**Rapport DBSCAN**

[**Algorithme**](#_fvqk0jjnbxh1) **1**

[**Heuristiques et matrice de confusion**](#_dvphusih7vhi) **4**

[**Conclusions**](#_hjx921djps8m) **7**

# Algorithme

Pour l’algorithme DBSCAN nous avons créé une classe contenant les attributs suivant :

self.\_points = X *# matrix of variables and their features*

self.\_status = **None** *# Create variable to track the status of each point*

self.\_is\_member = **None** *# Create variable to track membership of each point*

self.\_distance = **None** *# Create variable to store distance between points*

self.\_distance\_idx = **None** *# Create variable to store indexes of ordered neighbors based on distance*

self.\_eps = **None** *# epsilon*

self.\_min\_pts = **None** *# minimum points*

self.\_initialize() *# initialize the values*

self.\_clusters = **None** *# the clusters will be there*

self.\_fig = fig *# linked plot*

*# a color dict to plot the clusters*

self.\_color\_dict = {0: **'#8800FF'**, 1: **'#2F9599'**, 2: **'#999999'**, 3:**'red'**, 4: **'yellow'**, 5: **'blue'**, 6: **'green'**, 7: **'purple'**, 8: **'magenta'**, 9: **'#faebd7'**, 10: **'#2e8b57'**, 11: **'#eeefff'**, 12: **'#da70d6'**, 13: **'#ff7f50'**, 14: **'#cd853f'**,15: **'#bc8f8f'**, 16: **'#5f9ea0'**, 17: **'#daa520'**, 18: **'#ff69b4'**, 19: **'black'**, 20: **'cyan'**}

Les attributs sont initialisé comme ci-dessus dans la méthode constructeur :

**def** \_\_init\_\_(self, X, fig):

*"""*

*Init the object*

**:param** *X: matrix of points*

**:param** *fig: associated plot for the rendering*

*"""*

Nous avons aussi overridé la fonction \_\_str\_\_() pour pouvoir par la suite afficher dans la console les paramètres self.\_eps et self.\_min\_pts de notre objet.

Par la suite nous avons écrit l’algorithme du DBSCAN avec les fonctions suivantes :

**def** \_initialize(self):

*"""*

*Initialize the properties according to the matrix of points*

*"""*

**def** run(self):

*"""*

*DBSCAN algorithm*

*"""*

**def** \_expand\_cluster(self, point, neighbors):

*"""*

*To expand a found cluster*

**:param** *point: point from which the exploration is done*

**:param** *neighbors: its neighbors*

**:return***: the completed expanded cluster*

*"""*

**def** \_region\_query(self, center):

*"""*

*Query for the neighbors of a center point in the epsilon-radius*

**:param** *center: center point*

**:return***: array of neighbors*

*"""*

**def** \_add\_to\_cluster(self, cluster, k):

*"""*

*Add a point to a cluster*

**:param** *cluster: cluster*

**:param** *k: point index*

*"""*

Par contre nous avons effectué un changement par rapport à l’algorithme classique du DBSCAN puisque nous avons jugé plus logique de tester si un point appartenait déjà à un cluster avant de le considérer comme faisant partie d’une région et donc pour ne par biaser la fonction \_region\_query(self, center) puisque sans tester cela nous aurions pu avoir un point considéré comme un core avec dans ses points dans un rayon self.\_eps pouvant être déjà dans un autre cluster. Cela aurait été quelque peu absurde.

**def** \_region\_query(self, center):

*# Create set instead of a list because in this case it would be necessary to iterate through.*

neighbors = set()

i = 0

*# Use presorted/precomputed distances. When the closest accepted neighbors will be*

*# found, iteration will stops*

**while** i < self.\_distance\_idx.shape[1] **and**

self.\_distance[center, self.\_distance\_idx[center, i]] <= self.\_eps:

**if self.\_clusters is None or i not in self.\_clusters:**

neighbors.add(self.\_distance\_idx[center, i])

i += 1

**return** neighbors

Enfin nous avons réalisé la méthode

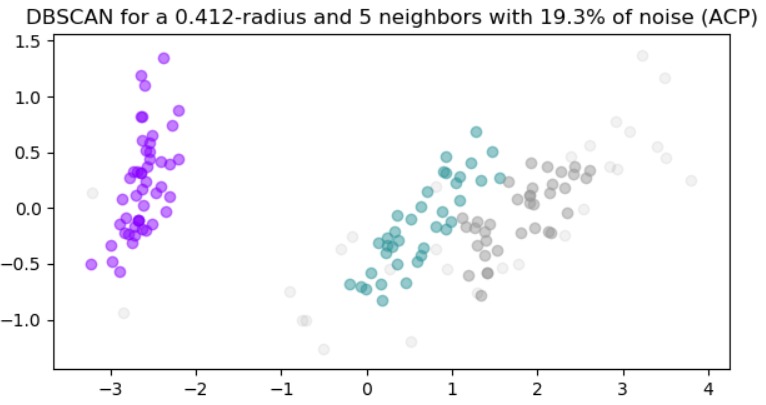
**def** to\_print(self):

*"""*

*To print the result on the plot and on the console*

*"""*

Le problème étant qu’il est facile d’afficher les clusters sur la console mais il est difficile de les représenter graphiquement puisque le nombre de dimension est potentiellement infini pour chaque point. Puisque nous avons affaire à des paramètre à valeur réelle nous avons pu réaliser une ACP à l’aide de la fonction PCA(n\_components=2) importé de sklearn.decomposition. Ainsi nous obtenons un graphique comme ci-dessous. Comme on peut le voir le titre du graphique indique quels ont été les paramètres qui ont aboutit à ce clustering précis. Ce sera aussi le seul moment où l’on présentera le pourcentage de points considérés comme du bruit avec ces paramètres. Ces derniers seront représentés en noir avec une transparence de 0.1. Ils sont donc volontairement difficile à distinguer d’un premier coup d’oeil. Les clusters sont représentés avec les couleurs provenant du dictionnaire attribut self.\_color\_dict.



# Heuristiques et matrice de confusion

Nous avons ensuite travaillé sur une heuristique pour choisir nos paramètres et nous nous sommes dit qu’un bonne heuristique serait une heuristique qui serait relativement directe. De fait nous avons la fonction suivante :

**def** choose\_parameters(self):

*"""*

*To choose heuristically the parameters epsilon and min\_pts*

*"""*

self.\_choose\_eps()

self.\_choose\_min\_pts()

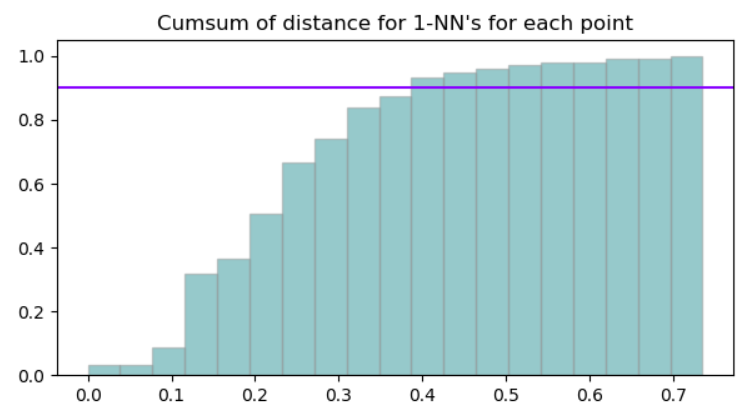
Commençons d’abord par le choix de self.\_eps. Pour cela nous sommes simplement parti du principe que nous allions considérer d’office 10% des points comme étant du bruit. Nous avons donc fait une sorte de 1-Nearest Neighbor en récupérant la distance de cet élément le plus proche pour chaque point. Nous cherchons dont la valeur de rayon qui représente en somme cumulée de fréquence 0.9 comme montré ci-dessous. Dans cet exemple on remarque clairement que la valeur de self.\_eps sera environ de 0.412. Nous avons donc la méthode suivante:

**def** \_choose\_eps(self):

*"""*

*Choose epsilon by a 90/10 principle as a Pareto analysis. We will consider that 90% of the points explain all the data*

*To choose epsilon we will take the distance for each point to its nearest neighbor (1-NN) We will sort these distances and make a distribution histogram to evaluate the value of epsilon, from which we are sure 90% of the points will have at least one neighbor.*

*"""*

Dans la même logique que la méthode précédente pour choisir self.\_eps, nous avons créé la méthode suivante pour établir une valeur de self.\_min\_pts en considérant 50% des points comme des cores.

**def** \_choose\_min\_pts(self):

*"""*

*Choose min\_pts by a 50/50 principle as a Pareto analysis*

*We will consider that 50% has to be a core of a cluster*

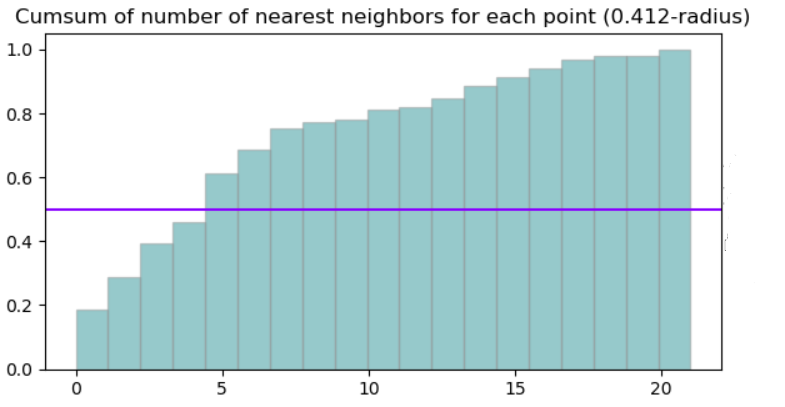
*To choose min\_pts we will take the distance for each point to the other*

*We count how many of these distances are inferior to epsilon*

*and make a distribution histogram to evaluate the value of min\_pts,*

*from which we are sure 50% of the points will be a core of a cluster.*

*"""*



Enfin pour tester si notre algorithme fonctionne correction nous avons créé la méthode

**def** confusion(self, y):

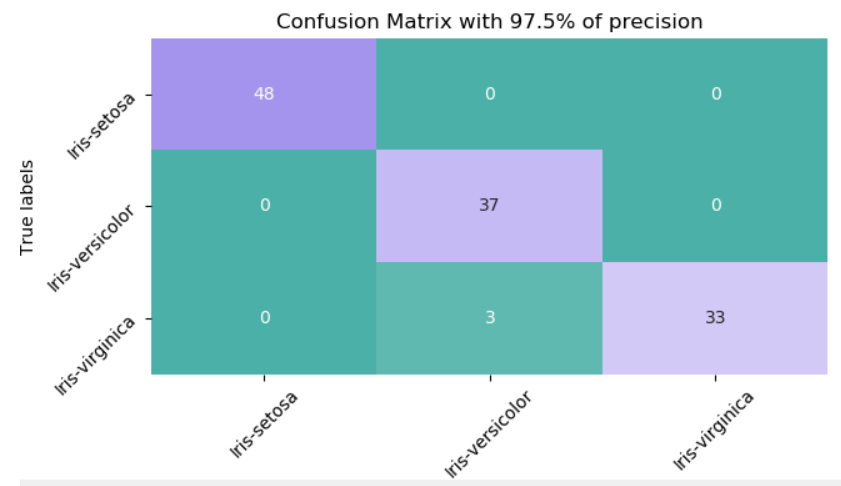
*"""*

*Display a confusion matrix ONLY IF WE KNOW THE LABELS y (tests)*

**:param** *y: labels*

*"""*

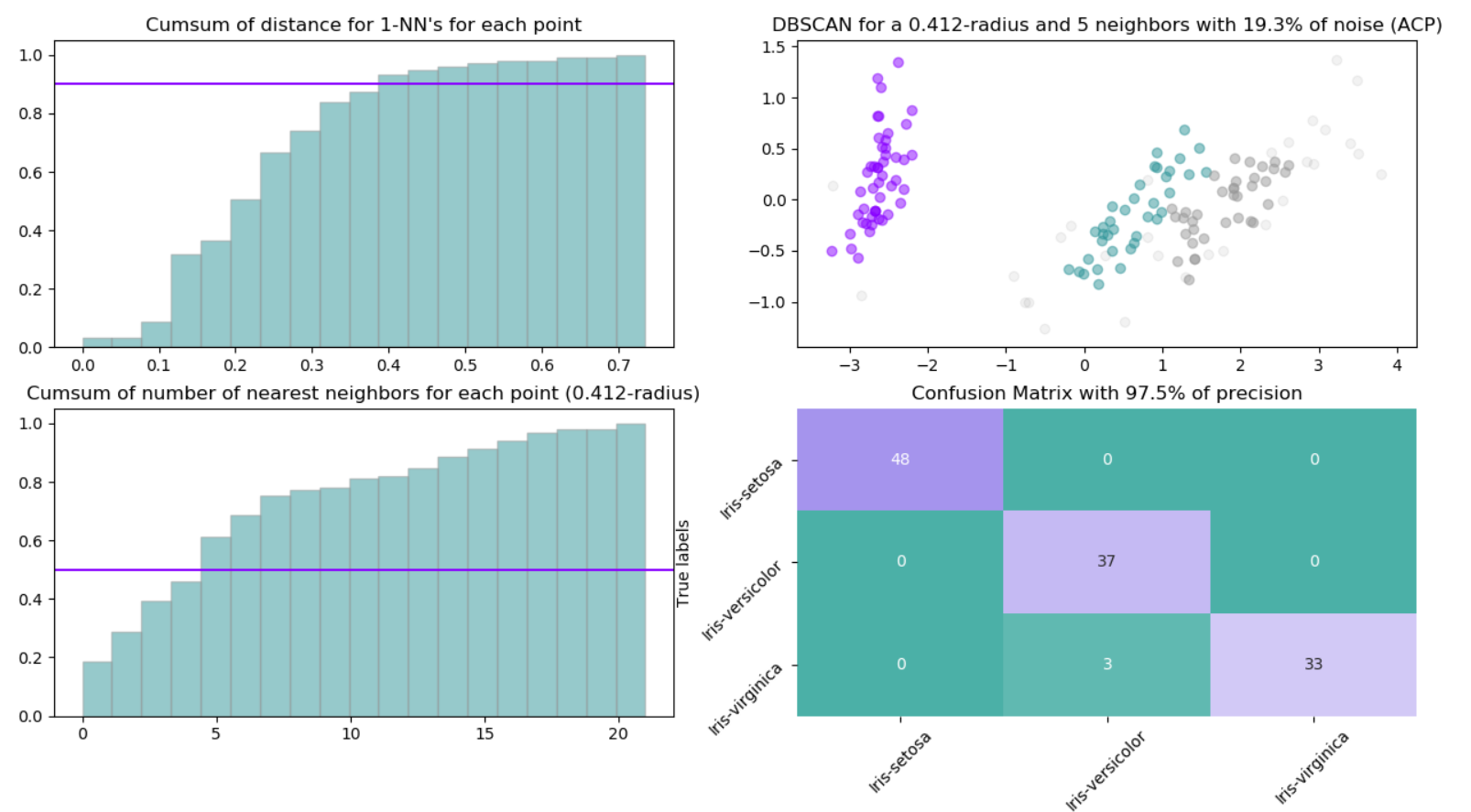
que l’on peut utiliser en “trichant” un peu. En effet DBSCAN est un algorithme non supervisé et donc intrinsèquement nous ne connaissons les catégories des points. Cependant dans notre cas nous connaissons déjà ce vers quoi notre algorithme doit tendre. Ainsi nous pouvons construire une matrice de confusion entre les clusters créés par DBSCAN et les catégories réelles. Pour ce faire nous allons considérer qu’un cluster sortant du DBSCAN prend comme catégorie le mode des catégories réelles des points du cluster. Pour créer notre matrice de confusion nous avons utilisé la librairie seaborn het cela nous donne le résultat ci-dessous ainsi que le pourcentage de précision.



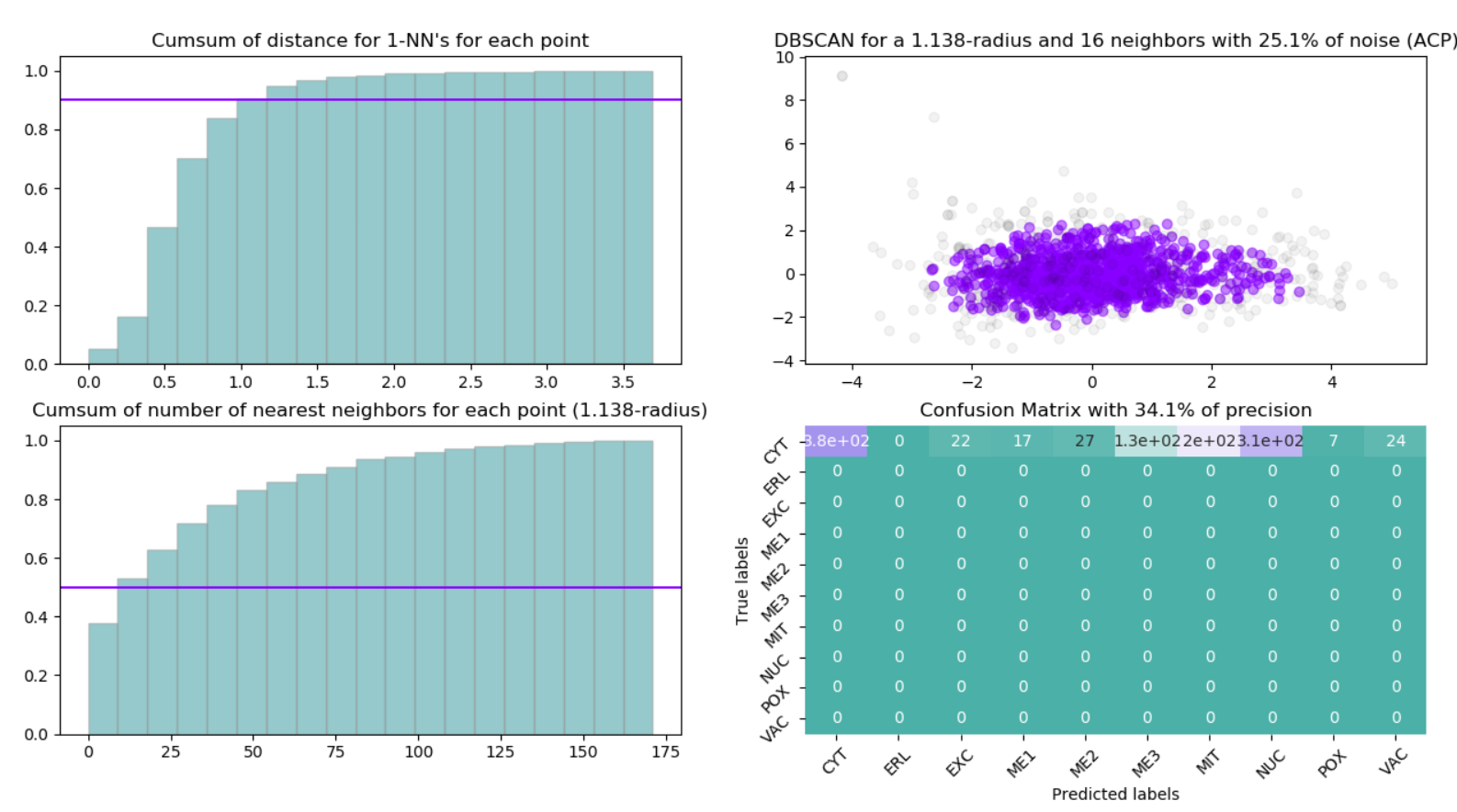
# 

# Conclusions

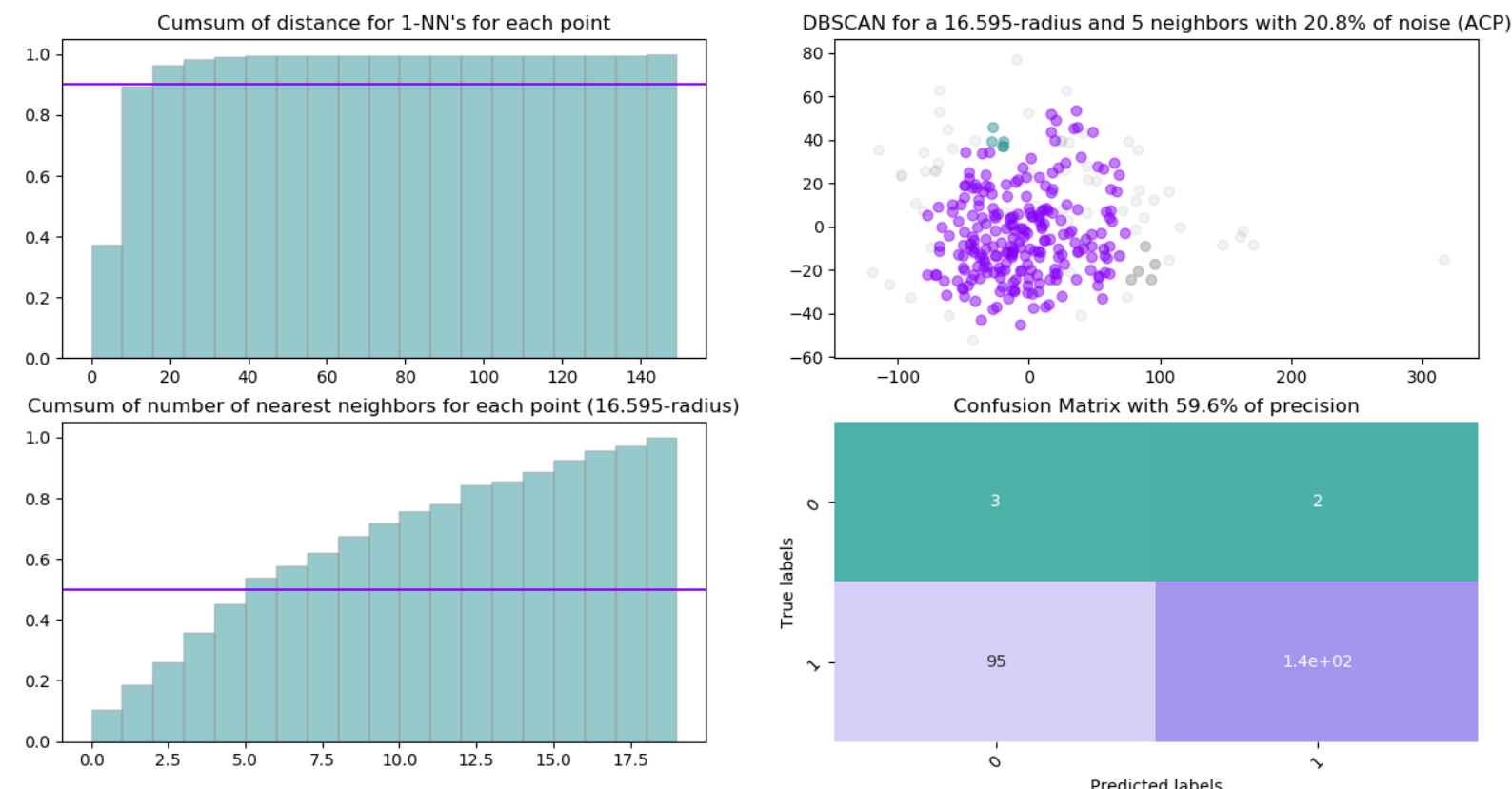
Nous pouvons admettre que notre heuristique est quasi parfaite quant à nos données s’il s’agit de viser une précision la plus forte possible peu en utilisant un nombre de points suffisant. En effet seuls 19.% des points, comme on peut le voir ci-dessous, sont considérés comme étant du bruit pour une précision sur les points appartenant à des clusters des 97.5%. C’est à se demander si nous n’avons pas révolutionné les heuristiques du DBSCAN avec seulement de simples analyses de Pareto 90/10 et 50/50 !! (Joke…)



Nous avons donc souhaité tester ces heuristiques sur d’autres données disponibles sur internet et nous avons aléatoirement choisi les données *yeast.data*. On peut clairement voir que non, nous n’avons pas révolutionné l’algorithme DSCAN. Nous ne savions pas quel poids accorder aux différentes variables nous avons donc centré et réduit les données. Avec ce jeu de donnée nous avons 25.1% de bruit pour une précision de 34.1% ce qui est sans valeur ajoutée.



De même nous avons testé avec les données heart.data que nous avons centré réduit. Avec ce jeu de donnée nous avons 20.8% de bruit pour une précision de 59.6% ce qui est encore une fois sans valeur ajoutée. Nous en avons conclu que ces heuristiques étaient très très mauvaise lorsque les données sont très compactes.



C’est pourquoi pour nous conforter dans l’avis que nos heuristiques sur l’algorithme DSBCAN sont meilleures sur des données moins denses nous avons testé un nouvel et dernier jeu de donnée. On peut voir ici que nous avons 21.7% de bruit et une précision de 74.9%. C’est nettement moins précis que pour le jeu de donnée des iris mais cela est relativement satisfaisant dans ce dernier cas bien qu’il pourrait y avoir un peu trop de clusters qui en ressortent.

