INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Rémi Negrier – Pierre Marigo
5e année Informatique et Réseaux (SDBD - A1)

Compte-rendu du TP clustering

Le code utilisé pour la rédaction de ce rapport ainsi que les sources de nos données sont disponibles sur notre répertoire Github :
`https://github.com/pierremrg/ApprentissageNS`

Avant-propos : présentation des jeux de données utilisés

Durant ce TP, nous avons utilisé plusieurs jeux de données sur lesquels nous avons répété tous nos tests. Néanmoins, tous ne seront pas présentés dans ce rapport (car tous ne sont pas forcément pertinents).

Ci-dessous, un tableau récapitulatif des données que nous avons utilisées pour la réalisation de ce rapport :

Forme	Convexes	Séparées	Densités	Bruité
Banana	Non	Oui	Similaires	Non
Aggregation	Oui	Oui	Similaires	Non
Spiral	Non	Oui	Similaires	Non
Cure	Oui	Oui	Non similaires	Oui
Diamonds	Oui	Oui	Similaires	Non
Elly	Oui	Non	Non similaires	Non
2d-4c-no4	Oui	Oui	Non similaires	Non

Comme nous pouvons le voir, nous avons essayé d'utiliser des données différentes afin de tester les algorithmes dans des cas éloignés. En particulier, nous nous intéressons à la forme *diamonds* qui est très "simple" ainsi qu'à la forme *elly*, pour laquelle il semble à priori difficile de définir des clusters.

Pour plus de détails, nos jeux de données sont disponibles en suivant les liens suivants :

— *Banana* : `https://github.com/deric/clustering-benchmark/blob/master/src/main/resources/datasets/artificial/banana.arff`
— *Aggregation* : `https://github.com/deric/clustering-benchmark/blob/master/src/main/resources/datasets/artificial/aggregation.arff`
— *Spiral* : `https://github.com/deric/clustering-benchmark/blob/master/src/main/resources/datasets/artificial/3-spiral.arff`
— *Cure* : `https://github.com/deric/clustering-benchmark/blob/master/src/main/resources/datasets/artificial/cure-t2-4k.arff`
— *Diamonds* : `https://github.com/deric/clustering-benchmark/blob/master/src/main/resources/datasets/artificial/diamond9.arff`
— *Elly* : `https://github.com/deric/clustering-benchmark/blob/master/src/main/resources/datasets/artificial/elly-2d10c13s.arff`
— *2d-4c-no4* : `https://github.com/deric/clustering-benchmark/blob/master/src/main/resources/datasets/artificial/2d-4c-no4.arff`

1 Clustering k-Means

La méthode k-Means est une méthode itérative qui propose de placer k centres aléatoirement pour commencer. Le but à chaque itération est des placer toutes les données dans des clusters, et de faire évoluer les centres de chaque cluster, jusqu'à un état stable.

1.1 Première utilisation

Avec k-Means, il est nécessaire de renseigner le nombre de clusters à chercher (le k).

Pour commencer, nous avons décidé de rechercher les 9 clusters présents dans la figure *diamonds*. La figure ci-dessous représente les résultats obtenus, les croix noires représentant les centres des clusters déterminés par l'algorithme.

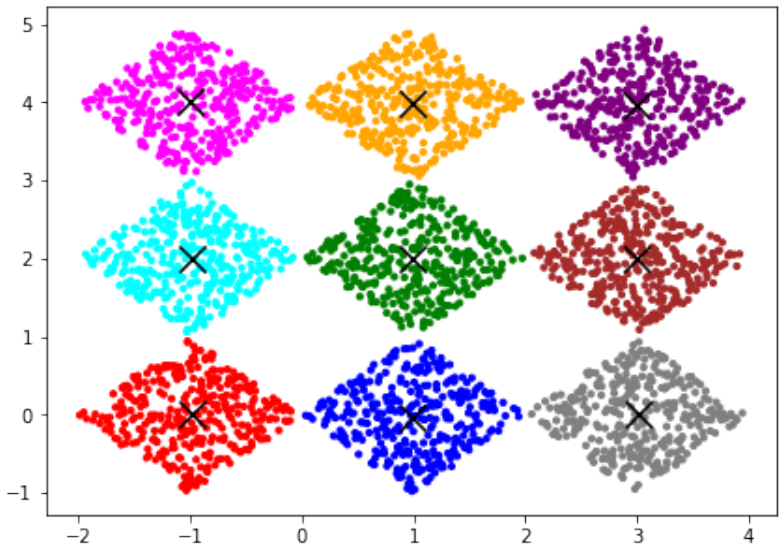

FIGURE 1 – Résultats obtenus avec k-Means pour diamonds.

A vue d'oeil, l'algorithme semble avoir déterminé correctement les centres de gravités pour les différents clusters.

1.2 Recherche du k optimal

Dans la première partie, nous connaissions le nombre de centres de gravité à placer : 9. Sur des problèmes réels de clustering, ce nombre n'est pas toujours connu, et il est nécessaire de le déterminer. Pour rechercher ce k, nous avons donc lancé l'algorithme k-Means plusieurs fois, avec un paramètre k différent à chaque fois, pour comparer les scores obtenus par l'algorithme à chaque fois.

Il est possible de calculer le score d'un résultat avec différentes méthodes d'évaluation. Parmi les métriques proposées par *scikit-learn*, nous avons décidé d'utiliser celles qui proposent de mesurer le score d'une solution calculée par l'algorithme sans pouvoir la comparer avec la vraie solution (pour les TP, la vraie solution est toujours fournie dans les jeux de données utilisés ; en pratique, le but des algorithmes de clustering est de classifier sans connaître cette solution : c'est le principe de l'apprentissage non supervisé). Ainsi, nous avons mesurer notre score avec :

— Silhouette : compris entre -1 et 1
— Calinski-Harabasz : calcul en fonction de la dispersion dans nos clusters
— Davies-Bouldin : se rapproche de 0 quand la solution est la meilleure

Pour toute la suite du TP, ces métriques seront utilisées. Voici les scores obtenus pour différentes valeurs de k :

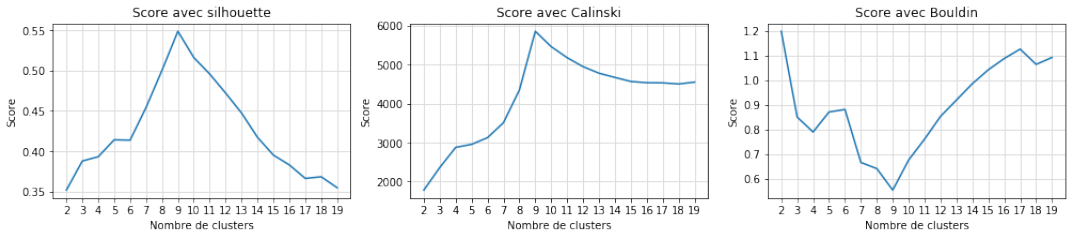

FIGURE 2 – Scores obtenus avec k-Means pour diamonds.

Les différentes métriques confirment bien la réalité : on a de meilleurs scores pour k=9, soit le bon nombre de clusters à trouver.

Concernant le temps de recherche en fonction de k, on peut remarquer que le temps est croissant en règle générale, mais le temps de recherche pour k=9 est réduit par rapport à ses voisins. Cela peut s'expliquer par le fait que la catégorisation en cluster est plus "facile", comme k est le bon nombre de clusters à trouver, et que l'algorithme se stabilise donc plus vite et fait moins d'itérations.

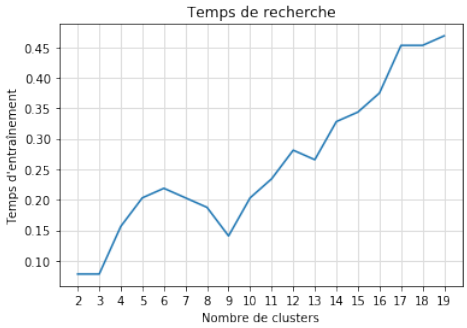

FIGURE 3 – Temps de recherche avec k-Means pour diamonds.

1.3 Généralisation à d'autres figures

Utilisé sur le jeu de données *diamonds*, k-Means est très efficace et présente un bon score. En revanche, on peut très vite observer les limites de cette méthode avec des données différentes : la détection des centres des clusters ne fonctionne pas pour des formes non convexes.

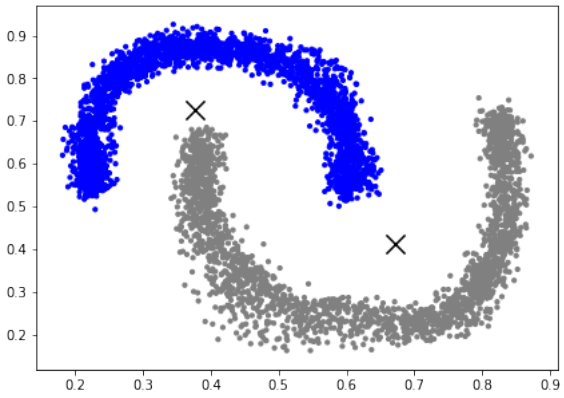

FIGURE 4 – Résultats obtenus avec k-Means pour la figure *banana*.

Il en est de même avec des jeux de données présentant des clusters moins bien séparés ou avec du bruit. Par exemple, voici les différents scores obtenus pour le jeu de données *elly* (nombre de clusters à trouver : 10) :

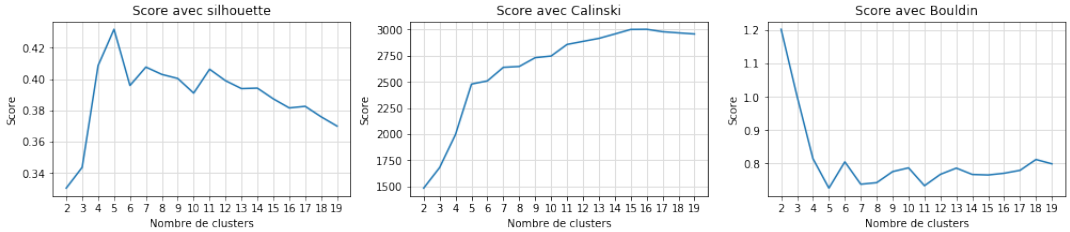

FIGURE 5 – Scores obtenus avec k-Means pour elly.

Et voici le score obtenu pour le jeu *cure* (nombre de clusters à trouver : 6, avec du bruit)

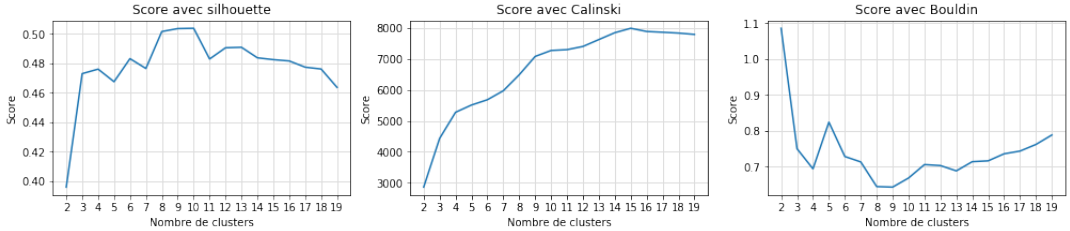

FIGURE 6 – Scores obtenus avec k-Means pour cure.

Sur les deux graphiques ci-dessus, il est impossible de distinguer un "pic" de score pour les k corrects, comme il était possible de le faire avec le *diamonds*. k-Means n'est pas adapté à ce type de données.

1.4 Conclusion

Comme nous l'avons vu, k-Means est plutôt efficace pour déterminer différents clusters si notre jeu de données est bien séparé et sans bruit. Cela devient plus compliqué avec des données plus complexes.

2 Clustering agglomératif

Le principe du clustering agglomératif et de considérer chaque point comme un cluster, puis de fusionner les clusters qui sont proches, jusqu'à obtenir le bon nombre de clusters.

2.1 Première utilisation

Dans ce premier cas, nous reprenons notre figure *diamonds* et nous tentons d'appliquer la méthode avec un nombre de clusters à trouver k=9.

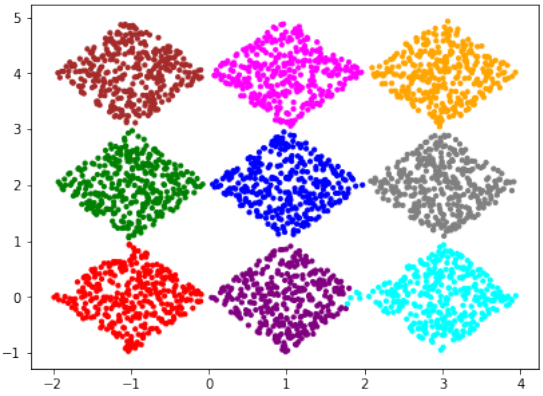

FIGURE 7 – Résultats obtenus avec le clustering agglomératif pour diamonds.

On note encore une fois que les résultats sont, pour cette figure aux propriétés simples, très bons.

2.2 Généralisation à d'autres figures

De la même façon qu'avec k-Means, les 9 clusters dans la figure précédente semblent avoir été trouvés presque parfaitement. En revanche, on note que pour des données bruitées et de densité variable, cette méthode rencontre plus de difficultés. Il en va de même dès lors que les figures sont non convexes : les résultats sont très mauvais.

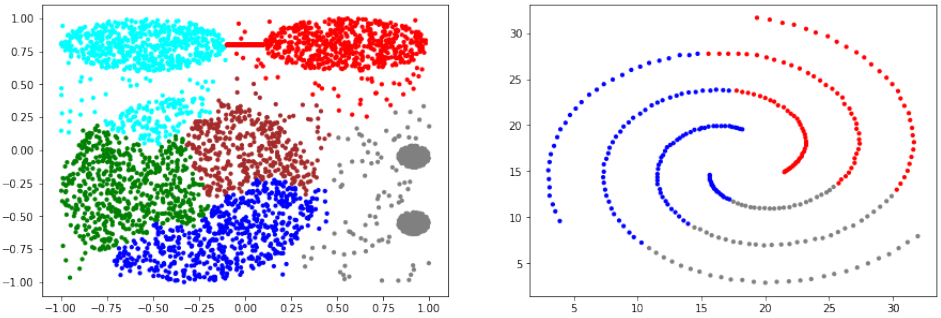

FIGURE 8 – Clustering agglomératif pour des données bruitées (gauche) et non convexes (droite).

2.3 Comparaison des différentes méthodes de linkage

Différentes méthodes peuvent être utilisées pour rassembler les clusters au fur et à mesure des itérations. Nous allons en tester quatre :

— ward
— complete
— single
— average

2.3.1 Détermination du k optimal

Les résultats varient beaucoup d'un linkage à l'autre et selon le type de données utilisées.

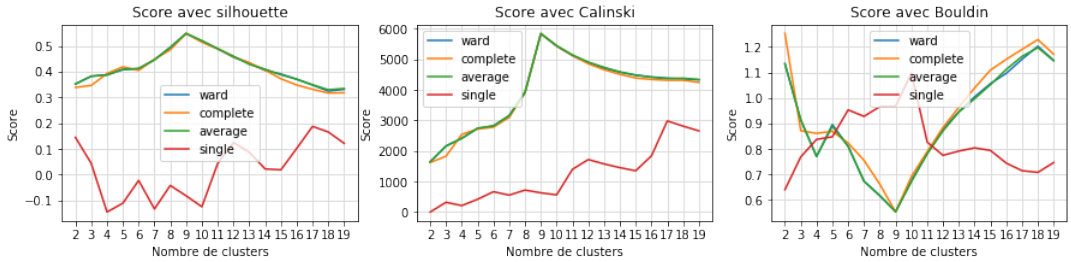

FIGURE 9 – Résultats pour différents linkages avec diamonds (k correct : 9).

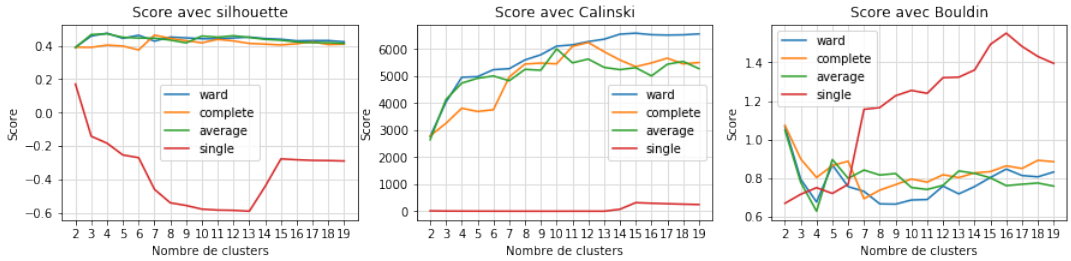

FIGURE 10 – Résultats pour différents linkages avec des données bruitées (k correct : 6).

Ainsi, on s'aperçoit que, bien que les méthodes de linkage *ward*, *complete* et *average* donnent des résultats plus ou moins similaires, la méthode *single*, elle, donne des résultats vraiment mauvais et ne permet visiblement pas de trouver les clusters efficacement, et ce quelle que soit la forme des données (bruitées ou non). Nous pouvons d'ailleurs le voir ci-dessous avec la coloration des figures selon les clusters trouvés avec le linkage *single* :

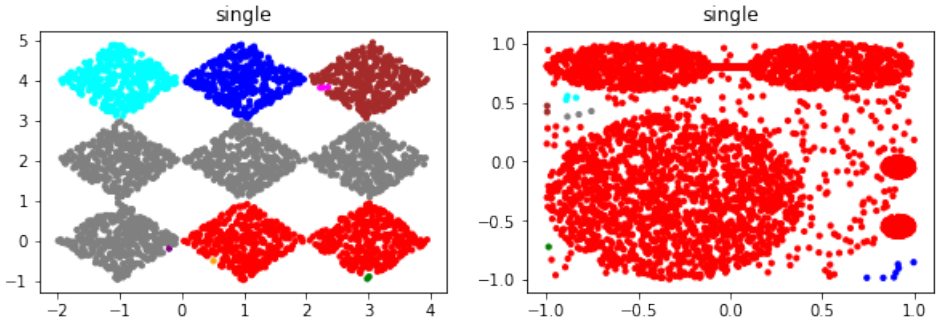

FIGURE 11 – Clustering agglomératif en utilisant la méthode single.

On note toutefois que, pour les autres linkages et pour des données bruitées, il reste difficile de dire quel est le meilleur k trouvé. Ils fonctionnent cependant très bien pour des données non bruitées et simples à délimiter. On retrouve notamment le k optimal de la figure *diamonds* (k=9) avec les linkages *ward*, *complete* et *average*, alors qu'aucun résultat exploitable n'est trouvé avec la méthode *single*.

2.3.2 Temps de recherche avec les différents linkages

Nous allons maintenant nous intéresser aux temps de recherche de l'algorithme de clustering agglomératif pour les différents linkages :

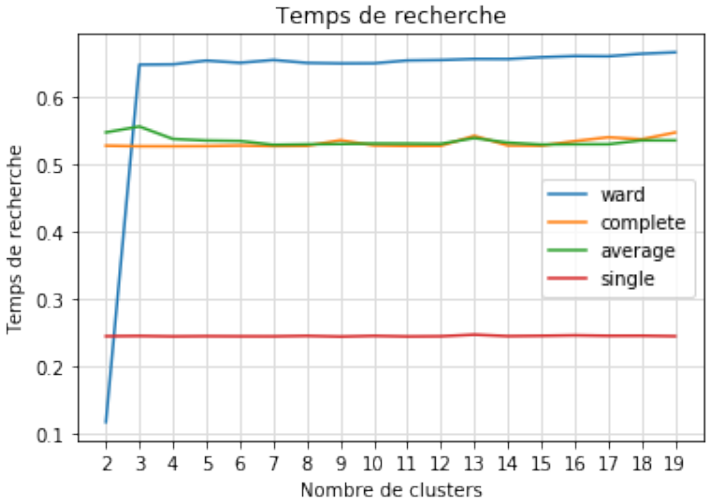

FIGURE 12 – Temps de recherche pour les différents linkages selon différent k (k correct : 9).

A première vue, le linkage *single* semble le plus rapide ; le *ward* le plus lent, bien que globalement proche des linkages *complete* et *average*. En revanche, on note que cette méthode ne semble pas sensible au nombre final de clusters à trouver.

2.4 Conclusion

Globalement, cette méthode présente des résultats similaires à la méthode k-Means, quoiqu'un petit peu moins rapide.

Cette méthode semble fonctionner très bien pour des formes claires et bien délimitées, aux densités proches. En revanche, dès lors que du bruit apparaît, ou que les formes ont des densités variables ou sont non convexes, cette méthode n'est plus capable de déterminer correctement les clusters.

3 Clustering DBSCAN & HDBSCAN

Dans cette partie, nous traitons les méthodes DBSCAN et HDBSCAN. Ces méthodes étant proches, nous nous proposons de les comparer.

La méthode DBSCAN est utilisée pour essayer de déterminer des clusters avec des formes non convexes. On utilise ici la notion de voisinage, en essayant de créer un chemin pour relier deux points d'un même cluster, tout en restant dans ce cluster.

3.1 Première utilisation de DBSCAN "au hasard"

Pour commencer, voici quelques tests effectués avec des valeurs *epsilon* choisies au hasard et le paramètre *min_samples* laissé par défaut :

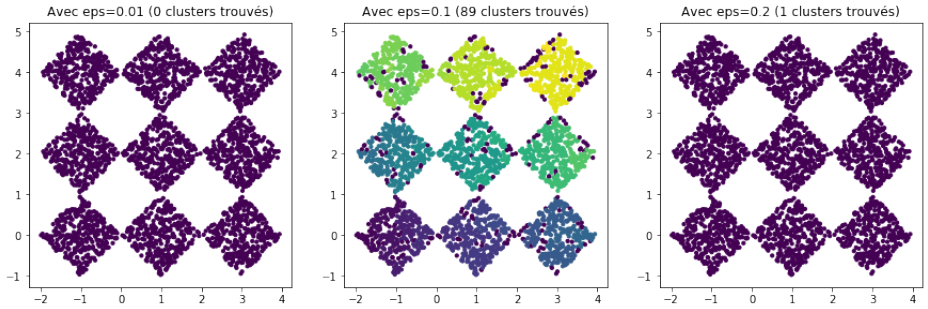

FIGURE 13 – Résultats avec DBSCAN paramétré au hasard.

On voit que DBSCAN n'est pas du tout efficace lorsqu'il n'est pas paramétré correctement. Dans la suite de cette partie, le but sera de déterminer les bonnes valeurs *epsilon* et *min_samples* pour ce dataset, et pour les autres jeux de données que nous avons retenus.

3.2 Détermination des paramètres optimaux

3.2.1 Recherche grossière

Pour trouver les paramètres optimaux, nous avons essayé de couvrir une large plage de valeurs pour *epsilon* et *min_samples*, afin de trouver la meilleure combinaison. Voici quelques résultats obtenus.

Pour chacune des solutions proposées ci-dessous, les données non clusterisées par l'algorithme (car considérées comme du bruit) apparaissent en rose. Même si les deux premiers résultats sont moins intéressants que les autres (le cluster vert est mal défini) ont voit qu'il est possible d'obtenir de très bon résultats avec DBSCAN sur ce genre de données.

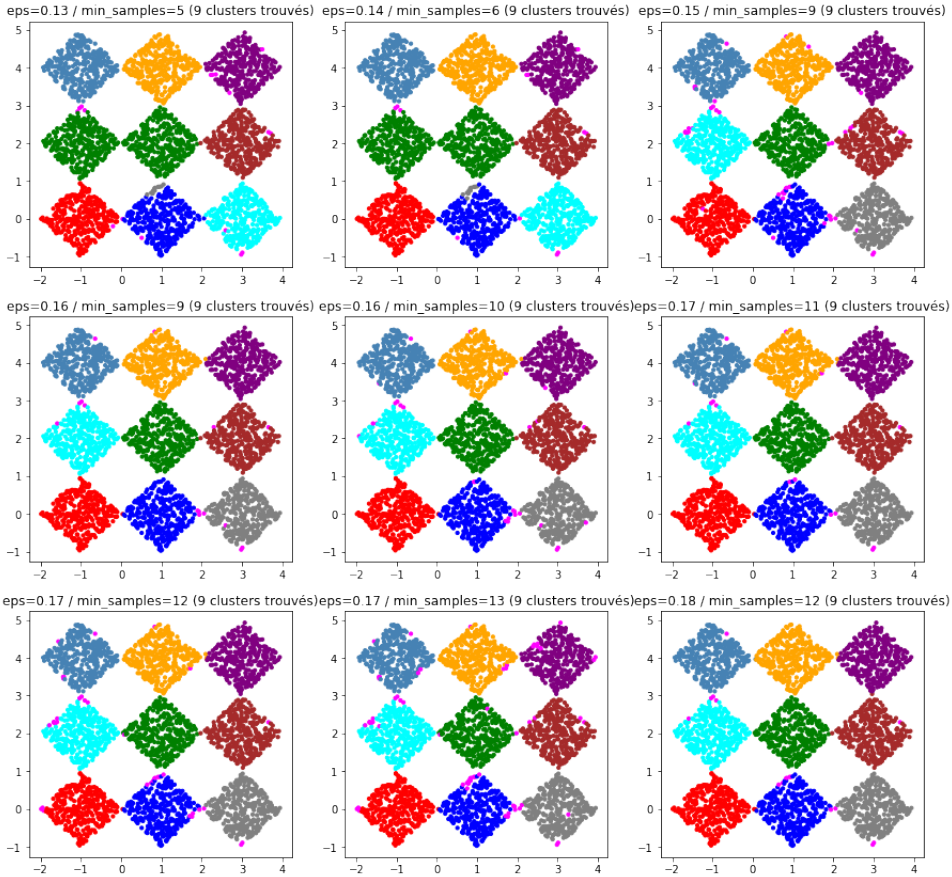

FIGURE 14 – Résultats avec DBSCAN : recherche des bons paramètres.

Comme nous l'avions fait pour les méthodes précédentes, il est possible encore une fois de calculer des scores avec les métriques de *scikit-learn*. En revanche, la visualisation est plus compliquée ici, car nous faisons varier deux paramètres et non plus 1 seul (pour rappel, le score était calculé en fonction du nombre de clusters à trouver dans les méthodes présentées précédemment). Il est nécessaire de représenter nos résultats en 3 dimensions, et cela rend la recherche du meilleur couple (*epsilon*, *min_samples*) compliquée à l'oeil nu. Voici un exemple de l'évolution des scores *silhouette* représentée en 3D :

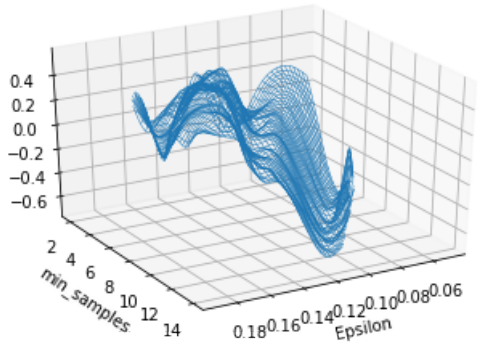

FIGURE 15 – Représentation 3D des résultats avec DBSCAN.

Bien qu'on puisse détecter certains sommets intéressants, il est effectivement difficile de déterminer le meilleur couple de paramètres. Nous avons donc cherché une méthode plus fine pour trouver précisément les valeurs optimales.

3.2.2 Recherche avancée

L'idée de cette recherche avancée est de d'abord rechercher un bon *epsilon* avant de faire varier, en utilisant le *epsilon* trouvé, le paramètre *min_samples*.

Le paramètre *epsilon* représente la distance maximale entre deux points pour qu'ils soient considérés comme voisins. Nous avons donc au préalable étudié la distance moyenne entre les points. Cette pré-étude consiste, pour chaque point, à mesurer la distance moyenne entre ledit point et ses n plus proches voisins.

Dans notre cas, nos données sont contenues dans un plan 2D. De ce fait, nous avons implémenté un calcul simple de distance euclidienne pour mesurer la distance entre deux points. Pour un point donné, une fois que les distances entre lui et tous les autres points sont calculées, nous gardons les n plus petites, puis nous calculons la moyenne de ces n plus petites distances (dans les faits, nous avons réalisé l'expérimentation avec $n = 5$).

Pour faciliter la visualisation des résultats dans le graphe ci-dessous, nous trions ces moyennes par ordre croissant. Nous avons donc, en abscisse, les points de notre figure (4000 dans notre cas), et en ordonnée, la distance moyenne entre lui et ses n plus proches voisins.

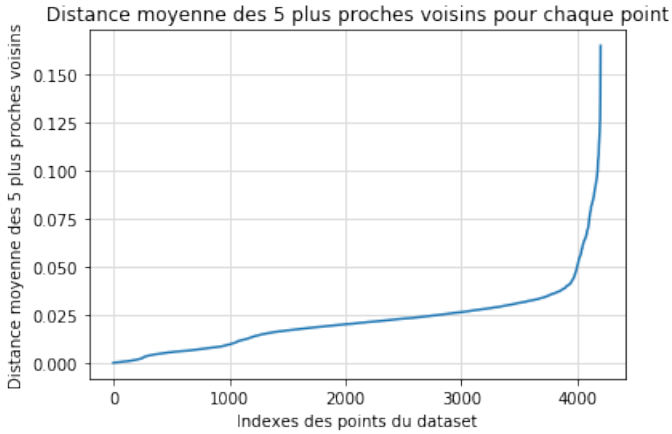

FIGURE 16 – Distance moyenne des 5 plus proches voisins pour chaque point.

Le tri croissant permet de regrouper les points aux moyennes semblables. Ainsi, nous pouvons voir que la très grande majorité des points ont leur plus proches voisins à une distance inférieure ou égale à 0.05 ; considérer des points au-delà de cette distance reviendrait à ajouter du bruit pour la plupart des points. On décide donc de garder *epsilon* compris entre 0.04 et 0.06.

Pour *epsilon* compris dans cet intervalle (avec un pas de 0.005), nous faisons maintenant varier *min_samples* entre 2 et 14 afin de trouver la meilleure combinaison possible.

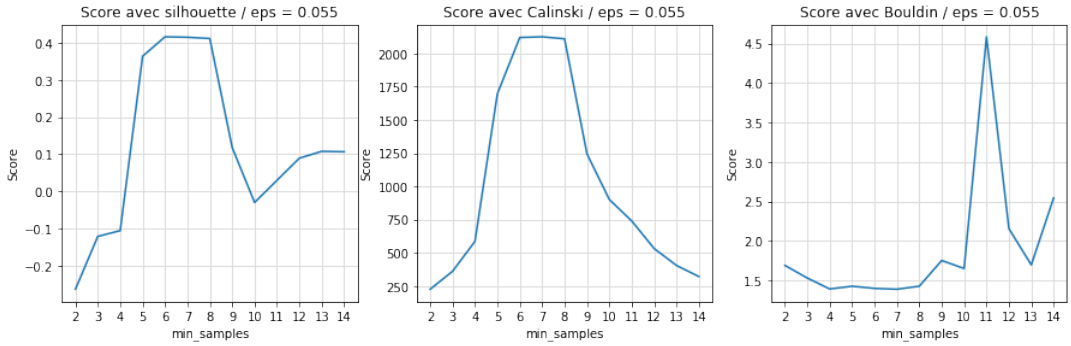

FIGURE 17 – Scores pour un *epsilon* = 0.055 et différents *min_samples*.

La figure montre les résultats de nos tests pour une certaine valeur de *epsilon*. On peut clairement voir que les résultats sont meilleurs pour des valeurs de *min_samples* compris entre 5 et 8. Notons que ce constat est globalement le même quelque soit la valeur de epsilon choisie dans l'intervalle.

Ainsi, en prenant des valeurs telles que 0.04 <= *eps* <= 0.06 et 5 <= *min_samples* <= 8, nous obtenons ce genre de résultat avec DBSCAN :

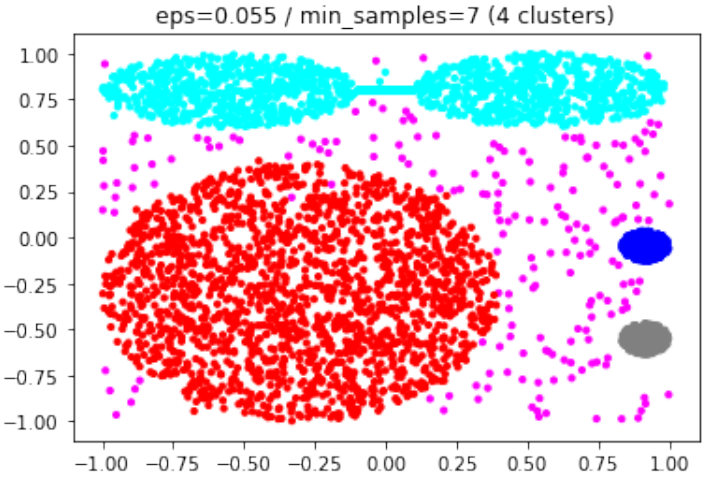

FIGURE 18 – Clusters trouvés par DBSCAN pour *epsilon* = 0.55 et *min_samples* = 7.

On observe notamment que DBSCAN détecte parfaitement le bruit (en rose) dans notre figure. En revanche, les clusters ne sont pas tous trouvés, et, de façon générale, l'algorithme semble avoir du mal avec les données de densités variables.

3.3 Généralisation à d'autres formes

Le but ici est d'appliquer DBSCAN à d'autre formes. Normalement, la méthode doit être efficace pour clusteriser correctement des formes non convexes et/ou avec du bruit.

Pour la figure *spiral*, DBSCAN fonctionne correctement avec *epsilon* = 1.5 et *min_samples* = 3. Voici les résultats obtenus :

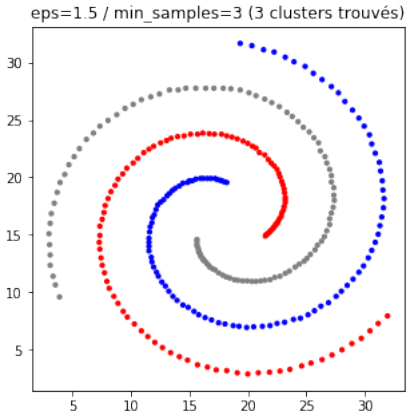

FIGURE 19 – Résultat DBSCAN avec *epsilon* = 1.5 et *min_samples* = 3.

Comme attendu, DBSCAN permet donc de clusteriser efficacement des formes non convexes. Il est également efficace dans la détection du bruit (même si nous avons vu que le clustering pouvait ne pas être parfait à cause des densités différentes dans les données).

3.4 Apports de HDBSCAN

L'algorithme HDBSCAN est en fait une application de DBSCAN pour plusieurs valeurs du rayon de recherche *epsilon*.

Pour prendre en main HDBSCAN, nous avons d'abord repris notre jeu de données *diamonds*, afin de retrouver la même précision obtenue avec DBSCAN, en jouant sur le paramètre *min_cluster_size*.

Voici le résultat obtenu pour *min_cluster_size* = 10 (les données classifiées comme du bruit sont en rose sur la figure) :

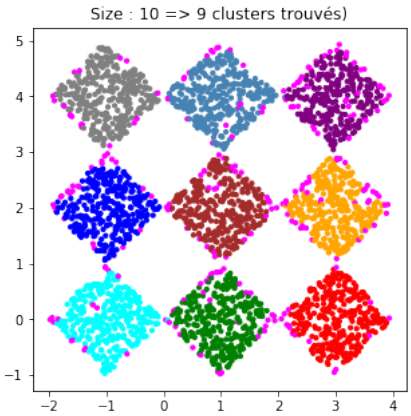

FIGURE 20 – Résultat HDBSCAN avec *min_cluster_size* = 10

De la même façon, nous avons appliqué la méthode sur la figure *spiral* afin de confirmer le bon fonctionnement de HDBSCAN pour des formes non convexes. Les résultats obtenus sont similaires à ceux obtenus avec DBSCAN et confirment nos attentes :

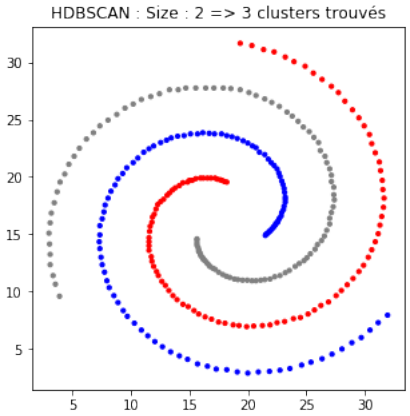

FIGURE 21 – Résultat HDBSCAN avec la figure *spiral*

En règle générale, l'utilisation de HDBSCAN permet d'identifier des clusters qui présentent des densités différentes (c'est la limite atteinte avec DBSCAN). Nous avons donc appliqué ces deux méthodes à notre jeu de données *2d-4c-no4* pour mettre en exergue cet avantage. Voici les meilleurs résultats que nous avons obtenus avec chacune des méthodes pour le même jeu de données :

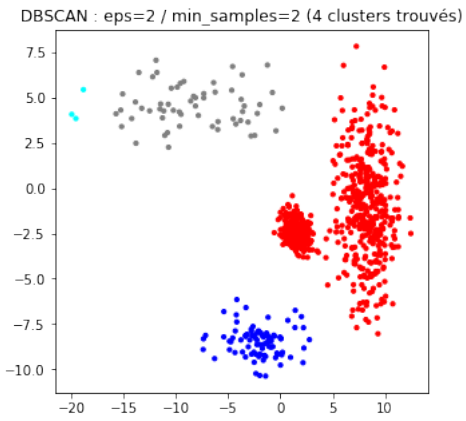

FIGURE 22 – Résultats avec DBSCAN pour des densités différentes.

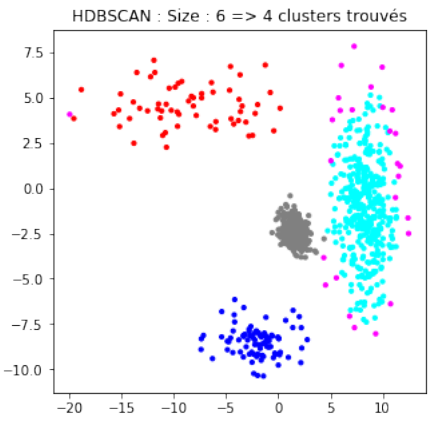

FIGURE 23 – Résultats avec HDBSCAN pour des densités différentes.

Dans les figures ci-dessus, le bruit est toujours représenté en rose. On voit effectivement que

le clustering effectué avec HDBSCAN est plus efficace, alors que nous avons seulement modifié le paramètre *min_cluster_size* et que les autres paramètres ont été automatiquement optimisés par l'algorithme (notamment *min_samples*).

Enfin, concernant le temps nécessaire pour la classification, on pourrait s'attendre à une différence entre les deux méthodes. En réalité, pour des jeux de données de cette taille, la différence n'est pas significative, et le calcul d'une solution nécessite en moyenne 20 millisecondes avec les deux méthodes (mesure effectuée sur le temps effectivement utilisé par le processeur).

3.5 Conclusion

Pour conclure, cette partie sur les méthodes DBSCAN et HDBSCAN a permis de mettre en évidence la principale force de ces algorithmes : la détection et la classification de formes non convexes, en prenant en compte le bruit.

En revanche, la limite de DBSCAN est atteinte rapidement en l'appliquant sur des données avec des densités différentes.

4 Comparaison des quatre méthodes étudiées

Le but de cette section est de reprendre chaque modèle utilisé dans les trois parties précédentes, afin de les comparer et de déterminer le plus adapté à chaque situation.

Afin de comparer au mieux les méthodes, nous avons sélectionné, lorsque c'était possible, les hyper-paramètres optimaux. Cela nous permet donc de comparer les meilleurs résultats de chacune des méthodes.

Note importante : pour être certains que les valeurs présentées dans cette conclusion soient cohérentes, nous avons relancé plusieurs fois les codes décrits dans ce rapport, pour plusieurs données différentes.

4.1 Les méthodes k-Means et clustering agglomératif efficaces pour des données "simples"

4.1.1 Comparaison des résultats

Les méthodes k-Means et clustering agglomératif sont globalement assez proches dans les résultats qu'elles présentent. Notamment, les résultats qu'elles donnent pour des figures qui paraissent "simples" sont très bons :

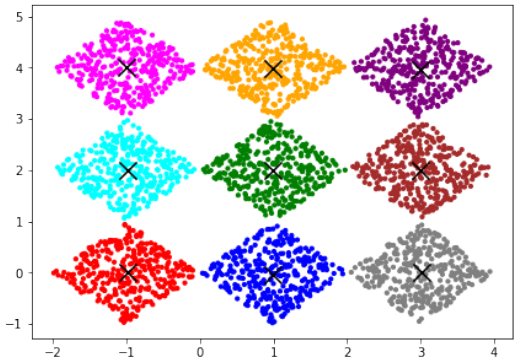

FIGURE 24 – Résultats obtenus avec k-Means pour *diamonds*.

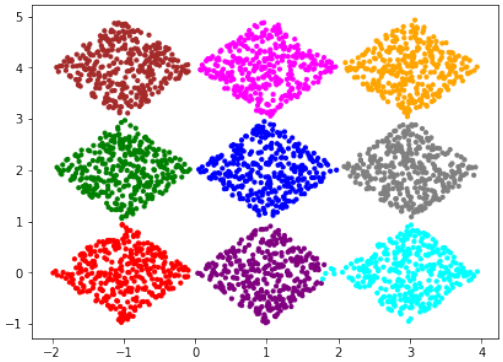

FIGURE 25 – Résultats obtenus avec le clustering agglomératif pour *diamonds*.

4.1.2 Étude des temps d'exécution

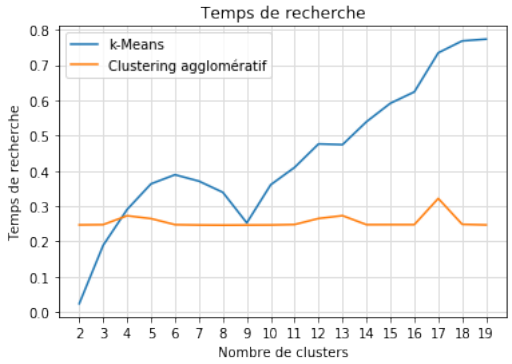

FIGURE 26 – Temps d'exécution de k-Means et du clustering agglomératif (linkage : single).

On remarque que le temps de traitement pour le clustering agglomératif est globalement constant quelque soit le nombre de clusters considérés. Pour k-Means, en revanche, bien qu'on note une amélioration pour les k correct ($k = 9$), le temps d'exécution a tendance à augmenter de façon linéaire en fonction du nombre de clusters recherchés.

Le clustering agglomératif sera donc préférable dans ce genre de situations. Notons toutefois que pour ces mesures, le linkage le plus rapide a été retenu (les résultats sont donc à nuancer, car la qualité du clustering est moins bonne avec cette méthode).

4.1.3 Limites de ces deux méthodes

En revanche, dès que les données sont non convexes ou bruitées, ces deux algorithmes ne sont plus capables de clusteriser correctement le jeu de données. Nous pouvons voir sur les figures ci-dessous que les résultats sont incorrects :

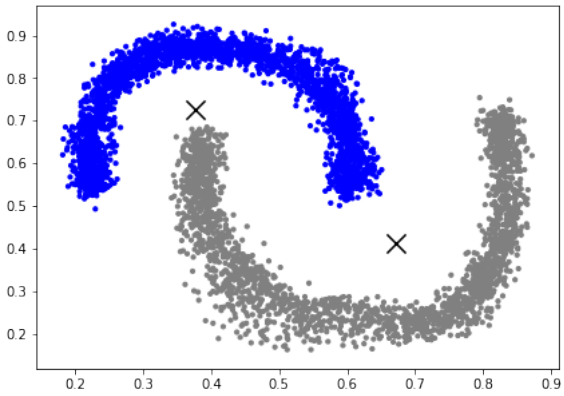

FIGURE 27 – Résultats obtenus avec k-Means pour la figure *banana*.

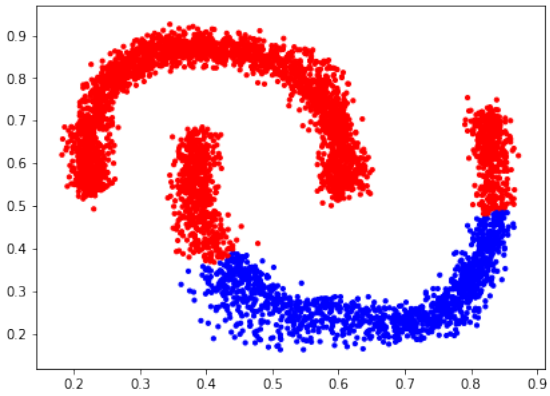

FIGURE 28 – Résultats obtenus avec le clustering agglomératif pour la figure *banana*.

En revanche, ces résultats sont bien meilleurs avec DBSCAN.

4.2 DBSCAN plus performant pour des données non convexes et bruitées

Note : nous considérons dans cette partie que l'algorithme DBSCAN a été réglé avec des paramètres epsilon et min_samples optimaux pour le dataset considéré.

Si nous nous intéressons de nouveau au jeu de données *banana*, DBSCAN permet effectivement de trouver parfaitement les deux clusters. De la même façon, nous avons vu que la détermination précise des hyper-paramètres à l'aide de la méthode décrite précédemment nous permet de totalement identifier le bruit sur une figure :

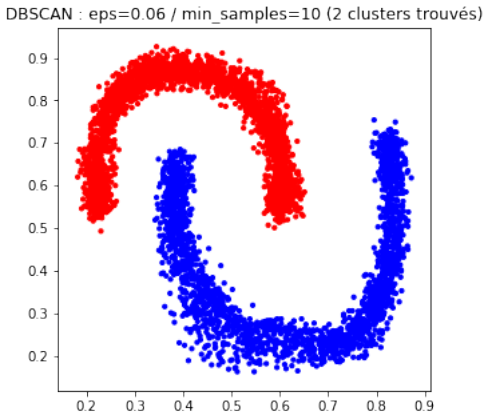

FIGURE 29 – Résultat de DBSCAN avec la figure *banana*.

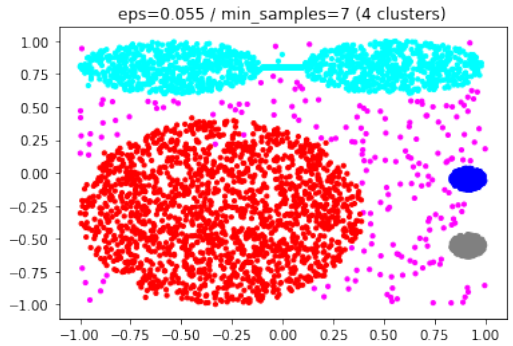

FIGURE 30 – Résultats de DBSCAN avec une figure bruitée.

Cependant, la figure de droite met aussi en évidence les limites de le méthode DBSCAN : elle n'est pas capable de clusteriser correctement des données de densités variables comme c'est le cas ici.

4.2.1 Temps d'exécution de l'algorithme

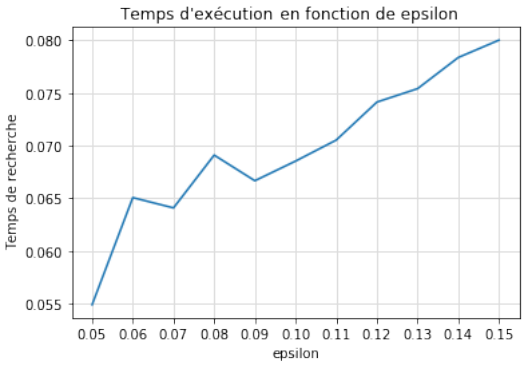

FIGURE 31 – Temps d'exécution de DBSCAN en fonction de *epsilon*.

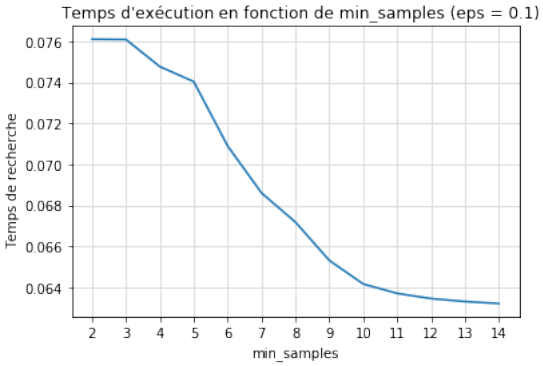

FIGURE 32 – Temps d'exécution de DBSCAN en fonction de *min_samples* ($eps = 0.1$).

On remarque que plus *epsilon* est grand, plus DBSCAN est lent. Cela s'explique par le fait que si *epsilon* est grand, alors deux points sont plus susceptibles d'appartenir au même cluster (donc un plus grand nombre de points seront considérés).

A l'inverse, plus *min_samples* est important, plus le temps d'exécution est petit. En effet, un nombre plus grand nombre de points à considérer entraîne un plus petit nombre de clusters.

4.3 HDBSCAN plus adapté aux données de densités variables

Pour reprendre une dernière fois notre jeu de test *banana*, il est également possible d'obtenir des résultats très satisfaisants avec HDBSCAN.

Dans la conclusion consacrée aux comparatifs entre DBSCAN et HDBSCAN, nous avions vu que HDBSCAN permettait une meilleure détection des formes avec des densités différentes. En revanche, pour le seul jeu de données bruité à notre disposition, HDBSCAN montre ses limites et il est compliqué d'obtenir un bon clustering, même en jouant sur les paramètres *min_cluster_size* et *min_samples*.

Ces deux constats sont visibles sur les figures ci-dessous :

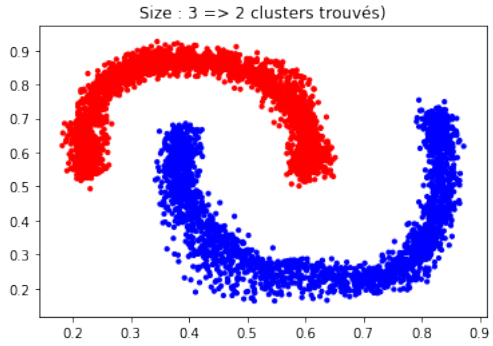

FIGURE 33 – Résultat HDBSCAN avec la figure *banana*.

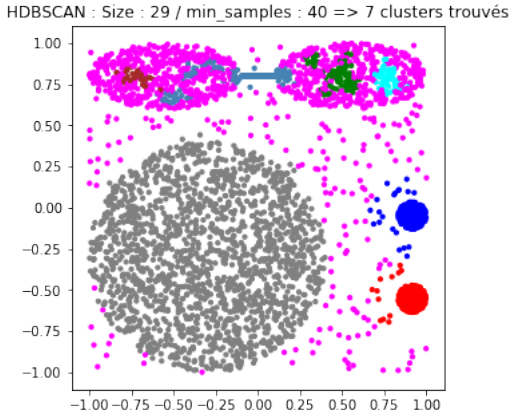

FIGURE 34 – Résultat HDBSCAN avec la figure *CURE*.

Concernant les temps de calcul, sur le jeu *banana*, il faut en moyenne 0.131 secondes pour trouver une solution (en comparaison, le temps nécessaire avec DBSCAN était en moyenne de 0.360 secondes). On note donc une nette amélioration pour ce jeu de données par exemple.