MDCGen
multidimensional dataset generator for clustering

Iglesias, Félix; Zseby, Tanja; Ferreira, Daniel; Zimek, Arthur

Published in:
Journal of Classification

DOI:
10.1007/s00357-019-9312-3

Publication date:
2019

Document version:
Final published version

Citation for pulished version (APA):
Iglesias, F., Zseby, T., Ferreira, D., & Zimek, A. (2019). MDCGen: multidimensional dataset generator for clustering. *Journal of Classification*, *36*(3), 599-618. https://doi.org/10.1007/s00357-019-9312-3

Journal of Classification
https://doi.org/10.1007/s00357-019-9312-3

MDCGen: Multidimensional Dataset Generator for Clustering

Félix Iglesias[1] · Tanja Zseby[1] · Daniel Ferreira[1] · Arthur Zimek[2]

Published online: 23 April 2019

Abstract
We present a tool for generating multidimensional synthetic datasets for testing, evaluating, and benchmarking unsupervised classification algorithms. Our proposal fills a gap observed in previous approaches with regard to underlying distributions for the creation of multidimensional clusters. As a novelty, normal and non-normal distributions can be combined for either independently defining values feature by feature (i.e., multivariate distributions) or establishing overall intra-cluster distances. Being highly flexible, parameterizable, and randomizable, MDCGen also implements classic pursued features: (a) customization of cluster-separation, (b) overlap control, (c) addition of outliers and noise, (d) definition of correlated variables and rotations, (e) flexibility for allowing or avoiding isolation constraints per dimension, (f) creation of subspace clusters and subspace outliers, (g) importing arbitrary distributions for the value generation, and (h) dataset quality evaluations, among others. As a result, the proposed tool offers an improved range of potential datasets to perform a more comprehensive testing of clustering algorithms.

Keywords Clustering · Dataset generator · Synthetic data

1 Introduction

Synthetic datasets are necessary since real data does not allow a controlled and flexible testing of data mining algorithms and cannot be used to obtain generalization. While real-world

✉ Félix Iglesias
felix.iglesias@nt.tuwien.ac.at

Tanja Zseby
tanja.zseby@tuwien.ac.at

Daniel Ferreira
daniel.ferreira@nt.tuwien.ac.at

Arthur Zimek
zimek@imada.sdu.dk

[1] Institute of Telecommunications, TU Wien, Gusshausstrae 25, E389, 1040, Vienna, Austria

[2] Department of Mathematics and Computer Science (IMADA), Campusvej 55, 5230, Odense M, Denmark

datasets and scenarios are the ultimate reality check for competitive algorithms, it can be counterproductive to rely on real data during design and development of new algorithms (Färber et al. 2010). In this respect, synthetic datasets are to algorithms like simulations to control strategies; i.e., they are intended to develop testbeds to undergo exhaustive testing. Dataset generators must be flexible and highly parameterizable, covering a broad scope of options and shapes; therefore, algorithms can be exhaustively proofed and stress-tested in a high variety of situations.

Based on previous works and expert knowledge, we emphasize some desired characteristics and functionalities that a dataset generator for clustering should implement. A good generator is expected to satisfy the following *requirements*:

- r1. Generate datasets in a broad range of dimensions (from 2 to a high-N).
- r2. Allow to use a variety of distributions for generating cluster object values, including the possibility to import users' own distributions.
- r3. Generate both globular and non-globular clusters regardless of the number of given dimensions.
- r4. Have control on cluster overlap (avoid, allow, measure).
- r5. Independently control and allow different cluster properties in the same dataset (e.g., size, number of objects, shape, orientation).
- r6. Have control over cluster inter-distances, allowing close clusters, clusters far away of each other, or arbitrarily close and far.
- r7. Define dependencies among features in clusters, i.e., to manipulate covariances and correlations.
- r8. Incorporate outliers and noisy variables to the dataset if desired. Outliers should cover global, local, and subspace outliers.
- r9. Independently rotate clusters.
- r10. Generate clusters separated in the overall space, but not necessarily when considering subspaces; i.e., cluster structures should not be always detected when scatter plots of paired dimensions are evaluated.
- r11. Generate subspace clusters if desired, i.e., groups of objects that show a clear cluster-structure in lower dimensional subspaces but become sparse or noisy when additional dimensions are considered.
- r12. Avoid iterative algorithms (i.e., trial and error) that could slow down or even freeze the generation process in demanding parameterizations.
- r13. Allow a high flexibility in the definition and randomization of parameters in a way that dataset variability is maximized and can satisfy specific application necessities.
- r14. Reproduce datasets based on random seeds.
- r15. Generate labeled datasets for subsequent evaluations.
- r16. Output evaluations of dataset quality, e.g., overlap evaluation.

Our cluster generator MDCGen (Multidimensional Dataset Generator for Clustering) has been designed to fulfill those requirements. MDCGen is intended for research purposes; therefore it is free, open source, and publicly available to download from our website [*omitted for double-blind reviewing*]. We provide MDCGen in MATLAB and in Python.

2 Related Work

A classic algorithm for generating datasets with clusters is presented by Milligan and Cooper (1986). Their method creates between one and five clusters located in a space of up to eight dimensions and assigns points to clusters based on three models that can generate clusters of equal and unequal sizes. The generator GenRandomClust (Qiu and Joe 2006) is an improved version of the method of Milligan and Cooper. GenRandomClust adds interesting functionalities, such as the possibility to set the separation between clusters by means of a *separation index*. In addition, clusters are distant in one dimension, but there is no constraint for the isolation in the remaining dimensions; this characteristic makes that clusters cannot be easily detected by scatter plots of paired dimensions. Covariance matrices can be manipulated to offer variable shapes, diameters, and orientations. GenRandomClust also allows the inclusion of noisy variables and outliers. Steinley and Henson present OCLUS as “an analytic method for generating clusters with known overlap” (Steinley and Henson 2005). OCLUS requires the establishment (as input parameters) of overlaps in each pair of adjacent clusters for every dimension. Precisely with regard to the *overlap control* GenRandomClust and OCLUS have been compared by Korzeniewski (2013), concluding that GenRandomClust is less robust as a consequence of only controlling the overlap of the two closest clusters. In any case, cluster overlap is not necessarily something to avoid at all costs, but to control, since “many real world datasets have inherently overlapping clusters” (Banerjee et al. 2005). Actually, studying how algorithms respond to datasets with overlaps is an interesting, necessary research line.

Julia Handl worked on clustering techniques and algorithms, e.g., Handl and Knowles (2005), and has published two generators of datasets for clustering that are, together with specific documentation, publicly available (accessed: Jul, Handl 2017). One generator creates clusters based on multivariate normal distributions, allowing the addition of dependencies among features by constructing symmetric, positive-definite random covariance matrices. Clusters are generated in an iterative way, rejecting overlapping clusters and regenerating them afterwards. Since multivariate normal clusters become globular when dimensions increase, a second generator for high-dimensional scenarios is proposed (50 to 100 dimensions). The second generator creates ellipsoidal clusters defining a main axis in a random orientation and points separated a “Gaussian-distributed distance from a uniformly random point on the major axis.” Cluster origins are translated based on a genetic algorithm that minimizes a score based on the overall deviation of the data and the overlap.

On the basis of uniform and normal-based distributions, the method of Pei and Zaïane (2006) offers a generator for two-dimensional datasets where final clusters take a high variety of shapes. This approach also includes the incorporation of outliers as random noise or following defined patterns.

Finally, the cluster generator comprised in the ELKI data mining framework (Schubert et al. 2015) creates multi-dimensional datasets where distributions (uniform, normal, or gamma) are established dimension by dimension. A deep characterization is possible by means of configuration files where parameters are provided with XML tags. “Random seed” is established as an input parameter to allow reproducibility. The ELKI generator implements the manipulation and control of cluster overlaps, cluster-sizes, rotations, correlations, scaling, and translations.

3 Implementation

In this section, we present features implemented in MDCGen. We emphasize improvements and aspects that differentiate our tool from previous proposals. For a better understanding, we show and follow the pseudocode in the explanations, paying special attention to characteristic parts of MDCGen. Listing 1 gives an overall view of the MDCGen sequential procedure. Input and outputs of the MDCGen algorithm are widely discussed in Section 4.

```
1  CHECK CONSISTENCY OF input_parameters
2  INITIALIZE global_variables
3  SET distributions FOR cluster_intra_distances OR cluster_dimensions
4  GENERATE underlying_grid
5  CALCULATE base_intersections BASED ON underlying_grid
6  CALCULATE centroid_coordinates_set BASED ON base_intersections
7  MODIFY cluster_compactness_set BASED ON cluster_scaling_factors
8  GENERATE clusters IN isolated_subspaces
9  MODIFY clusters BASED ON cluster_feature_correlations
10 MODIFY clusters BASED ON cluster_rotations
11 PLACE clusters IN output_space BASED ON centroid_coordinates_set
12 CALCULATE cluster_inter_distances AND cluster_intra_distances
13 PLACE outliers IN output_space
14 ADD noise IN output_space
15 GENERATE dataset_labels
16 CALCULATE silhouette_performance
```

Listing 1 Pseudocode of MDCGen (global view)

Let us start simply considering that, as for input arguments, MDCGen works with a set of parameters and, optionally, users have the possibility to import histograms from their own research, experiments, and applications and use them as empirical distributions for the generation of cluster point values.

The first steps taken by the MDCGen algorithm are the following:

lin.1 `CHECK CONSISTENCY OF input_parameters`, where consistency of the provided parameterization is checked. Wrong parameter combinations and assignments generate errors and exit the program execution.

lin.2 `INITIALIZE global_variables`, where the initialization of all global variables and structures is conducted. Not-defined parameters or parameters-to-randomize take definite values during this phase.

3.1 Object Distributions

In a synthetic dataset, cluster objects are points located in an N-dimensional space. For the generation of every singular cluster, an independent subspace is created with a cloud of points whose placement is determined by one or some underlying distributions. The presented tool enables the creation of N-dimensional clouds of points (r1) generated by

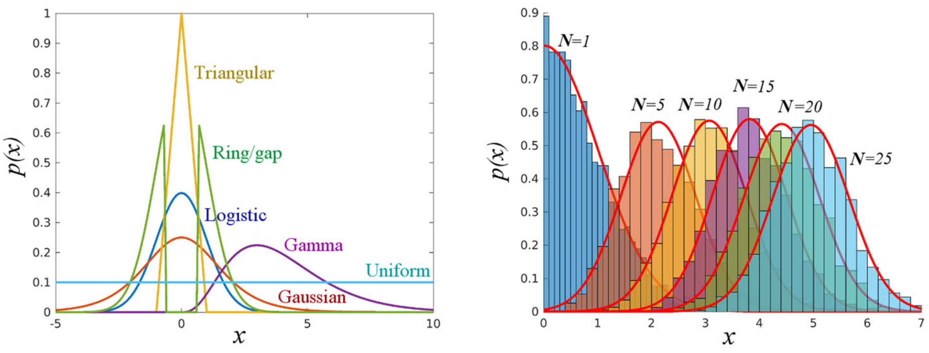

Fig. 1 Left plot: *pdf* curves of available distributions in MDCGen. Right plot: *pdf* curves—as shown by Thirey and Hickman (2015)—and histograms of euclidean distances between points in multivariate normal distributions; *N* stands for the number of dimensions

using any of the following distribution functions: (a) Uniform, (b) Normal (a.k.a. Gaussian), (c) Logistic, (d) Gamma, e) Triangular, and (f) Gap or Ring-shaped (depending on how it is finally applied)—Fig. 1, left plot, shows probability density function (*pdf*) curves of the available distributions. In subsequent steps, cluster subspaces are rotated, transformed, translated, and finally fused together in the output space. In addition to the distributions mentioned above, MDCGen allows importing arbitrary distributions by providing histograms as input arguments (r2).

Some previous dataset generators use Gaussian distributions to randomly establish values in every dimension; hence, cluster points are located following multivariate normal distributions. When no correlation between dimensions is set, whereas the overall variance is not affected by dimensionality, the Euclidean distances between points tend to be equal; therefore, they show an average value that increases in accordance with the number of dimensions. This phenomenon—thoroughly explained by Thirey and Hickman (2015)—is related to the *curse of dimensionality* and affects classifier capabilities to reach proper partitions (Beyer et al. 1999; François et al. 2007). Figure 1, right plot, reproduces a graph shown by Thirey and Hickman (2015) containing the theoretical *pdf* curves of multivariate normal distributions. We have verified the theory and superimposed the histograms of corresponding clusters generated with MDCGen.

A similar effect happens whenever point values are established variable by variable and based on distributions whose probability masses more or less coincide, provided there is no correlation or dependency between variables or they do not come from heavy-tailed distributions. Our tool allows to set distributions for every separated dimension or variable, as usual, but also to set a global distribution for the cluster intra-distances (r2, r3), i.e., objects take random values for all dimensions but what follows the selected distribution is their distance to the centroid.[1] As far as we know, this function has not been implemented

[1]It is important to remark that the capability of MDCGen to generate multivariate clusters or clusters which intra-distances follow radial-based distributions do not cover all multivariate or radial-based possible shapes. Additionally, note that available distribution functions in the current version of MDCGen show tails equal or lighter than Gaussian distributions (i.e., no heavy tails). The uniform distribution case is limited by max-min parameters, and imported histograms can resemble heavy-tailed distributions but there is no embedded curve fitting and points are generated directly with the histogram (meaning that distribution tail finishes according to the histogram binning). Therefore, if users work with a closed space, heavy-tailed distributions can be simulated.

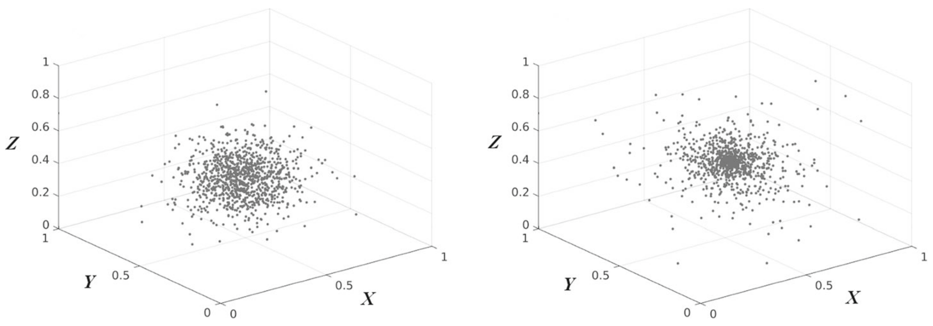

Fig. 2 Three-dimensional clusters generated with the same logistic distribution $f(x)$. Left plot: $P[a \leq X \leq b] = P[a \leq Y \leq b] = P[a \leq Z \leq b] = \int_a^b f(x)dx$. Right plot: $P[a \leq \sqrt{X^2+Y^2+Z^2} \leq b] = \int_a^b f(x)dx$

in any of the publicly available generators so far. Figure 2 shows two clusters formed by a logistic distribution with identical parameters and random seeds. The difference resides in the fact that, in plot (a), the distribution was assigned to every variable, whereas in plot (b), the distribution defined cluster intra-distances. Already with only three dimensions, it is possible to observe how Euclidean distances between points become more alike for the multivariate case (a).

In Listing 1, steps directly related to the generation of sample values are as follows:

lin.3 `SET distributions FOR cluster_intra_distances` OR cluster_dimensions, where distributions are linked either independently to every cluster dimension (if defined as multivariate) or to every cluster as a whole (radial-based case), in such a case by defining the distribution of point-to-center linear distances. A cluster being multivariate or having radial-based intra-distances is also a randomizable parameter.

lin.8 `GENERATE clusters IN isolated_subspaces`, where object values are generated for every cluster. Listing 2 explores this part of the algorithm. In multivariate cases, values are simply generated for every dimension according to the selected distribution. In radial-based cases, first an auxiliar object set is randomly generated with an uniform distribution. Every object vector is therefore divided by its magnitude to transform them into unit vectors (i.e., normalized; therefore, all vectors are separated from the origin by a distance equal to 1). Later, a set of distances is randomly generated based on the selected cluster distribution. Such distances are multiplied by the unit vectors to finally achieve the desired radial-based distribution for the cluster intra-distances (i.e., linear distances of cluster objects to the cluster center) in the N-dimensional space. The following example illustrates the difference between "radial-based" and "multivariate." Imagine a three-dimensional cluster A to be created with m samples. If "multivariate" is selected and Gaussian is the desired distribution function for all dimensions, the cluster generation process follows these steps:

1. Values are independently assigned to every dimension,

$$X = \{x_1, x_2, ..., x_m\}, \qquad X \in G$$

$$Y = \{y_1, y_2, ..., y_m\}, \qquad Y \in G$$

```
FOR EACH cluster
    IF cluster IS 'multivariate'
        FOR EACH dimension
            GENERATE cluster_dimension_values AT RANDOM BASED ON CORRESPONDING
                distribution AND cluster_compactness
        END
    ELSE % cluster IS 'radial-based'
        GENERATE cluster_values AT RANDOM BASED ON uniform_distribution
        FOR EACH cluster_vector
            NORMALIZE cluster_vector % divide it by its modulus
        END
        GENERATE distance_to_origin_values AT RANDOM BASED ON CORRESPONDING
            distribution AND cluster_compactness
        FOR EACH cluster_vector
            MULTIPLY cluster_vector BY CORRESPONDING distance_to_origin_value
        END
    END
END
RETURN clusters
```

Listing 2 Pseudocode of `GENERATE clusters IN isolated_subspaces`

$$Z = \{z_1, z_2, ..., z_m\}, \qquad Z \in G$$

where G is the set that contains all sets generated by Gaussian distributions.

2. Cluster A is formed, where the i-object of cluster A is:

$$\mathbf{a_i} = (x_i, y_i, z_i)$$

If, instead, "radial-based" is selected, being also Gaussian the desired distribution function, the cluster generation process is as follows:

1. Values are independently assigned to every dimension,

$$X = \{x_1, x_2, ..., x_m\}, \qquad X \in U$$
$$Y = \{y_1, y_2, ..., y_m\}, \qquad Y \in U$$
$$Z = \{z_1, z_2, ..., z_m\}, \qquad Z \in U$$

where U is the set that contains all sets generated by Uniform distributions.

2. The auxiliary cluster B is formed, where the i-object of B is

$$\mathbf{b_i} = (x_i, y_i, z_i)$$

3. Later, B objects are normalized; therefore, their magnitude (i.e., distance to the cluster origin) becomes 1. For the i-object of B:

$$\hat{\mathbf{b_i}} = \frac{\mathbf{b_i}}{|\mathbf{b_i}|}$$

4. A new set of values D that represent object-to-center distances is created:

$$D = \{d_1, d_2, ..., d_m\}, \qquad D \in G$$

where G is again the set that contains all sets generated by Gaussian distributions.

5. Cluster A is formed by multiplying every normalized object and its corresponding distance in D. Therefore, the i-object of cluster A is (note that d_i is a scalar):

$$\mathbf{a_i} = d_i \times \hat{\mathbf{b_i}}$$

3.2 Cluster Placement

Before cluster subspaces and their corresponding clouds of points are individually generated, it needs to be determined how and where to place such subspaces in the output space. This part of the dataset generation is tricky and has a considerable impact on classifier performances. It must be possible to create clusters with variable cluster inter-distances for the same dataset (some of them close to each other, some of them far from another). To address this issue, our tool initially limits each dimension to a closed [0, 1] value domain and later draws an imaginary grid to hang cluster subspaces in grid intersections ([0,1] boundaries might be crossed in certain special cases, e.g., when a cluster with high size or sparsity in at least one dimension is placed close to output space borders; in any case, the origin of cluster subspaces are always located within [0,1] ranges—example in Fig. 5). Every dimension is divided by α_i equidistant hyperplanes, where i marks the specific dimension. By default, we define that the grid granularity depends on the given number of clusters k.

Equation 1 provides the default definition of α_i:

$$\alpha_i = 2 + C_i \left\lfloor 1 + \frac{k}{\ln k} \right\rfloor \tag{1}$$

where C_i (`alpha_constant`) is a configurable parameter (set to 1 by default). If desired, α_i can be independently adjusted for each dimension (the tool ensures that the selection of the diverse α creates a grid whose total number of intersections is larger than k). For instance, in a two-dimensional space (x, y), given $k = 7$, by default $\alpha_x = \alpha_y = 2 + \lfloor 1 + \frac{7}{\ln 7} \rfloor = 6$. The addend "+2" corresponds to hyperplanes that take 0 and 1 values in the i-dimension—cluster subspaces are not allowed to be centered there. In our example, it means that from the $6 \cdot 6 = 36$ hyperplane intersections, we have 20 non-usable intersections at the grid borders and therefore 16 valid intersections available to locate the 7 cluster subspaces.

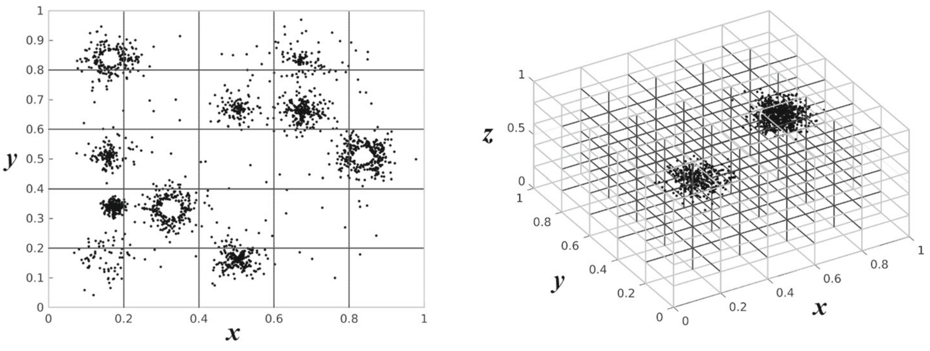

Fig. 3 Example of grids for two- and three-dimensional spaces, with datasets of 10 and 2 clusters respectively. Clusters are initially hooked in grid intersections but later displaced by final random translations that depend on the grid hyperrectangle size

The explained procedure is illustrated in the examples of Fig. 3. The related step in Listing 1 is:

lin.4 `GENERATE underlying_grid`, which establishes the number of hyperplanes per dimension and the valid hyperplane intersections. Details are provided in Listing 3. The WHILE loop ensures that the grid contains enough intersections for all clusters.

Space intersections are numbered and jumbled according to uniform random permutations. Later on, the first k intersections are selected and their indices decomposed according to α values. For example, in a three-dimensional space, intersection I_j transforms into (x_j, y_j, z_j), where $I_j = x_j\alpha_x + y_j\alpha_y + z_j\alpha_z$. Such indexing is only performed for a low number of dimensions to guarantee no subspace overlap, the remaining dimension coordinates are randomly generated (otherwise the one-dimensional indexing would become soon unfeasible for high-dimensional grids). To smooth the cluster alignment due to the grid arrangement, clusters are finally translated according to a random distance that depends on grid cell size (i.e., hyperplane separations). Steps in Listing 1 that cover this part are as follows:

lin.5 `CALCULATE base_intersections BASED ON underlying_grid`, which outputs an array with indexes that correspond to *base* intersections. *Base* indicates that intersections belong to a dimensional-reduced subspace with enough intersections to allocate all desired clusters. Unless users desire output spaces with

```
FOR EACH dimension
    IF alpha_mode IS ‘fixed’
        SET dimension_hyperplanes TO CORRESPONDING alpha_constant + 2
        % ‘2’ refers to grid borders
    ELSE % alpha_mode IS ‘k-based’
        SET dimension_hyperplanes PROPORTIONAL TO CORRESPONDING alpha_constant
            AND number_of_clusters
        % as shown in equation 1
    END
END
CALCULATE number_of_valid_intersections FROM total_hyperplanes
SET dimension_index TO 1
WHILE number_of_valid_intersections IS LESS OR EQUAL TO number_of_clusters
    SELECT dimension BASED ON dimension_index
    INCREASE dimension_hyperplanes BY 1
    CALCULATE number_of_valid_intersections FROM total_hyperplanes
    INCREASE dimension_index BY 1 % circular shift
END
RETURN underlying_grid
% underlying_grid contains all variables related to hyperplanes
```

Listing 3 Pseudocode of `GENERATE underlying_grid`

```
SET required_intersections PROPORTIONAL TO number_of_clusters AND number_of_outliers
% as shown in equation 2
IF required_intersections is GREATER THAN number_of_valid_intersections
  SET required_intersections TO number_of_valid_intersections
END
SET base_dimensions TO 1
DO
  INCREASE base_dimensions BY 1
  CALCULATE number_of_base_intersections FROM base_hyperplanes
WHILE number_of_base_intersections ARE LESS OR EQUAL TO required_intersections
PERMUTE base_intersections
RETURN base_intersections
```

Listing 4 Pseudocode of `CALCULATE base_intersections BASED ON underlying_grid`

very few intersections and design them accordingly, a reference for the minimum value of base intersections is fixed by the ad hoc, experimental (2):

$$\beta = 2k + \frac{\text{outliers}}{k} \tag{2}$$

where β stands for base intersections, k is the number of clusters and $outliers$ is the number of outliers. Listing 4 explores this step.

lin.6 `CALCULATE centroid_coordinates_set BASED ON base_intersections`, where every cluster centroid[2] is assigned a unique location in the final solution space based on the intersection index. Listing 5 delves into this step.

lin.11 `PLACE clusters IN output_space BASED ON centroid_coordinates_set` simply takes vectors of every cluster, adds the corresponding centroid vector, and joins all clusters in a single matrix (i.e., the dataset or `output_space`). Before this step, clusters hang in isolated subspaces with the preliminary centroid located in the coordinates origin.

If we retrieve the example in Section 3.1 in which a three-dimensional cluster A was generated, in this step, cluster A—after applying additional transformations and operations configured by the user—is joined to other clusters in the same space and *hanged* in its corresponding location by adding the cluster centroid coordinates to every object vector. If A' is the expression of the cluster in the final space and $\mathbf{c_A}$ stands for its corresponding centroid location, the i-object of A' becomes

$$\mathbf{a_{i'}} = \mathbf{a_i} + \mathbf{c_A}$$

The presented way of fusing cluster subspaces solves some issues related to cluster placement and makes it easy to implement some desired functionalities:

[2]Note that *centroids* are used here in a broad sense to designate reference points whose goal is to place clusters in the final output space. Therefore, such *centroids* are not necessarily required to exactly correspond to real cluster centroids, especially if the underlying distributions are skewed.

```
FOR EACH cluster
  ASSIGN FIRST base_intersection TO cluster
  TRANSFORM base_intersection INTO base_coordinates
  SET base_centroid_coordinates TO base_coordinates
  SET remaining_centroid_coordinates AT RANDOM BASED ON uniform_distribution
  SET deviation AT RANDOM BASED ON uniform_distribution AND hyperplane_separations
  ADD deviation TO centroid_coordinates
  REMOVE base_intersection FROM base_intersections
END
RETURN centroid_coordinates_set
```

Listing 5 Pseudocode of `CALCULATE centroid_coordinates_set BASED ON base_intersections`

- (r4) Cluster overlap is easily controlled by scaling distribution parameters in accordance with the size of the hyperrectangles (or N-dimensional cells) described by the grid. Examples are shown in Figs. 4 and 5.
- (r12) There is no need to implement iterative algorithms for the cluster placement as cluster subspaces are paired with unique grid intersections through a one-dimensional index. Also, when placing outliers, there is no need to check if outliers are falling inside cluster influence areas because outliers are directly scattered around the not-used grid intersections.
- (r6) Grid hyperplanes or divisions are configurable. The design of the grid will partially define if clusters are close to one another, far away from each other or a combination of both, therefore generating variable cluster inter-distances (example in Fig. 4).
- (r10) Grid hyperplanes are configurable per dimension. Provided the configuration suffices for the required number of intersections (above k), clusters can be distant in the

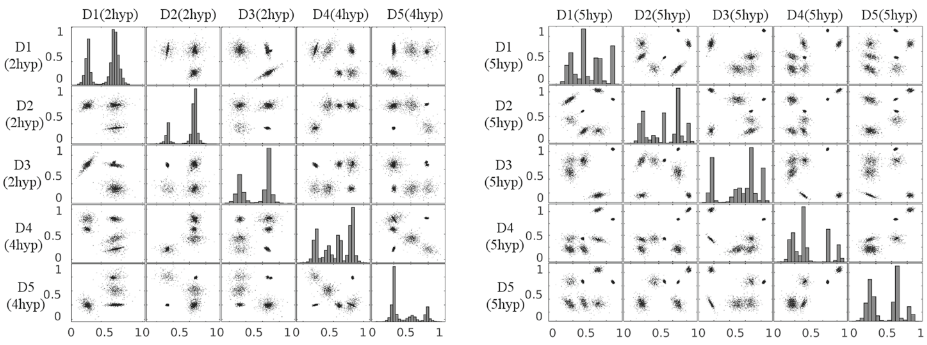

Fig. 4 Two five-dimensional datasets of 7 clusters generated with exactly the same parameters but with different grids. Left plot: the grid allocates clusters in {2,2,2,4,4} hyperplanes. Right plot: the grid allocates clusters in {5,5,5,5,5} hyperplanes. The distribution σ is scaled based on grid granularity. Note that, even though clusters are well separated in the overall five-dimensional space in both cases—Silhouette indices: $S = 0.71$ and $S = 0.80$ respectively—overlap in two-dimensional subspaces severely occurs for all cases in the left plot (less than 7 clusters are distinguishable) and only occasionally in the right plot (7 clusters are normally distinguishable)

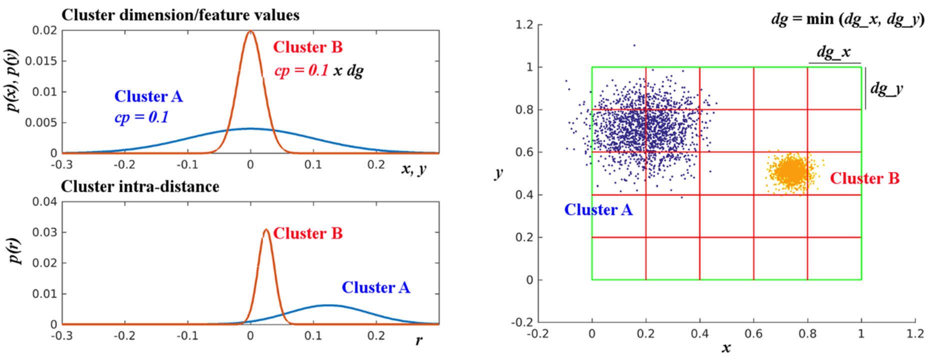

Fig. 5 Two-dimensional example with two multivariate Gaussian clusters. Cluster A is not scaled and Cluster B is scaled based on grid size. The right plot shows the solution space, whereas plots on the left display the *pdf* curves that generated cluster dimension values (upper) and the *pdf* curves of the final intra-distances (lower)

overall space but overlapping when subspaces are independently evaluated (example in Fig. 4).

3.3 Overlap Control

The overlap control is undertaken by the design of input parameters, mainly the type of distributions, compactness coefficients, and grid granularity together with the *scale* option. The *type of distribution* has an obvious effect on the potential overlap as distributions show different sparsity by definition (see Fig. 1, left plot). The MDCGen uses distributions to define either feature values independently or directly object intra-distances in the N-dimensional space, implying a direct impact in the space required by every cluster. *Compact coefficients* (*cp*) directly define variance parameters in the available distributions (e.g., σ in Gaussian or Logistic cases, *lower*, and *upper* thresholds in triangular ones), whereas mean parameters are set to “0” previous to any translation. In imported distributions, *cp* acts as an additional scaling factor not linked to the grid scaling. On the other hand, the *scale* parameter controls the scaling of cluster values by a factor as well as based on grid size (i.e., hyperplane separations). The example in Fig. 5 helps to understand how the overlap control works. In the two-dimensional example, two multivariate Gaussian clusters have been created with the same type of distribution and compactness coefficient (i.e., $cp = \sigma = 0.1$) but, whereas Cluster A is not scaled, Cluster B is scaled in line with grid size. As for the steps in Listing 1, it involves

lin.7 `MODIFY cluster_compactness_set BASED ON cluster_scaling_factors`, which defines a coefficient for every cluster before the generation of object values. Listing 6 offers further explanations for this step.

Given that the space for cluster placement is always enclosed within the [0,1]-hypercube, it is not difficult to control the overlap during the dataset parameterization. In any case, MDCGen evaluates overlap by means of Silhouettes (r16), which gives a score between “0” and “1” to assess intra-cluster compactness and inter-cluster separation (Rousseeuw 1987). Related steps in Listing 1 are as follows:

```
FOR EACH dimension
  CALCULATE dimension_hyperplanes
END
IF scaling_mode IS 'min_grid_separation'
  SET separation_quotient TO MINIMUM number_of_hyperplanes_in_a_dimension
ELSE % scaling_mode IS 'max_grid_separation'
  SET separation_quotient TO MAXIMUM number_of_hyperplanes_in_a_dimension
END
FOR EACH cluster
  MULTIPLY cluster_compactness BY CORRESPONDING cluster_scaling_factor
  DIVIDE cluster_compactness BY separation_quotient
END
RETURN cluster_compactness_set
```

Listing 6 `MODIFY cluster_compactness_set BASED ON cluster_scaling_factors`

lin.12 `CALCULATE cluster_inter_distances AND cluster_intra_distances`, where cluster inter- and intra-distances as well as dataset geometrical properties are calculated. Such calculations and estimations allow using other cluster compactness vs. distance coefficients and measures in addition to Silhouette.

lin.16 `CALCULATE silhouette_performance`, which calls Silhouette algorithms.

3.4 Subspaces, Outliers, and Noise

To avoid resorting to trial and error processes for ensuring that outliers do not fall within clustered areas, outliers are directly spread around grid intersections where there are no defined clusters. In addition, it is possible to add an arbitrary number of irrelevant noisy features (r8). Noise is generated by uniform distributions within the [0,1] value range and can be defined for specific clusters and dimensions, allowing the creation of subspace clusters (r11). The arrangement of clusters and outliers over the underlying grid structure enables the natural generation of subspace outliers. Figure 6 shows an example of a three-dimensional dataset with normal clusters, subspace clusters, global outliers, and subspace outliers. Detailed in Listing 7, steps of Listing 1 that cope with outliers and noise are as follows:

lin.13 `PLACE outliers IN output_space`, which uses the remaining free base intersections to locate outliers as we did before with cluster centroids. However, in this case, base intersections are reused following a circular shift. Note that this assignment is only performed for a reduced number of dimensions, the rest are again established at random. Similarly to the case of cluster centroids, a final deviation based on the hyperplane separation is applied to avoid alignment with grid intersections.

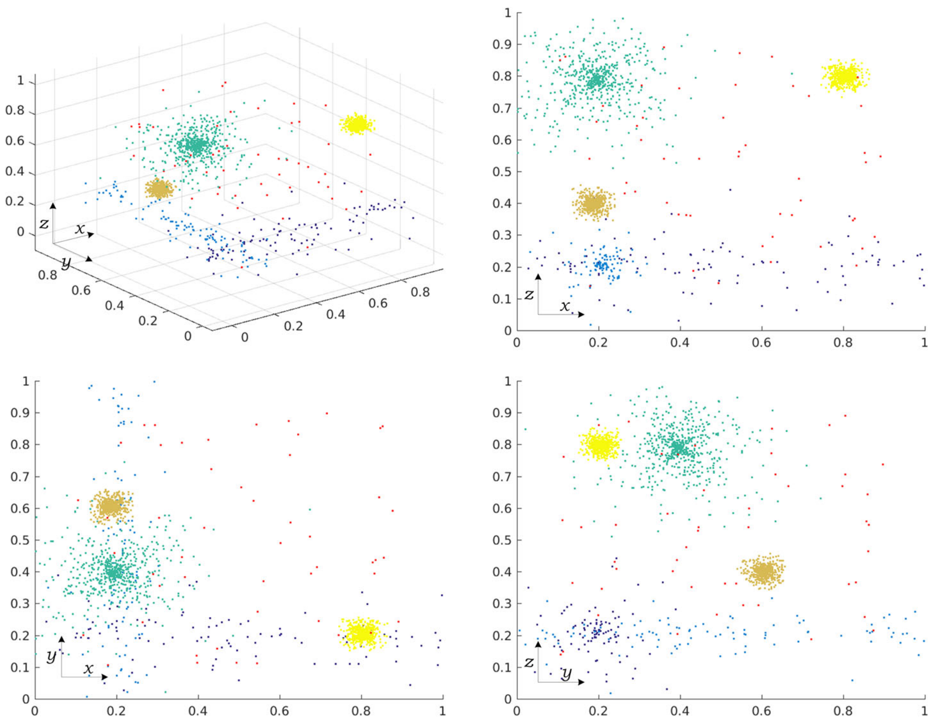

Fig. 6 Example of three-dimensional dataset with subspace clusters and subspace outliers. Outliers are shown in red color whereas other colors correspond to five different clusters. Upper left plot: (x, y, z); upper right plot: (x, z); lower left plot: (x, y); lower right plot: (y, z)

lin.14 `ADD noise IN output_space`, which simply adds noise to global dimensions or cluster dimensions by replacing the generated values by uniform noise.

3.5 Additional Features: Correlations, Rotation, and Labeling

Additional implemented features are:

- *Feature correlations* (r7): MDCGen allows the definition of correlated features by introducing coefficients (either per dataset or per cluster) that state the maximum allowed correlation (positive or negative) between two features. To do that, a correlation matrix C is created for each cluster and correlation coefficients are randomly generated but without exceeding the given threshold. To transform C into a valid covariance matrix, we use the method of Higham (1988), which is able to calculate the nearest symmetric positive semidefinite matrix S. Later, Cholesky decomposition is applied on S to find a matrix L, which accomplishes $S = L \cdot L^*$ (L^* is the conjugate transpose of L). Thus, it is possible to compute $Y = LX$, being X a set of vectors where object values of every cluster dimension are represented as random variables. Y contains the vectors of the final correlated variable values.

```
% --- PLACE outliers IN output_space
FOR EACH outlier
  ASSIGN FIRST base_intersection TO outlier
  TRANSFORM base_intersection INTO base_coordinates
  SET base_outlier_coordinates TO base_coordinates
  SET remaining_outlier_coordinates AT RANDOM BASED ON uniform_distribution
  SET deviation AT RANDOM BASED ON uniform_distribution AND hyperplane_separations
  ADD deviation TO outlier_coordinates
  SHIFT base_intersection TO LAST POSITION IN base_intersections
END
ADD outlier TO output_space
RETURN output_space

% --- ADD noise IN output_space
FOR EACH cluster % noisy cluster dimensions
  FOR EACH dimension
    IF dimension OF cluster IS 'noisy'
      GENERATE new_values AT RANDOM BASED ON uniform_distribution
      REPLACE cluster_dimension_values BY new_values
    END
  END
END
UPDATE output_space BASED ON clusters

FOR EACH dimension % noisy dataset dimensions
  IF dimension IS 'noisy'
    GENERATE new_values AT RANDOM BASED ON uniform_distribution
    REPLACE output_space_dimension_values BY new_values
  END
END
RETURN clusters, output_space
```

Listing 7 Adding outliers and noise

- *Cluster rotation* (r9): Given the difficulties to specifically define a *rotation* in spaces with more than three dimensions (Daniele 2001), the MDCGen tool is limited to implement cluster isometries by generating a random orthonormal matrix Q, which, by means of $Y = QX$, performs a unitary transformation on X.

- *Labeled dataset* (r15): In addition to the N-dimensional dataset, MDCGen generates an array with numerical labels that links objects to the created clusters. Outliers are labeled with the “0” value.

These additional features correspond to the following steps in Listing 1:

lin.9 `MODIFY clusters BASED ON cluster_feature_correlations`, which starts constructing a correlation matrix based on the parameterization and later applies the method in Higham (1988) and Cholesky decomposition, see Listing 8. `nearestSPD_matrix` refers to Higham’s method (Higham 1988).

lin.10 `MODIFY clusters BASED ON cluster_rotation`, which operates by creating a random orthogonal square matrix. Also detailed in Listing 8.

lin.15 `GENERATE dataset_labels`, where dataset labels are simply generated with positive numbers for clustered objects and 0 for outliers.

3.6 Cluster Generation Summary

Finally, we repeat here the key steps of the MDCGen data generation process in an intuitive and summarized way (Listing 1):

```
% --- MODIFY clusters BASED ON cluster_feature_correlations
FOR EACH cluster
  IF cluster IS ‘with_correlations’
    GENERATE correlation_matrix BASED ON cluster_correlation_parameters
    GENERATE nearestSPD_matrix BASED ON correlation_matrix
    GENERATE cholesky_covariance_matrix BASED ON nearestSPD_matrix
    MULTIPLY cluster_values BY cholesky_covariance_matrix
  END
END
RETURN clusters

% --- MODIFY clusters BASED ON cluster_rotation
FOR EACH cluster
  IF cluster IS ‘with_rotation’
    GENERATE square_rotation_matrix AT RANDOM BASED ON uniform_distribution
    ORTHOGONALIZE square_rotation_matrix
    MULTIPLY cluster_values BY square_rotation_matrix
  END
END
RETURN clusters
```

Listing 8 Adding feature correlations and cluster rotations

1. An N-dimensional grid is generated in the N-dimensional space. Grid granularity is adjusted based on the desired number of clusters, the desired number of dimensions, and configuration parameters related to cluster overlap.
2. Points in the space to locate cluster centroids are linked to unique grid intersections (plus some optional drift).
3. Cluster compactness factors are modified based on the size of grid cells and configuration parameters. Cluster compactness factors define how big clusters are in the final space.
4. Clusters are independently generated in isolated spaces according to the selected distributions, the modified compactness factors, and other configuration parameters.
5. Clusters are independently modified based on additional configuration parameters and options: rotation, correlations, etc.
6. Clusters are joined and placed together in the final space according to the locations reserved for their corresponding centroids (Point 2 of this list).
7. Outliers are generated according to configuration parameters and spread around free grid intersections.
8. Noise is generated according to configuration parameters and added into the final space.

4 Parameters and Configuration

This section shows the configurable parameters of MDCGen and the possibilities for either randomizing or specifying such parameters, therefore controlling the structure and generation of the final dataset (r13). If a parameter is not defined among the inputs, the tool randomizes its value or applies a value by default. One of the main challenges in the MDCGen design was allowing such randomization and, at the same time, a deep parameter specification. Configurations and decisions are possible at different levels: dataset, cluster, dimension, and cluster-dimension. Moreover, the MDCGen tool is devised to be integrated in testbeds, frameworks or chain processes to provide a stream of different datasets within a given set of desired characteristics. Hence, to enable covering a broad range of dataset possibilities, the parameters are multiple and some training for tuning the tool is required. The performance evaluation generated by MDCGen as well as scatter plots and histograms are suitable ways to control and check the new dataset.

In this section, we provide examples based on the MATLAB version of MDCGen. In the Python and HTML versions, parameters are equivalently defined by means of JSON input files. The generator can be called even with no parameters at all:

`>> [results]=mdcgen()` Nevertheless, it is expected that users carry out a minimum configuration. Parameters accept being defined by variables with one or more of the following types: scalars [sc], arrays [ar], and matrices [mx]. Input parameters are as follows:

- sd: random seed [sc], to allow dataset reproducibility (r14).
- M: total number of clustered objects (points) in the dataset [sc].
- N: number of dimensions [sc].
- k: number of clusters and cluster masses [sc, ar].

  If k is a scalar, k establishes the number of clusters and, therefore, M objects are randomly distributed among k clusters. By default, the minimum number of points allowed per cluster is stated as a function of k and M, or it equals the input parameter km, if defined. If k is entered as an array, the number of clusters is the array length, whereas array values are taken as the number of objects embraced by each individual cluster.

- *km*: minimum absolute number of objects per cluster [sc].
- *d*: cluster distribution [sc, ar, mx].

  If *d* is a scalar, the value defines all dataset distributions. If *d* is an array, *d* has length *k* and its values define distributions per cluster. If *d* is a matrix, it is a $k \times N$ matrix whose values define distributions per cluster and dimension.

  Allowed *d* values and their meanings are as follows: (0) Random distribution; (1) Uniform; (2) Gaussian; (3) Logistic; (4) Triangular; (5) Gamma; (6) Gap or Ring-shaped. In addition, alternative distributions can be imported as described below in this section (for configuration purposes, they take values for *d* and *dflag* indices starting from 7).
- *dflag*: enable distribution [sc, ar].

  As an array, *dflag* states which of the implemented distributions are available when $d = 0$, i.e., distributions are selected randomly. As a scalar, "1" enables all distributions and "0" disables all distributions except for Gaussian.
- mv: multivariate distributions [sc, ar].

  mv value ($\in \{1, -1, 0\}$) defines if distributions are applied to clusters dimension by dimension, if they are applied to cluster intra-distances, or if such decision is established at random (see Section 3.1).
- *cp*: compactness coefficient [sc, ar].

  cp determines the variance component of the applied distribution. For instance, σ in Gaussian, $upper$ and $lower$ thresholds in triangular and uniform cases, or the *b* parameter for the Gamma distribution. Given that feature domains in the whole dataset are enclosed within [0,1], a consequent, meaningful design of *cp* is endorsed. Again, *cp* can be defined affecting all clusters (scalar) or cluster by cluster (array).
- *scale*: scale to grid [sc, ar].

  By means of *scale*, cluster *cp* can be automatically scaled according to grid size, therefore controlling cluster overlap. If positive (`scaling_mode IS 'min_grid_separation'`), *scale* uses the minimum grid intersection distance for scaling; if negative (`scaling_mode IS 'max_grid_separation'`), it uses the maximum. *scale* can also be defined either for all clusters (scalar) or independently cluster by cluster (array).
- α: grid factor [sc, ar].

  α determines grid granularity as explained in Section 3.2. A positive value (`alpha_mode IS 'k_based'`) multiplies (1), whereas a negative value (`alpha_mode IS 'fixed'`) directly replaces (1) with the given input (after removing the negative sign). α can be defined either for all dimensions together (scalar) or independently dimension by dimension (array).
- *corr*: feature correlation [sc, ar].

  Feature correlations (see Section 3.5) are set by defining a maximum correlation coefficient, which is applied for all clusters and dimension likewise (scalar), or for cluster dimensions independently taken (array).
- *rot*: cluster rotation [sc, ar].

  As explained in Section 3.5. Also definable per cluster (array) or for all clusters (scalar).
- out: total number of outliers [sc].
- *Nnoise*: noisy dimensions [sc, ar, mx].

  A scalar value states the number of noisy dimensions to be added to the whole dataset. If *Nnoise* is defined as an array, values mark which dimensions of the dataset must be replaced by noise. If it is defined as a matrix, every column corresponds to a cluster and values state which dimensions (specific for every cluster) must be replaced by noise. Noisy dimensions are created after any other transformation (correlation, rotation, etc.). Sophisticated and flexible noise generation allows also to generate benchmarks for subspace clustering algorithms (Kriegel et al. 2009).

Output parameters are as follows:

- *data*: the final dataset, a matrix of M' rows and N' dimensions[3].
- *label*: array with M' labels.
- *perf*: data structure containing performance indices for the overall datasets as well as for every independent cluster.

Additionally, users can import any number of distributions from their experiments and applications[4]. Distributions are imported as histograms, which are interpreted by MDCGen as probability density estimations to underly randomization processes. The optional argument *dist* is to be provided when calling the tool. *dist* is a data structure that contains the following:

- *n*: the number of imported distributions.
- $d(1:n)$*.values*: arrays with histogram bin values of imported distributions. n arrays are required. The number of array elements is independent for each array.
- $d(1:n)$*.edges*: arrays with histogram bin boundaries (or edges). n arrays are required. Array lengths must be equal to the corresponding *values* array plus one (i.e., each *value* must fall between two *edges*).

5 Conclusions

This paper presents MDCGen, a tool for generating datasets of objects arranged in clusters. MDCGen is devised for research purposes, specifically to test clustering algorithms and clustering validation techniques. It has been designed to fulfill the principal features implemented in previous approaches as well as the requirements observed by expert data analysts. In addition to allow a high flexibility in randomization and parameterization, the main novelties of MDCGen are related to the overlap control and cluster placement, both driven by the creation of hyper-grids where cluster subspaces hang; and to the option of not only generating multivariate clusters, but also the possibility to directly define object distances to cluster centroids with a single distribution.

MDCGen opens a broad spectrum of possibilities to easily test data mining and machine learning algorithms, bringing them to demanding but controlled conditions. MDCGen is open source, free, and publicly available.

Acknowledgments This research has been partially funded by the Vienna Science and Technology Fund (WWTF) through project ICT15-129, “BigDAMA”.

Funding Information Open access funding provided by TU Wien (TUW).

[3]The apostrophe ’ marks that M and N can be finally increased by the addition of outliers and noise.

[4]Alternatively, if desired, the MDCGen software comes with a generator of random empirical distributions where parameters like multimodes or skewness can be configured.

References

Banerjee, A., Krumpelman, C., Ghosh, J., Basu, S., Mooney, R.J. (2005). Model-based Overlapping Clustering. In *Proceedings of the 11th ACM SIGKDD international conference on knowledge discovery in data mining* (pp. 532–537).

Beyer, K., Goldstein, J., Ramakrishnan, R., Shaft, U. (1999). When is "Nearest Neighbor" meaningful? In *Proceedings of the international conference on database theory (ICDT)* pp. 217–235.

Daniele, M. (2001). On the rigid rotation concept in n-dimensional spaces. *Journal of the Astronautical Sciences*, *49*(3), 401–420.

Färber, I., Günnemann, S., Kriegel, H.-P., Kröger, P., Müller, E., Schubert, E., Seidl, T., Zimek, A. (2010). On using class-labels in evaluation of clusterings. In *Proceedings of the 1st international workshop on discovering, summarizing and using multiple clusterings (MultiClust 2010) in conjunction with 16th ACM SIGKDD conference on knowledge discovery and data mining, KDD*: Washington.

François, D., Wertz, V., Verleysen, M. (2007). The concentration of fractional distances. *IEEE Transactions on Knowledge and Data Engineering*, *19*(7), 873–886.

Handl, J. (2017). Accessed: cluster generators. http://personalpages.manchester.ac.uk/mbs/julia.handl/generators.html.

Handl, J., & Knowles, J. (2005). Multiobjective Clustering around medoids. In *2005 IEEE Congress on evolutionary computation* (Vol. 1, pp. 632–639).

Higham, N.J. (1988). Computing a nearest symmetric positive semidefinite matrix. *Linear Algebra and its Applications*, *103*, 103–118.

Korzeniewski, J. (2013). Empirical evaluation of OCLUS and GenRandomClust algorithms of generating cluster structures. *Statistics in Transition New Series*, *14*(3), 487–494.

Kriegel, H.-P., Kröger, P., Zimek, A. (2009). Clustering high dimensional data: a survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM TKDD*, *3*(1), 1–58.

Milligan, G.W., & Cooper, M.C. (1986). A study of the comparability of external criteria for hierarchical cluster analysis. *Multivariate Behavioral Research*, *21*(4), 441–458.

Pei, Y., & Zaïane, O. (2006). A synthetic data generator for clustering and outlier analysis. Technical report, Department of Computing Science, University of Alberta Edmonton, AB, Canada.

Qiu, W., & Joe, H. (2006). Generation of random clusters with specified degree of separation. *Journal of Classification*, *23*(2), 315–334.

Rousseeuw, P.J. (1987). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, *20*, 53–65.

Schubert, E., Koos, A., Emrich, T., Züfle, A., Schmid, K.A., Zimek, A. (2015). A framework for clustering uncertain data. *PVLDB*, *8*(12), 1976–1979.

Steinley, D., & Henson, R. (2005). OCLUS: an analytic method for generating clusters with known overlap, (Vol. 22.

Thirey, B., & Hickman, R. (2015). Distribution of Euclidean Distances Between Randomly Distributed Gaussian Points. In *n-Space, SAO/NASA ADS arXiv e-prints Abstract Service* (pp. 1–13). arXiv:1508.02238.