GENERIC MULTIDIMENSIONAL DATA GENERATOR DESIGN AND IMPLEMENTATION (RESEARCH REPORT RB-16/14)

1. INTRODUCTION

This report describes the design, implementation, and use of the new version of the artificial data generator designed to create multidimensional and multiclass data sets. When creating a new generator, we used the experience related to its first version realized within the framework of a master's thesis [1]. The first version of the generator was limited to two-dimensional and two-class datasets and did not include all considered types of data points. However, it allowed to conduct a series of experiments described, among others, in [3].

The following requirements were defined for the new generator:

- Generation of multidimensional and multiclass datasets (from a practical point of view there are 3-4 classes and 3-10 conditional attributes corresponding to individual dimensions);
- Definition of classes consisting of one or more regions (meta-balls and meta-cubes) with the ability to create integumental regions that fill the space between regions (usually related to the majority class);
- Possibility of varying the relative distribution of points between regions and controlling the density of points within each region (there are two possible distributions – uniform and normal);
- Generation of data examples of all types, as considered in [2]: safe, borderline, rare, and outlier examples:
    - **Safe**: Examples lying within individual regions and having a homogeneous neighborhood consisting mainly of points of the same class;
    - **Borderline**: Coastal examples lying on the periphery/border of individual regions and mixed with points from other classes;
    - **Rare**: Examples forming small disjuncts (consisting of 2-3 examples) lying far from the regions that make up the class: they represent rare but correct observations;
    - **Outlier**: Single examples that are far from other examples of the same class, corresponding to “standing-out/thrown” examples.
- Generation of training/test sets maintaining the location of rare and outlier points (i.e., in a given train/test pair, they appear in similar locations);
- Java implementation that allows to write data in ARFF format used in WEKA environment.

2. HOW TO START THE GENERATOR

The generator does not have a specialized user interface - it needs to be called from the command line and looks like this:

```
java –jar DataGenSW –config <configuration file>
```

In the above example, it is assumed that the **DataGenSW.jar** file and additional libraries (located in the **./lib** subdirectory) are in the current directory. If they are stored in another location, then the full path to the file must be specified, for example:

```
java –jar \Projects\Generator\DataGenSW –config flower3-2d.conf
```

In case of correct operation, the generator will display a short report with a list of generated files (for example, as depicted below, the package names have been removed from the log entries to give a concise presentation):

```
> java –jar ..\dist\DataGenSW.jar –config paw3-2d.conf
[ main ] INFO Generator – Generating data set(s)
[ main ] INFO Generator – Pass 1...
[ main ] INFO Generator – Learning set
[ main ] INFO ARFFWriter – Saving file paw3-2d.arff ...
```

If there is an error in the configuration file, an exception is thrown, and the error message is displayed. A similar situation occurs if a fault occurs while generating data points (e.g., if the generator is not be able to create certain types of points according to the specified settings). The following shows the error resulting from an incorrect number of dimensions in the default value of the region (3 values instead of 2):

```
> java –jar ..\dist\DataGenSW.jar –config paw3-2d.conf
defaultRegion.radius: Invalid number of entries (3)
Exception in thread "main" java.lang.IllegalArgumentException:
      defaultRegion.radius: Invalid number of entries (3)
      at org. apache.commons.lang3.Validate.isTrue(Validate.java:155)
      ...
```

It is possible to change or add parameters directly in the command line (the values specified in the command lines will overwrite the existing ones). Theoretically, there is the possibility of defining all parameters via command line, without the need of creating a configuration file, although this is impractical. Nevertheless, this functionality makes it easy to create temporary modifications without having to create new configuration files:

```
> java –jar DataGenSW.jar –config <configuration file > –Dparam1 =
wart1 –Dparam2 = wart2 –Dparam3 = wart3 ...
```

In the example below, the total number of points and their distribution defined in the configuration file will be overwritten by the command line settings. **Note that, passing parameters in such a way, you must not insert spaces after the -D option, around the '=' character, or in the values themselves:**

```
> java -jar ..\dist\DataGenSW.jar -config paw3-2d.conf
-Dexamples=1800 -DclassRatio=1:2
```

By default, the information displayed while the generator is running is very limited. If there is a need for a thorough insight into the various stages of action, and in particular, the final settings and the designated absolute number of points in each class, region, etc., then you can enable accurate reporting by providing an additional switch:

```
-Dorg.slf4j.simpleLogger.defaultLogLevel=debug
```

which must be placed before the **-jar** parameter. Its results are as follows:

```
> java - Dorg. slf 4 j. simpleLogger. defaultLogLevel = debug - jar. . \ dist \ DataGenSW. jar
- config paw3 - 2d - 5pairs. conf - DlearnTestPairs = 1 - Dexamples = 1000 - DlearnTestRatio = 1: 1

[main] DEBUG GeneratorSettings - Final configuration:
attributes = 2
classes = 2
examples = 1000
names.attributes = A1, A2
names.classes = MIN, MAY
names. decision = DEC
learnTestRatio = 1.0:1.0
learnTestPairs = 1
fileName.learn = paw3-2d-learn-%d.arff
fileName.test = paw3- 2d-test-%d.arff
minOutlierDistance = 1.0
classRatio = 1.0:9.0
class.1.exampleTypeRatio = 40.0:20.0:30.0:10.0
class.1.regions = 3
class.1.region.1.weight = 1.0
class.1.region.1.shape = CIRCLE
class.1.region.1.distribution = UNIFORM
class.1.region.1.center = [5.0, 5.0]
class.1.region.1.radius = [2.0, 1.0]
class.1.region.1.borderZone = 1.0
class.1.region.1.noOutlierZone = 1.5
class.1.region.1.rotations = [(1, 2) -- > 45.0]
class.1.region.2.weight = 1.0
class.1.region.2.shape = CIRCLE
class.1.region.2.distribution = UNIFORM
class.1.region.2.center = [ - 5.0, 3.0]
class.1.region.2.radius = [2.0, 1.0]
class.1.region.2.borderZone = 1.0
class.1.region.2.noOutlierZone = 1.5
class.1.region.2.rotations = [(1, 2) -- > - 45.0]
class.1.region.3.weight = 1.0
class.1.region.3.shape = CIRCLE
class.1.region.3.distribution = UNIFORM
class.1.region.3.center = [0.0, - 5.0]
class.1.region.3.radius = [2.0, 1.0]
class.1.region.3.borderZone = 1.0
class.1.region.3.noOutlierZone = 1.5
class.1.region.3.rotations = []
class.2.exampleTypeRatio = 100.0: 0.0: 0.0: 0.0
class.2.regions = 1
class.2.region.1.weight = 1.0
class.2.region.1.shape = INTEGUMENTAL
class.2.region.1.distribution = UNIFORM
class.2.region.1.center = [0.0, 0.0]
class.2.region.1.radius = [10.0, 10.0]
```

```
class.2.region.1.borderZone = 1.0
class.2.region.1.noOutlierZone = 1.5
class.2.region.1.rotations = []
[main] DEBUG GeneratorSettings - Distributing all examples into learning and testing parts:
# 1000 (1.0:1.0) == > # 500.0:500.0
[main] DEBUG GeneratorSettings – Learning Part
[main] DEBUG GeneratorSettings – Distributing examples into classes:
# 500.0 (1.0:9.0) == > # 50.0:450.0 (initial attempt)
[main] DEBUG GeneratorSettings – Distributing class 1 into examples types:
# 50 (40.0: 20.0: 30.0: 10.0) == > # 19.0:10.0:16.0:5.0
[main] DEBUG GeneratorSettings – Distributing class 2 into examples types:
# 450 (100.0:0.0:0.0:0.0) == > # 450.0:0.0:0.0:0.0
[main] DEBUG GeneratorSettings - Distributing examples into classes:
# 500.0 (1.0:9.0) == > # 50.0:450.0 (after possible correction)
[main] DEBUG GeneratorSettings - Distributing class 1 into examples types:
# 50 (40.0:20.0:30.0:10.0) == > # 19.0:10.0:16.0:5.0
[main] DEBUG GeneratorSettings - Ratio of SAFE: BORDER examples: 0.6666:0.3333
[main] DEBUG GeneratorSettings - Distributing region 1 into examples types: # 9 == > 6.0:3.0
[main] DEBUG GeneratorSettings - Distributing region 2 into examples types: # 10 == > 6.0:4.0
[main] DEBUG GeneratorSettings - Distributing region 3 into examples types: # 10 == > 7.0:3.0
[main] DEBUG GeneratorSettings - Distributing class 2 into examples types:
# 450 (100.0:0.0:0.0:0.0) == > # 450.0:0.0:0.0:0.0
[main] DEBUG GeneratorSettings - Ratio of SAFE: BORDER examples: 1.0:0.0
[main] DEBUG GeneratorSettings - Distributing region 1 into examples types: # 450 == > 450.0:0.0
[main] DEBUG GeneratorSettings – Testing part
[main] DEBUG GeneratorSettings – Distributing examples into classes:
# 500.0 (1.0: 9.0) == > # 50.0: 450.0 (initial attempt)
[main] DEBUG GeneratorSettings – Distributing class 1 into examples types:
# 50 (40.0:20.0:30.0:10.0) == > # 19.0:10.0:16.0:5.0
[main] DEBUG GeneratorSettings – Distributing class 2 into examples types:
# 450 (100.0:0.0:0.0:0.0) == > # 450.0:0.0:0.0:0.0
[main] DEBUG GeneratorSettings – Distributing examples into classes:
# 500.0 (1.0:9.0) == > # 50.0:450.0 (after possible correction)
[main] DEBUG GeneratorSettings – Distributing class 1 into examples types:
# 50 (40.0:20.0:30.0:10.0) == > # 19.0:10.0:16.0:5.0
[main] DEBUG GeneratorSettings – Ratio of SAFE: BORDER examples: 0.6666:0.3333
[main] DEBUG GeneratorSettings – Distributing region 1 into examples types: # 9 == > 6.0:3.0
[main] DEBUG GeneratorSettings – Distributing region 2 into examples types: # 10 == > 6.0:4.0
[main] DEBUG GeneratorSettings – Distributing region 3 into examples types: # 10 == > 7.0:3.0
[main] DEBUG GeneratorSettings – Distributing class 2 into examples types:
# 450 (100.0: 0.0: 0.0: 0.0) == > # 450.0:0.0:0.0:0.0
[main] DEBUG GeneratorSettings – Ratio of SAFE: BORDER examples: 1.0:0.0
[main] DEBUG GeneratorSettings – Distributing region 1 into examples types: # 450 == > 450.0:0.0
[main] INFO Generator - Generating data set (s)
[main] INFO Generator - Pass 1. . .
[main] INFO Generator - Learning set
[main] INFO ARFFWriter - Saving file paw3 - 2d - learn - 1.arff ... [main] INFO Generator - Testing
set
[main] INFO ARFFWriter - Saving file paw3 - 2d - test - 1.arff ...
```

---

3. CONFIGURATION PARAMETERS

As mentioned in the previous section, the generator's operation is controlled by a set of parameters passed in the configuration file and/or in the command line (the parameters can be specified in both locations; the values from the command line override the values in the configuration file). Listing 1 presents a sample file, where the **paw3** dataset is created: a two-dimensional dataset, where points of the minority class have been assigned to labels indicating their type (safe, borderline, rare, and outlier).

Listing 1: A configuration file that allows you to generate one file.

```
# paw3-2d
attributes = 2
classes = 2
classRatio = 1:9
minOutlierDistance = 1
defaultRegion.weight = 1
defaultRegion.distribution = U
defaultRegion.borderZone = 1
```

```
defaultRegion.noOutlierZone = 1.5
defaultRegion.shape = C
defaultRegion.radius = 2, 1
defaultClass.exampleTypeRatio = 100:0:0:0
class.1.exampleTypeRatio = 40:20:30:10
class.1.regions = 3
class.1.region.1.center = 5,5
class.1.region.1.rotation = 1, 2, 45
class.1.region.2.center = -5,3
class.1.region.2.rotation = 1, 2, -45
class.1.region.3.center = 0,-5
class.2.regions = 1
class.2.region.1.shape = I
class.2.region.1.center = 0,0
class.2.region.1.radius = 10, 10
examples = 1500
fileName =paw3-2d.arff
exampleTypeLabels.classes = 1
```

The meaning of the individual parameters is as follows:

attributes	Number of attributes `attributes = 2`
classes	Number of classes `classes = 2`
names.classes	List of names of decision classes (comma-separated character strings) `names.classes = MIN, MAJ`
names.attributes	List of names of the attributes (comma-separated character strings) `names.attributes = A1, A2`
names.decision	Name of the decision attribute `names.decision = CLASS`
classRatio	Proportion of each class. They don't have to sum up to 1 or 100, the generator determines the proportion and converts it to the number of points in each class `classRatio = 1:9`
minOutlierDistance	Minimum distance between rare or outlier poins belonging to one class, expressed in absolute terms (euclidean distance). The distance is not checked for rare objects occurring in one "island" – they may appear very close to each other `minOutlierDistance = 1` **Note:** For certain configurations (large number of rare and outlier points and long distances between them) the generator may not be able to generate points. If, after a certain number of attempts (currently 10,000), the generation of points fails, the generator stops. To solve this problem, reduce the value of the `minOutlierDistance` parameter.

defaultRegion.*	Grouping of default settings that can be shared by several regions. They can be overwritten for special regions
	`defaultRegion.shape = C`
	`class.2.region.1.shape = I`

defaultRegion.weight	The weight of a region determines what proportion of points from a given class should be included in the region. The sum of weights does not have to sum to 1 or 100, the generator automatically computes the number of points in each region. `defaultRegion.weight = 1`
defaultRegion.shape	The shape of the region allows values C, R and I. C (circle) means meta-balls, R (rectangle) means meta-cubes and I (integumental) is a special case of a meta-cube, where the empty space is filled (usually with the majority class). Points from type I cannot appear in the core C and R regions. Also, type I regions cannot be rotated and cannot contain borderline, rare or outlier objects. In other words, type I regions can only be defined for classes that contain only safe points. `defaultRegion.shape = C`
defaultRegion.distribution	Distribution of points in a given region, allowing values U (uniform) and N (normal). If nothing is specified U is used. In the case of normal distribution (N), it is possible to indicate the number of standard deviations in the region by adding a value after the N (e.g., "N, 3"). Number of standard deviations is optional: if it is not explicitly stated, it is assumed to be 1.0. This setting only applies to safe object: borderline objects are generated using a uniform distribution (U). Also, regions of type I also ignore this setting (U distribution is used). `defaultRegion.distribution = U`
defaultRegion.center	The center of the region, given as a list of real numbers (must be equal to the number of attributes). `class.1.region.2.center = -5,3`

defaultRegion.border	Method to calculate region dimensions for areas containing safe and borderline points. Possible values are **fixed** and **auto**. The **fixed** value means that the generator uses the values of the **radius** and **borderZone** parameters (described below) given by the user and they are not modifiable in any way. On the other hand, the **auto** value means that the generator determines the dimensions of the areas for the safe and borderline points automatically, based on the order of point types (**exampleTypeRatio** parameter described below). The **radius** values are used as baseline values for the region sizes, and the method of their conversion is described in point 4.
defaultRegion.radius	The size of the region given as radii for the C regions and half of the side lengths for R and I regions (details shown in Figures 1 and 2). They are given as a list of real numbers, the length of the list must be equal to the number of attributes. If the **border** parameter is **fixed** (no conversion), this area is the "core" of the region - this is where safe objects are placed. If **border** = **auto**, the area contains safe and borderline points (see point 4); `class.2.region.1.radius = 10, 10`
defaultRegion.borderZone	Absolute width of the boundary of the area in which borderline points can be placed (must be a value > 0 if the borderline points are to appear in the class to which the area belongs). There is only one value for all attributes: this value is required if **border=fixed**, otherwise it is ignored.
defaultRegion.noOutlierZone	Absolute width of a zone in which rare and outlier points cannot appear. This zone is located around the zone with borderline points. The mutual alignment of the two zones is shown in Figures 1 and 2. `defaultRegion.noOutlierZone=1.5`
defaultRegion.rotation	The rotation of a region is defined by selecting two dimensions (unique indexes counted from 1) and an angle expressed in degrees. Multiple rotations can be defined for one region - either by one entry containing several triple digits, or by

	multiple entries. If the rotation is not defined, the regions are not converted. `class.1.region.1.rotation=1,2,45`
defaultClass.*	Grouping of default settings for all decision classes - a similar solution to `defaultRegion`, which aims to avoid multiple definitions for several classes with the same parameters.
defaultClass.exampleTypeRatio	Classify points in a class as four real numbers separated by a character ":". Successive numbers indicate points of type **safe**, **borderline**, **rare** and **outlier**. They do not have to sum to 1 or 100 - the generator determines the proportions and then sets the appropriate number of points. If the class is to contain type I regions, all objects belonging to it must be of type **safe**. If not specified, a 100:0:0:0 configuration (safe objects only) is used. `defaultClass.exampleTypeRatio = 100:0:0:0`
class.i.region.j.*	The definition of the jth region of the ith class overrides and replaces the default settings for the `defaultRegion`. `class.1.region.1.center = 5,5`
class.i.regions	Number of regions on the ith class. `class.1.regions = 3`
class.i.*	Properties of ith class that overrides and replaces the default settings entered for defaultClass. `class.2.region.1.shape = I`
examples	Total number of points in the entire dataset, i.e., in all decision classes and regions. `examples = 1500`
exampleTypeLabels.classes	Indication of decision classes for which labels indicating the type of data point are to appear in the generated dataset (labels are created by joining the class name with the name of the point type, e.g., MIN-BORDER). Classes for which different types of examples are to be generate should be indicated by specifying their indexes. If not specified, only the decision class labels will be written in the resulting dataset (without the types of individual points). To generate data for typical experiments, omit this setting. These labels are used for more detailed visualization and verification of

	generated collections of examples. `exampleTypeLabels.classes = 1`
fileName	The filename (in ARFF format) to which the generated data will be stored. Required if only one file is needed (the `learnTestPairs` parameter, described later in this text, is not provided). `fileName = paw3-2d.arff`

The generator also allows you to generate training/test pairs. This is not a simple cross-validation scheme: the generator considers the location of the rare and outlier objects in the training set and places the test points in similar areas. As a result, there is no need for manual modification of the files, as was the case with the first version of the generator. In Listing 2, we show that we can obtain 5 pairs of sets for the two-dimensional shape of the **paw3** dataset, also used in the previous example. Since the definition of individual areas has not changed, only the modified parts of the file are listed. This definition includes some new parameters, described below:

learnTestRatio	Distribution of points between training and test sets, expressed as a pair of numbers separated by a character ":". The individual values do not have to sum to 1 or 100 - the generator calculates the proportions and defined the appropriate number of points. If not given, only training sets are generated (i.e., a 100:0 ratio is defined). `learnTestRatio = 2:1`
learnTestPairs	Number of training/test pairs. If not specified, only the training set is generated. If the `learnTestRatio` parameter is given and the `learnTestPairs` is omitted, then the generator will report an error and stop running. `learnTestPairs = 5`
fileName.learn	Template for the name of the training sets. It should contain the suffix `-%d` which will be replaced by consecutive indexes from 1 to learnTestPairs. It is mandatory if the learnTestRatio is given, otherwise it can be omitted. `fileName.learn = paw3-2d-learn-%d.arff`
fileName.test	Template for the name of the training sets. It should contain the suffix `-%d` which will be replaced by consecutive indexes from 1 to learnTestPairs. It is mandatory if the

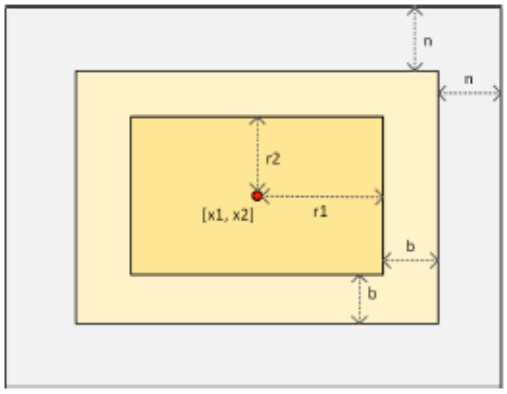
	learnTestRatio is given, otherwise it can be omitted. `fileName.test = paw3-2d-test-%d.arff`

Figures 4 and 5 show the training and test sets from the first generated pair. They show that the locations of rare and outlier points coincide, thus avoiding situations where test points from a class appear in a location where no training points exist. Of course, it is possible to make a traditional cross-validation scheme: you need to generate one file and then divide it into a learning and a test sets using an external tool (e.g., filter `StratifiedRemoveFolds` in WEKA).

Listing 2: A configuration file that allows you to generate 5 pairs of training-test files

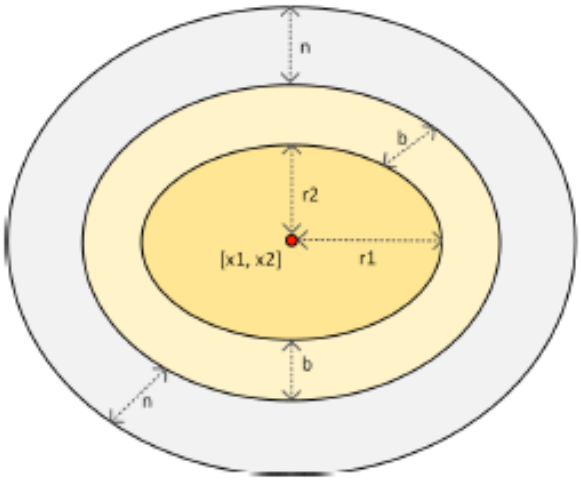
```
# paw3 -2d -5- pairs
# [...]
# Most settings are the same as for paw3-2d, the changes are below:

learn TestRatio = 2:1
learn TestPairs = 5
fileName.learn = paw3-2d-learn-%d.arff
fileName.test = paw3-2d-test-%d.arff
```




```
class.i.region.j.shape = R
class.i.region.j.center = x1, x2
class.i.region.j.radius = r1, r2
class.i.region.j.borderZone = b
class.i.region.j.noOutlierZone = n
```

Figure 1: Meta-cube region (R) and selected parameters describing its location and size (border = fixed)




```
class.i.region.j.shape = C
class.i.region.j.center = x1, x2
class.i.region.j.radius = r1, r2
class.i.region.j.borderZone = b
class.i.region.j.noOutlierZone = n
```

Figure 2: Meta-ball region (C) and selected parameters describing its location and size (border = fixed)

Finally, Listing 3 illustrates the configuration of a three-dimensional shape flower, shown in Figure 6 (using PlotViz3: `http://salsahpc.indiana.edu/pviz3/`). In order to increase the readability of minority class points, the labels indicating their type were generated (shown in color), whereas majority class points were temporarily disabled (shown in grey).

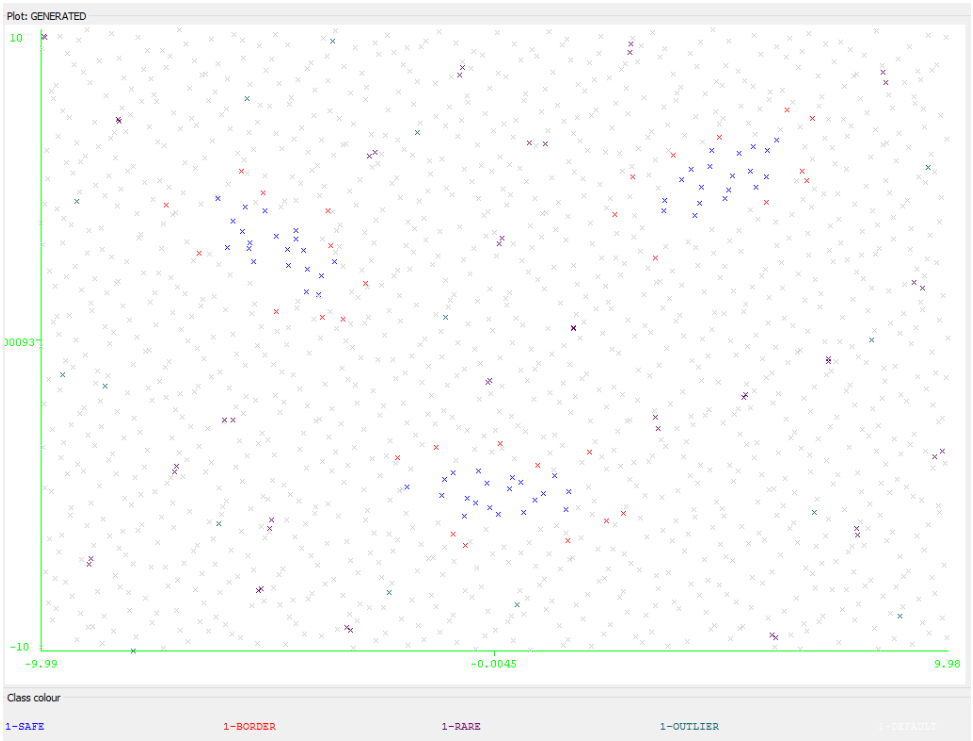


Figure 3: Visualization of the dataset obtained for the configuration from Listing 1. The different minority class types can be distinguished by the labels of each type. Minority class is marked as 1.

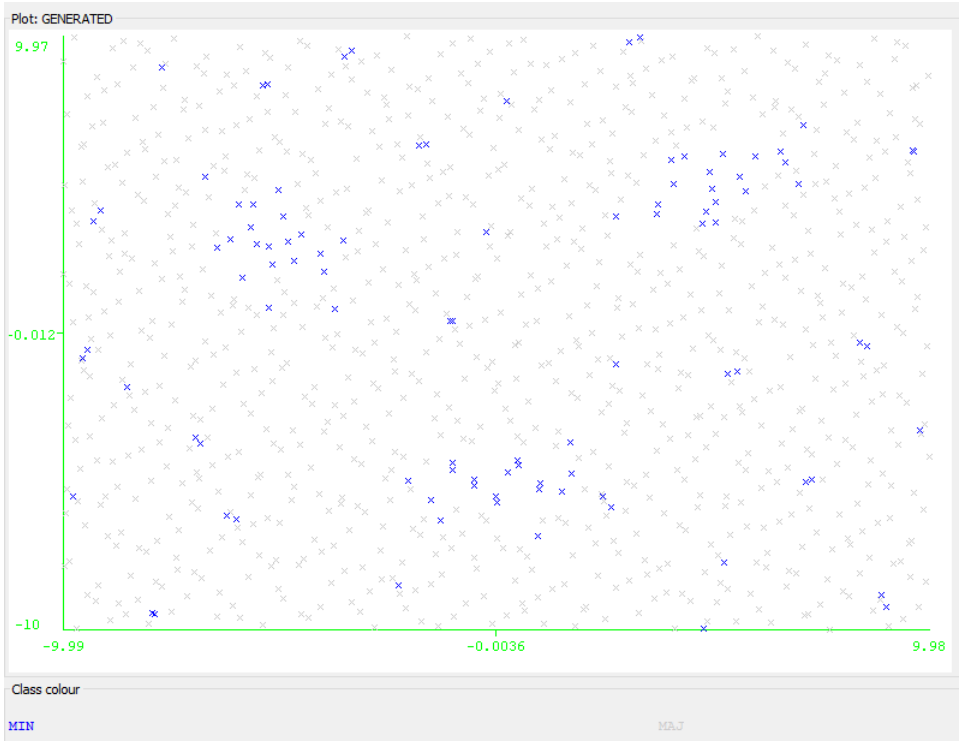


Figure 4: Visualization of the training set obtained from the configuration of Listing 2.

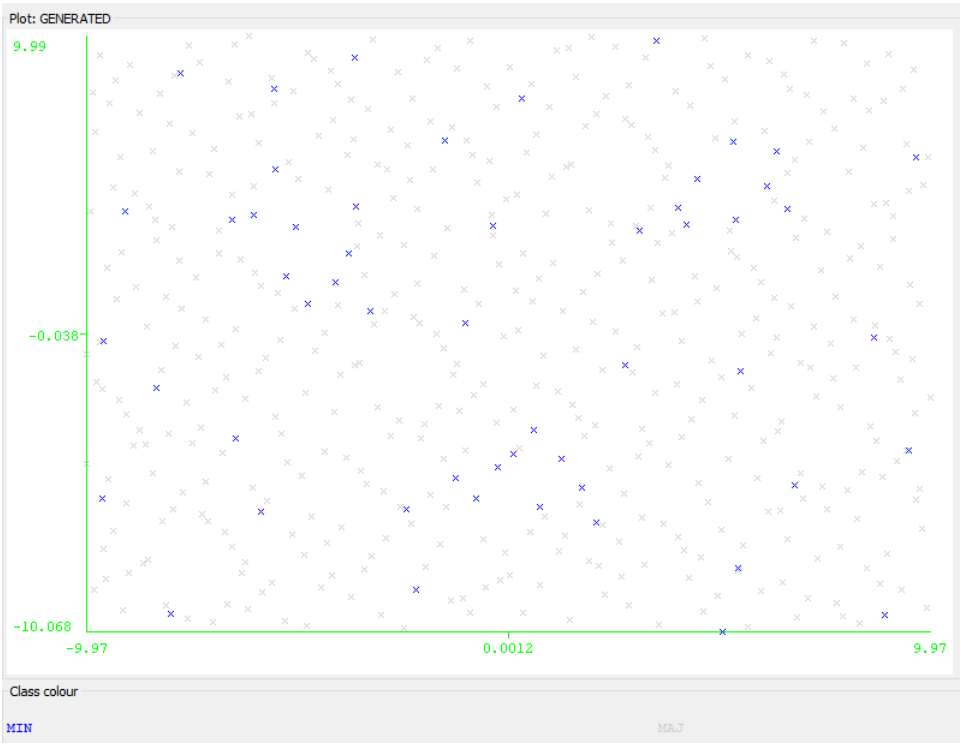


Figure 5: Visualization of the test set obtained from the configuration of Listing 2.

Listing 3: Configuration file for flower shape

```
# flower -3d
attributes = 3
classes = 2
names.classes = MIN , MAJ
names.attributes = A1 , A2 , A3
names.decision = CLASS
classRatio = 1:3
minOutlierDistance = 0.3
defaultRegion.weight = 1
defaultRegion.distribution = U
defaultRegion.borderZone = 0.5
defaultRegion.noOutlierZone = 0.5
defaultRegion.shape = C
defaultRegion.radius = 2, 1, 1
defaultClass.exampleTypeRatio = 100:0:0:0
class.1.exampleTypeRatio = 50:20:20:10
class.1.regions = 5
class.1.region.1.center = -3, 1.85 , 0
class.1.region.1.radius = 2, 1, 2
class.1.region.1.rotation = 1, 2, -45
class.1.region.2.center = 0, 2.8 , 0
class.1.region.2.radius = 1, 2, 2
class.1.region.2.distribution = N, 3
class.1.region.3.center = -1.5, -1.5, 0
class.1.region.3.radius = 1, 1, 2
class.1.region.3.distribution = N
class.1.region.4.center = 3, 1.85 , 0
class.1.region.4.radius = 2, 1, 2
class.1.region.4.rotation = 1, 2, 45
class.1.region.5.center = 0, 1.5 , 0
class.1.region.5.radius = 5.5 , 4.5 , 5
class.2.regions = 1
class.2.region.1.shape = I
class.2.region.1.center = 0, 1.5 , 0
class.2.region.1.radius = 5.5 , 4.5 , 5
examples = 10000
fileName =flower-3d.arff
exampleTypeLabels.classes = 1
```

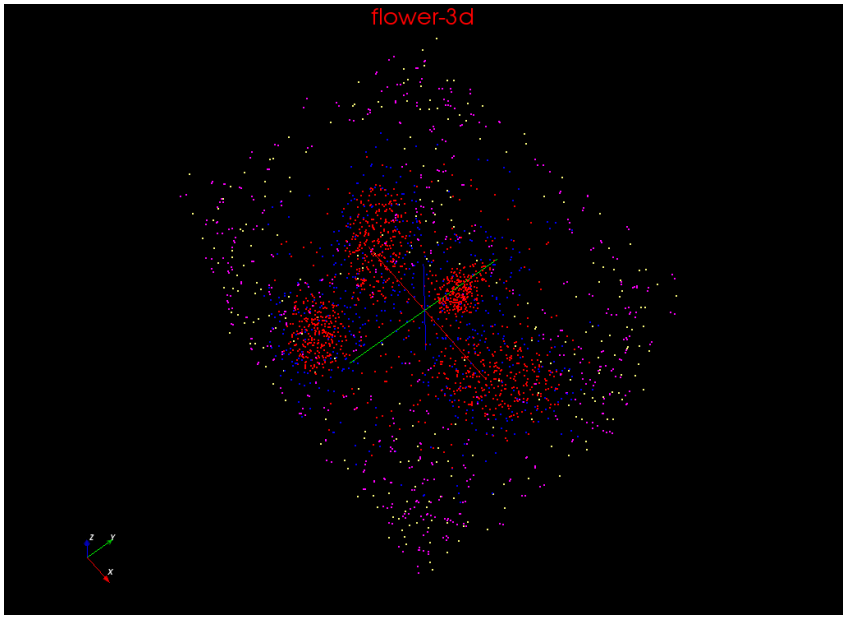


Figure 6: Visualization of the flower shape from Listing 3.

4. CONVERT AREA SIZES FOR SAFE AND BORDERLINE POINTS

Automatically recalculating the area sizes for safe and borderline points depending on the distribution of points in a class, allows them to maintain their density. This avoids a situation in which a significant reduction in the proportion of safe points will lead to a "thinning" of the area with these points (the same applies borderline points). Of course, this can be prevented by explicitly modifying the definition of selected regions, but such a solution can be difficult when you need to generate many different variants of the dataset.

The adopted method of converting sizes is based on the following assumptions:

- The ratio of the area between the safe + borderline points and the area of the whole region should correspond to the ratio between the number of both types in the region and the total number of objects in the region;
- The ratio between the area of safe points and the area of safe + borderline points should be equal to the ratio of the number of appropriate points in the region.

This leads to the following:

$$r_{safe+borderline} = r \cdot \sqrt[m]{\frac{n_{safe} + n_{borderline}}{n_{all}}},$$

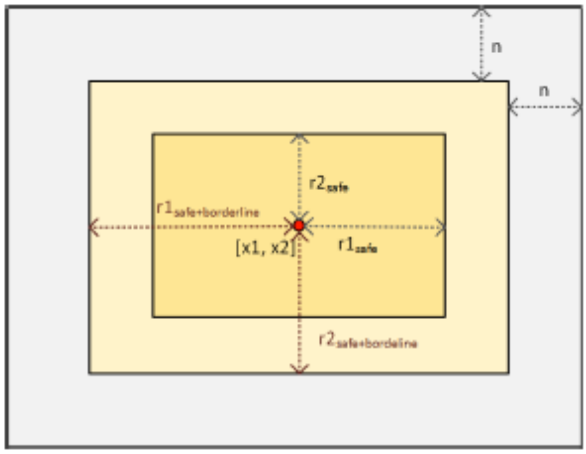
$$r_{safe} = r_{borderline} \cdot \sqrt[m]{\frac{n_{safe}}{n_{safe} + n_{borderline}}},$$

where:

- m = number of attributes (dimensions)
- rsafe + borderline = area size with safe and borderline objects

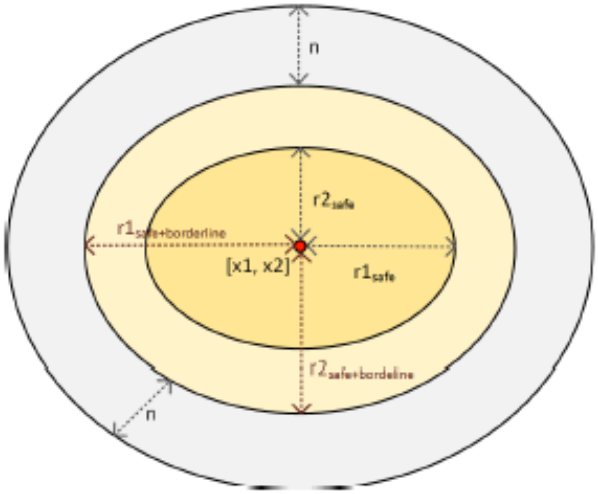
- rsafe = area size with safe objects
- r = size of the region (radius parameter)
- nall, nsafe and nborderline = the number of all objects, safe objects and borderline objects in the region, respectively

Figures 7 and 8 show a graphical interpretation of the parameters describing the location and characteristics of the region when the automatic radius conversion is established (borderZone parameter is ignored therefore it was crossed out.)




```
class.i.region.j.shape = R
class.i.region.j.center = x1, x2
class.i.region.j.radius = r1, r2
~~class.i.region.j.borderZone = b~~
class.i.region.j.noOutlierZone = n
```

Figure 7: A meta-cube (R) region and selected parameters describing its location and size (**border = auto**)




```
class.i.region.j.shape = C
class.i.region.j.center = x1, x2
class.i.region.j.radius = r1, r2
~~class.i.region.j.borderZone = b~~
class.i.region.j.noOutlierZone = n
```

Figure 7: A meta-ball (C) region and selected parameters describing its location and size (**border = auto**)

5. MAIN PACKAGES AND GENERATOR CLASSES

The generator was implemented in Java (JDK 1.8) and its source code was divided into 4 packages described below. Each package lists the most important classes that take direct part in the data generation process.

1. `pl.put.poznan.cs.idss.generator` - base classes wrapping the generator functionality
    a. `Generator` - main class (with the main method) - generates data according to the specified settings and saves them to files
    b. `ARFFWriter` - A class that supports writing data to ARFF format files
2. `pl.put.poznan.cs.idss.generator.factories` – factory classes responsible for creating specific types of points and data dsitributions (within regions)
    a. `RandomGeneratorFactory` - class that generates the appropriate point generators (`RandomGenerator` class and derivatives, described below) for different types of points and different types of distributions
    b. `RegionGeneratorFactory` - class that creates region-related generators (`RegionGenerator` and derived) for each region shape
    c. `DataSetGenratorFactory` - class that creates a new `DataSetGenerator` object based on descriptions of individual regions and descriptions of groups containing rare and outlier objects
    d. `AdditionalPointGeneratorFactory` - class that creates new `AdditionalOutlierPointGenerator` objects that are responsible for generating rare and outlier points
    e. `RegionGenerators` - class that generates and stores a list of generators for each region comprised in the dataset
3. `pl.put.poznan.cs.idss.generator.generation` - classes responsible for generating points of particular types and belonging to specific regions (most of them are created using classes from the `.factories` package)
    a. `RandomGenerator` – Generates points according to a certain distribution. This is the base class for `GaussianDistributionGenerator` and `LowDiscrepancySequenceGenerator`, which uses normal distributions and Halton sequences that simulate uniform distributions. There is also a `UniformDistributionGenerator` class that uses pseudo-random numbers, but this is not currently used
    b. `RegionGenerator` - Depending on the shape of the region, the generator creates one or two point generators - one for generating safe objects that are the core of the area and the other for generating borderline objects (for more information see the next section)
    c. `OutlierGenerator` - Generates rare and outlier objects (as opposed to region generator, only one generator is created for the dataset)
    d. `DataSetGenerator` - Generates an entire dataset. Uses `RegionGenerator` class and `OutlierGenerator` to get points belonging to individual regions and to "islands" (disjuncts)
4. `pl.put.poznan.cs.idss.generator.settings` - classes responsible for reading, storing and handling configuration settings for the generator

a. `GeneratorSettings` - class responsible for reading the settings from the configuration file and the command line
b. `ParameterExtractor` - a class that wraps the `GeneratorSettings` class, and allows existing classes from the factories and generation packages to read new settings

6. DIAGRAM OF GENERATOR'S OPERATIONS

Figure 9 shows the diagram of the generator's operations. It describes the various stages of the data generation process and points to the main classes that implement these stages. Currently, the first two stages are related to the processing of the settings and result from the changes made to the implementation - in the next version of the generator they can be merged. It also worth noting that, at present, the generator saves the objects to a file immediately after they have been generated and does not store them in memory. In situations where additional processing or long-term maintenance of memory objects would be required (e.g., overriding and moving to a data stream), it would be necessary to modify the Generator class (creating additional files - object collections for object storage).

7. SUMMARY

The new version of the generator presented in this report satisfies all the requirements set in the introduction and allows to obtain complex data sets. The generator will now be tested in a series of computational experiments investigating the impact of different types of "imperfections" (modeled by different distributions of different types of objects and the different regions that make up each class) on the performance of selected symbolic classifiers (decision trees, rules) and statistical classifiers (Naïve Bayes classifier, RBF neural networks).
As part of the further development of the generator, it is possible to add the following features:

- Definition of more complex regions by combining meta-balls and meta-cubes
- Support for symbolic (categorical) attributes
- Noise generation (attribute and class noise)
- Generation of missing data

8. REFERENCES

[1] K. Kaluzny. Metody dekompozycji w analizie niezrównowazonych liczebnie danych. praca magisterska, 2009. (Methods of decomposition in the analysis of unbalanced data. Master Thesis)
[2] K. Napierala and J. Stefanowski. Types of minority class examples and their influence on learning classifiers from imbalanced data. J. Intell. Inform. Syst., 2015 to appear.
[3] K. Napierala, J. Stefanowski, and Sz. Wilk. Learning from imbalanced data in presence of noisy and borderline examples. In Proceedings of the 7th International Conference RSCTC 2010, volume 6086 of LNAI, pages 158-167. Springer, 2010.

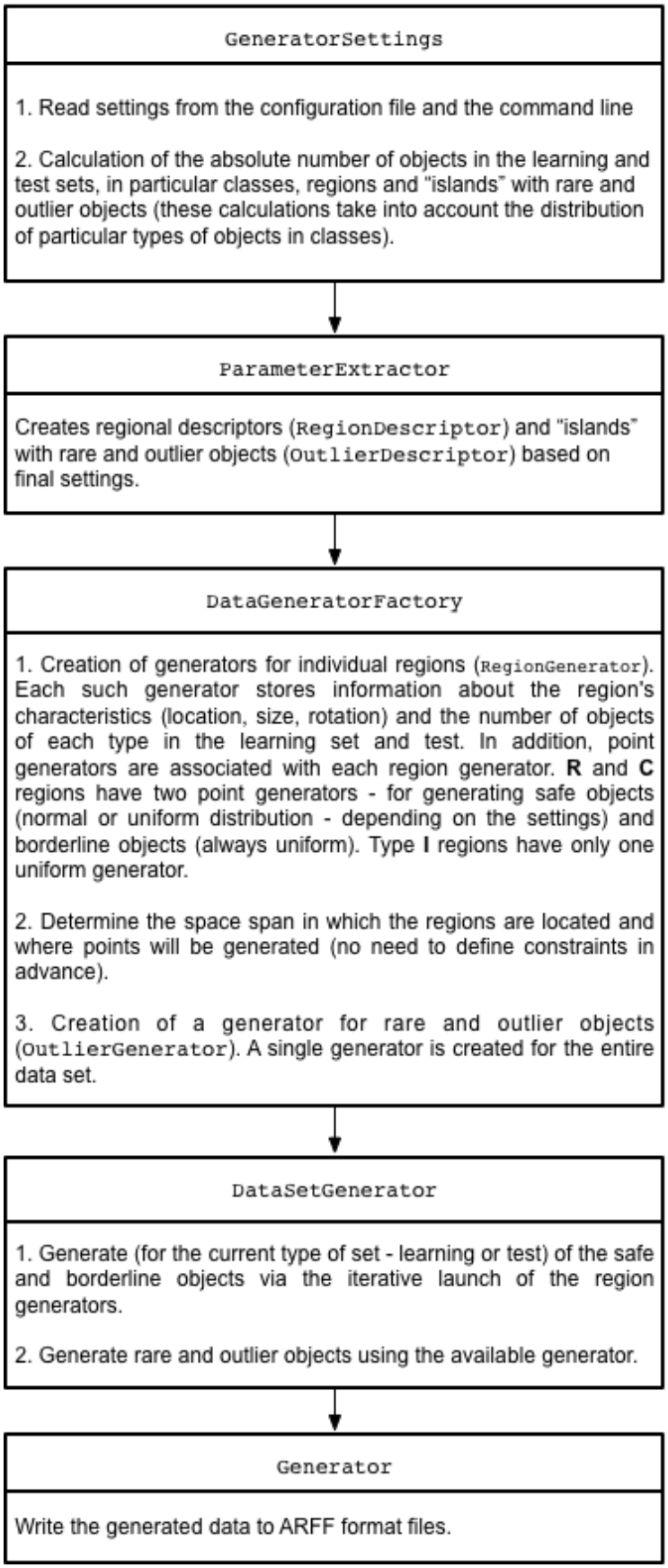


Figure 9: Diagram of the generator's operations