Rapport du projet Machine Learning and Programming Languages

Groupe E

Introduction

DBSCAN est un algorithme de clustering populaire introduit par Ester et al. en 1996 [**?**]. Contrairement aux techniques de clustering traditionnelles telles que k-means, DBSCAN ne nécessite pas que l'utilisateur spécifie le nombre de clusters. Au lieu de cela, il se base sur la densité des points pour former des clusters.

Concepts Clés

- **Points Noyaux :** Points avec au moins `MinPts` voisins dans un rayon donné ε.
- **Points Frontières :** Points qui ne sont pas des noyaux mais qui se trouvent dans le voisinage ε d'un point noyau.
- **Points Bruit :** Points qui ne sont ni des noyaux ni des points frontières et sont considérés comme des anomalies.

Étapes de l'Algorithme

1. Pour chaque point non visité P, déterminer son voisinage ε.
2. Si P est un point noyau, former un cluster en étendant P pour inclure tous les points atteignables par densité.
3. Marquer les points comme bruit s'ils n'appartiennent à aucun cluster.

Pseudo-code de l'Algorithme DBSCAN

Fonction `EuclDist`

Entrée :

- x: Ensemble des données.
- p: Point de référence.
- ε: Distance seuil pour trouver les voisins.
- MinPts: Nombre minimum de points pour classifier.

Algorithm 1 EuclDist

```
NeighboringPoints ← []
type ← 0
for q ← 1 à longueur(x) do
    dist ← ||q − p||
    if dist < ε then
        Ajouter q à NeighboringPoints
    end if
end for
if longueur(NeighboringPoints) > MinPts then
    type ← 1 {Point noyau}
else if longueur(NeighboringPoints) > 1 then
    type ← 2 {Point frontière}
else
    type ← 3 {Bruit}
end if
return (NeighboringPoints, type)
```

Fonction `DBSCAN`

Entrée :

- x: Ensemble des données.
- ε, MinPts: Paramètres de l'algorithme.

Algorithm 2 DBSCAN

```
currentCluster ← 0
points ← []
for chaque p dans x do
    (NeighboringPoints, type) ← EuclDist(x, p, ε, MinPts)
    Ajouter un point avec type, NeighboringPoints, cluster ← 1 – type à points
end for
for i ← 1 à longueur(points) do
    if points[i].cluster = 0 then
        currentCluster ← currentCluster + 1
        points[i].cluster ← currentCluster
        Appeler findClusterPoints(x, currentCluster, points, i, ε, MinPts)
    end if
end for
return points
```

Fonction `findClusterPoints`

Entrée :

- x: Ensemble des données.
- currentCluster: Numéro du cluster courant.
- points: Liste des points avec leurs propriétés.
- position: Index du point noyau en cours d'expansion.

Algorithm 3 findClusterPoints

```
ClusterMembers ← points[position].neighboringPoints
i ← 0
while i < longueur(ClusterMembers) do
    expansionPoint ← ClusterMembers[i]
    if points[expansionPoint].cluster = −1 then
        points[expansionPoint].cluster ← currentCluster
    else if points[expansionPoint].cluster = 0 then
        points[expansionPoint].cluster ← currentCluster
        Ajouter points[expansionPoint].neighboringPoints à ClusterMembers
    end if
    i ← i + 1
end while
```

Analyse de la Complexité

Fonction `EuclDist`

- **Boucle principale :** La fonction parcourt tous les points q de x, soit $O(n)$ où n est le nombre total de points.

- **Calcul de la distance :** Le calcul $||q - p||$ dépend de la dimension d des données, soit $O(d)$.

- **Complexité totale :** $O(n \cdot d)$.

Fonction `DBSCAN`

1. **Phase dínitialisation :**

- Pour chaque point p, la fonction appelle `EuclDist`.

- Chaque appel parcourt tous les points de x, ce qui coûte $O(n \cdot d)$.

- Avec n points, le coût total pour cette phase est $O(n \cdot (n \cdot d)) = O(n^2 \cdot d)$.

2. **Propagation des clusters :**

- La fonction `findClusterPoints` est appelée pour chaque point noyau identifié.

- Chaque voisin d'un noyau est exploré une seule fois.

- En supposant k noyaux avec en moyenne m voisins, le coût est proportionnel à $O(k \cdot m)$.

- Puisque $m \leq n$ et $k \leq n$, le coût est dominé par $O(n^2)$.

Complexité Totale

- La phase d'initialisation domine, résultant en une complexité totale de $O(n^2 \cdot d)$.

Optimisations Possibles

- **Accélération des recherches de voisins :** L'utilisation de structures comme les arbres k-d ou les grilles réduit la recherche de voisins à $O(\log n)$, abaissant le coût d'initialisation à $O(n \cdot \log n \cdot d)$.

- Avec cette optimisation, la complexité totale devient $O(n \cdot \log n \cdot d)$.

Impact de la Dimension d

- Pour des données de haute dimension, le calcul des distances $O(d)$ devient dominant.

Analyse comparative des résultats

Pour comparer les implémentations de DBSCAN dans trois langages (Python, Julia et R), nous avons mesuré les performances et évalué les résultats sur un même jeu de données synthétique. Voici les conclusions :

Temps d'exécution

- **Python :** Temps d'exécution modéré grâce à des bibliothèques optimisées comme `scikit-learn`.
- **Julia :** Performances élevées en raison de la compilation juste-à-temps (JIT).
- **R :** Temps d'exécution plus long, particulièrement pour les grands ensembles de données.

Lisibilité et simplicité du code

- **Python :** Code clair et concis grâce aux bibliothèques bien documentées.
- **Julia :** Syntaxe intuitive, mais bibliothèques moins matures que celles de Python.
- **R :** Code plus complexe à cause d'une gestion moins intuitive des objets.

Résultats de clustering

- Les trois implémentations ont produit des résultats similaires en termes de formation des clusters.
- **Adjusted Rand Index (ARI) :** Les scores ARI étaient équivalents, indiquant une cohérence des résultats.

Disponibilité des bibliothèques

- **Python :** Large gamme de bibliothèques prêtes à l'emploi.
- **Julia :** Moins de choix, mais des outils efficaces pour les tâches spécifiques.
- **R :** Fortement spécialisé dans l'analyse de données, mais moins flexible pour des tâches générales.

Conclusion

DBSCAN reste un algorithme puissant pour le clustering. Les différences dans les performances et la simplicité dépendent largement du langage utilisé et de ses bibliothèques associées. Python est recommandé pour un développement rapide, Julia pour des performances optimales, et R pour une analyse approfondie des données.