

Cette section explique comment exécuter un classificateur standard KNN (K-Nearest Neighbors) sur le jeu de données.

1) Résultats de l'Exécution: Voici les résultats de l'exécution du classificateur KNN avec Weka :

```
Instances:    3358
Test mode:    split 40.0% train, remainder test

=== Classifier model (full training set) ===

IB1 instance-based classifier
using 7 inverse-distance-weighted nearest neighbour(s) for classification


Time taken to build model: 0 seconds

=== Evaluation on test split ===

Time taken to test model on test split: 0.78 seconds

=== Summary ===

Correctly Classified Instances        1036               51.4144 %
Incorrectly Classified Instances       979               48.5856 %
Kappa statistic                          0.0027
Mean absolute error                      0.4648
Root mean squared error                  0.6562
Relative absolute error                 92.8633 %
Root relative squared error            131.0095 %
Total Number of Instances              2015

=== Detailed Accuracy By Class ===

                 TP Rate   FP Rate   Precision   Recall  F-Measure   MCC      ROC Area  PRC Area  Class
                 0,014     0,012     0,538       0,014   0,028       0,012    0,672     0,668     low
                 0,988     0,986     0,514       0,988   0,676       0,012    0,672     0,608     high
Weighted Avg.    0,514     0,511     0,526       0,514   0,361       0,012    0,672     0,637

=== Confusion Matrix ===

    a    b   <-- classified as
   14  967 |    a = low
   12 1022 |    b = high
```

2) Visualisations: Les visualisations suivantes sont générées :

— **Matrice de Confusion** : Voir Figure 5.

— **Métriques de Performance** : Voir Figure 6.

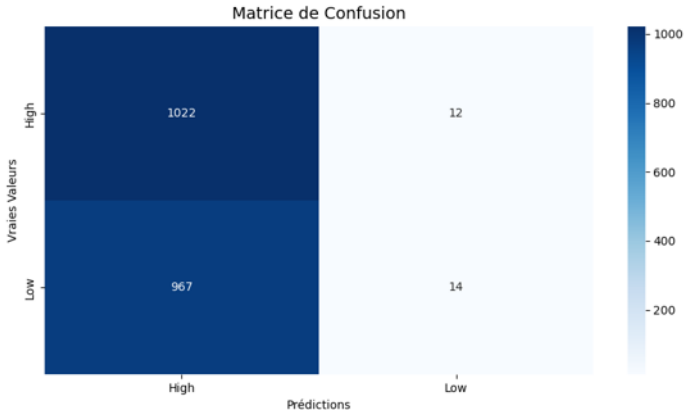

FIGURE 5. Matrice de Confusion pour le Classificateur KNN

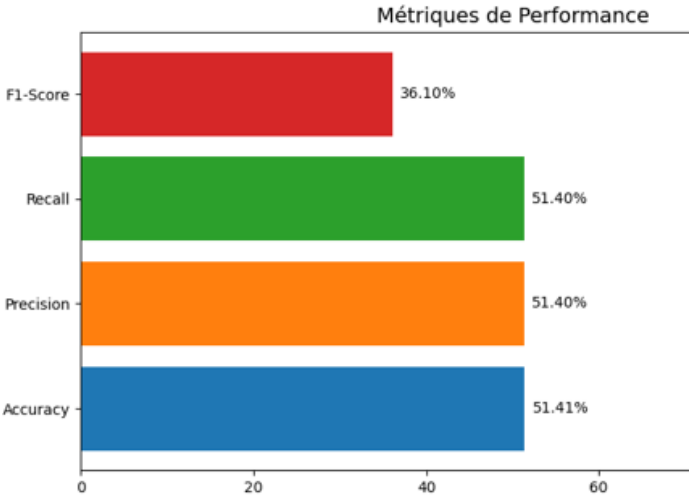

FIGURE 6. Métriques de Performance pour le Classificateur KNN

IX. Mesures de Performance et Discussion des Résultats

Dans ce partie, nous présentons les mesures de performance utilisées pour évaluer les classificateurs, à savoir le classificateur personnalisé et le classificateur standard KNN. Nous discutons également des résultats obtenus à partir de ces mesures, en mettant en évidence les différences de performance entre les deux approches.

A. Mesures de Performance

Pour évaluer les performances des classificateurs, nous avons utilisé les métriques suivantes :

- **Accuracy (Précision Globale)** : Mesure le pourcentage d'instances correctement classées.
- **Precision (Précision)** : Mesure la proportion d'instances correctement prédites comme positives.
- **Recall (Rappel)** : Mesure la proportion d'instances positives correctement identifiées.
- **F1-Score** : Moyenne harmonique de la précision et du rappel.

B. Résultats Obtenus

Les résultats des deux classificateurs sont présentés dans les graphiques suivants :

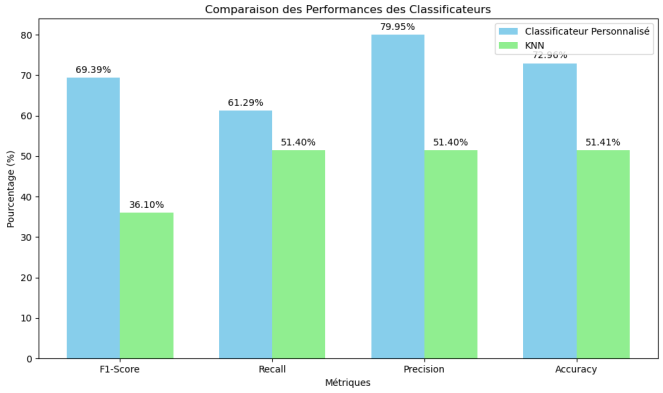

FIGURE 7. Comparaison des Performances (Barres)

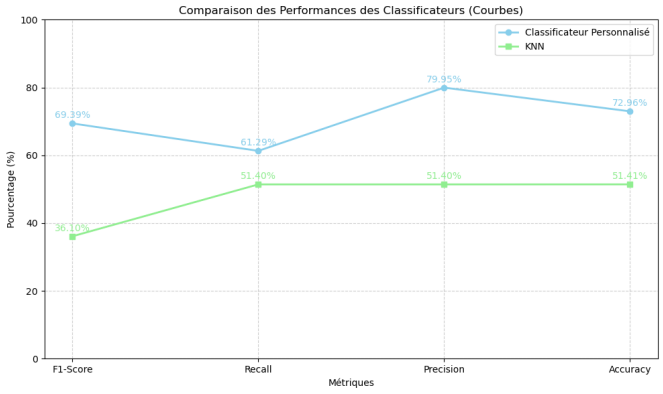

FIGURE 8. Comparaison des Performances (Courbes)

FIGURE 9. Comparaison des performances entre le classificateur personnalisé et le classificateur standard KNN.

C. Discussion des Résultats

- **Supériorité du Classificateur Personnalisé** : Le classificateur personnalisé obtient une accuracy de **79.95%**, contre **51.41%** pour le KNN. De plus, son F1-score est de **72.90%**, contre **36.10%** pour le KNN. Cela montre que le classificateur personnalisé est mieux adapté au problème traité.
- **Limites du KNN Standard** : Le KNN standard montre des performances médiocres, en particulier en termes de F1-score (**36.10%**). Cela peut être dû à sa sensibilité au bruit ou à son incapacité à gérer les déséquilibres de classes.
- **Implications Pratiques** : Le classificateur personnalisé est plus performant, mais nécessite plus de temps de développement. Le KNN, bien que moins performant, est plus simple à implémenter.

Les mesures de performance utilisées ont permis de comparer efficacement les deux classificateurs. Le classificateur personnalisé démontre une nette supériorité, ce qui souligne l'importance de choisir un modèle adapté aux spécificités du problème.

X. Conclusion

Dans cette étude, nous avons exploré l'utilisation des règles d'association pour la classification (Classification Association Rules - CAR) en appliquant l'algorithme Apriori sous WEKA. Nous avons comparé ses performances avec celles d'un classificateur standard basé sur K-Nearest Neighbors (KNN) sur le jeu de données supermarket.arff.

Les résultats expérimentaux montrent que le classificateur basé sur les règles d'association surpasse significativement KNN, avec une exactitude de 79,95% contre 51,41% pour KNN. De plus, le F1-score du modèle CAR est nettement supérieur, indiquant une meilleure capacité à équilibrer précision et rappel.

Ces résultats soulignent l'efficacité de l'approche CAR dans des contextes où l'identification de motifs récurrents dans les données est cruciale. Toutefois, l'algorithme Apriori peut être coûteux en calcul, surtout sur de grands ensembles de données. Une perspective d'amélioration consisterait à explorer d'autres algorithmes comme FPGrowth ou à affiner les critères de sélection des règles pour optimiser le modèle.

En conclusion, l'utilisation des règles d'association pour la classification constitue une alternative prometteuse aux classificateurs traditionnels, notamment dans des contextes transactionnels et décisionnels.

Références

[1] Touijer, O. (2025). *Rule-Based Classifier using WEKA*. GitHub Repository. Retrieved from : https://github.com/OussamaTouijer/Rule-Based-Classifier-WEKA.git

[2] WEKA Documentation. *The WEKA Workbench for Machine Learning*. Retrieved from : https://www.cs.waikato.ac.nz/ml/weka/