Practical Work 1: Clustering

Unsupervised and Reinforcement Learning

Guillermo Creus Botella

MSc in Artificial Intelligence

Barcelona, May 4, 2022

Contents

1 Introduction

One of the main purposes of Unsupervised Learning is to pull insights from unlabelled data. One way to do this is to group data into clusters with similar properties. As previously mentioned, this AI discipline deals with data which does not have labels. Consequently, there is not a clear cut solution in how to correctly classify data into different categories.

In [10], the authors lay out the main differences between Supervised and Unsupervised Learning: cost functions. In the former, one can carefully design a function that maps predictions and labels into a real number that encodes how close the predictions are to the labels. However, in unsupervised learning labels are not provided during the training phase so unsupervised cost functions do not know which of the many possible supervised tasks one cares about.

Despite the previous point, in some situations unsupervised is the only way to go forward. That is why their importance should not be overlooked and in this paper an effort will be made to prove that. The main objective of this work is to implement a clustering algorithm in Python following the Scikit-learn API conventions [9], reproduce the results from the paper that presented the algorithm and compare against known methods from the literature.

In particular, this paper is gonna analyze DBSCANN++ [5], which is an improvement over the density-based method DBSCAN [4]. Following the Python implementation and the adaptation to a Scikit-learn Mixin instance (class for all cluster estimators in Scikit-learn), the algorithm will be analyzed in different datasets using the most relevant metrics.

1.1 Task

The task that DBSCANN++ is trying to solve is clustering. It consists of the following: given a matrix $X = \{x_1, x_2, \dots x_N\}$with D columns and N rows – shape (n, D) –, the objective is to create a partition $U = \{U_1, U_2, \dots, U_R\}$ into R clusters.

At this stages there are two steps left. If one has the actual labels of each instance, several metrics can be computed so as to assess the differences between the ground truth clustering and the predicted clustering. In addition, one can also compute metrics that encode the differences in intra-cluster and inter-cluster distance. All of these will be covered in the next section; 1.2.

1.2 Metrics

In the literature the type of metrics internal validation which uses the internal information of the clustering process to evaluate the goodness of a clustering structure without reference to external information[3]. The three metrics that will be used mainly encode the goodness of a clustering: high separation between clusters and a high cohesion within clusters. They are described in [8]:

- **Silhouette coefficient** is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette ranges from -1 to 1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters. For each entity in our cluster Silhouette coefficient is determined by the following expression:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \tag{1}$$

  Where $a(i)$ is the mean distance intra-cluster and $b(i)$ the mean distance of i to all points in any other cluster, of which i is not a member. Then, the global silhouette coefficient is just the average of the particular silhouette coefficients for each example:

$$S \equiv \frac{1}{N} \sum_{i=1}^{N} s(i) \tag{2}$$

- **Calinski-Harabasz coefficient** is the ratio of the sum of inter-clusters dispersion and of intra-cluster dispersion for all clusters, so the higher the score, the better the clustering. The score is determined by the

following expression:

$$CH \equiv \frac{\frac{SSB_M}{(M-1)}}{\frac{SSE_M}{M}} \tag{3}$$

where SSE is the sum of squares of intra-cluster distances, SSB is the sum of squares of inter-cluster distances and M is number of clusters.

- **Davies-Bouldin coefficient**: Let $R_{i,j}$ be a measure of how good the clustering scheme is, where low means best. It is defines as a function of $M_{i,j}$, the separation between the ith and the jth cluster (large is best), and S_i, intra-cluster distance (low is best). This leads to $D_i \equiv \max_{j \neq i} R_{i,j}$ and the Davies-Bouldin coefficient (DBI):

$$DBI \equiv \frac{1}{M} \sum_{i=1}^{M} D_i$$

Where M is the number of clusters.

Although the previous metrics are used in the literature to determine the number of clusters by computing the scores for every number of cluster and getting the one that performs best, they assume that the algorithm can be tuned for the number of clusters. In DBSCAN the number of clusters is given by the algorithm itself so these measures are not as useful. However, they will still serve a purpose when the real labels are unknown and all of the previous metrics will be reported in the final results.

On the other hand, by using the true labels one can compute what in [8] is defined as **external validation**. It consists of comparing the results of a cluster analysis to an externally known result, such as externally provided class labels. Since the ground truth is known, this approach is mainly used for selecting the right clustering algorithm for a specific data set.

In this case it is important to set some notation. Given a set X of N elements $X = \{x_1, x_2, \dots x_N\}$, consider two partitions of X, namely $U = \{U_1, U_2, \dots, U_R\}$ with R clusters, and $V = \{V_1, V_2, \dots, V_C\}$ with C clusters. Then, one can define the Mutual Information (MI) index as:

$$\mathrm{MI}(U, V) = \sum_{i=1}^{R} \sum_{j=1}^{C} P_{UV}(i, j) \log \frac{P_{UV}(i, j)}{P_U(i) P_V(j)}$$

where $P_{UV}(i, j)$ denotes the probability that a point belongs to both the cluster U_i and cluster V_j: $P_{UV}(i, j) = \frac{|U_i \cap V_j|}{N}$, and $P_U(i) = \frac{|U_i|}{N}$. However, the score that will be used in this work is the Adjusted Mutual Information Index:

$$AMI(U, V) = \frac{MI(U, V) - \mathbb{E}[MI(U, V)]}{\max\{H(U), H(V)\} - \mathbb{E}[MI(U, V)]}$$

where $\mathbb{E}[Z]$ is the expected value of a r.v. Z and $H(W)$ is the entropy of a partition.

Lastly, in order to introduce the Adjusted Rand Index (ARI), one must define:

- True Positive (TP): The clustering algorithm predicts that a pair of instances belong to the same cluster and their true labels match.
- False Positive (FP): The clustering algorithm predicts that a pair of instances belong to the same cluster but their true labels do not match.
- True Negative (TN): The clustering algorithm predicts that a pair of instances do not belong to the same cluster and their true labels do not match.
- False Negative (FN): The clustering algorithm predicts that a pair of instances do not belong to the same cluster but their true labels match.

Noting that $TP + FP + TN + FN = \binom{N}{2}$, the Rand Index ($RI$) is defined as:

$$RI = \frac{TP + TN}{TP + FP + FN + TN}$$

And the Adjusted Rand Index (ARI):

$$ARI = \frac{RI - \mathbb{E}[RI]}{\max\{RI\} - \mathbb{E}[RI]}$$

2 Methods

Density-based clustering is a clustering technique that assigns clusters to areas that have a higher density [6]. Since one of the main focus of this work is the comparison of the density-based clustering methods, the scope of this report will be restricted to the following algorithms: DBSCAN [4], DBSCAN++ [5] and OPTICS [1].

2.1 DBSCAN

Starting with DBSCAN, one can see the pseudo code in figure 1. Given a data matrix X, and hyperparameters ϵ, minPts it is initialized by finding the core-points in X. These points are the points $x \in X$ such that:

$$|B(x, \epsilon) \cap X| \geq \text{minPts where } B(x, \epsilon) \equiv \{x' : ||x - x'|| \leq \epsilon\}$$

Algorithm 1 DBSCAN

Inputs: X, ε, minPts
$C \leftarrow$ core-points in X given ε and minPts
$G \leftarrow$ initialize empty graph
for $c \in C$ **do**
  Add an edge (and possibly a vertex or vertices) in G from c to all points in $X \cap B(c, \varepsilon)$
end for
return connected components of G.

Figure 1: DBSCAN pseudo code

Once the most dense points (core-points) are extracted, C, it is time to add an edge from every core-point c to all points in $B(c, \epsilon) \cap X$. Note that the graph G is a directed graph and that the algorithm would not work if the graph would not be directed. Since the clusters are the connected components, the paths will have to pass through the core-points. So, if two core-points are in the same neighbourhood, then they will be assigned the same cluster.

2.2 DBSCAN++

As it was seen in section 2.1, the DBSCAN algorithm samples the core-points C from the data matrix X. However, as mentioned in [5], this is a time consuming task since its worst case complexity is $O(N^2)$. In order to reduce the time complexity, DBSCAN++ (see figure 2 for the pseudo code) will restrict the core-points search to a reduced data set S of size m. Therefore, in order to determine this set S one needs an extra hyperparameter m and the sampling method.

The sampling methods that will be explored are the following:

- Uniform: choose m points uniformly from X without replacement.
- K-centers: starting from one point, the goal is to add points to S sequentially. These points are the furthest from the set S, and as it is shown in figure 3, one should repeat this step until m points are retrieved.

It is extremely important to understand that in the case of $m = N$, the sampling will cover all of the points of X, and consequently, $S = X$ so DBSCAN++ will be equal to DBSCAN.

Algorithm 2 DBSCAN++

Inputs: X, m, ε, minPts
$S \leftarrow$ sample m points from X.
$C \leftarrow$ all core-points in S w.r.t X, ε, and minPts
$G \leftarrow$ empty graph.
for $c \in C$ **do**
  Add an edge (and possibly a vertex or vertices) in G from c to all points in $X \cap B(c, \varepsilon)$
end for
return connected components of G.

Figure 2: DBSCAN++ pseudo code

Algorithm 3 Greedy K-center Initialization

Input: X, m.
$S \leftarrow \{x_1\}$.
for i from 1 to $m - 1$ **do**
  Add $\text{argmax}_{x \in X} \min_{s \in S} |x - s|$ to S.
end for
return S.

Figure 3: K-centers sampling pseudo code

2.3 OPTICS

Before exposing this method, it is important to define three concepts:

- **Number of neighbours:** $N_\epsilon(p) = |B(p, \epsilon) \cap X|$
- **Core distance:** $\text{core-dist}_{\varepsilon, MinPts}(p) = MinPts$-th smallest distance in $N_\epsilon(p)$, if $N_\epsilon(p) \geq \text{minPts}$, otherwise it is undefined.
- **Reachability distance:** $\text{reachability-dist}_{\epsilon, minPts}(o, p) = \max(\text{core-dist}_{\varepsilon, MinPts}(p), \text{dist}(p, o))$ if $N_\epsilon(p) \geq \text{minPts}$, otherwise it is undefined.

Having mentioned the previous concepts, OPTICS starts by computing the core distance and reachability distance of all points. The points are then arranged in an index such that the ones that have a lower reachability distance to each other are grouped closer together. Thanks to this information one can obtain the reachability plot, which is key to identify the clusters via this method. The clustering is decided using a parameter ξ that defines how much must the reachability distance change between two consecutive indices to decide that the frontier of the cluster has been reached.

3 Data

Regarding the data used in this paper, an effort was made to study artificial and real datasets. Regarding the real datasets, most of them were taken from the paper [5] in order to reproduce the results. Since their study was exhaustive, the majority of well known clustering datasets were used. However, in order to introduce some novelty, an artificial dataset is going to be analyzed which will test how the algorithms behave when non linearly separable data is presented.

In table 1 one can see the information of the real datasets used to test the methods presented in section 2, where N is the number of instances, D the number of features and c the number of clusters/classes. The datasets (A), (B), (C), (F), (H), (I) and (J) have been taken from the UCI ML repository [2], while (K) has been obtained from Stanford's "Stanford Large Network Dataset Collection" [7].

In order to generate the artificial dataset, `sklearn`'s function `make_circles` was used. It generates a 2D dataset as seen in figure 4. This toy dataset will be generated with the following parameters:

Table 1: Real datasets information

	N	D	c
(A) Iris	150	4	3
(B) Wine	178	13	3
(C) Spambase	4,601	57	2
(F) Libras	360	90	15
(H) Zoo	101	16	7
(I) Seeds	210	7	3
(J) Letters	20,000	16	26
(K) Phonemes	4,509	256	5

- `n_samples` = 200
- `factor` = 0.5, scale factor between inner and outer circle.
- `noise` = 0.05, standard deviation of Gaussian noise added to the data.

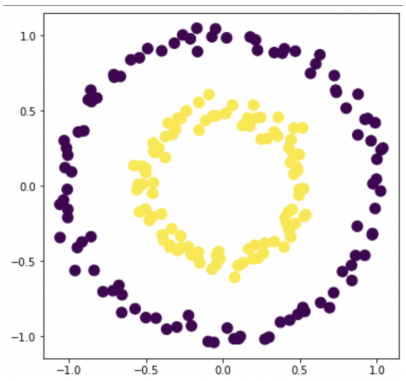


Figure 4: Toy dataset generated by make_circles function

By doing this, after the results from real datasets are reproduced, one will be able to visualize a dataset and check how good the methods can cluster non linearly separable data.

To end this section, it is important to highlight that all of the datasets will be normalized. That is, every feature in the data matrix will be transformed so that it is 0 mean and it has a standard deviation of 1. This has been done with the intention of keeping the importance of all features equal. To be more precise, if the range of one feature is not comparable to the other one, DBSCAN will be affected, since the core points are based on a radii ϵ.

4 DBSCAN++ implementation

In the Appendix 7.1 one can see the Python code of the DBSCAN++ implementation. It has been developed using `ClusterMixin`, a Mixin class for all cluster estimators in scikit-learn, and `BaseEstimator`, the base class for all estimators in scikit-learn. The only methods that must be developed for this child class are the constructor and `fit_predict`. The latter will take a data matrix and the sampling method and it will output the predicted labels.

It is important to note that K-dimensional trees have been used for this implementation to be as efficient as possible. Although the original authors quote their timing results using Cython, a superset of the Python language that supports some C usage which is considerably faster than Python, an effort has been made in order to be competitive nonetheless. What is more, the first approach used vectorized functions and it was nowhere near the performance of DBSCAN++ with KDTrees.

Regarding the complexity of the algorithm, the algorithm was developed with the same methods and pseudo code as in the original paper. Therefore, the quoted complexity does not change, and as proved in [5] it is $O(nm)$.

5 Results

In figure 5 one can see the Adjusted Rand Index and Adjusted Mutual Information for different values of ϵ. Note that the choice of m, for every method and ϵ is given by the maximum over the ones obtained using $m = p \cdot N^{\frac{D}{D+4}}$, where $p \in \{0.1, 0.2, 0.3\}$. This value comes from [5], where they use this heuristic to obtain the best compromise between speed and performance. Also, figure 5 is consistent with the results shown in the paper. An important remark is that the K-Center and Uniform graphs are translated to the right. That is, their maximum point is achieved at a much higher ϵ than the regular DBSCAN. That means that the cutoff ϵ, i.e., the value that makes all of the instances clustered into one single cluster, is higher.

Apart from that, figure 5 confirms what it was observed in [5]; DBSCAN++ is much more robust to hyperparameters than DBSCAN. In other words, there is a much wider range of ϵ that achieve higher values of Adjusted Rand Score and Adjusted Mutual Information Score. Consequently, the inverted V shape of the DBSCAN plot is much narrower than DBSCAN++ with Uniform and K-Center sampling. Also, it is important to repeat that in all of the datasets except (H) and (K), DBSCAN++ achieves a higher score. It could be that in those datasets a more precise sampling was needed.

On the other hand, in figure 6 one can see the effect that sampling precision has in the performance, execution times and outlier detection of the selected algorithms. First of all, the results coincide with the ones observed in the original DBSCAN++ paper, so everything points out to the correct reproduction of results.

The execution time is plotted in the second column (in millisencods) for every dataset for the DBSCAN++ with uniform sampling. As one can see, the trend is close to linear, which is consistent with the analysis of the time complexity of $O(mn)$. Therefore, it is linear in m/n.

Regarding the sampling size m, one would expect for a higher sampling to imply a higher Adjusted Rand Index or Adjusted Mutual Information scores. However, in datasets (B), (C), (F), (I) and (J) one can observe that the maximum is achieved well before $m/n = 1$, i.e., DBSCAN++ is performing a DBSCAN clustering ($m = N$). This proves that lower values of m/n can actually achieve better scores and less runtime, which is a win from a performance and timing perspection.

In the first column of figure 6 one can see a dotted line, which indicates the number of outliers with a DBSCAN clustering, and a purple line, which indicates the number of outliers of a DBSCAN++ clustering with a uniform sampling. Obviously, when $m/n = 1$, DBSCAN++ is going to do a DBSCAN clustering so the number of outliers will be the same. There is a point where the DBSCAN++ outliers stays close to the DBSCAN dotted line, however one needs to assume that a larger number of outliers will be outputted when a lower value of m/n is used. This confirms the fact that a lower value of m/n can be very beneficial to the performance and execution time of the chosen configuration.

Having analyzed the performance of the developed algorithms, it is time to provide the best performance they can achieve. To obtain the best performance, a subscore was defined and it was optimized over a wide range of ϵ and, when relevant, $m = p \cdot N^{\frac{D}{D+4}}$, where $p \in \{0.1, 0.2, 0.3\}$. The configuration that is displayed it is the one that achieves the highest mean of ARI and AMI. Even though this is not the best way to do multi objective optimization, this task is illustrative and by doing thing this way one can make sure that the configuration will try to increase equally both scores.

In table 2 one can see the ARI, AMI and configuration used in every algorithm. Also, for the sake of visibility, green colored cells are the winners in that parameter and the orange colored ones the respective runner-up. It is clear that the family of DBSCAN algorithms is superior than OPTICS with 13 wins out of 16. However, it is still important to analyze what has happened inside the DBSCAN family. DBSCAN++ K-Center is the clear winner since it beats DBSCAN 11 out of 16 times and DBSCAN++ Uniform 12 out 16 times. The runner-up is DBSCAN++ Uniform with 10/16 wins over DBSCAN. In addition, it is interesting to see that the chosen ϵ always follows this pattern: $\epsilon_{\text{DBSCAN}} < \epsilon_{\text{DBSCAN++ Uniform}} < \epsilon_{\text{DBSCAN++ K-Center}}$.

Lastly, in table 3 one can see the execution times of the DBSCAN family for their best configuration. Note that OPTICS is missing, since the implementation was not developed by the author and the same conditions can not

be met. Also, DBSCAN has been used with the custom implementation but modifying so it can take $m = N$ as a parameter. That way, the algorithms will be compared under the same conditions. However, one can not extract a lot of conclusions from this table as the ϵ and m might be different so the number of computations will be different. However, one can analyze it from the point of view of the amount of time one needs to achieve the highes performance possible from that algorithm.

Having said that, it is evident that the clear winner is DBSCAN++ Uniform, always achieving lower runtimes. This is a consequence of having and ϵ that is not as large as the K-Center sampling, and the lack of initialization in the sampling. Also, K-Center seems to have mixed results compared to DBSCAN since sometimes is slower and sometimes faster. However, the norm is that the larger the dataset the slower K-Center is, probably due to the initialization. This contrasts the results from the original paper, which states that K-Center is usually the fastest. However, they have not stated how they get to the best configuration. If they were using the same configuration for all three methods, it would seem reasonable to accept it, but with the data in hand, it is hard to judge.

As a final note, the results obtained in the artificial dataset can be seen in figures 7 and 8. Via a dotted line one can see the ground truth, and the algorithm's lines are a result of optimization over a given ϵ and $m \in \{20, 30, 40\}$. As it can be seen, DBSCAN is the only method able to achieve the same performance of the ground truth when $\epsilon = 0.4$. The problem with the other methods is that they overfit the data. As it can be seen in figure 7, there are certain values of ϵ where the score is better but when looking at figure 8 one sees that the AMI and ARI are lower. At closer inspecting, some are designing up to 10 clusters in a dataset where there are only two categories. This proves that when there is not a clear point with high density, oversampling is best, as DBSCAN does.

Table 2: Scores of the different algorithms and their configuration – $minPts = 10$)

Dataset	DBSCAN			DBSCAN++ Uniform				DBSCAN++ K-Center				OPTICS		
	ARI	AMI	ϵ	ARI	AMI	m	ϵ	ARI	AMI	m	ϵ	ARI	AMI	ϵ
(A)	0.5344	0.6396	0.92	0.4919 (± 0.03)	0.5815 (± 0.02)	1	2.57	0.6161	0.6109	3	3.39	0.3966	0.5990	1.74
(B)	0.3149	0.4038	2.37	0.5044 (± 0.14)	0.5833 (± 0.09)	5	3.90	0.6097	0.6386	5	6.20	0.2249	0.3336	2.37
(C)	0.0104	0.0978	1.92	0.0305 (± 0.01)	0.0611 (± 0.00)	793	3.74	0.0213	0.0869	793	5.57	0.0329	0.1190	9.21
(F)	0.0760	0.2499	5.23	0.1102 (± 0.02)	0.3339 (± 0.04)	56	6.78	0.2985	0.5265	28	9.87	0.0202	0.1749	8.32
(H)	0.8358	0.8049	3.53	0.8028 (± 0.02)	0.7560 (± 0.03)	12	4.44	0.8645	0.7760	8	5.36	0.3835	0.6071	4.44
(I)	0.3497	0.4175	0.99	0.4403 (± 0.08)	0.5120 (± 0.04)	6	2.37	0.4291	0.5199	6	3.06	0.072	0.2463	0.99
(J)	0.0272	0.3103	1.41	0.0486 (± 0.00)	0.3920 (± 0.02)	551	2.07	0.0932	0.4930	551	2.72	0.0016	0.1722	1.41
(K)	0.4685	0.6334	9.0	0.4698 (± 0.04)	0.6004 (± 0.04)	396	10.0	0.1447	0.3917	396	11.0	0.4697	0.6354	9.00

Table 3: Execution time (in milliseconds) of the different algorithms

	DBSCAN	Uniform	K-Center
(A)	6.02 (± 0.1)	0.58 (± 0.01)	0.75 (± 0.02)
(B)	2.84 (± 0.15)	0.73 (± 0.02)	1.31 (± 0.09)
(C)	1269.4 (± 12.9)	335.25 (± 9.5)	3297.97 (± 37.12)
(F)	19.47 (± 0.09)	4.80 (± 0.10)	25.92 (± 0.66)
(H)	3.01 (± 0.06)	0.65 (± 0.02)	1.33 (± 0.12)
(I)	3.87 (± 0.06)	0.77 (± 0.02)	1.33 (± 0.03)
(J)	4970.57 (± 24.79)	224.65 (± 1.5)	4300.48 (± 73.05)
(K)	6867.71 (± 35.27)	784.1 (± 5.6)	11063.68 (± 14.92)

6 Conclusions

In conclusion, this report has been a success since all of the initial objectives have been met. That is, DBSCAN++ has been implemented following sklearn API conventions, results from the original paper have been reproduced and they have been extended to another real dataset (extended version of Spambase) and an artificial one.

Regarding the performance of the algorithms, OPTICS was nowhere near the DBSCAN family's performance. Also, it is clear that if one needs performance one should go with DBSCAN++ with a sampling method of K-Center. However, as it was seen in the results section, it tends to use a large ϵ, which makes it slow when the size of the dataset increases.

Therefore, if one wants to improve the performance of DBSCAN while still maintaining a better time execution, the uniform sampling is better. In the end, with this method one can achieve better results than DBSCAN with a lower execution time, which is a win win situation.

However, one should bear in mind that under sampling is dangerous. As shown in the artificial dataset, when there are not a lot of points with high density, DBSCAN++ tends to overdetect clusters, providing an undesirable clustering. Therefore, one should inspect the problem before deciding to apply DBSCAN++.

Lastly, as future work, it would be interesting to implement this algorithm with Cython, which would make the implementation closer to the author's. Also, implementing it with GPUs would allow to test it in bigger datasets and see how DBSCAN++ scales.

References

[1] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. *ACM Sigmod record*, 28(2):49–60, 1999.

[2] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[3] Scott Emmons, Stephen Kobourov, Mike Gallant, and Katy Börner. Analysis of network clustering algorithms and cluster quality metrics at scale. *PloS one*, 11(7):e0159161, 2016.

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[5] Jennifer Jang and Heinrich Jiang. Dbscan++: Towards fast and scalable density clustering. In *International conference on machine learning*, pages 3019–3029. PMLR, 2019.

[6] Hans-Peter Kriegel, Peer Kröger, Jörg Sander, and Arthur Zimek. Density-based clustering. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 1(3):231–240, 2011.

[7] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[8] Julio-Omar Palacio-Niño and Fernando Berzal. Evaluation metrics for unsupervised learning algorithms. *arXiv preprint arXiv:1905.05667*, 2019.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[10] Ilya Sutskever, Rafal Jozefowicz, Karol Gregor, Danilo Rezende, Tim Lillicrap, and Oriol Vinyals. Towards principled unsupervised learning. *arXiv preprint arXiv:1511.06440*, 2015.

7 Appendix

7.1 Implementation

```
class DBSCANPP(ClusterMixin, BaseEstimator):

    def __init__(self, m, epsilon=0.5, minPts=10):
        self.m = m
        self.epsilon = epsilon
        self.minPts = minPts

    def k_centers(self, X):
        ind_to_add = 0
        inds_S = [ind_to_add]

        dists_X_to_S = np.full((len(X), 1), np.inf)

        for _ in range(self.m - 1):
            tree = KDTree(X[[ind_to_add]])
            new_dists, _ = tree.query(X, k=1, return_distance=True)
            dists_X_to_S = np.minimum(dists_X_to_S, new_dists)

            ind_to_add = np.argmax(dists_X_to_S)
            inds_S.append(ind_to_add)
        inds_S = np.array(inds_S)

        return inds_S

    def connected_components(self, inds_C, X):
        clusters = np.full(len(X), -1, dtype=np.int32)
        if not len(inds_C):
            return clusters

        tree_C = KDTree(X[inds_C])
        C_neighbors_C = tree_C.query_radius(X[inds_C], r=self.epsilon)
        convert_indices_C = {val: ind for ind, val in enumerate(inds_C)}
        stack = deque()
        current_cluster = 0

        for ind_c in inds_C:
            if clusters[ind_c] == -1:
                stack.append(ind_c)

                while len(stack):
                    node = stack.pop()

                    for neighbor in C_neighbors_C[convert_indices_C[node]]:
                        neighbor = inds_C[neighbor]
                        if clusters[neighbor] == -1:
                            stack.append(neighbor)
                            clusters[neighbor] = current_cluster

                current_cluster += 1

        dist_X_to_C, closest_C_from_X = tree_C.query(X, k=1)
        closest_C_from_X_conv = inds_C[closest_C_from_X.flatten()]

        for i in range(len(X)):
            if clusters[i] == -1:
                clusters[i] = clusters[closest_C_from_X_conv[i]]
```

```
        clusters[dist_X_to_C.flatten() > self.epsilon] = -1

        return clusters

    def fit_predict(self, X, sampling='uniform', seed=2):
        X = check_array(X)
        self.X_ = X

        np.random.seed(seed)
        if sampling == 'uniform':
            inds_S = np.random.choice(len(X), self.m, replace=False)
        elif sampling == 'k-center':
            inds_S = self.k_centers(X)
        else:
            inds_S = np.arange(len(X))

        tree_X = KDTree(X)
        dists_S_to_X, _ = tree_X.query(X[inds_S], k=self.minPts)
        # Results are sorted, so we check only the last point
        inds_C = inds_S[dists_S_to_X[:, -1] <= self.epsilon]

        # We build the clusters
        clusters = self.connected_components(inds_C, X)

        # Return the clusters
        return clusters
```

7.2 Plots (results)

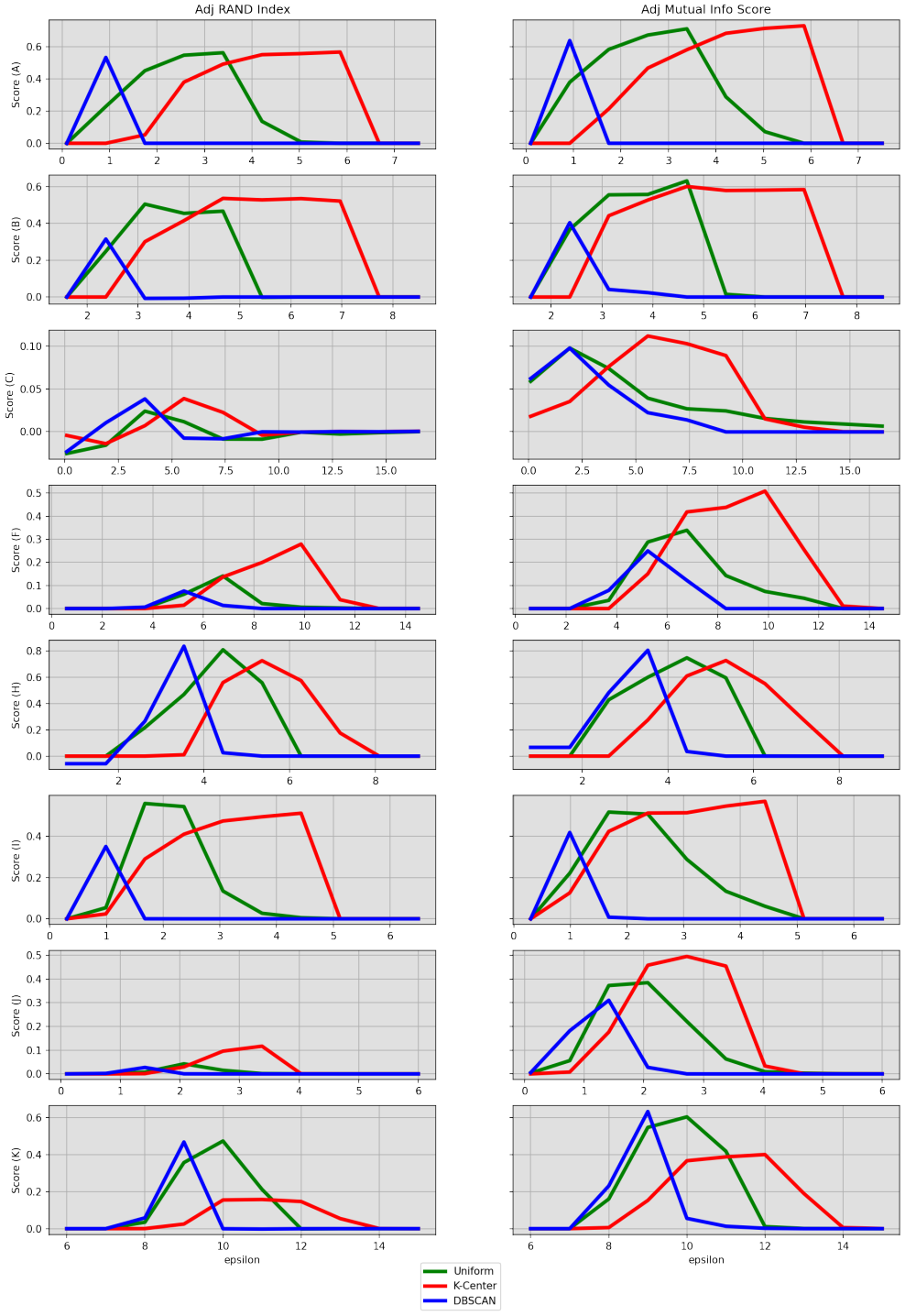


Figure 5: Adjusted Mutual Information and Adjusted Rand Index for different configurations of ϵ in all of the real datasets

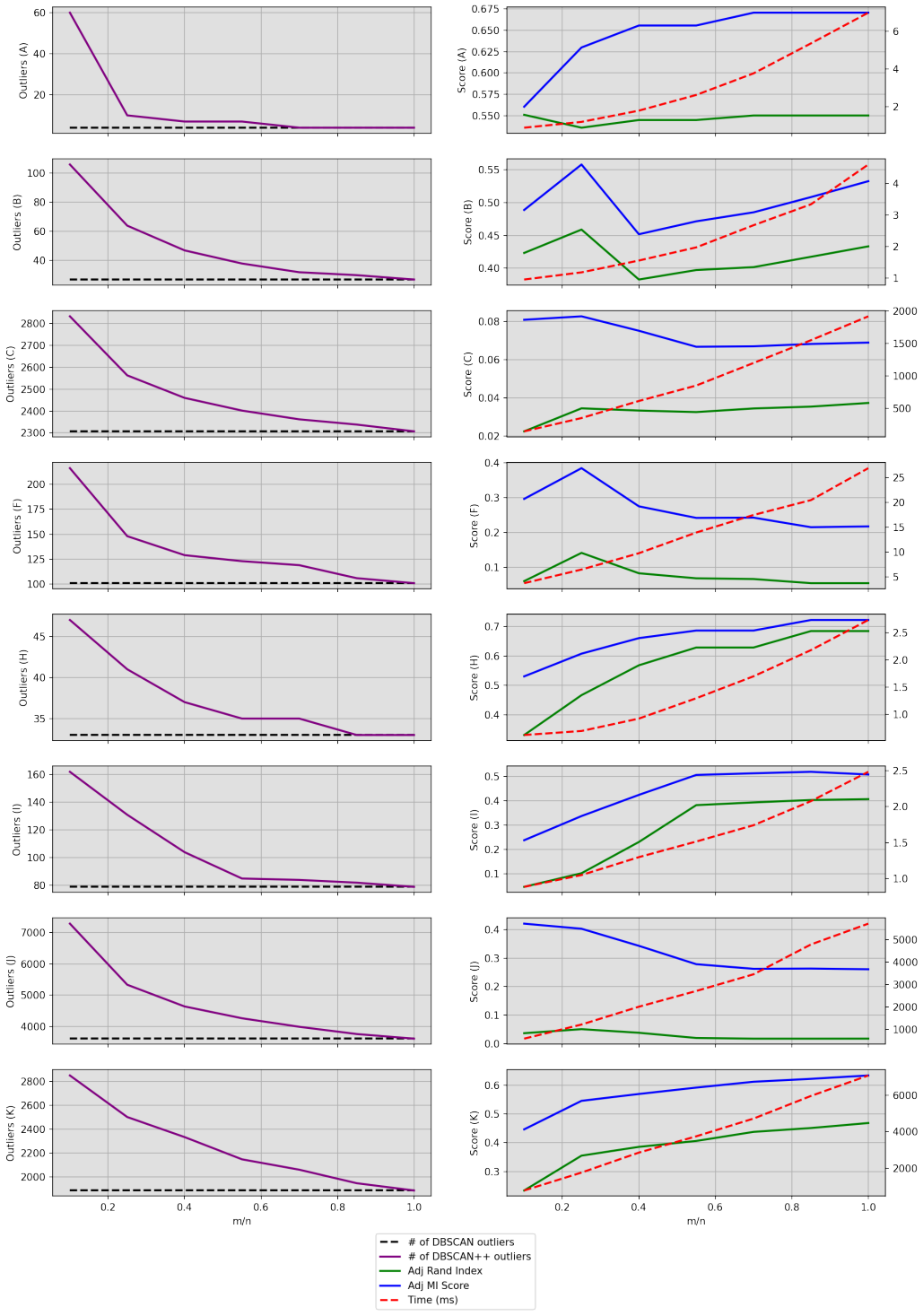
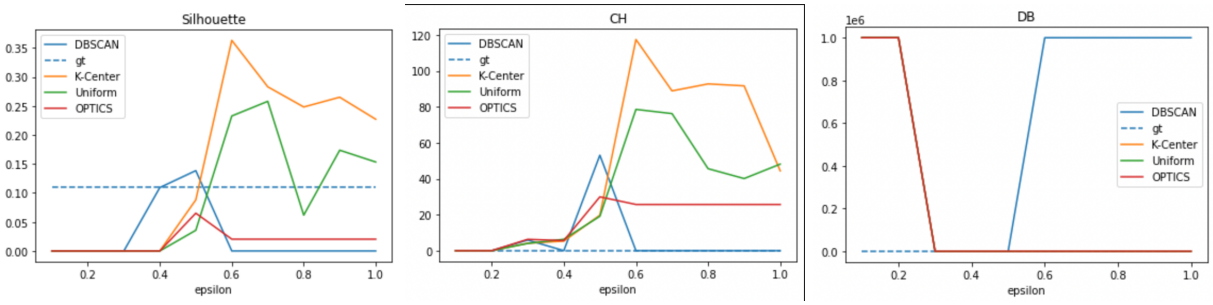
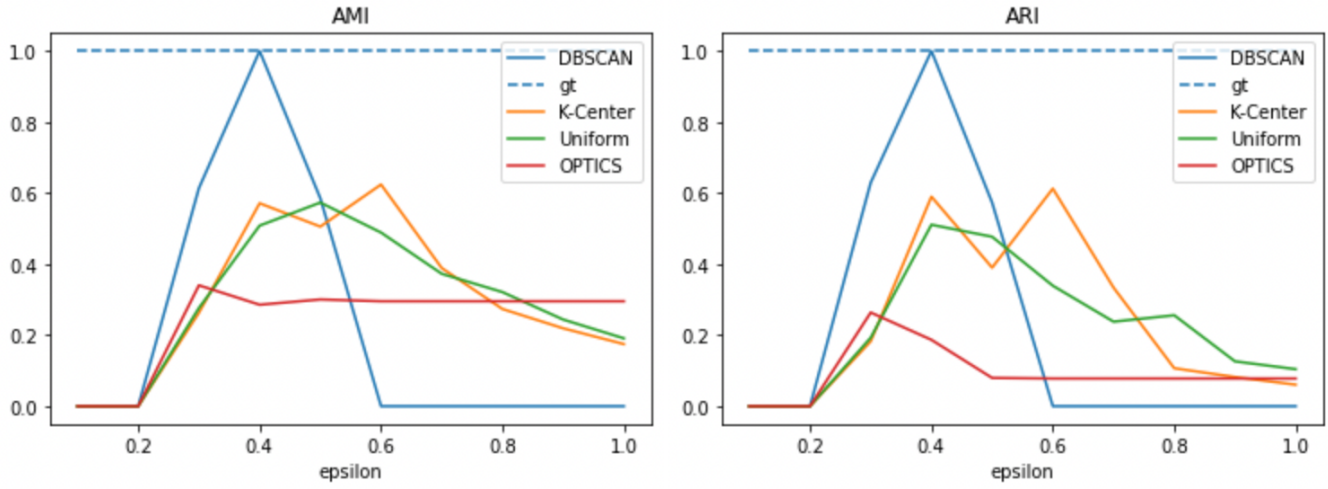


Figure 6: Number of outliers, Adjusted Mutual Information and Adjusted Rand Index for different configurations of m in all of the real datasets



(a) Silhouette (b) Calinski Harabasz (c) Davies Bouldin

Figure 7: Artificial dataset internal metrics



(a) AMI (b) ARI

Figure 8: Artificial dataset external metrics